



LVars:

**Lattice-based Data Structures
for Deterministic Parallelism**

Lindsey Kuper and Ryan Newton
Indiana University

FHPC '13, Boston, MA, USA
September 23, 2013

What does this program evaluate to?

```
p = do
  num <- newEmptyMVar
  forkIO $ do putMVar num 3
  forkIO $ do putMVar num 4
  v <- takeMVar num
  return v
```


[illegible]

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO $ do putMVar num 3
  forkIO $ do putMVar num 4
  v <- takeMVar num
  return v
```

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO $ do putMVar num 3
  forkIO $ do putMVar num 4
  v <- takeMVar num
  return v
```

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO $ do putMVar num 3
  forkIO $ do putMVar num 4
  v <- takeMVar num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 3
  fork $ do put num 4
  v <- get num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 3
  fork $ do put num 4
  v <- get num
  return v
```

Tesler and Enea, 1968

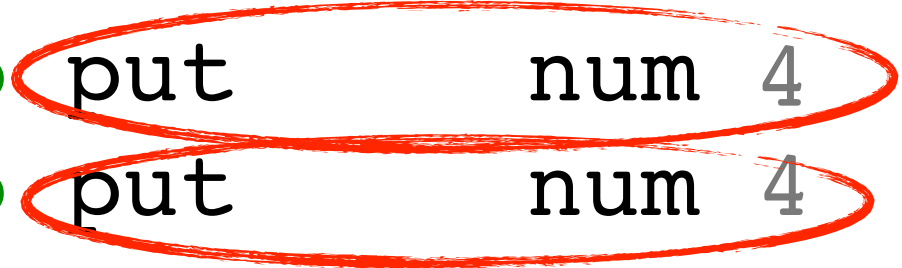
Arvind *et al.*, 1989

IVars

```
./ivar-example +RTS -N2
ivar-example: multiple put
```


Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 4
  fork $ do put num 4
  v <- get num
  return v
```



Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 4
  fork $ do put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

Deterministic programs that single-assignment forbids

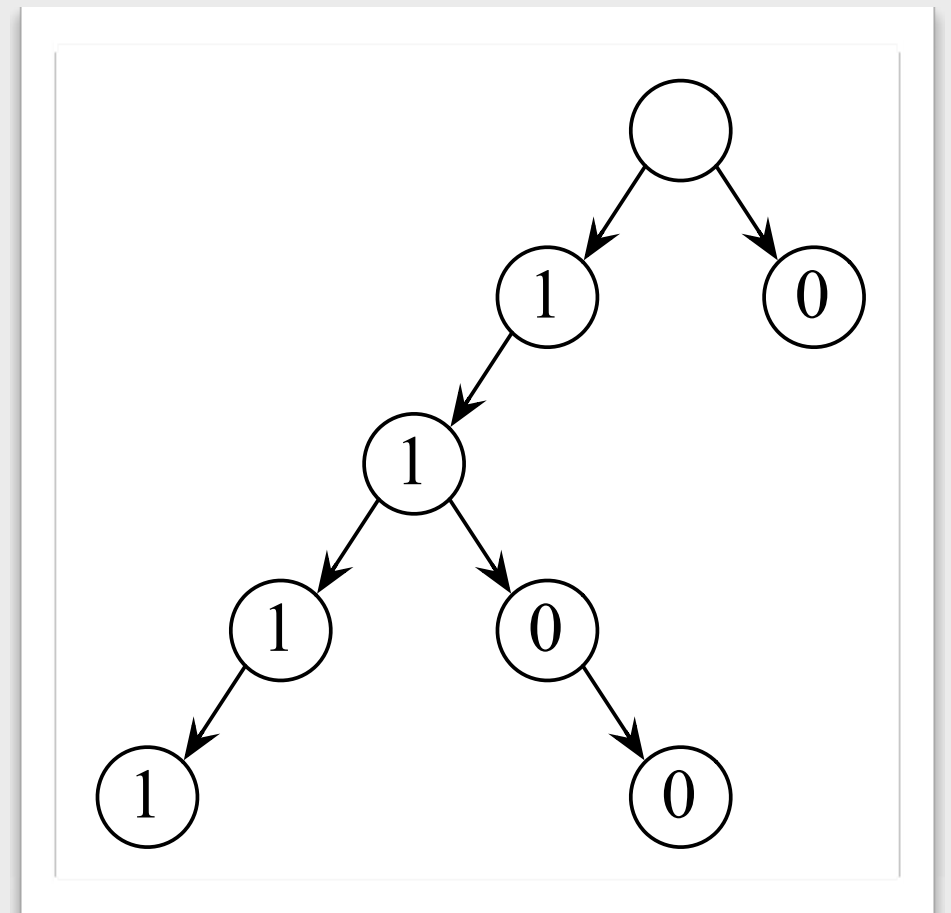
```
p :: Par Int
p = do
  num <- new
  fork $ do put num 4
  fork $ do put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 4
  fork $ do put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```



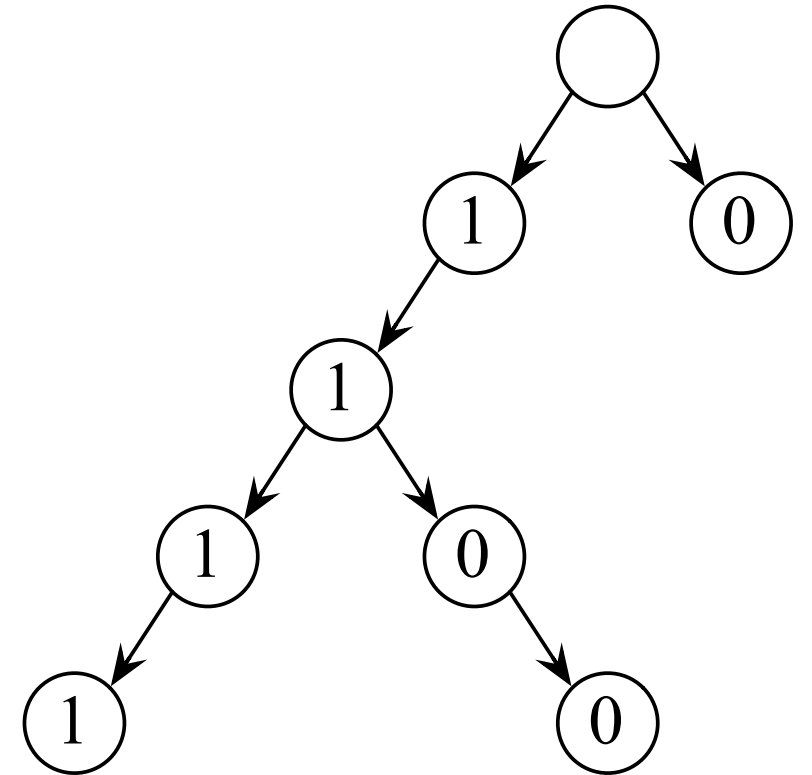
Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 4
  fork $ do put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

do

```
fork    $ do insert t "0"
fork    $ do insert t "1100"
fork    $ do insert t "1111"
v <- get t
return v
```



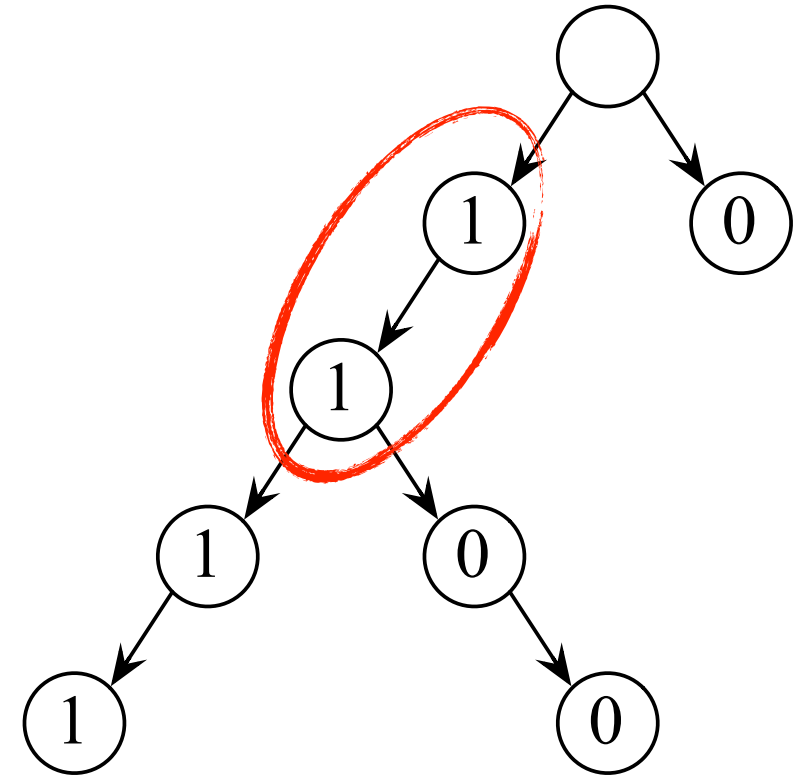
Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork $ do put num 4
  fork $ do put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

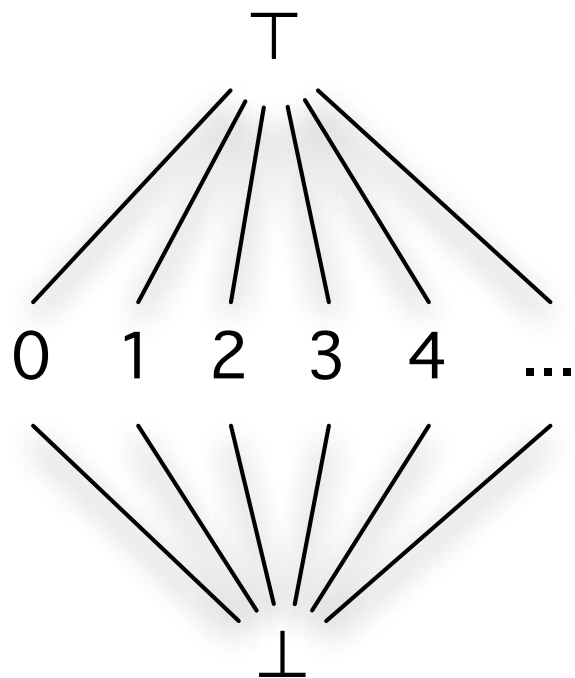
do

```
fork    $ do insert t "0"
fork    $ do insert t "1100"
fork    $ do insert t "1111"
v <- get t
return v
```



LVars: Multiple *least-upper-bound* writes

num



Raises an error, since $3 \sqcup 4 = \top$

do

fork \$ **do** put num 3

fork \$ **do** put num 4

Works fine, since $4 \sqcup 4 = 4$

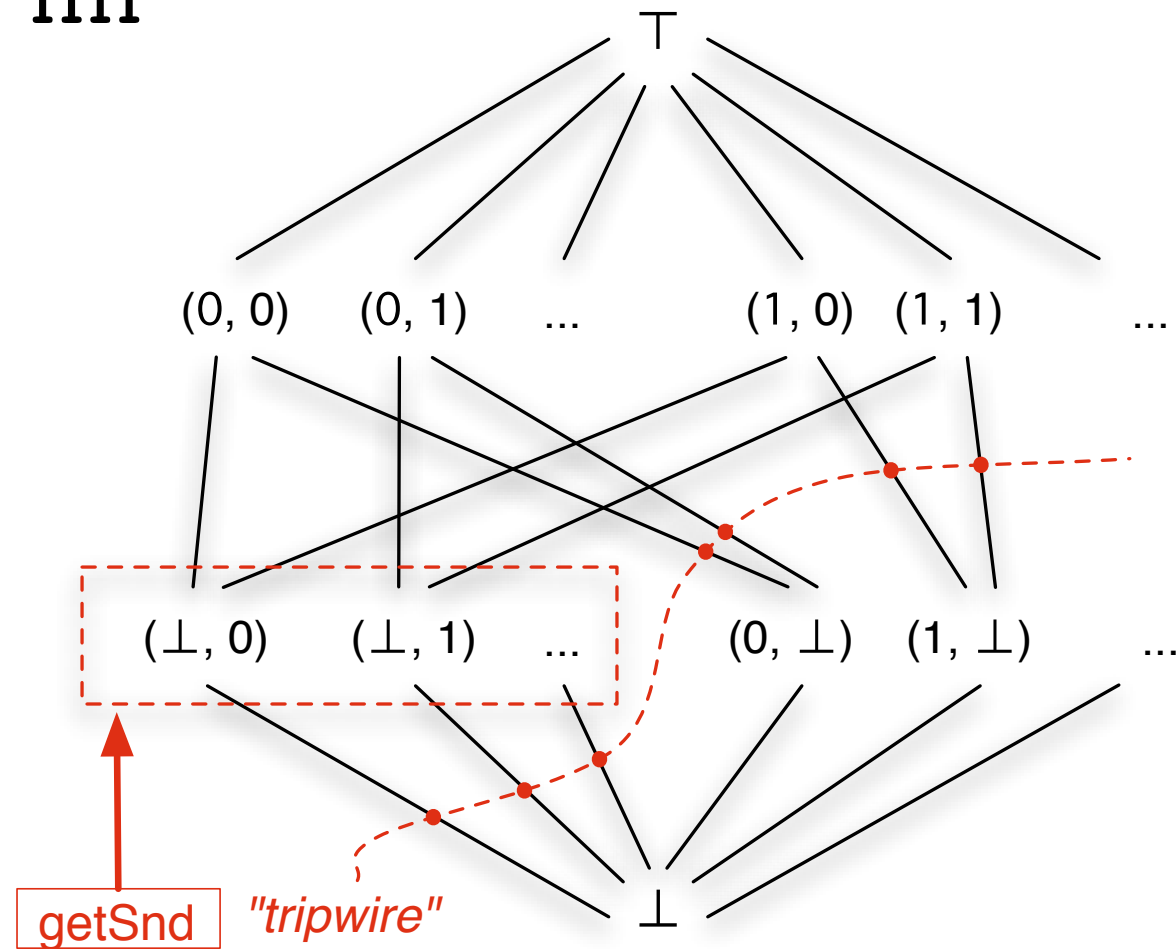
do

fork \$ **do** put num 4

fork \$ **do** put num 4

LVars: Threshold reads

nn

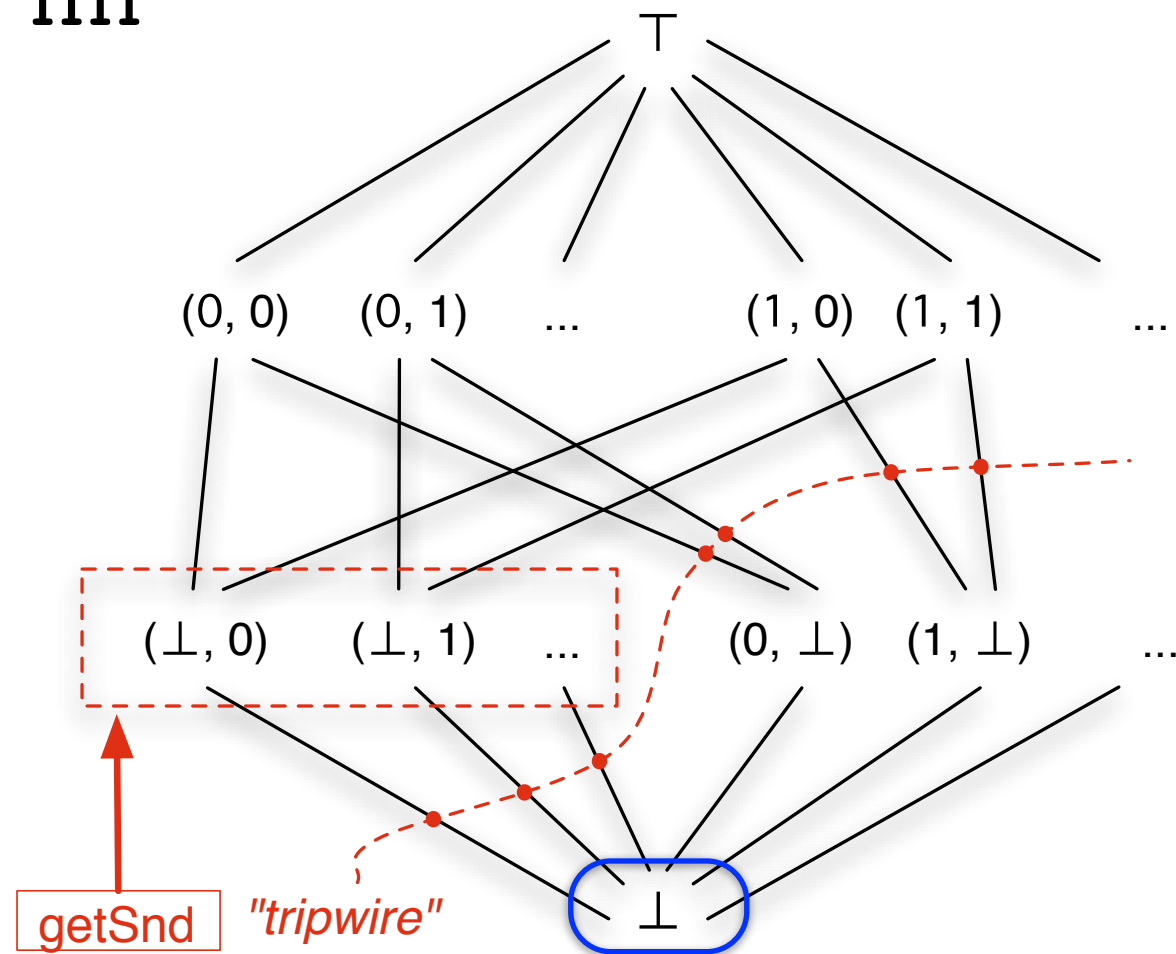


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```


LVars: Threshold reads

nn

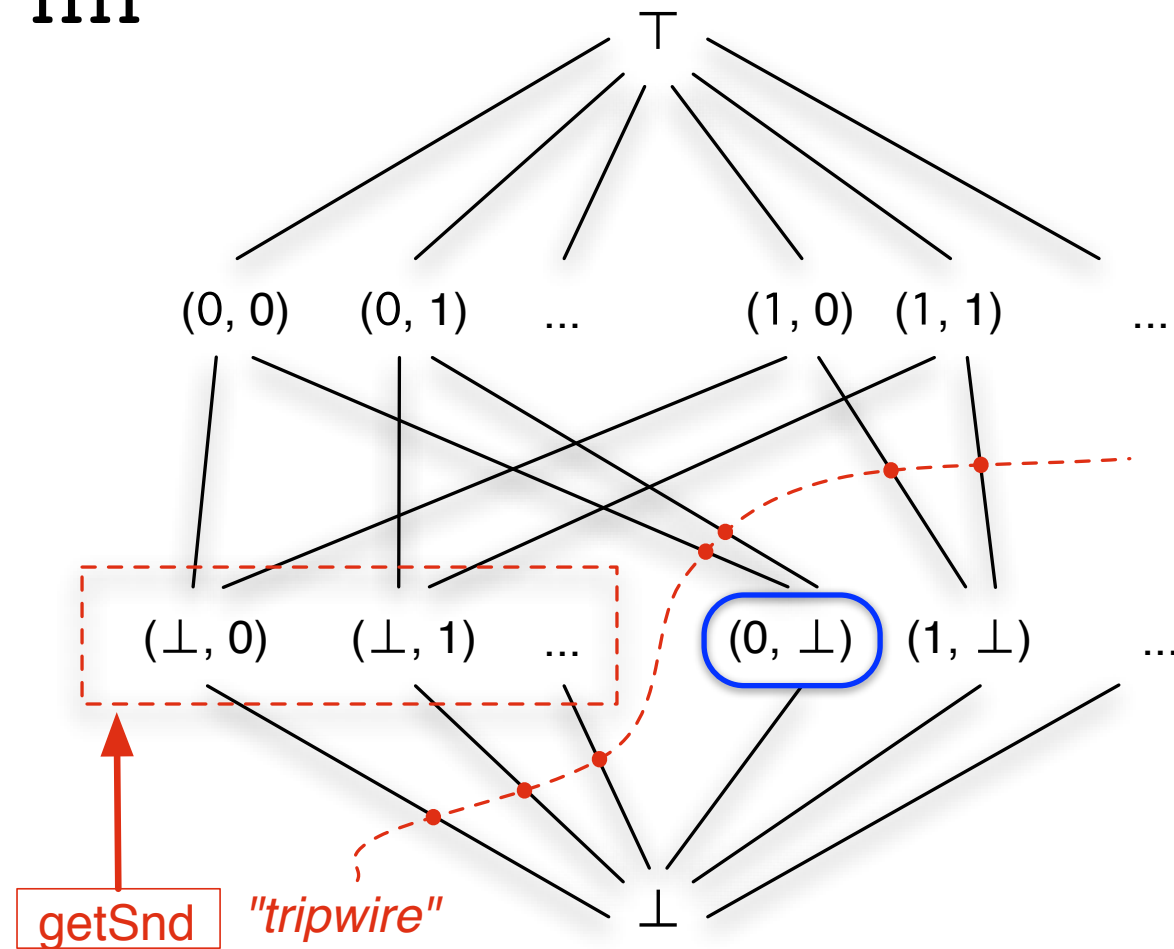


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

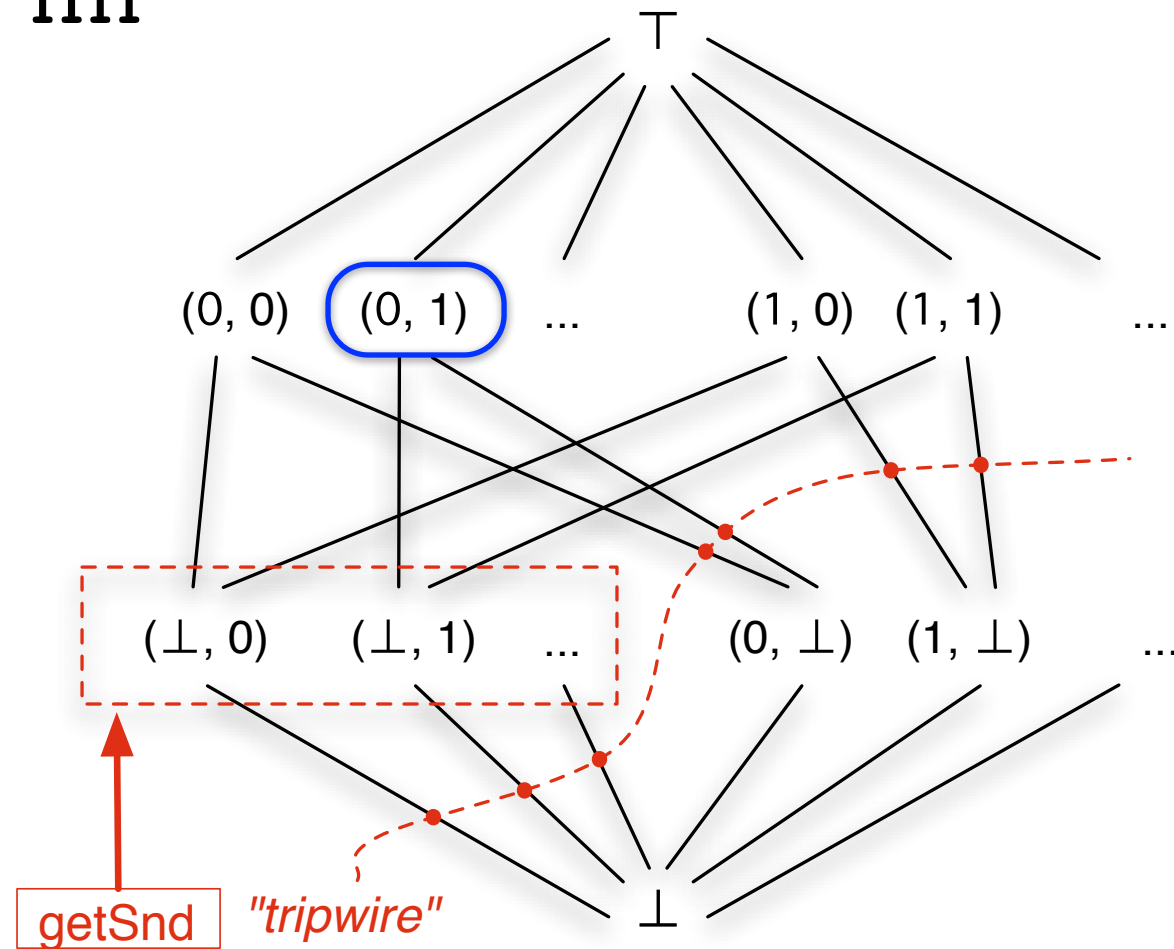


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

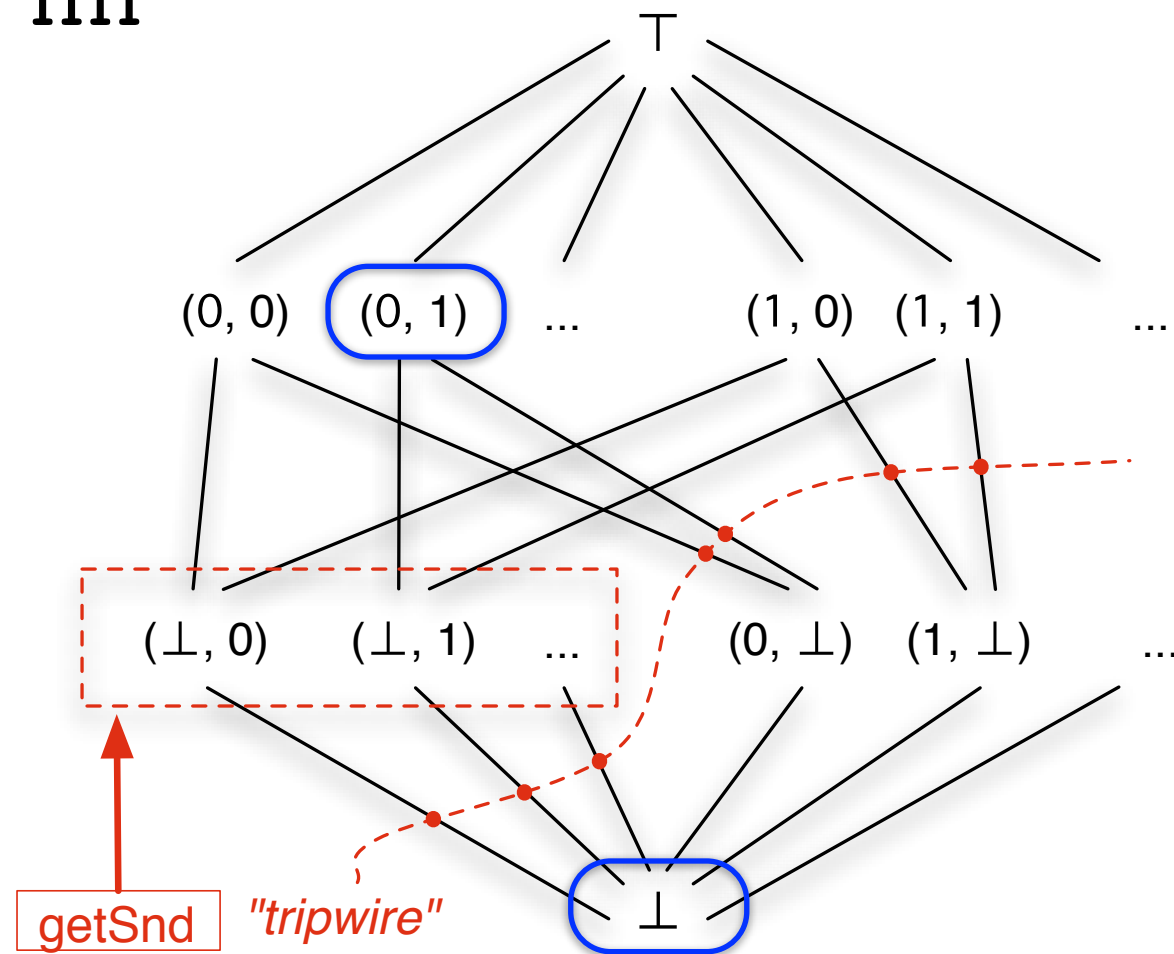


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

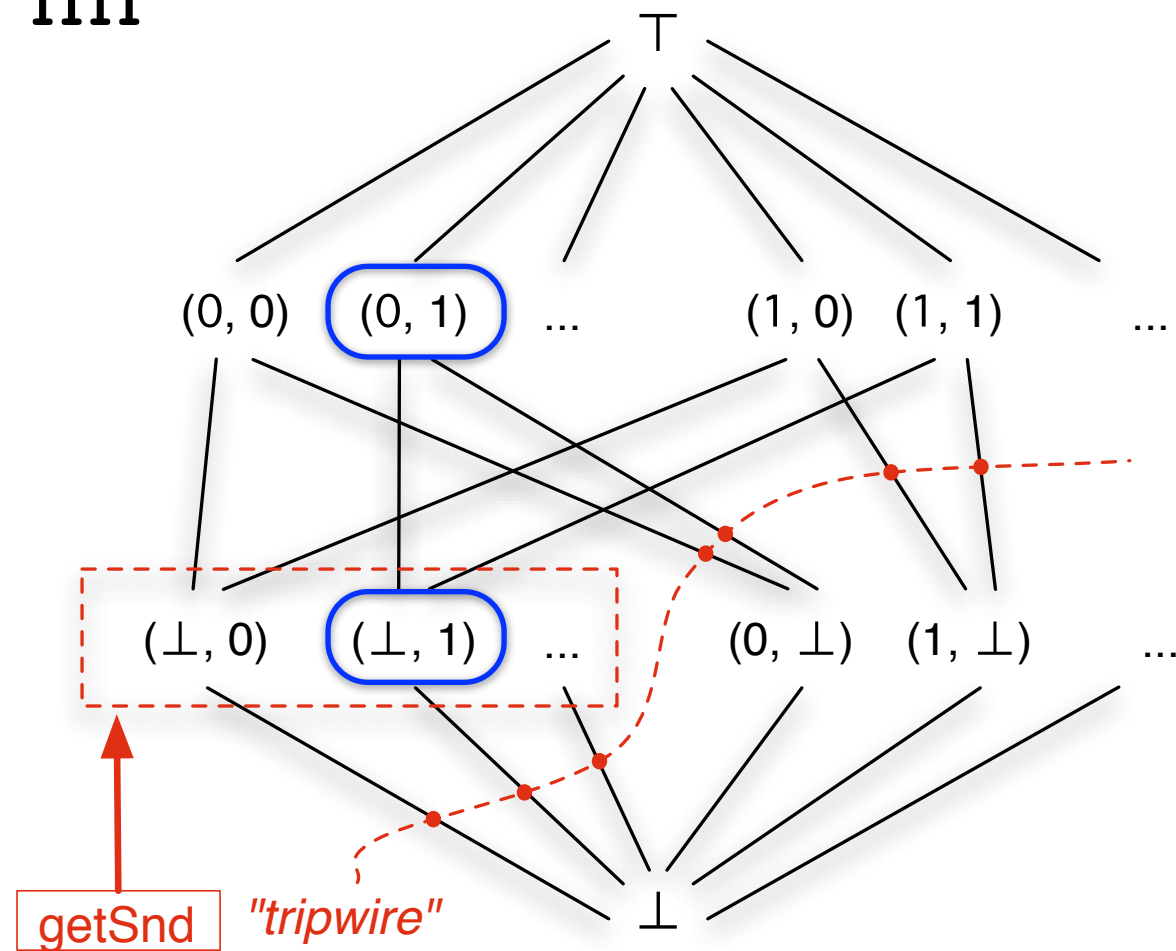


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```


LVars: Threshold reads

nn

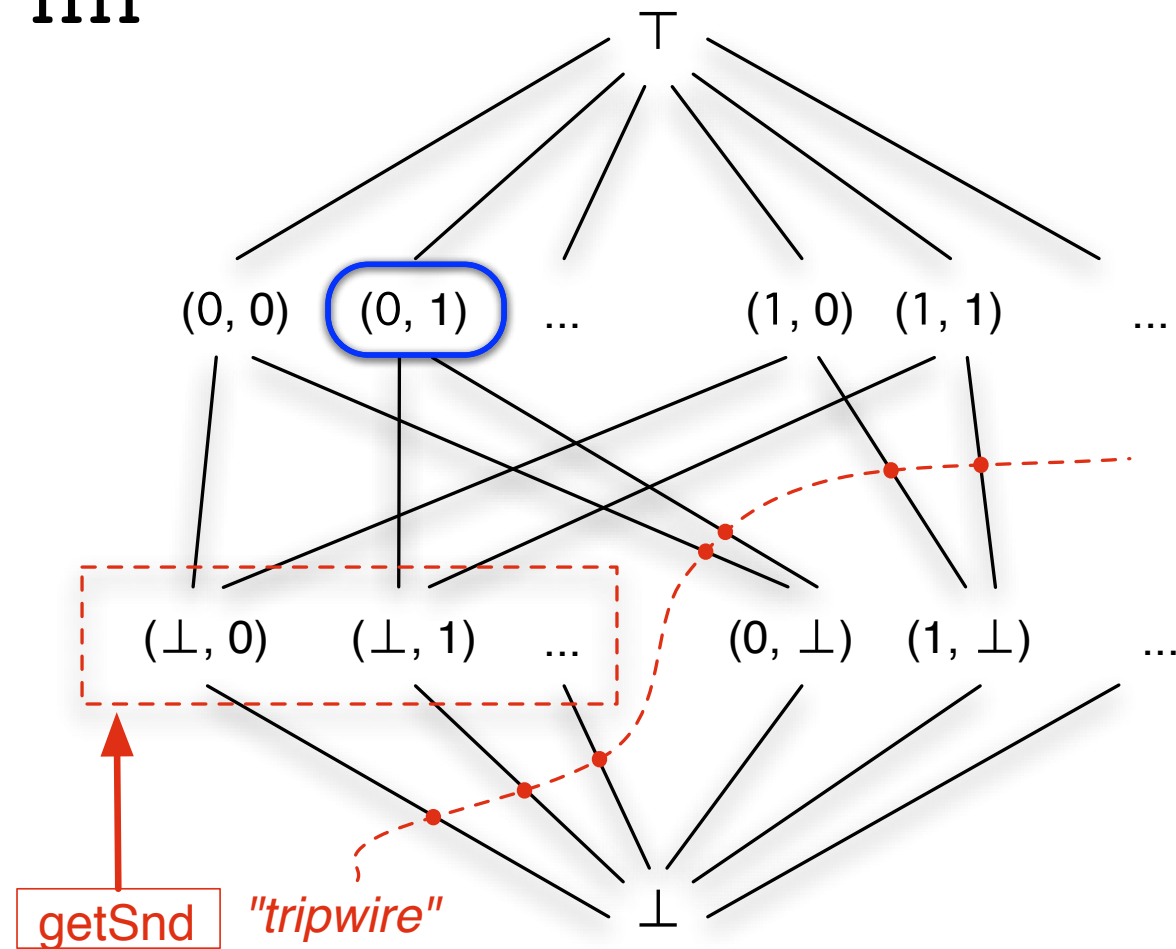


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

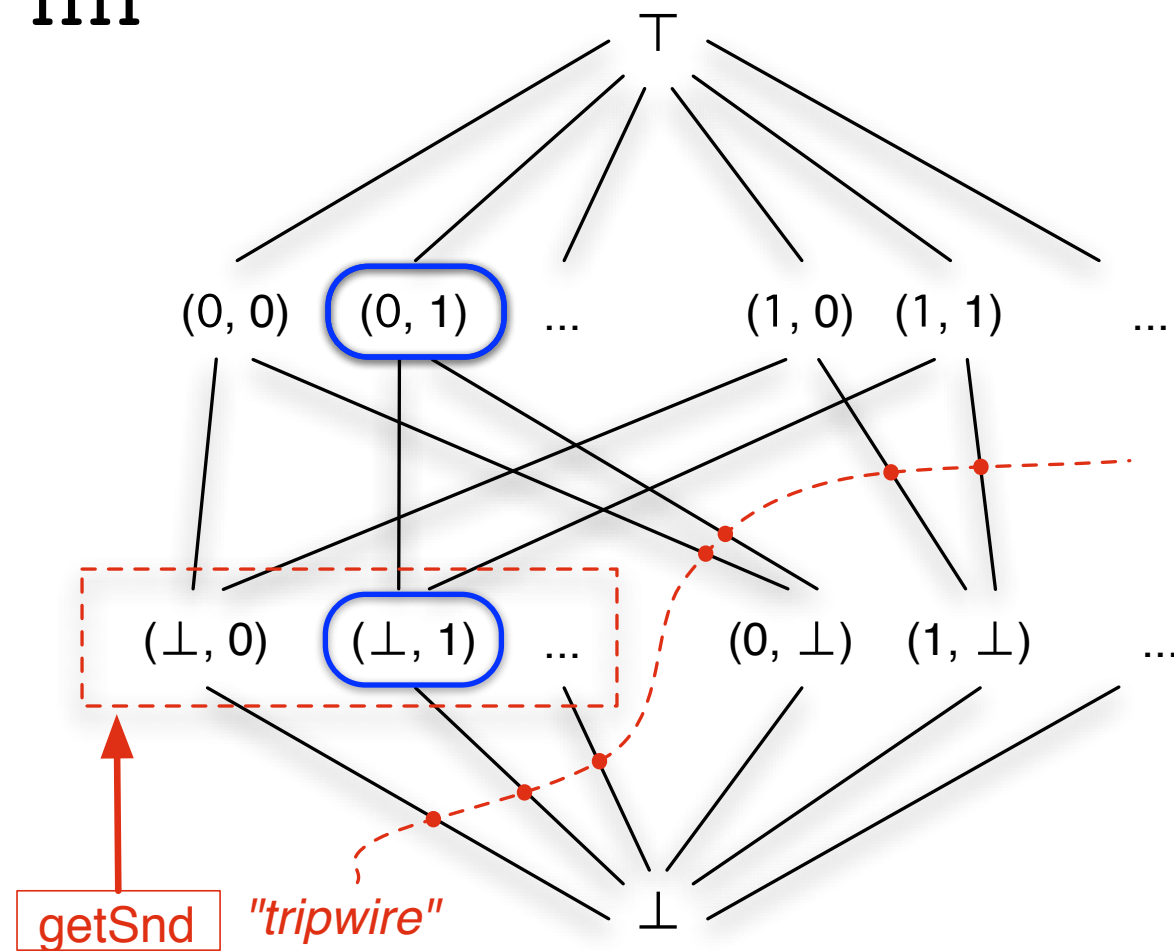


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

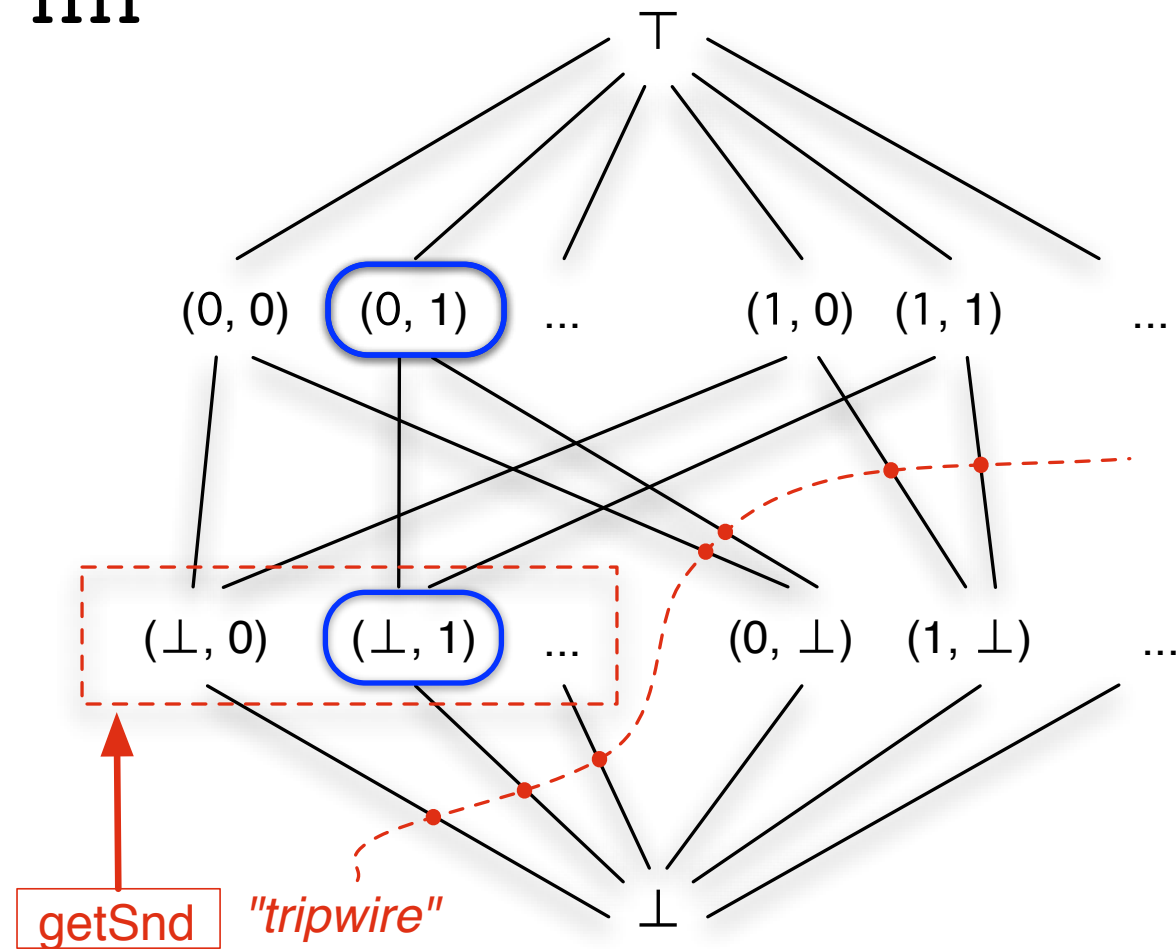


do

```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn



do

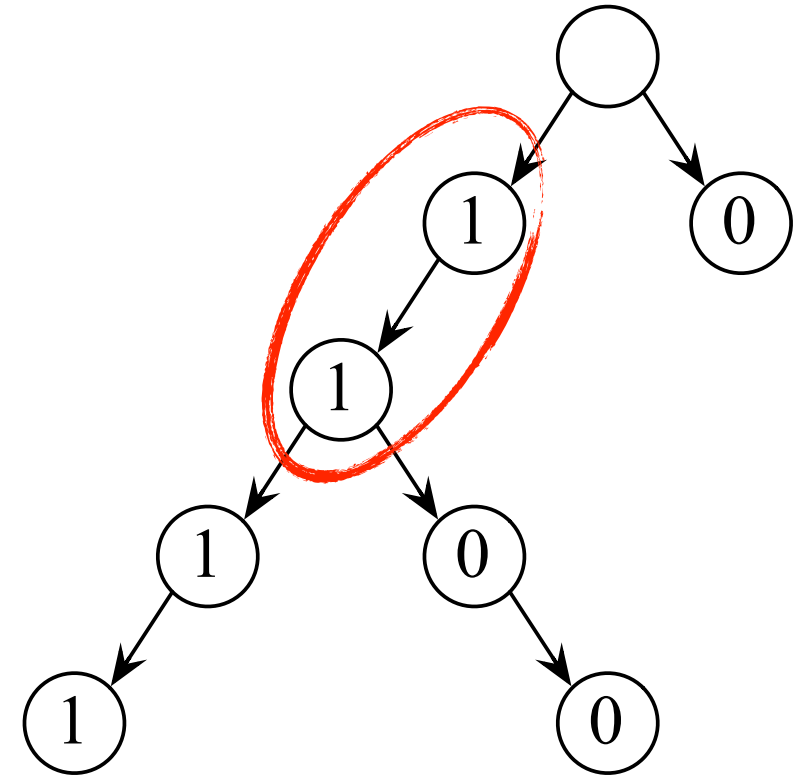
```
nn <- newPair
fork $ do putFst nn 0
fork $ do putSnd nn 1
v <- getSnd nn
return v -- returns 1
```

The threshold set must be *pairwise incompatible*

Overlapping writes are no problem

do

```
fork    $ do insert t "0"  
fork    $ do insert t "1100"  
fork    $ do insert t "1111"  
v <- get t  
return v
```



Monotonicity enables deterministic parallelism

INFORMATION PROCESSING 74 - NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type *o* (boolean, real, etc...) we could have declared a *o channel*. Then processes *f*, *g* and *h* are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (2) *f* is stated to communicate via two input lines that can carry integers, and one similar output line.

The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable on a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.

In other words, processes communicate via first-in first-out (fifo) queues. Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes. Such a style of programming is close to many systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.

What sort of things would we like to prove on a program like S? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.

The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```
Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; Logical B ;
    B := true ;
    Repeat Begin
(4)      I := if B then wait(U) else wait(V) ;
(7)      print (I) ;
(5)      send I on W ;
          B := ¬B ;
    end ;
    End ;
Process g(integer in U ; integer out V, W) ;
Begin integer I ; Logical B ;
B := true ;
Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := ¬B ;
End ;
End ;
(3) Process h(integer in U; integer out V; integer INIT);
Begin integer I ;
send INIT on V ;
Repeat Begin
    I := wait(U) ;
    send I on V ;
End ;
End ;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1)
End ;
```

Fig.1. Sample parallel program S.

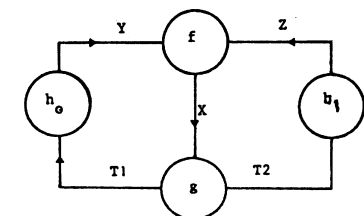


Fig.2. The schema P for the program S.

Monotonicity enables deterministic parallelism

f is *monotonic* iff, for a given \leq ,
 $x \leq y \implies f(x) \leq f(y)$

INFORMATION PROCESSING 74 - NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type *o* (boolean, real, etc...) we could have declared a *o channel*. Then processes *f*, *g* and *h* are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (2) *f* is stated to communicate via two input lines that can carry integers, and one similar output line.

The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable on a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.

In other words, processes communicate via first-in first-out (fifo) queues. Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes. Such a style of programming is close to many systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.

What sort of things would we like to prove on a program like S? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.

The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```

Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; Logical B ;
    B := true ;
    Repeat Begin
(4)      I := if B then wait(U) else wait(V) ;
(7)      print (I) ;
(5)      send I on W ;
          B := ¬B ;
    end ;
    End ;
Process g(integer in U ; integer out V, W) ;
Begin integer I ; Logical B ;
B := true ;
Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := ¬B ;
End ;
End ;
(3) Process h(integer in U; integer out V; integer INIT);
Begin integer I ;
send INIT on V ;
Repeat Begin
    I := wait(U) ;
    send I on V ;
End ;
End ;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1)
End ;
    
```

Fig.1. Sample parallel program S.

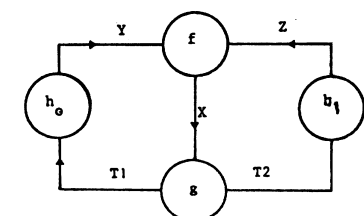


Fig.2. The schema P for the program S.

Kahn, 1974

Monotonicity enables deterministic parallelism

f is *monotonic* iff, for a given \leq ,
 $x \leq y \implies f(x) \leq f(y)$

Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this is a crucial property since it allows parallel operation : a machine need not have all of its input to start computing, since future input concerns only future output.

INFORMATION PROCESSING 74 - NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. Its deficiencies are exhibited by this theoretical analysis. We hope in this way to make a case for languages for systems programming and

```
integer channel X, Y, Z, T1, T2 ;
process f(integer in U, V; integer out W) ;
  in integer I ; logical B ;
  B := true ;
  Repeat Begin
    I := if B then wait(U) else wait(V) ;
    print (I) ;
    send I on W ;
    B := not B ;
  end ;
;
process g(integer in U; integer out V, W) ;
  in integer I ; logical B ;
  B := true ;
  Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := not B ;
  end ;
;
process h(integer in U; integer out V; integer INIT) ;
  in integer I ;
  and INIT on V ;
  Repeat Begin
    I := wait(U) ;
    send I on V ;
  end ;
;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1)
End ;
```

Fig.1. Sample parallel program S.

until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.
In other words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* indicates the parallel composition of the processes.

2. PARALLEL COMPUTATION.
Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

2. PARALLEL COMPUTATION.
Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

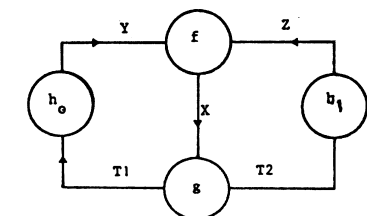


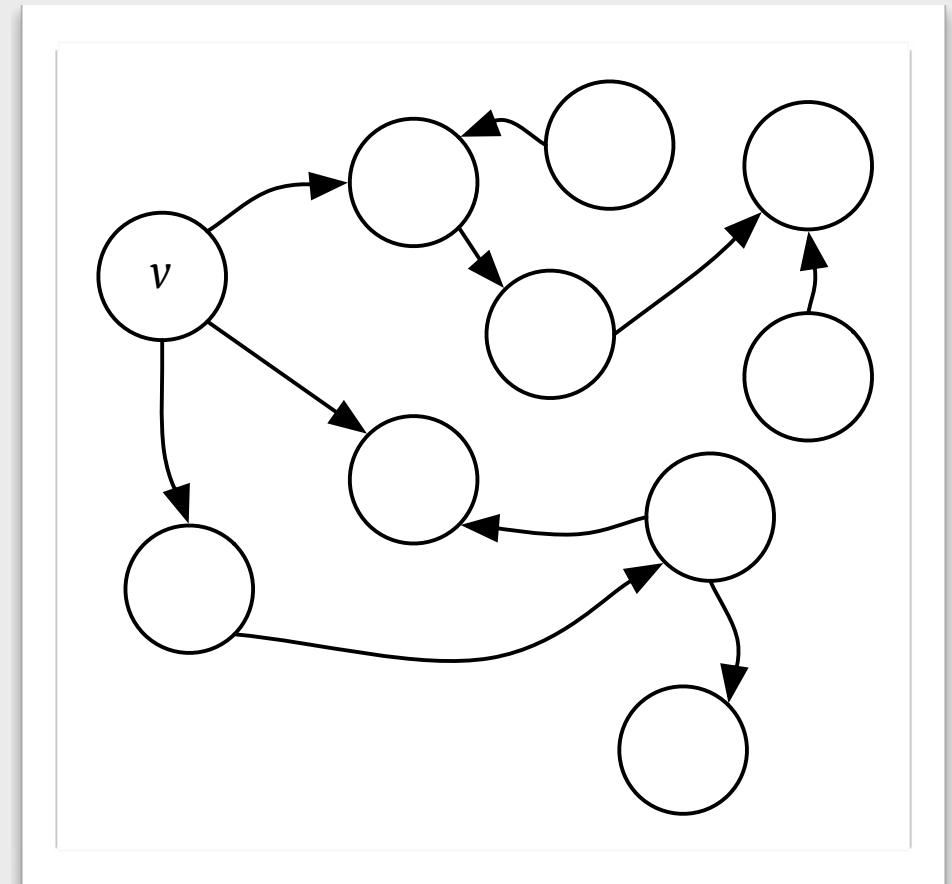
Fig.2. The schema P for the program S.

The kind of parallel programming we have studied in this paper is severely limited : it can produce only determinate programs.

Challenge problem

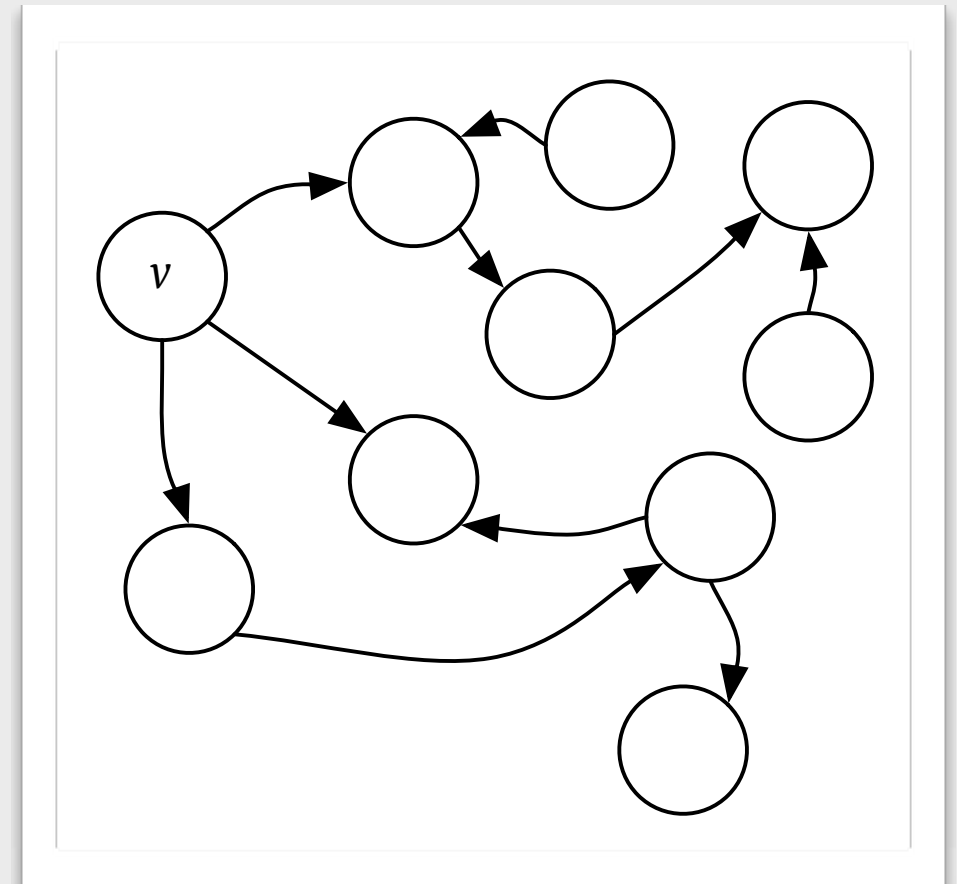
In a directed graph:

- find the connected component of all nodes within k hops of a vertex v
- and compute a function *analyze* over each vertex in that component
- **making the set of results available asynchronously to other computations**



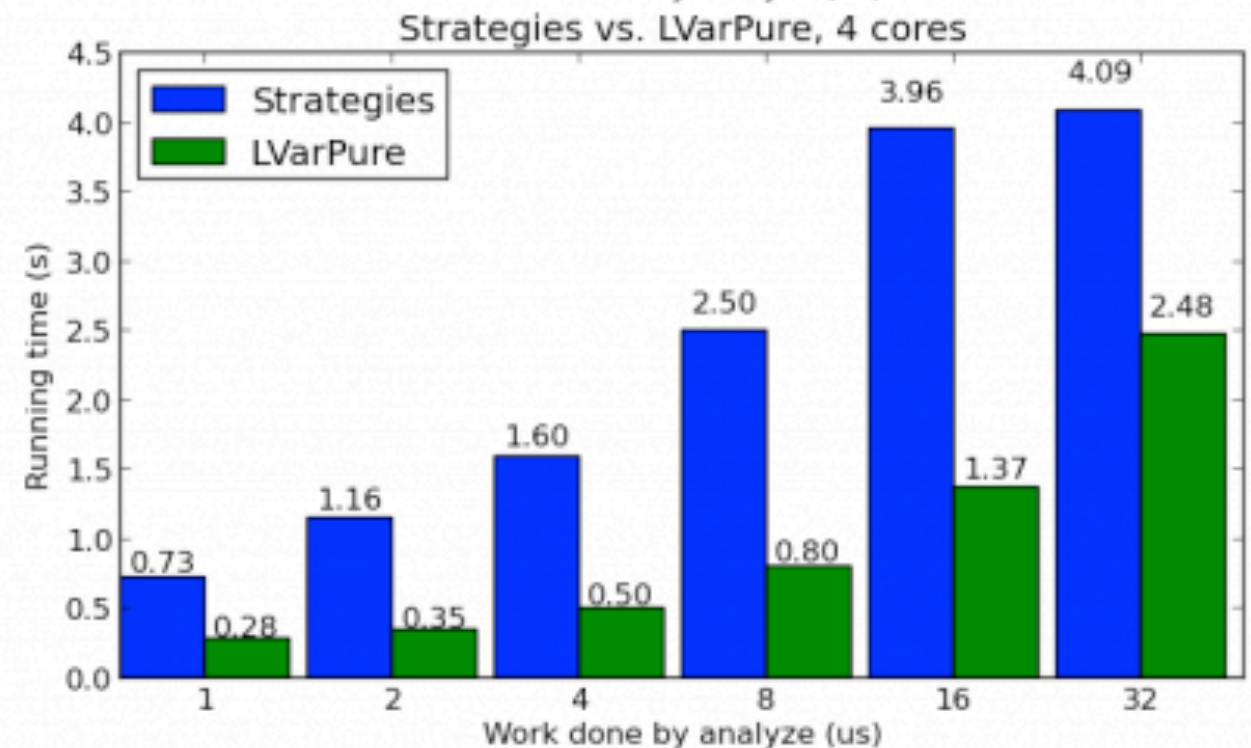
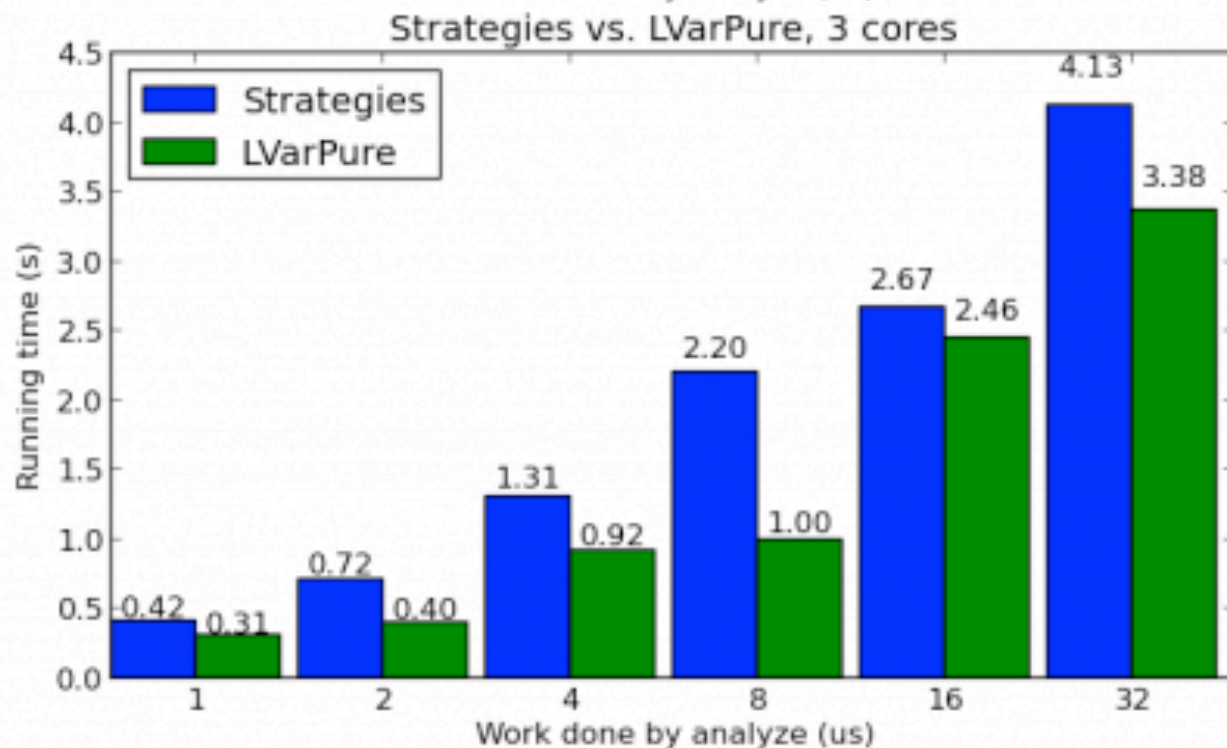
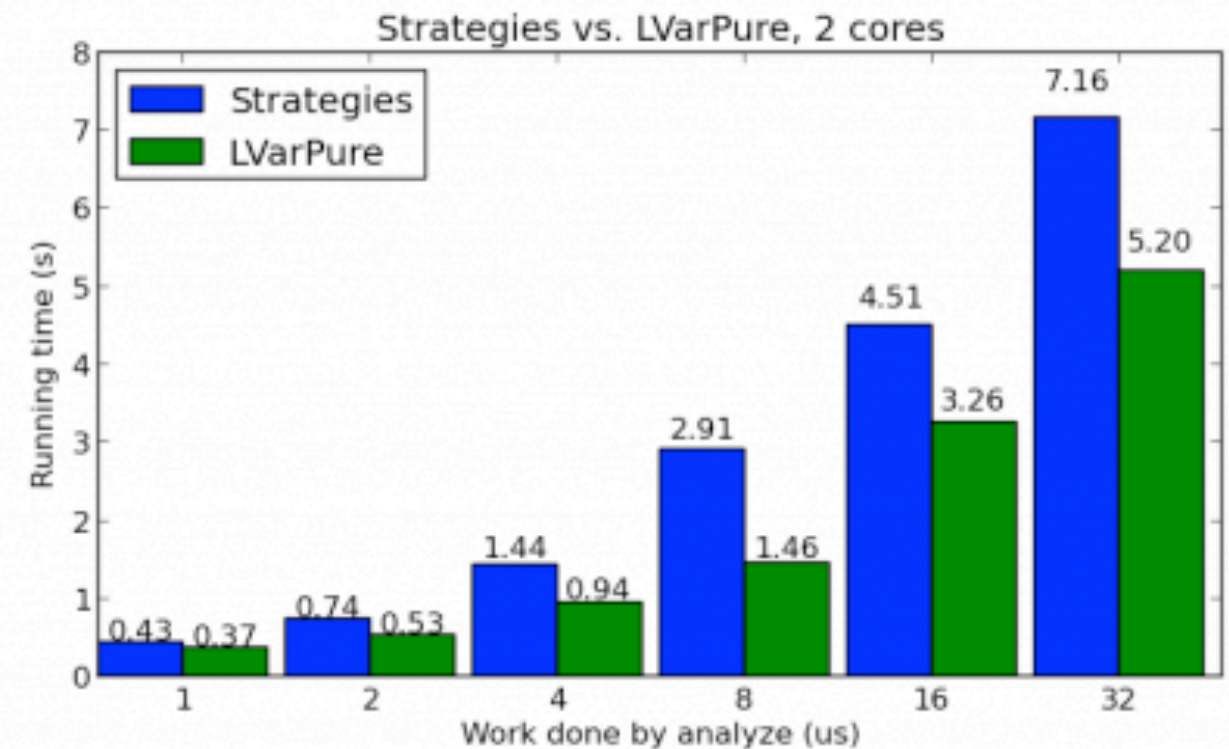
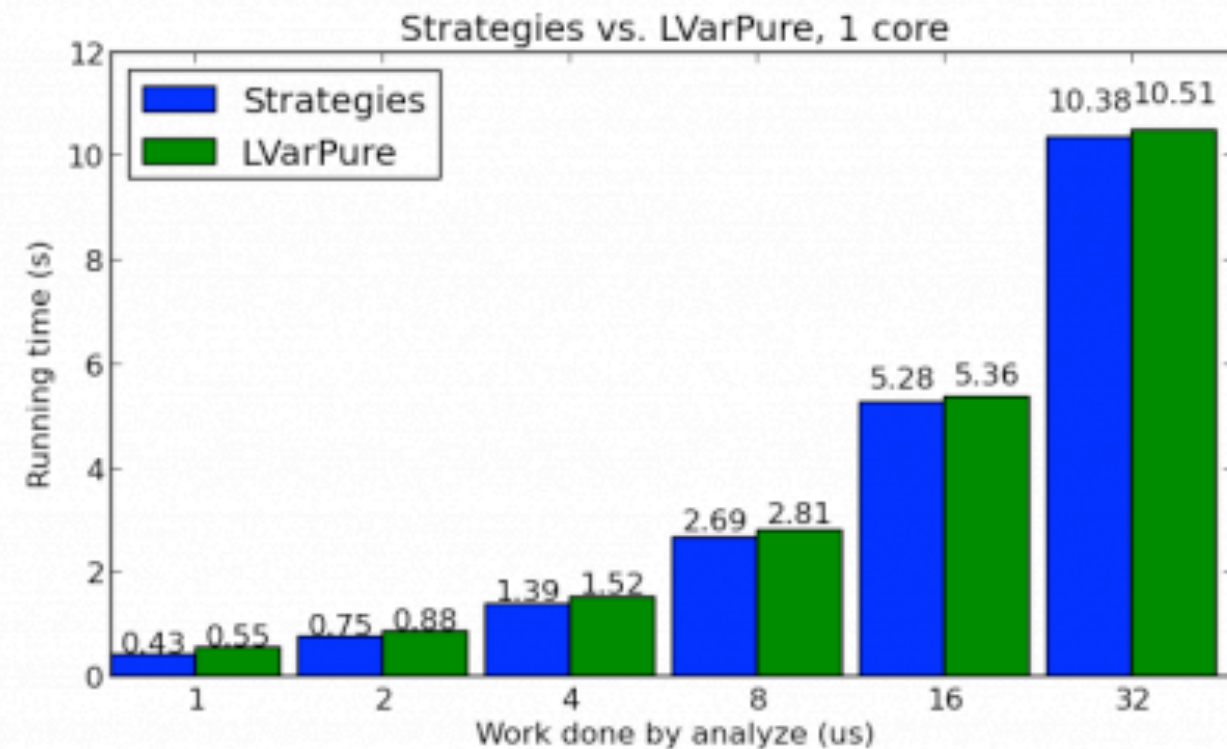
Challenge problem

- We compared two implementations:
 - `Control.Parallel.Strategies`
 - Our prototype LVar library
(tracking visited nodes in an LVar)
- Level-sync breadth-first traversal, $k = 10$
- Random graph; 320K edges; 40K nodes
- Varying:
 - number of cores
 - amount of work done by *analyze*



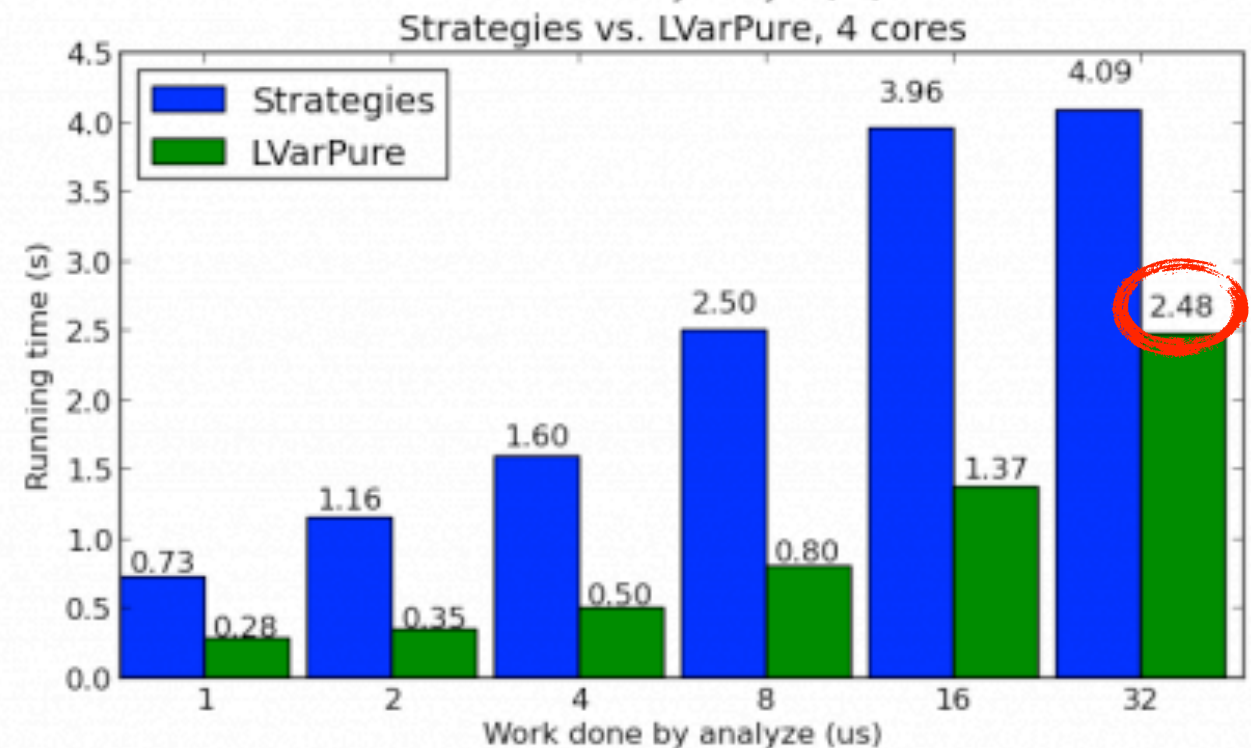
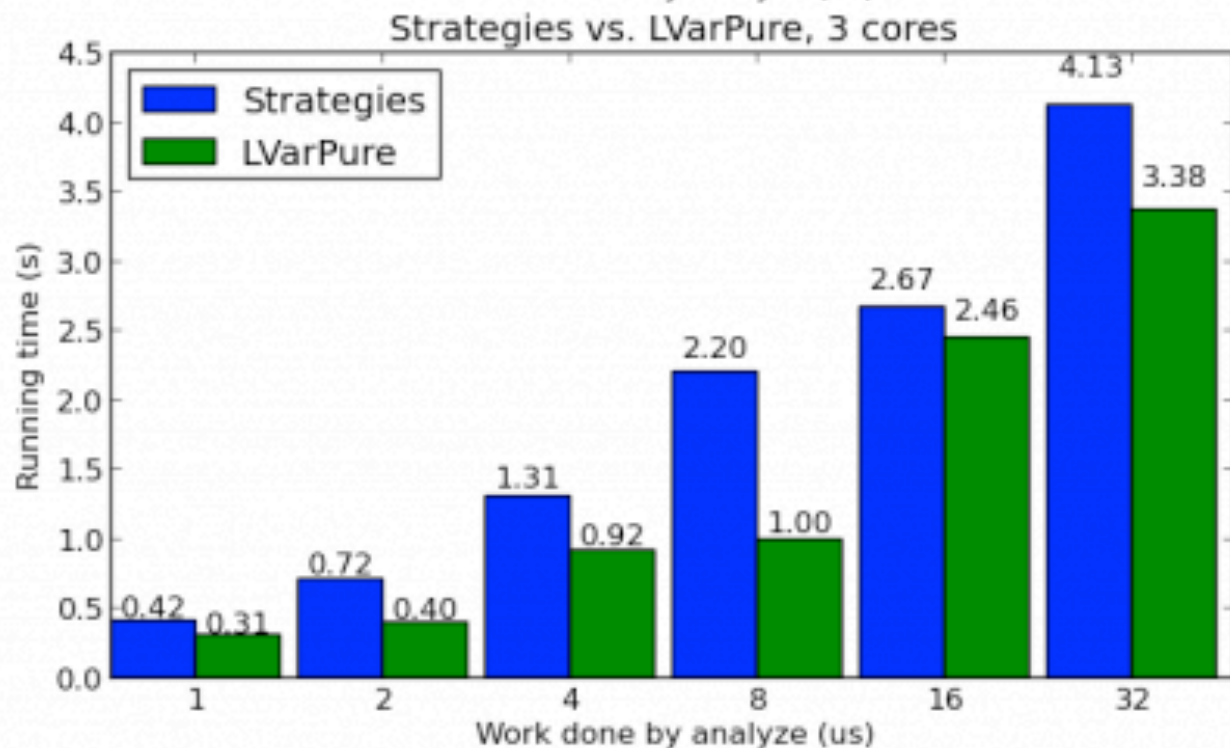
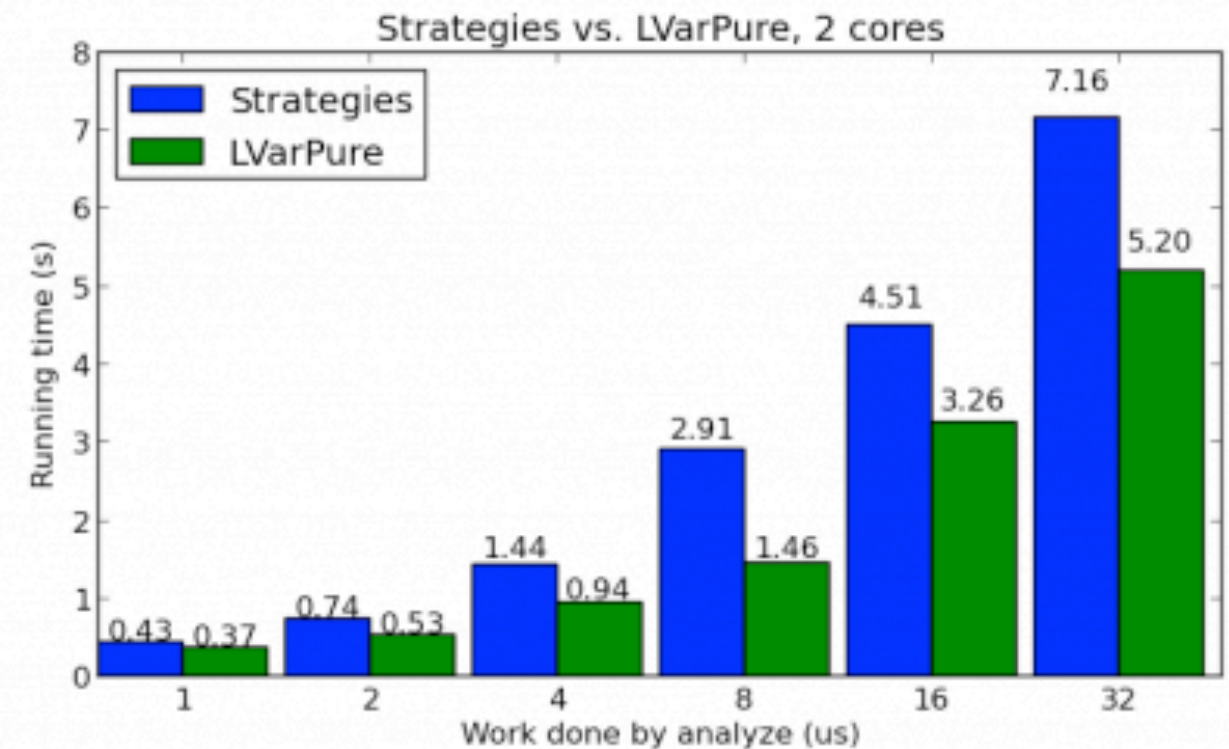
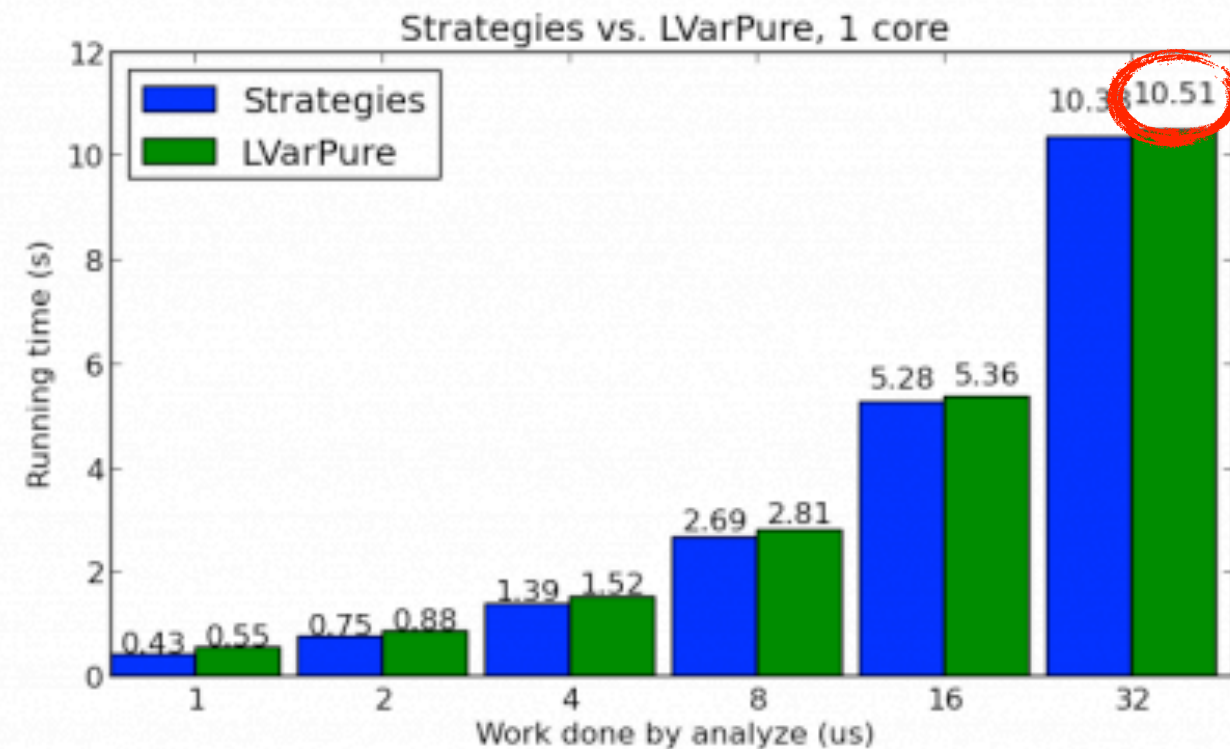
Challenge problem: Strategies vs. LVars

(lower is better)



Challenge problem: Strategies vs. LVars

(lower is better)



Challenge problem: Strategies vs. LVars

Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this is a crucial property since it allows parallel operation : a machine need not have all of its input to start computing, since future input concerns only future output.

- Average time from start of program to first invocation of *analyze*:
 - Strategies version: 64.64 ms
 - LVar version: **0.18 ms**

Lots more stuff in the paper and TR

- λ_{LVar} , a core calculus for LVars
- Proof of determinism for λ_{LVar}
 - Independence Lemma is a frame property!
- Details on our Haskell library
- Proposed extension: *quasi-determinism*
 - *Ask me about our new paper on this!*
 - Cool applications: k -CFA, phylogenetics, ...



Thank you!

Email: lkuper@cs.indiana.edu

Research blog: composition.al

Project repo: github.com/iu-parfunc/lvars

Code from this talk: github.com/lkuper/lvar-examples