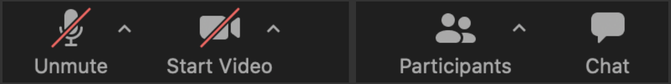
ThreadsX.jl: easier multithreading in Julia

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
  - please address chat questions to "Everyone" (not direct chat!)
- Raise your hand in Participants



- Email training@westgrid.ca

Parallel Julia

- In CC we teach programming in C, C++, Fortran, Python, R, Julia, Chapel
- Julia in WestGrid: serial and parallel courses and webinars `https://git.io/Jtdge`
- Feb-14,16,18 - upcoming "Parallel Julia" national training workshop will cover the checkboxes below - see details and register at `https://bit.ly/wg2022a`
- Today's topic: multi-threading, both on multi-core PCs and HPC clusters
- Not to be confused with Julia's multi-processing

✔ Base.Threads

✔ ThreadsX.jl

✔ Distributed.jl

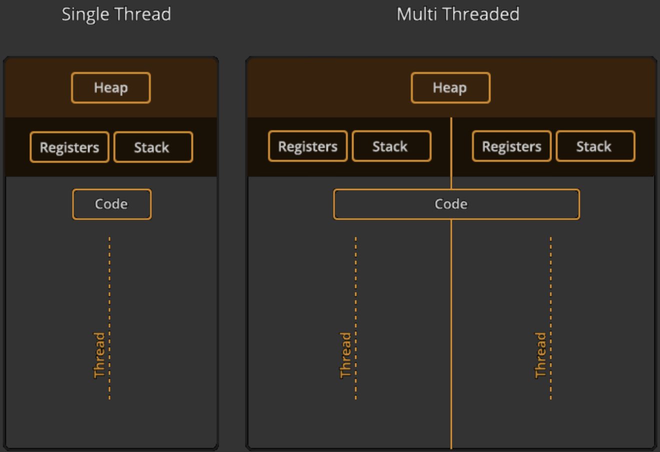
✔ DistributedArrays.jl

✔ SharedArrays.jl

- Dagger.jl
- Concurrent function calls ("lightweight threads" for suspending/resuming computations)
- MPI.jl
- MPIArrays.jl
- ClusterManagers.jl
- LoopVectorization.jl
- FLoops.jl
- Transducers.jl
- GPU-related packages

Threads vs. processes

- In Unix a process is the smallest independent unit of processing, with its own memory space – think of an instance of a running application
- A process can contain multiple threads, each running on its own CPU core, all sharing the virtual memory address space of that process ⇒ multi-threading always limited to one node



Builtin multi-threading since v1.3

Let's start Julia by typing "`julia`" in bash:

```
using Base.Threads     # otherwise will have to preface all functions/macros with 'Threads.'
nthreads()             # by default, Julia starts with a single thread of execution
```

If instead we start with "`julia -t 4`"
(or "`JULIA_NUM_THREADS=4 julia`" prior to v1.5):

```
using Base.Threads
nthreads()             # now 4 threads

@threads for i=1:10    # parallel for loop using all threads
    println("iteration $i on thread $(threadid())")
end
```

Let's compute $\sum_{i=1}^{10^6} i$ with multiple threads

- This code is not thread-safe:

```
total = 0
@threads for i = 1:1_000_000
    global total += i
end
println("total = ", total)
```

  - race condition: multiple threads updating the same variable at the same time
  - a new result every time
  - unfortunately, `@threads` does not have built-in reduction support

Let's compute $\sum_{i=1}^{10^6} i$ with multiple threads

- This code is not thread-safe:

```
total = 0
@threads for i = 1:1_000_000
    global total += i
end
println("total = ", total)
```

  - race condition: multiple threads updating the same variable at the same time
  - a new result every time
  - unfortunately, `@threads` does not have built-in reduction support

- Let's make it thread-safe (one of many solutions):

```
total = Atomic{Int64}(0)
@threads for i in 1:Int(1e6)
    atomic_add!(total, i)
end
println("total = ", total[])
```

  - this code is supposed to be much slower: threads waiting for others to finish updating the variable
  - atomic variables not really designed for this type of usage

  $\Rightarrow$ let's do some benchmarking

Benchmarking in Julia

(a) Running the loop in the global scope (without a function):

- direct summation
- @time includes JIT compilation time (marginal here)
- `total` is a global variable to the loop

```
n = Int64(1e9)
total = Int64(0)
@time for i in 1:n
    total += i
end
println("total = ", total)
# serial runtime: 51.80s 51.95s
```

Benchmarking in Julia

(a) Running the loop in the global scope (without a function):

- direct summation
- @time includes JIT compilation time (marginal here)
- `total` is a global variable to the loop

```
n = Int64(1e9)
total = Int64(0)
@time for i in 1:n
    total += i
end
println("total = ", total)
# serial runtime: 51.80s 51.95s
```

(b) Packaging the loop in the local scope of a function:

- Julia v1.6 and earlier will replace the loop with the formula $n(n+1)/2$ – we don't want this!
- v1.7 seems to be doing direct summation
- first function call results in compilation
- @time here includes only the loop runtime

```
function quick(n)
    total = Int64(0)
    @time for i in 1:n
        total += i
    end
    return(total)
end
quick(10)
println("total = ", quick(Int64(1e9)))
  # serial runtime: 0.000000s + correct result
println("total = ", quick(Int64(1e15)))
  # serial runtime: 0.000000s + incorrect result
  # due to limited Int64 precision
```

Benchmarking in Julia

(a) Running the loop in the global scope (without a function):

- direct summation
- @time includes JIT compilation time (marginal here)
- `total` is a global variable to the loop

```
n = Int64(1e9)
total = Int64(0)
@time for i in 1:n
    total += i
end
println("total = ", total)
# serial runtime: 51.80s 51.95s
```

(b) Packaging the loop in the local scope of a function:

- Julia v1.6 and earlier will replace the loop with the formula $n(n+1)/2$ – we don't want this!
- v1.7 seems to be doing direct summation
- first function call results in compilation
- @time here includes only the loop runtime

```
function quick(n)
    total = Int64(0)
    @time for i in 1:n
        total += i
    end
    return(total)
end
quick(10)
println("total = ", quick(Int64(1e9)))
  # serial runtime: 0.000000s + correct result
println("total = ", quick(Int64(1e15)))
  # serial runtime: 0.000000s + incorrect result
  # due to limited Int64 precision
```

1. force computation for any Julia version $\Rightarrow$ compute something more complex than simple integer summation
2. exclude compilation time, make use of optimizations for type stability $\Rightarrow$ package into a function + precompile it
3. time only the CPU-intensive loops
4. for shorter runs (ms - few seconds) use `@btime` from BenchmarkTools

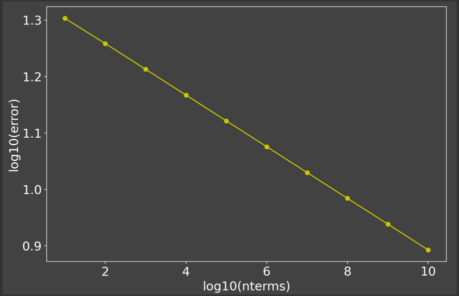
Slowly convergent series

- The traditional harmonic series $\sum_{k=1}^{\infty} \frac{1}{k}$ diverges
- However, if we omit the terms whose denominators in decimal notation contain any digit or string of digits, it converges, albeit very slowly (Schmelzer & Baillie 2008), e.g.

$$\sum_{\substack{k=1\\ \text{no “9”}}}^{\infty} \frac{1}{k} = 22.9206766192\ldots$$

$$\sum_{\substack{k=1\\ \text{no even digits}}}^{\infty} \frac{1}{k} = 3.1717654734\ldots$$

$$\sum_{\substack{k=1\\ \text{no string “314”}}}^{\infty} \frac{1}{k} = 2299.8297827675\ldots$$



- For no denominators with “9”, assuming linear convergence in the log-log space, we would need 10^{73} terms to reach 22.92, and almost 10^{205} terms to reach 22.92067661

Checking for substrings in Julia

- Checking for a substring is one possibility

```
if !occursin("9", string(i))
    <add the term>
end
```

- Integer exclusion is ∼4X faster (thanks to Paul Schrimpf from the Vancouver School of Economics @UBC)

```
function digitsin(digits::Int, num)   # decimal representation of `digits` has N digits
    base = 10
    while (digits ÷ base > 0)         # `digits ÷ base` is same as `floor(Int, digits/base)`
        base *= 10
    end
    # `base` is now the first Int power of 10 above `digits`, used to pick last N digits from `num`
    while num > 0
        if (num % base) == digits     # last N digits in `num` == digits
            return true
        end
        num ÷= 10                     # remove the last digit from `num`
    end
    return false
end
if !digitsin(9, i)
    <add the term>
end
```

Timing the summation: serial code

- Let's switch to 10^9 terms, start with the serial code:

```
using BenchmarkTools
function slow(n::Int64, digits::Int)
    total = Int64(0)
    for i in 1:n
        if !digitsin(digits, i)
            total += 1.0 / i
        end
    end
    return total
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)   # total = 14.2419130103833
```

```
$ julia serial.jl   # serial runtime: 5.214 s
```

Timing the summation: using an atomic variable

- Threads are waiting for the atomic variable to be released $\Rightarrow$ should be slow:

```
using BenchmarkTools, Base.Threads
function slow(n::Int64, digits::Int)
    total = Atomic{Float64}(0)
    @threads for i in 1:n
        if !digitsin(digits, i)
            atomic_add!(total, 1.0 / i)
        end
    end
    return total[]
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)    # total = 14.241913010383293
```

```
$ julia atomicThreads.jl        # runtime with 1 thread:  5.996 s
$ julia -t 8 atomicThreads.jl   # runtime with 8 threads: 37.759 s
```

Timing the summation: an alternative thread-safe implementation

- Each thread is updating its own sum, no waiting $\Rightarrow$ should be faster:

```
using BenchmarkTools, Base.Threads
function slow(n::Int64, digits::Int)
    total = zeros(Float64, nthreads())
    @threads for i in 1:n
        if !digitsin(digits, i)
            total[threadid()] += 1.0 / i
        end
    end
    return sum(total)
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)   # total = 14.241913010384215
```

```
$ julia separateSums.jl          # runtime with 1 thread:  5.262 s
$ julia -t 8 separateSums.jl     # runtime with 8 threads: 3.823 s
```

Timing the summation: fixing the *false sharing effect* in the last code

- Cache lines (~32-128 bytes in size) are chunks of memory handled by the cache
- Problem arises when several threads are writing into variables placed close enough to each other to end up in the same cache line (thanks to Pierre Fortin for pointing out this problem)

```
using BenchmarkTools, Base.Threads
function slow(n::Int64, digits::Int)
    space = 8   # assume a 64-byte cache line, hence 8 Float64 elements per cache line
    total = zeros(Float64, nthreads()*space)
    @threads for i in 1:n
        if !digitsin(digits, i)
            total[threadid()*space] += 1.0 / i
        end
    end
    return sum(total)
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)   # total = 14.241913010384215
```

```
$ julia spacedSeparateSums.jl        # runtime with 1 thread:  5.291 s
$ julia -t 8 spacedSeparateSums.jl   # runtime with 8 threads: 914.502 ms
```

Timing the summation: using heavy loops

- Another fast implementation:

```
using BenchmarkTools, Base.Threads
function slow(n::Int64, digits::Int)
    space = 8   # assume a 64-byte cache line, hence 8 Float64 elements per cache line
    numthreads = nthreads()
    threadSize = floor(Int64, n/numthreads)   # number of terms per thread (except last thread)
    total = zeros(Float64, numthreads*space);
    @threads for threadid in 1:numthreads
        local start = (threadid-1)*threadSize + 1
        local finish = threadid < numthreads ? (threadid-1)*threadSize+threadSize : n
        # println("thread $threadid: from $start to $finish");
        for i in start:finish
            if !digitsin(digits, i)
                total[threadid*space] += 1.0 / i
            end
        end
    end
    return sum(total)
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)   # total = 14.2419130103833
```

```
$ julia heavyThreads.jl         # runtime with 1 thread:  5.296 s
$ julia -t 8 heavyThreads.jl    # runtime with 8 threads: 914.541 ms
```

ThreadsX.jl

`https://github.com/tkf/ThreadsX.jl`

- With Base.Threads you can manually add multi-threaded reduction
  - solutions are somewhat awkward
  - inadvertently you can run into problems (thread safety, false sharing, other performance issues)
- Enter ThreadsX: parallelized subset of Base functions

```
using ThreadsX
ThreadsX.<TAB>
?ThreadsX.mapreduce
?mapreduce
```

- Consider Base function:

```
mapreduce(x->x^2, +, 1:10)
```

ThreadsX.jl

https://github.com/tkf/ThreadsX.jl

- With Base.Threads you can manually add multi-threaded reduction
  - solutions are somewhat awkward
  - inadvertently you can run into problems (thread safety, false sharing, other performance issues)
- Enter ThreadsX: parallelized subset of Base functions

```
using ThreadsX
ThreadsX.<TAB>
?ThreadsX.mapreduce
?mapreduce
```

- Consider Base function:

```
mapreduce(x->x^2, +, 1:10)
```

- Alternative syntax:

```
mapreduce(+,1:10) do i
    i^2   # plays the role of the function applied to each element
end
```

- Back to our slow series:

```
using BenchmarkTools, ThreadsX
function slow(n::Int64, digits::Int)
    total = ThreadsX.mapreduce(+,1:n) do i
        if !digitsin(digits, i)
            1.0 / i
        else
            0.0
        end
    end
    return total
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)    # total = 14.241913010384215
```

```
$ julia mapreduce.jl          # runtime with 1 thread: 5.255 s
$ julia -t 8 mapreduce.jl     # runtime with 8 threads: 900.995 ms
```

- In compact notation:

```julia
using BenchmarkTools, ThreadsX
function slow(n::Int64, digits::Int)
    total = ThreadsX.mapreduce(+,1:n) do i
        !digitsin(digits, i) ? 1.0 / i : 0
    end
    return total
end

total = @btime slow(Int64(1e9), 9)
println("total = ", total)    # total = 14.241913010384215
```

```
$ julia mapreduceCompact.jl          # runtime with 1 thread: 5.267 s
$ julia -t 8 mapreduceCompact.jl     # runtime with 8 threads: 914.470 ms
```

- Replacing `ThreadsX.mapreduce` with `mapreduce` above will give you a serial code

Base.sum → ThreadsX.sum

```
?sum
?Threads.sum
```

- The expression in the round brackets is a generator:

```
(i for i in 1:10)
collect(i for i in 1:10)    # construct a vector
collect(!digitsin(9, i) ? 1.0/i : 0 for i in 1:10)
[!digitsin(9, i) ? 1.0/i : 0 for i in 1:10]    # functionally the same
```

- How about the following one-liner:

```
using BenchmarkTools
@btime sum(!digitsin(9, i) ? 1.0/i : 0 for i in 1:1_000_000_000)
   # serial code: 5.061 s, prints 14.2419130103833
```

- Easy to parallelize:

```
using BenchmarkTools, ThreadsX
@btime ThreadsX.sum(!digitsin(9, i) ? 1.0/i : 0 for i in 1:1_000_000_000)
   # with 8 threads: 906.420 ms, prints 14.241913010381973
```

Alternative syntaxes

- Sum terms returned by a generator that only produces non-zero terms

```
@btime ThreadsX.sum(1.0/i for i in 1:1_000_000_000 if !digitsin(9, i))
    # with 8 threads: 888.853 ms, prints 14.241913010381973
```

- Sum the results of applying a function to all integers in a range

```
function numericTerm(i)
    !digitsin(9, i) ? 1.0/i : 0
end
@btime ThreadsX.sum(numericTerm, 1:Int64(1e9))              # 890.466 ms, same result
@btime ThreadsX.mapreduce(numericTerm, +, 1:Int64(1e9))     # 912.552 ms, same result
```

Sorting

Sorting is intrinsically hard to parallelize $\Rightarrow$ do not expect 100% parallel efficiency:

```
n = Int64(1e8)
r = rand(Float32, (n));
r[1:10]         # first 20 elements, same as first(r,10)
last(r,10)      # last 10 elements

?sort                  # underneath uses QuickSort (for numeric arrays) or MergeSort
@btime sort!(r);       # 1.391 s, serial sorting

r = rand(Float32, (n));
@btime ThreadsX.sort!(r);     # 586.541 ms, parallel sorting
?ThreadsX.sort!               # there is actually a good manual page

# similar speedup for integers
r = rand(Int32, (n));
@btime sort!(r);     # 889.817 ms

r = rand(Int32, (n));
@btime ThreadsX.sort!(r);     # 390.082 ms
```

Searching for extrema

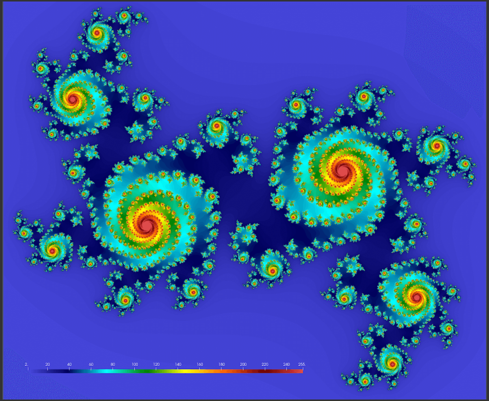
Searching for extrema is much more parallel-friendly:

```
n = Int64(1e9)
r = rand(Int32, (n));          # make sure we have enough memory
@btime maximum(r)              # 288.200 ms
@btime ThreadsX.maximum(r)     # 31.879 ms
```

Julia set (no relation to Julia language!)

A set of points on the complex plane that remain bound under infinite recursive transformation $f(z)$. We will use the traditional form $f(z) = z^2 + c$, where c is a complex constant.

1. pick a point $z_0 \in \mathbb{C}$
2. compute iterations $z_{i+1} = z_i^2 + c$ until $|z_i| > 4$
3. $\xi(z_0)$ is the iteration number at which $|z_i| > 4$
4. limit max iterations at 255
   - $\xi(z_0) = 255 \;\Rightarrow\; z_0$ is a stable point
   - the quicker a point diverges, the lower its $\xi(z_0)$ is
5. plot $\xi(z_0)$ for all z_0 in a rectangular region $-1 <= \mathfrak{Re}(z_0) <= 1$, $-1 <= \mathfrak{Im}(z_0) <= 1$



$c = 0.355 + 0.355i$

For different c we will get very different fractals. Try $-0.4 - 0.59i$, $1.34 - 0.45i$, $0.34 - 0.05i$

Demo: computing and plotting the Julia set for $c = 0.355 + 0.355i$

Code for presenter in `juliaSet/juliaSetSerial1.jl`

```julia
using BenchmarkTools, Plots

function pixel(z)
    c = 0.355 + 0.355im
    z *= 1.2    # zoom out
    for i = 1:255
        z = z^2 + c
        if abs(z) >= 4
            return i
        end
    end
    return 255
end

n = 2_000
height, width = n, n
```

```julia
println("Computing Julia set ...")
point = zeros(Complex{Float32}, height, width);
stability = zeros(Int32, height, width);
for i in 1:height, j in 1:width
    # rescale to -1:1 in the complex plane
    point[i,j] = (2*(j-0.5)/width-1) + (2*(i-0.5)/height-1)im
end
@btime for i in 1:height, j in 1:width
    stability[i,j] = pixel(point[i,j])
end

println("Plotting to PNG ...")
gr()     # initialize the gr backend
fname = "$(height)x$(width)"
png(heatmap(stability, size=(width,height), color=:gist_ncar), fname)
```

```
$ julia juliaSetSerial1.jl   # 1.160 s
```

- The reason for two $n \times n$ arrays will be explained later

Parallelizing the Julia set with Base.Threads

```diff
-using BenchmarkTools, Plots
+using Base.Threads, BenchmarkTools, Plots
```

Parallelizing the Julia set with Base.Threads

```
-using BenchmarkTools, Plots
+using Base.Threads, BenchmarkTools, Plots

-@btime for i in 1:height, j in 1:width
-    stability[i,j] = pixel(point[i,j])
-end
+@btime @threads for i in 1:height
+    for j in 1:width
+        stability[i,j] = pixel(point[i,j])
+    end
+end
```

```
julia -t 8 juliaSetThreaded1.jl   # 249.924 ms with 8 threads
```

How do we parallelize with ThreadsX? We want to process an array without reduction

How do we parallelize with ThreadsX? We want to process an array without reduction

Let’s first modify the serial code! We’ll use another function from Base library:

```
?map
map(x -> x * 2, [1, 2, 3])
map(+, [1, 2, 3], [10, 20, 30, 400, 5000])
```

How do we parallelize with ThreadsX? We want to process an array without reduction

Let's first modify the serial code! We'll use another function from Base library:

```
?map
map(x -> x * 2, [1, 2, 3])
map(+, [1, 2, 3], [10, 20, 30, 400, 5000])
```

```
-@btime for i in 1:height, j in 1:width
-    stability[i,j] = pixel(point[i,j])
-end
+stability = @btime map(pixel, point);
```

```
julia juliaSetSerial2.jl   # 917.683 ms
```

Parallelizing the Julia set with ThreadsX

```
-using BenchmarkTools, Plots
+using ThreadsX, BenchmarkTools, Plots

-stability = @btime map(pixel, point);
+stability = @btime ThreadsX.map(pixel, point);
```

```
julia -t 8 juliaSetThreaded2.jl   # 171.010 ms with 8 threads
```

Running on a cluster

```
#!/bin/bash
#SBATCH --mem-per-cpu=3600M
#SBATCH --time=00:10:00
#SBATCH --account=def-user
module load julia
julia juliaSetSerial1.jl
```

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=...
#SBATCH --mem-per-cpu=3600M
#SBATCH --time=00:10:00
#SBATCH --account=def-user
module load julia
julia -t $SLURM_CPUS_PER_TASK juliaSetThreaded2.jl
```

- Runtime 2.467 s (serial) and 180.003 ms (16 cores) on Cedar with `julia/1.7.0`
- By default, packages will be installed in `$HOME/.julia`
- You can install them elsewhere

```
empty!(DEPOT_PATH)
push!(DEPOT_PATH,"/scratch/path/to/julia")
] add BenchmarkTools
```

and then at runtime (double-check the syntax online!)

```
module load julia
export JULIA_DEPOT_PATH=/home/\$USER/.julia:/scratch/path/to/julia
export JULIA_LOAD_PATH=@:@v#.#:@stdlib:/scratch/path/to/julia
```

Summary

- ThreadsX.jl is a super easy way to parallelize some of the Base library functions
  - includes multi-threaded reduction
  - very impressive performance
- ThreadsX.jl will likely be incorporated into the main language in the not-too-distant future
- To list the supported functions, use `ThreadsX.<TAB>`
- Some of the functions are well-documented: `?ThreadsX.<function>`
- For others check its Base equivalent's documentation: `?<function>`
- Feb-14,16,18 - upcoming "Parallel Julia" national training workshop
  - three 3-hour sessions, many hands-on exercises
  - both multi-threading and multi-processing
  - working with shared and distributed arrays
  - link to register at `https://bit.ly/wg2022a`

Questions?