SAND2011-3299C

The Chapel Tasking Layer Over Qthreads

Kyle B. Wheeler, *Sandia National Laboratories*[*] *and*
Richard C. Murphy, *Sandia National Laboratories and*
Dylan Stark, *Sandia National Laboratories and*
Bradford L. Chamberlain, *Cray Inc.*[†]

ABSTRACT: *This paper describes the applicability of the third-party qthread lightweight threading library for implementing the tasking layer for Chapel applications on conventional multisocket multicore computing platforms. A collection of Chapel benchmark codes were used to demonstrate the correctness of the qthread implementation and the performance gain provided by using an optimized threading/tasking layer. The experience of porting Chapel to use qthreads also provides insights into additional requirements imposed by a lightweight user-level threading library, some of which have already been integrated into Chapel, and others that are posed here as open issues for future work. The initial performance results indicate an immediate performance benefit from using qthreads over the native multithreading support in Chapel. Both task and data parallel applications benefit from lower overheads in thread management. Future work on improved synchronization semantics are likely to further increase the efficiency of the qthreads implementation.*

KEYWORDS: Chapel, lightweight, threading, tasks

1. Introduction

It is increasingly recognized that, in order to obtain power and performance scalability, future hardware architectures will provide large amounts of parallelism. Taking full advantage of this parallelism requires an ability to specify the parallelism at multiple levels within a program. However, parallel programming is also widely recognized to be a difficult problem, and the set of programmers who can effectively leverage parallelism is a small fraction of those who are effective sequential programmers. Addressing the expressibility and programmability challenges are problems of wide interest.

Chapel is a new parallel programming language being developed by Cray Inc. as part of DARPA's High Productivity Computing System program (HPCS). One of its main motivating themes includes support for general parallel programming — data parallelism, task parallelism, concurrent programming, and arbitrary nestings of these styles. It also adopts a *multiresolution language design* in which higher-level features like arrays and data parallel loops are implemented in terms of lower-level features like classes and task parallelism. To this end, having a good implementation of Chapel's task parallel concepts is crucial since all parallelism is built in terms of it.

Task parallelism, in this case, refers not to the task/data parallelism distinction, but to the idea of a user-level threading concept, wherein tasks that can be executed in parallel are relatively short-lived and are created and destroyed rapidly. To maximize performance, applications must not only find parallel work, but must also match the amount of parallel work expressed to the available hardware. This lat-

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

[†]This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

ter task, however, is one best carried out by a runtime rather than the application itself.

Qthreads is a new lightweight threading, or tasking, library being developed by Sandia National Laboratories. The Qthreads runtime is designed to support dynamic programming and performance features not typically seen in either OpenMP or MPI systems. Parallel work is specified and the Qthreads runtime maps the work onto available hardware resources.

By comparing Qthreads dynamic mapping of tasks to hardware against the default "FIFO" scheduling mechanism of the Chapel runtime, an accurate picture of the benefits of the Qthread model can be obtained. In task parallelism situations, where `cobegin` is used, Qthreads can outperform the FIFO tasking layer by as much as 45%. In data parallelism situations, where `forall` and `coforall` are used, Qthreads can outperform the FIFO tasking layer by as much as 30%. Further work is planned to improve synchronization performance and eliminate additional bottlenecks.

Qthreads is described in more detail in Section 2. It is followed by a discussion of the Chapel tasking layer in Section 3. A discussion of the difficulties in mapping the Chapel tasking layer to the Qthreads API on single-node systems is in Section 4 and on multi-node systems is in Section 5. The results of our performance experiments are in Section 7.

2. Qthreads

Qthreads [4] is a cross-platform general purpose parallel runtime designed to support lightweight threading and synchronization within a flexible integrated locality framework. Qthreads directly supports programming with lightweight threads and a variety of synchronization methods, including both non-blocking atomic operations and potentially blocking full/empty bit (FEB) operations.

The Qthreads lightweight threading concept is intended to match future hardware threading environments more closely than existing concepts in three crucial aspects: anonymity, introspectable limited resources, and inherent localization. Unlike heavyweight threads, these threads do not support expensive features like per-thread identifiers, per-thread signal vectors, or preemptive multitasking. The thread scheduler in Qthreads presumes a cooperative-multitasking approach, which provides the flexibility to run threads in locations most convenient to the scheduler and the code. There are two scheduling regimes within qthreads: the single-threaded location mode, which does not use work-stealing, and the multi-threaded hierarchical location mode, which uses a shared work-queue between multiple workers in a single location and work-stealing between locations.

Blocking synchronization, such as when performing a FEB operation, triggers a user-space context switch. This context switch is done via function calls without trapping into the kernel, and therefore does not require saving as much state as preemptive context switches—such as signal masks and the full set of registers. This technique allows threads to process largely uninterrupted until data is needed that is not yet available, and allows the scheduler to attempt to hide communication latency by switching tasks when data is needed. Logically, this only hides communication latencies that take longer than a context switch.

3. Chapel Tasking Layer

Like many implementations of higher-level languages, the Chapel [2] compiler is implemented by compiling Chapel source code down to standard C. This permits the Chapel compiler to focus on high-level transformations and optimizations while leaving platform-specific targeting and optimizations to the native C compiler on each platform. Most of the lower-level code required to execute Chapel is implemented using Chapel's runtime libraries which are also implemented in C and then linked to the generated code.

The Chapel runtime libraries are organized as a number of *sub-interfaces*, each of which implements a specific subset of functionality such as communication, task management, memory management, or timing routines. Each sub-interface is designed such that several distinct implementations can be supplied as long as each supports the interface's semantics. An end-user can select from among the implementation options via an environment variable.

As an example, Chapel's task management layer defaults to `fifo`, a heavyweight but portable implementation that maps each task to a distinct POSIX thread (*pthread*). The work described in this paper adds a new lighter-weight tasking implementation that can be selected by setting the `CHPL_TASKS` environment variable to `qthreads`.

Chapel's task management sub-interface has two main responsibilities: The first is to implement the tasks that are generated by Chapel's `begin`, `cobegin`, and `coforall` statements; the second is to implement the full/empty semantics required to implement Chapel's *synchronization variables* which are the primary means of inter-task synchronization.

More specifically, the task interface must supply calls for:

Startup/Teardown: Initialize the task layer for program start-up and finalize it for program teardown;

Create Singleton Tasks: Used to implement Chapel's unstructured `begin` statements;

Create and Execute Task Lists: Used to implement Chapel's structured `cobegin` and `coforall` statements;

Synchronization: Used to implement the full/empty semantics of Chapel's synchronization variables;

Task Control: Functions such as yielding the processor or sleeping;

Queries: To optionally support queries about the number of tasks or threads in various states (running, blocked, etc.)

4. Single Locale Challenges

The first step in adapting Chapel's runtime to use qthreads as its tasking library was to get basic single-locale execution to work. The Chapel tasking layer conveniently provides a relatively simple header file of functions necessary for full functionality. Providing shim implementations of the expected functions is a relatively simple exercise, but exposed unexpected semantic issues. The work represented in this section is reflected in Chapel release 1.3.0[1].

4.1. Startup and Teardown

The major challenge here was that the Chapel tasking interface did not specify what operations were permitted before initializing the tasking library and after shutting down the tasking layer. All previous tasking layers had used native pthread constructs for synchronization and therefore were not sensitive to uses of synchronization variable that occurred prior to initializing the tasking layer or after tearing it down. Since qthreads' synchronization variables are less native, it held Chapel to a higher standard, requiring the program startup/teardown to be reordered to ensure that all task-based synchronization variables were used within the active scope of the tasking layer. This re-ordering involved re-architecting some components of the Chapel runtime to avoid relying on task synchronization in contexts where tasks are not permitted. Furthermore, the tasking interface has implicit semantics that are non-obvious, such as which functions may be called without initializing the tasking layer. As a result of integrating with qthreads, the interface was made more strict, forbidding the use of *any* tasking layer functions without initializing the tasking layer.

4.2. Unsupported Behavior

In addition to the implicit semantics of the tasking layer interface, there are a few semantics that qthreads does not support. In particular, the Chapel tasking interface assumes the existence of a limit on the number of operating system kernel-level threads. Qthreads, however, only allows the number of kernel.level threads to be specified or, if unspecified, will choose a number based on the number of currently available processing cores. In most cases, this is not a problem: the default Chapel limit is 255, and most systems don't have that many processing cores. However, if running on a system where the available processing cores exceeds the Chapel

[1] `http://sourceforge.net/projects/chapel/files/chapel/1.3.0/chapel-1.3.0.tar.gz/download`

limit, correct behavior is difficult to achieve because the situation cannot be detected before an excessive number of kernel threads have already been spawned. It is possible to then either abort or shutdown the qthread library and reinitialize, but both violate the Chapel-specified thread limit before correcting.

4.3. Remaining Problems

The most significant remaining difficulty is dealing with stack space limits. Tasking libraries of all sorts have two basic options with regard to stack space: either allow tasks to grow their stack as necessary at the cost of significant overhead to provide for detecting and correcting stack overruns or set fixed stack sizes that must not be violated. Qthreads exposes this problem frequently, since it assumes particularly small (4k) default stack sizes. The result is that codes can either segfault or silently corrupt memory when they run off the end of their stacks. Because the Chapel compiler does not currently have a way to estimate the amount of stack space that a given code will require per task, correct execution often requires the guess-and-check method of selecting a sufficient amount of stack space. This issue is a challenge for virtually all parallel compilers and associated runtimes, particularly when dealing with multiple ABI specifications in heterogeneous environments, and is not peculiar to Chapel and qthreads.

5. Multi-Locale Challenges

The second part of getting Chapel to use qthreads as its tasking library was to get multi-locale execution to work. Multi-locale behavior uses a communication layer — most commonly, GASNet [1] — which the tasking layer must inter-operate with.

5.1. Communication

The communication layer requires the ability to make blocking system calls in order to both send and wait for network traffic. Blocking system calls require special handling in user-level threading/tasking libraries because a blocking system call necessarily stops the kernel-level thread, which means it cannot participate in computation or processing user-level threads/tasks. For GASNet, the simplest solution was to establish a dedicated progress thread to ensure that GASNet operations operate independently of the task-layer's computation.

The Chapel runtime system automatically allocates a progress thread for GASNet on the first locale, but for all subsequent locales, the main execution thread is considered the GASNet progress thread, which means that the tasking library cannot take ownership of the main execution thread. To work around this, the qthread library needed to be initialized from a separate thread, which required careful bookkeeping to ensure that the same thread is used for both starting up and shutting down the tasking layer.

In the future, the creation of the GASNet progress thread will be a function provided by the tasking layer, to ensure that they can work together as efficiently as possible.

5.2. External Task Operations

One of the requirements of the communication progress thread is that it must be able to spawn tasks and use tasking synchronization primitives, despite not being a "task" itself. This required some workarounds within qthreads to allow external kernel-level threads to block on task-based synchronization primitives. This was accomplished by treating synchronization calls originating from outside the library as task spawn calls. The task that is spawned serves as a proxy for the external thread, using pthread mutexes to cause the external thread to block until the proxy task releases it.

6. Future Work

Synchronization is an interesting example of a mismatch between tasking layer assumptions and tasking library implementation. The Chapel tasking interface presupposes that the tasking library only provides mutex-like synchronization primitives, and uses this mutex semantic to implement the full/empty-style synchronization that the Chapel language's `sync`

variables require. In general, this is a reasonable assumption; while the use of full/empty semantics in the language stemmed from the DARPA Cascade project architecture, commodity architectures do not support native full/empty synchronization. The current approach is designed for generality and portability.

Some tasking implementations, however, such as qthreads and the MTA backend (which requires specialized hardware), have their own implementations of full/empty synchronization that can be quite fast. In order to support the semantics of the Chapel tasking interface, both the qthreads and MTA backends are required to use full/empty synchronization to provide mutex-like synchronization, which is then used to implement full/empty semantics. This mismatch in interface assumptions about the available synchronization semantics creates a great deal of overhead around synchronization operations.

It is possible to modify the Chapel runtime tasking layer interface to allow the tasking layer to implement the sync variable semantics directly, thus enabling the use of hardware primitives or new ideas about efficient sync variable implementation within the tasking layer. This would greatly improve synchronization efficiency, but may have some costs. One option to support high-speed full/empty synchronization is to use qthread `syncvar_t` variables, which keep state within the 64-bit word, thereby limiting the number of available bits. It may be useful to allow the user to choose how many bits are absolutely required to a greater degree than Chapel currently allows, and use different synchronization mechanisms based on those requirements. Another potential challenge includes considering the effect of compiler-introduced copies of synchronization variables since identity matters in some tasking libraries and copies may not only introduce extra synchronization operations, but may not transfer waiters across copies. The details of enabling such a direct implementation, however, require some creative thinking, and as such remains an open problem.

7. Performance

To demonstrate both the functionality and the performance impact of using the qthread tasking layer instead of the default FIFO tasking layer implementation, several benchmarks were run. Two kinds of parallelism are examined. First, task parallelism, as expressed by the quicksort and tree-exploration benchmarks provide as part of the Chapel distribution, is used to demonstrate the relative overhead of the Qthread tasking layer. Then data parallelism, as used in the HPCC benchmark suite [3], also part of the Chapel distribution, is used to demonstrate the broad applicability of the Qthread tasking layer's performance. As only the STREAM and RandomAccess benchmarks are described as "scalable" in the Chapel documentation, only results from those benchmarks are presented.

The results presented here were obtained on dual-socket twelve-core 3.33GHz Intel Xeon X5680 system (with HyperThreading and power management turned off). Chapel was compiled with GCC 4.1.2. All tests were done using a single Chapel locale.

7.1. QuickSort

This benchmark is a basic naïve implementation of a parallel quicksort. The benchmark picks a pivot value and partitions around the pivot value in serial and then uses a `cobegin` statement to spawn tasks to recursively execute quicksort on each partition. The benchmark has the capacity to serialize, via a recursive depth threshold, rather than spawn the maximum number of tasks. However, to demonstrate the behavior of the tasking library, the threshold for the results presented in Figure 1 was set high enough so as to never serialize.

The FIFO tasking library ranges from 182% to 71% slower than the Qthread tasking library in this benchmark, trending toward 80% slower as the problem size increases. With an array of 2^{28} elements, the FIFO implementation executed in 86.6 seconds, while the Qthread implementation took only 46.8 seconds.

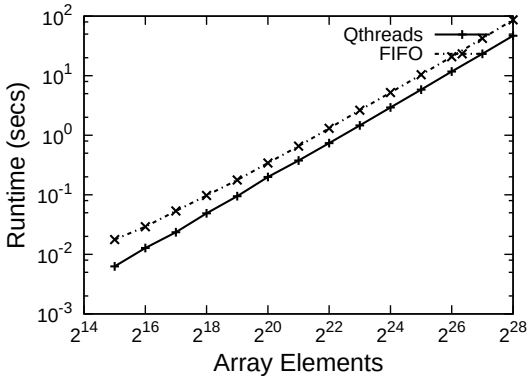

Figure 1: Chapel QuickSort

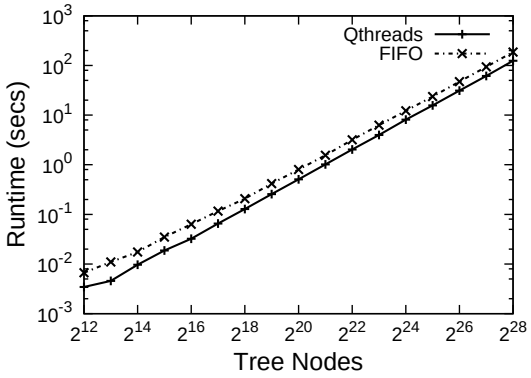

Figure 2: Chapel Tree Exploration

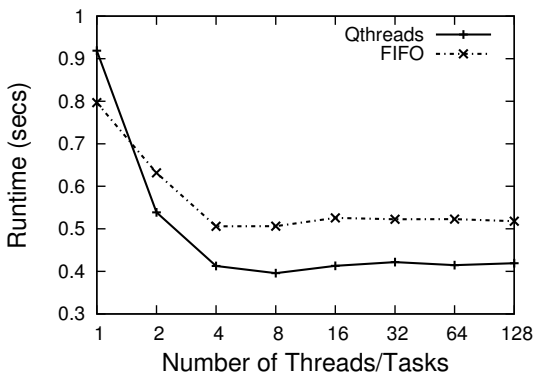

Figure 3: HPCC STREAM

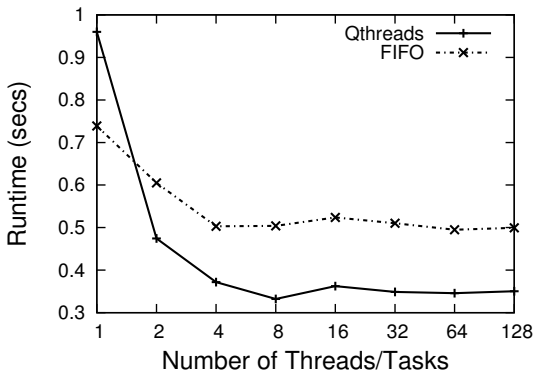

Figure 4: HPCC STREAM-EP

7.2. Tree Exploration

This benchmark constructs a binary tree in parallel where each node in the tree has a unique ID. It then iterates over the tree to compute the sum of the ID's in parallel using `cobegin`.

Figure 2 illustrates the performance benefit of using the Qthread tasking layer. The FIFO tasking library ranges from 80% to 50% slower than the Qthread tasking layer, trending toward 50% as the problem size increases. With a tree of 2^{28} nodes, the FIFO implementation executed in 186 seconds, while the Qthread implementation took only 124 seconds.

7.3. STREAM

This benchmark is a relatively simple program designed to measure sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. There is also an "embarrassingly parallel" (EP) version that does not do inter-locale communication.

Figures 3 and 4 represent results run on a 6GB problem size. The results demonstrate that the use of the Qthread tasking layer provides significant performance benefits over the FIFO tasking layer, when more than a single thread/task was used. The FIFO tasking implementation provides performance that is approximately 25% slower than the Qthreads tasking implementation in the STREAM benchmark, and

approximately 45% slower in the STREAM-EP benchmark. Some of the performance credit may be due to Qthread's automatic CPU-pinning, as well as to its lightweight task spawning.

Interestingly, the EP variant of the benchmark, in Figure 4, shows a larger performance improvement than the non-EP variant. This is a result of the synchronization overhead discussed in Section 6. The EP variant of the benchmark uses a `coforall` to spawn tasks to all of the available locales — in this case, there's only one — and each of those tasks then uses a parallel `forall` to implement the body of the benchmark. This is different from the non-EP variant in that the non-EP variant does not use the `coforall`. The performance difference is a consequence of the fact that the main thread is not a task within the tasking library, and synchronization between that non-task and tasks is not as efficient as synchronization between tasks. In the non-EP version, synchronization to wait for all the parallel work in the `forall` must use a combination of task spawns and pthread mutexes to allow the main task to wait, however in the EP version, it can use the inter-task synchronization directly.

7.4. RandomAccess

This benchmark measures the rate of random integer updates to memory; it is sometimes referred to as the GUPS benchmark. It is designed to stress the memory system of the machine by rendering the data cache almost useless. As such, one would expect that the performance of the tasking layer would be relatively minimal.

Figure 5 largely bears out that expectation. The Qthread tasking layer provides some small performance benefit for multiple tasks, but the benefit is relatively small—around 15%. As the number of tasks increases, the percentage of the memory bandwidth in use increases, which is the ultimate performance bottleneck for this benchmark.

8. Conclusion

The most important result of this paper is that the Chapel tasking layer can indeed be successfully run on top of third-party tasking libraries, like Qthreads. There are unexpected semantic mis-matches that have the potential for creating artificial performance problems. Beyond the basic synchronization issues, there are optimization concerns, including the behavior of tasks within a standard multi-socket multi-core locale that need to be addresses for optimum performance. However, Chapel proves to be a particularly powerful framework for expressing task parallelism, and benefits from a true task-parallel runtime like Qthreads.

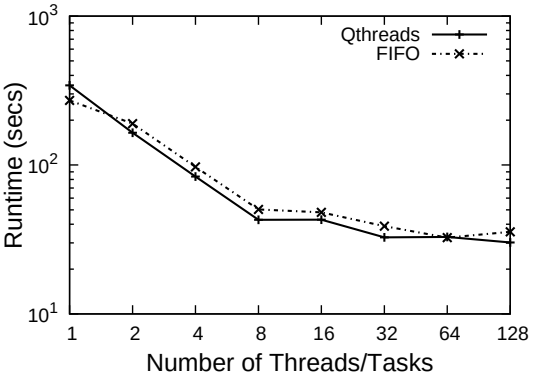

Figure 5: HPCC RandomAccess

References

[1] Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, University of California Berkeley, October 2002.

[2] David Callahan, Brad L. Chamberlain, and Hans P. Zima. The Cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60. IEEE, April 2004.

[3] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on*

Supercomputing, SC '06, New York, NY, USA, 2006. ACM.

[4] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS '08: Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*, pages 1–8. MTAAP '08, IEEE Computer Society Press, April 2008.