Actors to Threads Mapping Technique for JVM-based Actor Frameworks

Ganesha Upadhyaya and Hridesh Rajan

{ganeshau,hridesh}@iastate.edu

Iowa State University

This work was supported in part by the NSF grants CCF-08-46059, CCF-11-17937, and CCF-14-23370.
Problem

Actor System

actor1  actor2

actor3

actor4  actor5

Architecture

<table>
<thead>
<tr>
<th>core0</th>
<th>core1</th>
</tr>
</thead>
<tbody>
<tr>
<td>core2</td>
<td>core3</td>
</tr>
</tbody>
</table>
Problem

Actor System

actor1 → actor2 → actor3 → actor4 → actor5

Mapping

OS Scheduler

JVM threads

Architecture

<table>
<thead>
<tr>
<th>core0</th>
<th>core1</th>
</tr>
</thead>
<tbody>
<tr>
<td>core2</td>
<td>core3</td>
</tr>
</tbody>
</table>
**Problem**

**PROBLEM:** Mapping Actors to JVM threads
Problem

PROBLEM: Mapping Actors to JVM threads

- More Actors than JVM threads,
- Actors are short-lived [1],
- OS Scheduler maps JVM threads to Cores,

PROBLEM: Mapping Actors to JVM threads

- More Actors than JVM threads,
- Actors are short-lived [1],
- OS Scheduler maps JVM threads to Cores,

INTUITION: Actor characteristics and Communication behaviours could be used to decide the mapping

PROBLEM: Mapping Actors to JVM threads

- More Actors than JVM threads,
- Actors are short-lived [1],
- OS Scheduler maps JVM threads to Cores,

INTUITION: Actor characteristics and Communication behaviours could be used to decide the mapping

SOLUTION: Initial mapping of Actors to JVM threads

Discuss JVM based actor frameworks
Motivating examples
Solution
Illustrative example
Evaluation & Results
Limitations and Future Work.
JVM-based actor frameworks

Akka
- default
- pinned
- balancing
- calling-thread

Scala Actors, Actors Guild
- thread-based
- event-based

Kilim, Actor Foundry
- light-weight event-based actors,
  scheduler is a bundle composed of a thread-pool, scheduling policy, collection of runnable actors,
  scheduled in round-robin fashion

SALSA
- heavy-weight (individual stage)
- light-weight (stage-sharing)
- each stage (actor) is a bundle of a msgQ and JVM thread

➢ start with default mappings,
➢ iteratively refine the mappings to achieve desired performance
Example 1: Master-Worker (RayTracer)
easy to map actors to JVM threads,
because actors perform independent computations,
less interactions,
data-parallel.

Example 1: Master-Worker (RayTracer)
Example 2: BenchErl-Serialmsg
Example 2: BenchErl-SerialMsg

- $m$ instances of `Generator` actor,
- $m$ instances of `Receiver` actor,
- one `Dispatcher` actor,
Example 2: BenchErl-Serialmsg

- $m$ instances of `Generator` actor,
- $m$ instances of `Receiver` actor,
- one `Dispatcher` actor,

**Intuitive Mapping Process**

- start with a task-pool (size=#cores) & put all actors in there,
- looks like Dispatcher is a bottleneck,
- assign a thread-pool to Dispatcher,
- still not working, load-imbalance, how do I do it?
m instances of `Generator` actor,
m instances of `Receiver` actor,
one `Dispatcher` actor,

Intuitive Mapping Process

- start with a task-pool (size=#cores) & put all actors in there,
- looks like Dispatcher is a bottleneck,
- assign a thread-pool to Dispatcher,
- still not working, load-imbalance, how do I do it?

Insight

- each generator communicates with receiver often through dispatcher,
- whole communication (g0 -> d -> r0) could be made uninterrupted

Some aspects of actor applications can help to decide actors to JVM threads mapping
Some aspects of actor applications can help to decide actors to JVM threads mapping

- **blocking**, 
  - externally blocking behaviors using I/O, socket or database blocking primitives,
Some aspects of actor applications can help to decide actors to JVM threads mapping

- **blocking**,  
  - externally blocking behaviors using I/O, socket or database blocking primitives,

- **inherent parallelism**,  
  - actors may use blocking send primitives and receive results or use asynchronous send primitives. Actors may or may not require the results immediately,
Some aspects of actor applications can help to decide actors to JVM threads mapping

- **blocking**,  
  - externally blocking behaviors using I/O, socket or database blocking primitives,

- **inherent parallelism**,  
  - actors may use blocking send primitives and receive results or use asynchronous send primitives. Actors may or may not require the results immediately,

- **communication behavior**,  
  - leaf actor,
  - routing actor,
  - broadcast actor.

- **computations**,
## Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>
Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
</table>

- **BLK** = \{true, false\} represents blocking behavior,
### Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
</table>

- **BLK** = \{true, false\} represents blocking behavior,
- **STATE** = \{true, false\} represents stateful/stateless behavior,
Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
</table>

- **BLK** = \{true, false\} represents blocking behavior,
- **STATE** = \{true, false\} represents stateful/stateless behavior,
- **PAR** = \{low, med, high\} represents inherent parallelism,
  - low, if actor sends synchronous message and waits for the result, or consumes the result right-away,
  - high, if actor sends asynchronous message and does not require result,
  - med, otherwise.
### Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
</table>

- **BLK** = \{true, false\} represents blocking behavior,
- **STATE** = \{true, false\} represents stateful/stateless behavior,
- **PAR** = \{low, med, high\} represents inherent parallelism,
  - low, if actor sends synchronous message and waits for the result, or consumes the result right-away,
  - high, if actor sends asynchronous message and does not require result,
  - med, otherwise.
- **COMM** = \{low, med, high\} represents communication behavior,
  - low, does not send messages to other actors (leaf actor),
  - med, sends exactly one message for every message received (router actor),
  - high, sends more than one message (broadcast actor).
Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
</table>

- **BLK** = \{true, false\} represents blocking behavior,
- **STATE** = \{true, false\} represents stateful/stateless behavior,
- **PAR** = \{low, med, high\} represents inherent parallelism,
  - low, if actor sends synchronous message and waits for the result, or consumes the result right-away,
  - high, if actor sends asynchronous message and does not require result,
  - med, otherwise.
- **COMM** = \{low, med, high\} represents communication behavior,
  - low, does not send messages to other actors (leaf actor),
  - med, sends exactly one message for every message received (router actor),
  - high, sends more than one message (broadcast actor).
- **CPU** = \{low, high\} represents computational workload of the actor,
  - high, when recursive, loops with unknown bounds, makes high cost library calls,
  - low, otherwise.
Solution

➢ For mapping actors to threads,
  ○ we assign execution policy to actors,
➢ execution policy,
  ○ defines, how actor’s messages are processed?
Execution Policies

➢ **THREAD**, actor is assigned a dedicated thread,
➢ **TASK**, actor is assigned to a task-pool and the shared thread of the task-pool will process the messages,
➢ **SEQ/MONITOR**, calling actor thread itself

**TH**: Thread  **TA**: Task  **SEQ**: Sequential  **M**: Monitor
Actor Communication Graph (ACG) is a directed graph $G(V,E)$ where,

- $V = A_0, A_1, \ldots, A_n$ is a set of nodes, each node represents an actor,
- $E$ is a set of edges $(A_i, A_j)$ for all $i,j$ such that there is a communication from $A_i$ to $A_j$.

Mapping function $M(A_i \times P \times ACG) \rightarrow EP$ where,

- $A_i$ is actor definition,
- $P$ is the actor program,
- $ACG$ is the actor communication graph,
- $EP = \{ \text{THREAD} | \text{TASK} | \text{SEQ} | \text{MONITOR} \}$
Figure: Flow diagram of our mapping function that assigns actors one of the four execution policies.
### Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Actors</td>
<td>true</td>
<td>⚫⚫⚫⚫</td>
<td>⚫⚫⚫⚫</td>
<td>⚫⚫⚫⚫</td>
<td>⚫⚫⚫⚫</td>
</tr>
</tbody>
</table>

**Policy**: THREAD

* 🗝️ means not used to make decision
## Actor Characteristic Vector (cVector)

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>BLK</th>
<th>STATE</th>
<th>PAR</th>
<th>COMM</th>
<th>CPU</th>
<th>Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blocking Actors</strong></td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>THREAD</td>
</tr>
<tr>
<td><strong>Heavy Actors</strong></td>
<td>false</td>
<td></td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>THREAD</td>
</tr>
<tr>
<td><strong>HighCPU Actors</strong></td>
<td>false</td>
<td></td>
<td>high</td>
<td>low</td>
<td>high</td>
<td>TASK</td>
</tr>
<tr>
<td><strong>LowCPU Actors</strong></td>
<td>false</td>
<td></td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>MONITOR</td>
</tr>
<tr>
<td><strong>Hub Actors</strong></td>
<td>false</td>
<td></td>
<td>high</td>
<td>high</td>
<td>low</td>
<td>TASK</td>
</tr>
<tr>
<td><strong>Affinity Actors</strong></td>
<td>false</td>
<td></td>
<td>low/med</td>
<td>low/med</td>
<td>low</td>
<td>MONITOR</td>
</tr>
<tr>
<td><strong>Master Actors</strong></td>
<td>false</td>
<td></td>
<td>low/med</td>
<td>high</td>
<td>low</td>
<td>THREAD</td>
</tr>
<tr>
<td><strong>Worker Actors</strong></td>
<td>false</td>
<td></td>
<td>low/med</td>
<td>low/med</td>
<td>high</td>
<td>TASK</td>
</tr>
</tbody>
</table>

* means not used to make decision
An Example: FileSearch

```
<table>
<thead>
<tr>
<th>Capsule</th>
<th>cVector</th>
<th>Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileCrawler</td>
<td>&lt;false, __, high, high, high&gt;</td>
<td>Thread</td>
</tr>
<tr>
<td>FileScanner</td>
<td>&lt;false, __, high, high, low&gt;</td>
<td>Task</td>
</tr>
<tr>
<td>Indexer</td>
<td>&lt;false, __, low, low, low&gt;</td>
<td>Monitor</td>
</tr>
<tr>
<td>Searcher</td>
<td>&lt;true, __, __, __, __&gt;</td>
<td>Thread</td>
</tr>
</tbody>
</table>
```
Evaluation

- Benchmark programs (14 total)
  - that exhibits data, task, and pipeline parallelism at coarse and fine granularities.
- Comparing against default-thread and default-task,
- Measured reduction in program runtime over default mappings on different core settings.
Experimental Results

On average 50% improvement over default-thread and default-task mappings

Figure: Results show Ith (improvement over default-thread mapping) and Ita (improvement over default-task mapping) for the benchmarks.
Experimental Results

Small or no improvement for data parallel actor programs

Figure: Results show Ith (improvement over default-thread mapping) and Ita (improvement over default-task mapping) for the benchmarks.
Large improvements for actor programs with sub-optimal performance benefits.

Figure: Results show Ith (improvement over default-thread mapping) and Ita (improvement over default-task mapping) for the benchmarks.
In BenchErl/mbrot

- **WHERE:**
  - each Worker communicates with a ‘Mandel’ actor that checks if a pixel belongs to the Mandelbrot set or not,

- **PROBLEM:**
  - inefficient decision,
  - each Worker can perform this test independently,
  - introducing shared Mandel kills the parallel performance.

- **FIX:**
  - Mandel is assigned MONITOR execution policy,
  - each Worker now executes the Mandel actor’s code.

Can we reduce the performance penalties due to inefficient design of actor system?

**Future Work!!!**
application of our technique to wide-variety of JVM-based actor frameworks, Call for collaborations!
application of our technique to wide-variety of JVM-based actor frameworks, Call for collaborations!

- dynamism in actor-model
  - dynamic actor creation
    - execution policy for the actor type is still assigned!
  - dynamism in actor communication graph
    - our technique does not rely heavily on ACG, however availability of partial/full ACG helps to improve the mapping further!
    - also, programmers can use execution traces to gather ACG.
Future Work

- load-imbalance,
  - assigning execution policy that enables load-balancing.
Future Work

➢ load-imbalance,
  ○ assigning execution policy that enables load-balancing.

➢ contentions (bencherl/serialmsg)
  ○ solution: contention-aware assignment of execution policy.

<table>
<thead>
<tr>
<th>#cores</th>
<th>Th</th>
<th>Ta</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>215097/36875</td>
<td>80912/23586</td>
<td>185018/3267</td>
</tr>
<tr>
<td>4</td>
<td>250674/21657</td>
<td>177178/25536</td>
<td>208674/1318</td>
</tr>
<tr>
<td>8</td>
<td>268856/5555</td>
<td>228561/9306</td>
<td>222940/1437</td>
</tr>
<tr>
<td>12</td>
<td>273426/5353</td>
<td>237860/25855</td>
<td>224680/1757</td>
</tr>
</tbody>
</table>
Future Work

➢ load-imbalance,
   ○ assigning execution policy that enables load-balancing.

➢ contentions (bencherl/serialmsg)

<table>
<thead>
<tr>
<th>#cores</th>
<th>Th</th>
<th>Ta</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>215097/36875</td>
<td>80912/23586</td>
<td>185018/3267</td>
</tr>
<tr>
<td>4</td>
<td>250674/21657</td>
<td>177178/25536</td>
<td>208674/1318</td>
</tr>
<tr>
<td>8</td>
<td>268856/5555</td>
<td>228561/9306</td>
<td>222940/1437</td>
</tr>
<tr>
<td>12</td>
<td>273426/5353</td>
<td>237860/25855</td>
<td>224680/1757</td>
</tr>
</tbody>
</table>

   ○ solution: contention-aware assignment of execution policy.

➢ cache-miss (FileSearch)
   ○ about 10% LLC-load-misses
   ○ solution: cache-aware assignment of execution policy.
Conclusion

Ganesha Upadhyaya

ganeshau@iastate.edu

Questions?

This work was supported in part by the NSF grants CCF- 08-46059, CCF-11-17937, and CCF-14-23370.