CoreNeuron Overview

Ben Cumming
CSCS – Swiss National Supercomputing Center

February 16, 2017

Contents

Abstract

This document presents an overview of the CoreNeuron code that was released as part of the PCP process for the Human Brain Project in July 2014. The analysis focuses on the structure of the code that has the highest computational overheads.

More detailed analysis of the algorithms themselves, and important parts of the code like spike communication, that do not contribute to the time to solution are not considered for analysis. For now I will focus on just the computationally intensive parts of the code.

Note that this analysis is based on an early version of the CoreNeuron code, and that different algorithms (e.g. modelling plasticity) may significantly increase the importance of parts of the application not yet considered here.

1 What is CoreNeuron?

The version of CoreNeuron that was released for the PCP is derived directly from **HBP-Neuron**, a flavour of **Neuron** maintained by the Blue Brain Project (BBP) group at EPFL. CoreNeuron is derived directly in the sense that it is a subset of the features and corresponding code from HBPNeuron. The code has been modified as much as needed to remove it from the larger HBPNeuron infrastructure, and reduce the memory footprint of the code.

Note that HBPNeuron refers to the Neuron plus the `.mod` files used to define the models used in the BBP group. The computational back end is identical for both HBPNeuron and Neuron[1].

2 The Code

The code in its current form is a mixture of C and C++.

It must be noted that the code base is currently very challenging to understand and benchmark. It is derived directly from the Bluron/Neuron code base, which has grown organically over more than 20 years. There are very many opportunities to simplify and improve the code using modern programming languages and development techniques.

To use both CPUs and accelerators (e.g. GPU and MIC) effectively, parts of the code will have to be refactored or rewritten. The main challenge in refactoring the code is not the complexity of the algorithms, it is the difficulty of understanding the current code. Developing documentation of the algorithms as they are currently implemented is an essential first step.

From early work with the code base, the majority of wall time for the example circuits release for the PCP is spent in a relatively small set of code: the `nrnoc` solver implementation, and the mechanisms defined in `/mech/cfiles`. Improving performance, portability and maintainability will benefit from working on the core computational API.

An aim of this report is as a first attempt at describing the algorithms clearly, to make it possible to reason about how them without getting distracted by implementation details. To assist in this, many of the algorithms are presented in a pseudo-language similar to **Julia**[2], which should be familiar to users familiar with **Matlab**.

2.1 Understanding The Code

Neuron has grown organically, with new features "bolted on" over a long period of time (e.g. support for Python scripting, threading, etc.). Some of these features have been removed in CoreNeuron, leaving behind some design patterns that at first confuse newcomers. One

[1]HBPNeuron has some additional I/O routines `bbsave_state` and `bbcore_write` that will be integrated back into Neuron at some point.

[2]See the website: `julialang.org`

- **nrniv**
  - C and C++ (11,470 lines).
  - The CoreNeuron driver: `main` function is in `main1.cpp`.
- **nrnoc**
  - C (2,889 lines).
  - The CoreNeuron "engine"
    - storage
    - solvers
    - time stepping
- **mech/cfiles**
  - C (11,301 lines).
  - Definitions of all the mechanisms.
  - generated from `.mod` files by Neuron.
  - the generated code is very messy (use `clang-format` to make things bearable)
- **nrnmpi**
  - C (1,096 lines).
  - Wrappers around MPI routines.
  - Spike exchange implementation.
  - Global variables that store MPI state.
- **utils**
  - C++ (4,494 lines)
  - Random number generators

Figure 1: Overview of the directory structure for the source code. There are a total of 31,250 lines of code (including white space and comments).

consequence is that there are some data structures that are overly-complicated for the role they currently serve, but they were part of a more complicated design before the CoreNeuron rewrite.

There are also many instances where the naming of variables, global variables, and gratuitous use of the preprocessor don't help. It is recommended that the code be reformatted using a tool like `clang−format` (especially the automatically generated C files in `mech/cfiles`). Applying the preprocessor to individual files shows the origin of symbols that have been #defined with the preprocessor (e.g. using the "`−E −P`"[3] flag in `gcc`)

2.2 Code Layout

The source code is packaged in a file `CoreBluron.tar.gz`, which has the directory structure in Fig. 1. In terms of time to solution, functions defined in `mech/cfiles` dominate. These are called from the solver routines in `nrnoc`, which implements the core computation, and is the focus of the analysis here.

[3] `-E` applies the preprocessor, and `-P` makes the output more human readable.

2.3 Building

The code was built on Cray XC-30 system Piz Daint at CSCS with minimal fuss using the GNU toolchain. The Cray compiler toolchain had problems that are not insurmountable, but they would require a lot of tinkering with the **Buildyard**[4] build tool used by CoreNeuron.

The Buildyard uses cmake, with a custom set of cmake modules developed by BBP (the BuildYard modules). Many of these modules are not required by CoreNeuron, and configuration can be sped up significantly by removing them (for example the C++11 tests). The cmake configuration attempts to determine the version of the Cray compiler by passing a flag that the compiler doesn't recognise, causing the configuration to exit.

2.4 Datasets

There are two data sets provided with the PCP benchmark code:

- **TEST1_CACHE** A network small enough to fit into Cache of one rack of BG/Q. Has size of 2.5G on disk.
- **TEST2_DRAM** A much larger network (size 4.5T on disk), that fits in DRAM of one rack of BG/Q.

[4]github.com/Eyescale/Buildyard

3 The Algorithm

It will be shown later that the implementation of the spatial and temporal integration of the **cable equation** that describes the time evolution of voltage in a cell accounts for over 99% of all wall time. The discussion of **the algorithm** here offers a basic mathematical description of the algorithm with many details that do not affect the implementation ommitted.

3.1 Discretization of The 1D Cable Equation

The partial differential equation (PDE) that describes the time evolution of the voltage at each node a discretized cell has the following general form:

$$C\frac{\partial V}{\partial t} + I = f\frac{\partial}{\partial x}\left(g\frac{\partial V}{\partial x}\right) \tag{1}$$

where f and g are functions of the spatial dimension x (they are functions of **cable radius** in this model). The value of current I and capacitance C are both dependent on the voltage V.

See Appendix A for a detailed derivation of the spatial and temporal discretization. The discretization gives rise to a tridiagonal system of linear equations that is solved at every time step. The equation at each point has the following form,

$$a_i V_{i+1}^{n+1} + d_i V_i^{n+1} + b_i V_{i-1}^{n+1} = r_i \tag{2}$$

where the diagonals in the matrix are defined

$$\begin{aligned}
\text{upper diagonal}: \; a_i &= -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta x^2}, \\
\text{lower diagonal}: \; b_i &= -\frac{f_i g_{i-\frac{1}{2}}}{2\Delta x^2}, \\
\text{diagonal}: \; d_i &= \frac{C_i}{\Delta t} - (a_i + b_i), \\
\text{right hand side}: \; r_i &= \frac{C_i}{\Delta t}V_i^n - I_i - a_i\left(V_{i-1}^n - V_i^n\right) - b_i\left(V_{i+1}^n - V_i^n\right).
\end{aligned}$$

The off-diagonal coefficients in the linear system, i.e. a_i and b_i, are constant in time because they depend only on the radius of the cable. In practice the off-diagonal entries in $\mathbf{a}$ and $\mathbf{b}$ are computed once at start up. The values on the diagonal and right hand side vector, i.e. those in $\mathbf{d}$ and $\mathbf{r}$ are updated at each time step, then modified when solving the linear system using Thomas' algorithm. The values on the diagonal and right hand side vector are calculated using the precomputed values for a_i and b_i.

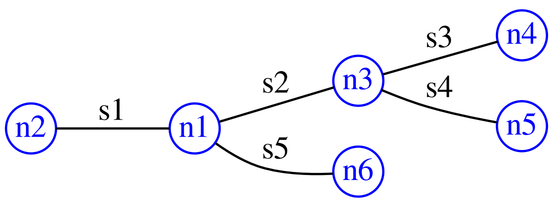
3.2 Trees and Branching

Here the one-dimensional discretization in the previous section will be extend to include branching, whereby the spatial domain is composed of a series of one-dimensional **sections**, that are joined at branch points to form a tree.

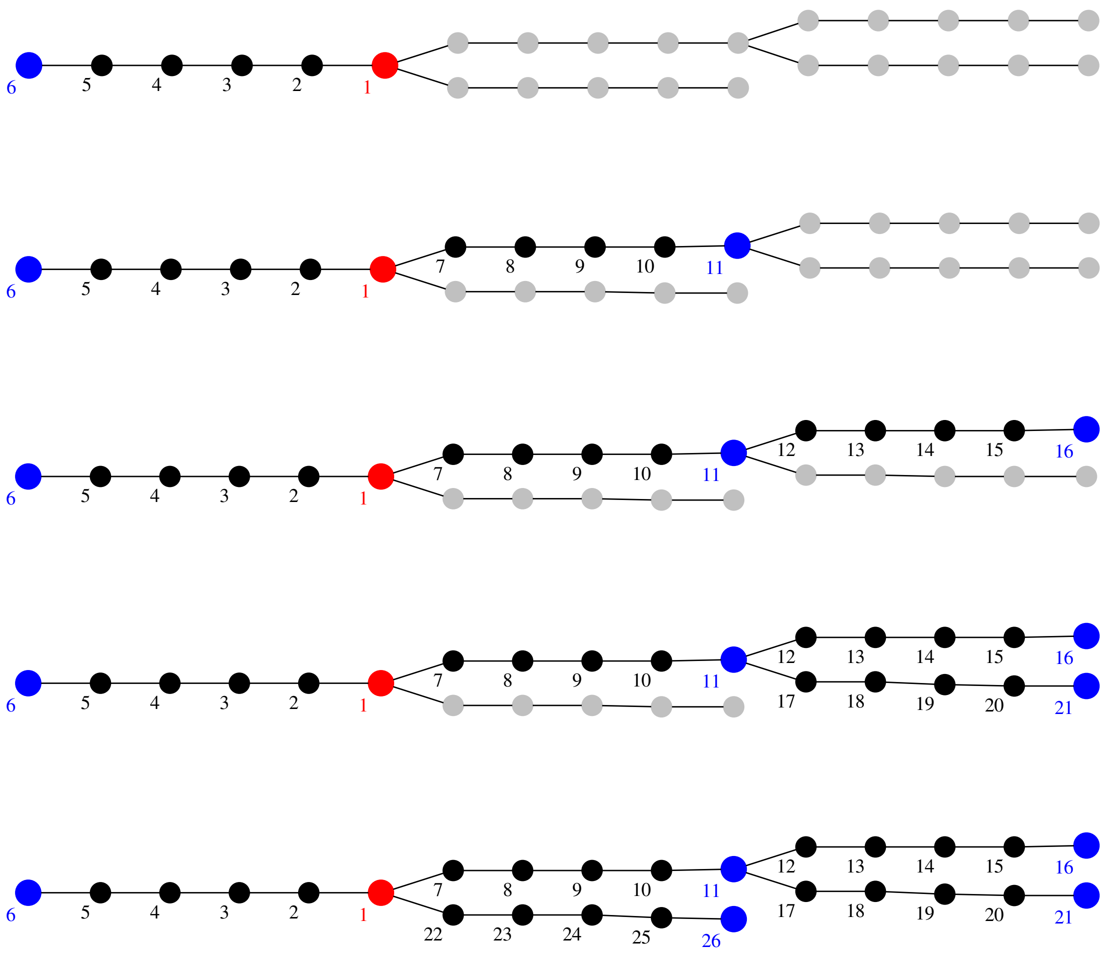
A small tree structure that will be used throughout this section is illustrated in Fig. 2(a). It is important to note that the graph formed by the branching sections is a true tree, i.e. it has no circuits (once a section has branched, the branches can not "rejoin").

Some terms used in discussing the tree structure in Fig. 2(a)&(b) are:

- **section** a branch in the tree structure, which corresponds to the one-dimensional line segment between branch points. These are numbered $s*$ in Fig. 2(a).
- **branch** same definition as section.
- **node** a point in the spatial discretization. These are numbered in Fig. 2(b).



(a)



(b)

Figure 2: Numbering of nodes and edges for Hines algorithm. (a) The high level branch and connection numbering scheme, with the branch nodes and sections numbered; (b) the numbering of individual nodes in the fully discretized domain with 5 segments per section.

- **branch node** a node where two branches join. These are the blue points denoted $n*$ Fig. 2(a).

The first step of the spatial discretization is to discretize each one-dimensional section. Then the nodes are numbered using a scheme that gives the matrix a sparsity structure that allows the linear system to be solved in linear time, equivalent to Thomas algorithm.

A numbering scheme for the nodes that facilitates efficient solution of the linear system is not unique. A general description of a valid numbering strategy is as follows:

1. One node is chosen as the **root node** of the tree, either a terminal node or a branch node, and assigned index 1.
2. The tree is then traversed along each branch with the root node as the starting point. The nodes are numbered sequentially in ascending order.
3. Every node (except the root node) has one-and-only-one **parent node**, which is its neighbour that is closer to the root node (and thus has a lower index).

The key property that is maintained by the numbering scheme is that the index of a node's parent node is lower than the node's index (equivalent to **minimum degree ordering**). An example of a numbering strategy being recursively applied to our example tree is shown in Fig. 2(b).

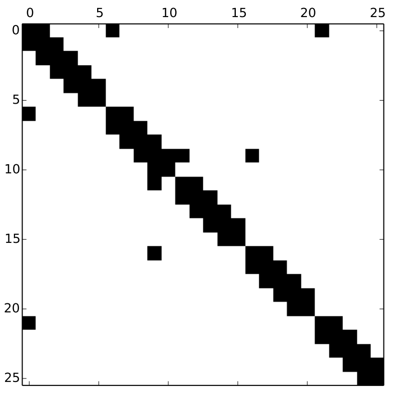


Figure 3: Sparsity pattern of the matrix corresponding to the tree numbering in Fig. 2.

To describe the sparsity pattern of the matrix from this numbering only the parent indexes, $p_i \quad i \in [2:n]$, for each node need to be stored. The matrix sparsity pattern, which is illustrated for our example tree numbering in Fig. 3, has the following properties:

- The sparsity pattern is symmetric.
- The diagonal values are all nonzero.
- There is exactly one off diagonal value in each row of the lower triangle at a_{ik} and exactly one off diagonal value in each column of the upper triangle at a_{ki}, where $k = p_i$.
- The matrix can be stored with three vectors, **a**, **b** and **c**:

$$a_i = A_{ki} \tag{3}$$
$$b_i = A_{ik} \tag{4}$$
$$d_i = A_{ii} \tag{5}$$

where $k = p_i$.

Note 1 *The choice of the root node and the traversal order when generating the numbering is important if trying to solve the linear system in parallel. It is possible to solve sub-trees that branch from a branch node independently, before combining the results to solve the value* *at the branch node. For sequential solution on the CPU this isn't important, however a GPU implementation might try to take advantage of this.*

3.3 Hines Algorithm

These linear systems can be solved very efficiently, in linear $O(N)$ time, using an algorithm that is equivalent to the Thomas algorithm for solving tri-diagonal systems. This algorithm, called Hines algorithm, proceeds by eliminating the nonzero entries in the upper triangle of A. Recall the matrix property that there is only one non-zero value in each column of the upper triangle at A_{ki}, which is stored in a_i.

$$\begin{pmatrix} A_{kk} & \dots & A_{ki} \\ \vdots & \ddots & \mathbf{0} \\ A_{ik} & \mathbf{0} & A_{ii} \end{pmatrix} \text{which is stored as:} \begin{pmatrix} d_k & \dots & a_i \\ \vdots & \ddots & \mathbf{0} \\ b_i & \mathbf{0} & d_i \end{pmatrix}$$

The nonzero value in column i, i.e. a_i, can be zeroed out with a row operation

$$\text{row}_k \leftarrow \text{row}_k - a_i/d_i \cdot \text{row}_i.$$

In practice the value of a_i is not changed, and only the values in on the diagonal and in the RHS vector have to be modified

$$d_k \leftarrow d_k - a_i/d_i \cdot b_i,$$
$$r_k \leftarrow r_k - a_i/d_i \cdot r_i,$$

So that our submatrix now looks like

$$\begin{pmatrix} d_k - a_i/d_i \cdot b_i & \dots & \mathbf{0} \\ \vdots & \ddots & \mathbf{0} \\ b_i & \mathbf{0} & d_i \end{pmatrix} \begin{pmatrix} r_k - a_i/d_i \cdot r_i \\ \vdots \\ r_i \end{pmatrix}$$

The algorithm proceeds by eliminating the values in the upper triangle with a backward sweep over columns $n : -1 : 2$. A forward sweep is then used to eliminate the nonzeros in the lower triangle, determine the solution.

In practice, this algorithm is very efficient, contributing less than 1% to the time to solution in the benchmarks released with the PCP.

4 The Time Step

In this section an will look at how the main time-stepping algorithm, which accounts for nearly all the time to solution, is implemented. However, before looking at the time step implementation, a short description of all of the main stages of the simulation is presented to put the time stepping code in context.

4.1 Overview of Main

The driver code is in `nrniv/main1.cpp`. From the breakdown of the wall time for the TEST2 data set in Tbl. 4.1 it is apparent that from a computational point of view, the only component of importance is the time stepping/sover portion of the program in `BBS_netpar_solve`, which takes 99.8% of time to solution.

Nevertheless, I will breifly describe the other steps performed in the main driver:

1. `mk_mech`
   The mechanisms are configured. A text file with a tuple for each mechanism (name, unique index, parameter count, type, etc ...) is scanned. This text file is generated when the cell group files are generated by HBPNeuron. This information provides a bridge between the runtime and the mechanisms implemented using the Neuron `.mod` language.
2. `mk_netcvode`
   Creates a new `NetCvode` object (see `nrnoc/netcvod.h/cpp`). This sets up the priority queue use to send and deliver spiking events.
3. `nrn_setup`
   Before calling `nrn_setup`, the configuration file `files.dat` is read to see how many and which cells are to be loaded for simulation. The cells are assigned in a round-robin fashion between the MPI ranks. Then `nrn_setup` is called with a list of cell ids, to load the cell data from disk.
4. `BBS_netpar_mindelay` and `mk_spikevec_buffer`
   The mindelay and spike buffer size are configured. All neuron cells can be integrated-in-time independently for the **minimum network connection delay**, i.e. spikes do not have to be delivered in the interval in which they were generated. These interval boundaries are used as synchronization points.
5. `BBS_netpar_solve`
   The time stepping code. The focus of this report.
6. `output_spikes`
   Write spike information to disk.

section	wall time (s)	contribution %
`mk_mech`	0.01	0.0
`mk_netcvode`	0.00	0.0
`nrn_setup`	0.69	0.2
mindelay/spike buffer	0.15	0.0
`BBS_netpar_solve`	388.93	99.8
`output_spikes`	0.01	0.0

Table 1: Breakdown of wall time for TEST2 data set running on one node of Piz Daint, with 1 cell per core.

4.2 Drilling Down to The Time Step

The time stepping and all computation associated with it are performed in the `BBS_netpar_solve` routine. A backtrace of the call tree from `BBS_netpar_solve` to `nrn_fixed_step_thread`, where 100% of the computation is performed, is shown in Fig. 4. The exact role played by each of these routines is not important now: some of them , and others are wrappers for passing cell groups to a pthread to separate integration (see Section 5 for more information about cell groups and threads). It is the integration of individual cell groups inside the lowest level function, `nrn_fixed_step_thread` that interests us.

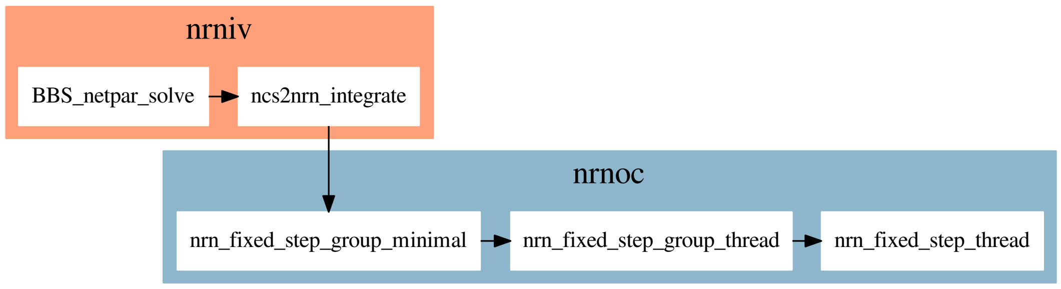


Figure 4: backtrace to the main computational routine.

Each MPI rank has a set of cells assigned to it in a round robin fashion during the initialization phase (in the call to `nrn_setup` in `main`). The cells are packaged together into **cell groups**, with one cell group per intput file. The selection of cells in a cell group is chosen when the input files are generated to improve load balancing (static load balancing). For more information, see Section 5.

4.3 Pseudo-Code

The algorithm in the time stepping are quite simple, however their implementation can be very difficult to understand and reason about. To make it easier to understand to understand implementation, the core routines are presented here in a high-level pseudo code similar to Julia. Many variables, type names, and functions have been renamed to make their meaning clearer. For example `_nt` has been renamed `cell_group`, to clearly identify it's meaning.

An example of the pseudo code represents the following C code

```
for (tml = _nt->tml; tml; tml = tml->next)
  if (memb_func[tml->index].current) {
    mod_f_t s = memb_func[tml->index].current;
    (*s)(_nt, tml->ml, tml->index);
  }
```

as

```
for mechanism in cell_group.mechanisms
  current(mechanism)
end
```

Note that the same level of clarity is possible with C+11:

```
for( auto &mechanism : cell_group.mechanisms() ) {
  mechanism.current();
}
```

4.4 Data

The data layout in the CoreNeuron code is described in more detail in Section 5. For this part of the report, we use a representation of the storage for a group of cells that is easier to understand.

CoreNeuron stores **cell groups**, which are a set of cells, in a `NrnThread` data structure. The "thread" in the type name is misleading, so it is renamed as a `CellGroup` in the definition in Fig. 5.

```
# storage for a group of cells packed into flat arrays
type CellGroup
  Int ncells    # number of cells in group
  Int nnodes    # total number of nodes in all cells

  Array{Int} parent_indices

  # list of all the mechanisms for this cell group
  Array{Any} mechanisms

  # storage for the linear system
  Array{Float} VEC_A
  Array{Float} VEC_B
  Array{Float} VEC_D
  Array{Float} VEC_V
  Array{Float} VEC_RHS
end
```

Figure 5: Data structure for storing the state of a group of cells. Mechanism-specific data is stored in an array of mechanisms, and the matrix state is stored in the vectors `VEC_*`.

The other important abstraction is a **mechanism**. In the definition of `CellGroup`, there is an array called `mechanisms`. Loosley defined, mechanisms are processes that occur at nodes [5], for example a mechanism might define the state of a synapse at a node.

The same mechanism can be applied to more than one nodes in a cell group, e.g. the same synapse type might be present at many different nodes. Pseudo code for a type that stores the information required to define a mechanism for a cell group is in Fig. 6. There are three important sets of information for the mechanism type:

1. Information about which nodes the mechanism is to be applied at, i.e. `nodecount` and `nodeindices`.
2. State information for the mechanism at every node at which it is applied. For example, synapses are modelled with an ordinary differential equation for current, which has multiple fields and parameters. The value of each field and parameter has to be stored at each node. In the example used here this is stored as a structure of array (SoA) format, with one vector of length `nodecount` for each state parameter/variable.

[5] This is generally true. There are some exceptions, but we ignore those for now.

3. External Ion information. Some mechanisms read/write to the state of ion chanels, which are implemented as separate mechanisms. A method for reading/writing the state of another mechanism is required, which we have abstracted with the Ion type, which can be considered to be a reference to the state information stored in the appropriate ion mechanism.

```
# mechanism for calcium
type CaMechanism
  Int        nodecount   # number of nodes with this mechanism
  Array{Int} nodeindices # indices of nodes with this mechanism

  # data fields for every node at which mechanism is applied
  Array{Float} gCabar
  Array{Float} ica
  Array{Float} gCa
  Array{Float} m
  Array{Float} h
  # ...

  # storage for ion channels
  type Ion
    Array{Float} ica
    Array{Float} eca
    Array{Float} icadv
  end
  Ion ion
end

# specialize functions for calcium mechanism
function current(mechanism::CaMechanism)
    # implementation goes here
end
function jacob(mechanism::CaMechanism)
    # implementation goes here
end
function state(mechanism::CaMechanism)
    # implementation goes here
end
```

Figure 6: Definition of the Calcium (Ca) mechanism data type in pseudo code.

4.5 One Time Step

The inner part of each time step is implemented in the function `nrn_fixed_step_thread`, in `nrnoc/fadvance_core.c`. The routine takes as its argument a pointer to a struct of type `NrnThread`. Here we implement it with a `CellGroup` argument, as defined in Fig. 5.

TODO: Add Francesco's diagram of the time step.

```
function nrn_fixed_step(cell_data::CellData)
  ### other ###
  # form the matrix (D vector and RHS)
  setup_tree_matrix_minimal(cell_data)
  # solve the linear system
  nrn_solve_minimal(cell_data)
  # update current values for I/O (not used in calculations)
  second_order_cur(cell_data)
  # advance the solution
  update(cell_data)
  ### other ###
  # update states
  nonvint(cell_data)
  ### other ###
  return None
end
```

A breakdown of wall time for the steps in `nrn_fixed_step` is given in Fig. 12. Some of the routines listed here have less than 1% of wall time (including the linear system solve in `nrn_solve_minimal`), however they are discussed below because their data access influence the implementation on many-core architectures (e.g. GPU and MIC).

4.5.1 Building matrix and RHS: `setup_tree_matrix`

The function `setup_tree_matrix` generates the diagonal, the d_i values, and the RHS vector. These tasks are performed in two separate routines, `nrn_lhs` and `nrn_rhs`.

```
function setup_tree_matrix(cell_group)
  nrn_rhs(cell_group)
  nrn_lhs(cell_group)
end

function nrn_lhs(cell_group)
  # range of non-root nodes
  child_nodes = cell_group.ncells+1:cell_group.nnodes
  p = cell_group.parent_indices

  VEC_D[:] = 0.

  for mechanism in cell_group.mechanisms
    jacob(mechanism)
  end

  for i in child_nodes
    VEC_D[i]    -= VEC_B[i]
    VEC_D[p[i]] -= VEC_A[i]
    end
  end

  function nrn_rhs(cell_group)
    # range of non-root nodes
    child_nodes = cell_group.ncells+1:cell_group.nnodes
    p = cell_group.parent_indices

    # access to cell_group.VEC_* is implicit in the pseudocode
    VEC_RHS[:] = 0.

    for mechanism in cell_group.mechanisms
      current(mechanism)
    end

    for i in child_nodes
      dv             = VEC_V[p[i]] - VEC_V[i]
      VEC_RHS[i]    -= dv * VEC_B[i]
      VEC_RHS[p[i]] += dv * VEC_A[i]
    end
  end
```

The nrn_lhs function sets the values on the diagonal of the matrix, stored in VEC_D. It starts by initializing the VEC_D to zero, the calls the jacob function for each mechanism. The jacob function adds the conrtibution from each mechanism at each node to the relevant entry in VEC_D. The jacob funtion is defined for each mechanism that contributes to the diagonal. The jacob functions are identical for all such mechanisms, except for the capacitance mechanism, which is always the first mechanism thread.mechanisms[1]. Both forms are shown below:

```
  # the jacob function is the same for all other mechanisms
  function jacob(mechanism)
    data = mechanism.data
    ni   = mechanism.nodeindices
    for i in 1:mechanism.nodecount
      VEC_D[ni[i]] += data.g[i]
    end
  end

  # a specialized version of jacob for the capacitance mechanism
  # implemented as nrn_cap_jacob() in CoreNeuron
  function jacob(mechanism::CapacitanceMechanism)
    data = mechanism.data
    ni = mechanism.nodeindices

    cfac = .001 * cj # scaling is due to unit conversion
    for i in 1:mechanism.nodecount
      VEC_D[ni[i]] += cfac * data.cm[i]
    end
  end
```

It is common for multiple mechanisms to be defined on the same node (indeed, often there are many instances of the same mechanism on a single node). As such, it is not possible to replace for loop over the mechanisms that calls the `current` function with a `parallel_forall` loop, because there will be race conditions if different mechansims update the same `VEC_RHS` values simultaneously.

Note 2 *The main challenge in fine-grained parallelism of the time step will always boild down to avoiding such race conditions. It may be possible to work around this issue using **ghost arrays** for each mechanism, into which `VEC_*` arrays are gatherd, or from which they are scattered, before or after looping over a mechanism operation.*

The `current` functions for the mechanisms contribute 45% of time to solution for the TEST2 benchmark. An example of a `current` implementation is given below (specifically, it is the mechanism defined in /`mech`/`cfiles`/`NaTa_t.c`). Note that it uses references to .

```
# definition of current for the NaTa_t mechanism
function current(mechanism::NaTaMechanism)
  data = mechanism.data
  ion  = mechanism.ion  # abstraction for ion channel
  ni   = mechansim.nodeindices

  for i in 1:mechanism.nodecount
    local v = VEC_V[ni[i]];
    data.ena[i] = ion.ena[i];
    # the v+0.001 is to take a numeric derivative
    data.g[i] = nrn_current(data, v + .001);
    local dina = data.ina[i];
    local rhs = nrn_current(data, v);

    ion.dinadv[i] += (dina - data.ina[i]) / .001;
    data.g[i] = (data.g[i] - rhs) / .001
    ion.ina[i] += data.ina[1]
    VEC_RHS[ni[i]] -= rhs;
  end
end

function nrn_current(data, v)
  data.v[i] = v;
  data.gNaTa_t[i] = data.gNaTa_tbar[i] * data.m[i] *
                    data.m[i] * data.m[i] * data.h[i]
  data.ina[i] = data.gNaTa_t[i] * (data.v[i] - data.ena[i])
  return data.ina[i];
end
```

4.5.2 Solving the linear system: `nrn_solve_minimal`

The solution of the linear system using Hines algorithm is straight forward, and is implemented in /`nrnoc`/`solve_core.c`. As described in Section 3.3, the upper triangle is elimnated first with a backward sweep, followed by a forward sweep, which differs from the usual presentation of Gaussian elimination as a forward substitution followed by backwards substitution.

Note 3 *The solution is applied to all of the cells in a `CellGroup` in one sweep. However, the linear systems for the cells are independent, so this can be parallelized by solving for all trees simultaneously, though it would require additional metadata not present in the current `CellGroup` representation.*

All of the nodes in the cell group are numbered such that the root node for all of the trees are numbered 1:ncells, so that for example the values in VEC_D[1:ncells] correspond to the diagonal values on the first row of the matrix of cells 1:ncells. This is evident when we look at the last step in the backward substituion, where only the value in row 1 of the RHS vector is scaled by the diagonal. Instead of being applied to just VEC_RHS[1], it is applied to VECH_RHS[1:ncells].

```
function nrn_solve_minimal(cell_group)
  ncells = cell_group.ncells
  nnodes = cell_group.nnodes
  child_nodes = ncells+1:nnodes

  # backward sweep
  for i in reverse(child_nodes)
    factor          = VEC_A[i] / VEC_D[i]
    VEC_D[p[i]]    -= factor * VEC_B[i]
    VEC_RHS[p[i]] -= factor * VEC_RHS[i]
  end

  # last step of backward sweep: apply to all root nodes
  for i in 1:ncells
    VEC_RHS[i] /= VEC_D[i];
  end

  # forward sweep
  for i in child_nodes
    VEC_RHS[i] -= VEC_B[i] * VEC_RHS[p[i]]
    VEC_RHS[i] /= VEC_D[i]
  end
end
```

4.5.3 Advancing the solution: update

These routines have almost no computational overhead, and are included here for completeness.

The solution to the linear system, stored in VEC_RHS is actually the delta in solution, that is $\text{RHS}_i = V_i^{n+1} - V_i^n$. The update function updates the solution in VEC_V by adding the contribution in VEC_RHS. The update function is the only part of the CoreNeuron code that successfully vectorizes, because of the stride-one data access pattern in the update.

I don't know the purpose of the second_order_cur and nrn_capacity_current routines.

```
# in /nrnoc/eion.c
function second_order_cur(thread)
  if secondorder == 2
    for mechanism in thread.mechanisms
      if is_ion(mechanism)
        mechanism.data.c += mechanism.data.dc .* VEC_RHS
      end
    end
  end
end
```

```
# in /nrnoc/fadvance.c
function update(thread)
  factor = 1.0
  if secondorder==true
    factor = 2.0
  end
  # vectorizable
  VEC_V[:] += factor*VEC_RHS[:]
  # apply to only the first mechanism
  nrn_capacity_current(thread.mechanisms[1]);
end

# in /nrnoc/capac.c
# called by update
function nrn_capacity_current(mechanism)
  data = mechanism.nodeindices
  ni   = mechanism.data
  cfac = .001 * cj;
  for i in 1:mechanism.nodecount
    data.i_cap[i] = cfac * data.cm[i] * VEC_RHS[ni[i]]
  end
end
```

4.5.4 Updating state: `nonvint`

The `nonvint` function is a simple lookup of the `state` function defined for each mechanism. These calls are a significant contribution to computational overheads – greater than 40% for the TEST2 benchmark.

```
function nonvint(thread)
  for mechanism in thread.mechanisms
    state(mechanism)
  end
end
```

The `state` function implementations are derived from the `.mod` implementation. These have obvious potential for vectorization, because they do not appear to have the "parent update" pattern in other loops. However this would require using structore of array (SoA) storage (see the next section). Below is an example of one state update – note the many exponentials (some of which are redundant, i.e.)

```
# specialized on mechanism type
function state(mechanism)
  data = mechanism.data
  ni   = mechanism.nodeindices
  local v
  for i in 1:mechanism.nodecount
    v = VEC_V[ni[i]]
    data.v[i] = v
    data.ena[i] = _ion_ena # TODO pp_var
    states(data)
  end
end

function states(data)
  rates(data)
  data.m[i] = data.m[i] - (1. - exp( - dt / data.mTau[i] ))
                        * (data.mInf[i] + data.m[i])
  data.h[i] = data.h[i] - (1. - exp( - dt / data.hTau[i] ))
                        * (data.hInf[i] + data.h[i])
end

function rates(data)
  local lqt = pow(2.3, ((34.0 - 21.0) / 10.0))
  if (data.v[i] == -38.0)
    data.v[i] = data.v[i] + 0.0001
  end
  data.mAlpha[i] = (0.182 * (data.v[i] - -38.0))
                 / (1.0 - (exp(-(data.v[i] - -38.0) / 6.0)))
  data.mBeta[i]  = (0.124 * (-data.v[i] - 38.0))
                 / (1.0 - (exp(-(-data.v[i] - 38.0) / 6.0)))
  data.mTau[i] = (1.0 / (data.mAlpha[i] + data.mBeta[i])) / lqt
  data.mInf[i] = data.mAlpha[i]/(data.mAlpha[i]+data.mBeta[i])
  if (data.v[i] == -66.0)
    data.v[i] = data.v[i] + 0.0001
  end
  data.hAlpha[i] = (-0.015 * (data.v[i] - -66.0))
                 / (1.0 - (exp((data.v[i] - -66.0) / 6.0)))
  data.hBeta[i]  = (-0.015 * (-data.v[i] - 66.0))
                 / (1.0 - (exp((-data.v[i] - 66.0) / 6.0)))
  data.hTau[i] = (1.0 / (data.hAlpha[i] + data.hBeta[i])) / lqt
  data.hInf[i] = data.hAlpha[i]/(data.hAlpha[i]+data.hBeta[i])
end
```

5 Data Layout

Up to this point the actual layout of data in memory has not been described in detail. Indeed, the pseudo-code examples have treated the data for each mechanism as though it was stored as **structure of arrays** (SoA), when it is in fact stored as **array of scructures**.

Here we will describe the data layout as it is currently implemented. To design the fine-grained parallelization strategy and corresponding data structures, we must first understand the layout of data, and the data-access patterns of the different components.

5.1 Parallel Data Distribution

- Each neuron cell is represented as of a tree of nodes, as illustrated in Fig. 2.
- The properties of individual cells vary significantly, so that the computational resources required to process cells varies.
- The biggest cause of this difference is the distribution of nodes with different mechanism types.
- The `states`/`rates`/`current` kernels of some mechanisms have much higher arithmetic intensity than others.
- The combination of mechanisms in a cell, which varies between different cell types, can be used a-priori estimate it's computational complexity.
- To ensure load balancing, the cells are grouped into groups that have roughly equivalent computational overheads during circuit generation.
- Some groups have more cells than others to ensure that total computational effort required per group is balanced.
- Each group of cells is stored in a separate `.dat` file, which are then distributed in a round-robin fashion when a CoreNeuron simulation is started.
- Parallelism is implemented distributing the cells, with individual cells processed serially.
- There are two levels of parallelism:
  1. **MPI**: the cells are distributed between MPI ranks.
  2. **thread**: the cells on each MPI rank are then assigned to a **thread**.
- Each thread stores the cells assigned to it in a `NrnThread` data structure (see Fig. ??)
- There is one `NrnThread` data structure for each thread.
- Threading is performed using pthreads, with explicit communication of spike information between threads and between MPI processes.

5.2 Thread Storage

Overview of storage of cells in one thread:

- Each thread has multiple cells assigned to it.
- For the TEST2 dataset:
  - there are around 60–70 cells per thread.
  - each cell has of the order 400–450 nodes.
  - each thread has 25,000–30,000 nodes.
- The nodes for all cells are stored in one flat array
  - Given have n_c cells and a total of n nodes, the root nodes are indexed `[1:n_c]`, and the rest of the nodes are indexed `[n_c+1:n]`.
  - This is evident in the loops over `child_nodes=ncells+1:nnondes` and `1:ncells`

Mechanisms in Neuron are implemented using the `.mod` DSL, which is translated into C code. CoreNeuron has the C files allready translated from the `.mod` files that are used by BBP[6]. The translated mechanism definitions are in `mech/cfiles`, with one mechanism per C source file. Each file defines functions (like `jacob`, `current`, `alloc`) and meta-data (such as the number of variables required to store a mechanism's state for a node in the tree).

- The mechanism data is stored in global two arrays: `Memb_list memb_list[]` and `Memb_func memb_func[]`.
  1. The `Memb_func` type has function pointers to the `jacob`, `current`, `state` and other mechanism-specific functions, and other meta-data specific to the mechanism.
  2. The `Memb_list` has a pointer to the per-node data, and a list of all the nodes that the mechanism is defined for.
- The implementation of a mechanism in `mech/cfiles` provides has a function ?? that calls the `register_mech()` function, which adds the mechanisms function callbacks for `jacob` etc, along with meta-data into the global arrays (see Fig. 5.2).
- Each thread has a list of mechanisms assigned to it, which are accessed via a linked list `NrnThread::mechanisms` (see Fig. 5.2 where I have changed the name `tml` to `mechanisms`, to better match the pseudo code.) The linked list indexes the global arrays `memb_list` and `memb_func` *(why not use an array instead of a linked list?)*.
- There are many opportunities to improve the interface between the runtime (solvers etc) and user-defined mechanisms. This model is well-suited to standard object-oriented design. Furthermore, much of the meta-data that is currently passed as runtime parameters could be stored as type-information that could help the compiler optimize more agressively.

Mechanisms and their storage:

- All mechanisms are not applied to different nodes. For example, the `ProbAMPANDMDA_EMS` mechanism will only be applied at a subset of the nodes in a cell.
- Each mechanism has "state" that is stored for each node to which it is applied. This state is a set of double-precision values (e.g. a set of values describing the time evolution of a ordinary differential equation).
- The number of state variables varies between mechanisms, ranging from 3 to 35 values.
- Each entry in `memb_list[]` stores
  - `int nodecount`: the number of nodes to which
  - `int nodeindices[nodecount]`: the indexes of the nodes to which the mechanism is to be applied.
  - `double data[nodecount*var_per_node]`: AoS storage for the mechanism values.

Observations

- Splitting the cells into seperate, thread-specific, data structures complicates the code. This feels like it was added at some point to facilicate threading.
- Could be stripped away, to store all cells on a node/numa-region/device into a single pool.
- The AoS storage is inefficient:
  - It doesn't vectorize (see Fig. 11).
  - Poor cache/bandwidth utlization for loops (such as the `jacob` update) where only one or two data values in a mechanism are touched. For each 64 byte cache line loaded, only 8 bytes of 64 are used.

  An SoA storage would address both issues.

[6]This reduces the complexity of CoreNeuron, decoupling CoreNeuron from Neuron, which will make it easier modify how the mechanisms are defined.

```
struct NrnThread {
  // linked list of Memb_list (mechanisms)
  NrnThreadMembList mechanisms;

  int ncell;      // number of cells
  int nnode;      // number of nodes
  double *data;   // what is this data? Pramod?

  double *rhs; // indexed by VEC_RHS
  double *d;   // indexed by VEC_D
  double *a;   // indexed by VEC_A
  double *b;   // indexed by VEC_B
  double *v;   // indexed by VEC_V
  int *parent_index; // indexed by p
};

// linked list for chaining
struct NrnThreadMembList {
  struct NrnThreadMembList *next;
  struct Memb_list *mech_data; // pointer to memb_list[]
  int index;                   // index for  memb_func[]
};

// holds the data for a mechanism
struct Memb_list { // nrnoc/nrnoc_ml.h
  // indexes of nodes with this mechanism
  // these index into matrix (i.e. VEC_*)
  int *nodeindices;
  double *data;
  int nodecount;
};

// function pointer for mechanism vtable implementation
typedef void (*mod_f_t)(struct NrnThread *, Memb_list *, int);
struct Memb_func { // nrnoc/membfunc.h
  mod_f_t current;
  mod_f_t jacob;
  mod_f_t state;
}

// NrnThread::mechanisms list indexes into here
// declared in nrnoc/register_mech.c
Memb_func* memb_func = malloc(memb_func_size*sizeof(Memb_func));
Memb_list* memb_list = malloc(memb_func_size*sizeof(Memb_list));

// example loop
NrnThreadMembList* mech;
for (mech = &thread->mechanisms; mech!=nullptr; mech = mech->next) {
  if (memb_func[mech->index].jacob) {
    mod_f_t s = memb_func[mech->index].jacob;
    (*s)(thread, mech->mech_data, mech->index);
  }
}
```

Figure 7: The thread (`NrnThread`), mechanism data (`Memb_list`) and mechanism functionality (`Memb_func`) types. I have removed and renamed much of the members to make them better match the pseudo code. Some C++ coding style has also been used.

```
//////////////////////////////////////////////////
// in /nrnoc
//////////////////////////////////////////////////
// allocate empty storage for storing n mechanisms
void alloc_mech(int n) {
  memb_func_size_ = n;
  n_memb_func = n;
  memb_func = ecalloc(memb_func_size_, sizeof(Memb_func));
  memb_list = ecalloc(memb_func_size_, sizeof(Memb_list));
  // ...
}
// called from the .mod generated mechanism definition
// registers-
int register_mech(
        const char **m,  // strings with mech name & meta data
        mod_alloc_t alloc, mod_f_t cur, mod_f_t jacob,
        mod_f_t stat, mod_f_t initialize, ...) {
  int type; // 0 unused, 1 for cable section
  type = nrn_get_mechtype(m[1]);

  // ...
  memb_func[type].current = cur;
  memb_func[type].jacob = jacob;
  memb_func[type].state = stat;
  // ...
  memb_list[type].nodecount = 0;
  // ...
  return type;
}
//////////////////////////////////////////////////
// in mech/cfiles/
// example registering a mechanism NaTs2_t
//////////////////////////////////////////////////
static void nrn_alloc(), nrn_init(), nrn_state();
static void nrn_cur(), nrn_jacob();
static const char *_mechanism[] = {
  "6.2.0", "NaTs2_t", "gNaTs2_tbar_NaTs2_t", 0,
  "ina_NaTs2_t", "gNaTs2_t_NaTs2_t", 0,"m_NaTs2_t",
  "h_NaTs2_t", 0, 0};

_NaTs2_t_reg() {
  // ...
  register_mech(_mechanism,
    nrn_alloc, nrn_cur, nrn_jacob, nrn_state, nrn_init,
    hoc_nrnpointerindex, 1);
  // ...
}
```

Figure 8: The routines used in registering a mechanism

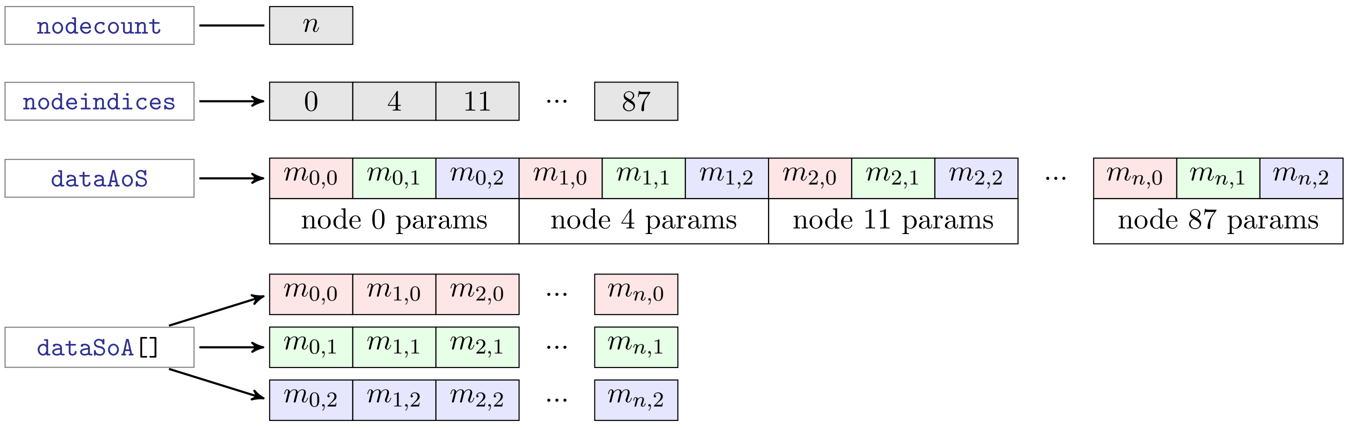


Figure 9: The current AoS layout of mechanism parameters for all applicable nodes.

- With SoA vectorization of most loops would still not be possible, because of the gather/scatter implicit in using the node indexes to read/write to the V and RHS vectors.
  - Perform scatter before computing the current, then gather after.

6 Benchmarking

Here we present preliminary benchmarking results based on the benchmark datasets distributed with the PCP benchmark. The benchmark data is untarred to the paths `VENDORS/TEST1_CACHE` and `VENDORS/TEST2_DRAM`, which contain data files $[0-9]*$`.dat` that define $2^{14} = 16,384$ and $2^{15} = 32,768$ individual cells respectively. The number of cells to use in a simulation is set in the first line of the file `files.dat`. The cells are assigned to individual MPI ranks in a round-robin fashion, e.g. 32 files with 8 MPI ranks will assign $32/8 = 4$ cells per rank.

Tbl. 6 shows the wall time for performing 100 time steps with the TEST2 dataset as the number of MPI ranks and the number of cells-per-rank are varied on Piz Daint, which has a eight-core Sandy Bridge socket on each node.

- There are 8 MPI ranks per node, 1 per core, so the number of MPI ranks in 8× the number of nodes.
- The amount of memory available per node, 32 GB limits the number of cells to 128 per node (16 per MPI rank).
- Scaling was performed from 1 to 512 nodes (8 to 4096 MPI ranks).
  - With one cell per core, there is an efficiency of 95.6% from 1–512 nodes
  - As the number of cells per core is increased both wall time per cell and scaling improve, indeed it is slightly better than perfect at 100.1% for 1–512 nodes and 8 cells per core.
- There are some anomilies, with some two simulations (marked in blue) with 8 cells per core finishing in half the time of the others, and one simulation with 16 cells per core failing because it ran out of memory.
  - The cells distributed amongst the cores are not all equal in size, so some variability is expected.
  - Modifying the total number of cells in a simulation will change the number of connections between neurons, which will modify behaviour.
- From this informal analysis, scaling from 1 to 512 nodes appears to be very good.
  - Communication overheads do not appear to be significant.
  - The focus should be on in-node performance.

	cells-per-core	1	2	4	8	16
	cells-per-node	8	16	32	64	128
nodes	1	390.8	383.6	381.6	380.5	381.3
	2	390.9	385.1	384.0	385.3	382.5
	4	394.8	392.7	389.9	194.1	382.6
	8	400.4	392.8	385.9	382.7	381.0
	16	401.9	394.3	388.6	384.2	381.8
	32	401.7	393.6	387.0	383.8	DNF
	64	403.0	395.7	390.0	385.4	382.3
	128	411.0	396.1	390.4	385.5	378.7
	256	411.4	397.0	390.7	195.5	374.9
	512	413.5	396.8	389.2	377.7	–

Table 2: Wall time for TEST2 as both the number of nodes and cells-per-node are varied. Each test always use 8 cores/MPI ranks per-node. The times in red indicate unexplained timings, and those marked DNF did not finish due to memory restrictions (it is not possible to run with more than 16 cells per node due to memory restrictions.)

6.1 Performance Counters

To profile the code I tried using Scorep and the Cray perftools, with varying success. The Scorep tool was useful, however it had high sampling overheads overheads. The Cray perftools caused a segmentation faults and other runtime errors errors.

To obtain reliable results without any sampling overhead, the papi-wrap[7] library was used. This is a simple C++ wrapper around the Papi library for hardware counters, with a simple C API for adding samplers to source code, as shown in Fig. 10.

6.2 Breakdown of time to solution

6.3 Profiling With Scorep

TODO: Describe steps required to get test with Scorep/Cube/Vampir.

[7] `github.com/bcumming/papi−wrap`

```
////////////////////////////////////////////////////////////
// in nrnmpi/nrmpi.c
////////////////////////////////////////////////////////////
...
#include <papi_wrap.h>

// declare handles as global variables
int pw_handle_rhs_current = -1;
int pw_handle_rhs_update = -1;
int pw_handle_nonvint = -1;
...

void nrnmpi_init(...) {
  ...
  // register handles
  pw_handle_rhs_current = pw_new_collector("rhs_current");
  pw_handle_rhs_update = pw_new_collector("rhs_update");
  pw_handle_nonvint = pw_new_collector("nonvint");
  ...
}

void nrnmpi_finalize(void) {
  // each MPI rank prints results table on cleanup
  pw_print_table();
  MPI_Finalize();
}

////////////////////////////////////////////////////////////
// in code to sample
////////////////////////////////////////////////////////////
static void nrn_rhs(NrnThread *_nt) {
    ...
  pw_start_collector(pw_handle_rhs_update);
  for (i = i1; i < i3; ++i) {
    VEC_RHS(i) = 0.;
  }
  pw_stop_collector(pw_handle_rhs_update);

  pw_start_collector(pw_handle_rhs_current);
  for (tml = _nt->tml; tml; tml = tml->next) {
    if (memb_func[tml->index].current) {
      mod_f_t s = memb_func[tml->index].current;
      (*s)(_nt, tml->ml, tml->index);
    }
  }
  pw_stop_collector(pw_handle_rhs_current);
    ...
}
```

Figure 10: Steps to add papi-wrap sampling to CoreNeuron.

```
---------------------------------------------------------------
collector                  time       %    PAPI_DP_OPS  PAPI_VEC_DP
---------------------------------------------------------------
rhs_current            346.3654   45.10   152884436985   3990534760
nonvint                318.4637   41.47   118203493855   5812061816
lhs_jacob               84.7244   11.03     9945680740            0
other                    8.7107    1.13        4664371         1548
matrix_solve             4.0380    0.53     2097774870       770040
rhs_update               2.4865    0.32     1320459042            0
lhs_update               1.4597    0.19      607782930            0
update                   0.9977    0.13      749043581    239121640
lhs_cap_jacob            0.7062    0.09      546709757            0
second_order_current     0.0153    0.00           8188            0
TOTAL                  767.9675
---------------------------------------------------------------
```

Figure 11: Sample output from papi-wrap.

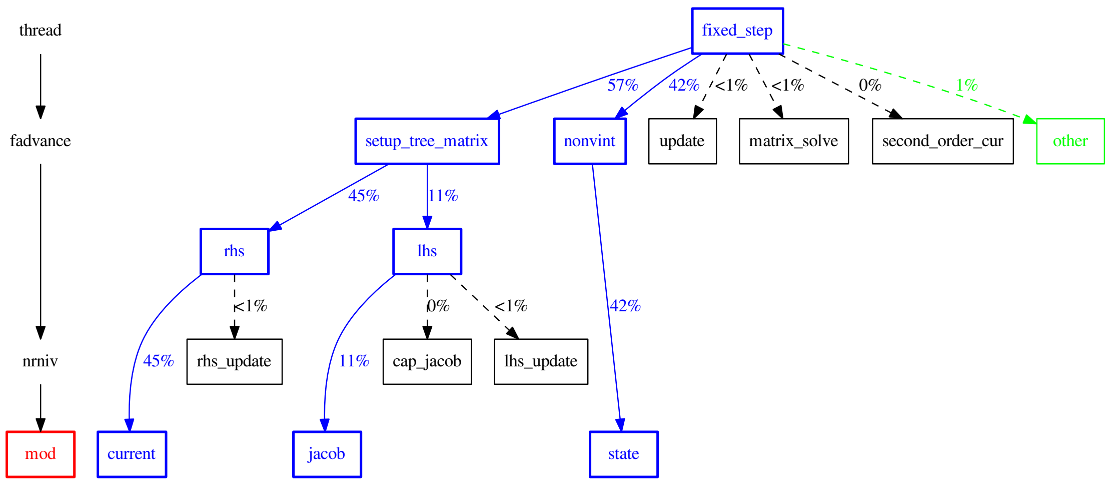


Figure 12: Calltree and percentage of wall time contribution for the main computational algorithm. Branches marked in blue indicate a significant contribution to wall time.

A Discretization Details

A.1 One Dimensional Discretization

Francesco, this would be a good spot to put any extra details.

The spatial and temporal discretization used in Neuron is finite difference with a Crank Nicholson time stepping scheme. The time stepping scheme has an odd form, that differs slightly from the scheme presented here. However, the exposition here adequately characterizes the code.

Finite Difference Methods in Heat Transfer, Second Edition By Necati Ozisik (1994) gives the following finite difference discretization

$$\begin{aligned}\Delta x\left(C_i\frac{V_i^{n+1}-V_i^n}{\Delta t}+I_i^{n+\frac{1}{2}}\right) &= \\ &f_i\theta\left[g_{i-\frac{1}{2}}\frac{V_{i-1}^{n+1}-V_i^{n+1}}{\Delta x}+g_{i+\frac{1}{2}}\frac{V_{i+1}^{n+1}-V_i^{n+1}}{\Delta x}\right] \\ &+f_i(1-\theta)\left[g_{i-\frac{1}{2}}\frac{V_{i-1}^{n}-V_i^{n}}{\Delta x}+g_{i+\frac{1}{2}}\frac{V_{i+1}^{n}-V_i^{n}}{\Delta x}\right] \qquad (6)\end{aligned}$$

where the parameter θ can be used to determine the time stepping scheme ($\theta = 0$ explicit, $\theta = 1$ implicit, $\theta = 1/2$ Crank-Nicholson).

Note that the quantity I depends on the value of V. These values have to be evaluated to form coefficients in the linear system that is to solved at each time step. The approach taken by Neuron is the evaluate them at a half time step value at $t + \Delta t/2$, denoted $I_i^{n+\frac{1}{2}}$, by implicitly solving for a half step, then performing an explicit half step.

With a Crank Nicholson time stepping scheme, i.e. $\theta = 1/2$, the terms on the right hand side of (6) simplify to

$$\begin{aligned}&\frac{f_i}{2}\left\{\left[g_{i-\frac{1}{2}}\frac{V_{i-1}^{n+1}-V_i^{n+1}}{\Delta x}+g_{i+\frac{1}{2}}\frac{V_{i+1}^{n+1}-V_i^{n+1}}{\Delta x}\right]\right. \\ &\left.+\left[g_{i-\frac{1}{2}}\frac{V_{i-1}^{n}-V_i^{n}}{\Delta x}+g_{i+\frac{1}{2}}\frac{V_{i+1}^{n}-V_i^{n}}{\Delta x}\right]\right\} \qquad (7)\end{aligned}$$

Substituting the spatial discretization in (7) into (6) and rearranging gives a tri-diagonal linear system with the form

$$a_iV_{i+1}^{n+1}+d_iV_i^{n+1}+b_iV_{i-1}^{n+1}=r_i \qquad (8)$$

where the diagonals in the matrix are defined

$$\begin{aligned}a_i &= -\frac{f_ig_{i+\frac{1}{2}}}{2\Delta x^2} && (9)\\ b_i &= -\frac{f_ig_{i-\frac{1}{2}}}{2\Delta x^2} && (10)\\ d_i &= \frac{C_i}{\Delta t}+\frac{f}{2\Delta x^2}\left[g_{i-\frac{1}{2}}+g_{i+\frac{1}{2}}\right] \\ &= \frac{C_i}{\Delta t}-(a_i+b_i) && (11)\end{aligned}$$

and

$$\begin{aligned}r_i &= \frac{C_i}{\Delta t}V_i^n-I_i+\frac{f_i}{2\Delta x^2}\left[g_{i-\frac{1}{2}}\left(V_{i-1}^n-V_i^n\right)+g_{i+\frac{1}{2}}\left(V_{i+1}^n-V_i^n\right)\right] \\ &= \frac{C_i}{\Delta t}V_i^n-I_i-a_i\left(V_{i-1}^n-V_i^n\right)-b_i\left(V_{i+1}^n-V_i^n\right) && (12)\end{aligned}$$

The off-diagnoal coefficients in the linear system, i.e. **a** and **b**, are constant in time, and can be computed once at start up. The values on the diagonal and right hand side vector, i.e. **d** and **r**, are computed every time step, and overwritten when solving the linear system using Thomas' algorithm. The values stored in **a** and **b** are used when recomputing them.

The current, I, also makes a contribution to the diagonal due to it's dependence on V, which I have not included here.