Introduction to Parallel Algorithms

Guy E. Blelloch, Laxman Dhulipala, Yihan Sun

February 20, 2024

Contents

1 Models **1**

2 Preliminaries **3**

3 Some Building Blocks **3**
3.1 Scan 3
3.2 Filter and Flatten 5
3.3 Search 6
From `https://www.cs.cmu.edu/~guyb/paralg/paralg/parallel.pdf`

1 Models

We use the **Random Access Machine** (RAM) model which consists of a single processor with some constant number of registers, an instruction counter and an arbitrarily large memory. The RAM model assumes that all instructions take unit time.

The RAM is by no stretch meant to model the runtime on a real machine with cycle-by-cycle level accuracy. It does not model, for example, that modern-day machines have cache hierarchies and therefore not all memory accesses are equally expensive.

In **work-span** models, algorithms still assume a shared random access memory, but allow dynamically creating tasks. Costs are measured in terms of the total number of operations, the **work** and the longest chain of dependence.

We call a parallel algorithm **work-efficient** if its work is work asymptotically the same as its best-known sequential counterpart.

The span for a parallel algorithm is the running time when you have an infinite number of processors.

The **Multi-Process Random-Access Machine** (MP-RAM) consists of a set of processes that share an unbounded memory. The MP-RAM extends the RAM with a fork instruction that takes a positive integer k and forks k new child processes. Each child process receives a unique integer in the range $[1, ..., k]$ in its first register and otherwise has the identical state as the parent (forking process), which has that register set to 0. All children start by running the next instruction, and the parent suspends until all the children terminate (execute an end instruction). The first instruction of the parent after all children terminate is called the **join** instruction. A **computation** starts with a single root process and finishes when that root process ends. This model supports **nested parallelism** -the ability to fork processes in a nested fashion. If the root process never does a fork, it is a standard sequential program.

A computation in the MP-RAM defines a partial order on the instructions. In particular

1. every instruction depends on its previous instruction in the same thread (if any),
2. every first instruction in a process depends on the fork instruction of the parent that generated it, and
3. every join instruction depends on the end instruction of all child processes of the corresponding fork generated.

The work of a computation is the total number of instructions, and the span is the longest sequences of dependent instructions.

Two instructions are said to be **concurrent** if they are unordered, and ordered otherwise. Two instructions **conflict** if one writes to a memory location that the other reads or writes the same location. We say two instructions race if they are concurrent and conflict.

A testAndSet(x) (TS) instruction takes a reference to a memory location x, checks if the value of x is `false` and if so atomically sets it to `true` and returns `true`; if already `true` it returns `false`.

A compareAndSwap(x, o, n) (CAS) instruction takes a reference to a memory location x, checks if the value of x equals o. If so, the instruction will change the value to n and return `true`. If not, the instruction does nothing and simply returns `false`.

A fetchAndAdd(x, y) (FA) instruction takes a reference to a memory location x, and a value y, and it adds y to the value of x, returning the old

value. Different from a TS or a CAS, an FA instruction always successfully adds y to the value stored in x.

A PRIORITYWRITE(x, y) (PW) instruction takes a reference to a memory location x, and checks if the value y is less than the current value in x. If so, it changes the value stored in x to y, and return `true`. If not, it does nothing and return `false`.

2 Preliminaries

Definition 2.1 (w.h.p.)**.** $g(n) \in O(f(n))$ with **high probability** (w.h.p.) if $g(n) \in O(cf(n))$ with probability at least $1 - (\frac{1}{n})^c$, for some constant c_0 and all $c \geq c_0$.

Theorem 2.2. *Consider a set of indicator random variables $X_1, \ldots, X_n$ for which $p(X_i = 1) \leq \bar{p}_i$ conditioned on all possible events $X_j = \{0, 1\}$, $i \neq j$. Let $X = \sum_{i=1}^{n} X_i$ and $\overline{E}[X] = \sum_{i=1}^{n} \bar{p}_i$, then*

$$\Pr[X \geq k] \leq \left(\frac{e\overline{E}[X]}{k}\right)^k$$

Proof. Let's first consider the special case that the $\bar{p}_i$ are all equal and have value p. If $X \geq k$, then we have that at least k of the random variables are 1. The probability of any particular k variables all being 1, and the others being anything, is upper bounded by p^k.

$$\Pr[X \geq k] \leq p^k \binom{n}{k} < p^k \left(\frac{ne}{k}\right)^k = \left(\frac{pne}{k}\right)^k = \left(\frac{e\overline{E}[X]}{k}\right)^k$$

Here we used a standard upper bound on the binomial coefficients: $\binom{n}{m} < \left(\frac{ne}{m}\right)^m$. □

3 Some Building Blocks

3.1 Scan

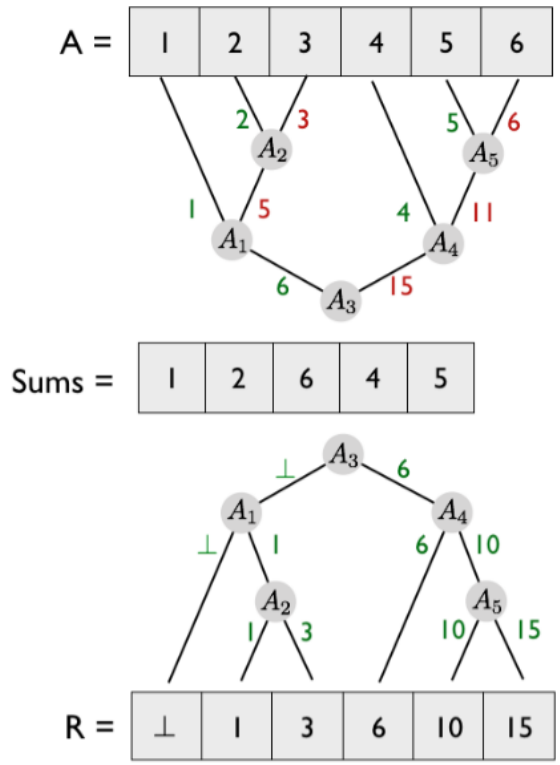
A **scan** or **prefix-sum** function takes a sequence A, an associative function f, and a left identity element $\perp$ and computes the values

$$r_i = \begin{cases} \perp & i = 0 \\ f(r_{i-1}, A_i) & 0 < i \leq |A| \end{cases}$$

Each r_i is the "sum" of the prefix $A[0, i]$ of A w.r.t. the function f.

```
function scanUp(A, L, f)
    if |A| = 1 then return |A[0]|
    else
        n ← |A|;
        m ← n/2;
        l ← scanUp(A[0 : m], L[0 : m − 1], f)    ||
        r ← scanUp([A[m : n], L[m : n − 1]], f);
        L[m − 1] ← l;
        return f(l, r)
function scanDown(R, L, f, s)
    if |R| = 1 then
        R[0] = s;
        return
    else
        n ← |A|;
        m ← |R|/2;
        scanDown(R[0 : m], L[0 : m − 1], s)    ||
        scanDown(R[m : n], L[m : n − 1], f(s, L[m − 1]));
        return
function scan(A, f, I)
    L ← array[|A| − 1];
    R ← array[|A|];
    total ← scanUp(A, L, f);
    scanDown(R, L, f, I);
    return ⟨R, total⟩
```

For scanUp it should be clear that the values written into L are indeed the sums of the left subtrees. For scanDown consider a node v in the tree and the value s passed to it. The algorithm maintains that the value s is the sum of all values to the left of the subtree rooted at v.



Work of scanUp and scanDown is

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

and the span is

$$D(n) = D(n/2) + O(1) = \log(n)$$

3.2 Filter and Flatten

function FILTER(A, p)
    $n \leftarrow |A|$;
    $F \leftarrow$ ARRAY$[n]$;
    parfor $i \in [0 : n]$ **do**
        $F[i] \leftarrow p(A[i])$
    $\langle X, m \rangle \leftarrow$ PLUSSCAN(F);
    $R \leftarrow$ ARRAY$[m]$;

```
    parfor i ∈ [0 : n] do
        if F[i] then
            R[X[i]] ← A[i];
    return R
```

Work $O(n)$, span $O(\log n)$

```
function FLATTEN(A)
    sizes ← ARRAY(|A|);
    parfor i ∈ [0 : |A|] do
        sizes[i] ← |A[i]|;
    ⟨X, m⟩ ← PLUSSCAN(sizes);
    R ← ARRAY(m)
    parfor i ∈ [0 : |A|] do
        o ← X[i];
        parfor j ∈ [0 : |A[i]|] do
            R[o + j] ← A[i][j]
    return R
```

3.3 Search

The sorted search problem is given a sorted sequence A and a key v, to find the position of the greatest element in A that is less than v.

```
▷ finds which of k blocks contains v, returns block and offset ◁
function FINDBLOCK(A, v, k)
    s ← |A|/k
    r ← k
    parfor i ∈ [0 : k] do
        if A[i × s] < v ∧ A[(i + 1) × s] > v then
            r ← i
    return (A[r × s, (r + 1) × s], i × s)
```