Building, Running, and Visualizing Parallel NEURON Models

Robert A. McDougal

Yale School of Medicine

11 August 2018

Why use parallel computation?

Four reasons:

- Get the results for a simulation in less real time.
- Run a larger simulation in the same amount of time.
- Run more simulations (e.g. parameter sweeps).
- Run models needing more memory than is available on one machine.

What are the downsides?

Parallel models introduce:

- Greater programming complexity.
- New kinds of bugs.

You have to decide if the time spent parallelizing your model can be recovered.

Other considerations

The 16384 core EPFL IBM BlueGene/P can theoretically do as many calculations in 1 hour at 850 MHz as a 3 GHz desktop computer can do in 6 months.

Building a parallelizable model typically requires little extra effort from building a serial model; converting a serial model to a parallel model is often more difficult.

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. NEURON supports this natively with bulletin boards; Calin-Jageman and Katz (2006) developed a screen saver solution.

Distributing networks across processors

Cells can communicate by

- logical spike events with significant axonal, 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 1, 2, or 3 of 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.

For parallel networks: cell classes should have a gid

In addition, it will be convenient to specify morphology in a dedicated method, and add a `__repr__` method to identify the object.

```
from neuron import h, gui
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 'Pyramidal[%d]' % self._gid
```

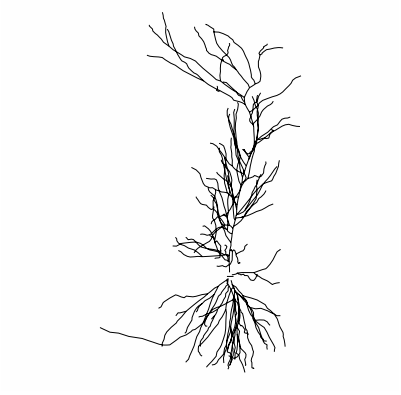
Here, the `gid` should be a globally unique identifying integer. We do not use class variables to generate the integer automatically because: (1) the numbers should not repeat between different processors, and (2) we may wish to recreate a single specific cell instead of the entire network.

Working with multiple cells

Suppose `Pyramidal` is defined as before and we create several copies:

```
mypyrs = [Pyramidal(i) for i in range(10)]
```

We then view these in a shape plot:



Where are the other 9 cells?

Working with multiple cells

To can create a method to reposition a cell and call it from `__init__`:

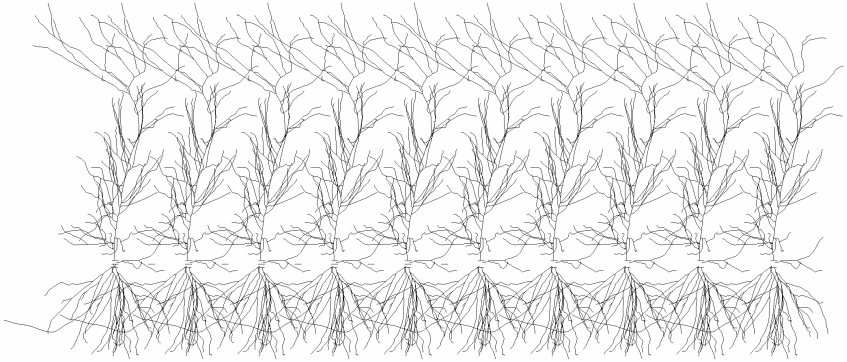
```
class Pyramidal:
  def _shift(self, x, y, z):
    for sec in self.all:
      n = sec.n3d()
      xs = [sec.x3d(i) for i in range(n)]
      ys = [sec.y3d(i) for i in range(n)]
      zs = [sec.z3d(i) for i in range(n)]
      ds = [sec.diam3d(i) for i in range(n)]
      i = 0
      for a, b, c, d in zip(xs, ys, zs, ds):
        sec.pt3dchange(i, a + x, b + y, c + z, d)
        i += 1
```

```
def __init__(self, gid, x, y, z):
  self._gid = gid
  self._setup_morphology()
  self._shift(x, y, z)

def _setup_morphology(self):
  cell = h.Import3d_SWC_read()
  cell.input('c91662.swc')
  i3d = h.Import3d_GUI(cell, 0)
  i3d.instantiate(self)
```

Now if we create ten, while specifying offsets,

```
mypyrs = [Pyramidal(i, i * 100, 0, 0) for i in range(10)]
```

The PlotShape will show all the cells separately:



Does position matter?

Sometimes.

Position matters with:

- Connections based on proximity of axon to dendrite.
- Connections based on cell-to-cell proximity.
- Extracellular diffusion.
- Communicating about your model to other humans.

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 'p[%d]' % self._gid
    def _discretize(self, max_seg_length=20):
        for sec in self.all:
            sec.nseg = 1 + 2 * int(sec.L / max_seg_length)
    def _add_channels(self):
        for sec in self.soma:
            sec.insert('hh')
        for sec in self.all:
            sec.insert('pas')
            for seg in sec:
                seg.pas.g = 0.001
```

Remember: you typically want to have an odd number of segments so there is a node at the middle.

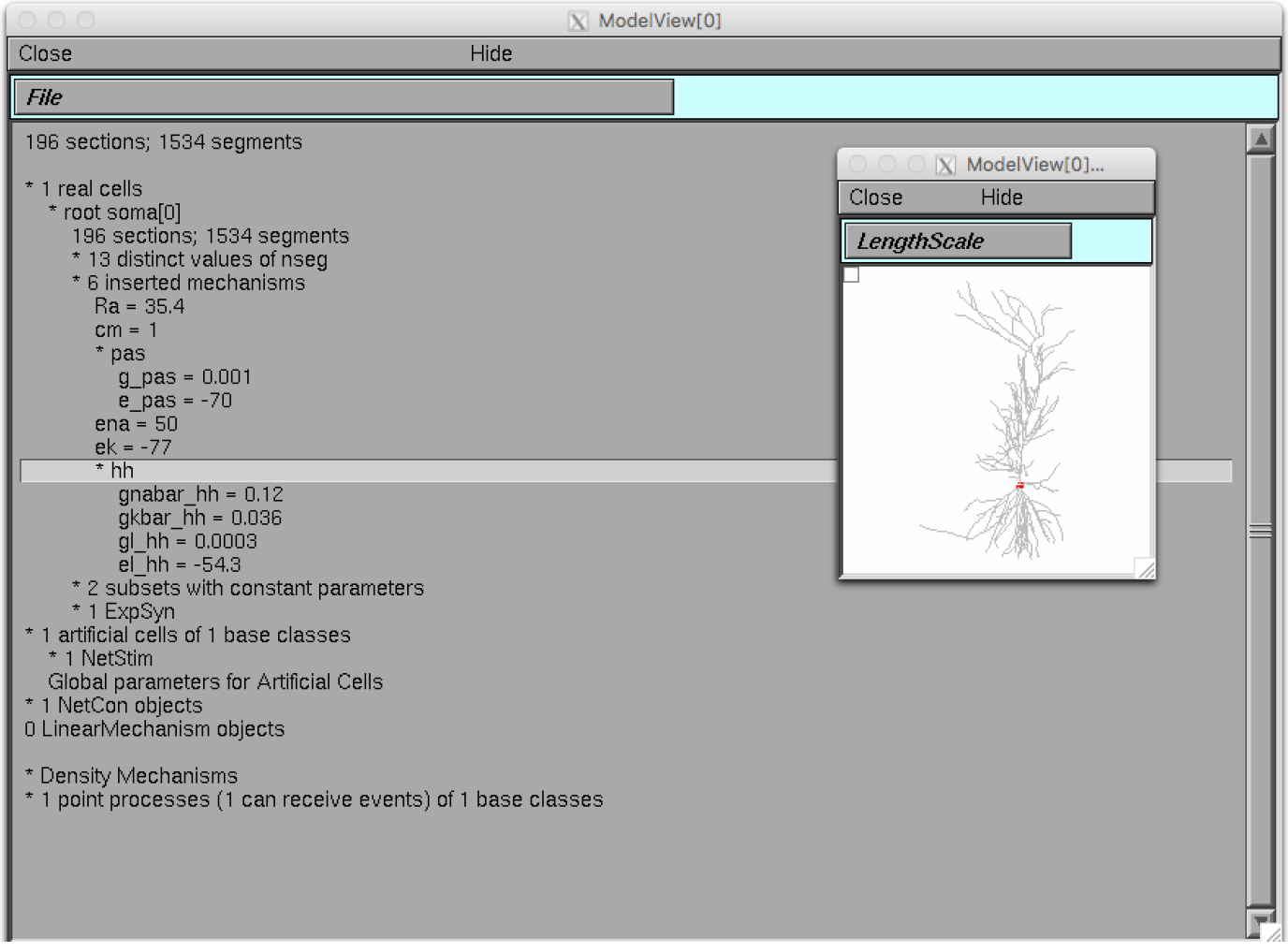
When refining a mesh, multiply by an odd number to preserve old nodes.

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

An alternative discretization strategy is to use the d_lambda rule:

```
def _discretize(self):
    h.load_file('stdlib.hoc')
    for sec in self.all:
        sec.nseg = int((sec.L/(0.1*h.lambda_f(100)) + .9)/2.)*2 + 1
```

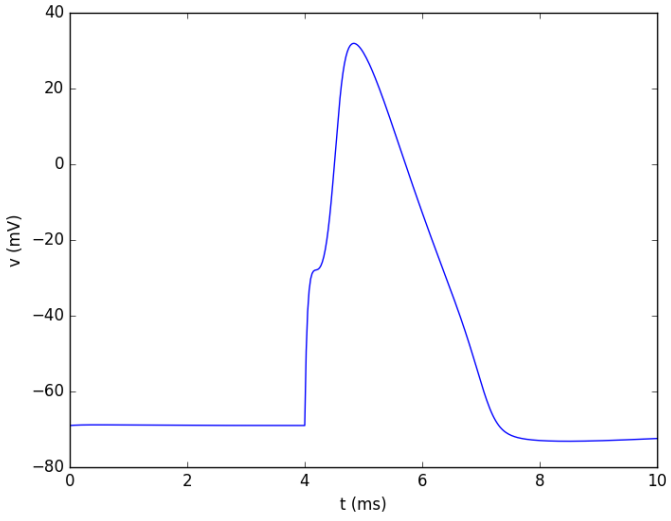
Examine for errors: Tools → ModelView



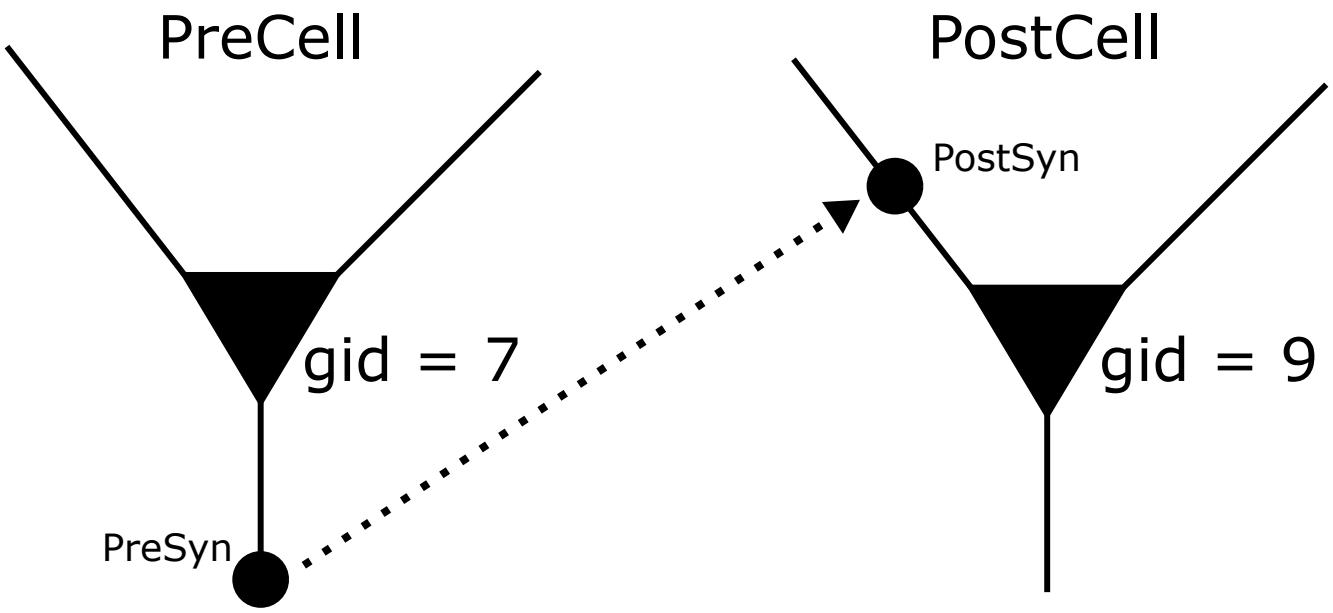
New way to run via h.ParallelContext()

```
from neuron import h
from PyNeuronToolbox import morphology
from matplotlib import pyplot

h.load_file('stdrun.hoc')

# class Pyramidal defined as before

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 = 1
ncstim.weight[0] = 1

t = h.Vector()
t.record(h._ref_t)
v = h.Vector()
v.record(myPyramidal.soma[0](0.5)._ref_v)
```

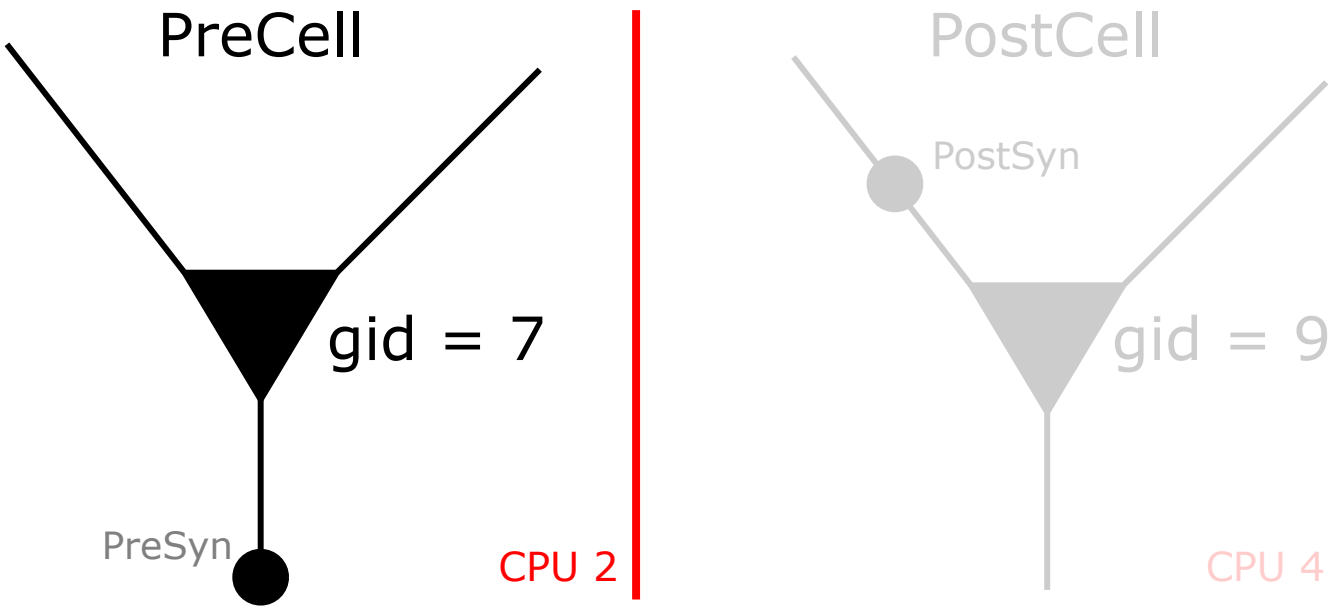
```
pc = h.ParallelContext()
pc.set_maxstep(10)
h.v_init = -69
h.stdinit()
pc.psolve(10)

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



Building synapses



Configuring the presynaptic connection site



Create cell only where the gid exists:

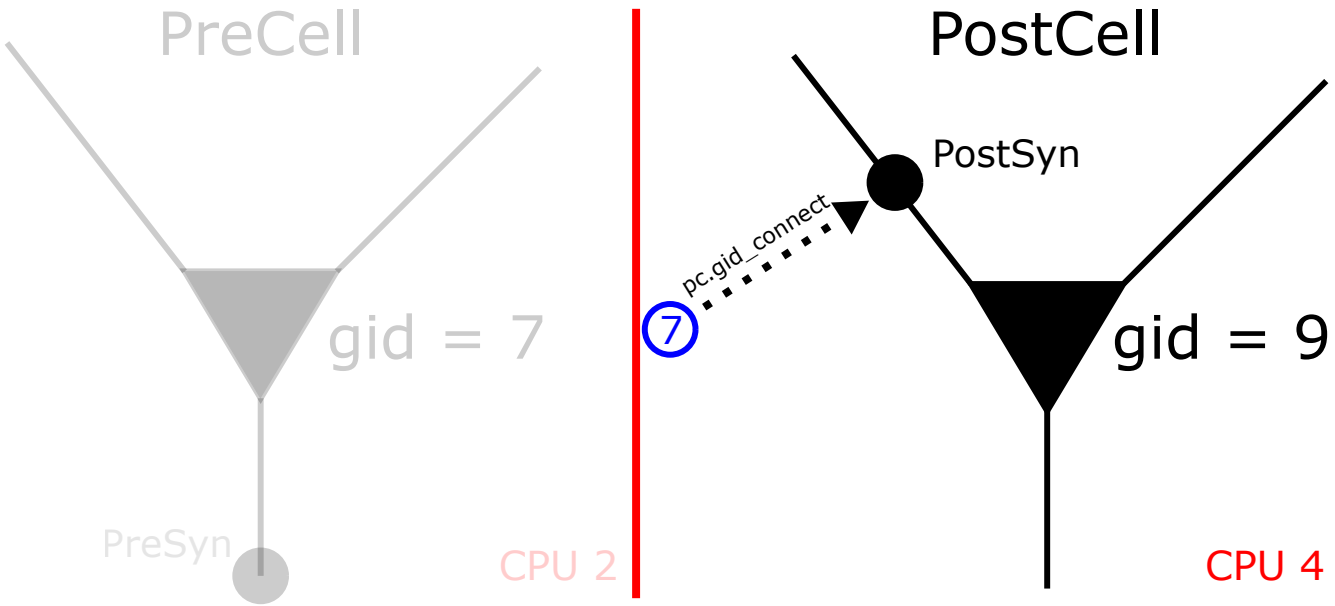
```
if pc.gid_exists(7):
    PreCell = Cell()
```

Associate gid with spike source:

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

PreSyn here is a **pointer**, e.g. `PreCell.soma(0.5)._ref_v`

Configuring the postsynaptic connection site

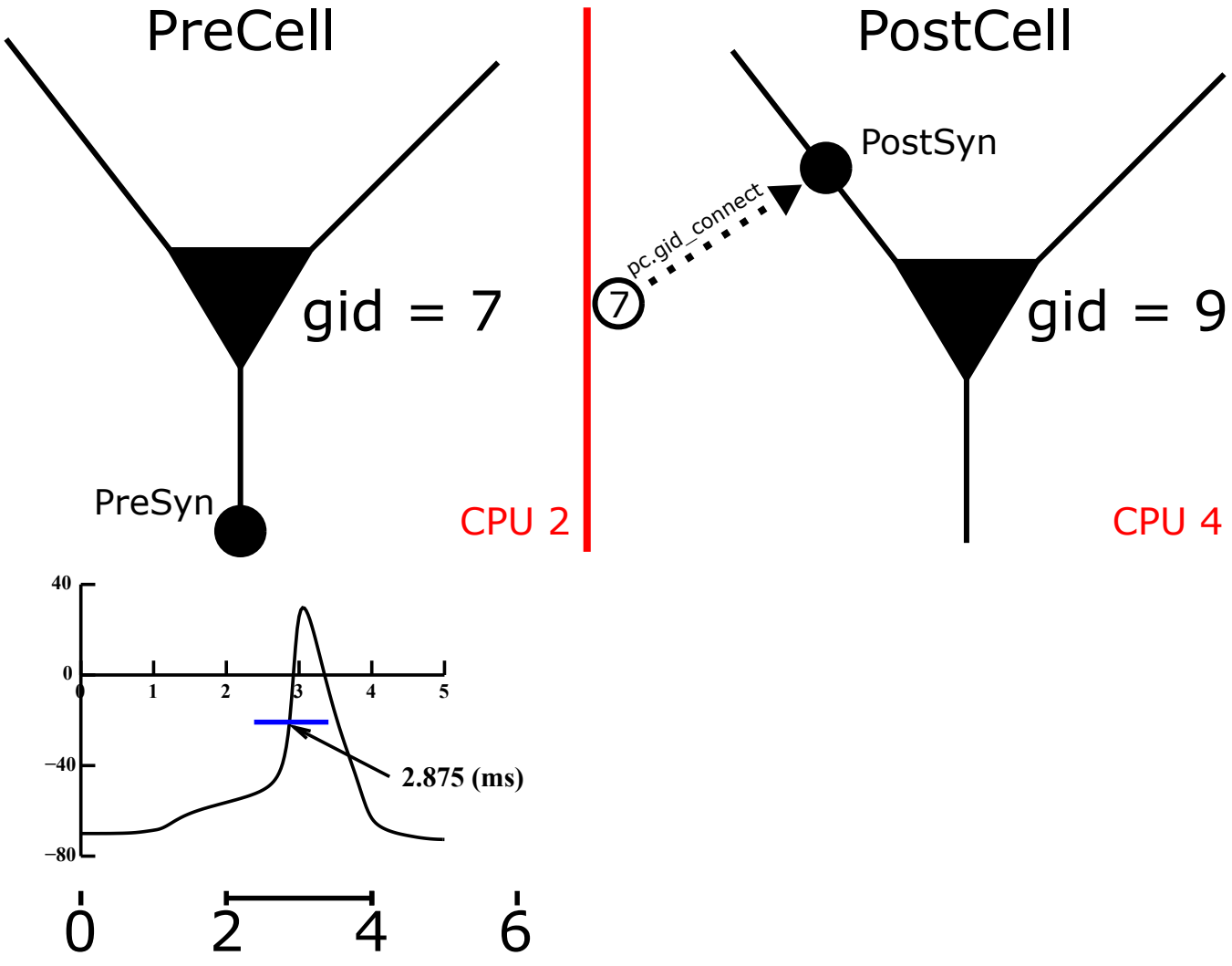


Create NetCon on node where target exists:

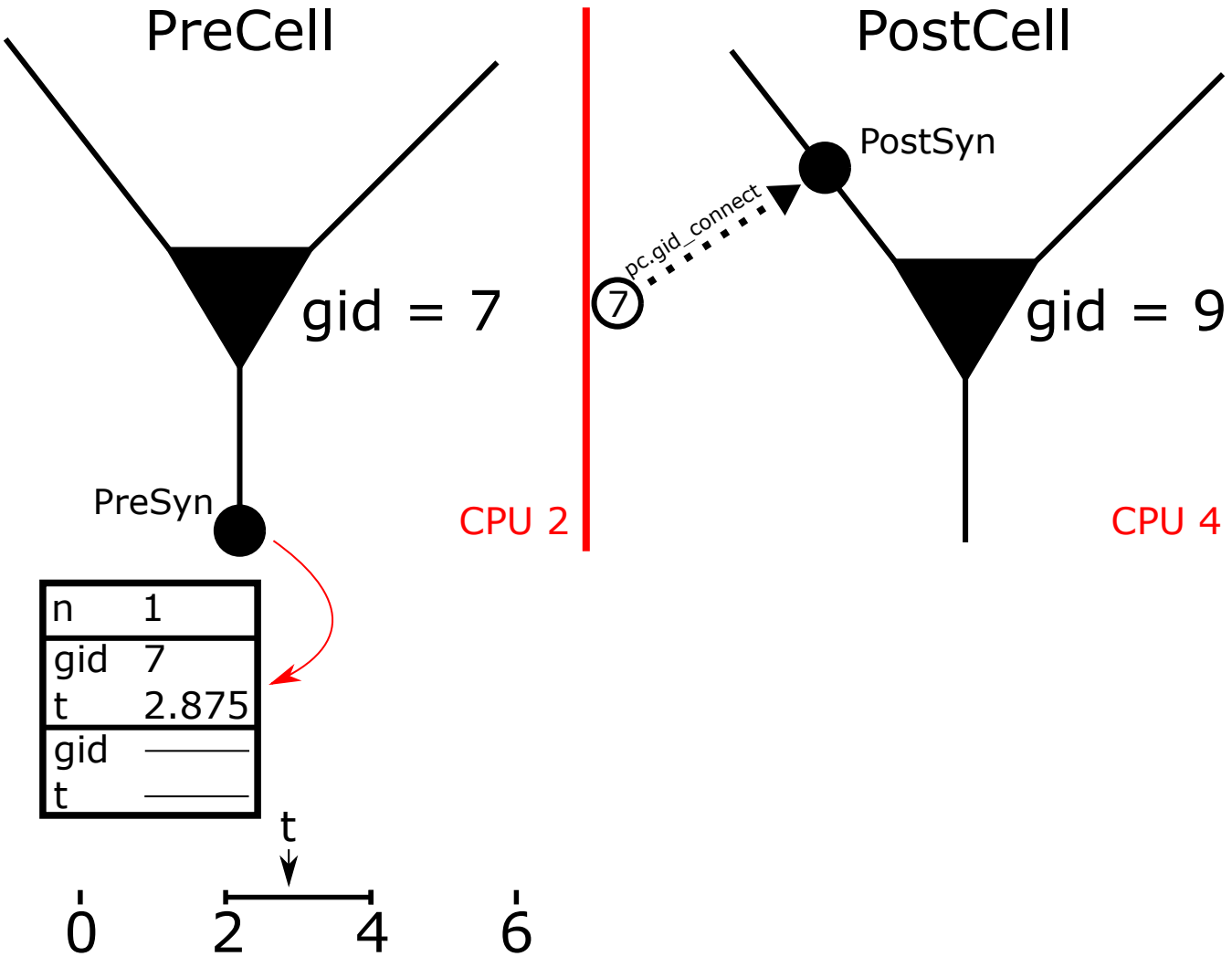
```
nc = pc.gid_connect(7, PostSyn)
```

PostSyn here is a Point Process, e.g. an ExpSyn.

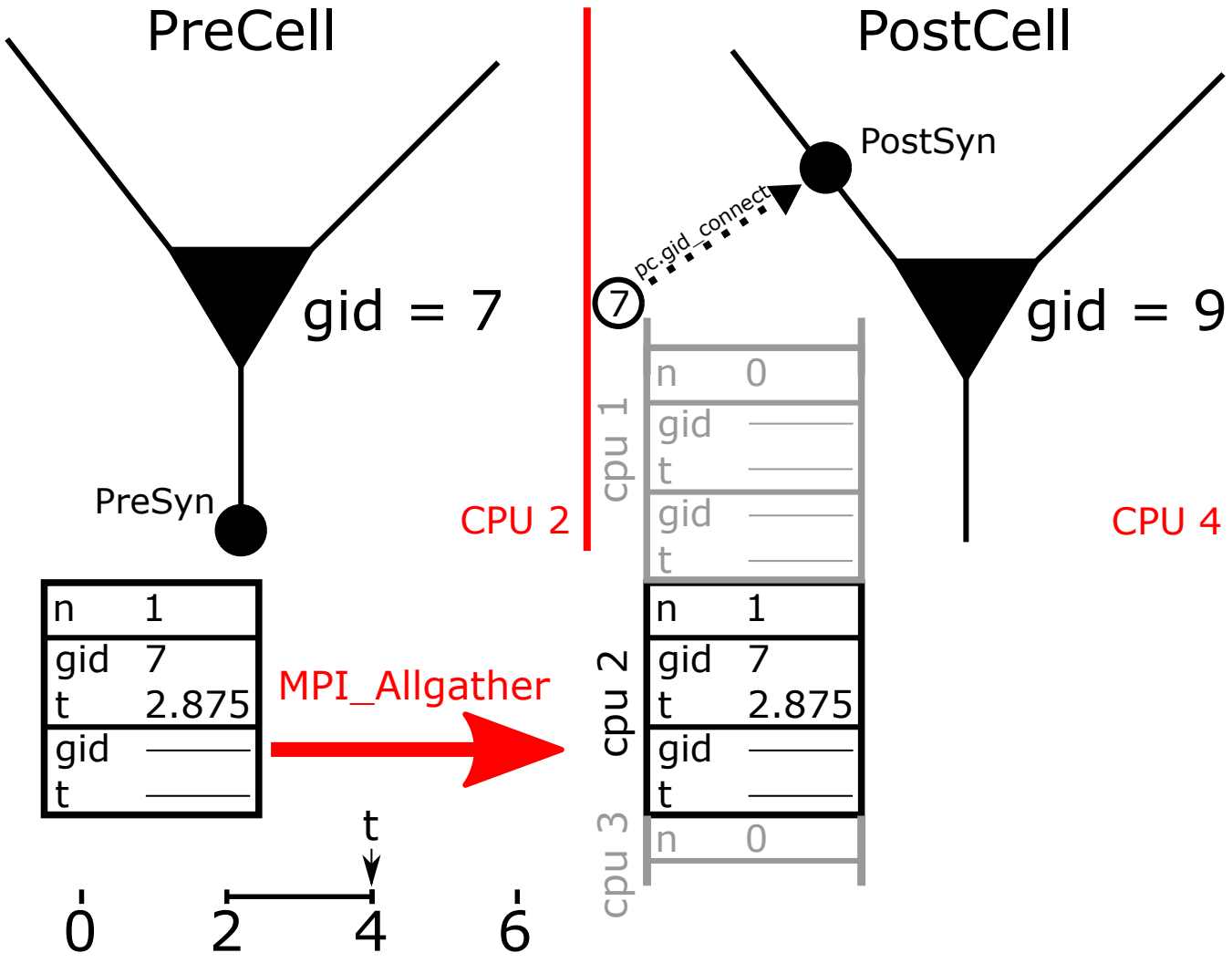
Spike exchange method



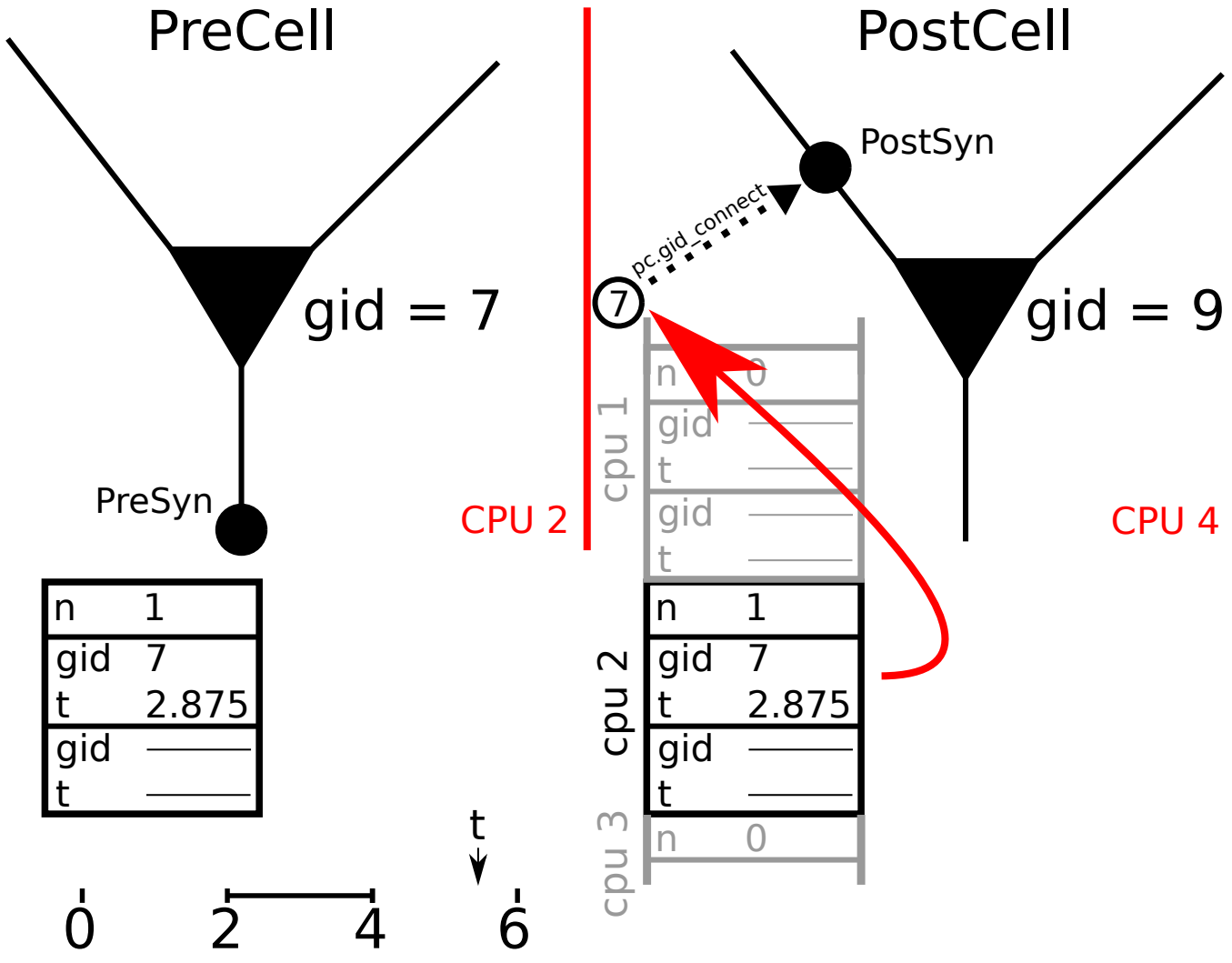
Spike exchange method



Spike exchange method



Spike exchange method

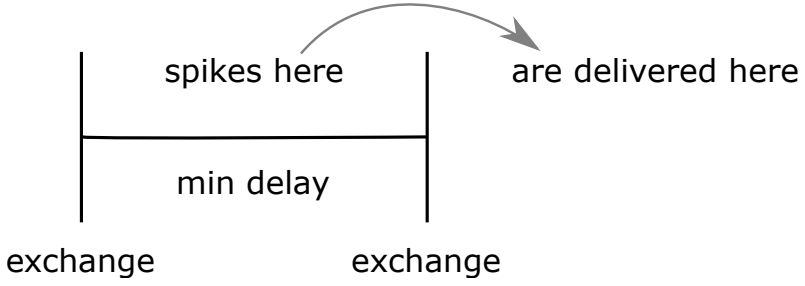


Exploit transmission delays: using `pc.set_maxstep`

Run using the idiom:

```
pc.set_maxstep(10)
h.stdinit()
pc.psolve(tstop)
```

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



`pc.set_maxstep` must be called on each node; it uses `MPI_Allreduce` to determine the minimum delay.

Adding a presynaptic site

```
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])
        pc = h.ParallelContext()
        pc.set_gid2node(self._gid, pc.id())
        pc.cell(self._gid, self.nc)
    # the rest of the class stays unchanged
```

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`.

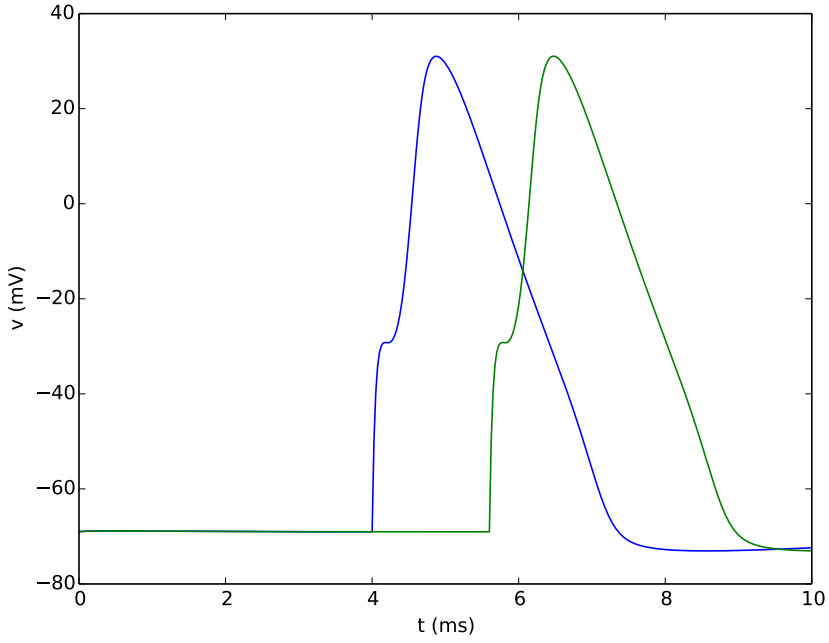
Building a two cell network

```
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
        # 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
        self.nc.weight[0] = 1

n = 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.

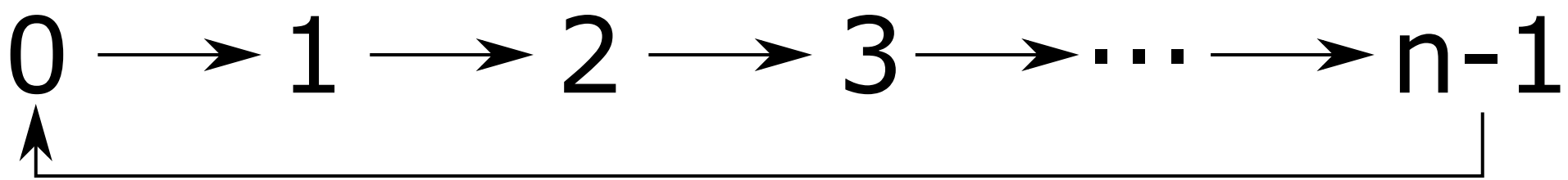
Running the two cell network

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

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

pc = h.ParallelContext()
pc.set_maxstep(10)
h.v_init = -69
h.stdinit()
pc.psolve(10)

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



Exercise: Generalizing to n cells in a ring network

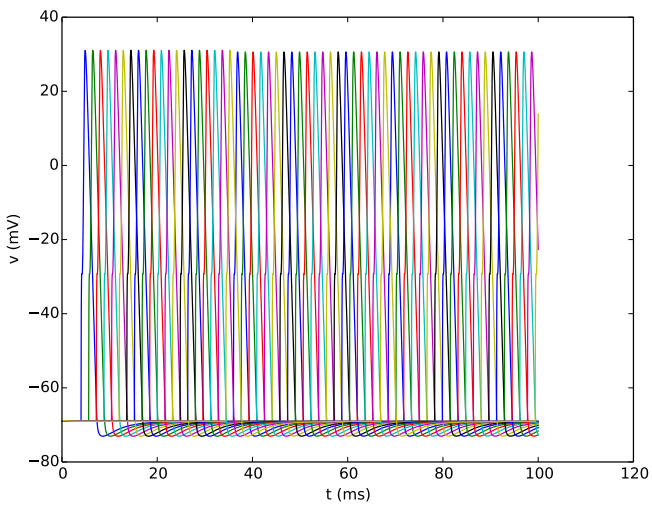
How can we generalize to a ring network with n cells?



Hint: As i increases, i % n counts: 0, 1, 2, ..., $n - 1$, 0, 1, ...

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

```
class Network:
    def __init__(self, num):
        self.cells = [Pyramidal(i) for i in range(num)]
        # 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
        # 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
            nc.weight[0] = 1
            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('%d: %r' % (i, list(cell.spike_times)))
```

Beginning of output:

```
0: [4.600000000100032, 36.62500000009977, 69.12500000010715]
1: [6.200000000100054, 38.25000000010014, 70.75000000010752]
2: [7.800000000100077, 39.875000000100506, 72.37500000010789]
3: [9.4000000001, 41.500000000100876, 74.00000000010826]
```

Storing spike times: JSON

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

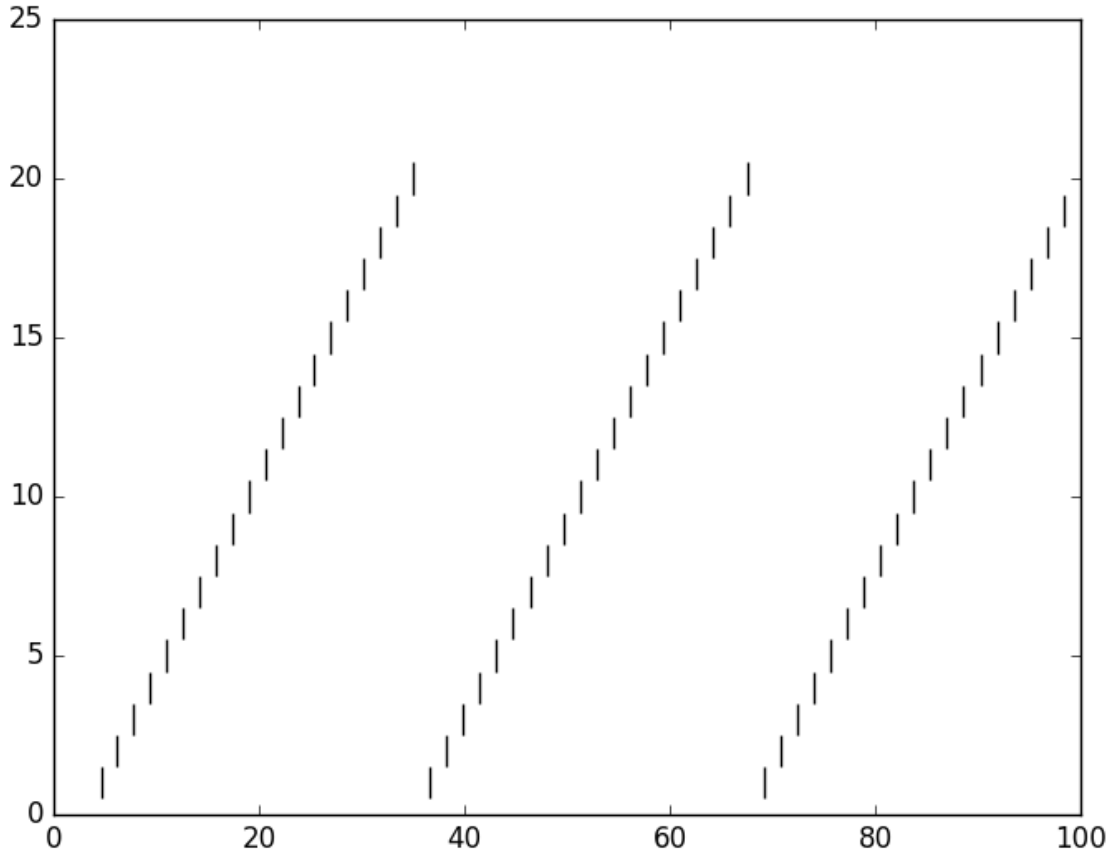
```
import json
with open('output.json', 'w') as f:
    f.write(json.dumps({i: list(cell.spike_times) for i, cell in enumerate(n.cells)},
                       indent=4))
```

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

```
    "0": [
        4.600000000100032,
        36.62500000009977,
        69.12500000010715
    ],
    "1": [
        6.200000000100054,
        38.25000000010014,
        70.75000000010752
    ],
    "2": [
        7.800000000100077,
        39.875000000100506,
        72.37500000010789
    ],
```

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

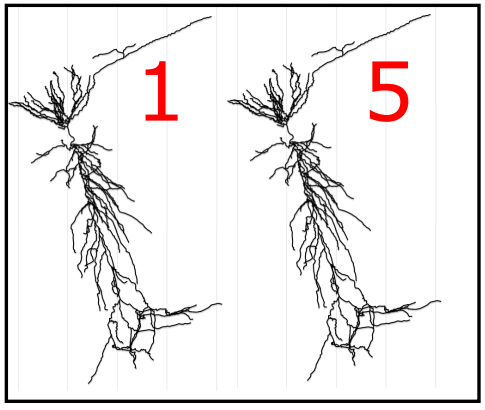
Raster plots



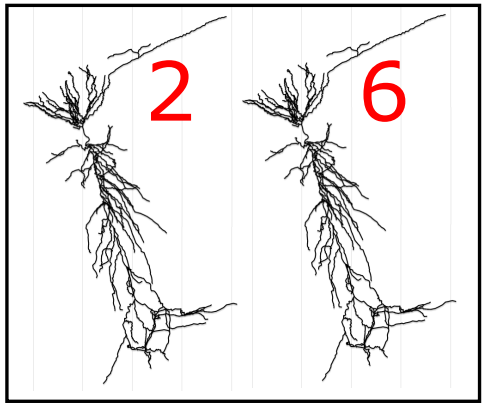

```
for i, cell in enumerate(n.cells):
    pyplot.vlines(cell.spike_times, i + 0.5, i + 1.5)
pyplot.show()
```

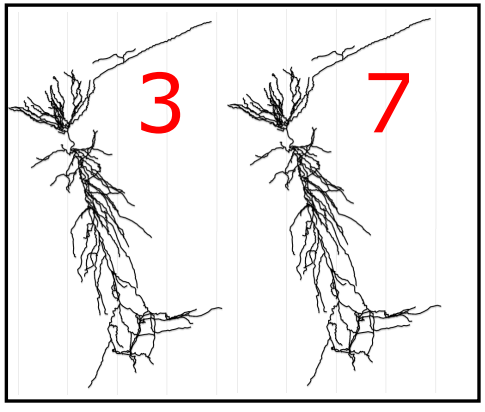
Simple load-balancing strategy: round-robin.

Processor 1

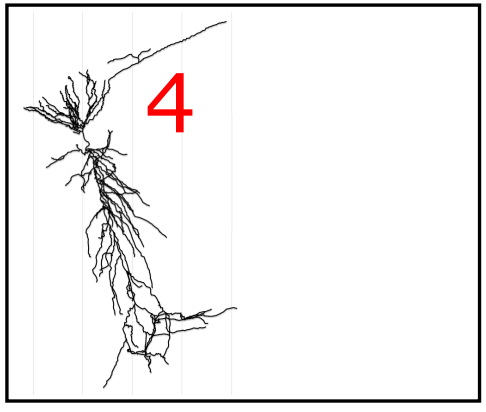


Processor 2





Processor 3



Processor 4

Simple load-balancing strategy: round-robin.

CPU 0		CPU 3		CPU 4	
`pc.id`	0	`pc.id`	3	`pc.id`	4
`pc.nhost`	5	`pc.nhost`	5	`pc.nhost`	5
`ncell`	14	`ncell`	14	`ncell`	14
gid		gid		gid	
0		3		4	
5		8		9	
10		13			

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 $\lceil$`ncell/nhost`$\rceil$ times, not `ncell`.)

Advanced load-balancing: balance work not number of cells

Strategy:

- 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.

For a more accurate (but computationally more intensive) estimate of complexity, use lb.ExperimentalMechComplex and lb.read_complex.

Parallelizing our ring network with round-robin

Very few changes are necessary.

An extra import at the very beginning:

```
from mpi4py import MPI
```

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
        # 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
            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
```

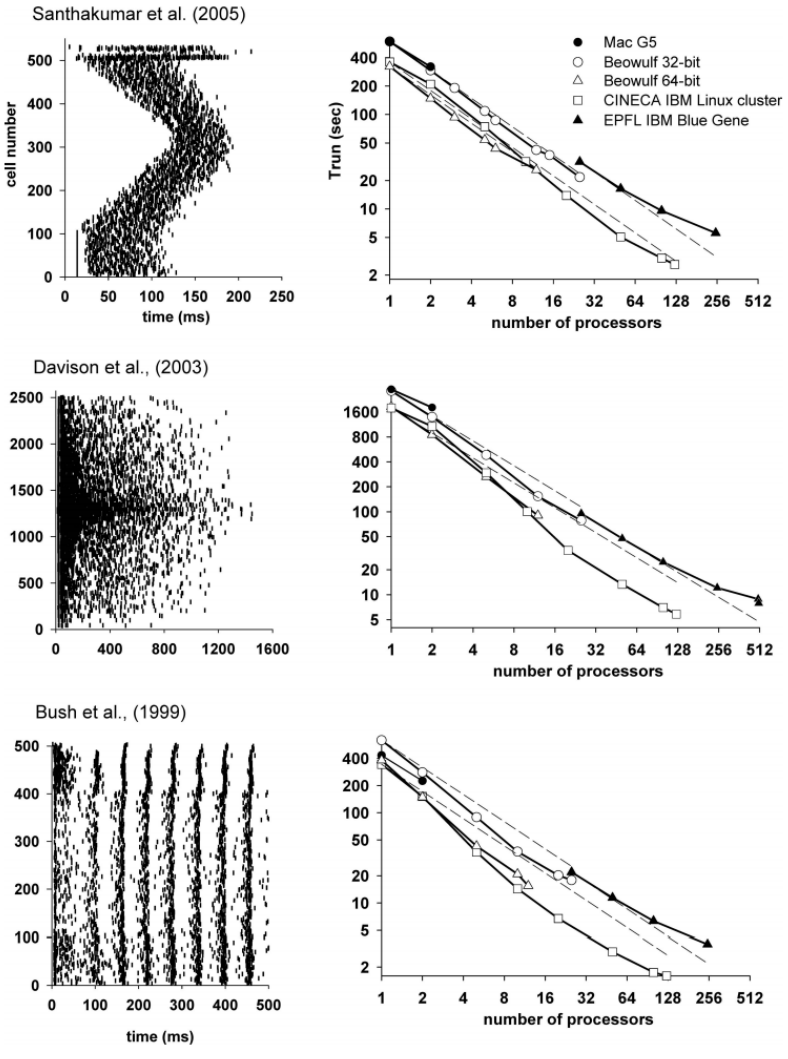
Finally, we modify the write to do it on a per-processor basis:

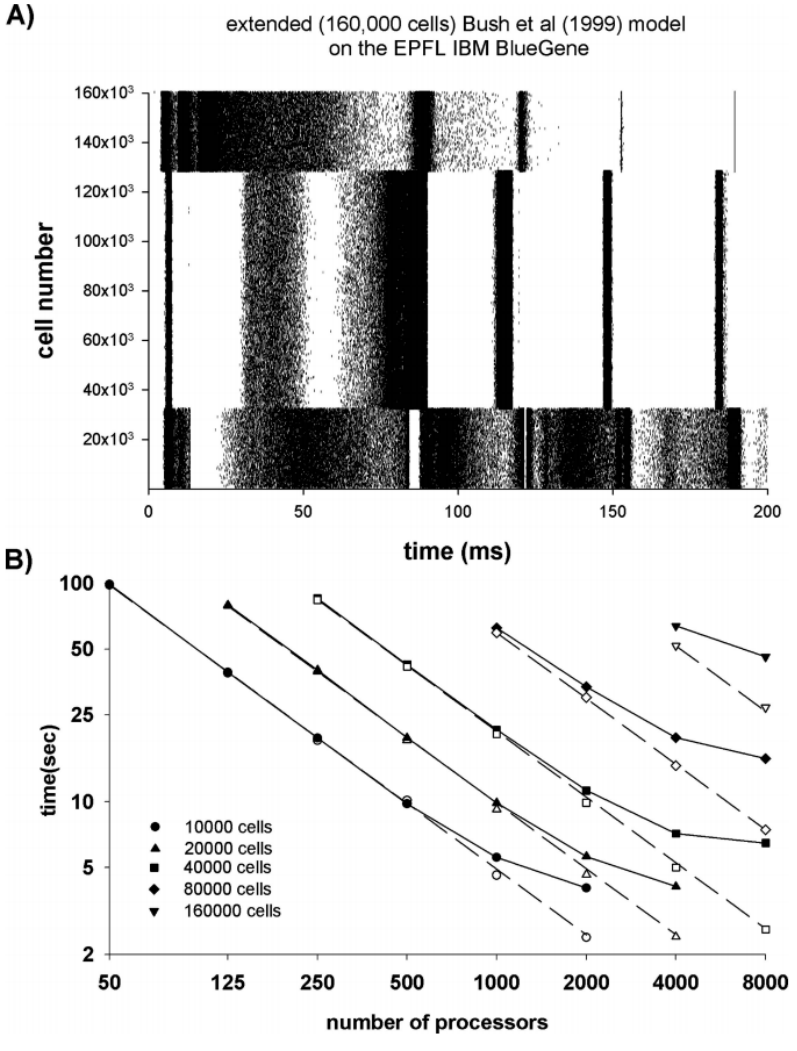
```
with open('output%d.json' % pc.id(), 'w') as f:
    f.write(json.dumps({cell._gid: list(cell.spike_times) for cell in n.cells},
        indent=4))
```

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

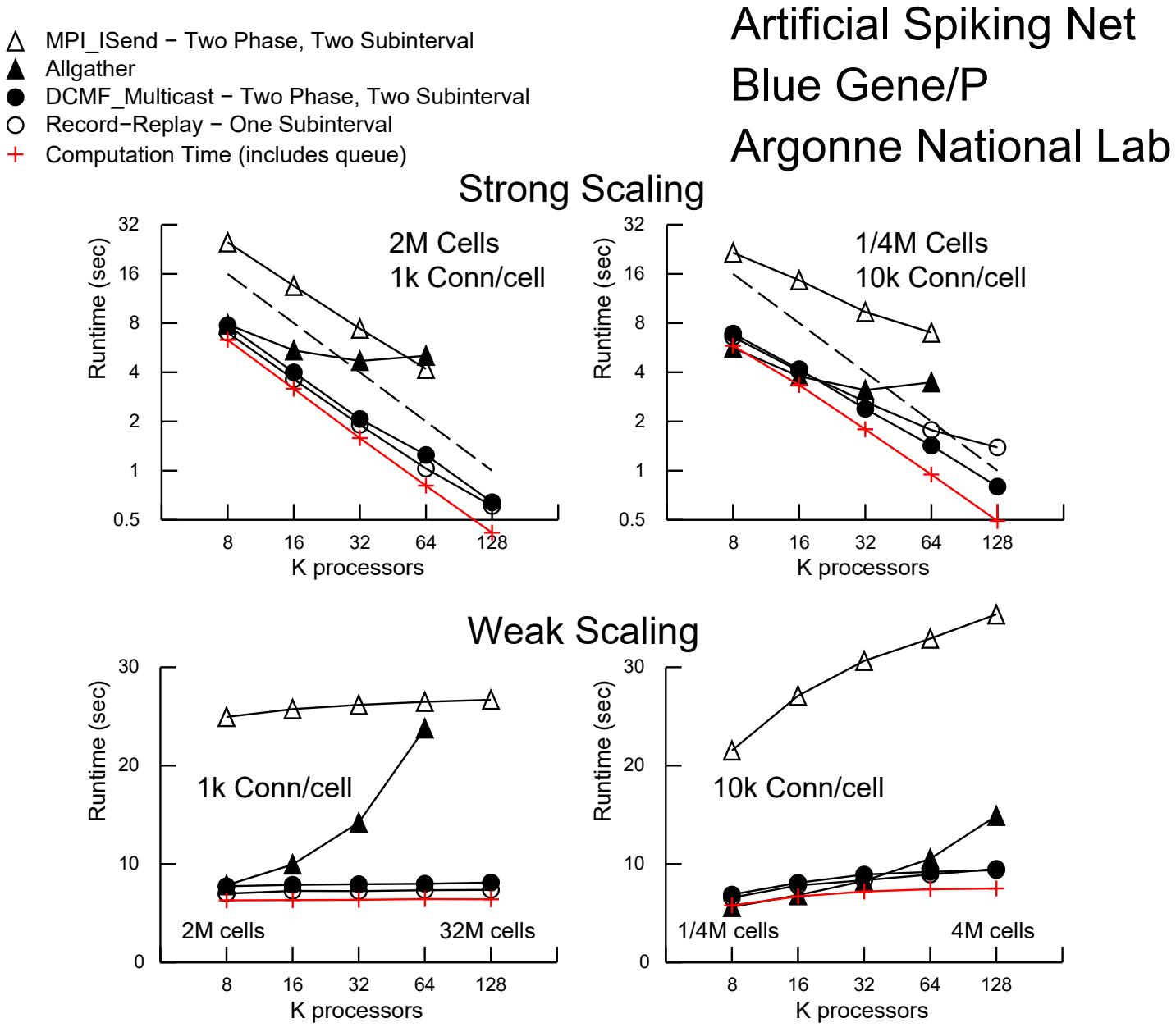
```
local_data = {cell._gid: list(cell.spike_times) for cell in n.cells}
all_data = pc.py_alltoall([local_data] + [None] * (pc.nhost() - 1))

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))
```

Performance: MPI scaling





Performance: Spike exchange strategies



Performance Tip

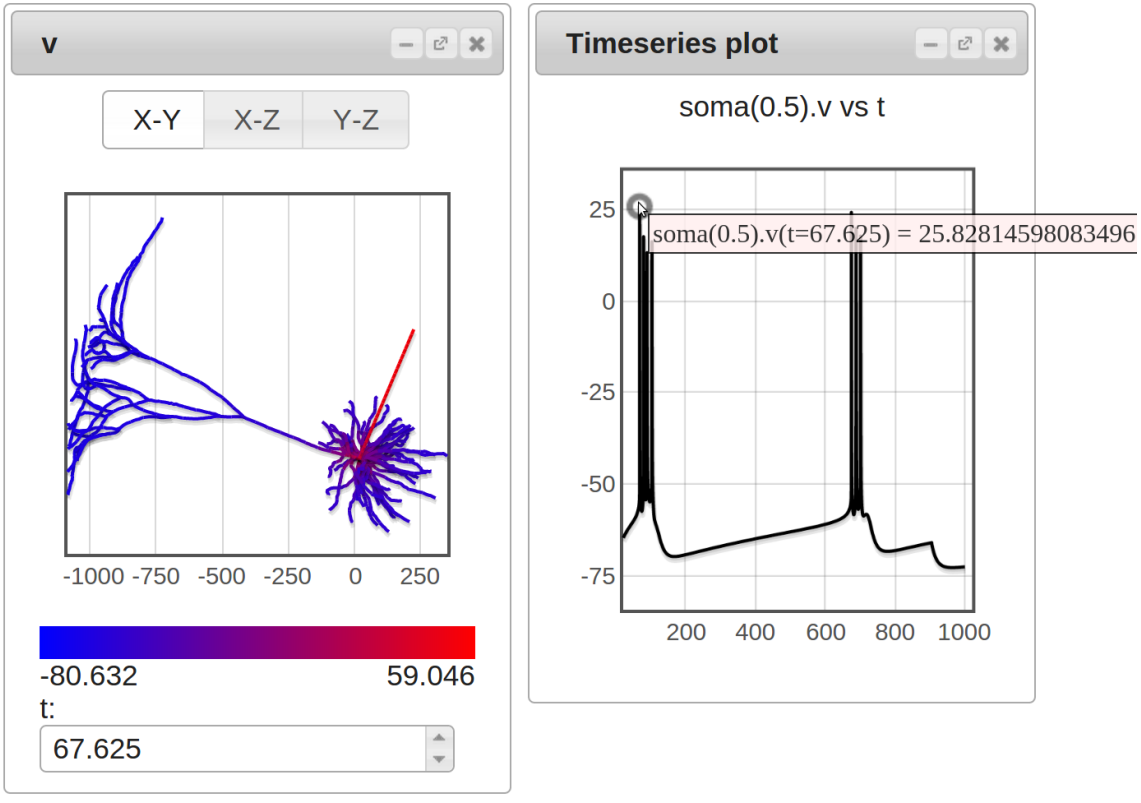
Tip: For network models, use a fixed step solver and not a variable step solver.

Question

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

Tip: Store synaptic events; recreate single 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

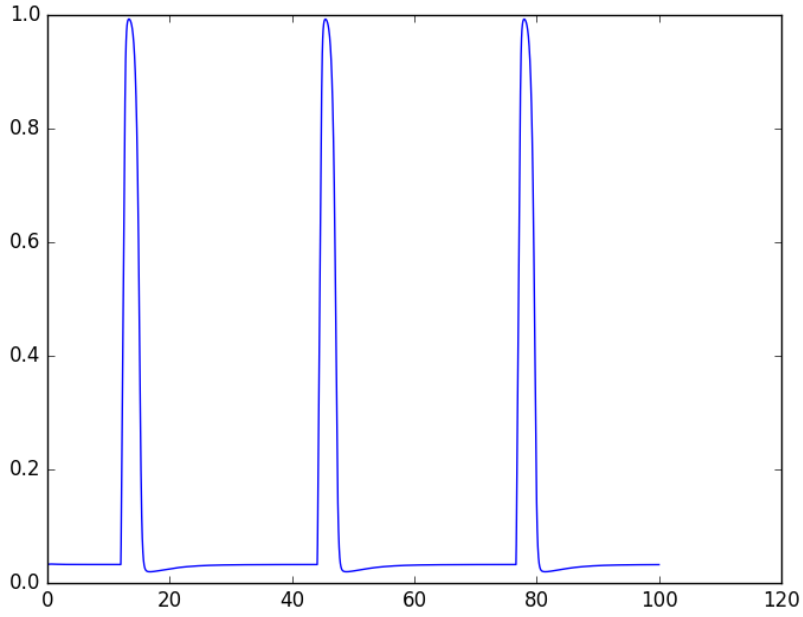
Using spike data to recreate a variable of interest

```
import json
from neuron import h
from 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)

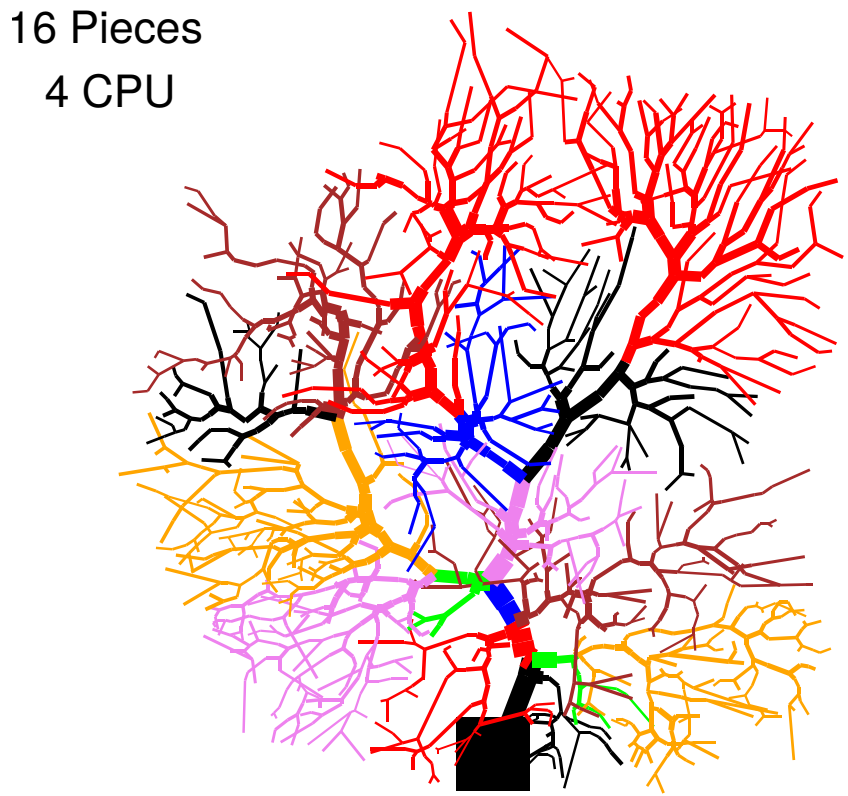
Using spike data to recreate a variable of interest

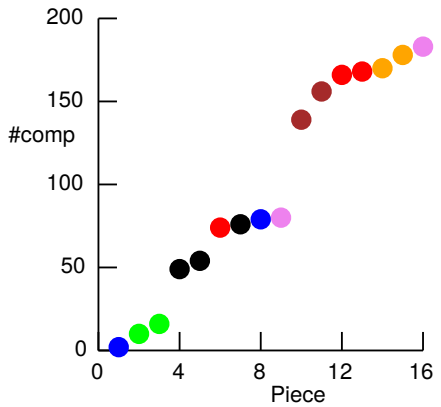
```
def get_m(gid):
    p = Pyramidal(gid)
    # recreate synaptic inputs (here, only one; you may have multiple)
    precell = (gid - 1) % num_cells
    vs = h.VecStim()
    spike_vec = h.Vector(spike_times_by_cell[str(precell)])
    vs.play(spike_vec)
    syn = h.ExpSyn(p.dend[0](0.5))
    nc = h.NetCon(vs,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

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

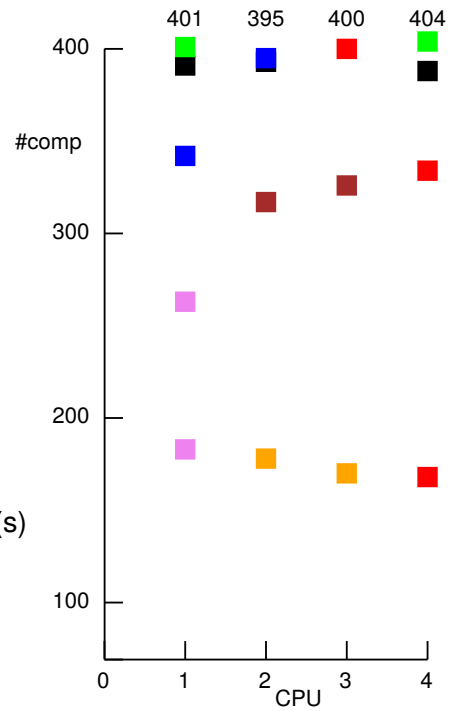


Multisplit

Improve load balancing with multisplit







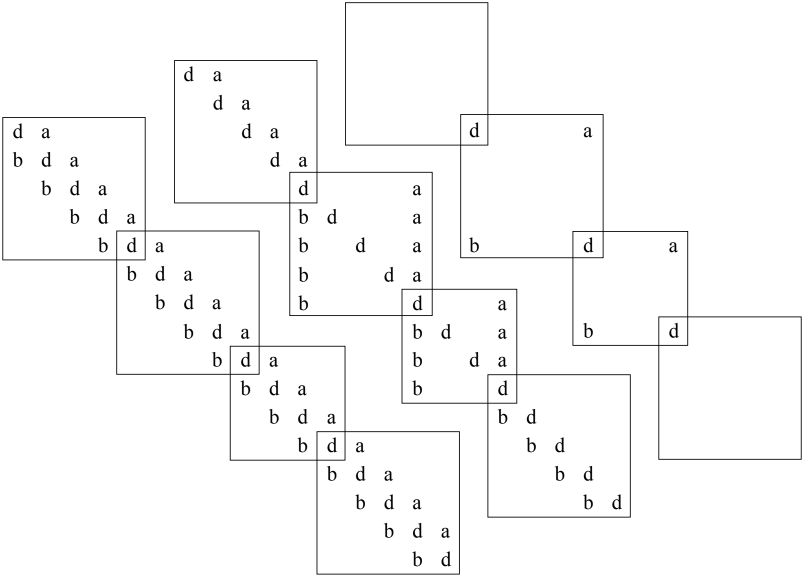
Time (s)

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

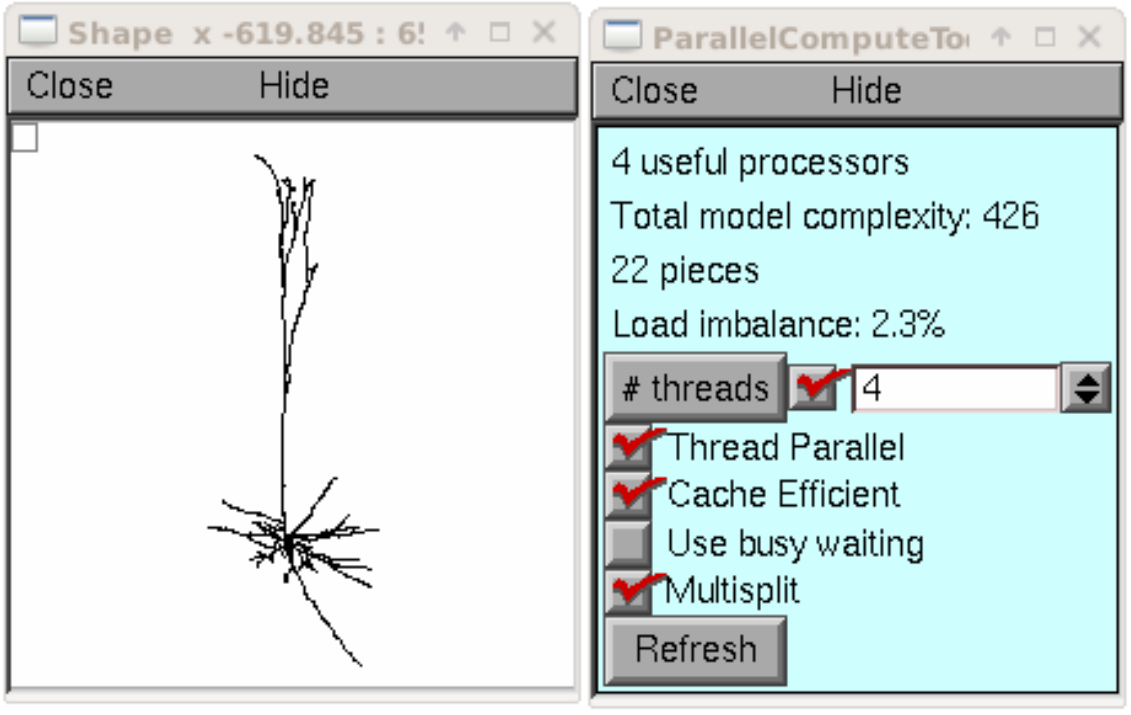
Multisplit algorithm described in Hines et al 2008. DOI: 10.1007/s10827-008-0087-5

Multisplit: methods



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(x, subtreeid, sec=sec)
```

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

```
pc.multisplit()
```

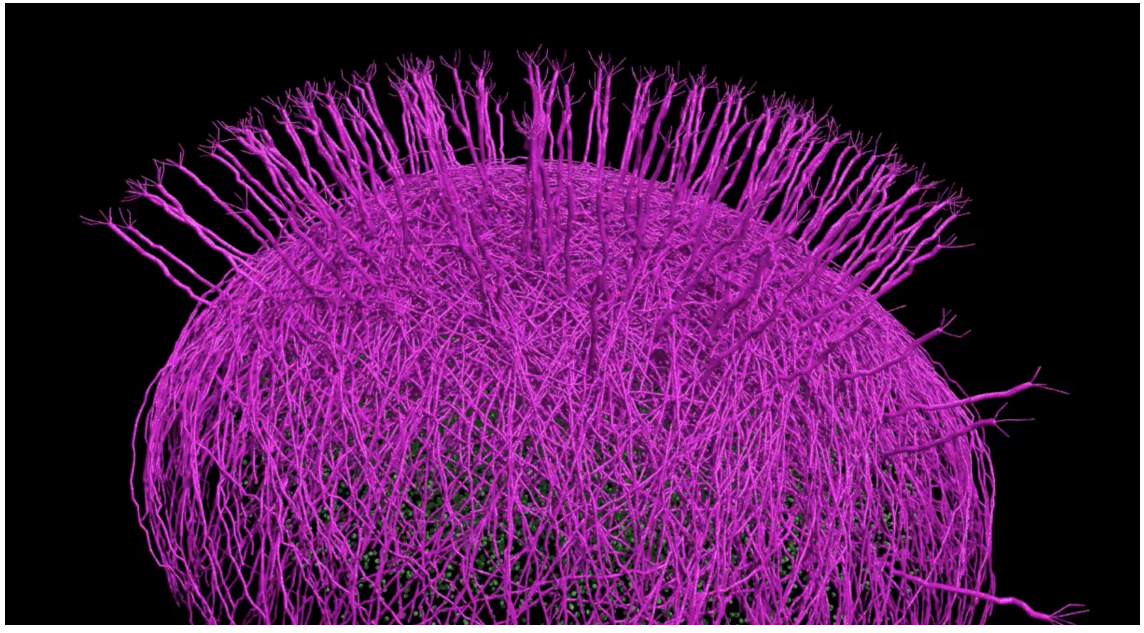
If created, destroy any parts of the cell that do not belong on the processor.

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



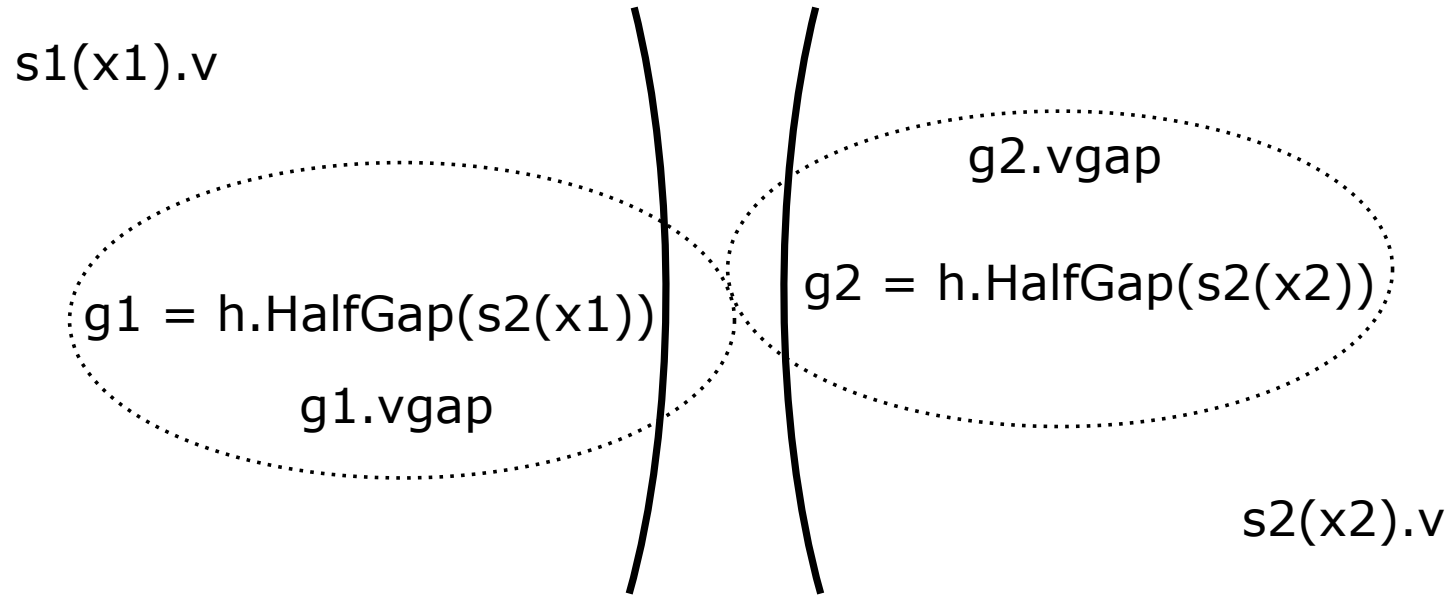
Migliore et al 2014 used multisplit to improve load balancing on a model of the olfactory bulb.

http://modeldb.yale.edu/151681

See, in particular, the file `multisplit_distrib.py`.

Gap Junctions

Continuous voltage exchange

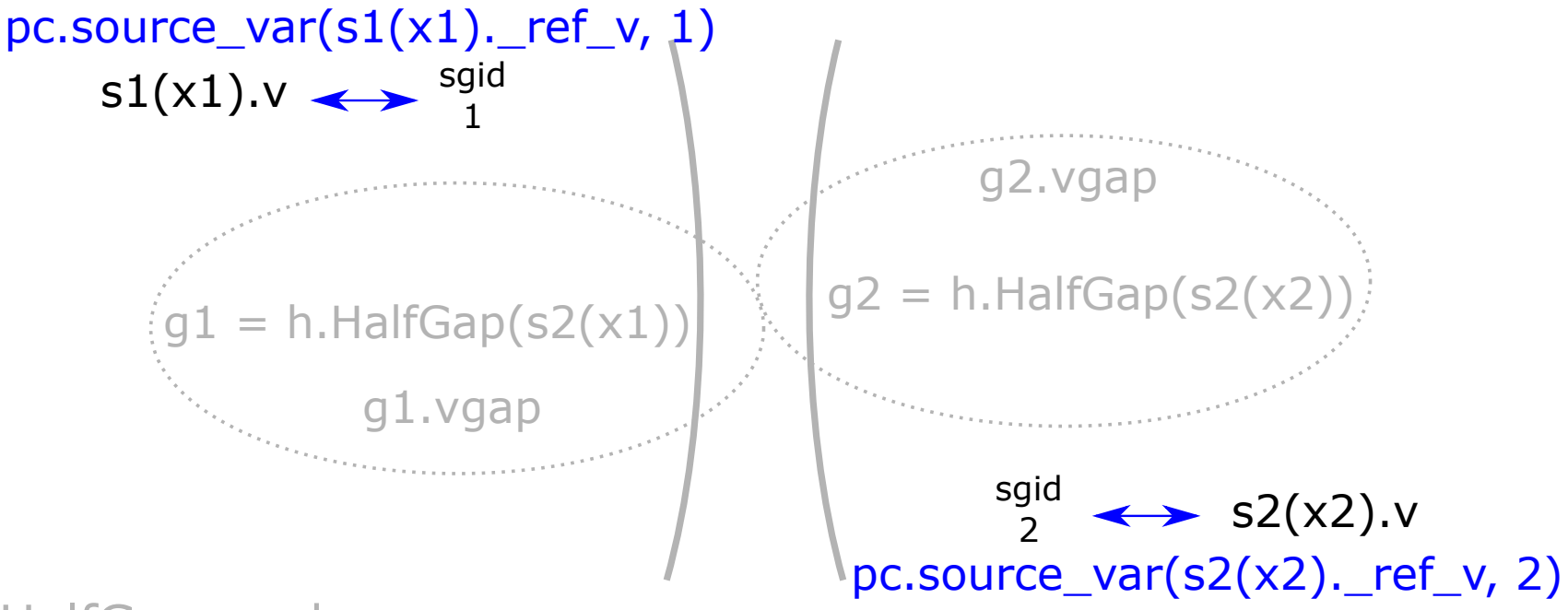


HalfGap.mod

```
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

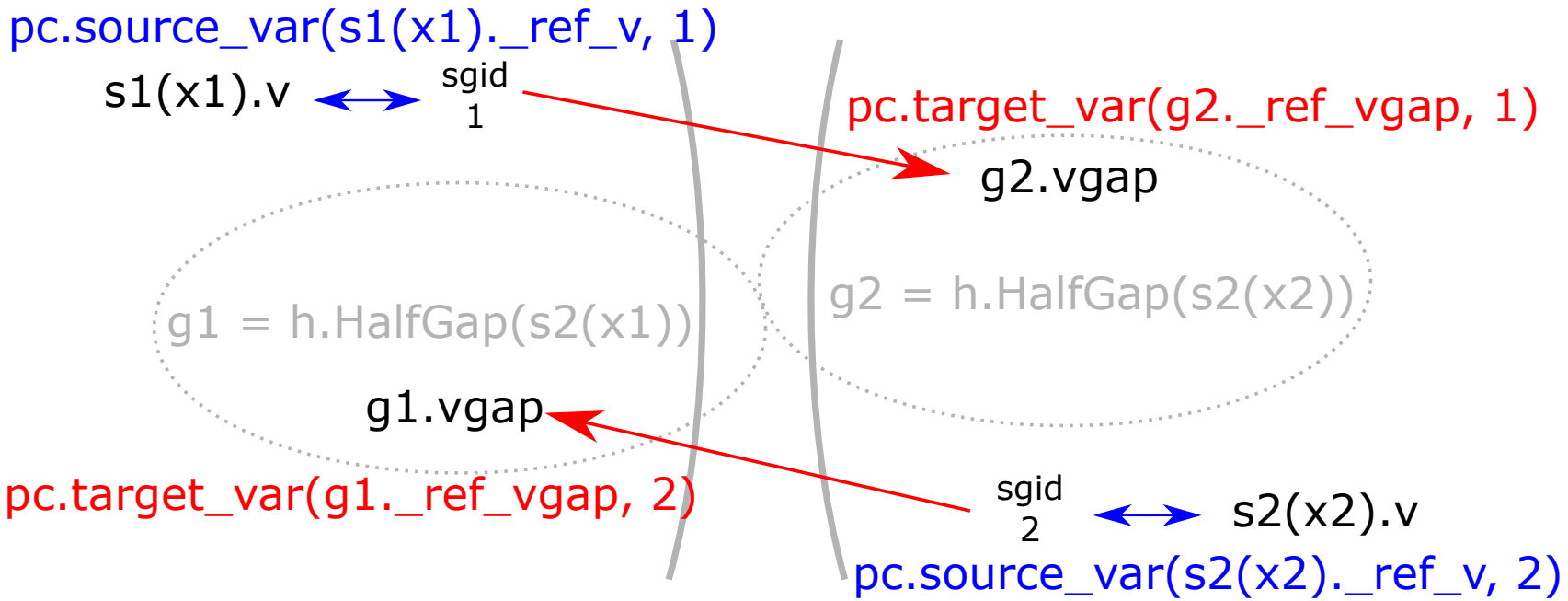


HalfGap.mod

```
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) }
```

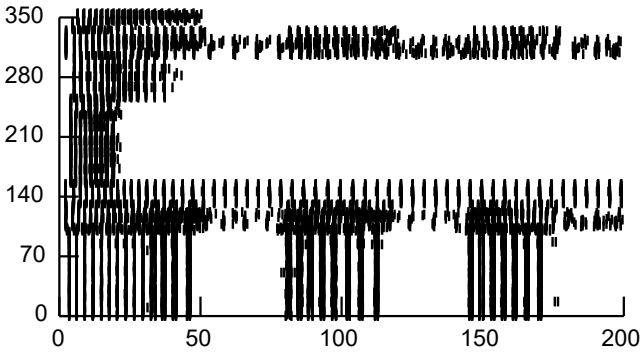
```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap - v) / r }
```

Performance: Traub model

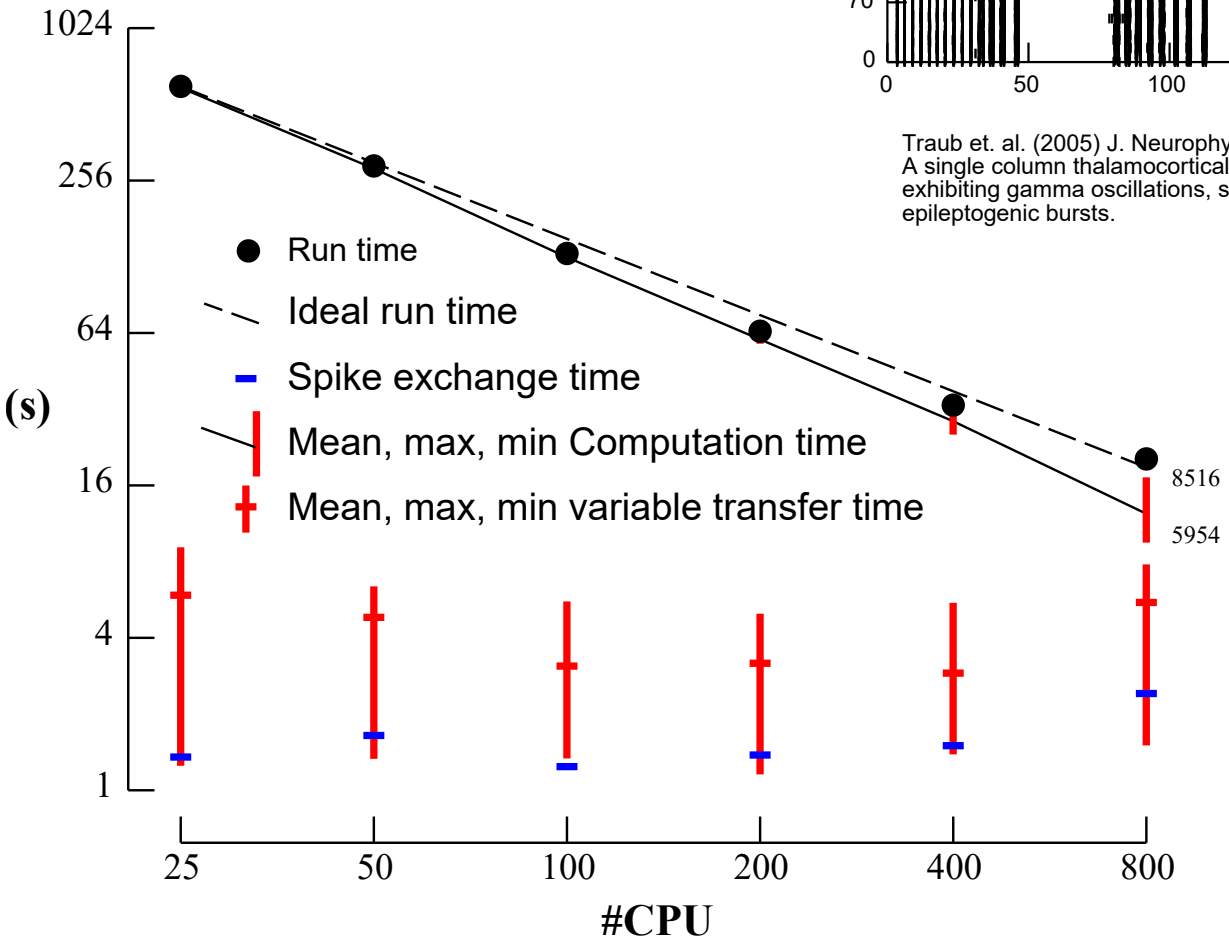
Pittsburgh Supercomputing Center

Bigben Cray XT3

2068 2.4 GHz Opteron Processors

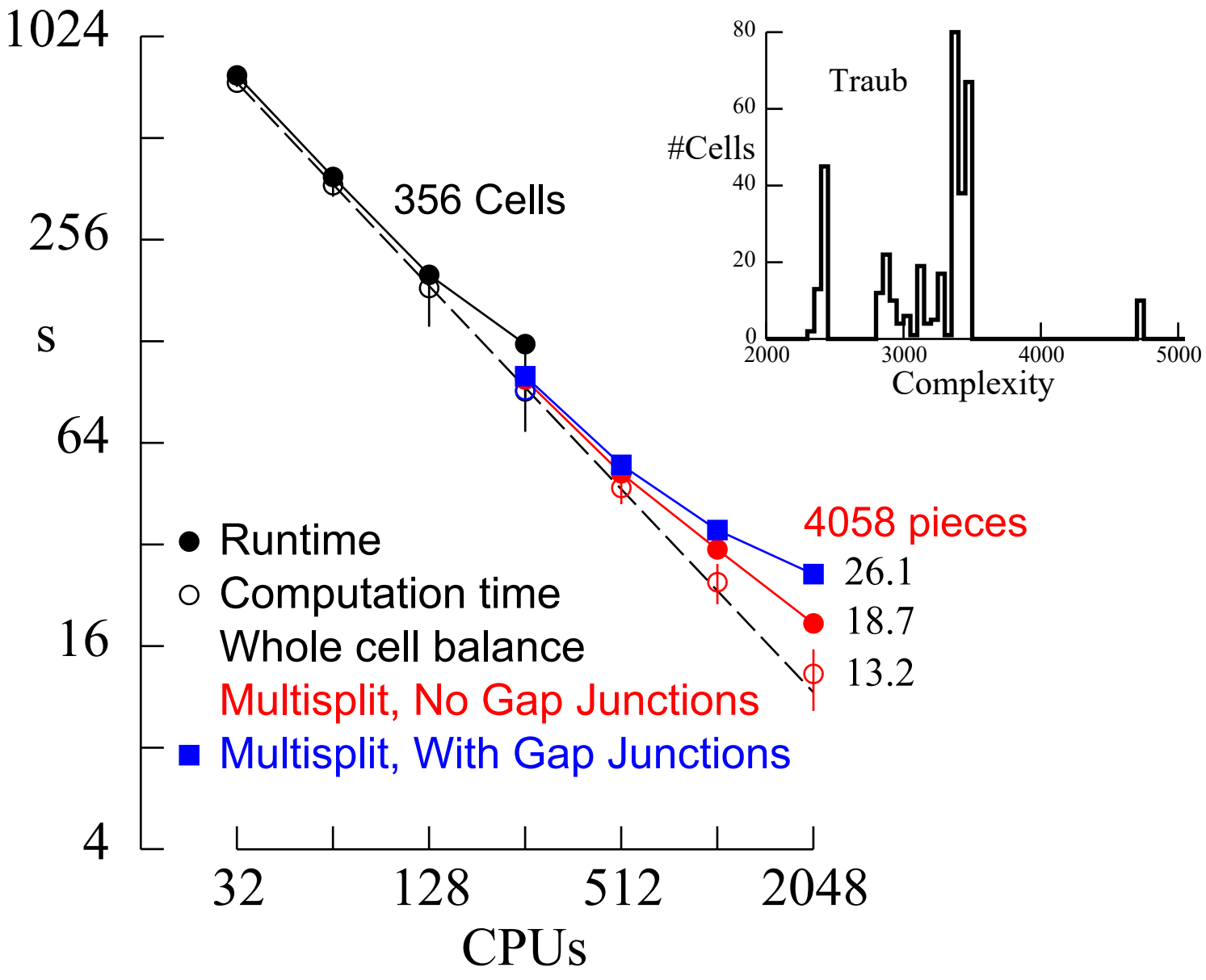


Traub et. al. (2005) J. Neurophysiol 93: 2194
A single column thalamocortical network model exhibiting gamma oscillations, sleep spindles and epileptogenic bursts.



3560 cells 14 types
3500 gap junctions
5,596,810 equations
73,465 spikes
1,122,520 connections
19,844,187 delivered

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
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.