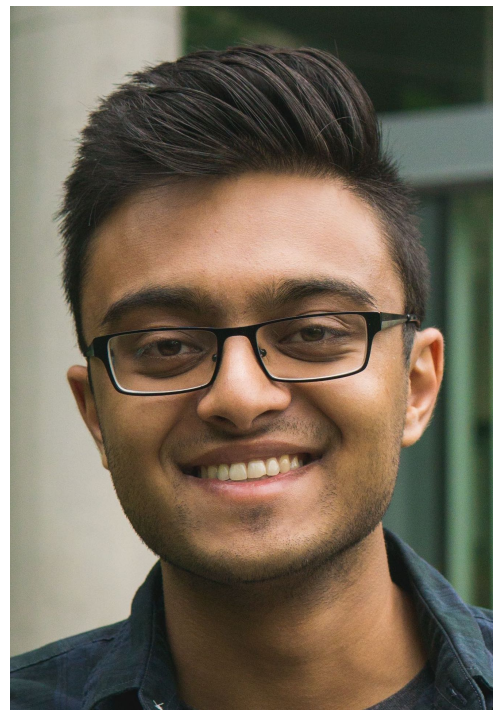
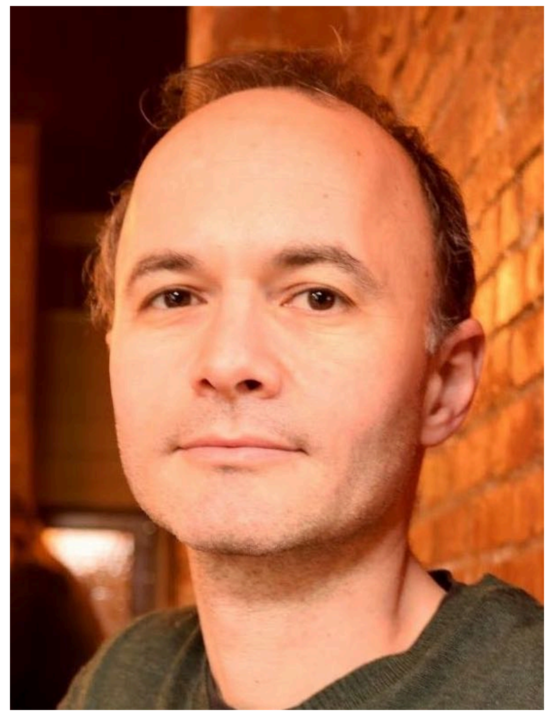
Disentanglement: Provably Efficient Parallel Functional Programming




Sam Westrick

Jatin Arora

Rohan Yadav

Umut Acar

Matthew Fluet

Parallel Programming

imperative

mutability
manual memory management
race conditions

immutability
automatic memory management
deterministic by default

functional

fast

can parallel functional programming be fast and scalable ?

slow?

Parallel Programming

imperative

mutability
manual memory management
race conditions

immutability
automatic memory management
deterministic by default

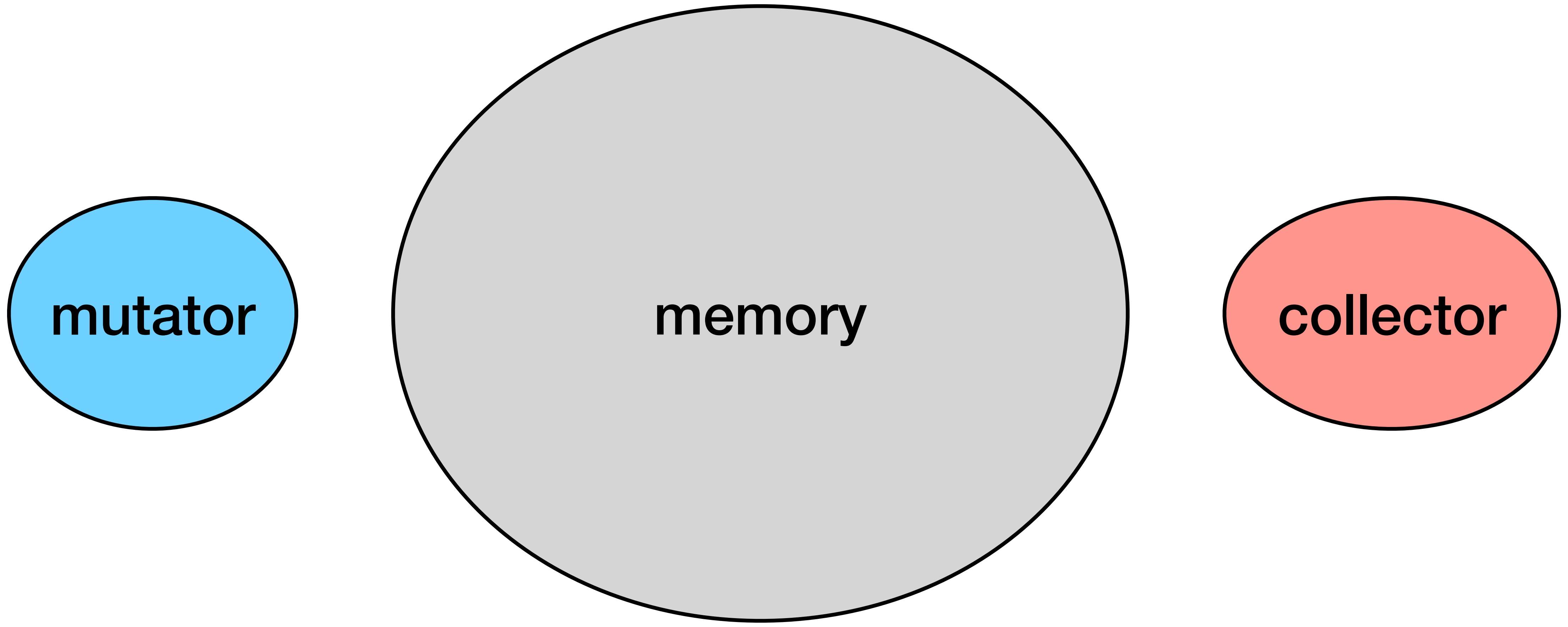
functional

high rate of allocation
heavy reliance on GC

fast

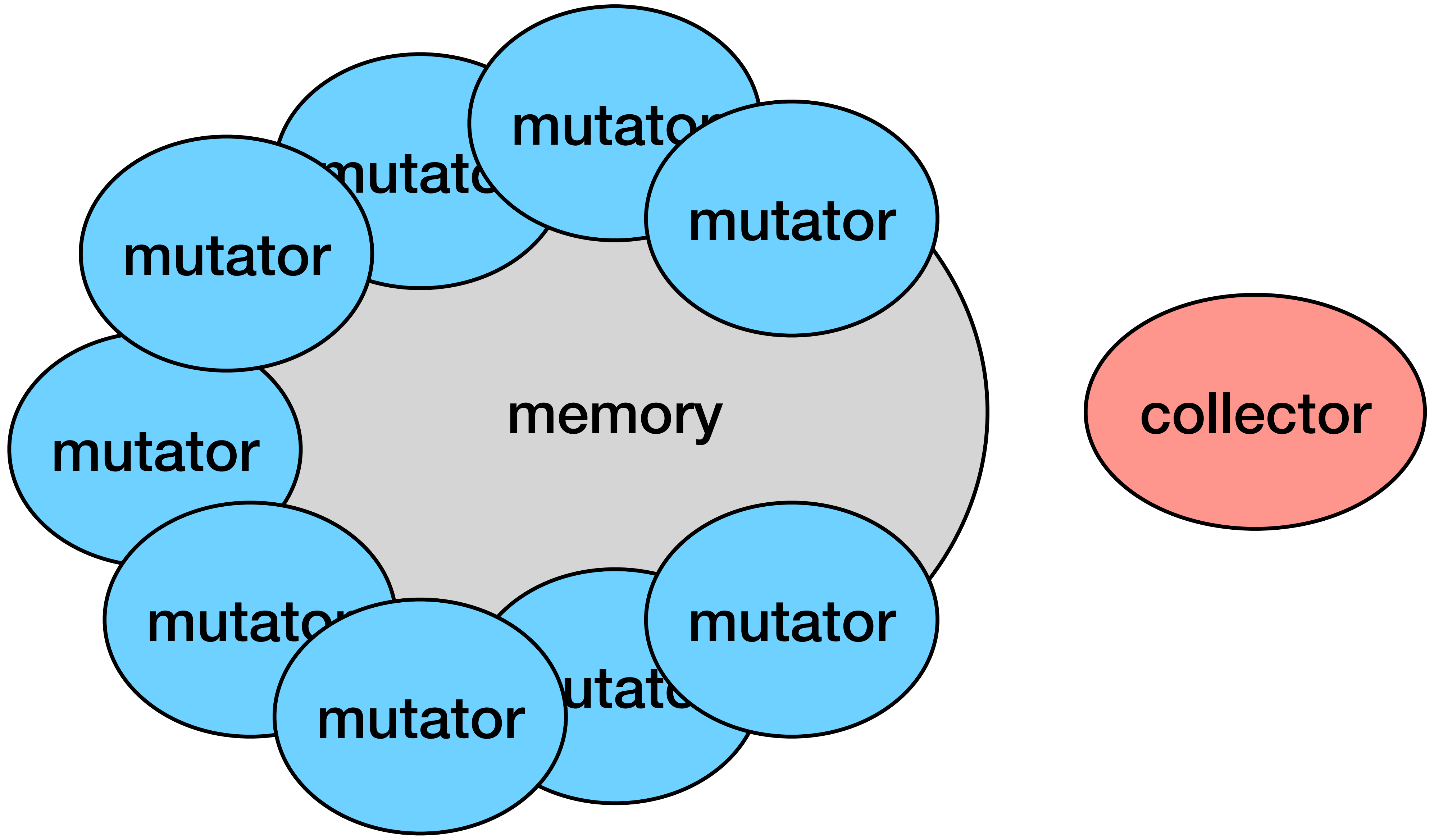
can parallel functional programming be fast and scalable ?

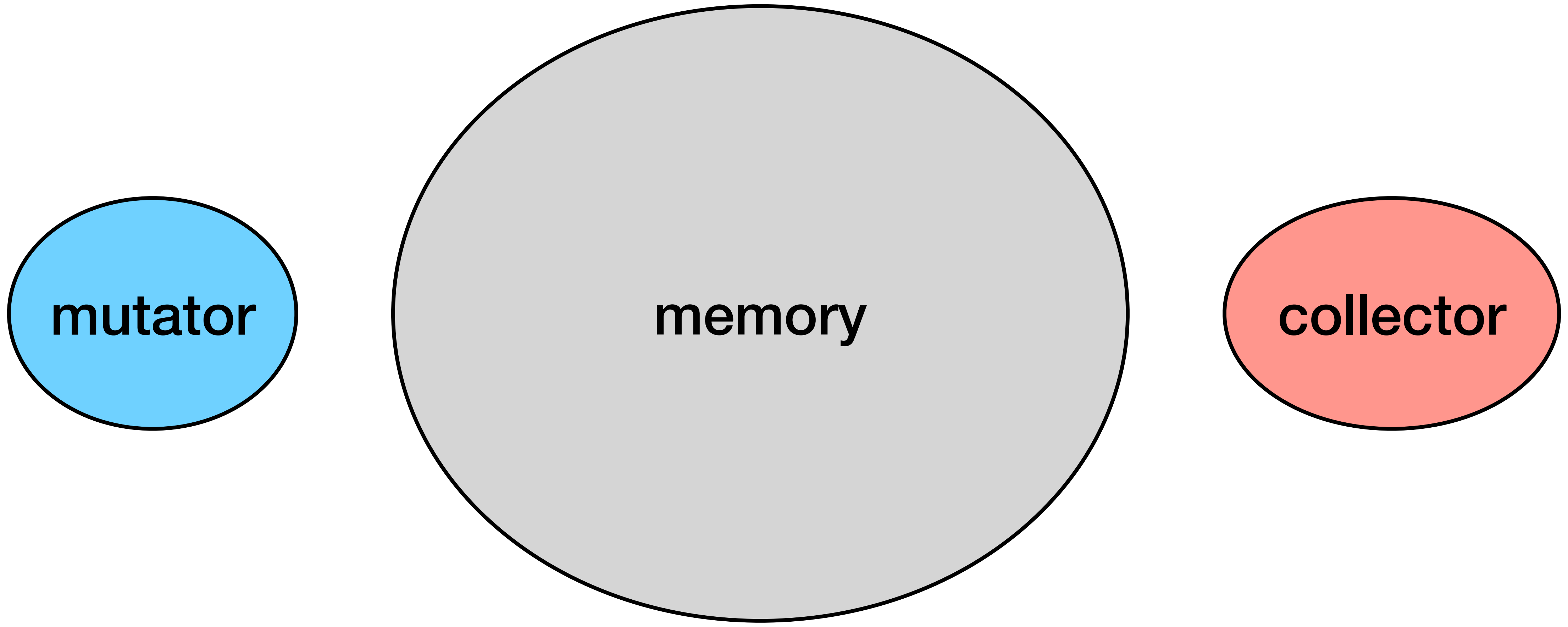
slow?



Sequential

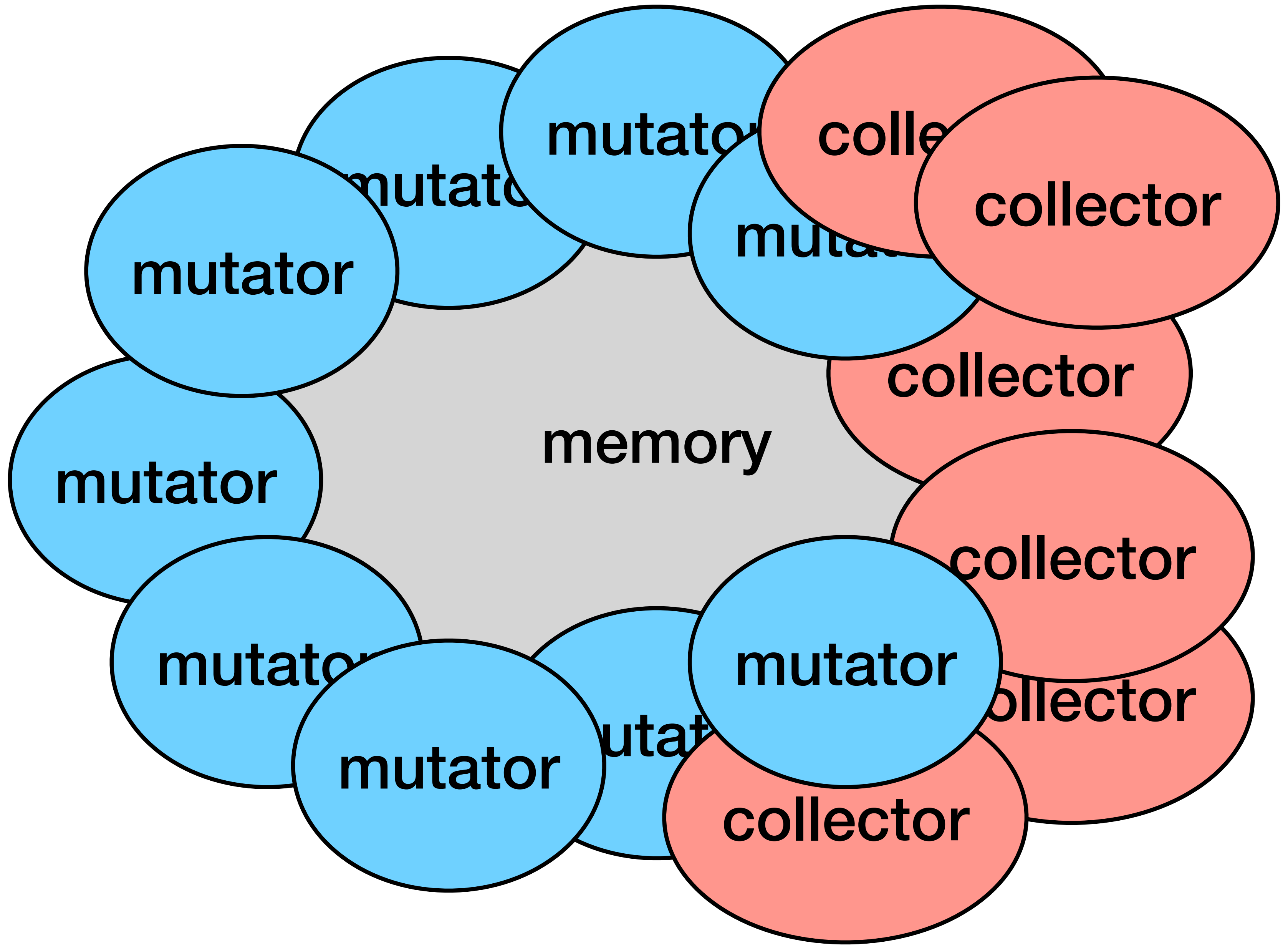
Parallel





Sequential

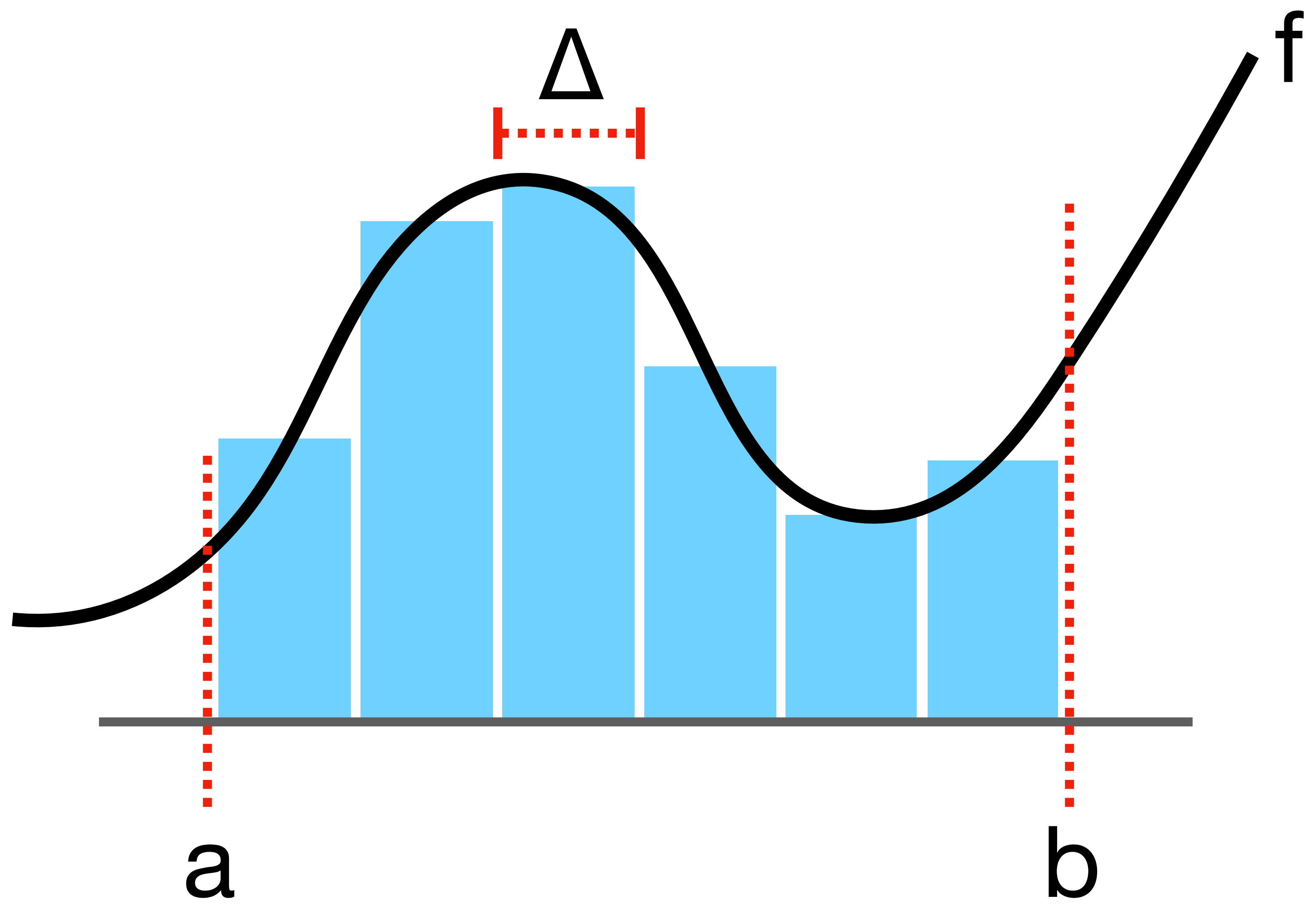
Parallel



Is there a better way?

Example: Numerical Integration

```
function integrate(f, a, b, n) {
  Δ = (b-a) / n
  heights = tabulate(n, fn i => f(a + Δ/2 + i*Δ) )
  return Δ * reduce(heights, fn (a,b) => a+b )
}
```

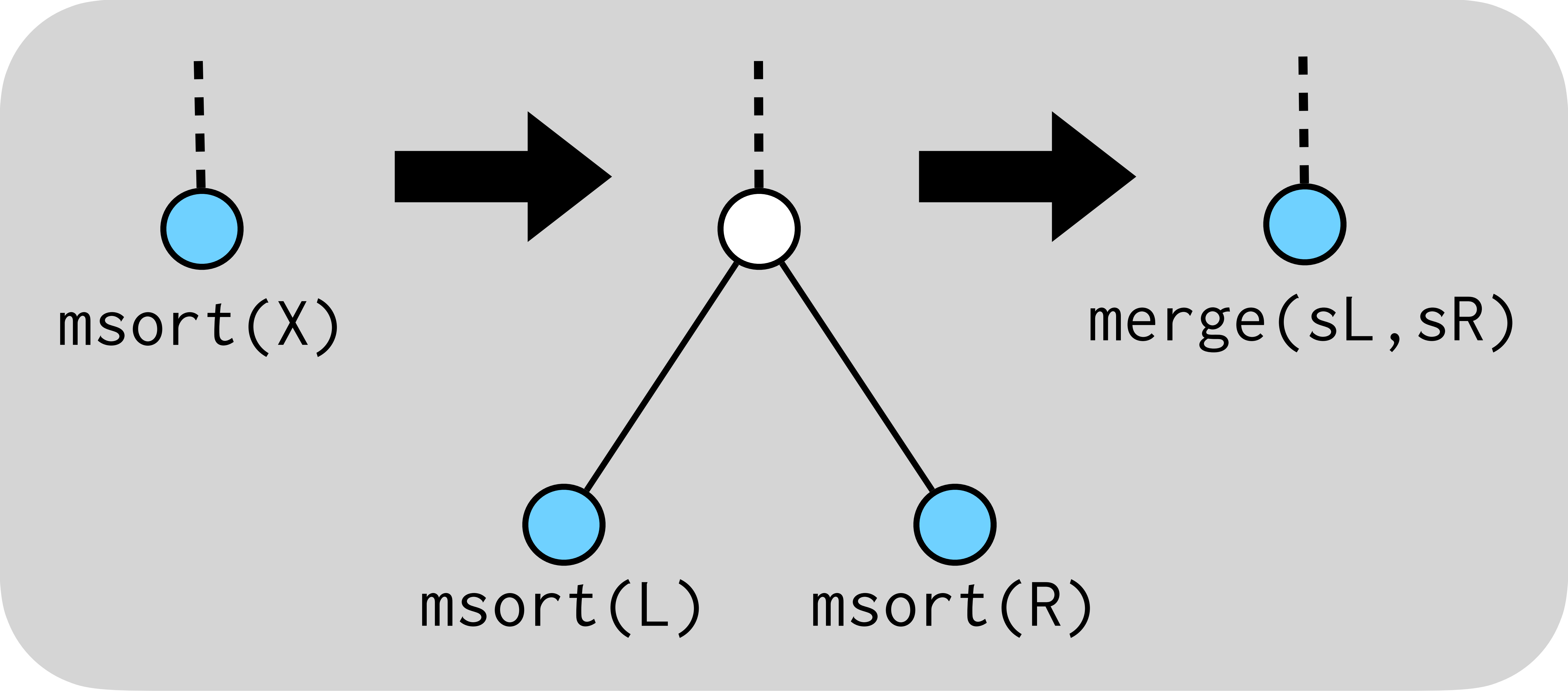


O(n) work
O(log n) span

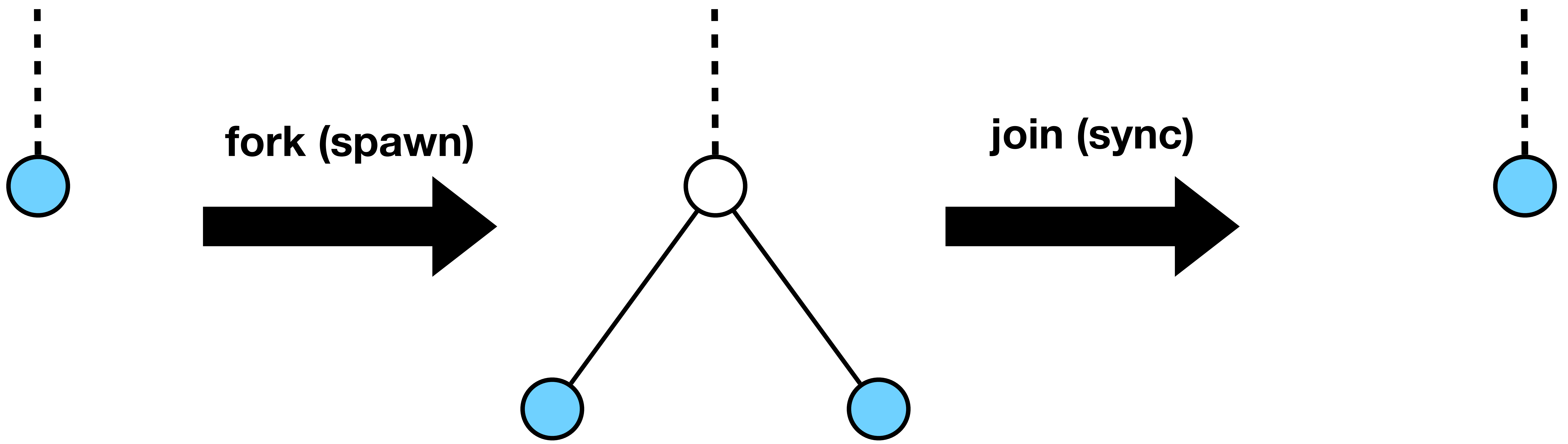
Example: Mergesort

```
function msort(X) {
  if length(X) <= 1 return X
  L, R = split(X)
  sL, sR = par(msort(L), msort(R))
  return merge(sL, sR)
}
```

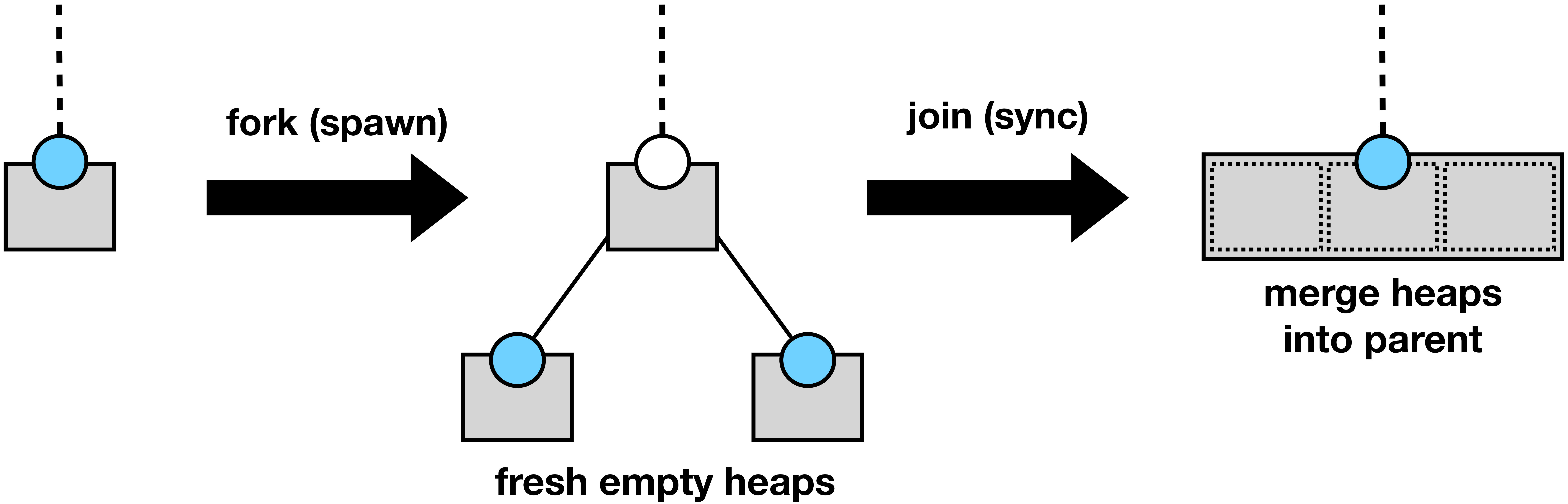
O(n log n) work
O($\log^k$ n) span



Task-Local Heaps

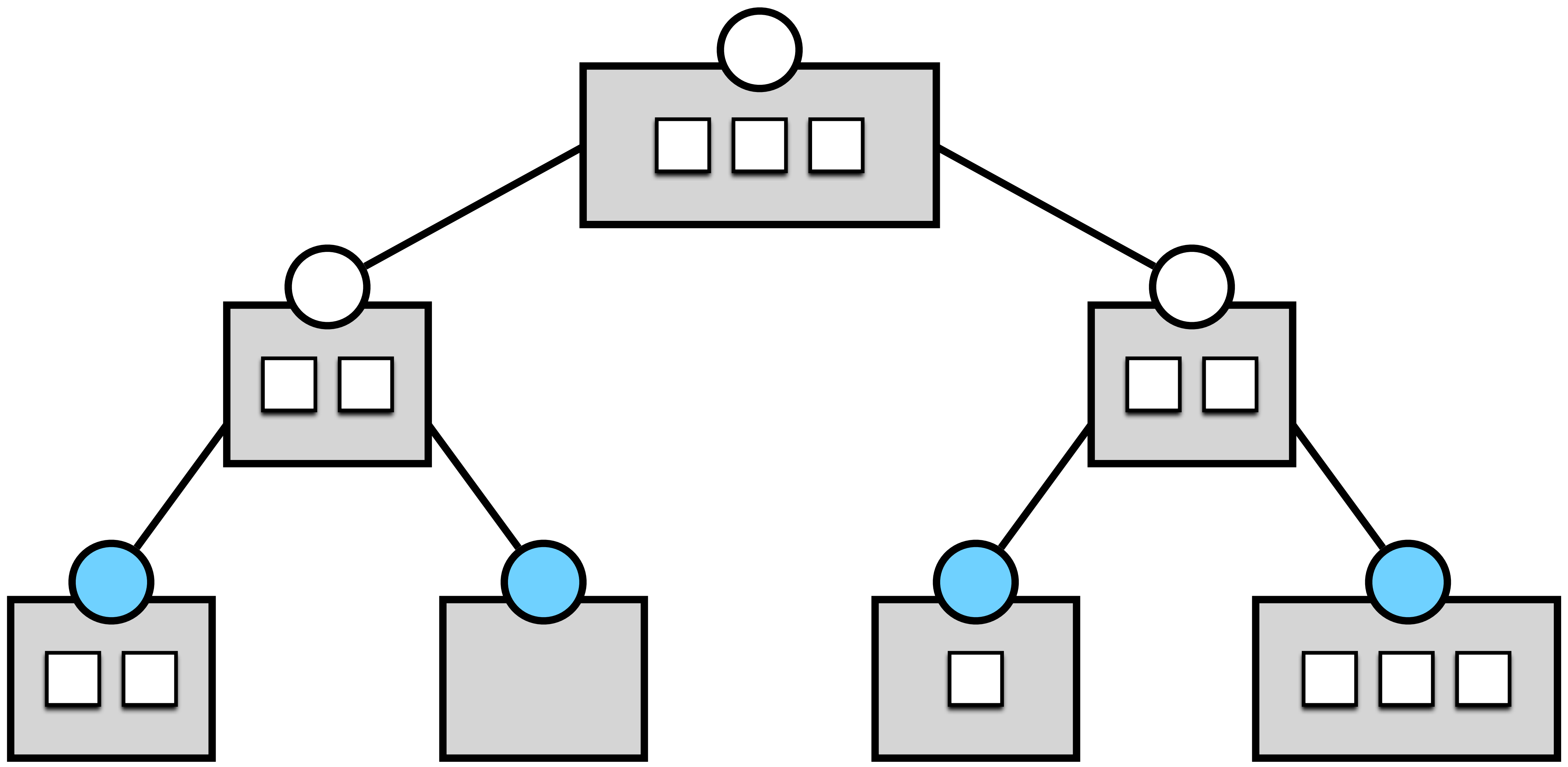


Task-Local Heaps



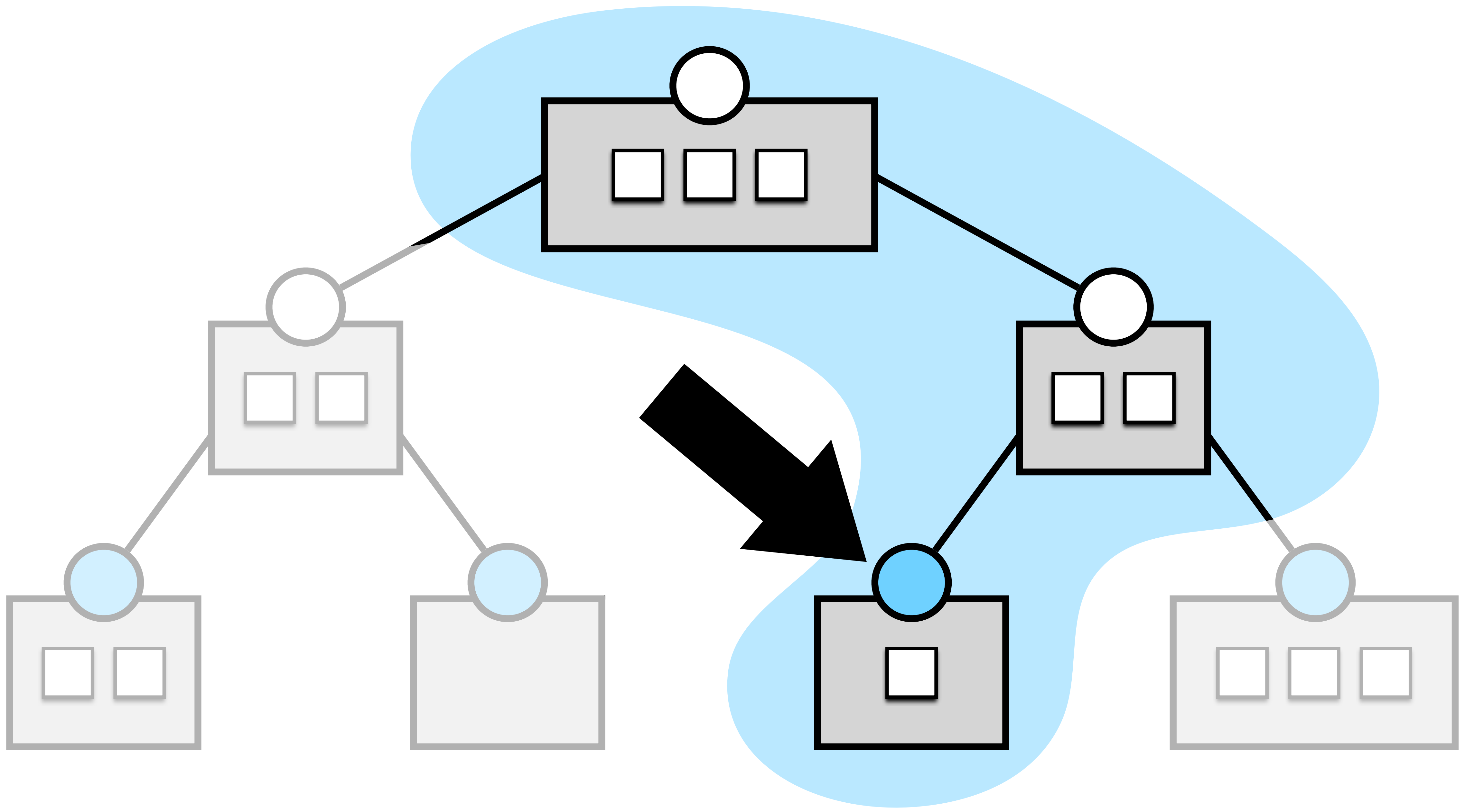
Disentanglement

definition
throughout execution, each task may only use data in **local** or **ancestor** heaps



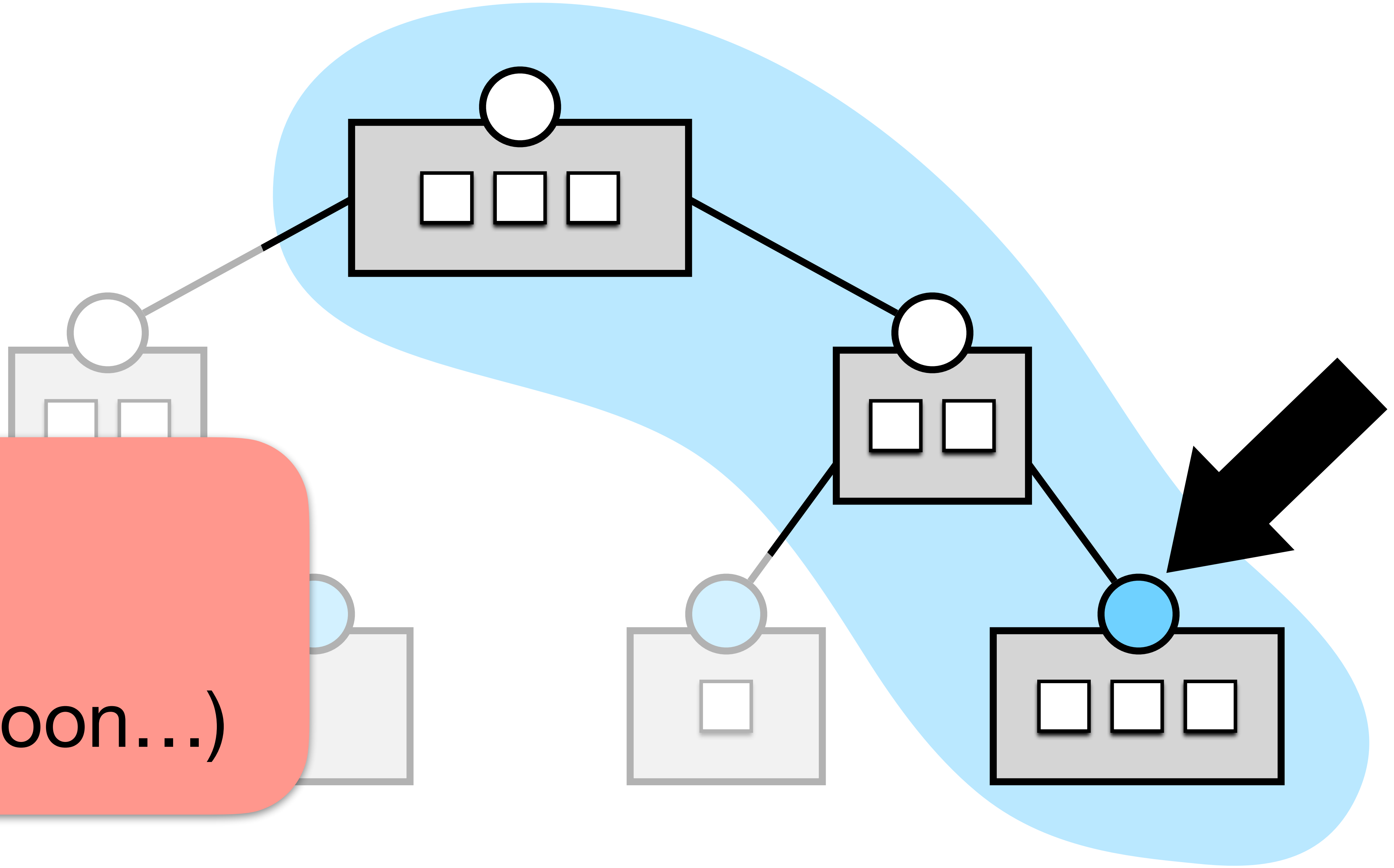
Disentanglement

definition
throughout execution, each task may only use data in **local** or **ancestor** heaps



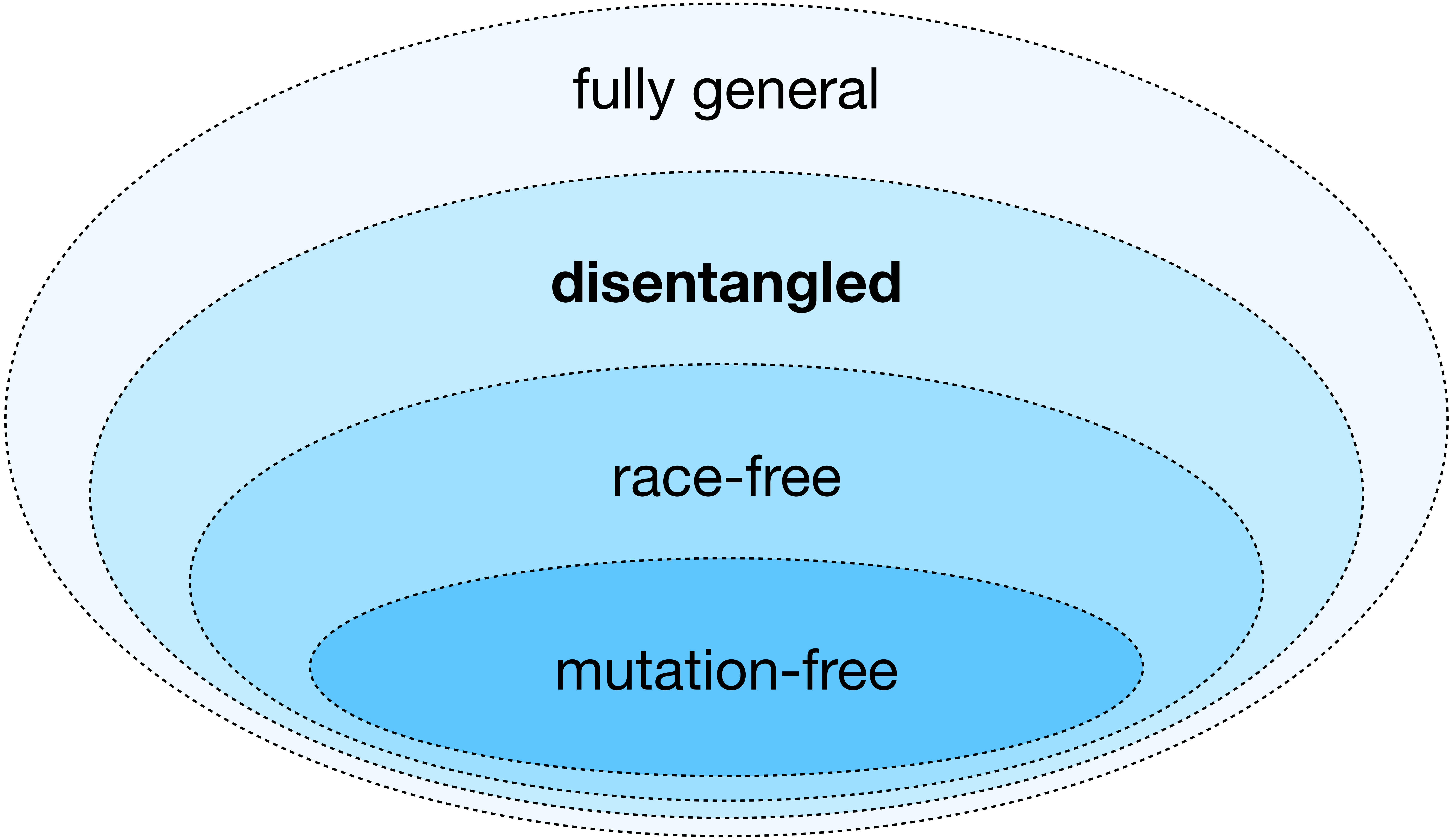
Disentanglement

definition
throughout execution, each task may only use data in **local** or **ancestor** heaps



Why do we care?
Hint: parallel GC
(will get back to this soon…)

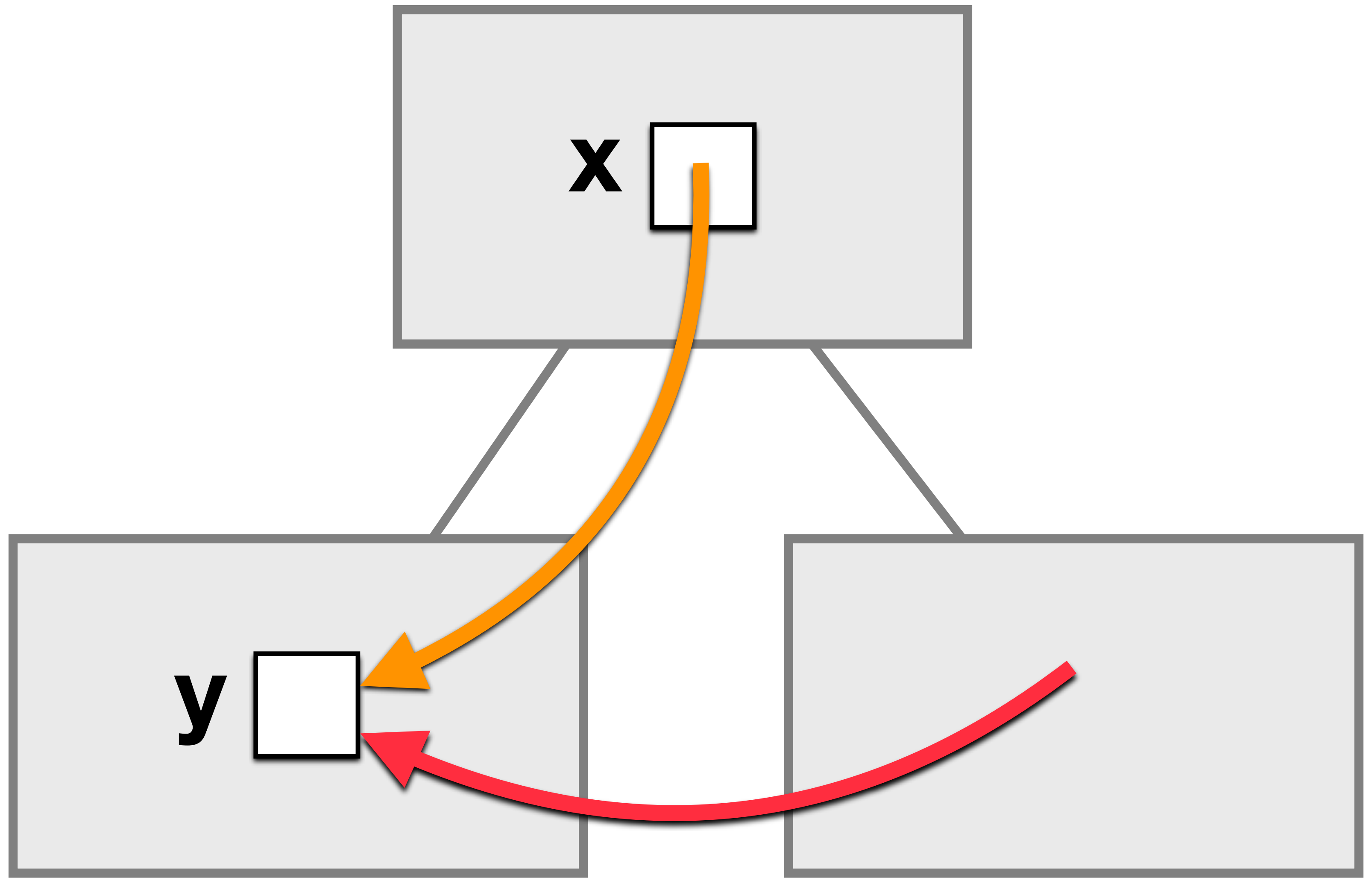
What programs are disentangled?



theorem [Westrick et al., POPL 20]
all race-free programs are disentangled

Intuition

- if entangled, must have read **down-pointer**
- down-pointer must have been created by concurrent write
- so, program has read/write race




```
y = malloc()        |  ...
*x = y              |  ...
...                 |  z = *x
```

Proof Sketch

- **single-step invariant:**
  if location *X* accessible without a race, then *neighbors(X)* are in root-to-leaf path
- carry invariant through race-free execution

Disentanglement in the Wild

Ligra

- BFS
- betweenness centrality
- Bellman-Ford
- *k*-Core
- Page Rank
- maximal independent set
- eccentricity estimation

PBBS

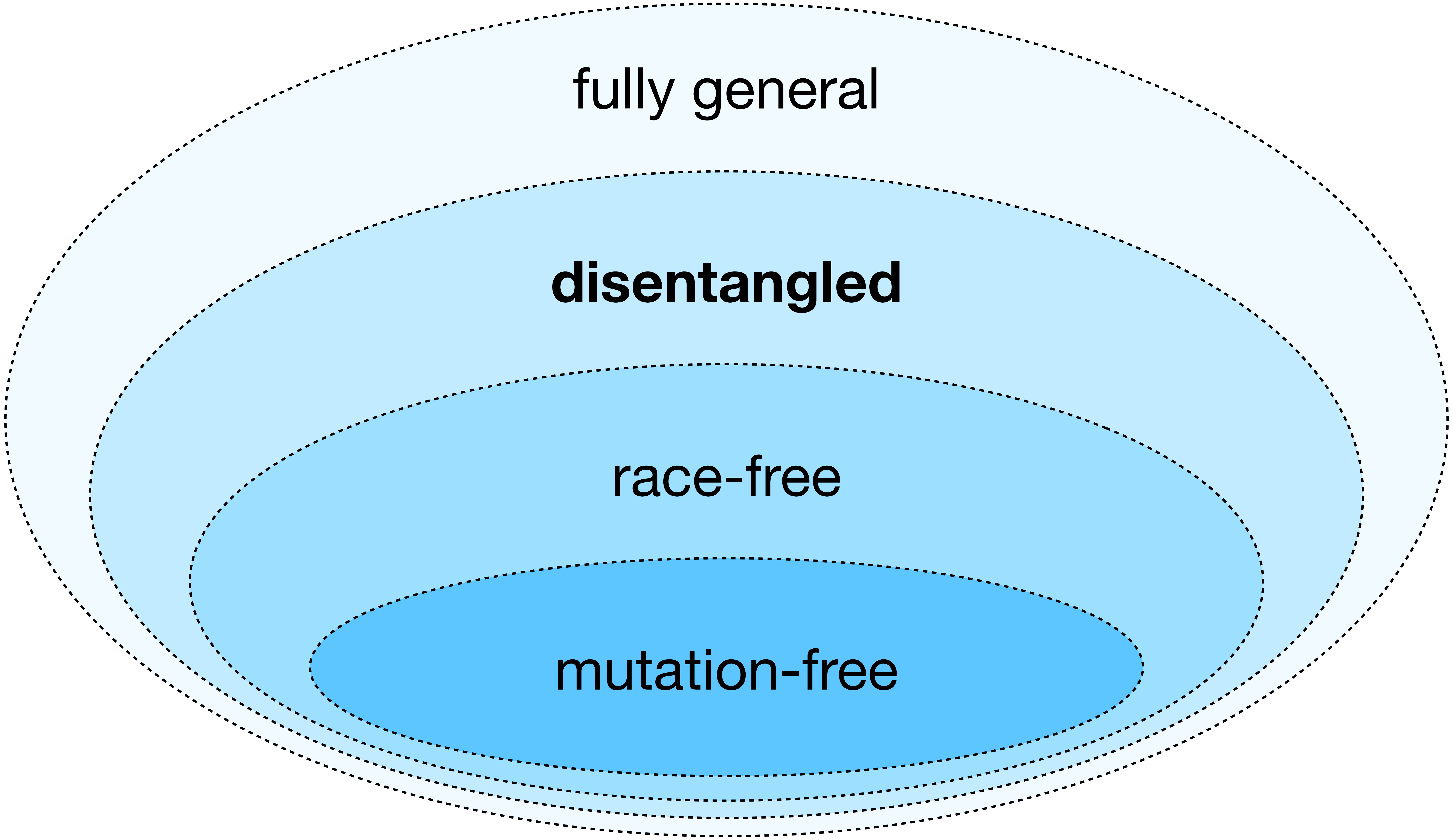
- quickhull
- deduplication
- sorting
- minimum spanning forest
- suffix array
- Barnes-Hut
- nearest neighbors
- ray casting

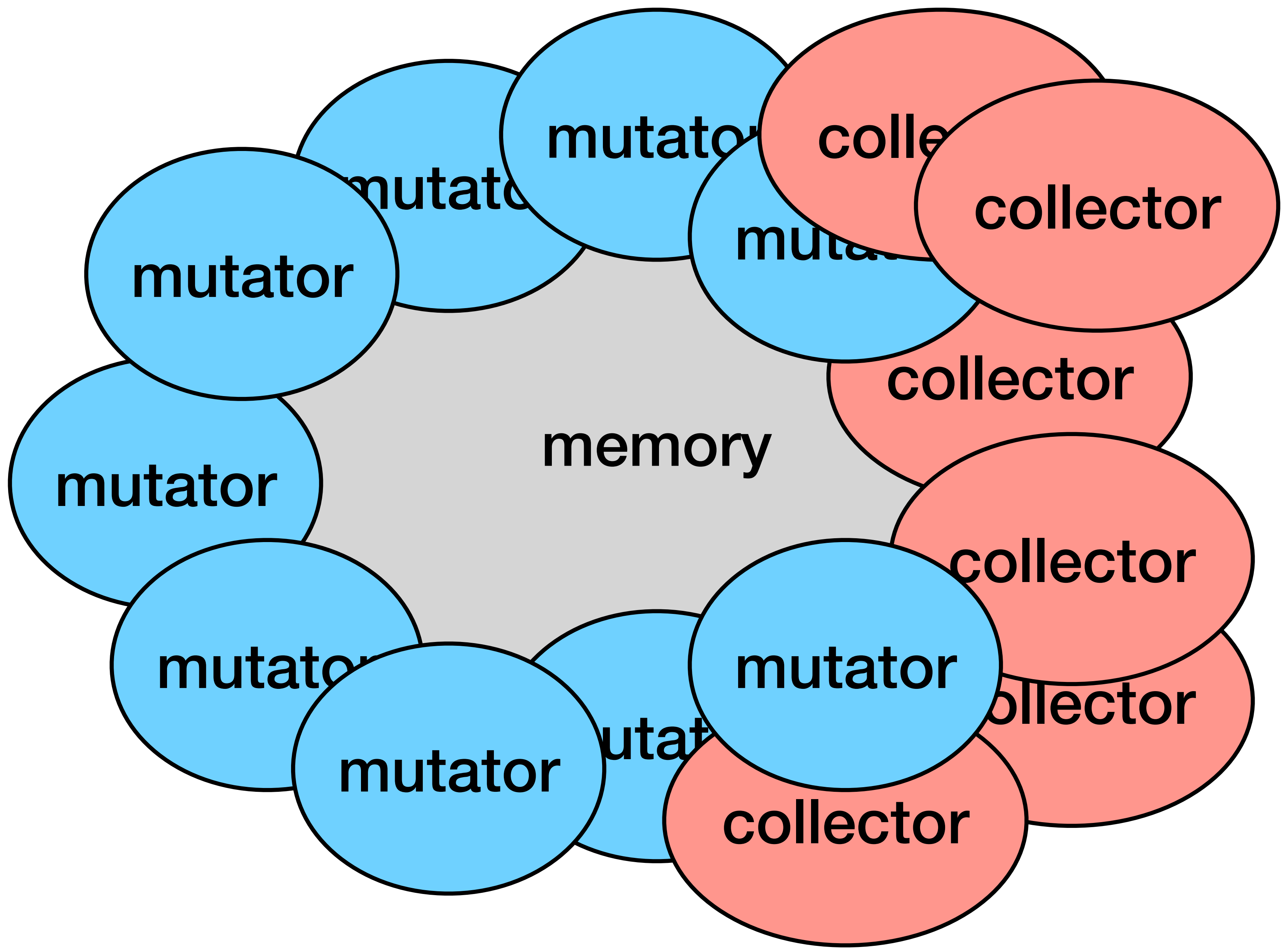
all disentangled

(and likely others too)

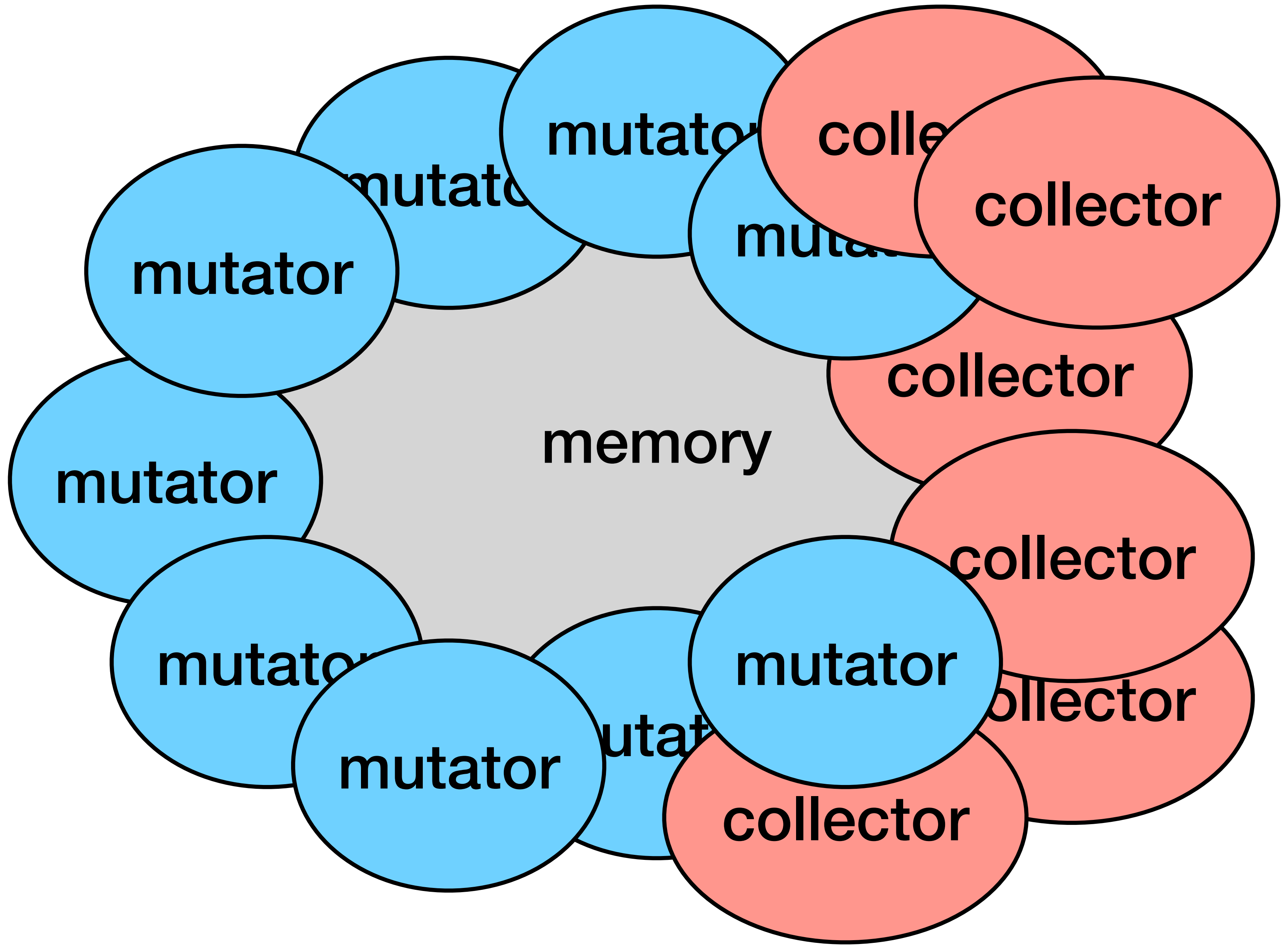
many benign races

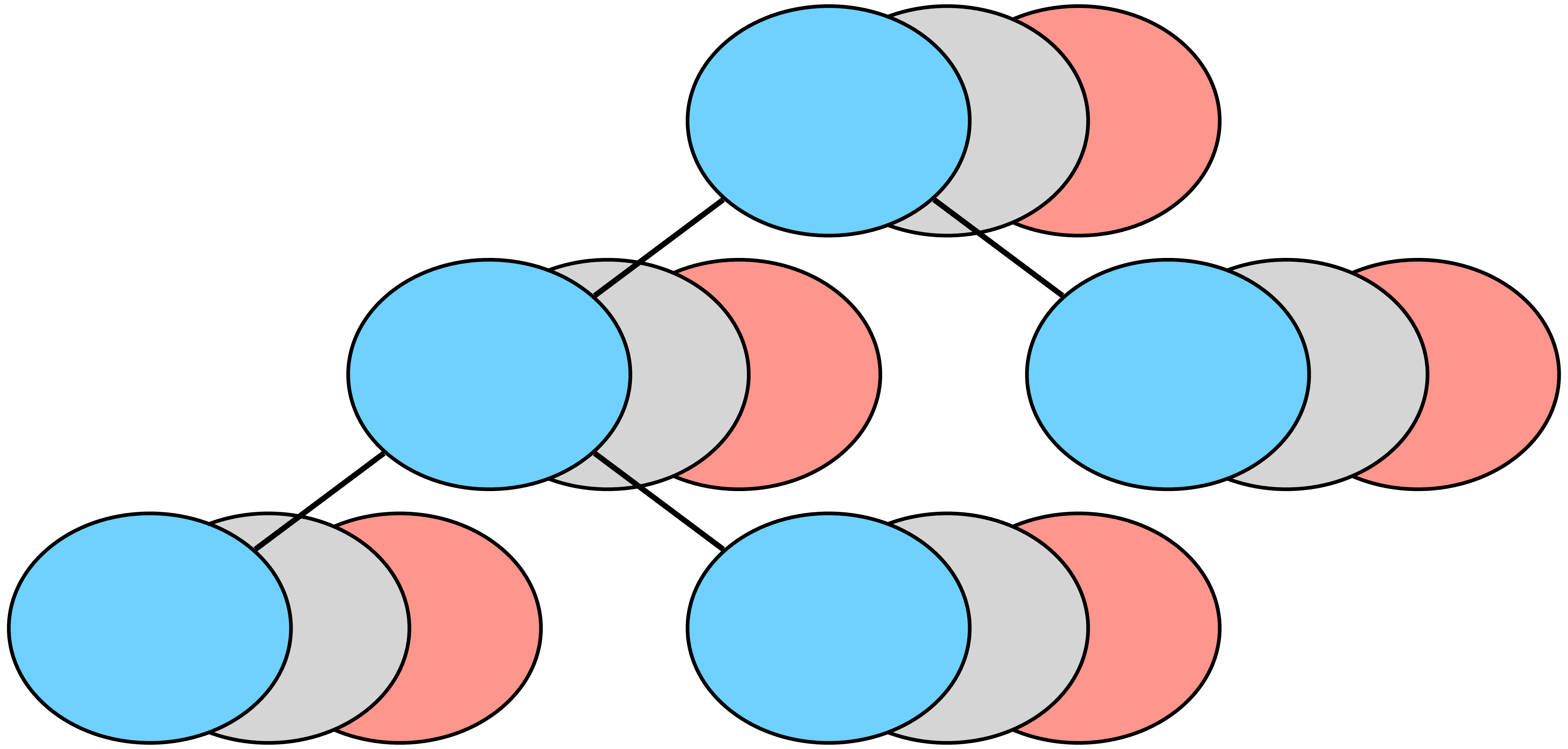
What programs are disentangled?





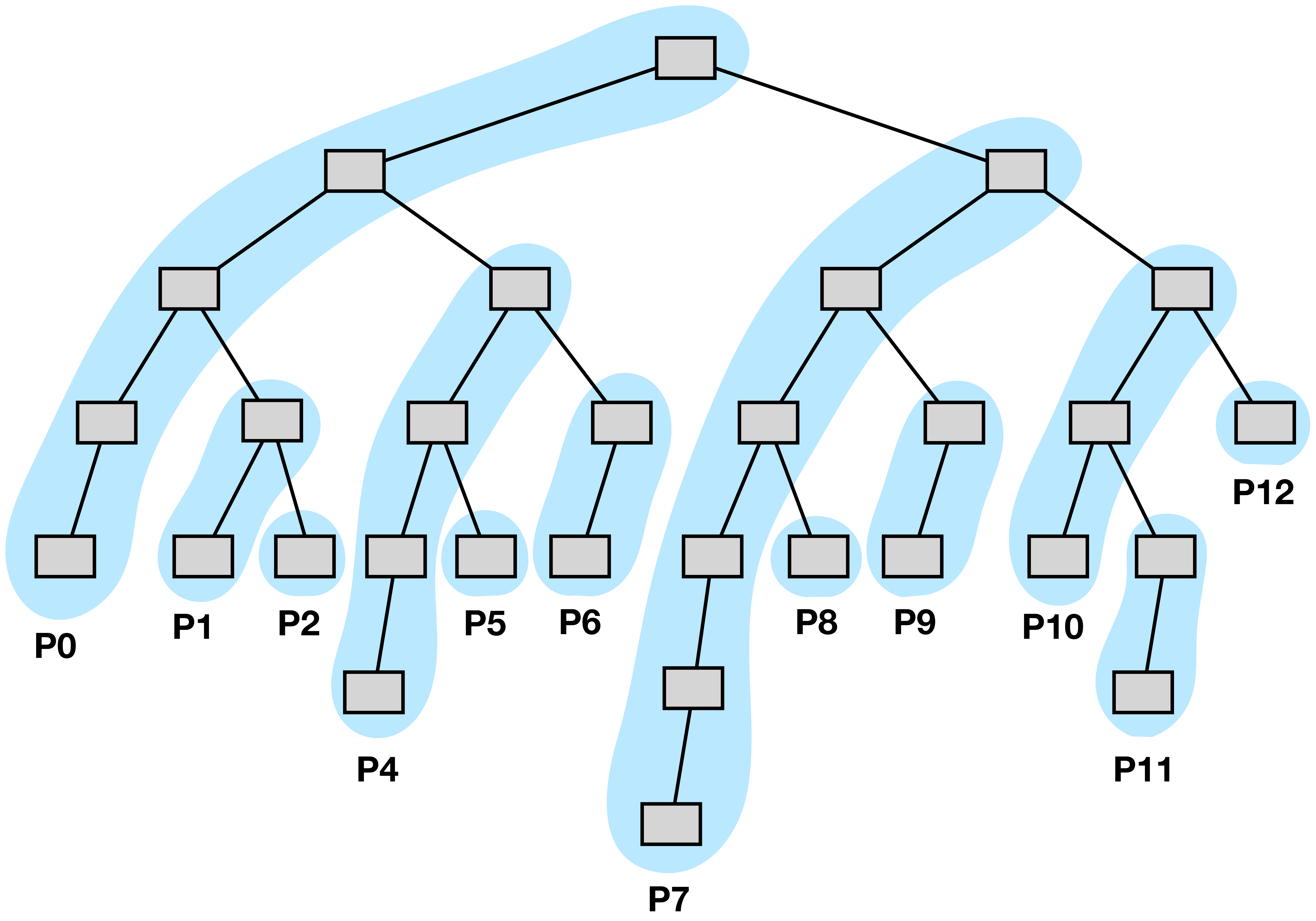
Is there a better way?


mutator
mutator
memory
mutator
mutator
collector
collector
collector
collector



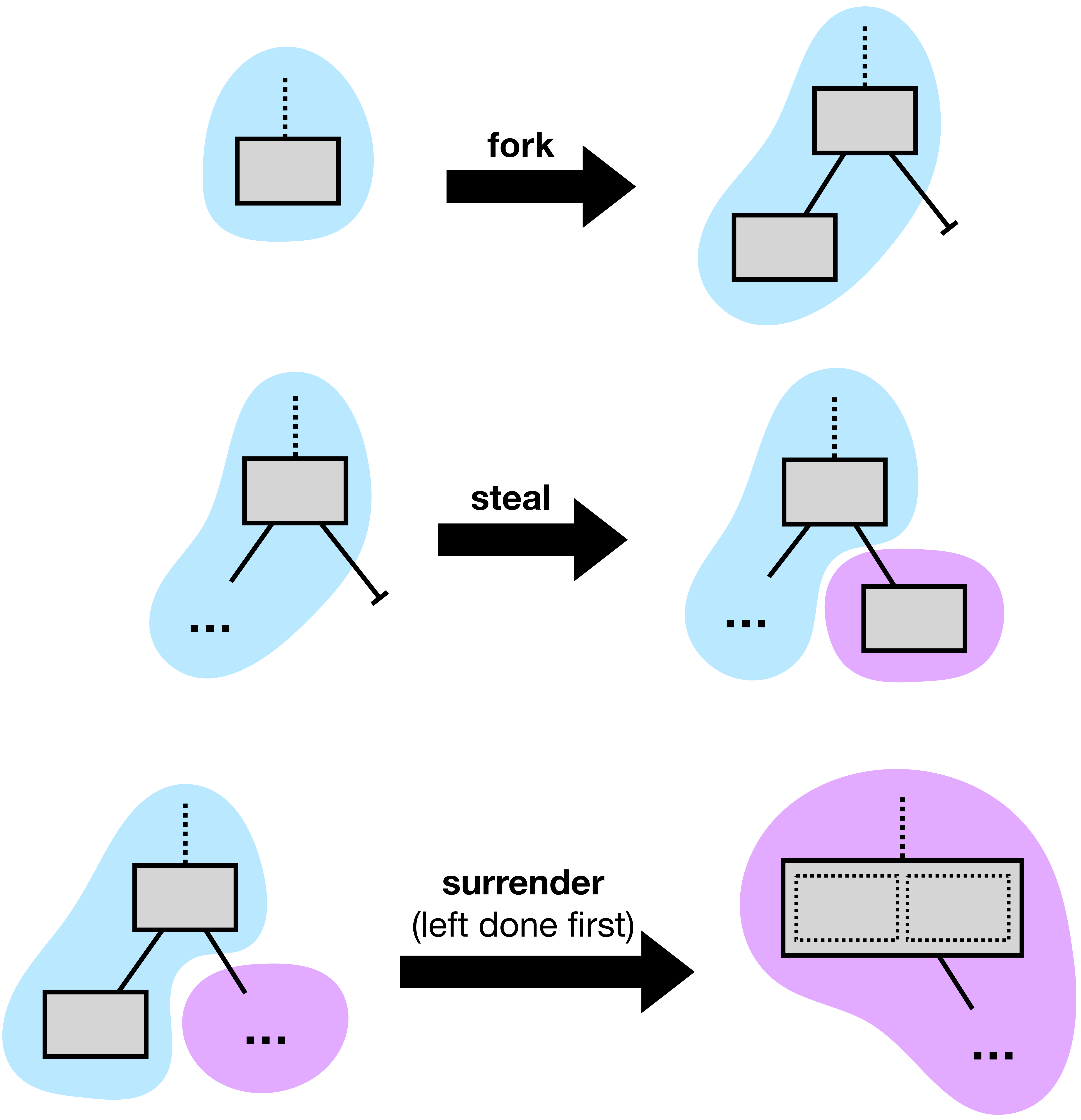
Heap Scheduling

- goal: assign heaps to processors
- each processor manages its own memory



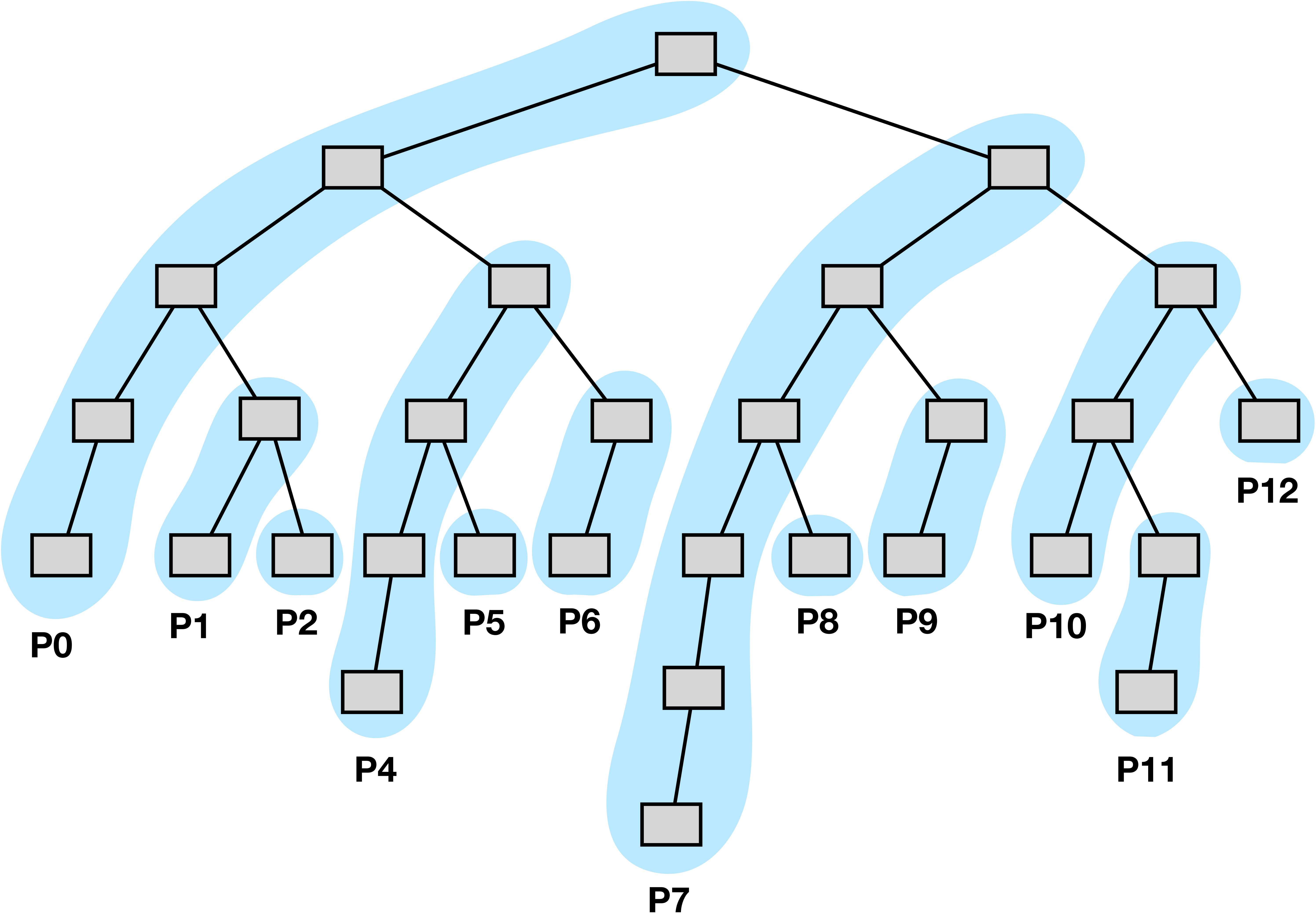
Heap Scheduling

- goal: assign heaps to processors
- each processor manages its own memory
- **integrate closely with thread scheduling** (work-stealing)
  - **fork**: new heap on left, assign to *same* proc
  - **steal:** new heap on right, assign to *new* proc
  - **surrender**: at join, give heaps to sibling

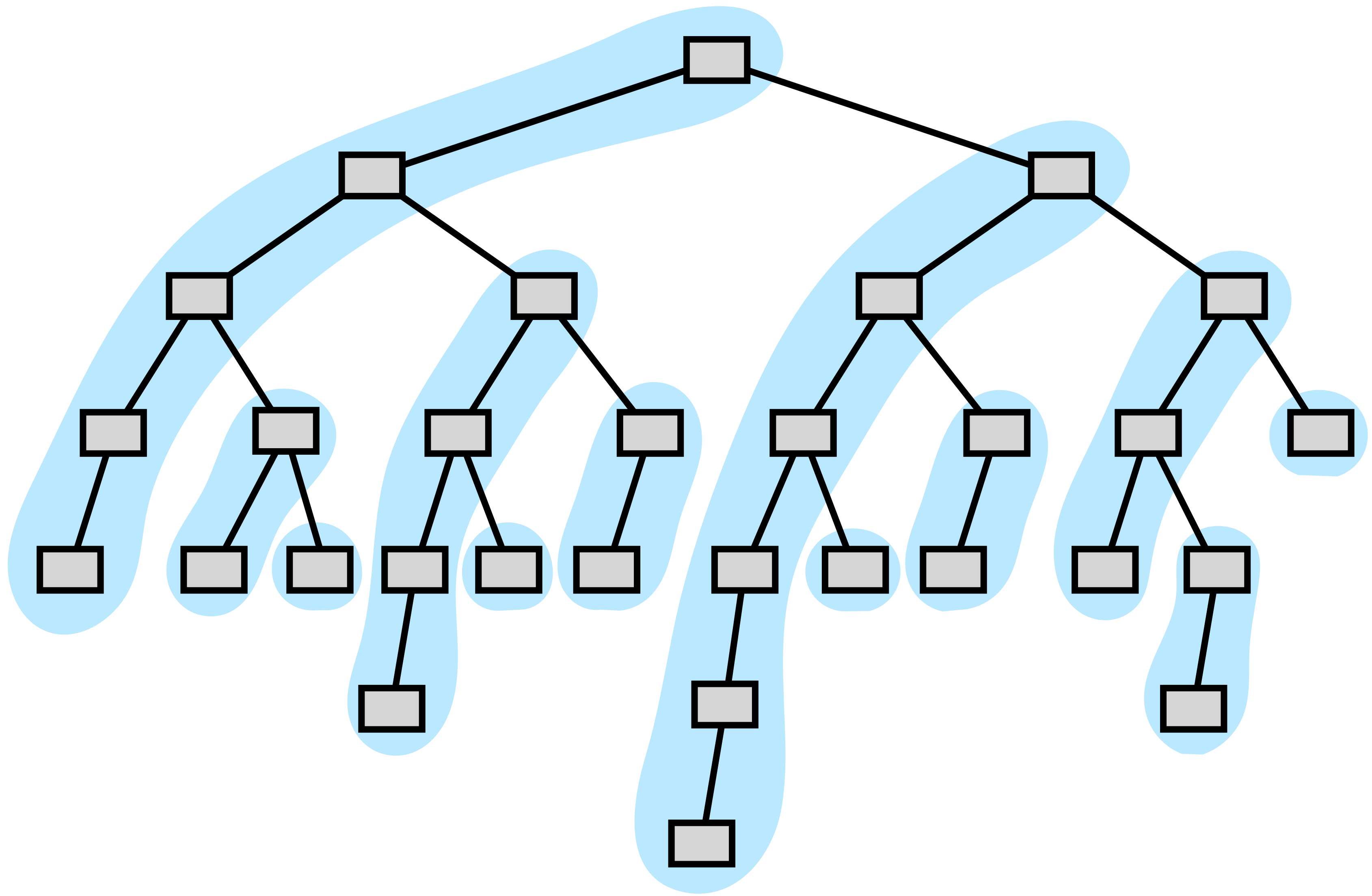


Heap Scheduling

- goal: assign heaps to processors
- each processor manages its own memory
- **integrate closely with thread scheduling** (work-stealing)
    - **fork**: new heap on left, assign to *same* proc
    - **steal:** new heap on right, assign to *new* proc
    - **surrender**: at join, give heaps to sibling



Collection Policy



algorithm
- each processor ***p*** has local counter L_p
- when cumulative size of ***p***'s heaps exceeds $k{\cdot}L_p$:
  - processor ***p*** performs GC on its heaps
  - set L_p to amount of memory that survives

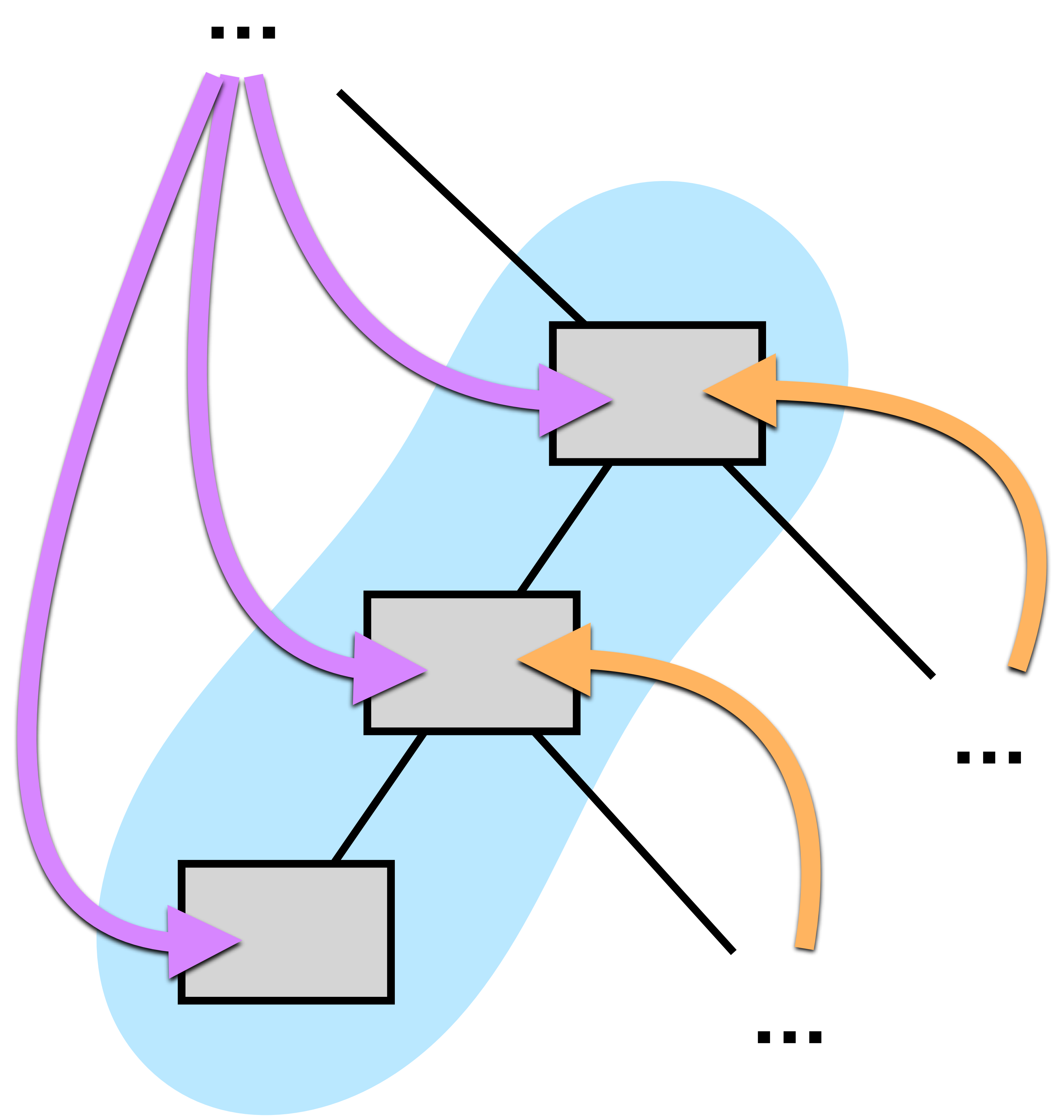
theorem [Arora et al., POPL 21]
a race-free program with work ***W*** and sequential space R^* requires $O(P{\cdot}R^*)$ space and $O(W + P{\cdot}R^*)$ work, including costs of memory management

Key ideas:

- after surrender, heaps resemble sequential execution
  - left-before-right, or right-before-left?
  - "unordered reachable space" R^* allows for both
- local counters L_p cannot exceed R^*

Disentangled Garbage Collection

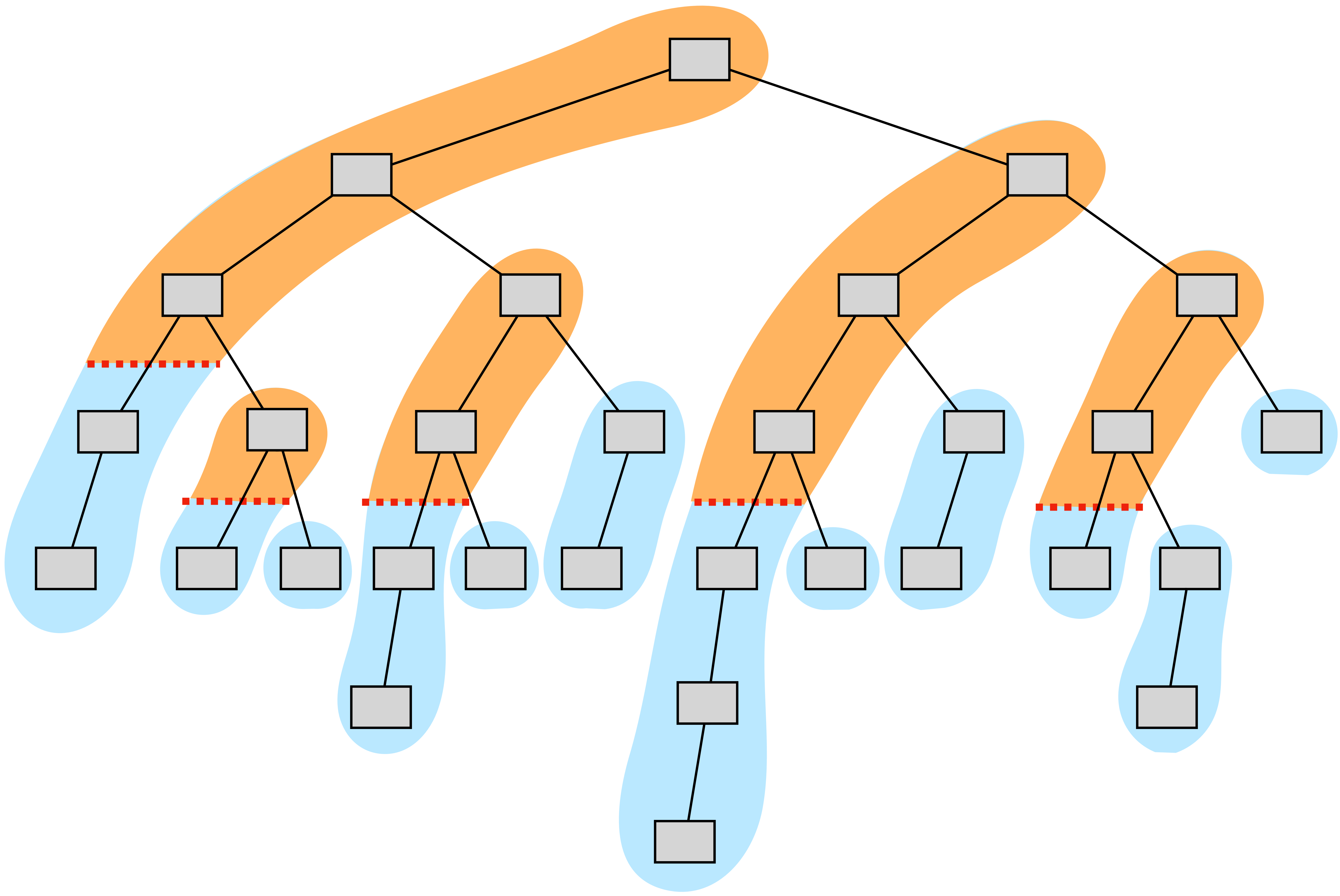
- every pointer points ***up*** or ***down***
  - ***disentanglement: no cross-pointers***
- leaves are active tasks with GC roots (think of these as up-pointers)
- write-barrier remembers down-pointers
- **snapshot-at-fork** summarizes up-pointers from stolen children
  - closure of right-side forked task is good enough (doesn't violate local ***R**** bound!)
  - write-barrier preserves reachability



Disentangled Garbage Collection

internal
- has to be concurrent GC
- non-moving mark-sweep

local
- no concurrency
- compactifying (copying) GC



MaPLe

- based on MLton compiler for Standard ML
- full Standard ML language, extended with fork-join library

```
val par: (unit -> ‘a) * (unit -> ‘b) -> ‘a * ‘b
```

- used by 500+ students at Carnegie Mellon University each year

github.com/MPLLang/mpl

Sorting Shootout

	T_1	T_{72}
C++ std::sort	8.8	–
Cilk samplesort	7.9	0.16
Cilk mergesort	12.7	0.24
MPL (Ours) mergesort	18.8	0.37
Go samplesort	27.2	0.52
Java mergesort	11.0	0.63
Haskell/C mergesort	10.6	1.3

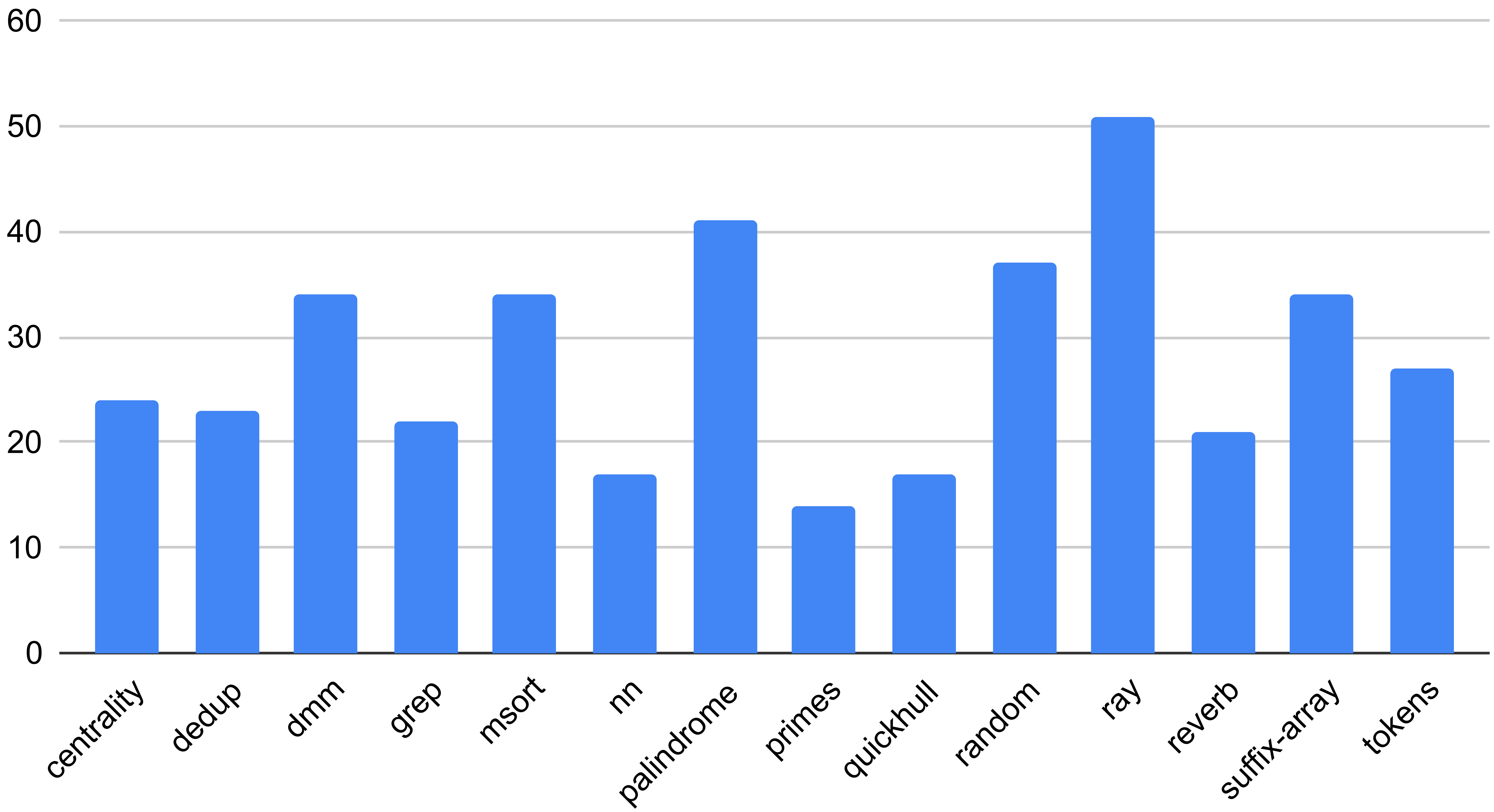
~24x speedup over C++ std::sort

2nd fastest, only behind C++/Cilk

40% faster than Go

70% faster than Java

Speedups

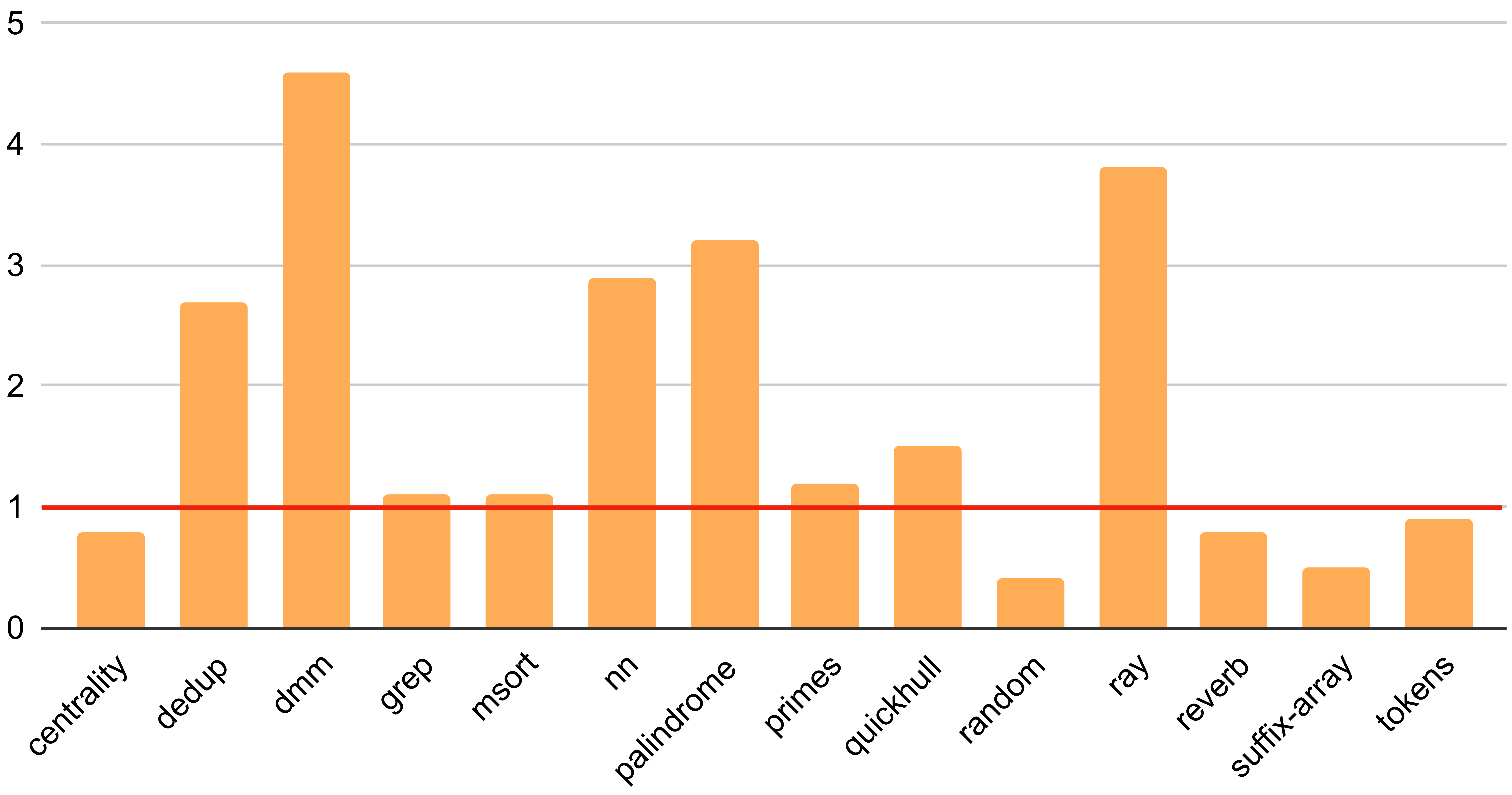


PBBS-style benchmarks

70 procs

relative to MLton

Space Overheads

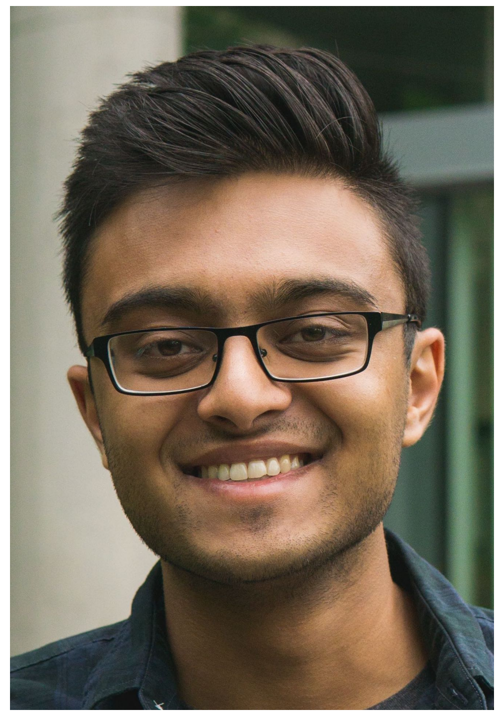


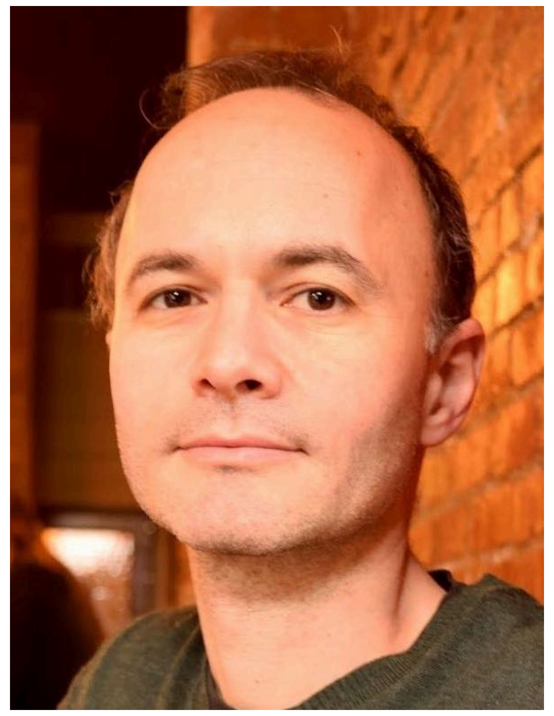
PBBS-style benchmarks

70 procs

relative to MLton

Thanks!





github.com/MPLLang/mpl