How to reconcile event-based performance analysis with tasking in OpenMP*

Daniel Lorenz[1], Bernd Mohr[1], Christian Rössel[1], Dirk Schmidl[2], and Felix Wolf[1,2,3]

[1] Forschungszentrum Jülich, Jülich Supercomputing Centre, Germany
[2] RWTH Aachen University, Dept. of Computer Science, Germany
[3] German Research School for Simulation Sciences, Aachen, Germany

Abstract. With version 3.0, the OpenMP specification introduced a task construct and with it an additional dimension of concurrency. While offering a convenient means to express task parallelism, the new construct presents a serious challenge to event-based performance analysis. Since tasking may disrupt the classic sequence of region entry and exit events, essential analysis procedures such as reconstructing dynamic call paths or correctly attributing performance metrics to individual task region instances may become impossible. To overcome this limitation, we describe a portable method to distinguish individual task instances and to track their suspension and resumption with event-based instrumentation. Implemented as an extension of the OPARI source-code instrumenter, our portable solution supports C/C++ programs with tied tasks and with untied tasks that are suspended only at implied scheduling points, while introducing only negligible measurement overhead. Finally, we discuss possible extensions of the OpenMP specification to provide general support for task identifiers with untied tasks.

1 Introduction

In parallel computing, a task denotes an independent unit of work that contributes to the solution of a larger problem. A task is usually specified as a sequence of instructions to process a given subproblem. Tasks can be assigned to different threads and can be executed concurrently with other tasks, as long as input and output dependencies between tasks are observed. To offer a more convenient way of expressing task parallelism, version 3.0 of the OpenMP specification [1] introduced a task construct along with synchronization mechanisms and task scheduling rules.

The OpenMP specification distinguishes between tied and untied tasks. Tied tasks can be suspended only at special scheduling points such as creation, completion, taskwait regions, or barriers. In contrast, untied tasks can be suspended

* This material is based upon work supported by the US Department of Energy under Grant No. DE-SC0001621 and by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IS07005C

at any point. In addition, a tied task can be resumed only by the thread that started its execution, whereas an untied task can be resumed by any thread of the team.

The task construct provides an additional concurrency dimension within OpenMP programs, as threads can migrate between tasks and tasks can migrate between threads, although the latter option is available only for untied tasks. This creates a challenge for traditional event-based performance analysis, which instruments certain code locations such as code region entries and exits. Event-based analysis characterizes the control flow of a thread as a sequence of code region enter and exit events, whose consistency may be disrupted by the task scheduler. For example, tasking may violate the proper nesting of enter and exit events, impeding the reconstruction of dynamic call paths or the distinction between inclusive and exclusive performance metrics for a given code region. Furthermore, the sudden suspension of one task in favor of another makes it hard to correctly attribute performance metrics to the first task.

To monitor the missing events that can restore the consistency of the observed sequence and properly expose this additional dimension of concurrency to performance measurement, we have developed portable methods

- to track task suspension and resumption so that it becomes known when a task region is left and reentered;
- to identify individual task instances so that different instances of the same task construct can be properly distinguished; and
- to recognize parent-child relationships between tasks so that it can be determined whether one task acts on behalf of another.

Implemented as an extension of the automatic source-code instrumenter OPARI [10], our solution supports C/C++ programs with tied tasks and with untied tasks that are suspended only at implied scheduling points, a restriction that some OpenMP implementations (e.g., the current Sun compiler) still satisfy. Introducing only negligible measurement overhead, the extra instrumentation inserted for tasking opens the door to a rich variety of performance-analysis applications including the mapping of task instances onto threads.

The outline of the paper is as follows: We start with a survey of related work in Section 2. Then, we present a detailed problem statement in Section 3. After explaining how to obtain unique task identifiers in Section 4, we describe how to establish parent-child relationships between tasks in Section 5. In Section 6, we discuss possible extensions of the OpenMP specification to provide general support for task identifiers with untied tasks. Automated instrumentation for tracking task suspension and resumption with OPARI is the subject of Section 7, followed by an experimental evaluation with respect to measurement dilation in Section 8. The last section draws a conclusion and outlines future work.

2 Related work

Unlike MPI, OpenMP does not specify a standard way of monitoring the dynamic behavior of OpenMP programs, although in the past various proposals have been presented.

The first proposal for a portable OpenMP monitoring interface (named POMP) was written by Mohr et. al [10]. Based on an abstract OpenMP execution model, POMP specifies the names and properties of a set of callback functions, including where and when they are invoked. The proposal also describes the reference implementation OPARI, a portable source-to-source translation tool that inserts POMP calls in Fortran, C, and C++ programs. OPARI is widely used for OpenMP instrumentation, for example, in the performance tools Scalasca [6] and TAU[11]. In an attempt to standardize an OpenMP monitoring interface, a second and significantly enhanced version of POMP [9] was developed by a larger group of people, taking into account experiences with OPARI, the European INTONE project, and the GuideView performance tool from KAI. A prototype tool based on binary instrumentation for this second version of POMP was implemented by DeRose et al. [3]. However, the OpenMP ARB decided to reject POMP 2 because it was seen as too complex and costly to implement and also dropped the idea of standardizing an official monitoring interface for OpenMP. The tools subgroup of the OpenMP ARB agreed on publishing a whitepaper describing a much simpler and less powerful monitoring interface based on a proposal by Sun [7]. OpenMP compiler vendors are encouraged to follow this specification if they want to provide tool support for their OpenMP implementations. To our knowledge, only Sun and Intel include (undocumented and incomplete) implementations of this interface in their compiler offerings. Finally, the group around the OpenUH research compiler investigated various ways of compiler-based instrumentation for OpenMP monitoring [2].

All this work was carried out before the introduction of tasks in OpenMP. Very little has been done so far on tools supporting the monitoring of OpenMP tasks. Fürlinger et al. [5] did initial work on instrumenting OpenMP tasks using OPARI. However, their solution, which simply encloses task regions with enter end exit calls, supports only tied tasks and lacks a task-identification mechanism. In addition, Lin and Mazurov [8] extended the whitepaper API to support tasking and presented a prototype implementation based on the Sun Studio Performance Analyzer.

3 An additional concurrency dimension

Before version 3.0, an OpenMP program had just one concurrency dimension – threads. The latest version of the OpenMP specification added a second dimension. Similar to a thread, a task can be created, suspended, resumed, and carries some state with it. This can have a serious impact on traditional event-based performance analysis tools, which usually consider only the first concurrency dimension.

```
#pragma omp task // Task 1
{
  f1();
}
#pragma omp task // Task 2
{
  f2();
}
#pragma omp taskwait

void f1()
{
  enter_event(f1);
  #pragma omp task // Task 3
  {
    // do something
  }
  #pragma omp taskwait
  exit_event(f1);
}

void f2()
{
  enter_event(f2);
  #pragma omp task // Task 4
  {
    // do something else
  }
  #pragma omp taskwait
  exit_event(f2);
}
```

Fig. 1. Example OpenMP code with tasking.

Event-based performance analysis tools, which are also sometimes referred to as direct-instrumentation tools, instrument certain points in the code, usually the entries and exits of nested code regions, and trigger an event whenever such a point is reached. At runtime, the execution of a thread then appears as a sequence of events delineating nested code region instances. For each code region instance, performance metrics can be individually calculated. Code region instances executed within one another are assumed to be executed on behalf of the enclosing code region instance, establishing the notion of inclusive and exclusive performance metrics (i.e., covering or not covering child region instances, respectively).

Tasking now disrupts this event sequence. A task can be suspended in the middle, and another arbitrary task can be assigned to the executing thread instead. As a consequence, a thread may switch between tasks potentially without

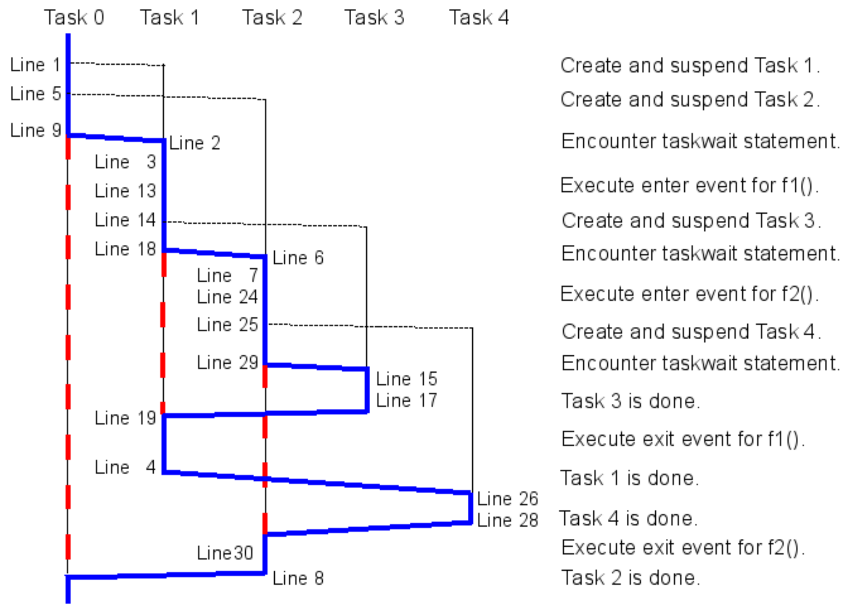
Task 0
Task 1
Task 2
Task 3
Task 4
Line 1
Line 5
Line 9
Line 2
Line 3
Line 13
Line 14
Line 18
Line 6
Line 7
Line 24
Line 25
Line 29
Line 15
Line 17
Line 19
Line 4
Line 26
Line 28
Line30
Line 8
Create and suspend Task 1.
Create and suspend Task 2.
Encounter taskwait statement.
Execute enter event for f1().
Create and suspend Task 3.
Encounter taskwait statement.
Execute enter event for f2().
Create and suspend Task 4.
Encounter taskwait statement.
Task 3 is done.
Execute exit event for f1().
Task 1 is done.
Task 4 is done.
Execute exit event for f2().
Task 2 is done.

Fig. 2. Timelines of tasks in the program from Figure 1 using a FIFO scheduler. Each task is represented as a separate timeline. The continuous thick line indicates the task currently being executed. The dashed thick lines indicate intervals during which tasks are temporarily blocked. The numbers next to the continuous thick line correspond to the source-line numbers in Figure 1.

any parent-child relationship between them. However, since the OpenMP specification provides no method for identifying such relationships, it becomes very hard to calculate meaningful inclusive metrics. Moreover, the execution of tasks may not be properly nested so that call paths can no longer be calculated in the traditional way. Even worse, defining call paths along the execution of tasks without a parent-child relationship might not make sense at all.

This problem is illustrated by the example shown in Figure 1. A potential scheduling order for this program when executed with a single thread is given in Figure 2. The example assumes FIFO scheduling with tasks being immediately suspended once they have been created. The code is supposed to be part of an initial Task 0. Obviously, the order of enter and exit events in this scenario violates the nesting property. Although executed by only a single thread, the events appear in the following order:

$$\text{enter f1()} \rightarrow \text{enter f2()} \rightarrow \text{exit f1()} \rightarrow \text{exit f2()}$$

In addition, we can observe that one task (Task 2) was suspended in favor of another (Task 3) that was not a descendant of the previous one. It is clear that the familiar notions of call paths and of inclusive and exclusive metric values cannot be maintained in this type of event sequence.

If task identifiers were available and events were produced on every task switch, we could correctly measure values which are exclusive for each single task and consider call-path information separately for each task instance. However, a complex computation often contains many tasks, which potentially create child tasks. Having aggregate (inclusive) metrics for a computation including the values of its child tasks eases the location of problem candidates. For this reason, knowing the creation relationships is of importance as well.

4 Obtaining task identifiers for tied tasks

One way to provide unique task identifiers for tied tasks is to have a *task-private* and globally accessible variable that is uniquely initialized on task creation and remains unchanged and valid throughout the active lifetime of the task – whether functions are called from within the task or whether the task is interrupted during its lifetime.

Such a task-private variable can be emulated using a threadprivate global variable that is constant during the execution of a task but changes its value at task scheduling points. If we proceed with a newly created task, we set the variable to the identifier of the new task, and if we resume a previously suspended task, we set the variable to the corresponding previous value. For this purpose, we need a mechanism to store and to restore task identifiers at scheduling points as well as a method to obtain unique identifiers for new tasks.

This can be implemented by declaring a threadprivate variable `current_task_id` in the global scope that will give us the valid task identifier any time during program execution (see Figure 3). It is initialized with a value corresponding to the initial, implicit task. A function to obtain fresh task identifiers, `get_new_task_id()`, can be implemented by concatenating the 32 bit OpenMP thread identifier with a 32 bit threadprivate variable that is incremented everytime the function is called. The resulting 64-bit combination will provide globally unique identifiers within every parallel region. Another advantage of this combination is that we do not require any synchronization to obtain new and unique task identifiers.

To store and to restore task identifiers, we place a local (i.e., automatic storage) variable `old_task_id` before each scheduling point and initialize it with the value of `current_task_id`. If a task is suspended and resumed later on, the corresponding local variable is still valid and used to restore the value of `current_task_id`.

To maintain task identifiers throughout the entire execution of an application, instrumentation must be applied

- to parallel regions,
- to taskwait as well as implicit and explicit barrier constructs, and
- to task constructs.

Before we enter a parallel region (see Figure 4), we need to store the current task identifier so that it can be restored afterwards. Therefore we add a local

```
int64_t current_task_id = ROOT_TASK_ID;
#pragma omp threadprivate(current_task_id)
```

Fig. 3. Declaration and initialization of task identifiers.

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp parallel
  {
    current_task_id = get_new_task_id();
    // do something
  }
  current_task_id = old_task_id;
}
```

Fig. 4. Maintaining task identifiers at parallel regions.

variable `old_task_id` and assign to it the identifier of the current task. After the completion of the parallel region, the local variable is still valid and we can restore the original value by assigning `old_task_id` to `current_task_id`.

Inside the parallel region, each thread creates an implicit task and we need to obtain a unique identifier for each of these tasks. This is done in parallel by assigning the return values of calls to `get_new_task_id()` to the threadprivate variables `current_task_id` right at the beginning of the parallel region. Each thread/task has now a unique `current_task_id` variable that is valid until we reach the next scheduling point.

Scheduling points within parallel regions occur at `taskwait` and `barrier` as well as at `task` constructs. Here we must maintain task identifiers because the scheduler can suspend the current task and continue either with a newly created or a resumed task. At `taskwait` and `barrier` constructs, we store the `current_task_id` in a new local variable `old_task_id` immediately before reaching the scheduling point and restore it afterwards, as shown in Figure 5. At task creation points the situation is slightly different. In addition to storing and restoring the identifier of the current task, we need to obtain a new task identifier by calling `get_new_task_id()` and assign it to `current_task_id`, similar to the procedure used for parallel regions. This is demonstrated in Figure 6.

If applied to all parallel regions and scheduling points, the code sequences in Figure 4, Figure 5 and Figure 6 are sufficient to maintain task identifiers throughout the entire execution of a program.

5 Tracking the task creation hierarchy

With task identifiers available, a task creation hierarchy can be constructed. The instrumentation presented in the previous section allows the identifier of the

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp taskwait
  current_task_id = old_task_id;
}
```

Fig. 5. Storing and resetting task identifiers at taskwait statements (applies also to barrier constructs).

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp task
  {
    current_task_id = get_new_task_id();
    // do something
  }
  current_task_id = old_task_id;
}
```

Fig. 6. Maintaining task identifiers at task creation points.

parent task to be obtained at the task creation point. Subsequently, the direct parent-child relationship can be easily extended to a full pedigree by appending the new task as a child node of the parent task node to the tree of the creation hierarchy.

When a new task is created, the parent task must have stored its identifier in the local variable `old_task_id` in order to restore the identifier when continuing its execution. Making the value of `old_task_id` firstprivate in the child task ensures that it is initialized with the parent's identifier, establishing a connection between the two. The instrumentation for task creation is shown in Figure 7, assuming the creation tree is built using a function named `add_child()`.

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp task firstprivate(old_task_id)
  {
    current_task_id = get_new_task_id();
    add_child(current_task_id, old_task_id);
    // do something
  }
  current_task_id = old_task_id;
}
```

Fig. 7. Tracking the task creation hierarchy.

6 Untied tasks

The mechanisms described so far assume that scheduling happens only at implied scheduling points that can be easily instrumented. This assumption is always true for tied tasks. Compared to tied tasks, the scheduling of untied tasks is more flexible in two ways. An untied task

– may be resumed by a thread different from the one that executed the task before it was suspended and
– may be suspended at any point, not only at implied scheduling points.

The first condition does not cause any problem as long as rescheduling occurs only at scheduling points. Since a performance-analysis tool using our interface is able to verify the identity of a thread, it can easily distinguish between different threads. For this reason, our solution also works with OpenMP implementations that reschedule untied tasks only at implied scheduling points. Although not prescribed by the OpenMP specification, some implementations (e.g., the current Sun compiler) still follow this rule because rescheduling at scheduling points, where control is trivially transferred to the OpenMP runtime, is technically simpler than it is at arbitrary points. However, untied tasks that are rescheduled at arbitrary points may disrupt the whole measurement.

General support for untied tasks requires additional services from the runtime system. At least, notification on task scheduling events is necessary so that whenever a task is preempted performance metrics can be collected and the task identifier can be set to the new task. One way of implementing such a notification mechanism would be the option to register a callback function `cb_resume_task()` with the OpenMP runtime that is called whenever the execution of a task is started or resumed. Inside `cb_resume_task()`, the environment of the task brought to execution should be visible. Furthermore, because the runtime system must maintain task identifiers anyway, it would be helpful and probably more efficient if the OpenMP runtime system offered a standard way of obtaining task identifiers rather than having to maintain them on the user level. While in our view an extension of the OpenMP specification would be the ideal solution, the current situation leaves us with only the following options:

– Exploit the fact that some OpenMP implementations (e.g., Sun) suspend untied tasks only at scheduling points.
– Let the instrumentation make all tasks tied. This changes the behavior of the measured program, but in some cases still allows a few very limited conclusions to be drawn (e.g., on the granularity of tasks).

7 Automated instrumentation with OPARI

OPARI [10] is a source-to-source instrumentation tool for OpenMP programs. To allow automated instrumentation of OpenMP C/C++ programs with tasking, OPARI was extended to instrument also the task and taskwait constructs.

Table 1. Examples of how OPARI instruments tasking-related constructs.

OMP directive	instrumented directive

#pragma omp task
{
// do something
}
	{
	POMP_Task_create_begin(pomp_region_1);
	POMP_Task_handle pomp_old_task =
	            POMP_Get_current_task();
	#pragma omp task firstprivate(pomp_old_task)
	{
	POMP_Set_current_task(POMP_Task_begin(
	            pomp_old_task, pomp_region_1));
	{
	// do something
	}
	POMP_Task_end(pomp_region_1);
	}
	POMP_Set_current_task(pomp_old_task);
	POMP_Task_create_end(pomp_region_1);
	}

#pragma omp taskwait	{
	POMP_Taskwait_begin(pomp_region_1);
	POMP_Task_handle pomp_old_task =
	            POMP_Get_current_task();
	#pragma omp taskwait
	POMP_Set_current_task(pomp_old_task);
	POMP_Taskwait_end(pomp_region_1);
	}

Furthermore, the instrumentation of OpenMP directives that contain (implicit) scheduling points or create implicit tasks was modified to maintain and to expose task identifiers and parent-child relationships based on the ideas presented in Sections 4 and 5.

Access to `current_task_id` is encapsulated and provided via two functions: `POMP_Get_current_task()` and `POMP_Set_current_task()`. The function `get_new_task_id()` is not called directly, but inside the region-begin function (e.g., in `POMP_Task_begin()`). Some examples of the instrumentation are shown in Table 1.

8 Overhead

To evaluate the runtime dilation of our instrumentation, we performed two tests, the first one based on an artificial benchmark, the second one based on a realistic code example, the Flexible Image Retrieval Engine (FIRE) code [4]. The instrumented calls generated unique identifiers for each task, but did not measure any further metrics.

8.1 Artificial benchmark

Our benchmark program contained a parallel region in which 10,000,000 tasks per thread were created. Every task just incremented an integer by one. Before executing the program with four threads, it was instrumented using the extended version of OPARI. The execution time was measured and compared against the uninstrumented version. This test was run on an i686 Linux system with a 2.66 GHz quadcore processor using four threads.

Running the uninstrumented program took 1.89 s, while running the instrumented program took 2.50 s. The difference was 0.61 s or 32.3 %. Ignoring program initialization and the increment instruction inside the tasks, the uninstrumented benchmark spent its execution time almost exclusively managing the tasks. This implies that our instrumentation adds approximately 32.3 % of the task management time to the overall runtime. An absolute overhead of 0.61 s for an application with 10,000,000 tasks per thread doing real work is probably negligible. However, acquisition of performance metrics upon the occurrence of task scheduling events might incur additional overhead.

8.2 The FIRE code

The Flexible Image Retrieval Engine (FIRE) [4] was developed at the Human Language Technology and Pattern Recognition Group of RWTH Aachen University. The benchmark version subject to our study consists of more than 35,000 lines of C++ code. Given a query image and the number of desired matches k, a score is calculated for every image in the database, and the k database entries with the highest scores are returned. Shared-memory parallelization is obviously more suitable than distributed-memory parallelization for the image retrieval task, as the image database can be easily accessed by all threads and need not be distributed.

The initial parallelization of the FIRE code used nested OpenMP threads on two nesting levels [12]. This version was later modified to use OpenMP tasks instead of nested threads. The task-based version creates one task per query image and inside these tasks every comparison of a query picture with a database entry is represented by another task. This approach offers more flexibility than using nested threads because every thread can work on any task. With nested threads, in contrast, we had to assign a fixed number of threads to the lower nesting level.

For our experiments, we used 18 query images and a database with 1000 elements. Since every comparison generates a task, 18000 tasks were created in total. We ran the code on an IBM eServer LS42 equipped with four AMD Opteron 8356 (Barcelona) processors. We conducted ten test runs with and without instrumentation, while varying the number of threads. The average runtimes are shown in Table 2.

The results clearly show that the overhead generated by the instrumentation is insignificant. The total absolute overhead when one thread is used is about 5 s. Compared to the total runtime of 527.56 s this is less than 1 %. When more

Table 2. Comparison of the instrumented against the uninstrumented version of the FIRE code.

	runtime		overhead	
threads	not instrumented	instrumented	in %	in seconds
1	522.57 s	527.56 s	0.96 %	4.99 s
2	259.55 s	262.64 s	1.19 %	3.09 s
4	129.52 s	129.93 s	0.32 %	0.41 s
6	86.42 s	86.43 s	0.01 %	0.01 s
8	64.86 s	65.13 s	0.41 %	0.27 s
12	43.13 s	43.00 s	-0.30 %	-0.13 s
16	32.12 s	32.43 s	0.95 %	0.31 s

threads are used, the overhead scales well with the number of threads. When 16 threads are used the overhead amounts to 0.31 s which is still less than 1 % of the total runtime of 32.43 s. So even for larger numbers of threads, the instrumentation overhead is very low compared to the overall execution time.

9 Conclusion and future work

A portable method was presented that allows the execution and scheduling of tied and untied OpenMP tasks to be tracked and exposed to performance measurement. Our method can be applied as long as task scheduling occurs only at the implied scheduling points defined in the OpenMP specification, which is always the case for tied tasks and in some implementations (e.g., the current Sun compiler) even for untied tasks.

Implemented as an extension of OPARI, the necessary instrumentation can be automatically inserted into the source code of OpenMP programs written in C/C++. In a next step, we plan to extend our solution to Fortran. The runtime dilation caused by the instrumentation was shown to be negligible, although we believe that the overhead could probably be further reduced if the task identifier was provided by the runtime environment. Performance tools using OPARI are now encouraged to implement a rich variety of analyses of the events delivered by our interface, taking advantage of the two concurrency dimensions (i.e., threads and tasks) being fully exposed – including the mapping between them. Application candidates include analyzing the task synchronization overhead in view of many small tasks, determining the granularity distribution among tasks, studying the task creation hierarchy, and drawing execution timelines of parallel tasks. Code studies will have to show which ones are most relevant. Displaying data related to the two concurrency dimensions in a meaningful way and handling the potential non-determinism of task scheduling will pose major challenges.

General support for untied tasks, which are in principle allowed to be suspended at arbitrary points, cannot be provided unless the OpenMP runtime exposes task scheduling events, which would require an extension of the OpenMP specification. At a minimum, the user should be given the option to register a function to be called whenever a task's execution is started or resumed.

References

1. OpenMP Architecture Review Board. OpenMP application progam interface version 3.0. Technical report, OpenMP Architecture Review Board, May 2008.
2. Van Bui, Oscar Hernandez, Barbara Chapman, Rick Kufrin, Danesh Tafti, and Pradeep Gopalkrishnan. Towards an implementation of the OpenMP collector API. In *Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the ParCo 2007 Conference*, Jülich, Germany, September 2007.
3. Luiz DeRose, Bernd Mohr, and Seetharami Seelam. Profiling and tracing OpenMP applications with POMP based monitoring libraries. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, volume 3149 of *Lecture Notes in Computer Science*, pages 47–54, Pisa, Italy, August - September 2004. Springer.
4. Thomas Deselaers, Daniel Keysers, and Hermann Ney. Features for image retrieval - a quantitative comparison. In *26th DAGM Symposium, Pattern Recognition (DAGM 2004)*, volume 3175 of *Lecture Notes in Computer Science*, pages 228 – 236, Tübingen, Germany, 2004.
5. Karl Führlinger and David 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.
6. Markus Geimer, Felix Wolf, Brian J.N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.
7. Marty Itzkowitz, Oleg Mazurov, Nawal Copty, and Yuan Lin. An OpenMP runtime API for profiling. Technical report, Sun Microsystems, Inc., 2007.
8. Yuan Lin and Oleg 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.
9. Bernd Mohr, Allen D. Malony, Hans-Christian Hoppe, Frank Schlimbach, Grant Haab, Jay Hoeflinger, and Sanjiv Shah. A performance monitoring interface for OpenMP. In *Proceedings of the 4th European Workshop on OpenMP (EWOMP'02)*, Rome, Italy, September 2002.
10. Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, August 2002.
11. Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.
12. Christian Terboven, Thomas Deselaers, Christian Bischof, and Hermann 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.