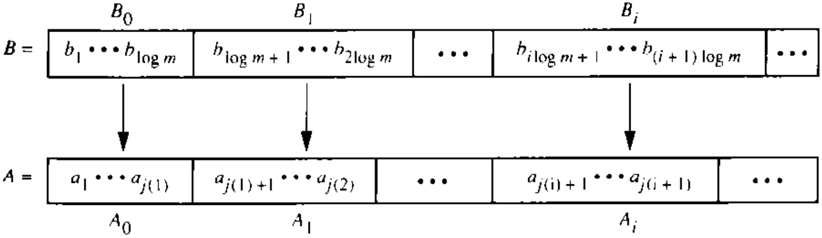
CSc 8530
Parallel Algorithms

Spring 2019

February 21st, 2019

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

- 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:
  - $T(n) = O(\log(n))$
  - $W(n) = O(n)$

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

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 $i_{t-1} \ldots i_k \ldots i_1 i_0$ 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

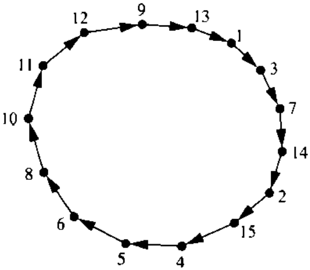
ALGORITHM 2.9
(Basic Coloring)

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

A basic coloring algorithm – example



v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

- We reduce the number of colors from 15 to 6

A basic coloring algorithm – analysis

- We will first show correctness: if c is a valid coloring, then c' will also be valid
- **Proof by contradiction:**

- We have a single parallel **for** loop for every vertex
  - With enough processors, it can be executed in one iteration
- Total running time and work:

A basic coloring algorithm – analysis

- We will first show correctness: if c is a valid coloring, then c' will also be valid
- **Proof by contradiction:**
  - Since c is a coloring, $c(i) \neq c(j)$, for all $(i, j) \in E$, so k always exists
  - Now, suppose $c'(i) = c'(j)$
  - Then, $c'(i) = 2k + c(i)_k$ and $c'(j) = 2l + c(j)_l$
  - Since $c'(i) = c'(j)$, then $k = l$
  - However, this would imply that $c(i)_k = c(j)_k$, which contradicts the definition of k
- We have a single parallel **for** loop for every vertex
  - With enough processors, it can be executed in one iteration
- Total running time and work:

A basic coloring algorithm – analysis

- We will first show correctness: if c is a valid coloring, then c' will also be valid
- **Proof by contradiction:**
  - Since c is a coloring, $c(i) \neq c(j)$, for all $(i, j) \in E$, so k always exists
  - Now, suppose $c'(i) = c'(j)$
  - Then, $c'(i) = 2k + c(i)_k$ and $c'(j) = 2l + c(j)_l$
  - Since $c'(i) = c'(j)$, then $k = l$
  - However, this would imply that $c(i)_k = c(j)_k$, which contradicts the definition of k
- We have a single parallel **for** loop for every vertex
  - With enough processors, it can be executed in one iteration
- Total running time and work:
  - $T(n) = O(1)$ and $W(n) = O(n)$

A fast 3-coloring algorithm

- We will now modify our basic algorithm to achieve a 3-coloring
- First, note that we can apply the previous algorithm iteratively
  - Let $t > 3$ be number of bits used to represent the q colors in c
  - Then, each color in c' can be represented with $\lceil \log(t) \rceil + 1$ bits
  - So, c' uses at most $2^{\lceil \log(t) \rceil + 1} = O(t) = O(\log(q))$ colors
    - The number of colors decreases exponentially
- We can apply the previous algorithm iteratively,
- Converges to a 6-coloring for all reasonable n (why?, how fast?)

A fast 3-coloring algorithm

- We will now modify our basic algorithm to achieve a 3-coloring
- First, note that we can apply the previous algorithm iteratively
    - Let $t > 3$ be number of bits used to represent the q colors in c
    - Then, each color in c' can be represented with $\lceil \log(t) \rceil + 1$ bits
    - So, c' uses at most $2^{\lceil \log(t) \rceil + 1} = O(t) = O(\log(q))$ colors
        - The number of colors decreases exponentially
- We can apply the previous algorithm iteratively,
- Converges to a 6-coloring for all reasonable n (why?, how fast?)
    - **Why:** when $t = 3$, $2k + c(i)_k$ ranges from 0 to 5

A fast 3-coloring algorithm

- We will now modify our basic algorithm to achieve a 3-coloring
- First, note that we can apply the previous algorithm iteratively
    - Let $t > 3$ be number of bits used to represent the q colors in c
    - Then, each color in c' can be represented with $\lceil \log(t) \rceil + 1$ bits
    - So, c' uses at most $2^{\lceil \log(t) \rceil + 1} = O(t) = O(\log(q))$ colors
        - The number of colors decreases exponentially
- We can apply the previous algorithm iteratively,
- Converges to a 6-coloring for all reasonable n (why?, how fast?)
    - **Why:** when $t = 3$, $2k + c(i)_k$ ranges from 0 to 5
    - **How fast:** After one iteration, we reduce the colors to $O(\log(n))$, after two to $O(\log(\log(n))) = O(\log^{(2)}(n))$
    - If $n \leq 2^{65536}$, then $\log^{(m)}(n) \leq 5$

A fast 3-coloring algorithm

- We go from six to three colors as follows:
  - **Parfor** each $c(i) = 3 : 5$
    - Set $c(i)$ to the smallest value from 0:2 that is different from its two neighbors
- The above procedure is correct because we never change any two neighboring nodes at the same time
- Running time and work of recoloring:
- Algorithm's total running time and work:

A fast 3-coloring algorithm

- We go from six to three colors as follows:
  - **Parfor** each $c(i) = 3:5$
    - Set $c(i)$ to the smallest value from 0:2 that is different from its two neighbors
- The above procedure is correct because we never change any two neighboring nodes at the same time
- Running time and work of recoloring:
  - $T(n) = O(1)$ and $W(n) = O(n)$
- Algorithm's total running time and work:

A fast 3-coloring algorithm

- We go from six to three colors as follows:
  - **Parfor** each $c(i) = 3:5$
    - Set $c(i)$ to the smallest value from 0:2 that is different from its two neighbors
- The above procedure is correct because we never change any two neighboring nodes at the same time
- Running time and work of recoloring:
  - $T(n) = O(1)$ and $W(n) = O(n)$
- Algorithm's total running time and work:
  - $T(n) = O(\log^{(m)}(n))$, $m \leq 5$ for most n

A fast 3-coloring algorithm

- We go from six to three colors as follows:
  - **Parfor** each $c(i) = 3 : 5$
    - Set $c(i)$ to the smallest value from 0:2 that is different from its two neighbors
- The above procedure is correct because we never change any two neighboring nodes at the same time
- Running time and work of recoloring:
  - $T(n) = O(1)$ and $W(n) = O(n)$
- Algorithm's total running time and work:
  - $T(n) = O(\log^{(m)}(n))$, $m \leq 5$ for most n
  - $W(n) = O(n \log^{(m)}(n))$
- The algorithm is constant-time for practical-sized inputs, but weakly non-optimal
- For those interested, the book presents an optimal algorithm

General remarks

- Understand the algorithms studied in class
  - e.g., given a forest F, draw the first iteration of the pointer-jumping algorithm
  - Analyze running time, convergence rate, etc.
- Understand the properties of every parallel model
  - e.g., imagine you have algorithm A for parallel model M (dag, PRAM, or network). Apply the WT scheduling principle to efficiently schedule A on M, etc.
- Analyze pseudocode
  - Turn a sequential version into a parallel one (parfor, etc.)
  - Analyze the running time, work, cost, etc.
- Analyze or write pseudocode for *variants* of the problems seen in class
  - e.g., prefix sums on a different data structure, divide and conquer on a graph, etc.

Parallel speedup

- Let P be a computational problem with inputs of size n
- We denote the best-possible sequential (i.e., classic) complexity of P as $T^*(n)$
- Let A be a parallel algorithm that solves P in time $T_p(n)$ using p processors
- Then, the **speedup** achieved by A is:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- By construction, $S_p(n) \leq p$
- We would like $S_p(n) \approx p$
    - i.e., each processor should do around $1/p$ of the work of a single one
- In practice, inefficiencies in concurrency, synchronization, communication, etc. reduce the actual speedup

Parallel efficiency

- The **efficiency** of a parallel algorithm A is given by:

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

- $T_1(n)$ is the running time of the parallel algorithm with a single processor
  - Not necessarily equal to $T^*(n)$
- Efficiency measures how much bang for our buck we get per processor
- Ideally, $E_p(n) \approx 1$
- Again, inefficiencies reduce this value in practice

Dag model

- In the dag model, we assume that:
    - *Every processor can access the data computed by any other processor without incurring additional cost*
- A particular implementation is defined by **scheduling** each node for execution on a processor
- Given p processors, we associate a pair (j_i, t_i) with each internal node i:
    - j_i is the processor used for node i
    - t_i is the time at which we process node i
- The following two conditions must hold:
    - If $t_i = t_k$, for some $i \neq k$, then $j_i \neq j_k$
        - Each processor can only process one node at a time
    - If (i, k) is an edge, then $t_k \geq t_i + 1$
        - Node i has to be processed before node k

Dag model

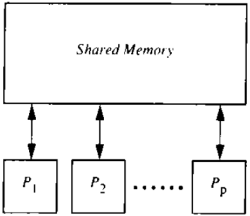
- Input nodes have $t_i = 0$ and no processor is allocated to them
- The sequence $\{(j_i, t_i) \,|\, i \in N\}$ is an execution **schedule**
    - With p processors
    - N is the number of nodes in the dag
- The time to execute a particular schedule is $\max_{i \in N} t_i$
- The **parallel complexity** is

$$T_p(n) = \min \{\max_{i \in N} t_i\}$$

- The minimum is taken over all possible schedules with p processors
- The depth of the dag is a lower bound on $T_p(n)$, for any p

The shared-memory model

- A natural extension of the sequential RAM model
- Many processors have access to a single, **shared memory** unit (also called **global memory**)
- Each processor also has its own **local memory**
- Processors communicate by exchanging data through the shared memory
- Each processor is indexed by a unique id



The shared-memory model

- In **synchronous** mode, all processors operate in lock-step, under a common clock
- In **asynchronous** mode, each has a independent clock
- Synchronous mode is called the **parallel random-access machine (PRAM)** model
  - It it the model we will primarily study in this class
- Asynchronous mode requires additional checks to make sure the data is up-to-date when accessed
- Both models are **multiple instruction multiple data (MIMD)**
- The amount of **communication** is given by the size of data transferred via the shared memory
- A **global read** (X, Y) moves the variable X into the local memory Y
- A **global write** (U, V) does the opposite

The network model

- A **network** is a graph $G = (V, E)$
  - The nodes V are the processors
  - The edges E are two-way communication links between processors
- There is **no** shared memory
  - Each processor does have local memory
- The model can be either **synchronous** or **asynchronous**
- **send**(X, i) instruction: sends X to processor P_i (and continue executing the next instruction immediately)
- **receive**(Y, j) operation: wait for Y from processor P_j (and suspend execution until data is received)

The network model

- The processors of an asynchronous network coordinate their activities through **message passing**
  - A pair of processors need not be adjacent
  - **Routing** algorithms transmit a message through a network
- The topological properties of the network affects the system's processing capabilities:
  - **Diameter:** maximum distance between any two nodes
  - **Maximum degree:** of any node in G
  - **Node and edge connectivity:** the minimum number of nodes (edges) whose removal disconnects the graph
- Some representative topologies:
  - Linear array
  - 2D mesh
  - Hypercube

Worst-case analysis

- Let Q be a problem that we can solve in $T(n)$ with $P(n)$ processors
- **Parallel cost:** $C(n) = T(n)P(n)$
- The parallel algorithm can be converted to a sequential algorithm that runs in $O(C(n))$
- More generally, we can simulate a single step in $O(P(n)/p)$ sub-steps:
  - In sub-step 1: simulate processors $[1, p]$
  - In sub-step 2: simulate processors $[p + 1, 2p]$, etc.
- We can simulate the entire process in $O(T(n)P(n)/p)$

Work vs. cost

- If a parallel algorithm runs in $T(n)$ with a total of $W(n)$ operations
    - Can be simulated in $O(\frac{W(n)}{p} + T(n))$ on a p-processor PRAM
    - The cost is $C_p(n) = T_p(n)p = O(W(n) + T(n)p)$
- Work and cost coincide asymptotically for $p = O(\frac{W(n)}{T(n)})$
- Otherwise they differ:
    - Work is independent of the number of processors
    - Cost is measured relative to the number of available processors
    - Cost $\geq$ Work due to inefficient processor utilization
- For computing the sum of n numbers:
    - Work: $O(n)$, running time: $O(\log(n))$
    - Cost: $C_p(n) = O(n + p\log(n))$
    - With n processors, the cost is $O(n\log(n))$, not $O(n)$ (Why?)

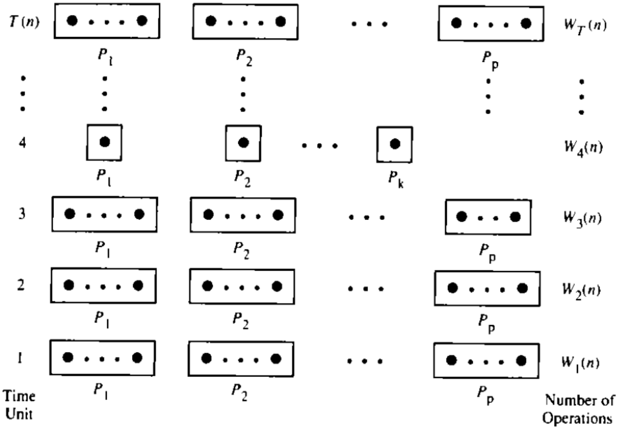
Work vs. cost

- If a parallel algorithm runs in $T(n)$ with a total of $W(n)$ operations
  - Can be simulated in $O(\frac{W(n)}{p} + T(n))$ on a p-processor PRAM
  - The cost is $C_p(n) = T_p(n)p = O(W(n) + T(n)p)$
- Work and cost coincide asymptotically for $p = O(\frac{W(n)}{T(n)})$
- Otherwise they differ:
  - Work is independent of the number of processors
  - Cost is measured relative to the number of available processors
  - Cost $\geq$ Work due to inefficient processor utilization
- For computing the sum of n numbers:
  - Work: $O(n)$, running time: $O(\log(n))$
  - Cost: $C_p(n) = O(n + p\log(n))$
  - With n processors, the cost is $O(n\log(n))$, not $O(n)$ (Why?)
  - We cannot use all the processors at all time steps, so the cost is higher than the total work

Work-time (WT) paradigm

- The **work-time (WT) paradigm** provides a two-level description of parallel algorithms
  - Upper level suppresses specific details
  - Lower level follows a general **scheduling principle**
- **Upper Level:** Describe the algorithm in terms of a sequence of time units
  - Each time unit may include any number of concurrent operations
- **Work:** total number of operations
- For convenience, at this level we can use a **pardo** statement
  - **for** $l \leq i \leq u$ **pardo** $\{$statement(s)$\}$
  - All the statements, for all valid indices, are executed concurrently

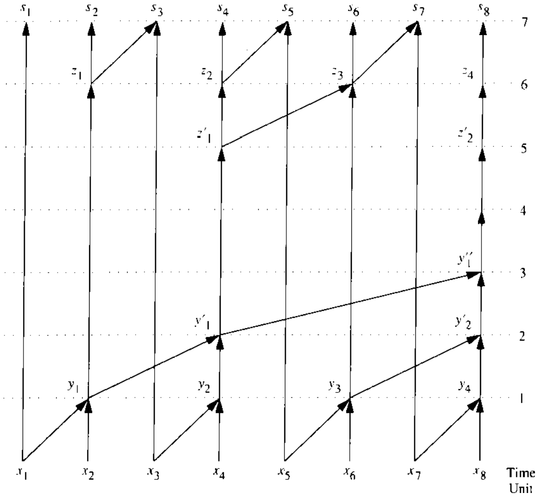
WT Scheduling Principle



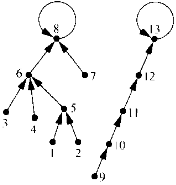
Optimality notions

- 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

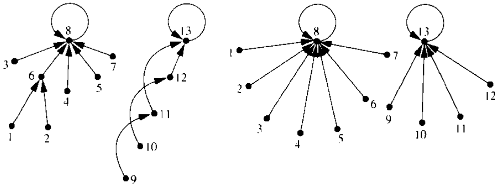
Recursive prefix-sums algorithm



Root finding: examples

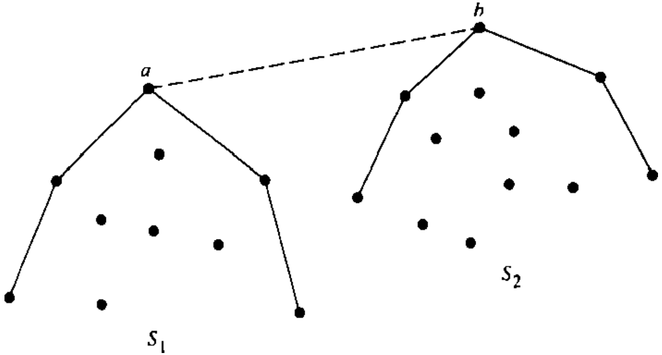


(a)



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

Upper common tangent example



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