Programmatic Flow of STACS

Felix Wang

January 13, 2017

Contents

1 Overview

This paper provides some basic technical documentation of the Simulation Tool for Asynchronous Cortical Streams (STACS). As a framework for embodiment and external feedback in simulated neural networks, STACS is composed of two main components. First is the neural network simulator developed using the Charm++ parallel programming framework (`http://charmplusplus.org/`). Second is the interface of this simulator to external devices using the YARP protocol (`http://www.yarp.it/`). Although these two components interact, neural network simulation may proceed standalone when no external feedback is needed.

2 Simulator

STACS proceeds in three main phases: initialization from disk, simulation of the neural network, and finalization to disk. On disk, snapshots of the neural network enable persistence between simulation runs. In addition to saving the network state at the end of a simulation run, periodic checkpointing of the network also occurs during simulation.

2.1 Charm++

An understanding of how programs written in Charm++ is helpful in understanding the programmatic flow of STACS. Charm++ exists as an extension to the C++ programming language, additionally providing an adaptive runtime system for parallel execution. The programming model is composed of parallel objects, called *chares*, where computation proceeds according to a message driven paradigm. That is, communication and the flow of computation in Charm++ occurs by way of asynchronous messages evoking methods on parallel objects. Here, data dependencies of the evoked methods are satisfied by the communicated messages. Subsequently, any data generated by these methods are communicated to dependent methods. Although this message driven paradigm may be achieved in any number of message passing libraries, Charm++ provides an expressive high level framework for developing parallel applications in this manner.

2.2 Persistence

A snapshot of the neural network is composed of details such as network's graph structure, the state of the neural models, and any messages in transit. This information is stored as files on disk and serializes everything that's needed for the neural network to persist between simulation runs. An initial snapshot of the neural network is generated by a companion program called *genet*, whereas STACS is only responsible for reading in a snapshot of the network prior to simulation and writing out a similar snapshot post simulation.

The files that compose the snapshot can be differentiated as "configuration" and "data", and are identified by an extension following a base file name for the network. Because data files contain

information that are generally split across multiple computing units, they are further identified by a number corresponding to the computing unit that data resides on.

- `filebase.model` - contains a list of the network models and parameters specific to the particular neural network being simulated
- `filebase.dist` - contains a rolling sum of the amount of data entries per partition of the network (these are the number of vertices, edges, states, and events)
- `filebase.coord.#` - three dimensional coordinate locations of the vertices in the neural network
- `filebase.adjcy.#` - connection information of the vertices following a condensed sparse row format
- `filebase.state.#` - state information that is relevant to the vertices and their corresponding incoming edges
- `filebase.event.#` - event information that is relevant to the vertices and their corresponding incoming edges

2.3 Execution

STACS is executed by the Charm++ runtime system.

An example execution call is: `./charmrun +p2 ./stacs config.yml`.

- `./charmrun` - launches the Charm++ runtime system on the parallel computing platform
- `+p2` - corresponds to launching the Charm++ runtime system with 2 computing units
- `./stacs` - is the program (STACS) to be executed by the Charm++ runtime system
- `config.yml` - is the main configuration file read by STACS

Some key items in the main configuration file are:

- `npdat` - the number of data files there are (this should match with the `+p` argument for the number of computing units provided to Charm++)
- `npnet` - the number of parallel partitions of the network there are (this should be greater than or equal to `npdat`)
- `filebase` - the base file name of the neural network snapshot files

2.4 Initialization

During the initialization phase, STACS processes the configuration files for the neural network and sets up the appropriate parallel objects (chares) across the computing units. It then reads the snapshot files from disk and starts the simulation after the neural network data is loaded into the appropriate chares. The relevant function calls and messages that are involved in this phase are as follows.

2.4.1 Main Entry Point

Programs written in Charm++ have a main entry point defined as the constructor of the "main chare". For STACS, this is simply named `Main`, and the entry point of the program is at `Main::Main(CkArgMsg *msg)`, where the `CkArgMsg` message provides the standard command line arguments to STACS as `msg->argc` and `msg->argv`. STACS expects that there is only one command line argument, which is the location (relative path) of the main configuration file.

Reading in the configuration files proceeds as follows:

- `Main::ParseConfig()` - parses the main configuration file, `config.yml`, and sets global read-only variables across the parallel computing platform
- `Main::ReadDist()` - reads in the network configuration file, `filebase.dist`
- `Main::ReadModel()` - reads in the network configuration file, `filebase.model`

The network information is then packaged into messages as:

- `Main::BuildDist()` - builds an `mDist` message, which provides information during construction of the `NetData` chare array
- `Main::BuildModel()` - builds an `mModel` message, which provides information during construction of the `Network` chare array

These messages are then passed to the constructors of the `NetData` and `Network` chare arrays, respectively.

2.4.2 Constructing Parallel Objects

When an entry method is evoked on the chare array, all chares in the chare array excecute the evoked method. There are two main groups of parallel objects utilized by STACS, realized as chare arrays in Charm++. Briefly, the `NetData` chare array is responsible for the network data de/serialization from/to disk, and the `Network` chare array is responsible for the neural network simulation.

During construction of these chare arrays, the main setup procedures involve allocating the relevant data structures according to the configuration of the neural network. Bookkeeping for which chare of the `NetData` array corresponds to which chare of the `Network` array is also calculated.

2.4.2.1 NetData Chare Array

`NetData::NetData(mDist *msg)` is the constructor for the `NetData` chare array, where the `mDist` message provides information needed to read in the network snapshot.

After the the relevant data structures have been allocated, the network snapshot is read in from disk. Here, `NetData::ReadCSR` reads in the network data files `filebase.coord`, `filebase.adjcy`, `filebase.state`, and `filebase.event`.

2.4.2.2 Network Chare Array

`Network::Network(mModel *msg)` is the constructor for the `Network` chare array, where the `mModel` message provides information regarding the network models and their parameters that will be used during simulation.

2.4.3 Loading Network Data

After the `NetData` and `Network` chare arrays have been initialized, they return control to `Main::InitSim()`, which acts as a coordination point before the next stage of the initialization phase occurs. Here, the data structures have been allocated, and data from `NetData` is loaded into `Network`.

The process of moving network data from `NetData` to `Network` proceeds as:

- `Main::InitSim()` - sends a reference to the `NetData` chare array to the `Network` chare array so that it may request network data
- `Network::LoadNetwork(CProxy_NetData cpdat)` - requests its specific network part from `NetData`
- `NetData::LoadNetwork(int prtidx, const CkCallback &cb)` - processes the request from `Network` and sends back the requested network part as a `mPart` message
- `Network::LoadNetwork(mPart *msg)` - receives the network part from `NetData` and populates its data structures accordingly

Once the network data has been loaded into the `Network` chare array, `Network::CreateGroup()` sets up the event communication structure of the network simulation according to the adjacency information. This communication structure provides multicast sections of the `Network` chare array.

2.5 Simulation

After the network data has been loaded from `NetData` to `Network`, control is returned back to `Main::StartSim()`, which acts as a coordination point before simulation starts. STACS enters into the simulation phase here.

2.5.1 Main Simulation Loop

`Network::Cycle()` is the main simulation loop that determines the forward progression of the neural network simulation as well as the periodic export of network data There are two main counters that are incremented as the simulation progresses, the iteration count, `iter`, and the simulation time elapsed, `tsim`.

The control flow per cycle checks for:

- `tsim >= tmax` - if the maximum simulation time has been reached
- `iter == checkiter` - if the simulation should checkpoint a network snapshot to disk

- `iter == reciter` - if the simulation should save record data to disk

If the above conditions pass through, the simulation enters into computation of the next iteration or time step of the network, generating event information as well as any simulation records.

This network data is then processed as follows:

- `Network::BuildEvent()` - builds an `mEvent` message, which contains event information to be communicated to other, dependent network parts
- `Network::StoreRecord()` - maintains a local copy of records until they may be written out to disk

At the end of an iteration the `mEvent` messages generated by a network part are communicated to its multicast section of the `Network` chare array.

2.5.2 Event Communication

`Network::CommEvent(mEvent *msg)` is the main entry method that processes event communication, passed as `mEvent` messages, between network parts.

Once all dependent messages for the next iteration have been received, the network part enters into its next cycle.

2.5.3 Checking Network Data

STACS periodically saves snapshots of the network data during simulation such that the simulation may be recovered and restarted in case of interruption.

The checkpointing process proceeds as follows:

- `Network::CheckNetwork()` - moves control out of the main simulation loop to perform checkpointing
  - `Network::BuildPart()` - builds an `mPart` message, which contains all the necessary information for serialization of a network partition
  - The network partition information is sent to `NetData`, and control is returned back to the main simulation loop
- `NetData::CheckNetwork(mPart *msg)` - receives network part data from `Network`
  - `NetData::WriteCSR()` - serializes the network data onto disk
- Information about the network distribution is reduced to `Main`
- `Main::CheckNetwork(CkReductionMsg *msg)`, where the `msg` is of type `netDist`, which contains information about the distribution of the network across the network partitions
  - `Main::WriteDist()` - writes network distribution information to disk

2.5.4 Checking Network Records

STACS periodically writes records generated from the simulation onto disk so that may be used for offline processing

The recording process proceeds as follows:

- `Network::CheckRecord()` - moves control out of the main simulation loop to perform checkpointing
  - `Network::BuildRecord()` - builds an `mRecord` message, which contains records information packages record information
  - The records information is sent to the `NetData` chare array, and control is returned back to the main simulation loop

  `NetData::CheckRecord(mRecord *msg)` - receives and processes record data from the `Network` chare array
  - `NetData::WriteRecord()` - serializes the record data onto disk

2.6 Finalization

When the simulation is complete, the `Network` chare array returns control to `Main::StopSim()`, which acts as a coordination point for the network to have finished any computation and communication processes.

2.6.1 Saving Network Data

The final state of neural network is saved to disk. This is very similar to checkpointing the network.

The saving process proceeds as follows:

- `Network::SaveNetwork()` - is called to perform the final checkpoint
  - `Network::BuildPart()` - builds an `mPart` message, which contains all the necessary information for serialization of a network partition
  - The network partition information is sent to `NetData`
- `NetData::SaveNetwork(mPart *msg)` - receives network part data from `Network`
  - `NetData::WriteCSR()` - serializes the network data onto disk
- Information about the network distribution is reduced to `Main`
- `Main::SaveNetwork(CkReductionMsg *msg)`, where the `msg` is of type `netDist`, which contains information about the distribution of the network across the network partitions
  - `Main::WriteDist()` - writes network distribution information to disk

2.6.2 Saving Network Records

Any records that haven't been saved yet are written out to disk as well.

The recording process proceeds as follows:

- `Network::SaveRecord()` - is called to perform the final saving of records
  - `Network::BuildRecord()` - builds an `mRecord` message, which contains records information packages record information
  - The records information is sent to the `NetData` chare array

  `NetData::SaveRecord(mRecord *msg)` - receives and processes record data from the `Network` chare array
  - `NetData::WriteRecord()` - serializes the record data onto disk

2.6.3 Halting

After network data and records have been successfully written, control is returned to `Main::FiniSim()` which exits STACS.

3 Interface

The interface between the simulation and external devices is handled through YARP. When STACS is compiled with YARP some additional elements in the programmatic flow become relevant. By default, there is a method that enables external, interactive control of the simulation by the user. If applicable, network models that utilize YARP may also be used during simulation to provide external input and output capabilities to a variety of external devices to and from the simulation, respectively.

For STACS to operate using YARP, a YARP server on the network needs to be running. This may be invoked by calling `yarp server` from the command line.

3.1 STACS with YARP

3.1.1 Initialization

During initialization, additions to the programmatic flow are as follows:

- In `Main::Main(CkArgMsg *msg)`, during configuration:
  - `Main::BuildVtxDist()` builds an `mVtxDist` message, which provides information about the distribution of vertices across the network partitions
  - An additional chare constructor is called, `StreamRPC::StreamRPC(mVtxDist *msg)`, which enables communication through YARP
- YARP is initialized on each computing unit during the `NetData` constructor call
- Once YARP is initialized, `Main::InitSim()` calls `StreamRPC::Open(CProxy_Network cpnet)` which opens up the RPC port for communication with a reference to the `Network` chare array
- In `Network::LoadNetwork(mPart *msg)`, when the network models are being loaded, any models with YARP ports have them opened for communication

3.1.2 Simulation

During the simulation, additions to the programmatic flow are as follows:

- RPC messages may be sent to the simulation on the `/stacs/rpc` port
  - `RPCReader::read(yarp::os::ConnectionReader &connection)` receives any RPC messages through YARP
  - `RPCReader::BuildRPCMsg(idx_t command, yarp::os::Bottle message)` builds an `mRPC` message, which then can be sent through Charm++
  - `Network::RPCMsg(mRPC *msg)` receives `mRPC` messages and processes them accordingly on the neural network
- During the main simulation loop, `iter == synciter` is an additional condition that is checked during the control flow
  - If the simulation is not paused or is told to unpause, `StreamRPC::RPCSync()` acts as a coordination point after an `mRPC` message is processed, and control is returned to the main simulation loop
  - If the simulation is paused or is told to pause, `StreamRPC::RPCPause()` acts as a coordination point after an `mRPC` message is processed

3.1.3 Finalization

During the finalization, additions to the programmatic flow are as follows:

- When the simulation is stopped, `Main::StopSim()` calls `StreamRPC::Close()` which closes the RPC port
- As the neural network is being saved, `Network::SaveNetwork()` closes any YARP ports that have been opened by the network models
- YARP is finalized on each computing unit during the `NetData::SaveNetwork(mPart *msg)` after network data has been serialized

3.2 External Control

With YARP, there is an additional chare in STACS, `StreamRPC`, which is responsible for external, interactive control of the simulation. Specifically, this sets up an `RPCReader` callback for STACS which accepts messages on the `/stacs/rpc` port.

Even if there are no streams available through network models, STACS provides a way for the simulation to be controlled interactively through the use of remote procedure calls (RPC) through YARP. This method may be used to send some basic commands for pausing the simulation, stepping through simulation time, and applying manual stimulation commands. The RPC interface may be invoked by calling `yarp rpc /stacs/rpc` from the command line.

RPC Commands to control simulation are:

- `pause` - pause or unpause the simulation

- `stop` - stop the simulation (final state of the network is written out)
- `check` - checkpoint the simulation (when paused)
- `step <t>` - step the simulation `<t>` *ms* (when paused)
- `stim <t o a d>` - apply arbitrary stimulation pulses to the network with specified `<t>` targets, `<o>` offsets, `<a>` amplitudes, and `<d>` durations

For the stimulation pulses, the targets may be specified as:

- Individual neurons - both the number of neurons and their indices need to be provided
- Spacial area - both the coordinate of the center point and a radius need to be provided

Each pulse in a stimulation command is parameterized as:

- Offset - time in *ms* from the time when a stimulation command is given that the pulse starts
- Amplitude - amplitude (unitless) of the pulse that will be received by the targeted network models
- Duration - time in *ms* from the time when the pulse starts that the pulse is active

3.3 Streams

YARP streams refer to network models that open ports to either receive or send data from the network simulation online through YARP. In general, streams that accept more standard YARP signals, such as for audio, will remain better suited to varying devices due to their increased modularity.

A YARP port is opened by the network models that get initialized by the `Network` chare array, which then sets up a callback function, `onRead`, for when data is available on that port. These ports are decoupled, according to the YARP philosophy, such that the input stream may stop and start at any time without the simulation stalling. When streaming data from real-world devices, care should be taken so that the complexity of the neural network matches the computing resources in order to run in real-time. For prerecorded data, all that is needed is for the rate of the streams to be comparable to that of the simulation.