Parallel programming in Julia

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

MARIE-HÉLÈNE BURLE
marie.burle@westgrid.ca

Zoom controls

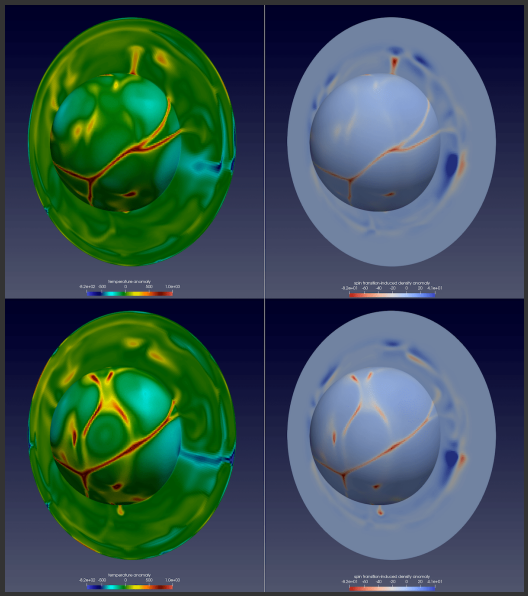
- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
  - please address chat questions to "Everyone" (not direct chat!)
- Raise your hand in Participants

- Email training@westgrid.ca

2021 IEEE Vis Contest

https://scivis2021.netlify.app

- Co-hosting 2021 SciVis Contest with IEEE Vis
- Dataset: 3D simulation of Earth's mantle convection covering 500 Myrs of geological time
- Contest is open to anyone (no research affiliation necessary), dataset available now
- Wanted: visualizations + problem-specific analysis of descending / rising flows
- Opportunity to present at IEEE Vis 2021
- July 31, 2021 - deadline for Contest entry submissions



WestGrid Training Modules 2021

https://wgtm21.netlify.app

- Weekly April 27th to July 27th
- 13 full-day research computing courses, grouped into 7 modules
- Register for individual modules
- Parallel coding module June 1st and 8th

Parallel Julia

- We teach serial and parallel full-day Julia courses
- Today's topic: what unique features does Julia bring to parallel programming?
- Targeting both multi-core PCs and distributed-memory clusters

- ✔ Base.Threads
- ✔ Distributed.jl
- ✔ ClusterManagers.jl
- ✔ DistributedArrays.jl
- ✔ SharedArrays.jl

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

...

Multi-threading

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 1.5):

```
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

a = zeros(10)
@threads for i=1:10
    a[i] = threadid() # should be no collision: each thread writes to its own part
end
a
```

Filling an array: perfect parallel scaling[1]

@threads are well-suited for shared-memory data parallelism without any reduction

```
nthreads()   # still running 4 threads

n = Int64(1e9)
a = zeros(n);

@time for i in 1:n
    a[i] = log10(i)
end
# runtime 129.03s 125.66s 125.60s

using Base.Threads
@time @threads for i in 1:n
    a[i] = log10(i)
end
# runtime 36.99s 25.75s 30.33s (4X speedup)
```

[1] Whether I am doing this inside or outside a function is not the point here ... besides, you don't know (more on this in slide 9)

Let's add reduction: summation $\sum_{i=1}^{10^6} i$ via threads

- This code is not thread-safe:

```
total = 0
@threads for i = 1:Int(1e6)
    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 add reduction: summation $\sum_{i=1}^{10^6} i$ via threads

- This code is not thread-safe:

```
total = 0
@threads for i = 1:Int(1e6)
    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

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: 92.72s 92.75s 91.05s
```

Packaging the loop in the local scope of a function:

- Julia replaces the loop with the formula $n(n+1)/2$ ✘ we don't want this!
- first function call results in compilation, ✔ @time here includes only 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 ⇒ compute something more complex than simple integer summation
2. exclude compilation time ⇒ package into a function + precompile it
3. make use of optimizations for type stability and other factors ⇒ package into a function
4. time only the CPU-intensive loops
5. for shorter runs (ms) may want to use `@btime` from BenchmarkTools

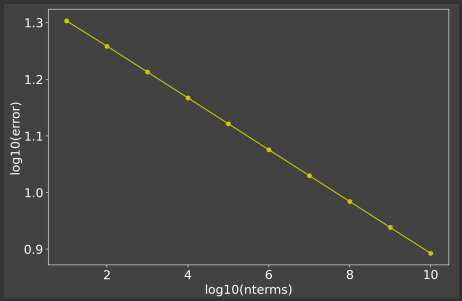
Slowly convergent series

- 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:

```
function slow(n::Int64, digits::Int)
    total = Int64(0)
    @time for i in 1:n
        if !digitsin(digits, i)
            total += 1.0 / i
        end
    end
    println("total = ", total)
end

slow(10, 9)
slow(Int64(1e9), 9)    # total = 14.2419130103833
```

```
$ julia serial.jl   # serial runtime: 22.00s 21.85s 22.03s
```

Timing the summation: using an atomic variable

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

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

slow(10, 9)
slow(Int64(1e9), 9)    # total = 14.2419130103833
```

```
$ julia atomicThreads.jl        # runtime on 1 thread:  25.66s 26.56s 27.26s
$ julia -t 4 atomicThreads.jl   # runtime on 4 threads: 17.35s 18.33s 18.86s
```

Timing the summation: an alternative thread-safe implementation

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

```
using Base.Threads
function slow(n::Int64, digits::Int)
    total = zeros(Float64, nthreads())
    @time @threads for i in 1:n
        if !digitsin(digits, i)
            total[threadid()] += 1.0 / i
        end
    end
    println("total = ", sum(total))
end
slow(10, 9)
slow(Int64(1e9), 9)    # total = 14.2419130103833
```

```
$ julia separateSums.jl        # runtime on 1 thread:  24.20s 24.52s 23.94s
$ julia -t 4 separateSums.jl   # runtime on 4 threads: 10.71s 10.81s 10.72s
```

Timing the summation: using heavy loops

- Might be the fastest of the three parallel implementations:

```
using Base.Threads
function slow(n::Int64, digits::Int)
    numthreads = nthreads()
    threadSize = floor(Int64, n/numthreads)    # number of terms per thread (except last thread)
    total = zeros(Float64, numthreads);
    @time @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] += 1.0 / i
            end
        end
    end
    println("total = ", sum(total))
end
slow(10, 9)
slow(Int64(1e9), 9)    # total = 14.2419130103833
```

```
$ julia heavyThreads.jl         # runtime on 1 thread:  24.05s 24.67s 24.75s
$ julia -t 4 heavyThreads.jl    # runtime on 4 threads: 9.93s 10.21s 10.24s
```

Timing the summation: using heavy loops (cont.)

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=...
#SBATCH --mem-per-cpu=3600M
#SBATCH --time=00:10:00
module load StdEnv/2020 julia/1.5.2
julia -t $SLURM_CPUS_PER_TASK heavyThreads.jl
```

Cedar (avg. over 3 runs):

code	computing
serial	47.8s
2 cores	27.5s
4 cores	15.9s
8 cores	18.5s
16 cores	8.9s

Parallelizing with multiple Unix processes (MPI tasks)

- Distributed provides multiprocessing environment to allow programs to run on multiple processors in shared or distributed memory
- Julia's implementation of message passing is one-sided, typically with higher-level operations like calls to user functions on a remote process
  - a remote call is a request by one processor to call a function on another processor; returns a remote/future reference
  - the processor that made the call proceeds to its next operation while the remote call is computing
  - you can obtain the remote result with fetch()
- Single control process + multiple worker processes
- Processes pass information via messages underneath, not via shared memory

Launching worker processes

1. From the terminal

```
$ julia -p 8              # open REPL, start Julia control process + 8 worker processes
$ julia -p 8 code.jl      # run the code with Julia control process + 8 worker processes
```

2. From a job submission script

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=3600M
#SBATCH --time=00:10:00
srun hostname -s > hostfile    # parallel I/O
sleep 5
julia --machine-file ./hostfile ./code.jl
```

- All three methods launch workers $\Rightarrow$ combining them will result in 16 (or 24!) workers
- Select one method and use it

3. From Julia

```
using Distributed
addprocs(8)
```

Important: use either (1) or (3) with Slurm on CC clusters as well: usually no need for a machine file

Process control

Let's start Julia with "`julia`" (single control process):

```
using Distributed
addprocs(4)        # add 4 worker processes

println("number of cores = ", nprocs())       # 5 cores
println("number of workers = ", nworkers())   # 4 workers
workers()                                     # list worker IDs

rmprocs(2, 3, waitfor=0)   # remove processes 2 and 3 immediately
workers()

for i in workers()       # remove all workers
    t = rmprocs(i, waitfor=0)
    wait(t)   # wait for this operation to finish
end
workers()

interrupt()   # will do the same (remove all workers)
addprocs(4)   # add 4 new worker processes (notice the new IDs!)
```

@everywhere

Let's restart Julia with "`julia`" (single control process):

```
using Distributed
addprocs(4)        # add 4 worker processes

@everywhere function showid()    # define the function everywhere
    println("my id = ", myid())
end
showid()                         # run the function on the control process
@everywhere showid()             # run the function on the control process + all workers

x = 5     # local (control process only)
@everywhere println(x)     # get errors: x is not defined elsewhere
                           # @everywhere does not capture any local variables,
                           #                 unlike @spawnat in the next slide

@everywhere println($x)    # use the value of `x` from the control process
```

@spawnat

```
a=12
@spawnat 2 println(a)      # will print 12 from worker 2
```

What `@spawnat` does here:

1. pass the namespace of local variables to worker 2
2. spawn function execution on worker 2
3. return a Future handle (referencing this running instance) to the control process
4. return REPL to the control process (while the function is running on worker 2)

@spawnat

```
a=12
@spawnat 2 println(a)      # will print 12 from worker 2
```

What `@spawnat` does here:

1. pass the namespace of local variables to worker 2
2. spawn function execution on worker 2
3. return a Future handle (referencing this running instance) to the control process
4. return REPL to the control process (while the function is running on worker 2)

```
a = 12
@spawnat 2 a+10           # Future returned but no visible calculation

r = @spawnat 2 a+10
typeof(r)
fetch(r)                  # get the result from the remote function; this will pause
                          #         the control process until the result is returned

fetch(@spawnat 2 a+10)    # combine both in one line; the control process will pause
@fetchfrom 2 a+10         # shorter notation; exactly the same as the previous command

r = @spawnat :any log10(a)    # start running on one of the workers
fetch(r)
```

Back to the slow series

Let's restart Julia with "`julia -p 2`" (control process + 2 workers):

```
using Distributed

@everywhere function digitsin(digits::Int, num)
    base = 10
    while (digits ÷ base > 0)
        base *= 10
    end
    while num > 0
        if (num % base) == digits
            return true
        end
        num ÷= 10
    end
    return false
end
```

```
@everywhere function slow(n::Int64, digits::Int)
    total = Int64(0)
    @time for i in 1:n
        if !digitsin(digits, i)
            total += 1.0 / i
        end
    end
    println("total = ", total)
end
```

```
slow(10, 9)
slow(Int64(1e9), 9)                # serial run: total = 14.2419130103833, 25.0s 24.7s 26.2s
@everywhere slow(Int64(1e9), 9)    # runs on 3 (control + 2 workers) cores simultaneously, 32.9s+32.6s+32.7s,
                                   # (with ~33s wallclock time) but each core performs the same calculation ...
```

Back to the slow series

Let's restart Julia with "`julia -p 2`" (control process + 2 workers):

```
using Distributed

@everywhere function digitsin(digits::Int, num)
    base = 10
    while (digits ÷ base > 0)
        base *= 10
    end
    while num > 0
        if (num % base) == digits
            return true
        end
        num ÷= 10
    end
    return false
end
```

```
@everywhere function slow(n::Int64, digits::Int)
    total = Int64(0)
    @time for i in 1:n
        if !digitsin(digits, i)
            total += 1.0 / i
        end
    end
    println("total = ", total)
end
```

```
slow(10, 9)
slow(Int64(1e9), 9)                # serial run: total = 14.2419130103833, 25.0s 24.7s 26.2s
@everywhere slow(Int64(1e9), 9)    # runs on 3 (control + 2 workers) cores simultaneously, 32.9s+32.6s+32.7s,
                                   # (with ~33s wallclock time) but each core performs the same calculation ...
```

Question: how long will the following code (last line) take?

```
addprocs(2)   # for the total of 4 workers
>>> redefine digitsin() and slow() everywhere
@everywhere slow(Int64(1e9), 9)
```

Parallelizing our slow series

Let's restart Julia with "`julia`" (single control process):

```
using Distributed
addprocs(2)       # add 2 worker processes
workers()

>>> redefine digitsin() everywhere

@everywhere function slow(n::Int, digits::Int, taskid, ntasks)   # two additional arguments
    println("running on worker ", myid())
    total = 0.
    @time for i in taskid:ntasks:n   # partial sum with a stride `ntasks`
        if !digitsin(digits, i)
            total += 1. / i
        end
    end
    return(total)
end

a = @spawnat :any slow(Int64(1e9), 9, 1, 2)
b = @spawnat :any slow(Int64(1e9), 9, 2, 2)
print("total = ", fetch(a) + fetch(b))   # 14.241913010372754, simultaneous 11.57s+12.90s
```

Parallelizing our slow series

Let's restart Julia with "`julia`" (single control process):

```
using Distributed
addprocs(2)       # add 2 worker processes
workers()

>>> redefine digitsin() everywhere

@everywhere function slow(n::Int, digits::Int, taskid, ntasks)   # two additional arguments
    println("running on worker ", myid())
    total = 0.
    @time for i in taskid:ntasks:n   # partial sum with a stride `ntasks`
        if !digitsin(digits, i)
            total += 1. / i
        end
    end
    return(total)
end

a = @spawnat :any slow(Int64(1e9), 9, 1, 2)
b = @spawnat :any slow(Int64(1e9), 9, 2, 2)
print("total = ", fetch(a) + fetch(b))   # 14.241913010372754, simultaneous 11.57s+12.90s
```

- 2X speedup!
- Different order of summation $\Rightarrow$ slightly different numerical result
- Not scalable: only limited to a small number of sums each spawned with its own Future reference

Solution 1: use an array of Future references

We could create an array (using *array comprehension*) of Future references and then up add their respective results:

```
r = [@spawnat p slow(Int64(1e9), 9, i, nworkers()) for (i,p) in enumerate(workers())]
print("total = ", sum([fetch(r[i]) for i in 1:nworkers()]))
# runtime with 2 simultaneous processes: 10.26+12.11s
```

Solution 2: parallel `for` loop with summation reduction

There is actually a simpler solution:

```
using Distributed
addprocs(2)

@everywhere function digitsin(digits::Int, num)
    base = 10
    while (digits ÷ base > 0)
        base *= 10
    end
    while num > 0
        if (num % base) == digits
            return true
        end
        num ÷= 10
    end
    return false
end
```

```
function slow(n::Int64, digits::Int)
    @time total = @distributed (+) for i in 1:n
        !digitsin(digits, i) ? 1.0 / i : 0
    end
    println("total = ", total);
end

slow(10, 9)
slow(Int64(1e9), 9)    # total = 14.241913010399013
```

```
$ julia parallelFor.jl # with 2 processes: 10.82s 11.34s 11.40s
$ julia parallelFor.jl # with 4 processes: 9.48s 10.37s 9.62s (changing to addprocs(4))
```

Parallel `for` on Cedar

```
#SBATCH --ntasks=...    # number of MPI tasks
#SBATCH --cpus-per-task=1
#SBATCH --nodes=1-1    # change process distribution across nodes
#SBATCH --mem-per-cpu=3600M
#SBATCH --time=0:5:0
#SBATCH --account=...
module load StdEnv/2020 julia/1.5.2
echo $SLURM_NODELIST
# comment out addprocs() in the code
julia -p $SLURM_NTASKS parallelFor.jl
```

Cedar (avg. over 3 runs):

code	computing
serial	48.2s
2 cores, same node	42.8s
4 cores, same node	12.2s
8 cores, same node	7.6s
16 cores, same node	6.8s
32 cores, same node	2.0s
32 cores across 6 nodes	11.3s

Solution 3: use `pmap` to map arguments to processes

```
using Distributed
addprocs(2)

@everywhere function digitsin(digits::Int, num)
    base = 10
    while (digits ÷ base > 0)
        base *= 10
    end
    while num > 0
        if (num % base) == digits
            return true
        end
        num ÷= 10
    end
    return false
end
```

```
@everywhere function slow((n, digits, taskid, ntasks))
                                # the argument is now a tuple
    println("running on worker ", myid())
    total = 0.0
    for i in taskid:ntasks:n   # partial sum
        if !digitsin(digits, i)
            total += 1.0 / i
        end
    end
    return(total)
end

slow((10, 9, 1, 1))
   # package arguments in a tuple
nw = nworkers()
args = [(Int64(1e9),9,j,nw) for j in 1:nw]
   # array of tuples to be mapped to workers
println("total = ", sum(pmap(slow, args)))
   # launches the function on each worker

sum(pmap(x->slow(x), args)) # alternative syntax
```

Optional integration with Slurm

`https://github.com/JuliaParallel/ClusterManagers.jl`

- To integrate Slurm launcher/flags into your Julia code
- Convenience, but not a necessity

DistributedArrays

- `] add DistributedArrays`
- A `DArray` is split across several processes (set of workers), either on the same or multiple nodes
  - this allows use of arrays that are too large to fit in memory on one node
  - each process operates on the part of the array it owns $\Rightarrow$ very natural way to achieve parallelism for large problems
- Each worker can read any elements using their global indices
- Each worker can write only to the part that it owns $\Rightarrow$ automatic parallelism and safe execution

DistributedArrays (cont.)

Code for presenter in `learning/distributedArrays.jl`

```
using Distributed
addprocs(4)
@everywhere using DistributedArrays

n = 10
data = dzeros(Float32, n, n);          # distributed 2D array of 0's

data                                   # can access the entire array
data[1,1], data[n,5]                   # can use global indices
data.dims                              # global dimensions (10, 10)
data[1,1] = 1.0                        # error: cannot write from the control process!
@spawnat 2 data.localpart[1,1] = 1.5   # success: can write locally

for i in workers()
    @spawnat i println(localindices(data))
end

@everywhere function fillLocalBlock(data)
    h, w = localindices(data)
    for i in collect(h)
        iLocal = i - h.start + 1
        for j in collect(w)
            jLocal = j - w.start + 1
            data.localpart[iLocal,jLocal] = i + j
        end
    end
end
```

```
for i in workers()
    @spawnat i fillLocalBlock(data)
end
data   # now the distributed array is filled
@fetchfrom 2 data.localpart   # stored on worker 2
minimum(data), maximum(data)  # parallel reduction
```

One-liners to generate distributed arrays

```
dzeros(100,100,100)       # 100^3 distributed array of 0's
dones(100,100,100)        # 100^3 distributed array of 1's
drand(100,100,100)        # 100^3 uniform [0,1]
drandn(100,100,100)       # 100^3 drawn from a Gaussian distribution
dfill(1.5,100,100,100)    # 100^3 fixed value

dzeros((10,10), workers()[1:4], [2,2])   # 10^2, use first 4 workers,
                                         # 2x2 array decomposition across processes

e = fill(1.5, (10,10))    # local array
de = distribute(e)        # distribute `e` across all workers
```

Building a distributed array from local pieces

Example adapted from Baolai Ge's presentation

Let's restart Julia with "`julia`" (single control process):

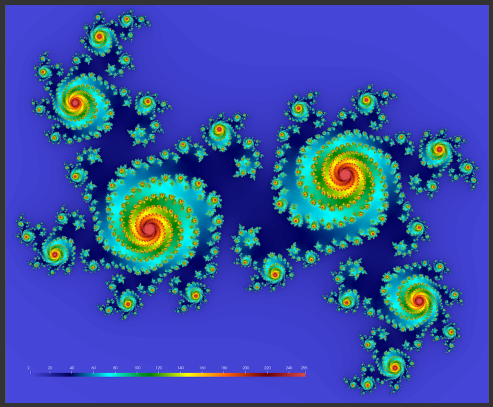
```
using Distributed
addprocs(4)
using DistributedArrays       # important to load this after addprocs()
@everywhere using LinearAlgebra
@everywhere function tridiagonal(n)
    la = zeros(n,n)
    la[diagind(la,0)] .= 2.   # diagind(la,k) provides indices of the kth diagonal of a matrix
    la[diagind(la,1)] .= -1.
    la[diagind(la,-1)] .= -1.
    return la
end
@everywhere function upperRight(n)
    la = zeros(n,n)
    la[n,1] = -1.
    return la
end
@everywhere function lowerLeft(n)
    la = zeros(n,n)
    la[1,n] = -1.
    return la
end
d11 = @spawnat 2 tridiagonal(4)
d12 = @spawnat 3 lowerLeft(4)
d21 = @spawnat 4 upperRight(4)
d22 = @spawnat 5 tridiagonal(4)
d = DArray(reshape([d11 d12 d21 d22],(2,2)))   # create a distributed 8x8 matrix on a 2x2 process grid
```

$$\left(\begin{array}{cccc|cccc} 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{array}\right)$$

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.

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

Code for presenter in `juliaSet/juliaSetSerial.jl`

```
using ProgressMeter, NetCDF

function pixel(i, j, width, height, c, zoomOut)
    z = (2*(j-0.5)/width-1)+(2*(i-0.5)/height-1)im
       # rescale to -1:1 in the complex plane
    z *= zoomOut
    for i = 1:255
        z = z^2 + c
        if abs(z) >= 4
            return i
        end
    end
    return 255
end

n = Int(8e3)
height, width = n, n
c, zoomOut = 0.355 + 0.355im, 1.2
```

```
println("Computing Julia set ...")
data = zeros(Float32, height, width);
@showprogress for i in 1:height, j in 1:width
    data[i,j] = pixel(i, j, width, height, c, zoomOut)
end

println("Writing NetCDF ...")
filename = "test.nc"
isfile(filename) && rm(filename)
nccreate(filename, "xi", "x", collect(1:height), "y",
         collect(1:width), t=NC_FLOAT,
         mode=NC_NETCDF4, compress=9);
ncwrite(data, filename, "xi");
```

- We experimented with plotting with `Plots` and `ImageView`, but these were very slow ...
- Instead, saving to NetCDF and plotting in ParaView

Parallelizing the Julia set

We have a large array $\Rightarrow$ let's use DistributedArrays and compute it in parallel

```
< using ProgressMeter, NetCDF
---
> using NetCDF
> @everywhere using Distributed, DistributedArrays

< function pixel(i, j, width, height, c, zoomOut)
---
> @everywhere function pixel(i, j, width, height, c, zoomOut)
```

Parallelizing the Julia set

We have a large array ⇒ let's use DistributedArrays and compute it in parallel

```
< using ProgressMeter, NetCDF
---
> using NetCDF
> @everywhere using Distributed, DistributedArrays

< function pixel(i, j, width, height, c, zoomOut)
---
> @everywhere function pixel(i, j, width, height, c, zoomOut)


> @everywhere function fillLocalBlock(data, width, height, c, zoomOut)
>     h, w = localindices(data)
>     for i in collect(h)
>         iLocal = i - h.start + 1
>         for j in collect(w)
>             jLocal = j - w.start + 1
>             data.localpart[iLocal,jLocal] = pixel(i, j, width, height, c, zoomOut)
>         end
>     end
> end
```

Parallelizing the Julia set (cont.)

```
< data = zeros(Float32, height, width);
< @showprogress for i in 1:height, j in 1:width
<     data[i,j] = pixel(i, j, width, height, c, zoomOut)
---
> data = dzeros(Float32, height, width);   # distributed 2D array of 0's
> @time @sync for i in workers()
>     @spawnat i fillLocalBlock(data, width, height, c, zoomOut)

> nonDistributed = zeros(Float32, height, width);
> nonDistributed[:,:] = data[:,:];         # ncwrite does not accept DArray type
>

< ncwrite(data, filename, "xi");
---
> ncwrite(nonDistributed, filename, "xi");
```

Parallel Julia set code

```
using NetCDF
@everywhere using Distributed, DistributedArrays

@everywhere function pixel(i, j, width, height, c, zoomOut)
    z = (2*(j-0.5)/width-1)+(2*(i-0.5)/height-1)im
       # rescale to -1:1 in the complex plane
    z *= zoomOut
    for i = 1:255
        z = z^2 + c
        if abs(z) >= 4
            return i
        end
    end
    return 255
end
n = Int(8e3)
height, width, c, zoomOut = n, n, 0.355 + 0.355im, 1.2
@everywhere function fillLocalBlock(data,width, height,
                                     c, zoomOut)
    h, w = localindices(data)
    for i in collect(h)
        iLocal = i - h.start + 1
        for j in collect(w)
            jLocal = j - w.start + 1
            data.localpart[iLocal,jLocal] =
                pixel(i, j, width, height, c, zoomOut)
        end
    end
end
```

```
println("Computing Julia set ...")
data = dzeros(Float32, height, width);
# distributed 2D array of 0's
@time @sync for i in workers()
    @spawnat i fillLocalBlock(data, width, height,
                              c, zoomOut)
end

nonDistributed = zeros(Float32, height, width);
nonDistributed[:,:] = data[:,:];
# ncwrite does not accept DArray type

println("Writing NetCDF ...")
filename = "test.nc"
isfile(filename) && rm(filename)
nccreate(filename, "xi", "x", collect(1:height), "y",
         collect(1:width), t=NC_FLOAT,
         mode=NC_NETCDF4, compress=9);
ncwrite(nonDistributed, filename, "xi");
```

Parallel Julia set code

```
using NetCDF
@everywhere using Distributed, DistributedArrays

@everywhere function pixel(i, j, width, height, c, zoomOut)
    z = (2*(j-0.5)/width-1)+(2*(i-0.5)/height-1)im
       # rescale to -1:1 in the complex plane
    z *= zoomOut
    for i = 1:255
        z = z^2 + c
        if abs(z) >= 4
            return i
        end
    end
    return 255
end
n = Int(8e3)
height, width, c, zoomOut = n, n, 0.355 + 0.355im, 1.2
@everywhere function fillLocalBlock(data,width, height,
                                     c, zoomOut)
    h, w = localindices(data)
    for i in collect(h)
        iLocal = i - h.start + 1
        for j in collect(w)
            jLocal = j - w.start + 1
            data.localpart[iLocal,jLocal] =
                pixel(i, j, width, height, c, zoomOut)
        end
    end
end
```

```
println("Computing Julia set ...")
data = dzeros(Float32, height, width);
# distributed 2D array of 0's
@time @sync for i in workers()
    @spawnat i fillLocalBlock(data, width, height,
                              c, zoomOut)
end

nonDistributed = zeros(Float32, height, width);
nonDistributed[:,:] = data[:,:];
# ncwrite does not accept DArray type

println("Writing NetCDF ...")
filename = "test.nc"
isfile(filename) && rm(filename)
nccreate(filename, "xi", "x", collect(1:height), "y",
         collect(1:width), t=NC_FLOAT,
         mode=NC_NETCDF4, compress=9);
ncwrite(nonDistributed, filename, "xi");
```

```
$ julia juliaSetSerial.jl                    # serial runtime: 37s 37s
$ julia -p 1 juliaSetDistributedArrays.jl    # serial runtime: 28.2s 31.6s 32.3s
$ julia -p 2 juliaSetDistributedArrays.jl    # with 2 processes: 14.9s 14.9s 15.8s
```

SharedArrays

- Part of the Julia Standard Library (comes with the language)
- A `SharedArray` is shared across processes (set of workers) on the same node
  - full array is stored on the control process
  - significant cache on each worker
- Similar to DistributedArrays, you can read elements using their global indices from any worker
- Unlike with DistributedArrays, with SharedArrays you
  - can write into any part of the array on any worker $\Rightarrow$ potential for a race condition and indeterministic outcome with a poorly written code!
  - are limited to a set of workers on the same node

SharedArrays (cont.)

```
using Distributed, SharedArrays
addprocs(4)

a = SharedArray{Float64}(30);

a[:] .= 1.0           # assign from the control process
@fetchfrom 2 sum(a)   # correct (30.0)
@fetchfrom 3 sum(a)   # correct (30.0)

@sync @spawnat 2 a[:] .= 2.0   # can assign from any worker!
@fetchfrom 3 sum(a)            # correct (60.0)

b = SharedArray{Float64}((1000), init = x -> x .= 1.0)   # use a function to initialize `b`

@everywhere using SharedArrays   # otherwise `localindices` won't be found on worker 2
@fetchfrom 2 localindices(b)     # this block is assigned for processing on worker 2

# Let's fill each element with its corresponding myd() value:

b = SharedArray{Float64}((1000), init = x -> x .= myid())   # indeterminate outcome!

b = SharedArray{Float64}((1000), init = x -> x[localindices(x)] .= myid()) # parallel init
```

2D SharedArray

```
using Distributed, SharedArrays
addprocs(4)

a = SharedArray{Float64}(10000,10000);
@distributed for i in 1:10000   # parallel for loop split across all workers
    for j in 1:10000
        a[i,j] = myid()         # ID of the worker that initialized this element
    end
end
a       # available on all workers

a[1:10,1:10]                # on the control process
@fetchfrom 2 a[1:10,1:10]   # on worker 2
```

Brute-force $\mathcal{O}(N^2)$ accurate solver

- Problem: place N identical particles randomly in a unit cube, zero initial velocities
- Method:
  - force evaluation via direct summation
  - single variable (adaptive) time step (smaller Δt when any two particles are close)
  - time integration: more accurate than simple forward Euler + one force evaluation per time step
  - two parameters: softening length and Courant number
- In a real simulation, you would replace:
  - direct summation with a tree- or mesh-based $\mathcal{O}(N \log N)$ code
  - current integrator with a higher-order scheme, e.g. Runge-Kutta
  - current timestepping with hierarchical particle updates
  - for long-term stable evolution with a small number of particles, use a symplectic orbit integrator
- Expected solutions:
  - 2 particles: should pass through each other, infinite oscillations
  - 3 particles: likely form a close binary + distant 3$^{\text{rd}}$ particle (hierarchical triple system)
  - many particles: likely form a gravitationally bound system, with occasional ejection

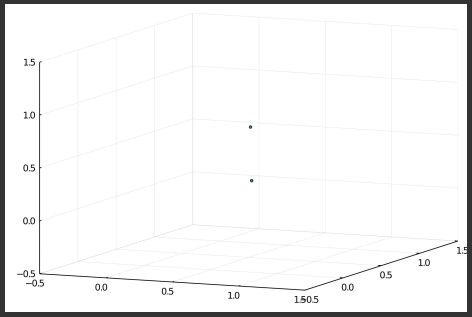
Serial N-body code

```
using Plots, ProgressMeter

npart = 20
niter = Int(1e5)
freq = 300
courant = 1e-3
softeningLength = 0.01

x = rand(npart, 3);   # uniform [0,1]
v = zeros(npart, 3);

nframes = floor(Int, niter/freq) + 1;
history = zeros(Float32, npart, 3, nframes);
history[:, :, 1] = x;
soft = softeningLength^2;

println("Computing ...");
force = zeros(Float32, npart, 3);
oldforce = zeros(Float32, npart, 3);
```

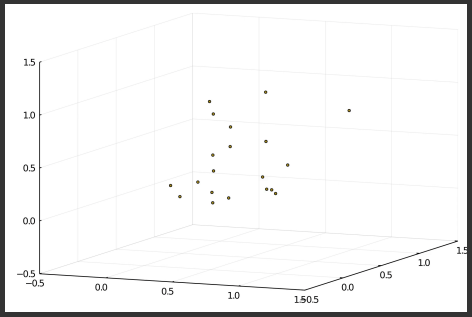
```
@showprogress for iter = 1:niter
    tmin = 1.e10
    for i = 1:npart
        force[i,:] .= 0.
        for j = 1:npart
            if i != j
                distSquared = sum((x[i,:] .- x[j,:]).^2) + soft;
                force[i,:] -= (x[i, :] .- x[j,:]) / distSquared^1.5;
                tmin = min(tmin, sqrt(distSquared /
                                      sum((v[i,:] .- v[j,:]).^2)));
            end
        end
    end
    dt = min(tmin*courant, 0.001);   # limit the initial step
    for i = 1:npart
        x[i,:] .+= v[i,:] .* dt .+ 0.5 .* oldforce[i,:] .* dt^2;
        v[i,:] .+= 0.5 .* (oldforce[i,:] .+ force[i,:]) .* dt;
        oldforce[i,:] .= force[i,:];
    end
    if iter%freq == 0
        history[:, :, trunc(Int, iter/freq)+1] = x;
    end
end

println("3D animation ...");
@showprogress for i = 1:nframes
    plt = plot(npart, xlim=(-0.5,1.5), ylim=(-0.5,1.5),
               zlim=(-0.5,1.5), seriestype=:scatter3d,
               legend=false, dpi=:300);
    scatter3d!(history[1:npart,1,i], history[1:npart,2,i],
               history[1:npart,3,i], markersize = 2);
    png("frame" * lpad(i, 4, '0'))
end
```

Solution

2 bodies



20 bodies



A frame is saved every 300 steps + variable timesteps
$\Rightarrow$ in these movies the time arrow represents the time step number (not time!)

Parallelizing the N-body code

Many small arrays $\Rightarrow$ let's use SharedArrays and fill them in parallel

```
> using Distributed, SharedArrays
> addprocs(2)
```

Parallelizing the N-body code

Many small arrays ⇒ let's use SharedArrays and fill them in parallel

```
> using Distributed, SharedArrays
> addprocs(2)

< v = zeros(npart, 3);
---
> x = SharedArray{Float32}(npart, 3);
> x[:,:] = rand(npart, 3);   # uniform [0,1]
> v = SharedArray{Float32}((npart, 3), init = x -> x .= 0.0);

< history = zeros(Float32, npart, 3, nframes);
---
> history = SharedArray{Float32}((npart, 3, nframes), init = x -> x .= 0.0);

< force = zeros(Float32, npart, 3);
< oldforce = zeros(Float32, npart, 3);
---
> force = SharedArray{Float32}(npart,3);
> oldforce = SharedArray{Float32}((npart,3), init = x -> x .= 0.0);
```

Parallelizing the N-body code

Many small arrays ⇒ let's use SharedArrays and fill them in parallel

```
> using Distributed, SharedArrays
> addprocs(2)

< v = zeros(npart, 3);
---
> x = SharedArray{Float32}(npart, 3);
> x[:,:] = rand(npart, 3);   # uniform [0,1]
> v = SharedArray{Float32}((npart, 3), init = x -> x .= 0.0);

< history = zeros(Float32, npart, 3, nframes);
---
> history = SharedArray{Float32}((npart, 3, nframes), init = x -> x .= 0.0);

< force = zeros(Float32, npart, 3);
< oldforce = zeros(Float32, npart, 3);
---
> force = SharedArray{Float32}(npart,3);
> oldforce = SharedArray{Float32}((npart,3), init = x -> x .= 0.0);

<     for i = 1:npart
---
>     tmin = @distributed (min) for i = 1:npart
```

Parallel N-body code

```
using Plots, ProgressMeter
using Distributed, SharedArrays
addprocs(4)

npart = 20
niter = Int(1e5)
freq = 300
courant = 1e-3
softeningLength = 0.01

x = SharedArray{Float32}(npart, 3);
x[:,:] = rand(npart, 3);   # uniform [0,1]
v = SharedArray{Float32}((npart, 3),
                init = x -> x .= 0.0);

nframes = floor(Int, niter/freq) + 1;
history = SharedArray{Float32}((npart, 3, nframes),
                init = x -> x .= 0.0);
history[:, :, 1] = x;
soft = softeningLength^2;

println("Computing ...");
force = SharedArray{Float32}(npart,3);
oldforce = SharedArray{Float32}((npart,3),
                        init = x -> x .= 0.0);
```

```
@showprogress for iter = 1:niter
    tmin = @distributed (min) for i = 1:npart
        tmin = 1.e10
        force[i,:] .= 0.
        for j = 1:npart
            if i != j
                distSquared = sum((x[i,:] .- x[j,:]).^2) + soft;
                force[i,:] -= (x[i, :] .- x[j,:]) / distSquared^1.5;
                tmin = min(tmin, sqrt(distSquared /
                                      sum((v[i,:] .- v[j,:]).^2)));
            end
        end
        tmin
    end
    dt = min(tmin*courant, 0.001);   # limit the initial step
    for i = 1:npart
        x[i,:] .+= v[i,:] .* dt .+ 0.5 .* oldforce[i,:] .* dt^2;
        v[i,:] .+= 0.5 .* (oldforce[i,:] .+ force[i,:]) .* dt;
        oldforce[i,:] .= force[i,:];
    end
    if iter%freq == 0
        history[:, :, trunc(Int, iter/freq)+1] = x;
    end
end

println("3D animation ...");
@showprogress for i = 1:nframes
    plt = plot(npart, xlim=(-0.5,1.5), ylim=(-0.5,1.5),
               zlim=(-0.5,1.5), seriestype=:scatter3d,
               legend=false, dpi=:300);
    scatter3d!(history[1:npart,1,i], history[1:npart,2,i],
               history[1:npart,3,i], markersize = 2);
    png("frame" * lpad(i, 4, '0'))
end
```

Parallel performance: 2-core laptop and Cedar

Laptop, 20 particles, 10^5 steps:

code	computing	animation
serial	3m47s	1m32s
2 parallel workers	3m50s	1m30s
4 parallel workers	4m17s	1m29s

Laptop, 100 particles, 10^3 steps:

code	computing
serial	59s
2 parallel workers	36s
4 parallel workers	37s

Laptop, 300 particles, 10^3 steps:

code	computing
serial	7m48s
2 parallel workers	4m52s
4 parallel workers	4m23s

Cedar, 100 particles, 10^3 steps:

code	computing
serial	1m23s
2 cores	46s
4 cores	29s
8 cores	22s
16 cores	18s
32 cores	19s

```
module load StdEnv/2020 julia/1.5.2

sbatch/salloc --nodes=1-1 --ntasks=...
julia -p $SLURM_NPROCS nbodyDistributedShared.jl

sbatch/salloc --ntasks=1 --cpus-per-task=...
julia -p $SLURM_CPUS_PER_TASK nbodyDistributedShared.jl
```

Summary

- We covered Julia's multi-threading and multi-processing
  - showed timings both on a 2-core laptop (with hyperthreading) and on up to 32 cores on Cedar
- DistributedArrays vs. SharedArrays
- Parallelized 3 computationally intensive problems: slow series, Julia set, N-body
- Useful resources:
  - "Julia at Scale" forum `https://discourse.julialang.org/c/domain/parallel`
  - Baolai Ge's (SHARCNET) webinar on parallel Julia `https://youtu.be/xTLFz-5a5Ec`
  - brief introduction to parallel computing in Julia (some additional concepts not covered in this webinar) `https://codingclubuc3m.github.io/2018-06-06-Parallel-computing-Julia.html`
  - performance tips `https://docs.julialang.org/en/v1/manual/performance-tips`

Questions?