Research Statement

Sam Westrick

In recent decades, architectural advances in multicore CPUs, GPUs, and specialized accelerators have brought parallelism to the mainstream. Many applications today rely critically on parallelism for performance. However, developing parallel software remains difficult, even for experts. Broadly speaking, the difficulty is that programmers have to account for low-level (often architecture-specific) details to ensure efficiency and scalability. High-level abstractions can help make parallel programming simpler and safer, but existing abstractions often come with a performance cost. As a result, programmers resort to writing low-level code, which is error-prone and can lead to subtle bugs.

To address the difficulty of parallel programming, my goal is to raise the level of abstraction at which programmers can achieve high performance. Specifically, my research focuses on **provably efficient implementations** of high-level languages, libraries, and systems. Such implementations use a variety of techniques (e.g., compilation, code generation, automatic memory management, dynamic scheduling, etc.) to automatically manage low-level issues and provide the programmer with provable guarantees on safety and performance. My aim is to make fundamental advances in the design of new implementation techniques for provable efficiency. Utilizing these advances, I aim to build new languages, libraries, and systems which make it simpler and safer to develop high-performance parallel software.

My work has already made progress on two long-standing problems in this area: the performance of parallel functional languages [8, 19, 5, 13, 6, 12], and the granularity control problem [14]. I describe these lines of work in detail below. I also have developed provably efficient techniques for data-parallel programming [17] and have done significant work in the area of dynamic scheduling [5, 21, 10, 15]. Schedulers play a critical role in achieving provable efficiency by providing direct control over key performance properties. I have helped develop schedulers which guarantee high utilization combined with space-efficiency [5], elasticity [21], and fairness [10]. Recently, I have been working on scheduling for heterogeneous CPU-GPU architectures, with the aim of automatically ensuring high utilization across different combinations of CPUs and GPUs [15].

In addition to my work on provably efficient implementations, I am interested generally in the areas of programming languages, algorithms, and systems. I have worked on a variety of topics, including parallel algorithms for dynamic trees and order maintenance [3, 2, 18], architectural support for parallel programs [20], and static verification of parallel programs [9]. I also have experience with specific applications, especially parallel simulation of quantum circuits [16] and scalable quantum circuit optimization [7].

My work has appeared at POPL, PLDI, PPoPP, ICFP, ESA, SPAA, and CGO, with two distinguished papers [5, 13]; in addition, my dissertation [12] on the topic of efficient and scalable parallel functional programming received a dissertation award from ACM SIGPLAN [11]. A significant contribution of my PhD was the core implementation of **MPL** [4], an open-source compiler and run-time system with provably efficient automatic memory management and scheduling. At Carnegie Mellon University, MPL is used to help teach parallel algorithms to over 500 undergraduate students each year. Students use MPL to implement sophisticated parallel algorithms which perform well, and are able to do so within a few weeks, with no prior parallelism experience. Additionally, our research has shown that MPL is able to compete with low-level programming techniques in terms of efficiency and scalability [6].

Moving forward, a number of challenges remain. High-level parallel languages today can provide limited guarantees on safety and performance, but new implementation techniques are needed to effectively manage

low-level factors. In the remainder of this statement, I describe two of my active lines of research, motivate specific challenges that remain to be solved, and finally discuss my future research plans, working towards the goal of making it easier to write and maintain parallel software.

Efficient and Scalable Parallel Functional Programming

Researchers have argued for decades that functional programming techniques can make parallel programming simpler and safer. Functional programming languages provide control over side-effects, enabling programmers to easily develop programs with no race conditions or subtle concurrency bugs. Additionally, functional languages support higher-order functions (e.g., `map`, `reduce`, and `filter` on collections of data) which can be used to express parallel algorithms elegantly and succinctly. However, parallel functional languages historically have underperformed in comparison to their imperative and procedural counterparts. The reason is that functional languages typically allocate data at a high rate, and this rate only increases with parallelism, causing existing memory management techniques to suffer high overheads under the increased pressure.

In this line of work [8, 19, 5, 13, 6, 12, 9], we developed new implementation techniques for parallel functional languages and demonstrated that these languages can deliver the same efficiency and scalability as lower-level languages. The key is a memory property called **disentanglement**, which limits communication between concurrent tasks and makes it possible for individual tasks to perform garbage collection independently, in parallel, with nearly no additional synchronization. We identified this disentanglement property and proved that it is guaranteed in race-free programs. This result implies that disentanglement is directly applicable to parallel functional programs, which are race-free by default. More generally, we empirically showed that disentanglement is overwhelmingly common, even when concurrent data structures (e.g., lock-free hash tables) are used under the hood for improved efficiency.

Utilizing disentanglement, we designed and implemented a provably efficient memory manager for a parallel version of the Standard ML language. Our implementation, called MPL [4], consists of a compiler and run-time system which work together to translate high-level parallel functional programs into executables with excellent multicore performance. Across a wide range of benchmarks [1], we have shown that MPL can outperform industry-standard memory-managed languages (such as Java and Go) and can compete with the performance of hand-optimized code written in low-level languages such as `C/C++`.

Automatic Parallelism Management and Granularity Control

Many task-parallel languages support *fine-grained parallelism* where the programmer can safely spawn millions of small tasks per second without harming performance. However, even fine-grained parallelism has limits. If the programmer is not careful, they might accidentally spawn too many tasks and cause overall performance to degrade significantly, with overheads in practice of as much as 10-100x.

This is broadly known as the *granularity control problem*, where the *granularity* of tasks refers to their "size", i.e., the amount of work each task performs. Programmers are expected to manually control granularity, to limit the overheads of task creation and management. However, this poses a number of problems: (1) manual granularity control is error-prone and time-consuming for the programmer; (2) granularities can depend on architecture-specific factors and therefore may not be portable across different machines; (3) manual granularity control is largely incompatible with high-level programming techniques, especially higher-order parallel functions, where the "correct" granularity might depend upon function arguments. For example, in the code `map(f,A)` which applies a function `f` in parallel across an array `A`, the "correct" granularity for the `map` depends on the complexity of the function `f`.

My recent work [14] makes progress on this granularity control problem. In our approach, the programmer does not control granularity; instead, the programmer expresses all *opportunities* for parallelism, and relies on the language implementation to figure out exactly when and where to spawn tasks. Our primary

contribution is a new implementation technique which avoids the cost of task creation by default. Specifically, our technique embeds each opportunity for parallelism directly into the call-stack. Then, during execution, a dynamic scheduler inspects the call-stacks and judiciously exposes only as much parallelism as necessary to ensure efficiency and scalability. In this way, the compiler and run-time system work together to **automatically manage parallelism**, providing the programmer with provable guarantees on efficiency and scalability.

We implemented this approach by extending the MPL compiler [4], and found that the approach is effective in practice. Existing language implementations do not perform well on programs without manual granularity control; in contrast, our approach is able to execute these programs with low overhead (less than 2x on average) and good scalability. Furthermore, we show that the technique reduces the performance benefits of manual granularity control significantly, making progress towards eliminating the need for manual granularity control in general.

Future Work

To make high-performance parallel programming simpler, safer, and more efficient, a number of challenges remain to be solved. Here I outline a few specific challenges that I plan to address in my research.

Closing the performance gap. Although high-level languages are simpler and safer to use than low-level languages, the reality is that this simplicity and safety often comes with a performance cost. That is, in the current state of the art, there is an apparent performance gap between high-level and low-level languages. I believe this gap is not fundamental, but rather a reflection of the limitations of current implementation techniques. In my future work, I aim to develop the necessary techniques to close the gap.

One factor that contributes to the gap has become clear: in high-level languages with automatic memory management, programmers do not have sufficient control over memory representation and layout. This can impact data locality and lead to significant overheads in terms of time and space usage. For example, the most natural representation of a collection of 2-dimensional geometric points might be an array of tuples/structs, but many languages will choose to store the elements (the tuples) indirectly through pointers, causing additional cache misses. It is sometimes possible to work around these issues by rewriting the program, but this is tedious and defeats the purpose of high-level language features.

I believe this particular problem can be solved while retaining the important safety and performance guarantees of automatic memory management, and I intend to tackle this challenge in future work. More generally, I plan to identify and address other sources of overhead which contribute to the performance gap between high-level and low-level languages.

Managing concurrency bugs. It is notoriously difficult for programmers to identify and fix concurrency bugs, especially *data races*, which are common and problematic. Roughly speaking, the problem is that if programmers are not careful to "properly synchronize" concurrent accesses to shared memory locations, then many compilers and run-time systems are unable to provide *any* guarantees. The behavior of the system can then become completely unpredictable; for example, the program could unexpectedly crash, or worse, the program could silently continue with corrupted data and have harmful effects. Programmers therefore go to great lengths to ensure that their programs are free of data races.

Ideally, a language implementation should be able to prevent data races entirely, but it is not clear how to accomplish this goal. Complex type systems (e.g. based on ownership and borrowing) can help avoid data races, but such type systems are highly restrictive and escape hatches are prevalent (for example, in the Rust language, programmers are able to circumvent the type system with code marked `unsafe`). Purely functional programming is another avenue towards a potential solution, but the problem with purely

functional programming is that it disallows in-place updates, which are an essential ingredient of many high-performance data structures and algorithms. Data races can also be prevented using *atomic* operations, but these operations incur additional overhead at run-time (e.g., to execute memory fences), which can degrade performance.

It is important to solve this problem, because data races undermine one of the primary goals of high-level languages: provable safety. I believe this problem can be solved, and that a solution will likely involve a combination of both static (i.e., compile-time) and dynamic (i.e., run-time) techniques.

Programming heterogeneous hardware. Modern architectures consist of many heterogeneous components which are specialized for different tasks, and there appears to be a trend towards greater specialization and heterogeneity. When targeting heterogeneous hardware, programmers face not only the challenges of parallel programming, but also issues of scheduling data and computation onto processors with different performance characteristics. It is difficult to determine an efficient mapping, in part because the mapping might depend upon dynamic factors such as the current load of the system. Additionally, after tuning software to perform well on a particular architecture, it is unlikely that the same software will also perform well on a different architecture, and therefore the software may need to be retuned and/or rewritten to run on a different machine or cluster.

In recent work, I have been developing new programming and scheduling techniques to automatically map computations onto heterogeneous CPU-GPU hardware, with the goal of guaranteeing high utilization, automatically, across different combinations of CPUs and GPUs [15]. This work shows promising results; in future work, I plan to continue this line of work, and more generally to develop new techniques which simplify the challenge of programming heterogeneous devices.

References

[1] The parallel ML benchmark suite. `https://github.com/MPLLang/parallel-ml-bench`, 2020-2023.

[2] Umut A. Acar, Vitaly Aksenov, and Sam Westrick. Brief announcement: Parallel dynamic tree contraction via self-adjusting computation. In Christian Scheideler and Mohammad Taghi Hajiaghayi, editors, *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 275–277. ACM, 2017.

[3] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. Parallel batch-dynamic trees via change propagation. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 2:1–2:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[4] Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. MPL: A high-performance compiler and run-time system for parallel ML. `https://github.com/MPLLang/mpl`, 2020-2023.

[5] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[6] Jatin Arora, Sam Westrick, and Umut A. Acar. Efficient parallel functional programming with effects. *Proc. ACM Program. Lang.*, 7(PLDI):1558–1583, 2023.

[7] Jatin Arora, Sam Westrick, Dantong Li, Yongshan Ding, and Umut A. Acar. Efficient optimization of quantum circuits via local optimality. Submitted for publication and under review.

[8] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 81–93, 2018.

[9] Alexandre Moine, Sam Westrick, and Stephanie Balzer. Dislog: A separation logic for disentanglement. In *Proceedings of the 51st Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2024.

[10] Stefan K. Muller, Sam Westrick, and Umut A. Acar. Fairness in responsive parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2019, 2019.

[11] ACM SIGPLAN John C. Reynolds Dissertation Award. `https://sigplan.org/Awards/Dissertation/`, 2023.

[12] Sam Westrick. *Efficient and Scalable Parallel Functional Programming Through Disentanglement*. PhD thesis, Carnegie Mellon University, August 2022.

[13] Sam Westrick, Jatin Arora, and Umut A. Acar. Entanglement detection with near-zero cost. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2022, 2022.

[14] Sam Westrick, Matthew Fluet, Mike Rainey, and Umut A. Acar. Automatic parallelism management. In *Proceedings of the 51st Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2024.

[15] Sam Westrick, Troels Henriksen, Sanil Rao, and Umut A. Acar. Hybrid CPU-GPU task parallelism. Submitted for publication and under review.

[16] Sam Westrick, Byeongjee Kang, Mike Rainey, Colin McDonald, Pengyu Liu, Mingkuan Xu, Jatin Arora, Yongshan Ding, and Umut A. Acar. GraFeyn: Efficient sparse-aware simulation of quantum circuits. Paper in preparation.

[17] Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. Parallel block-delayed sequences. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 61–75. ACM, 2022.

[18] Sam Westrick, Larry Wang, and Umut A. Acar. DePa: Simple, provably efficient, and practical order maintenance for task parallelism. *CoRR*, abs/2204.14168, 2022.

[19] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2020.

[20] Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut A. Acar, Peter A. Dinda, and Nikos Hardavellas. WARDen: Specializing cache coherence for high-level parallel languages. In Christophe Dubach, Derek Bruening, and Ben Hardekopf, editors, *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2023, Montréal, QC, Canada, 25 February 2023- 1 March 2023*, pages 122–135. ACM, 2023.

[21] Yue Yao, Sam Westrick, Mike Rainey, and Umut A. Acar. Elastic task scheduling. Paper in preparation.