COMP 322: Fundamentals of Parallel Programming

Lecture 23: Actors (continued)

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

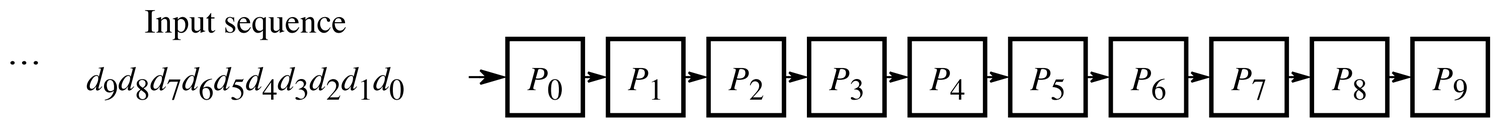
Announcements

- Checkpoint 2 for Homework 3 is due Friday, March 27th at 11:59pm
- The entire written + programming (Checkpoint #3) is due by Friday, April 3rd at 11:59pm
- Quiz for Unit 5 is due Monday, March 30th at 11:59pm
- Lab 5 is due Monday, March 30th at 11:59pm
  - —Commit solution and slurm output file to svn to get checked off
- You now have 5 slip days (up from 3)
- There will now be only 4 homework assignments (down from 5)
- Worksheets from now on will not be graded
- As of now, the COMP 322 final exam is still May 6th from 9am-12pm:
  - —Please let me know if this time doesn't work for you
  - —Scope of final exam (Exam 2) will be limited to Lectures 19 and later

Worksheet #22: Analyzing Parallelism in an Actor Pipeline

Consider a three-stage pipeline of actors (as in slide 5), set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below.

Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.



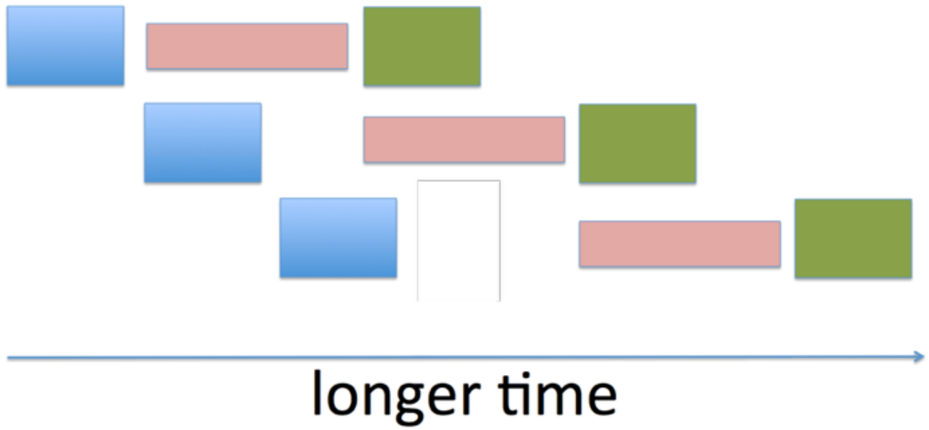

```
1.    protected void process(final Object msg) {
2.        if (msg == null) {
3.            exit();
4.        } else {
5.            doWork(1); // unit work
6.        }
7.        if (nextStage != null) {
8.            nextStage.send(msg);
9.        }
10.    }
```

WORK = 300, CPL = 102

Pipeline and Actors

Pipelined Parallelism:

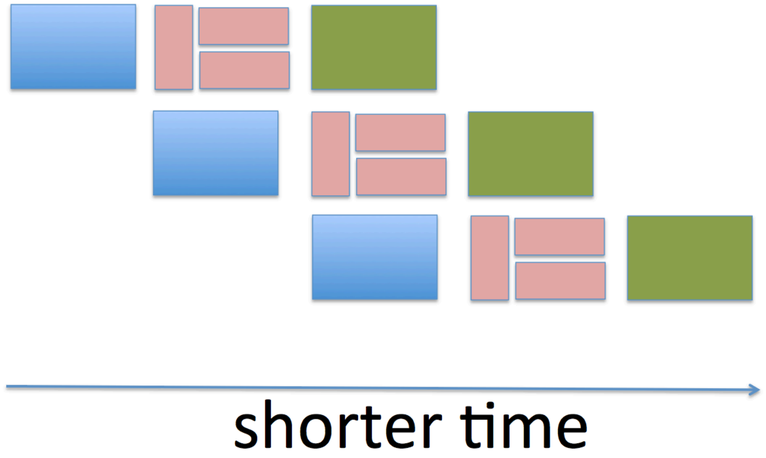
- Each stage can be represented as an actor
- Stages need to ensure ordering of messages while processing them
- Slowest stage is a throughput bottleneck



Motivation for Parallelizing Actors

Pipelined Parallelism:

- Reduce effects of slowest stage by introducing task parallelism.
- Increases the throughput.



Parallelism within an Actor's process() method

- Use `finish` construct within `process()` body and spawn child tasks
- Take care not to introduce data races on local state!

```
1.class ParallelActor extends Actor<Message> {
2.   void process(Message msg) {
3.       finish(() -> {
4.           async(() -> { S1; });
5.           async(() -> { S2; });
6.           async(() -> { S3; });
7.       });
8.    }
9. }
```

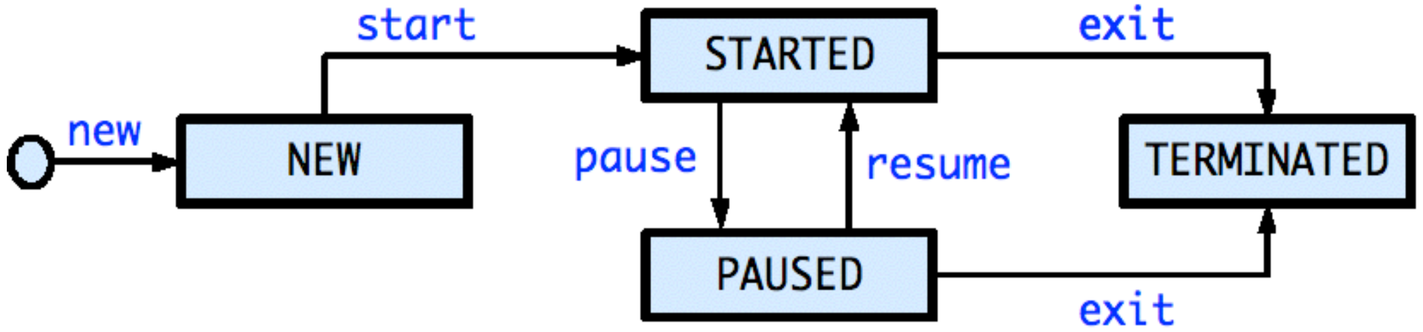

Example of Parallelizing Actors

```
1. class ArraySumActor extends Actor<Object> {
2.     private double resultSoFar = 0;
3.     @Override
4.     protected void process(final Object theMsg) {
5.       if (theMsg != null) {
6.         final double[] dataArray = (double[]) theMsg;
7.         final double localRes = doComputation(dataArray);
8.         resultSoFar += localRes;
9.       } else { ... }
10.    }
11.    private double doComputation(final double[] dataArray) {
12.      final double[] localSum = new double[2];
13.      finish(() -> { // Two-way parallel sum snippet
14.        final int length = dataArray.length;
15.        final int limit1 = length / 2;
16.        async(() -> {
17.          localSum[0] = doComputation(dataArray, 0, limit1);
18.        });
19.        localSum[1] = doComputation(dataArray, limit1, length);
20.      });
21.      return localSum[0] + localSum[1];
22.    }
23.  }
```


Parallelizing Actors in HJ-Lib

- Two techniques:
  - Use finish construct to wrap `asyncs` in message processing body
    - Finish ensures all spawned `asyncs` complete before next message returning from `process()`
  - Allow escaping `asyncs` inside `process()` method
    - WAIT! Won't escaping `asyncs` violate the one-message-at-a-time rule in actors
    - Solution: Use `pause` and `resume`

State Diagram for Extended Actors with Pause-Resume



- Paused state: actor will not process subsequent messages until it is resumed
- Resume actor when it is safe to process the next message
- Messages can accumulate in mailbox when actor is in PAUSED state

NOTE: Calls to exit(), pause(), resume() only impact the processing of the next message, and not the processing of the current message. These calls should just be viewed as "state change" operations.

Actors: pause and resume

- `pause()` operation:
  - Is a non-blocking operation, i.e. allows the next statement to be executed.
  - Calling `pause()` when the actor is already paused is a no-op.
  - Once paused, the state of the actor changes and it will no longer process messages sent (i.e. call `process(message)`) to it until it is resumed.

- `resume()` operation:
  - Is a non-blocking operation.
  - Calling `resume()` when the actor is not paused is an error, the HJ runtime will throw a runtime exception.
  - Moves the actor back to the STARTED state
    - the actor runtime spawns a new asynchronous thread to start processing messages from its mailbox.

Parallelizing Actors in HJ-Lib

Allow escaping `asyncs` inside process():

```
1. class ParallelActor2 extends Actor<Message> {
2.   void process(Message msg) {
3.     pause(); // process() will not be called until a resume() occurs
4.     async(() -> { S1; }); // escaping async
5.     async(() -> { S2; }); // escaping async
6.     async(() -> {
7.       // This async must be completed before next message
8.       // Can also use async-await if you want S3 to wait for S1 & S2
9.       S3;
10.       resume();
11.     });
12.   }
13. }
```


Synchronous Reply using Pause/Resume

- Actors are asynchronous, sync. replies require blocking operations
- We need notifications from recipient actor on when to resume
- Resumption needs to be triggered on sender actor
  - Use DDFs and `asyncAwait`

```
1.class SynchronousSenderActor
2.     extends Actor<Message> {
3.   void process(Msg msg) {
4.     ...
5.     DDF<T> ddf = newDDF();
6.     otherActor.send(ddf);
7.     pause(); // non-blocking
8.     asyncAwait(ddf, () -> {
9.       T synchronousReply = ddf.get();
10.       println("Response received");
11.       resume(); // non-blocking
12.     });
13.     ...
14.} }
```

```
1.class SynchronousReplyActor
2.     extends Actor<DDF> {
3.   void process(DDF msg) {
4.     ...
5.     println("Message received");
6.     // process message
7.     T responseResult = ...;
8.     msg.put(responseResult);
9.     ...
10.} }
```