Brief Announcement: ParlayLib – A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Daniel Anderson
Carnegie Mellon University
dlanders@cs.cmu.edu

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Abstract

ParlayLib is a C++ library for developing efficient parallel algorithms and software on shared-memory multicore machines. It provides additional tools and primitives that go beyond what is available in the C++ standard library, and simplifies the task of programming provably efficient and scalable parallel algorithms. It consists of a sequence data type (analogous to std::vector), many parallel routines and algorithms, a work-stealing scheduler to support nested parallelism, and a scalable memory allocator. It has been developed over a period of seven years and used in a variety of software including the PBBS benchmark suite, the Ligra, Julienne, and Aspen graph processing frameworks, the Graph Based Benchmark Suite, and the PAM library for parallel balanced binary search trees, and an implementation of the TPC-H benchmark suite.

ACM Reference Format:
Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. Brief Announcement: ParlayLib – A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400254

1 Introduction

Writing correct and efficient parallel algorithms is a challenging task, even for parallel programming experts. We have designed ParlayLib over a number of years to simplify programming parallel algorithms by supplying a variety of parallel primitives and functionality that has greatly helped us develop correct and fast code. In this short paper, we describe the library and compare it with existing C++ libraries in terms of both functionality and performance. ParlayLib is designed to make extensive use of modern features of C++ including lambdas and C++ threads. The library is used across a range of projects in our own and affiliated research groups, and is currently being used as part of ongoing work on shared-memory parallel graph algorithms in an industrial research lab. The library is portable, running to a few thousand lines of header files. The library can be found at https://github.com/cmuparlay/parlaylib.

Parallel Tools for C++. Over the years, many parallel libraries and extensions have been developed for C++. The Cilk environment supplies constructs for nested fork-join parallelism and an efficient scheduler [2]. The OpenMP API also supplies similar constructs as Cilk. It performs exceptionally well at processing regular loops, but has been observed to struggle with irregular loops and nested parallelism. Intel's Threading Building Blocks (TBB) is a library supplying a variety of functionality including a scheduler, a pool-based memory allocator, and a set of concurrent data structures. Recently, the C++ community has proposed a standardized set of parallel extensions to the C++ Standard Library [8]. These extensions make it easier for developers already using the Standard Library to transition to and integrate parallelism. Intel has developed an implementation of the proposal, called ParallelSTL, which builds on top of TBB. These tools all have desirable properties and support a very similar interface. They all support task parallelism via some form of forking, joining, and parallel loops, scheduled on worker threads with bounded id numbers. However, their support for a wide range of useful parallel operations outside of these fundamental primitives is lacking.

ParlayLib. ParlayLib provides additional primitives, algorithms, and data structures that go beyond what is offered by existing parallel programming tools, with an emphasis on performance, provable-efficiency, and scalability. The library supports parallel algorithms ranging from simple primitives such as map, filter, prefix sum (scan), and reduce, to sophisticated and work-efficient implementations of histograms, generating random permutations, integer sorting, suffix arrays, and many others. It also implements a broad subset of the algorithms in the C++ Extensions for Parallelism proposal [8]. The algorithms in ParlayLib are written in terms of scheduler-agnostic parallel primitives, and so the primitives provided by ParlayLib can be used in conjunction with Cilk, OpenMP, TBB (using their primitives for parallelism), or its own custom work-stealing scheduler written using standard C++ threads, with no additional dependencies.

At the level above the scheduler, the library has two basic constructs for parallelism: parallel_for(start, end, f) applies the function f (typically a lambda) to the integers from start (inclusive) to end (exclusive) in parallel, and parallel_do(f_1, f_2) runs the functions f_1 and f_2 in parallel. If run on Cilk, OpenMP, or TBB these primitives convert directly to their equivalents in these systems.

The parallel algorithms implemented in ParlayLib build on these basic parallel idioms, using a purely-functional methodology whenever possible. For example, most of our sequence algorithms do not mutate the input array, and return a new result. In our experience, this style prevents a large class of bugs, and leaves in-place optimizations as a special case to be used at the discretion of the programmer, and not as a default option.

Contributions. This paper introduces ParlayLib, a highly scalable toolkit of parallel primitives and datatypes. Specifically, it provides the following features:

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400254

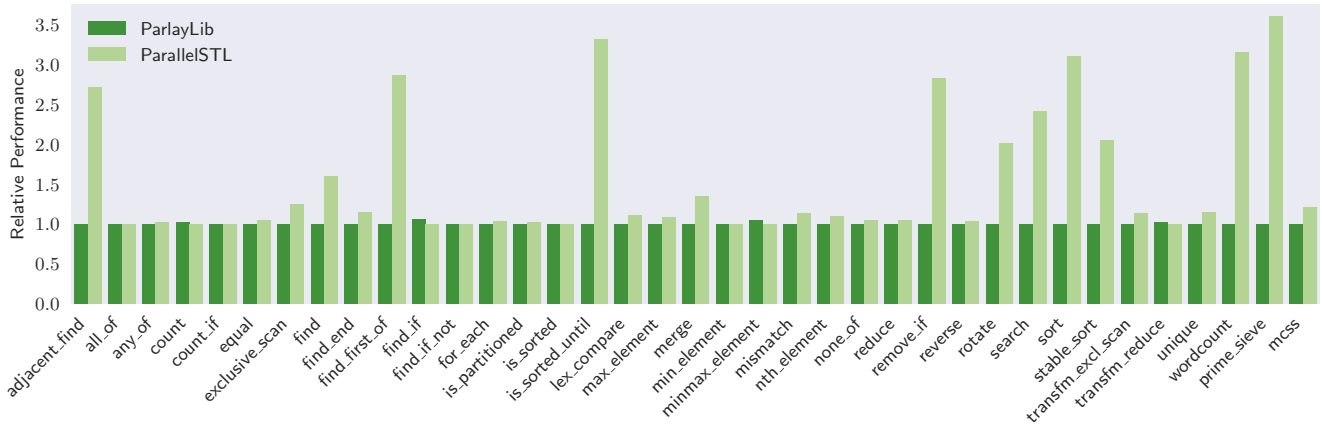

Figure 1: Relative performance of ParlayLib vs. ParallelSTL on a 72-core machine with 2-way hyper-threading enabled. All benchmarks are run on sequences of length 10^8 containing 8-byte elements.

(**1**) Parallel collection datatypes, including sequences, delayed (or lazy) sequences, parallel hash tables, and parallel bags.
(**2**) Header-only implementations of a work-stealing scheduler and a scalable memory allocator.
(**3**) A collection of provably-efficient parallel primitives for sequences with low work and depth.
(**4**) Implementations of most of the algorithms in the C++ Extensions for Parallelism proposal that achieve a 1.43x speedup on average over Intel's publicly available implementation of the proposal across a broad set of the implemented primitives.

2 ParlayLib Overview

Next, we give an overview of ParlayLib. We defer a full description of our library to our repository, where we describe the interfaces and cost bounds for our primitives and datatypes.

Sequences. One of the main datatypes of ParlayLib is a generic sequence data type. It can be thought of as a parallel version of C++'s vector, but has many features designed to make it better suited for parallelism. Importantly, unlike vectors, it allows parallel initialization, destruction, copying, and resizing. Sequences can also be used as a replacement for strings, with the added benefit that they support efficiently manipulating the underlying sequence of characters in parallel. In addition, ParlayLib implements the "small string optimization", which is important for high performance on a large amount of real-world code that processes strings.

Delayed Sequences. We provide a lazy sequence datatype that we refer to as a ***delayed sequence***. A delayed sequence avoids storing the elements of the sequence in memory in cases where the sequence can be succinctly represented by a function of the index. There are numerous advantages of employing delayed sequences, since they permit us to fuse parallel operations such as maps and reductions without storing intermediate results in memory. The sequence primitives in ParlayLib automatically work on both sequences and delayed sequences. This is an attractive feature of ParlayLib, since we can avoid the hassle of creating a _transform version of each primitive that takes an extra map argument, which significantly clutters the C++ Standard Library interfaces.

Other Parallel Datatypes. In addition to the sequence datatypes above, we provide an implementation of a phase-concurrent parallel hash table due to Shun and Blelloch [10], as well as specializations of the table to unordered sets. We note that the PAM library for trees [12], and the GBBS benchmark [5] for graphs which builds on the Ligra framework [9], and the Aspen framework for streaming graph processing [6] have been built as libraries on top of ParlayLib. Users desiring either tree or graph datatypes can easily integrate both ParlayLib and the desired package into their work.

Sequence Primitives. We provide fundamental primitives that work on sequences, or any type with random access iterators (including vectors, or arrays). This includes equivalents to most C++ Standard Library routines. In addition to implementations of these standard primitives, we provide generic implementations of many parallel sequence primitives not supported by the standard library, including pack_index, random_shuffle, histogram, integer_sort, counting_sort, remove_duplicates, collect_reduce, and semisort [7], amongst others.

All sequence primitives apply to both regular and delayed sequences. Furthermore, our primitives all have strong provable bounds on their work and depth, assuming the algorithms are scheduled on an efficient scheduler [1, 3]. Having cost bounds for our primitives enables higher-level libraries to derive strong provable bounds on the work and depth of their algorithms by composing the costs of the primitives in ParlayLib.

As an example, we show the actual C++ code for a prime sieve implementation using ParlayLib in Algorithm 1. The algorithm uses a number of primitives from ParlayLib. It initializes a boolean sequence, flags, in parallel on Line 8. The algorithm uses a parallel_for on Lines 11–18 to loop over the primes found in the recursive call in parallel, and for each prime, spawns a nested parallel_for (Lines 14–17) to mark all multiples of this prime as false in the flags array. The last argument to the parallel_for command used on Lines 17 and 18 is an optional *granularity* parameter specifying the minimum number of loop iterations that must be contained within a parallel task that is spawned. Note that the inner loop uses a higher granularity since the work it performs is regular, and the outer loop uses a small granularity since each outer loop iteration contains a variable amount of work. Our code is simple and as we discuss later, performs significantly better than the same

algorithm implemented in ParallelSTL, likely due to challenges with handling nested parallelism in the TBB scheduler.

2.1 Benchmarks

Comparison vs. Parallel STL. We compared our implementation of several algorithms in the C++ Extensions for Parallelism proposal [8] to Intel's implementation (ParallelSTL). We ran our experiments on a 72-core machine with 2-way hyper-threading enabled, and show the results of the comparison in Figure 1. We observed that our code achieves an average speedup of 1.43x over ParallelSTL across the standard library benchmarks. We observe that although both systems achieve similar performance on many benchmarks, ParlayLib consistently produces the fastest algorithms for important primitives such as sorting, scans, and reductions. We note that the superior performance of our *find* algorithms (e.g. adjacent_find, find_first_of) is attributable to the fact that they perform work proportional to the position of the target element, whereas ParallelSTL's implementation always performs linear work.

Additional Benchmarks. We also benchmarked four applications: word count (WC), maximum contiguous subsequence sum (MCSS), a prime sieve up to n (Primes), and numerical integration (Integrate). As far as possible, these applications use equivalent primitives in both ParlayLib (see Algorithm 2) and ParallelSTL. Our algorithm creates a delayed sequence on Line 4–6 by providing (i) the size of the sequence and (ii) a lambda indicating the value stored at the i'th index. For Integrate, ParallelSTL does not provide delayed sequences and thus cannot implement the memory-efficient algorithm using delayed sequences. Instead, it must either create a sequence of samples, or a sequence of indices and use the transform_reduce primitive (we report results for the latter, since for 10^8 samples we found that it is almost twice as fast as the former).

We show the results for WC, MCSS, and Primes in Figure 1. For these benchmarks, ParlayLib is 2.65x faster on average compared to ParallelSTL, which could be due to better support for nested parallelism in our scheduler for Primes, and using delayed sequences in WC. Finally, for Integrate, ParlayLib is *28x faster* than ParallelSTL due to the fact that ParallelSTL must allocate and initialize an array before performing the reduction, whereas ParlayLib can simply perform a reduction over a delayed sequence.

3 Conclusion

We have described the ParlayLib toolkit for parallel programming, which supplies fundamental parallel data structures, algorithms, primitives, and other utilities that are useful when implementing scalable and provably-efficient parallel algorithms on shared-memory multicore machines. The library is already widely used within other parallel algorithms research, including the Ligra [9] and Julienne [4] graph processing frameworks, the Graph Based Benchmark Suite (GBBS) [5], the Parallel Augmented Maps (PAM) library [12], an implementation of the TPC-H benchmark [11] and the graph-streaming system Aspen [6]. We invite contributions of new primitives and algorithms to the library.

Acknowledgements

Thanks to Daniel Ferizovic, Julian Shun, and Yihan Sun for their help with the library. This research was supported by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223.

Algorithm 1 Computing Primes in ParlayLib

```
1  template <typename Int>
2  parlay::sequence<Int> prime_sieve(Int n) {
3    if (n < 2) return parlay::sequence<Int>();
4    Int sqrt = std::sqrt(n);
5    // recursive call
6    auto primes_sqrt = prime_sieve(sqrt);
7    // flags to mark primes
8    parlay::sequence<bool> flags(n+1, true);
9    // 0 and 1 are not prime
10   flags[0] = flags[1] = false;
11   parlay::parallel_for(0,  primes_sqrt.size(),
12     [&] (size_t i) {
13       Int prime = primes_sqrt[i];
14       parlay::parallel_for(2, n/prime + 1,
15         [&] (size_t j) {
16           flags[prime * j] = false;
17         }, 1000);
18   }, 1);
19   // returns indices of the primes
20   return parlay::pack_index<Int>(flags);
21 }
```

Algorithm 2 Numerical Integration ParlayLib

```
1 template <typename F>
2 double integrate(size_t n, double a, double b, F f) {
3   double delta = (b-a)/n;
4   auto samples = parlay::delayed_seq<double>(n,
5     [&] (size_t i) {
6       return f(a + delta/2 + i * delta); });
7   return delta * parlay::reduce(samples);
8 }
```

References

[1] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[3] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[4] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[5] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[6] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[7] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[8] Switzerland International Organization for Standardization, Geneva. 2018. ISO/IEC TS 19570:2018: Programming Languages – Technical Specification for C++ Extensions for Parallelism. https://www.iso.org/standard/70588.html.

[9] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[10] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[11] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB* 13, 2 (2019), 211–225.

[12] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.