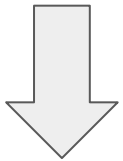
UCX.jl
Feature rich UCX bindings for Julia

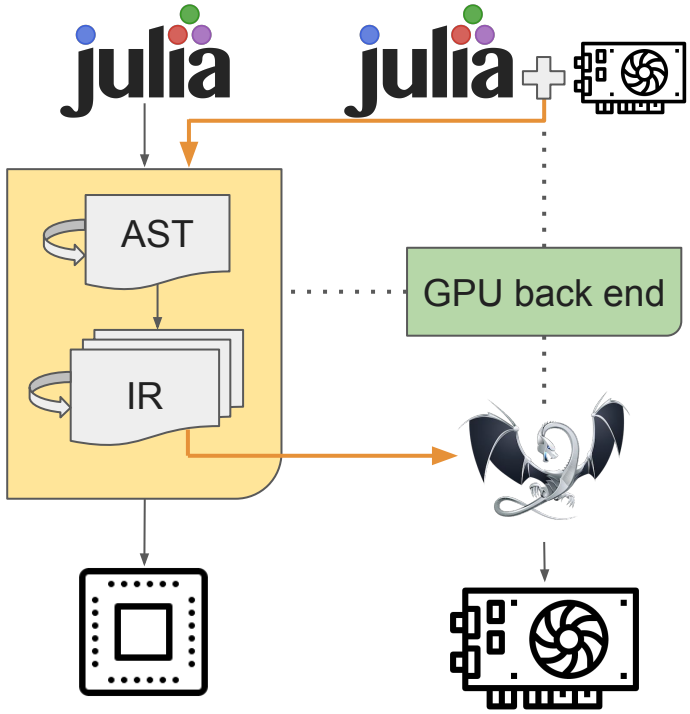
Valentin Churavy
vchuravy@mit.edu

Why Julia?

Language design

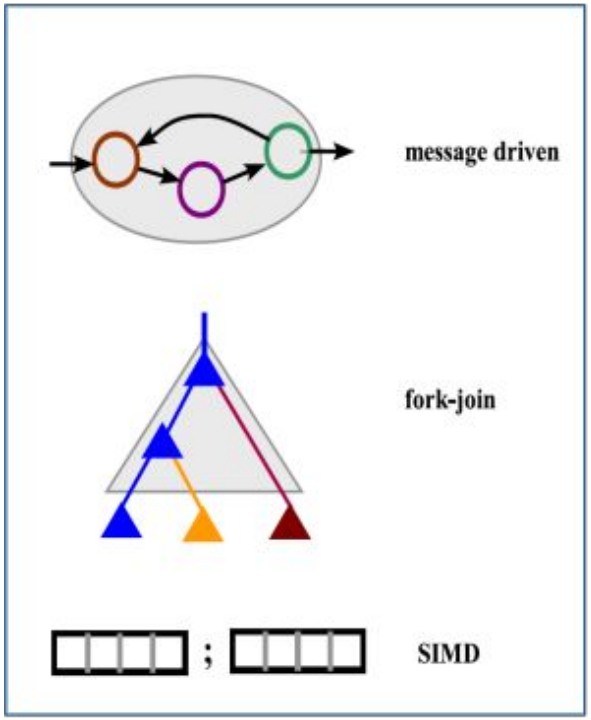


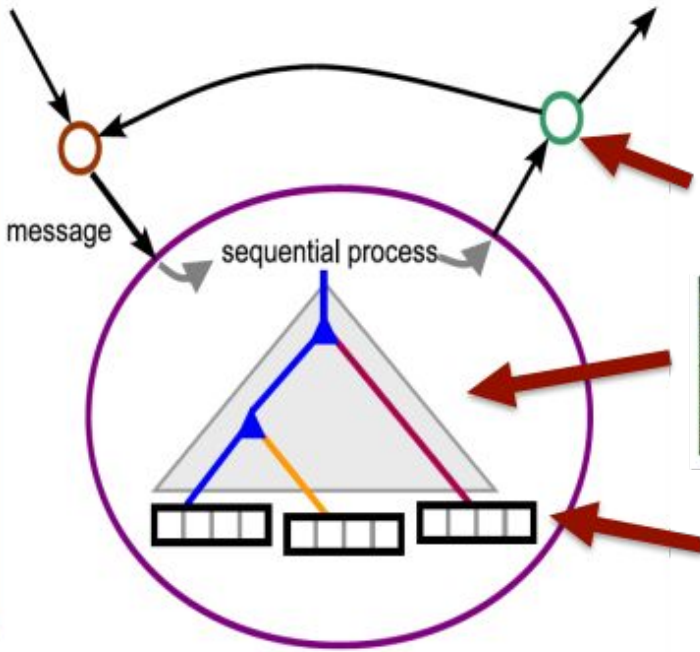
Efficient execution



Julia: Dynamism and Performance Reconciled by Design (doi:10.1145/3276490)

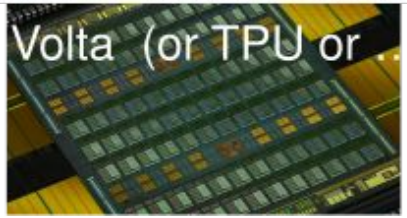
Effective Extensible Programming: Unleashing Julia on GPUs (doi:10.1109/TPDS.2018.2872064)


message driven
fork-join
;
SIMD


message
sequential process


summit
IBM
Distributed


Power9
Power9
Power9


Volta (or TPU or ...)

Parallelism in Julia

- Distributed:
  - MPI.jl
  - Distributed.jl — RPC based; primary-workers
    - Dagger.jl — similar to Dask

- Task-based parallelism:
  - @sync/@spawn

- Rich GPU support — It's Julia all the way down
  Supporting NVIDIA, AMD, Intel — vendor specific packages + common infrastructure

Features

1. Auto-generated wrappers (based on 1.10 for now) for UCS/UCP
2. High-Level support for
    a. Active Messages
    b. tag_send/tag_recv
    c. stream_send/stream_recv
3. Integration with Julia event-loop & multi-threading

Coming soon:

- High-level support for RMA and maybe AMO
- Update to 1.11

Example

```
function echo_server(ep::UCX.Endpoint)
    size = Int[0]
    wait(recv(ep, size, sizeof(Int)))

    data = Array{UInt8}(undef, size[1])
    wait(recv(ep, data, sizeof(data)))
    wait(send(ep, data, sizeof(data)))
end


function client(ep::UCX.Endpoint)
    data = "Hello world"
    req1 = send(ep, Int[sizeof(data)], sizeof(Int))
    req2 = send(ep, data, sizeof(data))
    wait(req1); wait(req2)

    buffer = Array{UInt8}(undef, sizeof(data))
    wait(recv(ep, buffer, sizeof(buffer)))
    @assert String(buffer) == data
end
```

```
function start_client(port=default_port)
    ctx    = UCX.UCXContext()
    worker = UCX.Worker(ctx)
    ep     = UCX.Endpoint(worker, IPv4("127.0.0.1"), port)

    try
        client(ep)
    finally
        close(worker)
    end
end
```

```
function start_server(port = default_port)
    ctx = UCX.UCXContext()
    worker = UCX.Worker(ctx)

    function listener_callback(::UCX.UCXListener, conn_request::UCX.UCXConnectionRequest)
        UCX.@spawn_showerr begin
            try
                echo_server(UCX.Endpoint($worker, $conn_request))
            finally
                close($worker)
            end
        end
        nothing
    end
    listener = UCX.UCXListener(worker.worker, listener_callback, port)

    GC.@preserve listener begin
        while isopen(worker)
            wait(worker)
        end
        close(worker)
    end
end
```

Integration with Julia event-loop

- Julia builds upon an LibUV event-loop
- UCX.jl should use `_nb` variants and execute other Julia code while waiting for completion

- We need to guarantee that progress is being made.
- Callbacks from UCX execute normal Julia code
- How do we get low AM latency

Progress being made

- After starting Listener user needs to call `wait` to guarantee that progress is being made automatically (akin to a background thread)

- Three progress modes:
  a. `Polling` — highest latency / lowest resource utilization
  b. `Busy` — low latency / unfair scheduling (task)
  c. `Idler` — low latency / unfair scheduling (OS)

```
@spawn begin
    while isopen(worker)
        wait(worker) # suspend task
    end
    close(worker)
end
```

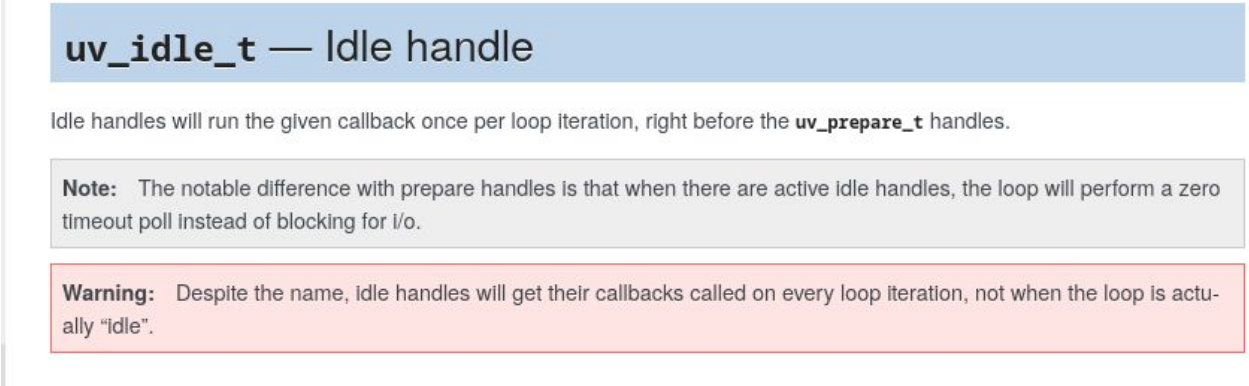
Progress mode: Polling

- Requires UCP_FEATURE_WAKEUP, but that disables SHMEM support (ucx#5322)
- Uses ucp_worker_get_efd + ucp_worker_arm and Julia FileWatching.poll_fd

https://github.com/JuliaParallel/UCX.jl/blob/d622f9b77272a458a2ee3a45fc1362030a62ccf1/src/UCX.jl#L331-L357

Progress mode: Idling

- Uses LibUV `uv_idle_t` http://docs.libuv.org/en/v1.x/idle.html
- Effectively a busy-loop on the OS level :/



`uv_idle_t` — Idle handle

Idle handles will run the given callback once per loop iteration, right before the **`uv_prepare_t`** handles.

Note: The notable difference with prepare handles is that when there are active idle handles, the loop will perform a zero timeout poll instead of blocking for i/o.

Warning: Despite the name, idle handles will get their callbacks called on every loop iteration, not when the loop is actually "idle".

Progress mode: Busy

- Does what it says on the tin — busy loop within Julia
- Could lead to pseudo-livelocks
  - Accidentally preventing other Julia tasks to run
  - What if those other Julia tasks are needed to make progress?

```
progress(worker)
while isopen(worker)
    ccall(:jl_gc_safepoint, Cvoid, ())
    yield()
    progress(worker)
end
```

Latency results

Configuration	Median	Mean	STD	System
Polling	23.455 µs	25.666 µs	6.095 µs	x86
Busy	4.260 µs	4.654 µs	1.081 µs	x86
Idler	4.120 µs	4.517 µs	1.361 µs	x86
Polling	69.301 µs	70.135 µs	4.859 µs	PPC
Busy	17.147 µs	16.511 µs	4.129 µs	PPC
Idler	16.907 µs	17.459 µs	3.097 µs	PPC

Other alternatives considered

1. Call `progress` on non-Julia thread
   a. Callbacks are Julia functions — need to interact with Julia runtime

2. Call `progress` on non-Julia thread + `uv_async_send`
   a. LibUV has a facility to inform the event loop from a foreign thread an `AsyncCondition` callback needs to be triggered.
   b. We could implement multi-stage callbacks. UCX invoked Callback, LibUV notifies Julia
   c. Not tested yet, but...
   d. Higher latency, additional allocations required
   e. Can't return status from callback to UCX proper. Needed for `am_recv_callback`

A wrinkle in the fabric: Active Messages

- Callbacks from UCX are invoked by **`progress`** and are Julia code.
- Most callbacks are “simple” and don’t trigger task-switches or allocations
  - Notify `UCXRequest` object of completion, very fast, no allocations
  - Considering adding a language level construct to guarantee that there are no runtime interactions
- Active Messages callbacks are many things, none of them is “simple”

```
progress()
  -> am_callback
    -> println
      -> task-switch
        -> ...
          -> progress()
```

UCX assumes that `progress` is not called recursively. If it is called recursively on the same thread, we will execute the currently executing callback again.

Effectively invoking an AM twice — **hilarity** ensues.

Roadmap/TODO

- Ensure that UCX+Julia Threading is well behaved

- GPU memory support — taking inspiration from MPI.jl
  - Informing UCX through `UCP_OP_ATTR_FIELD_MEMORY_TYPE`
  - In the works as UCX.jl#35, but needs to be disentangled

- Implement Distributed.jl over UCX.jl
  - Probably worth-it, but rather complex
  - Easier win: Enhance MemPool.jl/Dagger.jl to use UCX to move GPU data

Gripe: Signal handlers

Julia uses signal handlers to implement multi-threading and GC.

Have to disable it, also a problem for libraries like MPI.jl:

https://juliaparallel.github.io/MPI.jl/latest/knownissues/#UCX

```
# reinstall signal handlers
ccall((:ucs_debug_disable_signals, API.libucs), Cvoid, ())
```

```
https://github.com/JuliaParallel/UCX.jl/blob/d622f9b77272a458a2ee3
a45fc1362030a62ccf1/src/UCX.jl#L12-L21
```

Good defaults or bad defaults?

Sometimes choices that make it easier for C/C++ can make it harder for other languages: Like signal-handlers, or **dlopen** interception...

On the other hand high-level languages can have more information than C/C++:

- MEMORY_TYPE is very easy to support since we have “colored” pointers
- IIUC dlopen hijacking is used to reduce latency for memory detection

From a Julia perspective: We would like to opt-out (or better require explicit opt-in) for invasive features and collaborate with UCX

Thank you!

Happy to take contributions to UCX.jl
or talk about collaborative efforts.

vchuravy@mit.edu

Monthly Julia HPC call

Fourth Tuesday of the Month 2PM ET

Open to anyone

Agenda & Notes

Backup: Distributed.jl over UCX

VERY preliminary results