Assessing and Improving Garbage Collection Performance in the Julia Programming Language

by

Diogo Correia Netto

B.S. Computer Science and Engineering, Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author .....
Department of Electrical Engineering and Computer Science
January 20, 2023

Certified by .....
Alan Edelman
Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Comitee

Assessing and Improving Garbage Collection Performance in the Julia Programming Language

by

Diogo Correia Netto

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

With the increasing popularity of the Julia programming language for memory-intensive applications, garbage collection (GC) is becoming a performance bottleneck, with reports of poor GC performance ranging from differential equation solvers to large database benchmarks.

There have been several GC optimizations (such as the implementation of a generational collector) targeting the Julia GC over the last decade, but none of them was in the direction of a multithreaded GC.

This thesis assesses GC performance in the Julia programming language and implements optimizations focusing on parallelizing automatic memory management routines.

Thesis Supervisor: Alan Edelman
Title: Professor

Acknowledgments

First, I thank my supervisors Alan Edelman and Valentin Churavy for the opportunity of conducting my MEng thesis at Julia Lab.

Second, I thank the collaborators with whom I had the chance of working with during my thesis: Kiran Pamnany, Christine Flood, Jameson Nash, Julian Samaroo, Nathan Daly and others.

Finally, I thank my family and friends for the support during my thesis and during my time at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the context of programming languages, garbage collection refers to a form of automatic memory management that takes away from the programmer the responsibility of manually deallocating unused dynamically allocated memory and transfers it to a set of methods that are often integrated with the language runtime system. Although the use of garbage collectors (GCs) improves software memory safety and often increases programmers' productivity [9], it can also lead to a degradation of performance compared to manual memory management [8].

For the case of multithreaded programs, the so-called "stop-the-world" GCs may cause an interruption of the working threads for a non-negligible time when a collection cycle is run. This performance bottleneck is what motivated the implementation of garbage collectors (such as ZGC [3] and Shenandoah GC [3], both of which target the Java runtime system) which have a significantly reduced stop-time, which may be executed concurrently with the working threads, and which may also use multiple threads within the GC itself.

1.2 Guiding Questions

The effectiveness of multithreaded GCs has been well studied and established for programming languages with reference semantics and which generate large amounts of garbage (e.g. Java), as well as systems' programming languages with extensive use of escape analysis (e.g. Golang). The motivation of this thesis is to assess the performance impact of parallelizing GC in the context of the Julia language, a dynamic high-level language designed for technical computing.

Performance optimization in Julia often involves avoiding GC at the greatest possible extent (e.g. avoid allocating in parallel sections of code). One of our goals in this thesis is to provide a a high-performance implementation of a multithreaded GC for the Julia language and an analysis of implementation details that proved to be essential to achieve good GC performance, so that we can give the first steps towards shifting this paradigm of performance optimization in Julia and reduce the burden of avoiding allocations from programmers.

Chapter 2

An Analysis of Different Aspects of Garbage Collection

2.1 Background

Since the first implementation of a GC for LISP by McCarthy et al., different techniques for automatic management have been explored as a way to increase performance, increase application responsiveness, and to provide better memory efficiency. We analyze in this chapter some of these techniques, hoping to provide in a subsequent chapter some context into the implementation ideas that are currently used by Julia.

2.2 Generational GC

In a generational GC, objects are partitioned according to their lifetimes during execution. Such technique relies on the generational hypothesis [8], meaning that most objects die (that's it, become unreachable) at a young age. The hope is that during most collections, we are able to traverse a small fraction of the heap consisting of mainly young objects, thus reducing the length of GC pauses.

It should be noted, however, that the object graph of a programming language may also contain references from old to young objects, which need to be traversed

to ensure reachable objects are correctly traced. These references from old to young objects are often recorded into a data structure denoted as the remembered set, which must also be traversed during the marking phase.

2.3 Precise versus Conservative

In a precise GC, the language runtime is provided with information by the compiler that allows it to distinguish pointers from scalars, for example.

Most modern dynamic programming languages such as Julia have a precise collector, but conservative ones are still necessary for uncooperative environments such as those from a C program or to scan foreign objects which were allocated by another runtime but used when different programming languages inter-operate.

2.4 Moving versus Non-moving

In a moving (or copying) collector, objects are moved between different regions and compacted during a collection in such a way as to avoid fragmentation.

Besides the potential reduced fragmentation, a moving collector may also provide better spatial cache locality for the running program (mutator) and relatively cheap allocation in the form of bumping a pointer in the frontier between allocated and free memory.

Non-moving collectors often have an increased cost of allocation due to management of free lists, but may in certain cases provide better memory efficiency (take for instance, a moving semi-spaces [8] collector which may need a $2\times$ cost in memory to manage its "to" and "from" spaces).

2.5 Stop-the-world versus Concurrent

There are scenarios in which the GC pauses coming from a stop-the-world GC (which halts working threads when collection is running) are unacceptable from a program

responsiveness point of view.

To solve this problem, certain collectors allow the marking phase to be run concurrently with the mutator. It should be noted that new references may be created by the mutator while collection is running and read or write barriers [8] may need to be implemented by a concurrent collector to ensure references are not skipped by a marking or scanning phase.

Although mutator responsiveness is often increased by concurrent collectors, barriers put an extra cost into pointer reads or writes performed by the mutator, so the tradeoff between these two factors is often non-trivial and requires careful analysis.

Chapter 3

Garbage Collection in Julia

3.1 Julia's GC Algorithm

As of v1.9, Julia has a serial, stop-the-world, generational, non-moving mark-sweep garbage collector. Native objects are precisely scanned and foreign ones are conservatively marked.

An opaque tag is stored in the front of GC managed objects, and its lowest two bits are used for garbage collection. The lowest bit is set for marked objects and the second lowest bit stores age information (e.g. it's only set for old objects).

Objects are aligned by a multiple of 4 bytes to ensure this pointer tagging is legal.

Sufficiently small objects (up to 2032 bytes) are allocated on per-thread object pools.

A three-level tree (analogous to a three-level pagetable) is used to keep metadata (e.g. whether a page has been allocated, whether contains marked objects, number of free objects etc.) about address ranges spanning at least one page. Sweeping a pool allocated object consists of inserting it back into the free list maintained by its pool.

Two lists are used to keep track of the remaining heap-allocated objects: one for sufficiently large malloc'd arrays ($mallocarray\_t$) and one for sufficiently large objects ($bigval\_t$).

Sweeping these objects consists of unlinking them from their list and calling *free* on the corresponding address.

Field writes into old objects trigger a write barrier if the written field points to a young object and if a write barrier has not been triggered on the old object yet. In this case, the old object being written to is enqueued into a remembered set, and its mark bit is set to indicate that a write barrier has already been triggered on it.

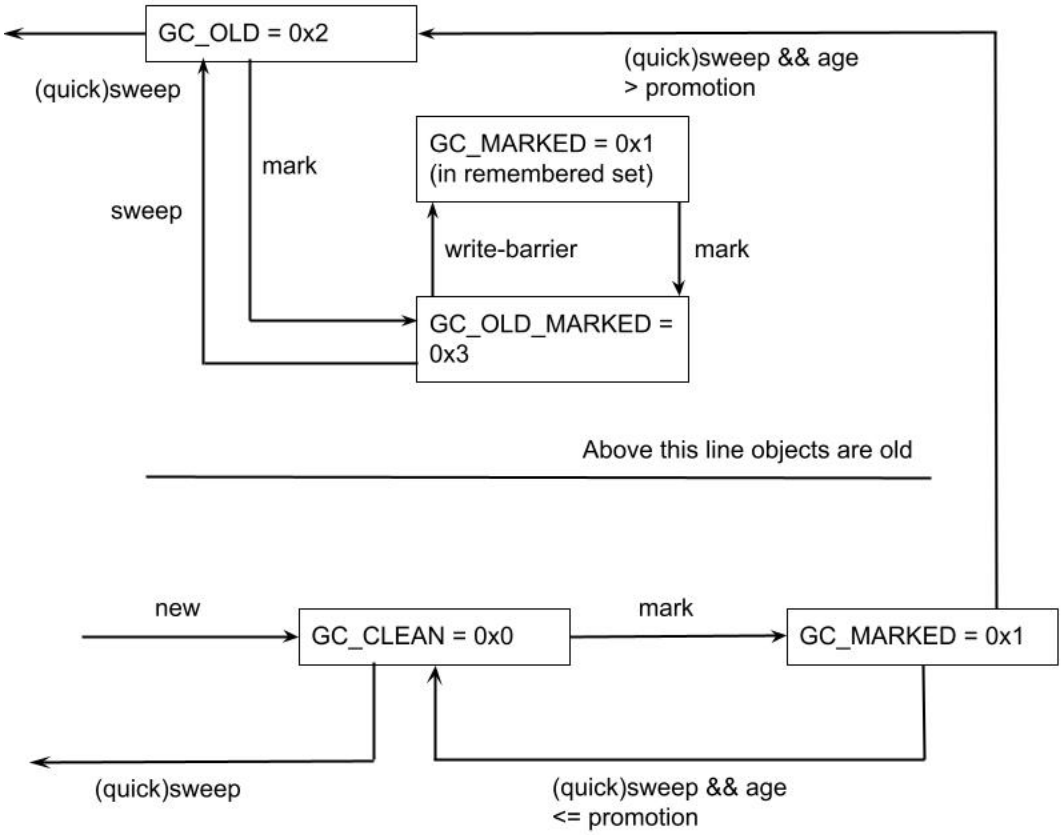
There is no explicit flag to determine whether a marking pass will scan the entire heap or only through young objects and remembered set. The mark bits of the objects themselves are used to determine whether a full mark happens. The mark-sweep algorithm follows this sequence of steps:

- Objects in the remembered set have their GC mark bits reset (these are set once write barrier is triggered, as described above) and are enqueued.
- Roots (e.g. thread locals) are enqueued.
- Object graph is traversed and mark bits are set.
- Object pools, malloc'd arrays and big objects are sweeped. On a full sweep, the mark bits of all marked objects are reset. On a generational sweep, only the mark bits of marked young objects are reset.
- Mark bits of objects in the remembered set are set, so we don't trigger the write barrier on them again.

After these stages, old objects will be left with their mark bits set, so that references from them are not explored in a subsequent generational collection. This scheme eliminates the need of explicitly keeping a flag to indicate a full mark (though a flag to indicate a full sweep is necessary).

The state diagram for object's GC mark-bits is shown in figure 3-1.

Figure 3-1: State diagram for GC bits, adapted from *src/gc.c* in JuliaLang's repository. (quick)sweep refers to either generational sweep or full sweep.



Chapter 4

Changing Single-Threaded GC Algorithm to Better Support Parallelization

4.1 Previous Work

As of v1.9, the GC marking phase was implemented through an iterative depth-first-search (DFS) that manually managed two stacks, one of which contained iterator states used to keep track of the object currently being traced. As an example, to mark arrays, the algorithm would pop the corresponding iterator state from the stack, iterate over the array until it found an unmarked reference, and if so, it would update the iterator state (to reflect the index it left off), "re-push" it into the stack and proceed with marking the unmarked reference it just found. The iterator states would act analogously to call stack frames for the objects being marked.

One of the motivations for choosing this design over a DFS implementation with recursive function calls was the possibility of marking arbitrarily deep objects without running into a stack-overflow, since the work queues would be heap-allocated and manually managed through the use of *push*, *pop* and *repush* calls.

4.2 Proposed Refactor

Although such choice of using iterator states is effective for reducing memory footprint associated with GC mark-queues, the creation of the same compressed states reduces the amount of parallelism exposed for parallel marking (which can be estimated by the number of work items in the mark-queues).

The first proposed refactoring was precisely changing the graph traversal algorithm used in the marking phase so that it exposed more parallelism in the form of work items that could potentially be claimed in a work-stealing algorithm [5].

To accomplish this goal, we changed the traversal code so that, essentially, when an object was being marked, all of its outgoing references were pushed into the mark-queue at once. To avoid the large memory footprint when marking large arrays of references we also implemented batching in the form of dividing the mark-work into array suffixes of decreasing size.

To mark large arrays, the algorithm would first push at most 2^{16} work items into the mark-queue, create an iterator state (*chunk*) corresponding to the remaining references and push the corresponding *chunk* into a separate work queue. Once the *chunk* was claimed, it would mark the next 2^{16} (at most) references and shrink the size to represent a smaller suffix left to be marked.

The algorithm for marking objects and for marking arrays is summarized in listings 4-1 and 4-2.

Figure 4-1: Code used to mark objects.

```
jl_gc_markqueue_t *mq = &ptls->mark_queue;
jl_value_t *new_obj;
for (; obj8_begin < obj8_end; obj8_begin++) {
    new_obj = ((jl_value_t **)obj8_parent)[*obj8_begin];
    if (new_obj)
        verify_parent2("object", obj8_parent, &new_obj, "field(%d)",
                       gc_slot_to_fieldidx(obj8_parent, &new_obj));
    gc_try_claim_and_push(mq, new_obj, &nptr);
}
```

Figure 4-2: Code used to mark arrays. *MAX_REFS_AT_ONCE* is set to 2^{16}.

```
jl_gc_markqueue_t *mq = &ptls->mark_queue;
jl_value_t *new_obj;
// Decide whether need to chunk objary
size_t nobjs = (obj_end - obj_begin) / step;
if (nobjs > MAX_REFS_AT_ONCE) {
    jl_gc_chunk_t c = {GC_objary_chunk, obj_parent,
                       obj_begin + step * MAX_REFS_AT_ONCE,
                       obj_end, NULL, NULL,
                        step, nptr};
    gc_chunkqueue_push(mq, &c);
    obj_end = obj_begin + step * MAX_REFS_AT_ONCE;
}
for (; obj_begin < obj_end; obj_begin += step) {
    new_obj = *obj_begin;
    if (new_obj)
        verify_parent2("obj array", obj_parent, obj_begin, "elem(%d)",
                       gc_slot_to_arrayidx(obj_parent, obj_begin));
    gc_try_claim_and_push(mq, new_obj, &nptr);
}
```

Chapter 5

Parallelizing the Marking Phase

5.1 Motivating the choice of Parallel Marking

As a language primarily designed for technical computing, most of Julia applications tend to be throughput oriented rather than latency oriented.

Due to the throughput oriented characteristics of the programming language, we judged that parallel marking and parallel sweeping would be the most suitable performance optimizations for garbage collection. This choice was also backed by performance observations coming from other language runtimes: as an example, the parallel stop-the-world collectors in Java are often referred as throughput collectors [2] and recommended for this class of applications.

Parallel marking seemed to be a less invasive change in the language runtime compared to parallel sweeping, and because of that, was chosen as the performance optimization targeted in this thesis.

As an example of a significant change required to parallelize sweeping, the three-level metadata tree (analogous to a three-level pagetable) mentioned in chapter 3 requires the acquisition of a lock to have its metadata modified, and would require a non-trivial refactoring to make it thread local or global and lock-free so that we could effectively parallelize sweeping without serializing it on the acquisition of the previously mentioned lock.

5.2 Julia's Safepoint Mechanism

Garbage collection for multithreaded Julia programs starts with the safepoint mechanism [8], which is implemented through the insertion of snippets of code by the compiler which read from a memory location whose permissions are changed whenever GC is about to start.

Figure 5-1: Safepoint read used by the Julia runtime (semantically equivalent to the safepoint read inserted by compiler).

```
#define jl_gc_safepoint_(ptls) do {                          \
        jl_signal_fence();                                   \
        size_t safepoint_load = *ptls->safepoint;            \
        jl_signal_fence();                                   \
        (void)safepoint_load;                                \
    } while (0)
```

In this case, threads incur a segmentation fault, are trapped in the signal handler, and halted while single-threaded GC is running.

Our design consisted of using the safepoint mechanism from Julia to recruit threads for parallel marking and running it within the SEGV signal handler.

As one may expect, this approach restricted the set of *libc* functions we could call in parallel marking and, as an example, made it impossible to dynamically grow or shrink mark-queues in the GC through the use of *malloc* or *realloc* calls.

5.3 Implementation

Dynamic load balancing is critical to ensure good parallelism for GC, where the nature of graph traversal makes it unsuitable for static partitioning. We implemented dynamic load balancing between threads running GC through work-stealing [5]. Each thread would keep its own double-ended queue of GC work items to be marked and could steal work items from other threads whenever its own queue was drained.

Figure 5-2: Work-stealing in *gc_mark_loop_*. *cong* refers to a random number generator.

```
void gc_mark_loop_(jl_ptls_t ptls, jl_gc_markqueue_t *mq)
{
    void *new_obj;
    pop : {
        new_obj = gc_markqueue_pop(&ptls->mark_queue);
        // Couldn't get object from own queue: try to
        // steal from someone else
        if (!new_obj)
            goto steal;
    }
    mark : {
        gc_mark_outrefs(ptls, mq, new_obj, 0);
        goto pop;
    }
    steal : {
        // Steal from a random victim
        for (int i = 0; i < 2 * jl_n_threads; i++) {
            uint32_t v = cong(UINT64_MAX,
                              UINT64_MAX, &ptls->rngseed) % jl_n_threads;
            jl_gc_markqueue_t *mq2 = &jl_all_tls_states[v]->mark_queue;
            new_obj = gc_markqueue_steal_from(mq2);
            if (new_obj)
                goto mark;
        }
    }
}
```

After trying to steal $2 \times jl\_n\_threads$ times from a random victim, a thread would offer termination by atomically decreasing an atomic counter used to keep track of the number of threads running marking. Termination of parallel marking was declared whenever such counter hit the value of 0.

We also used the spin-master protocol from [7], so that one thread not running parallel marking (the spin-master) would be responsible for peeking at other's threads

queues to estimate the amount of available mark work, and recruiting a number of threads proportional to that. It should be noted that the choice of which thread was the spin-master was not fixed but rather could change dynamically. Indeed, the spin-master could recruit itself for parallel marking if a large amount of GC mark work was available. In this case, when a thread that failed in its $2 \times jl\_n\_threads$ attempts to steal were to be dispatched back to the parallel marking waiting region, it could become a new spin-master upon successfully acquiring the spin-master lock.

5.4 Other Implementation Details

One of the most important implementation details to achieve good performance on AMD EPYC machines (e.g GPUCI) was to replace an atomic thread fence with a corresponding dummy instruction with a *lock* prefix on x86-64.

Figure 5-3: Inling an instruction with *lock* prefix on x86-64 for performance purposes.

```
#if defined(_CPU_X86_64_)
    __asm__ volatile ("lock orq $0, (%rsp)");
#else
    jl_fence();
#endif
```

Indeed, in the x86-TSO model [10], such dummy instruction with a *lock* prefix would provide the same sequential consistency guarantees as atomic thread fences while having a much lower cost on modern CPUs [4].

Chapter 6

Performance Evaluation of Parallel Marking (Single-Threaded)

6.1 Benchmarks

We ran the benchmarks from GCBenchmarks [1] summarized in table 6.1.

Table 6.1: Description of benchmarks from GCBenchmarks.

Benchmark	Description
append.jl	Repeatedly growing Vectors.
strings.jl	Heap fragmentation from multi-sized small strings
TimeZones.jl	Creation of repeated short String allocations
many_refs.jl	Large ($\sim$ GB) object array
single_ref.jl	Large ($\sim$ GB) object array
pidigits.jl	Tests small BigInts
pollard.jl	Tests small BigInts
list.jl	Small pointer-heavy data structure
tree.jl	Small pointer-heavy data structure
flux_multithreaded_training.jl	Multithreaded Machine Learning Training
tree_mutable.jl	Small pointer-heavy data structure (multithreaded)
tree_immutable.jl	Small pointer-heavy data structure (multithreaded)

6.2 Assessed Metrics and Machine

Our assessed metrics are GC wall-clock-time, aggregate (mutator + GC) wall-clock-time and number of CPU cycles spent in GC, taken as averages over ten runs. We didn't observe variation beyond 2% in the total memory consumption, so this metric is omitted.

It should be emphasized that our goal with this refactor is to better enable parallelization on the marking phase, so achieving performance neutrality in GC times (or close to that) is enough from a performance perspective.

The information of the machine used to run the benchmarks is summarized in table 6.2.

Table 6.2: Machine used to run benchmarks

Machine name	Description	Cores	L1 Cache	L2 Cache	Memory
GPUCI	AMD EPYC 7402	48	1.5MB	24MB	528GB

6.3 Limitations

A fundamental limitation of the proposed assessment is the lack of changes in the target heap size when running the benchmarks. As of v1.9, Julia provides a *–heap-size-hint* flag which, as suggested by the name, gives the runtime some information about what heap size it should set as the target. This isn't a hard limit, which made it unsuitable for benchmarking purposes.

6.4 Performance Results

Table 6.3: Relative speedup in GC time, aggregate time and cycles for GPUCI. Higher is better.

Benchmark	$\frac{GC_{master}}{GC_{pmark}}$	$\frac{T_{master}}{T_{pmark}}$	$\frac{cycles_{master}}{cycles_{pmark}}$
append.jl	1.00	0.90	1.07
strings.jl	1.24	1.05	1.29
TimeZones.jl	1.00	1.00	1.03
many_refs.jl	0.70	0.73	0.67
single_ref.jl	1.63	1.36	1.87
pidigits.jl	1.00	1.01	1.00
pollard.jl	1.03	1.01	1.01
list.jl	1.12	1.08	1.15
tree.jl	1.05	1.00	1.03
flux_multithreaded_training.jl	0.81	0.96	0.87
tree_mutable.jl	1.13	1.03	1.26
tree_immutable.jl	1.19	1.04	1.13

We observed a reduction of 5% in the geometric mean of GC times, a reduction of 0.5% in the geometric mean of aggregate times, and an 8% reduction in the geometric mean of the number of CPU cycles spent in GC.

Note that our implementation consists not only of the parallelization of the marking phase but also of the single-threaded refactor described in chapter 4. This is likely the main factor contributing to some of the performance improvements reported above.

The many_refs.jl and single_ref.jl benchmarks mostly have the purpose of ensuring the memory footprint of marking arrays of references under the new algorithm described in chapter 4 doesn't increase substantially. We judged their performance under memory consumption, runtime, and cycles as acceptable, even though they are outliers in terms of performance compared to the other benchmarks.

Chapter 7

Performance Evaluation of Parallel Marking (Multithreaded)

7.1 Performance Results

We partition the benchmarks into three categories: benchmarks which had positive scaling with parallel marking, benchmarks which had negative scaling and those for which parallel marking was close to performance neutral as we increased threads.

Table 7.1: Scalability for GCBenchmarks. ↑ refers to positive scaling, ↓ refers to negative scaling and − refers to performance neutral.

Benchmark	*scalability*
append.jl	↓
strings.jl	↓
TimeZones.jl	↑
many_refs.jl	↓
single_ref.jl	↑
pidigits.jl	↓
pollard.jl	−
list.jl	↓
tree.jl	↑
flux_multithreaded_training.jl	↑
tree_mutable.jl	↑
tree_immutable.jl	↑

Our assessed metrics are GC wall-clock time and aggregate (mutator + GC) wall-clock time. We ran our multithreaded GC implementation on GPUCI up to 16 threads.

For means of comparison, the parallel garbage collector implementation for Java described in [6] achieved $2 - 5\times$ speedup in garbage collection time on 8 processors. We set as our goal to achieve such speedup on graph benchmarks and benchmarks which allocate garbage in parallel, use cases in which we would have an object graph closer in shape to the ones observed in a language with reference semantics such as Java.

On both master and parallel marking, the maximum heap size in some of the benchmarks (e.g. tree_mutable.jl and tree_immutable.jl) changed with the number of threads, which would make an analysis of speedup relative to the single-threaded case unsuitable due to an uneven amount of garbage collector work. Because of that, we chose to analyze performance of parallel marking relative to the implementation in master running with the same number of threads.

7.2 Performance Results Relative to master

7.2.1 Positive Scaling

We observed a speedup relative to master in GC wall-clock time and aggregate wall-clock time for the benchmarks TimeZones.jl, single_ref.jl, tree.jl, flux_multithreaded_training.jl, tree_mutable.jl and tree_immutable.jl.

Figure 7-1: Relative speedup in GC times. Higher is better.

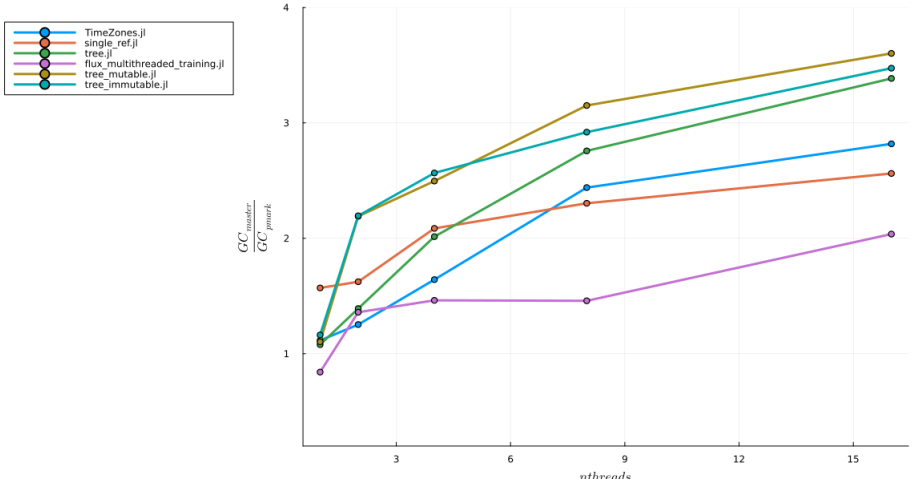
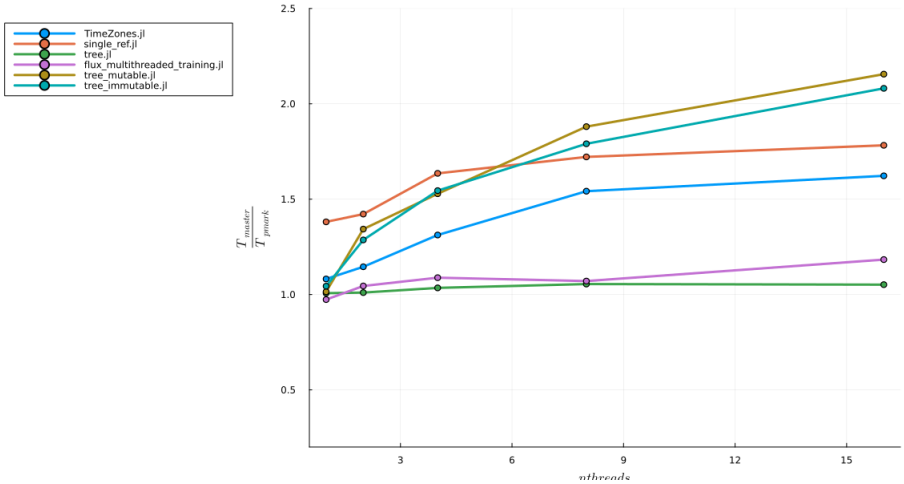


Figure 7-2: Relative speedup on aggregate times. Higher is better.



We found parallel marking was particularly effective for graph benchmarks such as tree.jl, tree_mutable.jl and tree_immutable.jl as well as benchmarks which allocate large arrays of pointers, such as TimeZones.jl (array of strings) and single_ref.jl (array

of references). Furthermore, the $2-3.5\times$ speedup we observed for these benchmarks is on par with the $2-5\times$ speedup achieved by the parallel garbage collector from Java described in [6].

7.2.2 Performance Neutral

We observed, on average, no significant effect in GC wall-clock time and aggregate wall-clock time relative to master (as we increased the number of threads) for the benchmark pollard.jl.

Figure 7-3: Relative speedup in GC times. Higher is better.

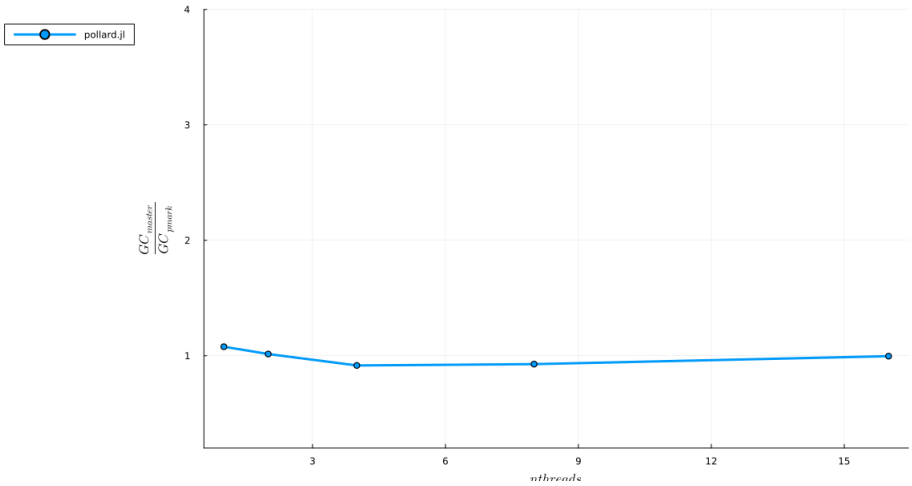
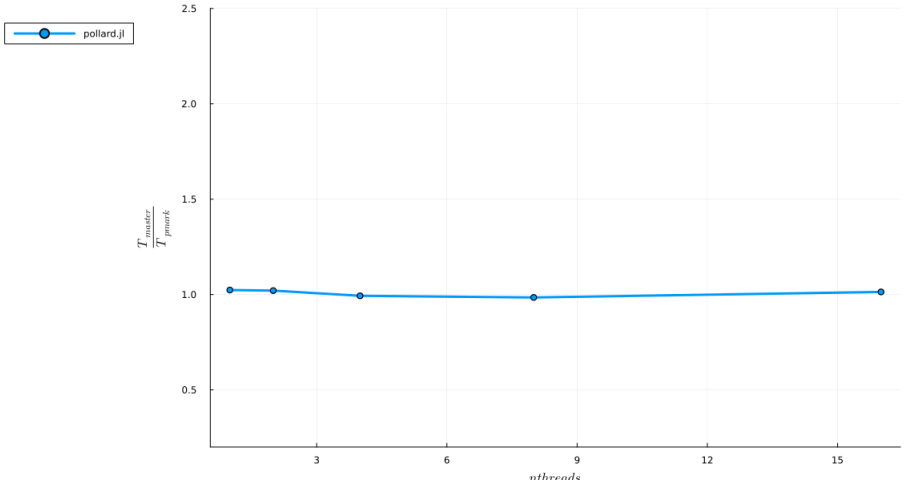


Figure 7-4: Relative speedup on aggregate times. Higher is better.



Since pollard.jl is a benchmark of arbitrary precision arithmetic, we didn't expect

its object graph to be particularly suitable for parallelization. Still, we didn't observe a performance regression coming from this benchmark either.

7.2.3 Negative Scaling

We observed a performance regression relative to master (as we increased the number of threads) in GC wall-clock time and aggregate wall-clock time for the benchmarks append.jl, strings.jl, pidigits.jl, list.jl and many_refs.jl.

Figure 7-5: Relative slowdown in GC times. Higher is better.

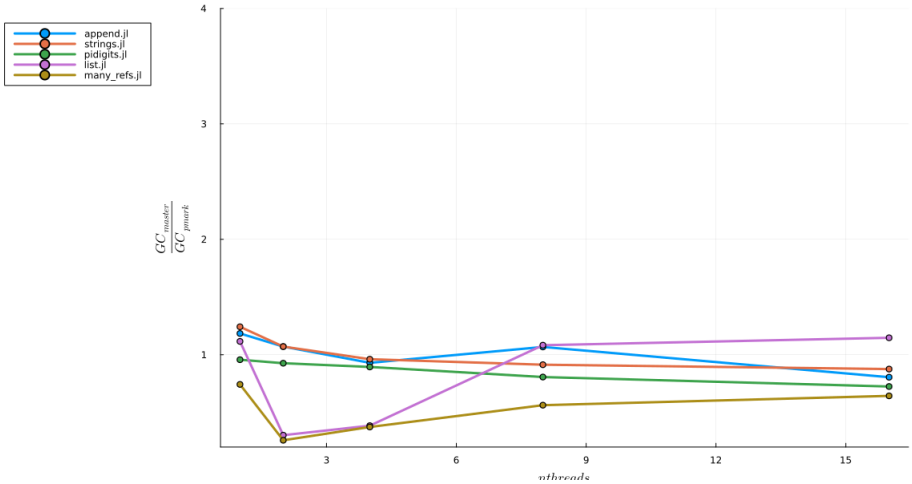
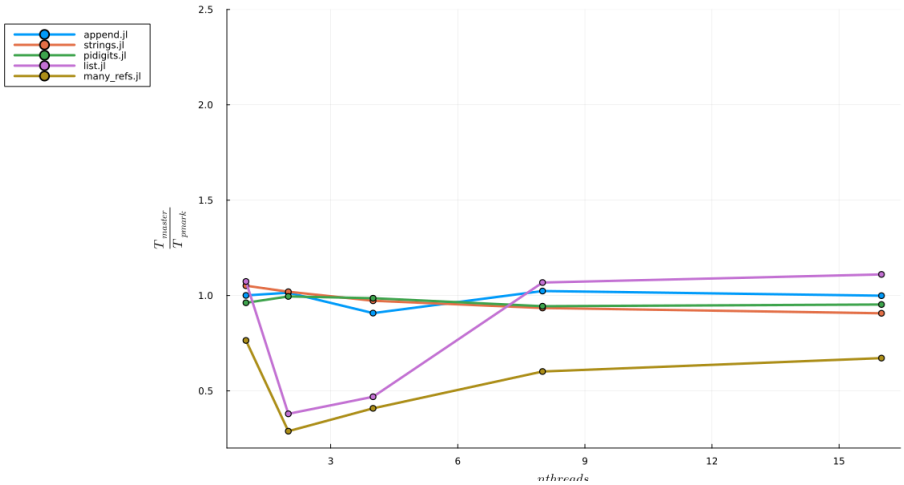


Figure 7-6: Relative slowdown on aggregate times. Higher is better.



It should be noted that not all programs have an object graph shape suitable for parallel marking. Indeed, we expect to observe parallel speedup on data structures

high enough of work-depth ratio [5] (metric adapted from task-scheduling), which explains the slowdown for list.jl (such benchmark creates a large singly-linked list to be marked). Other benchmarks such as many_refs.jl put an artificially high stress on the latency of mark-queue operations (e.g. *push, pop*), and thus have a slowdown with parallel marking, since the work-stealing queues used in the latter require extra synchronization primitives in the form of atomic operations or operations which are semantically equivalent to an atomic thread fence [4], for example.

Due to the parallelization being effective for a subset of the benchmarks, we chose to keep parallel marking under a runtime flag that could be set by users on startup of a Julia session.

Chapter 8

Conclusion and Future Work

We analyzed in this thesis the benefits of a multithreaded GC for the Julia programming language, and as a result, have also provided a GC implementation with comparable single-thread performance to the implementation from Julia v1.9 and with good scalability for a subset of benchmarks.

We observed that the parallelization was effective for a subset of our benchmarks, which motivated the choice of keeping our implementation under a runtime flag that could be set by users on startup of a Julia session.

All the measurements in this thesis were done with a research prototype, and we believe the most immediate direction of future work would be to finalize the adjustments necessary to push our implementation into production.

Bibliography

[1] GCBenchmarks. https://github.com/JuliaCI/GCBenchmarks. [Online; accessed 6-January-2023].

[2] Oracle Documentation of The Parallel Collector. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html. [Online; accessed 3-January-2023].

[3] D. Beronić, N. Novosel, B. Mihaljević, and A. Radovan. Assessing contemporary automated memory management in java – garbage first, shenandoah, and z garbage collectors comparison. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1495–1500, 2022.

[4] Uros Bizjak. Use lock prefixed insn instead of MFENCE. https://gcc.gnu.org/pipermail/gcc-cvs/2020-July/314418.html. [Online; accessed 24-December-2022].

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[6] Christine Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. 11 2001.

[7] Wessam Hassanein. Understanding and improving jvm gc work stealing at the data center scale. In *ISMM in conjunction with PLDI 2016*, 2016.

[8] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman Hall/CRC, 1st edition, 2011.

[9] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. *Softw. Pract. Exper.*, 29(4):345–358, apr 1999.

[10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, jul 2010.