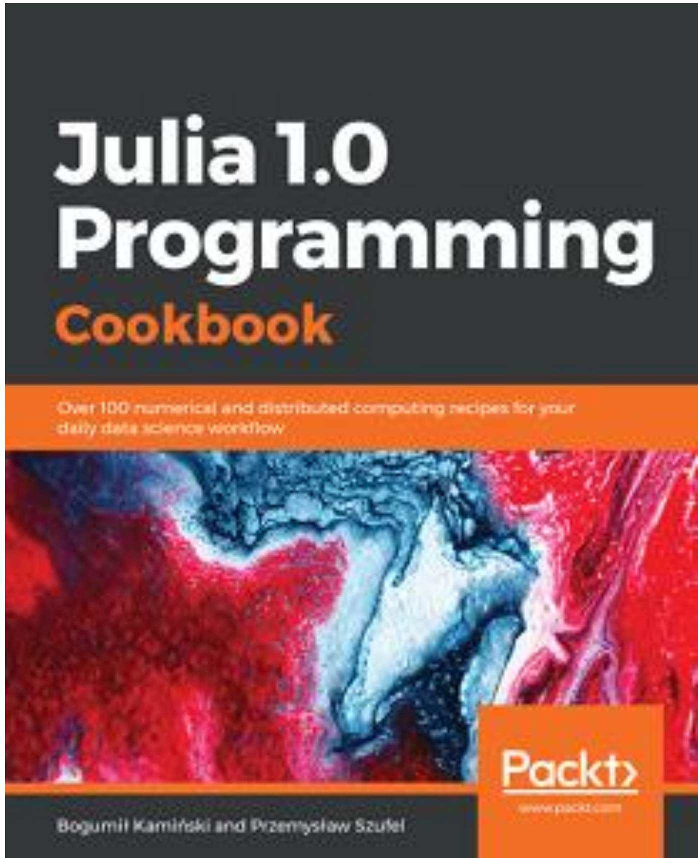
Parallel and distributed computing with Julia

Przemysław Szufel, PhD

Warsaw School of Economics

https://szufel.pl/unisa/

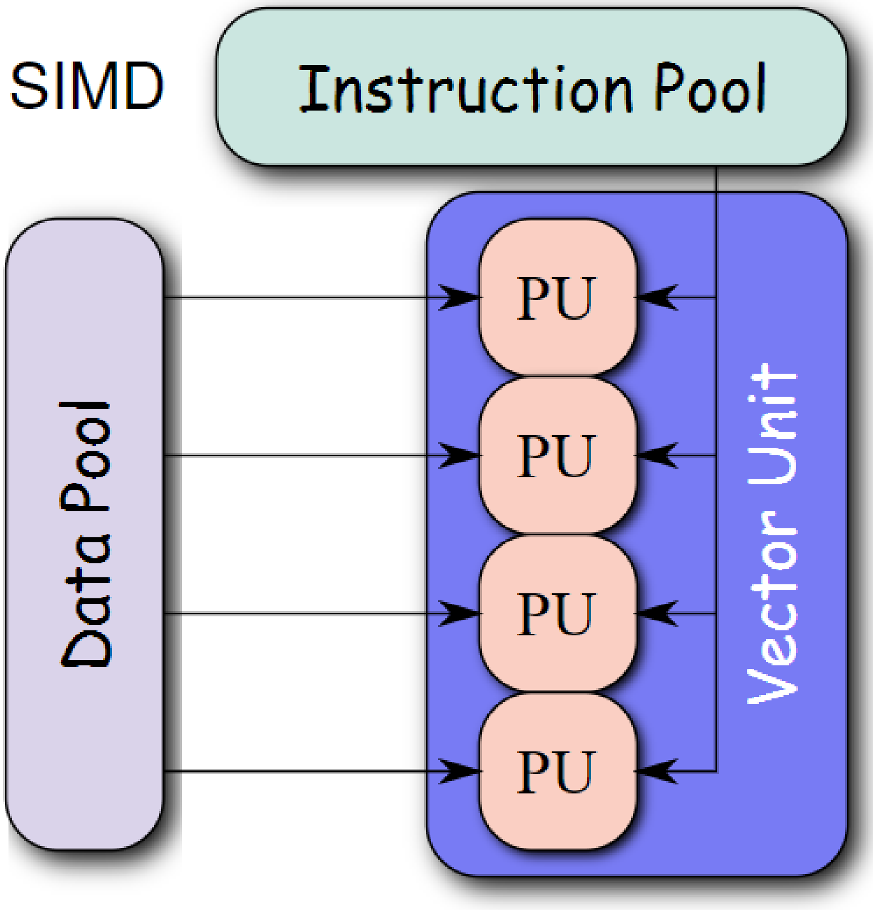


Parallelization options in programming languages

- Single instruction, multiple data (SIMD)
- Green-threads
- Multi-threading
  - Language
  - Libraries
- Multi-processing
  - single machine
  - distributed (cluster)
  - distributed (cluster) via external tools

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

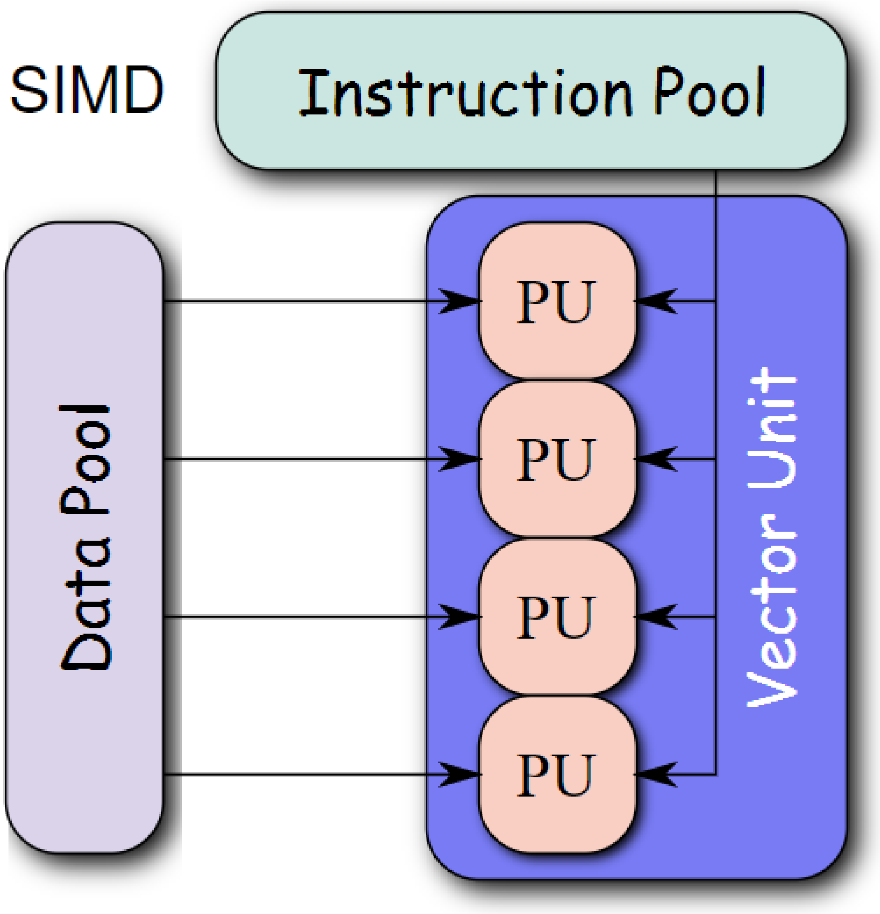


Image source: https://en.wikipedia.org/wiki/SIMD

```
#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

```
$ julia 1_dot_simd.jl
  113.066 ns (0 allocations: 0 bytes)
  21.760 ns (0 allocations: 0 bytes)
```

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

A simple web server with green threading

2_webserver.jl

```
server = Sockets.listen(8080)
while true
    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")
        println(sock, string("<html><body>", cmd, "=",
                        eval(Meta.parse(cmd)), "</body></html>"))
        close(sock)
    end
end
```

Comparison of parallelism types

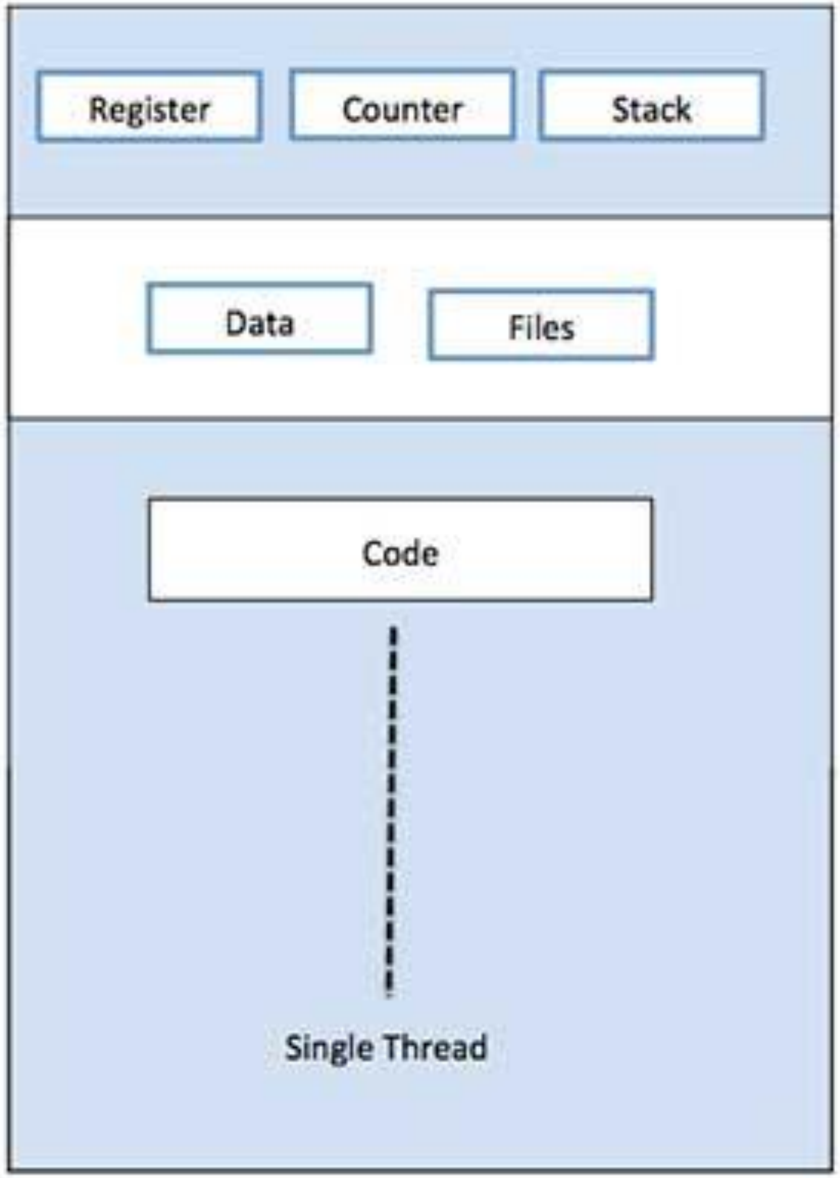
Threading

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

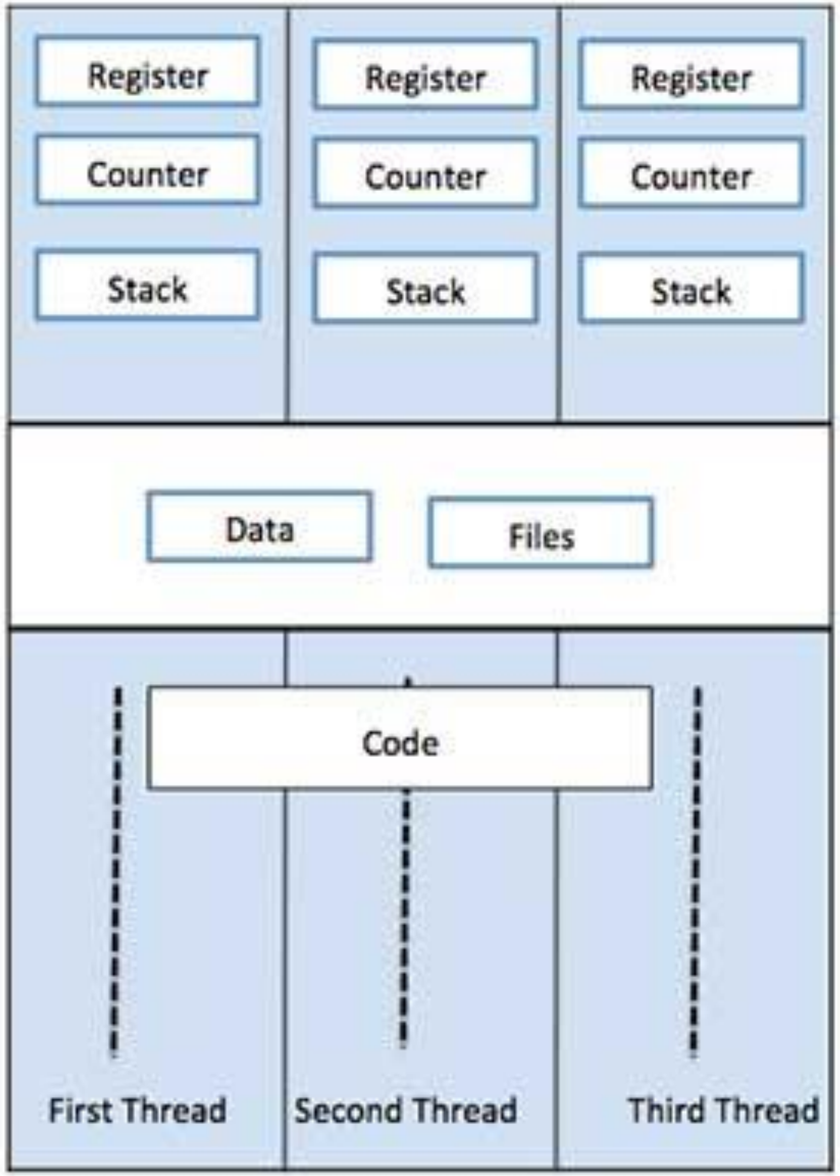
Multiprocessing

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

Threading



Single Process P with single thread



Single Process P with three threads

Simple example – threading
`3_sum_thread.jl`

Single threaded

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

Sum: output

```
$ ./3_run_sum_thread.sh
threads: 1
  0.963284 seconds (28.38 k allocations: 1.619 MiB)
  0.878460 seconds (6 allocations: 156.484 KiB)
  1.782968 seconds (53.94 k allocations: 2.891 MiB)
  1.764061 seconds (7 allocations: 156.531 KiB)
threads: 2
  0.799932 seconds (28.38 k allocations: 1.619 MiB)
  0.813326 seconds (6 allocations: 156.484 KiB)
  0.925774 seconds (53.94 k allocations: 2.891 MiB)
  0.891011 seconds (7 allocations: 156.531 KiB)
threads: 4
  0.779453 seconds (28.38 k allocations: 1.619 MiB)
  0.828074 seconds (6 allocations: 156.484 KiB)
  0.555077 seconds (53.94 k allocations: 2.891 MiB)
  0.556130 seconds (7 allocations: 156.531 KiB)
```

delta is compilation time

Threading: synchronization
`4_locking.jl`

Increment x 10^7 times using threads:

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

```
$ ./4_run_locking.sh
```

Locking: output on c4.4xlarge (16 vCPU)

1 thread

```
f_bad
10000000
  0.498997 seconds (10.01 M allocations: 153.318 MiB, 49.89% gc time)
10000000
  0.198711 seconds (10.00 M allocations: 152.580 MiB, 3.04% gc time)
f_atomic
10000000
  0.082628 seconds (7.54 k allocations: 403.376 KiB)
10000000
  0.059487 seconds (11 allocations: 288 bytes)
f_spin
10000000
  0.286315 seconds (10.01 M allocations: 153.074 MiB, 2.25% gc time)
10000000
  0.257490 seconds (10.00 M allocations: 152.580 MiB, 1.52% gc time)
f_mutex
10000000
  0.557977 seconds (10.01 M allocations: 153.260 MiB, 1.17% gc time)
10000000
  0.491197 seconds (10.00 M allocations: 152.580 MiB, 1.02% gc time)
```

16 threads

```
f_bad
950043
  0.449196 seconds (1.63 M allocations: 27.759 MiB)
630661
  0.922549 seconds (1.52 M allocations: 26.963 MiB, 61.86% gc time)
f_atomic
10000000
  0.217921 seconds (7.54 k allocations: 403.376 KiB)
10000000
  0.187748 seconds (12 allocations: 688 bytes)
f_spin
10000000
  2.238537 seconds (10.01 M allocations: 153.074 MiB, 15.81% gc time)
10000000
  1.602330 seconds (10.00 M allocations: 152.581 MiB, 19.85% gc time)
f_mutex
10000000
  4.862945 seconds (10.01 M allocations: 153.260 MiB, 3.67% gc time)
10000000
  4.662214 seconds (10.00 M allocations: 152.580 MiB)
```

Example – multiprocessing

5_rand_process.jl

```julia
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()
```

```julia
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()
```

$ julia -p $(nproc) rand_process.jl

Rand: output

```
$ 3_rand/run_rand_process.sh
  0.381071 seconds (46.21 k allocations: 765.124 MiB, 37.20% gc time)
  0.161149 seconds (20.00 k allocations: 763.703 MiB, 9.64% gc time)
  1.661893 seconds (230.81 k allocations: 12.494 MiB, 0.15% gc time)
  0.092413 seconds (1.89 k allocations: 155.766 KiB)
```

delta is compilation and process spawning time

Parallelizing Julia code

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

Typical pattern for distributed simulation

```
using Distributed
addprocs(4);

@everywhere include("sim_file.jl")

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

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

Writing distributed loops

```
data = @distributed (vcat) for i = 1:10000
    some_param_A = rand()
    some_param_B = rand()
    res_1, res_2, res_3 = run_sim();
    (sim_stats(res_1,res_2,res_3)...,
     some_param_A,
     some_param_B,
    myid())
end
```

Typical computation distribution pattern

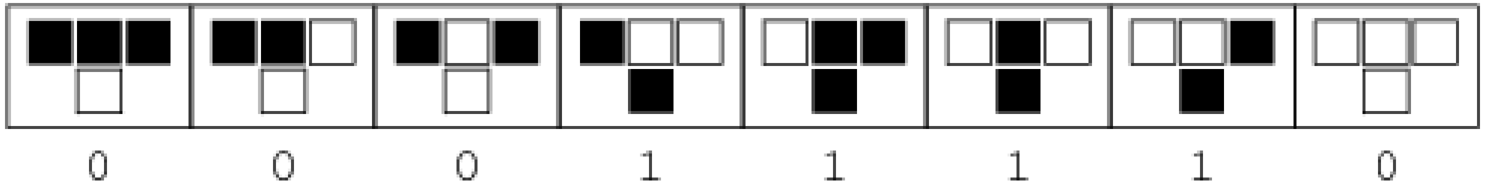
```julia
@everywhere function f()
    # do something
    return sum(rand(10000))
end

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

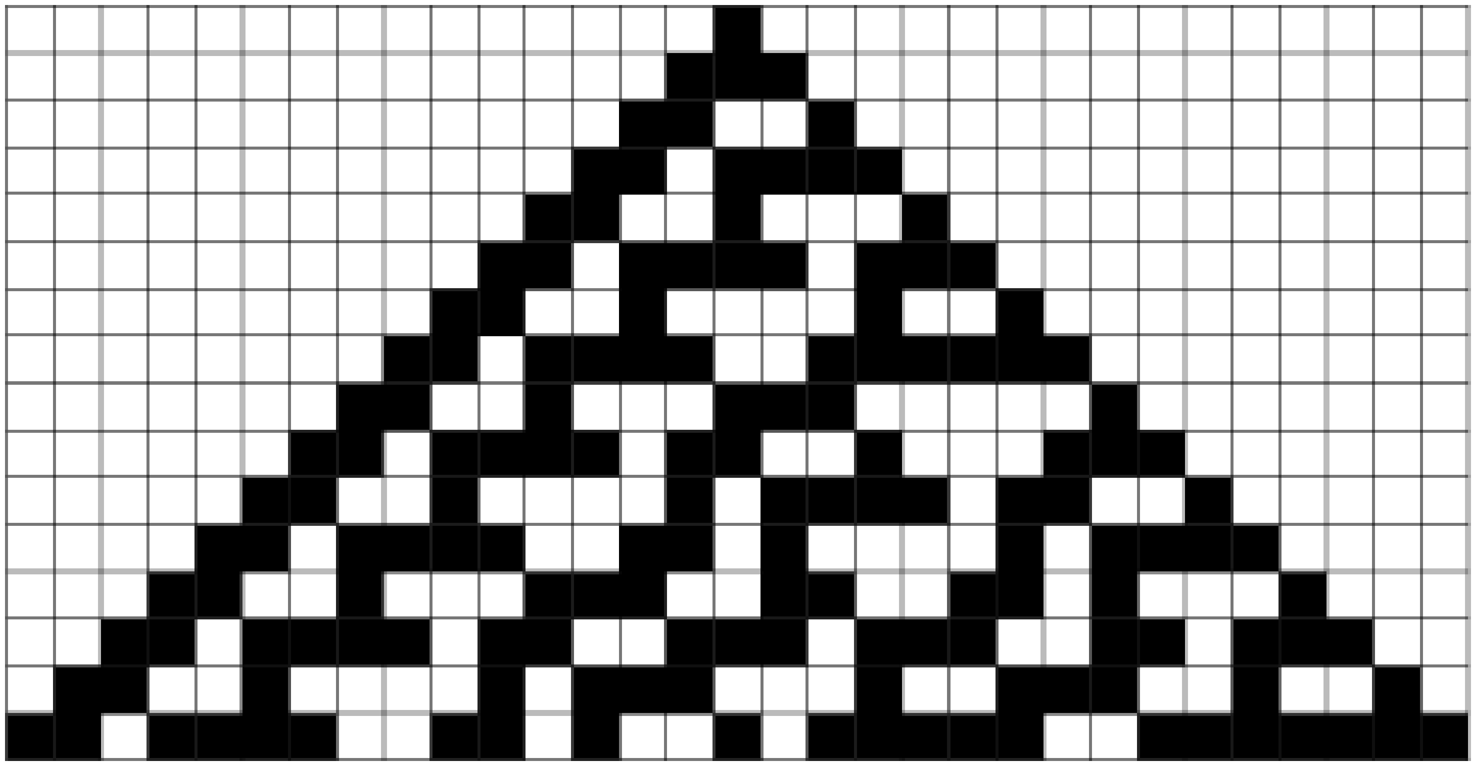
Sending data across cluster nodes

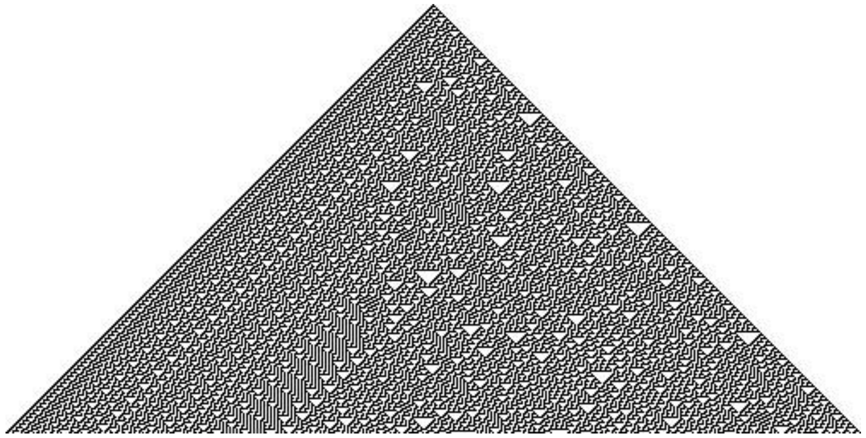
```
@everywhere using ParallelDataTransfer

sendto(workerid, vara = vara)

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

Cellular automaton

rule 30



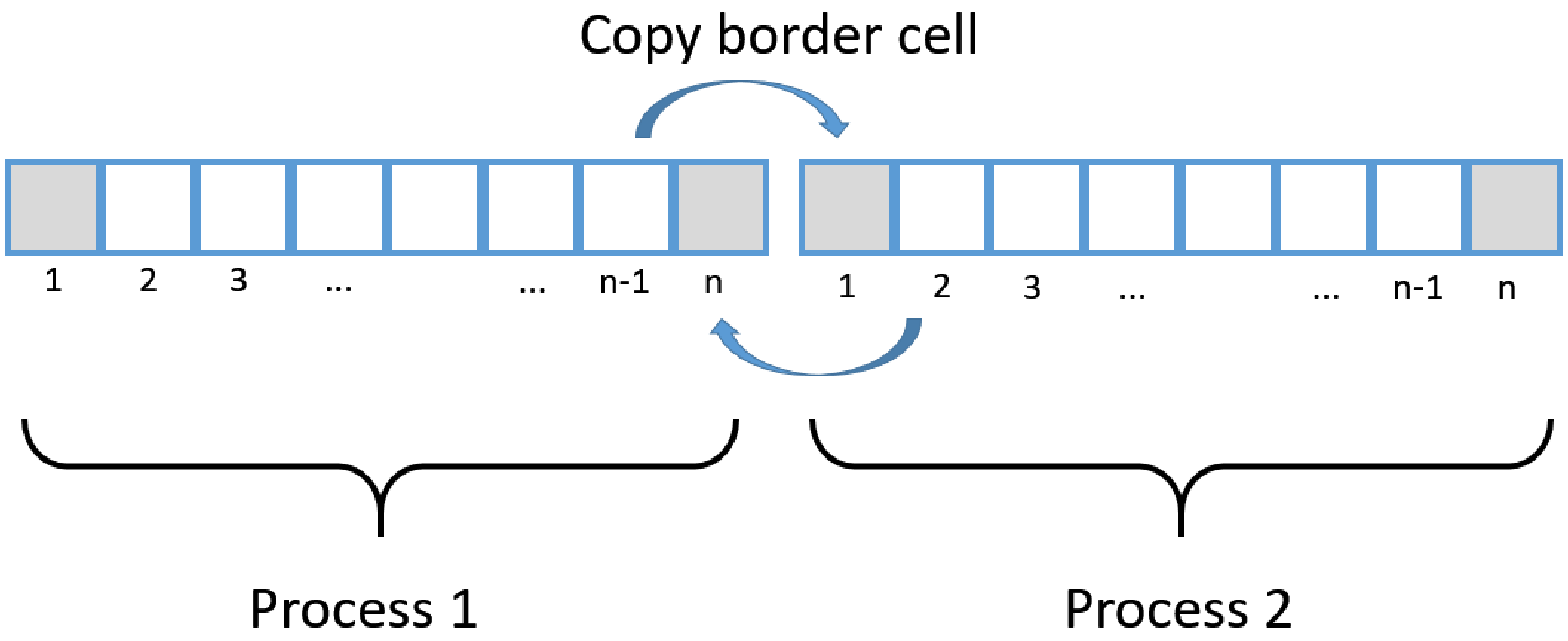




Source:
http://mathworld.wolfram.com/Rule30.html

Distributed cellular automaton

- Distributing data among worker processes



6_cellular_automaton.jl

```
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(ca::Array{Bool},
                                  neighbours::Tuple{Int64,Int64})
    ca[1] = (@fetchfrom neighbours[1] caa[end-1])
    ca[end] = (@fetchfrom neighbours[2] caa[2])
end
```

```julia
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 6_cellular_automaton.jl

```
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)
```

Typical approach for distributed processing

- Define a “reasonably large” work package
  - In our case one job is 1,000 sudoku problems (~10 seconds)
  - We have 100 such jobs (~20 mins on a single core)
- Julia distributed cluster manager

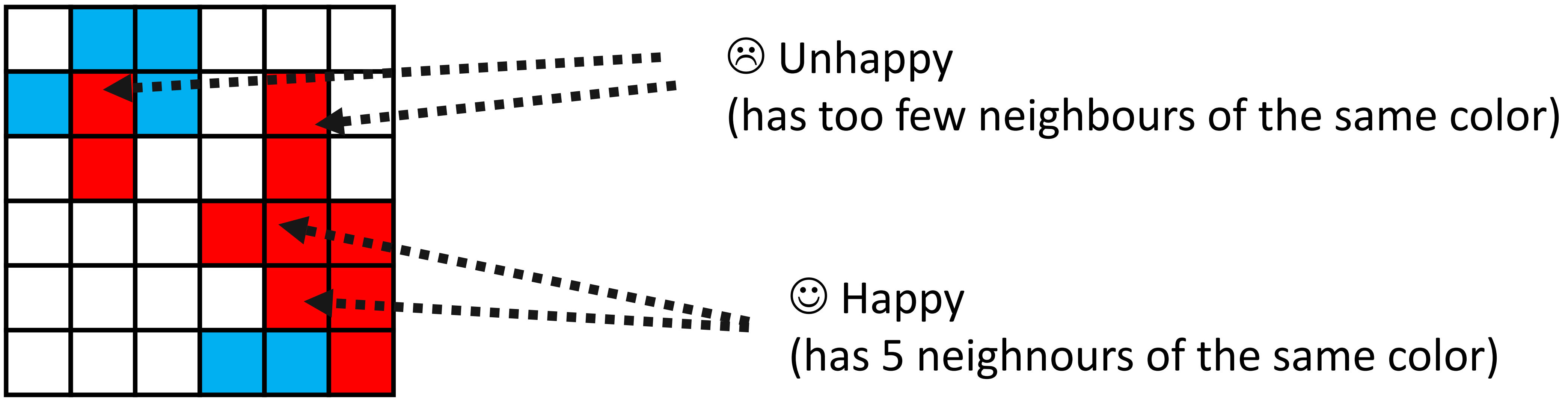
Julia cluster specification file and running distributed clusters

```
$ 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 --machinefile machinefile_julia program.jl

# REQUIRES PASSWORDLESS SSH TO BE CONFIGURED!
```

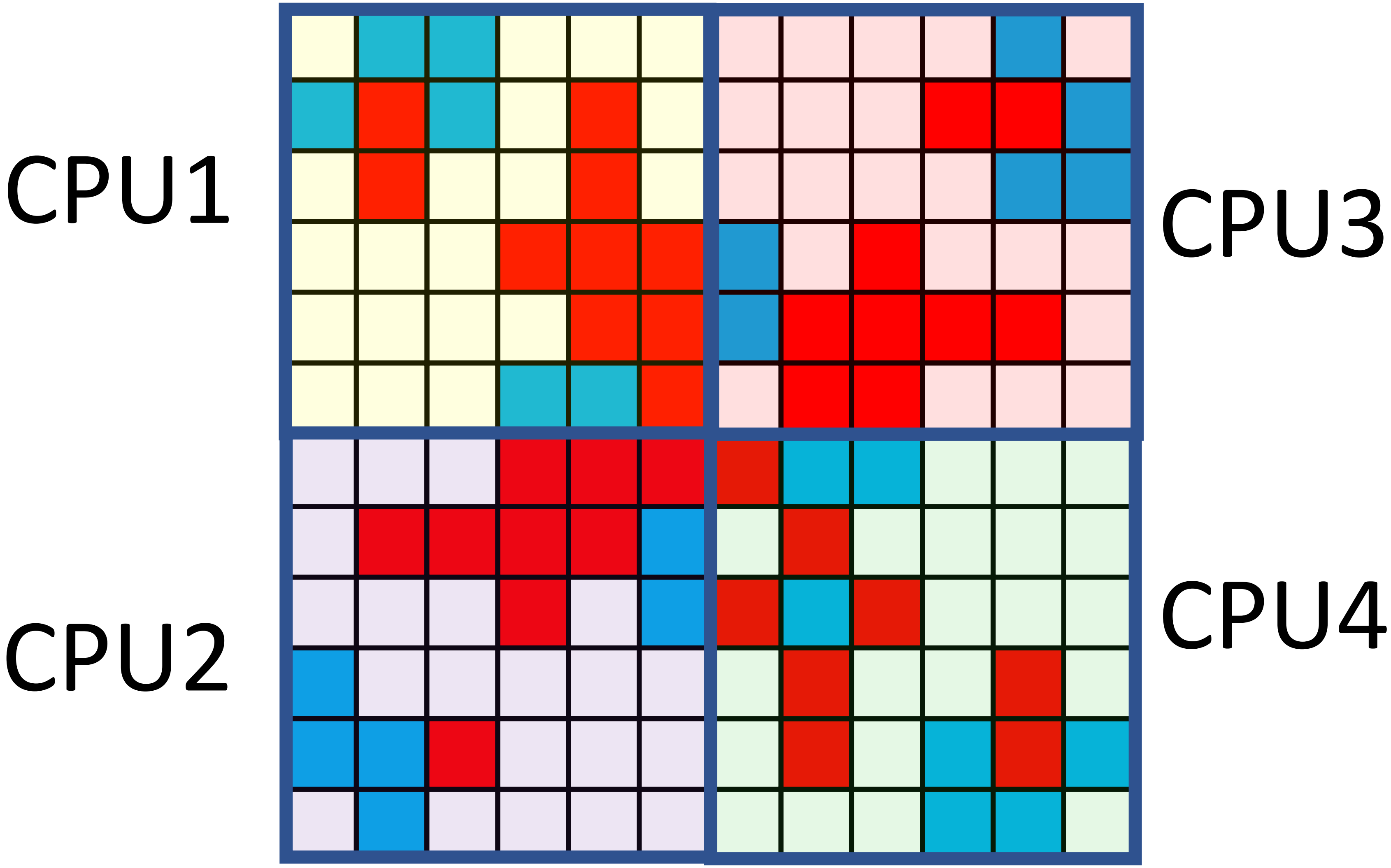
Use case scenario I: Performance of distributed code in Julia
Cray vs AWS

Schelling (1974) segregation model

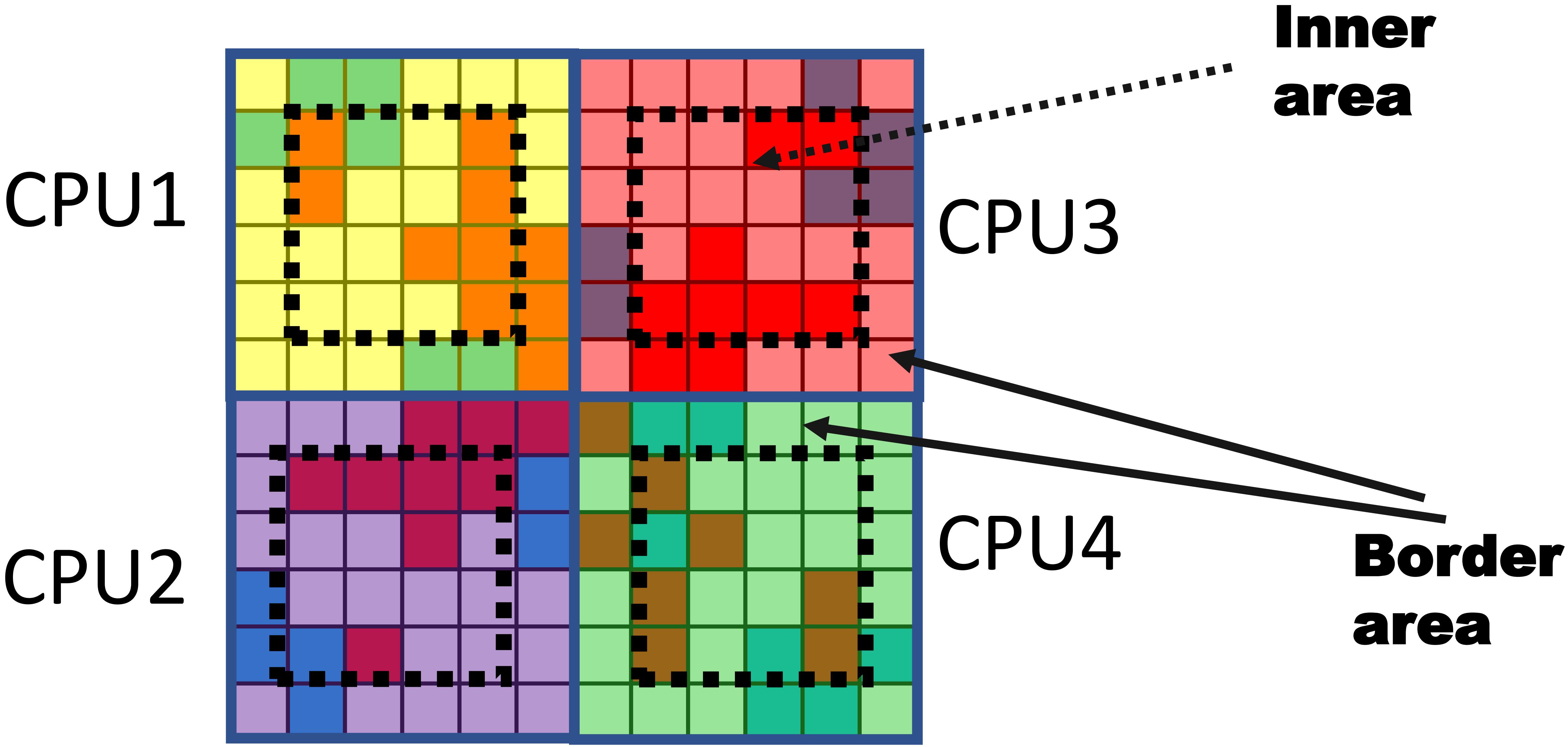
- Agents occupy cells of rectangular space
- Two types of agents (e.g. blue and red)
- When not happy with their neighbours randomly relocate



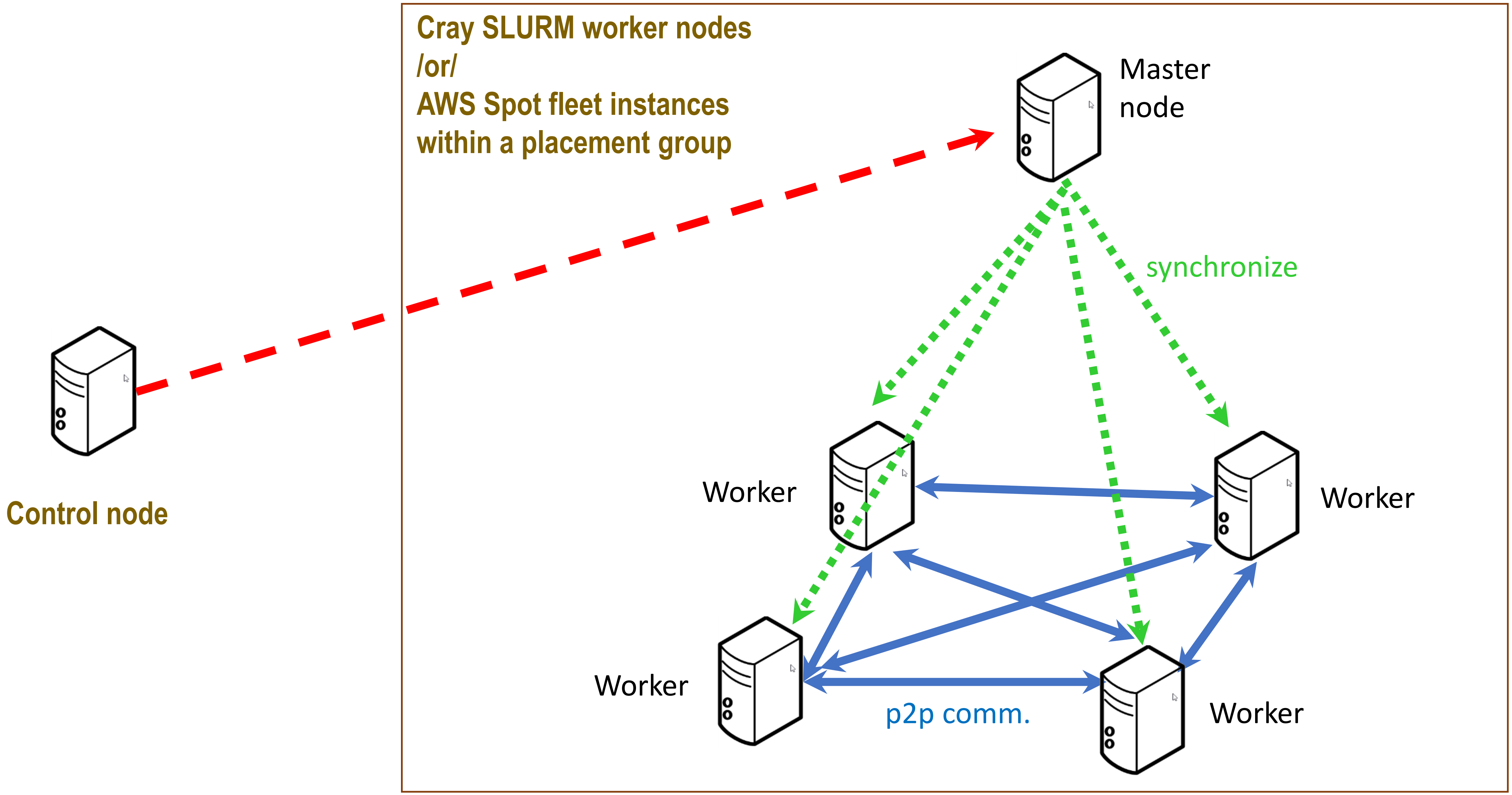
Distributed Schelling segregation model



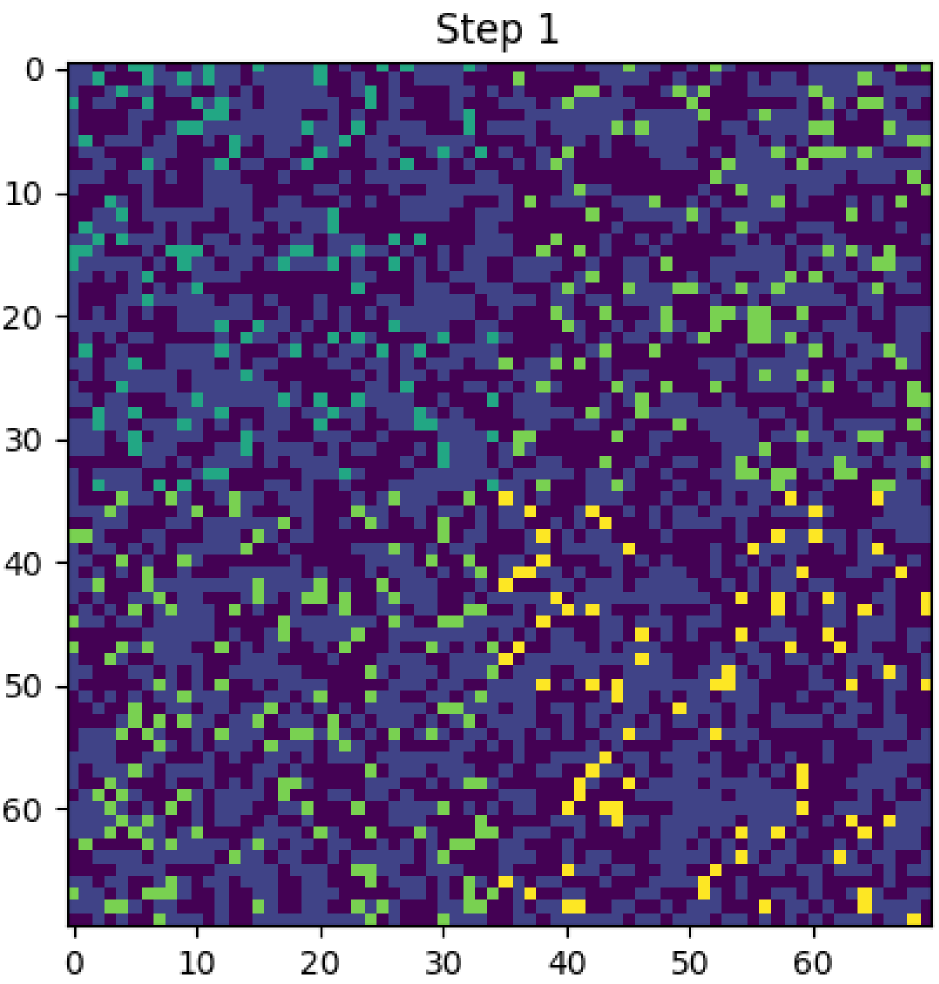
Distributed Schelling segregation model

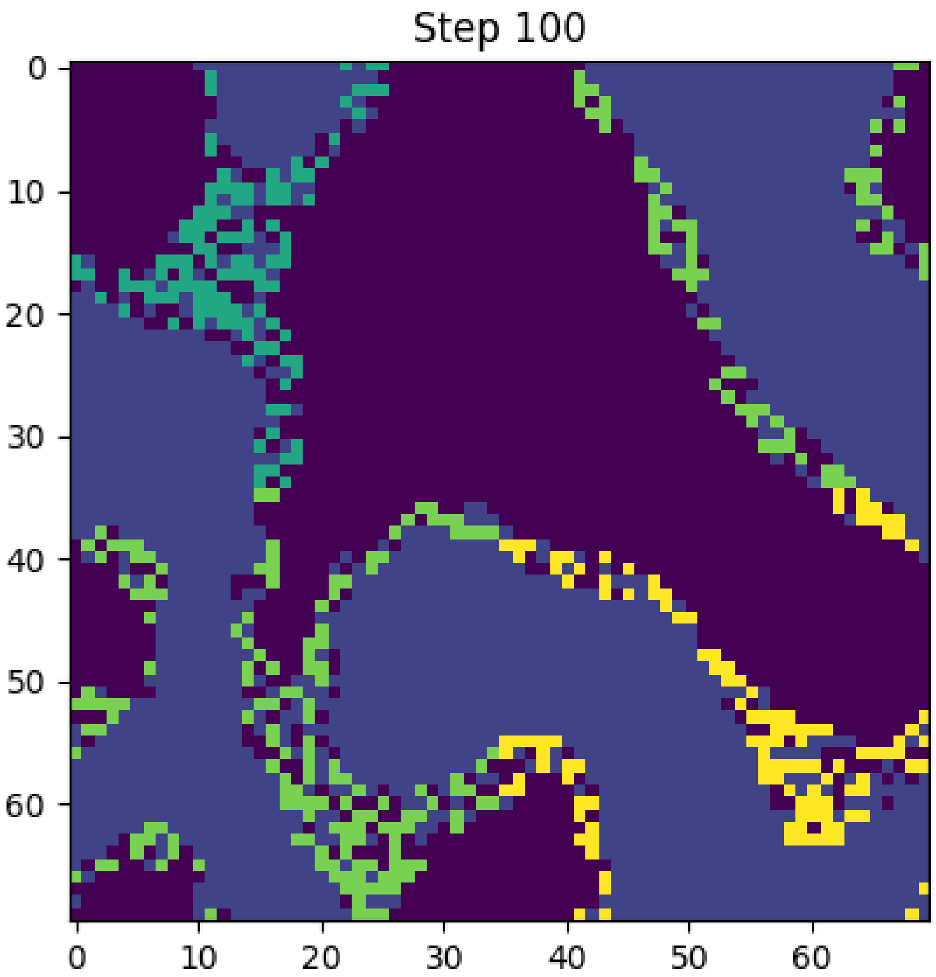


Distributed simulation architecture



Parallelized Schelling model (2x2)



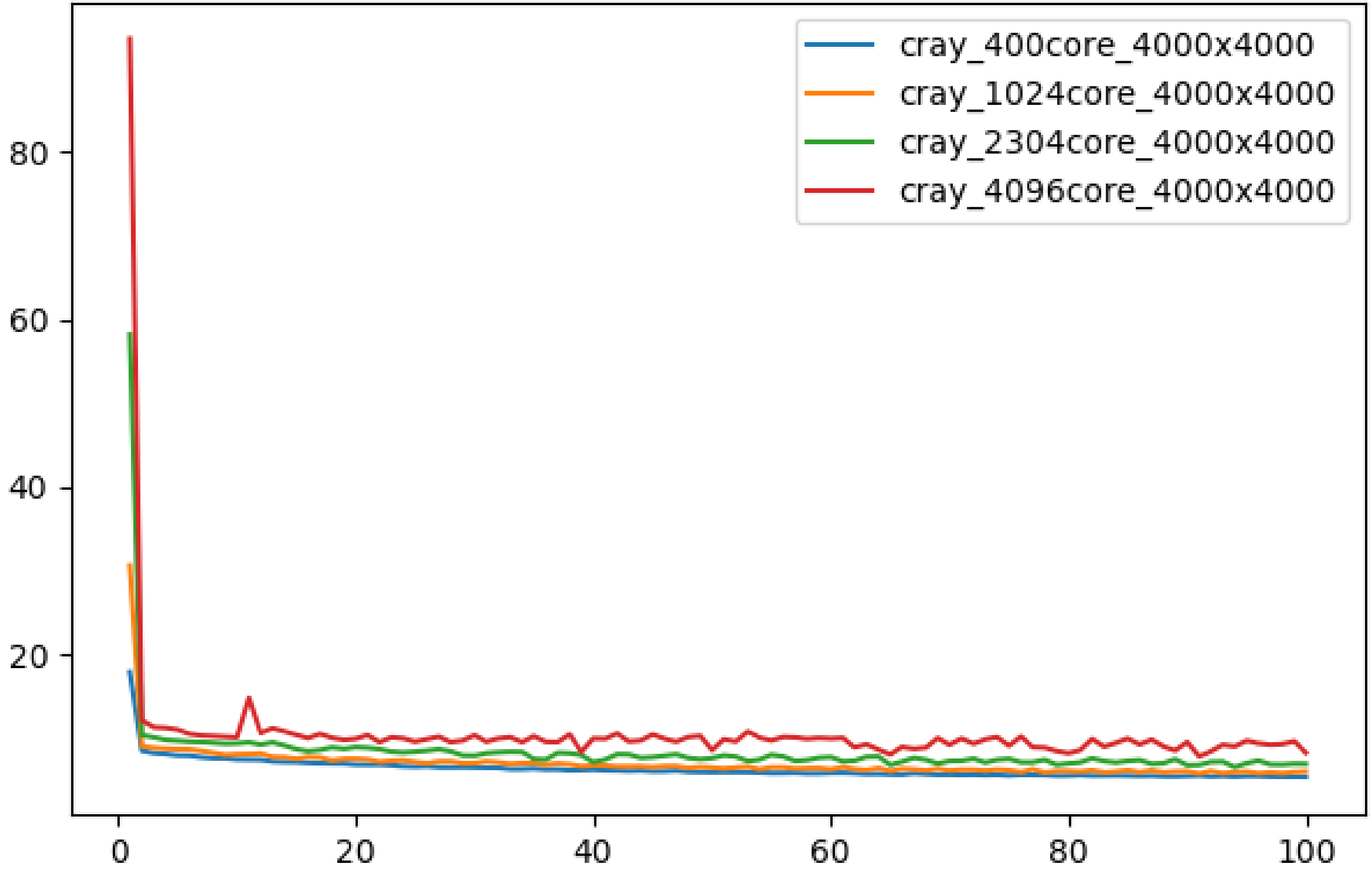


Parralelizing Julia on Cray with SLURM

```
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 ….

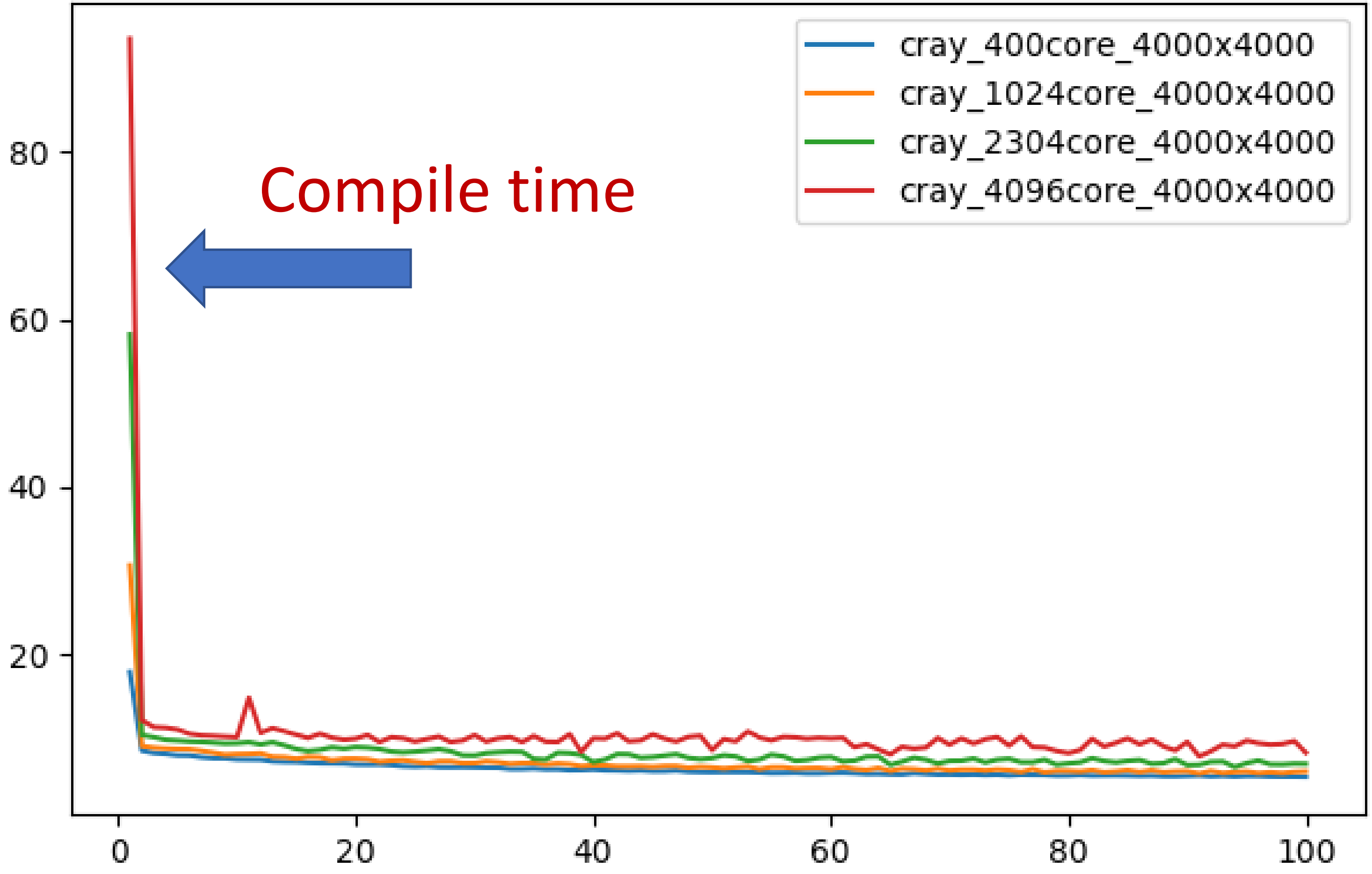
Time to execute step (seconds)



Simulation step

Typical Julia performance pattern ….

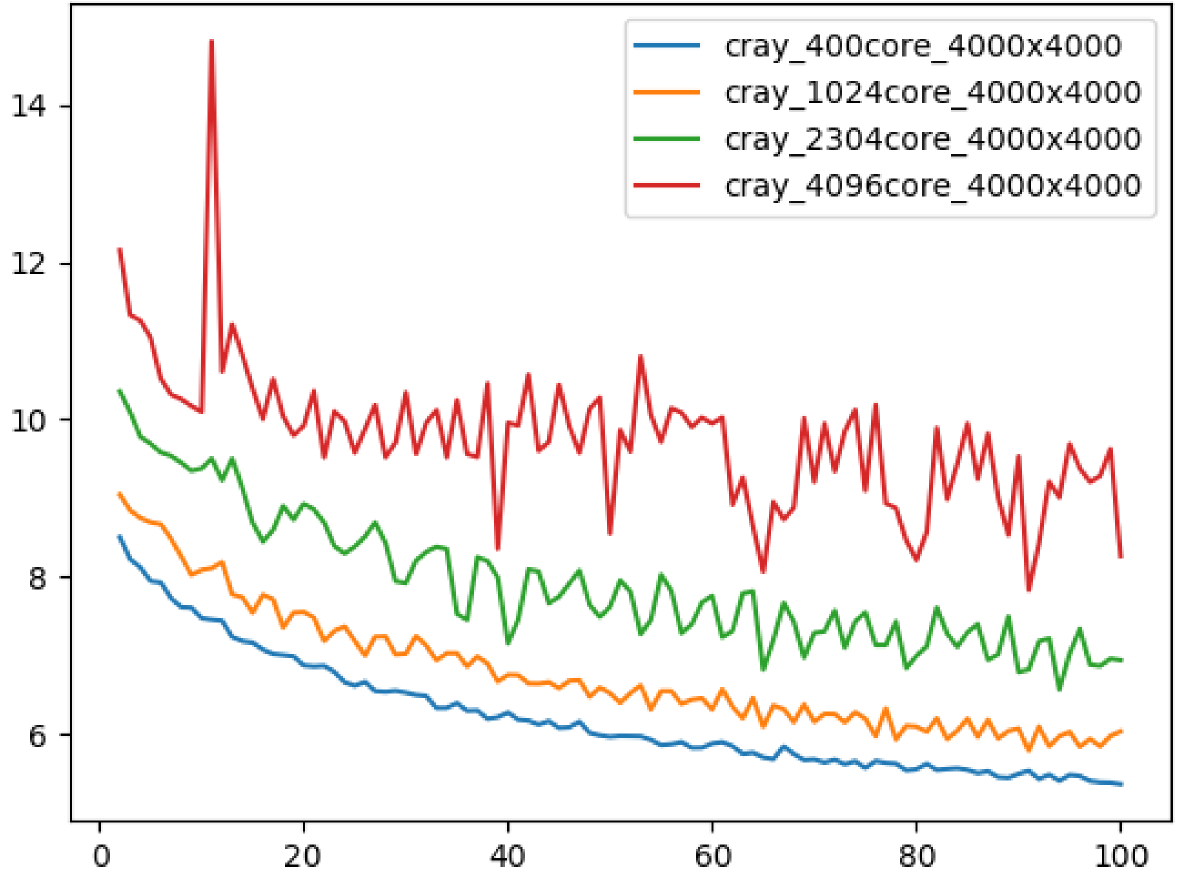
Time to execute step (seconds)



Simulation step

Distributed simulation scalability

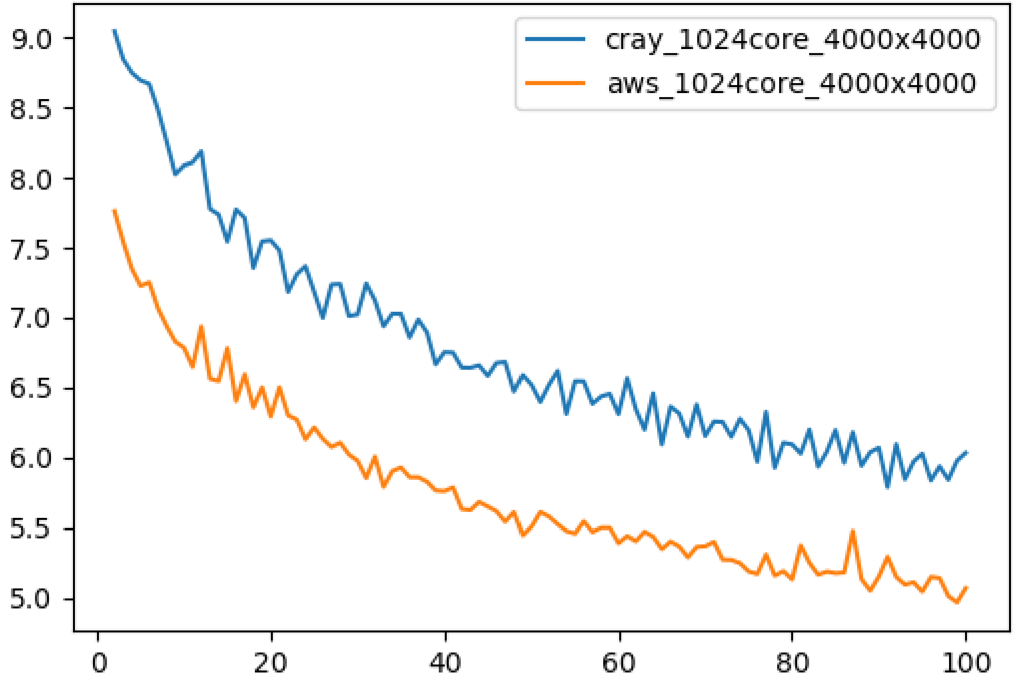
Time to execute step (seconds)

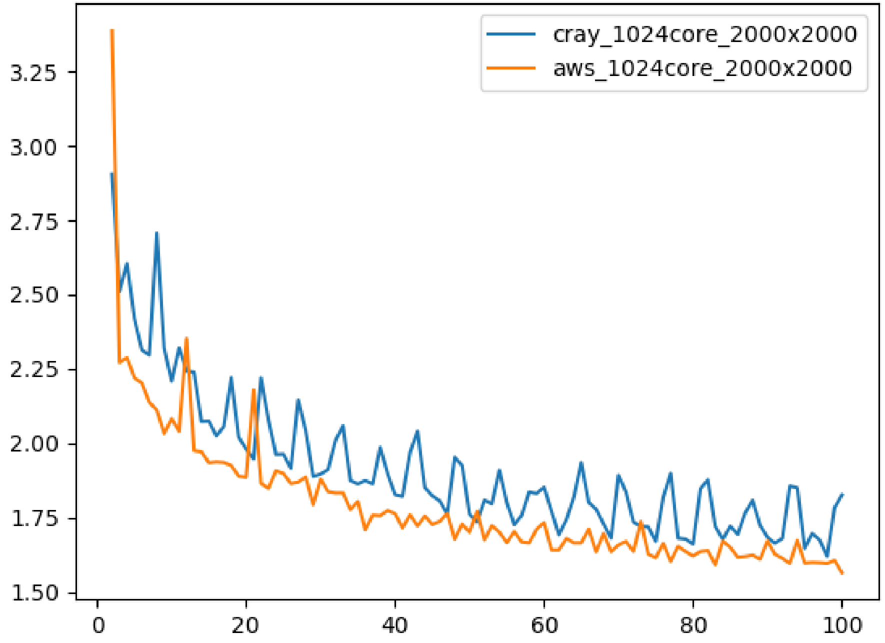


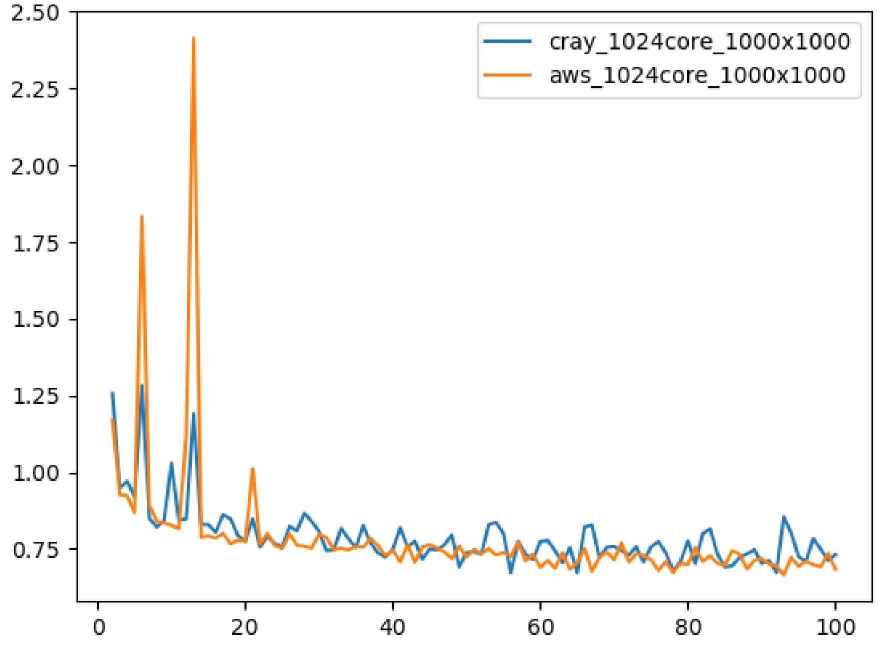
Simulation step

Note:
The first step has been removed

Cray vs AWS Spot Fleet







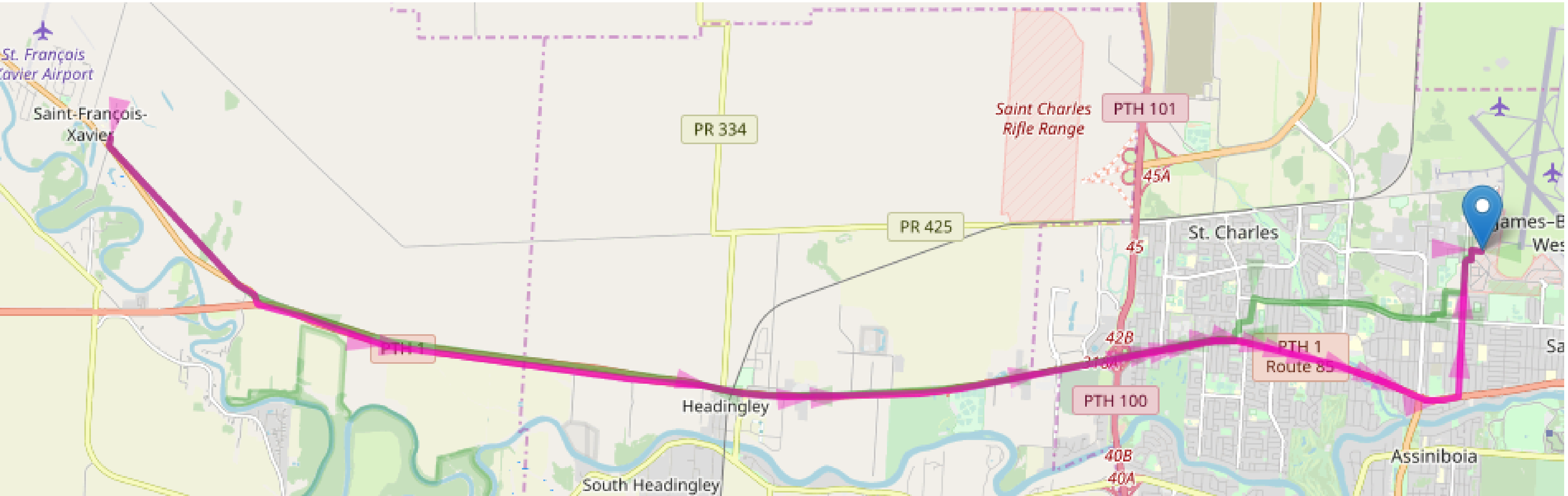
Use Case Scenario II
Agent-based simulation of cities
https://github.com/pszufe/OpenStreetMapX.jl

Project goals....

- A multi-agent simulation framework (Open Source) for modelling commuters' behavior within a city
  - individual travel patterns
  - sociodemographic profiles of commuters
  - exchange of information between travelling agents

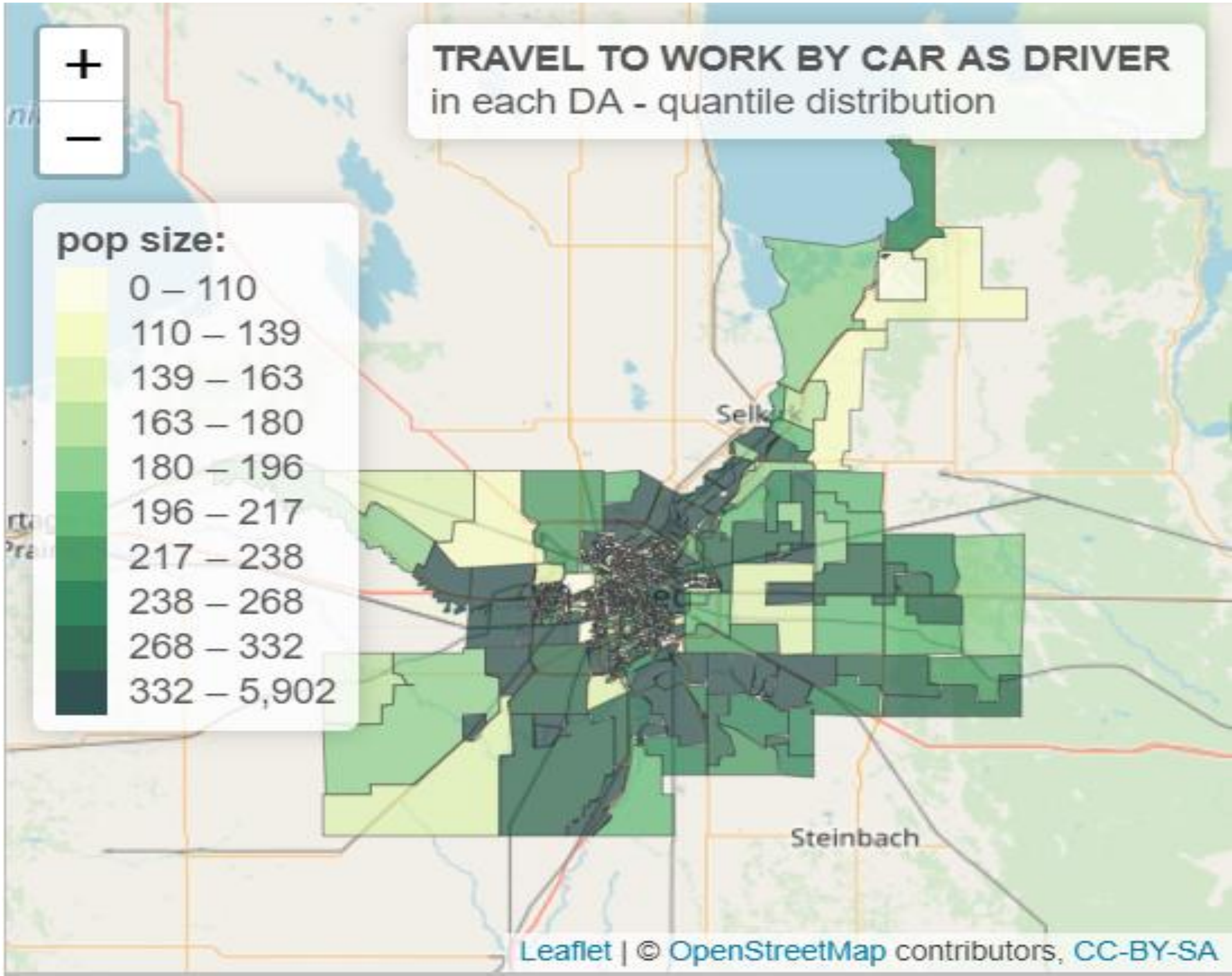
wykres/zdjęcie

- The simulation results have been validated against real traffic data

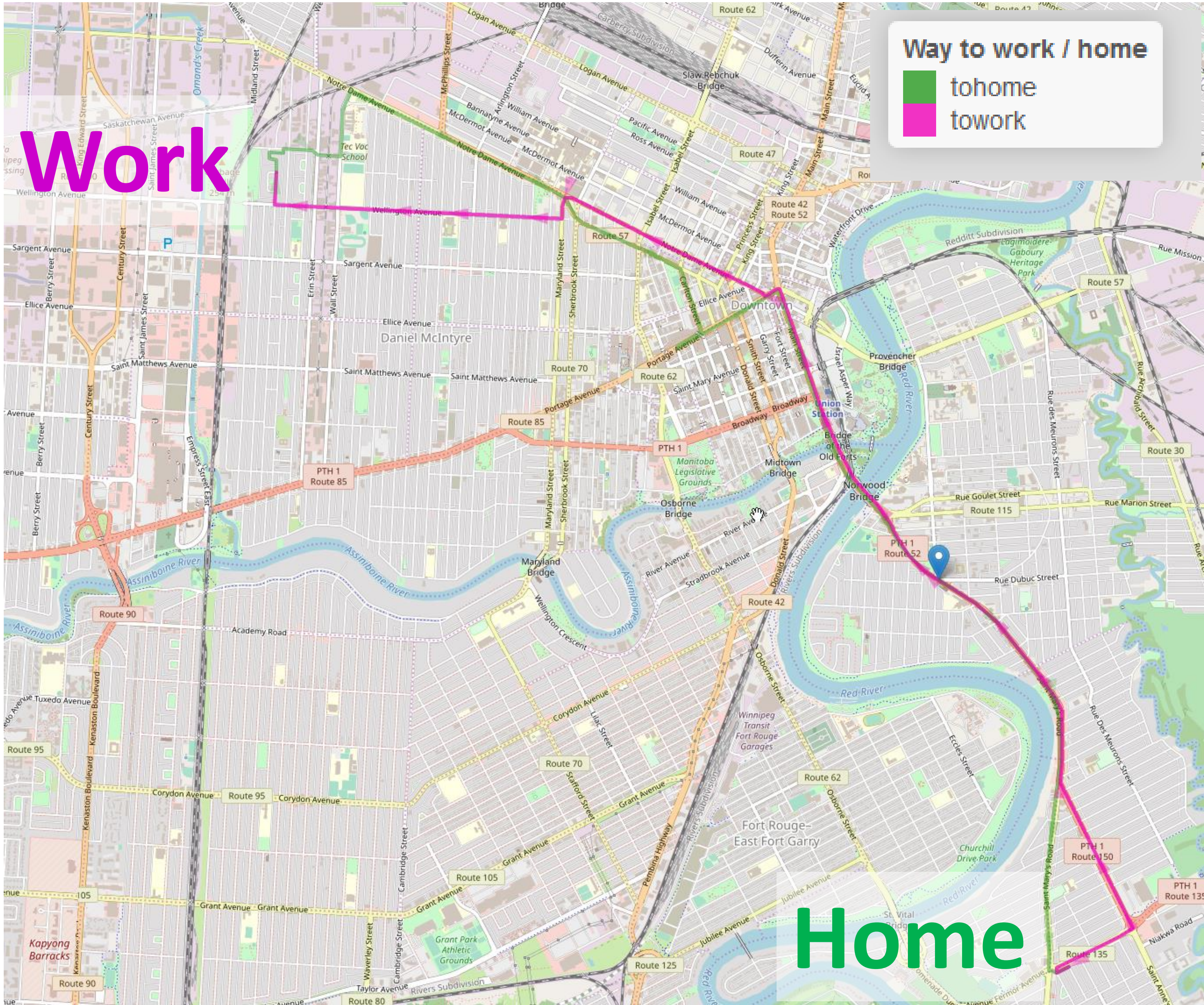


The modelled city – available data

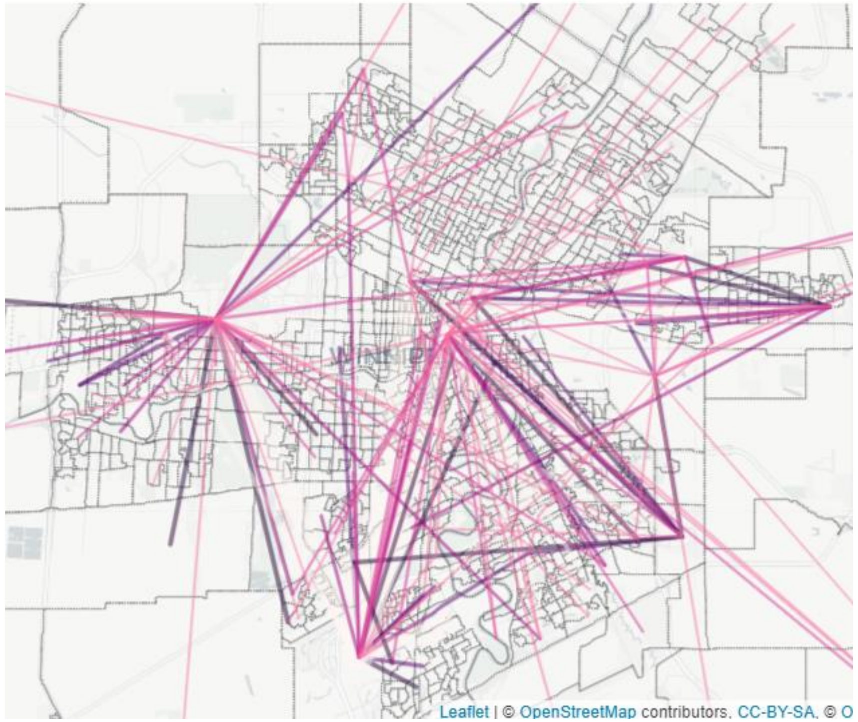
- Winnipeg CMA, Canada
- population ~1’000’000
- 1’200 dissemination areas

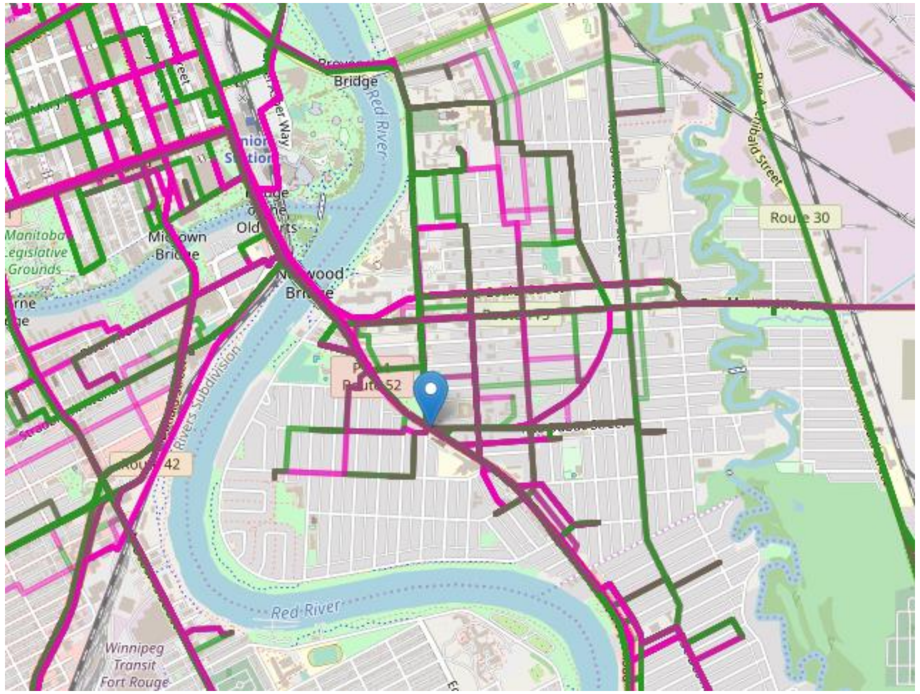


A single simulated commuter's behavior...



Simulation model travel destinations and commuters’ behavior

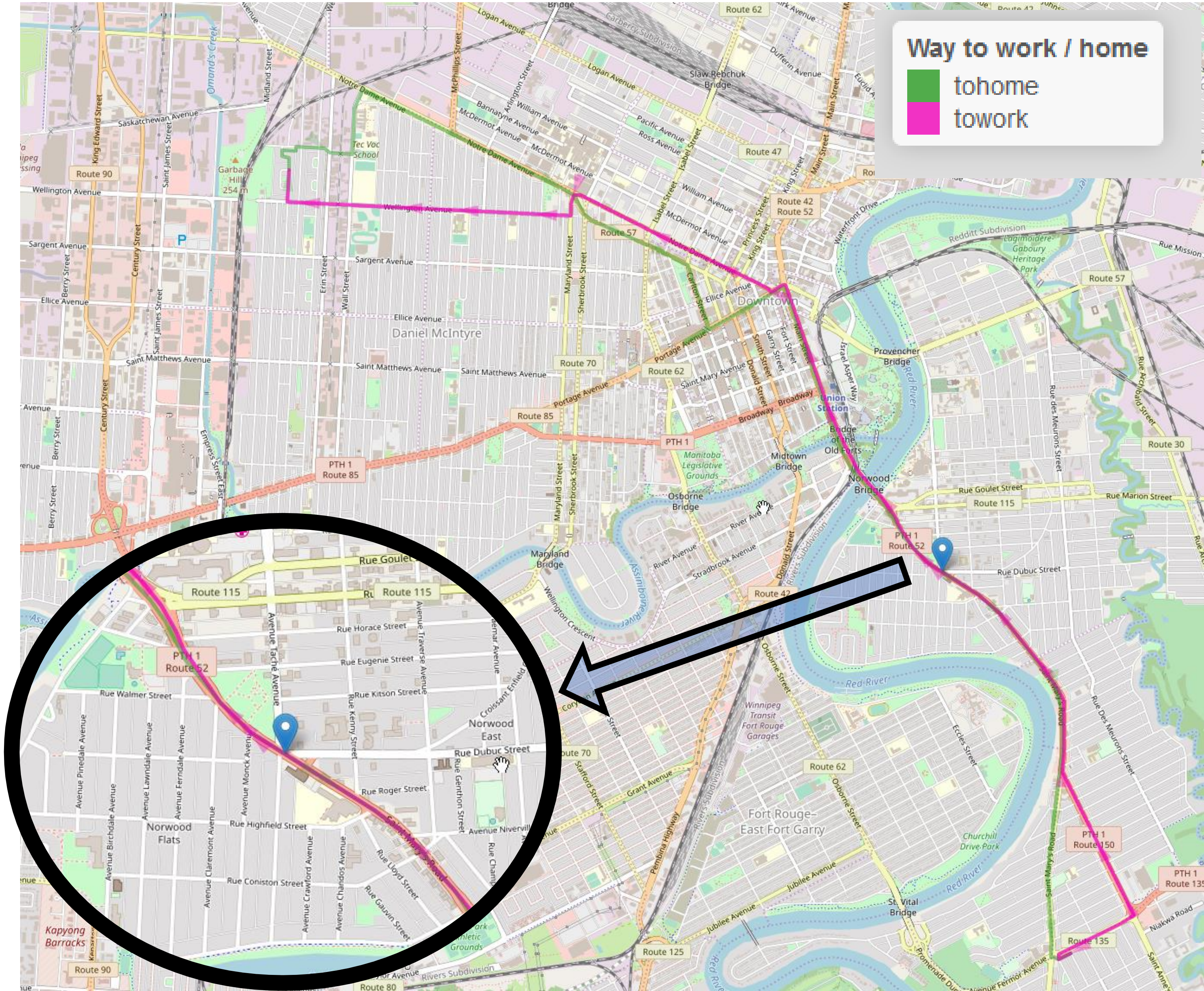


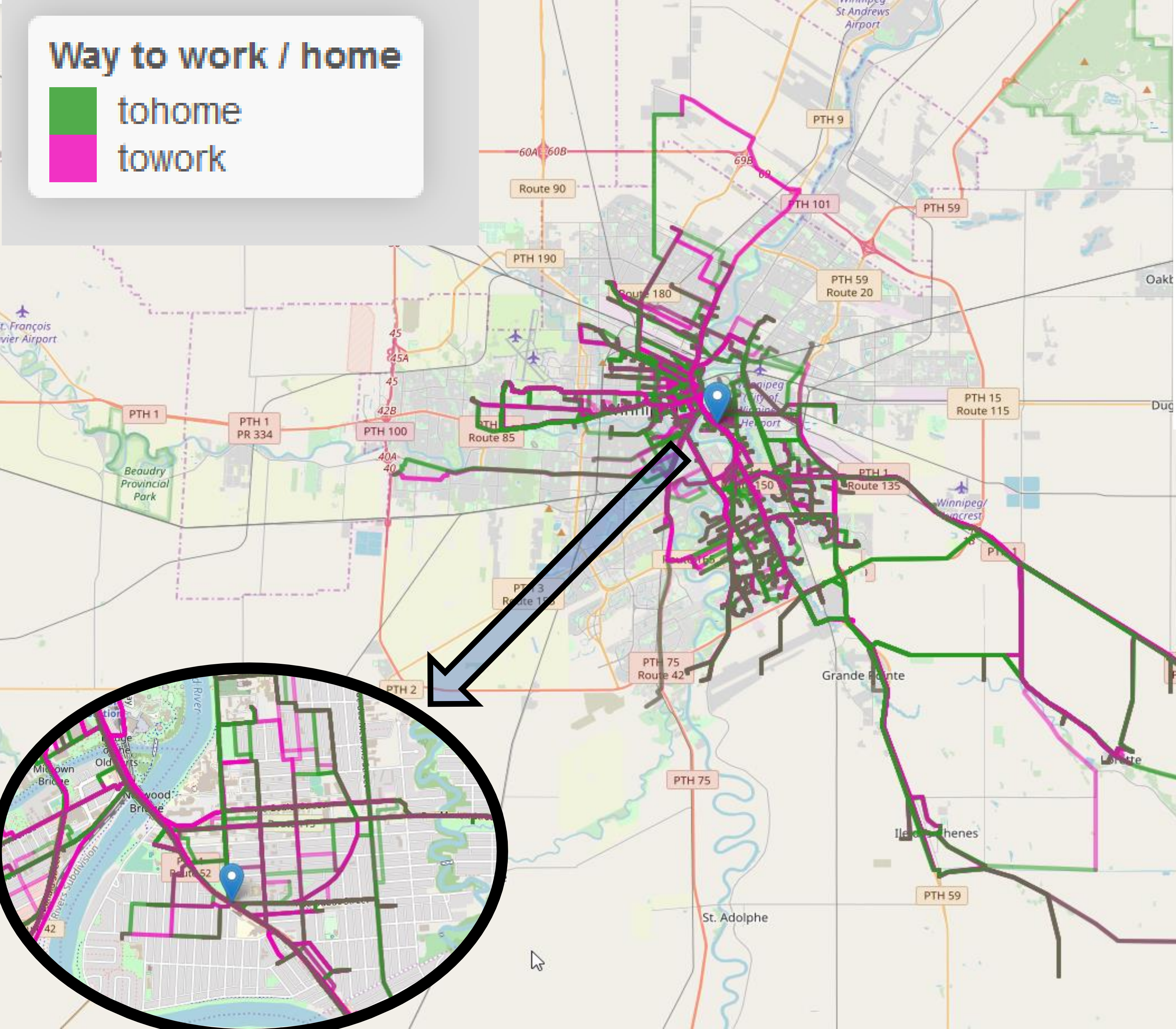


Each agent makes individual travel decisions on the base of her sociodemographic profile

Questions

- How many people went to the given road crossing
- Who are those people? Reach? Poor? Immigrants?
- If a place an advertising billboard – who is going to see it?
- What business makes sense in this area





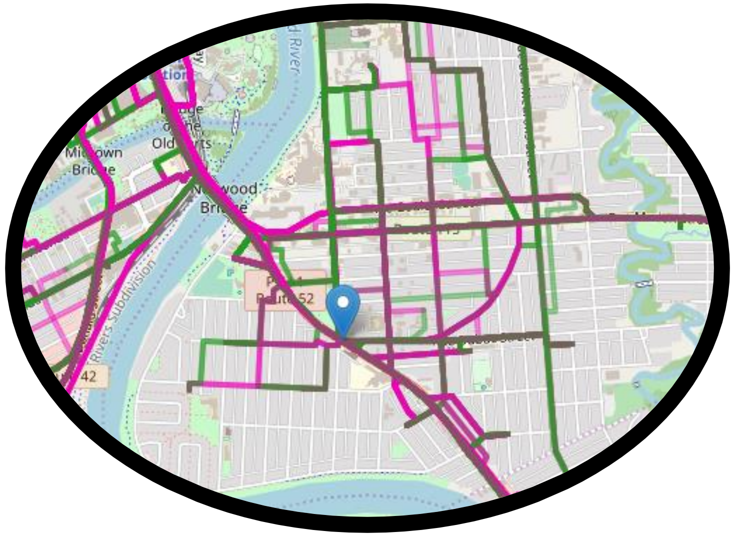
Simulated journey of virtual 1000 cars across the city

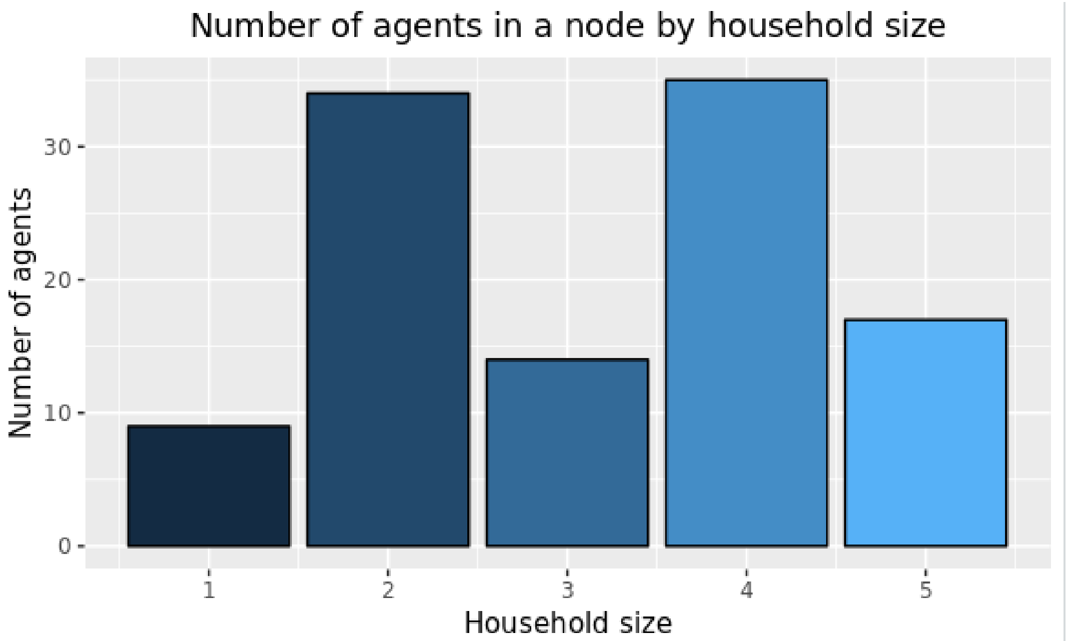
(only the cars that went through the marked crossing have been selected)

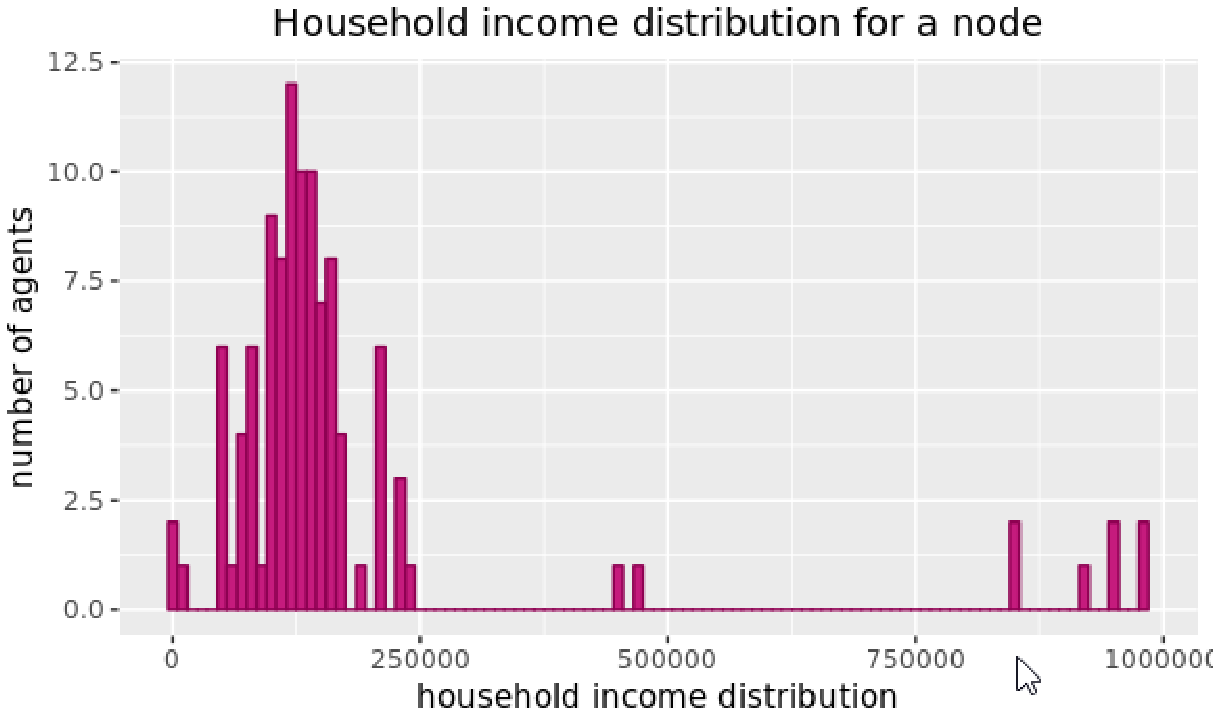
Sample simulation data

NODE_ID	longitude	latitude	DA_home	DA_work	gender	age	marital_st	work_indu	household	househol	no_of_chi	children_a	imigrant	imigrant_s	imigrant_r	id
369306773	-97.1122	49.90696	46110714	46110684	M	69	true	Retail Trac	89138	2	0	Int64[]	true	Before 20	Eastern Eu	2001000028
369306773	-97.1122	49.90696	46110230	46111151	M	28	false	Wholesale	65454	1	0	Int64[]	true	2012 To Pr	Central Ar	2001000030
369306773	-97.1122	49.90696	46110803	46111177	F	22	false	Retail Trac	48281	2	1	[5]	true	2006 To 20	Northern	2001000035
369306773	-97.1122	49.90696	46110845	46110632	M	52	true	Manufactu	63822	1	0	Int64[]	false			2001000103
369306773	-97.1122	49.90696	46110795	46110162	F	36	true	Transport	32480	1	0	Int64[]	false			2001000289
369306773	-97.1122	49.90696	46110869	46110100	F	20	false	Finance A	82354	2	0	Int64[]	false			2001000318
369306773	-97.1122	49.90696	46110801	46110117	M	60	false	Arts, Ente	621011	3	0	Int64[]	false			2001000403
369306773	-97.1122	49.90696	46110701	46110669	M	41	false	Public Adr	99562	2	1	[18]	false			2001000525
369306773	-97.1122	49.90696	46110735	46110075	M	41	false	Retail Trac	909754	2	1	[14]	false			2001000529
369306773	-97.1122	49.90696	46110845	46110667	M	51	false	Retail Trac	95722	3	0	Int64[]	false			2001000736
369306773	-97.1122	49.90696	46120046	46110669	M	53	true	Health Cai	84334	3	0	Int64[]	false			2001000773
369306773	-97.1122	49.90696	46110802	46110144	M	44	true	Public Adr	42891	4	0	Int64[]	false			2001000948
369306773	-97.1122	49.90696	46111151	46110145	M	37	true	Finance A	118452	1	0	Int64[]	true	2012 To Pr	Southeast	2002000044
369306773	-97.1122	49.90696	46110851	46111177	M	55	false	Health Cai	62681	3	0	Int64[]	false			2002000060
369306773	-97.1122	49.90696	46110453	46110683	F	34	true	Real Estat	29972	5	0	Int64[]	false			2002000172
369306773	-97.1122	49.90696	46110875	46110145	F	32	true	Retail Trac	50486	3	1	[46]	false			2002000265
369306773	-97.1122	49.90696	46110853	46110631	F	25	false	Public Adr	52526	1	0	Int64[]	false			2002000318
369306773	-97.1122	49.90696	46120053	46110669	F	38	true	Manufactu	119023	2	0	Int64[]	false			2002000336

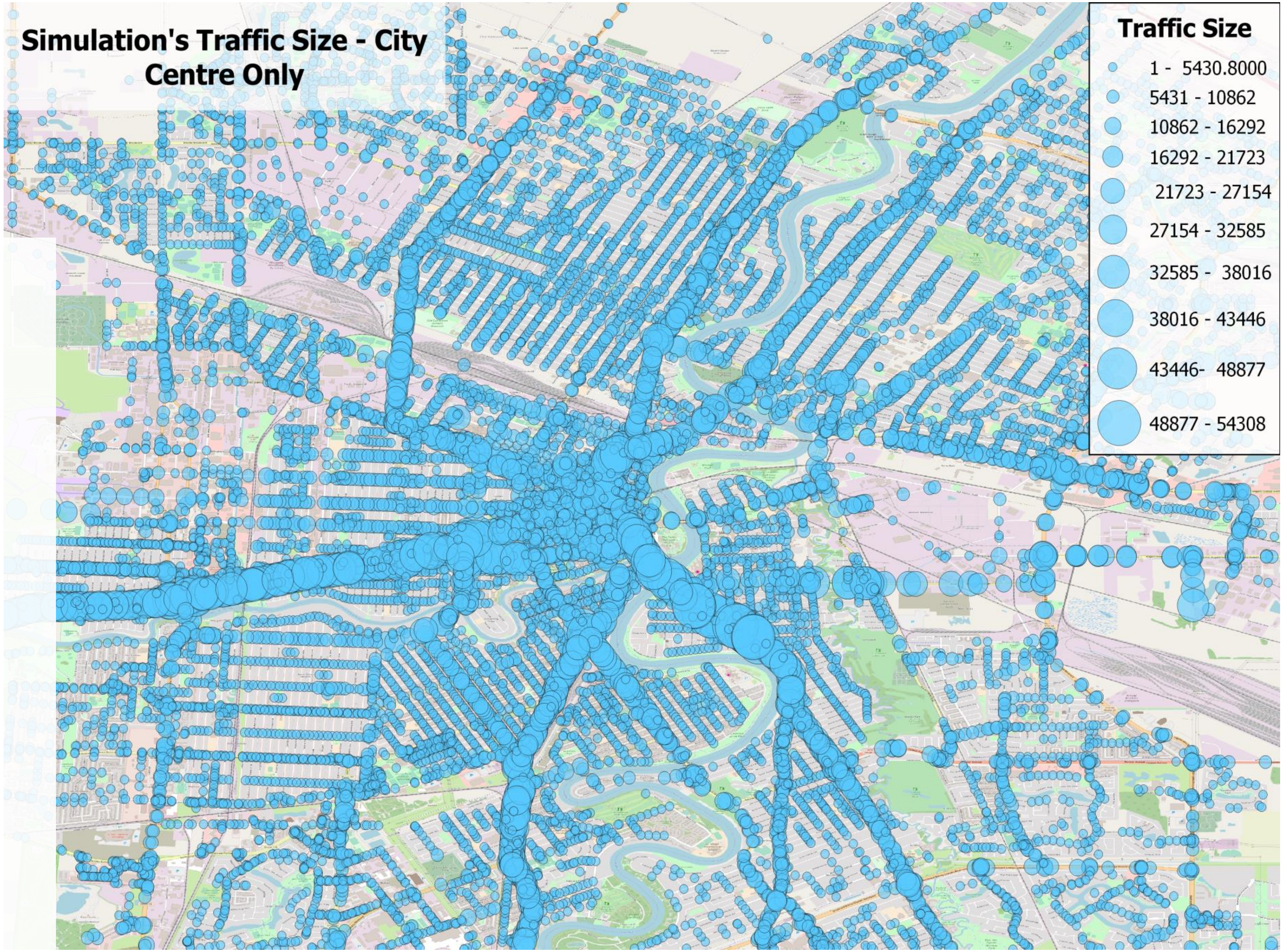
Outcome: Sociodemographic profiles available for each intersection within the city



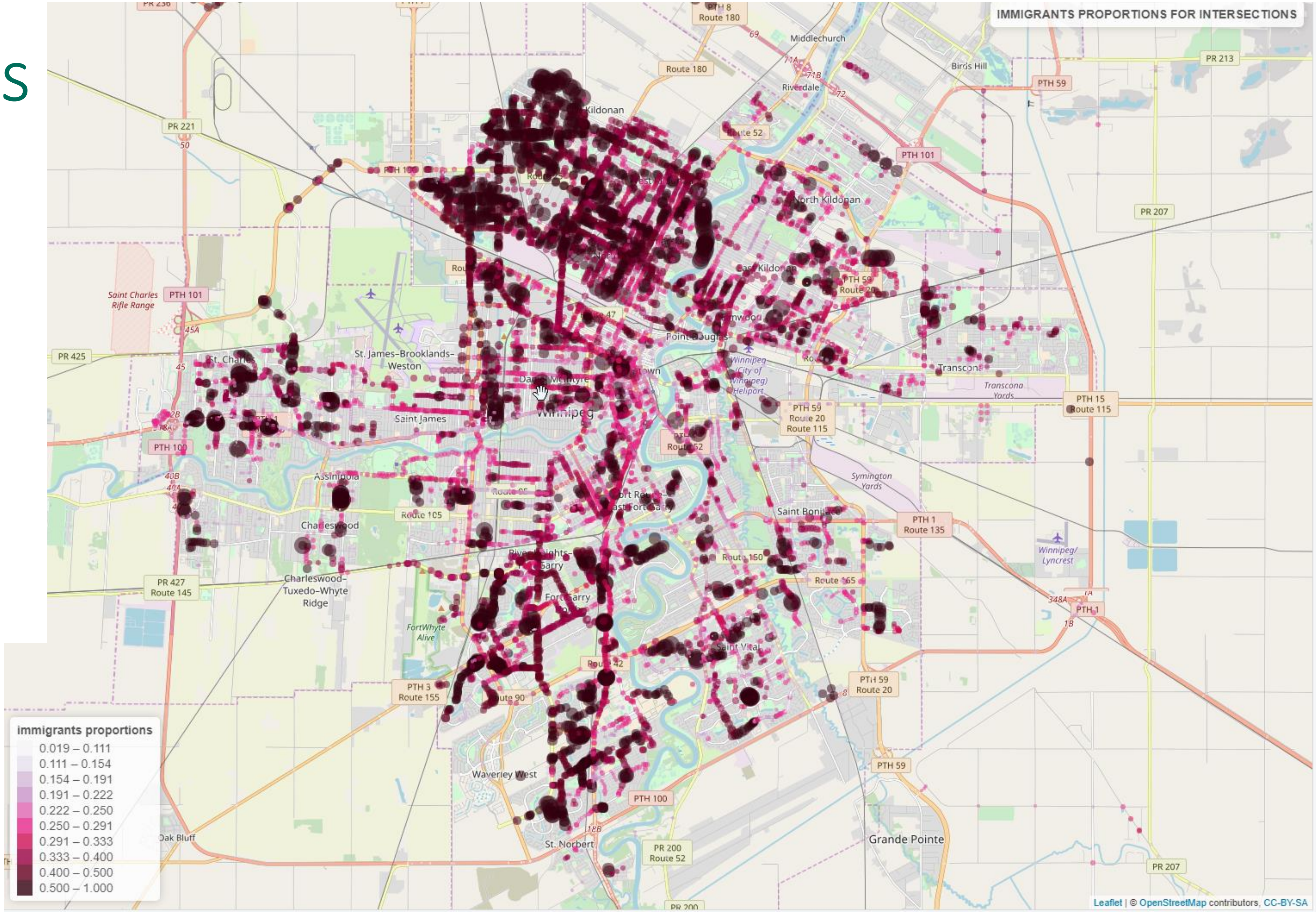




Simulated aggregated traffic on street crossing in the city center



Percentage of immigrants passing through each intersection within the city



Conclusions

- Julia
- PROs
  - distributed computing
  - high performance computing
  - scientific computing
- CONs
  - data visualization layer still under development