Profiling of OpenMP tasks with Score-P

Daniel Lorenz*, Peter Philippen*, Dirk Schmidl† and Felix Wolf‡*†
*Jülich Supercomputing Centre, 52425 Jülich, Germany
†RWTH Aachen University, 52056 Aachen, Germany
‡German Research School for Simulation Sciences, 52062 Aachen, Germany

Abstract*—With the task construct, the OpenMP 3.0 specification introduces an additional level of parallelism that challenges established schemes of performance profiling. First, a thread may execute a sequence of interleaved task fragments the profiling system must properly distinguish to enable correct performance analyses. Furthermore, the additional parallelization dimension requires new visualization methods for presenting analysis results. Finally, as a new programming paradigm, tasking implicitly introduces paradigm-specific performance issues and creates a need for corresponding optimization strategies. This paper presents solutions to overcome the challenges of profiling applications based on OpenMP tasks. Second, the paper describes metrics that may help uncover performance problems related to tasking. We present an implementation of our solution within the Score-P performance measurement system, which we evaluate using the *Barcelona OpenMP Task Suite.

I. Introduction

The efficient use of today's multi-core processors by multithreaded applications requires evenly balancing the load across the available cores, a non-trivial task—especially since the influencing parameters may vary with scale as well as system and network characteristics.

OpenMP is a programming interface for multithreaded shared-memory programs. It provides constructs for sharing work among threads, e.g., to assign the iterations of a loop to individual threads. In OpenMP 2.5, automated load-balancing possibilities were limited to loops with fixed bounds. To provide support in more dynamic cases, like recursions or if loop bounds are not known a priori, OpenMP 3.0 [1] added the task construct. It allows to specify independent work packages that can be executed in parallel, providing an automated dynamic load-balancing mechanism. However, balancing the load in this way introduces management overhead. Thus, for efficient use of the tasking paradigm, the appropriate granularity of tasks is essential. If they are too large, the load-balancing quality may suffer, if the tasks are too small, the task management overhead may become larger than the gain from any positive balancing effects.

Performance analysis tools such as HPCToolkit [2], ompP [3], Paraver [4], Scalasca [5], TAU [6], and Vampir [7], provide insight into the performance behavior of applications and have proven their usefulness for performance optimization. Call-path profiles are an established way of providing an overview of performance properties and are used by many tools [2], [3], [5], [6]. In the context of OpenMP programs, they can help detect idle times of threads and measure the amount of work each thread performs. When adapted for tasks, they can give the necessary information to determine and optimize the task throughput of an application. However, the difficulties implied by the additional parallelization dimension prevented the emergence of task analysis and profiling tools for tasking so far. A detailed overview on related work is given in Section II.

We provide the first profiling tool that provides call-path-level statistics about applications with OpenMP 3.0 tasks. However, our approach is limited to *tied tasks. Untied tasks* are not yet supported because the OpenMP runtime does not provide any standardized hooks to notify the measurement system of task switches. Section IV presents the detailed algorithm we developed—based on the results of the problem analysis in Section III.

We show that for reasonably-sized tasks the measurement overhead is limited (see Section V) and demonstrate the usefulness of the measured data with an example from the *Barcelona OpenMP Tasks Suite* [8] in Section VI. Finally, we present our conclusions in Section VII.

II. Related Work

Profiling is an established performance analysis technique for both sequential and parallel programs that collects performance metrics such as time and attributes them to static or dynamic program entities such as functions or call paths. This data acquisition method distinguishes two underlying approaches. The first approach is sampling, where metrics are derived from measurements taken in regular intervals, as done in HPCToolkit [2]. This tool can measure the amount of idle time and overhead of threads [9] if the run-time system provides information on the number of idle or working threads. However, the OpenMP runtime does not provide this information. Furthermore, it does not provide information that helps identify those tasks that may cause overhead or imbalance.

The second approach to obtain execution profiles is direct source-code instrumentation. Tools such as ompP [3], Scalasca [5], or TAU [6] insert calls to their measurement system into the application. They provide OpenMP support but without tasking. If single task instances can not be identified, task suspension and interruption violates the requirement for correct nesting of enter and exit events, which is necessary for call-path profiling based on direct instrumentation [10]. The first extension for tasking support was published by Fürlinger and Skinner [11]. However, their approach lacks task instance identification and, thus, supports only uninterrupted tasks.

For direct instrumentation of OpenMP constructs, many tools [3], [5], [6] utilize the source-to-source instrumenter OPARI [12]. However, the original version of OPARI was developed before the OpenMP board specified tasks, thus, tools relying on this version cannot support OpenMP tasks. For tied tasks, we introduced an extension to OPARI to track individual instances of tasks [10]. Its successor OPARI2 [13] includes this feature and provides automatic instrumentation of task constructs including support for task instance identification.

For tools using the sampling approach, Lin and Mazurov [14] proposed an interface to query the OpenMP runtime about tasks, extending an earlier interface from SUN [15], and provided an example implementation with the SUN Studio Analyzer. However, they focus on data acquisition, while our work builds upon existing instrumentation and focuses on the collection of profile data and its analysis. Furthermore, the analyses of sampling data and direct instrumentation data are very different.

Schmidl et al. [16] described possible performance problems with OpenMP tasks and visualized trace data of tasks with Vampir [7]. However, manually searching a time-line visualization for certain performance patterns is tedious and time consuming. Although it allows an in-depth analysis, a method to locate issues automatically on a full application scale is necessary.

Another trace-based time-line visualization and analysis tool is Paraver [4], developed at the Barcelona Supercomputing Center. It can capture and display task data of the OmpSs [17] programming model, which is also developed in Barcelona. In OmpSs, tasks are not restricted to parallel regions, but the whole program is decomposed into tasks. Dependencies between tasks are defined through data dependencies and implicitly specified by the input and output data of the tasks. Furthermore, OmpSs provides statements to wait for the completion of specific tasks. The manyfold ways of specifying task dependencies in OmpSs is likely to create performance issues different from those encountered with OpenMP tasks. In the latter case, dependencies between tasks are very limited and performance issues related to task size dominate.

III. Problem Analysis

Tasking provides an automatic work-scheduling scheme which helps overcome one of the fundamental performance concerns of multithreaded OpenMP applications: load imbalance. This comes at the cost of additional task management overhead. Schmidl et al. [16] identified three performance issues specifically related to OpenMP tasks:

- Very small tasks may cause high overhead.
- Very large tasks may reduce the load-balancing effect.
- On larger scales, the task creation may become a bottleneck if tasks are created only by a small number of threads.

The major strategy of optimizing performance for OpenMP tasks is to find the appropriate size for the tasks. It is difficult to specify general thresholds because they depend on various parameters. Thus, any performance analysis tool should provide the user with the necessary information to:

- Determine the appropriate limits for task runtime.
- Identify tasks that incur performance penalties.

To achieve this, the following measurements must be reliably taken by the performance analysis tool:

- **Runtime** of task instances. However, considering the number of task instances in an OpenMP program, the information must be statistically processed. Of particular interest are the mean, maximum, and minimum runtimes of the task instances.
- **Creation time** for a task instance.
- **Management overhead** for tasking. This includes task creation, but task suspension/resumption and task completion also contribute management time.
- **Waiting time** at task scheduling points.
- **Association of tasks** with a context to support identification of problematic task instances.

Considering the specific data that is necessary to provide the user with useful information, we propose call-path profiling based on direct instrumentation.

Besides the inefficiency situations already described, also task dependencies can cause idle times. For example, some threads may not be able to execute a task because all enqueued tasks depend on not yet completed tasks. However, in OpenMP 3.0 the options of specifying task dependencies are limited to taskwait statements, which cause a task to wait for the completion only of direct child tasks. Therefore, dependency-induced problems are more likely to occur in tasking systems with more possibilities to define task dependencies such as OmpSs [17].

IV. Profiling Design

Tasks introduce an additional and different level of parallelism. Thus, the question arises how to display tasks in a call-tree structure. This section describes our solution with the design of the task profiling system. It extends the profiling system of the Score-P measurement system [13], which is briefly introduced. A detailed discussion of considerations to be taken into account follows in Section IV-B. It is concluded in Section IV-C with the presentation of the actual profiling algorithm we developed.

A. The Score-P profiling system

The basis for our profiling approach is the Score-P measurement system [13]. It uses compiler instrumentation to detect function calls and implements the POMP2 interface, which specifies the calls the source-to-source instrumenter OPARI2 uses to instrument an application. The basic profiling algorithm in Score-P is derived from the Scalasca profiling system. A function in the measurement system is called for every specified event such as the enter or exit of a function or an OpenMP region. At runtime, this event stream is translated into a profile which is stored in a tree structure, the call tree (see Figure 1).

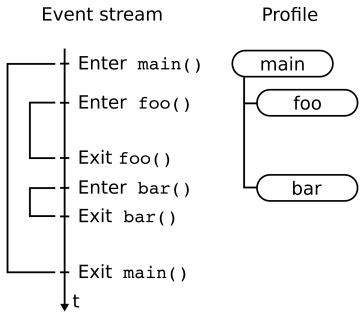

Fig. 1. An event stream and its translation into a profile. Note that the events are nested as needed for the profiling algorithm of Score-P. The functions `foo()` and `bar()` are entered and exited inside of `main` without overlap.

When the program is started, the first event is usually the enter event of the main function, for which the root node is created. For every enter event that follows the existing child nodes of the current node are checked whether a node for that code region already exists. If so, the metrics are attributed to that node, if not, a new node is created. When an exit event is encountered the pointer referencing the current node is moved to the parent of that node. Timestamps are taken which allow information on the inclusive runtime to be gained, i.e., the time that was used for that function including all the functions that were called from it.

Each node in the call tree thus refers to a source-code region. It stores the required data on certain metrics, e.g., the inclusive runtime and the number of visits, together with information required for statistical analysis, i.e. the sum, the minimum, the maximum and the number of samples. By subtracting the inclusive runtimes of all children the exclusive runtime is obtained, i.e., the time spent exclusively inside the function itself.

In multithreaded applications, every thread operates on a separate section of preallocated memory and constructs a separate call tree. This avoids overhead-prone locking.

B. Considerations on task sub-tree location

Event streams of OpenMP programs, before the introduction of the tasking feature, were—strictly speaking—indistinguishable from event streams of serial applications. To the measurement system Score-P an OpenMP region is no different from a function that is entered and left. Of course, there is one stream for each thread, but all in all, to the measurement system each single stream of events was comparable to that of a serial program or of an application parallelized with MPI running on different processes.

Tasks introduce a number of problems in this regard, as they can be very dynamic at runtime. The following issues must be addressed:

1) The nesting condition of enter/exit events may not be fulfilled.
2) Tasks may be executed under a node in the call tree different from that where they were created.

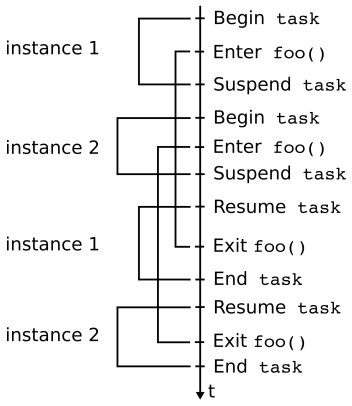

Fig. 2. Event stream with tasks. Both `task1` and `task2` are suspended inside of `foo()`. The enter and exit events of `foo()` inside of `task1` cannot be distinguished from the events in `task2`. This event stream cannot be translated into a profile with the standard profiling algorithm in Score-P.

3) When a task is suspended, another task starts or is resumed, without there necessarily being a direct parent-child relationship between the two.
4) After a task is suspended it may be resumed under a different node in the call tree.

Each of these issues is now discussed in detail. A deeper understanding of this new and different kind of parallelism and the implications for our traditional approach in Score-P leads to our new profiling and visualization scheme for OpenMP 3.0 tasks.

1) Nesting of events: We want to preserve the functionality the profiling system in Score-P provides as much as possible. Especially there should be no change for programs running without tasks. For the parallel region in programs employing tasks, there should also be no change in functionality. The execution of the explicit tasks however may lead to event streams which are not compatible with our profiling algorithm.

Fig. 1 shows how the enter and exit events of the functions `foo` and `bar` are nested inside of `main`. When tasks are used, the event stream may be as depicted in Fig. 2. Note that the nesting condition, as it is mandatory for the profiling algorithm described above, is not fulfilled anymore. The two tasks that are created, they both call function `foo`, and are then suspended. The exit events of `foo` that then follow cannot be distinguished from one another without storing information about the specific task instance that is active at the moment they are registered. This leads to the scheme introduced in [10] to keep track of specific task instances by modifying OPARI2 to insert instrumentation to store IDs of task instances inside the tasks' context itself.

2) Node assignment: In principle, there are two possibilities to assign a task to a node; it is possible to assign it to the node where it is created or to the node where it is actually executed (see Fig. 3). The former would present the call tree in the logical order of the program's source code. This would enable a visualization in a way a programmer would find intuitively familiar. On the other hand, the latter method would more

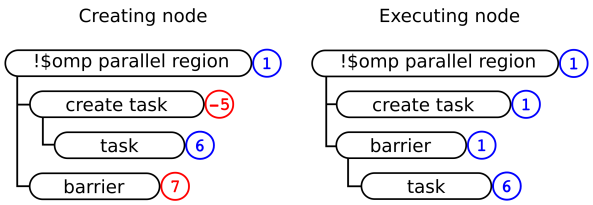

Fig. 3. Comparison of calculated exclusive times when assigning the task execution to the creating node vs. the executing node. In the former case negative values can occur, which does not make sense. Also the time attributed to the barrier is too large. The time spent executing the task is useful work and should not be attributed to the barrier. The exclusive execution time of a node is calculated by subtracting the inclusive execution time of its child nodes from its own inclusive execution time.

precisely reflect the runtime behavior of the program.

To assign the task to the node where it is created also leads to problems, as shown in Fig. 3. On the right hand side, the execution of the task is attributed to the barrier. The numbers shown stand for the exclusive execution times. The start of the parallel region, the task creation and the time spent waiting in the barrier are small. The actual work is done in the task. On the left hand side, however, the profiling scheme for calculating exclusive times leads to a task creation time of -5, which does not make sense. The waiting time in the barrier seems to be 7, which is too much, as most of that time was spent doing useful work. This example shows quite clearly that the attribution of metrics is only meaningful if the task is assigned to the node where it is actually executed.

3) Relationship between suspended and resumed task: If task instances are only created by implicit tasks and contain no scheduling points, their execution will automatically form one sub-tree which aggregates the execution of all task instances. Thus, it will provide the statistical information that is required. In contrast, Fig. 4 shows an example of an event stream where a task is suspended, in this case due to a `taskwait` statement. The task instance `task1` is suspended, `task2` is executed until it is suspended, too. Then, `task1` is resumed. As described above, we would create the node for `task2` in the profile as a child of the scheduling point where it was executed (the `taskwait` construct). However this leads to a number of problems:

- The call-tree structure would vary, depending on decisions of the runtime-system and the order in which the tasks are executed, reducing the reproducibility of performance results. Furthermore, it would become more difficult to compare results from multiple performance runs.
- The size of the profile may explode or the tree depth limits might kick in and most of the resulting tree would be pruned. This might especially be the case when considering recursive task creation.
- A fragmented call tree can not provide an easy overview of mean task execution time, minimum execution time or maximum execution time, which are some of the desired measurement results.

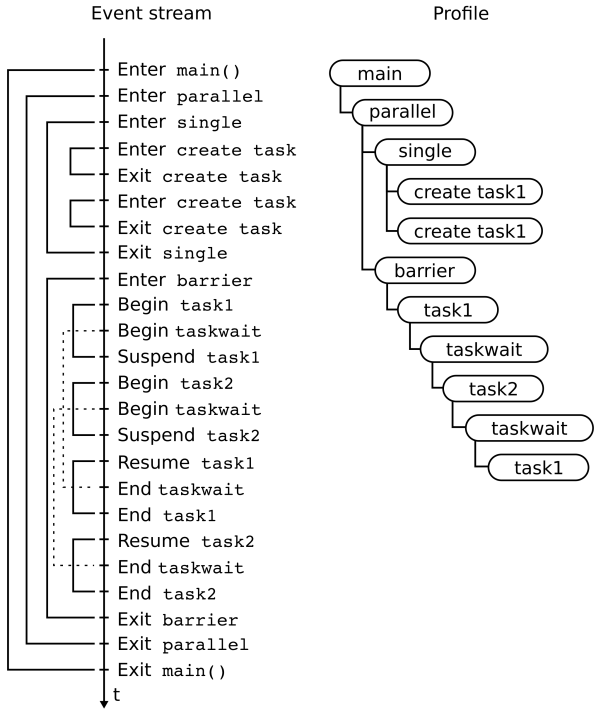

Fig. 4. Event stream where a task is suspended and another task is executed in the `taskwait` construct, which can be used as a scheduling point by the OpenMP runtime.

These issues are solved if the profile contains no parent/child relationship between explicit tasks. Then every tasks is recorded independently, although all task instances of the same task region will finally form a common sub-tree. However, for correct metric measurement, scheduling points of explicit tasks must measure exclusive values. This means, time measurements for a task must be stopped/resumed when the task is suspended/resumed.

4) Suspended task is resumed under another node: A task may be suspended and resumed at another scheduling point of the implicit task. In this case, as also shown in Fig. 4, the profile could simply copy the task's current call path to the new location. But how should we attribute not divisible metrics, like number of visits, or display meaningful maximum, minimum and mean of the execution time for the whole task? If we attribute them only to one fragment of the execution, the resulting picture is imbalanced, leading to misinterpretation of the data. To avoid this, the profile must present the task's sub-tree as a whole besides the implicit task's call tree.

However, we still want to follow the above argument (Section IV-B2) that the execution of a task appears as a child of the scheduling point, in which it was executed. As solution, a stub node is placed in the profile for each executed task region, as child of the scheduling point's node. The stub node contains the task's contribution to the measured implicit time at the scheduling point. Furthermore, it counts the number of

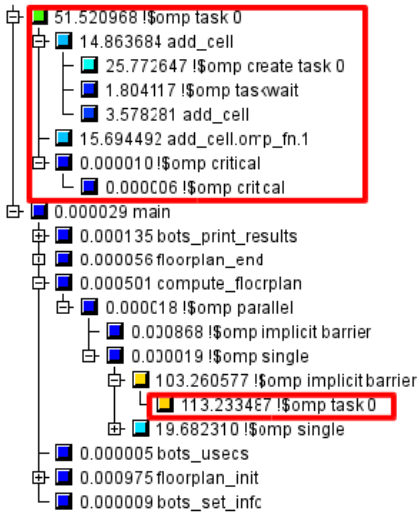

Fig. 5. Example for a call tree of an application with tasks. The lower red box marks the stub node for `task0`. It shows that 113s of task execution happened inside the barrier. 103s time is still spent inside the barrier not executing a task. This time may be overhead caused by task management and/or idle time. The upper red box marks the call tree of the task which gives insight into the task itself. E.g., we can see that the task region had 51.5s exclusive execution time and 25.8s were spent creating new tasks

times a task fragment was executed and also records other statistics for the executed fragments.

In addition, the task as a whole is presented above the main call tree. The task tree contains the statistics about the execution of the task as a whole and presents its inner structure. An example of a measurement with Score-P, visualized with CUBE, is shown in Figure 5.

C. Task profiling algorithm

For an application using tasks, it can not be guaranteed that all enter/exit events for a thread are correctly nested. The execution of multiple tasks may interleave, and thus, each task instance must be monitored as long as it has not been completed. It follows that we need to maintain the call-path information for each active task instance. In this context we refer to active tasks as tasks that started execution but have not yet completed, even though they may have been suspended. To keep track of active task instances we store the current position of every active instance and maintain a pointer to the position of the task that is currently being executed by a thread. As long as we have not created any tasks, the current task is the implicit task (see Figure 6).

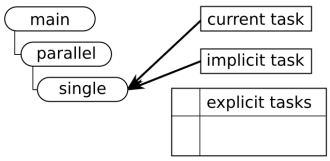

Fig. 6. The state of the profiling system before tasks are created. The table of explicit tasks is empty and the current task points to the position of the implicit task.

When the application creates a task, it enters and exits a task creation region. This will create a node in the call tree that contains information about the time spent for task creation. For other scheduling points, like taskwaits and barriers, the profiling algorithm issues an enter event for the region before the scheduling point and an exit event afterwards, measuring the time inside that region. Figure 7 continues the example of Figure 6.

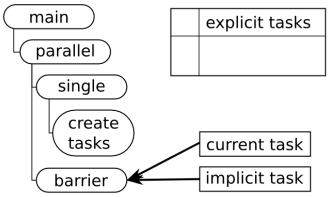

Fig. 7. The application has created tasks of the task region A, but not yet started execution. Afterwards, it has entered a barrier.

The implicit task can only proceed if all created tasks have been completed. Thus, the runtime will start the execution of a task (see Figure 8). As soon as the task starts execution, the profiling system creates an entry in the task instance table. For the new task instance, a separate call tree is created with the task region as root node. The *current task* pointer is now set to point to the position of the task that starts execution. Furthermore, we want to distinguish between waiting time and execution time of tasks inside the barrier of the implicit task. Thus, a second node for the task is created under the node of the barrier, with the *implicit task* pointer referencing it. This *stub node* contains the time the thread spent executing tasks inside this barrier.

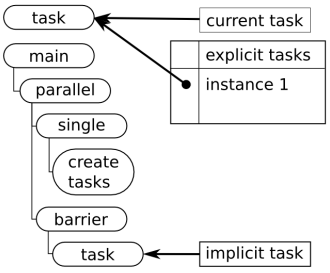

Fig. 8. Inside the barrier, the application has started execution of task A instance 1.

If a task contains a scheduling point, the measurement system creates an enter event for this scheduling point. If the thread switches to another task at this scheduling point, e.g., because it starts execution of another task instance, the profiling system creates the necessary data for this instance and updates the *current task* pointer. The implicit task leaves the node of the task region, when that task is suspended, and enters the task region of the resumed task. If both instances are created by the same task construct, it will be the same node. The situation after starting execution of a second instance is shown in Figure 9.

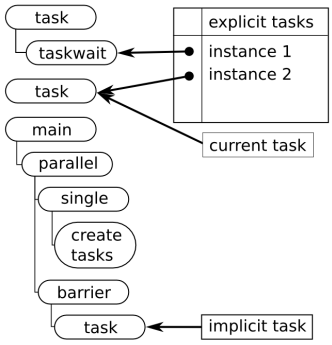

Fig. 9. Instance 1 entered a taskwait and was suspended, before a second instance of that tasking construct was started.

When a task completes execution, the following steps are carried out (Fig. 10 shows the results):

- The sub-tree is merged with the main profile tree, to appear as a separate tree besides the main tree. A new node is created for the first occurrence of this tasking construct. Later occurrences are merged with this node.
- The task instance's data structures are kept for later reuse.
- The *implicit task* exits the task region.
- If another explicit task is resumed or started, the *implicit task* enters the task region node of the resumed/started task.
- The *current task* pointer is updated to point to the position of the task that continues execution (either the *implicit task* or the next active task instance).

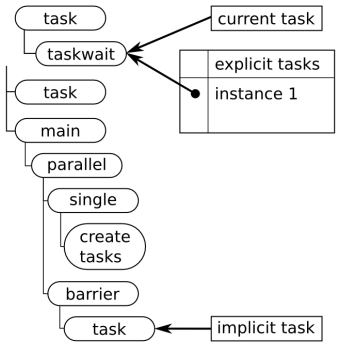

Fig. 10. Task instance 2 completed its execution without entering any other region. It was merged with the main profile before the thread continued execution of instance 1.

For explicit tasks, the interval between task suspension and resumption is subtracted from the time measurement and the measurement of other metrics. Thus, the task's tree contains only statistics about the execution of the task itself. Imbalance information is only required for each thread which is represented by the implicit task. Thus, only the implicit task's call tree contains task nodes as children of its scheduling points.

After all tasks are done, the profile contains the call tree of the implicit tasks and a call tree for each task construct which merges the statistics about the execution of all instances of this task construct. Figure 11 shows the profile after the completion of all tasks.

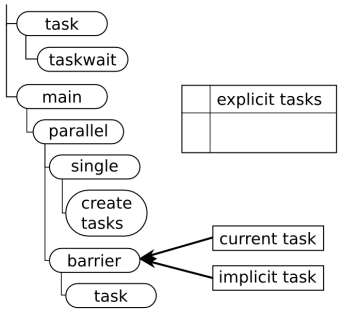

Fig. 11. Task instance 1 completed its execution.

Figure 12 compiles the the actions taken on task begin, task end and task switch events. On taskwaits, barriers and task creation only enter/exit events for the particular region are issued.

D. Untied tasks

Until now, the existing system supports only tied tasks. Untied tasks differ from tied tasks in two points: They may migrate to another thread and their execution can be interrupted at arbitrary points in the program's execution.

1) Migration: In principle the algorithm from Section IV-C also works for migrating tasks. The instrumentation allows to store pointers as identifiers. Only the executing thread needs to access the task-specific data and the task's call tree. Thus, if a task migrates, the pointer to the task-specific data migrates together with the task. Thus, if a task was started by a thread A and later task B resumes the task, it can safely access the task's call tree.

2) Interruption: Untied tasks may be interrupted anywhere during execution, preventing detection by source code instrumentation which encloses possible scheduling points. Because the task switch events are essential for our algorithm, we cannot support those tasks unless the runtime system provides support for these events. As a work-around, our instrumentation makes all tasks tied by default.

V. Overhead evaluation

The evaluation of the presented profiling mechanism of tasks has two major parts. In Section V-A, we evaluate the runtime measurement overhead. Therefore, we instrument and run the *Barcelona OpenMP Tasks Suite* [8] and compare the run times with those of the uninstrumented version. Section V-B evaluates the memory requirements for the intermediate task instance trees. The *Barcelona OpenMP Tasks Suite (BOTS)* [8] contains nine codes using OpenMP tasks. All experiments were done on the Juropa system [18] at the Jülich Supercomputing Centre, a Linux cluster with 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors on each node. The applications and Score-P were compiled with GCC version 4.6.2.

```
1   TaskBegin( task_region, task_instance )
2   {
3       Create task_instance specific data
4       TaskSwitch( task_instance )
5       Enter( task_instance, task_region )
6   }
7
8   TaskEnd( task_region, task_instance )
9   {
10      Exit( task_instance, task_region )
11      TaskSwitch( implicit task )
12
13      Merge task tree into global
14      profile of thread
15  }
16
17  TaskSwitch( task_instance )
18  {
19      if current task is an explicit task
20      {
21          Exit( implicit task,
22                root region of current task )
23
24          Stop time measurement on all open
25          regions of current task
26      }
27
28      Set current task to task_instance
29
30      if task_instance is an explicit task
31      {
32          Resume time measurement on all
33          open region of task instance
34
35          Enter( implicit task,
36                 root region of task_instance )
37      }
38  }
```

Fig. 12. Pseudocode for the task profiling algorithm, specifying the action for the task events. Enter/exit events and TaskBegin/TaskEnd events are called with the handle of the entered/exited source code region and the identifier of the executing task instance as parameters. TaskSwitch events are called with the identifier of the resumed task instance as parameter.

A. Overhead of the profiling system

To determine the overhead of the profiling system, we instrumented all nine *BOTS* applications with OPARI2 [13], the default compiler instrumentation of Score-P was disabled. The *BOTS* benchmark provides multiple versions of the applications. Because the current system can not support untied tasks, we evaluated only the versions with tied tasks. If a version with a cut-off for recursive task depth was provided (*fib, floorplan, health, nqueens, strassen*) we chose the cut-off version. For *sparselu* the version that creates tasks in a single construct was used.

The instrumented and uninstrumented version of the code ran on the Juropa system with 1, 2, 4, and 8 threads and medium input size. The benchmark gives the runtimes of its parallel region, containing the tasking kernel, as output. This was used to determine the measurement overhead of the instrumented applications. The results are shown in Figure 13.

For *alignment*, *sparselu* and *strassen*, the measurement

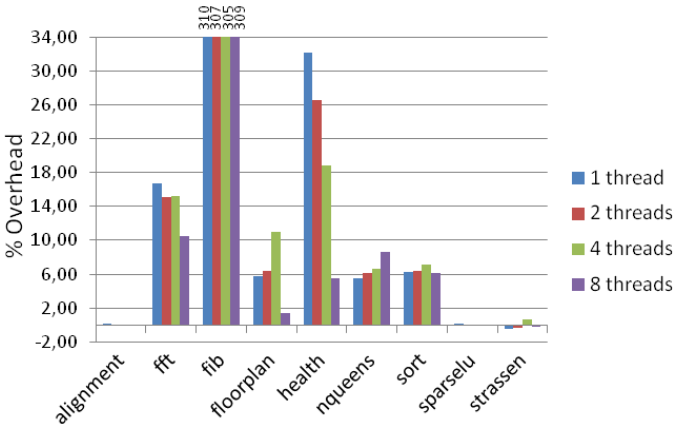

Fig. 13. The runtime overhead of the task profiling system with OPARI2 instrumentation (no compiler instrumentation), showing the runtime overhead of the computing kernel in percent compared to the uninstrumented version of the *BOTS* benchmark codes. This figure shows the measurements with the optimized (i.e., cut-off) version if available. Missing bars mean zero percent overhead. Negative values are due to measurement inaccuracy.

results show no measurable overhead. For *nqueens* and *sort*, the measured overhead is around 6 percent. For *floorplan*, the instrumented measurements with 2 and 4 threads can be divided into two classes. With 2 threads, class A contained 9 out of 50 measurements which took between 9.6s and 9.9s while class B contained the other 41 measurements which took between 19.4s and 19.9s. With 4 threads, class A contained 15 measurements which took between 4.5s and 5.2s while the other 35 measurements which belong to class B took between 7.8s and 8.0s. The uninstrumented measurements showed only results which are close to results of class B. Inspection of the profiles showed, that in case of class A, the execution time of the tasks was evenly distributed between the threads. The measurements of class B show that half of the threads executed no tasks, but idled the whole time. Thus, our conclusion is that, in the uninstrumented case all measurements behaved like class B. The overhead calculation for *floorplan* with 2 and 4 threads considered only the results of class B. This correction results in an overhead of approx. 6% for 1 and 2 threads, 11% for 4 threads and 2% for 8 threads. The measurements for class A would lead to an overhead of -47% for 2 threads.

On the other hand, the *fib* code with 310 % overhead, is an artificial pathological example. It basically creates 2 child tasks, waits for them and then only sums up two numbers. This is done recursively by each child task until a specified level of recursion is reached. However, the taskwait statements still create an enter and exit event. Because the computation time for one addition is small, these two events create large relative overhead.

Furthermore, *fft* and *health* show larger overhead, too. *fft* starts with 16.7% overhead for one thread which decreases to 10.5% for 8 threads. For *health* the decrease is even stronger, starting with 32.1% with one threads and ends with 5.6% with 8 threads.

Except *fib*, the application versions selected in Figure 13

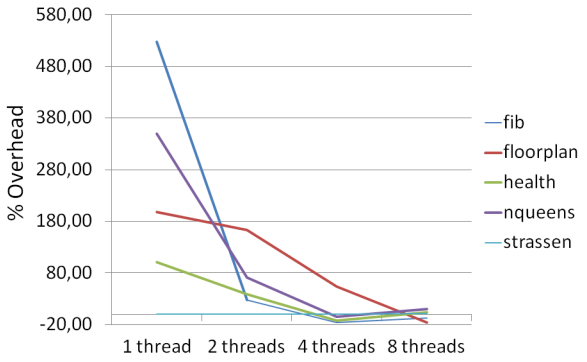

Fig. 14. The runtime overhead of the task profiling system with OPARI2 instrumentation (no compiler instrumentation), showing the runtime overhead of the computational kernel in percent compared to the uninstrumented version. In contrast to Figure 13, this figure shows the unoptimized versions of the codes.

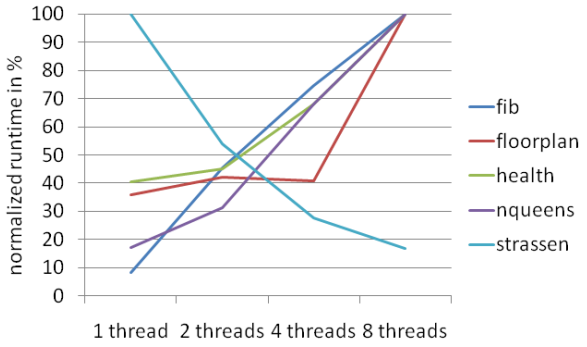

Fig. 15. The runtime of the uninstrumented *BOTS* benchmark codes version without cut-off, for those codes for which a version with cut-off is provided. The runtime is shown in percent compared to the highest measured value for that code.

represent real world applications where tasks perform a reasonable amount of work. However, as a stress test of the profiling system, we also ran the *BOTS* version without the cut-off (see Figure 14), which creates a large amount of small tasks. With increasing number of threads, the overhead decreases significantly from large values (e.g. 527% for *fib*) to values near or even below zero percent. The only exception is *strassen*, which always has a low overhead.

When looking at the runtimes of the codes in Figure 15, we can see that the overall runtime increases. The only exception is the *strassen* code. Table I shows the number of tasks and the mean execution time of tasks for each application. Obviously, the mean execution time of a task in *strassen* is approximately two magnitudes larger than the mean execution time in *fib*, *health*, and *nqueens* and still more than fifteen times larger than the tasks in *floorplan*. On the other hand, the number of tasks in *stassen* is significantly smaller. Our conclusion is that the mean execution time of 149 μs in *strassen* is reasonable, while the tasks in the other codes are too small.

Inspection of the profile of the other codes shows that the total time spent outside the OpenMP runtime does not increase, but stays constant. The additional time is spent inside

TABLE I
MEAN EXECUTION TIME OVER ALL TASKS AND NUMBER OF TASKS FOR CODE VERSIONS WITHOUT CUT-OFF.

code	mean time	number of tasks
fib	1.49 μs	3,690,000,000
floorplan	8.57 μs	73,700,000
health	2.35 μs	17,500,000
nqueens	1.24 μs	378,000,000
strassen	149.0 μs	960,800

OpenMP synchronization points or during task creation.

The reason is that the task management inside the OpenMP runtime system becomes a bottleneck, presumably due to necessary locking during access to internal data structures. Instrumentation shifts some of the overhead from the OpenMP runtime system to the profiling system, and thus, is shadowed when comparing the runtimes of the uninstrumented version with the instrumented version. Furthermore, it might reduce conflicts inside the OpenMP runtime system by increasing the runtime of the tasks.

This effect may also explain the overhead decrease for the *health* and *fft* code with cut-off.

A conclusion of these measurements is that for codes that are optimized and, thus, have small task management overhead, the additional runtime overhead for task profiling is limited, too. If the tasks are small, the measurement overhead is shadowed by the runtime overhead if multiple threads are used. However, massive use of taskwait statements or use of small tasks on one thread may result in huge overhead. In these cases, the the overhead increases the visibility of the performance problem.

B. Memory requirements

The profiling system maintains a separate call tree for every executing task instance. This task instance tree is created when the task instance starts execution (not when it is created). The task instance tree is merged into the main profile and the memory is released when the task instance completes execution. Considering the amount of tasks that may be created, the memory requirements need to be considered.

The memory requirements for the task instance trees depend on the complexity of a task instance tree and the number of concurrently executed tasks for which the profiling system must maintain a task tree. While the complexity of a task tree can be seen in the call-path profile of the tasks, the number of concurrently executing tasks is not obvious. Thus, we maintain a counter for the current number of task trees per thread and store the counter's maximum value for each parallel region. The results for the *BOTS* codes are given in Table II

The measurement results show that the number of concurrently executed tasks is never larger then 20. In 8 of the 14 cases the maximum number of tasks is even less than 5. In recursive algorithms, the maximum number of concurrent tasks reflects the recursion depth.

High numbers of tasks do not only cause management overhead in the profile, but the runtime system also needs to maintain data about all created tasks. Thus, keeping the

TABLE II
MAXIMUM NUMBER OF CONCURRENTLY EXECUTING TASKS PER THREAD. THE VERSIONS WITH CUT-OFFS ARE MARKED AS SUCH.

code	max tasks
alignment	1
fft	19
fib (cut-off)	4
floorplan	20
floorplan (cut-off)	5
health	4
health (cut-off)	3
nqueens	14
nqueens (cut-off)	3
sort	18
sparselu	2
strassen	8
strassen (cut-off)	3

number of created tasks below a probably system-dependent threshold also limits the memory requirements for the runtime system. Therefore, the longest dependency chain (e.g. the recursion depth) of an application may serve as a good estimate for the number of concurrent tasks.

Because released task-instance tree nodes are reused and the number of concurrent tasks was low in all test cases, the memory overhead seems to pose no general problem. It could become a limitation though if dependency chains become extremely long.

VI. ANALYSIS EXAMPLES

Since the *Barcelona OpenMP Tasks Suite* [8] provides multiple versions of each code, which, in some cases, show significant performance differences, they provide examples where we can demonstrate the usage of the task profiling system. Like the overhead evaluation in Section V, all tests ran on the Juropa system [18].

As an example, we chose the *nqueens* code. It calculates all possibilities to place n queens on a chess board with n^2 fields. It is a recursive algorithm that places a queen successively on each field of the current line. If the placement does not conflict with earlier placements it creates a new task to process the next line. The benchmark suite provides a version which cuts the creation of new tasks at a certain level of recursion and a version which continuously creates new tasks.

If we look a the runtime of the uninstrumented *nqueens* version without cut-off for different number of threads, we see that the runtime increases with increasing number of threads (see Figure 15), indicating a performance problem.

To get a first impression, we look at the profile of an instrumented run with 4 threads. It shows that three quarters of the time inside the tasks is spent creating child tasks. This indicates that too many tasks are created that are too small. The mean exclusive execution time of a task was only 0.30 μs, while the mean time to create a task was 0.86 μs.

Furthermore, comparison of profiles of instrumented runs with different numbers of threads shows that the sum of the exclusive execution times of the task region had only little variation. However, the measured execution time attributed to task creation, taskwait and implicit barriers, increase significantly (see Table III). Because the management time for task completion and task switches is attributed to these regions, we conclude that the increase in runtime is due to management overhead of the runtime system. The mean time for a management action increases with increasing number of threads. If the runtime system needs exclusive access to internal data structures, many small tasks causing many management actions on many concurrent threads increase the probability of synchronization overhead. This leads to a higher mean time for management actions. The solution is to create less but lager tasks.

TABLE III
EXCLUSIVE EXECUTION TIMES OF AN INSTRUMENTED *nqueens* RUN FOR THE TASKWAIT AND TASK CREATE REGIONS INSIDE THE *nqueens* TASK CONSTRUCT FOR DIFFERENT NUMBERS OF THREADS. FURTHERMORE THE EXCLUSIVE TIME OF THE BARRIER IN THE MAIN TREE IS SHOWN.

	1 thread	2 threads	4 threads	8 threads
task	106.0 s	112.6 s	114.3 s	106.65 s
taskwait	2.44 s	6.69 s	24.83 s	101.7 s
create task	56.0 s	95.9 s	323.8 s	1102.3 s
barrier	0 s	40.1 s	183.0 s	947.7 s

TABLE IV
SUM AND MEAN OF THE INCLUSIVE EXECUTION TIMES OF A TASK IN *nqueens* DEPENDING ON THE LEVEL OF RECURSION. ADDITIONALLY THE NUMBER OF TASKS AT EACH LEVEL IS SHOWN.

depth level	mean time	sum	number of tasks
0	25.5 μs	0.00036 s	14
1	19.8 μs	0.0039 s	196
2	15.8 μs	0.0344 s	2,184
3	12.3 μs	0.2340 s	19,096
4	9.42 μs	1.270 s	134,848
5	7.06 μs	5.347 s	756,952
6	5.01 μs	16.94 s	3,380,776
7	3.46 μs	40.45 s	11,690,784
8	2.35 μs	72.76 s	30,966,152
9	1.59 μs	97.00 s	61,487,832
10	1.12 μs	99.29 s	88,522,448
11	0.82 μs	74.71 s	90,606,208
12	0.61 μs	38.40 s	63,166,908
13	0.33 μs	8.889 s	27,176,000

The application uses a recursive approach to create tasks. In order to evaluate the link between runtime and recursion level, we inserted parameter instrumentation into the task. This results in a profile with separate sub-trees for the tasks of each recursion level. The result is shown in Table IV. It shows that the average runtime of tasks decreases with increasing depth. Furthermore, most of the time was spent in tasks of depths 9 to 13. While tasks in depth level 0 to 3 contributed only an insignificant fraction to the runtime, but provided tasks with a reasonable runtime. 2000 tasks should be enough to fill and balance up to 8 threads. Thus, stopping task creation at level 3, as done by the *nqueens* version with cut-off, reduces the runtime of the uninstrumented computing kernel from 187 s to 11.5 s with 4 threads, providing a speedup of 16.

VII. CONCLUSION

Finding the appropriate task granularity is essential for the performance optimization of task-based applications in

OpenMP 3.0. Examples show that the runtime overhead due to ill-sized tasks may lead to an increase in runtime when moving to larger scales.

We presented a first algorithm that correctly creates task profiles by measuring task execution via direct instrumentation. Due to the runtime system overhead, the relative measurement overhead of the profiling system is limited. The profile statistics of the task and the main tree provides the required information to pinpoint inefficient task usage, for example, by comparing task creation time, which may grow with the number of threads, to the execution time of tasks.

Also the time spent inside the runtime system at synchronization points is measured. However, it is not yet possible to distinguish if this time is required for management, or if it is waiting time on the completion of some tasks. Combined with the number of task switches and the task creation and execution time, it might be possible to derive an estimation. However, this would require more research.

Automated trace analysis, like Scalasca [5] does for other programming paradigms, might provide some additional information, and/or highlight particular performance problems. For example, the time between the enter of the last synchronization point and the task switch event would be of interest. In this way it would be possible to calculate the ratio of overall management time to exclusive execution time for tasks.

Furthermore, the profile does not contain any analysis or information on the effects of task dependencies. Trace-based analysis could provide insight into dependencies among tasks, which may on the one hand provide hints to distinguish between management and waiting time, and on the other hand it may help in optimizing dependency chains that cause load imbalance.

VIII. Acknowledgment

This work is based upon work supported by the US Department of Energy under Award Number DE-SC0001621.

References

[1] OpenMP Architecture Review Board, "OpenMP Application Progam Interface Version 3.0," OpenMP Architecture Review Board, Tech. Rep., May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[2] 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," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 685–701, April 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

[3] K. Fürlinger and M. Gerndt, "ompP: A Profiling Tool for OpenMP," in *1st Int. Workshop of OpenMP (IWOMP)*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 15–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1892830.1892833

[4] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to Visualize and Analyze Parallel Code," in *WoTUG-18: Transputer and occam Developments*, Mar. 1995, pp. 17–31.

[5] M. Geimer, F. Wolf, B. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[6] S. S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[7] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel, "The Vampir Performance Analysis Tool Set," in *Tools for High Performance Computing*. Springer, Jul. 2008, pp. 139–155.

[8] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. 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)*, IEEE Computer Society. Vienna, Austria: IEEE Computer Society, Sep. 2009, pp. 124–131.

[9] N. R. Tallent and J. M. Mellor-Crummey, "Effective Performance Measurement and Analysis of Multithreaded Applications," in *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM, 2009, pp. 229–240.

[10] D. Lorenz, B. Mohr, C. Rössel, D. Schmidl, and F. Wolf, "How to Reconcile Event-Based Performance Analysis with Tasking in OpenMP," in *6th Int. Workshop of OpenMP (IWOMP)*, ser. LNCS. Springer Berlin / Heidelberg, 2010, vol. 6132, pp. 109–121.

[11] K. Führlinger and D. Skinner, "Performance Profiling for OpenMP Tasks," in *5th Int. Workshop of OpenMP (IWOMP)*, ser. LNCS, vol. 5568. Springer, May 2009, pp. 132–139.

[12] B. Mohr, A. Malony, S. Shende, and F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, August 2002.

[13] 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 *Competence in High Performance Computing 2010 (CiHPC)*, Gauß-Allianz. Springer, 2012, pp. 85–97. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24025-6_8

[14] Y. Lin and O. Mazurov, "Providing Observability for OpenMP 3.0 Applications," in *5th Int. Workshop of OpenMP (IWOMP)*, ser. LNCS, vol. 5568. Springer, May 2009, pp. 104–117.

[15] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin, "An OpenMP Runtime API for Profiling," Sun Microsystems, Inc., Tech. Rep., 2007.

[16] D. Schmidl, P. Philippen, D. Lorenz, C. Rössel, M. Geimer, D. an Mey, B. Mohr, and F. Wolf, "Performance Analysis Techniques for Task-based OpenMP Applications," in *8th Int. Workshop of OpenMP (IWOMP)*, ser. LNCS, vol. 7312. Berlin / Heidelberg: Springer, Jun. 2012, pp. 196–209.

[17] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, "Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL," in *Languages and Compilers for Parallel Computing*, ser. LNCS, 2011, vol. 6548, pp. 215–229. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19595-2_15

[18] "JuRoPA," 2012. [Online]. Available: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/JUROPA_node.html