Efficient and Scalable Parallel Functional Programming Through Disentanglement

Sam Westrick
`samwestrick.com`
@shwestrick

March 2022

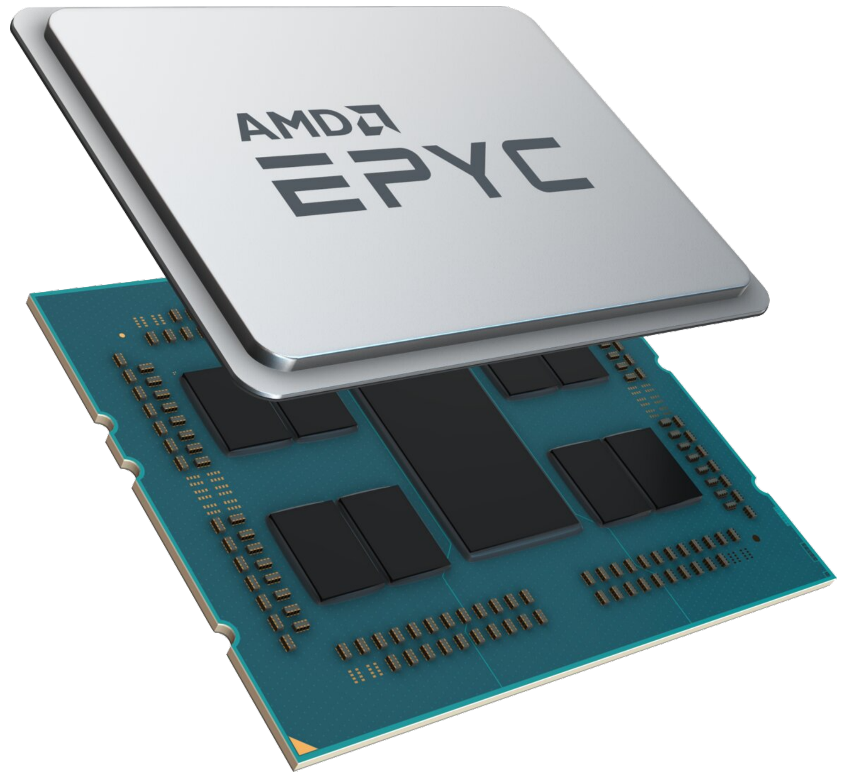
Parallel Hardware Today

AMD Ryzen Threadripper: 16 cores

Apple A14: 12 cores

Apple S4: 2 cores

nVidia GeForce 3090: 10496 (CUDA) cores



AMD Epyc: 64 cores

4x Intel Xeon E7: 72 cores

Parallel Programming

imperative

mutability (in-place updates)
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 (in-place updates)
manual memory management
race conditions

immutability
automatic memory management
deterministic by default

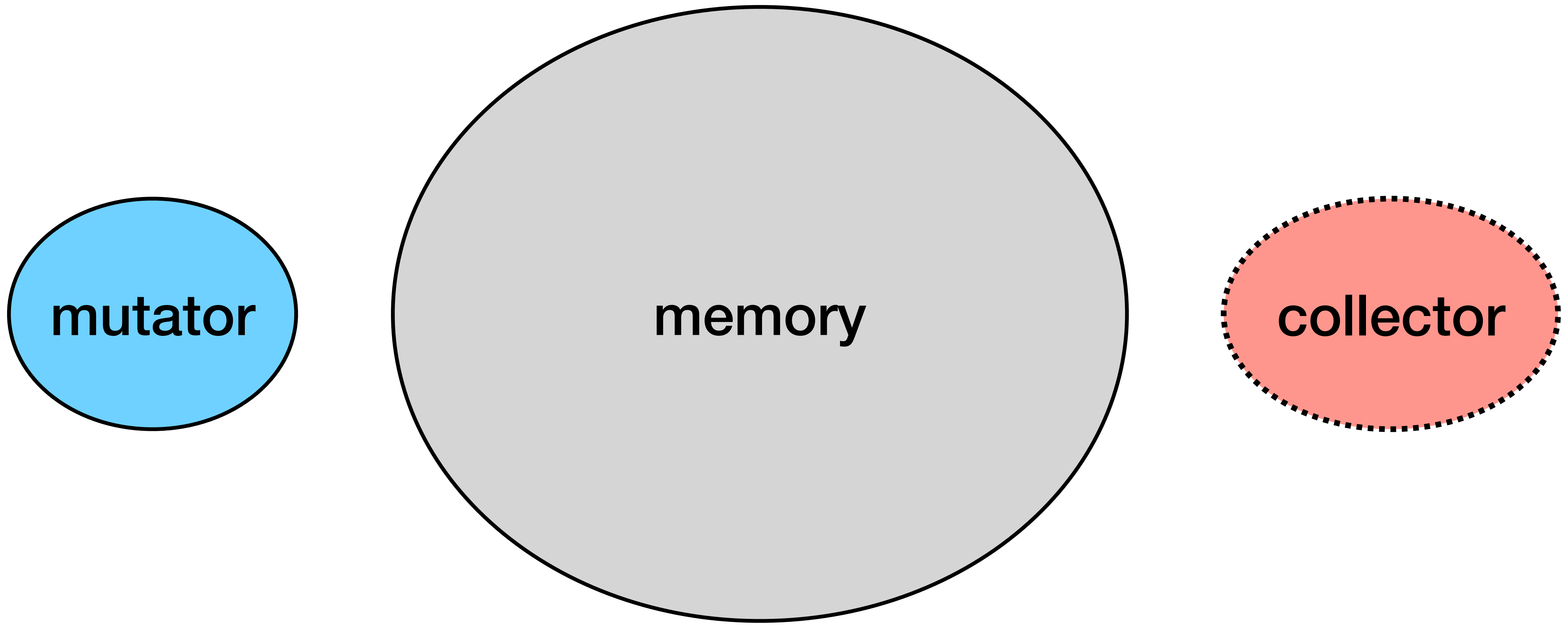
functional

high rate of allocation
heavy reliance on GC

fast

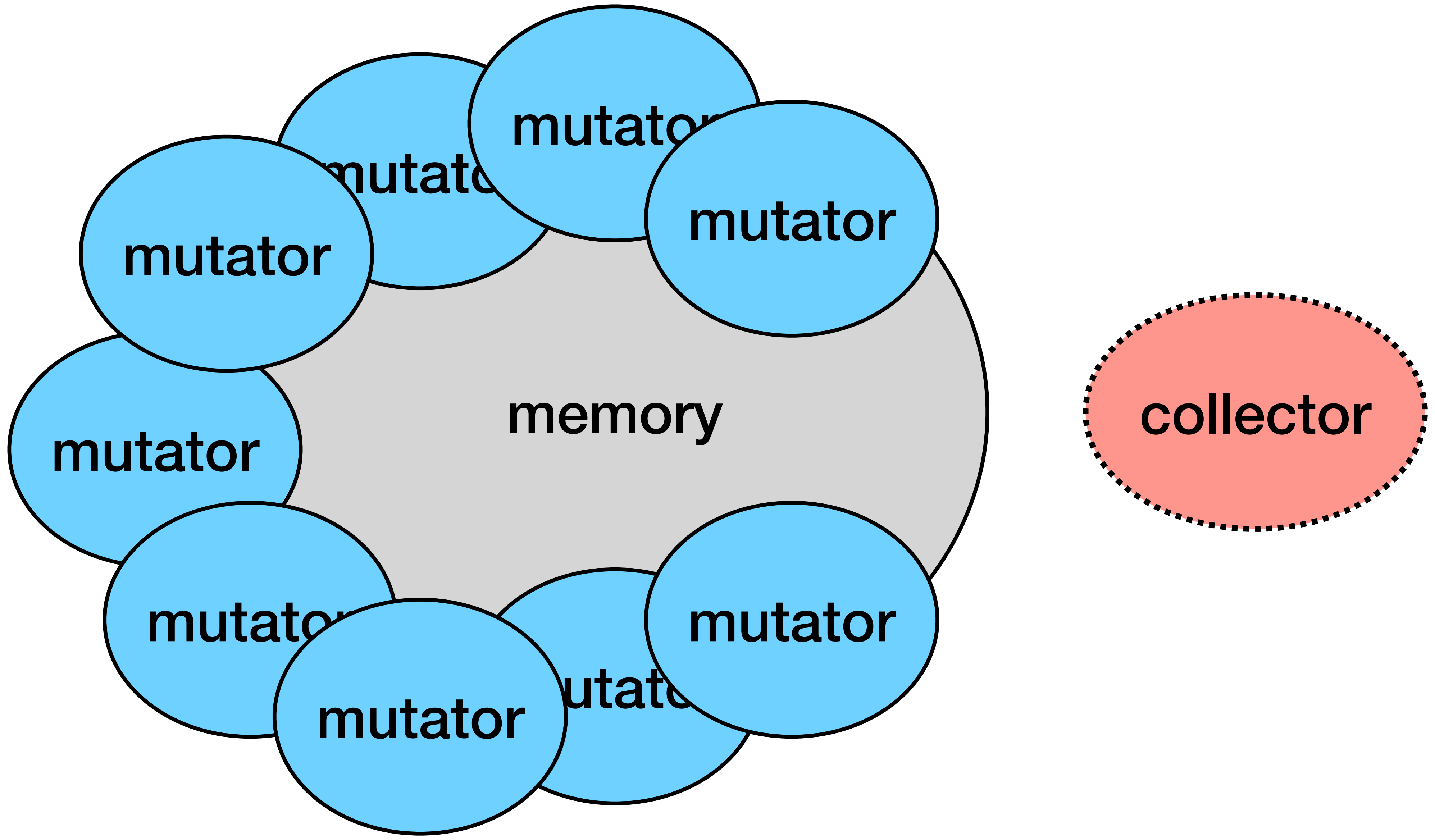
can parallel functional programming be fast and scalable?

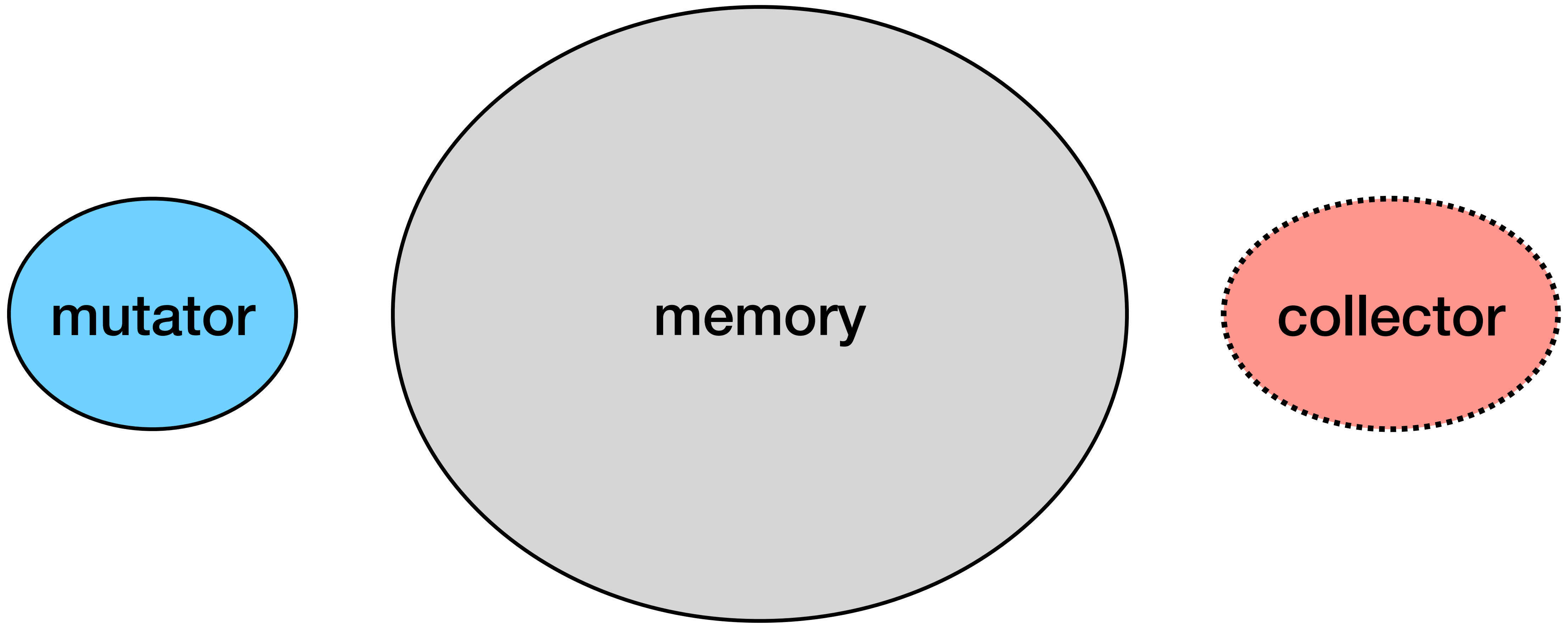
slow?



Sequential

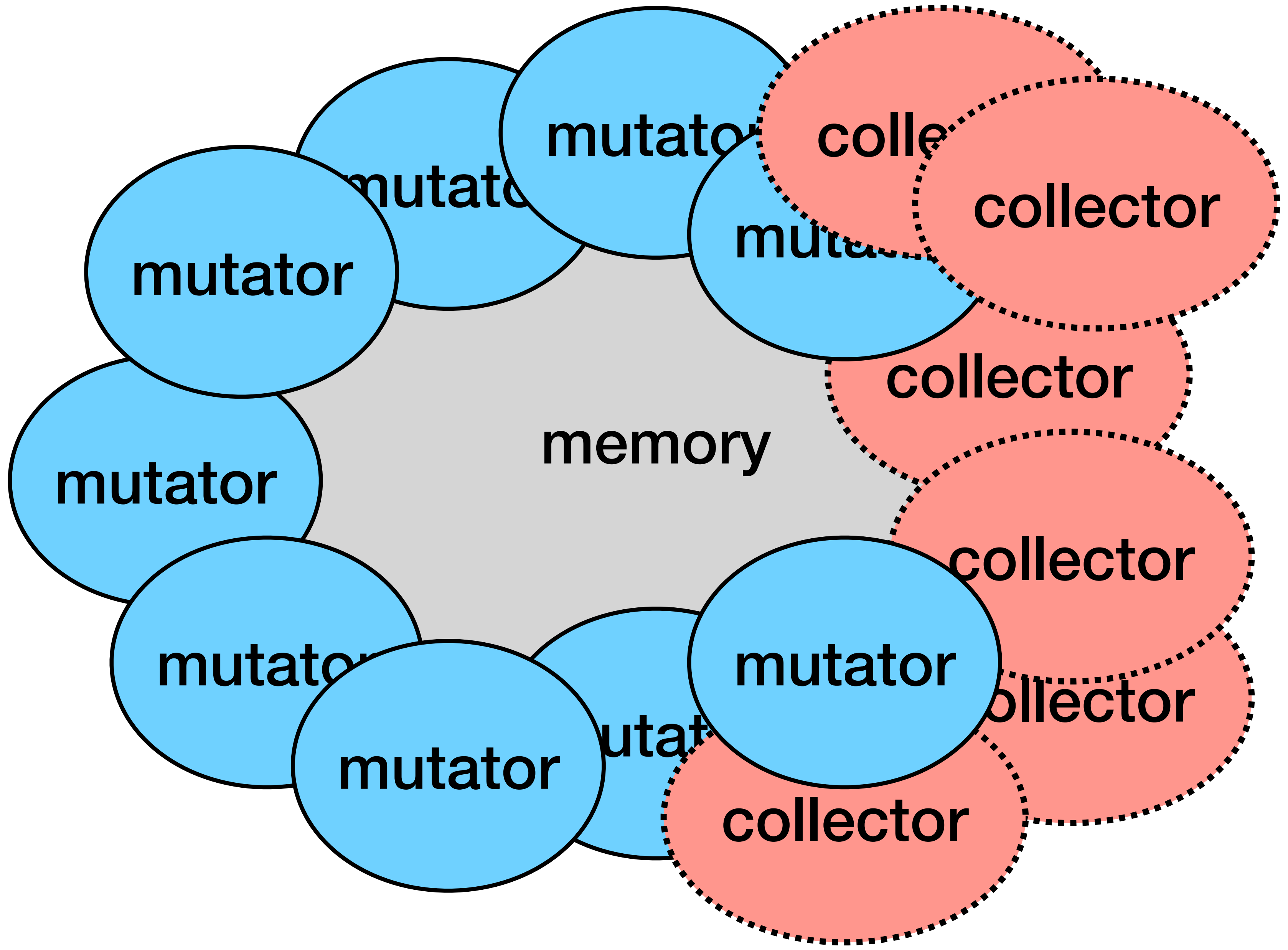
Parallel





Sequential

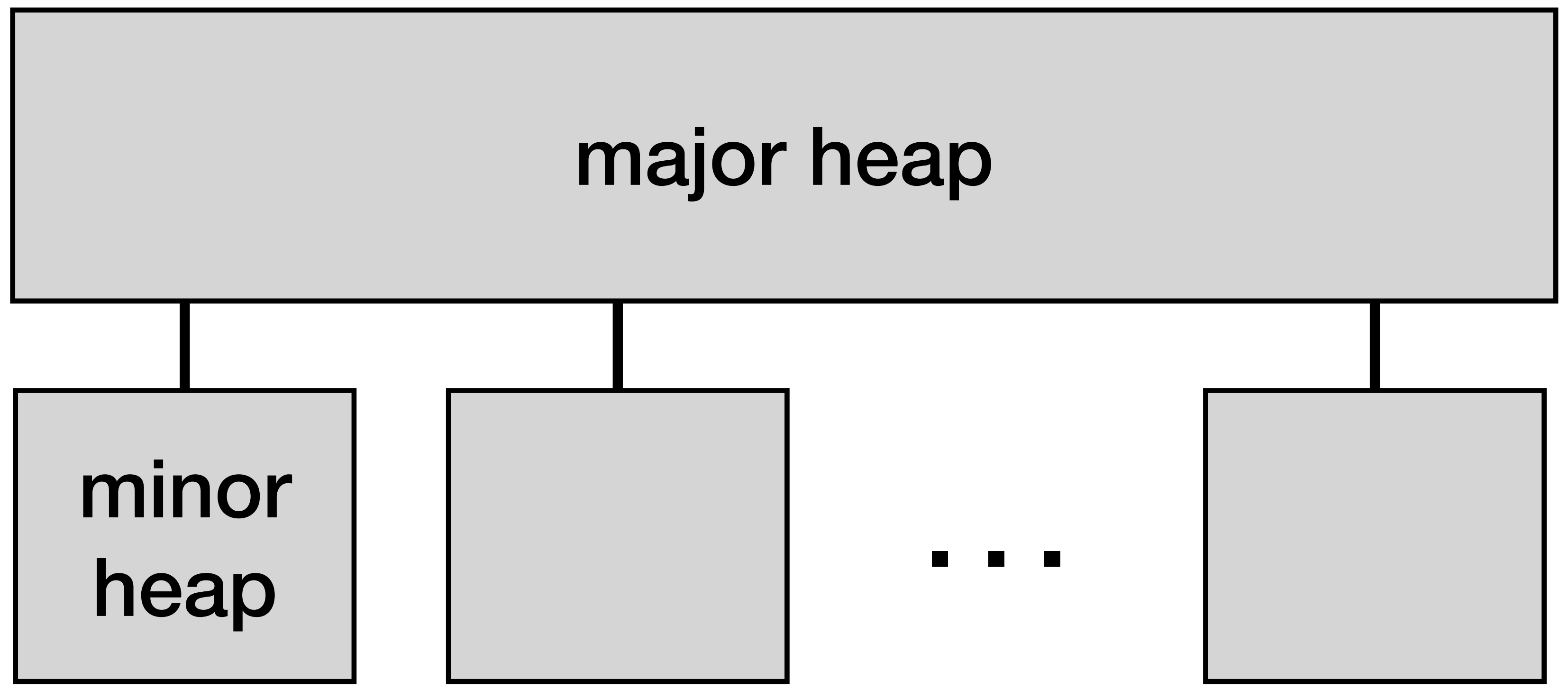
Parallel

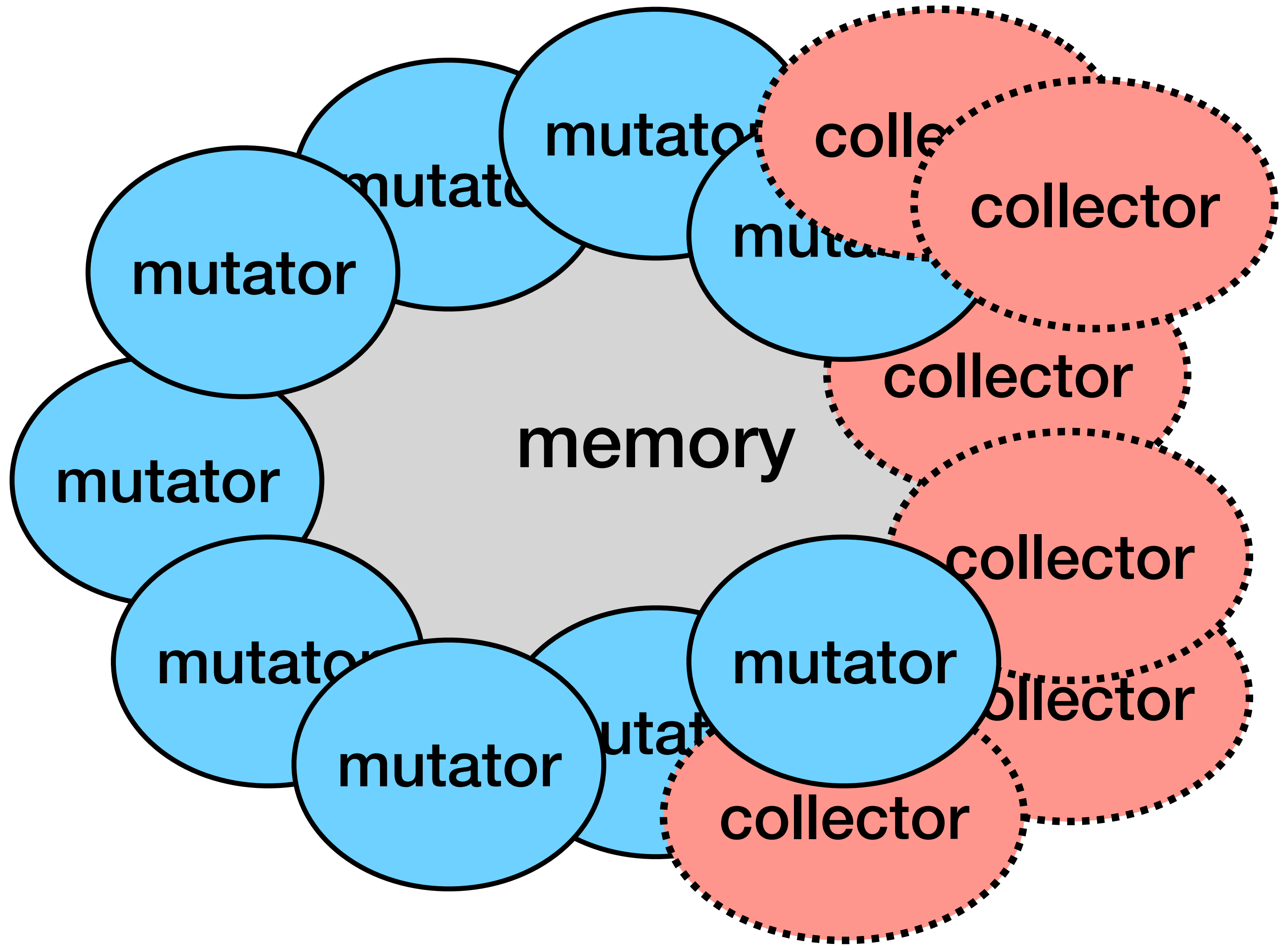


Is there a better way?

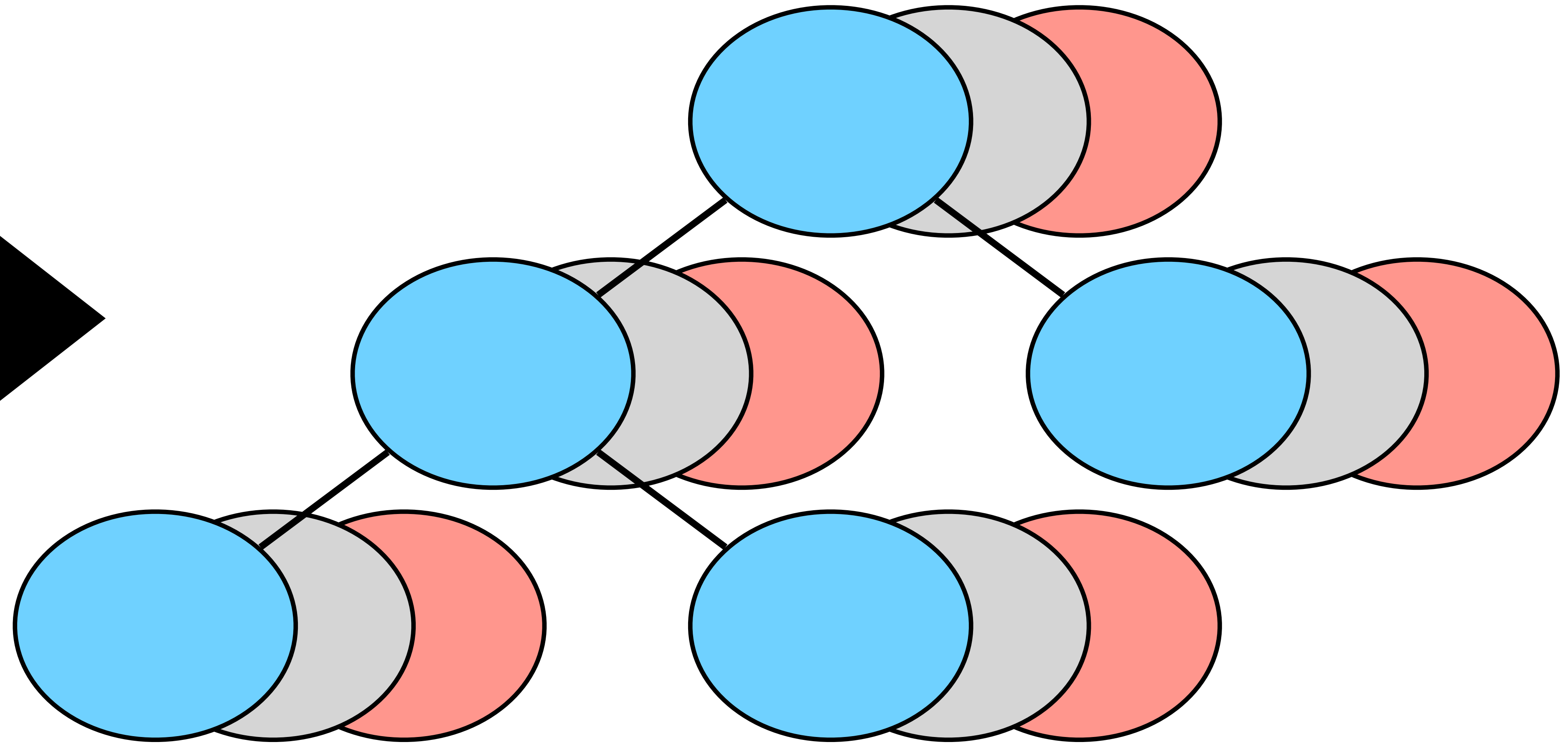
In Existing Functional Languages...

- popular “two-level” design [Doligez-Leroy-Gonthier]
  - used by multicore OCaml, GHC Haskell, Manticore, Caml Light, ...
  - minor and major heaps
  - parallel allocation+GC in minor heaps
- invariants:
  - **no cross-pointers** between minor heaps
  - restrictions between major and minor heaps
- **promotions** maintain invariants
  - moving (copying) data from minor to major
- **problem: shared data must live in major heap**
  - scheduler actions trigger promotions
  - high overhead, no provable efficiency (e.g. unbounded space)





Is there a better way?



Disentanglement

“concurrent tasks remain oblivious
to each other’s allocations”

MaPLe Compiler

github.com/mpllang/mpl

- based on MLton, **full Standard ML language**, extended with

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

- **parallel memory management based on disentanglement**
- used by 500+ students at CMU each year
- outperforms existing implementations of functional languages
- competitive with state-of-the-art imperative/procedural (including Java, Go, C/C++)

MPL vs multicore OCaml:
~2x average speedup [1]
MPL vs GHC Haskell:
~2x average speedup [1]
MPL vs Manticore:
2-50x speedup [2]

[1] *Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations.* Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. ICFP 2021

[2] *Disentanglement in Nested-Parallel Programs.* Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. POPL 2020

Sorting Shootout

	serial (1 proc) T_1	parallel (72 procs) 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

2^{nd} fastest, behind Cilk

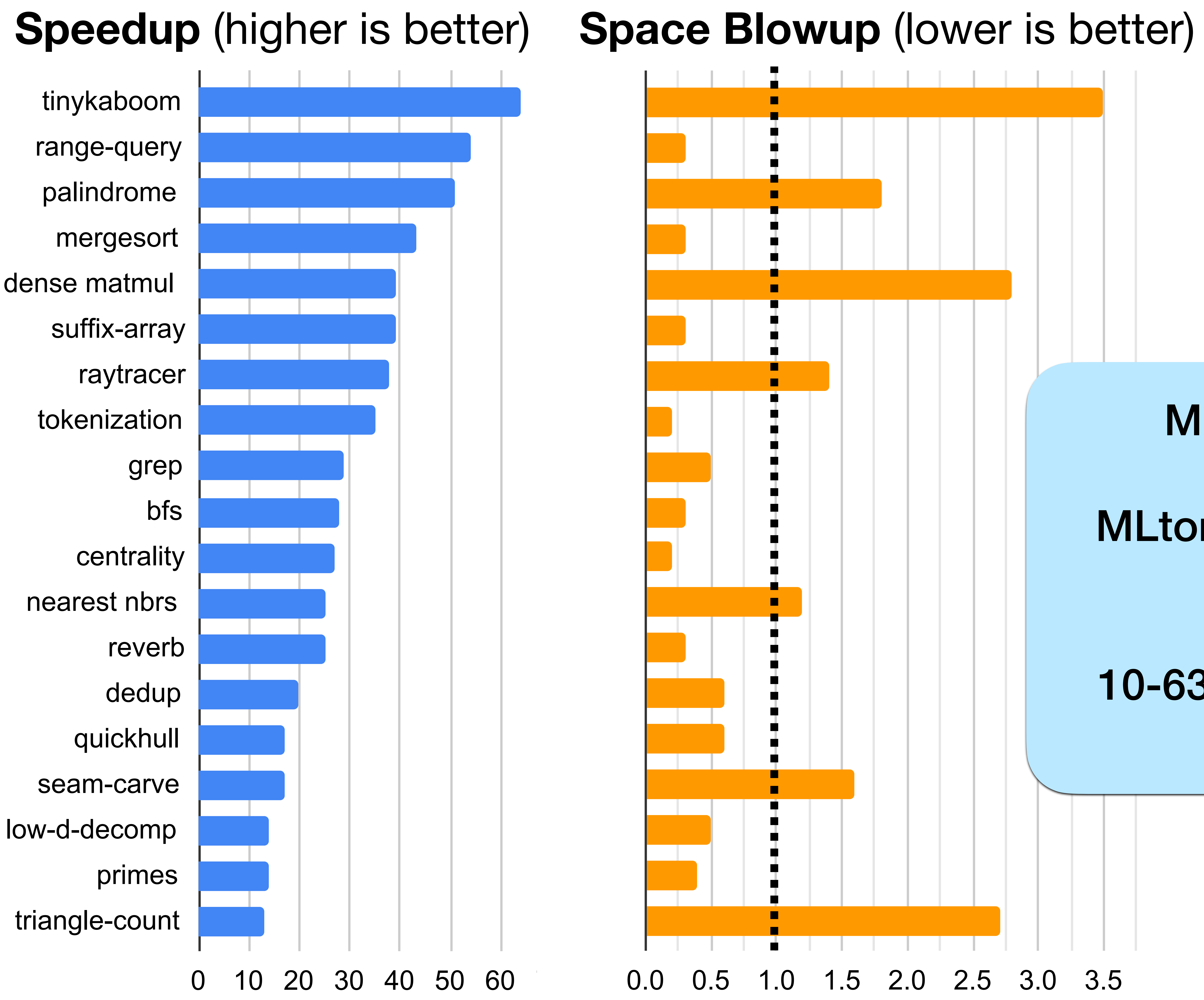
40% faster than Go

70% faster than Java

Parallel ML Benchmarks

- **all disentangled**
- many ported from highly-optimized C/C++
  - PBBS, Ligra, and PAM benchmark suites
- excellent performance
- in general, **within 2-3x of hand-optimized C/C++**
  - e.g. delaunay triangulation, factor 2
- in some cases, **can match C/C++**
  - e.g. linefit: near optimal on our 72-core machine (max read bandwidth)

graphs	betweenness centrality
	breadth-first search
	minimum spanning tree
	low-diameter decomposition
	triangle counting
geometry	delaunay triangulation
	quickhull
	nearest neighbors
	skyline
	2D range query
images	seam carving
	raytracing
	GIF encode+decode
audio	reverb
	WAV encode+decode
text	tokenization
	grep, wc
	palindrome
	suffix array
numeric	integration
	dense+sparse matrix mult
	LU-decomposition
	bignum add, mult
	mandelbrot
other	n-body
	sorting
	histogram
	line fit
	remove duplicates
	mcss
	n-queens



MPL (72 processors)
vs
MLton (sequential baseline)

10-63x speedup, often with less space (!)

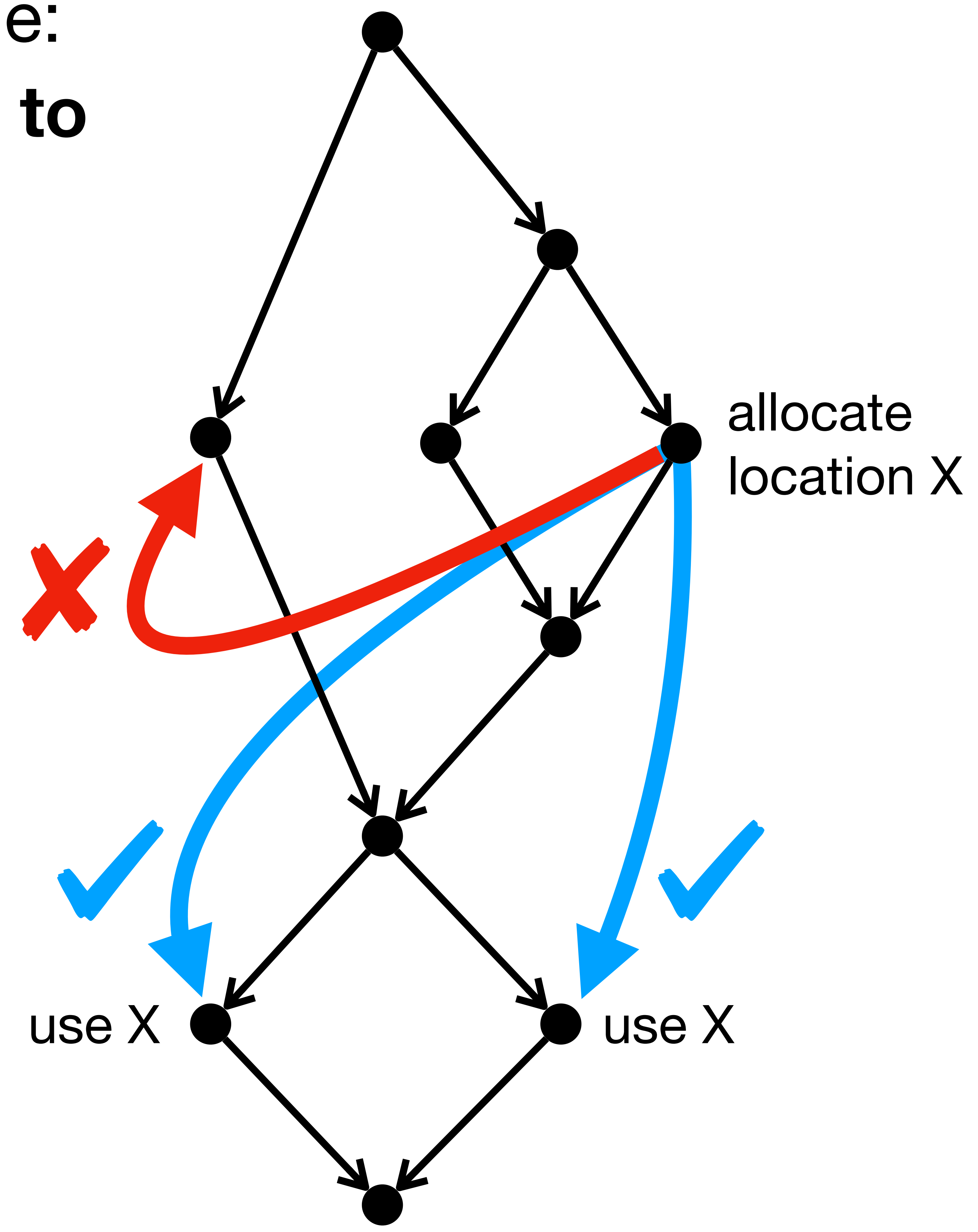
Disentanglement

graphs	betweenness centrality breadth-first search minimum spanning tree low-diameter decomposition triangle counting
geometry	delaunay triangulation quickhull nearest neighbors skyline 2D range query
images	seam carving raytracing GIF encode+decode
audio	reverb WAV encode+decode
text	tokenization grep, wc palindrome suffix array
numeric	integration dense+sparse matrix mult LU-decomposition bignum add, mult mandelbrot
other	n-body

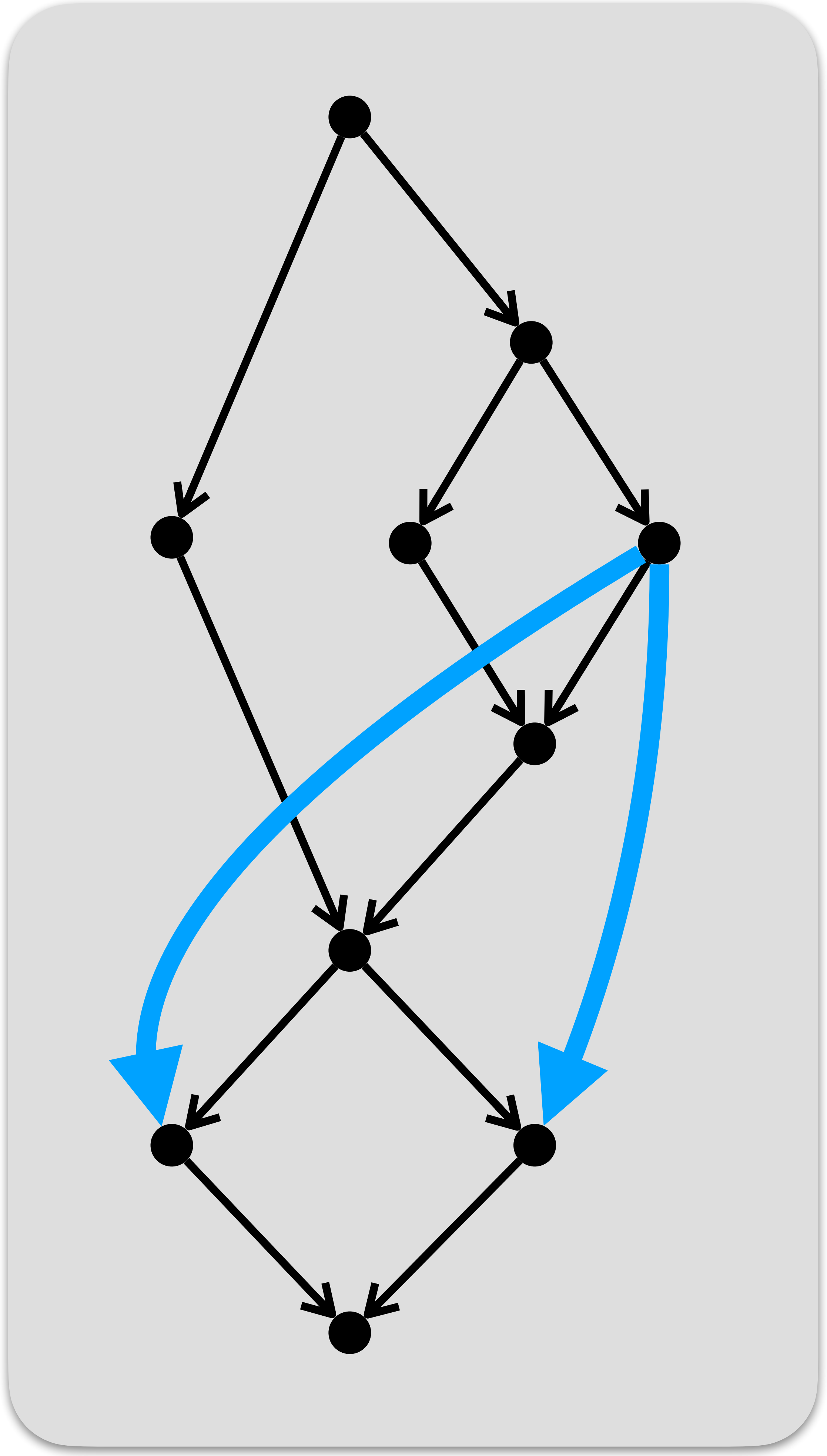
- observed in efficient parallel code: **concurrent tasks are oblivious to each other's allocations**

- in computation graph: **allocation precedes use**

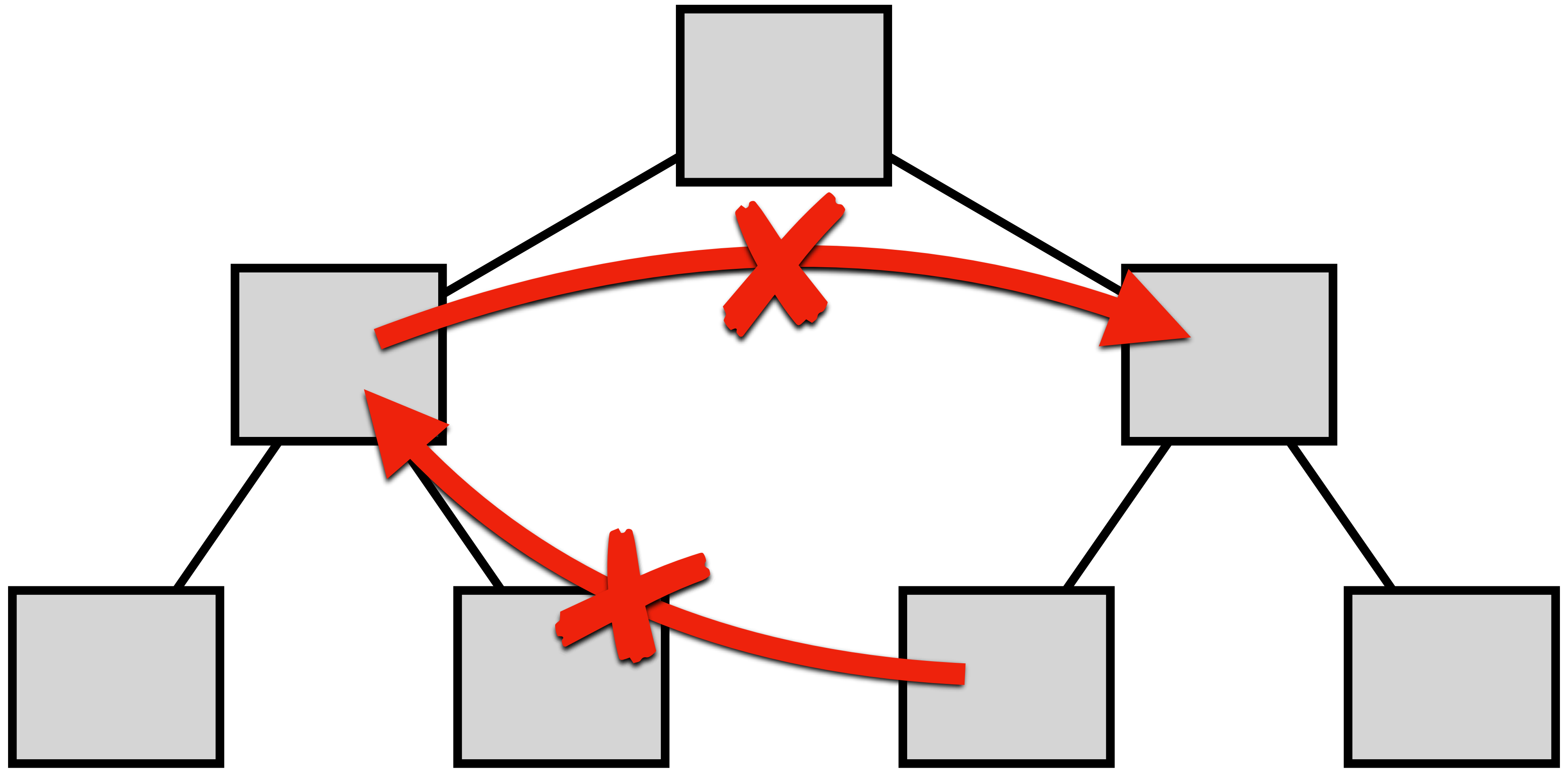
- arbitrary? no: **guaranteed by race-freedom** [Westrick et al. 2020]



Disentanglement



How to utilize disentanglement
for improved efficiency and scalability?

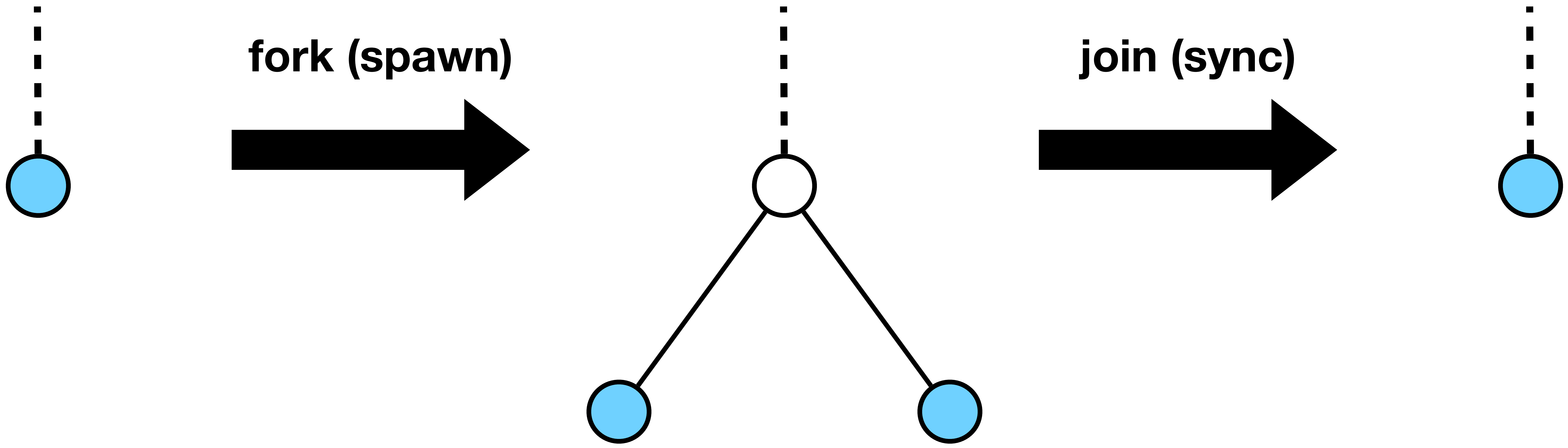


idea: organize memory to reflect **structure of parallelism**

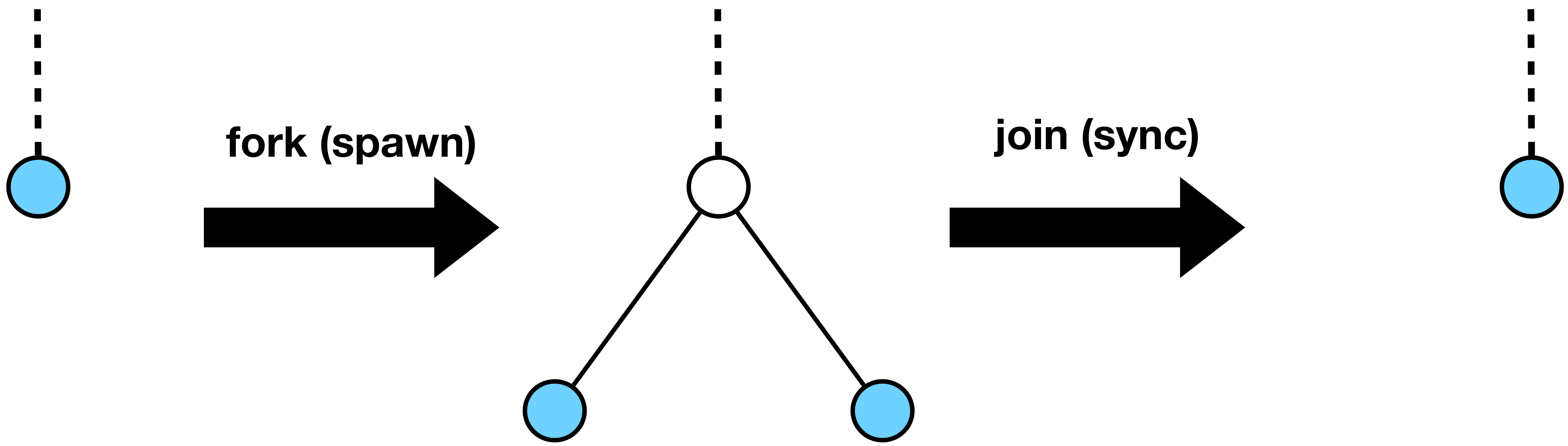
Nested Fork/Join Parallelism

classic and popular technique

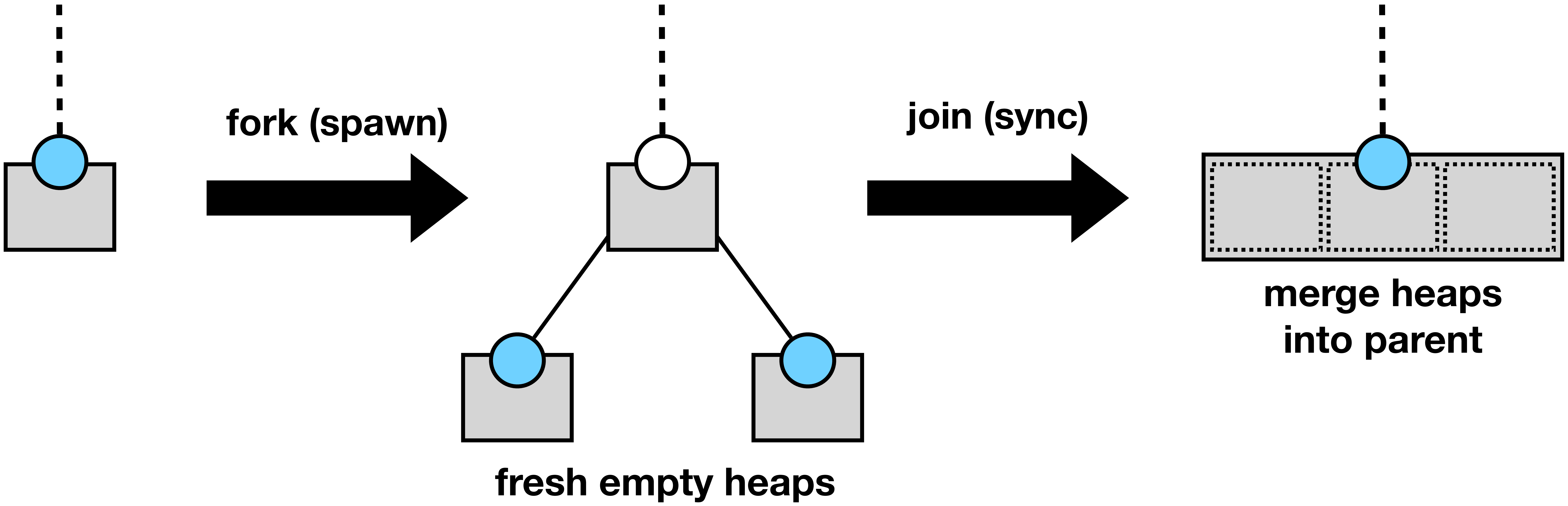
- Cilk, ParlayLib, Intel TBB, Microsoft TPL, OpenMP, Legion, Rayon, Fork/Join Java, Habanero Java, X10, multiLisp, Id, NESL, parallel Haskell, Manticore, Futhark, SML#, etc.



Task-Local Heaps

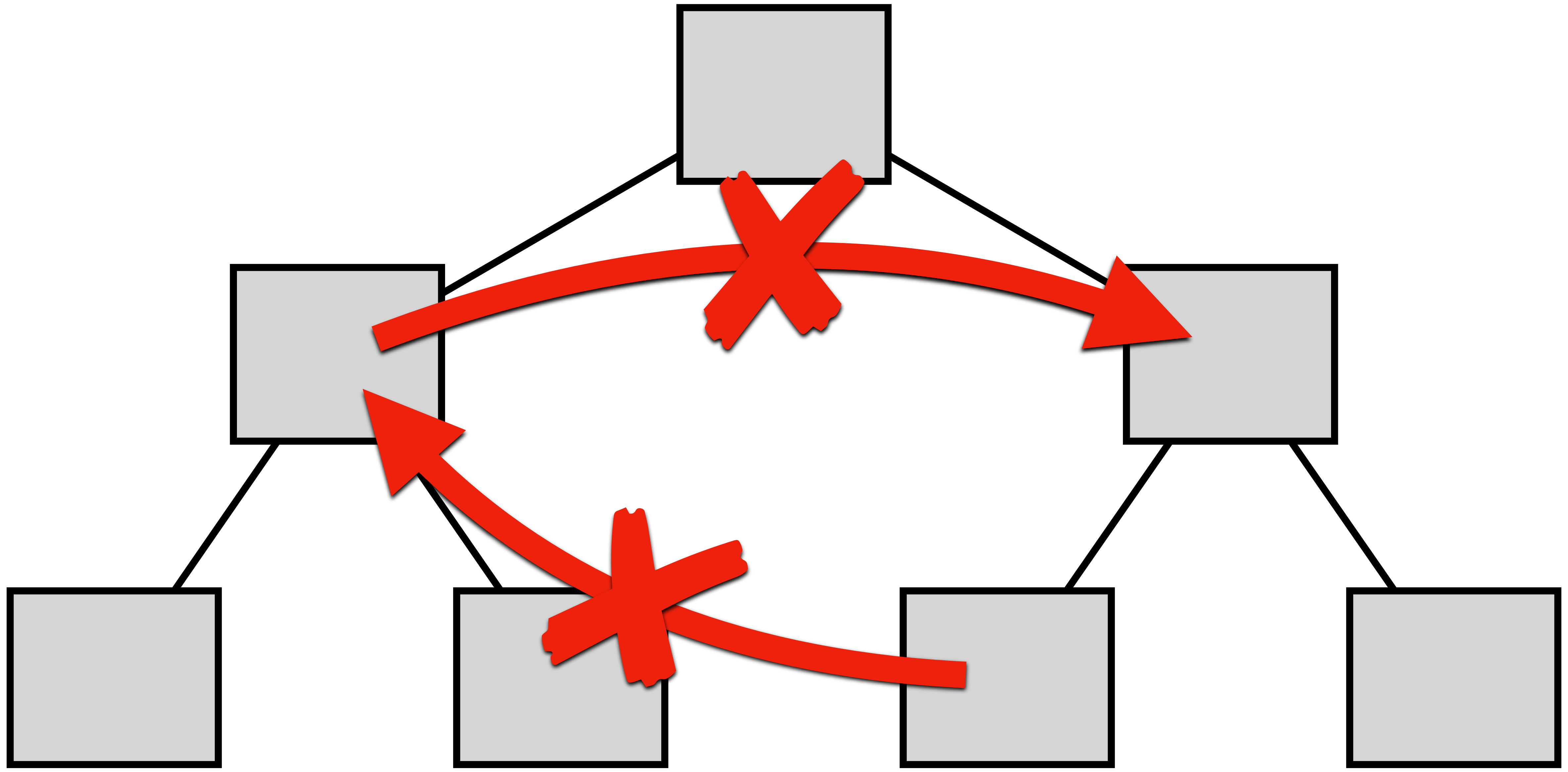


Task-Local Heaps



Disentangled Memory Management

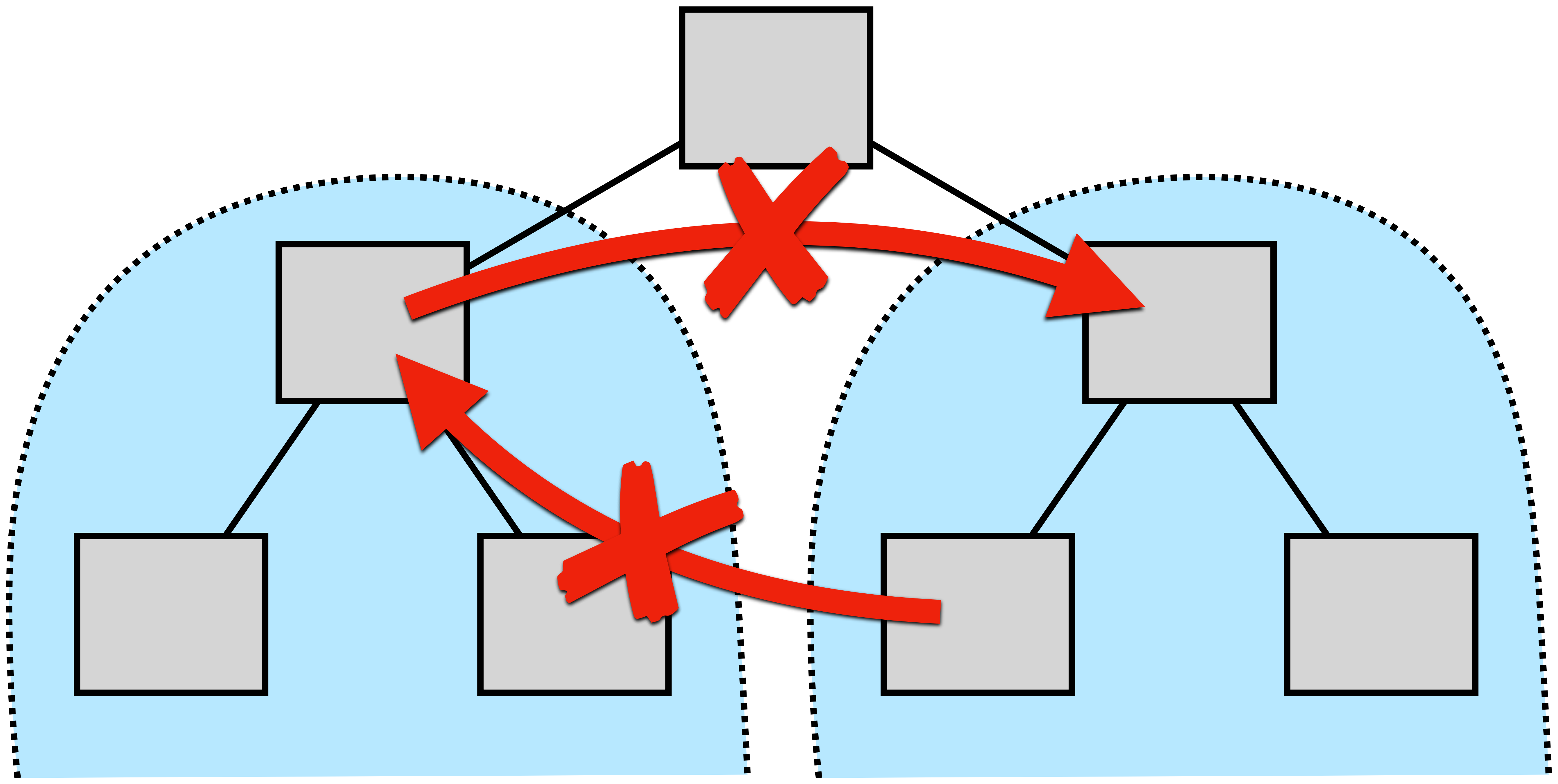
- disentanglement: ***no cross pointers***



Disentangled Memory Management

- disentanglement: ***no cross pointers***
- subtree collection

reorganize, compact, etc. inside subtree



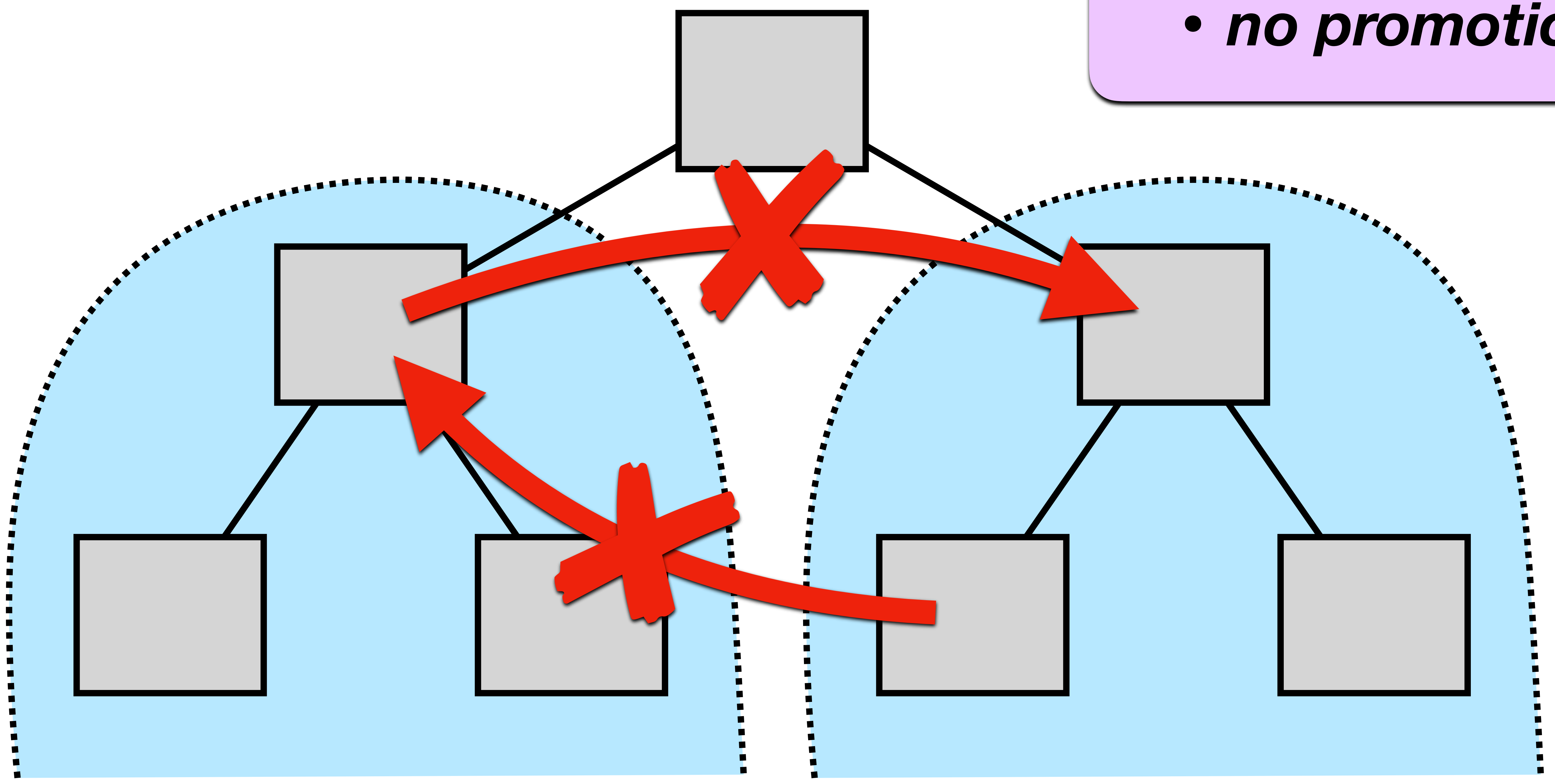
naturally parallel

Disentangled Memory Management

- disentanglement: ***no cross pointers***
- subtree collection
- internal collections and provable efficiency
  [Arora et al. POPL 21]

Implementation Notes:
- carefully integrated with scheduler
  - new heaps only on steals
- write barrier for down-pointers
- no read barrier
- ***no promotions necessary***

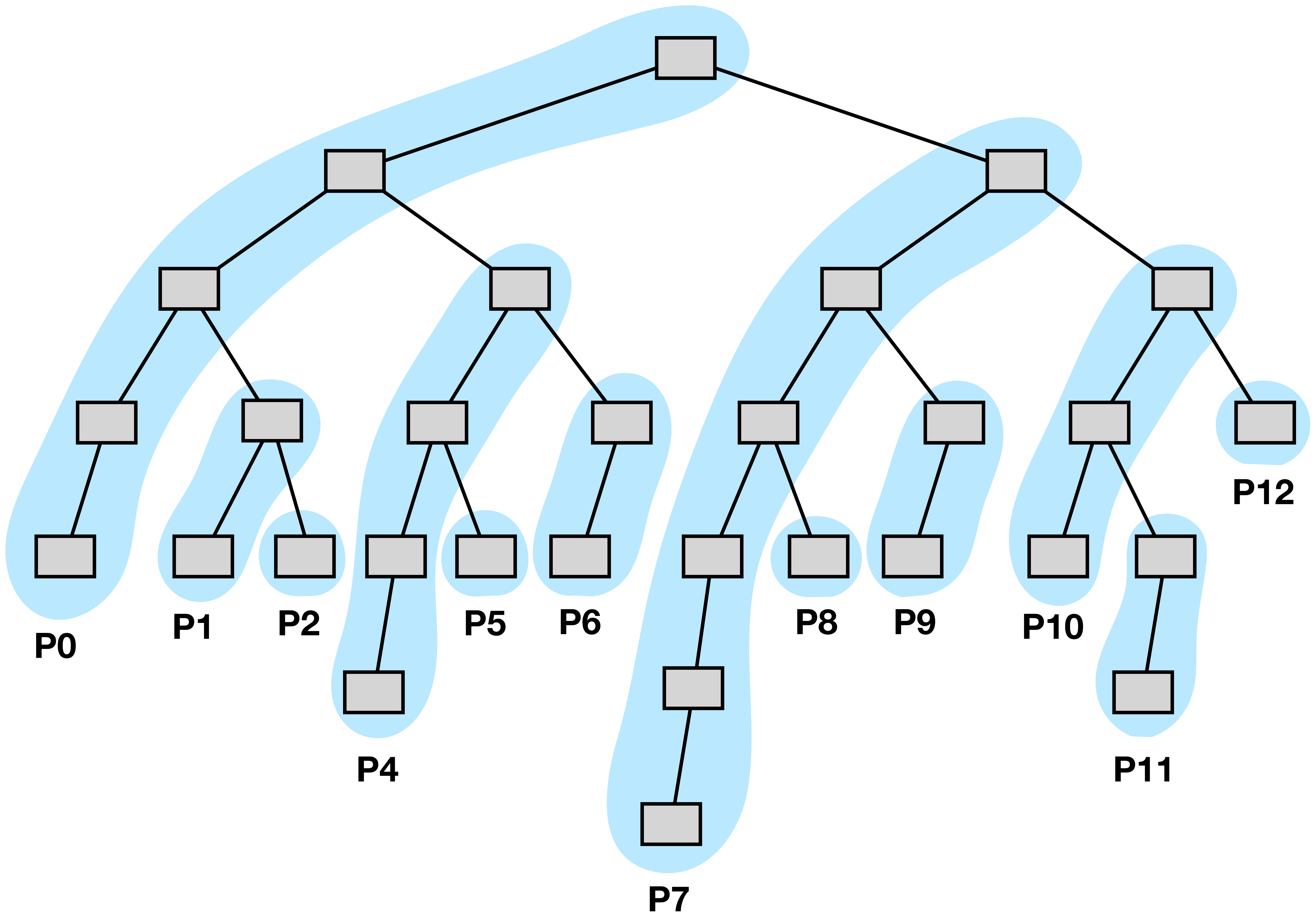
reorganize, compact, etc. inside subtree



naturally parallel

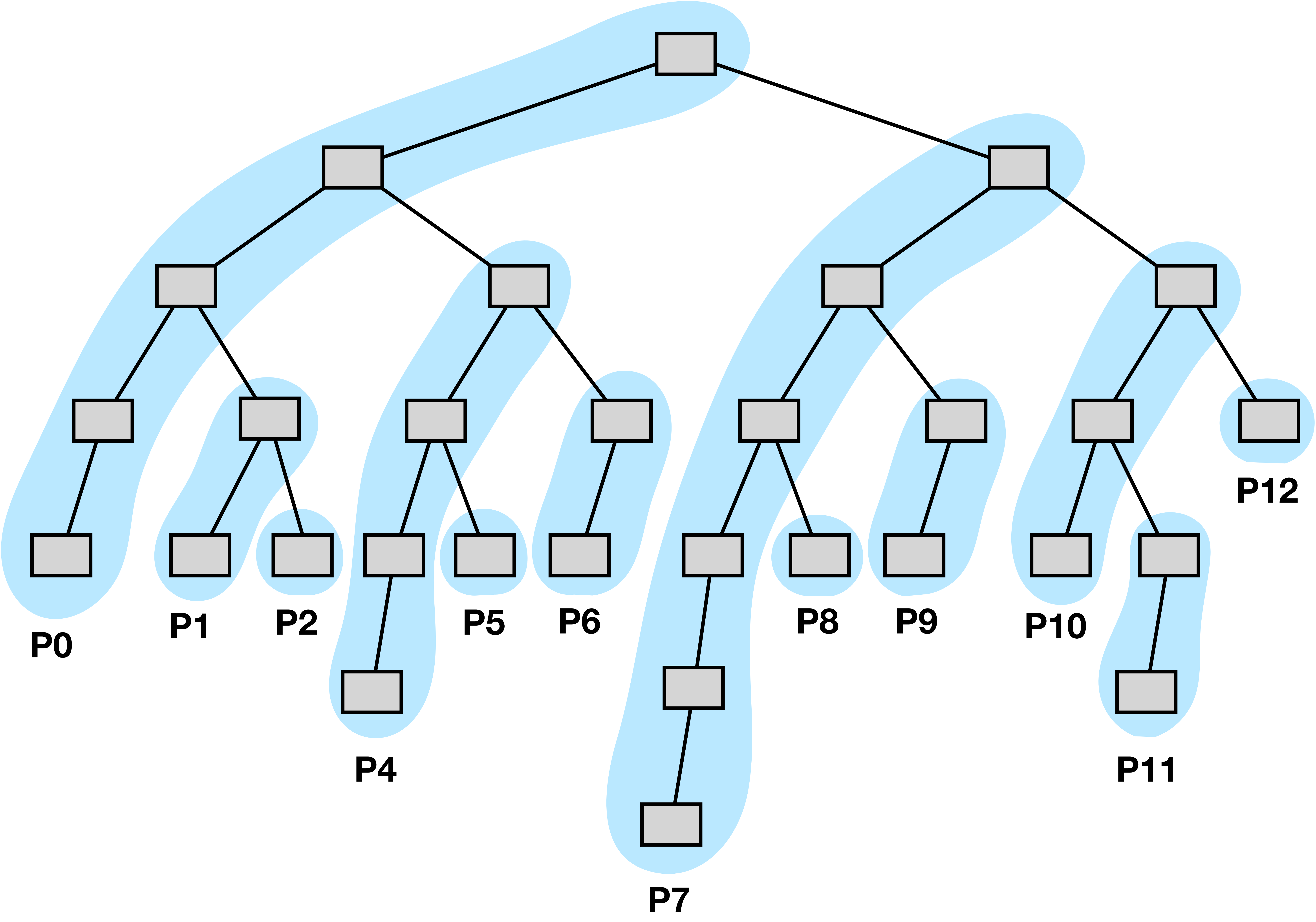
Heap Scheduling

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

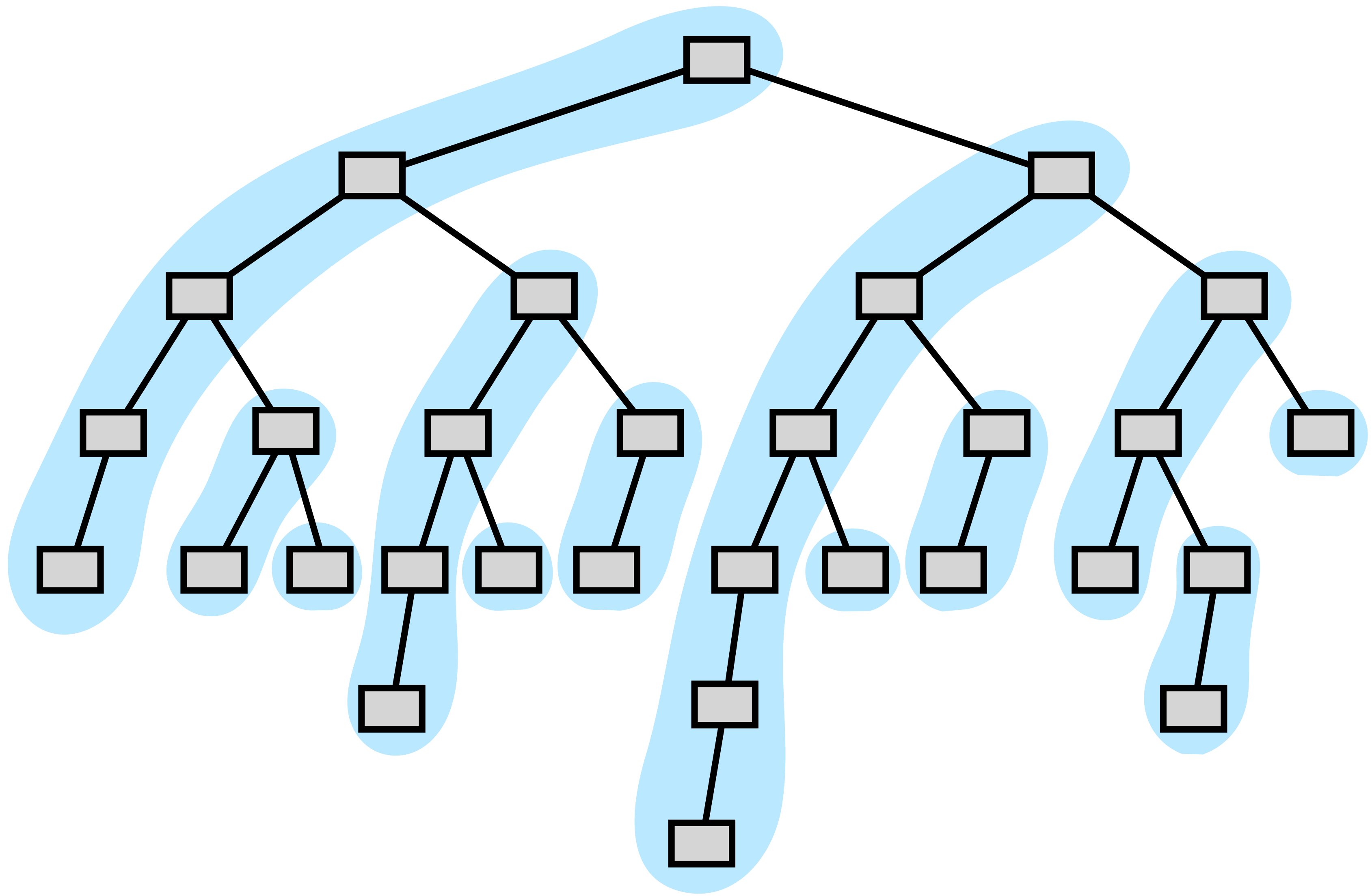


Heap Scheduling

- goal: assign heaps to processors
- each processor manages its own memory
- **integrate closely with thread scheduling** (work-stealing)



Collection Policy



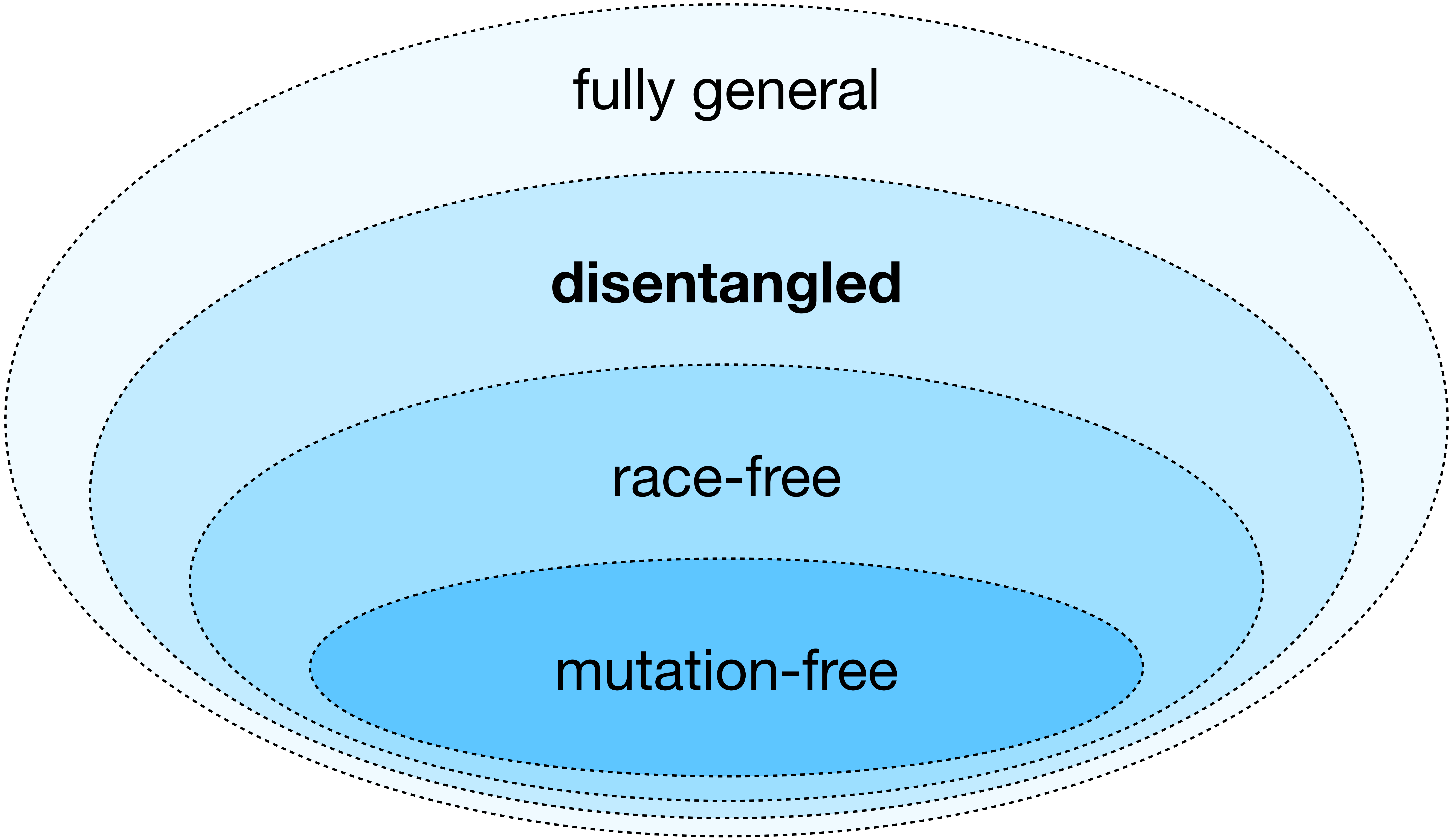
algorithm

- 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 idea:

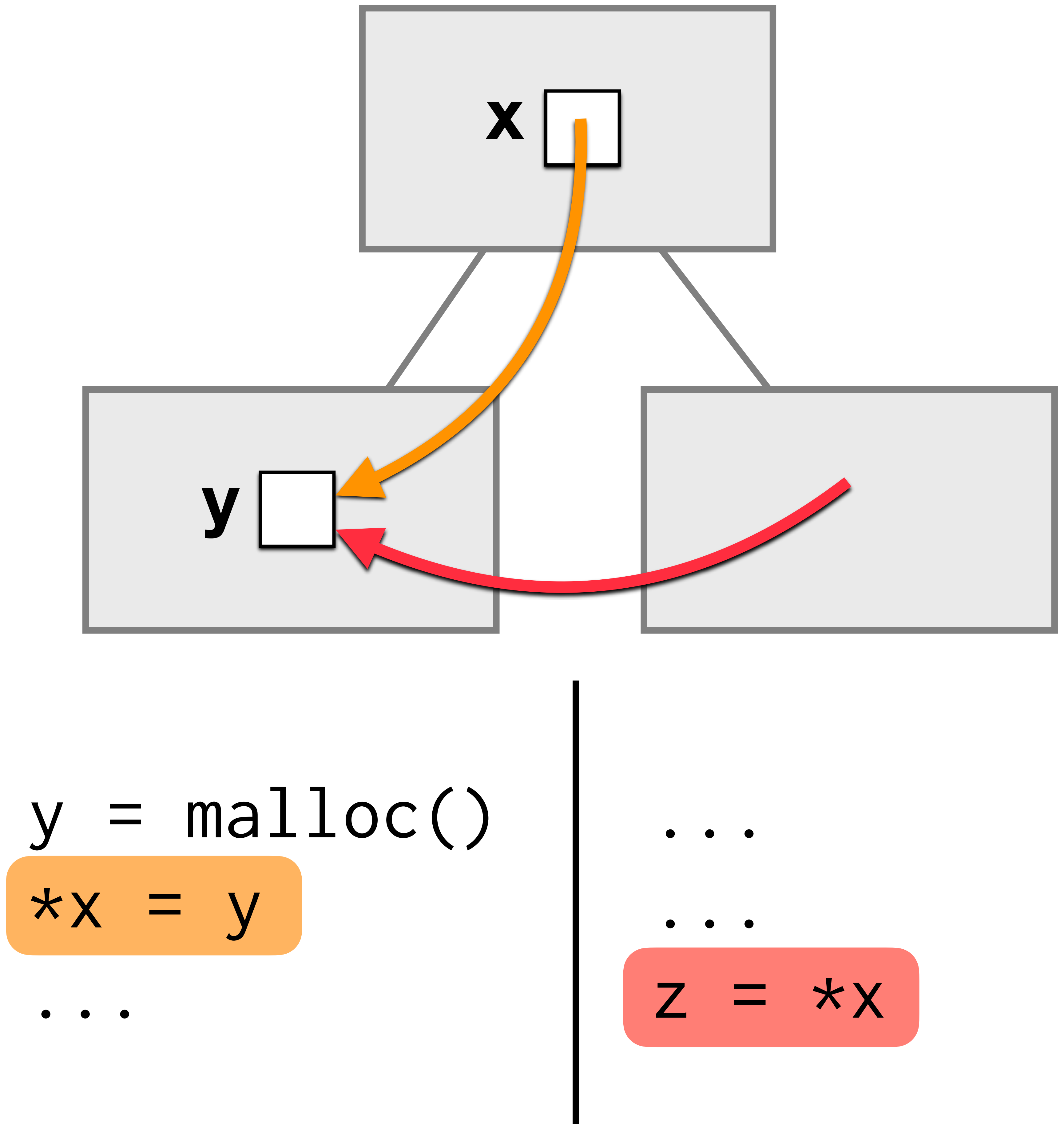
- spines resemble sequential execution
- local counters L_p cannot exceed ***R***


fully general
disentangled
race-free
mutation-free

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

Intuition

- if entangled, must be a **read/write** race
- **write:** creates down-pointer
- **read:** discovers data across



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

Writing Disentangled Programs

pure library interface

tabulate
map
reduce
scan
filter
flatten
merge
...

fast implementation w/ “local” effects

...

purely functional, parallel, disentangled algorithms

```
fun mergesort(X) =
  if length(X) <= granularity then
    quicksort(X)
  else
    let
      val (L,R) = split(X)
      val (sL,sR) = par(fn _ => mergesort(L),
                        fn _ => mergesort(R))
    in
      merge(sL,sR)
    end
```

no need to know about disentanglement!

only 10% more time+memory than hand-optimized

Writing Disentangled Programs

pure library interface

tabulate
map
reduce
scan
filter
flatten
merge
...

fast implementation w/ “local” effects

...

purely functional, parallel, disentangled algorithms

15-210 (Undergrad Course)
Parallel and Sequential
Data Structures and Algorithms

parentheses matching
max contiguous subsequence
prime sieve
sorting
order statistics
range query
graph search
connected components
shortest paths
minimum spanning forest
dynamic programming
hashing
...

no need to know about disentanglement!

Writing Disentangled Programs

pure library interface

tabulate filter
map flatten
reduce merge
scan ...

fast implementation w/ "local" effects

...

~~purely~~ mostly functional, parallel, disentangled algorithms

```
fun forwardBFS(G,s) =
  let
    fun outEdges(u) = map(fn v => (u,v), neighbors(G,u))
    val parents = tabulate(numVertices(G), fn v => -1)
    fun tryVisit(u,v) =
      if compareAndSwap(parents,v,-1,u) then SOME(v) else NONE
    fun search(F) =
      if length(F) = 0 then ()
      else search(filterOp(tryVisit, flatten(map(outEdges, F))))
  in
    tryVisit(s,s);
    search(singleton(s));
    parents
  end
```

Writing Disentangled Programs

pure library interface

tabulate	filter
map	flatten
reduce	merge
scan	...

fast implementation w/ “local” effects

...

Parallel Block-Delayed Sequences
Sam Westrick, Mike Rainey, Daniel Anderson, and Guy Blelloch
PPoPP’22

fusion across library calls

- e.g. only O(#processors) allocation for **map -> scan -> reduce**

Summary

disentanglement

- "concurrent tasks remain oblivious to each other's allocations"
- common property, guaranteed by race-freedom, functional programming
- enables provably efficient parallel memory management and GC

MaPLe implementation

- efficient and scalable
- competitive with low-level imperative code

`github.com/mpllang/mpl`

Future / Ongoing work

- static enforcement of disentanglement (e.g. type system)
- dynamic enforcement ("entanglement management")
- distributed computing?