OpenMP Task Scheduling Analysis via OpenMP Runtime API and Tool Visualization

Ahmad Qawasmeh
Department of Computer Science
University of Houston
Houston, Texas
Email: arqawasm@cs.uh.edu

Abid Malik
Department of Computer Science
University of Houston
Houston, Texas
Email: ammalik3@cs.uh.edu

Barbara Chapman
Department of Computer Science
University of Houston
Houston, Texas
Email: chapman@cs.uh.edu

***Abstract*—OpenMP tasks propose a new dimension of concurrency to cap irregular parallelism within applications. The addition of OpenMP tasks allows programmers to express concurrency at a high level of abstraction and makes the OpenMP runtime responsible about the burden of scheduling parallel execution. The ability to observe the performance of OpenMP task scheduling strategies portably across shared memory platforms has been a challenge due to the lack of performance interface standards in the runtime layer. In this paper, we exploit our proposed tasking extensions to the OpenMP Runtime API (ORA), Known as Collector APIs, for profiling task level parallelism. We describe the integration of these Collector APIs, implemented in the OpenUH compiler, into the TAU performance system. Our proposed task extensions are in line with the new interface specification called OMPT, which is currently under evaluation by the OpenMP community. We use this integration to analyze various OpenMP task scheduling strategies implemented in OpenUH. The capabilities of these scheduling strategies are evaluated with respect to exploiting data locality, maintaining load balance, and minimizing overhead costs. We present a comprehensive performance study of diverse OpenMP benchmarks, from the Barcelona OpenMP Test Suite, comparing different task pools (DEFAULT, SIMPLE, SIMPLE_2LEVEL, PUBLIC_PRIVATE), task queues (DEQUE, FIFO, CFIFO, LIFO, INV_DEQUE), and task queue storages (ARRAY, DYN_ARRAY, LIST, LOCKLESS) on an AMD Opteron multicore system (48 cores total). Our results show that the benchmarks with similar characteristics exhibit the same behavior in terms of the performance of the applied scheduling strategies. Moreover, the task pool configuration, which controls the organization of task queues, was found to have the highest impact on performance.**

***Keywords*-OpenMP; Task scheduling; Collector APIs; OpenMP tools;**

I. Introduction

OpenMP is a standard API for shared memory programming. OpenMP provides a robust directive-based programming approach to write multithreaded applications for shared memory platforms and generate parallel versions of C/C++ and fortran programs. The introduction of tasks in OpenMP allows applications exhibiting irregular parallelism in the form of recursive algorithms and pointer based data structures to be parallelized in a directive-based manner. The **task** construct allows the developer to dynamically create asynchronous units of work to be scheduled by the runtime, while task synchronization is provided by additional tasking constructs.

An OpenMP task scheduler can be distinguished based on its efficiency of handling tasks. A task scheduler should meet some challenges associated with parallelizing OpenMP task programs. Task schedulers vary in terms of the implementation of queue organizations used to arrange tasks, work-stealing capability, queue contention, and how tasks are synchronized and scheduled among threads.

Two crucial issues, related to tasks, should be managed by any OpenMP task scheduler; **1) Data locality 2) Load balancing**. Modern processors are cache-based machines. Caches are based on the principles of temporal and spatial locality; Hence, a task scheduler should take advantage of many applications' tendency to reuse blocks of data that have been used (temporal locality) and to also access data items near those already used (spatial locality). OpenMP tasks, accessing the same data blocks, might be assigned to the same thread using a greedy approach; However, this approach might result in an incapable work-stealing. Equitable distribution of work among threads is a necessity to prevent processors from idleness. However, this equal distribution can be hard to achieve in many cases. Synchronization among threads is one factor, which leads to load imbalance. Load balancing operations are applicable, but might produce overhead costs due to high-latency remote memory accesses between processors.

As a consequence, a comprehensive performance analysis of OpenMP task scheduling is essential to understand the aforementioned conflicting goals and to find a suitable scheduling strategy for any OpenMP application. This analysis requires an accurate profiling mechanism to obtain information about applications, task scheduling configurations, and an increasingly complex memory hierarchy, including shared caches and non-uniform memory access (NUMA) characteristics. Unfortunately, the lack of standards in the runtime layer has hampered the development of third-party tools to support OpenMP development and analysis.

The OpenMP Runtime API (ORA) for profiling OpenMP applications was presented in [1] as a potential candidate to become a standard OpenMP interface before the introduction of tasks. The ORA was sanctioned by the tools committee of the OpenMP Architecture Review Board (ARB) as a white paper. 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 was adopted and implemented in OpenUH [2], which is an open-source compiler. We recently presented new extensions [3], [4] to the ORA to support tasks. These extensions include task creation, scheduling, suspension, stealing, execution and completion at the task-instance level.

This paper presents a powerful performance framework to analyze various task scheduling strategies and provides accurate measurements through the following contributions:

1) **Finalizing the new Collector API extensions to support OpenMP tasks and integrating them into the TAU performance system**. In order to accomplish this integration, the task instances called from the same calling context were aggregated. This aggregation allows us to visualize the new task Collector APIs events for the working threads at the task construct-level.
2) **Defining the hardware performance counters that should be tracked and linking them to TAU**. Data locality for OpenMP tasks can best be expressed by exploring the cache behavior. We focused on L2 cache misses and accesses for each Collector API event that was captured. We were able to obtain the L2 cache miss rate of each individual task event to demonstrate how any particular event is affected by varying the task scheduling strategy.
3) **A detailed performance study on a multi-socket multicore AMD Opteron system**. Seven Collector API events related to tasks are captured. More than one hundred combinations of a task pool configuration, task queue, task storage, number of tasks that can be stolen, and size of task queue, implemented in OpenUH, are applied to ten BOTS benchmarks. Three sets of threads are used with each combination against each benchmark.

This performance study was our key contribution to analyze the different task scheduling strategies implemented in OpenUH. Through this detailed analysis, we illustrate the impact of using any particular task scheduler on the behavior of different OpenMP benchmarks. We explain how the characteristics of any given benchmark indicate the task scheduling strategy that should be used to improve the performance.

To the best of our knowledge, the work reported here is the first open-source implementation of the ORA with extensions to support tasks at both instance and construct levels with tool support. The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 briefly describes the different OpenMP task scheduling strategies in the runtime of OpenUH, Collector APIs and their integration into TAU, and the benchmarks used. Section 4 presents experiments evaluating the different scheduling strategies on ten BOTS benchmarks. Finally, Section 5 concludes our contributions and discusses directions for the future work.

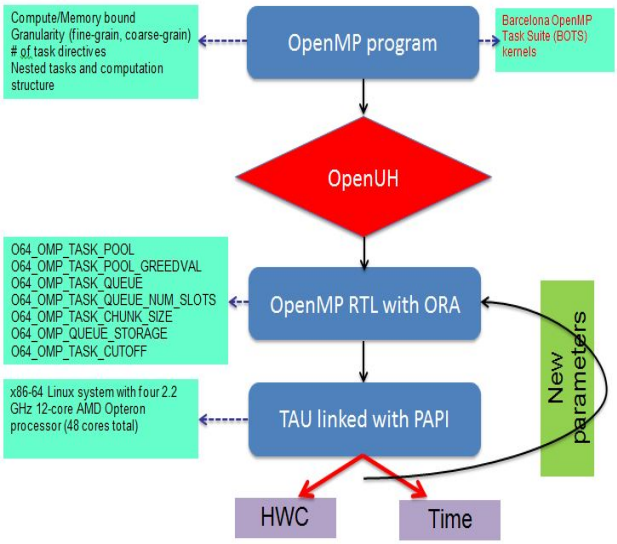

Fig. 1: Performance framework layers and analysis flow

II. Related Work

Profiler for OpenMP (POMP) [5] was the first effort to define a tool interface for OpenMP performance observation. It enables performance tools to detect OpenMP events by specifying the names and properties of some instrumentation calls. 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) [5] is a portable source-to-source translation tool that supports the POMP interface. The inclusion of instrumentation calls can notably affect the compiler optimizations and hence might not capture the true picture of either an OpenMP program or a task scheduling strategy. The OPARI has been broadly used for OpenMP instrumentation in different performance tools such as TAU [6], KOJAK [7], and Scalasca [8]. Vampir [9] is another tool, which provides thread-specific measurements that can include the OpenMP static and runtime context. On the other hand, thread-specific measurement of an OpenMP application can be performed by other performance tools, targeting one particular compiler, such as the Intel Thread Profiler [10]. Paraver [11] is another tool which provides detailed quantitative analysis of an OpenMP program performance depending on traces. However, it is untied to any specific programming model.

The work proposed by the Sun Microsystems [1] 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 [12], [13] for the ORA in the OpenUH runtime library before the introduction of tasks.

Recently, a new OpenMP API standard for tools [14], called

OMPT, has been proposed by the OpenMP Tools Working group. OMPT is currently being evaluated to be included in the OpenMP standard. A prototype implementation of this new interface has recently been added to the Intel OpenMP runtime system [15]. OMPT new objectives intend to provide stack frame support for sampling tools as well as to specify interfaces for applying blame shifting logic to resource synchronization. OMPT, as ORA, includes support for events and states.

Various implementations were previously proposed to handle task parallelism for languages and runtime systems. Cilk [16] is a language and runtime system developed at MIT to exploit parallelism using a multithreaded environment. The Cilk runtime system handles optimizations details such as load balancing, synchronization, and communication protocols. Intel's Thread Building Blocks (TBB) [17] is a library based on C++ templates where Building Blocks are tasks. LilyTask [18] is a task programming model designed for shared and distributed memory systems. Each task operates on its own view of memory with data transfers occurring at the beginning and end of execution.

Before the introduction of tasks in OpenMP, a dynamic task generation was integrated in the OpenMP API by the workqueuing model [19] developed by Intel. One thread enqueues tasks defined within a taskq block, while the other threads in the team dequeue the work from the queue. An implicit barrier at the completion of a taskq block ensures the end of execution for all tasks inside the block. The Nanos group has contributed to the evaluation of OpenMP and its tasking model in [20] and [21]. Finally, the HPCTools group at the University of Houston has its own designs of OpenMP tasking models in the OpenUH runtime system [22].

Each one of the aforementioned task implementations was evaluated via some profiling techniques. However, the lack of a standard interface, the way in which profiling was performed, and the lack of appropriate task events have hampered the observation of task creation and scheduling behavior and performance.

Modern microprocessors provide hardware performance monitors (HPMs) to help programmers understand the low-level behavior of their applications. Several library packages provide access to hardware performance counter information, including the HPM toolkit [23], PAPI [24], and OProfile [25]. These libraries provide facilities to instrument programs, record hardware counter data, and analyze the results. However, task scheduling strategies need comprehensive events and states to be linked to HPMs in order to obtain a sufficient evaluation.

III. Performance Framework Layers

OpenMP task scheduling infrastructure in the runtime library has many aspects to be observed. These aspects can be represented by states and events that can be defined in the OpenMP runtime library and captured via a performance tool. A robust performance framework is developed here that includes many layers. This framework offers a deep insight into the performance issues associated with OpenMP task scheduling and creation. Figure 1 demonstrates the different layers of this framework and analysis flow.

A. OpenUH Runtime Support for Tasks and ORA

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.

1) OpenMP task configuration in OpenUH: The HPCTools group implemented a configurable task pool framework that allows a user to choose an appropriate task queue organization at runtime. Furthermore, the user may control the order in which tasks are removed from a task queue for greater control over task scheduling. We currently have four different task pools implemented that utilize distributed, hierarchical, and hybrid queue organizations. Our task configuration provides approaches that employ both depth-first and breadth-first schedulers. The order in which tasks are removed can also be varied to obtain the best performance for a given application. The variation in choosing a specific task configuration has a big influence on data locality and load balancing optimizations. Task scheduling and creation in the OpenMP runtime of OpenUH can be configured by a set of environment variables. In the following, we describe the task configurations used in our analysis:

- O64_OMP_TASK_POOL: Gives control over the organization of task queues; hence, it has an influence on efficiency, queue contention and work-stealing. Four different options can selected:
  1) DEFAULT: Each thread has two queues; One for tied tasks and another for untied tasks.
  2) SIMPLE: Each thread has one queue which holds any unscheduled tied or untied tasks that it creates.
  3) SIMPLE_2LEVEL: Each thread has a fixed-size queue for the tasks it creates. A global queue can be used when the private queue is filled.
  4) PUBLIC_PRIVATE: Each thread has a public and private queue. An environment variable, O64_OMP_TASK_POOL_GREEDVAL, is used to control how many tasks are placed in the private and public queues.
- O64_OMP_TASK_QUEUE: Influences how tasks are scheduled. The types of queues used are:
  1) DEQUE: Puts task to the tail of the queue, gets task from the tail of the queue, steals task from the head of the queue, donates task to the head of the queue.
  2) FIFO: Puts and donates tasks to the tail of the queue, gets and steals tasks from the head of the queue.
  3) CFIFO: same as FIFO, except more efficient because it allows concurrent access to head and tail of a given queue.

4) LIFO: Puts and donates tasks to the tail of the queue, gets and steals tasks from the tail of the queue.
5) INV_DEQUE: Puts task to the tail of the queue, gets task from the head of the queue, steals task from the tail of the queue, donates task to the head of the queue.

- O64_OMP_QUEUE_STORAGE: Specifies low-level control of the queue API provided by the runtime.
  1) ARRAY: Queue has a fixed number of slots.
  2) DYN_ARRAY: Queue will grow (double in size) whenever an item needs to be added and it is full; the queue remains contiguous.
  3) LIST: Queue grows when adding an item and it is full, but does so by allocating linking together separately allocated buffers.
  4) LOCKLESS: Queue operates much like ARRAY, except no locks are used.
- O64_OMP_TASK_CHUNK_SIZE: Controls the number of tasks that can be stolen at once from a queue. We used two values in our experiments: one and two.
- O64_OMP_TASK_QUEUE_NUM_SLOTS: Controls the default size of the task queues. A value of 128 is used in our experiments.
- O64_OMP_TASK_POOL_GREEDVAL: Controls how many tasks are placed in the private and public queue, when the PUBLIC_PRIVATE pool is used. Two values were used: One means all tasks are placed in the public queue and available for stealing. Two offers an equal approach. Half the created tasks are placed in each queue. Every other task in the public queue.

The work queue is protected against concurrent accesses by using a mix of spinlocks and mutexes.

2) ORA Support in OpenUH: The ORA interface [1] 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. The ORA enables an interaction between the OpenMP runtime library and a performance tool. The ORA also consists of states, events, and requests that are implemented in the OpenMP runtime library. As a main advantage of the ORA, compiler analysis is not affected since no source code modifications are required. Hence, the performance measurements, collected by a performance tool, are more accurate. For a performance tool to take advantage of the ORA, it needs to implement one or more event handlers to process the events as they are encountered. The tool has the option to implement a single handler for all events, a different handler for each event, or some combination.

An open source implementation [13]for the ORA was developed in the OpenUH runtime library. Originally, there are 11 mutually exclusive states, 9 requests, and 22 defined callback events, representing the entry and exit for commonly used pragmas. We recently proposed new extensions to the ORA to support OpenMP tasks as well as a prototype profiler to selectively obtain measurements for individual task instances. The proposed states, events, and requests cover task creation, scheduling, suspension, switching, execution and completion. A full description of these new extension can be found in [3].

Figure 2 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. Each phase is represented by a set of runtime routines. The main task routines in this model are described in the following. __ompc_task_create() creates a task and inserts it into a queue. A new created task can be deferred (waiting in a pool) or executed immediately depending on the availability of threads and the scheduling strategy used. __ompc_task_wait() suspends a task until all of its child tasks complete execution. __ompc_task_switch() switches the execution from one task to a new task once this new task is removed from the pool. __ompc_task_exit() performs cleanup and allows a new task to be placed in a queue. These routines were modified to capture their associated events. Two new function calls are invoked in these routines to implement the different events and states associated with each task instance.

B. ORA Integration into TAU

A performance tool can interact with the runtime through some requests implemented in the ORA. OMP_REQ_START request should be used first to initialize the collector API to establish a connection with the runtime. OMP_REQ_REGISTER request should be used next to selectively register the task events required for our calculations. A tool should be able to request the state of any thread during execution. TAU now has full support for the ORA implemented in OpenUH. A single event handler is implemented to handle all events. When any master or worker thread enters any OpenMP region, it fires an event which is processed by TAU. The event contains the unique ID for the parallel region which is used to resolve the context for the worker thread. Regarding tasks, tracking each task instance will produce significant overheads; Hence, the task instances called from the same calling context are aggregated. In this way, we are able to thoroughly analyze the efficiency of the different task configurations implemented on OpenUH against the tested OpenMP benchmarks. This analysis is performed through capturing and visualizing the task events as well as the other encountered events. TAU obtains timing measurements associated with each captured event for each working thread. Additionally, we integrate PAPI into TAU to obtain hardware performance measurements associated with each event. Through two HWCs: PAPI_L2_TCM and PAPI_L2_TCA, L2 cache miss rate of each encountered event is calculated. Cache misses gave us a sufficient level of accuracy to distinguish the impact of using various schedulers on the performance. However, other counters such as TLB, CPI, etc will be considered in our future work analysis. We

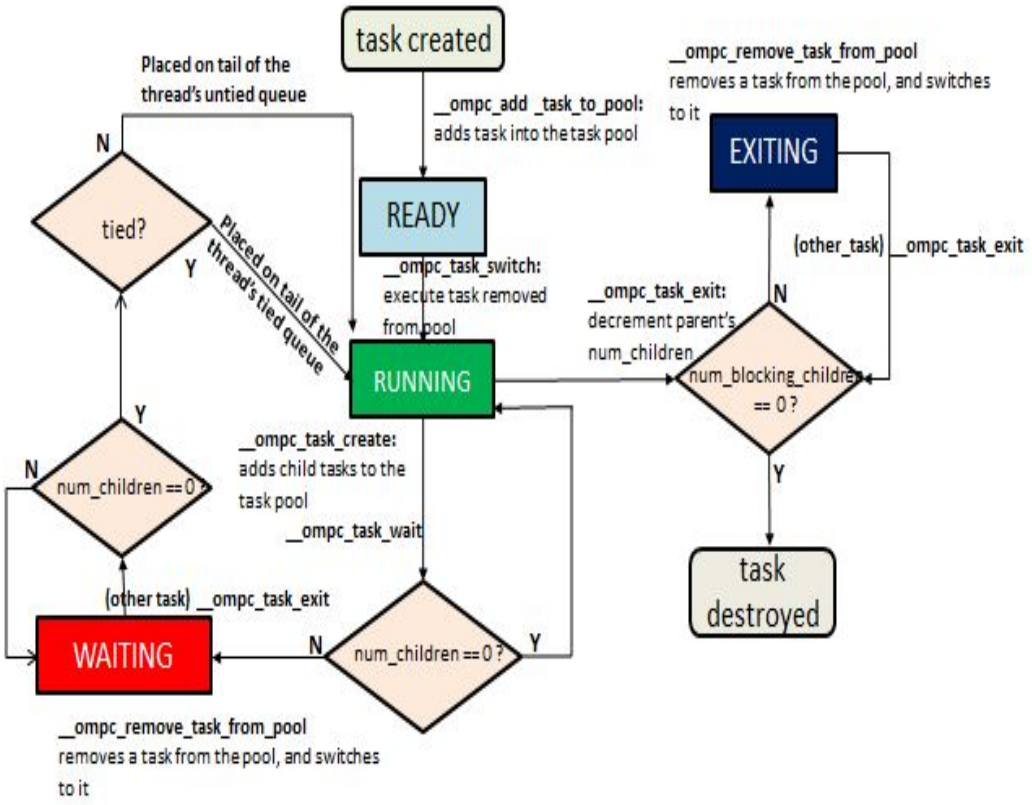

Fig. 2: OpenUH tasking execution model

have chosen level 2 cache as it is the last level of private caches in our system and reflects the performance of each thread per core.

C. Benchmarks Layer

We performed our analysis on eight different benchmarks implemented with tied and untied tasks in the Barcelona OpenMP Task Suite (BOTS) [20]. These benchmarks vary in terms of granularity, # of task directives, support for nested tasks, and computation structure. This variation is beneficial to comprise all possibilities. A brief description of these benchmarks is followed:

Fibonacci: Computes the nth Fibonacci number using a recursive paralellization. It represents a deep tree composed of very fine grain tasks.

FFT: Is a divide and conquer algorithm that computes the one-dimensional Fast Fourier Transform of a vector of n complex values. In each of the divisions multiple tasks are generated.

Strassen: Uses hierarchical decomposition of a matrix for multiplication of large dense matrices. For each decomposition, which is dividing each dimension into two halves, a task is created.

Sort: Sorts a random permutation of n 32-bit numbers with a variation of sorting algorithms. Tasks are used for each split and merge.

Health: Uses multilevel lists where each element in the structure represents a village with a list of potential patients and one hospital. The hospital has several double-linked lists representing the possible status of a patient inside it. A task is created for each village.

Alignment: Aligns sequences of proteins using dynamic programming.

Nqueens: Finds solutions of the n-queens problem using backtrack search.

SparseLU: Computes the LU factorization of a sparse matrix.

For both the Alignment and SparseLU benchmarks, BOTS provides two different versions. In the first (Alignment-single), the loop nest that generates the tasks is executed by a single thread. This version creates only task parallelism. In the second (Alignment-for), the outer loop is executed in parallel, creating both loop-level parallelism and task parallelism. Similarly, the two versions of SparseLU are one in which tasks are generated within single-threaded loop executions and another in which tasks are generated within parallel loop executions. We obtained results for both versions. However, we just record the for version in this paper.

IV. Evaluation

This section intends to accomplish several objectives, which are fundamental for OpenMP tasking application and runtime developers to have a deeper insight into the behavior of OpenMP task scheduling strategies and applications. It

also highlights the main efficiencies and weaknesses in the OpenUH tasking model implementation as well as in the various applications. This analysis leads to further development and optimizations.

In order to achieve these objectives, we have chosen a reliable shared memory system and a variety of well-known benchmarks, explained in the previous section, for our evaluation. The measurements were taken on an x86-64 cc-NUMA Linux system with a four 2.2 GHz 12-core AMD Opteron processor (48 cores total) and 64 GB RAM, 64 KB L1 instruction and 64 KB L1 data cache per core, 512 KB L2 cache per core, and 10 MB L3 cache shared by all cores. We use the average of three runs of a benchmark for all results. We chose three runs only since variation between them was negligible. All compilations of codes use the OpenUH C compiler, while it is attached to the TAU performance system. For L2 cache miss rate measurements, PAPI HWCs are also linked to this integration. We first build all the possible task scheduling configurations by varying the set of environment variables described in the previous section at runtime. More than one hundred combinations are configured and tested with each benchmark. We then illustrate the impact of this task configuration variation on the performance of each benchmark. Later, we provide a comprehensive and thorough analysis to justify the obtained measurements.

Table I through Table VIII show the timing measurements in seconds obtained for eight different BOTS benchmarks. Two input sizes were used with each benchmark. OMP_NUM_THREADS was set to three values: 2, 8, and 32. When 48 threads were used, inconsistent results were obtained and hence, we decided not to use this option. We used the default core mapping provided in our NUMA system. Tied and untied versions of the benchmarks were experimented. No cut-off configurations provided in BOTS are enabled to put more responsibility on the task scheduler itself. For the Alignment and SparseLU benchmarks, both for and single versions were used. Both versions offer similar behavior; However, only the for version measurements for Alignment and SparseLU are shown in Table I and Table II respectively. Best and Worst keywords in these tables and the following figures indicate the task scheduling configurations, which demonstrate the best and worst performance respectively associated with each benchmark. Accordingly, in the L2 cache miss rate Figures, Best means the task configuration, which gives the lowest execution time, while Worst means the task configuration, which gives the highest execution time. The best/worst/default task scheduling strategies include one option from each environment variable mentioned in the previous section. However, the other options, which define the best/worst scheduling strategies, intersect which make them less relevant, and hence we decided to focus on the organization of task queues in our analysis.

Unsurprisingly, the benchmarks, featuring similar characteristics, show similar behavior with respect to the task scheduling strategies performance. The performance obtained for tied and untied tasks in all benchmarks was almost identical.

TABLE I: Alignment timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	prot.20.aa	prot.100.aa	prot.20.aa	prot.100.aa	prot.20.aa	prot.100.aa	prot.20.aa	prot.100.aa
2	0.63	12.3	0.65	12.3	0.65	12.5	0.66	12.5
8	0.31	3.3	0.31	3.3	0.34	3.5	0.32	3.4
32	0.31	1.5	0.31	1.5	0.36	1.7	0.36	1.7

TABLE II: SparseLU timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	n=30/m=60	n=50/m=100	n=30/m=60	n=50/m=100	n=30/m=60	n=50/m=100	n=30/m=60	n=50/m=100
2	1.63	32	1.63	32	1.65	32.5	1.65	32.5
8	0.47	8.3	0.48	8.4	0.5	8.5	0.5	8.4
32	0.27	2.5	0.25	2.5	0.34	3.5	0.35	3.5

Out of ten benchmarks used, five benchmarks display a good speedup. These benchmarks are: both versions of Alignment, both versions of SparseLU, and Strassen. For these benchmarks, the highest efficiency is obtained when 8 threads are used. The for and single versions of Alignment and SparseLU present a speedup close to ideal with most task scheduling strategies. However, the best performance is gained when the DEFAULT task pool configuration is used while tasks are stolen from the tail of the queue. One reason is that a thread has more freedom to steal work (from the untied queues) when it has tasks tied to it which are not suspended in a barrier. On the other hand, the PUBLIC task pool displays the worst performance when the public and private queues, assigned to each thread, have the same size.

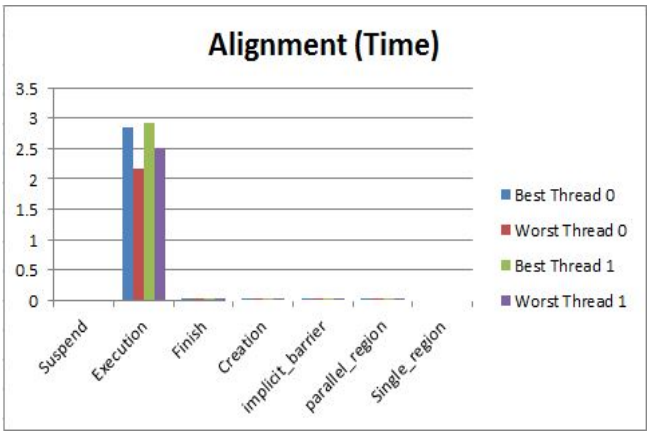

Fig. 3: A detailed timing measurements associated with Alignment benchmark

To elaborate on these findings, we consult our developed framework to find out the time distribution in seconds as well as the L2 cache miss rate for each individual event, encountered in four selective benchmarks. Figure 3 through Figure 10 give information about seven different events in detail in terms of timing and L2 cache miss rate. These captured events are: task suspension, task execution, task completion, task creation, implicit barrier, parallel-region, and single-region. The timing and L2 cache miss rate measurements, shown in these Figures, are exclusive for each captured event. The larger input size is

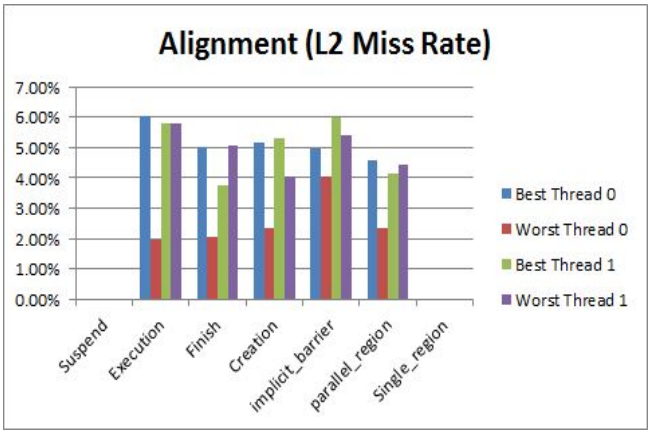

Fig. 4: A detailed L2 cache miss rate measurements associated with Alignment benchmark

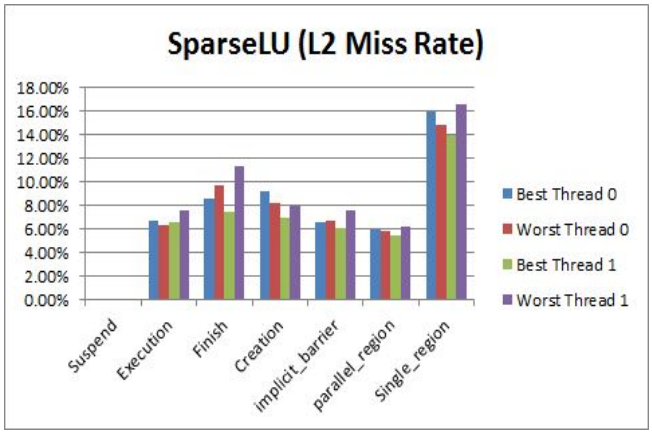

Fig. 6: A detailed L2 cache miss rate measurements associated with SparseLU benchmark

used and two threads are exploited. This should enable the user to see how load balancing is affected when the task scheduling strategy is varied. Data locality can also be expressed in the figures that display detailed information about the L2 cache miss rate of each event. Figure 3 demonstrates how time is spent in Alignment benchmark among the different events for the two working threads. As shown, the time spent on task execution was significant, compared with the other events. This means that synchronizations among the working threads have a negligible impact. The only factor which makes the slight performance difference between the best and worst task configuration is due to a slight load-imbalance between the two working threads. Figure 4 shows a similar L2 cache miss rate for all events. However, the difference in L2 miss rate between the two threads can be depicted, when the worst task configuration (PUBLIC) is used. This difference is due to the fact that half of the tasks created by a thread will be placed in a public queue, which is accessible by the other threads.

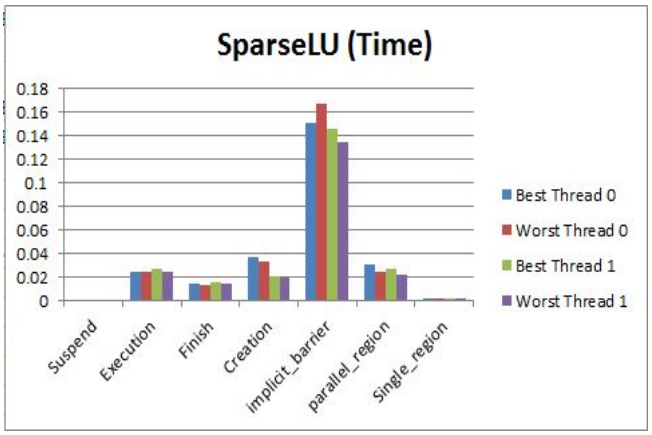

Fig. 5: A detailed timing measurements associated with SparseLU benchmark

SparseLU timing distribution, depicted in Figure 5, shows a high amount of time spent while waiting at the implicit barrier by both threads. This is due to the sparseness of the matrix when it is composed by pointers. As in Alignment, a little load-imbalance occurs with the PUBLIC task pool configuration. Figure 6 shows an L2 cache miss rate behavior similar to that obtained for Alignment. However, the existence of a single construct in the SparseLU source code justifies the difference. In Alignment and SparseLU, there are some coarse tasks and the challenge is to avoid load-balance situations. Using the DEFAULT task organization, more freedom is granted to threads to steal work from each other; As a consequence, the probability to have a load-balance situation is very limited.

TABLE III: Strassen timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	2K	4K	2K	4K	2K	4K	2k	4K
2	3	20.3	3	20	2.9	19	2.93	20
8	1.42	7.2	1.41	7.1	1.72	8	1.4	7.4
32	2	14	2	10	3.6	14.5	3.6	13

The Strassen benchmark measurements, shown in Table III, demonstrate a best performance when the SIMPLE scheduler is used. Strassen performs recursive matrix multiplication using hierarchical decomposition and is challenging for task configurations when high queue contention is encountered by multiple threads. Though, an approximate speedup of 3x is gained in the best case. The SIMPLE scheduler relieves high contentions by allocating one queue to each thread. This queue will hold unscheduled tied and untied tasks created by any particular thread. As a result, less interaction between threads might occur. As expected, the worst performance is encountered when a generous PUBLIC approach is used. When O64_OMP_TASK_POOL_GREEDVAL is set to 2, a public queue will be allocated to each working thread holding half of its created tasks. This approach results in high contentions on the public queues, especially when synchronizations become an issue. A more greedy PUBLIC approach might improve the performance of Strassen, since more tasks will be placed in the private queues.

The SIMPLE configuration has the downside that work-stealing is more restricted when a thread has unfinished tasks

TABLE IV: Fibonacci timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	n=20	n=24	n=20	n=24	n=20	n=24	n=20	n=24
2	0.91	6.1	0.88	5.9	1	6.63	0.93	6.37
8	0.98	6.66	0.98	6.56	1.03	6.93	1.03	6.8
32	1.4	8	2.13	14.33	2.23	14.6	2.13	14.4

TABLE V: Health timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	test	small	test	small	test	small	test	small
2	3.2	61.67	3.1	85.3	3.6	124	3.26	115
8	7.5	131	7.23	126	8.06	140	7.67	133
32	7.06	196	6.43	181	13.87	295	12.4	275

that are tied to it. Despite this restriction, our results show a similar task scheduling behavior to the one encountered by Strassen when Fibonacci, Health, and NQueens were tested. Fibonacci measurements, shown in Table IV, and Health results, shown in Table V, are found to use the SIMPLE task pool configuration for the best performance, and the PUBLIC task scheduler for the worst performance. However, unlike Strassen, these two benchmarks could not scale well with the increment of threads. As mentioned before, no cut-off mechanism is used in our experiments. These two applications have a massive amount of very fine-grained tasks. In such case, the main challenge for a task scheduler is to exploit the available parallelism while reducing the generated overheads.

In order to have a better explanation, Figure 7 depicts the time distribution among two working threads for each captured event in Fibonacci. To keep track of task creation and suspension, each fine-grained task instance has to be captured. Capturing these tasks might result in a time overheads larger than the actual time spent while creating and suspending these tasks. As can be seen in Figure 7, the time spent on task suspension is the most significant. This is due to the fact that, with the input size used; n=24, millions of fine-grained child tasks are created. The parent tasks at each level will be suspended waiting for their child tasks to finish execution. Time spent on task creation is also considerable, as a huge number of tasks is created. Work-stealing is important to manage the execution of tasks while new tasks are still created. However, data locality should efficiently be exploited to reduce the generated overheads. Removing tasks from queues and destroying them also contribute to the overall generated overheads. In Fibonacci, load balancing should not be a big concern. This conclusion can be depicted in the same Figure. Thread 0 and thread 1 spend almost the same time, when the best task configuration (SIMPLE) is used. These two threads also exhibit a similar behavior, even with the worst task configuration (PUBLIC). The SIMPLE scheduler exploits data locality by allocating one queue to each thread. This queue will hold unscheduled tied and untied tasks created by any particular thread. In particular, if a thread has any tasks tied to it which are not suspended in a barrier, then it may not steal work from the queues of other threads. As a result, tasks working on the same block of code will have higher chances to be placed in the same queue. To obtain the best performance for Fibonacci, tasks are scheduled in a LIFO order. The LIFO order allows a depth-first execution of the tasks while child tasks may be stolen from the back-end of the queue. This method utilizes the data locality offered in depth-first execution but requires more memory for storing the overabundant tasks.

As in Strassen, the worst performance for Fibonacci is encountered when a generous PUBLIC task pool organization is used. This pool organization might cause a deficiency in exploiting data locality as it allows other threads to steal more tasks from the public queues. The performance difference between the best and worst schedulers is very small.

Figure 8 shows the L2 cache miss rate behavior for each encountered event on Fibonacci. The main observation here is that a high value of L2 cache miss rate for a specific runtime event does not necessarily mean a high value of execution time for the same event. This conclusion can be depicted in the L2 cache miss rate measurements associated with implicit-barrier, parallel-region, and single-region events. Slight differences can be found between the two working threads. The only exception, which is due to synchronization overheads, is in the implicit barrier and single-region events.

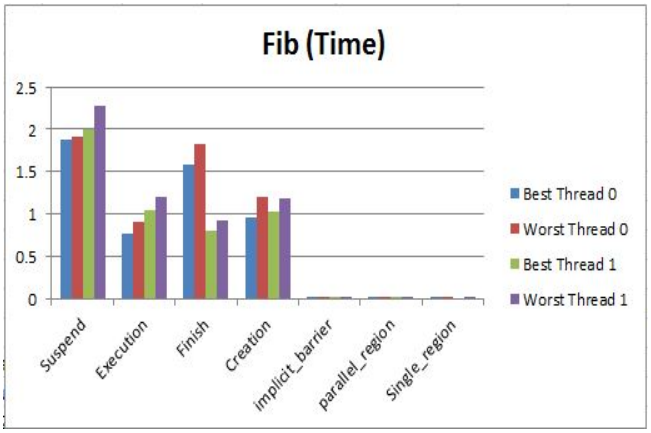

Fig. 7: A detailed timing measurements associated with Fibonacci benchmark

The Health benchmark, shown in Figure 9 and Figure 10, shows a very similar behavior, compared with the Fibonacci benchmark, with respect to the performance of the different task scheduling strategies. This similarity is due to the similar characteristics of these two benchmarks that include: task granularity, memory/computation bound operations, and the amount of data that is communicated from the parent to its child tasks at creation.

The NQueens benchmark, depicted in Table VI, is another example of an application with a massive amount of fine-grained tasks. NQueens uses a backtracking search algorithm with pruning. A task is created for each step of the solution. Again, the fine granularity makes this benchmark non-scalable for profiling. Regarding the performance of the different task schedulers, NQueens exhibits the same behavior as in

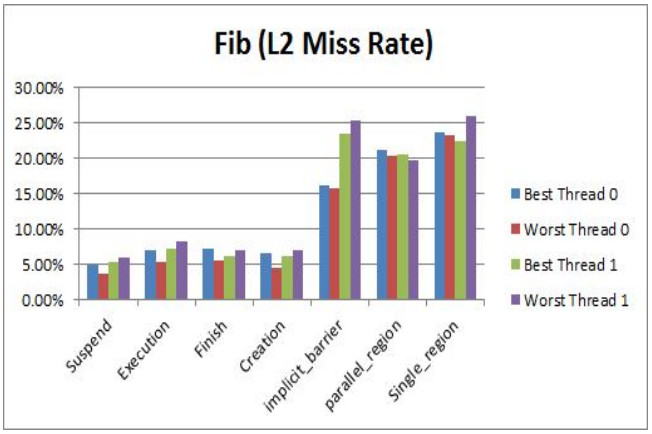

Fig. 8: A detailed L2 cache miss rate measurements associated with Fibonacci benchmark

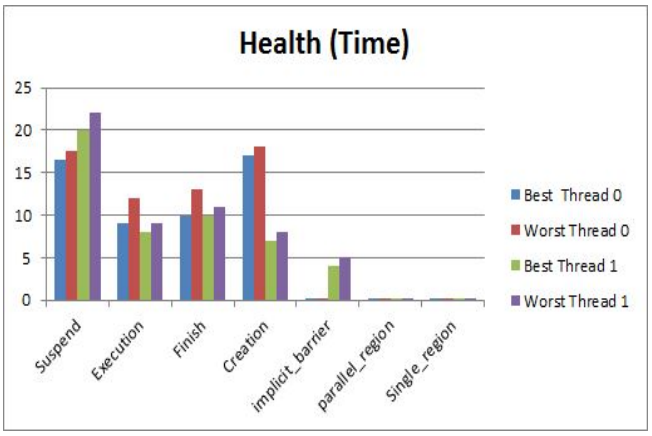

Fig. 9: A detailed timing measurements associated with Health benchmark

Fibonacci and Health benchmarks.

TABLE VI: NQueens timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	n=9	n=10	n=9	n=10	n=9	n=10	n=9	n=10
2	2.7	13	2.8	13	3	14.6	3	14
8	3.1	14.5	3.1	14.6	3.3	16	3.3	16
32	7	34	7.1	33.5	8	37	8	37

Finally, on the data-intensive Sort and FFT benchmarks, we have observed an increase in computation time due to increased load latencies and fine task granularity overheads. The measurements for Sort and FFT are shown in Table VII and Table VIII respectively. Both of these benchmarks implement a divide and conquer algorithm to perform computations. The best performance was obtained when the PUBLIC task pool organization was used. We gained the best performance for these two benchmarks when tasks are scheduled in a DEQUE order. On the other hand, The worst performance was obtained when the DEFAULT task pool organization was used.

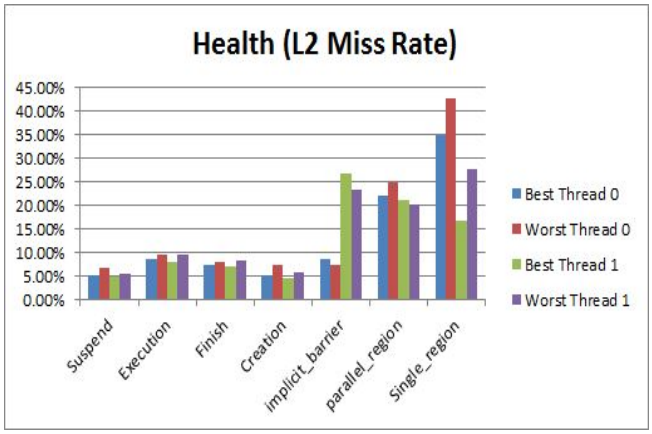

Fig. 10: A detailed L2 cache miss rate measurements associated with Health benchmark

TABLE VII: Sort timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	16m	32m	16m	32m	16m	32m	16m	32m
2	3.4	6.3	3.3	6.3	3.5	6.5	3.5	6.5
8	6.4	11.8	6.3	11.5	11.6	21	12.6	22.5
32	9.6	21	11	20	11	21	12	22

V. Conclusion and Future Work

In this work, we have presented our experiences in evaluating and analyzing various OpenMP task scheduling strategies implemented in OpenUH. We have extended the OpenMP Runtime API for profiling (ORA), implemented in the OpenUH open-source compiler, to support OpenMP task profiling. These extensions include task creation, scheduling, suspension, stealing, execution and completion. We integrated the ORA with our new task extensions into the TAU performance system. We then performed a comprehensive evaluation through this new robust OpenMP performance framework against different well-known OpenMP task applications on a multicore system. Our performance study revealed the strengths and limitations of the OpenUH task scheduling strategies with respect to their performance. The organization of task queues was proven to have the highest impact on performance, compared with the order in which tasks are scheduled, the queue API, and some other tasking variables. Profiling applications with a massive amount of fine-grained tasks generates some overheads associated with task suspension and creation. Restriction of work-stealing capability is beneficial for applications with fine task granularity, while applications with coarse tasks exploit the public queues to steal work for a better efficiency. A low L2 cache miss rate for a specific ORA event does not necessarily indicate a better execution time. Finally, the approach provided in this work for task scheduling analysis can be adopted by the other tool interfaces such as POMP and OMPT for efficient observability. This work is considered as a crucial step for further development and analysis. We plan to work on establishing a statistical runtime model to automate the process of choosing the best

TABLE VIII: FFT timing measurements

	Best				Worst			
	Tied		Untied		Tied		Untied	
#Thr/Input	2M	4M	2M	4M	2M	4M	2M	4M
2	2.6	10.7	2.7	10.6	3.4	14.1	3.6	14
8	5.3	21	5.2	20	5.5	22	5.5	22
32	10	44	10	41	11	45	11.5	46

task scheduling strategy for any given application. We will build a statistical scoring formula, which takes the different ORA events as inputs to categorize different applications into clusters. We also intend to include more benchmarks and real applications in our future analysis.

ACKNOWLEDGMENT

The authors would like to thank Kevin Huck from the TAU group at the University of Oregon and Deepak Eachempati from the HPCTools group at the University of Houston for their collaboration in this work. 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] 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*, vol. 314, pp. 181–190, 2007.

[2] A. Qawasmeh, B. Chapman, and A. Banerjee, "A compiler-based tool for array analysis in HPC applications," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 454–463.

[3] A. Qawasmeh, A. Malik, B. Chapman, K. Huck, and A. Malony, "Open source task profiling by extending the openmp runtime api," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 186–199.

[4] A. Qawasmeh, A. Malik, D. Eachempati, and B. Chapman, "Task profiling through openmp runtime api and tool support," Nov. 2013. [Online]. Available: http://sc13.supercomputing.org/sites/default/files/PostersArchive/post170.html

[5] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for OpenMP," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, 2001.

[6] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[7] B. Mohr and F. Wolf, "KOJAK–a tool set for automatic performance analysis of parallel programs," *Euro-Par 2003 Parallel Processing*, pp. 1301–1304, 2003.

[8] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[10] Intel. (2008) Intel Thread Profiler-Product Overview. [Online]. Available: http://software.intel.com/en-us/articles/intel-thread-profiler-product-overview/

[11] Barcelona Supercomputing Center. (2014) Paraver: a flexible performance analysis tool. [Online]. Available: http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview

[12] V. Bui, O. Hernandez, B. Chapman, R. Kufrin, D. Tafti, and P. Gopalkrishnan, "Towards an implementation of the OpenMP collector API," *Urbana*, vol. 51, p. 61801, 2007.

[13] O. Hernandez, R. C. Nanjegowda, B. Chapman, V. Bui, and R. Kufrin, "Open source software support for the OpenMP runtime API for profiling," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 130–137.

[14] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt and ompd: Openmp tools application programming interfaces for performance analysis and debugging," 2013.

[15] Intel. (2013) Intel open source OpenMP runtime. [Online]. Available: www.openmprtl.org

[16] S. T. G. M. L. for Computer Science, "Cilk 5.4.6 reference manual," Oct. 2010. [Online]. Available: http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

[17] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2010.

[18] T. Wang, N. Di, J. Shen, and Y. Tang, "Lilytask programming model and its implementations on smp & cluster." in *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, vol. 1, 2003, pp. 301–306.

[19] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen, "Compiler support of the workqueuing execution model for intel smp architectures," in *European Workshop on OpenMP (EWOMP02)*, 2002.

[20] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Parallel Processing, 2009. ICPP'09. International Conference on*. IEEE, 2009, pp. 124–131.

[21] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.

[22] B. Chapman, D. Eachempati, and O. Hernandez, "Experiences developing the openuh compiler and runtime infrastructure," *International Journal of Parallel Programming*, pp. 1–30, 2012.

[23] (2008) Hardware performance monitor(hpm) toolkit users guide. [Online]. Available: https://wiki.alcf.anl.gov/images/5/59/HPM_ug.pdf

[24] P. J. Mucci, S. Browne, C. Deane, and G. Ho. (1999, Sep.) Papi: A portable interface to hardware performance counters. dodugc99-papi.pdf. [Online]. Available: http://web.eecs.utk.edu/~mucci/latest/pubs/

[25] W. E. Cohen. (2004) Tuning programs with oprofile. Oprofile.pdf. [Online]. Available: http://people.redhat.com/wcohen/