COMP 322: Fundamentals of Parallel Programming

Lecture 10: Loop-Level Parallelism, Parallel Matrix Multiplication

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Announcements & Reminders

- Lab #2 is due by Tuesday, Feb 23rd at noon
- Quiz for Unit 2 (topics 2.1 - 2.8) is due by 11:59pm on Friday, Feb. 26th
  - Auto submitted quizzes can be retaken
- HW #2 is due Wednesday, Mar. 3rd at 11:59pm
- Midterm Exam on Thursday, Mar. 11th at 7pm

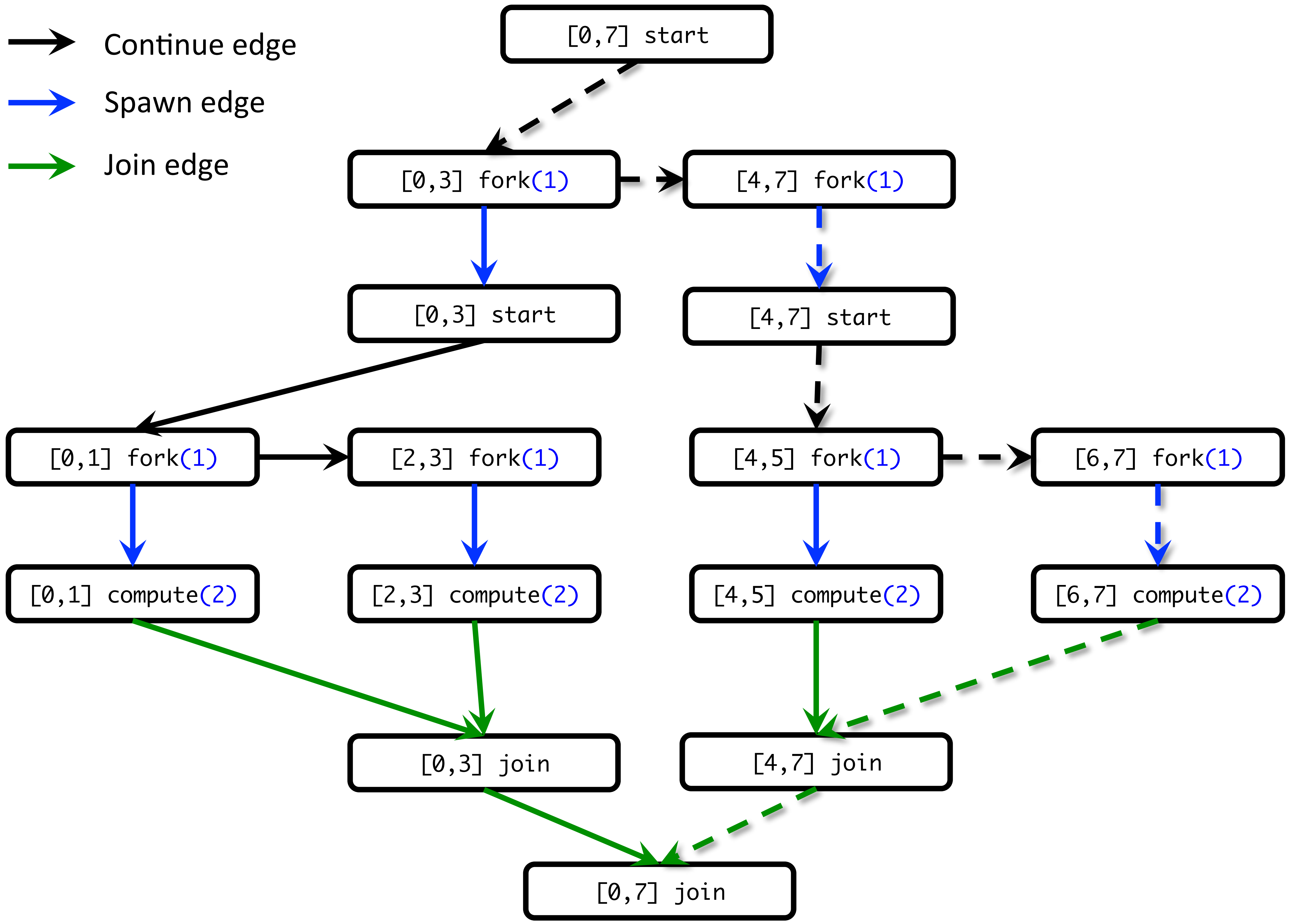
compute()

```
protected void compute() {
    if (hi - lo < THRESHOLD) {
      for (int i = lo; i <= hi; ++i)
        array[i] = array[i] / (i + 1);
    } else {
      int mid = (lo + hi) >>> 1;
      invokeAll(new DivideTask(array, lo, mid),
               new DivideTask(array, mid+1, hi));
    }
   }
```

Worksheet #9 solution: RecursiveAction Computation Graph

1) Consider the compute method on slide 9. Let us suppose we supply it with an 8 element array with values [0,1,2,3,4,5,6,7] and THRESHOLD value of 2. Draw a computation graph corresponding to a call to `compute` with the appropriate fork and join edges.

2) Define each direct (sequential) computation as 2 units of work and each recursive call as one unit of work. What is the total work? What is the critical path length?



TOTAL WORK = 14, CPL = 4 or 6 (depends on how recursive call is counted)

NOTE: each call to compute() takes 2 units because THRESHOLD = 2

Sequential Algorithm for Matrix Multiplication

```
// Sequential version
 for (int i = 0 ; i < n ; i++)
   for (int j = 0 ; j < n ; j++)
     c[i][j] = 0;
 for (int i = 0 ; i < n ; i++)
   for (int j = 0 ; j < n ; j++)
     for (int k = 0 ; k < n ; k++)
       c[i][j] += a[i][k] * b[k][j];
 // Print first element of output matrix
println(c[0][0]);
```

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

Parallelizing loops in Matrix Multiplication using finish & async

```
// Parallel version using finish & async
finish(() -> {
  for (int ii = 0 ; ii < n ; ii++)
    for (int jj = 0 ; jj < n ; jj++) {
      final int i = ii; final int j = jj;
      async(() -> {c[i][j] = 0; });
    }
});
finish(() -> {
  for (int ii = 0 ; ii < n ; ii++)
    for (int jj = 0 ; jj < n ; jj++){
      final int i = ii; final int j = jj;
      async(() -> {
        for (int k = 0 ; k < n ; k++)
          c[i][j] += a[i][k] * b[k][j];
      });
    }
});
// Print first element of output matrix
println(c[0][0])
```

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

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 loops in Matrix Multiplication example using forall

```
// Parallel version using forall
forall(0, n-1, 0, n-1, (i, j) -> {
    c[i][j] = 0;
 });
forall(0, n-1, 0, n-1, (i, j) -> {
   forseq(0, n-1, (k) -> {
     c[i][j] += a[i][k] * b[k][j];
   });
});
// Print first element of output matrix
println(c[0][0]);
```

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

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)

forall API's in HJlib (http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html)

- 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`. Also e0 is the "end" value, not 1 + end value.

Observations on forall version

- The combination of perfectly nested finish-for–for–async constructs is replaced by a single API, `forall`
  - `forall` includes an implicit `finish`
- 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) is a 2-dimensional point
- 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 `forseq` API that can also be used to iterate sequentially over a rectangular region
  - Simplifies conversion between `forseq` and `forall`

forall examples: updates to two-dimensional Java array

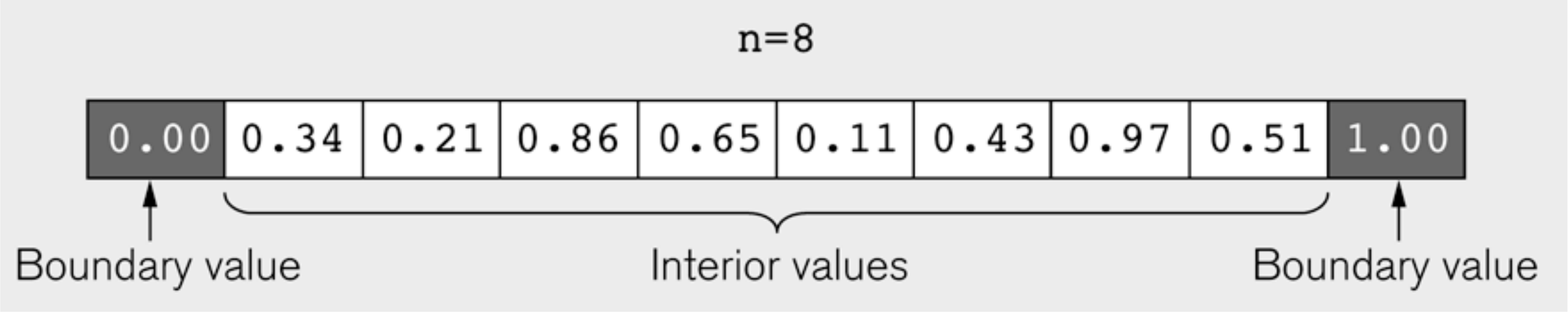
```
// Case 1: loops i,j can run in parallel
forall(0, m-1, 0, n-1, (i, j) -> { A[i][j] = F(A[i][j]);});

// Case 2: only loop i can run in parallel
forall(0, m-1, (i) -> {
  forseq(0, n-1, (j) -> { // Equivalent to “for (j=0;j<n;j++)”
     A[i][j] = F(A[i][j-1]) ;
}); });

// Case 3: only loop j can run in parallel
forseq(0, m-1, (i) -> { // Equivalent to “for (i=0;i<m;j++)”
  forall(0, n-1, (j) -> {
     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, 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] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n



0.00	0.34	0.21	0.86	0.65	0.11	0.43	0.97	0.51	1.00

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

Sequential code for One-Dimensional Iterative Averaging

```
1.// Intialize m, n, myVal, newVal
2.m = … ; n = … ;
3.float[] myVal = new float[n+2];
4.float[] myNew = new float[n+2];
5.forseq(0, m-1, (iter) -> {
6.  // Compute MyNew as function of input array MyVal
7.    forseq(1, n, (j) -> { // Create n tasks
8.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.    }); // forseq
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```

QUESTION: can either forseq() loop execute in parallel?

Worksheet #10: One-dimensional Iterative Averaging Example

Assuming n=9 and the input array below, perform a “half-iteration” of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array (different from the real algorithm). 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?

3) Write the formula for the final value of myNew[i] as a function of i and n. In general, this is the value that we will get if m (= #iterations in sequential for-iter loop) is large enough.