Teaching Parallel Algorithms Using the Binary-Forking Model

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

Yan Gu
University of California, Riverside
ygu@cs.ucr.edu

Yihan Sun
University of California, Riverside
yihans@cs.ucr.edu

***Abstract*—In this paper, we share our experience in teaching parallel algorithms with the binary-forking model. With hardware advances, multicore computers are now ubiquitous. This has created a substantial demand in both research and industry to harness the capabilities of parallel computing. It is thus important to incorporate parallelism in computer science education, especially in the early stages of the curriculum. However, it is commonly believed that understanding and using parallelism requires a deep understanding of computer systems and architecture, which complicates introducing parallelism to young students and non-experts.**

We propose to use the *binary-forking model* in teaching parallel algorithms, proposed by our previous research work. This model is meant to capture the performance of algorithms on modern multicore shared-memory machines, which is a simple abstraction to isolate algorithm design ideas with system-level details. The abstraction allows for simple analysis based on the work-span model in theory, and can be directly implemented as parallel programs in practice. In this paper, we briefly overview some basic primitives in this model, and provide a list of algorithms that we believe are well-suited in parallel algorithm courses.

I. Introduction

With the advent of Intel's dual-core E6320, and AMD's dual-core Athlon 64 in 2005, parallelism moved into the mainstream of processor design. Today, a typical laptop, desktop, or server has 4 to 200 cores that each can be hyper-threaded. Hardware advances have created a substantial demand in both research and industry to harness the capabilities of parallel computing. Take the June 2023 issue of VLDB as an example, 9 out of 23 papers incorporate multithreading in their implementation, regardless of whether their primary focus is on parallelism. Despite the excitement generated by parallelism, there is still a huge gap between the growing demand and the delay in educating students on how to effectively utilize parallelism. Most CS curriculums, especially in the fundamental courses on algorithms and programming, still focus on the sequential setting. Usually, parallelism is not introduced in the curriculum until upper-division or graduate-level classes, and is usually not part of the required courses.

Indeed, parallel programming is notoriously hard and is believed to involve a deep understanding of systems and architecture. This makes parallel algorithm design and programming much harder than the sequential setting in education: sequentially, the RAM (random access machine) model bridges algorithm design/analysis with programming, and isolates system-level details. Students learn algorithms and understand their efficiency with the intuitive measure of "time complexity". With these tools, it is not too hard to teach middle school students, or non-CS audiences, to design simple algorithms and write codes. In parallel, however, both theory and programming *seem* to be much harder.

One major hurdle in introducing parallelism to young students or non-experts with limited knowledge of computer systems is the absence of the "RAM" model in parallel contexts. This is *not* due to a lack of theoretical models in parallel settings; in fact, there is a rich history of the study of parallel computational models (and algorithms on them). For example, one of the most influential parallel models, the Parallel Random Access Machine (PRAM) model [65], was proposed in the 1980s. During the 80s and early 90s, there were hundreds of papers and several textbooks and survey articles [8, 52, 54, 61] for parallel algorithms on PRAM; many ideas are still relevant today. We refer the audience to the original paper [23] for a more detailed literature review.

Unfortunately, directly using the PRAM in teaching parallel algorithms and programming has several major challenges. First, the PRAM model assumes that P threads share the memory and run in parallel in lockstep, which contradicts the inherent nature of today's machines—they are highly asynchronous. Second, due to sharing with other jobs, the number of processors available can change over time, and is not a fixed number P. For these reasons, and more discussed in [21], existing parallel programming languages usually assume dynamic creation of threads, a dynamic scheduler to map threads to processors, and no assumption of tight synchronization. The model is the so-called fork-join / nested-parallelism model. It has also been used in research papers, but oftentimes, the results on fork-join models are reinterpreted on the PRAM to make fair comparisons with earlier results. Many of them assume a constant cost to fork an arbitrary number of threads, as is in PRAM. As a result, many strong theoretical results rely on utilizing tight synchronization, and may need extra engineering effort to achieve high performance on today's multicore machines. Such an issue complicates the way to teach these algorithms. An instructor usually needs to cover both the analysis and extra techniques to program them.

Motivated by these issues, the authors (along with others) formalized the *parallel binary-forking model* in 2020 [21]. The model is meant to capture the performance of algorithms on modern multicore shared-memory machines in a simpler yet more accurate way, such that algorithms with good theoretical guarantees can directly lead to high performance. For teaching purposes, we usually use the *binary fork-join model* formalized in that paper, allowing for certain atomic operations such as compare-and-swap. In this model, a computation starts with one thread. Each thread acts like a regular RAM with two additional operations: a fork operation that dynamically creates two child threads to run in parallel, and a join to synchronize the previously forked child threads, after which

the parent thread continues. As we will discuss in Section II, both asynchrony and binary-forking (as opposed to arbitrary-way forking) are important to make the model practical. Such assumptions closely resemble how a multicore algorithm is actually being executed by state-of-the-art schedulers, and can be easily implemented by most programming languages. With this model, students can learn and design algorithms similar to the sequential approach. The additional detail to specify is that *the following two tasks can run in parallel*. Theoretical analysis can be done using the standard work-span analysis (see details in Section II).

In this paper, we will discuss how this model is especially suitable for classes (even for entry-level classes). In fact, we have taught parallel algorithms on this model in various classes, including a sophomore-level course at CMU (15-210) and an intermediate-level undergraduate algorithm course at UCR (CS 141), both of which are required in the corresponding CS program. This model has also been taught in elective courses at both undergraduate and graduate levels, including CMU 15-853, UCR CS142 and CS214, UMD CMSC858N, and MIT 6.886. We have also used the model to give overview talks to K-12 students to introduce parallel algorithms.

In the following, we start by introducing the model, and discuss the advantages of using this model in teaching parallel algorithms. Section III reviews some simple building blocks in this model, which we believe fit in a one-week overview of parallel algorithms in classes. In Section IV, we further select more advanced algorithms that can be covered in a complete course about parallelism. Most of them have strong theoretical guarantees and available open-source code.

II. The Model

In this section, we introduce the binary fork-join model in [21], which we consider to be well-suited for teaching parallel algorithms. This model assumes a collection of threads that share memory. The threads can be created dynamically and can run asynchronously in parallel. A computation starts with one thread. Each thread acts like a standard Random-Access Machine (RAM) with two additional operations: fork and join. The fork operation creates two child threads to work in parallel. The join operation synchronizes the two child threads when they complete, after which the parent thread continues. A parallel for-loop can be simulated by using fork for a logarithmic number of levels. Costs are measured in *work* (the total number of instructions executed among all threads) and *span* (the longest sequence of dependent instructions). An algorithm is *work-efficient* if its work is asymptotically the same as the best sequential algorithm.

Variants of this model have been widely studied before we formalized the discussion [2, 3, 9, 11–15, 17, 24, 25, 29, 31–33, 40, 72]. The fork-join paradigm is also widely used in practice, and supported by programming systems such as Cilk [44], the Java fork-join framework [53], X10 [28], Habanero [27], Intel Threading Building Blocks (TBB) [51], and the Microsoft Task Parallel Library [73]. This model is recently added in the textbook *Introduction to Algorithms* [33]

```
1 int reduce(A[1..n]) {
2   if (n = 1) return A[1];
3   parallel_do {
4     x = reduce(A[1..n/2]);
5     y = reduce(A[(n/2+1)..n]); }
6   return x + y;
7 }
```

(a) Pseudocode for parallel reduce

```
1 int reduce(int* A, int n) {
2   if (n = 1) return A[0];
3   int x, y;
4   cilk_scope {
5     x = cilk_spawn reduce(A, n/2);
6     y = reduce(A + n/2, n - n/2); }
7   return x + y; }
```

(b) Implementation in OpenCilk [62]

Figure 1: The reduce primitive and its implementation.

(3rd and 4th editions) when discussing parallel algorithms as an optional, advanced topic to cover in algorithm courses.

For simplicity, we can write (pseudo) code in this model by simply using two more keywords in the language: a parallel_do that specifies the next two statements can run in parallel, and a parallel_for that allows all iterations in a for-loop to run in parallel. A parallel_do means to wrap the two statements with a fork-join pair. parallel_for is essentially simulated by using logarithmic levels of forking and joining them back at the end. To design a parallel algorithm, one can just identify the independent components in the algorithm, and specify that they can run in parallel.

As an example, consider the reduce operation that computes the sum of all values in an array (see Figure 1). A simple solution is to consider a divide-and-conquer scheme, which splits the array into two halves, computes the sum of the two halves in parallel, and when they both finish, returns the sum of the two recursive calls. We present the pseudocode in Figure 1. The algorithm is as straightforward as a sequential divide-and-conquer approach, with the key observation being that the two recursive calls can be executed in parallel. The students will be told that the actual execution will be finished by a scheduler to map each task to a processor. The number of available processors does not need to be specified.

In general, we require the algorithms to be race-free, i.e., if two logically parallel operations can access the same memory location, both of them must be read. When it is necessary to use concurrent writes, atomic operations will be introduced. In the model used in class, we assume unit-cost atomic operation compare-and-swap, which is supported in modern architecture.

There are several reasons that we recommend to use this model in classes. First of all, the model assumes the threads to run asynchronously, and allows threads (tasks) to be created dynamically by the fork operation. This closely resembles how a modern machine executes parallel code. Asynchrony is important because the processors (cores) on modern machines are themselves highly asynchronous, due to varying delays from cache misses, processor pipelines, branch prediction, hyper-threading, changing clock speeds, interrupts, the operating system scheduler, and several other factors.

Second, most programming languages support fork-join

semantics. Still take the reduce function as an example, the actual C++ code is very intuitive and similar to the pseudocode. Figure 1 also shows the code in OpenCilk [62]. OpenCilk (and many other parallel libraries and programming languages) directly support the parallel_do semantics. It is easy for students to translate the algorithmic ideas learned in class into parallel programs. There is only one additional implementation technique that needs to be mentioned, which is granularity control. In particular, to achieve high performance, the base case size needs to be set to a larger value, instead of 1. This is a common practice in parallel programming, aiming to avoid having too small parallel tasks such that the work to manage the threads is more expensive than the computation.

Third, the model allows simple theoretical analysis of the algorithms based on the work-span model. The work of an algorithm can be viewed as the time complexity when the algorithm is executed sequentially, and the span is the number of dependent steps when an infinite number of processors are available. More concretely, given two statements with work W_1 and W_2, and span S_1 and S_2, we can calculate the work and the span of their sequential and parallel composition as [1]:

	Work	Span
Executed sequentially	$O(1) + W_1 + W_2$	$O(1) + S_1 + S_2$
Executed in parallel	$O(1) + W_1 + W_2$	$O(1) + \max(S_1, S_2)$

With this idea, one can simply compute the work and span for the reduce algorithm by recurrences:

$$W(n) = 2W(n/2) + O(1)$$
$$S(n) = S(n/2) + O(1)$$

which solves to $O(n)$ work and $O(\log n)$ span.

Finally, binary forking (as opposed to arbitrary-way forking) is important to the simulation and scheduling results [2, 5, 25]. Any computation in the binary-forking model that has W work and S span can be simulated (scheduled) on P loosely synchronous processors in $W/P + O(S)$ time with high probability in W [25]—similar bounds hold when the number of processors and their relative processing rates change over time. In class, students will understand how the theoretical analysis of W and S affects the actual running time. Modern multicore machines usually have up to thousands of processors. Therefore, for a parallel algorithm where $S \ll W$, the running time is usually dominated by the term W/P. Rather than prioritizing span optimization, which is common in most PRAM algorithms, a practical algorithm typically aims for minimal work while maintaining a satisfactory level of parallelism (e.g., polylogarithmic or $o(n)$ span).

In the following, we will present a list of simple algorithms in this model in section III, and then show more advanced topics in section IV. The algorithms in section III are well-suited for a one-week introduction to parallel algorithms, and those in section IV can be covered in a complete schedule on parallel algorithms in 8–16 weeks.

III. Building Blocks

We now present some simple example algorithms in this model (see Figure 2). For the algorithms below, we select the simplest algorithm with efficient work and low span, although they may not be the theoretically best in terms of span. All of the algorithms are important building blocks used in more advanced parallel algorithms (e.g., those in Section IV).

Scan (prefix sum). Given a sequence $A[1..n]$, its prefix sum is an array B such that $B[i] = \sum_{j=1}^{i} A[i]$. The algorithm creates a new array C by combining every two adjacent elements in A. It then computes the prefix sum of C and uses it to recover the prefix sum of A. The algorithm has $O(n)$ work and $O(\log^2 n)$ span. Algorithms with $O(\log n)$ span exist [7]. The code is slightly longer but also conceptually simple.

Pack (filter). Given a sequence A and a predicate function f that maps each element in A to a boolean value, the pack function packs all elements in A that satisfies $f(A[i])$ into a new array B. The algorithm stores a flag array for $f(A[i])$, and computes the prefix sum of it in array s. This array gives the index of each element in the output array. The algorithm has the same work and span as the scan algorithm used.

Partition. Given an sequence $A[1..n]$ and a pivot p, the partition algorithm reorders elements in A such that all $A[i] < p$ are on the left to all $A[i] \geq p$. We can compute the left part by calling the pack function with a predicator $f(x) = (x < p)$. The right part can be computed symmetrically.

Merge. The merge algorithm takes two sorted arrays A and B and merge them into a sorted array C. We can use a divide-and-conquer scheme, which selects the median in A and binary searches it in B. Then we can merge the left part in A with the left part in B, and similarly for the right parts. Both of the subproblems can be solved recursively in parallel. The algorithm has $O(n)$ work and $O(\log^2 n)$ span, where $n = |A| + |B|$. There also exists an algorithm with $O(n)$ work and $O(\log n)$ span [19].

Merge sort and quicksort. Based on the partition and merge subroutines above, we can implement merge sort and quicksort. Both of them are classic comparison sorts using divide-and-conquer. For both of them, the two subproblems are independent and can be handled in parallel, and therefore the key component is to parallelize the partition in quicksort, and merge in merge sort. Using the parallel partition and merge algorithms, both sorting algorithms have $O(n \log n)$ work and polylogarithmic span (with high probability for quicksort).

These algorithms can be covered in one-week introduction of parallel algorithms in other classes. We have included them in general algorithms classes (e.g., CMU 15-853: Algorithms in the Real World, UCR CS 141: Intermediate Data Structures and Algorithms, UCR CS142: Algorithm Engineering).

Next, we overview other algorithms that we recommend to be included in a complete course on parallel algorithms.

IV. Other Algorithms

To include more algorithms in a complete course on parallel algorithms, we carefully select the following algorithms that can be easily understood and directly compatible with the binary-forking model. Most of the algorithms are work-efficient (or off by at most a polylogarithmic factor) in theory,

```
int* scan(A[1..n]) {
  if (n is 0) return;
  if (n is odd) n = n+1;
  parallel_for (i = 0 to n/2)
    C[i] = A[2i] + A[2i+1];
  D = scan(C, n/2);
  B[1] = A[1];
  parallel_for (i = 2 to n)
    if (i is even) B[i] = D[i/2];
    else B[i] = D[i/2] + A[i];
  return B; }
```

```
int* pack(A[1..n], f) {
  parallel_for (i = 1 to n)
    flag[i] = f(A[i]);
  s = scan(flag, n);
  parallel_for(i = 1 to n)
    if (f(A[i])) B[s[i]] = A[i];
  return B; }
```

```
int* merge(A[1..N], B[1..M]) {
  n = N/2;
  m = binary_search(B[1..M], A[n]);
  parallel_do {
    C[1..(n+m)] = merge(A[1..n], B[1..m]);
    C[(n+m+1)..(N+M)] = merge(A[(n+1)..N], B[(m+1)..M]);}
  return C; }
```

```
int* partition(A[1..n], p) {
  let function f(x) = (x<p);
  L = pack(A[1..n], f); // left side
  ... //symmetric for the right half
  Concatenate L and R into A;
  return size(L); }
```

```
int* mergesort(A[1..n]) {
  parallel_do {
    L = mergesort(A[1..n/2]);
    R = mergesort(A[(n/2+1)..n]);  }
  return merge(L[], R[]); }
```

```
int* quicksort(A[1..n]) {
  p = random(1, n);
  m = partition(A[1..n], A[p]);
  parallel_do {
    quicksort(A[1..m]);
    quicksort(A[(m+1)..n]);  } }
```

Figure 2: Examples of algorithmic building blocks in the binary-forking model.

and are implementable with high parallelism in practice. Almost all of them have open-source code available.

The following topics can be covered in 8–16 weeks.

Other Algorithmic Building Blocks

- The list ranking algorithm in [69] (or [21]). List ranking is a fundamental building block that can be considered as prefix sum but on link lists. They are widely used in parallel algorithms for trees and graphs listed below.
- Random permutation (Knuth's shuffle) in [21, 69].
- Sorting: samplesort [14], integer sort [43], semisort [47].

Data Structures

- The batch-parallel hash table in [68].
- The join-based balanced binary tree in [18, 39, 71].
- The concurrent linked list in [57].

Graph Algorithms

- Breadth-first search (BFS) in [6, 67]. It is a simple combination of the building blocks set up earlier.
- The maximal independent set algorithms in [16, 55] and related problems such as graph coloring [50].
- Single-source shortest-path (Bellman-Ford, Δ-stepping in [56], ρ-stepping and Δ^*-stepping in [41]).
- Connectivity algorithms. Connectivity in [36, 58, 66], biconnecitivity in [42], and strong connectivity in [22, 75].
- Minimum spanning tree in [26, 76].

Advanced Topics. These advanced topics (or a subset of them) can be covered for a graduate-level course for 12–16 weeks.

- Geometry algorithms (e.g., convex hull [23, 46], Delaunay triangulation [22]), and data structures [20, 49, 70].
- I/O-efficient parallel algorithms [10, 14, 30, 45].
- Advanced graph algorithms: low-diameter decomposition [58], k-core and other graph-mining problems [34, 63, 64], graph processing systems [37, 38, 60, 67], and dynamic graph algorithms/processing [4, 35, 39, 74].
- Scheduling algorithms [2, 25, 48].
- Other concurrent data structures (e.g., [59]).

V. Student Evaluation in Previous Classes

As mentioned, the binary-forking model can be used for teaching parallel algorithms in classes with different purposes, levels, and focus. As an example, we show student evaluation results from (anonymous) surveys in two previous courses. The first one is an undergraduate course UCR CS 142 (Algorithm Engineering) in Winter'23, which included introduction-level parallel algorithms in one-week lectures. These lectures covered the model and algorithms in section III. At the end of the class, students were asked to choose 3–10 topics that they found the most interesting from 20 topics covered in class. The topic of parallel reduce, scan, filter, and quicksort algorithms is the most-liked topic (56.7% of students chose it).

In another course UCR CS 214: Parallel Algorithms, we surveyed how the model helps to combine theory and practice. The model is introduced at the beginning to support both theoretical analysis and programming. In the survey, students were asked if the class balances theory and practice on a scale of 1–5 (1 = too much focus on theory; 5 = too much focus on programming). All the answers are within the range of 2–4, and 47.6% of them chose 3 (a perfect balance of theory and practice). We will keep collecting more evaluations and feedback from students in the future.

VI. Conlusion

In this paper, we discussed how to use the binary-forking model in teaching parallel algorithms. This model provides a simple abstraction for parallel algorithm design. Algorithms in this model can be easily analyzed in the work-span model, and can be implemented with languages supporting fork-join parallelism. For these reasons, we believe it is well-suited for classes and can help students understand both the theory and practice of parallel algorithms.

Acknowledgment. This work is supported by NSF CCF-2103483, CCF-2119352, CCF-2227669, and NSF CAREER Awards CCF-2238358 and CCF-2339310.

REFERENCES

[1] U. Acar and B. G. E. Algorithms: Parallel and sequential. https://drive.google.com/file/d/1nuRlrqrRRsxMhX20NTd9YTPVANQi5Of-/view?usp=sharing, 2022.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theoretical Computer Science (TCS)*, 35(3), 2002.

[3] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[4] D. Anderson, G. E. Blelloch, and K. Tangwongsan. Work-efficient batch-incremental minimum spanning trees with applications to the sliding-window model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2), Apr 2001.

[6] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 1–10, 2012.

[7] G. E. Blelloch. Prefix sums and their applications. In J. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[8] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39 (3), Mar. 1996.

[9] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004.

[10] G. E. Blelloch and Y. Gu. Improved parallel cache-oblivious algorithms for dynamic programming. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2020.

[11] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 16–26, 1998.

[12] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems (TOCS)*, 32(3):213–239, 1999.

[13] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.

[14] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[15] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–366, 2011.

[16] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[17] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[18] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[19] G. E. Blelloch, L. Dhulipala, and Y. Sun. Introduction to parallel algorithms (draft), 2018.

[20] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[21] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020.

[22] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. *J. ACM*, 67(5):1–27, 2020.

[23] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[24] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.

[25] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[26] O. Boruvka. O jistém problému minimálním. *Práce Mor. Prırodved. Spol. v Brne (Acta Societ. Scienc. Natur. Moravicae)*, 3(3):37–58, 1926.

[27] Z. Budimlić, V. Cavé, R. Raman, J. Shirako, S. Taşırlar, J. Zhao, and V. Sarkar. The design and implementation of the habanero-java parallel programming language. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 185–186, 2011.

[28] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[29] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 339–350, 2017.

[30] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2008.

[31] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

[32] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[34] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[35] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.

[36] L. Dhulipala, C. Hong, and J. Shun. Connectit: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 14(4):653–667, 2020.

[37] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Semi-asymmetric parallel graph algorithms for NVRAMs. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[38] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[39] L. Dhulipala, G. E. Blelloch, Y. Gu, and Y. Sun. PaC-trees: Supporting parallel and compressed purely-functional collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[40] D. Dinh, H. V. Simhadri, and Y. Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 49–60, 2016.

[41] X. Dong, Y. Gu, Y. Sun, and Y. Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 184–197, 2021.

[42] X. Dong, L. Wang, Y. Gu, and Y. Sun. Provably fast and space-efficient parallel biconnectivity. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 52–65, 2023.

[43] X. Dong, L. Dhulipala, Y. Gu, and Y. Sun. Parallel integer sort: Theory and practice. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2024.

[44] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 33(5), 1998.

[45] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[46] M. T. Goodrich. Finding the convex hull of a sorted point set in parallel. *Information Processing Letters*, 26(4):173–179, 1987.

[47] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[48] Y. Gu, Z. Napier, and Y. Sun. Analysis of work-stealing and parallel

cache complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 46–60. SIAM, 2022.

[49] Y. Gu, Z. Napier, Y. Sun, and L. Wang. Parallel cover trees and their applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 259–272, 2022.

[50] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014.

[51] Intel Threading Building Blocks. Intel threading building blocks (TBB). https://www.threadingbuildingblocks.org.

[52] J. JáJá. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[53] Java Fork-Join, Oracle Java Documentation. Java fork-join, oracle java documentation. http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, 2022.

[54] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 1990.

[55] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Computing*, 15:1036–1053, 1986.

[56] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[57] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1996. doi: 10.1145/248052.248106.

[58] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013.

[59] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of data structures and applications*, pages 741–762. Chapman and Hall/CRC, 2018.

[60] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013.

[61] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[62] T. B. Schardl and I.-T. A. Lee. Opencilk: A modular and extensible software infrastructure for fast task-parallel code. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 189–203, 2023.

[63] J. Shi, L. Dhulipala, and J. Shun. Parallel clique counting and peeling algorithms. pages 135–146. SIAM, 2021.

[64] J. Shi, L. Dhulipala, and J. Shun. Theoretically and practically efficient parallel nucleus decomposition. *Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[65] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms*, 2(1), 1981.

[66] J. Shun. Fast parallel computation of longest common prefixes. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 387–398. IEEE, 2014.

[67] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 135–146, 2013.

[68] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[69] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015.

[70] Y. Sun and G. E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

[71] Y. Sun, D. Ferizovic, and G. E. Blelloch. PAM: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

[72] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 205–214, 2015.

[73] Task Parallel Library (TPL). Task parallel library (TPL). https://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx.

[74] T. Tseng, L. Dhulipala, and G. Blelloch. Batch-parallel Euler tour trees. *Algorithm Engineering and Experiments (ALENEX)*, 2019.

[75] L. Wang, X. Dong, Y. Gu, and Y. Sun. Parallel strong connectivity based on faster reachability. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2023.

[76] W. Zhou. A practical scalable shared-memory parallel algorithm for computing minimum spanning trees. Master's thesis, Karlsruhe Institute of Technology, 2017.