Qthreads: An API for Programming with Millions of Lightweight Threads

Kyle B. Wheeler
University of Notre Dame
South Bend, Indiana, USA
kwheeler@cse.nd.edu

Richard C. Murphy
Sandia National Laboratories*
Albuquerque, New Mexico, USA
rcmurphy@sandia.gov

Douglas Thain
University of Notre Dame
South Bend, Indiana, USA
dthain@cse.nd.edu

Abstract

Large scale hardware-supported multithreading, an attractive means of increasing computational power, benefits significantly from low per-thread costs. Hardware support for lightweight threads is a developing area of research. Each architecture with such support provides a unique interface, hindering development for them and comparisons between them. A portable abstraction that provides basic lightweight thread control and synchronization primitives is needed. Such an abstraction would assist in exploring both the architectural needs of large scale threading and the semantic power of existing languages. Managing thread resources is a problem that must be addressed if massive parallelism is to be popularized. The qthread abstraction enables development of large-scale multithreading applications on commodity architectures. This paper introduces the qthread API and its Unix implementation, discusses resource management, and presents performance results from the HPCCG benchmark.

1. Introduction

Lightweight threading primitives, crucial to large scale multithreading, are typically either platform dependent or compiler-dependent. Generic programmer-visible multithreading interfaces, such as pthreads, were designed for "reasonable" numbers of threads—less than one hundred or so. In large-scale multithreading situations, the features and guarantees provided by these interfaces prevent them from scaling to "unreasonable" numbers of threads (a million or more), necessary for multithreaded teraflop-scale problems.

Parallel execution has largely existed two different worlds: the world of the very large, where programmers explicitly create parallel threads of execution, and the world of the very small, where processors extract parallelism from serial instruction streams. Recent hardware architectural research has investigated lightweight threading and programmer-defined large scale shared-memory parallelism. The lightweight threading concept allows exposure of greater potential parallelism, increasing performance via greater hardware parallelism. The Cray XMT [7], with the Threadstorm CPU architecture, avoids memory dependency stalls by switching among 128 concurrent threads. XMT systems support between over 8000 processors. To maximize throughput, the programmer must provide at least 128 threads per processor, or over 1,024,000 threads.

Taking advantage of large-scale parallel systems with current parallel programming APIs requires significant computational and memory overhead. For example, standard POSIX threads must be able to receive signals, which either requires an OS representation of every thread or requires user-level signal multiplexering [14]. Threads in a large-scale multithreading context often need only a few bytes of stack (if any) and do not require the ability to receive signals. Some architectures, such as the Processor-in-Memory (PIM) designs [5, 18, 23], suggest threads that are merely a few instructions included in the thread's context.

While hardware-based lightweight threading constructs are important developments, the methods for exposing such parallelism to the programmer are platform-specific and typically rely either on custom compilers [3, 4, 7, 10, 11, 26], entirely new languages [1, 6, 8, 13], or have architectural limitations that cannot scale to millions of threads [14, 22]. This makes useful comparisons between architectures difficult. With a standard way of expressing parallelism that can be used with existing compilers, comparing cross-platform algorithms becomes convenient. For example, the MPI standard allows a programmer to create a parallel application that is portable to any system providing an MPI library, and different systems can be compared with the same code on each system. Development and study of large-scale multithreaded applications is limited because of the platform-specific nature of the available interfaces. Having a portable large-scale multithreading interface al-

*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.

lows application development on commodity hardware that can exploit the resources available on large-scale systems.

Lightweight threading requires a lightweight synchronization model [9]. The model used by the Cray XMT and PIM designs, pioneered by the Denelcor HEP [15], uses full/empty bits (FEBs). This technique marks each word in memory with a "full" or "empty" state, allows programs to wait for either state, and makes the state change atomically with the word's contents. This technique can be implemented directly in hardware, as it is in the XMT. Alternatives include ADA-like protected records [24] and fork-sync [4], which lack a clear hardware analog.

This paper discusses programming models' impact on efficient multithreading and the resource management necessary for those models. It introduces the qthread lightweight threading API and its Unix implementation. The API is designed to be a lightweight threading standard for current and future architectures. The Unix implementation is a proof of concept that provides a basis for developing applications for large scale multithreaded architectures.

2. Recursive threaded programming models and resource management

Managing large numbers of threads requires managing per-thread resources, even if those requirements are low. This management can affect whether multithreaded applications run to completion and whether they execute faster than an equivalent serial implementation. The worst consequence of poor management is deadlock: if more threads are needed than resources are available, and reclaiming thread resources depends on spawning more threads, the system cannot make forward progress.

2.1. Parallel programming models

An illustrative example of the effect the programming model on resource management is the trivial problem of summing integers in an array. A serial solution is trivial: start at the beginning, and tally each sequential number until the end of the array. There are at least three parallel execution models that could compute the sum. They can be referred to as the recursive tree model, the equal distribution model, and the lagging-loop model.

A recursive tree solution to summing the numbers in an array is simple to program: divide the array in half, and spawn two threads to sum up both halves. Each thread does the same until its array has only one value, whereupon the thread returns that value. Thread that spawned threads wait for their children to return and return the sum of their values. This technique is parallel, but uses a large amount of state. At any point, most of the threads are not doing useful work. While convenient, this is a wasteful technique.

The equal distribution solution is also simple: divide the array equally among all of the available processors and spawn a thread for each. Each thread must sum its segment serially and return the result. The parent thread sums the return values. This technique is efficient because it matches the needed parallelism to the available parallelism, and the processors do minimal communication. However, equal distribution is not particularly tolerant of other load imbalances: execution is as slow as the slowest thread.

The lagging-loop model relies upon arbitrary workload divisions. It breaks the array into small chunks and spawns a thread for each chunk. Each thread sums its chunk and then waits for the preceding thread (if any) to return an answer before combining the sum and returning its own total. Eventually the parent thread will do the same with the last chunk. This model is more efficient than the tree model, and the number of threads depends on the chunk size. The increased number of threads makes it more tolerant of load imbalances, but has more overhead.

2.2. Handling resource exhaustion

These methods differ in the way resources must be managed to guarantee forward progress. Whenever new thread is requested, one of four things can be done:

1. Execute the function inline.
2. Create a new thread.
3. Create an record that will become a thread later.
4. Block until sufficient resources are available.

In a large enough parallel program, eventually the resources will run out. Requests for new threads must either block until resources become available or must fail and let the program handle the problem.

Blocking to wait for resources to become available affects each parallel model differently. The lagging loop method works well with blocking requests, because the spawned threads don't rely on spawning more threads. When these threads complete, their resources may be reused, and deadlock is easily avoided. The equal distribution method has a similar advantage. However, because it avoids using more than the minimum number of threads, it does not cope as well with load imbalances.

The recursive tree method gathers a lot of state quickly and slowly releases it, making the method particularly susceptible to resource exhaustion deadlock, where all running threads are blocked spawning more threads. In order to guarantee forward progress, resources must be reserved when threads spawn and threads must execute serially when reservation fails. The minimum state that must be reserved is the amount necessary get to the bottom of the recursive tree serially. Thus, if there are only enough resources for a single depth-first exploration of the tree, recursion may only

occur serially. If there are enough resources for two serial explorations of the tree, the tree may be divided into two segments to be explored in parallel, and so forth. Once resource reservation fails, only a serial traversal of the recursive tree may be performed. Thus, blocking for resources is a poor behavior for a recursive tree as forward progress cannot be assured.

Such an algorithm is only possible when the maximum depth of the recursive tree is known. If the depth is unknown, then sufficient resources for a serial execution cannot be reserved. *Any* resources reserved for a parallel execution could prevent the serial recursive tree from completing.

It is worth noting that a threading library can only be responsible for the resources necessary for basic thread state. Additional state required during recursion has the potential to cause deadlock and must be managed similarly.

3. Application programming interface

The qthread API provides several key features:

- Large scale lightweight multithreading support
- Access to or emulation of lightweight synchronization
- Basic thread-resource management
- Source-level compatibility between platforms
- A library-based API, forgoing custom compilers

The qthread API maximizes portability to architectures supporting lightweight threads and synchronization primitives by providing a stable interface to the programmer. Because architectures and operating systems supporting lightweight threading are difficult to obtain, initial analysis of the API's performance and usability studies commodity architectures such as Itanium and PowerPC processors.

The qthread API consists of three components: the core lightweight thread command set, a set of commands for resource-limit-aware threads ("futures"), and an interface for basic threaded loops. Qthreads have a restricted stack size, and provide a locking scheme based on the full/empty bit concept. The API provides alternate threads, called "futures", which are created as resources are available.

One of the likely features of machines supporting large scale multithreading is non-uniform memory access (NUMA). To take advantage of NUMA systems, they must be described to the library, in the form of "shepherds," which define memory locality.

3.1. Basic thread control

The API is an anonymous threading interface. Threads, once created, cannot be controlled by other threads. However, they can provide FEB-protected return values so that a thread can easily wait for another. FEBs do not require polling, which is discouraged as the library does not guarantee preemptive scheduling.

Threads are assigned to one of several "shepherds" at creation. A shepherd is a grouping construct. The number of shepherds is defined when the library is initialized. In an environment supporting traveling threads, shepherds allow threads to identify their location. Shepherds may correspond to nodes in the system, memory regions, or protection domains. In the Unix implementation, a shepherd is managed by at least one pthread which executes qthreads. It is worth noting that this hierarchical thread structure, particular to the Unix implementation (not inherent to the API), is not new but rather useful for mapping threads to mobility domains. A similar strategy was used by the Cray X-MP [30], as well as Cilk [4] and other threading models.

Only two functions are required for creating threads: qthread_init(shep), which initializes the library with shep shepherds; and qthread_fork(func,arg,ret), which creates a thread to perform the equivalent of $*ret = func(arg)$. The API also provides mutex-style and FEB-style locking functions. Using synchronization external to the qthread library is not encouraged, as that prevents the library from making scheduling decisions.

The mutex operations are qthread_lock(addr) and qthread_unlock(addr). The FEB semantics are more complex, with functions to manipulate the FEB state in a non-blocking way (qthread_empty(addr) and qthread_fill(addr)), as well as blocking reads and blocking writes. The blocking read functions wait for a given address to be full and then copy the contents of that address elsewhere. One (qthread_readFF()) will leave the address marked full, the other (qthread_readFE()) will then mark the address empty. There are also two write actions. Both will fill the address being written, but one (qthread_writeEF()) will wait for the address to be empty first, while the other (qthread_writeF()) won't. Using the two synchronization techniques on the same addresses at the same time produces undefined behavior, as they may be implemented using the same underlying mechanism.

3.2. Futures

Though the API has no built-in limits on the number of threads, thread creation may fail due to memory limits or other system-specific limits. "Futures" are threads that allow the programmer to set limits on the number of futures that may exist. The library tracks the futures that exist, and stalls attempts to create too many. Once a future exits, a future waiting to be created is spawned and its parent thread is unblocked. The futures API has its own initialization function (future_init(limit)) to specify the maximum number of futures per shepherd, and a way to create a future (future_fork(func,arg,ret)) that behaves like qthread_fork().

3.3. Threaded loops and utility functions

The qthread API includes several threaded loop interfaces, built on the core threading components. Both C++-based templated loops and C-based loops are provided. Several utility functions are also included as examples. These utility functions are relatively simple, such as summing all numbers in an array, finding the maximum value, or sorting an array.

There are two parallel loop behaviors: one spawns a separate thread for each iteration of the loop, and the other uses an equal distribution technique. The functions that provide one thread per iteration are qt_loop() and qt_loop_future(), using either qthreads or futures, respectively. The functions that use equal distribution are qt_loop_balance() and qt_loop_balance_future(). A variant of these, qt_loopaccum_balance(), allows iterations to return a value that is collected ("accumulated").

The qt_loop() functions take arguments start, stop, stride, func, and argptr. They behave like this loop:

```
unsigned int i;
for (i = start; i < stop; i += stride) {
    func(NULL, argptr);
}
```

The qt_loop_balance() functions, since they distribute the iteration space, require a function that takes its iteration space as an argument. Thus, while it behaves similar to qt_loop(), it requires that its func argument point to a function structured like this:

```
void func(qthread_t *me, const size_t startat,
          const size_t stopat, void *arg) {
    for (size_t i = startat; i < stopat; i++)
        /* do work */
}
```

The qt_loopaccum_balance() functions require an accumulation function so that return values can be gathered. The function behaves similar to the following loop:

```
unsigned int i;
for (i = start; i < stop; i++) {
    func(NULL, argptr, tmp);
    accumulate(retval, tmp);
}
```

Similar to the qt_loop_balance() function, it uses the equal distribution technique. The func function must store its return value in tmp, which is then given to the accumulate function to gather and store in retval.

4. Performance

The design of the qthread API is based around two primary goals: efficiency in handling large numbers of threads and portability to large-scale multithreaded architectures. The implementation of the API discussed in this section is the Unix implementation, which is for POSIX-compatible Unix-like systems running on traditional CPUs, such as PowerPC, x86, and IA-64 architectures. In this environment, the qthread library relies on pthreads to allow multiple threads to run in parallel. Lightweight threads are created as a processor context and a small (4k) stack. These lightweight threads are executed by the pthreads. Context-switching between qthreads is performed as necessary rather than on an interrupt basis. For performance, memory is pooled in shepherd-specific structures, allowing shepherds to operate independently.

Without hardware support, FEB locks are emulated via a central hash table. This table is a bottleneck that would not exist on a system with hardware lightweight synchronization support. However, the FEB semantics still allow applications to exploit asynchrony even when using a centralized implementation of those semantics.

4.1. Benchmarks

To demonstrate qthread's advantages, six micro-benchmarks were designed and tested using both pthreads and qthreads. The algorithms of both implementations are identical, with the exception that one uses qthreads as the basic unit of threading and the other uses pthreads. The benchmarks are as follows:

1. Ten threads atomically increment a shared counter one million times each
2. 1,000 threads lock and unlock a shared mutex ten thousand times each
3. Ten threads lock and unlock 1 million mutexes
4. Ten threads spinlock and unlock ten mutexes 100 times
5. Create and execute 1 million threads in blocks of 200 with at most 400 concurrently executing threads
6. Create and execute 1 million concurrent threads

Figure 1 illustrates the difference between using qthreads and pthreads on a 1.3Ghz dual-processor PowerPC G5 with 2GB of RAM. Figure 2 illustrates the same on a 48-node 1.5Ghz Itanium Altix with 64GB of RAM. Both systems used the Native Posix Thread Library Linux Pthread implementation. The bars in each chart in Figure 1 are, from left to right, the pthread implementation, the qthread implementation with a single shepherd, with two shepherds, and with four shepherds. The bars in each chart in Figure 2 are, from left to right, the pthread implementation, the qthread implementation with a single shepherd, with 16 shepherds, with 48 shepherds, and with 128 shepherds.

In Figures 1(a) and 2(a), using pthreads is outperformed by qthreads because qthreads uses a hardware-based atomic increment while pthreads is forced to rely on a mutex. Because of contention, additional shepherds do not improve the qthread performance but rather decrease it slightly. Since the qthread locking implementation is built with pthread mutexes, it cannot compete with raw pthread mutexes for speed, as illustrated in Figures 1(b), 2(b), 1(c),

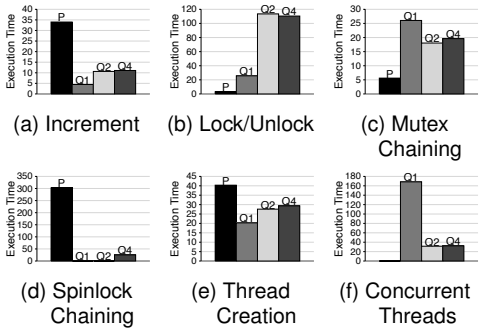

Figure 1: Microbenchmarks on a dual PPC

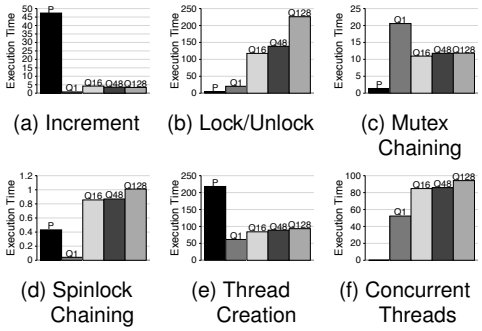

Figure 2: Microbenchmarks on a 48-node Altix

and 2(c). This is a detail that would likely not be true on a system that had hardware support for FEBs, and would be significantly improved with a better centralized data structure, such as a lock-free hash table. Because of the qthread library's simple scheduler, it outperforms pthreads when using spinlocks and a low number of shepherds, as illustrated in Figure 1(d). The impact of the scheduler is demonstrated by larger numbers of shepherds (Figure 2(d)).

The pthread library was incapable of more than several hundred concurrent threads—requesting too many threads deadlocked the kernel (Figures 1(f) and 2(f)). A benchmark was designed that worked within pthreads' limitations by allowing a maximum of 400 concurrent threads. Threads are spawned in blocks of 200, and after each block, threads are joined until there are only 200 outstanding before spawning a new block of 200 threads. In this benchmark, Figures 1(e) and 2(e), pthreads performs more closely qthreads—on the PowerPC system, it is only a factor of two more expensive.

5. Application development

Development of software that realistically takes advantage of lightweight threading is important to research, but difficult to achieve due to the lack of lightweight threading interfaces. To evaluate the performance potential of the API and how difficult it is to integrate into existing code, two representative applications were considered. First, a parallel quicksort algorithm was analyzed and modified to fit the qthread model. Secondly, a small parallel benchmark was modified to use qthreads.

5.1. Quicksort

Portability of an API does not free the programmer completely from taking the hardware into consideration when designing an algorithm. There are features of alternative threading environments that the qthread API does not emulate, such as the hashed memory design found in the Cray MTA-2. Memory addresses in the MTA-2 are distributed throughout the machine at word boundaries. When dividing work amongst several threads on the MTA-2, the boundaries of the work regions can be fine-grained without significant loss of performance. Conventional processors, on the other hand, assume that memory within page boundaries are all contiguous. Thus, conventional cache designs reward programs that allow an entire page to reside in a single processor's cache, and limit the degree to which tasks can be divided among multiple processors without paying a heavy cache coherency penalty.

An example wherein the granularity of data distribution can be crucial to performance is a parallel quicksort algorithm. In any quicksort algorithm, there are two phases: first the array is partitioned into two segments around a "pivot" point, and then both segments are sorted independently. Sorting the segments independently is relatively easy, but partitioning the array in parallel is more complex. On the MTA-2, elements of the array to be partitioned can be divided up among each thread without regard to the location of the elements. On conventional processors, however, that behavior is very likely to result in multiple processors transferring the same cache-line or memory page between processors. Constantly sending the same memory back and forth between processors prevents the parallel algorithm from exploiting the capabilities of multiple processors.

The qthread library includes an implementation of the quicksort algorithm that avoids contention problems by ensuring that work chunks are always at least the size of a page. This avoids cache-line competition between processors while still exploiting the parallel computational power of all available processors on sufficiently large arrays. Figure 3 illustrates the scalability of the qthread-based quicksort implementation, and compares its performance to the libc `qsort()` function. This benchmark sorts an array of one billion double-precision floating point numbers on a 48-node SGI Altix SMP with 1.5Ghz Itanium processors.

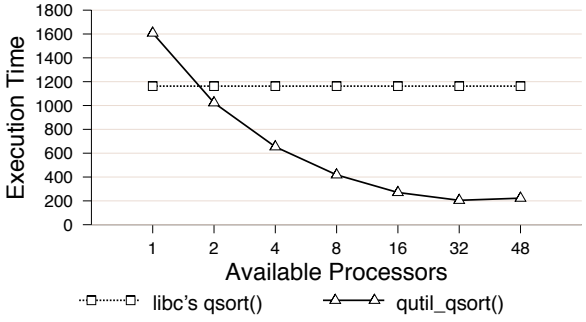

Figure 3: qutil_qsort() and libc's qsort()

5.2. High performance computing conjugate gradient benchmark

The qthread API makes parallelizing ordinary serial code simple. As a demonstration of its capabilities, the HPCCG benchmark from the Mantevo project [20] was parallelized with the Qloop interface of the qthread library. The HPCCG program is a conjugate gradient benchmarking code for a 3-D chimney domain, largely based on code in the Trilinos[21] solver package. The code relies largely upon tight loops where every iteration of the loop is essentially independent of every other iteration. With simple modifications to the code structure, the serial implementation of HPCCG was transformed into multithreaded code. As illustrated in Figure 4, the parallelization is able to scale well. Results are presented using strong scaling with a uniform 75x75x1024 domain on a 48-node SGI Altix SMP. The SGI MPI results are presented to 48 processes, or one process per CPU, as further results would over-subscribe the processors, which generally underperforms with SGI MPI.

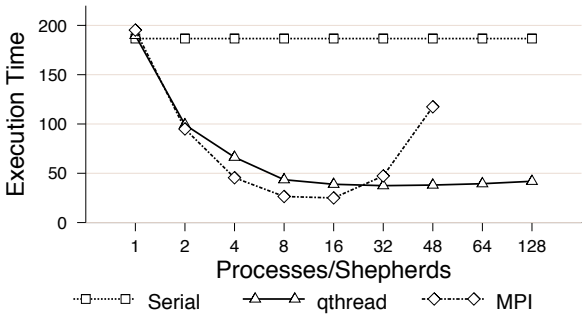

Figure 4: HPCCG on a 48-Node SGI Altix SMP

One of the features of the HPCCG benchmark is that it comes with an optimized MPI implementation. The MPI implementation, using SGI's MPI library, is entirely different from the qthread implementation and does not use shepherds. The qthread and MPI implementations scale approximately equally well up to about sixteen nodes. Beyond sixteen nodes however, MPI begins to behave very badly. At the same time, the qthread implementation's execution time does not change significantly.

Upon analysis of the MPI code, the poor performance of the MPI implementation is caused by MPI_Allreduce() in one of the main functions of the code. While this takes almost 18.9% of execution time with eight MPI processes, it takes 84.1% of the execution time with 48 MPI processes. While it is tempting to simply blame the problem on a bad implementation of MPI_Allreduce(), it is probably more valid to examine the difference between the qthread and MPI implementations. The qthread implementation performs the same computation as the MPI_Allreduce(), but rather than require all nodes to come to the same point before the reduction can be computed and distributed, the computation is performed as the component data becomes available from the threads returning, the computational threads can exit, and other threads scheduled on the shepherds can proceed. The qthread implementation exposes the asynchronous nature of the whole benchmark, while the MPI implementation does not. This asynchrony is revealed even though the Unix implementation of the qthread library relies upon centralized synchronization, and would likely provide further improvement on a real massively parallel architecture.

6. Related work

Lightweight threading models generally fit one of two descriptions: they either require a special compiler or they aren't sufficiently designed for large-scale threading (or both). For example, Python stackless threads [28] provide extremely lightweight threads. Putting aside issues of usability, which is a significant issue with stackless threads, the interface allows for no method of applying data parallelism to the stackless threads: a thread may be scheduled on any processor. Many other threading models, from nanothreads [26] to OpenMP [11], lack a sufficient means of allowing the programmer to specify locality. This becomes a significant issue as machines get larger and memory access becomes non-uniform [31]. Languages such as Chapel [8] and X10 [6], or modifications to existing languages such as UPC [13] and Cilk [4], that require special compilers are interesting and allow for better parallel semantic expressiveness than approaches based in adding library calls to existing languages. However, such models not only break compatibility with existing large codebases but also do not provide for strong comparisons between architectures. Some threading models, such as Cilk, use a fork-and-join style of synchronization that, while semantically convenient, does not allow for as fine-grained control over communication

between threads as the FEB-based model, which allows individual load and store instructions to be synchronized.

The drawbacks of heavyweight, kernel-supported threading such as pthreads are well-known [2], leading to the development of a plethora of user-level threading models. The GNU Portable Threads [14], for example, allow a programmer to use user-level threading on any system that supports the full C standard library. It uses a signal stack to allow the subthreads to receive signals, which limits its ability to scale. Coroutines [29] are another model that allow for virtual threading even in a serial-execution-only environment, by specifying alternative contexts that get used at specific times. Coroutines can be viewed as the most basic form of cooperative multitasking, though they can use more synchronization points than just context-switch barriers when run in an actual parallel context. One of the more powerful details of coroutines is that generally one routine specifies which routine gets processing time next, which is behavior that can also be obtained when using continuations [19, 27]. Continuations, in the most broad sense, are primarily a way of minimizing state during blocking operations. When using heavyweight threads, whenever are a thread does something that causes it to stop executing, its full context—local variables, a full set of processor registers, and the program counter—are saved so that when the thread becomes unblocked it may continue as if it had not blocked. A continuation allows the programmer to specify that when a thread blocks it exits, and that unblocking causes a new thread to be created with specific arguments, thus requiring the programmer to save any necessary state to memory while any unnecessary state can be disposed of. Protothreads [12, 17] and Python stackless threads [28], by contrast, assert that outside of CPU context there is no thread-specific state (i.e. "stack") at all. This makes them extremely lightweight but limits the flexibility (at most, only one of them can call a function), which has repercussions for ease-of-use. User-level threading models can be further enhanced with careful kernel modification [25] to enable convenient support of many of the features of heavyweight kernel threads, such as signals, advanced scheduling conventions, and even limited software interrupt handling.

The problem of resource exhaustion due to excessive parallelism was considered by Goldstein et. al. [16]. Their "lazy-threads" concept addresses the issue that most threading models conflate logical parallelism and actual parallelism. This semantics problem often requires that programmers tailor the expression of parallelism to the available parallelism, thereby forcing programmers to either require too much overhead in low-parallelism situations or forgo the full use of parallelism in high-parallelism situations.

The qthread API combines many of the advantages of other threading models. The API allows parallelism to be expressed independently of the parallelism used, much like Goldstein's lazy-thread approach. However, rather than require a customized compiler, the qthread API does this within a library that uses two different categories of threads: thread workers (shepherds) and stateful thread work units (qthreads). This technique, while convenient, has overhead that a compiler-based optional-inlining method would not: every qthread requires memory. This overhead can be limited arbitrarily through the use of futures, which is a powerful abstraction to express resource limitations without limiting the expressibility of inherent algorithmic parallelism.

7. Future work

Much work still remains in development of the qthread API. A demonstration of how well the API maps to the APIs of existing large scale architectures, such as the Cray MTA/XMT systems, is important to reinforce the claim of portability. Custom implementations for other architectures would be useful, if not crucial.

Along similar lines, development of additional benchmarks to demonstrate the potential of the qthread API and large-scale multithreading would be useful for studying the effect of large-scale multithreading on standard algorithms. The behavior and scalability of such benchmarks will provide guidance for the development of new large-scale multithreading architectures.

Thread migration is an important detail of large scale multithreading environments. The qthread API addresses this with the shepherd concept, but the details of mapping shepherds to real systems requires additional study. For example, shepherds may need to have limits enforced upon them, such as CPU-pinning, in some situations. The effect of such limitations on multithreaded application performance is unknown, and deserving of further study.

8. Conclusions

Large scale computation of the sort performed by common computational libraries can benefit significantly from low-cost threading, as demonstrated here. Lightweight threading with hardware support is a developing area of research that the qthread library assists in exploring while simultaneously providing a solid platform for lighter-weight threading on common operating systems. It provides basic lightweight thread control and synchronization primitives in a way that is portable to existing highly parallel architectures as well as to future and potential architectures. Because the API can provide scalable performance on existing platforms, it allows study and modeling of the behavior of large scale parallel scientific applications for the purposes of developing and refining such parallel architectures.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0β edition, March 2007.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support fot the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[3] A. Begel, J. MacDonald, and M. Shilman. Picothreads: Lightweight threads in java.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[5] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A low cost, multithreaded processing-in-memory system. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 16–22, New York, NY, USA, 2004. ACM Press.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[7] Cray XMT platforrm. http://www.cray.com/products/xmt/index.html, October 2007.

[8] Cray Inc., Seattle, WA 98104. *Chapel Language Specification*, 0.750 edition.

[9] H.-E. Crusader. High-end computing needs radical programming change. *HPCWire*, 13(37), September 2004.

[10] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Tama compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, 1993.

[11] L. Dagum and R. Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[13] T. El-Ghazawi and L. Smith. Upc: unified parallel c. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 2006. ACM.

[14] R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[15] M. C. Gilliland, B. J. Smith, and W. Calvert. Hep - a semaphore-synchronized multiprocessor with central control (heterogeneous element processor). In *Summer Computer Simulation Conference*, pages 57–62, Washington, D.C., July 1976.

[16] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[17] B. Gu, Y. Kim, J. Heo, and Y. Cho. Shared-stack cooperative threads. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied Computing*, pages 1181–1186, New York, NY, USA, 2007. ACM Press.

[18] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1999. ACM Press.

[19] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 293–298, New York, NY, USA, 1984. ACM Press.

[20] M. Heroux. Mantevo. http://software.sandia.gov/mantevo/index.html, December 2007.

[21] M. Heroux, R. Bartlett, V. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, et al. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[22] Institute of Electrical and Electronics Engineers. *IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1)*, 1990.

[23] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

[24] C. D. Locke, T. J. Mesler, and D. R. Vogel. Replacing passive tasks with ada9x protected records. *Ada Letters*, XIII(2):91–96, 1993.

[25] B. D. marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operaing Systems Principles*, pages 110–121, New York, NY, USA, 1991. ACM Press.

[26] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A library implementation of the nano-threads programming model. In *Euro-Par, Vol. II*, pages 644–649, 1996.

[27] A. Meyer and J. G. Riecke. Continuations may be unreasonable. In *LFP '88: Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 63–71, New York, NY, USA, 1988. ACM Press.

[28] Stackless python. http://www.stackless.org, January 2008.

[29] S. E. Sevcik. An analysis of uses of coroutines. Master's thesis, 1976.

[30] F. Szelényi and W. E. Nagel. A comparison of parallel processing on Cray X-MP and IBM 3090 VF multiprocessors. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 271–282, New York, NY, USA, 1989. ACM.

[31] J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs, 2001.