Transactional Tasks

Futures in Software Transactions

Janwillem Swalens

Overview

1. **Crash course: futures**
2. Crash course: transactions
3. Problem: nesting futures in transactions
4. Solution: transactional tasks

Crash course: Futures for parallelism

(**`future`** `e`) returns `f`

(**`deref`** `f`) returns result of `e`

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fib (- n 1))
          b (fib (- n 2))]
      (+ a b))))
```

Crash course: Futures for parallelism

`(future e)` returns `f`

`(deref f)` returns result of `e`

Property:

semantically transparent

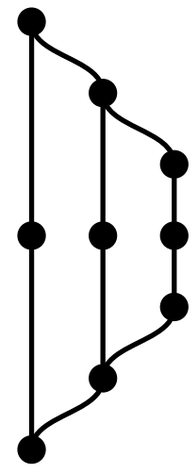
`(future e)` → `e`

`(deref f)` → `f`

is equivalent

(assuming no side-effects in **e**)

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (future (fib (- n 1)))
          b (future (fib (- n 2)))]
      (+ (deref a) (deref b)))))
```



Overview

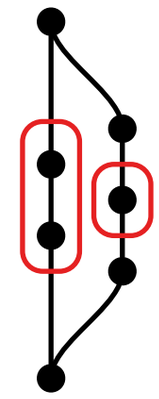
1. Crash course: futures
2. **Crash course: transactions**
3. Problem: nesting futures in transactions
4. Solution: transactional tasks

Crash course: transactions for shared memory

```
(ref v)
(dosync e)
(deref r)
(ref-set r v)
```

Property:
serializability

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```



```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?

T2: 🕐 0

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?
checking	100	

T2: 🕐 0

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?
checking	90	✓

T2: 🕐 0

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	500	

T2: 🕐 0

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

🕐 0

ref	v0		
checking	100		
savings	500		

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

commit T1:

1. copy written values (if no more recent values exist)

🕐 0

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

commit T1:

1. **copy written values** (if no more recent values exist)
2. increase global write-point

🕐 **1**

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

T1 committed

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

commit T2:

1. **copy written values** (if no more recent values exist)
2. increase global write-point

🕐 **2**

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

T2 committed

🕐 2

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

Result is **serializable**. Serialization = T2; T1

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

T1 committed

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	100	
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	90	✓
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

commit T2:

1. **copy written values** (if no more recent values exist)

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	90	✓
savings	500	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

commit T2:

1. copy written values (if no more recent values exist)

🕐 1

ref	v0	v1
checking	100	90
savings	500	510

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 0

ref	val	write?
checking	90	✓
savings	500	

Latest version is more recent than read-point of transaction: **abort**

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 **1**

ref	val	write?

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 1

ref	val	write?
checking	90	
savings	510	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

🕐 1

ref	v0	v1	
checking	100	90	
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 1

ref	val	write?
checking	80	✓
savings	510	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

commit T2:

1. **copy written values** (if no more recent values exist)

🕐 1

ref	v0	v1	v2
checking	100	90	80
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 1

ref	val	write?
checking	80	✓
savings	510	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

commit T2:

1. copy written values (if no more recent values exist)
2. increase global write-point

🕐 **2**

ref	v0	v1	v2
checking	100	90	80
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 1

ref	val	write?
checking	80	✓
savings	510	

```
(def checking (ref 100))
(def savings  (ref 500))
(future
  (dosync
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(future
  (dosync
    (println “You own €” (+ (deref checking)
                            (deref savings)))
    (ref-set checking (- (deref checking) 10))))
```

T2 committed

🕐 2

ref	v0	v1	v2
checking	100	90	80
savings	500	510	

T1: 🕐 0

ref	val	write?
checking	90	✓
savings	510	✓

T2: 🕐 1

ref	val	write?
checking	80	✓
savings	510	

Multi-version concurrency control (MVCC)

Multiple versions of refs stored

More parallelism: T2 can read older version even when T1 has updated the same ref

More memory usage ⇒ keep limited history

Optimization: **barging** detects write–write conflicts early

STM vs locking

```
(def checking (ref 100))
(def savings  (ref 500))
(dosync
  (ref-set checking 10)
  (ref-set savings  20))
```

- no deadlocks or livelocks
- no side-effects (retry)
- more overhead
- transactions can be nested
- “**optimistic**”: only take locks when needed

```
(def checking (atom 100))
(def savings  (atom 500))
(locking checking
  (locking savings
    (reset! checking 10)
    (reset! savings  20)))
```

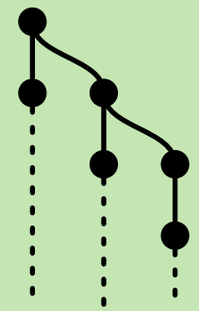
- prone to deadlocks: lock in the right order
- side-effects OK
- less overhead
- nesting can lead to deadlocks
- “**pessimistic**”: all locks at the start

Overview

1. Crash course: futures
2. Crash course: transactions
3. **Problem: nesting futures in transactions**
4. Solution: transactional tasks

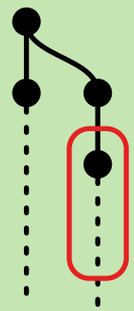
Nesting futures & transactions

```
(future
   (future
      …))
```



Nested task parallelism

```
(future
   (dosync
      …))
```

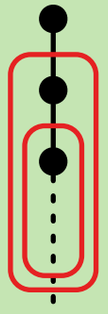


Transactions

```
(dosync
   (future
      …))
```

In-transaction parallelism

```
(dosync
   (dosync
      …))
```

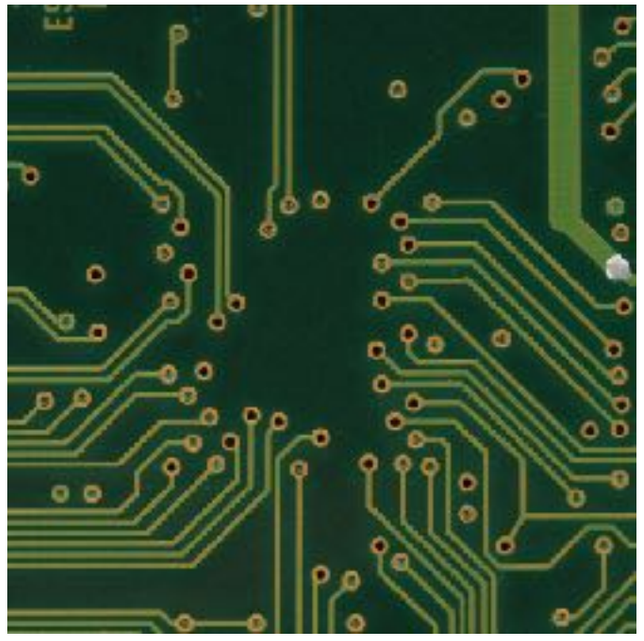


Nested transactions (open/closed)

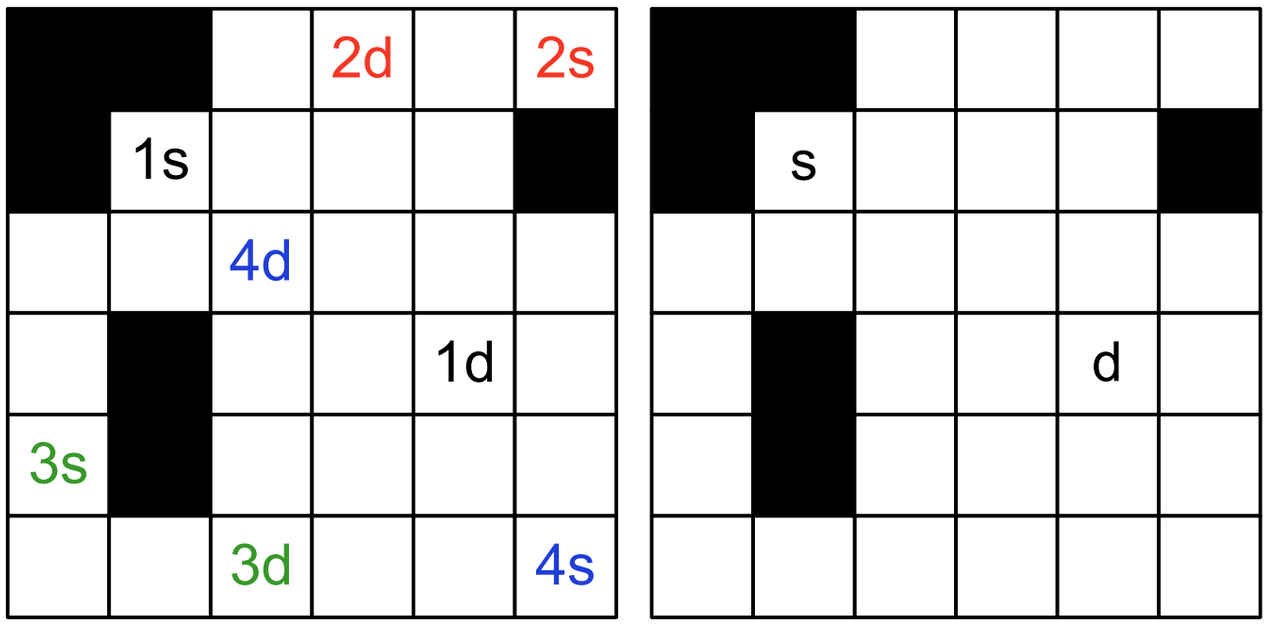
Example: Labyrinth

■	■		2d		2s
■	1s				■
		4d			
	■			1d	
3s	■				
		3d			4s

■	■		2d	2	2s
■	1s	1	1	1	■
		4d		1	
	■	4		1d	
3s	■	4	4	4	4
3	3	3d			4s



Example: Labyrinth

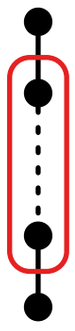



```
(for [[src dst] input-pairs]
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

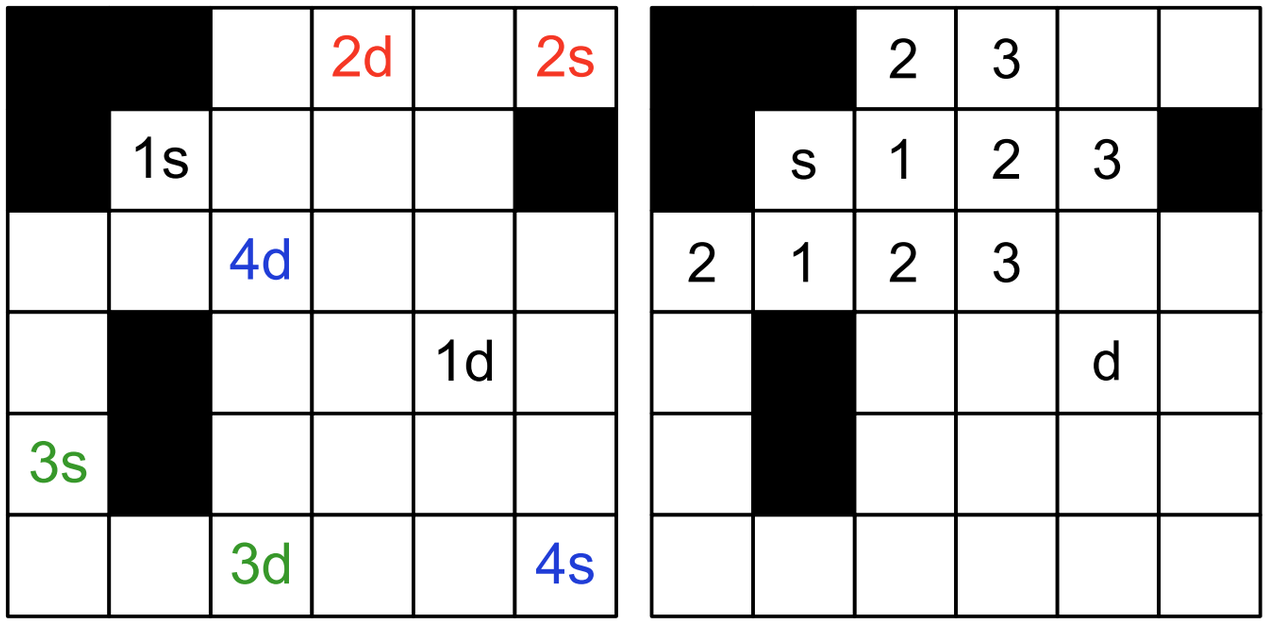
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

```
(defn expand-point [pt grid]
  (dosync
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

breadth-first search



Example: Labyrinth

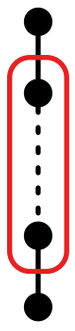



```
(for [[src dst] input-pairs]
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

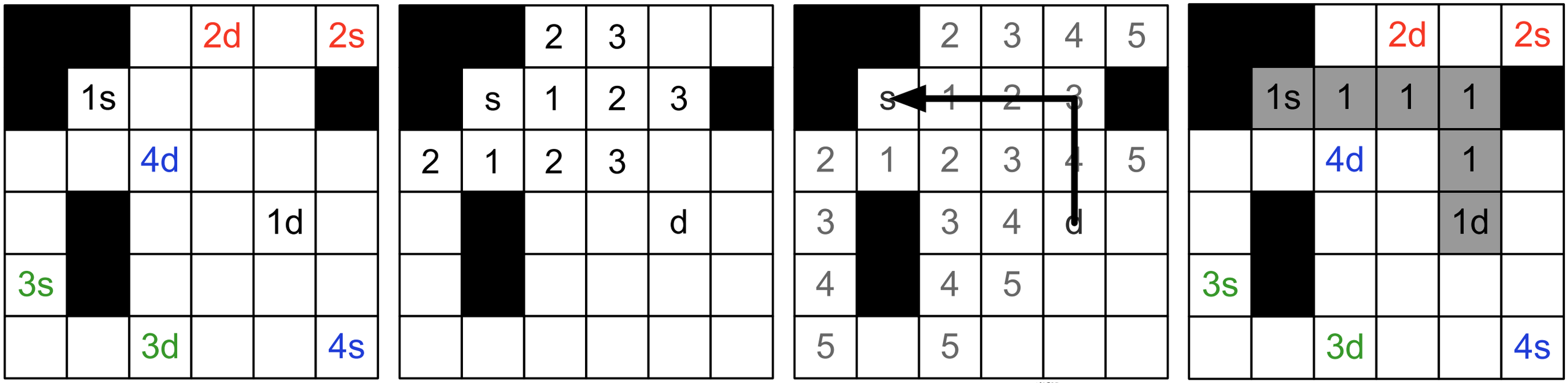
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

```
(defn expand-point [pt grid]
  (dosync
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

breadth-first search



Example: Labyrinth

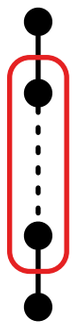



```
(for [[src dst] input-pairs]
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

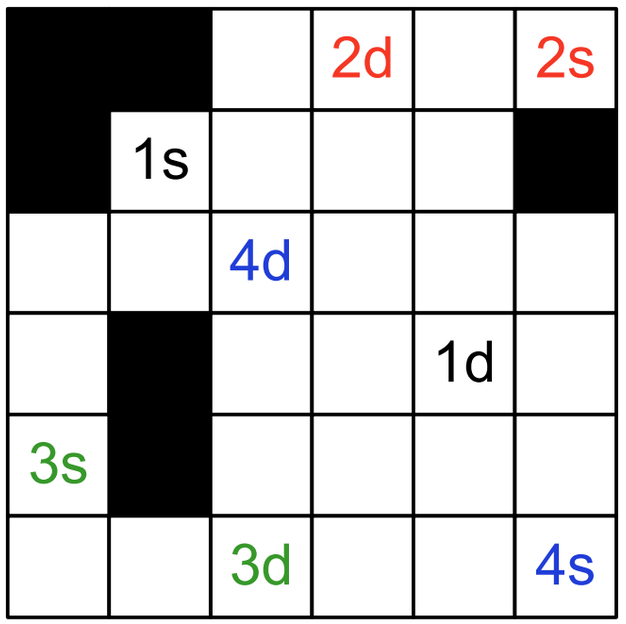
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

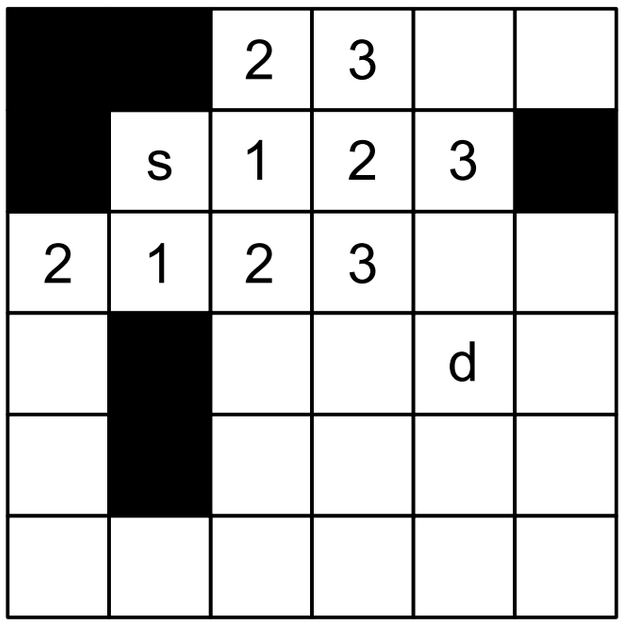
```
(defn expand-point [pt grid]
  (dosync
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

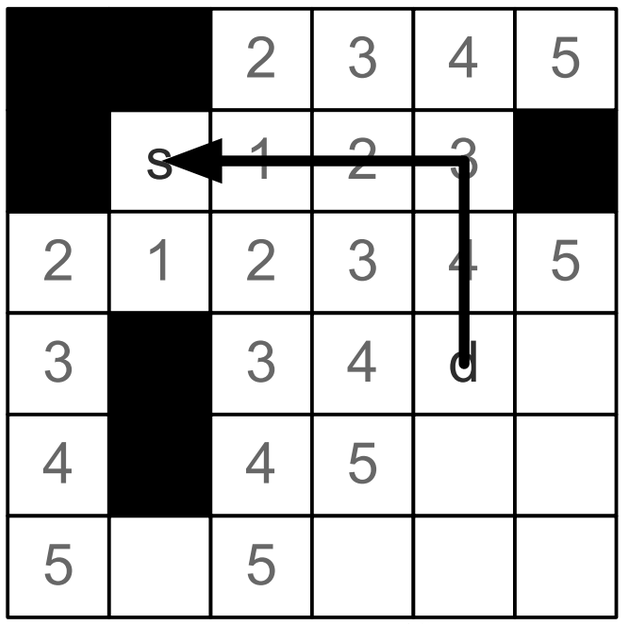
breadth-first search

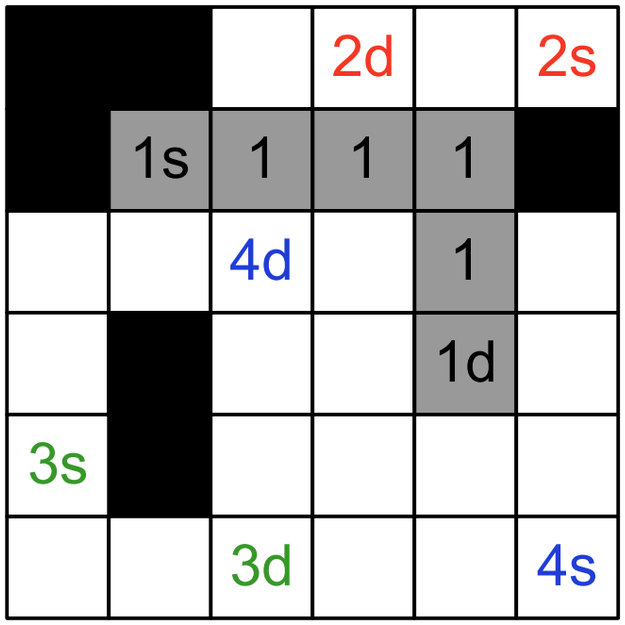


Example: Labyrinth





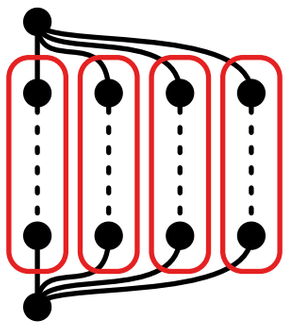





```
(for [[src dst] input-pairs]
  (future
    (dosync
      (let [local-grid (copy grid)]
        (expand src dst local-grid)
        (add-path grid
          (traceback local-grid dst))))))
```

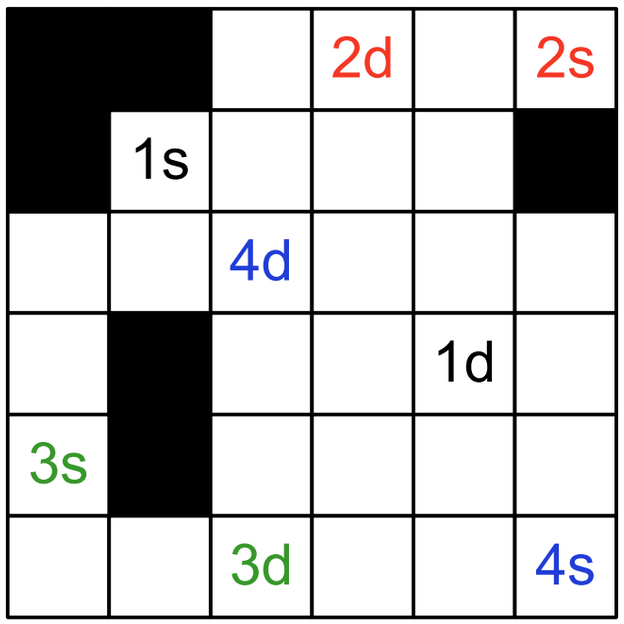
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

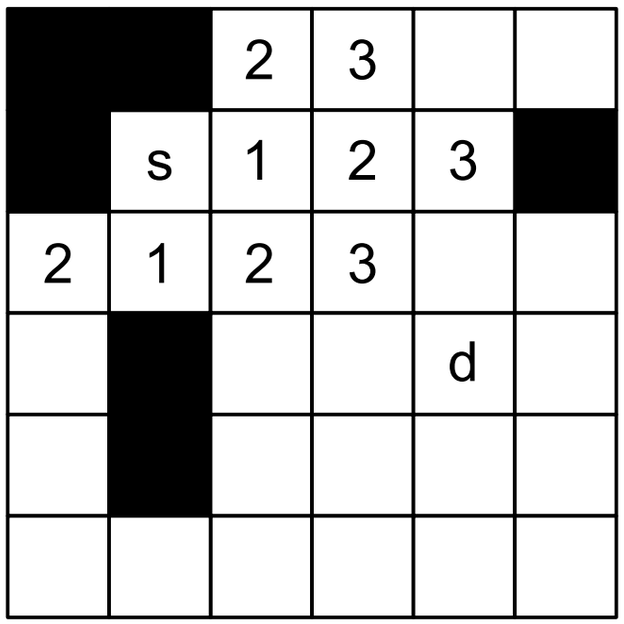
```
(defn expand-point [pt grid]
  (dosync
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

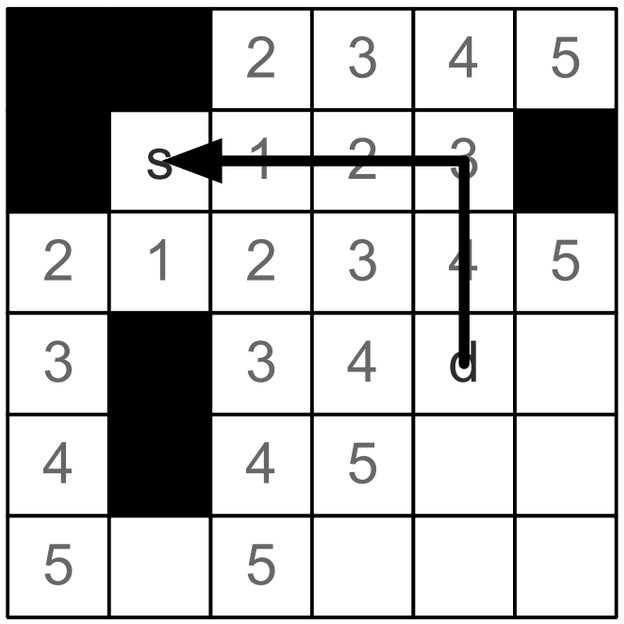


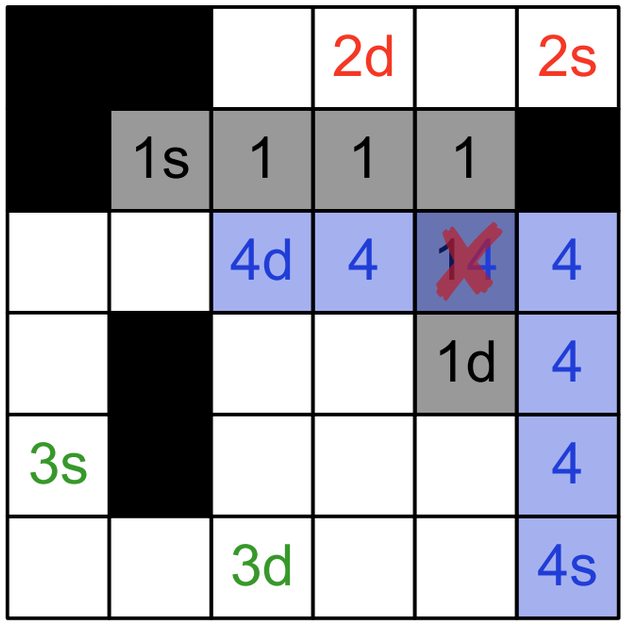
breadth-first search

Example: Labyrinth







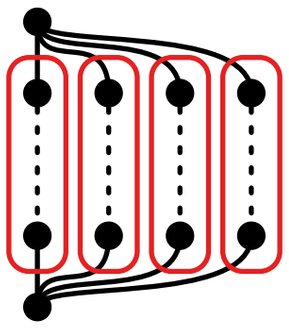



```
(for [[src dst] input-pairs]
  (future
    (dosync
      (let [local-grid (copy grid)]
        (expand src dst local-grid)
        (add-path grid
          (traceback local-grid dst))))))
```

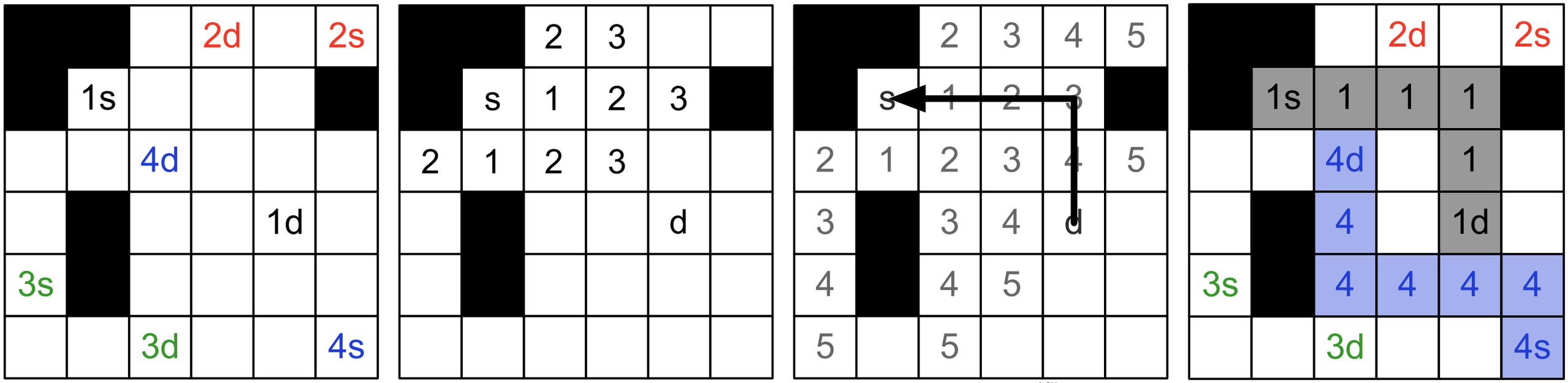
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

breadth-first search

```
(defn expand-point [pt grid]
  (dosync
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```



Example: Labyrinth

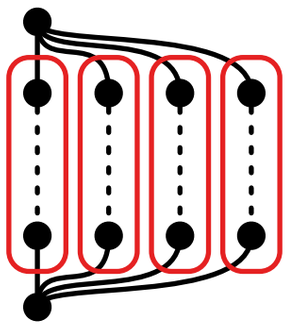



```
(for [[src dst] input-pairs]
  (future
    (dosync
      (let [local-grid (copy grid)]
        (expand src dst local-grid)
        (add-path grid
          (traceback local-grid dst))))))
```

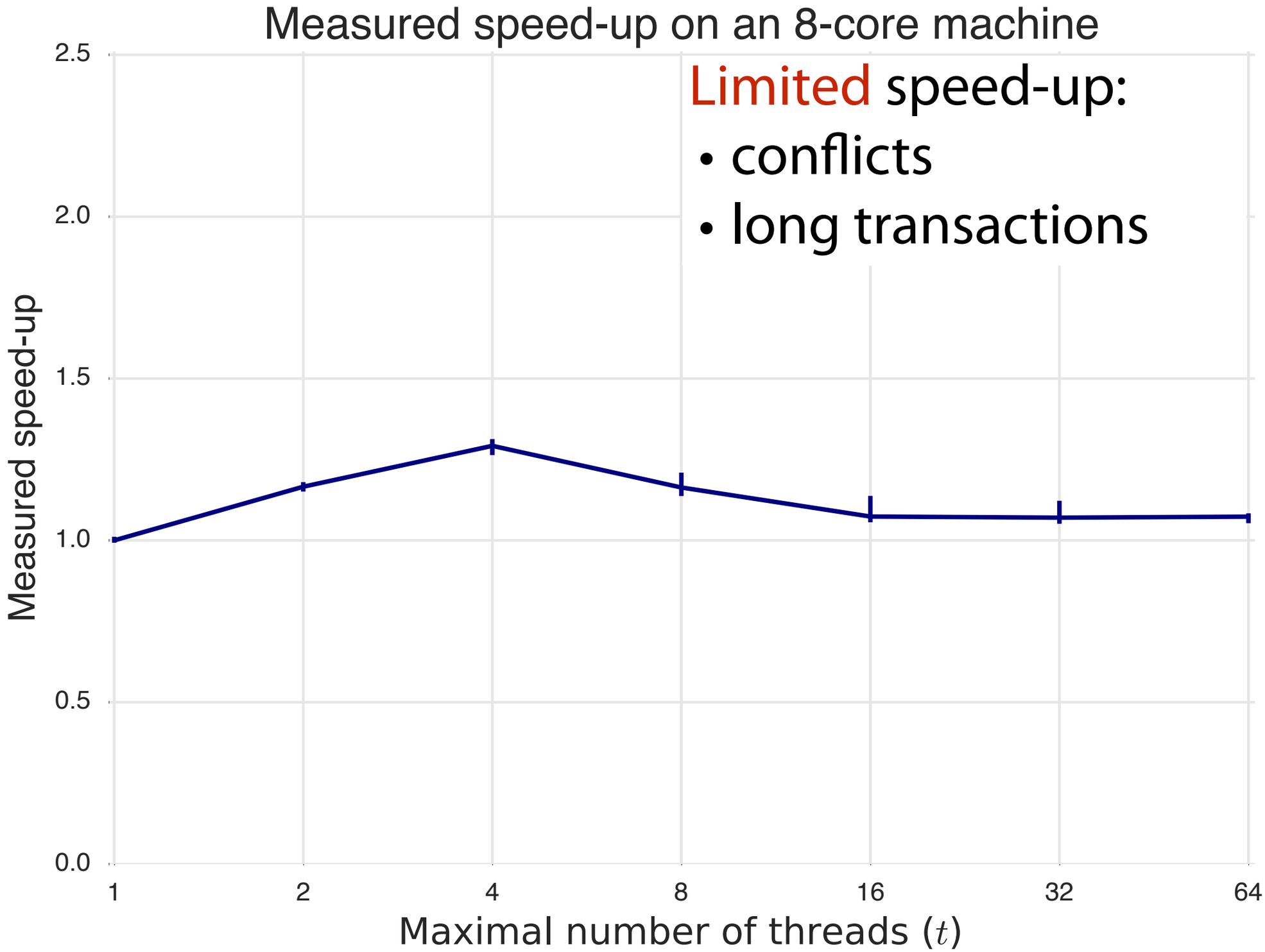
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

breadth-first search

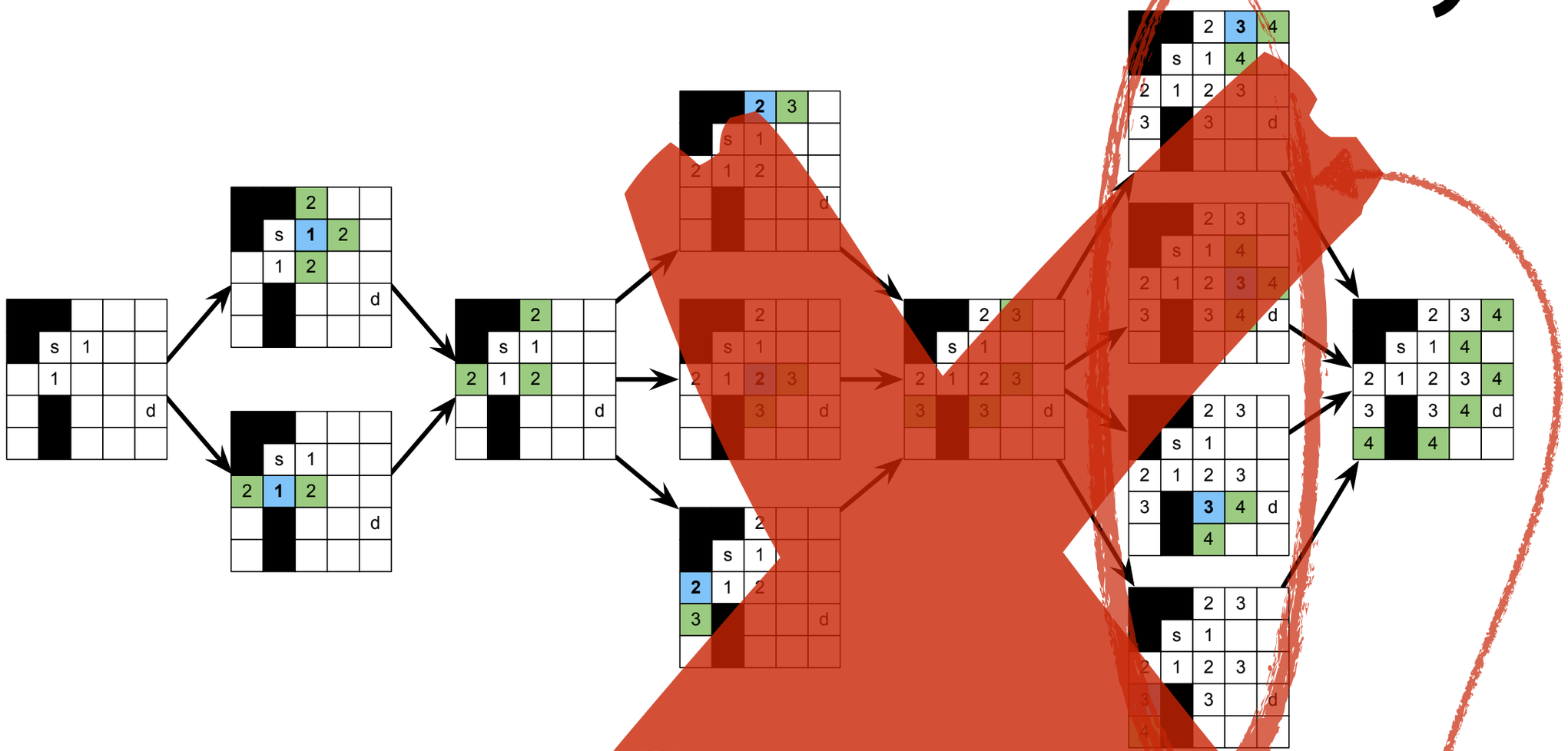
```
(defn expand-point [pt grid]
  (dosync
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```



Labyrinth has limited speed-up



Parallel search in Labyrinth



expand
in parallel

```
(for [[src dst] input-pairs]
  (future
    (dosync
      (let [local-grid (copy grid)]
        (expand src dst local-grid)
        (add-path grid
          (traceback local-grid dst))))))
```

```
(defn expand [src dst grid]
  (loop [q #{src}]
    (if (empty? q)
      false ; no path found
      (if (contains? q dst)
        true ; dst reached
        (recur
          (expand-step q grid))))))
```

```
(defn expand-step [q grid]
  (reduce union #{}
    (pmap
      (fn [p] (expand-point p grid))
      q)))
```

parallel breadth-first search

Does **not** work!

Problems when creating threads in a transaction

- Threads in transaction do not share context (Clojure, ScalaSTM)

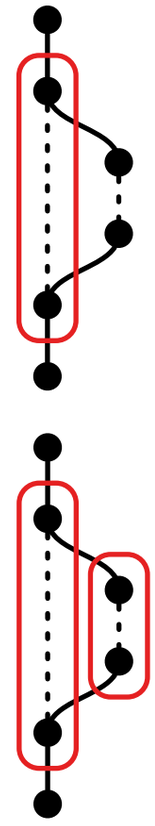
  ⇒ **no access to transactional state**

  or

  ⇒ **serializability violated**

```
(dosync
  (future
    (ref-set …)))
```

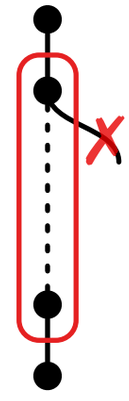
```
(dosync
  (future
    (dosync
      (ref-set …))))
```



- Threads in transaction prohibited (Haskell)

  ⇒ **parallelism limited**

```
atomically $
  do { forkIO … }
```



Overview

1. Crash course: futures
2. Crash course: transactions
3. Problem: nesting futures in transactions
4. **Solution: transactional tasks**

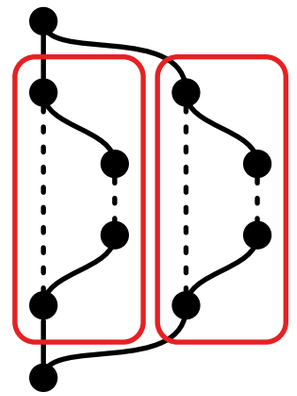
Transactional Tasks

Parallelism in transaction

⇒ Transactional task = thread created in transaction

Task can **access transactional variables**

⇒ Task adopts encapsulating transactional context



Isolation between tasks

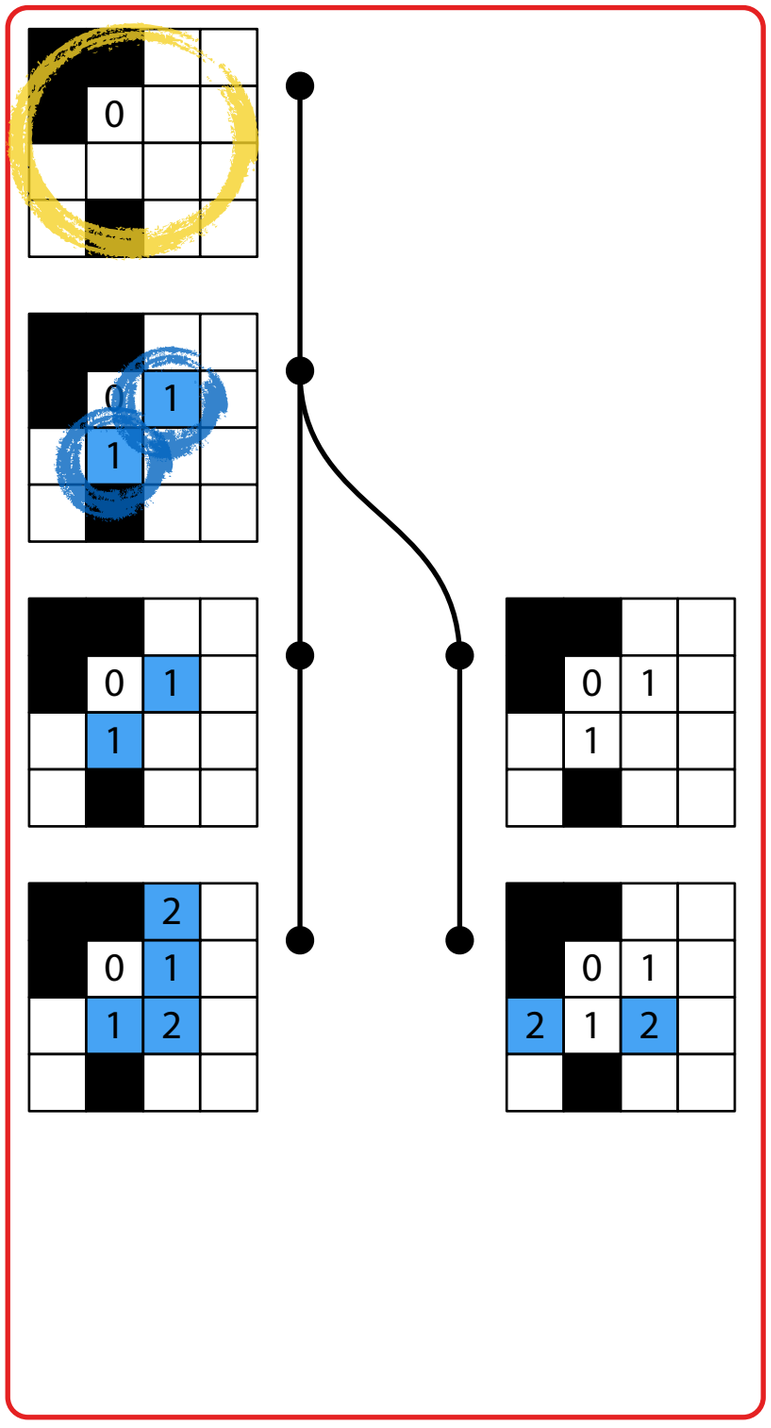
⇒ Tasks work on conceptual copy

Serializability

⇒ All tasks should join before transaction commits

On conflict, *all* tasks abort

Task = snapshot + store



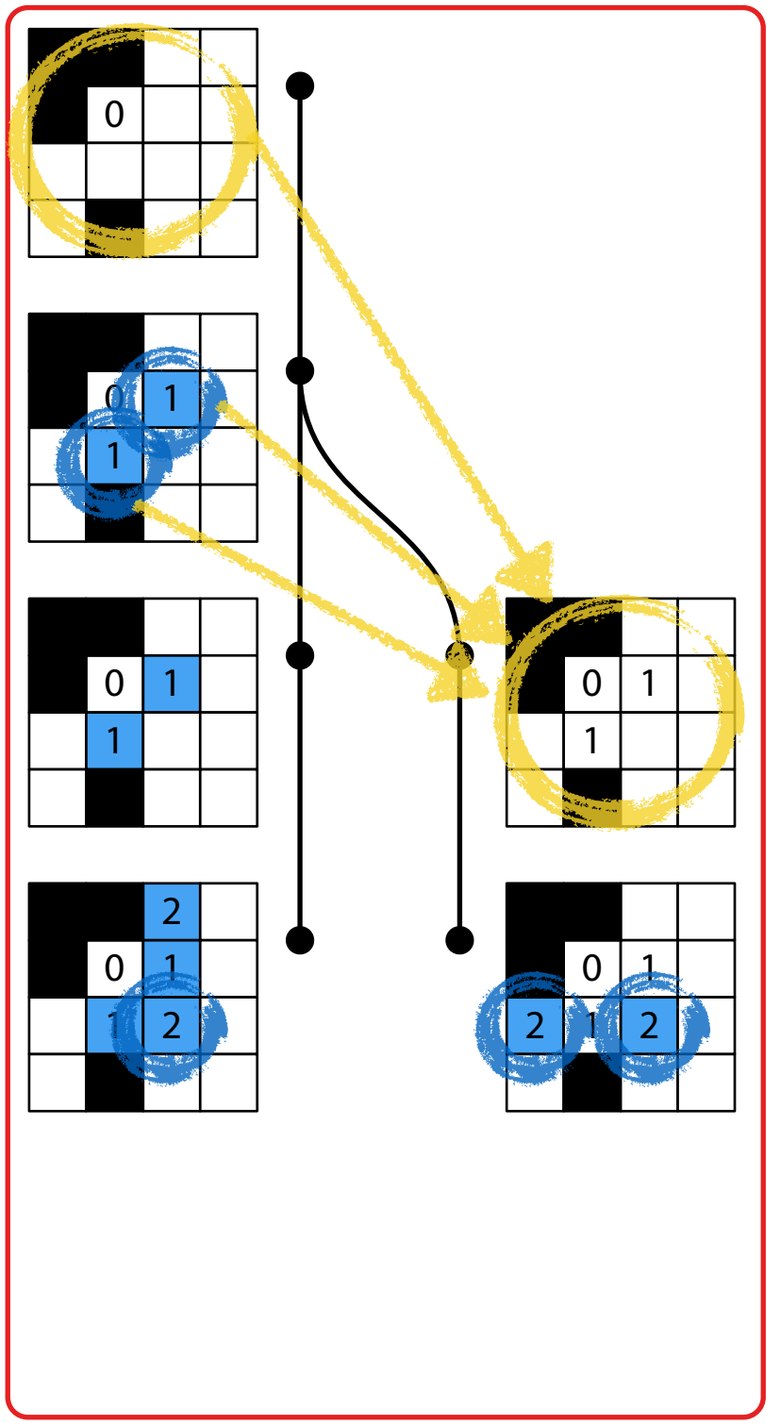

```
(dosync
  (ref-set … 1)
```

Each transactional task contains:

snapshot: transactional state on creation

local store: local modifications

`future` creates isolated task



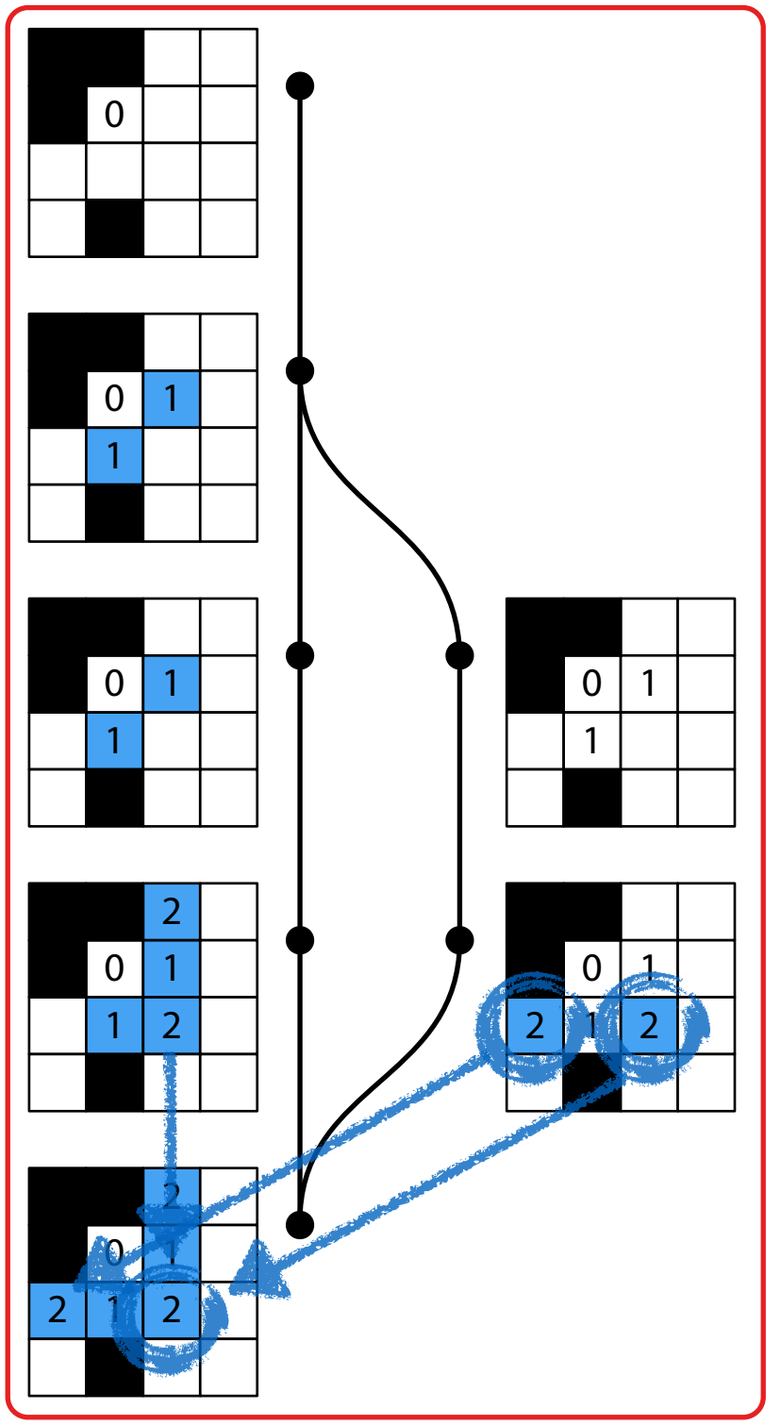

```
(dosync
  (ref-set … 1)
  (future
    (ref-set … 2))
  (ref-set … 2)
```

Each transactional task contains:

snapshot: transactional state on creation

local store: local modifications

`deref` merges changes




```
(dosync
  …
  (deref child))
```

merge local store of child into parent

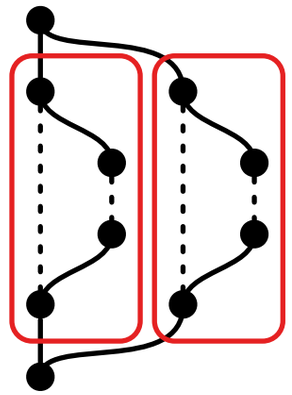
Conflict resolution function:

```
(ref 0 resolve)
```

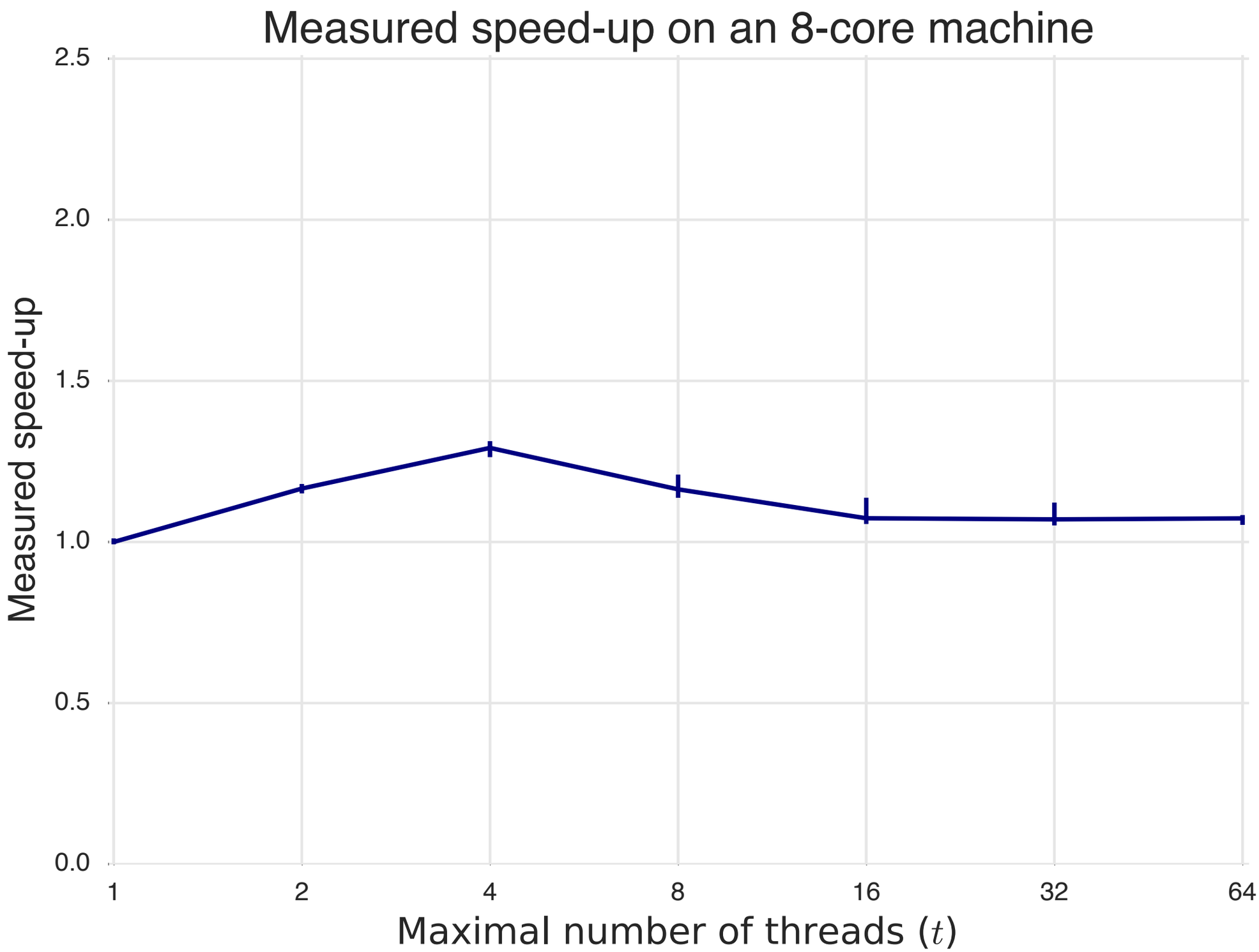
```
(defn resolve [o p c] c)
(defn resolve [o p c] p)
(defn resolve [o p c] (min p c))
(defn resolve [o p c] (+ p c))
(defn resolve [o p c] (error "merge conflict"))
```

Properties of transactional tasks

- **In-transaction parallelism** possible
- **Serializability** of transactions
- **Coordination** of tasks: all or none
- In-transaction **determinacy**

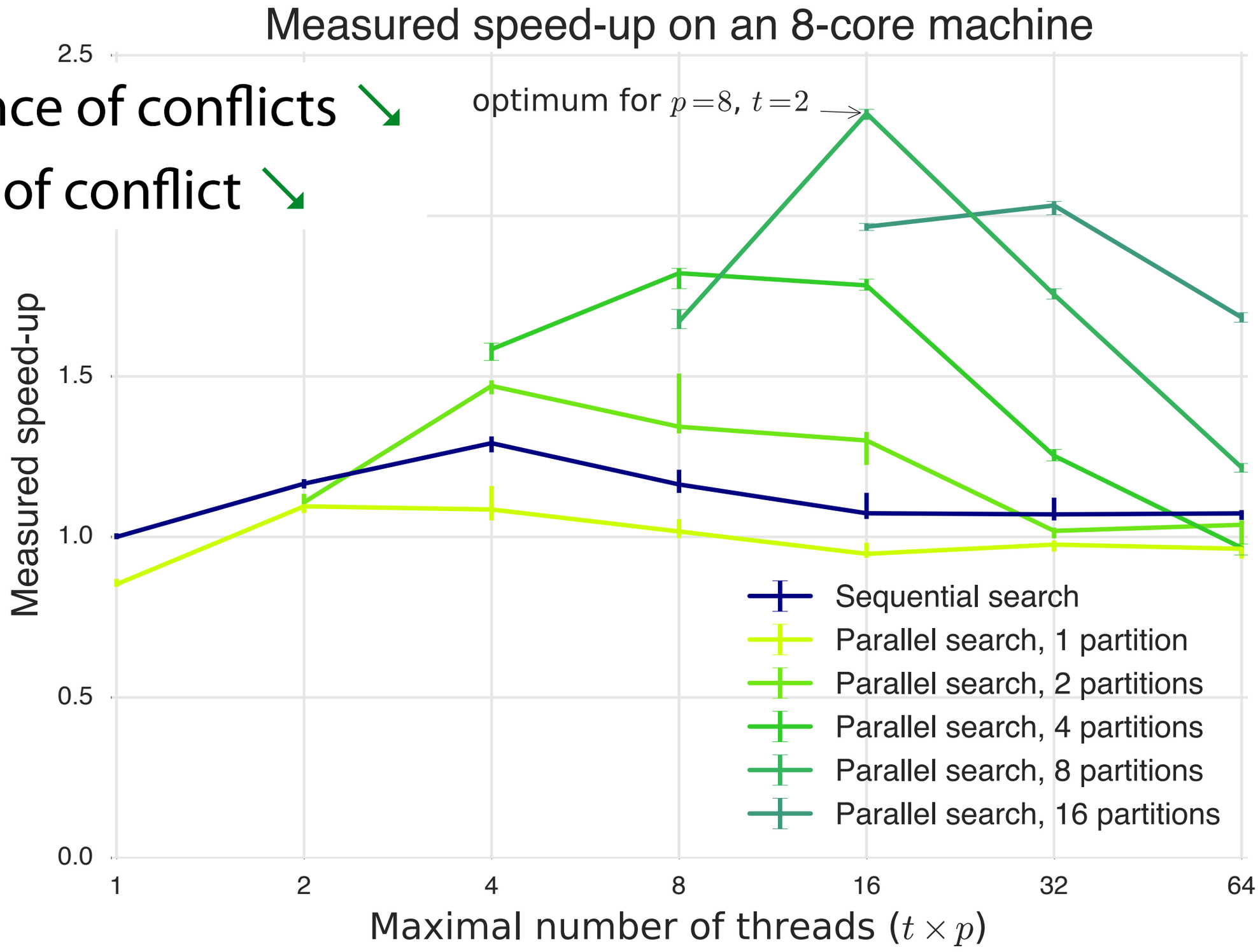


Evaluation: Labyrinth

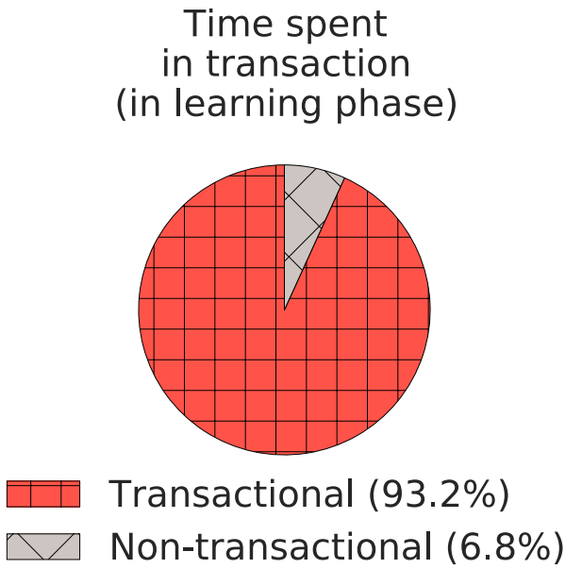


Evaluation: Labyrinth

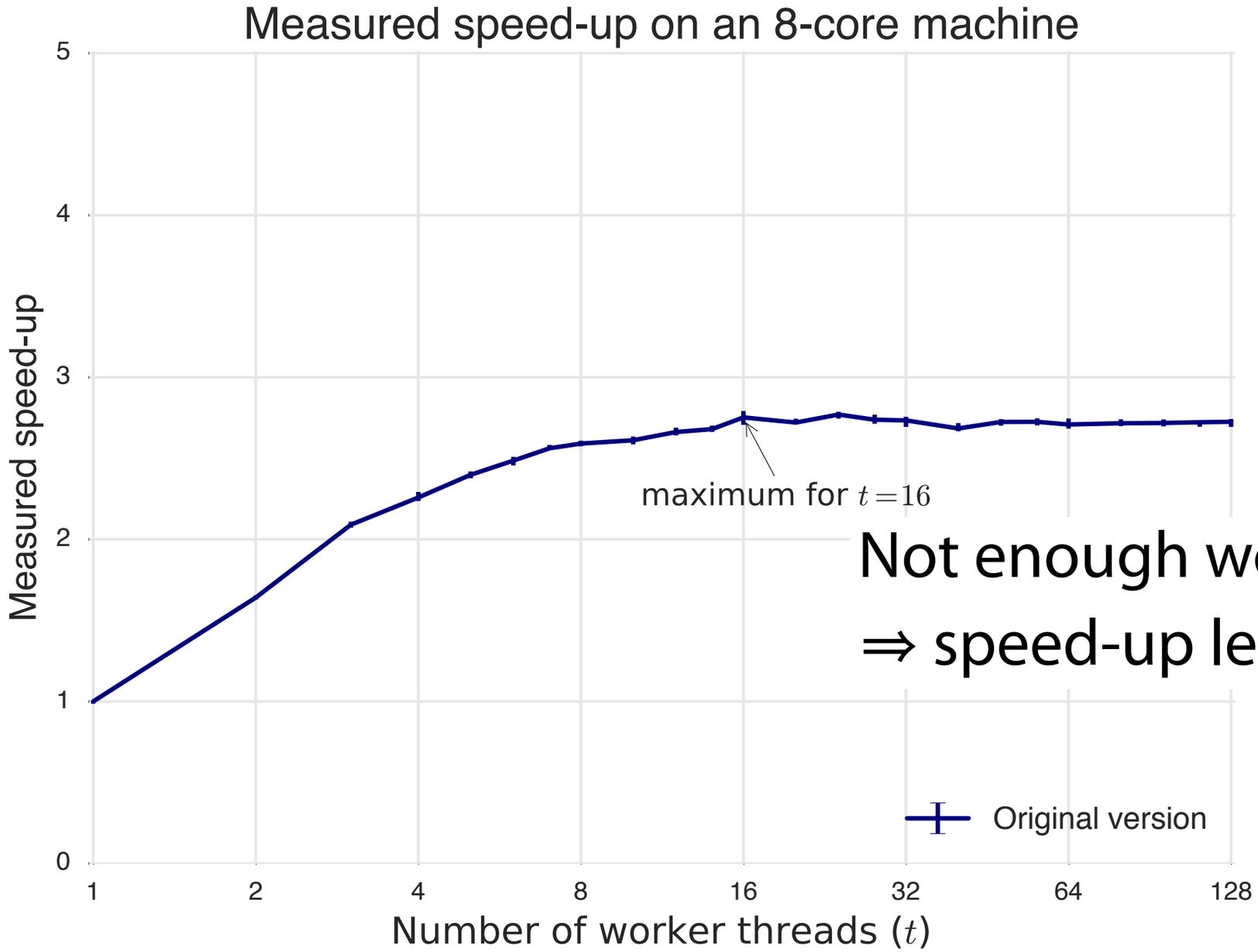
- Chance of conflicts ↘
- Cost of conflict ↘



Evaluation: Bayes



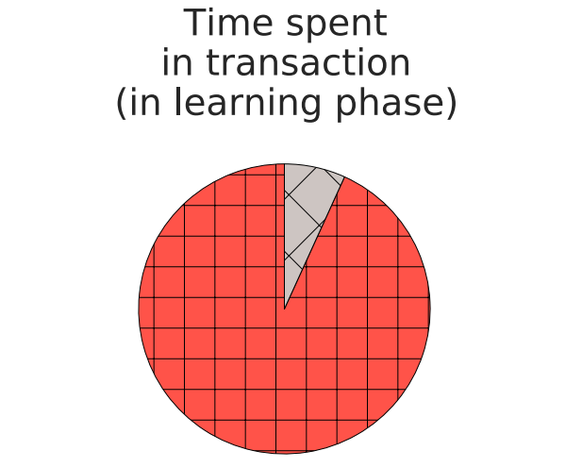

```
(dosync
  …
  (for [from-id (range (:n-var adtree))]

      (compute-local-log-likelihood …)))
```



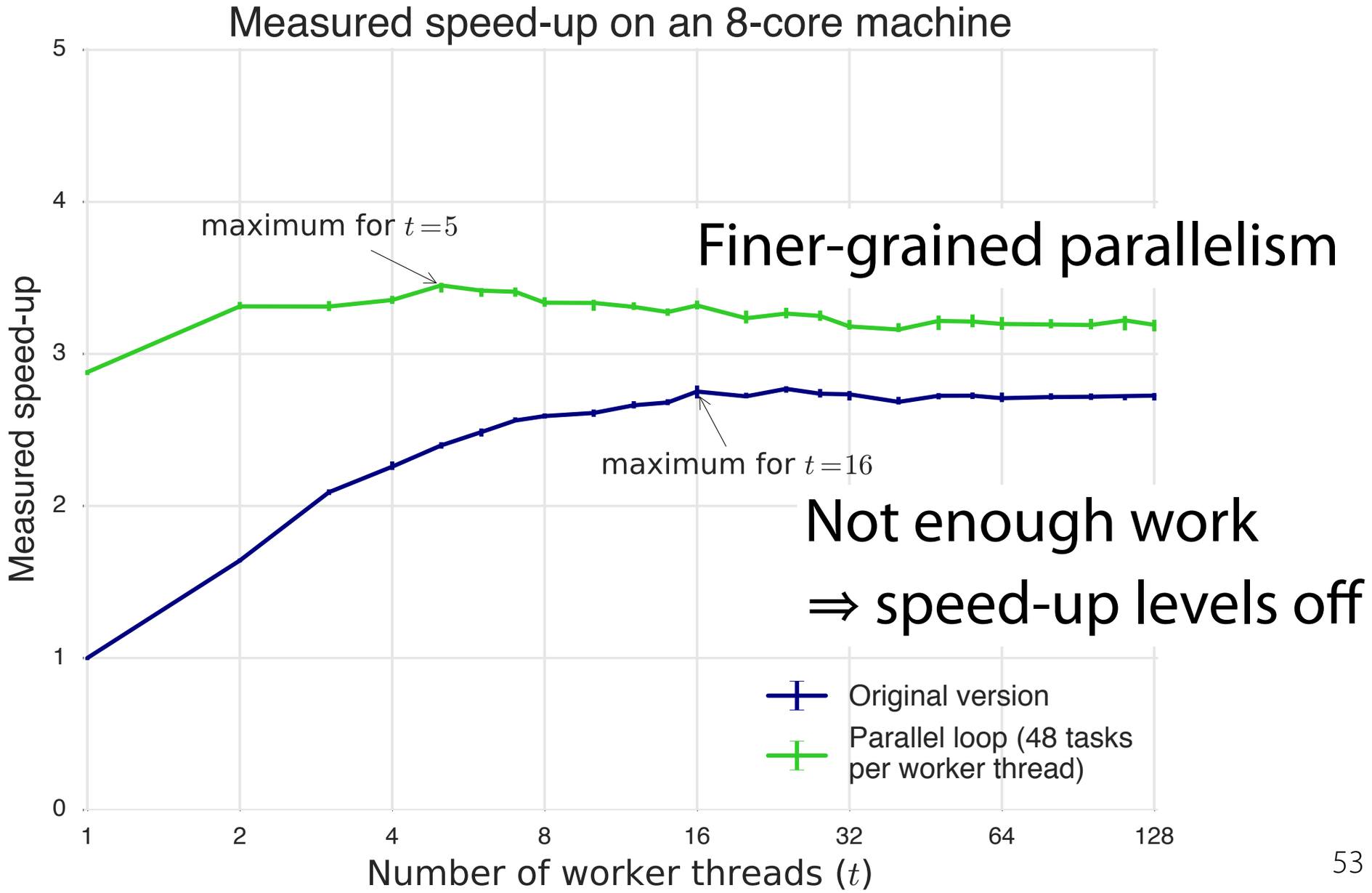
Not enough work

⇒ speed-up levels off

Evaluation: Bayes




```
(dosync
  …
  (for [from-id (range (:n-var adtree))]
    (future
      (compute-local-log-likelihood …))))
```



Finer-grained parallelism

Not enough work

⇒ speed-up levels off

Insights from experiments

- Labyrinth: parallelize search algorithm
  ⇒ fewer & cheaper conflicts

- Bayes: more fine-grained parallelism
  ⇒ better exploit hardware

- Low developer effort (re-use existing concepts)

- Suitable for applications with long transactions

Implementation

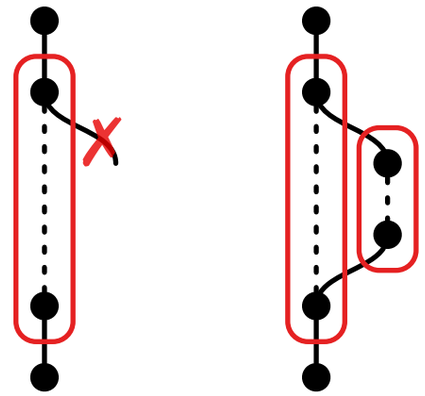
Fork of Clojure

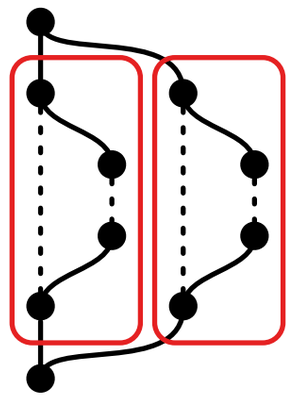
https://github.com/jswalens/transactional-futures/

http://soft.vub.ac.be/~jswalens/ecoop-2016-artifact/

Summary

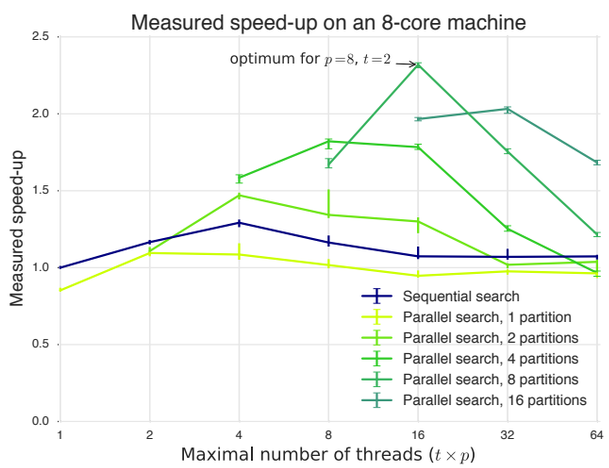
Parallelism in a transaction is useful
for programs with **long transactions**
But currently:
↗ **not allowed** (Haskell)
↘ **not serializable** (Clojure, Scala)





Idea: **transactional tasks**
- safe access to encapsulating transaction
- serializable, coordinated, determinate

Benefits:
- finer-grained parallelism ⇒ speed-up
- low developer effort



https://github.com/jswalens/transactional-futures/
http://soft.vub.ac.be/~jswalens/ecoop-2016-artifact/

Transactional Tasks vs. Nested Parallel Transactions

Guarantees in transaction

- NPT: `(dosync (future …))` → race conditions possible
- NPT: `(dosync (future (dosync …)))` → serializable, last writer wins (not deterministic)
- TT: conflict resolution → *in-transaction determinacy* (but may need to define resolution function)

Performance

- NPT: roll back and retry subtransaction
- TT: resolve conflict

→ different performance characteristics depending on application (chance of conflicts between threads in tx)

More fine-tuned conflict resolution (e.g. minimum for Labyrinth)

STAMP

Application	Instructions /tx (mean)	Time in tx
labyrinth	219,571 ●	100% ●
bayes	60,584 ●	83% ●
yada	9,795 ●	100% ●
vacation-high	3,223 ◐	86% ●
genome	1,717 ◐	97% ●
intruder	330 ○	33% ◐
kmeans-high	117 ○	7% ○
ssca2	50 ○	17% ○

Coarse-grained parallelism
between parts of the application

Fine-grained parallelism
within a part of the application

Transactions

- Conflicts span multiple variables
- Difficult to define conflict resolution functions
- Chance of conflicts depends on application

⇒ resolve high-level conflicts using serializability

Transactional tasks

- Conflict affects single variable
- Define conflict resolution function based on algorithm
- Conflicts likely, so rollback bad for performance

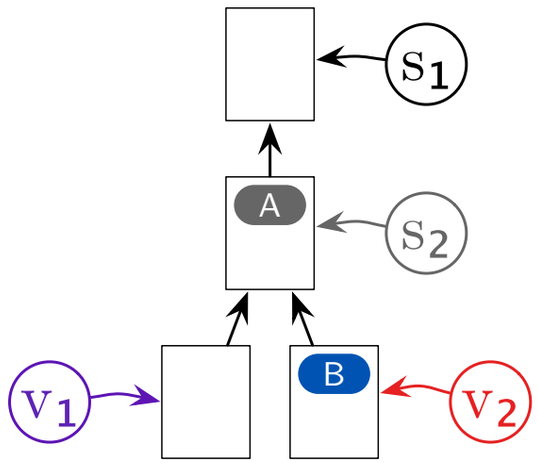
⇒ resolve low-level conflicts using conflict resolution functions

Implementation details

(a) Code example.

```
(atomic
  [1](ref-set gray A)
  [2](fork [3](ref-set blue B)
           [4](fork [5](ref-set green C))
           [6](ref-set red D)
           ...)
  [7](ref-set purple E)
  ...)
```

(b) Data after step 3.



(c) Data after step 7.