More Parallel Primitives and Parallel Sorting
1. More Parallel Primitives

2. Parallel Sorting
Maps and Reductions

Reductions

**INPUT**: An array

**OUTPUT**: A combination of the array by an associative operation

The general name for this type of problem is a **reduction**. Examples include: max, min, has-a, first, count, sorted

Maps

**INPUT**: An array

**OUTPUT**: Apply a function to every element of that array

The general name for this type of problem is a **map**. You can do this with any function, because the array elements are independent.

Today, we’ll add in two more:

- **Scan**
- **Pack (or filter)**

As we’ll see, both of these are quite a bit less intuitive **in parallel** than map and reduce.
Suppose we have an associative operation \( \oplus \) and an array \( a \):

\[
\begin{array}{cccc}
\text{a:} & a_0 & a_1 & a_2 & a_3 \\
\text{a[0]} & \text{a[1]} & \text{a[2]} & \text{a[3]}
\end{array}
\]

Then, \( \text{scan}(a) \) returns an array of “partial sums” (using \( \oplus \)):

\[
\begin{array}{cccc}
\text{scan(a):} & a_0 & a_0 \oplus a_1 & a_0 \oplus a_1 \oplus a_2 & a_0 \oplus a_1 \oplus a_2 \oplus a_3 \\
\text{b[0]} & \text{b[1]} & \text{b[2]} & \text{b[3]}
\end{array}
\]
Scan and Parallel Prefix-Sum

Suppose we have an associative operation $\oplus$ and an array $a$:

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

Then, $\text{scan}(a)$ returns an array of “partial sums” (using $\oplus$):

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

It’s hard to see at first, but this is actually a really powerful tool. It gives us a “partial trace” of the operation as we apply it to the array (for free).
Scan

Suppose we have an associative operation ⊕ and an array a:

\[
\begin{array}{cccc}
a_0 & a_1 & a_2 & a_3 \\
\text{a[0]} & \text{a[1]} & \text{a[2]} & \text{a[3]}
\end{array}
\]

Then, \text{scan}(a) returns an array of “partial sums” (using ⊕):

\[
\begin{array}{cccc}
a_0 & a_0 \oplus a_1 & a_0 \oplus a_1 \oplus a_2 & a_0 \oplus a_1 \oplus a_2 \oplus a_3 \\
\text{b[0]} & \text{b[1]} & \text{b[2]} & \text{b[3]}
\end{array}
\]

It’s hard to see at first, but this is actually a really powerful tool. It gives us a “partial trace” of the operation as we apply it to the array (for free).

No Seriously

splitting, load balancing, quicksort, line drawing, radix sort, designing binary adders, polynomial interpolation, decoding gray codes
Sequential Scan (with $\oplus = +$)

For the sake of being clear, we’ll discuss scan with $\oplus = +$. That is, “prefix sums” of an array:

Example (Prefix Sum)

\[
\begin{array}{cccccc}
\text{a:} & 5 & 1 & 3 & 4 & 2 \\
\text{scan(a):} & 5 & 6 & 9 & 13 & 15 \\
\end{array}
\]

Sequential Code

```java
int[] prefixSum(int[] input) {
    int[] output = new int[input.length];
    int sum = 0;
    for (int i = 0; i < input.length; i++) {
        sum += input[i];
        output[i] = sum;
    }
    return output;
}
```

If you have a really good memory, you’ll remember that on the very first day of lecture, we discussed a very similar problem.
Sequential Code

```java
int[] prefixSum(int[] input) {
    int[] output = new int[input.length];
    int sum = 0;
    for (int i = 0; i < input.length; i++) {
        sum += input[i];
        output[i] = sum;
    }
    return output;
}
```

Bad News

This algorithm does not parallelize well. Step $i$ needs the outputs from all the previous steps. This might as well be an algorithm on a linked list.

So, what do we do?

Come Up With A Better Algorithm!

The solution here will be to add a “pre-processing step”. This is essentially what we did in the first lecture.
We begin with an array as usual:

\[
\begin{array}{cccccc}
\end{array}
\]

Then, transform it into a **balanced tree**, because \( \lg n \) height will allow us to get a span of \( \lg n \), eventually:

```
1 PSTNode {
2    int lo, hi;
3    int sum;
4    PSTNode left, right;
5 }
```
Creating the tree is a standard divide-and-conquer recursive algorithm:

\[
\text{work}(n) = 2\text{work}(n/2) + 1
\]

\[
\text{sum}(n) = 5\text{sum}(n/2) + 1
\]

```java
public class PSTNode {
    int lo, hi;
    int sum;
    PSTNode left, right;
}

PSTNode processInput(int[] input, int lo, int hi) {
    if (hi - lo == 1) {
        return new PSTNode(lo, hi, input[lo]);
    } else {
        mid = lo + (hi - lo)/2;
        PSTNode left = processInput(lo, mid);
        PSTNode right = processInput(mid, hi);
        return new PSTNode(lo, hi, left.sum + right.sum, left, right);
    }
}
```
Now, we have the entire tree filled out:

\[ a = [\ldots, a_5, a_4, a_3, a_2, a_1, a_0, \ldots] \]

To fill in all the prefix sums, we recursively fill them in down the tree. Since the non-leaf nodes don’t have access to the elements of the array, we fill in a **pre-scan** (everything up to, but not including the range).
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```java
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the \textbf{pre-scans}, we recursively fill them in down the tree:

```cpp
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the pre-scans, we recursively fill them in down the tree:

```plaintext
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
Better Prefix-Sum: Constructing the Output

To fill in all the **pre-scans**, we recursively fill them in down the tree:
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```cpp
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```python
void makeOutput(int[] output, PSTNode current, int prescan) {
  if (current is a leaf) {
    output[current.lo] = prescan + current.sum;
  } else {
    makeOutput(output, current.left, prescan);
    makeOutput(output, current.right, prescan + current.left.sum);
  }
}
```
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the **pre-scans**, we recursively fill them in down the tree:
To fill in all the **pre-scans**, we recursively fill them in down the tree:
Better Prefix-Sum: Constructing the Output

To fill in all the **pre-scans**, we recursively fill them in down the tree:

```c
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    } else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
To fill in all the **pre-scans**, we recursively fill them in down the tree:

```c
void makeOutput(int[] output, PSTNode current, int prescan) {
    if (current is a leaf) {
        output[current.lo] = prescan + current.sum;
    }
    else {
        makeOutput(output, current.left, prescan);
        makeOutput(output, current.right, prescan + current.left.sum);
    }
}
```
Adding a sequential cut-off isn’t too bad:

1. \( \text{output}[lo] = \text{prescan} + \text{input}[lo] \); 

2. for \( i = lo + 1; i < hi; i++ \) { 
   \[ \text{output}[i] = \text{output}[i-1] + \text{input}[i] \]
   
3. Notice that this means we must pass the \( \text{input} \) array to this phase now.
Adding a sequential cut-off isn’t too bad:

**Processing the Input**

This is just a normal sequential cut-off. The leaves end up being cutoff size ranges instead of ranges of one.
Adding a sequential cut-off isn't too bad:

### Processing the Input

This is just a normal sequential cut-off. The leaves end up being cutoff size ranges instead of ranges of one.

### Constructing the Output

We must sequentially compute the prefix sum at our leaves as well:

```plaintext
1. output[lo] = prescan + input[lo];
2. for (i = lo + 1; i < hi; i++) {
    3.     output[i] = output[i-1] + input[i]
3. }
```

Notice that this means we must pass the input array to this phase now.
Here the idea is that we’d like to filter the array given some predicate (e.g., $\leq 7$). More specifically:

**Pack/Filter**

Suppose we have a function $f : E \rightarrow \text{boolean}$ and an array $a$ of type $E$:

$$a: \begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
  \end{array}$$

Then, $\text{pack}(a)$ returns an array of elements $x$ for which $f(x) = \text{true}$. For example, if $\text{arr} = [1, 3, 8, 6, 7, 2, 4, 9]$ and $f(x) = x \% 2 == 0$, then $\text{pack}(\text{arr}) = [8, 6, 2, 4]$. 

The key to doing this in parallel is scan!
Let \( f(x) = x \% 2 == 0 \).

### Parallel Pack

**input:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Use a map to compute a bitset for \( f(x) \) applied to each element:

**bitset:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Do a scan on the bit vector with \( \oplus = + \):

**bitsum:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Do a map on the bit sum to produce the output:

**output:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output = new E[bitsum[n - 1]];
Another Primitive: Parallel Pack (or “filter”)

Let \( f(x) = x \mod 2 == 0 \).

### Parallel Pack

**input:**

\[
\begin{array}{cccccccc}
1 & 3 & 8 & 6 & 7 & 2 & 4 & 9 \\
\end{array}
\]

1. Use a **map** to compute a bitset for \( f(x) \) applied to each element.

**bitset:**

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

```java
for (i = 0; i < input.length; i++) {
    if (bitset[i] == 1) {
        output[bitsum[i] - 1] = input[i];
    }
}
```
Let $f(x) = x \mod 2 == 0$.

1. Use a **map** to compute a bitset for $f(x)$ applied to each element.

   - **bitset:**
     
     |------|------|------|------|------|------|------|------|
     | 0    | 0    | 1    | 1    | 0    | 1    | 1    | 0    |

2. Do a **scan on the bit vector** with $\oplus = +$:

   - **bitsum:**
     
     |------|------|------|------|------|------|------|------|
     | 0    | 0    | 1    | 2    | 2    | 3    | 4    | 4    |

3. Do a **map** on the bit sum to produce the output:

   - **output:**
     
     |------|------|------|------|
     | 1    | 8    | 6    | 2    |
Another Primitive: Parallel Pack (or “filter”)

Let \( f(x) = x \mod 2 == 0 \).

### Parallel Pack

1. Use a **map** to compute a bitset for \( f(x) \) applied to each element
   
   \[
   \text{bitset: } \begin{bmatrix}
   0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
   \end{bmatrix}
   \]

2. Do a **scan** on the bit vector with \( \oplus = + \):
   
   \[
   \text{bitsum: } \begin{bmatrix}
   0 & 0 & 1 & 2 & 2 & 3 & 4 & 4 \\
   \end{bmatrix}
   \]

3. Do a **map** on the bit sum to produce the output:
   
   \[
   \text{output: } \begin{bmatrix}
   8 & 6 & 2 & 4 \\
   \end{bmatrix}
   \]
Let \( f(x) = x \mod 2 == 0 \).

**Parallel Pack**

1. Use a **map** to compute a bitset for \( f(x) \) applied to each element:

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>6</th>
<th>7</th>
<th>2</th>
<th>4</th>
<th>9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>bitset</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

2. Do a **scan** on the bit vector with \( \oplus = + \):

<table>
<thead>
<tr>
<th>bitsum</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
</table>

3. Do a **map** on the bit sum to produce the output:

<table>
<thead>
<tr>
<th>output</th>
<th>8</th>
<th>6</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
</table>

```java
output = new E[bitsum[n-1]];
for (i=0; i < input.length; i++) {
    if (bitset[i] == 1) {
        output[bitsum[i] - 1] = input[i];
    }
}
```
We can combine the first two passes into one (just use a different base case for prefix sum)

We can also combine the third step into the second part of prefix sum

Overall:
More on Pack

- We can combine the first two passes into one (just use a different base case for prefix sum)

- We can also combine the third step into the second part of prefix sum

- Overall: $O(n)$ work and $O(\lg n)$ span. (Why?)

We can use scan and pack in all kinds of situations!
Parallel Quicksort

```java
int[] quicksort(int[] arr) {
    int pivot = choosePivot();
    int[] left = filterLessThan(arr, pivot);
    int[] right = filterGreaterThan(arr, pivot);
    return quicksort(left) + quicksort(right);
}
```

Do The Recursive Calls in Parallel

Assuming a good pivot, we have:

\[
\text{work}(n) =
\]
Parallel Quicksort

int[] quicksort(int[] arr) {
    int pivot = choosePivot();
    int[] left = filterLessThan(arr, pivot);
    int[] right = filterGreaterThan(arr, pivot);
    return quicksort(left) + quicksort(right);
}

Do The Recursive Calls in Parallel

Assuming a good pivot, we have:

work(n) = \begin{cases} 
    O(1) & \text{if } n = 1 \\
    2\text{work}(n/2) + O(n) & \text{otherwise}
\end{cases}

and

span(n) =
Parallel Quicksort

```plaintext
int[] quicksort(int[] arr) {
    int pivot = choosePivot();
    int[] left = filterLessThan(arr, pivot);
    int[] right = filterGreaterThan(arr, pivot);
    return quicksort(left) + quicksort(right);
}
```

Do The Recursive Calls in Parallel

Assuming a good pivot, we have:

\[
\text{work}(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
2\text{work}(n/2) + O(n) & \text{otherwise}
\end{cases}
\]

and

\[
\text{span}(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
\max(\text{span}(n/2), \text{span}(n/2)) + O(n) & \text{otherwise}
\end{cases}
\]

These solve to $O(n \lg n)$ and $O(n)$. So, the parallelism is $O(\lg n)$. 
int[] quicksort(int[] arr) {
    int pivot = choosePivot();
    int[] left = filterLessThan(arr, pivot);
    int[] right = filterGreaterThan(arr, pivot);
    return quicksort(left) + quicksort(right);
}

Do The Partition in Parallel

The partition step is just two filters or packs. Each pack is $O(n)$ work, but $O(\log n)$ span! So, our new span recurrence is:

$$span(n) =$$
```java
int[] quicksort(int[] arr) {
    int pivot = choosePivot();
    int[] left = filterLessThan(arr, pivot);
    int[] right = filterGreaterThan(arr, pivot);
    return quicksort(left) + quicksort(right);
}
```

**Do The Partition in Parallel**

The partition step is just two filters or packs. Each pack is $O(n)$ work, but $O(\log n)$ span! So, our new span recurrence is:

$$
\text{span}(n) = \begin{cases} 
    O(1) & \text{if } n = 1 \\
    \max(\text{span}(n/2), \text{span}(n/2)) + O(\log n) & \text{otherwise}
\end{cases}
$$

Master Theorem says this is $O(\log^2 n)$ which is neat!
Do The Recursive Calls in Parallel

This will get us the same work and span we got for quicksort when we did this:

- $\text{work}(n) = \mathcal{O}(n \lg n)$
- $\text{span}(n) = \mathcal{O}(n)$
- Parallelism is $\mathcal{O}(\lg n)$

Now, let’s try to parallelize the merge part.
Parallel Mergesort

```java
int[] mergesort(int[] arr) {
    int[] left = getLeftHalf();
    int[] right = getRightHalf();
    return merge(mergesort(left), mergesort(right));
}
```

Do The Recursive Calls in Parallel

This will get us the same work and span we got for quicksort when we did this:

- `work(n) = \mathcal{O}(n \lg n)`
- `span(n) = \mathcal{O}(n)`
- Parallelism is \( \mathcal{O}(\lg n) \)

Now, let's try to parallelize the merge part.

As always, when we want to parallelize something, we can turn it into a divide-and-conquer algorithm.
Parallelizing Merge

Do The Merge in Parallel

Merge takes as input two arrays:

|--------|--------|--------|--------|--------|

Find the median of the larger array (just the middle index): X

Partition the smaller array using X as a pivot. To do this, binary search the smaller array: Y

Now, we have four pieces ≤ X, > X, ≤ Y, and > Y. In the sorted array, the ≤ pieces will be entirely before the > pieces.

Recursively apply the merge algorithm (until some cut-off)!
Do The Merge in Parallel

Merge takes as input two arrays:

1. Find the median of the \textbf{larger} array (just the middle index):

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Do The Merge in Parallel

Merge takes as input two arrays:

1. Find the median of the larger array (just the middle index):

2. Partition the smaller array using X as a pivot. To do this, binary search the smaller array:
Parallelizing Merge

Do The Merge in Parallel

Merge takes as input two arrays:

1. Find the median of the larger array (just the middle index):

2. Partition the smaller array using X as a pivot. To do this, binary search the smaller array:

3. Now, we have four pieces $\leq X$, $> X$, $\leq Y$, and $> Y$. In the sorted array, the $\leq$ pieces will be entirely before the $>$ pieces.
Parallelizing Merge

Do The Merge in Parallel

Merge takes as input two arrays:

1. Find the median of the larger array (just the middle index):

2. Partition the smaller array using $X$ as a pivot. To do this, binary search the smaller array:

3. Now, we have four pieces $\leq X$, $> X$, $\leq Y$, and $> Y$. In the sorted array, the $\leq$ pieces will be entirely before the $>$ pieces.

4. Recursively apply the merge algorithm (until some cut-off)!
First, we analyze just the parallel merge:

Parallel Merge Analysis

The non-recursive work is
First, we analyze just the parallel merge:

Parallel Merge Analysis

The non-recursive work is $\mathcal{O}(1) + \mathcal{O}(\lg n)$ to find the splits.

The worst case is when we split the bigger array in half and the smaller array is all on the left (or all on the right). In other words:

$$\text{work}(n) \leq$$
First, we analyze **just the parallel merge**:

**Parallel Merge Analysis**

The non-recursive work is $O(1) + O(\lg n)$ to find the splits.

The **worst case** is when we split the bigger array in half and the smaller array is all on the left (or all on the right). In other words:

$$work(n) \leq \begin{cases} 
O(1) \quad & \text{if } n = 1 \\
work(3n/4) + work(n/4) + O(\lg n) \quad & \text{otherwise}
\end{cases}$$

and

$$\text{span}(n) \leq$$
First, we analyze just the parallel merge:

Parallel Merge Analysis

The non-recursive work is $O(1) + O(\lg n)$ to find the splits.

The worst case is when we split the bigger array in half and the smaller array is all on the left (or all on the right). In other words:

$$\text{work}(n) \leq \begin{cases} O(1) & \text{if } n = 1 \\ \text{work}(3n/4) + \text{work}(n/4) + O(\lg n) & \text{otherwise} \end{cases}$$

and

$$\text{span}(n) \leq \begin{cases} O(1) & \text{if } n = 1 \\ \max(\text{span}(3n/4) + \text{span}(n/4)) + O(\lg n) & \text{otherwise} \end{cases}$$

These solve to $\text{work}(n) = O(n)$ and $\text{span}(n) = O(\lg^2 n)$. 
Now, we calculate the work and span of the entire parallel mergesort.

Putting It Together

\[
\begin{align*}
\text{work} (n) &= \mathcal{O} (n \lg n) \\
\text{span} (n) &\leq \begin{cases} 
\mathcal{O}(1) & \text{if } n = 1 \\
\text{span} (n/2) + \mathcal{O}(\lg^2 n) & \text{otherwise}
\end{cases}
\end{align*}
\]

This works out to \( \text{span} (n) = \mathcal{O}(\lg^3 n) \).

This isn’t quite as much parallelism as quicksort, but this one is a worst case guarantee!