Habanero-Java: Multicore Programming for the Masses

PPoPP 2014 Tutorial

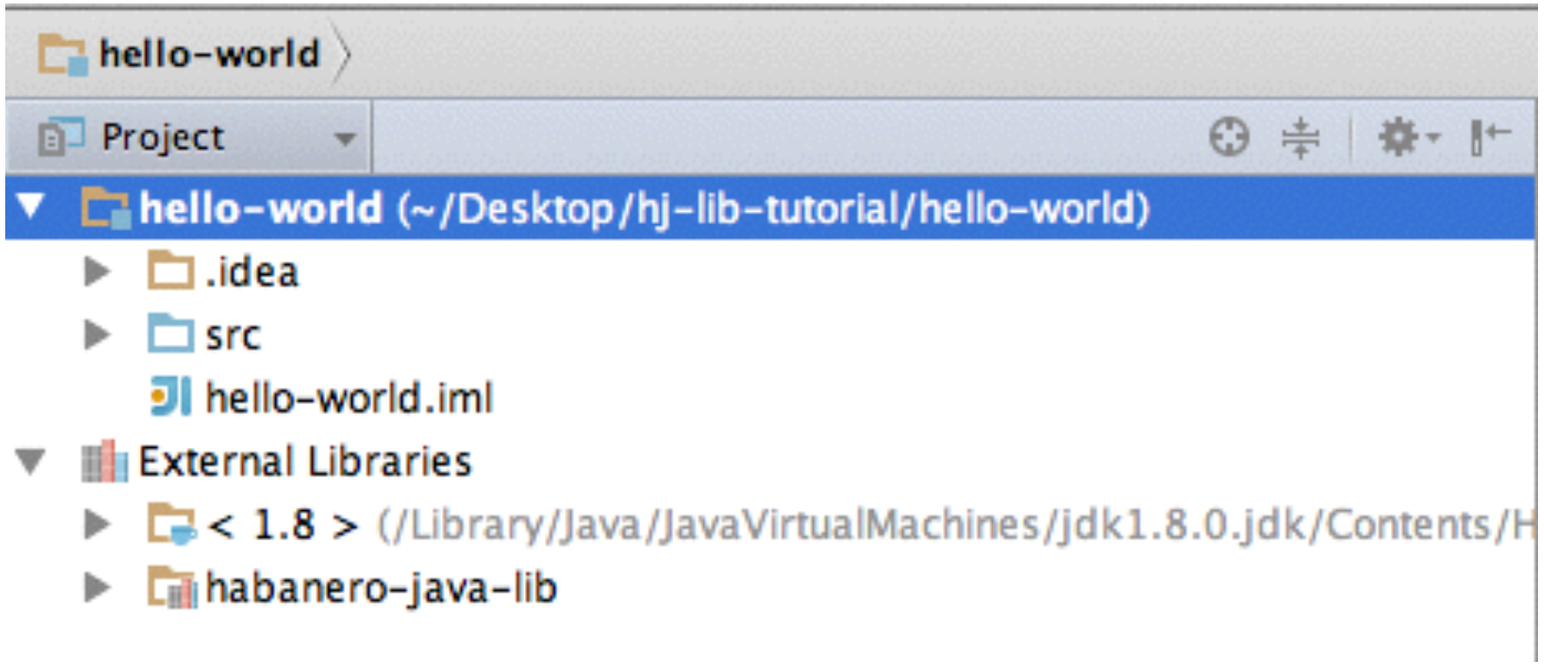
Shams Imam, Vivek Sarkar
Department of Computer Science, Rice University
{shams,vsarkar}@rice.edu

15 February 2014

Motivation: With Multicore Processors and Cloud Computing, all Computers are Parallel Computers!

Habanero-Java library

- Pure Java library without any other dependencies



- Uses Java 8 Lambda Expressions
  - Required for terse syntax
  - Early access release downloads of the Java 8 available
  - Target release of Java 8 is March 2014

Context: Rice Habanero Extreme Scale Software Project

Parallel Applications

Portable execution model

1) Lightweight asynchronous tasks and data transfers

- Creation: *async tasks, future tasks, data-driven tasks*
- Termination: *finish, future get, await*
- Data Transfers: *asyncPut, asyncGet, asyncISend, asyncIRecv*

2) Locality control for task and data distribution

- Task Distributions: *hierarchical places*
- Data Distributions: *hierarchical places, global name space*

3) Inter-task synchronization operations

- Mutual exclusion: *isolated, actors*
- Collective and point-to-point synchronization: *phasers*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime System

Two-level programming model

Declarative Coordination Language for Domain Experts:

CnC-HC, CnC-Java, CnC-Python, CnC-Matlab, … +

Task-Parallel Languages for Parallelism-aware Developers:

Habanero-C, Habanero-Java, Habanero-Scala

Extreme Scale Platforms

Outline

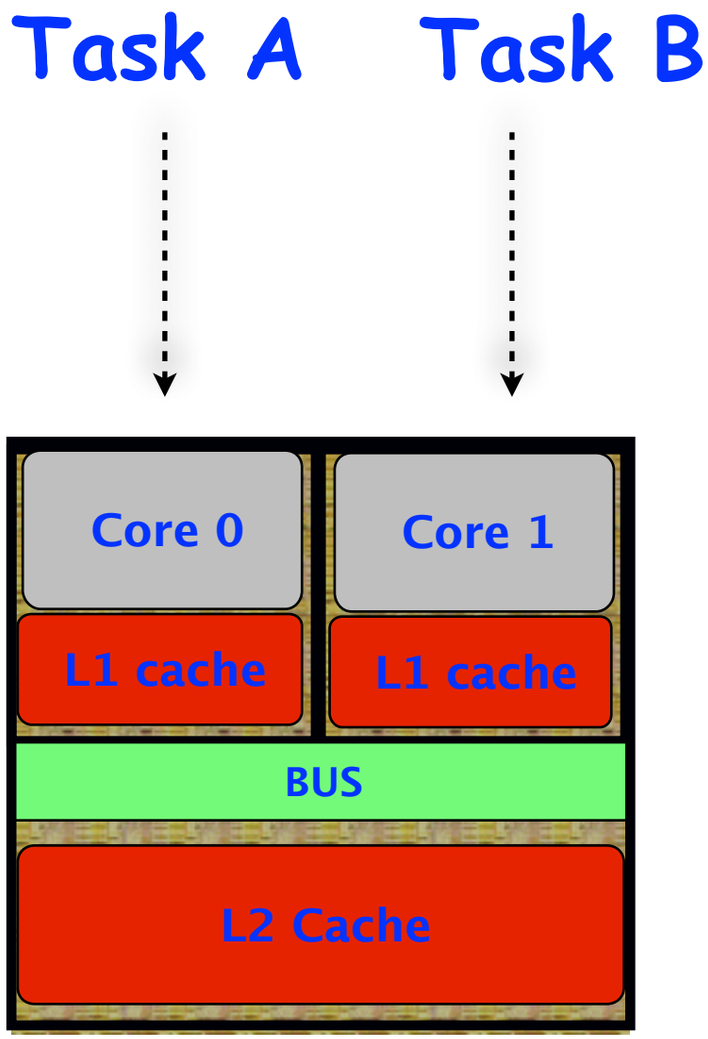
- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

What is Parallel Programming?

- Specification of operations that can be executed in parallel
- A parallel program is decomposed into sequential subcomputations called <u>tasks</u>
- Parallel programming constructs define task creation, termination, and interaction

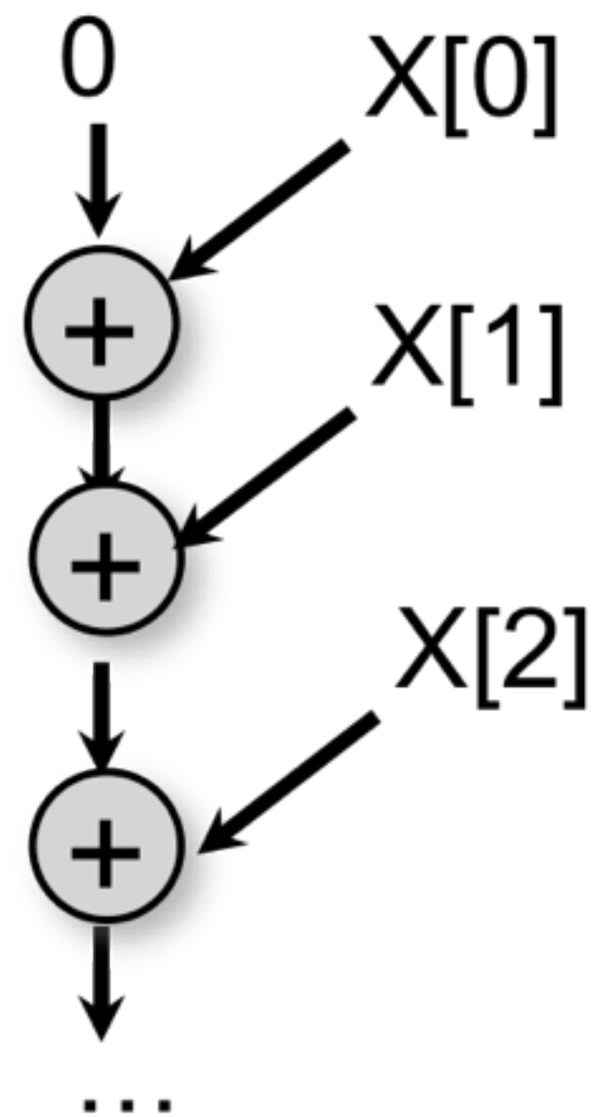


Schematic of a dual-core Processor

Example of a Sequential Program: Computing the sum of array elements

```
int sum = 0;
for (int i=0 ; i < X.length ; i++)
    sum += X[i];
```

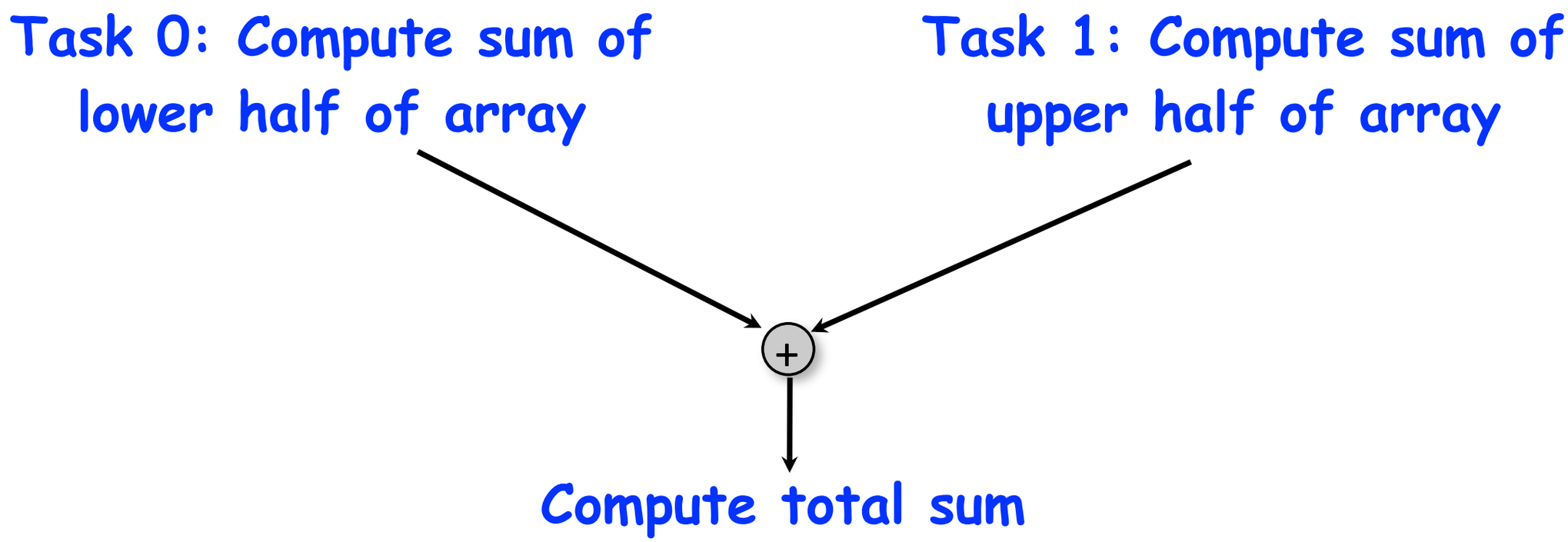
Computation Graph



Observations:

- The decision to sum up the elements from left to right was arbitrary
- The computation graph shows that all operations must be executed sequentially

Parallelization Strategy for two cores (Two-way Parallel Array Sum)



Basic idea:

- Decompose problem into two tasks for partial sums
- Combine results to obtain final answer
- Parallel divide-and-conquer pattern

Async and Finish Constructs for Task Creation and Termination

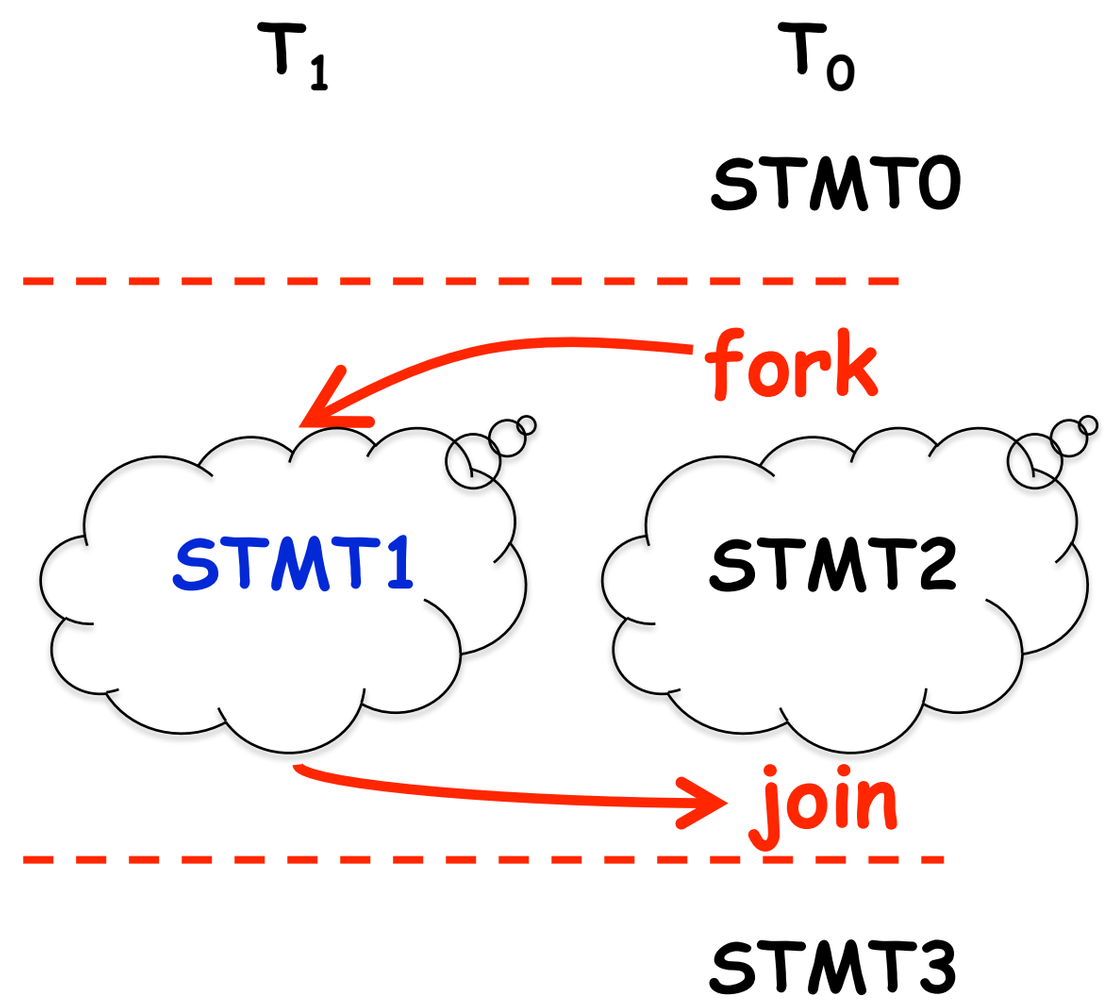
async S

- Creates a new child task that executes statement S

```
// 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
```

finish S

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



Acknowledgments: X10 and Habanero projects

Two-way Parallel Array Sum using HJ-Lib's finish & async API's

```
1.  // Start of Task T0 (main program)
2.  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3.  finish(() -> {
4.    async(() -> {
5.      // Child task computes sum of lower half of array
6.      for(int i=0; i < X.length/2; i++) sum1 += X[i];
7.    });
8.    // Parent task computes sum of upper half of array
9.    for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. });
11. // Parent task waits for child task to complete (join)
12. return sum1 + sum2;
```


Java 8 Lambda Expressions

- Behave like anonymous classes
  - lambda expressions can capture local variables
- Relies on Functional Interfaces
  - One abstract method.
  - Can omit the name of that method when you implement it.
- Example:

```
1. public interface Runnable {     void run();  }
2. void invoke(Runnable r) {
3.   r.run();
4. }
5. invoke(() -> {
6.   System.out.println("Inside Runnable.run!");
7. });
```

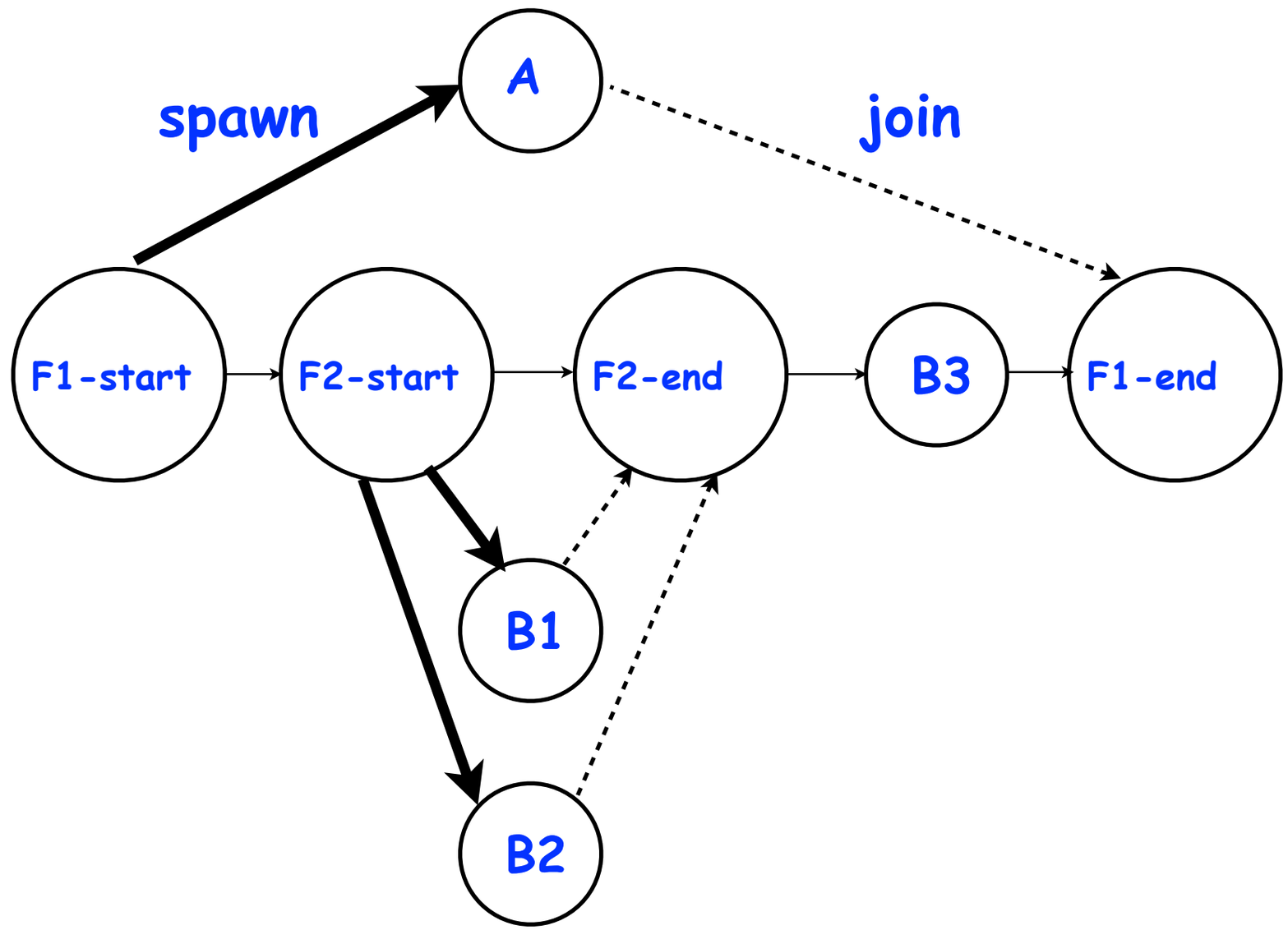

Computation Graphs

- A Computation Graph (CG) captures the dynamic execution of a parallel program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential subcomputation without any async, begin-finish and end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn" edges connect parent tasks to child async tasks
  - "Join" edges connect the end of each async task to its IEF's end-finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (dags)

Which statements can potentially be executed in parallel with each other?

```
1.  finish { // F1
2.    async A;
3.    finish { // F2
4.      async B1;
5.      async B2;
6.    } // F2
7.    B3;
8.  } // F1
```

Computation Graph



Complexity Measures for Computation Graphs

Define

- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the length of these paths (critical path length)
  - CPL(G) is also the smallest possible execution time for the computation graph

What is the critical path length of this parallel computation?

```
1.  finish { // F1
2.    async A; // Boil pasta
3.    finish { // F2
4.      async B1; // Chop veggies
5.      async B2; // Brown meat
6.    } // F2
7.    B3; // Make pasta sauce
8.  } // F1
```

Step B1

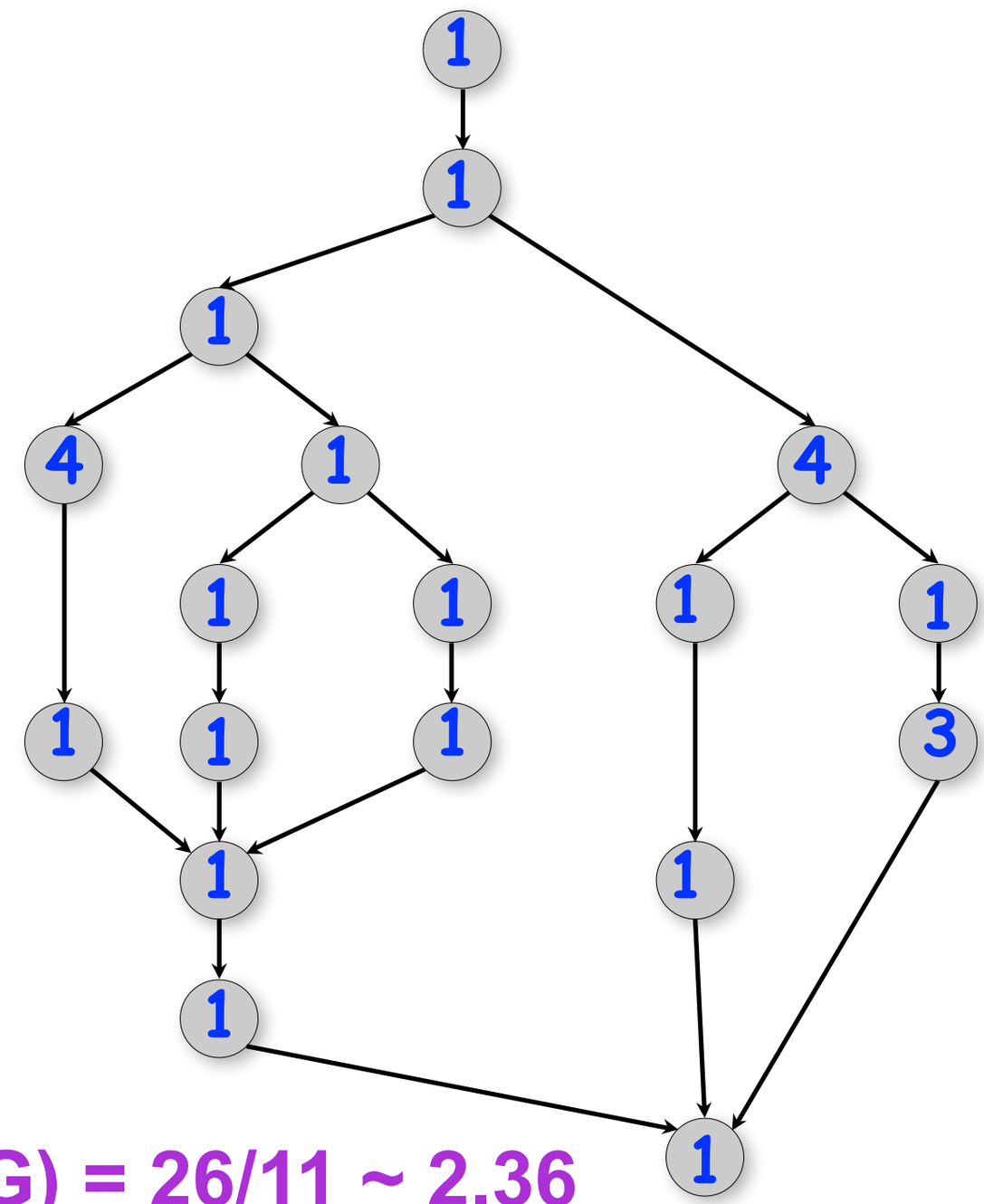
Step B2

Step B3

Step A

Ideal Parallelism

- Define **ideal parallelism** of Computation G Graph as the ratio, WORK(G)/CPL(G)
- Ideal Parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph



Example:

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36

HJ-Lib Abstract Performance Metrics

- Basic Idea
  - —Count operations of interest, as in big-O analysis
  - —Abstraction ignores overheads that occur on real systems
- Calls to doWork()
  - —Programmer inserts calls of the form, doWork(N), within a step to indicate abstraction execution of N application-specific abstract operations
    - – e.g., adds, compares, stencil ops, data structure ops
  - —Multiple calls add to the execution time of the step
- Enabling abstract metrics
  - `System.setProperty(HjSystemProperty.abstractMetrics.propertyKey(), "true");`
- If an HJ program is executed with this option, abstract metrics are printed at end of program execution with WORK(G), CPL(G), Ideal Speedup = WORK(G)/ CPL(G)

seq clause for async statements (pseudocode)

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

```
1. // Async task
2. async seq(size < thresholdSize) computeSum(X, lo,
   mid);
3.
4. // Future example
5. final future<int> sum1 = future seq(size <
   thresholdSize)
6.                                  { return computeSum(X, lo,
   mid); };
```

- **"seq" clause specifies condition under which async should be**

Parallel Quicksort using asyncSeq

```
protected static void quicksort(
   final Comparable[] A, final int M, final int N) {
        if (M < N) {
            // A point in HJ is an integer tuple
            HjPoint p = partition(A, M, N);
            int I = p.get(0);
            int J = p.get(1);
            asyncSeq(I - M <= 100, () -> quicksort(A, M, I));
            asyncSeq(N - J <= 100, () -> quicksort(A, J, N));
        }
    }
```


Sequential Algorithm for Matrix Multiplication

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

```
// 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
System.out.println(c[0][0]);
```

Parallelizing the loops in Matrix Multiplication example using finish & async

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

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


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

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

```
// Parallel version using finish & 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
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

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]) ;
}); });
```


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));
```


Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

Extending Async Tasks with Return Values

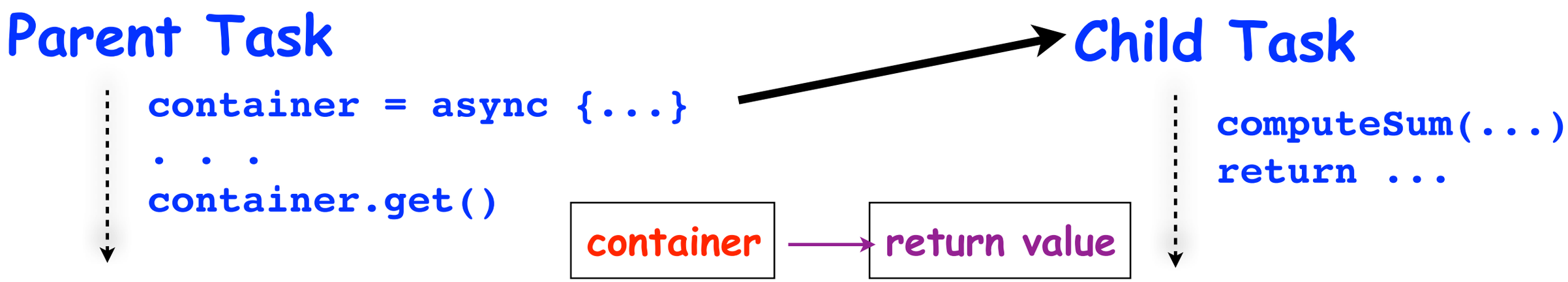
- Example Scenario in PseudoCode

```
// Parent task creates child async task
final future container =
  async { return computeSum(X, low, mid); };
. . .
// Later, parent examines the return value
int sum = container.get();
```

- Two issues to be addressed:

  1) Distinction between container and value in container (box)

  2) Synchronization to avoid race condition in container accesses



HJ Futures: Tasks with Return Values

`async { Stmt-Block }`

- Creates a new child task that executes Stmt-Block, which must terminate with a return statement and return value
- Async expression returns a reference to a container of type future

`Expr.get()`

- Evaluates Expr, and blocks if Expr's value is unavailable
- 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 in HJ-Lib

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final HjFuture sum1 = future (() -> { // Future Task T2
  int sum = 0;
  for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
  return sum;
});
final HjFuture sum2 = future (() ->{ // Future Task T3
  int sum = 0;
  for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
  return sum;
});
//Task T1 waits for Tasks T2 and T3 to complete
int total = sum1.get() + sum2.get();
```

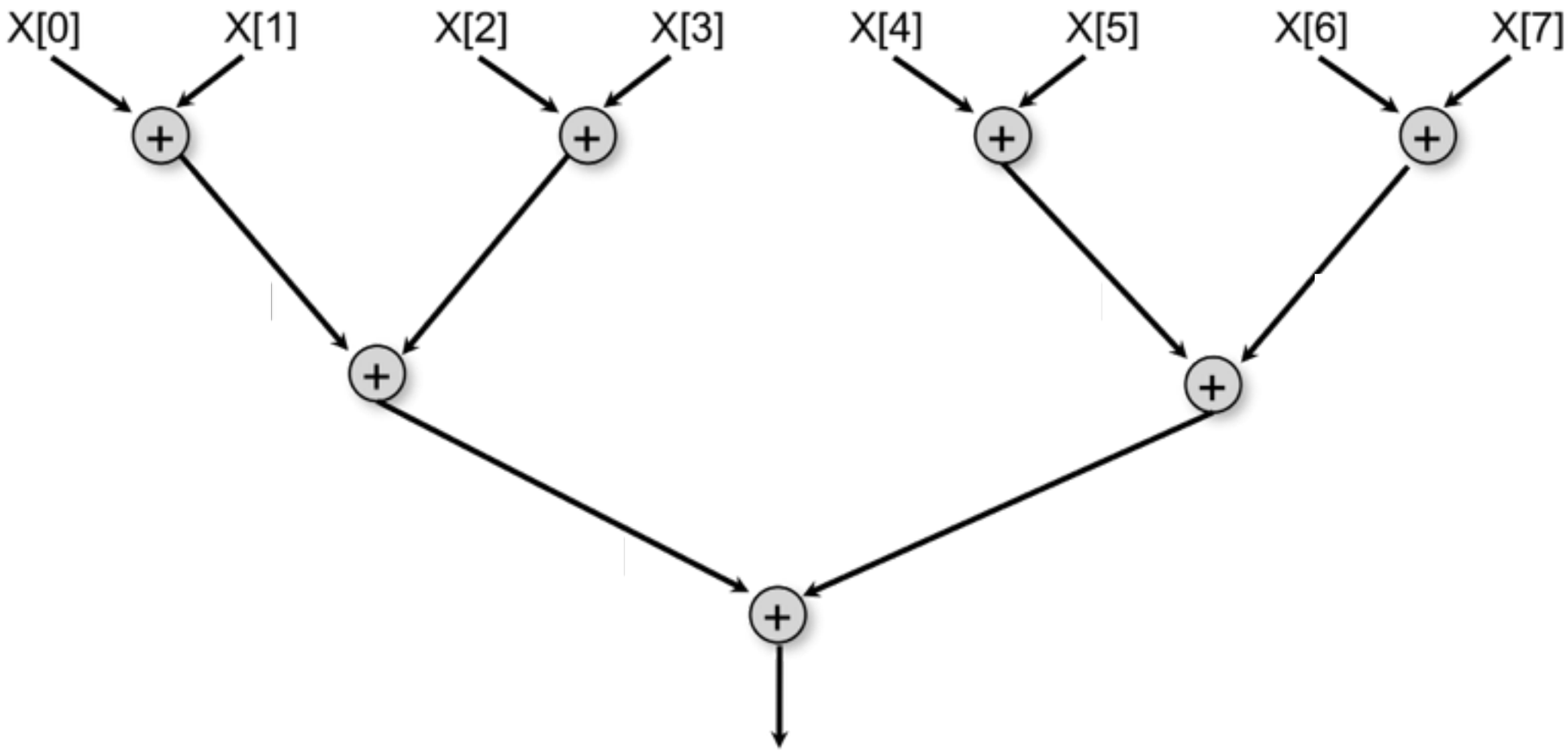

Future Task Declarations and Uses

- Variable of type future is a reference to a future object
  - —Container for return value from future task
  - —The reference to the container is also known as a "handle"
- Two operations that can be performed on variable V of type future:
  - — Assignment: V1 can be assigned value of type future
  - — Blocking read: V1.get() waits until the future task referred to by V1 has completed, and then propagates the return value

Comparison of Future Task and Regular Async Versions of Two-Way Array Sum

- Future task version initializes two references to future objects, sum1 and sum2, and both are declared as final
- No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1.get() and sum2.get()
- Easier to guarantee absence of race conditions in Future Task version
  - No race on sum because it is a local variable in tasks T2 and T3
  - No race on future variables, sum1 and sum2, because of blocking-read semantics

Reduction Tree Schema for computing Array Sum in parallel (beyond two-way parallelism)?



Question:

- **How can we implement this schema using future tasks?**

Array Sum using Future Tasks (Seq version)

Recursive divide-and-conquer pattern

```
1.  static int computeSum(int[] X, int lo, int hi) {
2.    if ( lo > hi ) return 0;
3.    else if ( lo == hi ) return X[lo];
4.    else {
5.      int mid = (lo+hi)/2;
        final sum1 = computeSum(X, lo, mid);
6.      final sum2 = computeSum(X, mid+1, hi);
7.     // Parent now waits for the container values
8.      return sum1 + sum2;
9.    }
10.  } // computeSum
11. int sum = computeSum(X, 0, X.length-1); // main program
```


Array Sum using Future Tasks (two futures per method call)

Recursive divide-and-conquer pattern

```
1.  static int computeSum(int[] X, int lo, int hi) {
2.    if ( lo > hi ) return 0;
3.    else if ( lo == hi ) return X[lo];
4.    else {
5.      int mid = (lo+hi)/2;
        final HjFuture sum1 =
6.            future(() -> { return computeSum(X, lo, mid); });
7.      final HjFuture sum2 =
8.            future(() -> { return computeSum(X, mid+1, hi); });
9.     // Parent now waits for the container values
10.     return sum1.get() + sum2.get();
11.    }
12.  } // computeSum
13. int sum = computeSum(X, 0, X.length-1); // main program
```


Exercise: Why must Future References be declared as final?

Consider the pseudocode on the right with futures declared as non-final static fields. Is there a possible execution in which a deadlock situation may occur between tasks T1 and T2 with this code (with each task waiting on the other due to get() operations)? Explain why or why not.

Yes, a deadlock can occur when future f1 does f2.get() and future f2 does f1.get().

```
static HjFuture f1=null;
static HjFuture f2=null;

void main(String[] args) {
  f1 = future(() -> {return a1();});
  f2 = future(() -> {return a2();});
}

int a1() { // Task T1
  while (f2 == null); // spin loop
  return f2.get(); //T1 waits for T2
}

int a2() { // Task T2
  while (f1 == null); // spin loop
  return f1.get(); //T2 waits for T1
}
```

deadlock

WARNING: such "spin" loops are an example of bad parallel programming practice in application code. Their semantics depends on the "memory model". In the Java memory model, there's no guarantee that the above spin loops will ever terminate.

Why should Future References be declared as final (contd)?

Now consider a modified version of the above code in which futures are declared as final local variables (which is permitted in HJ). Can you add get() operations to methods a1() and a2() to create a deadlock between tasks T1 and T2 with this code? Explain why or why not.

No, the final declarations make it impossible for future f1's task (T1) to receive a reference to f2.

Will your answer be different if f1 and f2 are final fields in objects or final static fields?

No.

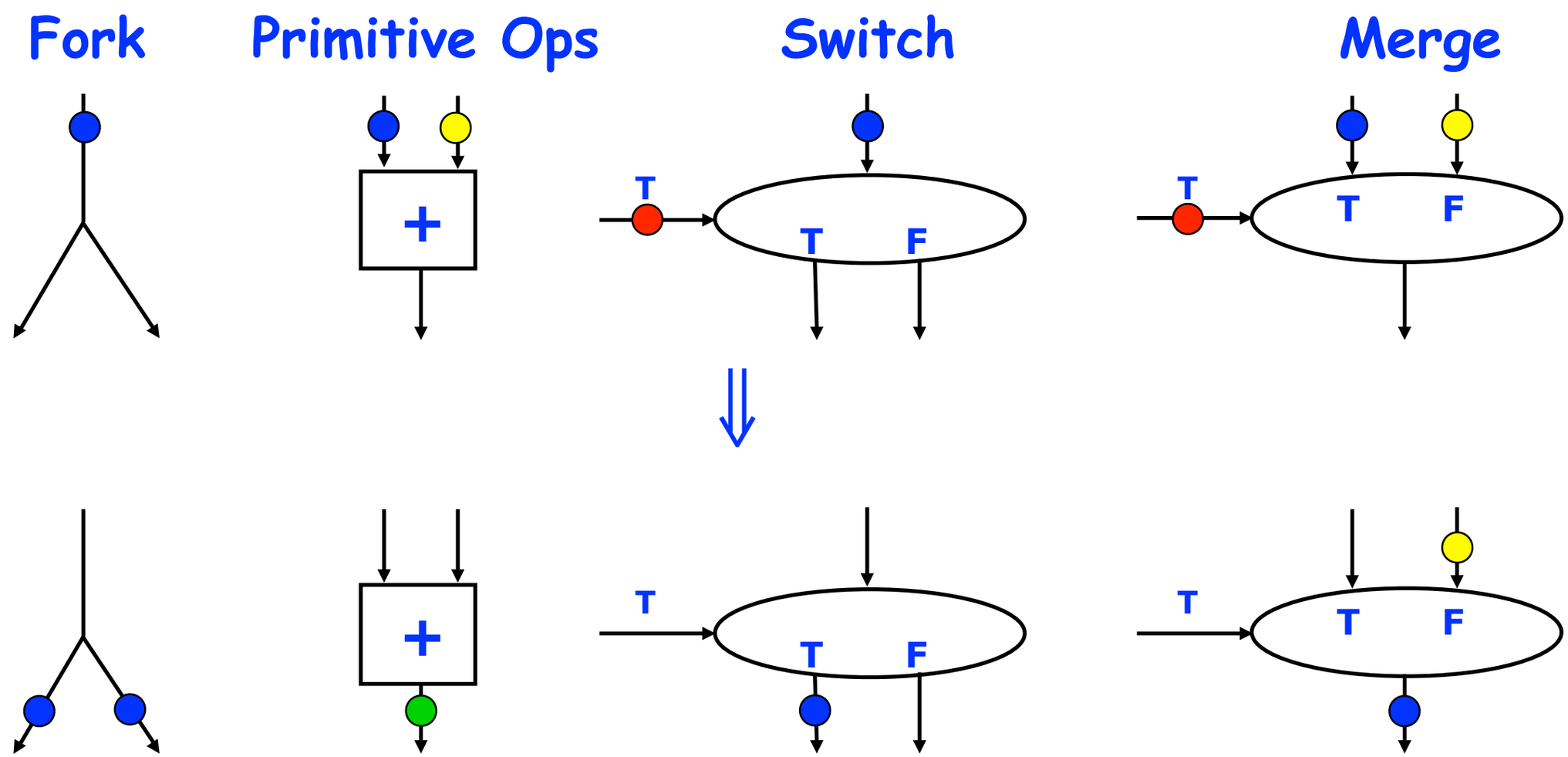
```
final future f1 =
    async {return a1();};
 final future f2 =
    async {return a2(f1);};
}

int a1() {
// Task T1 cannot receive a
// reference to f2
 . . .
}

int a2(HjFuture f1) {
// Task T2 can refer
// to f1 but that won't cause
// a deadlock.
... f1.get() ...
}
```


Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

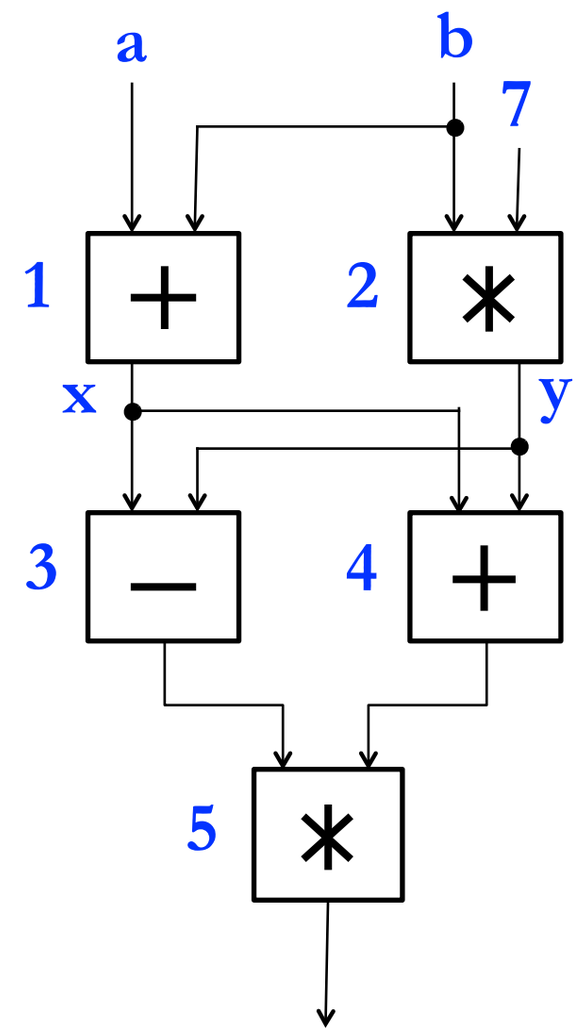
Dataflow Computing

- **Original idea: replace machine instructions by a small set of dataflow operators**



Example instruction sequence and its dataflow graph

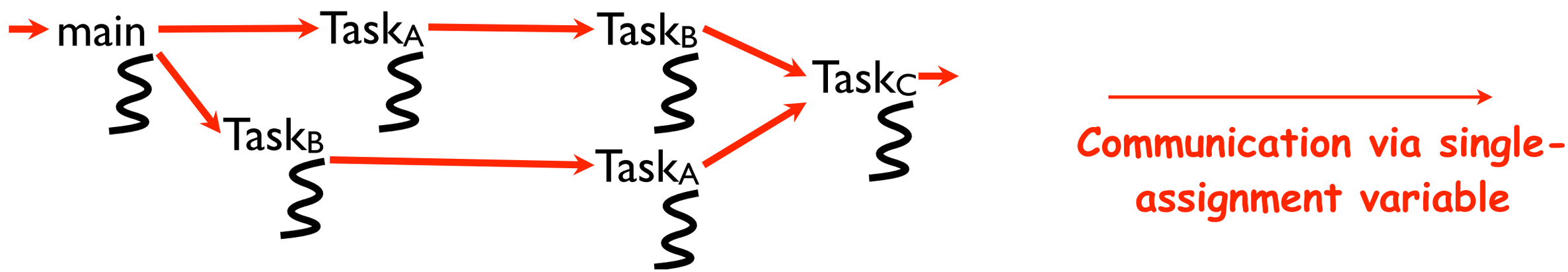
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.

No separate control flow

Macro-Dataflow Programming



- "Macro-dataflow" = extension of dataflow model from instruction-level to task-level operations
- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
  - Static dataflow ==> graph fixed when program execution starts
  - Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
  - Deadlocks are possible due to unavailable inputs (but they are deterministic)

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

```
HjDataDrivenFuture<T1> ddfA = newDataDrivenFuture();
```

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

```
asyncAwait(ddfA, ddfB, …, () -> Stmt);
```

- Create a new <u>data-driven-task</u> 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)

Implementing Future Tasks using DDFs

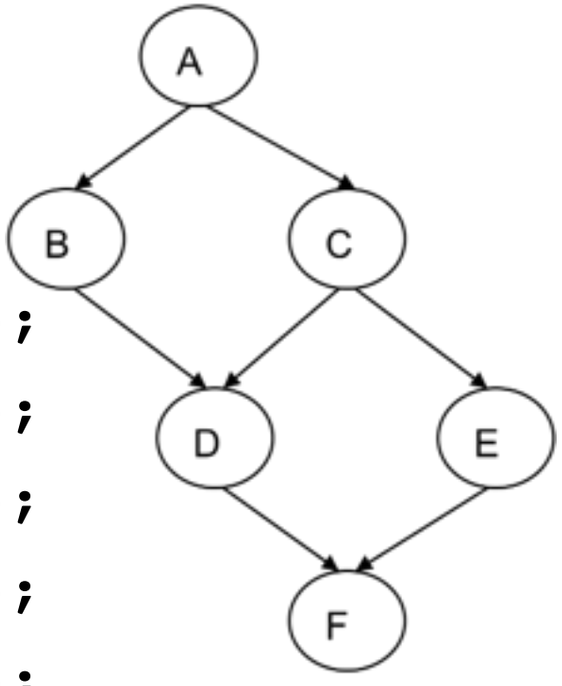
- **Future version**

```
1. final HjFuture<T> f = future(() -> { return g(); });
2. S1
3. async(() -> {
4.    ... = f.get();
5.    S2;
6.    S3;
7. });
```

- **DDF version**

```
1. HjDataDrivenFuture<T> f = newDataDrivenFuture();
2. async(() -> { f.put(g()) });
3. S1
4. asyncAwait(f, () -> {
5.    ... = f.get();
6.    S2;
7.    S3;
8. });
```


Use of DDFs with dummy objects (like future<void>)




```
1. finish(() -> {
2.   HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
3.   HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
4.   HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
5.   HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
6.   HjDataDrivenFuture<Void> ddfE = newDataDrivenFuture();
7.   async(() -> { ... ; ddfA.put(null); }); // Task A
8.   asyncAwait(ddfA, () -> { ... ;  ddfB.put(null); }); // Task B
9.   asyncAwait(ddfA, () -> { ... ;  ddfC.put(null); }); // Task C
10.  asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(null); }); // Task D
11.  asyncAwait(ddfC, () -> { ... ;  ddfE.put(null); }); // Task E
12.  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
13. }); // finish
```

- This example uses an empty string as a dummy object

Differences between Futures and DDFs/DDTs

- Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all DDFs are available
- Future tasks cannot deadlock, but it is possible for a DDT to block indefinitely ("deadlock") if one of its input DDFs never becomes available
- DDTs and DDFs are more general than futures
  - —Producer task can only write to a single future object, where as a DDT can write to multiple DDF objects
  - —The choice of which future object to write to is tied to a future task at creation time, where as the choice of output DDF can be deferred to any point with a DDT
- DDTs and DDFs can be more implemented more efficiently than futures
  - —An "asyncAwait" statement does not block the worker, unlike a future.get()
  - —You will never see the following message with "asyncAwait"
    - "ERROR: Maximum number of hj threads per place reached"

Two Exception (error) cases for DDFs that do not occur in futures

- <u>Case 1:</u> If two put's are attempted on the same DDF, an exception is thrown because of the violation of the single-assignment rule
  - —There can be at most one value provided for a future object (since it comes from the producer task's return statement)

- <u>Case 2:</u> If a get is attempted by a task on a DDF that was not in the task's await list, then an exception is thrown because DDF's do not support blocking gets
  - —Futures support blocking gets

Deadlock example with DDTs

```
1. HjDataDrivenFuture left = newDataDrivenFuture();
2. HjDataDrivenFuture right = newDataDrivenFuture();
3. finish(() -> {
4.   asyncAwait(left, () -> {
5.       right.put(rightWriter()); });
6.   asyncAwait(right, () -> {
7.       left.put(leftWriter()); });
8. });
```

- **HJ-Lib has deadlock detection mode**
- **Enabled using:**
  - `System.setProperty(HjSystemProperty.trackDeadlocks.propertyKey(), “true");`
  - Reports an edu.rice.hj.runtime.util.DeadlockException when deadlock detected

Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

Parallel Programming Challenges

- **Correctness**
  - —New classes of bugs can arise in parallel programming, relative to sequential programming
    - – Data races, deadlock, nondeterminism
- **Performance**
  - —Performance of parallel program depends on underlying parallel system
    - – Language compiler and runtime system
    - – Processor structure and memory hierarchy
    - – Degree of parallelism in program vs. hardware
- **Portability**
  - —A buggy program that runs correctly on one system may not run correctly on another (or even when re-executed on the same system)
  - —A parallel program that performs well on one system may perform poorly on another

What happens if we forget a finish?

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

Data race between accesses of sum1 in async and in main program

Formal Definition of Data Races

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. (L must be a shared location i.e., a static field, instance field, or array element.)**

- **A program is *data-race-free* it cannot exhibit a data race for any input**
- **Above definition includes all "potential" data races i.e., it's considered a data race even if S1 and S2 execute on the same processor**

Four Observations related to Data Races

1. <u>Immutability property</u>: there cannot be a data race on shared immutable data.
   - A location, L, is immutable if it is only written during initialization, and can only be read after initialization. In this case, no read can potentially execute in parallel with the write.

- Parallel programming tip: use immutable objects and arrays to avoid data races
  - Will require making copies of objects and arrays for updates
  - Copying overhead may be prohibitive in some cases, but acceptable in others
  - NOTE: future values are also immutable

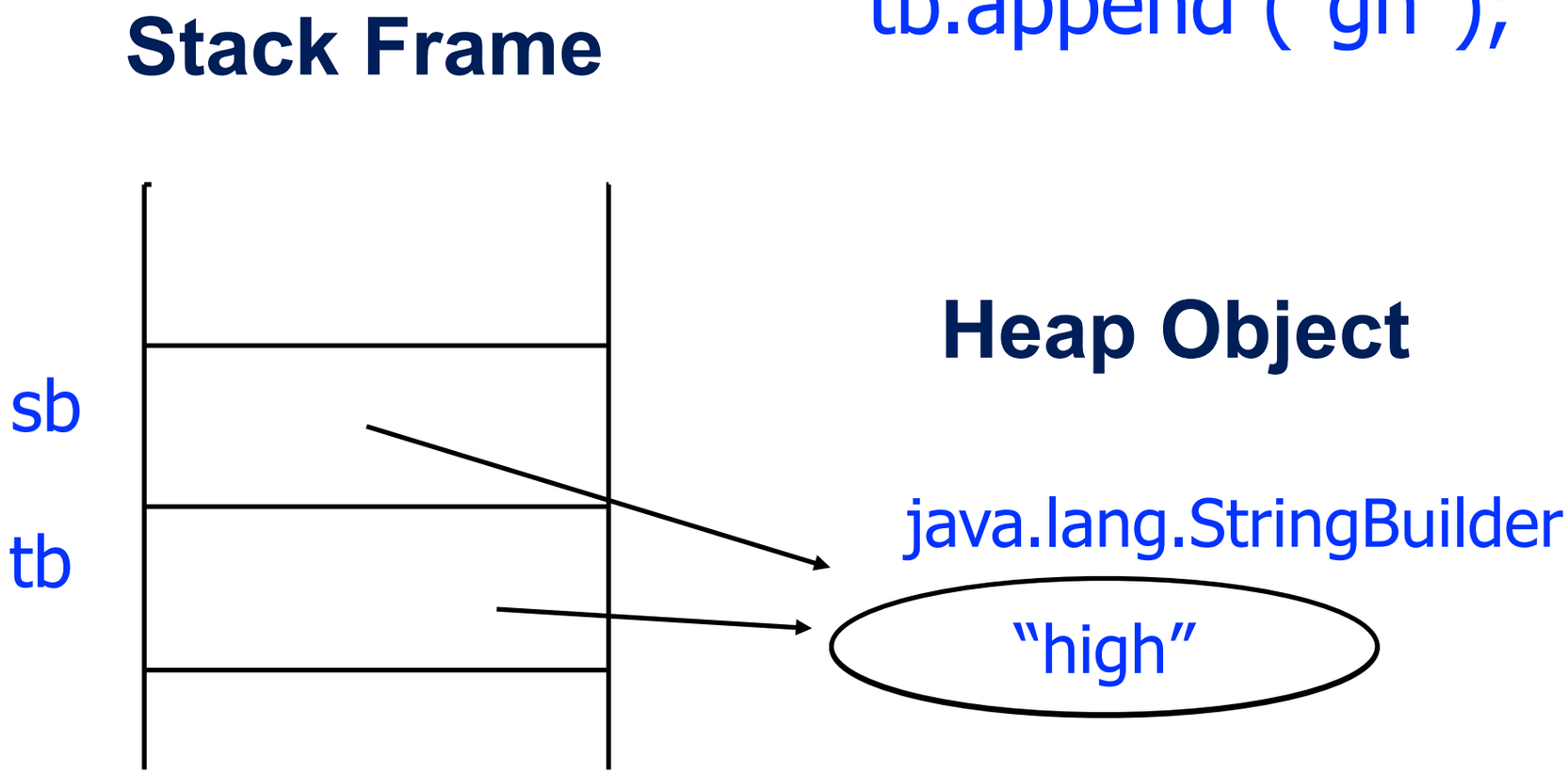
- Example with java.lang.String

```
1. finish(() -> {
2.    String s1 = "XYZ";
3.    async(() -> { String s2 = s1.toLowerCase(); ... });
4.    System.out.println(s1);
5. });
```


Example of a Mutable Object

- **If an object is modified, all references to the object see the new value**

```
StringBuilder sb = new ("hi");
StringBuilder tb = sb;
tb.append ("gh");
```



Observations

2. <u>Single-task ownership property</u>: there cannot be a data race on a location that is only read or written by a single task.
   - Define: step S in computation graph CG "owns" location L if S performs a read or write access on L. If step S belongs to Task T, we can also say that Task T owns L when executing S.
   - Consider a location L that is only owned by steps that belong to the same task, T. Since all steps in Task T must be connected by *continue* edges in CG, all reads and writes to L must be ordered by the dependences in CG. Therefore, no data race is possible on location L.

- Parallel programming tip: if an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.
  - Will require making copies when sharing the object or array with other tasks.

Example of Single-task ownership with Copying

- **If an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.**
  - Entails making copies when sharing the object with other tasks.
  - As with Immutability, copying overhead may be prohibitive in some cases, but acceptable in others.
- **Example**

```
1.finish(() -> { // Task T1 owns A
2.   int[] A = new int[n]; // ... initialize array A ...
3.   // create a copy of array A in B
4.   int[] B = new int[A.length];
5.   System.arraycopy(A, 0, B, 0, A.length);
6.   async(() -> { // Task T2 owns B
7.     int sum = computeSum(B,0,B.length-1);// Modifies B as in ArraySum1
8.     System.out.println("sum = " + sum);
9.   });
10.  // ... update Array A ...
11.  System.out.println(Arrays.toString(A)); //printed by task T1
12.});
```


Observations (contd)

3. <u>Ownership-transfer property:</u> there cannot be a data race on a location if all steps that read or write it are totally ordered in CG (i.e., if the steps belong to a single directed path)
   - Think of the ownership of L being ``transferred'' from one step to another, even across task boundaries, as execution follows the path of dependence edges in the total order.

- Parallel programming tip:
  - If an object or array needs to be written multiple times after initialization and also accessed by multiple tasks, then try and ensure that all the steps that read or write a location L in the object/array are totally ordered by dependences in CG.
    - Ownership transfer is even necessary to support single-task ownership. In the previous example, since Task T1 initializes array B as a copy of array A, T1 is the original owner of A. The ownership of B is then transferred from T1 to T2 when Task T2 is created.

Observations (contd)

4. <u>Local-variable ownership property:</u> there cannot be a data race on a local variable.
   - If L is a local variable, it can only be written by the task in which it is declared (L's owner). The "implicitly final" semantics for accessing outer local variables ensures that there is no race condition between the read access in the child task and the write access in L's owner (parent task).

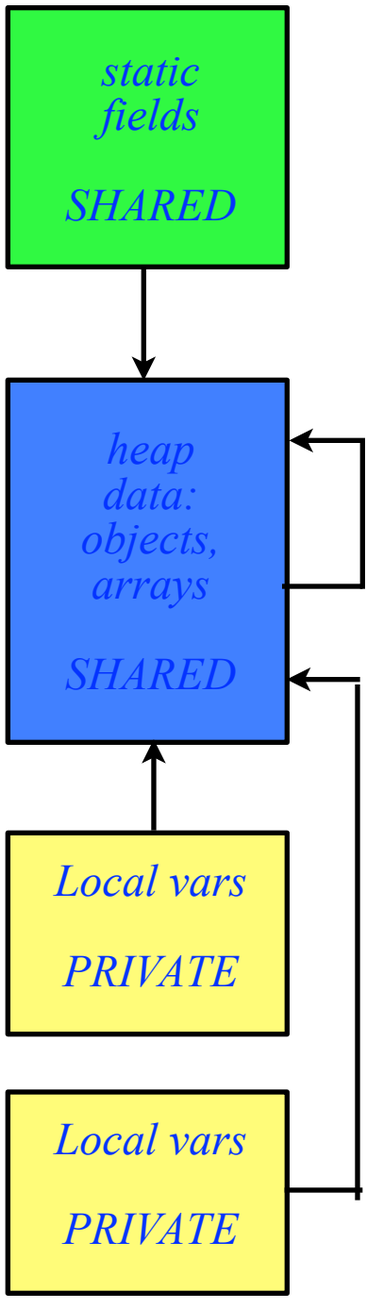
- Parallel programming tip:
  - You do not need to worry about data races on local variables, since they are not possible. However, local variables in Java are restricted to contain primitive data types (such as int) and references to objects and arrays. In the case of object/array references, be aware that there may be a data race on the underlying object even if there is no data race on the local variable that refers to (points to) the object.

Recap of Java's Storage Model

Java's storage model contains three memory regions:

1. Static Data: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as static fields.
   - Static fields can be shared among threads/tasks
2. Heap Data: region of memory for dynamically allocated objects and arrays (created by "new").
   - Heap data can be shared among threads/tasks
3. Stack Data: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its local variables
   - Local variables are private to a given thread/task

All references (pointers) must point to heap data --- no references can point to static or stack data



. . .

Functional vs. Structural Determinism

- A parallel program is said to be *functionally deterministic* if it always computes the same answer when given the same input
- A parallel program is said to be *structurally deterministic* if it always produces the same computation graph when given the same input
- Race-Free Determinism
  - If a parallel program is written using the constructs learned so far (finish, async, futures) and is known to be race-free, *then it must be both functionally deterministic and structurally deterministic*

String Search Problem

- **Inputs**
  - —text: a long string with N characters to search in
  - —pattern: a short string of M characters to search for
- **Output**
  - —Existence of an occurrence (boolean value)
- **Example**
  - —text: "abacadabrabracabracadababacadabrabracabracadabrabrabr"
  - —pattern: aca
  - —output: true (pattern found)
- **Applications**
  - —Word processing, virus scans, information retrieval, computational biology, web search engines, ...
- **Variations**
  - —Count of occurrences, index of any occurrence, indices of all occurrences

Brute Force Sequential Algorithm for String Search

```
public static boolean search(char[] pattern, char[] text) {
    int M = pattern.length; int N = text.length;
    boolean found = false;
    for (int i = 0; i <= N - M; i++) {
      int j; // search for pattern starting at text[i]
      for (j = 0; j < M; j++) {
        // Count each char comparison as 1 unit of work
        if (text[i+j] != pattern[j]) break;
      } // for (j = ... )
      if (j == M) found = true; // found at offset i
    }
    return found;
}
```

Let us now consider different parallel versions of this algorithm

V1: Functional + Structural Determinism (No data race)

```
1.  // Count all occurrences
2.  final FinishAccumulator ac =
3.          newFinishAccumulator(Operator.SUM, int.class);
4.  finish(ac, ()-> {
5.    for (int i = 0; i <= N - M; i++) {
6.      async(() -> {
7.        int j;
8.        for (j = 0; j < M; j++) {
9.          if (text[i+j] != pattern[j])
10.           break;
11.       }
12.       if (j == M)
13.         ac.put(1); // found
14.     });
15.   }
16. });
17. print a.get();
```


V2: Functional + Structural Determinism (Benign data race)

```
1.  // Existence of an occurrence
2.  boolean[] found = {false};
3.  finish(() -> {
4.    for (int i = 0; i <= N - M; i++) {
5.      async(() -> {
6.        for (j = 0; j < M; j++)
7.          if (text[i+j] != pattern[j])
8.            break;
9.        if (j == M)
10.         found[0] = true;
11.     });
12.   }
13. });
14. print found[0]
```


V3: Functional Nondeterminism + Structural Determinism

```
// Index of an occurrence
static int index = -1; // static field
. . .
finish(() -> {
  for (int i = 0; i <= N - M; i++)
    async(() -> {
      for (j = 0; j < M; j++)
        if (text[i+j] != pattern[j])
          break;
      if (j == M)
        index = i; // found at i
    });
});
```


V4: Functionally Deterministic + Structurally Nondeterministic

```
1.  static boolean found = false; //static field
2.  . . .
3.  finish(() -> {
4.    for (int i = 0; i <= N - M; i++) {
5.      if (found)
6.        break; // Eureka!
7.      async(() -> {
8.        for (j = 0; j < M; j++)
9.          if (text[i+j] != pattern[j])
10.           break;
11.       if (j == M)
12.         found = true;
13.     }); // async
14.   }
15. }); // finish-for
```


V5: Functionally Nondeterministic + Structurally Nondeterministic

```
static int index = -1; // static field
. . .
finish(() -> {
  for (int i = 0; i <= N - M; i++) {
    if (found)
      break; // Eureka!
    async(() -> {
      for (j = 0; j < M; j++)
        if (text[i+j] != pattern[j])
          break;
      if (j == M)
        index = i;
    }); // async
  }
}); // finish-for
```


A Classification of Parallel Programs

Data Race Free?	Functionally Deterministic?	Structurally Deterministic?	Example: String Search variation
Yes	Yes	Yes	Count of all occurrences
No	Yes	Yes	Existence of an occurrence
No	No	Yes	Index of any occurrence
No	Yes	No	"Eureka" extension for existence of an occurrence: do not create more async tasks after occurrence is found
No	No	No	"Eureka" extension for index of an occurrence: do not create more async tasks after occurrence is found

Data-Race-Free Determinism Property implies that it is not possible to write an HJ program with Yes in column 1, and No in column 2 or column 3 (when only using Module 1 constructs)

Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

Formal Definition of Data Races (Recap)

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.**

However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations without incurring data races

- **Special cases with determinism guarantees: finish accumulators, phaser accumulators**
- **How should conflicting accesses be handled in general, when outcome may be nondeterministic?**

Example of two tasks performing conflicting accesses --- need for "mutual exclusion"

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   { // start of desired mutual exclusion region
     this.prev.next = this.next;
     this.next.prev = this.prev;
   } // end of desired mutual exclusion region
   . . . // remaining code that doesn't need mutual exclusion
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); }); // conflicts with previous async
 });
}
```


How to enforce mutual exclusion?

- **The predominant approach to ensure mutual exclusion proposed many years ago is to enclose the code region in a critical section.**
  - **—"In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore."**

— Source: http://en.wikipedia.org/wiki/Critical_section

isolated construct

```
isolated(() -> { <body> });
```

- Isolated statement identifies a critical section
- Two tasks executing isolated statements must perform them in mutual exclusion
  - ➔Isolation guarantee applies to (isolated, isolated) pairs of statement instances, not to (isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested
  - — An inner isolated statement is redundant
- Parallel constructs should be avoided inside isolated statements
  - —Isolated statements must not contain any other parallel statement that performs a blocking operation: finish, future get, next, asyncAwait, etc.
  - —Non-blocking async operations are permitted, but isolation guarantee only applies to creation of async, not to its execution
- Isolated statements can never cause a deadlock
  - — Other techniques used to enforce mutual exclusion (e.g., locks) can lead to a deadlock, if used incorrectly

Use of isolated to fix previous example with conflicting accesses

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(() -> { // start of desired mutual exclusion region
     this.prev.next = this.next;
     this.next.prev = this.prev;
   } // end of desired mutual exclusion region
   . . . // remaining code that doesn't need mutual exclusion
 });
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); }); // conflicts with previous async
 });
}
```


Parallel Spanning Tree Algorithm using isolated statement

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    return isolatedWithReturn(() -> {
      if (parent == null) parent=n;
      return parent == n; // return true for success
    });
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this)) {
        async(() -> { child.compute(); }); //escaping async
      }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
. . .
```

Example graph (root=1, spanning tree edge shown as arrow from child to parent)

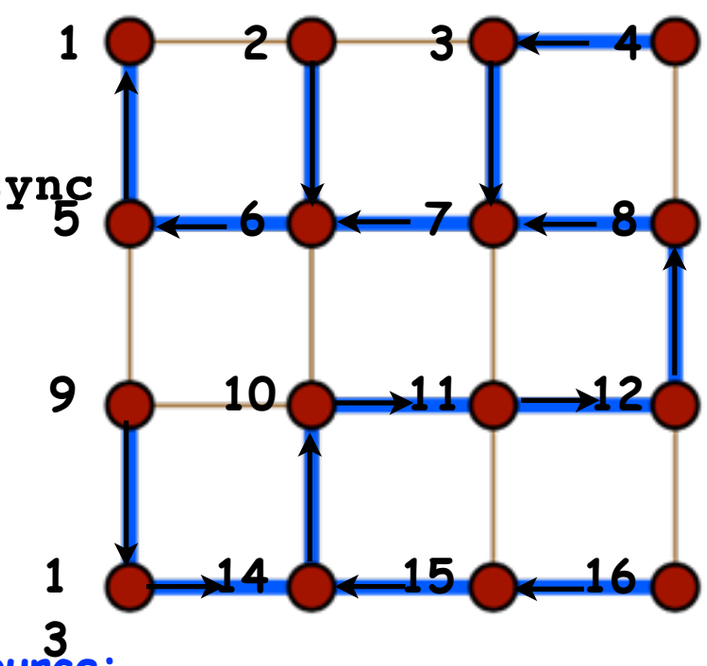


Figure source: http://en.wikipedia.org/wiki/Spanning_tree

Exercise: 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.

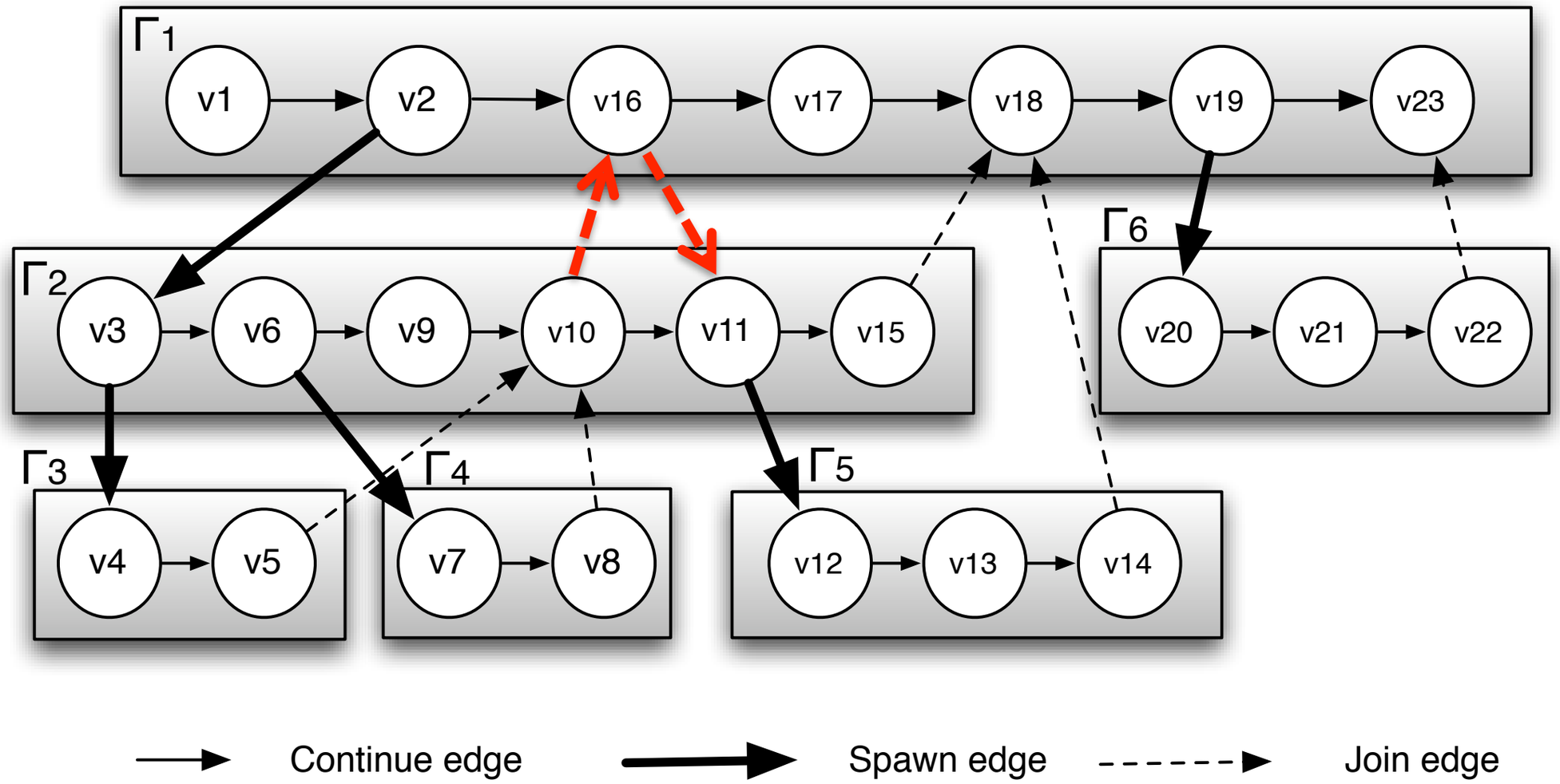
```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async(() -> { print r.nextSeed(); … });
  async(() ->{ print r.nextSeed(); … });
} // main()
```

```
class IsolatedPRNG {
  private int seed;
  public int nextSeed() {
    int retVal;
    retVal = seed;
    seed = nextInt(retVal);
    return retVal;
  } // nextSeed()
  . . .
} // IsolatedPRNG
```


Serialized Computation Graph for Isolated Statements

- **Model each instance of an isolated statement as a distinct step (node) in the CG.**
- **Need to reason about the *order* in which interfering isolated statements are executed**
  - —Complicated because the order of isolated statements may vary from execution to execution
- **Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated statements.**
  - —SCG consists of a CG with additional serialization edges.
  - —Each time an isolated step, S′, is executed, we add a serialization edge from S to S′ for each prior "interfering" isolated step, S
    - Two isolated statements always interfere with each other
    - Interference of "object-based isolated" statements depends on intersection of object sets
    - Serialization edge is not needed if S and S' are already ordered in CG
  - —An SCG represents a set of executions in which all interfering isolated statements execute in the same order.

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order



Serialization edge

v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs

- **Need to consider all possible orderings of interfering isolated statements to establish data race freedom**

Object-based isolation in HJ

```
isolated(obj1, obj2, …, () -> <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
- Example:
  - `isolated(a, b, () -> {})` and `isolated(c, d, () -> {})` can execute in parallel
  - `isolated(a, b, () -> {})` and `isolated(b, c, () -> {})` cannot execute in parallel

DoublyLinkedListNode Example revisited with Object-Based Isolation

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(this.prev, this, this.next, () -> { // object-based isolation
     this.prev.next = this.next;
     this.next.prev = this.prev;
   }
   . . .
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); });
 });
}
```


Read-Write Object-based isolation in HJ

```
isolated(readMode(obj1),
         writeMode(obj2), …,
         () -> <body> );
```

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Not specifying a mode is the same as specifying a write mode
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists
- Distinguish between read/write accesses for further parallelism

SortedLinkedList Example
Read-Write Object-Based Isolation

```
public boolean contains(Object object) {
  return isolatedWithReturn( readMode(this), () -> {
    Entry pred, curr;
    ...
    return (key == curr.key);
  });
}

public int add(Object object) {
  return isolatedWithReturn( writeMode(this), () -> {
    Entry pred, curr;
    ...
    if (...) return 1; else return 0;
  });
}
```


java.util.concurrent.AtomicInteger methods and their equivalent isolated statements

`j.u.c.atomic` Class and Constructors	`j.u.c.atomic` Methods	Equivalent HJ `isolated` statements
AtomicInteger	int j = v.**get**();	int j; `isolated` (v) j = v.val;
	v.**set**(newVal);	`isolated` (v) v.val = newVal;
AtomicInteger()	int j = v.**getAndSet**(newVal);	int j; `isolated` (v) { j = v.val; v.val = newVal; }
// init = 0	int j = v.**addAndGet**(delta);	`isolated` (v) { v.val += delta; j = v.val; }
	int j = v.**getAndAdd**(delta);	`isolated` (v) { j = v.val; v.val += delta; }
AtomicInteger(init)	boolean b = v.**compareAndSet** (expect,update);	boolean b; `isolated` (v) if (v.val==expect) {v.val=update; b=true;} else b = false;

Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements.

- **Variable v refers to an AtomicInteger object in column 2 and to a standard non-atomic Java object in column 3.**
- **val refers to a field of type int.**

java.util.concurrent. AtomicReference methods and their equivalent isolated statements

`j.u.c.atomic` Class and Constructors	`j.u.c.atomic` Methods	Equivalent HJ `isolated` statements
AtomicReference	Object o = v.**get**();	Object o; `isolated` (v) o = v.ref;
	v.**set**(newRef);	`isolated` (v) v.ref = newRef;
AtomicReference() // init = null	Object o = v.**getAndSet**(newRef);	Object o; `isolated` (v) { o = v.ref; v.ref = newRef; }

AtomicReference(init)	boolean b =
	v.**compareAndSet**
	(expect,update);
		boolean b;
		`isolated` (v)
		if (v.ref==expect) {v.ref=update; b=true;}
		else b = false;

Methods in java.util.concurrent.AtomicReference class and their equivalent HJ isolated statements.

- **Variable v refers to an AtomicReference object in column 2 and to a standard non-atomic Java object in column 3.**
- **ref refers to a field of type Object.**
- **AtomicReference<T> can be used to specify a type parameter.**

Three cases of contention among isolated statements

1. **Low contention: when isolated statements are executed infrequently**
   - Use of global isolated statements is usually the best approach. No visible benefit from other techniques because they incur overhead that is not needed since contention is low.
2. **Moderate contention (no variable is a "hot spot"): when serialization of all isolated statements limits performance, but serializing only interfering isolated statements results in good scalability**
   - Atomic variables and object-based isolation usually do well in this scenario since the benefit obtained from reduced serialization outweighs any extra overhead incurred.
3. **High contention (one or more variables are hot spots): when interfering isolated statements dominate the program execution time**
   - Best approach in such cases is to find an alternative approach to isolated e.g., use of finish/phaser accumulators

Properties of isolated statements

How small or big should an isolated statement be?

- Too small → may lose invariants desired from mutual exclusion
- Too big → limits parallelism

Deadlock freedom guarantees

- Observation: no combination of the following HJ constructs can create a deadlock cycle among tasks
  - —isolated + {async, finish, future, forasync, next, barriers, phasers, forall, asyncPhased}

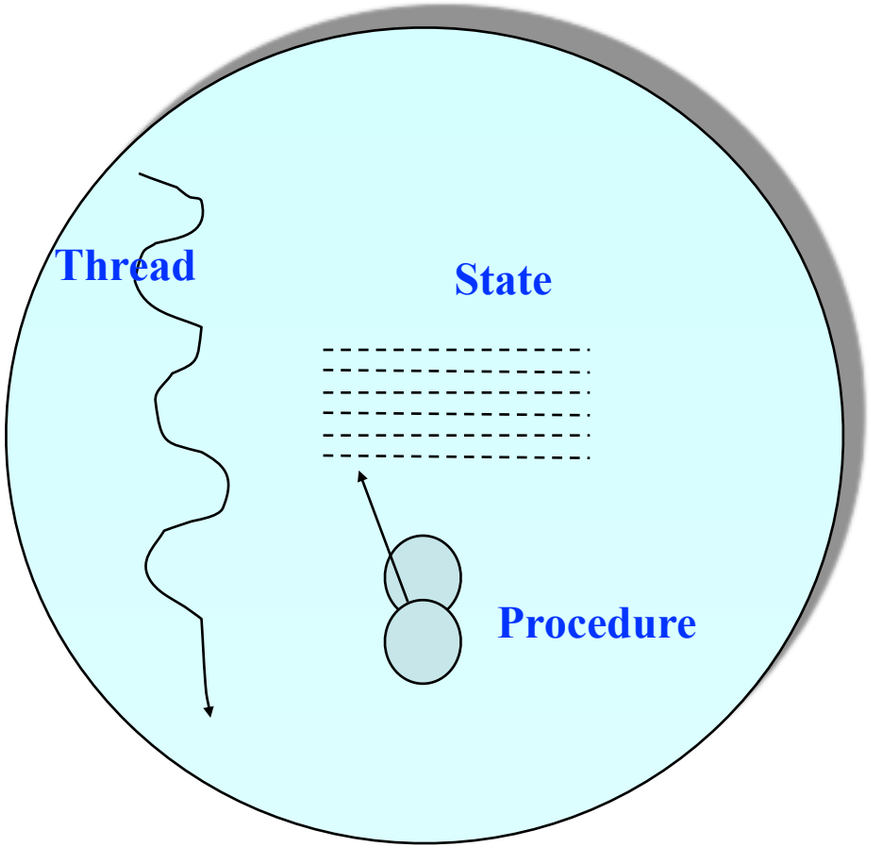
- There are only three HJ constructs that can lead to deadlock
  - —asyncAwait (data-driven tasks)
  - —explicit phaser wait operation (advanced construct)
  - —actors

Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

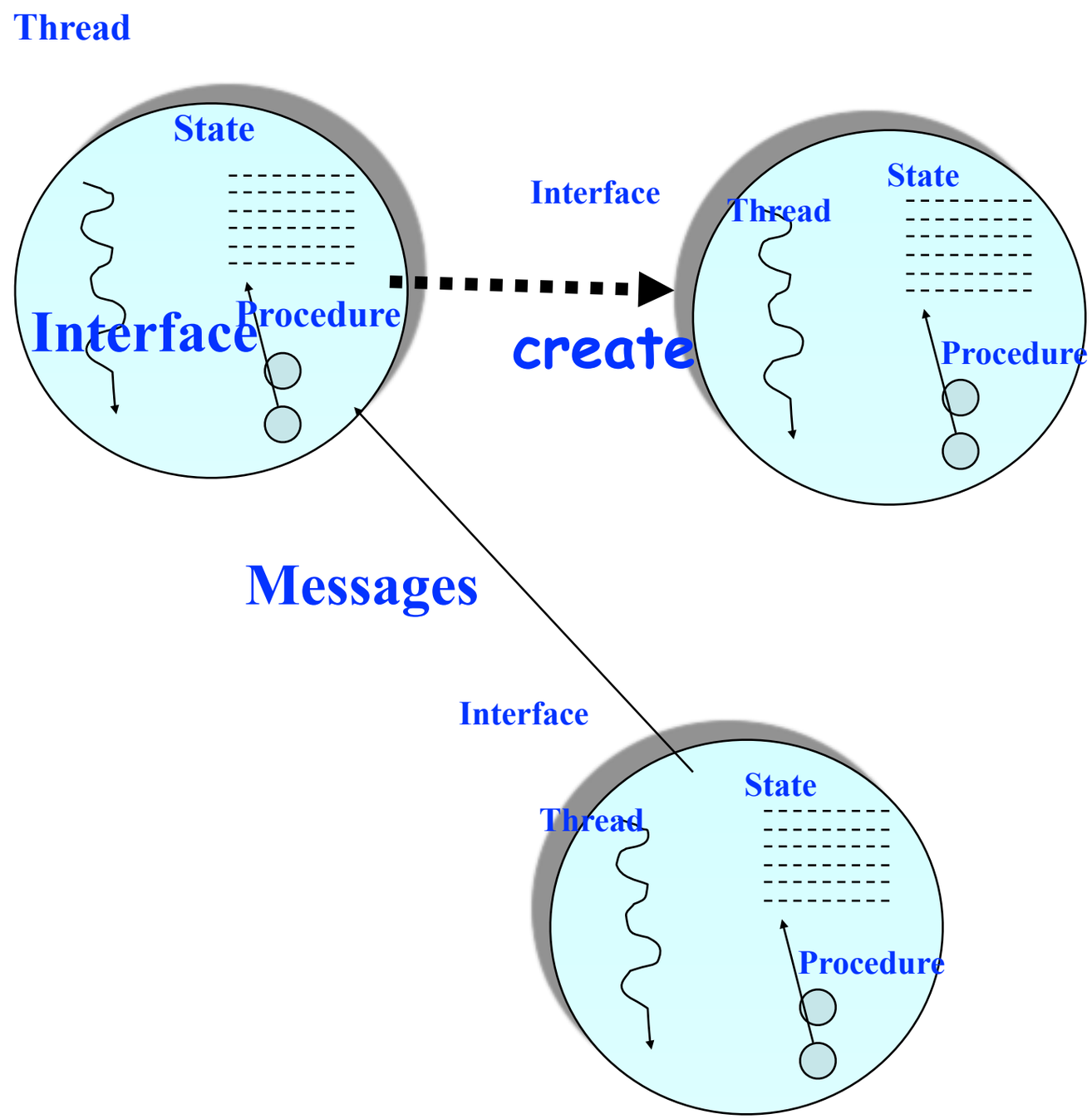
Actors as concurrent objects

- An actor is an autonomous, interacting component of a parallel system.
- An actor has:
  - —an immutable identity (name, global id)
  - —mutable local state (encapsulated)
  - —procedures to manipulate local state (interface)
  - —*a logical thread of control*

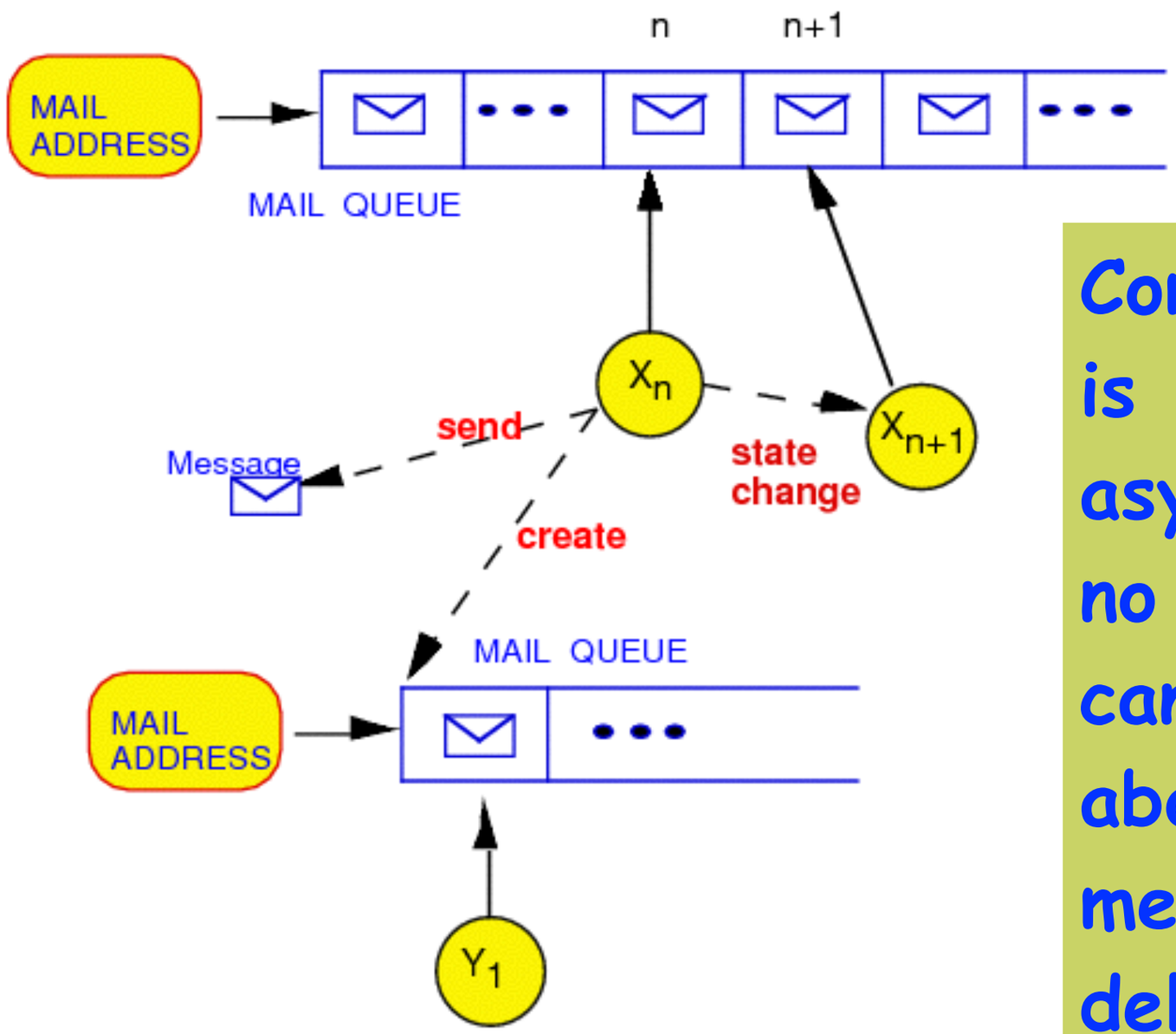


The Actor Model: Fundamentals

- An actor may:
  - process messages
  - send messages
  - change local state
  - create new actors



Arrival Order Nondeterminism

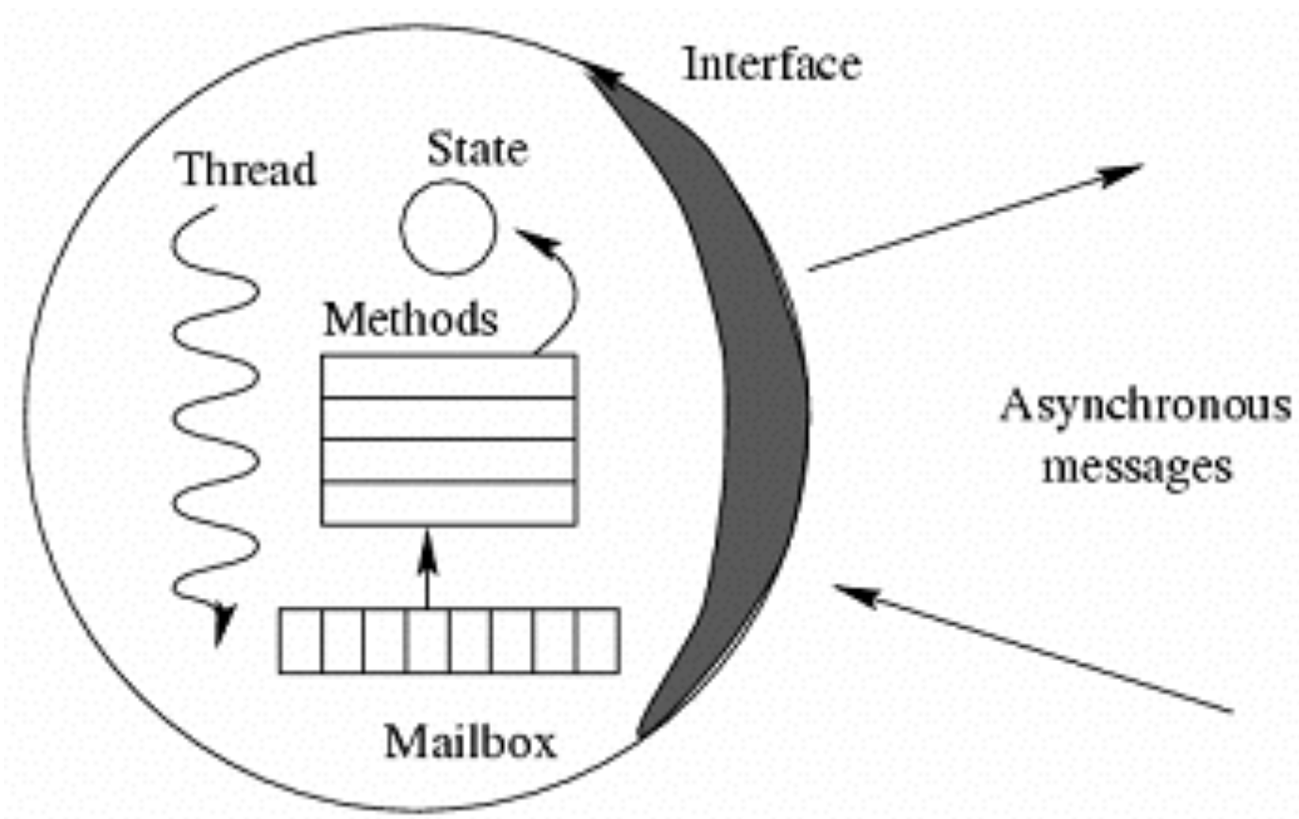


Communication is asynchronous: no assumption can be made about order of message delivery

Actor anatomy

Actors = encapsulated state + behavior (methods) +

thread of control + mailbox



Actor Model

- A message-based concurrency model to manage mutable shared state
- First defined in 1973 by Carl Hewitt

  Further theoretical development by Henry Baker and Gul Agha
- Key Ideas:
  - Everything is an Actor!
  - Analogous to "everything is an object" in OOP
  - Encapsulate shared state in Actors
  - Mutable state is not shared
- Other important features (we will get to these later)
  - Asynchronous message passing
  - Non-deterministic ordering of messages

Actors' Behavior

- **Actors are passive and lazy**
  - Only respond if messages are sent to them
    - Messages may come from other actors or from main program (environment)
  - Only process one message at a time
    - Pending messages are stored in a "*mailbox*"
    - *Parallelism comes from multiple actors processing messages in parallel*
  - Mutate local state **only** while processing a message
  - Mutating local state can result in actor responding differently to subsequent messages

Actor Analogy - Email

- Email accounts are a good simple analogy to Actors
- To notify some information to (i.e. change some state of) A1 another account A2 sends an email (i.e. sends a message) to A1
- A1 has a mailbox to store all incoming messages
- A1 can read (i.e. process) one email at a time
  - At least that is what normal people do :)
- Reading an email can change how you respond to a subsequent email
  - e.g. receiving pleasant news while reading current email can affect the response to a subsequent email
- Actor creation (stretching the analogy)
  - Create a new email account that can send/receive messages

Actor Life Cycle

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-Lib

- 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 completion of an actor!

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 {
           messageCount += 1;
18.        println("Message-" + messageCount + ": " + msg);
19. } } }
```

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

Integer Counter Example

Without Actors:

```
1. int counter = 0;
2. public void foo() {
3.   // do something
4.   isolated(() -> {
5.     counter++;
6.   });
7.   // do something else
8. }
9. public void bar() {
10.  // do something
11.  isolated(() -> {
12.    counter--;
13.  });
14.}
```

- Can also use atomic variables instead of isolated construct

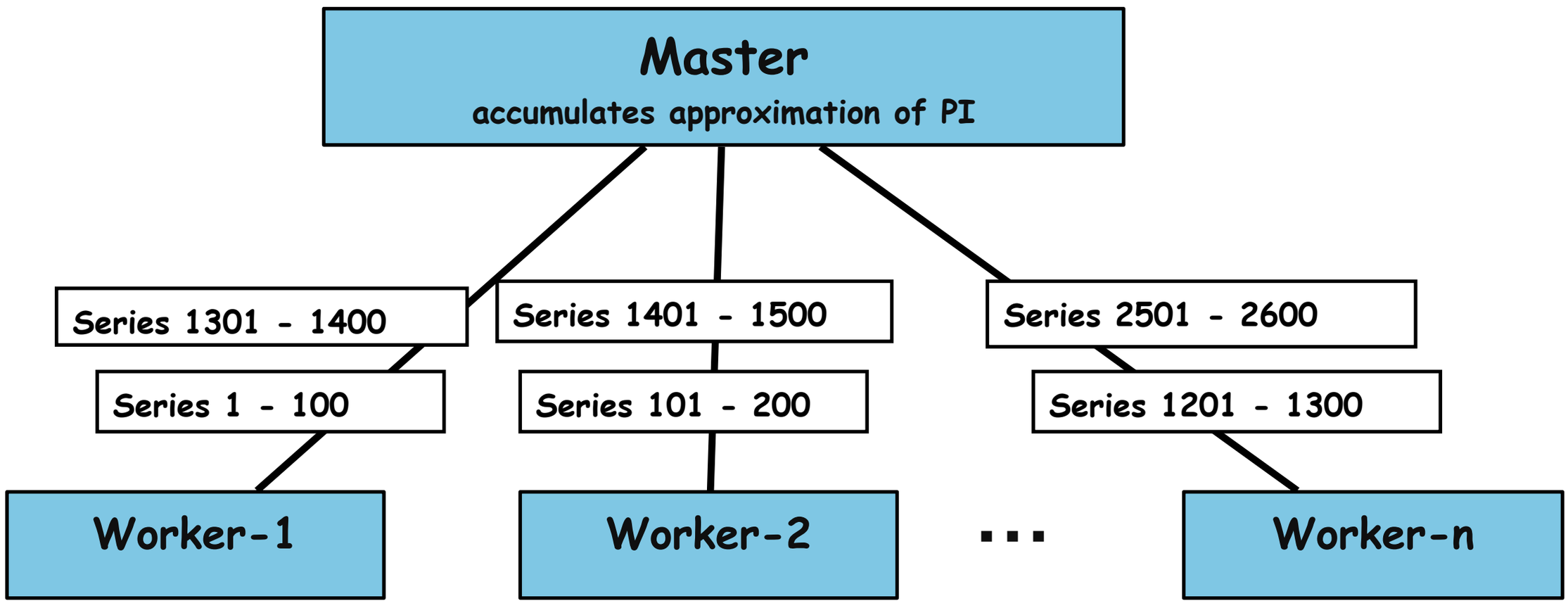
With Actors:

```
14.class Counter extends Actor {
15.  private int counter = 0; // local state
16.  public void process(Message msg) {
17.    if (msg instanceof IncMessage) {
18.      counter++;
19.    } else if (msg instanceof DecMessage){
20.      counter--;
21.} } }
22.  . . .
14.   Counter counter = new Counter();
15.   public void foo() {
16.  // do something
17.  counter.send(new IncrementMessage(1));
18.  // do something else
19.}
20.   public void bar() {
21.  // do something
22.  counter.send(new DecrementMessage(1));
23.}
```


Pi Computation Example

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots.$$

- **Use Master-Worker technique:**



Source: http://www.enotes.com/topic/Pi

Pi Calculation --- Master Actor

```
class Master extends Actor<Object> {
  private double result = 0; private int nrMsgsReceived = 0;
  private Worker[] workers;
  Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
  void start() {
    super.start(); // Starts the master actor
    // Create and start workers
    workers = new Worker[nrWrkrs];
    for (int i = 0; i < nrwrkrs; i++) {
      workers[i] = new Worker();
      workers[i].start();
    }
    // Send messages to workers
    for (int j = 0; j < nrMsgs; j++) {
      someWrkr = ... ; // Select worker for message j
      someWrkr.send(new Work(...));
    }
  } // start()
```


Pi Calculation --- Master Actor (contd)

```
19.  void exit() {
20.    for (int i = 0; i < nrWrkrs; i++) workers[i].send(new Stop());
21.    super.exit(); // Terminates the actor
22.  } // exit()
23.  void process(final Object msg) {
24.    if (msg instanceof Result) {
25.      result += ((Result) msg).result;
26.      nrMsgsReceived += 1;
27.      if (nrMsgsReceived == nrMsgs) exit();
28.    }
29.    // Handle other message cases here
30.  } // process()
31.} // Master
32. . . .
33. // Main program
34. Master master = new Master(w, e, m);
35. finish(() -> master.start(); });
36. println("PI = " + master.getResult());
```


Pi Calculation --- Worker Actor

```
class Worker extends Actor<Object> {
  void process(Object msg) {
    if (msg instanceof Stop) exit();
    else if (msg instanceof Work) {
      Work wm = (Work) msg;
      double result = calculatePiFor(wm.start, wm.end)
      master.send(new ResultMessage(result));    }
  } // process()

  private double calculatePiFor(int start, int end) {
    double acc = 0.0;
    for (int i = start; i < end; i++) {
      acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
    }
    return acc;
  }
} // Worker
```


Actors – Global Consensus

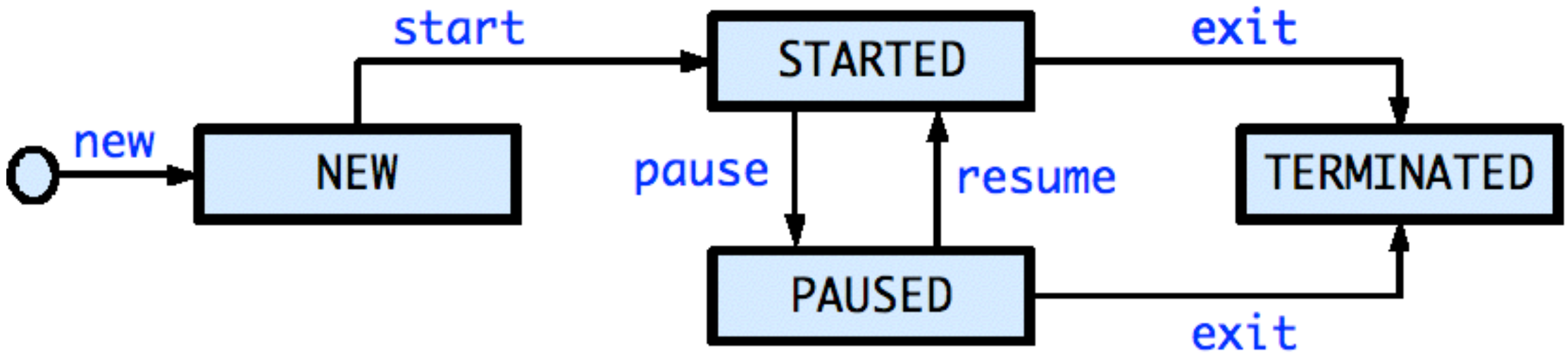
- **Global consensus is simple with barriers/phasers but can be complex with actors e.g.,**
  - **First send message from master actor to participant actors signaling intention**
  - **Wait for all participants to reply they are ready. Participants start ignoring messages sent to them apart from the master**
  - **Once master confirms all participants are ready, master sends the request to each participant and waits for reply from each**
  - **Master notifies participants that consensus has been reached, everyone can go back to normal functioning**

Limitations of Actor Model

- Deadlocks possible
  - Deadlock occurs when all started (but non-terminated) actors have empty mailboxes
- Data races possible when messages include shared objects
- Simulating synchronous replies requires some effort
  - e.g., does not support addAndGet()
- Implementing truly concurrent data structures is hard
  - No parallel reads, no reductions/accumulators
- Difficult to achieve global consensus
  - Finish and barriers not supported as first-class primitives

==> Some of these limitations can be overcome by using a hybrid model that combines task parallelism with actors

Hybrid Actors in HJ-Lib



- Paused state: actor will not process subsequent messages until it is resumed
- Resume actor when it is safe to process the next message
- Akin to Java's wait/notify operations with locks
- Messages can accumulate in mailbox when actor is in PAUSED state (analogous to NEW state)

Actors: pause and resume (contd)

- pause() operation:
  - Is a non-blocking operation, i.e. allows the next statement to be executed.
  - Calling pause() when the actor is already paused is a no-op.
  - Once paused, the state of the actor changes and it will no longer process messages sent (i.e. call process(message)) to it until it is resumed.
- resume() operation:
  - Is a non-blocking operation.
  - Calling resume() when the actor is not paused is an error, the HJ runtime will throw a runtime exception.
  - Moves the actor back to the STARTED state
    - the actor runtime spawns a new asynchronous thread to start processing messages from its mailbox.

Actors - Simulating synchronous replies

- Actors are inherently asynchronous
- Synchronous replies require blocking operations e.g., asyncAwait

```
class CountMessage {
  ... ddf = newDataDrivenFuture();
  int localCount = 0;

  static int getAnIncrement() {
    ... msg = new CountMessage();
    counterActor.send(msg);
    // use ddf to wai for response
    // THREAD-BLOCKING
    finish(() -> {
      asyncAwait(msg.ddf, () -> {});
    });
    // return count from the message
    return msg.localCount;
  }
}
```

```
class CounterActor extends
    Actor<Object> {
  int counter = 0;
  void process(Object m) {
    if (m instanceof CountMessage) {
     CountMessage cm = (CountMessage) m;
      counter++;
      msg.localCount = counter;
      msg.ddf.put(true);
    }
  }
}
```


Synchronous Reply using Async-Await (without pause/resume)

```
1. class SynchronousReplyActor1 extends Actor<Message> {
2.   void process(Message msg) {
3.     if (msg instanceof Ping) {
4.       finish(() -> {
5.         HjDataDrivenFuture<T> ddf = newDataDrivenFuture();
6.         otherActor.send(ddf);
7.         finish(() -> {
8.           asyncAwait(ddf, () -> {
9.             T synchronousReply = ddf.get();
10.            // do some processing with synchronous reply
11.          });
12.        });
13.      });
14.    } else if (msg instanceof ...) { ... } } }
```


Synchronous Reply using Pause/Resume

```
1.  class SynchronousReplyActor2 extends Actor<Message> {
2.    void process(Message msg) {
3.      if (msg instanceof Ping) {
4.        HjDataDrivenFuture<T> ddf = newDataDrivenFuture();
5.        otherActor.send(ddf);
6.        pause(); // when paused, the actor doesn't process messages
7.        asyncAwait(ddf, () -> { // processes synchronous reply
8.            T synchronousReply = ddf.get();
9.            // do some processing with synchronous reply
10.           resume(); // allow actor to process next message
11.       });
12.   } else if (msg instanceof ...) { ... } } }
```


Parallelizing Actors in HJ

- Two techniques:
  - Use finish construct to wrap asyncs in message processing body
    - Finish ensures all spawned asyncs complete before next message returning from process()
  - …

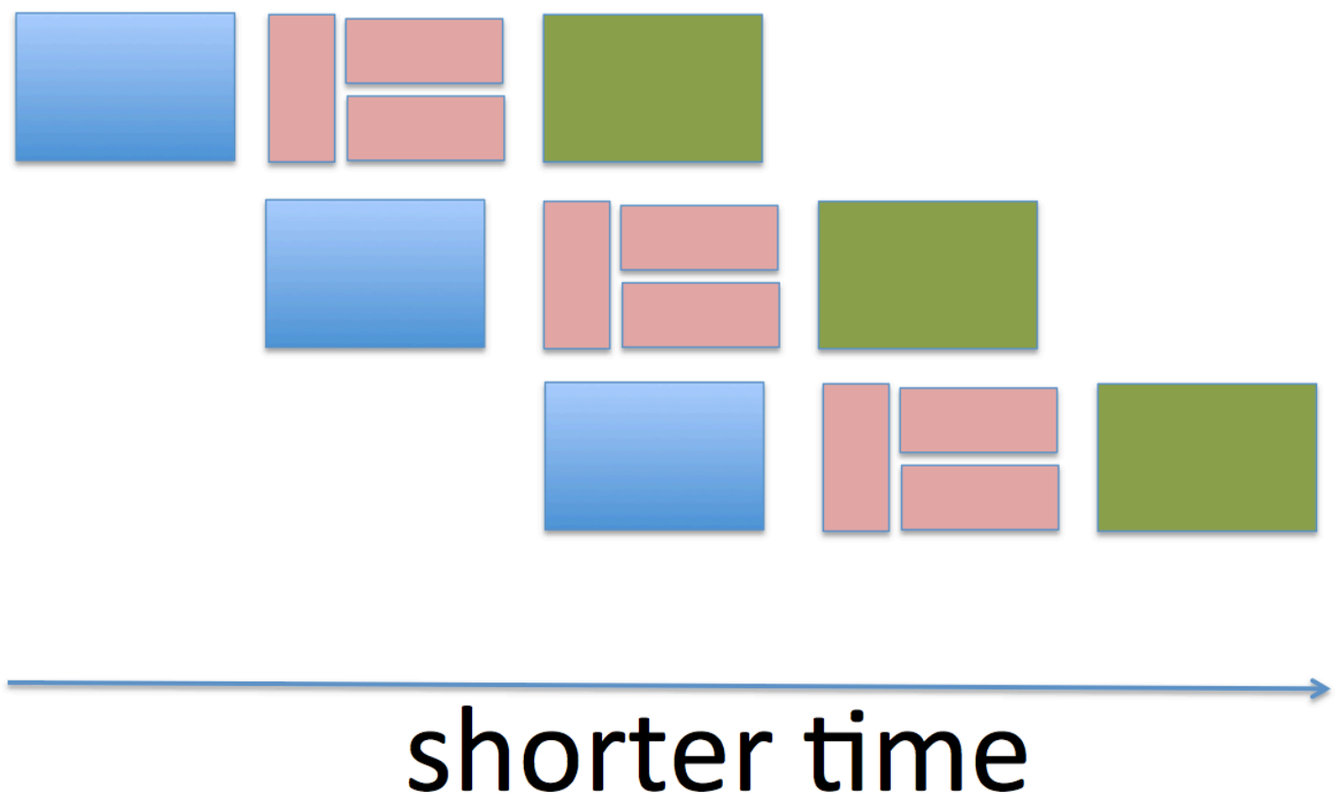
Parallelizing Actors in HJ

- **Use finish construct in MP body and spawn child tasks**

```
1.  class ParallelActor1 extends Actor<Message> {
2.    void process(Message msg) {
3.      finish(() -> {
4.        async(() -> { S1; });
5.        async(() -> { S2; });
6.        async(() -> { S3; });
7.      });
8.    }
9.  }
```


Parallelizing Actors Example

- **Pipelined Parallelism**
  - **Reduce effects of slowest stage by introducing task parallelism.**
  - **Increases the throughput.**



Parallelizing Actors Example

```
class ConsumerActor extends Actor<Object> {
  private double resultSoFar = 0;
  @Override
  protected void process(final Object theMsg) {
    if (theMsg != null) {
      final double[] dataArray = (double[]) theMsg;
      final double localRes = doComputation(dataArray);
      resultSoFar += localRes;
    } else { ... }
  }
  private double doComputation(final double[] dataArray) {
    final double[] localSum = new double[2];
    finish(() -> {
      final int length = dataArray.length;
      final int limit1 = length / 2;
      async(() -> {
        localSum[0] = doComputation(dataArray, 0, limit1);
      });
      localSum[1] = doComputation(dataArray, limit1, length);
    });
    return localSum[0] + localSum[1];
  }
}
```

Parallelizing Actors in HJ

- **Allow escaping asyncs inside MP body**

```
1.  class ParallelActor2 extends Actor<Message> {
2.    void process(Message msg) {
3.      pause();
4.      async(() -> { S1; }); // escaping async
5.      async(() -> { S2; }); // escaping async
6.      async(() -> { // async that must be executed before next message
7.        S3;
8.        resume();
9.      });
10.   }
11. }
```


Parallelizing Actors in HJ

- Two techniques:
  - Use finish construct to wrap asyncs in message processing body
    - Finish ensures all spawned asyncs complete before next message returning from process()
  - Allow escaping asyncs inside process() method
    - WAIT! Won't escaping asyncs violate the one-message-at-a-time rule in actors
    - Solution: Use pause and resume

Uses of hybrid actor+task parallelism

- Can use finish to detect actor termination
- Event-driven tasks
- Stateless Actors
  - If an actor has no state, it can process multiple messages in parallelism
- Pipeline Parallelism
  - Actors represent pipeline stages
  - Use tasks to balance pipeline by parallelizing slower stages

Summary of Mutual Exclusion approaches in HJ

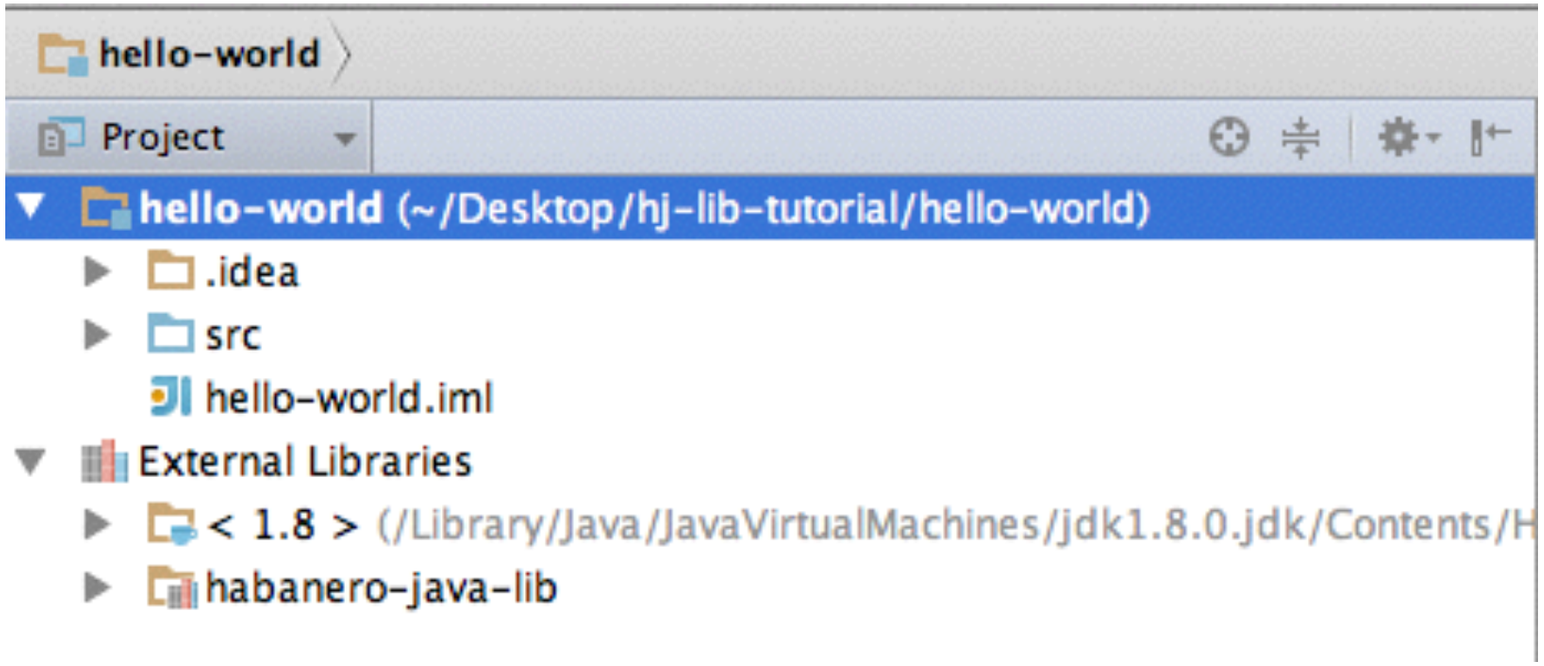
- Isolated --- analogous to critical sections
- Object-based isolation, isolated(a, b, ...)
  - Single object in list --- like monitor operations on object
  - Multiple objects in list --- deadlock-free mutual exclusion on sets of objects
- Java atomic variables --- optimized implementation of object-based isolation
- Java concurrent collections --- optimized implementation of monitors
- Actors --- different paradigm from task parallelism (mutual exclusion by default)

Outline

- Part 1 (Parallelism)
  - Task and Loop-level Parallelism (Async, Finish, Forall)
  - Futures
  - Data-driven Tasks
  - Data Races and Determinism
- BREAK
- Part 2 (Concurrency)
  - Global and Object-based Isolation
  - Actors
  - HJ-Lib Implementation

Habanero-Java library

- Pure Java library without any other dependencies



- Uses Java 8 Lambda Expressions
  - Required for terse syntax
  - Early access release downloads of the Java 8 available
  - Target release of Java 8 is March 2014

HJ-lib Implementation

- Runs standard Java code
  - Relies on JIT optimizations.
- async tasks are custom Runnable classes called Activity
  - Uses the ForkJoinPool implementation of the ExecutorService to schedule async tasks
- Synchronization constructs implemented with custom data structures
  - Based around DataDrivenContol

DataDrivenControl

- Dynamic single-assignment of value
- Binds a value and a list of code blocks
- Code blocks are suspended until value becomes available
  - Blocks the underlying worker thread.
- Code blocks executed only when value is resolved

Blocking Constructs

- Examples:
  - End of finish
  - Future get
  - Phaser next
- Blocks underlying worker thread
- Use ManagedBlocker interface to launch additional worker threads
- Too many blocking constructs can result in lack of performance and exceptions
  - `java.lang.IllegalStateException: Error in executing blocked code! [89 blocked threads]`
  - Maximum number of worker threads can be configured
  - `System.setProperty(HjSystemProperty.maxThreads.propertyKey(), "100");`

Deadlock Detection

- Runtime keeps track of number of ready and suspended tasks
- Implemented using Observer pattern
- Deadlock reported when there are no ready tasks but at least one suspended task
- Example:
  - `edu.rice.hj.runtime.util.DeadlockException: DEADLOCK: total tasks: 3, completed tasks: 0, suspended tasks: 3`

Implementation of finish statement

- Finish scopes are tracked using ThreadLocal variables
- Each new finish scope
  - Pushed on to a stack
  - Keeps track of number of spawned tasks
  - Number of completed tasks
  - Wraps a DataDrivenControl instance
- End of finish is blocking
  - Uses DataDrivenControl.pause()
  - DataDrivenControl is resolved when number of spawned tasks equals number of completed tasks

Isolated statement

- Uses ReentrantReadWriteLock instances
- Number of locks is configurable using:
  - `System.setProperty(HjSystemProperty.isolatedLocks.propertyKey(), "10");`
- Object-based isolation hashes to lock instances
- Global isolated required obtaining all lock instances in write mode

Implementation of Actors

- Mailbox uses custom concurrent linked-list implementation
- Message processed in asynchronous tasks
- Messages are multiplexed into single task instance to reduce overhead of task creation
- pause/resume implemented using DataDrivenControl to launch message processing tasks

HJ-lib Release

- Single jar available for download
  - http://www.cs.rice.edu/~vs3/hjlib/habanero-java-lib.jar
- Installation Instructions
  - https://wiki.rice.edu/confluence/display/PARPROG/Download+and+Set+Up
- PPoPP Tutorial
  - https://wiki.rice.edu/confluence/display/PARPROG/Habanero-Java%3A+Multicore+Programming+for+the+Masses

Future Directions

- Cooperative Runtime for HJ-lib
  - No threads are ever blocked!
  - Uses Delimited Continuations and DataDrivenControl
  - Introduce Kilim Weaver dependency
    - http://www.malhar.net/sriram/kilim/
  - Initial support for Java 7 syntax (Kilim / ASM limitation)

BACKUP SLIDES START HERE

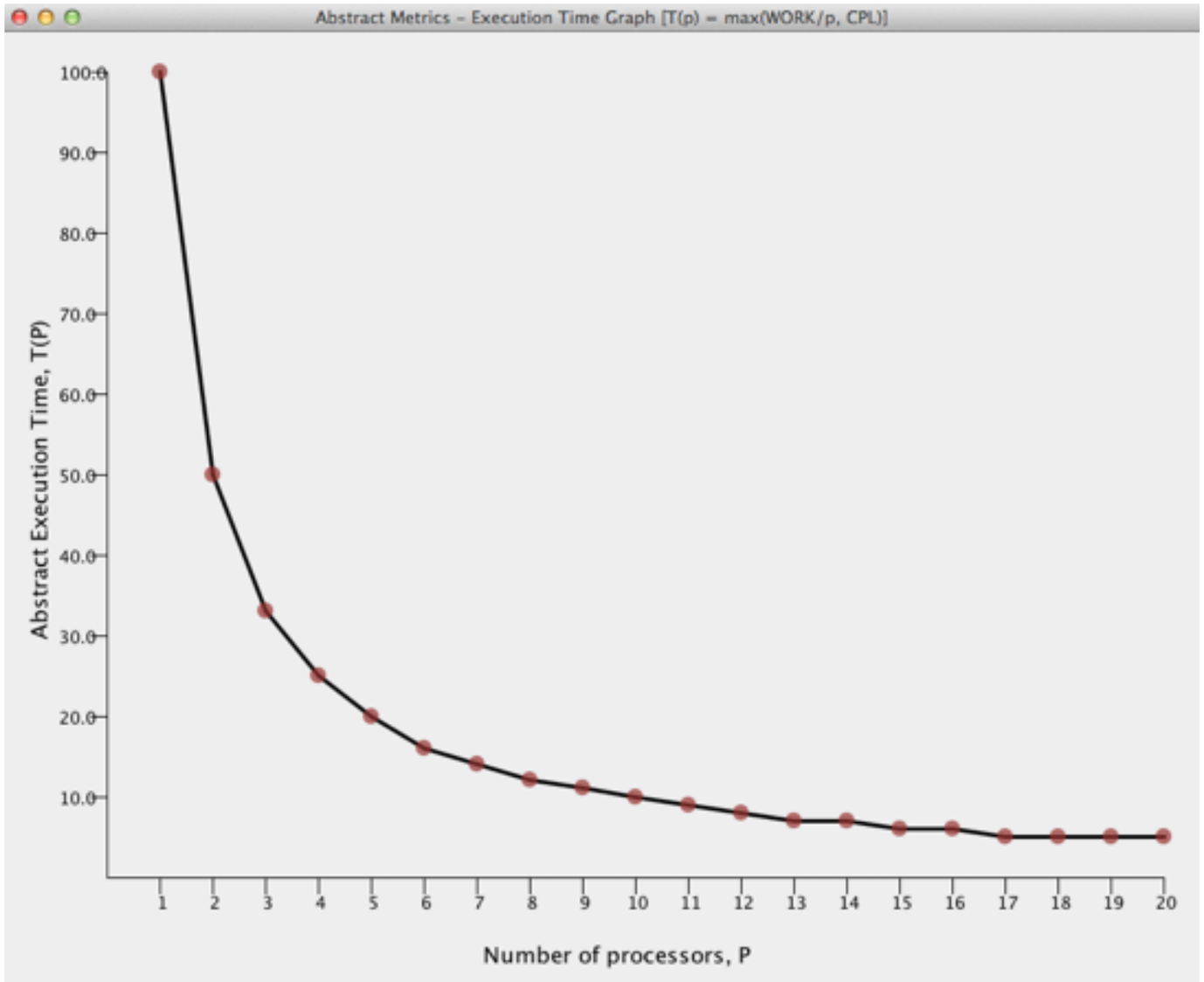
Abstract metrics

- Prints the total work and Critical Path Length
- Supported for all HJ-Lib constructs
- Enabled using:

```
System.setProperty(
  HjSystemProperty.
  abstractMetrics.propertyKey(),
  “true");
```

- Dump obtained by:

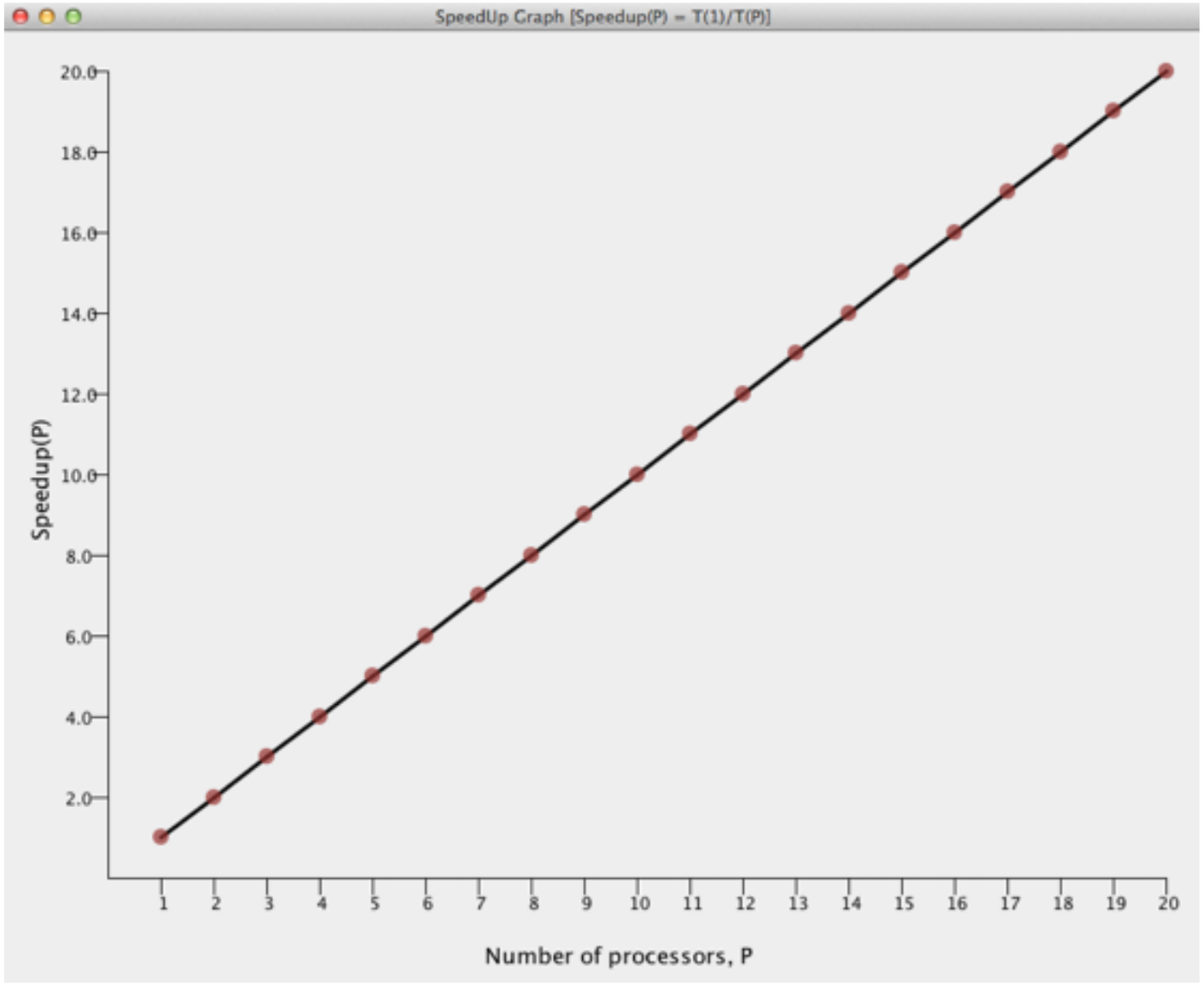
```
final HjMetrics actualMetrics =
  abstractMetrics();
AbstractMetricsManager.
  dumpStatistics(actualMetrics);
```



Speedup charts

- **Use Abstract Metrics to display speedup charts**
- **Enabled using:**

```
System.setProperty(
  HjSystemProperty.
  abstractMetrics.propertyKey(),
  "true");
System.setProperty(
  HjSystemProperty.
  executionGraph.propertyKey(),
  "true");
System.setProperty(
  HjSystemProperty.
  speedUpGraph.propertyKey(),
  “true");
```



Nqueens example with seq clause

```
void nqueensKernel(final int[] a, final int depth,
                   final FinishAccumulator ac) {
  if (size == depth) {
    ac.put(1);
    return;
  }
  /* try each possible position for queen <depth> */
  for (int i = 0; i < size; i++) {
    final int ii = i;
    asyncSeq(depth >= cutoff_value, () -> {
      /* allocate a temporary array and copy <a> into it */
      final int[] b = new int[depth + 1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = ii;
      if (boardValid((depth + 1), b)) {
        nqueensKernel(b, depth + 1, ac);
      }
    });
  }
}
```