Introduction to the Actor Model

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Worksheet #20: Sequential->Parallel Spanning Tree Algorithm

Insert finish, async, and atomic (includes a compareAndSet) constructs (pseudocode is fine) to convert the sequential spanning tree algorithm to a parallel algorithm

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

root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
Work-Sharing Pattern using AtomicInteger

1. import java.util.concurrent.atomic.AtomicInteger;
2. ...
3. String[] X = ... ; int numTasks = ...; int j;
4. int[] taskId = new int[X.length];
5. AtomicInteger a = new AtomicInteger();
6. ...
7. finish(() -> {
8.   for (int i=0; i<numTasks; i++)
9.     async(() -> {
10.    do {
11.      j = a.getAndAdd(1);
12.     // can also use a.getAndIncrement()
13.      if (j >= X.length) break;
14.      taskId[j] = i; // Task i processes string X[j]
15.      ...
16.    } while (true);
17.  });
18. }); // finish-for-async
Work-Sharing Pattern using AtomicInteger

1. import java.util.concurrent.atomic.AtomicInteger;
2. . . .
3. String[] X = ...; int numTasks = ...;
4. int[] taskId = new int[X.length];
5. AtomicInteger a = new AtomicInteger();
6. . . .
7. finish(() -> {
8. for (int i=0; i<numTasks; i++)
9. async(() -> {
10. do {
11. int j = a.getAndAdd(1);
12. // can also use a.getAndIncrement()
13. if (j >= X.length) break;
14. taskId[j] = i; // Task i processes string X[j]
15. . .
16. } while (true);
17. });
18. }); // finish-for-async
Actors: an alternative approach to isolation, atomics

- 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 Life Cycle

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
Using Actors in HJ-Lib

- 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

// Next lecture
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

```java
public class HelloWorld {
    public static void main(final String[] args) {
        finish(() -> {
            EchoActor actor = new EchoActor();
            actor.start(); // don't forget to start the actor
            actor.send("Hello"); // asynchronous send (returns immediately)
            actor.send("World"); // Non-actors can send messages to actors
            actor.send(EchoActor.STOP_MSG);
        });
        println("EchoActor terminated.")
    }

    private static class EchoActor extends Actor<Object> {
        static final Object STOP_MSG = new Object();
        private int messageCount = 0;
        protected void process(final Object msg) {
            if (STOP_MSG.equals(msg)) {
                println("Message-" + messageCount + ": terminating.");
                exit(); // never forget to terminate an actor
            } else {
                messageCount += 1;
                println("Message-" + messageCount + ": " + msg);
            }
        }
    }

    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.}

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. }

14
ThreadRing (Coordination) Example

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

1. ```class ThreadRingActor```
2. ```extends Actor<Integer> {``
3. ```private Actor<Integer> nextActor;``
4. ```private final int id;``
5. ```...```
6. ```public void nextActor(Actor<Object> nextActor) {...}```
7. ```protected void process(Integer n) {``
8. ```if (n > 0) {``
9. ```println("Thread-" + id + " active, remaining = " + n);``
10. ```nextActor.send(n - 1);``
11. ```} else {``
12. ```println("Exiting Thread-"+ id);``
13. ```nextActor.send(-1);``
14. ```exit();``
15. ```} } }```
Announcements & Reminders

• Quiz for Unit 4 is due Friday, March 6th at 11:59pm
• Lab 5 is tomorrow (setup before lab, try logging into NOTS)
• Quiz for Unit 5 will be in class on Wednesday, March 11th