A Runtime Implementation of OpenMP Tasks*

James LaGrone[1], Ayodunni Aribuki[1], Cody Addison[2], and Barbara Chapman[1]

[1] University of Houston,
Houston, TX, USA
{jlagrone,dunniie,chapman}@cs.uh.edu
www.cs.uh.edu/~hpctools/
[2] Texas Instruments Incorporated,
Stafford TX 77477, USA
c-addison@ti.com

Abstract. Many task-based programming models have been developed and refined in recent years to support application development for shared memory platforms. Asynchronous tasks are a powerful programming abstraction that offer flexibility in conjunction with great expressivity. Research involving standardized tasking models like OpenMP and non-standardized models like Cilk facilitate improvements in many tasking implementations. While the asynchronous task is arguably a fundamental element of parallel programming, it is the implementation, not the concept, that makes all the difference with respect to the performance that is obtained by a program that is parallelized using tasks. There are many approaches to implementing tasking constructs, but few have also given attention to providing the user with some capabilities for fine tuning the execution of their code. This paper provides an overview of one OpenMP implementation, highlights its main features, discusses the implementation, and demonstrates its performance with user controlled runtime variables.

Keywords: OpenMP Tasks, Parallel Programming Models, Runtime Systems

1 Introduction

OpenMP began as a loop-centric programming model consisting of a collection of worksharing and synchronization constructs to allow Fortran and C/C++ developers to write multithreaded applications for shared memory platforms. The introduction of tasks in the 3.0 specification of the application program interface (API) added needed flexibility to the previously loop-centric nature of OpenMP. Asynchronous tasks now allow the parallelization of applications exhibiting irregular parallelism in the form of recursive algorithms, and pointer-based data structures.

* This material is based upon work supported by the National Science Foundation under Grant No. CCF-0833201 and Grant No. CCF-0917285, and the Texas Space Grant Consortium.

In the new tasking model, the programmer specifies independent units of work, called *tasks*, and may use synchronization constructs to guarantee completion of tasks at certain points in the program. The scheduling decisions of where and when to execute the tasks is left to the runtime system.

As with any programming model or language, it is in the overall quality of the implementation and the detailed design choices, as well as the skill with which they are engineered, that performance will be realized, either good or poor. The implementation of the runtime system support for OpenMP tasks is key to the performance of task-based programs. There are a number of design decisions, ranging from how to store tasks in the system to how to schedule the tasks in a way that maximizes parallelism while incurring minimal overhead, that may have a substantial impact on the behavior of OpenMP 3.0 code. A robust and efficient runtime implementation of tasks is at the heart of achieving fast and scalable execution.

In the rest of this paper we survey the OpenMP task features in Section 2. Section 3 provides an overview of the OpenUH compiler and its support for OpenMP. We discuss the design considerations for runtime systems supporting OpenMP tasks in general, and describe our runtime support for tasks in Section 4 and evaluate our implementation in Section 5. We summarize related work in Section 6, and conclude and briefly outline our future plans in Section 7.

2 Overview of OpenMP Tasks

An OpenMP task is formed when a thread encounters a `task` or `parallel` construct. It is comprised of an instance of executable code and its associated data environment. The OpenMP specification distinguishes explicit tasks declared by the application developer from implicit tasks that are not declared by the programmer and that contain the work within the parallel portions of the code. When a `parallel` construct is encountered, a set of *implicit* tasks is created, one task per thread, and their execution is begun. A thread encountering the `task` construct will create an *explicit* task. The execution of an explicit task may be immediate or delayed. If a task contains an `if` clause which evaluates to *false*, the encountering thread must suspend the current task region and immediately execute the generated task. The specification further permits an implementation to suspend the generation of tasks, and to suspend the execution of specific tasks, in order to maintain system efficiency. It also defines *task scheduling points* – those points in the code where tasks may be suspended for completion at a later time. A task is by default *tied* to a thread, whereby, if suspended, the task may only resume on the thread which began its execution. A task marked `untied` may resume execution on any thread in the current thread team.

The data in a task may take the usual OpenMP data-sharing attributes of `shared`, `private` or `firstprivate`. The default data-sharing attribute for a task region is `firstprivate` unless otherwise specified. To avoid the data values changing or going out of scope before the execution of the task, the values of `firstprivate` variables are captured at the time the task is created. Synchro-

nizations are achieved through the use of `taskwait` and `barrier` constructs. The `taskwait` construct causes the encountering task region to suspend and wait for all of its child tasks to complete before resuming execution. When a `barrier` is encountered, all threads must wait until all other threads reach the barrier and all tasks created prior to the barrier are completed. Figure 1 shows the Fibonacci kernel with two task directives with shared clauses.

```
1  int fib(int n) {
2     int x, y;
3     if (n < 2)
4       return n;
5     else {
6        #pragma omp task shared(x)
7           x = fib(n - 1);
8        #pragma omp task shared(y)
9           y = fib(n - 2);
10       #pragma omp taskwait
11          return x + y;
12    }
13 }
```

Fig. 1. A naïve implementation of the Fibonacci kernel using OpenMP tasks.

3 OpenUH Implementation of OpenMP

OpenUH [22], [19], a branch of the Open64 compiler suite, is a near-production quality C/C++ and Fortran compiler under development at University of Houston that supports the bulk of the OpenMP 3.0 API. OpenUH provides highly optimized native code for both x8664 and IA64 architectures or source-to-source translation using a variety of state-of-the-art analyses and transformations, sometimes at multiple levels.

OpenUH also has the first implementation of the OpenMP Runtime API for tools [15], also know as the "collector API", in an open source compiler. OpenUH translates OpenMP directives and function calls into multithreaded code for use with a custom runtime library (RTL). The `omp parallel` and `omp task` constructs are transformed into separate program units using inlining [19] whereby the program unit is nested in its caller.

The OpenMP RTL in OpenUH manages the creation and synchronization of threads. In addition to supporting most of the OpenMP API, we have also designed and implemented novel extensions that provide greater flexibility and scalability when mapping work to many cores [6]. We have an enhanced barrier implementation that allows the user to designate the type of barrier used via an environment variable [21]. This portable RTL includes a built-in performance monitoring capability and has been coupled with external performance tools to support the analysis and visualization of a program's performance [16]. The RTL

provides the OpenMP library calls as specified in the API and also serves as an abstraction of the underlying thread library. In addition to thread management, we have assembled a competitive implementation for scheduling, executing, and synchronizing OpenMP tasks.

4 Runtime Support for Tasks

The inclusion of a task feature is important for applications exhibiting irregular parallelism, such as those that are highly recursive or use pointer-chasing algorithms. Due to its dynamic nature, a tasking implementation is more complicated than its static counterparts. Poor implementations can introduce avoidable overheads which will likely limit the scalability of the implementation. Tasks in OpenMP are still a relatively new feature and there is a high likelihood that the tasking model will be extended in the future. It is feasible that information from the compiler could be used with a runtime solely based on tasks [27]. It is therefore critical for the basis of a tasking runtime to be efficient before being extended.

In addition to creating and enforcing dependencies between tasks, the runtime now also functions as a scheduler, deciding how and when to schedule tasks. The efficiency of the runtime implementation will therefore heavily impact the performance of applications using tasks. An ideal task scheduler will schedule tasks for execution in a way that maximizes concurrency while accounting for load imbalance and locality to facilitate better performance. The efficiency of the task scheduler depends on data structures for storing unfinished tasks, how to manage task switching, and regulation of task creation. Additional concerns include task synchronization and the memory footprint of a task.

4.1 Queue Organization

Most schedulers are built around a set of queues for storing tasks that are ready for execution. The queues may be organized in a variety of ways [18]. The simplest approach consists of a single centralized queue shared by all the threads. While a centralized queue is easy to implement and promotes load balancing, it typically performs poorly due to contention for access to the queue. To relieve contention, multiple queues can be used to distribute the accesses. Another solution to this problem is to have threads remove multiple tasks from the queue at each access.

Another implementation strategy is a distributed queue organization where one queue is provided for each thread, with each thread managing its own queue. This eliminates the contention problem since each thread primarily accesses its local queue. However, load balancing becomes an issue if a thread runs out of tasks. Work-stealing may be employed to allow idle threads to steal work from another thread's queue. Distributed queues may also promote data locality by having the runtime place related tasks in the same queue.

Hierarchical queues are yet another approach, where queues are organized in a tree structure. When a task is created it is placed on a root queue. Typically threads access their private queues first, then progress up the tree until work is found. When work is found, some number of tasks are moved down one level in the tree, effectively assigning that work to a subset of the threads. A major advantage of this approach is that it provides a natural way to represent the underlying architecture, especially in terms of memory hierarchy. Shared queues may, for example, be associated with processors that share a common memory. However, since more queues exist, it can require multiple accesses to find work and move it down to private queues where they can be accessed quickly.

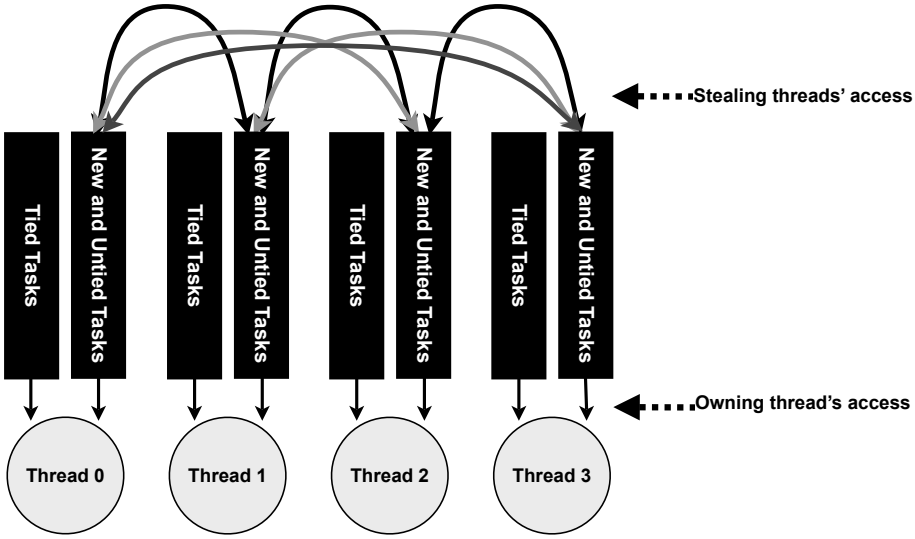


Fig. 2. Organization of queues (deques) for storing unfinished tasks in OpenUH.

In the OpenUH RTL implementation, we use a variation of the distributed queue strategy based on local queues with work stealing, where each thread has a public and private local queue (see Figure 2). The public queues are accessible by all threads and are used for work-stealing. A private queue, on the other hand, can only be accessed by its own thread. By using multiple public queues instead of a shared global queue, contention is distributed across the system. The queues are doubly-ended queues, or *deques*, to allow flexibility when developing scheduling algorithms, and are implemented as doubly-linked lists. Locks are used on public queues to guarantee mutual exclusion.

4.2 Task Scheduling

The OpenMP specification does not prescribe the algorithm for scheduling tasks for execution. The basic job of a task scheduler is to decide when to execute a task

and which thread executes it. Two important considerations of scheduler design are data locality and load balancing. To address the former, tasks operating on the same data should be scheduled for execution on the same thread to improve data reuse, especially on non-uniform memory access (NUMA) architectures. The scheduler should also be able to dynamically balance the workload on the threads, ensuring that all the threads do the same amount of work.

Task schedulers are broadly classified as work-first and breadth-first schedulers. Work-first schedulers execute tasks immediately after they are created and suspend execution of the parent task, leading to a depth-first creation and execution of tasks. Breadth-first schedulers, on the other hand, have parent tasks create all child tasks before executing them. Breadth-first schedulers may generate a large number of tasks, increasing opportunities for parallelism, but the depth-first execution of work-first schedulers will lead to better data reuse.

Our implementation approach attempts to merge the benefits of both classes of schedulers. We create tasks in a breadth-first order. As a task executes, any subsequent task constructs encountered will result in the enqueuing of the task, in LIFO order, for later execution. The LIFO ordering allows a depth-first execution of the tasks while child tasks may be stolen from the back-end of the deque. This method benefits from the data locality of depth-first execution but requires more memory for storing the excess tasks. While this can become a problem when the number of tasks goes beyond some threshold for a given system, it can also be alleviated with some nuancing of the task scheduling at run time. We discuss this further in Section 4.3.

A thread will first look for work in its private queue to complete execution of any tied tasks, then in its public queue, and finally in another thread's public queue. This provides more opportunities for other threads to steal tasks to promote better load-balancing. Our experimental results suggest that allowing a thread only a single attempt to steal results in the best performance. If the attempt to steal fails, the thread resumes execution of its implicit task. The implicit task is not placed on the deque but is merely suspended and restarted when there are no more tasks left to be executed.

4.3 Regulating Task Creation

A task-generating program can overload a system's resources, like memory usage, placing a strain on the system. Recursive algorithms, in particular, can quickly generate millions of tasks. While the inclusion of a mechanism for regulating task creation is necessary, not all mechanisms provide an ideal environment for all codes. We have implemented a variable for choosing the task creation condition, or *cutoff*, scheme that best fits a given application. If the condition is not met, tasks are not placed on the queue but executed immediately. In keeping with the spirit of other OpenMP runtime controls, the cutoff is controlled with a new environment variable, `OMP_TASK_CREATE_COND`, and a corresponding internal control variable. This allows flexibility at runtime and easy experimentation to find the optimal scheme for a given application.

The default value for `OMP_TASK_CREATE_COND` is `true` where no cutoff is employed and all tasks are enqueued. The `numtasks` and `depth` conditions [9] allows the programmer to set a limit on the total number of tasks and the depth in the task graph produced respectively. The `depthmod` condition is a variation on `depth` whereby the *depth* modulo n for some n is used for the cutoff. This allows the execution of n levels of the task graph at a time. A slightly different method is to regulate each thread's queue volume with an upper and lower limit by way of the `queue` condition. When a queue reaches the upper limit, tasks will be executed immediately upon creation until the queue reaches the lower limit. At that time, tasks will again be enqueued until the upper limit is reached, and so on until all tasks have completed. This enables the task's public queue to have tasks available for other threads to steal and help balance the load. This is an adaptive approach that lends itself to algorithms with an irregular task graph. These schemes will produce different task graphs for the same code.

The size of a task also plays a role in building an efficient runtime. In the creation of a task it is necessary to allocate an adequate amount of memory to accommodate the task without being excessive. The overhead associated with the amount of memory allocated has an impact on execution time. So keeping the memory allocation to a minimum generally allows lower execution times. OpenUH introduces the `OMP_TASK_STACK_SIZE` environment variable to allow fine tuning of the size of the task. While some applications can execute with this value as small as 8KB, others require significantly more. The default value of 64KB seems adequate for most applications. Our use of a user-level thread library allows one fixed stack size for all tasks rather than an optimal stack size for each. This causes an over-allocation of memory for most, if not all tasks, which leads to increased overhead. Experimental results show stack size can have a significant effect on scalability.

4.4 Task Synchronization

Synchronization constructs available for tasks are `omp taskwait` and `omp barrier` constructs. The `omp barrier` construct requires all threads to execute the barrier and all explicit tasks in the binding parallel region to complete execution before any execution proceeds beyond the barrier. Users of our compiler can choose different barrier implementations, as mentioned in Section 4.

The `omp taskwait` construct forces a parent task to wait for all its child tasks to complete before continuing execution. Note that this construct only affects the tasks generated by the parent and not subsequent tasks that may be generated by the child tasks. Each child task maintains a pointer to its parent task, and each parent keeps a count of the child tasks it creates. When a task finishes execution it decrements the parent's counter using an atomic operation. A parent task encountering a `taskwait` construct is suspended until its last completing child places it back on the proper deque – the private deque of the owning thread for a tied task, or any public deque for an untied task. Two likely choices are the deque of the thread which began its execution or that of the thread executing the child waking it up from the suspended state. In our

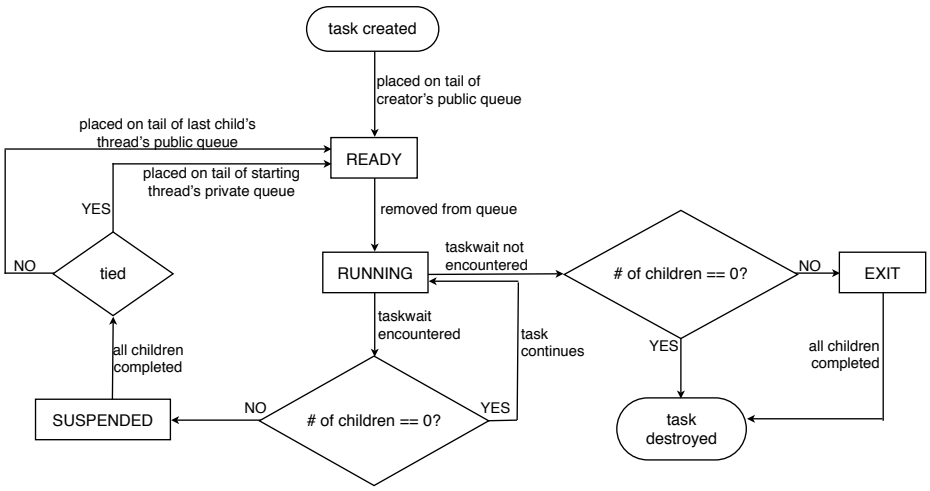


Fig. 3. Flow of a task through the system

experiments, the latter choice resulted in better performance. An overview of this algorithm is shown in Figure 3.

4.5 Task Switching

As mentioned in Section 2, OpenMP defines task scheduling points where a thread may suspend execution of its current task and begin or resume execution of another task bound to the current thread team. This is called task switching. In order to support this, an implementation must be able to move the data environment associated with tasks across threads. For simplicity and flexibility, we chose to implement task switching using a user-level thread library by extending the Portable Coroutines Library (PCL) [20]. The library uses either the `setjump/longjmp` or `ucontext` interfaces depending on the support of the system. We modified the library to make it thread safe, and extended the basic data structure used to implement coroutines to contain the data attributes needed for OpenMP tasks.

5 Evaluation

Due to significant differences in performance when using different architectures, we have chosen three shared memory systems for our evaluation. The first has dual 2.27 GHz 8-core Intel Xeon Nehalem E5520 processors and 32 GB RAM. Pairs of cores share 32KB L1 and 256KB L2 caches with each processor sharing 8MB L3 cache. The other has dual 2.4 GHz quad-core AMD Opteron 2378 processors with 16GB RAM. Each core has 64KB L1 and 512KB L2 caches

with a shared 6MB L3 cache per processor. Both of these systems are running CentOS 5.5 (final) with a Redhat 2.6.18 series kernel. The third system used for our evaluations is an SGI Altix UV 1000 with 16 2.67 GHz 6-core Intel Xeon Nehalem X7542 processors and 2 TB RAM, 32 KB L1 instruction and 32 KB L1 data cache per core, 256 KB L2 cache per core, and 18 MB L3 cache shared by all cores using the SUSE Linux Enterprise Server 11 (x86_64) Linux kernel 2.16.32.

We use the average of three runs of a benchmark for all results. All compilations of codes use the OpenUH C compiler, *uhcc*. Stack size and task creation conditions were all varied at runtime. To highlight features unique to the OpenUH tasking runtime, we have selected two applications, the Sort Benchmark [2] and the NAS Parallel Benchmark(NPB) [26] multi-zone version of BT (BT-MZ).

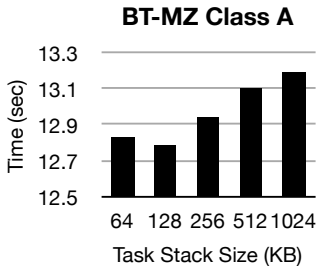
NPB BT-MZ with Tasks The multi-zone application benchmarks are extensions to the original NAS Parallel Benchmarks. The BT-MZ benchmark is a version of the BT application solved on collections of loosely coupled discretization meshes called zones. At each time step, the solutions on the meshes are independently updated, followed by the exchange of boundary value information. Fine-grained parallelism can be exploited within each zone. BT-MZ has uneven-sized zones necessitating a need for load balancing to achieve efficient parallel execution. The benchmark can be executed with different problem sizes (or classes), differing in how the number and size of zones are defined. We used the C version of BT-MZ whereby tasks are generated by multiple threads using `omp for` using `-O2 -LNO` optimization flags.

BOTS Sort The Sort application from the Barcelona OpenMP Tasks Suite [2] is a variation on a mergesort algorithm that sorts a random permutation of n numbers. By recursively dividing an array into four parts, each part can be sorted in parallel. It then merges the first two parts and the last two parts in parallel. As each chunk reaches a small enough size, the sort and merge phases switch to a serial quicksort to increase task granularity. An insertion sort is used for very small array to avoid quicksort's overhead. In our evaluation we used the 16-core Nehalem machine for all Sort executions.

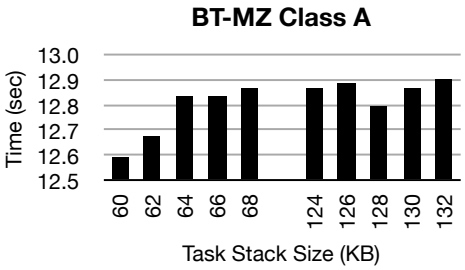
5.1 Impact of Task Stack Size

As mentioned previously, the amount of memory allocated for a task can have a significant impact on performance. The default task stack size of 64kB is a bit arbitrary and may not suit some applications. The minimum task stack size for the Class A and C problem sizes (determined experimentally) of BT-MZ to execute with OpenUH is 60kB and for Class B it is 74kB. Evalutions on both the 16-core Nehalem and the 8-core Opteron reveal that when this value is adjusted at runtime using the `OMP_TASK_STACK_SIZE` environment variable, the class A and C problem sizes show better performance with a task stack size of 60kB, the minimum, while class B shows better performance with a size of 128kB.

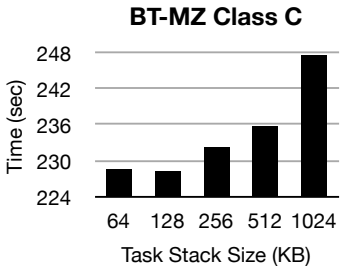
Figure 4 shows Class A and C results for the Opteron system. We speculate that 2^n stack sizes can yield better performance due to cache alignment. The best of the observed times for Class A and C with 60kB stack size may be due to the ability for the 6MB L3 cache to hold more tasks. A 6MB cache can hold 96 64kB tasks or 102 60kB tasks, about a 6% difference. There is an obvious tradeoff between optimizing the memory footprint and execution time. For the Class A execution on the 8-core Opteron, there is a nearly 3% increase in execution time when choosing a 256kB stack size instead of 60kB. While the naïve choice for task size may be a power of two, this choice is not necessarily the best. The ability to adjust this value at runtime removes the need to re-compile to change the stack size and would enable autotuning systems to dynamically adjust the runtime environment.



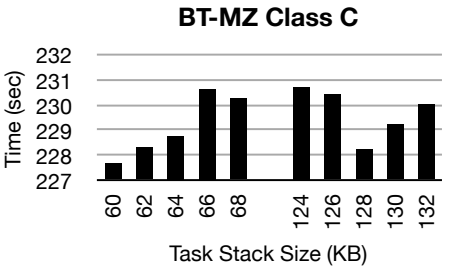
(a) Stack sizes in powers of 2.



(b) Fine-grain variations of stack size near "sweet spots".



(c) Stack sizes in powers of 2.



(d) Fine-grain variations of stack size near "sweet spots".

Fig. 4. Execution times for NPB BT-MZ with various task stack sizes on 8-core Opteron.

5.2 Impact of Work Stealing Policy

The policy employed for work stealing can also affect the performance of the runtime. By using a deque, our implementation is able to steal tasks from the

end of the queue opposite from which the owning thread accesses its tasks. If tasks are stolen and enqueued at the same end, the data structure acts like a stack. This may lead to contention among threads for access to the stack and impacts performance. Figure 5 shows the difference in performance for choosing a deque and a stack for task storage on the BT-MZ benchmark application on the Altix UV. The deque often outperforms the stack in for Class A and Class B tests performed for lower thread counts. However, this is not the case with Class C, where the stack performs as well as or better than the deque. Note that results on the Altix UV had higher variance than the smaller platforms used, possibly due to more noticeable NUMA effects as no binding was used.

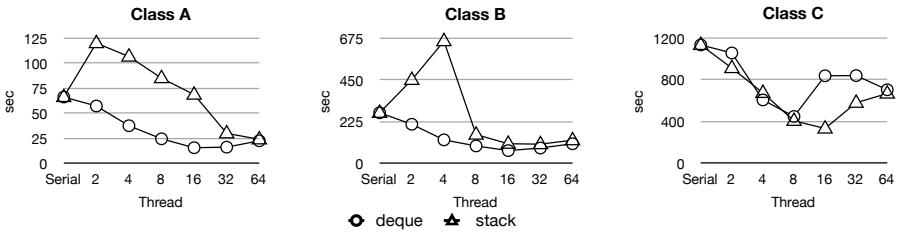


Fig. 5. Average execution times for the NPB BT-MZ for deques vs. stacks on Altix-UV

5.3 Impact of Task Creation Conditions

As shown previously [9], the decision to execute a task immediately or place it on the queue can greatly impact program performance. The OpenUH runtime system provides the ability to make these decisions at runtime. By altering the `OMP_TASK_CREATE_COND` environment variable for the execution of Sort, it is possible to quickly determine whether certain values yield better performance. Figure 6 shows speedup for Sort using different task create conditions. Default values were used for each except for *numtasks*, which uses a value of six times the size of the current thread team. See Section 4.3 for details on these conditions.

Of particular interest for Sort is the difference provided by create conditions and the sensitivity to problem size. Using default values, Sort with a 128MB problem size had an average execution time of 1.948 seconds. Using a *depth* task create condition with a default value (a limit of 3 levels deep in the task graph), the execution time drops to 1.686 seconds, or about 13% faster. However, for the 32MB problem size, use of *depthmod* with its default value provides an average execution time of 0.907 seconds versus 0.923 seconds for only a 2% improvement. This indicates that the problem size for this application has an effect on the performance of various runtime adjustments. All timings have been rounded to three decimal places. Keep in mind that only one environment variable was

changed for each runtime adjustment. We have yet to examine the effects of using multiple adjustments at runtime, like changing the task stack size and task creation condition for the same execution.

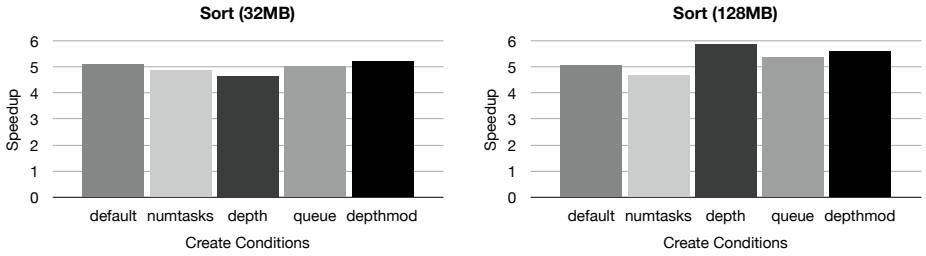


Fig. 6. Speedup for Sort using varying task creation conditions using 16 threads on 16-core Nehalem.

6 Related Work

A respectable body of work exists on task parallelism for languages [14], [11], [5] and runtime systems [12], [3]. These efforts have targeted both distributed memory systems and shared memory systems. More recent work with task parallelism includes Cilk [13], Sequoia [10], Intel's workqueing [24], and the High Productivity Computing Systems (HPCS) experimental languages Chapel [1] and X10 [23]. Libraries such as Intel's Thread Building Blocks (TBB), Microsoft Parallel Patterns Library, and PFunc [17] also support task parallelism.

Cilk [25] is a multithreaded parallel programming language developed at MIT as a parallel extension to C for expressing task-level parallelism. The runtime system uses a work-stealing scheduler under a work-first principle. A newly created task is immediately executed to minimize the overhead of computation while the creator-task is suspended. An idle thread can steal a task from another thread in the system. This achieves a "breadth-first theft, depth-first work" scheduling policy with minimal overhead while providing good data locality.

Before tasking was included in the OpenMP 3.0 specification, Intel extended the OpenMP API to allow dynamic task generation with their *workqueuing* model [24]. A single thread enqueues encountered tasks defined within a `taskq` block, while the other threads in the team participate in dequeuing the work from the queue. An implicit barrier at the completion of a `taskq` block ensures that all the tasks specified inside the block have finished execution.

Both Cilk and Intel's workqueuing models were influential in the design of OpenMP tasks. The Nanos group has contributed a great deal to OpenMP and its tasking model, including its design [4] and frameworks for the evaluation and testing of its various implementations [9], [8], [7].

7 Conclusions and Future Work

We have discussed various considerations for designing runtime systems that support OpenMP tasks and presented the design of our implementation in the OpenUH runtime system. We also showed how the runtime behavior can be customized by users, via environment variables, to fine tune code execution and presented some performance results. It is important to choose an appropriate stack size for a task to minimize memory footprint while improving execution time. Providing an adequate number of tasks without exceeding a performance-dampening threshold can be accomplished through varying queue size and volume. The choice of work stealing policy is not always obvious as using a stack may prove better than a queue for some application/system combinations. While application performance is known to be sensitive to architecture, operating system, and problem size, we have demonstrated that by fine tuning certain runtime variables, performance can be improved. It is likely that auto-tuning systems could adjust combinations of these values to provide a runtime environment customized to particular applications.

Prior work in our research group showed that all OpenMP programs, not just those written with tasks, can be represented and executed as a task graph [27]. It also showed the importance of attempting to co-locate tasks in order to make the best use of cache. We are currently exploring the provision of a purely task-based runtime as an alternative to our existing RTL. We plan to explore compiler optimizations for tasks, and opportunities for the compiler to pass useful information to the runtime for enhanced scheduling of tasks. Information such as tasks' data access patterns could help the RTL make more intelligent scheduling decisions. With heterogenous architectures becoming more prevalent in high performance computing systems, tasks are being considered as a vehicle for identifying code that is suitable for acceleration. We are investigating heuristics for automatically mapping tasks to various kinds of hardware.

Acknowlegements

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0833201 and Grant No. CCF-0917285, and the Texas Space Grant Consortium.

A special thanks to Haoqiang H. Jin from NASA Ames Research for help with the NPB BT-MZ implementation used here. As always, we are indebted to the HPCTools research group at the University of Houston for their help and collaboration.

References

1. *Chapel Specification 0.795*. April 2010.
2. Barcelona OpenMP Task Suite. `http://nanos.ac.upc.edu/content/barcelona-openmp-task-suite`, January 2011.

3. C. Augonnet, S. Thibault, and R. Namyst. StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. 2010.
4. Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in OpenMP. In *Proceedings of IWOMP 2007*, June, 2007.
5. Barbara Chapman, Piyush Mehrotra, John Van Rosendale, and Hans Zima. A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism. Technical Report 94-18, ICASE, MS 132C, NASA Langley Research Center, 1994.
6. Barbara M. Chapman, Lei Huang, Haoqiang Jin, Gabriele Jost, and Bronis R. de Supinski. Toward enhancing OpenMP's work-sharing directives. In *Europar 2006*, pages 645–654, 2006.
7. A. Duran, X. Teruel, R. Ferrer, et al. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 ICPP*, pages 124–131, 2009.
8. Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 36:1–36:11, 2008.
9. Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP task scheduling strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *Proceedings of the 4th IWOMP*, volume 5004/2010, pages 100–110, May 2008.
10. K. Fatahalian, D.R. Horn, T.J. Knight, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83. ACM, 2006.
11. I. Foster. Task parallelism and high-performance languages. *The Data Parallel Programming Model*, pages 179–196, 1996.
12. I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.
13. M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.
14. T. Gross, D.R. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, 1994.
15. O. Hernandez, R.C. Nanjegowda, B. Chapman, V. Bui, and R. Kufrin. Open Source Software Support for the OpenMP Runtime API for Profiling. In *ICPPW'09.*, pages 130–137. IEEE, 2009.
16. O. Hernandez, F. Song, B. Chapman, et al. Performance instrumentation and compiler optimizations for MPI/OpenMP applications. In *Second International Workshop on OpenMP*, 2006.
17. Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. PFunc: modern task parallelism for modern high performance computing. In *SC '09*, pages 43:1–43:11, New York, NY, USA, 2009. ACM.
18. M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.
19. Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: An optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, January 2006.
20. David Libenzi. Portable coroutine library. `http://www.xmailserver.org/libpcl.html`.

21. R. Nanjegowda, O. Hernandez, B. Chapman, and H. Jin. Scalability evaluation of barrier algorithms for OpenMP. *Evolving OpenMP in an Age of Extreme Parallelism*, pages 42–52, 2009.
22. The OpenUH Compiler Project. `http://www.cs.uh.edu/~openuh`, 2011.
23. Vijay Saraswat. Report on the experimental language X10 version 2.0.4. Technical report, IBM, June 2010.
24. E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Peterson. Compiler support of the workqueing execution model for Intel SMP architectures. In *EWOMP*, 2002.
25. Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.1 Reference Manual*, 2000.
26. R.F. Van der Wijngaart and H. Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Advanced Supercomputer (NAS) Division NASA Ames Research Center, 2003.
27. Tien-Hsiung Weng and Barbara Chapman. Implementing OpenMP using dataflow execution model for data locality and efficient parallel execution. In *Proceedings of the 7th workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-7)*. IEEE Press, 2002.