Analyzing Parallel Programs

Let $T_p$ be the running time on $P$ processors

Two key measures of run-time:
- **Work**: How long it would take 1 processor = $T_1$
- **Span**: How long it would take infinity processors = $T_\infty$

**Speed-up** on $P$ processors: $T_1 / T_p$

Amdahl's Fairly Trivial Observation

- Most programs have
  1. parts that parallelize well
  2. parts that don't parallelize at all

- Let $S =$ proportion that can't be parallelized, and normalize $T_1$ to 1
  
  $$1 = T_1 = S + (1 - S)$$

- Suppose we get perfect linear speedup on the parallel portion:
  $$T_p = S + (1-S)/P$$

- So the overall speed-up on $P$ processors is
  (Amdahl's Law): $T_1 / T_p = 1 / (S + (1-S)/P)$
  $$T_1 / T_\infty = 1 / S$$

Results from Friday

- Parallel Prefix
  - $O(N)$ Work
  - $O(\log N)$ Span

- Quicksort
  - Partition can be solved with Parallel Prefix
  - Overall result
    - $O(N \log N)$ work, $O(\log^2 N)$ Span

Prefix-sum:

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>6</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>36</td>
<td>40</td>
<td>47</td>
<td>55</td>
</tr>
</tbody>
</table>

Prefix sum:

```
6 3 11 10 6 2 7 8
```

```
4 2 1 0 5 3 7 6
```

```
1 0 3 2 6 5 4 7
```

```
0 0 0 0 1 1 1 1
```

```
0 0 0 0 0 0 1 1
```

```
0 0 0 0 0 0 0 1
```
Parallel Partition
- Pick pivot
  8 1 4 9 0 3 5 2 7 6
- Pack (test: <6)
  1 4 3 5 2
- Right pack (test: >=6)
  5 4 0 3 5 2 8 9 7

Parallel Quicksort
Quick sort
1. Pick a pivot \( O(1) \)
2. Partition into two sub-arrays \( O(\log n) \) 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)
- \( T(n) = O(\log n) + T(n/2) \)
- Span: \( O(\log^2 n) \)
- Parallelism (work/span) = \( O( n / \log n ) \)

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
1. Choose median M of left half \( O( ) \)
2. Split both arrays into < M, >=M \( O( ) \)
   - how?
3. Do two submerges in parallel
Parallel Mergesort Pseudocode

```c
Parallel Mergesort Pseudocode

Mergesort(arr[], left, right, out)
int leftSize = right - left;
int rightSize = rightSize;
if (leftSize + rightSize < CUTOFF)
    sequential merge and copy into out[left..right]
else
    int mid = (left + right)/2;
    binarySearch arr[left..right] to find j such that
    arr[j] ≤ arr[mid] ≤ arr[j+1];
    Merge(arr[], left, mid, right, j, out)
    Merge(arr[], mid+1, right, j+1, out)
```

Analysis

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

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

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: $O(n / \log n)$ avg case
- mergesort: $O(n / \log^2 n)$ worst case