Lecture 20:

Parallel Algorithms

- Performance Measures
- Scheduling
  - lower bounds
  - greedy
- Examples
  - Matrix Multiply
  - Merging sorted lists

Announcement: Quiz 2 is graded. Remember: do NOT discuss it yet!
See Stellar for scores/stats, after lecture.
Contact course staff for questions if you are concerned.
- Yu: problems 1, 3
- Rafael: problems 2, 4
What if you have "lots of help"?

Some problems are amenable to "teamwork"

Digging a ditch

Digging a hole

... others are not...

Our questions today:

- How does it help speed up computation?
- When doesn't it help?

But first we need a model:

Memory: shared or distributed?

↑ today (also multicore, so every new laptop/desktop)
How many processors?

Should the program know this information?

Could it change during execution?

today: assume \textit{polynomial in input size}

let \textit{"dynamic multithreading" platform}

take care of scheduling, communication, load balancing details.

- allow

\textit{"nested parallelism"}

\underline{\textit{Spawn}} processes

\underline{\textit{Sync}}

\textit{"parallel loops"}
Example: Adding $n$ integers $X_1 \ldots X_n$

DAG representation:

- **Initial call**
  - $\sum_{i=1}^{n} X_i$

- **Recursive calls**
  - $\sum_{i=1}^{n} X_i$
  - $\sum_{i=n+1}^{2n-1} X_i$
  - \ldots

- **log $n$ levels**

At each node, $O(1)$ time to add results of recursive calls

Each node of DAG represents $O(1)$ computation

Each level of the DAG can be computed in parallel

$\Rightarrow O(\log n)$ parallel time with $n$ processors

Work per level:
- $O(1)$
- $O(2)$
- $O(4)$
- \ldots

Max work per level $\rightarrow O(n)$

Total work $O(n)$

In general - length of longest path in DAG corresponds to parallel time
Performance Measures

**Work**: time to perform computation on one processor

\[ \text{work} = O(n) \]

**Span**: longest execution path on DAG

\[ \text{span} = O(\log n) \]

In example:

\[ \text{work} = O(n) \]

\[ \text{span} = O(\log n) \]

Running time on \( P \) processors: \( T_p \)

Some obvious lower bounds:

1) "Work law" - \( T_p \geq \frac{T_1}{P} \)

i.e., at each step can do at most \( P \) work in \( T_p \) steps, can do at most \( P \cdot T_p \)

2) "Span law" - \( T_p \geq T_\infty \)
Speedup: \( \frac{T_i}{T_p} \) = how many times faster with \( P \) processors

In example:
\[ \frac{T_i}{T_n} = \frac{n}{\log n} \]

Linear speedup: ("Holy Grail")
\[ \frac{T_i}{T_p} = \Theta(P) \]

Parallelism:
\[ \frac{T_i}{T_\infty} \]

In example:
\[ \frac{T_i}{T_\infty} = \frac{n}{\log n} \]

- average amount of work per step
- max possible achievable speedup
- limit on speedup
Scheduling:

In practice:
do not always know computation DAG
in advance, i.e., "spawned threads"
distributed control vs. centralized control
better for practice
more complicated

Greedy scheduler:
assign as many strands as possible at each time

Thm. For greedy schedule $T_p \leq \frac{T_i}{p} + T_{oo}$

Pf.
complete step: all processors compute

Claim: $\leq \left\lfloor \frac{T_i}{p} \right\rfloor$ complete steps
(since o.w. total work of complete steps

$\geq p \cdot (\left\lfloor \frac{T_i}{p} \right\rfloor + 1) = p \left\lfloor \frac{T_i}{p} \right\rfloor + p$

$= T_i - (T_i \mod p) + p$

$\geq T_i$ (which contradicts that total work $= T_i$)

Claim: incomplete step reduces span by 1 (since executes all strands)

$\Rightarrow \leq T_{oo}$ in complete steps

ie. $\leq$ strands $< \leq$ processors
Another Example: Matrix Multiply

\[ A \times B = C \]

**Idea:**
\[ C_{ij} = \sum_k A_{ik} B_{kj} \]
- Compute sum as before \(O(\log n)\) span

**Algorithm:**
- For all \(i, k, j\) in parallel, write \(A_{ik} \times B_{kj}\) to location \((i, k, j)\)
- For all \(i, j\) in parallel
  - Compute \(C_{ij} = \sum_k A_{ik} B_{kj}\) via sum routine
  - \(O(\log n)\) span

**Span:** \(O(\log n)\)
**Parallelism:** \(\frac{N^3}{\log n}\)

Note: Can also do Boolean matrix multiply in \(O(1)\) span. How?

A graph theoretic problem! \(\Rightarrow\) if \(A\) is an adjacency matrix, \((A + I)^n\) is transitive closure of \(A\), which can be computed in \(O(\log n)\) span. HOW?
Another example:

**Merging sorted lists**

Given: two sorted lists $T_1[1..n_1] + T_2[1..n_2]$ (wlog $n_1 \geq n_2$)

Output: sorted list $T[1..(n_1+n_2)]$ which is merge of $T_1 + T_2$

**Algorithm:**

- $q_1 \leftarrow \lfloor n_1/2 \rfloor$
- $X \leftarrow T_1[q_1] \quad \triangleright \text{find median of } T_1$
- binary search $T_2$ to find $q_2$ s.t. $T_2[q_2-1] \leq X \leq T_2[q_2]$
- Recursively merge $T_1[1..q_1-1] + T_2[1..q_2-1]$ and put result into $T[1..(q_1+q_2)-2]$
- Recursively merge $T_1[q_{1+1}..n_1] + T_2[q_2..n_2]$ and put result into $T[q_{1+q_2}..(n_1+n_2)]$
- $T[q_1+q_2-1] \leftarrow X$

(can do in parallel)
Idea behind Algorithm:

\[ T_1 \quad \text{Merge } A_{1B} \quad T_2 \quad \text{Merge } C_{1D} \]

\[ T \quad \text{Merge of } A_{1B} \quad X \quad \text{Merge of } C_{1D} \]

Running Time?

"Generalized DAG" of computation

merge \( T_1 \) + \( T_2 \) of size \( n_1, n_2 \)

merge \( T_1^L, T_2^L \) of size \( n_1^L, n_2^L \)

merge \( T_1^K, T_2^K \) of size \( n_1^K, n_2^K \)

(1) depth?

(2) runtime per node
Parallel runtime analysis of merge:

Main observation: let \( n = n_1 + n_2 \)
size of list in recursive calls each \( \leq 3n/4 \)

why?
\( n_1 \geq n_2 \)
recursive calls are on at most \( \frac{1}{2} \) of \( T_1 \)

size of recursive call
\[
\leq n_2 + \frac{1}{2} n_1 \\
\leq \frac{n_1 + n_2 + n_2}{2} \\
\leq \frac{n}{2} + \frac{n}{4} \\
\leq \frac{3n}{4} \quad \blacksquare
\]

Worst case span
\[
P_\infty(n) = P_\infty(3n/4) + \Theta(\log n) \\
= \Theta(\log^2 n) \quad \leftarrow \text{CLRS exercise}
\]

Can use to get \( \Theta(\log^2 n) \) Mergesort algorithm!