Case Study: Analyzing the Serial and Parallel Performance of MergeSort in Julia and C

Berj Chilingirian
berjc@mit.edu

Varun Mohan
vmohan@mit.edu

Luo Qian
wqian94@mit.edu

December 9, 2015

1 Overview

Julia is a high-level programming language designed for high performance computing as well as general purpose programming. Via its parametric type system, multiple dispatch, and parallel execution, Julia has surfaced as the programming language of choice for modern technical challenges. Its success, however, has also motivated questions about its performance as a high-level language in comparison to low-level languages like C and C++.

We are interested in what a programmer must sacrifice in performance (achieved via C) for the ease of developing in Julia. To understand such performance gaps we invoke the following procedure.

1. Select an algorithm implemented in Julia's `Base` library. We assume this code to be "high quality" Julia code as it has been written by Julia contributors who have a strong understanding of the language's strengths and weaknesses.

2. Translate the code to the C programming language. Any performance differences between these two implementations are the result of the C compiler (e.g. `GCC`) versus Julia's just-in-time (JIT) compiler. We refer to this stage of experimentation as the *Base Comparison Phase*.

3. Optimize both the C and Julia implementations of the code. Performance differences at this stage are two-fold. First, employing the same optimization in both implementations may not produce the same speedup. Second, there may exist an optimization for one implementation that is not possible in the other implementation. Both are critical to understanding the underlying barriers of Julia and where there may be room for improvement. We refer to this stage of experimentation as the *Optimization Comparison Phase*.

4. Parallelize both the C and Julia optimized implementations of the code. We use Cilk to parallelize our C implementation as its semantics are quite similar to Julia's parallelization macros. We refer to this stage of experimentation as the *Parallel Comparison Phase*.

In this report we employ the above procedure on the merge sort algorithm implemented as part of Julia's `Base.Sort` module. We select merge sort for the following reasons. First, it is *not* an embarrassingly parallel algorithm: a developer cannot simply place `@parallel` macros in front of for-loops or take advantage of Julia's `pmap` function. Rather, parallelizing merge sort requires a developer to consider load-balancing issues when using the `@spawnat` macro as well as the cost of using shared memory (e.g. Julia's `SharedArray`). Second, the merge sort algorithm is well-known and allows for our performance analysis to be the focal point of this report (rather than the code's complexity).

While comparing/interpreting performance is the primary goal of this report, we also detail challenges we faced during the development process. We hope that these can be used to further improve the usability of the Julia programming language.

The rest of this report is outlined as follows. §2 is a brief overview of works related to this project. §3 details the Base Sort Comparison Phase of our experimentation and delves into the native code produced by both implementations. §4 discusses improvements made during the Optimization Comparison Phase and their impact on performance. §5 details our efforts to parallelize both implementations and the resulting speedup. §6 concludes with remarks on our findings and challenges we faced when developing in Julia. §7 briefly outlines several future directions for this work. All code related to this case study can be found at `https://github.com/VarunMohan/JuliaCMerge`.

2 Related Work

There have already been several comparisons of Julia with other programming languages ([1], [2]) including one by the Julia team (`http://julialang.org/benchmarks/`). While these works have demonstrated Julia's impressive performance (especially in comparison to Matlab and Python), they have lacked a formal model explaining differences as well as an empirical analysis of the performance of Julia's built-in methods. Such details and studies are critical to explaining a Julia program's performance results as well as discovering areas of the language that can be made more efficient.

3 Base Comparison Phase

We began by porting the existing merge sort implementation of the `Base.Sort` module to our local machines. The `Base.Sort` implementation of merge sort is as follows. At the beginning, there is a recursive step where we perform the merge sort on both halves of the original array. After this, there is a copy phase wherein the first half of the second array is copied into a temporary portion. Finally, the temporary portion and the second half of the original array are merged into the original array. Note that once the size of the array is sufficiently small, we instead default to an insertion sort implementation to prune the base case.

The Julia code was then directly translated to C line-by-line. The only change we made to the original Julia implementation was to specify the input of the merge sort algorithm to be of type `Array{UInt32, 1}` instead of the original `AbstractVector` type. In this manner the Julia and C implementations are performing exactly the same procedures on the same types of data. To test code performance we benchmarked each implementation on randomly generated arrays of size 2^{23} for a total of 10 trials and averaged the results.

Julia	**C**
0.86s	0.80s

Table 1: Runtime of both the serial naive Julia and C implementations in seconds averaged over 10 trials for randomly generated arrays of size 2^{23}.

aAs seen in Table 1, the C implementation is 7.5% faster than the Julia implementation. Note that the differences between the performance can be attribtued to Julia's JIT compiler versus compiling the C implementation with `GCC` and optimization level `-O3`, which provides the highest level of optimization for the compiler (excluding the Ofast flag, which is mostly used for fast math operations). In the following section, we justify the speedup for C by analyzing the differences in assembly produced by corresponding code snippets.

3.1 Assembly Level Differences

The first difference in assembly code can be seen during the copy phase. More specifically, after performing the merge sort on the two halves of an array v, we copy the lower sorted half of v into a temporary array t as shown in Figure 1.

```
i, j = 1, 1
while j <= half
    t[i] = v[j]
    i += 1
    j += 1
end
```

Figure 1: Snippet of Julia's merge sort copy phase.

```
cmpl    %r14d, %r15d
jae     L215
cmpq    %rcx, %rax
jae     L348
movl    %r15d, (%rbx,%r9)
incq    %r10
jmpq    L231
cmpq    %rcx, %rax
jae     L348
movl    %r14d, (%rbx,%r9)
```

Figure 2: Snippet of Julia assembly for copy phase.

```
movdqu (%rsi,%rax,1),%xmm0
add    $0x1,%r8
movdqu %xmm0,(%rcx,%rax,1)
add    $0x10,%rax
cmp    %r8,%r10
ja     189
```

Figure 3: Snippet of C assembly for copy phase.

We first take examine the assembly created by the Julia compiler. In Figure 2, we see that the `movl` instruction is being called, meaning that for every iteration only 4 bytes are being copied to the temporary array. The assembly produced by the C compiler seen in Figure 3, instead performs operations on the `xmm` register, which is a vector register of length 16 bytes. The fact that we are acting on 16 byte portions is also validated by the `rax` register being incremented by 16 for every loop iterations. Furthermore, the `movdqa` operations act on 16 byte portions from memory, showing how the compiler performs SIMD operations to allow 4 times the number of `UInt32` moves in 1 cycle. Because of this, the copy phase in for the C executable is roughly 2 times faster. It was not exactly 4 times faster since we still pay a latency penalty by executing instructions for loop control. Note, that we could have sped up the copy in C even more by passing the compiler the `maxv2` compiler flag, which allows it to use AVX2 vector registers, which are 32 bytes in size.

We now compare the differences in assembly for the merge phase, wherein we merge the two sorted halves of the original array. In Figure 4, we see that the Julia assembly contains substantially more jump instructions than the C assembly in Figure 5. This implies that the Julia executable performs significantly more branches than the C executable. The reason for this difference is that the C assembly performs bithacks and a conditional move (`cmove`) instruction rather than compare and jump instructions. The `cmove` instruction is much faster in this case since we no longer need to rely on the branch predictor to perform speculative execution. Specifically, during merge sort, there is no inherent reason why the head of one sorted halve would be selected over the other to be added to the head of the resultant sorted array, meaning that branch prediction will never be efficient in this case. Furthermore, with the `cmove`, we pay the price of adding a few more instructions and some data dependencies but prevent the flushing of the pipeline whenever the incorrect branch is taken.

```
leaq    -1(%rsi), %r11
leaq    -4(,%rsi,4), %r9
movq    8(%rdi), %r14
xorl    %ebx, %ebx
xorl    %ecx, %ecx
leaq    (%r11,%rcx), %rax
cmpq    %r14, %rax
jae L345
cmpq    8(%r8), %rcx
jae L382
movq    (%rdi), %rax
addq    %r9, %rax
movl    (%rbx,%rax), %r10d
movq    (%r8), %rax
movl    %r10d, (%rax,%rbx)
leaq    1(%rsi,%rcx), %r10
addq    $4, %rbx
incq    %rcx
cmpq    %r15, %r10
jle L49
cmpq    %rsi, %r10
jle L334
movl    $1, %r11d
cmpq    %rdx, %r10
jg  L252
```

Figure 4: Snippet of Julia assembly for merge phase.

```
cmpq    %rdx, %r12
jbe .L1
leaq    12(,%rax,4), %rcx
leaq    0(,%rbx,4), %r8
movq    %r12, %rdi
subq    %rdx, %rdi
leaq    -16(%rcx), %r10
leaq    16(%r13,%r8), %r11
leaq    0(%r13,%r8), %r9
leaq    0(%rbp,%r10), %rsi
cmpq    %r11, %rsi
setnb   %r11b
addq    %rbp, %rcx
cmpq    %rcx, %r9
setnb   %cl
orb %cl, %r11b
je  .L27
cmpq    $12, %rdi
jbe .L27
movq    %r9, %rcx
andl    $15, %ecx
shrq    $2, %rcx
negq    %rcx
movq    %rcx, %rsi
andl    $3, %esi
cmpq    %rdi, %rsi
cmova   %rdi, %rsi
testq   %rsi, %rsi
movq    %rsi, %rcx
je  .L19
```

Figure 5: Snippet of C assembly for merge phase.

4 Optimization Comparison Phase

We next attempted to optimize both implementations. The first optimization was immediately apparent. During the copy phase, instead of iterating over every element, we could instead perform a variant of a `memcpy` instruction to maximize the performance of the copy phase. Note that the original code provided in Figure 1 and `Base.Sort` module does not take advantage of Julia's library `copy!` method that makes a `memmove` system call. Using `copy!` achieves the inherent vectorization of `memmove`, improving the potential performance of the algorithm. This change was also implemented in the C program (as a `memcpy`) and a side-by-side comparison of both optimized serial codes produced the results seen in Table 2.

Julia	**C**
0.86s	0.80s

Table 2: Runtime of both the serial optimized Julia and C implementations in seconds averaged over 10 trials for randomly generated arrays of size 2^{23}.

Note that C implementation is now only 3.75% faster. Note that the speedup achieved by the C implementation due to the optimization is close to 0% as compared to a 3.5% speedup for the Julia implementation.

The reason for the speedup in Julia is obvious since the naive implementation of moving `UInt32` one by one into the temporary array is now modified to use faster SIMD operations in the `memmove` function. However, the C implementation saw no improvement. This is primarly due to the fact that the C implementation was already very fast since it utilized vector registers, and adding the `memcpy` instruction albeit making the copy slightly faster, added function call overhead, increasing latency.

The only difference in speed between the Julia and C can as before be attributed to the fact that the C compiler performs more aggressive optimizations to the merge phase, that significantly reduces the number of branches. Regardless, the serial results indicate that Julia code performs relatively similar to C although it becomes apparent that the C compiler can perform far more high-powered optimizations to reduce run-time.

5 Parallel Comparison Phase

We next attempted to parallelize both implementations. In C this was quite simple with the Cilk libraries: we added the `cilk_spawn` keyword in front of the first recursive merge sort call and added the `cilk_sync` keyword after both recursive merge calls have completed in order to merge them without races. This implementation (simplified) is shown in Figure 6. The process of parallelizing the code above took approximately

```
cilk_spawn merge_sort(v, lo, half, t);
merge_sort(v, half + 1, hi, t);
cilk_sync;
```

Figure 6: Snippet of the parallelizing merge sort in C with Cilk.

30 seconds: as a developer we do not have to concern ourselves with load-balancing issues, we simply apply Cilk's parallel model to our code and Cilk handles work-stealing and scheduling of threads, given user input of the number of threads to utilize.

In Julia this process was not as straightforward. First, in order to parallelize the merge sort algorithm in Julia, our input array must be converted to a `SharedArray` in order to be shared across processes. This resulted in a bit of difficulty. Julia's library implementation of merge sort uses the `similar` method to create an array with the same type as the input array. Unfortunately `similar` does not currently support the `SharedArray` type. This is a known issue and requires a small hack to circumvent.

Next we had to share modules between all processes added via `addprocs`. At firs this seemed straightforward: simply add the `@everywhere` macro in front of modules that need to be shared before running any computations. We soon realized that the order in which sharing and adding processes is done can effect whether or not modules are shared. After finding conflicting information online we discovered the correct ordering in order to share modules across all processes. This process, however, consumed two hours of development time.

Finally we had to parallelize the actual implementation. Unlike when using Cilk, we as developers had to consider how spawning work on different processes may effect load-balancing. Moreover, we could not simply include keywords in front of the recursive calls to merge sort. In fact, we attempted this strategy at first (as we are all experienced with Cilk) and due to the recursive creation of remote references we incurred a variety of unreadable errors. In the end, we implemented a method that first performs a parallel merge sort until all processors have a roughly equal portion of the shared-array to sort. Once each processor had been spawned of all relevant subprocesses, we defaulted to a serial merge sort implementation. The method of load balancing the computation on each worker is shown in Figure 7 on the next page.

```
if num_procs < 2
    return MergeSortSerial!(v, lo, hi, t)
end

 ...

    next_proc = convert(Int, cur_proc + floor(num_procs/2))
    num_procs_remaining = convert(Int, floor(num_procs/2))
    r = @spawnat next_proc MergeSortParallel!(v, lo, m, next_proc, num_procs
                    - num_procs_remaining, t)
    MergeSortParallel!(v, m+1, hi, cur_proc, num_procs_remaining, t)

    wait(r)
```

Figure 7: Snippet of the parallelizing merge sort in Julia.

The performance of these two implementations are shown in Figure 8.

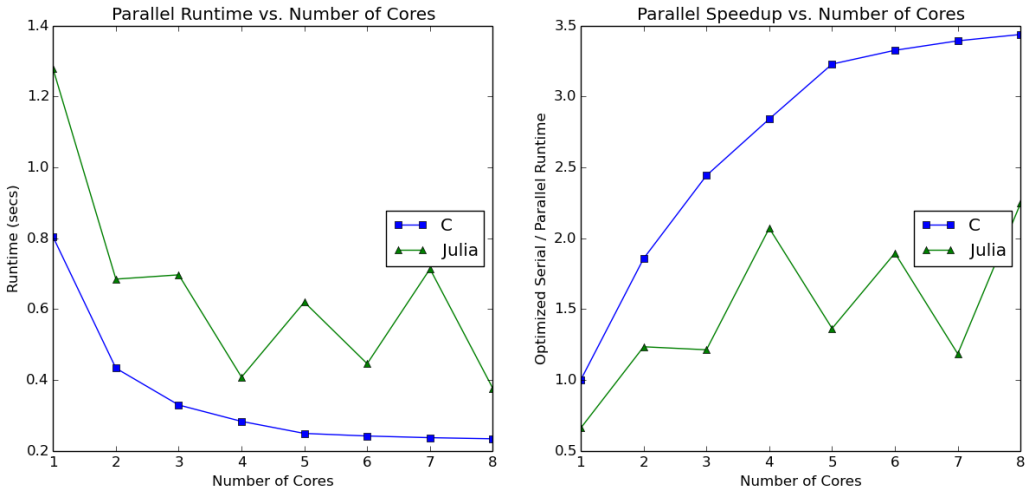

Figure 8: Comparison of Julia and C for parallel implementations across a varying number of processes.

We first see that the parallel code applied to one processor runs substantially slower than original serial implemenation with a speedup of around 0.65 in the case of one worker. This difference can be attributed to the overhead of using the Shared Array data structure, which performs an mmap on the data region. This is a very expensive operation since we potentially have to read pages from disk, which can take on the order of millions of cycles. Furthermore, declaring the original array and temporary array as shared under the current implementation of shared arrays, significantly reduces performance for a single threaded program. On the other hand, in the case of a single worker, the Cilk C implementation performs exactly the same, showing that the Cilk library does not have to perform any mapping operations in the case of a single thread.

In the case of two threads, the parallel Julia implementation performs roughly twice as fast as its parallel single thread implemenation as did the C implementaion. The Julia implemenation is still significantly slower than the parallel C program beacuse of the serial overhead of setting up the Shared Arrays.

In the case of three threads, however, Julia performs worse while Cilk continues to get speedup. This

is because Julia does not have a work stealing construct. Furthermore, in our implementation, for three workers, we have that two workers are responsible for sorting a fourth of the array while another worker is responsible for sorting an entire half. Furthermore, the program waits on the thread that has significantly more work, making the problem equivalent to the case where there are only two workers, except with the catch that there is the overhead of spawning an extra worker. Furthermore, by extending this logic, we see that our Julia implemenation will only achieve optimal speedup when the number of workers is a power of 2 since each worker is in charge of roughly the same amount of work. The results in Figure 8, validate this assertion, since we see significant speedups when the number of cores, $n = 2, 4, 8$. For the C implemenation, since work stealing is implemented, it can take advantage of the fact that a single thread has substantially more work, and allow free threads to steal work, adding slight overhead but reducing latency substantially.

Overall, however, when compared to the speedup against a single worker for executions with an optimal number of workers, the Julia implementation attains speedup comparable to that of Cilk. This demonstrates that spawning Cilk threads are similar in their overhead to spawning workers in Julia.

6 Conclusion

Going into this case study we knew that C would be faster than Julia: it is very hard for any language to compete with C and the optimizations produced by `GCC`. However, we were pleasantly surprised with the performance achieved by Julia. As the language is still very young, many of the challenges we faced during the development process are understandable. We did, however, find two features of our results to be curious and we believe they require further investigation.

First, as pointed out in §4.1, Julia's JIT compiler produces fairly naive code. If Julia were to employ methods similar to those used by Java's HotSpot virtual machine, it may be able to achieve a significant speedup. Second, as shown in §5, Julia's parallelism suffers significantly from its SharedArray implementation as well as the lack of work-stealing. The SharedArray Implementation, however, does prove useful in the case where the data is larger than memory since it amortizes the cost of performing the `mmap` call. If Julia were to offer a work-stealing construct for its `@spawn` semantics, code could be parallelized without the developer having to concern him/herself of load-balancing issues. In addition, the parallelism would have been uniform regardless of the number of processors, rather than being efficient for specific values.

7 Future Work

As mentioned in §6, Julia may benefit substantially from a work-stealing construct that allows processes to steal computation from each other. We are very interested in understanding the complexity of designing such a construct for Julia and pursuing its implementation.

8 References

[1] Aruoba, S. Boragan, and Jesus Fernandez-Villaverde. *A comparison of programming languages in economics*. No. w20263. National Bureau of Economic Research, 2014.
[2] Domkes, Justin. "Julia, Matlab, and C." Justin Domkes Weblog. N.p., 16 Sept. 2012. Web. 20 Oct. 2015.