Announcements

• Project 3 PartA due Thursday night
Recap

Last week
  – simple parallel programs
  – common patterns: map, reduce
  – analysis tools (work, span, parallelism)
  – Amdahl’s Law

Now
  – parallel quicksort, merge sort
  – useful building blocks: prefix, pack
Parallelizable?

Fibonacci (N)

\[ \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \]
\[ = 1 \text{ if } N = 1, 2 \]

Not really because of sequential dependencies.

Naive approach scales \( 2^n \) threads \( \rightarrow \) exponential time.
Parallelizable?

Prefix-sum:

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>6</td>
<td>9</td>
<td>20</td>
<td>30</td>
<td>38</td>
<td>40</td>
<td>47</td>
<td>55</td>
</tr>
</tbody>
</table>

\[
output[i] = \sum_{0}^{i-1} input[i]
\]
First Pass: Sum

Sum \([0,7]\): 35

6  3  11  10  8  2  7  8
First Pass: Sum

```
Sum [0,7]: 55

Sum [0,3]: 30
Sum [2,3]: 21
Sum [4,5]: 10
Sum [5,7]: 18

6 3 11 10 8 2 7 8
```
2nd Pass: Use Sum for Prefix-Sum

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>11</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
```
2nd Pass: Use Sum for Prefix-Sum

Go from root down to leaves

Root
- \text{sum<0} = \text{0}

Left-child
- \text{sum}<K = \text{parents sum} < \text{K}

Right-child
- \text{sum}<K = \text{parents sum} < \text{K} + \text{siblings sum}[L..K]
Prefix-Sum Analysis

• First Pass (Sum):
  – span = $O(\log n)$

• Second Pass:
  – single pass from root down to leaves
    • update children’s sum<K value based on parent and sibling
  – span = $O(\log n)$

• Total
  – span = $O(\log n)$
Parallel Prefix, Generalized

Prefix-sum is another common pattern (prefix problems)
  – maximum element 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
  – ...

We can solve all of these problems in the same way
Pack:

```
input:  6  3  11  10  8  2  7  8  
output: 6  3  2  7  
```

test: $X < 8$?

Output array of elements satisfying test, in original order
Parallel Pack?

Pack

input  | output  | test: $X < 8$?
---|---|---
6 3 11 10 8 2 7 8 | 6 3 2 7 |

• Determining **which** elements to include is **easy**
• Determining **where** each element goes in output is **hard**
  – seems to depend on previous results
Parallel Pack

1. map test input, output [0,1] bit vector

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

test: $X < 8$?
Parallel Pack

1. map test input, output [0,1] bit vector

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2. transform bit vector into array of indices into result array

| pos   | 1 | 2 | 2 | 2 | 3 | 4 | 4 |
Parallel Pack

1. map test input, output [0,1] bit vector

\[
\begin{array}{cccccccc}
\text{input} & 6 & 3 & 11 & 10 & 8 & 2 & 7 & 8 \\
\text{test} & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

2. prefix-sum on bit vector

\[
\begin{array}{cccccccc}
\text{pos} & 1 & 2 & 2 & 2 & 2 & 3 & 4 & 4 \\
\end{array}
\]

3. map input to corresponding positions in output

\[
\begin{array}{cccc}
\text{output} & 6 & 3 & 2 & 7 \\
\end{array}
\]

- if (test[i] == 1) output[pos[i]] = input[i]
Parallel Pack Analysis

• Parallel Pack
  1. map: \( O(\quad) \) span
  2. sum-prefix: \( O(\quad) \) span
  3. map: \( O(\quad) \) span

• Total: \( O(\quad) \) span
Sequential Quicksort

Quicksort (review):
1. Pick a pivot \( O(1) \)
2. Partition into two sub-arrays \( O(n) \)
   - A. values less than pivot
   - B. values greater than pivot
3. Recursively sort A and B \( 2T(n/2), \text{ avg} \)

Complexity (avg case)
- \( T(n) = n + 2T(n/2) \) \( T(0) = T(1) = 1 \)
- \( O(n \log n) \)

How to parallelize?
Parallel Quicksort

Quicksort

1. Pick a pivot \( O(1) \)
2. Partition into two sub-arrays \( O(n) \)
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B in parallel \( T(n/2) \), avg

Complexity (avg case)

- \( T(n) = n + T(n/2) \) \( T(0) = T(1) = 1 \)
- Span: \( O( ) \)
- Parallelism (work/\( \text{span} \)) = \( O( ) \)
Taking it to the next level…

• $O(\log n)$ speed-up with infinite processors is okay, but a bit underwhelming
  – Sort $10^9$ elements 30x faster

• Bottleneck:
Parallel Partition

Partition into sub-arrays

A. values less than pivot
B. values greater than pivot

What parallel operation can we use for this?
Parallel Partition

• Pick pivot

\[ 8 \ 1 \ 4 \ 9 \ 0 \ 3 \ 5 \ 2 \ 7 \ 6 \]

• Pack (test: <6)

\[ 1 \ 4 \ 0 \ 3 \ 5 \ 2 \]

• Right pack (test: >=6)

\[ 1 \ 4 \ 0 \ 3 \ 5 \ 2 \ 6 \ 8 \ 9 \ 7 \]
Parallel Quicksort

Quicksort

1. Pick a pivot \( O(1) \)
2. Partition into two sub-arrays \( O(\ ) \) span
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B in parallel \( T(n/2), \text{avg} \)

Complexity (avg case)

- \( T(n) = O(\ ) + T(n/2) \) \( T(0) = T(1) = 1 \)
- Span: \( O(\ ) \)
- Parallelism (work/span) = \( O(\ ) \)
Sequential Mergesort

Mergesort (review):
1. Sort left and right halves \( 2T(n/2) \)
2. Merge results \( O(n) \)

Complexity (worst case)
- \( T(n) = n + 2T(n/2) \) \( T(0) = T(1) = 1 \)
- \( O(n \log n) \)

How to parallelize?
- Do left + right in parallel, improves to \( O(n) \)
- To do better, we need to…
Parallel Merge

How to merge two sorted lists in parallel?
Parallel Merge

1. Choose median $M$ of left half $O(\quad)$
2. Split both arrays into $< M, \geq M$ $O(\quad)$
   - how?
Parallel Merge

1. Choose median M of left half
2. Split both arrays into < M, >=M
   - how?
3. Do two submerges in parallel
When we do each merge in parallel:
- we split the bigger array in half
- use binary search to split the smaller array
- And in base case we copy to the output array
Parallel Mergesort Pseudocode

Merge(arr[], left₁, left₂, right₁, right₂, out[], out₁, out₂)
  int leftSize = left₂ - left₁
  int rightSize = right₂ - right₁
  // Assert: out₂ - out₁ = leftSize + rightSize
  // We will assume leftSize > rightSize without loss of generality

  if (leftSize + rightSize < CUTOFF)
    sequential merge and copy into out[out₁..out₂]

  int mid = (left₂ - left₁)/2
  binarySearch arr[right₁..right₂] to find j such that
    arr[j] ≤ arr[mid] ≤ arr[j+1]

  Merge(arr[], left₁, mid, right₁, j, out[], out₁, out₁+mid+j)
  Merge(arr[], mid+1, left₂, j+1, right₂, out[], out₁+mid+j+1, out₂)
Analysis

Parallel Merge (worst case)
- Height of partition call tree with n elements: $O(\ )$
- Complexity of each thread (ignoring recursive call): $O(\ )$
- Span: $O(\ )$

Parallel Mergesort (worst case)
- Span: $O(\ )$
- Parallelism (work / span): $O(\ )$

Subtlety: uneven splits
- but even in worst case, get a 3/4 to 1/4 split
  - still gives $O(\log n)$ height
Parallel Quicksort vs. Mergesort

Parallelism (work / span)

- quicksort: $O(n / \log n)$ \hspace{1cm} \text{avg case}
- mergesort: $O(n / \log^2 n)$ \hspace{1cm} \text{worst case}