

CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar

Co-Instructor: Ran Libeskind-Hadas

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

Async and Finish Statements for Task Creation and Termination (Lecture 1)

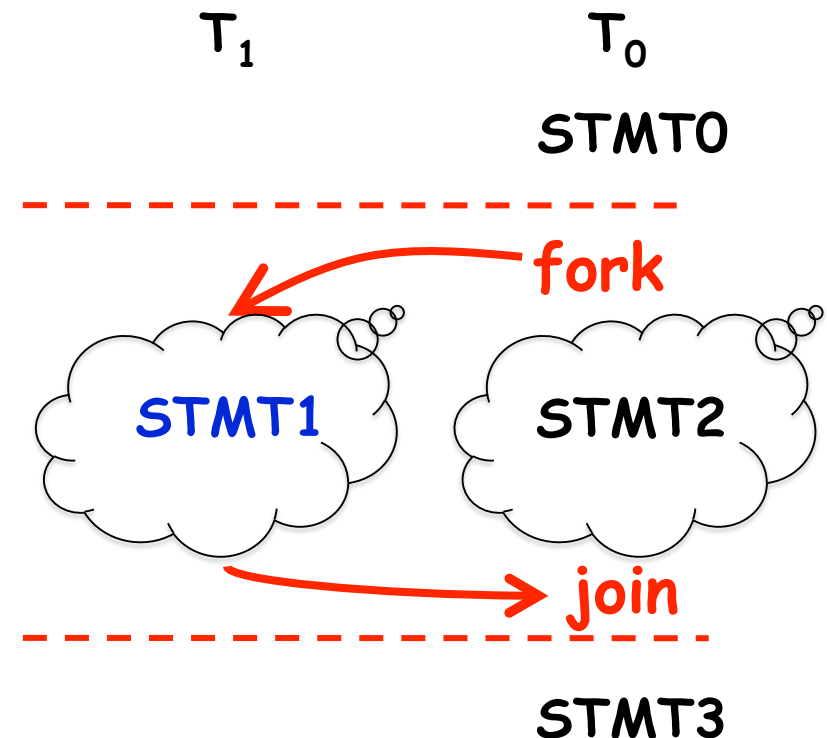
async S

- Creates a new child task that executes statement **S**

finish S

- Execute **S**, but wait until *all* asyncs in **S**'s scope have terminated.

```
// T0 (Parent task)
STMT0;
finish { //Begin finish
  async {
    STMT1; //T1 (Child task)
  }
  STMT2; //Continue in T0
          //Wait for T1
} //End finish
STMT3; //Continue in T0
```

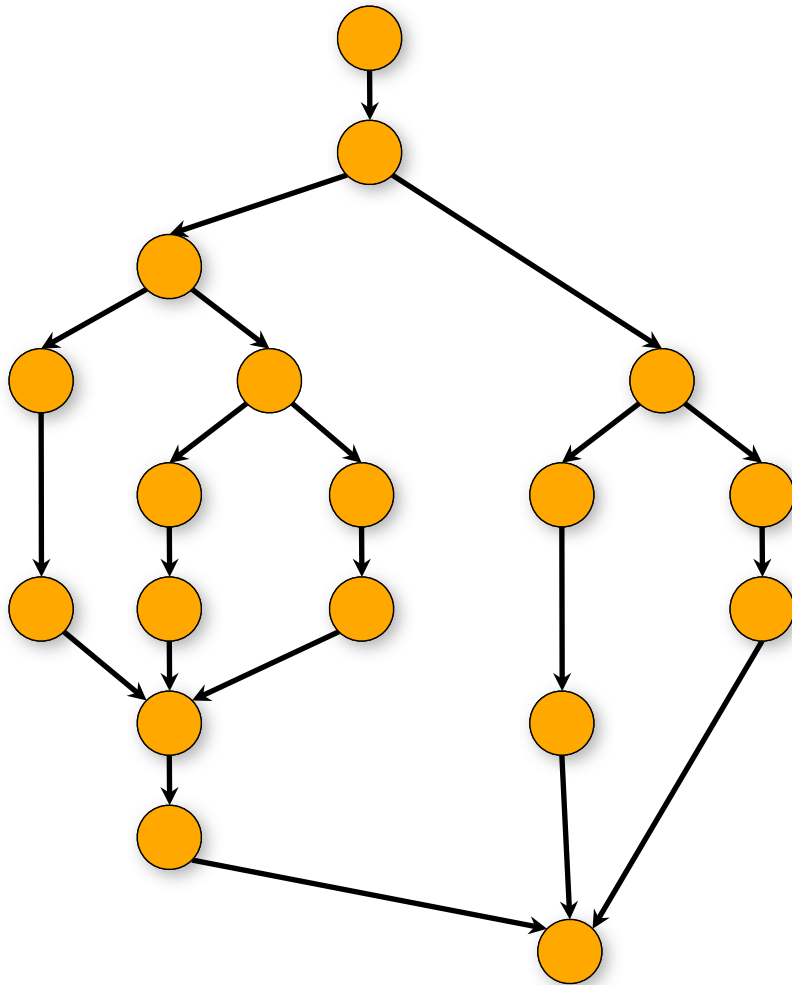


Worksheet #1 Solution: Insert finish to get correct Two-way Parallel Array Sum program

```
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.      // T0 computes sum of lower half of array
8.      for(int i=0; i < X.length/2; i++) sum1 += X[i];
9.  }
10. // Task T0 waits for Task T1 (join)
11. return sum1 + sum2;
```

Worksheet #2 solution: what is the critical path length and ideal speedup of this graph?

- Assume $\text{time}(N) = 1$ for all nodes in this graph



WORK(G) = 18

$CPL(G) = 9$

Ideal Speedup = 2

Formal Definition of Data Races

(Lecture 2)

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) $S1$ and $S2$ in CG such that:

1. $S1$ does not depend on $S2$ and $S2$ does not depend on $S1$ i.e., there is no path of dependence edges from $S1$ to $S2$ or from $S2$ to $S1$ in CG , and
2. Both $S1$ and $S2$ read or write L , and at least one of the accesses is a write.

Data races are challenging because of

- Nondeterminism: different executions of the parallel program with the same input may result in different outputs.
- Debugging and Testing: it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program debugging and testing.

Example of Incorrect Parallel Program in Homework 0 (Problem 1.2)

```
1. // Sequential version
2. for ( p = first; p != null; p = p.next) p.x = p.y + p.z;
3. for ( p = first; p != null; p = p.next) sum += p.x;
4.
5. // Incorrect parallel version
6. for ( p = first; p != null; p = p.next)
7.     async p.x = p.y + p.z;
8. for ( p = first; p != null; p = p.next)
9.     sum += p.x;
```

Why is the parallel version incorrect?

Data race between write of `p.x` in line 7 and read of `p.x` in line 9 !

Relating Data Races and Determinism

- A parallel program is said to be deterministic with respect to its inputs if it always computes the same answer when given the same inputs.
- Structural Determinism Property
 - If a parallel program is written using the constructs in Module 1 and is guaranteed to be race-free, then it must be deterministic with respect to its inputs. The final computation graph is also guaranteed to be the same for all executions of the program with the same inputs.
- Constructs introduced in Module 1 (“Deterministic Shared-Memory Parallelism”) include `async`, `finish`, `finish accumulators`, `futures`, `data-driven tasks (async await)`, `forall`, `barriers`, `phasers`, and `phaser accumulators`.
 - The notable exceptions are critical sections, isolated statements, and actors, all of which will be covered in Module 2 (“Nondeterministic Shared-Memory Parallelism”)

Worksheet #3 solution: Complexity analysis of k-way Parallel Array Sum algorithm

- Consider a k-way parallel array-sum algorithm, where $1 \leq k \leq n$
 - Compute k partial sums in parallel, each of size n/k
 - Sequentially combine the k partial sums into a single sum
- Total number of additions, $WORK = k(n/k - 1) + k = O(n)$
- What is the critical path length?
 - CPL = $O(n/k + k)$
 - Stage 1 takes $O(n/k)$ time and Stage 2 takes $O(k)$ time
- What value of k gives the smallest value of CPL?
 - Optimal value of $k = \sqrt{n}$

HJ Futures: Tasks with Return Values

(Lecture 3)

async<T> { Stmt-Block }

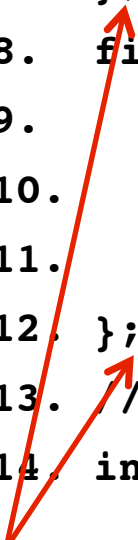
- Creates a new child task that executes **Stmt-Block**, which must terminate with a **return** statement returning a value of type **T**
- Async expression returns a reference to a container of type **future<T>**
- Values of type **future<T>** can only be assigned to final variables

Expr.get()

- Evaluates **Expr**, and blocks if **Expr**'s value is unavailable
- **Expr** must be of type **future<T>**
- Return value from **Expr.get()** will then be **T**
- Unlike **finish** which waits for all tasks in the **finish** scope, a **get()** operation only waits for the specified **async** expression

Example: Two-way Parallel Array Sum using Future Tasks

```
1.  // Parent Task T1 (main program)
2.  // Compute sum1 (lower half) and sum2 (upper half) in parallel
3.  final future<int> sum1 = async<int> { // Future Task T2
4.      int sum = 0;
5.      for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
6.      return sum;
7.  }; //NOTE: semicolon needed to terminate assignment to sum1
8.  final future<int> sum2 = async<int> { // Future Task T3
9.      int sum = 0;
10.     for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
11.     return sum;
12. }; //NOTE: semicolon needed to terminate assignment to sum2
13. //Task T1 waits for Tasks T2 and T3 to complete
14. int total = sum1.get() + sum2.get();
```



Why are these semicolons needed?

Extending HJ Futures for Macro-Dataflow: Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

```
ddfA = new DataDrivenFuture<T1>();
```

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1

```
async await(ddfA, ddfB, ...) Stmt
```

- Create a new data-driven-task to start executing **Stmt** after all of **ddfA, ddfB, ...** become available (i.e., after task becomes “enabled”)

```
ddfA.put(V) ;
```

- Store object V (of type T1) in **ddfA**, thereby making **ddfA** available
- Single-assignment rule: at most one put is permitted on a given DDF

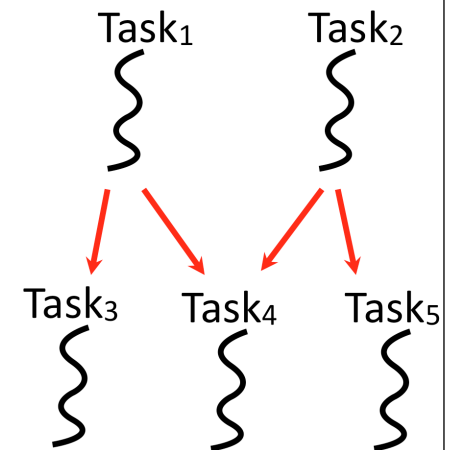
```
ddfA.get()
```

- Return value (of type T1) stored in **ddfA**
- Can only be performed by async's that contain **ddfA** in their await clause (hence no blocking is necessary for DDF gets)

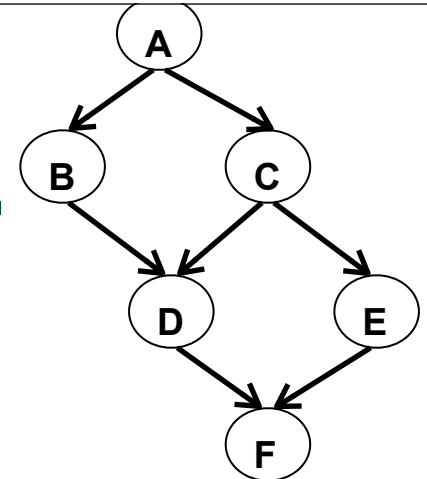
Example Habanero Java code fragment with Data-Driven Futures

```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.     async await(left) leftReader(left); // Task3
5.     async await(right) rightReader(right); // Task5
6.     async await(left, right)
7.         bothReader(left, right); // Task4
8.     async left.put(leftWriter()); // Task1
9.     async right.put(rightWriter()); // Task2
10. }
```

- **await** clauses capture data flow relationships
- **type** parameter is optional for DataDrivenFuture
 - if omitted, may require cast operators to be inserted instead
 - (just as with standard Java generics in sequential programs)



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



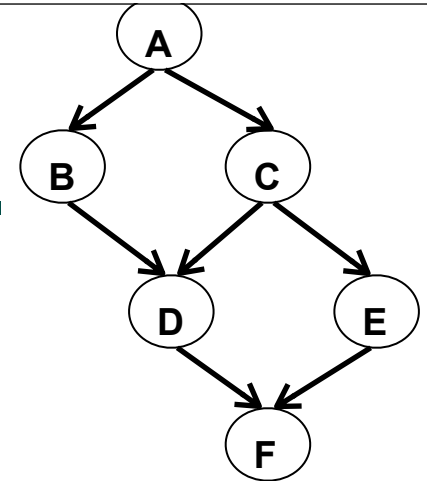
1) Can you write an HJ program with async-finish constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? **No**

2) Can you write an HJ program with future async-get 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

```
1. final future<void> A = async<void>
2.     { . . . };
3. final future<void> B = async<void>
4.     { A.get(); . . . };
5. final future<void> C = async<void>
6.     { A.get(); . . . };
7. final future<void> D = async<void>
8.     { B.get(); C.get(); . . . };
9. final future<void> E = async<void>
10.    { C.get(); . . . };
11. final future<void> F = async<void>
12.    { D.get(); E.get(); . . . };
```

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



1) Can you write an HJ program with async-finish 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 future async-get 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

```
1. final future<void> A = async<void>
2.     { . . . };
3. final future<void> B = async<void>
4.     { A.get(); . . . };
5. final future<void> C = async<void>
6.     { A.get(); . . . };
7. final future<void> D = async<void>
8.     { B.get(); C.get(); . . . };
9. final future<void> E = async<void>
10.    { C.get(); . . . };
11. final future<void> F = async<void>
12.    { D.get(); E.get(); . . . };
```

HJ's pointwise for & forasync statements (Lecture 4)

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];
```

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], ...
 - Dimensions of a point are numbered from 0 to $n-1$
 - n is also referred to as the rank 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 $(i \bmod p1.rank)$ if $i < 0$ or $i \geq p1.rank$
 - $p1.lt(p2)$, $p1.le(p2)$, $p1.gt(p2)$, $p1.ge(p2)$
 - Returns true iff $p1$ is lexicographically $<$, \leq , $>$, or \geq $p2$
 - Only defined when $p1.rank$ and $p2.rank$ are equal
-

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`

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

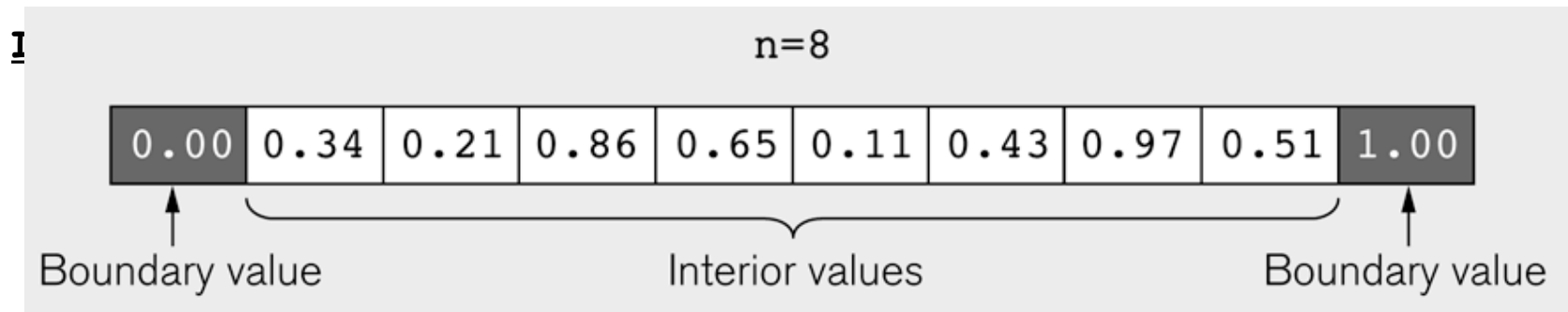
```
for (point[i] : [1:m-1]) // Equivalent to "for (i=1;i<m;i++)"
```

```
  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 $1..n$



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

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

- **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)

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

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

- Overhead issue --- this version creates $(\text{numIters} * n)$ async tasks

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

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

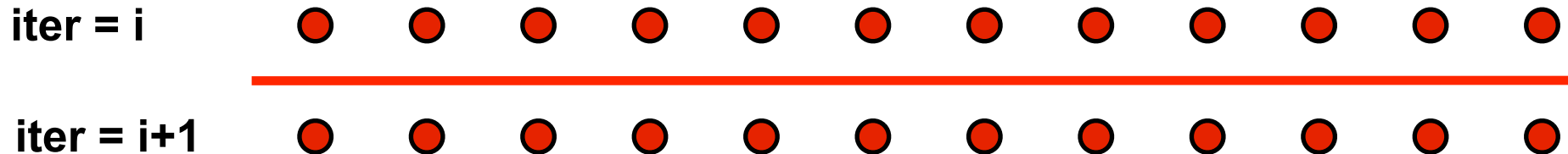
- Overhead issue --- this version creates n async tasks, but performs numIters barrier operations on n tasks
 - Good trade-off since barrier operations have lower overhead than task creation

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

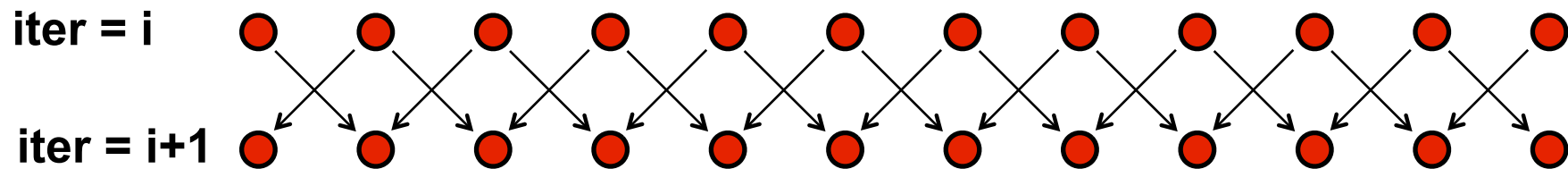
Insert seq clauses
on the right to
ensure that an
async is only
created for calls to
quicksort with \geq
10,000 elements

```
1. static void quicksort(int[] A, int M, int N) {
2.   if (M < N) { // sort A[M...N]
3.     // partition() selects a pivot element in A[M...N]
4.     // to partition A[M...N] into A[M...J] and A[I...N]
5.     point p = partition(A, M, N);
6.     int I=p.get(0); int J=p.get(1);
7.     async seq(J-M+1 < 10000) quicksort(A, M, J);
8.     async seq(N-I+1 < 10000) quicksort(A, I, N);
9.   }
10.} //quicksort
11. . . .
12. finish quicksort(A, 0, A.length-1);
```


Barrier vs Point-to-Point Synchronization for One-Dimensional Iterative Averaging Example (Lecture 6)



Barrier synchronization



Point-to-point synchronization

(Left-right neighbor synchronization)

Summary of Phaser Construct

- Phaser allocation
 - `phaser ph = new phaser(mode);`
 - Phaser `ph` is allocated with registration mode
 - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)
- Registration Modes
 - `phaserMode.SIG`, `phaserMode.WAIT`, `phaserMode.SIG_WAIT`, `phaserMode.SIG_WAIT_SINGLE`
 - NOTE: phaser `WAIT` has no relationship to Java `wait/notify`
- Phaser registration
 - `async phased (ph1<mode1>, ph2<mode2>, ...) <stmt>`
 - Spawned task is registered with `ph1` in `mode1`, `ph2` in `mode2`, ...
 - Child task's capabilities must be subset of parent's
 - `async phased <stmt>` propagates all of parent's phaser registrations to child
- Synchronization
 - `next;`
 - Advance each phaser that current task is registered on to its next phase
 - Semantics depends on registration mode

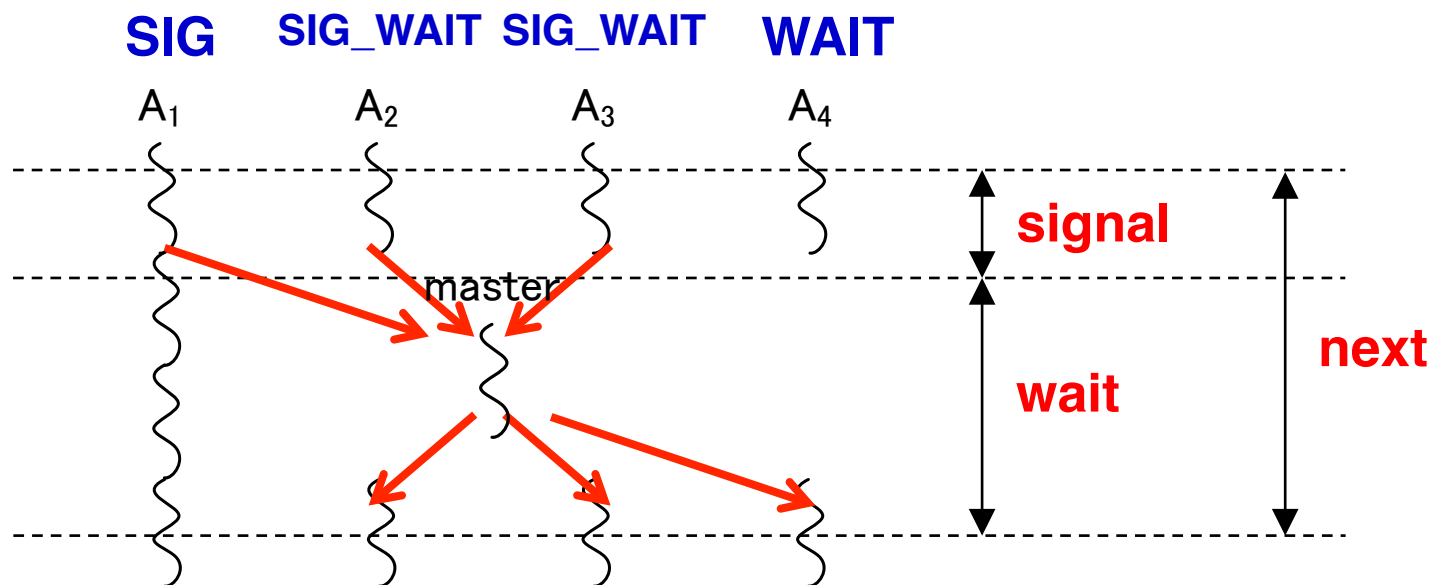
Simple Example with Four Async Tasks and One Phaser (contd)

Semantics of **next** depends on registration mode

SIG_WAIT: **next** = **signal** + **wait**

SIG: **next** = **signal** (Don't wait for any task)

WAIT: **next** = **wait** (Don't disturb any task)



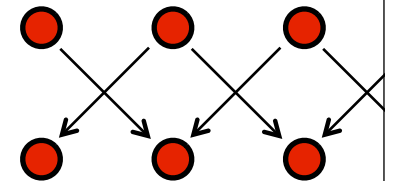
A master task **receives all signals** and **broadcasts a barrier completion**

One-Dimensional Iterative Averaging with Point-to-Point Synchronization

```
1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[n+1] = 1; gNew[n+1] = 1;
3. phaser ph = new phaser[n+2];
4. finish { // phasers must be allocated in finish scope
5.   forall(point [i]:[0:n+1]) ph[i] = new phaser();
6.   forasync(point [j]:[1:n]) phased(ph[j]<SIG>,ph[j-1]<WAIT>,ph[j+1]<WAIT>){
7.     double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
8.     for (point [iter] : [0:numIters-1]) {
9.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.      next; // Point-to-point synchronization
11.      // Swap myVal and myNew
12.      double[] temp=myVal; myVal=myNew; myNew=temp;
13.      // myNew becomes input array for next iter
14.    } // for-iter
15.  } // forasync-j
16.} // finish
```

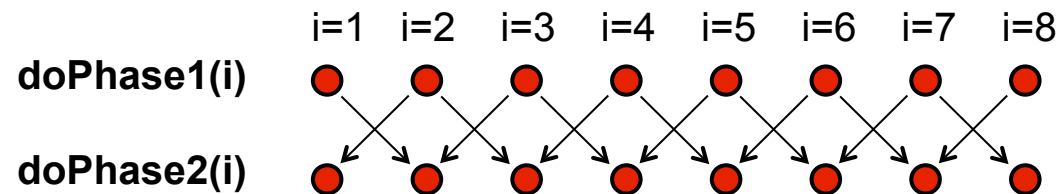
iter = i

iter = i+1



Worksheet #7 solution:

Left-Right Neighbor Synchronization using Phasers



Complete the phased clause below to implement the left-right neighbor synchronization shown above

```
1. finish {
2.   phaser[] ph = new phaser[m+2];
3.   for(point [i]:[0:m+1]) ph[i] = new phaser();
4.   for(point [i] : [1:m])
5.     async phased(ph[i-1]<WAIT>, ph[i]<SIG>, ph[i+1]<WAIT>) {
6.       doPhase1(i);
7.       next;
8.       doPhase2(i);
9.     }
10. }
```

Summary of Module 1: Deterministic Shared-Memory Parallelism

- **Serializable subset of HJ**
 - {async, finish, future, forasync}
 - Erasure property: any HJ program written using the above constructs can be converted to an equivalent sequential program by erasing all parallel constructs
 - **Deadlock-free subset of HJ**
 - {next, barriers, phasers, forall, async phased} + serializable subset
 - Deadlock-freedom property: any HJ program written using the above constructs is guaranteed to never deadlock
 - **Deterministic subset of HJ**
 - {data driven futures, async await} + deadlock-free subset
 - Data-race-free structural determinism property: if any HJ program written using the above constructs is guaranteed to be data-race-free for a given input, then it must also be deterministic for that input i.e., all executions with the same input must generate the same output AND the same computation graph
-

HJ isolated statement (Lecture 7)

isolated <body>

- Isolated statement identifies a critical section
 - Two tasks executing isolated statements must perform them in mutual exclusion
 - Weak isolation guarantee: mutual exclusion applies to (isolated, isolated) pairs of statement instances, but not to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
 - That's why we call this construct “isolated” instead of “atomic”
 - Isolated statements may be nested (redundant)
 - Isolated statements must not contain any other parallel statement that performs a blocking operation: *finish, future get, next, async await*
 - Non-blocking operations (e.g., *async*) are fine
 - Isolated statements can never deadlock
-

Object-based isolation in HJ

`isolated(<object-list>) <body>`

- In this case, programmer specifies list of objects for which isolation is required
 - Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists
 - Standard isolated is equivalent to `isolated(*)` by default i.e., isolation across all objects
 - Implementation can choose to distinguish between read/write accesses for further parallelism
 - Current HJ implementation supports object-based isolation, but does not exploit read/write distinction
-

DoublyLinkedListNode Example revisited with Object-Based Isolation

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         isolated(this.prev, this, this.next) { // object-based isolation
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         }
9.         . . .
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(DoublyLinkedListNode L) {
14.     finish {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async second.delete();
18.         async third.delete();
19.     }
20. }
```

java.util.concurrent.atomic.AtomicInteger

- **Constructors**
 - `new AtomicInteger()`
 - Creates a new `AtomicInteger` with initial value 0
 - `new AtomicInteger(int initialValue)`
 - Creates a new `AtomicInteger` with the given initial value
- **Selected methods**
 - `int addAndGet(int delta)`
 - Atomically adds delta to the current value of the atomic variable, and returns the new value
 - `int getAndAdd(int delta)`
 - Atomically returns the current value of the atomic variable, and adds delta to the current value
- Similar interfaces available for `LongInteger`

Worksheet #8 solution:

Insertion of isolated for correctness

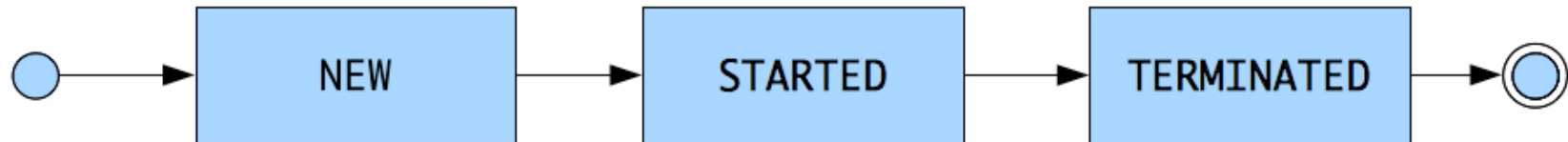
The goal of IsolatedPRNG is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated statement(s) that you can insert in method nextSeed() to avoid data races and guarantee proper semantics.

```
class IsolatedPRNG {
  private int seed;
  public int nextSeed() {
    int retVal;
    isolated {
      retVal = seed;

      seed = nextInt(retVal);
    }
    return retVal;
  } // nextSeed()
  . . .
} // IsolatedPRNG
```

```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async { print r.nextSeed(); ... }
  async { print r.nextSeed(); ... }
} // main()
```

Actor Life Cycle (Lecture 9)



Actor states

- New: Actor has been created
 - e.g., email account has been created
- Started: Actor can receive and process messages
 - e.g., email account has been activated
- Terminated: Actor will no longer processes messages
 - e.g., termination of email account after graduation

Using Actors in HJ

- Create your custom class which extends `hj.lang.Actor<Object>` ,and implement the `void process()` method

```
class MyActor extends Actor<Object> {  
    protected void process(Object message) {  
        System.out.println("Processing " + message);  
    }  
}
```

- Instantiate and start your actor

```
Actor<Object> anActor = new MyActor(); anActor.start();
```

- Send messages to the actor

```
anActor.send(aMessage); //aMessage can be any object in general
```

- Use a special message to terminate an actor

```
protected void process(Object message) {  
    if (message.someCondition()) exit();  
}
```

- Actor execution implemented as async tasks in HJ

- Can use **finish** to await their completion
-

Hello World Example

```
1. public class HelloWorld {
2.     public static void main(String[] args) {
3.         EchoActor actor = new EchoActor();
4.         actor.start(); // don't forget to start the actor
5.         actor.send("Hello"); // asynchronous send (returns immediately)
6.         actor.send("World");
7.         actor.send(EchoActor.STOP_MSG);
8.     }
9. }
10. class EchoActor extends Actor<Object> {
11.     static final Object STOP_MSG = new Object();
12.     private int messageCount = 0;
13.     protected void process(final Object msg) {
14.         if (STOP_MSG.equals(msg)) {
15.             println("Message-" + messageCount + ": terminating.");
16.             exit(); // never forget to terminate an actor
17.         } else {
18.             messageCount += 1;
19.             println("Message-" + messageCount + ": " + msg);
20.         }
21.     }
22. }
```

**Sends are asynchronous
in actor model, but HJ
Actor library preserves
order of messages
between same sender and
receiver**

Worksheet #9 solution: Interaction between finish and actors

What would happen if the end-finish operation from slide 29 was moved from line 13 to line 11 as shown below?

```
1. finish {
2.     int numThreads = 4;
3.     int numberOfHops = 10;
4.     ThreadRingActor[] ring = new ThreadRingActor[numThreads];
5.     for(int i=numThreads-1;i>=0; i--) {
6.         ring[i] = new ThreadRingActor(i);
7.         ring[i].start();
8.         if (i < numThreads - 1) {
9.             ring[i].nextActor(ring[i + 1]);
10.    } }
11. } // finish
12. ring[numThreads-1].nextActor(ring[0]);
13. ring[0].send(numberOfHops);
```

Deadlock: the end-finish operation in line 11 waits for all the actors created in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit()

Linearizability of Concurrent Objects (Lecture 10)

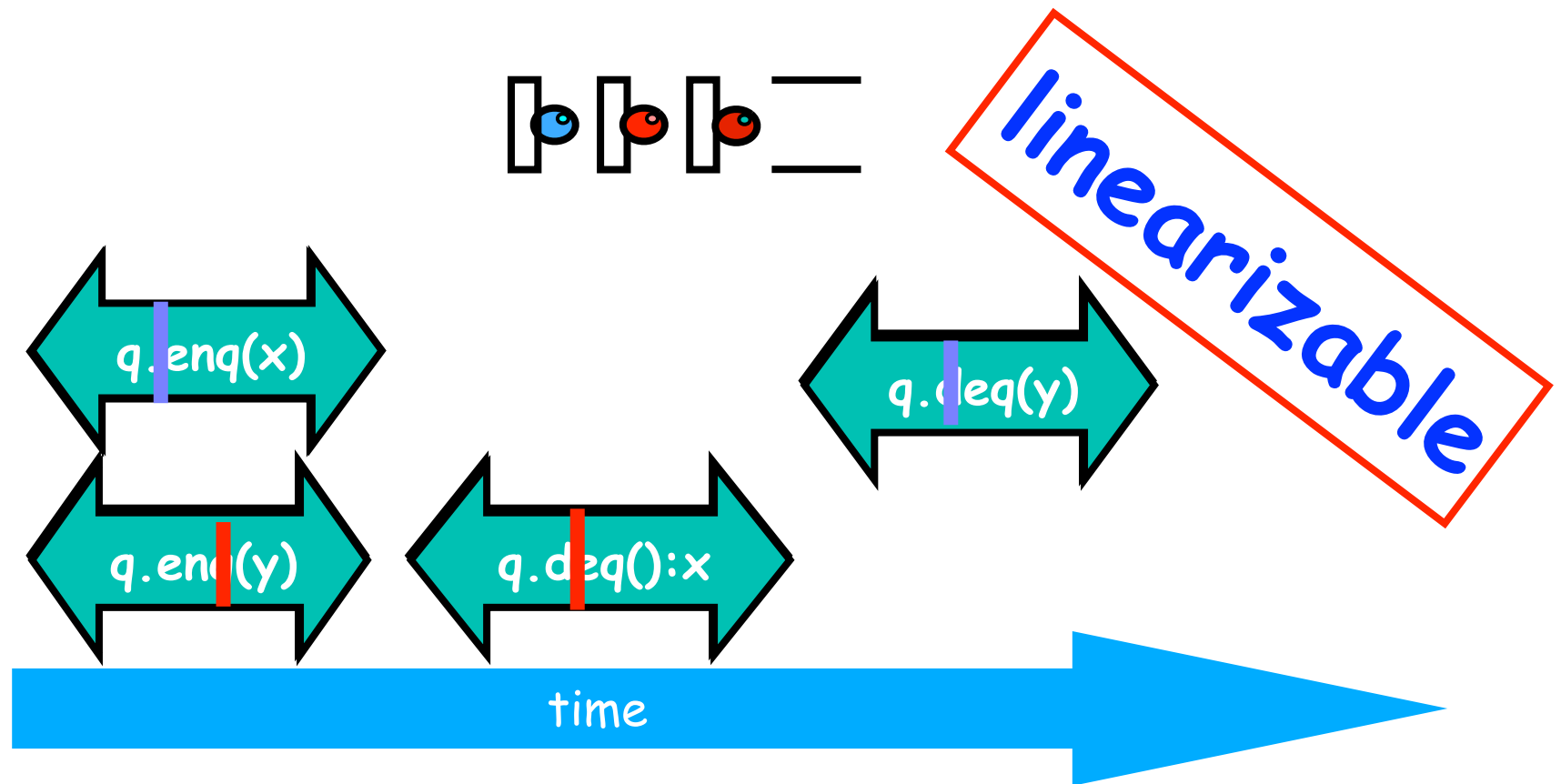
Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
 - Examples: concurrent queue, AtomicInteger

Linearizability

- Assume that each method call takes effect “instantaneously” at some distinct point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- An object is linearizable if all its possible executions are linearizable

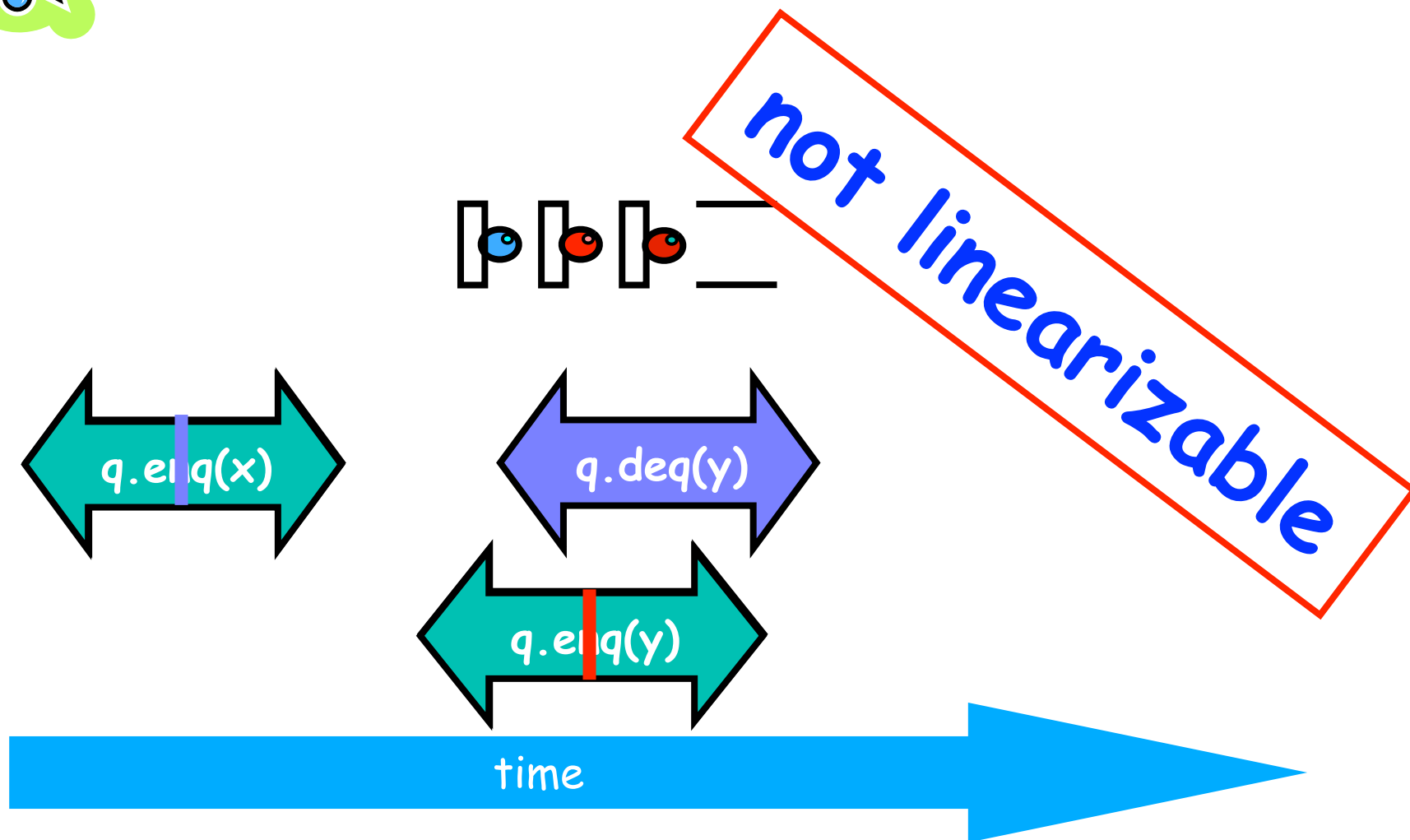
Example 1



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



Example 2



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Worksheet #10 solution:

Linearizability of method calls on a concurrent object

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke <code>q.enq(x)</code>	
1	Return from <code>q.enq(x)</code>	
2		Invoke <code>q.enq(y)</code>
3	Invoke <code>q.deq()</code>	Work on <code>q.enq(y)</code>
4	Work on <code>q.deq()</code>	Return from <code>q.enq(y)</code>
5	Return <code>y</code> from <code>q.deq()</code>	

No! `q.enq(x)` must precede `q.enq(y)` in all linear sequences of method calls invoked on `q`. It is illegal for the `q.deq()` operation to return `y`.

Places in HJ (Lecture 11)

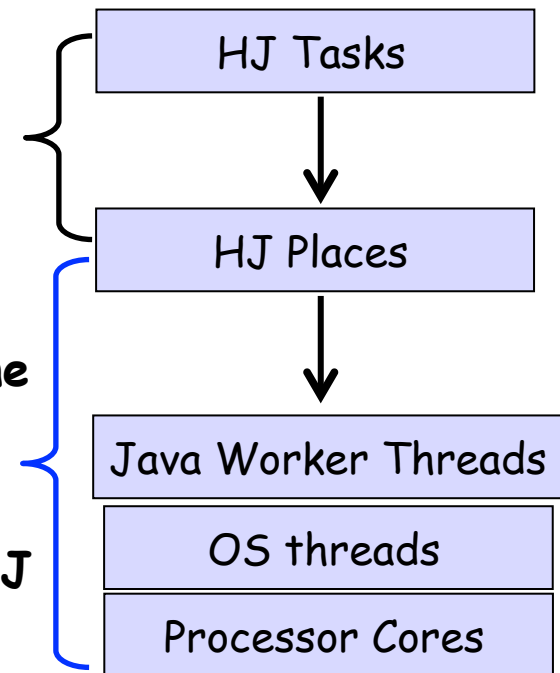
HJ programmer defines mapping from HJ tasks to set of places

HJ runtime defines mapping from places to one or more worker Java threads per place

The option “-places **p:w**” when executing an HJ program can be used to specify

p, the number of places

w, the number of worker threads per place



Places in HJ

here = place at which current task is executing

place.MAX_PLACES = total number of places (runtime constant)

Specified by value of **p** in runtime option, **-places p:w**

place.factory.place(i) = place corresponding to index *i*

<place-expr>.toString() returns a string of the form "place(id=0)"

<place-expr>.id returns the id of the place as an int

async at(P) S

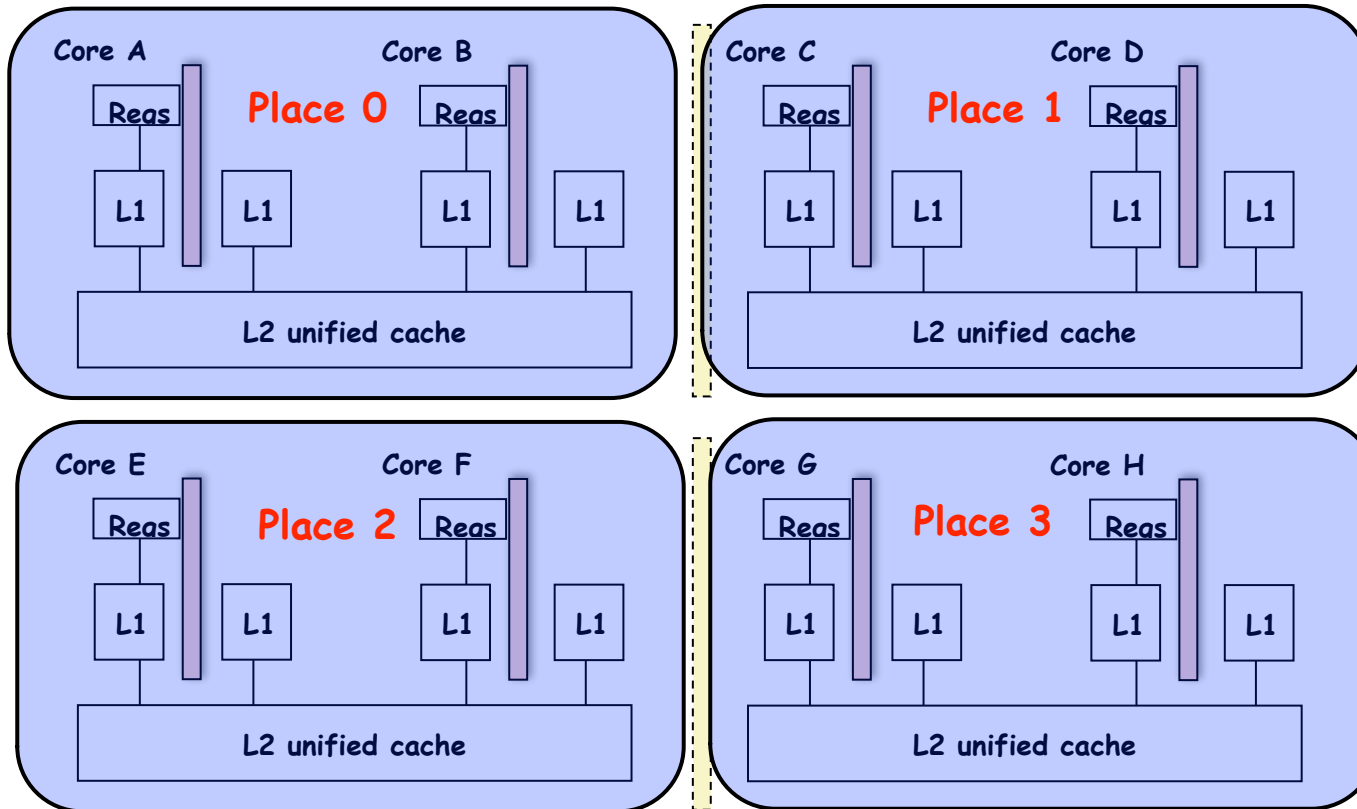
- Creates new task to execute statement *S* at place *P*
- **async S** is equivalent to **async at(here) S**
- Main program task starts at **place.factory.place(0)**

Note that **here** in a child task refers to the place *P* at which the child task is executing, not the place where the parent task is executing

Example of –places 4:2 option on an 8-core node (4 places w/ 2 workers per place)

```
// Main program starts at place 0  
async at(place.factory.place(0)) S1;  
async at(place.factory.place(0)) S2;
```

```
async at(place.factory.place(1)) S3;  
async at(place.factory.place(1)) S4;  
async at(place.factory.place(1)) S5;
```



```
async at(place.factory.place(2)) S6;  
async at(place.factory.place(2)) S7;  
async at(place.factory.place(2)) S8;
```

```
async at(place.factory.place(3)) S9;  
async at(place.factory.place(3)) S10;
```

Example HJ program with places

```
1  class T1 {
2      final place affinity;
3      . . .
4      // T1's constructor sets affinity to place where instance was created
5      T1() { affinity = here; ... }
6      . . .
7  }
8  . . .
9  finish { // Inter-place parallelism
10     System.out.println("Parent_place_=", here); // Parent task's place
11     for (T1 a = . . .) {
12         async at (a.affinity) { // Execute async at place with affinity to a
13             a.foo();
14             System.out.println("Child_place_=", here); // Child task's place
15         } // async
16     } // for
17 } // finish
18 . . .
```

MPI: The Message Passing Interface

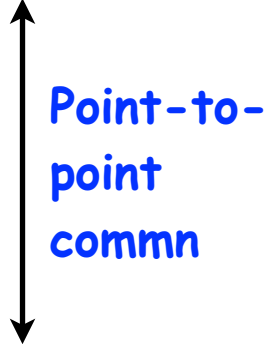
- Sockets and Remote Method Invocation (RMI) are communication primitives used for distributed Java programs.
 - Designed for standard TCP/IP networks rather than high-performance interconnects
- The Message Passing Interface (MPI) standard was designed to exploit high-performance interconnects
 - MPI was standardized in the early 1990s by the MPI Forum—a substantial consortium of vendors and researchers
 - <http://www-unix.mcs.anl.gov/mpi>
 - It is an API for communication between nodes of a distributed memory parallel computer
 - The original standard defines bindings to C and Fortran (later C++)
 - Java support is available from a research project, mpiJava, developed at Indiana University 10+ years ago
<http://www.hpjava.org/mpiJava.html>

Our First MPI Program (mpiJava version)

main() is enclosed in an implicit "forall" --- each process runs a separate instance of main() with "index variable" = myrank

```
1.import mpi.*;
2.class Hello {
3.    static public void main(String[] args) {
4.        // Init() be called before other MPI calls
5.        MPI.Init(args); /
6.        int npes = MPI.COMM_WORLD.Size()
7.        int myrank = MPI.COMM_WORLD.Rank() ;
8.        System.out.println("My process number is " + myrank);
9.        MPI.Finalize(); // Shutdown and clean-up
10.    }
11.}
```

Adding Send() and Recv() to the Minimal Set of MPI Routines (mpiJava)

- `MPI.Init(args)`
 - initialize MPI in each process
 - `MPI.Finalize()`
 - terminate MPI
 - `MPI.COMM_WORLD.Size()`
 - number of processes in `COMM_WORLD` communicator
 - `MPI.COMM_WORLD.Rank()`
 - rank of this process in `COMM_WORLD` communicator
 - `MPI.COMM_WORLD.Send()`
 - send message using `COMM_WORLD` communicator
 - `MPI.COMM_WORLD.Recv()`
 - receive message using `COMM_WORLD` communicator
- 

Example with Send() and Recv() calls

```
1.import mpi.*;

3.class myProg {
4.  public static void main( String[] args ) {
5.      int tag0 = 0;
6.      MPI.Init( args );                // Start MPI computation
7.      if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
8.          int loop[] = new int[1]; loop[0] = 3;
9.          MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
10.         MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
11.     } else {                          // rank 1 = receiver
12.         int loop[] = new int[1]; char msg[] = new char[12];
13.         MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
14.         MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
15.         for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
16.     }
17.     MPI.Finalize( );                  // Finish MPI computation
18. }
19.}
```

Send() and Recv() calls are blocking operations by default

mpiJava send and receive (Lecture 12)

- **Send and receive members of Comm:**

`void Send(Object buf, int offset, int count, Datatype type, int dst, int tag) ;`

`Status Recv(Object buf, int offset, int count, Datatype type, int src, int tag) ;`

- The arguments `buf`, `offset`, `count`, `type` describe the data buffer—the storage of the data that is sent or received. They will be discussed on the next slide.
 - `dst` is the rank of the destination process relative to this communicator. Similarly in `Recv()`, `src` is the rank of the source process.
 - An arbitrarily chosen tag value can be used in `Recv()` to select between several incoming messages: the call will wait until a message sent with a matching tag value arrives.
 - The `Recv()` method returns a `Status` value, discussed later.
 - Both `Send()` and `Recv()` are blocking operations
—Analogous to phaser next operations
-

Deadlock Scenario #1

Consider:

```
int a[], b[];
...
if (MPI.COMM_WORLD.rank() == 0) {
    MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
    MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
}
else {
    Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
    Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
}
...
```

Blocking semantics for Send() and Recv() can lead to a deadlock.

Deadlock Scenario #2

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (modulo the number of processes)

```
int a[], b[];
. . .
int npes = MPI.COMM_WORLD.siz();
int myrank = MPI.COMM_WORLD.rank()
MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, (myrank+1)%npes, 1);
MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, (myrank-1+npes)%npes, 1);
```

Once again, we have a deadlock, since `Send()` and `Recv()` are blocking operations

Non-Blocking Send and Receive operations

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate")
- The method signatures for `Isend()` and `Irecv()` are similar to those for `Send()` and `Recv()`, except that `Isend()` and `Irecv()` return objects of type `Request`:
`Request Isend(Object buf, int offset, int count, Datatype type, int dst, int tag) ;`
`Request Irecv(Object buf, int offset, int count, Datatype type, int src, int tag) ;`
- Function `Test()` tests whether or not the non-blocking send or receive operation identified by its request has finished.
`Status Test(Request request)`
- `Wait` waits() for the operation to complete (like a future `get()` operation)
`Status Wait(Request request)`

Simple Irecv() example

- The simplest way of waiting for completion of a single non-blocking operation is to use the instance method `Wait()` in the `Request` class, e.g:

```
// Post a receive operation
Request request =
    Irecv(intBuf, 0, n, MPI.INT, MPI.ANY_SOURCE, 0) ;
// Do some work while the receive is in progress
...
// Finished that work, now make sure the message has arrived
Status status = request.Wait() ;
// Do something with data received in intBuf
...
```

- The `Wait()` operation is declared to return a `Status` object. In the case of a non-blocking receive operation, this object has the same interpretation as the `Status` object returned by a blocking `Recv()` operation.
-

Collective Communications

- A popular feature of MPI is its family of collective communication operations.
- Each of these operations is defined over a communicator.
 - All processes in a communicator must perform the same operation
 - Implicit barrier (next)
- The simplest example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

`void Bcast(Object buf, int offset, int count, Datatype type, int root)`

- Broadcast a message from the process with rank root to all processes of the group.

More Examples of Collective Operations

- All the following are instance methods of Intracom:

`void Barrier()`

- Blocks the caller until all processes in the group have called it.

`void Gather(Object sendbuf, int sendoffset, int sendcount,
Datatype sendtype, Object recvbuf, int recvoffset, int recvcount,
Datatype recvtype, int root)`

- Each process sends the contents of its send buffer to the root process.

`void Scatter(Object sendbuf, int sendoffset, int sendcount,
Datatype sendtype, Object recvbuf, int recvoffset, int recvcount,
Datatype recvtype, int root)`

- Inverse of the operation Gather.

`void Reduce(Object sendbuf, int sendoffset, Object recvbuf,
int recvoffset, int count, Datatype datatype, Op op, int root)`

- Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the root process.

MPI_Reduce

```
void MPI.COMM_WORLD.Reduce(
```

Object[]	sendbuf	/* in */,
int	sendoffset	/* in */,
Object[]	recvbuf	/* out */,
int	recvoffset	/* in */,
int	count	/* in */,
MPI.Datatype	datatype	/* in */,
MPI.Op	operator	/* in */,
int	root	/* in */)



```
MPI.COMM_WORLD.Reduce( msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```

java.lang.Thread class

(a glimpse of real-world parallel programming)

- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.
- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```
1 public class Thread extends Object implements Runnable {
2     Thread() { ... } // Creates a new Thread
3     Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
4     void run() { ... } // Code to be executed by thread
5     // Case 1: If this thread was created using a Runnable object ,
6     //           then that object's run method is called
7     // Case 2: If this class is subclassed , then the run() method
8     //           in the subclass is called
9     void start() { ... } // Causes this thread to start execution
10    void join() { ... } // Wait for this thread to die
11    void join(long m) // Wait at most m milliseconds for thread to die
12    static Thread currentThread() // Returns currently executing thread
13    . . .
14 }
```

Listing 3: java.lang.Thread class

java.lang.Runnable interface

- Any class that implements java.lang.Runnable must provide a parameter-less run() method with void return type
- Lines 3-7 in Listing 2 show the creation of an instance of an anonymous inner class that implements the Runnable interface
- The computation in the run() method can be invoked sequentially by calling r.run()
 - We will see next how it can be invoked in parallel

```
1  . . .
2  final int len = X.length;
3  Runnable r = new Runnable() {
4      public void run() {
5          for(int i=0 ; i < len/2 ; i++) sum1 += X[i];
6      }
7  };
8  . . .
```

Listing 2: Example of creating a Java Runnable instance as a closure

start() and join() methods

- A Thread instance starts executing when its start() method is invoked
 - start() can be invoked at most once per Thread instance
 - As with async, the parent thread can immediately move to the next statement after invoking t.start()
- A t.join() call forces the invoking thread to wait till thread t completes.
 - Lower-level primitive than finish since it only waits for a single thread rather than a collection of threads
 - No restriction on which thread performs a join on which thread, so it is possible to create a deadlock cycle using join()
 - No notion of an Immediately Enclosing Finish in Java threads
 - No propagation of exceptions to parent/ancestor threads

Listing 4: Two-way Parallel ArraySum using Java threads

```
1 // Start of Task T1 (main program)
2 sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
3 // Compute sum1 (lower half) and sum2 (upper half) in parallel
4 final int len = X.length;
5 Runnable r1 = new Runnable() {
6     public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
7 };
8 Thread t1 = new Thread(r1);
9 t1.start();
10 Runnable r2 = new Runnable() {
11     public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
12 };
13 Thread t2 = new Thread(r2);
14 t2.start();
15 // Wait for threads t1 and t2 to complete
16 t1.join(); t2.join();
17 int sum = sum1 + sum2;
```

Acknowledgments

- Grutors
 - Matt Prince
 - Mary Rachel Stimson
- Rice Habanero Research group members
 - Vincent Cave
 - Shams Imam
- Guest lecturer
 - Prof. Melissa O'Neil
- Administrative assistant
 - Joyce Greene

