Building, Running, and Visualizing Parallel NEURON Models

Robert A. McDougal

Yale School of Public Health

24 November 2020

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.

Other considerations

The 16384 core EPFL IBM BlueGene/P could 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

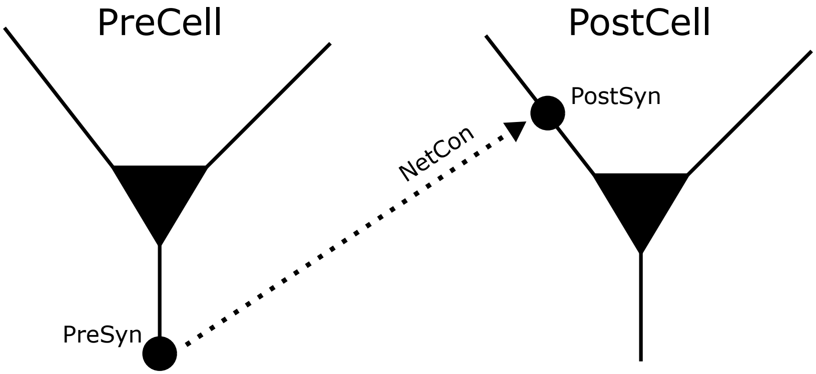
- A network of neurons is composed of many individual neurons of potentially many cell types. Design and debug each cell type separately before building the network.
- A simulation should give the same results regardless of the number of processors used to run it.
- When possible, parameterize your network so you can run a small test first.

Connecting to MPI

Before we can do any MPI simulations, we need to let the computer know to initialize communication between multiple processors:

```
h.nrnmpi_init()
```

Synaptic connections with one processor

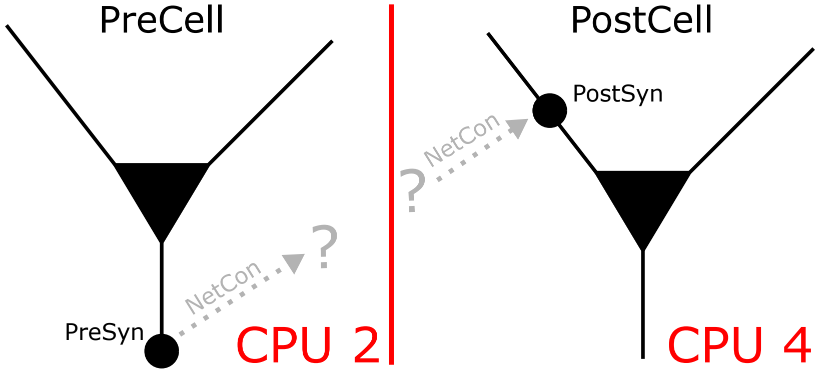


```
nc = h.NetCon(PreSynPtr, PostSyn, sec=presyn_section)
nc.delay = 1 * ms
```

By default, `delay` is measured in ms.

We can also set: `nc.weight` and `nc.threshold[]`.

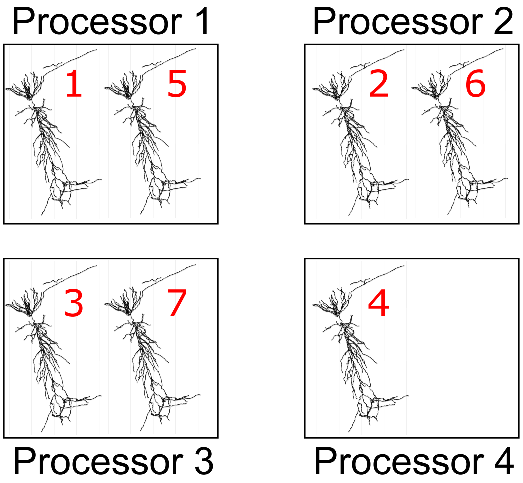
If cells in different processes, a different approach is needed



The `ParallelContext` object facilitates building parallel models.

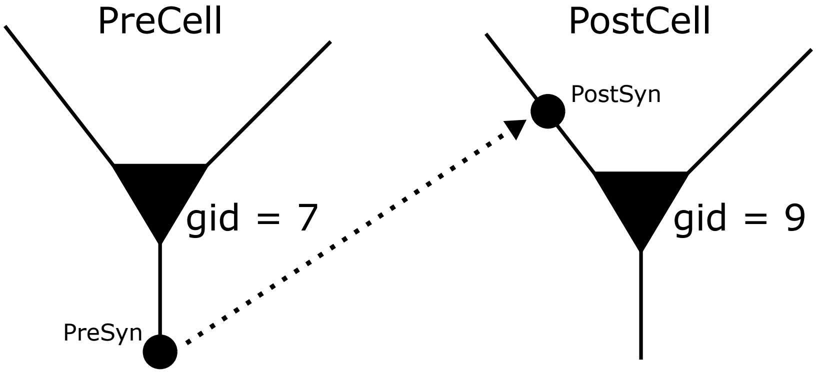
```
pc = h.ParallelContext()
```

Every spike source **must** have a GID.

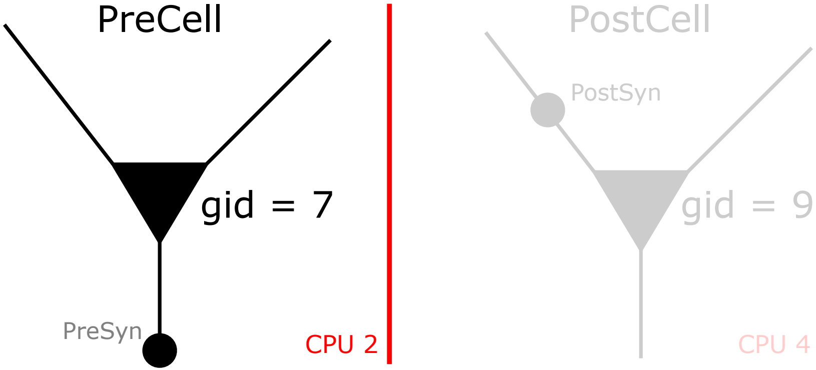


Note: to ensure the model produces identical results regardless of the number of processors, also use GIDs to selecting random streams (e.g. `Random123`).

Building synapses



Configuring the presynaptic connection site



Create cell only where the gid exists:

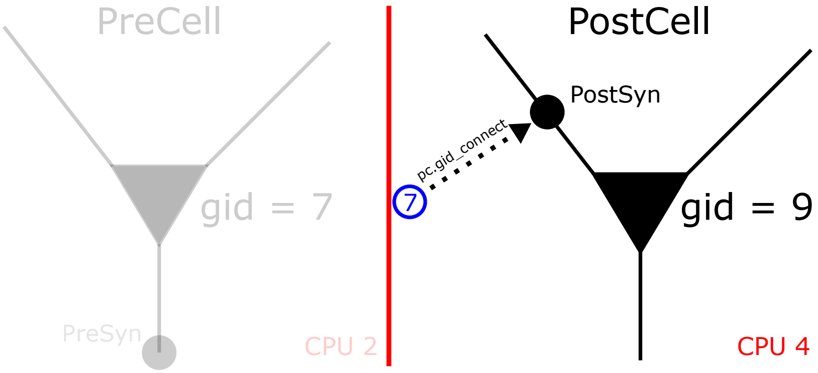
```
if pc.gid_exists(7):
    PreCell = Cell()
```

Associate gid with spike source:

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

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

Configuring the postsynaptic connection site

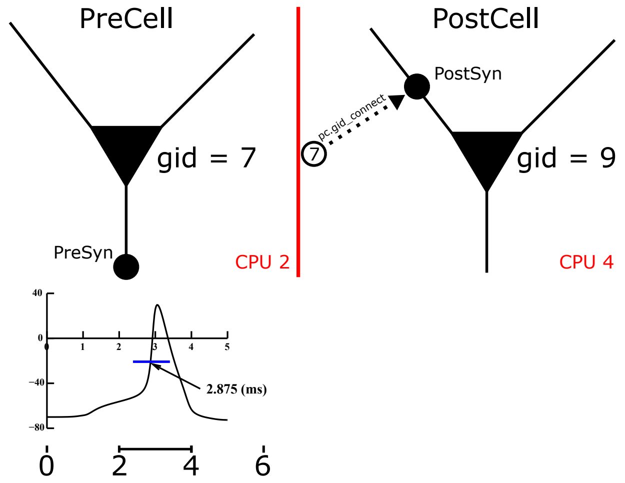


Create NetCon on node where target exists:

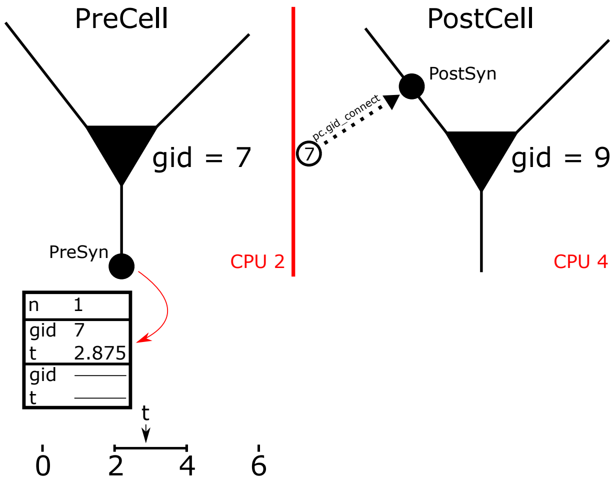
```
nc = pc.gid_connect(7, PostSyn)
```

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

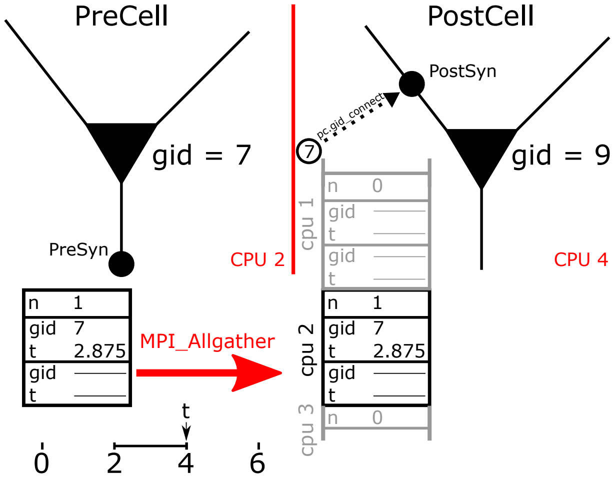
Spike exchange method



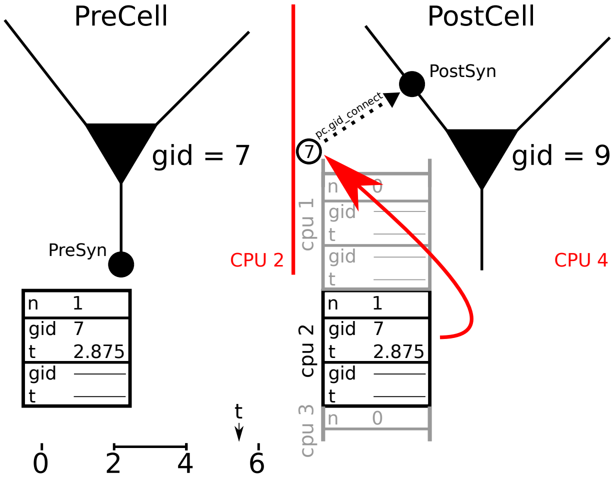
Spike exchange method



Spike exchange method



Spike exchange method

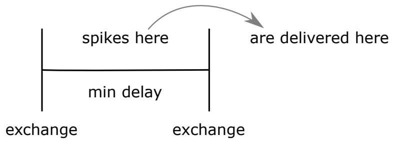


Exploit transmission delays: using `pc.set_maxstep`

Run using the idiom:

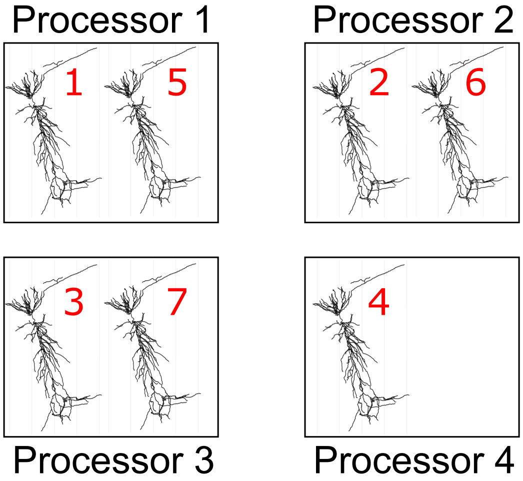
```
pc.set_maxstep(10)
h.stdinit()
pc.psolve(tstop)
```

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.

Simple load-balancing strategy: round-robin.



Simple load-balancing strategy: round-robin.

CPU 0

`pc.id`	0
`pc.nhost`	5
`ncell`	14

gid
0
5
10

●●●

CPU 3

`pc.id`	3
`pc.nhost`	5
`ncell`	14

gid
3
8
13

CPU 4

`pc.id`	4
`pc.nhost`	5
`ncell`	14

gid
4
9

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:

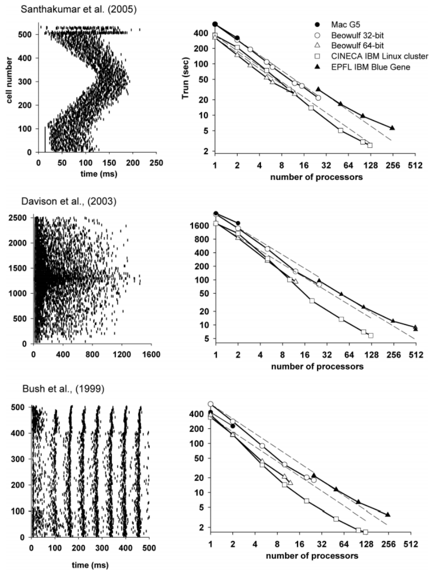
- Distribute cells round-robin to all processors, instantiate them.
- Compute an estimate of the computational complexity:

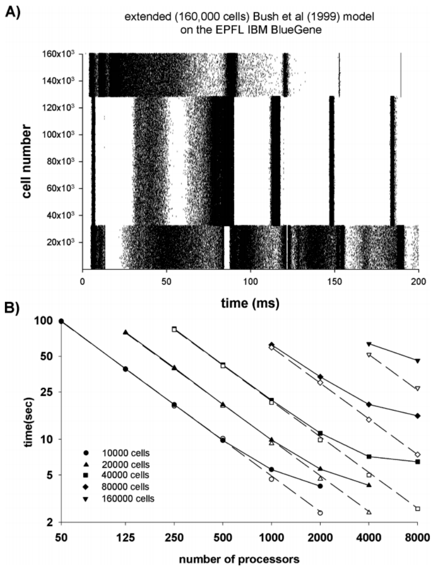
```
def complexity(self):
    h.load_file('loadbal.hoc')
    lb = h.LoadBalance()
    return lb.cell_complexity(sec=self.all[0])
```

- 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.

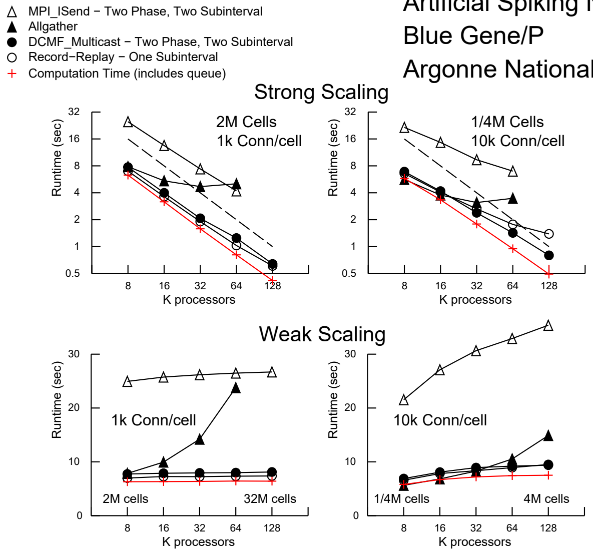
Performance: MPI scaling





Performance: Spike exchange strategies

Artificial Spiking Net
Blue Gene/P
Argonne National Lab



Performance Tip

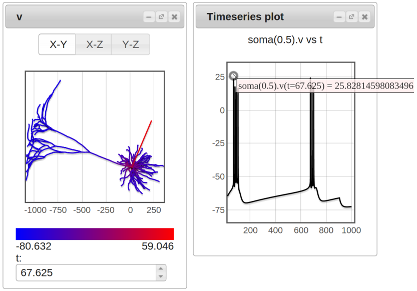
Tip: For network models, use a fixed step solver and not a variable step solver.

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

$$\begin{array}{c}\text{initial conditions}\\ + \\ \text{synaptic events}\end{array} \longrightarrow \text{neuron dynamics}$$



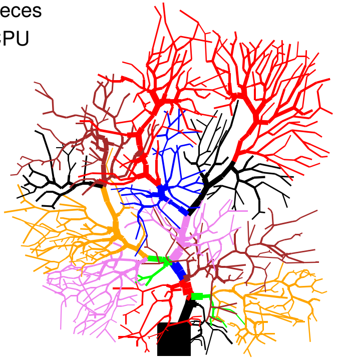
soma(0.5).v(t=67.625) = 25.82814598083496

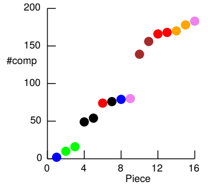
Use `NetCon.record` method to store spike times; save them as e.g. JSON. Play them back into a single cell simulation using `h.PatternStim()` and its `play(time, gid)` method.

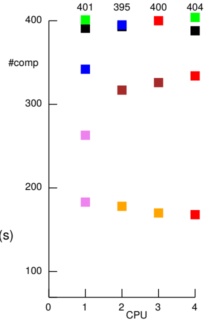
Multisplit

Improve load balancing with multisplit

16 Pieces
4 CPU







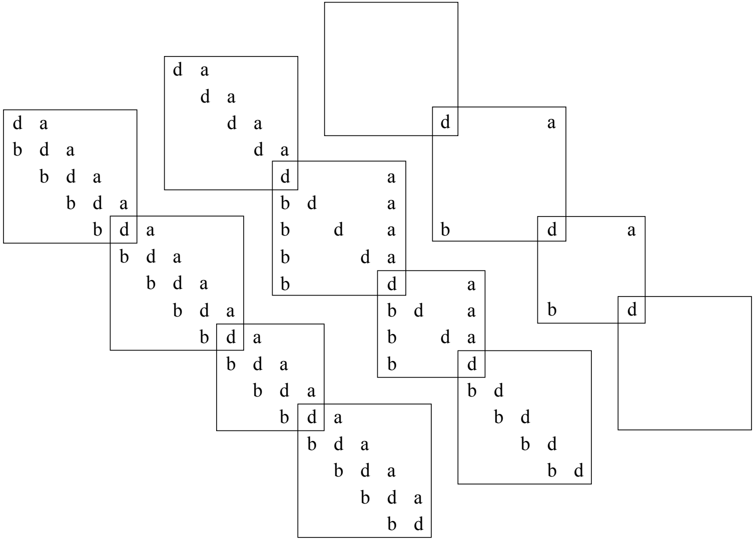
Time (s)

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

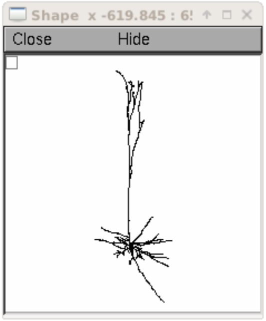
Multisplit algorithm described in Hines et al 2008. DOI: 10.1007/s10827-008-0087-5

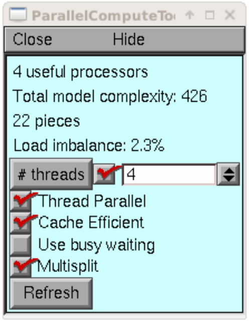
Multisplit: methods



Using multisplit (threads)

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





4 useful processors
Total model complexity: 426
22 pieces
Load imbalance: 2.3%

threads ✓ 4
✓ Thread Parallel
✓ Cache Efficient
Use busy waiting
✓ Multisplit

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()
```

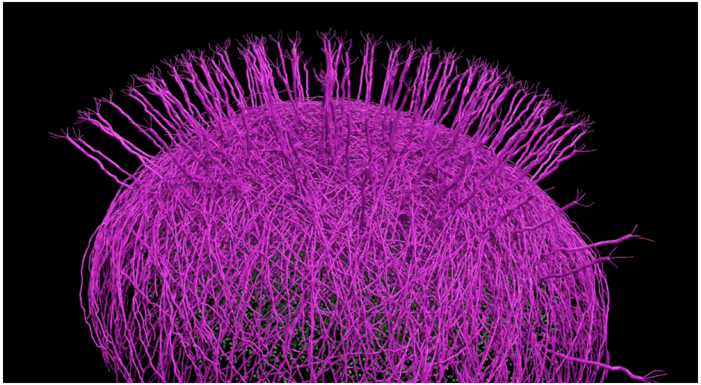
If created, destroy any parts of the cell that do not belong on the processor.

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



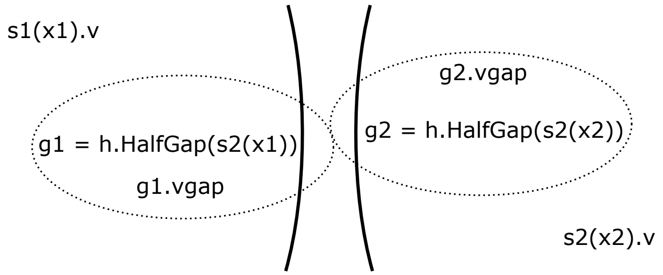
Migliore et al 2014 used multisplit to improve load balancing on a model of the olfactory bulb.

http://modeldb.yale.edu/151681

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

Gap Junctions

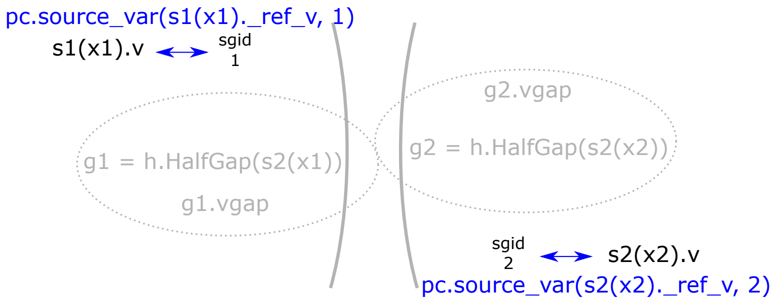
Continuous voltage exchange



HalfGap.mod

```
NEURON  {                               ASSIGNED {
  POINT_PROCESS HalfGap                   v (millivolt)
  ELECTRODE_CURRENT i                     vgap (millivolt)
  RANGE r, i, vgap                        i (nanoamp)
}                                       }
PARAMETER { r = 1e9 (megohm) }          CURRENT { i = (vgap - v) / r }
```

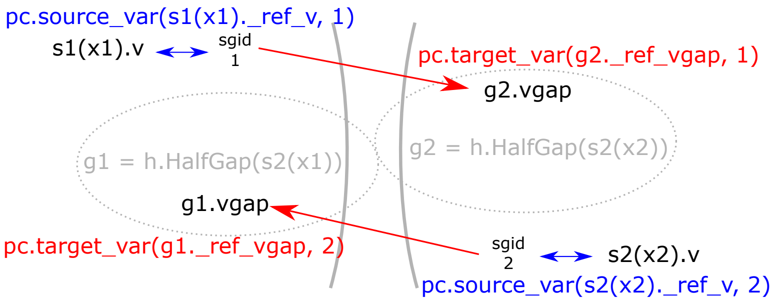
pc.source_var to declare source sgid



HalfGap.mod

```
NEURON  {                            ASSIGNED {
  POINT_PROCESS HalfGap                v (millivolt)
  ELECTRODE_CURRENT i                  vgap (millivolt)
  RANGE r, i, vgap                     i (nanoamp)
}                                    }
PARAMETER { r = 1e9 (megohm) }       CURRENT { i = (vgap - v) / r }
```

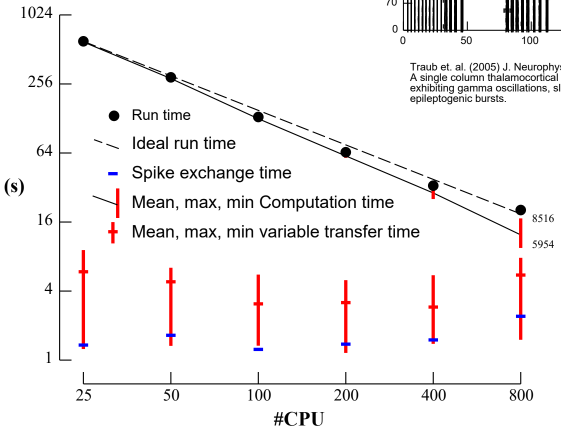
pc.target_var to declare target connection

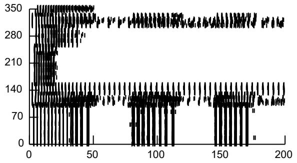

pc.source_var(s1(x1)._ref_v, 1)
s1(x1).v
sgid 1
pc.target_var(g2._ref_vgap, 1)
g2.vgap
g1 = h.HalfGap(s2(x1))
g2 = h.HalfGap(s2(x2))
g1.vgap
pc.target_var(g1._ref_vgap, 2)
sgid 2
s2(x2).v
pc.source_var(s2(x2)._ref_v, 2)

HalfGap.mod

```
NEURON  {                             ASSIGNED {
  POINT_PROCESS HalfGap                 v (millivolt)
  ELECTRODE_CURRENT i                   vgap (millivolt)
  RANGE r, i, vgap                      i (nanoamp)
}                                     }
PARAMETER { r = 1e9 (megohm) }        CURRENT { i = (vgap - v) / r }
```

Performance: Traub model

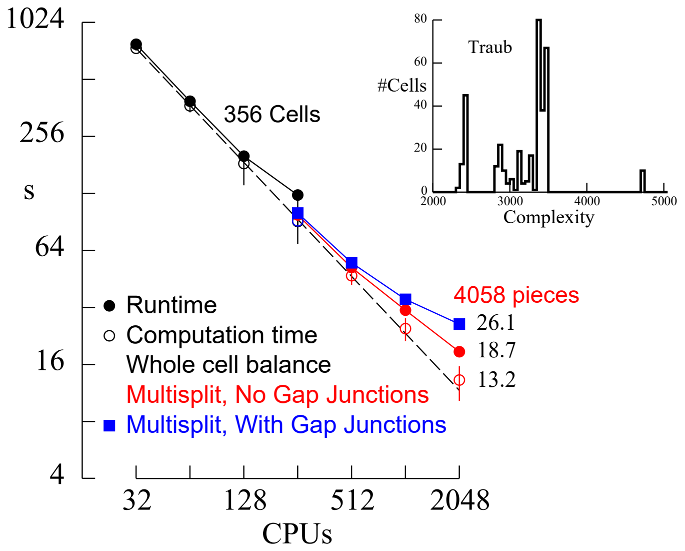




Traub et. al. (2005) J. Neurophysiol 93: 2194
A single column thalamocortical network model exhibiting gamma oscillations, sleep spindles and epileptogenic bursts.

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


s
CPUs
356 Cells
● Runtime
○ Computation time
Whole cell balance
Multisplit, No Gap Junctions
■ Multisplit, With Gap Junctions
4058 pieces
#Cells
Traub
Complexity