Task Profiling through OpenMP Runtime API and Tool Support

Ahmad Qawasmeh, Abid Malik, Deepak Eachempati, Barbara Chapman

{arqawasm,ammalik3,dreachem,chapman}@cs.uh.edu, Dept. of Computer Science, University of Houston

Introduction

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) approved an interface specification known as the "OpenMP Runtime API (ORA) for Profiling" to enable performance tools to collect performance data for OpenMP programs.

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. 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.

Additionally, we integrate our ORA into the performance tool TAU to visualize task measurements at the construct level. The proposed extensions are in line with the newest specification recently proposed by the OpenMP tools committee for task profiling.

OpenMP Task Implementation in OpenUH

❖ The OpenUH compiler supports OpenMP on the IA-64, IA-32, x86_64, and Opteron platforms. OpenMP is a standard directive-based API for shared memory programming. The OpenMP 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 task 2) Untied tasks**. Tied tasks can be suspended at specific scheduling points, while untied tasks can be suspended at any point in OpenMP programs. Moreover, a tied task is linked only to one thread, while an untied task can be resumed by any thread in the team.

```
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;
}
```

Figure 1: Fibonacci code

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

❖ We use the popular Fibonacci code in Figure 1 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. 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. The OpenMP tasking directives in Figure 1 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 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.

OpenMP Task Profiling

❖ The ORA 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 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. This communication can be depicted in Figure 3.

❖ The ORA interface 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.

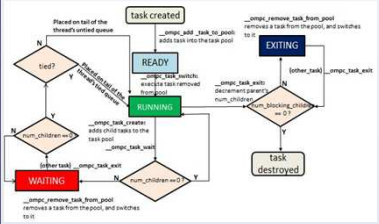

Fig. 2: OpenUH tasking execution model

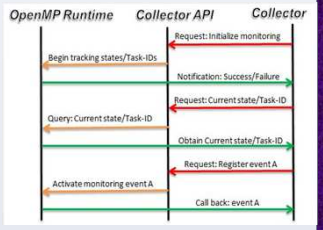

Fig. 3: Interaction between collector and Runtime

❖ Our ORA extensions for task profiling can be categorized as follows:

1. **Task Creation Events and States**
2. **Task Suspension Events and States**
3. **Task Execution/Exiting Events and States**
4. **Task IDs and Parent Task IDs**

❖ The task execution model implemented in the OpenUH runtime to support OpenMP tasks is shown in Figure 2. This 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.

Evaluation

❖ In order to evaluate our implementation in the OpenUH compiler, we performed the following analyses;

1. We measured the overheads introduced by the inclusion of our implementation in the runtime. The absolute overhead percentage ranges from 0% to 5% of the execution time. The average overhead percentage obtained for ten different BOTS kernels is less than 1%.

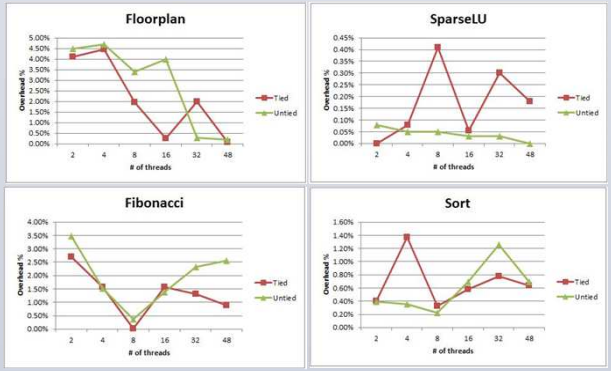

Fig. 4: Overhead comparison (Tied vs. Untied) BOTS kernels

2. We tracked the newly developed task IDs, states, and events via our employed requests at the task-instance level. We developed a prototype OpenMP task profiler for that.

Table 2: Fibonacci code time measurements for Task-ID=2 (sec)

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

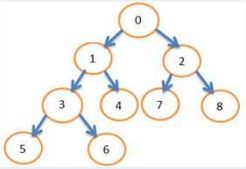

Fig. 5: A possible task tree (n=4)

3. We integrated our ORA into the performance tool TAU. We aggregated the task instances called from the same calling context to visualize task measurements at the construct level.

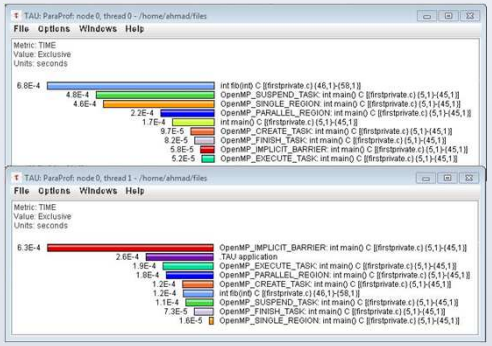

Fig. 6: Thread comparison using TAU for the Fibonacci code with task events capturing

❖ We used the Barcelona OpenMP Task Suite (BOTS) kernels 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).

Conclusion

❖ We have presented our experiences in implementing a new API for OpenMP task profiling.

❖ We have extended the ORA in the open-source OpenUH compiler to support profiling for OpenMP tied and untied tasks at both task-instance level and construct level.

❖ Our extensions to the ORA allow the execution and scheduling of OpenMP tasks to be tracked by a tool to collect performance measurements.

❖ The obtained measurements assist application developers to gain more insight into the dynamic behavior of OpenMP based applications while producing negligible overheads.

❖ Visualization for task events capturing was achieved through the integration of ORA into TAU.

References

[1] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. OpenMP ARB as an official ARB White Paper available online at http://www.compunity.org/futures/omp-api.html, 314:181-190, 2007.

[2] O. Hernandez, V. Bui, R. Kufrin, R. Nanjegowda and B. Chapman. Open Source Software Support for the OpenMP Runtime API for Profiling. ICPPW'09, pp. 130-137. IEEE, 2009.

Acknowledgement: This work is supported by NSF 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.