I-Ting Angelina Lee

Research Statement

I aim to make parallel programming accessible for everyone, so that every programmer, particularly the non-experts, can rapidly develop high performance software that takes advantage of commodity multicore hardware. With the advent and widespread deployment of multicores, writing parallel programs has become a necessity. Due to the limits of power efficiency, this need will persist and gain greater importance in the years to come.

Writing parallel programs is inherently challenging, however. A parallel program, unlike a serial program, must manage and coordinate among multiple threads of control, and many challenges of parallel programming stem from this need to coordinate. For instance, load balancing — dynamically partitioning the computation so that each processing unit receives approximately the same load — often involves complex communication protocols among threads to encode the scheduling logic. Nondeterministic program behaviors that occur due to the relative timing of communicating threads, such as when one thread arrives at a lock before another, require the programmer to think through all possible thread interleavings and ensure that the program invariants hold.

I believe that a properly designed ***concurrency platform***, with appropriate linguistic abstractions and efficient underlying system support, can address these challenges and make parallel programming accessible for everyone. A concurrency platform provides a linguistic interface for the programmer to specify parallel computations and a software layer that coordinates, manages, and schedules the processing resources. In contrast to approaches which focus either on new language constructs or new systems primitives, I take a ***cross-layer approach*** that cuts across all layers of the computing stack, from linguistic abstractions to system-level mechanisms, with the components coherently integrated to provide a clean, powerful, and comprehensive platform for parallel programming.

My research centers around the development of a comprehensive concurrency platform with this cross-layer approach, and I have been actively pursuing my research agenda via three important directions:

1. designing clean linguistic abstractions to allow larger class of computations to be expressed [1–3, 7];
2. developing system-level mechanisms to facilitate the abstractions supported [4–6]; and
3. building tools to facilitate debugging and performance engineering of parallel computations [8, 9].

I prefer to design systems upon solid theoretical foundations, so that the system can provide a clear mental model which a user can utilize to debug and reason about program behavior. A good theoretical foundation leads to systems that offer provable guarantees, allowing researchers and practitioners to predict how the system will behave when program inputs or underlying hardware parameters change.

My research demonstrates the efficacy of this cross-layer approach. In joint work with Boyd-Wickizer et al., we augmented the virtual memory mechanism to provide support for ***thread-local memory mapping (TLMM)***, in order to efficiently support interoperability between parallel code and legacy serial binaries [4]. We developed the only known software solution to date, which involved integrating the use of TLMM into the platform, in addition to necessary changes to the compiler and runtime. My joint work with Shafi and Leiserson showed that the TLMM mechanism is also useful for facilitating reducers, a high-level linguistic abstraction that coordinates concurrent accesses to global variables, typically maintained by the runtime [6]. In joint work with Ladan-Mozes and Vyukov, we proposed ***location-based memory fences*** [5], a lightweight hardware mechanism that could significantly cut down scheduling overhead in the runtime.

Recently, I started investigating the technology necessary to enable deterministic parallel programming [6–8]. My collaborators and I developed linguistic and scheduling support for expressing ***deterministic pipeline parallelism***, which allowed us to determinize three PARSEC benchmarks (a popular multithreaded benchmark suite) that could not easily be expressed with the fork-join primitives supported by existing concurrency platforms. The proposed linguistic constructs are composable with fork-join primitives and follow

the same spirit: they allow the programmer to express the *logical* parallelism within an application (i.e., which subcomputations *may* run in parallel) without worrying about how they are scheduled.

As part of an on-going effort, I am developing tools that operate within the high-level linguistic abstractions provided by the concurrency platform. These tools constitute part of the software ecosystem necessary to aid debugging [8] and performance engineering [9] when writing parallel programs using the concurrency platform.

A Cactus Stack that Interoperates with a Linear Stack [4]

In a concurrency platform that supports fork-join parallelism such as Cilk, the runtime system employs an important abstraction called a cactus stack to support multiple stack views for all the active children running in parallel. In all known software implementations of cactus stacks, however, transitioning from serial code (using a linear stack) to parallel code (using a cactus stack) is problematic, because the type of stack impacts the calling conventions used to allocate activation frames and pass arguments. Consequently, concurrency platforms that employ a cactus stack have trouble supporting arbitrary calls between parallel and serial code, especially legacy and third-party serial binaries.

The lack of interoperability impedes the acceptance of dynamic multithreading concurrency platforms for mainstream computing. These platforms typically employ a work-stealing scheduler for automatic load-balancing, which guarantees provably good time and space bounds. There seems to be an inherent trade-off between supporting interoperability and maintaining good time and space bounds, however, and existing work-stealing concurrency platforms fail to satisfy at least one of these three criteria.

It turns out that, by augmenting the virtual memory mechanism in existing operating systems, one can devise a platform that satisfies all three criteria simultaneously. My collaborators and I modified the Linux operating system kernel to provide support for ***thread-local memory mapping (TLMM)***, a new memory mechanism that designates a region of the process's virtual-address space as "local" to each thread that can be mapped independently. With support for TLMM, we built ***Cilk-M***, a theoretically sound work-stealing runtime system which also maintains a cactus-stack abstraction compatible with the traditional linear stack, thereby achieving interoperability. Experimental results indicate that Cilk-M performs comparably compared to the Cilk Plus runtime system, a production-grade piece of software, and consumes much less memory for its cactus stack.

This work provides an example where the cross-layer approach is necessary. By augmenting the virtual memory mechanism to provide for TLMM, we enabled the legacy compatible cactus stack in the concurrency platform. The TLMM mechanism is also useful for other parallel linguistic abstractions, such as reducer hyperobjects, which I describe in a later section.

Deterministic Pipeline Parallelism [7]

Many concurrency platforms support fork-join parallelism, which represents an important step toward determinism. In the fork-join model, subroutines can be spawned in parallel, and the synchronization of subtasks is managed automatically by the runtime system. Not all applications can be naturally expressed with fork-join primitives, however. My collaborators and I attempted to determinize the PARSEC benchmarks by replacing the nondeterministic mechanisms used in the pthreaded implementations with the fork-join primitives, and only five could be made deterministic. This observation led to our work on ***pipeline parallelism***, which allows three additional benchmarks to be determinized.

It turns out that three simple keywords suffice to express any linear pipeline, a pattern commonly found in streaming applications from the domains of video, audio, and digital signal processing. These keywords compose naturally with fork-join primitives and follow the same spirit: they denote the logical parallelism of the pipeline and have ***serial semantics***, which allows the programmer to debug the functionality of her

code without worrying about how it is scheduled. Distinct from the pipeline model supported by many existing systems, these keywords allow one to express pipelines ***on-the-fly***, where the structure of the pipeline emerges as the program executes rather than being specified *a priori*.

We designed and implemented a provably-efficient Cilk-based work-stealing scheduler that supports both fork-join and pipeline parallelism, named ***Cilk-P***, and ported the three pipeline benchmarks from PARSEC to use Cilk-P. The resulting programs have comparable performance to the pthreaded implementations but are much simpler to code and comprehend: the parallel control dependencies enforced by the linguistics entirely obviate the need for nondeterministic mechanisms such as concurrent queues, locks, and conditional variables. Inspired by this work, Intel released a version of Cilk Plus that incorporates on-the-fly pipeline parallelism.

Memory-Mapping Support for Efficient Reducer Hyperobjects [6] (SPAA '13 Best Paper)

A determinacy race typically constitutes a bug; it can cause a program to exhibit nondeterministic behaviors depending on how the parallel subcomputations involved in the race are scheduled. Reducer hyperobjects proposed by Frigo et al. are a useful linguistic mechanism to avoid determinacy races when accessing nonlocal variables or data structures in a fork-join parallel computation. Existing support for reducers incurs high overhead per reducer access, however, rendering the mechanism inefficient.

In order to slash this overhead, my collaborators and I investigated an alternative approach using thread-local memory mapping, called ***memory-mapped reducers***, which leverage the virtual-address translation provided by the underlying hardware to manage accesses to reducers. Empirical results show that indeed the TLMM-based approach yields four times faster reducer access time compared to the existing approach.

Location-Based Memory Fences that Reduce Scheduling Overhead [5]

In a multithreaded runtime environment, such as a work-stealing scheduler used by concurrency platforms or a Java Virtual Machine (JVM), the implementation often employs a Dijkstra-like protocol, synchronizing two threads with only simple loads and stores on shared variables. Since a modern architecture typically implements a memory model other than sequential consistency, one must insert memory fences to ensure correct execution of the protocol, which incurs high overhead. For instance, half of the overhead from a *spawn* statement in Cilk that forks off a logically parallel task stems from the use of memory fences, as indicated by measurements on an AMD Opteron.

To mitigate this overhead, we proposed the concept of a ***location-based memory fence***, which behaves like an ordinary memory fence, but incurs overhead only when synchronization is necessary. We designed a light-weight hardware mechanism for location-based memory fences on architectures that implement "Total Store Order" or the "Processor Ordering" model, and showed that they have the same semantics as the ordinary program-counter based fences. This work provides an example where the high-level abstractions implemented in concurrency platforms can benefit from integrating the use of hardware primitives.

Other Linguistic Abstractions Provided by a Concurrency Platform [1–3]

I also worked to design simple and intuitive linguistics for synchronization and parallel control in concurrency platforms, to enable a wider class of parallel computations. In joint work with Danaher and Leiserson on JCilk [1, 2], we investigated semantics for exception handling in the context of dynamic multithreading, which was a scarcely studied topic at the time. JCilk's strategy of integrating dynamic multithreading with Java's exception handling yields some surprising semantic synergy. In particular, it allows programs with speculative computations, such as chess programming, to be programmed easily. Agrawal, Sukha, and I formulated ***ownership-aware transactions (OAT)*** [3], a transactional memory design that incorporates open nesting, a methodology for handling nested transactions proposed by Moss and Hosking, in a modular and disciplined fashion. The OAT model provides provable guarantees and constitutes the first transactional

memory design that incorporates open nesting in a provably safe fashion.

Tools for Nondeterministic Computations [8, 9] (Work in Progress)

Due to the nondeterministic nature of how the runtime handles reducers — although this nondeterminacy is encapsulated from the user when used correctly — existing tools such as the race detector and parallelism analyzer do not work with computations that employ reducers. Reducers, albeit designed to avoid races on concurrent accesses to shared variables, can still allow races to occur when used incorrectly. In joint work with Schardl [8], we are investigating a tool to detect incorrect usage of reducers and races that arise from incorrect usage. In joint work with Xing and Leiserson, we are expanding the parallelism analyzer to handle computations with a nondeterministic amount of work, such as ones that employ reducers.

Directions for Future Work

Enabling deterministic parallel programming: Nondeterminacy still poses a major challenge in parallel programming. Most parallel programs today are still nondeterministic and written using pthreads and nondeterministic mechanisms such as locks and conditional variables. Part of the reason is that existing parallel-programming environments are immature and do not provide sufficient support for deterministic parallel programming. Even though our work on pipeline parallelism represents progress towards determinism, seven out of the thirteen PARSEC benchmarks still require nondeterministic mechanisms for correct behavior.

This mixed success in eliminating nondeterministic mechanisms in PARSEC benchmarks demonstrates that there is likely to be no single "silver bullet" that can slay the nondeterministic beast. An analogy can be drawn with the situation in 1968 when Edsger Dijkstra wrote his famous "Go to statement considered harmful" letter to CACM. The `goto` statement, although simple, led to code that was hard to understand, much as locking and other nondeterministic mechanisms make parallel programs difficult to understand. The solution to the `goto` problem was to favor structured control constructs, but no single control construct sufficed. With respect to nondeterminacy, too, I believe that a relatively small, but nonsingular, set of constructs can enable well-structured deterministic parallelism.

I plan to investigate ways to achieve determinism in parallel programming. I take a broad view of deterministic parallel programming, and pursue solutions on multiple fronts: I will design and implement linguistic abstractions with deterministic interfaces that encapsulate nondeterminacy; seek and define alternative synchronization mechanisms so that the programmer can avoid using mechanisms that cause nondeterminacy; and lastly, investigate and build software tools to determinize executions to aid testing and debugging in the cases where nondeterminacy cannot be encapsulated or avoided.

Automating locality management: The need for locality management poses another major challenge, that is, managing the accesses to data so that the computation effectively utilizes the cache. Ineffective use of the cache hierarchy can manifest itself by incurring high memory latency and impeding scalability. Unfortunately, as the number of cores increases, the cache hierarchy can only get deeper and more complex.

Although a parallel program can underperform for various reasons, ineffective use of the cache hierarchy constitutes the most difficult performance bottleneck to overcome. The main difficulty stems from the fact that the issue of locality management seems to be inextricably linked with the problem of scheduling and the configuration of the underlying hardware. Yet, existing concurrency platforms provide little support for locality management and tend to make scheduling decisions independent of how the underlying hardware operates. Similarly, the hardware employs optimizations, such as prefetching, with heuristics that do not account for scheduling and over which the systems or the programmer have little control.

I plan to study ways in which the different layers along the computing stack interact with respect to locality, to gain a deeper understanding of the locality-management problem. I contend that, to address

the issue of locality management, potential solutions will span multiple layers, ranging from languages, to compilers, runtime systems, and operating systems, all the way down to hardware mechanisms. Last but not least, a set of well designed tools with intuitive visualizations can assist the programmer in narrowing down the source of the performance bottleneck.

The research directions that I have laid out involve taking a broad view of parallel computing and seeking innovative solutions that span multiple layers. I plan to take a holistic approach and invest efforts in all the layers of the computing stack. I would like to go to an institution where I have the flexibility to explore different areas of computer science, and collaborate with researchers from various backgrounds and with different expertise to advance the state of parallel computing.

References

[1] John Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Exception handling in JCilk. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005. Available at `http://hdl.handle.net/1802/2095`.

[2] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming (SCP)*, 63(2):147–171, December 2006.

[3] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, USA, February 2009. ACM.

[4] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, Vienna, Austria, September 2010. ACM.

[5] Edya Ladan-Mozes, I-Ting Angelina Lee, and Dmitry Vyukov. Location-based memory fences. In *SPAA '11: Proceedings of the 23rd ACM Symposium on Parallel Algorithms and Architectures*, pages 75–84, San Jose, California, June 2011.

[6] I-Ting Angelina Lee, Aamir Shafi, and Charles E. Leiserson. Memory-mapping support for reducer hyperobjects. In *SPAA '12: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 287–297, Pittsburgh, Pennsylvania, June 2012. **(Best paper)**.

[7] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. In *SPAA '13: Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 140–151, Montreal, Canada, July 2013.

[8] I-Ting Angelina Lee and Tao B. Schardl. Detecting determinacy races in Cilk programs that use reducers. In preparation, 2014.

[9] Kerry Xing, I-Ting Angelina Lee, and Charles E. Leiserson. Cilkview-2: A scalability analyzer for nondeterministic parallel computations. In preparation, 2014.