COMP 322: Fundamentals of Parallel Programming

Lecture 10: Forasync chunking, Parallel Prefix Sum algorithm

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments for Today’s Lecture

• Prof. Kathy Yelick, UC Berkeley, CS 194 Lecture, Fall 2007
Goals for Today’s Lecture

• Chunking of forasync loops (contd)
• Parallel prefix algorithm
Summary of forasync statement

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

... 

• forasync statement creates multiple async child tasks, one per iteration of the forasync
  — all child tasks can execute <body> in parallel
  — child tasks are distinguished by index “points” ([i1], [i1,i2], …)

• <body> can read local variables from parent (copy-in semantics like async)

• forasync needs a finish for termination, just like regular async tasks
  — Later, we will learn about replacing “finish forasync” by “forall”
Chunking of 1-D forasync loops

// BEFORE CHUNKING:

// Original forasync loop iterates over region R = [0:n-1]
forasync (point [i] : [0:n-1]) <body>

// AFTER PARTITIONING INTO Ci CHUNKS:
// Outer forasync-ii loop iterates over Ci chunks with
// point [ii] in region chunks([0:n-1],[Ci]).
// Inner for-i loop iterates over getChunk([0:n-1],[Ci],[ii])
forasync (point [ii] : [0:Ci-1])
  for (point [i] : getChunk([0:n-1],[Ci],[ii]))

forasync

<table>
<thead>
<tr>
<th>ii</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
getChunk() method for 1-D regions

1. /*
2. * @param r = region for original loop
3. * @param c = tuple containing # chunks per dimension
4. * @param p = current iteration for outer loop
5. * @return region of iterations for chunk p
6. */
7. static region getChunk(region r, point c, point p) {
8.  region retVal;
9.  // Set retVal = region for chunk p if region r is partitioned
10.  // according to number of chunks specified in c
11.  . . .
12.  return retVal
13.}
Implementing `getChunk()` method for 1-D regions

1. `static region getChunk(region r, point c, point p) {`
2.     // Assume that r, c, p all are 1D (rank = 1)
3.     int rLo = r.rank(0).low(); int rHi = r.rank(0).high();
4.     if (rLo > rHi) return [0:-1]; // Empty region
5.     int c0 = c.get(0); assert(c0>0); // numChunks must be > 0
6.     int ii = p.get(0); assert(0<=ii && ii<c0);
7.     // ii must be in the range [0:c0-1]
8.     int chunkSize = ceilDiv(rHi-rLo+1, c0);
9.     int myLo = rLo + ii*chunkSize;
10.    int myHi = Math.min(rHi, rLo + (ii+1)*chunkSize - 1);
11.    return [myLo:myHi];
12.}`
One-Dimensional Iterative Averaging Example

- Initialize a one-dimensional array of \((n+2)\) double's with boundary conditions, \(myVal[0] = 0\) and \(myVal[n+1] = 1\).

- In each iteration, each interior element \(myVal[i]\) in \(1..n\) is replaced by the average of its left and right neighbors.
  - Two separate arrays are used in each iteration, one for old values and the other for the new values.

- After a sufficient number of iterations, we expect each element of the array to converge to \(myVal[i] = i/(n+1)\)
  - In this case, \(myVal[i] = (myVal[i-1] + myVal[i+1])/2\), for all \(i\) in \(1..n\)

Illustration of an intermediate step for \(n = 8\) (source: Figure 6.19 in Lin-Snyder book)
HJ code for One-Dimensional Iterative Averaging using nested for-finish-forasync structure

1. for (point [iter]: [0:iterations-1]) {
2.   // Compute MyNew as function of input array MyVal
3.   finish forasync (point [j]: [1:n]) { // Create n tasks
4.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
5.   } // finish forasync
6.   temp=myVal; myVal=myNew; myNew=temp; // Swap myVal & myNew;
7.   // myNew becomes input array for next iteration
8. } // for

• How many tasks does this version create?
• This is an idealized version with no “chunking” of forasync iterations
Example: HJ code for One-Dimensional Iterative Averaging with chunked for-finish-forasync-for

1. for (point [iter] : [0:iterations-1]) {
2.   // Compute MyNew as function of input array MyVal
3.   int Cj = ...; // Set to desired number of chunks
4.   finish forasync (point [jj]:[0:Cj-1]) {
5.     for (point [j]:getChunk([1:n],[Cj],[jj]))
6.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.     } // finish forasync
8.   temp=myVal; myVal=myNew; myNew=temp; // Swap myVal & myNew;
9.   // myNew becomes input array for next iteration
10.} // for

• How many tasks does this chunked version create?
Goals for Today’s Lecture

- Chunking of forasync loops (contd)
- Parallel prefix algorithm
- Parallel quicksort algorithm
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

```java
// 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

```java
finish {
    for (int i=0 ; i < X.length ; i++ )
        // invoke computeSum() function from Lecture 7
        // (see ArraySum2.hj)
        async X[i] = computeSum(A, 0, i);
}
```

Observations:

- Critical path length, $CPL = O(\log n)$
- Total number of operations, $WORK = O(n^2)$

$\Rightarrow$ 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.


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

Input array, A: 3 1 2 0 4 1 1 3

```
15
15 6 6
4 4
3 2
1 5
2 4
0 1
6 9
15
```
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
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 $A[0...i]$

- **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.)