

# Building, Running, and Visualizing Parallel NEURON Models

Robert A. McDougal

Yale School of Medicine

10 November 2017

## Why use parallel computation?

Three reasons:

- Get the results for a simulation in less real time.
- Run a larger simulation in the same amount of time.
- 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

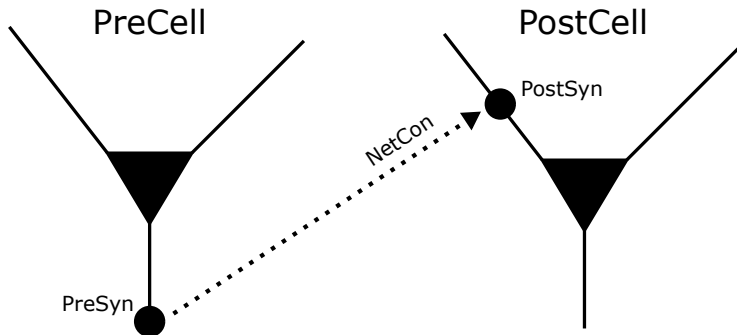
The *multisplit* method distributes portions of the tree cable equation across different machines.

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.

# Synaptic connections with one processor



```
nc = h.NetCon(PreSyn, PostSyn, sec=presyn_section)
nc.delay = 1
```

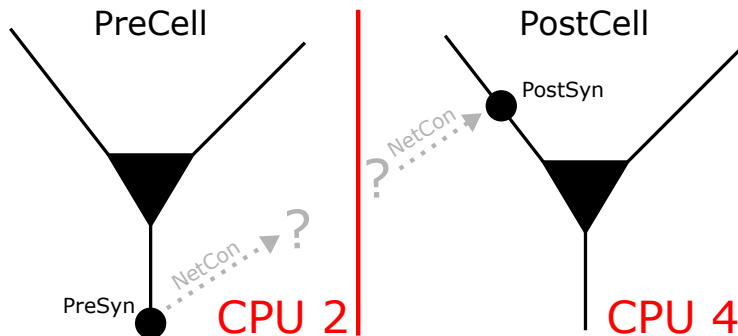
---

Delay is measured in ms.

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

`PreSyn` is a pointer, e.g. `soma(0.5).ref_v`; `PostSyn` is a point process e.g. an instance of `h.ExpSyn`.

If cells in different processes, a different approach is needed

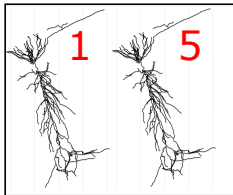


The ParallelContext object facilitates building parallel models.

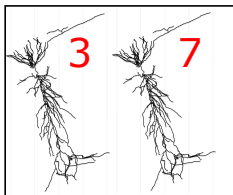
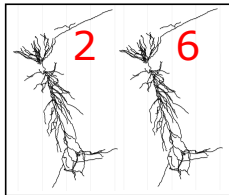
```
pc = h.ParallelContext()
```

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

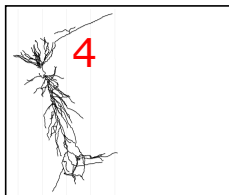
Processor 1



Processor 2



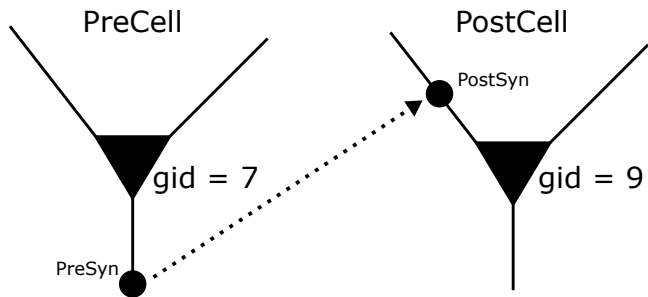
Processor 3



Processor 4

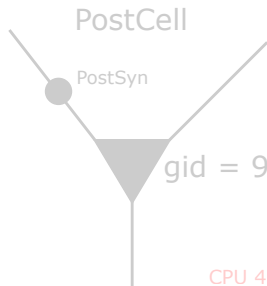
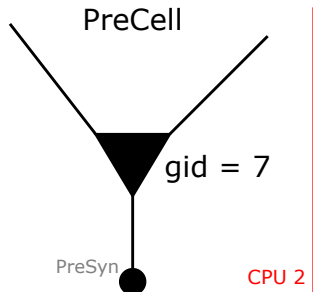
Note: to ensure the model produces identical results regardless of the number of processors, also use GIDs to selecting random streams (e.g. Random123).

# Building synapses





# Configuring the presynaptic connection site



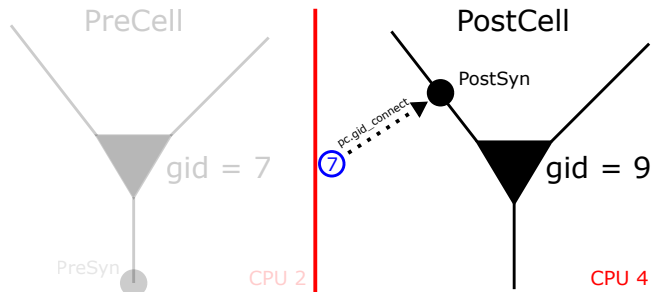
Create cell only where the gid exists:

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

Associate gid with spike source:

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

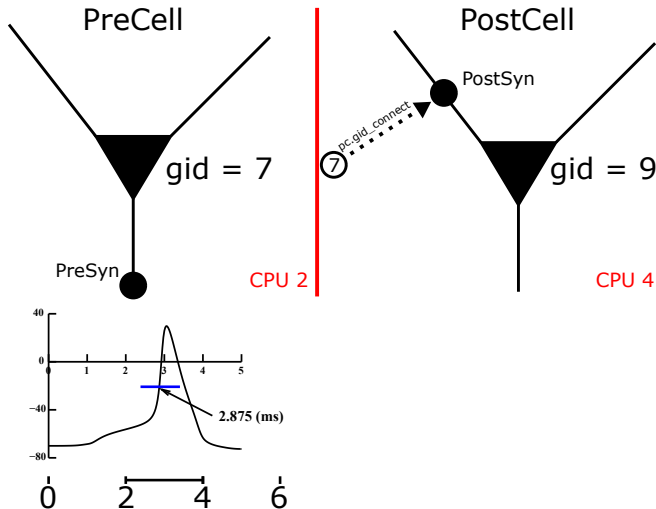
# Configuring the postsynaptic connection site



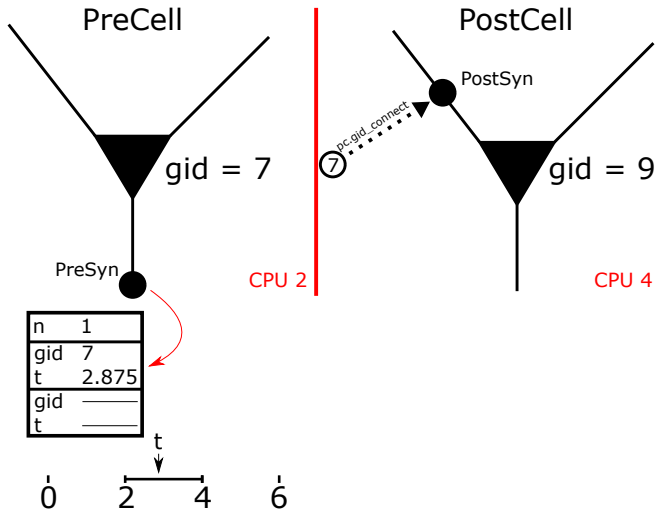
Create NetCon on node where target exists:

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

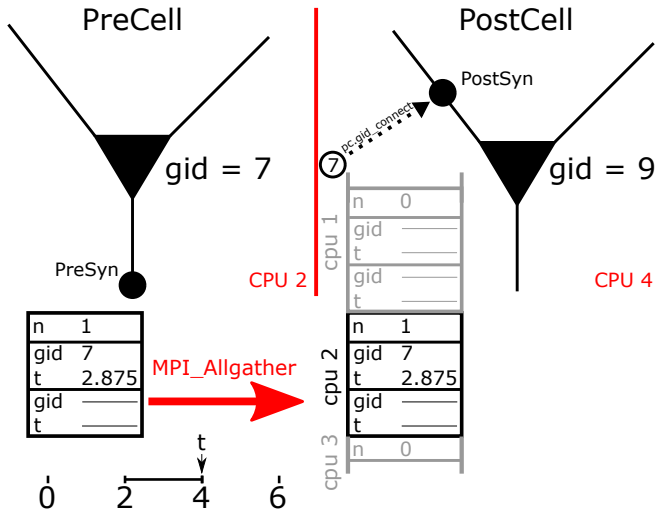
# Spike exchange method



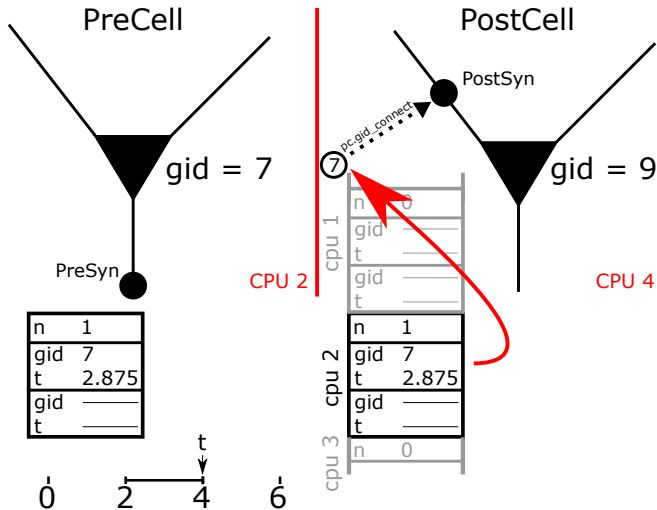
# Spike exchange method



# Spike exchange method



# Spike exchange method

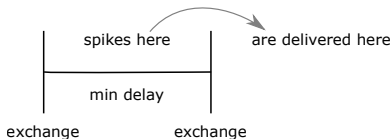


# Exploit transmission delays: using `pc.set_maxstep`

Run using the idiom:

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

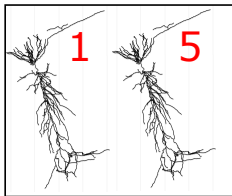
NEURON will pick an event exchange interval equal to the smaller of all the NetCon delays and of the argument to `pc.set_maxstep`. In general, larger intervals are better because they reduce communication overhead.



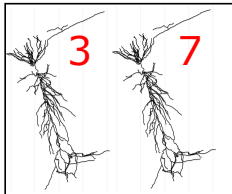
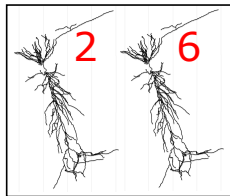
`pc.set_maxstep` must be called on each node; it uses `MPI_Allreduce` to determine the minimum delay.

# Simple load-balancing strategy: round-robin.

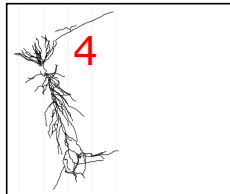
Processor 1



Processor 2



Processor 3



Processor 4



## Simple load-balancing strategy: round-robin.

### CPU 0

pc.id            0  
pc.nhost        5  
ncell           14

...

### CPU 3

pc.id            3  
pc.nhost        5  
ncell           14

### CPU 4

pc.id            4  
pc.nhost        5  
ncell           14

gid

0  
5  
10

gid

3  
8  
13

gid

4  
9

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

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

(Note: the body is executed at most  $\lceil \text{ncell}/\text{nhost} \rceil$  times, not `ncell`.)

# Advanced load-balancing: balance work not number of cells

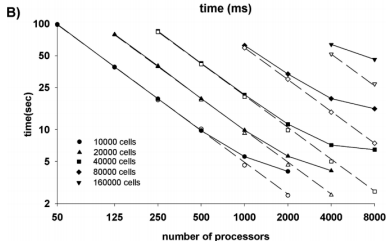
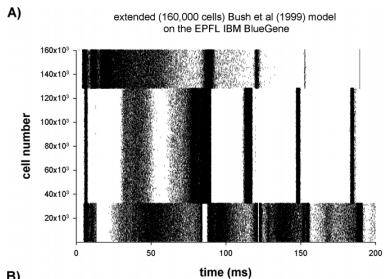
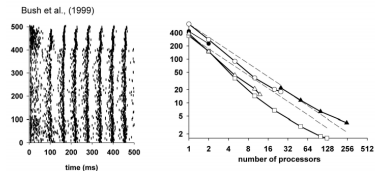
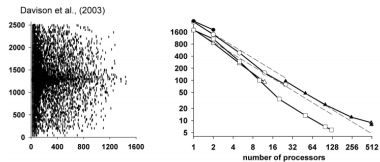
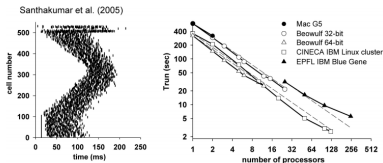
## Strategy:

- 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.

# Performance: MPI scaling

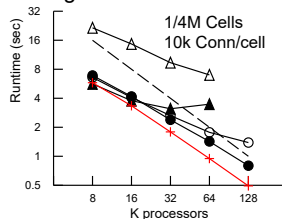
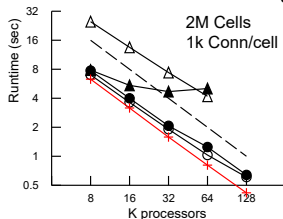


# Performance: Spike exchange strategies

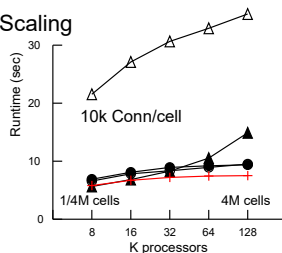
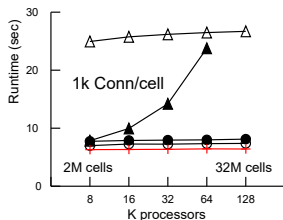
- △ MPI\_ISend – Two Phase, Two Subinterval
- ▲ Allgather
- DCMF\_Multicast – Two Phase, Two Subinterval
- Record-Replay – One Subinterval
- + Computation Time (includes queue)

Artificial Spiking Net  
Blue Gene/P  
Argonne National Lab

## Strong Scaling



## Weak Scaling

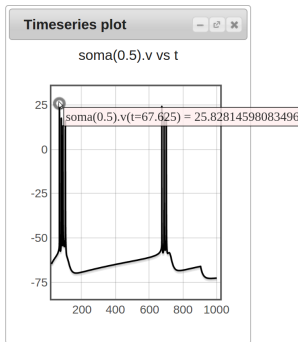
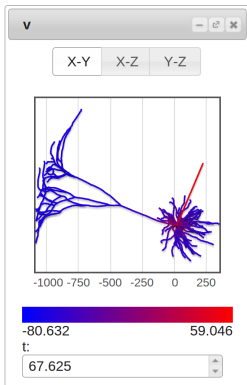


## Performance Tip

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

# Tip: Store synaptic events; recreate single cells as needed

initial conditions  
+  
synaptic events  $\rightarrow$  neuron dynamics

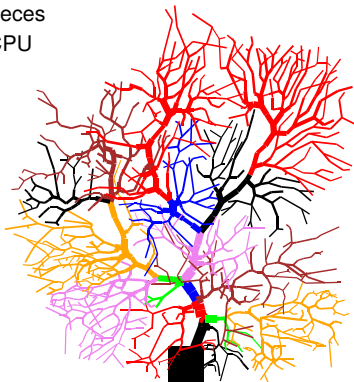


Use `NetCon.record` method to store spike times. Save them as e.g. JSON. Play them back into a single cell simulation using `VecStim`.

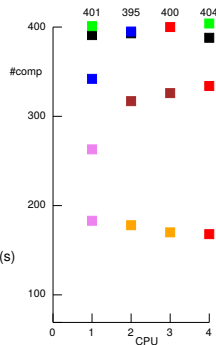
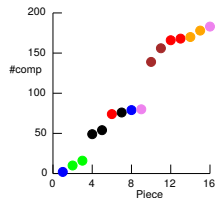
Multisplit

# Improve load balancing with multisplit

16 Pieces  
4 CPU



CPU	Time (s)			Runtime(s)
	Computation	Exchange		
0	13.82	0.56	16 pieces, 1 cpu	55.0
1	13.35	1.03	wholecell, 1 cpu	56.2
2	13.47	0.90	16 pieces, 4 cpu	14.4
3	13.56	0.82		





# 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.

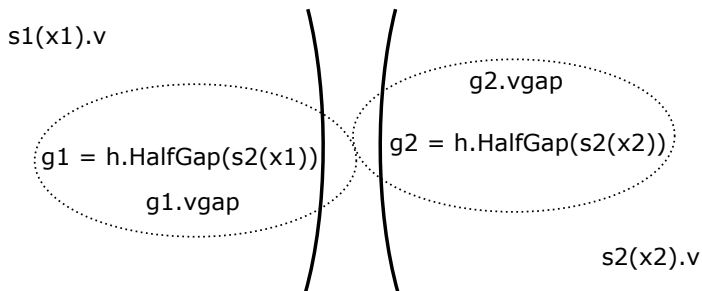
Rules:

- 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.

## 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

pc.source\_var(s1(x1).\_ref\_v, 1)

s1(x1).v  $\longleftrightarrow$  sgid<sub>1</sub>

g1 = h.HalfGap(s2(x1))

g1.vgap

g2.vgap

g2 = h.HalfGap(s2(x2))

sgid<sub>2</sub>  $\longleftrightarrow$  s2(x2).v

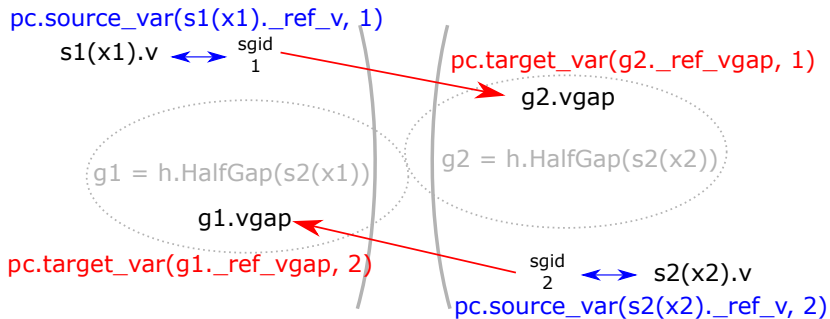
pc.source\_var(s2(x2).\_ref\_v, 2)

## 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 }
```

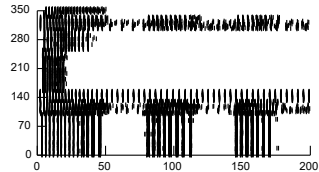
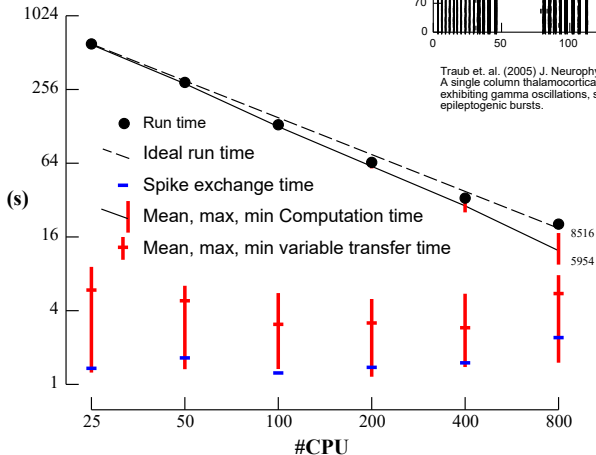
# Performance: Traub model

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

