Retrofitting *Parallelism* onto OCaml

"KC" Sivaramakrishnan

IIT MADRAS

OCaml Labs

Tarides

λ 0x1F42B Segfault Systems

OCaml

Industry

ahrefs
docker
Jane Street
SimCorp
FACEBOOK
Tarides
Tezos
Bloomberg

Projects

The Astrée Static Analyzer
flow
hack
COMPCERT

Industry

Projects

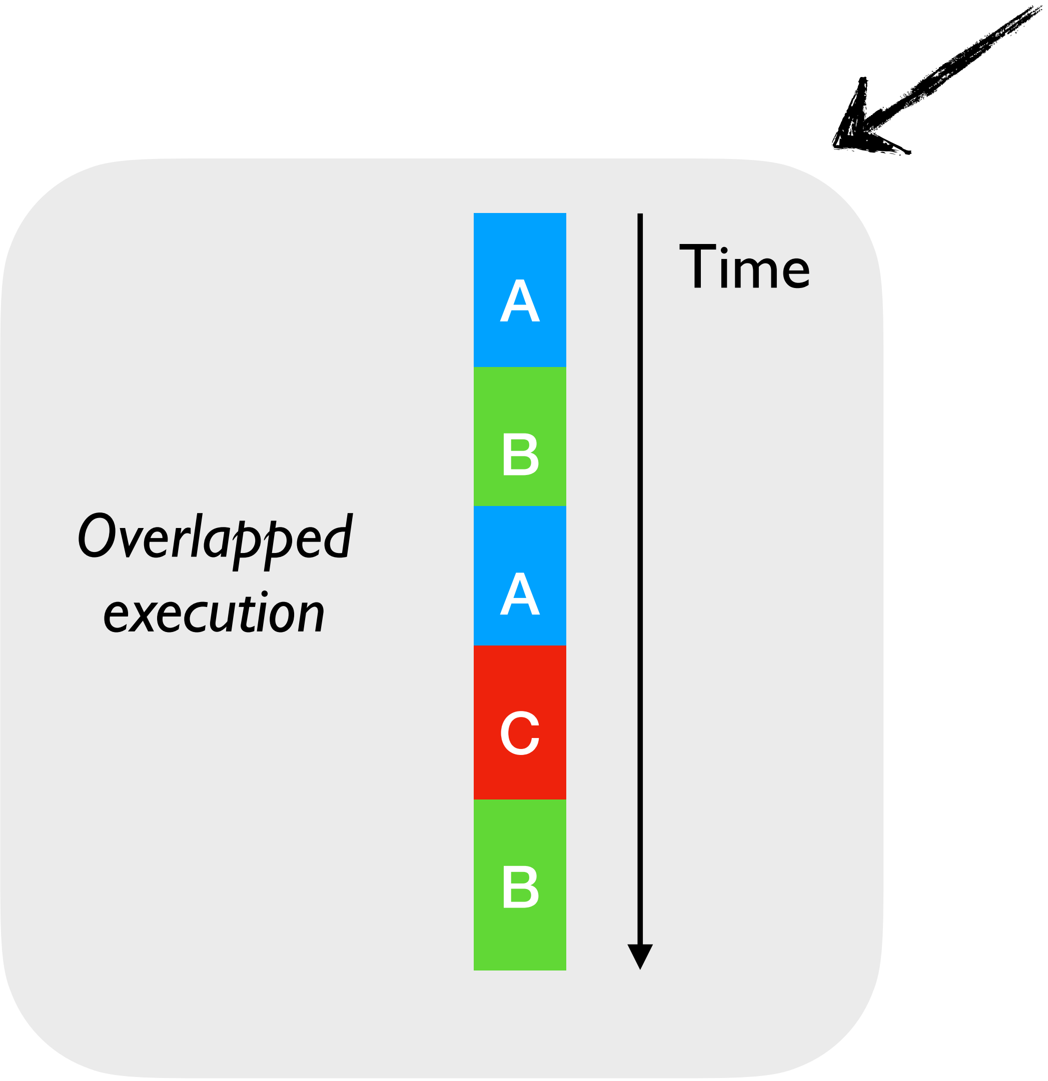
No multicore support!

Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml

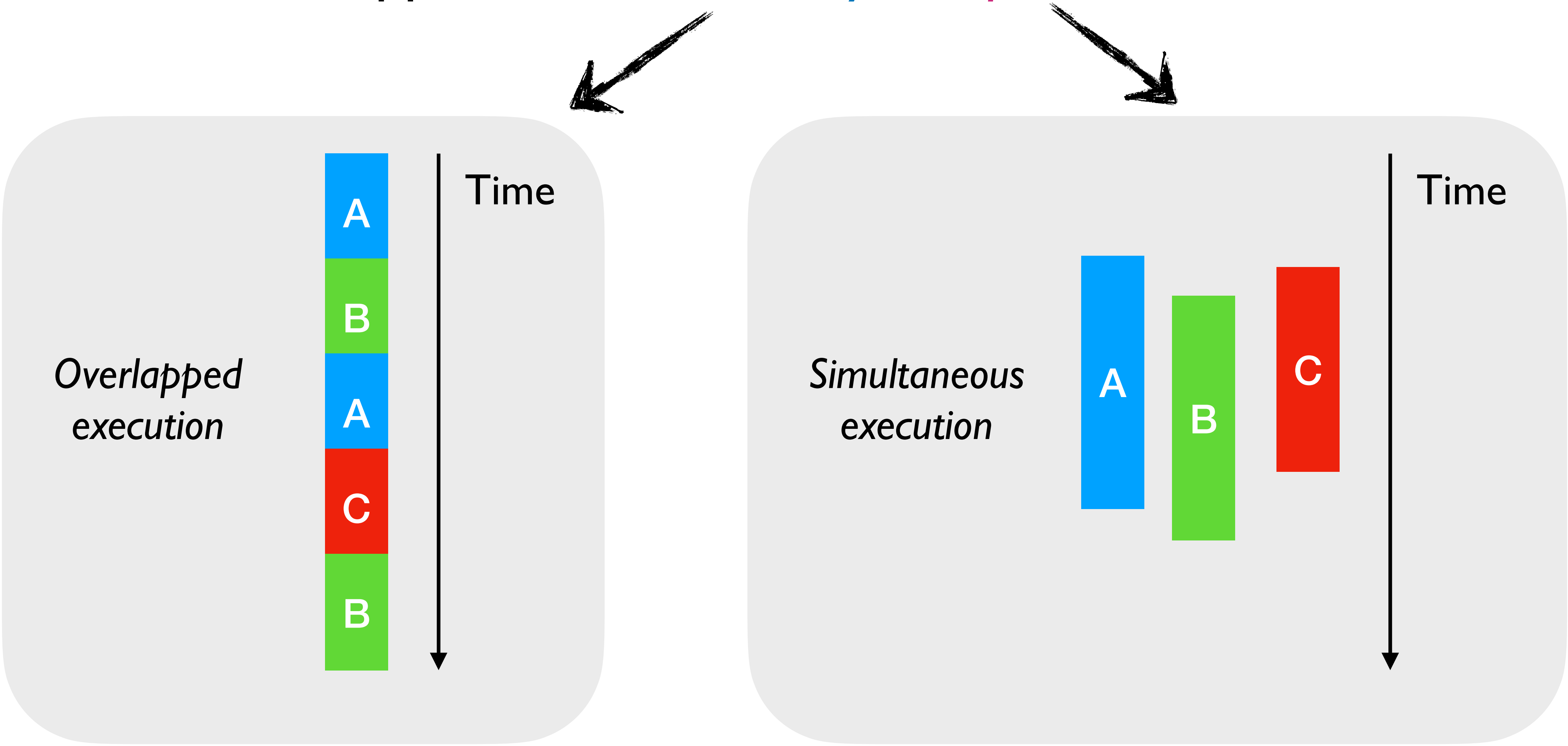
Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml



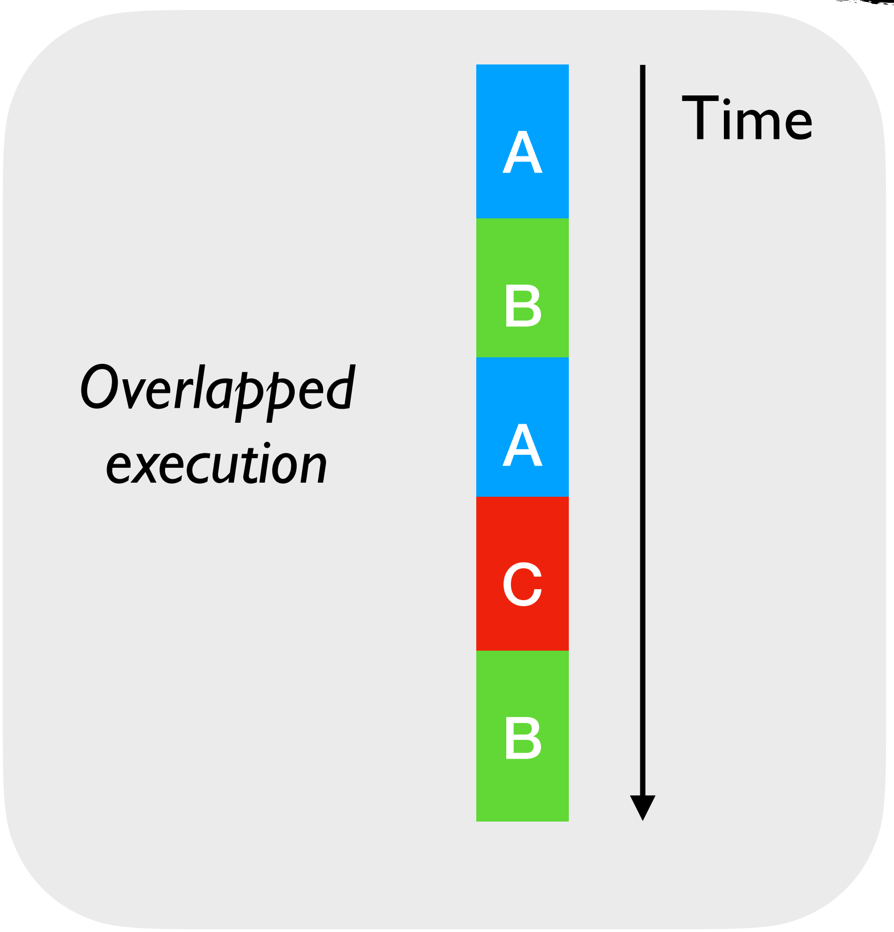
Multicore OCaml

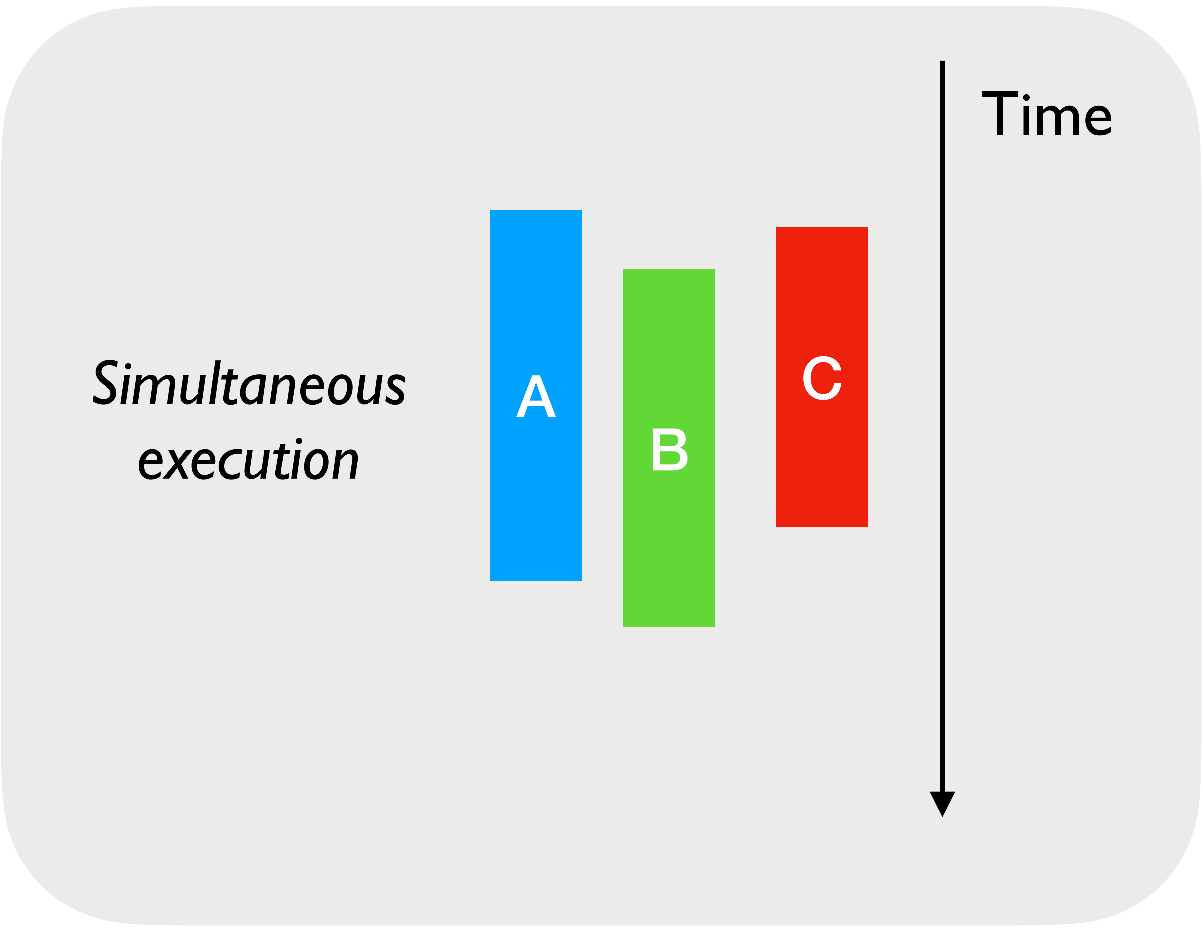
- Adds native support for *concurrency* and *parallelism* to OCaml



Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml

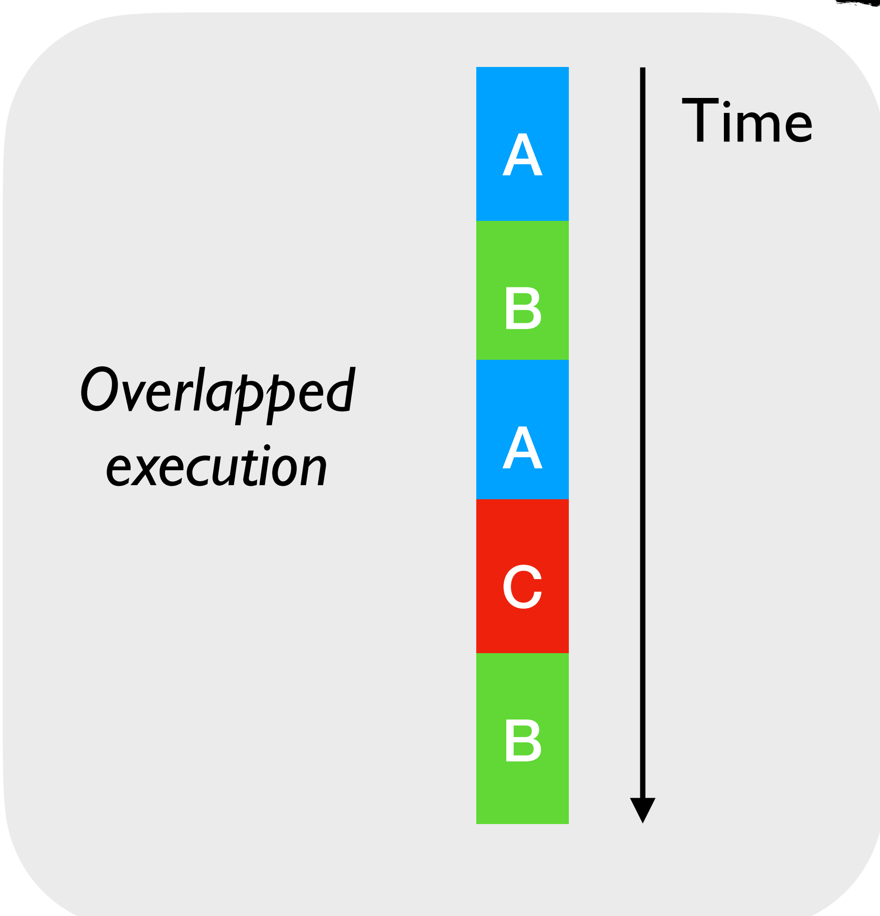




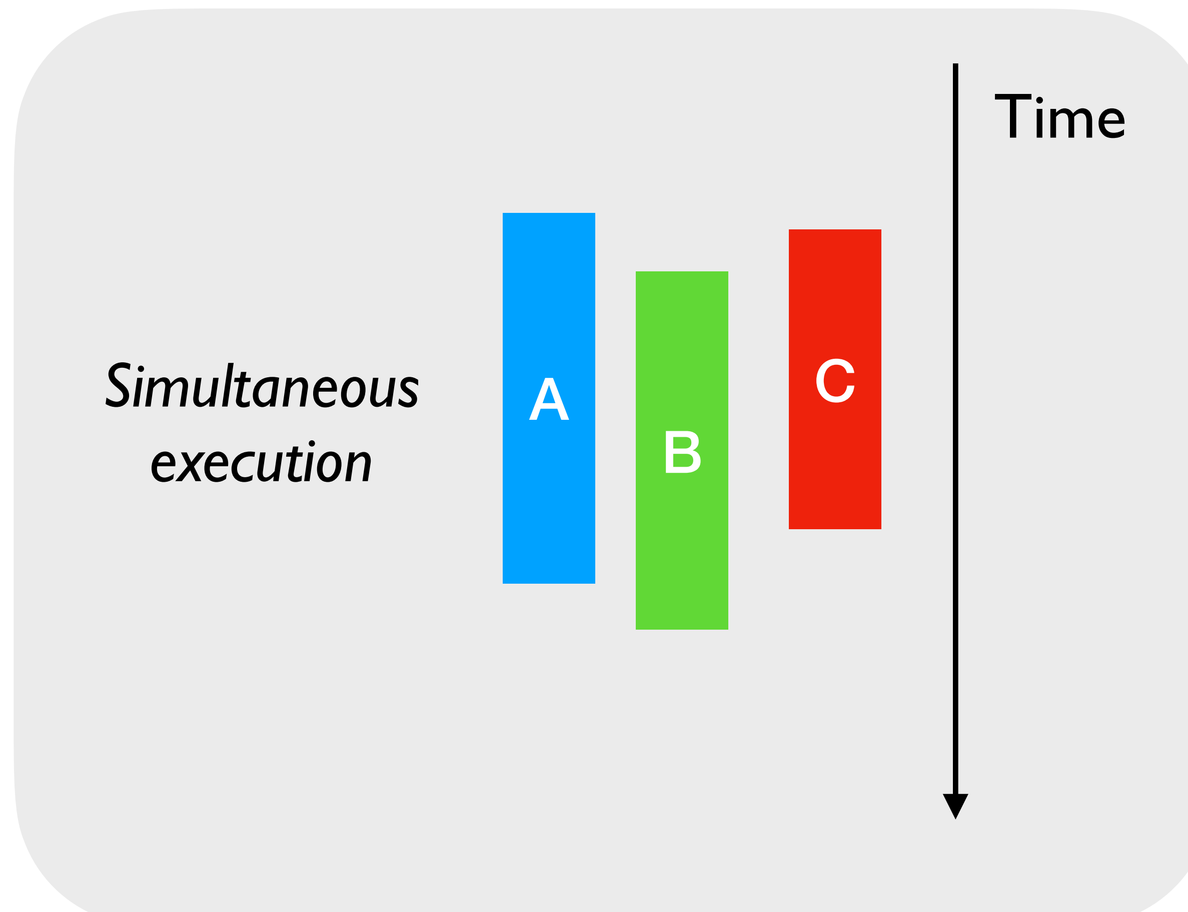
Effect Handlers

Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml



Effect Handlers



Domains

OCaml 5.0

- Domains and *the runtime system support* for effect handlers will land in OCaml 5.0
  - Expected to be released alongside 4.14 (Q2 2022)
  - OCaml 4.xx version will have long-term support

OCaml 5.0

- Domains and *the runtime system support* for effect handlers will land in OCaml 5.0
  - Expected to be released alongside 4.14 (Q2 2022)
  - OCaml 4.xx version will have long-term support

```
opam install 4.12.0+domains+effects
```

Outline of the talk

1. Challenges of adding parallelism to OCaml
2. Deep dive into the new parallel GC for OCaml
3. High-level parallel programming with *Domainslib*

Challenges

- Millions of lines of legacy code
  - Written without *concurrency* and *parallelism* in mind
  - Cost of refactoring sequential code itself is *prohibitive*

Challenges

- Millions of lines of legacy code
  - ✦ Written without *concurrency* and *parallelism* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- Low-latency and predictable performance
  - ✦ Great for applications that tolerate *~10ms* latency

Challenges

- Millions of lines of legacy code
  - ✦ Written without *concurrency* and *parallelism* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- Low-latency and predictable performance
  - ✦ Great for applications that tolerate *~10ms* latency
- Excellent compatibility with debugging and profiling tools
  - ✦ gdb, lldb, perf, libunwind, etc.

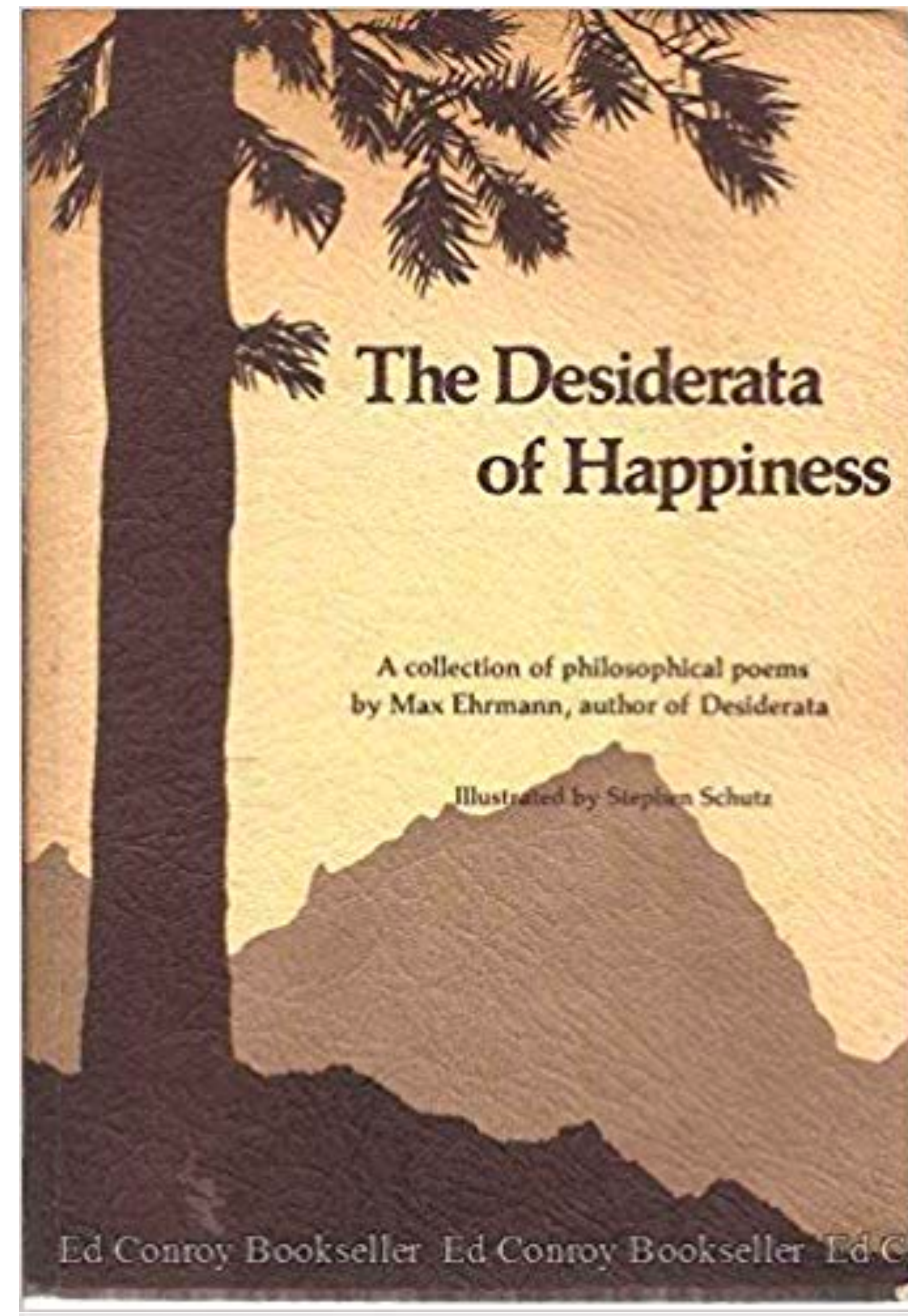
Challenges

- Millions of lines of legacy code
  - Written without *concurrency* and *parallelism* in mind
  - Cost of refactoring sequential code itself is *prohibitive*
- Low-latency and predictable performance
  - Great for applications that tolerate *~10ms* latency
- Excellent compatibility with debugging and profiling tools
  - gdb, lldb, perf, libunwind, etc.

Backwards compatibility before scalability

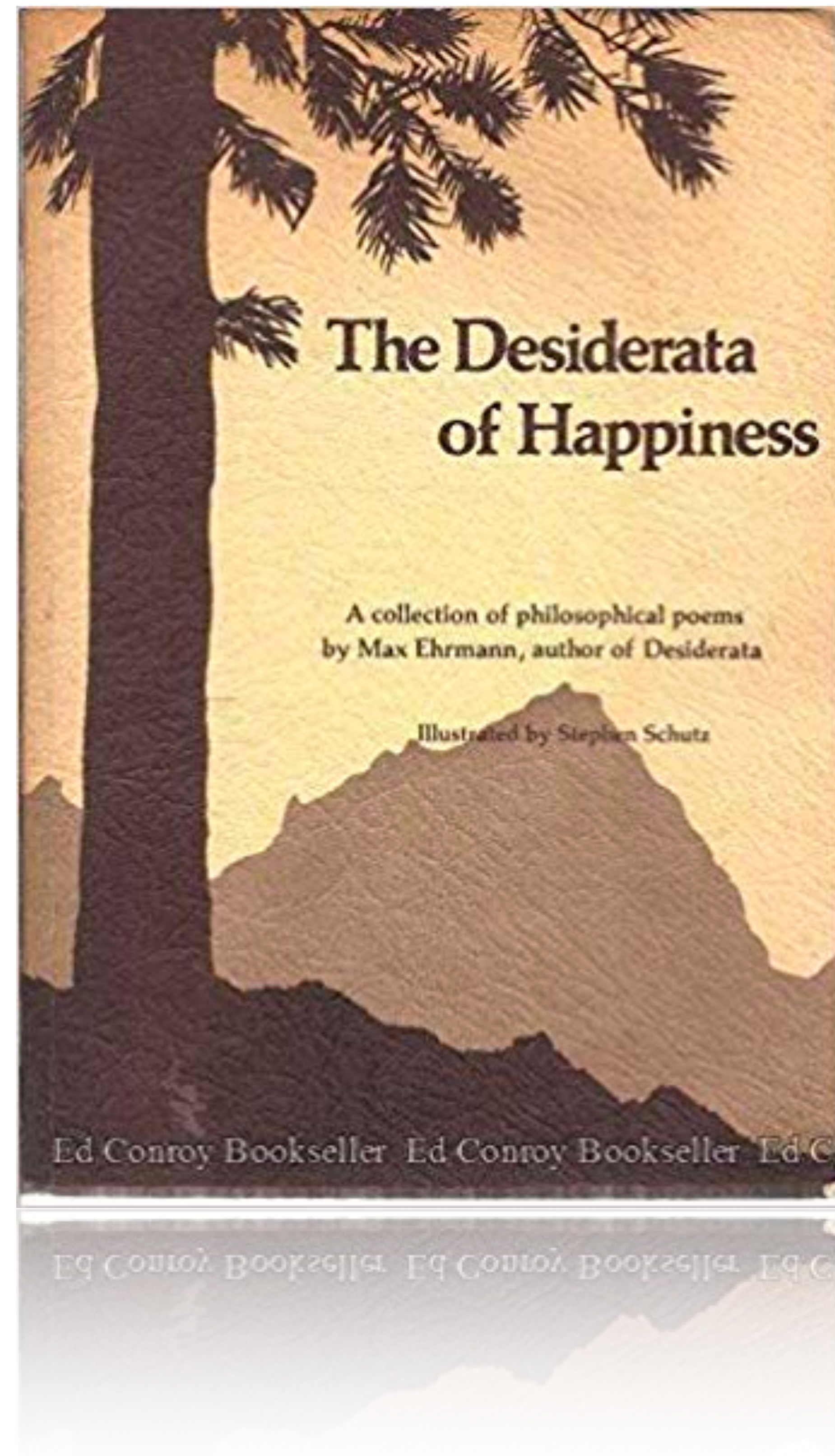
Desiderata

- Feature backwards compatibility
    - ✦ Do not break existing code



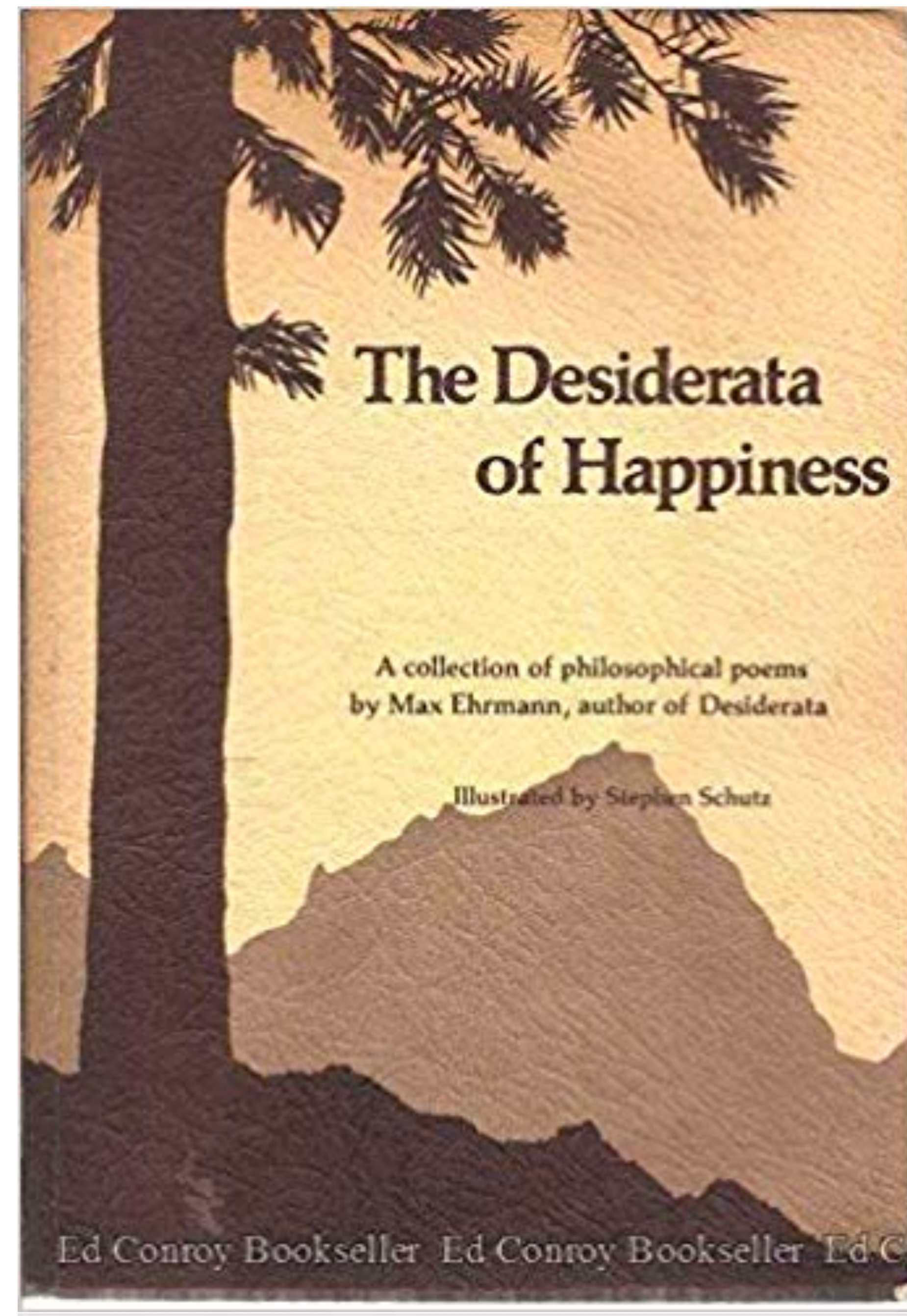
Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory



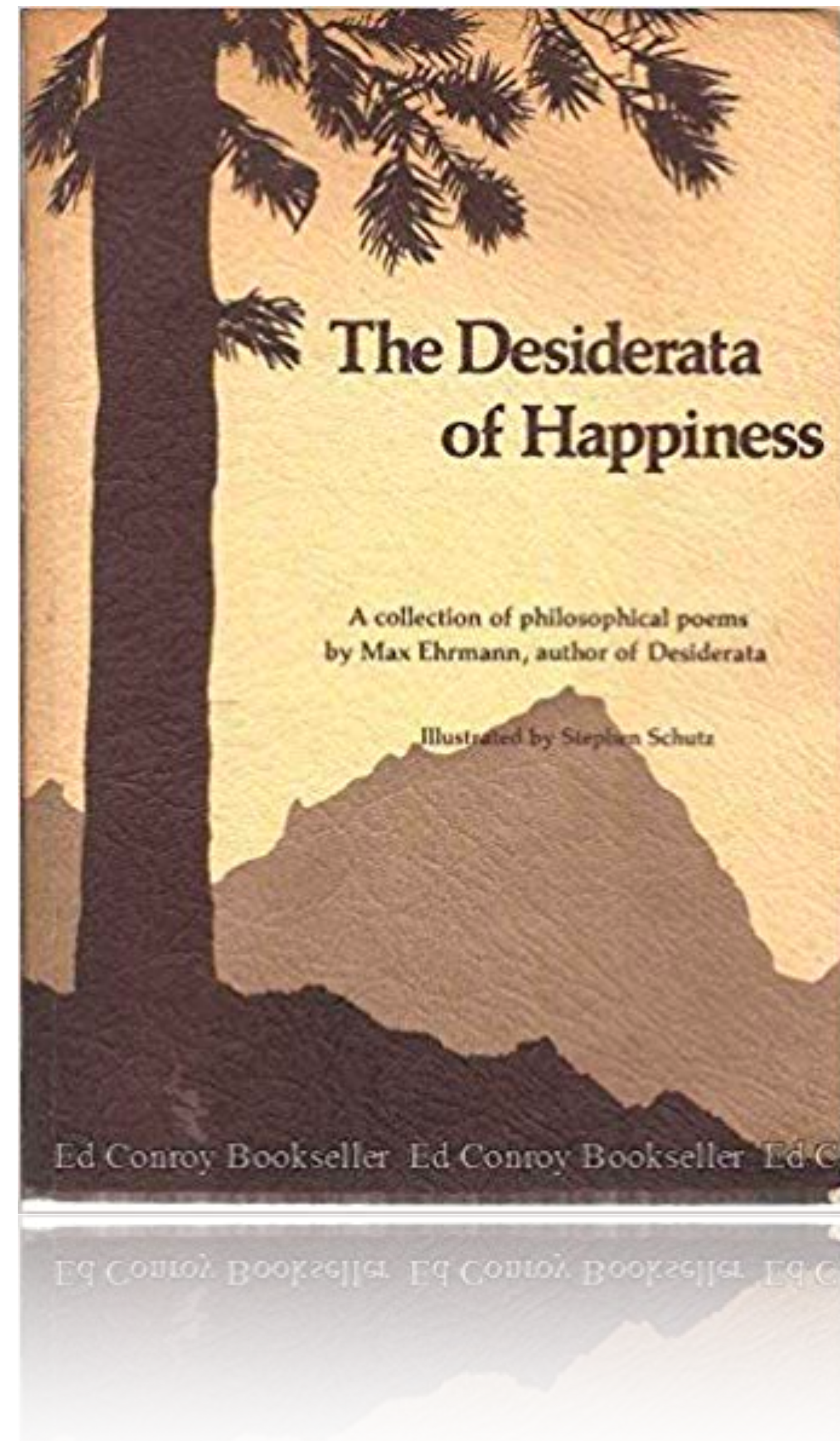
Desiderata

- Feature backwards compatibility
  - Do not break existing code
- Performance backwards compatibility
  - Existing programs run just as fast using just the same memory
- GC Latency before multicore scalability



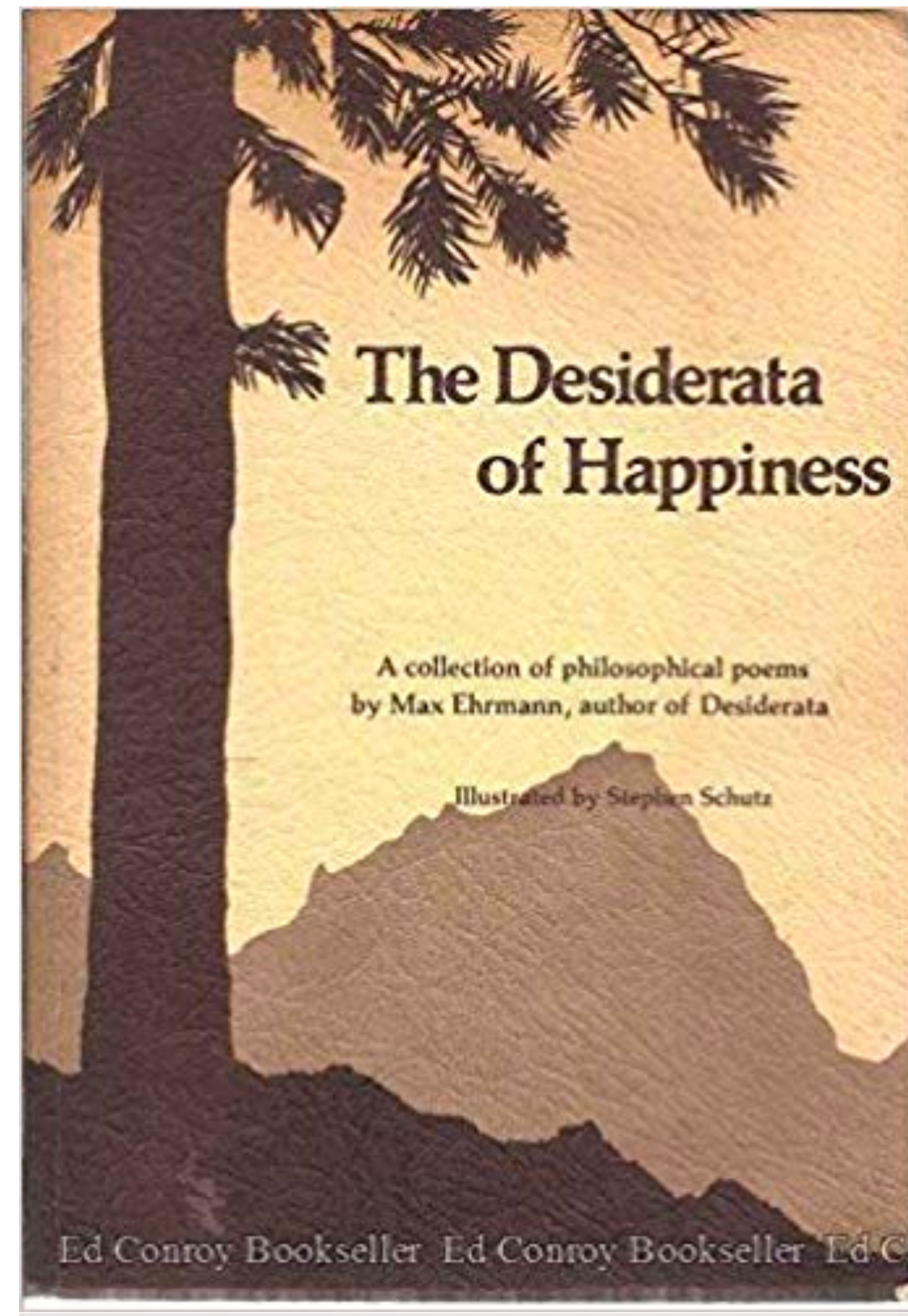
Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory
- GC Latency before multicore scalability
- Compatibility with program inspection tools



Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory
- GC Latency before multicore scalability
- Compatibility with program inspection tools
- Performant concurrent and parallel programming abstractions



Domains for Parallelism

Domains for Parallelism

- A unit of parallelism

Domains for Parallelism

- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
  - Recommended to have 1 domain per core

Domains for Parallelism

- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
    - ✦ Recommended to have 1 domain per core
- Low-level domain API
    - ✦ Spawn & join, wait & notify

```
Domain.spawn : (unit -> ‘a) -> ‘a Domain.t
```

Domains for Parallelism

- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
  - Recommended to have 1 domain per core
- Low-level domain API
  - Spawn & join, wait & notify

    `Domain.spawn : (unit -> ‘a) -> ‘a Domain.t`
  - Domain-local storage
  - Atomic memory operations
    - Dolan et al, “Bounding Data Races in Space and Time”, PLDI’18

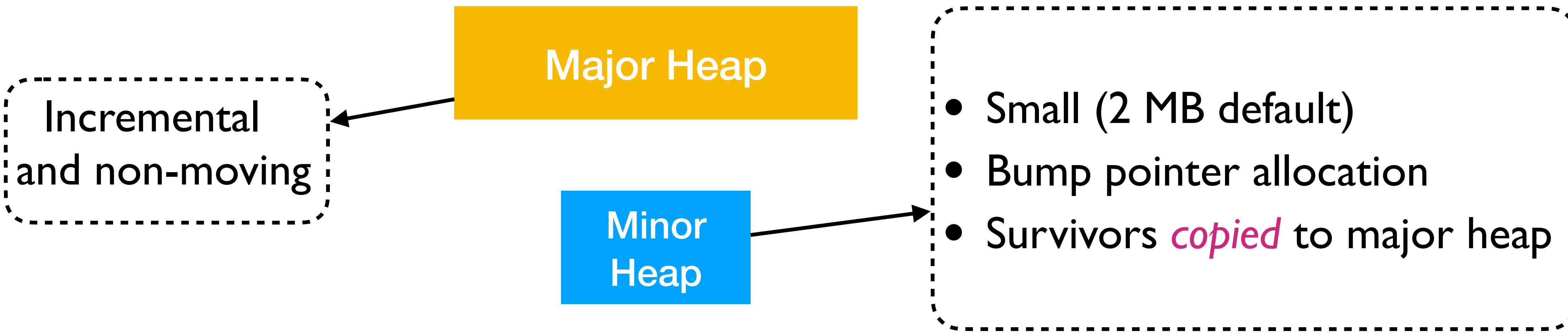
Domains for Parallelism

- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
  - Recommended to have 1 domain per core
- Low-level domain API
  - Spawn & join, wait & notify

    ```
    Domain.spawn : (unit -> 'a) -> 'a Domain.t
    ```
  - Domain-local storage
  - Atomic memory operations
    - Dolan et al, "Bounding Data Races in Space and Time", PLDI'18
- No restrictions on sharing objects between domains
  - But how does it work?

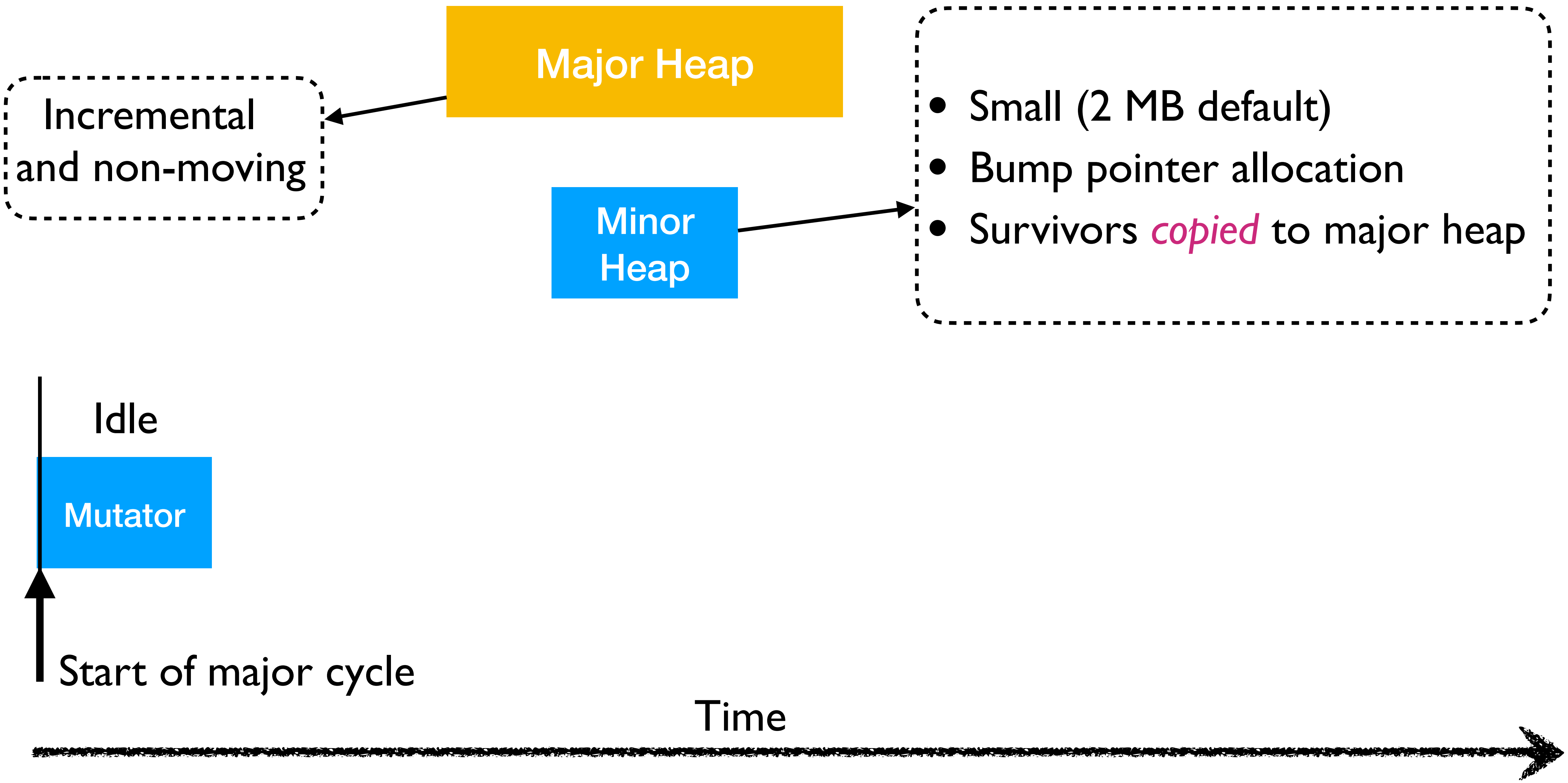
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



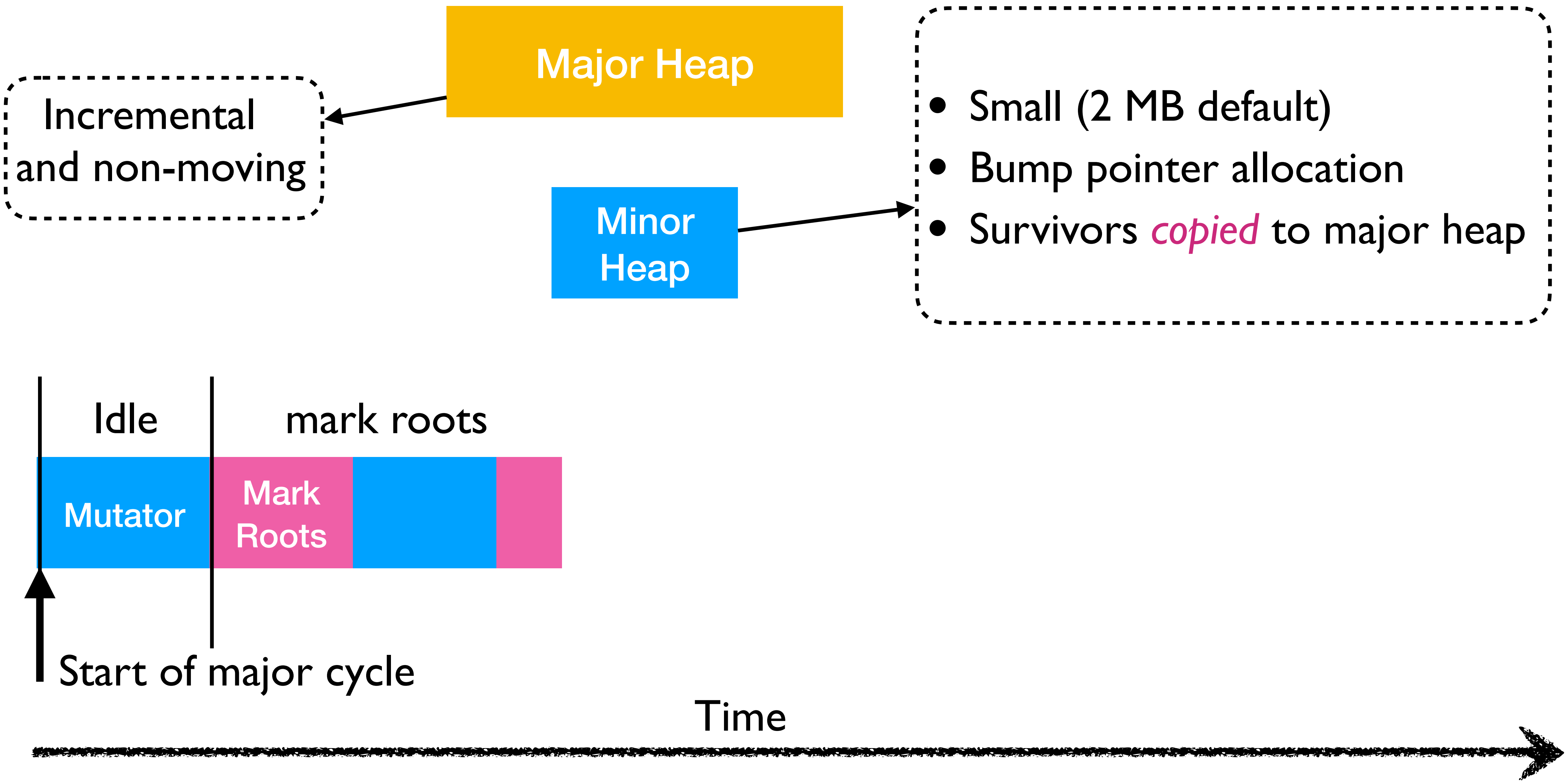
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



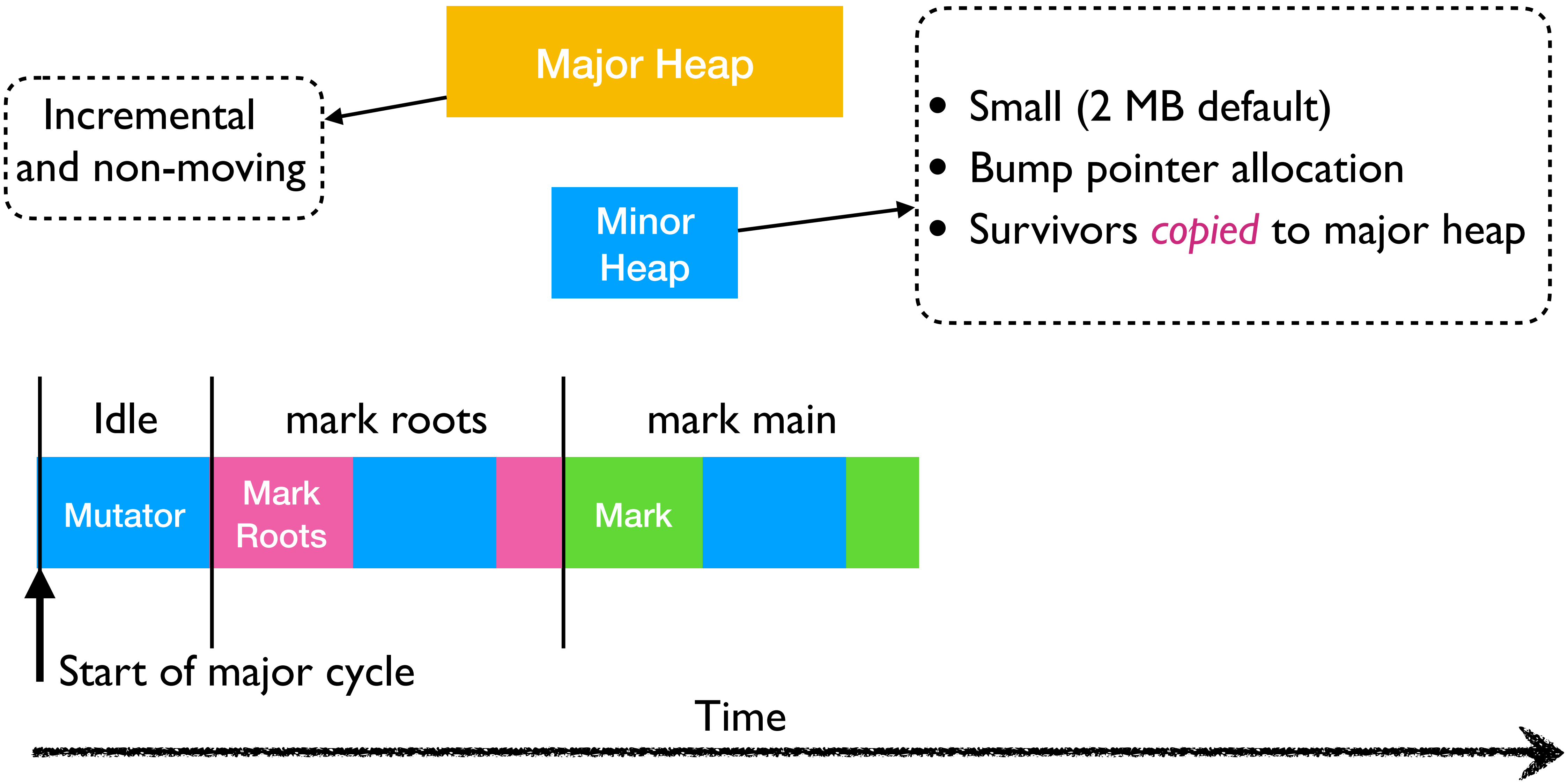
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



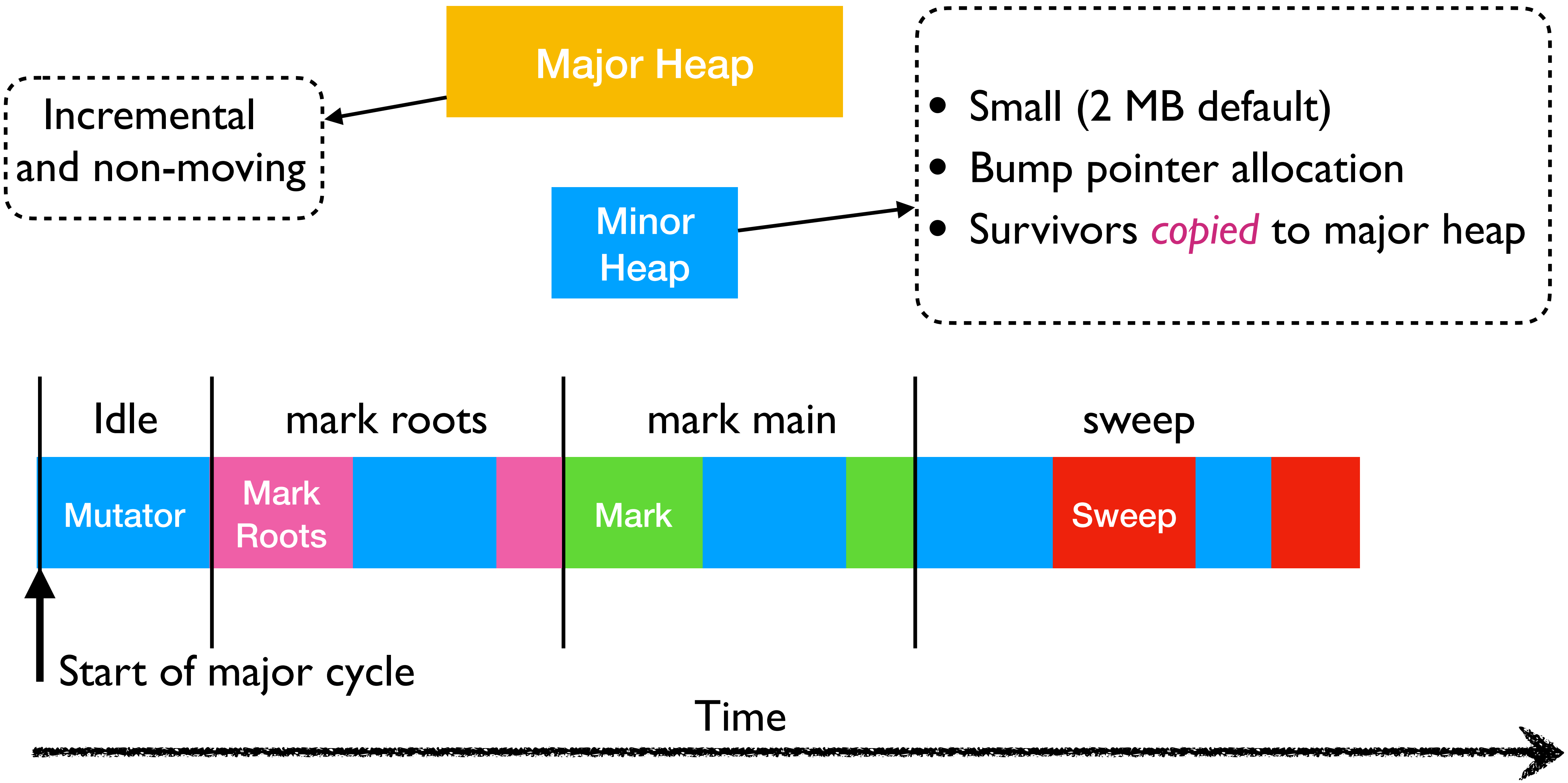
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



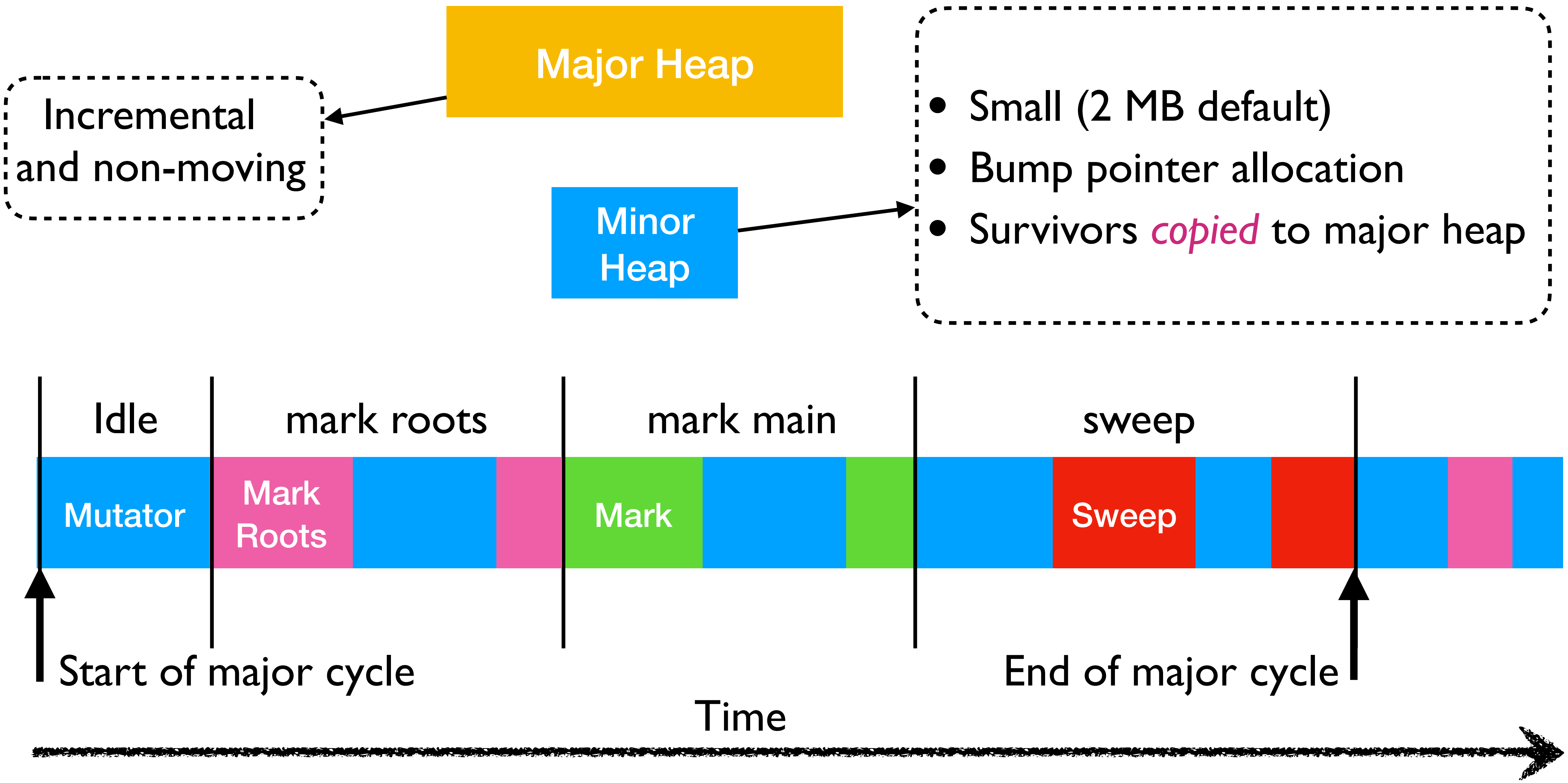
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



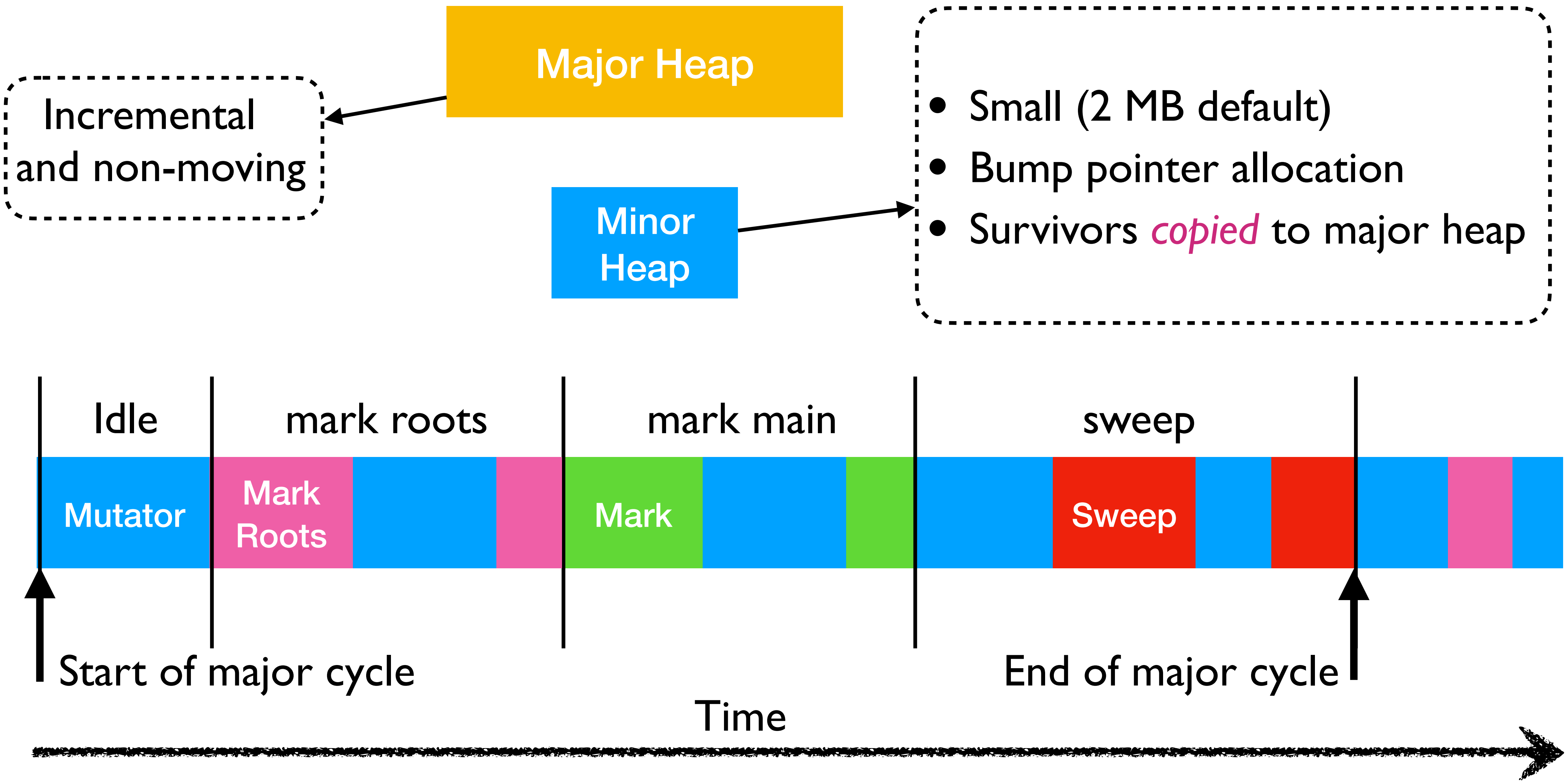
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



Stock OCaml GC

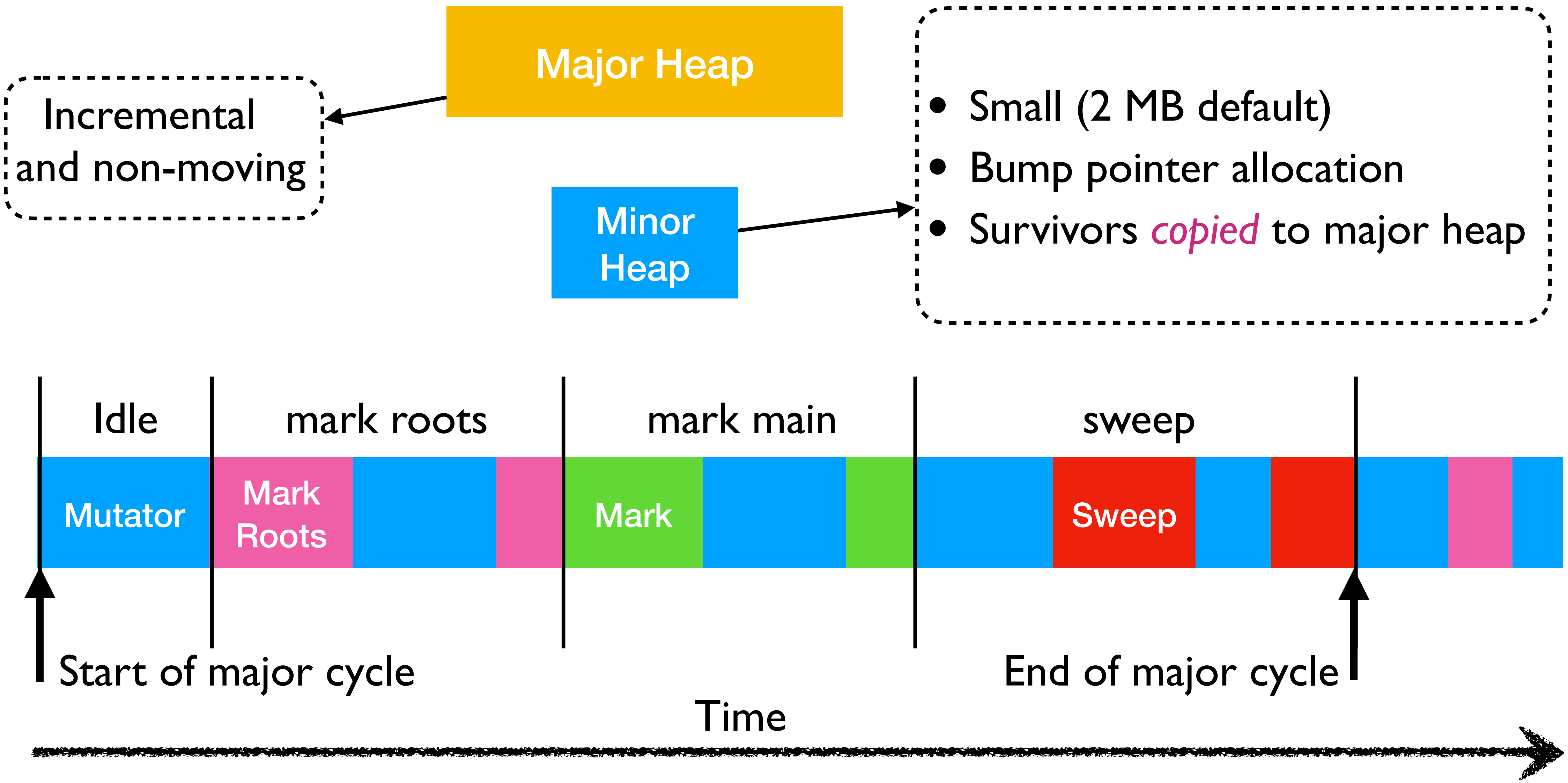
- A generational, non-moving, incremental, mark-and-sweep GC



- Fast allocations

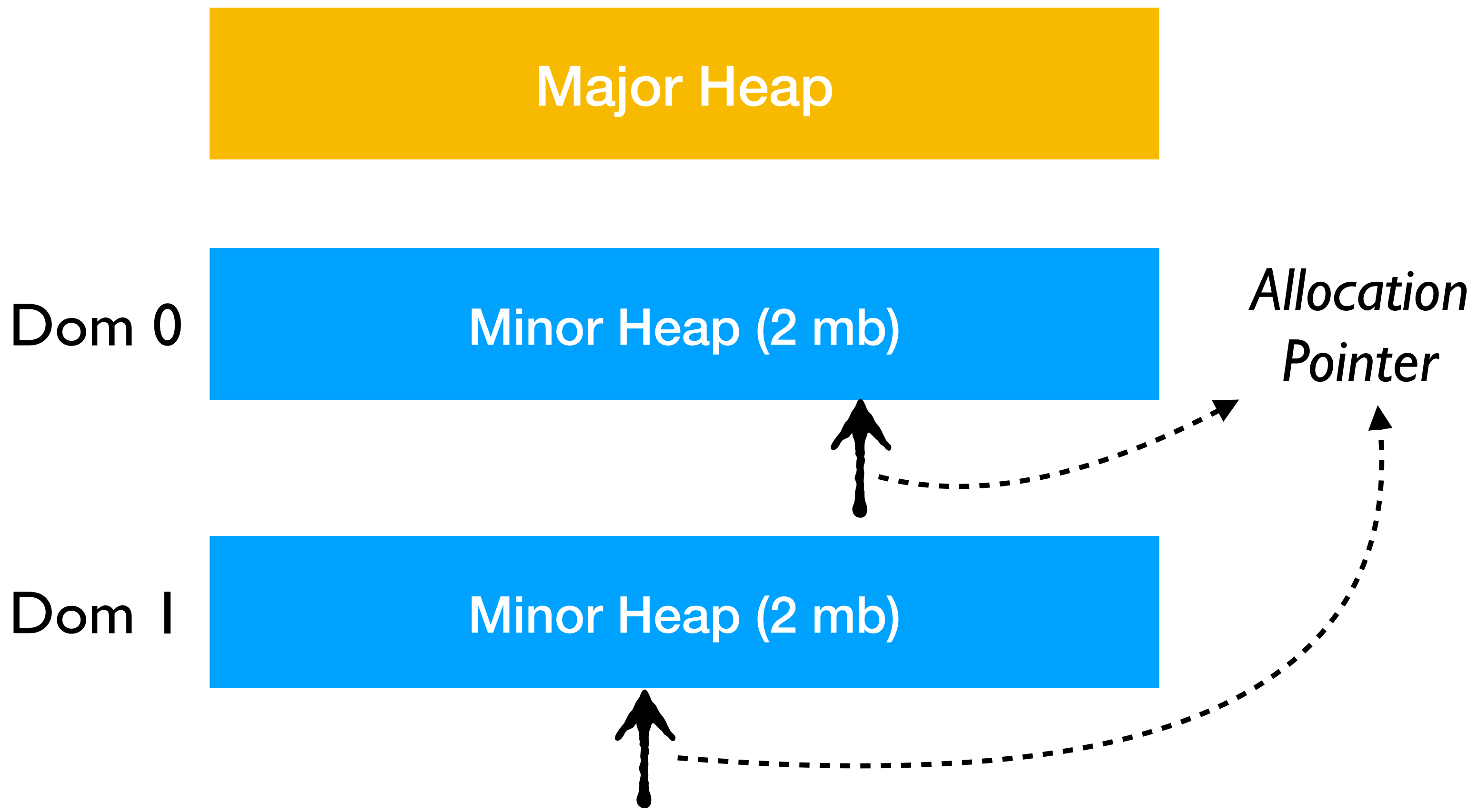
Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC



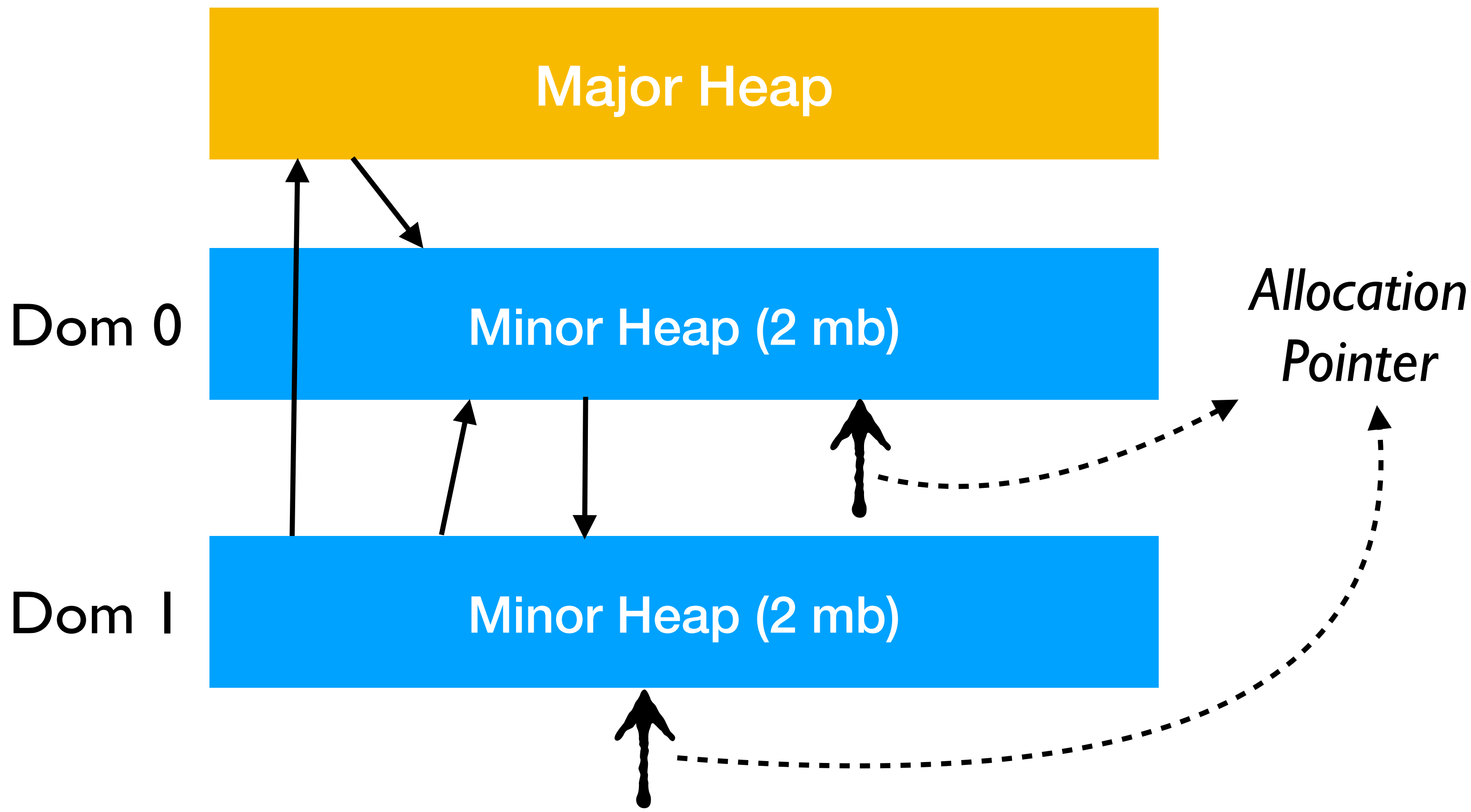
- Fast allocations
- Max GC latency **< 10 ms**, 99th percentile latency **< 1 ms**

Multicore OCaml: Minor GC



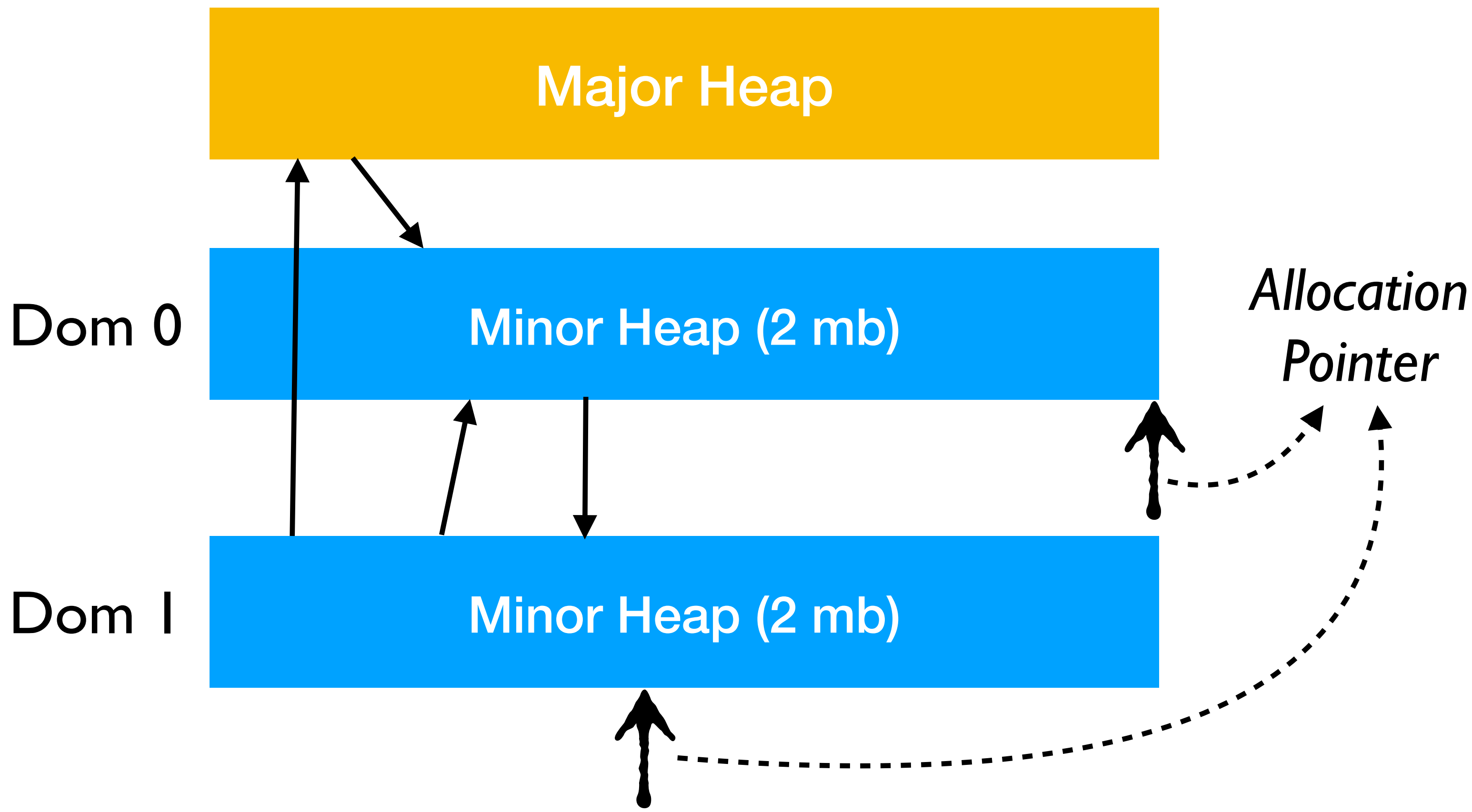
- Private minor heap arenas per domain
  - ✦ Fast allocations without synchronisation

Multicore OCaml: Minor GC



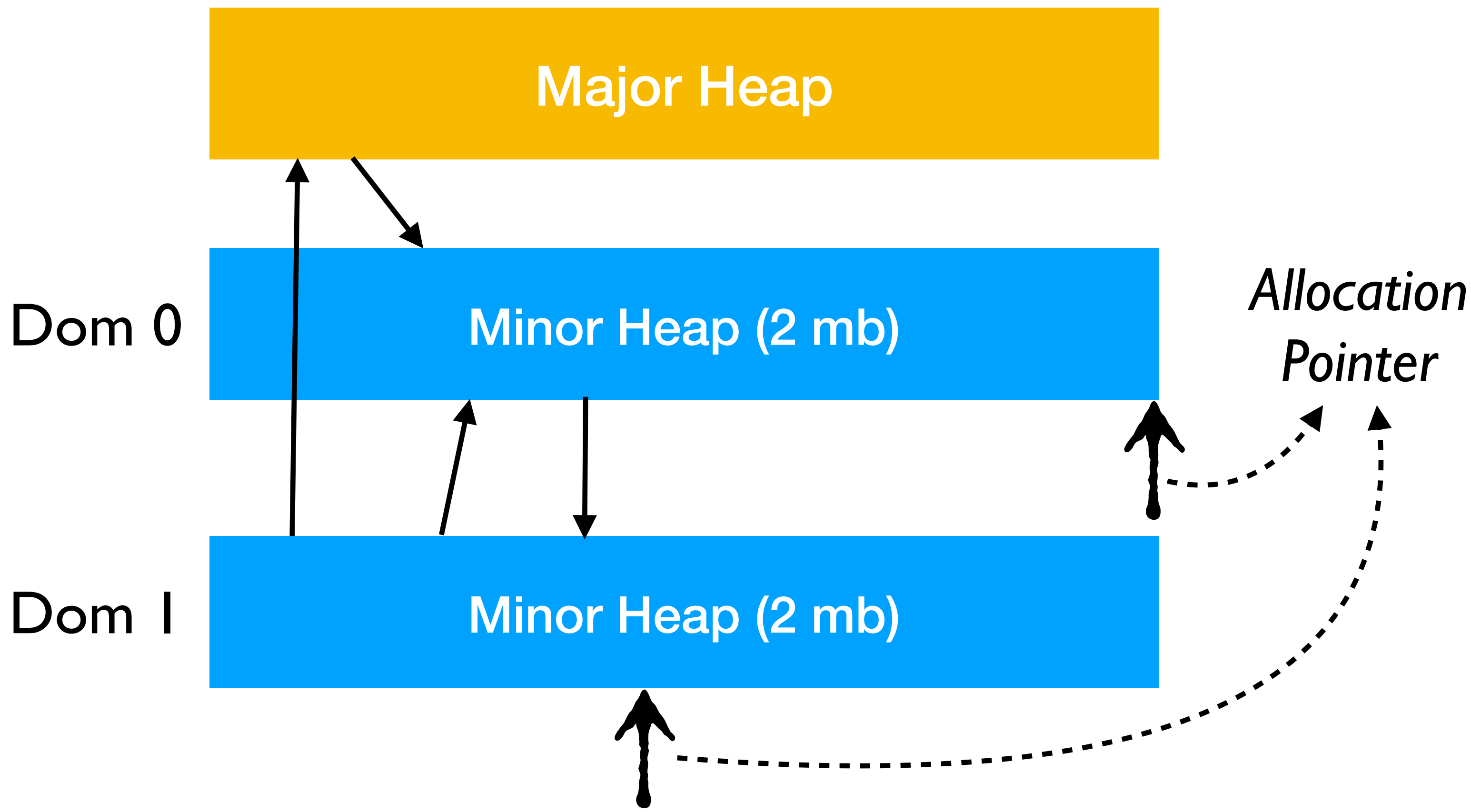
- Private minor heap arenas per domain
  - ✦ Fast allocations without synchronisation
- Pointers permitted between minor arenas and major heap

Multicore OCaml: Minor GC



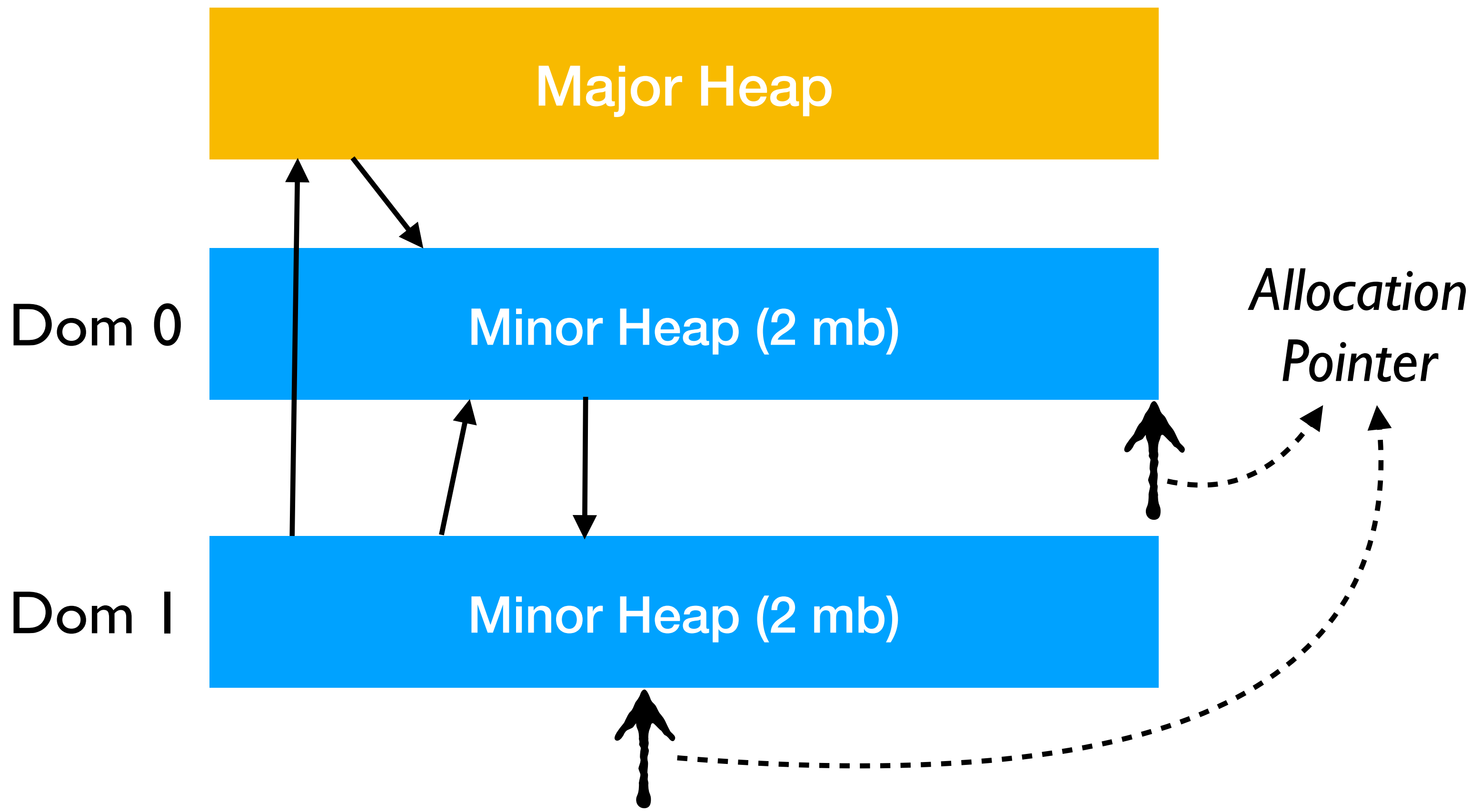
- *Stop-the-world parallel minor collection* when a domain runs out of minor heap
  - 2 global barriers / minor gc

Multicore OCaml: Minor GC



- *Stop-the-world parallel minor collection* when a domain runs out of minor heap
  - ✦ 2 global barriers / minor gc
- Bringing domains to a stop surprisingly fast
  - ✦ Poll points inserted into non-allocating loops

Multicore OCaml: Minor GC



- *Stop-the-world parallel minor collection* when a domain runs out of minor heap
  - ✦ 2 global barriers / minor gc
- Bringing domains to a stop surprisingly fast
  - ✦ Poll points inserted into non-allocating loops
- On 24 cores, ~10 ms pauses for completing stop-the-world minor GC

Multicore OCaml: Allocator

- Multicore-aware allocator for major heap
  - ✦ Based on Streamflow [Schneider et al. 2006]

Multicore OCaml: Allocator

- Multicore-aware allocator for major heap
  - Based on Streamflow [Schneider et al. 2006]
- *Thread-local, size-segmented free lists* for small objects
  - Most allocations in OCaml are small (99% of objects < 5 words in size)
  - Malloc for large allocations

Multicore OCaml: Allocator

- Multicore-aware allocator for major heap
  - Based on Streamflow [Schneider et al. 2006]
- *Thread-local, size-segmented free lists* for small objects
  - Most allocations in OCaml are small (99% of objects < 5 words in size)
  - Malloc for large allocations
- Most allocations do not need synchronisation

Multicore OCaml: Allocator

- Multicore-aware allocator for major heap
  - ✦ Based on Streamflow [Schneider et al. 2006]
- *Thread-local, size-segmented free lists* for small objects
  - ✦ Most allocations in OCaml are small (99% of objects < 5 words in size)
  - ✦ Malloc for large allocations
- Most allocations do not need synchronisation
- Sequential performance on par with OCaml's recent *best-fit* allocator

Multicore OCaml: Major GC

- *Mostly-concurrent mark-and-sweep* for major collection
  - ✦ Based on VCGC [Huelsbergen and Winterbottom 1998]

Multicore OCaml: Major GC

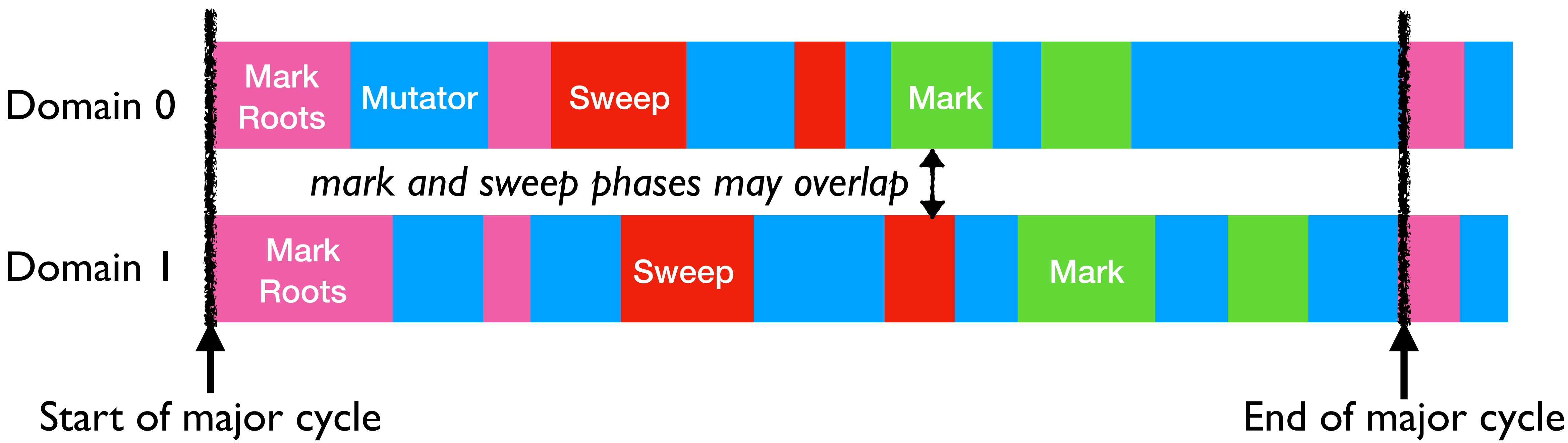
- *Mostly-concurrent mark-and-sweep* for major collection
  - Based on VCGC [Huelsbergen and Winterbottom 1998]
- GC colours Unmarked Marked Garbage Free

Multicore OCaml: Major GC

- *Mostly-concurrent mark-and-sweep* for major collection
    - Based on VCGC [Huelsbergen and Winterbottom 1998]
- GC colours Unmarked Marked Garbage Free
- Domains alternate between running the mutator and the GC

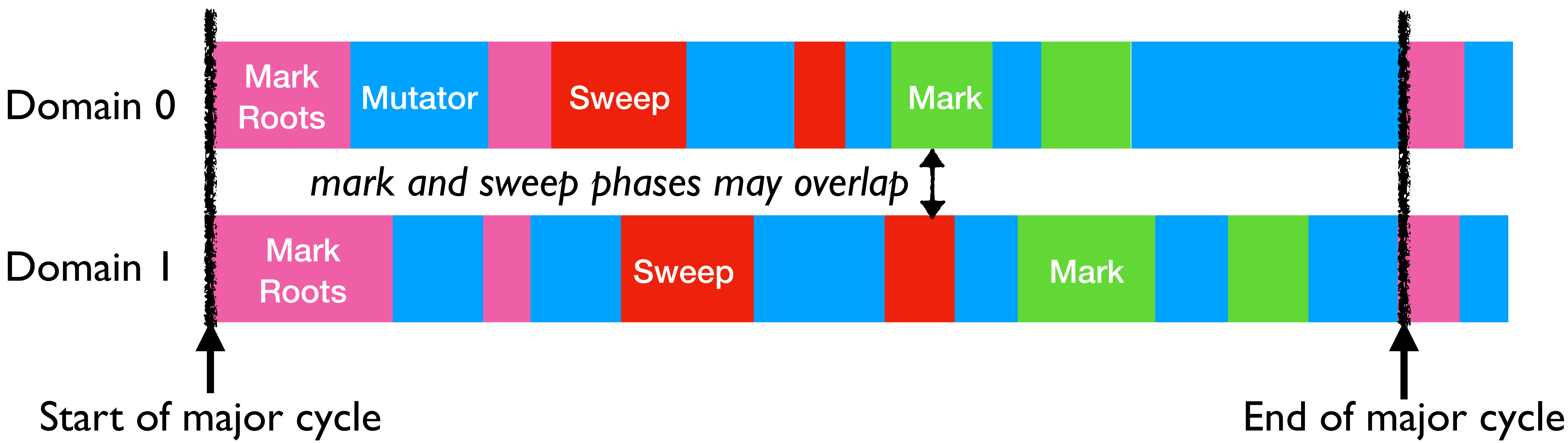
Multicore OCaml: Major GC

- *Mostly-concurrent mark-and-sweep* for major collection
  - ✦ Based on VCGC [Huelsbergen and Winterbottom 1998]
- GC colours **Unmarked** **Marked** **Garbage** **Free**
- Domains alternate between running the mutator and the GC



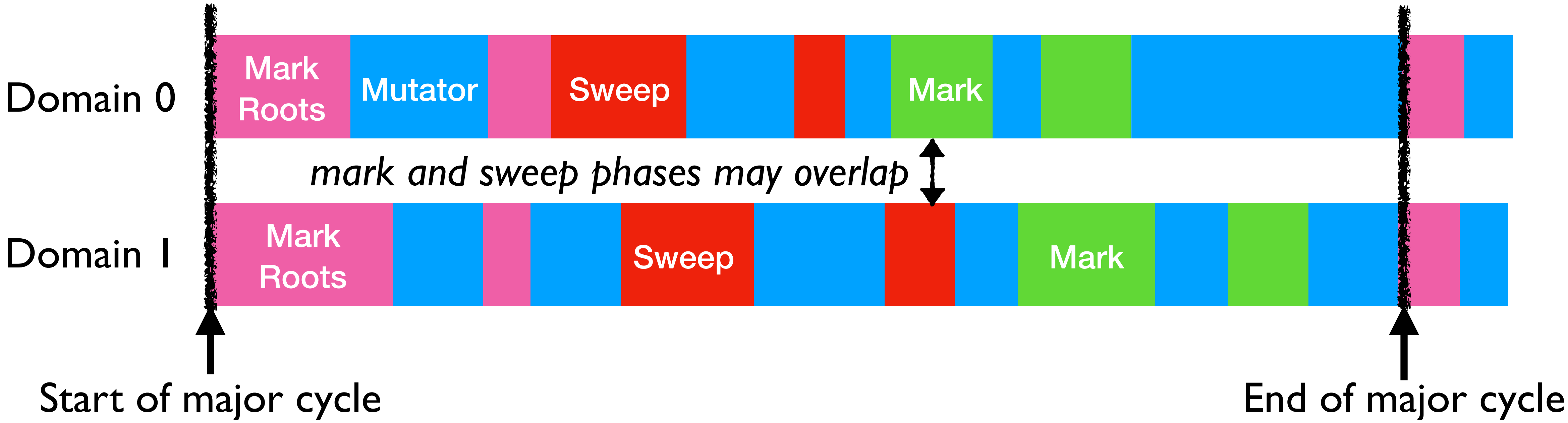
Multicore OCaml: Major GC

- *Mostly-concurrent mark-and-sweep* for major collection
  - ✦ Based on VCGC [Huelsbergen and Winterbottom 1998]
- GC colours: Unmarked, Marked, Garbage, Free
- Domains alternate between running the mutator and the GC



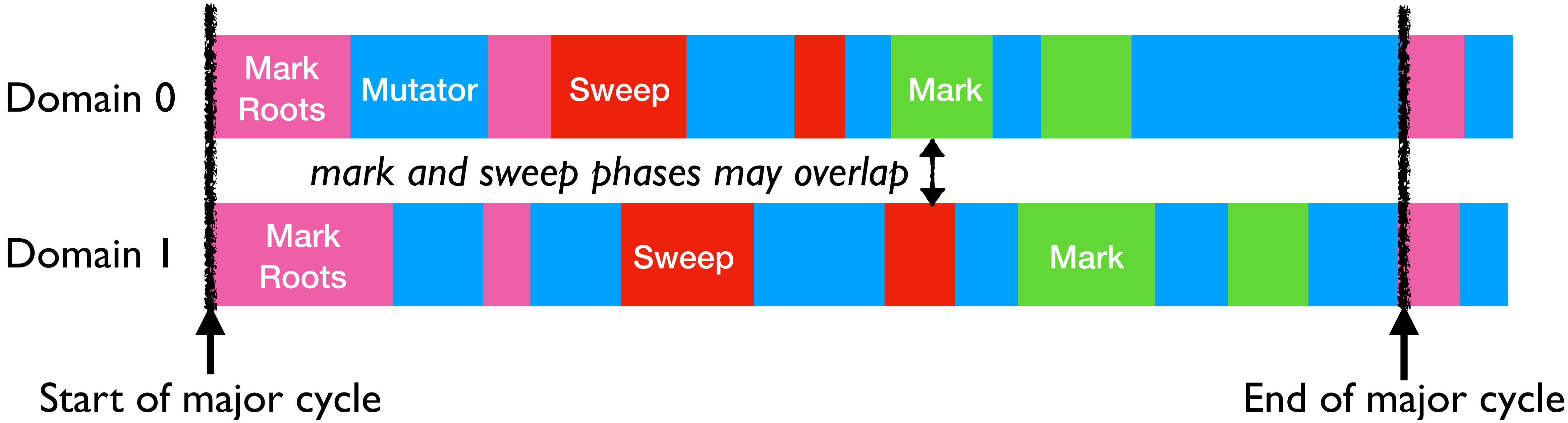
- Marking: Unmarked → Marked Sweeping: Garbage → Free
  - ✦ Marking is *racy* but uses plain *non-atomic stores and idempotent*

Multicore OCaml: Major GC

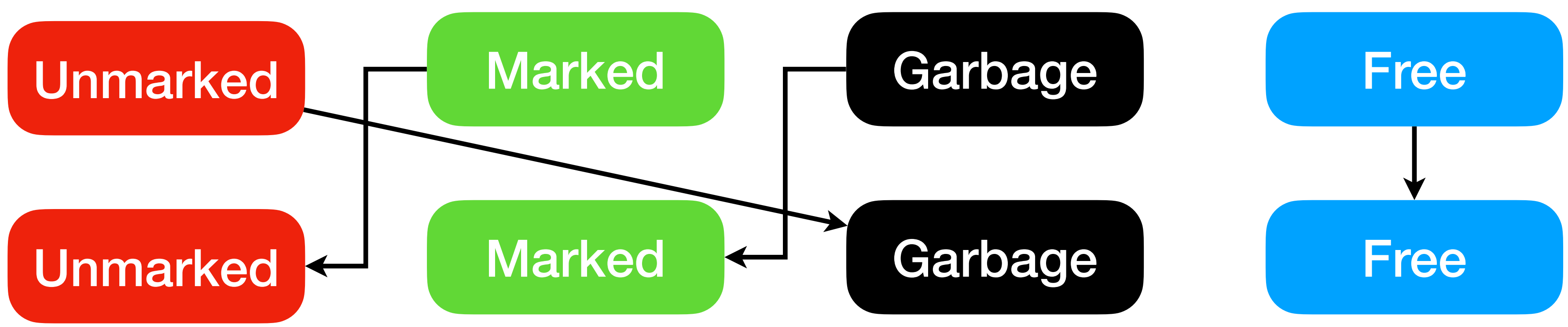


- Marking & sweeping done ⇒ stop-the-world

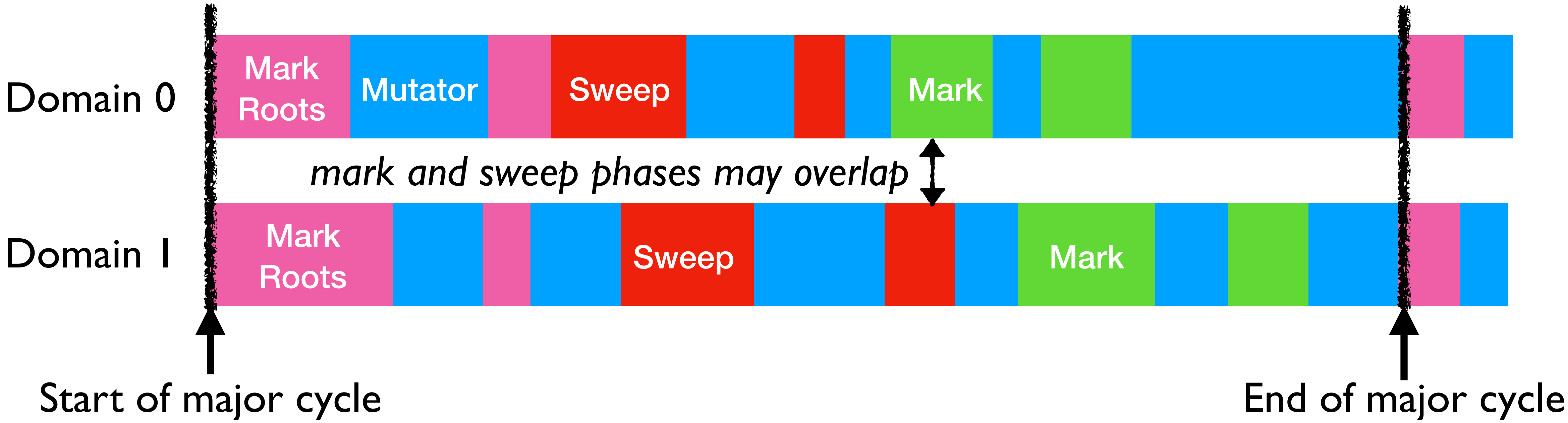
Multicore OCaml: Major GC



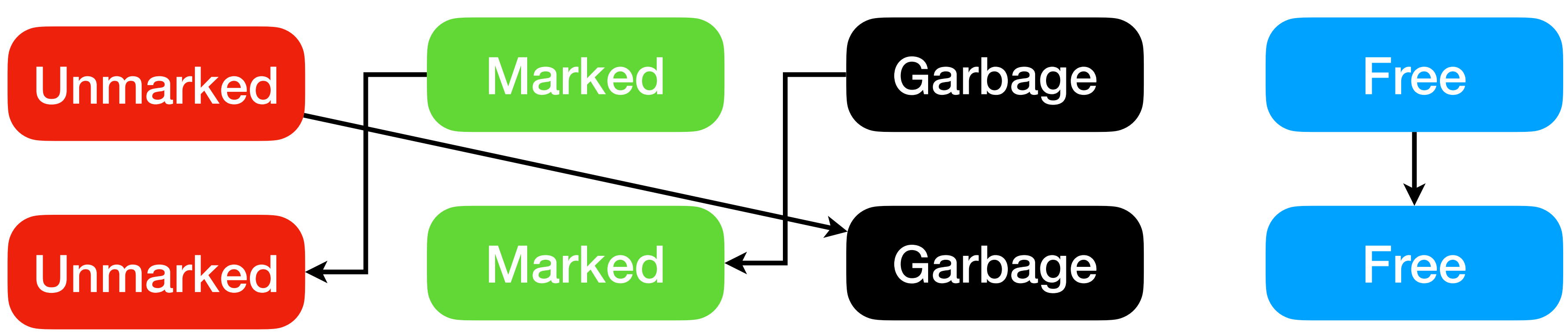
- Marking & sweeping done ⇒ stop-the-world



Multicore OCaml: Major GC

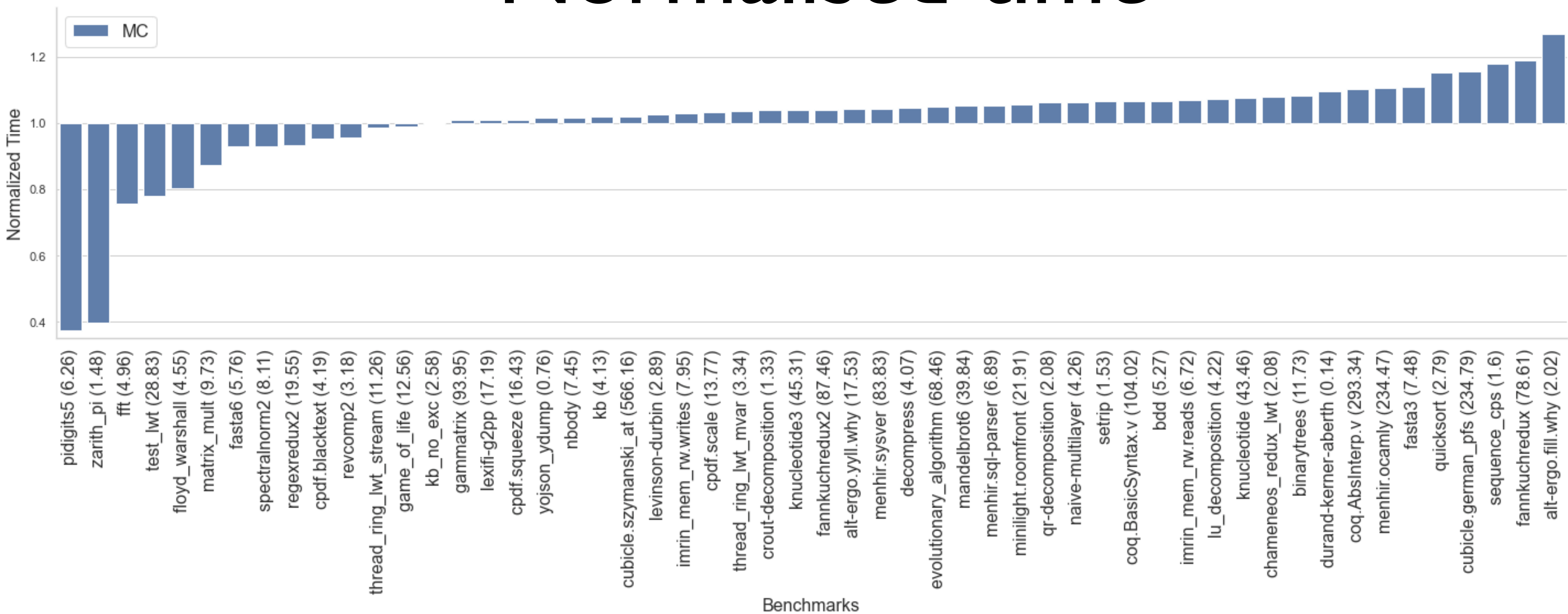


- Marking & sweeping done ⇒ stop-the-world

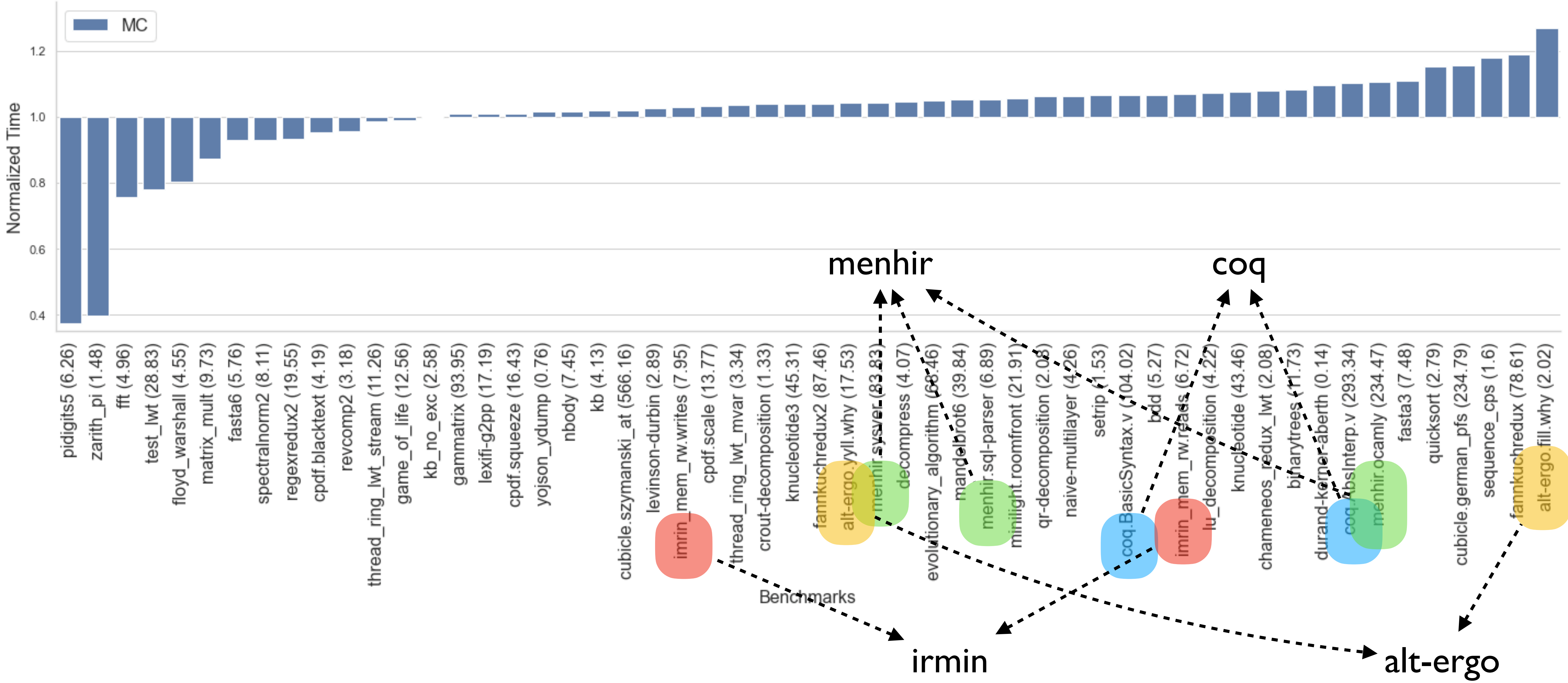


- Stop-the-world pauses are ~5 ms on 24 cores

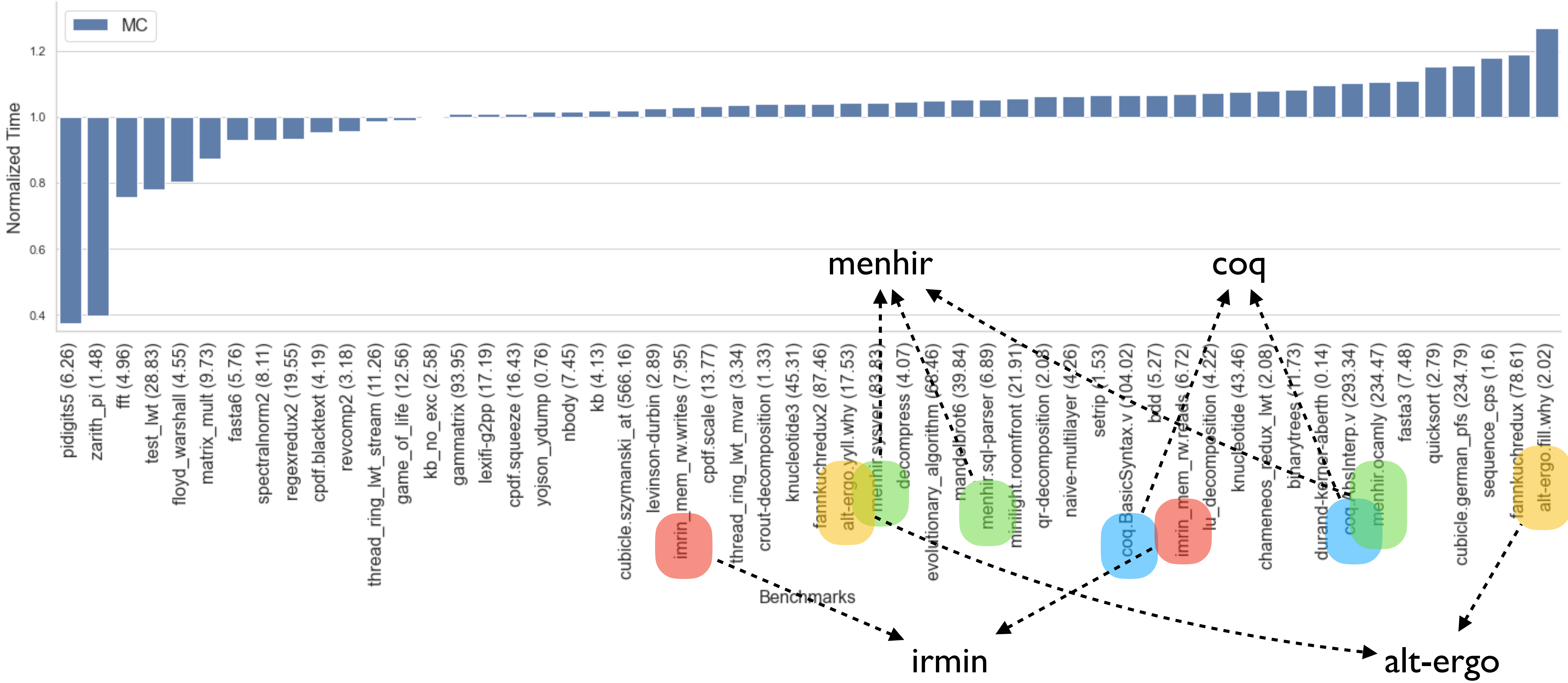
Sequential Performance: Normalised time



Sequential Performance: Normalised time

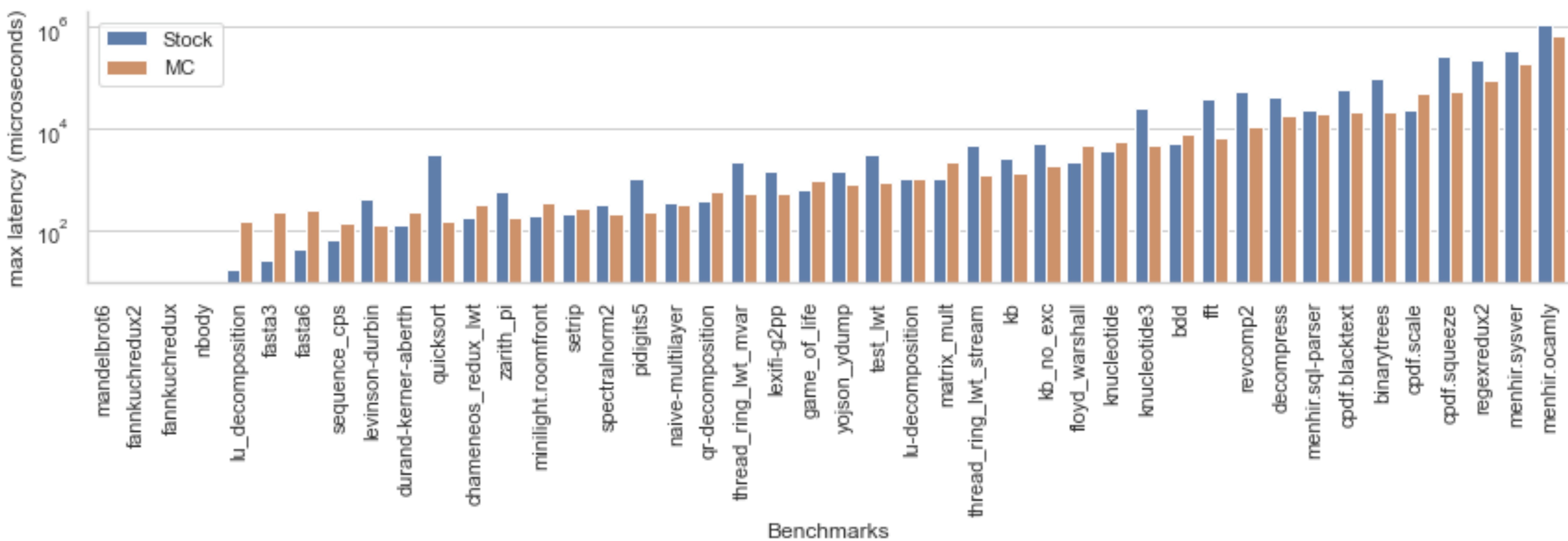


Sequential Performance: Normalised time

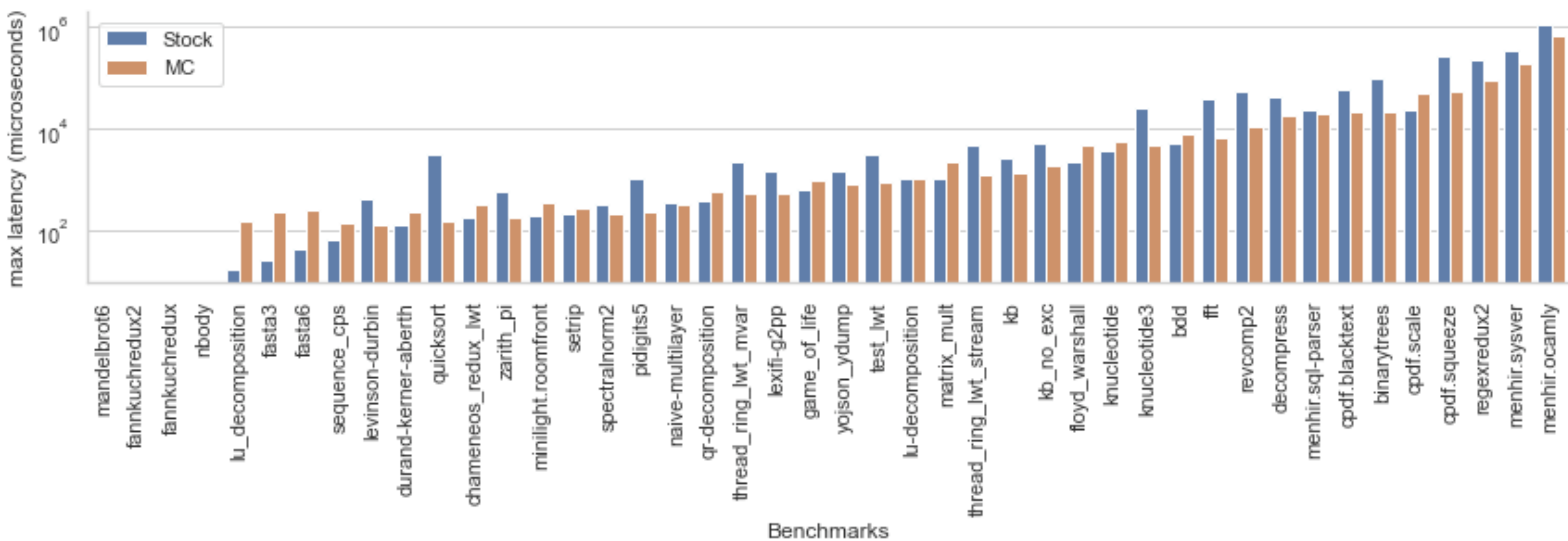


- ~1% faster than stock (geomean of normalised running times)
  - Difference under measurement noise mostly
  - Outliers due to difference in allocators

Sequential Performance: Max pause time



Sequential Performance: Max pause time



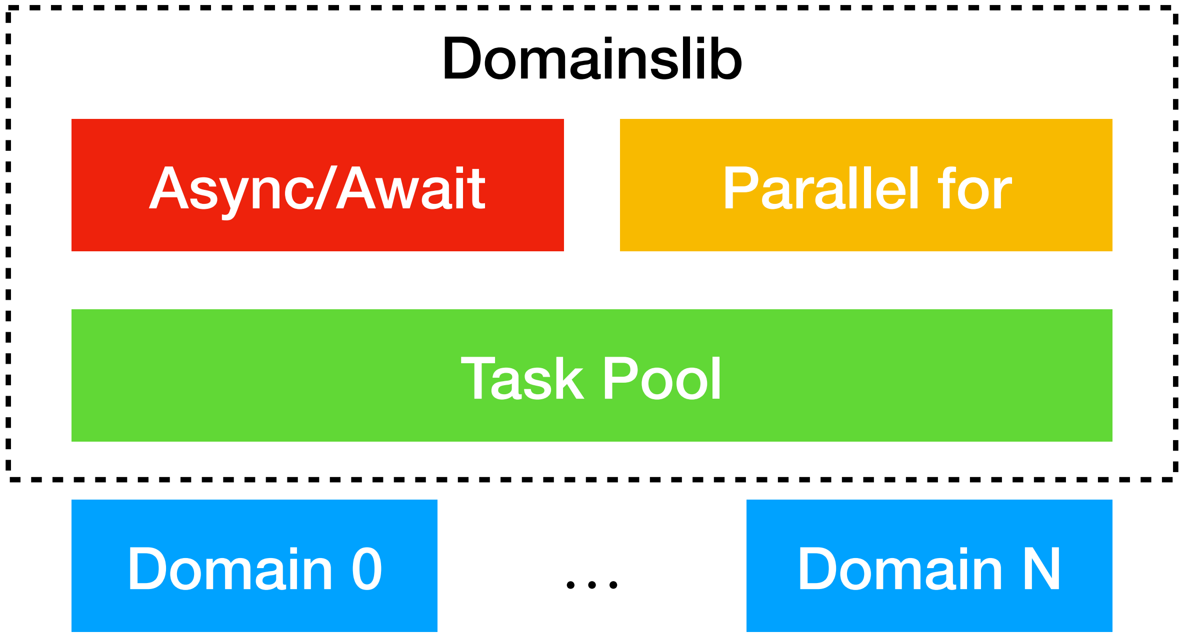
- *Pausetimes are lower under Multicore OCaml than stock OCaml*

Domainslib for parallel programming

- Domain API exposed by the compiler is too low-level
  - Mutex, condition variables, …

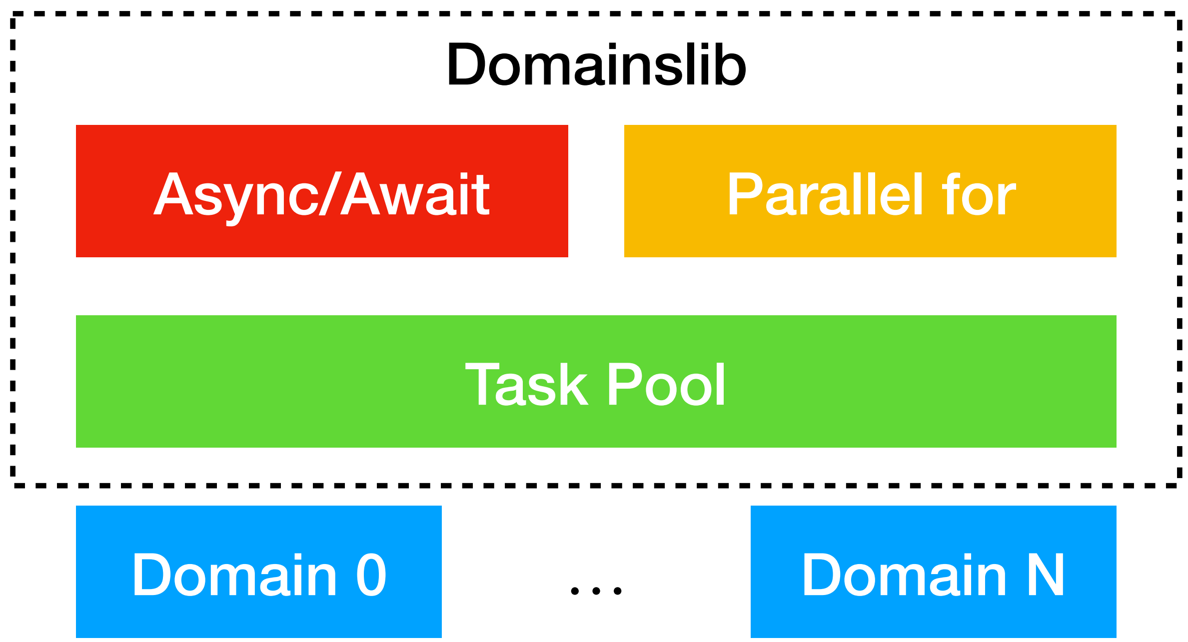
Domainslib for parallel programming

- Domain API exposed by the compiler is too low-level
  - Mutex, condition variables, …
- Domainslib - *https://github.com/ocaml-multicore/domainslib*



Domainslib for parallel programming

- Domain API exposed by the compiler is too low-level
  - ✦ Mutex, condition variables, …
- Domainslib - *https://github.com/ocaml-multicore/domainslib*



Let's look at examples!

Recursive Fibonacci - Sequential

```
let rec fib n =
  if n < 2 then 1
  else fib (n-1) + fib (n-2)
```

Recursive Fibonacci - Parallel

```
module T = Domainslib.Task
```

```
let fib n =
  let pool = T.setup_pool
    ~num_additional_domains:(num_domains - 1) in
  let res = fib_par pool n in
  T.teardown_pool pool;
  res
```

Recursive Fibonacci - Parallel

```
module T = Domainslib.Task




let rec fib_par pool n =
  if n <= 40 then fib_seq n
  else
    let a = T.async pool (fun _ -> fib_par pool (n-1)) in
    let b = T.async pool (fun _ -> fib_par pool (n-2)) in
    T.await pool a + T.await pool b

let fib n =
  let pool = T.setup_pool
    ~num_additional_domains:(num_domains - 1) in
  let res = fib_par pool n in
  T.teardown_pool pool;
  res
```

Recursive Fibonacci - Parallel

```
module T = Domainslib.Task

let rec fib_seq n =
  if n < 2 then 1
  else fib_seq (n-1) + fib_seq (n-2)

let rec fib_par pool n =
  if n <= 40 then fib_seq n
  else
    let a = T.async pool (fun _ -> fib_par pool (n-1)) in
    let b = T.async pool (fun _ -> fib_par pool (n-2)) in
    T.await pool a + T.await pool b

let fib n =
  let pool = T.setup_pool
    ~num_additional_domains:(num_domains - 1) in
  let res = fib_par pool n in
  T.teardown_pool pool;
  res
```

Performance: fib(48)

Cores	Time (Seconds)	Vs Serial	Vs Self
1	37.787	0.98	1
2	19.034	1.94	1.99
4	9.723	3.8	3.89
8	5.023	7.36	7.52
16	2.914	12.68	12.97
24	2.201	**16.79**	**17.17**

Conway's Game of Life

Conway's Game of Life

Conway's Game of Life

```
let next () =
  ...
  for x = 0 to board_size - 1 do
    for y = 0 to board_size - 1 do
      next_board.(x).(y) <- next_cell cur_board x y
    done
  done;
  ...
```

Conway's Game of Life

```
let next () =
  ...
  for x = 0 to board_size - 1 do
    for y = 0 to board_size - 1 do
      next_board.(x).(y) <- next_cell cur_board x y
    done
  done;
  ...

let next () =
  ...
  T.parallel_for pool ~start:0 ~finish:(board_size - 1)
    ~body:(fun x ->
       for y = 0 to board_size - 1 do
         next_board.(x).(y) <- next_cell cur_board x y
       done);
  ...
```

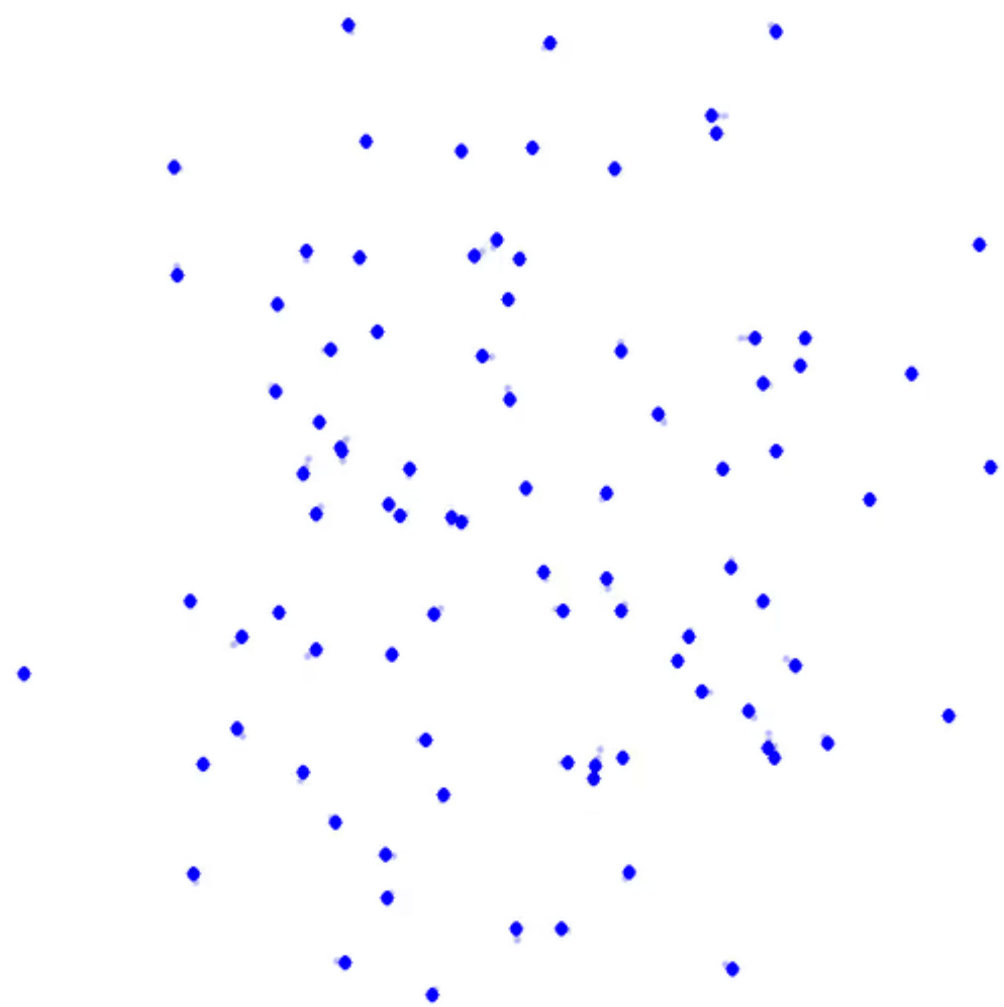
Performance: Game of Life

Board size = 1024, Iterations = 512

Cores	Time (Seconds)	Vs Serial	Vs Self
1	24.326	1	1
2	12.290	1.980	1.98
4	6.260	3.890	3.89
8	3.238	7.51	7.51
16	1.726	14.09	14.09
24	1.212	**20.07**	**20.07**

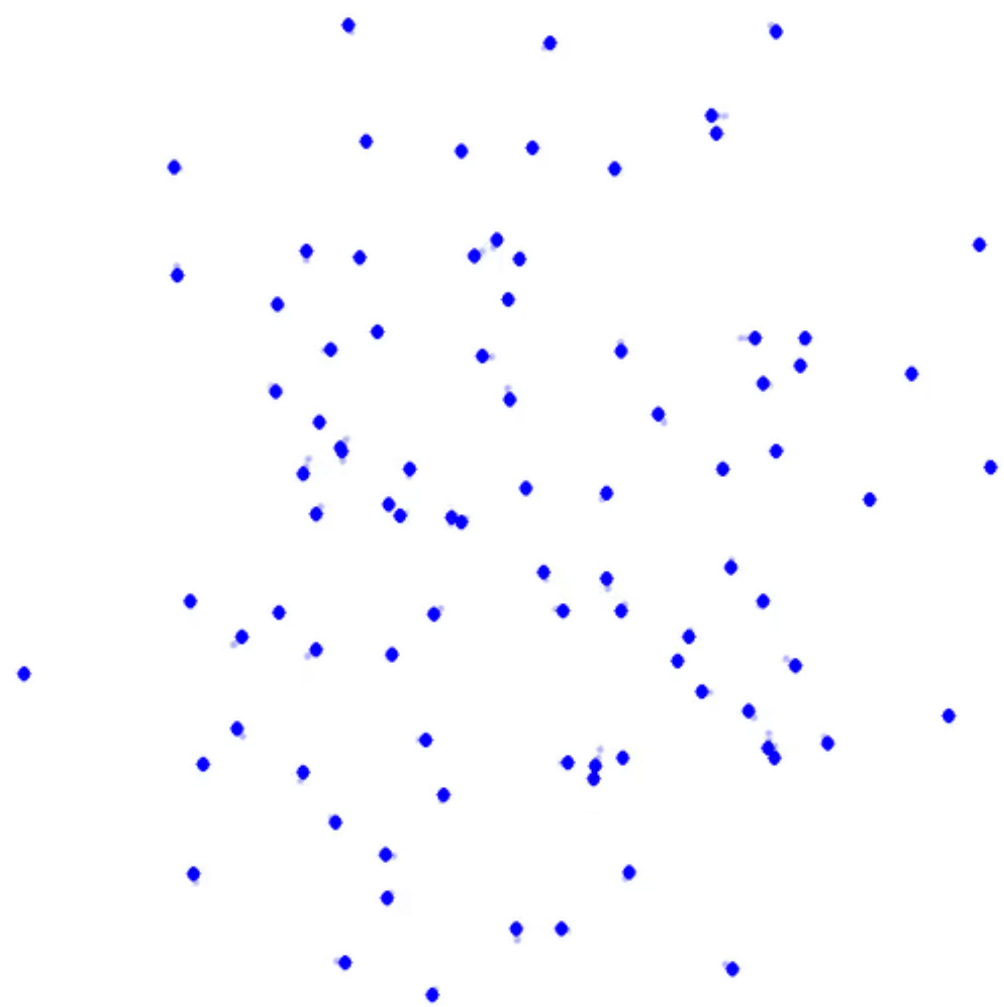
Performance Hacking: N-body

- Simulates the orbits of large number of astronomical objects
- Taken from computer language benchmarks game



Performance Hacking: N-body

- Simulates the orbits of large number of astronomical objects
- Taken from computer language benchmarks game



Sequential N-body

```
let advance bodies dt =
  for i = 0 to Array.length bodies - 1 do
    let b = bodies.(i) in
    for j = 0 to Array.length bodies - 1 do
      let b' = bodies.(j) in
      if (i!=j) then begin
        let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
        let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
        let mag = dt /. (dist2 *. sqrt(dist2)) in
        b.vx <- b.vx -. dx *. b'.mass *. mag;
        b.vy <- b.vy -. dy *. b'.mass *. mag;
        b.vz <- b.vz -. dz *. b'.mass *. mag;
      end
    done
  done;
```

Sequential N-body

```
let advance bodies dt =
  for i = 0 to Array.length bodies - 1 do
    let b = bodies.(i) in
    for j = 0 to Array.length bodies - 1 do
      let b' = bodies.(j) in
      if (i!=j) then begin
        let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
        let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
        let mag = dt /. (dist2 *. sqrt(dist2)) in
        b.vx <- b.vx -. dx *. b'.mass *. mag;
        b.vy <- b.vy -. dy *. b'.mass *. mag;
        b.vz <- b.vz -. dz *. b'.mass *. mag;
      end
    done
  done;
```

```
$ perf record ./nbody.exe
$ perf report
```

Sequential N-body

```
let advance bodies dt =
  for i = 0 to Array.length bodies - 1 do
    let b = bodies.(i) in
    for j = 0 to Array.length bodies - 1 do
      let b' = bodies.(j) in
      if (i!=j) then begin
        let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
        let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
        let mag = dt /. (dist2 *. sqrt(dist2)) in
        b.vx <- b.vx -. dx *. b'.mass *. mag;
        b.vy <- b.vy -. dy *. b'.mass *. mag;
        b.vz <- b.vz -. dz *. b'.mass *. mag;
      end
    done
  done;
```

```
$ perf record ./nbody.exe
$ perf report
```

```
Samples: 17K of event 'cycles:uppp', Event count (approx.): 12363186877
Overhead  Command    Shared Object      Symbol
  99.87%  nbody.exe  nbody.exe          [.] camlDune__exe__Nbody__advance_230
   0.12%  nbody.exe  nbody.exe          [.] camlDune__exe__Nbody__energy_246
   0.00%  nbody.exe  nbody.exe          [.] camlStdlib__domain__get_355
   0.00%  nbody.exe  [kernel]           [k] 0xffffffffb44009e7
```

Parallel N-Body

```
let advance pool bodies dt =
  T.parallel_for pool
    ~start:0
    ~finish:(num_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      for j = 0 to Array.length bodies - 1 do
        let b' = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          b.vx <- b.vx -. dx *. b'.mass *. mag;
          b.vy <- b.vy -. dy *. b'.mass *. mag;
          b.vz <- b.vz -. dz *. b'.mass *. mag;
        end
      done);
```

Parallel N-Body

```
let advance pool bodies dt =
  T.parallel_for pool
    ~start:0
    ~finish:(num_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      for j = 0 to Array.length bodies - 1 do
        let b' = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          b.vx <- b.vx -. dx *. b'.mass *. mag;
          b.vy <- b.vy -. dy *. b'.mass *. mag;
          b.vz <- b.vz -. dz *. b'.mass *. mag;
        end
      done);
```

- ~5X speedup on 8 cores compared to sequential version.

Parallel N-Body

```
let advance pool bodies dt =
  T.parallel_for pool
    ~start:0
    ~finish:(num_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      for j = 0 to Array.length bodies - 1 do
        let b' = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          b.vx <- b.vx -. dx *. b'.mass *. mag;
          b.vy <- b.vy -. dy *. b'.mass *. mag;
          b.vz <- b.vz -. dz *. b'.mass *. mag;
        end
      done);
```

- ~5X speedup on 8 cores compared to sequential version.
- *Can we do better?*

Parallel N-Body

```
let advance pool bodies dt =
  T.parallel_for pool
    ~start:0
    ~finish:(num_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      for j = 0 to Array.length bodies - 1 do
        let b' = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          b.vx <- b.vx -. dx *. b'.mass *. mag;
          b.vy <- b.vy -. dy *. b'.mass *. mag;
          b.vz <- b.vz -. dz *. b'.mass *. mag;
        end
      done);
```

- ~5X speedup on 8 cores compared to sequential version.
- *Can we do better?*
- All the domains writing to the same shared array *bodies* in the inner loop
  - Leads to poor cache behaviour

Parallel N-Body (cache friendly)

```
let advance pool bodies dt =
  T.parallel_for pool
    ~start:0
    ~finish:(num_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      let vx, vy, vz = ref b.vx, ref b.vy, ref b.vz in
      for j = 0 to Array.length bodies - 1 do
        let b' = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          vx := !vx -. dx *. b'.mass *. mag;
          vy := !vy -. dy *. b'.mass *. mag;
          vz := !vz -. dz *. b'.mass *. mag;
        end
      done;
      b.vx <- !vx;
      b.vy <- !vy;
      b.vz <- !vz);
```

Parallel N-Body (cache friendly)

```
let advance pool bodies dt =
  T.parallel_for pool
    ~start:0
    ~finish:(num_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      let vx, vy, vz = ref b.vx, ref b.vy, ref b.vz in
      for j = 0 to Array.length bodies - 1 do
        let b' = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b'.x  and dy = b.y -. b'.y  and dz = b.z -. b'.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          vx := !vx -. dx *. b'.mass *. mag;
          vy := !vy -. dy *. b'.mass *. mag;
          vz := !vz -. dz *. b'.mass *. mag;
        end
      done;
      b.vx <- !vx;
      b.vy <- !vy;
      b.vz <- !vz);
```

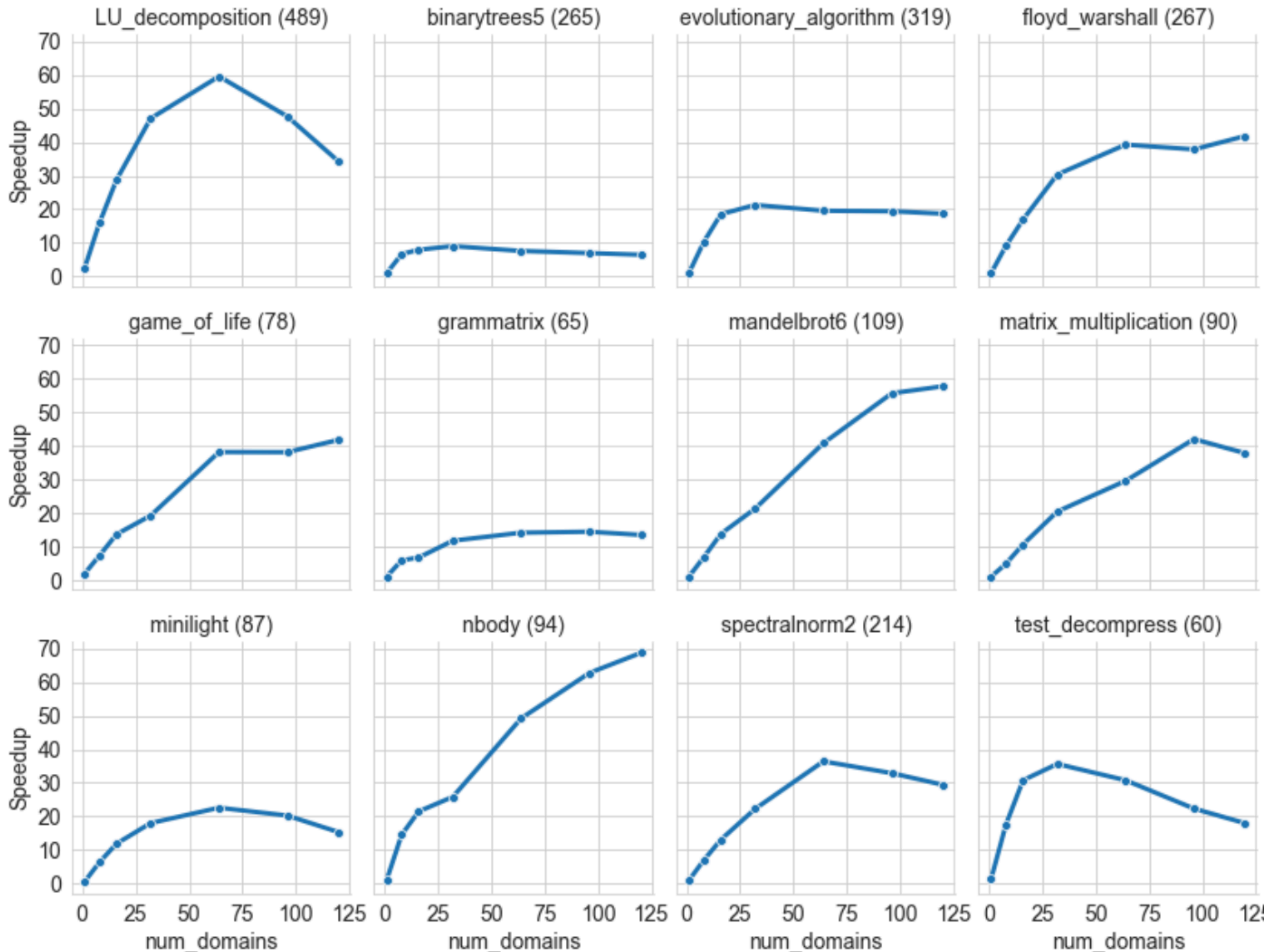
- 22% faster than the unoptimised version

Parallel N-Body (cache friendly)

```
kc@kaveri:~/temp/nbody$ perf stat -e cycles,cycle_activity.cycles_no_execute,cycle_activity.stalls_mem_any,
mem_load_uops_l3_miss_retired.remote_hitm ./_build/default/nbody_multicore.exe 8
10745700105645904.000000000
10745765317367518.000000000

 Performance counter stats for './_build/default/nbody_multicore.exe 8':

    16,919,777,849      cycles
     4,099,415,910      cycle_activity.cycles_no_execute
     3,864,941,761      cycle_activity.stalls_mem_any
         1,374,487      mem_load_uops_l3_miss_retired.remote_hitm

       0.861178677 seconds time elapsed

kc@kaveri:~/temp/nbody$ perf stat -e cycles,cycle_activity.cycles_no_execute,cycle_activity.stalls_mem_any,
mem_load_uops_l3_miss_retired.remote_hitm ./_build/default/nbody_multicore_optimised.exe 8
10745700105645904.000000000
10745765317367518.000000000

 Performance counter stats for './_build/default/nbody_multicore_optimised.exe 8':

    13,993,448,825      cycles
     2,926,815,349      cycle_activity.cycles_no_execute
     2,844,213,127      cycle_activity.stalls_mem_any
           409,388      mem_load_uops_l3_miss_retired.remote_hitm

       0.701971803 seconds time elapsed
```

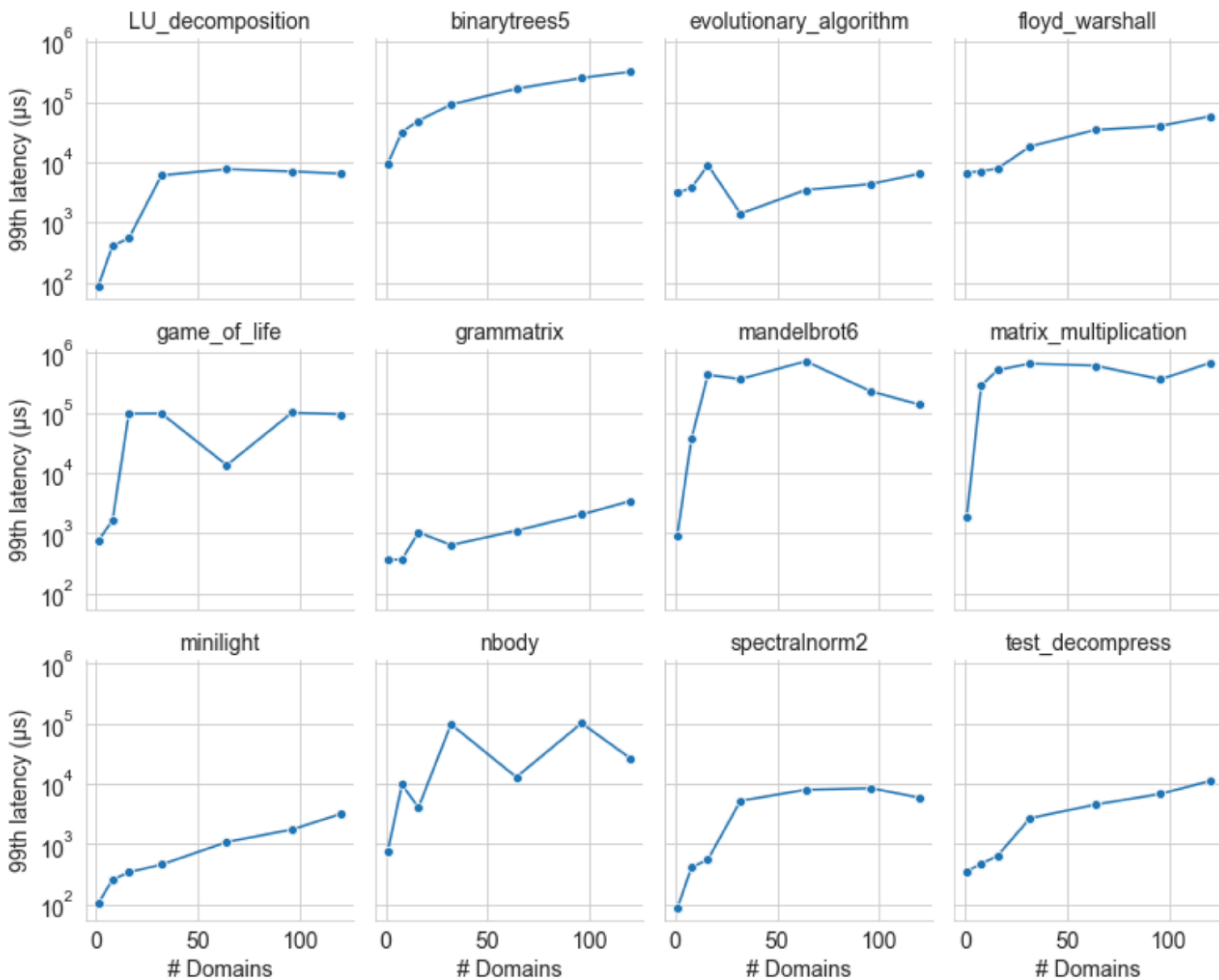
https://www.brendangregg.com/linuxperf.html

Parallel Scalability



On 4 x 32-core AMD EPYC 7551

Parallel Latency



Summary

- Multicore OCaml brings shared-memory parallelism to OCaml
  - A new concurrent GC that optimises for *latency* and retains *throughput*

Summary

- Multicore OCaml brings shared-memory parallelism to OCaml
  - A new concurrent GC that optimises for *latency* and retains *throughput*
- *Domainslib* for high-level parallel programming
  - Async/await & parallel for loops
  - Scales well to large number (128) of cores

Summary

- Multicore OCaml brings shared-memory parallelism to OCaml
  - A new concurrent GC that optimises for *latency* and retains *throughput*
- *Domainslib* for high-level parallel programming
  - Async/await & parallel for loops
  - Scales well to large number (128) of cores
- OCaml 5.0 will be x86-64 only
  - Arm64 and Power after the 5.0 release
  - Linux, MacOS and Windows (mingw-w64 only)

Summary

- Multicore OCaml brings shared-memory parallelism to OCaml
  - A new concurrent GC that optimises for *latency* and retains *throughput*
- *Domainslib* for high-level parallel programming
  - Async/await & parallel for loops
  - Scales well to large number (128) of cores
- OCaml 5.0 will be x86-64 only
  - Arm64 and Power after the 5.0 release
  - Linux, MacOS and Windows (mingw-w64 only)
- Sivaramakrishnan et al, "*Retrofitting Parallelism onto OCaml*", ICFP 2020