Lithe: Enabling Efficient Composition of Parallel Libraries

Heidi Pan
Massachusetts Institute of Technology
xoxo@csail.mit.edu

Benjamin Hindman Krste Asanović
University of California, Berkeley
{benh,krste}@eecs.berkeley.edu

Abstract

For the software industry to take advantage of multicore processors, we must allow programmers to arbitrarily compose parallel libraries without sacrificing performance. We argue that high-level task or thread abstractions and a common global scheduler cannot provide effective library composition. Instead, the operating system should expose unvirtualized processing resources that can be shared cooperatively between parallel libraries within an application. In this paper, we describe a system that standardizes and facilitates the exchange of these unvirtualized processing resources between libraries.

1 Introduction

Despite the rapid spread of parallel hardware, parallel software will not become commonplace without support for efficient composition of parallel libraries. Composability is essential for programmer productivity, allowing applications to be constructed from reusable libraries that can be developed and maintained independently, and possibly even written in different languages.

Unfortunately, when current parallel libraries are composed within the same application, they will generally interfere with each other and degrade overall performance. This is because, unlike sequential libraries, parallel libraries encapsulate resource management in addition to functionality. A parallel library decomposes computation into independent tasks that it executes using some abstraction of the hardware resources of the underlying machine, typically kernel-scheduled virtual threads. The threads used by various libraries within the application are then time-multiplexed by the operating system scheduler onto the *same* hardware resources, potentially causing cache interference, gratuitous synchronization contention, and other performance-degrading conflicts.

Current libraries supply ad hoc solutions to this composability problem. For example, Intel's Math Kernel Library (MKL) instructs its clients to call the sequential version of the library whenever it might be running in parallel with another part of the application [1]. This places a difficult burden on programmers who have to manually choose between different library implementations depending on the calling environment. Worse, a programmer writing a new parallel library that calls MKL will export the exact same problem to its users.

We claim the problem will not be solved by attempting to coerce all parallel libraries to map their computation to some universal task or thread abstraction, with the hope that some standard global scheduler (either at user-level or in the operating system) can map these to available hardware processors efficiently. First, there has been no agreement on the best parallel practices, as evidenced by the proliferation of new parallel languages and runtimes over the years. Second, it is also unlikely that a competitive general-purpose scheduler even exists, as they have been repeatedly outperformed by code optimizations that leverage domain or application-specific knowledge (e.g. [10, 7]). Finally, even if a competitive task or thread abstraction emerges, it is unlikely that the growing body of legacy parallel libraries will be rewritten to suit.

We argue that using a common high-level parallelism abstraction fundamentally impairs the composability and efficiency of parallel libraries, and that instead libraries should cooperatively share the underlying physical processing resources. In this way, the physical resources are never oversubscribed, and each library can customize its own mapping on to the parallel execution substrate.

Our proposal has two main components. First, we export a new low-level unvirtualized hardware thread abstraction, or *hart*, from the operating system to applications. Second, we provide a new resource management interface, *Lithe*, that defines how harts are transferred between parallel libraries within an application. Lithe is fully compatible with sequential code, and can be inserted underneath the runtimes of legacy parallel software environments to provide "bolt-on" composabil-

ity. Lithe also enables the construction of new parallel libraries using customized task representations and scheduling policies.

2 Towards a Better Processing Abstraction

Virtualized kernel threads have become the central abstraction in modern operating systems, providing a processing substrate for application programmers and the schedulable unit for the OS. In this section, we motivate our move towards unvirtualized hardware threads as the primary processing abstraction in a parallel system.

Historically, kernel threads were criticized for being too heavyweight for parallel programs. This gave rise to N:M style user-level thread libraries, which had acceptable performance provided the application eschewed blocking system calls (specifically I/O). Scheduler activations [2] were proposed to handle this particular deficiency of N:M user-level threading, but the approach was not widely adopted. Instead, kernel threads were improved to bridge the performance gap and are now the *de facto* choice on which to build modern parallel applications and libraries.

Unfortunately, because the OS performs oblivious multiplexing of virtualized kernel threads onto available physical processors, there is often serious interference between parallel libraries executing within an application. This makes it difficult for programmers to reason about performance and drastically reduces the utility of well-known optimizations such as software prefetching and cache blocking.

Instead of kernel threads, we argue an operating system should export a low-level unvirtualized hardware thread abstraction that we call a *hart*, a contraction of *hardware thread*. A hart abstracts a single physical processing engine, i.e., an entire single-threaded core or one hardware thread context in a multithreaded core (e.g. a SMT core [13]). As far as an application is concerned, a hart is continually "beating", i.e., advancing a program counter and executing code. Harts have a one-to-one mapping to a physical processing engine while the application is running, and hence cannot be oversubscribed. An application is allocated an initial hart and must request subsequent harts from the operating system, which is free to postpone, perhaps indefinitely, the granting of additional harts.

The hart abstraction is sufficiently low level that within an application a programmer need not be concerned with possible oversubscription. However, the hart abstraction is sufficiently high level that it enables an OS to do both space and time multiplexing of applications on a parallel system. For example, an operating system can space-multiplex applications by never allocating a processing engine to more than one hart of one application at a time. An operating system may time-multiplex applications by gang-scheduling all harts of one application onto the physical engines after swapping off all harts of another application. Combinations of both space and time multiplexing are also possible [11].

The set of harts allocated to an application is analogous to the virtual multiprocessor allocated to an application with scheduler activations. However, harts were designed to faithfully represent the underlying parallel hardware to prevent oversubscription, while scheduler activations were designed to handle the problems caused by blocking system calls in user-level threading. We are exploring how to best handle system calls with harts, either with an approach based on scheduler activations or by moving to a non-blocking system call interface.

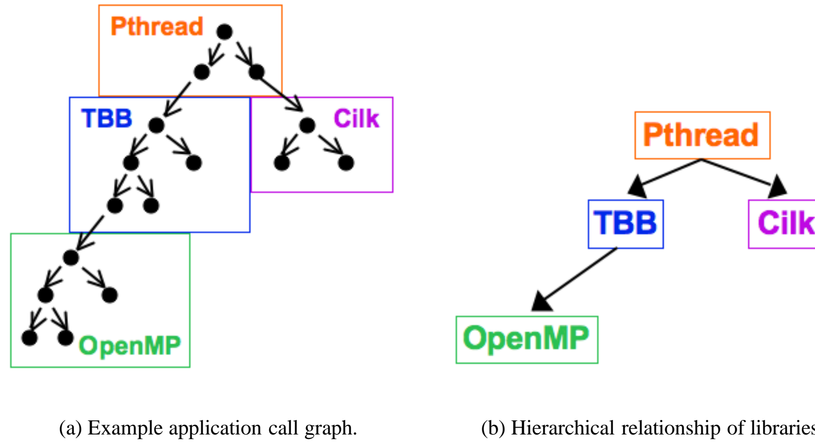
Although we believe future parallel operating systems should export some form of harts to applications instead of kernel threads [11], the concept of harts is still effective even without OS support. In fact, many parallel libraries and frameworks approximate harts today by never creating more threads than the number of processing engine in the underlying machine. Some operating systems even allow threads to be pinned to different processing engines so that the threads don't interfere. In essence, the hart abstraction is meant to capture this discipline, both within different libraries in an application and across multiple applications in a system.

3 Composable Hart Management

In this section, we present our key design decisions on how to share the finite number of harts allocated to an application among all the libraries within the application.

As a concrete example, consider an application containing libraries written using POSIX Threads (Pthreads), Cilk [3], Intel's Threading Building Blocks (TBB) [12], and OpenMP [4], with the call graph shown in Figure 1(a). The call graph emphasizes the hierarchy created by the caller/callee relationship of the libraries (Figure 1(b)). For example, the library written using Pthreads calls the library written using TBB which in turn calls the library written using OpenMP.

Our first design decision is that *callers should be responsible for providing harts to callees*. Interestingly, this is already the standard mechanism in the sequential world, where a caller implicitly grants its processing resource to a callee until the callee returns. The justification is that when a caller invokes a callee, it is asking it to perform work on the caller's behalf. More importantly, a routine that makes calls in parallel might exploit local knowledge (e.g., about a critical path) to prioritize giving harts to one callee over another. Such knowledge would in general be difficult to communicate if we instead relegated the decision to some non-local or even



(a) Example application call graph.

(b) Hierarchical relationship of libraries.

Figure 1: An example parallel application that consists of libraries implemented using Pthreads, Cilk, TBB, and OpenMP.

global scheduling entity.

Our second design decision is that *libraries should cooperatively share harts*. Callers should not preempt harts already granted to a library. Instead, callees should cooperatively return harts when (a) they are done with them, (b) have diminishing returns in their effective use of them, or (c) can not currently use them because, for example, they are waiting on a synchronization event (see Section 4.3). We believe preemption is only necessary for servicing events like I/O or timers, a task better supported at the operating system level. A cooperative model does not imply that a programmer needs to take a defensive strategy against losing harts—a sequential library might contain a misbehaving infinite loop but programmers do not generally protect against this possibility at every call site.

4 Lithe

Lithe standardizes and facilitates the sharing of harts between parallel libraries. Much like how the application binary interface (ABI) enables interoperability of libraries by defining standard mechanisms for invoking functions and passing arguments, Lithe enables the efficient composition of parallel libraries by defining standard mechanisms for exchanging harts.

Lithe consists of two components – a callback interface and a runtime. The Lithe callback interface defines a standard set of functions that must be implemented by all libraries that want to manage their own set of harts. The Lithe runtime keeps track of all parallel libraries in an application, and invokes the appropriate library's callback on behalf of libraries that want to request or share harts. Table 1 enumerates the callback interface and the corresponding runtime functions, which we next describe in detail. Note that the Lithe runtime functions have the `lithe_` prefix.

4.1 Lithe Callback Interface and Runtime

We define a *scheduler* as an object (i.e. code and state) that implements the Lithe callback interface for a library. At any point in time, each hart is managed by exactly one scheduler, its *current* scheduler. The Lithe runtime keeps track of the current scheduler for every hart.

Whenever a library wants to manage harts for parallel execution, it instantiates and registers its scheduler with Lithe using the runtime function `lithe_register`. This establishes the scheduler as the *child* of the executing hart's current scheduler (the *parent*). The Lithe runtime in turn invokes the parent scheduler's `register` callback, which records the new child scheduler as wanting to manage harts for its associated computation.

To request additional harts, a scheduler invokes `lithe_request`, which invokes its parent scheduler's `request` callback. If the parent does not have enough harts to service the child, it can in turn request harts from its own parent (i.e., the child's grandparent).

To grant a particular hart to a child scheduler, the parent scheduler calls `lithe_enter` with that hart, passing the child as an argument. The Lithe runtime then invokes the child scheduler's `enter` callback, effectively giving the child control of the hart.

When a scheduler is done with a particular hart, it returns the hart to its parent scheduler by calling `lithe_yield` with that hart. The Lithe runtime then

invokes the parent scheduler's `yield` callback, effectively returning control of the hart back to the parent.

When a child scheduler eventually completes its computation and no longer needs to manage harts, it calls `lithe_unregister`, which in turn invokes the parent scheduler's `unregister` callback. The parent then knows not to give any more harts to this child, and can clean up any associated state.

The Lithe runtime provides a base scheduler for the application, which acts as an intermediary between the application and the operating system. It has at most one child scheduler, the root scheduler of the application. When the root scheduler invokes `lithe_request`, the base scheduler requests additional harts from the operating system, and passes all harts allocated by the operating system to the root scheduler using `lithe_enter`.

The `register`, `unregister`, and `request` callbacks return to their callers immediately, whereas the `enter` and `yield` callbacks do not return but instead cause the hart to flow to a new code path within a different library.

4.2 Parallel Quicksort Example

To illlustrate how a library can manage its own parallel execution using Lithe, consider the parallel quicksort example in Figure 2. The top-level function is `sort` (lines 1–16). Since our sort library wants to manage its own harts, it instantiates a `par_sort_sched` scheduler object (line 6). Assume that the scheduler object contains an internal queue and a function dispatch table for the Lithe runtime to find its callbacks. For simplicity, we have only shown one of its callbacks, `par_sort_enter` (line 30).

Figure 3 shows how harts move in and out of the sort scheduler. First, `sort` registers the new scheduler with the Lithe runtime (line 3-6), effectively transferring the ownership of the current hart to `sort`, and notifying the parent scheduler of its intent to parallelize. Once the scheduler is registered, we request as many harts as possible (line 8).

Our quicksort example is based on the standard recursive divide-and-conquer algorithm that partitions a vector of elements into two halves that are sorted in parallel (`par_sort`, called in line 10, defined in lines 18–28). The base case is when the partitions become too small to be split any further, and are simply sorted sequentially (lines 20–21).

To parallelize, we utilize a queue to store partitions that have not been sorted. As shown in lines 25 and 27, we sort the left half of the partition, while enqueuing the right half to be executed by the next available hart. In contrast, a Pthread implementation might create a virtualized thread to sort the right half in parallel, which may result in many virtualized threads competing counterproductively for a limited set of processors.

```
1  void sort(vector *v)
2  {
3    par_sort_sched sched;
4    sched.q.init();
5
6    lithe_register(&sched);
7
8    lithe_request(MAX_NUM_HARTS);
9
10   par_sort(v, &sched.q);
11   vector *next;
12   while (sched.q.dequeue(&next))
13     par_sort(next, &sched.q);
14
15   lithe_unregister();
16 }
17
18 void par_sort(vector *v, queue *q)
19 {
20   if (v->length < 1000)
21     seq_sort(v);
22
23   vector *left, *right;
24   v->partition(&left, &right);
25   q->enqueue(right);
26
27   par_sort(left, q);
28 }
29
30 void par_sort_enter(par_sort_sched *sched)
31 {
32   vector *next;
33   while (sched->q.dequeue(&next))
34     par_sort(next, &sched->q);
35   lithe_yield();
36 }
```

Figure 2: Parallel quicksort pseudocode example.

The main hart will continue to process the enqueued partitions one-by-one until the queue is empty, i.e. all partitions have been sorted (lines 12–13). This means that the sort can be completed on a single hart if necessary. However, if the `par_sort_sched` is granted additional harts by its parent scheduler, each of these new harts will execute `par_sort_enter` and help the main hart finish sorting the remaining partitions (lines 33–34).

When the vector is completely sorted, we yield all of our additional harts back to our parent scheduler (line 35), and unregister (line 15) using the main hart so that the parent scheduler can stop granting more harts.

4.3 Synchronization

Synchronization across libraries within Lithe requires extra care because the system might deadlock if one of the libraries is not allocated a hart. Therefore, harts are forbidden from spinning indefinitely on synchronization objects. Instead, Lithe provides synchronization primitives (mutexes, barriers, and semaphores) which can be used directly or from which other synchronization objects can be created.

The Lithe synchronization primitives are imple-

Table 1: The Lithe runtime and callback interface.

Runtime Function	Callback	Description
lithe_register(scheduler)	register(child)	install this scheduler as the current scheduler
lithe_unregister()	unregister(child)	terminate the current scheduler and reinstall its parent
lithe_request(nharts)	request(child, nharts)	request additional harts for the current scheduler from its parent
lithe_enter(child) lithe_block(context)	enter()	grant this hart from the current scheduler to its chosen child save current context & reenter current scheduler to do something else
lithe_yield()	yield(child)	return this hart from the current scheduler to its parent
lithe_unblock(context)	unblock(context)	signal to this context's scheduler that it is ready to run

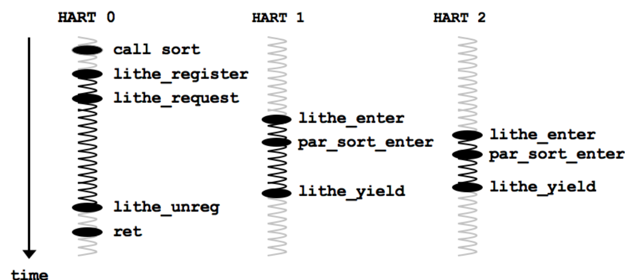


Figure 3: Possible execution of quicksort example.

mented using the `lithe_block` runtime routine. The `lithe_block` routine saves the current execution context, and re-enters the current scheduler by invoking its `enter` callback. When the synchronization primitive is ready to resume an execution context (for example, because the mutex was released), it invokes the `lithe_unblock` runtime routine with the resumable execution context as an argument. This in turn invokes the `unblock` callback on the scheduler that owned the execution context when it was blocked. Note that the hart invoking the `unblock` callback may be from a different scheduler and can not be used to resume the execution context, but that hart may be used to request additional harts from within the `unblock` callback (for example, if the scheduler had already yielded all of its own harts). Like `register`, `unregister`, and `request`, the `unblock` callback does not indicate a transfer of hart management, but should just return to the caller.

Note that synchronization within a scheduler often does not need to use these primitives. For example, a Lithe-compliant TBB scheduler implementation would use the Lithe synchronization primitives to implement external synchronization objects (i.e. their reentrant queuing mutex), but internally use spinlocks to protect scheduler data structures.

4.4 Discussion

Lithe was designed to reinforce a clear separation of interface and implementation so that a programmer can call a library routine without knowing whether the routine is sequential or parallel. This enables incremental parallelization of sequential libraries.

Lithe was also designed such that only libraries desiring parallel execution need to interact with Lithe, providing backwards compatibility with existing sequential libraries without adding any execution overhead.

Most application programmers will not need to know the details of Lithe. Rather, they can leverage parallel libraries, frameworks, or languages that have already been modified to be Lithe-compliant.

5 Related Work

Several other systems support multiple co-existing schedulers, but with varying motivations. The most similar to Lithe is the Converse system [8], which also strives to enable interoperability between different parallel runtimes. Converse also built synchronization mechanisms on top of the scheduling interface. However, the two systems are fundamentally different in how schedulers cooperate. In Lithe, a parent scheduler grants harts to a child scheduler to do with as it pleases. In Converse, the child scheduler must register individual *threads* with a parent scheduler, effectively breaking modularity and forcing the parent to manage individual threads on its behalf; the child only gets harts implicitly when its parent decides to invoke these threads. Furthermore, Converse does not facilitate the plug-and-play modularity supported by Lithe, since a scheduler in Converse must know its parent's specific interface in order to register its threads, effectively limiting with which codes it can interoperate. In addition, the hierarchical relationship of the schedulers does not follow the call graph, making it harder to use local knowledge to prioritize between schedulers.

The CPU inheritance scheme [6] allows both operating system and application modules to customize their scheduling policies, although the focus is more on OS schedulers. As in Converse, CPU inheritance schedulers cooperate by scheduling another scheduler's threads rather than by exchanging resources. Moreover, all blocking, unblocking, and CPU donations must go through a "dispatcher,“ which is in the kernel, forcing inter-scheduler cooperation to be heavyweight even if all of the schedulers are within the same application.

The Manticore [5] framework consists of a compiler and runtime that exposes a small collection of scheduling primitives upon which complex interoperable scheduling policies can be implemented. However, unlike unvirtualized harts, the *vproc* abstraction is designed to be time-multiplexed by default, thus cannot prevent scheduler oversubscription of hardware resources. Furthermore, schedulers do not interact with each other to share resources but rather all request additional vprocs from a global entity.

The Concert framework [9] is designed to achieve better load balancing of large irregular parallel applications. It defines the same threading model across all schedulers, but allows them to customize their load-balancing policies. Concert's root FIFO scheduler time-multiplexes its child schedulers, causing the same potential oversubscription problem as with virtualized kernel threads.

6 Conclusion

There cannot be a thriving parallel software industry unless programmers can compose arbitrary parallel libraries without sacrificing performance. We believe that the interoperability of parallel codes is best achieved by exposing unvirtualized hardware resources and sharing these cooperatively across parallel libraries within an application. Our solution, Lithe, defines a standardized resource sharing interface which is added unobtrusively to the side of the standard function call interface, preserving existing software investments while enabling incremental parallelization.

References

[1] *Intel Math Kernel Library for the Linux Operating System: User's Guide*. October 2007.

[2] T. E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *SOSP*, Pacific Grove, CA, 1991.

[3] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *PPOPP*, Santa Barbara, CA, July 1995.

[4] R. Chandra et al. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[5] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *International Conference on Functional Programming*, Victoria, British Columbia, Canada, September 2008.

[6] B. Ford and S. Susarla. CPU inheritance scheduling. In *Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[7] P. Husbands and K. Yelick. Multithreading and one-sided communication in parallel lu factorization. In *Supercomputing*, Reno, NV, November 2007.

[8] L. V. Kale, J. Yelon, and T. Knauff. Threads for interoperable parallel programming. *Languages and Compilers for Parallel Computing*, April 1996.

[9] V. Karamcheti and A. A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Supercomputing*, Orlando, FL, November 2002.

[10] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. Technical report, LAPACK, 2006.

[11] R. Liu et al. Tessellation: Space-time partitioning in a manycore client OS. In *HotPar*, Berkeley, CA, 2009.

[12] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core PRocessor Parallelism*. O'Reilly, 2007.

[13] D. Tullsen et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, Philadelphia, PA, 1996.

Acknowledgements

We would like to thank George Necula and the rest of Berkeley Par Lab for their feedback on this work.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). This work has also been in part supported by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors also acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.