COMP 322: Fundamentals of Parallel Programming

Lecture 8: Parallel N-Queens algorithm, Parallel Prefix Sum Algorithm

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #7 solution: Why must Future References be declared as final?

1) Consider the code on the right with futures declared as non-final static fields (though that's not permitted in HJ). Can a deadlock situation occur between tasks T1 and T2 with this code? Explain why or why not.

Yes, a deadlock can occur when future f1 does f2.get() and future f2 does f1.get().

WARNING: such "spin" loops are an example of bad parallel programming practice in application code (they should only be used by expert systems programmers, and even then sparingly) Their semantics depends on the memory model. In HJ's memory model, there's no guarantee that the above spin loops will ever terminate.

```
static future<int> f1=null;
static future<int> f2=null;

void main(String[] args) {
  f1 = async<int> {return a1();};
  f2 = async<int> {return a2();};
}

int a1() { // Task T1
  while (f2 == null); // spin loop
  return f2.get(); //T1 waits for T2
}

int a2() { // Task T2
  while (f1 == null); // spin loop
  return f1.get(); //T2 waits for T1
}
```

deadlock

Worksheet #7 solution: Why must Future References be declared as final

2) Now consider a modified version of the above code in which futures are declared as final local variables (which is permitted in HJ). Can you add get() operations to methods a1() and a2() to create a deadlock between tasks T1 and T2 with this code? Explain why or why not.

No, the final declarations make it impossible for future f1's task (T1) to receive a reference to f2.

Will your answer be different if f1 and f2 are final fields in objects or final static fields?

No.

```
void main(String[] args) {
  final future<int> f1 =
    async<int> {return a1();};
  final future<int> f2 =
    async<int> {return a2(f1);};
}

int a1() {
// Task T1 cannot receive a
// reference to f2

}

int a2(future<int> f1) {
// Task T2 can receive a reference
// to f1 but that won't cause
// a deadlock.
... f1.get() ...
}
```

Outline of Today's Lecture

- **<u>Parallel N-Queens Algorithm</u>**
- **Parallel Prefix Sum Algorithm**

Acknowledgments

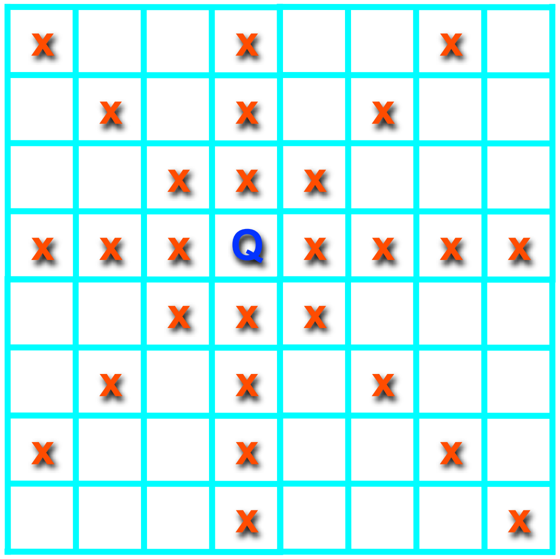
- COMP 322 Module 1 handout, Chapter 6, Chapter 7
- Marc Pomplun, U.Mass Boston, CS 271, Introduction to Cognitive Science, Lecture 17: Game-Playing Algorithms
  - http://www.cs.umb.edu/~marc/cs271/cog11-10.ppt
- Prof. Kathy Yelick, UC Berkeley, CS 194 Lecture, Fall 2007
  - http://www.cs.berkeley.edu/~yelick/cs194f07/lectures/lect09-dataparallel.pdf

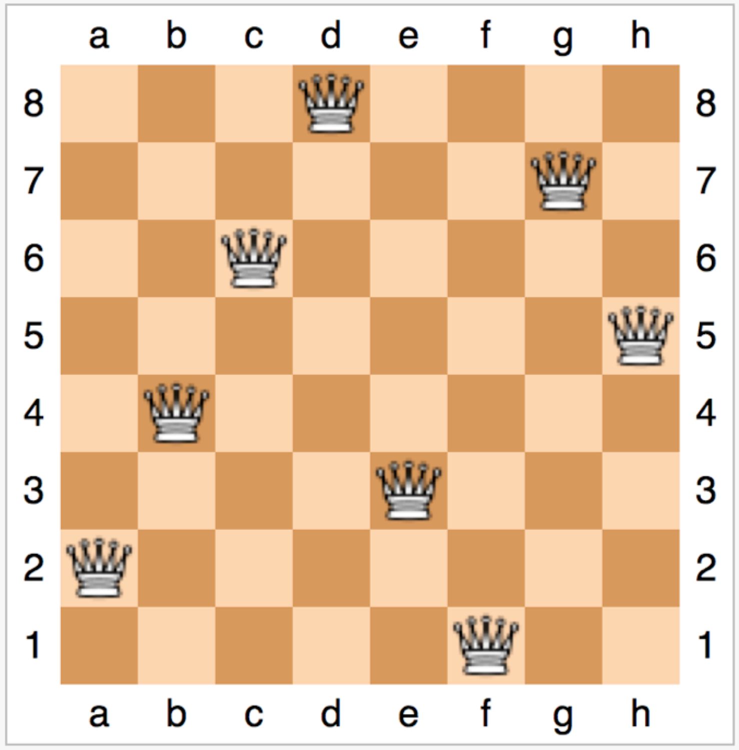
The N-Queens Problem

How can we place n queens on an n×n chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.

Here, the possible target squares of the queen Q are marked with an x.





One solution to the eight queens puzzle

Decision Trees

- In any solution of the n-queens problem, there must be exactly one queen in each column of the board.
- Otherwise, the two queens in the same column could capture each other.
- Therefore, we can describe the solution of this problem as a sequence of n decisions:
- Decision 1: Place a queen in the first column.
- Decision 2: Place a queen in the second column.
- .
  .
  .

  Decision n: Place a queen in the n-th column.
- Since there are multiple choices for each decision, we get a "decision tree"

Backtracking in Decision Trees

- **There are problems that require us to perform an exhaustive search of all possible sequences of decisions in order to find the solution.**
- **We can solve such problems by constructing the complete decision tree and then find a path from its root to a leaf that corresponds to a solution of the problem**
- **In many cases, the efficiency of this procedure can be dramatically increased by a technique called backtracking (depth-first search).**

Backtracking and Decision Tree states

- Idea: Start at the root of the decision tree and move downwards, that is, make a sequence of decisions, until you either reach a solution or you enter a state from where no solution can be reached by any further sequence of decisions.
- In the latter case, backtrack to the parent of the current state and take a different path downwards from there. If all paths from this state have already been explored, backtrack to its parent.
- Continue this procedure until you find a solution (or all solutions), or establish that no solution exists.
- A state in the decision tree can be encoded as an array, a[0..c-1] for c columns, where a[i] = row position of queen in column i.

Backtracking in Decision Trees

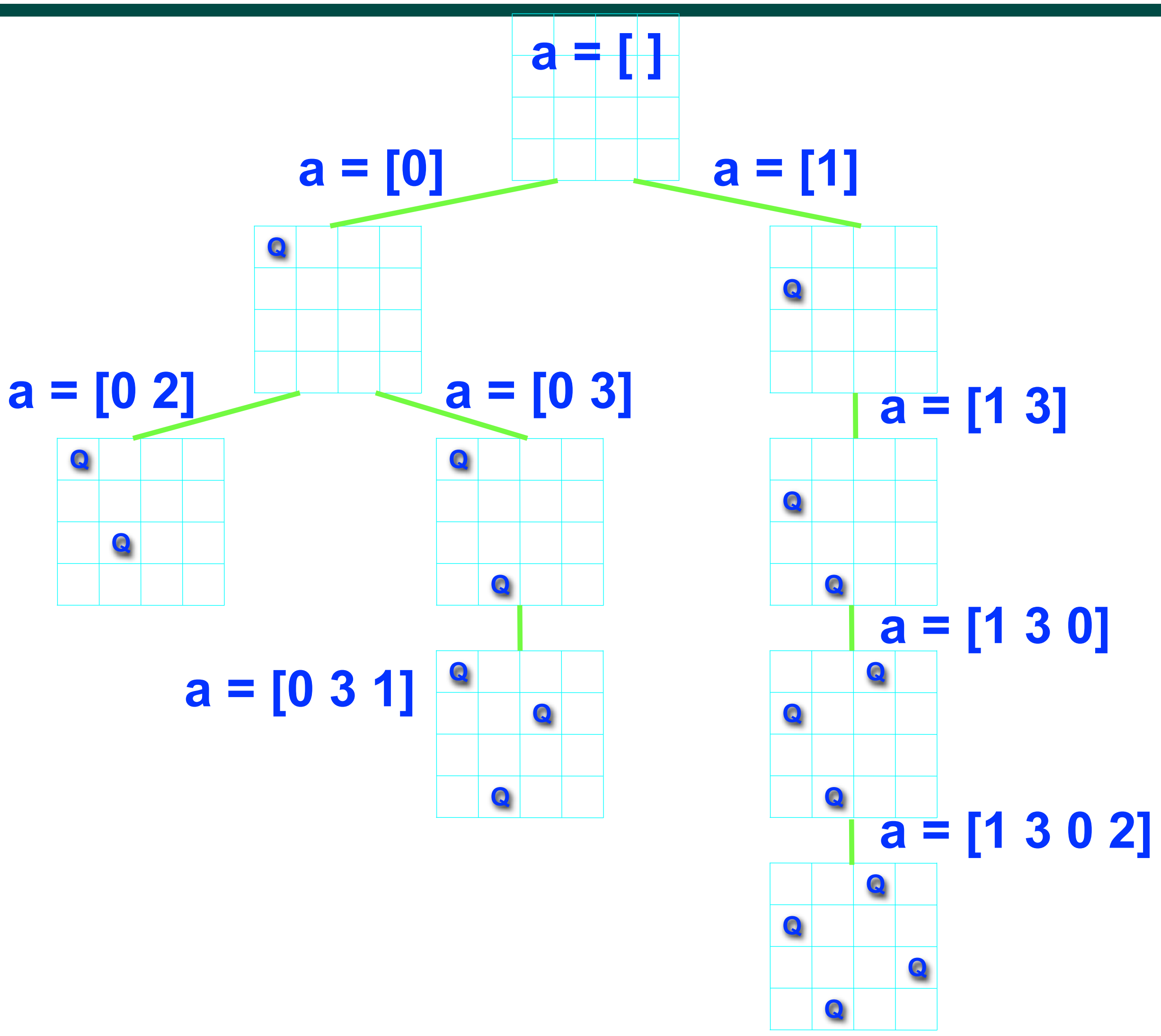
empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen



Sequential solution for NQueens (counting all solutions)

```
static int count;
. . .
count = 0;
nqueens_kernel(new int[0], 0);
System.out.println(“No. of solutions = “ + count);
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count++;
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Parallel Solution to NQueens with Finish Accumulators (counting all solutions)

```
static accumulator count;
. . .
count = accumulator.factory.accumulator(SUM, int.class);
finish(a) nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + count.get().intValue());
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count.put(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Outline of Today's Lecture

- **Parallel N-Queens Algorithm**
- **<u>Parallel Prefix Sum Algorithm</u>**

Acknowledgments

- COMP 322 Module 1 handout, Chapter 6, Chapter 7
- Marc Pomplun, U.Mass Boston, CS 271, Introduction to Cognitive Science, Lecture 17: Game-Playing Algorithms
  - http://www.cs.umb.edu/~marc/cs271/cog11-10.ppt
- Prof. Kathy Yelick, UC Berkeley, CS 194 Lecture, Fall 2007
  - http://www.cs.berkeley.edu/~yelick/cs194f07/lectures/lect09-dataparallel.pdf

Prefix Sum (Scan) Problem Statement

Given input array A, compute output array X as follows

$$X[i] = \sum_{0 \leq j \leq i} A[j]$$

- **The above is an <u>inclusive</u> prefix sum since X[i] includes A[i]**
- **For an <u>exclusive</u> prefix sum, perform the summation for 0 <=j <i**
- **It is easy to see that prefix sums can be computed sequentially in O(n) time**

```
// Copy input array A into output array X

X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);

// Update array X with prefix sums

for (int i=1 ; i < X.length ; i++ ) X[i] += X[i-1];
```

An Inefficient Parallel Prefix Sum program

```
finish {
 for (int i=0 ; i < X.length ; i++ )
   // computeSum() adds A[0..i] in parallel
   async X[i] = computeSum(A, 0, i);
}
```

Observations:

- **Critical path length, CPL = O(log n)**
- **Total number of operations, WORK = O(n^2)**
- **With P = O(n) processors, the best execution time that you can achieve is T_P = max(CPL, WORK/P) = O(n), which is no better than sequential!**

How can we do better?

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$\begin{aligned} X[6] &= A[0]+A[1]+A[2]+A[3]+A[4]+A[5]+A[6] \\ &= (A[0]+A[1]+A[2]+A[3])+(A[4]+A[5])+A[6] \end{aligned}$$

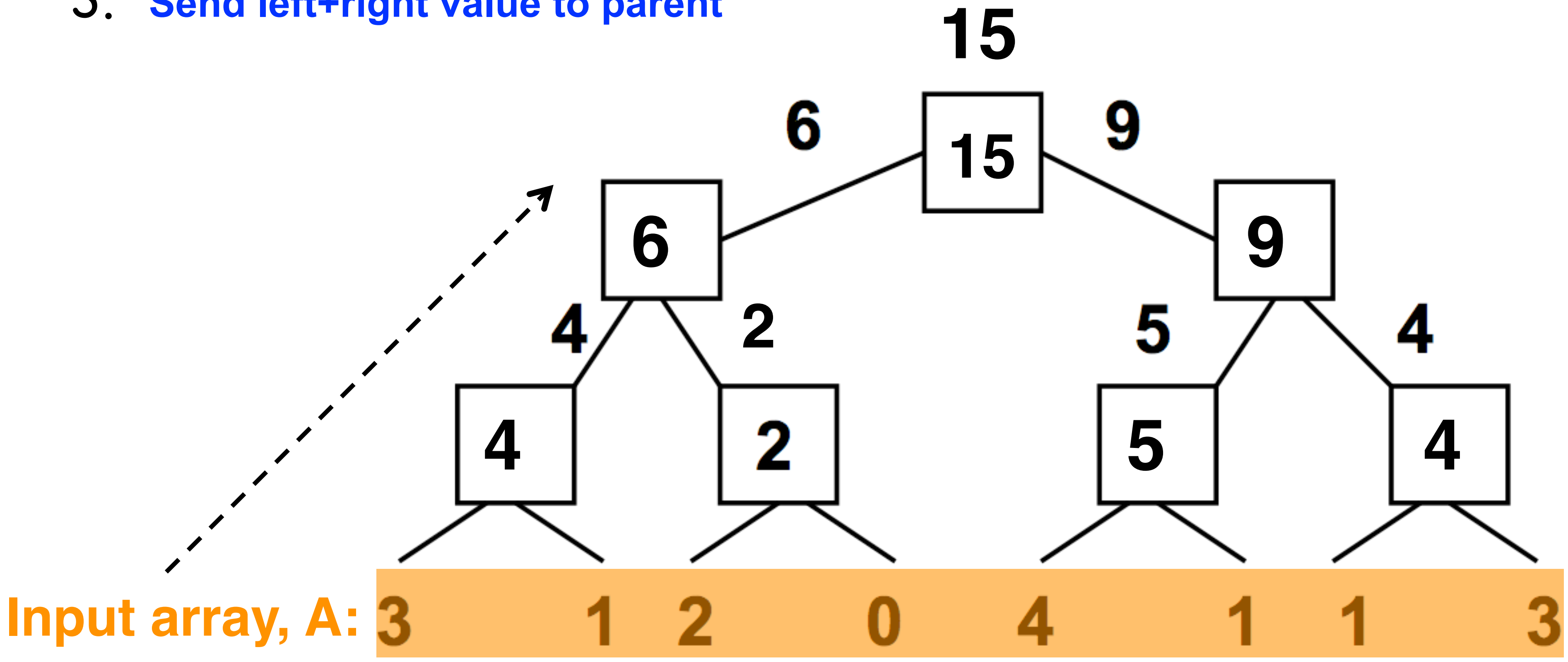
Approach:

- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an "upward sweep" to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a "downward sweep" to compute prefix sums while reusing partial sum terms stored in upward sweep

Parallel Prefix Sum: Upward Sweep (Alternate formulation to Lecture 10)

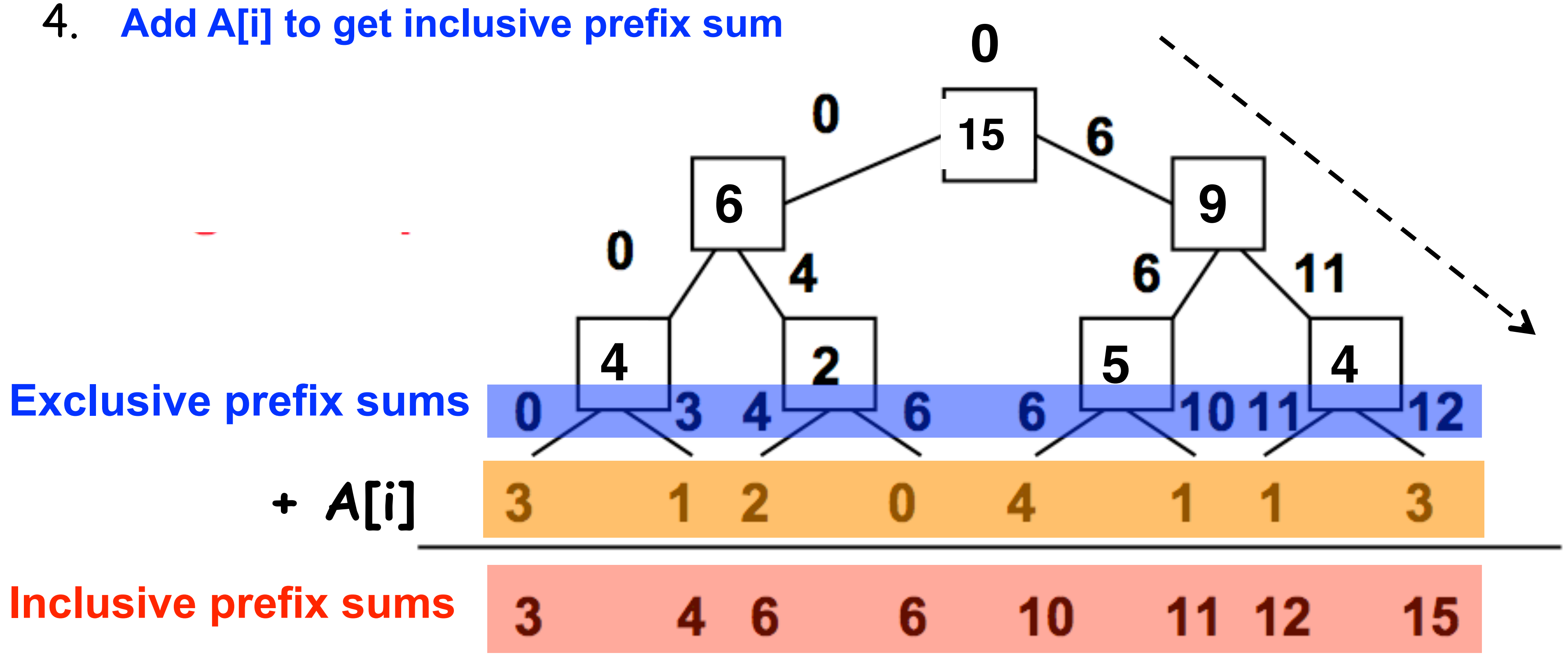
Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent



Parallel Prefix Sum: Downward Sweep

1. Receive value from parent (root receives 0)
2. Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)
3. Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)
4. Add A[i] to get inclusive prefix sum



Summary of Parallel Prefix Sum Algorithm

- Critical path length, CPL = O(log n)
- Total number of add operations, WORK = O(n)
- Optimal algorithm for P = O(n/log n) processors
  - —Adding more processors does not help
- Parallel Prefix Sum has several applications that go beyond computing the sum of array elements
- Parallel Prefix Sum can be used for any operation that is associative (need not be commutative)
  - —In contrast, finish accumulators require the operator to be both associative and commutative

How do associativity and commutativity make a difference?

Time for worksheet #8!

Example Applications of Parallel Prefix Algorithm

- <u>Prefix Max with Index of First Occurrence</u>: given an input array A, output an array X of objects such that X[i].max is the maximum of elements A[0…i] and X[i].index contains the index of the first occurrence of X[i].max in A[0…i]
  - Homework 2 includes this problem just for the entire array (not intermediate prefix "sums")
- <u>Filter and Packing of Strings</u>: given an input array A identify elements that satisfy some desired property (e.g., uppercase), and pack them in a new output array. (First create a 0/1 array for elements that satisfy the property, and then compute prefix sums to identify locations of elements to be packed.)
  - Useful for parallelizing partitioning step in Parallel Quicksort algorithm (Approaches 2 and 3)

Use of Prefix Sums to parallelize partition() in Quicksort (Approach 2, Summary of Listing 30)

```
1. partition(int[] A, int M, int N) { // choose pivot from M..N
2.  forall (point [k] : [0:N-M]) { // parallel loop
3.   lt[k] = (A[M+k] < A[pivot] ? 1 : 0);  // bit vector with < comparisons
4.   eq[k] = (A[M+k] == A[pivot] ? 1 : 0); // bit vector with = comparisons
5.   gt[k] = (A[M+k] > A[pivot] ? 1 : 0);  // bit vector with > comparisons
6.   buffer[k] = A[M+k];                    // Copy A[M..N] into buffer
7.  }
8.  Copy lt, eq, gt, into ltPS, eqPS, gtPS before step 9
9.  final int ltCount = computePrefixSums(ltPS); //update lt with prefix sums
10. final int eqCount = computePrefixSums(eqPS); //update eq with prefix sums
11. final int gtCount = computePrefixSums(gtPS); //update gt with prefix sums
12. // Parallel move from buffer into A
13. forall (point [k] : [0:N-M]) {
14.   if(lt[k]==1) A[M+ltPS[k]-1] = buffer[k];
15.   else if(eq[k]==1) A[M+ltCount+eqPS[k]-1] = buffer[k];
16.   else A[M+ltCount+eqCount+gtPS[k]-1] = buffer[k];
17.  }
18.  . . .
19.}  // partition
```

Worksheet #8: Associativity and Commutativity

Name 1: ______________________   Name 2: ______________________

Finish accumulators can be used for any associative and commutative binary function.
Parallel Prefix Sum algorithm can be used for any associative binary function.

A binary function f is *associative* if f(f(x,y),z) = f(x,f(y,z)).
A binary function f is *commutative* if f(x,y) = f(y,x).

For each of the following functions, indicate if it can be used in a finish accumulator or a parallel prefix sum algorithm or both or neither.

1) f(x,y) = x+y, for integers x, y

2) g(x,y) = (x+y)/2, for integers x, y

3) h(s1,s2) = concat(s1, s2) for strings s1, s2 e.g., h("ab","cd") = "abcd"

Use the space below for your answers.