COMP 322: Fundamentals of Parallel Programming

Lecture 10: Abstract vs. Real Performance, Parallel Prefix Sum, Associative vs. Commutative Operators

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

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

Outline

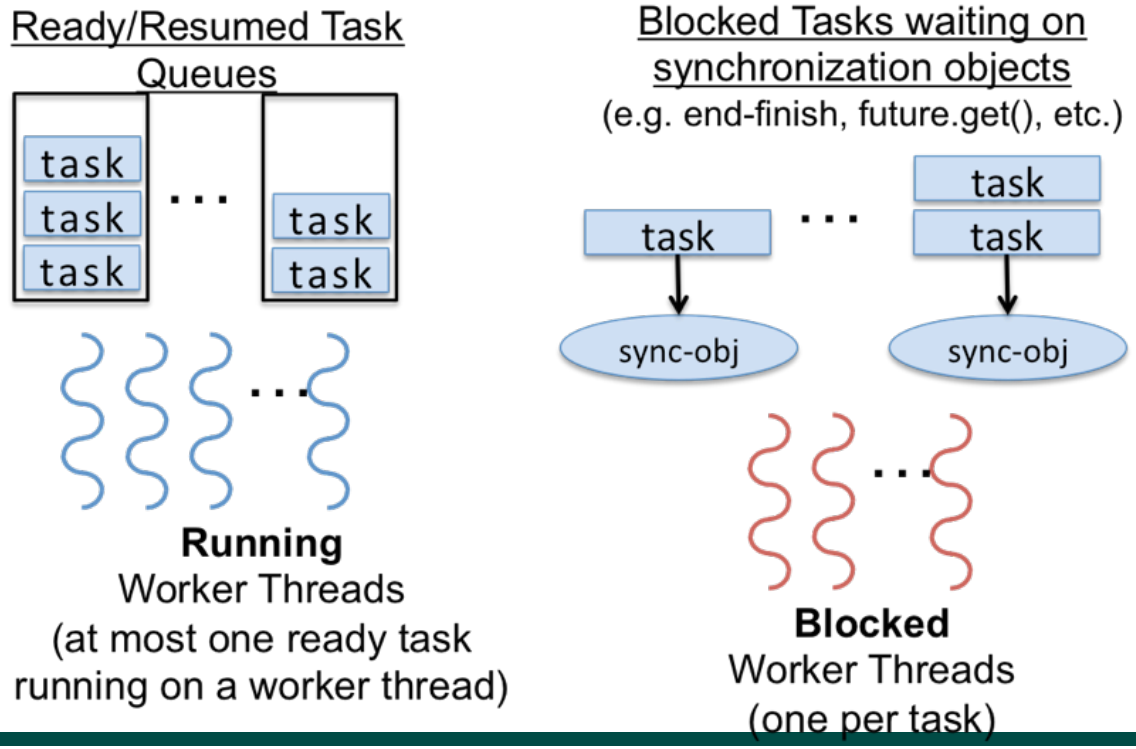
- **Abstract vs. Real Performance**
- **Parallel Prefix Sum Algorithm**
- **Associative vs. Commutative Operators**

Acknowledgments

- COMP 322 Module 1 lecture handout

Abstract vs. Real Performance in HJlib

- **Abstract Performance**
  - —Abstract metrics focus on operation counts for WORK and CPL, regardless of actual execution time
- **Real Performance**
  - —HJlib uses ForkJoinPool implementation of Java Executor interface



We'll learn about these Java libraries later in the course --- they manage parallelism at a lower level than HJ

Some Real-world Performance Issues

- Overhead in creating async tasks can overshadow benefits from parallelism
  - —Need to control granularity of async tasks by using cut-off limits for parallelism e.g., depth cutoff for recursive parallelism, iteration grouping (chunking) for loop parallelism
- Wait operations (finish, future get) increase the number of blocked worker threads
  - —May need to increase limit on blocked threads, e.g.,
    - System.setProperty(HjSystemProperty.maxThreads.propertyKey(), ”200”);
  - —Extra “context switch” overheads are incurred when blocked worker threads become unblocked

Problem: creating too many small async tasks can be a source of overhead (recursive ArraySum2)

```
1.  static int computeSum(int[] X, int lo, int hi) {
2.    if ( lo > hi ) return 0;
3.    else if ( lo == hi ) return X[lo];
4.    else {
5.      int mid = (lo+hi)/2;
6.      final future<int> sum1 =
7.            future { return computeSum(X, lo, mid); };
8.      final future<int> sum2 =
9.           future { return computeSum(X, mid+1, hi); };
10.     // Parent now waits for the container values
11.     return sum1.get() + sum2.get();
12.    }
13.  } // computeSum
14. int sum = computeSum(X, 0, X.length-1); // main program
```

Creating a future for each method call leads to too many tasks!

Iterative Fork-Join Microbenchmark

```
1. finish {
2.     for (int i=1; i<k; i++)
3.         async Ti;  // task i
4.     T0; //task 0
5. }
```

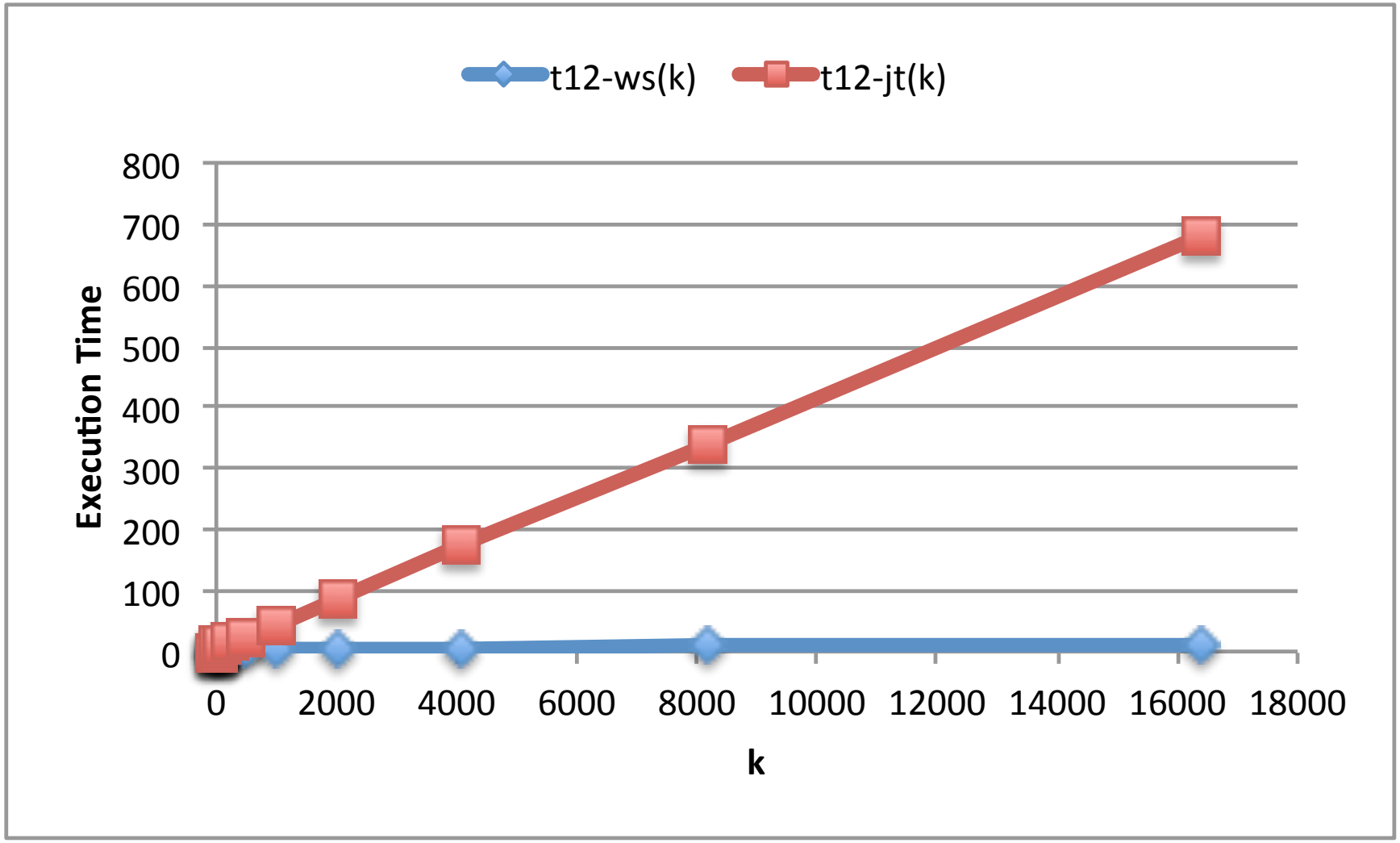
Single-Worker execution times to model overhead (Section 9.2.1)

- **k = number of tasks**
- **$t_s(k)$ = sequential time**
- **$t_1^{ws}(k)$ = 1-worker time for work-sharing**
- **$t_{12}^{ws}(k)$ = 12-worker time for work-sharing**
- **$t_1^{jt}(k)$ = create a Java thread for each async, run on 1 core**
- **$t_{12}^{jt}(k)$ = create a Java thread for each async, run on 12 cores**

Fork-Join Microbenchmark Measurements
(execution time in milliseconds)

k	$t_s(k)$	$t_1^{ws}(k)$	$t_1^{jt}(k)$		$t_{12}^{ws}(k)$	$t_{12}^{jt}(k)$
1	0.00550	1.67180	0.00264		2.05552	0.00874
2	0.00640	1.61984	0.64944		2.00564	0.49688
4	0.00752	1.67401	1.26081		1.90105	1.78910
8	0.00962	1.68423	5.39852		2.39091	4.59425
16	0.01117	1.71121	7.49290		3.00064	5.35545
32	0.01341	2.04591	8.14587		3.29302	7.01638
64	0.01962	2.07918	11.07557		3.02092	7.79610
128	0.02337	2.07780	12.03547		3.56084	10.19936
256	0.05199	2.13682	17.67796		3.74917	19.37605
512	0.07282	2.29679	28.28268		4.18872	24.16757
1024	0.14978	2.63632	51.30504		4.86663	44.30347
2048	0.31606	2.99007	90.20563		5.62178	89.07167
4096	0.57622	3.61543	175.49042		6.28115	177.19051
8192	0.75838	8.55980	333.09688		7.89189	336.22245
16384	1.07625	9.50611	667.73758		9.47608	678.99951

Comparing Overheads for Work-Sharing and Java-Threads



Common fix in parallel divide-and-conquer algorithms --- add a "threshold" test

```
// Minimum size for which an async task is justified
int thresholdSize = 1000000;
. . .
int mid = (lo+hi)/2;
int size = hi - lo + 1;
if (size < thresholdSize) { // Sequential case
  sum1 = computeSum(X, lo, mid); sum2 = computeSum(X, mid+1, hi);
}
else { // Parallel case --- pseudocode
  sum1 = async computeSum(X, lo, mid); sum2 = async computeSum(X, mid+1,
hi);
}
. . .
```

- **The "size < thresholdSize" condition ensures that async tasks are only created for upper nodes in the reduction tree; lower nodes (closer to the leaves) are executed sequentially.**
- **A large thresholdSize value leads to larger async tasks with less (shallower) parallelism**
- **A small thresholdSize value leads to smaller async tasks with more (deeper) parallelism**

seq clause for async statements (pseudocode)

async seq(cond) <stmt> ≡ if (cond) <stmt> else async <stmt>

```
1. // Async task
2. async seq(size < thresholdSize) computeSum(X, lo, mid);
3.
4. // Future example
5. final future<int> sum1 = future seq(size < thresholdSize)
6.                                  { return computeSum(X, lo,
   mid); };
```

- "seq" clause specifies condition under which async should be executed sequentially
  - False ⇒ an async is created
  - True ⇒ the parent executes async body sequentially
- Avoids the need to duplicate code for both cases

Threshold Condition depends on application e.g., NQueens

```
1.  static accumulator count;
2.  . . .
3.  count = accumulator.factory.accumulator(SUM, int.class);
4.  finish(a) nqueens_kernel(new int[0], 0);
5.  System.out.println("No. of solutions = " + count.get().intValue());
6.  . . .
7.  void nqueens_kernel(int [] a, int depth) {
8.    if (size == depth) count.put(1);
9.    else
10.     /* try each possible position for queen at depth */
11.     for (int i =  0; i < size; i++) async seq(depth >= cutoff) {
12.       /* allocate a temporary array and copy array a into it */
13.       int [] b = new int [depth+1];
14.       System.arraycopy(a, 0, b, 0, depth);
15.       b[depth] = i;
16.       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
17.     } // for-async
18. } // nqueens_kernel()
```


Use of asyncSeq API in HJlib (Quicksort example)

```
1.protected static void quicksort(
2.   final Comparable[] A, final int M, final int N) {
3.        if (M < N) {
4.            // A point in HJ is an integer tuple
5.            HjPoint p = partition(A, M, N);
6.            int I = p.get(0);
7.            int J = p.get(1);
8.            asyncSeq(I - M > 5, () -> quicksort(A, M, I));
9.            asyncSeq(N - J > 5, () -> quicksort(A, J, N));
10.        }
11.    }
```


Outline

- **Abstract vs. Real Performance**
- **Parallel Prefix Sum Algorithm**
- **Associative vs. Commutative Operators**

Acknowledgments

- COMP 322 Module 1 lecture handout
- 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 inclusive prefix sum since X[i] includes A[i]**
- **For an exclusive 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

```
1. finish {
2.   for (int i=0 ; i < X.length ; i++ )
3.     // computeSum() adds A[0..i] in parallel
4.     async X[i] = computeSum(A, 0, i);
5. }
```

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.

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

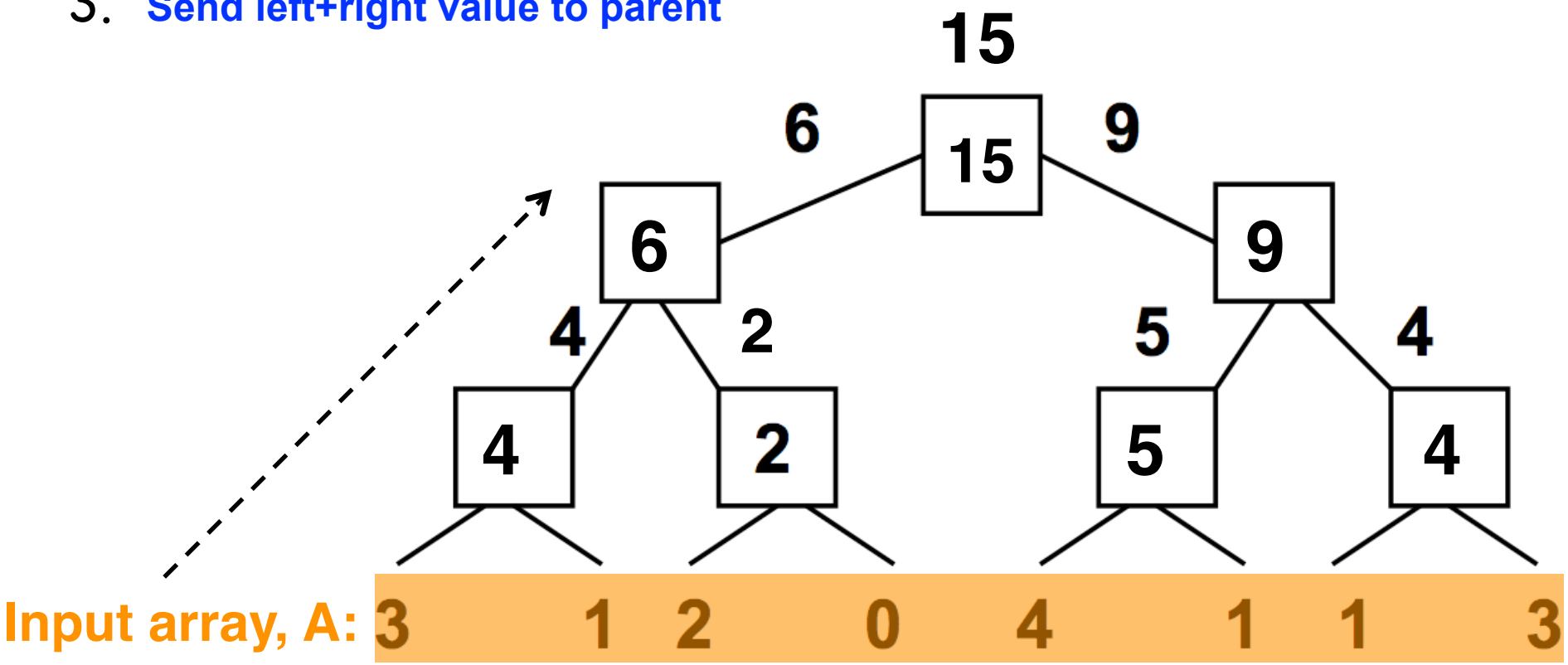
Approach:

- **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

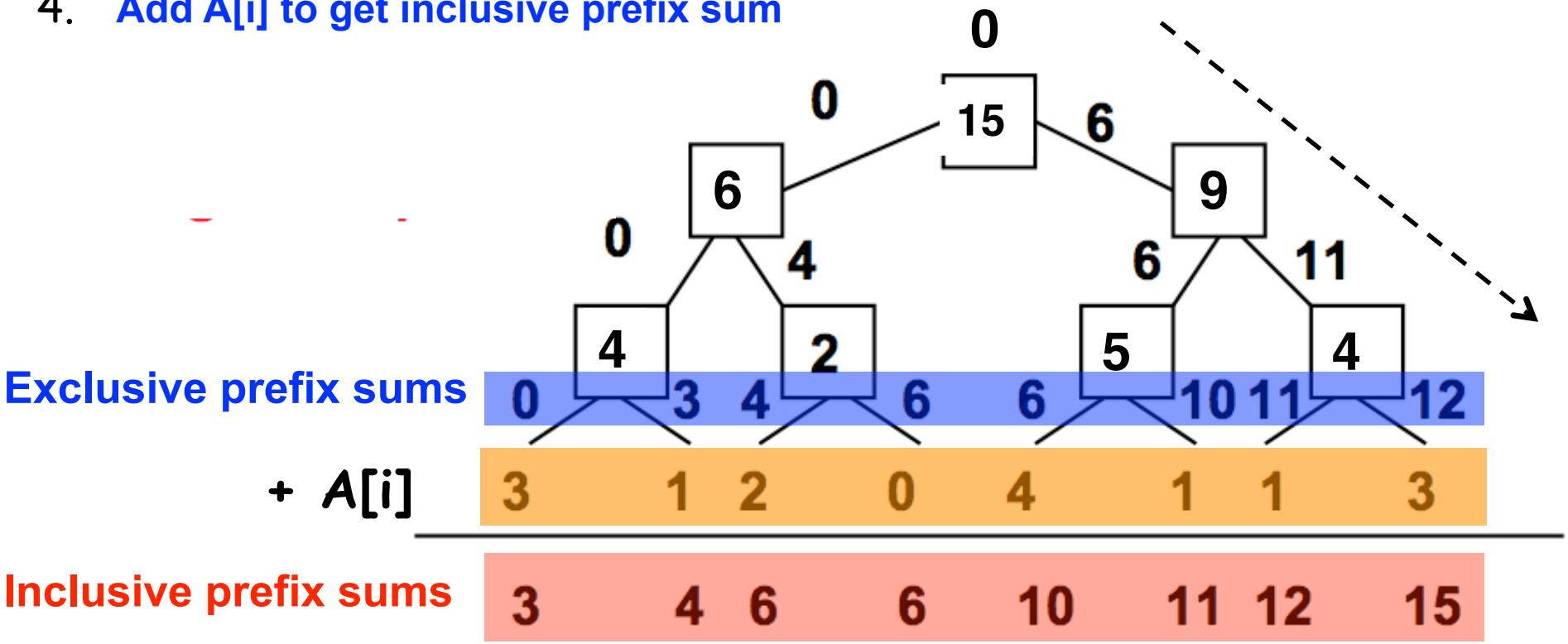
Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

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

Example Applications of Parallel Prefix Algorithm

- Prefix Max with Index of First Occurrence: 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")
- Filter and Packing of Strings: 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
```


Outline

- **Abstract vs. Real Performance**
- **Parallel Prefix Sum Algorithm**
- **Associative vs. Commutative Operators**

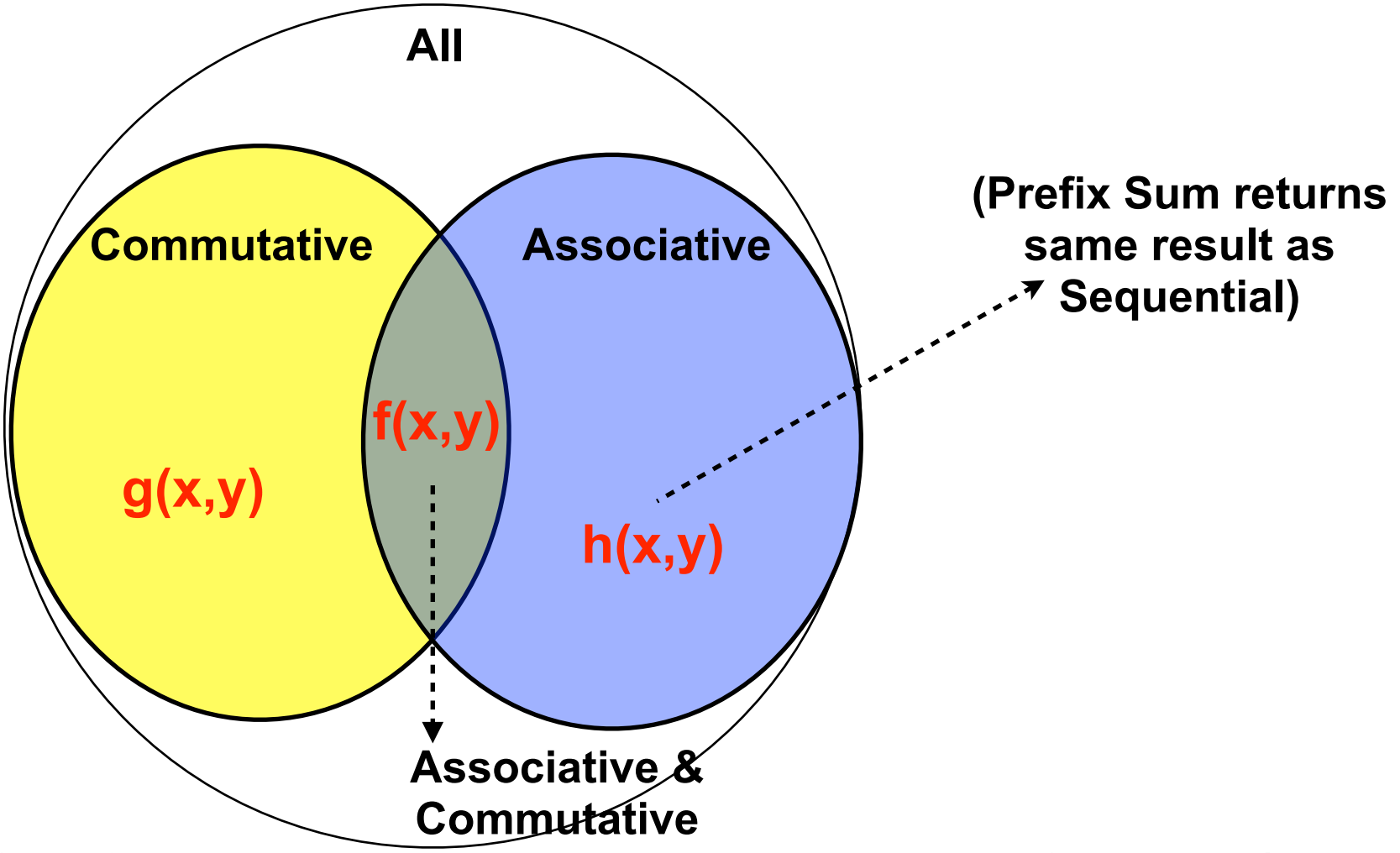
Acknowledgments

- COMP 322 Module 1 lecture handout

Generalized Reduce

- Basic idea: given a binary function, *f(x,y)*, and an identity element, *i*, compute the reduction of an array X[0], X[1], ... as follows
  - Reduction = f(f(f(i,X[0]),X[1]) ...), which can be computed sequentially as follows
    - temp := i; // identity element
    - temp := f(temp, X[0]); // f(i,X[0])
    - temp := f(temp, X[1]); // f(f(i,X[0]),X[1])
    - . . .
- In Homework 2, you have to write an HJ program to compute the reduction in parallel i.e., to obtain the same answer as the sequential version, assuming that *f(x,y)* is associative and commutative.
  - *f(x,y)* is specified by the `combine()` method and the identity element is specified by the `init()` method

Venn diagram of binary functions



(Prefix Sum & Finish Accumulator return same result as Sequential)

Worksheet #10: Associativity and Commutativity

Name: _____ Netid: _____

A Finish Accumulator (FA) can be used with any *associative and commutative* binary function. The Parallel Prefix Sum (PPS) algorithm can be used with 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 (FA) or a Parallel Prefix Sum (PPS) 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"