Parallel Simulations with NEURON

Installation (pages 2-3)
Tutorial notes: conceptual (pages 4-11)
Performance (pages 12-13)
Debugging (pages 14-15)
Ring example (pages 16-19)

Parallel Examples

Parallel network simulations with NEURON (Migliore et al 2006)
http://senselab.med.yale.edu/modeldb/showmodel.cshtml?model=64229
Translating network models to parallel hardware in NEURON (Hines, Carnevale 2008)
http://senselab.med.yale.edu/modeldb/showmodel.cshtml?model=96444
see: http://neuron.yale.edu/neuron/nrnpubs

Simulation of networks of spiking neurons: A review of tools and strategies (Brette et al 2007).
http://senselab.med.yale.edu/modeldb/showmodel.cshtml?model=83319
These are reminiscent of Vogels and Abbott (2005) J. Neurosci. 25, "Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons.

FORTRAN to NEURON translation of Traub el al (2005)
Single-column thalamocortical network model exhibiting gamma oscillations sleep spindles, and epileptogenic bursts. J Neurophysiol. 2005 Apr;93(4):1829-30.
http://senselab.med.yale.edu/modeldb/showmodel.cshtml?model=45539
But see the most recent parallelized version in the mercurial repository:
http://neuron.yale.edu/hg/z/models/nrntraub/

A sequence of transformations of the Dentate Gyrus network model (Santhakumar et al 2005)
http://senselab.med.yale.edu/modeldb/showmodel.cshtml?model=51781
into an MPI parallel (as well as threadsafe) version is in the mercurial repository:
http://neuron.yale.edu/hg/z/models/santhakumar2005

Further Info
http://neuron.yale.edu/neuron/static/new_doc/index.html
ParallelContext

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Installation

http://neuron.yale.edu/ftp/neuron/versions/alpha (recommended, latest features and bug fixes) or http://www.neuron.yale.edu/neuron/download (standard distribution)

MSWindows

Execute the latest 64 or 32 bit nrn*-setup.exe
(Works with full mswin python packages such as Enthought Canopy or Anaconda)

Test NEURON + MPI by opening the NEURON Group's "mintty bash" window, move to the directory where you installed NEURON and type:

```
mpiexec -n 4 nrniv -mpi test0.hoc
```

Should get 4 lines of the style "I am <i> of 4" where <i> ranges from 0 to 3.
Try leaving out the -mpi argument. Also try

```
mpiexec -n 4 nrniv -mpi -python test0.py
```

Mac OSX 10.7 – 10.11 (El Capitan)

nrn-...-osx.pkg (open and follow instructions)
Installs in /Applications
Verify you can launch nrniv from a Terminal window. If not, try '`nrniv -nopython`'.
The path is something like /Applications/NEURON-7.5/nrn/x86_64/bin
Install openmpi. See http://neuron.yale.edu/neuron/download/troubleshoot_mac_install .
Test by creating the file 'test0.hoc'
http://neuron.yale.edu/hg/neuron/nrn/file/c5ef66325387/src/parallel/test0.hoc

```
mpiexec -n 4 nrniv -mpi test0.hoc
```

Linux

Is mpi installed? ie. `mpiexec -n 2 echo 'hello'`
prints "hello" twice.
If not, install openmpi. e.g. `sudo apt-get install openmpi-dev`
Install NEURON from a deb file (e.g. download latest nrn.*.deb from
http://neuron.yale.edu/ftp/neuron/versions/alpha/
and open with synaptic)
… or install NEURON from sources. (nrn...tar.gz or http://neuron.yale.edu/hg/neuron)
For the nrn source installation there are many possibilities, but all involve --with-paranrn

```
./configure --prefix=`pwd` --with-paranrn –with-nrnpython
```

Test with

```
mpiexec -n 4 nrniv -mpi <nrn sources>/src/parallel/test0.hoc
```

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Parallel Simulations page 3 of 19

Supercomputers are usually special and often require cross-compiling.
e.g. EPFL IBM BlueGene/P 16384 cores 750MHz powerpc each with 512MB

```
../nrn/configure --prefix=`pwd` --without-x --with-nmodl-only
make; make install

./nrn/configure --prefix=`pwd` --enable-bluegeneP \
  –with-paranrn –with-nrnpython --host=powerpc64-suse-linux
make; make install
```

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Tutorial notes: conceptual (pages 4-11)

Performance vs clarity.
Clarity should win but want performance proportional to nhost. Can only get that performance by paying attention to load balance. If balance attainable by an optimum distribution of whole cells then great --- clarity is easy to preserve. Otherwise need to use multsplit. Balance > 95% generally attainable :) Start with round robin (card dealing algorithm) and measure balance. For heterogeneous networks often need to distribute cells using the simple greedy LPT (least processing time) algorithm. Do not worry about communication time for spike coupled nets on supercomputers. “Performance proportional to nhost” should also include setup time.

Clarity.

Specification should be model centric and independent of the specific cell distribution and nhost.

The model should be scalable. Helps with debugging since can quickly setup and run on desktop.

Organization and hierarchical structure of the model should be compatible with your various conceptual views of the model. This is generally dominated by the lowest physical view you are interested in. Good when a small change in concept ends up as a small change in the model.

Implementing a network model consists of:
1. Define cell types
2. Create cells (and distribute them)
3. Connect cells (on the processor where the target exists)
4. Specify stimulation

The most fundamental requirement for spike coupled cells is to watch source cells for a spike triggering event, and deliver the spike event after some delay to target synapses. In the NEURON parallel environment the channel from source cell to target synapse is a NetCon object and is constructed via

```
  netcon = pc.gid_connect(srcgid, target_synapse_object)
```

From this, several things naturally follow.
1) netcon is a NetCon object with delay and weight and must be on the same process as the synapse object (which, of course, is on the same process as the cell).
2) the reason srcgid is an integer and not `srccell.axon(1)._ref_v` is because, generally, the source cell does not exist on the same process as the target cell and integers along with spike times are easy to transfer between machines. Below we will show how to associate srcgid with srccell.axon(1)._ref_v on the process where srccell exists.

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Parallel Simulations page 5 of 19

Except for defining the cell types, Python is preferred for implementing the network model.
  Because very easy to construct dictionaries that associate a property with a cell.
  Can also define cell types in Python, but when creating a section, must tell the section which cell it belongs to.

ParallelContext
http://neuron.yale.edu/neuron/static/new_doc/modelspec/programmatic/network/parcon.html

```
from neuron import h
pc = h.ParallelContext()
rank = int(pc.id())
nhost = int(pc.nhost())
```

rank is the uniqe integer in range(0, nhost) that identifies the process. nhost is the number of processes. ParallelContext is mostly a namespace object that contains all the methods relevant to parallel operations. Those operations serve the function of
  Bulletin Board
  Spike coupled nets
  Voltage transfer (gap junctions)
  Threads
  A few MPI collective wrappers.

The above is used so often that we often put the above fragment in common.py and, in every other file,

```
from common import h, pc, rank, nhost
```

Why take the trouble to wrap rank and nhost? Partly for debugging purposes. At the most fundamental level, the only distinction between processes is the rank. Specific cells and connections are only implicitly constructed based on the rank. We try to describe the model in terms completely independent of the distribution of cells on ranks and nhost. Sometimes it is useful on the desktop to pretend the single debugging process we use has a user specified nhost and rank. Only occasionally do we use the explicit pc.id() –-- for example in printing progress information such as :

```
  if pc.id() == 0: print ('setuptime = %g' % setuptime)
```

Want same result no matter how many machines or how cells distributed.
  even with random connections   even with random stimulation
Distribute cells on machines for load balance.
Target centric. Looking to use:

```
 netcon = pc.gid_connect(srcgid, synapse_object)
```

on machine where synapse exists.

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Parallel Simulations page 6 of 19

GID distribution

Round robin

```
gids = [gid for gid in range(rank, ncell, nhost)]
```

GID distribution (instantiate)

```
for gid in gids:
  pc.set_gid2node(gid, rank)
```

eg.

```
>>> rank=2; nhost=5; ncell=20
>>> print ([gid for gid in range(rank, ncell, nhost)])
[2, 7, 12, 17]
```

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Cell Types

Exactly the same as for the serial program

Cell creation

```
cells = {}
for gid in gids:
  # somehow figure out what class and parameters to
  # use based on gid.
  cell[gid]= ...
  netcon = cells[gid].connect2target(nil)
  pc.set_gid2node(gid, rank)
  pc.cell(gid, netcon)
  # netcon does not have to stay in existence after pc.cell
```

Real cells should be instantiated from a class and have connect2target method.
Artificial (spiking) cells can be bare.

```
cells[gid] = h.IntFire1()
```

The cells dictionary is very useful. NEURON also supplies a built-in dictionary.
If you know a gid you can:

1) know if the cell is on this machine

```
    if pc.gid_exists(gid):
```

2) get a reference to the cell

```
    cell = pc.gid2cell(gid)
```

It has not so far been essential that a cell contain a field that specifies the gid.
But iteration over all the gids on a rank is ubiquitous.

```
    for gid in gids:
        cell = cells[gid]
        rs = ranstreams[gid]
        ... other things in dicts indexed by gid ...
```

Of course, dicts, sets, lists, tuples, and your own property classes can be used according to taste.

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Network Connections

It seems every network is unique, but if you can imagine making the connections in a serial model, it is about as straightforward to make them in a parallel model. The goal is to make setup speed proportional to nhost, and that is easiest if the connection rules fit into the idiom:

```
for targetgid in gids:
  # iterate over all (global) srcgid:
    #able to determine if there should be a connection
    #between srcgid and targetgid and all the properties of
    #the connection (including synapse creation).
```

NetCon needs to be on the same rank as the synapse.
Need an nclist to store all the NetCon objects.

For networks of stylized cells, synapses are often created when the cell is created and the synapse (usually a generalized synapse so that it can receive inputs from many sources) is known by its index.

```
for targid in gids:
  cell = pc.gid2cell(targid)  # or gids[targid]
  for srcgid in range[ncell]:
    ... if there should be a connection, determine the synlist
              index, weight, and delay (some may be random) ...
    netcon = pc.gid_connect(srcgid, cell.synlist.o(index))
    nclist.append(netcon)
    netcon.weight = ...
    netcon.delay = ...
```

Generally, a synapse is created when the NetCon is created, the synapse type is determined by source cell type, and the synapse location is determined by both source and target cell properties. Complex connection rules are more easily implemented in Python than HOC because of Python dictionaries. HOC is limited to its internal {gid : cell} dictionary (the first purpose of gids is to implement spike exchange) and it might be difficult or impossible to design a gid for which there are simple administration functions:

```
property = gid2prop(gid)
gid = prop2gid(property)
```

(especially if the gid does not exist on the rank). However, in Python it is straightforward for each rank to create {property : object} dictionaries for objects that they “have” and then send those objects to the ranks which are interested in a specific set of property values.

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Parallel Simulations page 9 of 19

The (possibly) worst case is to find for each targid, the set of srcgids that satisfies
f(srcgid, targid) == True
when the function requires information that exists only on srcgid and targid ranks.

Naively, ncell^2 evaluations might be needed but we want to do it in time proportional to (ncell^2)/nhost (or, if we are lucky, (ncell/nhost)^2 time) in circumstances where no rank has enough memory to hold a nhost size list of data.
This is generally accomplished with the aid of rendezvous rank functions.

For example, suppose ncell gids are randomly distributed on nhost ranks, each gid has a random (x,y,z) position property, and we want to give each target gid a set of source gids that are within distance d of the target. If the network volume is divided into nhost cubic voxels and d is smaller than the edge size of a voxel, then everyone only needs to send their (rank, targid, (x,y,z)) data to the rank responsible for the voxel containing (x,y,z) and their (rank, srcgid, (x,y,z)) data to the 3x3x3 voxel group where (x,y,z) is in the center voxel. Each rendezvous rank can now compare its (on the order of) ntarget/nhost targets with the 27*nsrc/nhost sources and send back the satisfying srcgid set to the target rank for each targid. The “elementary” transfer operation that underlies the algorithm is

```
destlist = pc.py_alltoall(srclist)
```

where destlist and srclist are nhost size lists of python objects (None is a possible object). The jth object on rank i becomes the ith object on rank j. In this case, 3 py_alltoall operations are required.

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

running

```
def prun():
  pc.set_maxstep(10)
  runtime=startsw()
  h.stdinit()
  pc.psolve(tstop)
  if rank == 0 print ('runtime=%g' %(startsw() - runtime))
```

Always save spikes.

```
spiketime = h.Vector()
spikegid = h.Vector()
pc.spike_record(-1, spiketime, spikegid)
```

Save spikes to a file at end of run. (Note the serialization idiom).

```
def spike2file():
  pc.barrier()
  for r in range(nhost);
    if r == rank:
      outf = open(“out.dat”, 'a' if rank else 'w')
      for i in range(len(spiketime)):
        outf.write(“%g %d\n” % (spiketime.x[i], spikegid.x[i]))
      outf.close()
    pc.barrier()
```

Postprocess the file with the sortspike script. Basically,

```
sort -k 1n,1n -k 2n,2n out.dat > out.spk
```

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Repeatable random simulations

A separate random stream for each cell using Random123

```
r[gid] = h.Random()
r[gid].Random123(gid, 0, 0)
```

Use the last two integers if you need more than one conceptual random stream per cell.
E.g. cell synapse id, several streams for distinct purposes.
Able to reproduce or do a statistically independent run by specifying a conceptually global run_number variable, e.g before doing anything requiring randomness:

```
h.Random().Random123_globalindex(run_number)
```

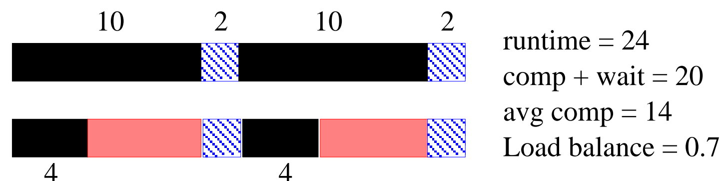
Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Performance Statistics

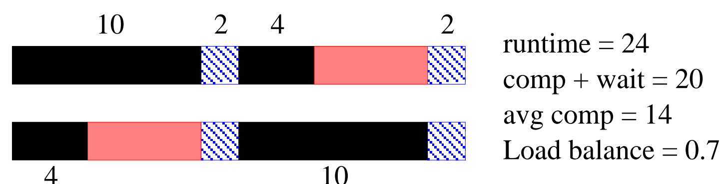
Always at least measure:

```
# first thing in init.py
begintime = h.startsw()
 ...
# after model setup
h.load_file("loadbal.hoc")
lb = h.LoadBalance()
cpu_cx = lb.cpu_complexity()
max_cx = pc.allreduce(cpu_cx, 2)
avg_cx = pc.allreduce(cpu_cx, 1)/nhost
expected_load_balance = avg_cx/max_cx
# just before calling prun()
setuptime = startsw() - begintime

def prun():
   ...
  runtime=startsw()
  h.stdinit()
  pc.psolve(h.tstop)
  runtime = startsw() - runtime
  computation_time = pc.step_time()
  cw_time = computation_time + pc.step_wait()
  max_cw_time = pc.allreduce(cw_time, 2)
  avg_comp_time = pc.allreduce(computation_time, 1)/nhost
  load_balance = avg_comp_time/max_cw_time
  return runtime, load_balance, avg_comp_time
```



The load balance calculation is accurate even if there is significant dynamic spike handling imbalance on a per integration interval basis.



Sometimes useful to get more detailed statistics.

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Efficiency

Setup: for the outer loop of connections, iterate over the gids on this rank.

Run: load balance (so that time to compute each subnet for maxstep time on each machine is as similar as possible.)

1) Round robin gid distribution
2) `h.load_file("loadbal.hoc")`
   `lb = h.LoadBalance()`
3) for each of the gids on this process, store the gid and
   `cx = lb.cell_complexity(pc.gid2cell(gid))`
   into gidvec and cxvec
4) `pc.gid_clear()` and destroy the model (first nclist then cells list).
5) Gather all the gidvec and cxvec (using pc.alltoall or pc.py_alltoall) to rank 0 and
   `rankvec = lb.lpt(cxvec, nhost)`
   on rank 0 where rankvec.x[i] is the rank where gidvec[i] should be located.
6) Sort the gidvec according to rankvec and send each gidvec subset to the proper rank.
7) Store gidvec for each rank in a file so 1-6 do not have to be done again.
8) Each rank constructs its balanced gidvec portion of the model.

If rate limited by spike exchange
and using fixed step method so spikes on fixed time step boundaries then
`pc.spike_compress(nspike, gid_compress)`
(nspike>0 and gid_compress=1 means 2 bytes per spike instead of at least 12)
"optimal" choice of nspike aided by perusal of spike exchange histogram statistics.
`pc.max_histogram(histvec)`
E.g. the Traub model with nhost=256, ncell=3560, and integration interval = 0.05 shows
histogram of #spikes vs #exchanges

```
0	40
1	708
2	1006
3	220
4	26
5	1
```

end of histogram

If rate limited by the event queue
due to heavy use of self events, set second arg to 1 of
`h.cvode.queue_mode(use_fixed_step_binqueue, use_self_queue)`
and if you don't mind events being on fixed dt boundaries
set that first arg to 1

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Debugging

Design the model so that ncell is a parameter and does not have to be large.

SAVE the network spike pattern into a (spiketime gid) file, out.dat.

What to do when

```
  sortspike  out.dat  out.np
  diff out.1 out.np
```

says the files are not identical.

What is the spiketime and gid of the first difference? Focus on that gid.
Is the problem in the spike input to the cell or the cell parameters/initialization?

Print all parameters, states, synapses, incoming netcons for that gid.

```
prun(time_just_before_first_difference)
pc.prcellstate(gid, "t%g_%d"%(h.t, nhost))
```

will create an outputt file of form <gid>_t<t>_<nhost>`.nrndat`

Do for earlier times til the files are the same and see what variable first goes bad.

When searching for what variable first goes bad, usually a good idea to first check at `h.t=0` just after `stdinit()`.

See the prcellstate file format at
http://neuron.yale.edu/neuron/static/new_doc/modelspec/programmatic/network/parcon.html#ParallelContext.prcellstate

Can run on a serial machine that computes exactly the same thing as though it were a particular rank on an nhost machine.
(If you use rank and nhost parameters as much as possible instead of pc.id() and pc.nhost())

Use PatternStim

```
    class ApplyPattern:
      def __init__(self, fname):
        spikefile = open(fname, 'r')

        spiketime = h.Vector()
        idvec = h.Vector()
        for line in spikefile:
          values = [float(x) for x in line.split()]
           spiketime.append(values[0])
```

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

```
            idvec.append(values[1])

        self.pattern = h.PatternStim()
        self.pattern.play(spiketime, idvec)

pat = ApplyPattern(out.spk)
```

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Ring Example (pages 16-19)

ModelDB accession #96444 (original HOC version) or mercurial repository version http://neuron.yale.edu/hg/z/models/ring/ The latter includes an up to date Python implementation as well as subworld implementations for HOC and Python.

NCELL BallStick cells where cell i is connected to cell i + 1 with large weight and cell 0 is stimulated with a NetStim. Cell NCELL-1 connected to cell 0.

```
$ mpiexec -n 4 nrniv -mpi -python ringpar.py
```

excerpts from cell.py

```
class BallStick(object):
  def __init__(self):
    #print 'construct ', self
    self.topol()
    self.subsets()
    …
    self.synapses()

  def topol(self):
    self.soma = h.Section(name='soma', cell=self)
    self.dend = h.Section(name='dend', cell= self)
    self.dend.connect(self.soma(1))
    self.basic_shape()

  def subsets(self):
    self.all = h.SectionList()
    self.all.append(sec=self.soma)
    self.all.append(sec=self.dend)

  def connect2target(self, target):
    nc = h.NetCon(self.soma(1)._ref_v, target, sec = self.soma)
    nc.threshold = 10
    return nc

  def synapses(self):
    s = h.ExpSyn(self.dend(0.8)) # E0
    s.tau = 2
    self.synlist.append(s)
    …
```

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

ringpar.py

```
from neuron import h
h.load_file('nrngui.hoc')
pc = h.ParallelContext()
rank = int(pc.id())
nhost = int(pc.nhost())

from cell import BallStick

# Network Creation

NCELL = 20

cells = []
nclist = []

def mkring(ncell):
  mkcells(ncell)
  connectcells()

def mkcells(ncell):
  global cells, rank, nhost
  cells = []
  for i in range(rank, ncell, nhost):
    cell = BallStick()
    cells.append(cell)
    pc.set_gid2node(i, rank)
    nc = cell.connect2target(None)
    pc.cell(i, nc)

def connectcells():
  global cells, nclist, rank, nhost, NCELL
  nclist = []
  # not efficient but demonstrates use of pc.gid_exists
  for i in range(NCELL):
    targid = (i+1)%NCELL
    if pc.gid_exists(targid):
      target = pc.gid2cell(targid)
      syn = target.synlist[0]
      nc = pc.gid_connect(i, syn)
      nclist.append(nc)
      nc.delay = 1; nc.weight[0] = 0.01

mkring(NCELL)
```

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Parallel Simulations page 18 of 19

#Instrumentation - stimulation and recording

```
def mkstim():
  ''' stimulate gid 0 with NetStim to start ring '''
  global stim, ncstim
  if not pc.gid_exists(0):
    return
  stim = h.NetStim()
  stim.number = 1
  stim.start = 0
  ncstim = h.NetCon(stim, pc.gid2cell(0).synlist[0])
  ncstim.delay = 0
  ncstim.weight[0] = 0.01

mkstim()

def spike_record():
  ''' record spikes from all gids '''
  global tvec, idvec
  tvec = h.Vector()
  idvec = h.Vector()
  pc.spike_record(-1, tvec, idvec)

def prun(tstop):
  ''' simulation control '''
  pc.set_maxstep(10)
  h.stdinit()
  pc.psolve(tstop)
```


Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Parallel Simulations page 19 of 19

```
def spikeout():
  ''' report simulation results to stdout '''
  global rank, tvec, idvec
  pc.barrier()
  for i in range(nhost):
    if i == rank:
      for i in range(len(tvec)):
        print '%g %d' % (tvec.x[i], int(idvec.x[i]))
    pc.barrier()


def finish():
  ''' proper exit '''
  pc.barrier()
  h.quit()

if __name__ == '__main__':
  spike_record()
  prun(100)
  spikeout()
  if (nhost > 1):
    finish()
```

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

Receipt

Received:

From:

For: Parallel Simulation component of NEURON 2016 Summer Course
http://www.neuron.yale.edu/neuron/static/courses/summer2016/summer2016.html

Date:

By: N.T. Carnevale
Director, NEURON 2016 Summer Course
203-494-7381
ted.carnevale@yale.edu

For deposit in: Yale University account "NNC--Fees"

Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved

 Copyright © 1998-2016 N.T. Carnevale and M.L. Hines, all rights reserved