CSc 8530
Parallel Algorithms

Spring 2019

February 14th, 2019

Parallel prefix using pointer jumping

ALGORITHM 2.5

(Parallel Prefix on Rooted Directed Trees)

Input: *A forest of rooted directed trees, each with a self-loop at its root such that (1) each arc is specified by* $(i, P(i))$, *(2) each vertex* i *has a weight* $W(i)$, *and (3) for each root* r, $W(r) = 0$.

Output: *For each vertex* i, $W(i)$ *is set equal to the sum of the weights of vertices on the path from* i *to the root of its tree.*

begin

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

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

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

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

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

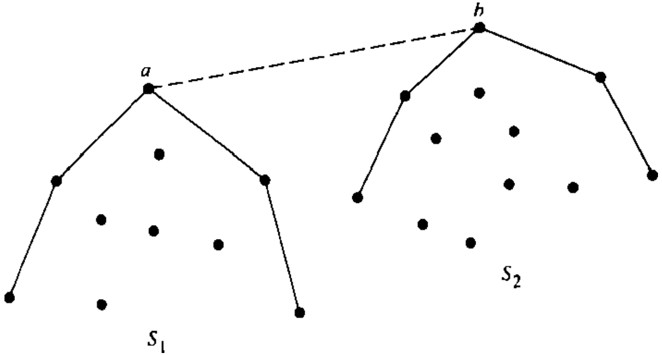
end

- The running time and work are asymptotically the same as the previous algorithm
  - Because we only added an $O(1)$ operation

Parallel convex-hull algorithm

- We first sort all the points by their x coordinates
  - As noted, this takes $\Theta(n \log(n))$
- We can parallel sort n numbers in $T(n) = O(\log(n))$ and $W(n) = O(n \log(n))$
  - We will prove this result in a few classes
- Let $x(p_1) < x(p_2) < \ldots < x(p_n)$
- Let $S_1 = (p_1, p_2, \ldots, p_{n/2})$ and $S_2 = (p_{n/2+1}, \ldots, p_n)$
- Suppose that we already have $UH(S_1)$ and $UH(S_2)$
- Then, the **upper common tangent** is the closest line that lies above both $UH(S_1)$ and $UH(S_2)$

Upper common tangent example



- Both a and b have to be part of the convex hull of S

Parallel convex hull pseudocode

ALGORITHM 2.6
(Simple Upper Hull)

Input: *A set S of n points in the plane, no two of which have the same x or y coordinates such that $x(p_1) < x(p_2) < \cdots < x(p_n)$, where n is a power of 2.*
Output: *The upper hull of S.*

begin

1. *If $n \leq 4$, then use a brute-force method to determine $UH(S)$, and exit.*
2. *Let $S_1 = (p_1, p_2, \cdots, p_{\frac{n}{2}})$ and $S_2 = (p_{\frac{n}{2}+1}, \ldots, p_n)$. Recursively, compute $UH(S_1)$ and $UH(S_2)$ in parallel.*
3. *Find the upper common tangent between $UH(S_1)$ and $UH(S_2)$, and deduce the upper hull of S.*

end

- The pseudocode for $LH(S)$ is identical
- The final convex hull is then given by $CH(S) = UH(S) \cup LH(S)$

Parallel convex hull – analysis

- Step 1 takes $O(1)$ sequential time
- Step 2 takes $T(n/2)$ using $2W(n/2)$ operations
- Step 3:
  - Upper common tangent: $O(\log(n))$
  - Combining upper hulls: $O(1)$ parallel time and $O(n)$ work
- Thus:

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

  a and b are positive constants
- Total running time and work:
  - $T(n) = O(\log^2(n))$
  - $W(n) = O(n\log(n))$

Parallel convex hull – other analyses

- The previous algorithm requires the CREW PRAM model (why?)
  - Merging $UH(S_1)$ and $UH(S_2)$ may require accessing the same point simultaneously
- For p processors, the algorithm runs in $O\left(\frac{n\log(n)}{p} + \log^2(n)\right)$
  - What values of p achieve an optimal speedup?
  - We want $\frac{n\log(n)}{p} \leq \log^2(n)$ to eliminate the left-hand term
  - Thus:

$$\frac{n\log(n)}{p} \leq \log^2(n)$$
$$n\log(n) \leq p\log^2(n)$$
$$\frac{n}{\log(n)} \leq p$$

  - Any value of p in this range will be asymptotically dominated by the $\log^2(n)$ term

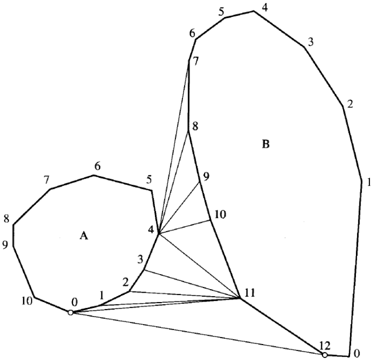
Upper common tangent – sequential algorithm

- We previously noted that can compute the upper common tangent sequentially in $O(\log{(n)})$
- The actual $O(\log{(n)})$ algorithms Overmars & van Leeuwen, 1981] and [Kirkpatrick & Snoeyink, 1995] are rather technical
- Here, we will discuss an $O(n)$ algorithm instead, which is much easier to follow
- Later in the course, we will study a constant time parallel version

Upper common tangent – sequential algorithm

- First, let a and b be the rightmost and leftmost vertices of $UH(S_1)$ and $UH(S_2)$, resp.
    - i.e., $a = UH(S_1)[i]$ and $b = UH(S_2)[j]$, for some i, j
- Then, **while** (a, b) dips below either hull
    - **while** (a, b) dips below $UH(S_2)$ (i.e., not tangent)
        - $b = UH(S_2)[j + 1]$
    - **while** (a, b) dips below $UH(S_1)$ (i.e., not tangent)
        - $a = UH(S_1)[i - 1]$
- Intuitively, we alternate between moving b to the right and a to the left

Upper common tangent – sequential algorithm



- Example taken from [O'Rourke, 1997]
- Here, we compute the lower common tangent, but the idea is the same

Upper common tangent – sequential algorithm

- We can check if a point is above or below a line (a, b) by using the equation for a line:

$$(y - y_a)/(x - x_a) = (y_b - y_a)/(x_b - x_a)$$
$$(y_b - y_a)x - (x_b - x_a)y = x_a * y_b - x_b * y_a$$
$$\alpha x + \beta y = \gamma$$

- All the points such that $\alpha x + \beta y < \gamma$ are below the line (and vice versa for above)
- For the UH and LH, we only need to check the two neighbors of the two current candidate points
  - Because the hulls are convex
  - Takes constant sequential time

Partitioning strategy

- The **partitioning strategy** consists of:
  1. Breaking up a problem into p *independent* problems of roughly equal size
  2. Solving the subproblems concurrently
- Differs from divide and conquer:
  - The splits are not (necessarily) recursive
  - The main work lies in partitioning the input, not in combining the solutions of the subproblems
- In the simplest case, we simply break up the data into p non-overlapping chunks
- More generally, we ensure that the subproblems are independent, even if some of the data they access is the same

Partitioning example – merging sorted sequences

- Given a set S with a partial order relation $\leq$, S is **totally ordered** if, $a \leq b$ or $b \leq a$, for all $a, b \in S$
- Given two sorted sequences A and B drawn from S, we want to merge these two sequences into one
- The basic sequential algorithm is $O(n)$
    - Repeatedly move the smallest of the two lists to the final list
- We will explore a parallel solution that partitions A and B into many pairs of subsequences

A simple merging algorithm

- Let $X = (x_1, x_2, \ldots, x_t)$ be a sequence of elements drawn from S (not necessarily sorted)
- The rank of a new element $y \notin X$, $rank(y : X)$ is the number of elements of X that are less than or equal to y
- Let $Y = (y_1, y_2, \ldots, y_s)$ be another list of elements drawn from S
- We want to determine $rank(Y : X)$
  - Example: $X = (25, -13, 26, 31, 54, 7)$ and $Y = (13, 27, -27)$

A simple merging algorithm

- Let $X = (x_1, x_2, \ldots, x_t)$ be a sequence of elements drawn from S (not necessarily sorted)
- The rank of a new element $y \notin X$, $rank(y : X)$ is the number of elements of X that are less than or equal to y
- Let $Y = (y_1, y_2, \ldots, y_s)$ be another list of elements drawn from S
- We want to determine $rank(Y : X)$
  - Example: $X = (25, -13, 26, 31, 54, 7)$ and $Y = (13, 27, -27)$
  - $rank(Y : X) = (2, 4, 0)$
  - Note, $rank(y_i : X)$ is calculated independently for every element of Y
  - The elements of Y are not inserted into X

A simple merging algorithm

- Here, we assume for simplicity that all elements of A and B are distinct
- The merging problem is equivalent to determining the rank of every element from A or B in the union $A \cup B$
  - If $rank(x : A \cup B) = i$, then x should be placed in the i element of the combined list
- Thus, $rank(x : A \cup B) = rank(x : A) + rank(x : B)$ (why?)
- We can solve the merging problem by determining $rank(A : B)$ and $rank(B : A)$ independently

A simple merging algorithm

- Let b_i be an arbitrary element of B
- Since A is sorted, we can find $rank(b_i : A)$ using binary search
  - Runs in $O(\log{(n)})$ (why?)
- If we run $O(n)$ binary searches in parallel, we can solve the merge problem in:

A simple merging algorithm

- Let b_i be an arbitrary element of B
- Since A is sorted, we can find $rank(b_i : A)$ using binary search
  - Runs in $O(\log{(n)})$ (why?)
- If we run $O(n)$ binary searches in parallel, we can solve the merge problem in:
  - $T(n) = O(\log{(n)})$
  - $W(n) = O(n\log{(n)})$
- The work is non-optimal (why?)

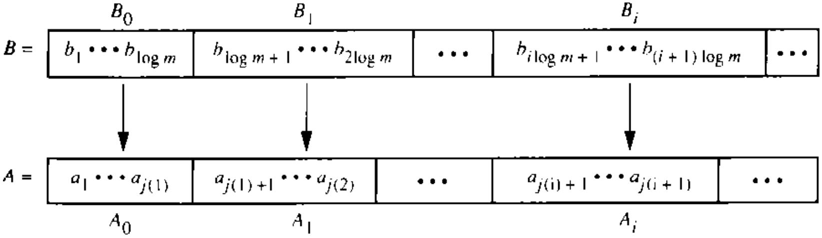
A simple merging algorithm

- Let b_i be an arbitrary element of B
- Since A is sorted, we can find $rank(b_i : A)$ using binary search
  - Runs in $O(\log(n))$ (why?)
- If we run $O(n)$ binary searches in parallel, we can solve the merge problem in:
  - $T(n) = O(\log(n))$
  - $W(n) = O(n\log(n))$
- The work is non-optimal (why?)
  - The sequential algorithm is $O(n)$

An optimal merging algorithm

- We can design an optimal merging algorithm as follows:
  1. Choose approximately $n/\log(n)$ elements of each of A and B that partition A and B into blocks of almost equal lengths
  2. Apply the binary search method to rank each of the chosen elements in the other sequence
- We reduce the problem to merging pairs of $O(\log(n))$ sequences
- For simplicity, though, we will discuss a slight variant in which we only partition B into equal-sized blocks
  - The blocks of A may vary in size

An optimal merging algorithm – partitioning illustration



- Each B_i is of size $\log(m)$
- The A_j blocks could be of different sizes
- Here, $j(i) = rank(b_{i\log(m)} : A)$
  - That is, $A(j) \leq b_{i\log(m)}$, for all $j \leq j(i)$

An optimal merging algorithm – partitioning pseudocode

ALGORITHM 2.7

(Partition)

Input: *Two arrays* $A = (a_1, \dots, a_n)$ *and* $B = (b_1, \dots, b_m)$ *in increasing order, where both* $\log m$ *and* $k(m) = m/\log m$ *are integers.*

Output: $k(m)$ *pairs* (A_i, B_i) *of subsequences of* A *and* B *such that (1)* $|B_i| = \log m$, *(2)* $\sum_i |A_i| = n$, *and (3) each element of* A_i *and* B_i *is larger than each element of* A_{i-1} *or* B_{i-1}, *for all* $1 \leq i \leq k(m) - 1$.

begin

1. *Set* $j(0) := 0, j(k(m)) := n$
2. **for** $1 \leq i \leq k(m) - 1$ **pardo**
   2.1. *Rank* $b_{i \log m}$ *in* A *using the binary search method, and let* $j(i) = rank(b_{i \log m} : A)$
3. **for** $0 \leq i \leq k(m) - 1$ **pardo**
   3.1. *Set* $B_i := (b_{i \log m + 1}, \dots, b_{(i+1) \log m})$
   3.2. *Set* $A_i := (a_{j(i)+1}, \dots, a_{j(i+1)})$
   (A_i *is empty if* $j(i) = j(i + 1)$)

end