CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar

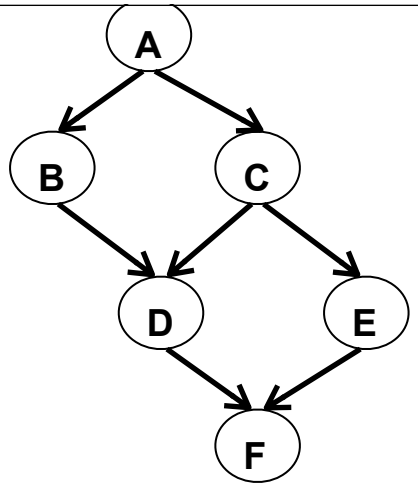
Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

Recap of Lecture 3

- Futures --- Tasks with Return Values
- Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)
- Finish Accumulators

Worksheet #4: Computation Graphs for Async-Finish and Future Constructs



1) Can you write an HJ program with <u>async-finish</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? No

2) Can you write an HJ program with <u>future async-get</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

Yes, see HJ code on the right

```
final future<void> A = async<void>
                { . . . };
final future<void> B = async<void>
                { A.get(); . . . };
final future<void> C = async<void>
                { A.get(); . . . };
final future<void> D = async<void>
                { B.get(); C.get(); . . . };
final future<void> E = async<void>
                { C.get(); . . . };
final future<void> F = async<void>
                { D.get(); E.get(); . . . };
```

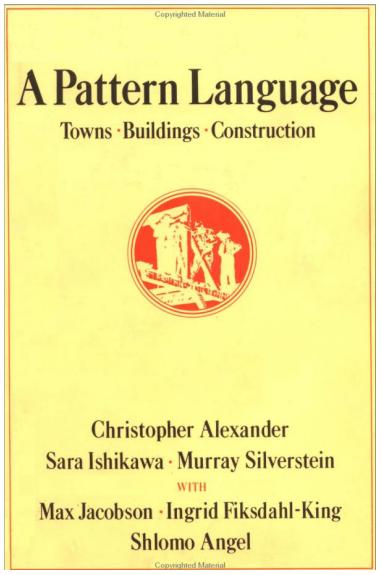
Acknowledgments for Today's Lecture

- "A Pattern language for Parallel Programming"
  - —Presentation by Beverly Sanders, U. Florida
  - —http://www.cise.ufl.edu/research/ParallelPatterns/sasplas.ppt
- "Parallel Programming with Microsoft .NET"
  - —Book published by Microsoft; free download available at http://parallelpatterns.codeplex.com
- "Introduction to Concurrency in Programming Languages", Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen
  - —Book published by CRC press; slides available from book web site, http://www.parlang.com/
- "Parallel Programming Patterns", ME964 lecture, Oct 2008, U. Wisconsin
  - —http://sbel.wisc.edu/Courses/ME964/2008/.../me964Oct16.ppt

Outline

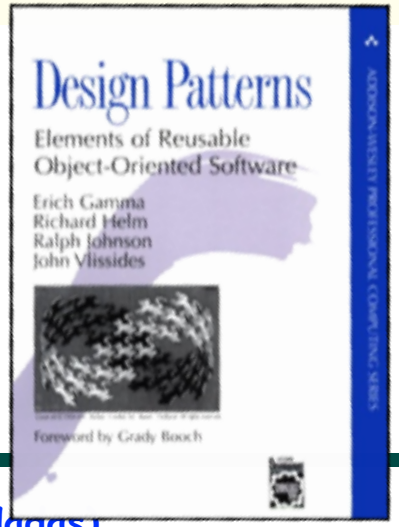
- <u>Design Patterns for Parallel Programming</u>
- Forasync and Forall loops

Design Patterns = formal discipline of design



- **Christopher Alexander's approach to (civil) architecture:**
  - **A design pattern "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."** *Page x, A Pattern Language,* **Christopher Alexander, 1977**
- **A pattern language is an organized way of tackling an architectural problem using patterns**

- The gang of 4 used patterns to bring order to the chaos of object oriented design.
- The book "overnight" turned object oriented design from "an art" to a systematic design discipline.



GOF Pattern Example

- Behavioral Pattern: Visitor
  - Separate the structure of an object collection from the operations performed on that collection.
  - Example: Abstract syntax tree in a compiler
    - Multiple node types (declaration, command, expression, etc.)
    - Action during traversal depends on both type of node and compiler pass (type checking, code generation)
    - Can add new functionality by implementing new visitor without modifying AST code.

Parallel Programming

- Can a pattern language/taxonomy providing guidance for the entire development process make parallel programming easier?
  - Need to identify basic patterns, along with refinements (usually for efficiency)
- By relating HJ constructs to parallel programming patterns, you can apply HJ concepts to any parallel programming model you encounter in the future

Algorithm Patterns vs. Supporting Patterns

- Algorithm Patterns
  - Selection of task and data decompositions to solve a given problem in parallel
    - Task decomposition = identification of parallel steps
    - Data decomposition = partitioning of data into task-local vs. shared storage classes (with ownership specified for local data)
  - Examples: Parallel Loops, Parallel Tasks, Reductions, Dataflow, Pipeline
- Supporting Patterns (to be discussed later in course)
  - Selection of execution model for a given algorithm pattern
    - Execution model specifies scheduling and synchronization of tasks
    - May be implemented using code and/or configuration parameters
  - Examples of supporting patterns for task scheduling: Master-worker, Single Instruction Multiple Data (SIMD), Single Program Multiple Data (SPMD)
  - Examples of supporting patterns for mutual exclusion: Isolated, Object-based isolation, Atomic variables, Concurrent collections, Actors
- Boundary between Algorithm & Supporting Patterns can sometimes be blurred

Selecting the Right Pattern
(adapted from page 9, Parallel Programming w/ Microsoft .Net)

Application characteristics	Algorithmic pattern	Relevant HJ constructs
Sequential loop with independent iterations	1) Parallel Loop	forall, forasync (to be covered later)
Independent operations with well-defined control flow	2) Parallel Task	async, finish
Aggregating data from independent tasks/iterations	3) Parallel Aggregation (reductions)	finish accumulators
Ordering of steps based on data flow constraints	4) Futures	futures, data-driven tasks
Divide-and-conquer algorithms with recursive data structures	5) Dynamic Task Parallelism	async, finish
Repetitive operations on data streams	6) Pipelines	streaming phasers (to be covered later)

2) Example of Parallel Task Pattern using HJ async and finish statements

```
1.  // Start of Task T0 (main program)
2.  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3.  finish {
4.    async { // Task T1 computes sum of upper half of array
5.      for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
6.    }
7.    // Continue in T0 and compute sum of lower half of array
8.    for(int i=0; i < X.length/2; i++) sum1 += X[i];
9.  } // finish
10. // Task T0 waits for Task T1 (join)
11. return sum1 + sum2;
```

5) Example of Dynamic Task Pattern using HJ async and finish constructs

```
finish nqueens_kernel(new int[0], 0)
// Array [a] contains array of queen positions found thus far
void nqueens_kernel(int [] a, int depth) {
    if (size == depth) {
      . . . ; // Solution found
      return;
    }
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) {
      async { // NOTE: creating async's at large depth can be expensive
        /* allocate a temporary array and copy a[] into it */
        int [] b = new int [depth+1];
        System.arraycopy(a, 0, b, 0, depth);
        b[depth] = i;
        if (ok(depth+1, b))
          nqueens_kernel(b, depth+1);
      }
    }
  }
```

seq clause in HJ async statement (Section 8.5, Module 1 handout)

```
async seq(cond) <stmt> ≡ if (cond) <stmt> else async <stmt>
```

- seq clause specifies condition under which async should be executed sequentially

```
void nqueens_kernel(int [] a, int depth) {
    if (size == depth) {
      . . . ; // Solution found
      return;
    }
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) {
      async seq(depth >= cutoff_value) {
        /* allocate a temporary array and copy a[] into it */
        int [] b = new i[depth+1];
        System.arraycopy(a,
        b[depth] = i;
        if (ok(depth+1, b))
          nqueens_kernel(b,
      }
    }
  }
```

The seq clause looks like fun! Let's try out another example in worksheet #5. Please do this worksheet in pairs!

3) Example of Parallel Aggregation Pattern using HJ finish accumulators

```
static accumulator a;
. . .
a = accumulator.factory.accumulator(SUM, int.class);
finish(a) nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + a.get().intValue())
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) a.put(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

4) Example of Future pattern using HJ future tasks (Lecture 3)

```
1.public MarketRecommendation DoAnalysisParallelDDT() {
2.  future<StockDataCollection> nyseData = async<StockDataCollection> {return LoadNyseData();};
3.  future<StockDataCollection> nasdaqData = async<StockDataCollection> {return LoadNasdaqData();};
4.  future<StockDataCollection> mergedMarketData =
      async<StockDataCollection> {return MergeMarketData(new[]{nyseData.get(), nasdaqData.get()});};
5.  future<StockDataCollection> normalizedMarketData =
      async<StockDataCollection> {return NormalizeData(mergedMarketData.get());};
6.  future<StockDataCollection> fedHistoricalData = async<StockDataCollection> {return LoadFedHistoricalData();};
7.  future<StockDataCollection> normalizedHistoricalData =
      async<StockDataCollection> {return NormalizeData(fedHistoricalData.get());};
8.  future<StockAnalysisCollection> analyzedStockData =
      async<StockAnalysisCollection> {return AnalyzeData(normalizedMarketData.get());};
9.  future<MarketModel> modeledMarketData = async<MarketModel> {return RunModel(analyzedStockData.get());};
10. future<StockAnalysisCollection> analyzedHistoricalData =
      async<StockAnalysisCollection> {return AnalyzeData(normalizedHistoricalData.get());};
11. future<MarketModel> modeledHistoricalData = async<MarketModel>{return RunModel(analyzedHistoricalData.get());};
12. MarketRecommendation recommendation =
      CompareModels(new[] {modeledMarketData.get(), modeledHistoricalData.get()});
13. return recommendation;
14.}
```

Refinement of Future Pattern using Data-Driven Tasks for Efficiency (Lecture 3)

```
1.public MarketRecommendation DoAnalysisParallelDDT() {
2.  async nyseData.put(LoadNyseData());
3.  async nasdaqData.put(LoadNasdaqData());
4.  async await(nyseData, nasdaqData)
      mergedMarketData.put(MergeMarketData(new[]{nyseData.get(), nasdaqData.get()}));
5.  async await(mergedMarketData) normalizedMarketData.put(NormalizeData(mergedMarketData.get()));
6.  async fedHistoricalData.put(LoadFedHistoricalData());
7.  async await(fedHistoricalData) normalizedHistoricalData.put(NormalizeData(fedHistoricalData.get()));
8.  async await(normalizedMarketData) analyzedStockData.put(AnalyzeData(normalizedMarketData.get()));
9.  async await(analyzedStockData) modeledMarketData.put(RunModel(analyzedStockData.get()));
10. async await(normalizedHistoricalData) analyzedHistoricalData.put(AnalyzeData(normalizedHistoricalData.get()));
11. async await(analyzedHistoricalData) modeledHistoricalData.put(RunModel(analyzedHistoricalData.get()));
12. MarketRecommendation recommendation =
      CompareModels(new[] {modeledMarketData.get(), modeledHistoricalData.get()});
13. return recommendation;
14.}
```

Outline

- Design Patterns for Parallel Programming
- <u>Forasync and Forall loops</u>

HJ's pointwise for & forasync statements

Goal: capture common for-async pattern in a single construct for multidimensional loops 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

```
finish forasync (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 for-async is replaced by a single keyword, forasync
- Multiple loops can be collapsed into a single forasync, with a multi-dimensional iteration space.
- Iteration variable for a forasync 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 forasync

hj.lang.point, an index type for multi-dimensional loops

- A point is an element of an n-dimensional Cartesian space (n>=1) with integer-valued coordinates e.g., [5], [1, 2], …
  - Dimensions of a point are numbered from 0 to n-1
  - n is also referred to as the <u>rank</u> 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 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

```
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

- 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) ::= projection of region R on dimension i (a one-dimensional region)
- R.rank(i).low() ::= lower bound of ith dimension of region R
- R.rank(i).high() ::= upper bound of ith 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

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"

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

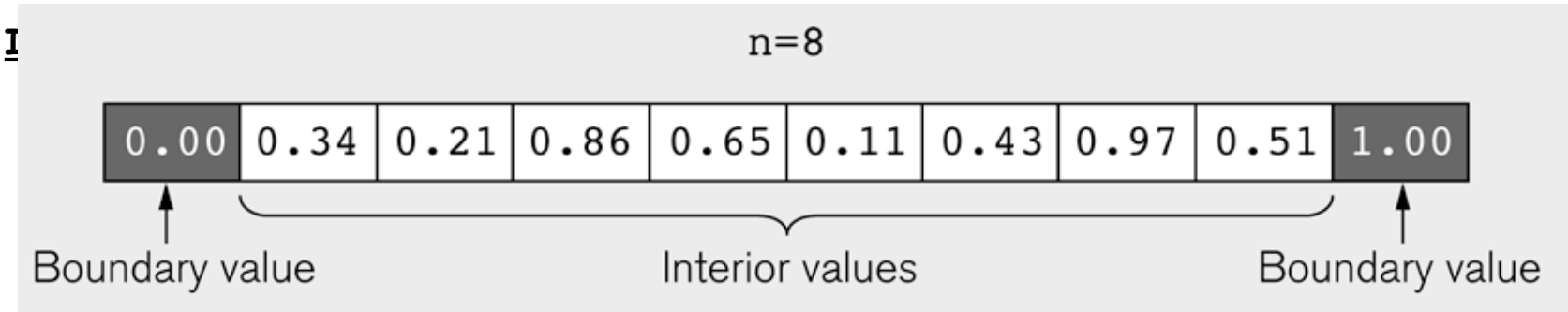
```
// Case 1: loops i,j can run in parallel
forasync (point[i,j] : [0:m-1,0:n-1]) A[i][j] = F(A[i][j]) ;


// 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, 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



HJ code for One-Dimensional Iterative Averaging using nested for-finish-forasync structure

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

- How many tasks does this version create?
- This is an idealized version with no "chunking" of forasync iterations (to be covered later)

HJ's forall statement = finish + forasync + barriers

Goal 1 (minor): replace common finish-forasync idiom by forall
e.g., replace

```
finish forasync (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    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];
```

Goal 2 (major): Also support "barrier" synchronization

Hello-Goodbye Forall Example

```
1. forall (point[i] : [0:m-1]) {
2.   String s = taskString(i); // returns "task 0" for i=0
3.   System.out.println("Hello from " + s);
4.   System.out.println("Goodbye from " + s);
5. }
```

- Sample output for m = 4

```
Hello from task 0
Hello from task 1
Goodbye from task 0
Hello from task 2
Goodbye from task 2
Goodbye from task 1
Hello from task 3
Goodbye from task 3
```

Hello-Goodbye Forall Example (contd)

```
forall (point[i] : [0:m-1]) {
  String s = taskString(i); // returns "task 0" for i=0
  System.out.println("Hello from " + s);
  System.out.println("Goodbye from " + s);
}
```

- Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye?
- Approach 1: Replace the forall loop by two forall loops, one for the hello's and one for the goodbye's
  - Need to communicate local r values from one forall to the next
- Approach 2: insert a "barrier" between the hello's and goodbye's
  - "next" statement in HJ's forall loops

Barrier Synchronization: HJ's "next" statement in forall loops

```
1. forall (point[i] : [0:m-1]) {
2.   String s = taskString(i); // returns "task 0" fo
3.   System.out.println("Hello from task " + i);
4.   next; // Acts as barrier between phases 0 and 1
5.   System.out.println("Goodbye from task " + i);
6. }
```

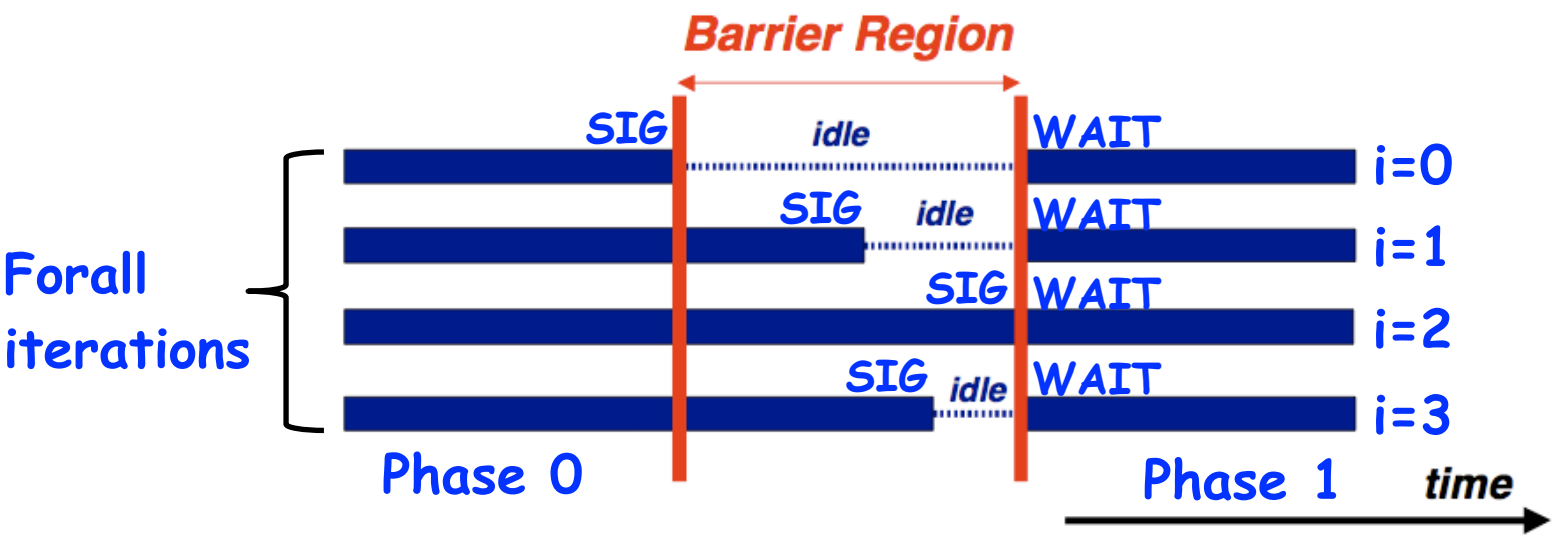
i=0

Phase 0

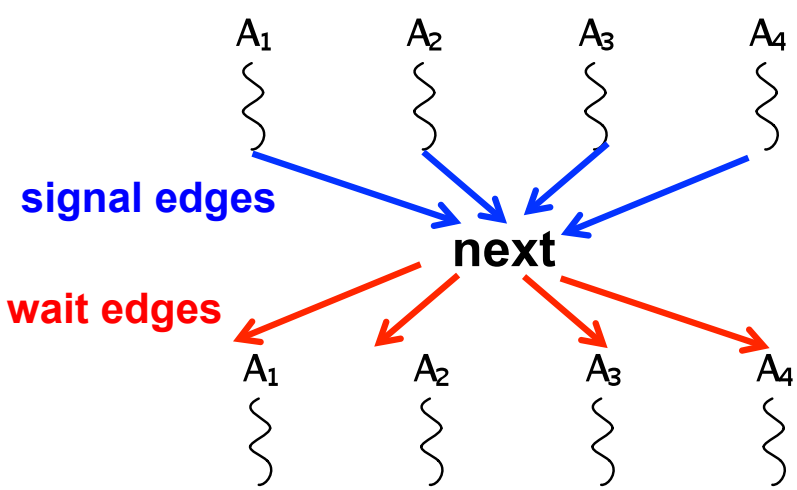
Phase 1

- next → each forall iteration suspends at next until all iterations arrive (complete previous phase), after which the phase can be advanced
  - If a forall iteration terminates before executing "next", then the other iterations do not wait for it
  - Scope of synchronization is the closest enclosing forall statement
  - Special case of "phaser" construct (will be covered in following lectures)

Impact of barrier on scheduling forall iterations



Modeling a next operation in the computation graph



Scope of synchronization for "next" is closest enclosing forall statement

```
forall (point [i] : [0:m-1]) {
  System.out.println("Starting forall iteration " + i);
  next; // Acts as barrier for forall-i
  forall (point [j] : [0:n-1]) {
    System.out.println("Hello from task (" + i + ","
                       + j + ")");
    next; // Acts as barrier for forall-j
    System.out.println("Goodbye from task (" + i + ","
                       + j + ")");
  } // forall-j
  next; // Acts as barrier for forall-i
  System.out.println("Ending forall iteration " + i);
} // forall-i
```

HJ code for One-Dimensional Iterative Averaging with nested for-forall structure

```
double[] myVal=new double[n+2]; double[] myNew=new double[n+2];
myVal[n+1] = 1; // Boundary condition
for (point [iter] : [0:numIters-1]) {
  // Compute MyNew as function of input array MyVal
  forall (point [j] : [1:n]) { // Create n tasks
     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  } // forall
  // Swap myVal and myNew
  double[] temp=myVal; myVal=myNew; myNew=temp;
  // myNew becomes input array for next iteration
} // for
```

- Replace "finish async" from Lecture 10 by "forall"
- Overhead issue --- this version creates (numIters * n) async tasks

HJ code for One-Dimensional Iterative Averaging with barriers (forall-for-next structure)

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2]; gVal[n+1] = 1;
forall (point [j] : [1:n]) {
  double[] myVal = gVal; double[] myNew = gNew; // Local copy of myVal/myNew pointers
  for (point [iter] : [0:numIters-1]) {
    // Compute MyNew as function of input array MyVal
    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next; // Barrier before executing next iteration of iter loop
    // Swap myVal and myNew (each forall iteration swaps
    // its pointers in local vars)
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iter
  } // for
} // forall
```

- Overhead issue --- this version creates n async tasks, but performs numIters barrier operations on n tasks
  - Good trade-off since barrier operations have lower overhead than task creation

Parallel Odd-Even Sort with Shared-Memory Model (teaser for Lecture 5)

```
static void oddEvenSort(int[] A) {
  forall (point [j] : [0:A.length-2]) {
    int numIters = A.length;
    for (point [iter] : [0:numIters-1]) {
     // Assume that numIters is large enough to guarantee sorting
     // In practice, a while loop will be used instead of a sequential-for loop
     if (j%2 == 1 && A[j] > A[j+1]) exchange(A, j, j+1); // ODD PHASE
     next; // Barrier before executing even phase
     if (j%2 == 0 && A[j] > A[j+1]) exchange(A, j, ij+1); // EVEN PHASE
     next; // Barrier before executing odd phase
    } // for
  } // forall
} // oddEvenSort
```

Worksheet #5 (to be done in pairs): Use of seq clause in Quicksort() program

Name 1: _____

Name 2: _____

Insert seq clauses on the right to ensure that an async is only created for calls to quicksort with >= 10,000 elements

```
static void quicksort(int[] A, int M, int N) {
  if (M < N) { // sort A[M...N]
    // partition() selects a pivot element in A[M…N]
    // to partition A[M…N] into A[M…J] and A[I…N]
    point p = partition(A, M, N);
    int I=p.get(0); int J=p.get(1);
    async seq(_______________) quicksort(A, M, J);
    async seq(_______________) quicksort(A, I, N);
  }
} //quicksort
. . .
finish quicksort(A, 0, A.length-1);
```