Parallel Prefix Scan

COS 326
Speaker: Andrew Appel
Princeton University

Credits:
Dan Grossman, UW
http://homes.cs.washington.edu/~djg/teachingMaterials/spac
Blelloch, Harper, Licata (CMU, Wesleyan)
The prefix-sum problem

prefix_sum : int seq -> int seq

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 computes the "from-left-of-me" sum

Historical note:

- Original algorithm due to R. Ladner and M. Fischer, 1977
Example

```
<table>
<thead>
<tr>
<th>range</th>
<th>0,4</th>
<th>0,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>36</td>
<td>76</td>
</tr>
<tr>
<td>fromleft</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>0,2</th>
<th>4,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>fromleft</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>2,4</th>
<th>6,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>26</td>
<td>10</td>
</tr>
<tr>
<td>fromleft</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>4,6</th>
<th>6,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>fromleft</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>range</th>
<th>6,8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>fromleft</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
Example

input

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<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>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

output

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>66</td>
<td>68</td>
<td>76</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The algorithm, pass 1

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., nth 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
2. **Down:** Pass down a value `fromLeft`
   - Root given a `fromLeft` of 0
   - Node takes its `fromLeft` value and
     - Passes its left child the same `fromLeft`
     - Passes its right child its `fromLeft` plus its left child’s `sum`
       - as stored in part 1
   - At the leaf for sequence position `i`,
     - `nth output i == fromLeft + nth input i`

This is an easy parallel divide-and-conquer algorithm:
traverse the tree built in step 1 and produce no result
- Leaves create `output`
- Invariant: `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:
  – do a sequential scan
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 *to the left of* \(i\)

- Is there an element *to the left of* \(i\) satisfying some property?

- Count of elements *to the left of* \(i\) 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

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

pre_scan (o) base \langle x_1, \ldots, x_n \rangle 
==
\langle base, base \circ x_1, \ldots, base \circ x_1 \circ \ldots \circ x_{n-1} \rangle

- Operator o must be associative!
- base must be a unit for operator o
- sequence with o applied to all items to the left of index in input
- like a fold, except return the folded prefix at each step
Parallel Filter

Given a sequence **input**, produce a sequence **output** containing only elements \( v \) such that \( (f \ v) \) is **true**

Example: let \( f \ x = x > 10 \)

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

Parallelizable?

- Finding elements for the output is easy
- *But getting them in the right place seems hard*
Use parallel map to compute a bit-vector for true elements:

\[
\text{input} \ <17, 4, 6, 8, 11, 5, 13, 19, 0, 24> \\
\text{bits} \ <1, 0, 0, 0, 1, 0, 1, 1, 0, 1>
\]

Use parallel-prefix sum on the bit-vector:

\[
\text{bitsum} \ <1, 1, 1, 1, 2, 2, 3, 4, 4, 5>
\]

For each i, if bits[i] == 1 then write input[i] to output[bitsum[i]] to produce the final result:

\[
\text{output} \ <17, 11, 13, 19, 24>
\]
QUICKSORT
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 \( \text{work} \)

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 \( T(n) = O(n) + 1T(n/2) = O(n) \)
As with mergesort, 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...

*These days, most hits get this right, and discuss parallel 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 \textit{aux} array
- Parallel filter elements greater than pivot into right size of \textit{aux} array
- Put pivot between them and recursively sort

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

$$T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$$
Step 1: pick pivot as median of three

8 1 4 9 0 3 5 2 7 6

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

1 4 0 3 5 2

1 4 0 3 5 2 6 8 9 7

Step 3: Two recursive sorts in parallel
  – Can copy back into original array (like in mergesort)
More Algorithms

- To add multiprecision 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”
Summary

• 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 "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
PARALLEL COLLECTIONS IN THE "REAL WORLD"
Big Data

If Google wants to index all the web pages (or images or gmasils or google docs or ...) in the world, they have a lot of work to do

- Same with Facebook for all the facebook pages/entries
- Same with Twitter
- Same with Amazon
- Same with ...

Internet has approximately 100 trillion web pages \( (10^{14}) \)

Suppose: server farm with 100 million pages handled per server \( (10^8) \)

Need: 1 million servers \( (10^6) \)

Suppose: average server computer has mean-time-to-failure of 3 years \( (10^3 \text{ days}) \)
Fault tolerance

Internet has approximately 100 trillion web pages \( (10^{14}) \)

Suppose: server farm; 100 million pages handled per server \( (10^8) \)

Need: 1 million servers \( (10^6) \)

Parallel web-indexing algorithm will take a few hours; run it every day.

Suppose: average server computer has mean-time-to-failure of 3 years \( (10^3 \text{ days}) \)

*This was true in 2005 with rotating disks; MTTF probably longer now with SSD

Then: Mean time to first server fail = \( 10^{-3} \text{ days} = 1 \text{ minute} \)

**Impossible to index the web?**
The solution

Build a framework (language, system) for large-scale, many-server, fault-tolerant, big-data parallel programming.

It must be general-purpose (because there are many tasks to do besides web indexing: search, maps, advertising auctions, etc.)

Idea:
Many of these tasks come down to map, filter, fold, reduce, scan
Google MapReduce (2004): a fault tolerant, massively parallel functional programming paradigm

- based on our friends "map" and "reduce"
- Hadoop is the open-source variant
- Database people complain that they have been doing it for a while
  - ... but it was hard to define

Fun stats circa 2012:
- Big clusters were ~4000 nodes
- Facebook had 100 PB in Hadoop
- TritonSort (UCSD) sorts 900GB/minute on a 52-node, 800-disk hadoop cluster
Data Model & Operations

- Map-reduce operates over collections of key-value pairs
  - millions of files (eg: web pages) drawn from the file system
- The map-reduce engine is parameterized by 3 functions:

  ```
  map : key1 * value1 -> (key2 * value2) list
  combine : key2 * (value2 list) -> value2 option
  reduce : key2 * (value2 list) -> key3 * (value3 list)
  ```

optional
Hadoop interfaces:

interface Mapper<K1,V1,K2,V2> {
    public void map (K1 key,
                     V1 value,
                     OutputCollector<K2,V2> output)
    ...
}

interface Reducer<K2,V2,K3,V3> {
    public void reduce (K2 key,
                        Iterator<V2> values,
                        OutputCollector<K3,V3> output)
    ...
}
class WordCountMap implements Map {
    public void map(DocID key,
        List<String> values,
        OutputCollector<String,Integer> output)
    {
        for (String s : values)
            output.collect(s,1);
    }
}

class WordCountReduce {
    public void reduce(String key,
        Iterator<Integer> values,
        OutputCollector<String,Integer> output)
    {
        int count = 0;
        for (int v : values)
            count += 1;
        output.collect(key, count)
    }
}
Architecture

Input Data

Mapper

Local Storage

Reducer

Local Storage

Reducer

Local Storage

Mapper

Output Data

Combine

Map

Shuffle/Sort

Reduce
Iterative Jobs are Common
Jobs, Tasks and Attempts

- A single *job* is split into many *tasks*
- Each *task* may include many calls to map and reduce
- *Workers* are long-running processes that are assigned many tasks
- Multiple workers may *attempt* the same task
  - each invocation of the same task is called an attempt
  - the first worker to finish "wins"
- Why have multiple machines attempt the same task?
  - machines will fail
    - approximately speaking: 5% of high-end disks fail/year
    - if you have 1000 machines: 1 failure per week
    - *repeated failures become the common case*
  - machines can partially fail or be slow for some reason
    - reducers can't start until *all* mappers complete
Flow of Information

- **Worker**
  - Heartbeats
  - Tasks to start
  - Completed

- **Controller**
  - Job config.
  - OK

- **User Program**

The flow of information between these components is as follows:

1. The **Worker** communicates heartbeats to the **Controller**.
2. The **Controller** sends tasks to start to the **Worker**.
3. The **Controller** receives completed tasks from the **Worker**.
4. The **Controller** sends job configuration to the **User Program**.
5. The **Controller** receives an OK signal from the **User Program**.
A Modern Software Stack

Workload Manager

High-level scripting language

Cluster Node

Cluster Node

Cluster Node

Cluster Node

For more: See COS 418, distributed systems
Folds and reduces are easily coded as parallel divide-and-conquer algorithms with $O(n)$ work and $O(\log n)$ span.

Scans are trickier and use a 2-pass algorithm that builds a tree.

The map-reduce-fold paradigm, inspired by functional programming, is a big winner when it comes to big data processing.

Hadoop is an industry standard but higher-level data processing languages have been built on top.
Folds and reduces are easily coded as parallel divide-and-conquer algorithms with $O(n)$ work and $O(\log n)$ span.

Scans are trickier and use a 2-pass algorithm that builds a tree.

The map-reduce-fold paradigm, inspired by functional programming, is a big winner when it comes to big data processing.

Hadoop is an industry standard but higher-level data processing languages have been built on top.

Even though the *local* programming may be “imperative” (in C++, Java, etc.), it must be “as if functional” (no side effects).