
COMP 322: Fundamentals of Parallel Programming

Lecture 23: Actors

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

<https://wiki.rice.edu/confluence/display/PARPROG/COMP322>



Worksheet #22 solution:

Abstract Metrics with Isolated Constructs

Q: Compute the WORK and CPL metrics for this program. Indicate if your answer depends on the execution order of isolated constructs.

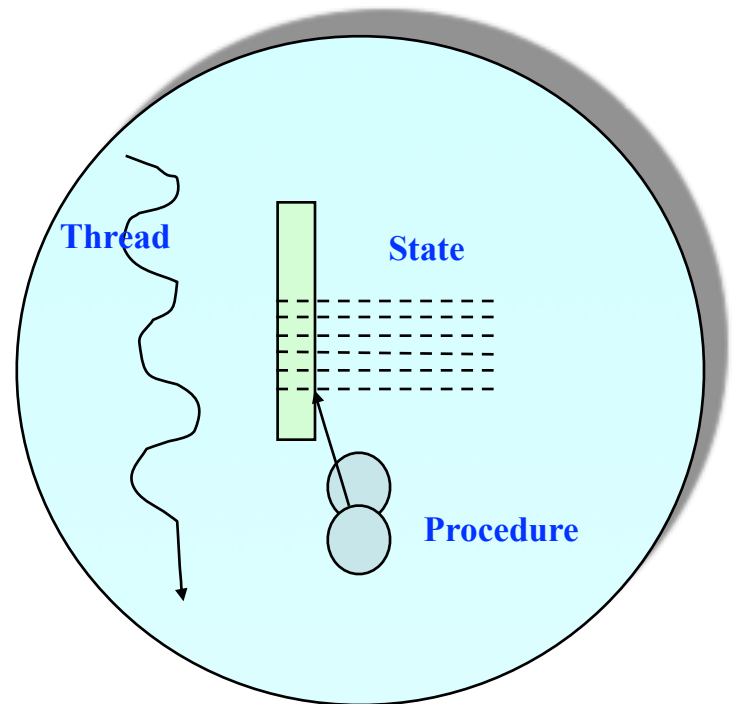
```
1.  finish(() -> {  
2.      for (int i = 0; i < 5; i++) {  
3.          async(() -> {  
4.              doWork(2);  
5.              isolated(() -> { doWork(1); });  
6.              doWork(2);  
7.          }); // async  
8.      } // for  
9.  }); // finish
```

Answer: WORK = 25, CPL = 9. These metrics do not depend on the execution order of isolated constructs.



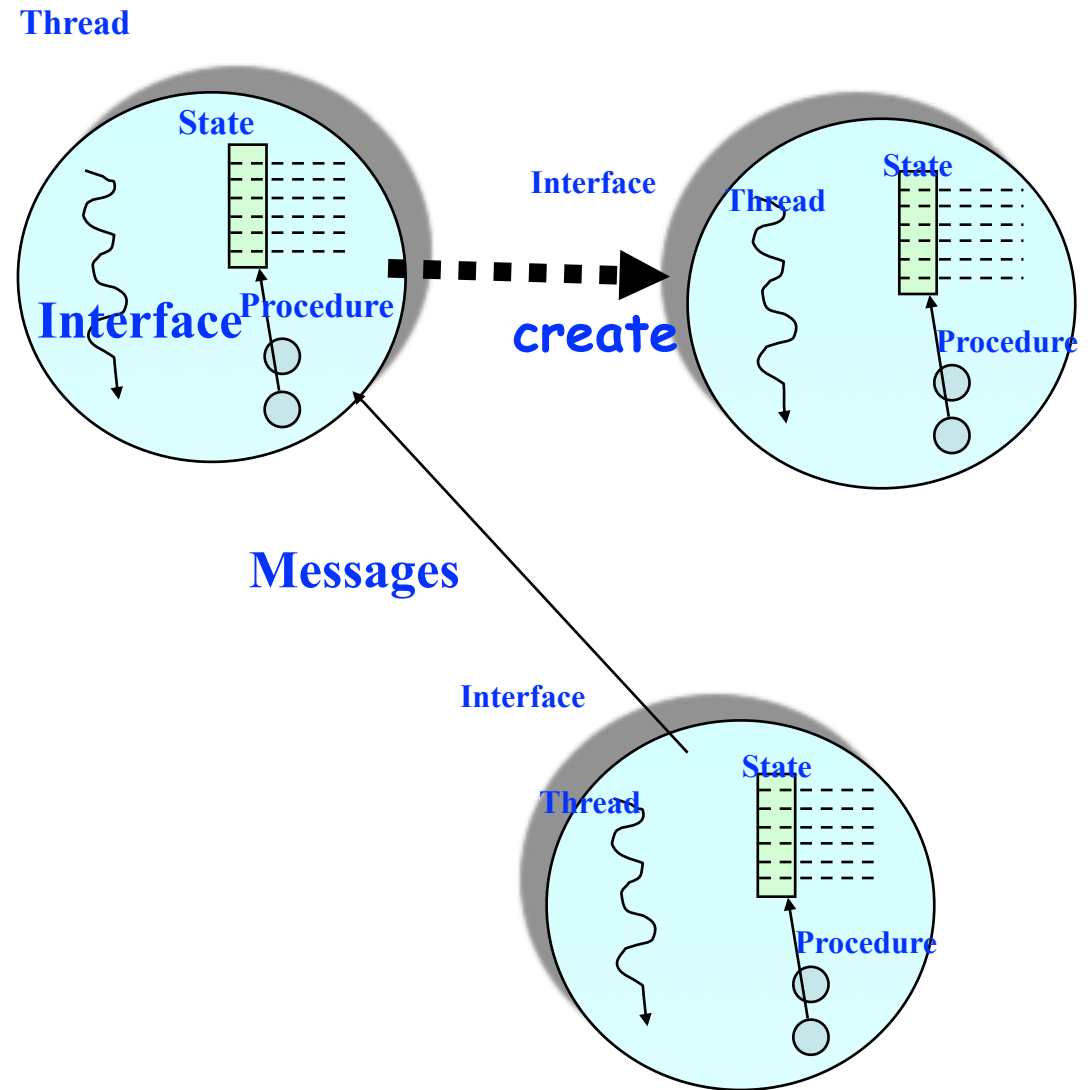
Actors: an alternative approach to isolation

- An actor is an autonomous, interacting component of a parallel system.
- An actor has:
 - an immutable identity (name, global id)
 - *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**
 - **send messages**
 - **change local state**
 - **create new actors**



Actor Model

- **A message-based concurrency model to manage mutable shared state**
 - **First defined in 1973 by Carl Hewitt**
 - **Further theoretical development by Henry Baker and Gul Agha**
- **Key Ideas:**
 - **Everything is an Actor!**
 - **Analogous to “everything is an object” in OOP**
 - **Encapsulate shared state in Actors**
 - **Mutable state is not shared**
- **Other important features**
 - **Asynchronous message passing**
 - **Non-deterministic ordering of messages**

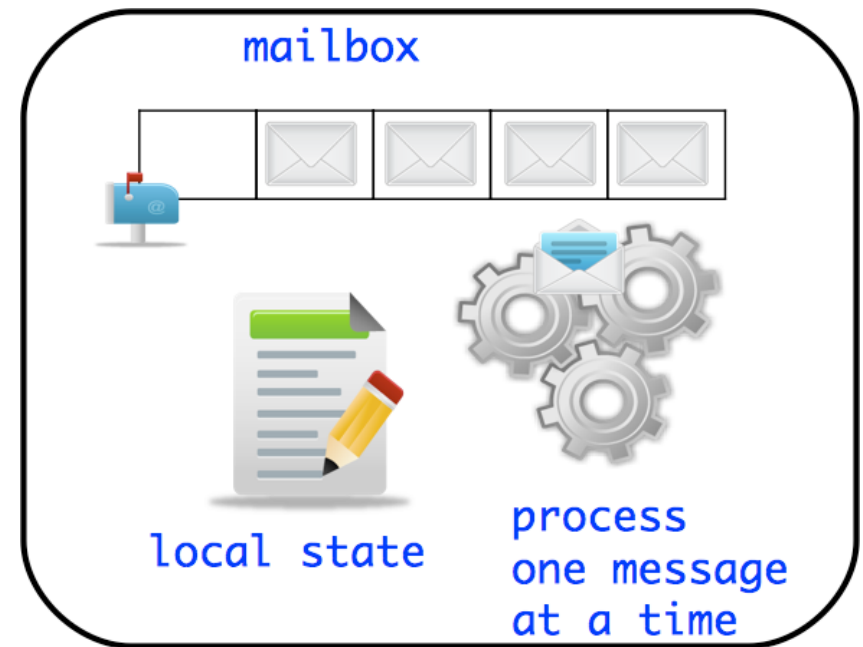


Actor Life Cycle



Actor states

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



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 HJ-Lib

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

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

- Instantiate and start your actor

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

- Send messages to the actor

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

- Use a special message to terminate an actor

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

- Actor execution implemented as async tasks in HJ
- Can use `finish` to await completion of an actor!



Hello World Example

```
1. public class HelloWorld {
2.     public static void main(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");
8.             actor.send(EchoActor.STOP_MSG);
9.         });
10.    }
11.    private static class EchoActor extends Actor<Object> {
12.        static final Object STOP_MSG = new Object();
13.        private int messageCount = 0;
14.        protected void process(final Object msg) {
15.            if (STOP_MSG.equals(msg)) {
16.                println("Message-" + messageCount + ": terminating.");
17.                exit(); // never forget to terminate an actor
18.            } else {
19.                messageCount += 1;
20.                println("Message-" + messageCount + ": " + msg);
21.            }
22.        }
23.    }
24. }
```

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



Integer Counter Example

Without Actors:

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

- Can also use atomic variables instead of isolated construct

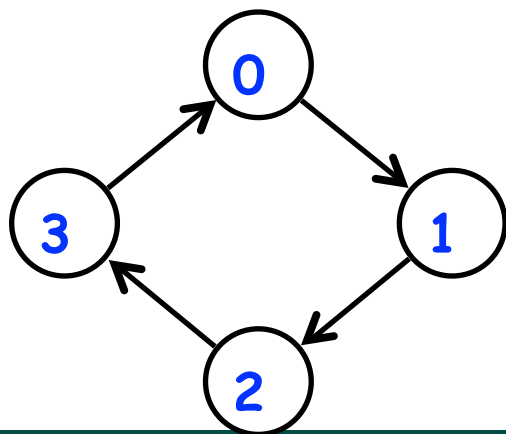
With Actors:

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



ThreadRing (Coordination) Example

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



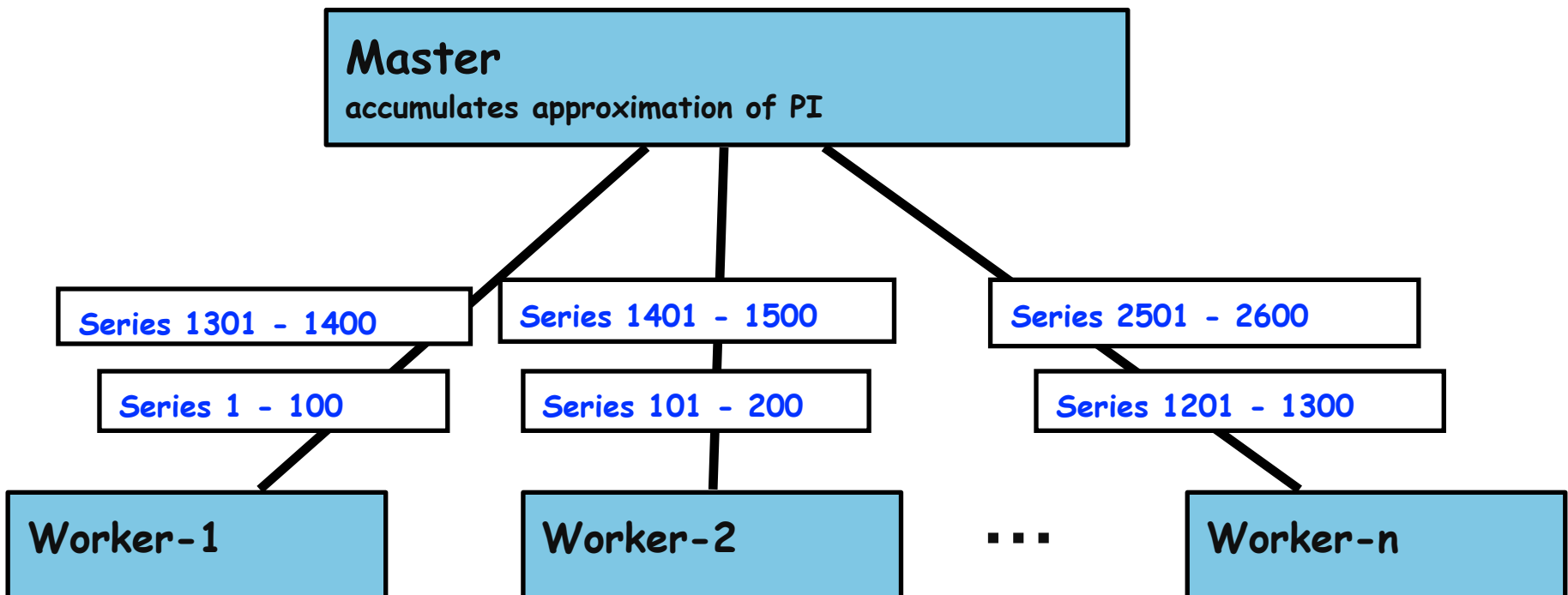
```
14. class ThreadRingActor
15.     extends Actor<Object> {
16.         private Actor<Object> nextActor;
17.         private final int id;
18.         ...
19.         public void nextActor(
20.             Actor<Object> nextActor) {...}
21.         void process(Object theMsg) {
22.             if (theMsg instanceof Integer) {
23.                 Integer n = (Integer) theMsg;
24.                 if (n > 0) {
25.                     println("Thread-" + id +
26.                         " active, remaining = " + n);
27.                     nextActor.send(n - 1);
28.                 } else {
29.                     println("Exiting Thread-" + id);
30.                     nextActor.send(-1);
31.                     exit();
32.                 }
33.             } else {
34.                 /* ERROR - handle appropriately */
35.             }
36.         }
37.     }
38. }
```



Pi Computation Example

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

- Use Master-Worker technique:



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



Pi Calculation --- Master Actor

```
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.     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()
```



Pi Calculation --- Master Actor (contd)

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



Pi Calculation --- Worker Actor

```
1.  class Worker extends Actor<Object> {
2.      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.     private double calculatePiFor(int start, int end) {
12.         double acc = 0.0;
13.         for (int i = start; i < end; i++) {
14.             acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
15.         }
16.         return acc;
17.     }
18. } // Worker
```



Limitations of Actor Model

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

==> Some of these limitations can be overcome by using a hybrid model that combines task parallelism with actors (more on this in the next lecture!)

