Scalable Performance for Scala Message-Passing Concurrency

Andrew Bate

Department of Computer Science
University of Oxford
Motivation

Multi-core commodity hardware

Non-uniform shared memory

Expose potential parallelism

Correctness and formal verification

Compatibility

int arr[x][y];
EMBEDDED
DOMAIN-SPECIFIC LANGUAGE

1. Embedded DSL
2. Bytecode rewriting
3. Channels
4. Scheduler
5. Deadlock detection
Why an Embedded DSL?

Ease of implementation

Leverage existing tools

Leverage known syntax

Higher-order functions
Rich type system

Lightweight syntax
Compile-time macros
def map[I, O](f: I => O)(in: ?[I], out: ![O]) =
proc {
    repeat {
        out !(f(in?))
    }
    run (proc { in.closein } || proc { out.closeout })
}
def tee[@specialized T](in: ?[T], outs: Seq[!T]) =
  proc {
    var v = null
    val outputs = (|| (out <- outs) proc { out ! v } )
    repeat { v = in?; run outputs }
    run (proc { in.closein } || (|| (out <- outs) proc { out.closeout } ))
  }
SCALABLE PERFORMANCE
through bytecode rewriting

1. Embedded DSL
2. Bytecode rewriting
3. Channels
4. Scheduler
5. Deadlock detection
CPS Transformation

Init
Call n()
Return

Pre-call
Call n(f)
Post-call
Prelude
Init
Return

rewinding
pausing
Analysing the call graph

Transform these methods
Engineering

Live variable analysis

Lazy load and store

Constant inlining
Functional Expressions

```scala
for (i <- 0 until n; j <- i until n) println(i)
```

Compiles to

```scala
intWrapper(0).until(n).foreach(
    i: Int => intWrapper(i).until(n).foreach(j: Int => println(i))
)
```

Transforms to

```scala
var i = 0
while (i < n) {
    var j = i
    while (j < n) { println(i); j += 1 }
    i += 1
}
```
More Features

Tail call optimisations

Shared memory

SBT plugin support
CHANNELS

1. Embedded DSL
2. Bytecode rewriting
3. Channels
4. Scheduler
5. Deadlock detection
More Features

Generalised alt

Specialization for primitives

Optimised extended rendezvous
Scheduler States

- Created
- Waiting
- Running
- Paused
- Terminated
Scheduling: Central FIFO
Scheduling: FIFO per thread
Scheduling: Batches per thread
Scheduling: Batches per thread

Scheduler

Dispatch Count = \( \max(\text{const} \times \text{Batch Length}, \text{Dispatch Limit}) \)
DEADLOCK DETECTION

1. Embedded DSL
2. Bytecode rewriting
3. Channels
4. Scheduler
5. Deadlock detection
Example

Tee

x2

Tee

x3

Merge

Tee

Prefix 1

Console

Tee

x5

Merge
Deadlock detected! The cycle of ungranted requests is:
Prefix1 -!-> Tee1        Tee3   -!-> x5
Tee1    -!-> Tee2        x5     -!-> Merge2
Tee2    -!-> Tee3        Merge2 -!-> Prefix1
PERFORMANCE EVALUATION

1. Embedded DSL
2. Bytecode rewriting
3. Channels
4. Scheduler
5. Deadlock detection
Ring topology

Time to pass a message 300 times around an n process ring (ms)

Number of processes spawned

- Java primitives
- CSO₂ FIFO Scheduler
- CSO₂ Batch Scheduler
Ring topology

Time to pass a message 300 times around an n process ring (ms)

Number n of processes spawned
Fully connected topology
Fully connected topology

- Erlang
- Scala Actors
- JCSP
- Java Primitives
- Occam
- CSO₂ FIFO Scheduler
- CSO₂ Batch Scheduler
- Go

Time to pass $n^2$ messages (ms)

Number $n$ of processes / actors spawned
Fully connected topology

Number $n$ of processes / actors spawned

Time to pass $n^2$ messages (ms)

- JCSP
- Occam
- CSO₂ FIFO Scheduler
- CSO₂ Batch Scheduler
- Go
Fully connected topology

Time to pass $n^2$ messages (ms)

Number of processes / actors spawned

CSO$_2$ FIFO Scheduler

CSO$_2$ Batch Scheduler

Go
Summary

• High performance library for building massively concurrent systems on the JVM

• Deadlock detection

• Outperforms Java primitives, JCSP, Scala Actors, Occam, and very close to Go