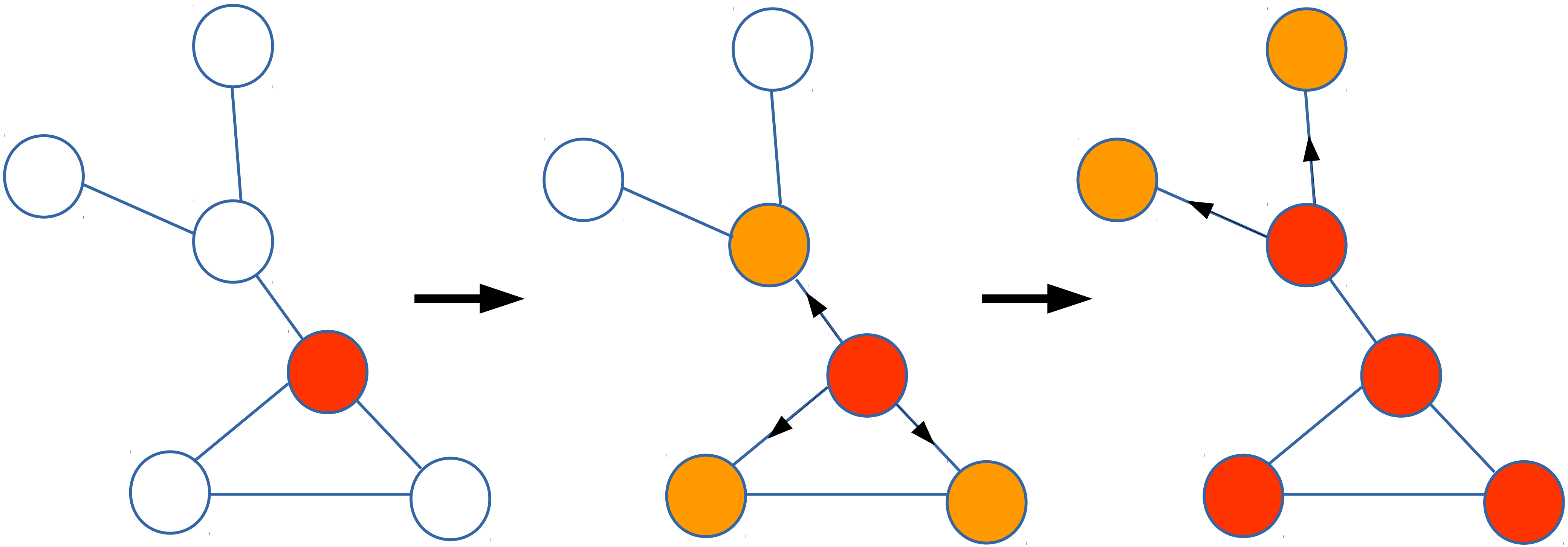
Parallel Graph Algorithms in Julia

MIT 6.338

Julian Kates-Harbeck



Outline

1) Introduction

2) Coarse grained parallelization: **mulitprocessing**
(shared + distributed memory)

3) Fine grained parallelization: **multithreading**
(shared memory)

Speedup Progression

- **Rewrite in Julia** (Graphs.jl) of Python algorithm (Networkx)
  - ➔ **~ 5-10x**
- **Serial optimization**, including LightGraphs.jl
  - ➔ **~ 5-10x**
- **Parallelism** (Focus of this talk!)
  - ➔ **> 100x**
- Total: ~ **3-4 orders of magnitude!**

Introduction

- **Graph algorithms** and **Monte Carlo** (MC) methods are very common
- Our problem
  - Many **independent** Monte Carlo iterations
  - Each one is a (complex) graph algorithm
    - Think something like PageRank

```
results = map(run_graph_simulation,1:num_trials)

#analyze results...
```

Two types of parallelism

```
results = map(run_graph_simulation,1:num_trials)

#analyze results...
```

Coarse Grained Parallelism

Fine Grained Parallelization Needed

Coarse Grained Parallelism

In a perfect world, `map` → `pmap`

```
#results = map(run_graph_simulation,1:num_trials)
results = pmap(run_graph_simulation,1:num_trials)

#analyze results...
```

But, we need to manage the processes!

```
addprocs(N_PROCS)
```

- How many processes to add?
- How many cores are available?
- What if the cores are on different machines?

Automatic Multiprocess Management

- Ideally

```
addprocs(N_PROCS)
```

  just works for any number of processes.
- Under the hood
  - X_i cores per machine i, Y machines
  - On a shared cluster, X and Y might differ for each allocation!
    - ➔ **Don't want to hardcode!**

Automatic Multiprocess Management

- **Use case**: SLURM (Simple Linux Utility for Resource Management) on Harvard's Odyssey Cluster
  - One allocation gives variable number of machines.
  - Variable number of cores per machine.
- **Solution**: Fill up cores on each machine with one processes each, up to N:

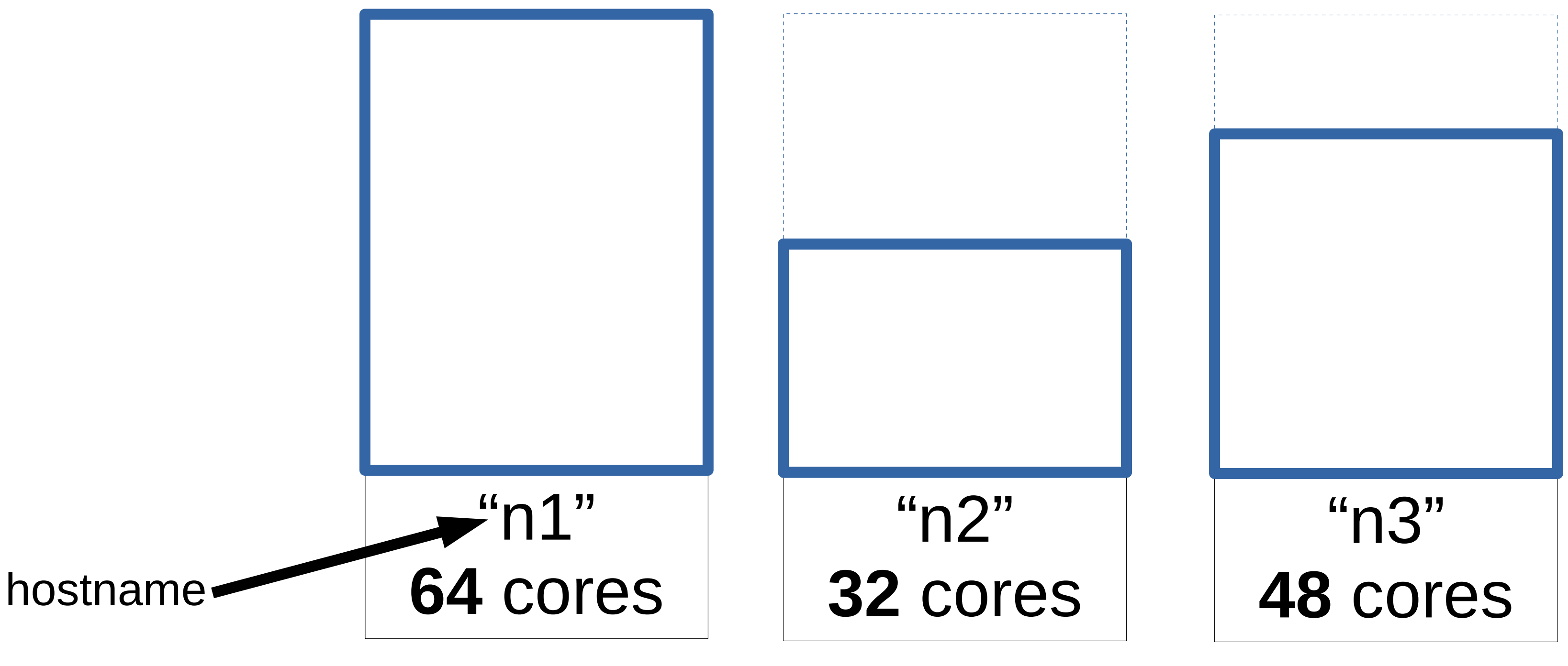
```
nl = get_partial_list_of_nodes(N)
addprocs(nl)
```

Behind the scenes: *Environment Variables*

(SLURM_NODELIST, SLURM_JOB_CPUS_PER_NODE)

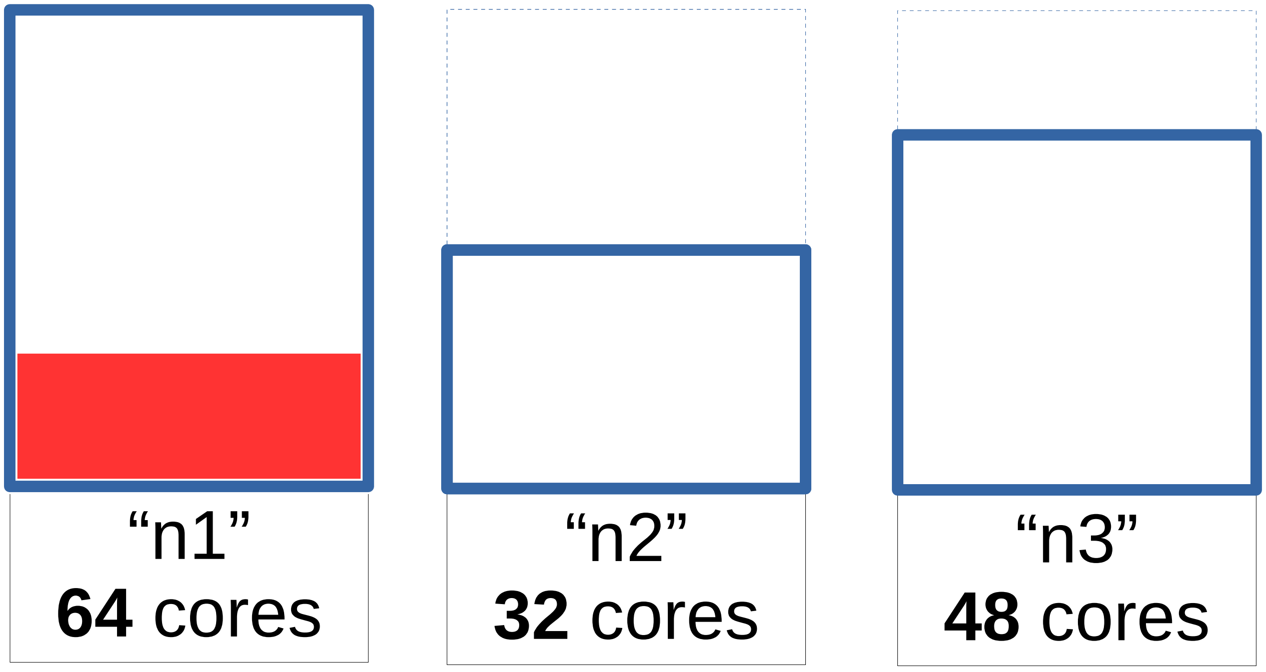
Allocation Example

Resource Allocator gives me:



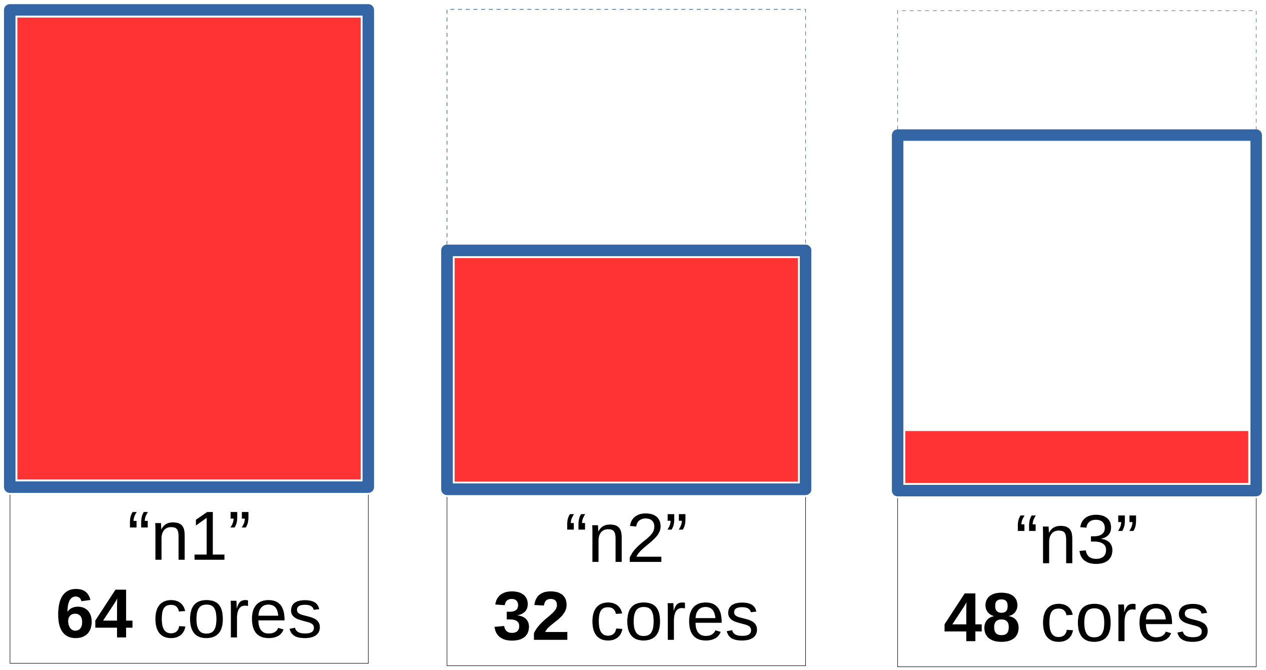
Allocation Example

```
nl = get_partial_list_of_nodes(20)
addprocs(nl)
```



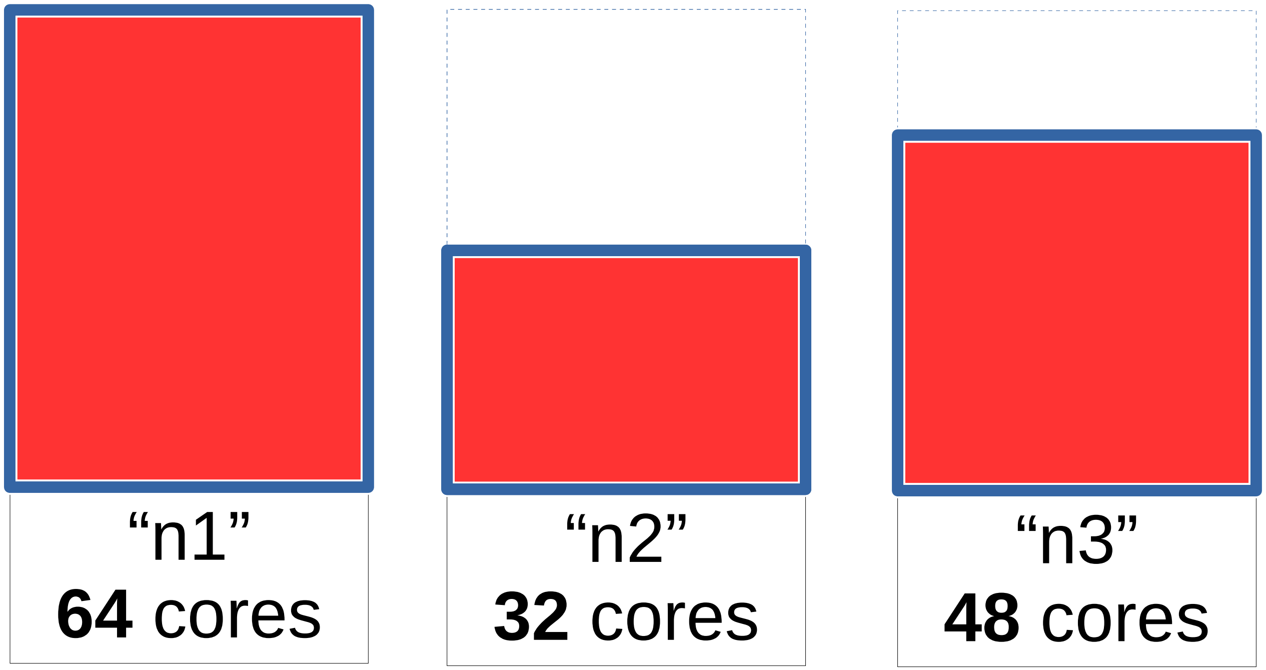
Allocation Example

```
nl = get_partial_list_of_nodes(100)
addprocs(nl)
```



Allocation Example

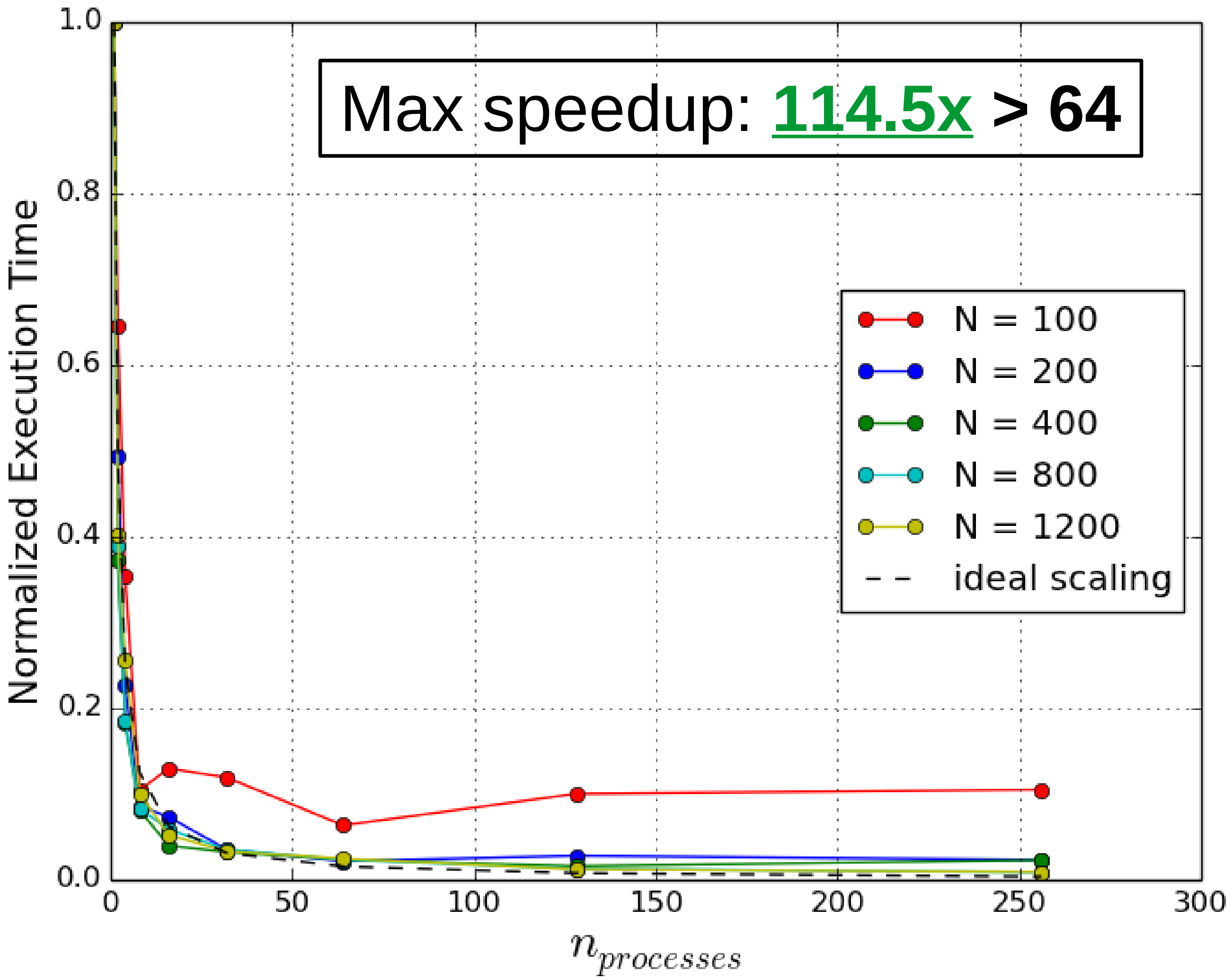
```
nl = get_list_of_nodes()
addprocs(nl)
```



Timing Results

```julia
@everywhere myfun(N,M) = sum(randn(N,M)^2)

map(N -> myfun(N,N),repmat([N],250)) #serial
pmap(N -> myfun(N,N),repmat([N],250)) #parallel
```

$\Big\} \sim N^3$



64 cores per machine max.

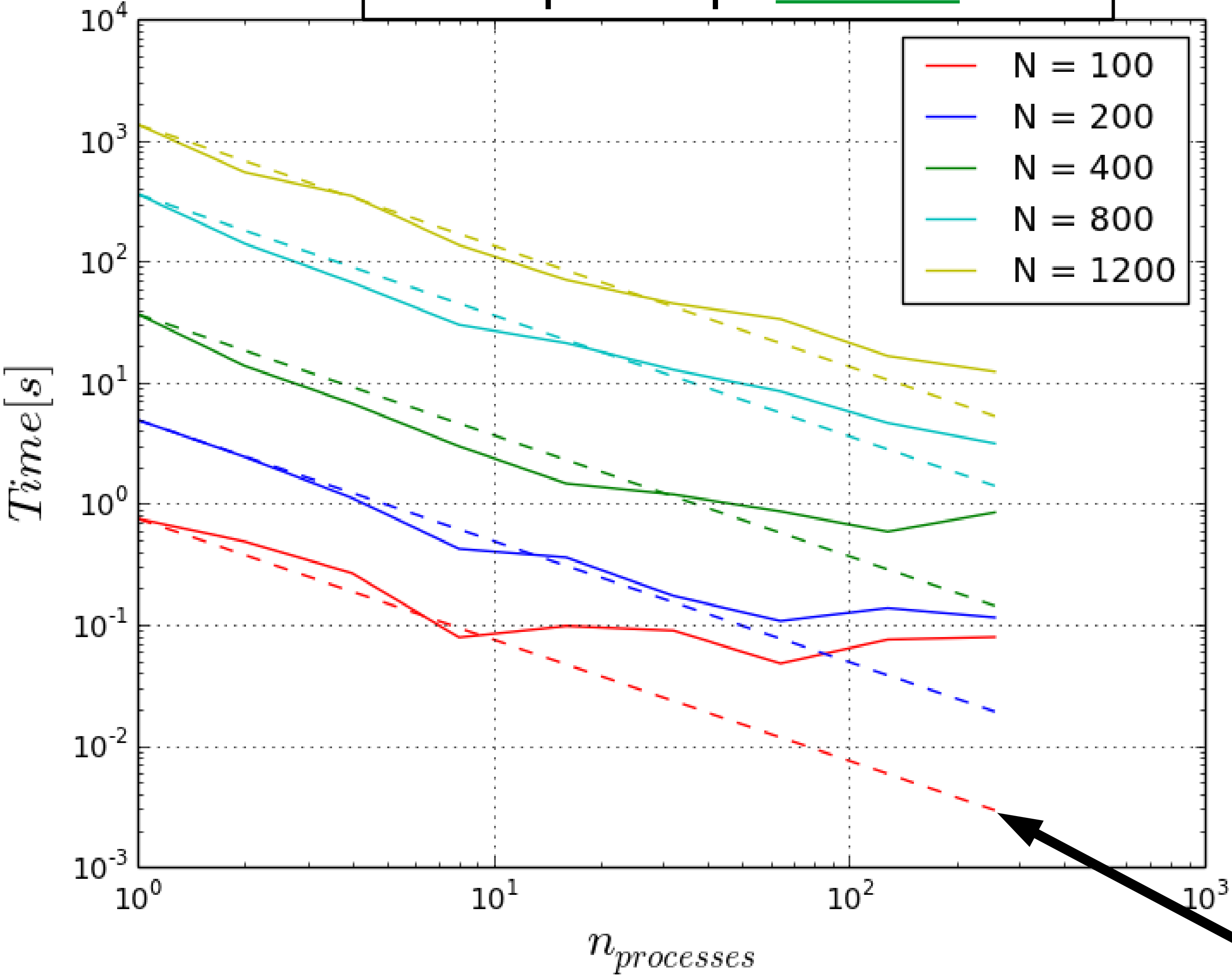
256 cores **total**.

This allocation:

- 5 nodes
- cpus_per_node: 56,16,64(x2),56

Timing Results

Max speedup: **114.5x** > **64**



$$\sim \frac{1}{n_{processes}}$$

64 cores per machine max.

256 cores **total**.

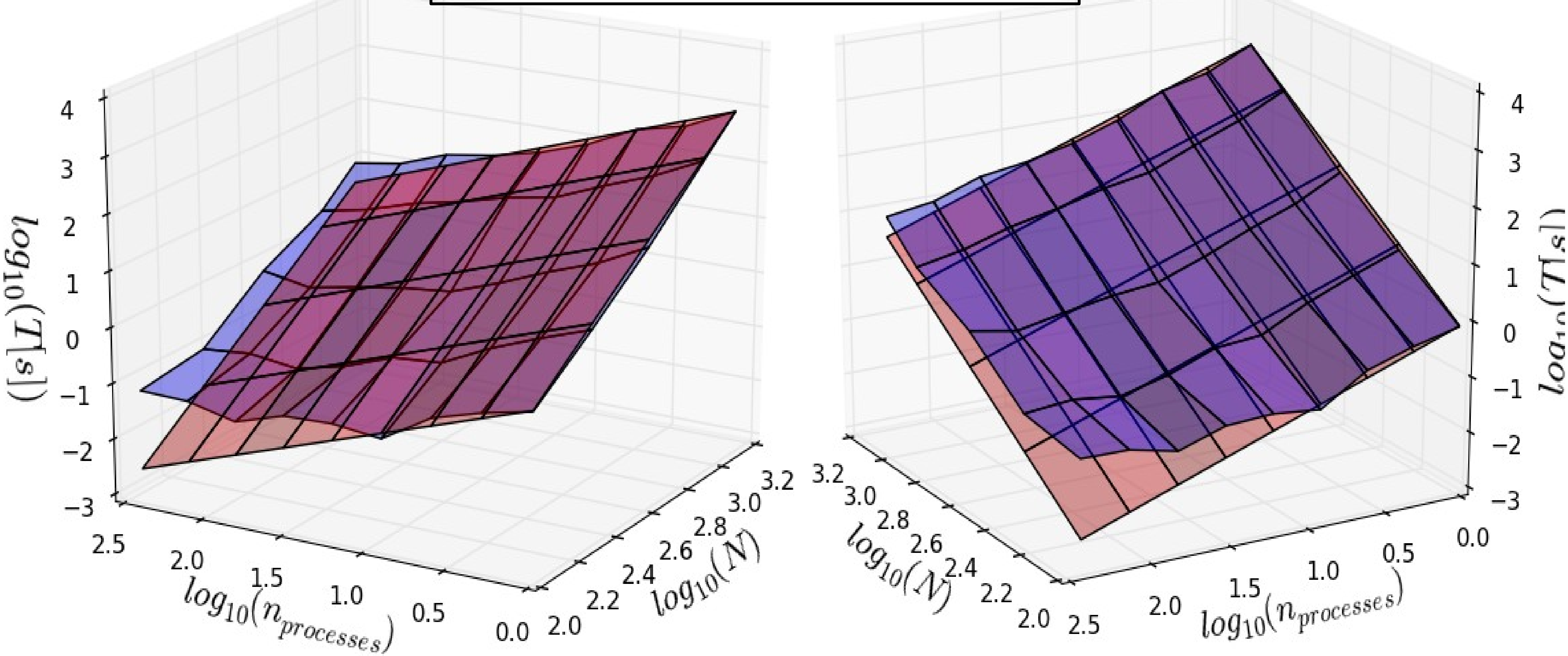
This allocation:

- 5 nodes
- cpus_per_node: 56,16,64(x2),56

Timing Results (cont'd)

Ideal scaling (red) $\sim \dfrac{N^3}{n_{threads}}$

Max speedup: **<u>114.5x</u> > 64**



Multiprocessing Potential Bugs

- Need to define @everywhere:
  - Variables, Functions and Modules used in @parallel
- Careful with anonymous/curried functions!

```
addprocs(2)

Nlist = repmat([1000],10)

#define function to execute
@everywhere myfun(N,M) = sum(randn(N,M)^2)

#define some local variable
@everywhere M = 1000 #will not work without @everywhere!

#map over curried function: make sure all captured variables
are defined @everywhere!
@time pmap(N -> myfun(N,M),Nlist)
```

Fine Grained Parallelization

Opportunity for **shared memory parallelism**!

Need to **lock**!

```
function run_graph_simulation(g::Graph)
  #main simulation loop
    for t in 1:num_timesteps

        #outer loop
        for v in vertices(g)
            result = 0

            #inner loop
            for w in neighbors(g,v)
                #computation
                result =
some_function(result,w)
            end

            #write operation!
            update_node_value(g,v,result)

        end

    end
end
```

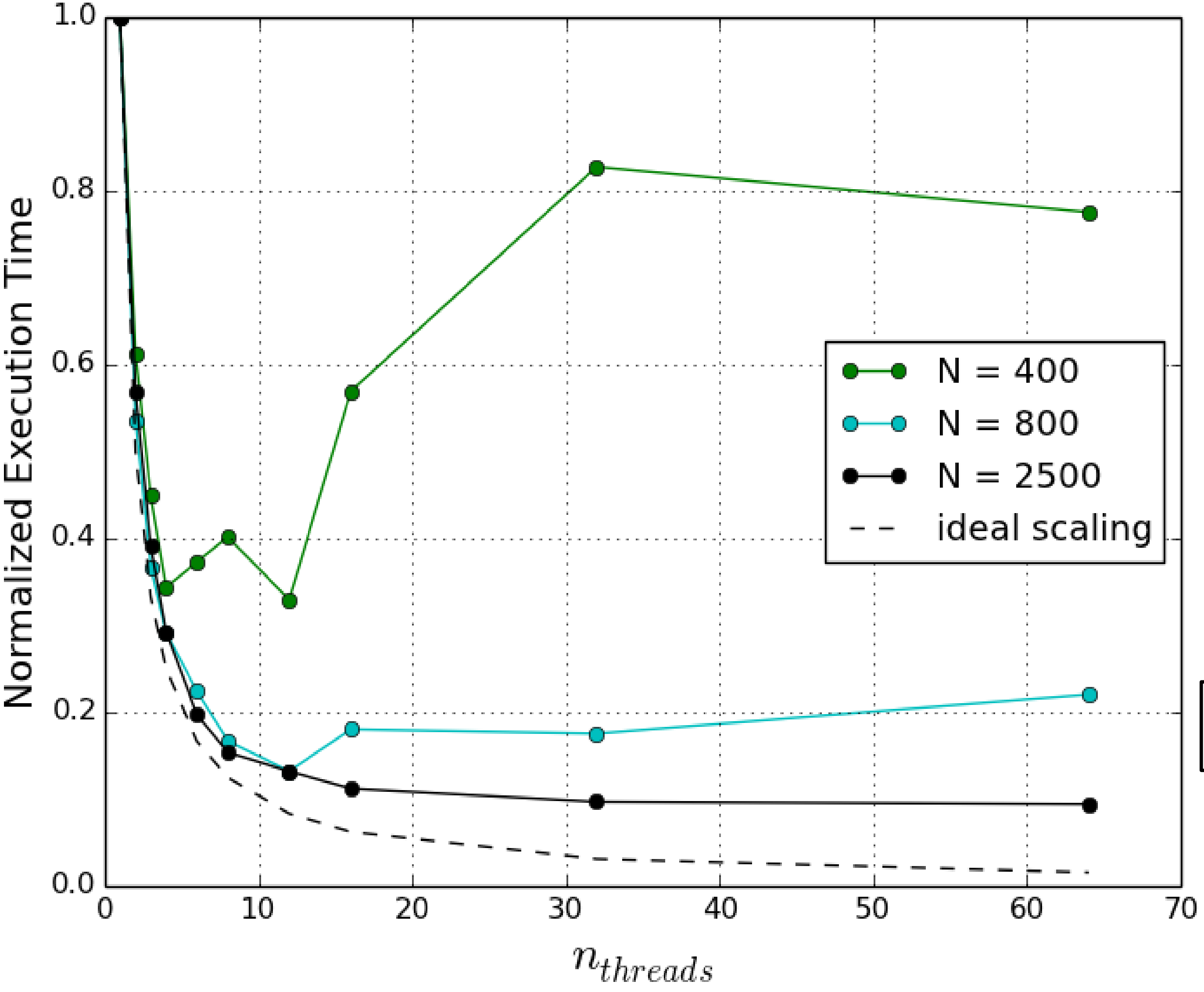
Fine Grained Parallelization Cont'd

```
function run_graph_simulation(g::Graph)
  #main simulation loop
  m = Mutex()
    for t in 1:num_timesteps

        #outer loop
        @threads all for v in vertices(g)
            result = 0

            #inner loop
            for w in neighbors(g,v)
                #computation
                result = some_function(result,w)
            end

            #write operation!
            lock!(m);
            update_node_value(g,v,result)
            unlock!(m);

        end

    end
end
```

This is all we need!

…in the future

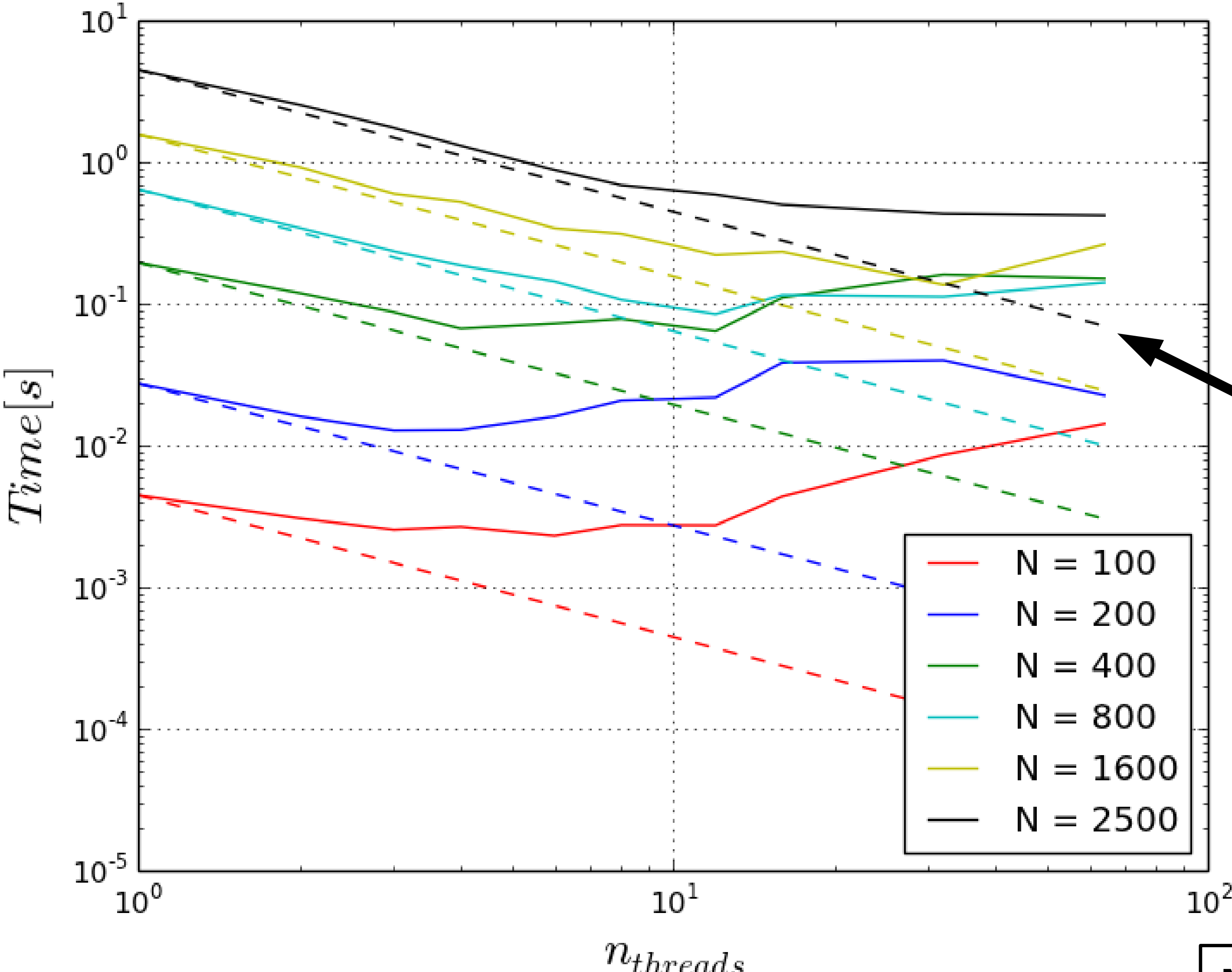
Timing Results



80 core machine, with subdivision of **8** cores.

Max speedup: **11.5x**

Timing Results



80 core machine, with subdivision of **8** cores.

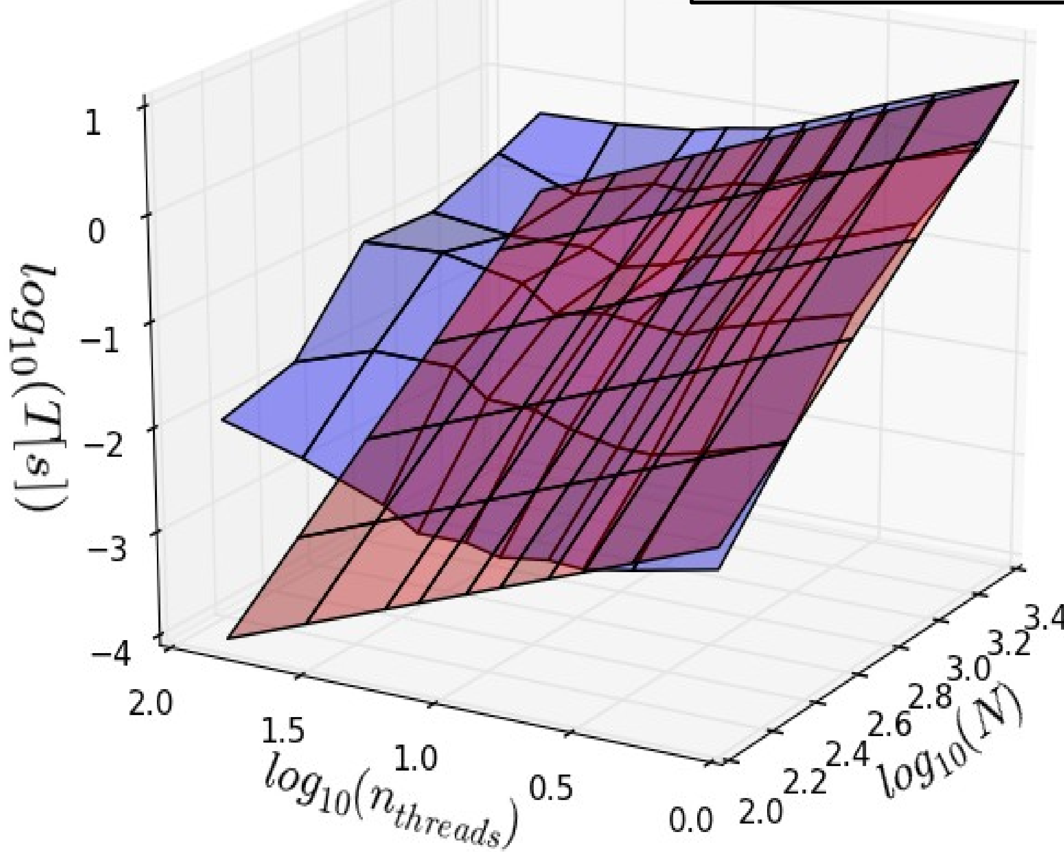
$$\sim \frac{1}{n_{threads}}$$

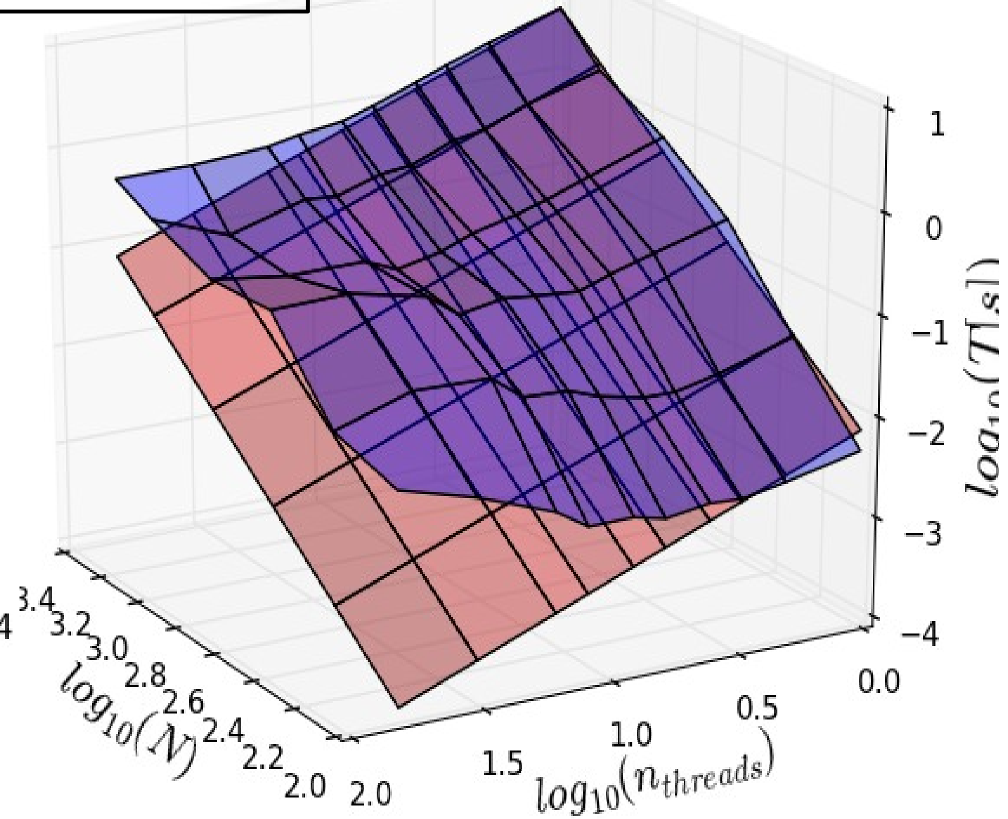
Max speedup: **11.5x**

Timing Results (cont'd)

Ideal scaling (red) $\sim \dfrac{N^2}{n_{threads}}$

Max speedup: **11.5x**





Examples of threading bugs:

- All errors (syntax, compiler, runtime) are *ignored* during threaded execution... **silent no-op**.
- Any modification to global state breaks.
  - Random number generators
  - Type instabilities, etc.
- Can't do too much within `lock!() - unlock!()` block.
- Functions passed as data break.
  - But globally defined functions don't! (→ Example)

Minimal Threading Instability:

```
type CarryFunction
  fn::Function
end

alpha = 0.1
fn(x) = alpha*x

function use_anonymous(N::Int,c::CarryFunction)
  a = zeros(N)
  @threads all for i in 1:length(a)
    # a[i] = fn(i) #NO SEGFAULT
    a[i] = c.fn(i) #SEGFAULT (sometimes... but not always!)
  end
  println(a[1],a[end])
end

length = 10000
repetitions = 100
for j = 1:repetitions
  use_anonymous(length, CarryFunction(fn) )
end
```

Conclusions

- Developed parallel graph algorithms using
  - **Cluster Multiprocessing**
  - **Mulithreading**
    - Also tried multiprocessing for fine grained parallelism: much slower
      - Lots of sharing required (shared memory multiprocessing in its infancy)
- Developed **general process manager** for SLURM clusters
- Speedups indicate **full utilization** of computing resources by Julia
- **Most time** spent: **debugging parallel code**, both multiprocessing and multithreading
  - Cryptic errors messages, unknown culprits (“which line was it anyway?”)
    - Binary search!
  - Heisenbugs (once every 100,000 runs!?)
  - Getting the data parallelism/sharing right.
  - Making sure all resources are properly utilized

Questions?

Thank you :)