# STM-based concurrent heaps

Mateusz Machalica

## Heap (a priority queue)

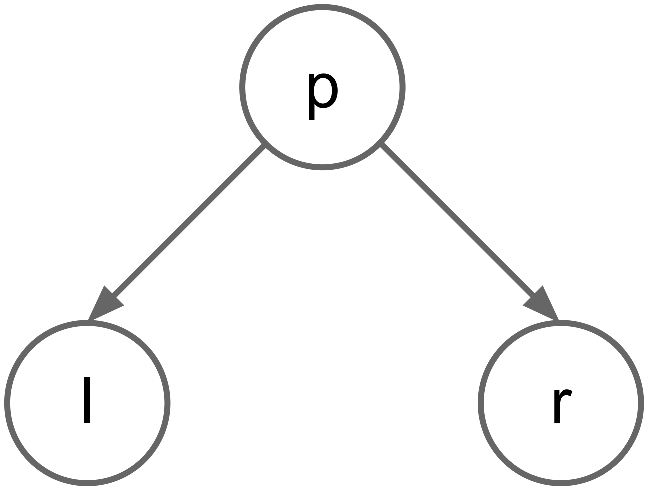
- for simplicity: min-heap
- operations on heap of size n
  - `min()` in `O(1)` time
  - `push(e)` in `O(log(n))` time
  - `pop()` in `O(log(n))` time, possibly amortized

## Take 1: coarse-grained locking

- take array-based heap implementation
- replace array with `TArray Int e`
  - equivalently with `MArray Int (TVar e)`
- replace `size` with `TVar Int`
- put everything in a transaction

## Take 1: correctness (trivial invariant)

“A heap has the heap property after each transaction.”



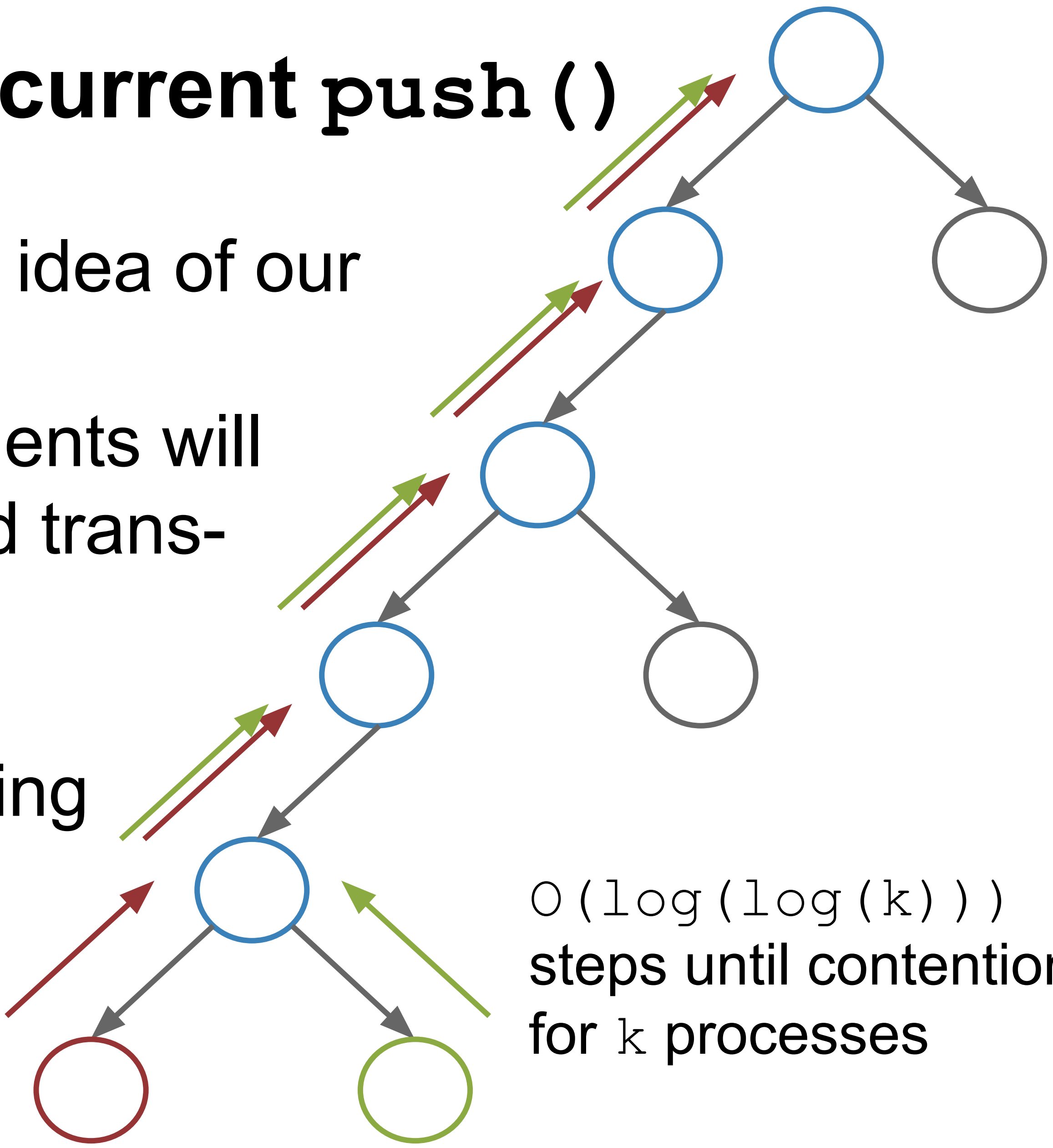
p <= min(l, r)

## Take 1: problems

- full contention on `size` required for `push()` and `pop()`
- consecutive `push()` operations follow nearly the same path
  - despite inserted values
  - long operations shuffle many elements
  - STM creates data dependencies for elements on the path
  - two updates cannot “follow” each other

## Take 1: concurrent `push()`

- STM has no idea of our invariant
- altered elements will force second transaction to be rejected
- full sequencing



`O(log(log(k)))` steps until contention for `k` processes

## Problems to fight with

- STM thinks that if element is touched then it necessarily breaks the transaction
    - very strong invariant
    - but not always true
    - and hard to weaken
- arrays `#2imperative4us`
    - cannot make things immutable
    - but a tree would be nicer for dynamic sized heaps
- updating `size` cannot end up in a global transaction

## Assumption

“Mutual exclusion between `push()` and `pop()` operations.”

In case we drop it:

- what is the semantics?
- how to avoid contention in the root?
- hard to maintain any invariants
- not necessary for most applications I can think about...

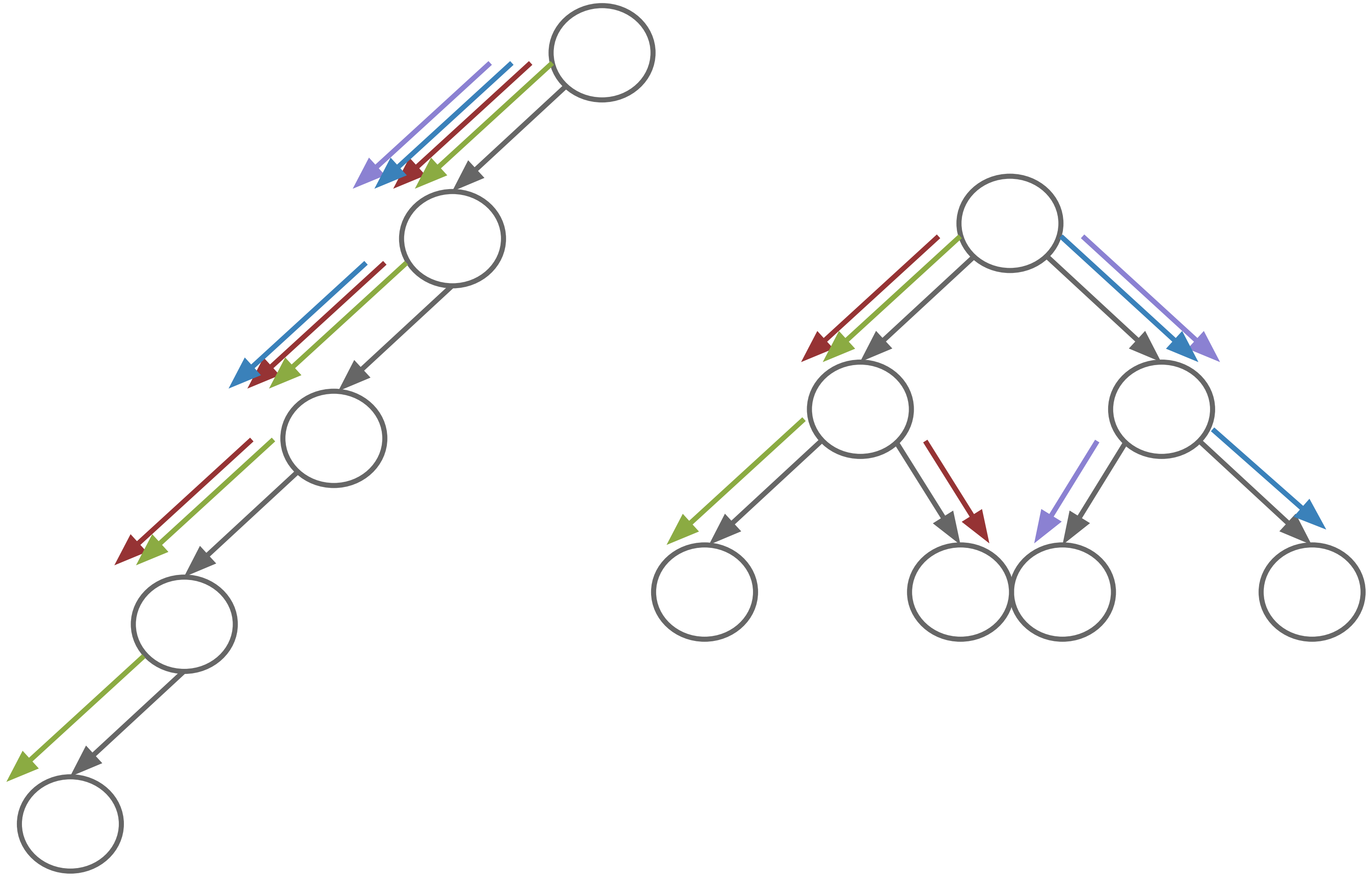
## Take 2: fine-grained locking

- implement heap as a tree
    - all operations go top-down
    - balancing achieved by maintaining `size` of a subtree rooted at each node
- single transaction covers `O(1)` nodes
    - we need some fancy invariant
    - if the old one works, we can sort in `O(n)` time in comparison model `#impossibru`

## Why it *should* work

- updates can proceed one after another on the same path
  - we do not look back after the transaction
  - each update waits for preceding to leave the root
  - `k` updates take `O(k)` time if `log(n)=O(k)`
- or diverge as early as possible
  - each update descents in a different subtree
  - `k` updates take `O(k)` time if `log(log(n))=O(k)`
- in sequential case
  - `k` updates take `O(k*log(n))`

## Why it *should* work



## The `push()` operation

- in transaction
  - choose a child node with smaller `size` field
  - update `size`
  - if the node is empty, insert our value and finish
  - swap element to be inserted `e` with the one in current node `x` if `e < x`
- recurse into the chosen node
  - after committing the transaction

## Correctness of `push()`

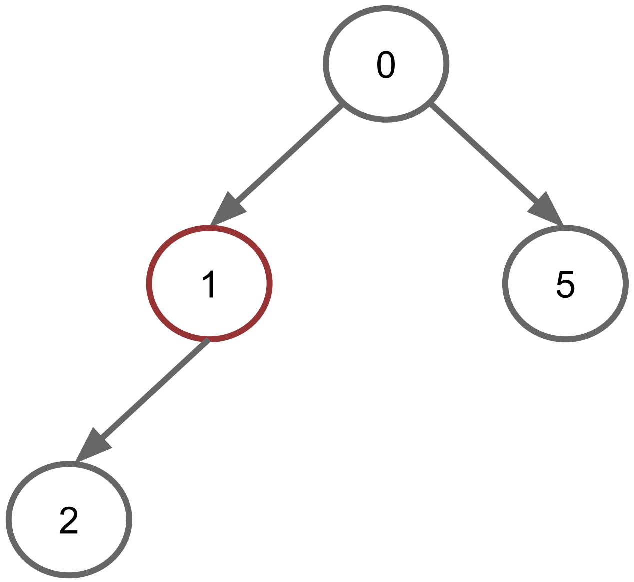
- invariant of a transaction
  - the simple invariant works for the tree, but need to cover the extra element `e`
  - “The heap property holds for each tree node and the element to be inserted `e` is greater than all elements on the path from the current node to the root”
- in absence of `pop()` operations
  - elements in each node are non-increasing
  - traversing a node = it will be at most as big as we have seen until the next deletion

## The `pop()` operation

- in transaction
  - replace root’s element with “a gap”
  - decrease `size`
- in transaction
  - choose a child node with smaller element
  - decrease `size`
  - move element in the node to the parent, place gap here
- recurse into the chosen node
  - after committing the transaction
  - only if it’s subtree is non-empty

## The `pop()` operation - caveat

- if a root has a gap `retry`
- if any of the child nodes have gaps and `size > 0`, then `retry`
  - there is at least one node having gap-free children
  - no deadlock
- are these necessary?
  - we don't know which element will end up here
  - possibly the smallest among children of the node



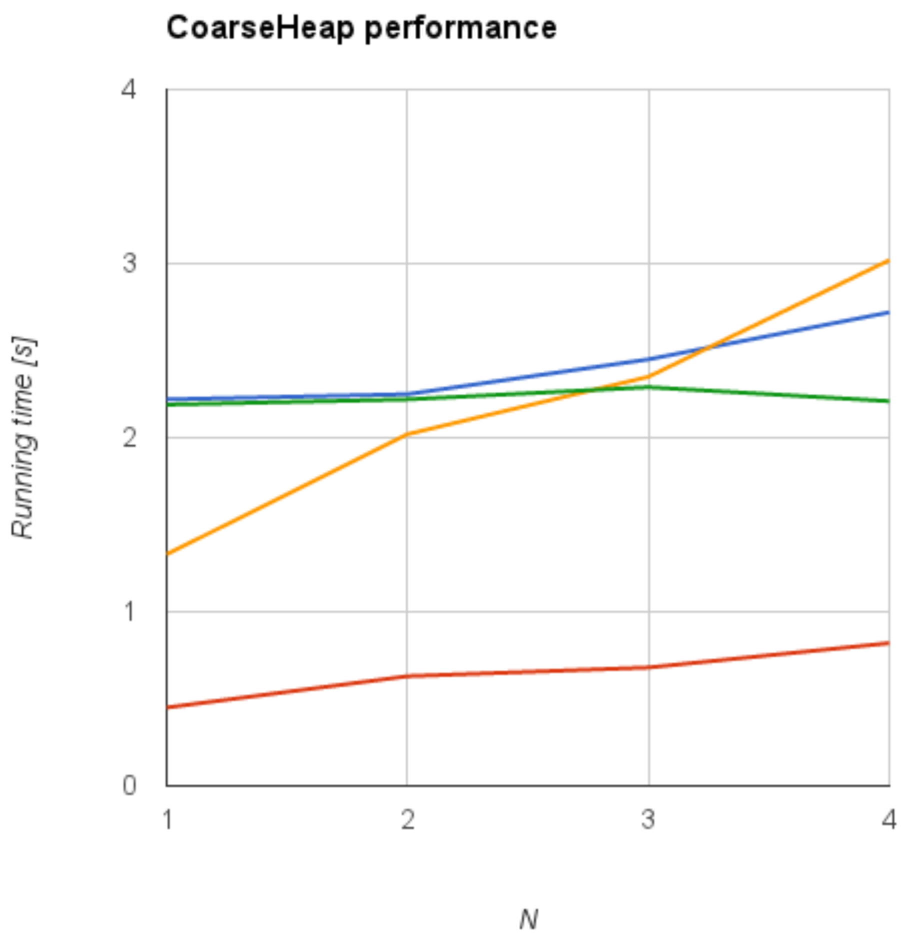
## Correctness of `pop()`

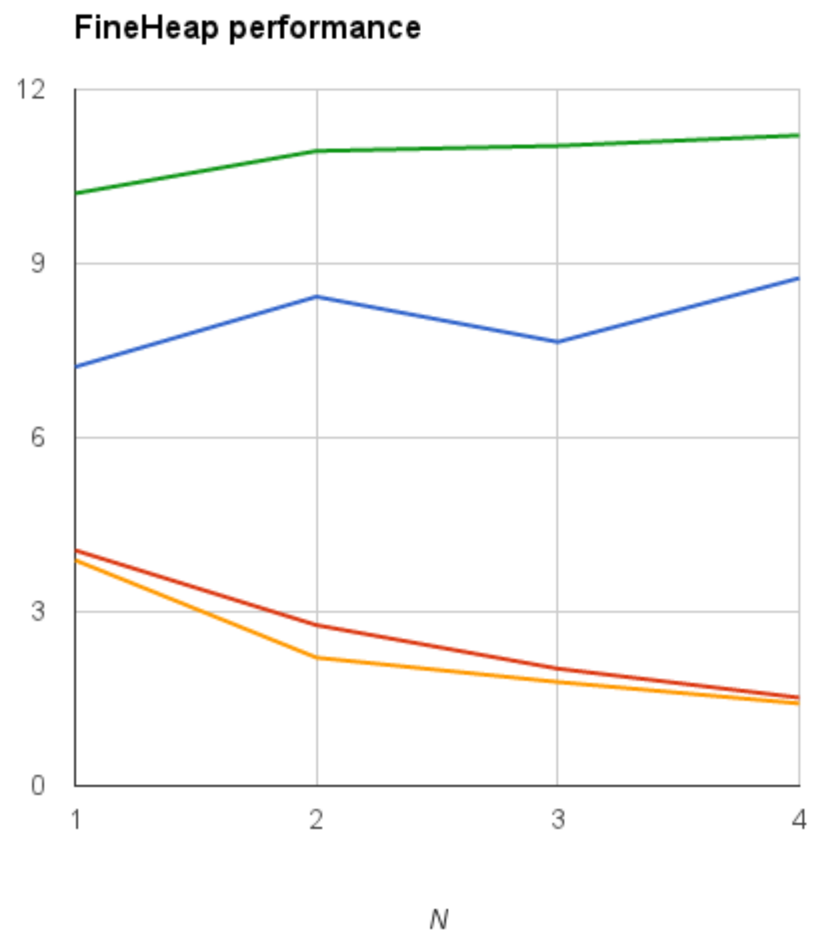
- invariant of a transaction
  - the simple invariant no longer works!
  - “The heap property holds for each tree node, when one assigns each gap-node an element from it’s parent (recursively)”
- for each gap we have a processor trying to push it down the tree
- the conditions from previous slide ensure
  - that we know upper bound on each subtree before replacing root’s gap
  - we never steal the gap from someone else

## Performance comparison

- initially 50k elements in the heap
- each (of W) writer inserts 50k / W
- each (of R) reader removes 50k / R
- different scenarios (parametrized by N)
    - W = max(1, N - 1), R = 1
    - W = N, R = 0
    - W = 0, R = N
    - W = 1, R = 1, sequentially, i.e. first all writes, then all removals
- averaged over 5 random seeds

## Performance comparison





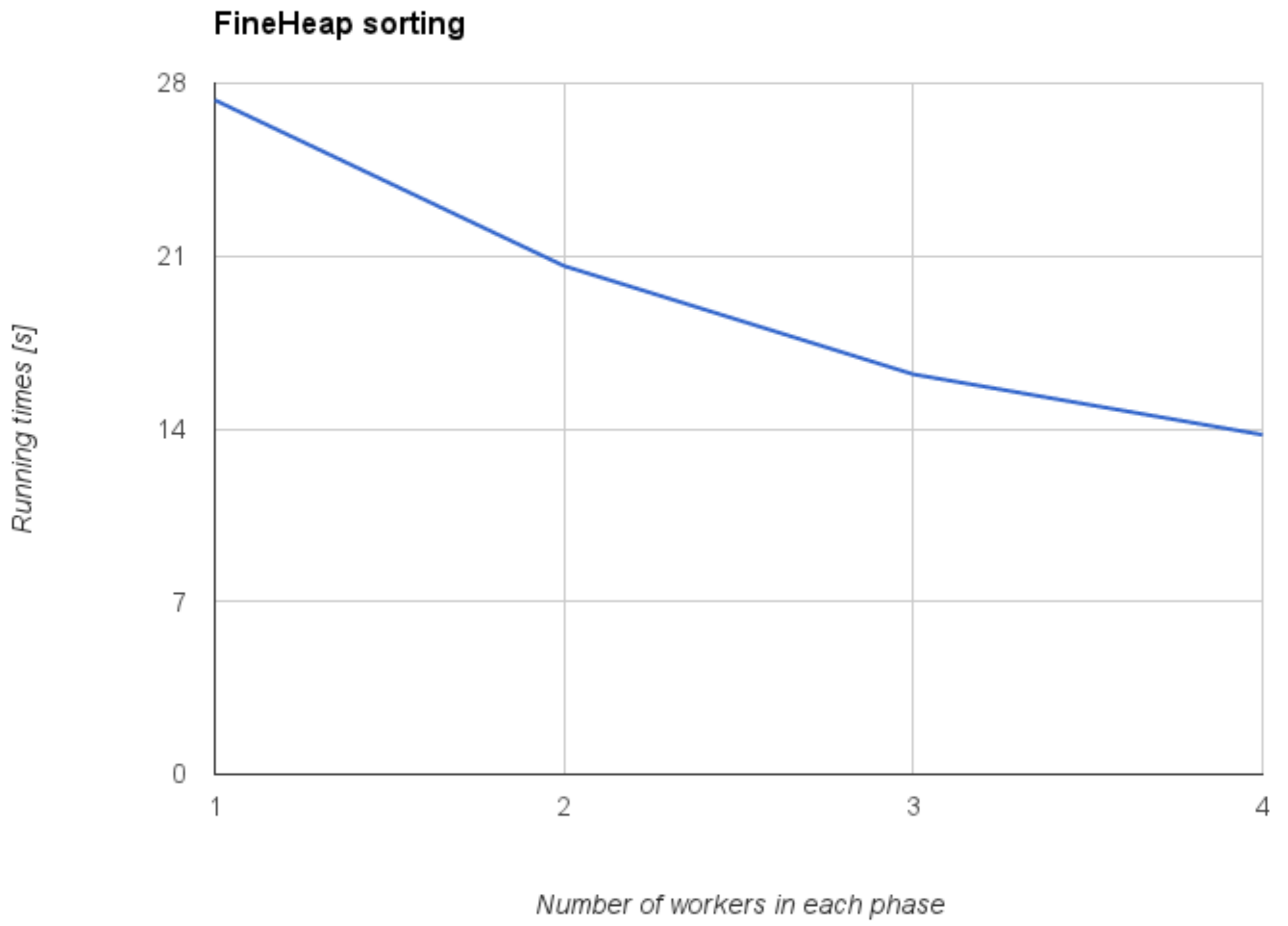
## Application: concurrent sorting

- insert all elements into a heap
  - using `N` processors
  - each inserts around `n/N` elements
- initialize `n` element `TArray Int (Maybe e)` with `Nothing`
- assign threads to array cells in round-robin fashion
  - waits for previous cell to be filled
  - replaces heap’s root with a gap, places the element in the array
  - pushes the gap down the tree
  - proceeds to it’s next index

# Performance: concurrent sorting

- 100k random elements
- phase 1 (insertion)
  - each (of N) writer inserts 100k / N elements
- phase 2 (removal)
  - each (of N) reader
    - waits for previous cell of output array to be written
    - takes element from the heap
    - writes to the output array into its cell
  - cells assigned to workers in round-robin fashion

## Performance: concurrent sorting


FineHeap sorting
Running times [s]
Number of workers in each phase