More PRAM Algorithms

Arvind Krishnamurthy
Fall 2004

Techniques Covered

- Analysis technique:
  - Brent's scheduling lemma
  - Parallel algorithm is simply characterized by \( W(n) \) and \( S(n) \)
- Parallel techniques:
  - Scans
  - Pointer doubling
  - Euler tours
  - List ranking and list suffix operations
  - Parallel divide and conquer techniques
- Today:
  - Connected components
  - Sorting algorithms

Sequential Algorithm

- Pretty straightforward:
  - Just perform some kind of traversal of the graph
  - Depth-first search (DFS), breadth-first search (BFS), etc.
  - Label the components
- Performance of sequential algorithm:
  - \( O(n + e) \)
  - Cache locality? Sometimes BFS turns out to be a better option than DFS

PRAM Algorithm: high level description

- Proposed by Shiloach and Vishkin
- Start with a forest of singleton vertices
- At each iteration, perform:
  - Hooking: attach a star (or a singleton vertex) with another tree
    - Comes in two forms: conditional and unconditional hooking
  - Pointer doubling: collapse the trees using pointer doubling
- Algorithm terminates when the trees in the forest do not have edges between them
- Parallelism details:
  - There is one processor for each vertex and each edge
  - The edge processors are active for “hooking” and the vertex processors are active for pointer doubling

Connected Components

- Compute the connected components of a graph
- Has many applications: vision, physics simulations, etc.

Example of conditional hooking

- Conditional hooking:
  - Attaches a star to a tree
  - Only if target tree-vertex has a lower number
Example of conditional hooking

- Decreases the height of trees in the forest
- Collapses the tree by taking each vertex and making its current grand-parent the new parent
- Propagate grand-parent’s identity to current node

Unconditional hooking

- Just having the conditional hooking and pointer doubling isn’t sufficient to have an asymptotically fast (O(log n)) algorithm
- Throw in unconditional hooking
  - Perform unconditional hooking only on “stagnant” stars
  - Stagnant stars: those stars which had an opportunity to hook up using conditional hooking but failed to do so
  - Eliminate the condition to hook the star during unconditional hooking
- Refined algorithm is loop over:
  - Perform conditional hooking for all stars (using edge processors)
  - For stagnant stars, perform unconditional hooking (with edge-processors)
  - Perform pointer doubling (using vertex-processors)

Sorting

- Traditional CS problem
- Sort a sequence of numbers stored in shared memory
- Can we solve it based on the techniques that we have seen so far
  - With n^2 processors and log n time?

Odd-Even Merge - classic parallel sort

<table>
<thead>
<tr>
<th>N values to be sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Sort each separately</td>
</tr>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Redistribute into even and odd sublists</td>
</tr>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Merge into two sorted lists</td>
</tr>
<tr>
<td>E_0, E_1, E_2, E_3, O_0, O_1, O_2, O_3</td>
</tr>
<tr>
<td>Pairwise swaps of E_i and O_i will put it in order</td>
</tr>
</tbody>
</table>

Odd-Even Merge - classic parallel sort

<table>
<thead>
<tr>
<th>N values to be sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Sort each separately</td>
</tr>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Redistribute into even and odd sublists</td>
</tr>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Merge into two sorted lists</td>
</tr>
<tr>
<td>E_0, E_1, E_2, E_3, O_0, O_1, O_2, O_3</td>
</tr>
<tr>
<td>Pairwise swaps of E_i and O_i will put it in order</td>
</tr>
</tbody>
</table>

Odd-Even Merge - classic parallel sort

<table>
<thead>
<tr>
<th>N values to be sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Sort each separately</td>
</tr>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Redistribute into even and odd sublists</td>
</tr>
<tr>
<td>A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3</td>
</tr>
<tr>
<td>Merge into two sorted lists</td>
</tr>
<tr>
<td>E_0, E_1, E_2, E_3, O_0, O_1, O_2, O_3</td>
</tr>
<tr>
<td>Pairwise swaps of E_i and O_i will put it in order</td>
</tr>
</tbody>
</table>

Example

Sort (recursive)

2 4 6 8
1 5 6 7

Merge (recursive)

2 4 6 8
1 3 6 8

Proof of correctness by 0-1 sorting lemma

2 1 3 4 5 6 7 9
1 2 3 4 5 6 7 8
Functional Specification

- **Sort(S):**
  Let $S = A \| B$
  $C = \text{Sort}(A); \ D = \text{Sort}(B);$
  return $\text{Merge}(C, D);$

- **Merge(A, B):**
  $C = \text{Merge}(\text{even}(A), \text{odd}(B)); \ D = \text{Merge}(\text{odd}(A), \text{even}(B));$
  $E = \text{interleave}(C, D);$
  return $\text{pairwise}_{-}\text{comp}(E);$

Proof that Merge works

- Requires the 0-1 sorting lemma:
  - If an algorithm that uses just comparisons works with any sequence of 0's and 1's, then it works with any sequence of numbers.
  - $\text{Merge}(A, B)$ produces the correct output:
    - Let $A$ have $x$ 0's and $n/2-x$ 1's. Let $B$ have $y$ 0's and $n/2-y$ 1's.
    - $C$ then has $\lfloor x/2 \rfloor + \lfloor y/2 \rfloor$ 0's and $D$ has $\lceil y/2 \rceil + \lceil x/2 \rceil$ 0's.
    - It follows that the number of 0's in $C$ and $D$ can differ by at most one.
    - So $\text{pairwise}_{-}\text{comp}$ after interleaving should sort the two sequence

Complexity Measures

- Analyze merge operation separately:
  - What is work complexity?
  - What is step complexity?

- Sorting is simply a sequence of merge operations:
  - What is work complexity?
  - What is step complexity?

Bitonic Sort

- A bitonic sequence is one that is:
  - Monotonically increasing and then monotonically decreasing
  - Or monotonically decreasing and then increasing

- Examples:
  - $1\ 4\ 7\ 9\ 11\ 8\ 6\ 4$
  - $11\ 9\ 8\ 7\ 4\ 6\ 12\ 13$

- Bitonic sequences are "almost" sorted

Where's the Parallelism?

- A half-cleaner takes a bitonic sequence and produces
  - First half is smaller than smallest element in 2nd half
  - Both halves are bitonic

Half cleaner

- $0\ 0\ 0\ 0\ 0\ 1\ 1\ 1$
- $0\ 0\ 1\ 1\ 1\ 0\ 0\ 0$
- $0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$
- $0\ 0\ 0\ 0\ 0\ 0\ 1\ 0$
- $0\ 0\ 0\ 0\ 0\ 1\ 0\ 0$
- $0\ 0\ 0\ 0\ 1\ 0\ 0\ 0$
- $0\ 0\ 0\ 1\ 0\ 0\ 0\ 0$
- $0\ 0\ 1\ 0\ 0\ 0\ 0\ 0$
Proof
- Consider all possible bitonic sequences of 0's and 1's
- What happens after one level of comparisons:
  - Case 1:
  - Case 2:
  - Case 3:
  - Case 4:

Complexity Issues
- What is the complexity of the half-cleaner:
  - Number of operations = n
  - Number of steps = 1
- What is the complexity of the cleaner:
  - Number of operations = n \log n
  - Number of steps = \log n
- What is the complexity of the sorting algorithm:
  - Number of operations = n \log^2 n
  - Number of steps = \log n

Uses of a half-cleaner
- Question: how can we use the half-cleaner to sort a bitonic sequence?
  - In other words, accomplish the following: input is a bitonic sequence, output is a sorted sequence

Announcements
- Homework on PRAM algorithms posted on class website
  - Due next Wednesday. Individual work.
- Start doing preparatory work for class project:
  - Topics and pointers to links will be posted on the class website
  - Groups of two students each
  - Start by becoming an "expert" in some topic and then turn it into a semester-long project
- Upcoming lectures:
  - Shared memory architectures and programming models
  - Distributed memory topologies and programming models
  - Distributed algorithms

Bitonic Sort
- Problem 1: cleaning a bitonic sequence (solved)
- Problem 2: create a bitonic sequence from two sorted sequences:
  - Reverse the second sequence
  - Concatenate with first
- Sort a sequence: pulling together the pieces
  \[ \text{Sort (S):} \]
  - Let \( S = A \| B \)
  - \( C = \text{Sort}(A) \)
  - \( D = \text{Sort}(B) \)
  - \( E = C \| \text{reverse}(D) \)
  - return Clean(E)

Minimum Spanning Trees
- Computed the minimum weight spanning tree of a graph
- All the vertices of the graph must be included
**Sequential Algorithm**

- Start with singleton vertices
- Repeat:
  - Select an arbitrary set
  - Choose an edge with the minimum weight outgoing from this set
  - Combine the two sets
  - Stop when there is just one set left

- Avoid creating cycles
- Kruskal’s algorithm: combine along the minimum-weight edge in the graph
- Prim-Dijkstra: start with just one distinguished vertex and grow the spanning tree

**Parallel Algorithm**

- Which of the various techniques that we have studied could be used for designing an efficient parallel algorithm?

- Which sequential algorithm would serve as a good starting point?