Design and Implementation of OpenMP Tasks in the OMPi Compiler

Spiros N. Agathos, Panagiotis E. Hadjidoukas, Vassilios V. Dimakopoulos
Department of Computer Science
University of Ioannina
Ioannina, Greece, GR-45110
Email: {*sagathos,phadjido,dimako*}*@cs.uoi.gr*

***Abstract*—In this paper we present the design and implementation of tasks in the context of the OMPI OpenMP compiler. The modular architecture of OMPI's runtime system allows a wide range of choices for experimenting with OpenMP structures. We present two fully-fledged implementations of tasks: one based on POSIX threads, with the addition of a tasking layer, and another one based on an almost unmodified user-level threading library. Both allow the tuning of their scheduling parameters so as to optimize memory consumption and execution times, resulting in highly competitive performance.**

***Keywords*-task parallelism, OpenMP, threads, compiler transformations, runtime system**

I. Introduction

The tasking model has been used successfully in a wide range of parallel and distributed computing systems, programming languages, tools and applications. OpenMP, the dominant programming model for shared-memory platforms, has been recently extended in its version 3.0 with support for task-based parallelism [1]. With tasking, the expressiveness of OpenMP goes beyond loop-level and coarse-grain parallelism. OpenMP is enriched with flexible management or irregular and dynamic parallelism, in the spirit of Cilk [2].

An OpenMP task is a unit of work scheduled for asynchronous execution by an OpenMP thread. Each task has its own data environment and can share data with other tasks. Tasks are distinguished as *explicit* and *implicit* ones. The former are declared by the programmer with the `task` construct, while the latter are created by the runtime library, when a `parallel` construct is encountered; each OpenMP thread corresponds to a single implicit task. A task may generate new explicit tasks and can suspend its execution waiting for the completion of all its child tasks with the `taskwait` construct. Within a parallel region, task execution is also synchronized at barriers: OpenMP threads resume only when all of them have reached the barrier and there are no pending explicit tasks left. Tasks are also classified as *tied* and *untied*, depending on whether resumption of suspended tasks is allowed only on the OpenMP thread that suspended them or on any other thread.

The tasking model of OpenMP was initially presented in [3] and a prototype was implemented in the NANOS Mercurium compiler [4]. Currently, there exist several platforms that provide support of the OpenMP tasking model; these include both commercial (e.g. Intel, Sun, IBM), freeware, and research compilers, such as GNU GCC [5] and OpenUH [6].

OMPI [7] is a lightweight source-to-source compiler for OpenMP/C; it takes C source code with OpenMP pragmas and outputs transformed multithreaded C code augmented with calls to OMPI's runtime system. OMPI offers a unique modular runtime system which allows the integration of a multitude of thread libraries. It currently includes libraries based on POSIX threads, custom user-level threads and heavyweight processes. The latter allow transparent OpenMP application execution on clusters of multicore nodes, through a software DSM layer. Additionally, OMPI includes highly efficient support for nested parallelism.

In this paper we discuss the implementation of OpenMP tasks in the OMPI compiler. In contrast to existing OpenMP compilers, which follow a single and particular way of implementing the tasking model, OMPI utilizes two different and fully functional approaches that build on top of a common runtime infrastructure. The first approach is based on POSIX-threads while the second one exploits user-level multithreading. Several design and optimization issues that concern both cases are also discussed, while the experimental evaluation with benchmarks demonstrates the efficient support of task parallelism by the OMPI compiler.

The rest of the paper is organized as follows: Section II gives an overview of the OMPI compilation environment and presents the compiler support of OpenMP tasks. Sections III and IV discuss the two runtime designs. Experimental evaluation is reported in Section V. Finally, we conclude in Section VI.

II. Compiler Transformations

Consider the following general OpenMP task statement, which provides code to be executed as a task, requiring a snapshot of variables `x` and `y`. The `if` expression forces immediate execution if `cond` evaluates to false.

```
#pragma omp task if(cond) firstprivate(x,y)
{
  <CODE>
}
```

The code produced by the compiler is shown in Fig. 1. The original code gets replaced by lines 2–12. In essence, there is

```
1   /* Tranformed code */
2   {
3     struct taskenv { int x; int y; } *tenv;
4
5     tenv = ort_taskenv_alloc(sizeof(struct taskenv));
6     tenv->x = x;
7     tenv->y = y;
8     if (cond)
9       ort_new_task(0, taskFunc0, (void *) tenv);
10    else
11      ort_new_task_exec(0, taskFunc0, (void *) tenv);
12  }
13
14  /* Produced task function */
15  void taskFunc0(void *tdata)
16  {
17    struct taskenv { int x; int y; } *tenv = tdata;
18    int x = tenv->x;
19    int y = tenv->y;
20
21    <CODE>
22
23    ort_taskenv_free(tenv, sizeof(struct taskenv));
24  }
```

Figure 1. Source code transformation for tasks

a new structure declared that captures the data environment (firstprivate variables) that will be used by the task, and then the runtime system (see next section) is instructed to simply create a new task (`ort_new_task()`) or create it and execute it immediately (`ort_new_task_exec()`), depending on the given condition (`cond`). The actual task code is moved to a new function (`taskFunc0()`), shown in lines 15–31. The task function has exactly one argument, its data environment. Notice that the data environment is allocated at line 5, using `ort_taskenv_alloc()`, and the size (in bytes) of the environment. The data is deallocated at the end of the task function, through `ort_taskenv_free()`.

A. *Optimized Execution Path*

It should be clear from the above transformation that preparing a task for execution incurs the overheads of allocating the data environment, invoking a driver function (`ort_new_task()`), calling the actual function that includes the original code (`taskFunc0()`), copying the data environment and finally freeing it (in addition to runtime queues management and scheduling overheads presented in the next section). Such overheads become significant especially for very fine-grain tasks. While one cannot avoid them in the general case, OMPI alleviates most of them *in the case of immediate task execution*. This occurs when either the condition `cond` evaluates to false, or the runtime system has a sizable backlog of tasks waiting for execution, and *throttles* the creation of new tasks. In this case, the compiler produces a fast execution path, by copying the original code verbatim. The transformation is shown in Fig. 2; local variables are declared to capture the firstprivate ones and the original code is executed unmodified. The two runtime functions (`ort_task_immediate_start/end()`) are required for internal bookkeeping.

Notice that the optimization introduces code duplication,

```
if (!cond || ort_task_throttling())
{
  int _fy=y, y=_fy, _fx=x, x=_fx; /* Capture values */
  ort_task_immediate_start();
  <CODE>
  ort_task_immediate_end();
}
else
  <standard transformation of Fig. 1>
```

Figure 2. Optimized code for immediately executed tasks

thus increasing the size of the executable. For size-critical applications, OMPI provides a flag that disables the production of the fast execution path.

III. Runtime Organization

This section describes the design and implementation of tasking support in the runtime system of the OMPI OpenMP C compiler, based on the default POSIX threading library.

A. *Key Structures and Memory Management*

The most important data structure is the task 'node', which holds the task descriptor, i.e. all the information needed for the execution of a task. Every OpenMP thread owns a list where task nodes can be stored. This list is called task queue and task nodes that are stored there represent tasks whose execution is pending. Task queues are implemented as special double-ended queues (deques); the owner thread can add and remove elements from the top of the queue, while any other thread can only extract elements from the bottom of the queue. In addition, OMPI makes use of per-thread recycle bins which store task node structures that have been deallocated. This allows task nodes to be reused, leading to better memory utilization and faster allocation.

When preparing for a new task, the need for memory allocation is two-fold: (a) allocate a new task node and (b) allocate space for the task data environment. The latter refers to memory allocated by the compiler-produced code (see Fig. 1). In contrast to the task node structure, which has a fixed size, the size of task data environment varies, depending on the size of the captured firstprivate variables of each task. However, the different memory sizes needed for task data environments are few in number, and equal to the number of lexical task regions in the program. Thus, the memory allocator is required to handle only a few different memory sizes. As a result, the allocator used in `ort_taskenv_alloc()` (see Fig. 1) implements a list of buckets; each bucket stores memory regions of a particular size. Given the size of the task data environment (say s), the allocator searches the list until it hits the bucket responsible for size s. If such a bucket does not exist, it gets created and a certain number of memory regions are allocated and added to the bucket. Finally, a memory region out of the bucket is returned. Upon deallocation (`ort_taskenv_free()`), the memory region is returned to the bucket it was removed from, making it possible to recycle memory regions for subsequent tasks.

B. Task Scheduling and Work-Stealing

OMPI's task scheduler is based on work stealing [2], whereby idle threads try to execute tasks created by other threads. After a new task is created, it is placed in a thread's queue until some thread decides to execute it. Task queues have fixed length, which means that they can store up to a certain number of pending tasks. This number is one of OMPI's runtime parameters, controlled through an environment variable (`OMPI_TASKQ_SIZE`). The manipulation of task queues is based on a highly efficient lock-free algorithm by Herlihy & Shavit [8]. Because the original algorithm implements an unbounded deque, we have modified it in order to handle our fixed-length task queues.

When a thread is about to execute its implicit task (parallel region), a new task node is allocated, the code of this task is executed immediately and finally the task node is deallocated. Whenever a thread reaches an explicit `task` construct, it can either allocate a new task node and submit the corresponding task for deferred execution, or it can suspend the execution of the current task and execute the new task immediately; OMPI's default behavior is to choose the former. That is, it implements a *breadth-first* task scheduling policy (BFS). It resorts to the second alternative (*depth-first* task execution) when the task queue is full. In that case the thread enters *throttling* mode, where every encountered task is executed immediately to completion. Notice that in this case the current task (although temporarily suspended in favor of the new task) does not enter the task queue, so it can never be resumed by another thread. In effect, all tasks are *tied*. Throttling mode is disabled when 30% of the task queue capacity becomes available.

The execution of pending explicit tasks takes place in two different cases. The first occurs when the thread reaches a `taskwait` construct, where it executes all the pending tasks it has created. In the second case, a thread will in addition try to execute pending tasks of every sibling in its team of threads. This occurs when a thread reaches a barrier. In both cases a thread repeatedly checks its queue for pending tasks; if there is one, it executes it. Otherwise, it tries to steal and execute a task from a sibling's task queue.

C. Fast Execution Path

In OpenMP V3.0, every task is required to keep a private copy of certain user-modifiable variables known as internal control variables (ICVs). When a thread is in throttling mode, as described above, it executes any encountered tasks immediately. Normally, because of the ICVs storage requirements, the newly executed task will suffer all the overheads of task node allocation and deallocation.

As an optimization, when in throttling mode, OMPI follows a lazy policy, whereby no task node is allocated until actually needed. Tasks are executed without a descriptor, as long as there is no access to the ICVs; a new task node will be allocated upon the first modification of the task's ICVs. The optimization is enabled by the code generated by the compiler shown in Fig. 2. The `ort_task_immediate_start()` call is needed to allow the suspended task remember that it is about to start a new task which has no allocated task node.

IV. Tasking with the PSThreads Library

In addition to the default threading layer, which is based on POSIX threads, OMPI's runtime system can be built on top of PSThreads, a package which provides lightweight non-preemptive user-level (UL) threads, which are used to implement OpenMP threads. These threads are executed on top of kernel-level POSIX threads which select the next-to-run UL thread from a set of ready queues, where threads are submitted for execution. One of the most significant advantages that PSThreads offers to the OMPI runtime environment is the efficient and low-overhead support of deep nested parallelism [9].

When PSThreads are employed, both implicit and explicit tasks are also instantiated with UL threads, submitted for execution in the ready queues and executed by the virtual processors of the library. Therefore, the terms tasks and PSThreads are practically equivalent and can be used interchangeably. In essence, all tasks are created as untied. To force tied execution, a task needs to know the id number of the thread it was initiated on and reuse it every time it gets rescheduled. As in the native implementation of tasks, a unique task node is associated with each task. The task node is equipped with a counter for the number of pending child tasks, along with a condition variable that suspends the parent task upon encountering a `taskwait` construct till all child tasks complete. Upon completion, child tasks decrease this counter, while the last one signals the condition variable, allowing the parent task to get rescheduled.

For the runtime management of explicit tasks, we utilize two counters and condition variables within every OpenMP thread team. The counters track the number of active (pending) and running tasks of the team. The first condition variable is used to limit the number of running tasks to that of the team size: a spawned or rescheduled task will wait until the number of running threads does not exceed that of the OpenMP threads in the team. A finished task first decreases the counter and then signals the condition variable allowing the next user-level thread to run. The second condition variable is used within an OpenMP barrier: an implicit task will wait until the number of all pending tasks of the team reaches zero.

At barrier entry, a frequent case is that all but a single OpenMP thread find the pending-tasks counter zero and proceed to the actual OpenMP barrier call, suspending their execution. That thread will create tasks and then wait for their completion at the condition variable (similarly to a `taskwait` construct). Task execution is independent of the suspended OpenMP threads as it is performed by the underlying virtual processors. At barrier exit, implicit tasks are given a higher execution priority with respect to possibly created explicit ones, eliminating thus any contention for thread number assignment.

The scheduling of OpenMP tasks follows a breadth-first approach and has been designed over the queue architecture and the pre-existing work stealing mechanism of the PSThreads library. Spawned user-level threads that correspond to explicit tasks are inserted at the front of the queue that belongs to the virtual processor the parent task is currently running on. To favor data locality further, a resumed thread (task) is always inserted in the queue of the virtual processor it was previously scheduled on.

Throttling of tasks is performed on a team basis, using the counter of pending tasks that belong to a specific team. In particular, a special flag is set whenever the number of pending tasks exceeds a specific threshold. From that point, task creation switches to direct task execution for the particular parallel region. Task throttling is disabled as soon as the number of pending tasks becomes equal to a low-level threshold. Both thresholds are set as multiples of the team size. The default high-level threshold is equal to 32*(team size), while and low-level threshold is always equal to the team size. Task throttling is also activated after a particular task-depth (set to 64 by default).

V. Performance Experiments

In this section we present experimental results on a server with 4 dual-core Intel Xeon Paxville 3.0GHz CPUs running Debian Linux 2.6. We have used the BOTS benchmark suite [10], which has been developed for testing and evaluating OpenMP task implementations. We present here results for four BOTS applications: Alignment, FFT, Sort and NQueens. Alignment aligns all protein sequences from a big set against every other sequence using the Myers and Miller algorithm. The alignments are scored and the best score for each pair is provided as a result. FFT computes the one-dimensional Fast Fourier Transform of a vector of complex values using the Cooley-Tukey algorithm. Sort sorts a random permutation of a sequence of 32-bit numbers with a fast parallel sorting variation of the ordinary merge sort. Finally, NQueens computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an $n \times n$ chessboard such that none of the queens attack any other. It uses a backtracking search algorithm with pruning.

Besides OMPI with the two threading libraries (POSIX and PSThreads), we provide results for three additional freely available OpenMP compilers that support tasks, the Intel C++ 12.0 compiler (ICC), Sun Studio 12.2 (SUNCC) for Linux, and GNU GCC 4.4.5. We have used the default settings of the OpenMP runtime libraries and the $-O3$ optimization flag in all experiments.

Regarding OMPI's POSIX library, task queue sizes were set to 24; if a thread tries to store more tasks in its queue it switches to throttling mode. In addition, throttling is disabled when a thread's queue has more than 24*30%=7 available nodes. When PSThreads are used, the stack size of user-level threads is set to a value large enough (4MB) to avoid stack corruption in particular benchmarks. The library has been configured to perform user-level context switching with setjmp/longjmp calls and synchronization with POSIX spin-locks. For the particular experiments, throttling is enabled when 32*8=256 tasks are pending and switches off when 8 of them remain active.

Table I presents the sequential execution (ignoring OpenMP pragmas) time in seconds of the four BOTS benchmarks with their default runtime parameters. Because we use GNU GCC as the native back-end compiler of OMPI, we use the sequential execution times of GCC as a baseline

Table I
Serial execution time of benchmarks

Compiler	Alignment	FFT	Sort	Nqueens
ICC	55.94	41.94	8.07	9.71
SUNCC	66.44	43.14	7.96	10.51
GCC	55.44	46.73	7.72	9.30

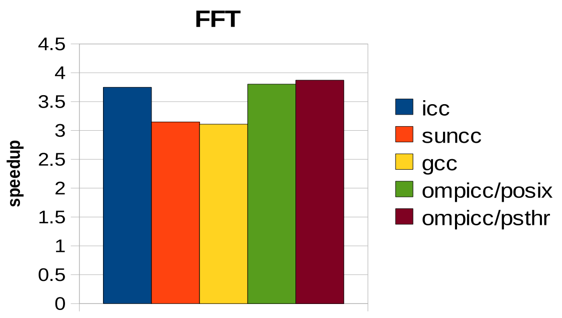

Figure 3. FFT (matrix size=32M)

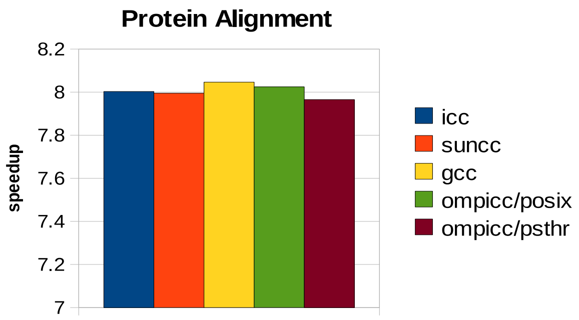

Figure 4. Protein Alignment (100 protein sequences))

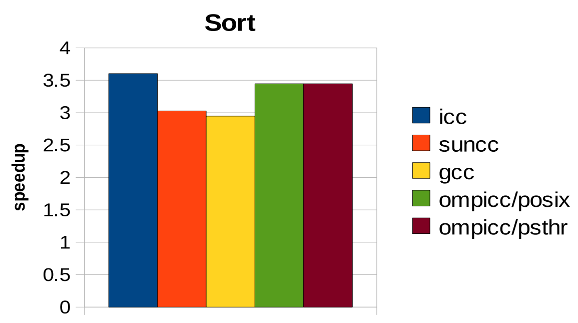

Figure 5. Sort (matrix size=32M)

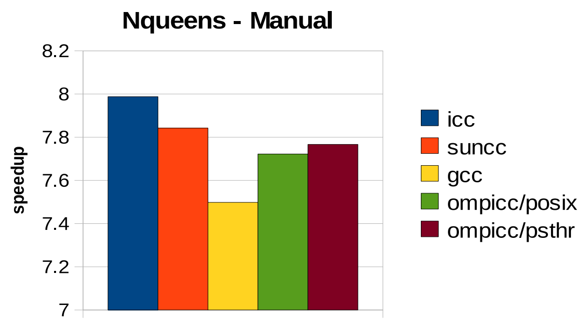

Figure 6. NQueens with manual cut-off (board size=14)

for caclulating the speedups for OMPI. With the exception of SUNCC and the protein Alignment benchmark, the compilers exhibit similar execution times.

Figs. 3 to 6 present the speedups of the four BOTS benchmarks when running on the 8 cores of our server. Each benchmark was run several times and the average execution time was calculated. For the FFT benchmark (32M matrix size) we observe that for OMPI both threading libraries achieve higher speedup as compared to all other compilers. For Alignment (100 protein sequences) all the compilers exhibit comparable performance. As far as the Sort benchmark (32 matrix size) is concerned, OMPI outperforms GCC and SUNCC, but here ICC obtains the best speedup. Fig 6 shows the experimental results of the NQueens benchmark (14 board size) when manual cut-off of task parallelism is employed by the application itself. For this particular experiment, ICC and SUNCC exhibit slightly better performance than OMPI, which outperforms GCC.

VI. Conclusions

We presented the implementation of the OpenMP V3.0 tasking model in the OMPI compiler. Source code transformations and optimizations performed by the compiler and two alternative approaches for runtime support of tasks were discussed. The experimental assessment with benchmarks demonstrated the efficiency of OMPI, which attains highly competitive performance, sometimes surpassing that of commercial OpenMP compilers.

We are currently optimizing the performance of the runtime system. We focus on alternative work stealing strategies that try to exploit the platform's topology in order to favor locality. OMPI is an open-source project, available at `http://www.cs.uoi.gr/~ompi`.

Acknowledgments

The authors would like to express their gratitude to Intel Corporation for donating the multicore server used to perform the experiments. This work was supported in part by the European Commission through the Artemisia SMECY project (grant 100230). S.N. Agathos was supported by the Greek State Scholarships Foundation (IKY).

References

[1] OpenMP ARB, "OpenMP Application Program Interface V3.0," May 2008.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A Proposal for Task Parallelism in OpenMP," in *Proc. 3rd IWOMP, Int'l Workshop on OpenMP*, Beijing, China, 2007, pp. 1–12.

[4] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, "An Experimental Evaluation of the New OpenMP Tasking Model," in *Proc. of the 20th Int'l Workshop on Languages and Compilers for Parallel Computing*, vol. LNCS 5234, Oct. 2007, pp. 63–77.

[5] Free Software Foundation, "GCC, the GNU compiler collection." [Online]. Available: http://www.gnu.org/software/gcc

[6] C. Addison and J. LaGrone and L. Huang and B. Chapman, "OpenMP 3.0 tasking implementation in OpenUH," in *Open64 Workshop in Conjunction with the Int'l Symposium on Code Generation and Optimization*, Seattle, USA, Mar. 2009.

[7] G. Philos, V. Dimakopoulos, and P. Hadjidoukas, "A Runtime Architecture for Ubiquitous Support of OpenMP," in *Proc. ISPDC 2008, 7th Int'l Symposium on Parallel and Distributed Computing*, Krakow, Poland, jun 2008, pp. 189–196.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan-Kaufmann, 2008, pp. 386–390.

[9] P. E. Hadjidoukas and V. V. Dimakopoulos, "Nested Parallelism in the OMPi OpenMP/C Compiler," in *Euro-Par 2007, 13th Int'l Euro-Par Conference on Parallel Processing*. Rennes, France: Springer LNCS 4641, Aug. 2007, pp. 662–671.

[10] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *38th Int'l Conference on Parallel Processing (ICPP '09)*, Vienna, Austria, Sept. 2009, pp. 124–131.