The *Myriad* simulator: parallel computation for densely integrated models

Pedro Rittner , Andrew J. Davies, Thomas A. Cleland[1]

[1]Dept. of Psychology, Cornell University, Ithaca, NY

Computational Physiology Laboratory

451.12 / DD-58

1 Why another simulator?

- Larger network models, especially many biophysical models, often have complex inter- and intra-neuron interactions with multiple non-linear, cyclic dependencies. Such *densely integrated* circuits are poorly optimized in general purpose simulators, devolving into solving multiple stiff equations linearly, step by step.
- Modelers must often trade performance for biophysical accuracy due to general purpose simulators' implementations being difficult to parallelize, often requiring special coding to achieve limited multiprocessing capabilities.
- *Myriad* separates model design and code optimization into two discrete systems connected by powerful code generation middleware, enabling users to fully utilize multithreading capabilities on commodity hardware, GPUs, and clusters.

2 Design overview of the *Myriad* simulator

User Code & Support Libraries (Python)

- Idiomatic Python 3 with optional C code specified verbatim.
- High-level abstractions for neurons, 'sections', synapses, and network properties.
- Mechanisms and other elements (particles, ions, channels, pumps, etc.) are user-definable with object-based inheritance (e.g., channels inherit properties based on the ions that they flux).
- Simulations are represented as objects to facilitate iterative parameter searches and reproducibility of results
- Inheritance functionality via Python's native object system
  - **Automatic access** to properties of parent component
  - Functionality can be **extended & overridden** at will

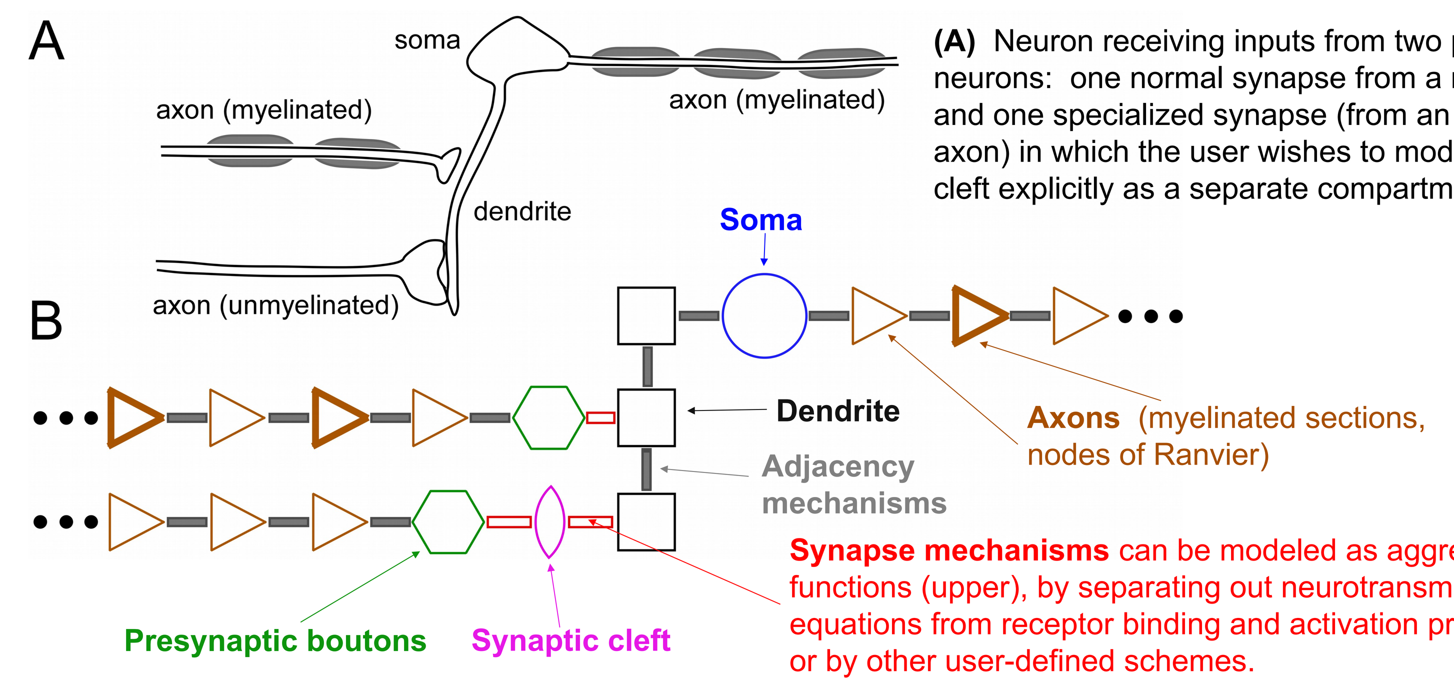
Compilation & Data Transfer Middleware (CPython)

- Mako templates for converting user-defined objects into C and Python-compatible structures for use in simulation
- Removes all hierarchy from compartmental models, recognizing only two computational elements:
  - ***Compartments*** (isometric) with passive properties
  - ***Mechanisms*** that connect exactly two compartments in exactly one direction. (Connections to a common extracellular medium are a special case of this).
- AST-to-AST translation of Python code to C for compiling

Simulation Backend & Optimization (C + CUDA C)

- Each compartment or mechanism is treated as a separate computational element, **enabling parallel execution** on separate CPU threads or GPU CUDA cores. Parameters are updated via shared memory access.
- Synchronization via barrier intrinsics at each timestep ensures safe access to shared data without races/blocking
- Separate compilation phase enables compilers to aggressively inline mechanism and compartment functions
- POSIX message queues used as signaling method for communicating data between the front-end and the simulation executable via Unix Domain Sockets

3 Granularity enables highly flexible model design



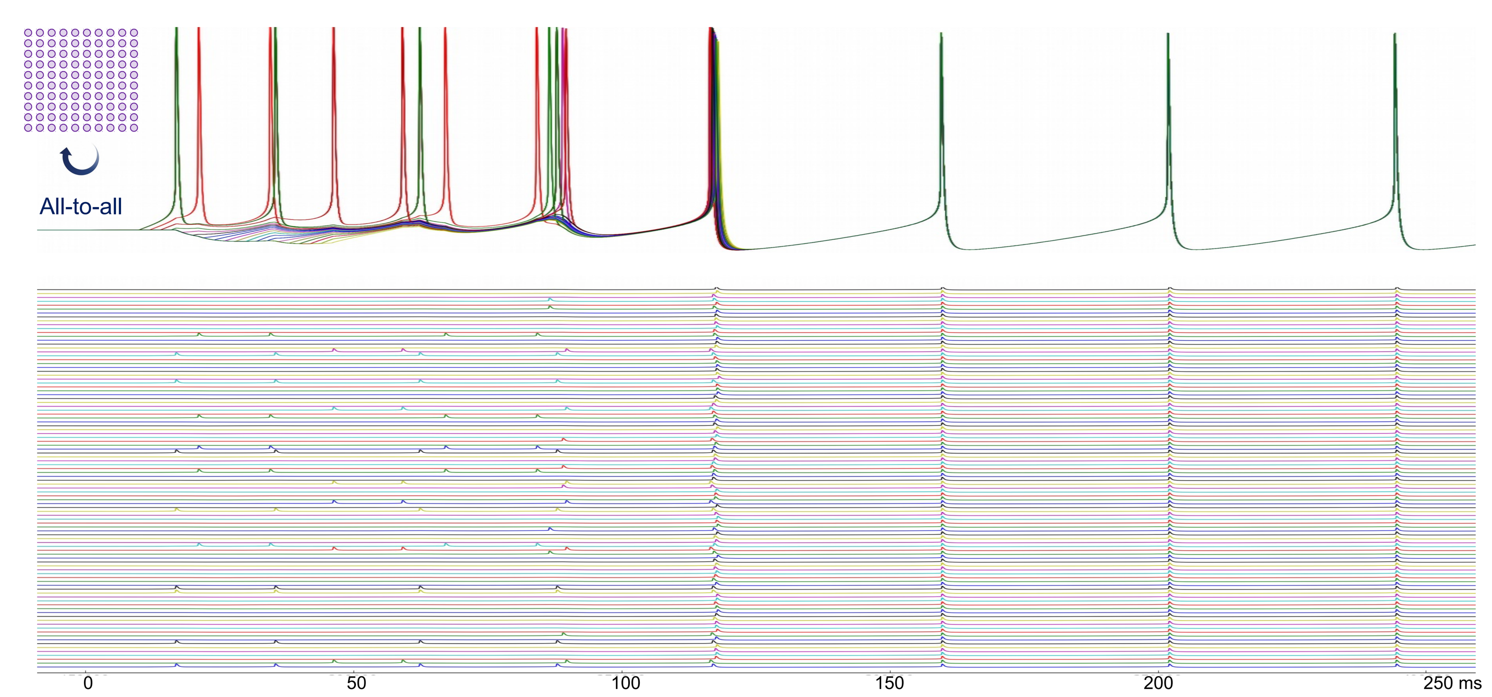
(A) Neuron receiving inputs from two presynaptic neurons: one normal synapse from a myelinated axon and one specialized synapse (from an unmyelinated axon) in which the user wishes to model the synaptic cleft explicitly as a separate compartment.

Synapse mechanisms can be modeled as aggregate transfer functions (upper), by separating out neurotransmitter release equations from receptor binding and activation processes (lower), or by other user-defined schemes.

(B) Schematic example of Myriad compartments and mechanisms. Compartment and mechanism objects incorporate user-defined equations. A single extracellular space compartment here connects to all compartments except the synaptic cleft (multiple extracellular spaces also are supported). *Adjacency* is implemented as *paired* reciprocal mechanisms.

4 Simulation examples

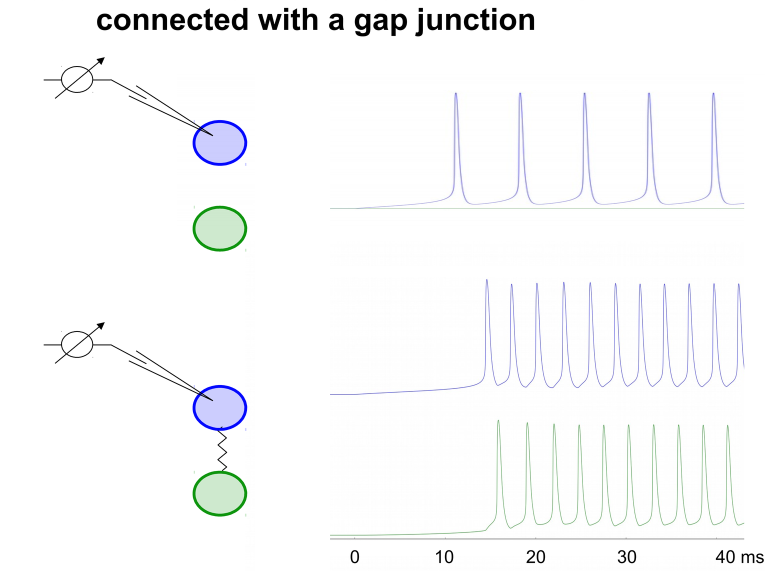
(A) **100 Hodgkin-Huxley neurons coupled with inhibitory synapses form an *interneuron network gamma* (ING) oscillatory network.**



(B) Python metaprogramming example and *user level* script prototype of (A)

```
from myriad import *
from random import choice

class Simul_4B(MyriadSimul):

    def simul_setup(self):
        # Override values from parent object
        self.simul_len = Unit(s, 1.0)
        self.dt = Unit(s, 0.000001) # 1.0µs

        # This loop 'produces' 100 identical neurons
        for i in range(100):
            # Creates the point neuron compartment
            hh_nrn = HHPointNeuron(
                cm = Unit(pF, 1.0),
                init_vm = Unit(mV, -65.0))
            # Creates a leak mechanism
            leak_m = HHLeakCurrMech(
                g_leak = Unit(nS, 1.0),
                e_rev = Unit(mV, -65.0))
            # Creates a sodium mechanism
            na_m = HHNaCurrMech(
                g_na = Unit(nS, 35.0),
                e_na = Unit(mV, 55.0))
            # Creates a potassium mechanism
            k_m = HHKCurrMech(
                g_k = Unit(nS, 9.0),
                e_k = Unit(mV, -90.0))
            # Add all the mechanisms to the neuron
            hh_nrn.add_mech(leak_m)
            hh_nrn.add_mech(na_m)
            hh_nrn.add_mech(k_m)
            # Add the neuron to our compartment list
            self.compartments.add(hh_nrn)

        # This loop connects each neuron to a random other
        for hh_nrn in self.compartments:
            # Choose random other neuron
            other_nrn = choice(self.compartments)
            # Add inhibitory GABAA synapse between neurons
            if other_nrn is not hh_nrn:
                gabaa_syn = HHGABAASyn(
                    hh_nrn, other_nrn,
                    gaba_vm_thresh = Unit(mV, 0),
                    gaba_g_max = Unit(nS, 0.1),
                    gaba_tau_alpha = 0.083,
                    gaba_tau_beta = 10.0,
                    gaba_rev = Unit(mV, -75.0))

# Instantiates the simulation object
my_simul = Simul_4B()
# Runs the simulation; setup is automatic
my_simul.run()
```

(C) Two Hodgkin-Huxley neurons, separated or connected with a gap junction



(D) *User-level* script prototype simulating (C) in the connected case only

```
from myriad import *

class Simul_4C(MyriadSimul):
    # Sets up simulation objects
    def simul_setup(self):
        # Default parameters from Squid Giant Axon
        hh_neuron1 = HHPointNeuron()
        # Exact parameters specified using keywords
        hh_neuron2 = HHPointNeuron(cm = Unit(nF, 1.0))
        # Add inhibitory GABAA synapse from 1 -> 2
        gabaa_syn = HHGABAASyn(hh_neuron1, hh_neuron2)
        # Add neurons to the simulation
        self.compartments.add(hh_neuron1)
        self.compartments.add(hh_neuron2)

# Instantiates the simulation object
my_simul = Simul_4C()
# Runs the simulation; setup is automatic
my_simul.run()
```

5 User extension example: AMPA Synapse with STDP

```
from myriad import *

# Using Model 3 from Sjostrom, Turrigiano, & Nelson
class STDPAMPASyn(HHAMPASyn):

    # Post-synaptic compartment spike threshold
    post_spike_thresh = Unit(mV, 10.0)
    # Pre-synaptic compartment spike threshold
    pre_spike_thresh = Unit(mV, 10.0)
    # State to keep track of time each compartment fired
    t_pre = MDouble(0)
    t_post = MDouble(0)
    # STDP LTP synaptic strengthening factor
    a_plus = MDouble(0.02)
    # STDP LTD synaptic weakening factor
    a_minus = MDouble(0.05)
    # STDP time windows
    tau_plus = MDouble(15.0)
    tau_minus = MDouble(15.0)
    t_ltp_edge = MDouble(15.0)
    t_ltd_edge = MDouble(15.0)
    # Synaptic weight and maximum weight value
    omega = MDouble(2.0)
    omega_max = MDouble(2.5)
    @myriad_method
    def calc_current(self, pre_syn_comp, post_syn_comp, step, global_time):
        # We can get the AMPA current calculation from our parent
        I_AMPA = super().calc_current(pre_syn_comp, post_syn_comp)
        # Weight change (to be calculated)
        d_omega = 0.0

        # Get post-synaptic cell's current/previous membrane voltage
        post_vm_curr = post_syn_comp.vm[step]
        post_vm_prev = post_syn_comp.vm[step - 1]
        # If the post-synaptic cell has spiked
        if (post_vm_curr >= self.post_spike_thresh
            and post_vm_prev <= self.post_spike_thresh):
            # If the presynaptic spike is in our window, strengthen the weight
            if (global_time - self.t_pre <= self.t_ltp_edge):
                delta_t = m_fabs(self.t_pre - global_time)
                d_omega = self.a_plus * m_exp(delta_t / self.tau_plus)
                self.t_post = -INFINITY
            # Otherwise, just set the time we fired (current global time)
            else:
                self.t_post = global_time

        # Get pre-synaptic cell's current/previous membrane voltage
        pre_vm_curr = pre_syn_comp.vm[step]
        pre_vm_prev = pre_syn_comp.vm[step - 1]
        # If the pre-synaptic cell has spiked
        if (pre_vm_curr >= self.pre_spike_thresh and
            pre_vm_prev <= my_stdp_ampa_curr->pre_spike_thresh):
            # If the postsynaptic spike is in our window, weaken the weight
            if (global_time - self.t_post <= self.t_ltd_edge and
                    self->t_pre < self->t_post):
                delta_t = m_fabs(self.t_post - global_time)
                d_omega = self.a_plus * m_exp(delta_t / self.tau_plus)
                self.t_post = -INFINITY
            self.t_pre = global_time

        # Alter the weight based on above calculations
        self->omega += delta_omega

        # Bound the weight between 0 and omega_max
        if (self.omega < 0.0):
            self.omega = 0.0;
        elif (self.omega > self.omega_max):
            self.omega = self.omega_max

        # Return the current times the synaptic weight
        return I_AMPA * self->omega
```

This example inherits from a previously-defined, out-of-source AMPA synapse **mechanism.**

Here, the user defines several state variables that are *owned* by the STDPAMPASyn mechanism, **where each instance has its own copy.**

Default values are provided here, though these are optional. The constructor is elided by the middleware.

Values are expressed as either of *Unit* or *MDouble*, a generic unitless type.

The parser uses the **@myriad_method** annotation (highlighted in a red rectangle) in order to know which methods to convert at parse-time.

6 Planned Extensions

- Docker support for facilitating deployments is planned on release
- Implement simulation governor to run multiple instances in series or in parallel (e.g., on distributed-architecture GPU clusters), to support parameter exploration and algorithmic optimization.
- Myriad is an arbitrarily programmable GPU-enabled computational framework that is in principle as appropriate for (e.g.) 3-D spatial diffusion models as for neuronal modeling. Assess Myriad's utility for these different applications, and their synthesis.
- Extend Myriad to a nonuniform memory access architecture to support multiple CUDA cards on a single high-speed bus (NVLink).

7 References & Acknowledgments

Rittner P, Cleland TA (2014) The MYRIAD simulator: densely coupled realistic neural networks on GPU. *GPU Technology Conference*, San Jose, CA. http://www.gputechconf.com/

Carnevale NT, Hines ML (2006) The NEURON Book. Cambridge, UK: Cambridge University Press.

This research was supported by an equipment grant from the NVIDIA Corporation.