# COMP 322: Fundamentals of Parallel Programming

## Module 2: Nondeterministic Shared-Memory Parallelism and Concurrency

©2013 by Vivek Sarkar

March 13, 2013

DRAFT VERSION – PLEASE DO NOT DISTRIBUTE

## Contents

1. Critical Sections and the HJ “isolated” statement
2. Serialized Computation Graph for Isolated Statements
3. Parallel Spanning Tree Example
4. Data Races and the Isolated Statement
5. Implementations of Isolated Statements
6. Atomic Variables
   6.1 AtomicInteger and AtomicIntegerArray
   6.2 AtomicReference
7. Java Collections Framework
   7.1 java.util.concurrent.ConcurrentHashMap
   7.2 java.util.concurrent.ConcurrentLinkedQueue
   7.3 java.util.concurrent.CopyOnWriteArraySet
8. The Actor Model
   8.1 Actors
   8.2 Desirable Properties
   8.3 Habanero-Java Actors
      8.3.1 Tips and Pitfalls
9. Linearizability of Concurrent Objects
10. Relating Linearizability to the Computation Graph model
11. Closures, HJ Asyncs, and Java Runnable Objects
Module 2: Nondeterministic Shared-Memory Parallelism and Concurrency

12 Java Threads 28

13 Using Callable objects with Java threads 30

14 Introduction to Java’s synchronized statement 31
1 Critical Sections and the HJ “isolated” statement

For the programming constructs `async`, `finish`, `future`, `get`, `forall`, the following situation was defined to be a *data race* error — when two accesses on the same shared location can potentially execute in parallel such that at least one access is a write. However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations.

Consider the example shown below in Listing 1. The job of method `deleteTwoNodes()` in line 14 is to delete the first two nodes in doubly-linked list `L`. It does so by calling `L.delete()` and `L.next.delete()` in parallel in lines 16 and 17. Each call to `delete()` needs to perform the book-keeping operations in lines 6 and 7 in mutual exclusion from the other, so that there’s no chance of statement instances from the two different calls being interleaved in any way. The term “mutual exclusion” refers to the requirement that lines 6 and 7 are are executed as a single indivisible step by each task, thereby excluding the other tasks when one task is executing those statements.

```java
class DoublyLinkedList {
    DoublyLinkedList prev, next;
    ...
    void delete() {
        isolated { // start of mutual exclusion region (critical section)
            if (this.prev != null) this.prev.next = this.next;
            if (this.next != null) this.next.prev = this.prev
        } // end of mutual exclusion region (critical section)
        ...
    }
    ...

    static void deleteTwoNodes(DoublyLinkedList L) {
        finish {
            async L.delete();
            async L.next.delete();
        }
    }
}
```

Listing 1: Example of two tasks performing conflicting accesses

The predominant approach to ensure mutual exclusion proposed many years ago is to enclose a code region such as lines 6 and 7 in a *critical section* [17]. The following definition of critical sections from [32] captures the idea:

> "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."

The primary mechanisms available to a Java programmer for implementing the synchronization necessary for critical sections is the `synchronized` language constructs, and the `java.util.concurrent.locks` library package. You will learn about both mechanisms later in the course. Instead, the Habanero Java (HJ) language offers a simpler construct, `isolated`, that can be used to directly implement critical sections, as shown in lines 5–8 of Listing 1.

Specifically, the HJ construct, `isolated ⟨stmt1⟩`, guarantees that each instance of `⟨stmt1⟩` will be performed in mutual exclusion with all other potentially parallel instances of `isolated` statements `⟨stmt2⟩`. Thus, the
use of isolated in line 5 of Listing 1 is sufficient to ensure that the computations in the two calls to delete() (lines 16 and 17) can safely be executed in parallel. After each task executes the isolated statement in lines 6 and 7, it can perform the computation in line 9 in parallel with the other task. Unlike finish, isolated does not dictate the order in which the two isolated instances should be executed; it just ensures that they execute in mutual exclusion regardless of the order.

The body of an isolated statement may perform any sequential Java computation including method calls and exceptions. It is illegal to execute a parallel construct (such as async, finish, get, or forall) within an isolated statement. If an exception is thrown within an isolated statement, $S$, it can be caught by a handler within or outside $S$. If control exits $S$ after an exception, then all updates performed by $S$ before throwing the exception will be observable after exiting $S$. isolated statements may be nested, but an inner isolated statement is essentially a no-op since isolation is guaranteed by the outer statement.

There is a trade-off between making isolated statements too big or too small in scope. If they are too big, then the parallelism in the program will be limited because interfering isolated statements cannot be executed in parallel. (Remember Amdahl’s Law?) If they are too small, then they may not provide the desired program semantics. For example, if the isolated statement in line 5 of Listing 1 is replaced by two separate isolated statements for lines 6 and 7 of Listing 1, we would lose the invariant that lines 6 and 7 execute as an indivisible operation with respect to other tasks. It is also important to note that no combination of finish, async, get, forall and isolated constructs can create a deadlock cycle among tasks.

2 Serialized Computation Graph for Isolated Statements

How can the Computation Graph (CG) structure be extended to model isolated statements? We start by modeling each instance of an isolated statement as a distinct step (node) in the CG. This is permissible since the execution of an isolated statement is purely sequential with no internal continuation points. Next, we reason about the order in which interfering isolated statements are executed. This is complicated because the order may vary from execution to execution e.g., the isolated statement instance invoked by the async in line 16 of Listing 1 may execute before the isolated statement in line 17 in one execution, and vice versa in another execution.

To solve this dilemma, we introduce a family of Serialized Computation Graphs (SCG’s) that can be obtained when executing a program for a given input. Each SCG consists of a CG with additional serialization edges. Consider a CG being constructed on-the-fly as a parallel program executes. Each time an isolated step, $S'$, is executed, we add a serialization edge from $S$ to $S'$ for each isolated step, $S$, that has already executes. For simplicity, we omit serialization edges when the source and destination steps belong to the same task, since they will always be redundant.

Each SCG represents a set of executions in which all interfering isolated statements execute in the same order. Different SCG’s are used to reason about different orders of execution of interfering isolated statements. We can use SCG’s to reason about properties of parallel programs that you have already studied with respect to a specific CG. For example, the critical path length (CPL) of the execution of a program with isolated statements can be obtained by computing the CPL of the corresponding SCG.

Consider the computation graph in Figure 1 (ignoring the red serialization edges) and assume that nodes v10, v11, v16 all consist of interfering isolated steps as shown in the bottom right of the figure. There are three possible orderings for these nodes, when taking continue, spawn and join edges into account: v16 → v10 → v11, v10 → v16 → v11 and v10 → v11 → v16. Each order leads to a different SCG, when serialization edges are added. The red edges in Figure 1 show serialization edges added when the isolated steps execute in the sequence, v10 → v16 → v11. These edges increase the critical path length of the SCG from 17 nodes (without serialization edges) to 18 nodes (with serialization edges). Alternate SCG’s are also possible with serialization edges v10 → v11 → v16 and v16 → v10 → v11 which result in a critical path length of 17 nodes.
3 Parallel Spanning Tree Example

In this section, we discuss a more complicated use of isolated statements in a program written to find a spanning tree of an undirected graph. An undirected graph \( G = (V, E) \) consists of a finite set of vertices, \( V \), and a set of unordered edges, \( E \). An edge, \( \{u, v\} \) connects vertices \( u \neq v \), but the order is not significant. This is unlike directed graphs, such as a computation graph, where the direction of the edge matters. A common representation for undirected graphs is the adjacency list. For each vertex \( u \) in the graph, an array \( \text{neighbors} \) can be used to store the set of vertices \( v \neq u \) such that there is an edge connecting vertices \( u \) and \( v \).

A path from vertex \( a \) to vertex \( b \) in graph \( G = (V, E) \) is a sequence \( \langle v_0, v_1, v_2, \ldots, v_k \rangle \) of vertices where \( v_0 = a \), \( v_k = b \), and \( \{v_i, v_{i+1}\} \in E \) for \( i = 0, 1, \ldots, k-1 \). A graph is connected if each pair of vertices is connected by at least one path. A cycle (or circuit) is a path that starts and ends with the same vertex. A tree is a special kind of undirected graph that is connected and contains no cycles. A tree always contains \( |V| - 1 \) edges. A rooted tree has a designated vertex that is called the root. A rooted tree can be represented compactly by a parent field in each non-root vertex such that there is a tree edge \( \{u, v\} \) if and only if \( u\.parent = v \) or \( v\.parent = u \). A spanning tree of a connected undirected graph \( G \) is a tree containing all the vertices of \( G \), and a subset of \( G \)'s edges. A graph may have multiple spanning trees. Figure 2 contains a simple example of an undirected graph.

A path from vertex \( a \) to vertex \( b \) in graph \( G = (V, E) \) is a sequence \( \langle v_0, v_1, v_2, \ldots, v_k \rangle \) of vertices where \( v_0 = a \), \( v_k = b \), and \( \{v_i, v_{i+1}\} \in E \) for \( i = 0, 1, \ldots, k-1 \). A graph is connected if each pair of vertices is connected by at least one path. A cycle (or circuit) is a path that starts and ends with the same vertex. A tree is a special kind of undirected graph that is connected and contains no cycles. A tree always contains \( |V| - 1 \) edges. A rooted tree has a designated vertex that is called the root. A rooted tree can be represented compactly by a parent field in each non-root vertex such that there is a tree edge \( \{u, v\} \) if and only if \( u\.parent = v \) or \( v\.parent = u \). A spanning tree of a connected undirected graph \( G \) is a tree containing all the vertices of \( G \), and a subset of \( G \)'s edges. A graph may have multiple spanning trees. Figure 3 shows three possible spanning trees for the graph in Figure 2.

Spanning trees have multiple applications in areas as diverse as telephone networks and transportation, since a spanning tree contains the minimum number of edges \( (|V| - 1) \) to ensure that the vertices in a graph remain connected. Removing an edge from a spanning tree with cause the resulting graph to become disconnected. If each edge \( e = \{u, v\} \) in the input graph is assigned a weight (e.g., based on the distance or cost of the connection between the two vertices), then a minimum spanning tree is a spanning tree with the smallest total cost, when adding the weights of edges in the tree. The problem of finding the minimum spanning tree has received much attention in Computer Science. In this section, we will study the simpler problem of finding any spanning tree of an input graph.
Figure 2: Example undirected graph (source [28])

Figure 3: Three spanning trees for undirected graph in Figure 2 (source [28])
Listing 2 shows the sketch of a Java class $V$ that implements a sequential program to compute the spanning tree of an undirected graph. Each vertex in the input graph is created as an instance of class $V$, and the its neighbors array contains its adjacency list. The algorithm starts by setting $root.parent = root$ in line 17. Even though the root vertex does not have a defined parent, this convention simplifies the implementation of the tryLabeling() method, as we will see. Line 18 invokes the recursive method, compute(), on root.

The body of method compute() is defined in lines 9 – 13. It iterates through each vertex (named child) in the neighbors list of the current (this) vertex, and attempts to become the parent of child by calling child.tryLabeling(). Consider an example where the this vertex is $a$ and the child vertex is $b$ in line 11. Line 5 in method tryLabeling() will then check if $b$’s parent is null. If so, it sets $b$.parent to equal $a$ and returns true. Otherwise, the call to tryLabeling() returns false. Back in line 11, method compute() will recursively call itself on child $a$ if tryLabeling() returns true, thereby ensuring that compute() is called exactly once for each vertex. When all recursive calls have completed, the output spanning tree is represented by the parent fields.

```
1 class V {
2    V [] neighbors; // adjacency list for input graph
3    V parent; // output value of parent in spanning tree
4    boolean tryLabeling(V n) {
5        if (parent == null) parent=n;
6        return parent == n;
7    } // tryLabeling
8    void compute() {
9        for (int i=0; i<neighbors.length; i++) {
10           V child = neighbors[i];
11           if (child.tryLabeling(this))
12              child.compute(); //escaping async
13        }
14    } // compute
15 } // class V
. . .
17 root.parent = root; // Use self-cycle to identify root
18 root.compute();
. . .
```

Listing 2: Sequential Algorithm for computing the Spanning Tree of an undirected graph

Listing 3 shows a parallel version of the spanning tree program in Listing 2. The only changes are the addition of isolated in line 5, the addition of async in line 12, and the addition of finish in line 18. The addition of these keywords lead to a parallel program which computes a valid spanning tree of the graph, but without preserving the left-to-right order when traversing neighbors in the original sequential version. This is a nondeterministic algorithm since different executions may yield different spanning trees, all of which are valid solutions.

```
1 class V {
2    V [] neighbors; // adjacency list for input graph
3    V parent; // output value of parent in spanning tree
4    boolean tryLabeling(V n) {
5        if (parent == null) parent=n;
6        return parent == n;
7    } // tryLabeling
8    void compute() {
9        for (int i=0; i<neighbors.length; i++) {
10           V child = neighbors[i];
11           if (child.tryLabeling(this))
12              async child.compute(); //escaping async
13        }
14    } // compute
15 } // class V
```

Listing 3: Parallel Algorithm for computing the Spanning Tree of an undirected graph
Listing 3: Parallel Spanning Tree of an undirected graph

There are some other points worth observing in this example. Method `compute()` in lines 8–14 contains an `async` but no `finish` statement. In such a case, the `async` in line 12 is called an *escaping async*, because its parent task’s method can return before the `async` has completed. Instead, a `finish` is included in line 18 to ensure that all `async`’s terminate before the program proceeds to line 19. The `isolated` statement in line 5 is necessary because multiple neighbors of a vertex `v2` may compete to be its parent. The winning vertex `v1` is the one whose `async` finds `v2.parent == null`, in which case it sets `v2.parent = v1`. All other neighbors will then fail to become the parent of `v2` since `v2.parent == null` will no longer be true.

4 Data Races and the Isolated Statement

The following definition of data races studied earlier can also be directly applied to an SCG:

"Formally, a data race occurs on location `L` in a program execution with computation graph `CG` if there exist steps `S_1` and `S_2` in `CG` such that:

1. `S_1` does not depend on `S_2` and `S_2` does not depend on `S_1` i.e., there is no path of dependence edges from `S_1` to `S_2` or from `S_2` to `S_1` in `CG`, and
2. both `S_1` and `S_2` read or write `L`, and at least one of the accesses is a write."

By this definition, there is no data race between (say) node `v10` and `v16` in Figure 1, since the insertion of serialization edges ensures that there cannot be a data race between any two interfering `isolated` statement instances. The absence of a data race appears reasonable for variable `x` in Figure 1 (since all executions will result in `x++` being performed exactly three times in isolation), but what about variable `y`? The final value of `y` depends on the order in which the `isolated` statements are executed. While this appears to be a race of some sort, it is not considered to be a data race according to the above definition. In general, data races are considered more harmful than other forms of races in parallel programs, because the semantics of data races depends on intricate details of the underlying memory consistency model.

As another example, consider the `forall` loop in Listing 4. The read and write accesses to `rank.count` in line 3 result in data races between all pairs of `forall` iterations.

```
1   rank.count = 0; // rank object contains an int field, count
2   forall (point[i] : [0:m-1]) {
3       int r = rank.count++;
4       StringArray[i] = Hello, World from task with rank = + r;
5   }
```

Listing 4: Example of a forall loop with a data race error

Earlier, you learned how `finish` statements can be inserted to fix data race errors. For this example, an `isolated` statement needs to be used instead, as shown in line 4 of Listing 5.

```
1   rank.count = 0; // rank object contains an int field, count
```
forall (point[i] : [0:m-1]) {
  int r;
  isolated { r = rank.count++; }
  StringArray[i] = Hello, World from task with rank = + r;
}

Listing 5: Using an isolated statement to fix the data race error in Listing 5

Note that the program in Listing 5 is nondeterministic because different executions with the same input can result in different outputs (related to which iteration \( i \) gets assigned which rank \( r \)). It is informative to review the following property studied earlier, in light of isolated constructs:

**Determinism property**: if a parallel program with async, finish, forall, future and get operations can never have a data race, then it must be deterministic with respect to its inputs.

This Determinism property does not hold in general for data-race-free programs that include isolated statements, as seen in the examples in Figure 1 and in Listing 5. Thus, the distinction between deterministic and data-race-free programs needs to be understood with more care, when isolated statements are also under consideration.

Finally, we note that the isolated construct supports weak isolation [26]. With weak isolation, any memory accesses performed outside isolated constructs are not checked for interference with accesses within isolated constructs. Thus, there is the possibility of the “indivisibility” associated with an isolated statement being broken due to data races with non-isolated accesses, such as (say) a non-isolated x++ operation in node v17 in Figure 1.

### 5 Implementations of Isolated Statements

While it is convenient for the programmer to use isolated statements, this convenience imposes major challenges on the implementation of isolated constructs. The discussion in the previous section highlights this issue, especially if we consider the fact that the n1.delete() and n2.delete() calls may or may not interfere depending on where n1 and n2 are located in the linked list.

The Habanero-Java implementation available for COMP 322 takes a simple single-lock approach to implementing isolated statements. You will learn more about locks later in the class. The idea behind a single-lock approach is to treat each entry of an isolated statement as an acquire() operation on the lock, and each exit of an isolated statement as a release() operation on the lock. Though correct, this approach essentially implements isolated statements as critical sections.

An alternate approach for implementing isolated statements being explored by the research community is Transactional Memory (TM) [26]. In Software Transactional Memory (STM), a combination of compiler and runtime techniques is used to optimistically execute transactions (instances of isolated statements) in parallel while checking for possible interference. If an interference is detected, one of the transactions is “rolled back” and restarted. Given the large amount of book-keeping necessary for logging read and write operations to check for interference, it is widely believed that software-only approaches to TM incur too much overhead to be useful in practice. Instead, there have been multiple proposals for Hardware Transactional Memory (HTM) support to assist with this book-keeping. As yet, no computer with HTM support is widely available, but a few hardware prototypes have begun to show promise in this direction e.g., [15].

Since no practical TM approach is currently available in software or hardware, we will focus our attention in this lecture on atomic variables described next in Section 6.1. The basic idea behind atomic variables is to implement common access patterns occurring in isolated statements as predefined methods with implementations that avoid the use of locks. Thus, atomic variables provided a restricted solution to scalable implementations of isolated. If an isolated statement matches an available pattern, then it can be
implemented by using an atomic variable; otherwise, the default single-lock implementation of isolated has to be used instead.

When comparing implementations of isolated statements, the three cases to consider in practice can be qualitatively described as follows:

1. **Low contention:** In this case, isolated statements are executed infrequently, and a single-lock approach as in HJ is often the best solution. Other solutions, such as TM and atomic variables, incur additional overhead compared to the single-lock approach because of their book-keeping and checks necessary but there is no visible benefit from that overhead because contention is low.

2. **Moderate contention:** In this case, the serialization of all isolated statements in a single-lock approach limits the performance of the parallel program due to Amdahl’s Law, but a finer-grained approach that only serializes interfering isolated statements results in good scalability. Atomic variables usually shine in this scenario since the benefit obtained from reduced serialization far outweighs any extra overhead incurred.

3. **High contention:** In this case, there are phases in the program when interfering isolated statements dominate the program execution time. In such situations, approaches such as TM and atomic variables are of little help since they cannot eliminate the interference inherent in the program. The best approach in such cases is to find an alternative approach to isolated e.g., to use a parallel Array Sum algorithm instead of an isolated statement to compute the sum of values generated by different tasks.

### 6 Atomic Variables

#### 6.1 AtomicInteger and AtomicIntegerArray

The java.util.concurrent package [27][18] (also referred to as j.u.c) was introduced over five years ago as part of Java 5.0 to offer a standard set of library utilities for writing parallel programs in Java. You will learn more details of the j.u.c. package later in the class, when you graduate from parallel programming in HJ to parallel programming in pure Java. The use of many j.u.c. features by the programmer is prohibited in HJ, since they can easily violate HJ’s properties such as deadlock freedom.

However, there are two groups of utilities in j.u.c. that can be freely used in HJ code for performance and convenience — atomic variables and concurrent collections [18]. This lecture focuses on the former, which is embodied in the java.util.concurrent.atomic sub-package [6][27]. Specifically, the j.u.c.atomic sub-package provides library calls that can be used by HJ programmers as a more efficient substitute for certain patterns of isolated statements. It does so by encapsulating a single variable in an object, and providing certain read, write and read-modify-write operations that are guaranteed to be performed “atomically” on the object i.e., as though they occurred in an isolated statement.

The operations of interest for two j.u.c. atomic classes, AtomicInteger and AtomicIntegerArray, are summarized in Table. Let us start with AtomicInteger. It has two constructors, AtomicInteger() (with a default initial value value of 0) and AtomicInteger(init) (with a specified initial value). A single instance of AtomicInteger encapsulates an object with a single integer field, val, that can only be read or written using predefined methods, as shown in Table. Each such method call is guaranteed to execute in isolation with other methods invoked on the same object. To use these classes in HJ, you will need to include the following statement at the start of your program, import java.util.concurrent.atomic.*.

Table shows equivalent HJ isolated statements for AtomicInteger methods get(), set(), getAndSet(), addAndGet(), getAndAdd(), and compareAndSet(). While the functionality of your HJ program will remain unchanged if you use AtomicInteger methods as in column 2 or isolated statements as in column 3, the performance of your program in the moderate contention case will usually be superior when using

---

1 The HJ implementation uses j.u.c. features to implement parallel constructs that you have already learned, such as async, finish, future and phaser, but this usage is not visible to the HJ programmer.
<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicInteger</td>
<td>int j = v.get();</td>
<td>int j; isolated j = v.val;</td>
</tr>
<tr>
<td></td>
<td>v.set(newVal);</td>
<td>isolated v.val = newVal;</td>
</tr>
<tr>
<td>AtomicInteger</td>
<td>int j = v.getAndSet(newVal);</td>
<td>int j; isolated { j = v.val; v.val = newVal; }</td>
</tr>
<tr>
<td>// init = 0</td>
<td>int j = v.addAndGet(delta);</td>
<td>isolated { v.val += delta; j = v.val; }</td>
</tr>
<tr>
<td>AtomicInteger</td>
<td>int j = v.getAndAdd(delta);</td>
<td>isolated { j = v.val; v.val += delta; }</td>
</tr>
<tr>
<td>(init)</td>
<td>boolean b =</td>
<td>boolean b; isolated</td>
</tr>
<tr>
<td></td>
<td>v.compareAndSet</td>
<td>if (v.val==expect) {v.val=update; b=true;}</td>
</tr>
<tr>
<td></td>
<td>(expect,update);</td>
<td>else b = false;</td>
</tr>
<tr>
<td>AtomicIntegerArray</td>
<td>int j = v.get(i);</td>
<td>int j; isolated j = v.arr[i];</td>
</tr>
<tr>
<td></td>
<td>v.set(i,newVal);</td>
<td>isolated v.arr[i] = newVal;</td>
</tr>
<tr>
<td>(length) // init = 0</td>
<td>int j = v.getAndSet(i,newVal);</td>
<td>int j; isolated { j = v.arr[i]; v.arr[i] = newVal; }</td>
</tr>
<tr>
<td></td>
<td>int j = v.addAndGet(i,delta);</td>
<td>isolated { v.arr[i] += delta; j = v.arr[i]; }</td>
</tr>
<tr>
<td></td>
<td>int j = v.getAndAdd(i,delta);</td>
<td>isolated { j = v.arr[i]; v.arr[i] += delta; }</td>
</tr>
<tr>
<td></td>
<td>boolean b =</td>
<td>boolean b; isolated</td>
</tr>
<tr>
<td></td>
<td>v.compareAndSet</td>
<td>if (v.arr[i]==expect) {v.arr[i]=update; b=true;}</td>
</tr>
<tr>
<td></td>
<td>(i,expect,update);</td>
<td>else b = false;</td>
</tr>
</tbody>
</table>

Table 1: Methods in `java.util.concurrent.atomic` classes `AtomicInteger` and `AtomicIntegerArray` and their equivalent HJ `isolated` statements. Variable `v` refers to a `j.u.c.atomic` object in column 2 and to an equivalent non-atomic object in column 3. `val` refers to a field of type `int`, and `arr` refers to a field of type `int[]`.

1) Rank computation:

---

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2) Work assignment:

---

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3) Counting semaphore:

---

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4) Sum reduction:

---

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Examples of common `isolated` statement idioms and their equivalent `AtomicInteger` implementations.
<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicReference</td>
<td>Object o = v.get();</td>
<td>Object o; isolated o = v.ref;</td>
</tr>
<tr>
<td></td>
<td>v.set(newRef);</td>
<td>isolated v.ref = newRef;</td>
</tr>
<tr>
<td>AtomicReference()</td>
<td>Object o = v.getAndSet(newRef);</td>
<td>Object o; isolated { o = v.ref; v.ref = newRef; }</td>
</tr>
<tr>
<td>// init = null</td>
<td>boolean b = v.compareAndSet(expect,update);</td>
<td>boolean b; isolated</td>
</tr>
<tr>
<td></td>
<td>if (v.ref==expect) {v.ref=update; b=true;}; else b = false;</td>
<td></td>
</tr>
<tr>
<td>AtomicReference(init)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Object o = v.get();</td>
<td>Object o; isolated o = v.arr[i];</td>
</tr>
<tr>
<td></td>
<td>v.set(i,newRef);</td>
<td>isolated v.arr[i] = newRef;</td>
</tr>
<tr>
<td>AtomicReferenceArray (length)</td>
<td>Object o = v.get();</td>
<td>Object o; isolated o = v.arr[i];</td>
</tr>
<tr>
<td>// init = null</td>
<td>v.set(i,newRef);</td>
<td>isolated v.arr[i] = newRef;</td>
</tr>
<tr>
<td>AtomicReferenceArray (arr)</td>
<td>Object o = v.get();</td>
<td>Object o; isolated o = v.arr[i];</td>
</tr>
<tr>
<td></td>
<td>v.set(i,newRef);</td>
<td>isolated v.arr[i] = newRef;</td>
</tr>
<tr>
<td>AtomicIntegerArray (arr)</td>
<td>Object o = v.get();</td>
<td>Object o; isolated o = v.arr[i];</td>
</tr>
<tr>
<td></td>
<td>v.set(i,newRef);</td>
<td>isolated v.arr[i] = newRef;</td>
</tr>
<tr>
<td>Table 3: Methods in java.util.concurrent atomic classes AtomicReference and AtomicReferenceArray and their equivalent HJ isolated statements. Variable v refers to a j.u.c.atomic object in column 2 and to an equivalent non-atomic object in column 3. ref refers to a field of type Object, and arr refers to a field of type Object[].</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AtomicInteger methods for the reasons discussed in Section 5 The complete list of AtomicInteger methods can be found in [3]. The methods omitted from Table 1 include decrementAndGet(), getAndDecrement(), incrementAndGet(), and getAndIncrement() (since they are equivalent to the add methods in Table 1 with delta=1 or delta=-1), doubleValue(), floatValue() and intValue() (since their functionality can be obtained by simple cast operations), and weakCompareAndSet() (since it is identical to compareAndSet() in HJ, but differs from compareAndSet() in some fine points of the Java Memory Model that are not observed on current hardware). While the isolated patterns supported by the AtomicInteger methods in Table 1 may appear to be limited, they capture idioms that occur frequently in parallel programs as shown in Table 2. In addition, variants of the addAndGet() and getAndAdd() methods have been studied extensively for the last 30 years [20, 19] as primitives for building scalable parallel algorithms.

Table 4 also shows constructors and methods for the AtomicIntegerArray class. The main advantage of using an instance of AtomicIntegerArray instead of creating an array of AtomicInteger’s is that an AtomicIntegerArray instance occupies less space since its book-keeping overhead is amortized over an entire integer array, whereas a AtomicInteger[] array is essentially an array of objects. An element access in AtomicInteger[] also incurs an extra indirection relative to an element access in AtomicIntegerArray.

The j.u.c.atomic sub-package also includes AtomicLong, AtomicLongArray, and AtomicBoolean classes, with methods that are easy to understand once you know the methods available in AtomicInteger and AtomicIntegerArray.

### 6.2 AtomicReference

It is useful to perform atomic operations on object references, in addition to atomic operations on the primitive data types outlined in Section 6.1. Table 3 summarizes the operations available for AtomicReference and AtomicReferenceArray classes in the j.u.c.atomic sub-package. The compareAndSet() method can be especially useful in practice. As an example, consider the code in Listing 6 with an isolated statement studied in Lecture 10 for the Parallel Spanning Tree example.
The `isolated` statement in line 5 of Listing 6 can be replaced by a `compareAndSet()` method if `parent` is stored as an `AtomicReference`, as shown above in Listing 7. There are additional J.u.c.atomic classes available called `AtomicMarkableReference` and `AtomicStampedReference` that support atomic manipulation of a reference+boolean pair or a reference+int pair respectively. Unfortunately, there is no pre-defined method that can support atomic manipulation of multiple objects as in the `isolated` statement in Listing 1.

### 7 Java Collections Framework

One of the biggest successes in object-oriented programming has been the creation of **frameworks** that encapsulate reusable code in different domains. A foundational domain that has received a lot of attention in this regard is **collection data structures**. A collection (container) is a data structure that “collects” multiple element objects into a single group. The Java Collections Framework \[14\] includes a number of collection classes, some of which you may have already encountered when learning how to write sequential programs in Java (e.g., Vector, ArrayList, Stack, TreeSet, and HashMap). These classes implement interfaces such as List, Queue, Set, SortedSet, and Map.

While the original Java Collections Framework (JCF) has been very successful for sequential programs, they pose a number of limitations for parallel programs. Many of the original JCF classes attempted to be “thread-safe” by ensuring that each public method in the class is “synchronized” i.e., acquires and releases a single lock on the collection each time a public method is invoked on the collection. Though simple, the one-lock-per-collection approach proved to be limiting in multiple ways — it did not offer a convenient way to lock multiple collections at the same time, it ended up serializing more computations than needed (thereby exacerbating the effect of Amdahl’s Law), and it did not provide a clean deadlock avoidance guarantee as with HJ `isolated` statements.

Instead, it is more desirable to provide collection classes that are designed for concurrency up-front, and can provide the simplicity of `isolated` semantics coupled with the performance and scalability advantages akin to those offered by Java atomic variables. The java.util.concurrent collection classes \[27\] \[18\] address this requirement. We will discuss three of these classes in today’s lecture, ConcurrentHashMap, ConcurrentLinkedQueue and CopyOnWriteArraySet. All operations in these three classes are nonblocking and can be freely used in any HJ program, just like the J.u.c.atomic.* classes. Note that there
are many j.u.c. classes that include blocking operations, and should not be used in an HJ program e.g., 
ArrayBlockingQueue, CountDownLatch, CyclicBarrier, DelayQueue, Exchanger, FutureTask, LinkedBlockingQueue, 
Phaser PriorityBlockingQueue, Semaphore, and SynchronousQueue. However, most of these classes pro-
vide functionalities that are already available in HJ e.g., CyclicBarrier, Phaser, CountDownLatch, and 
Semaphore are all subsumed by HJ phasers.

7.1 java.util.concurrent.ConcurrentHashMap

The Map interface in Java [7] is used to define key-value mappings for a given domain. It includes basic 
operations such as put(key, value), get(key) and remove(key). There are many applications where it is 
important to keep track of key-value mappings including software caches (where a program stores key-value 
maps obtained from an external source such as a database), dictionaries, and even sparse arrays.

There are multiple implementations for the Map interface available in a Java runtime environment. The 
HashMap class is completely unsynchronized and assumes that all necessary synchronization is handled by 
the caller. In the absence of detailed knowledge of the internals of the HashMap implementation, any client 
will likely play safe and ensure that an entire operation such as put or get is performed in an isolated 
statement or while some lock is held. However, these operations may take more than constant time due 
to traversals of multiple objects and the need to call an equals() method on each object traversed. As a 
result, this approach could add significantly to the serialization in a parallel program.

The j.u.c.ConcurrentHashMap class package implements the Map interface with more parallelism (less serial-
ization) in a parallel context than the approaches discussed above. In fact, it implements the ConcurrentMap 
sub-interface of Map which includes four additional methods — putIfAbsent(key, value), remove(key, value), 
replace(key, value), and replace(key, oldValue, newValue).

An instance of ConcurrentHashMap permits retrieval operations, such as get(), to overlap with update 
operations, such as put() and remove(). The result of a get() operation will depend on which update 
operations have completed prior to the start of the get(). Thus, single-element operations such as put(), 
get(), and remove() all appear to be atomic with respect to each other.

However, ConcurrentHashMap also supports aggregate operations such as putAll() and clear(). It is not 
practical to make these operations appear atomic in a parallel context since they involve multiple elements.
As a result, a single-element operation may observe an intermediate state where only some of the entries 
being inserted by a putAll() operation have been inserted, or where only some of the entries being removed 
by a clear() operation have been removed.

Given the widespread use of multithreaded and parallel applications in mainstream software, the ConcurrentHashMap 
is increasing in popularity. Listing 8 shows a sample usage of ConcurrentHashMap from an open source 
project.

```java
public abstract class BaseDirectory extends BaseItem implements Directory {

    Map files = new ConcurrentHashMap();

    public Map getFiles() {
        return files;
    }

    public boolean has(File item) {
        return getFiles().containsValue(item);
    }

    public Directory add(File file) {
        String key = file.getName();
        if (key == null) throw new Error( . . . );
        getFiles().put(key, file);
        return this;
    }

    public Directory remove(File item) throws NotFoundException {
```
if (has(item)) {
    getFiles().remove(item.getName());
    ...
} else throw new NotFoundException("can't remove unrelated item");


7.2 java.util.concurrent.ConcurrentLinkedQueue

The Queue interface in Java [8] defines a collection of elements typically inserted and removed in FIFO (first-in-first-out) order, using operations such as add() and poll(). The j.u.c.ConcurrentLinkedQueue class implements the Queue interface with more parallelism (less serialization) than a synchronized queue. Consistent with FIFO order, new elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

There is no bound on the size of a ConcurrentLinkedQueue, so an add(o) operation for a non-null object always succeeds (assuming that the program does not run out of memory). offer(o) is an alternative name for add(o).

The semantics for removing an object from a queue leads to some interesting possibilities. There is a remove(o) operation that attempts to remove a specified object and returns null if the object is in the queue. The ConcurrentLinkedQueue class also has a poll() operation that returns null if the queue is empty and removes the head of the queue otherwise. This is in contrast to the LinkedBlockingQueue.take() operation which blocks when the queue is empty.

Like ConcurrentHashMap, ConcurrentLinkedQueue is also widely used in practice. Listing 9 shows a sample usage of ConcurrentLinkedQueue from an open source project.

class BufferPool15Impl implements BufferPool.BufferPoolAPI {
    protected int maxSize;
    protected AtomicInteger size = new AtomicInteger(0);
    protected ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
    ...
    public XByteBuffer getBuffer(int minSize, boolean discard) {
        XByteBuffer buffer = (XByteBuffer) queue.poll();
        if (buffer != null) size.addAndGet(-buffer.getCapacity());
        else if (buffer == null) buffer = new XByteBuffer(minSize, discard);
        return buffer;
    }
    public void returnBuffer(XByteBuffer buffer) {
        if ((size.get() + buffer.getCapacity()) <= maxSize) {
            size.addAndGet(buffer.getCapacity());
            queue.offer(buffer);
        }
    }
}


7.3 java.util.concurrent.CopyOnWriteArraySet

Like a set in Mathematics, the Set interface in Java [11] defines a collection of elements that is prohibited from containing any duplicates, as defined by the equals() method on the objects in the collection. The
j.u.c.CopyOnWriteArraySet class implements the Set interface with more parallelism (less serialization) than a synchronized set data structure by using an array-based data structure. Unlike ConcurrentHashMap and ConcurrentLinkedQueue, the CopyOnWriteArraySet class achieves thread safety by making copies of the set as needed. This makes it well suited for applications in which sets are not large, and read operations dominate update operations in frequency. This is because update operations such as add() and remove() involve making copies of the array. However, iterators can traverse array “snapshots” efficiently without worrying about changes during the traversal.

Listing 10 shows a sample usage of CopyOnWriteArraySet from an open source project.
8 The Actor Model

The Actor Model (AM) promotes a no-shared mutable state and an event-driven philosophy. It was first defined in 1973 by Carl Hewitt et al. during their research on Artificial Intelligent (AI) agents [23]. It was designed to address the problems that arise while writing distributed applications. Further work by Henry Baker [24], Gul Agha [13], and others added to the theoretical development of the AM. The AM is different from task parallelism in that it is primarily an asynchronous message-based concurrency model. An actor is the central entity in the AM that defines how computation proceeds. The key idea is to encapsulate mutable state and use asynchronous messaging to coordinate activities among actors.

8.1 Actors

An actor is defined as an object that has the capability to process incoming messages. Usually the actor has a mailbox, as shown in Figure 4, to store its incoming messages. Other actors act as producers for messages that go into the mailbox. An actor also maintains local state which is initialized during creation. Henceforth, the actor is only allowed to update its local state using data (usually immutable) from the messages it receives and intermediate results it computes while processing the message. The actor is restricted to process at most one message at a time. This allows actors to avoid data races and to avoid the need for synchronization as there is no other actor contending for access to its local data. There is no restriction on the order in which the actor decides to process incoming messages. As an actor processes a message, it is allowed to change its behavior that affects how it processes the subsequent messages.

An actor has a well-defined life cycle and restrictions on the actions it performs in the different states. During its life cycle an actor is in one of the following three states:

- **new**: An instance of the actor has been created; however, the actor is not yet ready to receive or process messages.
Figure 4: Actors store incoming messages in a mailbox, maintain a local state which is not directly exposed to other actors, and process at most one message at a time.

- **started**: An actor moves to this state from the new state when it has been started using the `start` operation. It can now receive asynchronous messages and process them one at a time. While processing a message, the actor should continually receive any messages sent to it without blocking the sender.

- **terminated**: The actor moves to this state from the started state when it has been terminated and will not process any messages in its mailbox or new messages sent to it. An actor signals termination by using the `exit` operation on itself while processing some message.

Figure 5: Actors have a simple life cycle. The most interesting state is started which is where the actor is receiving and processing messages.

An actor interacts with other actors in two ways as shown in Figure 6. Firstly, it can send and receive messages to and from other actors. The sending and receiving of messages is done asynchronously, i.e. the sending actor can deliver a message without waiting for the receiving actor to be ready to process the message. An actor learns about the existence of other actors by either receiving their addresses in incoming messages or during creation. This brings us to the second manner of actor interaction: an actor can create new actors. This new actor can have its local state initialized with information from the parent actor. It is important to note that the network of actors an actor knows about can grow dynamically thus allowing formation of arbitrary connection graphs among actors and a wide range of communication and coordination patterns between them. In summary, while processing a message an actor may perform a finite combination of the following steps:

1. Asynchronously send a message to another actor whose address is known;

2. Create a new actor providing all the parameters required for initialization;
3. Become another actor, which specifies the replacement behavior to use while processing the subsequent messages.

Figure 6: During the processing of a message, actor interactions include exchanging messages with other actors and creating new actors.

8.2 Desirable Properties

The only way an actor conveys its internal state to other actors is explicitly via messages and responses to messages. This property obtains benefits similar to encapsulation in object-oriented programming and encourages modularity. The encapsulation of the local state also helps prevent data races because only the actor can modify its local state. Due to the asynchronous mode of communication, the lack of restriction on the order of processing messages sent from different actors, and the absence of synchronization via encapsulation of local data, actors expose inherent concurrency and can work in parallel with other actors.

8.3 Habanero-Java Actors

Habanero-Java (HJ) actors are defined by extending the `hj.lang.Actor` base class. Concrete sub-classes are required to implement the method used to process messages. This method is named `process()` in HJ light actors. Actors are like other objects and can be created by a `new` operation on concrete classes. An actor is activated by the `start()` method, after which the runtime ensures that the actor’s message processing method is called for each message sent to the actor’s mailbox. The actor can terminate itself by calling the `exit()` method while processing a message. Messages can be sent to actors from actor code or non-actor code by invoking the actor’s `send()` method using a call as follows, `someActor.send(aMessage)`. A `send()` operation is non-blocking and the recipient actor processes the message asynchronously. The HJ runtime preserves the order of messages with the same sender task and receiver actor, but messages from different senders may be interleaved in an arbitrary order. This is similar to the message ordering provided by ABCL.

All async tasks created internally within an actor are registered on the `finish` scope that contained the actor’s `start()` operation. The `finish` scope will block until all actors started within it terminate. This is similar to the `finish` semantics while dealing with `asyncs`.

```java
import hj.lang.Actor;
```
Listing 11: HelloWorld using HJ actors

Listing 11 shows a `HelloWorld` example using HJ actors. We are guaranteed ordered sends, i.e. though `Hello` and `World` will be processed asynchronously, they will be processed in that order.
Listing 12: A simple pipeline using HJ actors

Listing 12 shows an example of implementing a static pipeline using HJ actors. The pipeline is built up of three stages:

- **EvenLengthFilter**: The first stage of the pipeline filters out strings which are of odd length. It forwards strings of even length to the next stage in the pipeline.

- **LowerCaseFilter**: The second stage of the pipeline converts all the input strings to lowercase before forwarding to the third (and final) stage of the pipeline.

- **PrintStage**: The final stage of the pipeline prints the input strings. These strings are of even length and in lowercase due to the processing in the earlier stages of the pipeline.
8.3.1 Tips and Pitfalls

- Use an actor-first approach when designing programs that use actors i.e., think about which actors need to be created and how they will communicate with each other. This step will also require you to think about the communication objects used as messages.

- If possible, use immutable objects for messages, since doing so avoids data races and simplifies debugging of parallel programs.

- When overriding the start() or exit() methods in actor classes, remember to make the appropriate calls to the parent’s implementation with super.start() or super.exit(), respectively.

- The HJ actor start() method is not idempotent. Take care to ensure you do not invoke start() on the same actor instance more than once. The exit() method on the other hand is idempotent, invoking exit() multiple times is safe within the same call to process().

- Always remember to terminate a started actor using the exit() method. If an actor that has been started is not terminated, the enclosing finish will wait forever (deadlock).

9 Linearizability of Concurrent Objects

A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads. Concurrent objects were originally referred to as monitors [25], and are also informally referred to as “thread-safe objects”. For simplicity, it is usually assumed that the body of each method in a concurrent object is itself sequential i.e., it does not create child async tasks. A classical approach to making a class thread-safe (that dates back to the original concept of monitors) is to enclose each of its methods in a critical section to guarantee mutual exclusion. However, this approach leads to scalability bottlenecks since it effectively serializes all method invocations on the same object.

In the previous lecture, you were introduced to three java.util.concurrent collection classes — ConcurrentHashMap, ConcurrentLinkedQueue and CopyOnWriteArraySet. Instances of these classes are concurrent objects that are designed for high scalability in the presence of method invocations from parallel tasks. In addition, these classes ensure that all their methods are nonblocking. In general, it is very challenging to design and implement concurrent objects with these scalability and nonblocking properties, while retaining semantics equivalent to that of monitors.

In today’s lecture, you will learn about an important correctness property of concurrent objects called linearizability [22][21], which guarantees semantics equivalent to that of methods in critical sections.

A formal definition of linearizability can be found in [22][21]. An informal definition is as follows:

1. A linearizable execution is one in which the semantics of a set of method calls performed in parallel on a concurrent object is equivalent to that of some legal linear sequence of those method calls.

2. A linearizable concurrent object is one for which all possible executions are linearizable.

To understand this definition of linearizability, consider a canonical example of a concurrent object — a FIFO (First In, First Out) queue. A FIFO queue, q, typically supports two methods: q.enq(o) inserts object o at the tail of the queue and q.deq() removes and returns the item at the head of the queue. The q.deq() method call will typically throw an exception (e.g., EmptyException) if the queue is empty. For simplicity, let us assume that we have unbounded space to store the queue. If (as in real-world situations) we have a bounded amount of space for the queue, then the q.enq(o) method call will throw an exception if the queue is full.

Consider two async tasks, A and B, repeatedly performing enq() and deq() operations on the FIFO queue. Table 4 shows one possible execution of a monitor-based implementation of a FIFO queue using critical
Table 4: Example of a linearizable execution of method calls on a monitor-based implementation of FIFO queue \( q \) by parallel tasks \( A \) and \( B \)

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke ( q.\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Work on ( q.\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Work on ( q.\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Return from ( q.\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Invoke ( q.\text{enq}(y) )</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Work on ( q.\text{enq}(y) )</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Work on ( q.\text{enq}(y) )</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Return from ( q.\text{enq}(y) )</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Invoke ( q.\text{deq}() )</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Return ( x ) from ( q.\text{deq}() )</td>
</tr>
</tbody>
</table>

sections. The critical sections ensure that none of the \( \text{enq}() \) and \( \text{deq}() \) methods run in parallel with each other. This is a linearizable execution because its semantics is equivalent to the following sequence:

\[
q.\text{enq}(x) \quad ; \quad q.\text{enq}(y) \quad ; \quad q.\text{deq}() \quad \text{returns} \quad x
\]

Now consider a concurrent (rather than a monitor-based) implementation of a FIFO queue, such as Java’s \texttt{ConcurrentLinkedQueue}. One possible parallel execution is shown as a schedule in Table 5. This is a linearizable execution because its semantics is equivalent to the following sequence:

\[
q.\text{enq}(x) \quad ; \quad q.\text{enq}(y) \quad ; \quad q.\text{deq}() \quad \text{returns} \quad x
\]

In fact, the execution in Table 5 would also be linearizable if the \( q.\text{deq}() \) operation returned \( y \) instead of \( x \) at time \( t = 5 \) in Table 4. In this case, its semantics would be equivalent to the following sequence:

\[
q.\text{enq}(y) \quad ; \quad q.\text{enq}(x) \quad ; \quad q.\text{deq}() \quad \text{returns} \quad y
\]

We have the flexibility of considering different sequences for the schedule in Table 5 because of the parallel overlap between operations \( q.\text{enq}(x) \) and \( q.\text{enq}(y) \).

Another way of understanding linearizability is that each method call of a linearizable concurrent object should appear to take effect “instantaneously” at some point in time between its invocation and return. For the schedule in Table 5, we can assume that (say) \( q.\text{enq}(x) \) took effect at time \( t = 1 \) and \( q.\text{enq}(y) \) took effect at time \( t = 2 \). For the alternate schedule, we can assume that (say) \( q.\text{enq}(x) \) took effect at time \( t = 3 \) and \( q.\text{enq}(y) \) took effect at time \( t = 2 \). We can assume that \( q.\text{deq}() \) took effect at time \( t = 4 \) in both cases. For the monitor-based implementation in Table 4, we have no choice but to pick times consistent with the \( q.\text{enq}(x) \) ; \( q.\text{enq}(y) \) ; \( q.\text{deq}() \) sequence.

Now, consider the parallel execution shown in Table 6. In this case, there is no parallel overlap between \( q.\text{enq}(x) \) and \( q.\text{enq}(y) \) since the return from \( q.\text{enq}(x) \) precedes the invocation of \( q.\text{enq}(y) \). Thus, \( q.\text{enq}(x) \) must precede \( q.\text{enq}(y) \) in all linear sequences of method calls invoked on \( q \). It would be illegal for the next \( q.\text{deq}() \) operation to return \( y \). Hence, the execution in Table 6 is non-linearizable.

Even with this brief discussion, it should be clear that linearizability is a desirable correctness property for concurrent objects.
Table 5: Example of a linearizable execution of method calls on a concurrent FIFO queue \( q \) by parallel tasks \( A \) and \( B \) (source: Figure 1 in [22]).

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke ( q\text{.}\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Work on ( q\text{.}\text{enq}(x) )</td>
<td>Invoke ( q\text{.}\text{enq}(y) )</td>
</tr>
<tr>
<td>2</td>
<td>Work on ( q\text{.}\text{enq}(x) )</td>
<td>Return from ( q\text{.}\text{enq}(y) )</td>
</tr>
<tr>
<td>3</td>
<td>Return from ( q\text{.}\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Invoke ( q\text{.}\text{deq}() )</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Return ( x ) from ( q\text{.}\text{deq}() )</td>
</tr>
</tbody>
</table>

Table 6: Example of a non-linearizable execution of method calls on a concurrent FIFO queue \( q \) by parallel tasks \( A \) and \( B \) (source: Figure 1 in [22]).

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke ( q\text{.}\text{enq}(x) )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Return from ( q\text{.}\text{enq}(x) )</td>
<td>Invoke ( q\text{.}\text{enq}(y) )</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Work on ( q\text{.}\text{enq}(y) )</td>
</tr>
<tr>
<td>3</td>
<td>Invoke ( q\text{.}\text{deq}() )</td>
<td>Return from ( q\text{.}\text{enq}(y) )</td>
</tr>
<tr>
<td>4</td>
<td>Work on ( q\text{.}\text{deq}() )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Return ( y ) from ( q\text{.}\text{deq}() )</td>
<td></td>
</tr>
</tbody>
</table>

10 Relating Linearizability to the Computation Graph model

How does the definition of linearizability in Section 9 relate to the Computation Graph (CG) model that we have studied in the class? Since each method executed on a concurrent object is assumed to be sequential, the method’s execution can be represented in the CG as a sequence of nodes (steps) connected by continue edges. Some of these nodes will be instances of isolated statements, and each such node may be the source and/or destination of serialization edges as a result.

Figure 7 shows a CG for the monitor-based FIFO execution shown in Table 4. Since it’s a monitor-based implementation, we assume that each \( \text{enq}() \) and \( \text{deq}() \) method is enclosed in an isolated statement. The \( i\text{-}\text{begin} \) and \( i\text{-}\text{end} \) nodes refer to the beginning and end of isolated statements, and \( q\text{.}\text{deq}():x \) refers to a dequeue operation that returns \( x \). As before, the CG contains continue edges and serialization edges to represent orderings within and across tasks.

![Figure 7: A Computation Graph for the execution in Table 4. \( i\text{-}\text{begin} \) and \( i\text{-}\text{end} \) refer to the beginning and end of isolated statements. \( q\text{.}\text{deq}():x \) refers to a dequeue operation that returns \( x \).](image-url)
Recall the following rule from Lecture 10 for adding serialization edges to a CG:

*Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each isolated step, S, that has already executed.*

That is why we have two serialization edges from Task A to Task B in Figure 7, even though the second edge is redundant (transitive) if we take the continue edges in Task B into account. (For simplicity, we omit serialization edges when the source and destination steps belong to the same task, so there is no serialization edge from the first i-end node to the second i-begin node in Task B.) It is clear from this CG that the only legal sequence for the operations on q in this execution is `q.enq(x);q.enq(y);q.deq():x`. However, different executions with different serialization edges could lead to sequences `q.enq(y);q.enq(x);q.deq():y` and `q.enq(y);q.deq():y;q.enq(x)`.

The concept of a method call taking effect “instantaneously” can be translated to the CG by creating a reduced method-level version of the CG as shown in Figure 8. This reduction can be defined as follows: take the sequence of CG nodes \( a_i \) corresponding to a method call of interest and “collapse” it into a single node \( b \) to obtain a new graph, CG’. Each edge with source \( a_i \) in CG is redirected to source \( b \) in CG’. Likewise, each edge with destination \( a_i \) in CG is redirected to destination \( b \) in CG’. Finally, if these redirections result in any self-loop edges of the form \( b \rightarrow b \) in CG’, those self-loop edges are removed from CG’.

Given a reduced CG, a *sufficient condition* for linearizability is that the reduced CG is acyclic as in Figure 8. This means that if the reduced CG does not contain a directed cycle, then the underlying execution must be linearizable. However, the converse is not necessarily true, as we will see.

Figure 9 shows a CG for the concurrent FIFO execution shown in Table 5. For this example, we assumed that the concurrent implementation split the `q.enq(x)` method into three steps — \( q.enq(x)_1 \), \( q.enq(x)_2 \) and \( q.enq(x)_3 \) — the first and third of which are enclosed in isolated statements. However, the `q.enq(y)` and the `q.deq():x` operations were implemented as single steps. Note that Figure 9 shows serialization edges that are consistent with the execution in Table 5.

Figure 10 shows the reduced method-level graph for the CG in Figure 9. This graph has a cycle because it contains edges in both directions between `q.enq(x)` and `q.enq(y)`. However, this execution is still linearizable, as discussed earlier. This confirms that it is not necessary that the reduced CG of a linearizable execution be acyclic.
Task A

Computation Graph

Task B

Method-level Reduced Graph

Figure 9: A Computation Graph for the execution in Table 5. \( \text{q.enq}(x)_1, \text{q.enq}(x)_2 \) and \( \text{q.enq}(x)_3 \) refer to three steps in the \( \text{q.enq}(x) \) method, the first and third of which are enclosed in isolated statements.

Figure 10: Reduced method-level graph for CG in Figure 9.
11 Closures, HJ Asyncs, and Java Runnable Objects

Now that you have learned the fundamentals of parallel programming over the course of 26 lectures, it is time to remove the “training wheels” and introduce you to parallel programming in the real world starting with Java threads. A unique aspect of Java compared to most of its predecessor mainstream programming languages (e.g., C, C++, Pascal, FORTRAN, PL/1, COBOL) is that Java included the notion of threads and synchronization as part of its language definition, whereas support for parallelism in the predecessor languages was achieved through extra-language mechanisms such as libraries, e.g., POSIX threads [29], Message-Passing Interface (MPI) [31], and Map-Reduce, or pseudo-comment “pragmas” e.g., OpenMP [16].

As we will see, Java essentially takes a library-based approach to parallelism. The key mechanisms used to communicate task definitions to Java Concurrency libraries are Runnable and Callable objects that provide support for a limited form of closures. Runnable objects are reviewed next in Section 11 and Callable objects in Section 13.

Recall that a closure is a first-class function with free variables that are bound in function’s lexical environment [3]. It is called a “closure” because it is “closed over” its free variables i.e., the values of the free variables are captured when the closure is created.

As an example, consider the following Scheme function, best-selling-books, which applies a filter function on an input book-list [3]. In the example included below, the function passed as filter’s first argument is a closure i.e., an anonymous lambda expression that compares a book’s sales with the threshold value obtained from the outer scope for the definition of function best-selling-books. The value of threshold is captured when the lambda expression is evaluated (but not applied). The lambda expression can be called (applied) multiple times by function filter:

```
; Return a list of all books with at least THRESHOLD copies sold.
(define (best-selling-books threshold)
  (filter
    (lambda (book)
      (>= (book-sales book) threshold))
    book-list))
```

In this example, the lambda expression, (lambda (book) (>= (book-sales book) threshold)) appears within the function, best-selling-books. When the lambda expression is evaluated, Scheme creates a closure consisting of the code for the lambda and a reference to the threshold variable, which is a free variable inside the lambda. The closure is then passed to the filter function, which calls it repeatedly to determine which books are to be added to the result list and which are to be discarded. Because the closure itself has a reference to threshold, it can use that variable each time filter calls it.

What is the relationship between closures and async statements? The body of an async is essentially a parameter-less closure that is both created and enabled for execution at the point when the async statement is executed. Though there are differences, an important similarity between an async statement and a closure is that an async also captures the values of free variables (local variables in outer scopes) when it is created.

Recall the HJ code below in Listing 13 that you learned in Lecture 1 for computing the sum of an array using two parallel async tasks:

```
1 // Start of Task T1 (main program)
2 sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
3 finish {
4    // Compute sum1 (lower half) and sum2 (upper half) in parallel
5    int len = X.length
6    async for (int i=0 ; i < len/2 ; i++) sum1 += X[i]; // Task T2
7    async for (int i=len/2 ; i < len ; i++) sum2 += X[i]; // Task T3
8 }
```
9 //Task T1 waits for Tasks T2 and T3 to complete
10 int sum = sum1 + sum2; // Continuation of Task T1

Listing 13: Two-way Parallel ArraySum in HJ

As in closures, the two async statements in lines 6 and 7 capture the value of local variable, len, from their outer scope.

Unfortunately, Java neither has closures nor async statements. However, the Runnable interface in Java embodies a convention used by programmers to create a limited form of closures in Java. Any instance of a class that implements the Runnable interface must contain a run() method with a void return type. As an example, the following Java code in Listing 14 shows how the body of the first async in line 6 of Listing 13 can be created as a Runnable instance:

```java
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 14: Example of creating a Java Runnable instance as a closure

The new constructor in line 3 creates an instance of an anonymous inner class defined in lines 3–7 and assigns a reference to that instance to variable r. It is an inner class because it is declared within an existing class, and it is anonymous because it has been declared without a name. This class will only be accessible only at the point where it is defined. It can access any local final variables in the scope that it’s defined, such as variable len in line 2. Unlike the HJ case, it is an error to access len in a Java anonymous inner class if len was not declared to be final.

Anonymous inner classes were originally intended for advanced Java users, and little attention was paid to simplifying their usage. However, a number of mainstream libraries for event handling (e.g., java.awt.EventQueue) and for parallel programming (java.util.concurrent) expect client code to create instances of Runnable, which is most conveniently achieved via anonymous inner classes. Thus, even though the syntax for anonymous inner classes is quite cryptic in Java, many Java programmers have to learn this syntax so that they can use these libraries.

12 Java Threads

A summary of the Java Threads functionality can be found in the definition of the standard java.lang.Thread class, which is defined as follows:

```java
public class Thread extends Object implements Runnable {
    Thread() { ... } // Creates a new Thread
    Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
    void run() { ... } // Code to be executed by thread
    // Case 1: If this thread was created using a Runnable object,
    // then that object’s run method is called
    // Case 2: If this class is subclassed, then the run() method
    // in the subclass is called
    void start() { ... } // Causes this thread to start execution
    void join() { ... } // Wait for this thread to die
```
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. The static method, Thread.currentThread(), returns a reference to the currently executing thread.

Threads have richer functionality than async tasks, but are also more expensive. In Lecture 11 (Table 1), we saw that the overhead of Java threads can be more than \(10 \times\) greater than that of async tasks implemented with work-sharing schedulers, and more than \(1000 \times\) greater than that of async tasks implemented with work-stealing schedulers. As a result, it is usually not practical to create large numbers of Java threads in a single program execution.

There are two ways to specify the code to be executed by a Java thread:

1. Define a class that implements the Runnable interface and pass an instance of that class to the Thread constructor in line 3 of Listing 15. It is common to create an instance of an anonymous inner class that implements Runnable for this purpose, as discussed in Section 11. In this case, the Runnable instance defines the work to be performed, and the Thread instance identifies the worker that will perform the work.

2. Subclass Thread and override the run() method. This is usually inconvenient in practice.

When an instance of Thread is created, it does not start executing right away. Instead, it can only start 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().

The operation analogous to finish is join(). A t.join() call forces the invoking thread to wait till thread t completes. This is a lower-level primitive than finish since it only waits for a single thread rather than a collection of threads. Further, there is no restriction on which thread performs a join on which thread, so it is easy for a programmer to erroneously create a deadlock cycle with join operations. Likewise, there is no analog of an Immediately Enclosing Finish for Java threads.

Putting it all together, Listing 16 shows a Java threads version of the HJ code in Listing 13. It is instructive to compare the two versions and see the differences. There are three steps in initiating a thread to execute a computation analogous to async. The first step is to create a Runnable instance as in lines 5–7 for r1 and lines 10–12 for r2. The bodies of the run() methods in lines 6 and 11 specify the work that was done in the two async statements in Listing 13. The next step is to create a Thread instance as in lines 8 and 13 for t1 and t2. The third step is to start the threads as in line 9 and 14. After the threads have been created, the parent waits for their completion by performing the join() calls in line 16.
13 Using Callable objects with Java threads

In the previous section, you learned how the `start()` and `join()` operations in the Java `Thread` class can be used to support the functionality of `async` and `finish` statements (albeit with greater overhead). With some additional steps, this approach can be extended to also support `future` tasks. It builds on the `Callable` interface introduced in Java 5 [1]. Any instance of a class that implements the `Callable<V>` interface must contain a `call()` method with a `V` return type. As a simple example of using `Callable`, consider the following sequential Java code in Listing 17:

```
1 ImageData image1 = imageInfo.downloadImage(1);
2 ImageData image2 = imageInfo.downloadImage(2);
3 . . .
4 renderImage(image1);
5 renderImage(image2);
```

Listing 17: HTML renderer in Java before decomposition into Callable tasks

This code can be rewritten as follows (Listing 18) by encapsulating the two calls to `downloadImage()` in `Callable` objects in lines 1–4. As a result, the actual invocation of these calls is postponed to lines 6 and 7 when the `call()` method is invoked on the `Callable` objects.

```
1 Callable<ImageData> c1 = new Callable<ImageData>() {
   public ImageData call() { return imageInfo.downloadImage(1); };
};
2 Callable<ImageData> c2 = new Callable<ImageData>() {
   public ImageData call() { return imageInfo.downloadImage(2); };
};
3 . . .
4 renderImage(c1.call());
5 renderImage(c2.call());
```

Listing 18: HTML renderer in Java after decomposition into Callable tasks

However, this use of `Callable` objects does not directly lead to parallelism because there is no `Thread` constructor that takes a `Callable` object as input. Instead, we need to wrap the `Callable` object in an object that implements the `Runnable` interface. A convenient adapter class for this purpose is `FutureTask<V>` [4]. It has a constructor, `FutureTask(Callable<V> c)`, that takes a `Callable` instance as a parameter. With this adapter, the following sequence of steps can be used to execute a `Callable` object in a separate thread:

1. Create a parameter-less callable closure using a statement like
   "Callable<Object> c = new Callable<Object>() {public Object call() { return ...; }}; ".
2. Encapsulate the closure as a task using a statement like
   "FutureTask<Object> ft = new FutureTask<Object>(c);".
3. Start executing the task in a new thread by issuing the statement, "new Thread(ft).start();".
4. Wait for the task to complete, and get its result by issuing the statement, “Object o = ft.get();”.

These steps can be used to transform the code to Listing 18 to a parallel version shown below in Listing 19:

```java
1 Callable<ImageData> c1 = new Callable<ImageData>() {
2   public ImageData call() { return imageInfo.downloadImage(1); }
3 };
4 new FutureTask<Object>(c1).start();
5 Callable<ImageData> c2 = new Callable<ImageData>() {
6   public ImageData call() { return imageInfo.downloadImage(2); }
7 };
8 new FutureTask<Object>(c2).start();
9 ...
10 renderImage(ft1.get());
11 renderImage(ft2.get());
```

Listing 19: HTML renderer in Java after parallelization of Callable tasks

Note that the above code in Listing 19 is just equivalent to the following HJ code in Listing 20 using future tasks (other than the fact that Thread creation in Java is more expensive than task creation in HJ):

```java
1 future<ImageData> ft1 = async<ImageData>{ return imageInfo.downloadImage(1); }
2 future<ImageData> ft2 = async<ImageData>{ return imageInfo.downloadImage(2); }
3 ...
4 renderImage(ft1.get());
5 renderImage(ft2.get());
```

Listing 20: Equivalent HJ code for the parallel Java code in Listing 19

14 Introduction to Java’s synchronized statement

The previous sections showed how async and future tasks in HJ can be implemented in standard Java using threads and instances of Runnable and Callable objects. Now let us consider how to implemented HJ’s isolated statement. You have already learned how the java.util.concurrent.atomic package can be used to implement special cases of isolated statements. The key mechanism that can be used to implement more general forms of isolated statements (including critical sections) is locks. The basic idea behind the use of locks is to implement a statement such as isolated <stmt> as follows:

1. Acquire lock \( L_i \)
2. Execute <stmt>
3. Release lock \( L_i \)

The responsibility for ensuring that the choice of locks correctly implements the semantics of isolated falls entirely on the shoulders of the programmer. The main guarantee provided by locks is that only one thread can hold a lock at a time, and the thread is blocked if the lock is unavailable.

The Java language attempts to simplify the use of locks by providing the following built-in support:

- Every Java object and Java class has an associated lock which can be accessed by using the synchronized statement.
• The statement, `synchronized(p) <stmt>` acquires the lock for object `p`, executes `<stmt>`, and releases the lock for object `p`. (The lock is released even if an exception is thrown in `<stmt>`. There is no constraint that the data accesses in `<stmt>` be confined to members of object `p`.

• A method can also be declared to be `synchronized`. If it is a virtual method, then the method is executed as if its body was enclosed in a “`synchronized (this)`” statement i.e., it is guarded by the lock for the underlying object. If it is a static method, then it is executed as if its body was enclosed in a “`synchronized (A.class)`” statement i.e., it is guarded by the lock for the underlying class (`A`).

• `synchronized` statements may be nested.

• Reentrant (“recursive”) locking is permitted if two nested `synchronized` statements executed in the same thread attempt to acquire the same lock.

References


[28] Rashid Bin Muhammad. Trees. URL http://www.personal.kent.edu/~rmuhamma/GraphTheory/MyGraphTheory/trees.htm


