Recap analysis of parallel programs (tasks or loops)

- Sequential execution time (aka work performed by the program/algorithm)
  - Depends on parameter $N$ (input size)
- Parallel execution time
  - Depends on both $N$ and $p$ (number of threads/processors/cores)
- Best case parallel execution time (aka critical path or depth)
- General rule of thumb $N$ grows much faster than $p$
- Link to Amdahl’s Law
Inherently Sequential Programs

```
// foo is some function with no (internal parallelism)
Y = X[0];
for (i=1; i<N; i++){
    Y = foo(Y, X[i]);
}
```

- Impossible to parallelize
  - Result of the previous iteration is needed in order to start the next one
- Under what conditions (properties of foo), is parallelization possible

Another cousin (scan)

```
// foo is some function with no (internal parallelism)
Y[0] = X[0];
for (i=0; i<N; i++){
    Y[i] = foo(Y[i], X[i]);
}
```

- What if foo is associative?
  - That's all that's needed
Divide & Conquer: 
expose parallelism

- First a simple problem: reduction (only last answer needed)
  \[ Y = X[0]; \]
  \[ \text{for } (i=0; i<N; i++) \]
  \[ Y = \text{foo}(Y, X[i]); \]
  \[ Y = \text{reduce(lo, hi)} \]
  \[ \text{reduce(lo, hi)} \{ \]
  \[ \text{if } (lo=hi) \text{ return } X[lo]; \]
  \[ \text{else } \{ \]
  \[ \text{mid} = (lo+hi)/2; \]
  \[ \text{left} = \text{reduce(lo, mid); } \]
  \[ \text{right} = \text{reduce(mid+1, hi); } \]
  \[ \text{return } \text{foo(left, right); } \]

Analysis of D&C reduction

- Work complexity: \( O(n) \)
- Span (critical path): \( \lg n \)
- On a real machine (OpenMP): \( n/p \)
- Throttle parallelism with if clause
  - From the bottom, e.g., if \( (hi-lo < 2000) \)
  - From the top e.g., if \( (“# siblings < p”) \)
    - Exercise to solve in PA2
D&C scan: extra work to expose parallelism

\[Y[0] = X[0];\]
\[
\text{for } (i=1; i<N; i++)
\]
\[Y[i] = \text{foo}(Y, X[i]);\]

\[Y = \text{scan}(lo, hi)\]
\[
\text{scan}(lo, hi)\{
\]
\[\text{if } (lo=hi) \{Y[lo] = X[lo]; \text{return}\}
\]
\[\text{else } \{
\]
\[\text{mid} = (lo + hi)/2;
\]
\[Y[lo : \text{mid}] = \text{scan}(lo, \text{mid});
\]
\[Y[\text{mid}+1 : hi] = \text{scan}(\text{mid}+1, hi);
\]
\[Y[\text{mid}+1 : hi] = \text{foo}(Y[\text{mid}], Y[\text{mid}+1 : hi]);
\]
\[\text{return } Y;\}
\]

Parallel D&C Scan

\[Y = \text{reduce}(lo, hi)\]
\[
\text{reduce}(lo, hi)\{
\]
\[\text{if } (lo=hi) \text{ return } X[lo];
\]
\[\text{else } \{
\]
\[\text{mid} = (lo+hi)/2;
\]
\[\#\text{pragma omp task}
\]
\[Y[lo : \text{mid}] = \text{scan}(lo, \text{mid});
\]
\[Y[\text{mid}+1 : hi] = \text{scan}(\text{mid}+1, hi);
\]
\[\#\text{pragma omp taskwait}
\]
\[\#\text{pragma omp parallel for}
\]
\[Y[\text{mid}+1 : hi] = \text{foo}(Y[\text{mid}], Y[\text{mid}+1 : hi]);
\]
\]
Analysis of D&C scan

Same recursion pattern, work is done “on the way up” after the children return

- Work complexity: $O(n \ lg \ n)$
- Span (critical path): $\lg n$
  - if the calls to foo are in a parallel for
- In practice (OpenMP): $(n \ lg \ n)/p$
  - must throttle (from the root) for efficiency

Colorado State University