Selectors: Actors with Multiple Guarded Mailboxes

AGERE! 2014
Monday, October 20, 2014

Shams Imam, Vivek Sarkar
shams@rice.edu, vsarkar@rice.edu
Rice University

Introduction

- Multicore processors are now ubiquitous
- Parallelism is the future of computing
- Actor Model regained popularity
  - Erlang – flagship language
- Actors give stronger guarantees about concurrent code
  - Data race freedom
  - Location transparency

Motivation

- Actor Model (AM) is not a silver bullet
- Synchronization and coordination harder
  - Compared to shared-memory model
  - Coordination patterns involving multiple actors are particularly difficult
- Until message is processed solutions may require the actor to
  - Buffer messages
  - Resend messages to itself

Goals

- Simplify writing of synchronization and coordination patterns
  - Using an extension to Actors
- Patterns of interest
  - synchronous request-reply
  - join patterns common in streaming applications
  - priorities in message processing
  - variants of reader-writer concurrency
  - producer-consumer with bounded buffer
- Future Work: Other patterns

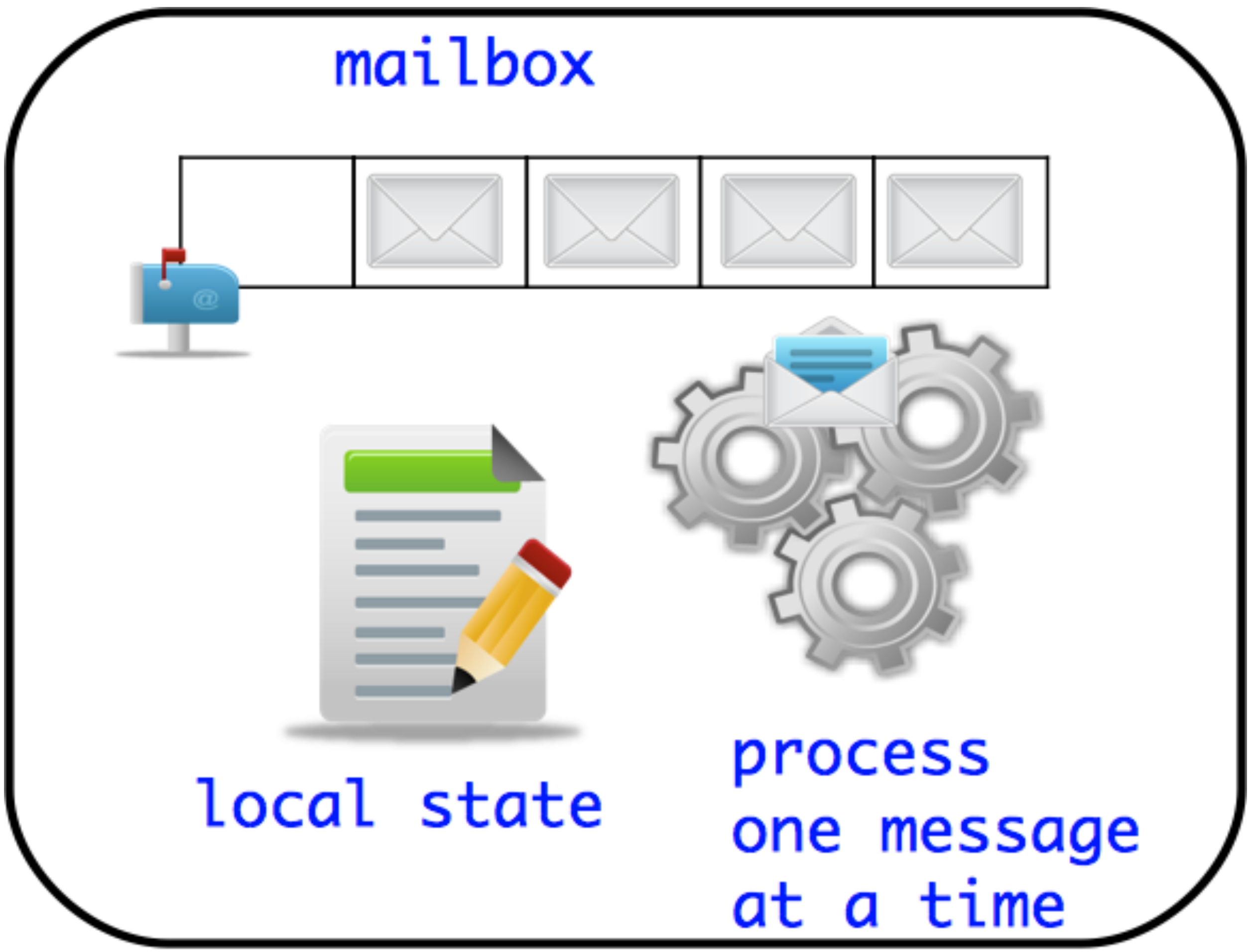
Outline

- Introduce Selector extension
- Join patterns in streaming applications
- Synchronous request-reply pattern
- Reader Writer Concurrency
- Performance Evaluation
- Summary and Future Work

Actor Model

- First defined in 1973 by Carl Hewitt
- A message-based concurrency model
- An Actor encapsulates mutable state
- Processes one message at a time
- Actors coordinate using asynchronous messaging

Actor Diagrammatic Representation



Actor / Selector Similarities

- A message-based concurrency model
- A Selector encapsulates mutable state
- Processes one message at a time
- Selectors coordinate using asynchronous messaging
- Benefits of modularity from the AM are preserved
- Data locality properties of the AM continue to hold

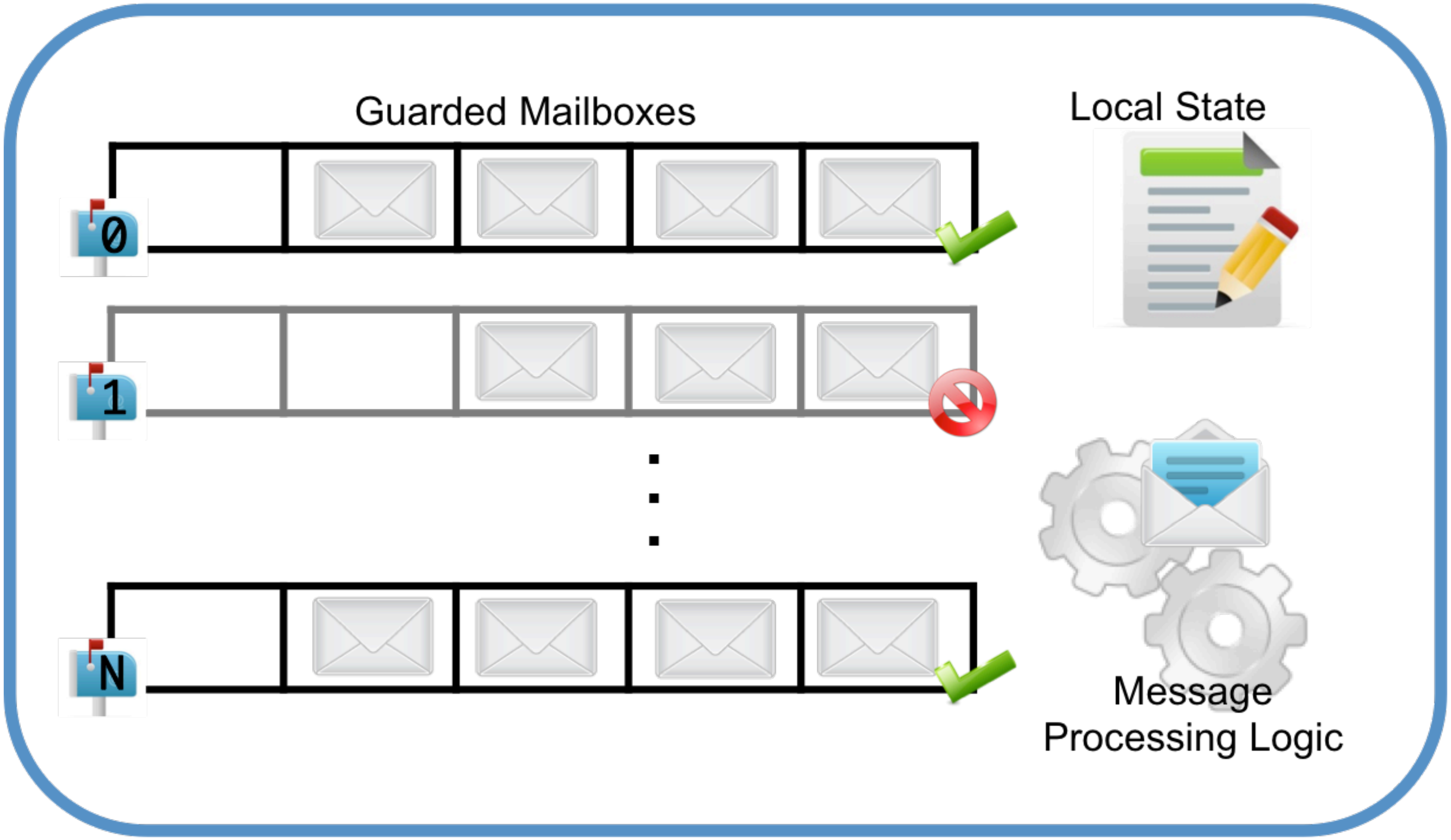
Actor / Selector Lifecycle

- NEW: actor instance has been created
- STARTED: actor can receive and process messages sent to it
- TERMINATED: actor will no longer process messages sent to it

Actor / Selector Differences

- Multiple mailboxes
  - Messages can be concurrently sent to different mailboxes
- Each mailbox maintains a mutable guard
  - Mailbox can always receive messages
  - Guard changed using `enable`/`disable` operations
  - Affects which mailboxes provides next message to process
- Actor is a Selector with a single mailbox
  - Guard on the mailbox always enabled

Selector Diagrammatic Representation



Sending messages to a Selector

- The send operation receives two arguments:
  - Target mailbox name
  - Actual message to send
- Flexibility in determining the target mailbox
  - By the sender entity
  - By the recipient selector
  - Hybrid policy using combination of both schemes
- Message ordering preserved between same sender-receiver pair in a given mailbox

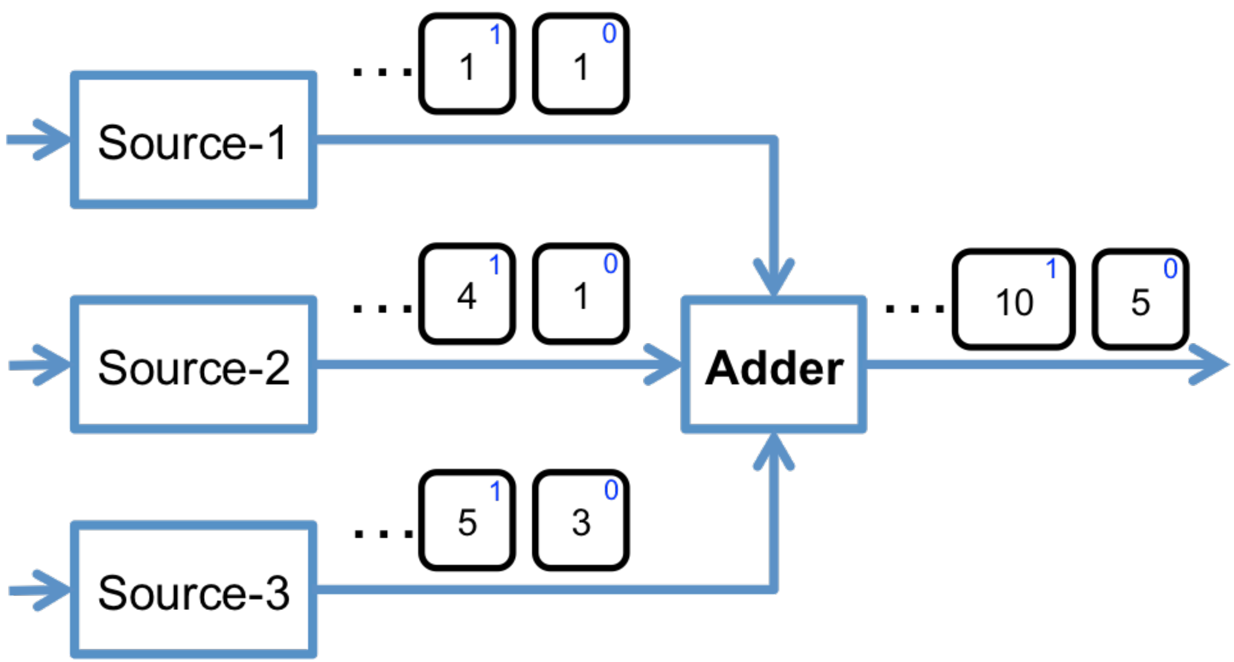
HelloWorld Primer

```
object SelectorPrimer extends App {
  val s = new EchoSelector()
  s.start()
  s.send(MBX_1, "Hello")
  s.send(MBX_2, "World")
}

class EchoSelector extends Selector {
  var msgProc = 0
  disable(MBX_2)
  def process(message: AnyRef) {
    println(message)
    msgProc += 1
    if (msgProc == 1) { enable(MBX_2) }
    else if (msgProc == 2) { exit() }
  }
}
```

Join Patterns in Streaming Applications

- Messages from two or more data streams are combined together into a single message
- Joins need to match inputs from each source
- Wait until all corresponding inputs become available



Actor-based Solution

- Actors lack guarantee of which message is processed next
- Data structure to track in-flight items from various sources
- Wait for items from all sources for the oldest (lowest) sequence number to be available
- Aggregator actor then reduces the items into a single value and forwards it to the consumer

Selector-based Solution

- One mailbox for each source
- Sources send their messages to corresponding mailboxes
- Two policies…

Arbitrary Order Policy

- Disable mailbox of source as an item is processed
- Disallow processing items not part of current sequence
- Reset when items from all sources have been received
- Non-determinism from ordering of messages processed

Round-Robin Order Policy

- Initially disables all the mailboxes except the first mailbox
- As each item is received the current mailbox is disabled
- Mailbox of the next source in round-robin order is enabled
- Determinism from message processing order

Synchronous Request-Response Pattern

- Requester sends a message to a replier system
- Replier receives and processes the request
- Replier returns a message in response
- Requester can make further progress after receiving response

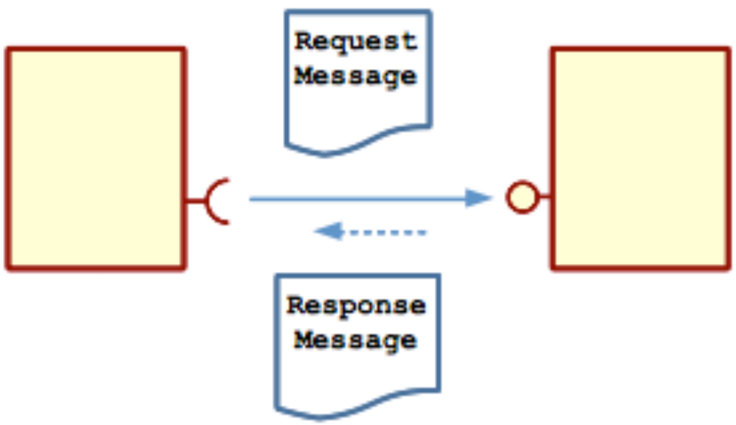


Image source: http://soi-toolkit.googlecode.com/svn/wiki/ConceptsAndDefinitions/EP-RequestResponse-Synchronous.png

Actor-based Solution

- Hard to implement efficiently
- Requestor actor's single mailbox must handle both
  - Response message from replier
  - Other messages sent to it from other actors

Solutions

- Pattern matching on the set of pending messages
  - Increases time for searching next message to process
- Some notion of blocking explicitly and usually limits scalability
- Non-blocking solution stashes messages until reply message found

Selector-based Solution

- Two mailboxes
  - one to receive regular messages
  - one to receives synchronous response messages
- Whenever expecting a synchronous response
  - disables the regular mailbox ensuring next message processed is from reply mailbox

Reader-Writer Concurrency

- Multiple entities accessing a resource, some reading and some writing
- No entity may access the resource for reading or writing while another process is in the act of writing to it
- The first readers-writers variant:
  - No read request shall be kept waiting if the resource is currently opened for reading
- The second readers-writers variant:
  - No write request, once added to the message queue, shall be kept waiting longer than absolutely necessary
- Actors do not support intra-actor concurrency!

Previous Work: Actors + Task Parallelism

- Unify `async-finish` task parallelism and actors
  - All parallel constructs are first class
- Benefits
  - Enable intra-actor parallelism
  - Simplify termination detection
- Implementation: Habanero-Scala Actors

Integrating Task Parallelism with Actors. Shams Imam, Vivek Sarkar. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2012.

Selector-based Solution

- Extend Habanero actors support for intra-actor parallelism
- Spawn separate task for read requests
- Maintain counter for in-flight read tasks
- Write requests
  - Wait for in-flight read tasks to complete
  - Disallow other messages from being processed
  - Enable mailboxes only after write request completes processing

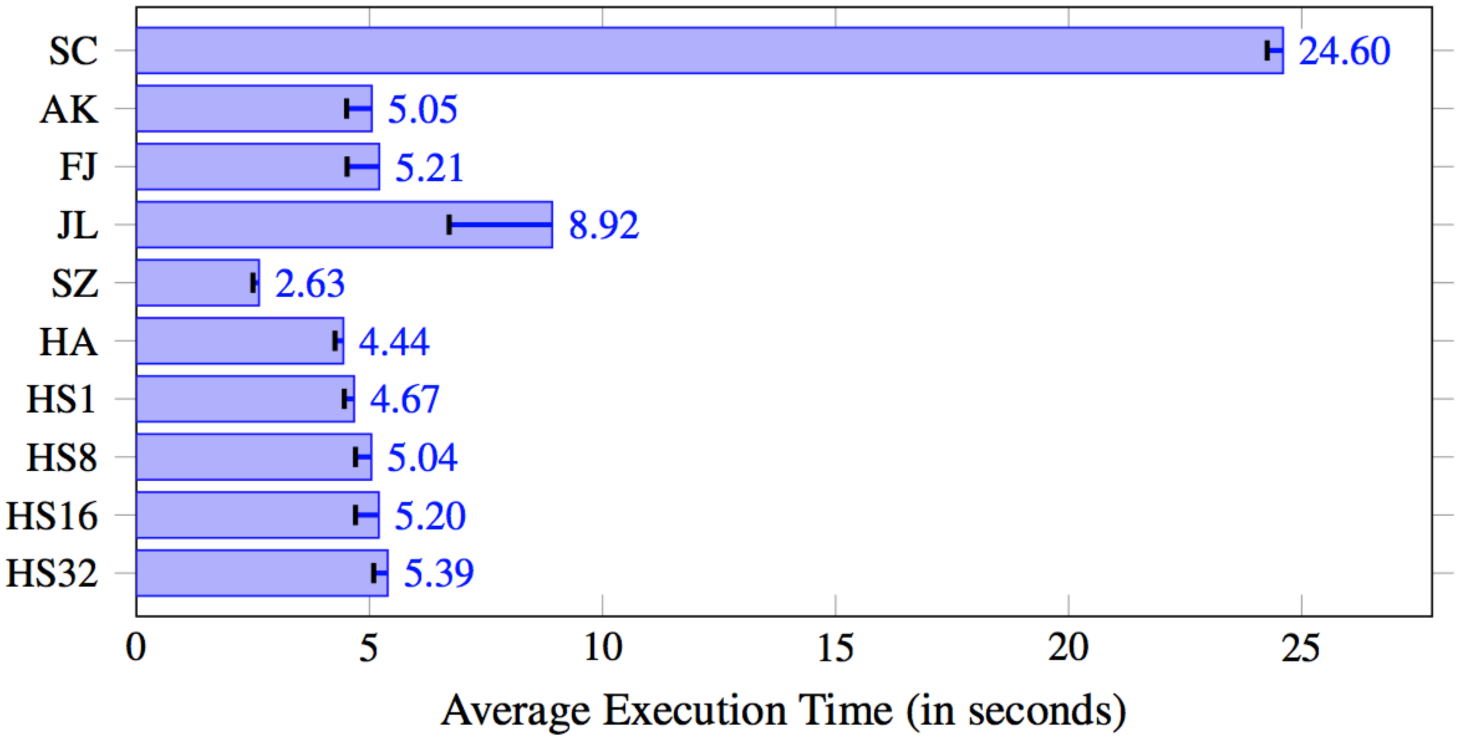
Selector-based Solution

- Arrival-Order variant:
  - Maintain single mailbox for READ and WRITE
- The first readers-writers variant:
  - Maintain two mailboxes
  - READ mailbox gets higher priority than WRITE mailbox
- The second readers-writers variant:
  - Maintain two mailboxes
  - WRITE mailbox gets higher priority than READ mailbox

Experimental Results

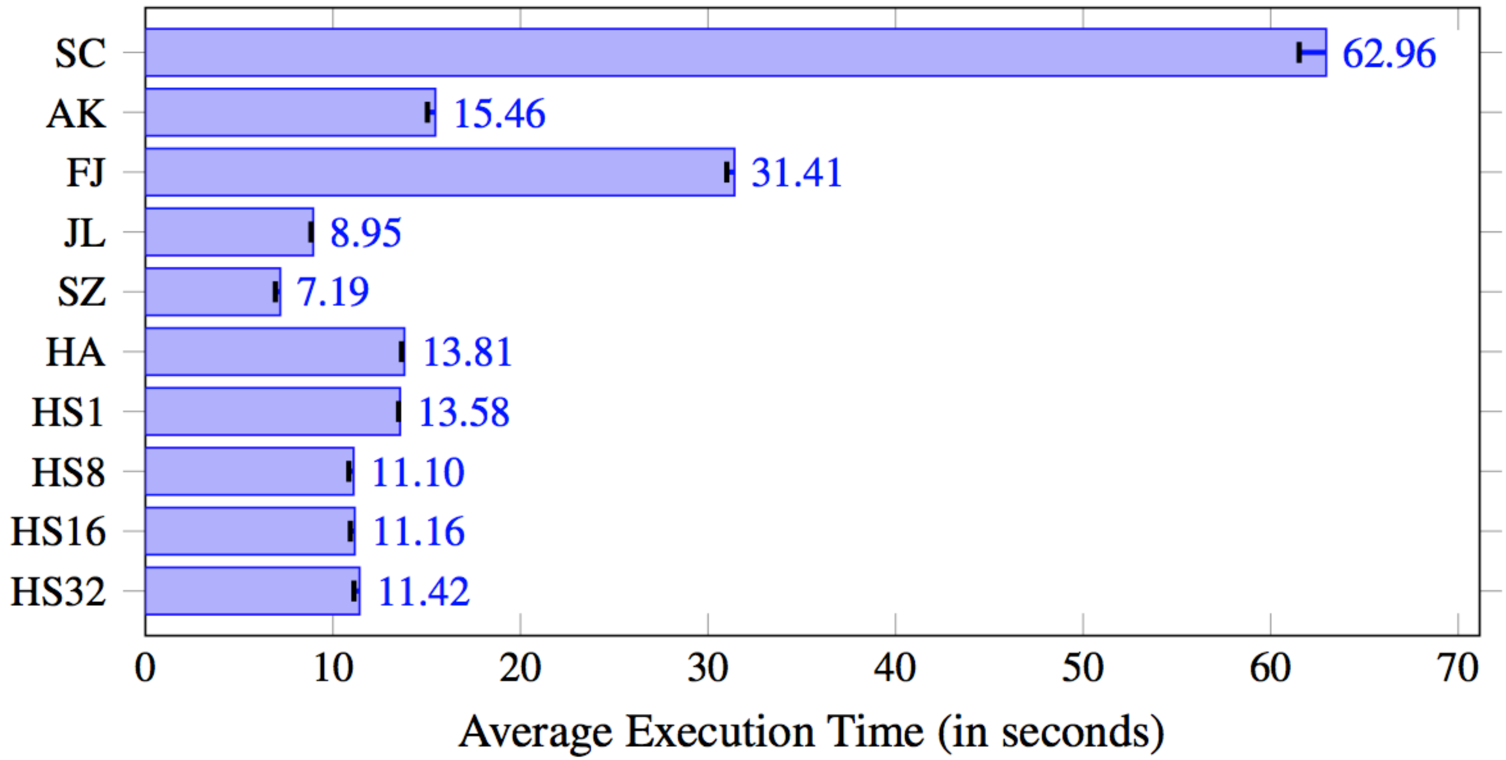
- 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node
- Java Hotspot JDK 1.8.0
- Our implementation:
  - Habanero Selector (HS), pure library impl on Java 8
- Other libraries:
  - Habanero Actors (HA) 0.1.2
  - Scala 2.11.0 actors (SC)
  - Akka 2.3.2 (AK)
  - Functional Java 4.1 (FJ)
  - Jetlang 0.2.12 (JL)
  - Scalaz 7.1.0-M6 (SZ)

Message Throughput (ForkJoin)



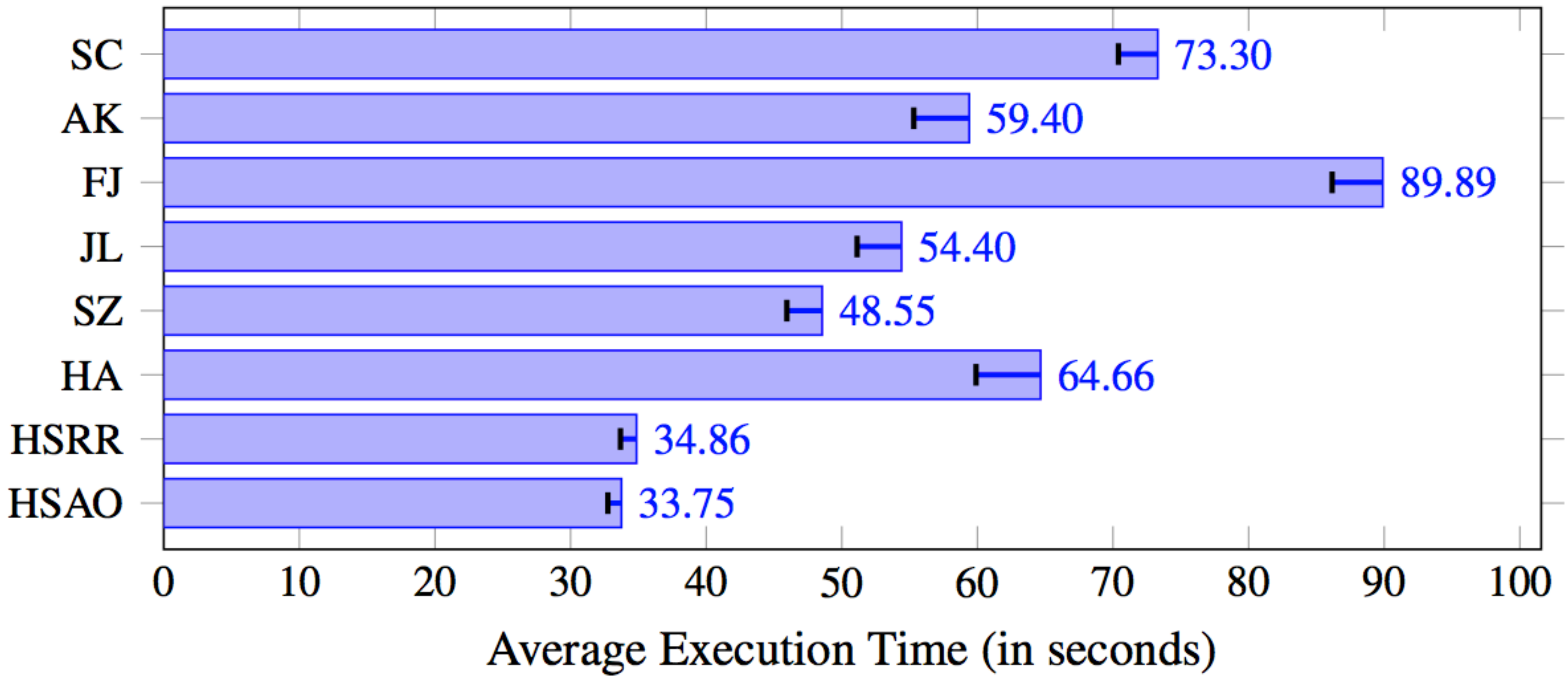
- 60 actors
- Each actor sent 400K messages
- Selector version: message sent round-robin to mailboxes

Mailbox Contention (Chameneos)



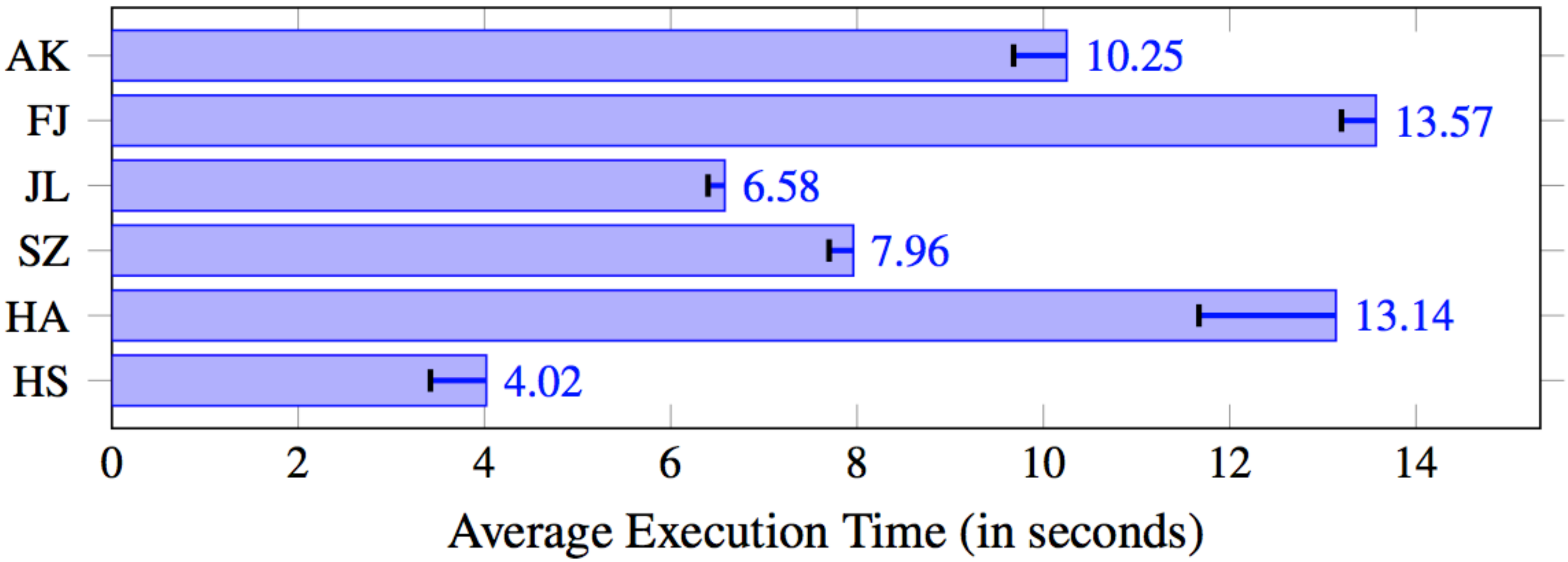
- 500 Chameneos actors
- 8 million meetings

Filter Bank benchmark



- 8-way joins
- 300K data items
- 131,072 columns

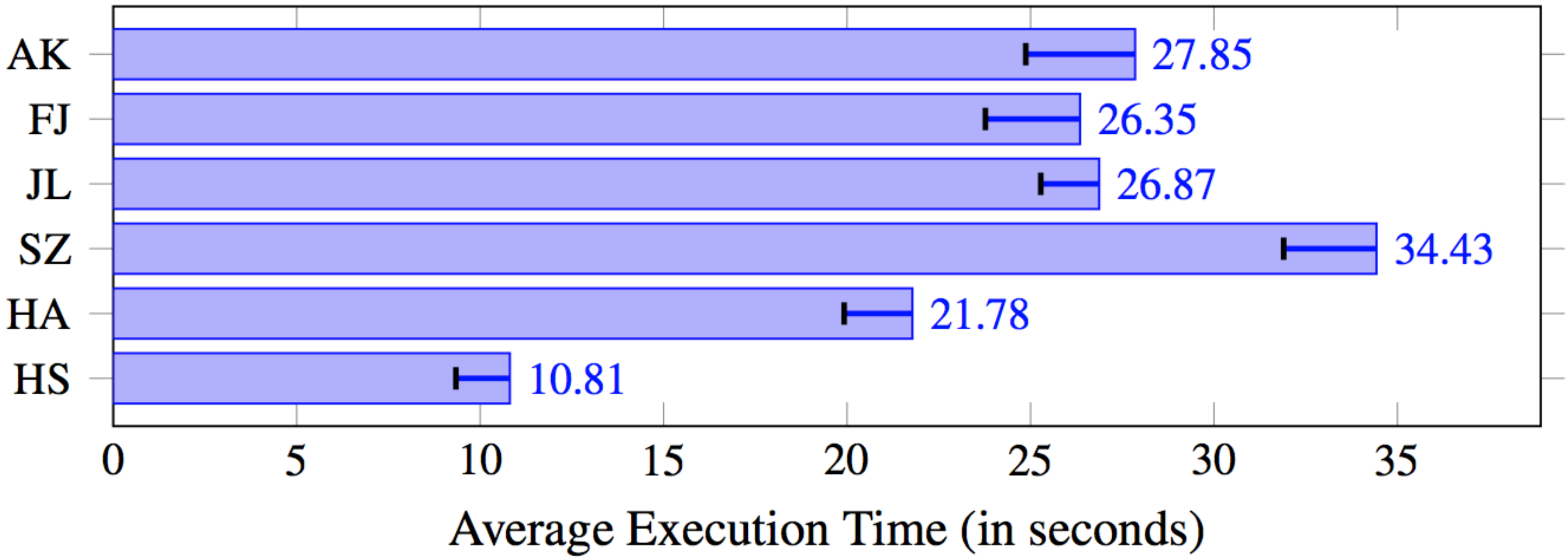
LogisticMap benchmark



$$x_{n+1} = r x_n (1 - x_n)$$

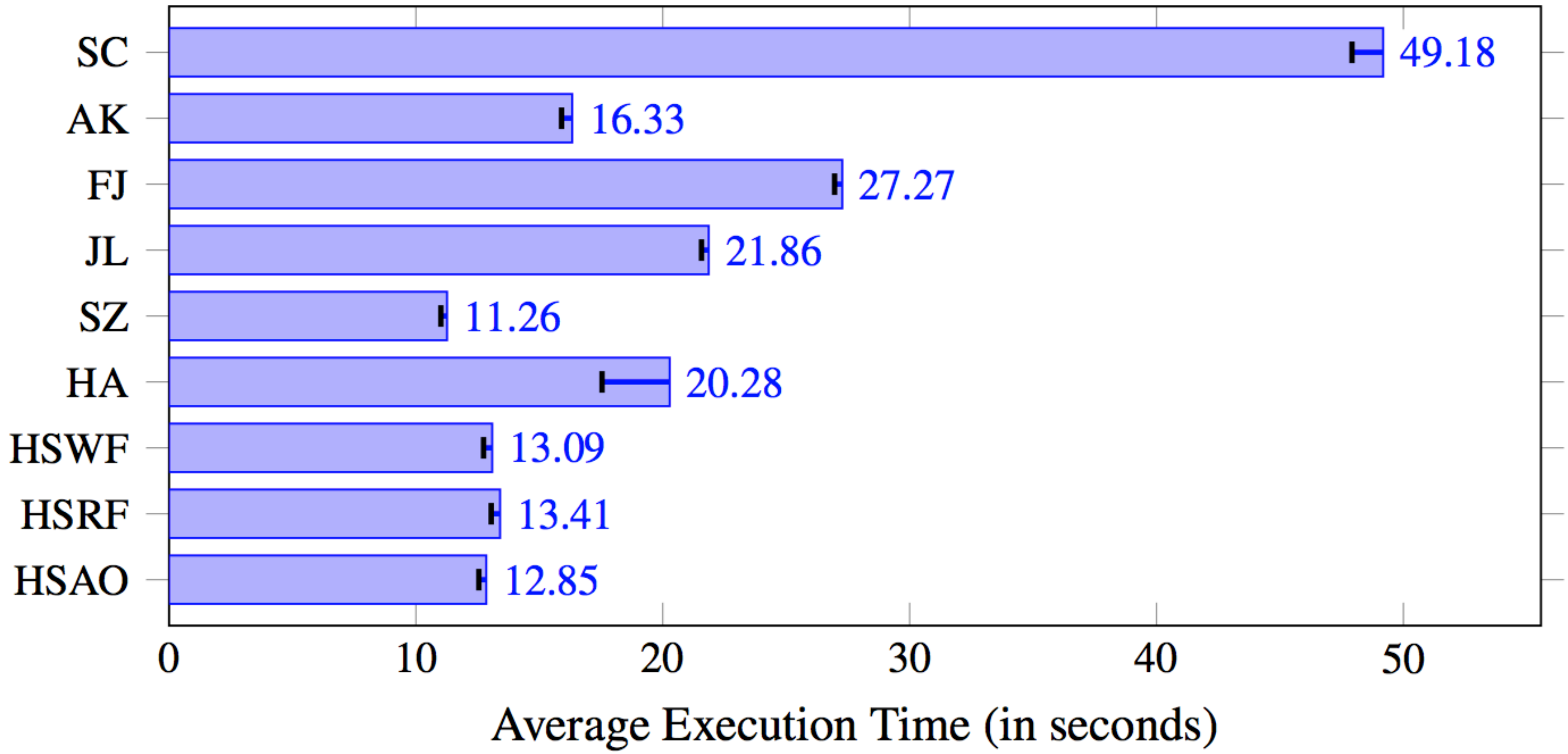
- 150 helper term actors, 150 ratio actors
- 150K terms computed

Bank Transaction benchmark



- 100 bank accounts
- 10 million transactions

Concurrent Dictionary benchmark



- 24 Workers, 400K messages per worker
- Write Percent of 10
- Each operation is O(1)

Concurrent Sorted Linked-List benchmark

- 24 Workers, 15K messages per worker
- Write Percent of 10
- Each operation is O(N)

More in the Paper

- Declarative Style Guards
- Supporting Priorities in Message Processing
- Producer Consumer with Bounded Buffer
- Code Snippets for various solutions
- Additional Performance Results

Related Work

- Pattern matching on receive
  - Enabled-sets by Tomlinson and Singh
  - Scala Actors by Haller
- Aggregator Pattern from Akka
  - Does not match sender
- Message priorities
  - SALSA provides two-level priority
- Parallel Actor Monitors
  - Solves the symmetric reader-writer problems
  - Does not support priorities, hence other variants

Future Work and Availability

- Discover and support further synchronization and coordination patterns
  - Nondeterministic Finite Automata
  - Multiple-message selection patterns
- Experiment with message selection policies

- Implementation available in Habanero-Java library

  https://wiki.rice.edu/confluence/display/PARPROG/HJ+Library
- Benchmarks available as part of Savina Benchmark Suite
  - See talk later today ☺

Summary

- Simplify writing of synchronization and coordination patterns
  - Using a simple extension to Actors
  - Multiple guarded mailboxes
- Patterns of interest
  - Join patterns common in streaming applications
  - Synchronous request-reply
  - Variants of reader-writer concurrency
  - Priorities in message processing
  - Producer-consumer with bounded buffer

Questions

- Simplify writing of synchronization and coordination patterns
  - Using a simple extension to Actors
  - Multiple guarded mailboxes
- Patterns of interest
  - Join patterns common in streaming applications
  - Synchronous request-reply
  - Variants of reader-writer concurrency
  - Priorities in message processing
  - Producer-consumer with bounded buffer

```
import agere.audience.Questions
import agere.audience.Comments
```

Backup-Slides

Declarative Guards

- Move away from imperative style towards functional style
- Register predicated guard expressions on mailboxes
- Mailboxes enabled or disabled after processing each message
- Separates message processing logic from logic to enable or disable mailboxes

Req/Resp Selector-based Solution

```
class ReqRespSelector extends Selector {
  def process(theMsg: AnyRef) {
    theMsg match {
      case m: SomeMessage =>
        // a case where we want a response
        val req = new SomeRequest(this, m)
        anotherActor.send(req)
        // move to reply-blocked state
        disable(REGULAR)
      case someReply: SomeReply =>
        // process the reply (from REPLY mailbox)
        ...
        // resume processing regular messages
        enable(REGULAR)
    } } }
```

Req/Resp Selector-based Solution

```
class ResponseActor extends Actor {
  def process(theMsg: AnyRef) {
    theMsg match {
      case m: SomeRequest =>
        val reply = compute(m.data)
        // send to response mailbox
        sender().send(REPLY, reply)
      ...
    } } }
```

Supporting Priorities in Message Processing

- Messages with a higher priority processed before those with lower priority
  - Even if they were sent earlier
- Useful for recursive data structure traversal algorithms
  - Deeper nodes are more probable to produce results

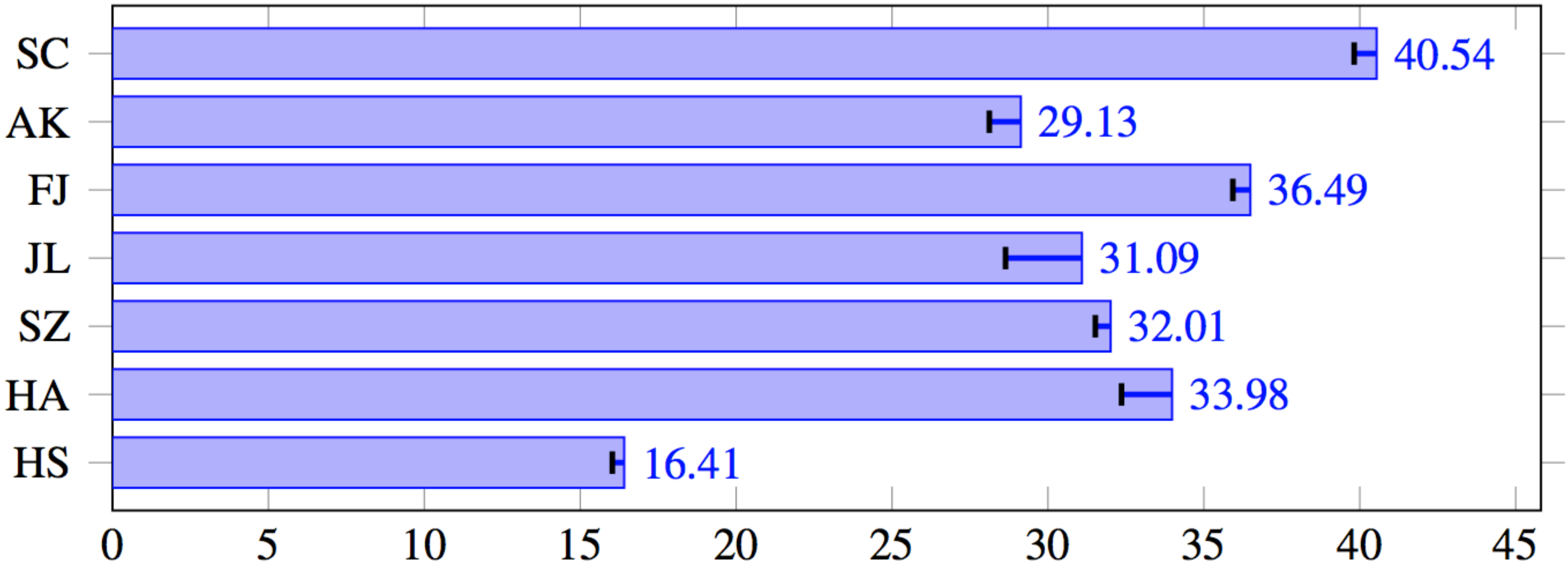
Actor-based Solution

- Normally actors do not support priorities while processing messages
- Use a priority queue to store messages in the mailbox
- Adds overhead to the concurrent mailbox

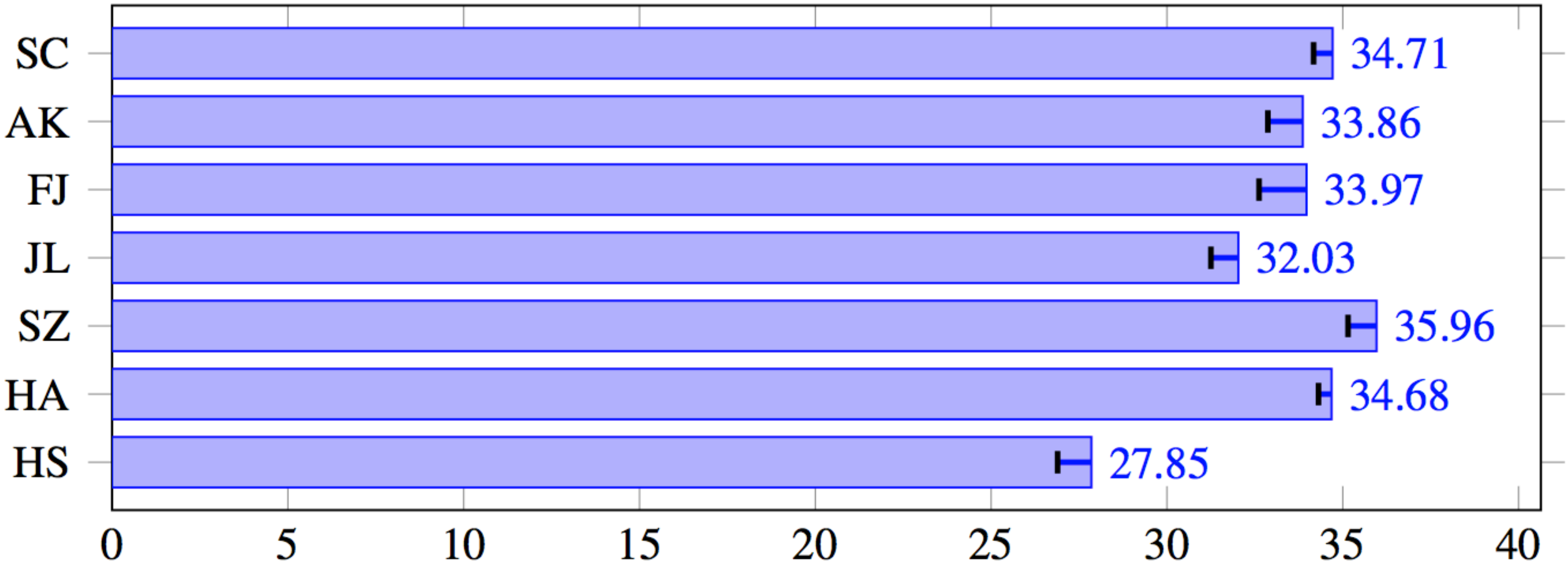
Selector-based Solution

- Support priorities for message processing non-intrusively without changing the message processing body
- Selectors have multiple mailboxes, each mailbox is used to store messages of a given priority
- Messages be categorized by priority

NQueens benchmark



A* Search benchmark



Producer-Consumer with Bounded Buffer

- Classic example of a multi-process synchronization problem
- Producers push work into the buffer
- Consumers pull work from the buffer

Actor-based solution

- Buffer actor needs to keep track of
  - Whether the data buffer is full or empty
  - Store consumers when buffer empty buffer
  - Stall producers when buffer full
  - Notify producers when space becomes available in buffer

Selector-based Solution

- Separate mailboxes for producers consumers
- Ideal example for declarative guards
- Producer Mailbox guard
  - Buffer is not full
- Consumer Mailbox guard
  - Buffer is not empty

Selector-based Solution

```
class BufferSelector extends DeclarativeSelector {
 def registerGuards() {
  // disable producer msgs if buffer might overflow
  guard(MBX_PRODUCER,
   (theMsg) => dataBuffer.size() < thresholdSize)
  // disable consumer msgs when buffer empty
  guard(MBX_CONSUMER,
   (theMsg) => !dataBuffer.isEmpty())
 }
 def doProcess(theMsg: AnyRef) {
   theMsg match {
     case dm: ProducerMsg =>
       // store the data in the buffer
       dataBuffer.add(dm)
       // request producer to produce next data
       dm.producer.send(ProduceDataMsg.ONLY)
     case cm: ConsumerMsg =>
       // send data item to consumer
       cm.consumer.send(dataBuffer.poll())
       tryExit()
     case em: ProdExitMsg =>
       numTerminatedProducers += 1
       tryExit()
   } } }
```

Producer Consumer benchmark