Neptune: A Safe, Parallel Garbage Collector for Julia

Mehmet Emre Michael Christensen

1 Summary

As dynamic languages become increasingly more popular, the need for quicker, safer, and more scalable automatic memory management in these systems becomes increasingly more important. Garbage collection in such systems involves reclaiming memory that is no longer needed by a program, and if done improperly, can cause undesirable degradation in program performance and throughputs and unacceptable increases in latency. Julia is a dynamically-typed, scientific programming language that aims to be highly performant, but unfortunately, its garbage collector is single-threaded and stop-the-world, causing it to miss out on possible speedups from the utilization of multicore processors. Our project, **Neptune**, is a multithreaded garbage collector for Julia, written in Rust, that aims to be safe and highly-parallel. Neptune achieves a $1.95\times$ average speedup, a $1.5\times$ increase in throughput, and a 27% decrease in latency, over Julia's garbage collector. We show that exploiting the strengths of a modern, strongly- and statically-typed systems language allows us to create a performant parallel garbage collector for Julia.

2 Introduction

Dynamic languages like Python, Ruby, and Julia have become popular languages of choice for doing scientific research. Their ease of use and quick prototype-test development loop have attracted masses of scientists from fields outside of computer science. These languages have large collections of mathematics-, graphics-, and machine-learning-related libraries (e.g. for doing quick computation on matrices, visualizing a data set, or performing NLP on a corpus of text), making adoption even more enticing. Their biggest advantage, however, could arguably be that they free the programmer from worrying about dynamic memory allocation and reclamation; they hide memory management from the user by wrapping allocation into the runtime system itself.

This ease of use, however, has a cost; trivial memory management can often lead to unpredictable, unreliable, and unsatisfactory performance issues. For one, the user relinquishes almost all control to when, how often, and for how long garbage collection (the process of finding and reclaiming unused memory) occurs. This essential system service often involves several phases of identifying, marking, and/or moving the contents of the heap around, and depending on the particular implementation, can cause intermittent, and possibly noticeable, pauses in program execution.

In particular, Julia's garbage collector is stop-the-world, single-threaded, mark-and-sweep, and non-copying. Despite billing itself as high-performant, we posit that Julia's garbage collector is anything but, and we seek to create a garbage collector that is most importantly, highly-parallel. We use Rust, a systems programming language created in 2010, rewriting small portions of the Julia source to call into our Rust garbage collector for its allocation and cleanup routines. As a happy consequence of rewriting Julia's garbage collector in Rust, we gain the added assurance of type-safety, something the C language could hardly be argued to provide.

We proceed as follows:

- In section 3, we discuss the Julia programming language and its design, particularly the design of the data structures and algorithms relating to its garbage collector.
- In section 4, we discuss our approach to creating a parallel garbage collector. We briefly explain the Rust programming language and our rationale for using it as the implementation language. We discuss the structure of the new multithreaded garbage collector and detail some of the challenges encountered in the process.
- In section 5, we compare Julia's garbage collector to Neptune on a suite of benchmarks, offering several explanations for the results achieved.
- In section 6, we conclude our report and discuss possible further improvements.

3 The Julia Language

Julia is a dynamic scientific programming language which aims for end-to-end throughout, rather latency. It has function-level just-in-time compilation via LLVM, and uses stable, highly-performant numeric computation libraries written in FORTRAN.

3.1 Julia's Garbage Collector

Julia's garbage collector uses a mark-and-sweep algorithm since it cannot move data which is pointed to native (non-Julia) code. This design decision was made because the language aims for a really fast foreign function interface to reuse numerical computation libraries written in other languages. Although Julia itself can use multiple threads during normal execution, for garbage collection, it is single-threaded. It does some pre-mark work concurrently, but in general, it is stop-the-world, meaning it waits until all threads have reached a safe point to pause, then begins garbage collection on a single thread. It uses two generations for marking, "young" and "old", and has two type of collections, "quick" and "full", which involve collecting just the young generation or the entire mass of memory, respectively.

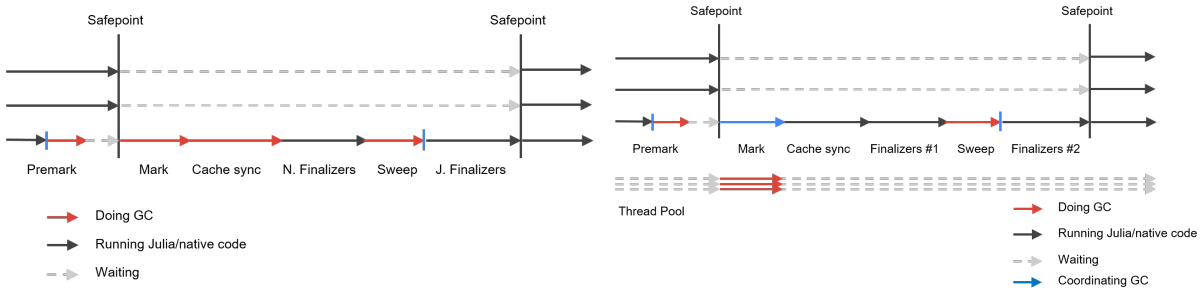
A new object begins life in the young generation, with its mark bit initially set to clean. During any garbage collection (quick or full), if it is marked and has survived fewer than some arbitrary number of collections, it is returned to a clean state and persists for further use by the program. If it is marked and is old enough to be promoted, it is marked as old. During subsequent full collections, when this old object is marked, its mark bit gets changed once again to be identifiable as being both old and marked, and a write barrier is set up so that it can be stored in the remembered set if it points to young objects.

Each program thread maintains its own set of thread-local object pools, categorized by size, for allocating small objects. Big objects, however, are allocated with `malloc` and are consequently stored on global big object lists. Julia doesn't impose a maximum heap size; consequently, garbage collection kicks in on dynamic periods because there is no internal out-of-memory limit to rely on.

3.2 Julia's Page Manager

Julia uses it own page manager for memory allocation. It allocates huge (8GB) memory regions via the `mmap` system call, relying on overcommitting from the operating system, and splits these regions into 32-page chunks and likewise these chunks into pages. Each region tracks which chunks of pages are allocated via an allocation map, which is an N-length array storing 32-bit bitmaps representing each page within a chunk. This way, Julia can track which pages have young data or marked objects, making checking during the sweep phase quick because only a single bit needs to be checked in order to skip a page.

3.3 GC Stages



(a) Stages of Julia GC

(b) Stages of Neptune GC

Figure 1: Stages of Julia GC and Neptune

Figure 1a shows the various stages of Julia's garbage collection. Julia begins garbage collection choosing a thread to perform the collection. This thread acquires the global GC lock and begins doing some premarking work concurrently with other program threads, which soon learn that they must stop normal execution because garbage collection is going to begin.

Once all the threads have suspended normal execution and reached a safe point, the GC begins the process of marking. Marking is the process of traversing the root set and all objects recursively reachable from said root set in order to mark all memory in use as actually in use. Specifically, it first marks all pointer-free objects (the empty tuple type, **true**, **false**, etc.). Then it marks each thread's remembered set objects (i.e. all old objects that point to younger ones, which were remembered from the last garbage collection). Next it marks each thread's local roots (the thread's modules, tasks, etc.). It also marks the initial root set (built-in values, assorted constants, etc.) and objects referenced by objects on the finalizer lists. Finally, it marks objects put on the mark stack during the previous marking process just described, because of a marking depth limit that is imposed to prevent stack overflows during recursive calls in the marking algorithm.

The process of marking an individual object is fairly straightforward. The collector first checks the current recursion depth, queuing the object if too high. Otherwise, it essentially performs a large switch-case statement on the object's type tag. If it is a vector or array, it marks and recursively scans the object's vector/array contents. If the object is a user data type, it marks and scans its fields; and similarly for other objects.

After marking, the Julia GC flushes the mark cache, which involves moving big objects in the big object cache to the corresponding big objects list so subsequent sweeping can see them. It also then causes native finalizers to run. Once flushed, the collector begins the sweep phase. Sweeping is the process of traversing all of memory and reclaiming the space previously used by unmarked (i.e. unreachable and therefore garbage) objects. Specifically, Julia's GC sweeps the finalizer list (at the same time scheduling objects for finalization), weak references, malloc'ed arrays, big objects, and each pool in each thread.

Finally, the collector thread releases its hold of the global GC lock, calling the Julia finalizers. Once completed, it causes all threads suspended at their safe points to resume normal execution.

4 Neptune

Neptune is our parallel answer to Julia's single-threaded garbage collector. We use a fixed number of threads for marking and a single thread for sweeping, for reasons to be explained shortly. Its overall structure is highly similar to Julia's for several reasons (as can be seen in figure 1b). First, our goal was to test the effects of parallelism on garbage collection, not to experiment with novel approaches to the general garbage collection problem. Second, Julia's C implementation is highly complex, relying on delicate memory layout requirements, among other things. We did not want to make our job of understanding, tracking, marking, and sweeping the contents of memory as organized originally by Julia any harder by attempting optimizations on their algorithm, apart from the complexities of adding parallelism.

4.1 Why Rust

Rust is a modern systems programming language, created in 2010. It prides itself in providing many "zero-cost" abstractions – abstractions and guarantees that in other systems language normally require extensive library support or checks. It gives the user memory safety and automatic memory management **without** garbage collection due to its sophisticated type system, which incorporates "affine" types. All of this makes concurrency quick and easy in Rust, and also importantly, it provides a nice foreign function interface, which would prove essential for interacting with Julia.

4.2 Design Decisions

Once we completed our implementation of Julia's garbage collector in Rust (which took the bulk of our time), we had to decide where to proceed in achieving our goal of a parallel garbage collector. Our general design was inspired by [2] and [3], especially the parts about thread-local data structures and handling parallelism. Our specific design decisions were primarily profile-guided, meaning we used the output of tools like Valgrind and OProfile, as well as in-code timers and measurements, to determine where to prioritize parallelization implementation. We saw that while marking would benefit greatly from having multiple threads, parallel sweeping actual hindered performance. Since sweeping is mostly memory-bound, parallelizing it increased cache misses. We also took advantage of Rust's libaries in heavily using lock-free data structures and caches in many places to prevent unnecessary blocking and lock contention.

Neptune reads the value of the environment variable `NEPTUNE_THREADS` and creates a work-stealing thread pool. The parallel marking algorithm described in section 4.3 uses this thread pool rather than creating new threads every single time to avoid the overhead of thread creation.

4.3 The Parallel Marking Algorithm

We adapted Julia's marking algorithm to be multithreaded. The original marking algorithm is implemented as a recursive algorithm that walks the root set and pushes all the leaves onto a stack (called the *mark stack*) when it reaches a certain recursion depth. To parallelize this algorithm, we made a couple changes: Firstly, we replaced the original mark stack with a Treiber stack, a simple lock-free, thread-safe, and memory-safe stack algorithm. Secondly, we enhanced all marking with atomic updates to be thread-safe. Finally, we implemented some thread-local caches that get synchronized after marking to prevent having too many expensive accesses to global data structures accompanied with locks. To do so, we extended Julia's mark caches and added one thread-local mark cache per garbage collection worker thread. Our thread-local mark caches contain local updates to remsets, big object lists, and statistics for future collection decisions.

Our marking algorithm gives marking jobs for all objects in the mark stack to the thread pool and waits for the worker threads to finish and synchronize. If the mark stack is not empty because worker threads added more objects in the meantime, the algorithm will do more iterations of the same job assignment until the mark stack is empty after synchronization.

5 Evaluation

We evaluated Neptune against Julia on a computer with 16 GB RAM and an Intel Core i7-4790 CPU with 8 cores reported to the OS, a frequency of up to 3.60 GHz, and 32KB, 256KB, 8MB L1, L2, L3 caches, respectively. The processor's hardware frequency governor was set to "performance" mode to disable power saving features. We also excercised the processor before measuring the results for each benchmark to simulate a long-running execution as the CPU frequency governor dials down the frequency when many cores are used for a long time to maintain the temperature. We used the following benchmarks for evaluation: building Julia's core image and system image, i.e. compiling the standard library while using the type inference engine *written in Julia* (`coreimg`); microbenchmarks used for promoting Julia on its website (`micro`); a matrix concatenation benchmark (`cat`), since matrix concatenation is used heavily by programmers with a MATLAB background, which constitute a considerable amount of Julia's user base; Peter Norvig's spellchecker (`spell`), a benchmark used in other garbage collection papers such as [2]; a handwritten benchmark (`shallow`) that creates lots of garbage in the young generation; and an implementation of Hans Boehm's "An Artificial Garbage Collection Benchmark" in Julia (`boehm`).

The optimal speedup is observed with 5-6 worker threads (Figure 2a); after that point, threads start racing for cores fewer than the threads[1]. [3] observes similar dramatic congestion with more threads than available CPU cores. There are two reasons that we didn't achieve a unit increase per thread (e.g. 6 threads don't give $6\times$ performance increase): firstly, our current load balancing algorithm is suboptimal since it measures work by the depth of traversal rather than the total number of pointers seen. We have also experimented with a variation that makes a stopping decision based on the total number of pointers seen; however, so far our results using this second approach aren't much better due to the extra arithmetic involved with counting all the pointers seen while processing a work unit. Secondly (and more importantly), we do not parallelize sweeping (which takes 40% of the single-threaded time on average on these benchmarks) and got bitten by Amdahl's law. Although we implemented a parallelized sweeping algorithm to get a better result, we observed that sweep is mostly memory-bound and that multithreaded sweep actually got worse performance due to threads fighting for memory bandwidth, dirtying the cache (since all the pages are aligned to same boundary), and the sheer amount of atomic operations. [1] suggests a lazy sweeping algorithm that skips allocated but unused pages and offloads adding such pages to freelists to memory allocation time. Julia implements such a lazy sweeping algorithm, which causes GC times measured for Julia to be underapproximations because of some of this sweeping[2]. We did not implement that lazy algorithm due to lack of time available; we expect that implementing it would decrease sweeping time dramatically across the board and would result in a steeper speedup curve.

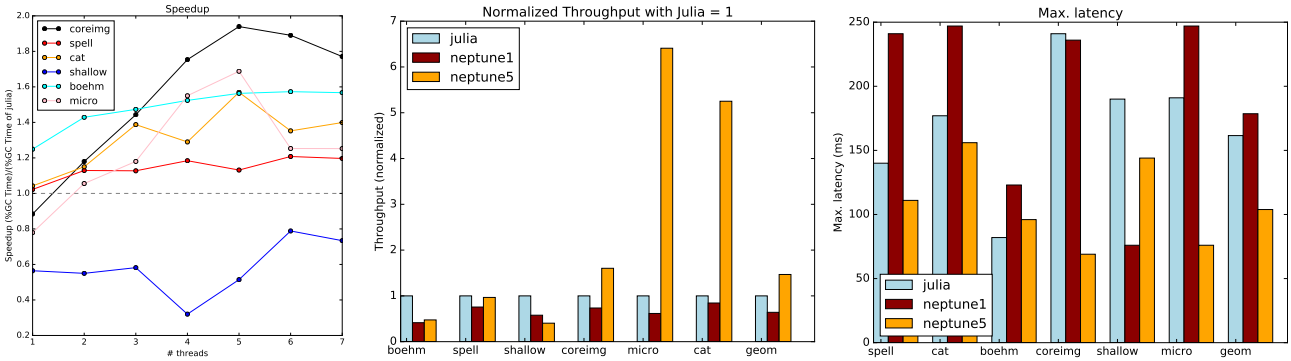
As seen in the throughput results in Figure 2b, our garbage collector has on average 30% worse single-threaded performance due to the overhead of maintaining thread safety and the lack of the aforementioned optimization. Marlow et al. [2] also report similar overhead when parallelizing GHC's garbage collector, which is a mark-and-copy collector.

Figure 2c shows that our parallel GC achieves a better maximum latency on most cases except for when many young and unreachable objects are created (in which case the marking time is relatively small enough that

[1]There is one additional thread to coordinate the worker threads; the machine was running `X` and `ssh` during our benchmarks to simulate a realistic situation.

[2]We observed that Julia's GC skips 70% of pages and offloads them to memory allocation time.

coordinating the thread pool takes more time than marking itself). The main reason behind our lower single-threaded latency figure is the same as our single-threaded throughput results.



(a) Speedup over Julia (b) Normalized throughput (c) Maximum GC latency (ms)

Figure 2: Experiment results. Figure 2a shows speedup over Julia in terms of % time taken in GC normalized with Julia = 1. Figure 2b shows throughput normalized with Julia = 1.

6 Conclusion

We observed an up to $1.95\times$ increase in performance in terms of % time spent in GC and on average $1.5\times$ throughput improvement by implementing a parallel garbage collector. Using a language with static safety guarantees (Rust) helped in implementing parallelization correctly by catching non-thread safe accesses at compile time.

In the future, implementing the lazy sweeping algorithm mentioned in the Evaluation section would be a good direction to take (it is a rather low-hanging fruit that just requires some more time to implement; unfortunately we didn't have enough time to implement it before deadline). Another future improvement is designing a better load balancing method (like keeping a thread-local counter for work being done) to divide the work into more even chunks and to increase parallelization further. We are planning to try other thread-safe stack algorithms to implement the mark stack due to the Treiber stack not scaling well under high loads. Having a more adaptive garbage collection algorithm that makes decisions about parallelization based on past data about live objects would be an interesting direction to follow. Such an algorithm can adaptively choose parallelization based on an estimation of the size of the object graph being walked, e.g. a simplistic approach would be parallelizing only when doing a full collection or keeping track of the ratio of live data to allocated memory after each marking to make an estimation.

References

[1] Boehm, H.-J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1991), PLDI '91, ACM, pp. 157–164.

[2] Marlow, S., Harris, T., James, R. P., and Peyton Jones, S. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management* (New York, NY, USA, 2008), ISMM '08, ACM, pp. 11–20.

[3] Marlow, S., and Peyton Jones, S. Multicore garbage collection with local heaps. In *Proceedings of the International Symposium on Memory Management* (New York, NY, USA, 2011), ISMM '11, ACM, pp. 21–32.