COMP 322: Fundamentals of Parallel Programming

Lecture 11: Loop-Level Parallelism, Parallel Matrix Multiplication

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

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

Worksheet #10 solution: Associativity and Commutativity

A Finish Accumulator (FA) can be used for any *associative and commutative* binary function.
Parallel Prefix (PP) algorithm can be used for any *associative* binary function (the same applies for parallel reductions in ArraySum1 and ArraySum2).

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, is associative and commutative
   ⇒ both FA and PP can be used

2) g(x,y) = (x+y)/2, for integers x, y, is commutative but not associative
   ⇒ neither FA nor PP can be used

3) h(s1,s2) = concat(s1, s2) for strings s1, s2 e.g., h("ab","cd") = "abcd" is associative but not commutative
   ⇒ PP can be used, but not FA

Use of asyncSeq API in HJlib (Quicksort example --- correction)

```
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 of Today's Lecture

- Multidimensional parallel loops

- Chunking of parallel loops

Sequential Algorithm for Matrix Multiplication

$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$

```
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

$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$

```
1.  // Parallel version using finish & async
2.  finish(() -> {
3.    for (int i = 0 ; i < n ; i++)
4.      for (int j = 0 ; j < n ; j++)
5.        async(() -> {c[i][j] = 0; });
6.  });
7.  finish(() -> {
8.    for (int i = 0 ; i < n ; i++)
9.      for (int j = 0 ; j < n ; j++)
10.       async(() -> {
11.         for (int k = 0 ; k < n ; k++)
12.           c[i][j] += a[i][k] * b[k][j];
13.       });
14. });
15. // Print first element of output matrix
16. System.out.println(c[0][0])
```


Observations on finish-for-async version

- 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 forall

$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$

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


forall API's in HJlib

(http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html)

- `static void forall(edu.rice.hj.api.HjRegion.HjRegion1D hjRegion, edu.rice.hj.api.HjProcedureInt1D body)`
- `static void forall(edu.rice.hj.api.HjRegion.HjRegion2D hjRegion, edu.rice.hj.api.HjProcedureInt2D body)`
- `static void forall(edu.rice.hj.api.HjRegion.HjRegion3D hjRegion, edu.rice.hj.api.HjProcedureInt3D body)`
- `static void forall(int s0, int e0, edu.rice.hj.api.HjProcedure<java.lang.Integer> body)`
- `static void forall(int s0, int e0, int s1, int e1, edu.rice.hj.api.HjProcedureInt2D body)`
- `static <T> void forall(java.lang.Iterable<T> iterable, edu.rice.hj.api.HjProcedure<T> body)`
- **NOTE: all forall API's include an implicit finish. forasync is like forall, but without the finish.**

Observations on forall version

- The combination of perfectly nested for–for–async constructs is replaced by a single API, forall
- Multiple loops can be collapsed into a single forall with a multi-dimensional iteration space (can be 1D, 2D, 3D, ...)
- The iteration variable for a forall is a HjPoint (integer tuple), e.g., (i,j)
- The loop bounds can be specified as a rectangular HjRegion (product of dimension ranges), e.g., (0:n−1) x (0:n−1)
- HJlib also provides a sequential for API that can also be used to iterate sequentially over a rectangular region
  - —Simplifies conversion between for and forall

HjPoint, an index type for multi-dimensional sequential and parallel loops

- A point is an element of an n-dimensional Cartesian space (n>=1) with integer-valued coordinates
  - 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.,
  - HjPoint p; p = newPoint(1); … p = newPoint(2, 3); …
- The following operations are defined on 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.compareTo(p2)
    - Returns -1, 0, 1 iff p1 is lexicographically <, =, or > p2
    - Only defined when p1.rank() and p2.rank() are equal
- You can think of an HjPoint as an immutable int array

Example

```
public class TutPoint {
  public static void main(String[] args) {
    HjPoint p1 = newPoint(1, 2, 3, 4, 5);
    HjPoint p2 = newPoint(1, 2);
    HjPoint p3 = newPoint(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.compareTo(p3) = " + p2.compareTo(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.compareTo(p3) = -1
```


HjRegion, 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.,

- HjRegion R;
  R = newRectangularRegion1D(0, 10); …
  R = newRectangularRegion2D(-100, 100, -100, 100); …
  R = newRectangularRegion1D(0, -1); …

Operations

- R.rank() ::= # dimensions in region;
- R.numElements() ::= # points in region
- R.lowerLimit(i) ::= lower limit of dimension i
- R.upperLimit(i) ::= upper limit of dimension i
- R.toLinearIndex(i1, i2, …, iN) ::= ordinal value of point P in region R
- R.toRegionIndex(N) ::= point in region R with ordinal value = N

forall examples: updates to a two-dimensional Java array

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

// Case 2: only loop i can run in parallel
forall(1, m-1, (i) -> {
  forseq(1, n-1, (j) -> { // 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
forseq(1, m-1, (i) -> { // Equivalent to "for (i=1;i<m;j++)"
  forall(1, n-1, (j) -> {
     A[i][j] = F(A[i-1][j]) ;
}); });
```


Outline of Today's Lecture

- Multidimensional Forasync loops

- Chunking of parallel loops

What about overheads?

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

```
forall(0, 99, (i) -> BODY(i)); // 100 tasks
```

  - by

```
forall(0, 3, (ii) -> { // 4 tasks
  // Each task executes a "chunk" of 25 iterations
  forseq(25*ii, 25*(ii+1)-1], (i) -> BODY(i));
});
```


forallChunked APIs

- forallChunked(int s0, int e0, int chunkSize, edu.rice.hj.api.HjProcedure<java.lang.Integer> body)
- **Like** forall(int s0, int e0, edu.rice.hj.api.HjProcedure<java.lang.Integer> body) **but forallChunked includes chunkSize as the third parameter**
  - **e.g., replace**

```
forall(0, 99, (i) -> BODY(i)); // 100 tasks
```

  - **by**

```
forall(0, 99, 100/4, (i) -> BODY(i));
```


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

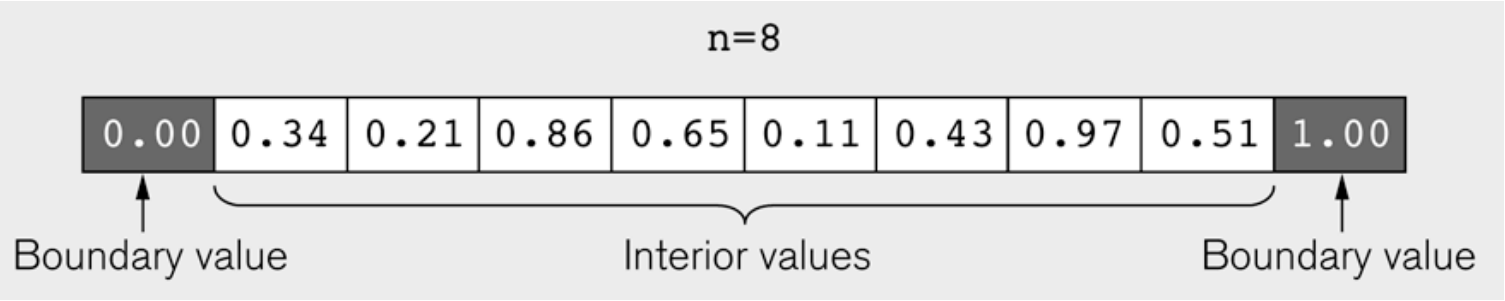

Illustration of an intermediate step for n = 8 (source: Figure 6.19 in Lin-Snyder book)

HJ code for One-Dimensional Iterative Averaging using nested forseq-forall structure

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


Example: HJ code for One-Dimensional Iterative Averaging with forseq-forall structure w/ chunking

```
1. int nc = numWorkerThreads();
2. forseq(0, m-1, (iter) -> {
3.   // Compute MyNew as function of input array MyVal
4.     forallChunked(1, n, n/nc, (j) -> { // Create n tasks
5.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
6.     }); // forall
7.   temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
8.   // myNew becomes input array for next iteration
9. }); // for
```


Worksheet #11: One-dimensional Iterative Averaging Example

Name: _____ Netid: _____

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;"

index, j	0	1	2	3	4	5	6	7	8	9	10
myVal	0	0	0.2	0	0.4	0	0.6	0	0.8	0	1
myNew	0		0.2		0.4		0.6		0.8		1

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