Cooperative Hierarchical Resource Management for Efficient Composition of Parallel Software

by

Heidi Pan

Bachelor of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, June 2001

Master of Engineering in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, June 2002

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Heidi Pan
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by .....
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Cooperative Hierarchical Resource Management for Efficient Composition of Parallel Software

by

Heidi Pan

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

There cannot be a thriving software industry in the upcoming manycore era unless programmers can compose arbitrary parallel codes without sacrificing performance. We believe that the efficient composition of parallel codes is best achieved by exposing unvirtualized hardware resources and sharing these cooperatively across parallel codes within an application.

This thesis presents *Lithe*, a user-level framework that enables efficient composition of parallel software components. Lithe provides the basic primitives, standard interface, and thin runtime to enable parallel codes to efficiently use and share processing resources. Lithe can be inserted underneath the runtimes of legacy parallel software environments to provide *bolt-on* composability – without changing a single line of the original application code. Lithe can also serve as the foundation for building new parallel abstractions and runtime systems that automatically interoperate with one another.

We have built and ported a wide range of interoperable scheduling, synchronization, and domain-specific libraries using Lithe. We show that the modifications needed are small and impose no performance penalty when running each library standalone. We also show that Lithe improves the performance of real world applications composed of multiple parallel libraries by simply relinking them with the new library binaries. Moreover, the Lithe version of an application even outperformed a third-party expert-tuned implementation by being more adaptive to different phases of the computation.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

I am extremely fortunate to have Professor Krste Asanović as my PhD advisor. He gave me incredible freedom to explore and mature. In fact, the Lithe project wouldn't have happened if he hadn't been so patient letting me learn my way up the system stack. He has also been my greatest resource, being so creative and knowledgeable. Thanks Krste for being such an amazing guiding force in my life for the past decade!

Thanks to Professor Arvind and Professor Saman Amarasinghe for agreeing to be on my committee, and for all of their great feedback on Lithe. Thanks again to Arvind for being my first and only UROP advisor, and introducing me to the wonders of computer architecture.

I am lucky to have met great mentors outside of school. Thanks to Robert Cohn, Geoff Lowney, and Joel Emer for always giving me great advice and encouragement. Thanks to Alan Khazei, who inspired me with the idea that any definition of a successful life must include serving others.

The best thing I did for Lithe was to convince Benjamin Hindman to join the project. He was an integral part of designing and implementing Lithe, as well as porting TBB and OpenMP. Being able to bounce ideas, get a fresh PL perspective, and simply hanging out into the wee hours working on paper submissions with him also made the project that much more enjoyable.

Thanks to Rimas Avizienis for porting his audio processing application to Lithe, and helping us to better understand real-time issues, especially since I seem to have a knack for needing him when he's at his busiest.

Thanks to Chris Batten for a conversation on the whiteboard of Cory Hall in the very early days of Lithe, in which he challenged me to really define what "scheduling" means. That led to our turning point of giving parent schedulers the control over how to allocate resources to their children, thus making Lithe into a truly hierarchically modular design. Chris' immense curiosity and logical reasoning has also been a real inspiration for me.

Thanks to Arch Robison for explaining the intricacies of TBB and IRML, as well as pointing us to the SPQR application. Thanks to Tim Davis for providing us with the SPQR code, and answering our many questions. Thanks to Greg Henry for helping us understand the behavior of MKL. Thanks to Niklas Gustafsson and David Callahan for interesting chats about ConcRT.

Rose Liu moved across the country with me three times in three consecutive years to follow Krste between Boston and Berkeley. I miss the days at ICSI, where we would dive into hours-long discussion on the topic of the day. I also never cease to be amazed at her tenacity to dig to the bottom of something.

I worked on the MTR project with Ken Barr. It was my first group project, and first

concrete project on SMPs. This was the project that really taught me how to design something from scratch, and piqued my interest in cache coherence. Ken was also my hero for coming up with his own thesis topic from being inspired from a talk he attended.

Although I barely overlapped with Emmett Witchel in grad school, he not only set the precedence for systems research in our group, but the precedence for the design philosophy of providing efficient hardware mechanisms and enabling flexible software policies.

The people at MIT CSAIL taught me the important lesson of blurring the line between colleagues and friends. It was great to have gone through graduate school with Ronny Krashinsky, Michael Gordon, Sam Larsen, Mark Stephenson, Rodric Rabbah, Mark Hampton, Jessica Tseng, Steve Gerding, Jaewook Lee, Ian Bratt, Asif Khan, David Huynh, Jonathan Ragan-Kelley, and many more.

Thanks to everyone at the ParLab for really opening my eyes up to the entire vertical stack, and making Berkeley instantaneously feel like a second home: Leo Meyerovich, Sam Williams, Kaushik Datta, Rajesh Nishtala, Archana Ganapathi, Bryan Catanzaro, Jike Chong, Barrett Rhoden, Kevin Klues, David Zhu, Henry Cook, Sarah Bird, Scott Beamer, Andrew Waterman, Juan Colmenares, Karl Fuerlinger, Jimmy Su, Shoaib Kamil, Mark Hoemmen, Jim Demmel, John Kubiatowicz, Ras Bodik, Kurt Keutzer, Tim Mattson, David Wessel, John Shalf, and many more.

Thanks to Mary McDavitt for making it logistically possible for me to be physically on the West Coast and still graduate from MIT, and for making me feel like I still have roots back in Boston.

To Douglas Yeung and Alan Chen, my PhD (and life) comrades from the same year. To Vijay Janapa Reddi, who I'm really glad got stuck with me in a small cubicle at Intel Hudson for an entire summer.

I am blessed to be surrounded by loving friends and family. You know who you are, and I would be a completely different person without you in my life. Thanks to Michael Zhang for being my rock, and for being infinitely patient when I'm always stuck in grad school 3,000 miles away, regardless of which coast he's on. Thanks to my parents, Chung-Hsing Ouyang and Chung-Shih Pan, for being the most supportive and caring parents a girl could ever wish for.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building applications by composing modular libraries is a cornerstone of software engineering. Modularity enhances productivity. Functionality can be implemented once, and reused in different contexts and applications. Different libraries can be designed, built, and maintained independently by different teams. A complex application can be decomposed into smaller library modules that are easier to reason about. Modularity can also help with performance. A programmer can leverage the performance of a highly optimized library without having the expertise and time needed to achieve that level of sophistication. Finally, modularity enables specialization. Each library can implement its functionality in the most efficient manner, without having to accommodate the needs of other libraries and incurring the overhead of generalization. Each library can also use high-level, tailored abstractions that are most natural for expressing its own computation. Different libraries can choose different productivity-performance tradeoffs. A performance-critical portion of an application can be hand-coded in assembly, while a user interface portion can be written in a domain-specific language to enable rapid functionality updates.

With the increasing adoption of multicore microprocessors and parallelization of software libraries, modularity is evermore important. Parallel programming is hard. It is difficult to write correct, let alone optimized, parallel software. Programmers must be able to leverage each other's efforts through code reuse. Most programmers will not understand parallelism, and instead work at a higher abstraction level, so they must be able to use libraries without caring whether the libraries are parallel or sequential. Furthermore, we expect specialization to play a big role in delivering energy-efficient performance. Having more, but simpler, cores shifts much of the responsibility for attaining performance from hardware to software. But this also opens up the opportunity for software to leverage code-specific information to achieve the desired level of performance more efficiently, which is especially crucial now that battery life and electricity bills are first-order concerns.

Unfortunately, today's software cannot be modular because parallel composition does not work well. Composing disparate parallel libraries within an application can often cause the libraries to interfere with one another and degrade the overall performance. In fact, a

parallel implementation of an application may actually run slower than a sequential implementation [Daved]. Current day parallel software vendors are becoming increasingly aware of this problem, but have only supplied ad hoc solutions. For example, Intel's Math Kernel Library (MKL) advises against using more than one parallel library within an application. It instructs its clients to call the *sequential* version of the library whenever it might be running in parallel with another part of the application [Int07]. 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 export the problem to its clients.

The crux of the problem is that systems today provide an inadequate foundation for building parallel programs: *OS threads*. The operating system exports threads as the abstraction for processing resources. User-level codes can create as many threads as they want, and map their computation onto these threads for parallel execution. The OS then time-multiplexes the threads onto the physical cores. However, threads are not free. The more threads there are, the more time it takes the OS to cycle through all of them, causing each thread to run at more infrequent and unpredictable times. Computation that is supposed to be tightly synchronized may not be able to run simultaneously, causing gratuitous waiting. Too many threads can also easily lead to resource oversubscription. More threads of execution will likely access more data, kicking each other's working set out of the shared caches and causing more TLB misses and page faults. Furthermore, most parallel codes cannot achieve perfect scalability, so using more threads to do the same amount of work produces diminishing returns.

One common solution is for user-level codes to only create as many OS threads as there are physical cores in the machine. However, this is a difficult goal to achieve in a modular fashion. Even if each software component creates a reasonable number of threads, the application can still collectively create too many threads when its components run in parallel. Hence, we need a systemic solution to coordinate the different software components within an application, rather than having the components implicitly use and share the machine resources by creating OS threads.

This thesis presents *Lithe*, a framework that enables efficient composition of parallel software components. Lithe provides the basic primitives, standard interface, and thin runtime to enable parallel codes to efficiently use and share processing resources. In particular, Lithe replaces the virtualized thread abstraction with an unvirtualized *har*dware *t*hread primitive, or *hart*, to represent a processing resource (Figure 1-1). Whereas threads provide the false illusion of unlimited resources, harts must be explicitly allocated and shared amongst the different software components. Lithe can be inserted underneath the runtimes of legacy parallel software environments to provide *bolt-on* composability – without changing a single line of the original application code. Lithe can also serve as the foundation

for building new parallel abstractions and runtime systems that automatically interoperate with one another.

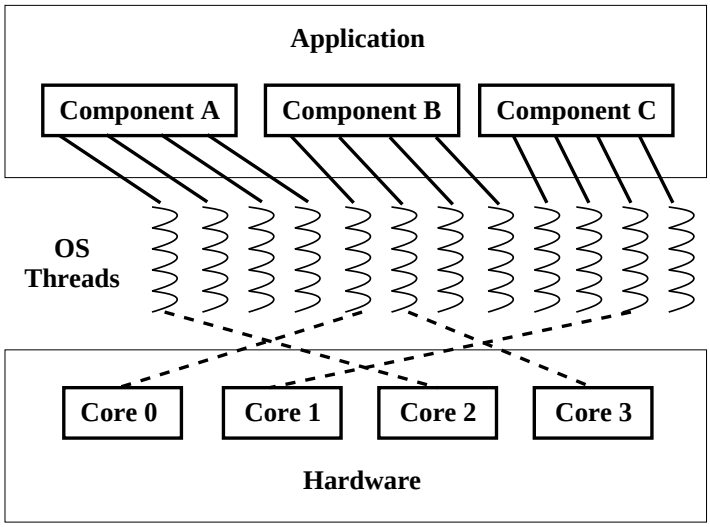
1.1 Parallel Programming Models

To better understand the overall composition problem, it helps to first understand the individual components that may be composed. This section describes how each parallel software component is built using a particular programming model, and the potential diversity of programming models that can be composed.

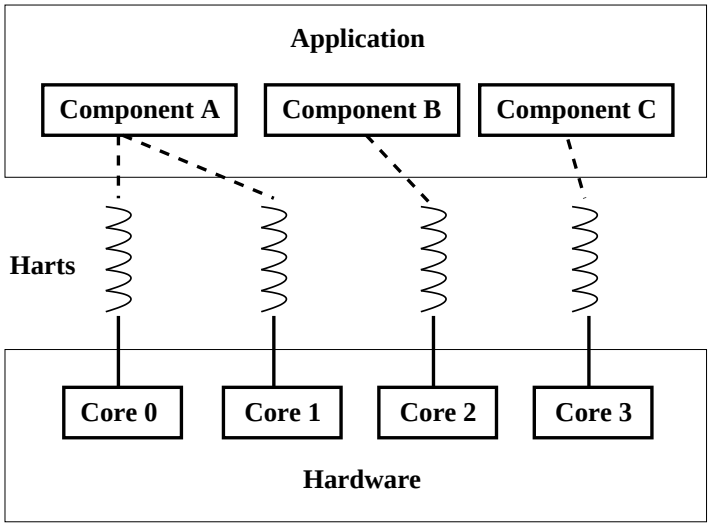
Each parallel software component encapsulates a computational problem to be computed by multiple processing resources. Its programming model governs how computational behaviors and needs are translated into execution policies. Figure 1-2 shows the typical steps in using a programming model to map computation onto resources. First, a programmer decomposes the computation into smaller, independent units of work, or *tasks*. Then, a language or library runtime – a *scheduler* – decides where and when to run the tasks. A programming model improves productivity by equipping the programmer with abstractions for framing the problem. It improves performance by equipping the scheduler with information on how the work should be executed. Finally, it improves efficiency by directly conveying pertinent information between the programmer and the scheduler.

Parallel programming models can vary in three main dimensions: computational pattern, level of abstraction, and restrictions on behavior. We describe each of those dimensions below, and discuss how choosing different design points along a dimension can affect productivity, performance, and efficiency.

1. Different programming models may embody different *computational patterns* [Mat04]. It is more natural to express certain types of computations using certain programming models. For example, fork-join parallelism is often well-suited for divide-and-conquer algorithms [BJL+95], whereas pipeline parallelism is often well-suited for video decoding algorithms [DHRA06]. The right programming model also helps the programmer decompose the computation in a way that minimizes the dependencies between the tasks. This boosts productivity, as fewer interface dependencies lead to a cleaner, more modular software architecture. This also boosts performance, as fewer data dependencies translate into less synchronization overhead and fewer scheduling restrictions. Furthermore, each computational pattern may encapsulate information that can be used by schedulers to order and co-locate tasks intelligently. For example, a transactor [Asa07] represents a group of tasks sharing state that should be scheduled onto the same core to maximize locality. Transactors connected by channels should be co-scheduled on neighboring cores to minimize communication costs. Using an appropriate programming model improves productivity and efficiency by sparing the programmer from having to annotate low-level performance-critical information and



(a)



(b)

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

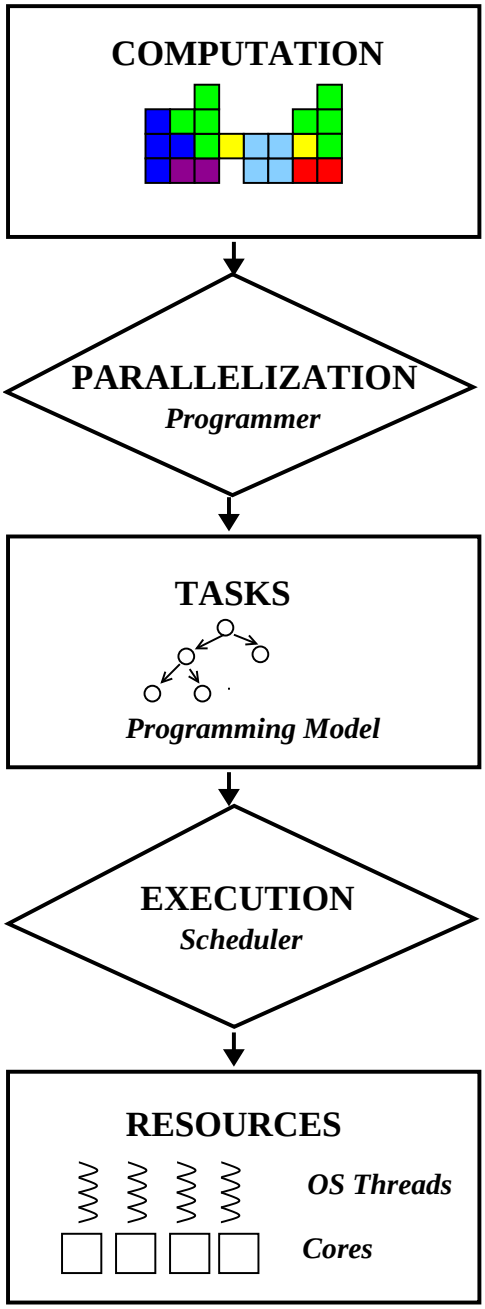


Figure 1-2: The two main steps of mapping computation onto resources, encapsulated within each parallel software component. First, the programmer decomposes the computation into tasks. Then, the scheduler executes the tasks with its resources. The parallel programming model provides the task abstraction, and governs how computation behaviors and needs are translated into execution policies.

the scheduler from having to process the annotations.

2. The *level of abstraction* may vary widely for different types of programming models. Domain experts with only tangential knowledge of programming may want to use high-level domain-specific abstractions. For example, MAX/MSP [Cyc] enables musicians to use a graphical drag-and-drop "language" interface to compose music synthesis building blocks, such as filters and oscillators. While higher-level abstractions may come with a heavier performance tax [BBK09], they enable non-experts to become productive, and create opportunities to convey domain-specific information that is relevant to scheduling decisions in a natural and efficient manner. A musician could use a language construct to express multiple input channels, for example, which would inform the language runtime scheduler about the potential parallelism between the channels. In contrast, that domain-specific knowledge about channels being independent may become lost if the musician were to use a more general programming model, or if the channel construct were implemented by layering on top of a more general abstraction. On the other hand, a low-level system programmer may know exactly how a computation should be executed to wring out every last drop of performance. Such a programmer could use low-level language constructs, such as Go's co-routines [Goo] to control the ordering of tasks, or Sequoia's array blocking and task mapping primitives [FKH+06] to control the movement of data across the memory hierarchy.

3. Programming models are often useful because of *what they forbid*, rather than what they allow. Restricting the space of possible behaviors makes the code easier to understand and more likely to be correct. A deterministic language can greatly reduce complexity, improve debuggability, and curtail data races when nondeterminism is not needed [BAAS09]. Partitioned global address space (PGAS) languages define data objects to be private by default, and help to prevent unintended sharing that often leads to hard-to-find bugs [YBC+07]. A more restrictive programming model can also simplify the corresponding runtime and scheduler. A general hierarchical task model may warrant a generic stack frame to represent each task, scheduling queues to store dynamically spawned tasks, and a sophisticated work-stealing algorithm to load balance the tasks. In contrast, a simpler data-parallel model could be implemented with just a global atomic counter that tracks the index of the next array element to be processed.

Using the wrong programming model for a particular computation can result in subpar performance. For example, Cilk [BJL+95] is considered to be high-performance and general purpose. However, it lacks expressiveness and destroys cache locality for certain linear algebra operations, achieving worse performance than an equivalent pipeline implementation [KLDB09]. Cilk has a fork-join programming model that enforces the completion of

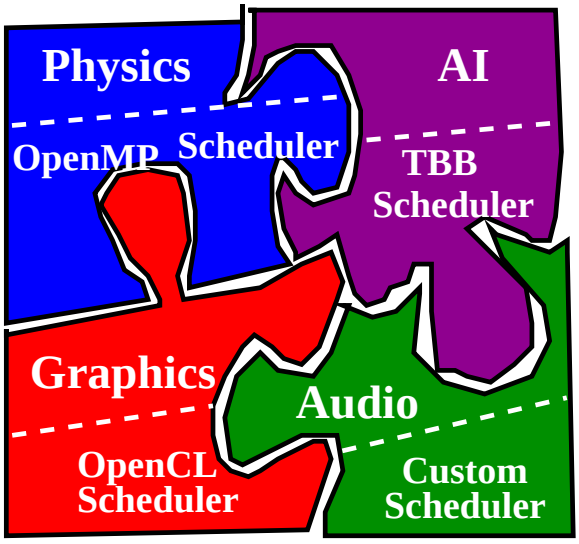


Figure 1-3: An example gaming application composed of several parallel components, which are built in turn using different parallel libraries. The physics component is written using OpenMP [CMD+01]. The AI component is written using TBB [Rei07], the graphics component is written using OpenCL [Gro]. The audio component is written using a customized framework.

all child tasks before executing their parent's successor. In contrast, the pipeline model imposes no such false dependencies, allowing the necessary operations to be overlapped to avoid gratuitous idling. Additionally, Cilk is designed for divide-and-conquer algorithms, so it assumes that tasks within the same subtree of the task graph share the same data. But the most natural way of expressing the linear algebra computation in Cilk results in subtrees that contain tasks of the same type of operation working on different sets of data. The pipeline model is able to execute operations that share data consecutively on the same core. An alternate Cilk implementation groups operations that share data into the same task, which improves the locality but creates load imbalance and exacerbates the idling problem.

To achieve its desired level of productivity, performance, and efficiency, each component needs to utilize an appropriate programming model. Thus, we need to support the efficient composition of multiple parallel programming models.

1.2 Composing Programming Models

In practice, an application can be composed of many different parallel programming models. Figure 1-3 shows an example gaming application that incorporates physics, graphics, audio, and AI components. Each component can be built using a different parallel library or language, because they have very different computational behaviors and constraints (as described Section 1.1), or simply contain legacy or third-party codes that cannot be changed.

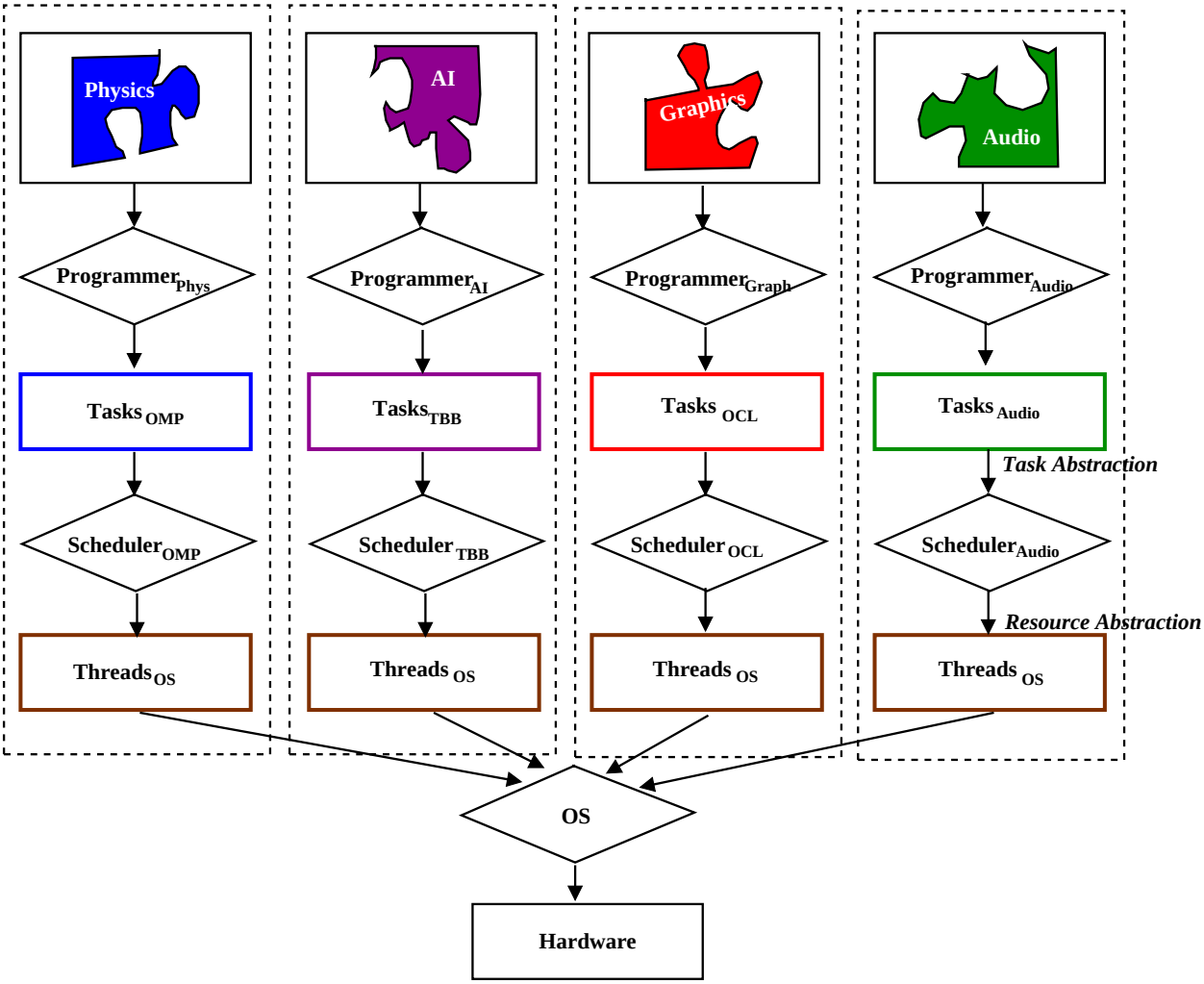


Figure 1-4: The main steps of mapping computation onto resources in the face of composition. Each scheduler supports a potentially different task abstraction, but uses the same resource abstraction. Each parallel software component is parallelized and managed independently, leaving the integration of components to the OS.

There are numerous potential complications in composing these heterogeneous components, from transposing data between different formats, to creating new language bindings. This thesis chooses to address a challenge that is unique to composing parallel components: the composition of multiple schedulers.

Figure 1-4 shows the steps of mapping computation onto resources in the face of composition. Each scheduler supports a potentially different task abstraction, but uses the same resource abstraction. The operating system virtualizes processing resources using threads to provide schedulers with autonomy and a clean execution environment. A scheduler operates on its own set of OS threads, and does not need to know about other schedulers. Each parallel software component is parallelized and managed independently, leaving the integration of components to the OS. This enables each component to be specialized by using an appropriate programming model, while enabling multiple parallel programming models to co-exist within the same application.

However, resource virtualization only provides design modularity, not performance isolation. Although each component's computation is decomposed and mapped independently onto its own OS threads, the different components can still interfere with one another during execution when all threads are time-multiplexed onto the same set of processing resources. Components can collectively create too many threads and a large memory footprint, inadvertently causing resource oversubscription and cache thrashing. Worse yet, schedulers often cannot function properly because they naively assume uninterrupted access to all of the machine's resources. A work-stealing scheduler [BJL+95, Rei07] should only spend the effort of stealing more tasks to execute on an "idle" core if no other schedulers are using that core. Ordering tasks to maximize the temporal reuse of cache-sized data blocks [WOV+07, TSG07] only works if no other schedulers are using the same cache. Overlapping the computation of the current task with prefetches for the next task to hide the latency of data transfers [TSG07] becomes useless if another scheduler evicts the prefetched data before it can be used. The execution behavior of each software component depends heavily on the OS thread-multiplexing policy and other components. It is thus difficult for any one scheduler to reason about its own behavior, let alone optimize its performance.

The main challenge, then, is to preserve design modularity between parallel software components, while giving each component more control over its own execution. We argue that this is best achieved by exposing *unvirtualized* hardware resources and sharing them cooperatively across parallel components within an application. Each scheduler would manage its own set of resources. This provides schedulers with the same type of autonomy and freedom to specialize as they had when managing their own threads. But given unvirtualized resources, the schedulers can now be effective in managing the parallel execution of its computation. The only tradeoff is that schedulers have to be cognizant of the fact that they may be sharing the machine resources with other schedulers. We believe that this slight increase in the complexity of low-level scheduling code is a small price to pay

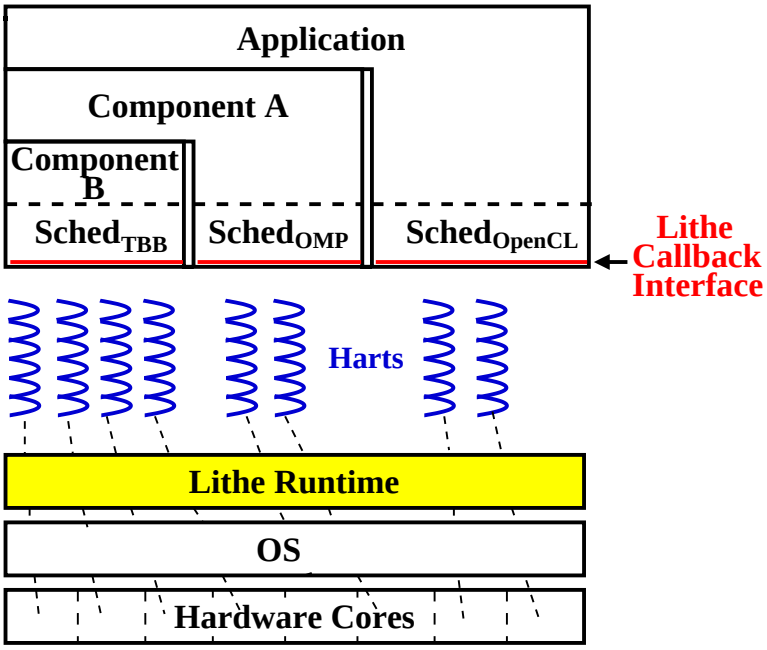


Figure 1-5: The system stack with Lithe. Lithe establishes a hierarchy of schedulers based on the call structure of their corresponding computation. Each scheduler implements the standard Lithe callback interface, and calls into the Lithe runtime to use Lithe primitives for parallel execution.

to gain not only performance and efficiency, but also separation of concerns. High-level application developers can productively compose arbitrary libraries, instead of needing to understand the implementation of each library to avoid destroying performance through combining multiple parallel programming models.

1.3 Lithe Overview

This thesis presents Lithe, a framework for schedulers to dynamically and flexibly share processing resources. Lithe provides a minimal common ground to enable an arbitrary and diverse set of schedulers to interoperate and compose efficiently. Lithe consists of the following components, as illustrated in Figure 1-5:

- Lithe establishes a *cooperative, hierarchical relationship* between all schedulers within an application based on the hierarchical call structure of their corresponding computation. The scheduler hierarchy distributes control over processing resources in a modular, specializable fashion. Parent schedulers choose how to allocate resources for their child schedulers to use, but do not micromanage how these children use the given resources.

- Lithe exports basic *primitives* for schedulers to use and share processing resources. A *hart* represents a processing resource that is explicitly allocated and shared. A

context acts as the execution vessel for the computation running on the hart, so that the current computation can be suspended (and perhaps resumed elsewhere) without blocking the current hart.

- Lithe defines a standard *scheduler callback interface* that is implemented by all schedulers that want to manage their own harts and interoperate with other schedulers. The callbacks encapsulate how a scheduler manages harts and child schedulers.

- Lithe provides a thin, user-level *runtime system* to facilitate the sharing of harts between schedulers, as well as execution with contexts. The runtime keeps track of the scheduler hierarchy, connecting children with their parent and invoking the appropriate callbacks to transfer harts between schedulers. The runtime also starts, pauses, and resumes contexts on behalf of their corresponding scheduler.

Chapter 2

Cooperative Hierarchical Scheduling

This chapter describes Lithe's *cooperative hierarchical scheduling* model that enables modularity and specialization of parallel codes without sacrificing overall application performance. Lithe advocates explicit resource *sharing*, rather than resource virtualization, to compose an arbitrary number of schedulers to use a finite set of machine resources. Below, we describe the common resource model used by each scheduler, and the hierarchical interaction between the schedulers to share resources.

2.1 Resource Model

The Lithe resource model presents an accurate view of the machine resources that enables parallel components to achieve efficient modular scheduling. At any point in time, each scheduler is allocated an exclusive subset of the resources available to the application. Since an application can potentially employ many schedulers, each scheduler may not get as many resources as it would like. But once a scheduler is allocated a resource, it can do whatever it wishes with that resource. A scheduler also keeps its resources until it explicitly releases them. This cooperative resource model ensures that no other schedulers within the same application can use those resources simultaneously and cause unintentional interference. In addition, a scheduler can allocate and yield each of its resources independently to enable efficient fine-grained sharing. This curbs resource undersubscription by allowing a scheduler to relinquish individual resources whenever they are not needed, such as when a scheduler has to wait for additional tasks to be spawned, or for a synchronization or long-running I/O operation to complete.

Lithe enables modular scheduling by introducing resource sharing into the interface between parallel software components. Each software component encapsulates functionality, data, and execution control flow. While sequential components only specify a single thread

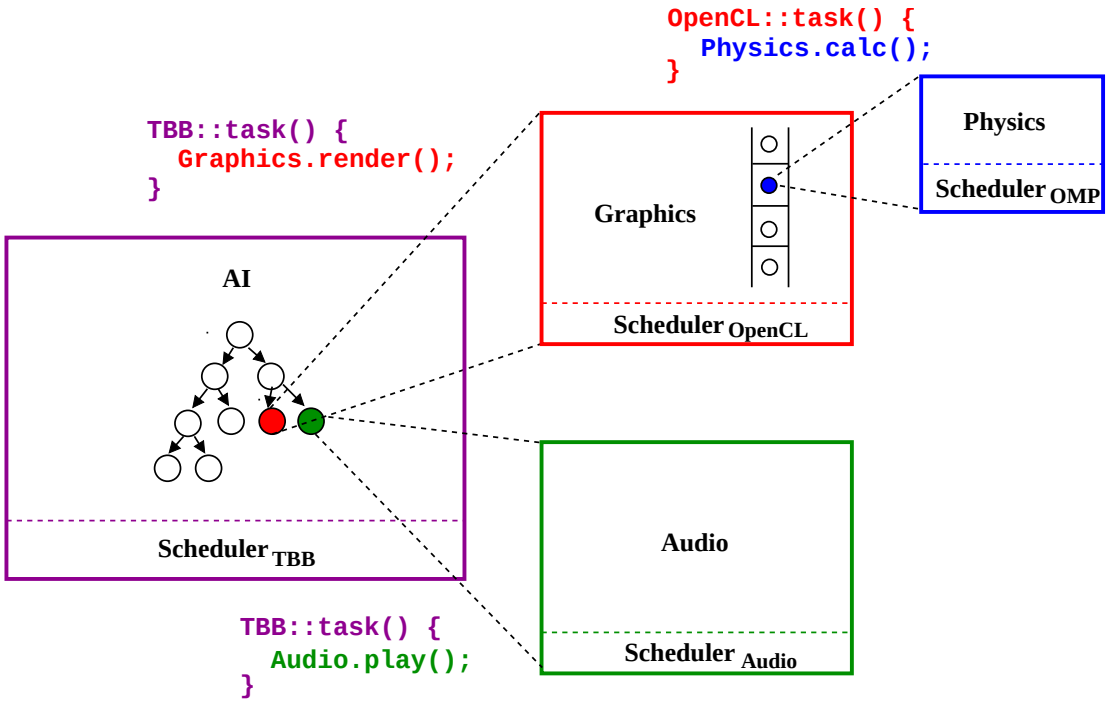
of control, parallel components may encapsulate multiple threads of control that may change dynamically based on the amount of available resources. With Lithe, each parallel component is given processing resources to manage without external interference or concern. In contrast to using virtualized resources, each scheduler can now precisely control the parallel execution of its computation and adapt its behavior to the number of available resources, rather than simply delineating the potential parallelism by specifying an arbitrary number of threads of execution a priori.

2.2 Cooperative Scheduler Hierarchy

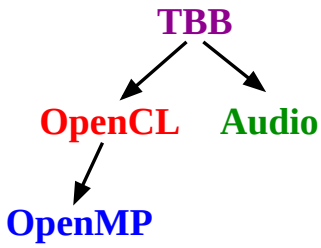
Lithe establishes a hierarchy of schedulers for cooperatively sharing resources based on the hierarchical call structure of their corresponding computation. Each software component within an application can be a hierarchical composite of other software components. One component's "task" can contain both computation and a scheduler to manage that computation. Figure 2-1(a) shows the hierarchical structure of the gaming application example from Section 1.2. The AI component controls all of the gaming logic, and is implemented using TBB tasks. It invokes a graphics component to render some objects as one of its tasks, and invokes an audio component to play some sounds as another task. The graphics component, in turn, invokes a physics component as one of its OpenCL tasks to calculate the trajectory for each rendered object.

Figure 2-1(b) shows the corresponding scheduler hierarchy. The scheduler of the calling component is the *parent*, while the scheduler of the callee is the *child*. A parent scheduler allocates resources to its child schedulers, and its children give back the resources when they are done with them. By coupling resource allocation with control flow, Lithe makes each caller responsible for providing its callees with the resources needed to perform work on its behalf. This is already true in the sequential world, where a caller implicitly grants its only processing resource to a callee until the callee returns.

The scheduler hierarchy distributes control over resources in a hierarchically modular fashion. Every scheduler only has to communicate with its parent and its children. Peer schedulers are insulated from having to know about each other. Each leaf scheduler maps its unit tasks onto its allocated resources. Each parent scheduler *"schedules"* its composite tasks by deciding how to allocate its resources between the corresponding child schedulers. This enables a scheduler to prioritize between its tasks and accurately account for the usage of its own resources. But a parent does not micromanage how its children use their given resources, allowing the children to specialize as they please.



(a)



(b)

Figure 2-1: (a) The hierarchical call structure of the example gaming application. The AI component invokes the graphics and audio components as two of its TBB tasks. The graphics component, in turn, invokes the physics component as one of its OpenCL tasks. (b) The corresponding hierarchy of schedulers. TBB allocates resources between OpenCL, the custom audio scheduler, and itself. OpenCL allocates resources between OpenMP and itself.

Chapter 3

Lithe Primitives

Lithe provides two basic primitives, harts and contexts, to enable schedulers to perform efficient parallel execution. The *hart* primitive represents a bare-metal processing resource. The *context* primitive acts as the execution vessel for computation to run on a hart. This chapter describes each of these primitives in more detail, while the next two chapters describe how schedulers can use the Lithe callback and runtime interfaces to operate on these primitives.

3.1 Harts

A *hart*, short for *har*dware *t*hread, represents a processing resource. Within an application, there is a fixed one-to-one mapping between a hart and a physical hardware thread. There is typically one hardware thread per physical core, except for multithreaded machines like SMTs [TEE+96] where there are multiple harts per core.

The hart primitive is fundamentally different from the OS thread abstraction. Whereas a scheduler can create an unlimited number of threads on-demand, a scheduler must explicitly request harts and only use the harts that it is allocated. Threads can also be used to support various levels of abstractions. A scheduler can create as many threads as there are physical cores in the machine, and use them as a *resource* abstraction. Or a scheduler can create as many threads as there are tasks, and export them as a *programming* abstraction to its user. In contrast, a hart only represents resources, and a scheduler must explicitly multiplex its tasks onto its harts.

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 current scheduler of a hart has exclusive control over not only the physical processing units, but also any private caches, TLB, and local store associated with the hardware thread. This prevents unintended inter-component cache and TLB thrashing, as well as unnecessary context switching overheads.

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, while allowing the underlying hart to continue "beating" and running other contexts. A blocked context stores the continuation of the computation, including the stack that the computation was using, and any context-local state.

Contexts support efficient overlap of communication with computation. A pending synchronization or I/O operation can block its context, and return the hart back to the current scheduler to do useful work in the meantime. The current scheduler can then choose to run another task or give the hart to another scheduler. Whereas OS thread-multiplexing may also achieve overlapped computation and communication, it does so by forcing fine-grained sharing without any regard to whether there are even synchronization or I/O latencies to mitigate, and without any say from the current scheduler about what to run instead.

Each scheduler is responsible for allocating and managing its own contexts, giving it full control over how to manage its stacks, and enabling optimizations such as linked stacks [vBCZ+03], stacklets [GSC96], and frames [Sta]. It is important for each scheduler to be able to manage its own stacks. Schedulers often store the execution state in the stack, so the stack implementation dictates both the task creation overhead and the parallel execution model [GSC96]. A scheduler can also allocate and manage any context-local state.

Chapter 4

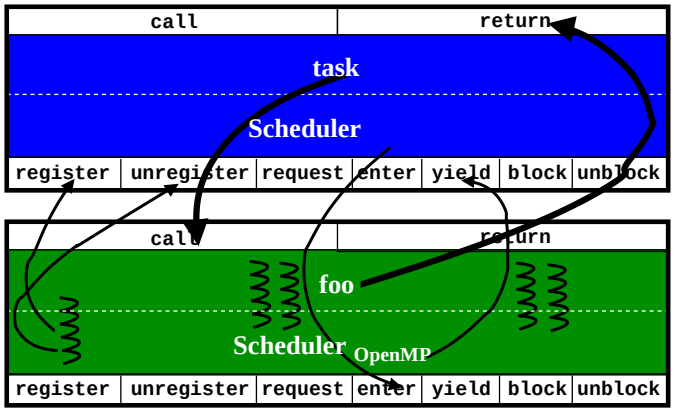
Lithe Scheduler Callback Interface

This chapter describes Lithe's callback interface (Table 4.1), which is implemented by all schedulers within an application. The standardized interface enables any parent and child scheduler to share harts, providing interoperability between diverse parallel software components. The scheduler callback interface is orthogonal to the function call interface, as illustrated in Figure 4-1. This enables the separation of concerns between high-level functionality and low-level resource management, as well as provides "plug-and-play" interchange of any sequential or parallel implementation.

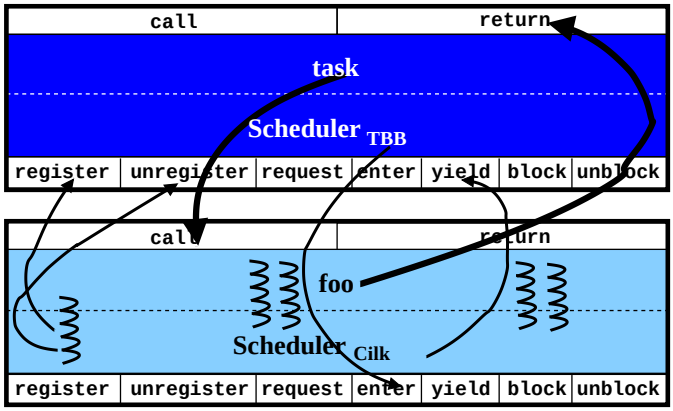
The callbacks form the backbone of Lithe's distributed modular scheduling. Upon instantiation, each scheduler is only aware of its current hart and parent scheduler. It is dynamically given additional harts or information from its environment whenever one of its callbacks is invoked. Callbacks fall under two categories: transfer and update (Figure 4-2). Using a hart to invoke a scheduler's *transfer callback* effectively transfers control of the hart over to the scheduler in a continuation-passing style. Using a hart to invoke a scheduler's *update callback*, on the other hand, only lets the scheduler borrow the hart to process the given information and update any relevant scheduling state. Below, we describe each of the callbacks in more detail.

Callback	**Type**	**Invoker**
`void` **register**`(lithe_sched_t child);`	Update	Child Scheduler
`void` **unregister**`(lithe_sched_t child);`	Update	Child Scheduler
`void` **request**`(lithe_sched_t child, int nharts);`	Update	Child Scheduler
`void` **enter**`();`	Transfer	Parent Scheduler
`void` **yield**`(lithe_sched_t child);`	Transfer	Child Scheduler
`void` **block**`(lithe_ctx_t ctx);`	Transfer	Computation
`void` **unblock**`(lithe_ctx_t ctx);`	Update	Computation

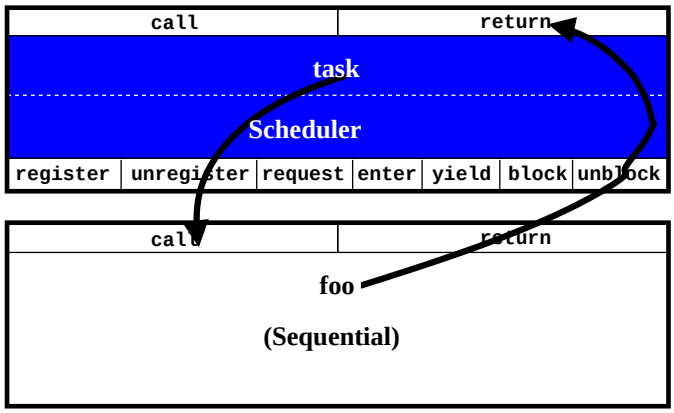
Table 4.1: The standard Lithe callbacks.




```
tbb::task(){
  foo(){
    #pragma OMP parallel
    :
  }
}
```




```
tbb::task(){
  foo(){
    spawn bar();
    :
  }
}
```




```
tbb::task(){
  foo(){
    :
    :
  }
}
```

Figure 4-1: The Lithe scheduler callback interface, which is orthogonal to the function call interface, enables the plug and play of an OpenMP, Cilk, or sequential implementation of `foo`. Furthermore, there is a separation of concerns between the high-level functionality of `task` invoking `foo`, and the low-level resource management of their respective schedulers exchanging harts.

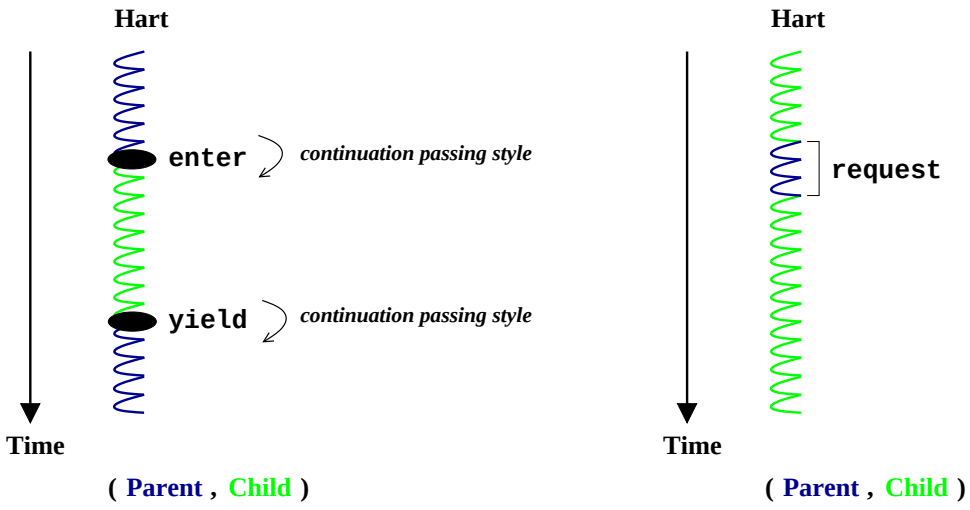


Figure 4-2: (a) Transfer callbacks transfer control of the hart from one scheduler to another in continuation-passing style. (b) Update callbacks allow a scheduler to borrow a hart temporarily to process the given information and update its internal state.

4.1 Transfer Callbacks

Transfer callbacks are Lithe's minimalistic mechanism for a scheduler to transfer control of one of its harts to another scheduler. A scheduler effectively gains control of a hart when one of its transfer callbacks are invoked using that hart. The scheduler's code inside the transfer callback decides what to do with the hart, then carries out the decision by either executing a task with the hart or transferring the hart to another scheduler. Transfer callbacks are essentially entry points for additional harts to dynamically join a parallel software component after an initial hart has invoked the component and instantiated its scheduler.

Each transfer callback is invoked with a special context provided by the Lithe runtime system for the hart, called the *transition context*. A transition context also provides a small transition stack for the callback to execute with before it starts or resumes one of the scheduler's own contexts. Transition contexts remove the need to share stacks across parallel components during hart transfers by decoupling harts from stacks. This enables components to manage their stacks in diverse ways. This also prevents a component from accidentally trampling on another component's state, or inhibiting another component's continuation stack frame from being resumed. Because transfer callbacks are invoked with a transition stack, they must not return, but must instead explicitly resume another context or invoke another scheduler's transfer callback. Note that parallel languages and libraries will typically provide their tasks with a higher-level execution environment that abstracts

away harts and contexts, so only low-level scheduling codes will have to be written in such continuation-passing style.

Transfer callbacks enable a scheduler to inform another scheduler that it can use a hart by simply giving it the hart. No other harts need to be involved. The donor scheduler does not need to interrupt the recipient scheduler, and the recipient does not need to poll periodically to see if it has received more harts. Furthermore, a scheduler does not need to hand off tasks (in the form of threads) for its parent scheduler or the OS to schedule in order to access additional processing resources. This enables modularity, as a parallel component does not need to expose how it manages its parallelism. This also enables greater efficiency, as schedulers can adapt their policies to the number of available harts at any given time, and avoid paying the overhead of parallelization (including stack creation) until additional harts become available.

Below, we describe the three transfer callbacks defined by Lithe, which are differentiated by where the hart originated.

Enter The `enter` callback is invoked by a scheduler's parent to allocate the hart to the scheduler. The scheduler can use its additional hart to execute its unit tasks, or to grant to one of its children.

Yield The `yield` callback is invoked by a scheduler's child to give back a previously allocated hart. The scheduler can remember the fact that this child has yielded one of its harts to influence its future hart allocation decisions. The scheduler can then use the hart to execute other computation, grant the hart to another child, or return the hart back to its own parent.

Block The `block` callback is invoked with a hart that is currently managed by the scheduler. Rather than being invoked by another scheduler, the callback is invoked from within an executing task, typically by a synchronization or I/O library. The callback notifies the scheduler that the context previously running on the hart has been paused, and that the hart is now free to do other work. The callback enables a synchronization or I/O library to cooperate with the current scheduler to achieve efficient overlap of communication with computation. In addition, this allows a synchronization or I/O library to be interoperable with a wide range of schedulers, as well as allowing data, files, and devices to be shared and synchronized between tasks that are managed by different schedulers.

Example Each scheduler typically keeps track of its harts, children, and tasks. A scheduler's callbacks access and update this internal state concurrently to coordinate the parallel execution of its computation across multiple harts. Figure 4-3 shows an example scheduler with a task queue that is accessed by each of its `enter` callbacks.

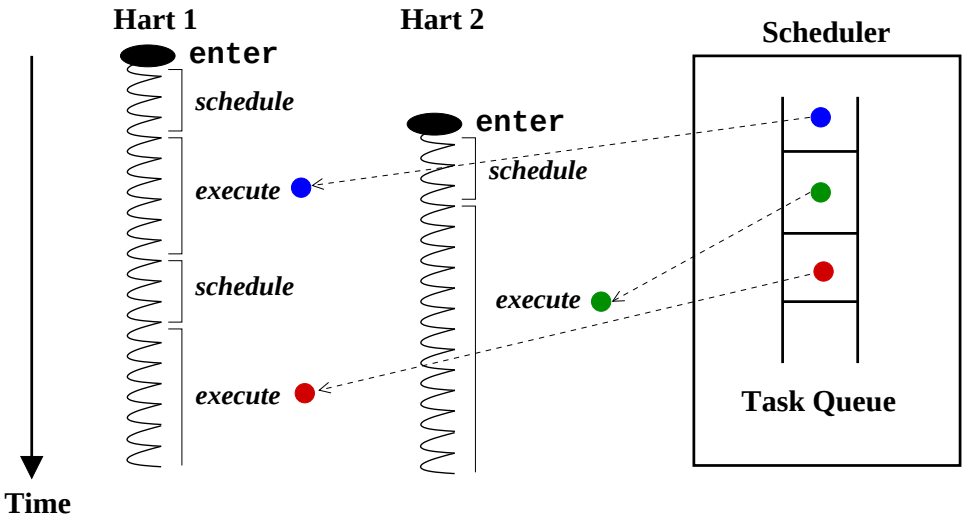


Figure 4-3: An example scheduler using its task queue to manage two of its harts from independent `enter` callbacks.

4.2 Update Callbacks

Update callbacks are used when a scheduler wants to communicate information to another scheduler. An update callback uses the hart it was invoked with to update its scheduler's state appropriately, then returns to effectively send the hart back to its original scheduler.

Below, we describe the four update callbacks defined by Lithe.

Register The `register` callback notifies its scheduler about a new child scheduler, which has taken over the current hart to manage the task last scheduled on that hart. The scheduler can associate the child with the task managed by the child, so that it can later decide how to allocate harts to the child based on the importance of its task.

Unregister The `unregister` callback notifies its scheduler about the completion of a child scheduler. The scheduler uses this callback to clean up any state it may have associated with the child. A child may unregister with a different hart than the one it used to register itself, but it always registers and unregisters using the same parent context (and stack).

Request The `request` callback notifies its scheduler about how many additional harts a child scheduler wants. The same child can request harts multiple times between the time it registers and unregisters. The scheduler can use this callback to increment the child's unfulfilled request count, and possibly request additional harts from its own parent. The scheduler is not obligated to satisfy all of the requests, but it can use the information to help guide its scheduling decisions.

Unblock The `unblock` callback notifies its scheduler that a blocked context is now runnable. Similar to the `block` callback, `unblock` is invoked from a task. Unlike `block`, the invoking task, along with the hart, may be managed by a different scheduler if there is cross-component synchronization. The scheduler merely uses the callback to mark the context as runnable, rather than hijack the hart to resume the context. It can then choose to resume the context the next time it makes a scheduling decision for any one of its hart.

Chapter 5

Lithe Runtime

This chapter describes the Lithe runtime system, which facilitates hart sharing and context switching. The runtime consists of a small amount of user-level bookkeeping and transitioning code to maximize modularity between parallel software components. It keeps track of the hierarchy of schedulers, current scheduler for each hart, and the standard callbacks for each scheduler. It also provides the mechanism to transfer a hart atomically between different schedulers and/or contexts.

5.1 Runtime Interface

The Lithe runtime provides functions to operate on the hart and context primitives, as listed in Table 5.1. The first set of functions enable schedulers to communicate with each other to share harts. The second set of functions can be used to start, pause, and resume contexts to use and share harts at a fine granularity. Below, we describe each function in detail.

5.1.1 Functions for Sharing Harts

This section describes the runtime functions used by schedulers to communicate with their parent and child schedulers. The runtime is responsible for connecting a scheduler to its parent or child by invoking the appropriate callback.

Building the Scheduler Hierarchy A scheduler "registers" itself when it is instantiated from within the current task to manage the rest of the task's computation. By calling `lithe_sched_register`, the scheduler effectively establishes itself as the current scheduler's child and takes control of the hart.

The lithe_sched_register function performs a few basic steps to update the internal runtime state and inform the current scheduler. First, the function assigns the scheduler a unique handle of type `lithe_sched_t`. It then inserts the scheduler into the scheduler

Lithe Runtime Interface	Scheduler Callback Interface
lithe_sched_register(callbacks)	register
lithe_sched_unregister()	unregister
lithe_sched_request(nharts)	request
lithe_sched_enter(child)	enter
lithe_sched_yield()	yield
lithe_ctx_init(ctx, stack)	N/A
lithe_ctx_fini(ctx)	N/A
lithe_ctx_setcls(cls)	N/A
lithe_ctx_getcls()	N/A
lithe_ctx_run(ctx, fn)	N/A
lithe_ctx_pause(fn)	N/A
lithe_ctx_resume(ctx)	N/A
lithe_ctx_block(ctx)	block
lithe_ctx_unblock(ctx)	unblock

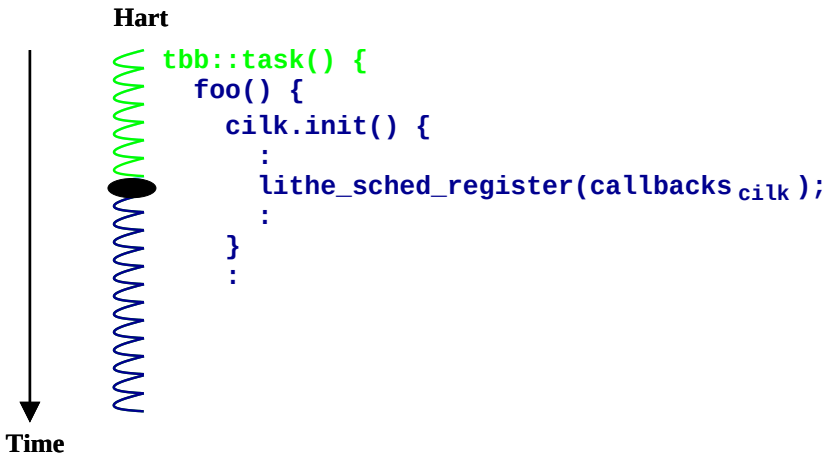
Table 5.1: The Lithe runtime functions and their corresponding scheduler callbacks. Note that the function, `fn`, passed to `lithe_ctx_pause` takes the current context as an argument.

hierarchy, and records the scheduler's callbacks. Next, it invokes the current scheduler's `register` callback, identifying the new scheduler by the opaque `lithe_sched_t` handle. Finally, it updates the current scheduler of the hart to be the new scheduler, effectively giving the new scheduler control of the hart upon returning.

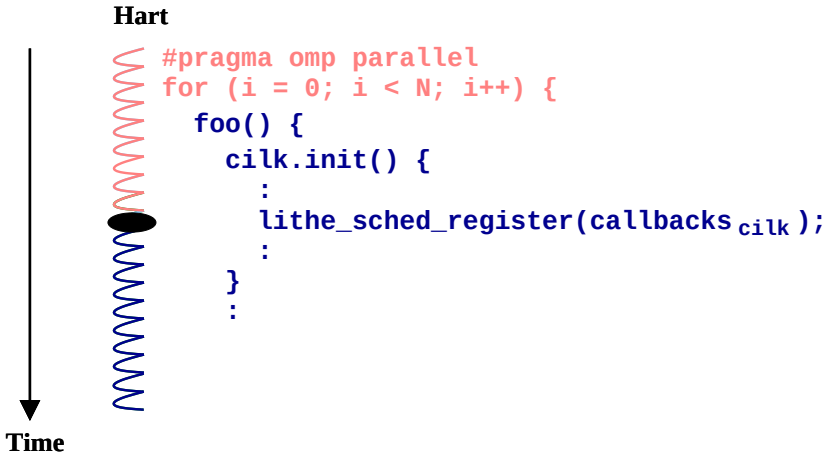
Figure 5-1(a) shows an example of using `lithe_sched_register`. A TBB task invokes a function, `foo`, which is written in Cilk. `foo` begins by instantiating a Cilk scheduler, which in turn registers itself to obtain control of the current hart from the TBB scheduler. Figure 5-1(b) shows how the Lithe runtime enables code reuse by insulating a scheduler from knowing its parent scheduling environment. The same Cilk implementation of `foo` can be reused within an OpenMP loop, since the Cilk scheduler does not care whether it is communicating with TBB or OpenMP as its parent.

Once a child scheduler completes its computation and no longer needs to manage harts, it calls `lithe_sched_unregister`. First, this runtime function waits until the unregistering scheduler has finished yielding all of its remaining harts to its parent, then invokes the parent scheduler's `unregister` callback. This is to prevent straggling harts from remaining under a child scheduler's control once the child has unregistered. Next, the function removes the unregistering scheduler from the hierarchy. Finally, it reverts the current scheduler of the hart to be the parent scheduler, effectively giving control of the hart back to the parent upon returning.

Requesting Harts To request additional harts, a scheduler invokes `lithe_sched_request`, passing the number of harts desired. This function relays the request directly to the parent scheduler by invoking the parent's `request` callback.



(a)



(b)

Figure 5-1: (a) `foo` is written in Cilk, and begins by instantiating a Cilk scheduler, which registers itself to obtain control of the current hart from the TBB scheduler. (b) The same `foo` function can be reused in an OpenMP loop, since the Cilk scheduler is insulated from its parent scheduling environment, and does not care whether it is communicating with TBB or OpenMP as its parent.

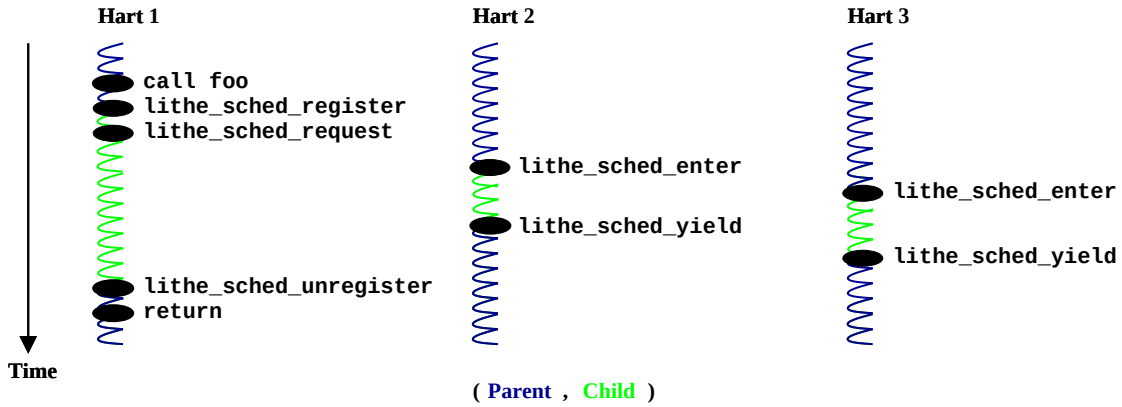


Figure 5-2: An example of how multiple harts might flow between a parent and child scheduler.

Allocation of Additional Harts A parent scheduler allocates a hart to a child scheduler by invoking the `lithe_sched_enter` function, and passing the child as an argument. This function updates the current scheduler to be the child, then invokes the child's `enter` callback using the transition context. This transfers the control of the current hart from a parent to the designated child in a continuation-passing style.

Whenever a scheduler is done with a hart, it can invoke `lithe_sched_yield`. This function updates the current scheduler from the child to the parent, then invokes the parent's `yield` callback using the transition context. This transfers the control of the current hart from a child back to its parent in a continuation-passing style.

Putting it All Together Figure 5-2 shows an example of how a parent and child scheduler use Lithe to cooperatively share harts. A parent component calls child component'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 invoking component, 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 `lithe_sched_enter` on each of these harts. Each `lithe_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 by invoking `lithe_sched_yield`. It will also unregister itself with its parent scheduler, then return from `foo`.

5.1.2 Functions for Manipulating Contexts

This section describes the runtime functions used to start, pause, and resume contexts. These functions enable portability by shielding schedulers from knowing the platform-specific details of how to save and set machine state. They also enable atomic context switching within and across parallel components.

Executing with Contexts A scheduler sets up a context by calling the `lithe_ctx_init` function and passing a stack for the context to use. The Lithe runtime identifies each context by an opaque handle of type `lithe_ctx_t`. A scheduler starts the context by invoking `lithe_ctx_run`, and passing it a function that should be invoked by the context. Note that after the computation using the context has completed, the context can be reused by subsequent calls to `lithe_ctx_run`. A scheduler cleans up a context by calling the runtime function `lithe_ctx_fini`.

Context-Local State A scheduler can associate some data with each context by calling the `lithe_ctx_setcls` function, and passing a pointer to the data. A scheduler can access the context-local state for the current context by invoking `lithe_ctx_getcls`.

Context Switching The current context can be paused using the `lithe_ctx_pause` function. The `lithe_ctx_pause` function is similar to the call-with-current-continuation (`call/cc`) control operator [Wil]. When `lithe_ctx_pause` is invoked, it stores the current continuation in the current context, switches to the transition context, and calls the function passed to `lithe_ctx_pause` with the current context as an argument. Switching to a different context before invoking the function passed to `lithe_ctx_pause` allows a programmer to manipulate a context without worrying about running on said context's stack. A paused context can be resumed using `lithe_ctx_resume`.

Figure 5-3 shows an example of how to transition from one context to another in a continuation-passing style using the runtime functions. A scheduler receives a new hart when its `enter` callback is invoked with the transition context. It initializes a new context with `lithe_ctx_init`, and uses the context to run some tasks by invoking `lithe_ctx_pause` with its scheduling function. Once it is done executing the tasks, it cleans up the current context using the transition context by invoking `lithe_ctx_pause` with the clean up function. `lithe_ctx_pause` switches the contexts atomically, thus getting the scheduler off the previous stack cleanly, while providing a stack for the scheduler to use before it can decide what to do next. From there, the scheduler can resume another context that had been previously suspended by calling `lithe_ctx_resume`. Without the transition context, the scheduler would need complex cross-context synchronization to coordinate the stack switching [FR02, LTMJ07, FRR08].

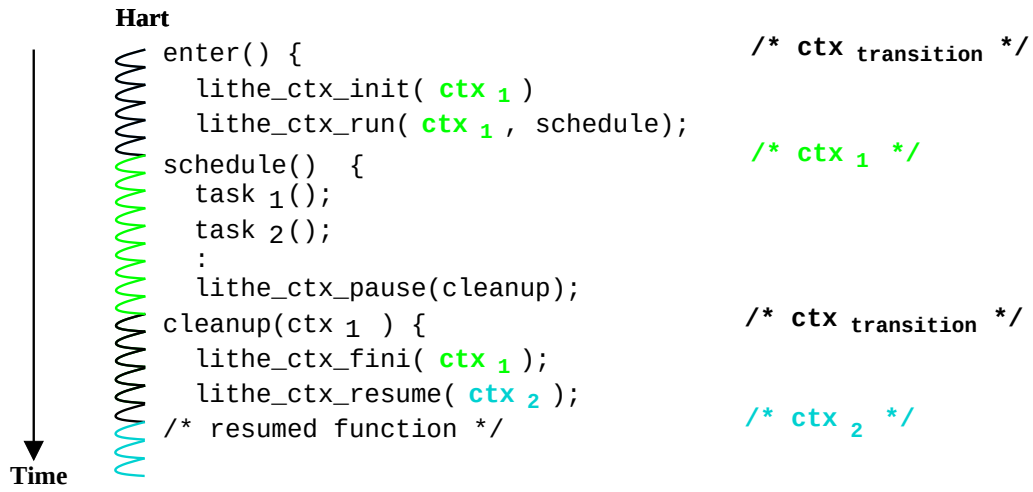


Figure 5-3: An example of transitioning from one context to another in a continuation-passing style using the Lithe runtime functions. A scheduler's `enter` callback starts on the transition context, launches a new context to run tasks, cleans up that context using the transition context, then resumes another context.

Communicating with Schedulers To enable efficient fine-grained sharing of harts, a pending synchronization or I/O operation can block its context, and return the hart back to the current scheduler to do useful work in the meantime. Lithe provides the same mechanism for any such long-running operation to communicate with its scheduler. This thesis focuses on the discussion of the interaction between synchronization and scheduling. See [PHA10] for a similar discussion of the interaction between I/O and scheduling.

A task that is waiting to complete a synchronization operation can suspend itself using `lithe_ctx_pause`. While on the transition context, the corresponding synchronization library can record the suspended context, and return control of the current hart back to its scheduler by invoking `lithe_ctx_block`. This runtime function invokes the `block` callback of the current scheduler, also using the transition context. Note that all transfer callbacks (as well as the function passed into `lithe_ctx_pause`) start executing at the top of the transition stack. The transition stack is rewound upon each callback, so that a long sequence of callbacks does not overflow the transition stack.

At a later point in time, another task involved in the rendezvous will invoke the synchronization library to complete the synchronization operation. This can occur on a different hart, and even on a hart managed by a different scheduler when different parallel components are synchronizing with one another. The synchronization library, which would have kept track of the suspended context, can inform the appropriate scheduler that the context is now runnable by invoking `lithe_ctx_unblock`. Since the Lithe runtime keeps track of each context's associated scheduler, the synchronization library only has to name the unblocked context, but not its scheduler. This enables the synchronization library to be interoperable with a wide range of schedulers. `lithe_ctx_unblock`, in turn, invokes the

```
1  void Mutex::lock() {
2    __lock();
3    if (m_locked)
4      lithe_ctx_pause(savectx);
5      __lock();
6    m_locked = true;
7    __unlock();
8  }
9
10 void Mutex::unlock() {
11   __lock();
12   m_locked = false;
13   lithe_ctx_t *ctx;
14   (m_waitqueue->pop(&ctx))
15   __unlock();
16   if (ctx)
17     lithe_ctx_unblock(ctx);
18 }
19
20 void Mutex::savectx(lithe_ctx_t *ctx) {
21   m_waitqueue->push(ctx);
22   __unlock();
23   lithe_sched_block(ctx);
24 }
```

Figure 5-4: Pseudocode example of a Lithe-compliant mutex lock.

original scheduler's `unblock` callback. Since `unblock` is an update callback, the scheduler will simply borrow the hart to mark the context as runnable, rather than hijacking the hart to resume the context.

Figure 5-5 illustrates how a mutex synchronization example works with Lithe, and Figure 5-4 shows the corresponding pseudocode of the mutex library implementation. The mutex library has an internal boolean `m_locked` that keeps track of whether the mutex is currently locked, and an internal queue `m_waitqueue` that keeps track of all the contexts waiting to acquire the mutex. `__lock` and `__unlock` acquires and releases an internal spinlock (not shown) that prevents race conditions by ensuring updates to `m_locked` and `m_waitqueue` happen atomically with respect to each other.

The functions `foo1` and `foo2` are contending for the same mutex, and `foo1` acquires the mutex first (line 6). The `lock` function cannot allow `foo2` to proceed when the mutex is already acquired (line 3), so it pauses the context of `foo2` (line 4), saves the context (line 20), and informs the context's scheduler that it is blocked (line 23). The scheduler decides to run `bar` while waiting for the mutex to be free.

When `foo1` releases the mutex (line 12), the library unblocks one of the contexts waiting

Figure 5-5: An example of how mutual exclusion works with Lithe. `foo1` and `foo2` are contending for the same mutex. `foo1` acquires the mutex first, so `foo2` suspends itself and gives the hart back to the scheduler. The scheduler uses the hart to execute `bar`, then resumes the unblocked context of `foo2`.

on the mutex, if any (lines 14, 16-17). In this example, it unblocks the context of `foo2`. Upon the completion of `bar` on the other hart, the scheduler decides to resume the unblocked context, which will recommence at line 5 and acquire the mutex at line 6.

5.2 Runtime Implementation

We have implemented a complete Lithe runtime for x86 64-bit Linux. The runtime is extremely small. It is only approximately 2,300 lines of C, C++, and x86 assembly. It provides the common glue code to transition between schedulers and contexts, just as a

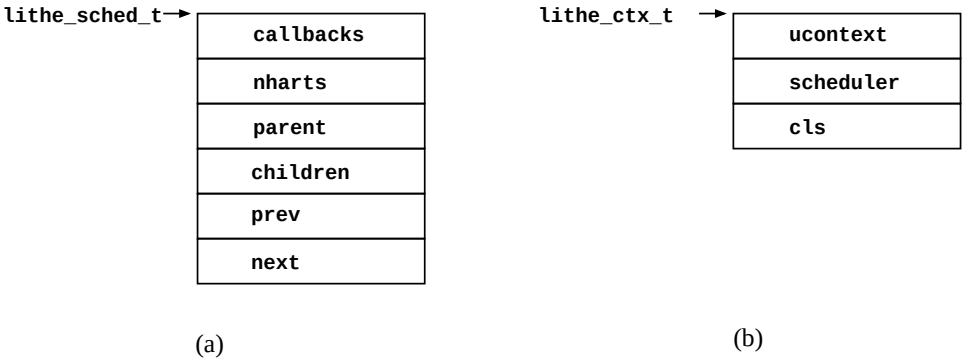


Figure 5-6: The internal Lithe scheduler and context objects.

low-level language runtime (e.g. C) provides the glue to transition between functions.

Below, we describe how the Lithe runtime keeps track of the schedulers within an application. We also describe the POSIX mechanisms leveraged by the Lithe runtime to implement its primitives.

Schedulers For each scheduler, the Lithe runtime keeps state about its callbacks, harts, parent, and children schedulers. The internal Lithe scheduler object is shown in Figure 5-6(a), and the external `lithe_sched_t` handle to a scheduler is simply a pointer to this object. The `callbacks` field contains the location of the function pointer table provided by the scheduler upon registering. The `nharts` field keeps track of how many harts the scheduler is managing at any time. The `parent` field is a pointer to the scheduler's parent. The `children` field is a pointer to one of the scheduler's children, if any. The `prev` and `next` fields form a doubly linked list connecting all schedulers with the same parent.

The Lithe runtime also provides a *base scheduler* for the application. The sole purpose of this scheduler is to obtain harts for the application to use from the OS, and serve as the parent of the first scheduler to register with the Lithe runtime, 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 `lithe_sched_enter`.

Contexts As shown in Figure 5-6(b), the internal Lithe context object contains the hart's machine state, a pointer to its owning 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 does not incur any kernel crossings.

The Lithe runtime allocates the transition context for each hart, along with the associated transition stacks. To switch to a transition context, the runtime uses assembly code to change the stack atomically and pass any arguments through the registers, as not to touch the original stack once the corresponding runtime function has been invoked.

Harts The current implementation of Lithe interfaces with a user-level implementation of harts. Every hart is represented using a POSIX thread (pthread), since Linux maps each pthread to its own kernel thread. The affinity extension of pthreads is supported by Linux, and is used to pin each pthread to a unique core.

Chapter 6

Interoperable Parallel Libraries

In this chapter, we describe the wide range of interoperable task scheduling, synchronization, and domain-specific libraries that we have built and ported to demonstrate the utility and versatility of Lithe. 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). These modifications do not change the interface of the libraries. Hence, we can provide "bolt-on" performance composability without modifying a single line of high-level application code, and by simply relinking the applications with the new library binaries.

6.1 Task Scheduling Libraries

We have ported two commonly used task scheduling libraries that support different programming models and features: Threading Building Blocks (TBB) [Rei07] and OpenMP [CMD+01]. TBB is a C++ library that supports a hierarchical fork-join task model, and provides high-level parallel algorithm templates and scalable memory allocation. OpenMP is a C, C++, and Fortran language extension that supports a master/slave model, and provides simple loop annotations and cross-platform portability.

Porting Linux implementations of TBB and OpenMP to use Lithe involves a small, straightforward change, as illustrated in Figure 6-1. Both the TBB and OpenMP schedulers assume that they have the entire machine to themselves, and use pthreads as proxies for physical processing resources (since there is a one-to-one correspondence between a pthread and a Linux kernel thread). Each scheduler creates N pthreads up front on an N-core machine, then maps its respective tasks onto these pthreads. With Lithe, each scheduler no longer creates pthreads, but instead requests harts and only uses the harts that it is granted.

We have also built a new Single-Program Multiple-Data (SPMD) library [Kar87] using Lithe. The SPMD library supports a minimalistic programming model that enables its users to simultaneously spawn N independent threads of control. Below, we begin by explaining how to implement this SPMD library with pthreads and with harts, since it is the easiest

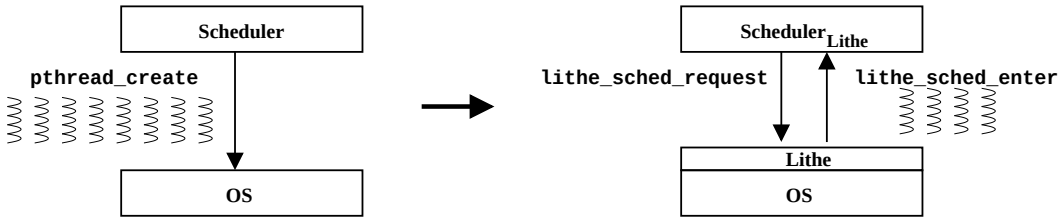


Figure 6-1: Porting a scheduler to use Lithe replaces arbitrary pthread creation with explicit requests for harts.

to understand. We then explain the details of how to port TBB and OpenMP to use Lithe.

6.1.1 Single-Program Multiple-Data (SPMD)

The SPMD library supports the Single-Program Multiple-Data programming model [Kar87]. It 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 then figures out what to do using its id, obtained through `spmd_tid`. `spmd_spawn` returns after all its spawned tasks have completed. The baseline implementation simply uses a unique pthread to implement each of the SPMD tasks, thus leaving all of the scheduling decisions to the OS. The Lithe implementation, on the other hand, maps each of the tasks consecutively onto its next available hart.

The crux of the Lithe implementation is shown as pseudocode in Figure 6-2. First, `spmd_spawn` instantiates a SPMD scheduler, and registers that scheduler (including its standard callbacks) with the Lithe runtime using `lithe_sched_register` (lines 2-3). The scheduler requests additional harts using `sched_request` (line 4). The scheduler then pauses the initial context (line 5), saves it (line 11), then invokes the main scheduling loop (line 12). Any additional harts granted by its parent will join the SPMD scheduler through its `enter` callback (line 15), which also invokes the main scheduling loop (line 16).

The scheduler uses its main scheduling loop to decide what to do with each of its harts. First, the scheduler tries to resume any paused context that is now runnable (lines 20-21). Otherwise, the scheduler tries to satisfy requests from its children (lines 22-23). Next, it will try to create a new context to start a new SPMD task, by invoking the runtime function `lithe_ctx_start` with the new context and the `start` function (lines 24-26). `start` will continually run the next SPMD task using that context (with the correct task id environment) until they are all completed (lines 34-36). The context is then cleaned up (lines 37, 41), and the hart returns to the scheduling loop in search for more work (line 42). When all of the work is done, one of the harts is used to resume the initial context (lines 27-28) to unregister the scheduler (line 6). The remaining harts are yielded back to the parent scheduler (lines 29-30). Finally, the `spmd_spawn` function cleans up the scheduler before returning back to its caller (line 7).

```
1  void spmd_spawn(int N, void (*func)(void*), void *arg) {
2    SpmdSched *sched = new SpmdSched(N, func, arg);
3    lithe_sched_register(sched);
4    lithe_sched_request(N-1);
5    lithe_ctx_pause(spawn);
6    lithe_sched_unregister();
7    delete sched;
8  }
9
10 void SpmdSched::spawn(ctx0) {
11   /* save ctx0 */
12   schedule();
13 }
14
15 void SpmdSched::enter() {
16   schedule();
17 }
18
19 void SpmdSched::schedule() {
20   if (/* unblocked paused contexts */)
21     lithe_ctx_resume(/* next unblocked context */);
22   else if (/* requests from children */)
23     lithe_sched_enter(/* next child scheduler */);
24   else if (/* unstarted tasks */)
25     ctx = new SpmdCtx();
26     lithe_ctx_run(ctx, start);
27   else if (/* done and not yet unregistered */)
28     lithe_ctx_resume(ctx0);
29   else
30     lithe_sched_yield();
31 }
32
33 void SpmdSched::start() {
34   while (/* unstarted tasks */)
35     /* set up tid environment */
36     func(arg);
37   lithe_ctx_pause(cleanup);
38 }
39
40 void SpmdSched::cleanup(ctx) {
41   delete ctx;
42   schedule();
43 }
```

Figure 6-2: SPMD scheduler pseudocode example.

We do not show the implementation for the scheduler's `register`, `unregister`, `request`, `yield`, `block`, and `unblock` standard callbacks, but they essentially update the various parts of the scheduler's state that are checked in the main scheduling loop (lines 20, 22, 24, 27).

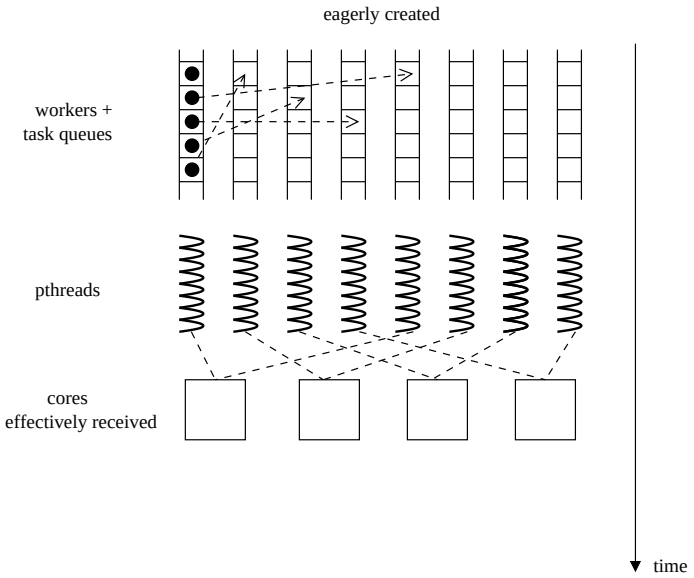
Our experience shows that the SPMD library can serve a stepping stone for third-party developers to port their codes to use Lithe, since its programming model closely resembles how pthreads are used by schedulers to represent processing resources. A task library can be built on top of the SPMD library, using SPMD tasks instead of pthreads, as an indirect way to leverage harts. The small, no-frills code base of the SPMD library can also serve as a simple example for understanding how to manage harts.

6.1.2 Threading Building Blocks (TBB)

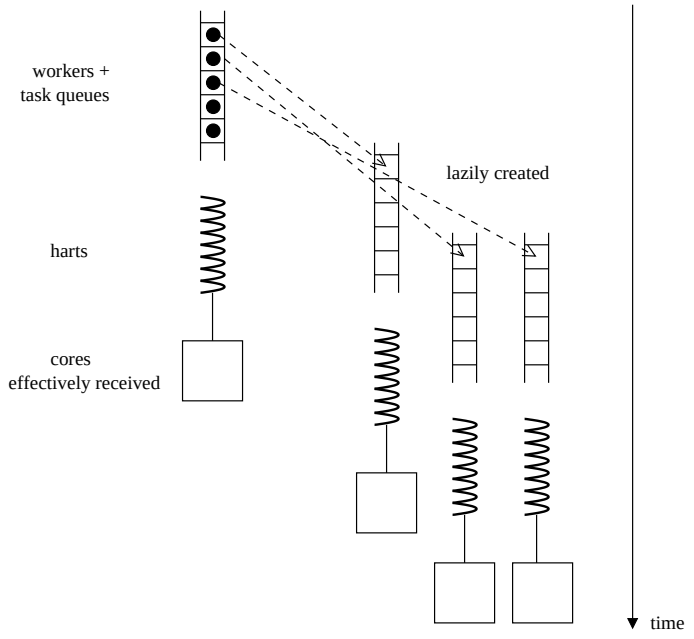
Intel's Threading Building Blocks (TBB) library [Rei07] provides a high-level task abstraction to help programmers achieve performance and scalability by simply exposing the parallelism without having to manually manage the tasks' parallel execution. Instead, a TBB scheduler uses a dynamic work-stealing [BJL+95] approach to automatically map the tasks onto the cores.

The original open-source Linux implementation of the TBB library (version 2.1 update 2) uses pthreads as its resource abstraction. The TBB library creates a fixed-size global pool of pthreads upon its initialization, and creates a worker for each of these pthreads. As shown in Figure 6-3(a), 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 a 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. This allows tasks to be dynamically load-balanced across all the available pthreads. The original TBB implementation already tries to prevent resource oversubscription by limiting all TBB schedulers to share the global fixed-size pool of pthreads. However, this still does not prevent TBB schedulers from interfering with other non-TBB parallel libraries. The original implementation also indirectly curbs oversubscription by putting worker threads to sleep whenever they cannot find work to steal.

In the ported implementation, each TBB scheduler manages harts rather than pthreads (Figure 6-3(b)). Since a TBB scheduler does not know how many harts it can use up front, it lazily creates a worker and a context for each hart only when the hart arrives 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 their corresponding number of harts requested. It uses a simple round-robin allocation policy to grant harts to its children.



(a)



(b)

Figure 6-3: (a) The original TBB implementation creates as many workers as the number of cores it thinks it has, without regard to other libraries within the application. (b) The Lithe TBB implementation lazily creates workers as it receives more harts over time.

6.1.3 OpenMP (OMP)

OpenMP [CMD+01] 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 worker threads, each with a unique thread id. The number of worker threads 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 worker threads, the programmer can also manually assign work to each of these worker threads using SPMD-styled codes.

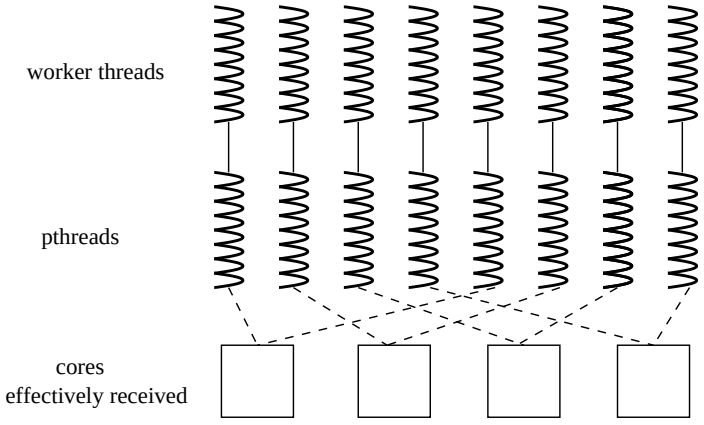
We ported the GNU Compiler Collection's (GCC) open-source Linux implementation of the OpenMP runtime library (libgomp from the GCC 4.4 trunk which implements version 3.0 of the OpenMP standard). As with the original TBB library implementation, the original OpenMP library uses pthreads as its resource abstraction. An implicit scheduler for every team maps each of its worker threads onto a different pthread (Figure 6-4(a)). For efficiency, the library keeps a global pool of the pthreads to reuse across multiple teams. Note that when multiple teams are active, a team scheduler will create additional pthreads rather than wait for pthreads from other teams to return to the global pool.

In the ported version, each team scheduler multiplexes its worker threads onto its available harts (Figure 6-4(b)). Similar to the simple SPMD library, a team scheduler lets each worker thread 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 worker threads 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.

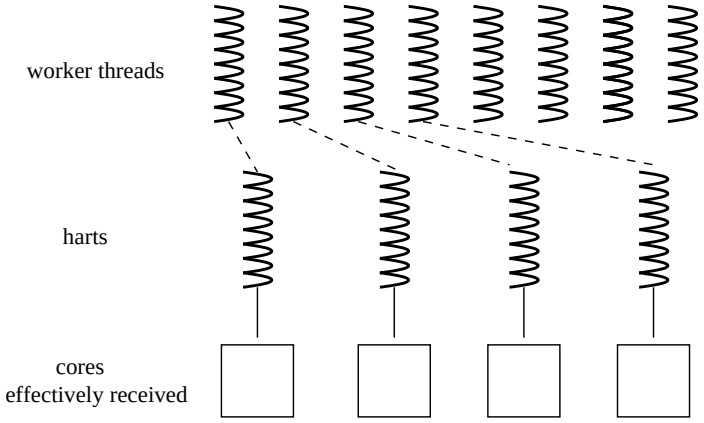
6.2 Synchronization Libraries

In this section, we discuss how synchronization interacts with scheduling, and how synchronization mechanisms work under Lithe's cooperative scheduling model. Synchronization operations dynamically coordinate different tasks to ensure a certain task ordering or to provide mutually exclusive access to shared data. A task can be blocked during the middle of its execution, waiting for the synchronization operation it has invoked to complete. This may entail waiting for other task(s) to reach the same synchronization rendezvous or to release exclusive access to shared data.

While a task is waiting, it can cooperatively give its hart back to its scheduler to do useful work in the interim. In current systems, the task only communicates with the global OS scheduler, as shown in Figure 6-5(a). The task can use `pthread_cond_wait` to give up its processing resource until the synchronization is complete. Alternatively,



(a)



(b)

Figure 6-4: (a) The original OpenMP implementation maps each worker thread to a pthread, effectively leaving all scheduling decisions to the OS. (b) The Lithe OpenMP implementation maps each worker thread to a hart in a more efficient manner, executing each worker thread to completion to eliminate unnecessary context switching and interference.

it can use `pthread_yield` if it does not care about being resumed before it is runnable again. With Lithe, there can be multiple user-level schedulers active at the same time (Figure 6-5(b)). A blocked task can use `lithe_ctx_block` to give the hart back to the appropriate scheduler, without caring who the scheduler is or how it is implemented. While `pthread_cond_wait` requires a mutex to synchronize between the user-level task and the OS scheduler, `lithe_ctx_block` uses the transition context to obviate the need to pass a mutex between the task and its scheduler.

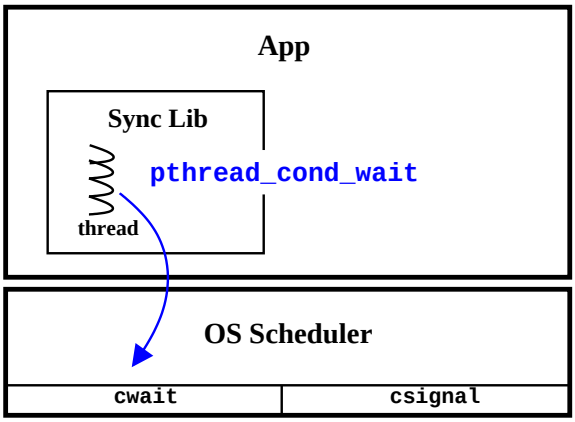
An alternative to giving back a hart is busy waiting. A task spins on its hart constantly until it has successfully synchronized. This wastes computing resources in hope of reacting more quickly to any changes in the synchronization condition. This tends to work well when synchronization is on the critical path, and there is nothing better to do in the meantime. On the other hand, this tends to hurt performance when the resources can be used to do more useful work. The latter often occurs when the system is oversubscribed, and there are many more runnable (and synchronizing) tasks than there are processing resources. Worse yet, in a purely cooperative system like Lithe, busy waiting can potentially deadlock the system when performing producer-consumer synchronization. Harts could be hogged indefinitely by consumer tasks, preventing producer tasks from ever running. Busy-waiting mutual exclusion will not deadlock as long as the task that has acquired exclusive access does not suspend itself and yield its hart before it has released its exclusive privilege.

A hybrid approach, competitive spin-sleep, aims to achieve the benefits of both cooperative yielding and busy waiting. A spin-sleep approach waits for a small amount of time, then gives up its processing resource. This avoids the overhead of context switching in the optimal case where the tasks are fairly in sync, while enabling the scheduler to dynamically load balance when the tasks are skewed.

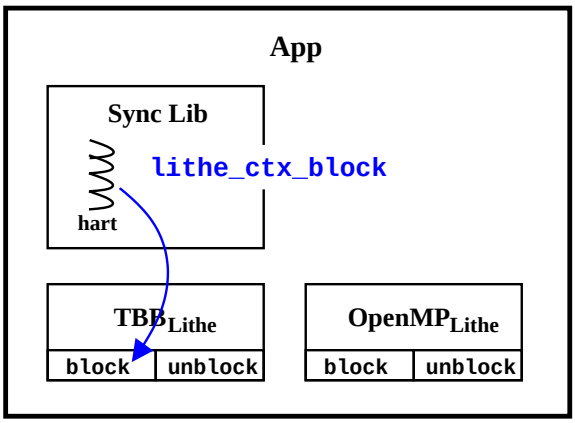
We have already described how to implement a mutex using Lithe when explaining the Lithe context manipulation functions in Section 5.1.2. Below, we describe a Lithe-compliant implementation of a barrier in contrast to a comparable busy-waiting implementation.

6.2.1 Barriers

We built two different user-level implementations of a simple barrier: one using Lithe to competitively spin-sleep, and one that busy waits. Both implementations share the bulk of their code. A barrier is initialized with the number of tasks that are synchronizing with each other. Upon reaching the barrier, each task increments a shared 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 respective context, and yield the control of their respective 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 will then decide when to resume each of these



(a)



(b)

Figure 6-5: (a) In current systems, a waiting task gives its thread back to the OS scheduler using `pthread_cond_wait`. (b) With Lithe, a waiting task gives its hart back to its user-level scheduler using `lithe_ctx_block`.

paused contexts.

6.3 Domain-Specific Libraries

In this section, we give an example of how a domain-specific library uses Lithe. Lithe's hart primitive eliminates OS multiplexing within an application, and gives user-level codes much more control over their own parallel execution. Specialized libraries can thus leverage both their domain-specific knowledge and their uninterrupted control over their harts to deliver energy-efficient performance.

6.3.1 Fastest Fourier Transform in the West (FFTW)

The FFTW library computes discrete Fourier transforms in one or more dimensions of arbitrary input size [FJ05]. FFTW selects from many different FFT algorithms and implementation strategies to run the best combination for a given platform. It also uses divide-and-conquer techniques to take advantage of the memory hierarchy.

The original implementation (version 3.2.2) decomposes each FFT subproblem into a global array of tasks, and maps each task to a worker thread. The library uses a pthread worker thread pool very similar to the one implemented by the original OpenMP implementation described earlier. The library creates an initial thread pool to reuse across different FFTs, and creates additional pthreads rather than wait for pthreads to return to the global pool when multiple FFTs are invoked in parallel.

The version ported to Lithe retains the same global array of decomposed tasks, and uses each available hart to repeatedly execute the next task in the array (until there are no more tasks). This implementation also keeps a global pool of contexts to reuse across different FFTs for efficiency.

Chapter 7

Evaluation

In this chapter, we present results from experiments on commodity hardware that show the effectiveness of the Lithe substrate. First, we present numbers for the individual task scheduling libraries, which will show that we can implement existing abstractions with no measurable overhead. Next, we use a simple microbenchmark to take a deeper look into the interaction between synchronization and scheduling. Finally, we present two application case studies to illustrate how we improve the current state-of-the-art in parallel library composition.

Unless otherwise specified, the hardware platform used was a quad-socket AMD Barcelona with four 2.3 GHz Opteron-2356 cores per socket (16 cores total). Each core has private access to a 2-way associative 64KB L1 cache, and a 16-way associative 512KB L2 cache. Each socket has a 32-way associative 2MB outer-level victim cache shared by all 4 cores. The OS platform used was the 2.6.26 64-bit Linux kernel, which includes the Completely Fair Scheduler [WTKW08] and the NPTL fast pthread library.

7.1 Baseline Scheduling Library Performance

We begin by examining the impact of using Lithe when there is only one parallel task scheduling library used in the entire application, as many parallel codes today are built that way. In this section, we will show that no performance overhead is introduced for this baseline case.

Performance Baseline We evaluated the baseline performance of TBB and OpenMP by comparing the performance of their original and ported implementations. We chose to use small microbenchmarks, in which the computations are short and scheduling constitutes a significant portion of the total execution time. This is to ensure that any performance penalty potentially introduced by Lithe is not obscured by the computation. Since no Lithe hart sharing runtime functions are invoked without the presence of additional schedulers,

	Tree Sum	Preorder	Fibonacci
Original TBB	1.44	1.61	1.65
Lithe TBB	1.23	1.40	1.45

Table 7.1: Comparison of the standalone performance for the original and Lithe-compliant implementations of TBB, as measured in average runtime in seconds for three different microbenchmarks.

the measured performance differences reflect the cost of using Lithe contexts and the cost of adapting to additional harts (as opposed to using a predetermined number of pthreads).

To evaluate TBB, we randomly chose a few of the microbenchmarks included with the TBB distribution: a parallel tree summing algorithm, a parallel preorder tree traversal, and a parallel implementation of Fibonacci. These microbenchmarks were designed to highlight the performance and scalability of TBB. Table 7.1 shows the average execution time, in seconds, across ten runs of each microbenchmark for both the original and ported implementations. Across all three microbenchmarks, the ported TBB actually performs marginally better than the original. This is probably due to the greedy nature of the harts implementation. The original TBB implementation cooperatively puts a pthread to sleep after a few failed work stealing attempts. The hart implementation used by the Lithe port greedily keeps any hart returned by the Lithe base scheduler in a busy waiting loop until additional harts are requested or until the application terminates. The Lithe port, therefore, can react faster to the dynamically changing amount of parallelism in each of these microbenchmarks.

To evaluate OpenMP, we randomly chose a few of the NAS parallel benchmarks [JMF99]: conjugate gradient (CG), fast Fourier transform (FT), and integer sort (IS), from NPB version 3.3; problem size W. Table 7.2 shows the average execution times across ten runs of each benchmark for both the original and ported implementations. The ported OpenMP is also competitive with, if not better than, the original implementation. Unlike the ported TBB, the ported OpenMP cannot exploit the hart implementation's competitive advantage of adapting more quickly to changing parallelism, since the OpenMP programming model keeps the amount of parallelism constant across each microbenchmark. The ported OpenMP does, however, have the advantage of being able to load balance its work across its harts compared with the original OpenMP. The original OpenMP's execution of a microbenchmark cannot complete until the OS has scheduled each of the worker thread's underlying pthreads. The ported OpenMP scheduler, in contrast, can dynamically reshuffle worker threads to execute on its available pthreads if its other pthreads are not being scheduled fast enough by the OS.

Porting Effort We also wanted to show that porting libraries to use Lithe involves very little effort. Even if a parallel library is only used in standalone situations today, there is very little cost to making it much more robust and useful. We quantify the porting

	CG	FT	IS
Original OpenMP	0.22	0.19	0.54
Lithe OpenMP	0.21	0.13	0.53

Table 7.2: Comparison of the standalone performance for the original and Lithe-compliant implementations of OpenMP, as measured in average runtime in seconds for three different microbenchmarks.

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

Table 7.3: Approximate lines of code required for each of the Lithe-compliant task scheduling libraries.

effort based on the code changes between the original and the port. Table 7.3 shows the approximate number of lines that needed to be added, removed, or modified when porting TBB and OpenMP to use Lithe. The affected number of lines are extremely small, even just compared to the relevant lines of code (which only refer to the portion of each runtime that performs actual scheduling, as opposed to, for example, scalable memory allocation in TBB). We can also quantify the porting effort in terms of our own anecdotal experience. It took one "graduate-student-week" to port TBB to Lithe, and about two "graduate-student-weeks" to port OpenMP to Lithe.

7.2 Barrier Synchronization Case Study

In this section, we study how synchronization and scheduling interact in the face of parallel composition by analyzing a simple barrier microbenchmark. Barriers are commonly used by programmers to deploy a bulk-synchronous programming style that is often easier to reason about [NY09]. Barriers are also commonly used by library runtimes to implement

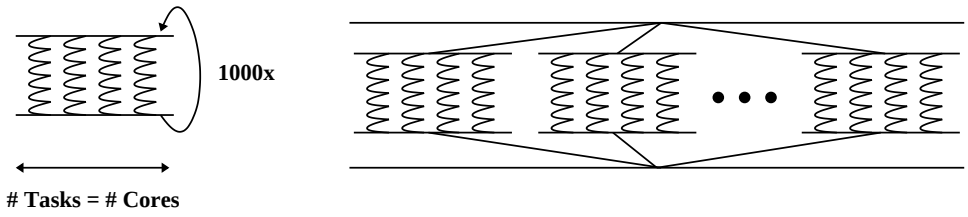


Figure 7-1: (a) The baseline barrier microbenchmark of N SPMD tasks synchronizing with each other 1000 times at back-to-back barrier rendezvous. (b) Launching multiple baseline barrier microbenchmarks in parallel to examine the behavior of different barrier implementations in the face of parallel composition.

Barrier Impl	SPMD Scheduler Impl
Pthread	Baseline, Pinned Tasks Baseline, Unpinned Tasks
Lithe	Lithe
Spin	Baseline, Pinned Tasks Baseline, Unpinned Tasks

Table 7.4: The different barrier implementation and their compatible scheduler implementations. Refer back to Section 6.2.1 for details on the individual barrier implementations, and refer back to Section 6.1.1 for more details on the individual SPMD scheduler implementations. There are five total possible barrier/scheduler configurations evaluated in this thesis.

the join points in fork-join models.

As shown in Figure 7-1(a), 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). The number 16 was chosen to be equivalent to the number of cores on our experimental Barcelona machine, which reflects the typical usage of a barrier inside today's bulk-synchronous libraries. To evaluate how different barrier implementations behave in the face of parallel composition, we tried launching between 1 and 10 of the basic units of work in parallel, as shown in Figure 7-1(b).

We compared three barrier implementations, as shown in Table 7.4: (1) the Glibc implementation of a pthread barrier ("pthread"), which cooperatively yields pthreads back to the OS scheduler upon waiting; (2) our spin-sleep implementation ("Lithe"), which uses Lithe to cooperatively yield harts back to its user-level scheduler after not being able to synchronize within a short amount of time; and (3) our busy waiting implementation ("spin"), which circumvents any scheduler, and simply spins on each hart until it can proceed. Each barrier implementation is only compatible with a certain subset of SPMD scheduler implementations. (Recall from Section 6.1.1 that the baseline SPMD scheduler implementation maps each task to its own pthread, effectively letting the OS schedule the tasks.) The pthread barrier is hardcoded to communicate with the OS scheduler. The Lithe barrier can communicate with any Lithe-compliant scheduler. The spin barrier will only function correctly with a pre-emptive scheduler, so we pair it with the OS scheduler as well. Additionally, we experiment with two variants of the baseline SPMD scheduler implementation, one where each SPMD task is pinned to the core corresponding to its task ID, and the other where the tasks are unpinned.

Figure 7-2 shows the performance results for all possible combinations of barrier and scheduler implementations. Each line in the graph shows the performance of a unique barrier/scheduler configuration. The points along the x-axis correspond with different versions of the microbenchmark with differing units of work launched in parallel. The performance measured by the y-axis reflects the execution time normalized to the total number

of `barrier_wait` synchronization operations in the corresponding microbenchmark. This is designed to showcase how the performance of an individual barrier operation varies with its execution environment in the face of parallel composition.

We discuss the performance of different configurations in each execution environment by analyzing Figure 7-2 from left to right. First, we examine the scenario where only a single unit of work was launched (x-axis = 1). This is the simple environment in which barrier developers typically evaluate their implementations. Using the pthread barrier in conjunction with pinned tasks instead unpinned tasks achieved slightly worse performance, as pinned tasks cannot be executed elsewhere even if their corresponding cores are temporarily busy processing interrupts or daemons. As expected, using the busy waiting barrier with unpinned tasks performed better than the two pthread barrier configurations, because it avoided the overhead of entering into the OS scheduler and context switching. However, busy waiting with pinned tasks performed the worst. Not only did it suffer load imbalance from pinning, but it also wasted many of its OS scheduling timeslice allocations on busy waiting. The Lithe implementation performed by far the best. Most of the time, it can spin for only a short amount of time, just as the busy wait implementation. It can also reap the benefits of fine-grained resource sharing when its tasks are skewed, just as the pthread implementation, but at a much lower cost since it does not have to cross any kernel boundaries to communicate with its user-level scheduler.

Next, we examine when two units of work are launched in parallel (x-axis = 2). The pthread configurations improved because they can achieve better load balancing with more runnable tasks than cores in the machine. The busy waiting configurations slowed down by orders of magnitude because of destructive interference. Each of the SPMD tasks hogs its processing resource rather than letting the other tasks run. The Lithe implementation is less efficient executing two parallel units instead of one, because it can no longer get away with simply spinning most of the time. Nonetheless, the Lithe implementation remains the fastest by a wide margin, since yielding harts to a user-level scheduler is much cheaper than yielding pthreads to the OS scheduler.

Lastly, we examine the scalability of the different barrier implementations as more and more units of work are launched in parallel (x-axis > 2). 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 slower rate of improvement as the pinned pthread version. Since Lithe is a user-level implementation, it is still at the mercy of the OS scheduler to schedule its harts' underlying pthreads.

This simple microbenchmark shows that the performance of different barrier implementations can vary vastly depending on the execution environment. To build robust parallel software, the performance of synchronization mechanisms must not deteriorate in the face of parallel composition, even if their baseline performance is slightly better.

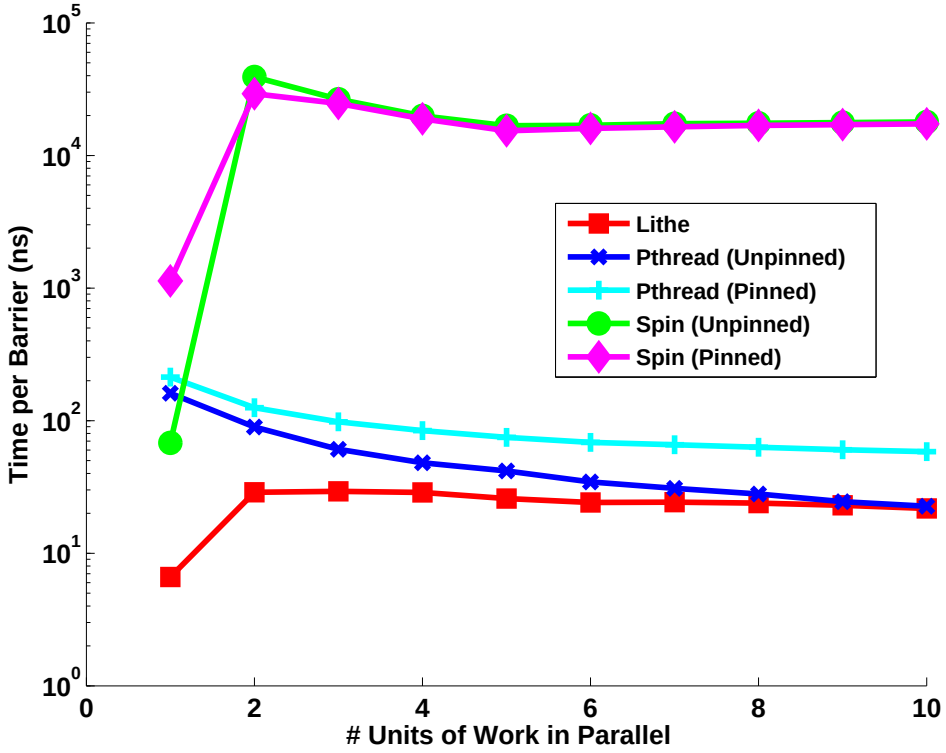


Figure 7-2: Performance of different barrier implementations under different parallel composition configurations in terms of runtime in nanoseconds.

7.3 Application Case Studies

In this section, we use two case studies to demonstrate that Lithe improves the performance of real world applications composed of multiple parallel libraries by simply relinking them with the new library binaries.

7.3.1 Sparse QR Factorization

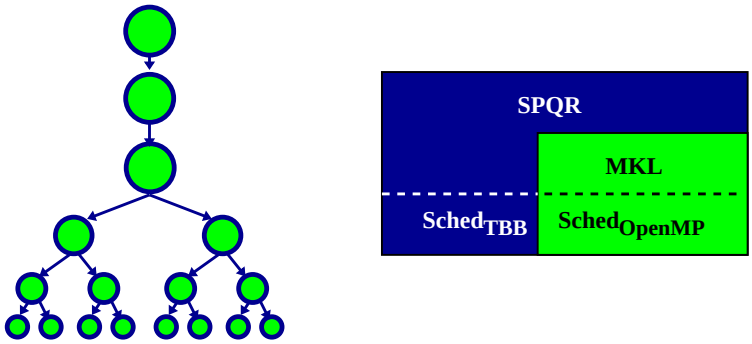
As our first application case study, we used an algorithm developed by Tim Davis for sparse QR factorization (SPQR) [Daved], which is commonly used in the linear least-squares method for solving a variety of problems arising from geodetic survey, photogrammetry, tomography, structural analysis, surface 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.

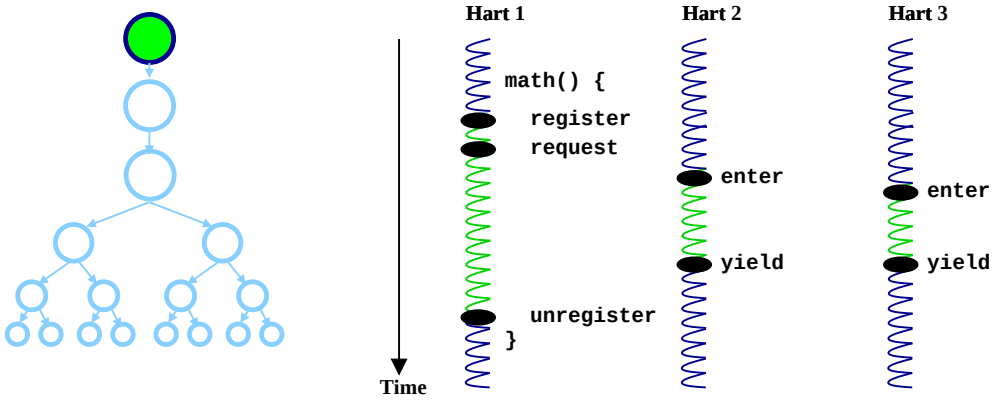
Original Out-of-the-Box Implementation Davis implemented his algorithm using TBB to create the parallel tasks, each of which then calls parallel BLAS [LHKK79] matrix routines from MKL (Figure 7-3(a)). MKL, in turn, uses OpenMP to parallelize itself. Unfortunately, although the two types of parallelism should be complementary, TBB and MKL compete counterproductively with each other for machine 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. While each of the operations was carefully crafted by MKL to fit perfectly in the cache, they are now being multiplexed by the operating system and interfering with one another.

Original Manually Tuned Implementation To curtail resource oversubscription, Davis manually limited 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.

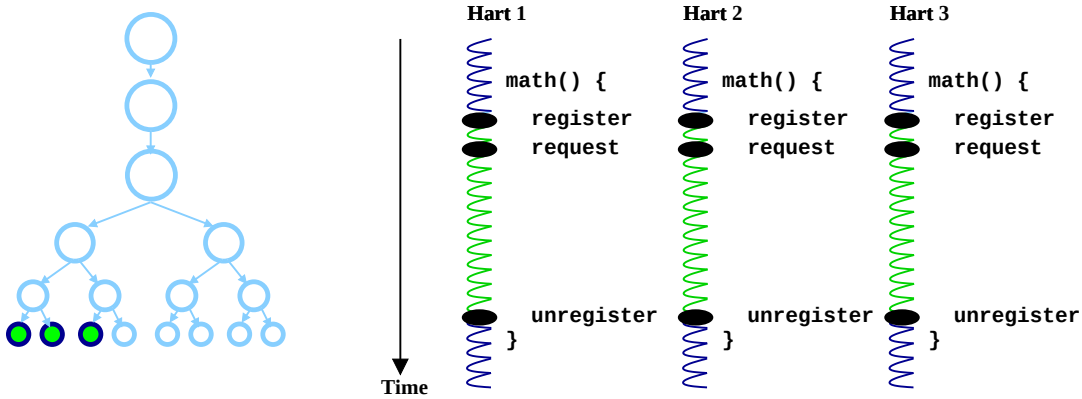
We reproduced Davis' manual tuning runs for all four original input matrices on our own machine (see Table 7.5 for a description of the input matrices). Figure 7-4 shows the performance of a representative input matrix across all the different thread configurations. The out-of-the-box configuration (OMP=16, TBB=16) is where each library creates the



(a)



(b)



(c)

Figure 7-3: (a) The SPQR computation represented as a task tree, where each task performs several parallel linear algebra matrix operations. The corresponding software stack for Tim Davis' implementation is also shown, where tasks are written using TBB, and each of the tasks calls MKL routines that are parallelized internally using OpenMP. (b) The execution of SPQR with Lithe at the point where only one task is executing and OpenMP gets all the harts. (c) The execution of SPQR with Lithe at the point where many tasks are executing and each OpenMP uses its only hart.

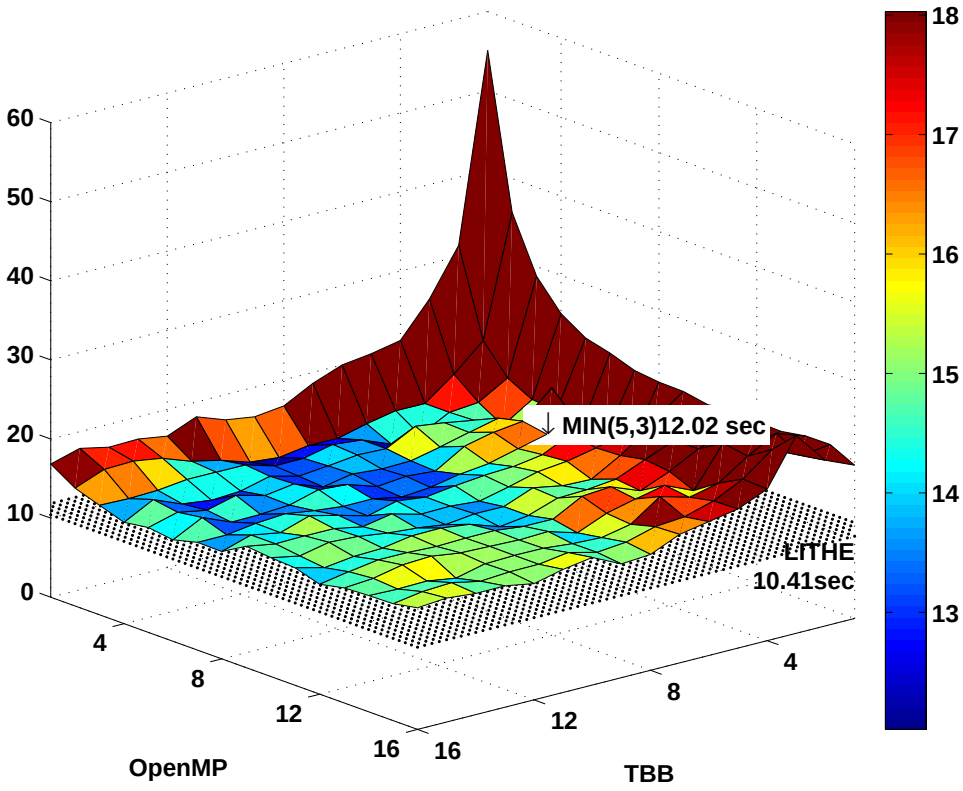


Figure 7-4: Runtime in seconds for the original SPQR implementation across different thread allocations to TBB and OpenMP for the deltaX input matrix, compared with Lithe.

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, TBB=1), it is not close to optimal. Many of the configurations that limit resource oversubscription achieved 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 7.6 shows the performance of noteworthy configurations for all of the input matrices. The out-of-the-box configuration performed worse than the manually tuned configuration for all inputs. Note that 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 7-5-**??**. As expected, Figure 7-5 shows that

	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 7.5: 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 7.6: SPQR (TBB/OpenMP) performance with and without Lithe on a 16-core AMD Barcelona (runtime in seconds).

the average number of threads for the out-of-the-box configuration was more than double the number of cores in the machine. Figure 7-6 shows the number of OS thread context switches that occur during the run. The number of context switches increased as the number of threads given to OpenMP increased. However, for any fixed OpenMP configuration, increasing the number of threads given to TBB decreased 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. Figure 7-7 and 7-8 show the L2 data cache misses and data TLB misses, respectively, both of which increased as the number of threads given to OpenMP/TBB increased.

Lithe 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 7.6 shows the performance results of using Lithe. This implementation even outperformed 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.

Figures 7-3(b) and 7-3(c) illustrate how Lithe enables the flexible hart sharing in SPQR. At one point in the SPQR computation, depicted by Figure 7-3(b), only one TBB task can be

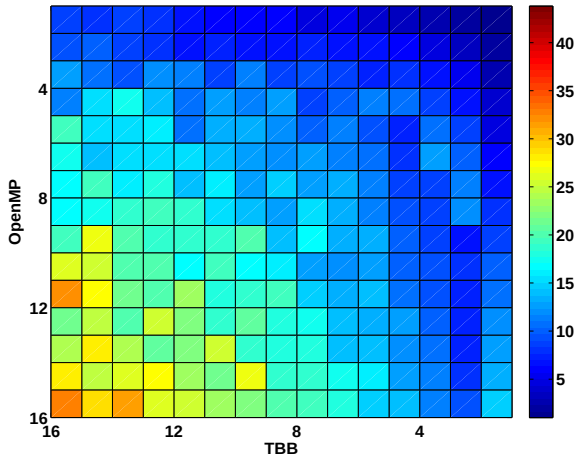


Figure 7-5: Average number of runnable OS threads for the original SPQR implementation across different thread allocations to TBB and OpenMP for the deltaX input matrix.

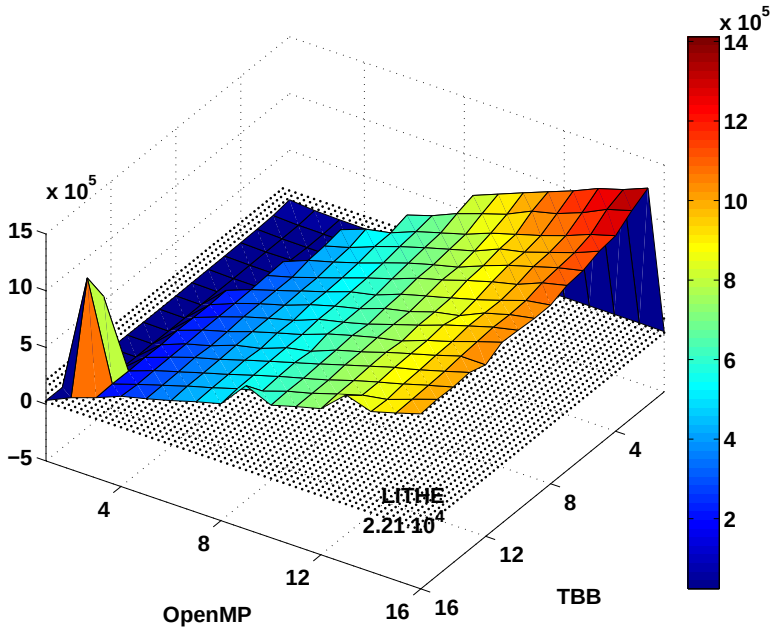


Figure 7-6: Number of OS context switches for the original SPQR implementation across different thread allocations to TBB and OpenMP for the deltaX input matrix, compared with Lithe.

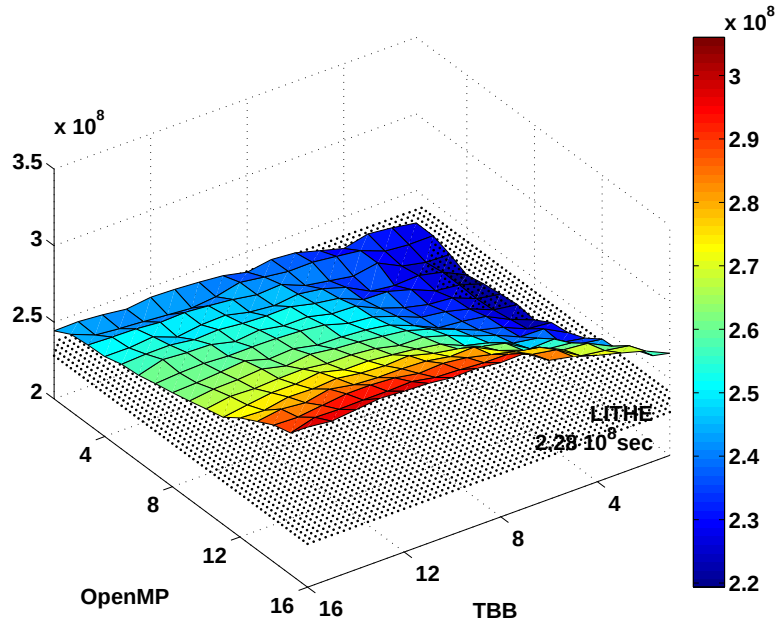


Figure 7-7: Number of L2 data cache misses for the original SPQR implementation across different thread allocations to TBB and OpenMP for the deltaX input matrix, compared with Lithe.

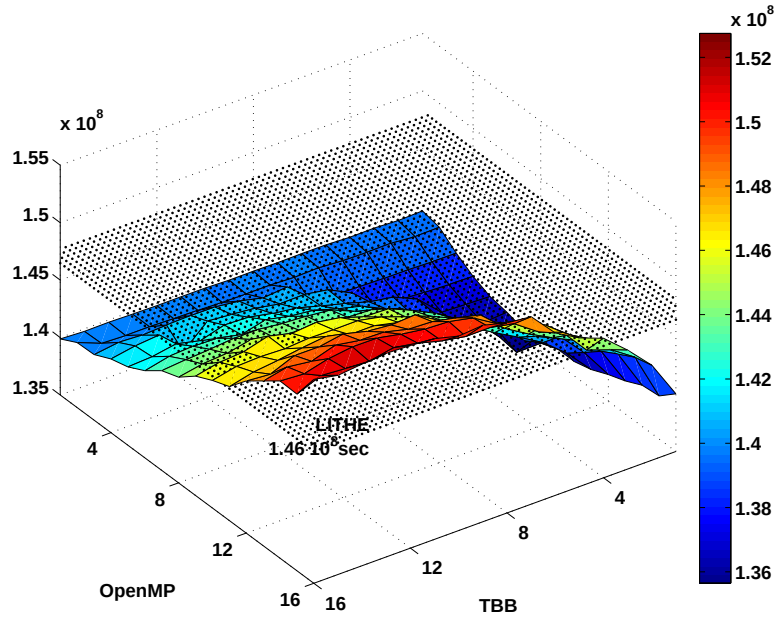


Figure 7-8: Number of data TLB misses for the original SPQR implementation across different thread allocations to TBB and OpenMP for the deltaX input matrix, compared with Lithe.

	landmark	**deltaX**	**ESOC**	**Rucci1**
Out-of-the-Box	1.03×10^5	1.04×10^6	3.40×10^6	6.08×10^6
Manually Tuned	5.77×10^4	2.40×10^4	1.11×10^6	3.19×10^6
Lithe	1.47×10^4	2.21×10^4	7.79×10^4	2.50×10^5

Table 7.7: SPQR Context Switches.

executing. The task invokes an MKL math routine, which in turn instantiates an OpenMP scheduler to manage itself. The OpenMP scheduler registers itself, then requests additional harts from the TBB parent scheduler. Since TBB has nothing else to do, it will grant all of its harts to OpenMP. When the math routine is completed, OpenMP will yield all of its harts back to TBB and unregister itself. At another point in the SPQR computation, depicted by Figure 7-3(c), there are multiple TBB tasks executing in parallel. Every task invokes an MKL math routine, each of which instantiates its own OpenMP scheduler that registers and requests from the parent TBB scheduler. Because all of the harts are already busy executing their respective tasks, the parent TBB scheduler does not grant any of these requests. Each OpenMP scheduler has to make do with the hart it already holds, and completes the math routine using only one hart.

To pinpoint the precise benefits of using Lithe, we compared the detailed metrics regarding active threads, cache misses, and context switching for the Lithe implementation to those of the original implementation. Table 7.7 shows that the Lithe implementation had orders of magnitude fewer OS thread context switches than the out-of-the-box and manually tuned configurations across all input matrices. As shown in Table 7.8 and 7.9, the Lithe implementation had fewer L2 cache misses and data TLB misses than the out-of-the-box implementation. There were, however, a few cases where the Lithe implementation incurred more misses than the manually tuned configuration. For instance, the Lithe implementation incurred more L2 data cache misses for the landmark matrix. This is because the landmark input matrix is extremely small, so the manually tuned configuration decomposed the task graph and matrix subproblems into very few pieces, and undersubscribed the machine in order to save on parallelization overhead. In contrast, the TBB and OpenMP schedulers in the Lithe implementation used the same task decomposition logic as the out-of-the-box implementation to provide a fair comparison between the two. The Lithe implementation, thereby, executed many more matrix subproblems in parallel than the manually tuned implementation, and incurred more L2 cache misses. A quick fix to this problem would be to specify a minimum task granularity for both TBB and OpenMP to use for task decomposition. Additionally, the Lithe implementation generally incurred more TLB misses than the manually tuned implementation, since the Lithe runtime dynamically allocates memory for each new scheduler. A future runtime implementation can use more sophisticated memory management techniques.

	landmark	deltaX	ESOC	Rucci1
Out-of-the-Box	9.53×10^6	3.04×10^8	1.36×10^9	5.47×10^9
Manually Tuned	7.65×10^6	2.29×10^8	1.20×10^9	5.55×10^9
Lithe	8.23×10^6	2.28×10^8	1.11×10^9	5.19×10^9

Table 7.8: SPQR L2 Data Cache Misses.

	landmark	deltaX	ESOC	Rucci1
Out-of-the-Box	9.27×10^5	1.53×10^8	5.19×10^8	1.42×10^9
Manually Tuned	5.50×10^5	1.37×10^8	4.84×10^8	1.33×10^9
Lithe	4.57×10^5	1.46×10^8	4.98×10^8	1.37×10^9

Table 7.9: SPQR Data TLB Misses.

Performance Portability We also demonstrate that Lithe provides similar performance improvements for the same SPQR binary on another x86 machine with a different core count and cache configuration. The second machine used was a dual-socket Intel Clovertown with four 2.66GHz Xeon-E5345 cores per socket (8 cores total). Each core had a 8-way associative 32KB L1 cache, and every two cores shared a 16-way 4MB L2 cache. The same OS platform was used.

Table 7.10 shows the performance of noteworthy configurations for the same input matrices. For all four inputs, Lithe version outperformed the manually tuned version, which in turn outperformed the out-of-the-box configuration. For one of the inputs (landmark), the out-of-the-box parallel version performed even worse than the sequential version. Note that the manually-tuned thread configuration for each input on the Clovertown is completely unrelated to its manually-tuned thread configuration on the Barcelona, whereas Lithe achieves better performance on both machines executing the same application binary.

Discussion The SPQR case study came from a third-party domain expert, Tim Davis, who ran into the composition problem independently by simply trying to use TBB and MKL together to implement his algorithm. The out-of-the-box performance was bad enough that

		landmark	deltaX	ESOC	Rucci1
Seq OMP / Seq TBB	OMP=1, TBB=1	3.4	37.9	172.1	970.5
Par OMP / Seq TBB	OMP=8, TBB=1	4.0	26.7	126.1	633.0
Seq OMP / Par TBB	OMP=1, TBB=8	2.5	15.6	74.4	503.0
Out-of-the-Box	OMP=8, TBB=8	4.1	26.8	111.8	576.9
Manually Tuned	OMP=1, TBB=8	2.5			
	OMP=2, TBB=4			70.8	360.0
	OMP=2, TBB=8		14.5		
Lithe		2.3	10.4	60.4	248.3

Table 7.10: SPQR (TBB/OpenMP) performance with and without Lithe on a 8-core Intel Clovertown (runtime in seconds).

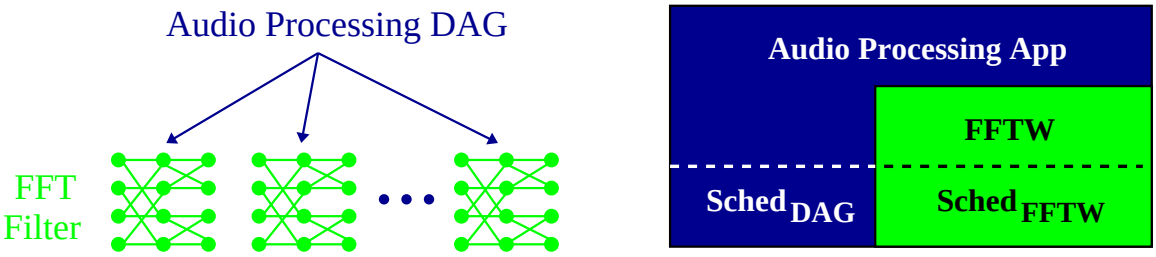


Figure 7-9: The audio processing application consists of a DAG that applies an FFT filter to each of its input channels. The implementation composes a custom DAG scheduler with the FFTW scheduler.

he had to understand the implementation of TBB and MKL (and even OpenMP, which he does not even invoke directly), and tune low-level knobs to control the active parallelism of each library. This is an extremely unproductive form of code reuse. As a domain expert, Davis should have only needed to understand the task abstractions of TBB and linear algebra interface of MKL, and should be able to leave resource management details to TBB and OpenMP. Furthermore, Davis' manually tuned performance results are very fragile. He would have to do additional tuning whenever he wants to run on a new machine, process a new input matrix, use a new release of TBB or MKL, or invoke the SPQR codes from within a parallel application. With Lithe, he would only have to link with the Lithe-compliant versions of TBB and OpenMP to achieve performance across different platforms, library releases, and execution environments.

7.3.2 Real-Time Audio Processing

As our second case study, we modeled a real-time audio processing application. This is an example within a class of computer music applications, which uses software running on commodity hardware to aid in the composition, synthesis, and live performance of digital music [Roa96].

The application takes one or more input streams of audio signal samples, processes them in some way, and generates some number of output streams. The application can be represented by a directed acyclic graph (DAG), where nodes are "plugin" modules that perform the actual processing, and edges indicate the flow of data. The application receives interrupts at some frequency, determined by the sampling frequency and block size. Before the next interrupt arrives, the application must process a block of input samples by executing the plugins in the DAG to generate a block of output samples. For the purposes of this study, we analyzed the behavior of a single type of plugin: a finite impulse response (FIR) filter, implemented using forward and inverse fast Fourier transforms from the FFTW library. An FIR filter with an N sample impulse response is implemented by performing an FFT and IFFT with a frame size of $2N$.

There are two types of parallelism that can be exploited during the execution of the

plugins in the DAG: "inter-plugin" parallelism, where multiple plugins execute in parallel, and "intra-plugin" parallelism, where the computation performed by a single plugin is parallelized. As shown in Figure 7-9, the application has a custom scheduler that schedules all of the plugins in the application's DAG. The original version of the DAG scheduler maps the filters onto a pool of pthreads, and invokes the original implementation of FFTW routines. The Lithe version uses the same task ordering to map the filters onto available harts, and invokes the Lithe implementation of the same FFTW routines.

To evaluate the audio processing application and its FFTW library component, we used a dual-socket Intel Nehalem with four 2.3 GHz Xeon-X5520 cores per socket and 2 hardware threads per core (16 harts total). Each core has private access to an 8-way associative 32KB L1 cache, and an 8-way associative 256KB L2 cache. Each socket has a 16-way associative 8MB outer-level cache shared by all 4 cores. We also used the 2.6.31 64-bit Linux kernel, which includes the Completely Fair Scheduler and the NPTL fast pthread library. We chose to use a different machine setup because it was already the default for our domain expert collaborators, and provided more performance predictability than our Barcelona setup that had frequency scaling enabled.

Baseline FFTW Performance First, we examined the baseline performance of a single FIR filter plugin. We measured the execution times of the FIR filter using impulse responses of two different lengths with varying degrees of parallelism. A "N-way parallel" configuration means that the FFTW library decomposed its tasks assuming that it would have N processing resources, and was allowed to run on N hardware thread contexts. The FFTW autotuner picked the optimal settings for each configuration offline, so the tuning is not part of the measured run time. FIR filters used in real-time audio processing applications tend to have relatively short impulse responses. Preliminary experiments confirmed that for the filter sizes we were interested in, the cost of cross-socket communication outweighed any potential benefits of parallel execution. Consequently, we only measured the baseline performance when all the computation was performed using the cores on a single socket of the Nehalem.

Figure 7-10 shows the performance of both the original and ported implementations computing an FFT and IFFT of size 131,072. The x-axis reflects the different amounts of parallelism that the FFT was allowed to use within the socket. The y-axis is the performance in terms of execution time, normalized to the single-threaded performance of the original FFTW implementation. The Lithe implementation performs competitively with the original implementation, and scales even better than the original. This is because the Lithe implementation has much lower synchronization overheads than the original implementation. The FFT tasks in the Lithe implementation are managed by a user-level scheduler, whereas the tasks in the original implementation are pthreads managed by the OS scheduler.

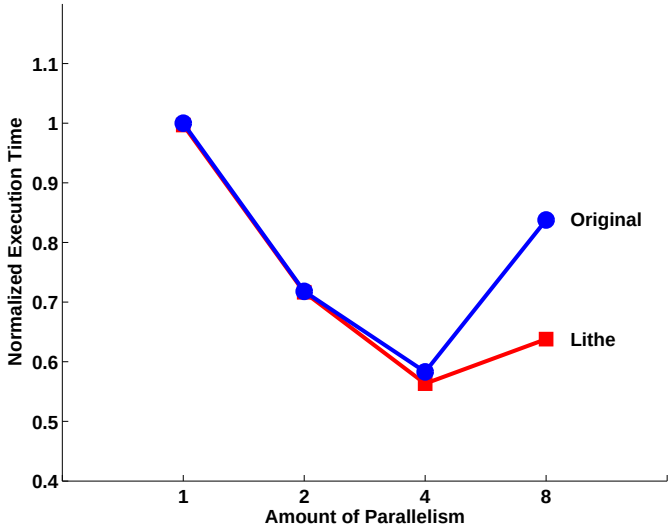


Figure 7-10: Performance comparison of the original and ported implementation of the FFTW library computing an FFT of size 131,072.

In the context of a real-time audio processing application, an FIR filter with 65,536 taps is considered to be large, as the most efficient way to compute the necessary FFTs and IFFTs requires waiting for that many input samples to accumulate before any processing can occur. The latency that this introduces is unacceptably long for real-time contexts, such as live performances. Figure 7-11 shows the performance of a smaller FIR filter (impulse response length = 16,384) that is more representative of a plugin that would be used in a real-time context. FFTs this small are usually not parallelized, because the synchronization cost may surpass the parallel execution speedup, as demonstrated by the poor scalability of the original implementation shown in Figure 7-11. In contrast, the Lithe implementation scales much better because of the reduced synchronization costs of user-level scheduling enabled by Lithe.

Performance of the Overall App Next, we study the behavior of multiple FFTW schedulers composed with a custom DAG scheduler within the audio processing application. The application uses FIR filters with impulse responses that are 16,384 samples long and implemented using 32,768 point FFTs and IFFTs. Each filter is granted 2-way parallelism, and each trial involved processing approximately 5 seconds of audio at a 96 kilohertz sample rate. Figure 7-12 compares the performance of the application with the original and Lithe implementations of the FFTW and DAG schedulers running on the Nehalem. When the number of channels is small, the Lithe implementation performs competitively with the original. When the number of channels is large, and the total amount of potential parallelism (double the number of channels, since 2-way parallel FFT filter is applied to each channel) exceeds the number of hardware thread contexts (16 harts on the Nehalem), the Lithe

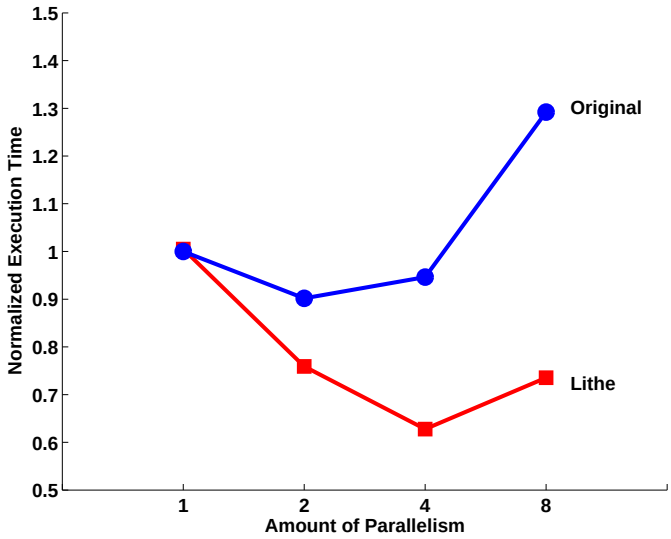


Figure 7-11: Performance comparison of the original and ported FFTW implementation of the FFTW library computing an FFT of size 32,768.

implementation outperforms the original because it curbs oversubscription.

Discussion Fixed-frame-rate multimedia applications, like real-time audio processing, have very different performance metrics than traditional throughput computing applications, like sparse QR factorization. As long as the application meets the desired frame rate, there may be no need to run any faster. However, using Lithe can still benefit these applications. Speeding up the application's core computations can leave enough time per frame to either power down the machine to save on energy, or perform additional computations to increase the audio or visual quality as perceived by the end user [CKSP09]. Another important consideration for applications with real-time constraints is performance predictability. By enabling an application to precisely manage the execution of its parallel components, Lithe eliminates much of the unpredictably resulting from OS scheduling decisions.

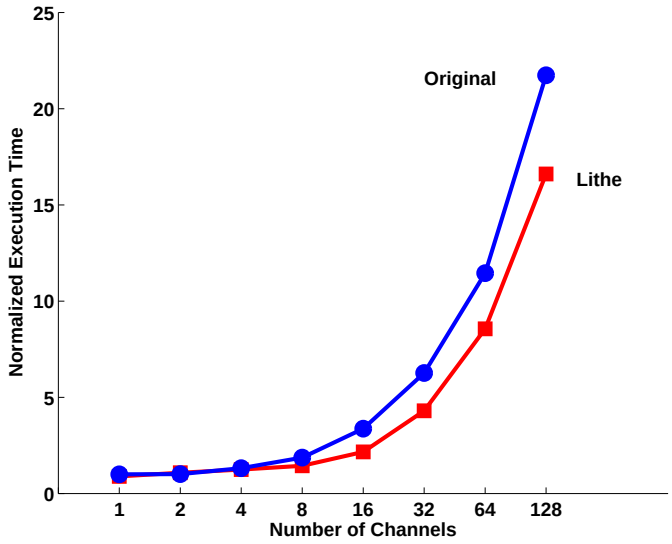


Figure 7-12: Performance of the audio processing application with varying number of input channels, with and without Lithe-compliant schedulers.

Chapter 8

Toward Composability

This chapter discusses the implications of adopting Lithe, and the extent of changes necessary for writing composable parallel software modules.

8.1 Implications for Application Development

First, and of foremost importance, most programmers should be blissfully unaware of Lithe. Many parallel programmers will use task abstractions and synchronization primitives, such as those provided by TBB and OpenMP, rather than managing their own resources. Even more programmers will simply invoke image processing, machine learning, and other libraries that happen to execute in parallel. With Lithe working transparently under the hood, these high-level programmers can now compose arbitrary abstractions and libraries without worrying about inadvertently hurting performance.

This is made possible by Lithe's conscientious design decision to maintain a clear separation of interface and implementation. Lithe provides a resource management interface orthogonal to the standard function call interface, so that there is no distinction between calling a sequential function and calling a parallel function. After all, parallelism is simply a different way of implementing the same functionality.

Additionally, Lithe facilitates a smoother transition from sequential implementation to parallel implementation. Applications can be incrementally parallelized, since the functional interface of each module remains unchanged, thus providing backward compatibility with existing sequential libraries.

8.2 Implications for Parallel Runtime Development

Lithe supports low-level programmers, typically parallel language and library developers, who want to manage their own resources and schedule their own tasks. Rather than creating an arbitrary number of OS threads, they are now given harts to manage.

As with all other forms of cooperative scheduling, Lithe-compliant synchronization operations cannot wait indefinitely for their rendezvous to complete. Monopolizing a hart may lead to poor performance, or even deadlock, since the operations needed to complete the rendezvous may not have resources to execute on. In practice, this is not a serious restriction, as many synchronization primitives for time-multiplexed threads already adopt a hybrid approach between spinning and blocking, since the penalty of load imbalance severely outweighs the performance gains of busy waiting.

To share resources efficiently between multiple schedulers, Lithe-compliant schedulers should also be able to adapt to a dynamically changing number of harts. Because of the cooperative model of Lithe, however, schedulers only have to adapt to *additional* harts, since they control when to relinquish the harts that they already have. Below, we categorize existing schedulers based on their adaptivity to the execution environment, and describe how to extend each category of schedulers to deal with extra harts. Since most schedulers today need to dynamically load balance to achieve good performance, adapting to additional harts becomes a trivial extension.

Fixed Decomposition, Static Mapping (Compile Time) In this least adaptive category, the computation is statically decomposed into tasks, and the tasks are statically mapped onto a fixed number of threads. The static mapping enables tasks to be carefully orchestrated to achieve an optimal schedule. This is especially prevalent in embedded systems, where the execution environment is fairly predictable, and the execution and communication times vary little between runs (e.g. [Sat09]). This type of scheduler can never use more harts than the anticipated number of threads, although the threads can be dynamically multiplexed onto a smaller or equal number of harts.

Fixed Decomposition, Static Mapping (Run Time) To provide portability across machines with different number of cores, some schedulers determine the number of threads to use statically at run time. All scheduling decisions are thus parameterized by the number of threads. Most SPMD schedulers, such as OpenMP, fit in this category. The data is divided statically at run time so that each thread operates on an equal amount of data.

To port these schedulers to Lithe, the number of threads can be set to be higher than the number of harts expected (i.e. the maximum number of harts supported by that machine), and the threads can be dynamically multiplexed onto the harts. Like the Lithe port of OpenMP, a thread should be run on a hart until its completion or until its context blocks on a synchronization operation, as not to reintroduce the destructive interference of arbitrary time-multiplexing. However, if the computation is not strongly scalable, the number of threads needs to be carefully set to reduce parallelization overhead while not causing load imbalance.

Fixed Decomposition, Dynamic Mapping Load-balancing schedulers map tasks dynamically onto a fixed number of threads to keep all the cores fully utilized regardless of the number and duration of the tasks. Although some of the tasks may be spawned conditionally at run time, all of the potential tasks were specified statically by the programmer.

Load-balancing schedulers can be work-stealing, work-sharing, or a hybrid of both. In work-stealing schemes, each thread asks for additional work when it has no more to do. In work-sharing schemes, each thread offloads work to other threads when it has too much to do. These schedulers can also choose to maintain either a global task queue or one task queue per thread. The TBB runtime is an example of a work-stealing scheduler with decentralized queues.

Porting load-balancing schedulers to Lithe is straightforward. Rather than dynamically mapping the tasks to a fixed number of threads, the scheduler can dynamically map the tasks to a monotonically increasing number of harts. In work-stealing schemes, any additional hart will simply act as any existing hart that has finished all of its tasks, and start asking for more work. In work-sharing schemes, any additional hart will make itself available for others to offload work to. In decentralized schemes, a new scheduling queue will be allocated for each new hart.

Variable Decomposition, Dynamic Mapping To eliminate unnecessary parallelization overhead, some schedulers effectively decompose the computation dynamically based on the available resources. For example, Cilk [BJL+95] has compiler support to coarsen the task granularity on the fly. The Cilk programming model encourages programmers to expose all of the parallelism available in the computation without having to worry about parallelization overhead. The compiler, in turn, generates both a sequential and a parallel version of each parallelizable function. The runtime invokes the fast sequential calls by default, and the parallel versions are only invoked when there are additional available resources.

As another example, the OpenMP runtime provides a "guided" option for scheduling loop iterations. A large chunk of contiguous loop iterations is mapped onto each thread during the initial allocation. Each thread is given an exponentially smaller chunk to execute in every successive allocation. The user can set a minimum chunk size to ensure that the amount of computation within the chunk justifies the parallelization overhead. The larger chunk sizes amortizes the same parallelization overhead across more loop iterations, while the smaller chunk sizes enables better load balancing.

Porting Cilk to Lithe should be very similar to porting TBB to Lithe, and porting OpenMP's guided scheduler to Lithe should be comparable to porting any other work-sharing scheduler to Lithe. Because these schemes minimize the cost of task over-decomposition, they effectively minimize the cost of the adaptivity encouraged by Lithe.

Dynamic Algorithmic Selection The most adaptive schedulers choose different implementations, even different algorithms, dynamically based on the number of available resources. For example, Petabricks [ACW+09] switches dynamically between different algorithms at different phases of a recursive application in order to always utilize the algorithm best suited for each subproblem size. Petabricks already supports multiple architectures. Hence, it has the ability to generate optimal code snippets for different numbers of cores.

Porting Petabricks to Lithe should be painless. Since the Petabricks runtime uses a work-stealing scheduler, it can start utilizing additional harts right away. At each recursion level, it can evaluate what algorithm to use based not only on the subproblem size, but also on the number of harts currently available.

Chapter 9

Related Work

Lithe's cooperative hierarchical scheduling model is essentially the confluence of two lines of research: *hierarchical scheduling* and *cooperative scheduling*. This chapter reviews previous scheduling frameworks in both lines of research, and describes how Lithe differs from previous work to enable true modularity and maximal flexibility when composing parallel codes. In addition, this chapter describes two recent industry solutions that aim to solve the composition problem through a global user-level resource management layer.

9.1 Hierarchical Scheduling

Hierarchical scheduling schemes utilize hierarchies for modular and heterogeneous resource management. These schemes enable the co-existence of different scheduling policies that suit a wide range of application needs. They also aim to provide modularity by allowing different components to insulate their scheduling policies from each other.

Most of the previous hierarchical scheduling schemes differ fundamentally from Lithe in how their schedulers interact. Schedulers must register threads with either their parent (or the framework), effectively breaking modularity and forcing the parent (or the framework) to manage individual threads on the child scheduler's behalf. The scheduler is only granted harts implicitly when its parent (or the framework) decides to invoke these threads. In contrast, a Lithe scheduler grants harts to a child scheduler to do with as it pleases. This allows a scheduler to incur parallelization overhead as needed based on the amount of resources available, rather than forcing it to decompose its computation into a fixed number of threads a priori.

Furthermore, many of the previous schemes are implemented in the OS to coordinate schedulers of different applications and processes, and rely on time-multiplexing to forcibly transfer control of resources from one scheduler to another. In contrast, Lithe schedulers cooperatively yield control of resources at their own will. This eliminates unnecessary context-switching overhead and destructive interference, while providing cleaner semantics for schedulers to reason about.

We describe each of the hierarchical scheduling schemes in more detail below.

Lottery Scheduling In lottery scheduling [WW94], a scheduler is given some number of tickets proportional to its priority, and each scheduler can in turn allocate its tickets to its child schedulers and its own threads. The OS holds a lottery at each scheduling quantum to choose a ticket at random, and runs the thread with that ticket during the quantum. Thus, lottery scheduling is probabilistically fair. The time share of resources allotted to each scheduler is expected to be proportional to the number of tickets it holds.

CPU Inheritance In CPU Inheritance [FS96], a scheduling thread can temporarily donate its CPU time to another thread. The client thread may also be a scheduling thread, and may decide to re-donate the CPU time to one of its client threads. For a scheduler to have control over multiple CPUs, it must create multiple scheduling threads. Furthermore, a scheduler inadvertently loses control of a CPU upon blocking on I/O or a timer interrupt, as specified by its parent.

Hierarchical Loadable Scheduler (HLS) The Hierarchical Loadable Scheduler (HLS) architecture [Reg01] allows soft real-time schedulers to be dynamically loaded into the OS. The user controls how the schedulers are inserted into the hierarchy, and can dynamically assign and reassign application threads to any scheduler. HLS defines a standard scheduler interface based on the virtual processor abstraction. Each scheduler registers a virtual processor with a parent scheduler for every CPU it would like to manage. A parent scheduler can grant and revoke its child's virtual processor, effectively indicating whether the virtual processor is running on an actual physical processor. Since each virtual processor only interfaces between a single pair of parent-child schedulers, granting access to a physical processor down the hierarchy involves activating a different virtual processor at every generation. All scheduling activities are serialized, making HLS difficult to scale to many cores.

Converse Converse [KYK96] is the most similar to Lithe. It strives to enable interoperability between different parallel runtimes, provides a standard scheduler interface, and builds synchronization mechanisms on top of the scheduling interface. However, unlike Lithe, Converse breaks modularity by forcing parent schedulers to manage their children's threads. Converse also does not support true plug-and-play interoperability, since a child scheduler must know its parent's specific interface in order to register its threads. In addition, the hierarchical relationship of Converse schedulers does not follow the call graph, making it harder to use local knowledge to prioritize between schedulers.

Concert The Concert framework [KC02] 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 virtualized OS threads.

Tlib Tlib [RR02] enables scientific applications to be decomposed into hierarchical SPMD tasks. Each task can statically specify the relative resource allocation of its child tasks as percentages of its own resources. The Tlib runtime then dynamically maps the tasks onto their allocated resources, and runs peer tasks consecutively if the number of peer tasks exceeds the number of resources belonging to the parent task.

9.2 Cooperative Scheduling

In cooperative scheduling, tasks voluntarily yield control of processing resources to let other tasks run. This enables control to be relinquished at convenient points, such as when the computation is blocked waiting for a synchronization or I/O operation to complete.

Lithe builds on the well-known technique of implementing cooperative multitasking by using continuations [Wan80] to save and restore state. Lithe augments 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. In addition, Lithe's transition stacks eliminates the need for messy cross-context synchronization during context-switching, since the paused context cannot be resumed until its successor context moves onto its own stack.

The continuation-based cooperative multitasking schemes most closely related to Lithe are language-specific frameworks, as described below, that export basic parallel primitives used to implement customizable scheduling policies [FRR08, LTMJ07]. Although these schemes claim to support hierarchical nesting of schedulers, none describe how multiple schedulers would interact.

In fact, continuations are essentially parallel programming's counterpart to goto statements for sequential programming. Both continuations and goto statements are simply a mechanism for transferring control. However, when used in an unstructured manner, they are difficult to reason about, complicating the analysis and the verification of programs [Dij68]. Just as subroutines introduced structure and modularity into sequential programming, Lithe introduces structure and modularity into parallel programming.

Manticore Manticore [FRR08] is a scheduling framework for CML. Manticore exports a vproc primitive similar to Lithe's hart primitive. However, schedulers do not interact with each other to share resources, but rather all request additional vprocs from a global entity. Rather than enabling a wide variety of task abstractions and scheduling implementations, Manticore imposes a thread abstraction and scheduling queue primitive on all schedulers.

Glasgow Haskell Compiler (GHC) A recent extension of the Glasgow Haskell Compiler (GHC) [LTMJ07] exports a set of primitives and defines a standard scheduler callback interface for Haskell. GHC's HEC primitive is very similar to Lithe's context primitive, but GHC relies on OS threads rather than exporting a better resource abstraction to prevent oversubscription. Unlike Lithe, GHC also makes transactional memory a primitive, thus building scheduling operations on transactional memory rather than the other way around. In fact, GHC uses transactional memory to implement a `switch` primitive for atomic cross-context stack switching.

9.3 Global Resource Management Layer

There are two very recent industry solutions [Mic10, Int10], both released in the past one or two months, that aim to solve the same parallel library composition problem as Lithe. Rather than distribute control over resources in a hierarchically modular fashion, these schemes provide a global user-level resource management layer that decides how to allocate resources to all schedulers. We describe each of them below.

Concurrent Runtime (ConcRT) Resource Manager Microsoft's Concurrent Runtime (ConcRT) provides a comprehensive user-level framework for writing parallel programs in the Visual Studio 2010 integrated development environment for the Windows platform [Mic10]. To enable interoperability with third-party libraries, ConcRT exposes its lowest level in the component stack: the Resource Manager. It also exports two high-level libraries, the Parallel Patterns Library and Asynchronous Agents Library, both of which are built on top of the Resource Manager.

Similar to Lithe, the ConcRT Resource Manager represents a processing resource using a *virtual processor* abstraction. Unlike Lithe, the Resource Manager also exports a *thread* abstraction, and allows its user to specify a target oversubscription factor to suggest how many threads should be mapped onto the same virtual processor. Like Lithe, the Resource Manager provides execution *contexts* to enable cooperative blocking. Unlike Lithe, a context is defined as a thread that is currently attached to some scheduler, rather than execution state allocated and managed by a particular scheduler. A context can attach and detach itself to and from a scheduler at any time. The Resource Manager keeps a stack of schedulers associated with a context, so that the previous scheduler can resume control when the current scheduler is detached.

Before executing any work, a scheduler registers itself and requests resources from the Resource Manager. Each scheduler specifies the minimum and maximum number of virtual processors it wants. The Resource Manager will never allocate fewer resources than the minimum requested or more than the maximum requested. Higher-priority schedulers receive up to their maximum requests before lower-priority schedulers receive more than

their minimum requests. Schedulers of the same priority receive the same proportion of resources until their maximum requests are satisfied. If the Resource Manager cannot satisfy all minimum requests, then it will start sharing cores between schedulers.

The Resource Manager invokes a scheduler's designated callback to grant access to a set of virtual processors before the scheduler's request returns. The scheduler can then tell the Resource Manager which context to start or resume on each of those virtual processors.

Intel Resource Management Layer (IRML) Intel's Resource Management Layer (IRML) [Int10], released with the latest version of TBB (v3.0), uses a global user-level resource allocator to enable its TBB and OpenMP library implementations to share resources with one another, as well as with ConcRT's Parallel Patterns Library and other third-party libraries.

IRML is implemented as a distributed server model. Each scheduler instantiates its own IRML server to provide itself with threads to use. In one implementation variant, the different servers coordinate with one another through a centralized "bank." A server can grant one thread to its scheduler client for each "coin" that it obtains from the bank. A server returns a coin to the bank whenever it puts the corresponding thread to sleep. The bank can be overdrafted, indicating resource oversubscription. In another variant, IRML uses ConcRT to allocate resources. Each IRML server registers with and requests resources from the ConcRT Resource Manager on behalf of its scheduler client.

A scheduler asks for threads by informing its server about the number of jobs it wants to run. A scheduler can also inform its server about the number of non-IRML threads currently executing in the system, effectively hinting IRML to activate fewer threads if it wants to avoid oversubscription. This is an attempt to enable efficient composition with parallel codes that have not been ported to use IRML. IRML provides different server interfaces for different types of scheduler clients. The TBB server interface enables a TBB scheduler to adjust its job count over time. The OpenMP server interface enables an OpenMP scheduler to request N threads, and to know immediately how many of those requests have been satisfied (or to demand that all N threads be granted, regardless of the execution environment).

Chapter 10

Conclusion

With the increasing adoption of multicore microprocessors and the parallelization of software libraries, modularity and separation of concerns are evermore important. High-level programmers must be able to reuse and compose arbitrary codes without caring about low-level parallel resource management. Different software components should be customizable to provide productive domain-specific abstractions and deliver energy-efficient performance. This thesis has presented a solution that enables efficient composition of parallel libraries by allowing the libraries to cooperatively share processing resources without imposing any constraints on how the resources are used to implement different parallel abstractions.

10.1 Thesis Summary and Contributions

In this thesis, we have shown the difficulties of composing parallel libraries efficiently, as well as demonstrated the need for a systemic solution. We should not place the burden on programmers to choose between sequential and parallel library implementations depending on the calling environment, or to manually tune low-level thread allocation knobs. Worse, these programmers will simply export the same problem to their clients as their codes are reused.

Lithe's cooperative hierarchical model is a new way of thinking about scheduling and parallel composition. We argue that low-level scheduling libraries must be cognizant of the fact that they may be sharing the machine with other libraries, to ensure that their high-level clients never need to worry about resource management. We argue that the management of resources should be coupled with the hierarchical control flow, giving user-level schedulers even greater control and freedom than two-level scheduling environments [LKB+09].

Lithe's transfer callbacks are a lightweight, straightforward mechanism to pass control of a hart directly from one scheduler to another. The transition contexts enable clean transitioning between schedulers, without the typical messy cross-scheduler protocols for handing off stacks atomically.

In essence, Lithe is the parallel world equivalent to subroutines for enabling composition.

Just as subroutines replaced unstructured goto statements with structured control flow, Lithe replaces unstructured and implicit sharing of processing resources with cooperative hierarchical scheduling. The Lithe scheduler callback interface also provides the same type of interoperability between parallel software components, as provided by the function call ABI between sequential software components.

10.2 Future Work

We are looking into extending this work in the following main areas.

OS Support for Lithe Primitives The current implementation of the hart and context primitives are purely in user space. This was a deliberate design decision to enable parallel codes within an application to interoperate efficiently without the commitment of adopting a new operating system. Nonetheless, a kernel-level implementation of the primitives would 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. At any point in time, the OS gives each application a disjoint subset of the harts to manage. In Lithe-compliant applications, the base scheduler provided by the Lithe runtime would be in charge of interacting with the OS scheduler, shielding the OS scheduler from having to know about the scheduler hierarchy. The base scheduler would directly pass any harts granted by the OS scheduler to the root scheduler. It would also make requests for additional harts on behalf of the root scheduler, and return any unneeded harts yielded by the root scheduler back to the OS.

In contrast to user-level harts that are time-multiplexed by the OS, kernel-level harts can be revoked by the OS. At the very least, the OS should be able to prevent a malicious or buggy application from hogging too many resources and hanging the entire system. The OS may also want to do minimal coarse-grained reallocation of harts to start new applications, readjust for changing application needs, or multiplex resources when the system is slightly overloaded. We imagine the OS to use admission control (probably with input from the end user) rather than fine-grained time-multiplexing when the system is highly overloaded. Delaying the start of a few jobs is often preferable to making everything run badly. After all, no one wants background disk scans to kick in unexpectedly and mess up an important presentation.

We expect hart revocation to work similarly to Scheduler Activations [ABLL91] and Psyche [MSLM91]. Here is a sketch of one possible revocation protocol under Lithe. If the OS revokes an application's only hart, it is responsible for allocating another hart at a later time to resume the paused context. Otherwise, the OS would temporarily suspend the context of another hart belonging to the application, and borrow that hart to invoke

the application's revocation event handler. In Lithe-compliant applications, the revocation event handler would be provided by the Lithe runtime. This handler would determine the scheduler that was managing the revoked hart, and invoke its `revoke` callback. The `revoke` callback would be an addition to the Lithe standard scheduler callback interface, and would inform its scheduler about the revoked hart and the corresponding paused but runnable context. Like the `unblock` callback, `revoke` only updates the necessary scheduling state, and does not use the current hart to resume the paused context. Upon the return of the revocation event handler, the OS resumes the borrowed hart's interrupted context.

Furthermore, providing kernel support for contexts enables I/O calls to interoperate seamlessly with user-level schedulers. In contrast, a user-level implementation of contexts limits applications to using the asynchronous subset of the OS's I/O interface or employing special wrappers around blocking I/O calls [PHA10, AHT+02, vBCB03]. While waiting for a long-running I/O operation to complete, the kernel would simply return the hart to the application by invoking the application's block event handler with that hart. In Lithe-compliant applications, the block event handler would be provided by the Lithe runtime. This handler would determine the scheduler that was managing the blocked context, and invoke its `block` callback to repurpose the hart. Upon the completion of the I/O operation, the OS could borrow a hart to invoke the application's unblock event handler, which would, in turn, invoke the `unblock` callback of the context's scheduler. Alternatively, the appropriate scheduler can poll periodically to determine whether the I/O operation has completed.

Preemptive Programming Models Lithe's cooperative hierarchical scheduling model is designed to support the large set of applications that perform well with cooperative scheduling, for which preemption is often expensive and unnecessary [PBR+08]. For example, pre-emptive round-robin scheduling of threads within an application can just introduce gratuitous interference when the threads are at the same priority level. Hence, user-level schedulers should favor efficiency over an arbitrary notion of fairness.

However, an application may 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 the Lithe framework to enable a parent scheduler to ask for a hart back from its child. On one end of the design spectrum, a parent scheduler can simply revoke a hart from a child in a similar fashion to the OS scheduler revoking a hart from the application. On the other end of the design spectrum, a parent scheduler can send a request to its child, and wait for the child to yield a hart cooperatively.

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. [Iye04, LNA08]), beyond what is generally available in commercial machines today.

Ports of Additional Language Features In this thesis, we explored how parallel abstractions from different languages and libraries can be composed efficiently. In the future, we would also like to port managed and functional language runtimes onto our substrate, 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. We imagine experimenting with dynamic compilation to generate and register schedulers on the fly that better suit the program execution environment. And we imagine enabling different parallel components to share lazily-evaluated data, which is computed as needed by blocking the consuming context and alerting the producer to schedule the evaluation.

Exploration of Hierarchical Scheduling Policies We have faced the chicken-and-the-egg problem when trying to evaluate Lithe: application developers will not compose multiple programming models unless there is an efficient way to do so, and we cannot design an efficient composition framework without the driving applications. In fact, Tim Davis' implementation of sparse QR factorization is a very rare example in that composition was even attempted. We hope that by releasing the source code of Lithe, along with ported implementations of popular parallel libraries, and demonstrating Lithe's effectiveness (and nonintrusiveness) in some initial nontrivial examples, we will inspire more application developers to build their codes in a modular, specializable fashion.

With additional applications, we can better understand the interactions between parent and child schedulers, and explore a wider range of hierarchical scheduling policies to study their effects on the overall application performance. We also believe that the Lithe interface can be extended for parent schedulers to convey additional information to their children, but we are still waiting for a driving application that compels us to do so.

Bibliography

[ABLL91] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[ACW+09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

[AHT+02] Atul Adya, Jon Howell, Marvin Theimer, William Bolosky, and John Douceur. Cooperative task management without manual stack management. In *USENIX*, Monterey, CA, June 2002.

[Asa07] Krste Asanović. Transactors for parallel hardware and software co-design. In *High Level Design Validation and Test Workshop*, Irvine, CA, September 2007.

[BAAS09] Robert Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

[BBK09] Ras Bodik, Justin Bonnar, and Doug Kimelman. Productivity programming for future computers. In *Workshop on Fun Ideas and Thoughts*, Dublin, Ireland, June 2009.

[BJL+95] Robert Blumofe, Christopher Joerg, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[CKSP09] Romain Cledat, Tushar Kumar, Jaswanth Sreeram, and Santosh Pande. Opportunistic computing: A new paradigm for scalable realism on many-cores. In *Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

[CMD+01] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[Cyc] Cycling74. Max: Interactive visual programming environment for music, audio, and media. http://cycling74.com/products/maxmspjitter.

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

[DHRA06] Matthew Drake, Hank Hoffmann, Rodric Rabbah, and Saman Amarasinghe. MPEG-2 decoding in a stream programming language. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[Dij68] Edsger Dijkstra. Go to statement considered harmful. *Communications of the ACM*, February 1968.

[FJ05] Matteo Frigo and Steven Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 2005. Special Issue on "Program Generation, Optimization, and Adaptation".

[FKH+06] Kayvon Fatahalian, Timothy Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Supercomputing*, Tampa, FL, November 2006.

[FR02] Kathleen Fisher and John Reppy. Compiler support for lightweight concurrency. Technical report, Bell Labs, March 2002.

[FRR08] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *International Conference on Functional Programming*, Victoria, Canada, September 2008.

[FS96] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *Symposium on Operating Systems Design and Implementation*, Seattle, WA, 1996.

[Goo] Google. The Go programming language. http://golang.org.

[Gro] Khronos Group. OpenCL. http://www.khronos.org/opencl.

[GSC96] Seth Goldstein, Klaus Schauser, and David Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, August 1996.

[Int07] Intel. *Math Kernel Library for the Linux Operating System: User's Guide*. October 2007.

[Int10] Intel. Threading building blocks 3.0 for open source, May 2010. http://www.threadingbuildingblocks.org/ver.php?fid=154.

[Iye04] Ravi Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *International Conference on Supercomputing*, Saint-Malo, France, June 2004.

[JMF99] Haoqiang Ji and Jerry Yan Michael Frumkin. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NASA Ames Research Center, October 1999.

[Kar87] Alan Karp. Programming for parallelism. *Computer*, May 1987.

[KC02] Vijay Karamcheti and Andrew Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Supercomputing*, Orlando, FL, November 2002.

[KLDB09] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa Badia. Scheduling linear algebra operations on multicore processors. Technical report, LAPACK, February 2009.

[KYK96] Laxmikant Kale, Joshua Yelon, and Timothy Knauff. Threads for interoperable parallel programming. *Languages and Compilers for Parallel Computing*, April 1996.

[LHKK79] Charles Lawson, Richard Hanson, David Kincaid, and Fred Krogh. Basic linear algebra subprograms for FORTRAN usage. *Transactions on Mathematical Software*, September 1979.

[LKB+09] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

[LNA08] Jae Lee, Man-Cheuk Ng, and Krste Asanović. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *International Symposium on Computer Architecture*, Beijing, China, June 2008.

[LTMJ07] Peng Li, Andrew Tolmach, Simon Marlow, and Simon Peyton Jones. Lightweight concurrency primitives. In *Workshop on Haskell*, Freiburg, Germany, September 2007.

[Mat04] Timothy Mattson. *Patterns for Parallel Programming*. September 2004.

[Mic10] Microsoft. Concurrency Runtime, April 2010. http://msdn.microsoft.com/en-us/library/dd504870.

[MSLM91] Brian Marsh, Michael Scott, Thomas LeBlanc, and Evangelos Markatos. First-class user-level threads. *Operating Systems Review*, October 1991.

[NY09] Rajesh Nishtala and Kathy Yelick. Optimizing collective communication on multicores. In *Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

[PBR+08] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough! a study of operating system timer usage. In *Eurosys*, Glasgow, Scotland, April 2008.

[PHA10] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Conference on Programming Language Design and Implementation*, Toronto, Canada, June 2010.

[Reg01] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001.

[Rei07] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[Roa96] Curtis Roads. *The Computer Music Tutorial*. The MIT Press, February 1996.

[RR02] Thomas Rauber and Gudula Rünger. Library support for hierarchical multiprocessor tasks. In *Supercomputing*, Baltimore, MD, November 2002.

[Sat09] Nadathur Satish. *Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems*. PhD thesis, UC Berkeley, January 2009.

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

[TEE+96] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[TSG07] Guangming Tan, Ninghui Sun, and Guang Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Symposium on Parallel Algorithms and Architectures*, San Diego, CA, March 2007.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea. In *Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[vBCZ+03] Rob von Behren, Jeremy Condit, Feng Zhou, George Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference on LISP and Functional Programming*, Stanford, CA, August 1980.

[Wil] Paul Wilson. An introduction to scheme and its implementation. http://www.cs.utexas.edu/ftp/ftp/pub/garbage/cs345/schintro-v14/schintro_toc.html.

[WOV+07] Sam Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplciation on emerging multicore platforms. In *Supercomputing*, Reno, NV, November 2007.

[WTKW08] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the Linux scheduler. *Operating Systems Review*, July 2008.

[WW94] Carl Waldspurger and William Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[YBC+07] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Workshop on Parallel Symbolic Computation*, London, Canada, July 2007.