CSc 8530
Parallel Algorithms

Spring 2019

February 7th, 2019

Optimality notions

- A sequential algorithm is **time optimal** iff its running time $T^*(n)$ cannot be improved asymptotically
- Two notions of optimality for parallel algorithms:
  - **Weak:** a WT presentation level algorithm is optimal iff $W(n) = \Theta(T^*(n))$
  - The total number of operations (not the running time) of the parallel algorithm is asymptotically equivalent to the sequential one
  - **Strong:** The running time $T(n)$ cannot be improved by any other parallel algorithm

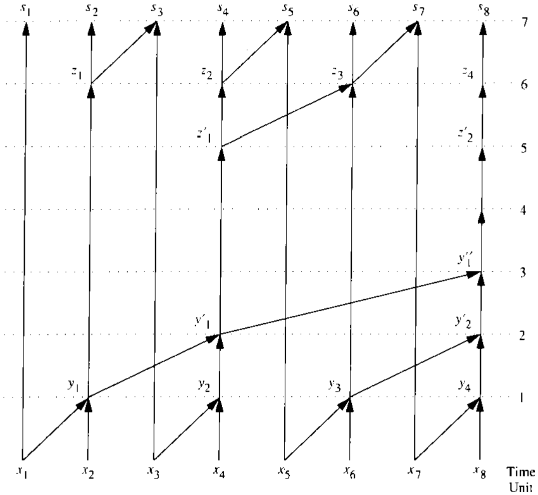
Example: prefix sums

- Let $S = \{x_1, x_2, \dots, x_n\}$ be an n-element set
- Let $*$ be a binary associate operation (e.g., sum or product)
- A **prefix sum** is the partial sum defined by:

$$s_i = x_1 * x_2 * \dots * x_i, 1 \leq i \leq n$$

- The **prefix sums** are the n partial products s_1 to s_n
- A trivial sequential algorithm can compute s_i from s_{i-1} as $s_i = s_{i-1} * x_i$
  - Clearly, this algorithm is $O(n)$

Recursive prefix-sums algorithm



Recursive prefix-sums algorithm – analysis

- **Resources required:**
- Step 1 takes $O(1)$ (sequential) time
- Steps 2 and 4 take $O(1)$ (parallel) time
  - With $O(n)$ operations per step
- Thus, the running time and work satisfy the following recurrences:

$$T(n) = T\left(\frac{n}{2}\right) + a$$
$$W(n) = W\left(\frac{n}{2}\right) + bn$$

where a and b are constants

- Their respective solutions are:

$T(n) = O(\log n)$ We reduce T(n) by half in each step

$W(n) = O(n)$ The sum at each level decreases geometrically

Non-recursive prefix-sums algorithm

ALGORITHM 2.2

(Nonrecursive Prefix Sums)

Input: *An array A of size $n = 2^k$, where k is a nonnegative integer.*
Output: *An array C such that $C(0, j)$ is the jth prefix sum, for $1 \le j \le n$.*

begin

1. **for** $1 \le j \le n$ **pardo**
  Set $B(0, j) := A(j)$

2. **for** $h = 1$ **to** $\log n$ **do**
  for $1 \le j \le n/2^h$ **pardo**
    Set $B(h, j) := B(h-1, 2j-1) * B(h-1, 2j)$

3. **for** $h = \log n$ **to** 0 **do**
  for $1 \le j \le n/2^h$ **pardo**

$$\begin{cases} j \text{ even} & : \text{Set } C(h,j) := C\left(h+1, \frac{j}{2}\right) \\ j = 1 & : \text{Set } C(h,1) := B(h,1) \\ j \text{ odd} > 1 & : \text{Set } C(h,j) := C\left(h+1, \frac{j-1}{2}\right) * B(h,j) \end{cases}$$

end

Pointer jumping

- We will now explore another basic parallel design technique
  - Will be particularly applicable to some types of graph data
- A **rooted-directed tree** T is a directed graph such that:
  1. There is a root node with out-degree 0
  2. Every other node has out-degree 1
  3. There is directed path from every node to the root
- **Pointer jumping** allows the fast processing of data stored as rooted-directed trees

Example: finding the roots of a forest

- A forest F is a set of rooted-directed trees
  - Equivalently, it is a disconnected graph where every connected component is a rooted-directed tree
- We specify F using an array P
  - $P(i) = j$ if (i, j) is an arc in F
  - That is, j is the parent of i
  - If $P(i) = i$, then i is a root
- Our goal is to determine the root $S(j)$, for each j
- A sequential algorithm can easily solve this problem
  - How?
  - How quickly?

Example: finding the roots of a forest

- A forest F is a set of rooted-directed trees
  - Equivalently, it is a disconnected graph where every connected component is a rooted-directed tree
- We specify F using an array P
  - $P(i) = j$ if (i, j) is an arc in F
  - That is, j is the parent of i
  - If $P(i) = i$, then i is a root
- Our goal is to determine the root $S(j)$, for each j
- A sequential algorithm can easily solve this problem
  - How?
    - First identify the roots and then perform BFS or DFS from each one
  - How quickly?

Example: finding the roots of a forest

- A forest F is a set of rooted-directed trees
  - Equivalently, it is a disconnected graph where every connected component is a rooted-directed tree
- We specify F using an array P
  - $P(i) = j$ if (i, j) is an arc in F
  - That is, j is the parent of i
  - If $P(i) = i$, then i is a root
- Our goal is to determine the root $S(j)$, for each j
- A sequential algorithm can easily solve this problem
  - How?
    - First identify the roots and then perform BFS or DFS from each one
  - How quickly?
    - In $O(|V| + |E|)$
    - Since trees are sparse graphs, the above simplifies to $O(|V|)$

Root finding: parallel approach

- Pointer jumping consists of *updating the successor of each node by that successor's successor*
- By iterating, we gradually move closer to the root
- The distance between a node and its successor doubles after each iteration
  - Except when the successor is a root
- After k iterations,the distance between i and $S(i)$ is 2^k
- Equivalently, the distance to the root is cut in half in each iteration

Root finding: parallel algorithm

ALGORITHM 2.4

(Pointer Jumping)

Input: *A forest of rooted directed trees, each with a self-loop at its root, such that each arc is specified by* $(i, P(i))$, *where* $1 \le i \le n$.
Output: *For each vertex* i, *the root* $S(i)$ *of the tree containing* i.

begin

1. **for** $1 \le i \le n$ **pardo**

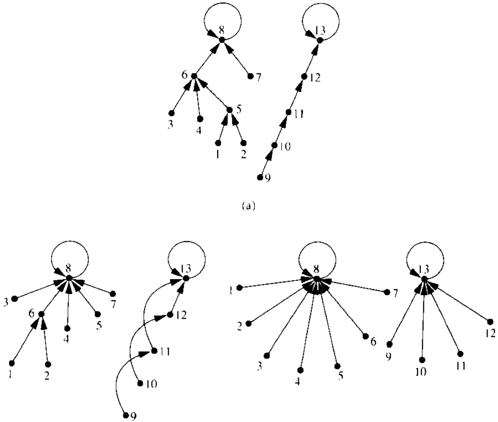
  Set $S(i) := P(i)$

  while $(S(i)) \ne S(S(i))$ **do**

    Set $S(i) := S(S(i))$

end

Root finding: examples



- Notice how the distance to the root is cut in half in each iteration

Root finding: analysis

- Let h be the maximum height of any tree in F
  - In the worst case, $h = O(n)$ (why?)
- Clearly, at the end $S(i)$ is a root, for all i
  - Intuitively, the roots are attractors or steady states of the iterative process
- The convergence rate to a root is 2^h
  - Because we half the distance to the root at each iteration
  - Hence, the number of iterations is $O(\log(h))$
- Each iteration takes $O(1)$ parallel time for $O(n)$ operations (why?)
- **Running time:**
- **Work:**

Root finding: analysis

- Let h be the maximum height of any tree in F
  - In the worst case, $h = O(n)$ (why?)
- Clearly, at the end $S(i)$ is a root, for all i
  - Intuitively, the roots are attractors or steady states of the iterative process
- The convergence rate to a root is 2^h
  - Because we half the distance to the root at each iteration
  - Hence, the number of iterations is $O(\log(h))$
- Each iteration takes $O(1)$ parallel time for $O(n)$ operations (why?)
- **Running time:**
  - $O(\log(h))$
- **Work:**

Root finding: analysis

- Let h be the maximum height of any tree in F
  - In the worst case, $h = O(n)$ (why?)
- Clearly, at the end $S(i)$ is a root, for all i
  - Intuitively, the roots are attractors or steady states of the iterative process
- The convergence rate to a root is 2^h
  - Because we half the distance to the root at each iteration
  - Hence, the number of iterations is $O(\log(h))$
- Each iteration takes $O(1)$ parallel time for $O(n)$ operations (why?)
- **Running time:**
  - $O(\log(h))$
- **Work:**
  - $O(n \log(h))$

Root finding

- We can easily show that this algorithm is not optimal (how?, in what sense?)

Root finding

- We can easily show that this algorithm is not optimal (how?, in what sense?)
  - A $T_s(n) = O(n)$ sequential algorithm exists
  - $W(n) > T_s(n)$, so the algorithm cannot be **weakly** optimal
  - Intuitively, we are creating more work for ourselves by operating in parallel
- Could this algorithm be **strongly** optimal?

Root finding

- We can easily show that this algorithm is not optimal (how?, in what sense?)
  - A $T_s(n) = O(n)$ sequential algorithm exists
  - $W(n) > T_s(n)$, so the algorithm cannot be **weakly** optimal
  - Intuitively, we are creating more work for ourselves by operating in parallel
- Could this algorithm be **strongly** optimal?
  - Theoretically yes, if $T(n)$ cannot be improved by *any* other parallel algorithm