Productive Parallel Programming with Dagger.jl

Julian Samaroo

Massachusetts Institute of Technology

Przemysław Szufel

SGH Warsaw School of Economics

Available parallelism types in Julia

(skipping @simd and green threads via @async)

Multithreading (-t)

JULIA_NUM_THREADS

Threads.@spawn

Threads.fetch

Threads.@threads

(t=Task(f)).sticky=false

Multiprocessing/distributed (-p, --machine-file)

addprocs/ClusterManagers

Distributed.@spawn

@fetch

@distributed

Future

GPU

CUDA.jl or ...

broadcasting/@cuda

threadIdx()

blockDim()

synchronize()

- **different APIs, different approaches**
- **no composability via representation of jobs/tasks**

The ~~two~~three-paradigm problem of parallel computing in Julia

Threads + Distributed + GPU

- Different approaches for parallelizing tasks
- Challenges for heterogenous computing environments
  - task allocation
  - task orchestration
  - data transfer

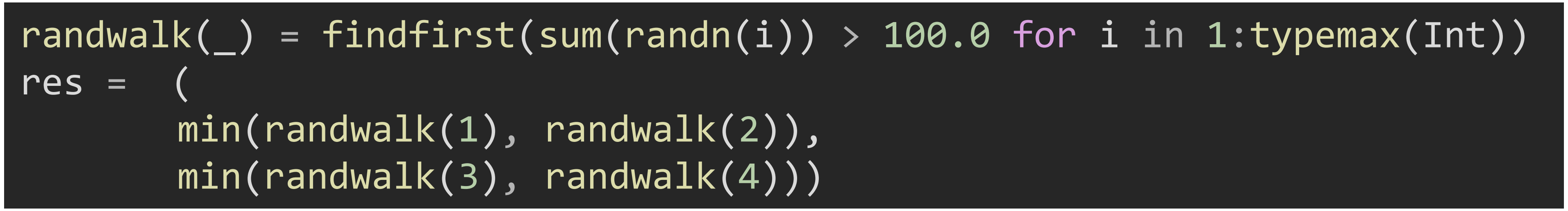
Development focused on designing Patterns for moving data between computing paradigms

DAGGER.jl

- Define tasks and their dependencies
- Let Dagger decide to which heterogenous worker the task should be assigned
- Agree to loose some control…

Development focused on writing your algorithms

The two paradigm problem illustrated (1)



```
randwalk(_) = findfirst(sum(randn(i)) > 100.0 for i in 1:typemax(Int))
res =  (
        min(randwalk(1), randwalk(2)),
        min(randwalk(3), randwalk(4)))
```

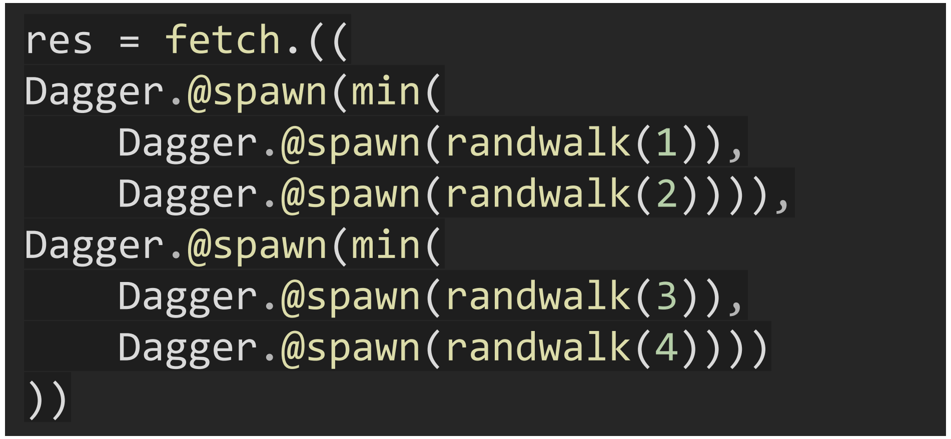
Executes on threads

```
v1 = Threads.@spawn randwalk(1)
v2 = Threads.@spawn randwalk(2)
v3 = Threads.@spawn randwalk(3)
v4 = Threads.@spawn randwalk(4)
m1 = Threads.@spawn min(fetch(v1),fetch(v2))
m2 = Threads.@spawn min(fetch(v3),fetch(v4))
res = fetch.((m1,m2))
```

Executes on workers

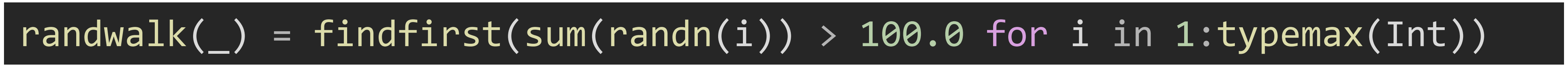
```
v1 = Distributed.@spawn randwalk(1)
v2 = Distributed.@spawn randwalk(2)
v3 = Distributed.@spawn randwalk(3)
v4 = Distributed.@spawn randwalk(4)
m1 = Distributed.@spawn min(fetch(v1),fetch(v2))
m2 = Distributed.@spawn min(fetch(v3),fetch(v4))
res = fetch.((m1,m2))
```

Executes on threads, or workers, or both



```
res = fetch.((
Dagger.@spawn(min(
    Dagger.@spawn(randwalk(1)),
    Dagger.@spawn(randwalk(2)))),
Dagger.@spawn(min(
    Dagger.@spawn(randwalk(3)),
    Dagger.@spawn(randwalk(4))))
))
```

The two paradigm problem illustrated (2)



```julia
randwalk(_) = findfirst(sum(randn(i)) > 100.0 for i in 1:typemax(Int))
```

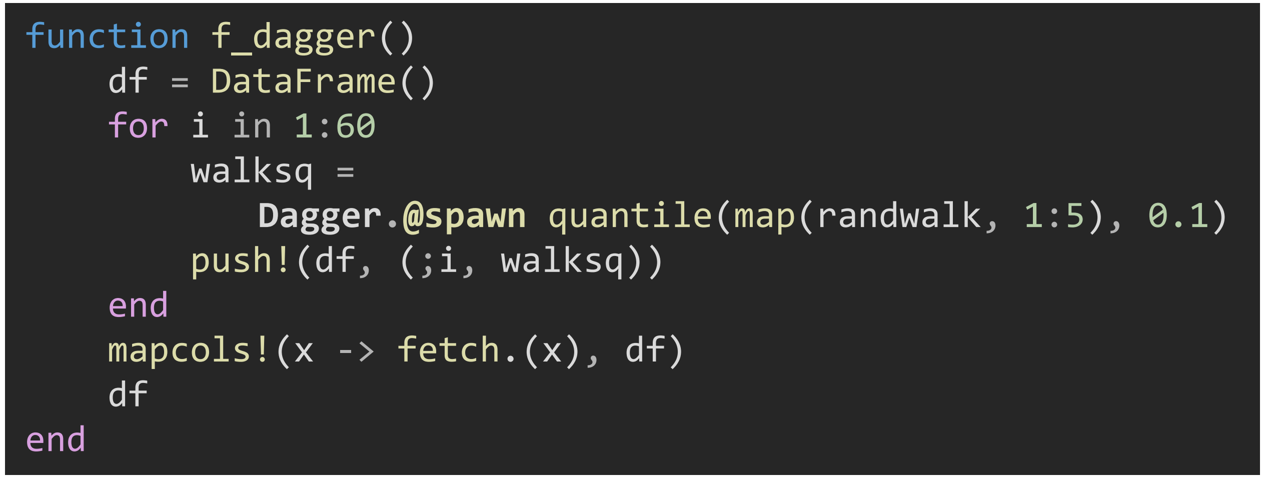
Executes on threads

```julia
function f_parallel()
    df = DataFrame()
    l = Threads.ReentrantLock()
    Threads.@threads for i in 1:30
        walksq = quantile(
                        map(randwalk, 1:100),0.1)
        Threads.lock(l) do
            push!(df, (;i,walksq))
        end
    end
    df
end
```

Executes on workers

```julia
function f_distibuted()
    @distributed (append!) for i in 1:30
        walksq = quantile(
                        map(randwalk, 1:30),0.1)
        DataFrame(;i,walksq)
    end
end
```

Executes on threads, or workers, or both



```julia
function f_dagger()
    df = DataFrame()
    for i in 1:60
        walksq =
            Dagger.@spawn quantile(map(randwalk, 1:5), 0.1)
        push!(df, (;i, walksq))
    end
    mapcols!(x -> fetch.(x), df)
    df
end
```