COMP 322: Fundamentals of Parallel Programming

Lecture 7: Parallel Prefix Sum, Forall Statement

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

Announcements

- Homework 3 is due by 5pm on Monday, Feb 7th
  - This is a programming assignment with abstract performance metrics
  - To prepare for HW3, please make sure that you can compile and run the programs from Lab 2 on your own, using the –perf option. In case of problems, please send email to comp322-staff @ mailman.rice.edu
- We have requested 24-hour access to Ryon building and Ryon 102 lab for all students enrolled in COMP 322

Acknowledgments for Today's Lecture

- Prof. Kathy Yelick, UC Berkeley, CS 194 Lecture, Fall 2007
  - http://www.cs.berkeley.edu/~yelick/cs194f07/lectures/lect09-dataparallel.pdf
- PLDI 2007 tutorial on X10 co-authored with Vijay Saraswat and Christoph von Praun
- COMP 322 Lecture 6 handout

Prefix Sum (Scan) Problem Statement

Given input array A, compute output array X as follows

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

Observations:

- Mathematical specification may suggest that *$O(n^2)$* additions are required since each *X[i]* is the sum of *i* terms
- However, 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++ )
    // invoke computeSum() function from Lecture 5
    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.

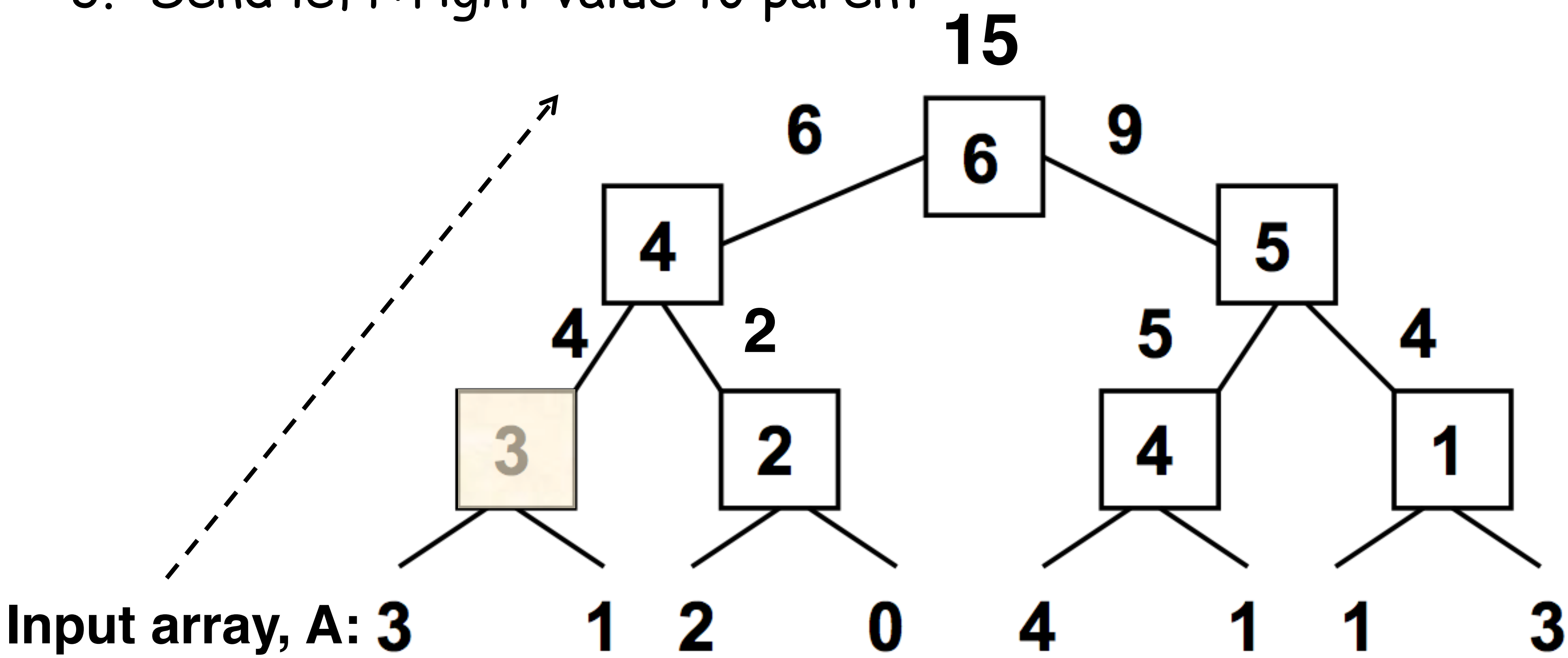
$$\begin{aligned} X[6] &= A[0]+A[1]+A[2]+A[3]+A[4]+A[5]+A[6] \\ &= (A[0]+A[1]+A[2]+A[3])+(A[4]+A[5])+A[6] \end{aligned}$$

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

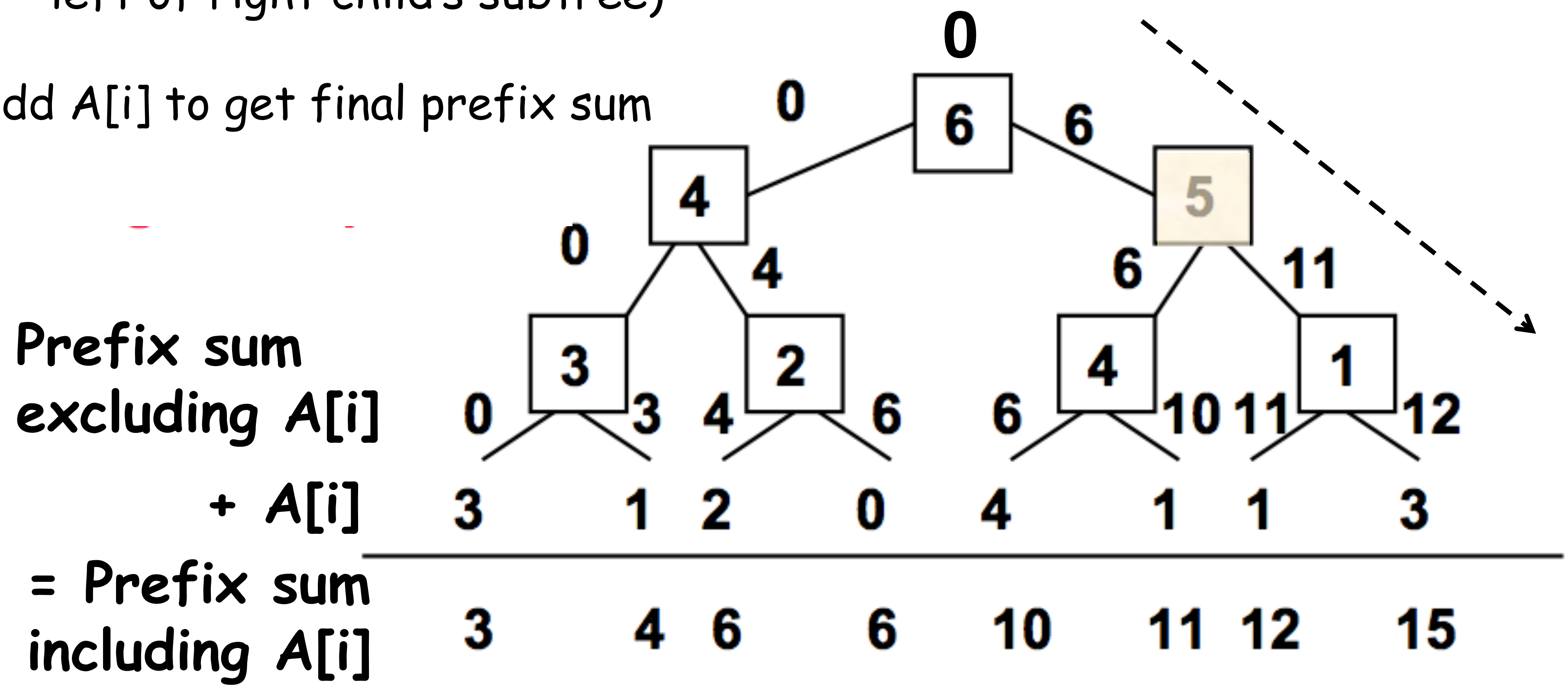
1. Receive values from children
2. Store left value in box (will contribute to prefix sum for right subtree in downward sweep)
3. Send left+right value to parent



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+box value to right child (prefix sum for elements to left of right child's subtree)

Add A[i] to get final 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
- Like Array Sum Reduction, Parallel Prefix Sum has several applications that go beyond computing the sum of array elements e.g.,
  - 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]
  - 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.)

HJ's forall statement

Goal: capture common finish-for-async pattern in a single construct e.g., replace

```
finish {
  for (int I = 0 ; I < N ; I++)
    for (int J = 0 ; J < N ; J++)
      async
        for (int K = 0 ; K < N ; K++)
          C[I][J] += A[I][K] * B[K][J];
}
```

by

```
forall (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

Observations

- Combination of finish-for-async is replaced by a single keyword, forall
- Multiple loops can be collapsed into a single forall, with a multi-dimensional iteration space.
- Iteration variable for a forall is a point (integer tuple) such as [I,J]
- Loop bounds can be specified as a rectangular region (dimension ranges) such as [0:N-1,0:N-1]
- HJ also extends the sequential for statement so as to iterate sequentially over a rectangular region
  - Simplifies conversion between for and forall

Points

- A *point* is an element of an n-dimensional Cartesian space (n>=1) with integer-valued coordinates e.g., [5], [1, 2], …
  - Dimensions are numbered from 0 to n-1
  - n is also referred to as the *rank* of the point
- A point variable can hold values of different ranks e.g.,
  - point p; p = [1]; … p = [2,3]; …
- The following operations are defined on a point-valued expression p1
  - p1.rank --- returns rank of point p1
  - p1.get(i) --- returns element i of point p1
    - Returns element (i mod p1.rank) if i < 0 or i >= p1.rank
  - p1.lt(p2), p1.le(p2), p1.gt(p2), p1.ge(p2)
    - Returns true iff p1 is lexicographically <, <=, >, or >= p2
    - Only defined when p1.rank and p1.rank are equal

Example: point

```
public class TutPoint {
    public static void main(String[] args) {
        point p1 = [1,2,3,4,5];
        point p2 = [1,2];
        point p3 = [2,1];
        System.out.println("p1 = " + p1 +
               " ; p1.rank = " + p1.rank +
               " ; p1[2] = " + p1[2]);
        System.out.println("p2 = " + p2 +
               " ; p3 = " + p3 + " ; p2.lt(p3) = " +
               p2.lt(p3));
    }
}
```

Console output:

```
p1 = [1,2,3,4,5] ; p1.rank = 5 ; p1[2] = 3
p2 = [1,2] ; p3 = [2,1] ; p2.lt(p3) = true
```

Rectangular Regions

A **rectangular region** is the set of points contained in a rectangular subspace

A region variable can hold values of different ranks e.g.,

- **region R; R = [0:10]; … R = [-100:100, -100:100]; … R = [0:-1]; …**

Operations

- **R.rank ::= # dimensions in region;**
- **R.size() ::= # points in region**
- **R.contains(P) ::= predicate if region R contains point P**
- **R.contains(S) ::= predicate if region R contains region S**
- **R.equal(S) ::= true if region R equals region S**
- **R.rank(i).low() ::= lower bound of i^{th} dimension of region R**
- **R.rank(i).high() ::= upper bound of i^{th} dimension of region R**
- **R.ordinal(P) ::= ordinal value of point P in region R**
- **R.coord(N) ::= point in region R with ordinal value = N**

Example: region

```
public class TutRegion {
    public static void main(String[] args) {
        region R1 = [1:10, -100:100];
        System.out.println("R1 = " + R1 + " ; R1.rank = " +
  R1.rank + " ; R1.size() = " + R1.size() + " ; R1.ordinal
  ([10,100]) = " + R1.ordinal([10,100]));
        region R2 = [1:10,90:100];
        System.out.println("R2 = " + R2 + " ; R1.contains(R2) =
  " + R1.contains(R2) + " ; R2.rank(1).low() = " + R2.rank
  (1).low() + " ; R2.coord(0) = " + R2.coord(0));
    }
}
```

Console output:

```
R1 = {1:10,-100:100} ; R1.rank = 2 ; R1.size() = 2010 ;
  R1.ordinal([10,100]) = 2009
R2 = {1:10,90:100} ; R1.contains(R2) = true ; R2.rank(1).low
  () = 90 ; R2.coord(0) = [1,90]
```

Summary of forall statement

```
forall (point [i1] : [lo1:hi1]) <body>
forall (point [i1,i2] : [lo1:hi1,lo2:hi2]) <body>
forall (point [i1,i2,i3] : [lo1:hi1,lo2:hi2,lo3:hi3]) <body>
. . .
```

- forall statement creates multiple async child tasks, one per iteration of the forall
  - all child tasks can execute <body> in parallel
  - child tasks are distinguished by index "points" ([i1], [i1,i2], …)
- forall statement completes and parent task proceeds to the following statement when all child tasks have completed (implicit finish)
- <body> can read local variables from parent (copy-in semantics like async)

forall examples: updates to a two-dimensional Java array

```
// Case 1: A[i][j]=F(A[i][j]) → loops i,j can run in parallel
forall (point[i,j] : [0:m-1,0:n-1]) A[i][j] = F(A[i][j]) ;

// Case 2: A[i][j]=F(A[i][j-1]) → only loop i can run in
   parallel
forall (point[i] : [1:m-1])
  for (point[j] : [1:n-1]) // Equivalent to "for (j=1;j<n;j++)"
     A[i][j] = F(A[i][j-1]) ;

// Case 3: A[i][j]=F(A[i-1][j]) → only loop j can run in
   parallel
for (point[i] : [1:m-1]) // Equivalent to "for (i=1;i<m;j++)"
  forall (point[j] : [1:n-1])
     A[i][j] = F(A[i-1][j]) ;
```

Pointwise for loop

- HJ extends Java's for loop to support sequential iteration over points in region R in canonical lexicographic order
  - for (point p : R) . . .
- Iteration space is define as in forall
  - Standard point operations can be used to extract individual index values from point p
    - for (point p : R)

      { int i = p.get(0); int j = p.get(1); . . . }
  - Or an "exploded" syntax can be used instead of explicitly declaring a point variable
    - for (point [i,j] : R) { . . . }
  - The exploded syntax declares the constituent variables (i, j, …) as local int variables in the scope of the for loop body

Example

```
public class TutFor {
    public static void main(String[] args) {
        region R = [0:1,0:2];
        System.out.print("Points in region " + R + " =");
        for ( point p : R ) System.out.print(" " + p);
        System.out.println();
        // Use exploded syntax instead
        System.out.print("(i,j) pairs in region " + R + " =");
        for ( point[i,j] : R )
            System.out.print("(" + i + "," + j + ")");
        System.out.println();
    } // main()
} //
```

Console output:

```
Points in region {0:1,0:2} = [0,0] [0,1] [0,2] [1,0] [1,1] [1,2]
(i,j) pairs in region {0:1,0:2} =(0,0)(0,1)(0,2)(1,0)(1,1)(1,2)
```