CSE 613: Parallel Programming

Lecture 6
( Basic Parallel Algorithmic Techniques )

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2019
Some Basic Techniques

1. Divide-and-Conquer
   - Recursive
   - Non-recursive
   - Contraction

2. Pointer Techniques
   - Pointer Jumping
   - Graph Contraction

3. Randomization
   - Sampling
   - Symmetry Breaking
Divide-and-Conquer

1. **Divide:** divide the original problem into smaller subproblems that are easier are to solve

2. **Conquer:** solve the smaller subproblems (perhaps recursively)

3. **Merge:** combine the solutions to the smaller subproblems to obtain a solution for the original problem
Divide-and-Conquer

- The divide-and-conquer paradigm improves program modularity, and often leads to simple and efficient algorithms.
- Since the subproblems created in the divide step are often independent, they can be solved in parallel.
- If the subproblems are solved recursively, each recursive divide step generates even more independent subproblems to be solved in parallel.
- In order to obtain a highly parallel algorithm it is often necessary to parallelize the divide and merge steps, too.
Recursive D&C: Parallel Merge Sort

Merge-Sort (A, p, r) { sort the elements in A[p ... r] }

1. if p < r then
2. q ← ⌊(p + r) / 2⌋
3. Merge-Sort (A, p, q)
4. Merge-Sort (A, q + 1, r)
5. Merge (A, p, q, r)

Par-Merge-Sort (A, p, r) { sort the elements in A[p ... r] }

1. if p < r then
2. q ← ⌊(p + r) / 2⌋
3. spawn Merge-Sort (A, p, q)
4. Merge-Sort (A, q + 1, r)
5. sync
6. Merge (A, p, q, r)
**Recursive D&C: Parallel Merge Sort**

Par-Merge-Sort (A, p, r) { sort the elements in A[p ... r] }

1. if p < r then
2. q ← ⌊(p + r) / 2⌋
3. spawn Merge-Sort (A, p, q)
4. Merge-Sort (A, q + 1, r)
5. sync
6. Merge (A, p, q, r)

**Work:**

\[ T_1(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1, \\
2T_1\left(\frac{n}{2}\right) + \Theta(n), & \text{otherwise.} 
\end{cases} \]

\[ = \Theta(n \log n) \]

**Span:**

\[ T_\infty(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1, \\
T_\infty\left(\frac{n}{2}\right) + \Theta(n), & \text{otherwise.} 
\end{cases} \]

\[ = \Theta(n) \]

**Parallelism:**

\[ \frac{T_1(n)}{T_\infty(n)} = \Theta(\log n) \]

Too small! Must parallelize the Merge routine.
Non-Recursive D&C: Parallel Sample Sort

Task: Sort an array $A[1, \ldots, n]$ of $n$ distinct keys using $p \leq n$ processors.

Steps (without oversampling):

1. **Pivot Selection:** Select (uniformly at random) and sort $m = p - 1$ pivot elements $e_1, e_2, \ldots, e_m$. These elements define $m + 1 = p$ buckets: $(-\infty, e_1), (e_1, e_2), \ldots, (e_{m-1}, e_m), (e_m, +\infty)$.

2. **Local Sort:** Divide $A$ into $p$ segments of equal size, assign each segment to different processor, and sort locally.

3. **Local Bucketing:** If $m \leq \frac{n}{p}$, each processor inserts the pivot elements into its local sorted sequence using binary search, otherwise inserts its local elements into the sorted pivot elements. Thus the keys are divided among $m + 1 = p$ buckets.

4. **Merge Local Buckets:** Processor $i$ ($1 \leq i \leq p$) merges the contents of bucket $i$ from all processors through a local sort.

5. **Final Result:** Each processor copies its bucket to a global output array so that bucket $i$ ($1 \leq i \leq p - 1$) precedes bucket $i + 1$ in the output.
Non-Recursive D&C: Parallel Sample Sort

Steps (without oversampling):

1. **Pivot Selection**: \( O(m \log(m)) = O(p \log p) \) [worst case]

2. **Local Sort**: \( O\left(\frac{n}{p} \log \frac{n}{p}\right) \) [worst case]

3. **Local Bucketing**:

   \[ O\left(\min(m \log \frac{n}{p}, \frac{n}{p} \log m)\right) = O\left(\frac{n}{p} \log \frac{n}{p}\right) \] [worst case]

4. **Merge Local Buckets**: \( O\left(\frac{n}{m} \log \frac{n}{m}\right) = O\left(\frac{n}{p} \log \frac{n}{p}\right) \) [expected]

   (not quite correct as the largest bucket can have \( \Theta\left(\frac{n}{m} \log m\right) \) keys with significant probability)

5. **Final Result**: \( O\left(\frac{n}{m}\right) = O\left(\frac{n}{p}\right) \) [expected]

**Overall**: \( O\left(\frac{n}{p} \log \frac{n}{p} + p \log p\right) \) [expected]
Contraction

1. **Reduce**: reduce the original problem to a smaller problem
2. **Conquer**: solve the smaller problem (often recursively)
3. **Expand**: use the solution to the smaller problem to obtain a solution for the original larger problem
Contraction: Prefix Sums

**Input:** A sequence of \( n \) elements \( \{x_1, x_2, \ldots, x_n\} \) drawn from a set \( S \) with a binary associative operation, denoted by \( \oplus \).

**Output:** A sequence of \( n \) partial sums \( \{s_1, s_2, \ldots, s_n\} \), where

\[
s_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i \text{ for } 1 \leq i \leq n.
\]

\[
\begin{array}{cccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\
  5   & 3   & 7   & 1   & 3   & 6   & 2   & 4   \\
\end{array}
\]

\( \oplus = \text{binary addition} \)

\[
\begin{array}{cccccccc}
  s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \\
  5   & 8   & 15  & 16  & 19  & 25  & 27  & 31  \\
\end{array}
\]
Contraction: Prefix Sums

Prefix-Sum \( (x_1, x_2, ..., x_n, \oplus) \) \{ \( n = 2^k \) for some \( k \geq 0 \).

Return prefix sums \( \langle s_1, s_2, ..., s_n \rangle \) \}

1. \( \text{if } n = 1 \text{ then} \)
2. \( s_1 \leftarrow x_1 \)
3. \( \text{else} \)
4. \( \text{parallel for } i \leftarrow 1 \text{ to } n/2 \text{ do} \)
5. \( y_i \leftarrow x_{2i-1} \oplus x_{2i} \)
6. \( \langle z_1, z_2, ..., z_{n/2} \rangle \leftarrow \text{Prefix-Sum}( \langle y_1, y_2, ..., y_{n/2} \rangle, \oplus) \)
7. \( \text{parallel for } i \leftarrow 1 \text{ to } n \text{ do} \)
8. \( \text{if } i = 1 \text{ then } s_1 \leftarrow x_1 \)
9. \( \text{else if } i = \text{even then } s_i \leftarrow z_{i/2} \)
10. \( \text{else } s_i \leftarrow z_{(i-1)/2} \oplus x_i \)
11. \( \text{return } \langle s_1, s_2, ..., s_n \rangle \)
Contraction: Prefix Sums
Contraction: Prefix Sums
Contraction: Prefix Sums

Prefix-Sum \( \langle x_1, x_2, \ldots, x_n \rangle, \oplus \) \{ \( n = 2^k \) for some \( k \geq 0 \).
Return prefix sums \( \langle s_1, s_2, \ldots, s_n \rangle \} \)

1. if \( n = 1 \) then
2. \( s_1 \leftarrow x_1 \)
3. else
4. parallel for \( i \leftarrow 1 \) to \( n/2 \) do
5. \( y_i \leftarrow x_{2i-1} \oplus x_{2i} \)
6. \( \langle z_1, z_2, \ldots, z_{n/2} \rangle \leftarrow \text{Prefix-Sum}( \langle y_1, y_2, \ldots, y_{n/2} \rangle, \oplus \) \)
7. parallel for \( i \leftarrow 1 \) to \( n \) do
8. if \( i = 1 \) then \( s_1 \leftarrow x_1 \)
9. else if \( i = \text{even} \) then \( s_i \leftarrow z_{i/2} \)
10. else \( s_i \leftarrow z_{(i-1)/2} \oplus x_i \)
11. return \( \langle s_1, s_2, \ldots, s_n \rangle \)

Observe that we have assumed here that a \textit{parallel for loop} can be executed in \( \Theta(1) \) time. But recall that \textit{cilk\_for} is implemented using divide-and-conquer, and so in practice, it will take \( \Theta(\log n) \) time. In that case, we will have \( T_\infty(n) = \Theta(\log^2 n) \), and parallelism \( = \Theta \left( \frac{n}{\log^2 n} \right) \).
Pointer Techniques: Pointer Jumping

The *pointer jumping* (or *path doubling*) technique allows fast processing of data stored in the form of a set of rooted directed trees.

For every node $v$ in the set pointer jumping involves replacing $v \rightarrow \text{next}$ with $v \rightarrow \text{next} \rightarrow \text{next}$ at every step.

Some Applications

- Finding the roots of a forest of directed trees
- Parallel prefix on rooted directed trees
- List ranking
**Find-Roots** \( (n, P, S) \) \{ \text{Input: A forest of rooted directed trees, each with a self-loop at its root, such that each edge is specified by } (v, P(v)) \text{ for } 1 \leq v \leq n. \text{ Output: For each } v, \text{ the root } S(v) \text{ of the tree containing } v. \} \}

1. parallel for \( v \leftarrow 1 \) \text{ to } n \text{ do}
2. \( S(v) \leftarrow P(v) \)
3. \( \text{flag} \leftarrow \text{true} \)
4. while \( \text{flag} = \text{true} \) do
5. \( \text{flag} \leftarrow \text{false} \)
6. parallel for \( v \leftarrow 1 \) \text{ to } n \text{ do}
7. \( S(v) \leftarrow S(S(v)) \)
8. \text{if } S(v) \neq S(S(v)) \text{ then } \text{flag} \leftarrow \text{true} \)
Pointer Jumping: Roots of a Forest of Directed Trees

Let $h$ be the maximum height of any tree in the forest. Observe that the distance between $v$ and $S(v)$ doubles after each iteration until $S(S(v))$ is the root of the tree containing $v$.

Hence, the number of iterations is $\log h$. Thus (assuming that each parallel for loop takes $\Theta(1)$ time to execute),

**Work:** $T_1(n) = O(n \log h)$ and **Span:** $T_\infty(n) = \Theta(\log h)$

**Parallelism:** $\frac{T_1(n)}{T_\infty(n)} = O(n)$

---

**Find-Roots** ($n, P, S$)  
{ **Input:** A forest of rooted directed trees, each with a self-loop at its root, such that each edge is specified by $(v, P(v))$ for $1 \leq v \leq n$.  
**Output:** For each $v$, the root $S(v)$ of the tree containing $v$. }

1. parallel for $v \leftarrow 1$ to $n$ do
2. $S(v) \leftarrow P(v)$
3. flag $\leftarrow$ true
4. while flag = true do
5. flag $\leftarrow$ false
6. parallel for $v \leftarrow 1$ to $n$ do
7. $S(v) \leftarrow S(S(v))$
8. if $S(v) \neq S(S(v))$ then flag $\leftarrow$ true
Pointer Techniques: Graph Contraction

1. **Contract**: the graph is reduced in size while maintaining some of its original properties (depending on the problem)

2. **Conquer**: solve the problem on the contracted graph (often recursively)

3. **Expand**: use the solution to the contracted graph to obtain a solution for the original graph

Some Applications

- Finding connected components of a graph
- Minimum spanning trees
Graph Contraction: Connected Components (CC)

1. Direct the edges to form a forest of rooted directed trees
2. Use pointer jumping to contract each such tree to a single vertex
3. Recursively find the CCs of the contracted graph
4. Expand those CCs to label the vertices of the original graph with CC numbers
Randomization: Symmetry Breaking

A technique to break symmetry in a structure, e.g., a graph which can locally look the same to all vertices.

Some Applications

- Prefix sums in a linked list (list ranking)
- Selecting a large independent set from a graph
- Graph contraction
Symmetry Breaking: List Ranking

1. Flip a coin for each list node
2. If a node $u$ points to a node $v$, and $u$ got a head while $v$ got a tail, combine $u$ and $v$
3. Recursively solve the problem on the contracted list
4. Project this solution back to the original list
Symmetry Breaking: List Ranking

In every iteration a node gets removed with probability $\frac{1}{4}$ (as a node gets head with probability $\frac{1}{2}$ and the next node gets tail with probability $\frac{1}{2}$).

Hence, a quarter of the nodes get removed in each iteration (expected number).

Thus the expected number of iterations is $\Theta(\log n)$.

In fact, it can be shown that with high probability,

$$T_1(n) = O(n) \text{ and } T_\infty(n) = O(\log n)$$