COMP 322: Fundamentals of Parallel Programming Module 2: Concurrency

March 6, 2017

DRAFT VERSION – PLEASE DO NOT DISTRIBUTE

Contents

5 Mutual Exclusion

5.1 Critical Sections and the "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.

```
 1  class DoublyLinkedList {
 2    DoublyLinkedList prev, next;
 3    . . .
 4    void delete() {
 5      isolated { // start of mutual exclusion region (critical section)
 6        if (this.prev != null) this.prev.next = this.next;
 7        if (this.next != null) this.next.prev = this.prev
 8      } // end of mutual exclusion region (critical section)
 9      . . . // additional work to delete node (mutual exclusion not needed)
10    }
11    . . .
12  }
13  . . .
14  static void deleteTwoNodes(DoublyLinkedList L) {
15    finish {
16      async L.delete();
17      async L.next.delete();
18    }
19  }
```

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* [4]. The following definition of critical sections from [16] 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 construct, 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 throw 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`, and `isolated` constructs can create a *deadlock cycle* among tasks.

5.1.1 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) [12]. 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.,* [3].

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, object-based isolation (Section 5.2), and atomic variables (Section 5.4.1), 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. This is the case that motivates the use of approaches such as TM, object-based isolation, and atomic variables, since the the benefit that they offer from reduced serialization outweighs the extra overhead that they incur.

3. *High contention:* In this case, there are phases in the program when interfering `isolated` statements on a single object (often referred to as a "hot spot" object) 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 accessing a single object. 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.

5.2 Object-based Isolation

In this section, we introduce the basic functionality of object-based isolation [11]. As mentioned earlier, the focus of object-based isolation is on mutual exclusion rather than strong atomicity *i.e.,* mutual exclusion is only guaranteed between instances of `isolated` statements, unlike strong atomicity where a mutual exclusion guarantees may also exist between `atomic` and non-`atomic` statements. Two given `isolated` statements execute in mutual exclusion if the intersection of their object list is non-empty. Further, object-based `isolated` statements can be combined with global `isolated` statements that enforce mutual exclusion on all objects.

We first review the existing global `isolated` statement in HJ [8]. The HJ construct, `isolated` $\langle stmt1 \rangle$, guarantees that each instance of $\langle stmt1 \rangle$ will be performed in mutual exclusion with all other potentially parallel *interfering* instances of `isolated` statements $\langle stmt2 \rangle$. Two instances of `isolated` statements, $\langle stmt1 \rangle$ and $\langle stmt2 \rangle$, are said to interfere with each other if both access the same shared location, such that at least one of the accesses is a write.

The current HJ implementation takes a simple *single-lock* approach to implementing `isolated` statements, by treating 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*, thereby serializing interfering and non-interfering `isolated` statement instances. We refer to this approach as *global mutual exclusion.*

The motivation for object-based isolation is that there are many cases when the programmer knows the set of objects that will be accessed in the body of the isolated statement. In that case, they can use a statement of the form, `isolated(`obj_0, obj_1, ...`)` $\langle stmt1 \rangle$, to specify the set of objects involved. (The order of objects is not significant.) We refer to this approach as *partial mutual exclusion.* In this case, two `isolated` statements are only guaranteed to execute in mutual execution if they have a non-empty intersection in their object sets. For convenience, the standard `isolated` statement in HJ is assumed to be equivalent to `isolated(*)` *i.e.,* an object-based `isolated` statement on the universal set of objects.

Figure 1 contains an example that uses global mutual exclusion to implement the `insert` function in the `SortList` benchmark. Lines 14 to 21 contain the critical section that performs node insertion and executes in mutual exclusion with any other critical sections that operate on objects `prev` and `curr`.

Figure 2 shows how the example presented in Figure 1 can be rewritten to use object-based `isolated` statements instead. In this case, the programmer explicitly identifies objects `prev` and `curr` as being involved in the mutual exclusion. As discussed in Section 5.1.1, lock-based implementations of object-based isolation rely on an ability to order the objects. This ordering is in turn used to guarantee an absence of deadlock in the implementation of object-based isolation.

As mentioned earlier, imposing a total order on the isolated objects is the key mechanism to avoid deadlocks. This can be done easily for objects in a single isolated list, but nested isolated constructs can pose a challenge. A sufficient condition for deadlock avoidance with nested isolation is to prohibit an inner `isolated` statement from including an object that was not already acquired by an outer `isolated` statement. (Note that this condition permits a task to re-acquire the same object, as is done with reentrant locks.) Figure 3 contains an example to illustrate this rule.

```
public boolean insert(final int v) {
  while (true) {
    INode curr, prev = null;
    for (curr = first; curr != null;
         curr = curr.getNext()) {
      final int val = curr.getValue();
       // v already exists
      if (val == v) return false;
      else if (val > v) break;
      prev = curr;
    }

    boolean set = false;
    isolated {
      if (validate(prev, curr)) {
        final INode neo = new INode();
        neo.setValue(v);
        link(prev, neo, curr);
        set = true;
      }
    }
    if (set) return true;
  }
}
```

Figure 1: SortList Insert Operation with Global Isolation

```
public boolean insert(final int v) {
  while (true) {
    INode curr, prev = null;
    for (curr = first; curr != null;
         curr = curr.getNext()) {
      final int val = curr.getValue();
      if (val == v) return false; // v already exists
      else if (val > v) break;
      prev = curr;
    }

    boolean set = false;
    if (prev != null && curr != null) {
      isolated(prev, curr) {
        if (validate(prev, curr)) {
          final INode neo = new INode();
          neo.setValue(v);
          link(prev, neo, curr);
          set = true;
        }
      }
    }
  }
}
```

Figure 2: SortList Insert Operation with Partial Mutual Exclusion

`isolated (obj1, obj2) {`
`  isolated (obj3) {`
`    S;`
`  }`
`}`
(a)
	`isolated {`
	`  isolated (obj1, obj2) {`
	`    S;`
	`  }`
	`}`
	(b)

Figure 3: Simple cases for nested isolation.

In Figure 3 (a), if `obj3` is neither an alias of `obj1` nor of `obj2`, then the statement `S` should not be permitted to execute and a runtime exception is thrown at that point. This is because the inner `isolated` region tries to acquire a new object that was not included in the outer `isolated` region, thereby opening the possibility of a deadlock. Figure 3 (b) shows a legal example for nested isolation. The outer `isolated` region is a global mutual exclusion construct (i.e. it acquires all objects), which makes it is legal to acquire any object in the inner `isolated` region. To ensure the correctness of nested isolation, the implementation relies on runtime checking as compile-time checking is undecidable in general.

Finally, a `null` object reference in list for an object-based `isolated` statement is essentially no-op (unlike the Java synchronized statement, which throws a NullPointerException in that case). Consider the following example:

isolated (obj1, obj2) S;

If `obj1` is `null`, then an implementation of the `isolated` statement only needs to acquire `obj2`, thereby making the previous statement is equivalent to the following:

isolated (obj2) S;

If both `obj1` and `obj2` are `null`, then the `isolated` statement degenerates to a no-op and no isolation is imposed on `S`.

5.3 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 `neighbors` can be used to store the set of vertices `v` ≠ `u` such that there is an edge connecting vertices `u` and `v`. Figure 4 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 5 shows three possible spanning trees for the graph in Figure 4.

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.

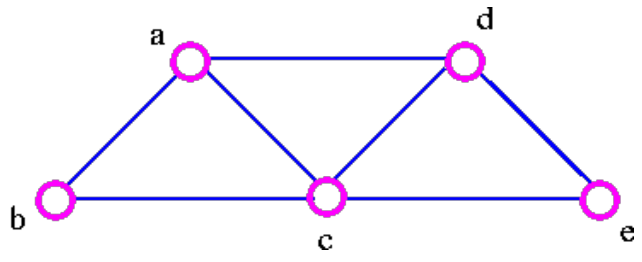

Figure 4: Example undirected graph (source [14])

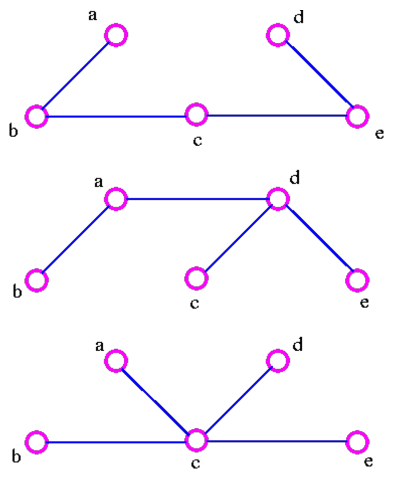

Figure 5: Three spanning trees for undirected graph in Figure 4 (source [14])

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
16 . . .
17 root.parent = root; // Use self-cycle to identify root
18 root.compute();
19 . . .
```

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     isolated 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
16  . . .
17  root.parent = root; // Use self-cycle to identify root
18  finish root.compute();
19  . . .
```

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.

5.4 Atomic Variables

5.4.1 AtomicInteger and AtomicIntegerArray

The `java.util.concurrent` package [13, 5] (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[1].

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* [5]. This section focuses on the former, which is embodied in the `java.util.concurrent.atomic` sub-package [1, 13]. 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 source of efficiency arises from teh fact that many platforms offer hardware support for executing these read-modify-write operations atomically.

Thus, the basic idea behind atomic variables is to implement common access patterns occurring in `isolated` statements as predefined methods that can be invoked by the programmer, with efficient implementations that avoid the use of locks. Atomic variables provided a restricted solution to scalable implementations of `isolated`. If an `isolated` statement matches an available atomic pattern, then it can be implemented by using an atomic variable; otherwise, the default implementation of `isolated` or object-based isolation has to be used instead.

The operations of interest for two `j.u.c.` atomic classes, `AtomicInteger` and `AtomicIntegerArray`, are summarized in Table 1. 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 1. 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 1 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 *per-*

[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.

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

Table 1: Methods in `java.util.concurrent` atomic classes `AtomicInteger` and `AtomicIntegerArray` and their equivalent HJ object-based 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: `rank = new ...; rank.count = 0;` `. . .` `isolated(rank) r = ++rank.count;`	`AtomicInteger rank = new AtomicInteger();` `. . .` `r = rank.incrementAndGet();`
2) Work assignment: `rem = new ...; rem.count = n;` `. . .` `isolated(rem) r = rem.count--;` `if ( r > 0 ) . . .`	`AtomicInteger rem = new AtomicInteger(n);` `. . .` `r = rem.getAndDecrement();` `if ( r > 0 ) . . .`
3) Counting semaphore: `sem = new ...; sem.count = 0;` `. . .` `isolated(sem) r = ++sem.count;` `. . .` `isolated(sem) r = --sem.count;` `. . .` `isolated(sem) s = sem.count; isZero = (s==0);`	`AtomicInteger sem = new AtomicInteger();` `. . .` `r = sem.incrementAndGet();` `. . .` `r = sem.decrementAndGet();` `. . .` `s = sem.get(); isZero = (s==0);`
4) Sum reduction: `sum = new ...; sum.val = 0;` `. . .` `isolated(sum) sum.val += x;`	`AtomicInteger sum = new AtomicInteger();` `. . .` `sum.addAndGet(x);`

Table 2: Examples of common `isolated` statement idioms and their equivalent `AtomicInteger` implementations

formance of your program in the *moderate contention* case will usually be superior when using `AtomicInteger` methods for the reasons discussed in Section 5.1.1. The complete list of `AtomicInteger` methods can be found in [1]. 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 [7, 6] as primitives for building scalable parallel algorithms.

Table 1 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`.

5.4.2 AtomicReference

It is useful to perform atomic operations on object references, in addition to atomic operations on the primitive data types outlined in Section 5.4.1. 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 4 with an object-based isolated statement used in the Parallel Spanning Tree example.

```
 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      boolean retVal;
 6      isolated(this) { if (parent == null) parent=n; retVal = (parent == n);
 7      return retVal;
 8    } // tryLabeling
 9    . . .
10  } // class V
```

Listing 4: Use of isolated in Parallel Spanning Tree example

```
class V {
  V [] neighbors; // adjacency list for input graph
  AtomicReference parent;       // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    return parent.compareAndSet(null, n);
  } // tryLabeling
  . . .
} // class V
```

Listing 5: Use of `compareAndSet()` as a replacement for `isolated` in Listing 4

The `isolated` statement in line 5 of Listing 4 can be replaced by a `compareAndSet()` method if `parent` is stored as an `AtomicReference`, as shown above in Listing 5. There are additional `j.u.c.atomic` classes

`j.u.c.atomic` Class and Constructors	`j.u.c.atomic` Methods	Equivalent HJ `isolated` statements
AtomicReference	Object o = v.**get**();	Object o; `isolated` o = v.ref;
	v.**set**(newRef);	`isolated` v.ref = newRef;
AtomicReference() // init = null	Object o = v.**getAndSet**(newRef);	Object o; `isolated` { o = v.ref; v.ref = newRef; }
AtomicReference(init)	boolean b = v.**compareAndSet** (expect,update);	boolean b; `isolated` if (v.ref==expect) {v.ref=update; b=true;} else b = false;
AtomicReferenceArray	Object o = v.**get**(i);	Object o; `isolated` o = v.arr[i];
	v.**set**(i,newRef);	`isolated` v.arr[i] = newRef;
AtomicReferenceArray (length) // init = null	Object o = v.**getAndSet**(i,newRef);	Object o; `isolated` { o = v.arr[i]; v.arr[i] = newRef; }
AtomicIntegerArray (arr)	boolean b = v.**compareAndSet** (i,expect,update);	boolean b; `isolated` if (v.arr[i]==expect) {v.arr[i]=update; b=true;} else b = false;

Table 3: Methods in `java.util.concurrent` atomic classes `AtomicReference` and `AtomicReferenceArray` and their equivalent HJ object-based 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[]`.

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.

5.5 Read-Write Modes in Object-based Isolation

TO BE COMPLETED

5.6 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 executed. 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 6 (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: $v16 \rightarrow v10 \rightarrow v11$, $v10 \rightarrow v16 \rightarrow v11$ and $v10 \rightarrow v11 \rightarrow v16$. Each order leads to a different SCG, when serialization edges are added. The red edges in Figure 6 show serialization edges added when the `isolated` steps execute in the sequence, $v10 \rightarrow v16 \rightarrow v11$. 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 $v10 \rightarrow v11 \rightarrow v16$ and $v16 \rightarrow v10 \rightarrow v11$ which result in a critical path length of 17 nodes.

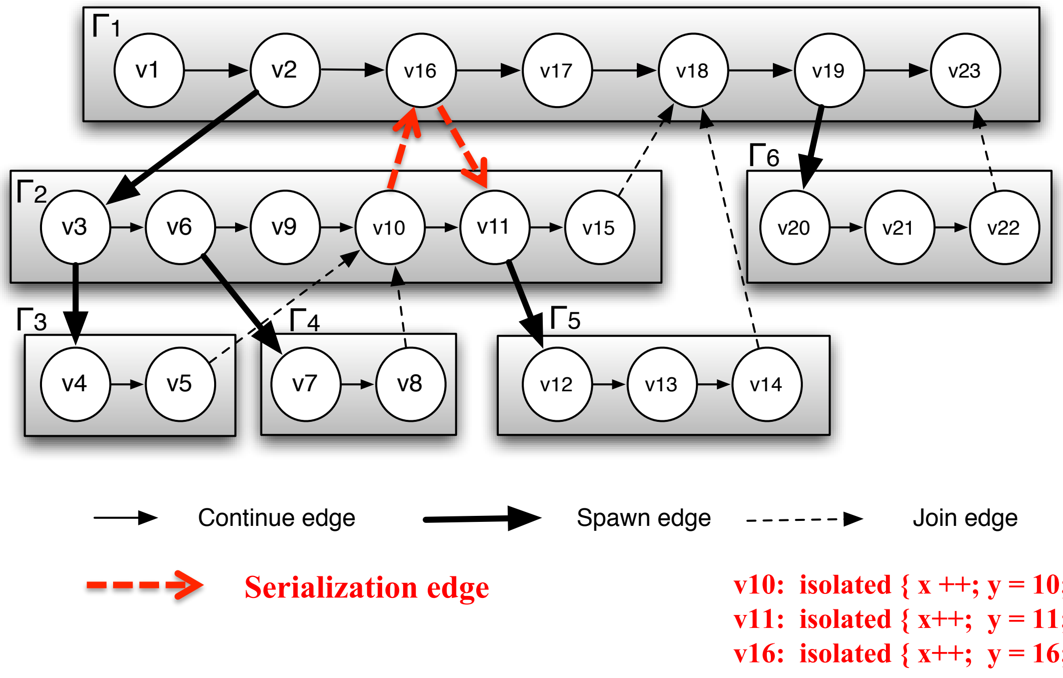

Figure 6: Serialized Computation Graph with Serialization Edges

5.6.1 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 6, 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 6 (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 6. 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 6: 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 7.

```
1 rank.count = 0; // rank object contains an int field, count
2 forall (point[i] : [0:m-1]) {
3   int r;
4   isolated { r = rank.count++; }
5   StringArray[i] =  Hello , World from task with rank =    + r;
6 }
```

Listing 7: Using an isolated statement to fix the data race error in Listing 7

Note that the program in Listing 7 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 6 and in Listing 7. 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* [12]. 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 6.

6 The Actor Model

Acknowledgment: the content in this section was developed in collaboration with Shams Imam.

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 [9]. It was designed to address the problems that arise while writing distributed applications. Further work by Henry Baker [10], Gul Agha [2], 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.

6.1 Introduction to 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 7, 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.

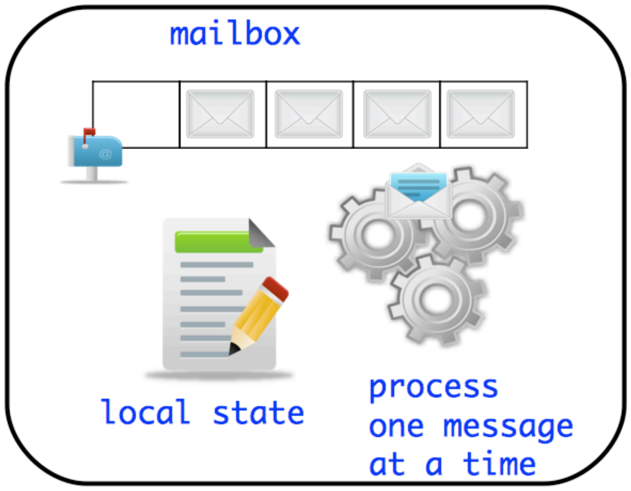

Figure 7: Actors store incoming messages in a mailbox, maintain a local state which is not directly exposed to other actors, and process at most one message at a time.

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.
- *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.

An actor interacts with other actors in two ways as shown in Figure 9. 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

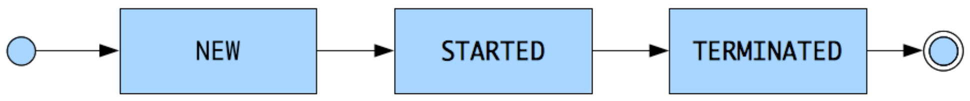

Figure 8: Actors have a simple life cycle. The most interesting state is *started* which is where the actor is receiving and processing messages.

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 [15].

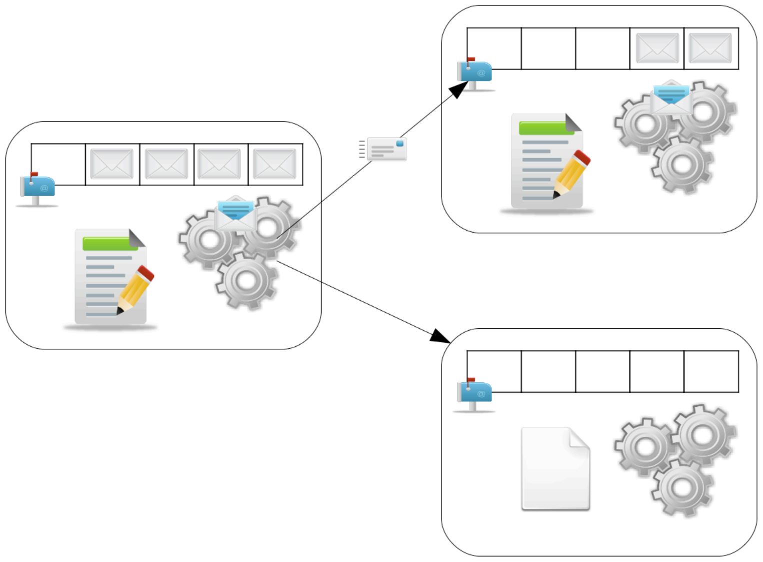

Figure 9: During the processing of a message, actor interactions include exchanging messages with other actors and creating new actors.

6.1.1 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.

6.1.2 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.

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;

public class HelloWorld {
    public static void main(final String[] args) {
        PrintActor actor = new PrintActor();
        actor.start();
        actor.send("Hello");
        actor.send("World");
        actor.send(PrintActor.STOP_MSG);
    }
}

class PrintActor extends Actor<Object> {
    static final Object STOP_MSG = new Object();
    protected void process(final Object msg) {
        if (STOP_MSG.equals(msg)) {
            exit();
        } else {
            System.out.println(msg);
        }
    }
}
```

Listing 8: HelloWorld using HJ actors

Listing 8 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.

```java

import hj.lang.Actor;

public class SimplePipeline {
    public static void main(final String[] args) {
        // rely on the implicit IEF for HJ programs
        Actor<Object> firstStage =
            new EvenLengthFilter(
                new LowerCaseFilter(
                    new PrintStage()));

        firstStage.start();

        firstStage.send("pipeline");
        firstStage.send("Filter");
        firstStage.send("example");

        firstStage.send(new StopMessage());
    }
}

class StopMessage {}

class EvenLengthFilter extends Actor<Object> {
    private final Actor<Object> nextStage;
    EvenLengthFilter(final Actor<Object> nextStage) {
        this.nextStage = nextStage;
    }
    public void start() {
        super.start();
        nextStage.start();
    }
    protected void process(final Object msg) {
        if (msg instanceof StopMessage) {
            nextStage.send(msg);
            exit();
        } else if (msg instanceof String) {
            String msgStr = (String) msg;
            if (msgStr.length() % 2 == 0) {
                nextStage.send(msgStr);
            }
        }
    }
}
class LowerCaseFilter extends Actor<Object> {
    private final Actor<Object> nextStage;
    LowerCaseFilter(final Actor<Object> nextStage) {
        this.nextStage = nextStage;
    }
    public void start() {
        super.start();
        nextStage.start();
    }
    protected void process(final Object msg) {
        if (msg instanceof StopMessage) {
            exit();
            nextStage.send(msg);
        } else if (msg instanceof String) {
            String msgStr = (String) msg;
            if (msgStr.toLowerCase().equals(msgStr)) {
                nextStage.send(msgStr);
            }
        }
    }
}
class PrintStage extends Actor<Object> {
    protected void process(final Object msg) {
        if (msg instanceof StopMessage) {
            exit();
        } else if (msg instanceof String) {
            System.out.println(msg);
        }
    }
}
```

Listing 9: A simple pipeline using HJ actors

Listing 9 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.

6.1.3 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).

6.2 Ordering of Messages

Habanero actors preserve 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 [18].

6.3 Sieve of Eratosthenes

Pipelining is used for repetitive tasks where each task can be broken down into independent sub-tasks (also called stages) which must be performed sequentially, one after the other. Each stage partially processes data and then forwards the partially processed result to the next stage in the pipeline for further processing. This pattern works best if the operations performed by the various stages of the pipeline are balanced, i.e. take comparable time. If the stages in the pipeline vary widely in computational effort, the slowest stage creates a bottleneck for the aggregate throughput.

The pipeline pattern is a natural fit with the actor model since each stage can be represented as an actor. The single message processing rule ensures that each stage/actor processes one message at a time before handing off to the next actor in the pipeline. The stages however need to ensure ordering of messages while processing them. In our actor model, this ordering support is provided by default for messages from the same actor to another actor. However, the amount of parallelism in a full pipeline is limited by the number of stages.

One algorithm that can be solved elegantly using a dynamic pipeline is the Sieve of Eratosthenes [17]. The algorithm incrementally builds knowledge of primes. The basic idea is to create multiple stages of the pipeline that forward a candidate prime number to the next stage only if the stage determines the candidate is locally prime. When the candidate reaches the end of the pipeline, the pipeline may need to be extended. Thus, this is also an example of a dynamic pipeline where the number of stages is not necessarily known in advance. A simple diagrammatic explanation of how the pipeline would work is shown in Figure 10. Note that to reduce the relative overhead in a real implementation, you will need to increase the amount of work done in each stage by having it store and process multiple prime numbers as a batch.

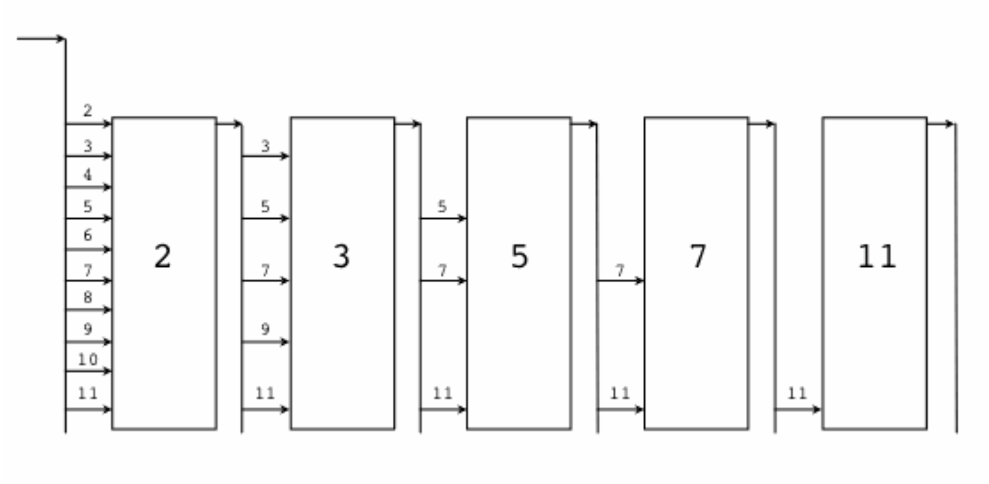

Figure 10: Illustration of Sieve of Eratosthenes algorithm (source: `http://golang.org/doc/sieve.gif`)

6.4 Producer-Consumer with Unbounded Buffer

TO BE COMPLETED

6.5 Producer-Consumer with Bounded Buffer

TO BE COMPLETED

6.6 Integration with Task Parallelism

There is internal concurrency in an actor in that it can be processing a message, receiving messages from other actors and sending messages to other actors at the same time. However, the requirement that the actor must process at most one message at a time is often misunderstood to mean that the processing must be done via sequential execution. In fact, there can be parallelism exposed even while message processing as long as the semantics is equivalent to that of processing at most one message at a time. A unique feature of the Habanero Actor model lies in its integration with task parallelism, so that (for example) async and finish constructs can be used when processing a message.

6.6.1 Using `finish` constructs during message processing

The traditional actor model already ensures that the actor processes one message at a time. Since no additional restrictions are placed on the message processing body, we can achieve parallelism by creating new `async-finish` constructs inside the message processing body. This will mean that we will spawn off new tasks to achieve the parallelism at the cost of blocking the original message processing task at the new `finish`. Since the main message processing task only returns after all spawned tasks have completed, the invariant that only one message is processed at a time is maintained. Figure 11 shows an example code snippet that achieves this. Note that there is no restriction on the constructs used inside the newly constructed `finish`.

```
class ParallelizedProcessingActor() extends HybridActor {
  override def behavior() = {
    case msg: SomeMessage =>
      // preprocess the message
      finish { // finish to ensure all spawned tasks complete
        async {
          // do some processing in parallel
        }
        async {
          // do some more processing in parallel
        }
      }
      // postprocess the message after spawned tasks inside finish complete
    ...
  }
}
```

Figure 11: An actor exploiting the `async-finish` parallelism inside actors message processing body. The nested `finish` ensures no spawned tasks escape away causing the actor to process multiple messages at a time.

6.6.2 Pause and Resume operations

Requiring all spawned asyncs inside the message processing body are captured is too strict. This restriction can be relaxed based on the observation that the *at most one message processing rule* is required to ensure there are no internal state changes of an actor being effected by two or more message processing tasks of the same actor. We can achieve the same invariant by introducing a *paused* state in the actor life cycle and adding two new operations: `pause` and `resume`. In the *paused* state the actor is not processing any messages from its mailbox. The actor is simply idle as in the *new* state, however the actor can continue receiving messages from other actors. The actor will resume processing its messages, at most one at a time, when it returns to the *started* state. The `pause` operation takes the actor from a *started* state to a *paused* state while the `resume` operation achieves the reverse. The actor is also allowed to terminate from the *paused* state using the `exit` operation. Neither of these operations are blocking, they only affect the internal state of the actor which controls when messages are processed from the mailbox.

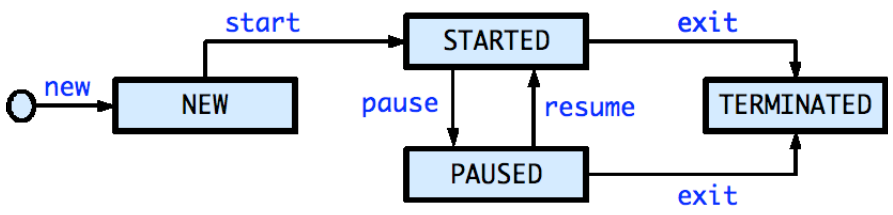

Figure 12: Actor life cycle from Figure 8 extended with a *paused* state. The actor can now continually switch between a *started* and *paused* state.

With the two new operations, we can now allow spawned tasks to escape the main message processing task. These spawned tasks are safe to run in parallel with the next message processing task of the same actor as long as they are not concurrently affecting the internal state of the actor. The actor can be suspended in a *paused* state while these spawned tasks are executing and can be signaled to resume processing messages once the spawned tasks determine they will no longer be modifying the internal state of the actor and hence not violating the one message processing rule. Figure 13 shows an example where the `pause` and `resume` operations are used to achieve parallelism inside the message processing body.

```
class ParallelizedWithEscapingAsyncsActor() extends HybridActor {
  override def behavior() = {
    case msg: SomeMessage =>
      // preprocess the message
      async {
        // do some processing in parallel
      }
      pause // to prevent the actor from processing the next message
      // note that pause/resume is not blocking
      async {
        // do some more processing in parallel
        // it is now safe for the actor to resume processing other messages
        resume
        // some more processing
      }
     ...
  }
}
```

Figure 13: An actor exploiting parallelism via `asyncs` while avoiding an enclosing `finish`. The asyncs escape the message processing body, but the `pause` and `resume` operations are used to control processing of subsequent messages by the actor.

6.6.3 Stateless Actors

The `http://www.gpars.org/guide/`GPars project, implemented in Groovy, has a notion of *stateless actors* that are allowed to process multiple messages simultaneously when they do not update internal state. As shown in Figure 14 (with Scala syntax!), it is easy to create stateless actors in Habanero's actor model because of its integration with task parallelism. There is no need to use the `pause` operation for stateless actors; the escaping `async` tasks can process multiple messages to the same actor in parallel.

References

[1] Package java.util.concurrent.atomic. URL `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/atomic/package-summary.html`.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[3] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29:6–16, March 2009. ISSN 0272-1732. URL `http://portal.acm.org/citation.cfm?id=1550399.1550516`.

[4] Edsger W. Dijkstra. The structure of the "the"-multiprogramming system. In *Proceedings of the first*

```
class StatelessActor() extends HybridActor {
  override def behavior() = {
    case msg: SomeMessage =>
      async {
        // process the current message
      }
      if (enoughMessagesProcessed) {
        exit()
      }
      // return immediately to be ready to process the next message
  }
}
```

Figure 14: A simple stateless actor created using the hybrid model. The message processing body spawns a new task to process the current message and returns immediately to process the next message. Because the `async` tasks are allowed to escape, the actor may be processing multiple messages simultaneously.

ACM symposium on Operating System Principles, SOSP '67, pages 10.1–10.6, New York, NY, USA, 1967. ACM. URL `http://doi.acm.org/10.1145/800001.811672`.

[5] B. Goetz. *Java Concurrency In Practice.* Addison-Wesley, 2007.

[6] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer — designing an mimd shared memory parallel computer. *IEEE Transactions on Computers*, 32:175–189, 1983. ISSN 0018-9340. URL `http://doi.ieeecomputersociety.org/10.1109/TC.1983.1676201`.

[7] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5:164–189, April 1983. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/69624.357206`.

[8] Habanero. Habanero Java programming language. http://habanero.rice.edu/hj, Dec 2009.

[9] Carl Hewitt, Peter Bishop, and Richard Steiger. Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, August 1973.

[10] Hewitt, Carl and Baker, Henry G. Actors and Continuous Functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, February 1978.

[11] Shams Imam, Jisheng Zhao, and Vivek Sarkar. A composable deadlock-free approach to object-based isolation. In Jesper Larsson Trff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, volume 9233 of *Lecture Notes in Computer Science*, pages 426–437. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-48095-3. doi: 10.1007/978-3-662-48096-0_33. URL `http://dx.doi.org/10.1007/978-3-662-48096-0_33`.

[12] James R. Larus and Ravi Rajwar. *Transactional Memory.* Morgan and Claypool, 2006. URL `http://www.morganclaypool.com/doi/abs/10.2200/S00070ED1V01Y200611CAC002`.

[13] Qusay H. Mahmoud. Concurrent Programming with J2SE 5.0, March 2005. URL `http://java.sun.com/developer/technicalArticles/J2SE/concurrency/`.

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

[15] Natarajan Raja and Rudrapatna K. Shyamasundar. Actors as a Coordinating Model of Computation. In *Proceedings of the 2nd International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 191–202. Springer-Verlag, 2004. ISBN 3-540-62064-8.

[16] Wikipedia. Critical section, 2010. URL `http://en.wikipedia.org/wiki/Critical_section`.

[17] Wikipedia, The Free Encyclopedia. Sieve of Eratosthenes. `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`. URL `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

[18] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 258–268, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28722. URL `http://doi.acm.org/10.1145/28697.28722`.