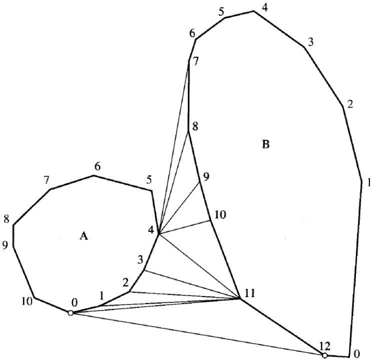
CSc 8530
Parallel Algorithms

Spring 2019

February 19th, 2019

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
- Differers 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

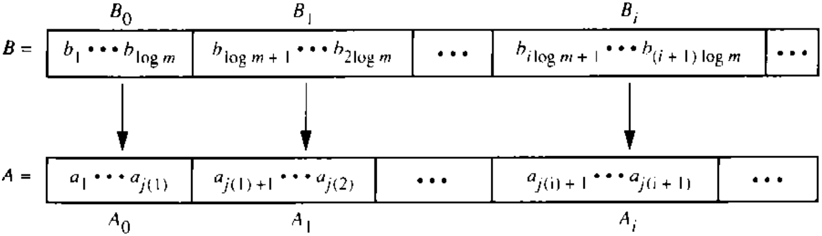
A simple merging algorithm

- 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* $A = (a_1, \ldots, a_n)$ *and* $B = (b_1, \ldots, b_m)$ *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* $1 \le i \le k(m) - 1$.

begin

1. *Set* $j(0) := 0, j(k(m)) := n$

2. **for** $1 \le i \le k(m) - 1$ **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** $0 \le i \le k(m) - 1$ **pardo**

3.1. *Set* $B_i := (b_{i \log m + 1}, \ldots, b_{(i+1)\log m})$

3.2. *Set* $A_i := (a_{j(i)+1}, \ldots, a_{j(i+1)})$

$(A_i$ *is empty if* $j(i) = j(i+1))$

end

An optimal merging algorithm – partitioning analysis

- Let $C = (C_0, C_1, \ldots)$ be the sorted sequence obtained by merging each A_i and B_i
    - By definition, C is equivalent to merging A and B directly (why?)

- Step 1 takes $O(1)$ sequential time
- Step 2 takes $O(\log(n))$ parallel time with $O((\log(n)) \times (m/\log(m)) = O(n+m)$ work
    - Note that $(m \log(n)/\log(m)) < (m \log(n+m)/\log(m)) \leq n+m$, for $n, m \geq 4$
- Step 3 takes $O(1)$ parallel time with $O(n)$ work
- Total running time and work:

An optimal merging algorithm – partitioning analysis

- Let $C = (C_0, C_1, \dots)$ be the sorted sequence obtained by merging each A_i and B_i
    - By definition, C is equivalent to merging A and B directly (why?)
    - $A_{i-1}(j) \leq A_i(j) \leq A_{i+1}(j)$, for all i, j. (Same for B)
- Step 1 takes $O(1)$ sequential time
- Step 2 takes $O(\log(n))$ parallel time with $O((\log(n)) \times (m/\log(m)) = O(n+m)$ work
    - Note that $(m \log(n)/\log(m)) < (m \log(n+m)/\log(m)) \leq n+m$, for $n, m \geq 4$
- Step 3 takes $O(1)$ parallel time with $O(n)$ work
- Total running time and work:

An optimal merging algorithm – partitioning analysis

- Let $C = (C_0, C_1, \ldots)$ be the sorted sequence obtained by merging each A_i and B_i
  - By definition, C is equivalent to merging A and B directly (why?)
  - $A_{i-1}(j) \leq A_i(j) \leq A_{i+1}(j)$, for all i, j. (Same for B)
- Step 1 takes $O(1)$ sequential time
- Step 2 takes $O(\log(n))$ parallel time with $O((\log(n)) \times (m/\log(m)) = O(n+m)$ work
  - Note that $(m \log(n)/\log(m)) < (m \log(n+m)/\log(m)) \leq n+m$, for $n, m \geq 4$
- Step 3 takes $O(1)$ parallel time with $O(n)$ work
- Total running time and work:
  - $T(n) = O(\log n)$

An optimal merging algorithm – partitioning analysis

- Let $C = (C_0, C_1, \dots)$ be the sorted sequence obtained by merging each A_i and B_i
  - By definition, C is equivalent to merging A and B directly (why?)
  - $A_{i-1}(j) \leq A_i(j) \leq A_{i+1}(j)$, for all i, j. (Same for B)
- Step 1 takes $O(1)$ sequential time
- Step 2 takes $O(\log(n))$ parallel time with $O((\log(n)) \times (m/\log(m)) = O(n+m)$ work
  - Note that $(m \log(n)/\log(m)) < (m \log(n+m)/\log(m)) \leq n+m$, for $n, m \geq 4$
- Step 3 takes $O(1)$ parallel time with $O(n)$ work
- Total running time and work:
  - $T(n) = O(\log n)$
  - $W(n) = O(n+m)$

An optimal merging algorithm

- For simplicity, assume A and B are both $O(n)$
- After applying the previous algorithm, we are left with $O(n/\log(n))$ merging subproblems
- We then tackle each subproblem separately
- Let A_i, B_i be an arbitrary subproblem
  - $|B_i| = O(\log(n))$, by construction
  - If $|A_i| = O(\log(n))$, then apply an optimal sequential algorithm to sort these two blocks
  - Otherwise, apply the previous algorithm in reverse:
    - Partition A_i into $O(\log(n))$ blocks
    - This step takes $O(\log\log(n))$ with $O(|A_i|)$ work
    - We then apply the sorting algorithm to each pair of sub-blocks
- Total running time and work:

An optimal merging algorithm

- For simplicity, assume A and B are both $O(n)$
- After applying the previous algorithm, we are left with $O(n/\log(n))$ merging subproblems
- We then tackle each subproblem separately
- Let A_i, B_i be an arbitrary subproblem
  - $|B_i| = O(\log(n))$, by construction
  - If $|A_i| = O(\log(n))$, then apply an optimal sequential algorithm to sort these two blocks
  - Otherwise, apply the previous algorithm in reverse:
    - Partition A_i into $O(\log(n))$ blocks
    - This step takes $O(\log\log(n))$ with $O(|A_i|)$ work
    - We then apply the sorting algorithm to each pair of sub-blocks
- Total running time and work:
  - $T(n) = O(\log(n))$

An optimal merging algorithm

- For simplicity, assume A and B are both $O(n)$
- After applying the previous algorithm, we are left with $O(n/\log(n))$ merging subproblems
- We then tackle each subproblem separately
- Let A_i, B_i be an arbitrary subproblem
    - $|B_i| = O(\log(n))$, by construction
    - If $|A_i| = O(\log(n))$, then apply an optimal sequential algorithm to sort these two blocks
    - Otherwise, apply the previous algorithm in reverse:
        - Partition A_i into $O(\log(n))$ blocks
        - This step takes $O(\log\log(n))$ with $O(|A_i|)$ work
        - We then apply the sorting algorithm to each pair of sub-blocks
- Total running time and work:
    - $T(n) = O(\log(n))$
    - $W(n) = O(n)$

Symmetry breaking

- Some problems have inherent dependencies between subsets of the data
- How we process a given chunk will depend on (potentially arbitrary) choices on how we process other parts of the data
- In other words, our input data is *symmetric* (i.e., indistinguishable)
- But, our processing of it is asymmetric
- We will now see how to parallelize these types of situations

k-coloring a directed ring

- Let $G = (V, E)$ be a directed cycle
  - The in-degree and out-degree are 1
  - For any two vertices, there is a directed path between them
- A **k-coloring** of G is a mapping $c : V \mapsto \{0, 1, \ldots, k-1\}$
  - Such that $c(i) \neq c(j)$ if $(i, j) \in E$
  - In other words, adjacent vertices cannot have the same color
- The minimum coloring problem in general graphs is NP-hard
- For directed cycles, though, we will always need either 2 or 3 colors (why?)

k-coloring a directed ring

- Let $G = (V, E)$ be a directed cycle
  - The in-degree and out-degree are 1
  - For any two vertices, there is a directed path between them
- A **k-coloring** of G is a mapping $c : V \mapsto \{0, 1, \ldots, k-1\}$
  - Such that $c(i) \neq c(j)$ if $(i, j) \in E$
  - In other words, adjacent vertices cannot have the same color
- The minimum coloring problem in general graphs is NP-hard
- For directed cycles, though, we will always need either 2 or 3 colors (why?)
  - 2 colors for even cycles and 3 for odd cycles
- Thus, we will focus on 3-colorings

3-coloring a directed ring

- Given G's restricted topology, it is easy to define an optimal sequential algorithm (how?)

3-coloring a directed ring

- Given G's restricted topology, it is easy to define an optimal sequential algorithm (how?)
  - Follow the cycle, alternating between two colors.
  - If the cycle is odd, use the third color for the last node.
- Unfortunately, this scheme is not amenable to parallelization (why?)

3-coloring a directed ring

- Given G's restricted topology, it is easy to define an optimal sequential algorithm (how?)
  - Follow the cycle, alternating between two colors.
  - If the cycle is odd, use the third color for the last node.
- Unfortunately, this scheme is not amenable to parallelization (why?)
  - We have to arbitrarily map vertices to colors
  - But the choices are **interdependent**

3-coloring a directed ring

- Given G's restricted topology, it is easy to define an optimal sequential algorithm (how?)
  - Follow the cycle, alternating between two colors.
  - If the cycle is odd, use the third color for the last node.
- Unfortunately, this scheme is not amenable to parallelization (why?)
  - We have to arbitrarily map vertices to colors
  - But the choices are **interdependent**
- We need a mechanism for partitioning the vetices into classes such that each class is assigned the same color

A basic coloring algorithm

- We will explore and almost constant-time algorithm for **breaking the node symmetry**
- Assume G is represented by an array S
  - Such that $S(i) = j$ whenever $(i, j) \in E$
  - The predecessor of a node is $P(S(i)) = i$, for all i
- The array is not necessarily sorted based on the path
- Assume that we have an initial coloring c
  - We can start with $c(i) = i$, if needed
  - Let $i_{t-1} \dots i_k \dots i_1 i_0$ be the **binary expansion** of i
  - The kth least significant bit is i_k
- We will use this binary representation to reduce the number of colors

A basic coloring algorithm – pseudocode

ALGORITHM 2.9

(Basic Coloring)

Input: *A directed cycle whose arcs are specified by an array S of size n and a coloring c of the vertices.*

Output: *Another coloring c' of the vertices of the cycle.*

begin

for $1 \leq i \leq n$ **pardo**

1. *Set k to the least significant bit position in which $c(i)$ and $c(S(i))$ disagree.*
2. *Set $c'(i) := 2k + c(i)_k$*

end