Transactional Tasks

Parallelism in Software Transactions

Janwillem Swalens, Joeri De Koster, Wolfgang De Meuter

Futures for parallelism

(**`fork`** `e`) returns `f`

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

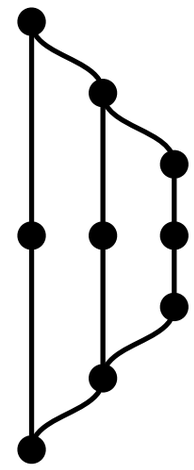
```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fib (- n 1))
          b (fib (- n 2))]
      (+ a b))))
```

Futures for parallelism

(**fork** e) returns f

(**join** f) returns result of e

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

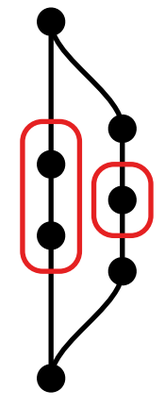


Transactions for shared memory

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

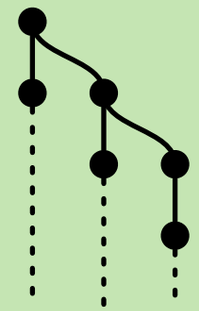
serializability

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



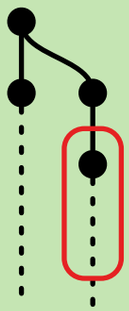
Nesting futures & transactions

```
(fork
  (fork
    …))
```



Nested task parallelism

```
(fork
  (atomic
    …))
```

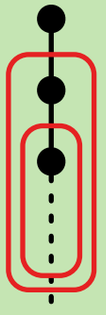


Transactions

```
(atomic
  (fork
    …))
```

In-transaction parallelism

```
(atomic
  (atomic
    …))
```

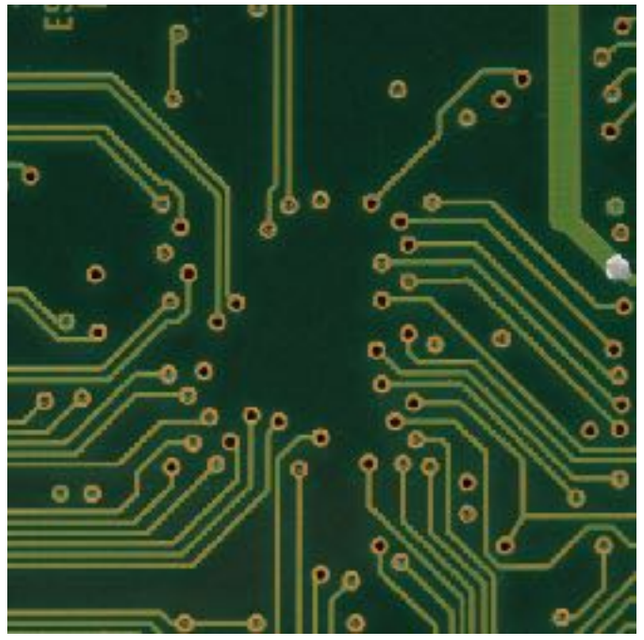


Nested transactions (open/closed)

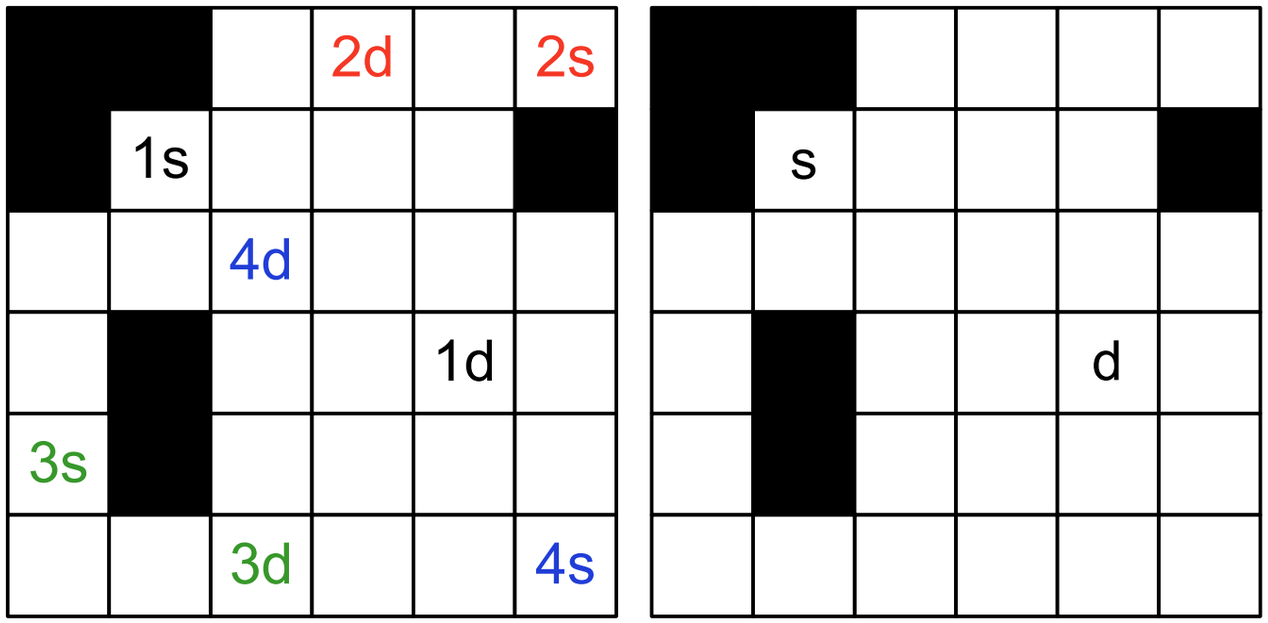
Example: Labyrinth

■	■		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

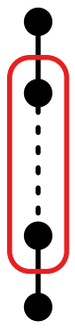



```
(for [[src dst] input-pairs]
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

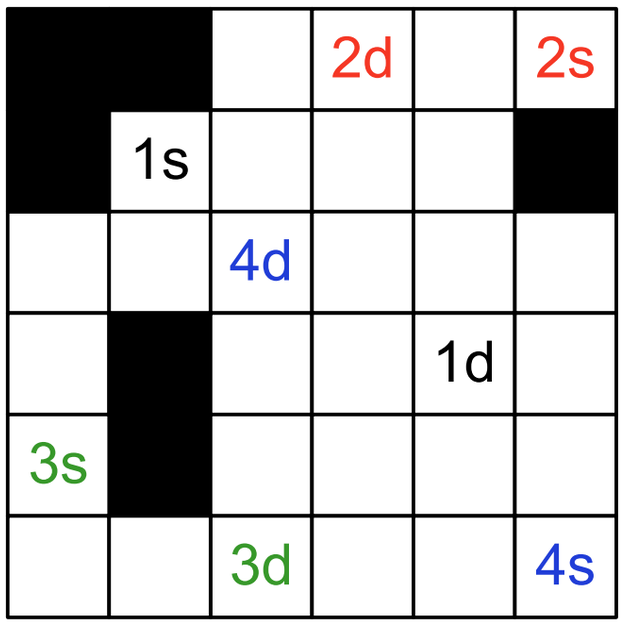
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

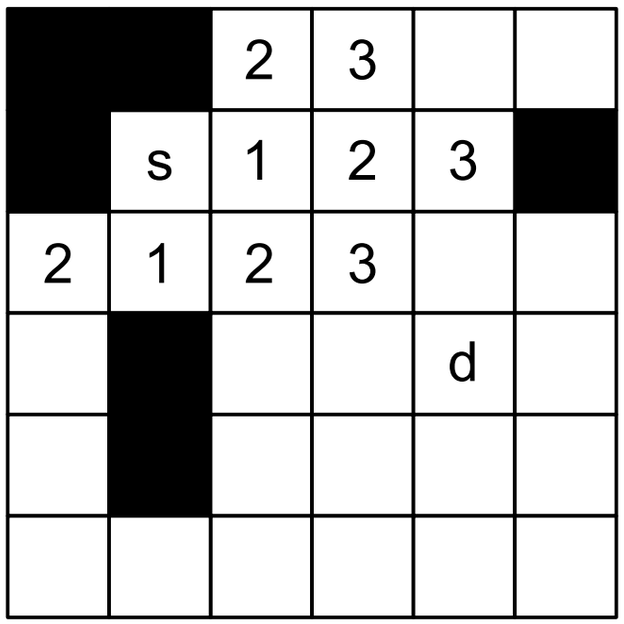
```
(defn expand-point [pt grid]
  (atomic
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

breadth-first search



Example: Labyrinth



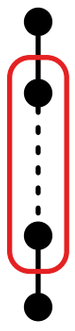



```
(for [[src dst] input-pairs]
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

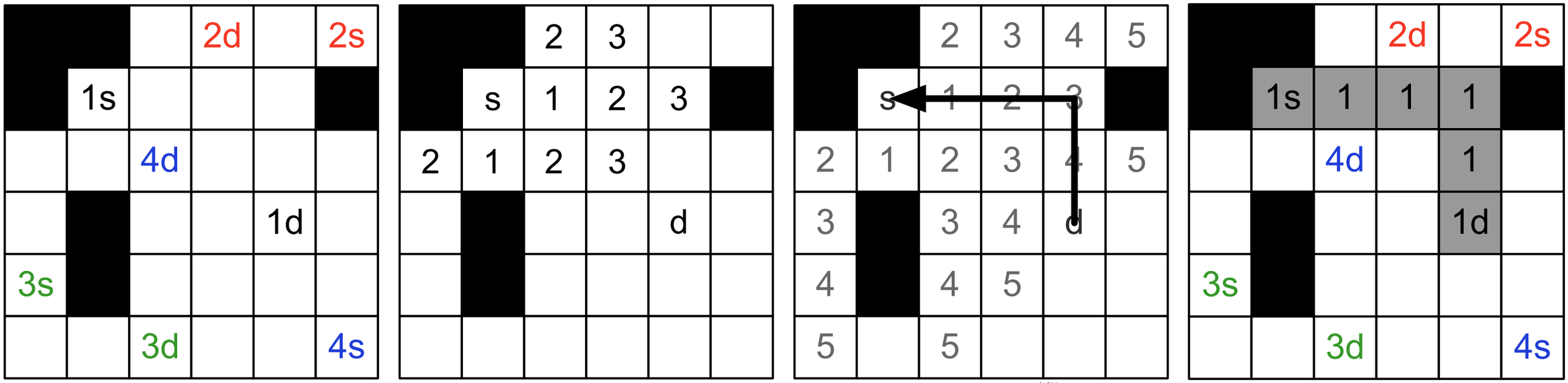
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

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

breadth-first search



Example: Labyrinth



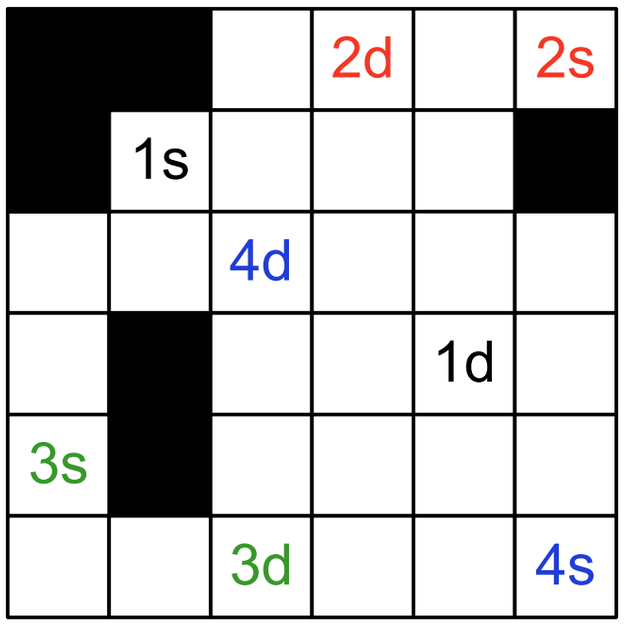

```
(for [[src dst] input-pairs]
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

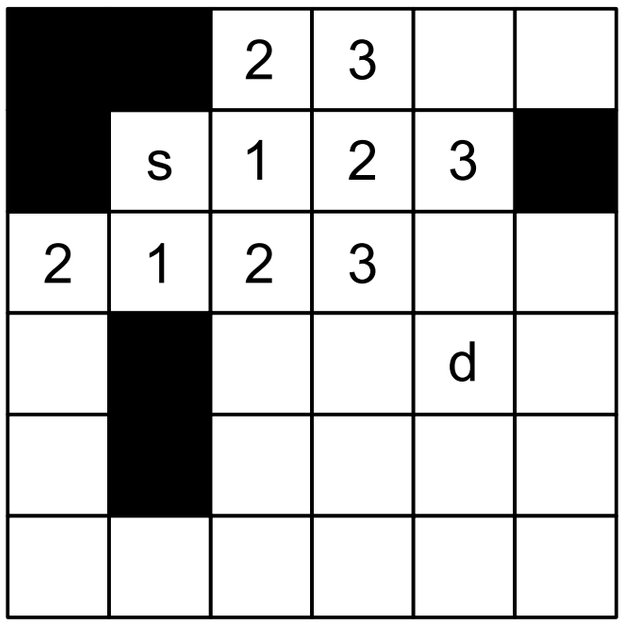
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

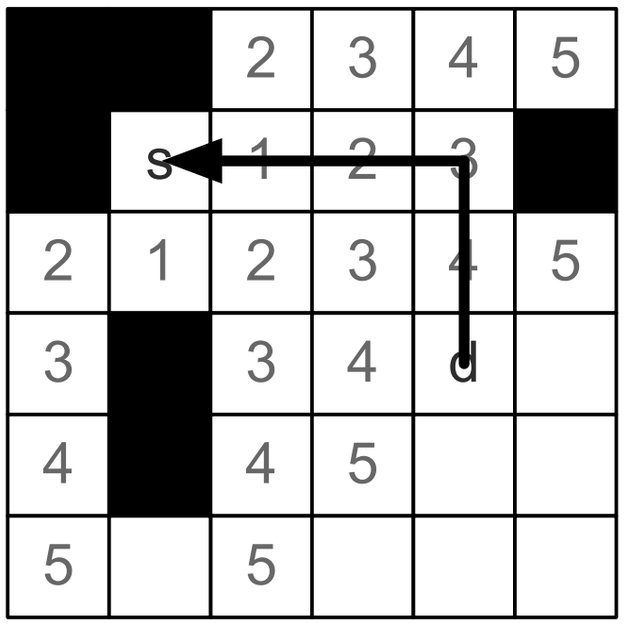
```
(defn expand-point [pt grid]
  (atomic
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

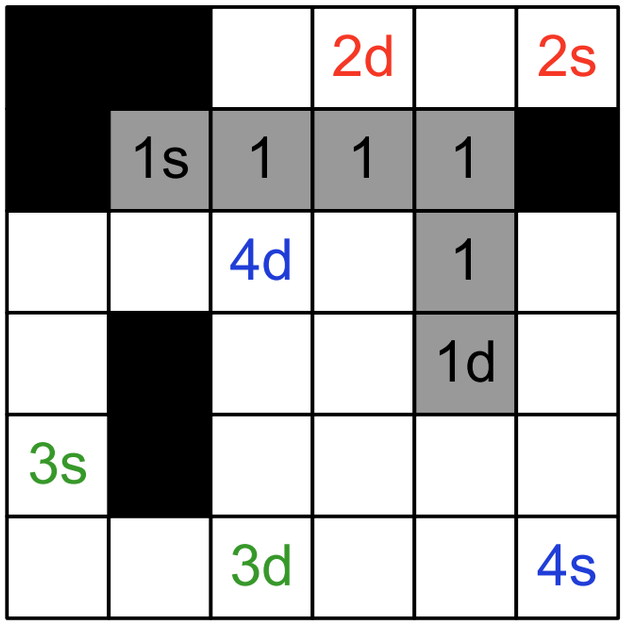
breadth-first search

Example: Labyrinth







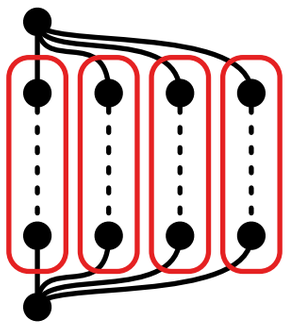



```
(for [[src dst] input-pairs]
  (fork [local-grid (copy grid)]
    (atomic
      (mark src dst local-grid)
      (add-path grid
        (traceback local-grid dst)))))
```

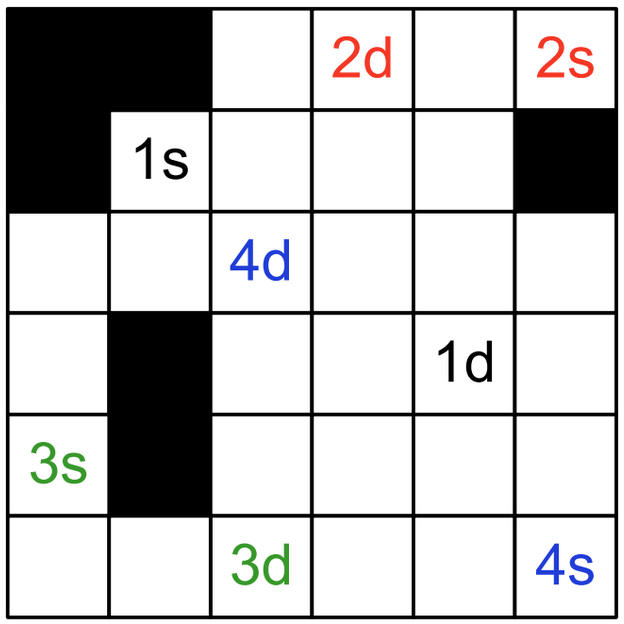
```
(defn expand-point [pt grid]
  (atomic
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

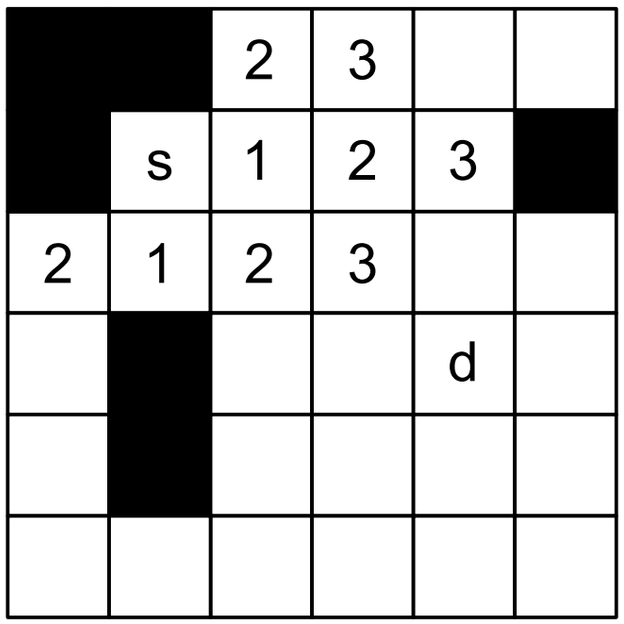
```
))
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

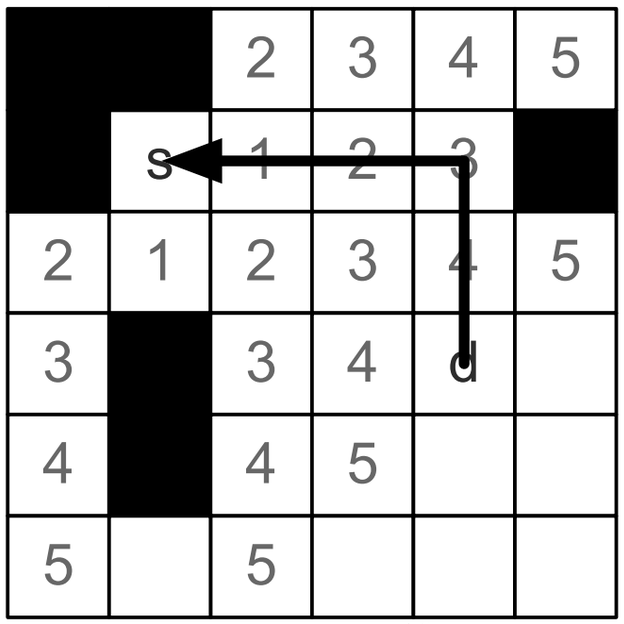
breadth-first search

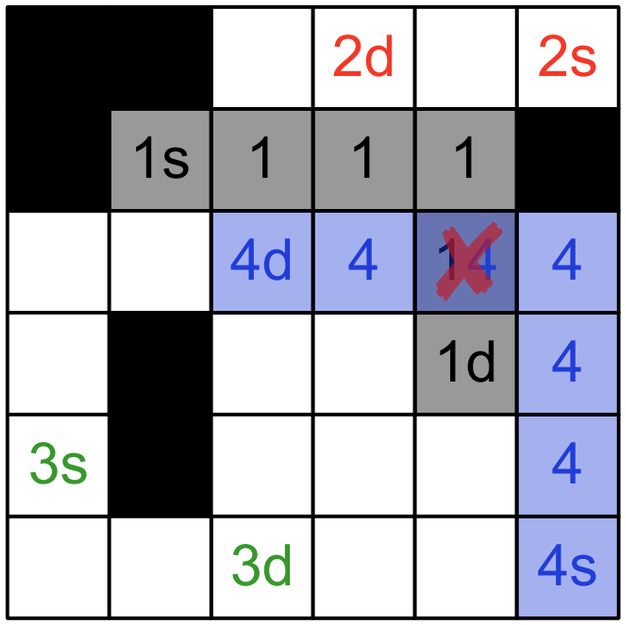


Example: Labyrinth







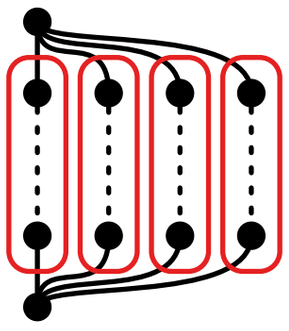



```
(for [[src dst] input-pairs]
  (fork
    (atomic
      (let [local-grid (copy grid)]
        (expand src dst local-grid)
        (add-path grid
          (traceback local-grid dst))))))
```

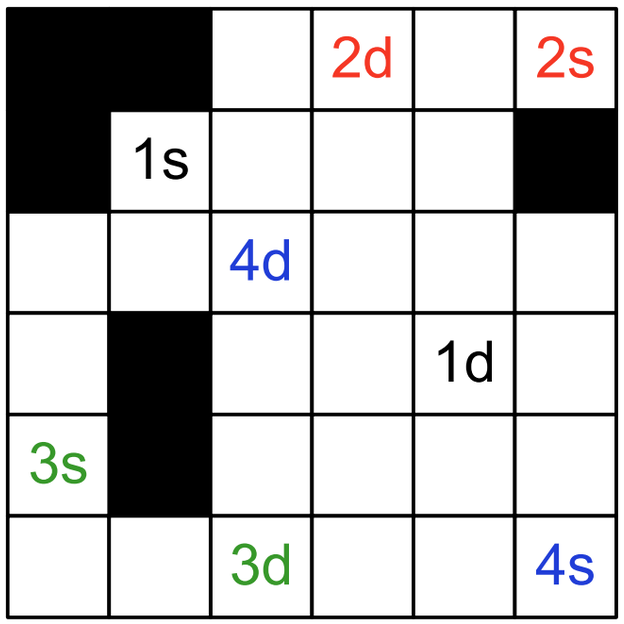
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

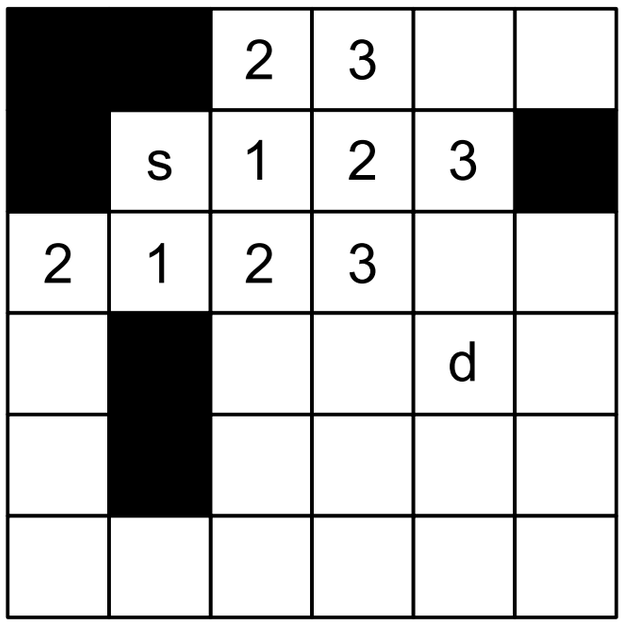
breadth-first search

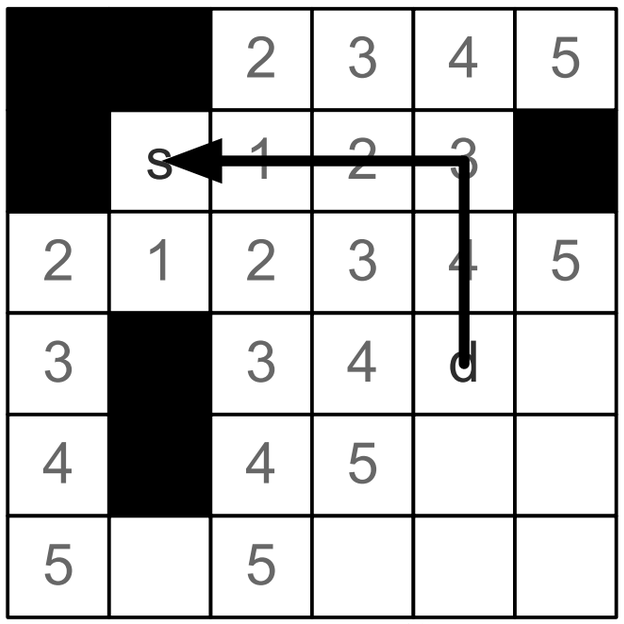
```
(defn expand-point [pt grid]
  (atomic
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```

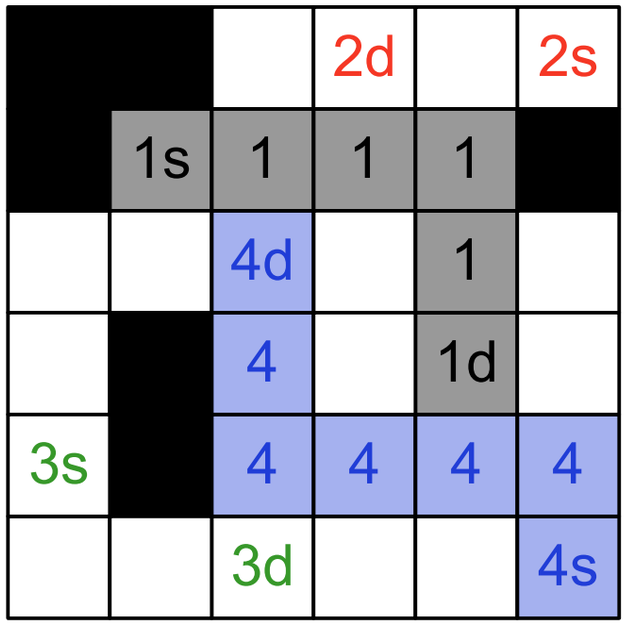


Example: Labyrinth







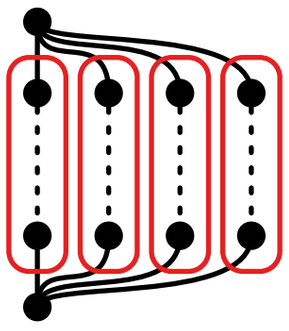



```
(for [[src dst] input-pairs]
  (fork
    (atomic
      (let [local-grid (copy grid)]
        (expand src dst local-grid)
        (add-path grid
          (traceback local-grid dst))))))
```

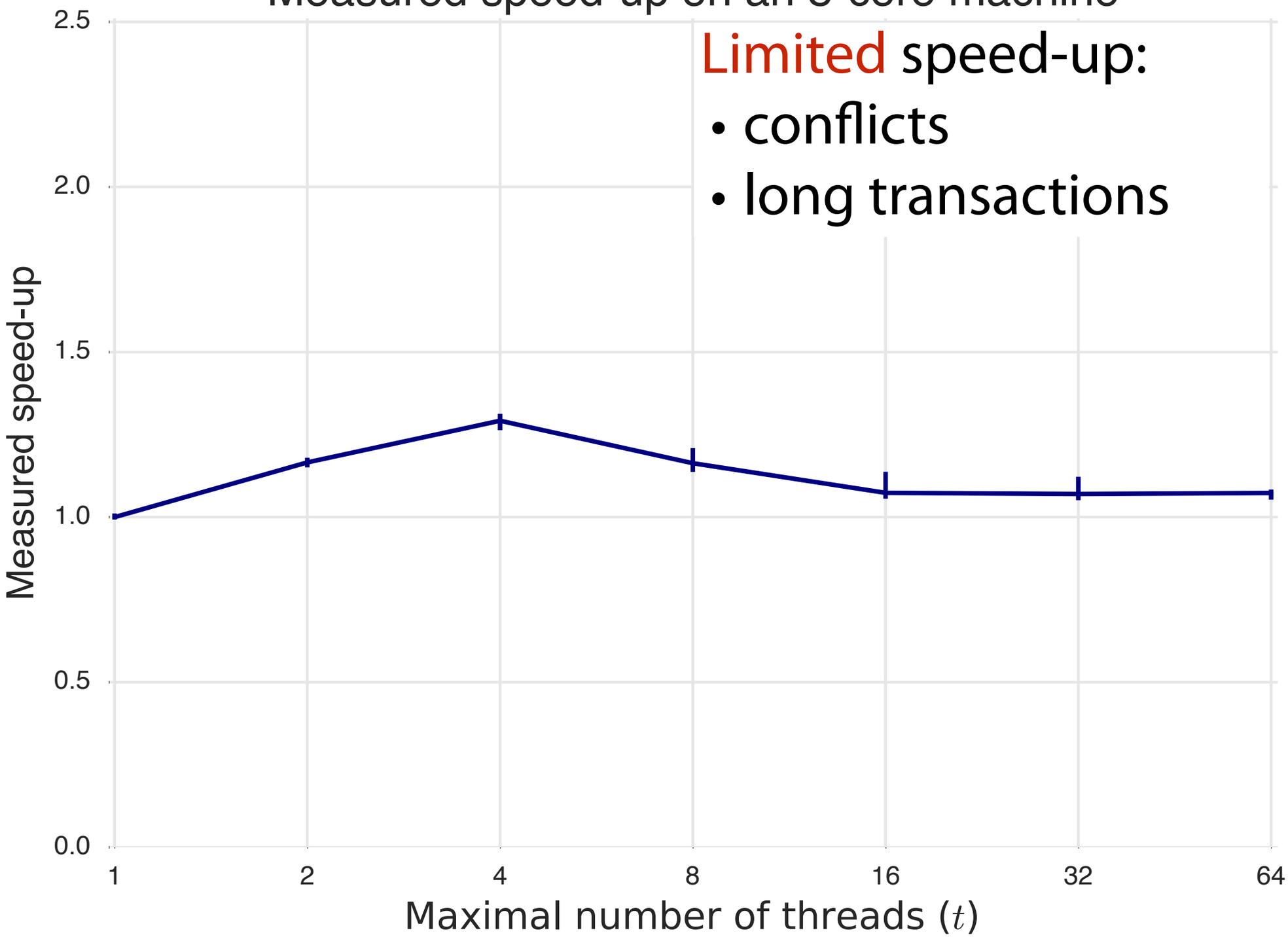
```
(defn expand [src dst grid]
  (loop [q (list src)]
    (if (empty? q)
      false ; no path found
      (if (= (first q) dst)
        true ; dst reached
        (recur
          (concat
            (rest q)
            (expand-point (first q) grid)))))))
```

breadth-first search

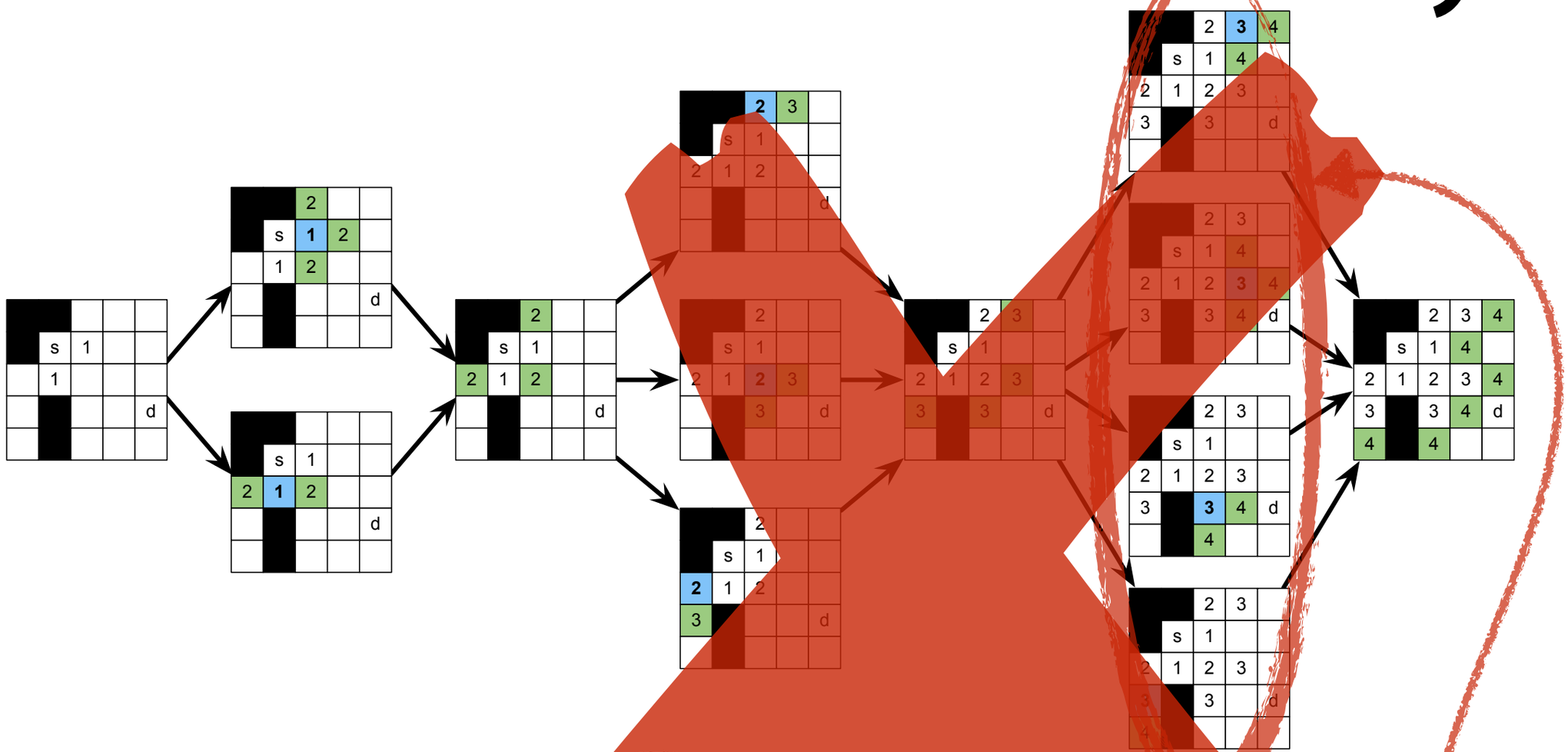
```
(defn expand-point [pt grid]
  (atomic
    (let [neighbors …]
      (for [n neighbors]
        (ref-set n …))
      neighbors)))
```



Labyrinth has limited speed-up



Parallel search in Labyrinth



expand
in parallel

```
(for [[src dst] input-pairs]
  (fork
    (atomic
      (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

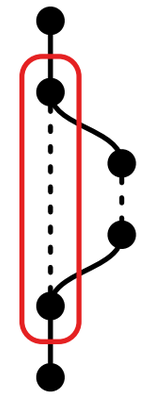
- Threads in transaction do not share context (Clojure, ScalaSTM)

  ⇒ **no access to transactional state**

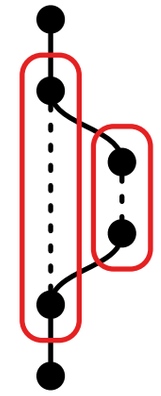
  or

  ⇒ **serializability violated**

```
(atomic
  (fork
    (ref-set …)))
```



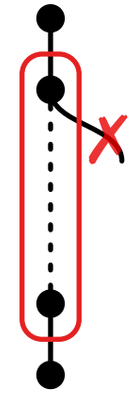
```
(atomic
  (fork
    (atomic
      (ref-set …))))
```



- Threads in transaction prohibited (Haskell)

  ⇒ **parallelism limited**

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



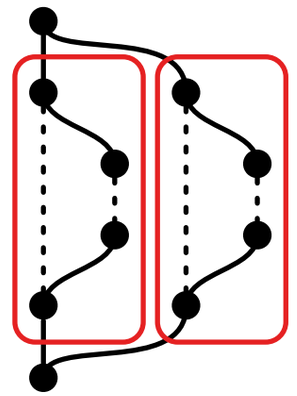
Transactional Tasks

Parallelism in transaction

⇒ Transactional task = thread created in transaction

Task can **access transactional variables**

⇒ Task adopts encapsulating transactional context



Isolation between tasks

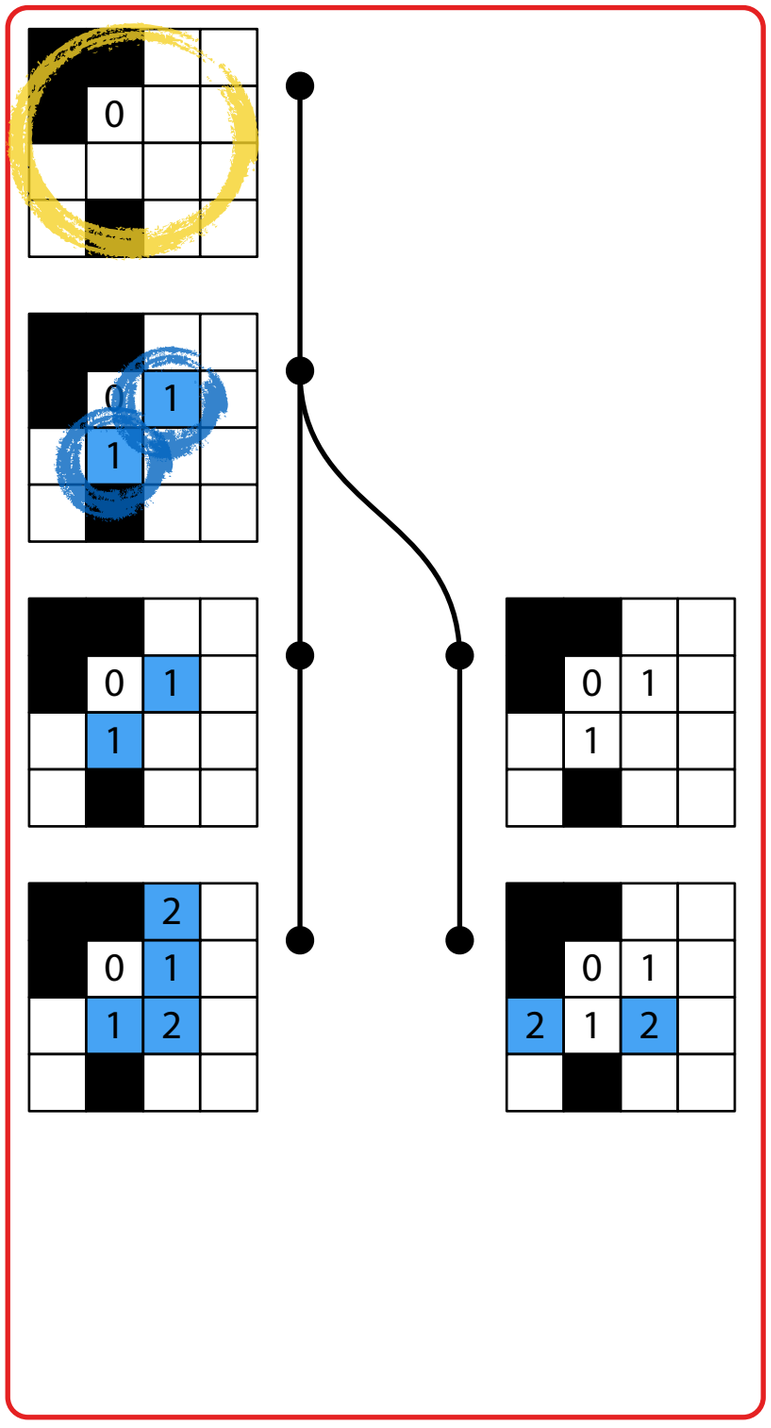
⇒ Tasks work on conceptual copy

Serializability

⇒ All tasks should join before transaction commits

On conflict, *all* tasks abort

Task = snapshot + store




```
(atomic
  (ref-set … 1)
```

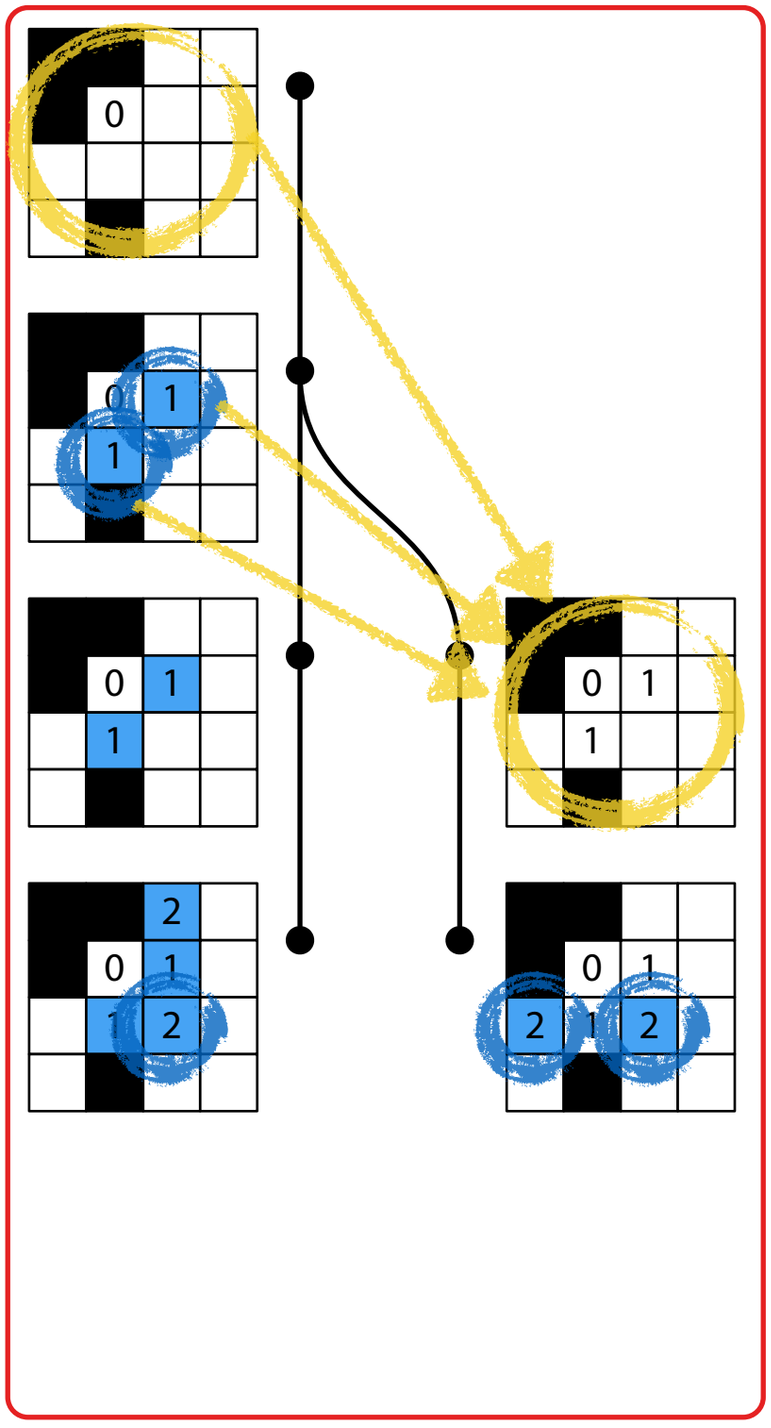
Each transactional task contains:

snapshot σ: transactional state on creation

local store τ: local modifications

$$\begin{aligned} & \mathrm{T_x} \uplus \langle f_p, \sigma, \tau, \mathrm{F_s}, && \mathrm{F_j}, \mathcal{E}[\texttt{fork}\ e] \rangle \\ \Rightarrow_{\mathrm{tf}}\ & \mathrm{T_x} \uplus \langle f_p, \sigma, \tau, \mathrm{F_s} \uplus \{f'\}, \mathrm{F_j}, \mathcal{E}[f_c] && \rangle \uplus \langle f_c, \sigma :: \tau, \varnothing, \varnothing, \mathrm{F_j}, e \rangle \\ & \text{with } f_c \text{ fresh} \end{aligned}$$

`fork` creates isolated task




```
(atomic
  (ref-set … 1)
  (fork
    (ref-set … 2))
  (ref-set … 2)
```

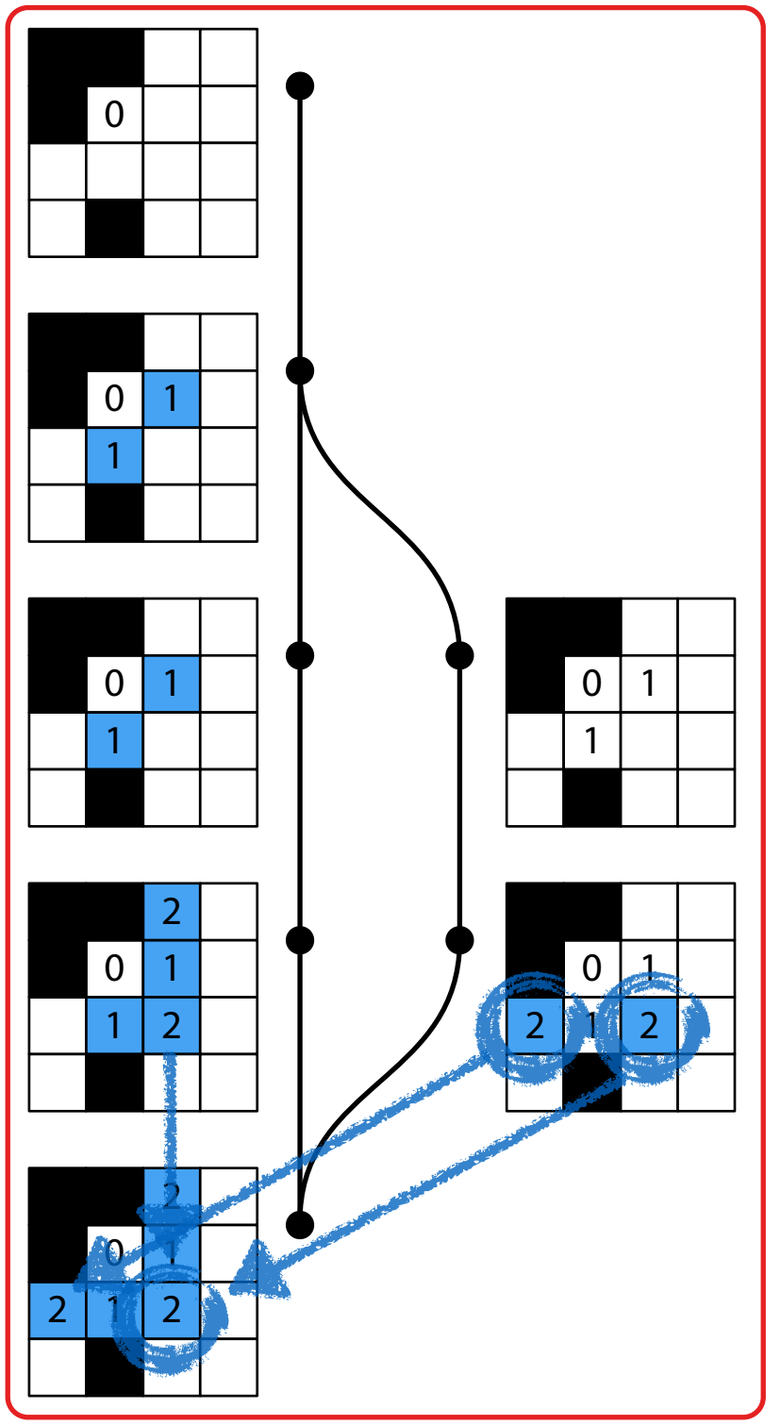
Each transactional task contains:

snapshot σ: transactional state on creation

local store τ: local modifications

$$\mathrm{T_x} \cup \langle f_p, \sigma, \tau, \mathrm{F_s}, \quad \mathrm{F_j}, \mathcal{E}[\texttt{fork}\ e] \rangle$$
$$\Rightarrow_{\mathrm{tf}} \mathrm{T_x} \cup \langle f_p, \sigma, \tau, \mathrm{F_s} \cup \{f'\}, \mathrm{F_j}, \mathcal{E}[f_c] \rangle \cup \langle f_c, \sigma :: \tau, \varnothing, \varnothing, \mathrm{F_j}, e \rangle$$
with f_c fresh

`join` merges changes




```
(atomic
  …
  (join child))
```

merge local store τ' of child into parent

$$\mathrm{T_x} \cup \langle f_p, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[\texttt{join}\ f_c] \rangle \cup \langle f_c, \sigma', \tau', \mathrm{F'_s}, \mathrm{F'_j}, v \rangle$$
$$\Rightarrow_{\mathrm{tf}} \mathrm{T_x} \cup \langle f_p, \sigma, \tau :: \tau', \mathrm{F_s} \cup \mathrm{F'_s} \cup \mathrm{F_j} \cup \mathrm{F'_j} \cup \{f_c\}, \mathcal{E}[v] \rangle \cup \langle f_c, \sigma', \tau', \mathrm{F'_s}, \mathrm{F'_j}, v \rangle$$
$$\text{if } f_c \notin \mathrm{F_j} \text{ and } \mathrm{F'_s} \subseteq \mathrm{F'_j}$$

Conflict resolution function:

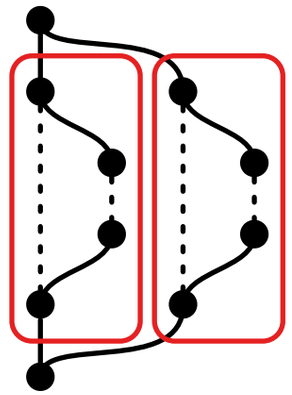
```
(ref 0 resolve)
```

$\texttt{resolve} :: \mathsf{T} \times \mathsf{T} \times \mathsf{T} \to \mathsf{T}$

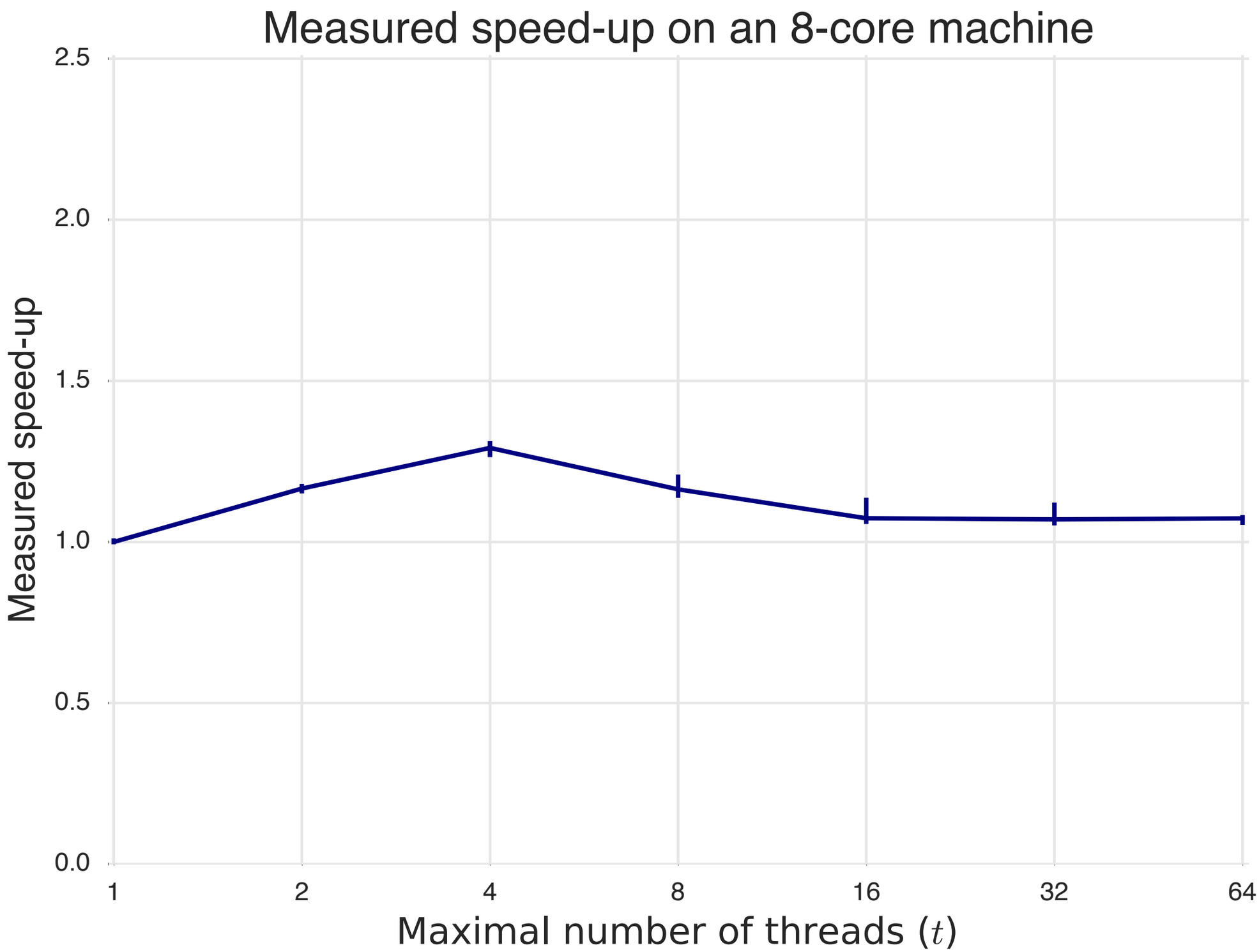
```
(defn resolve [o p c] c)
(defn resolve [o p c] p)
(defn resolve [o p c] (min p c))
(defn resolve [o p c] (+ p c))
(defn resolve [o p c] (error "merge conflict"))
```

Properties of transactional tasks

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

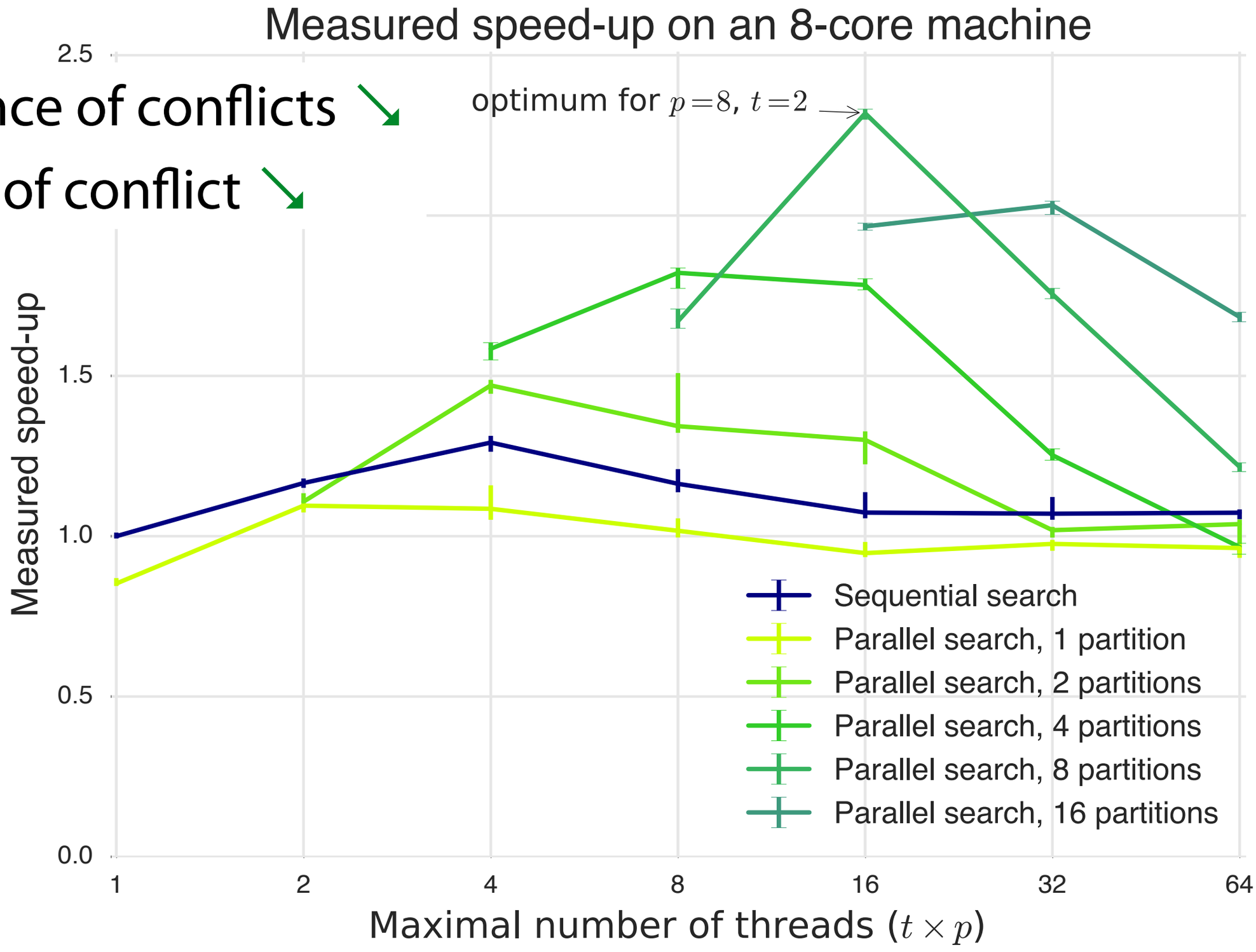


Evaluation: Labyrinth

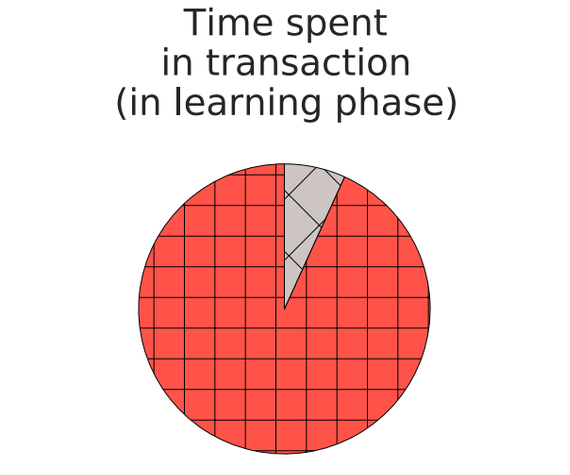


Evaluation: Labyrinth

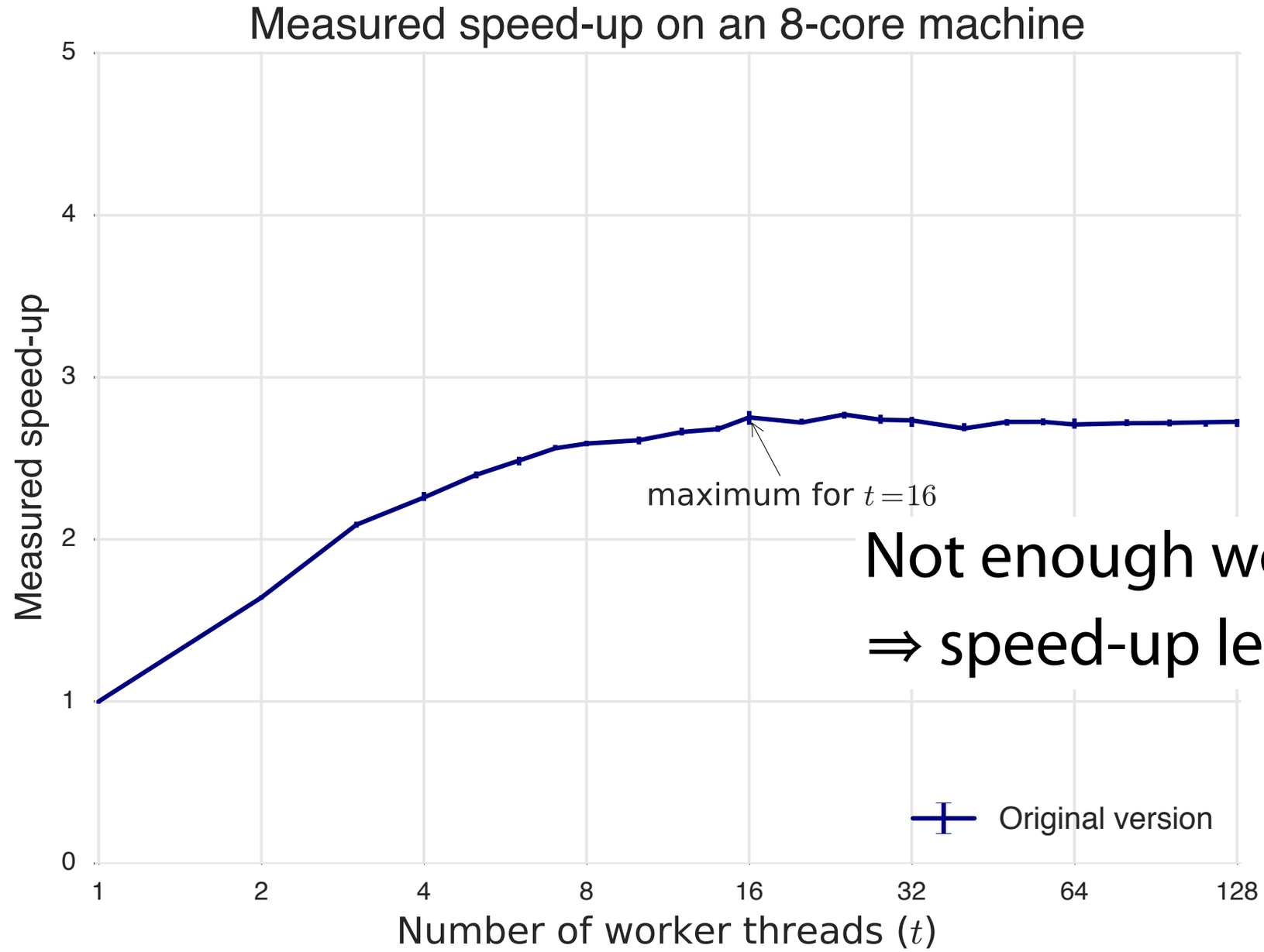
- Chance of conflicts ↘
- Cost of conflict ↘



Evaluation: Bayes



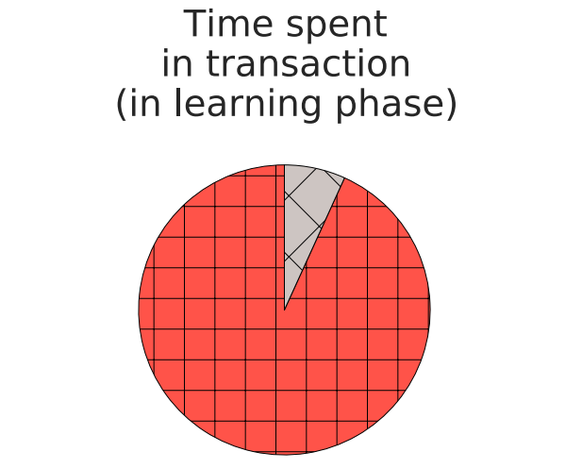

```
(atomic
  …
  (for [from-id (range (:n-var adtree))]

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



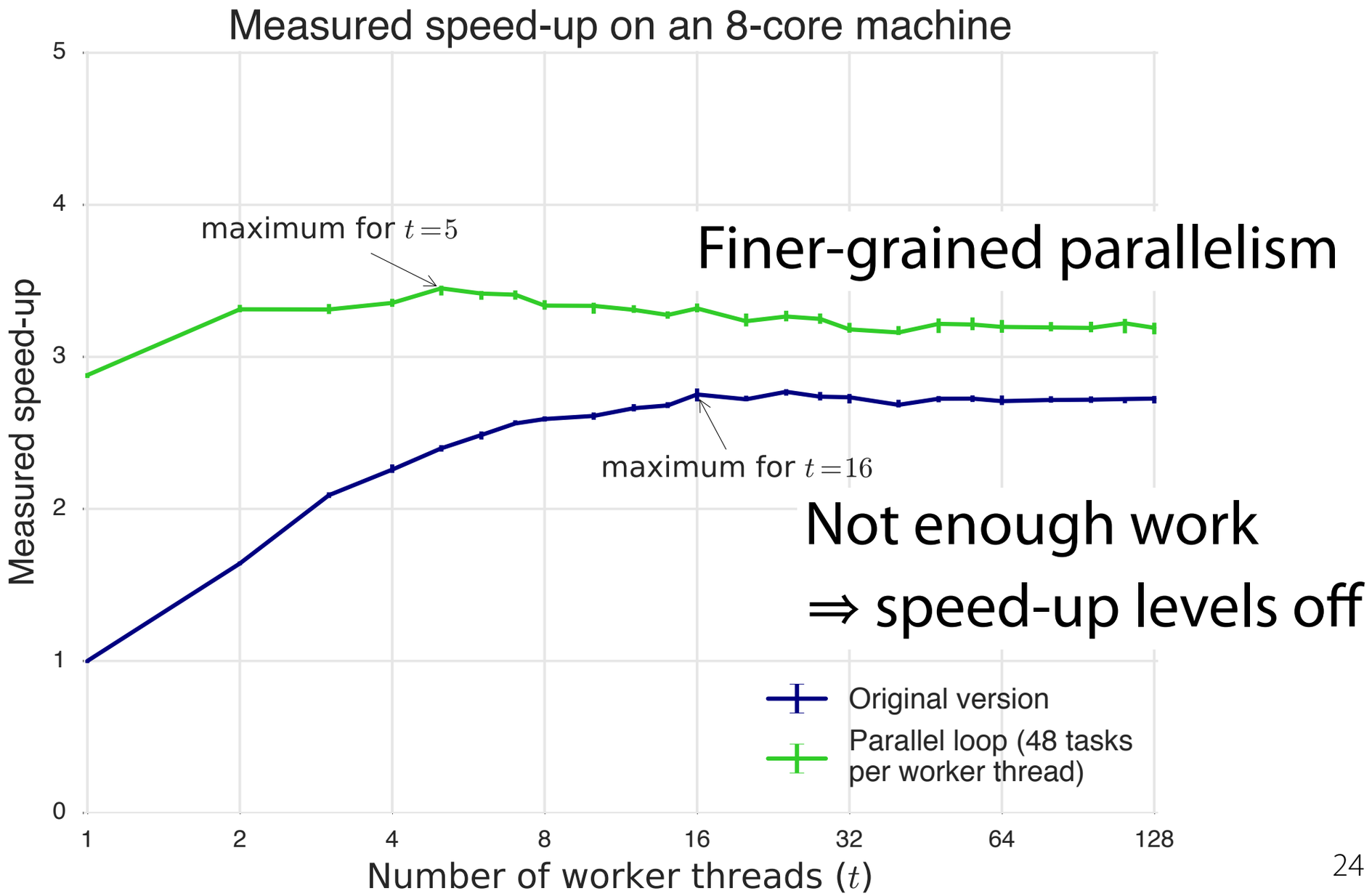
Not enough work

⇒ speed-up levels off

Evaluation: Bayes




```
(atomic
  …
  (for [from-id (range (:n-var adtree))]
    (fork
      (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/

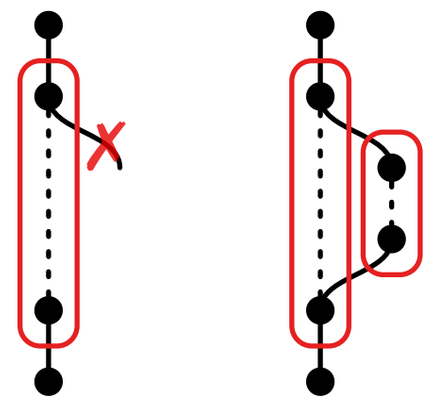
Details in paper

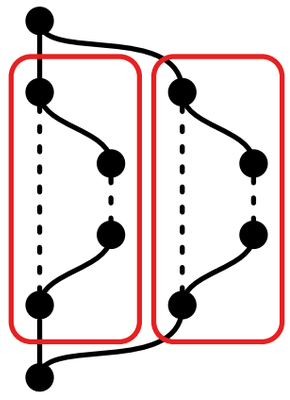
Summary

Parallelism in a transaction is useful
for programs with **long transactions**
But currently:

↗ **not allowed** (Haskell)

↘ **not serializable** (Clojure, Scala)



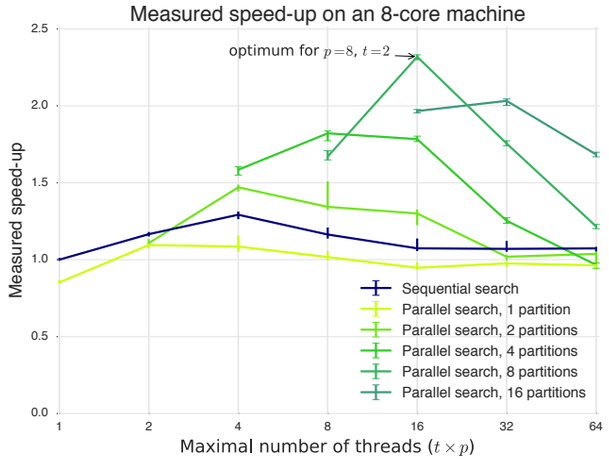


Idea: **transactional tasks**

- safe access to encapsulating transaction
- serializable, coordinated, determinate

Benefits:

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



Transactional Tasks vs. Nested Parallel Transactions

Guarantees in transaction

- NPT: `(atomic (fork …))` → race conditions possible
- NPT: `(atomic (fork (atomic …)))` → 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

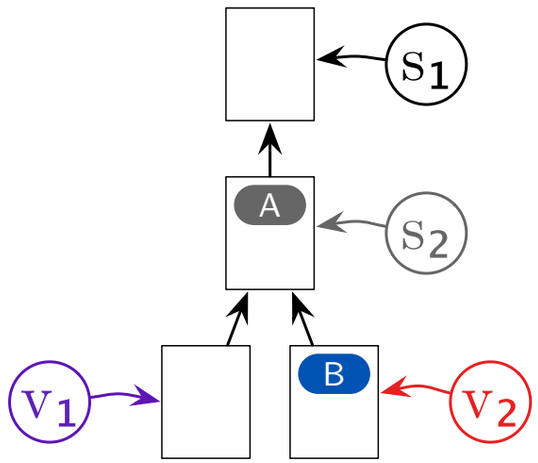
⇒ resolve low-level conflicts using conflict resolution functions

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.