A String of Ponies
Transparent Distributed Programming with Actors

Sebastian Blessing

Supervised by Prof. Sophia Drossopoulou and Sylvan Clebsch

M.Sc in Computing Science (Software Engineering), 13. September 2013
(Concurrent) Pony

• Object-oriented
• Actor-based with *causal messaging*
  – Zero-copy message passing semantics
• Fully concurrent garbage collection of actors and objects
  – No “stop-the-world” step necessary
• One kernel thread per core
• Work stealing scheduler
• Termination based on *quiescence*
Extend Pony with *transparent* Distribution

- Distribution through Actor *mobility*

- Requirements and Contributions
  - Networking Layer (Async. I/O multiplexing)

1. Causal Messaging in Distributed Systems
2. Join nodes to a cluster of *Ponies*
3. Distributed Work Stealing Scheduler
   a. Proxies and causality
4. Serialisation and Deserialisation
5. Distributed Garbage Collection of Actors and Objects
6. Distributed Termination
Architectural Overview
1. Causal Messaging in Distributed Systems

Causality: m1 -> m3 at C
3. Distributed Work Stealing

• **Challenges:**
  – Detect resources that or not used to capacity
  – Migrate Actors to nodes in a network

• **Problem:** busy nodes should not cause their children to suffer from starvation

• Leverage tree topology
3. Distributed Work Stealing

Dashed UP arrows: report free core counts (transitively)

Dashed DOWN arrows: report free cores reachable via parent

Total core count: 132
3. Distributed Work Stealing
3a. Migrated Actors become *Proxies*

Node A:
- A1
  - (1) m1 to A2
  - (2) m2 to B1
- Da
  - (3) m3 to A2

Node B:
- Db
  - (1) m1 from A1
  - (3) m3 from B1

m1 -> m3 guaranteed
3a. Problem: Migration can break causality
3a. Answer: Causality-aware Migration Protocol

• Force A2 to have processed m1 *before* migration

• Introduce *conf-ack* cycle
  – Stop delegating messages once proxy received conf. message
  – Migrate when remote actor receives ack. message
5. Garbage Collection: Cycle Detection

• (Concurrent) Pony employs a cycle detector (implemented as actor)

• A *blocked* actor sends its ID, its ref-count and the set of actors it references to the cycle detector
• If an actor unblocks, it sends UNBLOCK(ID) to the cycle detector
• Cycle detector determines *perceived* actor state

• Problem: *Asynchronicity*, actor states may be out of sync.
5. True Cycle Detection Protocol

- Cycle detector sends CONF(t) to all actors in a perceived cycle
- Actors respond with ACK(t)

- *Causality* helps:
  - If the cycle detector received ACK(t) from all actors in t without having received UNBLOCK(a) after CONF(t) from some actor, the perceived cycle must be a true cycle and can be collected
5. Distributed Cycle Detection
5. Distributed Cycle Detection
5. Distributed Cycle Detection
5. Distributed Cycle Detection
6. Distributed Termination

- Re-use conf-ack protocol from cycle detection
- Employ central termination detector on the master node

- Nodes starved from work report “idle“
- Once work is available report “busy”

- The termination detector having received “idle” from all nodes (including the master) sends CONF(t) to all nodes

- Nodes acknowledge the termination attempt \( t \).

- If all nodes have acknowledged termination without a single node having sent “idle”, the detector sends a termination message to its children (which delegate the message). Otherwise, the termination attempt is cancelled.
Evaluation
Pony vs. Erlang
Micro Benchmark

Diagram:
- **Main Actor**
  - Create
  - **Master**
    - Prime factorisation
    - Spawn ring
  - C-many
  - **Master**
    - Prime factorisation
    - Spawn ring
- Token is sent p-times
Computation-Boundness

Diagram:
- Main Actor
- Create
- Prime factorisation
- Spawn ring
- C-many
- Token is sent p-times
Concurrent Setting

The graph shows the execution time (in seconds) for different factorisation methods under various core configurations. The methods include:

- Dist. Erlang - 4 cores
- Dist. Erlang - 8 cores (HT)
- Pony - 4 cores
- Pony - 8 cores (HT)

The x-axis represents the number of factorisation actors, while the y-axis represents the execution time. The lines indicate the performance improvement as the number of actors increases.
2 nodes

![Graph showing execution time for different factorisation actors across 2 nodes. The graph compares Dist. Erlang with 8 and 16 (HT) cores, and Pony with 8 and 16 (HT) cores.](image)
3 nodes

Execution Time (seconds)

- Dist. Erlang - 12 cores
- Dist. Erlang - 24 (HT) cores
- Pony - 12 cores
- Pony - 24 (HT) cores

#Factorisation Actors
4 nodes

Execution Time (seconds) vs. #Factorisation Actors

- Dist. Erlang - 16 cores
- Dist. Erlang - 32 (HT) cores
- Pony - 16 cores
- Pony - 32 (HT) cores
5 nodes

![Graph showing execution time for different factorisation actors with 5 nodes.](image-url)
6 nodes
Trend
## 1000 Factorisation Actors

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Dist. Erlang</th>
<th>Dist. Erlang (HT)</th>
<th>Pony</th>
<th>Pony (HT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>473.85s</td>
<td>380.55s</td>
<td>393.39s</td>
<td>244.35s</td>
</tr>
<tr>
<td>2</td>
<td>241.45s</td>
<td>205.43s</td>
<td>321.90s</td>
<td>184.98s</td>
</tr>
<tr>
<td>3</td>
<td>165.45s</td>
<td>142.70s</td>
<td>276.37s</td>
<td>157.22s</td>
</tr>
<tr>
<td>4</td>
<td>126.11s</td>
<td>115.62s</td>
<td>239.23s</td>
<td>143.14s</td>
</tr>
<tr>
<td>5</td>
<td>101.91s</td>
<td>87.53s</td>
<td>223.22s</td>
<td>141.13s</td>
</tr>
<tr>
<td>6</td>
<td>86.76s</td>
<td>75.07s</td>
<td>225.25s</td>
<td>142.16s</td>
</tr>
</tbody>
</table>

1 node = 1x Intel i7 – Quad Core @ 3.40 GHz HT, 8 GB Memory
Future Work

• Improve I/O Multiplexing
• Failure Detection and Dynamic Tree Topology
• Locality-aware Tree Topology
• Tune Distributed Work Stealing and Migration
• Compiler, Type System, Formal Models
• Generalisation

• Publications
Backup Slides
Joining new Slave Nodes

• Adding new nodes at runtime
• Configure Tree Topology

• Challenges:
  – No need for maintaining global view at each node
  – Nodes should not be required to store any information in order to determine the path via which a new slave should be delegated
  – Guarantee that the tree is *almost* balanced at any point in time

• Answer: Find a location of a new node based on its ordinal number
Joining new Slave Nodes
Joining Algorithm

\[ All(d) = \sum_{i=1, i \in \mathbb{N}}^{d} k^i \]

\[ Ord : \mathbb{N}^* \rightarrow \mathbb{N} \]
\[ Ord(a_1...a_d) = All(d - 1) + \left( ((a_1 \cdot k + a_2) \cdot k + a_3) \right) \cdot k + ... + a_{d-1} \right) \cdot k + a_d + 1 \]

\[ Ord^{-1} : \mathbb{N}^* \rightarrow \mathbb{N} \]
\[ \forall m \in \mathbb{N}^+: Ord^{-1}(m) = a_1...a_n \text{ such that } \forall i \in \mathbb{N}^+: 0 < a_i \leq k \land Ord(a_1...a_n) = m \]

\[ b_p = m - All(p - 1) - 1 \]
for i in range(p, 1):
\[ a_i := b_i \mod k \]
\[ b_{i-1} := \frac{(b_i - a_i)}{k} \]
Handling Actor References

Node A

A1
ref: a1

A2
sends a1 to B1

B1p

Da

m: {type_A1, glob_ID_A1}

Node B

Db
deser. m and creates proxy for A1

B1

m: {A1p}

A1p
Object Identity Comparison
Distributed Reference Counting (Actors)

Node A
- A1
- A2
  - ref: a1
  - sends a1 to B1
- Da
- B1p

Node B
- B1
- Db
  - m: {type_A1, glob_ID_A1}
  - deser. m and creates proxy for A1
- A1p
Distributed Reference Counting (Objects)

Node A

A1

Determine owner, delegate DEC message

{ref_obj_C}

B2p

delegate

Determine owner, delegate DEC message

{glob_ID_C, DEC(count)}

Node B

B2

local ref owner of Cb is Db

{ref_Cb}

If Cb unreachable by B2 send DEC to Db

Heap Db
Cb: {...}

allocate object at address Cb