Neptune: A Safe, Parallel Garbage Collector for Julia

Mehmet Emre & Michael Christensen

UCSB CS263 Runtime Systems - In-Class Presentation
May 31, 2017

Objective: Parallel GC in Julia

Julia

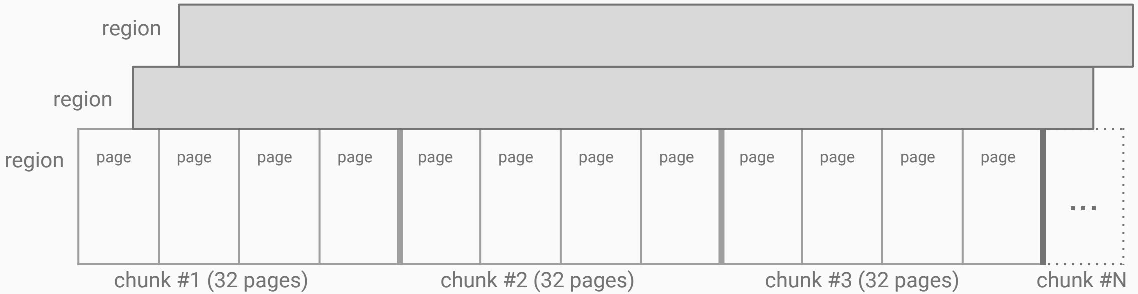
- A scientific programming language
  - Aims for **end-to-end performance** more than latency
- Dynamically typed
- Function-level JIT compilation via LLVM
- Uses stable numeric computation libraries written in FORTRAN

Julia's GC (i.e. the **problem** to solve)

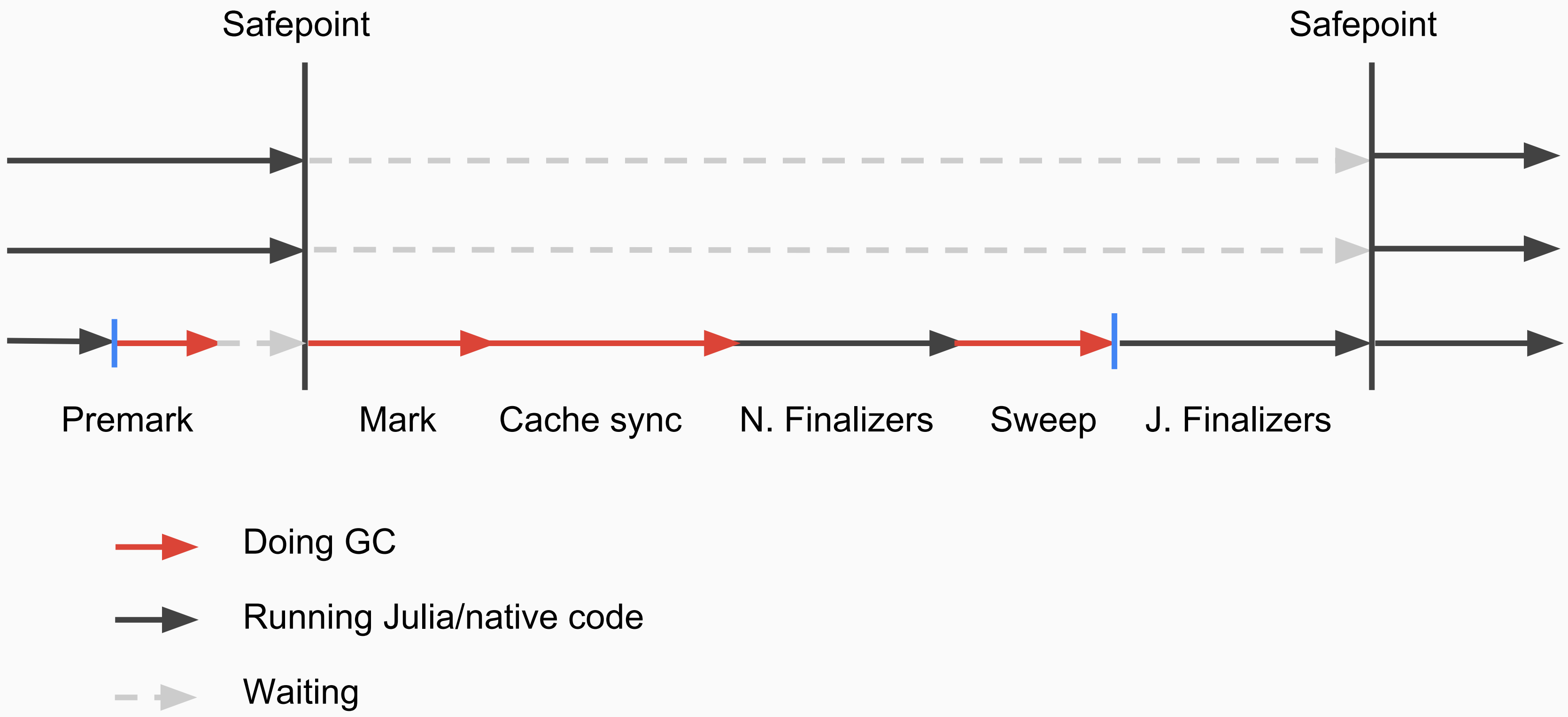
- Mark & Sweep
  - no copying/compacting
- Very little concurrency
  - Mostly single-threaded
  - Stop-the-world
  - Some pre-mark work concurrently
- Generational
  - 2 generations
  - Promotion: when a reachable object survives more than PROMOTE_AGE collections
  - “remembered sets”
- Quick vs. Full
- Small objects allocated from thread-local object pools
  - Categorized by object size
- Big objects are allocated with `malloc`
- No maximum heap size
  - A huge heap (8GB) is requested from OS
  - Relies on overcommitting by OS
- GC kicks in with dynamic periods rather than internal OOM limits

Page Manager

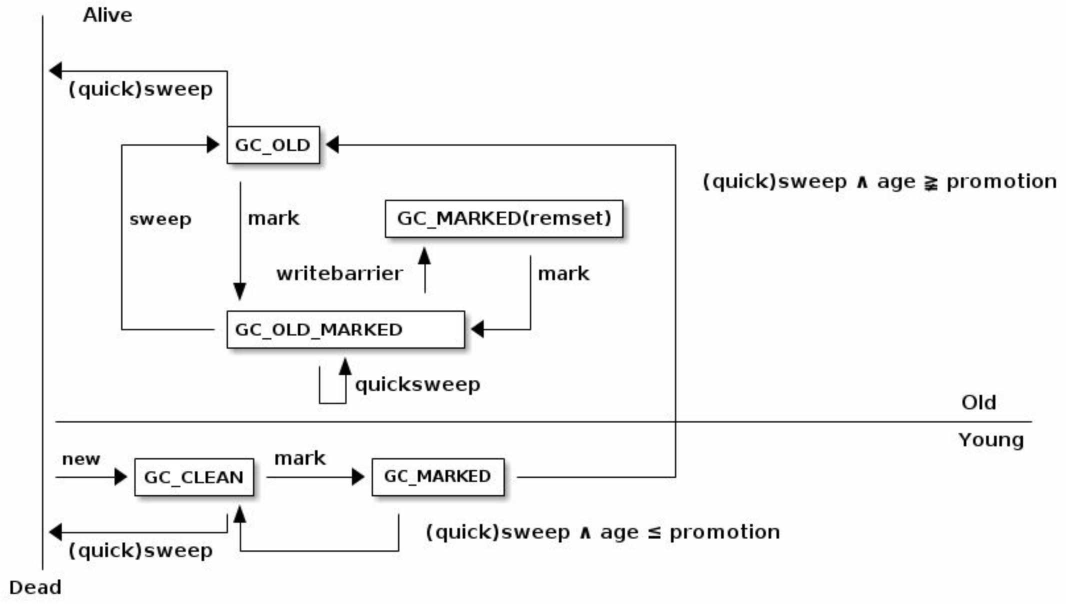
- Julia uses its own page manager for memory allocation
- The page manager organizes memory in **regions** and **pages**
- Pages in a region allocated in Rust via `mmap()` system call
- Track which chunks of pages are allocated via allocation map



A region's allocmap[]	Bit 31	Bit 30	Bit 29	...	Bit 1	Bit 0	
allocmap[0]	1	1	1	...	1	1	(chunk 1's bitmap)
allocmap[1]	0	0	0	...	0	1	(chunk 2's bitmap)
...	...	...	...	...	...	...	...
allocmap[N]	0	0	0	0	0	0	(chunk N's bitmap)


Safepoint
Safepoint
Premark
Mark
Cache sync
N. Finalizers
Sweep
J. Finalizers
Doing GC
Running Julia/native code
Waiting

Object Lifetime



Neptune

Neptune

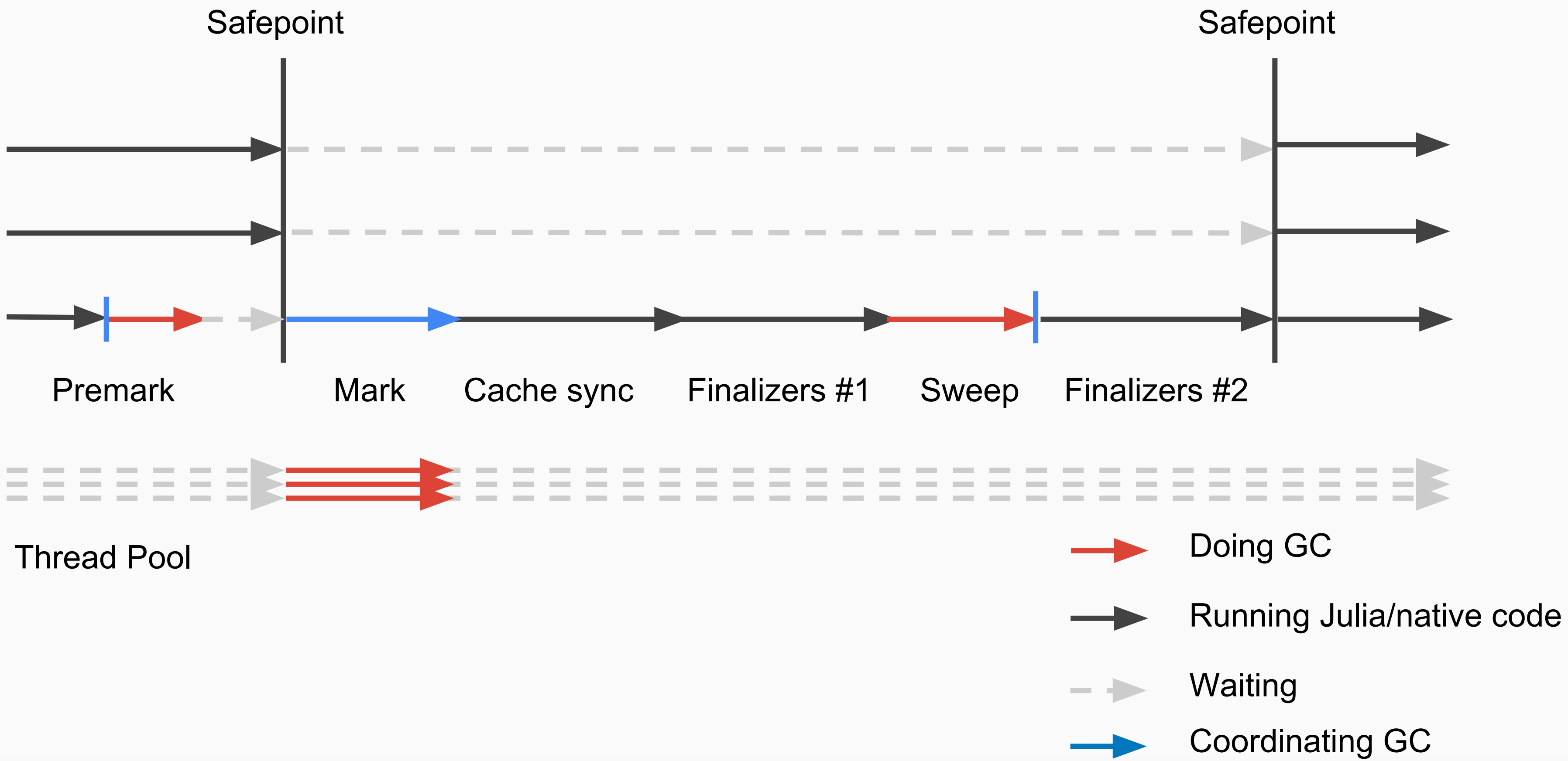
- Our parallel garbage collector
- A fixed number of threads for marking
- Single-threaded sweeping
- Mostly same as Julia's GC, to test effect of parallel GC
  - Significant effort in trying to exactly mark and sweep the same objects that Julia does, even without parallelism yet involved

Why Rust

- A **modern** systems programming language
- Affine types for “zero-cost” abstractions
- Easy concurrency
- Memory-safety without GC
- Nice foreign function interface

Design Decisions

- Profile-guided. Using Valgrind, OProfile, in-code timers
- Parallel sweep hindered performance
  - Sweep is mostly memory-bound, parallelizing it increased cache misses
- Heavy use of lock-free data structures and caches in many places to prevent blocking


Safepoint
Safepoint
Premark
Mark
Cache sync
Finalizers #1
Sweep
Finalizers #2
Thread Pool
Doing GC
Running Julia/native code
Waiting
Coordinating GC

Thread-local mark cache

- **Remsets**
- **Big object lists**
- Statistics for decisions

Mark stack & load balancing

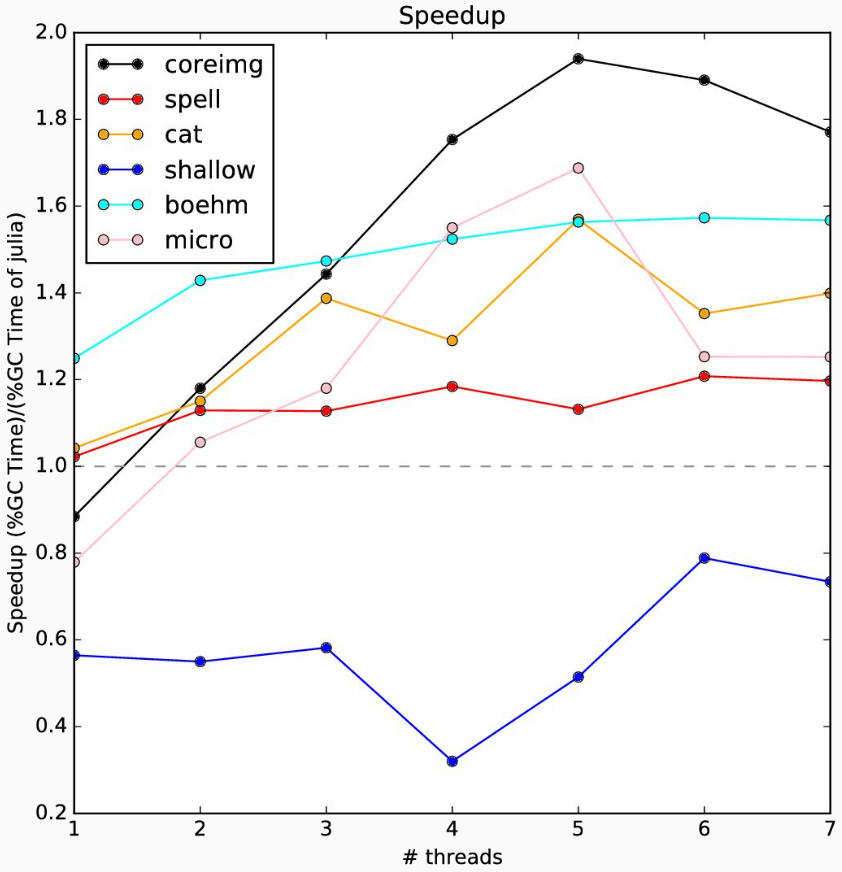
- Work-stealing thread pool
- A Treiber-stack with epoch-based memory management
- Each thread
    1. Takes a job from mark stack
    2. Follows pointers until a certain depth
    3. Puts remaining pointers to the mark stack

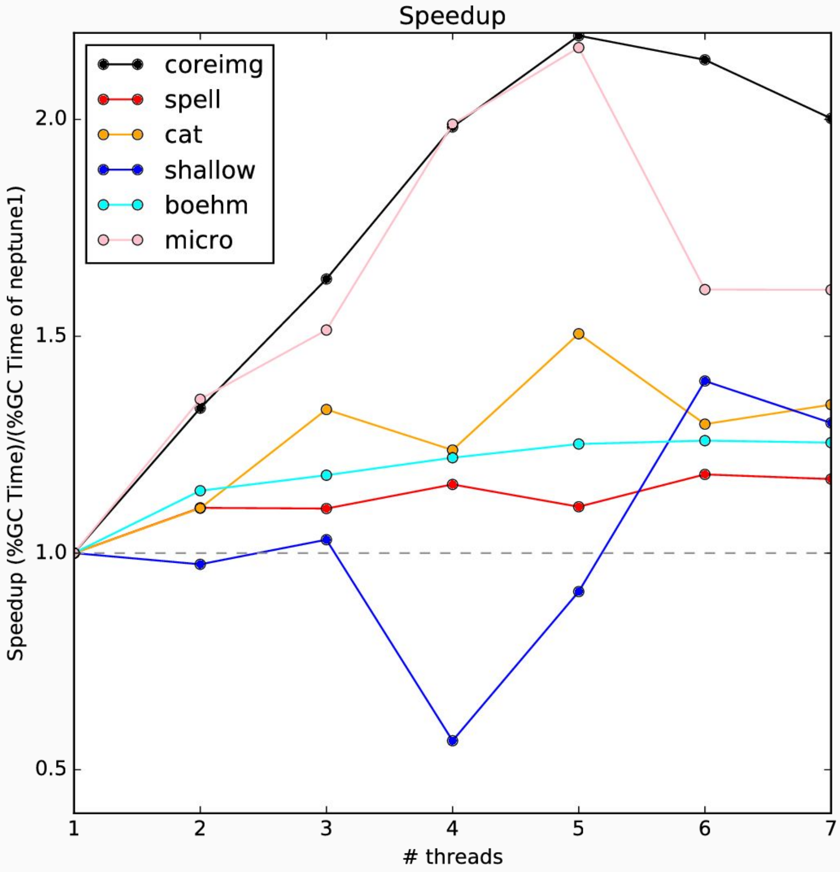
Evaluation

Benchmarks

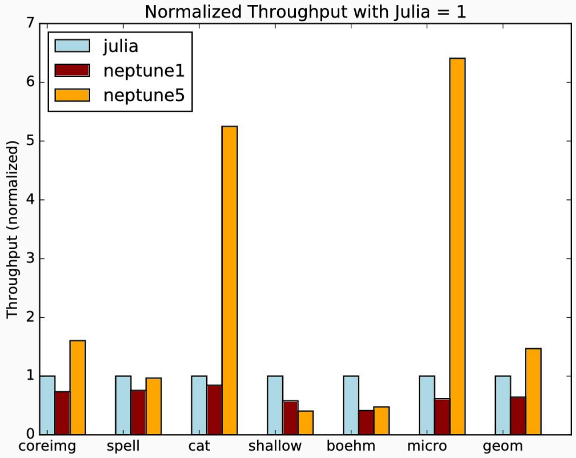
- Julia system core image (**coreimg**)
- Julia microbenchmarks (**micro**)
- Matrix concatenation (**cat**)
- Peter Norvig's spell checker (**spell**)
- Handwritten benchmark generating lots of young garbage (**shallow**)
- Hans Boehm's “An Artificial Garbage Collection Benchmark” (**boehm**)

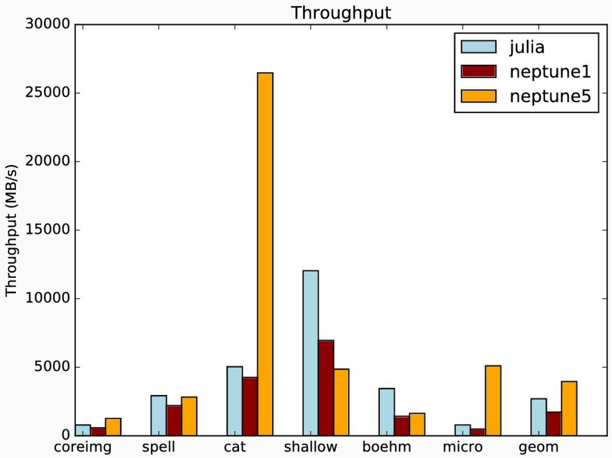
Speedup (in terms of %GC Time)

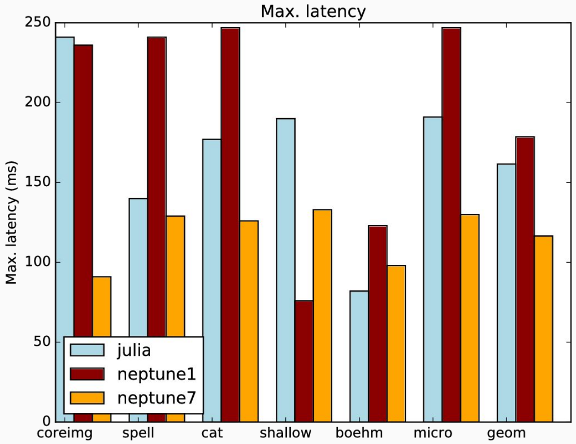




Latency and Throughput







Takeaways

Takeaways

- Just parallelizing a GC can gain some sufficient performance
- Multi-language code performance suffers from passing the FFI boundary
- It is possible to write a GC with static thread-safety guarantees
- Profile before making performance-related decisions

Parallel GC for Julia

It’s possible and can improve performance.

“There are only two hard things in Computer Science: cache invalidation and naming things.”

- Phil Karlton

“There are only two hard things in Computer Science: cache invalidation, naming things and off-by-one errors.”

- Leon Bambrick

Don’t write garbage collectors.

Just kidding.

Thanks!

On a memory allocation, check if GC condition is met; if so, begin the following collection steps:

Collect:

1. Wait for all threads to suspend
2. **Mark**:
   a. Pointer-free objects (empty tuple type, true, false, etc.)
   b. Each thread's remset objects (old objects pointing to younger ones, from last GC)
   c. Each thread's local roots (thread's module, tasks, etc.)
   d. Initial root set (builtin values, assorted constants, etc.)
   e. Objects referenced by objects on "finalize list"
   f. Objects put on mark stack during a. - e. above (queued to marking depth limit)
3. Flush mark cache (move big objects in cache to corresponding big object lists so sweep can see it)
4. **Sweep**:
   a. Finalizer list (and schedule finalization)
   b. Weak references
   c. Malloced arrays
   d. Big objects
   e. Each pool in each thread
5. Run finalizers

Marking Process:

1. Check current recursion depth; queue object if too high
2. Case on object's type tag, marking and recursively scanning its children/array contents/fields/etc.:
    a. Vector
    b. Array
    c. Module
    d. Task
    e. Other data type

Sweeping Process:
(Basically, depending on the object, free it…)