COMP 322: Fundamentals of Parallel Programming

Lecture 11: Multidimensional forasync loops, Chunking of parallel loops

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Worksheet #10 solution: Scheduling Program Q2 using Work-First & Help-First Schedulers

Complete work-first and help-first schedules for the program shown below (using step times from the computation graph)

1. // Program Q2
2. A;
3. finish {
4.   async { C; E; }
5.   async F;
6.   async { B; D; }
7. }

Work-First Schedule

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
Worksheet #10 solution: Scheduling Program Q2 using Work-First & Help-First Schedulers (contd)

1. // Program Q2
2. A;
3. finish {
4.   async { C; E; }
5.   async F;
6.   async { B; D; }
7. }

Help-First Schedule

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>12</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>13</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Outline of Today’s Lecture

- Multidimensional Forasync loops
- Chunking of parallel loops

Acknowledgments

- COMP 322 Module 1 handout, Section 8.1, Section 9.4.
HJ’s pointwise for & forasync statements

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

```java
finish {
    for (int I = 0 ; I < N ; I++)
        for (int J = 0 ; J < N ; J++)
            async
                for (int K = 0 ; K < N ; K++)
}
```

by

```java
finish forasync (point [I,J] : [0:N-1,0:N-1])
    for (point[K] : [0:N-1])
```
Sequential Algorithm for Matrix Multiplication

\[ c[i,j] = \sum_{0 \leq k < n} a[i,k] \times b[k,j] \]

1. // Sequential version
2. for (int i = 0 ; i < n ; i++)
3. for (int j = 0 ; j < n ; j++)
4. c[i][j] = 0;
5. for (int i = 0 ; i < n ; i++)
6. for (int j = 0 ; j < n ; j++)
7. for (int k = 0 ; k < n ; k++)
8. c[i][j] += a[i][k] * b[k][j];
9. // Print first element of output matrix
10. System.out.println(c[0][0]);
Parallelizing the loops in Matrix Multiplication example using finish & async (Listing 27)

\[ c[i,j] = \sum_{0 \leq k < n} a[i,k] \times b[k,j] \]

1. // Parallel version using finish & async
2. finish for (int i = 0 ; i < n ; i++)
3. for (int j = 0 ; j < n ; j++)
4. async c[i][j] = 0;
5. finish for (int i = 0 ; i < n ; i++)
6. for (int j = 0 ; j < n ; j++)
7. async for (int k = 0 ; k < n ; k++)
8. c[i][j] += a[i][k] * b[k][j];
9. // Print first element of output matrix
10. System.out.println(c[0][0]);
Observations

• **finish** and **async** are general constructs, and are not specific to loops
  —Not easy to discern from a quick glance which loops are sequential vs. parallel

• Loops in sequential version of matrix multiplication are “perfectly nested”
  —e.g., no intervening statement between “for(i = ...)” and “for(j = ...)”

• The ordering of loops nested between **finish** and **async** is arbitrary
  —They are parallel loops and their iterations can be executed in any order
Parallelizing the loops in Matrix Multiplication example using finish, forasync & for (Listing 28)

\[ c[i,j] = \sum_{0 \leq k < n} a[i,k] \times b[k,j] \]

1. // Parallel version using finish & forasync
2. finish forasync(point[i,j] : [0:n-1,0:n-1])
3. c[i][j] = 0;
4. finish forasync(point[i,j] : [0:n-1,0:n-1]) {
5. for(point[k] : [0:n-1])
6. c[i][j] += a[i][k] * b[k][j];
7. }
8. // Print first element of output matrix
9. System.out.println(c[0][0]);
Observations

• The combination of perfectly nested for–for–async constructs is replaced by a single keyword, `forasync`

• Multiple loops can be collapsed into a single `forasync` with a multi-dimensional iteration space (can be 1D, 2D, 3D, ...)

• The iteration variable for a `forasync` is a point (integer tuple) such as `[i,j]`.

• The loop bounds can be specified as a rectangular region (product of 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 forasync
Summary of HJ’s 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”

• In addition to its convenient syntax, parallel loop constructs are easier to manage with “chunking”, compared to for-for-async structures
**hj.lang.point, an index type for multi-dimensional loops**

- A point is an element of an n-dimensional Cartesian space \((n \geq 1)\) with integer-valued coordinates e.g., \([5], [1, 2], \ldots\)
  - Dimensions of a point are numbered from 0 to \(n-1\)
  - \(n\) is also referred to as the **rank** (size) 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 point-valued expression \(p_1\)
  - \(p_1.rank\) --- returns rank of point \(p_1\)
  - \(p_1.get(i)\) --- returns element \(i\) of point \(p_1\)
    - Returns element \((i \mod p_1.rank)\) if \(i < 0\) or \(i \geq p_1.rank\)
  - \(p_1.lt(p_2), p_1.le(p_2), p_1.gt(p_2), p_1.ge(p_2)\)
    - Returns true iff \(p_1\) is lexicographically <, <=, >, or >= \(p_2\)
    - Only defined when \(p_1.rank\) and \(p_1.rank\) are equal

- You can think of a point as an int array with additional operator support in the HJ language
Example

```java
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.get(2) = " + p1.get(2));
        System.out.println("p2 = " + p2 + " ; p3 = " + p3 + " ; p2.lt(p3) = " + p2.lt(p3));
    } // main()
} // TutPoint
```

Console output:

```
p1 = [1,2,3,4,5] ; p1.rank = 5 ; p1.get(2) = 3
p2 = [1,2] ; p3 = [2,1] ; p2.lt(p3) = true
```
hj.lang.region, a rectangular iteration space for multi-dimensional loops

A 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
- \( \text{R.rank} \) ::= \# dimensions in region;
- \( \text{R.size()} \) ::= \# points in region
- \( \text{R.contains(P)} \) ::= predicate if region R contains point P
- \( \text{R.contains(S)} \) ::= predicate if region R contains region S
- \( \text{R.equal(S)} \) ::= true if region R equals region S
- \( \text{R.rank(i)} \) ::= projection of region R on dimension i (a one-dimensional region)
- \( \text{R.rank(i).low()} \) ::= lower bound of \( i^{\text{th}} \) dimension of region R
- \( \text{R.rank(i).high()} \) ::= upper bound of \( i^{\text{th}} \) dimension of region R
- \( \text{R.ordinal(P)} \) ::= ordinal value of point P in region R
- \( \text{R.coord(N)} \) ::= point in region R with ordinal value = N
Pointwise sequential 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 ) . . .

- 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 is commonly 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
forasync examples: updates to a two-dimensional Java array

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

// Case 2: only loop i can run in parallel
forasync (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: only loop j can run in parallel
for (point[i] : [1:m-1]) // Equivalent to “for (i=1;i<m;j++)”
    finish forasync (point[j] : [1:n-1])
      A[i][j] = F(A[i-1][j]) ;
One-Dimensional Iterative Averaging Example

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

- In each iteration, each interior element \(\text{myVal}[i]\) 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 \(\text{myVal}[i] = i/(n+1)\)
  - In this case, \(\text{myVal}[i] = (\text{myVal}[i-1]+\text{myVal}[i+1])/2\), 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:m-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 does this algorithm work? Let's try Worksheet #11!
Outline of Today’s Lecture

• Multidimensional Forasync loops

• Chunking of parallel loops

Acknowledgments

• COMP 322 Module 1 handout, Section 8.1, Section 9.4.
What about overheads?

- We learned in Lecture 10 that it is inefficient to create async tasks that do little work

- In the Iterative Averaging example, each async task (forasync iteration) performs only a few operations
  
  — \( \text{myNew}[j] = (\text{myVal}[j-1] + \text{myVal}[j+1])/2.0; \)

- The “seq” clause doesn’t help in this case because it will just sequentialize the entire forasync loop

- An alternate approach is “loop chunking”
  
  — e.g., replace
    
    \[
    \text{forasync(point}[i] : [0:99]) \ \text{BODY}(i); \ // \ 100 \ \text{tasks}
    \]

  — by
    
    \[
    \text{forasync(point}[ii] : [0:3]) \ // \ 4 \ \text{tasks}
    \]
    
    // Each task executes a “chunk” of 25 iterations
    
    \[
    \text{for (point}[i] : [25*ii:25*(ii+1)-1]) \ \text{BODY}(i);
    \]
Chunking a 1-dimensional forasync loop  
(General approach)

- Assume that the forasync loop originally iterates over region \( r \)
  \[
  \text{forasync(} \text{point}[i] : r) \\
  \quad \text{BODY}(i); // \text{No. of tasks} = r.size()
  \]

- Assume that we have a parameter, \( nc \), for the desired number of chunks (tasks)
  —A good choice is \( nc = \text{Runtime.getNumOfWorkers()} \), as in Listing 31

- Assume that we have a helper method, \( \text{getChunk}(r, nc, ii) \) that returns the iteration range for chunk \# \( ii \) as an HJ region
  —e.g., \( \text{getChunk([0:99], 4, 0) = [0:24] and getChunk([0:99], 4, 3) = [75:99]} \)
  —No requirement for \( nc \) to evenly divide \( r.size() \)

- The original forasync above can then be rewritten as
  \[
  \text{forasync(} \text{point}[ii] : [0:nc-1]) \\
  \quad \text{for(point}[i] : \text{getChunk}(r, nc, ii)) \\
  \quad \text{BODY}(i); // \text{No. of tasks} = nc
  \]
Implementation of getChunk()
helper method in HJ

1. static region getChunk(region r, int nc, int ii) {
2.     // Assume that r is a 1D region
3.     int rLo = r.rank(0).low(); int rHi = r.rank(0).high();
4.     if (rLo > rHi) return [0:-1]; // Empty region
5.     assert(nc > 0); // nc must be > 0
6.     assert(0 <= ii && ii < nc); // ii must be in [0:nc-1]
7.     int chunkSize = ceilDiv(rHi-rLo+1, nc);
8.     int myLo = rLo + ii*chunkSize;
9.     int myHi = Math.min(rHi, rLo + (ii+1)*chunkSize - 1);
10.    region retVal = [myLo:myHi];
11.    return retVal;
12. }

14. static int ceilDiv(int n, int d) {
15.     assert(n>=0 && d>0); return (n+d-1)/ d;
16. }
Example: HJ code for One-Dimensional Iterative Averaging with chunked for-finish-forasync-for structure

1. int nc = Runtime.getNumOfWorkers();
2. for (point [iter] : [0:m-1]) {
3.     // Compute MyNew as function of input array MyVal
4.     finish forasync (point [jj] : [0:nc-1]) {
5.         for(point [j] : getChunk([1:n],nc,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
**Chunking a k-dimensional forasync loop**

(General approach)

- Assume that the forasync loop originally iterates over region r
  
  ```plaintext
  forasync(point p : r)
  BODY(p); // No. of tasks = r.size()
  ```

- Assume that we have an int array, nc = {nc0, nc1, ...}, for the desired number of chunks in each dimension
  
  — A good choice is to choose these values such that the product of \( nc[0]*nc[1]*... = \text{Runtime.getNumOfWorkers()} \)

- Assume that we have a helper method, getChunk(r, nc, pp) that returns the iteration range for chunk pp as an HJ region
  
  — e.g., `getChunk([0:99,0:99], {2,2}, [0,0]) = [0:49,0:49]`

- The original forasync above can then be rewritten as
  
  ```plaintext
  forasync(point pp : [0:nc[0]-1,0:nc[1]-1,...])
  for(point p : getChunk(r,nc,pp))
  BODY(p);
  ```
Worksheet #11: One-dimensional Iterative Averaging Example

Name 1: ___________________        Name 2: ___________________

1) Assuming n=9 and the input array below, perform one iteration of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array. Recall that the computation is “myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;”

<table>
<thead>
<tr>
<th>index, j</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>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>myVal</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0</td>
<td>0.4</td>
<td>0</td>
<td>0.6</td>
<td>0</td>
<td>0.8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>myNew</td>
<td>0</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2) Will the contents of myVal[] and myNew[] change in further iterations, after myNew above in 1) becomes myVal[] in the next iteration?