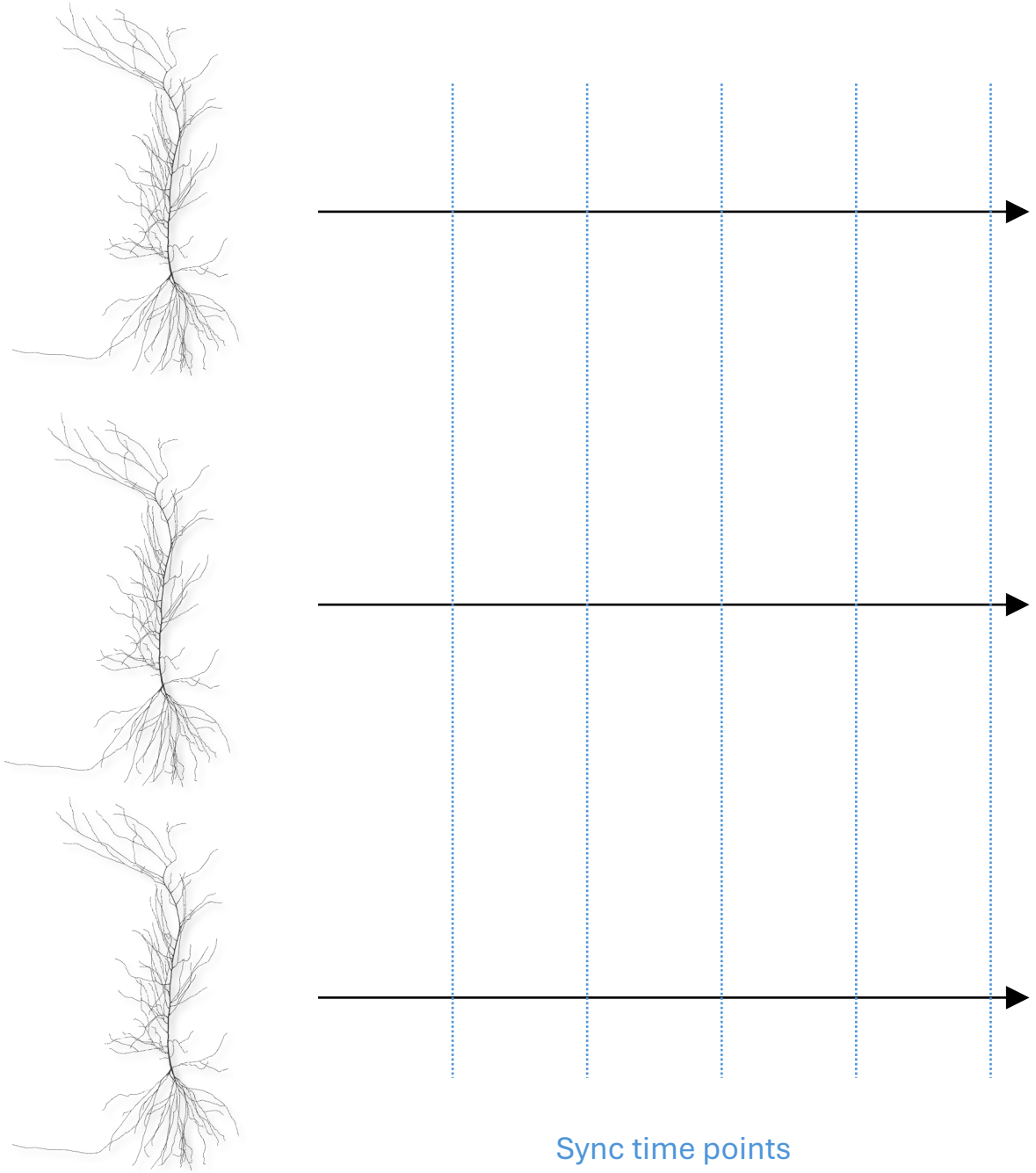
Building, running, and visualizing parallel NEURON models



Why write parallel code?

Get results faster.

Solve larger problems in the same amount of time.

Work on large problems that do not fit in a single machine’s memory.

Run more simulations (e.g., parameter sweeps)

Move forward while waiting for data (e.g., from a networked machine.)

Why to **not** write parallel code

- Parallel code is more difficult to write and to debug.
- There are new types of bugs that only occur in parallel contexts. (e.g., race conditions).
- It is even possible to parallelize code and end up with something that runs slower than the original (e.g., due to communication overhead).
- Your time matters too; will the run-time savings be more than the extra time that you spend writing?

Three main classes of parallel problems

Parameter sweeps

- Running the same (typically fast) simulation 1000s of times with different parameters is an example of an *embarrassingly parallel problem*. This can be done without MPI using Python's built-in `multiprocessing` module, or with MPI using NEURON's bulletin boards.

Distributing networks across processors

- Cells can communicate via:
  - Logical spike events with significant axonal or synaptic delay.
  - Postsynaptic conductance depending continuously on presynaptic voltage.
  - Gap junctions.

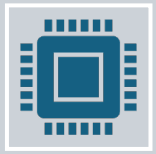
Distributing single cells across processors

- The *multisplit* method distributes portions of the tree cable equation across different machines.

A parallel model can fall in any or all these classes.

Some parallel philosophy

A network of neurons is composed of many individual neurons of potentially many cell types. As much as possible, design and debug each cell type separately before building the network.



A simulation should give the same results regardless of the number of processors used to run it.

When possible, parameterize your network so you can run a small test first.

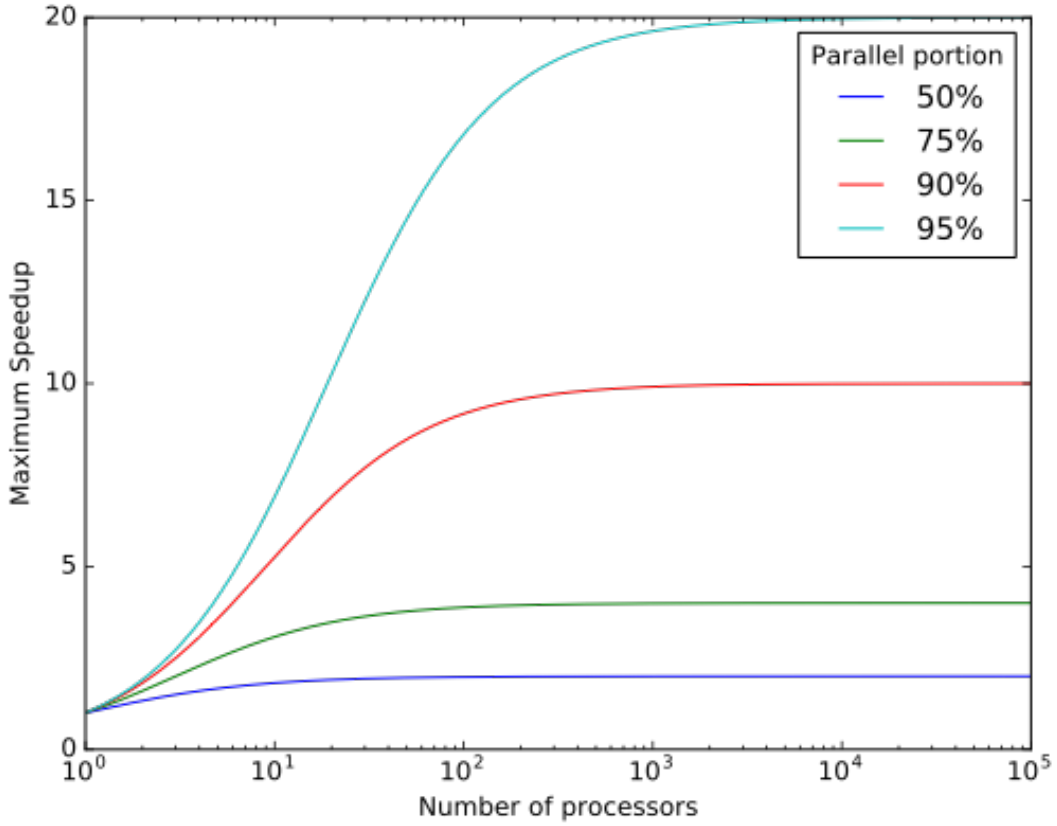
A tiny bit of theory...

Parallel scaling performance

- Suppose I have a serial simulation that runs in 100 minutes. How fast could I expect a parallel version of it to run on 1000 processors?

- Suppose further that 1% of the code could not be parallelized. How does that affect your answer?

Amdahl's Law: Max speedup = $\frac{1}{(1-p)+\frac{p}{n}}$

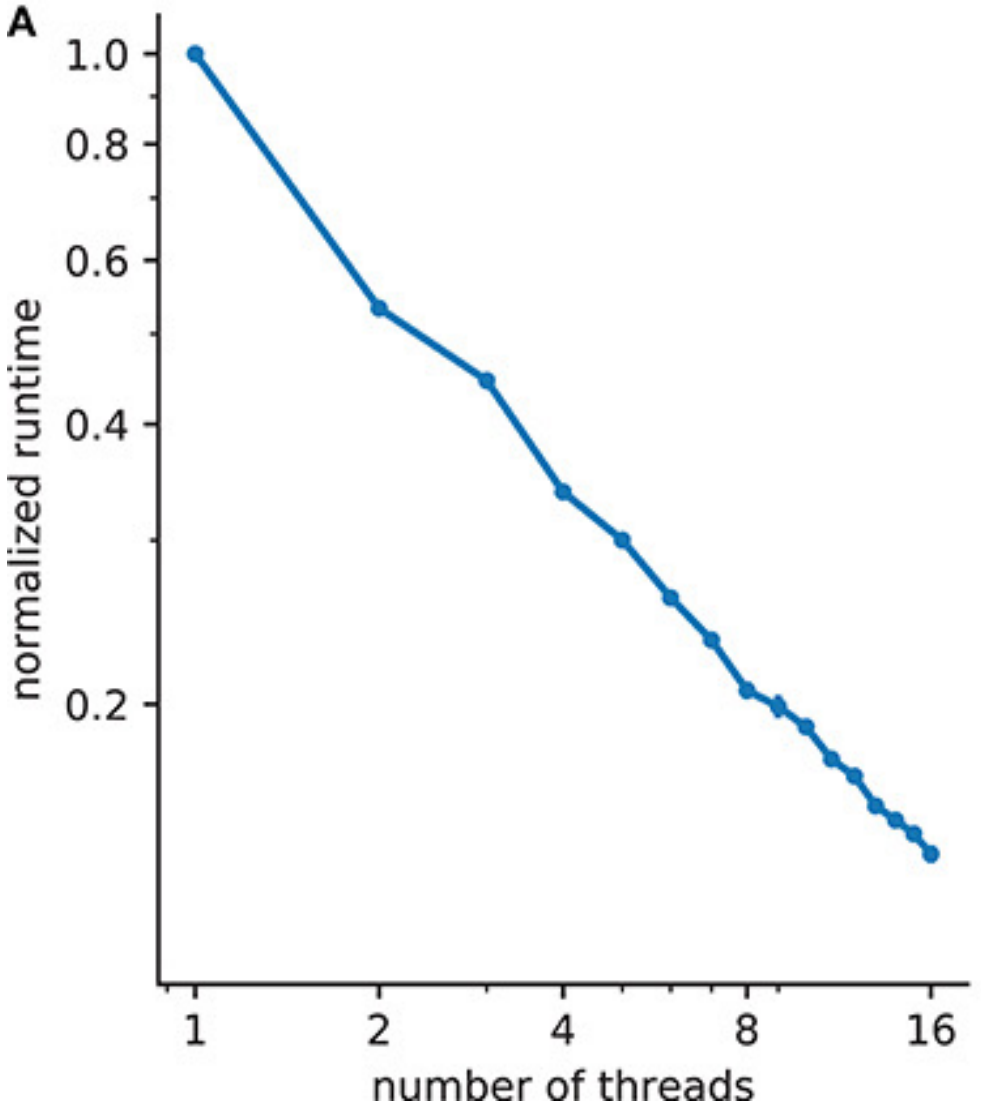


Implicit assumption

- Amdahl's law assumes that p is a constant.
- In practice, we probably do not want to do the same simulation in parallel that we did in serial; we probably want to do a bigger model.
- Loading e.g. neuron takes the same amount of time regardless of the size of our problem. So p may increase as n increases.

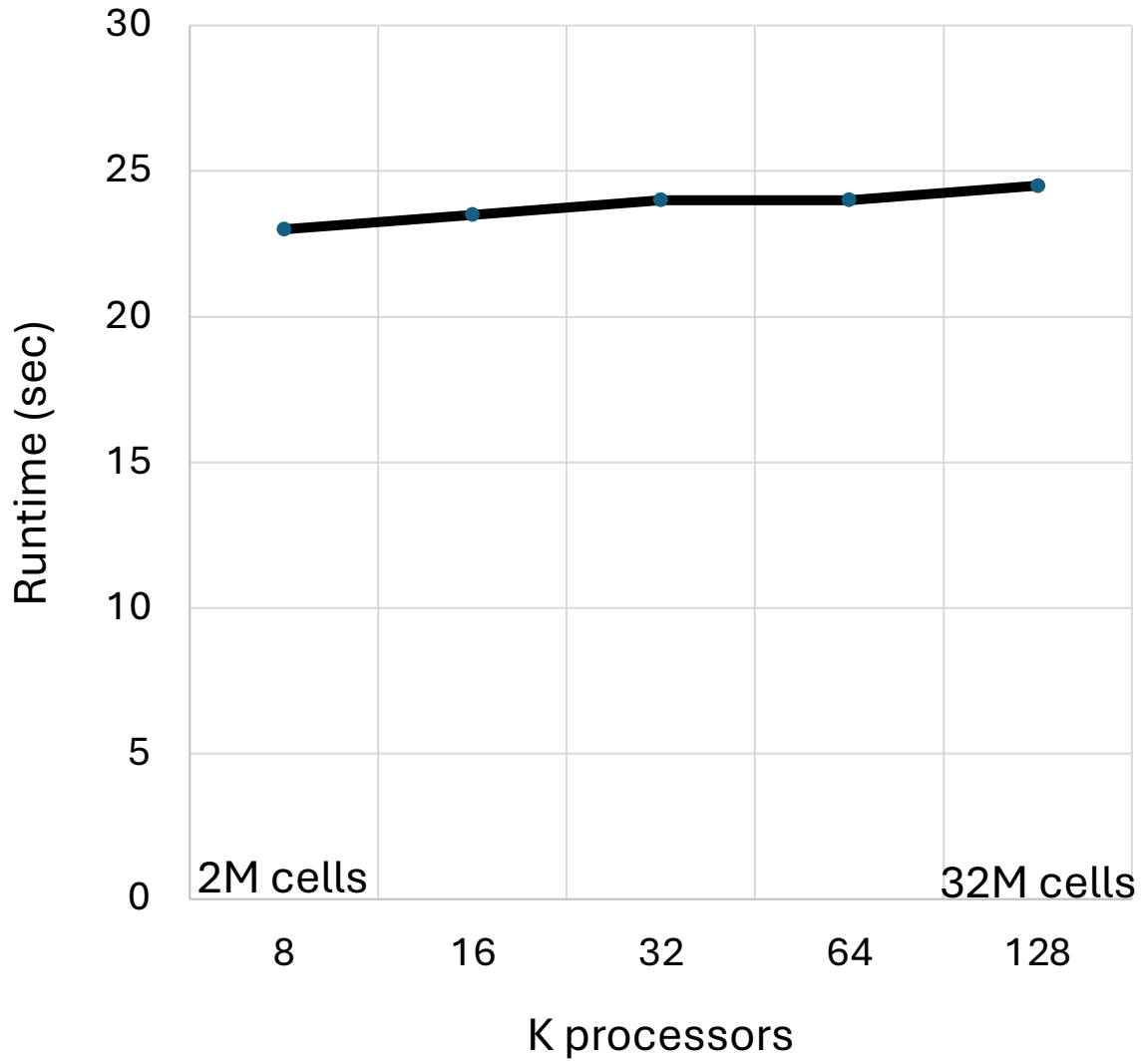
Strong scaling

- Fixed problem
- Measure speedup per processor
- On a log-log graph of threads/processes vs runtime, ideal is a graph that looks linear with slope -1



Weak scaling

- Fixed amount of work per thread/processor.
- Measure relationship between size of problem and runtime.
- Ideal is a flat relationship.



So how do we design a model to be parallelizable?

Who does what?

`pc.nhost` (size) and `pc.id` (rank)

Idea: Use a predefined rule for splitting up the simulation. Since all processors know the rule, they know which parts of the simulation they must setup and simulate.

squaretest.py

```
from neuron import h
h.nrnmpi_init()
pc = h.ParallelContext()

val = pc.id()
squared = val ** 2

print(
    f"I am rank {val} (of {pc.nhost()}) "
    f"and my square is {squared}"
)
```

```
% mpiexec -n 4 python squaretest.py
numprocs=4
I am rank 1 (of 4) and my square is 1
I am rank 2 (of 4) and my square is 4
I am rank 3 (of 4) and my square is 9
I am rank 0 (of 4) and my square is 0
```

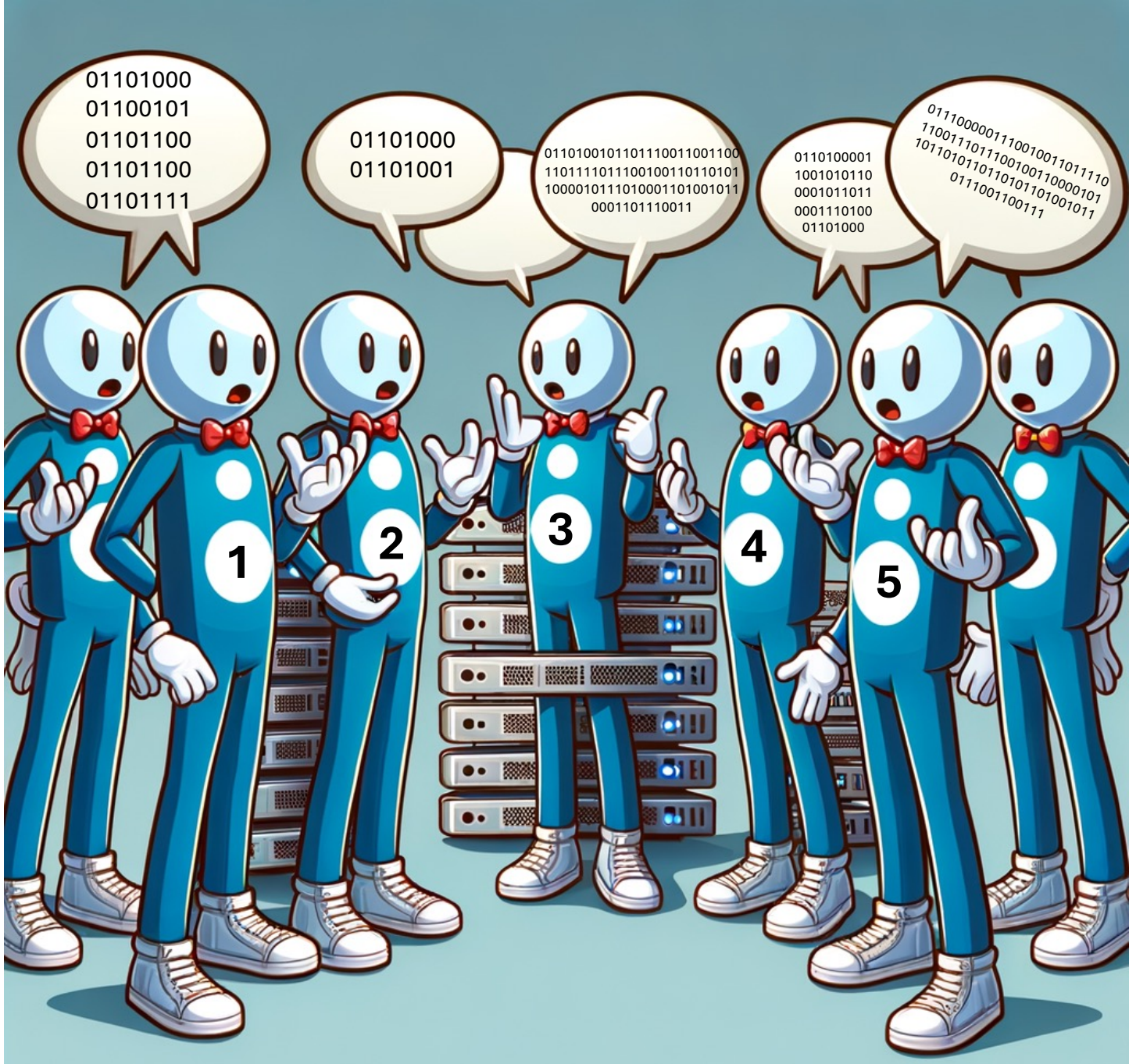


Image by Dall-E 3 2024-11-12 • Additional elements by Robert McDougal

That is, let's give all our cell classes a `gid`. If we parallelize, that will help us decide which processor id handles it.

- In addition, it will be convenient to specify morphology in a dedicated method and add a `__repr__` method to identify the object.
- Here, the `gid` should be a globally unique identifying integer. We do not use class variables to generate the integer automatically because: (1) the numbers should not repeat between different processors, and (2) we may wish to recreate a single specific cell instead of the entire network.

```
from neuron import h
h.load_file('import3d.hoc')

class Pyramidal:
  def __init__(self, gid):
    self._gid = gid
    self._setup_morphology()
  def _setup_morphology(self):
    cell = h.Import3d_SWC_read()
    cell.input('c91662.swc’)
    i3d = h.Import3d_GUI(cell, 0)
    i3d.instantiate(self)
  def __repr__(self):
    return f'Pyramidal[{self._gid}]'
```

Discretize, declare channels, set parameters

```
class Pyramidal:
  def __init__(self, gid):
    self._gid = gid
    self._setup_morphology()
    self._discretize()
    self._add_channels()
  def _setup_morphology(self):
    cell = h.Import3d_SWC_read()
    cell.input('c91662.swc’)
    i3d = h.Import3d_GUI(cell, 0)
    i3d.instantiate(self)
  def __repr__(self):
    return f'Pyramidal[{self._gid}]'
  def _discretize(self, d_lambda=0.1, freq=100):
    h.load_file('stdlib.hoc')
    for sec in self.all:
      sec.nseg = 1 + 2 * math.ceil(
        (sec.L / (d_lambda * h.lambda_f(freq, sec=sec))) / 2)
  def _add_channels(self):
    h.hh.insert(self.soma)
    h.pas.insert(self.all)
    h.pas.uninsert(self.soma)
    for sec in self.axon + self.dend + self.apic:
      for seg in sec:
        seg.pas.g = 0.001
```

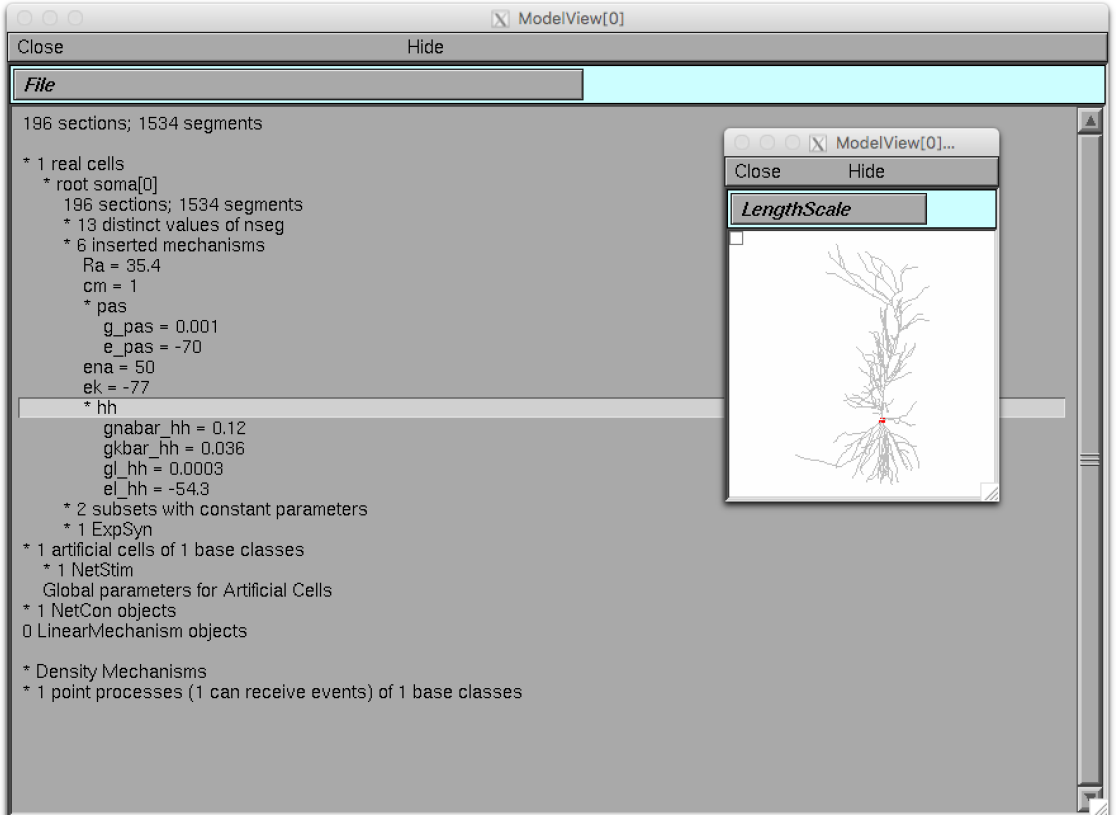
Python implementation of the d_lambda rule (Hines and Carnevale 2001).

Also: Ensures that there is an odd number of segments, which means there is a middle node. To test convergence, always multiply nseg by an odd number, e.g.,

```
for sec in self.all:
    sec.nseg *= 3
```

Examine for errors: Tools → ModelView

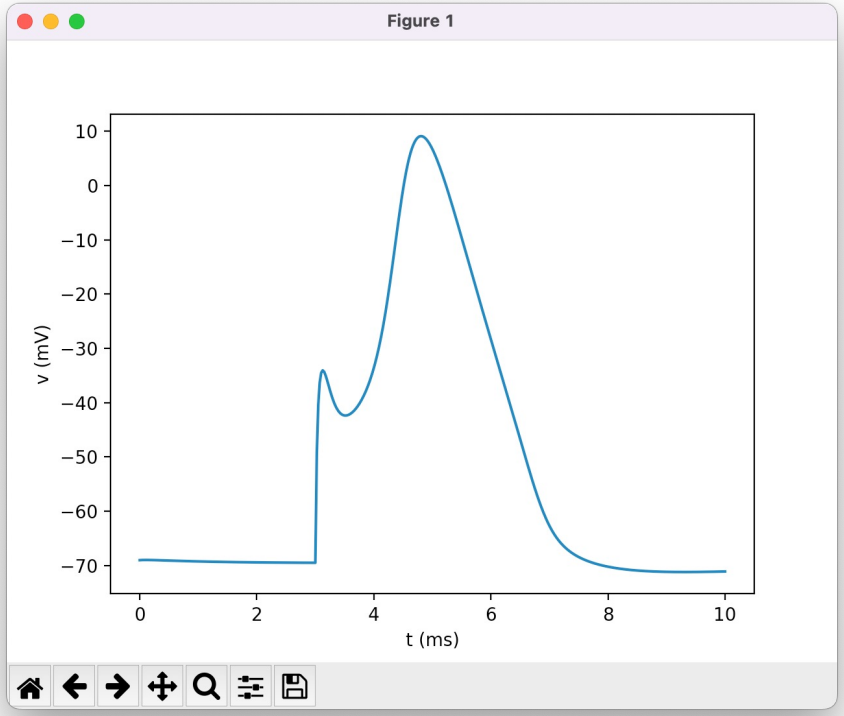
- Networks are made of cells.
- Always check each neuron or neuron type before combining into a network.
- Errors can still occur when neurons are combined into networks, but if the neurons are wrong then the network simulation definitely will be wrong.



Now: Run with `pc.psolve(tstop)`

```
from neuron import h
from matplotlib import pyplot
import math
h.load_file('stdrun.hoc')
h.load_file("import3d.hoc")
h.load_file("stdlib.hoc")

# class Pyramidal defined as before (omitted)

myPyramidal = Pyramidal(0)

postsyn = h.ExpSyn(myPyramidal.dend[0](0.5))
postsyn.e = 0 # reversal potential

stim = h.NetStim()
stim.number = 1
stim.start = 3
ncstim = h.NetCon(stim, postsyn)
ncstim.delay = 0
ncstim.weight[0] = 0.5

t = h.Vector().record(h._ref_t)
v = h.Vector().record(myPyramidal.soma[0](0.5)._ref_v)
pc = h.ParallelContext()
pc.set_maxstep(10)
h.finitialize(-69)
pc.psolve(10)
```

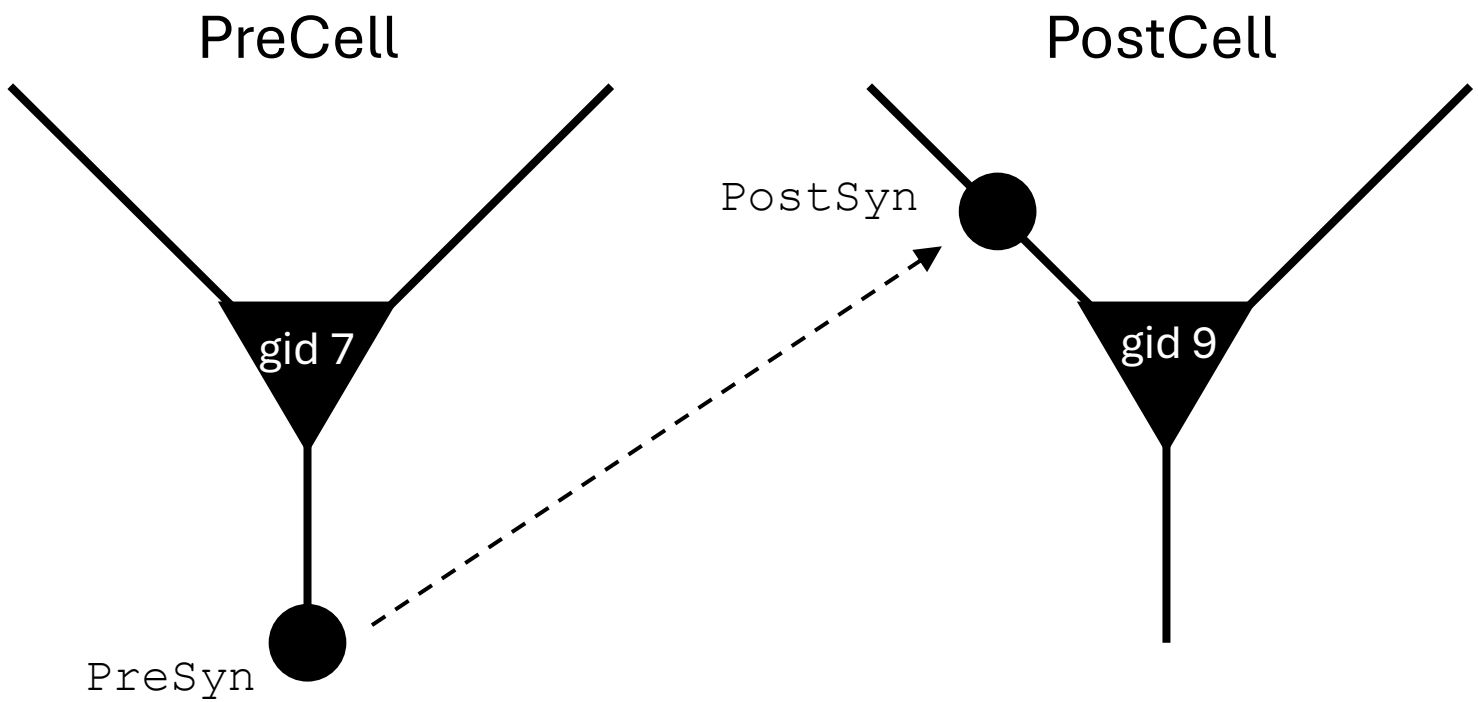
```
pyplot.plot(t, v)
pyplot.xlabel('t (ms)')
pyplot.ylabel('v (mV)')
pyplot.show()
```



For this example, assume all synaptic delays are 2 ms.

Building synapses

- Typically, we may think of `PreSyn` as a voltage at a location that we want to monitor for crossing a threshold.
- `PostSyn` is some synaptic mechanism point process, e.g., `h.ExpSyn`.
- In a serial model, we would connect directly with an `h.NetCon`.



For this example, assume all synaptic delays are 2 ms.

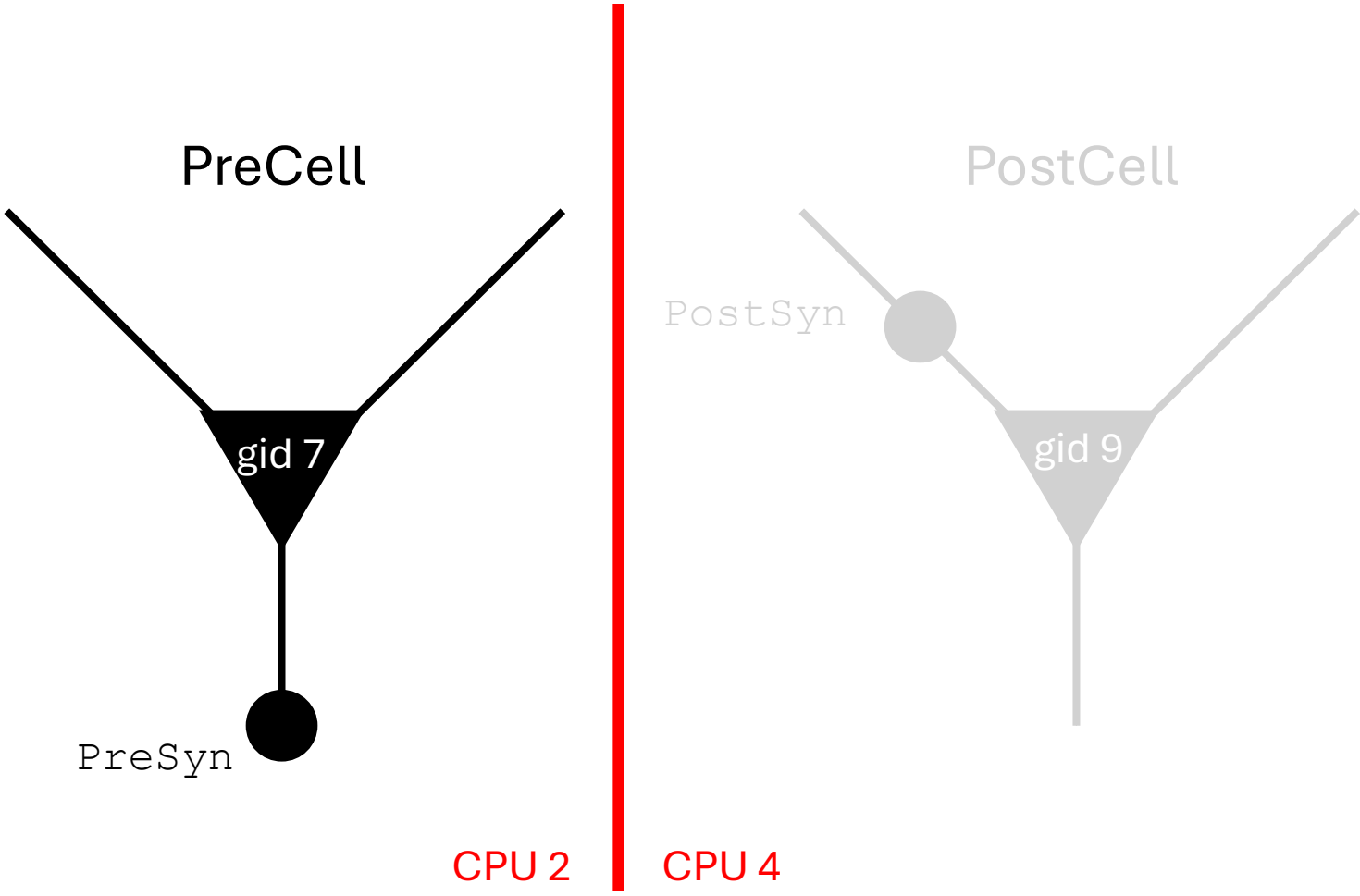
Configuring the presynaptic connection site

- In parallel contexts, one CPU cannot access variables stored on another CPU, so instead we must use the gids.
- Create cell only where the gid exists:

```
if pc.gid_exists(7):
  PreCell = Cell()
```

- Associate gid with spike source:

```
nc = h.NetCon(PreSyn, None, sec=presec)
pc.cell(7, nc)
```



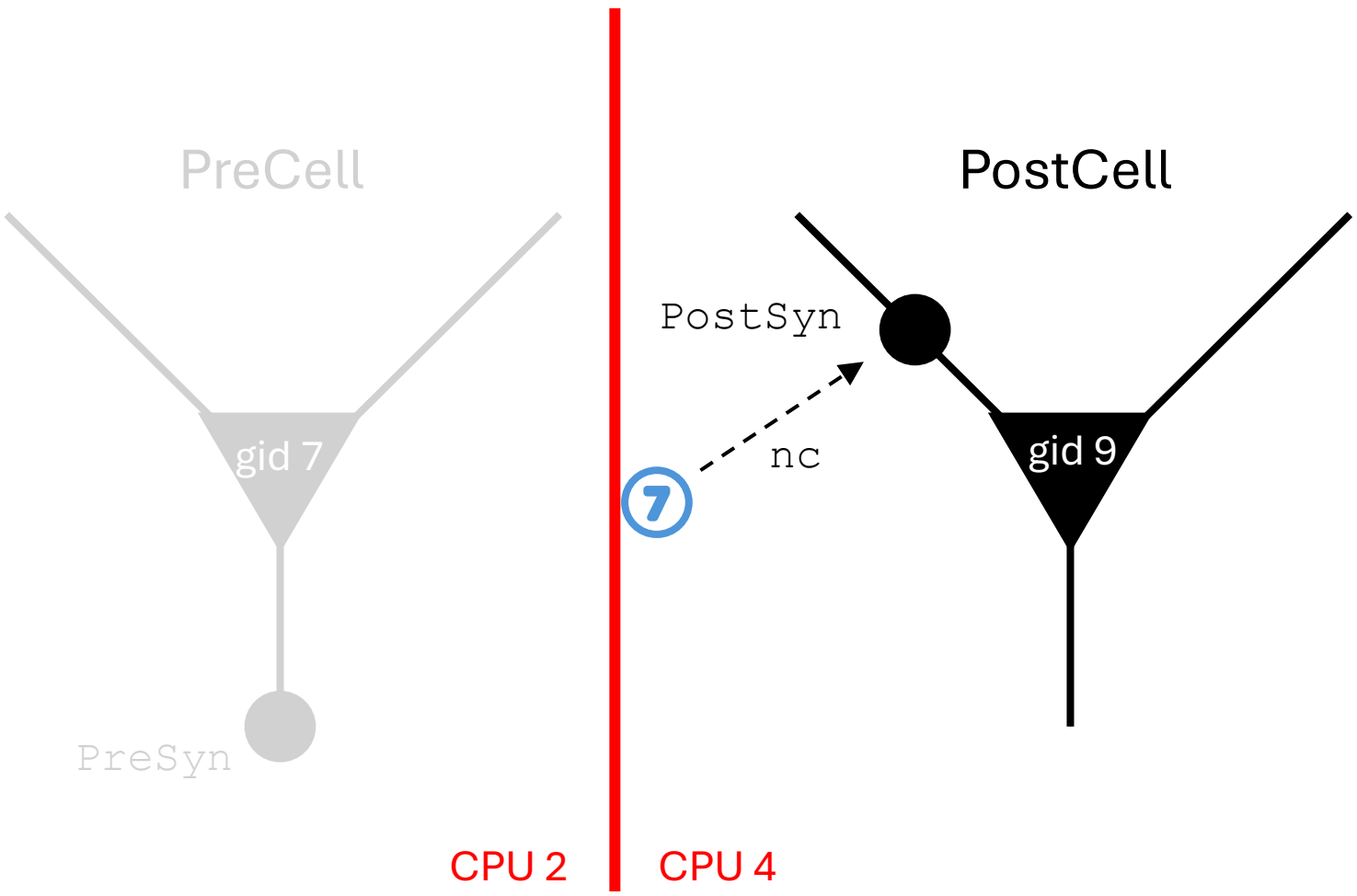
For this example, assume all synaptic delays are 2 ms.

Configuring the postsynaptic connection site

- Create `h.NetCon` on node where target exists:

```
nc = pc.gid_connect(7, PostSyn)
```

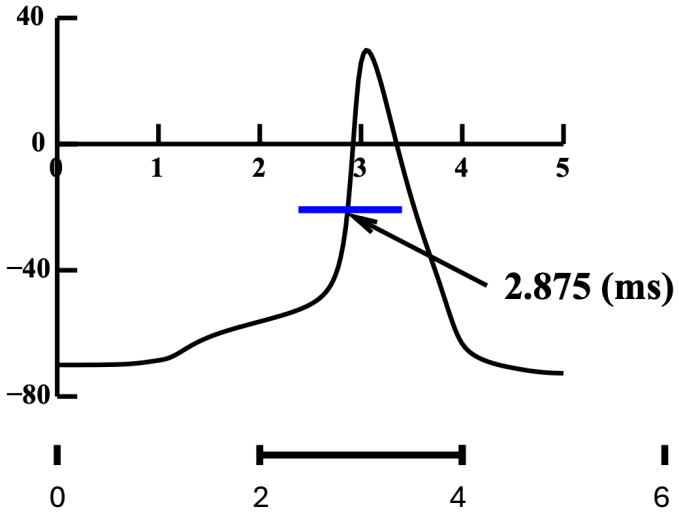
- `PostSyn` here is a Point Process with a `NET_RECEIVE` block, e.g., an `h.ExpSyn` but there are many synapse models available at https://modeldb.science

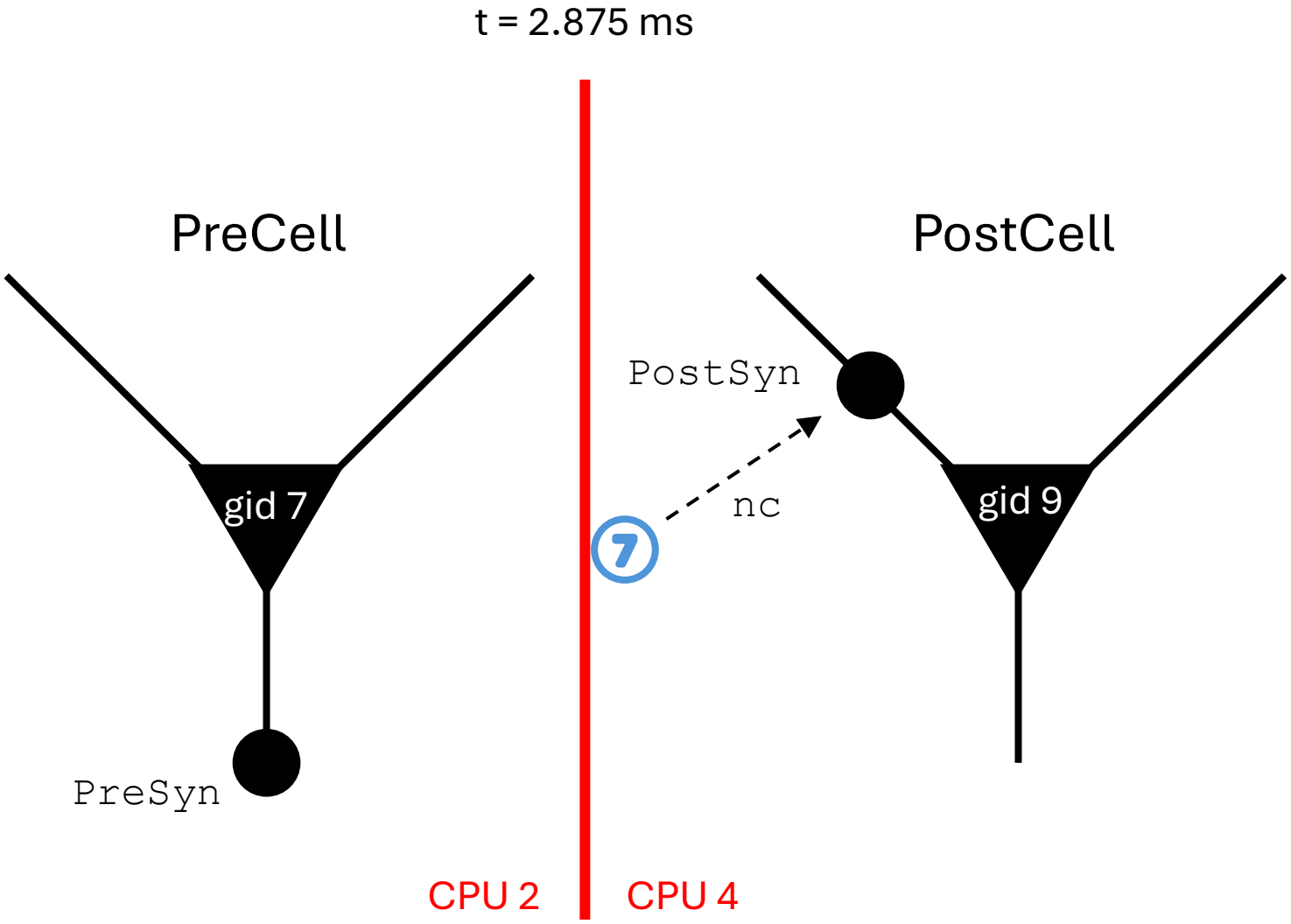


For this example, assume all synaptic delays are 2 ms.

Spike exchange method

- Step 1: Detect the time of a spike.

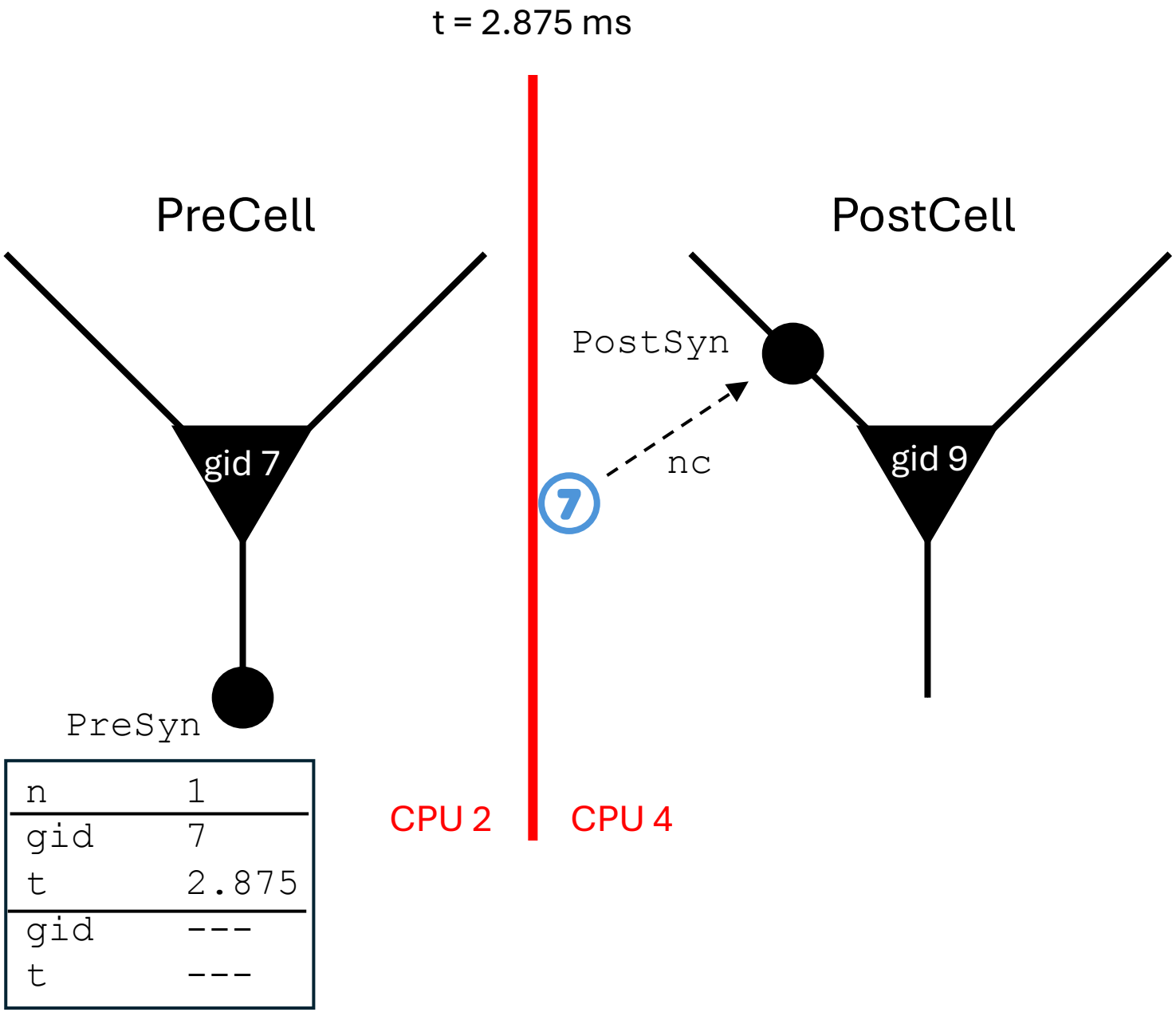




For this example, assume all synaptic delays are 2 ms.

Spike exchange method

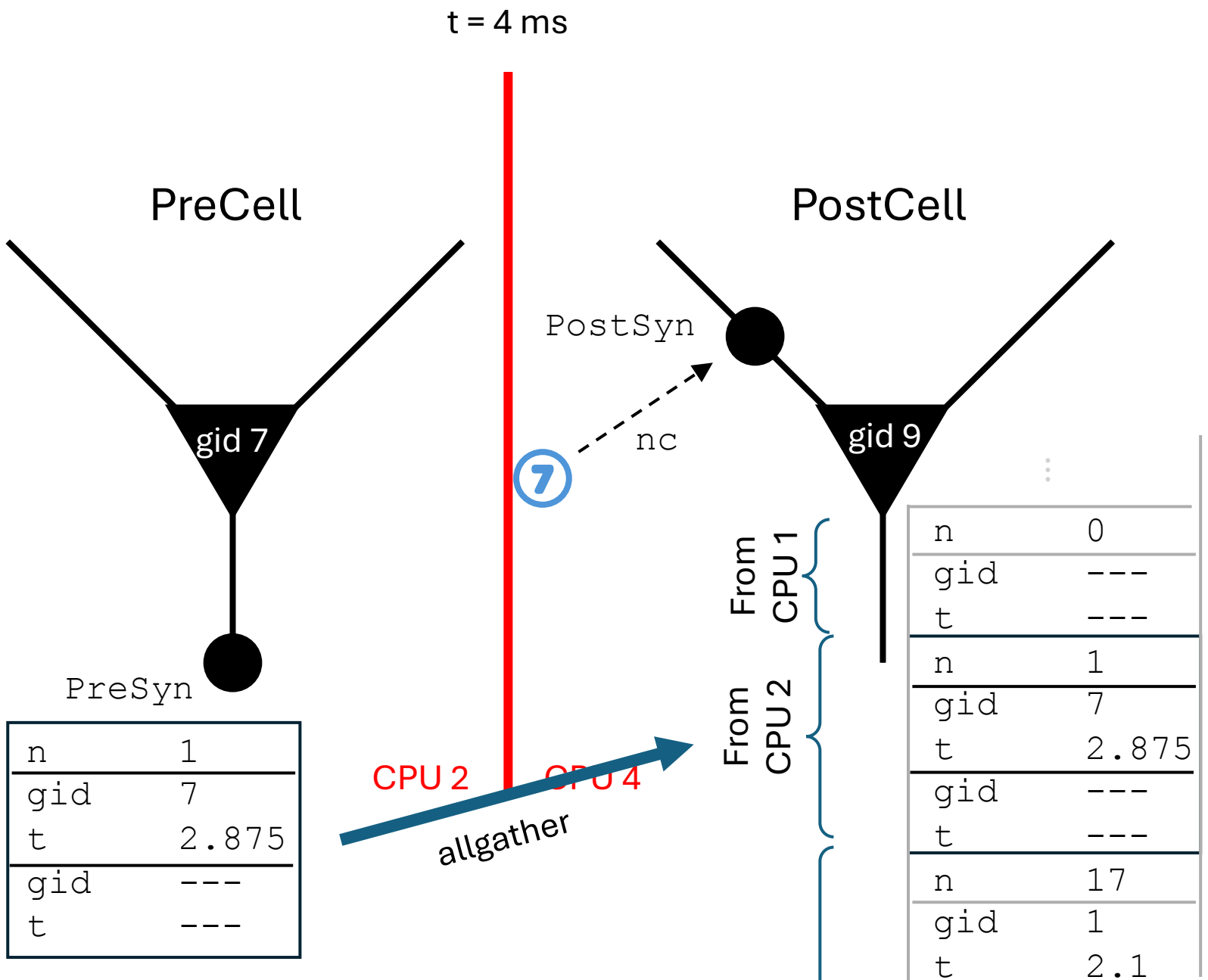
- Step 1: Detect the time of a spike.
- Step 2: Store in a queue.



For this example, assume all synaptic delays are 2 ms.

Spike exchange method

- Step 1: Detect the time of a spike.
- Step 2: Store in a queue.
- Step 3: Transfer via MPI's allgather.

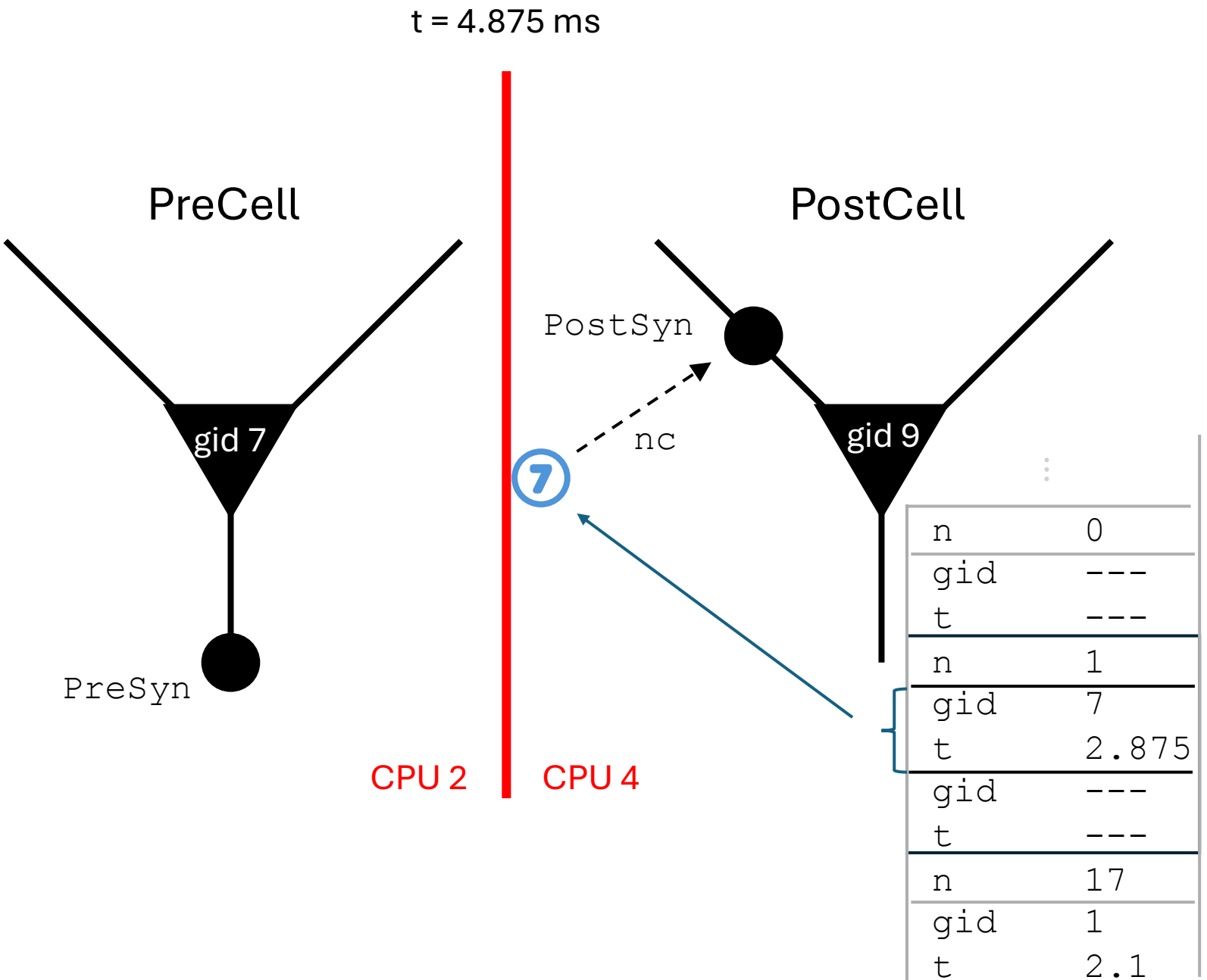


Collective communication: `allgather`

For this example, assume all synaptic delays are 2 ms.

Spike exchange method

- Step 1: Detect the time of a spike.
- Step 2: Store in a queue.
- Step 3: Transfer via MPI's allgather.
- Step 4: Deliver events at the appropriate time.



Minimize communication

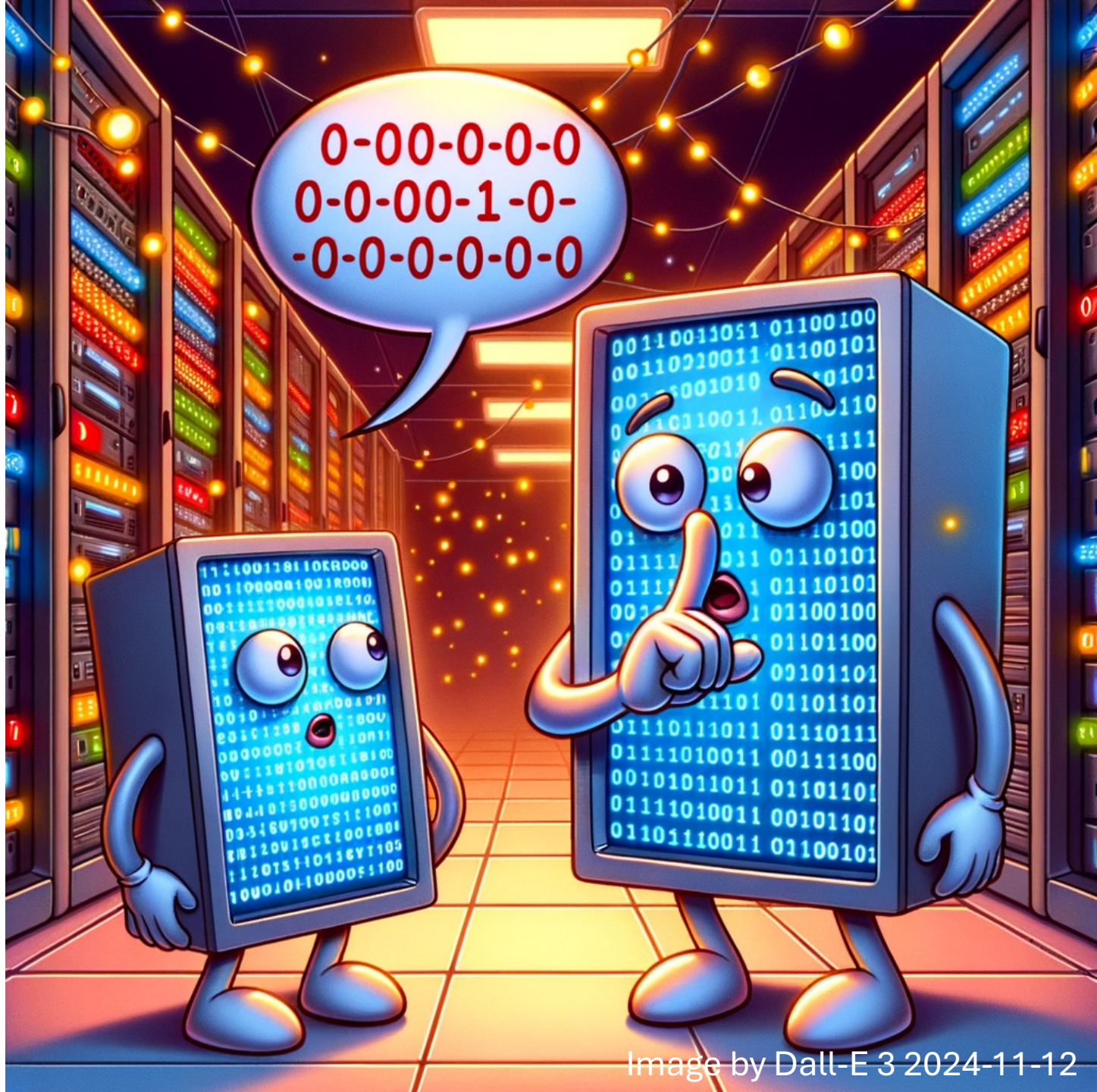


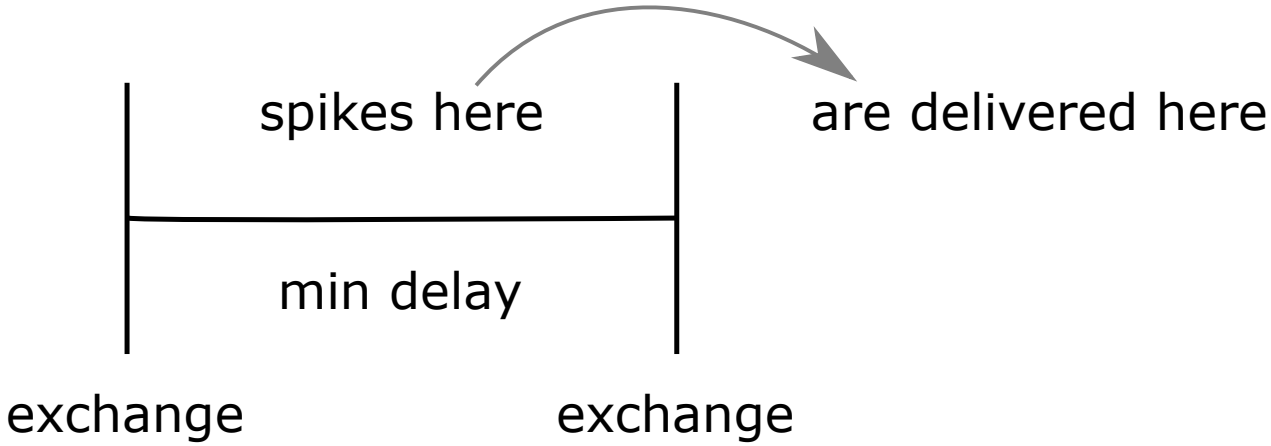
Image by Dall-E 3 2024-11-12

pc.set_maxstep

- NEURON will pick an event exchange interval equal to the smaller of all the `h.NetCon` delays and of the argument to `pc.set_maxstep`.
- In general, larger intervals are better because they reduce communication overhead.

Run parallel simulations using the idiom:

```
pc.set_maxstep(10)
h.finitialize(-65)
pc.psolve(tstop)
```



`pc.set_maxstep` must be called on each node; it uses MPI's `allreduce` to determine the minimum delay.

Back to our example...

- For most models, the delay due to axon propagation can be incorporated into a synaptic delay and thus it suffices to only make one connection point at the soma or axon hillock.
- `pc.set_gid2node` must be called before `pc.cell`.

```
class Pyramidal:
  def __init__(self, gid):
    self._gid = gid
    self._setup_morphology()
    self._discretize()
    self._add_channels()
    self._register_netcon()
  def _register_netcon(self):
    self.nc = h.NetCon(
      self.soma[0](0.5)._ref_v, None,
      sec=self.soma[0])
    self.nc.threshold = 0
    pc = h.ParallelContext()
    pc.set_gid2node(self._gid, pc.id())
    pc.cell(self._gid, self.nc)
  # the rest of the class stays unchanged
```

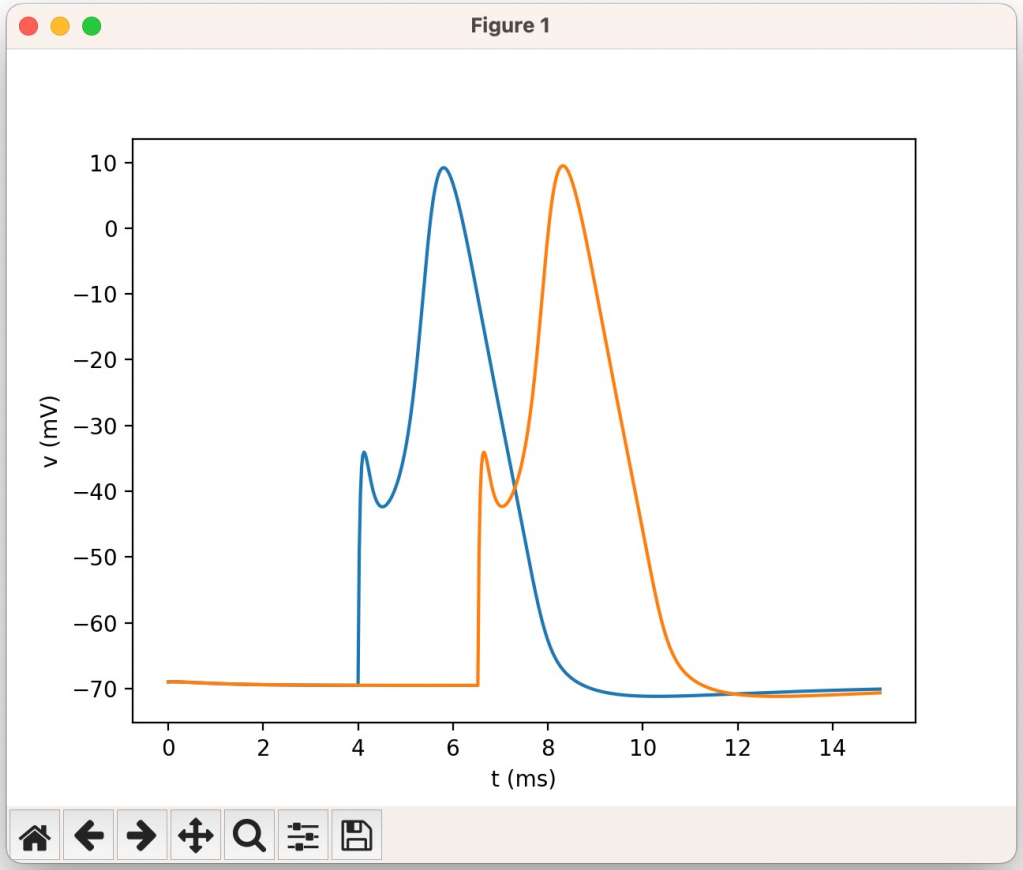
Building a two cell network

Note: we use `for` loops and list comprehensions even when there is only two cells to avoid repeating ourselves (the DRY-principle) and to allow future generalization.

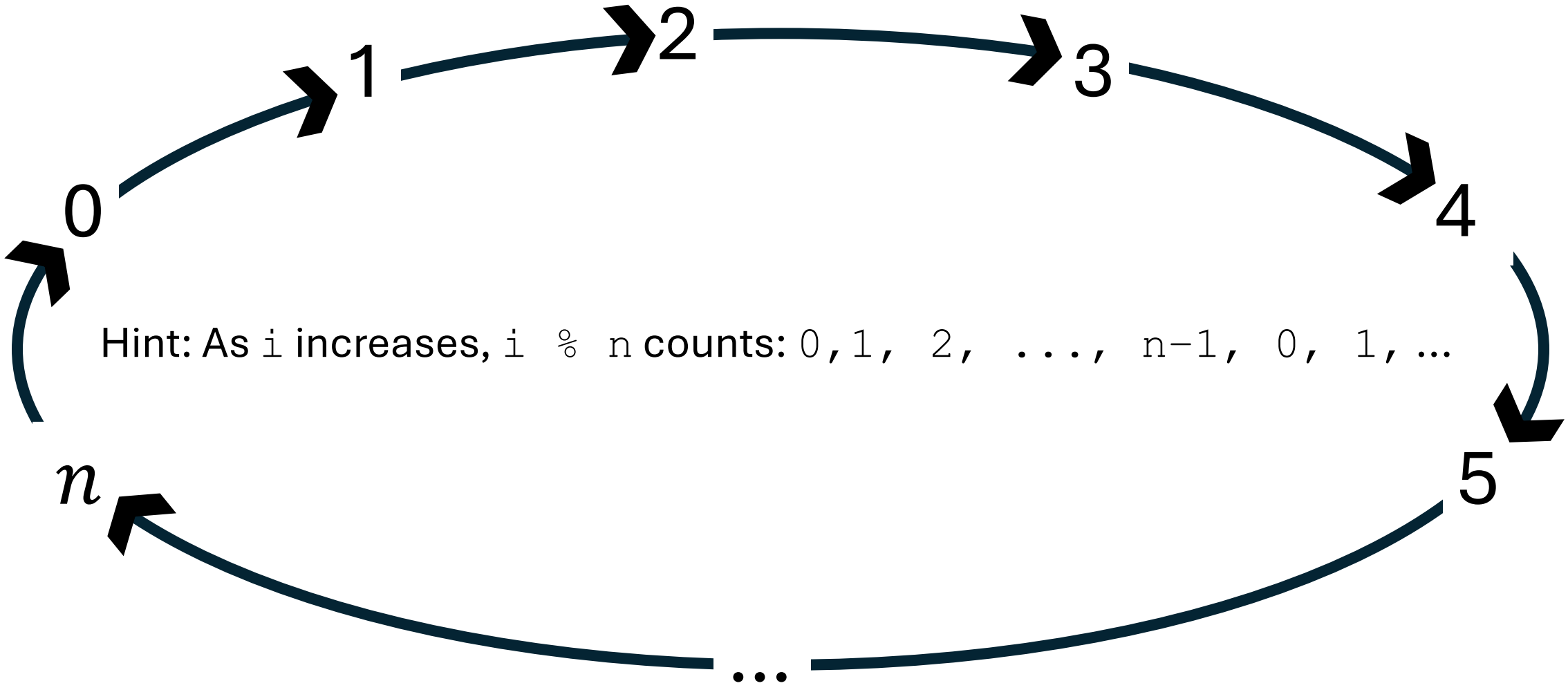
```
from neuron.units import ms, mV

class Network:
  def __init__(self):
    self.cells = [Pyramidal(i) for i in range(2)]
    # setup an exciteable ExpSyn on each cell's dendrites
    self.syns = [h.ExpSyn(cell.dend[0](0.5)) for cell in self.cells]
    for syn in self.syns:
      syn.e = 0 * mV
    # connect cell 0 to cell 1
    pc = h.ParallelContext()
    pre = 0
    post = 1
    self.nc = pc.gid_connect(pre, self.syns[post])
    self.nc.delay = 1 * ms
    self.nc.weight[0] = 0.5

n = Network()
```

Running the two cell network

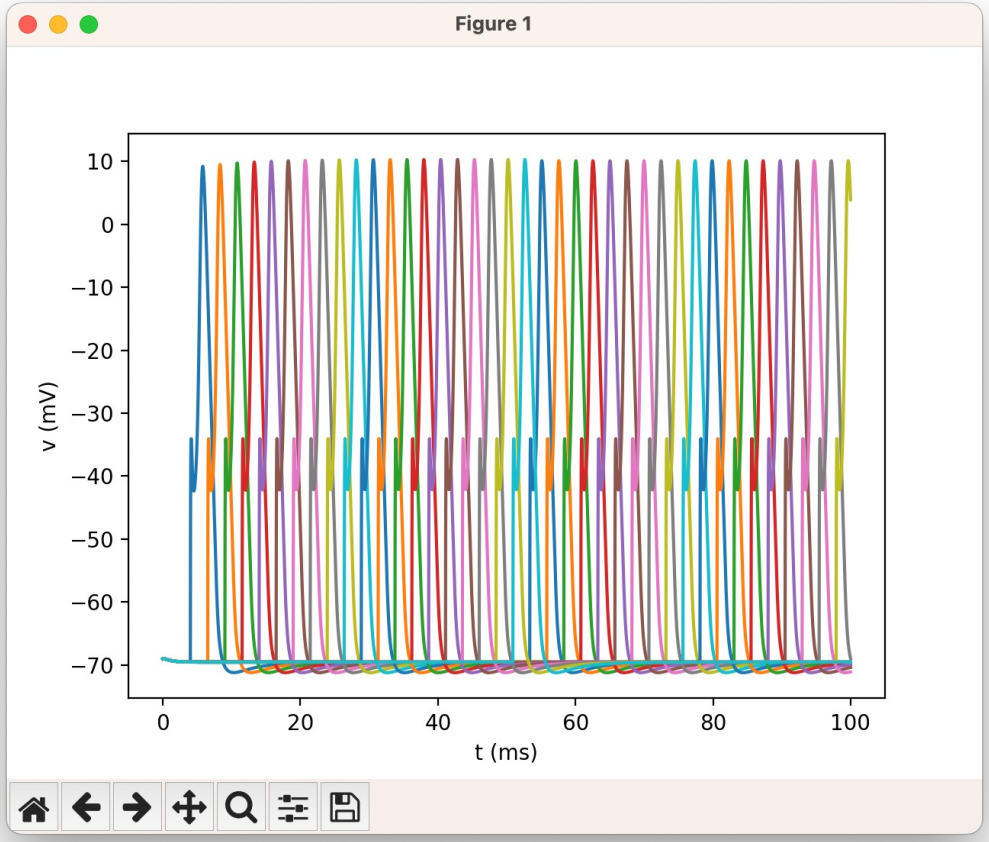
```
# drive the 0th cell
stim = h.NetStim()
stim.number = 1
stim.start = 3 * ms
ncstim = h.NetCon(stim, n.syns[0])
ncstim.delay = 1 * ms
ncstim.weight[0] = 0.5

t = h.Vector().record(h._ref_t)
v = [h.Vector().record(cell.soma[0](0.5)._ref_v)
     for cell in n.cells]

pc = h.ParallelContext()
pc.set_maxstep(10 * ms)
h.finitialize(-69 * mV)
pc.psolve(10 * ms)

for myv in v:
  pyplot.plot(t / ms, myv / mV)
pyplot.xlabel('t (ms)')
pyplot.ylabel('v (mV)')
pyplot.show()
```



How do we generalize to a ring of n neurons?



Generalizing to n cells in a ring network (100 ms)




```
class Network:
  def __init__(self, num):
    self.cells = [Pyramidal(i) for i in range(num)]
    # setup an excitable ExpSyn on each cell's
    # dendrites
    self.syns = [h.ExpSyn(cell.dend[0](0.5)) for
      cell in self.cells]
    for syn in self.syns:
      syn.e = 0 * mV
    # connect cell i to cell (i + 1) % num
    pc = h.ParallelContext()
    self.ncs = []
    for i in range(num):
      nc = pc.gid_connect(i, self.syns[(i + 1) % num])
      nc.delay = 1 * ms
      nc.weight[0] = 0.5
      self.ncs.append(nc)

n = Network(20)
```

Storing spike times

- With 20 cells, it is hard to distinguish the cells when simultaneously plotting the membrane potentials. Let's just store the spike times.
- We begin by modifying `Pyramidal._register_netcon`:

```
def _register_netcon(self):
  self.nc = h.NetCon(
    self.soma[0](0.5)._ref_v,
    None,
    sec=self.soma[0])
  pc = h.ParallelContext()
  pc.set_gid2node(self._gid, pc.id())
  pc.cell(self._gid, self.nc)
  self.spike_times = h.Vector()
  self.nc.record(self.spike_times)
```

- When the simulation is over, we can print out the spike times:

```
for i, cell in enumerate(n.cells):
  print(f'{i}: {list(cell.spike_times)}')
```

- Beginning of output:

```
0: [5.525000000100045, 54.825000000103906]
1: [8.025000000100079, 57.300000000010447]
2: [10.525000000099936, 59.775000000010503]
3: [13.000000000099796, 62.250000000105594]
```

Storing spike times in JSON

- To store spike times in JSON, we can use the following code:

```
import json

with open('output.json', 'w') as f:
  f.write(
    json.dumps({
        i: list(cell.spike_times)
        for i, cell in enumerate(n.cells)},
      indent=4))
```

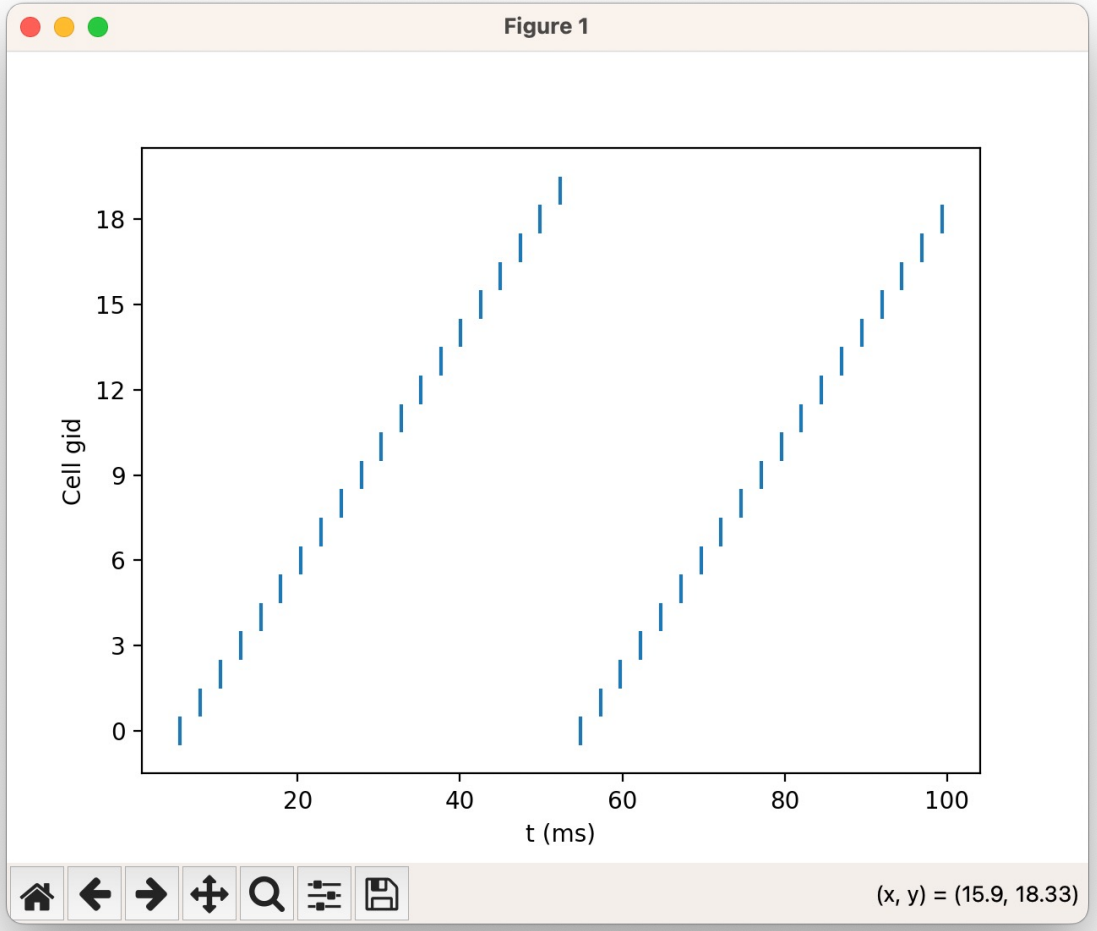
- JSON is a standard format for data interchange. Libraries are available for most programming languages.

- This creates a file `output.json` which begins:

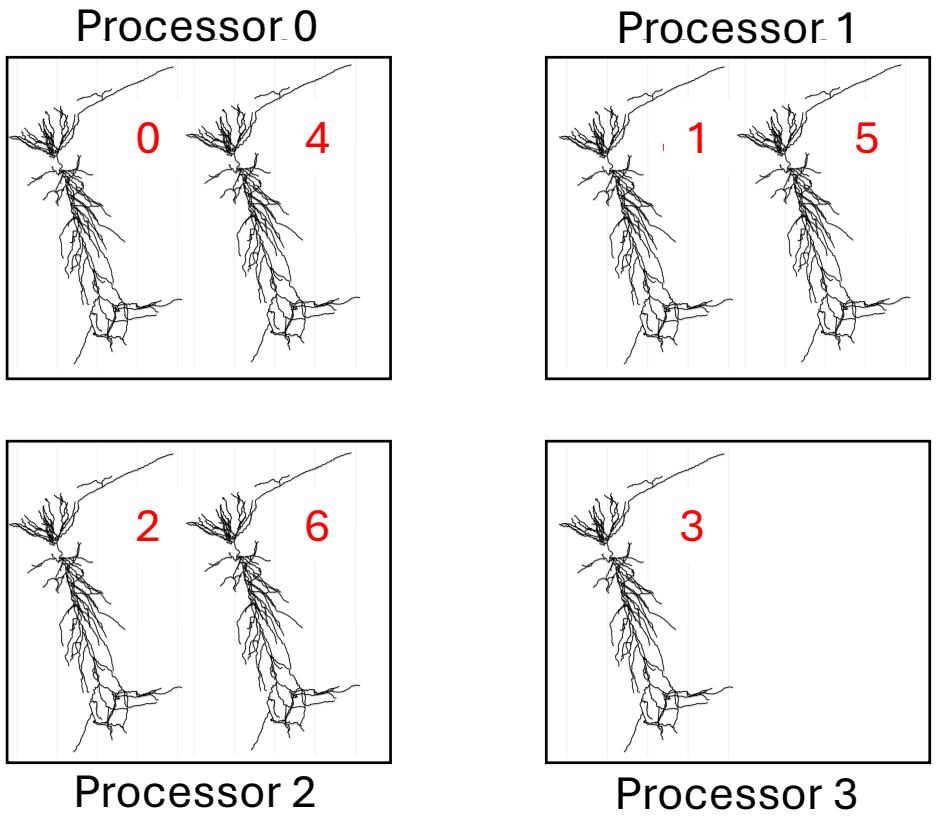
```
{
   "0": [
        5.525000000100045,
        54.825000000103906
   ],
   "1": [
        8.025000000100079,
        57.300000000010447
   ],
```

Raster plots

```
from matplotlib.ticker import MaxNLocator

for i, cell in enumerate(n.cells):
  pyplot.vlines(
    list(cell.spike_times), i - 0.5, i + 0.5)

# gids must be integers
pyplot.gca().yaxis.set_major_locator(
  MaxNLocator(integer=True))

pyplot.xlabel('t (ms)')
pyplot.ylabel('Cell gid')

pyplot.show()
```



Simple load-balancing: round-robin



An efficient way to distribute, especially if all cells similar:

```
for gid in range(pc.id(), ncell, pc.nhost()):
  pc.set_gid2node(gid, pc.id())
  ...
```

Note: The body is executed at most [`ncell` / `nhost`] times in a given process; not `ncell` times.

Advanced load-balancing: balance the *work* not the cells

- Distribute cells round-robin to all processors, instantiate them.
- Compute an estimate of the computational complexity:

```
def complexity(self):
  h.load_file('loadbal.hoc')
  lb = h.LoadBalance()
  return lb.cell_complexity(sec=self.all[0])
```

- Destroy the cells, send the gid-complexity data to node 0.
- (On node 0): distribute gids such that the next gid goes to the node with the least amount of complexity.
- Send the gids to the nodes; instantiate the cells.

Parallelizing our ring network with round-robin

Very few changes are necessary.

MPI must be initialized before we can use it:

```
h.nrnmpi_init()
```

The `Network` class only instantiates gids on the current processor.

```
class Network:
    def __init__(self, num):
        pc = h.ParallelContext()
        mygids = list(range(pc.id(), num, pc.nhost()))
        self.cells = [Pyramidal(i) for i in mygids]
        # setup an exciteable ExpSyn on each cell's dendrites
        self.syns = [h.ExpSyn(cell.dend[0](0.5)) for cell in self.cells]
        for syn in self.syns:
            syn.e = 0 * mV
        # connect cell (i - 1) % num to cell i
        self.ncs = []
        for i, syn in zip(mygids, self.syns):
            nc = pc.gid_connect((i - 1) % num, syn)
            nc.delay = 1 * ms
            nc.weight[0] = 1
            self.ncs.append(nc)
```

Parallelizing our ring network

We must modify the initial netstim to ensure it only attaches to gid 0 not to the 0th cell in each process.

```
# drive the 0th cell
if pc.gid_exists(0):
    stim = h.NetStim()
    stim.number = 1
    stim.start = 3
    ncstim = h.NetCon(stim, n.syns[0])
    ncstim.delay = 1
    ncstim.weight[0] = 1
```

Finally, we modify the write to do it on a per-processor basis:

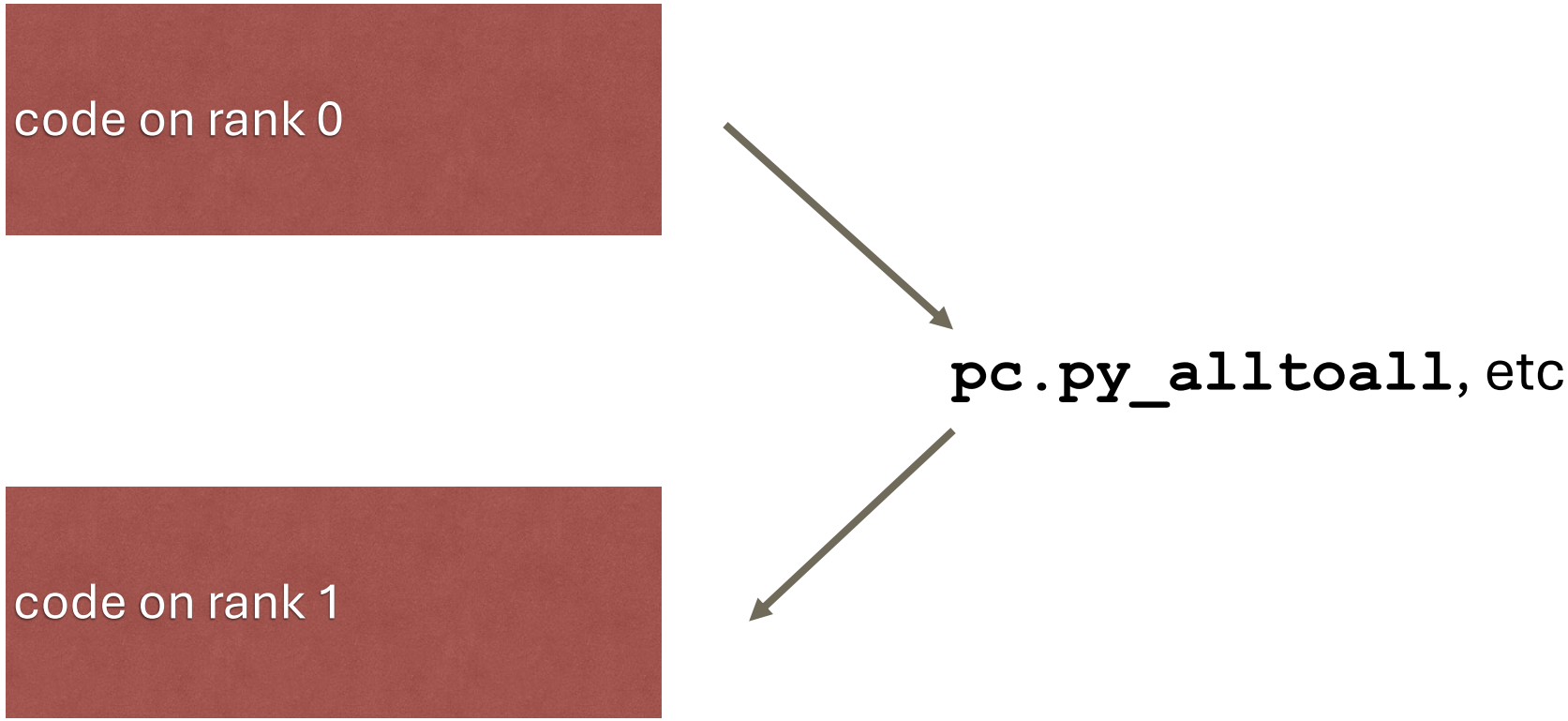
```
with open(f'output{pc.id()}.json', 'w') as f:
    f.write(json.dumps({cell._gid: list(cell.spike_times) for cell in n.cells}, indent=4))
```

Optional: use pc.py_alltoall to send all spikes to node 0

```
local_data = {cell._gid: list(cell.spike_times) for cell in n.cells}
all_data = pc.py_alltoall([local_data] + [None] * (pc.nhost() - 1))

if pc.id() == 0:
    # only do output from node 0
    import json
    combined_data = {}
    for node_data in all_data:
        combined_data.update(node_data)
    with open('output.json', 'w') as f:
        f.write(json.dumps(combined_data, indent=4))
```

Behind the scenes

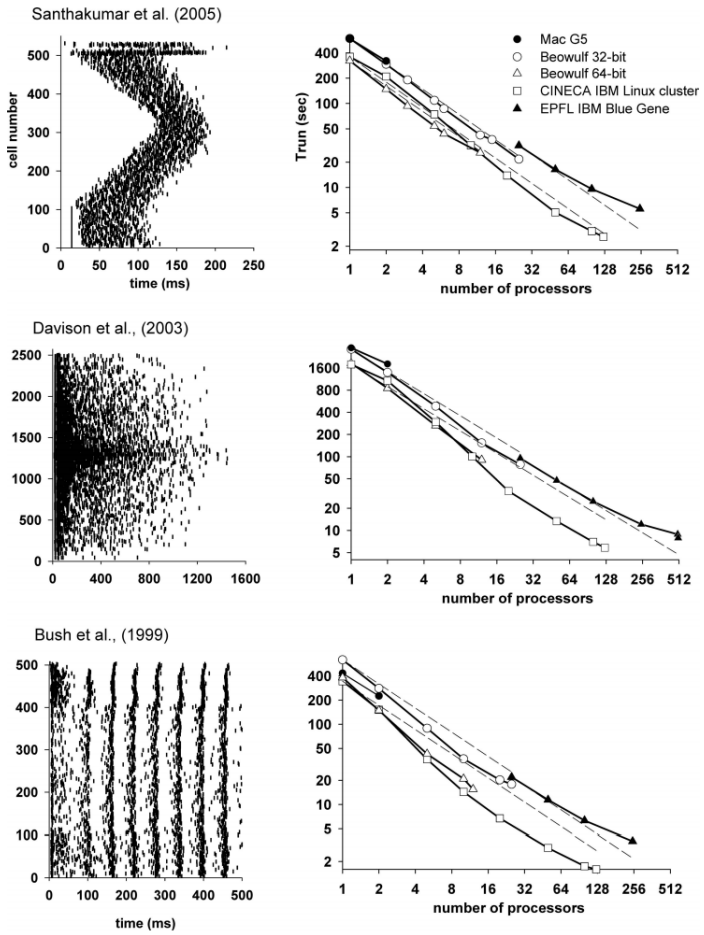
Automatically exchange any picklable data type, but there is overhead.

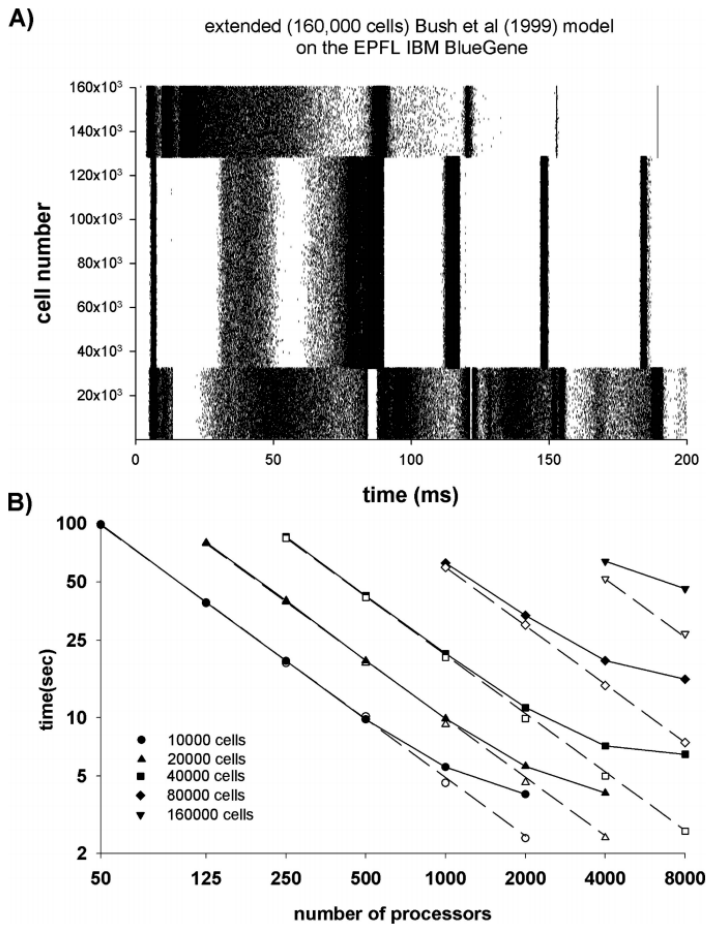


Behind the scenes

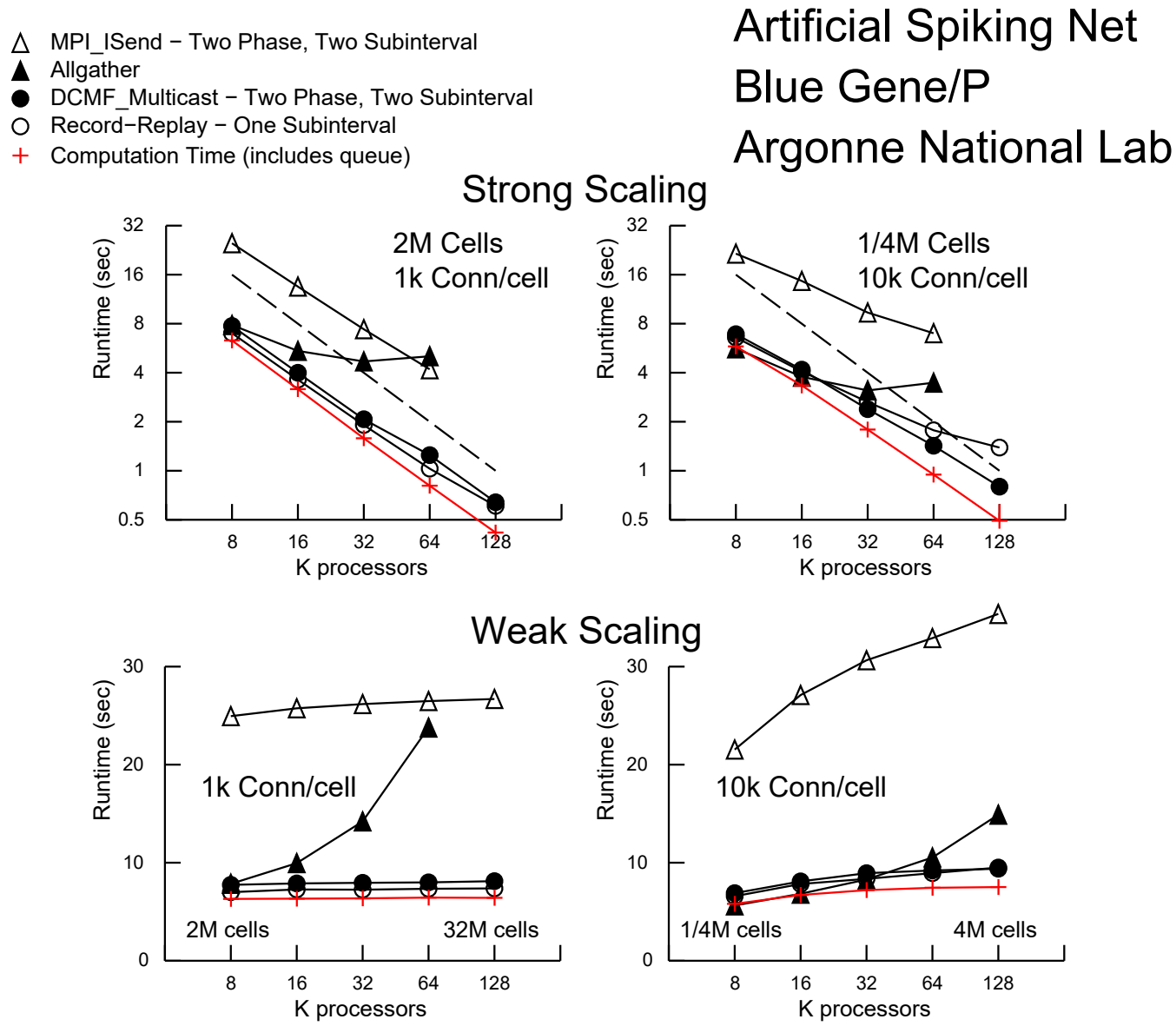
Automatically exchange any picklable data type, but there is overhead.

Performance: MPI scaling





Performance: Spike exchange strategies



Tip

- In large network models, we expect an average of more than one spike per 0.025 ms (NEURON's default timestep).

- If we use a variable step solver, we will capture those times correctly, but this will require restarting the integrator frequently.

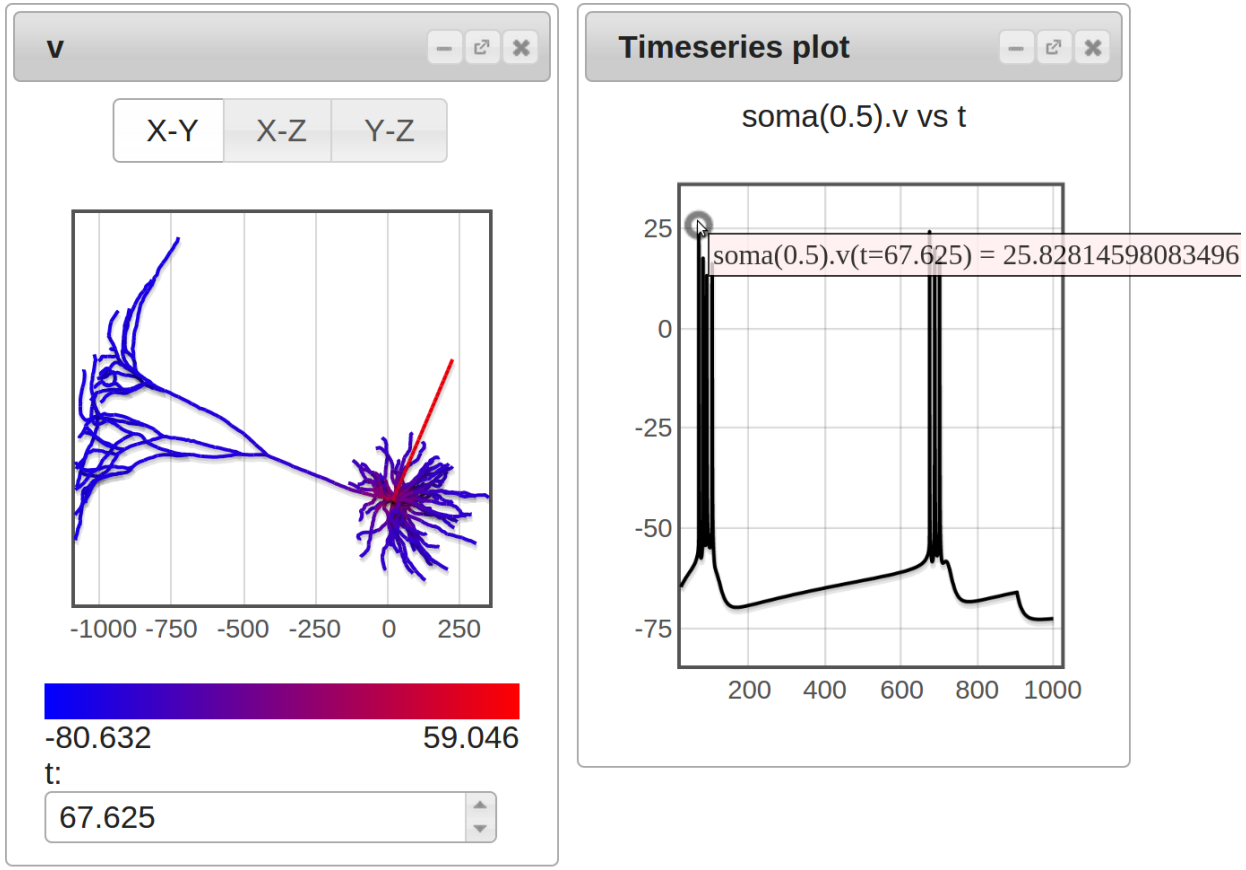
- In practice, it is usually better to use a fixed stoep solver for network models.

Question

- Suppose we ran a large network simulation and stored the spike times.
- Suppose further we now realize we want to know the time series of the m variable in the center of the soma of cell 5.
- Do we have to modify our code to store that variable and rerun the entire simulation?

Tip: Store synaptic events and recreate cells as needed

initial conditions
+
synaptic events
→ neuron dynamics



Using spike data to recreate a variable of interest

We will need `vecevent.mod`. If you have NEURON, this file should be on your computer somewhere. Alternatively, you can download it from:

https://github.com/neuronsimulator/nrn/blob/master/share/examples/nrniv/netcon/vecevent.mod

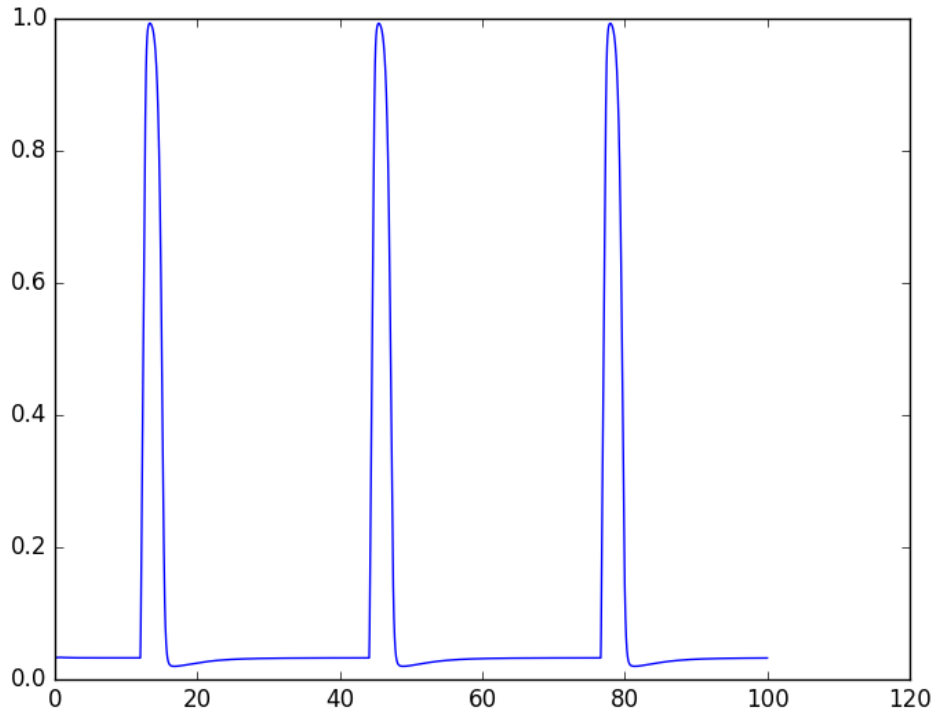
Using spike data to recreate a variable of interest

```
import json
from neuron import h
from neuron.units import ms, mV
from PyNeuronToolbox import morphology
from matplotlib import pyplot
h.load_file('stdrun.hoc')
num_cells = 20

# class Pyramidal as before

# read spike times
with open('output.json') as f:
    spike_times_by_cell = json.load(f)
```

(continued)

Using spike data to recreate a variable of interest

```
def get_m(gid):
    p = Pyramidal(gid)
    # recreate synaptic inputs (here, only one; you may have multiple)
    precell = (gid - 1) % num_cells
    vs = h.VecStim()
    spike_vec = h.Vector(spike_times_by_cell[str(precell)])
    vs.play(spike_vec)
    syn = h.ExpSyn(p.dend[0](0.5))
    nc = h.NetCon(vs, syn)
    nc.delay = 1 * ms
    nc.weight[0] = 1
    # setup recording
    t = h.Vector().record(h._ref_t)
    m = h.Vector().record(p.soma[0](0.5)._ref_m_hh)
    # do run
    pc = h.ParallelContext()
    pc.set_maxstep(10 * ms)
    h.finitialize(-69 * mV)
    pc.psolve(100 * ms)
    return t, m

t, m = get_m(5)
pyplot.plot(t, m)
pyplot.show()
```



Pseudo-randomness

Ensuring reproducibility while sampling from distributions.

When would we want randomness?

- Stochastic input (e.g., spikes arriving via a Poisson process).
- Stochastic channel gating.
- Random channel parameters.
- Random morphology.
- Random network connections.

Guiding principles

- Randomness should be **reproducible**. e.g., use random seeds.
- Randomness should be **independent of the number of processors**.

Philox/Random123

- Random123 is a **cryptographic quality** generator, suitable for managing separate **independent**, **reproducible**, and **restartable** streams; the algorithm used is called Philox in numpy and in the paper:

Parallel Random Numbers: As Easy as 1, 2, 3

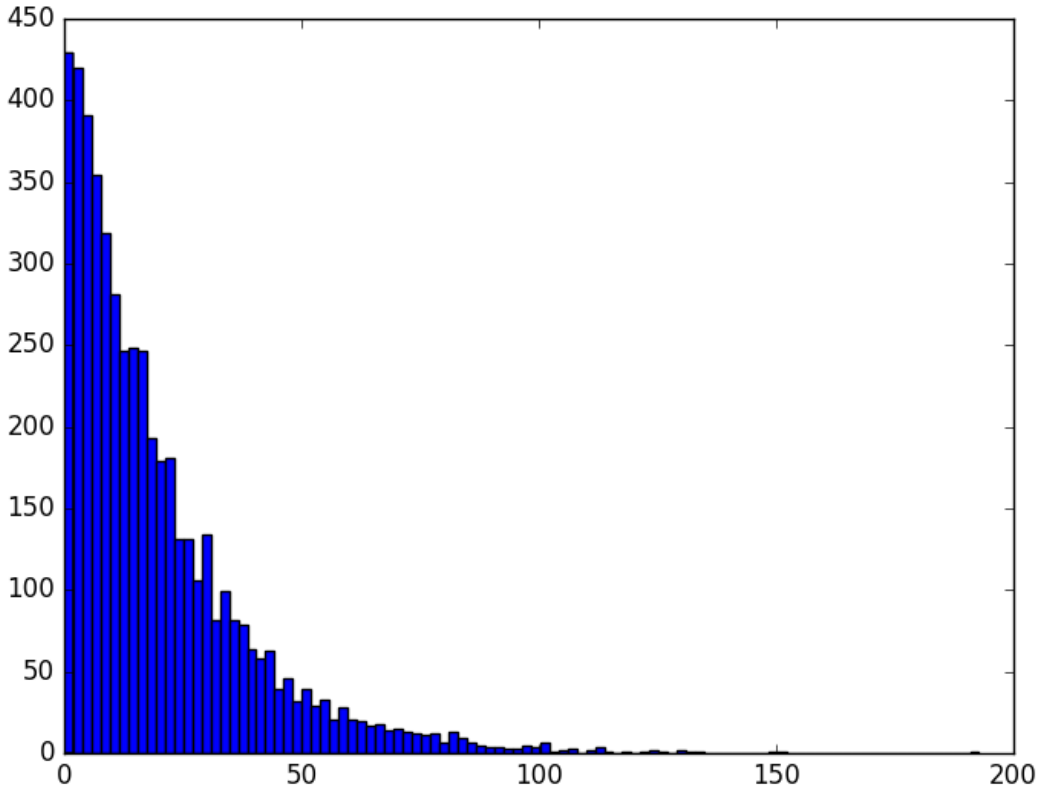
John K. Salmon,* Mark A. Moraes, Ron O. Dror, and David E. Shaw*†
D. E. Shaw Research, New York, NY 10036, USA

```
from neuron import h
r = h.Random()
r.Random123(seed, id1, id2)
print(r.uniform(0,1)) # or binomial, or geometric, or lognormal, or negexp, or ...
```

- A reasonable choice for `id1` is the gid of the cell; a reasonable choice for `id2` is the section index in `cell.all` or the synapse index.

Example: random synaptic inputs ("noise")

```
from neuron import h, gui
import numpy
from matplotlib import pyplot

r = h.Random()
r.negexp(1)
r.Random123(1, 2, 3)
ns = h.NetStim()
ns.noiseFromRandom(r)
ns.number = 1e20
ns.start = 5
ns.interval = 20
ns.noise = 1

spike_times = h.Vector()
nc = h.NetCon(ns, None)
nc.record(spike_times)
pc = h.ParallelContext()
pc.set_maxstep(10)
h.finitialize(-65)
pc.psolve(100000)

spike_times = spike_times.as_numpy()
isis = spike_times[1:] - spike_times[:-1]
print(numpy.mean(isis), numpy.std(isis))
pyplot.hist(isis, 100)
pyplot.show()
```



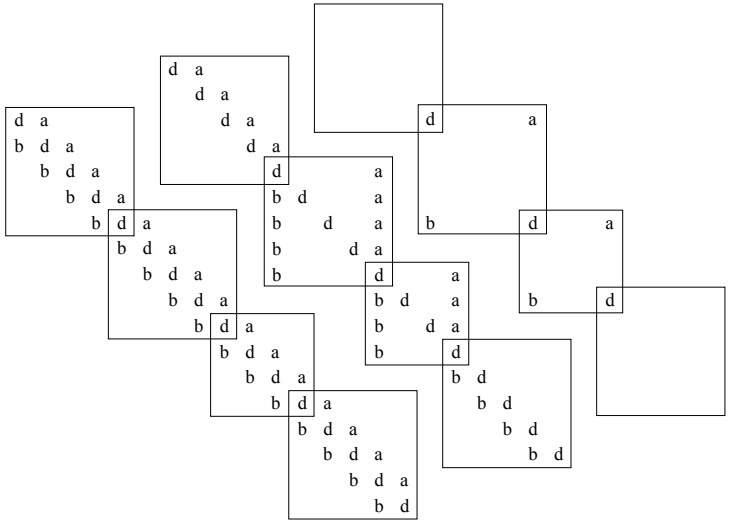
Example: random distribution of leak conductance

```
import sys
seed = float(sys.argv[1])

class Pyramidal:
    def __init__(self, gid):
        self._gid = gid
        self._setup_morphology()
        self._discretize()
        self.random_stream = [self.new_random_stream(i) for i in range(len(self.all))]
        self._add_channels()
        self._register_netcon()
    def new_random_stream(self, id2):
        r = h.Random()
        r.Random123(seed, self._gid, id2)
        return r
    def _add_channels(self):
        for sec in self.soma:
            sec.insert(h.hh)
        for sec, stream in zip(self.all, self.random_stream):
            sec.insert(h.pas)
            for seg in sec:
                seg.pas.g = max(0, stream.normal(1e-3, 8e-6))
    # the rest stays as before
```

If you try this with seed 1, you'll notice the raster looks significantly different. Why?

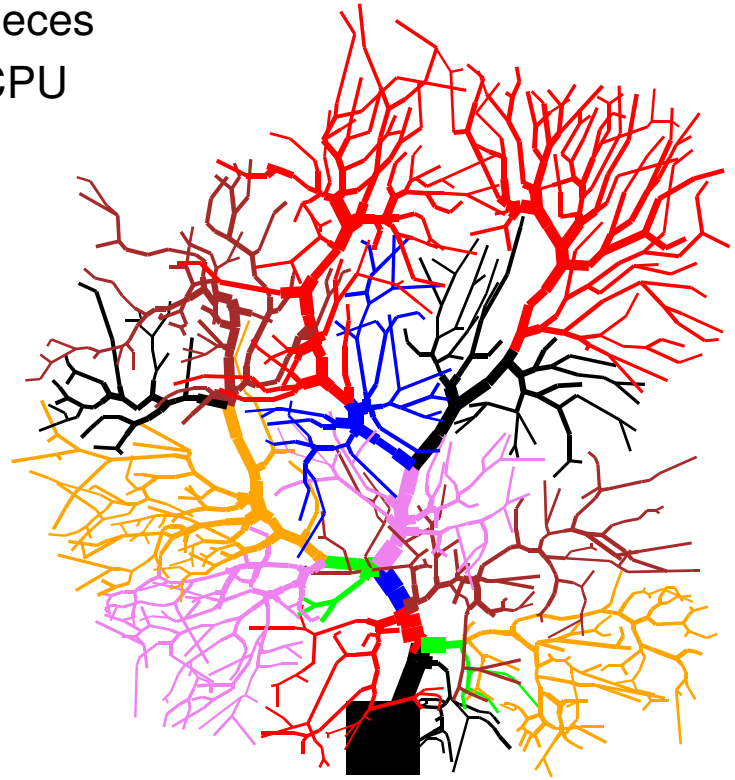
Multisplit

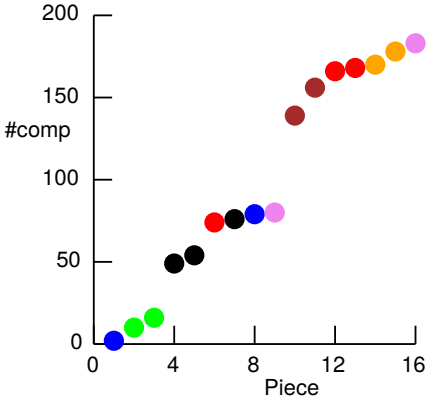
Improve load balancing with multisplit

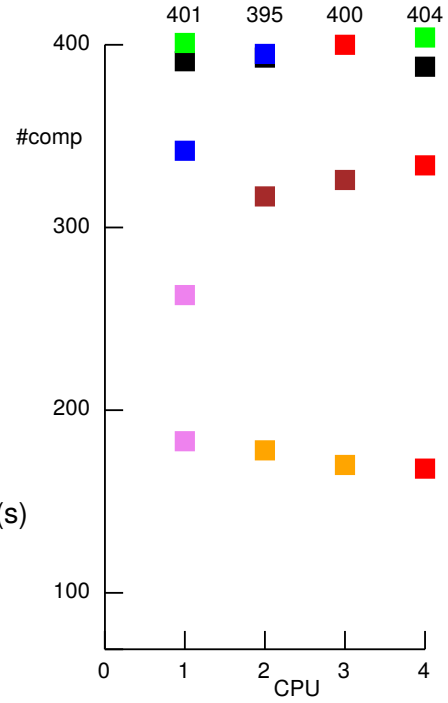


Multisplit algorithm described in Hines et al 2008.
DOI: 10.1007/s10827-008-0087-5

16 Pieces
4 CPU







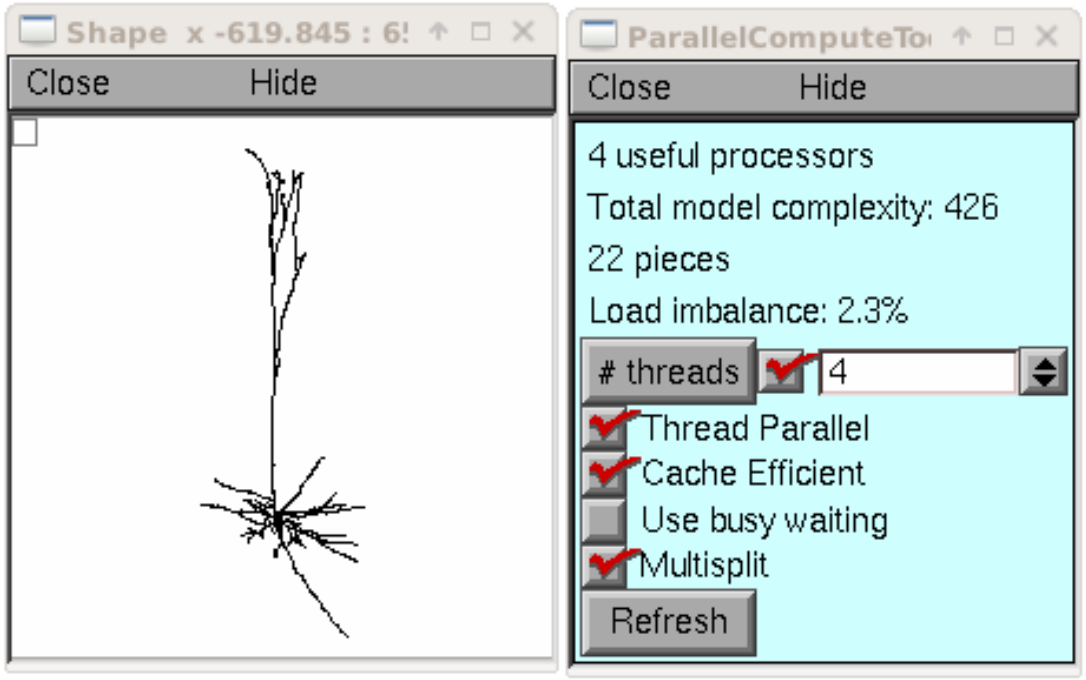
Time (s)

CPU	Computation	Exchange
0	13.82	0.56
1	13.35	1.03
2	13.47	0.90
3	13.56	0.82

	Runtime(s)
16 pieces, 1 cpu	55.0
wholecell, 1 cpu	56.2
16 pieces, 4 cpu	14.4

Using multisplit (threads)

When not using MPI, enabling thread-based multisplit is as easy as clicking a checkbox:



Using multisplit (MPI)

For process-based multisplit (with MPI), use `pc.multisplit` to declare split nodes:

```
pc.multisplit(seg, subtreeid)
```

After all split nodes are declared, **every** process must execute:

```
pc.multisplit()
```

If created, destroy any parts of the cell that do not belong on the processor.

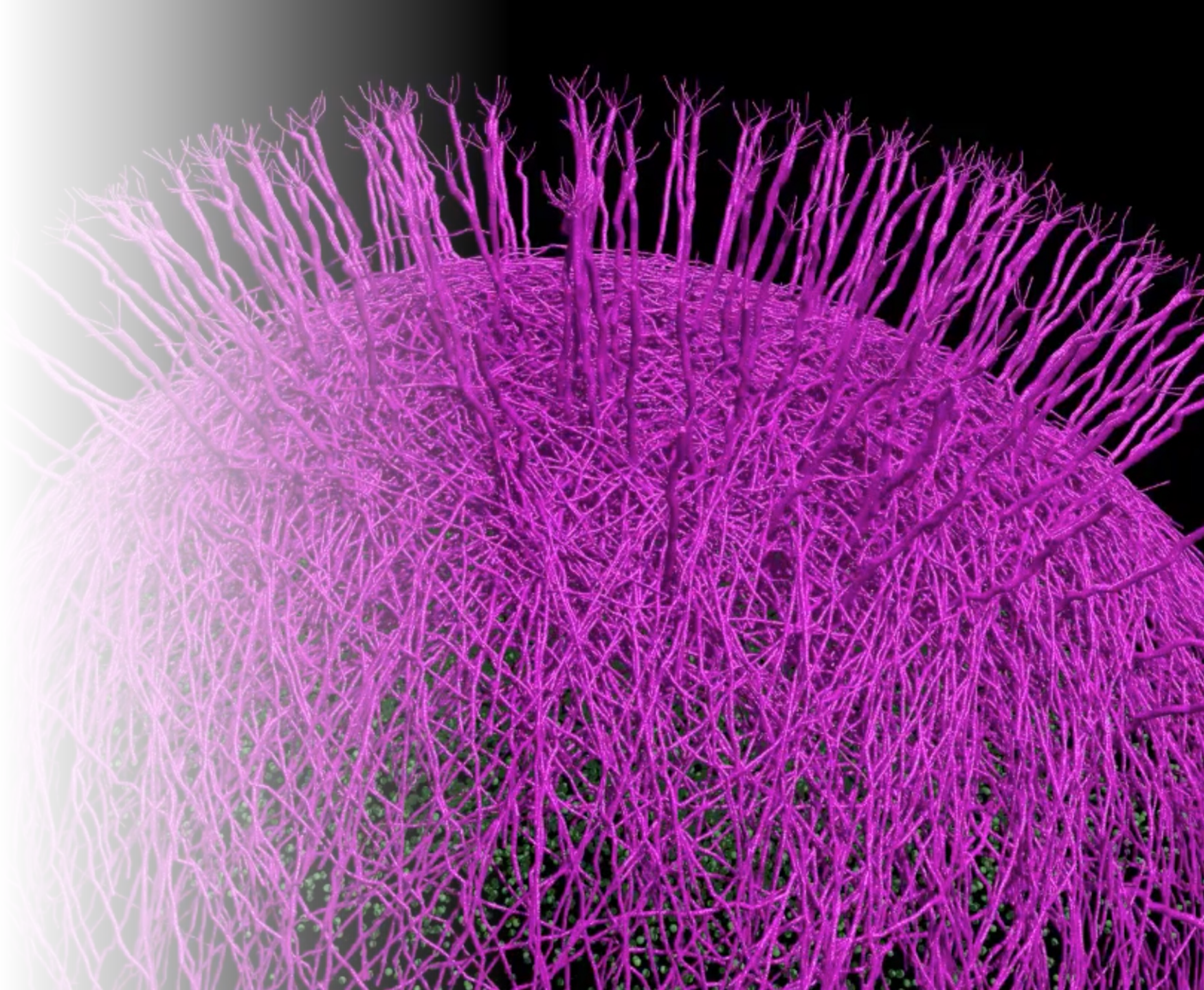
Rules:

- Each subtree can have at most two split nodes.
- Does not support variable step, linear mechanisms, extracellular, or reaction-diffusion.
- `h.distance` cannot compute path distances that cross a split node.

Tip: For load balancing, it is sometimes convenient to split cells into more pieces than processes.

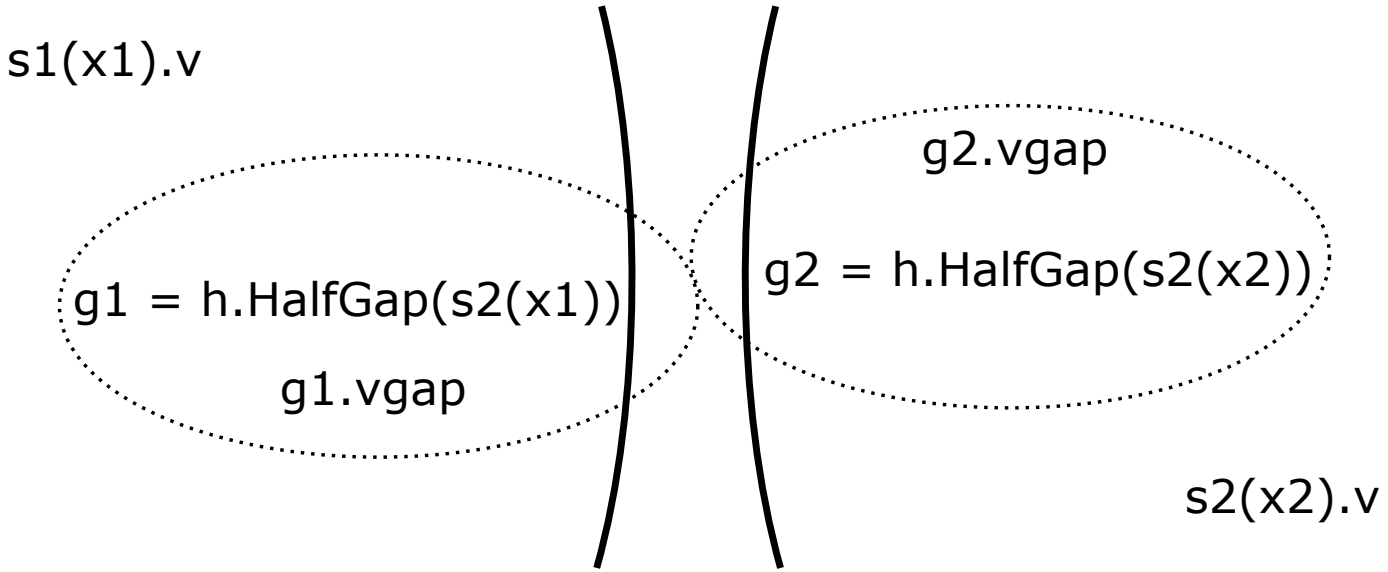
Example: Migliore et al., 2014

- Migliore et al 2014 used multisplit to improve load balancing on a model of the olfactory bulb. modeldb.science/151681
- See, in particular, the file `multisplit distrib.py`.



Gap junctions

Continuous voltage exchange

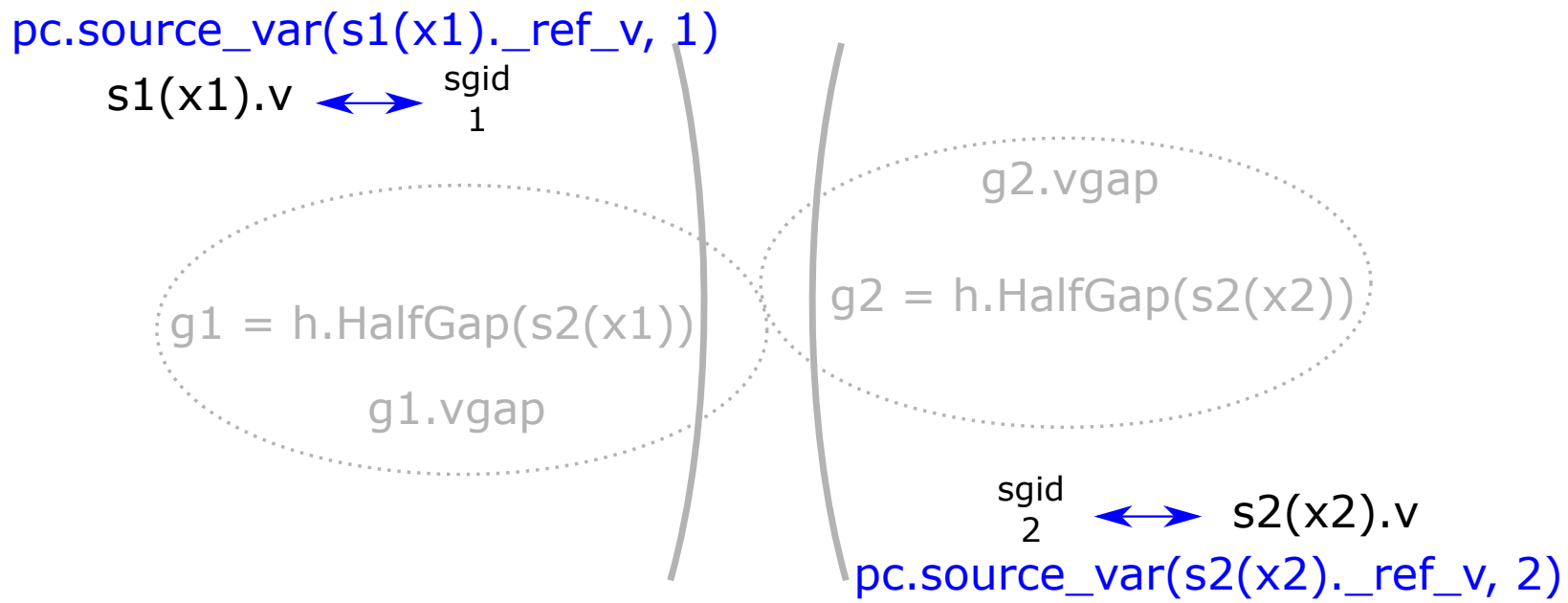


HalfGap.mod

```
NEURON  {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap - v) / r }
```

pc.source_var to declare source sgid

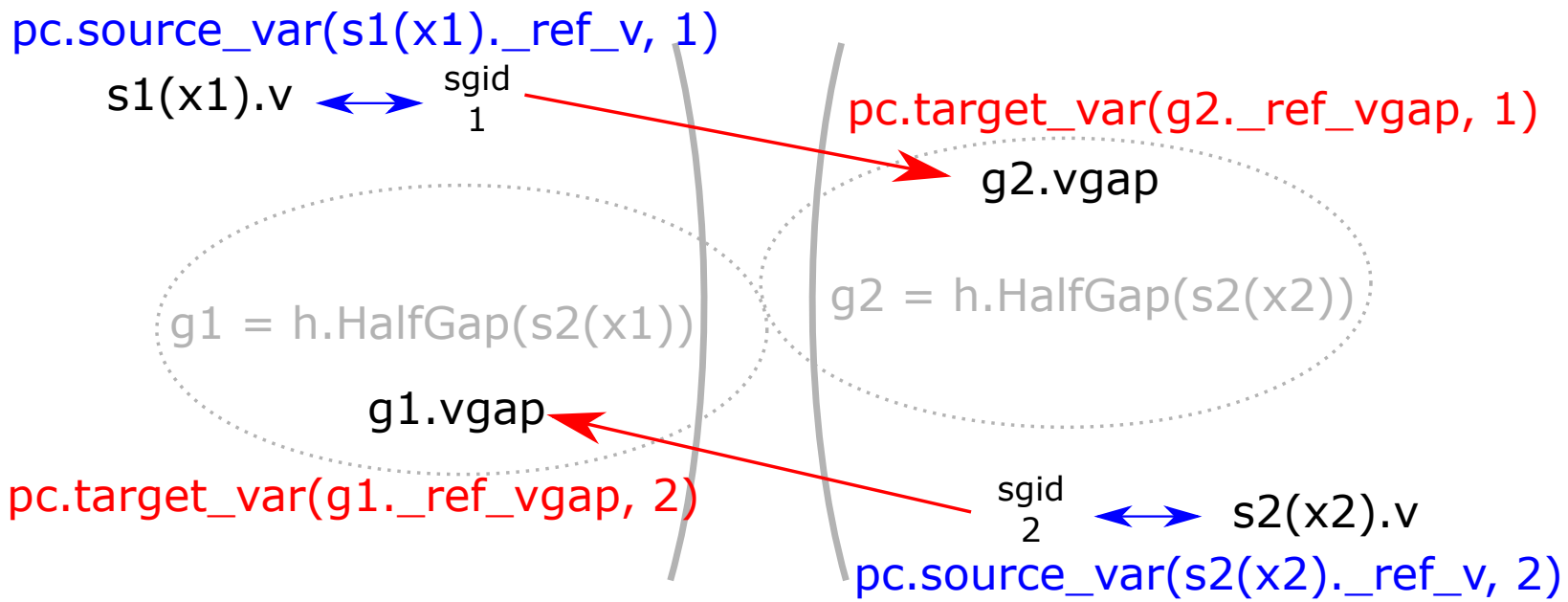


HalfGap.mod

```
NEURON  {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap - v) / r }
```

pc.target_var to declare target connection

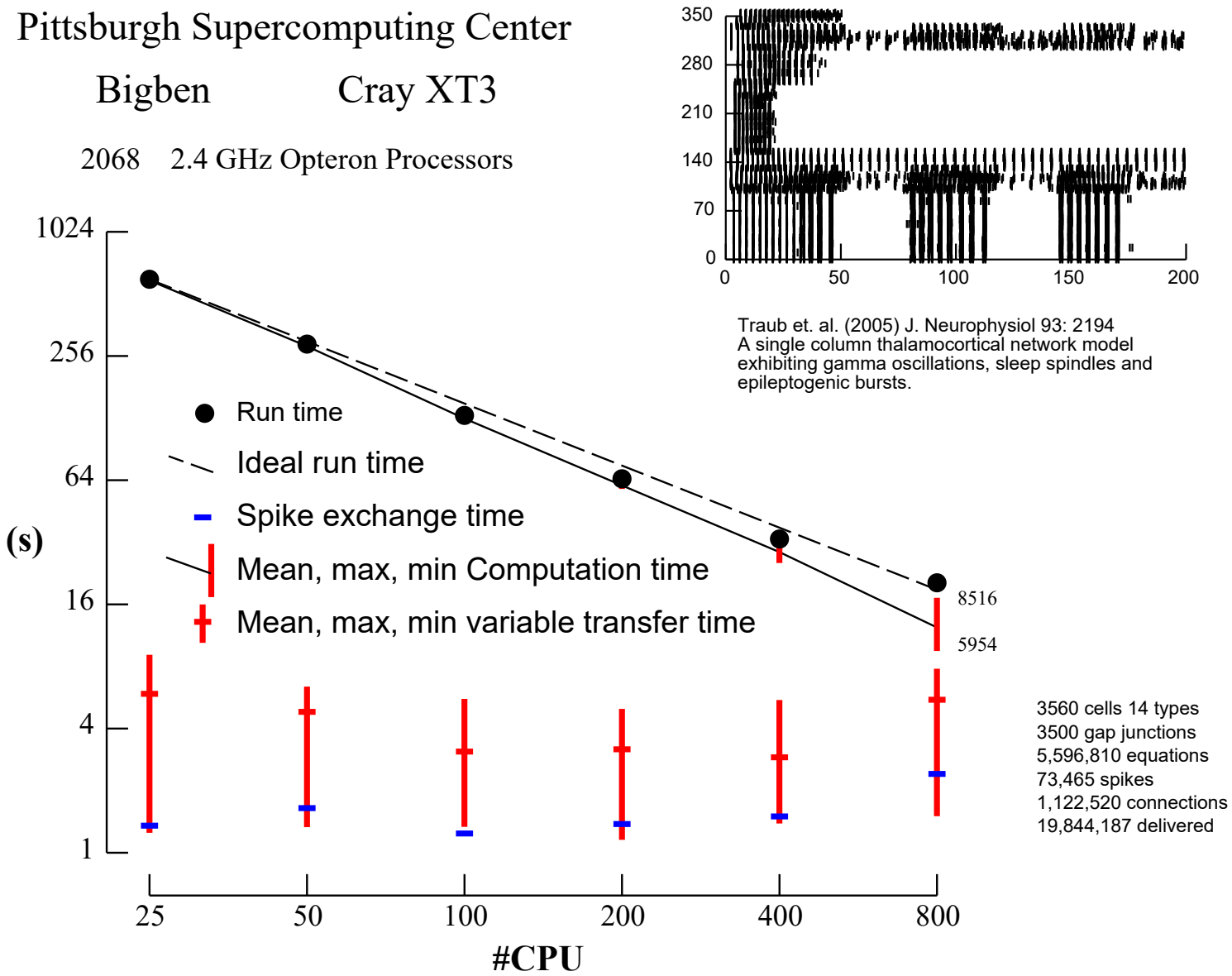


HalfGap.mod

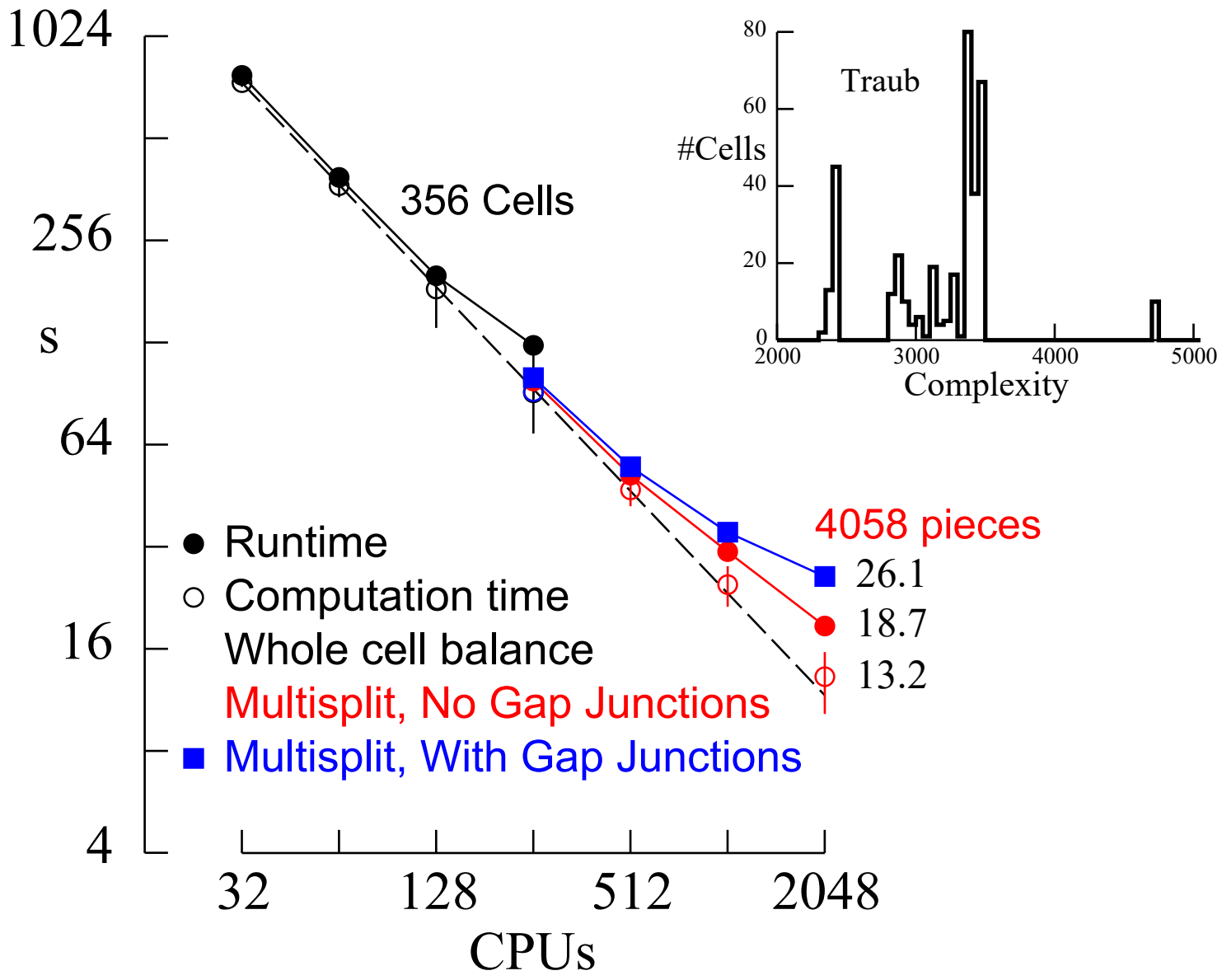
```
NEURON  {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap - v) / r }
```

Performance: Traub model



Performance: Traub model with multisplit



Finally: Subworlds

Use `pc.subworlds` to combine parallel simulation with parallel bulletin-board based parameter search.

```
from neuron import h
h.nrnmpi_init()
pc = h.ParallelContext()
pc.subworlds(2)

from model import runmodel

pc.runworker()

for ncell in range(5, 10):
    pc.submit(runmodel, ncell, 1, 100)

while (pc.working()):
    print(pc.pyret())

pc.done()
h.quit()
```

Note: Unless memory on a single node is a limiting factor, you will likely want either 1 subworld (everything) or `pc.nhost()` subworlds. In the first case, there is no need to use subworlds since simulations are run one at a time; in the other extreme, there is also no need since each simulation runs on a single processor.