2/17/09

L7: Scan and applications

1. Review of last lecture.
   - Scalability: overhead vs. problem size.
   - Isoefficiency
     - Minimum execution time.
       (exists because overhead often a function of # processes)
     - Cost-optimal minimum execution time
   - Scan.

   - Only left nodes are needed.
   - Allowing for in place computation.
   - Pre-scan \rightarrow scan? \rightarrow add element half.

3. Complexity. \rightarrow \mathcal{O}(\frac{n}{p} + 10gp)
   on \( p \) processors
   (demo)

4. Associate operators (such as max, min)

2. Use scan for line of sight. \rightarrow scan
   
   list computation, split, radix sort.
Q. Line of sight [Blev noch 90]

Problem: Determine visible points relative to observation point X on a terrain map.

Steps:
- Compute angle $\alpha$.
- Do a map-scan to find the maximum angle so far.
- Compare angle $\alpha$ with map-scan results.

Q. List comparison:

Problem: Given a data array and a binary array (indicating whether a test has passed or not), produce a computed array that contains only the successful elements.

<table>
<thead>
<tr>
<th>data</th>
<th>$\Rightarrow$</th>
<th>@ scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
<td>$\Rightarrow$</td>
<td>an a ... an</td>
</tr>
</tbody>
</table>

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 3 3 4 5 5 5 ...]
```

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

Steps:
- First scan of valid array
- In parallel, output valid elements to the output buffer indicated by the index.

```
| split: [bleh | bleh | bleh | bleh | bleh]
```

Basically a list comprehension, except we want to also output the invalid elements on the right side of the valid elements.

Example:

```
[1.0, 1.2, 1.3, 1.4, 1.5, 1.6] -> [1.0, 1.1, 1.0, 1.0, ...]
```

Application in `quicksplit`:

```
([1.0, 1.2, 1.3, 1.5, 1.3, 1.4, 1.6] -> [1.0, 1.1, 1.0, 1.0, ...])
```

Can compute the index for valid elements as before. What about invalid elements?
Solution: Side simultaneously outputting

scans

two arrays, one for valid elements
one for invalid.

The invalid array is produced by inverting

scan

the flags.

Another example: classification

and resoring.

\[
\begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & \cdots \\
\end{array}
\]

\[
\begin{array}{cccc}
  3 & 2 & 1 & 2 & 4 & 1 & \cdots \\
\end{array}
\]

\(\ell\) type:

\[
\begin{array}{cccc}
  a_2 & a_5 & a_1 & a_3 & a_0 & a_4 & \cdots \\
  \text{type 1} & \text{type 2} & 3 & 4 & \cdots \\
\end{array}
\]

probably 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

[blehbleh 90]
recurrence equations: \( x_i = f_\ell(x_{i-1}, x_{i-2}, \ldots, x_{i-m}) \)

- prefix sum is just a special kind of recurrence:
  \[
  x_i = x_{i-1} + x_i \quad b_i
  \]
  \( x_0 = b_0 \)

more generally:

first order: \( x_i = \int_{b_0}^{b_i} (x_{i-1} - x_{\text{ai}}) + b_i \quad 0 < i < n \)

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

---

the big picture: this is seemingly a sequential computation, how do we parallelize 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 part is similar:

\[
\begin{align*}
&\ a_2 a_3 (a_1 b_0 + b_1) + (a_3 b_2 + b_3) \\
&\ (a_1 b_0 + b_1) \\
&\ a_2 a_3 (a_1 b_0 + b_1) + (a_3 b_2 + b_3) \\
&\ (a_0, b_0) \\
&\ (a_1, b_1) \\
&\ (a_2, b_2) \\
&\ (a_3, b_3) \\
&\ (a_1 b_0 + b_1) a_2 + b_2 \\
&= a_2 a_3 (a_1 b_0 + b_1) + (a_3 b_2 + b_3)
\end{align*}
\]

Looks like we should keep \( a_2 a_3 \) with the right tree node here.

So, let's construct a pair to keep at every tree node. Start from the leaves.

The update rules are:

\[
\begin{align*}
&\ (a_i, b_i) \cdot (a_j, b_j) \\
&= [a_i \times a_j, b_i \times a_j + b_j]
\end{align*}
\]

So:
\[
\left( a_0 a_1 a_2 a_3, \quad a_2 a_3 (a_1 b_0 + b_1) + (a_3 b_0 + b_3) \right)
\]

Proof that this is correct: prove it is associative.

\[
\left( [a_1, b_1] \cdot [a_2, b_2] \right) \cdot [a_3, b_3]
\]

\[
= [a_1, b_1] \cdot \left( [a_2, b_2] \cdot [a_3, b_3] \right)
\]

Therefore we can do scan of on operation.

(since it's associative)

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

Second order recurrence:

Such as Fibonacci sequence:

\[
\begin{align*}
x_i &= \begin{cases} 
  b_0 & i = 0 \\
  b_1 & i = 1 \\
  x_{i-1} + x_{i-2} & i > 2
\end{cases}
\end{align*}
\]