2/17/09

L7: scan and applications

1. Review of last lecture.

analytical models
- scalability. overhead vs. problem size.
- isoefficiency
- minimum execution time.
  (exists because overhead often a function of # processors)
- cost-optimal minimum execution time.

- scan.

① steps. show actual algorithm.
only left nodes are needed.
allowing for in place computation

② prescan → scan? → add element back.

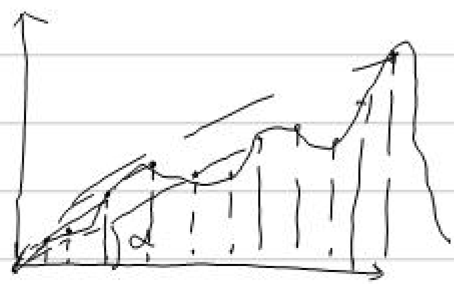
③ complexity. → still $O(\frac{n}{p} + \log p)$
on p processors
(demo).

④ associate operations. (such as max, min). ⊕scan

2. Use scan for line of sight.
- list compaction, split.
- radix sort.

① line of sight. [Blelloch 90]

problem: determine visible points relative to observation point X on a terrain map.

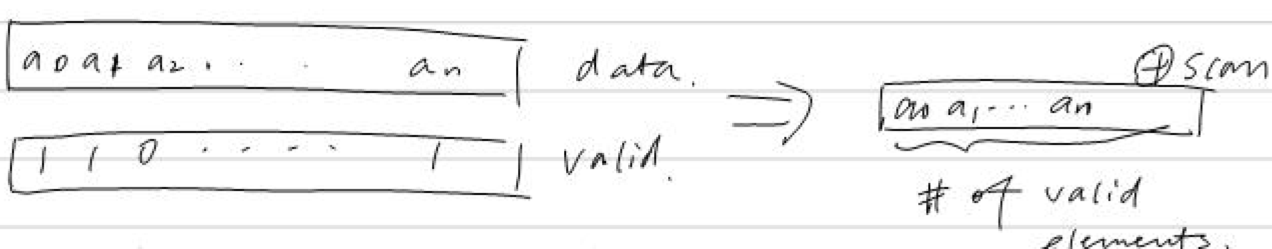



steps:

- compute angle α.
- do a max-scan to find the maximum angle so far
- compare angle α with max-scan results.

② list compaction:

problem: given a data array and a binary array (indicating whether a test has passed or not). produce a compacted array that contains only the successful elements.




~~problem:~~ solution: do a +-scan on the valid array to find out the output index of each element.

for example:

valid: 1 1 0 0 1 0 1 1 0 0 0

0 1 2 2 2 3 3 4 5 5 5 . . . .

these are the indices where we should put the elements to.

steps: — +-prescan of valid array

— in parallel. output valid elements to the ~~out~~ output buffer indicated by the index.

---

∘ split: [blelloch93].

basically a list compaction. except we want to also output the invalid elements on the right side of the valid elements:

example: a0 a1 a2 a3 a4 a5 a6

1 1 1 0 0 1 0

output: a0 a1 a2 a5 | a3 a4 a6

application in quick split

can compute the index for valid elements as before. what about invalid elements?

solution: ~~sith~~ simultaneously outputting two ~~arrays~~ scans, one for valid elements one for invalid.

the invalid ~~array~~ scan is ~~pr~~ produced by inverting the flags.

---

another example. classification and resorting.

| | | | | | | |
|---|---|---|---|---|---|---|
| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_n$ |

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 2 | 4 | 1 | .... |

← type:

| | | | | |
|---|---|---|---|---|
| $a_2$ $a_5$ | $a_1$ $a_3$ | $a_0$ | $a_4$ | .... |
| type 1 | type 2 | 3 | 4 | |

~~probabl~~ probably not too efficient using scan.

alternative: keep a buffer that stores for each type the total # of elements belonging to this type.

and use atomic add to increment.

---

radix sort: refer to paper [blelloch 90]

☆ recurrence equations: $x_i = f_i(x_{i-1}, x_{i-2} \ldots x_{i-m})$
$m \le i \le n$.

- prefix ~~so~~ sum is just a special kind of recurrence:

$$\begin{cases} x_i = x_{i-1} + \cancel{a_i}\ b_i \\ x_0 = \cancel{a_0}\ b_0 \end{cases}$$

more generally:

first order recurrence: $x_i = \begin{cases} b_0 & i=0 \\ (x_{i-1} \times a_i) + b_i & 0<i<n \end{cases}$

( prefix sum is just a special case of $a_i = 1$ )

---

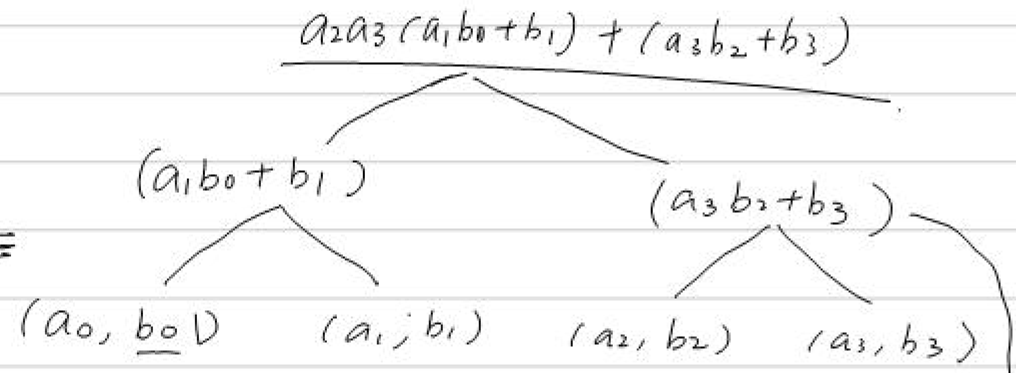
the big picture: this is seemingly a sequential computation. how do we paralleliize it?

idea: split the problem into small segments work on each segment at a time.

then find a way to combine the ~~result~~ results.

so the key question is: how to collaborate?

Take a look at an example to find out insights.
Let's first work on the reduction part (up-sweep), once this is done, the down-sweep pass is similar:




~~$a_0$ $a_1$~~

$$((a_1 b_0 + b_1) a_2 + b_2) a_3 + b_3$$

$$= a_2 a_3 (a_1 b_0 + b_1) + (a_3 b_2 + b_3)$$

looks like we should keep $a_2 \cdot a_3$ with the right tree node here

So, let's construct a ~~ex~~ pair to keep at every tree node. start from the leafs

The upate rules are:

$$\cancel{(a_i, a_j} \quad (a_i, b_i) \bullet (a_j, b_j) = [a_i \times a_j,\ b_i \times a_j + b_j]$$

So:

$$[a_0a_1a_2a_3,\ a_2a_3(a_1b_0+b_1)+(a_3b_2+b_3)]$$

$$[a_0a_1,\ a_1b_0+b_1] \qquad [a_2a_3,\ a_3b_2+b_3]$$

$$[a_0, b_0] \quad [a_1, b_1] \qquad [a_2, b_2] \quad [a_3, b_3]$$

proof that this is correct: prove $\bullet$ is associative

$$([a_i, b_i] \bullet [a_j, b_j]) \bullet [a_k, b_k]$$

$$= [a_i, b_i] \bullet ([a_j, b_j] \bullet [a_k, b_k])$$

therefore we can do scan ~~of~~ on $\bullet$ operation.

(since it's associative)

the associative nature allowed us to partition the job, compute each sub-part. and the ~~cotcob~~ combine results to get the final solution.

---

~~sea~~ second order Recurrence:

such as Fibonacci sequence:

$$x_i = \begin{cases} b_0 & i=0 \\ b_1 & i=1 \\ x_{i-1} + x_{i-2} & i \geq 2 \end{cases}$$