COMP 322: Fundamentals of Parallel Programming

Lecture 22: Introduction to the Actor Model

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/
Q: Compute the WORK and CPL metrics for this program with a global isolated construct. Indicate if your answer depends on the execution order of isolated constructs.

```java
1. finish(() -> {
2.     for (int i = 0; i < 5; i++) {
3.         async(() -> {
4.             doWork(2);
5.             isolated(() -> { doWork(1); });
6.             doWork(2);
7.         }); // async
8.     } // for
9. }); // finish
```

Answer: WORK = 25, CPL = 9. These metrics do not depend on the execution order of isolated constructs.
Q: Compute the WORK and CPL metrics for this program with an object-based isolated construct. Indicate if your answer depends on the execution order of isolated constructs.

1. `finish(() -> {
2.     // Assume X is an array of distinct objects
3.     for (int i = 0; i < 5; i++) {
4.         async(() -> { // Async task A_i
5.             doWork(2);
6.             isolated(X[i], X[i+1],
7.             () -> { doWork(1); });
8.             doWork(2);
9.         }); // async
10.     } // for
11. }); // finish

Answer: WORK = 25, worst-case CPL = 7 (e.g., if A_1, A_4 execute in parallel first, then the isolated sections in A_2, A_3 must be serialized thereafter.)
Actors: an alternative approach to isolation

- An actor is an autonomous, interacting component of a parallel system.
- An actor has:
  - an immutable identity (global reference)
  - a single logical thread of control
  - mutable local state (isolated by default)
  - procedures to manipulate local state (interface)
The Actor Model: Fundamentals

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

- A message-based concurrency model to manage mutable shared state
  - First defined in 1973 by Carl Hewitt
  - Further theoretical development by Henry Baker and Gul Agha

- Key Ideas:
  - Everything is an Actor!
  - Analogous to “everything is an object” in OOP
  - Encapsulate shared state in Actors
  - Mutable state is not shared - i.e., no data races

- Other important features
  - Asynchronous message passing
  - Non-deterministic ordering of messages
Actor states

- **New**: Actor has been created
  - e.g., email account has been created, messages can be received
- **Started**: Actor can process messages
  - e.g., email account has been activated
- **Terminated**: Actor will no longer processes messages
  - e.g., termination of email account after graduation
Actor Analogy - Email

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

- Create your custom class which extends `edu.rice.hj.runtime.actors.Actor<T>`, and implement the void `process()` method (type parameter T specifies message type)

  ```java
class MyActor extends Actor<T> {
    protected void process(T message) {
      println("Processing " + message);
    }
  }
```

- Instantiate and start your actor

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

- Send messages to the actor (can be performed by actor or non-actor)

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

- Use a special message to terminate an actor

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

- Actor execution implemented as async tasks
  Can use `finish` to await completion of an actor, if the actor is start-ed inside the finish.
Summary of HJlib Actor API

```c
void process(MessageType theMsg) // Specification of actor's "behavior" when processing messages

void send(MessageType msg) // Send a message to the actor

void start() // Cause the actor to start processing messages
void onPreStart() // Convenience: specify code to be executed before actor is started
void onPostStart() // Convenience: specify code to be executed after actor is started

void exit() // Actor calls exit() to terminate itself
void onPreExit() // Convenience: specify code to be executed before actor is terminated
void onPostExit() // Convenience: specify code to be executed after actor is terminated

// In Lecture 23
void pause() // Pause the actor, i.e. the actors stops processing messages in its mailbox
void resume() // Resume a paused actor, i.e. actor resumes processing messages in mailbox

```
Hello World Example

1. public class HelloWorld {
2.     public static void main(final String[] args) {
3.         finish() -> {
4.             EchoActor actor = new EchoActor();
5.             actor.start(); // don’t forget to start the actor
6.             actor.send("Hello"); // asynchronous send (returns immediately)
7.             actor.send("World"); // Non-actors can send messages to actors
8.             actor.send(EchoActor.STOP_MSG);
9.         }
10.        println("EchoActor terminated.");
11.    }
12.    private static class EchoActor extends Actor<Object> {
13.        static final Object STOP_MSG = new Object();
14.        private int messageCount = 0;
15.        protected void process(final Object msg) {
16.            if (STOP_MSG.equals(msg)) {
17.                println("Message-" + messageCount + ": terminating.");
18.                exit(); // never forget to terminate an actor
19.            } else {
20.                messageCount += 1;
21.                println("Message-" + messageCount + ": " + msg);
22.            }
23.        }
24.    }
25.}

Though sends are asynchronous, many actor libraries (including HJlib) preserve the order of messages between the same sender actor/task and the same receiver actor.
Integer Counter Example

Without Actors:

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

• Can also use atomic variables instead of isolated construct

With Actors:

15. class Counter extends Actor<Message> {
16.   private int counter = 0; // local state
17.   protected void process(Message msg) {
18.     if (msg instanceof IncMessage) {
19.       counter++;
20.     } else if (msg instanceof DecMessage){
21.       counter--;
22.     } } } }
23. . . .
24. Counter counter = new Counter();
25. counter.start();
26. public void foo() {
27.   // do something
28.   counter.send(new IncrementMessage(1));
29.   // do something else
30. }
31. public void bar() {
32.   // do something
33.   counter.send(new DecrementMessage(1));
34. }
ThreadRing (Coordination) Example

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

14. class ThreadRingActor
15.     extends Actor<Integer> {
16.         private Actor<Integer> nextActor;
17.         private final int id;
18.         ...
19.         public void nextActor(Actor<Object> nextActor) {...}

21.     protected void process(Integer n) {
22.          if (n > 0) {
23.              println("Thread-" + id +
24.                 " active, remaining = " + n);
25.              nextActor.send(n - 1);
26.          } else {
27.              println("Exiting Thread-"+ id);
28.              nextActor.send(-1);
29.              exit();
30.      }
```

```
0

3

1

2

ThreadRing (Coordination) Example

COMP 322, Spring 2017 (V. Sarkar, M. Joyner)
```
Pi Computation Example

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

- Use Master-Worker technique:

```
Master
accumulates approximation of PI

Worker-1

Worker-2

Series 1 - 100

Series 101 - 200

Series 1301 - 1400

Series 1401 - 1500

Series 1201 - 1300

Series 2501 - 2600
```

Source: [http://www.enotes.com/topic/Pi](http://www.enotes.com/topic/Pi)
1. class Master extends Actor<Object> {
2.     private double result = 0; private int nrMsgsReceived = 0;
3.     private Worker[] workers;
4.     Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
5.     protected void onPostStart() {
6.         // Create and start workers
7.         workers = new Worker[nrWrkrs];
8.         for (int i = 0; i < nrwrkrs; i++) {
9.             workers[i] = new Worker();
10.            workers[i].start();
11.         }
12.         // Send messages to workers
13.         for (int j = 0; j < nrMsgs; j++) {
14.             someWrkr = ... ; // Select worker for message j
15.             someWrkr.send(new Work(...));
16.         }
17.     } // start()
19. `protected void onPostExecute()` {
20.     for (int i = 0; i < nrWrkrs; i++)
21.         workers[i].send(new Stop());
22. } // post-exit()
23. `protected void process(final Object msg) {`
24.     if (msg instanceof Result) {
25.         result += ((Result) msg).result;
26.         nrMsgsReceived += 1;
27.         if (nrMsgsReceived == nrMsgs) exit();
28.     }
29.     // Handle other message cases here
30. } // process()
31. } // Master
32. . . .
33. // Main program
34. Master master = new Master(w, e, m);
35. finish(() -> { master.start(); });
36. println("PI = " + master.getResult());
Pi Calculation --- Worker Actor

```java
1. class Worker extends Actor<Object> {
2.     protected void process(final Object msg) {
3.         if (msg instanceof Stop)
4.             exit();
5.         else if (msg instanceof Work) {
6.             Work wm = (Work) msg;
7.             double result = calculatePiFor(wm.start, wm.end)
8.                 master.send(new ResultMessage(result));
9.         } // process()
10. }
11. }
12. private double calculatePiFor(int start, int end) {
13.     double acc = 0.0;
14.     for (int k = start; k < end; k++) {
15.         acc += 4.0 * (1 - (k % 2) * 2) / (2 * k + 1);
16.     }
17.     return acc;
18. } // Worker
```
What output will be printed if the end-finish operation from slide 13 is moved from line 13 to line 11 as shown below? (Hint: see rule re. finish and actor start operations at bottom of slide 9.)

1. `finish()` -> {
2. \hspace{1em} int threads = 4;
3. \hspace{1em} int numberOfHops = 10;
4. \hspace{1em} ThreadRingActor[] ring = new ThreadRingActor[threads];
5. \hspace{1em} for(int i=threads-1;i>=0; i--) {
6. \hspace{2em} ring[i] = new ThreadRingActor(i);
7. \hspace{2em} ring[i].start();
8. \hspace{2em} if (i < threads - 1) {
9. \hspace{3em} ring[i].nextActor(ring[i + 1]);
10. \hspace{2em} }
11. \hspace{1em} }
12. // finish
13. ring[threads-1].nextActor(ring[0]);
14. ring[0].send(numberOfHops);
Limitations of Actor Model (move to Lec 23)

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

=> Some of these limitations can be overcome by using a hybrid model that combines task parallelism with actors (more on this in the next lecture!)
1. Insert finish, async, and isolated constructs (pseudocode is fine) to convert the sequential spanning tree algorithm below into a parallel algorithm.

   See slide 3, as well as the isolatedWithReturn() API in slide 4 for convenience in implementing the pseudocode.

2. Is it better to use a global isolated or an object-based isolated construct for the parallelization in question 1? If object-based is better, which object(s) should be included in the isolated list?

   Object-based isolation should be better with a singleton object list containing the “this” object for the makeParent() method.
Parallel Spanning Tree Algorithm using object-based isolated construct

1. class V {
2.   V[] neighbors; // adjacency list for input graph
3.   V parent; // output value of parent in spanning tree
4.   boolean makeParent(final V n) {
5.       return isolatedWithReturn(this, () -> {
6.           if (parent == null) { parent = n; return true; }
7.           else return false; // return true if n became parent
8.       });
9.   } // makeParent
10.  void compute() {
11.      for (int i=0; i<neighbors.length; i++) {
12.          final V child = neighbors[i];
13.          if (child.makeParent(this))
14.             async(() -> { child.compute(); });
15.      }
16.  } // compute
17. } // class V
18. . . .
19. root.parent = root; // Use self-cycle to identify root
20. finish(() -> { root.compute(); });
21. . . .
HJ isolatedWithReturn construct

// <body> must contain return statement

isolatedWithReturn (obj1, obj2, ..., () -> <body> );

Motivation: isolated() construct cannot modify local variables due to restrictions imposed by Java 8 lambdas

- Workaround 1: use isolated() and modify objects rather than local variables
  - Pro: code can be easier to understand than modifying local variables
  - Con: source of errors if multiple tasks read/write same object

- Workaround 2: use isolatedWithReturn()
  - Pro: cleaner than modifying local variables
  - Con: can only return one value
### Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements

<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 (v) j = v.val;</td>
</tr>
<tr>
<td></td>
<td>v.set(newVal);</td>
<td>isolated (v) v.val = newVal;</td>
</tr>
<tr>
<td>AtomicInteger()</td>
<td>int j = v.getAndSet(newVal);</td>
<td>int j; isolated (v) { j = v.val; v.val = newVal; }</td>
</tr>
<tr>
<td>// init = 0</td>
<td>int j = v.addAndGet(delta);</td>
<td>isolated (v) { v.val += delta; j = v.val; }</td>
</tr>
<tr>
<td>AtomicInteger(init)</td>
<td>int j = v.getAndAdd(delta);</td>
<td>isolated (v) { j = v.val; v.val += delta; }</td>
</tr>
<tr>
<td></td>
<td>boolean b = v.compareAndSet(expect,update);</td>
<td>boolean b; isolated (v) if (v.val==expect) {v.val=update; b=true;} else b = false;</td>
</tr>
</tbody>
</table>
Atomic Variables represent a special (and more efficient) case of Object-based isolation

```java
1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     AtomicReference<V> parent; // output value of parent in spanning tree
4.     boolean makeParent(final V n) {
5.         // compareAndSet() is a more efficient implementation of
6.         // object-based isolation
7.         return parent.compareAndSet(null, n);
8.     } // makeParent
9.     void compute() {
10.        for (int i=0; i<neighbors.length; i++) {
11.            final V child = neighbors[i];
12.            if (child.makeParent(this))
13.               async(() -> { child.compute(); }); // escaping async
14.        }
15.     } // compute
16. } // class V
17. ...
18. root.parent = root; // Use self-cycle to identify root
19. finish(() -> { root.compute(); });
20. ...
```
Motivation for Read-Write Object-based isolation

Sorted List example

1. public boolean contains(Object object) {
2.     // Observation: multiple calls to contains() should not
3.     // interfere with each other
4.     return isolatedWithReturn(this, () -> {
5.         Entry pred, curr;
6.         ...
7.         return (key == curr.key);
8.     });
9. }
10. }

11. public int add(Object object) {
12.     return isolatedWithReturn(this, () -> {
13.         Entry pred, curr;
14.         ...
15.         if (...) return 1; else return 0;
16.     });
17. }
isolated(readMode(obj1),writeMode(obj2), …, () -> <body> );

- Programmer specifies list of objects as well as their read-write modes for which isolation is required.
- Not specifying a mode is the same as specifying a write mode (default mode = read + write).
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in writeMode.

**Sorted List example**

1. public boolean contains(Object object) {
2.     return isolatedWithReturn(readMode(this), () -> {
3.         Entry pred, curr;
4.         ...
5.         return (key == curr.key);
6.     });
7. }
8.
9. public int add(Object object) {
10.    return isolatedWithReturn(writeMode(this), () -> {
11.        Entry pred, curr;
12.        ...
13.        if (...) return 1; else return 0;
14.    });
15. }
The world according to Module 1 without & with Phasers

- All the non-phaser parallel constructs that we learned focused on task creation and termination
  - async creates a task
    - forasync creates a set of tasks specified by an iteration region
  - finish waits for a set of tasks to terminate
    - forall (like “finish forasync”) creates and waits for a set of tasks specified by an iteration region
  - future get() waits for a specific task to terminate
  - asyncAwait() waits for a set of DataDrivenFuture values before starting

- Motivation for phasers
  - Deterministic directed synchronization within tasks for barriers, point-to-point synchronization, pipelining
  - Separate from synchronization associated with task creation and termination
  - next operations are much more efficient than task creation/termination (async/finish), but they only help reduce overhead if you perform multiple next operations in a task
Medical imaging pipeline with three stages

1. Denoising stage generates a sequence of results, one per image.
2. Registration stage’s input is Denoising stage’s output.
3. Segmentation stage’s input is Registration stage’s output.

Even though the processing is sequential for a single image, pipeline parallelism can be exploited via point-to-point synchronization between neighboring stages.
Implementation of Medical Imaging Pipeline

1. final List<PhaserPair> phList1 = Arrays.asList(ph0.inMode(PhaserMode.SIG));
2. final List<PhaserPair> phList2 = Arrays.asList(ph0.inMode(PhaserMode.WAIT), ph1.inMode(PhaserMode.SIG));
3. final List<PhaserPair> phList3 = Arrays.asList(ph1.inMode(PhaserMode.WAIT));
4.
5. asyncPhased(phList1, () -> { // DENOISE stage
6.     for (int i = 0; i < n; i++) {
7.         dowork(1);
8.         signal(); // same as ph0.signal(); as only ph0 is registered in this async
9.     }
10. })
11. }
12. asyncPhased(phList2, () -> { // REGISTER stage
13.     for (int i = 0; i < n; i++) {
14.         ph0.dowait(); // WARNING: Explicit calls to dowait() can lead to deadlock in general
15.         dowork(1);
16.         ph1.signal();
17.     }
18. })
19. }
20. asyncPhased(phList3, () -> { // SEGMENT stage
21.     for (int i = 0; i < n; i++) {
22.         ph1.dowait();
23.         dowork(1);
24.     }
25. })

COMP 322, Spring 2017 (V. Sarkar, M. Joyner)
Announcements

• Reminder: Quiz for Unit 4 is due today

• Reminder: Checkpoint #2 for Homework 3 is due by Wednesday, March 8th, and the entire homework is due by March 22nd

• The registrar has announced the schedule for the COMP 322 final exam:
  — 2-MAY-2017
  — 9:00AM - 12:00PM
  — Location TBD

• Scope of final exam (Exam 2) will be limited to Lectures 19 - 38
Serialized Computation Graph for Isolated Constructs (Recap)

- Model each instance of an isolated construct as a distinct step (node) in the CG.
- Need to reason about the order in which interfering isolated constructs are executed
  - Complicated because the order of isolated constructs may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated constructs.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each prior “interfering” isolated step, S
    - Two isolated constructs always interfere with each other
    - Interference of “object-based isolated” constructs depends on intersection of object sets
    - Serialization edge is not needed if S and S' are already ordered in CG
  - An SCG represents a set of schedules in which all interfering isolated constructs execute in the same order.
Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order (Recap)

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

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

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

v10: isolated \{ x++; y = 10; \}

v11: isolated \{ x++; y = 11; \}

v16: isolated \{ x++; y = 16; \}

Need to consider all possible orderings of interfering isolated constructs to establish data race freedom