Joint Laboratory for Petascale Computation

The Actor Model and Multi-core Architectures

Emilio Francesquini

emilio@ime.usp.br

Alfredo Goldman

gold@ime.usp.br

Jean-François Méhaut

Jean-Francois.Mehaut@imag.fr

University of São Paulo

University of Grenoble

8th Workshop of the INRIA-ANL-Illinois Joint Laboratory on Petascale Computing. Argonne, 11/19/2012

Outline

- The Actor Model
- Erlang
  - Erlang's Actor Model Implementation
  - Application Characteristics
- Ongoing Research

The Actor Model

The Actor Model

- Introduced by Hewitt et al in '73, followed by Agha in '86
  - No shared memory
  - Asynchronous message passing

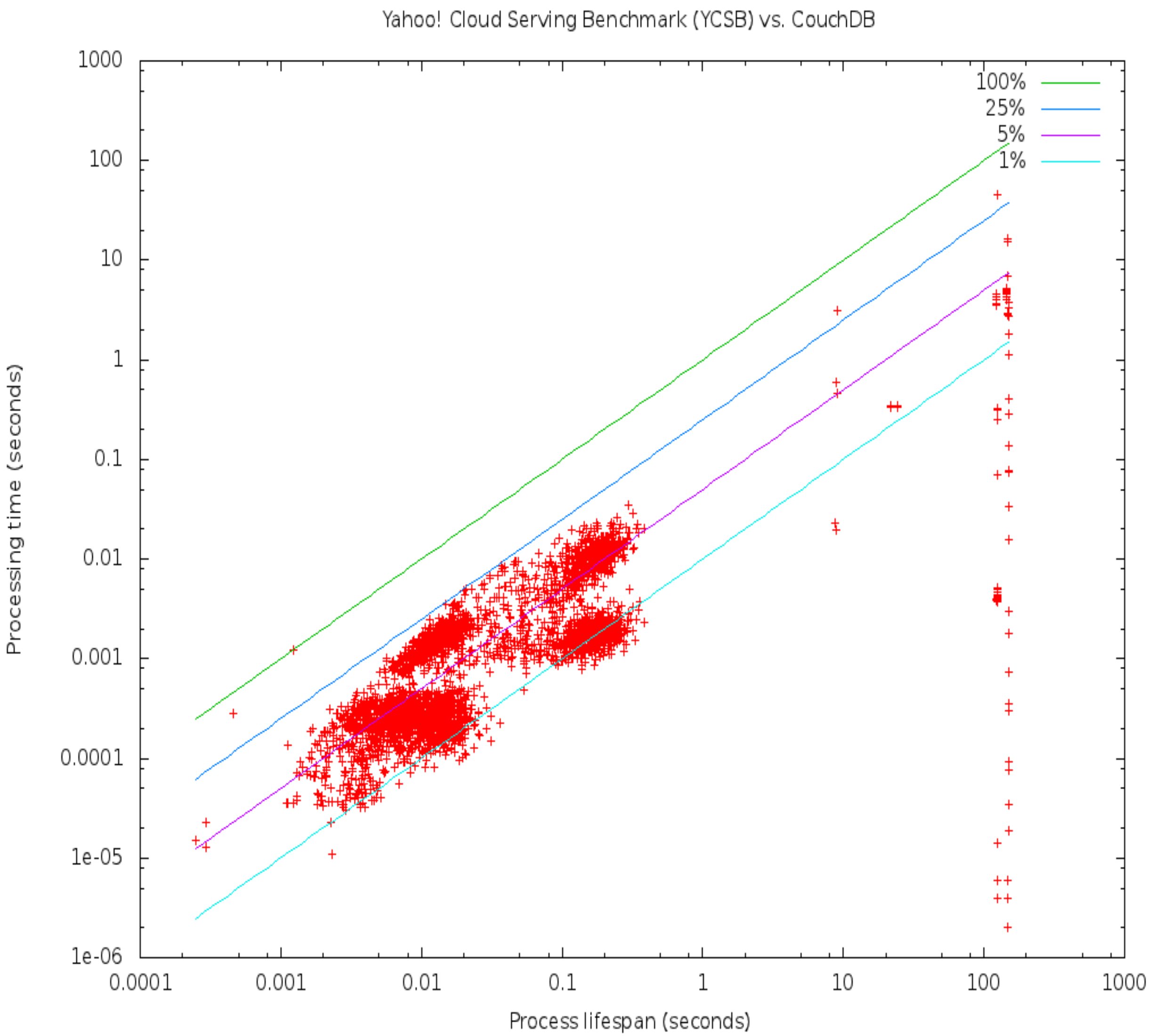
- Strong integration with programming languages, not only as a library but as an integral part of it

- Actors are much like people. An actor
    - is born, spawns and kills other actors, and dies
    - Cannot read (or write) other actor's memories

The Actor Model Fine-Grained Parallelism

- The number of actors is typically much bigger than the number of processors.
  - In a typical simple application, the number of actors, is in the order of dozens or even hundreds
  - Charm++, for example, has a finer grain than MPI. The actor model is even more fine-grained
- This is reasonable since for most of the time they are just waiting for mail, i.e., inactive
- The lifespan of an actor, although variable, is usually short

Actor Model – CouchDB Application Characteristics

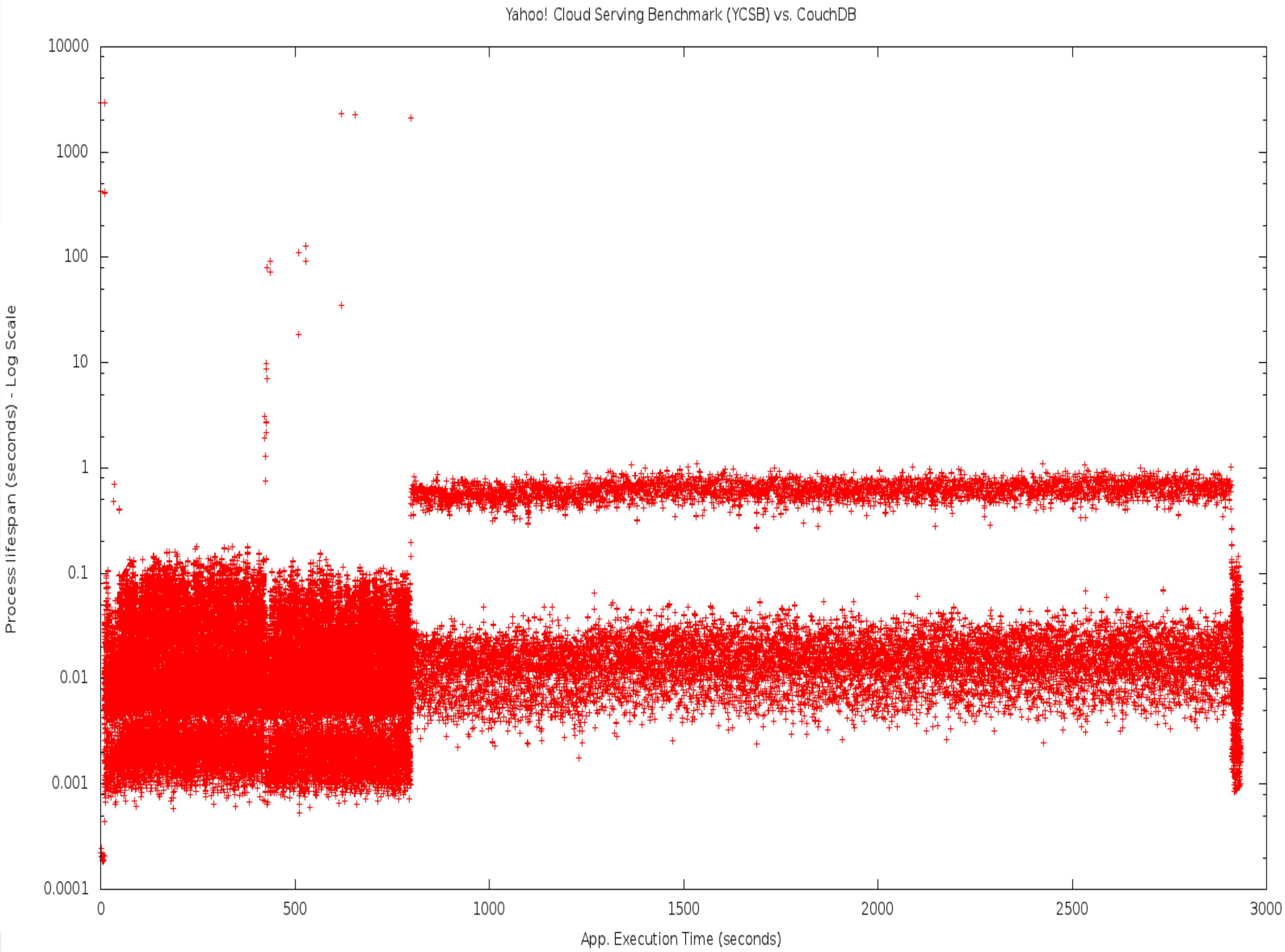


Long-lived actors are mostly inactive

- Job distribution
- Overall application housekeeping

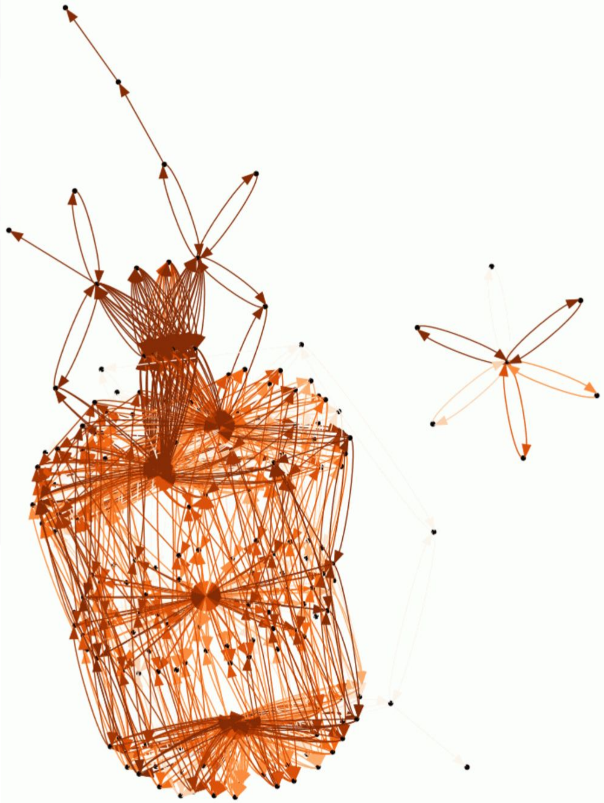
Some worker actors can achieve 100% activity ratio

Actor Model - CouchDB
Actor Lifespan



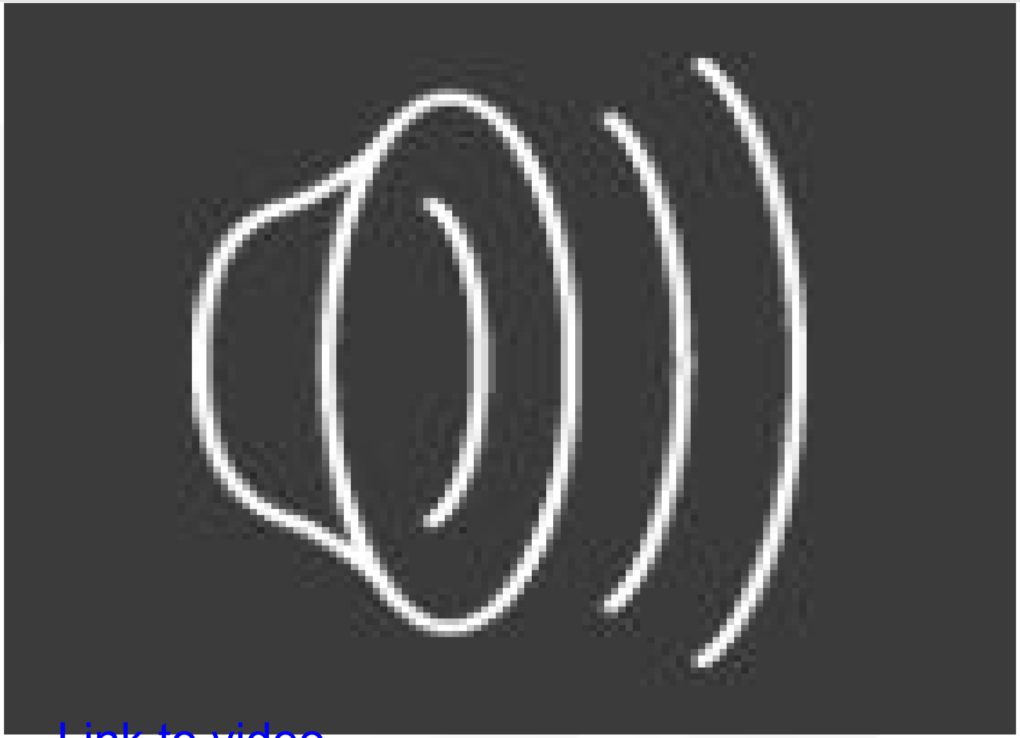
- Short lifespan
  - 99.5% < 1.5s
  - 89.5% < 0.25s
  - 83% > 0.005s
- Very cheap to be created
  - ~1.5µs

Couch DB Communication Graph



Evolution of the actor's communication graph during the execution of the YCSB against CouchDB on a 24-core machine

1/8 of the actual speed



Link to video.

Shows that, although not simple, there is a clearly defined structure of communication throughout the application's lifetime

There are clearly defined hub actors

Communication Graph

- The communication graph is extremely dynamic
  - It depends on the application and on the data
- MapReduce is just one example of several possible sub-graph patterns
- Some actors are clearly hubs, while others are just helpers

The Actor Model – Summing-up

- Scales well in local and distributed systems

- Architecture agnostic programming
  - The runtime environment is responsible for all the necessary adaptations:
    - To efficiently use different machines (memory topologies, caches, processors, OSs, ...)
    - To conform to the needs of each individual application

- Concurrency model used by Erlang, Scala, ...

Erlang

- Functional programming language
- Created in the 80's by Ericsson
- Originally used for Ericsson's telephony routers
  - High Availability Requirements
  - Hot Code Replacement
- Available since 98 as Open Source Software

Erlang – Notable Use Cases

- Famous use cases:
  - Apache's CouchDB
  - WhatsApp (115K Messages/sec.)
  - Amazon's SimpleDB
  - Ericsson's AXD301 160Gbps ATM switch
  - Sim-Diasca, EDF's Discrete Event Simulator

- Concurrent (parallel and distributed) generic synchronous discrete-event simulation engine
- Fully implemented in Erlang
- Actively used by EDF and other partners
  - Used by the Smart Energy Supply Clever Project
    - The idea is to dynamically link energy supply and consumption
- Maintained by EDF R&D
- Released in 2010 by EDF R&D as free software (LGPL)

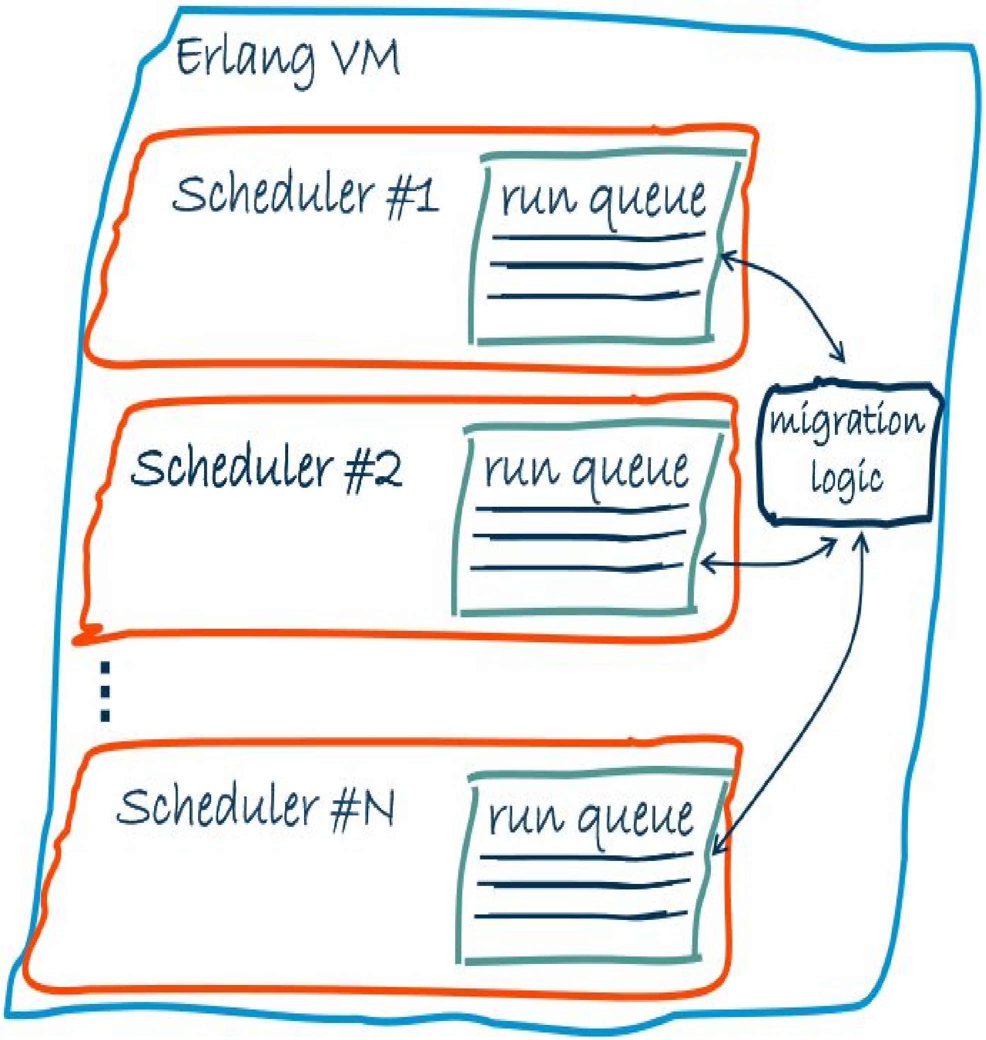
Erlang Virtual Machine

- Runs on top of a custom built virtual machine

Application			
Erlang Virtual Machine			
Machine 1	Machine 2	...	Machine N

- Only 3 concurrency constructs
  - Spawn – Creates a new process
  - ! - Sends a message
  - receive – Receives a message

Erlang Virtual Machine – Scheduler Architecture



- For each PU, one OS thread (called scheduler) is created
- There are specific VM options to determine how these schedulers are distributed throughout the available Pus
- Each scheduler has a run queue of actors ready to be executed

Erlang Virtual Machine – Memory Management

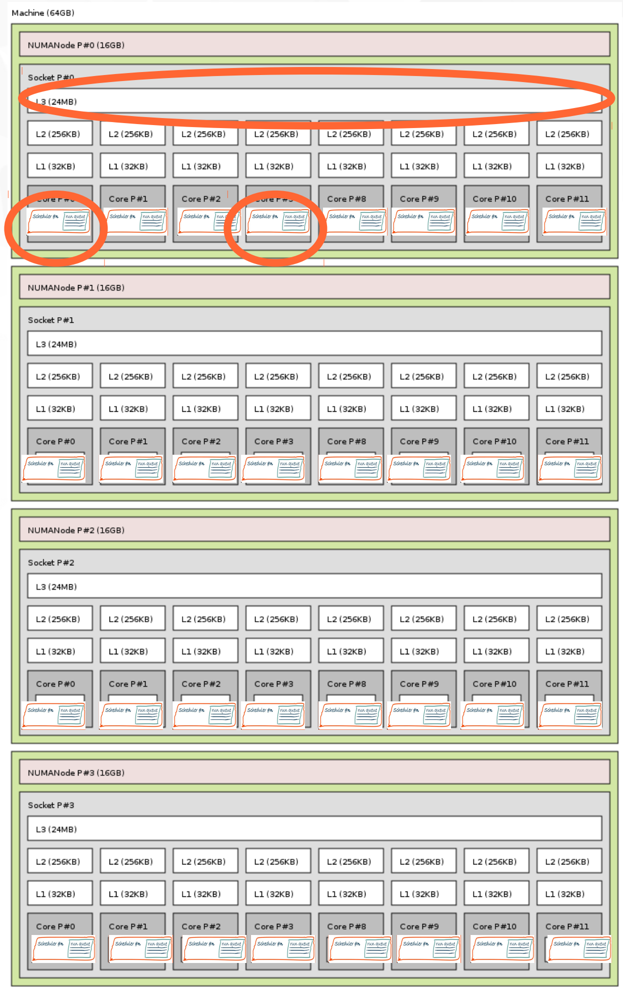
- Each actor in the system has its private heap
  - Small heap
  - No need to “stop the world” to do a garbage collect (at most only the actor being inspected is going to be stopped)
  - As the lifespan of each actor is usually short, most of them never experience garbage collections during their lives

- Message delivery is done by copying the message from the sending actor heap to the receiving actor heap

Erlang Virtual Machine

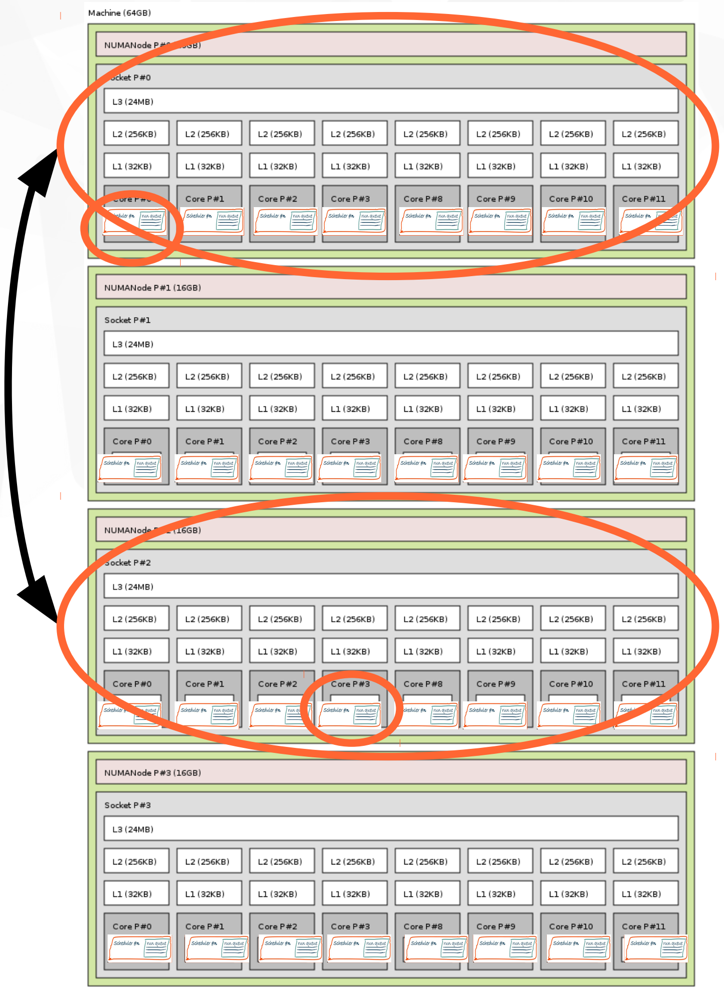
- The current implementation does not take into consideration
  - The impact of the initial placement of each actor
  - Actor's migration
    - Cache
    - Bus contention
    - Memory topology

Erlang Actor Communication – NUMA Machine



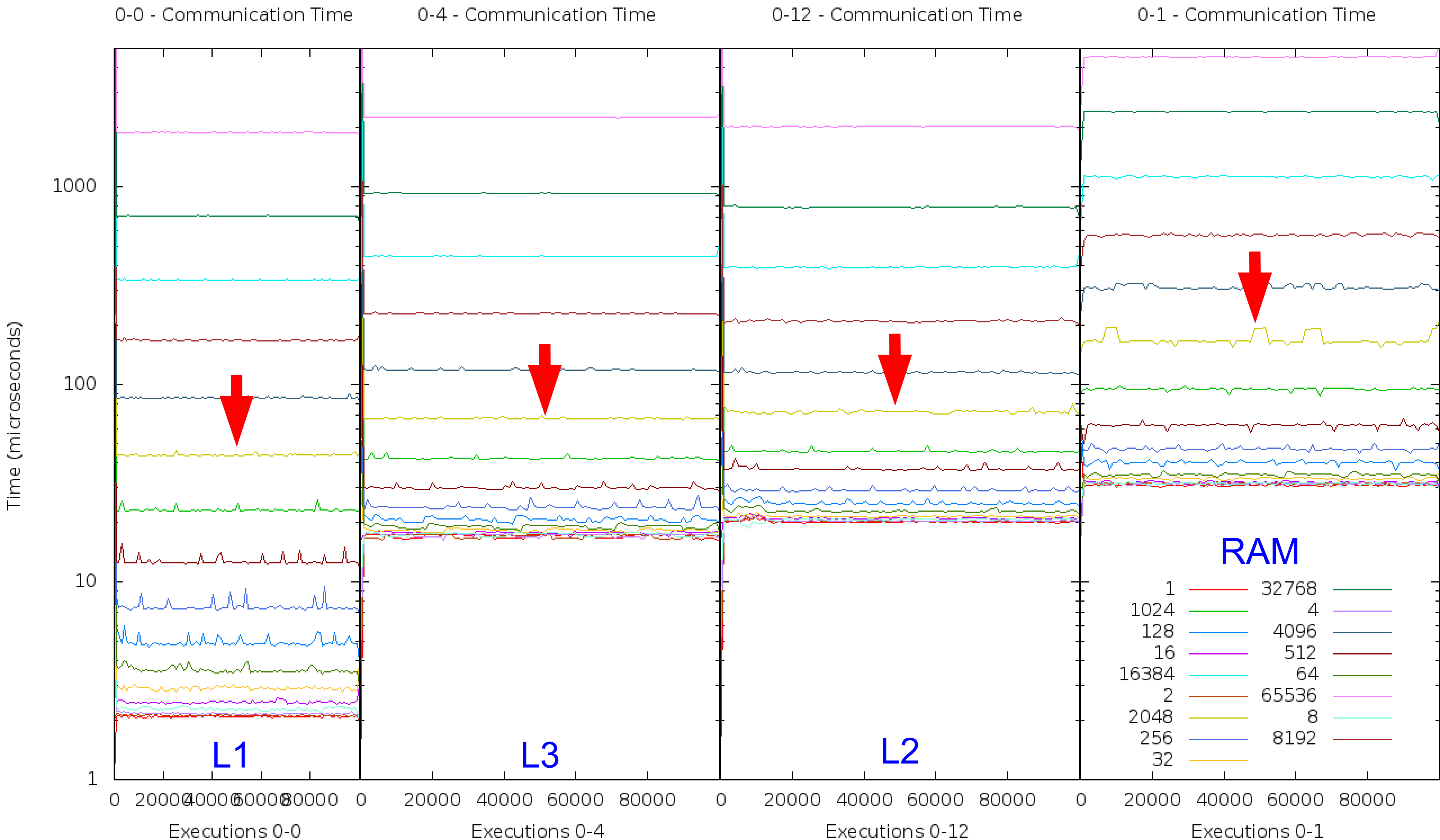
- Message exchange between two actors on the same NUMA node
- If the message is small enough to fit the caches, it can be sent using them, otherwise it ends up using the local node memory

Erlang Actor Communication – NUMA Machine



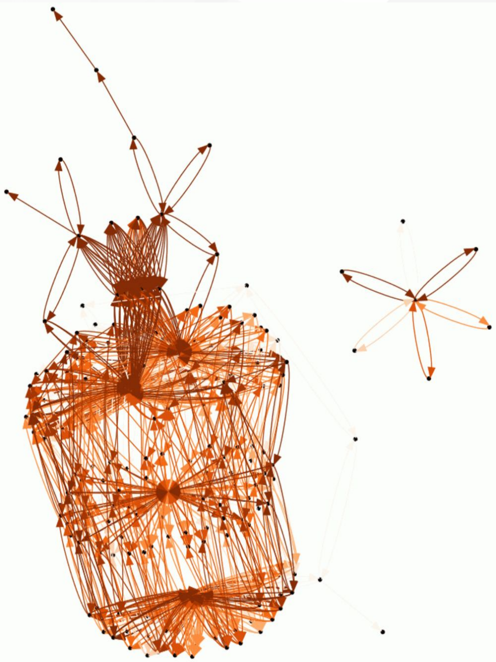
- Message exchange between two actors on distinct NUMA nodes
- Communication through the node's interconnect

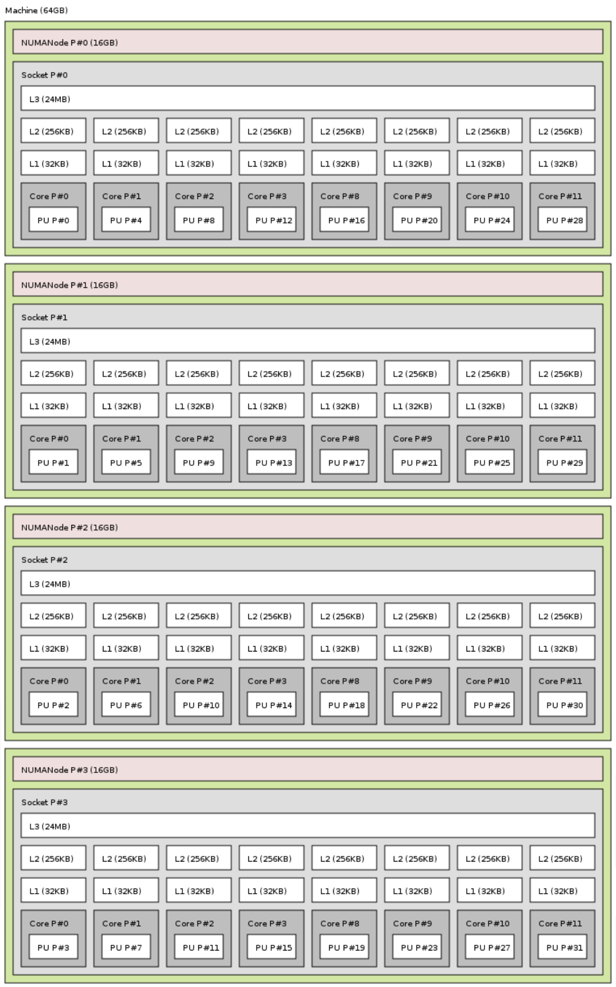
Actor Communication Time - Message sizes from 1 B to 65KB – **idkonn** – 4 sockets – 6 cores per socket. Two cores/L2, Six cores/L3



Ongoing Research

Actor Mapping to Multicore Machines



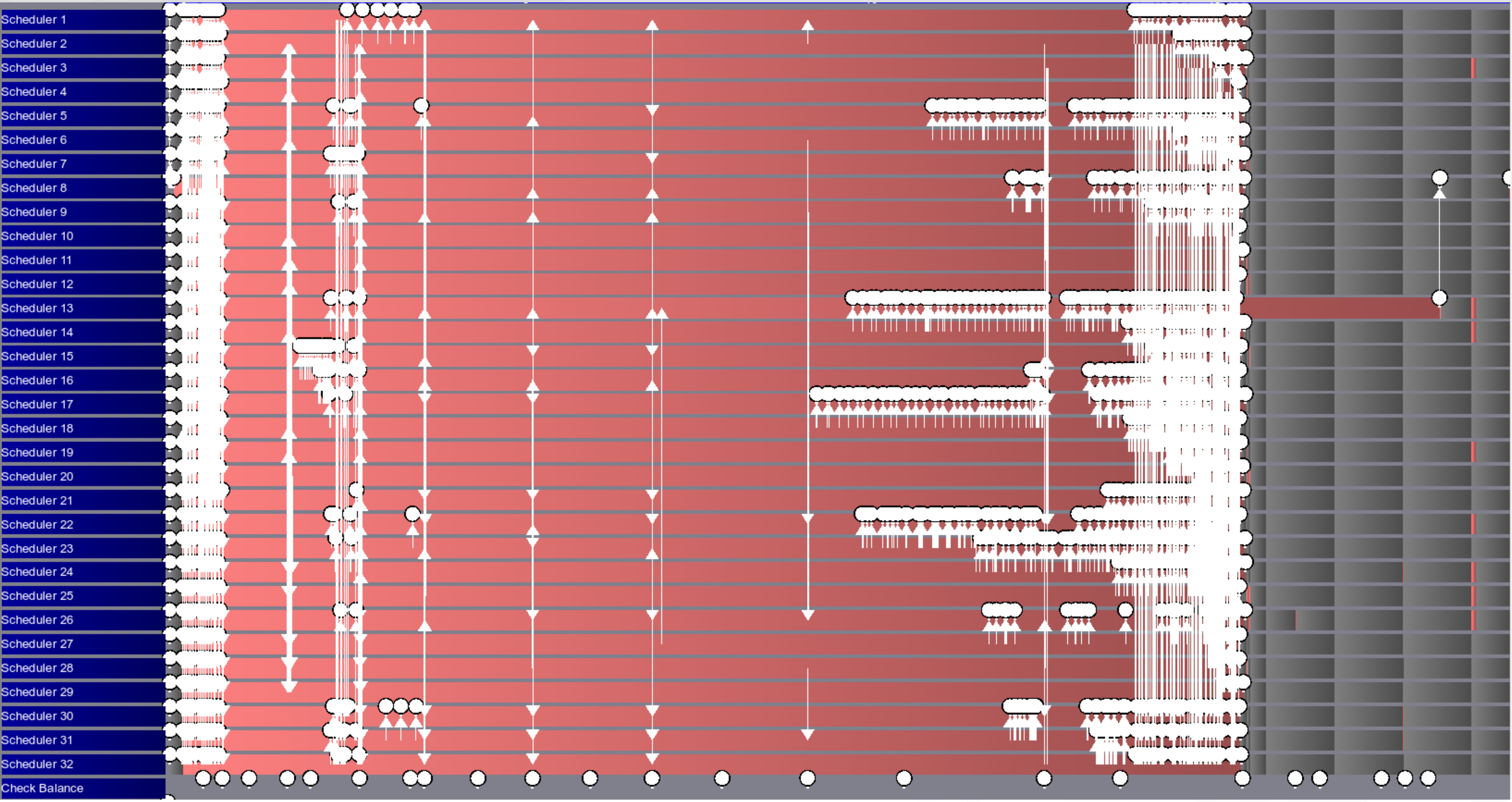


- How to distribute the actors throughout the schedulers, i.e., find a good mapping of the communication graph to the machine
  - To minimize the makespan of the application
  - To take advantage of the hierarchical memory for communication efficiency

Initial Actor Placement

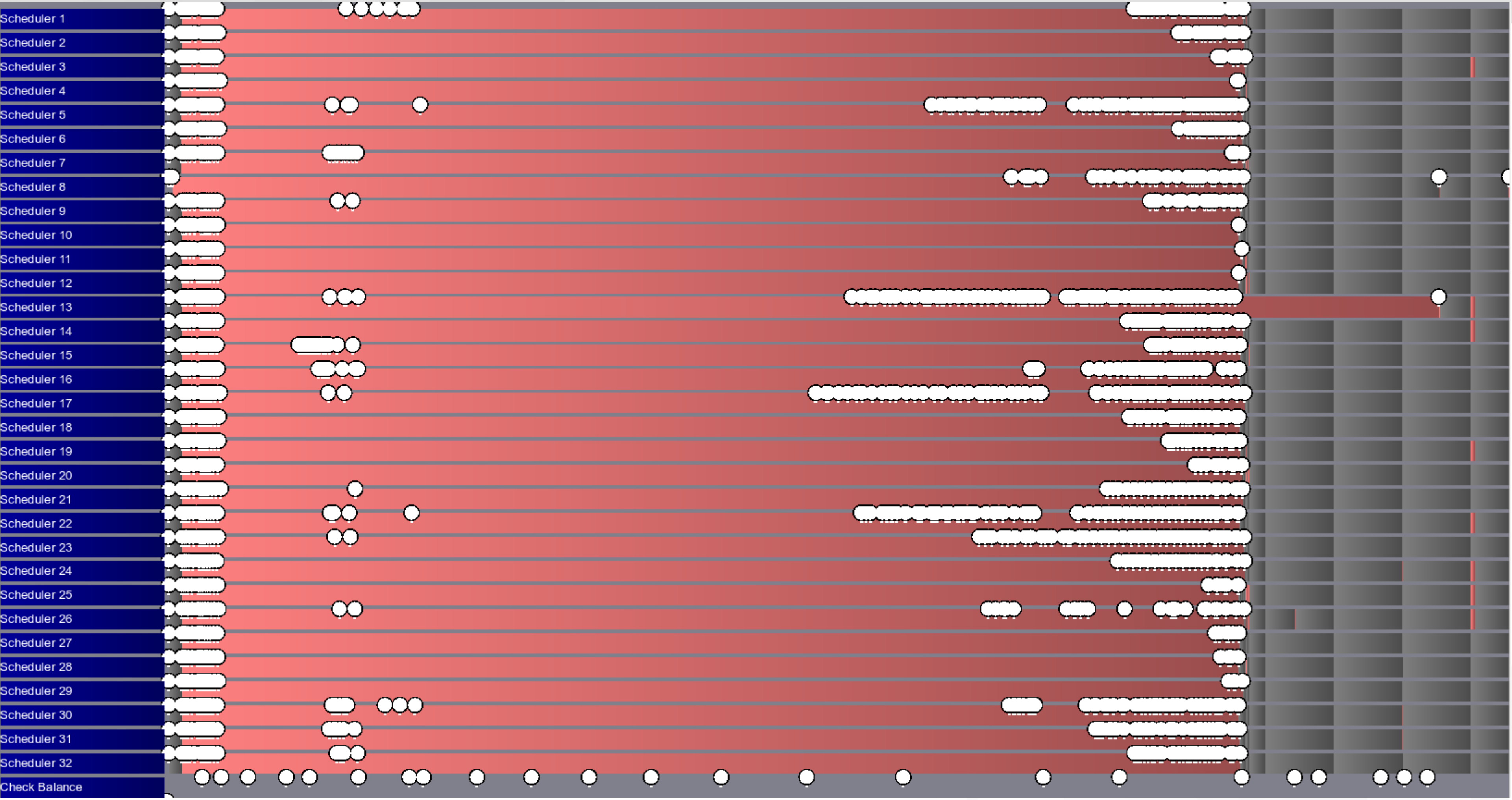
- Most of the actors have a short life
  - initial placement importance
  - We could use strategies like BubbleSched for this since the decision must be done fast
- Is the father of the actor a good indicator of its behavior?
- We do not know the actual behavior of the actor. Can we predict it based on the past?

Initial Placement Impact



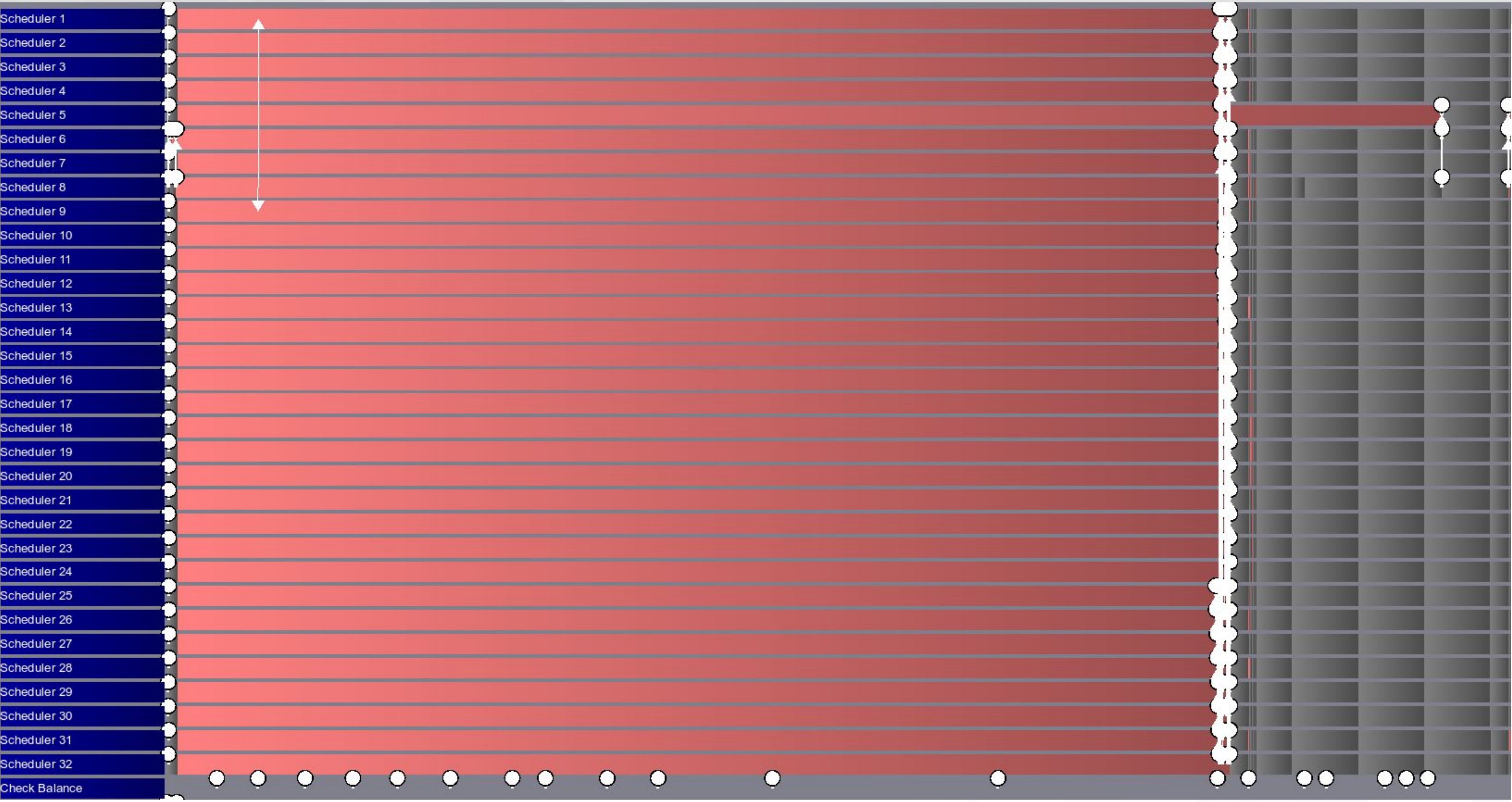
Default Strategy – 3,030 Actors
~4,560 migrations

Initial Placement Impact



Default Strategy – 3,030 Actors
~4,560 migrations

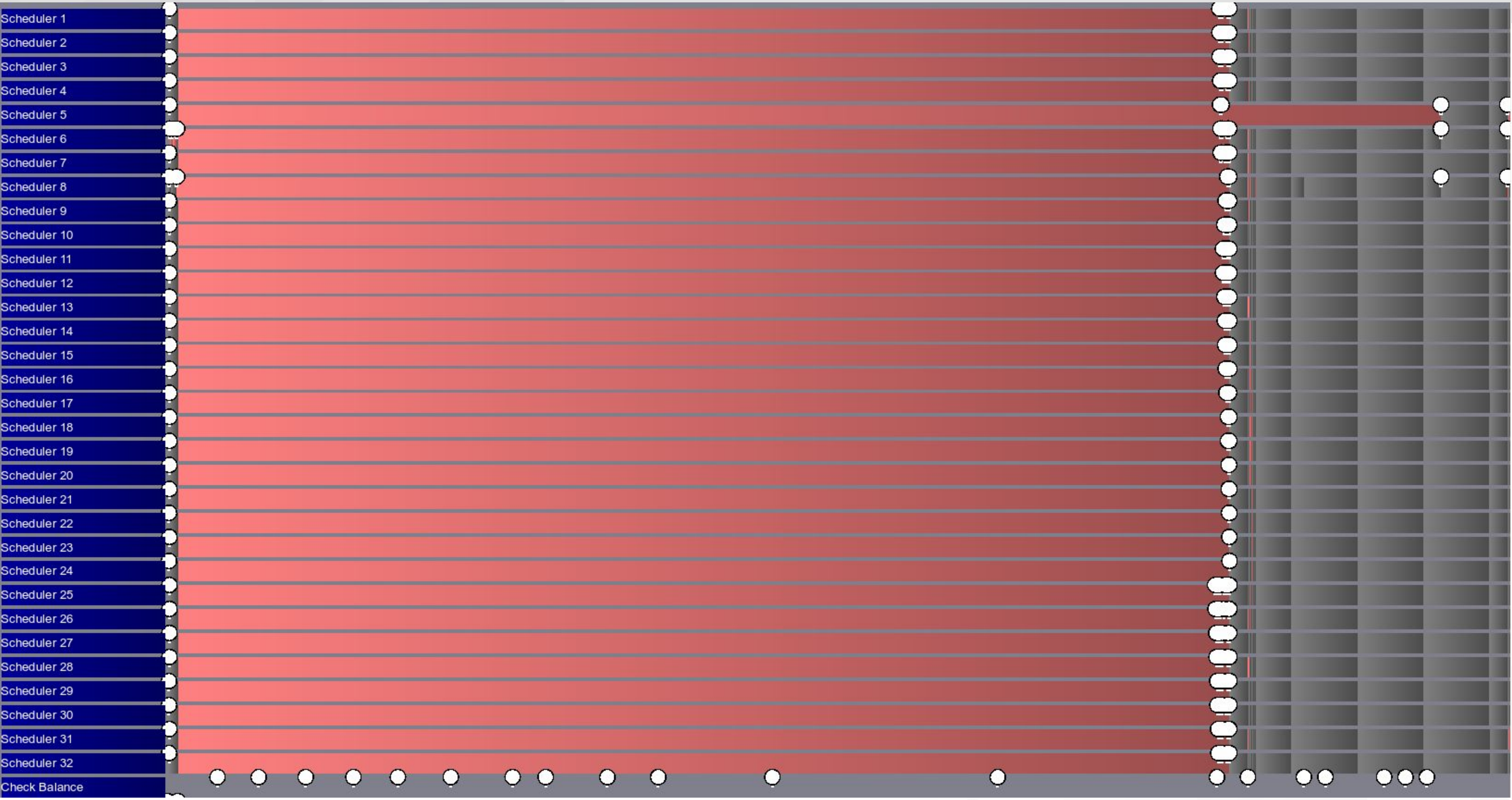
Initial Placement Impact



Round Robin Strategy – 3,030 Actors

~610 migrations

Initial Placement Impact



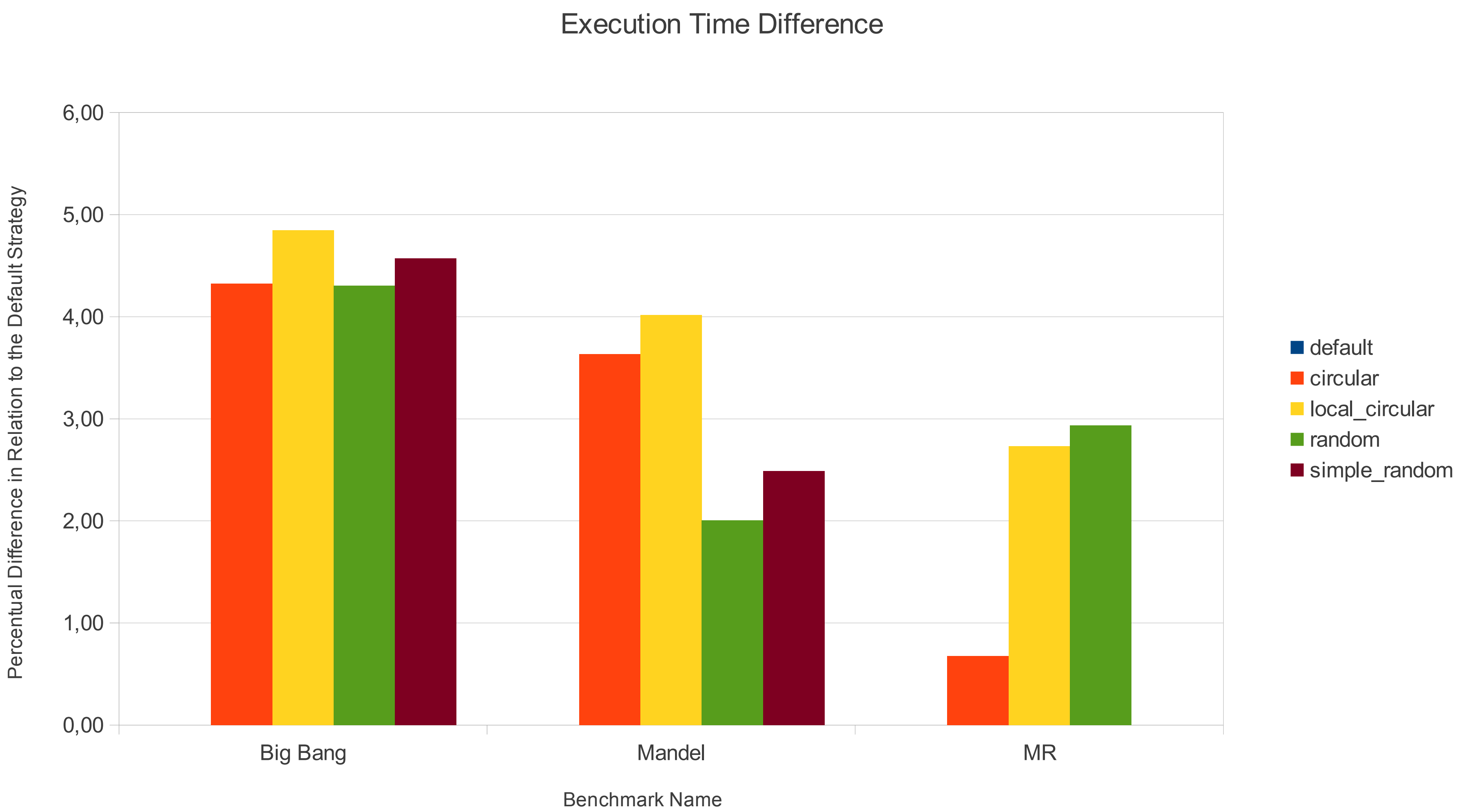
Round Robin Strategy – 3,030 Actors
~610 migrations

Migration Count X Initial Placement Strategy

Strategy	Avg. of 30 Executions
Default	4,560
Round Robin	573
Random	1,244

3,030 Actors
~20 seconds of execution time

Initial Placement and Execution Times



Ongoing Research

- Actor migration
  - Only makes sense for the long lived actors
  - The analysis of the communication graph (highly dynamic) should be taken into consideration
  - The procurement of the communication graph is trivial, doing it fast is not
  - Iterative and continuous process
  - Should be automatic
  - Trying to choose the best placement based on the current and historical data?

Work in progress

- Initial Actor Placement
  - Actors in general have a short life, therefore the initial placement decision must be fast
  - Idea: Implement something similar to BubbleSched

- Actor Migration
  - Establish migration paths for each actor to minimize the makespan based on the communication graph and the machine topology
  - Idea: Hub actors, if placed together, might saturate the bus. Place the hub nodes as far as possible to increase performance

- Actor Pinning
  - Automatic for hub actors, and application developers might have insights as to the best placement of each actor
  - Idea: Implement an interface that allows the application developer to specify where the actors should be placed

Thank you!