Composing Parallel Software Efficiently with Lithe

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

Benjamin Hindman
UC Berkeley
benh@eecs.berkeley.edu

Krste Asanović
UC Berkeley
krste@eecs.berkeley.edu

Abstract

Applications composed of multiple parallel libraries perform poorly when those libraries interfere with one another by obliviously using the same physical cores, leading to destructive resource oversubscription. This paper presents the design and implementation of *Lithe*, a low-level substrate that provides the basic primitives and a standard interface for composing parallel codes efficiently. Lithe can be inserted underneath the runtimes of legacy parallel libraries to provide *bolt-on* composability without needing to change existing application code. Lithe can also serve as the foundation for building new parallel abstractions and libraries that automatically interoperate with one another.

In this paper, we show versions of Threading Building Blocks (TBB) and OpenMP perform competitively with their original implementations when ported to Lithe. Furthermore, for two applications composed of multiple parallel libraries, we show that leveraging our substrate outperforms their original, even expertly tuned, implementations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Run-time Environments; D.4.1 [*Operating Systems*]: Process Management; D.2.12 [*Software Engineering*]: Interoperability

General Terms Algorithms, Design, Languages, Performance

Keywords Composability, Cooperative Scheduling, Hierarchical Scheduling, Oversubscription, Parallelism, Resource Management, User-Level Scheduling

1. Introduction

With the widespread adoption of multicore microprocessors, many software libraries are now becoming parallelized. These libraries exploit decades of parallel programming innovation in novel abstractions, performance optimizations, and correctness. For productivity, programmers would like to *reuse* these parallel libraries by *composing* them together to build applications, as is standard practice with sequential libraries. Unfortunately, composing parallel libraries *efficiently* is hard. The parallel libraries may interfere with one another, sometimes delivering performance that is much worse than a sequential implementation.

The root of this problem is poor resource management across parallel libraries. Each parallel library creates its own set of operating system threads to represent the processor cores in the machine. While threads provide a natural *programming abstraction* for some types of computation (e.g. [40]), they are a poor *resource abstraction* for parallel programming, since they are multiplexed onto the same physical cores (Figure 1(a)). When too many threads are active, each thread runs at infrequent and unpredictable times, interfering with synchronization mechanisms [31] and custom scheduling policies [23]. Unpredictable multiplexing also leads to cache interference between threads, hindering optimizations such as prefetching [18] and cache-blocking [43].

Current software libraries supply ad hoc solutions to this problem. A glaring example is Intel's Math Kernel Library (MKL), which instructs its clients to call the *sequential* version of the library whenever it might be running in parallel with another part of the application [19]. Such solutions destroy any separation between interface and implementation, and place a difficult burden on programmers who have to manually choose between different library implementations depending on the calling environment. Worse still, a programmer writing a new parallel library that encapsulates MKL will just deflect the problem to its clients. A thriving parallel software industry will only be possible if programmers can arbitrarily compose parallel libraries without sacrificing performance.

One possible solution is to require all parallelism to be expressed using a universal high-level abstraction. While attractive in principle, this goal has proven elusive in practice. First, there has been no agreement on the best parallel practices, as evidenced by the proliferation of new parallel languages and abstractions over the years. Second, it is unlikely that a competitive universal abstraction even exists, as they have been repeatedly outperformed by optimizations that leverage domain or application-specific knowledge (e.g. [6, 37]).

Our solution, *Lithe*, is a low-level substrate that provides the basic primitives for parallel execution and a standard interface for composing arbitrary parallel libraries efficiently. Lithe replaces the virtualized thread abstraction with an unvirtualized hardware thread primitive, or *hart*, to represent a processing resource (Figure 1(b)). Whereas threads provide the false illusion of unlimited processing resources, harts must be explicitly allocated and shared amongst the different libraries. Libraries within an application are given complete control over how to manage the harts they have been allocated, including how to allocate harts to parallel libraries they invoke.

The runtimes of existing languages and abstractions can be easily ported to Lithe. This allows, for example, an application written in a high-level language like Haskell to efficiently interoperate with low-level libraries that use OpenMP. In addition, Lithe serves as a basis for building a wide range of new parallel abstractions.

We have ported Intel's Threading Building Blocks (TBB) [34] and GNU's OpenMP [5] libraries to run with Lithe. We have also implemented a new parallel library, *libprocess* [17], which exports an Erlang-like actor-based programming model. As a baseline, we show that the standalone execution of the ported libraries performs

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06...$10.00

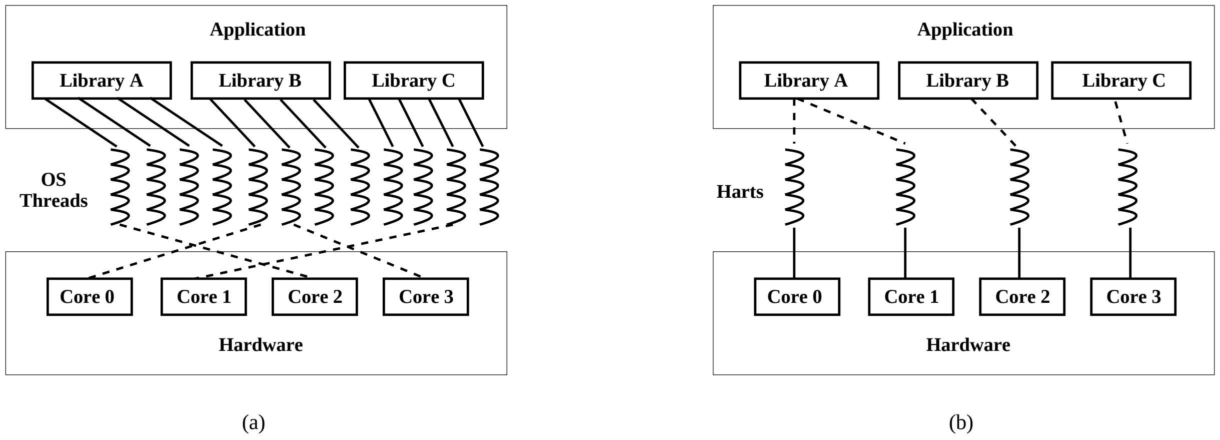


Figure 1. In conventional systems (a), each library is only aware of its own set of virtualized threads, which the operating system multiplexes onto available physical processors. Lithe, (b), provides unvirtualized processing resources, or harts, which are shared cooperatively by the application libraries, preventing resource oversubscription.

competitively with their original implementations. We also show significant performance improvements when composing these libraries in two application case studies. Our first case study, an application server modeled after Flickr.com, uses the popular GraphicsMagick [16] library to resize images uploaded by clients, and achieves roughly 2× latency improvement for the same throughput when leveraging Lithe. Our second case study, a third-party sparse QR application [7], achieves up to a 1.5× speedup over the original naive implementation when leveraging Lithe, and even outperforms the original implementation with expert manual tuning of resource allocation. In addition, we show how parallel composition can cause synchronization and scheduling to interact poorly, and use a simple barrier case study to illustrate how Lithe can improve the situation.

2. Cooperative Hierarchical Resource Management

Applications are built by composing libraries *hierarchically*. For example, an image processing application may use the popular GraphicsMagick library [16], which in turn uses the network graphics library `libpng` (amongst others), which in turn uses the compression library `zlib`. Programmers tend to think of this hierarchy in terms of control: a caller (`libpng`) invokes a callee (`zlib`).

In sequential computing, this *explicit* transfer of control is accompanied with an *implicit* allocation of processing resources (e.g., processor, cache) for the callee to use. In fact, the caller only continues after the callee has returned, i.e., yielded back its processing resources to the caller. Callers want to *cooperatively* provide their callees with these resources because their callees are doing something on their behalf. Likewise, callees cooperatively return their processing resources when they are done performing their computation.

In parallel computing, a single processing resource is still implicitly allocated when a caller invokes a callee, but callees must obtain subsequent processing resources via other means, often by creating additional operating system threads. Callers, however, would often like to manage the allocation of resources to each of their callees. This enables a caller with local knowledge (e.g., about a critical path) to optimize their execution.

In this work, we enable the management of resources to be *coupled* with the hierarchical transfer of control between libraries. Furthermore, just as with control, processing resources are allocated and yielded cooperatively: libraries allocate resources to libraries they call, and libraries return resources allocated to them when they are finished computing.

This is essentially the confluence of two lines of research: hierarchical scheduling (e.g. [13, 41]) and cooperative multitasking (e.g. [1, 30]). The hierarchical scheduling schemes use hierarchies for modular resource management, but often between untrusted entities (typically processes). The cooperative multitasking schemes allow each task to execute without being interrupted, but tasks often cannot control who runs next. Our cooperative hierarchical scheme combines the two, enabling local control of resource allocation and allowing a library to make effective use of the resources allocated to it without worrying about possible interference.

3. Lithe Primitives

Lithe provides two basic primitives, harts and contexts, to enable library runtimes to perform parallel execution. The *hart* primitive prevents *uncontrolled oversubscription* of the machine, while the *context* primitive prevents *undersubscription* of the machine.

A majority of programmers will be blissfully unaware of Lithe. Only low-level runtime programmers, such as the implementors of TBB and OpenMP, will use Lithe primitives directly. Many programmers, such as the implementors of GraphicsMagick and MKL, will use high-level parallel constructs and abstractions whose runtimes will be implemented using the Lithe primtives. The remainder of programmers will simply invoke library routines directly, such as those in GraphicsMagick, without knowing whether they are even implemented or executed sequentially or in parallel.

In the remainder of this section, we describe each of our primitives in more detail. In the following section, we describe how parallel runtimes use these primitives and the rest of our substrate to share resources in a cooperative and hierarchical manner.

3.1 Harts

A **hart**, short for **har**dware **t**hread, represents a processing resource. There is typically one hardware thread per physical core, except for multithreaded machines like SMTs [39].

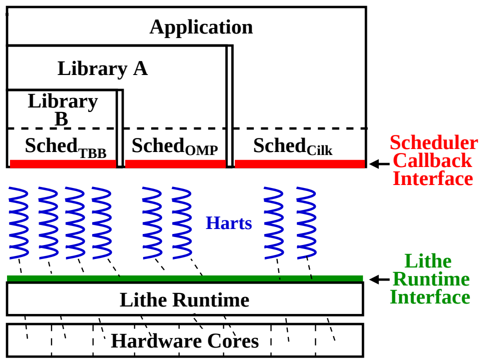


Figure 2. Software stack showing the callback and runtime interfaces.

The hart primitive prevents processing resource oversubscription in two ways. First, within an application, there is a *fixed one-to-one mapping* between harts and the physical hardware thread contexts. Thus, two libraries running on two different harts will not interfere with each other. Second, a hart must be allocated to a runtime before that runtime can execute code. This is in contrast to threads, which may be created by a runtime on-demand to execute code.

3.2 Contexts

Each hart always has an associated **context**, which acts as the execution vessel for the computation running on the hart. The context primitive allows one to suspend the current computation by *blocking* its context. A blocked context stores the *continuation* of the computation, including the stack that the computation is using, and any "context-local state." Note that these are not first-class continuations, but can only be used once.

Contexts allow runtimes to *interoperate* with libraries that may need to block the current computation. For example, a library that provides a mutual exclusion lock may want to block some computation until a lock is released. Similarly, some computation calling into a network I/O library may need to be blocked until a packet arrives. A library can avoid undersubscribing the machine by blocking the current context while having the underlying hart continue "beating" and running other contexts.

4. Lithe Interface

Much like how an application binary interface (ABI) enables interoperability of codes by defining standard mechanisms for invoking functions and passing arguments, Lithe enables the efficient composition of parallel codes by defining standard mechanisms for exchanging harts. As shown in Figure 2, the Lithe substrate defines two standard interfaces:

1. A *runtime interface* that parallel runtimes use to share harts and manipulate contexts.
2. A *scheduler callback interface* that each parallel runtime must implement to manage its own harts and interoperate with others.

For the remainder of this paper, we refer to the portion of each parallel runtime that implements the callback interface as a *scheduler*. At any point in time, a hart is managed by exactly one scheduler, and each scheduler knows exactly which harts are under its control. We call the scheduler that is currently managing a hart the *current scheduler*. The Lithe runtime keeps track of the current scheduler for each hart, as well as the hierarchy of schedulers.

Table 1. The Lithe runtime functions and their corresponding scheduler callbacks.

Lithe Runtime Interface	Scheduler Callback Interface
`sched_register(sched)`	`register(child)`
`sched_unregister()`	`unregister(child)`
`sched_request(nharts)`	`request(child, nharts)`
`sched_enter(child)`	`enter()`
`sched_yield()`	`yield(child)`
`sched_reenter()`	`enter()`
`ctx_init(ctx, stack)`	N/A
`ctx_fini(ctx)`	N/A
`ctx_run(ctx, fn)`	N/A
`ctx_pause(fn)`	N/A
`ctx_resume(ctx)`	N/A
`ctx_block(ctx)`	`block(ctx)`
`ctx_unblock(ctx)`	`unblock(ctx)`

Table 1 lists all of the runtime functions and their corresponding scheduler callbacks. We discuss each of them in detail below.

4.1 Sharing Harts

4.1.1 Runtime Functions and Callbacks

Register A new scheduler registers itself with the Lithe runtime using the `sched_register` function. This function performs four actions: (1) it records the scheduler as the *child* of the current scheduler, (2) invokes the current scheduler's (i.e., *parent's*) `register` callback, (3) updates the current scheduler of the hart to be the new scheduler, and (4) inserts the new scheduler into the scheduler hierarchy.

Invoking the `register` callback informs the parent that the new child scheduler is taking over a hart. With this callback, a parent scheduler can update any necessary state before returning the hart to the child.

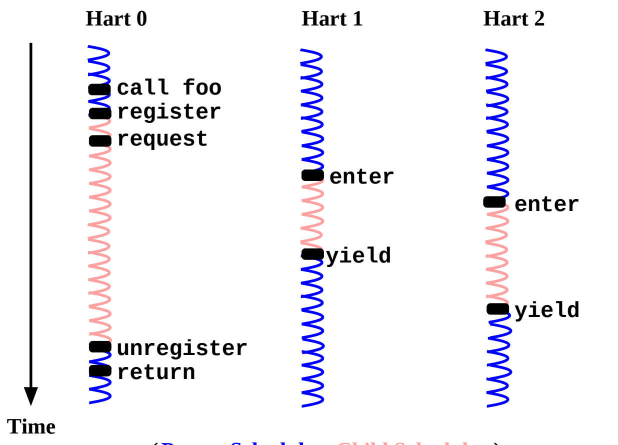
Unregister Once a child scheduler completes its computation and no longer needs to manage harts, it calls `sched_unregister`. This function performs three actions: (1) it invokes the parent scheduler's `unregister` callback, (2) reverts the current scheduler of the hart to be the parent scheduler, and (3) removes the unregistering scheduler from the hierarchy. The `sched_unregister` function does not return until all of the child's harts have returned back to the parent.

Invoking the `unregister` callback informs the parent that this child scheduler will no longer want more harts. It also gives the parent a chance to clean up any state it may have associated with the child.

Request To request additional harts, a scheduler invokes `sched_request`, passing the number of harts desired. This function simply invokes the parent scheduler's `request` callback, passing both the child that has made the request and the number of harts requested. If the parent chooses to, it can itself invoke `sched_request`, and propagate the resource request up the scheduler hierarchy.

Enter/Reenter A parent scheduler can allocate a hart to a child scheduler by invoking the `sched_enter` function and passing the child as an argument. This function performs two actions: (1) it updates the current scheduler to be the child, and (2) invokes the child scheduler's `enter` callback. With this callback, the child scheduler can use the hart to execute some computation, or grant the hart to one of its children by invoking `sched_enter`. Once a hart completes a computation it can return to the current scheduler to obtain the next task by invoking `sched_reenter`, which will invoke the current scheduler's `enter` callback.

Yield Whenever a scheduler is done with a hart, it can invoke `sched_yield`. The `sched_yield` function performs the fol-



(Parent Scheduler, Child Scheduler)

Figure 3. An example of how multiple harts might flow between a parent and child scheduler.

lowing actions: (1) it updates the current scheduler from the child to the parent, and (2) invokes the parent scheduler's `yield` callback, specifying the child that yielded this hart. With this callback, the parent can use the hart to execute some other computation, grant the hart to another child using `sched_enter`, or return the hart back to its parent using `sched_yield`.

4.1.2 Example

Figure 3 shows an example of how a parent and child scheduler use Lithe to cooperatively share harts. A parent library calls a child library's `foo` function to do some computation. The implementation of `foo` then instantiates a scheduler to manage the parallelism within that function. The scheduler uses the initial hart to register itself with its parent scheduler, who is managing the parallelism for the caller library, then requests additional harts from that parent scheduler. It then begins to execute the computation of `foo` using that hart.

After the parent scheduler has received the request, it may decide to grant additional harts to the new scheduler by invoking `sched_enter` on each of these harts. Each `sched_enter`, in turn, will invoke the new scheduler's `enter` callback, which uses its extra hart to help with executing the computation. When the scheduler is done with the computation, it yields all of the additionally granted harts explicitly by invoking `sched_yield`. It will also unregister itself with its parent scheduler, then return from `foo`.

4.1.3 Transitioning Between Schedulers

The Lithe runtime provides a special *transition context* for every hart, each including a small preallocated stack. When a hart transitions between schedulers using the `sched_enter` and `sched_yield` routines, it uses its transition context to execute the `enter` and `yield` callbacks, respectively. The transition context acts as a temporary context for a scheduler to run on before it starts or resumes one of its own contexts. Using the transition context frees a scheduler from having to coordinate with other schedulers to share contexts and their associated stacks.

4.1.4 Base Scheduler

The Lithe runtime provides a *base scheduler* for the application. The base scheduler obtains harts for the application to use from the operating system environment and serves as the parent of the first scheduler to register with Lithe, i.e., the *root scheduler*. Upon receiving requests from the root scheduler, the base scheduler simply passes available harts directly to the root scheduler via `sched_enter`.

4.2 Contexts

A scheduler can allocate and manage its own contexts, giving it full control over how to manage its stacks, enabling optimizations such as linked stacks [40], stacklets [14], or simply frames [36]. The scheduler can also allocate and manage any context-local state.

A scheduler sets up a context by calling the `ctx_init` function and passing a stack for the context to use. A scheduler starts the context by invoking `ctx_run`, and passing it a function that should be invoked by the context. Note that after the computation using the context has completed, it can be reused by subsequent calls to `ctx_run`. A scheduler cleans up a context by calling the runtime function `ctx_fini`.

Context Switching The current context can be paused using the `ctx_pause` function. The `ctx_pause` function works similarly to the `call/cc` control operator. When `ctx_pause` is invoked, it stores the current continuation in the current context, *switches* to the transition context, and calls the function passed to `ctx_pause` with the current context as an argument. Switching to a *different* context before invoking the function passed to `ctx_pause` allows a programmer to manipulate a context without worrying about running on said context's stack. Fisher and Reppy recognized this dilemma when they realized they needed to enqueue a continuation *and* atomically get off its stack before the continuation was used by another processor [10]. Using `ctx_pause` deals with this problem, and others like it, in an elegant and easy to reason about manner.

After pausing the context, a library can perform the steps necessary to save or discard the context before switching to something else. For example, an I/O or synchronization library can alert the current scheduler of the blocked context by using the `ctx_block` function, which calls the context's scheduler's `block` callback. After `ctx_block` returns, the library can relinquish the hart back to the current scheduler using `sched_reenter`, which invokes the current scheduler's `enter` callback.

To signify that a blocked context is now runnable, an I/O or synchronization library can invoke the `ctx_unblock` function, passing the runnable context as an argument. This function invokes the specified context's scheduler's `unblock` callback. The scheduler's `unblock` function *should not* resume the unblocked context using the hart used to execute the callback, but should store the context to be run later.

4.3 Putting it All Together: SPMD Scheduler Example

To help explain how to manage harts and contexts using the Lithe runtime, we describe a simple parallel library that supports the Single-Program Multiple-Data (SPMD) programming model.

The library provides two functions to its users: `spmd_spawn` and `spmd_tid`. `spmd_spawn` spawns N tasks, each invoking the same given function with the same given argument. Each task can then figure out what to do using its task id, obtained through `spmd_tid`. `spmd_spawn` returns after all its spawned tasks have completed.

The crux of the SPMD library is shown as pseudocode in Figure 4. First, `spmd_spawn` instantiates a SPMD scheduler, and registers that scheduler (including its callbacks) with the Lithe runtime using `sched_register` (lines 2-3). The scheduler requests additional harts using `sched_request` (line 4). Using the initial hart, the scheduler executes its computation (line 5), essentially running one SPMD task after another until they are all completed (lines 10-13). It then unregisters itself with its parent (line 6). The

```
 1 void spmd_spawn(int N, void (*func)(void*), void *arg) {
 2   SpmdSched *sched = new SpmdSched(N, func, arg);
 3   sched_register(sched);
 4   sched_request(N-1);
 5   sched->compute();
 6   sched_unregister();
 7   delete sched;
 8 }
 9
10 void SpmdSched::compute() {
11   while (/* unstarted tasks */)
12     func(arg);
13 }
14
15 void SpmdSched::enter() {
16   if (/* unblocked paused contexts */)
17     ctx_resume(/* next unblocked context */);
18   else if (/* requests from children */)
19     sched_enter(/* next child scheduler */);
20   else if (/* unstarted tasks */)
21     ctx = new SpmdCtx();
22     ctx_run(ctx, start);
23   else sched_yield();
24 }
25
26 void SpmdSched::start() {
27   compute();
28   ctx_pause(cleanup);
29 }
30
31 void SpmdSched::cleanup(ctx) {
32   delete ctx;
33   if (/* not all tasks completed */)
34     sched_reenter();
35   else
36     sched_yield();
37 }
```

Figure 4. SMPD scheduler pseudocode example.

spmd_spawn function cleans up the scheduler before returning back to its caller (line 7).

Any additional harts granted by its parent will join the SPMD scheduler through its enter callback (line 15). First, the scheduler tries to resume any paused context that is now runnable (lines 16-17). Otherwise, the scheduler tries to satisfy requests from its children (lines 18-19). Finally, it will try to create a new context to start a new SPMD task, by invoking the runtime function ctx_start with the new context and the start function (lines 20-22). start will continually run the next SPMD task using that context until they are all completed (line 27). It will then clean up that context (line 28), and either send the hart back into the main scheduling loop to do more work (lines 33-34), or yield the hart back to its parent (lines 35-36).

5. Implementation

We have implemented a complete Lithe runtime, as well as a user-level implementation of harts, for 64-bit Linux. Both require only modest amounts of code. The implementation of Lithe is approximately 2,300 lines of C, C++, and x86 assembly, while the harts implementation is approximately 600 lines of C and x86 assembly.

5.1 Lithe

The Lithe runtime defines two opaque types: schedulers and contexts. A Lithe scheduler object contains the location of the function pointer table of standard callbacks, and the location of the scheduling state, both of which are supplied by the corresponding user-level scheduler upon registering with the runtime. A Lithe scheduler object also contains a pointer to its parent, and a pointer to a doubly linked list of its children. Internally, the runtime keeps track of the scheduler hierarchy, as well as the current scheduler object for each hart. Externally, the runtime gives the pointer to each scheduler object to its parent scheduler as an opaque handle for that child. The core runtime interface is in C, but we provide a C++ scheduler abstract base class that delineates the required standard callbacks.

A Lithe context object contains the hart's machine state, a pointer to its corresponding Lithe scheduler object, and the location of the context-local state as specified by its scheduler. The runtime uses the POSIX ucontext to represent the hart's machine state, and uses the GNU ucontext library to save and resume contexts. We have made minor changes to GNU's ucontext library to eliminate unnecessary system calls to change the signal mask, so that saving and resuming contexts do not incur any kernel crossings.

5.2 Harts

Our user-level implementation represents each hart using a pthread, since Linux maps each pthread to its own kernel thread, and uses the affinity extension of pthreads supported by Linux to pin each pthread to a unique core.

We create and initialize a global pool of N harts at the beginning of the application, with N being the number of cores in the machine. One hart is used to call the application's main function. The remaining harts sleep at a gate, waiting for work. When Lithe's base scheduler requests additional resources using hart_request, the harts are released from the gate to call into the application's entry function. As harts return from the entry function or via an explicit hart_yield, they go back to waiting at the gate.

5.3 Operating System Support

We choose to implement both Lithe and harts as completely user-level libraries rather than modifying an existing operating system. The main benefit of our approach is portability across existing platforms that provide the necessary POSIX constructs. The main downfall is that an operating system is still free to multiplex our harts with other applications running in the system. As the results in this paper show (Section 7), however, the benefits of reducing the number of operating system threads and carefully managing their use provides better behavior even if the operating system is unaware of our user-level schedulers.

Using a two-level scheduling mechanism such as scheduler activations could provide user-level Lithe schedulers with even more information about their running computations, such as when contexts block due to page faults, or when a hart has been preempted. In addition, the operating system could notify schedulers when their computations block on I/O. In lieu of such support, we can use the existing event-based interfaces in many operating system's to avoid blocking a hart on I/O (similar to previous work such as Capriccio [40]). In fact, in Section 6.3 we present the libprocess library which provides a blocking I/O interface that pauses the underlying context and uses event-based mechanisms from operating systems to perform I/O. This allows the hart to execute other computations/contexts while the previous computation/context is waiting on I/O.

6. Interoperable Parallel Libraries

We have built a wide range of interoperable scheduling, I/O, and synchronization libraries that use Lithe. We have ported some existing popular libraries, as well as implemented our own. In this section, we describe the original baseline for each of these libraries, then show how to modify them to use Lithe (and thus, work with each other).

Table 2. Approximate lines of code required to port TBB and OpenMP.

	Added	Removed	Modified	Relevant	Total
TBB	180	5	70	1,500	8,000
OpenMP	220	35	150	1,000	6,000

6.1 Threading Building Blocks

Intel's Threading Building Blocks (TBB) library [34] provides a high-level task abstraction to help programmers achieve performance and scalability without having to manually manage tasks' parallel execution. Instead, a TBB scheduler uses a dynamic work-stealing [4] approach to automatically map tasks onto cores.

The original open-source Linux implementation of the TBB library uses pthreads as its resource abstraction. The TBB library creates a fixed-size global pool of threads upon its initialization, and associates a worker with each of these threads. Each TBB scheduler uses N workers from the global pool to execute its tasks, with N either being the number of cores in the machine or the number specified by the user. Each of these workers repeatedly completes its own share of work, then tries to steal more work from other workers. The original TBB library tries to prevent resource oversubscription by sharing the global fixed-size pool of threads between all the TBB schedulers within an application. Of course, this still does not preclude TBB schedulers from interfering with other non-TBB parallel libraries.

In the ported implementation of TBB, each TBB scheduler manages harts rather than threads. Since a TBB scheduler does not know how many harts it can use upfront, it instead lazily creates its workers for each hart that it obtains through its `enter` callback. The core algorithm for each worker remains the same, since each worker can just start stealing work from existing workers as soon as it is instantiated. The ported implementation also keeps track of its child schedulers and the corresponding number of harts they requested. In this initial implementation, harts are granted to children in a round-robin fashion *only* after there are no tasks left to steal and/or execute.

Porting TBB was a relatively straightforward task. It took one of the authors no more than a week. The first row of Table 2 shows the approximate number of lines of code that were added, removed, and modified. The column labeled "Relevant" refers to the number of lines of the TBB runtime that actually do scheduling (as opposed to scalable memory allocation, for example) in contrast to the total number of lines (last column).

6.2 OpenMP

OpenMP [5] is a portable parallel programming interface containing compiler directives for C, C++, and Fortran. Programmers use the directives to specify parallel regions in their source code, which are then compiled into calls into the OpenMP runtime library. Each parallel region is executed by a team of workers, each with a unique id. The number of workers in a team is specified by the user, but defaults to the number of cores in the machine. Although there are higher level directives that allow the compiler to automatically split work within a parallel region across workers, the programmer can also manually assign work to each of worker by writing SPMD-style code.

We ported the GNU Compiler Collection's (GCC) open-source Linux implementation of the OpenMP runtime library (libgomp). As with the original TBB library implementation, the original OpenMP library also uses threads as its resource abstraction. An implicit scheduler for every team maps each of its workers onto a different thread. For efficiency, the library keeps a global pool of the available threads to reuse across multiple teams. Note that when multiple teams are active, a team scheduler will create additional threads rather than waiting for threads from other teams to return to the global pool.

In the ported version of OpenMP, each team scheduler multiplexes its workers onto its available harts. An OpenMP scheduler lets each worker run on a hart until it either completes or its context blocks on a synchronization operation, before running the next worker thread on that hart. In this initial implementation, a scheduler cannot take advantage of additional harts in the middle of executing a parallel region, since it never redistributes its workers once it assigns them to their respective harts. However, the implementation is optimized to keep a global pool of contexts to reuse across multiple teams for efficiency.

Porting OpenMP was more difficult than TBB because it required carefully multiplexing the worker threads, since they are exposed to the application programmer. It took one of the authors approximately one week to port OpenMP, and another week to profile and optimize the implementation to preserve cache affinity for each worker thread. The second row of Table 2 shows the approximate number of lines of code that were added, removed, and modified in the OpenMP library.

6.3 Libprocess

Libprocess [17] is a library written in C/C++ that provides an actor-style message-passing programming model. Libprocess is very similar to Erlang's process model, including basic constructs for sending and receiving messages. Libprocess scales to hundreds of thousands of concurrent processes which are scheduled using pluggable scheduling policies. In fact, scheduling policies can be implemented directly by a client of the library, to precisely control a process's ordering, priority, etc.

In addition to the actor model, libprocess provides a collection of blocking I/O routines for using sockets (`accept`, `connect`, `recv`, `send`). Similar to Capriccio, libprocess exploits non-blocking operating system interfaces. This allows libprocess to run a large number of processes that are all performing socket I/O simultaneously. By using Lithe, libprocess is able to simultaneously run multiple processes across multiple harts. This is one of libprocesses important improvements over Capriccio.

The libprocess implementation is roughly 2,000 lines of code.

6.4 Barriers

In order to experiment with how synchronization primitives interact with parallel libraries, we implemented some simple user-level barriers with and without the Lithe substrate. Both implementations share the bulk of the code. A barrier is initialized with the number of threads that are synchronizing with each other. Upon reaching the barrier, each task increments a counter atomically, then checks to see if it was the last to arrive. If so, it signals all other tasks to proceed. Without Lithe, all other tasks simply spin on their respective signals until they can proceed. With Lithe, all other tasks spin for a very short amount of time, then pause their contexts and yield control of their hart back to the current scheduler using `ctx_block`; the last task to reach the barrier will alert the scheduler to unblock any blocked contexts using `ctx_unblock`. The scheduler can then decide when to resume each of these resumable contexts.

7. Evaluation

In this section, we present results from experiments on commodity hardware that show the effectiveness of the Lithe substrate. First, we present numbers for the individual libraries that show that we can implement existing abstractions with no measurable overhead. Then, we present two real-world case studies to illustrate how we improve the current state-of-the-art in parallel library composition. Lastly, we show how Lithe can improve the performance of barrier synchronization in the face of parallel composition.

	tree sum	preorder	fibonacci
Original TBB	1.44	1.61	1.65
Ported TBB	1.23	1.40	1.45

Table 3. TBB benchmark runs (in seconds).

	cg	ft	is
Original OMP	0.22	0.19	0.54
Ported OMP	0.21	0.13	0.53

Table 4. OpenMP benchmark runs (in seconds).

The hardware platform used was a quad-socket 2.3 GHz AMD Opteron with 4 cores per socket (16 cores total). The OS platform used was the 2.6.18 64-bit Linux kernel (which includes the NPTL fast pthread library). We also used Glibc 2.3.6, TBB version 2.1 update 2, and GNU OpenMP from the GCC 4.4 trunk.

7.1 Ported Libraries

To validate that porting libraries to Lithe incurs negligible or no overhead, we ran a series of benchmarks for both OpenMP and TBB to compare the performance between the original and ported implementations.

The TBB distribution includes a series of examples designed to highlight the performance and scalability of TBB. We randomly chose three of these examples: a parallel tree summing algorithm, a parallel preorder tree traversal, and a parallel implementation of fibonacci. Table 3 shows the average runtime of ten runs of these benchmarks for both the ported and original version.

Across all three examples, the ported TBB performs slightly better than the original. We believe that because harts are created eagerly during the application initialization (similar to a thread pool), we are able to overlap thread creation with computation that occurs before TBB is actually invoked.

To measure the overheads of the OpenMP port, we chose to run three of the NAS parallel benchmarks [21]: conjugate gradient, fast Fourier transform, and integer sort (from NPB version 3.3; problem size W). Table 4 shows the average runtimes for ten runs of these benchmarks for both the ported and original versions of OpenMP. The ported OpenMP is also competitive with, if not better than, the original implementation.

7.2 Case Study 1: Application Server

In our first case study, we investigated the efficacy of Lithe for server-like applications that tightly integrate I/O with parallel computation. Most servers use thread-per-connection parallelism because threads provide *both* (a) a natural abstraction of the control-flow required to process a request and prepare a response (especially with respect to blocking while reading and writing to/from the network) and (b) a natural form of parallelism for execution on a multicore machine.

Newer “application servers” serve dynamic content and may potentially exploit *parallel* computations within a single request in addition to thread-per-connection parallelism. Examples include video encoding [44], speech recognition [15], video production [3], and online gaming (rendering, physics, and AI) [9].

We chose to study an image processing application server, modeled after Flickr’s [11] user image upload server. The software architecture of our application server is straightforward: for each image the server receives, it performs five resize operations, creating large, medium, small, thumbnail, and square versions of the image. Like Flickr, we use the popular GraphicsMagick [16] library, which is parallelized using OpenMP, to resize images.

To collect our performance numbers, we ran the application server using the quad-socket Opteron, and ran the client on a separate machine connected by an Ethernet link. We measured the throughput versus service latency of each configuration as we increased the rate at which the client made requests.

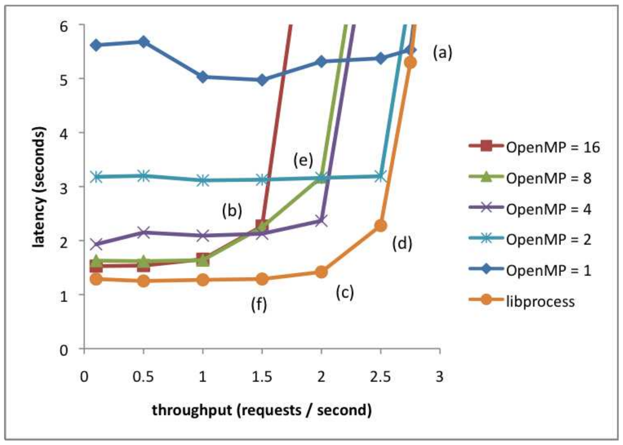


Figure 5. Throughput versus latency for different configurations of image resizing application server.

Because GraphicsMagick is parallelized using OpenMP, we can explore a spectrum of possible configurations. We manually tuned GraphicsMagick by configuring OpenMP to use N threads for each invocation (labeled “OpenMP = *N*”), and show results in Figure 5. The most straightforward approach to implementing our application server is to create a thread-per-connection that performs each of the resizes sequentially (“OpenMP = 1”). Although this version performs well at high load (point (a)), it has high latency when lightly loaded and many cores sit idle causing the server to be underutilized. Manually increasing the number of threads ($N > 1$) allocated to each request helps reduce lightly loaded latency, but also reduces the saturation throughput (e.g. point (e) to point (b)). In general, the more threads used to reduce lightly loaded latency, the greater the loss of saturation throughput.

Next, we implemented the application server using the libprocess library rather than threads. Only the code that sets up either an actor or a thread per connection was different between the two versions. Most source code was unchanged, including the code that read and wrote from sockets and files, and the code that called into GraphicsMagick. While we used the same GraphicsMagick library in both versions, we linked in the Lithe-compliant OpenMP with the libprocess version. We implemented a very basic fair-sharing scheduling policy for use by libprocess, where all concurrent invocations of GraphicsMagick receive roughly the same number of harts. The line labeled “libprocess” in Figure 5 shows the performance of this implementation. Clearly, the libprocess implementation dominates the convex hull of all the manually tuned variants across the range of offered throughputs (points (f), (c), (d)), providing lower latency at each load point until the machine saturates and each request receives only a single hart, at which point performance is the same as the “OpenMP = 1” case.

The workload used in testing our application servers was very homogeneous, favoring the manual static tuning used in the non-libprocess variants. With a more heterogeneous mix of image sizes, image operations (not just resize), and network speeds, a far greater degree of dynamic “scheduling” across the two levels of parallelism might be required and we would expect a greater advantage when using Lithe.

7.3 Case Study 2: Sparse QR Factorization

As our second case study, we used an algorithm for sparse QR factorization (SPQR) developed by Davis [7], which is commonly

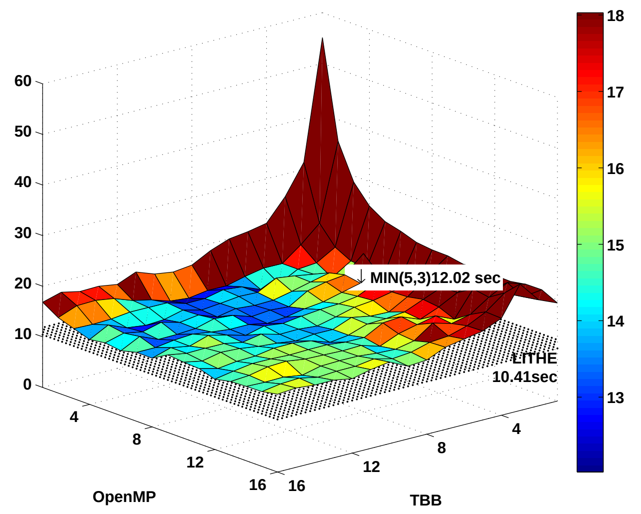


Figure 6. Original SPQR performance across different thread allocations to TBB and OpenMP for the deltaX input matrix (runtime in seconds). The performance of the Lithe version is shown as a plane to compare against all original configurations.

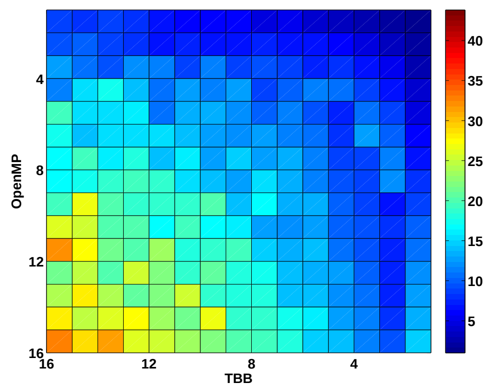


Figure 7. Average number of active SPQR threads across different thread allocations to TBB and OpenMP for the deltaX input matrix.

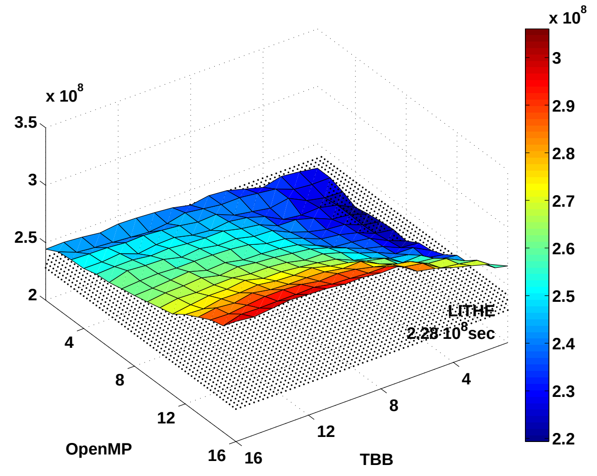


Figure 8. Number of L2 data cache misses of the original SPQR for the deltaX input matrix, compared with the Lithe version.

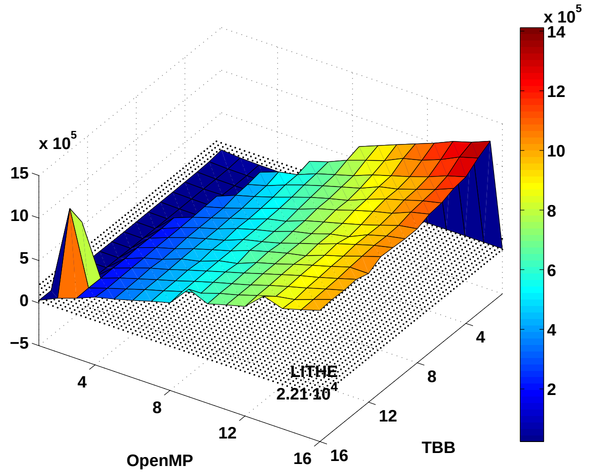


Figure 9. Number of context switches of the original SPQR for the deltaX input matrix, compared with the Lithe version.

used in the linear least-squares method for solving a variety of problems arising from geodetic survey, photogrammetry, tomography, structural analysis, surface fitting, and numerical optimization.

The algorithm can be viewed as a task tree, where each task performs several parallel linear algebra matrix operations, and peer tasks can be executed in parallel. Near the root of the tree, there is generally very little task parallelism, but each task operates on a large matrix that can be easily parallelized. Near the leaves of the tree, however, there is a substantial task parallelism but each task operates on a small matrix.

Original Out-of-the-Box Implementation. Davis implemented his algorithm using TBB to create the parallel tasks, each of which then calls parallel BLAS (basic linear algebra subprogram) matrix routines [24] from MKL. MKL, in turn, uses OpenMP to parallelize itself. Unfortunately, although the two types of parallelism should be complementary, TBB and MKL compete counterproductively with each other for resources. On an N-core machine, TBB will try to run up to N tasks in parallel, and each of the tasks will call MKL, which will try to operate on N blocks of a matrix in parallel. This potentially creates N^2 linear algebra operations, each of which were carefully crafted by MKL to fit perfectly in the cache, but are now being multiplexed by the operating system and interfering with one another.

Original Manually Tuned Implementation. To curtail resource oversubscription, Davis manually limits the number of OS threads that can be created by each library, effectively partitioning machine resources between the libraries. The optimal configuration depends on the size of the input matrix, the threading behavior of both libraries, the BLAS version in MKL, and the available hardware resources. To find the optimal configuration, Davis had to run every combination of thread allocations for TBB and OpenMP with each input matrix on each of his target machines.

We reproduced Davis' manual tuning runs for all four original input matrices on our own machine (see Table 5 for a description of the input matrices). Figure 6 shows the performance of a representative input matrix across all the different thread configurations. The out-of-the-box configuration (OMP=16, TBB=16) is where each library creates the default number of threads, which is equal to the number of cores on the machine. Although the out-of-the-box configuration is better than the sequential version (OMP=1,

	landmark	deltaX	ESOC	Rucci1
Size	71,952 x 2,704	68,600 x 21,961	327,062 x 37,830	1,977,885 x 109,900
Nonzeros	1,146,868	247,424	6,019,939	7,791,168
Domain	surveying	computer graphics	orbit estimates	ill-conditioned least-square

Table 5. SPQR matrix workload characteristics.

		landmark	deltaX	ESOC	Rucci1
Seq OMP / Seq TBB	OMP=1, TBB=1	7.8	55.1	230.4	1352.7
Par OMP / Seq TBB	OMP=16, TBB=1	5.8	19.4	106.9	448.8
Seq OMP / Par TBB	OMP=1, TBB=16	3.1	16.8	78.7	585.8
Out-of-the-Box	OMP=16, TBB=16	3.2	15.4	73.4	271.7
Manually Tuned	OMP=5, TBB=3		12.0		
	OMP=5, TBB=16			61.1	
	OMP=8, TBB=11	2.9			
	OMP=8, TBB=16				265.6
Lithe		2.7	10.4	60.4	248.3

Table 6. SPQR (TBB/OpenMP) performance with and without Lithe (runtime in seconds).

TBB=1), it is not close to optimal. Many of the configurations that limit resource oversubscription achieve better performance than the out-of-the-box configuration. We refer to the configuration with the shortest run time as the manually tuned version (OMP=5, TBB=3 for this particular input).

The top portion of Table 6 shows the performance of noteworthy configurations for all the input matrices. For all of the inputs, the out-of-the-box configuration performs much worse than the manually tuned configuration. The table includes the special cases where one of the libraries gets all of the resources (OMP=1, TBB=16 and OMP=16, TBB=1). Giving OpenMP all the resources is suboptimal because the task-level parallelism is much more scalable than matrix-level parallelism. However, giving TBB all the resources is also suboptimal because there are parts of the computation with no task-level parallelism but lots of matrix-level parallelism.

To obtain a better understanding of the original performance, we measured active threads, cache miss, and context switching behavior for all input matrices. The results for the deltaX input are shown in Figures 7, 8, and 9. As expected, Figure 7 shows that the average number of threads for the out-of-the-box configuration was more than double the number of cores in the machine. Figure 8 shows the L2 data cache misses, which increase as the number of threads given to OpenMP/TBB increase. Figure 9 shows the number of OS thread context switches that occur during the run. The number of context switches increases as the number of threads given to OpenMP increases. However, for any fixed OpenMP configuration, increasing the number of threads given to TBB decreases the number of total context switches, since the duration of the run dramatically decreases. We hypothesize that the performance anomaly for the OMP=2, TBB=14,15 configurations may be because the effects of the OS not co-scheduling worker threads from the same OpenMP parallel region are more pronounced when each thread depends on exactly one other thread, and there is a large, irregular number of threads for the OS scheduler to cycle through.

Ported Implementation. We relinked the SPQR application with the modified TBB and OpenMP libraries to run with Lithe. We did not have to change a single line of Davis' original code, since the TBB and OpenMP interfaces remained the same. The bottom portion of Table 6 shows the performance results of using Lithe. This implementation even outperforms the manually tuned configuration, because Lithe enables the harts to be more flexibly shared between MKL and TBB, and adapt to the different amounts of task and matrix-level parallelism throughout the computation.

	landmark	deltaX	ESOC	Rucci1
Out-of-the-Box	9.53×10^6	3.04×10^8	1.36×10^9	5.47×10^9
Lithe	8.23×10^6	2.28×10^8	1.11×10^9	5.19×10^9

Table 7. SPQR L2 Data Cache Misses.

	landmark	deltaX	ESOC	Rucci1
Out-of-the-Box	1.03×10^5	1.04×10^6	3.40×10^6	6.08×10^6
Lithe	1.47×10^4	2.21×10^4	7.79×10^4	2.50×10^5

Table 8. SPQR Context Switches.

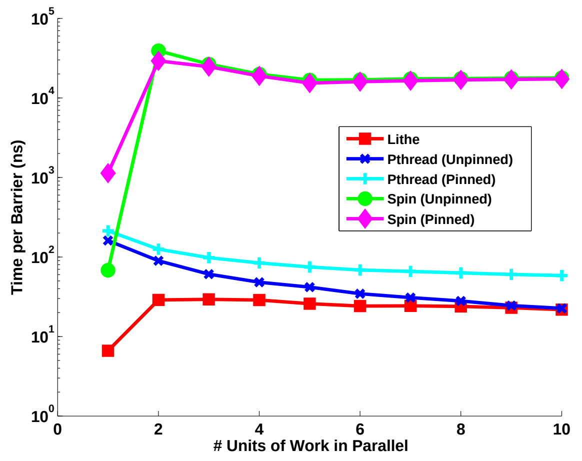


Figure 10. Performance of different barrier implementations under different parallel composition configurations in terms of runtime in nanoseconds.

Table 7 shows that the Lithe implementation has fewer L2 cache misses, and Table 8 shows that the Lithe implementation has orders of magnitude fewer OS thread context switches than the out-of-the-box configuration across all input matrices.

7.4 Barrier Synchronization

In this section, we study how synchronization and scheduling interact in the face of parallel composition using a simple barrier example. The basic unit of work for our microbenchmark consists of 16 SPMD tasks synchronizing with each other 1000 times at back-to-back barrier rendezvous (with no work in between the barrier rendezvous). We launched between 1 and 10 of the basic units of work in parallel to crudely model, for instance, GraphicsMagick or MKL routines running in parallel, each using barriers to synchronize internally.

We compared three implementations of barriers: the Glibc implementation of the pthread barrier, and the two user-level barriers described in Section 6 (one that spins, and one that cooperatively yields using Lithe). Because the implementation of the barrier depends on the task model, the Lithe barrier runs with the Lithe implementation of SPMD tasks (as described in Section 4.3), and both the pthread and spin barriers run with a pthread implementation of SPMD tasks (in which each SPMD task is simply a pthread). Furthermore, there are two different versions of the pthread SPMD implementation, one where each SPMD task is pinned to the core corresponding to its task ID, and one where the tasks are unpinned. The reported measurements reflect the performance of the SPMD implementation as well as the barrier implementation.

First, we examine the scenario where only a single unit of work is launched. The two versions of the pthread barrier perform comparably, with the unpinned version doing slightly better due to better load balancing. As expected, the unpinned spin version performs better than the pthread versions. However, the pinned spin version performs worse than the pthread versions due to load imbalance. The Lithe version performs the best, because of the lower SPMD task creation cost, and because most of the SPMD tasks can simply spin for a little bit instead of having to incur the state saving and rescheduling costs of going back into the scheduler.

Next, we examine when two units of work are launched in parallel. The pthread versions improved because they can achieve better load balancing with more tasks than cores in the machine. The spin versions slowed down by orders of magnitude because of destructive interference; each of the SPMD tasks hog up the machine resources spin-waiting rather than letting the other tasks run. The Lithe version slowed down because it must go into the scheduler after spin-waiting for a finite amount of time; however, the Lithe version is still faster than the pthread versions since the scheduling is done at user-level.

Lastly, we examine the scalability of the barrier performance as more and more units of work are launched in parallel. The unpinned pthread version achieves much better load balancing as the number of SPMD tasks increases. Lithe also achieves slightly better load balancing as the number of tasks increases, but only at the same slow rate of improvement as the pinned pthread version (since the harts are essentially pinned pthreads). Since Lithe is a user-level implementation, it is still at the mercy of the OS scheduler to schedule its harts.

8. Related Work

Previous systems have attempted to prevent resource oversubscription when *multiple applications* interfere with one another. These previous schemes fall into two main categories: OS spatial partitions [8, 27, 29], and user-level adaptive heuristics [35, 38]. We believe our work will integrate well with the first category to form a two-level scheduling ecosystem, where the OS uses minimal coarse-grained scheduling to mediate between multiple applications and gives the hardware resources allocated in each spatial partition to the user-level schedulers of each application to manage directly. We believe explicit sharing of resources is preferable to the second category of heuristic approaches that require each entity periodically query the overall system load to guess how many resources to use, as the latter does not allow prioritization between the different entities, and can easily lead to system instability.

Other systems have also attempted to prevent resource underutilization due to blocking of I/O (Scheduler Activations [2] and CPU Inheritance [13]) or synchronization (Psyche [28] and Converse [22]). In contrast, our context primitive provides a single uniform mechanism to address both I/O and synchronization blocking, while not requiring changes to existing OS systems.

This work builds on the well-known technique of expressing parallelism with continuations [42]. We improve on previous work by providing a transition stack, thus obviating the need for complex cross-context synchronization during context-switching (e.g. [10, 12, 26]). In addition, we augment previous work to support multiple parallel components, by transferring execution control to the appropriate scheduler, and ensuring that the components do not share stacks and trample on each other's execution state.

Several other systems support multiple co-existing schedulers, but with varying motivations. However, all of the previous work differ from us in one or more of the following ways:

- Converse [22] and CPU Inheritance [13] require that child schedulers register individual *threads* with a parent scheduler, effectively breaking modularity and forcing the parent to manage individual threads on a child's behalf. The child only receives harts implicitly when its parent decides to invoke these threads. Our parent schedulers grant harts to a child to do with as it pleases.
- Both GHC [26] and Manticore [12] are primarily motivated by the desire to design language primitives that enable customizable scheduling, rather than enabling interoperability of multiple custom schedulers. Although both claim to support hierarchical nesting of schedulers, neither describe how multiple schedulers would interact. Furthermore, a global entity in Manticore [12] decides how many virtual processors to allocate to each scheduler, preventing each scheduler from deciding how best to allocate resources among its children.
- Manticore [12] requires all schedulers use the same scheduling queue primitive, rather than enabling each scheduler to manage its own parallelism in a more efficient code-specific manner.
- Converse [22] does not define a standard scheduler callback interface, and thus does not support true plug-and-play interoperability. A child scheduler must know its parent's specific interface in order to register its threads.
- Unlike GHC [26] and Manticore [12], Lithe is language-agnostic. Unlike Psyche [28], CPU Inheritance [13], and HLS [33], we do not impose the adoption of a new OS.
- The virtual processor abstraction of HLS [33] only interfaces between a single pair of parent-child schedulers. Thus, granting access to a physical processor down the hierarchy involves activating a different virtual processor at every generation. Furthermore, all scheduling activity are serialized, making HLS difficult to scale to many cores.

9. Future Work

We are looking into extending this work in four main areas:

Ports of Additional Language Features. In this paper, we explored how parallel abstractions from different languages and libraries can be composed efficiently. We ported and created runtimes that support different styles of parallel abstractions, ranging from tasks to loops to actors. In the future, we would also like to port managed and functional language runtimes onto Lithe, to explore how other language features interact with scheduling and composition. For example, we imagine implementing a parallel garbage collector as a child scheduler of its language runtime, which when given its own harts to manage can decide how best to parallelize and execute memory reallocation operations while not interfering with the main computation.

Preemptive Programming Models. The current system is designed to support the large set of applications that perform well with cooperative scheduling, where preemption is often expensive and unnecessary [32]. For example, preemptive round-robin scheduling of threads within an application can introduce gratuitous interference when the threads are at the same priority level. User-level schedulers should favor efficiency over an arbitrary notion of fairness. An application may, however, want to reorder its computation based on dynamic information from inputs, events, or performance measurements. For this class of applications, we are looking into extending Lithe to enable a parent scheduler to ask for a hart back from its child.

Kernel-Level Implementation of Primitives. Our current system implements the hart and context primitives purely in user space. This enables parallel codes within an application to interoperate efficiently without the requirement of adopting a new operating system. Nonetheless, supporting the hart and context abstractions in the OS will provide additional benefits. While a user-level hart implementation can provide performance isolation between parallel codes within a single application, a kernel-level hart implementation can provide further performance isolation between multiple applications. In addition, a kernel-level context implementation would enable blocking I/O calls to interoperate seamlessly with user-level schedulers (e.g. [2]).

Management of Non-Processing Resources. Depending on the characteristics of the code and the machine, a library may be more memory and bandwidth-bound than compute-bound. By providing a better resource abstraction for cores, we have already reduced the number of threads of control that are obliviously multiplexed, thus implicitly reducing the pressure on the memory system and network. However, to give libraries greater control over the machine, we are looking into providing primitives beyond harts to represent other resources, such as on-chip cache capacity and off-chip memory bandwidth. This may require hardware support for partitioning those resources (e.g. [20, 25]), beyond that generally available in commercial machines today.

10. Conclusion

In this paper, we have shown the difficulties of composing parallel libraries efficiently. We have also argued that the management of resources should be coupled with the hierarchical transfer of control between libraries. Our solution, Lithe, is a low-level substrate that allows libraries to cooperatively share processing resources without imposing any constraints on how the resources are used to implement parallel abstractions. Using Lithe, we are able to implement multiple existing parallel abstractions with no measurable overhead, while enabling parallel libraries to be composed efficiently. We believe this capability is essential for parallel software to become commonplace.

11. Acknowledgements

We would like to thank George Necula and the rest of Berkeley Par Lab for their continued feedback on this work. We'd like to thank Tim Davis for providing us with our SPQR case study and answering all our questions about its complicated details. We would also like to thank Arch Robison and Greg Henry for their helpful feedback on our uses of TBB and MKL. We'd like to thank Ali Ghodsi and Sam Larsen, as well as our anonymous reviewers, for their helpful comments for strengthening our presentation.

This research was supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. The authors would also like to 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. Benjamin Hindman has in part been 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.

References

[1] Atul Adya et al. Cooperative task management without manual stack management. In *USENIX*, 2002.

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

[3] Animoto. http://www.animoto.com.

[4] Robert Blumofe et al. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[5] Rohit Chandra et al. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[6] Jike Chong et al. Scalable hmm based inference engine in large vocabulary continuous speech recognition. In *ICME*, 2009.

[7] Timothy Davis. Multifrontal multithreaded rank-revealing sparse QR factorization. *Transactions on Mathematical Software*, Submitted.

[8] K. Dussa et al. Dynamic partitioning in a Transputer environment. In *SIGMETRICS*, 1990.

[9] EVE Online. http://www.eveonline.com.

[10] Kathleen Fisher and John Reppy. Compiler support for lightweight concurrency. Technical report, Bell Labs, 2002.

[11] Flickr. http://www.flickr.com.

[12] Matthew Fluet et al. A scheduling framework for general-purpose parallel languages. In *ICFP*, 2008.

[13] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *OSDI*, 1996.

[14] Seth Copen Goldstein et al. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 1996.

[15] Google Voice. http://voice.google.com.

[16] GraphicsMagick. http://www.graphicsmagick.org.

[17] Benjamin Hindman. Libprocess. http://www.eecs.berkeley.edu/ benh/libprocess.

[18] Parry Husbands and Katherine Yelick. Multithreading and one-sided communication in parallel lu factorization. In *Supercomputing*, 2007.

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

[20] Ravi Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.

[21] Haoqiang Ji et al. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NASA Ames Research Center, 1999.

[22] Laxmikant V. Kale, Joshua Yelon, and Timothy Knauff. Threads for interoperable parallel programming. *Languages and Compilers for Parallel Computing*, 1996.

[23] Jakub Kurzak et al. Scheduling linear algebra operations on multicore processors. Technical report, LAPACK, 2009.

[24] C. L. Lawson et al. Basic linear algebra subprograms for FORTRAN usage. *Transactions on Mathematical Software*, 1979.

[25] Jae Lee et al. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ISCA*, 2008.

[26] Peng Li et al. Lightweight concurrency primitives. In *Haskell*, 2007.

[27] Rose Liu et al. Tessellation: Space-time partitioning in a manycore client OS. In *HotPar*, 2009.

[28] Brian Marsh et al. First-class user-level threads. *OS Review*, 1991.

[29] Cathy McCann et al. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *Transactions on Computer Systems*, 1993.

[30] Ana Lucia De Moura and Robert Ierusalimschy. Revisiting coroutines. *Transactions on Programming Languages and Systems*, 2009.

[31] Rajesh Nishtala and Kathy Yelick. Optimizing collective communication on multicores. In *HotPar*, 2009.

[32] Simon Peter et al. 30 seconds is not enough! a study of operating system timer usage. In *Eurosys*, 2008.

[33] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[34] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[35] Charles Severance and Richard Enbody. Comparing gang scheduling with dynamic space sharing on symmetric multiprocessors using automatic self-allocating threads. In *IPPS*, 1997.

[36] Stackless Python. http://www.stackless.com.

[37] Guangming Tan et al. A parallel dynamic programming algorithm on a multi-core architecture. In *SPAA*, 2007.

[38] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *OS Review*, 1989.

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

[40] Rob von Behren et al. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.

[41] Carl Waldspurger and William Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, 1994.

[42] Mitchell Wand. Continuation-based multiprocessing. In *LFP*, 1980.

[43] Samuel Williams et al. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing*, 2007.

[44] YouTube. http://www.youtube.com.