Scalable Parallel Algorithms and Data Structures

Study notes

Lam Nguyen

Last updated November 23, 2026

About these notes

These are my notes from following *Scalable Parallel Algorithms and Data Structures* (CMSC 858N, University of Maryland, Fall 2026), taught by Laxman Dhulipala, through its public schedule and readings:

https://www.cs.umd.edu/~laxman/paralg-f26.html

Each chapter covers one lecture or reading, in my own words: what I understood, the proofs I worked through, and the questions I still have. The mistakes are mine.

Alongside the notes, I write the algorithms myself and time them on my own machine. That code, with a test and a benchmark for it, is in my competitive programming notebook:

https://github.com/lamng3/competitive-programming-notebook/tree/main/notebook/parallel

Contents

1 Models and Primitives **3**

1.1 The question . . . 3
1.2 The idea . . . 3
1.2.1 The RAM . . . 3
1.2.2 The PRAM: processors in lockstep . . . 3
1.2.3 The MP-RAM: processes that fork . . . 4
1.2.4 A computation is a DAG . . . 4
1.2.5 From work and span to running time . . . 5
1.2.6 Partial orders and linearizations . . . 6
1.2.7 Conflicts and races . . . 7
1.3 Practice . . . 7
1.3.1 ParlayLib . . . 8
1.3.2 Map . . . 9
1.3.3 Sum . . . 9
1.4 Questions I still have . . . 12
1.5 What to remember . . . 12

Chapter 1

Models and Primitives

Lecture 1, September 8, 2026: work and depth, scheduling, basic parallel primitives.
Reading: Guy E. Blelloch, Laxman Dhulipala, and Yihan Sun, *Introduction to Parallel Algorithms*, draft of February 6, 2023, Section 2 (Models).
Code: `notebook/parallel/book/map.cpp` and `notebook/parallel/book/sum.cpp`.

1.1 The question

To say that one algorithm is faster than another, we need a model: a concrete machine with a well-defined cost for every step. For sequential algorithms that model has been settled for decades. For parallel algorithms, the first instinct is to measure the running time on P processors. But then every bound depends on P, and every algorithm has to say which processor does what at each moment, which has little to do with the algorithm itself.

This lecture replaces the processors with two numbers that never mention them: the *work*, the total number of operations, and the *span*, the longest chain of operations that must happen one after another. A scheduler turns the two into a running time on any number of processors.

1.2 The idea

1.2.1 The RAM

The *Random Access Machine* (RAM) is the model behind sequential algorithms. The name describes the machine, not random-access memory: one processor with a few registers and a program counter, and a memory it can read and write at any address. Every instruction takes one unit of time, so the cost of a program is the number of instructions it runs.

The RAM leaves out caches and the rest of the hardware on purpose. It does not predict how many seconds a program takes on a real machine. What it does well is compare algorithms and show how their cost grows with the input, so it is used for big-O analysis only.

1.2.2 The PRAM: processors in lockstep

The first parallel model was the *Parallel RAM* (PRAM): P RAM processors that share one memory and run in lockstep. I picture a marching band. On every tick of the clock, every processor takes exactly one step, and no one moves on to the next tick until everyone has finished this one.

PRAMs differ in what may happen when processors touch the same memory cell in the same tick:

- **EREW** (exclusive read, exclusive write): no two processors access the same cell in one tick.
- **CREW** (concurrent read, exclusive write): many may read a cell, but only one may write it.
- **CRCW** (concurrent read, concurrent write): many may read and write a cell; the model also needs a rule for which write wins.

A PRAM algorithm has two costs: its *time*, or depth, the number of ticks, and its *work*, the total number of operations over all processors. (Charging idle processors too makes the work P times the time.) An algorithm is *work-efficient* if its work matches, asymptotically, the best sequential algorithm on the RAM.

The trouble is the fixed P. A PRAM algorithm has to say what each of the P processors does on every tick, so its pseudocode is mostly about handing out tasks to processors.

1.2.3 The MP-RAM: processes that fork

The *Multi-Process RAM* (MP-RAM) has no processors at all. It has *processes*, which share an unbounded memory and each run RAM instructions with their own registers and program counter, and one new instruction, FORK. A process that runs FORK k creates k children, numbered $1, \ldots, k$, which all start at the next instruction. The parent waits until all k children have finished; the first instruction it runs after that is the *join*. Children may fork too, so parallelism nests. An algorithm now says only what *may* run in parallel, and nothing about who runs it.

The reading's pseudocode has two constructs for this, and both are forks underneath:

- a *parallel loop*, **parfor** i **in** $[s : e]$, runs its iterations in parallel. As in the reading, $[s : e]$ means the integers from s up to but not including e;
- a *parallel do*, written with double bars: in $a \parallel b$, statements a and b run in parallel, and a semicolon after them waits for both.

1.2.4 A computation is a DAG

A run of an MP-RAM program orders its instructions by three rules:

1. **Sequence.** An instruction comes after the previous instruction of the same process.
2. **Fork.** A child's first instruction comes after the FORK that created it.
3. **Join.** The join comes after the last instruction of every child.

With a node for each instruction and an edge for each rule, the run becomes a directed acyclic graph (DAG). Figure 1.1 is the DAG of a small program that adds up nine numbers in three uneven pieces: the parent forks three children, each child adds its piece into its own variable, and the join adds the three results.

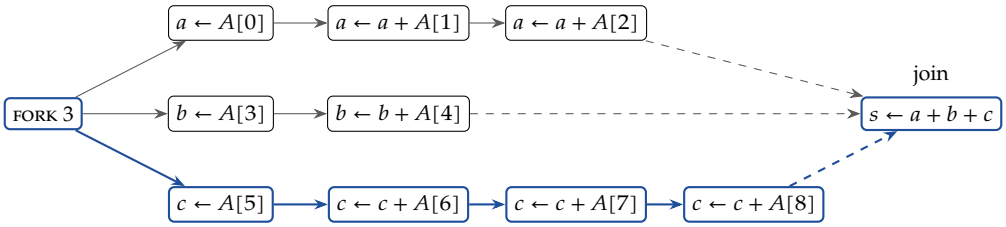


Figure 1.1: A fork with three children of 3, 2, and 4 instructions, which add up $A[0..8]$ in three pieces. Solid edges come from the sequence and fork rules, dashed edges from the join rule. The longest path, in blue, gives the span.

Work and span. The *work* W is the number of nodes, which is the running time on one processor. The *span* S (the reading writes D, for depth) is the number of nodes on the longest path, which is the running time with unlimited processors: no number of processors can shorten a chain of instructions that each wait for the one before. In Figure 1.1,

$$W = \underbrace{1}_{\text{FORK}} + 3 + 2 + 4 + \underbrace{1}_{\text{join}} = 11, \qquad S = 1 + \max(3, 2, 4) + 1 = 6.$$

That is the rule for every fork: the work of the children adds up, and only the longest child counts toward the span. Analyzing a parallel algorithm therefore comes down to two recurrences, one with a sum and one with a maximum.

Parallelism. The ratio W/S is the *parallelism*: roughly how many processors the computation can keep busy. Here it is $11/6 \approx 1.8$, so about two. My first guess at the units had the span in seconds per processor, and that is backwards: the processor factor belongs to the work. Work is counted in processor-seconds, every operation that some processor must perform, while span is counted in plain seconds, steps that must happen one after another. So W/S is measured in processors.

1.2.5 From work and span to running time

Let T_P be the running time on P processors. Two lower bounds hold for any schedule: P processors perform at most P instructions per step, so $T_P \geq W/P$, and a chain of S dependent instructions takes S steps, so $T_P \geq S$. Equation (1) of the reading pairs them with an upper bound of the same shape:

$$\max\left(\frac{W}{P}, S\right) \leq T_P \leq O\left(\frac{W}{P} + S\right).$$

The upper bound is the scaling law of parallel algorithms, and its simplest form has a short proof.

Theorem 1.1 (Brent)**.** *Call a schedule greedy if at every step it runs as many ready instructions as it can, up to P. Every greedy schedule finishes within $W/P + S$ steps.*

Proof. Call a step *full* if all P processors run an instruction, and *partial* otherwise. Each full step runs P of the W instructions, so there are at most W/P full steps. A partial step runs every ready instruction. The first instruction of a longest path among the instructions still to run is always

ready, since an unfinished instruction before it would make the path longer. So every partial step shortens the longest remaining path by one, and there are at most S partial steps. □

Since $W/P + S \leq 2\max(W/P, S)$, a greedy schedule is never more than twice as slow as the best one. Figure 1.1 shows the bound at work:

- with $P = 1$, everything runs in turn: $T_1 = 11 = W$;
- with $P = 3$, each child gets its own processor: $T_3 = 6 = S$;
- with $P = 2$, the bound says $6 \leq T_2 \leq 11.5$. The best schedule takes 7 steps: after the fork, one processor runs the long child while the other runs the two short ones, and then the join runs. A greedy schedule that starts the two short children first takes 8 steps, still within the bound.

A real scheduler cannot see the whole DAG in advance, so it cannot be exactly greedy. ParlayLib's scheduler uses *work stealing*, which achieves $O(W/P + S)$ in expectation; the reading proves bounds of this kind in Section 8.

What to aim for. Here *scalability* means gaining speed as the number of processors grows while the input stays fixed. (Elsewhere the word usually means coping with a growing input.) On real machines P is small and W is at least linear in n, so the W/P term dominates. The first goal is therefore to be work-efficient: no number of processors makes up for work that a sequential algorithm never does. The span then decides how far the speedup can go, since once P approaches W/S, more processors stop helping. The second goal is a span that never dominates, ideally polylogarithmic in n, which keeps the parallelism W/S far above any real P.

1.2.6 Partial orders and linearizations

The three rules order some pairs of instructions and leave other pairs unordered. That is exactly a partial order.

Definition 1.2. A *partial order* $\prec$ on a set is a "comes before" relation that is *irreflexive*, so never $x \prec x$, and *transitive*, so $x \prec y$ and $y \prec z$ give $x \prec z$. Two elements are *comparable* if $x \prec y$ or $y \prec x$, and *incomparable* otherwise: neither has to come before the other.

Reachability in a DAG is a partial order: $x \prec y$ when there is a path from x to y, and acyclicity is what makes it irreflexive. In Figure 1.1, the fork comes before everything and everything comes before the join, but $a \leftarrow A[0]$ and $b \leftarrow A[3]$ are incomparable. Two instructions are *concurrent* if they are incomparable, and *ordered* otherwise.

A *linearization* is a total order on the instructions that respects the partial order: a topological sort of the DAG, which flattens it into a single timeline with every edge pointing forward. Figure 1.1 has many. The fork must come first and the join last, and in between the three children may interleave in any way that keeps each child's own order, which gives

$$\binom{9}{3,\,2,\,4} = \frac{9!}{3!\,2!\,4!} = 1260$$

linearizations. One of them is the *sequential ordering*, which runs the children one after another, in order: exactly what the program does as plain sequential code. Another starts $a \leftarrow A[0]$, $c \leftarrow A[5]$, $b \leftarrow A[3]$, $a \leftarrow a + A[1]$, and so on.

The MP-RAM ties memory to linearizations: the results of all memory operations are consistent with *some* linearization, in which every read returns the value of the latest write to the same location before it. The model does not say which linearization, and a real machine may pick a different one on every run.

1.2.7 Conflicts and races

Definition 1.3. Two instructions *conflict* if they access the same memory location and at least one of them writes it. They *race* if they conflict and are concurrent.

A race is what makes the choice of linearization visible. Suppose a parallel loop adds every $A[i]$ into one shared variable with $total \leftarrow total + A[i]$. On a RAM that is two instructions, a read of *total* and then a write. For two iterations, the linearization "read, write, read, write" gives the right sum, but "read, read, write, write" has both iterations read the old total, and the second write erases the first. The program in Figure 1.1 has no races: the children only read A, each writes its own variable, and the join reads a, b, and c only after every child has finished.

Without races, the choice stops mattering.

Lemma 1.4. *Any two linearizations of a partial order on a finite set are connected by a sequence of swaps, each exchanging two adjacent incomparable elements.*

Proof. Let L and L' be two different linearizations, let i be the first position where they differ, and let x be the element at position i of L'. In L, x sits at some position $j > i$; let y be the element just before it, at position $j - 1 \geq i$. Then $y \prec x$ is impossible: L' would have to place y before position i, where L and L' hold the same elements, yet y is at position $j - 1 \geq i$ of L. And $x \prec y$ is impossible because L places y first. So x and y are incomparable, and swapping them gives another linearization. Repeating this moves x to position i, after which L and L' agree on one more position. □

Theorem 1.5. *If a computation has no races, every linearization gives the same result: each read returns the same value, and memory ends in the same state.*

Proof. Swap two adjacent concurrent instructions x and y in a linearization. They do not race, so they do not conflict: they touch different locations, or both only read a common one. Each therefore reads what it read before, because the other writes nothing it reads, and every other instruction still sees the same latest writes, because x and y are still both before it or both after it. So every read returns the same value, and every write writes the same value. By Lemma 1.4, every linearization is a sequence of such swaps away from the sequential ordering. □

So to prove a race-free parallel program correct, it is enough to prove its sequential ordering correct, and that is an ordinary sequential program. The other half, checking that a program is race-free, has a tool in practice: ThreadSanitizer, which `cppar --check --tsan` runs, reports any two accesses to the same location, at least one a write, that it saw happen with nothing ordering them.

1.3 Practice

The code for this chapter is in the `notebook/parallel/book` folder of my notebook, one file per example from the reading. Each file times several versions of one algorithm, with two tools from the notebook:

- `parbench::run`, from `tools/parbench.h`, times a version and checks its answer against the first version's;
- `cppar`, from `tools/cppar.py`, builds a file with ParlayLib and runs it on 1, 2, 4, 8, and 12 threads.

All times below come from my laptop, an Apple M2 Pro with 12 cores (8 performance cores and 4 slower efficiency cores), on $n = 2^{24}$ random integers below 100. Each is the median of 7 runs. Sequential code ignores the thread count, so its row in each table shows how much runs vary.

1.3.1 ParlayLib

ParlayLib is a C++ library that runs the MP-RAM's constructs on a real machine. Table 1.1 lists the ones this chapter uses.

Table 1.1: The reading's constructs in ParlayLib.

The reading	ParlayLib
$a \parallel b$	`parlay::par_do(a, b)`, where `a` and `b` are lambdas
parfor i **in** $[s : e]$	`parlay::parallel_for(s, e, [&](std::size_t i) { ... })`
an array	`parlay::sequence<T>`
the slice $A[s : e]$	`parlay::make_slice(A).cut(s, e)`

`par_do` is a binary fork. It puts the second lambda on the current thread's queue of jobs, runs the first lambda itself, and then runs the second as well, unless an idle thread stole it in the meantime. Idle threads steal from the queues of random other threads, which is the work stealing of the previous section. More than two tasks become a tree of binary forks; `parlay::par_do3` is just

```
auto left_mid = [&]() { par_do(left, mid); };
par_do(left_mid, right);
```

`parallel_for` is a tree of `par_do`s that stops early. Before splitting, it runs the first iterations itself, in batches of doubling size, until a batch takes a microsecond; it never splits a range below that many iterations, or below $n/(128P)$, so each thread gets about 128 pieces to balance. That built-in cutoff matters below.

A slice is two pointers into an existing array. `make_slice` and `cut` copy nothing and cost $O(1)$, as slices do in the reading. The type of a slice names its pointers; for a slice of a `parlay::sequence<long long>` it is

```
parlay::slice<const long long*, const long long*>,
```

so functions that take a slice are templates.

1.3.2 Map

Algorithm 1.1 applies a function f to every element of A. A parallel loop over n iterations is one fork with n children in the MP-RAM, so a map has $O(n)$ work and $O(1)$ span. With only binary forks, as in ParlayLib, the n children become a tree of depth $\log_2 n$, and the span is $O(\log n)$.

Algorithm 1.1 Map

function MAP(f, A)
  $B \leftarrow$ a new array of length $|A|$
  parfor i **in** $[0 : |A|]$
    $B[i] \leftarrow f(A[i])$
  return B

My version, from book/map.cpp, with $f(x) = 10x$:

```
parlay::sequence<long long> map_parallel(const parlay::sequence<long long>& a) {
    const std::size_t n = a.size();
    auto b = parlay::sequence<long long>::uninitialized(n);
    parlay::parallel_for(0, n, [&](std::size_t i) {
        b[i] = f(a[i]);
    });
    return b;
}
```

Table 1.2 has the times. The map runs 4.8 times faster than the loop on 8 threads. On one thread it is a little slower than the loop, 8.06 against 6.85 milliseconds.

Table 1.2: Milliseconds to map $f(x) = 10x$ over 2^{24} integers, in book/map.cpp.

	threads				
version	1	2	4	8	12
`sequential`, a loop	6.85	8.15	8.15	8.02	8.01
`map`, with `parallel_for`	8.06	4.08	2.21	1.67	1.77

1.3.3 Sum

Algorithm 1.2 adds up an array by splitting it in half and adding up both halves in parallel. The two recursive calls run at the same time, and the return waits for both. Each call does a constant amount of work besides its two recursive calls, so

$$W(n) = 2\,W(n/2) + O(1) = O(n), \qquad S(n) = S(n/2) + O(1) = O(\log n).$$

The algorithm is work-efficient, and its parallelism is $\Theta(n/\log n)$: about 700,000 for $n = 2^{24}$, far more than 12 threads need.

Algorithm 1.2 Sum

function SUM(A)
  if $|A| = 0$ **then**
    return 0
  if $|A| = 1$ **then**
    return $A[0]$
  $m \leftarrow \lfloor |A|/2 \rfloor$
  $\ell \leftarrow$ SUM($A[0 : m]$) $\parallel$ $r \leftarrow$ SUM($A[m : |A|]$)
  return $\ell + r$

My version, from `book/sum.cpp`, recurses on slices:

```
template<typename It>
long long sum_parallel(parlay::slice<It, It> s) {
    const std::size_t n = s.size();
    if (n == 0) return 0;
    if (n == 1) return s[0];
    auto mid = n / 2;
    long long left, right;
    parlay::par_do(
        [&]() { left = sum_parallel(s.cut(0, mid)); },
        [&]() { right = sum_parallel(s.cut(mid, n)); }
    );
    return left + right;
}
```

The two lambdas write different variables, `left` and `right`, and the parent reads them only after `par_do` returns, so there is no race; `cppar sum --check --tsan` agrees.

Table 1.3 compares it with a plain loop, the same recursion without `par_do`, a version with a cutoff, and ParlayLib's own `parlay::reduce`.

Table 1.3: Milliseconds to add up 2^{24} integers with each version in `book/sum.cpp`.

	threads				
version	1	2	4	8	12
`sequential`, a loop	2.78	2.24	2.20	2.68	2.41
`recursive`, without `par_do`	34.7	35.1	35.2	34.8	34.7
`parallel`, Algorithm 1.2	180	91.4	46.3	23.8	20.6
`parallel + cutoff`, Algorithm 1.3	2.37	1.24	1.17	1.07	1.05
`parlay::reduce`	2.21	1.20	1.20	1.09	1.10

It scales, and it still loses. The `parallel` row behaves as Theorem 1.1 predicts: 180 milliseconds on one thread, 23.8 on 8 threads (7.6 times faster), and 20.6 on 12, where the efficiency cores add less than the first 8. Yet on 12 threads it is still 8.5 times slower than a plain loop. Work-efficiency is an asymptotic statement: both versions do $O(n)$ work, but the parallel one pays for a function

call and a `par_do` on every element. On one thread, the recursion alone is 12.5 times slower than the loop (34.7 against 2.78 milliseconds), and the forks add 145 milliseconds more, about 8.7 nanoseconds per `par_do`. The map never paid this, because `parallel_for` chose its own pieces.

A cutoff. The fix is to stop forking once a piece is small and add it up with a loop instead, as in Algorithm 1.3.

Algorithm 1.3 Sum with a cutoff c

function SUM(A)
  if $|A| \leq c$ **then**
    return $A[0] + A[1] + \cdots + A[|A| - 1]$, added in a loop
  $m \leftarrow \lfloor |A|/2 \rfloor$
  $\ell \leftarrow \text{SUM}(A[0 : m]) \parallel r \leftarrow \text{SUM}(A[m : |A|])$
  return $\ell + r$

With pieces of at most c elements, there are about n/c pieces and as many forks, so the forks cost c times less. The work is still $O(n)$, and the span grows only to $O(c + \log(n/c))$. In the code, `sum_parallel_cutoff` differs from `sum_parallel` in its base case, which hands small slices to the loop:

```
if (n <= cutoff) return sum_sequential(s);
```

Choosing c is a trade-off. Too small, and the forks still cost; too large, and there are too few pieces to keep every thread busy. Table 1.4 measures it:

- at $c = 64$ there are still 262,144 pieces, and one thread takes 4.44 milliseconds, about 7 nanoseconds per fork again;
- from $c = 256$ to $c = 65{,}536$ the times are all about the same;
- at $c = 2^{22}$ there are only 4 pieces, so at most 4 threads can work, and 8 threads take 1.88 milliseconds instead of about 1.2.

I use $c = 4096$. With it, `parallel + cutoff` in Table 1.3 matches the loop on one thread (2.37 milliseconds) and takes 1.05 milliseconds on 12 threads: 2.3 times faster than the loop, and as fast as `parlay::reduce`. That is no accident, since `parlay::reduce` does the same thing in another shape. It splits the array into blocks of $\max(1024, 4\lceil \sqrt{n} \rceil)$ elements (16,384 here), adds up each block with a loop inside a `parallel_for`, writing block i's total into slot i of an array of sums so that no two blocks race, and then adds up the sums.

The ceiling is memory. No version gets much below 1.05 milliseconds, however many threads it has. Adding up 2^{24} eight-byte integers reads 134 megabytes, and 134 megabytes in 1.05 milliseconds is about 128 gigabytes per second, a large share of the M2 Pro's rated 200; one core alone reads about 57. A sum does one addition for every 8 bytes it reads, so from about 4 threads on it waits for memory, not for processors. The work-span model cannot see this: like the RAM, it charges every instruction one unit, reads from memory included.

Table 1.4: Milliseconds to add up 2^{24} integers with the cutoff version, for a range of cutoffs. Measured in one run, with a copy of `sum.cpp` that takes the cutoff as a template parameter.

		threads				
cutoff c	pieces	1	2	4	8	12
64	262,144	4.44	2.50	1.48	1.08	1.27
256	65,536	2.65	1.87	1.31	1.12	1.24
1,024	16,384	2.54	2.16	1.25	1.19	1.22
4,096	4,096	2.55	2.17	1.22	1.33	1.31
16,384	1,024	2.61	2.17	1.25	1.18	1.33
65,536	256	2.25	1.85	1.31	1.26	1.30
2^{20}	16	2.22	1.93	1.32	1.47	1.22
2^{22}	4	2.31	1.53	1.29	1.88	1.42
the loop		2.61	2.69	2.46	2.26	2.63
`parlay::reduce`		2.57	1.53	1.24	1.19	1.18

1.4 Questions I still have

- The map runs 4.8 times faster than its loop on 8 threads, while the sum stops near 2.5 times. Both stream through memory once. What limits each?
- ParlayLib's scheduler steals work without ever seeing the DAG. How does work stealing still achieve $O(W/P + S)$ in expectation? (Reading, Section 8.5.)

1.5 What to remember

- The RAM charges one unit per instruction. It is for comparing algorithms asymptotically, not for predicting seconds.
- The PRAM runs P processors in lockstep, as EREW, CREW, or CRCW. Its fixed P makes every algorithm schedule its own processors.
- The MP-RAM forks processes instead, and a run is a DAG. The work W is its number of nodes and the span S its longest path. A fork adds its children's work and takes the largest of their spans.
- $\max(W/P, S) \le T_P$, and every greedy schedule achieves $T_P \le W/P + S$ (Theorem 1.1). Aim for work-efficient algorithms with polylogarithmic span, so that W/S is far above P.
- Without races, every linearization gives the same result (Theorem 1.5), so a race-free program can be proved correct as a sequential one.
- The constant hidden in $O(n)$ work decides whether a parallel algorithm beats a loop. A `par_do` per element costs about 9 nanoseconds. A cutoff, which ParlayLib calls the granularity, removes that cost, and the sum then runs into the memory bandwidth at about 2.3 times the loop.