Performance Profiling for OpenMP Tasks

Karl Fürlinger[1] and David Skinner[2]

[1] Computer Science Division, EECS Department
University of California at Berkeley
Soda Hall 593, Berkeley CA 94720, U.S.A.
fuerling@eecs.berkeley.edu
[2] Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley CA 94720, U.S.A.
deskinner@lbl.gov

Abstract. Tasking in OpenMP 3.0 allows irregular parallelism to be expressed much more easily and it is expected to be a major step towards the widespread adoption of OpenMP for multicore programming. We discuss the issues encountered in providing monitoring support for tasking in an existing OpenMP profiling tool with respect to instrumentation, measurement, and result presentation.

1 Introduction

The direct support for task parallelism in version 3.0 of the OpenMP standard is expected to be a major step towards the widespread adoption of OpenMP for shared memory multicore programming. Tasking allows irregular forms of parallelism to be expressed more easily and it will allow OpenMP to be employed in new application areas.

In this paper we discuss the issues we encountered in providing monitoring support for tasking in the `ompP` profiling tool with respect to instrumentation and measurement and result presentation. Since tasking results in more dynamic and unpredictable execution characteristics of OpenMP codes, we believe tool support will be more important for users that would like to understand how their code executes and what performance it achieves. As an example, the OpenMP v3.0 specification states that, when a thread encounters a task construct, "[it] may immediately execute the task, or defer its execution". To some application developers it will be important to know what decision the runtime took and `ompP`'s profiles offer this kind of information, among other things.

The rest of this paper is organized as follows: in Sect. 2 we give a short overview of the OpenMP profiling tool we have extended in this study to support tasking. In Sect. 3 we describe the extensions and modifications made, at the instrumentation, measurement, and result presentation stages. In Sect. 4 we discuss related work and in Sect. 5 we conclude and discuss areas for future work.

2 The OpenMP Profiler ompP

ompP is a profiling tool for OpenMP applications that does not rely on nor is limited to a particular OpenMP compiler and runtime system. ompP differs from other profiling tools like gprof or OProfile [6] in primarily two ways. First, ompP is a measurement-based profiler and does not use program counter sampling. The application with source code instrumentation invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). The direct measurement approach can potentially lead to higher overheads when events are generated very frequently, but this can be avoided by instrumenting such constructs selectively. An advantage of the direct approach is that the results are not subject to sampling inaccuracy and hence they can also be used for correctness testing in certain contexts.

The second difference lies in the way of data collection and representation. While general profilers work on the level of routines, ompP collects and displays performance data in the user model of the execution of OpenMP events [5]. For example, the data reported for critical sections contain not only the execution time but also list the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each threads spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC). An example profile for a critical section is given in Fig. 1

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID      execT      execC      bodyT     enterT      exitT
   0       1.00          1       1.00       0.00       0.00
   1       3.01          1       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       0.00
   3       4.01          1       1.00       3.01       0.00
 SUM      10.02          4       4.01       6.01       0.00
```

Fig. 1: Profiling data delivered by ompP for a critical section.

Profiling data in a similar style is also delivered for other OpenMP constructs, the columns (execution times and counts) depend on the particular construct. Furthermore, ompP supports the query of hardware performance counters through PAPI [3] and the measured counter values appear as additional columns in the profiles.

Profiling data are displayed by ompP both as flat profiles and as callgraph profiles, giving both inclusive and exclusive times in the latter case. The callgraph profiles are based on the callgraph that is recorded by ompP. An example callgraph is shown in Fig. 2. The callgraph is largely similar to the callgraphs given by other tools, such as callgrind [9], with the exception that the nodes are not only functions but also OpenMP constructs and user-defined regions, and the (runtime) nesting of those constructs is shown in the callgraph view. The

callgraph that ompP records represents the union of the callgraph of each thread. That is, each node reported has been executed by at least one thread.

```
    ROOT  [critical.i686.ompp: 4 threads]
  REGION  +-R00004 main.c (40-51) ('main')
PARALLEL     +-R00005 main.c (44-48)
  REGION        |-R00001 main.c (20-22) ('foo')
  REGION        |  +-R00002 main.c (27-32) ('bar')
CRITICAL        |     +-R00003 main.c (28-31) (unnamed)
  REGION        +-R00002 main.c (27-32) ('bar')
CRITICAL           +-R00003 main.c (28-31) (unnamed)
```

Fig. 2: Example callgraph view of ompP.

3 Supporting Tasks in ompP

The OpenMP 3.0 specification introduces two new constructs for tasking, `task` and `taskwait`. If a thread encounters a `task` construct, it packages up the code and data environment and creates the task to be executed in the future, potentially by a different thread. The `taskwait` construct is used to synchronize the execution of tasks. A thread can suspend the execution only at a task scheduling point (TSP). The same thread will pick up the execution of a task, unless the task is untied. In this case, any thread can resume the execution and no restriction on the location of TSPs in untied tasks exists.

3.1 Instrumentation

ompP relies on source code instrumentation using Opari [7] to add monitoring calls according to the POMP specification inside and around OpenMP constructs. We extended Opari to handle the `task` and `taskwait` constructs as described below.

For `task`, an instrumented piece of code looks similar to the pseudocode depicted in Fig. 3. I.e., `enter/exit` instrumentation calls are placed on the outside of the task construct and `begin/end` calls are placed as the first and last statements inside the tasking code, respectively.

If specified, the untied clause is detected and `POMP_Utask_*` calls are generated in this case. A simple enter/exit pair of instrumentation calls is added for the taskwait clause.

3.2 Measurement

ompP's measurement routines implement the `POMP_Task_*`, `POMP_Utask_*`, and `POMP_Taskwait_*` calls. An important observation is that during execution a

```
POMP_Task_enter(...)
#pragma omp task
{
POMP_Task_begin(...)
// user's task code
POMP_Task_end(...)
}
POMP_Task_exit(...)
```

(a) POMP instrumentation for the `task` construct.

```
POMP_Taskwait_enter(...)
#pragma omp taskwait
POMP_Taskwait_exit(...)
```

(b) POMP instrumentation for the `taskwait` construct.

Fig. 3: Instrumentation for tasking related constructs.

task construct is best represented by two separate entities: one for task creation and one for task execution. Following this idea, we create two `ompP` regions for each source code task construct, one of type `TASK` for task creation and one of type `TASKEXEC` to record profiling data related to task execution. In the terminology of the OpenMP specification, `TASK` corresponds to the task construct, while `TASKEXEC` corresponds to the task region. `UTASK` and `UTASKEXEC` are used for untied tasks.

```
void main(int argc, char* argv[]) {
#pragma omp parallel {
    int i;
#pragma omp single nowait {
      for( i=0; i<5; i++ ) {
#pragma omp task /* if(0) */ {
       mytask();
      }
    }
  }
}
```

```
void mytask() {
  sleep(1);
 }
```

Fig. 4: (Pseudo) source code with tasking.

Consider the simple code example in Fig. 4 and its corresponding callgraph as delivered by `ompP` in Fig. 5. Task creation occurs inside the single region while task get executed when threads hit the implicit barrier at the end of the parallel construct. If, alternatively, an `if(0)` clause is specified, tasks are

```
PARALLEL  +-R00001          |  PARALLEL  +-R00001
  SINGLE     |-R00002       |    SINGLE     +-R00002
    TASK     |  +-R00003    |      TASK        +-R00003
TASKEXEC     +-R00003       |  TASKEXEC           +-R00003
```

Fig. 5: Dynamic callgraph of the code shown in Fig. 4 (left). The right side shows the callgraph when an `if()` clause is present and evaluates to false.

executed immediately and this is visible in the callgraph, where the `TASKEXEC` is a child of the `TASK` node. [3]

Support for monitoring untied tasks is incomplete at this time. We chose to offer the user the option to disable any monitoring of untied tasks completely or to monitor them in the same way as tied tasks (assuming that the executing thread does not change during the lifetime of a task). Without a way to observe the suspension and resumption of tasks at general task scheduling points, this seems to be the best we can do.

3.3 Profiling Data Analysis and Presentation

Flat profiles and callgraph profiles are recorded for the `[U]TASK`, `[U]TASKEXEC`, and `TASKWAIT` regions. Fig. 6 shows the possible immediate dynamic nesting of these three region types and their interpretation. The location of the `[U]TASKEXEC` region in the callgraph allows the analysis of when tasks were executed dynamically. The execution might happen nested in the `[U]TASK` region if an `if()` clause evaluates to false. If tasks are executed at a `TASKWAIT` region, this will also be indicated by the dynamic nesting and if threads execute tasks while at the implicit exit barrier of a parallel or workshare construct, the `TASKEXEC` region will be shown as a child region of this parallel or workshare region.

One of `ompP`'s more advanced features is its overhead analysis. When threads execute a worksharing region with an imbalanced amount of work, the waiting time of threads in the implicit exit barrier of that worksharing construct is measured by `ompP` and reported as load imbalance overhead. A total of four overhead classes are defined: load imbalance; synchronization overhead; limited parallelism; and thread management. The reporting of the overhead relies on the fact that OpenMP threads do not perform useful work on behalf of the application in certain program phases (such as when entering a critical section or at implicit or explicit thread barriers).

This assumption is no longer valid with OpenMP 3.0 when tasking is used. When threads hit an implicit barrier, instead of idling they can do useful work by executing ready tasks. To account for this, we modified the overhead reporting

[3] All experiments reported in this paper have been performed on a Linux machine using a beta version of Intel's C/C++ compiler suite v11.0.044, which supports tasking. We suspect this implementation might be not be fully optimized but it was a sufficient as a vehicle to test the feasibility of our monitoring approach, as this paper is concerned about functionality and not performance.

	inner region		
outer region	`[U]TASK`	`TASKEXEC`	`TASKWAIT`
`[U]TASK`	–	×	–
`TASKEXEC`	×	×	×
`TASKWAIT`	–	×	–

```
+-....
|- outer
   |- inner
   +-....
```

Fig. 6: Possible nesting of the tasking related region types in `ompP`. A dash symbol (–) indicates a nesting that can not occur, × indicates valid nestings. The `[U]TASK`–`TASKEXEC` nesting signifies immediate execution either because the `if()` clause evaluates to false or runtime decided not to defer the execution for other reasons such as resource exhaustion.

of `ompP` by subtracting from the overheads the time spent executing tasks. The required timing data is available from the callgraph recorded by `ompP` (we know that a `[U]TASKEXEC` happens in the context of the implicit exit barrier) and from the callgraph profiles recorded for the task execution on a per-thread basis.

```
Overheads wrt. each individual parallel region:
      Total    Ovhds (%) =  Synch(%) + Imbal  (%) + Limpar (%) +  Mgmt (%)
R00001 6.00 1.00 (16.68) 0.00 (0.00) 1.00 (16.66)  0.00 (0.00) 0.00 (0.02)

Overheads wrt. whole program:
      Total    Ovhds (%) =  Synch(%) + Imbal  (%) + Limpar (%) +  Mgmt (%)
R00001 6.00 1.00 (15.64) 0.00 (0.00) 1.00 (15.63)  0.00 (0.00) 0.00 (0.02)
   SUM 6.00 1.00 (15.64) 0.00 (0.00) 1.00 (15.63)  0.00 (0.00) 0.00 (0.02)
```

Fig. 7: Overhead analysis report corresponding to the code shown in Fig. 4

An example of an overhead report that takes task execution into account is shown in Fig. 7. This overhead report corresponds to the code fragment shown in Fig. 4, the application executes with two threads and creates 5 tasks with an execution time of 1 second each. As shown, ompP correctly accounts for the task execution by reporting the imbalance overhead as 1.0 second due to the uneven distribution of tasks to threads.

To allow the application developers to analyze when tasks get executed further, we added a new timing category `taskT` to OpenMP parallel regions and worksharing regions. Fig. 8 shows the profile of a parallel region. While at the implicit exit barrier of the parallel construct, thread 0 spent 3.0 seconds executing tasks, while thread 1 spent 2.0 seconds, 1.0 second remains as the waiting time of thread 1, as shown in the `exitBarT` column.

```
R00001 main.c (15-26) PARALLEL
 TID    execT    execC     bodyT   exitBarT  startupT  shutdwnT     taskT
   0     3.00        1      0.00       0.00      0.00      0.00      3.00
   1     3.00        1      0.00       1.00      0.00      0.00      2.00
 SUM     6.00        2      0.00       1.00      0.00      0.00      5.00
```

Fig. 8: Flat region profile, showing the time threads spend executing tasks while waiting at the implicit exit barrier of the parallel region. This data corresponds to the code shown in Fig. 4 when executed with two threads.

4 Related Work

Opari and the POMP interface are the basis of OpenMP monitoring for several performance tools for scientific computing like TAU [8], KOJAK [10], and Scalasca [4]. To the best of our knowledge, there is currently no work under way to support tasking within these projects [2]. However, we believe that our work on extending Opari and the experience we gathered with respect to supporting tasking in the monitoring system will be of use for adding tasking support for these tools.

Sun has developed an extension to their proposed performance profiling API [5] for OpenMP and is supporting tasking in the new version of their performance tool suite. [1]. The nature of this interface and Sun's implementation are different from ompP's approach (callbacks and sampling vs. direct measurement)

5 Conclusion and Future Work

We have described our experiences in supporting tasking in a measurement based profiler for OpenMP. We have made additions to a source code instrumentation, measurement, and result presentation stages of the tool.

With respect to measurement, a fundamental difference arose in the way waiting time at implicit barriers was accounted for in the overhead analysis. The modified data reporting allows users to see which threads execute tasks at which point in the application. Due to the dynamic execution characteristics of OpenMP with tasking, we believe this capability is important both for performance considerations as well as a pedagogical tool for people learning to use OpenMP tasking.

We found that monitoring overhead directly correlates with the frequency of monitored events. With very frequent, short lived tasks overheads can be substantial. However, such an application is unlikely to scale or to perform well even without any monitoring. For reasonably sized tasks we found that monitoring overhead can be expected to be less than 5 percent of execution time.

For the future, work is planned in several directions. Clearly, how untied tasks are handled currently is unsatisfactory. However, without notifications of task switches form the runtime, the options for a source code instrumentation based

tool like ompP are very limited. The most promising solution for this issue seems to lie in an incorporation of the profiling API [5] for providing such a notification mechanism. The currently limited adoption of this API by vendors is a practical problem, however.

References

1. OpenMP 3.0: Ushering in a new era of parallelism. Birds of a Feather meeting at Supercomputing 2008.
2. Personal communication at supercomputing 2008.
3. Shirley Browne, Jack Dongarra, N. Garner, G. Ho, and Philip J. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.
4. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006)*, pages 303–312, Bonn, Germany, 2006.
5. Marty Itzkowitz, Oleg Mazurov, Nawal Copty, and Yuan Lin. An OpenMP runtime API for profiling. Accepted by the OpenMP ARB as an official ARB White Paper available online at `http://www.compunity.org/futures/omp-api.html`.
6. J. Levon. OProfile, A system-wide profiler for Linux systems. Homepage: `http://oprofile.sourceforge.net`.
7. Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, September 2001.
8. Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
9. Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.
10. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 13–22. IEEE Computer Society, February 2003.