juliacon

Basic Threading Examples in JuliaLang v1.3

Jameson Nash[1], Jeff Bezanson[1], and Kiran Pamnany[2]

[1]Julia Computing, Inc.
[2]Intel Corporation

ABSTRACT

A major distinguishing point of any programming language is how it deals with concurrency. Programmers want to extract the best throughput possible for their applications, but it is well known that taking advantage of all available CPU cores correctly and efficiently is hard. Here, we look at how JuliaLang unleashes the full power of a modern CPU's multiple cores.
One of our key considerations is to reduce the programmer's burden. We will discuss how JuliaLang aims to provide a range of modern primitives that are designed to automatically compose effectively, and some of the trade-offs we make to try to simplify the mental model for the programmer. We'll also briefly discuss our thoughts on future development.

Keywords

Julia, JuliaLang, Multithreading, Parallelism, Concurrency, Optimization, Compiler, Runtime VM

1. Introduction

A design principle for JuliaLang is to make common tasks easy and difficult tasks possible. This is demonstrated in multiple aspects of the language, from automatic memory management vs. manual memory reuse, to having one type of method with dispatch, to optional type-inference for performance. We now extend this to concurrency. By extending design work that has been present in the language for many years to provide concurrency, we have developed a means to add parallelism that preserves the (relative) simplicity of single-threaded execution for existing code, while allowing new code to benefit from multi-threaded execution. This work has been inspired by parallel programming systems such as Cilk, Intel Threading Building Blocks (TBB) and Go.
In this paradigm, any piece of a program can be marked for execution in parallel, and a *task* will be started to run that piece of code automatically on an available thread. A dynamic scheduler handles making cache-aware decisions on when and where to launch tasks. This model of parallelism has many helpful properties. We see it as somewhat analogous to garbage collection: with GC, you freely allocate objects without worrying about when and how they are freed. With task parallelism, you freely spawn tasks–potentially millions of them–without worrying about where they may eventually run.
The model is portable and free from low-level details. The programmer does not need to manage threads, nor even know how many processors or threads are available.
The model is nestable and composable: parallel tasks can be started that call library functions that themselves start parallel tasks, and everything works correctly. This property is crucial for a high-level language where a lot of work is done by library functions. The programmer can write serial or parallel code without worrying about how the libraries used are implemented. This model isn't limited to JuliaLang libraries alone: we've shown that it can be extended to native libraries such as FFTW and are working on extending it to OpenBLAS.

1.1 Background History

Initially, JuliaLang exclusively provided users the ability to use cooperatively scheduled workers. In other contexts, these may be known as "green threads", "threadlets", or "coroutines". In Julia, we've called these tasks. A task is a unit of work with its own context (stack) whose execution can be interleaved with that of any other task. Additionally, these have the ability to produce a value or an exception, making them useful for structured concurrency, without requiring an extra channel to manage. But they haven't had the ability to work in parallel (simultaneously).
Tasks have been useful for writing generators and outstanding for dealing with I/O workloads. Such cases present unpredictable latency and therefore the ability to quickly switch between different control flows is essential. The arrival of an event requires a quick context switch to react to the event, and quickly switching back to resume the original work.
A common pattern for a server is to provide a context for each child. Consider the following code snippet for a toy socket echo server. With JavaScript promises:

```
while (true) {
    listen(port)
        .then(client => {
            return client.read()
                .then(data => {
                    return client.write(data);
                })
                .then(() => {
                    return client.close();
                })
                .done();
        })
        .done();
};
```

This can be made clearer and the structure improved with async/await:

```
(async () => {
    while (true) {
        let client = await listen(port);
        (async () => {
```

```
            let data = await client.read();
            await client.write(data);
            await client.close();
        })();
    }
})();
```

This leaves some slightly obnoxious syntax, creating distracting line noise. When most parts of the language are loosely typed and inferred by the runtime, this distinction is forced upon the programmer.
Consider the Julia version:

```
# function accept(cb, server)
#     while true
#         let client = accept(server)
#             @async cb(client)
#         end
#     end
# end
accept(listen(port)) do client
    data = read(client)
    write(client, data)
    close(client)
end
```

Notice here how we leverage the do-block syntax to additionally reuse the loop logic, task allocation, and error handling from a central place.
This works well for latency bound activities such as a web server. Note that this code only uses one core, although as it performs only thread-local actions, it *could* be run simultaneously on multiple cores without significant change. However, there is also a lot of code that is not written with the expectation of simultaneous access. As such, we want to continue to provide concurrency[1] as well as parallelism[2]. If we define a thread as a unit of work managed by the runtime system, we can call this $N:1$ threading, where the runtime library manages N independent operations and maps them onto one system thread (approximately representing a CPU core). In the new system, we'll let the user now additionally have M CPU cores. This goes beyond the classic $N{:}M$ threading model however, as the programmer can specify cooperative affinities where certain tasks won't interrupt each other. We'll call this $N(k:1):M$ scheduling since we're combining the advantages of single-threaded work queues with multiple cores.
In $N(k:1):M$ scheduling, we have N units of work mapped onto M CPU threads. Additionally, each of those units of work may be composed of k cooperatively scheduled tasks. This is achieved by pinning the k tasks in a group to one CPU thread, while load balancing units of work across the available cores using the `partr` scheduler, a novel implementation of a parallel depth-first scheduling algorithm. This will be discussed further in section 5 on implementation.
We can extend this model further by factoring out a common factor of $P:P$ to write this as $N(k:1):M\ +\ P(k:1):P$ and derive one further useful use case: the ability to pin one thread to running one task (or task-group). These P tasks could be an over-subscription of the CPU cores, or take away from M, or both. Typical uses for this mode of operation would be high-availability tasks (with low latency requirements, but also minimal computation), such as background I/O processing, blocking work pools (foreign library integration), finalizers, or message server queues.
For a couple years, JuliaLang has been able to perform simple loop parallelism with the '`@threads for`' macro, roughly analogous to OpenMP's '`#pragma omp parallel for schedule(static)`' without support for reductions. This had been labelled "experimental" while we focused on making JuliaLang's runtime reentrant and threadsafe and clarified requirements for the final parallelism capabilities. The experimental threading infrastructure had no scheduler, could not interact with regular tasks or do I/O, and parallel loops could not be nested. This made it nearly impossible to write many common algorithms or use a large portion of the language while running a threaded region of code.
The new threading runtime addresses all these shortcomings. Furthermore, in addition to the new parallelism constructs that have been introduced, the previous loop parallelism capability has been rewritten on top of this runtime, demonstrating its power and flexibility.

Running Julia with Threads

In the examples below, we will be using JuliaLang v1.3 launched with multiple threads. To follow along on your own machine, you will need to download the upcoming JuliaLang release (currently v1.3.0-rc1) from `https://julialang.org/downloads`. Run `./julia` with the environment variable `JULIA_NUM_THREADS` set to the number of threads to use.
Alternatively, after installing JuliaLang, follow the steps at `http://docs.junolab.org/latest/man/installation/` to install the Juno IDE. It will automatically set the number of threads based on the number of available processor cores, and also provides a graphical interface for changing the number of threads.

2. Motivating Examples

The presence and usability aspects of threading, as exemplified here, reflect JuliaLang's general policy of giving users control. One driving philosophy is that users should have the *ability* to access the full power of their machine. And it should be easy when needed but ignorable when not required.
While many, or even most, programs can be written without needing to touch multithreading, some require them, while some benefit from them. In this paper, we'll primarily examine some cases where threads aren't required, but are improved by their presence. Additionally we'll look at a case where the work can be run sequentially with cooperative scheduling, but at greatly reduced performance.
Most thread-specific functionality is exported from the `Threads` submodule of the `Base` module. For example, we can querying it for the runtime number of threads and the id of the current thread:

```
julia> Threads.nthreads()
4

julia> Threads.threadid()
1
```

2.1 Stochastic Ordering

One of the more visual ways to show we have threads working is to show the scheduler picking up work in semi-random, interleaving,

[1] When at least two threads are making progress; a more general form of parallelism that can include time-slicing.
[2] When at least two threads are executing simultaneously

orders. JuliaLang's existing '`@threads for`' macro would split a range and run a portion on each thread with a static schedule. So in the range below, thread 1 would run items 1 and 2, thread 2 would run items 3 and 4, and so on. Now these threads support doing I/O too with that same schedule.

```
bash$> JULIA_NUM_THREADS=8 julia <<EOF
Threads.@threads for i = 1:12
    println(i, " on thread ", Threads.threadid())
end
EOF
1 on thread 1
3 on thread 2
12 on thread 8
9 on thread 5
7 on thread 4
2 on thread 1
4 on thread 2
5 on thread 3
8 on thread 4
11 on thread 7
10 on thread 6
6 on thread 3
```

But now, it's also now possible to do the same example but with a completely dynamic schedule. With the improved language runtime, this takes a few small tweaks now. We use the new '`@spawn`' macro with the existing '`@sync`' macro to delineate the work items. The '`@spawn`' macro marks a block of code that can immediately start executing, asynchronously, on any free thread. The preexisting '`@sync`' macro then waits for all (lexical) subtasks to complete, eliminating the boilerplate necessary to track and wait on each task block separately.

```
bash$> JULIA_NUM_THREADS=8 julia <<EOF
@sync for i = 1:12
    Threads.@spawn println(i, " on thread ", Threads.threadid())
end
EOF
2 on thread 5
3 on thread 4
8 on thread 7
6 on thread 5
12 on thread 7
7 on thread 6
9 on thread 8
10 on thread 5
4 on thread 3
1 on thread 2
5 on thread 1
11 on thread 1
```

But on to even more fun stuff...

2.2 Parallel Merge Sort

A classic algorithm, parallel merge sort shows nice performance benefit and scaling from using multiple threads. This function will create $O(log(n))$ subtasks which will sort independent portions of the array before merging them into a final sorted copy of the input. We use here the ability of each task to return a value to directly `fetch` the result without requiring an additional channel for data! This operation implicitly waits for the task to finish, then accesses the result value of the Task.

```
# perform a merge sort on `v` using parallel threads
function psort(v::AbstractVector)
    hi = length(v)
    if hi < 100_000 # below some cutoff, run in serial
        return sort(v, alg = MergeSort)
    end
# split the range and sort the halves in parallel recursively
    mid = (1 + hi) >>> 1
    half = Threads.@spawn psort(view(v, 1:mid))
    right = psort(view(v, (mid + 1):hi))
    left = fetch(half)::typeof(right)
# perform the merge on the result
    out = similar(v)
    merge!(out, left, right)
    return out
end

function merge!(out, left, right)
    ll, lr = length(left), length(right)
    @assert ll + lr == length(out)
    i, il, ir = 1, 1, 1
    @inbounds while il <= ll && ir <= lr
        l, r = left[il], right[ir]
        if isless(r, l)
            out[i] = r
            ir += 1
        else
            out[i] = l
            il += 1
        end
        i += 1
    end
    @inbounds while il <= ll
        out[i] = left[il]
        il += 1
        i += 1
    end
    @inbounds while ir <= lr
        out[i] = right[ir]
        ir += 1
        i += 1
    end
    return out
end
```

To see the timing results as we add threads, refer to figure 3 at the end.

While not demonstrated here, `fetch` would also automatically propagate errors, with the result of it being an error thrown if the child task ended by throwing an exception.

Since we are using in-process threads, we could further optimize this to instead mutate the input in-place and to reuse work buffers for additional performance. We have elsewhere tested that and shown the performance improvement is as expected. However, since the scaling improvement was similar between them, we've opted not to include it here.

On a single thread, this code is already quite competitive to the optimized serial implementation in the standard library, which does not use any threading:

```
julia> @time psort(a);
  2.676906 seconds (3.06 k allocations: 1.416 GiB, 3.66% gc time)

julia> @time sort(a);
  1.716137 seconds (2 allocations: 152.588 MiB)

julia> @time sort(a, alg=MergeSort);
  2.123958 seconds (5 allocations: 228.882 MiB)
```

This shows we are adding some overhead, but it is not substantial. In fact, with 2 threads, we'll already be faster than the serial implementations!

The algorithm given here is limited in the theoretical scaling capability, since the merge step is not parallelized. On large core counts, that can be important, so please see our supplementary code in appendix A for the version with optimal theoretical scaling.

2.3 Parallel Primes Sieve

An unusual use of high-level threading operations can be used to (inefficiently) compute prime numbers using the sieve of Eratosthenes. This use of threaded channels is translated from Thomas Hoare's seminal 1978 paper "Communicating Sequential Processes"[3] example 6.1. It works by creating a task for each prime number being generated. Upon receiving (and outputing) a prime, each task will then take responsibility for filtering out multiples of that prime from the input list, as represented in figure 1.

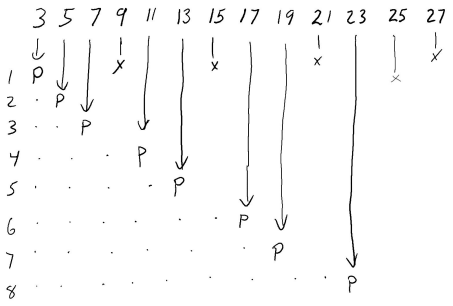

Fig. 1. Primes sieve in operation. Inputs 'n' across the top. Task numbers 'i' down the side. Outputs 'P' marked in the center.

```
function S61_SIEVE(numPrimes::Integer, nqueue::Int=5)
    done = Threads.Atomic{Bool}(false)
    primes = Int[]
    sieves = [Channel{Int}(nqueue) for i = 1:numPrimes]
    for i in 1:numPrimes
        Threads.@spawn begin
            sieve = sieves[i]
            p = take!(sieve)
            push!(primes, p)
            if length(primes) == numPrimes
                # don't pass it on--we're done now
                #= TODO: add an atomic write release barrier here =#
                done[] = true
                return
            end
            mp = p # mp is a multiple of p
            for m in sieve
                while m > mp
                    mp += p
                end
                if m < mp
                    put!(sieves[i + 1], m)
                end
            end
        end
    end
    put!(sieves[1], 2)
    n = 3
    while !done[]
        put!(sieves[1], n)
        n += 2
    end
    foreach(close, sieves)
    return primes
end
```

To see the timing results as we add threads, refer to figure 3 at the end.
Since we're creating one thread for each number, the overhead here overwhelms the computational cost of the additions. That makes this implementation *much* slower than the optimized routines typically used now, such as those provided in `Primes.jl` to compute `primes`. But is also means we show exceptional (super-linear) scaling. This is because we end up being able to run a better schedule when we can fill and empty the channels in parallel. That is also why the presence of at least a small buffer on the channel can be a significant advantage for the implementation.

2.4 Parallel Prefix Scan

Prefix-scan-sum is another classic algorithm that is able to benefit nicely from having multiple threads. Without going into any details about how this operation works or what it does, the short code below can take advantage of all cores and SIMD units available on the native machine—even with a generic ahead-of-time-compiled system image:

```
using .Threads: @threads
function prefix_threads!(⊕, y::AbstractVector)
    l = length(y)
    k = ceil(Int, log2(l))
    # do reduce phase
    for j = 1:k
        @threads for i = 2^j:2^j:min(l, 2^k)
            @inbounds y[i] = y[i - 2^(j - 1)] ⊕ y[i]
        end
    end
    # do expand phase
    for j = (k - 1):-1:1
        @threads for i = 3*2^(j - 1):2^j:min(l, 2^k)
            @inbounds y[i] = y[i - 2^(j - 1)] ⊕ y[i]
        end
    end
    return y
end

A = fill(1, 500_000)
prefix_threads!(+, A)
```

JuliaLang can express this operation so well because it defines an expressive front-end to describe optimizations to the compiler. Under the hood, it puts together a comprehensive set of features that free the user from dealing with memory management, thread management, nor compile/runtime distinction. The runtime is able to prepare a version of this function specifically optimized for the arguments types. And it spawns closures to be run on all available CPUs. The compiler can also automatically specialize the function for the current processor (both ahead-of-time and just-in-time), adjusting the ABI on-the-fly (with trampolines as needed). And our lightweight threading system will dynamically schedule the work chunks.

3. Performance

Each of the examples above shows a performance benefit attained from adding threads!
On a quad-core laptop (Intel(R) Core(TM) i7-8559U CPU @ 2.70GHz), we observed the scaling and timing numbers shown in Tables 1 and 2.
These can be seen plotted graphically in figure 2 and 3.

Table 1. Measured timing of the examples given above.

nthreads	1	2	3	4
S61_SIEVE	14.704s	7.709s	4.065s	2.241s
psort	2.609s	1.528s	1.321s	0.993s
prefix_threads!	2.375ms	1.462ms	1.100ms	1.043ms

Table 2. Timing figures from table 1 converted to scaling ratios relative to the first column.

nthreads	1	2	3	4
S61_SIEVE	1x	1.91x	3.62x	6.56x
psort	1x	1.71x	1.98x	2.63x
prefix_threads!	1x	1.62x	2.16x	2.28x

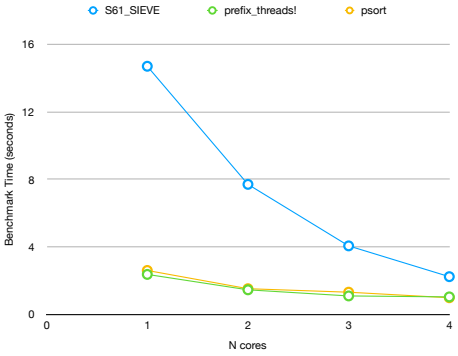

Fig. 2. Timings from table 1 plotted graphically.

4. Integration into an existing language

Another challenge we faced was seeing what would be needed to integrate this work with pre-existing code. JuliaLang is an existing, post version 1.0 language with promises to maintain backwards compatibility and a large third-party code base that depends on it. Any changes needed to have an upgrade-path. Whenever there were existing code that might reasonably be expected to be safe to use from multiple threads, that code needed to be identified and fixed. Fortunately, many key aspects of the language had previously been designed in expectations of becoming threaded. In some other popular languages, we see they have not been able to add unrestricted threading. There were several areas that needed to be tackled to determine the appropriate upgrade path:
User-facing APIs:

—concurrency basics: `Task`, and associated functions including `schedule`, `yield`, `wait`

—mutexes: `ReentrantLock` and `Condition` variables, including `lock`, `unlock`, `wait`

—synchronization primitives: `Channel`, `Event`, `AsyncEvent`, `Semaphore`

—IO and other delays: including `read`, `write`, `open`, `close`, `sleep`

—experimental `Threads` module: random assortment of building blocks and atomics

—memoization-type caches (e.g. inside `Regex.PCRE` and the `Random.GLOBAL_RNG` object reference)

Once we determined we wanted to make concurrency and parallelism use the same concept (named a `Task`), that set many priorities. Many of the APIs in our list of user-facing APIs were able to directly add thread-safety "under-the-hood", as they say. This

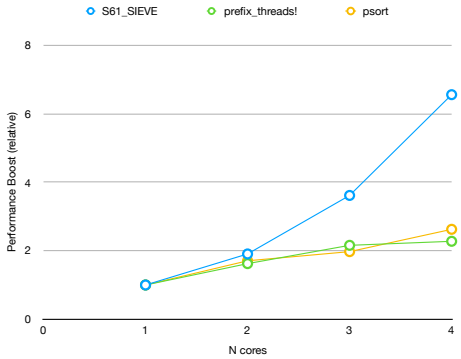

Fig. 3. Scaling ratios from table 2 plotted graphically.

meant that we found that typical user code that already interacted with IO, synchronization, locks, and tasks could continue to operate unchanged. In most cases, we achieved this by adding fine-grained locks on each critical resource. There were a few notable cases:

4.1 Changes to Tasks

The existing concurrency primitive of `Tasks` was enhanced by exposing a new, optional flag to enable thread-migration for it. We call this concept "sticky" tasks, as a default task is only cooperatively runnable on the thread that `scheduled` them. When set to false, however, the task becomes eligible to be picked up by any other thread. Combined with the internal changes to make `wait` on events and channels thread-safe, we believe this provides an easy-to-use mechanism for selecting between the simpler cooperatively concurrent usage (single-threaded) and the more general simultaneous parallelism (multi-threaded).

```
t = Task(() -> [closure code])
t.sticky = false # t may now get run on any thread
schedule(t)
...
wait(t)
```

However, while conceptually simple, the above felt slightly awkward compared to the fairly succinct `@async` syntax used for creating a concurrent task. We wanted to make it similarly convenient, so we also created a new `Threads.@spawn` macro and integrated it with the existing `@sync` macro.

```
using Threads: @spawn
@sync begin
    @async concurrent_closure()
    @spawn parallel_closure()
end # wait for all
```

4.2 Changes to Condition

The existing `Condition` object couldn't be made thread-safe. There were two replacements identified: one, replace it with an auto-resetting event with the same API; or two, replace it with a new mutex-based API. We decided to go with the latter option. This meant that existing usage of Condition was only correct if it remained on a single-thread. We decided to mechanically enforce this by asserting on usage that it was always used from the same

thread it was created on. The new API requires writing the following more verbose code pattern:

```
c = Threads.Condition()
# or alternately
#   l = ReentrantLock()
#   ...
#   c1 = Threads.Condition(l)
#   c2 = Threads.Condition(l)
...
lock(c)
try
    while !is_condition_met()
        ...
        wait(c)
        ...
    end
finally
    unlock(c)
end
...
```

Previously, this would have been more simply `c = Condition(); is_condition_met() wait(c)`. While this change may seem more difficult at first glance, we observed that while the lock acquisition here could be hidden inside `wait` in the first replacement, all of this structure will usually still be required by the `is_condition_met` function. And the code would get much further complicated by the need to release the lock before calling `wait`. We concluded therefore that in most cases the code would be made simpler and faster by changing the API to the second option. This also meant that when code was being changed to be thread-safe, it would need to replace uses of `Condition` with the new `Threads.Condition`.

4.3 Changes to I/O

Changing the I/O code (files, streams, folders, and other platform code) to work on from any thread was another big project. The existing design requires an underlying asynchronous library, with a design similar to Windows IOCP, to efficiently manage large numbers of open event sources and provide the simplicity and concision of the logic shown in section 1.1 on all platforms. For this, we have been using the libuv library. This lets us have most platform-specific code isolated in a separately tested library and provide more commonality in our runtime library. As an initial implementation to make this library safe to use from threads, we've used one big I/O lock around all calls to it. However, this library also has callbacks and will block to wait for external incoming events, so we also needed to integrate it fully with the task scheduler to get it to cooperatively release the lock on demand. We were able to do so by adding an asynchronous channel (`uv_async_t`) to wake the one thread running the event loop while all other threads sleep on a system mutex (`uv_cond_t`) when there is no work for them to perform. When we try entering the event loop, we do so only if the count of currently waiting tasks is zero. In the future, this work may allow us to move the event loop entirely to a separate thread (and/or multiple threads). It seems that this design change may thus be making threading support a mandatory requirement for the underlying VM—with the advantage we that we can get more throughput on the large-core systems that are only becoming more common.

4.4 Changes to Memoization Caches

The usual strategy for dealing with these was to turn them from true globals into thread-local variables. To assist in that goal, we assign all threads a low numbered `threadid`. This can then be used to index a global array to access the cache for that thread. For example, instead of one global `Random.GLOBAL_RNG` object representing the global `MersenneTwister` pseudo-random number generator (PRNG) state, we use a `Random.default_rng()` function to retrieve the current PRNG for that thread (or to lazy-initialize one from system randomness on first use).

```
function default_rng()
    tid = Threads.threadid()
    @assert 0 < tid <= length(THREAD_RNGs)
    if @inbounds isassigned(THREAD_RNGs, tid)
        @inbounds MT = THREAD_RNGs[tid]
    else
        MT = MersenneTwister()
        @inbounds THREAD_RNGs[tid] = MT
    end
    return MT
end
function __init__()
    resize!(empty!(THREAD_RNGs), Threads.nthreads())
end
```

4.5 Changes to the Julia Runtime Library

The functionality provided in libjulia also needed to be thread-safe. While some of it consists of stateless helper functions, much of it is where the shared global state for the language lives (by contrast, much of the system library is written in the JuliaLang language itself and as a general principle, the whole system has avoided using mutable global state unless essential).

Due to the design of the rest of the language avoiding access to mutable state inside the runtime library, we felt it would acceptable to use fine-grained locked for protecting most accesses. Many of these were added in an earlier version of JuliaLang, while threading was still under highly experimental development. These included such aspects as code-generation (JIT compilation) and GC (memory allocation and freeing).

Discussion of the GC design and subsequent updates to make it work well with threads could occupy an entire article of its own, so it will not be discussed here. Although in the future work section later in this paper, some improvements being investigated for the compiler will be discussed.

When using locks, there is a hierarchy of access that must be respected to avoid deadlocks. This is documented somewhat sparsely at `https://docs.julialang.org/en/v1/devdocs/locks/`. Over time, we'll extend this list as we discover problems or are able to simplify shared resources. There are some known issues already such as the lack of a lock around certain "toplevel-only" operations and an invalid design for the ordering of the `Module->lock`. These issues will be addressed in time—they are not believed to be insurmountable issues.

The missing toplevel lock is interesting, since it is a lock against concurrent execution of *any* other code. This will require halting all other threads in some way to inhibit accidental observation of the global state while it's in an intermediate inconsistent state. This should be possible in coordination with the GC-safepoint lock, which already has a very similar problem.

Some aspects were still too performance critical however to be able to use a lock there, so we also make careful use of atomic pointer-

publishing updates in a few specific places. As special-cases of that, we use RCU-type (read-copy-update) updates in some places and write-once in other places). This is known to work on most computer architectures. Others, such as the notorious DEC Alpha, we are content to exclude. In a code-base that already supports garbage-collection, the RCU algorithm is greatly simplified (and writers pay no additional cost), so this is typically preferred if mutation is absolutely required and reads must be fast. Otherwise, a simple lock is used.

5. Implementation

A prototype implementation of the `partr` scheduler was first written for us in C by Kiran Pamnany of Intel back in late 2016[3], following research done on scheduling threads for beneficial cache sharing for best throughput[1]. The goal of this work was effortless composition of threading-capable libraries with a globally depth-first work ordering (as opposed to $1:1$/preemptive scheduling, which would try to make progress on all work, or work-stealing scheduling, which is only depth-first local to a thread and is globally breadth-first).
The next stage of this work was then to integrate it with the existing JuliaLang runtime system and hoist as much of the implementation as possible into native Julian code. (Aside: one outcome of this work has been to allow us to delete much of the special support code from the C runtime for our prior experimental '`@threads`' fork/join-style API!)
A big challenge of this work has been implementing a sound algorithm for determining when threads should "park" themselves in a sleep mutex or wait for I/O. This requires careful coordination to ensure we don't create a single contention point when trying to `schedule` and run tasks, but also are responsive to resume when new work arrives (either internally, from another thread, or externally, from I/O streams). This is done by setting a flag in the task to notify it after work is added to the queue. If the running task sees that the thread was previously sleeping, it then additionally notifies its condition variable to wake it up.

6. Foreign Libraries

An important motivation for this work was our desire to better support multi-threaded capable libraries, without considerable CPU over-subscription killing performance due to cache-thrashing and frequent preemptive CPU context switches. Previously, the only options were often for the user to decide up-front to limit JuliaLang to N threads, and tell the threaded library (such as libfftw or libblas) use $\lfloor M \div N \rfloor$ (floordiv) cores. The most common choices probably being 1 and M, so only part of the application and running time is able to benefit from the presence of multiple cores in the system. However, given our ability to quickly create and run work items in our thread pool, we are looking at how to work with external libraries also and let them also integrate with our existing thread-pool.
This is an on-going area of exploration as we get feedback on the performance and API needs of various libraries.
We've successfully adapted FFTW to run on top of our threading runtime instead of its own[4] (a pthreads-based workpool). This took us only a few hours (we were fortunate to be able to enlist the assistance of that library's author). Without any performance tweaking (yet), we saw competitive performance results! We learned important lessons in needing to tightly optimize our scheduler latency, which is now ongoing work to achieve exact performance parity[5]. Even with some overhead imposed by generality however, we expect that the ability to compose thread-aware users and enable the better resource sharing created by the `partr` scheduler will make this an overall improvement in program operation.

7. Future work

While this work has been ongoing for several years already, there are still many interesting and important improvements to consider. We'd like to investigate ways to further expand the thread-safe API surface and integrate powerful thread-sanitizer tooling to help users write better code. There's also substantial room for the standard library to start using this threading runtime whenever possible. However, we need to explore ways to safely and conveniently expose this option to users (which often seem contradictory).
Additional performance testing is necessary to fine-tune the heuristic numbers. For example, when adding work items to the dynamic scheduler to run on P cores, what is a good ratio factor k to use when creating chunks of work? Should we make $1P$ items (assume a static schedule)? or $105P$ (assume a static schedule on either of P or $P-1$ processors for $P = 4, 6, 8, 16$)? Or perhaps simply $3P$ is sufficient to balance out much variation? And yet other workloads, however, may want one work item for every input value (like `Distributed.pmap` does)!
We specifically highlight one additional item: concurrent garbage collection. Currently, JuliaLang's runtime library needs to wait for all threads to arrive at a safe-point or be in a safe-region (such as foreign code) before GC can start. This can introduce long pauses if one or more threads are far away from hitting such a region. Presently, those only exist where manually inserted into the code, such as while waiting for a lock or doing allocation. In the future, we intend to investigate options for automatic placement of safe-points by the system to minimize GC start latency without unduly impacting allocation-free code. There are certainly more approaches to handle releasing memory then there are language implementations in existence, possibly multiple times over. So suffice to say this is an area with many possible trade-offs! For an example of where we might also go with this, please take a look at the Mono project's documentation on cooperative thread suspension[5] for how a different language, which shares a common code-generation strategy, handles this.

8. Conclusions and summary

JuliaLang's approach to multi-threading combines many previously known ideas in a novel framework. While each in isolation is useful, we believe that–as is so often the case–the sum is greater than the parts.

9. Acknowledgements

The authors would like to gratefully acknowledge funding support from Intel and relationalAI that made it possible to develop these new capabilities.
We are also grateful to the several people who patiently tried this functionality while it was in development and filed bug reports or pull requests, and spurred us to keep going.

[3] `partr` codebase: `https://github.com/kpamnany/partr`
[4] FFTW.jl partr thread support: `https://github.com/JuliaMath/FFTW.jl/pull/105`
[5] FFTW.jl performance comparison: `https://github.com/JuliaMath/FFTW.jl/pull/151`

10. Bad puns

We liken the addition of thread-safety as moving from the age of mechanization...

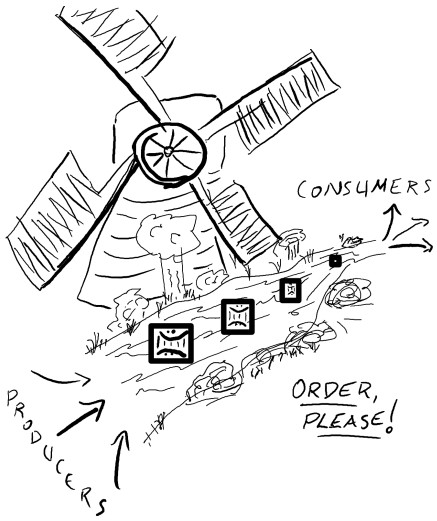

Fig. 4. Ye olde age of Newtonian power.

to the atomic age!

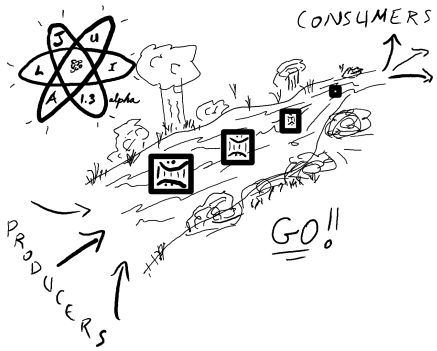

Fig. 5. The atomic age.

The figure 4 is a windmill. The figure 5 is a atom.

11. References

[1] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

[2] Fănică Gavril. Merging with parallel processors. *Commun. ACM*, 18(10):588–591, October 1975.

[3] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[4] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[5] The Mono Project. *Cooperative Suspend*, Apr 16, 2018.

[6] Arch D. Robison. *A Parallel Stable Sort Using C++ for TBB, Cilk Plus, and OpenMP*, Apr 11, 2014.

APPENDIX

A. Parallel Merge

For doing the merge step of a merge sort in parallel, we only need to add a bit of code to the `merge!` function. Let's call the result `pmerge!`. The details of this algorithm can be found elsewhere, although most textbooks do not give the stable algorithm used here. This is a minor enhancement and simplification of the algorithm given in the original paper[2], as described in "Structured Parallel Programming"[4] in section 13.1 on page 300, and in a later blog post by one of the authors[6].

```
function pmerge!(out, left, right)
    ll, lr = length(left), length(right)
    @assert ll + lr == length(out)
    if length(out) < 100_000
        # below some threshold, just do the merge
        merge!(out, left, right)
    else
        # split the larger chunk in half, then binary search the
        # smaller half to split it
        if ll > lr
            jl = ll ÷ 2
            # stable sort: find the last entry in right
            # strictly smaller than l
            jr = searchsortedfirst(right, left[jl]) - 1
        else
            jr = lr ÷ 2
            # stable sort: find the last entry in left not bigger
            # than r
            jl = searchsortedlast(left, right[jr])
        end
        @sync begin
            let left = view(left, 1:jl),
                right = view(right, 1:jr),
                out = view(out, 1:(jl + jr))
                Threads.@spawn pmerge!(out, left, right)
            end
            let left = view(left, (jl + 1):ll),
                right = view(right, (jr + 1):lr),
                out = view(out, (jl + jr + 1):length(out))
                pmerge!(out, left, right)
           end
        end
    end
    nothing
end
```

This was tested with 8 hyperthreads on the same 4 core laptop as before on a vector `a` consisting of 20M random doubles (Float64 elements). With the parallel `pmerge!`, `@btime psort(a)` reported a runtime of 533 ms vs. 743 ms for the serial merge `merge!`!