Open Source Task Profiling by Extending the OpenMP Runtime API

Ahmad Qawasmeh[1], Abid Malik[1], Barbara Chapman[1], Kevin Huck[2], and Allen Malony[2]

[1] University of Houston, Dept. of Computer Science, Houston, Texas
{arqawasm,malik,chapman}@cs.uh.edu
www2.cs.uh.edu/~hpctools
[2] University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon
{khuck,malony}@cs.uoregon.edu
www.cs.uoregon.edu/research/tau/home.php

Abstract. The introduction of tasks in the OpenMP programming model brings a new level of parallelism. This also creates new challenges with respect to its meanings and applicability through an event-based performance profiling. The OpenMP Architecture Review Board (ARB) has approved an interface specification known as the "OpenMP Runtime API for Profiling" to enable performance tools to collect performance data for OpenMP programs. In this paper, we propose new extensions to the OpenMP Runtime API for profiling task level parallelism. We present an efficient method to distinguish individual task instances in order to track their associated events at the micro level. We implement the proposed extensions in the OpenUH compiler which is an open-source OpenMP compiler. With negligible overheads, we are able to capture important events like task creation, execution, suspension, and exiting. These events help in identifying overheads associated with the OpenMP tasking model, e.g., task waiting until a task starts execution or task cleanup etc. These events also help in constructing important parent-child relationships that define tasks' call paths. The proposed extensions are in line with the newest specifications recently proposed by the OpenMP tools committee for task profiling.

Keywords: OpenMP, OpenMP Runtime API for Profiling, Open-Source Implementation, OpenMP Tasks.

1 Introduction

OpenMP is a standard API for shared memory programming. It provides a directive-based programming approach for generating parallel versions of programs from the sequential ones. The compiler generated code invokes the OpenMP runtime library routines to create and manage threads and tasks. The lack of standards in the runtime layer has hampered the development of third-party tools to

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 186–199, 2013.

support OpenMP application development. The OpenMP Runtime API (ORA) for profiling OpenMP applications was presented in [9]. The ORA has been accepted by the tools committee of the OpenMP Architecture Review Board (ARB). The API is designed to permit a tool, known as a collector, to gather information about an OpenMP program from the runtime system in such a manner that neither the collector nor the runtime system needs to know any details about each other. The ORA is designed to ensure that tool developers are not required to have an insight into the details of the different OpenMP implementations. However, these tools should maintain information about the OpenMP execution model in order to trouble-shoot the OpenMP specific performance problems.

The ORA is an event-based interface that relies on bi-directional communications between a performance tool, i.e. collector, and an OpenMP runtime library. The communication is established through a collection of requests that take a send-receive protocol with a distinct functionality for each request. The main advantage of the ORA is that no modifications to an application's source code are required. Consequently, compiler analysis and optimizations will not be affected. Hence, the performance measurements of an application, collected by a performance tool, are more accurate and specific.

Traditionally, parallel programming models for shared memory multiprocessors had focused on scientific applications using large arrays and exhibiting loop level parallelism. In order to exploit the new massive parallelism provided by the modern architectures, the parallel programming models had to propose a new dimension of concurrency to cap available parallelism within high performance computing applications. Applications exhibiting irregular parallelism in the form of recursive algorithms and pointer based data structures were not taken care of before the introduction of tasking in the OpenMP programming model. Tasking has added a new dimension of concurrency, represented by the task construct, to OpenMP applications. The task construct allows a developer to dynamically create asynchronous units of work to be scheduled at runtime. Two types of tasks have been introduced in the OpenMP specification; **1) Tied tasks 2) Untied tasks**. Tied tasks can be suspended at specific scheduling points that include the creation of tasks, taskwait constructs, barriers, and completion of tasks etc. Untied tasks can be suspended at any point in an OpenMP program according to OpenMP 3.1 specifications. Moreover, a tied task can be resumed only by the thread that started its execution while an untied task can be resumed by any thread in the team.

In order to handle the challenges and performance issues associated with the introduction of tasks, we propose new extensions to the ORA in the OpenMP runtime library of the OpenUH compiler [17], [2]. The main motivation behind this work lies in observing the viability of monitoring the individual task instances and tracking the events associated with each one of them at the micro level. The new API we propose allows developers to:

– Distinguish the individual task instances in the same task construct by assigning a distinct ID to each task instance.

- Distinguish the task instances that get suspended at the different scheduling points.
- Construct parent-child relationships between tasks, which can be used to construct a task tree.
- Track task creation, switching, suspension, resumption, exiting, and completion.
- Allow collector tools to maintain performance measurements associated with the aforementioned events.

Our implementation supports C/C++ and Fortran programs with tied tasks and untied tasks. Furthermore, our implementation is in line with the task profiling specification proposed by the OpenMP ARB tools committee [4]. Reference implementations of the ORA are sparse since it requires compiler and OpenMP runtime library support. To the best of our knowledge, the work reported here is the first open-source implementation of the ORA with extensions to support tasks.

The remainder of the paper is organized as follows. Section 2 briefly describes the OpenMP task implementation in the OpenUH compiler runtime library. Section 3 gives details about our new extensions in the ORA for task profiling. Section 4 presents the experimental framework used to evaluate the implementation. The related work is discussed in Section 5. Finally, Section 6 concludes our contributions and discusses directions for the future work.

2 OpenMP Tasking Implementation in OpenUH

The OpenUH compiler supports OpenMP 3.0 tasking on the IA-64, IA-32, x86_64, and Opteron Linux ABI platforms. This includes the front-end support ported from the GNU C/C++ compiler, back-end translation implemented by the HPCTool group at the University of Houston jointly with the Tsinghua University, and an efficient task scheduling infrastructure developed by the HPCTools group. The HPCTools group also implemented a configurable task pool framework that allows a user to choose an appropriate task queue organization at runtime.

We use the popular Fibonacci code in Figure 1a to explain the role of the OpenMP runtime library regarding our implementation. In the code, two task constructs have been inserted to handle recursion in a dynamic parallel fashion. In the same manner, a taskwait construct has been used to get correct results by preventing parent tasks from proceeding while child tasks are still running. The number of tasks created depends on the value of integer n. The task construct will create a task instance. The execution of the task will be deferred based on the availability of threads and the status of its children. Figure 2 shows how the OpenMP tasking directives in Figure 1a are translated into the OpenMP runtime routines. A description of the tasking runtime routines is given in Table 1.

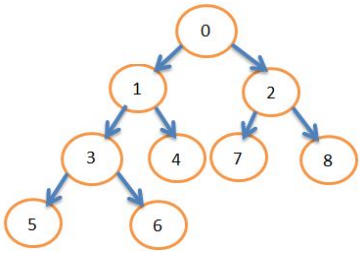
We use the OpenMP runtime routines to capture the OpenMP events and states related to the OpenMP task, such as task creation, task waiting in the task

Table 1. Description of the OpenMP tasking runtime routines in OpenUH

Routine	Description
__ompc_task_create()	creates a task and inserts it into a queue
__ompc_task_wait()	suspends a task until all of its children complete
__ompc_task_exit()	called at the end of a task to perform cleanup and schedule a new task
__ompc_task_switch()	switches the execution from one task to another
__ompc_task_firstprivates_alloc()	allocate memory for firstprivate copies
__ompc_task_will_defer()	checks if a task should be deferred or executed immediately
__ompc_task_firstprivates_free()	deallocate memory for firstprivate copies

```
int fib(int n)
{
  int x,y;
  if (n<2)
    return n;
#pragma omp task shared(x)
  x=fib(n-1);
#pragma omp task shared(y)
  y=fib(n-2);
#pragma omp taskwait
  return x+y;
}
```

(a) Fibonacci code



(b) A task tree (n=4)

Fig. 1. Fibonacci OpenMP tasking example

pool, task switching from a create state to a suspend state etc. These states and events are captured by simply modifying the OpenMP runtime routines, without modifying the OpenMP translation of the source code. The ORA extensions to support task profiling provide an API to query the OpenMP runtime library for task states and event notifications using callback functions.

3 Implementation of the OpenMP Tasking Profiling APIs

The ORA interface [9] consists of a single routine that takes the form: *int __omp_collector_api (void *arg)*. The **arg** parameter is a pointer to a byte array that can be used by a collector tool to pass one or more requests for information from the runtime. The collector requests notification of a specific event by passing the name of the event to be tracked as well as a callback routine to be invoked by the OpenMP runtime each time the event occurs. Figure 4 demonstrates the interaction between the collector and the OpenMP runtime library through the Collector API. As shown, this interaction is acheived by a set of implemented requests.

The aforementioned single routine is implemented once in the runtime and its symbol is exported in the OpenMP runtime library. This strategy allows the tool

```
extern _INT32 fib(_INT32 n)
{
  register _INT32 _w2c___ompv_task_is_deferred;
  register _UINT64 _w2c_reg3;
  register _INT32 _w2c___ompv_task_is_deferred0;
  _INT32 x;
  _INT32 y;
  struct task_args__omprg_fib_1 * __ompv_taskarg1;
  struct task_args__omprg_fib_2 * __ompv_taskarg2;
  if(n <= 1)
  {
    return n;
  }
  _w2c___ompv_task_is_deferred = __ompc_task_will_defer(1);
  if(_w2c___ompv_task_is_deferred)
  {
    __ompc_task_firstprivates_alloc(&__ompv_taskarg1, 4);
    (__ompv_taskarg1) -> n = n;
  }
  else  { __ompv_taskarg1 = (struct task_args__omprg_fib_1 *)(0ULL);}
  __ompc_task_create(&__omprg_fib_1, _w2c_reg3, __ompv_taskarg1, 1, 1, 0);
  _w2c___ompv_task_is_deferred0 = __ompc_task_will_defer(1);
  if(_w2c___ompv_task_is_deferred0)
  {
    __ompc_task_firstprivates_alloc(&__ompv_taskarg2, 4);
    (__ompv_taskarg2) -> n = n;
  }
  else  {__ompv_taskarg2 = (struct task_args__omprg_fib_2 *)(0ULL);}
  __ompc_task_create(&__omprg_fib_2, _w2c_reg3, __ompv_taskarg2, 1, 1, 0);
  __ompc_task_wait();
  return x + y;
}
```

Fig. 2. OpenUH translation of the Fibonacci code in Fig. 1a into explicitly multi-threaded code

to check whether the symbol exists via a dynamic linker in order to establish a communication with the runtime and start sending requests and monitoring events and states. The OpenMP runtime should distinguish thread's states, which are related to tasks. These states include when a task is created, suspended, existing, or being executed. When the collector tool makes a request for notification of a specified task event(s), the OpenMP runtime will start keeping track of this event inside its environment. The collector may also make requests to pause, resume, or stop event generation.

When a collector tool sends a request to register any event through the ORA, the event type *OMP_COLLECTORAPI_REQUEST* and a callback function pointer is passed as an argument to the API call in the runtime. Race conditions might occur when multiple threads try to register the same event with multiple callbacks. The callback function pointer is stored in a table in which each entry has a lock associated with it to prevent race conditions. This table contains the event callbacks shared by all the threads. The frequency in which the events are registered relies on the nature of the collector tool. Two functions, *__ompc_event_callback(event)* and *__ompc_set_state(state)*, are inserted at different positions in the OpenMP runtime task routines specifed in Table 1. These functions implement the different events and states associated with

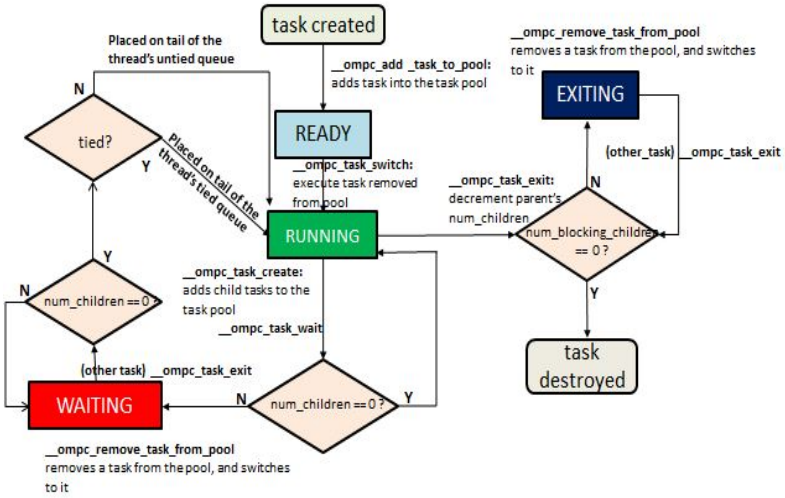


Fig. 3. OpenUH tasking execution model

the task instances. The state values are stored in the OpenMP thread descriptor in the runtime. Once a thread reaches an event point, the function *__ompc_event_callback((OMP_COLLECTORAPI_EVENT) e)* is executed and the callback function, associated with this event, is invoked. The functionality of the callback is determined by a performance tool in order to collect the required performance measurements.

The OpenMP ARB tools committee proposed a framework for task profiling in the face to face meeting recently held at the University of Houston [4]. The proposal categorizes the profiling events into two groups 1) mandatory events 2) optional events. The proposal defines the OpenMP task creation and task exiting as mandatory events, while task waiting and task switching have been defined as optional events. Our extensions include support for the mandatory events proposed during this meeting. We also provide support for some optional events in addition to some other events specific to our tasking model. Figure 3 shows the task execution model implemented in the OpenUH runtime to support OpenMP tasks. The model depicts all the different states encountered by each task instance starting from task's creation to its completion ,i.e., when the task is destroyed. In the following sections, we describe our extensions in detail.

3.1 Task Creation Events and States

These events and states are designed to capture the start and completion of a task instance creation. The following states and events have been defined:

– *THR_TASK_CREATE_STATE*: The enumerated value of this state is assigned to the thread's state field in the descriptor whenever the thread is working on a task creation.

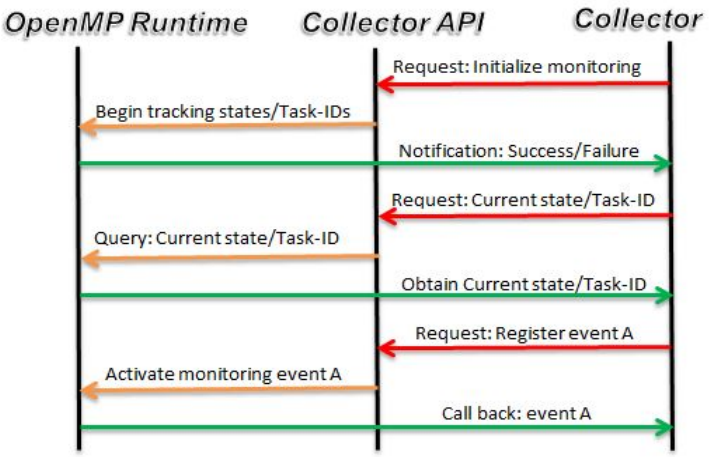


Fig. 4. Example of an interaction between collector and OpenMP runtime

- *OMP_EVENT_THR_BEGIN_CREATE_TASK*: This event indicates that the parent task creates a new explicit task before the new task starts execution.
- *OMP_EVENT_THR_END_CREATE_TASK_IMM*: This event indicates that the process of creating the task is done and its execution will start immediately.
- *OMP_EVENT_THR_END_CREATE_TASK_DEL*: This event indicates that the process of creating the task is done and its execution will start with a delay.

3.2 Task Suspension Events and States

These events and states are designed to capture the start and completion of a task instance suspension. This suspension occurs when the taskwait construct is encountered.

- *THR_TASK_SUSPEND_STATE*: The enumerated value of this state is instantly assigned to the thread's descriptor once the parent task encounters a taskwait construct. The thread, working on the parent task, will later be asssigned to another work. Child tasks, associated with the suspended task, should finish their execution in order for the suspended task to resume its execution.
- *OMP_EVENT_THR_BEGIN_SUSPEND_TASK*: This event indicates that the parent task has been suspended.
- *OMP_EVENT_THR_END_SUSPEND_TASK*: This event indicates the completion of the parent task's suspension.

3.3 Task Execution/Exiting Events and States

These events and states are designed to capture the start and completion of a task instance execution and exiting. Once a new task is created, it may start executing immediately or with some delay depending on the availability of threads.

- *THR_WORK_STATE*: The enumerated value of this state is assigned to the thread's descriptor once the thread starts the execution of a task.
- *OMP_EVENT_THR_BEGIN_EXEC_TASK*: This event is hit once the task's execution begins.
- *OMP_EVENT_THR_BEGIN_FINISH_TASK*: This event indicates that the task's execution is done. This event is hit immediately after the previous event if the task being executed does not encounter a taskwait construct.
- *OMP_EVENT_THR_END_FINISH_TASK*: This event indicates that the removal of the task from the task-pool and the required cleanup have successfully been completed.

By defining these new states, we guarantee that a thread will always have a distinct state associated with it while working on tasks. The collector tool can request the state of a thread at any given point during the execution of the program.

3.4 Task IDs and Parent Task IDs

In order to distinguish the various task instances, keep track of their associated events, and construct parent-child relationships between tasks, we have added a new OpenMP task ID field to the task data structure descriptor. It is initialized with a value corresponding to the initial implicit task. Each time a new task is created, the task ID is incremented atomically to ensure that only one thread can modify this field at any instance of time. The parent task ID is obtained by having a pointer to the parent task. Two requests are defined to enable the collector tool to obtain these IDs at any given point of the program execution.

4 Evaluation

We evaluated our implementation in the OpenUH compiler. We performed the following two analyses;

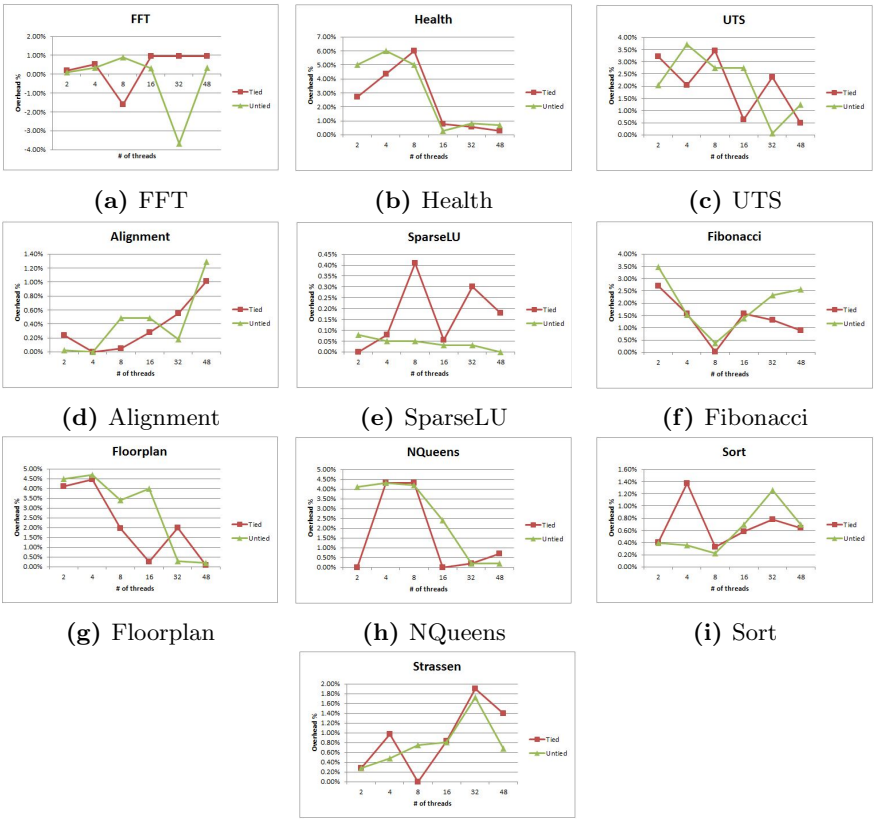
- We measured the overheads introduced by the inclusion of our implementation in the runtime.
- We tracked the newly developed task IDs, states, and events through our employed requests. This part was achieved by developing a prototype OpenMP task profiler tool.

We used the Barcelona OpenMP Task Suite (BOTS) kernels [3] as benchmark applications. The experiments were done using the x86_64 Linux system with four 2.2 GHz 12-core AMD Opteron processor (48 cores total).

4.1 Overhead Measurements

Table 2 gives details about our measurements. Each sub-table represents a kernel in the BOTS. Interested readers can consult the work [3] for full details about these kernels. We used six different numbers of threads. We collected data for both *tied* and *untied tasks*. Each kernel has two versions. One with tied tasks and the other with untied tasks. To calculate the overheads, we compiled the benchmark kernels using the OpenUH compiler. We ran the binaries with our OpenMP runtime library, while the tool is not attached. We employed a without vs. a with scenario, in which the without case excludes: assigning an ID to each task instance, assigning states to threads while working on tasks, tracking tasking events, and implementing task requests.

The results show that the overhead associted with our implementation is insignificant. The absolute overhead percentage ranges from 0% to 6% of the execution time. The average overhead percentage obtained is less than 1%. Overhead detail from Floorplan and NQueens kernels are given by Table 2g and Table 2h



(a) FFT **(b)** Health **(c)** UTS

(d) Alignment **(e)** SparseLU **(f)** Fibonacci

(g) Floorplan **(h)** NQueens **(i)** Sort

(j) Strassen

Fig. 5. BOTS overhead comparison (Tied vs. Untied)

respectively. These two kernels produced the maximum overhead. SparseLU and FFT, overhead described in Table 2e and Table 2a respectively, generated the minimum overhead. The variation in overhead is due to the behavior of these benchmarks.

Figure 5 demonstrates the overhead percentage obtained using tied vs. untied tasks for all the aforementioned kernels. The x-axis in each sub-figure represents the number of threads, while the y-axis represents the overhead percentage. As we can see from Figure 5, the behavior of tied and untied tasks, in terms of the range of deviations and the overhead's average, is very similar, except for the SparseLU benchmark shown in Figure 5e, where tied tasks have higher overheads.

Furthermore, when more threads are used, the overhead scales well with the increment of threads. As an example, when 48 threads are used, the worst overhead percentage obtained, among the different kernels, is 2.56%.

Table 2 can be consulted to obtain detailed overhead measurements about each benchmark.

4.2 Prototype OpenMP Task Profiler Tool

The motivation behind having such a profiler is to evaluate our proposed extensions in the runtime. The tool's functionality is to collect profiling measurements regarding task instances. These measurements can lead to a better utilization of task load-balancing, scheduling, and reducing overheads. OMP_REQ_START request should be used first to initialize the collector API to establish a connection with the runtime. OMP_REQ_RE

$GISTER$ request should be used next to selectively register the task events required for our calculations. $OMP\_REQ\_TASK\_ID$ and $OMP\_REQ\_TASK\_PID$ are used to get the task ID and the parent task ID respectively to construct the task-tree. By tracking the task events, we are able to request the thread-ID associated with each task as well as the thread's state while working on these tasks at any instance of time. Our tool also enables developers to obtain measurements about:

- Task instance creation time
- Task instance suspension time
- Task instance execution time
- Task instance cleanup and destroying time
- Task instance overhead waiting after creation to start execution

We have tested our tool with all the BOTS kernels. Our tool is capable of tracking millions of task instances including their IDs, states, and events. For the sake of simplicity, we show how our tool is useful by using the Fibonacci code shown in Figure 1a with different input sizes N and two threads. Figure 1b displays the task tree showing the parent and child tasks for each instance associated with their task IDs when N=4. Task 2 has to wait for tasks 7 and 8 until they finish their execution. We found that task 7 and task 8 were running in parallel since the two

Table 2. The Barcelona OpenMP Task Suite (BOTS) overhead measurements

(a) FFT

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	10.35/10.26	10.37/10.27	0.02/0.01	0.19%/0.097%
4	5.94/5.65	5.971/5.67	0.031/0.02	0.52%/0.35%
8	3.73/3.36	3.67/3.39	(-0.06)/0.03	(-1.61%)/0.89%
16	2.839/3.3	2.866/3.31	0.027/0.01	0.95%/0.30%
32	3.17/4.33	3.2/4.17	0.03/-0.16	0.95%/-3.69%
48	4.22/5.73	4.26/5.75	0.04/0.02	0.95%/0.35%

(b) Health

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	1.85/1.8	1.9/1.9	0.05/0.1	2.7%/5%
4	1.15/1.13	1.2/1.2	0.05/0.07	4.34%/6%
8	1/0.95	1.06/1	0.06/0.05	6%/5%
16	1.28/1.25	1.29/1.26	0.01/0.004	0.78%/0.3%
32	1.25/1.19	1.26/1.2	0.007/0.01	0.55%/0.8%
48	1.36/1.36	1.37/1.37	0.004/0.01	0.29%/0.7%

(c) UTS

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	1.55/1.47	1.6/1.5	0.05/0.03	3.22%/2.04%
4	1.47/1.35	1.5/1.4	0.03/0.05	2.04%/3.70%
8	1.45/1.45	1.5/1.49	0.05/0.04	3.44%/2.75%
16	1.59/1.46	1.6/1.5	0.01/0.04	0.63%/2.74%
32	1.68/1.499	1.72/1.5	0.04/0.001	2.38%/0.07%
48	1.99/1.59	2/1.61	0.01/0.02	0.5%/1.25%

(d) Alignment

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	8.28/8.27	8.3/8.272	0.05/0.002	0.24%/0.024%
4	4.13/4.14	4.13/4.14	0/0	0%/0%
8	2.079/2.07	2.08/2.08	0.001/0.01	0.05%/0.48%
16	1.046/1.044	1.049/1.049	0.003/0.005	0.28%/0.48%
32	0.546/0.541	0.549/0.542	0.003/0.001	0.55%/0.18%
48	0.396/0.389	0.4/0.394	0.004/0.005	1.01%/1.29%

(e) SparseLU

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	1.27/1.278	1.27/1.28	0/0.001	0%/0.08%
4	3.661/3.663	3.664/3.66	0.003/0.002	0.08%/0.05%
8	1.945/1.955	1.953/1.956	0.008/0.001	0.41%/0.05%
16	1.0884/1.08	1.089/1.082	0.001/0.0003	0.055%/0.03%
32	0.658/0.659	0.66/0.659	0.002/0.0002	0.30%/0.030%
48	0.54/0.54	0.541/0.54	0.001/0	0.18%/0%

(f) Fibonacci

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	4.38/2.899	4.5/3	0.12/0.101	2.7%/3.48%
4	3.79/1.97	3.85/2	0.06/0.03	1.58%/1.52%
8	3.529/2.59	3.53/2.6	0.001/0.01	0.03%/0.38%
16	3.19/2.86	3.24/2.9	0.05/0.04	1.56%/1.39%
32	5.28/2.15	5.28/2.2	0.07/0.05	1.32%/2.32%
48	6.68/1.95	6.74/2	0.06/0.05	0.89%/2.56%

(g) Floorplan

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	24/11	25/11.5	1/0.5	4.1%/4.5%
4	11/7.5	11.49/8	0.49/0.5	4.45%/6.6%
8	9.7/8.7	9.89/9	0.19/0.3	1.95%/3.4%
16	11.87/12.5	11.9/13	0.03/0.5	0.25%/4%
32	11.02/9.7	11.24/9.73	0.22/0.03	1.99%/0.3%
48	10.98/9.66	10.99/9.68	0.01/0.02	0.09%/0.2%

(h) NQueens

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	5/4.8	4/5	0/0.2	0%/4.1%
4	3.5/3.5	3.65/3.65	0.15/0.15	4.3%/4.3%
8	3.5/3.6	3.65/3.75	0.15/0.15	4.3%/4.2%
16	4.35/4.12	4.35/4.22	0/0.1	0%/2.4%
32	5.17/4.68	5.18/4.69	0.01/0.01	0.19%/0.21%
48	5.64/5.01	5.68/5.02	0.04/0.01	0.7%/0.19%

(i) Sort

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	2.5/2.51	2.51/2.52	0.01/0.01	0.4%/0.39%
4	1.45/1.36	1.47/1.36	0.02/0.005	1.37%/0.36%
8	0.905/0.89	0.908/0.892	0.003/0.002	0.33%/0.22%
16	0.686/0.723	0.69/0.728	0.004/0.005	0.58%/0.69%
32	0.635/0.71	0.64/0.719	0.005/0.009	0.78%/1.26%
48	0.78/1.43	0.785/1.44	0.005/0.01	0.64%/0.69%

(j) Strassen

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	0.353/0.353	0.354/0.354	0.001/0.001	0.28%/0.28%
4	0.205/0.207	0.207/0.208	0.002/0.001	0.97%/0.48%
8	1.4/0.132	1.4/0.133	0/0.001	0%/0.75%
16	0.121/0.124	0.122/0.125	0.001/0.001	0.828%/0.81%
32	0.201/0.23	0.205/0.234	0.004/0.004	1.9%/1.73%
48	0.345/0.44	0.35/0.445	0.005/0.003	1.4%/0.68%

threads were available at that instance of time. Table 3 records the timing measurements in seconds when Task-ID=2 with different values of N. The different task instance timings that include creation, execution (not including suspension), waiting before execution, and exiting were not affected by the input size, which is normal due to the fact that these events cannot be interrupted by another thread or task instance once they start. The main variation was found in the suspension time, which is due to the fact that the number of child tasks is positively proportional to the input size in a 2^N relationship. The parent task-id, which is 2 in Table 3, has to wait for all its child tasks before it can resume execution. The suspension time measurements, indicated by our tool, grow with the number of child tasks in the same relationship 2^N. These measurements present the efficiency and necessity of using our tool to get precise profiling information about the different OpenMP task applications.

Tasking collector API, proposed in this paper, are crucial to validate the various OpenMP task scheduling algorithms. OpenMP runtime library developers can consult our proposal to find the best approach in which a task should start execution and the thread that should be assigned to it. Two main optimizations (load balancing and data locality) related to task scheduling can be tested using our proposal. Load balancing should be taken care of when scheduling OpenMP tasks. Assigning tasks to the working threads in an equivalent manner is mandatory for any task scheduling algorithm. Our tool shows the thread associated with each task instance during all the different phases of the task execution. On the other hand, data locality is another concern for any scheduling algorithm. Tasks operating on the same data should be scheduled for execution on the same thread to improve data reuse, especially on non-uniform memory access (NUMA) architectures. The task tree constructed by our tool can indicate the data that was assigned to each task by mapping this tree back to the source code application.

Table 3. Fibonacci code timing measurements for Task-ID=2

Input	#Childs	Creation	Pool-waiting	Execution	Suspension	Exiting
2	0	0.0001	0.0001	0.0001	0.0	0.0001
4	2	0.0001	0.0001	0.0001	0.0002	0.0001
8	33	0.0001	0.0001	0.0001	0.0011	0.0001
16	1596	0.0001	0.0001	0.0001	0.3100	0.0001
32	3524577	0.0001	0.0001	0.0001	970	0.0001

5 Related Work

Profiler for OpenMP (POMP) [15] was the first profiling mechanism for OpenMP runtime. It enables performance tools to detect OpenMP events by specifying the names and properties of some instrumentation calls, including the invocation position and time associated with each event. The POMP adheres to the abstract OpenMP execution model and is independent of a compiler and an OpenMP runtime library. OpenMP Pragma and Region Instrumentor (OPARI) [15] is a portable source-to-source translation tool that inserts the POMP instrumentation calls in Fortran, C, and C++ programs. However, these instrumentation calls can notably affect the compiler optimizations and hence might not capture the true picture of an OpenMP program. The OPARI has been broadly used for OpenMP instrumentation in different performance tools such as TAU [18], KOJAK [16], and Scalasca [7]. Vampir [10] is another tool, which provides thread-specific measurements that can include the OpenMP static and runtime context. Another version of the POMP [14] was proposed as an attempt to standardize the OpenMP monitoring interface. However, this version was rejected by the OpenMP ARB because of its complexity and its implementation cost.

The work proposed by the Sun Microsystems [9] describes the OpenMP Runtime API (ORA) for profiling OpenMP applications. The ORA was accepted by the OpenMP Architecture Review Board (ARB). The ORA provides a framework to the performance collector tools to collect necessary information. This information is needed to enhance the performance of OpenMP programs. The OpenUH

research compiler group has developed an open source implementation [1], [8] for the ORA in the OpenUH runtime library. Another paper by Lin [11] presents a data model that captures the runtime behavior of OpenMP applications with tasks constructs. However, the work only captures the abstraction-level (construct-level) information of the OpenMP tasking constructs.

In order to measure the performance of task instances, Lorenz et al. [12] describe a portable method to distinguish individual task instances and track their suspension and resumption events using instrumentation calls implemented as an extension of OPARI. Lorenz et al. [13] also present an implementation within the Score-P performance measurement system to overcome the performance issues related to task profiling. Furlinger and Skinner [6] describe the support for task profiling using instrumentation in the ompP tool [5].

6 Conclusions and Future Work

In this work, we have presented our experiences in implementing a new API for OpenMP task profiling. The OpenMP Runtime API for profiling (ORA) was approved by the OpenMP tool committee to create a standardized tool interface for OpenMP programs. We have extended the ORA to support profiling for OpenMP tasks at the micro level. We have implemented our extensions using the OpenUH open-source compiler. Our extensions to the ORA allow the execution and scheduling of tied and untied OpenMP tasks to be tracked by a tool to collect performance measurements. These measurements assist OpenMP application developers to gain more insight into the dynamic behavior of OpenMP based applications. Our extensions adhere to the proposal recently suggested by the OpenMP tool committee for task profiling. Moreover, Our experimental results show that the overheads associated with our implementation are negligible. Finally, C/C++ and Fortran programs are supported by our implementation.

Our next step is to integrate our implementation with TAU, a powerful performance tool, to visualize the ORA measurements. We also plan to extend the ORA to support taskgroup and work-sharing constructs in order to make the ORA more powerful and comprehensive. We also plan to use the task related dynamic information, extracted through the ORA, for task-related optimizations using a feedback framework.

Acknowledgments. The authors would like to thank their colleagues in the HPCTools group, especially Deepak Eachempati, for their extensive collaboration to make this work a reality. This work is supported by the National Science Foundation under grant CCF-1148052. Development at the University of Houston was supported in part by the NSFs Computer Systems Research program under Award No. CRI-0958464.

References

1. Bui, V., Hernandez, O., Chapman, B., Kufrin, R., Tafti, D., Gopalkrishnan, P.: Towards an implementation of the OpenMP collector API. Urbana 51, 61801 (2007)

2. Chapman, B., Eachempati, D., Hernandez, O.: Experiences developing the OpenUH compiler and runtime infrastructure. International Journal of Parallel Programming, 1–30 (2012)
3. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE (2009)
4. Eichenberger, A., Mellor-Crummey, J., Schulz, M., Copty, N., DelSignore, J., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An openMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013)
5. Fürlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. OpenMP Shared Memory Parallel Programming, 15–23 (2008)
6. Fürlinger, K., Skinner, D.: Performance profiling for openMP tasks. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 132–139. Springer, Heidelberg (2009)
7. Geimer, M., Wolf, F., Wylie, B.J., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
8. Hernandez, O., Nanjegowda, R.C., Chapman, B., Bui, V., Kufrin, R.: Open source software support for the OpenMP runtime API for profiling. In: International Conference on Parallel Processing Workshops, ICPPW 2009, pp. 130–137. IEEE (2009)
9. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling. OpenMP ARB as an official ARB White Paper 314, 181–190 (2007), `http://www.compunity.org/futures/omp-api.html`
10. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool-set. In: Tools for High Performance Computing, pp. 139–155. Springer (2008)
11. Lin, Y., Mazurov, O.: Providing observability for openMP 3.0 applications. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 104–117. Springer, Heidelberg (2009)
12. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to reconcile event-based performance analysis with tasking in openMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 109–121. Springer, Heidelberg (2010)
13. Lorenz, D., Philippen, P., Schmidl, D., Wolf, F.: Profiling of OpenMP tasks with score-p. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW), pp. 444–453. IEEE (2012)
14. Mohr, B., Malony, A.D., Hoppe, H.-C., Schlimbach, F., Haab, G., Hoeflinger, J., Shah, S.: A performance monitoring interface for OpenMP. In: Proceedings of the Fourth Workshop on OpenMP, EWOMP 2002 (2002)
15. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. The Journal of Supercomputing 23(1), 105–128 (2001)
16. Mohr, B., Wolf, F.: KOJAK–a tool set for automatic performance analysis of parallel programs. In: Euro-Par 2003 Parallel Processing, pp. 1301–1304 (2003)
17. Qawasmeh, A., Chapman, B., Banerjee, A.: A compiler-based tool for array analysis in HPC applications. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW), pp. 454–463. IEEE (2012)
18. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)