CSc 8530
Parallel Algorithms

Spring 2019

February 5th, 2019

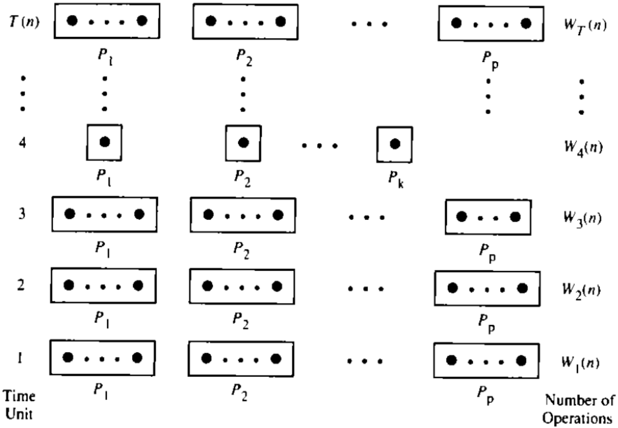
Worst-case analysis

- 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-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



WT vs. lower-level pseudocode

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$

```
begin
  1. for 1 ≤ i ≤ n pardo
        Set B(i): = A(i)
  2. for h = 1 to log n do
        for 1 ≤ i ≤ n/2^h pardo
            Set B(i): = B(2i − 1) + B(2i)
  3. Set S: = B(1)
end
```

WT pseudocode

ALGORITHM 1.8
(Sum Algorithm for Processor P_s)
Input: *An array A of size $n = 2^k$ stored in the shared memory. The initialized local variables are (1) the order n; (2) the number p of processors, where $p = 2^q \leq n$, and (3) the processor number s.*
Output: *The sum of the elements of A stored in the shared variable S. The array A retains its original value.*

```
begin
  1. for j = 1 to l (= n/p) do
        Set B(l(s − 1) + j): = A(l(s − 1) + j)
  2. for h = 1 to log n do
        2.1. if (k − h − q ≥ 0) then
                for j = 2^(k−h−q)(s − 1) + 1 to 2^(k−h−q)s do
                    Set B(j): = B(2j − 1) + B(2j)
        2.2. else {if (s ≤ 2^(k−h)) then
                    Set B(s): = B(2s − 1) + B(2s)}
  3. if (s = 1) then set S: = B(1)
end
```

Lower-level pseudocode

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

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

Algorithmic techniques

- Designing parallel algorithms involves additional challenges compared to sequential methods
- We will now review some basic techniques for breaking down a problem into parallel chunks
- The example problems will often arise as sub-problems in more complicated applications

Balanced trees

- We have already encountered balanced binary trees
  - e.g., for summing the values of an array
- **General strategy:** *build a balanced binary tree on the input elements and traverse the tree forwards and backwards*
- An internal node u usually holds information about the data stored in the leaves of the subtree rooted at u
  - This strategy is useful when we can calculate this information quickly

Example: prefix sums

- Let $S = \{x_1, x_2, \ldots, x_n\}$ 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:

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

- 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 $O(n)$

Example: prefix sums

- We can use a balanced binary tree to compute the prefix sums in $O(\log(n))$
- We compute pairwise $*$ operations during the forward pass
- Each internal node will hold the sum of the elements stored in the leaves of its subtree
- During the backward pass, we compute the prefix sums at each level of the tree

Recursive prefix-sums algorithm

ALGORITHM 2.1

(Prefix Sums)

Input: *An array of* $n = 2^k$ *elements* $(x_1, x_2, \ldots, x_n)$, *where* k *is a nonnegative integer.*

Output: *The prefix sums* s_i, *for* $1 \le i \le n$.

begin

1. **if** $n = 1$ **then** {*set* $s_1 := x_1$; **exit**}
2. **for** $1 \le i \le n/2$ **pardo**

    Set $y_i := x_{2i-1} * x_{2i}$
3. *Recursively, compute the prefix sums of* $\{y_1, y_2, \ldots, y_{n/2}\}$, *and store them in* $z_1, z_2, \ldots, z_{n/2}$.
4. **for** $1 \le i \le n$ **pardo**

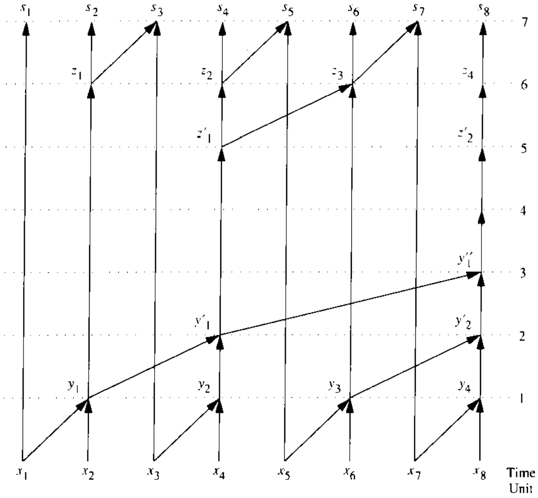
    $\{i$ *even* : *set* $s_i := z_{i/2}$

    $i = 1$: *set* $s_1 := x_1$

    i *odd* > 1 : *set* $s_i := z_{(i-1)/2} * x_i\}$

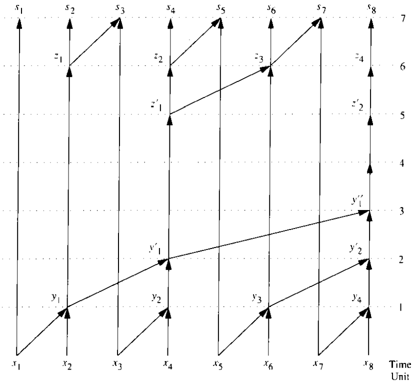
end

Recursive prefix-sums algorithm



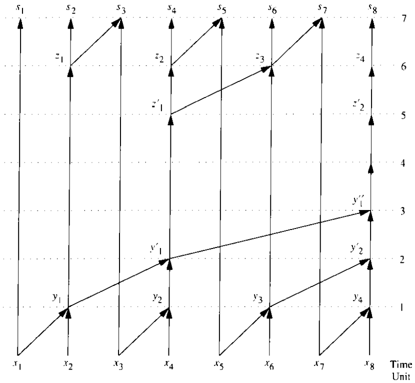
Recursive prefix-sums algorithm

- **First time unit:** we compute $y_1 = x_1 * x_2$, etc.
- **Second:** $y'_1 = y_1 * y_2$ and $y'_2 = y_3 * y_4$
- **Third:** $y"_1 = y'_1 * y'_2$
- **Fourth:** We generate the prefix sum of the n elements



Recursive prefix-sums algorithm

- **Fifth:** generate z'_1 and z'_2 from y'_1 and y'_2, resp.
- **Sixth:** z_1 to z_4 from y_1 to y_4
- **Seventh:** compute s_i for each i using the z and x values



Recursive prefix-sums algorithm – analysis

- For inputs of size $n = 2^k$, this algorithm requires $2^k + 1$ time units
  - We move up the k-level binary tree in the forward step
  - And down k steps in the backward step
- The algorithm can run in-place
  - In the sense that the y and z variables can be mapped to each other
- It is straightforward to show that this algorithm has:
  - **Running time:**
  - **Work:**

Recursive prefix-sums algorithm – analysis

- For inputs of size $n = 2^k$, this algorithm requires $2^k + 1$ time units
  - We move up the k-level binary tree in the forward step
  - And down k steps in the backward step
- The algorithm can run in-place
  - In the sense that the y and z variables can be mapped to each other
- It is straightforward to show that this algorithm has:
  - **Running time:**
  - $T(n) = O(\log(n))$
  - **Work:**

Recursive prefix-sums algorithm – analysis

- For inputs of size $n = 2^k$, this algorithm requires $2^k + 1$ time units
  - We move up the k-level binary tree in the forward step
  - And down k steps in the backward step
- The algorithm can run in-place
  - In the sense that the y and z variables can be mapped to each other
- It is straightforward to show that this algorithm has:
  - **Running time:**
  - $T(n) = O(\log(n))$
  - **Work:**
  - $W(n) = O(n)$

Recursive prefix-sums algorithm – analysis

- **Proof by induction:**
- The base case $k = 0$ is handled by step 1 of the algorithm
- Assume the algorithms works for $n = 2^k$
- We will prove it computes the prefix sums for $n = 2^{k+1}$
- The variables $z_1, z_2, \ldots, z_{n/2}$ hold the prefix sums of the sequence $\{y_1, y_2, \ldots, y_{n/2}\}$
- In particular,

$$\begin{aligned} z_j &= y_1 * y_2 * \ldots y_j \\ &= x_1 * x_2 * \ldots x_{2j-1} * x_{2j} \end{aligned}$$

- Thus, $z_j = s_{2j}$ for $1 \leq j \leq n/2$

Recursive prefix-sums algorithm – analysis

- **Proof by induction:**
- If i is even, then $s_i = z_{i/2}$
- If i is odd (and > 1):

$$\begin{aligned} s_i &= s_{2j+1} \\ &= s_{2j} * x_{2j+1} \\ &= z_{(i-1)/2} * x_i \end{aligned}$$

- All cases are handled appropriately, thus the algorithm works correctly for all inputs

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-sum algorithm

- The previous algorithm was recursive
- We can easily develop a non-recursive, yet still parallel version
- Here, we use auxiliary arrays $B(h, j)$ (forward values) and $C(h, j)$ (backward values) to simplify data storage
- Where $0 \leq h \leq \log(n)$ and $1 \leq j \leq n/2^h$
- For simplicity of analysis, we assume $n = 2^k$, for some k

Non-recursive prefix-sums algorithm

ALGORITHM 2.2

(Nonrecursive Prefix Sums)

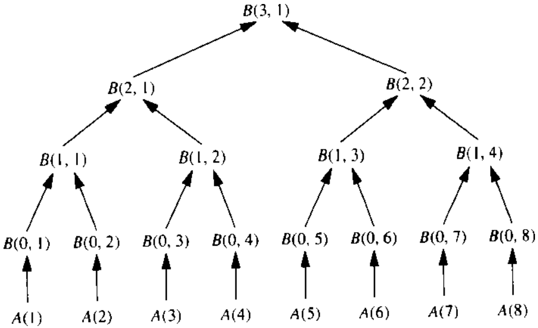
Input: *An array A of size* $n = 2^k$, *where k is a nonnegative integer.*
Output: *An array C such that* $C(0, j)$ *is the jth prefix sum, for* $1 \le j \le n$.

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

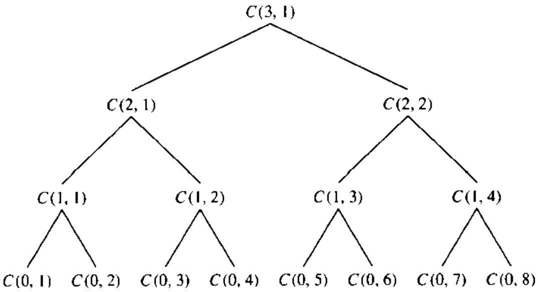
end

Non-recursive prefix-sums algorithm



- Similar to the problem of summing an individual array
- In the non-recursive version, we store all intermediate values in an auxiliary array

Non-recursive prefix-sums algorithm



- Here, we recurse forwards and backwards

Recursive vs. non-recursive versions

ALGORITHM 2.1
(Prefix Sums)
Input: *An array of* $n = 2^k$ *elements* $(x_1, x_2, \ldots, x_n)$, *where* k *is a nonnegative integer.*
Output: *The prefix sums* s_i, *for* $1 \le i \le n$.
begin
1. **if** $n = 1$ **then** {*set* $s_1 := x_1$; **exit**}
2. **for** $1 \le i \le n/2$ **pardo**
   Set $y_i := x_{2i-1} * x_{2i}$
3. *Recursively, compute the prefix sums of* $\{y_1, y_2, \ldots, y_{n/2}\}$, *and store them in* $z_1, z_2, \ldots, z_{n/2}$.
4. **for** $1 \le i \le n$ **pardo**
   {i *even* : *set* $s_i := z_{i/2}$
   $i = 1$: *set* $s_1 := x_1$
   i *odd* > 1 : *set* $s_i := z_{(i-1)/2} * x_i$}

end

Recursive

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

end

Non-recursive

Review

- Building a balanced (binary) tree is a fundamental technique
- One of the most useful in parallel processing
- Other example problems:
  - Broadcasting a value to all processors
  - Compacting the labeled elements of an array
- Can be generalized to non-binary trees
  - Example: computing the maximum of n elements