Parallel Algorithms

Background: Moore's Law

- Every X years, # transistors/chip doubles
- More transistors = more speed!

<table>
<thead>
<tr>
<th>Year</th>
<th>Processor</th>
<th>Clock Speed</th>
<th>Instruction CAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>Intel 8080</td>
<td>2 MHz</td>
<td>&lt;1 cycle/Inst.</td>
</tr>
<tr>
<td>1982</td>
<td>Intel 286</td>
<td>6 MHz</td>
<td>1 IPC</td>
</tr>
<tr>
<td>1985</td>
<td>Intel 386</td>
<td>17-40 MHz</td>
<td>2 IPC</td>
</tr>
<tr>
<td>1993</td>
<td>Pentium</td>
<td>50-146 MHz</td>
<td>9 IPC</td>
</tr>
<tr>
<td>2008</td>
<td>Pentium 1176</td>
<td>2.8 GHz</td>
<td>&gt;32x8</td>
</tr>
<tr>
<td>2017</td>
<td>Core i7</td>
<td>4.2 GHz</td>
<td></td>
</tr>
</tbody>
</table>
Moral: Clock speeds are not increasing

# instructions/cycle growing rapidly

If we want to process big data fast, do it in parallel!!

In CS5234: Sampling → lots of parallelism
Sketches → lots of parallelism
Cache-efficient? Many open questions

Types of Parallelism:
1) multicore
2) multisoCKET
3) clusters/data centers
4) distributed networks

Different settings ⇒ different costs
⇒ different solutions

Today: multicore/multisoCKET
Next week: Clusters
How to model/program?

1) PRAM: \( p \) processors (where \( p \to \infty \))

- shared memory
- program each proc separately

Ex. Given \( A[1, n] \), return true if all \( A[i] = 0 \)
false otherwise

\[
\text{AllZero}(A, 1, n, p) \leftarrow \text{running on proc } j \in [1, p] \\
\text{for } i = \left( \frac{n}{p} \right) (j - 1) + 1 \text{ to } \left( \frac{n}{p} \right) j \text{ do } \text{someone initialized } \text{answer=true?} \\
\quad \text{if } A[i] \neq 0 \text{ then } \text{answer} = \text{false} \\
\text{done} = \text{done} + 1 \leftarrow \text{race condition?} \\
\text{Wait until done} = p \leftarrow \text{use a lock?} \\
\text{return answer}
\]

- must carefully manage process interactions
- manually divide problem among pros, or coordinate
- low-level way to design parallel algorithms
Another example: sum an array (without locks)

Idea: a tree

```
3 5 2 -1 6 4 2 8
```

RandomSum: Repeat until root is not empty

1. Choose a random node in tree
2. If both children are not empty, then fill in node.

Claim

Finishes in \( \mathcal{O}\left(\frac{n \log n}{p} + \log n\right) \) (Non-trivial example of Coupon collector)
Or: statically assign procs to nodes in tree
   Each proc computes all assigned nodes,
   In order from deepest to highest
   \[ \mathcal{O} \left( \frac{n}{p} + \log n \right) \]
   \[ \uparrow \] why \( \log(n) \)?

→ messy to express algorithm

Alternate approach: (For any # procs)

\[
\text{SUM}(A, B, E) \quad \text{if } (B=E) \text{ return } A[B] \\
\text{mid} = \left\lfloor \frac{B+E}{2} \right\rfloor \\
\text{in parallel:} \\
\text{1} \quad L = \text{SUM}(A, B, \text{mid}) \\
\text{2} \quad R = \text{SUM}(A, \text{mid}+1, E) \\
\text{return } L+R
\]

Same tree calculation!
   → base case = leaves
   → each \( L+R \) calculates one node in tree
How fast?  How many processors?

Which processor executes what?

If all executed by 1 processor:
\[ T_1(n) = 2T_1\left(\frac{n}{2}\right) + O(1) \]
\[ = O(n) \]

one proc per node in tree

If \( p = \infty \) and each recursive cell is executed by a different processor:
\[ T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + O(1) \]
\[ = O(\log n) \]

both recursive calls in parallel

Take same time

On \( p \) processors?  Depends on scheduler

Work = cost on 1 proc = total compute steps

Span = cost on \( \infty \) proc = critical path

= longest sequential part of program
Fork-Join Model of parallelism

- program can spawn parallel threads
  - "fork"
  - "in parallel"
- program can wait until (all) parallel threads resolve ("join")
- procs assigned to threads by scheduler
- alg does not specify # procs

Implemented in most parallel programming platforms (in Java, Intel Parallel Studio, Cilk, etc.)

Clean way to think about algorithms (even if you implement it differently)

Schedulers

Claim: Greedy scheduling finishes a fork-join program with work $W$ and span $S$ on $p$ procs in time:

$$O\left(\frac{W}{p} + S\right)$$
Greedy: Schedule each proc to a thread, if any available. Otherwise, do nothing.

Why? \( \leq \frac{W}{p} \) steps when all procs busy
\( \leq S \) steps when not all procs busy
- Make progress on critical path
  just as good as \( p = \infty \)!

Also: Provably-efficient work-stealing schedulers are fast and also guarantee \( O\left(\frac{W}{p} + S\right) \) time.

Time for SUM: \( O\left(\frac{W}{p} + \log n\right) \)

How many procs do you need?

\[ T_p \geq S, \ T_p \geq \frac{W}{p} \]

If \( p = \frac{W}{S} \), then \( T_p = O\left(\frac{W}{W/S} + S\right) = O(S) \) ← optimal

Parallelism = \( \frac{W}{S} \)
Data Structure: Set

- `insert, delete: W = O(\log n), S = O(\log n)`

- `divide: W = O(\log n), S = O(\log n)`

  Divide set into two pieces, approximately the same size: \(|S_1| \geq \frac{1}{3} |S|, |S_2| \geq \frac{1}{3} |S|\)

- `union: W = O(n + m), S = O(\log n)`

  Combines 2 sets, removing duplicates

  Set sizes: \(n, m\), where \(n \geq m\)

- `set difference: W = O(n + m), S = O(\log (n))`

- `set minus: W = O(n + m), S = O(\log n)`

- `intersection: W = O(n + m), S = O(\log n)`

Union/intersection are expensive because must look at all elements → duplicates!
Implement a set as a balanced binary tree

- E.g., a red-black tree or a treap

Support:

1) Split \((T, K) \rightarrow T_1, T_2\)

\(\begin{align*}
T_1 &= \text{items in } T < K \\
T_2 &= \text{items in } T > K \\
X &= K \text{ if } K \in T, \bot \text{ otherwise}
\end{align*}\)

2) Join \((T_1, T_2) \rightarrow T\)

\(\begin{align*}
\text{Note: Union} \\
\text{if } \max(T_1) < \min(T_2) \leftrightarrow \text{does not require}
\end{align*}\)

\[\text{Hint: split does tree walk searching for key, dividing tree into many subtrees that are joined to form } T_1, T_2. \text{ join pastes trees and balances} \]

For both: \(W = O(\log n + \log m)\)

\(S = O(\log n + \log m)\)

Insert/delete: normal, sequential

Divide: find median: \(K\)

Split \((T, K)\) \(\text{or choose random median or use root to split if weight balanced}\)
\[ \text{Union}(T_1, T_2) \]

\begin{align*}
&\text{if } T_1 = \text{null} \text{ return } T_2 \\
&\text{if } T_2 = \text{null} \text{ return } T_1 \\
&\text{Key}_g = \text{Root } T_1 \\
&(L, R, X) = \text{split} (T_2, \text{Key}_g) \\
&R = \text{Root } T_1 \\
&\text{in parallel:} \\
&\quad 1) \ T_L = \text{Union}(R._{\text{left}}, L) \\
&\quad 2) \ T_R = \text{Union}(R._{\text{right}}, R) \\
&T = \text{join} (T_L, T_R) \quad \text{cost: } O(h_1 + h_2) \\
&\text{if } X \neq \perp \text{ then } \text{insert} (T, X) \quad \text{cost: } O(h_1 + h_2) \\
&\text{return } T \\
\end{align*}

Analysis: \( h_1 = \text{height } T_1, \ h_2 = \text{height } T_2 \)

\[
S(h_1, h_2) \leq S(h_1 - 1, h_2) + C(h_1 + h_2)
\]

\[
= O(h_1 (h_1 + h_2))
\]

\[
= O(\log^2 n + \log n \log m)
\]

\[
= O(\log^3 n) \text{ if } n > m
\]

More optimization: \( O(\log n) \)
\[ W(n, m) = W(n_1, m_1) + W(n_2, m_2) + O(\log n + \log m) \]
\[ \leq W(n_1, m) + W(n_2, m) + O(\log n + \log m) \]
\[ n_1 + n_2 = n \]
\[ = O(n \log m) \]

Optimized: \( O(m \log \frac{n}{m}) \)

Smaller tree

Recurrence tree = \( T_1 \)

Assigns cost \( \log m + \log(n_j) \)

to node \( n_j \)

\( \Rightarrow \) total cost: \( O(n \log m) \)
Intersection: if root of $T_1$ is in $T_2$: include root $T_1$

split $T_1$ by $T_1$’s root

Recursively find difference $T_1.L \oplus L$, $T_1.R \oplus R$

Join 2 results

if include root $T_1$: insert root $T_1$

Difference: if root of $T_1$ not in $T_2$: include root $T_1$

split $T_1$ by $T_1$’s root

Recursively find difference $T_1.L \oplus L$, $T_1.R \oplus R$

Join 2 results

if include root $T_1$: insert root $T_1$

SetMinus:
**BFS**

\[ F \leftarrow \{ s \} \]

repeat until \( F = \emptyset \):

\[ F' = \emptyset \]

for each \( u \in F \):

\[ \text{visited}[u] = \text{true} \]

for each nbr \( v \) of \( u \):

if \( \text{visited}[v] = \text{false} \)

then \( F'.\text{add}(v) \)

\[ F = F' \]

---

**Parallel BFS**

\[ \rightarrow \text{Process frontier in parallel} \]

\[ \rightarrow \text{assume } u.\text{nbrs is a set} \]

[Ex: build sets in parallel from adjacency list]

\[ \rightarrow \text{use set operations to manipulate frontier } F, \text{ visited} \]

---

**Par BFS (G, s)**

\[ F = \{ s \} \]

\[ D = \emptyset \]

repeat until \( F = \emptyset \):

\[ D = \text{Union}(D, F) \leftarrow \text{Mark frontier done} \]

\[ F = \text{Process Frontier}(F) \]

\[ F = \text{SetMinus}(F, D) \]
ProcessFrontier \((F)\)

if \(|F| = 1\) then: \(U = \text{node in } F\)

\(\text{return } U.\text{nbrs} \leftarrow \text{set}\)

\(\langle F_1, F_2 \rangle = \text{Divide}(F)\)

in parallel:

1) \(F_1 = \text{ProcessFrontier}(F_1)\)

2) \(F_2 = \text{ProcessFrontier}(F_2)\)

\(\text{return } \text{Union}(F_1, F_2)\)

Analysis:

For \(\text{ProcessFrontier}\) on \(n\) nodes with \(m\) nbrs:

\[
W(n,m) \leq 2W(\frac{n}{2},m) + O(m \log m) + O(\log n)
\]

\[
= O(m \log n \cdot \log m)
\]

\[
S(n,m) = S(\frac{n}{2}, m) + O(\log n) + O(\log^2 n)
\]

\[
= O(\log^2 m \log n)
\]
For BFS: \( \text{Work} \leq \sum_{F_i} W(F_i, m) + O(m \log m) \)

\( \leq O(m \log^3 n) \quad (m \geq n), \quad (m < n^2) \)

\( \text{Span} \leq \sum_{F_i} S(F_i, m) + O(\log^3 m) \)

\( \leq O(D \log^2 m \log n) \)

\( = O(D \log^3 n) \)

\( D = \text{diameter} \)

Conclusion: Par BFS on \( p \) proc runs in time:

\( O\left(\frac{m \log^2 n + D \log^3 n}{p}\right) \)

Note: \( \mathcal{O}(n) \) is almost inevitable

Only good if \( p > \log^2 n \) [improvements exist]

DFS is, as best we know, \( \mathcal{O}(n) \)

\( \rightarrow \) use BFS, not DFS