Transactional Tasks: Parallelism in Software Transactions

Janwillem Swalens

Futures

(**`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

`(`**`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)))))
```

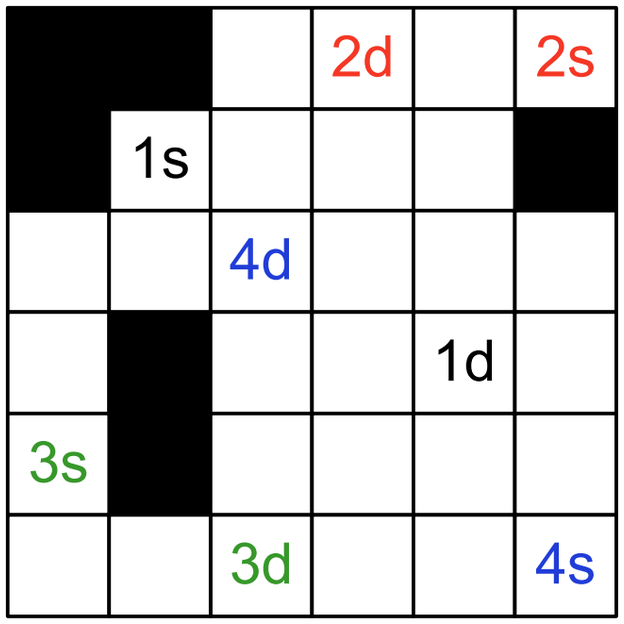
Transactions

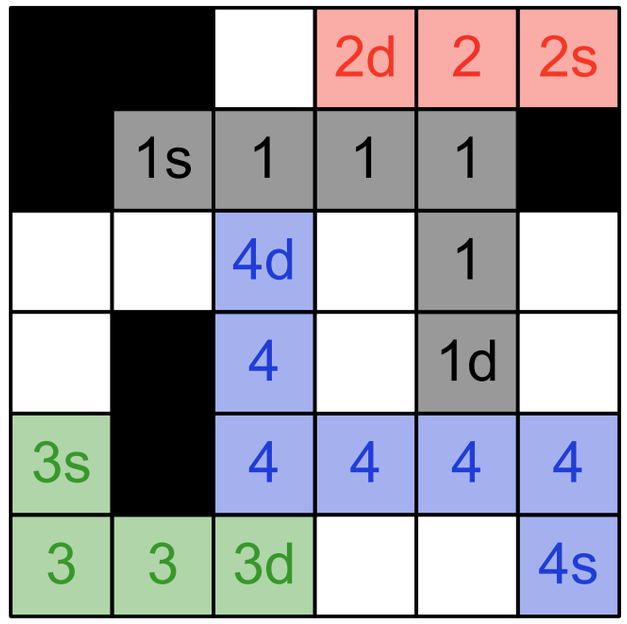
`(ref v)`
`(atomic e)`
`(deref r)` or `@r`
`(ref-set r v)`

serializability

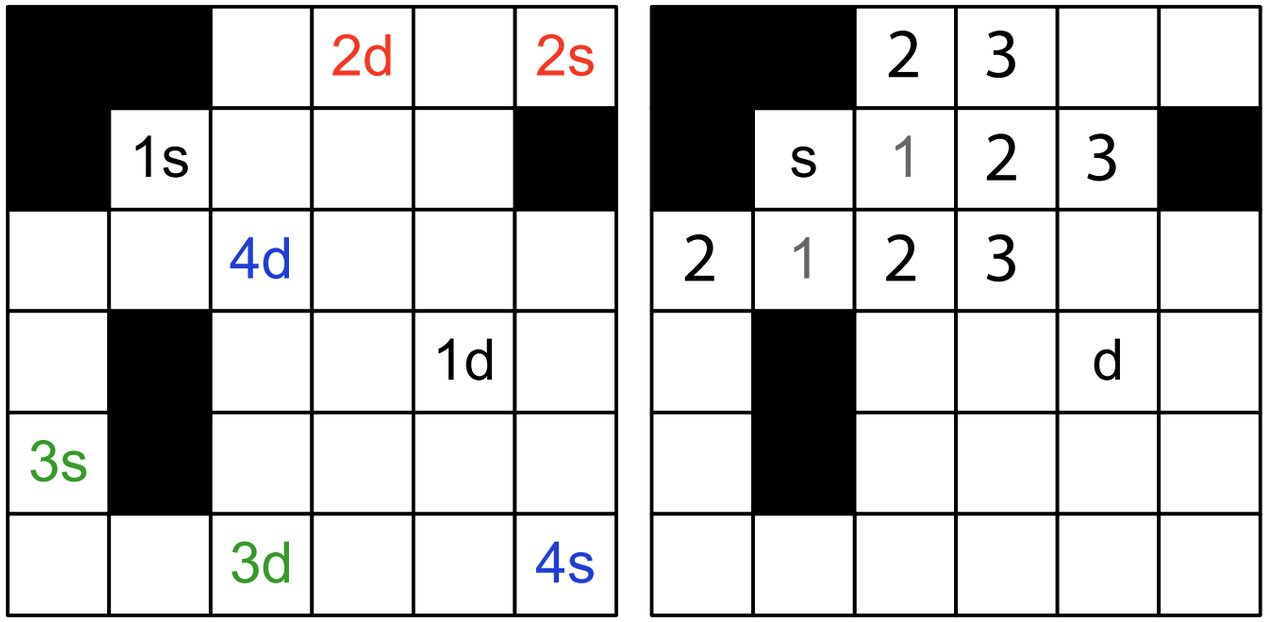
```
(def checking (ref 100))
(def savings  (ref 500))
(fork
  (atomic
    (ref-set checking (- @checking 10))
    (ref-set savings  (+ @savings  10))))
(fork
  (atomic
    (println “You own €” (+ @checking @savings))))
```

Example: Labyrinth





Example: Labyrinth



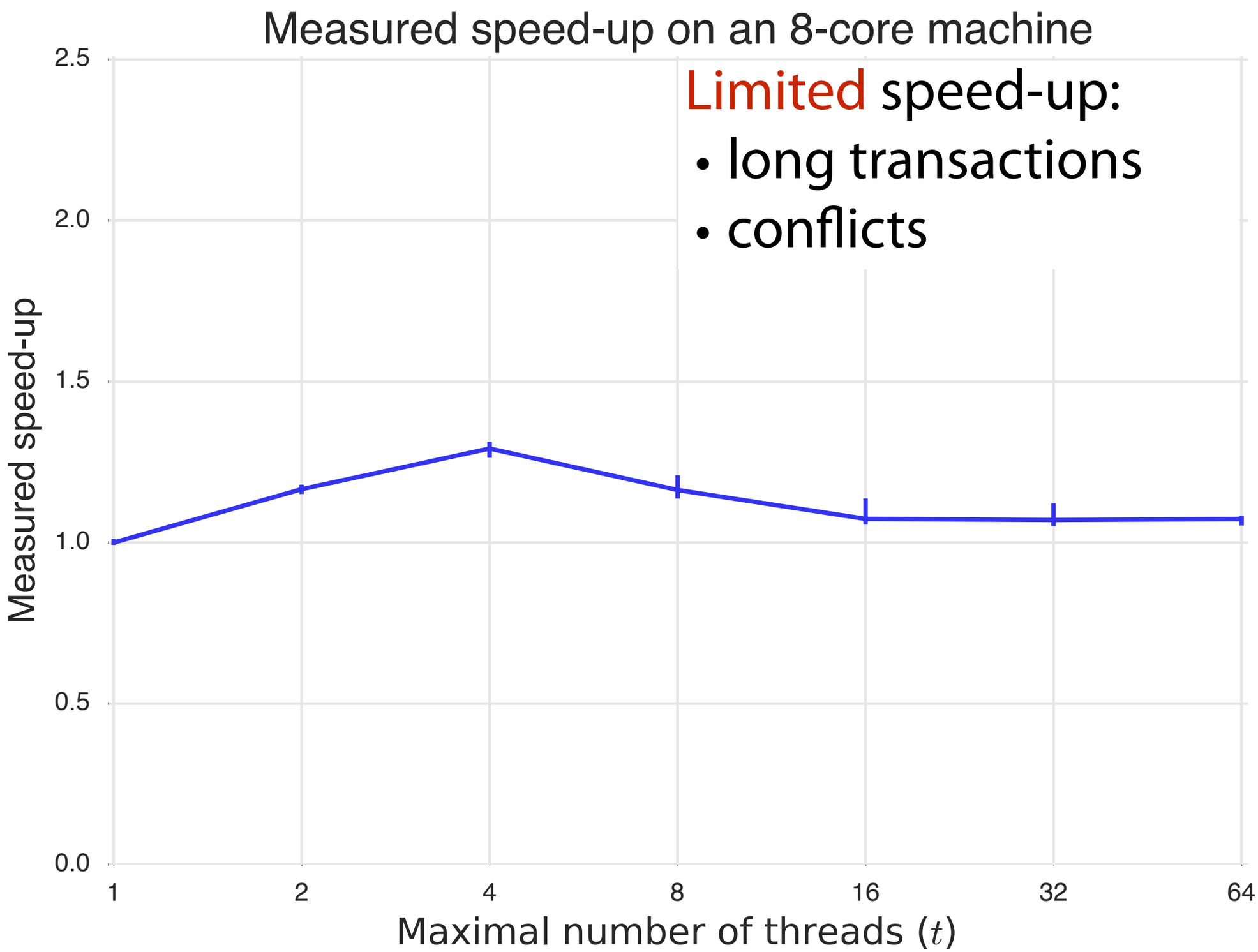

```
; for each (src, dst) pair:

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

breadth-first search

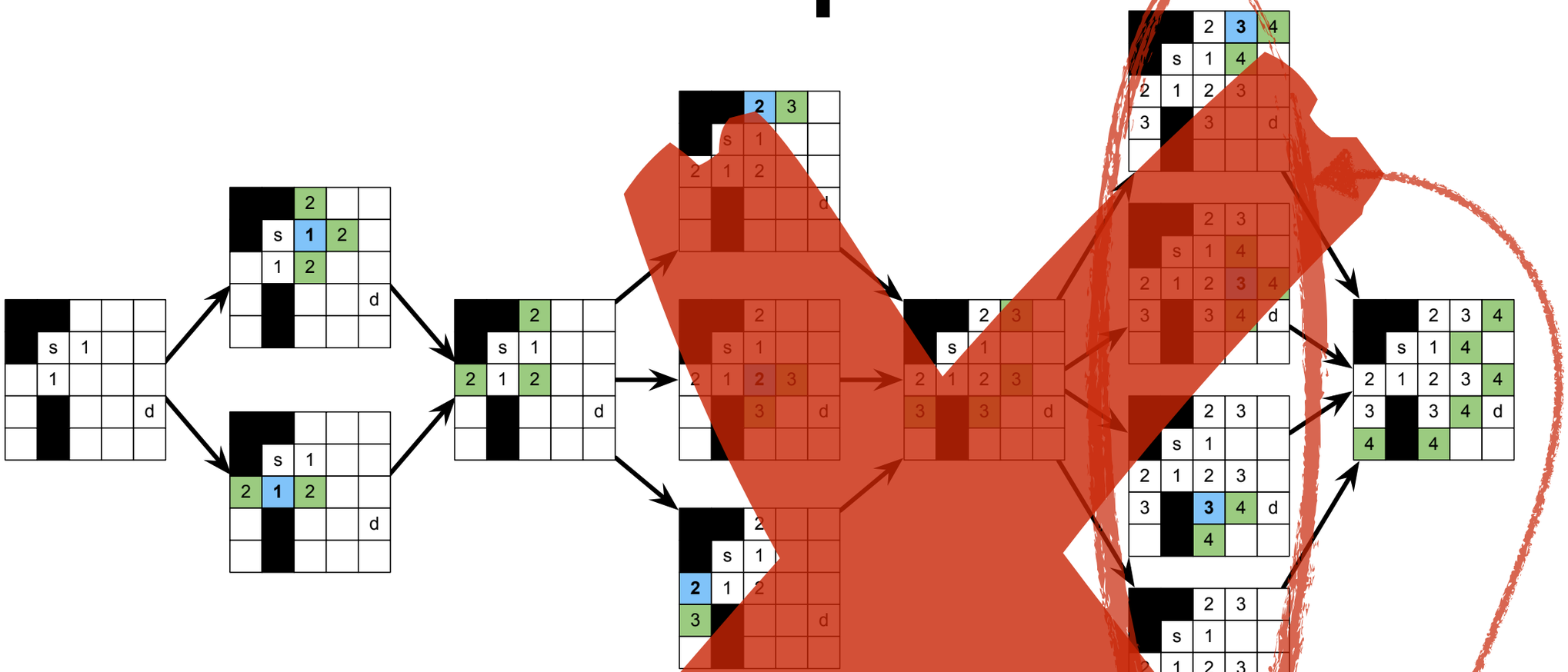
```
(defn expand [src dst
  (loop [q
    (if (empty? q
      false

      true

      (rest q
      (expand-point (first q)
```

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

Labyrinth speed-up



Solution: parallel search



expand
in parallel

```
; for each (src, dst) pair:
(atomic
  (let [local-grid (copy grid)]
    (expand src dst local-grid)
    (add-path grid
      (traceback local-grid dst))))
```

```
(defn expand [src dst
  (loop [q #{src}]
    (if (empty? q
      false
      (if (contains? q dst)
        true
        (recur
          (expand-step q
```

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

parallel breadth-first search

Does **not** work!

Problems

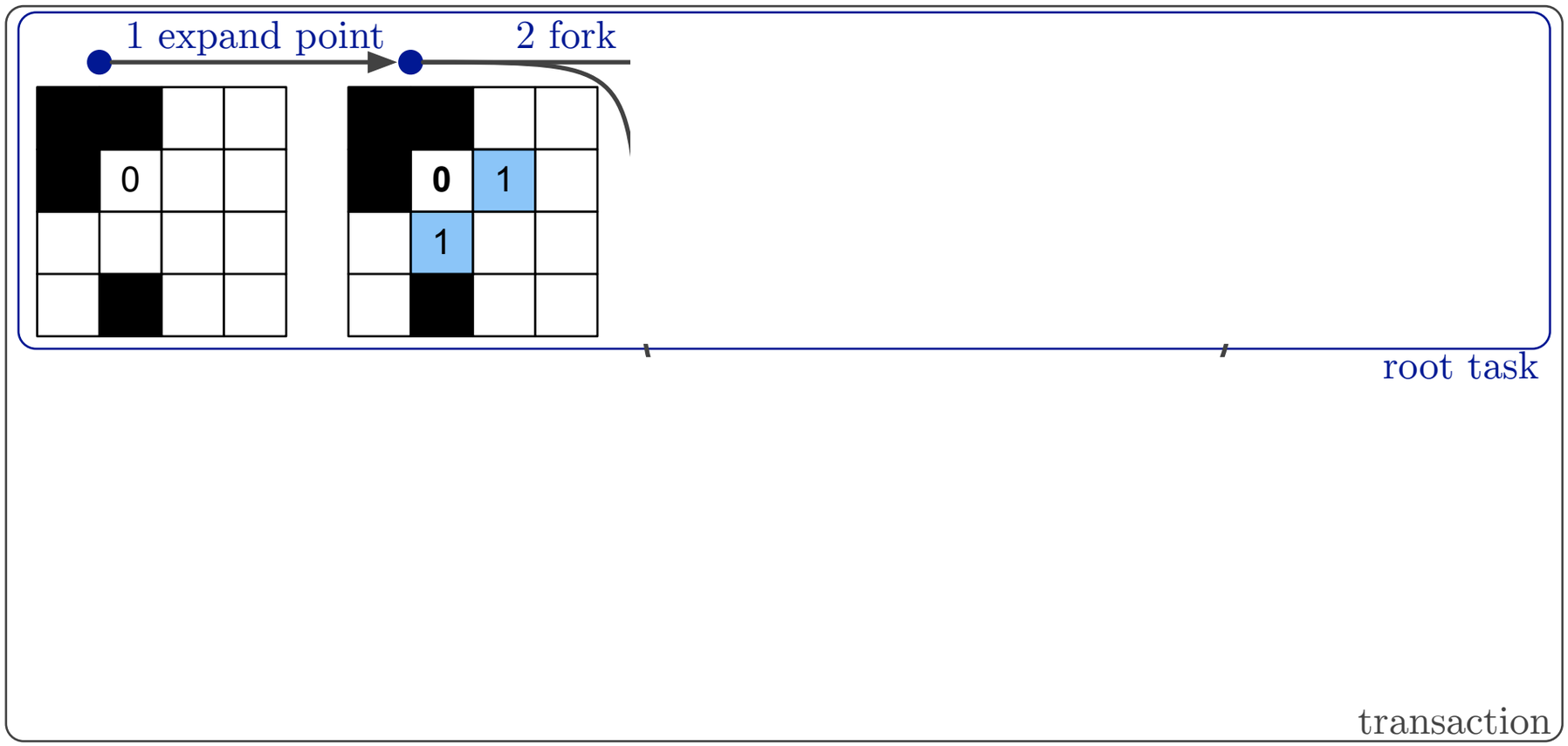
- Threads in tx prohibited (Haskell)
  ⇒ **parallelism limited**

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

- Threads in tx allowed but don’t share context (Clojure, Scala)
  ⇒ **serializability violated**

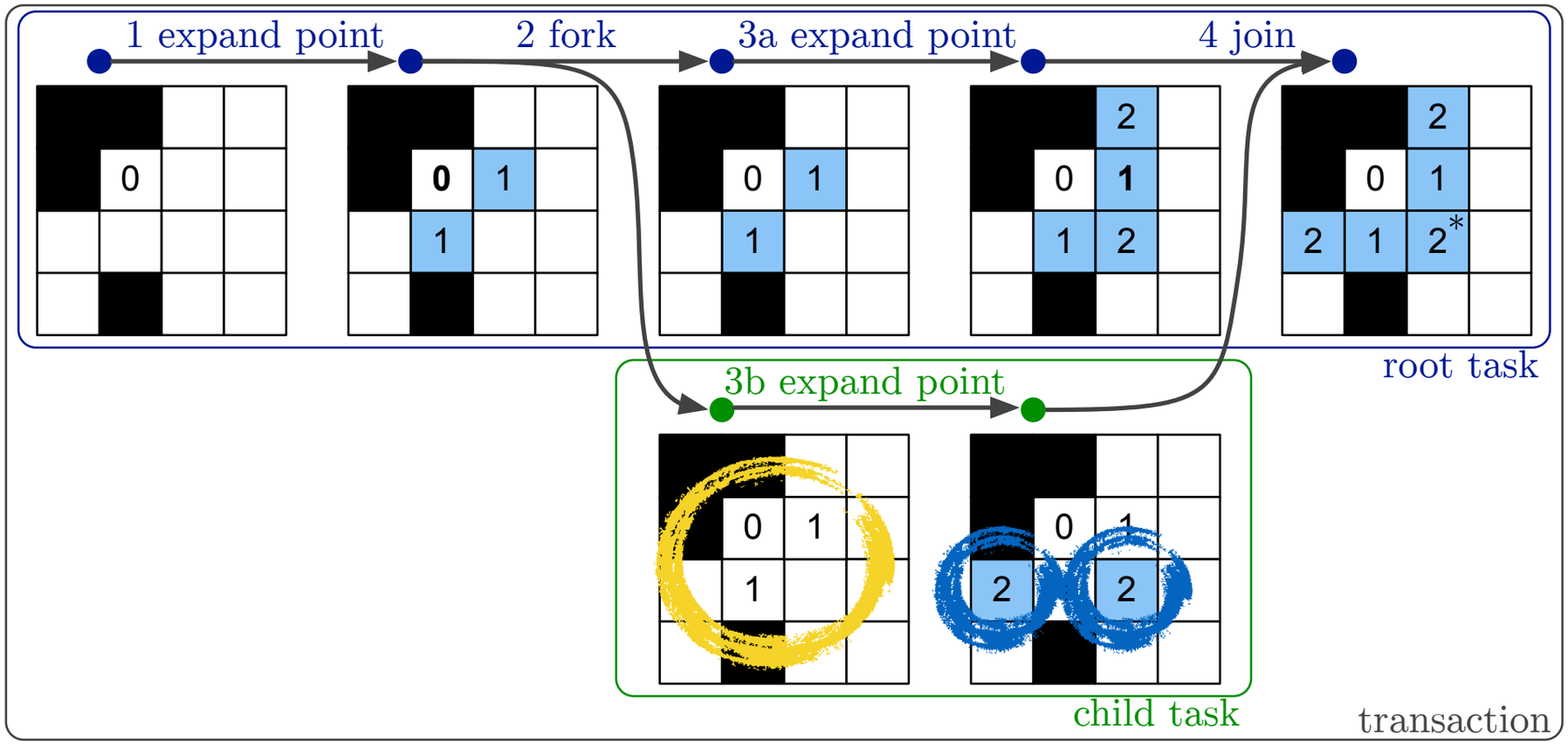
```
(atomic
  (fork
    (atomic …)))
```

Transactional Tasks



- `fork`: copy the transactional state
- each task runs in isolation
- `join`: merge changes

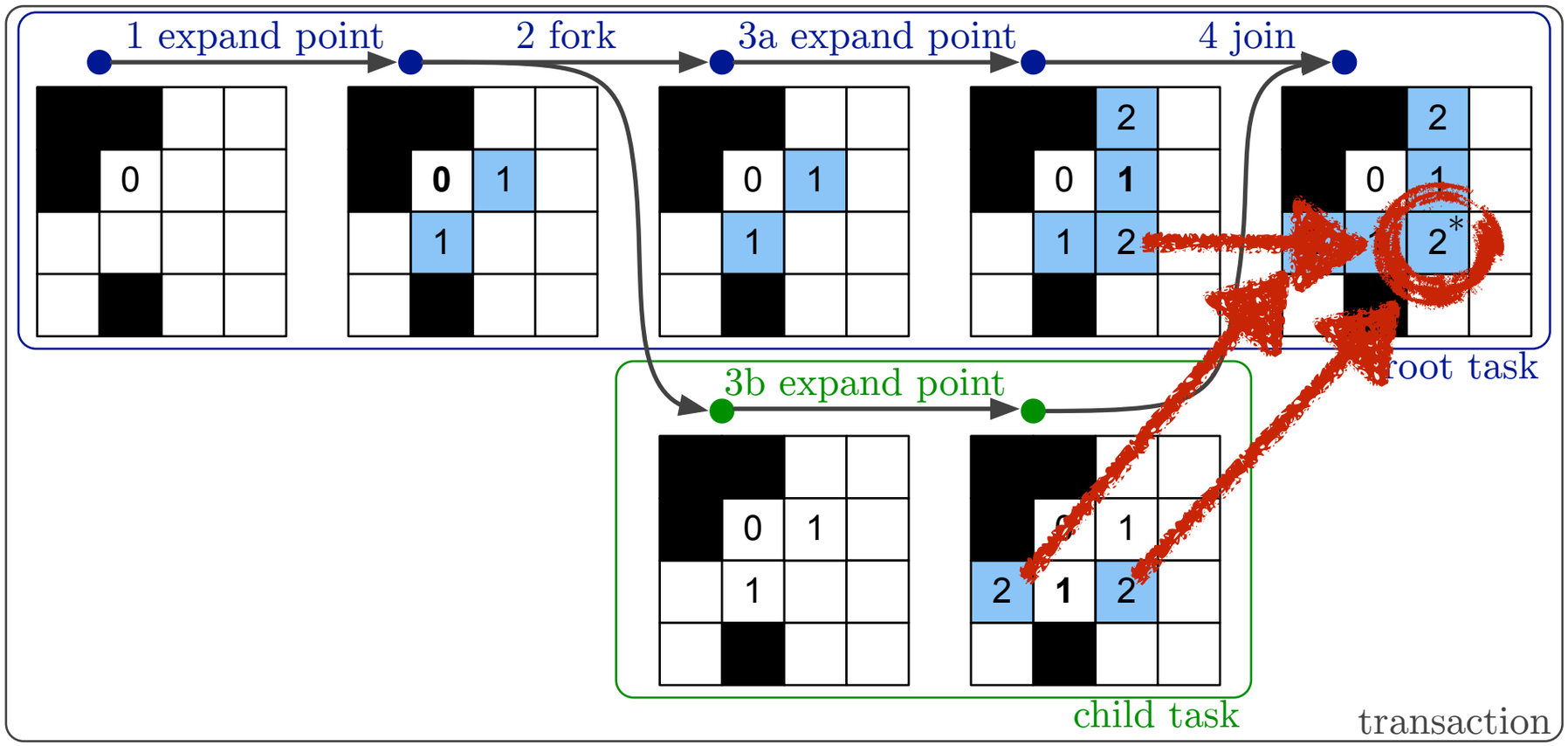
fork



snapshot = tx state when created

local store = local modifications

`join` & conflicts



Merge local store

Conflict resolution function: `(ref 0 resolve)`

`resolve` :: $\top \times \top \times \top \rightarrow \top$

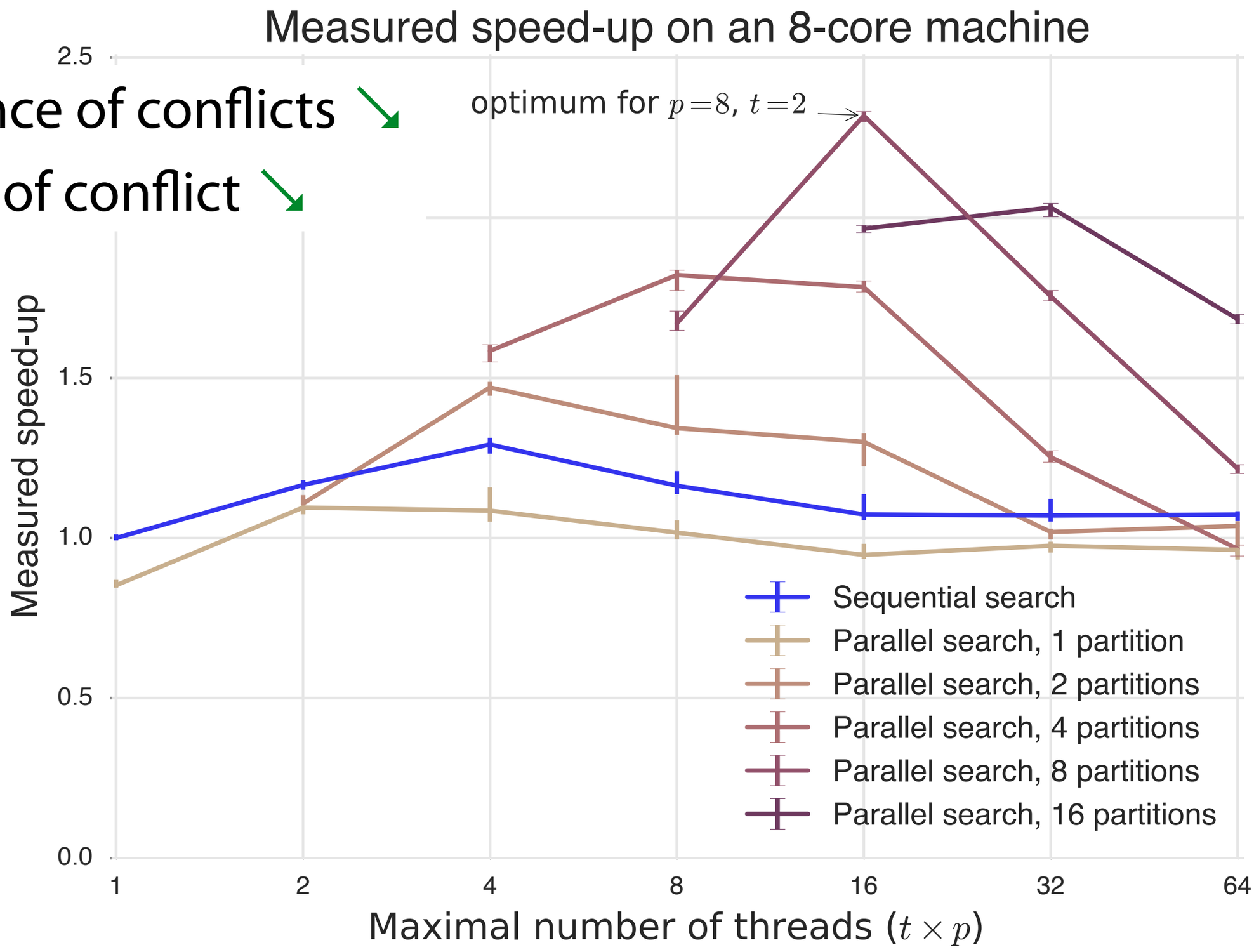
```
(defn resolve [o p c] c)
(defn resolve [o p c] p)
(defn resolve [o p c] (min p c))
(defn resolve [o p c] (+ p c))
(defn resolve [o p c] (error “Conflict on merge”))
```

Properties

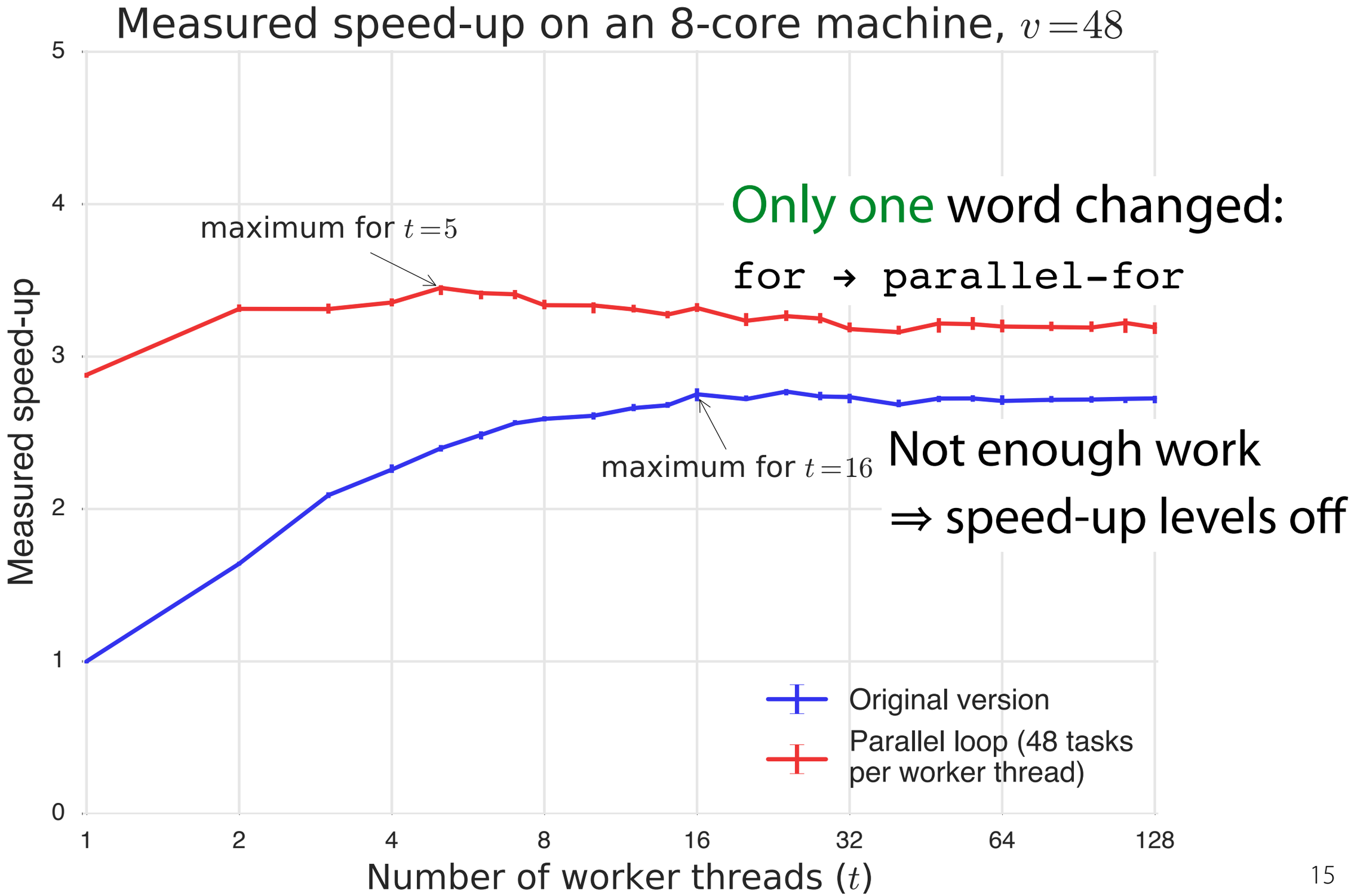
- **In-transaction parallelism** possible
- **Serializability** of transactions
- **Coordination** of tasks: all or none
- In-transaction **determinacy**
- Non-transactional tasks **unmodified**

Validation: Labyrinth

- Chance of conflicts ↘
- Cost of conflict ↘



Validation: Bayes



Only one word changed:

```
for → parallel-for
```

Not enough work

⇒ speed-up levels off

Take-away from experiments

- Labyrinth: parallelize search algorithm
  ⇒ fewer & cheaper conflicts
- Bayes: more fine-grained parallelism
- Low developer effort

Summary

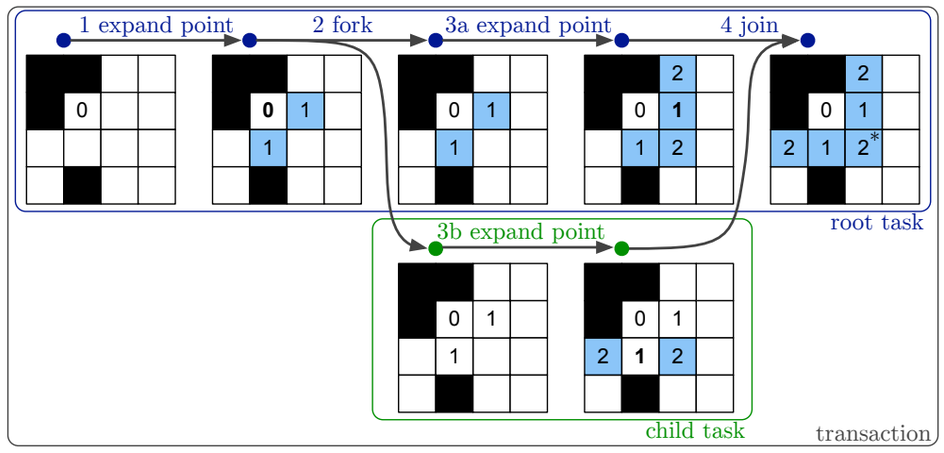
Parallelism in a transaction is **useful**

But currently:

↗ **not allowed** (Haskell)

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

```
atomically $
  do { forkIO … }

(atomic
  (fork
    (atomic …)))
```

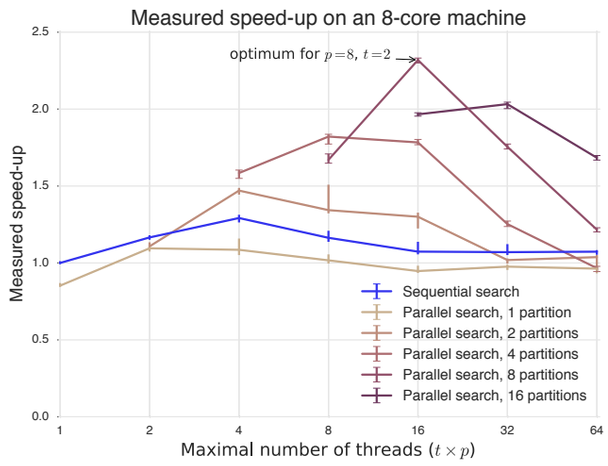


Idea: **transactional tasks**

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

Benefits:

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



More details:

- J. Swalens et al. "Transactional Tasks: Parallelism in Software Transactions", ECOOP 2016
- https://github.com/jswalens/transactional-futures/