

LIGHTWEIGHT THREADING FOR ARCHITECTURAL DESIGN RESEARCH

KYLE B. WHEELER* AND RICHARD C. MURPHY†

Abstract. Increased parallelism is a powerful method for increasing computational power, and one way that such parallelism is expressed is as threads. Large scale threading benefits significantly from lower per-thread overhead, and lightweight threading with hardware support is a developing area of research. Each architecture with support for lightweight threading provides a different interface to the programmer, making comparisons between them difficult. As such there is a need for an abstraction that 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 to assist in exploring both the architectural needs of large scale threading and the extent to which threading can be expressed in existing programming languages. This paper introduces the qthread API and its Unix implementation.

1. Introduction. Modern supercomputers have largely taken the route of parallel computation to achieve increased computational power. Use of parallelism to increase execution speed has largely operated in two entirely 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 attempt to extract parallelism from streams of instructions. Parallelism has been exploited at low levels of the architecture, including hardware-based techniques such as Simultaneous Multi-Threading and out-of-order execution, both of which identify work that is sufficiently unrelated to be performed at the same time. However, programmers have only been given interfaces such as MPI [5], Pthreads [8], and more recently OpenMP [4] to express the parallelism of their algorithms. This is a significant semantic disconnect that limits or prevents full exploitation of the available parallelism.

As the amount of available lower-level hardware parallelism increases, it becomes more and more necessary to allow the programmer to express parallel execution simply, and without the overhead usually associated with standard parallel programming models.

Support for lightweight threading in hardware is a rapidly developing field, and current portable programming interfaces do not expose these newly available capabilities to the programmer for use on architectures that have them.

The qthread API is designed to change that. Designed to be sufficiently similar to existing threading libraries so as to be simple to use, the qthread API does not have many features that heavyweight threading uses its large amount of thread state to provide. Qthreads, by contrast, have very small amounts of thread-specific state, and provide initialization-free synchronization methods based around the Full/Empty Bit (FEB) synchronization technique [10] that interact directly with their simple scheduling mechanism. While being lightweight, the qthread API is designed to be simple to port to alternative and experimental architectures, such as Cray's Multi-Threaded Architecture (MTA) [1] and the developing Processor-In-Memory (PIM) [9, 3, 6] designs.

2. Design Goals. The qthread API was designed to maximize portability to experimental architectures supporting lightweight threads and unusual synchronization primitives while providing a stable interface to the programmer for using these

*University of Notre Dame, kwheeler@cse.nd.edu

†Sandia National Laboratories, rcmurphy@sandia.gov

lightweight threads. Lightweight threads, and thus qthreads, have inherent restrictions on their stack size, and provide an easily emulated universal locking scheme based on the Full/Empty Bit (FEB) concept [10]. For control of resources, qthreads can use a definition of a thread, called a “future”, that is only created when there are resources available for it.

In addition to portability, performance is important to the design of the API. The API’s goal is to add as little overhead as possible to experimental architectural threading primitives while maintaining its generic nature. For example, to reduce overhead, qthreads are considered mostly anonymous, and cannot be affected directly by other threads—there is no way to cancel a currently running thread, and threads are not expected to respond to signals. The sole way to communicate reliably between threads is to use shared data structures protected by the locking functions provided by the API. Waiting for threads to finish, for example, is best done by waiting for them to return a value, which allows all qthreads to release their resources when they exit even if no one is waiting for them.

The generic design of the API is the primary contribution of this work, though the Unix implementation of the API is important to demonstrate the workability and functionality of the system. The development of the implementation informed the development of the API design. For example, since lightweight threads have limited stack space, a function was needed in the API that would allow the programmer to monitor this limited resource, for debugging and runtime decision-making. Both the API and the implementation, with the exception of the futurelib, were designed and implemented entirely by the authors of this paper. The futurelib component was designed and implemented by Megan Vance.

2.1. Basic Design. The qthread API consists of four components: the core lightweight thread interaction command set, a set of commands for resource-limit-aware threads (“futures”), a C++ interface for basic threaded loops, and a C-language interface for several varieties of basic threaded loops. The loop packages are built on top of the core thread control and resource-limit-aware threading components.

2.2. Basic Thread Control. The qthread library implements a nearly-anonymous threading interface. Threads, once created, cannot be controlled by other threads. However, they can provide return values which are protected by a synchronization method, such as an FEB, enabling a thread to wait for another thread to complete. When threads are waiting for a synchronization event—i.e. “blocked”—they are removed from the scheduling queues and are only available to be scheduled when they are unblocked.

Threads must assume that they are running in a cooperatively scheduled environment (although they may not be), and thus spin-locking rather than using one of the provided synchronization primitives is discouraged.

Lightweight threads are scheduled in the bailiwick of one of several “shepherds.” The number of shepherds is defined when the library is initialized. A shepherd is a grouping construct, or a thread mobility domain. Qthreads are assigned a shepherd when they are created and cannot move between shepherds. In the Unix version of the qthread library, a shepherd is a pthread responsible for organizing and executing the qthreads. In other qthread implementations, shepherds may be nodes in the system, memory regions, or protection domains.

The outline of the qthread thread-manipulation API is given below. For a complete specification, see the qthread library’s documentation:

`qthread_init (n)` Initialize the threading library, able to use `n` shepherds.

`qthread_fork(func, arg, ret)` Create a thread to run `func(self, arg)` and make it available for execution. Its return value will be stored in `*ret`. The thread may be created on any shepherd.

`qthread_fork_to(func, arg, ret, shep)` Create a thread and make it available for execution on the shepherd `shep`.

`qthread_self()` returns (`self`). Obtain a reference to the executing thread. This is primarily for use in avoiding extra lookups when doing locking; use of this reference by other threads is undefined.

`qthread_shep()` returns (`shepherd`). Discover which shepherd this thread has been assigned to.

`qthread_retloc()` returns (`addr`). Discover where the return value for this function will be stored.

`qthread_stackleft()` returns (`bytes`). Discover approximately how many bytes are available in the thread's stack.

`qthread_finalize()` Destroys all threads, releases all related memory.

The `qthread` API also provides functions to manipulate locking structures. These mimic both a generic mutex and full/empty bits, though the implementation of each may not have direct hardware support. The two categories of synchronization may be implemented using the same underlying mechanism, or may not, so using the two techniques on the same addresses at the same time results in undefined behavior. When these locking primitives do not have direct hardware support, they must be emulated using what locking primitives are available. The current Unix implementation implements them using `pthread` mutexes, and a 32-way striped mutex-protected hash table.

The outline of the `qthread` locking API is given below. For a complete specification, see the `qthread` library's documentation:

`qthread_lock(self, addr)` Obtain a lock on the address `addr`.

`qthread_unlock(self, addr)` Release the lock on the address `addr`.

`qthread_empty(self, addr)` Set the full/empty state of address `addr` to be empty.

`qthread_fill(self, addr)` Set the full/empty state of address `addr` to be full.

`qthread_readFE(self, dest, src)` Wait for `src` to become full, then copy the data in `*src` into `*dest` and set `src` to be empty. This is atomic.

`qthread_readFF(self, dest, src)` Wait for `src` to become full and then copy the data in `*src` into `*dest`. This is atomic.

`qthread_writeEF(self, dest, src)` Wait for `dest` to become empty and then copy the data in `*src` into `*dest` and set `dest` to be full. This is atomic.

`qthread_writeF(self, dest, src)` Copy the data in `*src` into `*dest` and set `dest` to be full. This is atomic.

`qthread_feb_status(addr)` returns (`status`). Discover the current full/empty state of `addr`.

`qthread_incr(addr, incr)` returns (`value`). Discover the current value stored at `addr` and add `incr` to it. May or may not use `qthread_lock()` and `qthread_unlock()`, depending on compile-time options and hardware/compiler support. This is atomic.

2.3. Futures. The `qthread` API has no built-in limitations on the number of threads spawned other than the amount of memory necessary for the threads' contexts. Memory allocation failures are the only limit. When confronted with the memory limits, applications have few options other than to wait and try again. To make memory management easier, a variant of `qthread` creation was created to allow the

programmer to set arbitrary limits on the number of threads that may exist at the same time. These resource-limit-aware threads are called “futures”. Attempting to create a future when the maximum number of futures already exist causes the creating thread to block until the future can be successfully created. The future-related API is a simple extension to the `qthread` thread-manipulation API:

- `future_init (limit)` Defines the maximum number of futures per shepherd to be `limit`.
- `future_fork (func, arg, ret)` Essentially identical to `qthread_fork()`, but creates a future and as such will block until the future has been created.
- `future_yield (self)` Causes a future to become merely a `qthread` temporarily. Useful for avoiding deadlock in some situations.
- `future_acquire (self)` Causes a `qthread` to become a future if it had previously yielded; may block if there are too many futures already.

The `qthread` API also includes some convenience functions, built on top of the core threading, locking, and futures functionality. These convenience functions generally provide threaded loops and the ability to perform simple actions over large sets of data. These convenience functions are separated into three categories: the `futurelib` (written by Megan Vance), the `Qloops`, and `Qutils`.

2.3.1. Futurelib. The `futurelib` is a template-based C++ interface to the `qthread` and futures primitives. The most important and basic functions in the `futurelib` are `mt_loop()`, for parallel iterations that do not return values, and `mt_loop_returns()`, for parallel iterations that do. The distinction between the two is not always obvious, and can often be treated as merely a programmer convenience. Both functions are implementations of a parallel for-loop. The `mt_loop()` function is used in a format like this:

```
mt_loop<... argtypelist ..., looptype>
    (function, ... arglist ..., startval, stopval, stepval);
```

This construction is relatively straightforward. The `function` argument specifies a function that will be used for each iteration of the loop. The iterations are defined as each having a number starting at `startval`, ending at `stopval`, incremented by `stepval` (which is optional, and defaults to 1). The `looptype` option specifies the kind of parallelism, and has four possible values:

- `mt_loop_traits :: Par` All iterations are created at once as `qthreads` and the `mt_loop()` function will not return until all iterations are finished.
- `mt_loop_traits :: ParNoJoin` Similar to `Par`, but does not wait for iterations to finish before returning flow control to the parent thread.
- `mt_loop_traits :: Future` Similar to `Par`, but uses futures instead of `qthreads`.
- `mt_loop_traits :: FutureNoJoin` Similar to `ParNoJoin`, but uses futures instead of `qthreads`.

The `argtypelist` in the `mt_loop()` construction is a list of conceptual types defining how the arguments to the iteration function will be handled. Each conceptual type corresponds to a single argument to the iteration function. Valid conceptual types are:

- Iterator** This corresponds to the integer value associated with each iteration, being a number between `startval` and `stopval`.
- ArrayPtr** This corresponds to an argument that is a pointer to an array. Each iteration function instance will be passed the value of `array[iteration]`.
- Ref** The corresponding argument will be passed to the iteration function as a reference.
- Val** The corresponding argument will be passed to the iteration function as a constant value (i.e. the same value will be passed to all iterations).

As an example, here is a simple loop:

```
for (int i = 0; i < 10; i++) {
    array[i] = i;
}
```

This simple loop could be threaded with the `futurelib` like so:

```
void assign(int &array_value, const int i) {
    array_value = i;
}

mt_loop<ArrayPtr, Iterator, mt_loop_traits::Par>
    (assign, array, 0, 0, 10);
```

Alternatively, the following code would achieve the same goal:

```
void assign(const int i, const int * a) {
    a[i] = i;
}

mt_loop<Iterator, Val, mt_loop_traits::Par>
    (assign, 0, array, 0, 10);
```

The `mt_loop_returns()` variant adds the specification of what to do with the return values. The pattern is like this:

```
mt_loop_returns<returnvaltype, ... argtypelist..., looptype>
    (retval, function, ... arglist..., startval, stopval, stepval);
```

The only difference is in the *returnvaltype* and the *retval*. The *returnvaltype* can be either an `ArrayPtr` or a `Collect`. If it is an `ArrayPtr`, each return value will be stored in a separate entry in the *retval* array, corresponding to the iteration. The parallel loop will behave similar to the following loop:

```
for (int i = startval; i < stopval; i += stepval) {
    retval[i] = function(... arglist...);
}
```

The `Collect` type is more interesting, and can be one of the following:

`Collect<mt_loop_traits::Add>` This sums all of the return values in parallel and stores the value in *retval*.

`Collect<mt_loop_traits::Sub>` This subtracts all of the return values in parallel and stores the value in *retval*. Note that the answer may be nondeterministic.

`Collect<mt_loop_traits::Mult>` This multiplies all of the return values in parallel and stores the value in *retval*.

`Collect<mt_loop_traits::Div>` This divides all of the return values in parallel and stores the value in *retval*. Note that the answer may be nondeterministic.

The accumulation functions occur as data is made available by iteration functions returning. Thus, adding or multiplying multiple integers is predictable, while doing the same with floating point numbers exposes the user to potential rounding problems due to the operation ordering. Use of `Collect<mt_loop_traits::Add>` is roughly equivalent to the following loop:

```
for (int i = startval; i < stopval; i += stepval) {
    retval += function(... arglist...);
}
```

2.3.2. Qloop, and Qutil. The qloop component provides futurelib-like functionality with a strictly C-language interface. Because of the inherent limitations of the C language, this interface is slightly more cumbersome than the futurelib’s C++ interface. The qloop component provides two alternative parallel loop behaviors: one that, like futurelib, spawns a separate qthread or future for each iteration, and the other which spawns one qthread for each shepherd and gives each thread a segment of the iteration-space to compute.

An outline of the qloop parallel for-loop interface is given below. For the complete specification, see the qthread documentation:

`qt_loop(start ,stop, stride ,func, argptr)` Spawns `func` as a qthread with the argument `argptr` for every iteration between `start` and `stop` with a stride of `stride`.

`qt_loop_future(start ,stop, stride ,func, argptr)` Similar to `qt_loop()`, but uses futures instead of qthreads.

`qt_loop_balance(start ,stop,func, argptr)` Spawns `func` as a qthread for each shepherd. Each instance of `func` is assigned a specific range of the iteration space between `start` and `stop` over which it can operate.

`qt_loop_balance_future(start ,stop,func, argptr)` Similar to `qt_loop_balance()`, but uses futures instead of qthreads.

`qt_loopaccum_balance(start ,stop, size ,out,func,arg, accfunc)` Similar to `qt_loop_balance()`, however each `func` may have a return value `size` bytes long that will be stored in `out`. The combination of the return values is controlled by the accumulator function `accfunc`.

As an example, `qt_loop()` behaves similarly to the following loop:

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

The `qt_loop_balance()` function behaves like the following loop. For simplicity, this example loop assumes that the number of iterations is evenly divisible by the number of shepherds, though `qt_loop_balance()` does not. Note that it has access to the number of shepherds, which is stored internally in the qthreads library:

```
int stride = (stop - start)/num_shepherds;
for (int i = start; i < stop; i += stride) {
    func(i, i+stride, argptr);
}
```

Additionally, qloop provides several simple utility functions that use the balanced loop interface to perform simple tasks, such as compute the sum of every element in an array. The qutil component also provides utility functions for performing these simple tasks, but implements them with a lagging-loop structure. Table 2.1 compares the equivalent functions for operating on arrays of doubles. Similar functions are available for integers and unsigned integers.

| Qloop | Qutil |
|-------------------------------|----------------------------------|
| <code>qt_double_min()</code> | <code>qutil_double_min()</code> |
| <code>qt_double_max()</code> | <code>qutil_double_max()</code> |
| <code>qt_double_prod()</code> | <code>qutil_double_prod()</code> |
| <code>qt_double_sum()</code> | <code>qutil_double_sum()</code> |

TABLE 2.1
Qloop compared to Qutil

Assuming that each shepherd maps to a single processor's ability to execute, the balanced loop design provided by `qt_loop_balance()` allows the computational power of a parallel system to be maximized with a minimum amount of overhead, but presumes that the iterations do not interact or depend on each other in any way, and does not compensate for additional threads that may also be executing.

The lagging-loop design used in Qutil functions spawns threads for fixed-size segments of the iteration space. The number of threads active at any given time is thus either limited only by the problem size, or limited by the imposed limit on the number of active futures. This technique cooperates with unrelated executing threads well, and handles interaction between threads better because there are typically more than one thread assigned to each shepherd, so when one blocks, other threads can execute. However, this behavior incurs more thread-management overhead, which varies by implementation. This loop design also requires a well-chosen segment size, to minimize thread overhead without reducing the parallelism to a point where it cannot cooperate with other unrelated threads sufficiently. In that sense, a balanced loop design is a special case of the lagging-loop design, where the library chooses the segment size to match the available shepherd-level parallelism.

3. Application Development. Development of software that realistically takes advantage of lightweight threading is important to research, but difficult to achieve, as there are few organizations willing to devote the time to develop large scale threaded applications for architectures that may never exist.

3.1. Cray's Multi-Threaded Architecture. Cray's MTA [1] is of particular interest to lightweight threading researchers, as it provides lightweight threads, fast locks, FEB support, and a toolchain to take advantage of them. The architecture even has some large-scale software available for it. In particular, the Multi-Threaded Graph Library (MTGL) [2] provides a good example of high-performance code written for an architecture that provides these capabilities.

The platform-specific features of the MTA are primarily accessed through the use of C-language pragmas that instruct the custom MTA compiler how to parallelize the code. Because MTA applications are so dependent on the compiler, porting such code to other architectures is rather difficult.

Because the qthread library provides similar features, the MTGL can be ported relatively easily to use the qthread API rather than the native MTA API, and once that is done, can be run on other architectures relatively easily, including any experimental architectures for which a qthread implementation has been developed.

3.2. High Performance Computing Conjugate Gradient Benchmark. The qthread API makes parallelizing ordinary serial code extremely simple. As a demonstration of its capabilities, the HPCCG benchmark written by Mike Heroux for Sandia National Labs was parallelized with the qloop interface of the qthread library. The HPCCG program is a simple conjugate gradient benchmarking code for a 3-D chimney domain, largely based on code in the Trilinos[7] solver package. The code relies primarily upon tight loops where every iteration of the loop is essentially independent of every other iteration. This type of code is ripe for parallelization. With simple modifications to the code structure, the serial execution of HPCCG was transformed into multithreaded code. As illustrated in Figure 3.1, the parallelization is able to scale well. The performance numbers in this graph were obtained by running the code on a 48-node SGI Altix SMP.

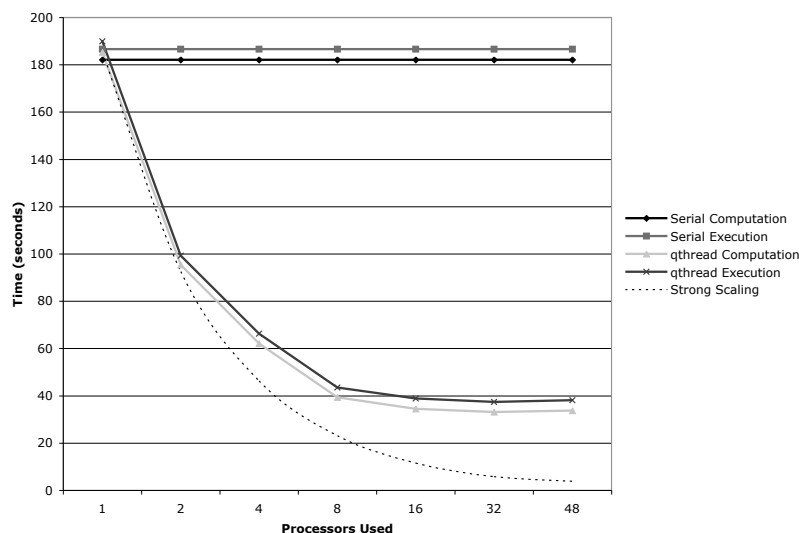


FIG. 3.1. HPCCG Benchmark on a 48-node SGI Altix SMP

4. 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 qthreads 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 be implemented and can provide MPI-like performance on existing platforms without a custom compiler, 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] Cray MTA-2 system - HPC technology initiatives, November 2006.
- [2] J. W. BERRY, B. A. HENDRICKSON, S. KAHAN, AND P. KONECNY, *Software and algorithms for graph queries on multithreaded architectures*, in Proceedings of the International Parallel & Distributed Processing Symposium, IEEE, 2007.
- [3] J. B. BROCKMAN, S. THOZIYOOR, S. K. KUNTZ, AND P. M. KOGGE, *A low cost, multithreaded processing-in-memory system*, in WPMI '04: Proceedings of the 3rd workshop on Memory performance issues, New York, NY, USA, 2004, ACM Press, pp. 16–22.
- [4] L. DAGUM AND R. MENON, *OpenMP: An industry-standard api for shared-memory programming*, IEEE Computational Science & Engineering, 5 (1998), pp. 46–55.
- [5] M. P. I. FORUM, *MPI: A message-passing interface standard*, Tech. Rep. UT-CS-94-230, 1994.
- [6] 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), New York, NY, USA, 1999, ACM Press, p. 57.
- [7] 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*, Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.
- [8] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, *IEEE Std 1003.1-1990: Portable*

- Operating Systems Interface (POSIX.1)*, 1990.
- [9] 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.
 - [10] C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *Efficient synchronization of multiprocessors with shared memory*, in PODC '86: Proceedings for the fifth annual ACM symposium on Principles of distributed computing, New York, NY, USA, 1986, ACM Press, pp. 218–228.