Testing Speculative Work in a Lazy/Eager Parallel Functional Language

Alberto de la Encina, Ismael Rodríguez, and Fernando Rubio

Facultad de Informática

Universidad Complutense de Madrid

Index

Motivation

- Eden:
  - Sequentially lazy (Haskell)
  - Eager parallel computation

    (outputs of processes are always demanded)

- Eagerness ⇒ Risk of unneeded work

  ⇒ Need of profiling tools

- Functional language ⇒ Fast development but difficult profiling

  ⇒ no state!!

The Language Haskell

- functional (without side effects)
- polymorphic

```
length (1:2:3:[]) or length ('a':'b':'c':[])
> 3
```

- lazy (only demanded things are evaluated)

```
f x = 7
f (3/0)
> 7
```

- higher order (functions are first class citizens)

```
map f xs
```

The language Eden

Eden = Haskell + syntactical extensions for creating process topologies
+ eager evaluation of some expressions
+ eager process instantiations

- Process abstractions:

  p :: Process $(t_1, \ldots, t_m)$ $(t'_1, \ldots, t'_n)$

  p = **process** $(i_1, \ldots, i_m)$ **->** $(e_1, \ldots, e_n)$

  where equations

- Process instantiations:

  (#) :: Process a b -> a -> b

  e_1 # e_2

What is a Skeleton?

A skeleton is a parallel problem solving scheme

The aim is to reuse a parallel structure for many problems

It consists of:

1. A functional specification
2. One or more implementations. For each one:
   - A parallel algorithm
   - A cost model predicting the parallel execution time

Skeletons In Eden

Main idea: Processes are first class citizens in a higher-order language

$\Downarrow$

Processes can receive/be parameters

Functional specifications: Written in Haskell

Parallel algorithms: Written in Eden itself $\Rightarrow$ extensible

Parallel map

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

A simple parallel version creates one process per list element:

```
map_par :: (a -> b) -> [a] -> [b]
map_par f xs = [pf #  x | x <- xs] `using` spine
    where pf = process x -> f x
```

Parallel map — farm

A better approach creates a fix number of processes:

```
map_farm :: Int -> (Int -> [a] -> [[a]]) -> ([[b]] -> [b]) ->
            (a -> b) -> [a] -> [b]
map_farm np unshuffle shuffle f tasks
  = shuffle (map_par (map f) (unshuffle np tasks))
```

Different strategies provided that: `shuffle (unshuffle np xs) == xs`

Introduction to Hood

- In imperative programs debugging is simple. Intermediate values and the final result can be shown.
- Hood allows the programmer to observe the intermediate structures.

```
natural = reverse . map (`mod` 10)
                 . takeWhile (/= 0) . iterate (`div` 10)
natural 3408
> 3:4:0:8:[]

-- after iterate 3408:340:34:3:0:_
-- after takeWhile 3408:340:34:3:[]
-- after map 8:0:4:3:[]
```

Introduction to Hood

observe: String → a → a

- observe s a = a
- as a side effect, the value of a associated to s is saved in a file.

```
natural = reverse
      . observe "after map"           . map (`mod` 10)
      . observe "after takeWhile" . takeWhile (/= 0)
      . observe "after iterate"     . iterate (`div` 10)
```

```
observe "sum" sum (4:2:5:[])
-- sum { \ (4:2:5:[]) -> 11  }
```

```
observe "length" length (4:2:5:[])
-- length { \ (_:_:_:[]) -> 3 }
```

Observing communication of processes — an example

- Process for generating infinite primes $\geq$ `n`:

```
pprimes = process n -> outputs
  where outputs = generatePrimes n
generatePrimes x = if (isPrime x) then x : restOfPrimes
                                  else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
```

- Process for computing the shortest list of consecutive primes from `initialNumber` such that its multiplication is $\geq$ `threshold`:

```
myComputation initialNumber threshold = take neededNumber primes
  where primes    = pprimes # initialNumber
        products = scanl (*) 1 primes
        neededNumber = length (takeWhile (< threshold) products)
```

Observing communication of processes — an example

- Process for generating infinite primes $\geq$ `n`:

```
pprimes = process n -> (observe "outsFromProcess" outputs)
  where outputs = generatePrimes n
generatePrimes x = if (isPrime x) then x : restOfPrimes
                                  else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
```

- Process for computing the shortest list of consecutive primes from `initialNumber` such that its multiplication is $\geq$ `threshold`:

```
myComputation initialNumber threshold = take neededNumber primes
  where primes    = pprimes # initialNumber
        products = scanl (*) 1 primes
        neededNumber = length (takeWhile (< threshold) products)
```

Observing communication of processes — an example

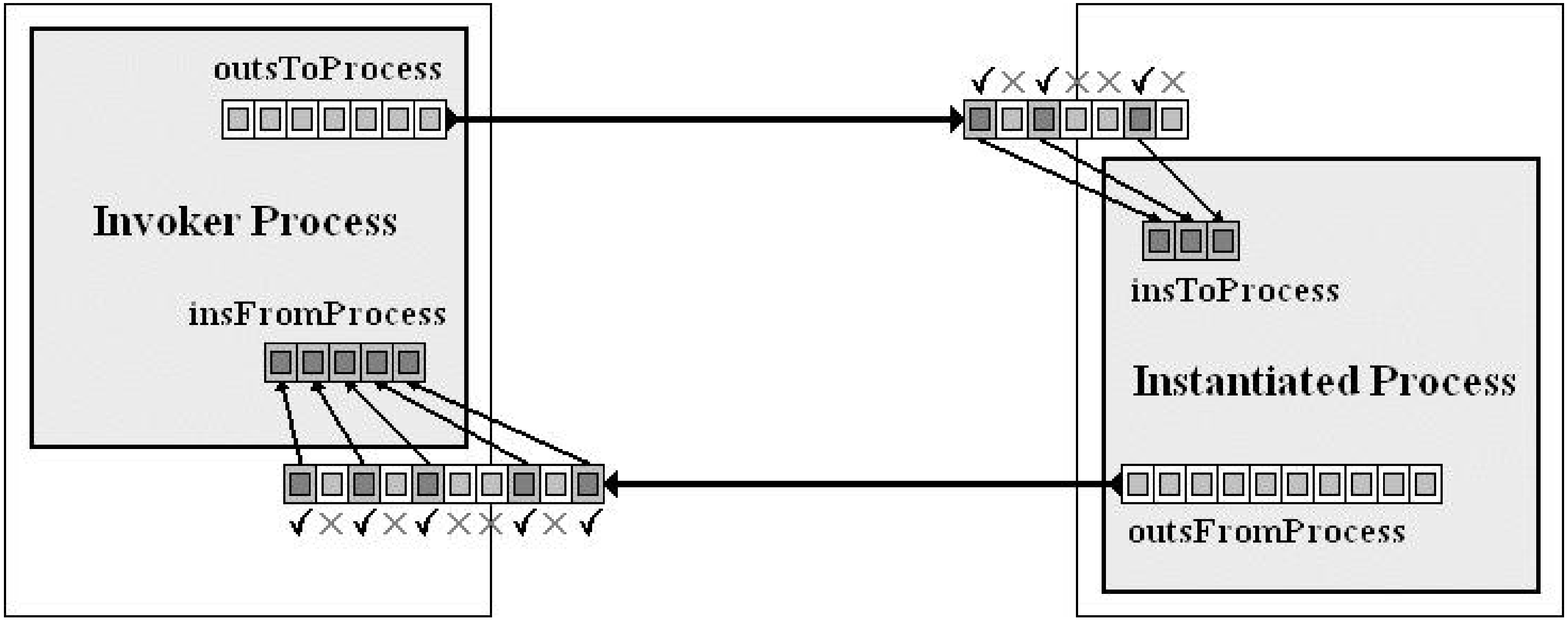
- Process for generating infinite primes ≥ `n`:

```
pprimes = process n -> (observe "outsFromProcess" outputs)
  where outputs = generatePrimes n
generatePrimes x = if (isPrime x) then x : restOfPrimes
                                  else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
```

- Process for computing the shortest list of consecutive primes from `initialNumber` such that its multiplication is ≥ `threshold`:

```
myComputation initialNumber threshold = take neededNumber primes
  where primes    = observe "insFromProcess" (pprimes # initialNumber)
        products = scanl (*) 1 primes
        neededNumber = length (takeWhile (< threshold) products)
```

The general case



The general case

- `processObs`: function to run a given function as a new process

```
processObs f = process ins -> outs
           where outs = f ins'
                 ins' = ins
```

- `##`: (dummy) function to instantiate a process

```
p ## actualParameters =
          p # actualParameters
```

The general case

- `processObs`: construction to observe the instantiated process

```
processObs f = process ins -> (observe "outsFromProcess" outs)
           where outs = f ins'
                 ins' = observe "insToProcess" ins
```

- `##`: (dummy) function to instantiate a process

```
p ## actualParameters =
          p # actualParameters
```

The general case

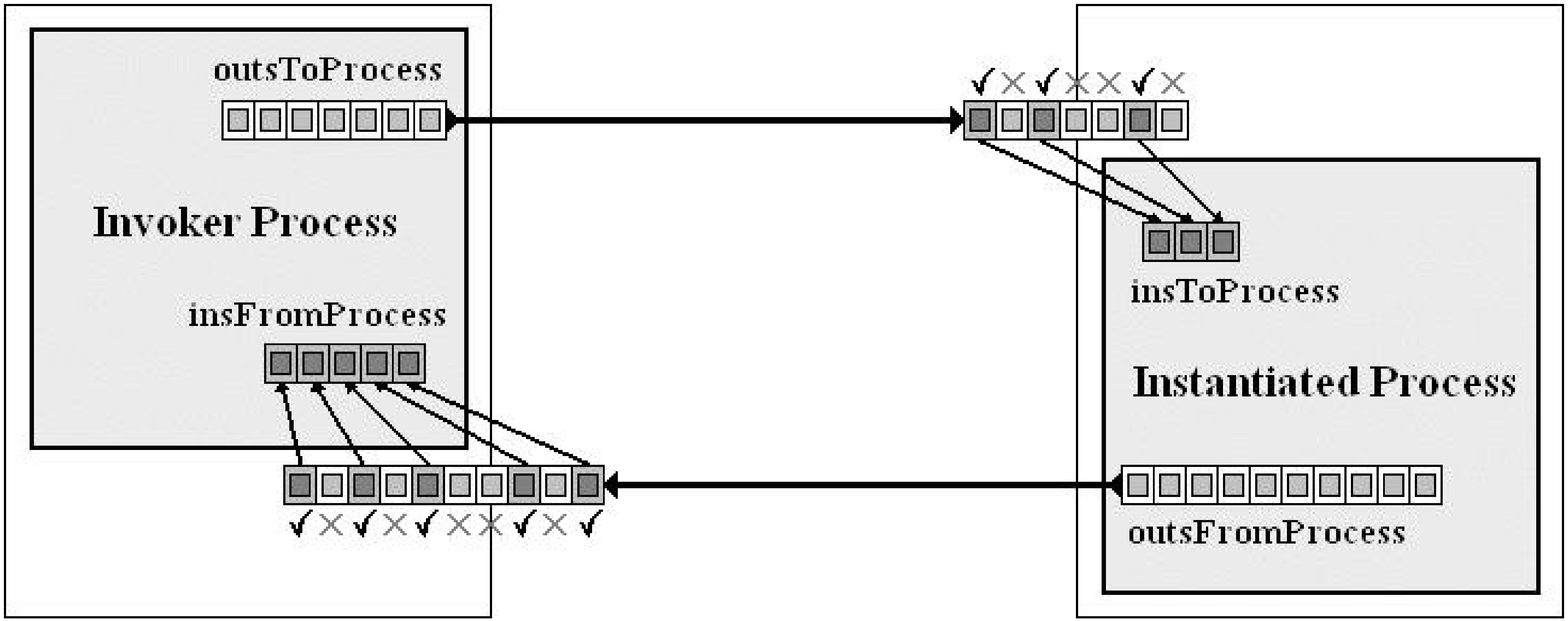
- `processObs`: construction to observe the instantiated process

```
processObs f = process ins -> (observe "outsFromProcess" outs)
            where outs = f ins'
                  ins' = observe "insToProcess" ins
```

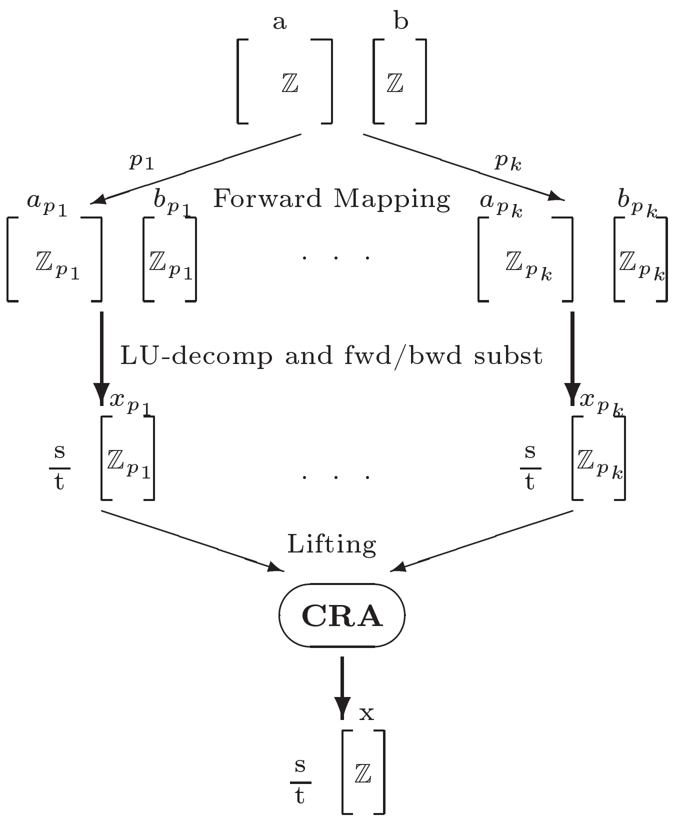
- `##`: construction to observe the invoker process

```
p ## actualParameters =
  observe "insFromProcess"
          (p # (observe "outsToProcess" actualParameters))
```

The general case



LinSolv Scheme



Exact solution arbitrary precision integers

$Ax = b$ where $A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^n, n \in \mathbb{N}$

Multiple homomorphic images approach:

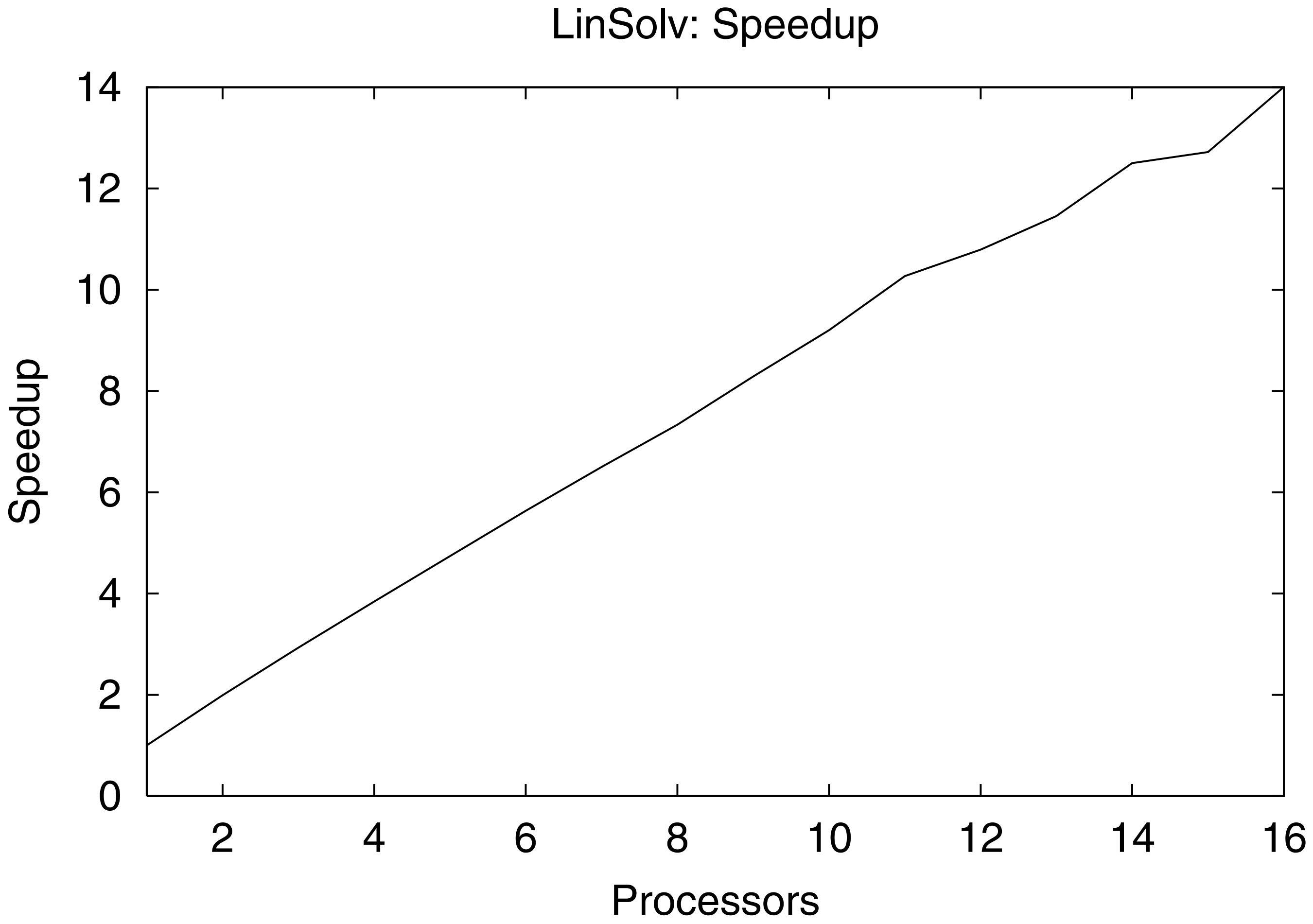
- map the input data into several homomorphic images
- compute the solution in each image
- combine the results of all images to a result in the original domain.

LinSolv: To Speculate or not To Speculate

How much speculation should be included in LinSolv?

- `mapfarm`: hundreds of useless messages — Too much!!!!!!!!
- `on demand`: No speculation at all — Bottleneck!!!
- `maprw`: 15 useless messages — Controlled

LinSolv: Results on a Beowulf



Conclusions and Future Work

- Profiling tool reporting speculative work
  - Rewriting Hood to parallelize it
  - Rewriting basic Eden constructions
- Rewriting the skeletons library
- Application to a real example

- Graphical interface
- Application to more examples
- Formal semantics