Sandia National Laboratories

Exceptional service in the national interest

Simulation Tool for Asynchronous Cortical Streams (STACS)

Overview and Tutorial

Felix Wang

NICE 2024 Tutorials

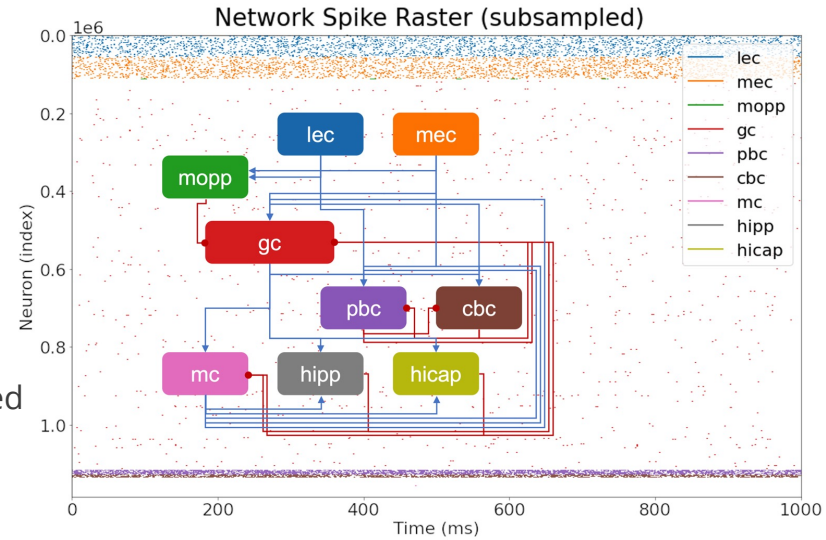
https://github.com/sandialabs/STACS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2024-05196C

Simulation Tool for Asynchronous Cortical Streams (STACS)

- STACS is a large-scale spiking neural network simulator built on top of the Charm++ parallel programming framework (workload decomposed as parallel objects)
  - Network models are represented as a directed graph, and neuron and synapse model dynamics are formulated as stochastic differential equations:
    - $dx = f(x)dt + \sum_{i=1}^{n} g_i(x)dN_i$: time-driven computation, event-driven communication
  - Simulation is supported by partition-based data structures, and serialized through SNN extensions to the distributed compressed sparse row (SNN-dCSR) data format
    - These facilitate network generation, graph-aware partitioning, checkpoint/restart, scalable multicast communication, and tool interoperability
  - STACS may be used as a standalone simulator, as a backend to tools like N2A or Fugu, and also interface with external devices through YARP
- Available at: https://github.com/sandialabs/STACS



Simulation of a biologically inspired spiking neural network of about 12M neurons and 70B synapses

Where STACS sits within the neuromorphic ecosystem

- Description Languages
  - PyNN, NeuroML, NineML, etc.
- Software Frameworks
  - N2A, Fugu, Lava, Nengo, etc.

STACS is able to interoperate with software frameworks such as N2A or Fugu through:

- Translating between higher-level network description languages (N2A → YAML)
- As a simulation backend for instantiated networks (Fugu → NetworkX → SNN-dCSR)

- Network Simulators
  - NEST, NEURON, Brian, GeNN, etc.

STACS is primarily a spiking neural network simulator

- Data Formats
  - NIR, SONATA, NetworkX, GEXF, etc.
- Hardware Platforms
  - Loihi, SpiNNaker, BrainScaleS, etc.

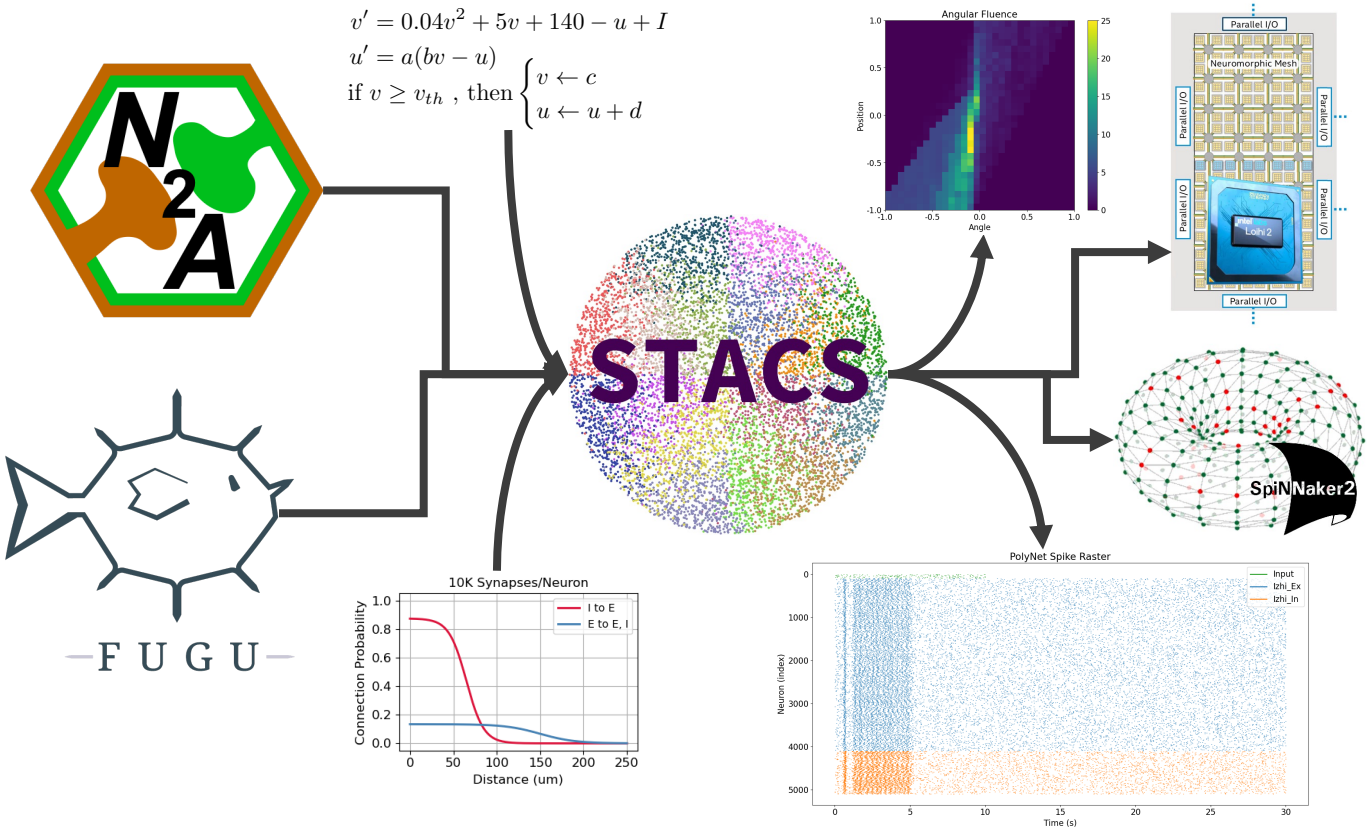
The partition-based SNN-dCSR data format supports external tool interoperability:

- Such as graph partitioners (e.g. ParMETIS), and network analysis (e.g. GNATFinder)
- It also provides a path toward mapping instantiated networks to different neuromorphic hardware platforms

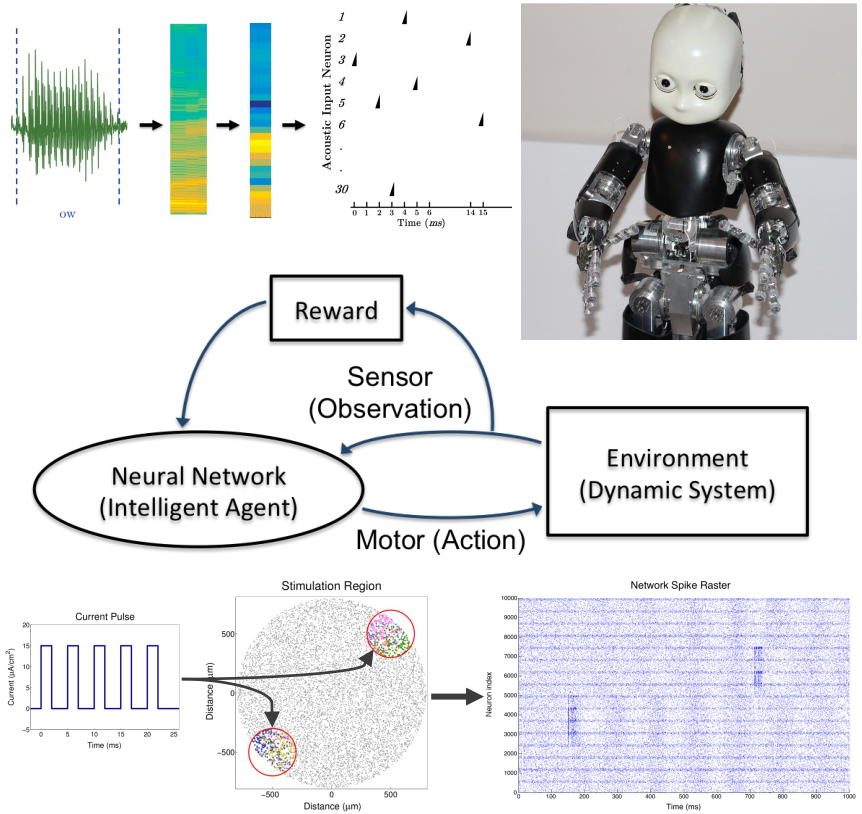
Selected features of STACS

STACS supports user-defined model dynamics (structured as SDEs through an abstract class), which enables algorithm and model exploration, or the emulation of hardware implementations.

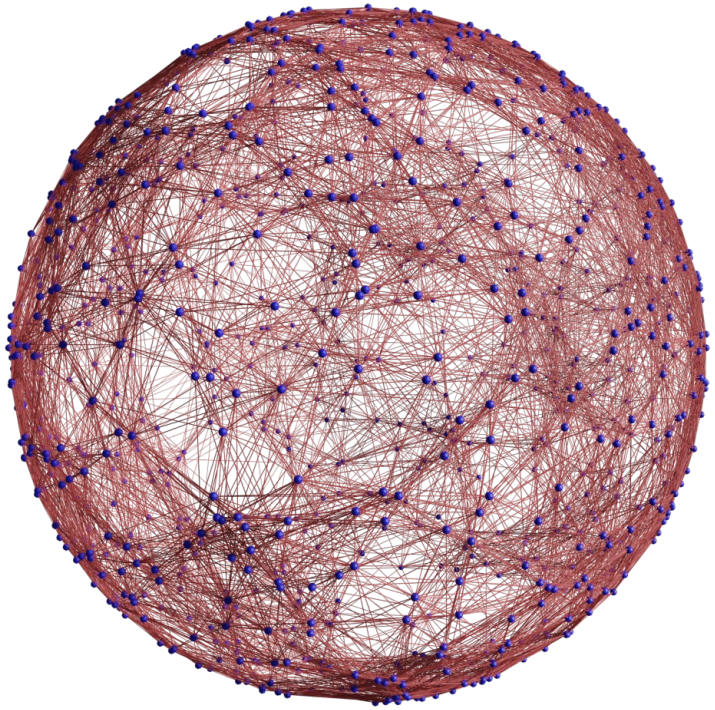
$$v' = 0.04v^2 + 5v + 140 - u + I$$
$$u' = a(bv - u)$$
$$\text{if } v \geq v_{th} \text{ , then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases}$$



STACS also supports interfacing with devices through YARP, which enables closed-loop system exploration and external communication and control.

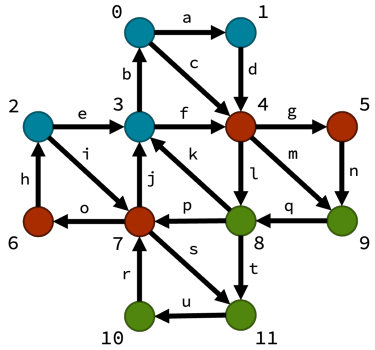


Selected features of STACS



STACS supports network generation over spatially-defined topologies. Here, a visualization of a 6000 neuron network over a sphere.

The SNN-dCSR format supports network (re)partitioning, making it suitable for computational parallelism, whether its target platform is between nodes on an HPC system or between cores/chips on neuromorphic hardware.

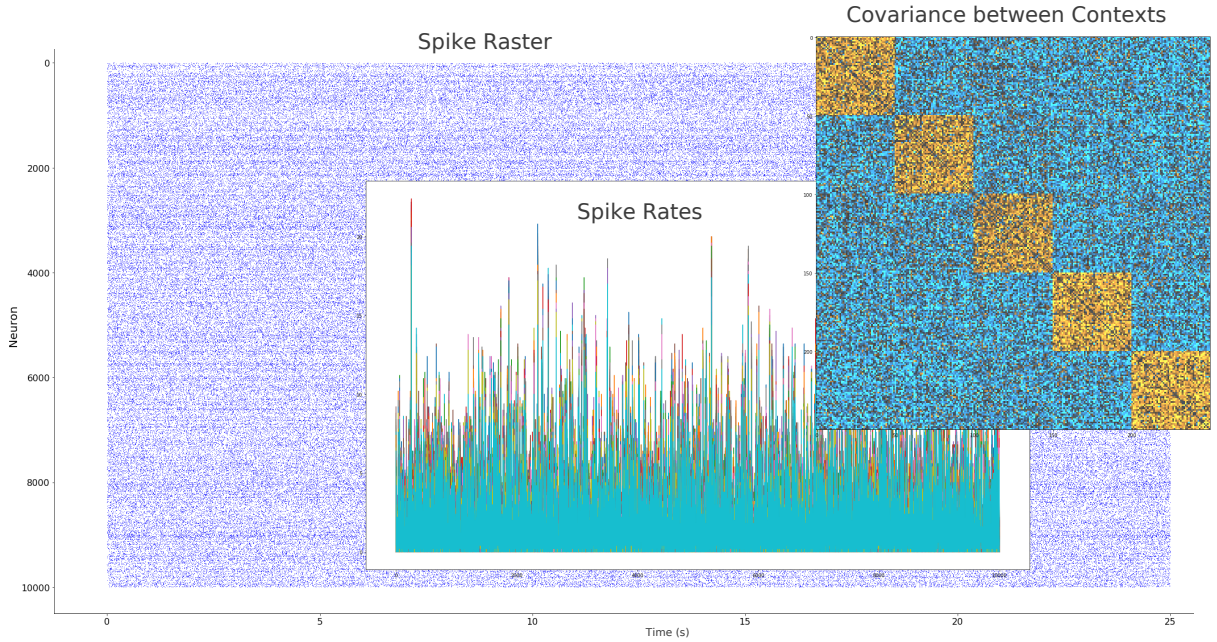


dist	adjcy.0	adjcy.1	adjcy.2
0 0	1 3 4	0 1 3 5 8 9	3 4 7 9 11
4 13	0 4	4 9	4 5 8
8 29	3 6 7	2 7	7 11
12 42	0 2 4 7 8	2 3 6 8 10 11	7 8 10

state.0	state.1	state.2
0 Ø b Ø	4 c d f Ø Ø Ø	8 Ø l Ø q Ø
1 a Ø	5 g Ø	9 m n Ø
2 Ø h Ø	6 Ø o	10 Ø u
3 Ø e Ø j k	7 i Ø Ø p r Ø	11 s t Ø

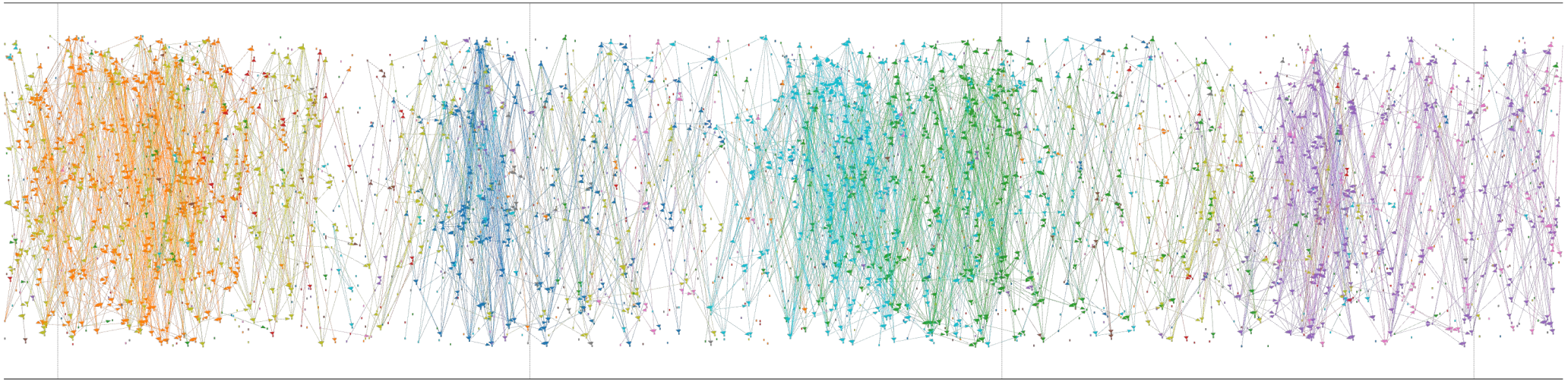
Example 3-way partitioning of a simple network with 12 vertices and 21 directed edges using the SNN-dCSR format. Directed edges with associated state are bolded.

Selected features of STACS



Network snapshots, state probes, and spike events can be recorded from large-scale and long-running simulations for sophisticated offline analysis. Computing the separability of spike rates between contexts (left). Identifying recurrent causal activity patterns through a combination of network structure and spiking activity (below).

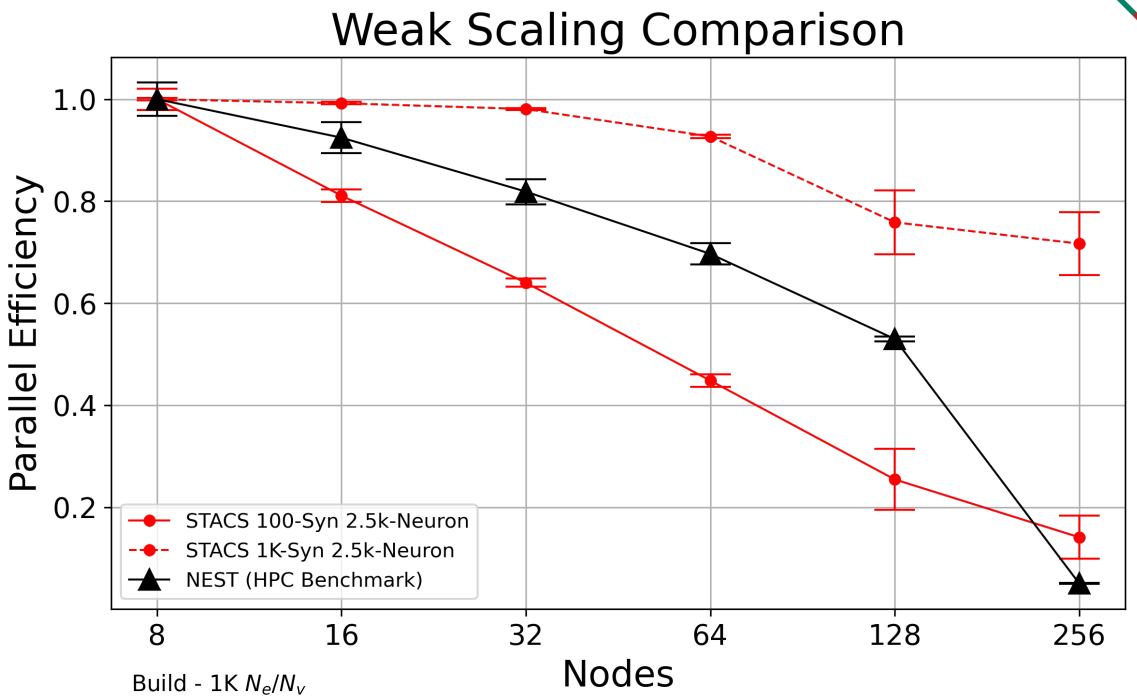
Graphical Neural Activity Threads (GNATs)



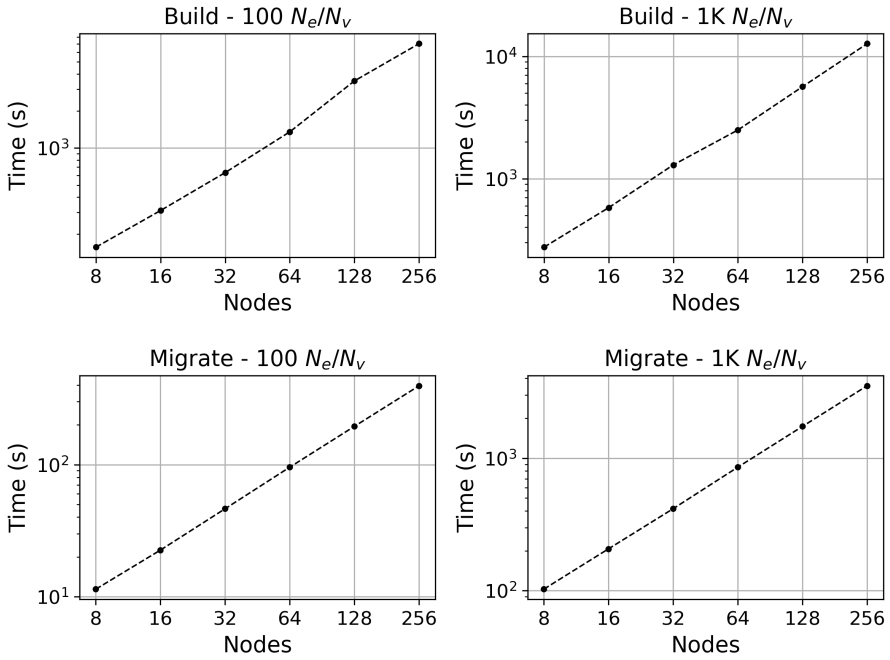
Selected features of STACS

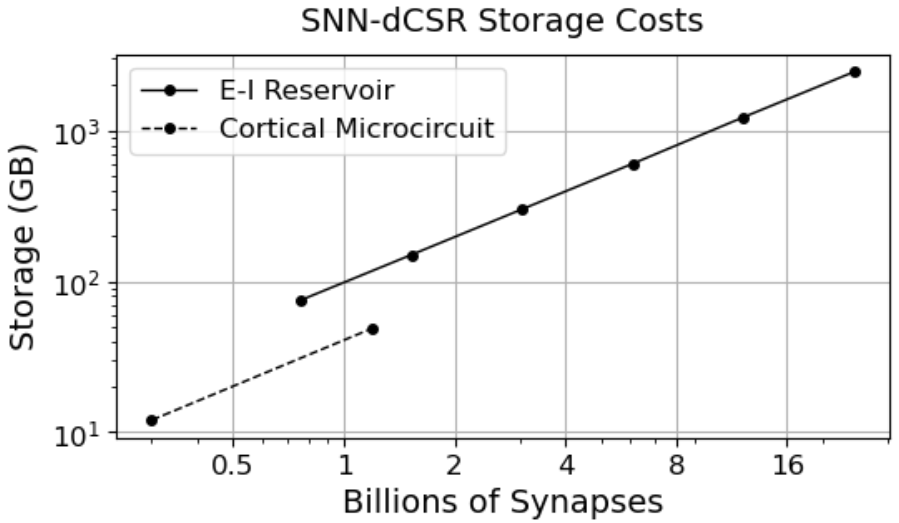
STACS provides scalable parallel methods for:

- Graph generation (building the network)
- Repartitioning (migrating neurons for better hardware mapping)
- Simulation (with competitive simulation time parallel efficiency)
- Serialization (allowing for neuromorphic software tools interoperability)



Scaling experiments with a spatially parameterized E-I reservoir model showed *O(n)* build times, vertex migration times, and storage costs





TUTORIAL

Tutorial Overview

- Installation
- Quick start example
- Overview of basic configuration files (using predefined models)
  - Configuring neuron and synapse parameters
  - Configuring network connectivity structure
  - Configuring a simulation
  - Building a network and running a simulation
- Overview of intermediate functionality
  - Python interface and inputs
  - Defining new neuron models
  - Defining new synapse models
- Limitations, future work

Installation (Environment)

- The hardest part of installing STACS is setting up the Charm++ environment
  - Charm++ is actually quite portable, and can run on basically any machine

- Setting up the dependencies/environment (Linux, MacOS)
  - General development environment
    - `sudo apt-get install build-essential gfortran cmake`
    - Install Homebrew: https://brew.sh/
    - `brew install wget htop gcc cmake git`
  - STACS specific libraries
    - `sudo apt-get install libyaml-cpp-dev libfftw3-dev`
    - `brew install yaml-cpp fftw`
  - MPI (communication method)
    - `sudo apt-get install mpich`
    - `brew install mpich`

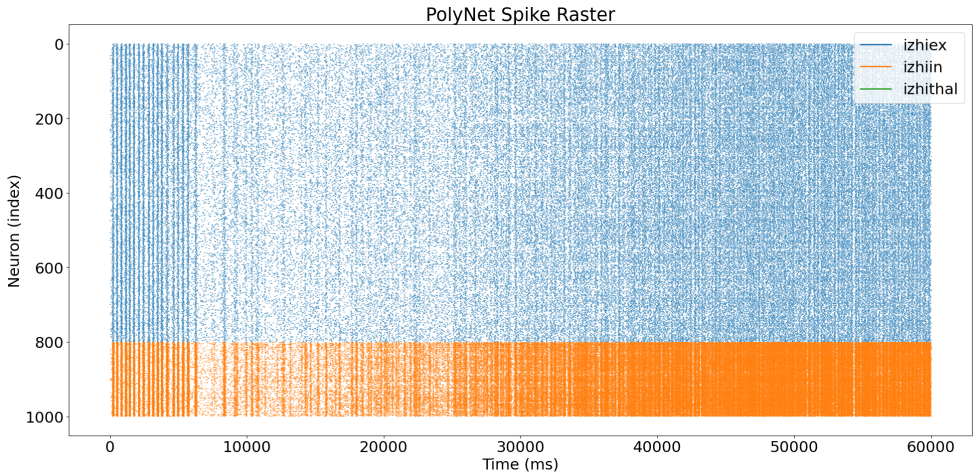
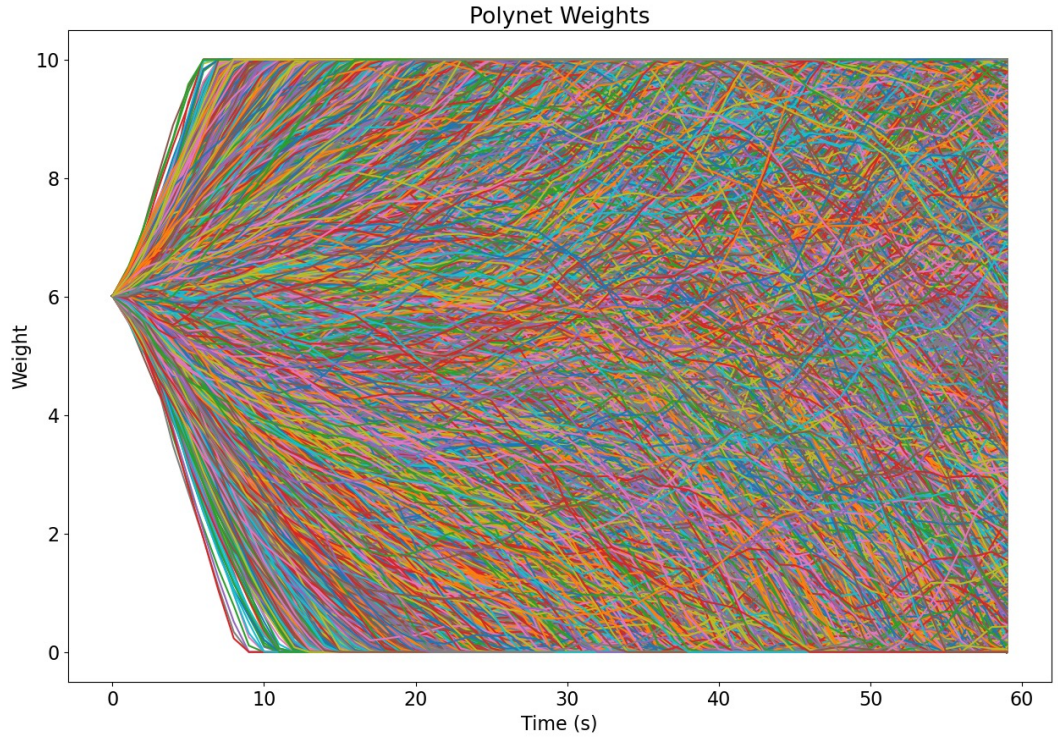
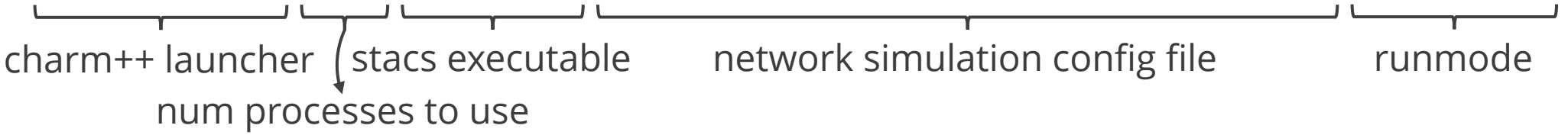
Installation (Charm++)

- Setting up Charm++ (Linux, MacOS, both)
  - Clone repo from github: https://github.com/UIUC-PPL/charm
  - Alternatively, get version 7.0.0 (available as tarball, zip)
    - `./build charm++ mpi-linux-x86_64 -j8`
    - If on Intel Mac, set the min version variable in the file `charm/src/arch/mpi-darwin-x86_64` to `12.6`
    - `./build charm++ mpi-darwin-arm8 -j8`
  - Set up associated environment variables (e.g. in `.bashrc`)
    - `export PATH=/path/to/charm/bin:$PATH`
- There are quite a few additional settings available for Charm++
  - On a different OS (e.g. Windows, IBM Blue Gene/Q, Cray XE/XC, PowerPC)
  - Using a different communication substrate than MPI (e.g. IB verbs, multicore)
  - Using a different compiler (e.g. `icc`, `xlc`)
  - More information in `charm/README.md`

Installation (STACS)

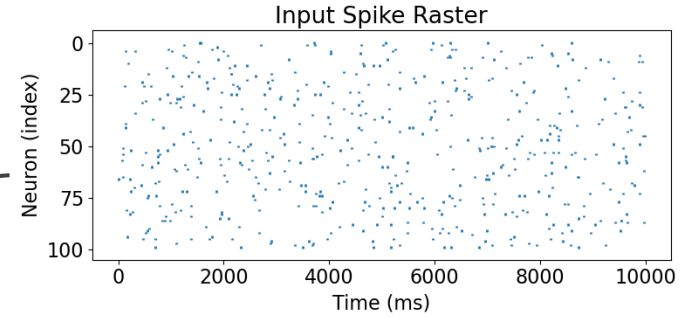
- After Charm++ is installed, STACS uses a trivial `Makefile` to compile
- Setting up STACS
  - Clone repo from github: https://github.com/sandialabs/STACS
    - `make -j8`
- Setting up Python
  - Use your typical python environment (e.g. Anaconda)
    - `pip install pyyaml`
- STACS is a command line executable, which is great for running large batch jobs on an HPC allocation, but the workflow(s) can be unfamiliar to those mainly used to single-node development and working within Python
  - STACS is also just not as user-friendly as it could be (work in progress)
  - There are some ways we can loosely interface STACS with Python
    - Developing YAML configuration files through the `pyyaml` package
    - Parsing and importing simulation output (e.g. to display a spike raster)

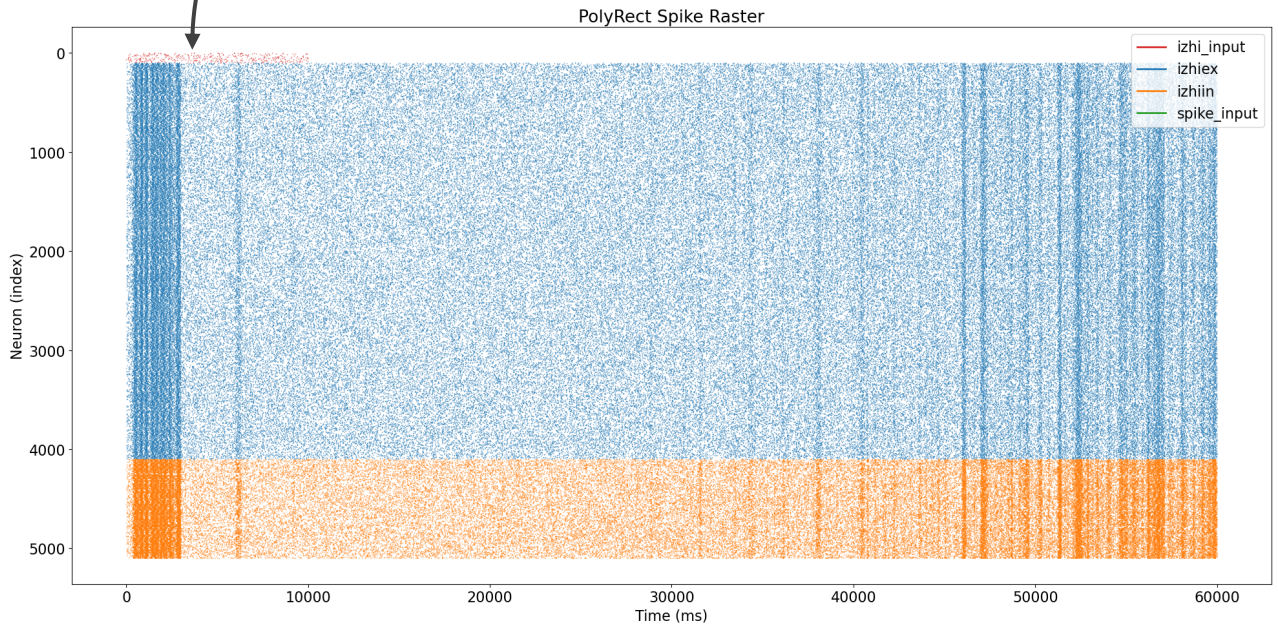
Quick Start Example

- From the base directory of the STACS repository you will find a “networks” folder
- For a network, there’s three YAML configuration files that need to be composed
  - Network simulation config (`.yml`): sets up information related to how the network will be processed by the simulator
  - Network graph model (`.graph`) : defines the structure of the vertices and edges that compose the network
  - Network substrate model (`.model`): defines the substrate dynamics used by the vertices and edges
- We’ll be looking at polynet for this quick introduction
  - This is an 80/20 excitatory/inhibitory network with STDP (from Izhikevich 2006)
    - ```
      cp polynet/examples/polynet.yml.eg polynet/polynet.yml
      cp polynet/examples/polynet.graph.eg polynet/polynet.graph
      cp polynet/examples/polynet.model.eg polynet/polynet.model
      ```
    - Also create a directory to store records:
      ```
      mkdir polynet/record
      ```
      ```

Quick Start Example

- To run this network, we point STACS to the simulation config
  - From the base directory of the STACS repository
    - `./charmrun +p8 ./stacs ./networks/polynet/polynet.yml`

charm++ launcher | num processes to use | stacs executable | network simulation config file

- This will first build the network and then simulate it
  - An initial network snapshot will be saved in the networks/polynet directory
  - A final snapshot will also be saved (with a different filebase “polynet.out”)
  - Event logs (of the spikes) will be saved in networks/polynet/record
- At this point, we can shift over to Python to perform some data analysis (e.g. plot the spike raster)



Defining a Network Model

- A network model is composed of three YAML-formatted configuration files:
  - The main config file sets up information related to how the network model will be processed by the simulator (this was `polynet/polynet.yml`)
    - Parallelization, total simulation time, random number seed, episodic trials
  - The network graph file defines the connectivity structure of the vertices and edges that compose the network (this was `polynet/polynet.graph`)
    - Neuron populations and synaptic projections
  - The network substrate model file parameterizes the time- and event-driven dynamics used by the vertices and edges (this was `polynet/polynet.model`)
    - State initialization (potentially from file) during network generation
    - Setting up event watchers and state probes
    - Parameterizing streams which open up ports for I/O

Defining a Network Model (Vertices and Edges)

- Separated into multiple YAML "files" using the using the `---` and `...` syntax
- Vertices and edges both mostly follow the same model format
- Model names (`modname`) are user-defined strings and are used to differentiate between different parameterizations of substrate dynamics (model names should be unique)
- Model types (`modtype`, the integer identifier) are defined in the model files (located in `stacs/models`) and provide a pointer to the substrate dynamics being used
- Parameters (`param`): information that is related to the model dynamics, but are otherwise shared across all instantiations of a given model type (e.g. the membrane leakage rate for a given neuron population)
  - These are set as simple name-value pairs
- States (`state`): information that are unique per model instantiation and/or are mutable throughout a network simulation (e.g. the membrane voltage for a given neuron)
  - These are set as initialization parameters for network generation (e.g. random number drawn from a uniform interval, bounded normal distribution, etc.)

Defining a Network Model (Streams, Records)

- In addition to vertices and edges, another substrate type is a stream
  - These were originally used to connect a network to external devices through YARP
  - We can also use these to provide inputs from file (e.g. a spike generator)
    - For a simple inputs, the SpikeInput model reads in a YAML-formatted event list
      - These are parameterized as the number of inputs and the path to the file
- For recording more information from a network model, a record “file” can be used
  - Different event types (applied current stimulus, “clamped” spikes)
  - State probes (defined by the model name, state, and recording frequency)
    - These work for both vertices and edges (indexed by vertex, local edge)



Defining a Network Model (Graph)

- There are two main sections in the network graph file, a list of vertices and a list of edges
  - There is also a section for streams for communication in/out of the network
  - The `modname` references the parameterized substrate models from earlier
- Vertices define groups/populations, and edges define how these are connected
  - The `order` of a vertex entry is how many there are in the group
  - Edge entries have a `source` group and a list of `target` groups
  - Vertex model names and source-target pairs have to be unique
- Vertices allow for spatial instantiation parameters (e.g. within a circle, rectangle)
- Edges are instantiated based on information from the source-target vertices
  - Spatial connectivity parameters (e.g. cutoff distance, periodic boundaries)
  - Indexical connectivity parameters (e.g. local group index, offsets)
  - Connectivity matrix can also be supplied from file (target-major order)
- This system allows for quite a bit of freedom in mixing/matching different substrate models with each other (however, this also includes incompatible models)

Defining a Network Model (Simulation)

- The simulation parameters are what you would typically expect out of SNN simulators
  - Main simulation information
    - Different run modes (network generation, repartitioning, and simulation)
      - These can be overridden by passing a runmode to STACS
    - Random number seeds, plasticity, load balancing
  - Network storage information
    - Location to save snapshots and records, filenames
    - Number of files and network partitions to use
  - Timing information
    - Maximum simulation time, timestep, length of event queue (max delay)
    - How often to save records, checkpoint network
    - Episodic simulation number of trials and duration
- Constructing configuration files is mostly fill-in-the-blank

Building and Running a Network Model

- To run a network, we point STACS to the simulation config
  - `./charmrun +p8 ./stacs ./networks/polynet/polynet.yml buildsim`



- The run command is composed of a few different components
  - Charm++ launcher (this should also just be on the `$PATH`)
  - Number of Charm++ processes to use (e.g. number of network partitions)
  - Optional path to an MPI hostfile (`[-f mpi_hostfile]`, if running on a cluster)
  - Path to the STACS executable (relative to current directory)
  - Path to the network simulation configuration file (relative to current directory)
  - Optional runmode (`buildsim` (`build`, `simulate`), `migrate` / `repart`)
    - Migrate/repart methods require `.part` files that provide new partition indexes (produced from a network partitioner, e.g. ParMETIS)

Building and Running a Network Model

- An example workflow for running an experiment could be:
  - Construct a network model
    - `network.yml`, `network.graph`, `network.model`
  - Build the network
    - `./charmrun +pN ./stacs /path/to/network.yml build`
  - Feed into a partitioner
    - `./charmrun +pN ./genet /path/to/network.yml part`
    - (GeNet is an earlier application that connects SNN-dCSR to ParMETIS)
  - Migrate the network vertices (overwrites original network snapshot)
    - `./charmrun +pN ./stacs /path/to/network.yml migrate`
  - Simulate the network for some time
    - `./charmrun +pN ./stacs /path/to/network.yml simulate`
  - Analyze the output (e.g. in Python, Matlab, etc.)
    - Network data in `.adjcy`, `.state`, `record/network.evtlog`, etc.

Building and Running a Network Model

- Some other ways to use STACS:
  - Build the network and simulate it
    - `./charmrun +pN ./stacs /path/to/network.yml buildsim`
  - Copy over and overwrite the original simulation output
    - ```
      for i in $( ls network.out.* )
        do j="$(echo ${i} | sed -e 's/.out//')"
        mv "${i}" "${j}"
      done
      ```
  - Modify the model parameterization (e.g. turn plasticity off, change out dynamics)
    - In `network.yml`, set `plastic = no`
    - In `network.model`, change `modtype = 20` to `modtype = 24`
      - When changing model types, the states must actually match between them
  - Simulate the network some more
    - `./charmrun +pN ./stacs /path/to/network.yml simulate`
      ```

Supplying Spike Input

- We can provide spikes into a network through an event list (in a YAML-formatted file)
  - This uses the `SpikeInput` stream model
  - Spikes are formatted as a list of input neurons with a list of timestamps per neuron
- For this example, we use the network in `networks/polyrect`
- We can plot the spike raster as usual





Developing Network Substrate Models

- All the STACS model files are found in `stacs/models`, and are defined as derived classes from the `NetModel` base class (using the `ModelTmpl` template, factory method)
  - Network models are represented as a directed graph, and neuron and synapse model dynamics are formulated as stochastic differential equations:
    - $dx = f(x)dt + \sum_{i=1}^{n} g_i(x)dN_i$: time-driven computation, event-driven communication
- When defining a network model, an integer identifier (the `modtype`) must be supplied
- Within the class constructure, there are multiple named lists for model parameters, model state (for real-valued and ”tick” representations), auxiliary states (for allowing an edge model to access and modify vertex state, e.g. synaptic current), and ports
- For the neurons (vertices), the dynamics are located in the `Step` function, which are executed at each timestep
- For the synapses (edges), the dynamics are located in the `Jump` function, which are executed at discrete event times
- Other functions that may be `Reset` (for episodic trials) and `Leap` (for periodic updates)

Developing Vertex Models (Step)

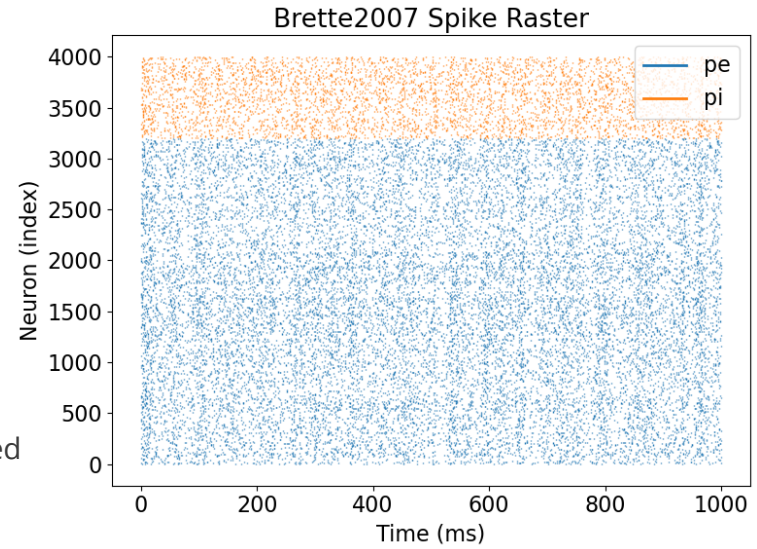
- For neuron models, we define how the states change over time
  - Model equations/dynamics are a function of model parameters and states
  - There is a lot of freedom available in how dynamics are computed
    - There are some random number generators available (uniform, normal)
    - Different numerical integration methods can be used (Euler, Runge-Kutta, etc.)
- Time is provided as the current simulation time `tdrift`
  and the amount of time left to simulate in the cycle `tdiff`
  - The “tick” representation in STACS is an integer type for precision
  - This can be converted to a real number (float type) through `TICKS_PER_MS`
- Any events (e.g. spikes) that are generated by the model are added to an event list
  - Events are defined as a `diffuse` timestamp, event `type`, and optional payload `data`
  - Helper indexical variables (`source` and `index`) are used for different target rules
    - This is any combination of local (self) and remote (target) vertices and edges

Developing Edge Models (Jump)

- For edge models, we define how the states change with discrete event
  - Model equations/dynamics are now also a function of event information, and auxiliary states (corresponding to named states on the target vertex model)
  - There is still a lot of freedom available in how dynamics are computed
- All edge models must have delay set as its first “tick” type state variable
  - This is used to place events into their correct event queues
  - The delay must also be long as the timestep resolution used
- Although STACS mostly follows a point-neuron model, the vertex and (incoming) edge states are collated, which enables potentially interleaved dynamics (e.g. dendrites)
  - The list of vertex states are on index `[0]`, and the `auxstate` provides an indexical pointer to the relevant state in this list
- Information about where the event came from is provided in the `event.source` variable and may be used to select between different state updates (e.g. either side of STDP)
- If time since last spike is required, it will need to be saved in an edge state in the model

Developing an Example Network Model

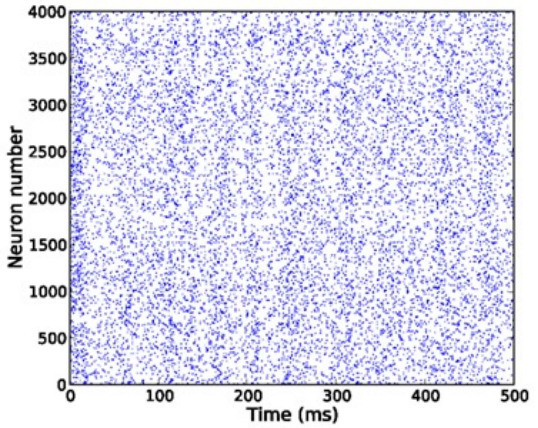
- For this example, we look at the simple benchmark found in Brette 2007: “Simulation of networks of spiking neurons: a review of tools and strategies”
- We replicate the network model found in Goodman 2008 (Figure 1): “Brian: a simulator for spiking neural networks in Python”
- This is an 80/20 reservoir network of 4000 leaky-integrate-and-fire (LIF) neurons with current-based (CUBA) exponential synapses



[Reproduced in STACS]

```
from brian import *
eqs = '''
dV/dt  = (ge+gi-(V+49*mV))/(20*ms) : volt
dge/dt = -ge/(5*ms)                : volt
dgi/dt = -gi/(10*ms)               : volt
'''
P = NeuronGroup(4000, model=eqs,
        threshold=-50*mV, reset=-60*mV)
Pe = P.subgroup(3200)
Pi = P.subgroup(800)
Ce = Connection(Pe, P, 'ge')
Ci = Connection(Pi, P, 'gi')
Ce.connect_random(Pe, P, p=0.02,
                  weight=1.62*mV)
Ci.connect_random(Pi, P, p=0.02,
                  weight=-9*mV)
M = SpikeMonitor(P)
P.V = -60*mV+10*mV*rand(len(P))
run(.5*second)
raster_plot(M)
show()
```

$$\tau_m \frac{dV}{dt} = -(V - E_L) + g_e + g_i$$
$$\tau_e \frac{dg_e}{dt} = -g_e$$
$$\tau_i \frac{dg_i}{dt} = -g_i$$



[Goodman 2008]

Limitations, Future Work

- As can be seen in several of the provided examples, STACS is not as user-friendly
  - Model types are stored as integers as opposed to human-readable strings
  - Although portable (e.g. in Python, Matlab), the analysis of simulation output data requires some file parsing and knowledge of the representation format
  - The auxiliary state is currently set in the model definition rather than being provided as a string through the model configuration file (like a port name)
  - Developing new network substrate models requires some proficiency in C++, and can involve a non-trivial degree of bookkeeping for state and parameter lists
    - There's a lot of freedom in defining the computation, but not as many shortcuts
    - That said, it's relatively straightforward to modify and add to model dynamics
    - It should be possible to provide a Python frontend to this (similar to using N2A)
- Tool interoperability (e.g. Fugu, ParMETIS) requires a fair amount of file parsing
  - But it's doable through the SNN-dCSR intermediate representation format
- Any synaptic dynamics (e.g. learning rules) are local (although this is by design)
  - Global dynamics will necessarily involve the multiple local interactions