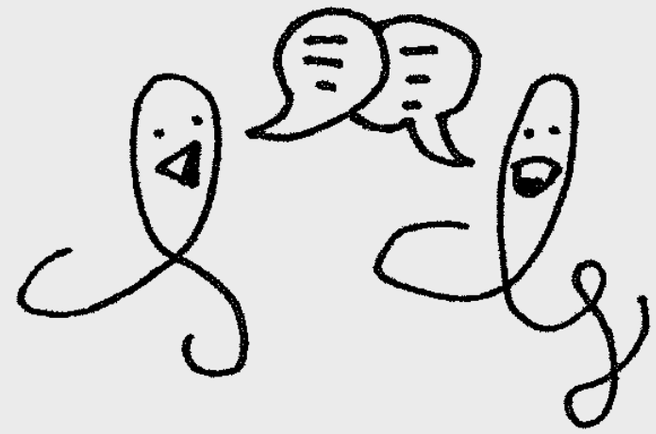
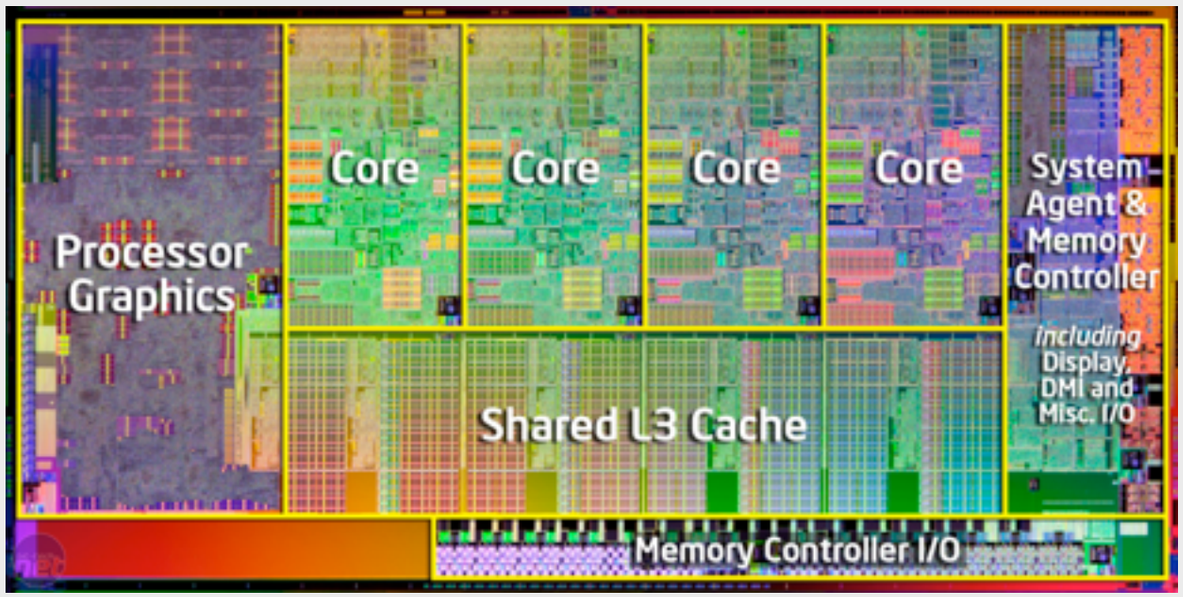
Lattice-based Data Structures *for* Deterministic Parallel *and* Distributed Programming



Lindsey Kuper

September 8, 2014

(with illustrations by Jason Reed)



Parallel systems

Distributed systems

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures

are a general and practical unifying abstraction for deterministic and quasi-deterministic parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures

are a general and practical unifying abstraction

for deterministic and quasi-deterministic

parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

spoiler:
deterministic
modulo exceptions

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

My thesis:

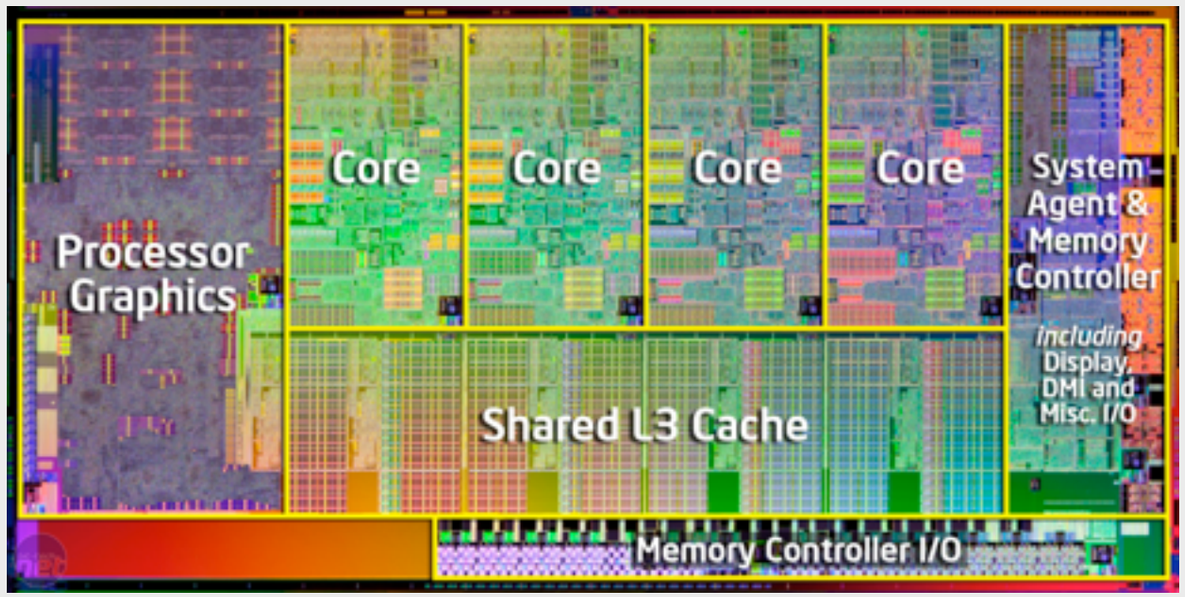
aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

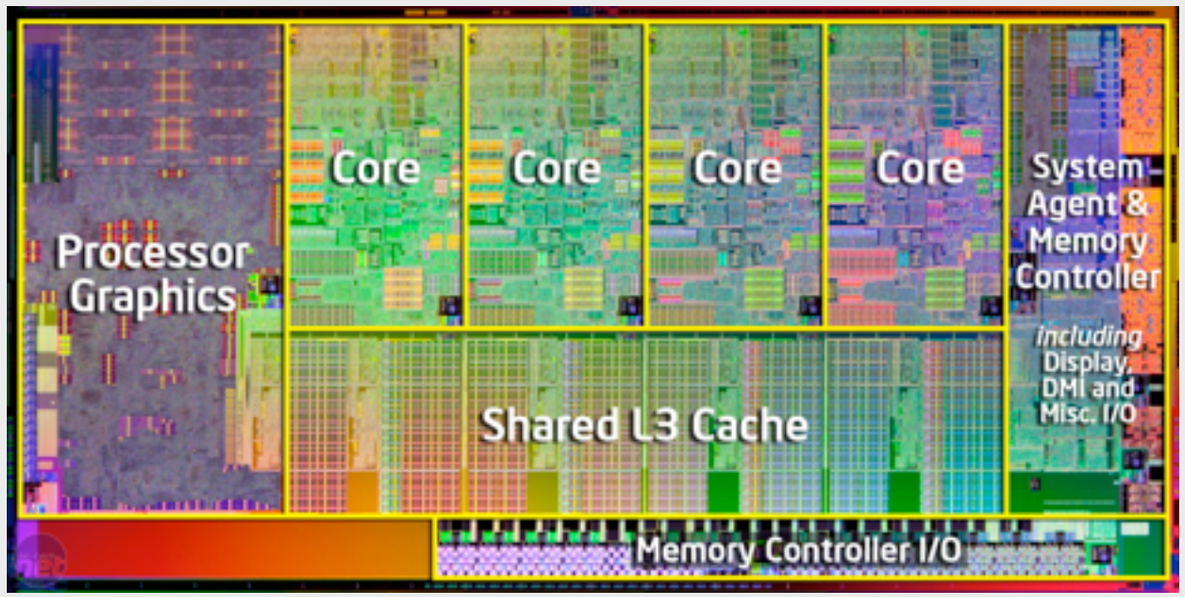
My thesis:

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.



Deterministic Parallel Programming



(observably)

Deterministic Parallel Programming

```haskell
data Item = Book | Shoes | ...
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)] (Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoe
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(B
ook,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
[Shoes,1)] (Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)] [(Shoes,1)] (Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

```
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(B
ook,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)] [(Book,1)] [(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)] [(Book,1)] [(Book,1),(Shoes,1)
[(Shoes,1)] [(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

if we want determinism,
we have to learn to *share nicely*

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
                (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
                    (\m -> (insert Book 1 m, ())))
       a2 <- async (atomicModifyIORef cart
                    (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
                    (\m -> (insert Book 1 m, ())))
       a2 <- async (atomicModifyIORef cart
                    (\m -> (insert Shoes 1 m, ())))
       res <- async (do waitBoth a1 a2
       wait res            readIORef cart)
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
                    (\m -> (insert Book 1 m, ())))
       a2 <- async (atomicModifyIORef cart
                    (\m -> (insert Shoes 1 m, ())))
       res <- async (do waitBoth a1 a2
                        readIORef cart)
       wait res
```

```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic

```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic...now

```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic...now...we hope

```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic...now...we hope

```haskell
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

deterministic by construction

[FHPC '13, POPL '14]

The deterministic by construction parallel programming landscape:

The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

```
f (g x)
(h y)
```

The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

```
f (g x)
(h y)
```

The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
(h y)
```

The deterministic by construction parallel programming landscape:

Kahn process networks

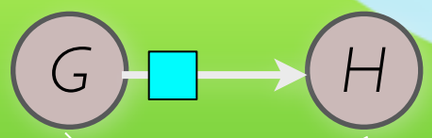
single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
(h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

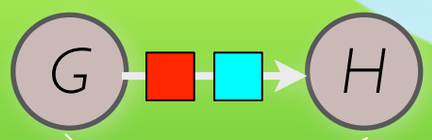
single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
(h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

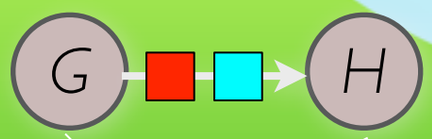
single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
(h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

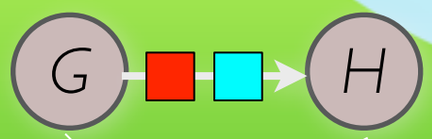
single-assignment *(IVars, CnC, ...)*

imperative disjoint

λ-calculus *(& array langs, ...)*

```
f (g x)
(h y)
```

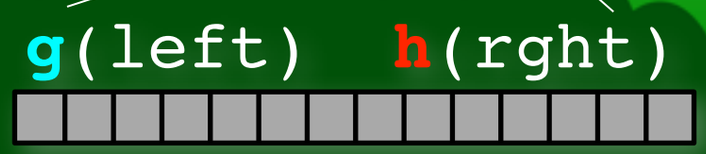
The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment *(IVars, CnC, ...)*

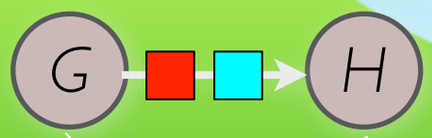
imperative disjoint



λ-calculus *(& array langs, ...)*

```
f (g x)
  (h y)
```

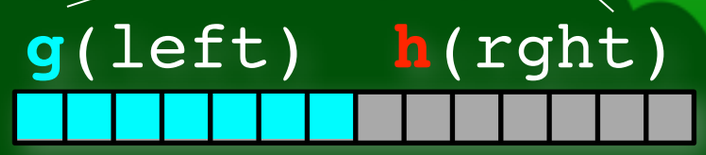
The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment *(IVars, CnC, ...)*

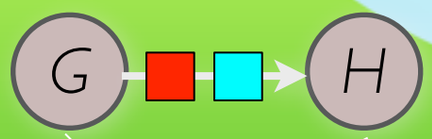
imperative disjoint



λ-calculus *(& array langs, ...)*

```
f (g x)
  (h y)
```

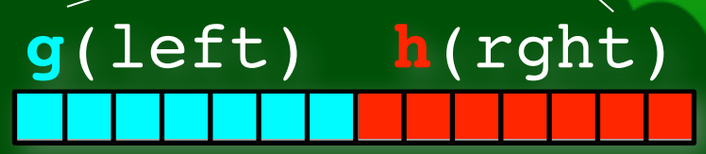
The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment *(IVars, CnC, ...)*

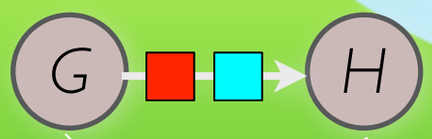
imperative disjoint



λ-calculus *(& array langs, ...)*

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

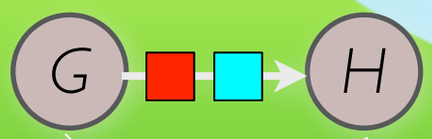
single-assignment *(IVars, CnC, ...)*

imperative disjoint *(DPJ, ...)*

λ-calculus *(& array langs, ...)*

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment *(IVars, CnC, ...)*

imperative disjoint *(DPJ, ...)*

λ-calculus *(& array langs, ...)*

```
f (g x)
  (h y)
```

Can we *generalize* and *unify* these points in the space?

aka "LVars"

Lattice-based data structures

are a general and practical unifying abstraction for deterministic and quasi-deterministic parallel and distributed programming.

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

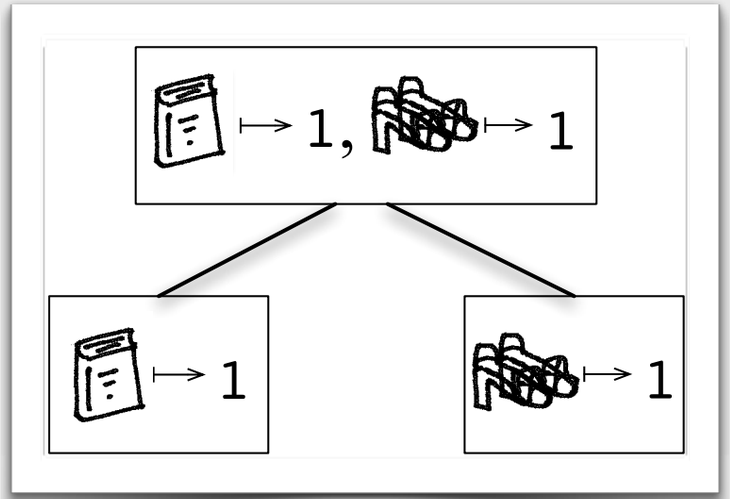
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]

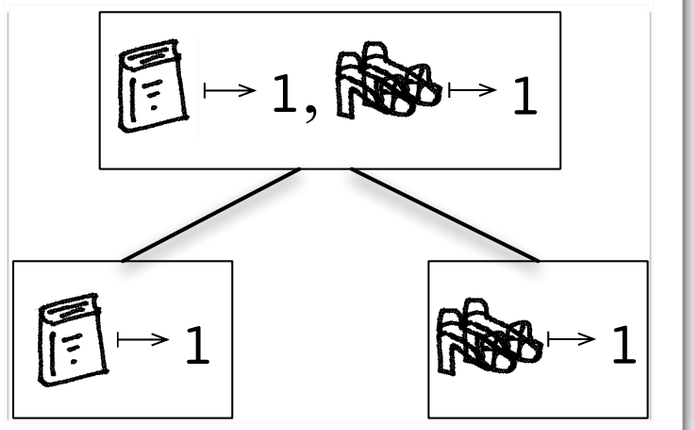
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]

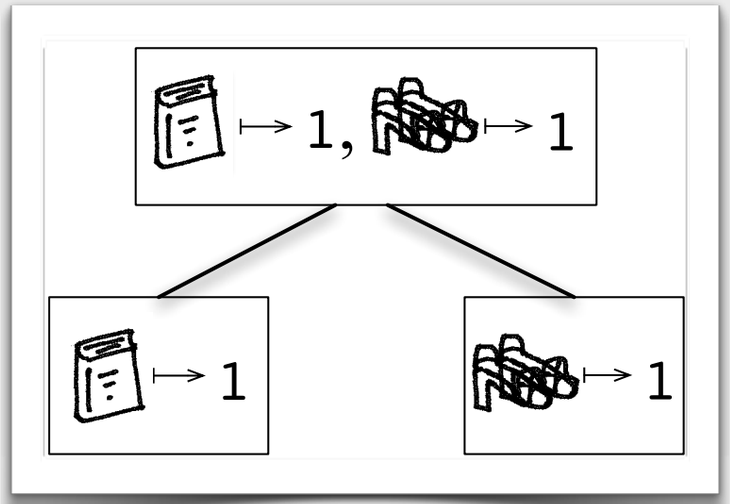
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: multiple *commutative and inflationary* writes, blocking *threshold* reads

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

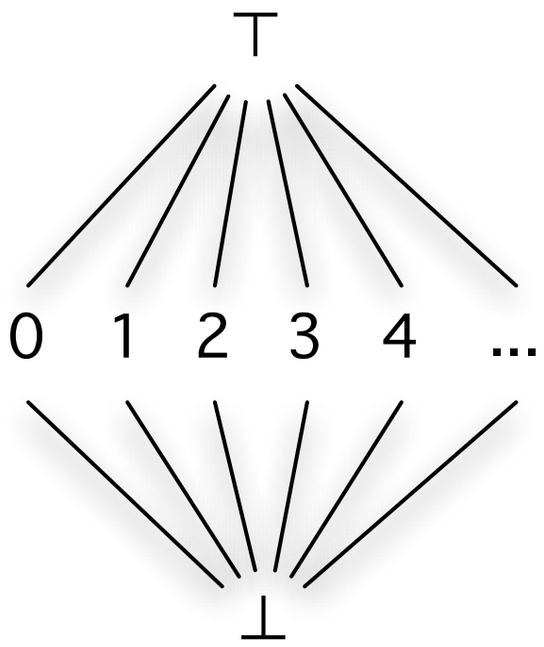


IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: multiple *commutative and inflationary* writes, blocking *threshold* reads

* actually a bounded join-semilattice

num



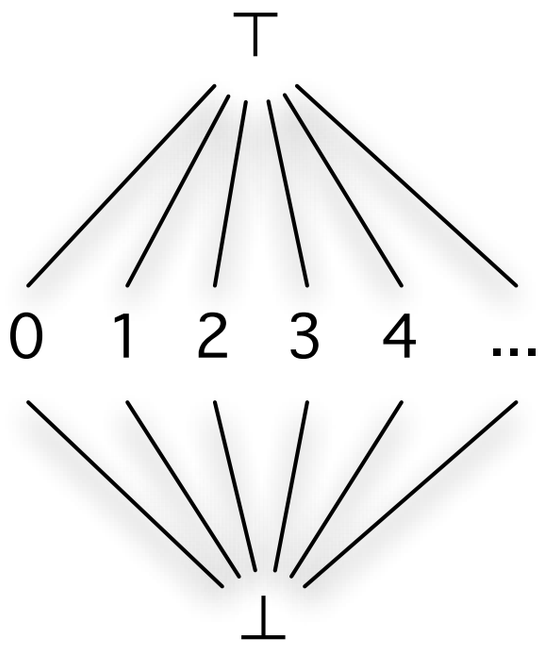
Raises an error, since $3 \sqcup 4 = \top$

```
do
   fork (put num 3)
   fork (put num 4)
```

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

```
do
   fork (put num 4)
   fork (put num 4)
```

num



Raises an error, since 3 ⊔ 4 = ⊤

```
do
   fork (put num 3)
   fork (put num 4)
```

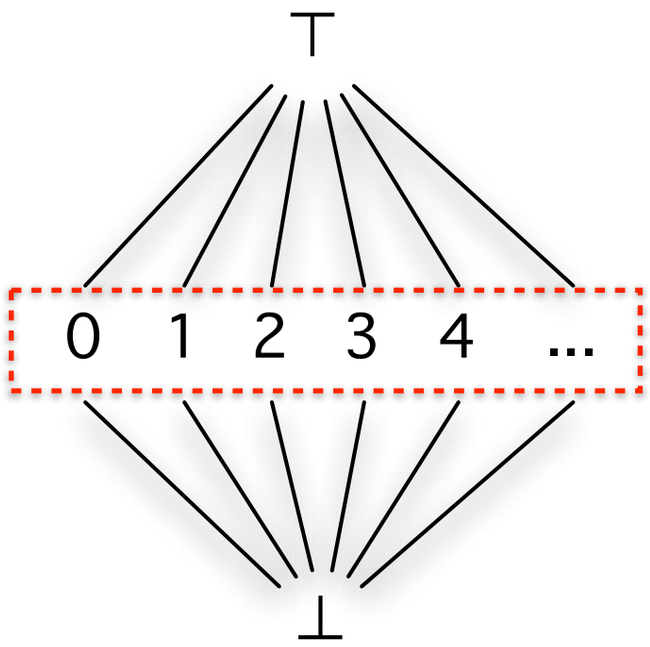
Works fine, since 4 ⊔ 4 = 4

```
do
   fork (put num 4)
   fork (put num 4)
```

`get` blocks until threshold is reached

```
do
   fork (put num 4)
   get num
```

num



Raises an error, since 3 ⊔ 4 = ⊤

```
do
   fork (put num 3)
   fork (put num 4)
```

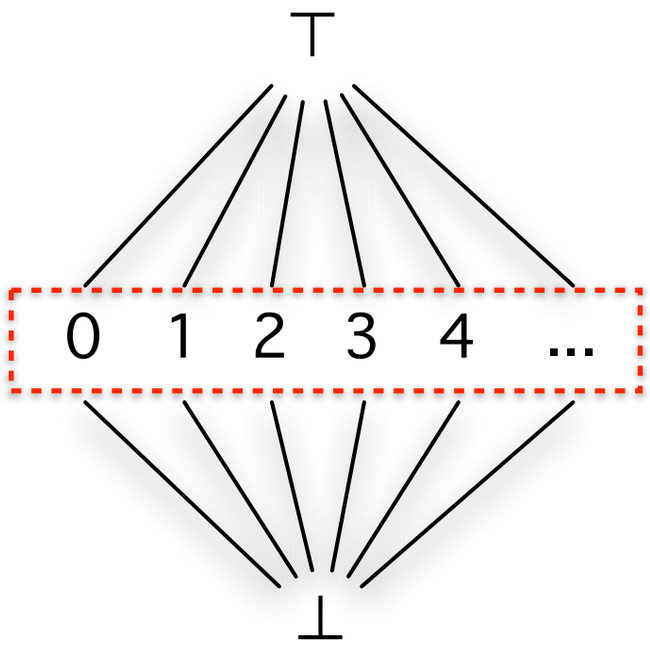
Works fine, since 4 ⊔ 4 = 4

```
do
   fork (put num 4)
   fork (put num 4)
```

`get` blocks until threshold is reached

```
do
   fork (put num 4)
   get num
```

num



threshold set elements must be *pairwise incompatible*

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

```
do
   fork (put num 3)
   fork (put num 4)
```

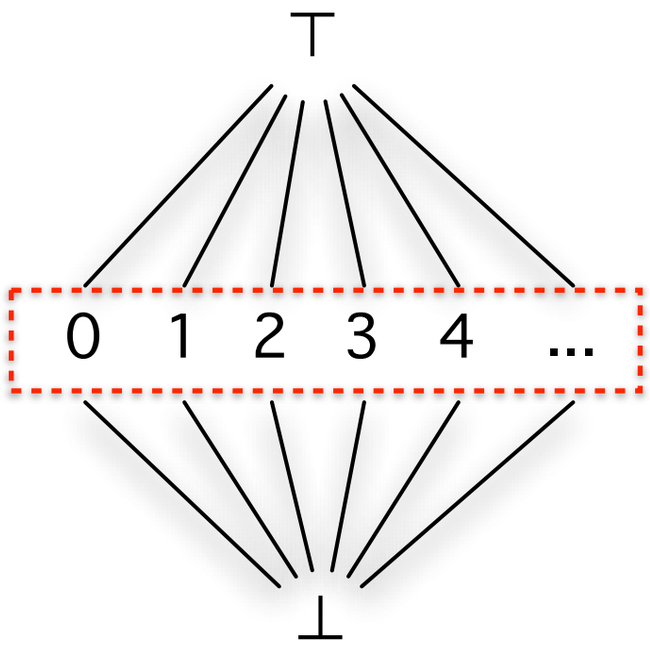
Works fine, since $4 \sqcup 4 = 4$

```
do
   fork (put num 4)
   fork (put num 4)
```

`get` blocks until threshold is reached

```
do
   fork (put num 4)
   get num
```

num



Data structure author's obligation:

threshold set elements must be *pairwise incompatible*

Raises an error, since 3 ⊔ 4 = ⊤

```
do
   fork (put num 3)
   fork (put num 4)
```

Works fine, since 4 ⊔ 4 = 4

```
do
   fork (put num 4)
   fork (put num 4)
```

`get` blocks until threshold is reached

```
do
   fork (put num 4)
   get num
```

counter

Works fine, since **`incr`**s commute

```
do
   fork (incr1 counter)
   fork (incr42 counter)
```

counter

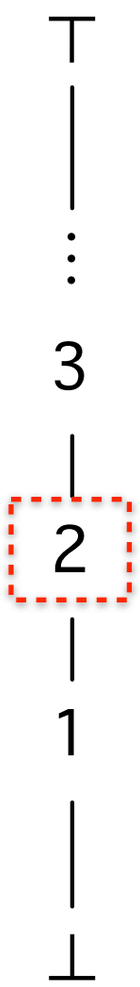
Works fine, since `incr`s commute

```
do
  fork (incr1 counter)
  fork (incr42 counter)
```

`get` blocks until threshold is reached

```
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter 2
```

counter

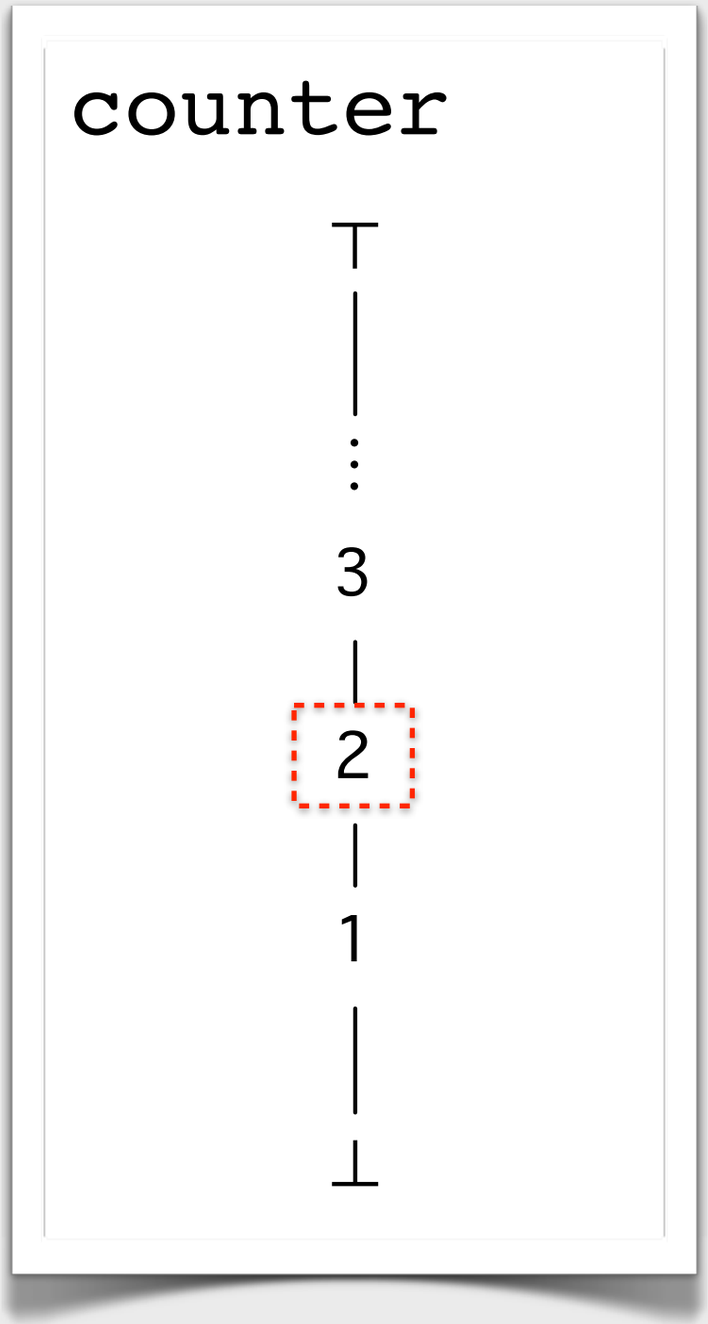


Works fine, since `incr`s commute

```
do
   fork (incr1 counter)
   fork (incr42 counter)
```

`get` blocks until threshold is reached

```
do
   fork (incr1 counter)
   fork (incr42 counter)
   get counter 2
```



Works fine, since **`incr`**s commute

```
do
   fork (incr1 counter)
   fork (incr42 counter)
```

`get` blocks until threshold is reached

```
do
   fork (incr1 counter)
   fork (incr42 counter)
   get counter 2
```

unblocks when **`counter`** is at least **2**

exact contents of **`counter`** not observable

✖ Can’t see the exact, complete contents of an LVar

- ✗ Can't see the exact, complete contents of an LVar
- ✗ Can't iterate over the contents of an LVar

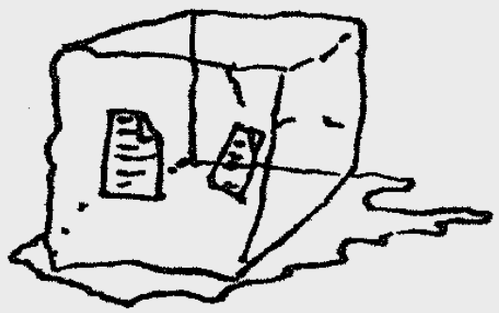
- ✗ Can't see the exact, complete contents of an LVar
- ✗ Can't iterate over the contents of an LVar
- ✗ Can't determine if something *isn't* in the LVar

- ✗ Can't see the exact, complete contents of an LVar
- ✗ Can't iterate over the contents of an LVar
- ✗ Can't determine if something *isn't* in the LVar
- ✗ Can't react to writes that we weren't expecting

- ✓ Can see the exact, complete contents of an LVar
- ✓ Can iterate over the contents of an LVar
- ✓ Can determine if something *isn't* in the LVar
- ✓ Can react to writes that we weren't expecting

- ✓ Can see the exact, complete contents of an LVar
- ✓ Can iterate over the contents of an LVar
- ✓ Can determine if something *isn't* in the LVar
- ✓ Can react to writes that we weren't expecting

} handlers, quiescence, freezing

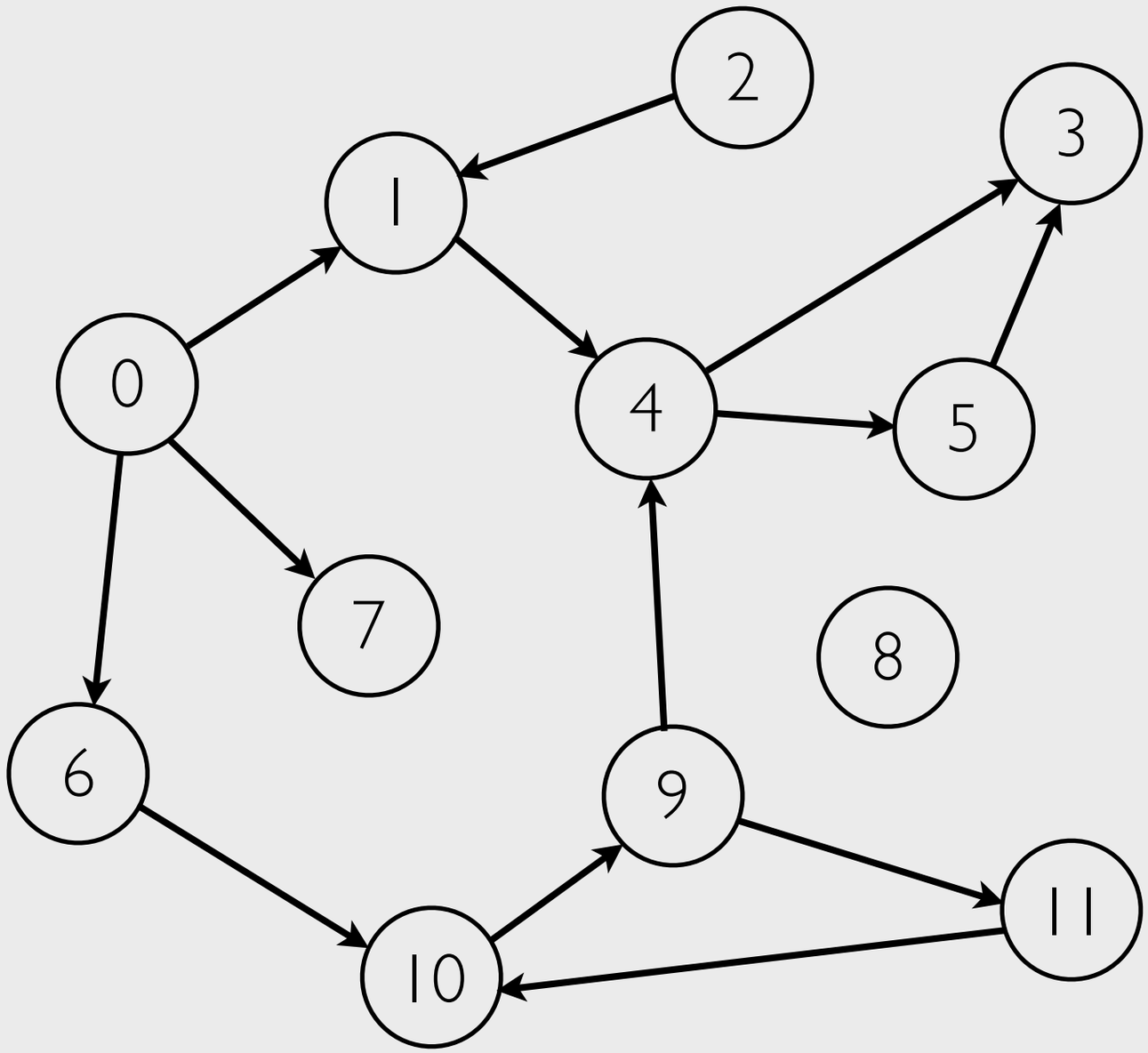


*fr*eeze after writing
(or before reading)

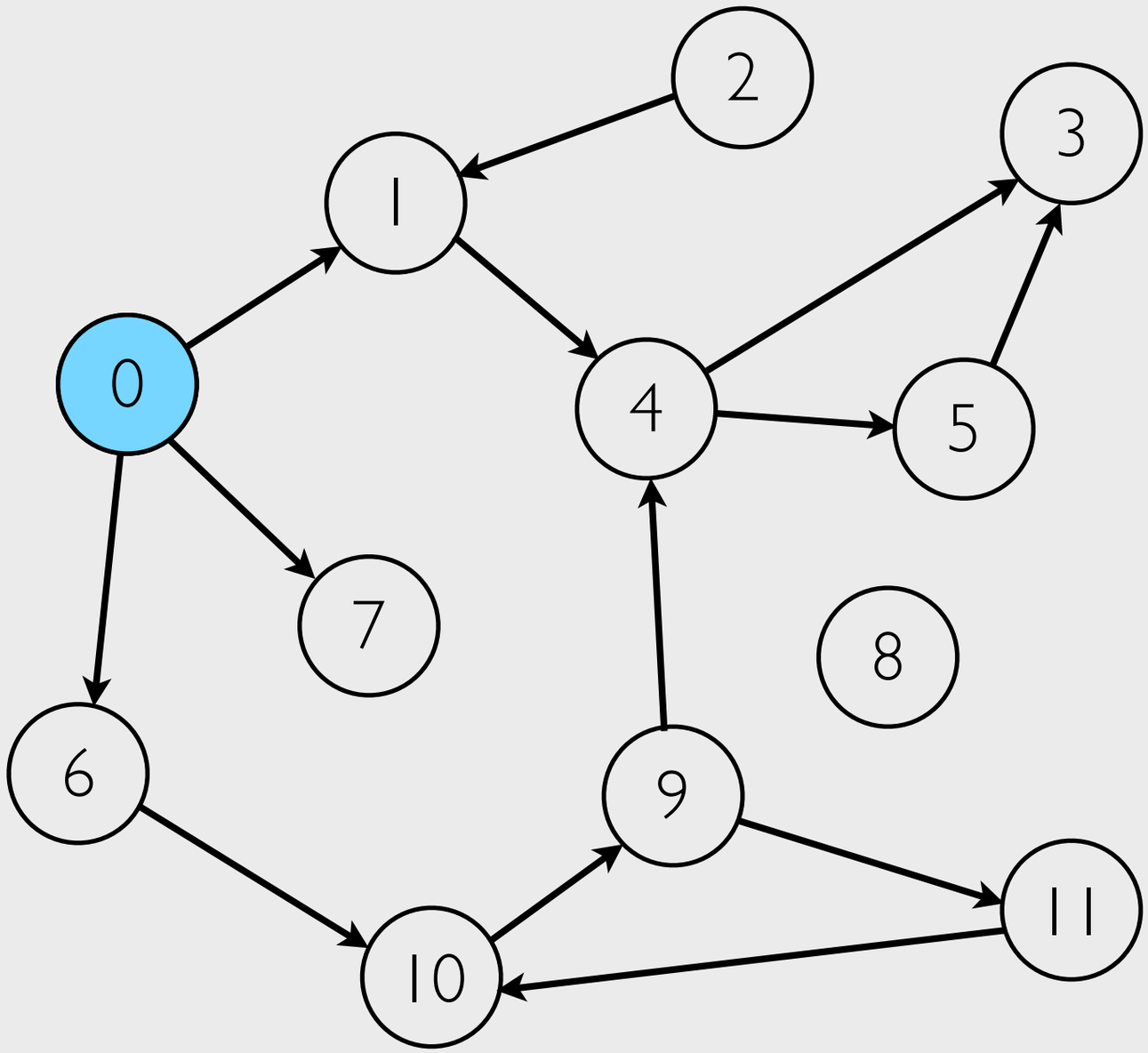
aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

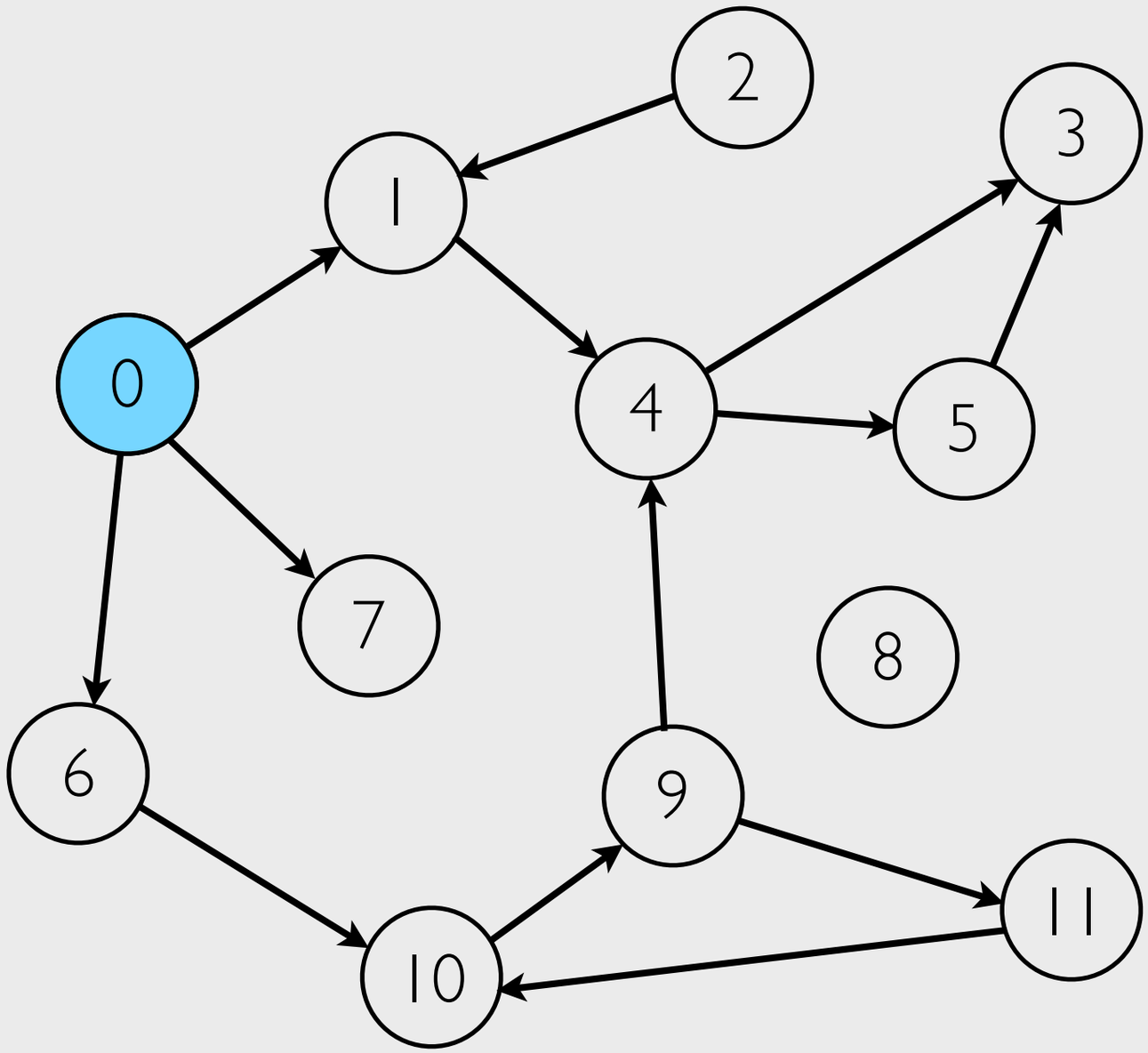
spoiler:
deterministic
modulo exceptions



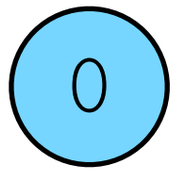
seen nodes

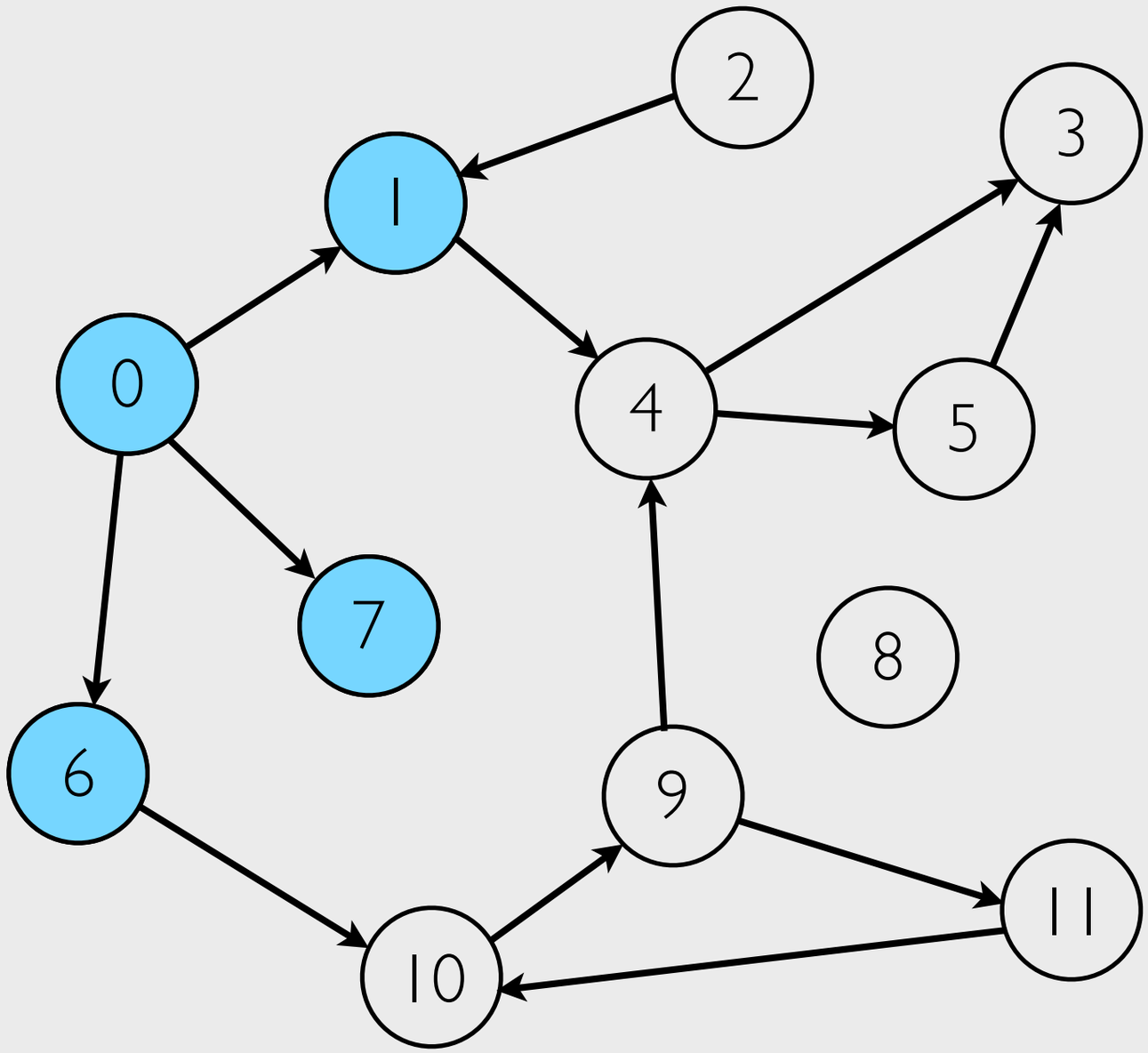


seen nodes



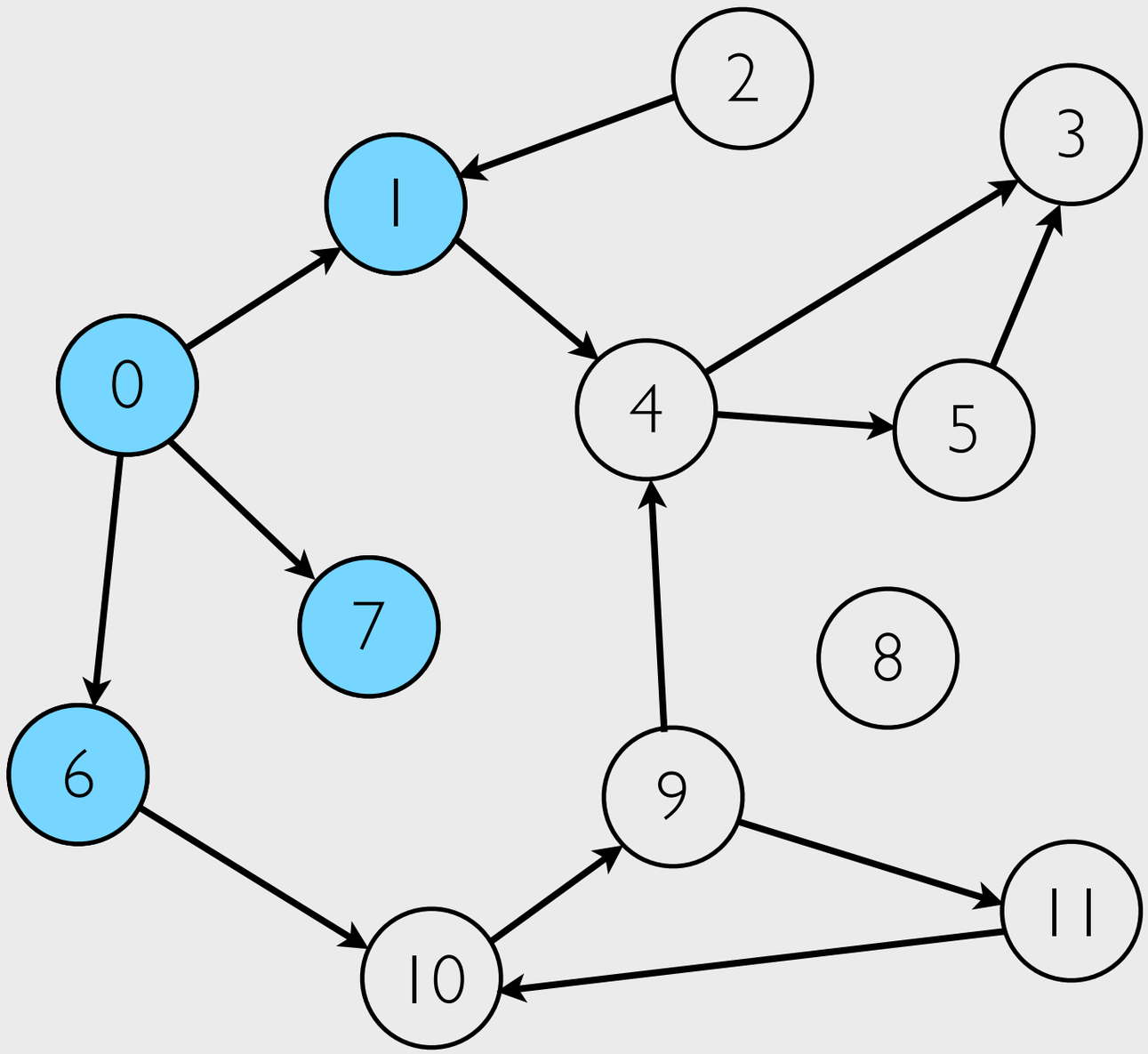
seen nodes



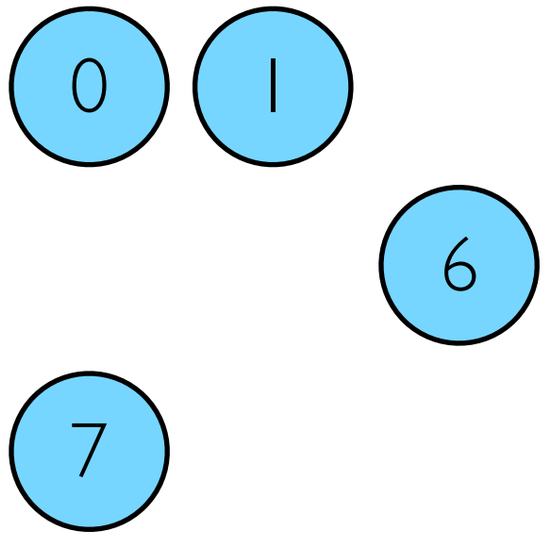


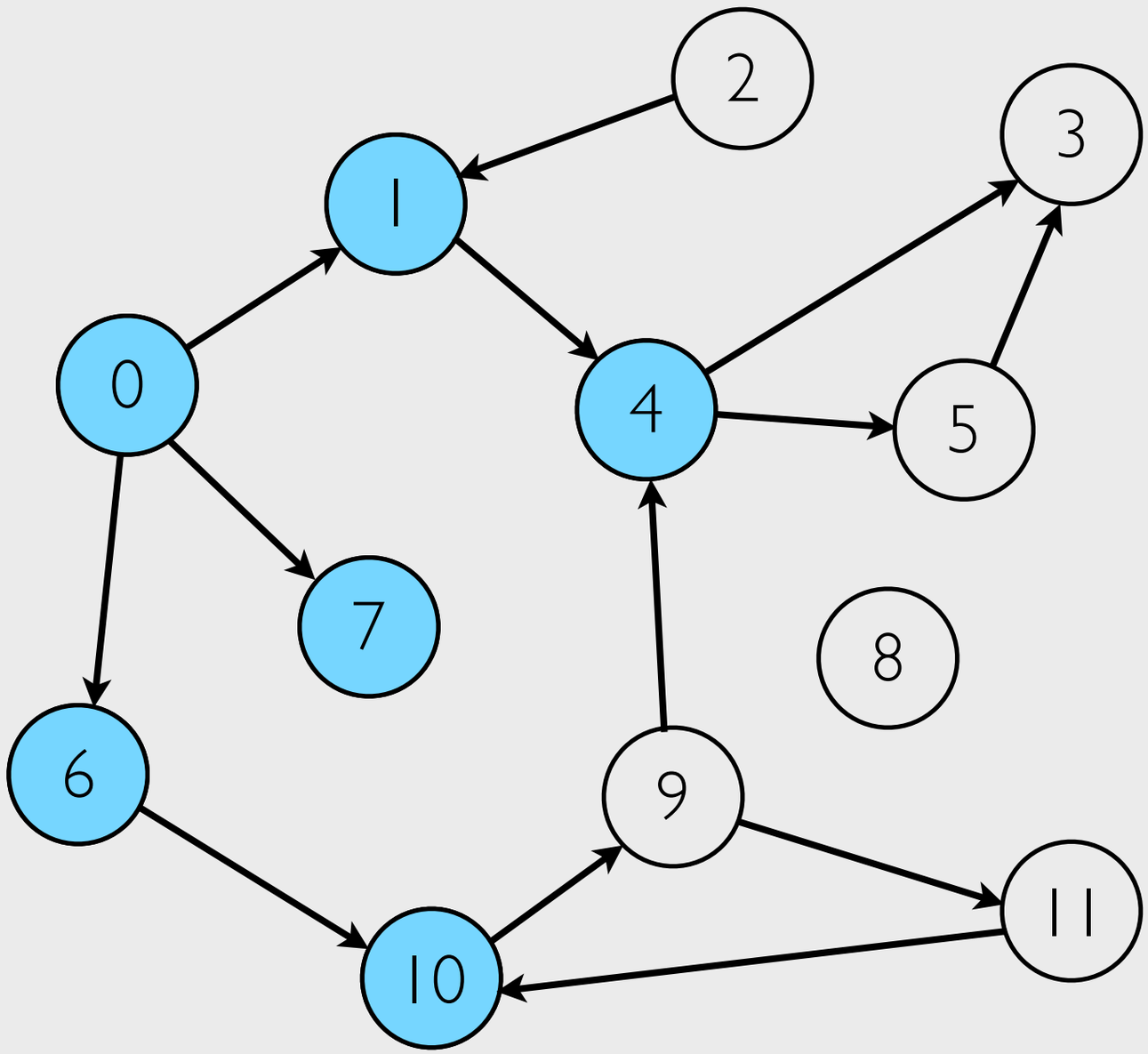
seen nodes

0

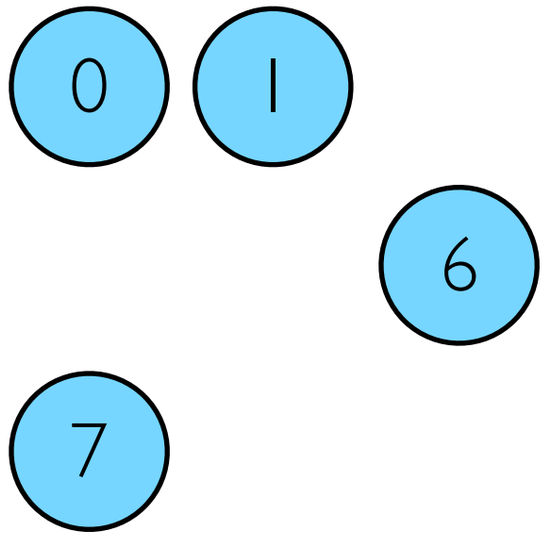


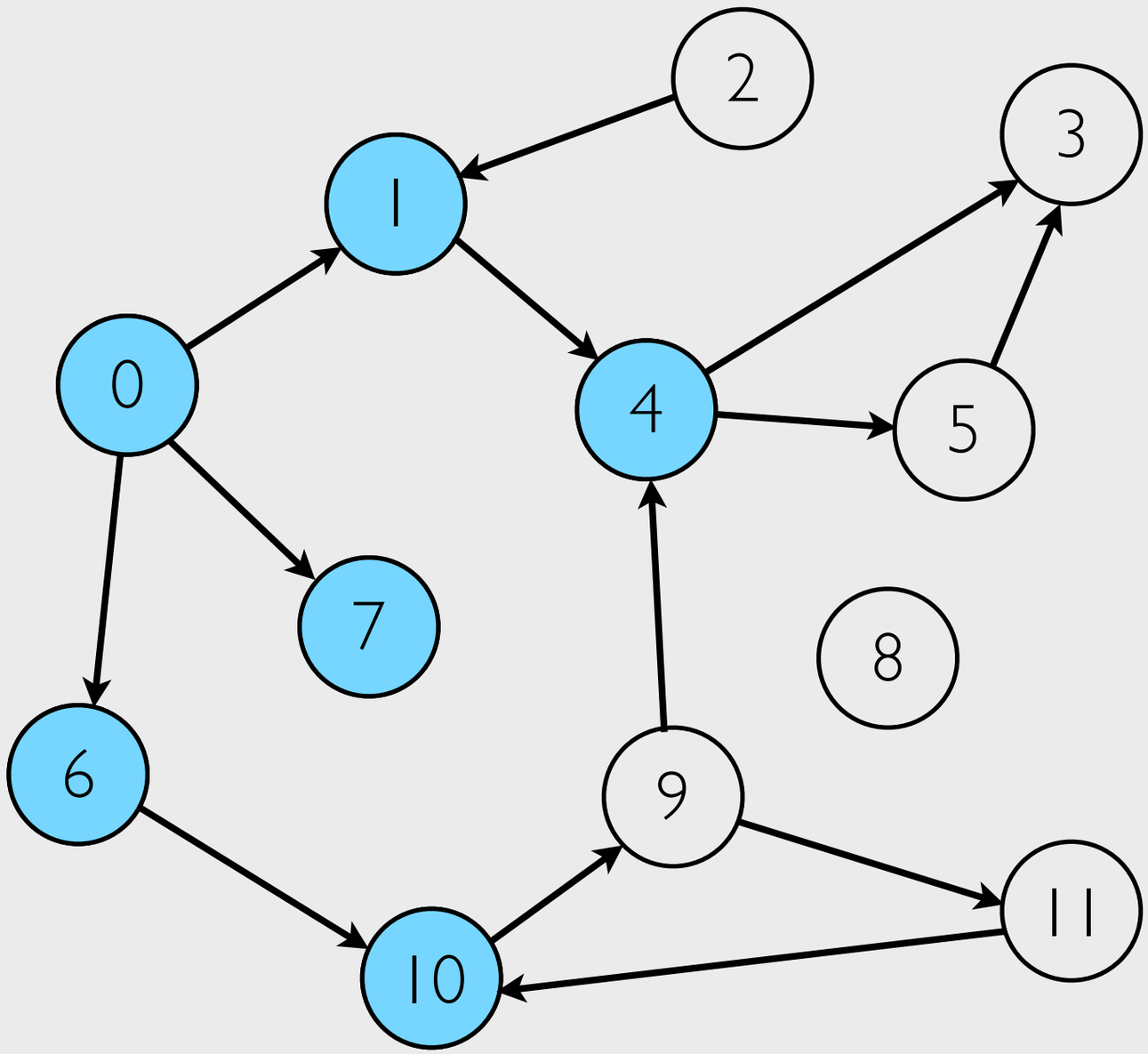
seen nodes



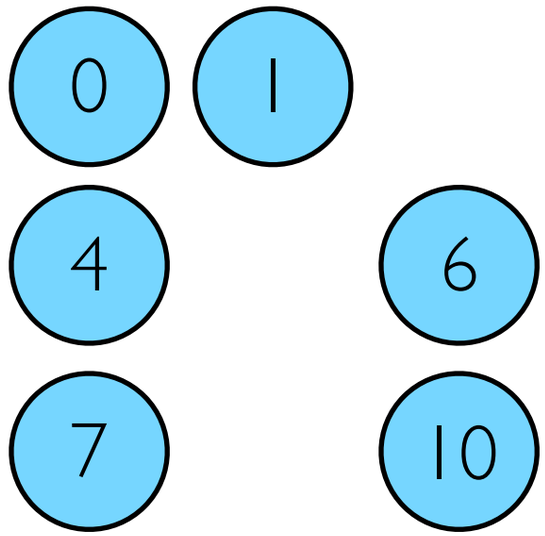


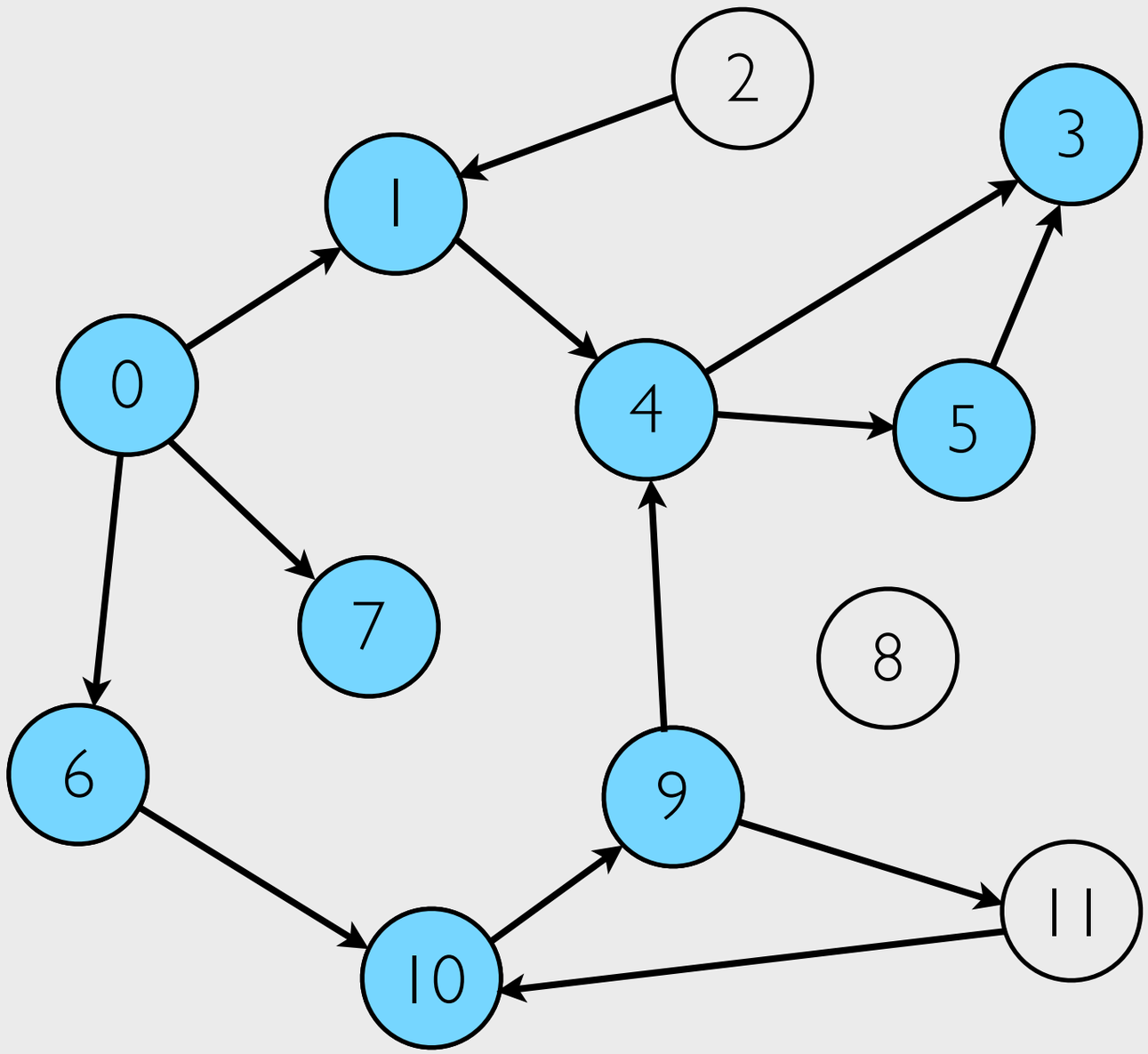
seen nodes



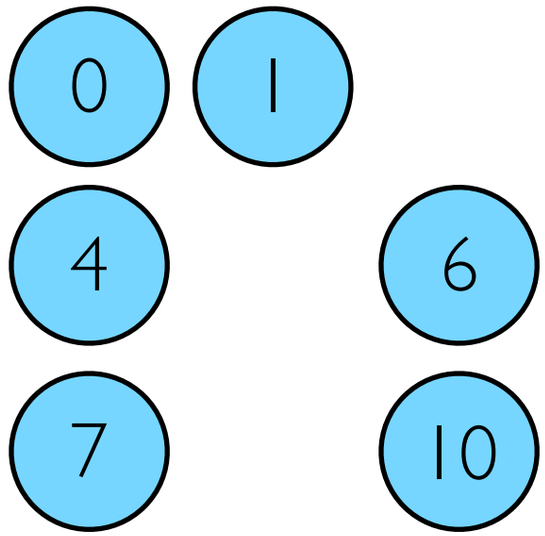


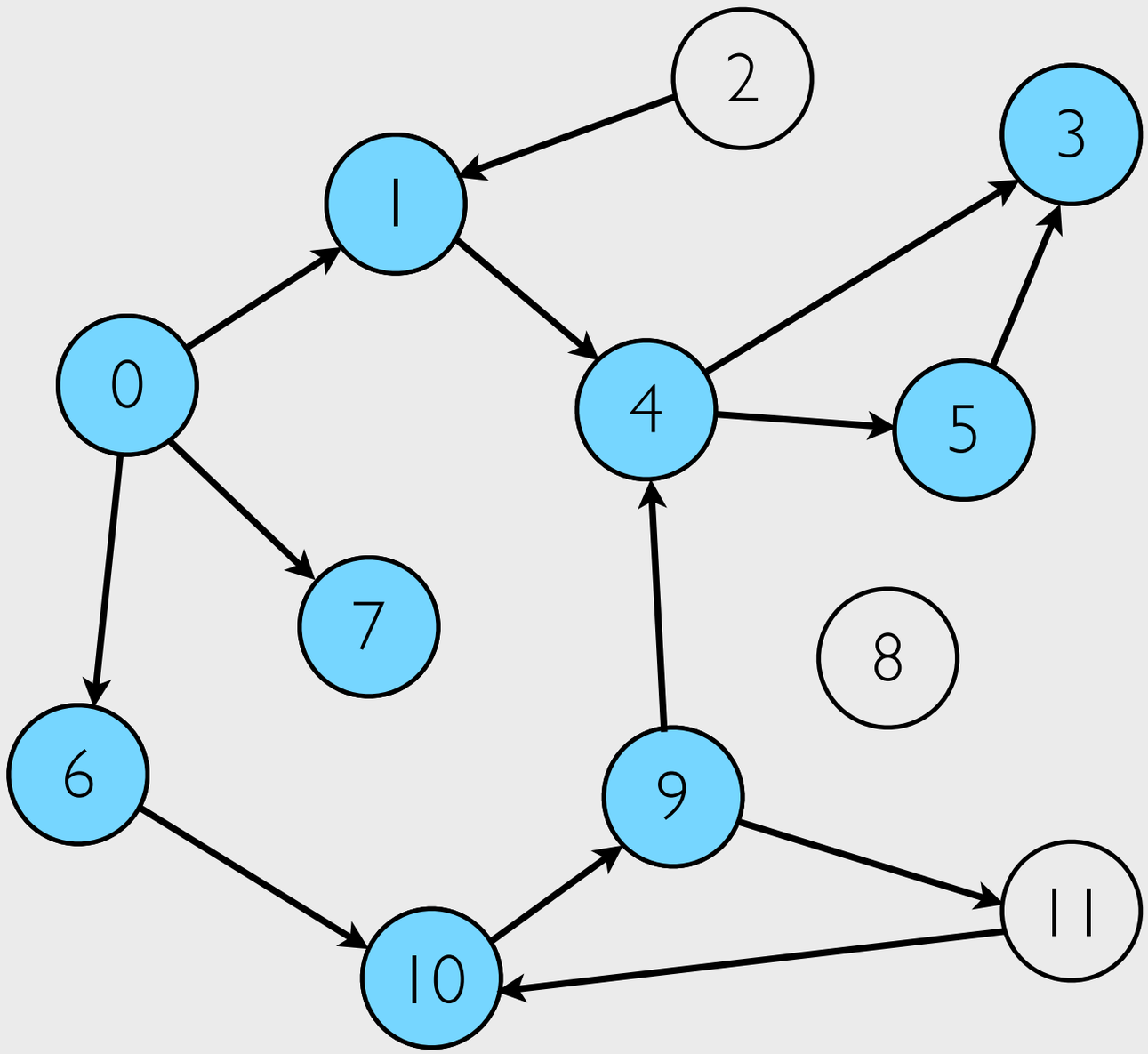
seen nodes



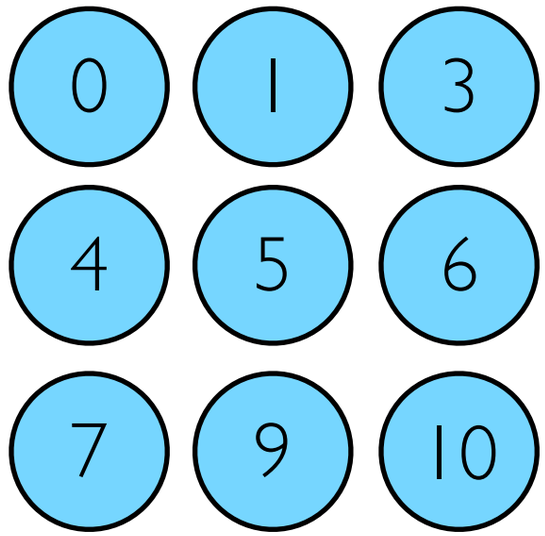


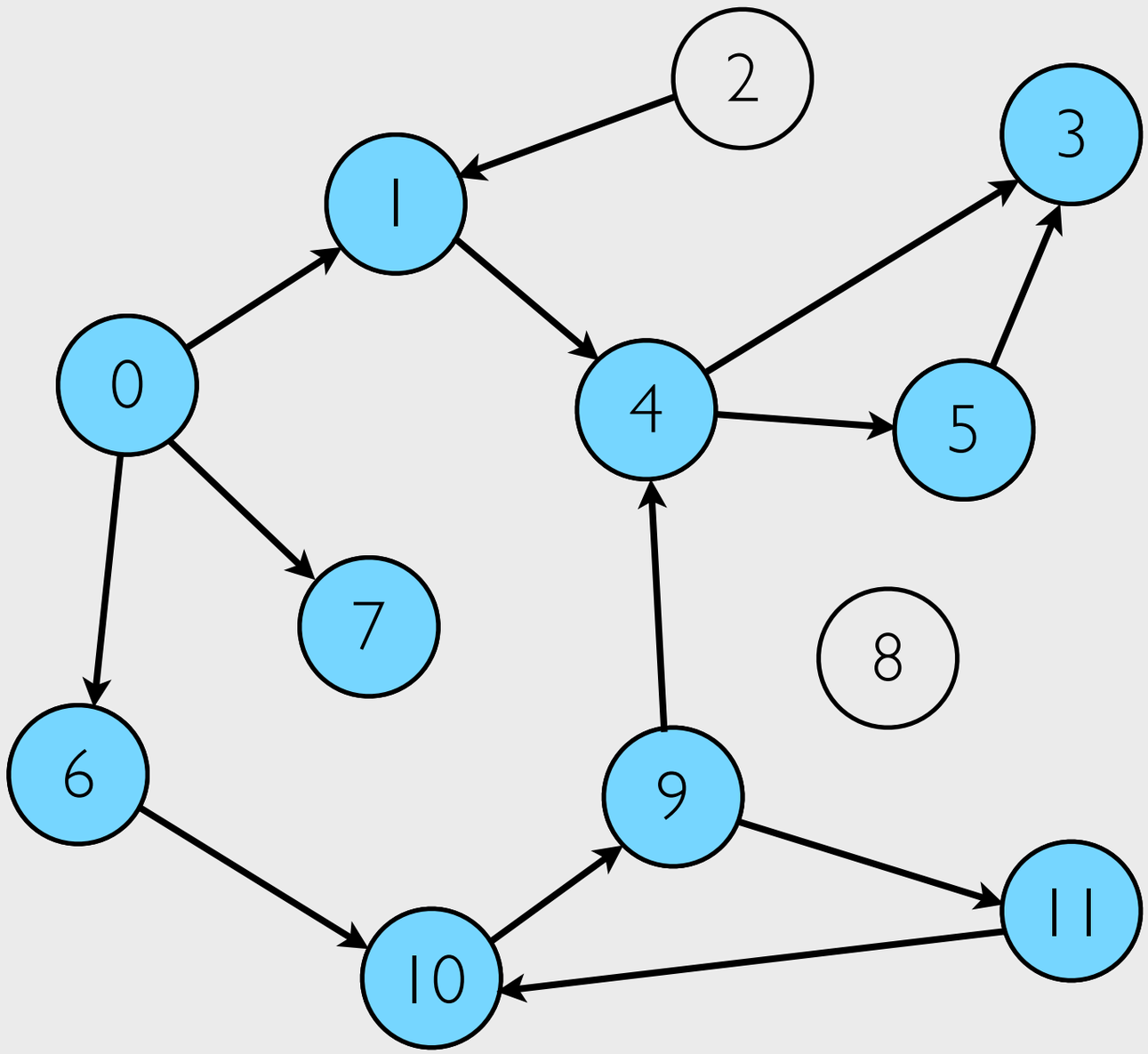
seen nodes



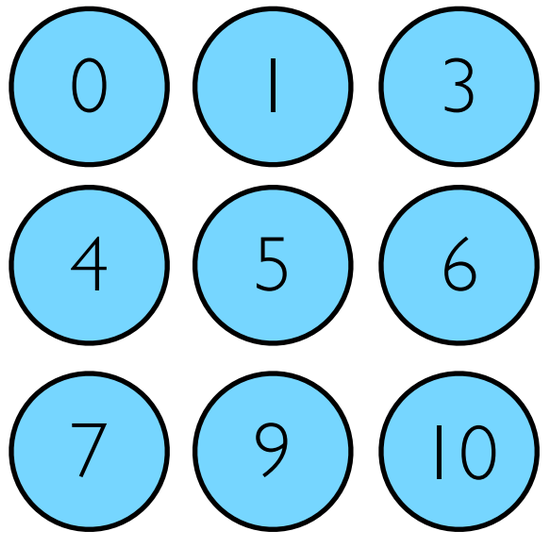


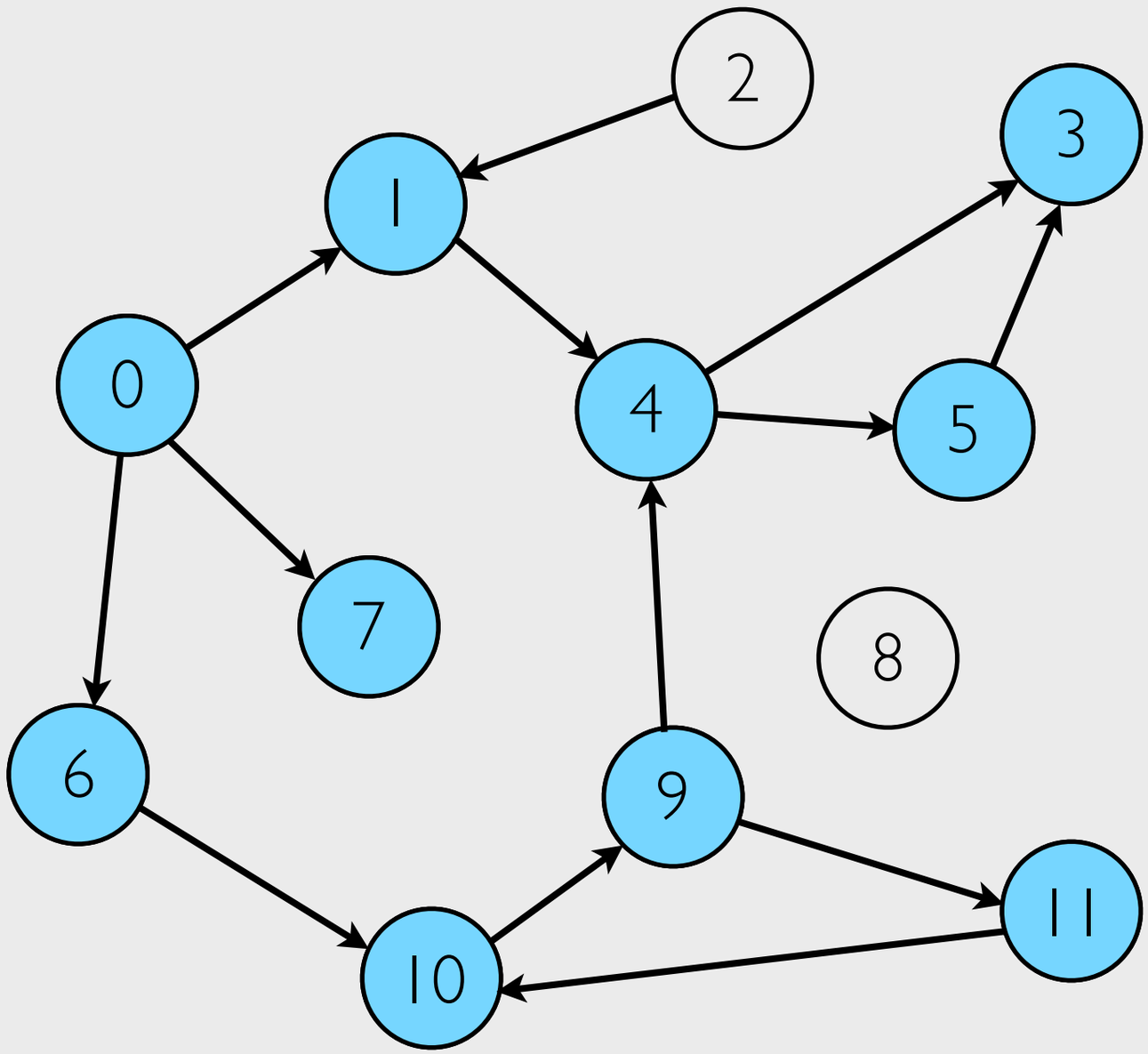
seen nodes



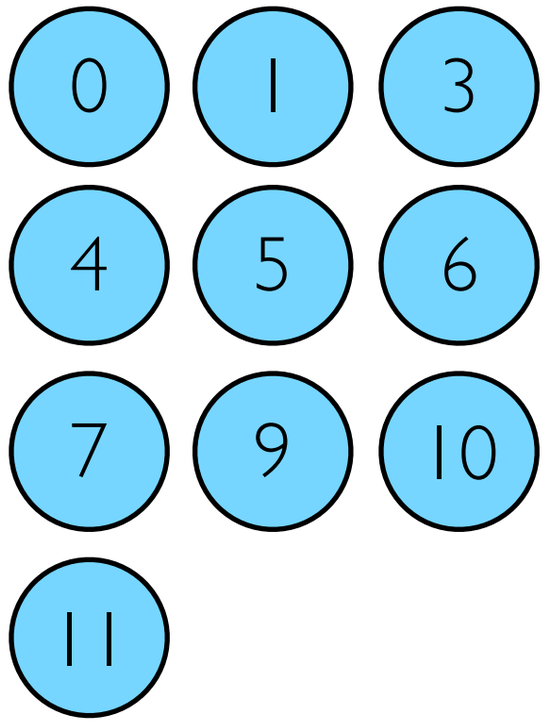


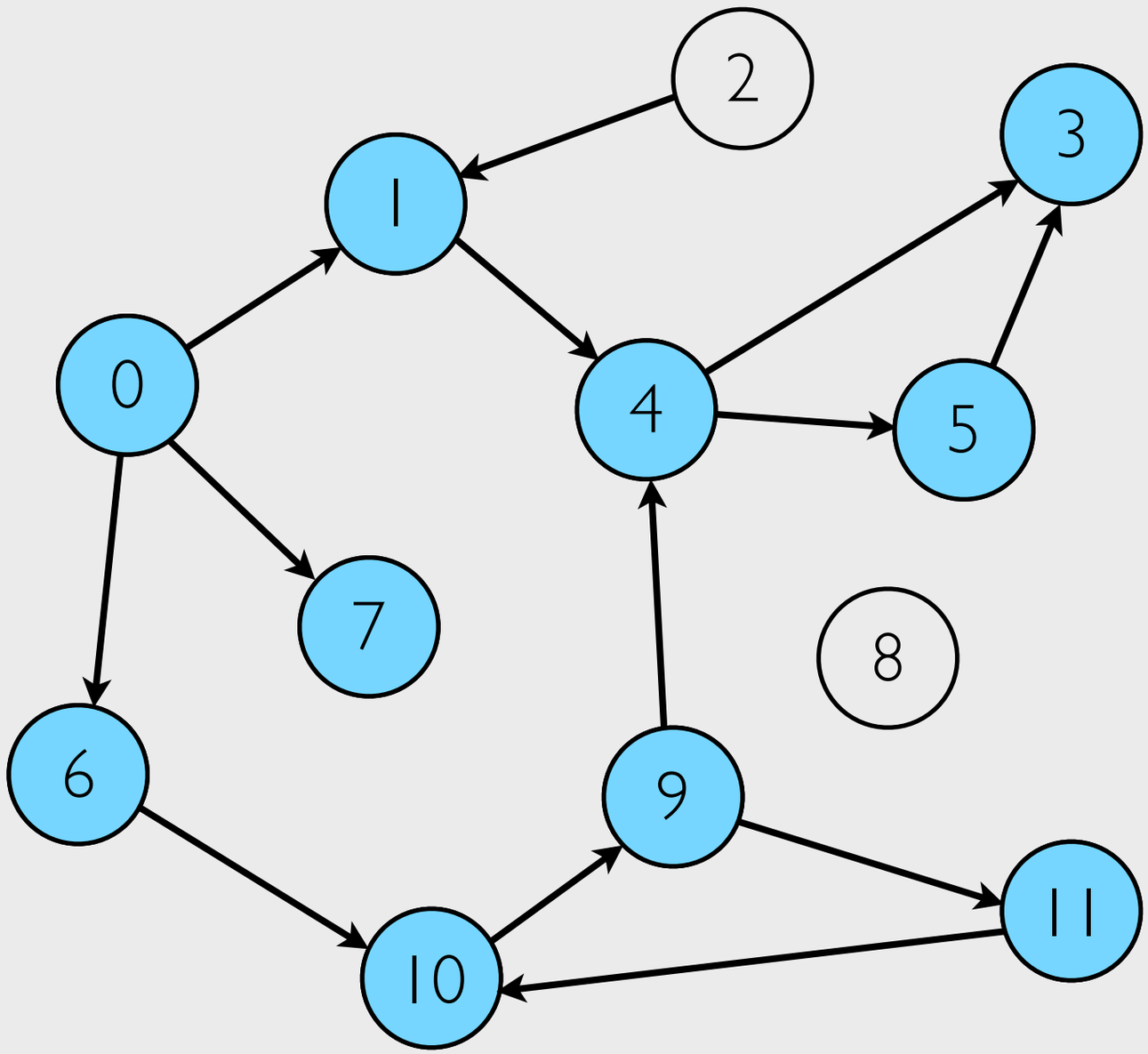
seen nodes



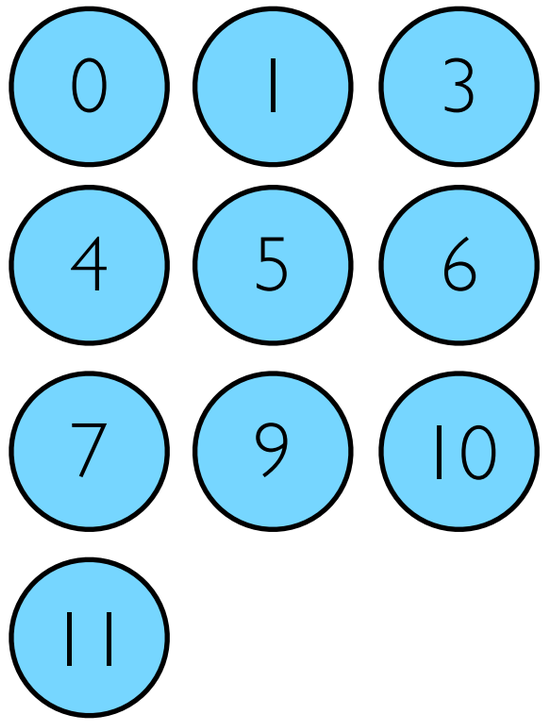


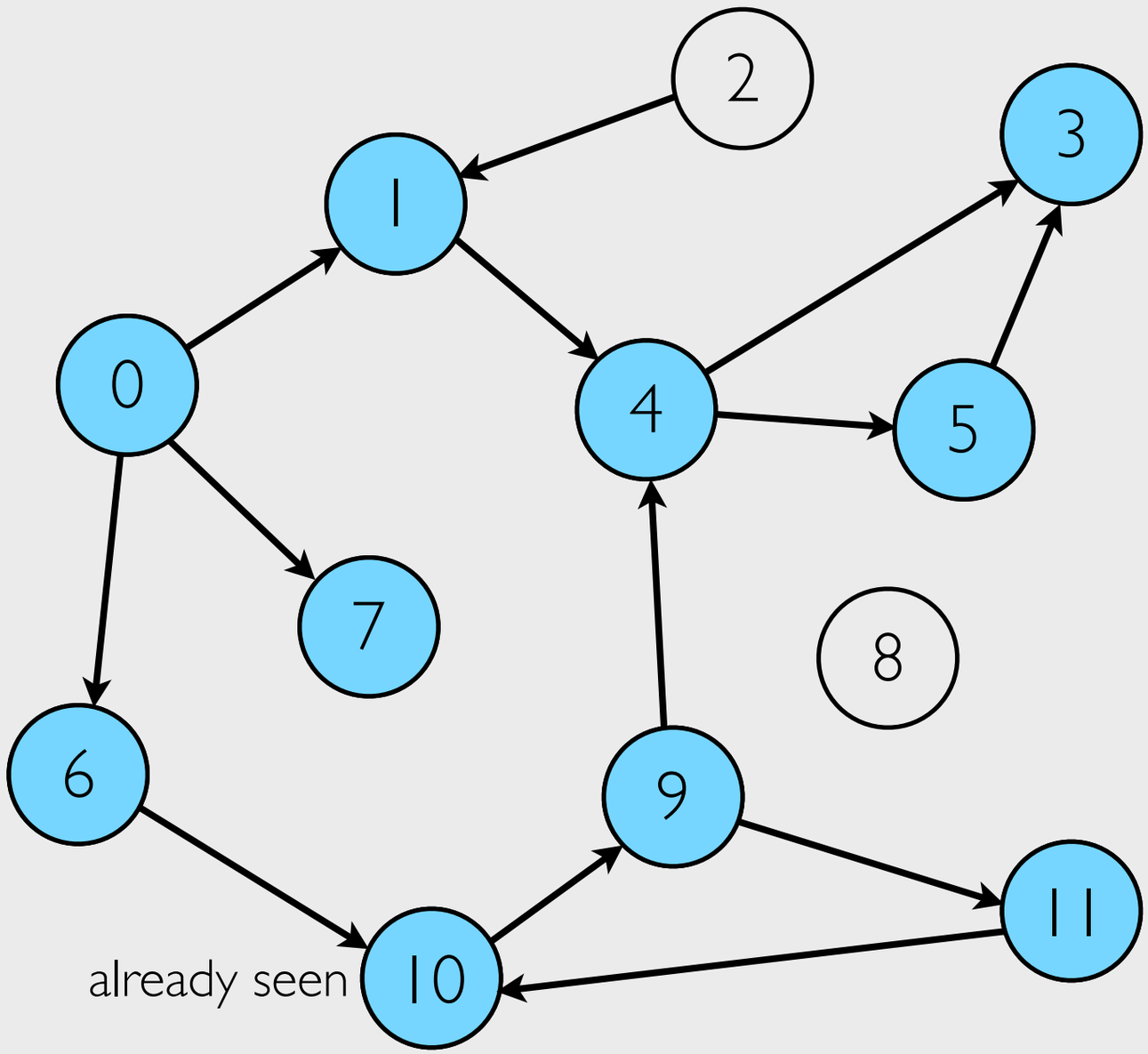
seen nodes



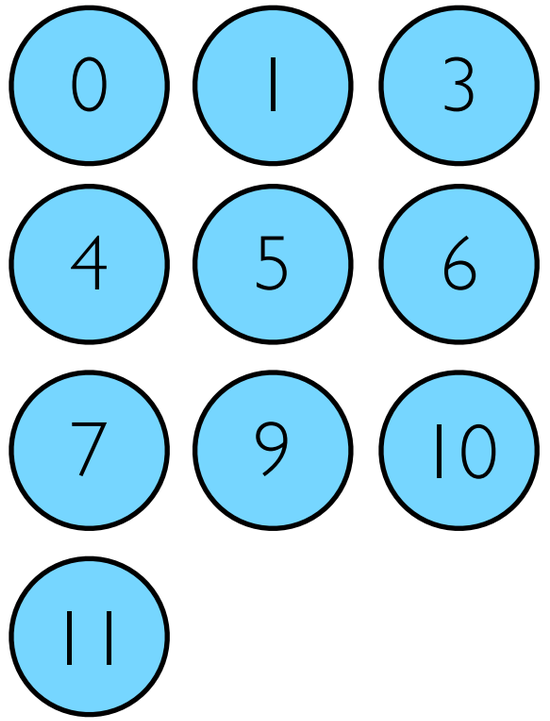


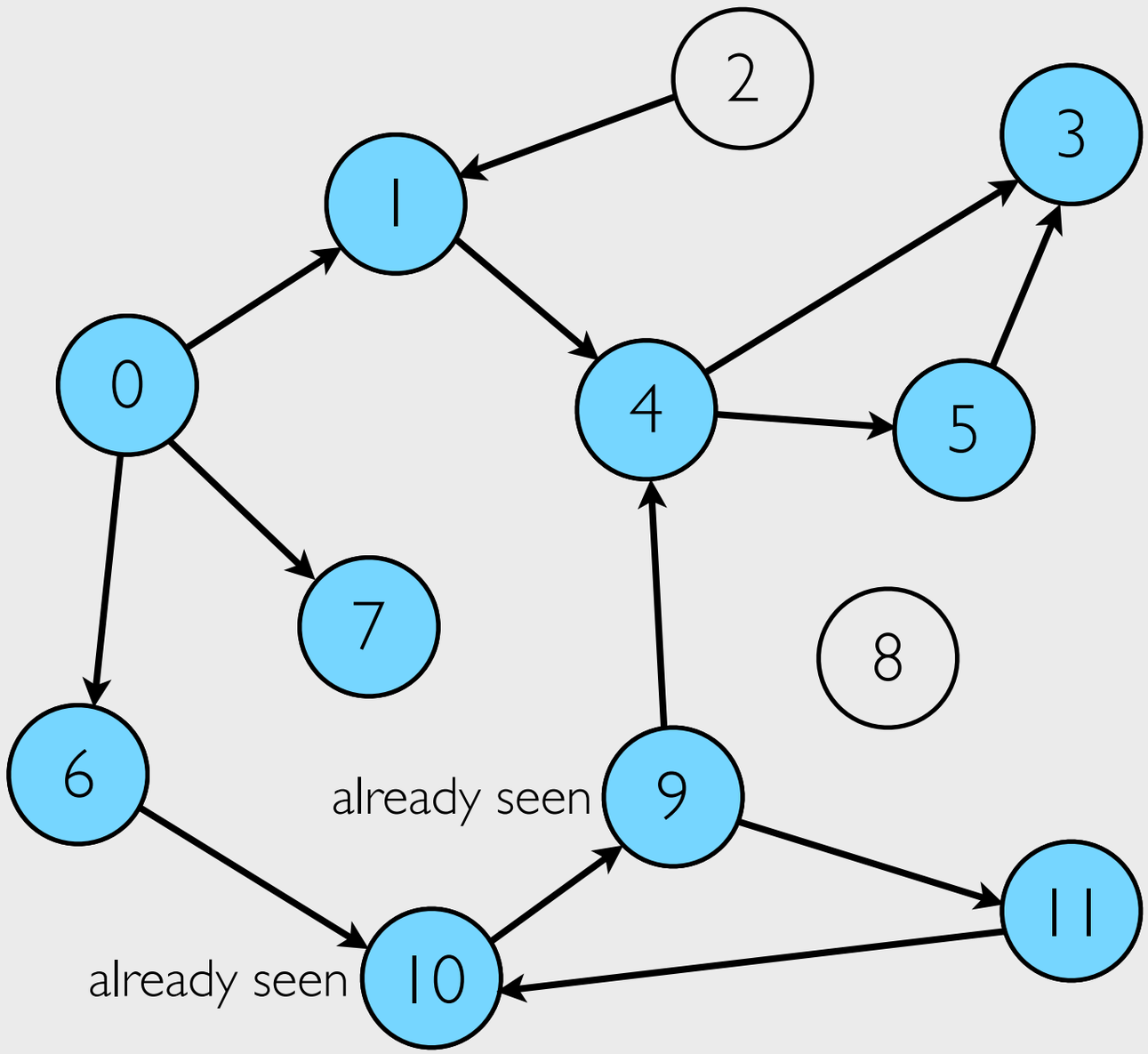
seen nodes



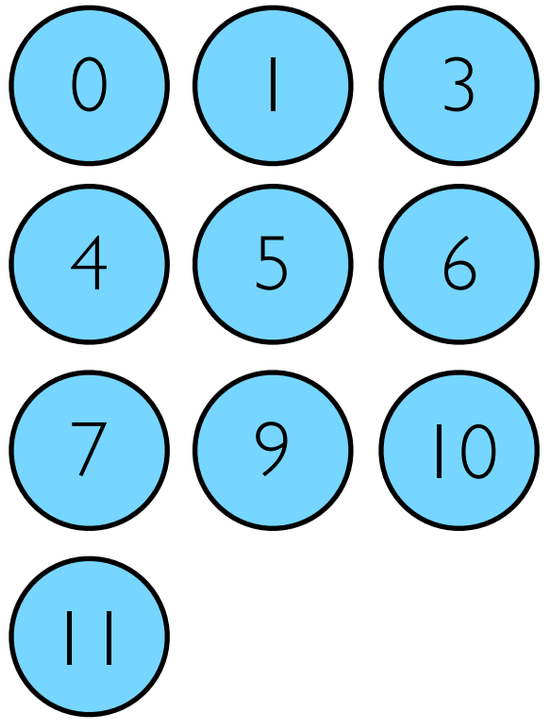


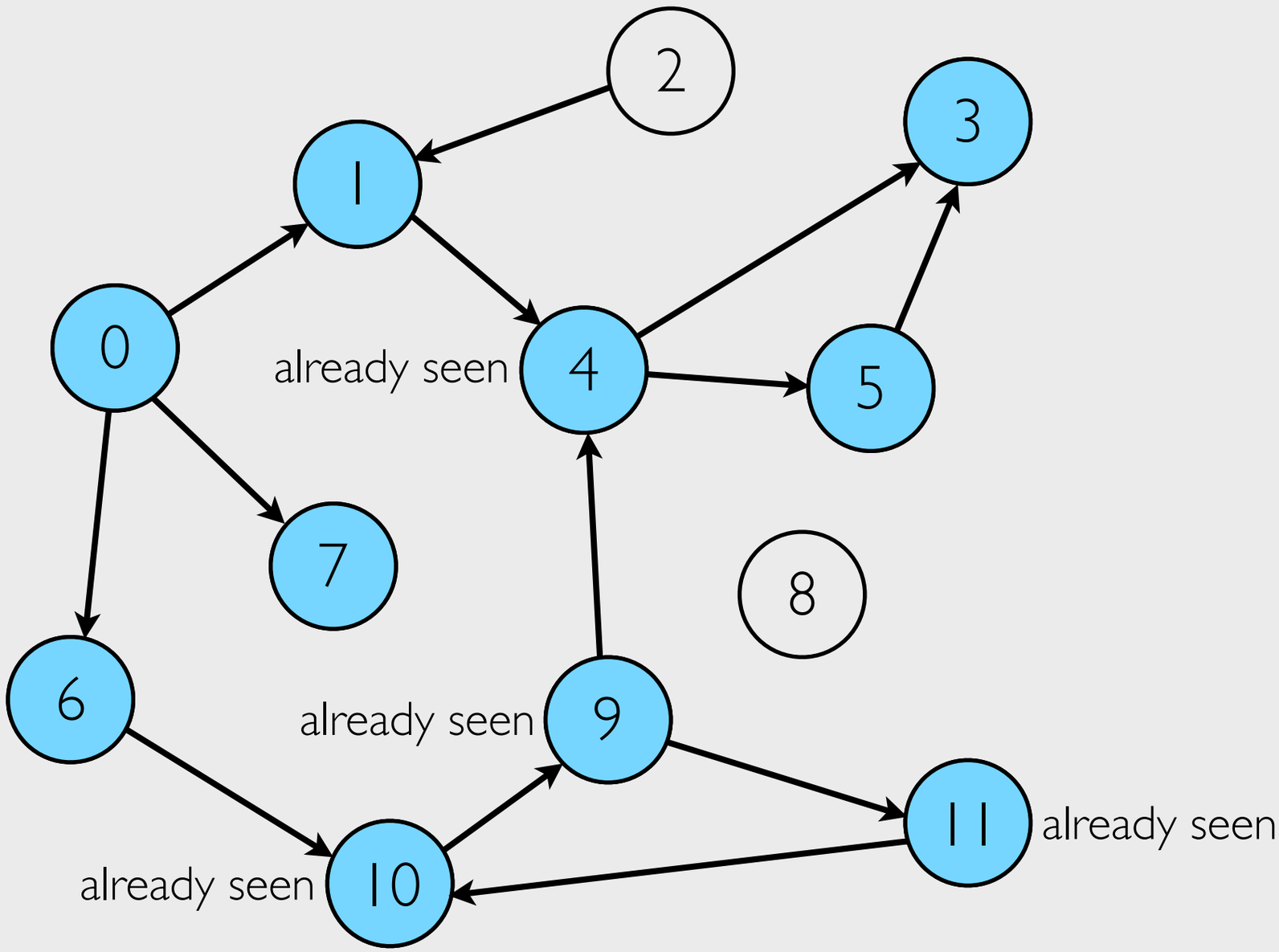
seen nodes



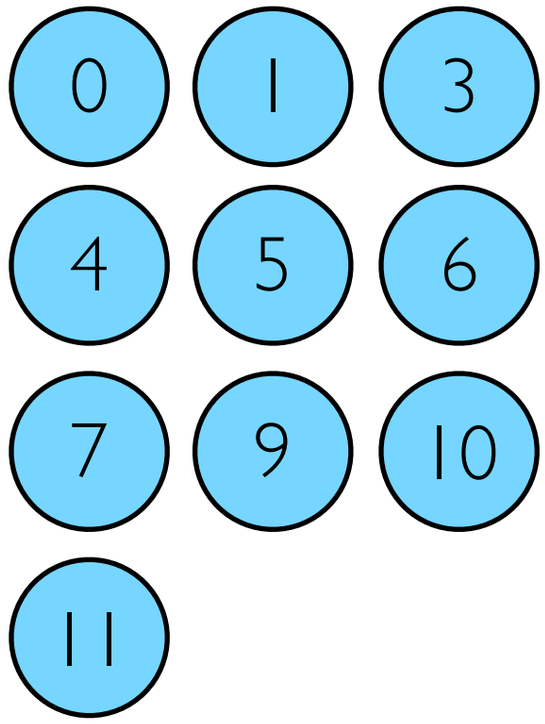


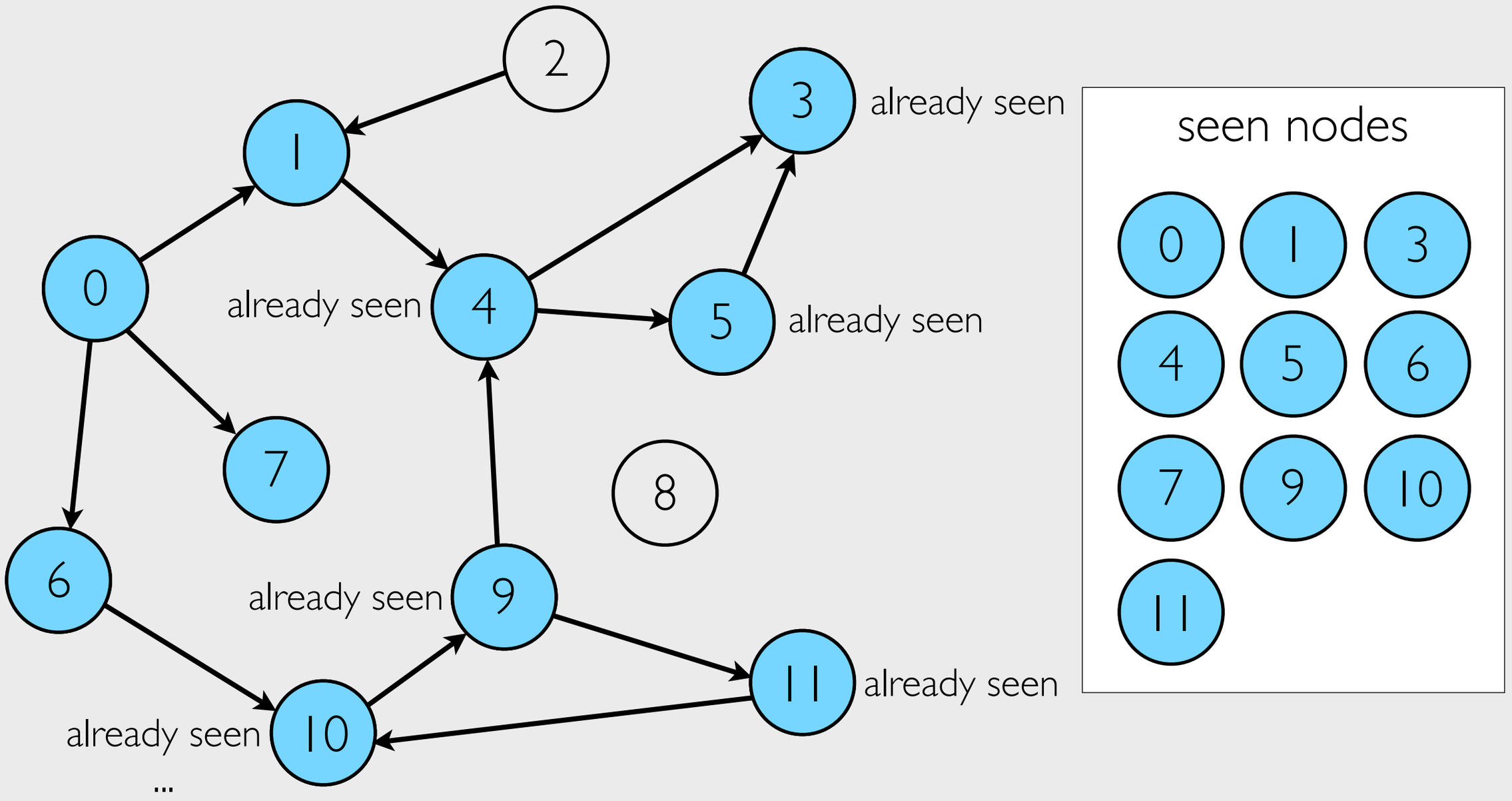
seen nodes

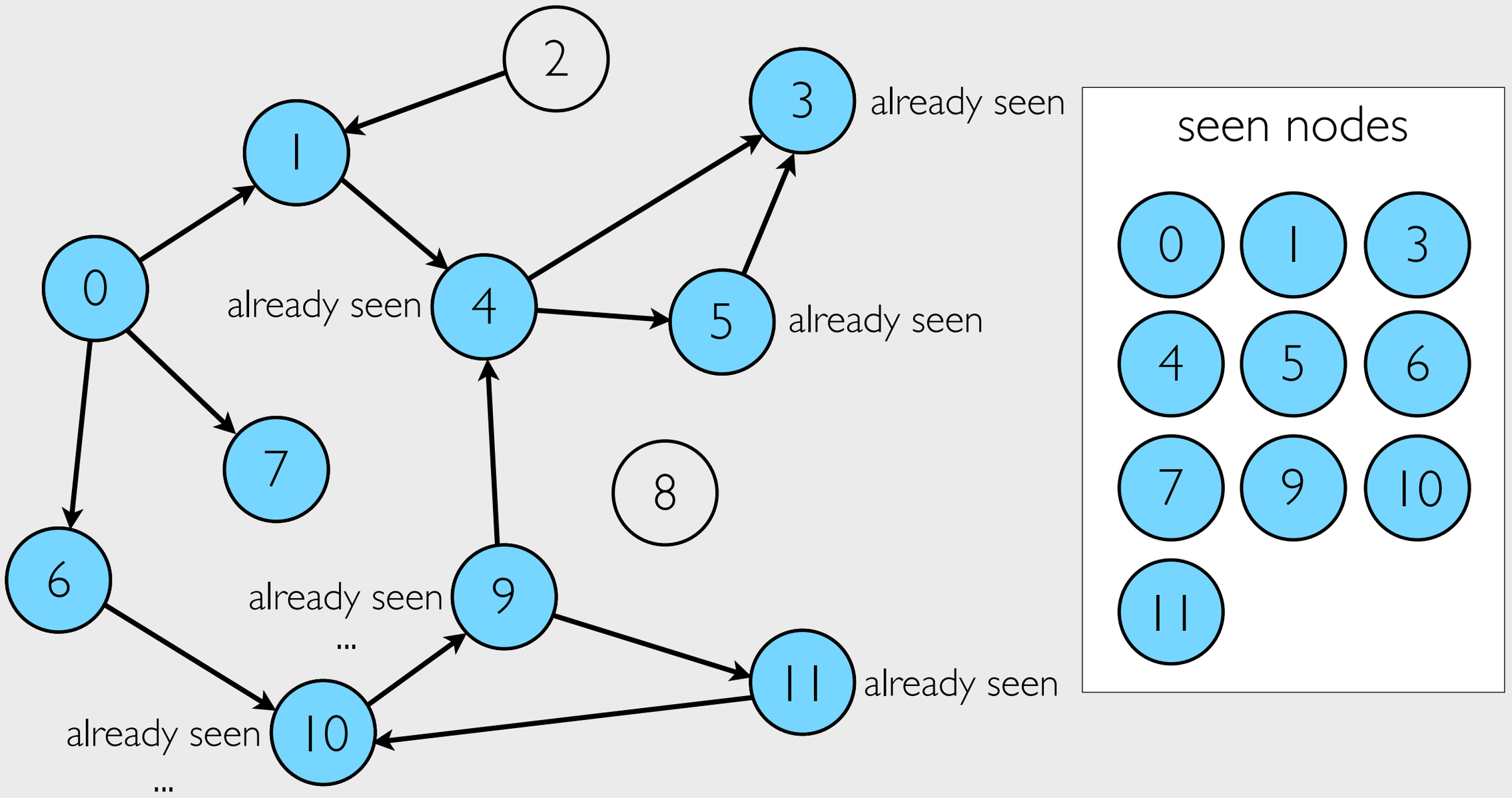


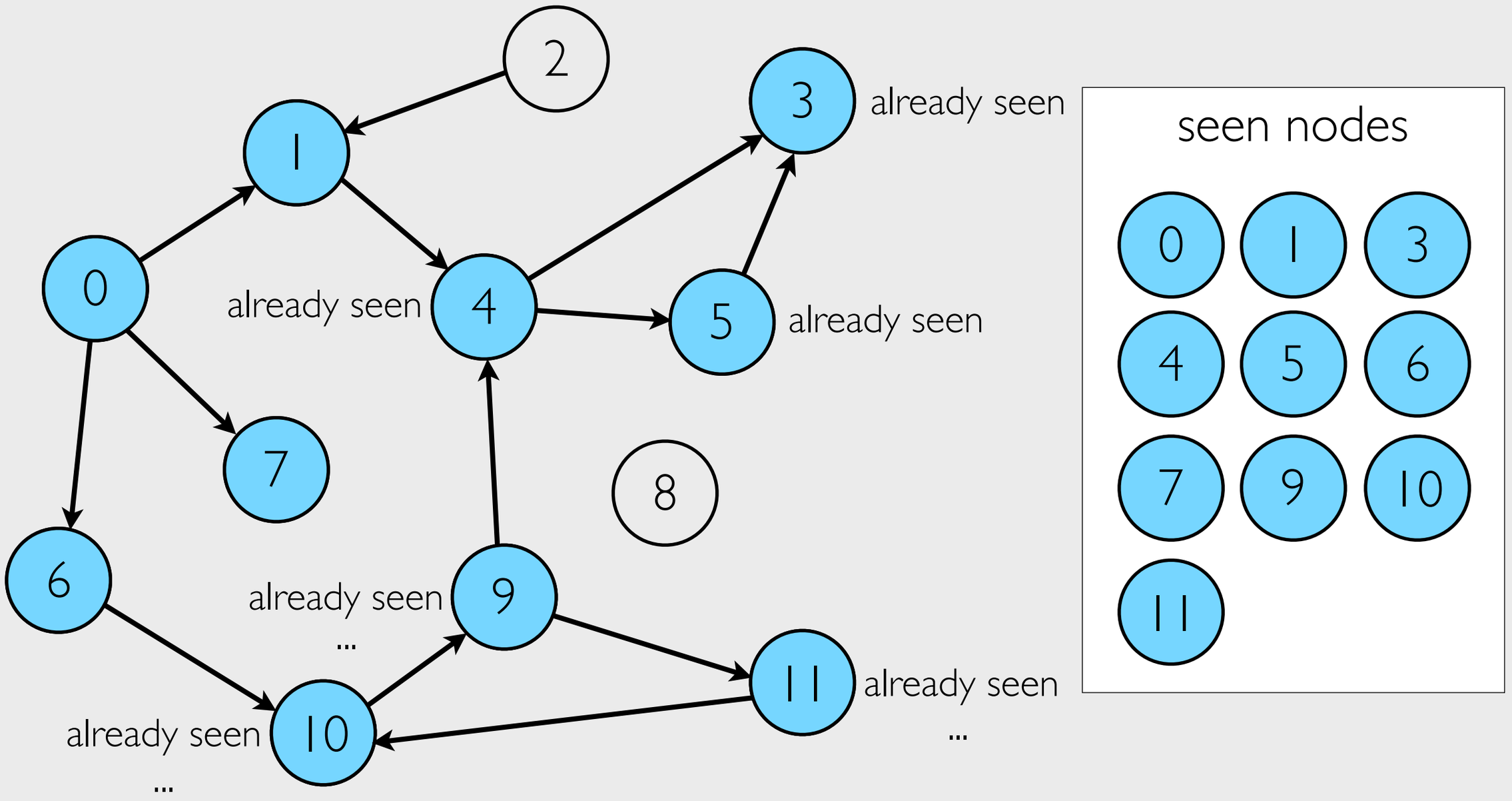

2
1
3
0
already seen
4
5
7
8
6
already seen
9
11
already seen
already seen
10

seen nodes

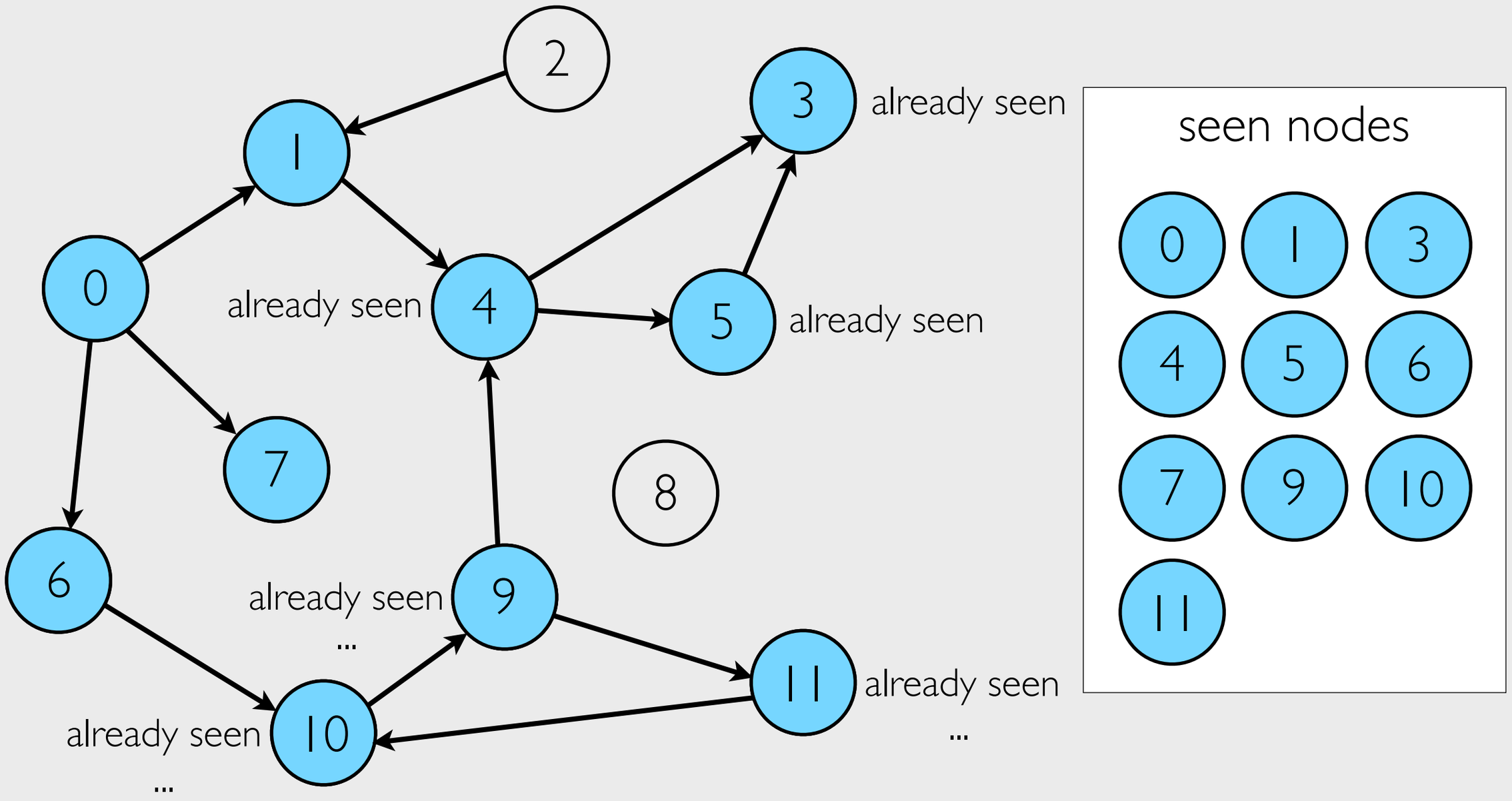

0
1
3
4
5
6
7
9
10
11


2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
9
11
already seen
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11


2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
...
9
11
already seen
already seen
...
10
seen nodes
0
1
3
4
5
6
7
9
10
11

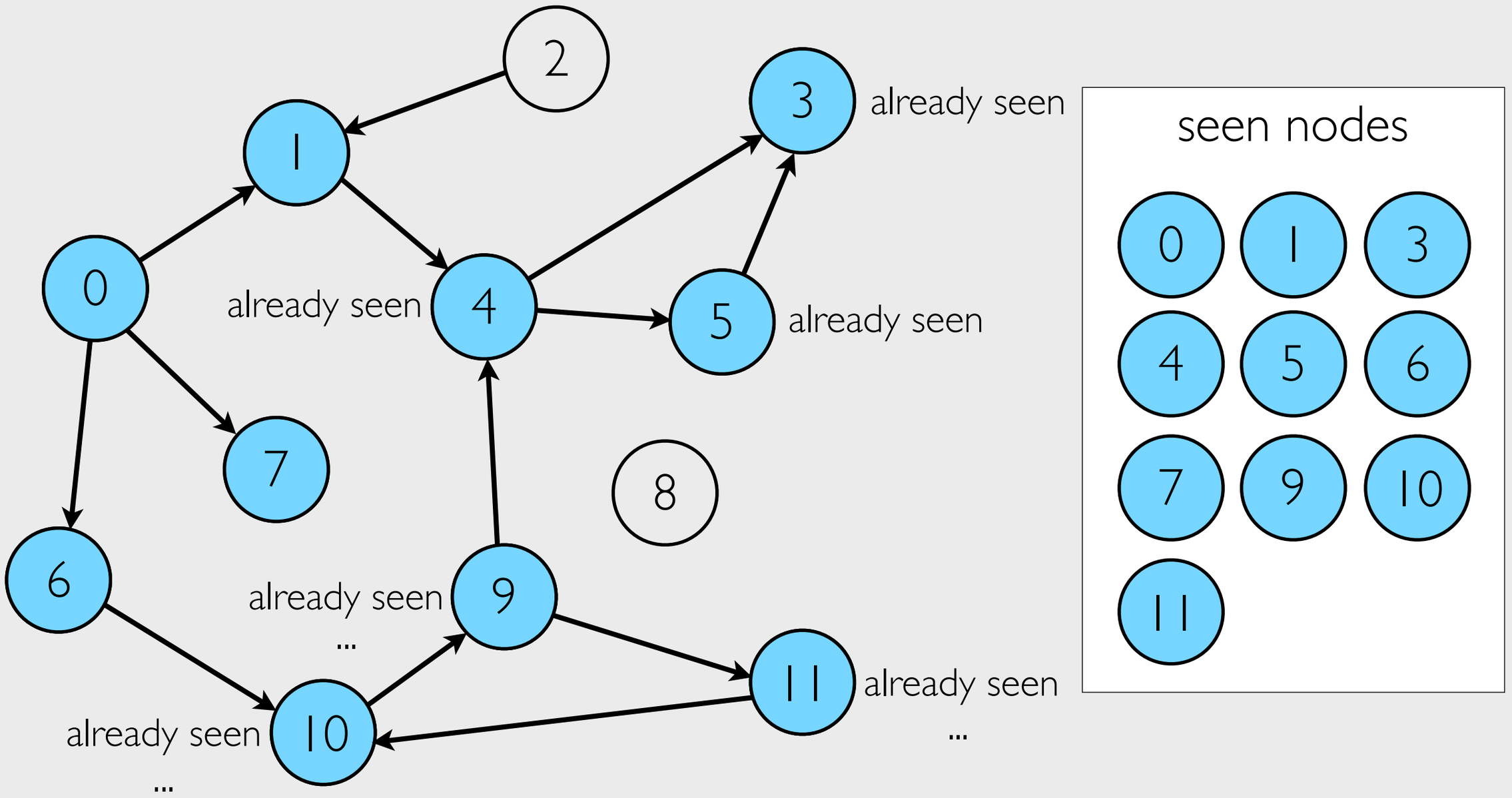

2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
...
9
11
already seen
...
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11

Events are updates that change an LVar's state



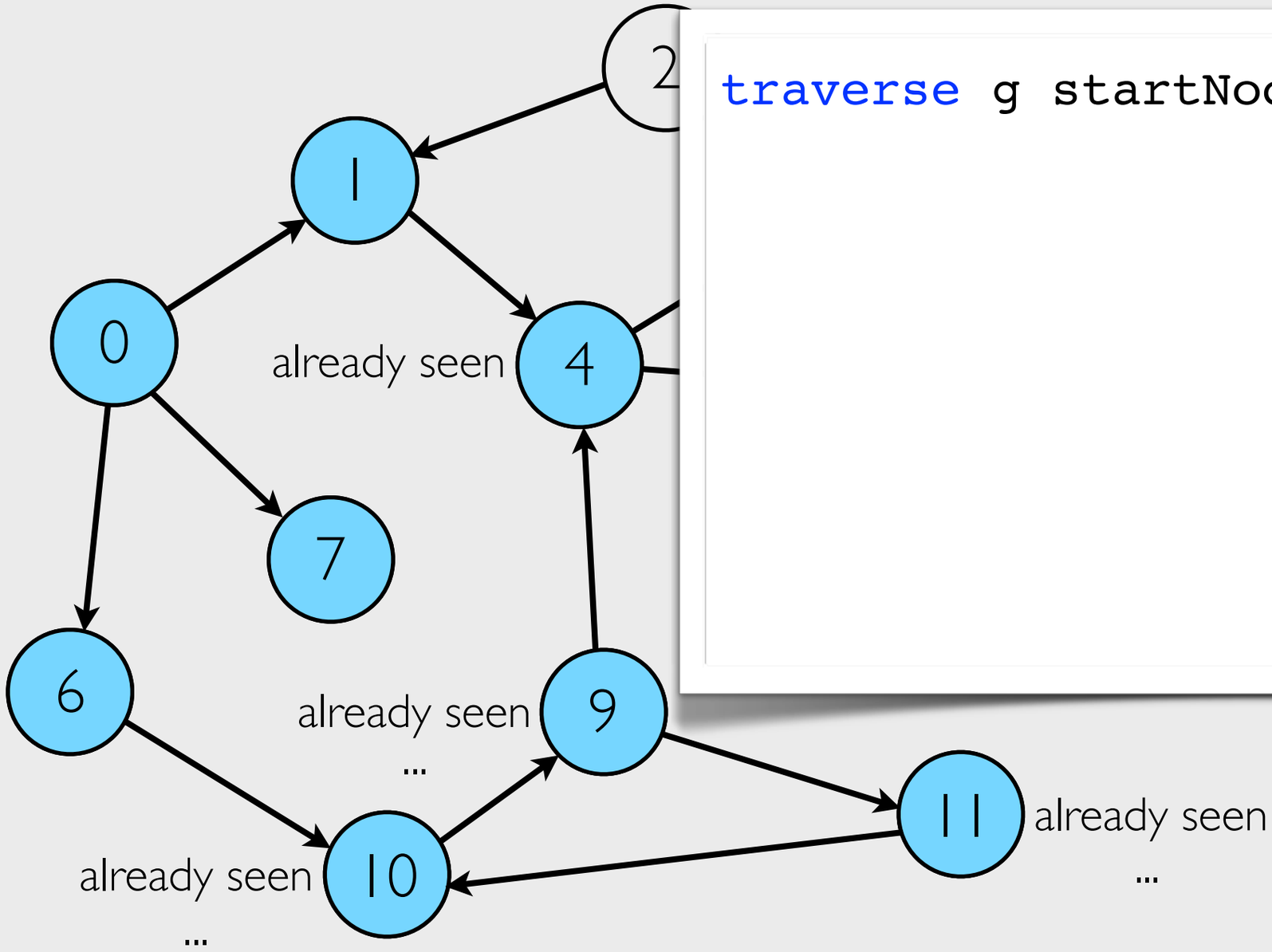
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response



Events are updates that change an LVar's state

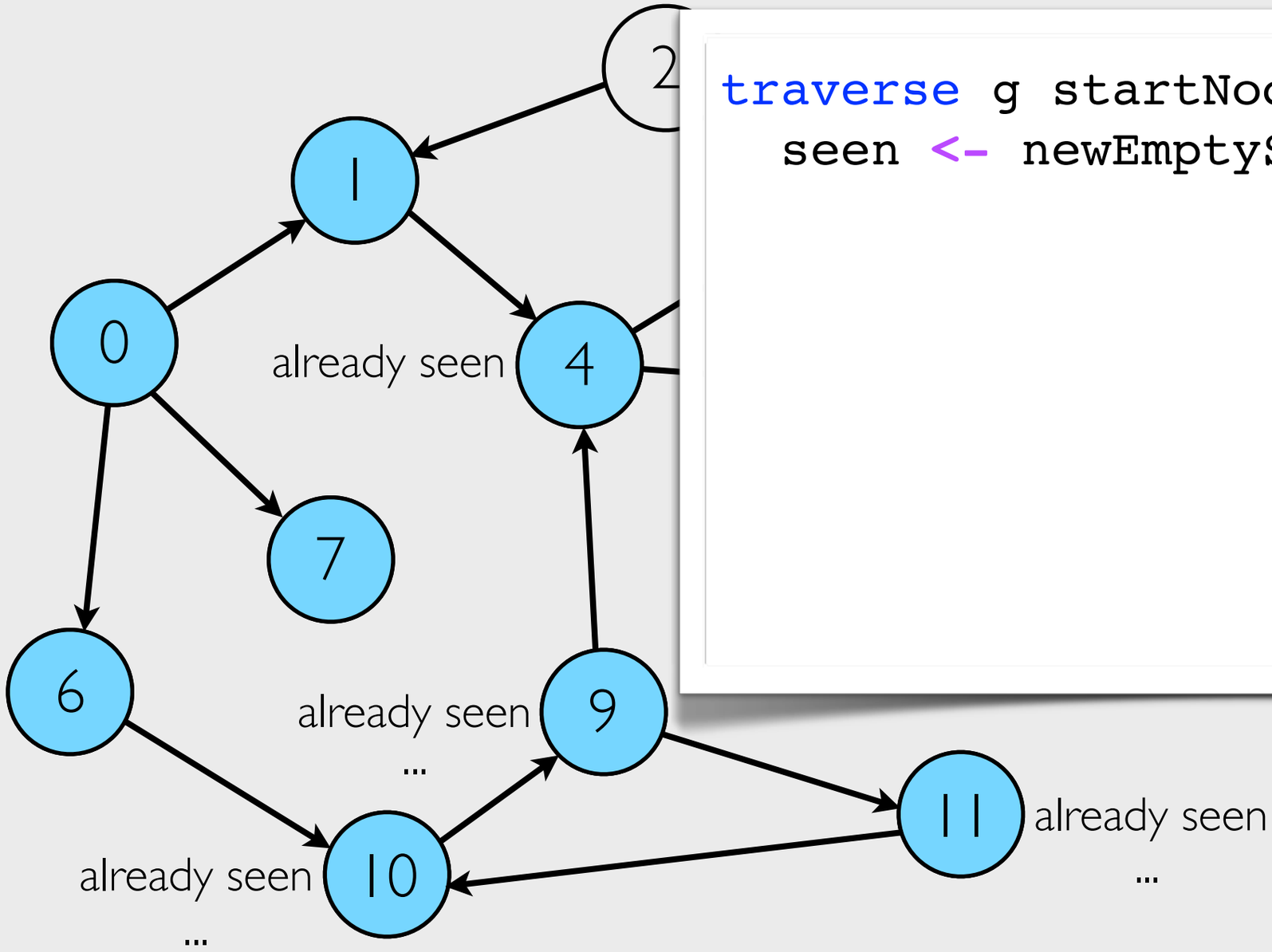
Event handlers listen for events and launch callbacks in response




```
traverse g startNode = do
```

Events are updates that change an LVar's state

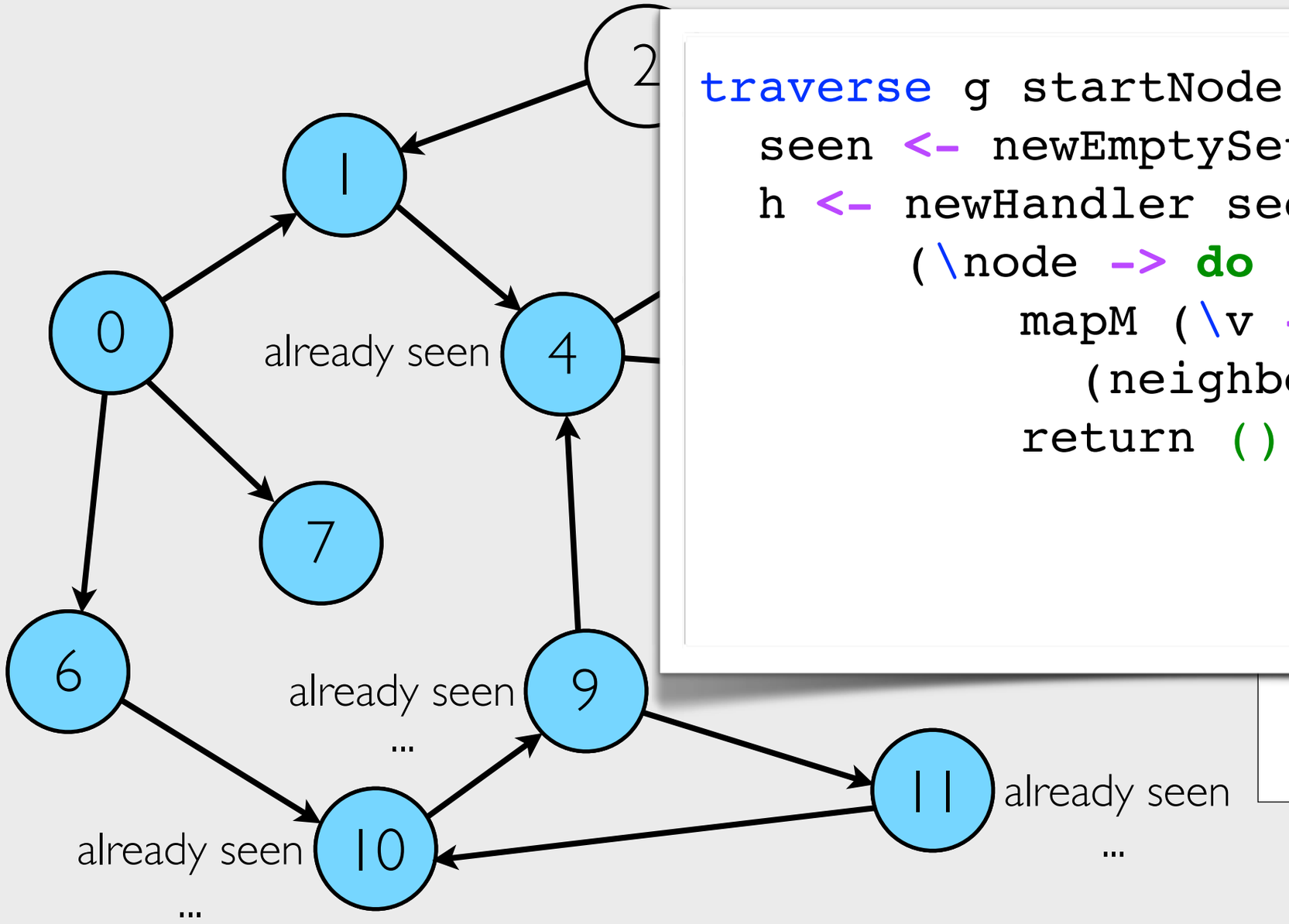
Event handlers listen for events and launch callbacks in response




```
traverse g startNode = do
  seen <- newEmptySet
```

Events are updates that change an LVar's state

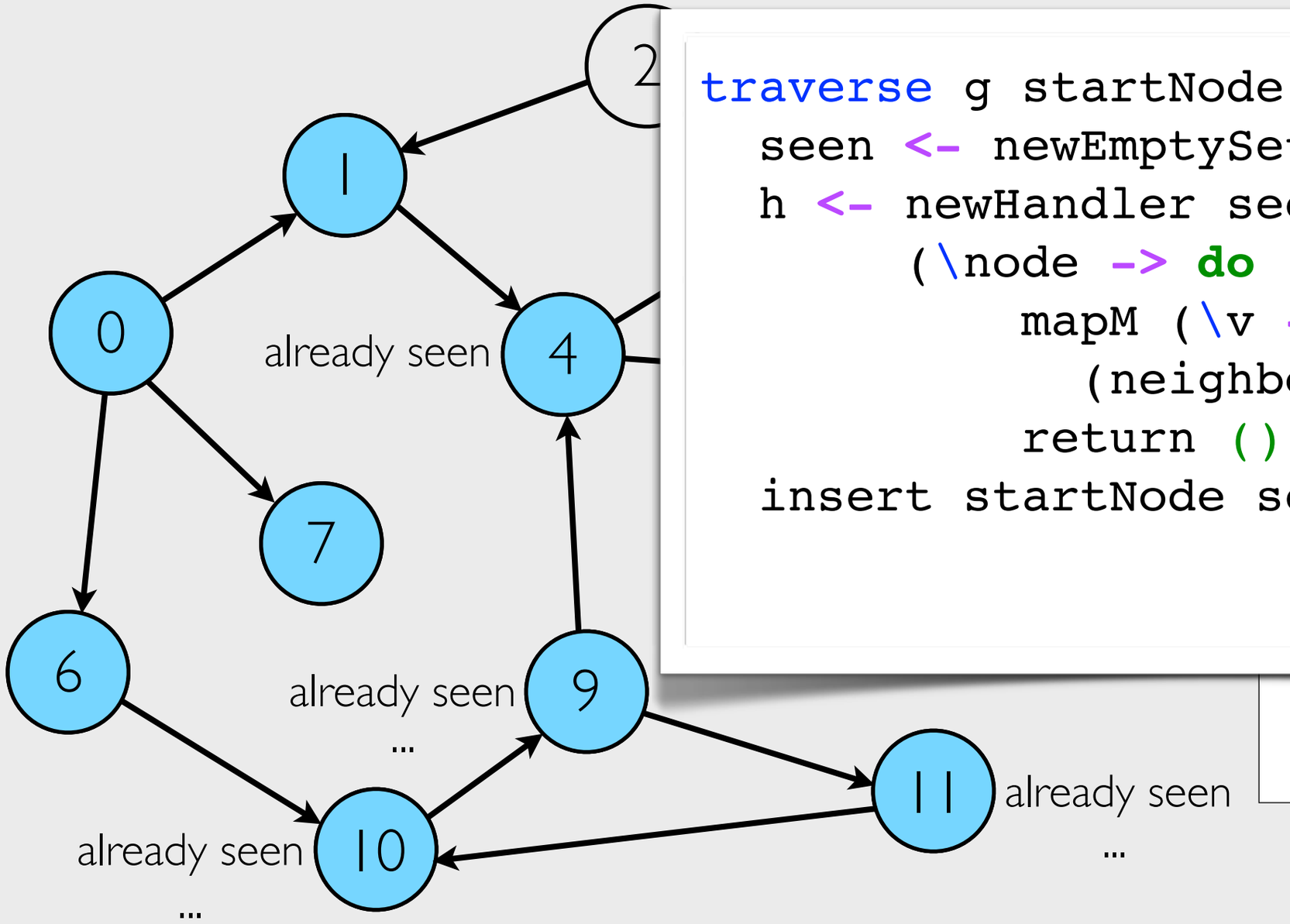
Event handlers listen for events and launch callbacks in response




```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

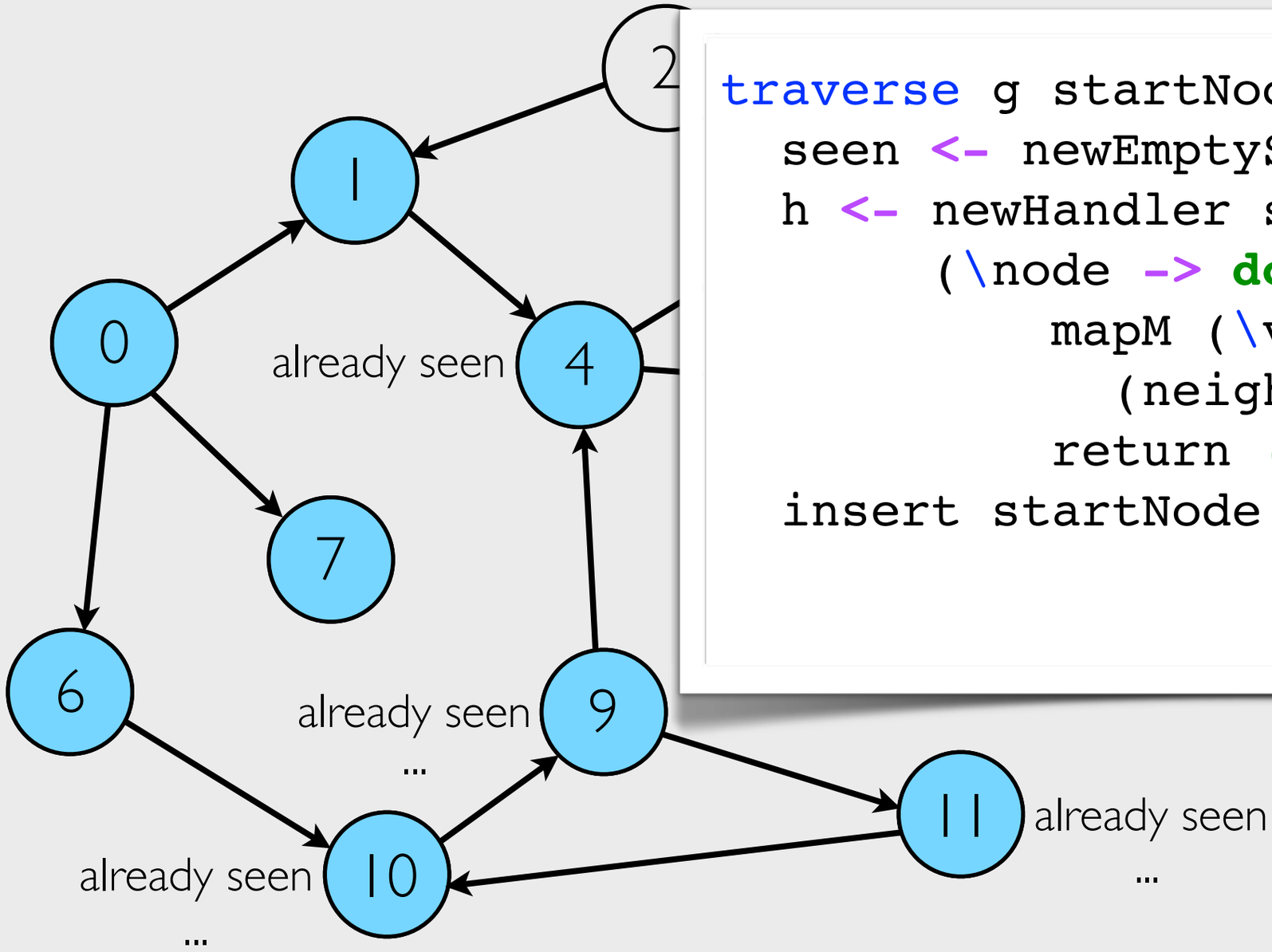



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running

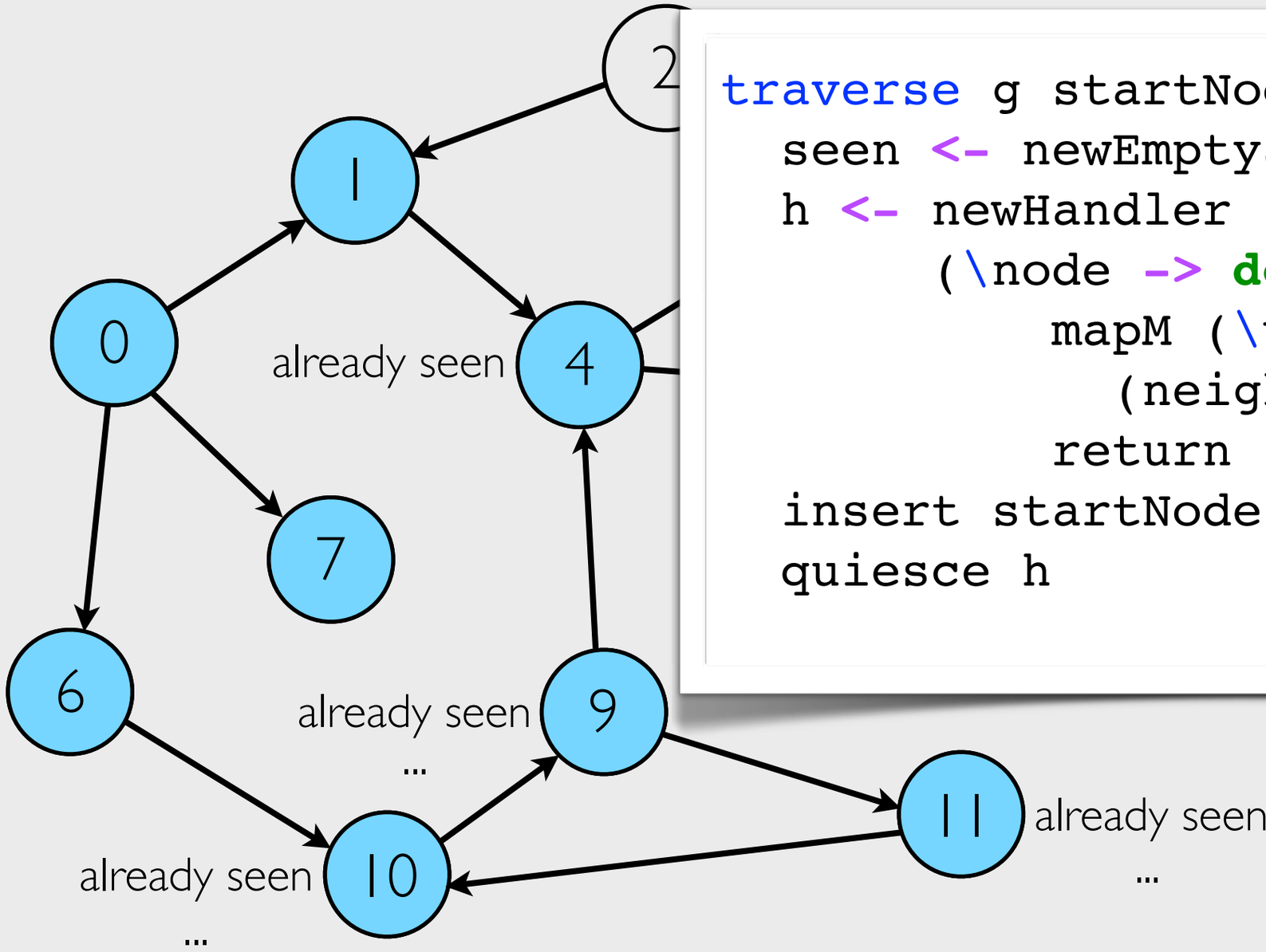



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running

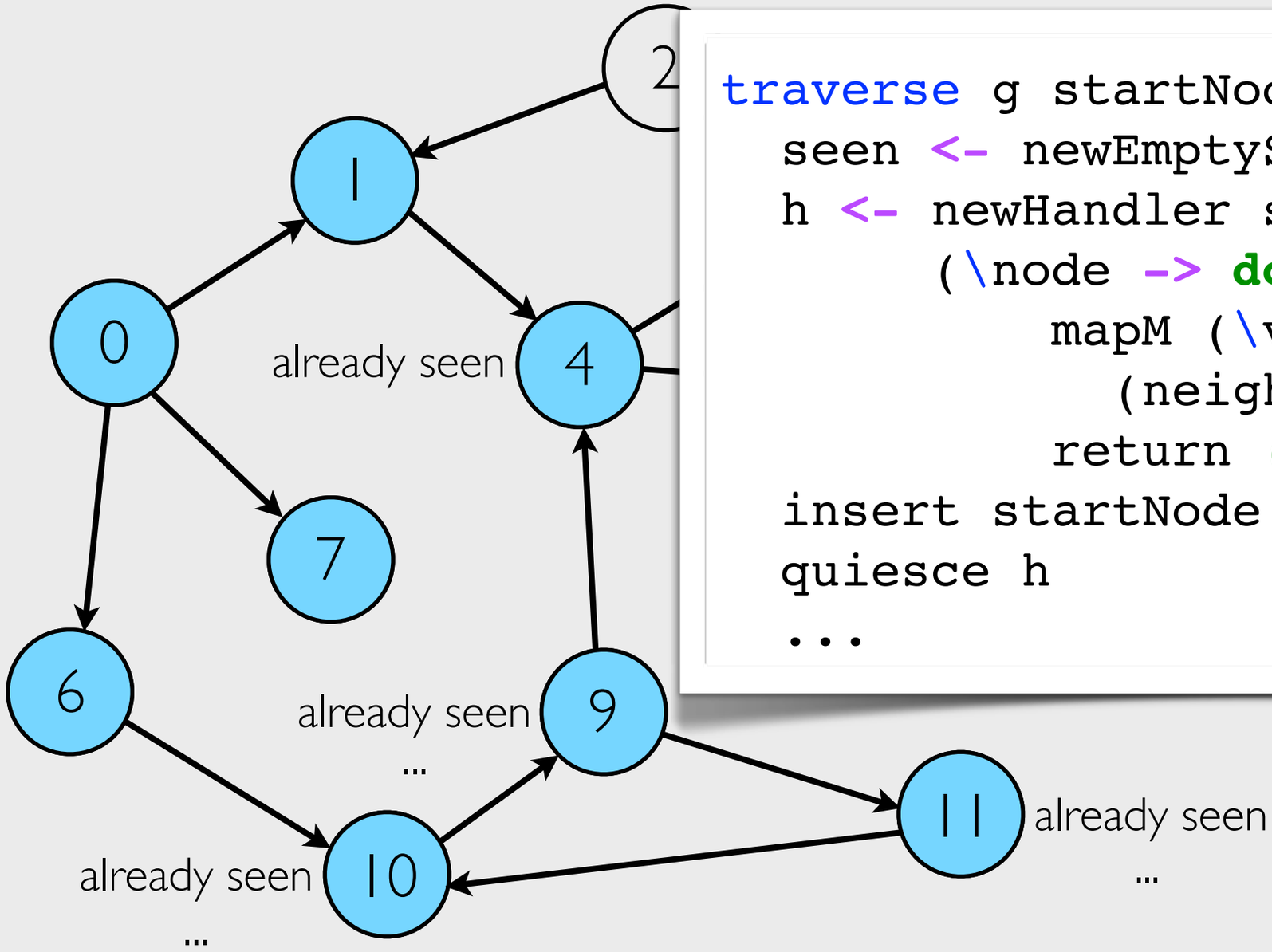



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running




```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
        (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longmapsto^* \sigma'$ *and* $\sigma \longmapsto^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

[POPL '14]

```
do
                                  nsert v seen)
                                  g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

[POPL '14]

```
do
                                  nsert v seen)
                                  g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```

[(Book,1),(Shoes,1)]

[(Book,1)]

[(Shoes,1)]

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

[POPL '14]


```
do
                                    nsert v seen)
                                  g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

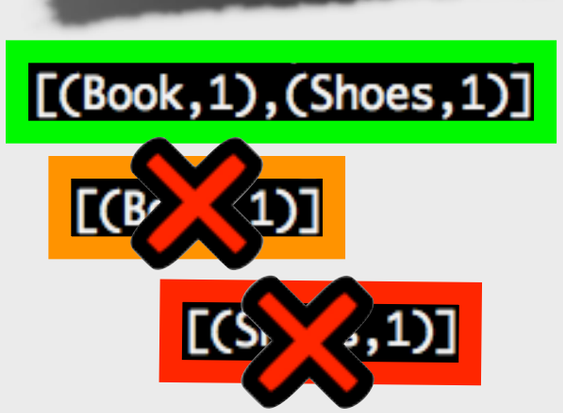
Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

[POPL '14]

```
do
                                  nsert v seen)
                                  g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```



`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \hookrightarrow^* \sigma'$ *and* $\sigma \hookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

[POPL '14]

```
do
nsert v seen)
g node)
return ())
insert startNode seen
quiesce h
freeze seen
```

[(Book,1),(Shoes,1)] or error.

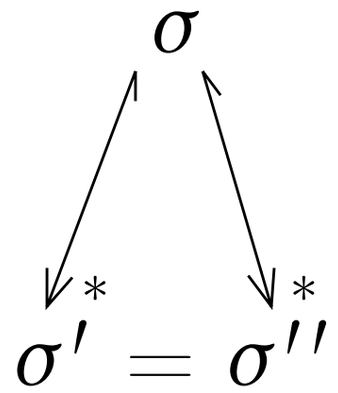
[(B 1)]

[(S ,1)]

aka "LVars"

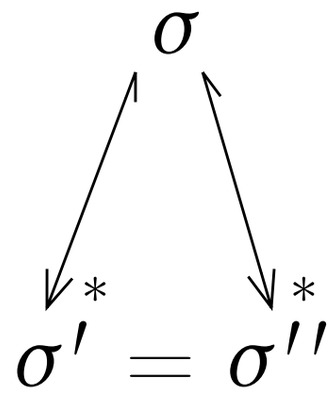
Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

Determinism



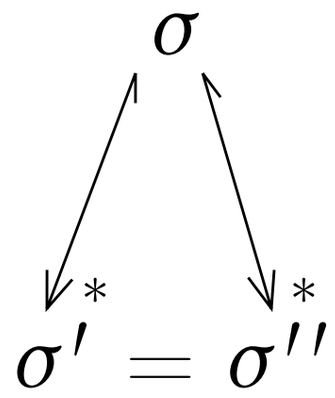
```
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter
```

Determinism

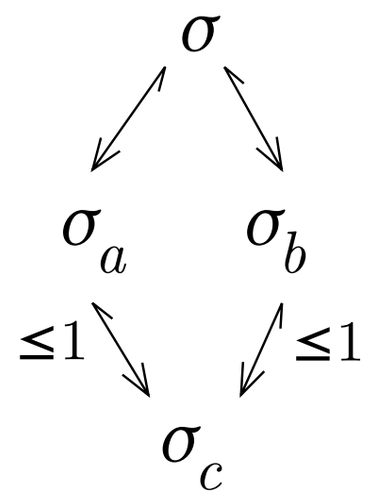


```
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter
```

Determinism



Strong Local Confluence

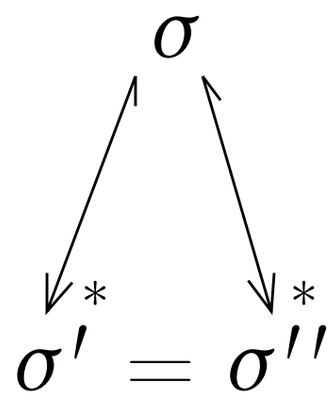


```
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter
```

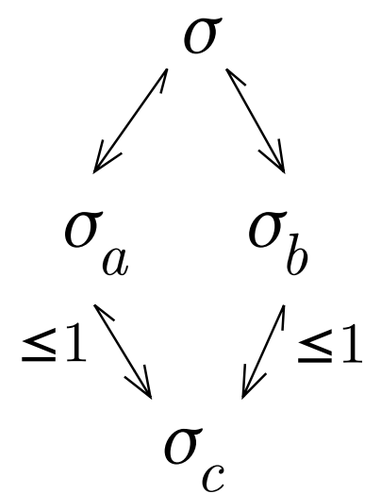
Independence

$$\frac{\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle}{\langle S \sqcup_S S'';\, e\rangle \hookrightarrow \langle S' \sqcup_S S'';\, e'\rangle}$$

Determinism



Strong Local Confluence



Frame rule

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

[O'Hearn *et al.*, 2001]

Independence

$$\frac{\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle}{\langle S \sqcup_S S'';\ e\rangle \hookrightarrow \langle S' \sqcup_S S'';\ e'\rangle}$$

Frame rule

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

[O'Hearn *et al.*, 2001]

Independence

$$\frac{\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle}{\langle S \sqcup_S S'';\ e\rangle \hookrightarrow \langle S' \sqcup_S S'';\ e'\rangle}$$

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

LVish

a Haskell library for parallel programming with LVars

LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
```

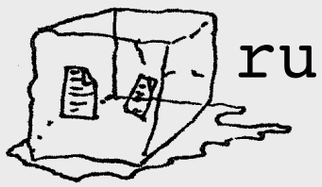
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
```

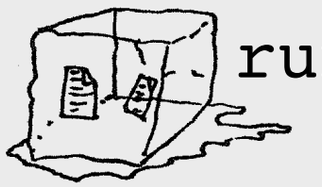
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart
```

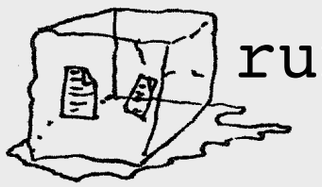
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

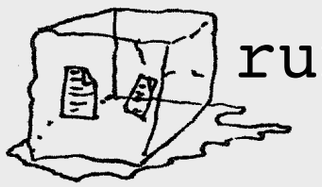
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

Efficient lock-free sets, maps, *etc.*

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

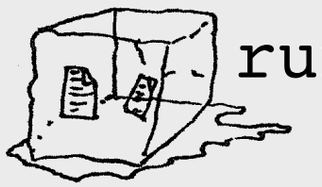
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

Efficient lock-free sets, maps, *etc.*

Implement your own LVars, too

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

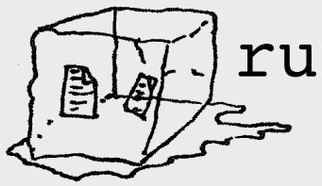
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

Efficient lock-free sets, maps, *etc.*

Implement your own LVars, too

```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

`hackage.haskell.org/package/lvish`

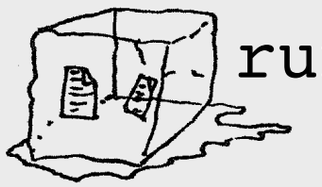
LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

Lightweight threads

`Par` computations indexed by *effect level*



`runParThenFreeze` expresses the freeze-after-writing idiom

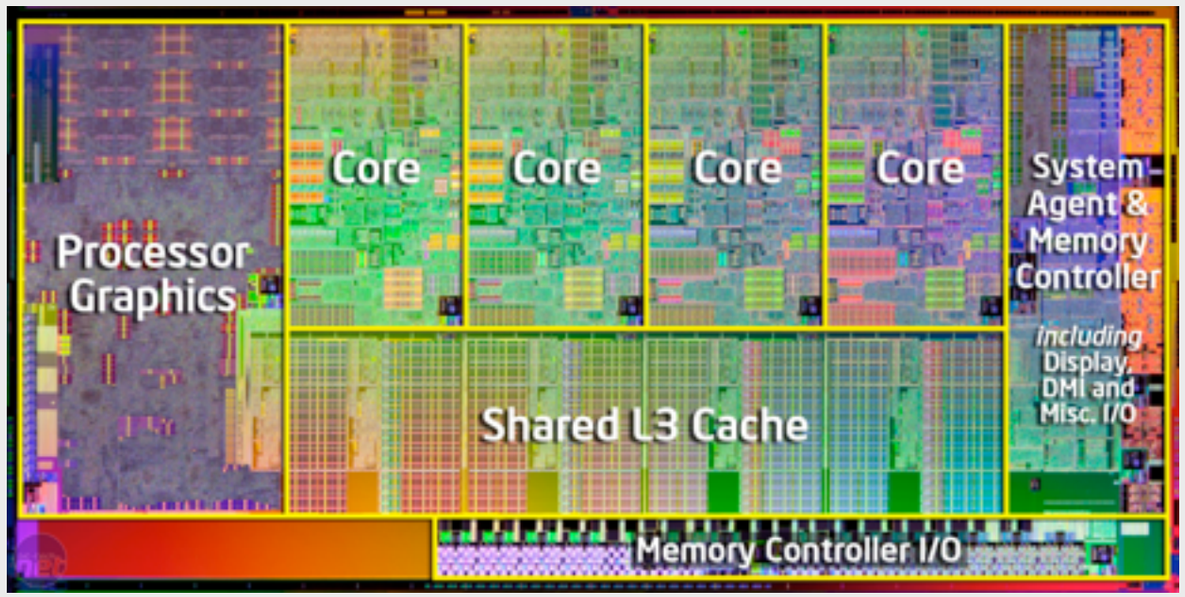
Efficient lock-free sets, maps, *etc.*

Implement your own LVars, too

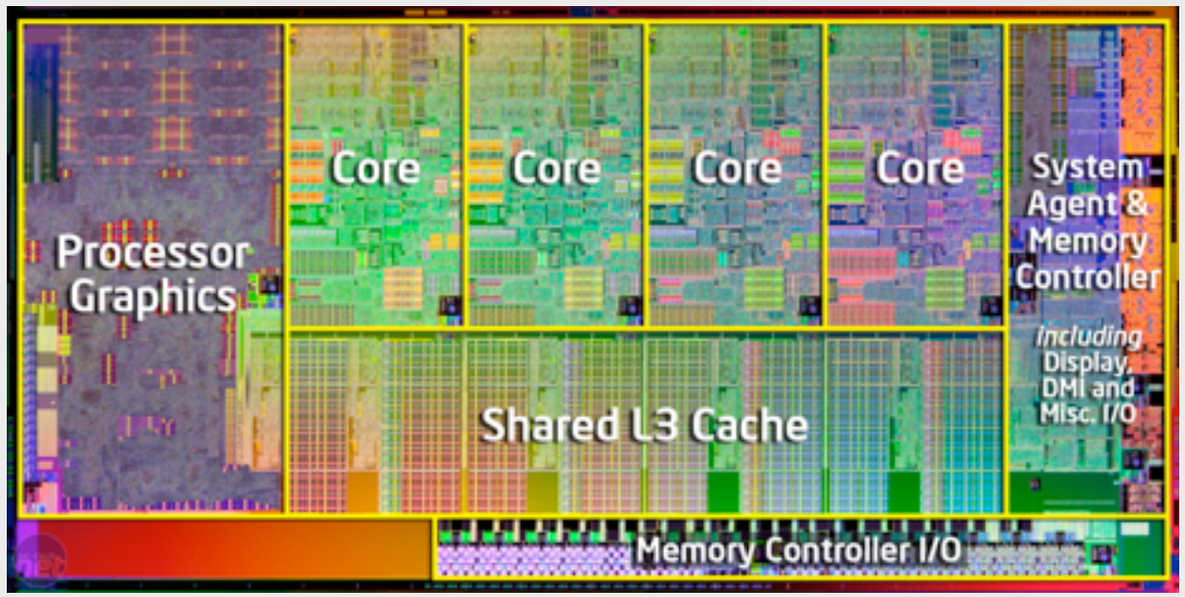
```
p :: HasPut e =>
     Par e s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

`hackage.haskell.org/package/lvish`

`github.com/iu-parfunc/lvars`

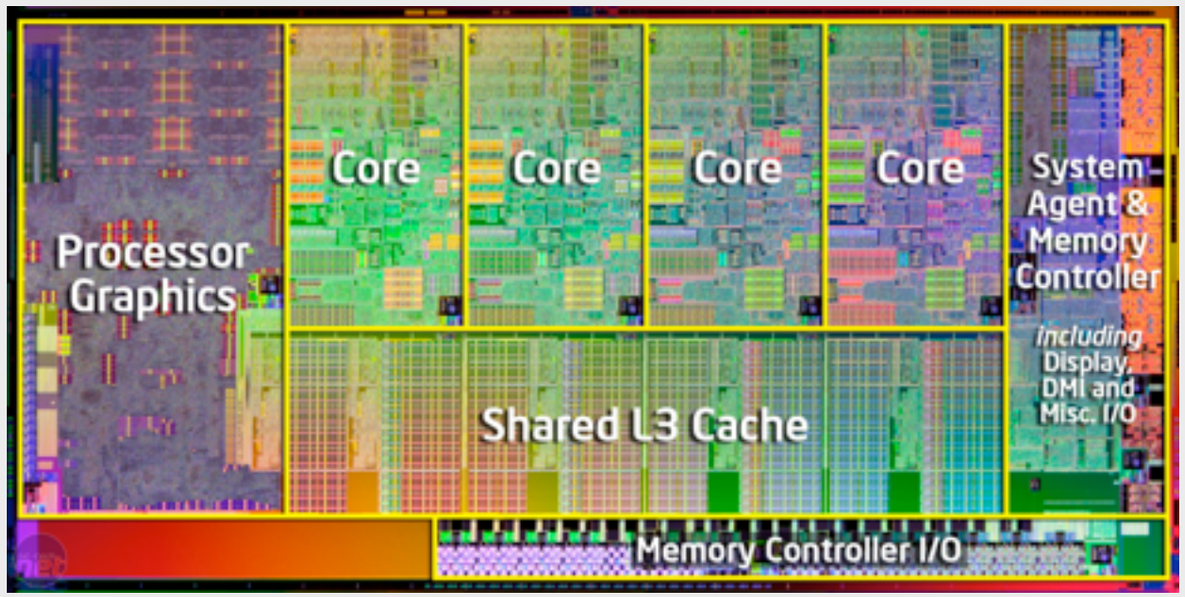


Deterministic Parallel Programming



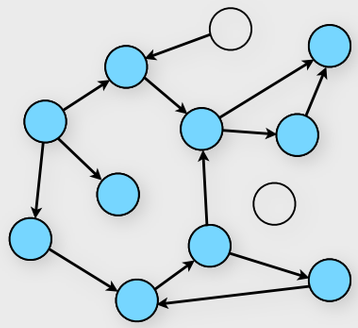
(observably)

Deterministic Parallel Programming

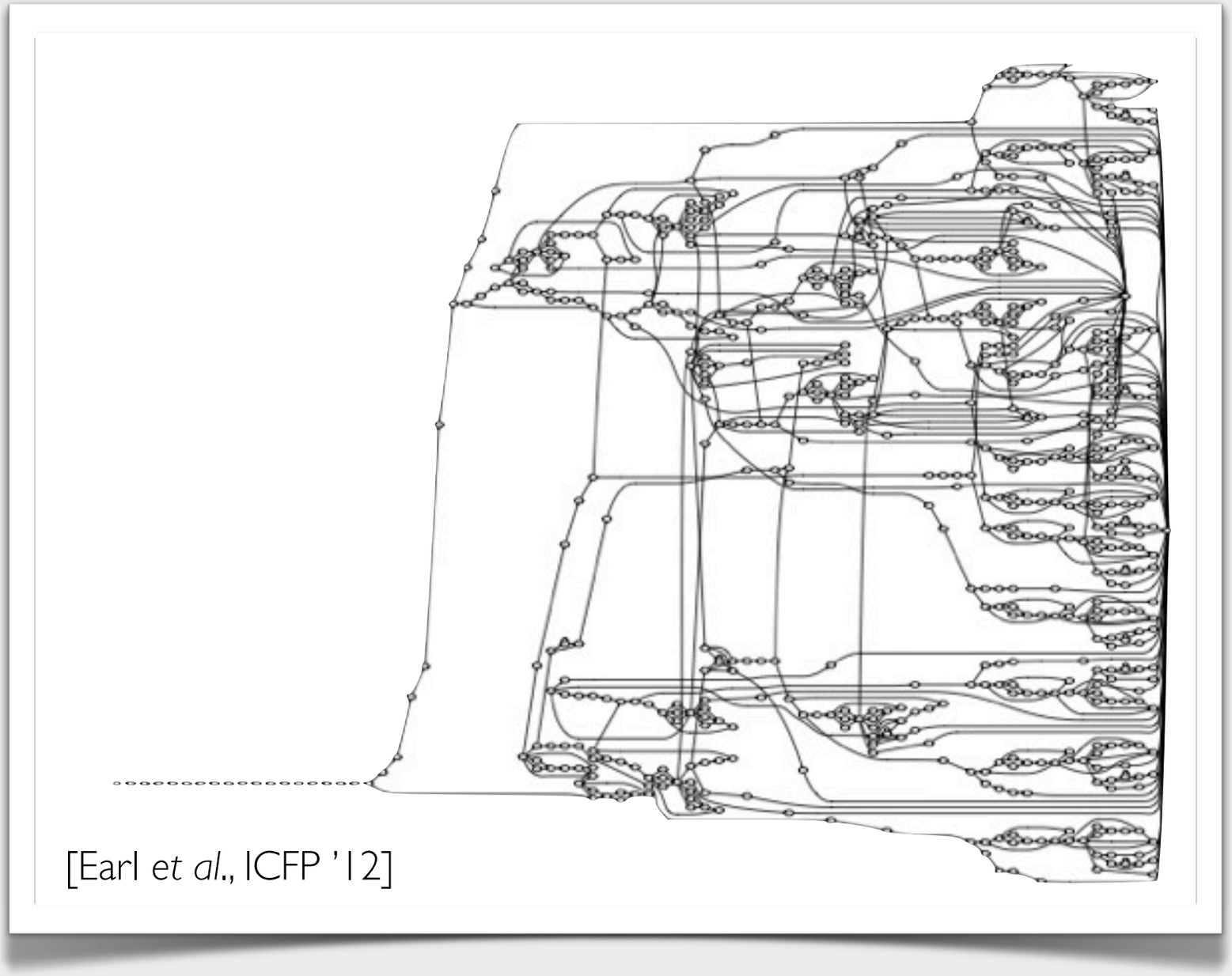


(observably) *(irregular)*

Deterministic Parallel Programming



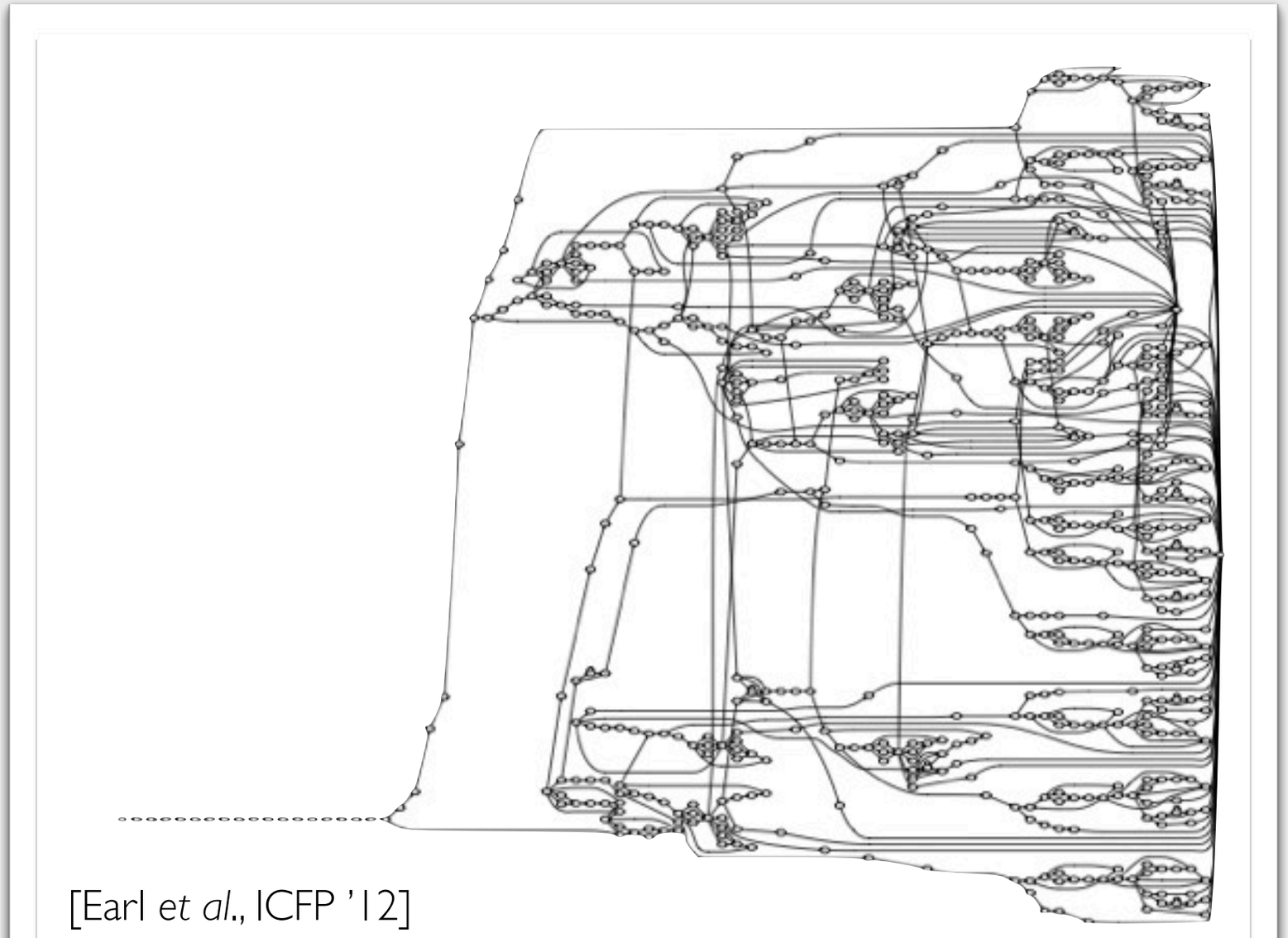
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]



[Earl *et al.*, ICFP '12]

Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

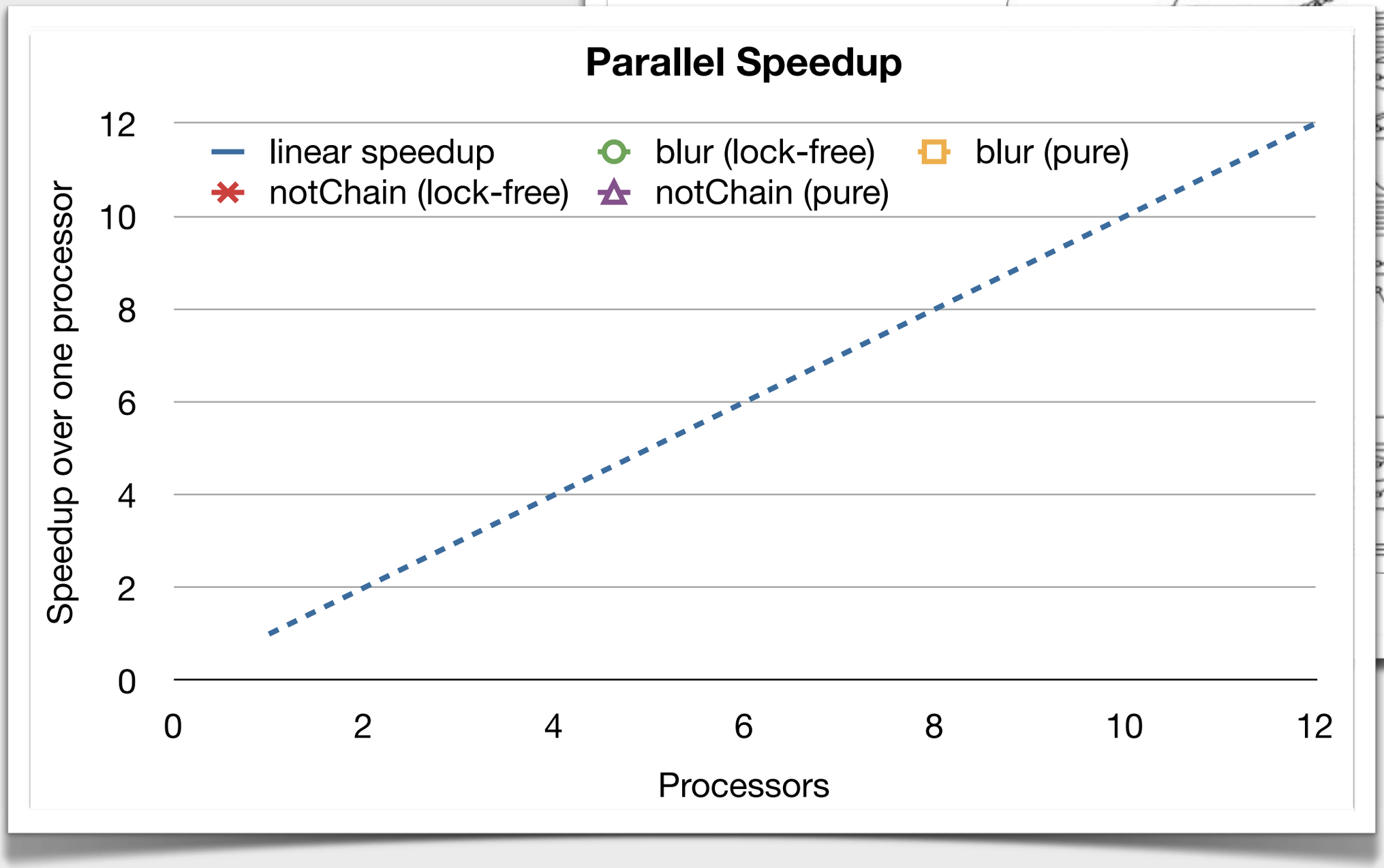
- up to 25x speedup, *even on one core*, from not having to copy data!



[Earl *et al.*, ICFP '12]

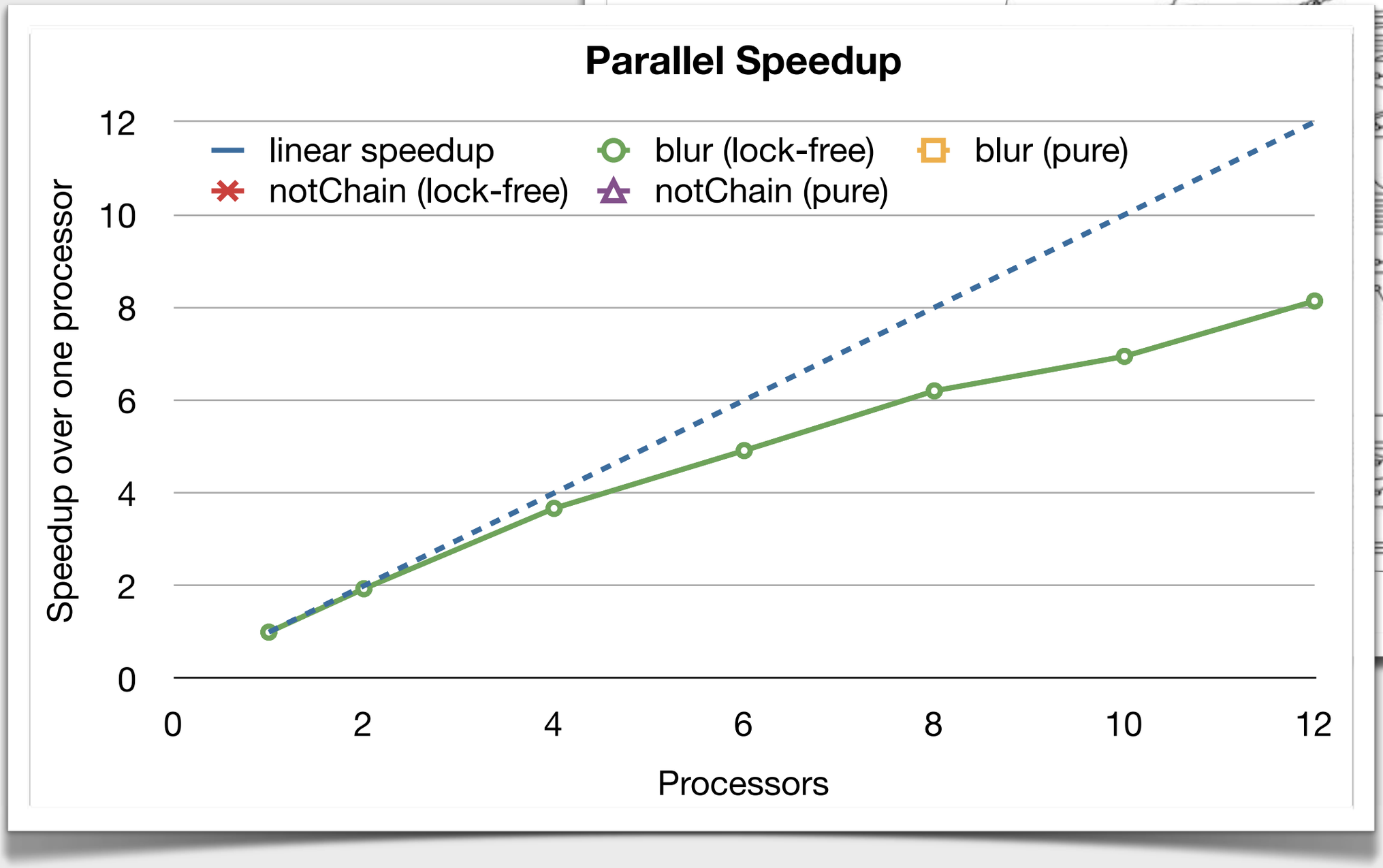
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

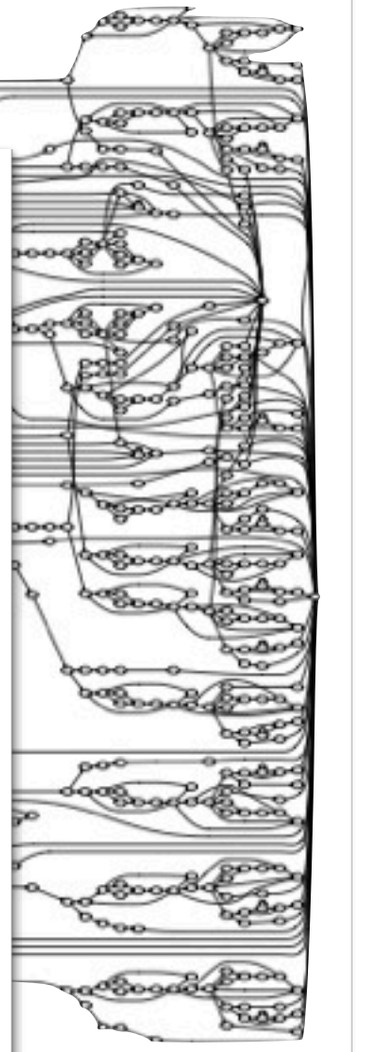
- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup



Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

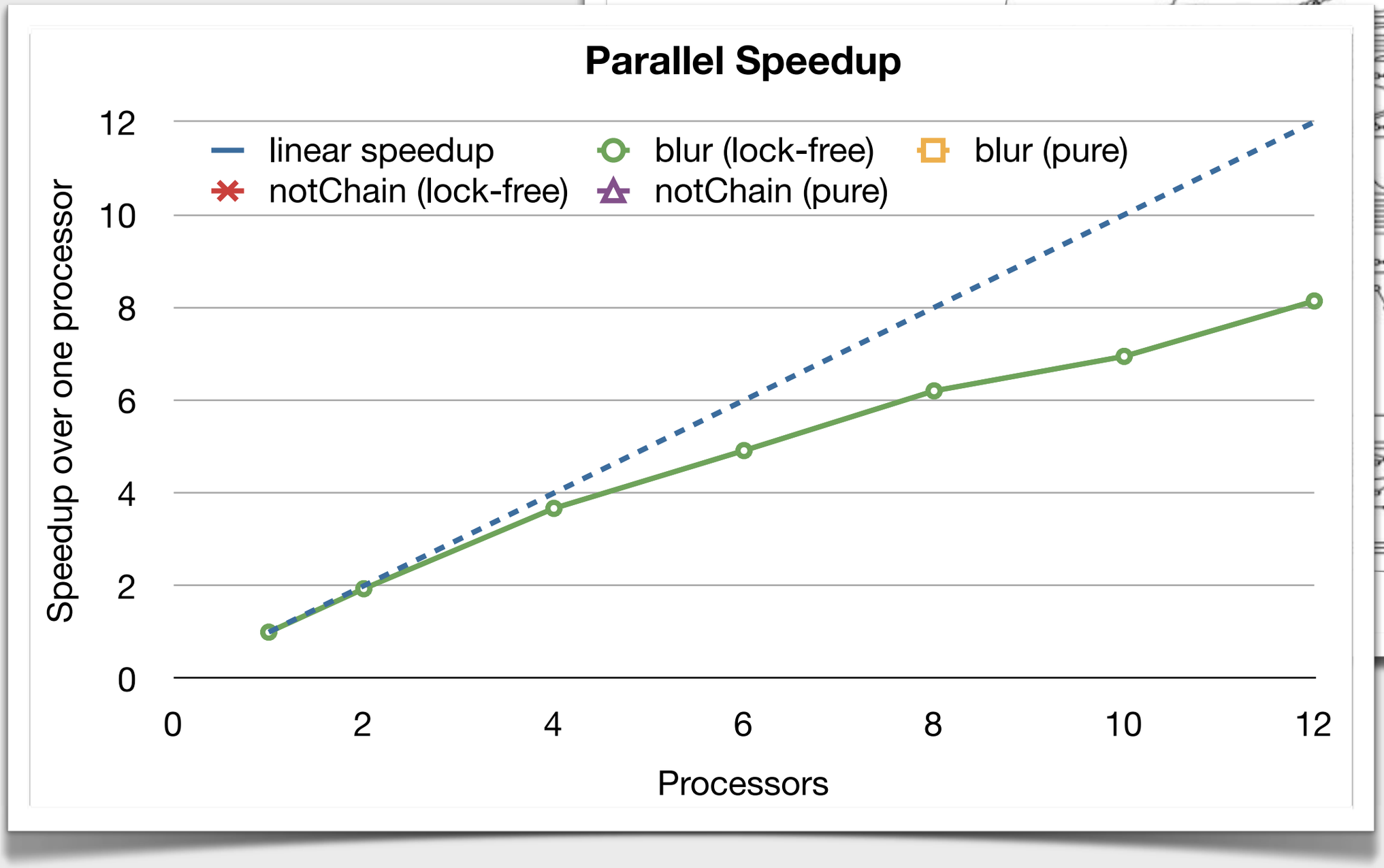
- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup

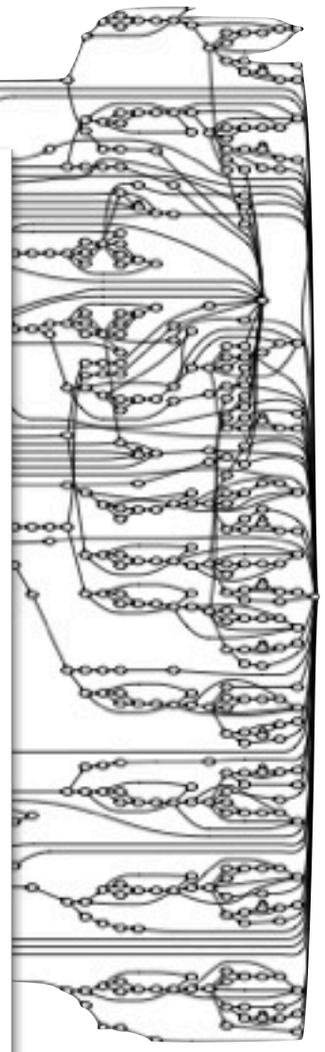




Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

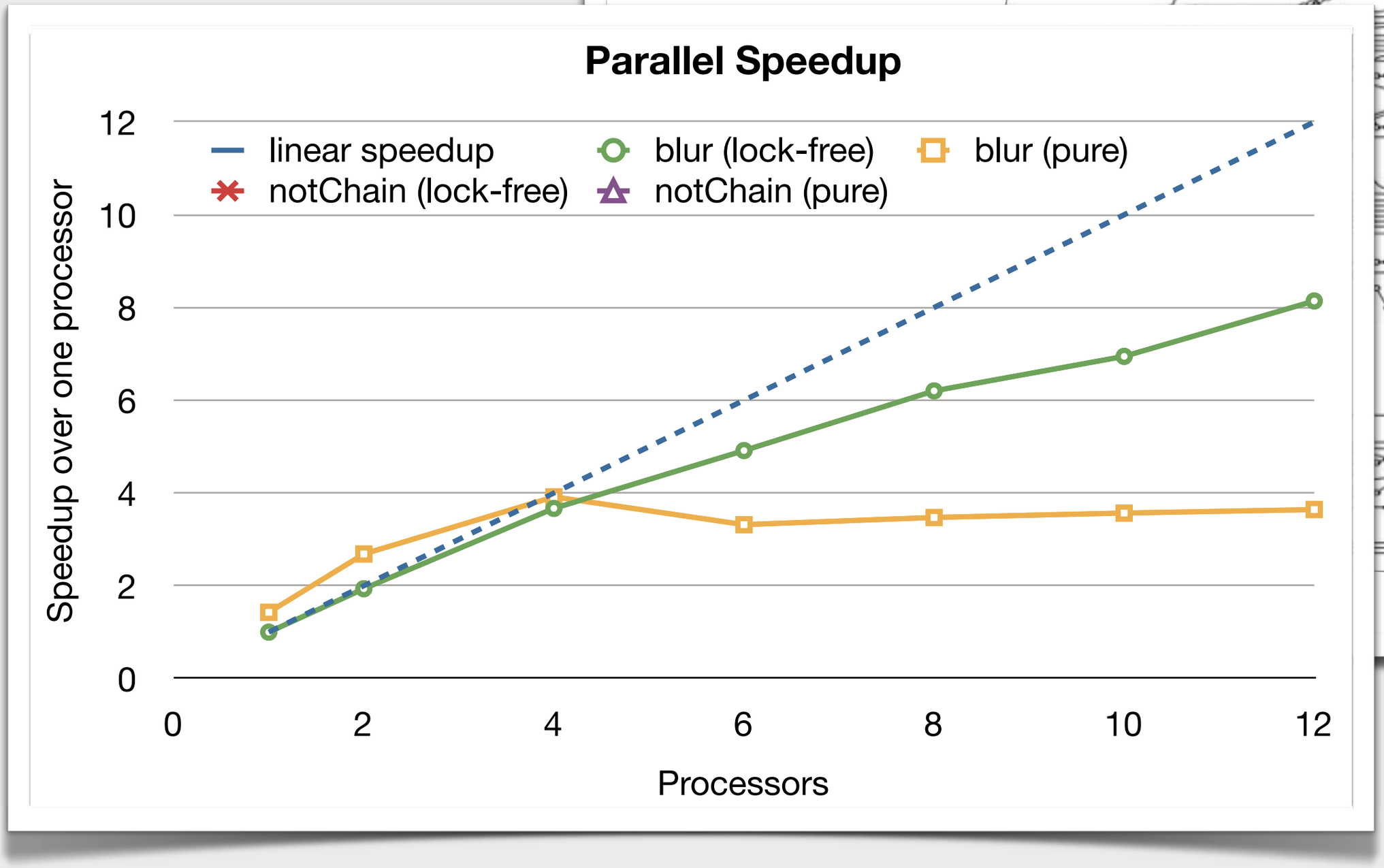
- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help

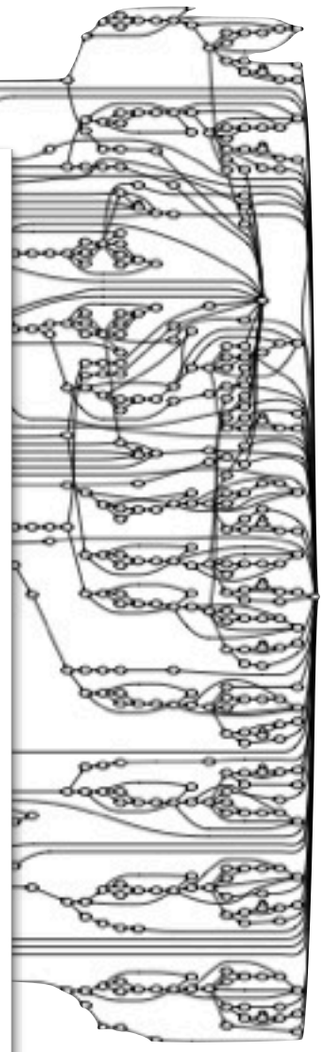




Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

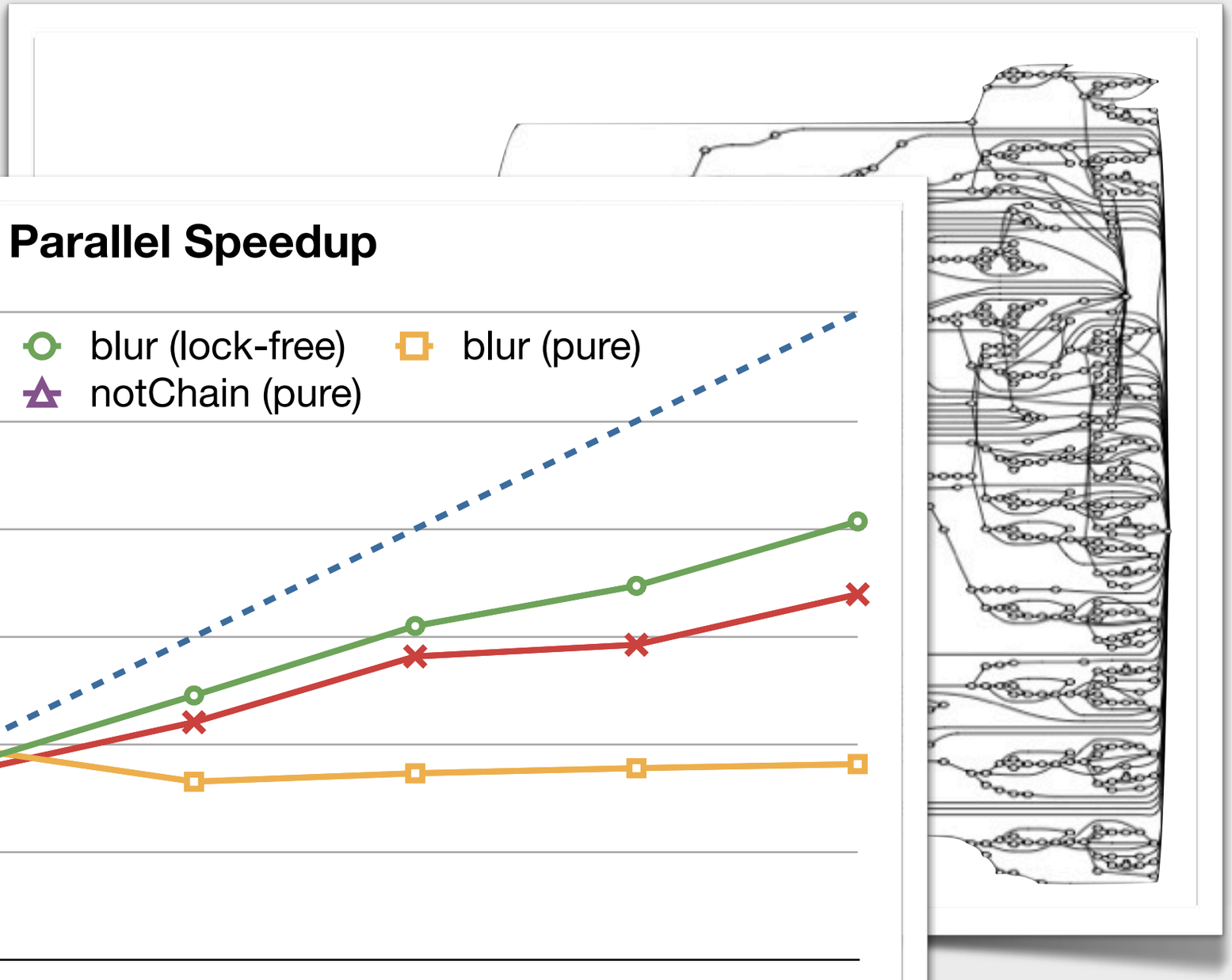
- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help





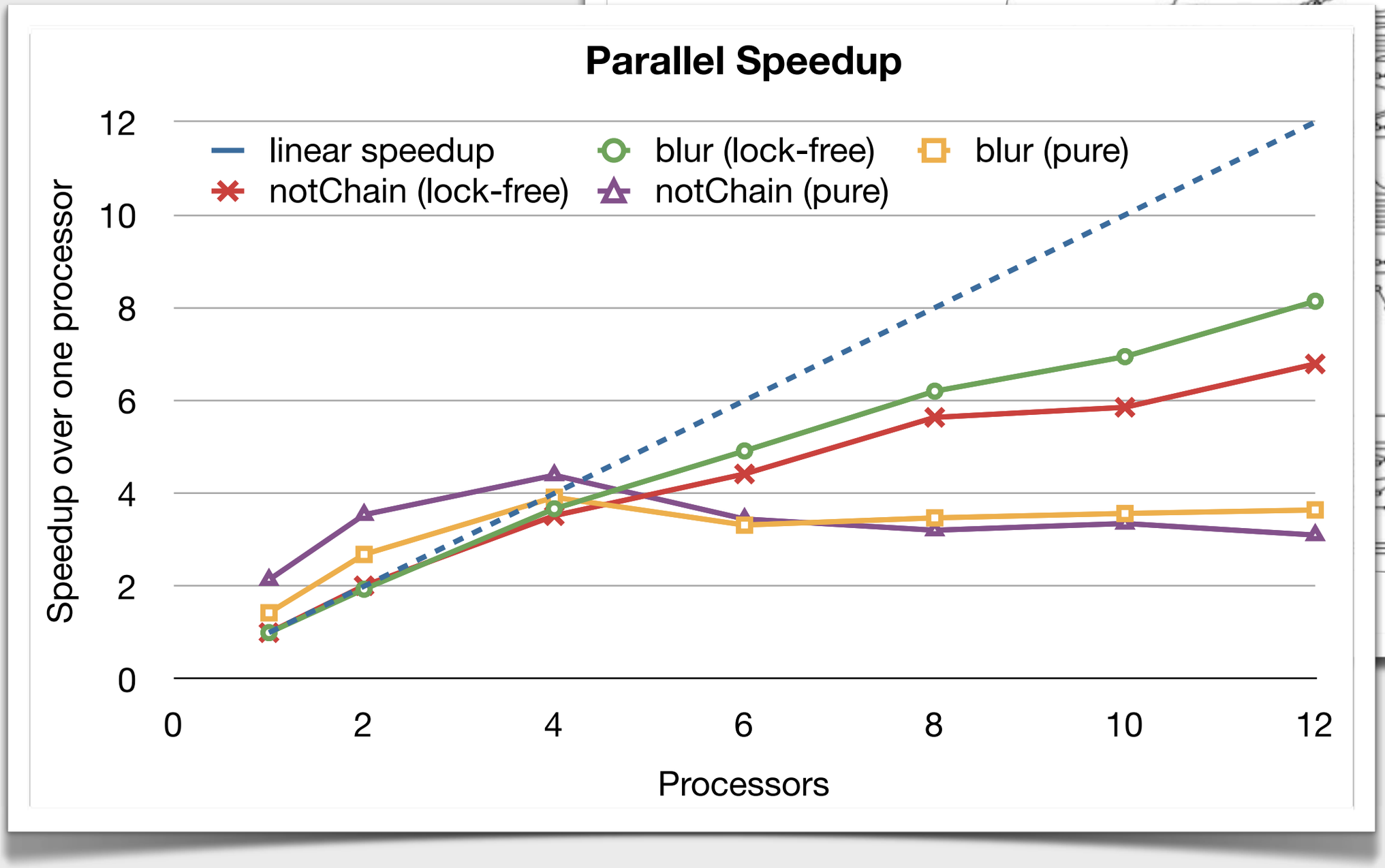
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

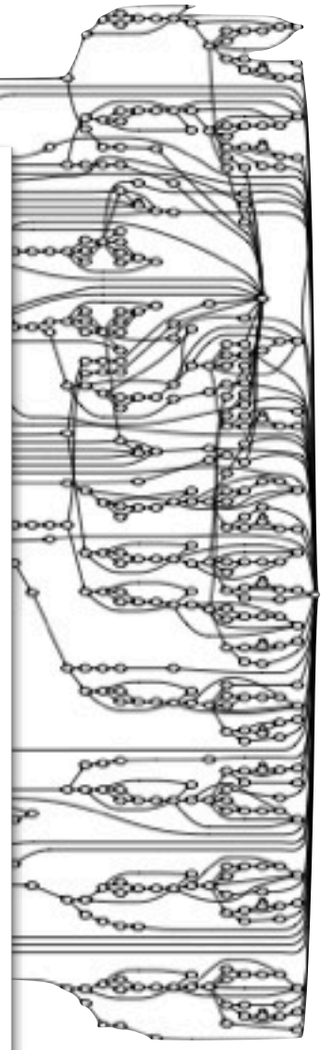
- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help



Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

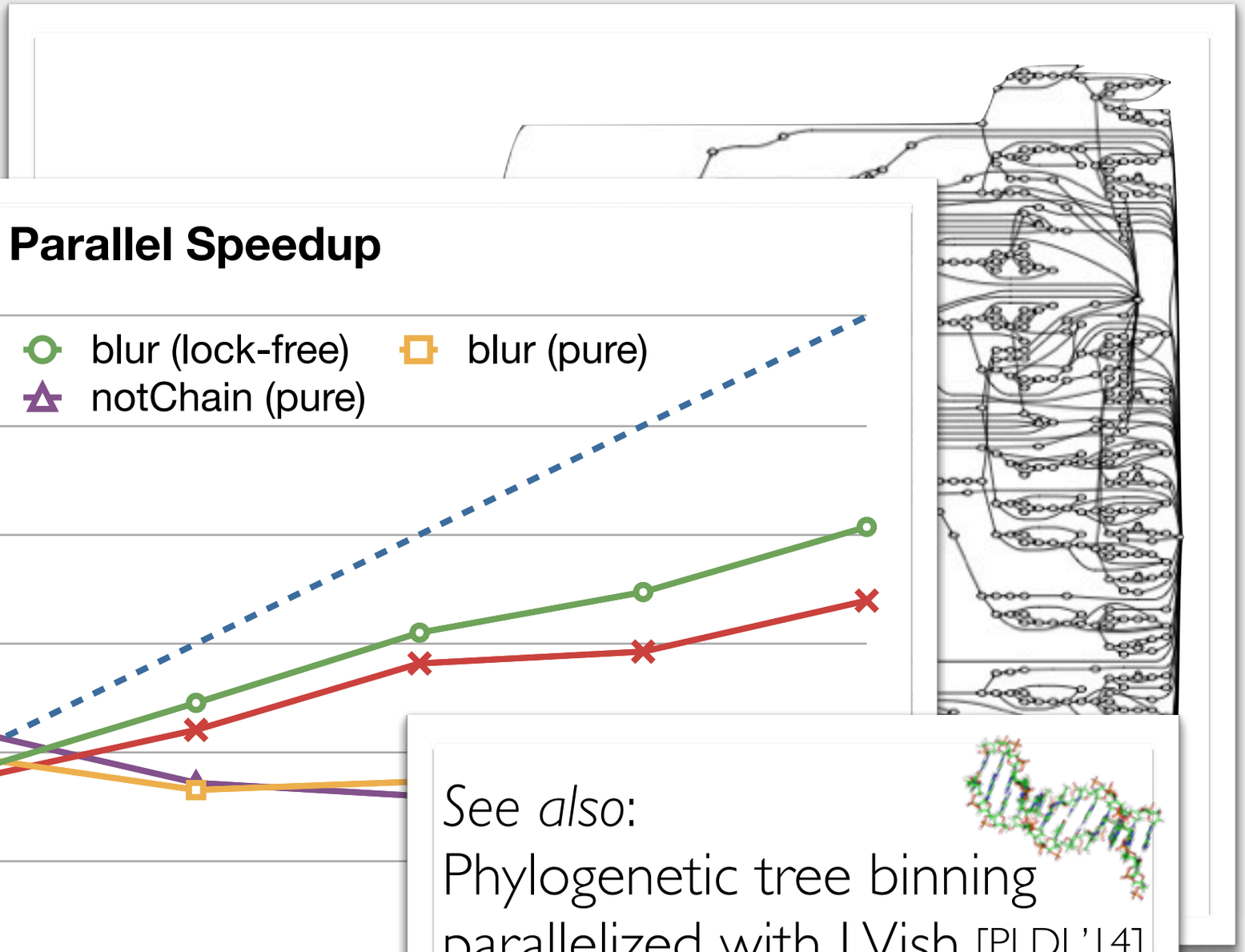
- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help





Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

- up to 25x speedup, *even on one core*, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help

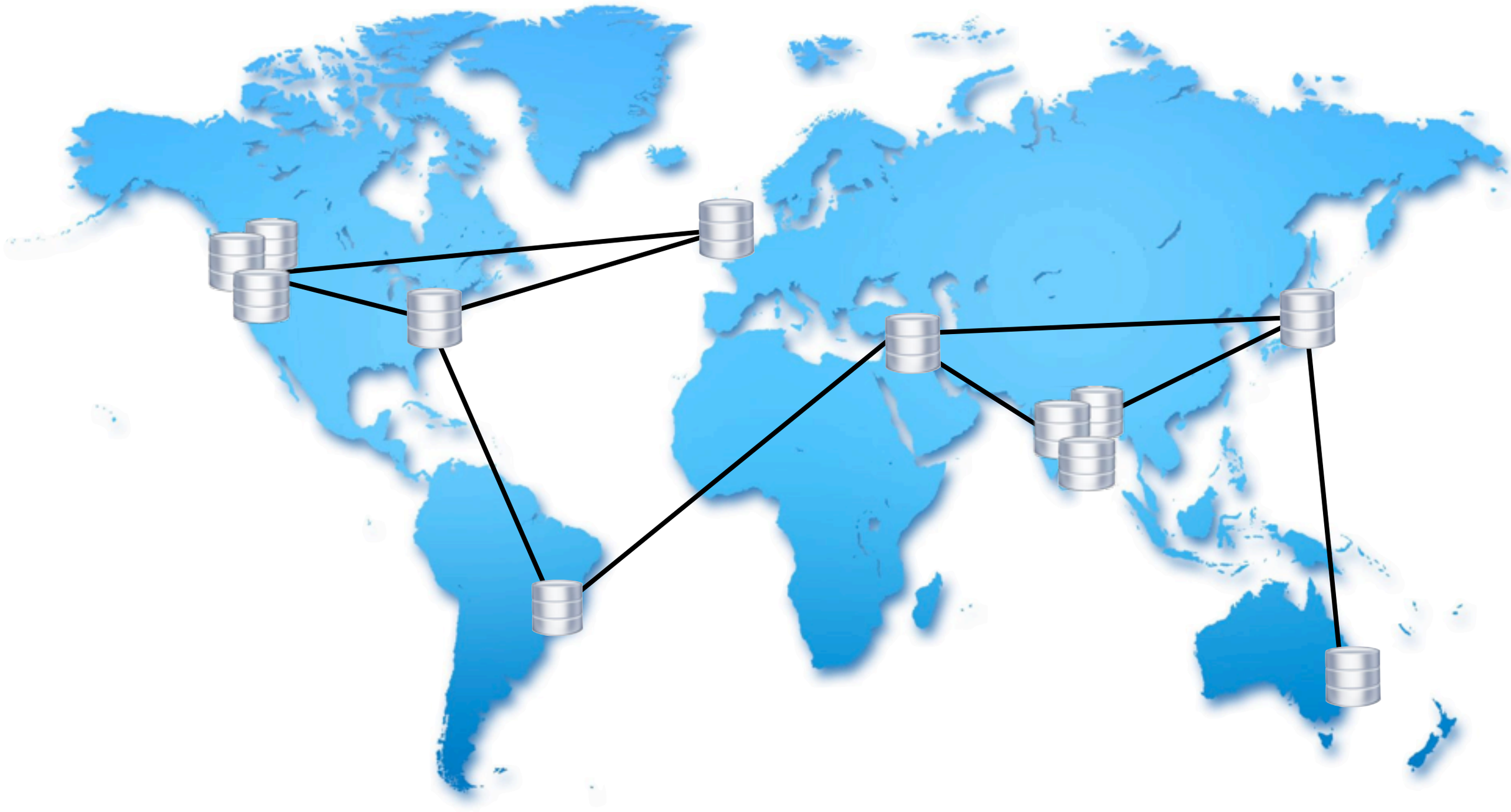


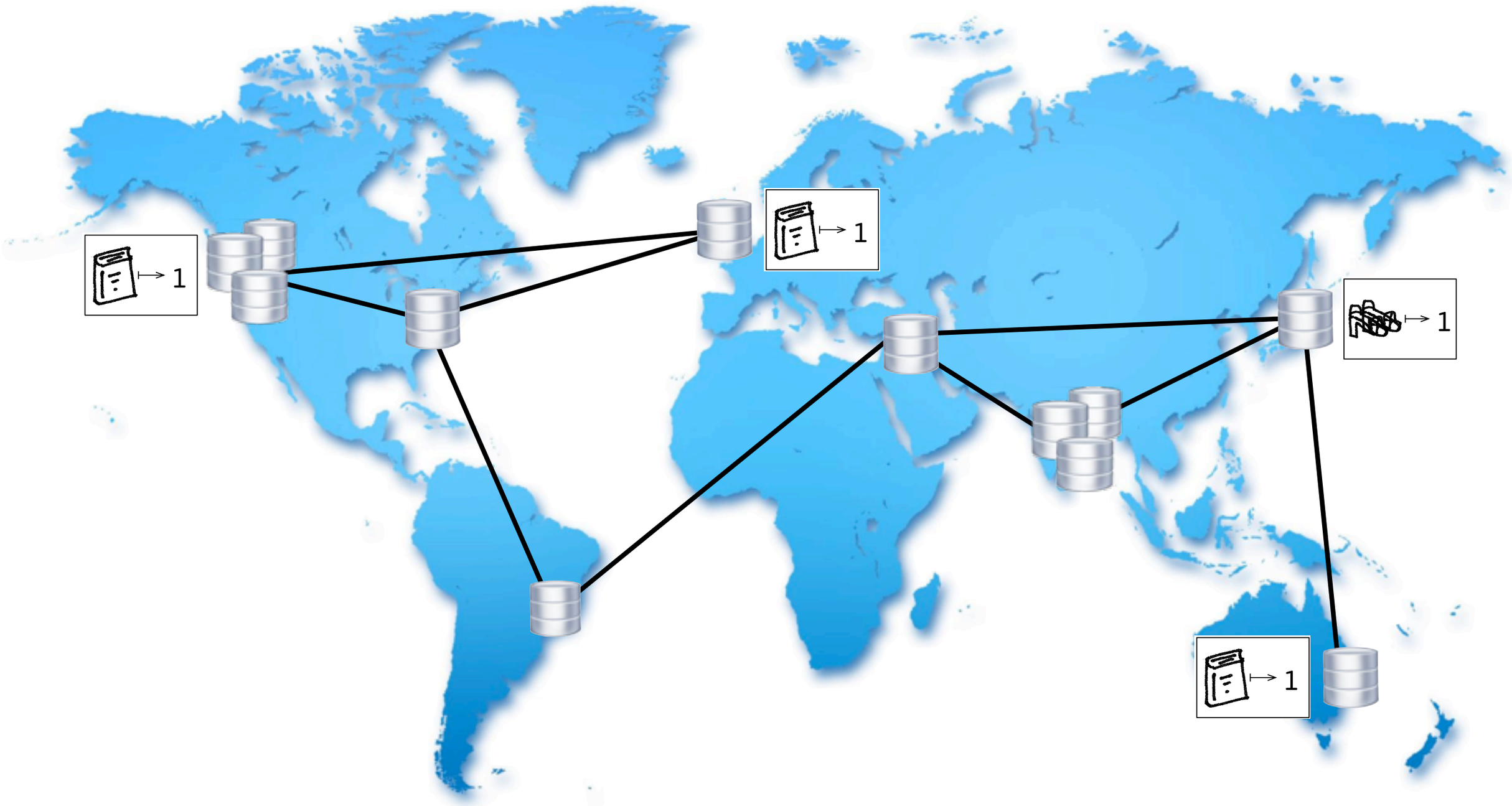
See *also*:
Phylogenetic tree binning parallelized with LVish [PLDI '14]

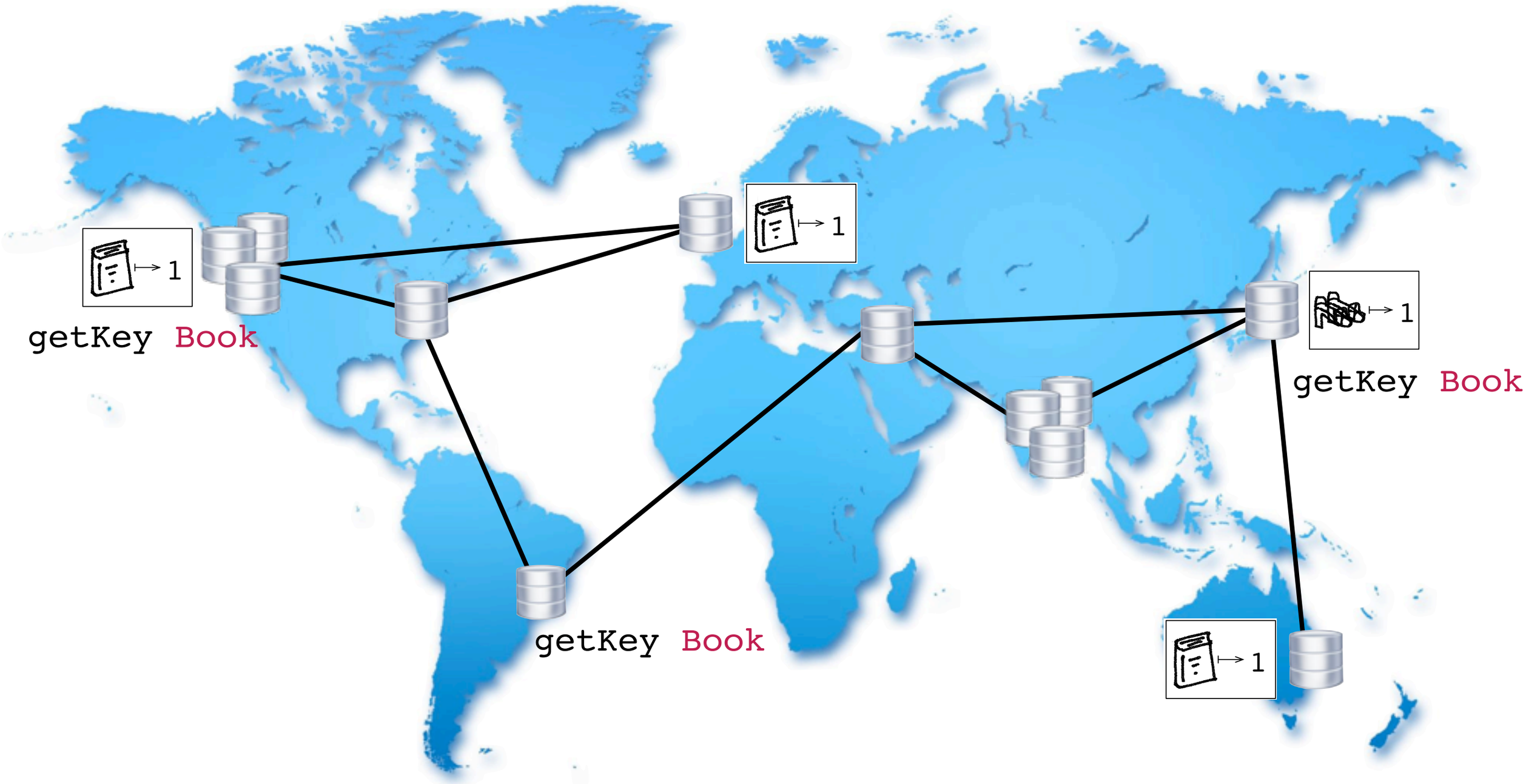
aka "LVars"

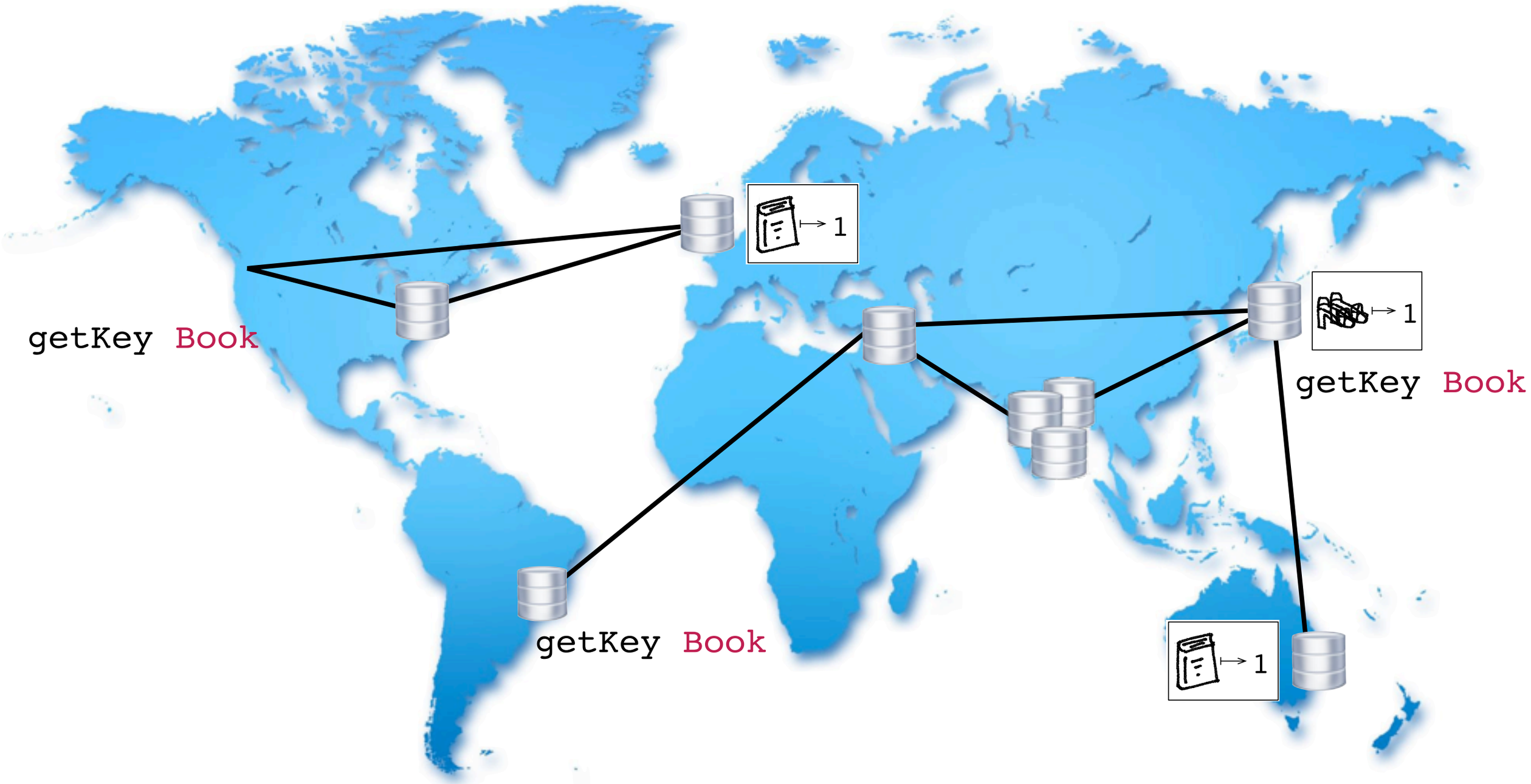
Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.

Distributed systems




1
1
1
1


1
getKey Book
1
getKey Book
1
getKey Book
getKey Book
1


getKey Book
getKey Book
getKey Book
1
1
1

Eventual consistency.

"if updates stop arriving, replicas will eventually agree"

Eventual consistency...but how?

"if updates stop arriving,
replicas will eventually agree"

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

[DeCandia *et al.*, SOSP '07]

since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

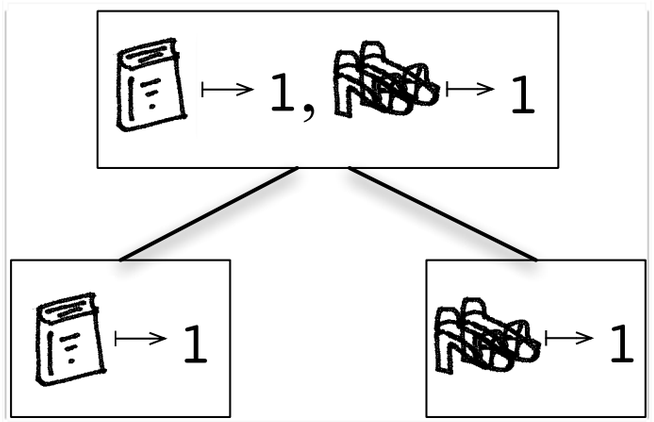
Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as [illegible] S3) is probably the best known. This paper presents the [illegible]ementation of Dynamo, another highly available [illegible]ributed data store built for Amazon's platform. [illegible] to manage the state of services that have very [illegible] requirements and need tight control over the [illegible]en availability, consistency, cost-effectiveness and [illegible]Amazon's platform has a very diverse set of [illegible] different storage requirements. A select set of [illegible]ires a storage technology that is flexible enough [illegible] designers configure their data store appropriately [illegible] tradeoffs to achieve high availability and [illegible]rmance in the most cost effective manner.

[illegible] services on Amazon's platform that only need [illegible]ss to a data store. For many services, such as [illegible]de best seller lists, shopping carts, customer [illegible]on management, sales rank, and product catalog, [illegible]rn of using a relational database would lead to [illegible] limit scale and availability. Dynamo provides a [illegible]ey only interface to meet the requirements of [illegible].

[illegible] ynthesis of well known techniques to achieve [illegible] vailability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

[DeCandia *et al.*, SOSP '07]



since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as S3) is probably the best known. This paper presents the ementation of Dynamo, another highly available ributed data store built for Amazon's platform. to manage the state of services that have very requirements and need tight control over the en availability, consistency, cost-effectiveness and Amazon's platform has a very diverse set of different storage requirements. A select set of ires a storage technology that is flexible enough designers configure their data store appropriately tradeoffs to achieve high availability and rmance in the most cost effective manner.

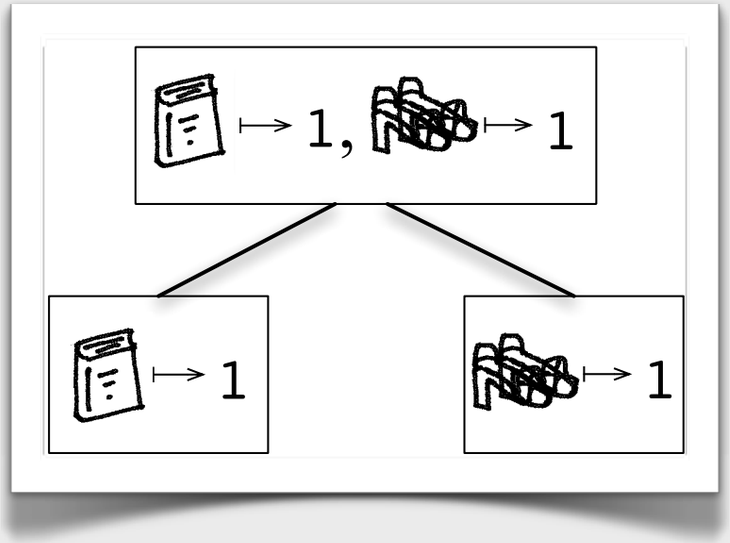
services on Amazon's platform that only need ss to a data store. For many services, such as de best seller lists, shopping carts, customer on management, sales rank, and product catalog, rn of using a relational database would lead to limit scale and availability. Dynamo provides a ey only interface to meet the requirements of .

ynthesis of well known techniques to achieve vailability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

[DeCandia *et al.*, SOSP '07]

Convergent replicated data types (CvRDTs)

[Shapiro *et al.*, 2011]



since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

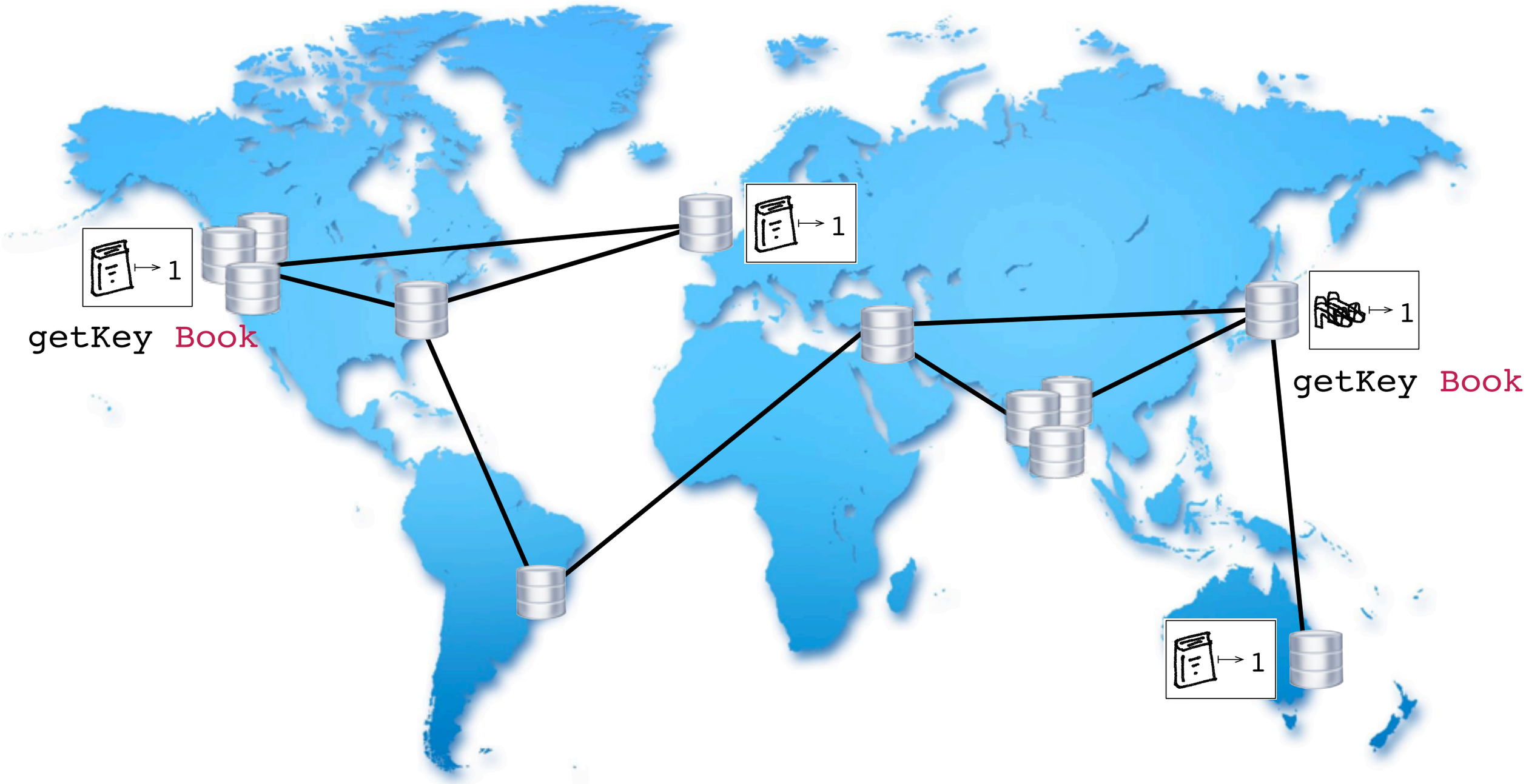
Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

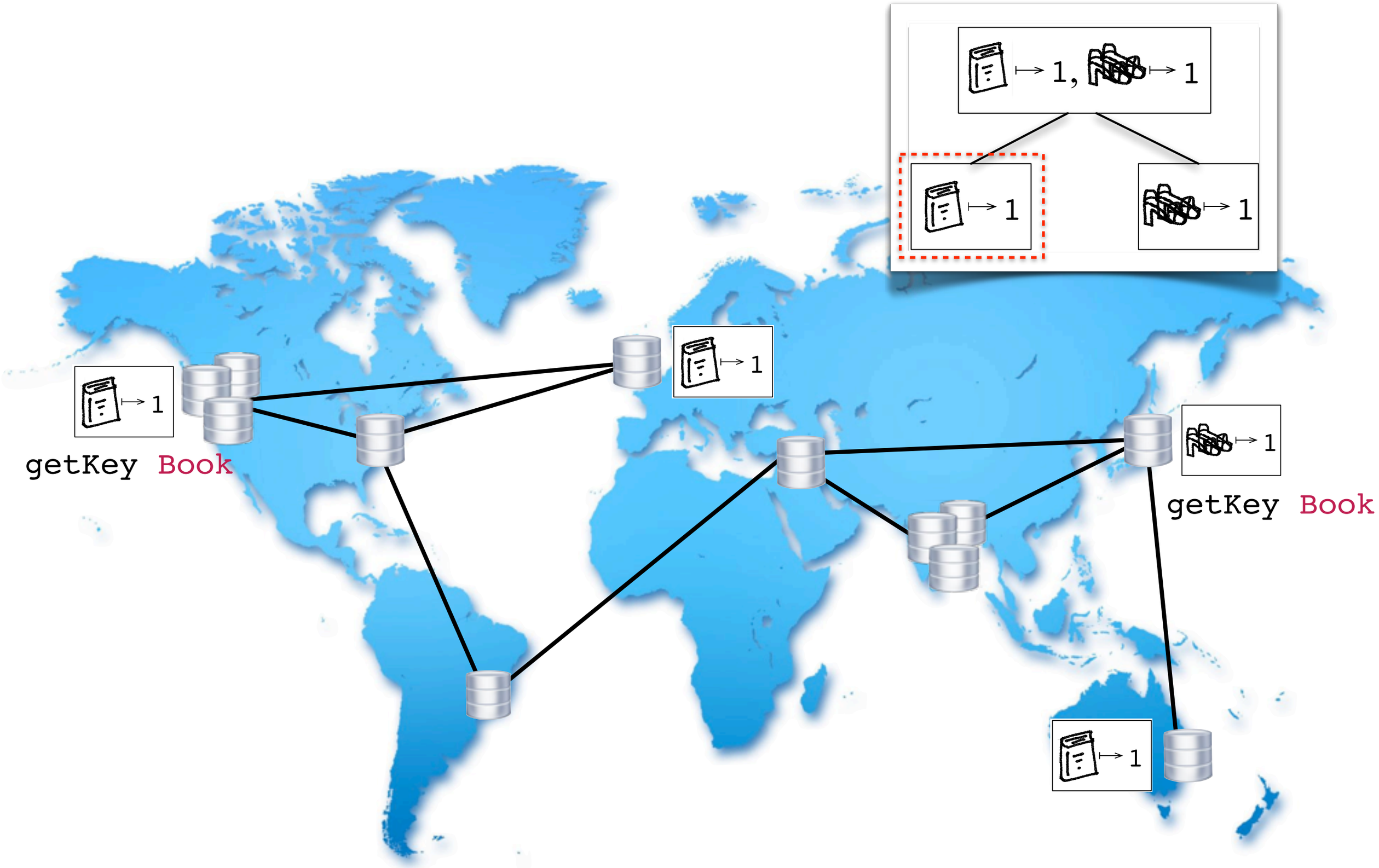
To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as S3) is probably the best known. This paper presents the ementation of Dynamo, another highly available ributed data store built for Amazon's platform. to manage the state of services that have very requirements and need tight control over the en availability, consistency, cost-effectiveness and Amazon's platform has a very diverse set of different storage requirements. A select set of ires a storage technology that is flexible enough designers configure their data store appropriately tradeoffs to achieve high availability and rmance in the most cost effective manner.

services on Amazon's platform that only need ss to a data store. For many services, such as de best seller lists, shopping carts, customer on management, sales rank, and product catalog, rn of using a relational database would lead to limit scale and availability. Dynamo provides a ey only interface to meet the requirements of .

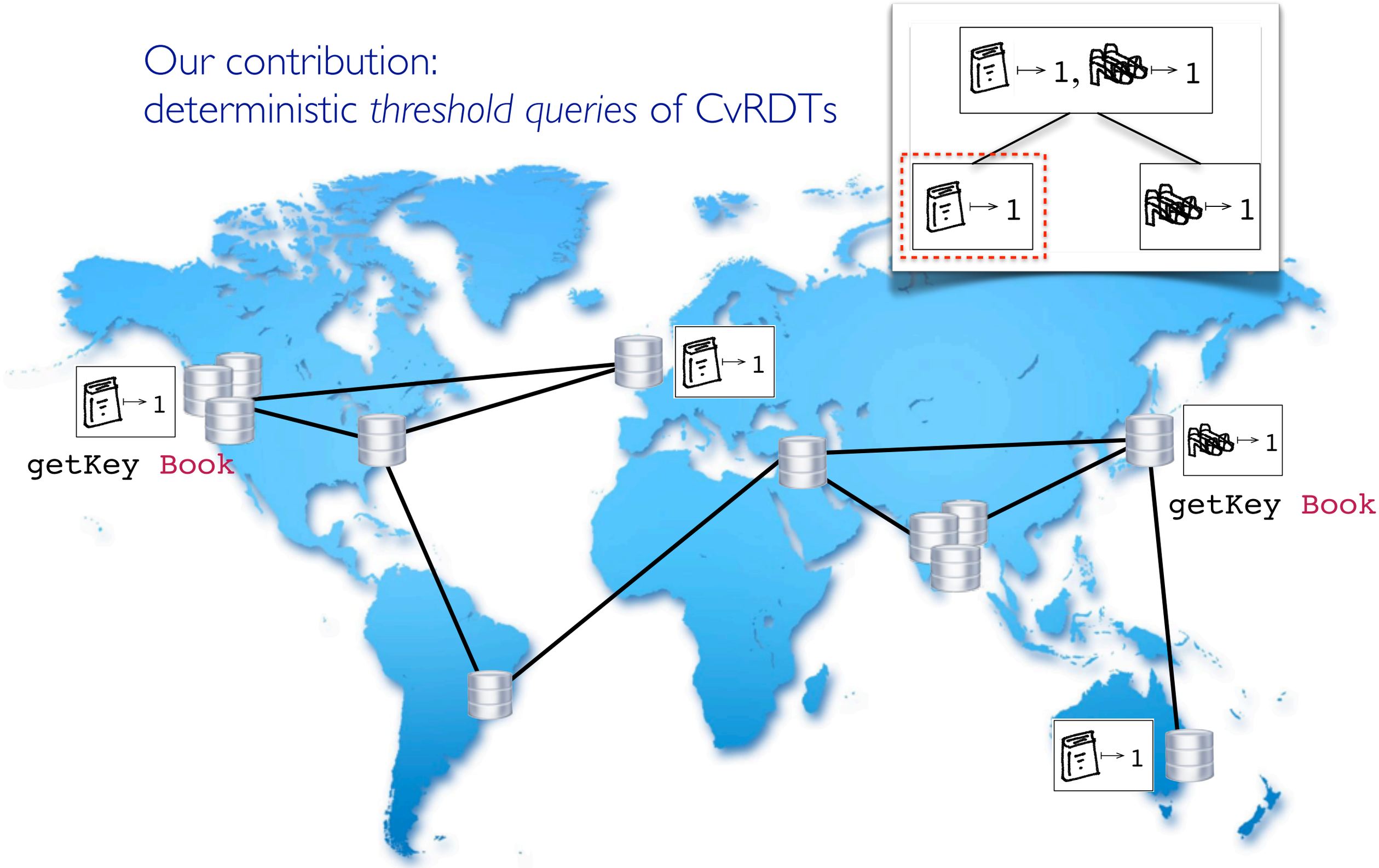
ynthesis of well known techniques to achieve vailability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

[DeCandia *et al.*, SOSP '07]

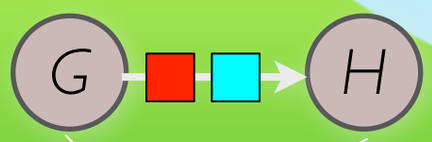

1
getKey Book
1
1
getKey Book
1


1, 1
1
1
1
1
1
getKey Book
1
getKey Book

Our contribution:
deterministic *threshold queries* of CvRDTs



LVars and LVish across the landscape:



Kahn process networks

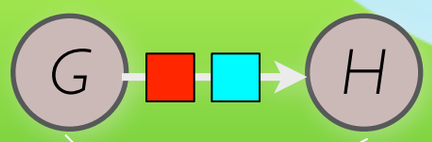
single-assignment

imperative disjoint

λ-calculus

```
f  (g  x)
(h  y)
```

LVars and LVish across the landscape:



Kahn process networks

single-assignment

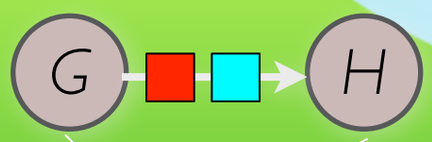
λ-calculus

```
f (g x)
(h y)
```

imperative disjoint

```
g(left)  h(rght)
```

LVars and LVish across the landscape:



Kahn process networks

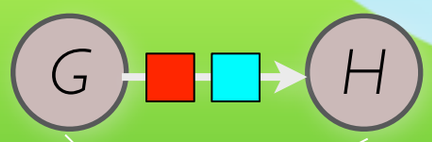
single-assignment

imperative disjoint

λ-calculus

```
f  (g  x)
(h  y)
```

LVars and LVish across the landscape:



Kahn process networks

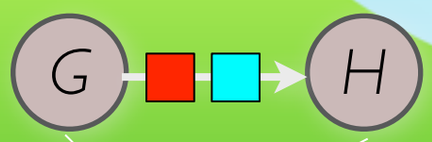
single-assignment

imperative disjoint

λ-calculus

```
f  (g  x)
(h  y)
```

LVars and LVish across the landscape:

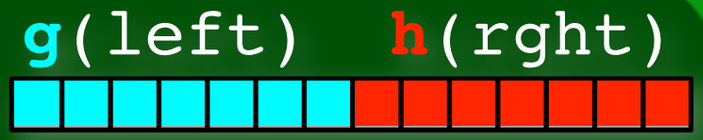


Kahn process networks

single-assignment

quasi-det.

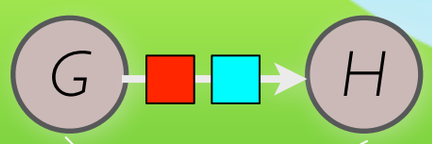
imperative disjoint



λ-calculus

```
f  (g  x)
   (h  y)
```

LVars and LVish across the landscape:



Kahn process networks

single-assignment

quasi-det.

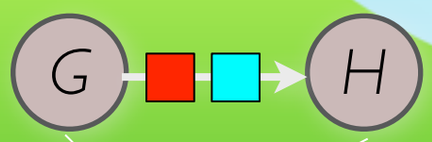
imperative disjoint [PLDI '14]

λ-calculus

```
f  (g  x)
   (h  y)
```

threshold-readable CvRDTs

LVars and LVish across the landscape:



Kahn process networks

single-assignment

quasi-det.

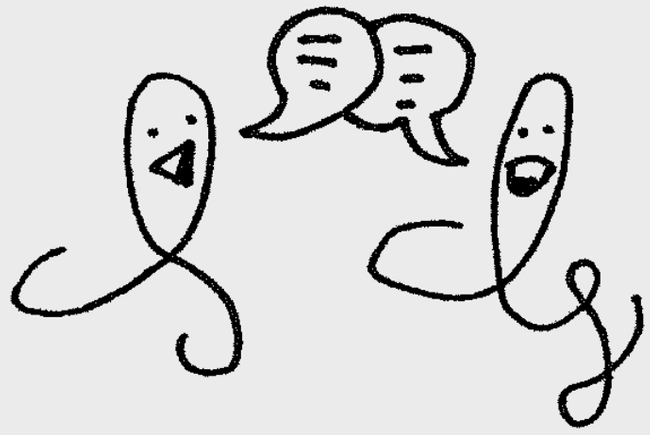
imperative disjoint [PLDI '14]

λ-calculus

```
f (g x)
(h y)
```

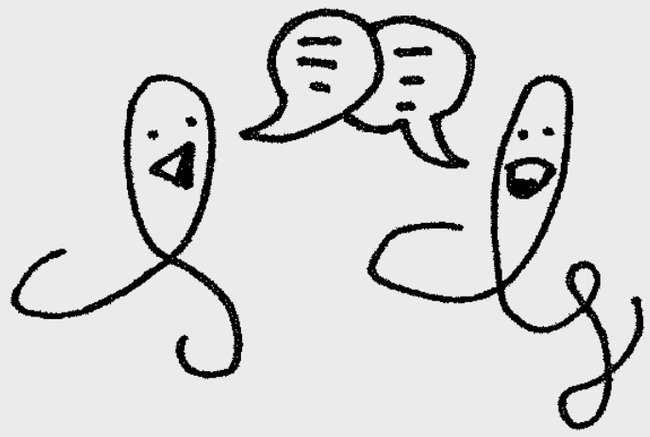
aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.



aka "LVars"

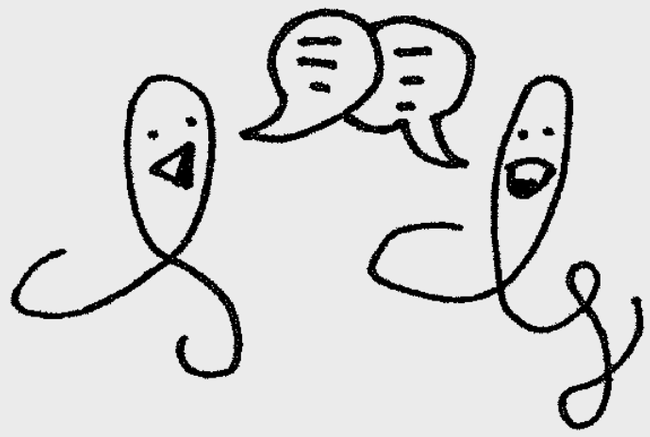
Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.



`github.com/lkuper/dissertation`

aka "LVars"

Lattice-based data structures
are a general and practical unifying abstraction
for deterministic and quasi-deterministic
parallel and distributed programming.



`github.com/lkuper/dissertation`

Thank you!