Straightforward parallelization of polynomial multiplication using parallel collections in Scala

Raphaël Jolly
Databeans

EOOPS 2013
Barcelona

Parallelization of symbolic computations

* Numeric computations

  Several arithmetic operations executed in parallel
  Linear algebra
  CPU-intensive

* Symbolic computations : polynomials

  Same as above, and:
  Arithmetic operation itself is parallelized
  Multiplication, division, gcd
  Reduction, Gröbner bases (multivariate)
  CPU and memory-intensive (cache issues)

Polynomial multiplication

Multivariate polynomials

Distributive representation

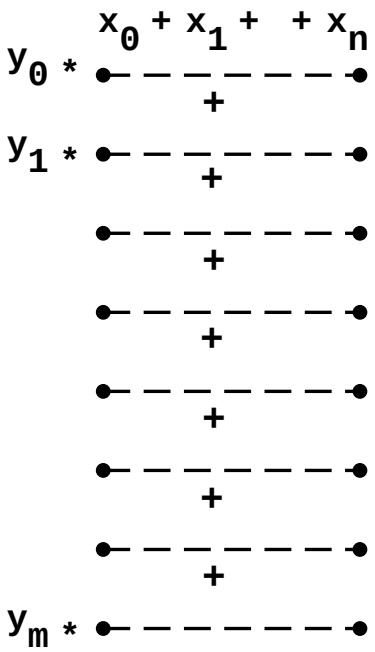
$$x = x_0 + x_1 + \ldots + x_n$$

$$x_\mathrm{i} = c_\mathrm{i} m_\mathrm{i}$$

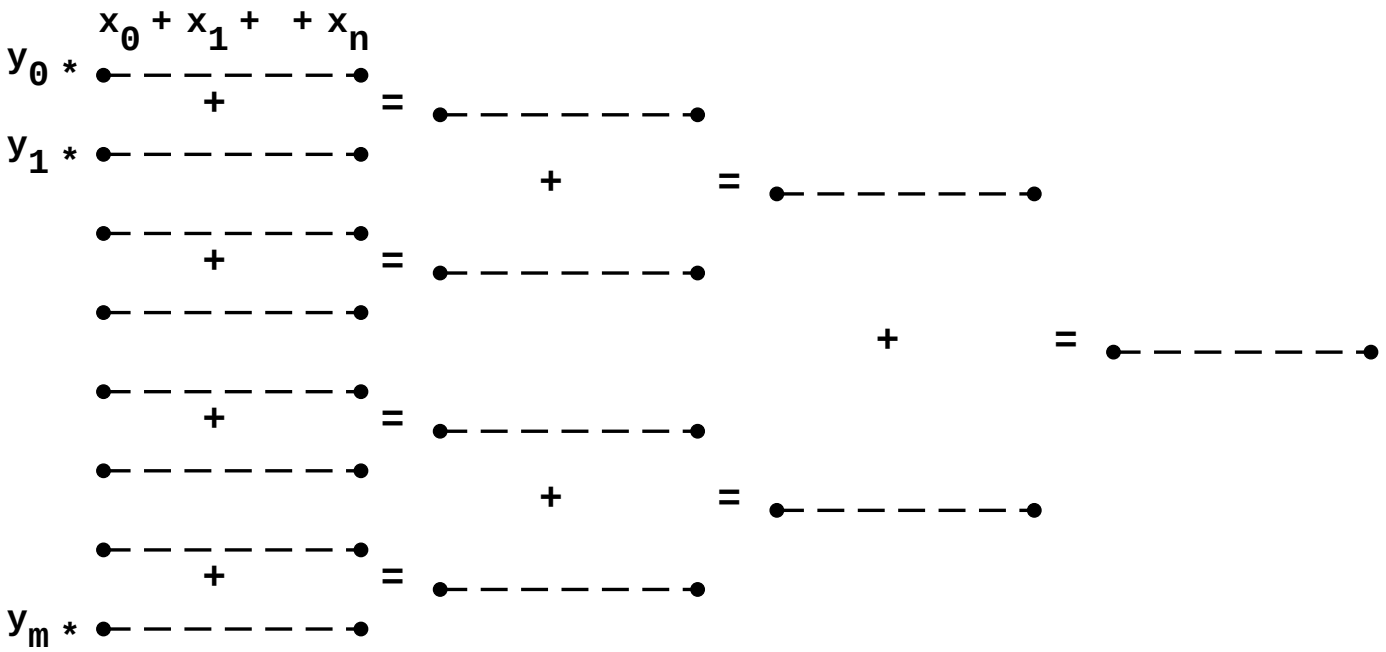
Product

$$xy$$

Polynomial multiplication : sequential



Polynomial multiplication : parallel



Polynomial multiplication : sequential

```
  type T = List[(Array[N], C)]

  def times(x: T, y: T) = (zero /: y) { (l, r) =>
    val (a, b) = r
    l + multiply(x, a, b)
  }
```

Polynomial multiplication : sequential

```
  type T = List[(Array[N], C)]

  def times(x: T, y: T) = y.foldLeft(zero)({ (l, r) =>
    val (a, b) = r
    l + multiply(x, a, b)
  })
```

Polynomial multiplication : sequential

```
  type T = List[(Array[N], C)]

  def times(x: T, y: T) = y.foldLeft(zero)({ (l, r) =>
    val (a, b) = r
    l + multiply(x, a, b)
  })

  def multiply(x: T, m: Array[N], c: C) = x.map { r =>
    val (s, a) = r
    (s * m, a * c)
  } filter { r =>
    val (_, a) = r
    !a.isZero
  }
```

Polynomial multiplication : parallel

```
  type T = List[(Array[N], C)]

  def times(x: T, y: T) = y.par.aggregate(zero)({ (l, r) =>
    val (a, b) = r
    l + multiply(x, a, b)
  }, _ + _)

  def multiply(x: T, m: Array[N], c: C) = x.map { r =>
    val (s, a) = r
    (s * m, a * c)
  } filter { r =>
    val (_, a) = r
    !a.isZero
  }
```

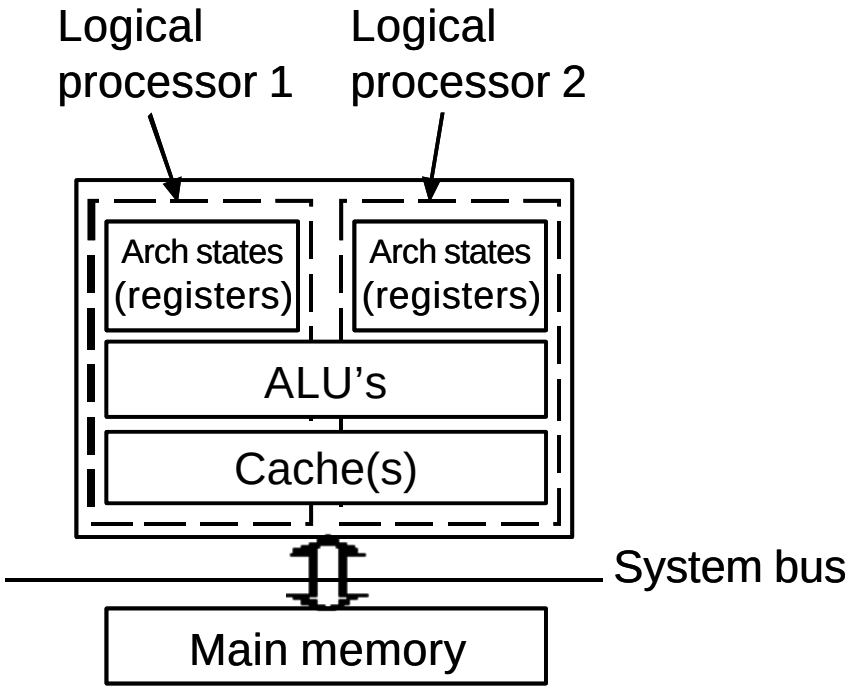
Polynomial multiplication : parallel

```
  type T = List[(Array[N], C)]

  def times(x: T, y: T) = y.par.aggregate(zero)({ (l, r) =>
    val (a, b) = r
    l + multiply(x, a, b)
  }, _ + _)

  def multiply(x: T, m: Array[N], c: C) = x.par.map { r =>
    val (s, a) = r
    (s * m, a * c)
  } filter { r =>
    val (_, a) = r
    !a.isZero
  }
```

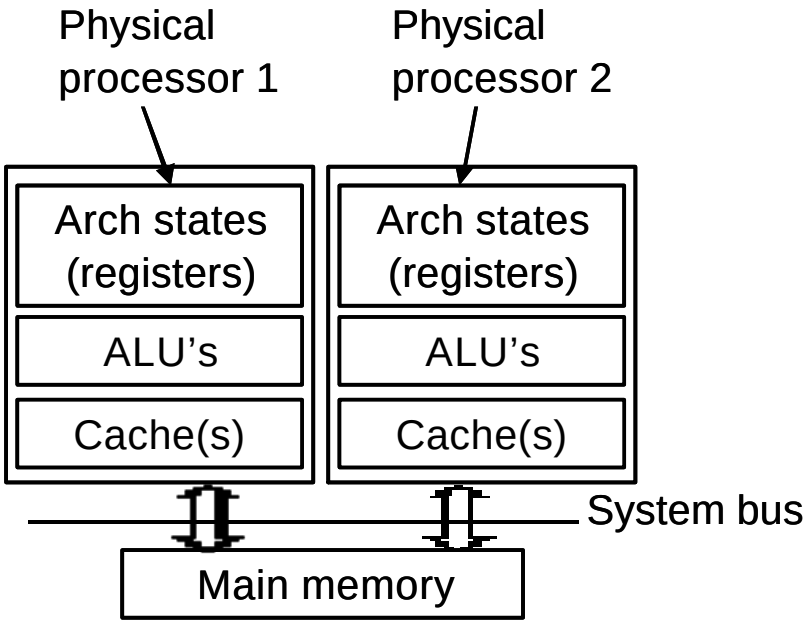
Experimental setup

- Intel Atom D410 at 1.66Ghz with ((32K, 24K), 512K) cache
- Single core
- Hyper-threading
- Parallel timings should not be worse than sequential
- Could be eventually better (20 %)
- Further experiments need to be done on multicore hardware

Experimental setup



Hyper-threading



Dual-processor

(Chen et al. Media Applications on Hyper-Threading Technology - Intel Technology Journal, Q1, 2002)

Test case

Squaring a sparse polynomial p^n with

$p = 1 + x + y + z \in \mathbb{Z}[x, y, z]$ and

$n \in \mathbb{N}$ sufficiently large : $n >= 20$

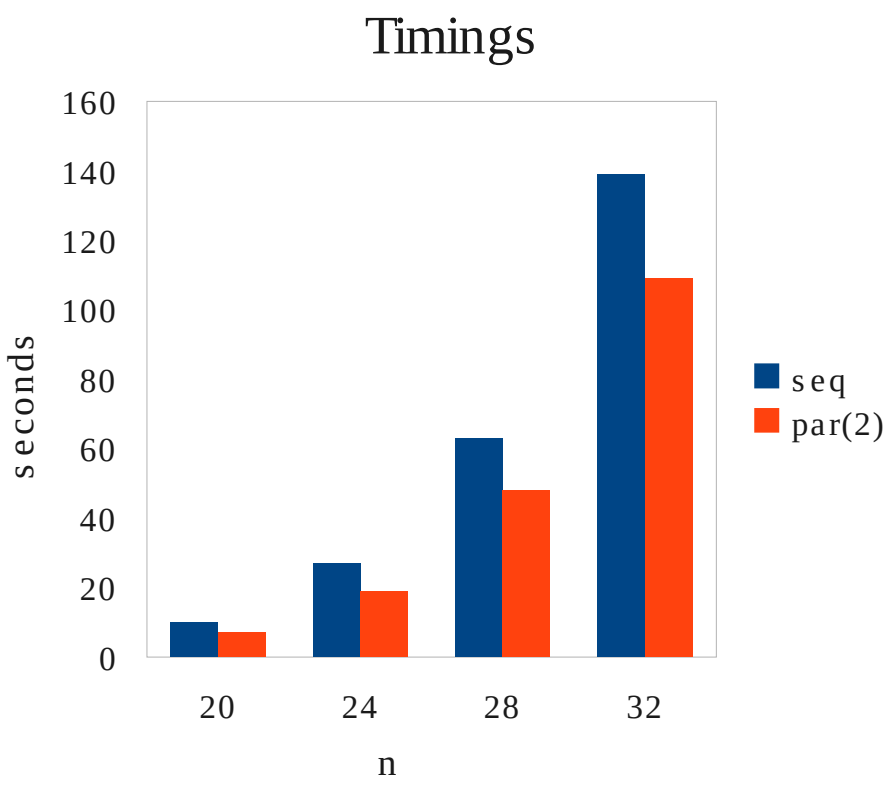
(Fateman, R. J. DRAFT: Comparing the speed of programs for sparse polynomial multiplication, 2002)

Test case : implementation

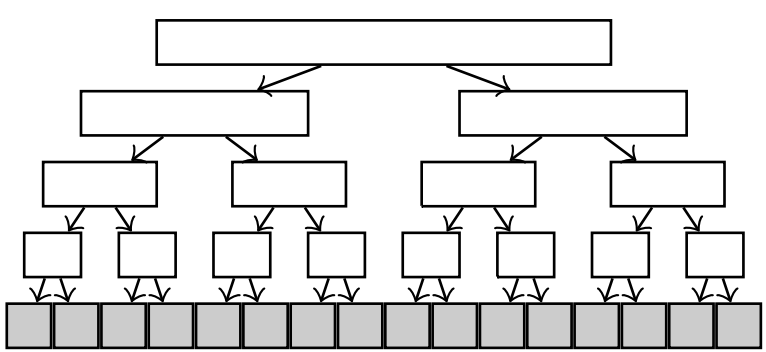
```
  import scas._
  import Implicits.ZZ
  implicit val r = Polynomial(ZZ, 'x, 'y, 'z)
  val Array(x, y, z) = r.generators
  val p = 1 + x + y + z
  val q = pow(p, 20)
  val q1 = 1 + q
  val q2 = q * q1
```

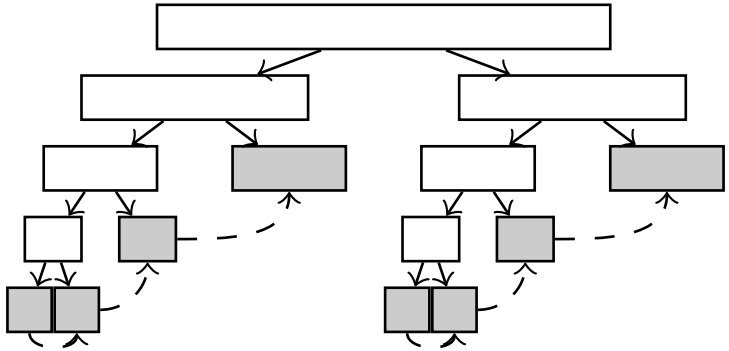
Timings

n	seq	par(2)	speedup
20	10	7	1.38
24	27	19	1.37
28	63	48	1.32
32	139	109	1.27



Fine-grained and exponential task splitting

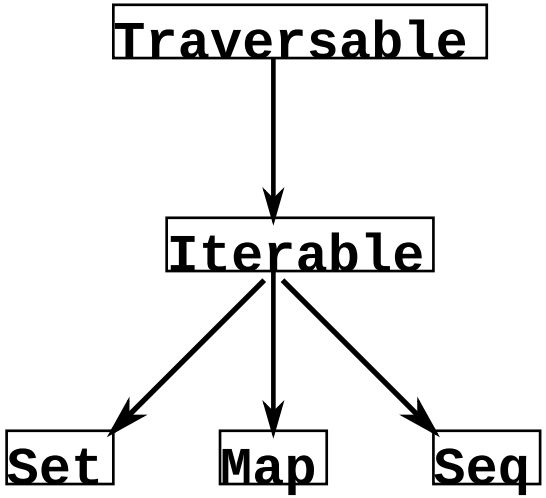




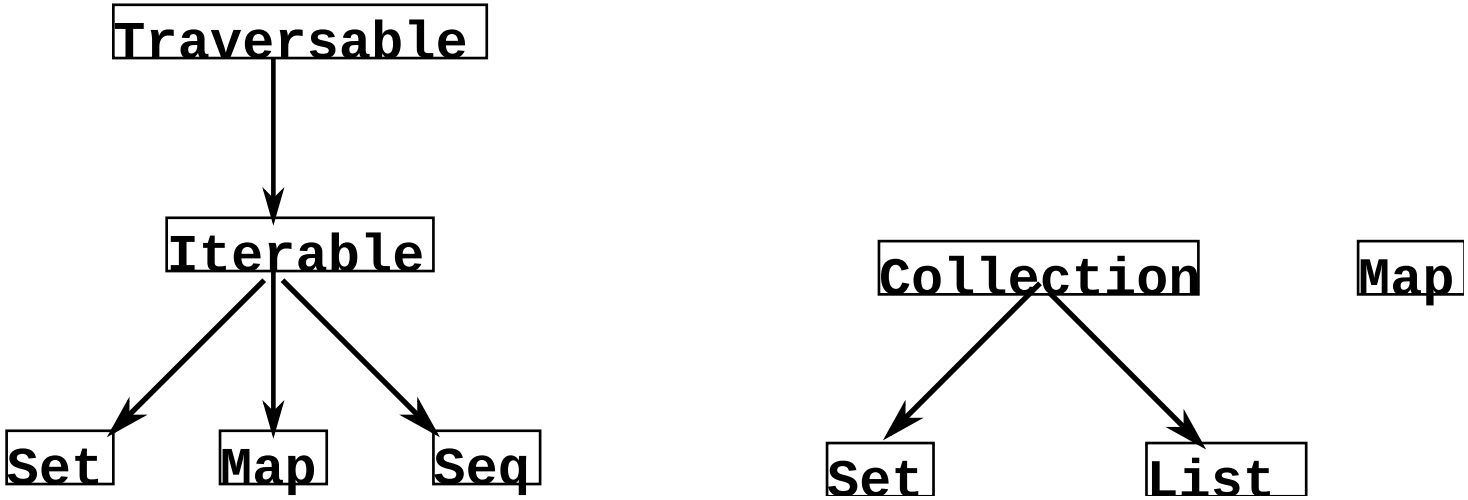
$$\text{threshold} = \max\left\{ n, \frac{1}{8P} \right\}$$

"stolen tasks are divided into exponentially smaller tasks until a threshold is reached and then handled sequentially starting from the smallest one, while tasks that came from the processor's own queue are handled sequentially straight away" (Prokopec, A.; Bawgell, P.; Rompf, T. & Odersky, M. On a Generic Parallel Collection Framework, 2011)

Collection base classes hierarchy



Collection base classes hierarchy

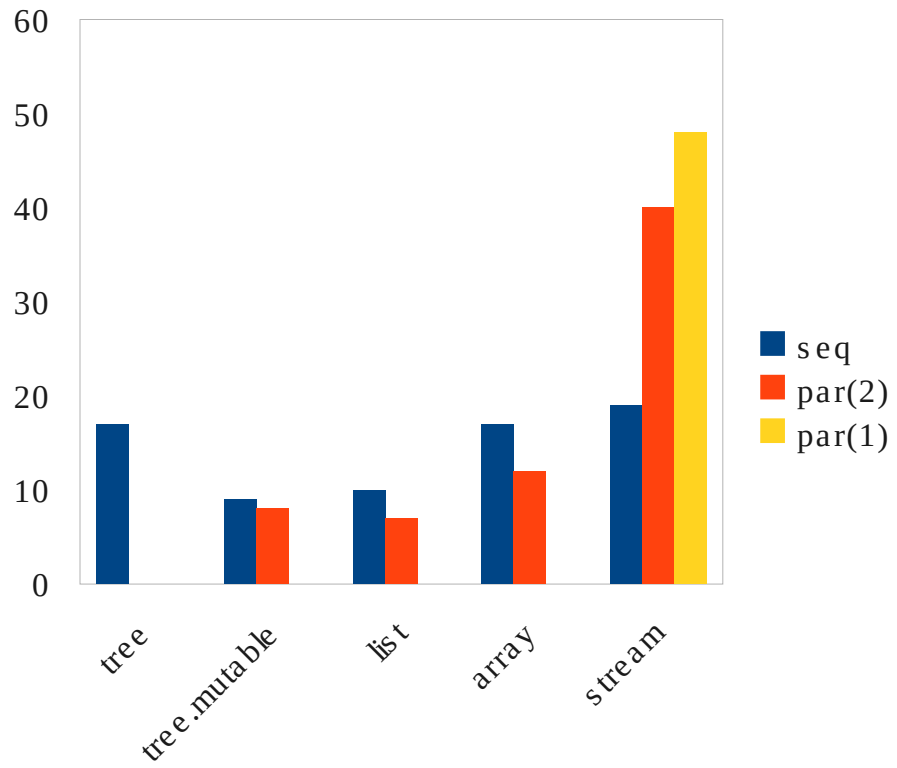


Traversable[A]

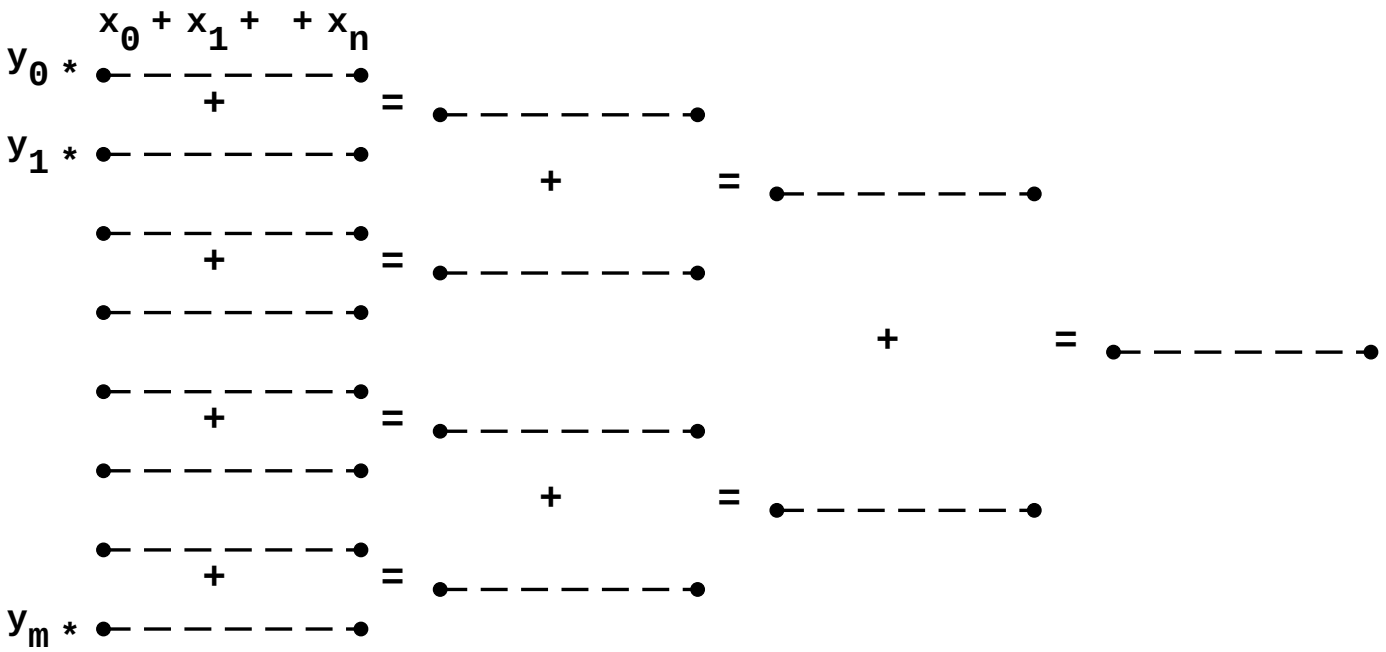
```
  def map[B, That](f: A => B): That
  def flatMap[B, That](f: A => GenTraversableOnce[B]): That
  def filter(p: A => Boolean): Traversable[A]
  def foreach[U](f: A => U): Unit
  def forall(p: A => Boolean): Boolean
  def exists(p: A => Boolean): Boolean
  def count(p: A => Boolean): Int
  def reduce[A1 >: A](op: (A1, A1) => A1): A1
  def aggregate[B](z: B)(seqop: (B, A) => B, combop: (B, B)
=> B): B
  def sum[B >: A](implicit num: Numeric[B]): B
  def product[B >: A](implicit num: Numeric[B]): B
  def min[B >: A](implicit cmp: Ordering[B]): A
  def max[B >: A](implicit cmp: Ordering[B]): A
```

Other data structures (n = 20)

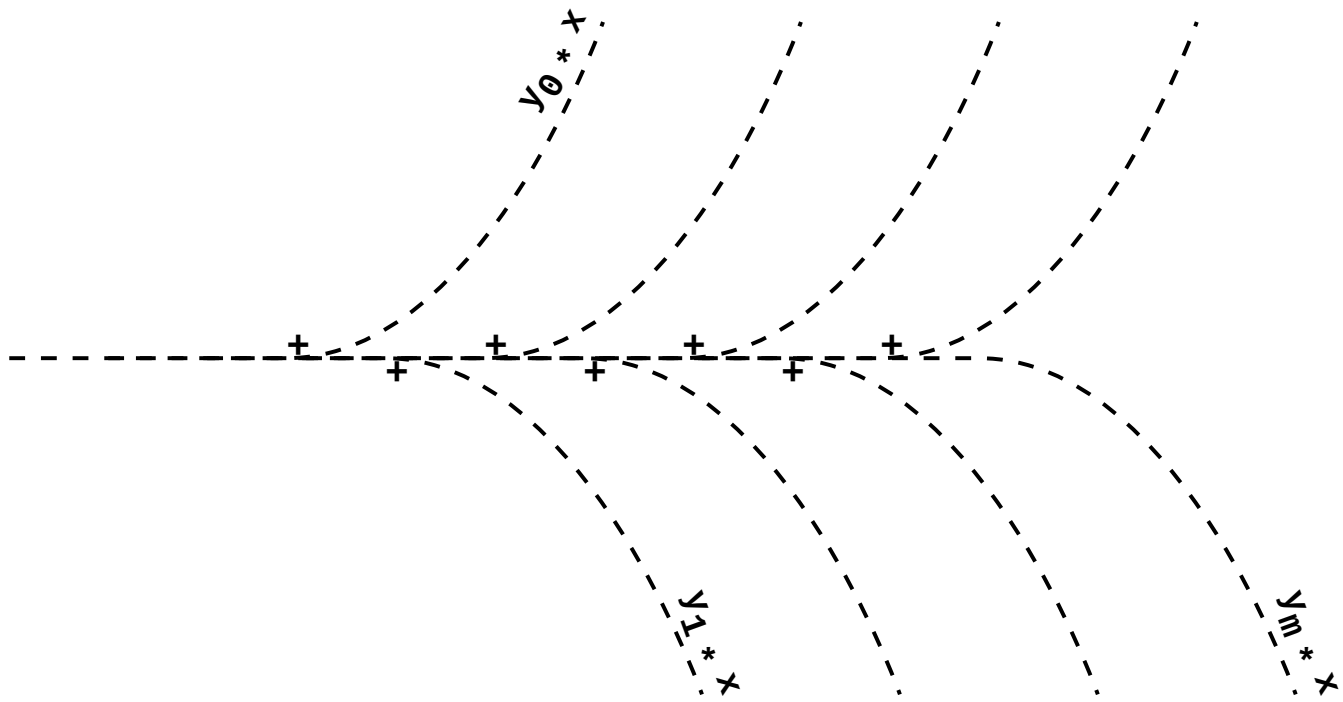
structure	seq	par(2)	par(1)
tree	17		
tree.mutable	9	8	
list	10	7	
array	17	12	
stream	19	40	48



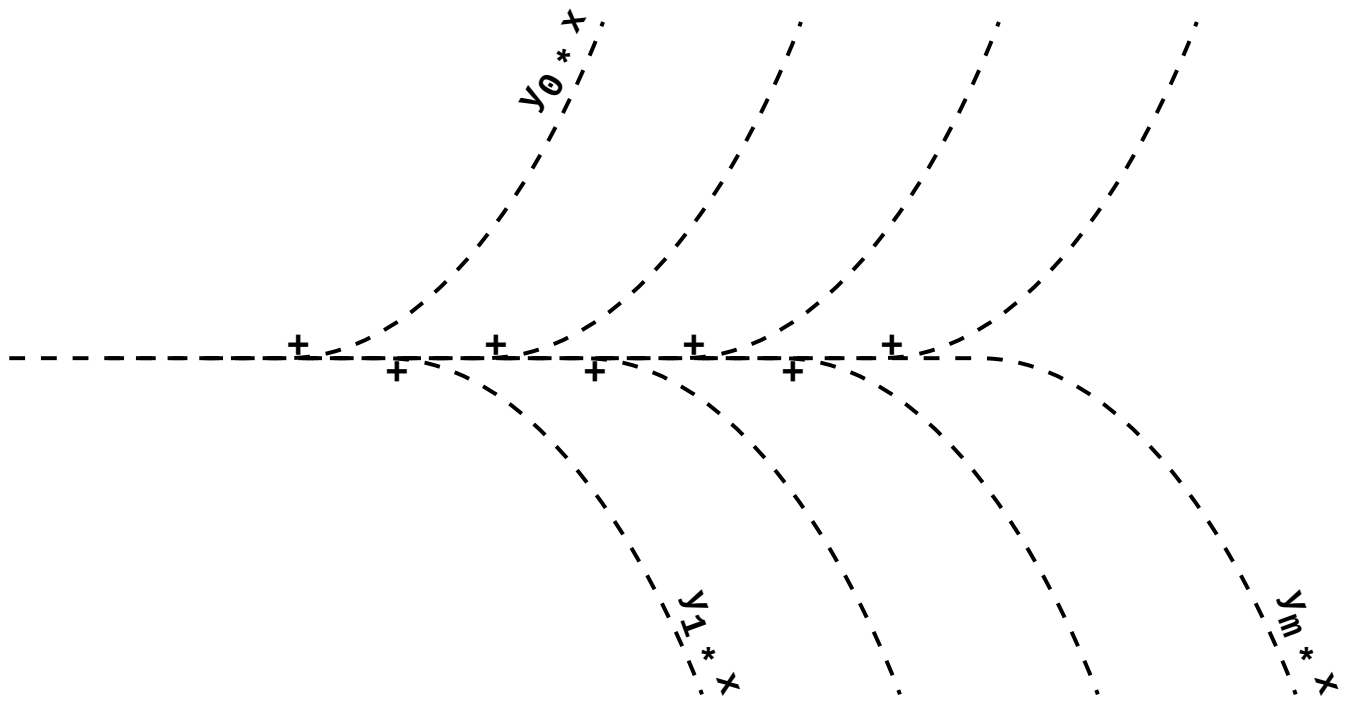
Data paralellism



Task paralellism



Task paralellism



Thank you !

http://github.com/rjolly/scas