Writing parallel code in julia

Jiahao Chen
Julia Labs
MIT CSAIL
GitHub: @jiahao

CSAIL

julia labs


Alan Edelman


Jiahao Chen


Andreas Noack


Jarrett Revels


Jake Bolewski (alum; USAP)


Amit Murthy


Tanmay Mohapatra

Julia computing


Viral Shah

SMART

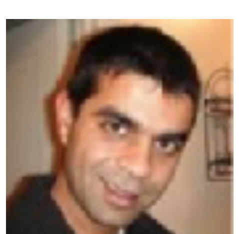
ISTC BIG DATA

XDATA

NIH Big Data to Knowledge (BD2K)

bigdata@CSAIL
MIT BIG DATA INITIATIVE

intel


Kiran Pamnany


Tim Mattson

https://github.com/jiahao/hpcc.jl

Multiple parallelism models in Julia

Partitioned global address space model (DistributedArrays.jl)

Master-worker multiprocess remote procedure calls (over TCP sockets)

DSL compiler for array-level vectorization (ParallelAccelerator.jl from Intel Labs)

Multithreading (experimental `@threads` in current v0.5 release; from Intel Labs and Julia Computing)

Instruction-level vectorization (`@simd`)

Multiple parallelism models in Julia

Partitioned global address space model (DistributedArrays.jl)

Master-worker multiprocess remote procedure calls (over TCP sockets)

DSL compiler for array-level vectorization (ParallelAccelerator.jl from Intel Labs)

Multithreading (experimental `@threads` in current v0.5 release; from Intel Labs and Julia Computing)

Instruction-level vectorization (`@simd`)

Master-worker parallelism

Remote procedure calls between one master and many workers form the most basic parallel constructs in Julia.

```
julia> addprocs(1) #Add one worker
1-element Array{Int64,1}:
 2

julia> remotecall(()->eval(:(f()=1)), 3) #send function definition
Future(3,1,45,Nullable{Any}())

julia> wait(ans) #stall until Future is available
Future(3,1,45,Nullable{Any}())

julia> w = remotecall(f, 3) #Create remote pointer to answer
Future(3,1,47,Nullable{Any}())

julia> fetch(w) #dereference Future when available
1
```

Master-worker parallelism

Using Julia's metaprogramming features, we can write a nicer API for parallel computing.

```
addprocs(3)

@everywhere f()=1 #broadcast function definition

w = @spawnat 3 f() #Create remote pointer to answer

fetch(w) #dereference Future when available
```

Master-worker parallelism

Using Julia's metaprogramming features, we can write a nicer API for parallel computing.

```
addprocs(3)

@everywhere f()=1 #broadcast function definition

w = @spawnat 3 f() #Create remote pointer to answer

fetch(w) #dereference Future when available
```

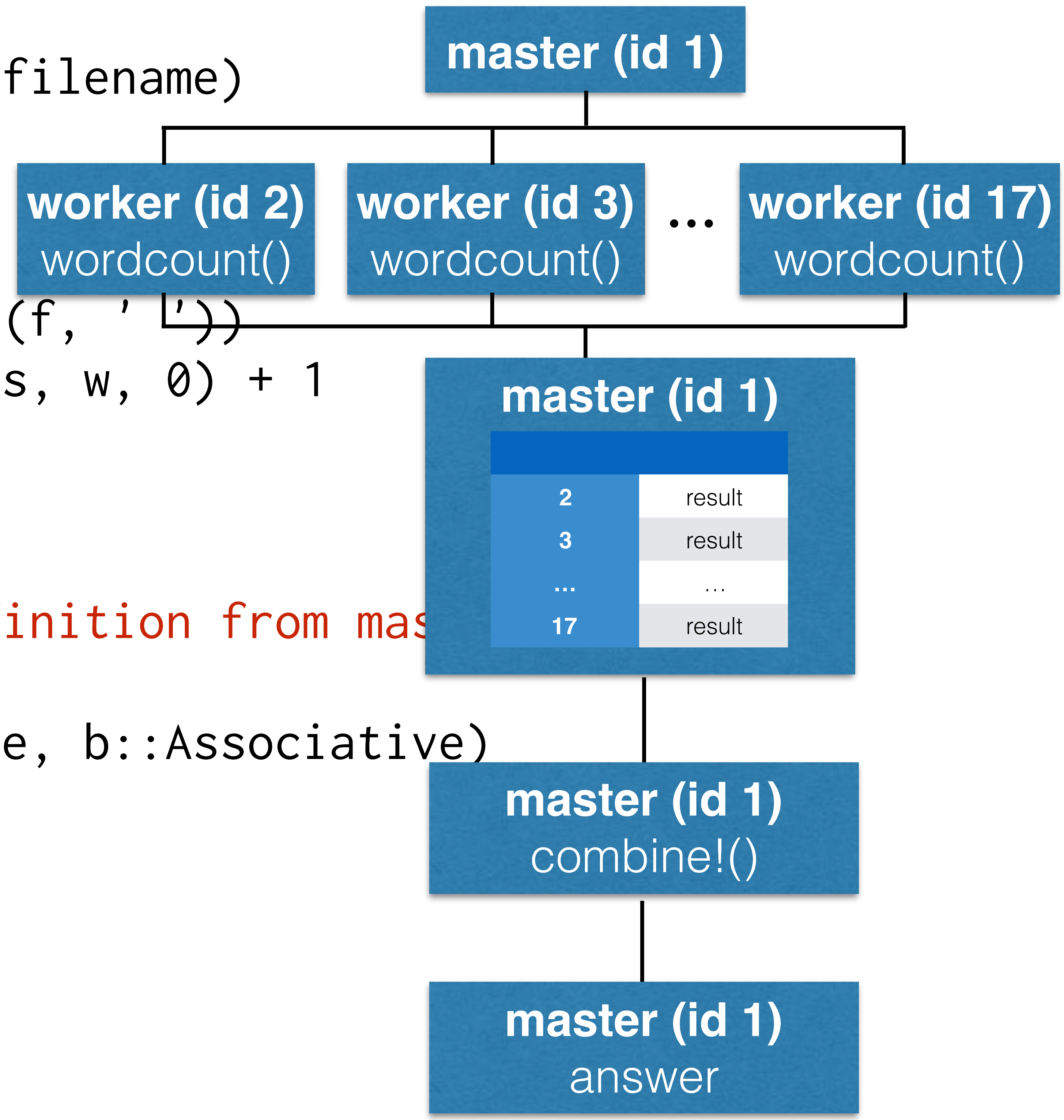
All higher level abstractions in Julia like parallel for loops (`@parallel (_) for`), parallel map (`pmap`), and distributed arrays are built on top of this model.

MapReduce in Julia

```
addprocs(16) #Attach 16 workers

@everywhere function wordcount(filename)
    words = Dict{String,Int}()
    open(filename) do f
        while !eof(f)
            w = chomp(readuntil(f, ' '))
            words[w] = get(words, w, 0) + 1
        end
    end
    return words
end #@everywhere broadcasts definition from master to workers

function combine!(a::Associative, b::Associative)
    for (k, v) in b
        a[k] = get(a, k, 0) + v
    end
    return a
end

answer = reduce(combine!, pmap(wordcount, ["a.txt", "b.txt"]))
```

MapReduce in Julia

```
addprocs(16) #Attach 16 workers

@everywhere function wordcount(filename)
    words = Dict{String,Int}()
    open(filename) do f
        while !eof(f)
            w = chomp(readuntil(f, ' '))
            words[w] = get(words, w, 0) + 1
        end
    end
    return words
end #@everywhere broadcasts definition from mas

function combine!(a::Associative, b::Associative)
    for (k, v) in b
        a[k] = get(a, k, 0) + v
    end
    return a
end

answer = reduce(combine!, pmap(wordcount, ["a.txt", "b.txt"]))
```

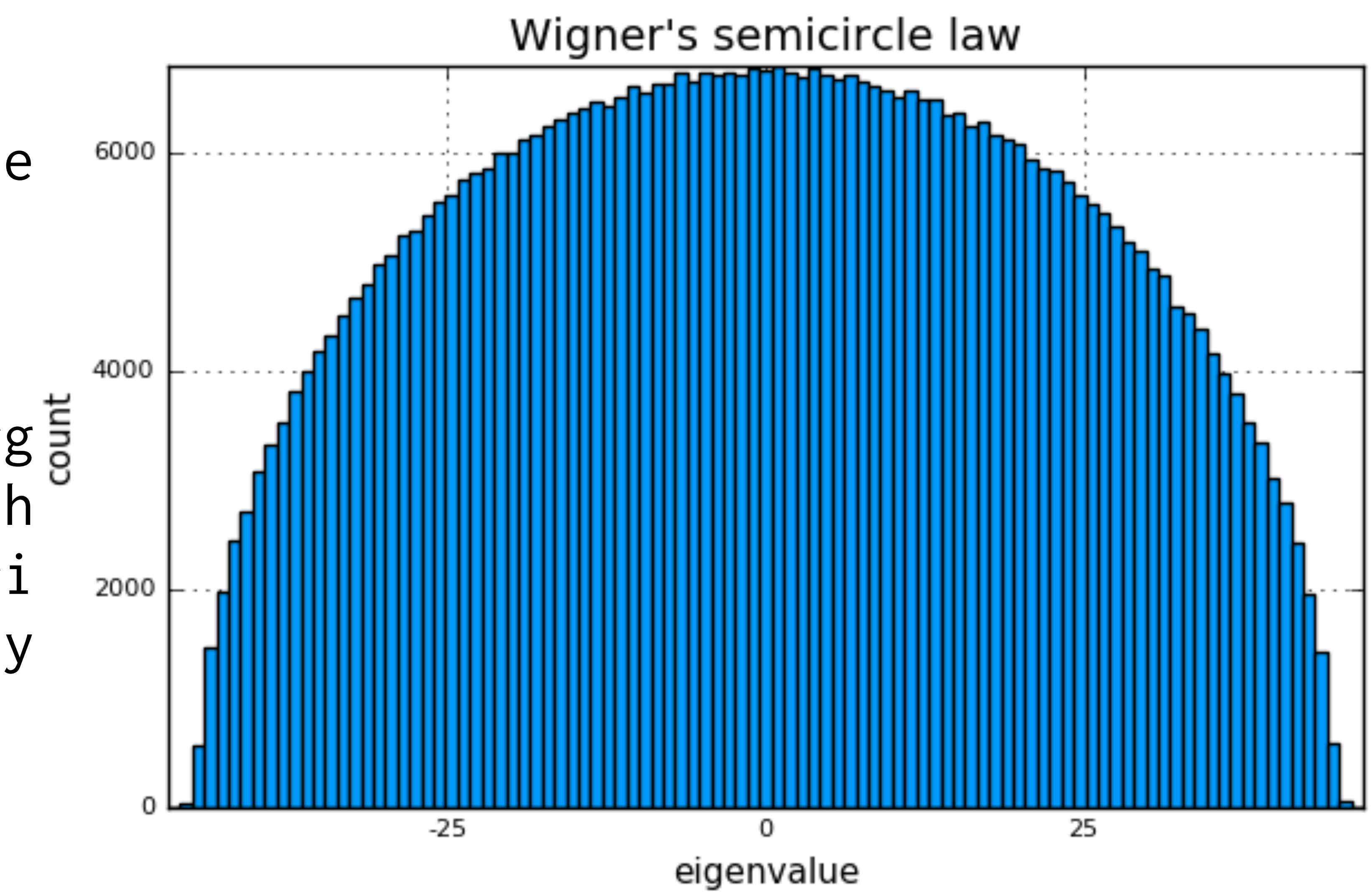


Parallel reduction in Julia

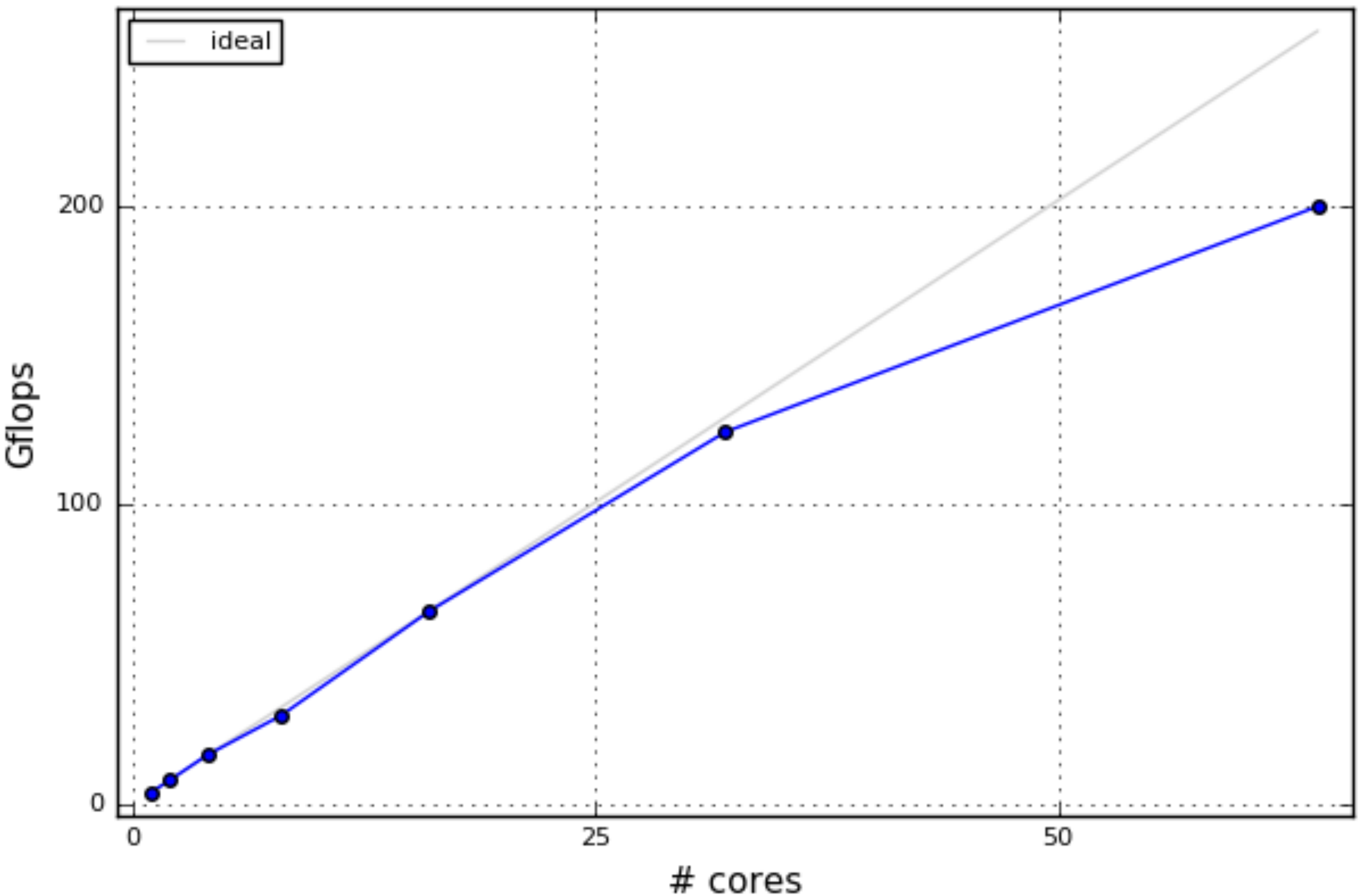
```
function randeig(t, n; xlims=(-2.1*√n, 2.1*√n), bins=100)
    xgrid = linspace(xlims..., bins)
    counts = @parallel (+) for _ = 1:t
        fit(Histogram, eigvals(Symmetric(randn(n, n))), xgrid).weights
    end
    Histogram(xgrid, counts)
end

@everywhere using StatsBase
h = randeig(1000, 500)

using Plots
Plots.bar(h::Histogram, args...) = bar(h.edges, h.weights)
bar(h, label = "", xlims=(h.edges[1].start, h.edges[1].stop),
   title = "Wigner's semicircle law”,
   xlabel = "eigenvalue", ylabel = "count")
png("semicircle")
```

Parallel reduction in Julia

```
function randeig(t, n; xlims=(-2.1*√n, 2.1*√n), bins=100)
    xgrid = linspace(xlims..., bins)
    counts = @parallel (+) for _ = 1:t
        fit(Histogram, eigvals(Symmetric(randn(n, n))), xgrid).weights
    end
    Histogram(xgrid, counts)
end

@everywhere using StatsBase
h = randeig(1000, 500)

using Plots
Plots.bar(h::Histogram, arg
bar(h, label = "", xlims=(h
   title = "Wigner's semici
   xlabel = "eigenvalue", y
png("semicircle")
```



Performance



Parallel prefix in Julia

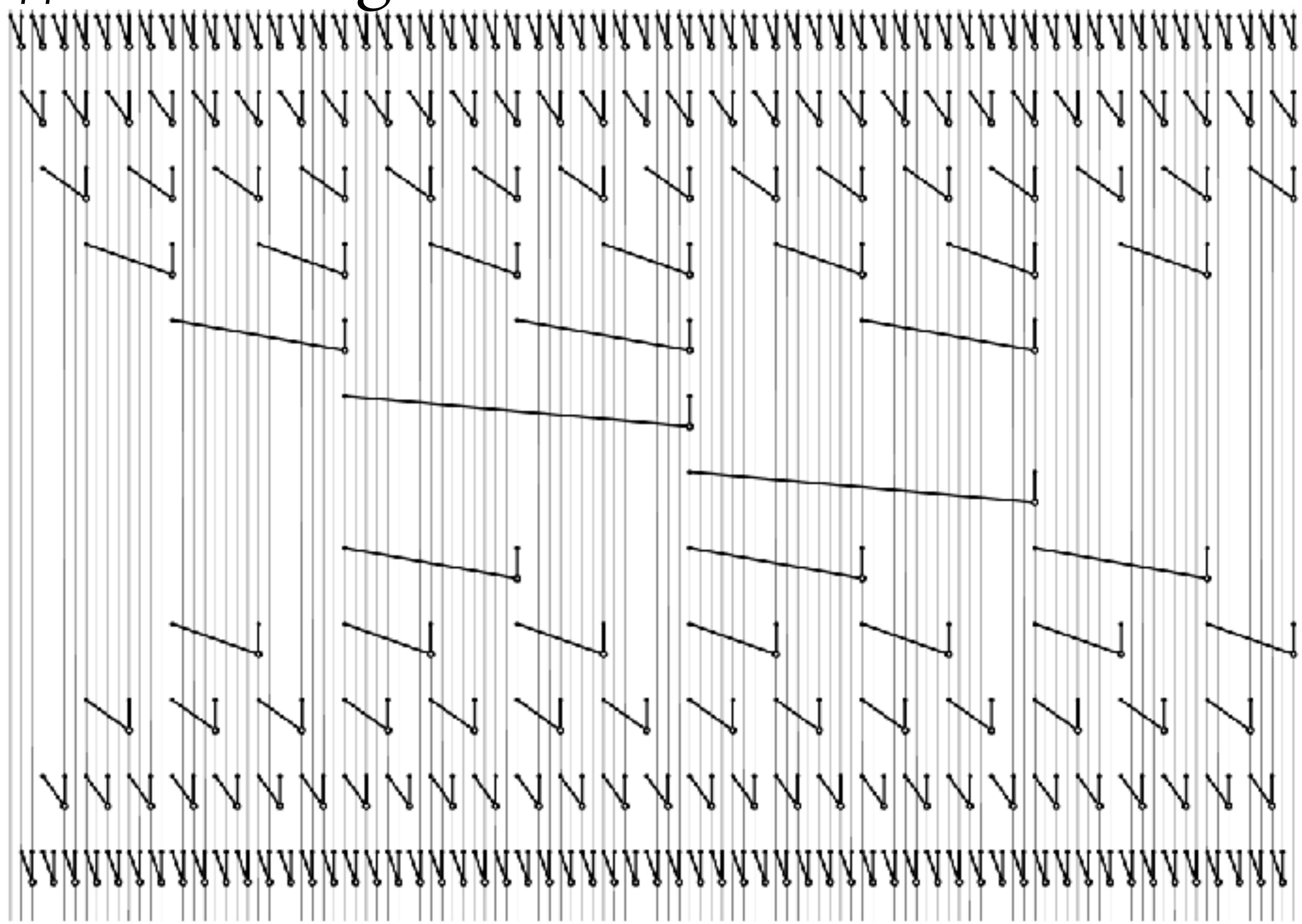
```
function prefix!(y, +)
    l=length(y)
    k=ceil(Int, log2(l))
    #The "reduce" tree
    for j=1:k, i=2^j:2^j:min(l, 2^k)
        y[i] = y[i-2^(j-1)] + y[i]
    end
    #The "broadcast" tree
    for j=(k-1):-1:1, i=3*2^(j-1):2^j:min(l, 2^k)
        y[i] = y[i-2^(j-1)] + y[i]
    end
    return y
end

#Define elementary operations on remote data
Base.(*)(r1::RemoteRef,r2::RemoteRef)
    @spawnat r2.where fetch(r1)*fetch(r2)
data = [@spawnat i rand(4096, 4096) for i=1:80]
prefix!(data, *)
```

Operator-level parallelism allows code reuse across serial and distributed data (doi:10.1109/HPTCDL.2014.9)

Parallel prefix in Julia

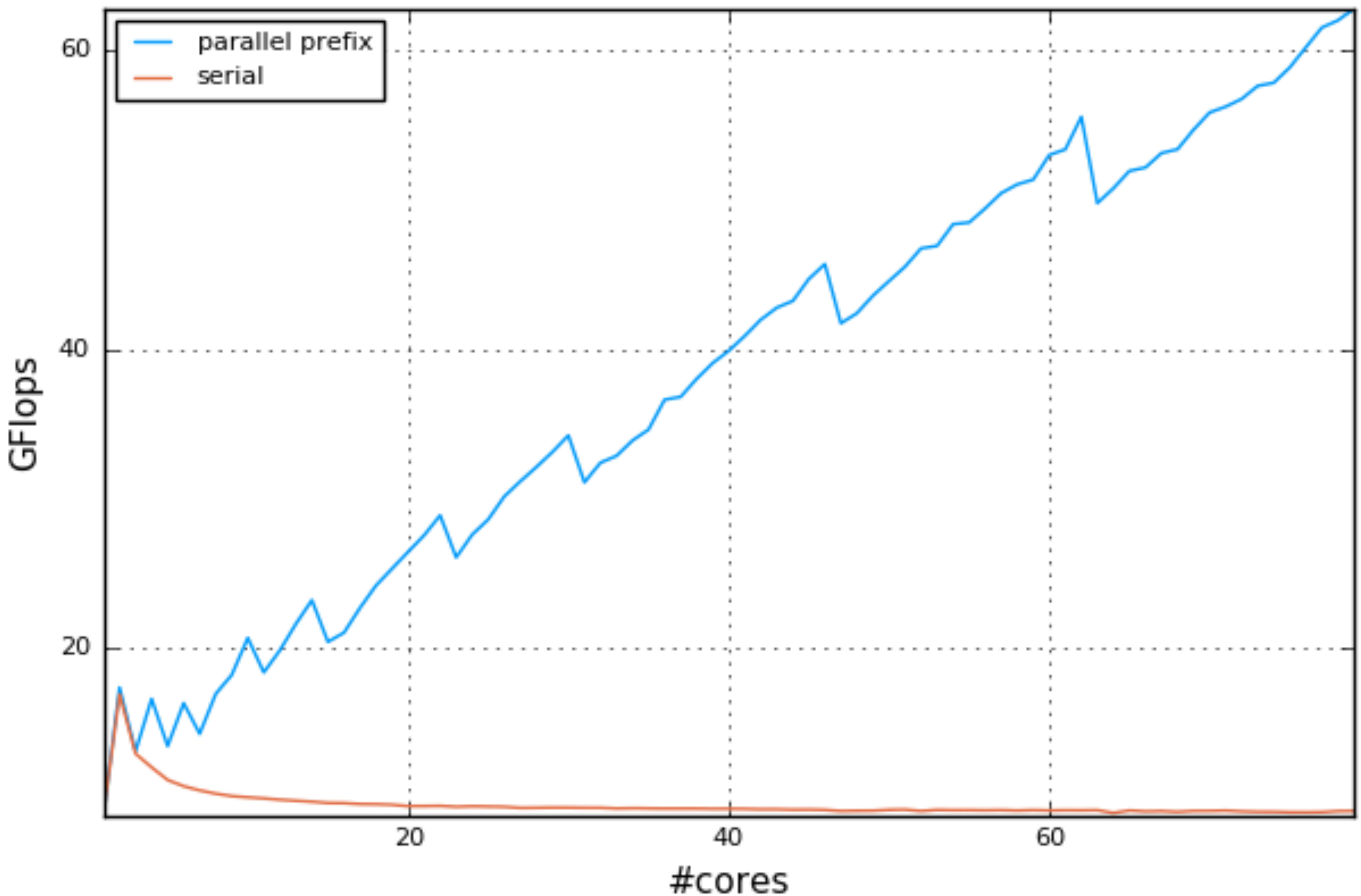
```
function prefix!(y, +)
    l=length(y)
    k=ceil(Int, log2(l))
    #The "reduce" tree
    for j=1:k, i=2^j:2
        y[i] = y[i-2^(
    end
    #The "broadcast" t
    for j=(k-1):-1:1,
        y[i] = y[i-2^
    end
    return y
end

#Define elementary ope
Base.(*)(r1::RemoteRef
    @spawnat r2.where
data = [@spawnat i rar
prefix!(data, *)
```

code can generate its own visualization!



Operator-level parallelism allows code reuse across serial and distributed data (doi:10.1109/HPTCDL.2014.9)

Performance



DistributedArrays

```
julia> addprocs(4);

julia> @everywhere using DistributedArrays

julia> A = @DArray [rand() for i=1:4000, j=1:4000]
4000×4000 DistributedArrays.DArray{Float64,2,Array{Float64,2}}:
 0.0981102  0.761717   0.932331   0.477988   0.648195    …
```

DistributedArrays

```
julia> dump(A)
```

dump() shows you the internal representation of the variable

DistributedArrays

```
julia> dump(A)
DistributedArrays.DArray{Float64,2,Array{Float64,2}}
  identity: …
  dims: Tuple{Int64,Int64}
    1: Int64 4000
    2: Int64 4000
  pids: Array{Int64}((2,2)) [2 4; 3 5]
  indexes:
Array{Tuple{UnitRange{Int64},UnitRange{Int64}}}
((2,2))
    1: Tuple{UnitRange{Int64},UnitRange{Int64}}
      1: UnitRange{Int64}
        start: Int64 1
        stop: Int64 2000
      2: UnitRange{Int64}
        start: Int64 1
        stop: Int64 2000
    2: …
  cuts: Array{Array{Int64,1}}((2,))
    1: Array{Int64}((3,)) [1,2001,4001]
    2: Array{Int64}((3,)) [1,2001,4001]
  release: Bool true
```

dump() shows you the internal representation of the variable

DistributedArrays

```
julia> dump(A)
DistributedArrays.DArray{Float64,2,Array{Float64,2}}
  identity: …
  dims: Tuple{Int64,Int64}
    1: Int64 4000
    2: Int64 4000
  pids: Array{Int64}((2,2)) [2 4; 3 5]
  indexes:
Array{Tuple{UnitRange{Int64},UnitRange{Int64}}}
((2,2))
    1: Tuple{UnitRange{Int64},UnitRange{Int64}}
      1: UnitRange{Int64}
        start: Int64 1
        stop: Int64 2000
      2: UnitRange{Int64}
        start: Int64 1
        stop: Int64 2000
    2: …
  cuts: Array{Array{Int64,1}}((2,))
    1: Array{Int64}((3,)) [1,2001,4001]
    2: Array{Int64}((3,)) [1,2001,4001]
  release: Bool true
```

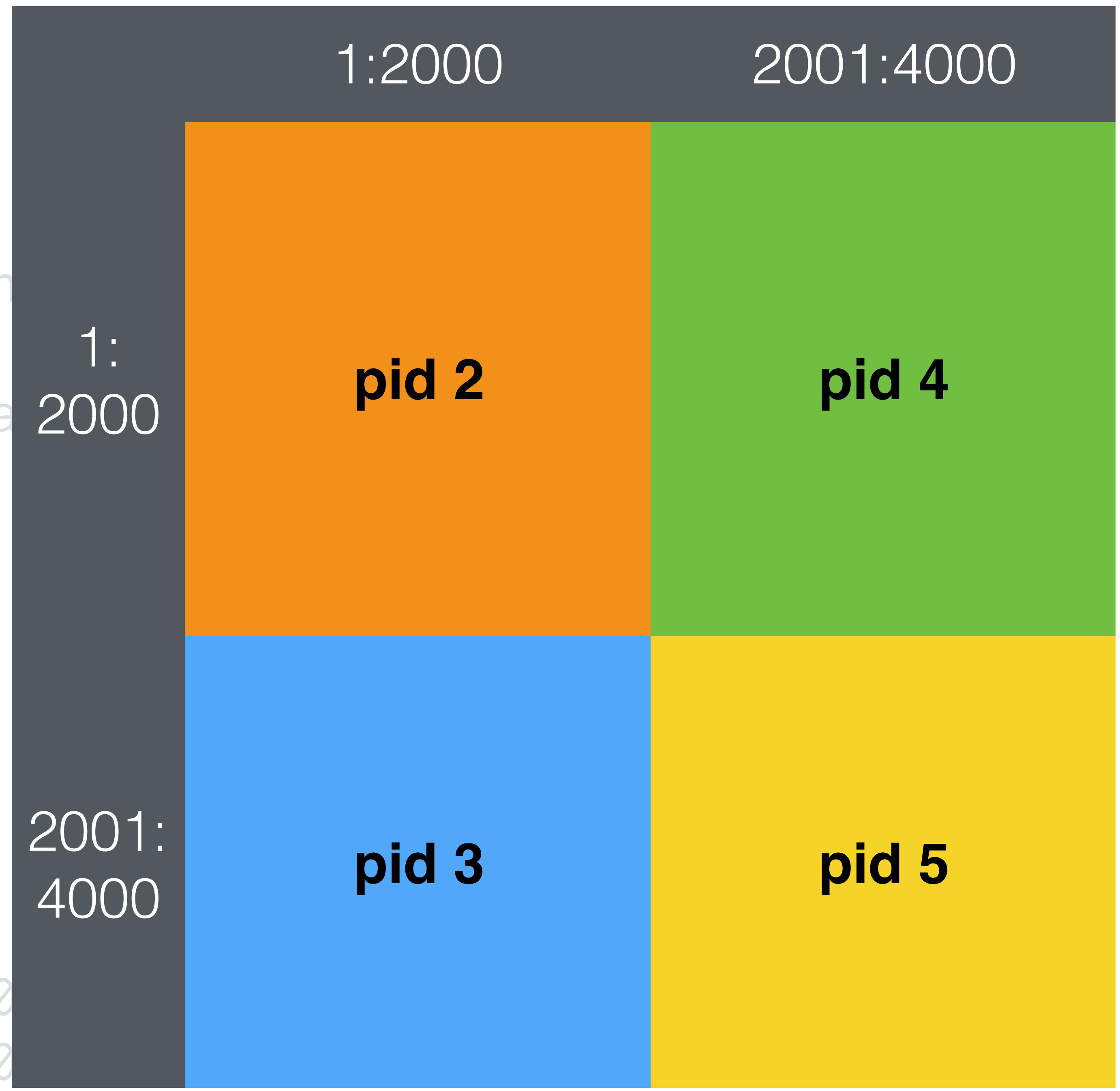
dump() shows you the internal representation of the variable

```
indexes = [
 (   1:2000,    1:2000),
 (2001:4000,    1:2000),
 (   1:2000, 2001:4000),
 (2001:4000, 2001:4000)
]
```

DistributedArrays

```
julia> dump(A)
DistributedArrays.DArray{Float64,2,Array{Float64,2}}
  identity: …
  dims: Tuple{Int64,Int64}
    1: Int64 4000
    2: Int64 4000
  pids: Array{Int64}((2,2)) [2 4; 3
  indexes:
Array{Tuple{UnitRange{Int64},UnitRan
((2,2))
    1: Tuple{UnitRange{Int64},UnitRa
      1: UnitRange{Int64}
        start: Int64 1
        stop: Int64 2000
      2: UnitRange{Int64}
        start: Int64 1
        stop: Int64 2000
    2: …
  cuts: Array{Array{Int64,1}}((2,))
    1: Array{Int64}((3,)) [1,2001,40
    2: Array{Int64}((3,)) [1,2001,40
  release: Bool true
```

```
indexes = [
  (   1:2000
  (2001:4000
  (   1:2000
  (2001:4000
]
```



distribution can be customized

DistributedArrays

	1:100	100:1899	1900: 2000	2001:4000
1: 2000	**pid 2**	**pid 8**	**pid 6**	**pid 4**
2001: 4000	**pid 3**	**pid 7**	**pid 9**	**pid 5**

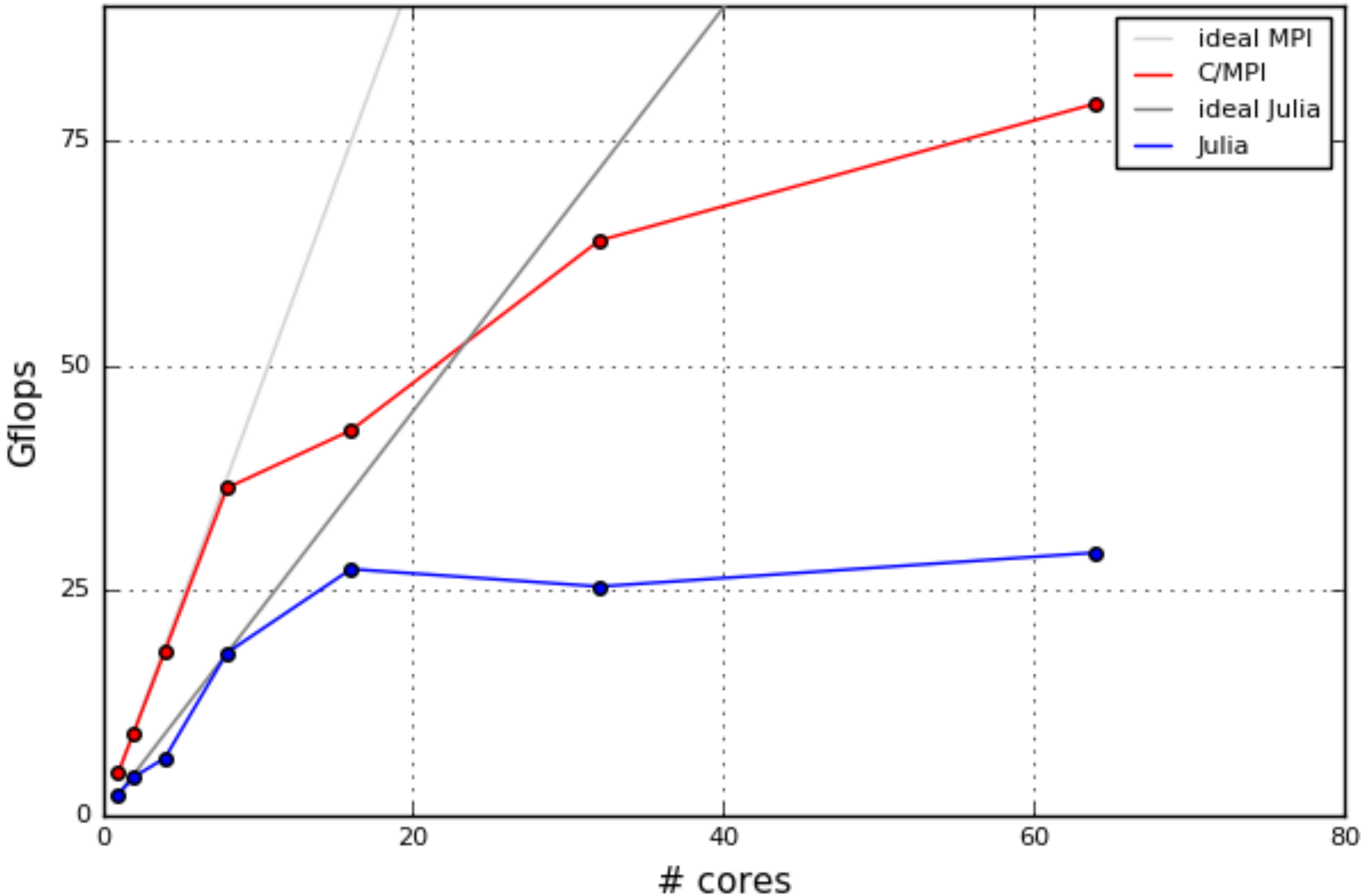
distribution can be customized

Streaming triad

Lines of code: 12

```
function streamtriad!(a, b, α, c)
    m = size(a, 1)
    for i in eachindex(a)
        a[i] = b[i] + α*c[i]
    end
end

function streamtriad!{T}(a::DArray{T,1}, b::DArray{T,1}, α::T, c::DArray{T,1})
    m = size(a, 1)
    @sync for p in a.pids
        @async remotecall_fetch(
            (a′, b′, α′, c′)->
            (streamtriad!(localpart(a′), localpart(b′), α′, localpart(c′))),
          p, a, b, α, c)
     end
end
```

Performance (N=1e8)

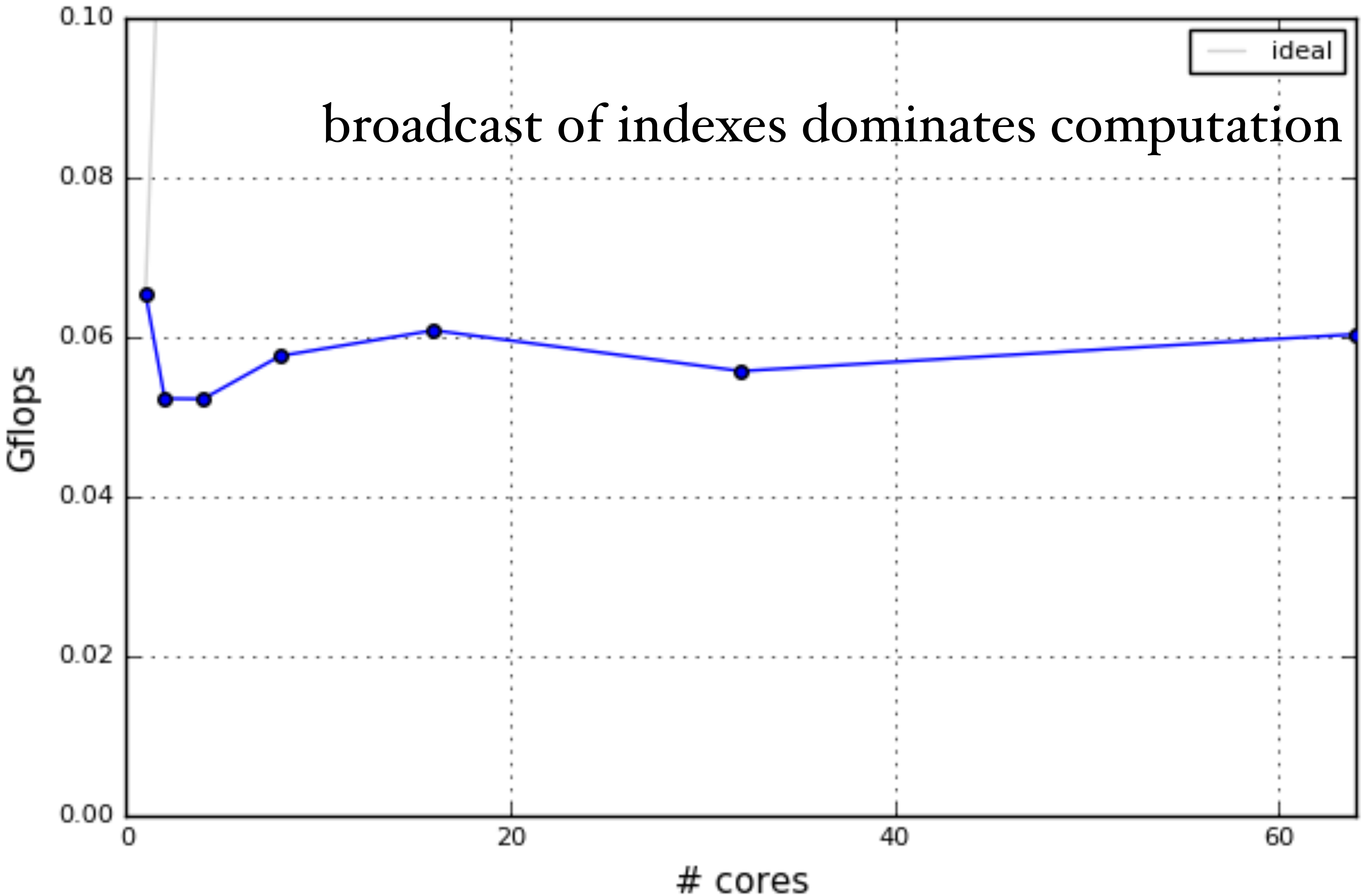


Random update

Lines of code: 23

```
#Look up which processor owns index idx in DArray
function procof{T}(A::DArray{T,1}, idx::Integer)
    i = searchsortedlast(A.cuts[1], idx)
    return A.pids[i], idx-A.indexes[i][1]+1
end

function updatearray!(A, w)
    Al = localpart(A)
    for (i, r) in w
        Al[i] $= r
    end
end

function randomupdate!{T<:Integer}(A::DArray{T,1}, nupdate)
    m = size(A, 1)
    work = Dict()
    for i=1:nupdate
        r = rand(T)
        index = r & (m-1) + 1
        p, i = procof(A, index)
        work[p] = push!(get(work, p, Tuple{UInt64,UInt64}[]), (i, r))
    end
    @sync for (p, w) in work
        remotecall(updatearray!, p, A, w)
    end
end
```

Performance



A\b

1. Factorize distributed array A using Communication-Avoiding LU (CALU; LAWNS 226)
   1. Divide A into block rows and run Tall-and-Skinny LU (TSLU) on each panel
      1. Find set of good pivot rows using tournament pivoting
      2. Permute pivot rows into first b rows of the panel
      3. Perform unpivoted LU on each panel
2. Gather distributed array A to local memory and solve by backsubstitution

Lines of code: 216

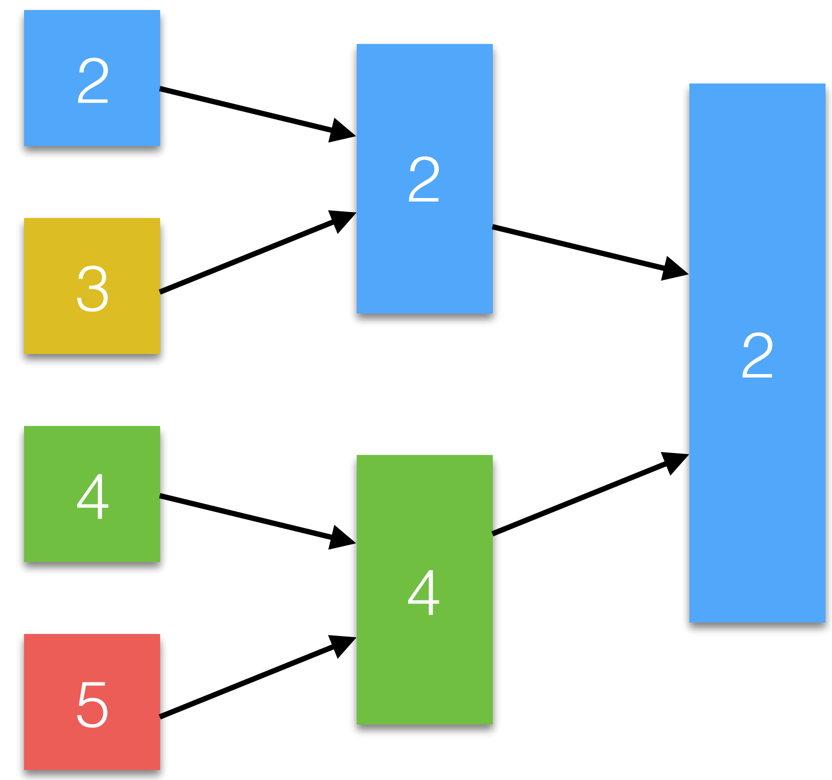
Tall and skinny LU

```
function tslu!(A, piv, k, b)
     #Step 1: Find set of good pivot rows
     lus = [@spawnat whoowns(br) lufact!(Array(br))
                for br in blockrows(A, k)]
     while length(lus) > 1
         npairs, isodd = divrem(length(lus), 2)
         newlus =[@spawnat lus[2i-1].where begin
             lufact!([fetch(lus[2i-1])[:U]; fetch(lus[2i])[:U]])
           end for i=1:npairs]
         isodd==1 && push!(newlus, lus[end])
         lus = newlus
    end

    #Step 2: Permute pivot rows into first b rows of the panel
    perm = LinAlg.ipiv2perm(fetch(lus[1])[:p], size(A, 2))[1:b2]
    permuterows!(A, perm)

    #Step 3: Unpivoted LU on panel
    return lufact!(view(A, :, 1:b), Val{false})
end
```

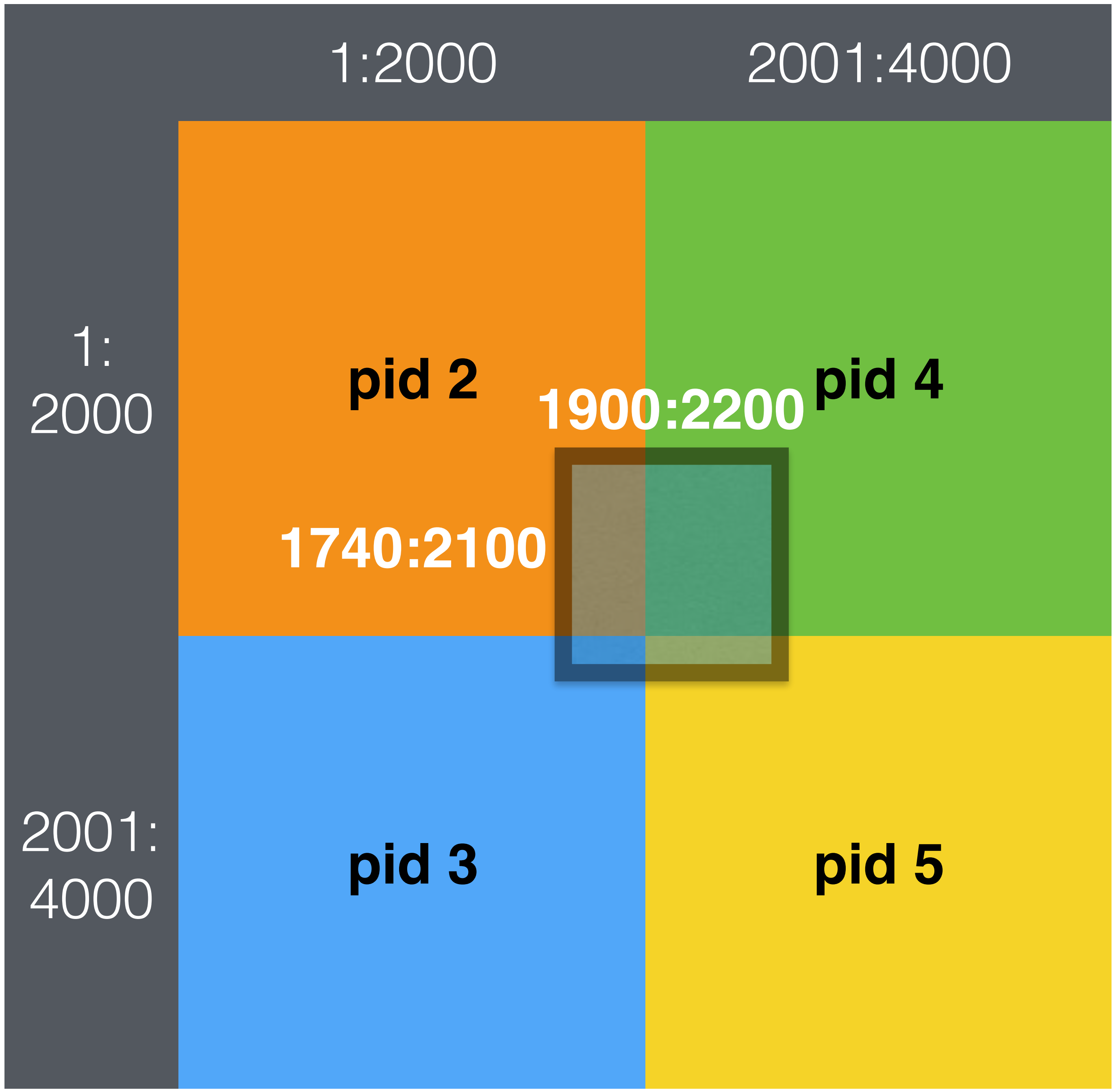
Tall and skinny LU

```
#Step 1: Find set of good pivot rows
lus = [@spawnat whoowns(br) lufact!(Array(br))
           for br in blockrows(A, k)]
while length(lus) > 1
    npairs, isodd = divrem(length(lus), 2)
    newlus =[@spawnat lus[2i-1].where begin
        lufact!([fetch(lus[2i-1])[:U];
                 fetch(lus[2i])[:U]])
      end for i=1:npairs]
    isodd==1 && push!(newlus, lus[end])
    lus = newlus
end
```

Runtime graph of dependencies



Views into distributed arrays



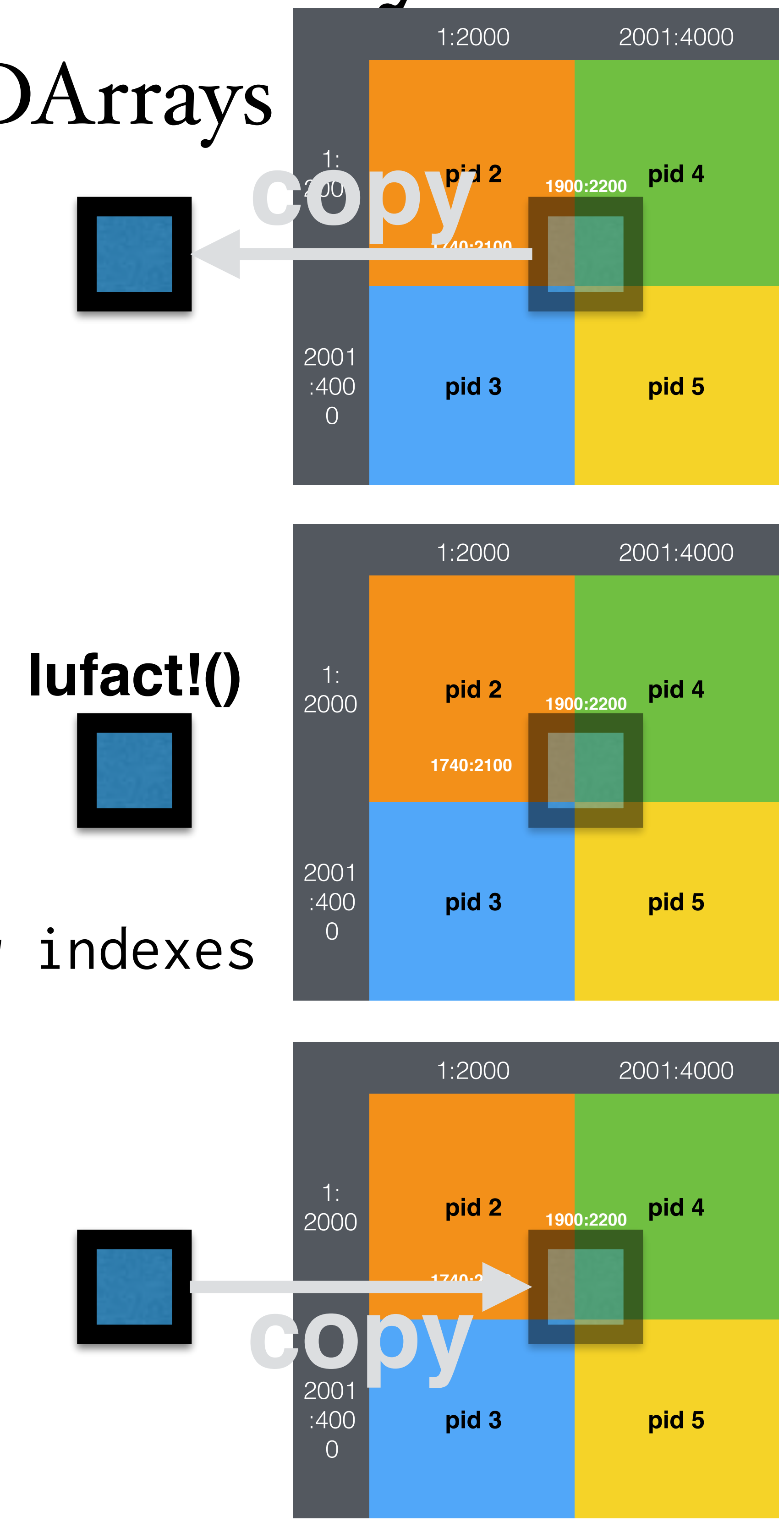

```
julia> B = view(A, 1740:2100, 1900:2200)
361×301
SubArray{Float64,2,DistributedArrays.DArray{Float64,2,Array{Float64,2}},Tuple{UnitRange{Int64},UnitRange{Int64}},false}:
 0.774241   0.60596    0.629025
0.829958  0.0268451  …
```

Julia's type system can describe views into distributed arrays

Dispatch on SubDArrays

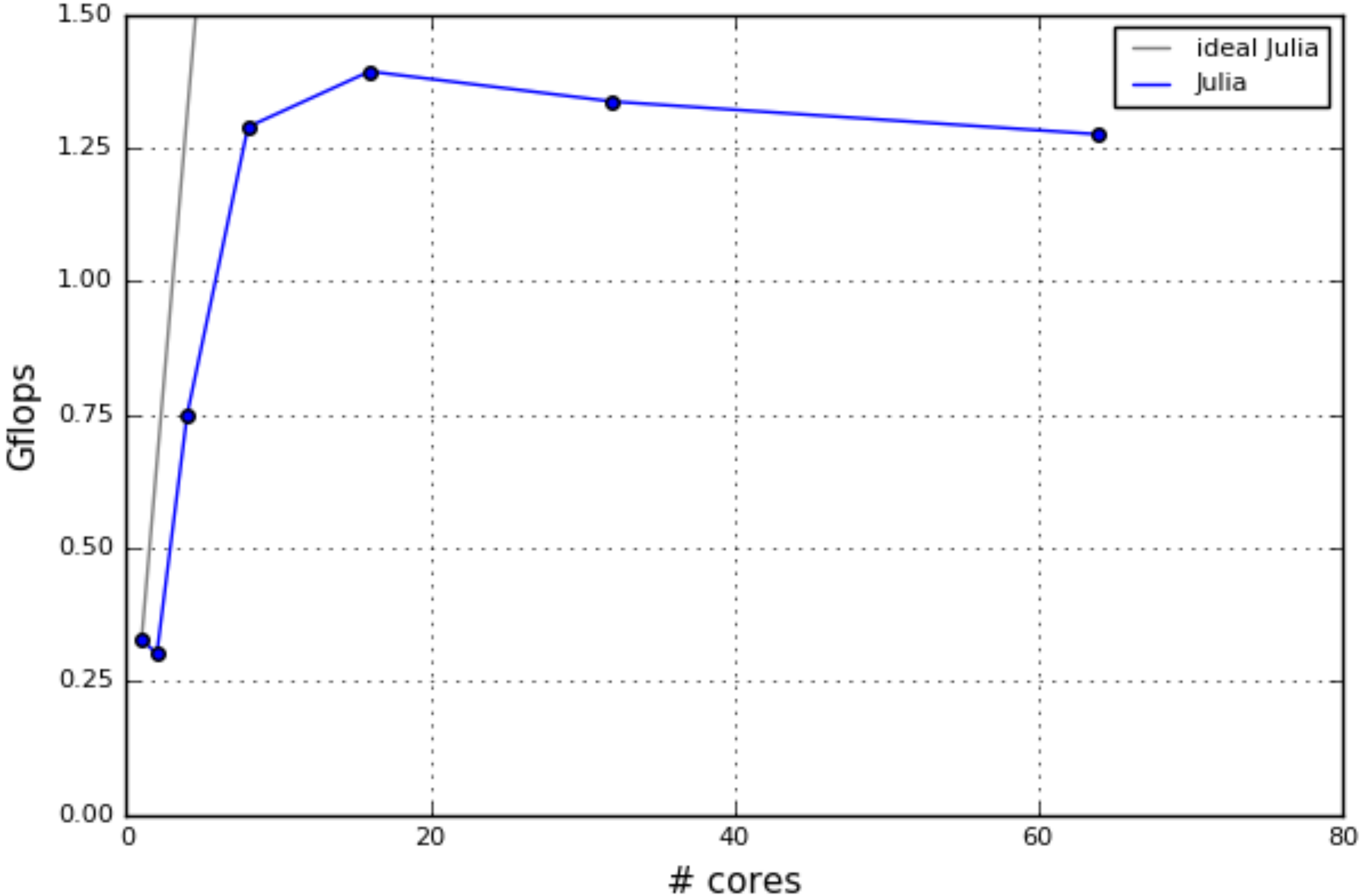
Overload lufact!() with new method for SubDArrays

```
function Base.lufact!{T,piv}(
        A::SubArray{T,2,DArray{T,2,Matrix{T}}},
        ::Type{Val{piv}}
    )

    tmpA = Array(A) #Copy to local memory
    lufact!(tmpA, Val{piv}) #Compute locally

    #Redistribute results
    pids = pidmap(A) #Compute global, local and view indexes
    @sync for (p, (gr, gc, lr, lc, sr, sc)) in pids
        @spawnat p localpart(A.parent)[lr, lc] =
            view(tmpA, sr, sc)
    end
end
```



gather-compute-scatter is a common pattern for SubDArray computations

Performance (N=4096)



Work in progress

Configurable transport layer in DistributedArrays.jl: use default TCP or MPI

Eliminating type stability of Futures

Stabilizing multithreading support

More SIMD (currently 256-bit AVX and 128-bit SSE2 only)

Code generation to Xeon Phi and NVIDIA NVPTX backends

Large-scale application to astronomical imaging (Celeste.jl, running on 8,192 Xeon cores, arXiv:1611.03404)