# COMP 322: Fundamentals of Parallel Programming

# Lecture 23: Actors (continued)

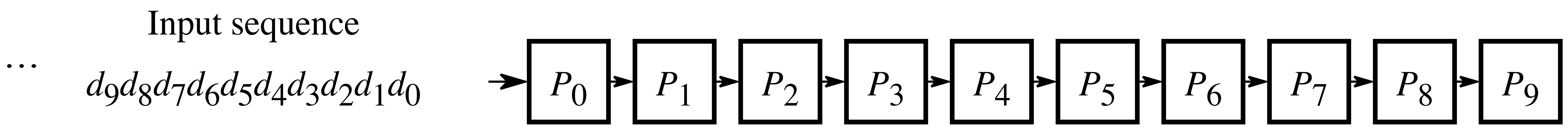
Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

# 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.**



```
protected void process(final Object msg) {
    if (msg == null) {
        exit();
    } else {
        doWork(1); // unit work
    }
    if (nextStage != null) {
        nextStage.send(msg);
    }
}
```

**WORK = 300, CPL = 102**

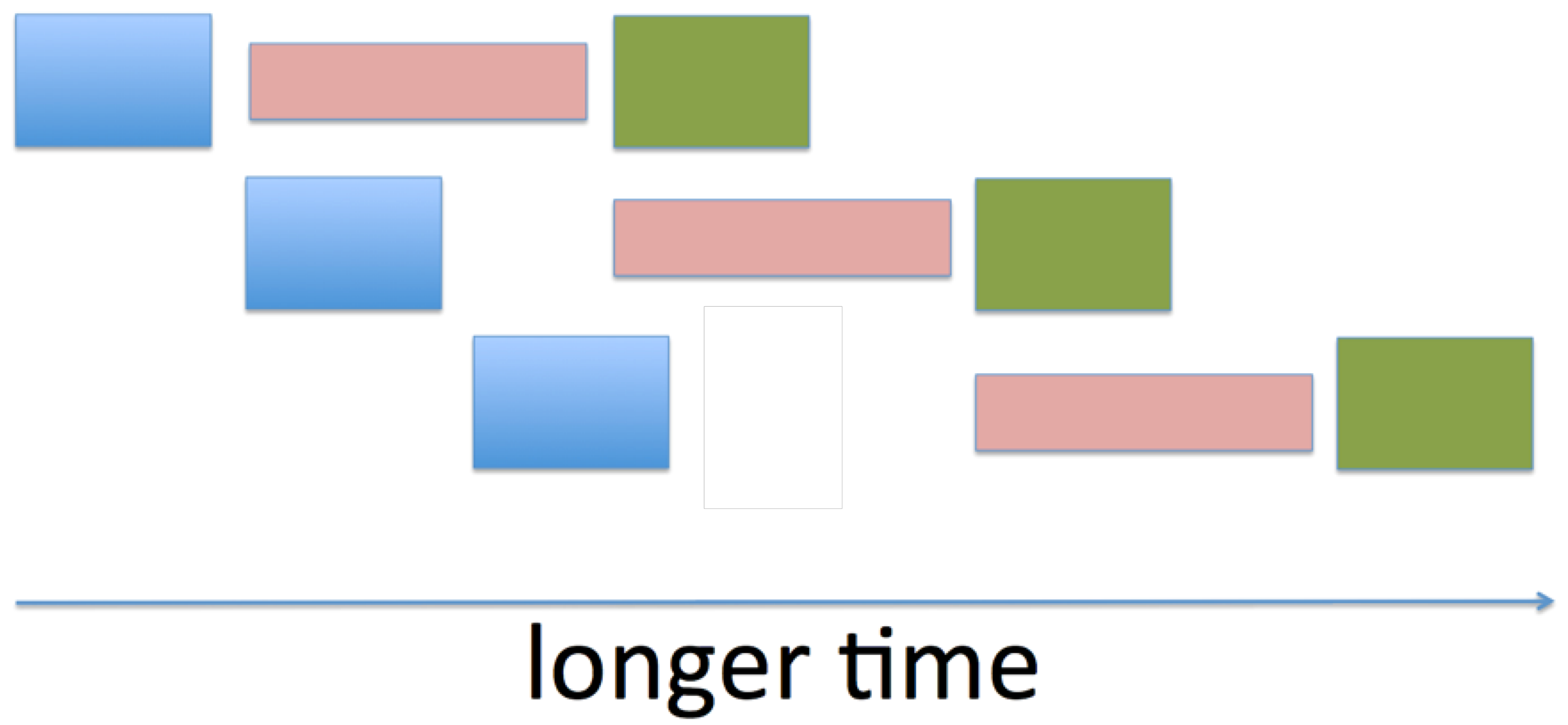
# Announcements

- Quiz for Unit 5 is due today at 11:59pm
- Lab 5 is due tomorrow at 12pm (noon)
- Lab 6 is this week (run on local machine)

# Pipeline and Actors

Pipelined Parallelism:

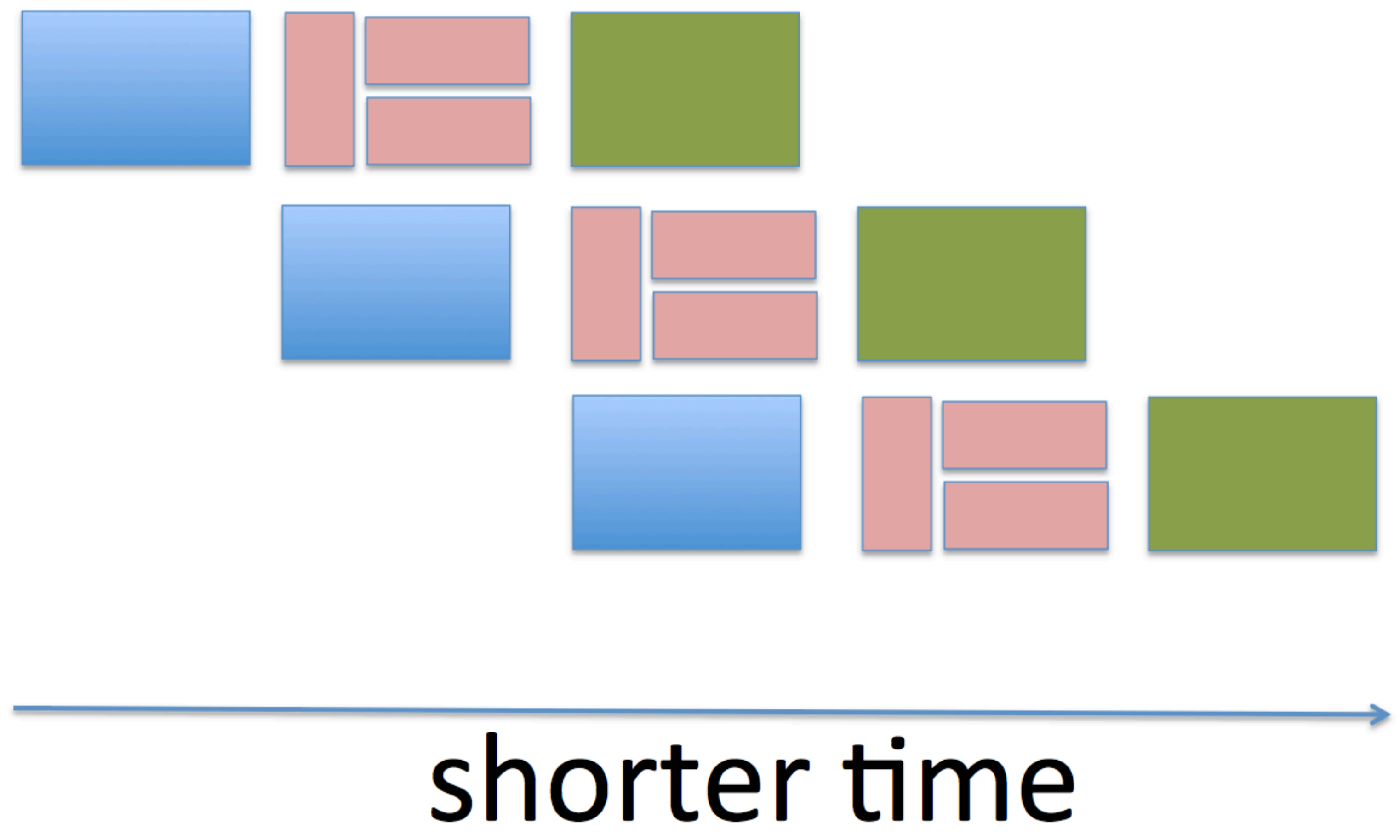
- Each stage can be represented as an actor
- Stages need to ensure ordering of messages while processing them
- Slowest stage is a throughput bottleneck



# 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!

```
class ParallelActor extends Actor<Message> {
  void process(Message msg) {
     finish(() -> {
        async(() -> { S1; });
        async(() -> { S2; });
        async(() -> { S3; });
     });
   }
 }
```

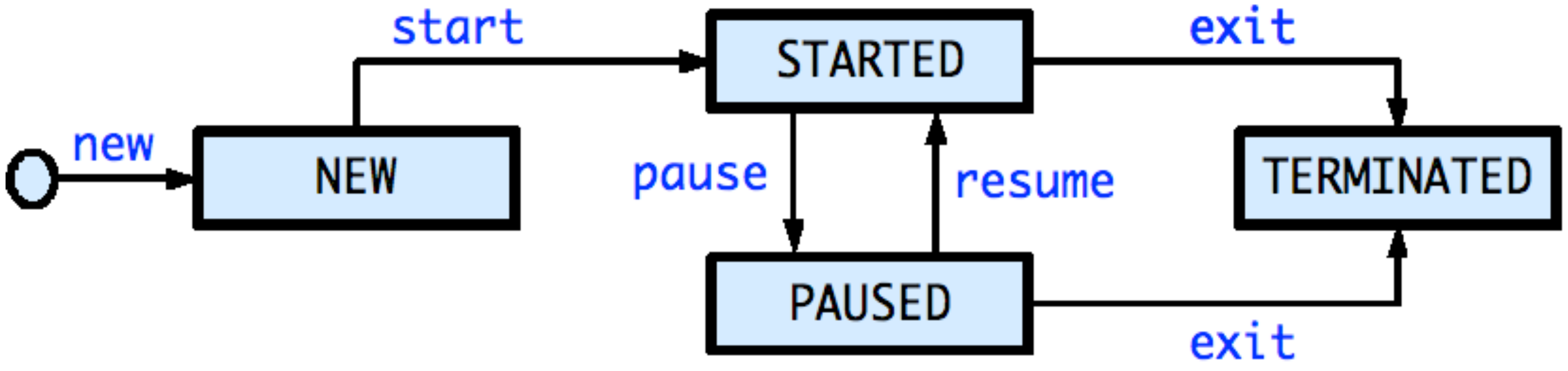
# Example of Parallelizing Actors

```
class ArraySumActor extends Actor<Object> {
   private double resultSoFar = 0;
   @Override
   protected void process(final Object theMsg) {
     if (theMsg != null) {
       final double[] dataArray = (double[]) theMsg;
       final double localRes = doComputation(dataArray);
       resultSoFar += localRes;
     } else { ... }
    }
    private double doComputation(final double[] dataArray) {
      final double[] localSum = new double[2];
      finish(() -> { // Two-way parallel sum snippet
        final int length = dataArray.length;
        final int limit1 = length / 2;
        async(() -> {
          localSum[0] = doComputation(dataArray, 0, limit1);
        });
        localSum[1] = doComputation(dataArray, limit1, length);
      });
      return localSum[0] + localSum[1];
    }
  }
```

# 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() 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.

# Actors: 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():

```
class ParallelActor2 extends Actor<Message> {
  void process(Message msg) {
    pause(); // process() will not be called until a resume() occurs
    async(() -> { S1; }); // escaping async
    async(() -> { S2; }); // escaping async
    async(() -> {
      // This async must be completed before next message
      // Can also use async-await if you want S3 to wait for S1 & S2
      S3;
       resume();
     });
   }
}
```

# Worksheet #23: Synchronized Reply using Pause/Resume

Actors don’t normally require synchronization with other actors. However, sometimes we might want actors to be in synch with one another. Using a DDF and pause/resume, ensure that the SynchSenderActor doesn’t process the next message until notified by the SyncReplyActor that the message was received and processed.

```
class SynchSenderActor
    extends Actor<Message> {
  private Actor otherActor = …
  void process(Msg msg) {
    ...
    DDF<T> ddf = newDDF();
    otherActor.send(ddf);
    println("Response received");
    ...
} }
```

```
class SynchReplyActor
    extends Actor<DDF> {
  void process(DDF msg) {
    ...
    println("Message received");
    // process message
    T responseResult = ...;
    ...
} }
```