Parallel Scans & Prefix Sums

COS 326

David Walker
Princeton University

Slide credits: Dan Grossman, UW
http://homes.cs.washington.edu/~djg/teachingMaterials/spac
So far we’ve seen a number of parallel divide-and-conquer algorithms

Today: One more key algorithm

- Parallel prefix:
  - Another “relentlessly sequential” algorithm parallelized
  - And its generalization to a parallel scan

- Application:
  - Parallel quicksort
  - Easy to get a little parallelism
  - With cleverness can get a lot
The prefix-sum problem

val prefix_sum : int array -> int array

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>6</td>
<td>10</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>66</td>
<td>68</td>
<td>76</td>
</tr>
</tbody>
</table>

The simple sequential algorithm: accumulate the sum from left to right

- Sequential algorithm: Work: $O(n)$, Span: $O(n)$
- Goal: a parallel algorithm with Work: $O(n)$, Span: $O(\log n)$
Parallel prefix-sum

The trick: *Use two passes*

- Each pass has $O(n)$ work and $O(\log n)$ span
- So in total there is $O(n)$ work and $O(\log n)$ span

First pass *builds a tree of sums bottom-up*
- the “up” pass

Second pass *traverses the tree top-down to compute prefixes*
- the “down” pass

Historical note:
- Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977
Example

- range 0,8
  - sum 76
    - fromleft

- range 0,4
  - sum 36
    - fromleft

- range 2,4
  - sum 26
    - fromleft

- range 4,6
  - sum 30
    - fromleft

- range 6,8
  - sum 10
    - fromleft

input:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>14</td>
<td>2</td>
</tr>
</tbody>
</table>

output:
Example

```
<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,8</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,8</td>
<td>40</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,8</td>
<td>40</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,8</td>
<td>40</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,8</td>
<td>40</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>range</th>
<th>sum</th>
<th>fromleft</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,8</td>
<td>40</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
1. Up: Build a binary tree where
   – Root has sum of the range \([x, y]\)
   – If a node has sum of \([lo, hi]\) and \(hi > lo\),
     • Left child has sum of \([lo, middle]\)
     • Right child has sum of \([middle, hi]\)
     • A leaf has sum of \([i, i+1]\), i.e., \(input[i]\)

This is an easy parallel divide-and-conquer algorithm: “combine” results by actually building a binary tree with all the range-sums
   – Tree built bottom-up in parallel

Analysis: \(O(n)\) work, \(O(\log n)\) span
The algorithm, pass 2

2. Down: Pass down a value \texttt{fromLeft}
   - Root given a \texttt{fromLeft} of 0
   - Node takes its \texttt{fromLeft} value and
     - Passes its left child the same \texttt{fromLeft}
     - Passes its right child its \texttt{fromLeft} plus its left child’s \texttt{sum}
       - as stored in part 1
   - At the leaf for array position $i$,
     - $\text{output}[i] = \text{fromLeft} + \text{input}[i]$

This is an easy parallel divide-and-conquer algorithm: traverse the tree built in step 1 and produce no result
   - Leaves assign to \texttt{output}
   - Invariant: \texttt{fromLeft} is sum of elements left of the node’s range

Analysis: $O(n)$ work, $O(\log n)$ span
Sequential cut-off

For performance, we need a sequential cut-off:

- **Up:**
  
  just a sum, have leaf node hold the sum of a range

- **Down:**
  
  \[
  \text{output.}(\text{lo}) = \text{fromLeft} + \text{input.}(\text{lo});
  \]
  
  \[
  \text{for } i=\text{lo}+1 \text{ up to } \text{hi}-1 \text{ do}
  \]
  
  \[
  \text{output.}(i) = \text{output.}(i-1) + \text{input.}(i)
  \]
Parallel prefix, generalized

Just as map and reduce are the simplest examples of a common pattern, prefix-sum illustrates a pattern that arises in many, many problems

- Minimum, maximum of all elements \textit{to the left of }\textit{i}.

- Is there an element \textit{to the left of }\textit{i} satisfying some property?

- Count of elements \textit{to the left of }\textit{i} satisfying some property
  - This last one is perfect for an efficient parallel filter ...
  - Perfect for building on top of the “parallel prefix trick”
Parallel Scan

\[ \text{scan} (o) \langle x_1, \ldots, x_n \rangle \]
\[ = \]
\[ \langle x_1, x_1 \circ x_2, \ldots, x_1 \circ \ldots \circ x_n \rangle \]

like a fold, except return the folded prefix at each step

\[ \text{pre\_scan} (o) \text{ base} \langle x_1, \ldots, x_n \rangle \]
\[ = \]
\[ \langle \text{base}, \text{base} \circ x_1, \ldots, \text{base} \circ x_1 \circ \ldots \circ x_{n-1} \rangle \]

sequence with \( o \) applied to all items to the left of index in input
Filter

Given an array **input**, produce an array **output** containing only elements such that \((f \text{ \texttt{elt}})\) is **true**

Example: \(\text{let } f x = x > 10\)

\[
\text{filter } f \text{ \texttt{<17, 4, 6, 8, 11, 5, 13, 19, 0, 24>}} \\
= \text{\texttt{<17, 11, 13, 19, 24>}}
\]

Parallelizable?

- Finding elements for the output is easy
- *But getting them in the right place seems hard*
1. Parallel map to compute a **bit-vector** for true elements
   
   **input**  <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
   
   **bits**   <1, 0, 0, 0, 1, 0, 1, 1, 0, 1>

2. Parallel-prefix sum on the bit-vector
   
   **bitsum**  <1, 1, 1, 1, 2, 2, 3, 4, 4, 5>

3. Parallel map to produce the output
   
   **output**  <17, 11, 13, 19, 24>
Recall quicksort was sequential, in-place, expected time $O(n \log n)$

Best / expected case work

1. Pick a pivot element $O(1)$
2. Partition all the data into:
   A. The elements less than the pivot $O(n)$
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C $2T(n/2)$

How should we parallelize this?
Quicksort

Best / expected case work

1. Pick a pivot element \( O(1) \)
2. Partition all the data into:
   A. The elements less than the pivot \( O(n) \)
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C \( 2T(n/2) \)

Easy: Do the two recursive calls in parallel

- Work: unchanged. Total: \( O(n \log n) \)
- Span: now \( T(n) = O(n) + 1T(n/2) = O(n) \)
We get a $O(\log n)$ speed-up with an *infinite* number of processors. That is a bit underwhelming

- Sort $10^9$ elements 30 times faster

(Some) Google searches suggest quicksort cannot do better because the partition cannot be parallelized

- The Internet has been known to be wrong 😊
- But we need auxiliary storage (no longer in place)
- In practice, constant factors may make it not worth it

Already have everything we need to parallelize the partition...
Parallel partition (not in place)

Partition all the data into:
A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two filters!

– We know a parallel filter is $O(n)$ work, $O(\log n)$ span
– Parallel filter elements less than pivot into left side of aux array
– Parallel filter elements greater than pivot into right size of aux array
– Put pivot between them and recursively sort
– With a little more cleverness, can do both filters at once but no effect on asymptotic complexity

With $O(\log n)$ span for partition, the total best-case and expected-case span for quicksort is

$$T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$$
Example

Step 1: pick pivot as median of three

Step 2a and 2c (combinable): filter less than, then filter greater than into a second array

Step 3: Two recursive sorts in parallel

– Can copy back into original array (like in mergesort)
More Algorithms

• To add multi precision numbers.
• To evaluate polynomials
• To solve recurrences.
• To implement radix sort
• To delete marked elements from an array
• To dynamically allocate processors
• To perform lexical analysis. For example, to parse a program into tokens.
• To search for regular expressions. For example, to implement the UNIX grep program.
• To implement some tree operations. For example, to find the depth of every vertex in a tree
• To label components in two dimensional images.

See Guy Blelloch “Prefix Sums and Their Applications”
• Parallel prefix sums and scans have many applications
  – A good algorithm to have in your toolkit!

• Key idea: An algorithm in 2 passes:
  – Pass 1: build a sum (or “reduce”) tree from the bottom up
  – Pass 2: compute the prefix top-down, looking at the left-subchild to help you compute the prefix for the right subchild