Scaling up numerical computing in Julia

Przemysław Szufel
https://szufel.pl/

Efficient code in Julia and matrices

- Avoid abstract types in data structures (including vectors, matrices…)
- Process data from arrays according to their order of representation in memory (columns)
- Preallocate result matrices
- Use the @. macro for vectorizing multiple operations - this will result in chaining vectorized operations
- Use views for array slicing
- Know about macros: @inbounds, @fastmath, @simd

https://docs.julialang.org/en/v1/manual/performance-tips/

Use tools for performance monitoring and optimization

- Debugger
- Profiler
- @time
- BenchmarkTools.jl and @btime
- @code_warntype

Scaling computations using parallel computing

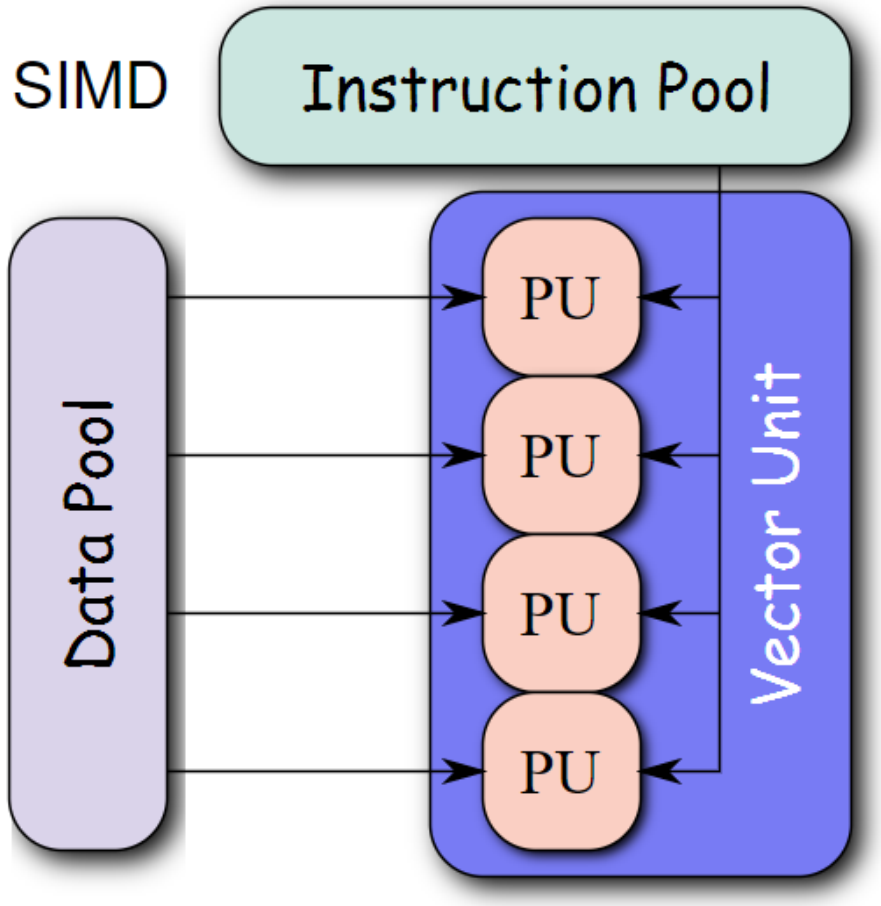
Przemysław Szufel
https://szufel.pl/

Parallelization options in programming languages

- Single instruction, multiple data (SIMD)
- Green-threads (co-routines) only where I/O is bottleneck
- Multi-threading
  - Language
  - Libraries
- Multi-processing – many julia processes at the same time
  - single machine
  - distributed (cluster)
  - distributed (cluster) via external tools
- GPU computing

SIMD

- Single instruction, multiple data (SIMD) describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.



Source: https://en.wikipedia.org/wiki/SIMD

Data level parallelism

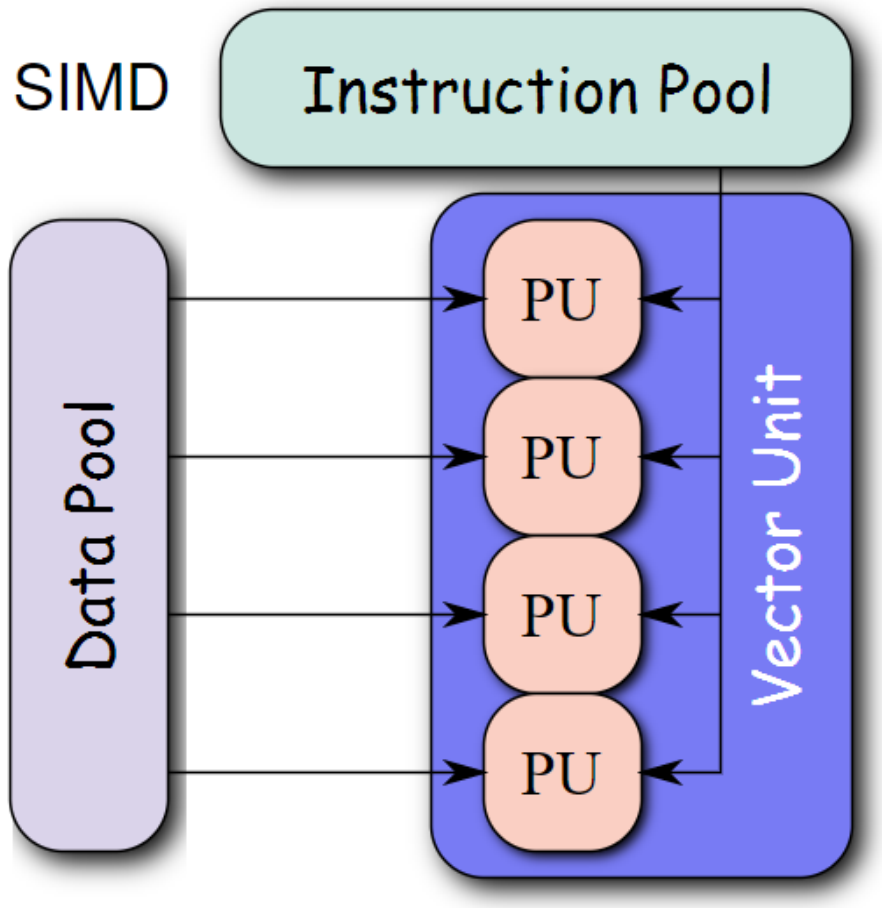


Image source: https://en.wikipedia.org/wiki/SIMD

```
#1_dot_simd.jl

function dot1(x, y)
    s = 0.0
    for i in 1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end

function dot2(x, y)
    s = 0.0
    @simd for i in 1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end
```

Dot product: output

```
12.900 µs (0 allocations: 0 bytes)
  1.760 µs (0 allocations: 0 bytes)
2.4592981764713477e7
2.4592981764713492e7
```

Green threading

- In computer programming, green threads are threads that are scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system. Green threads **emulate** multithreaded environments without relying on any native OS capabilities, and they are managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support.

https://en.wikipedia.org/wiki/Green_threads

Green threads in Julia

```
julia>  @time sleep(2)
  2.004142 seconds (11 allocations: 320 bytes)

julia> @time @async sleep(2)
  0.000128 seconds (27 allocations: 2.250 KiB)
Task (runnable) @0x0000000012c22e10
```

Green threads - sample

```
function dojob(i)
    val = rand()
    sleep(val)   # this could be external computations with I/O
    i, val
end

result = Vector{Tuple{Int,Float64}}(undef, 8)

@sync for i=1:8
   @async x[i] = dojob(i)
end
```

A simple web server with green threading

```
server = Sockets.listen(9991)
contt = Ref(true)
while contt[]
    sock = Sockets.accept(server)
    @async begin
        data = readline(sock)
        print("Got request:\n", data, "\n")
        cmd = split(data, " ")[2][2:end]
        println(sock, "\nHTTP/1.1 200 OK\nContent-Type: text/html\n")
        contt[] = contt[] && (!occursin("stopme", data))
        if contt[]
             println(sock, string("<html><body>", cmd, "=", eval(Meta.parse(cmd)), "</body></html>"))
        else
            println(sock,"<html><body>stopping</body></html>")
        end
        close(sock)
    end
end
println("Server stopped")
```

Comparison of parallelism types

Threading

- Single process (cheap)
- Shared memory
- Number of threads running simultaneously limited by number of processors
- Possible issues with locking and false sharing

Multiprocessing

- Multiple processes
- Separate memory
- Number of processes running simultaneously limited by cluster size
- Possible issues if inter-process communication is needed

Distributed computing

- Multiple hosts
- Multiple processes (single- or multi-threaded)

Julia command line option for parallelism

multi-threading

`-t, --threads {N|auto}`	Enable N threads; "auto" currently sets N to the number of local CPU threads but this might change in the future

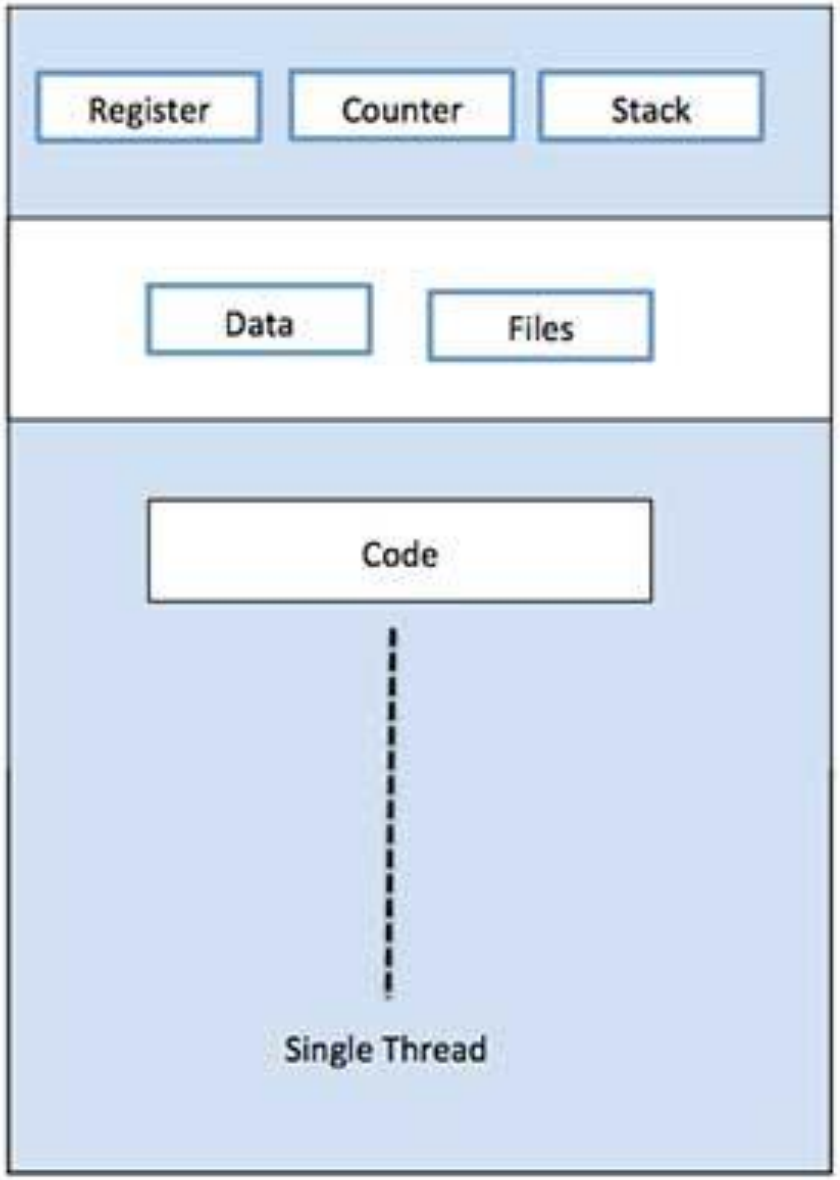
multi-processing

`-p, --procs {N|auto}`	Integer value N launches N additional local worker processes "auto" launches as many workers as the number of local CPU threads (logical cores)

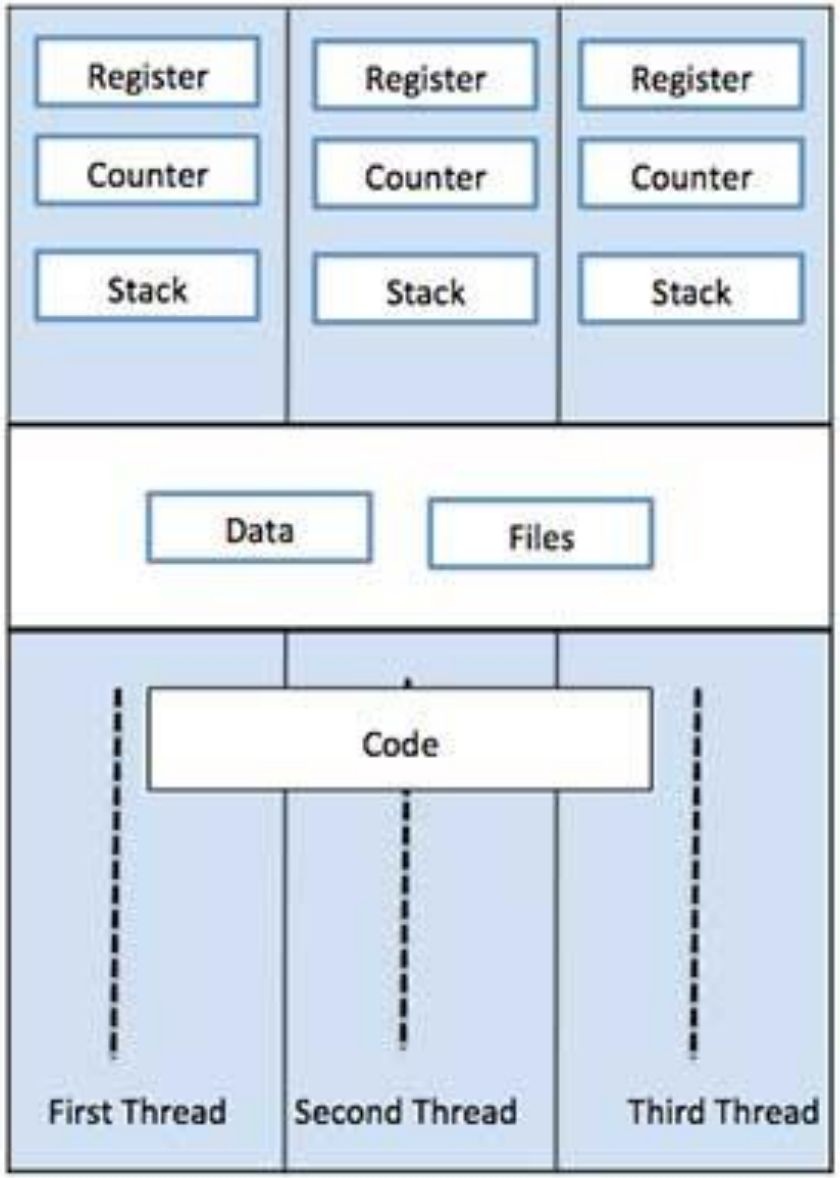
distributed computing

`--machine-file <file>`	Run processes on hosts listed in <file>

Threading



Single Process P with single thread



Single Process P with three threads

Simple example – threading

Single threaded

```
function ssum(x)
    r, c = size(x)
    y = zeros(c)
    for i in 1:c
        for j in 1:r
            @inbounds y[i] += x[j, i]
        end
    end
    y
end
```

Multithreading

```
function tsum(x)
    r, c = size(x)
    y = zeros(c)
    Threads.@threads for i in 1:c
        for j in 1:r
            @inbounds y[i] += x[j, i]
        end
    end
    y
end
```

Sum: output

```
threads: 1
  0.721701 seconds (2 allocations: 156.328 KiB)
  0.858513 seconds (2 allocations: 156.328 KiB)
  0.730469 seconds (60.27 k allocations: 3.834 MiB, 8.09% compilation time)
  0.914933 seconds (8 allocations: 156.938 KiB)

threads: 2
  0.750043 seconds (2 allocations: 156.328 KiB)
  0.882158 seconds (2 allocations: 156.328 KiB)
  0.415769 seconds (60.29 k allocations: 3.835 MiB, 14.25% compilation time)
  0.341232 seconds (14 allocations: 157.469 KiB)

threads: 4
  0.709045 seconds (2 allocations: 156.328 KiB)
  0.690816 seconds (2 allocations: 156.328 KiB)
  0.261915 seconds (60.32 k allocations: 3.838 MiB, 23.98% compilation time)
  0.224480 seconds (24 allocations: 158.484 KiB)
```

delta is compilation time

Threading: synchronization

Increment x 10^6 times using threads:

- Atomic operations
- SpinLock (busy waiting)
- Mutex (OS provided lock)

Locking: output

1 thread

Row	f	i	value	timems
1	f_bad	1	10000000	65.416
2	f_bad	2	10000000	47.856
3	f_atomic	1	10000000	12.144
4	f_atomic	2	10000000	5.9725
5	f_spin	1	10000000	116.92
6	f_spin	2	10000000	107.22
7	f_reentrant	1	10000000	175.71
8	f_reentrant	2	10000000	180.96

4 threads

Row	f	i	value	timems
1	f_bad	1	**2500984**	32.145
2	f_bad	2	**2501422**	30.683
3	f_atomic	1	10000000	26.352
4	f_atomic	2	10000000	18.803
5	f_spin	1	10000000	186.86
6	f_spin	2	10000000	185.11
7	f_reentrant	1	10000000	998.39
8	f_reentrant	2	10000000	1003.6

Example – multiprocessing

```
function s_rand()
    n = 10^4
    x = 0.0
    for i in 1:n
        x += sum(rand(10^4))
    end
    x / n
end

@time s_rand()
@time s_rand()
```

```
using Distributed


function p_rand()
    n = 10^4
    x = @distributed (+) for i in 1:n
        sum(rand(10^4))
    end
    x / n
end

@time p_rand()
@time p_rand()
```

Rand: output

```
0.685125 seconds (20.01 k allocations: 763.722 MiB, 16.21% gc time)
0.642642 seconds (20.00 k allocations: 763.702 MiB, 15.37% gc time)
0.927246 seconds (356.68 k allocations: 783.715 MiB, 11.54% gc time, 26.08% compilation time)
0.471032 seconds (20.03 k allocations: 763.704 MiB, 14.47% gc time)
```

Parallelizing Julia code

- `@distributed`
- `@spawnat`
- `@everywhere`
- `@async`
- `@sync`
- `fetch()`

Typical pattern for distributed computation

```
using Distributed
addprocs(4);  # instead -p

@everywhere include("worker_setup.jl")

function init_worker()
   Random.seed!(myid())
end

@sync for wid in workers()
    @async fetch(@spawnat wid init_worker())
end
```

Writing distributed loops

```
data = @distributed (append!) for (i, j) =
    vec(collect(Iterators.product(1:4, 1:5)))
    a = rand(1:499)
    b = rand(1:9)*1000
    c = calc(a, b)
    DataFrame(;i,j,a,b,c,procid = myid())
end
```

Typical computation distribution pattern

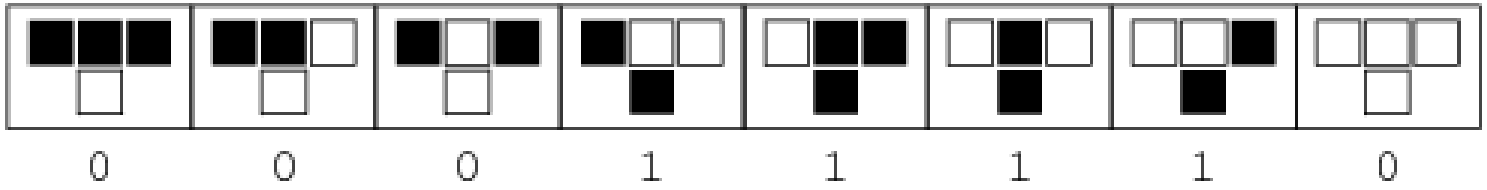
```
@everywhere function f()
      # do something
      return sum(rand(10000))
end

@sync for w in workers()
      @async begin
            res = @spawnat w f()
            values[w-1]=fetch(res)
      end
end
```

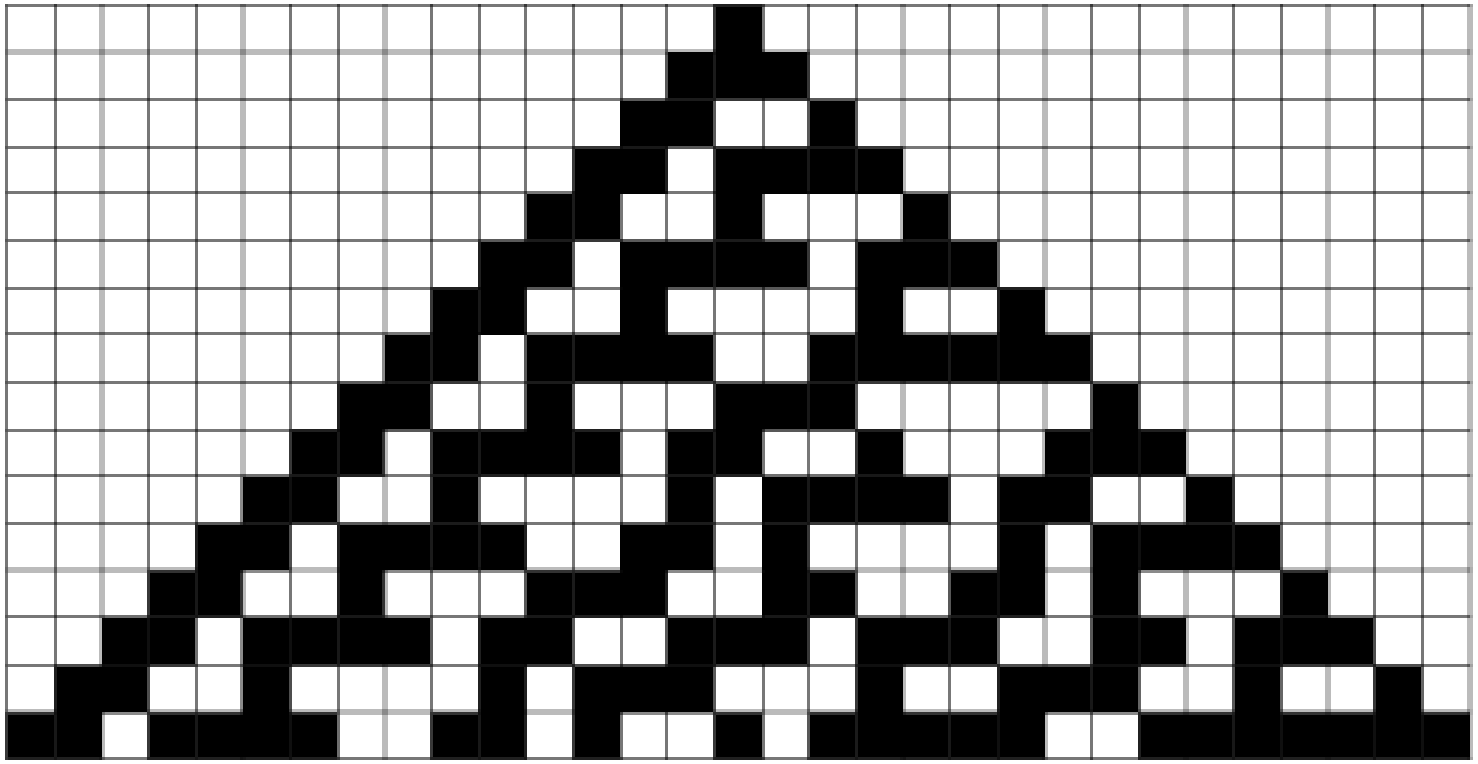
Sending data across cluster nodes

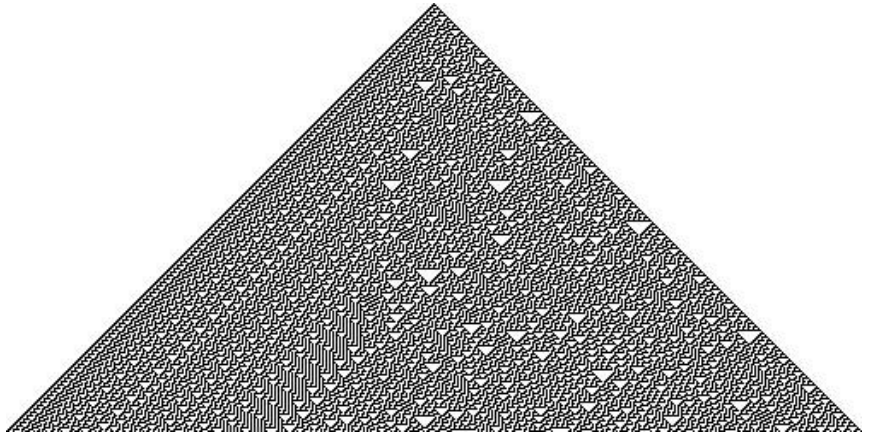
```
@everywhere using ParallelDataTransfer

sendto(workerid, vara = vara)

sendto([workerid1, workerid2], varb = varb)
```

Cellular automaton

rule 30



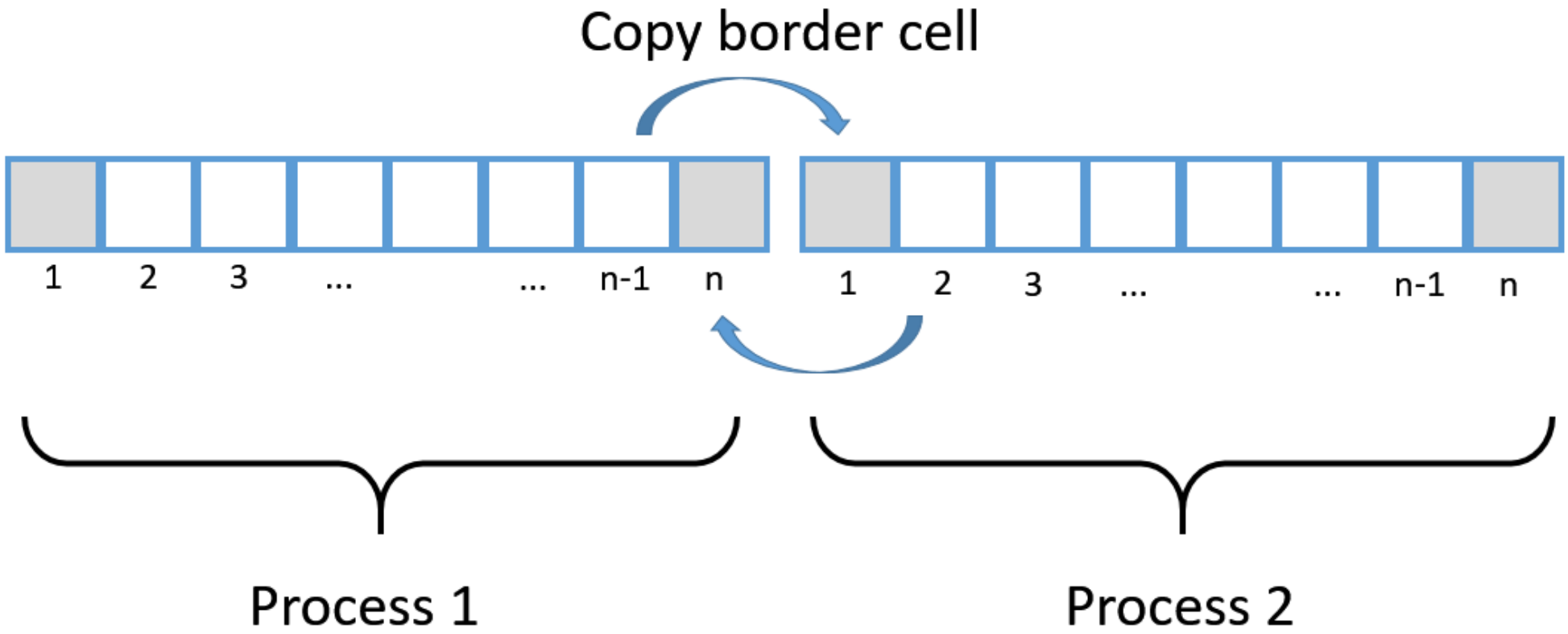




Source:
http://mathworld.wolfram.com/Rule30.html

Distributed cellular automaton

- Distributing data among worker processes



cellular automaton

```
using Distributed
@everywhere using ParallelDataTransfer

@everywhere function rule30(ca::Array{Bool})
    lastv = ca[1]
    for i in 2:(length(ca)-1)
        current = ca[i]
        ca[i] = xor(lastv, ca[i] || ca[i+1])
        lastv = current
    end
end
```

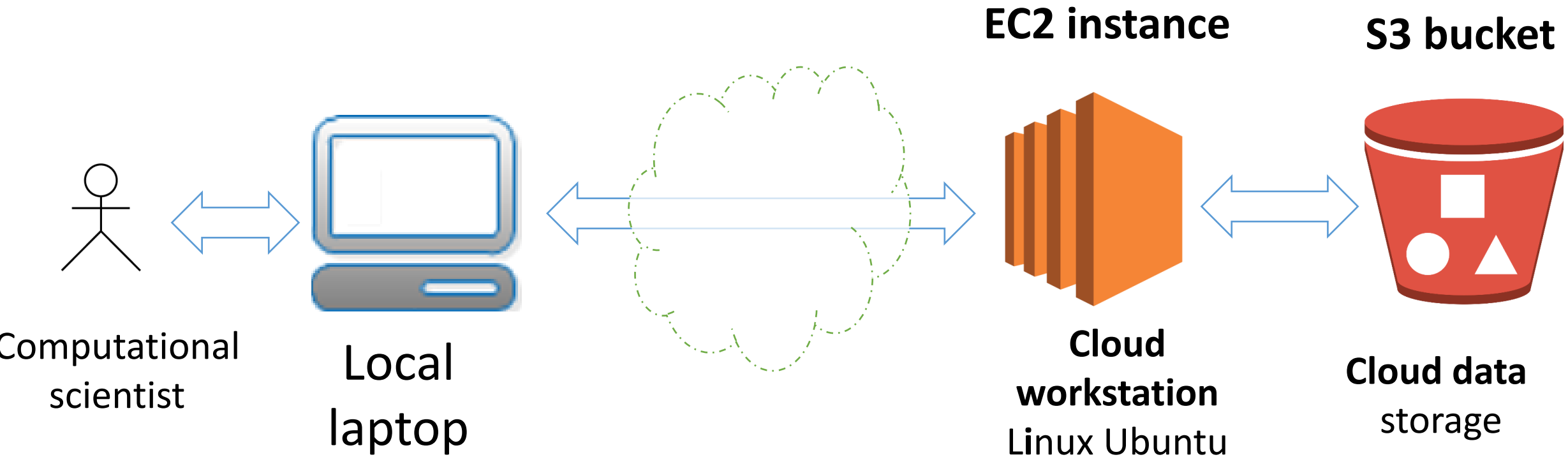
```
@everywhere function getsetborder()
    caa[1] = (@fetchfrom neighbours[1] getcaa[end-1])
    caa[end] = (@fetchfrom neighbours[2] getcaa[2])
end
```

```
function runca(steps::Int, visualize::Bool)
    @sync for w in workers()
        @async @fetchfrom w fill!(caa, false)
    end
    @fetchfrom wks[Int(nwks/2)+1] caa[2]=true
    visualize && printsimdist(workers())
    for i in 1:steps
        @sync for w in workers()
            @async @fetchfrom w getsetborder(caa, neighbours)
        end
        @sync for w in workers()
            @async @fetchfrom w rule30(caa)
        end
        visualize && printsimdist(workers())
    end
end
```

Running the cellular automaton

```
wks = workers()
nwks = length(wks)
for i in 1:nwks
    sendto(wks[i],neighbours = (i==1 ? wks[nwks] : wks[i-1],
                                i==nwks ? wks[1] : wks[i+1]))
    fetch(@defineat wks[i] const caa = zeros(Bool, 15+2));
end

runca(20,true)
```

A typical „single server in the cloud” configuration



Connecting to a cloud instance

- Connect to the cloud server

```
$ ssh -i keyfile.pem ubuntu@ec2-18-218-237-1.us-east-2.compute.amazonaws.com
```

- Copy a local file (note the slash type on Windows)

```
$ scp -i keyfile.pem c:\temp/local.txt ubuntu@ec2-18-218-237-1.us-east-2.compute.amazonaws.com:/home/ubuntu/
```

- Copy a local folder (note the slash type on Windows)

```
$ scp -r -i c:\temp\folder ubuntu@ec2-18-218-237-1.us-east-2.compute.amazonaws.com:/home/ubuntu/
```

Notes:

Mac OSX and Linux: run `chmod 600 keyfile.pem` before using the keyfile

Windows: The best SSH/SCP for windows is contained within Git: https://git-scm.com/download/win

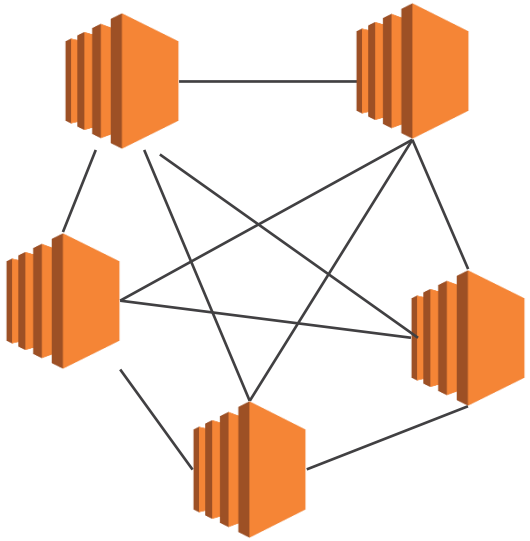
Typical enviroment

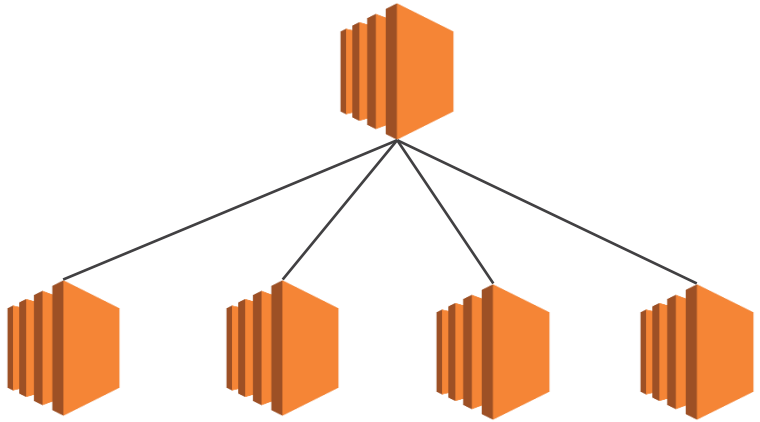
- Cluster controller
  - Machine type: t2.micro (free tier)
  - Number of machines: 1
- Cluster nodes
  - Machine type: c5.large (Spot Fleet)
  - Number of machines: 5

- You should know how to:
  - Basic understanding of HTTP protocol
  - Use console
  - Use SSH client
- All work throughout the workshop will take place in the cloud
  (no local software except for an SSH client a web browser is required)

What is a computing cluster

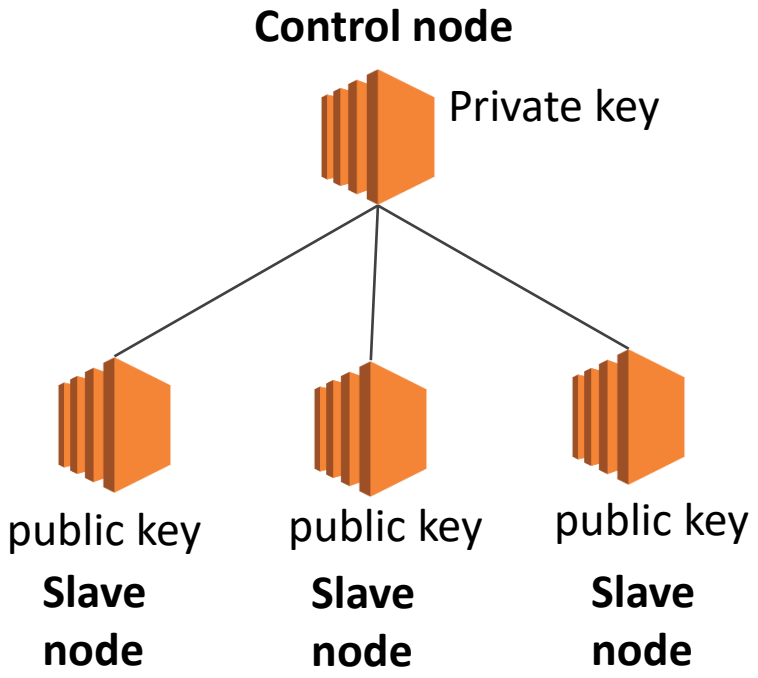
A group of computers working towards achieving a common goal.





A computer within a cluster is called a **node**

Passwordless SSH cluster



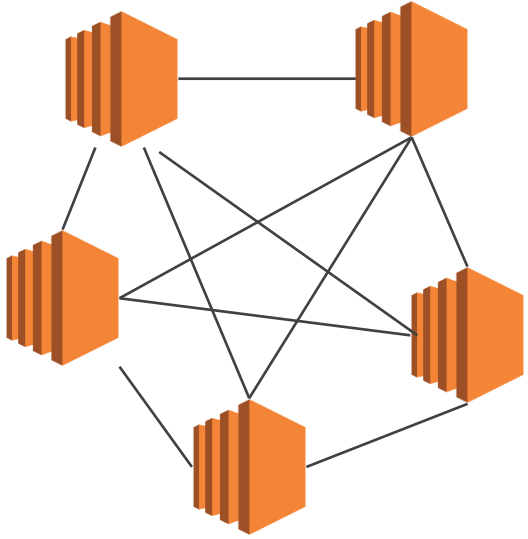
- Enables direct execution of commands on other nodes via command:
  `ssh user@hostname command`
- Environments such as Gnu R parallel and Julia parallel use this feature to spawn processes on worker nodes
- Control node has the private key while each slave node needs to have the public key in the `~/.ssh/authorized_users` file
- Network connections between nodes should be open
  - Create a SecurityGroup that allows unlimited connections to itself.

Setting up SSH for passwordless SSH

- `ssh-keygen -P "" -t rsa -f ~/.ssh/cluster`
- `printf "\nUser ubuntu\nPubKeyAuthentication yes\nStrictHostKeyChecking no\nIdentityFile ~/.ssh/cluster\n" >> ~/.ssh/config`
- `cat ~/.ssh/cluster.pub >> ~/.ssh/authorized_keys`

- Test configuration:
  - `$ ssh localhost`

Paswordless SSH cluster



Self-referencing SecurityGroup

Name	Group ID	Group Name	VPC ID	Description
	sg-ebdb1b80	launch-wizard-40	vpc-c30fe6aa	launch-wizard-40 created 2018-01-25T21:55:12.263+01:...

Security Group: sg-ebdb1b80

Description | **Inbound** | Outbound | Tags

Edit

Type	Protocol	Port Range	Source	Description
All traffic	All	All	sg-ebdb1b80 (launch-wizard-40)	
SSH	TCP	22	0.0.0.0/0	

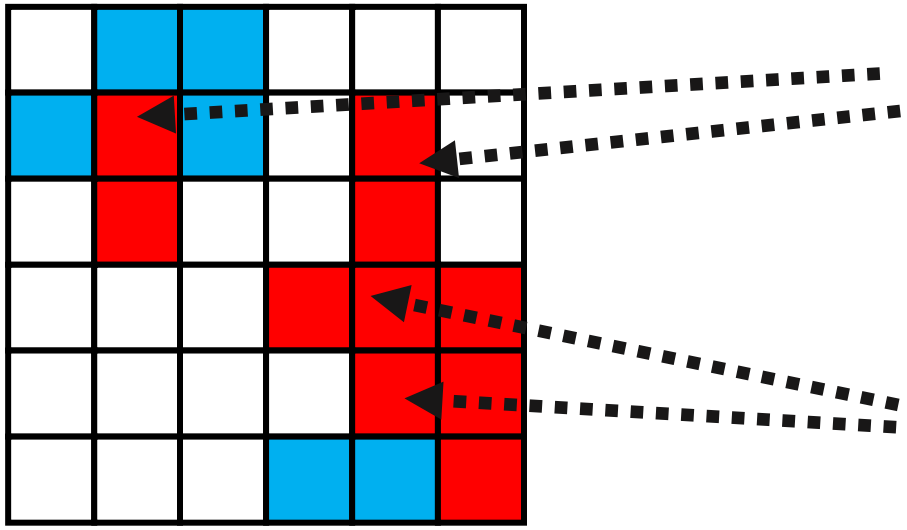
Julia cluster specification file and running distributed clusters

```
$ more machinefile_julia
4*ubuntu@172.31.10.229
4*ubuntu@172.31.11.44
4*ubuntu@172.31.0.243
4*ubuntu@172.31.13.134
4*ubuntu@172.31.14.219

$ julia –machine-file machinefile_julia program.jl

# REQUIRES PASSWORDLESS SSH TO BE CONFIGURED!
```

Use case scenario: Performance of distributed code in Julia
Cray vs AWS

Schelling (1974) segregation model

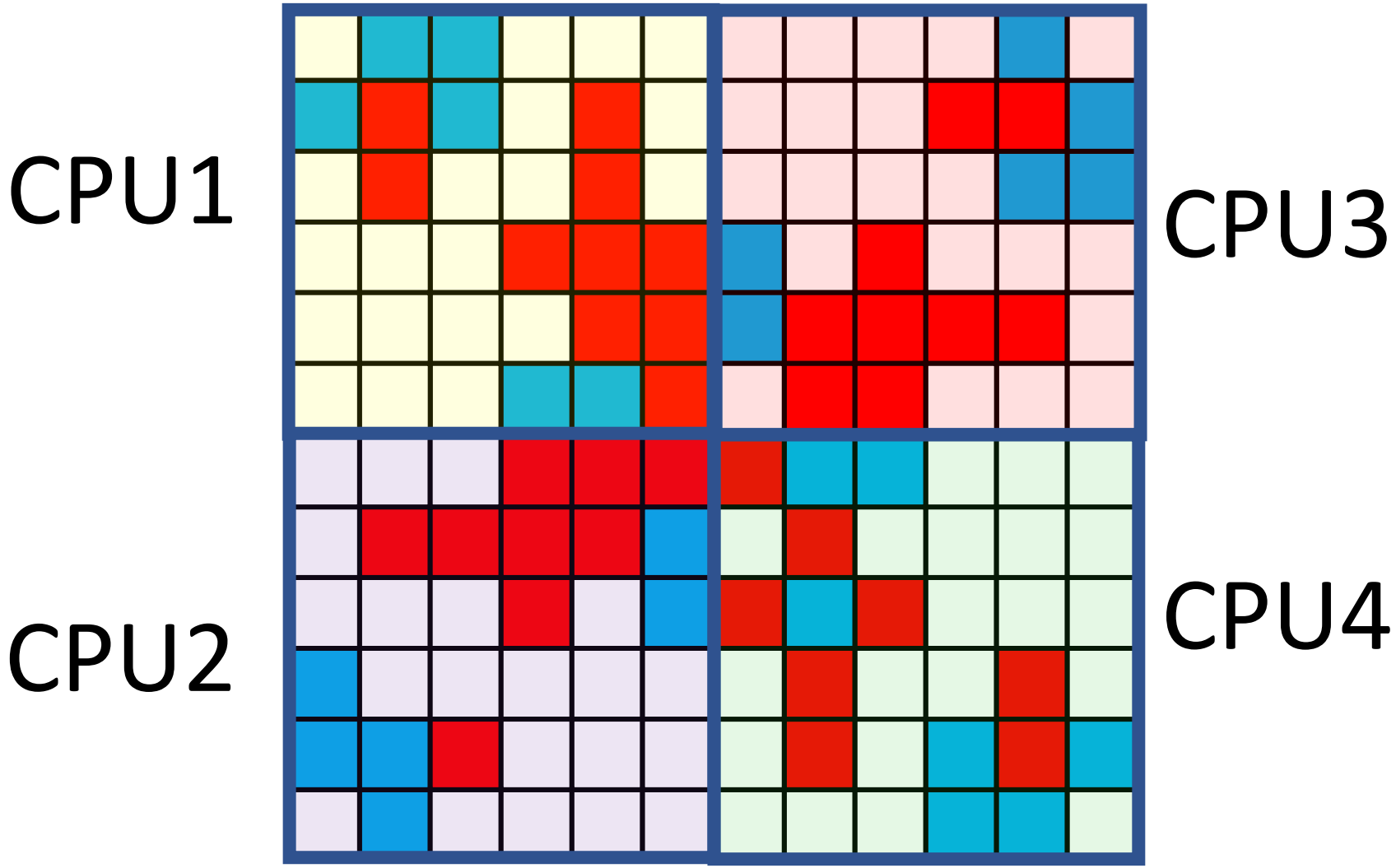
- Agents occupy cells of rectangular space
- Two types of agents (e.g. blue and red)
- When not happy with their neighbours randomly relocate



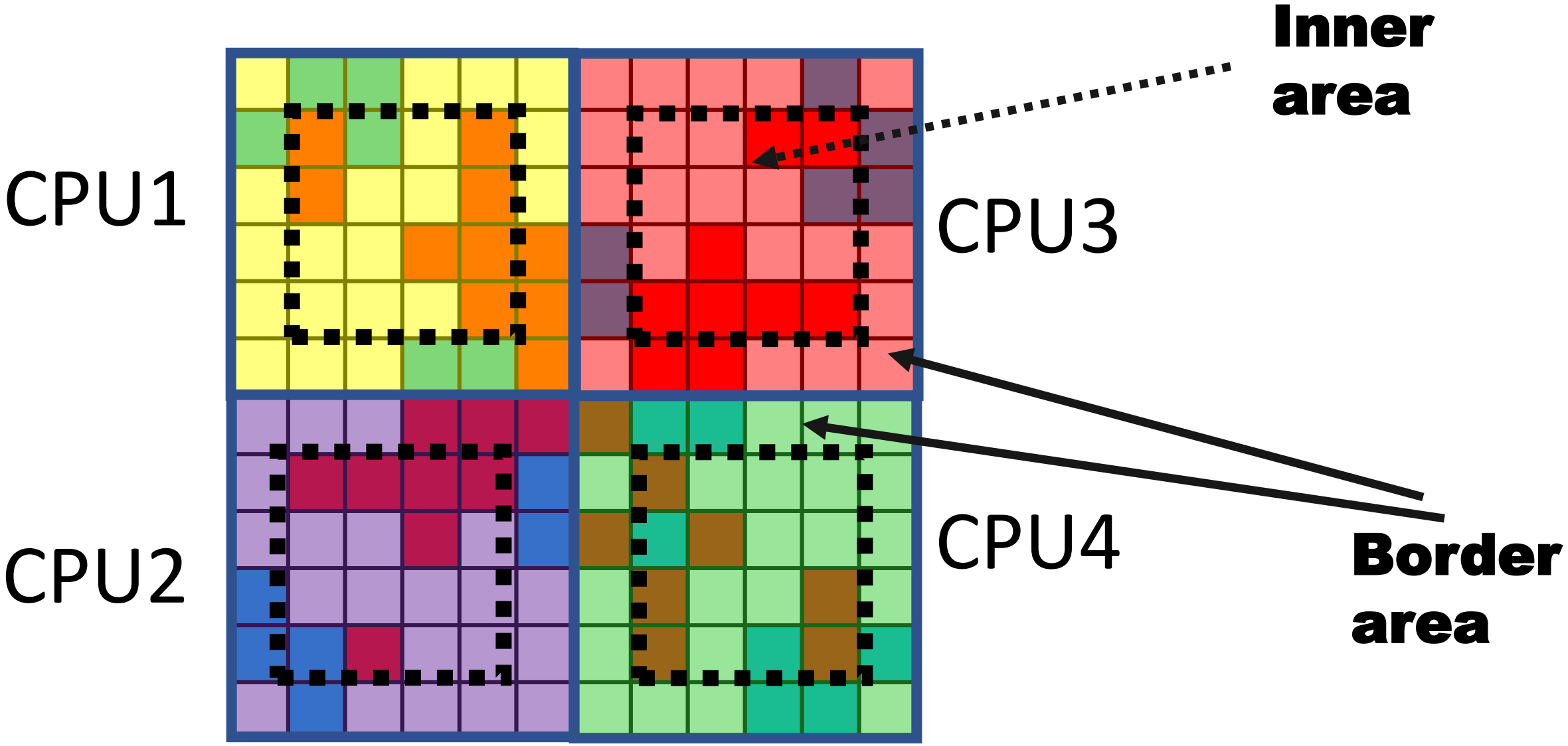
☹ Unhappy
(has too few neighbours of the same color)

☺ Happy
(has 5 neighnours of the same color)

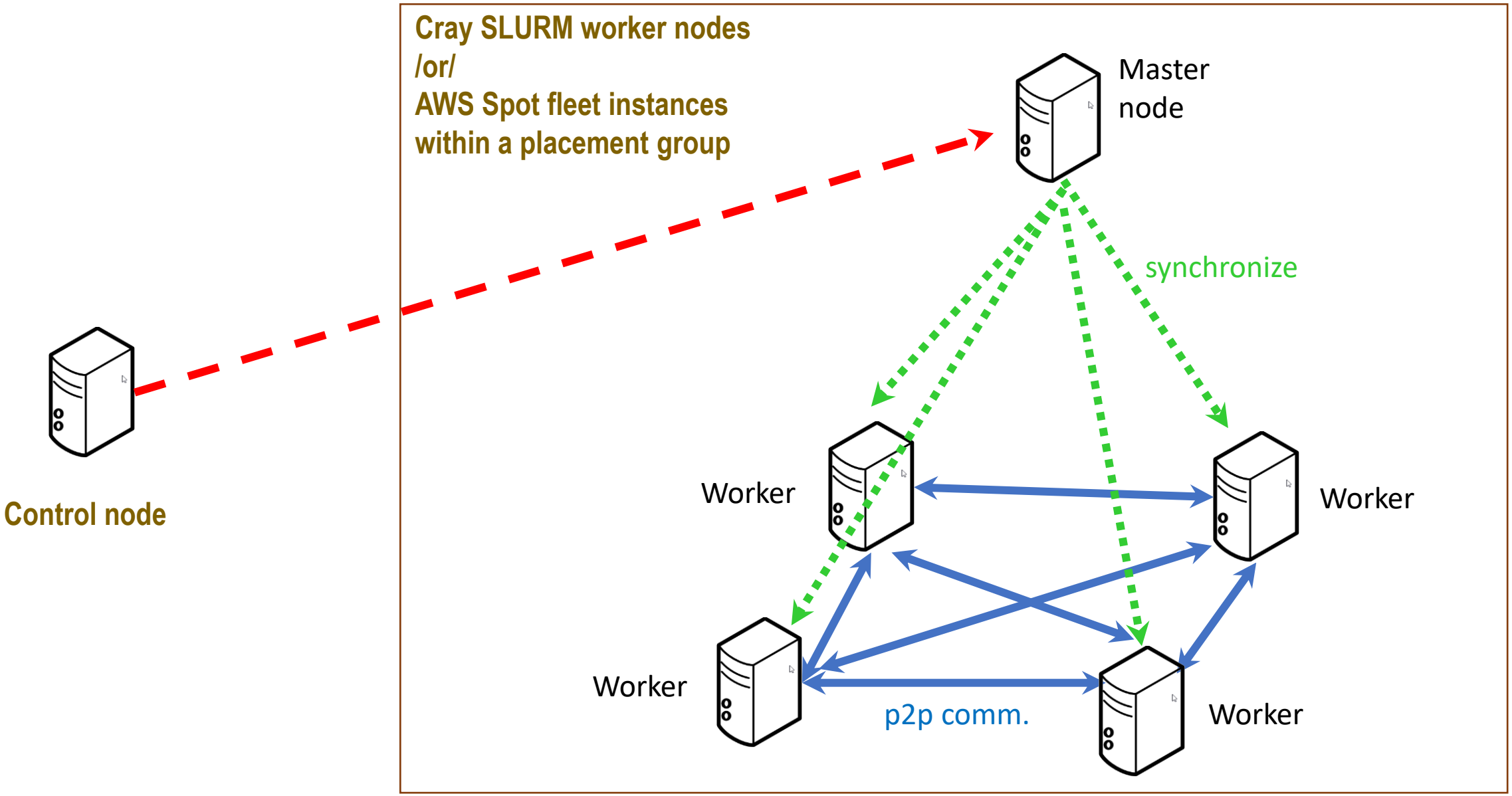
Distributed Schelling segregation model



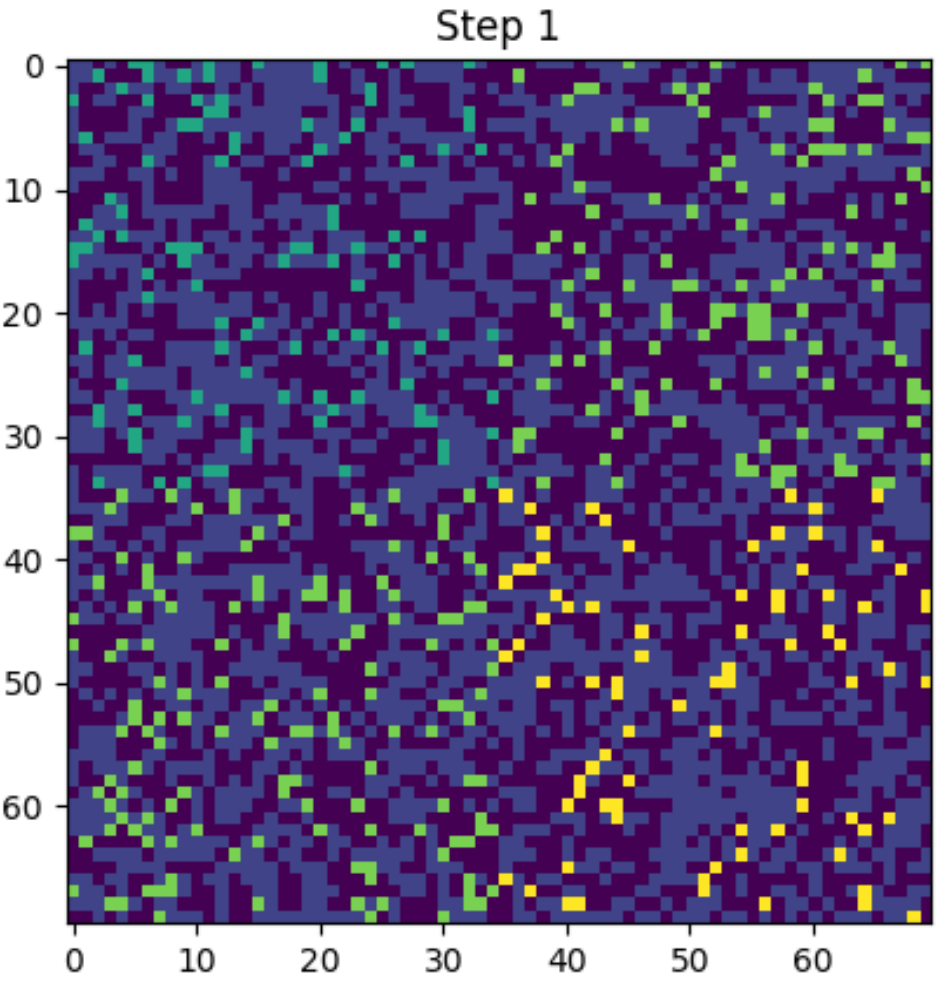
Distributed Schelling segregation model

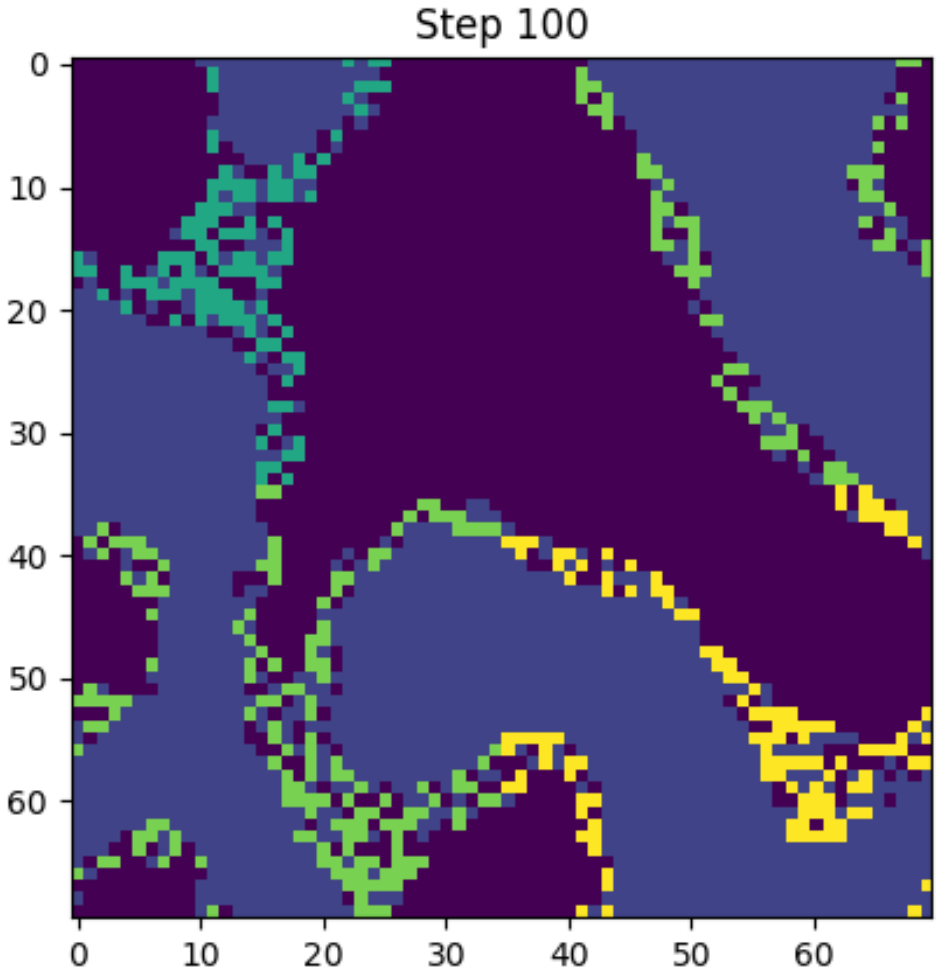


Distributed simulation architecture



Parallelized Schelling model (2x2)



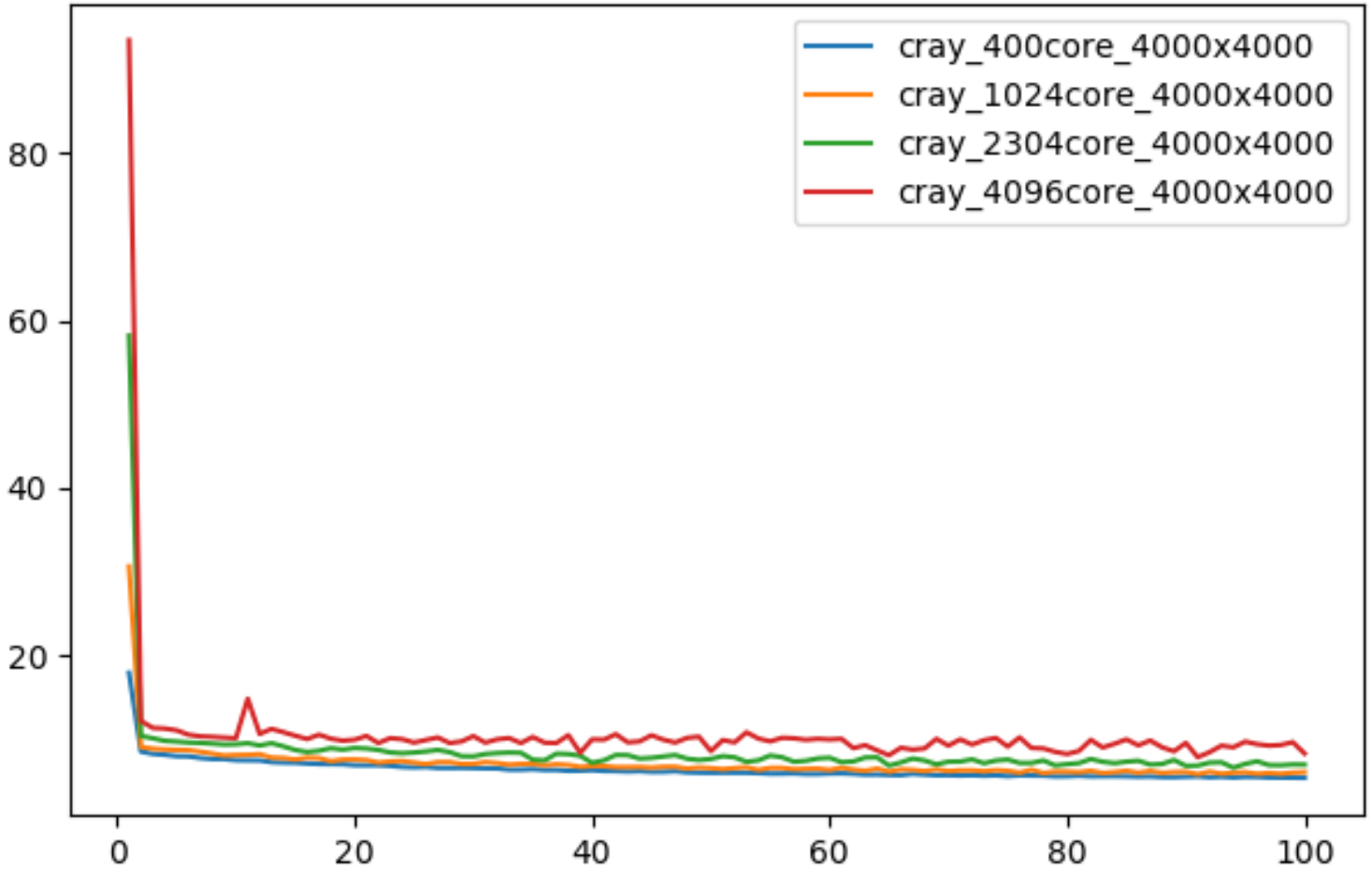


Parralelizing Julia on Cray with SLURM

```
julia> using ClusterManagers
julia> addprocs_slurm(200,job_name="my_job",
     account="GC71-37", time="00:10:00",
     exename="/lustre/tetyda/home/pszufe/julia/usr/bi
n/julia")
```

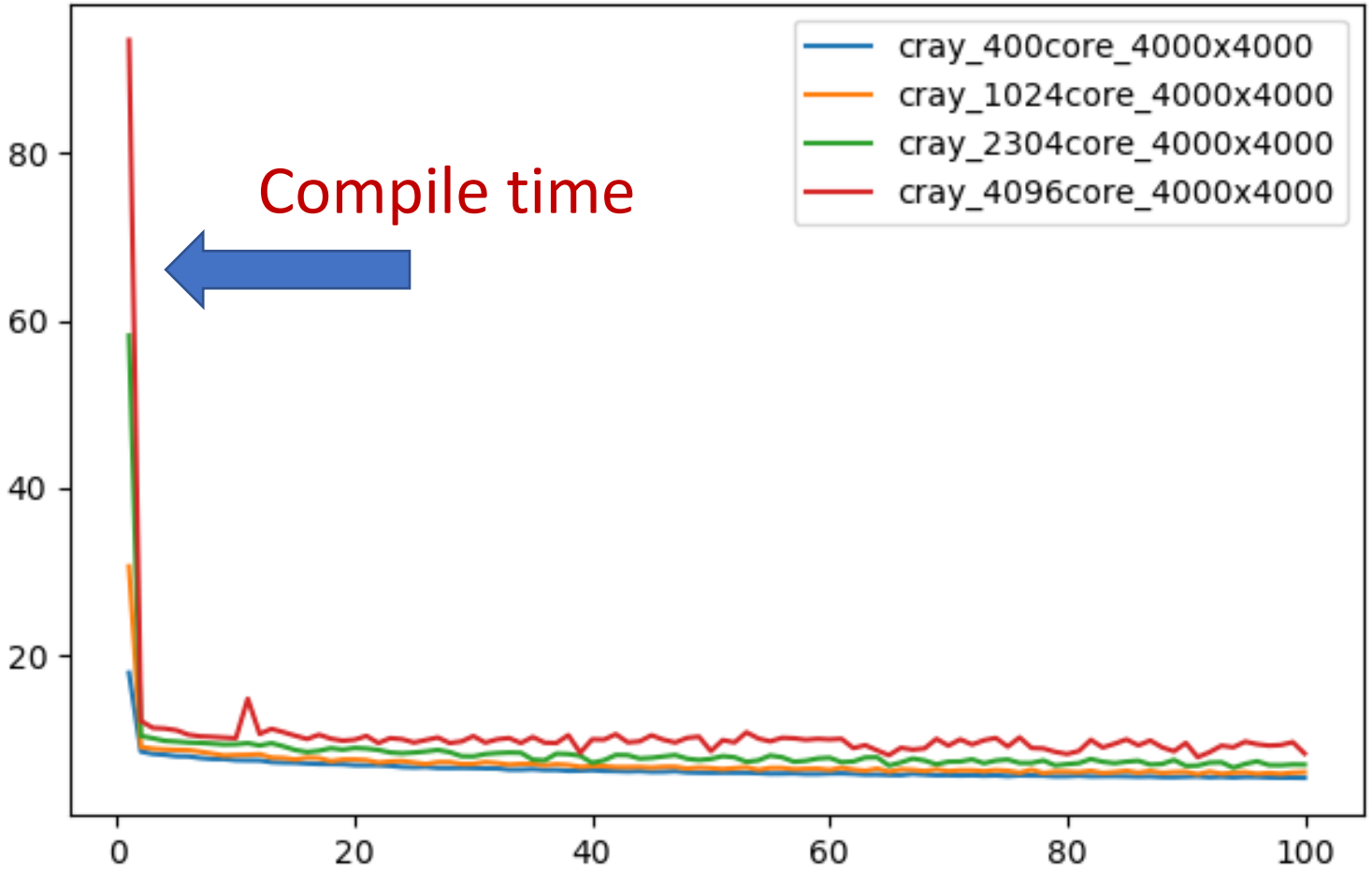
Typical Julia performance pattern ….

Time to execute step (seconds)



Simulation step

Typical Julia performance pattern ….

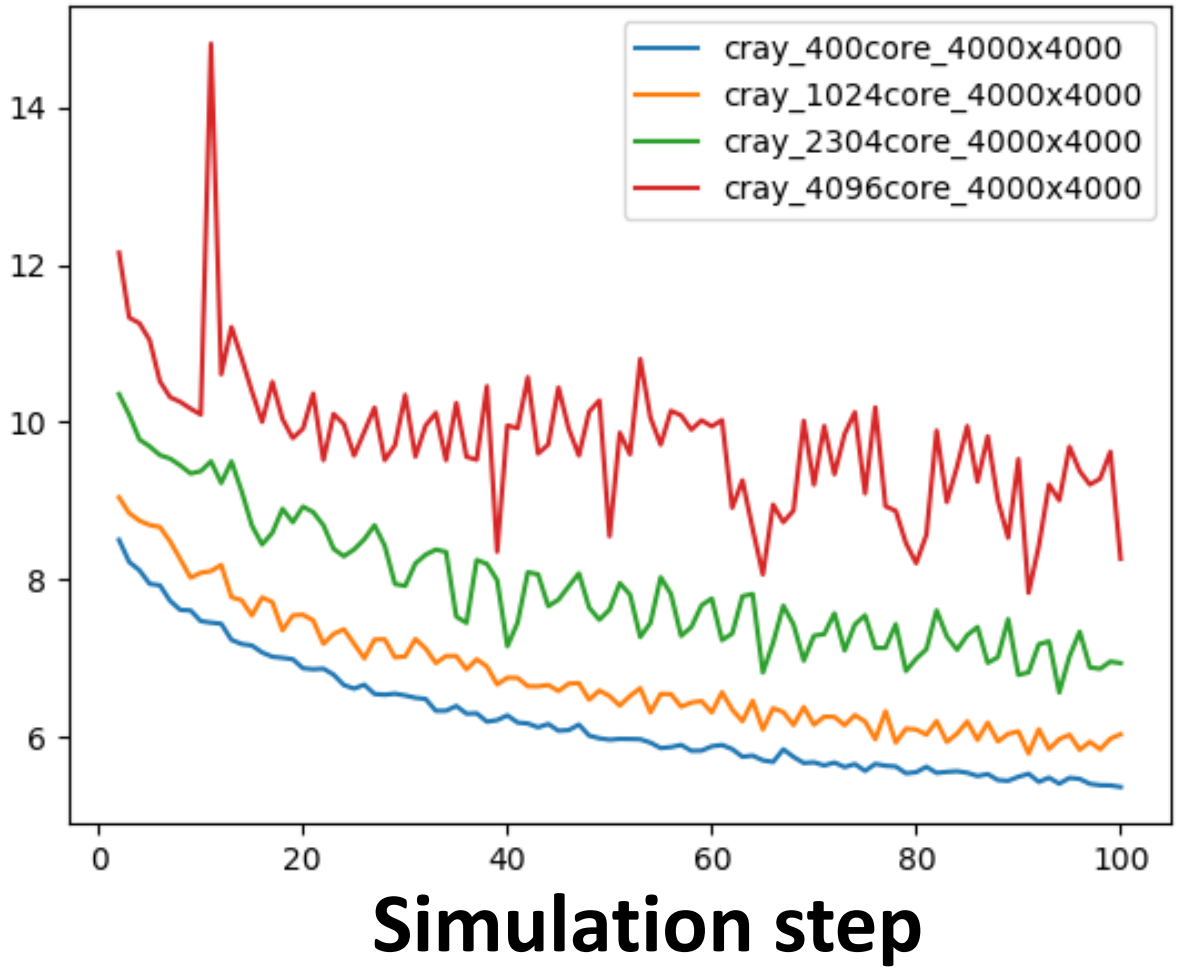


Time to execute step (seconds)

Simulation step

Distributed simulation scalability

Time to execute step (seconds)



Simulation step

Note:

The first step has been removed

Cray vs AWS Spot Fleet