Modeling Schedulers in Parallel and Distributed Systems

Serbanescu Vlad Nicolae
The ABS language

• ABS is designed with analysis/code generation tools in mind.

• State-of-art programming language concepts
  – ADTs + functions + objects
  – type-safety, data race-freeness by design
  – modules, components

• Layered concurrency model
  – Upper tier: asynchronous, no shared state, actor-based
  – Lower tier: synchronous, shared state, cooperative multitasking
Parallel Application

• The Sieve of Erathostenes.

• Prime number generation is highly relevant in authentication mechanisms, encryption and security in general.

• Parallel algorithm that partitions the memory among the available processors.

• Search and sieve operations are done in parallel.
Algorithm

- Restrict number of processors/ partitions: \( p < \sqrt{n} \)

```python
foreach partition p:
    init (p.list, start, end)

prime = p1.list[0]

while prime ^ 2 < n:
    foreach partition p:
        p.sieve (prime)
    p1.sieve (prime)
    prime = p1.get_next_prime(p1.list)
```
JAVA ABS-API

- Actor programming model.
- Default interfaces.
- Defining lambda expressions instead of anonymous inner classes or reflective code.
- Representation of actors that communicate using lambda expressions.
How does it work?

• Messages are modeled as instances of Runnable or Callable.

• Concurrent objects are modeled as classes that implement the Actor interface.

• Actors communicate by passing messages in terms of method invocations.

• Each actor has queue in which the messages are stored and retrieved in a FIFO order.
**ABS-API: where are we now?**

<table>
<thead>
<tr>
<th>ABS</th>
<th>ABS-API</th>
</tr>
</thead>
<tbody>
<tr>
<td>• OO programming constructs.</td>
<td>• JAVA OO programming constructs.</td>
</tr>
<tr>
<td>• COGs in ABS.</td>
<td>• Objects that implement the <em>Actor</em> interface.</td>
</tr>
<tr>
<td>• Asynchronous method calls.</td>
<td>• Lambda expressions.</td>
</tr>
<tr>
<td>• Synchronous method calls.</td>
<td>• JAVA method calls.</td>
</tr>
<tr>
<td>• Deployment components.</td>
<td>• Objects that implement the <em>ActorServer</em> interface.</td>
</tr>
</tbody>
</table>
Scheduling Real-Time Applications

• Actor queue modeled as an extension of
  – java.util.PriorityQueue<E>
  – java.util.concurrent.PriorityBlockingQueue<E>
  – java.util.HashMap<K,V>

• Pre-determined task dependencies (DDAG) and costs:
  – Task priorities
  – Load balancing
  – Completion time
Synchronization in JAVA backend

Main Thread

Thread 1

m3 = () -> A1.f()
A2.send(m3)

Actor 1

m4 = () -> A2.g()
A1.send(m4)

Actor 2

Thread 2
Control Flow (Futures)

m3 = () -> A1.f() A2.send(m3)

m4 = () -> A2.g() A1.send(m4)

future1.get()
Control Flow (Await)

Actor 1

Process 1

m3  m2  m1

await future1

Process 1'

Actor 1

Process 1' (suspended by the JVM)

m1
Control Flow (Await)

```
Actor 1
  m3  m2
  Process 1
    |
  Process 2
    |
Actor 2
  m2  m1
    return x;
    (future 1 return value)

Process 1'
(Suspended on future 1)
  m1
```
Control Flow (Await)
Thread Explosion Solution

• Modelling continuations as lambda expressions.

• Interleaving synchronous and asynchronous method calls (NESTED PARALLELISM.)

• Dynamic thread creation

• New formal model: this.m -> Future f = this!m; await f?