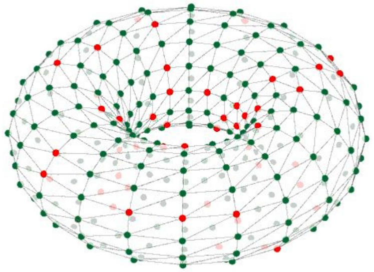
Graph Front End - Advanced



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

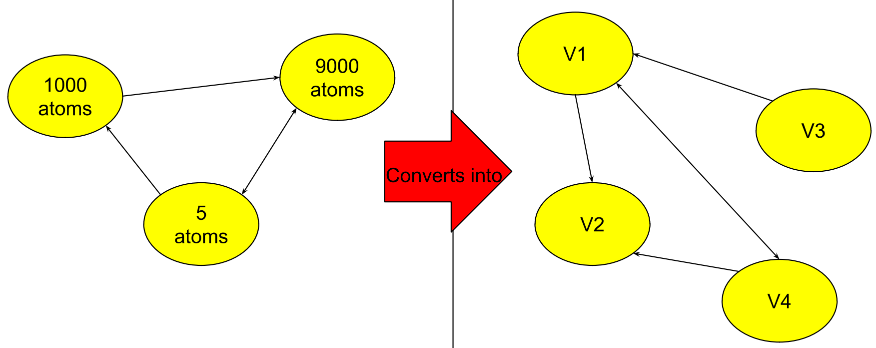
Contents

- Working with application graphs
- Buffered recordings
- Auto pause and resume
- Provenance data

Supported graphs (PACMAN)

Application Graph

Machine Graph



Needs breaking down into core sized chunks

Already has a 1:1 ratio between vertices and core.

Basic script to add application vertices into the graph

```
import spinnaker_graph_front_end as front_end

from spinnaker_graph_front_end.examples.Conways.conways_application_cell\
   import ConwayApplicationCell

# set up the front end and ask for the detected machines dimensions
front_end.setup()

front_end.add_application_vertex_instance(
      ConwayApplicationCell(800, "ConwayCells"))

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```

Creating a new type of application vertex

```python
from pacman.model.graphs.application.impl.application_vertex  import ApplicationVertex
from pacman.model.resources.resource_container import ResourceContainer
from pacman.model.resources.cpu_cycles_per_tick_resource import CPUCyclesPerTickResource
from pacman.model.resources.dtcm_resource import DTCMResource
from pacman.model.resources.sdram_resource import SDRAMResource

class ConwayApplicationCell(ApplicationVertex):
    """ Represents a collection of cells within the 2D grid
    """
    def __init__(self, n_atoms, label):
        ApplicationVertex.__init__(self, label=label, max_atoms_per_core=200)
        self._n_atoms = n_atoms

    def get_resources_used_by_atoms(self, vertex_slice):
        resources = ResourceContainer(
            sdram=SDRAMResource(4 * vertex_slice.n_atoms),
            dtcm=DTCMResource(4 * vertex_slice.n_atoms),
            cpu_cycles=CPUCyclesPerTickResource(100 * vertex_slice.n_atoms)
        )
```


Creating a new type of application vertex

```python
    def create_machine_vertex(
        self, vertex_slice, resources_required, label=None, constraints=None):

        # return a partitioned vertex that's designed to handle multiple atoms within it
        return ConwayMachineCell(
            label=label, resources_required=resources_required,
            constraints=constraints)

    @property
    def n_atoms(self):

        # return the atoms this vertex contains
        return self._n_atoms
```


Basic Script adding application edges

```python
import spinnaker_graph_front_end as front_end

# build and add application vertex
vertex = ConwayApplicationCell(800, "ConwayCells")
front_end.add_application_vertex_instance(vertex)

# build an application edge
front_end.add_application_edge_instance(
    ApplicationEdge(vertex, vertex), "State")

front_end.run(5000)

front_end.stop()
```

Partition id

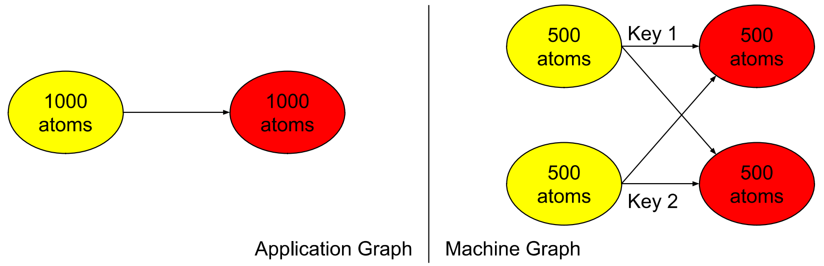
Data generation

```python
...
def generate_application_data_specification(
        self,  spec, placement, graph_mapper, application_graph, machine_graph,
        routing_info, iptags,  reverse_iptags, machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
    region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
    region=1, size=8, label="inputs")
...

# get slice of atoms for machine vertex
vertex_slice = graph_mapper.get_slice(placement.vertex)
```

Application vertex c code



Hints:

1. You need to be able to distinguish from the received key which atoms it effects on the core you are writing the data for
2. You need to execute your application c code for every atom on the core

Buffered Recordings

Problem

1. SDRAM is limited on the SpiNNaker machines.
2. Recording of data is more reliable on SDRAM than live transmissions.
3. Simulations run for long periods of time gathering data.

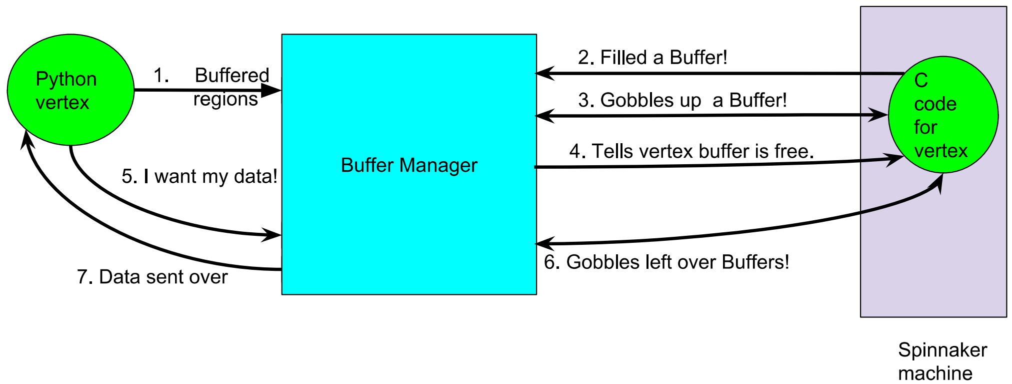
Buffered Recordings

Solution

1. Store data in small chunks called buffers
2. During simulation, or during a pause, extract the buffers

NOTE: This only works in tandem with the simulation.h and data_specification.h and python interfaces.

How does a extracted buffered data region work?



MANCHESTER 1824

Buffered Recording - Python

```
class MyBufferedVertex(..., ReceiveBuffersToHostBasicImpl):

def __init__(...):
   ReceiveBuffersToHostBasicImpl.__init__(self)
   ...
```


MANCHESTER 1824

Buffered Recording - Python

```
class MyBufferedVertex(..., ReceiveBuffersToHostBasicImpl):
  ...

   def generate_data_spec(...):
      ...
      spec.reserve_memory_region(
         region=2, size=self.get_recording_data_size(3), label="recording")
      ...
      spec.reserve_memory_region(
         region=6, size=self.get_buffer_state_region_size(3), label="state")
      ...

      ...
      self.reserve_buffer_regions(spec, 6, [4,5,7], [1000000, 1000000, 100000])
      ...
      spec.switch_write_focus(2)
      self.write_recording_data(spec, iptags, [1000000, 1000000, 100000], 16384)
      ...
```

Number of buffered regions

Extra region for storing buffered state

Buffered region ids

Allocated buffer sizes

IP Tags holder

Buffer size before request sent

MANCHESTER 1824

Buffered Recording - C

```
static uint32_t recording_flags = 0;

void c_main(void) {
    ...
    address_t address = data_specification_get_data_address();
    address_t recording_region = data_specification_get_region(2, address);
    uint8_t *regions_to_record[] = {4,5,7};

    bool success = recording_initialize(
       3, regions_to_record, recording_region, 6, &recording_flags);

    ...
    simulation_run();
}
```

Buffered region ids (channels 0, 1 and 2)

Number of buffered regions

Extra region for storing buffered state

MANCHESTER 1824

Buffered Recording - C

```
...
void timer_callback(uint unused0, uint unused1) {
   ...
   if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
      recording_finalize();
      ...
   }

if (recording_is_channel_enabled(recording_flags, 0)) {

      uint32_t data = 23;
      recording_record(0, &data, 4);
   }

   recording_do_timestep_update(time);
   ...
}
```

Recording channel number (= region 4)

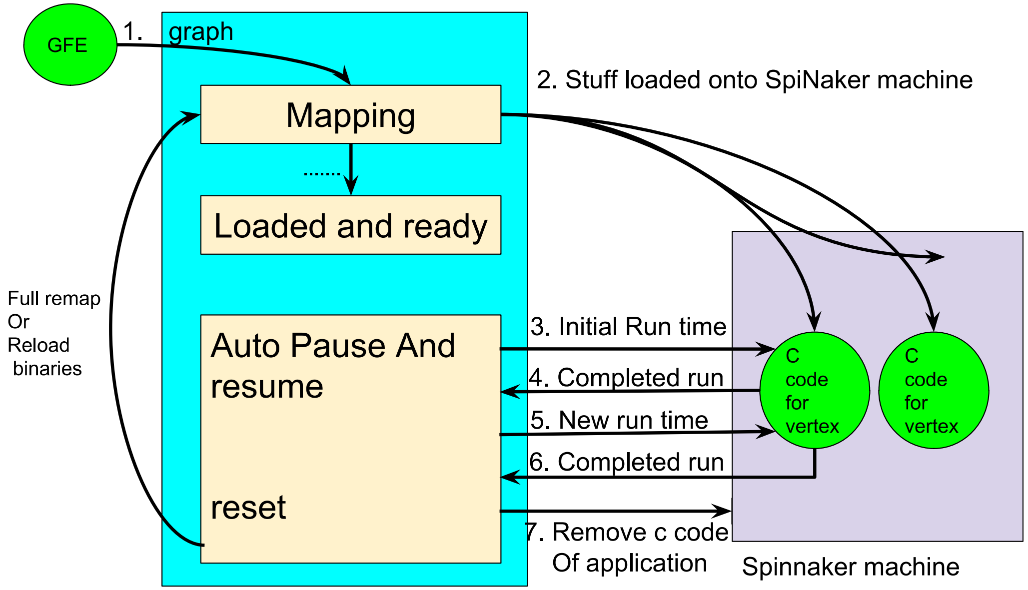
Pointer to data to record

Size of data to record in bytes

Auto pause and resume functionality

1. Provides the ability to run a simulation for multiple periods without remapping the application.
2. Provides the ability to extract buffers without affecting the running simulation.
3. Supports the ability to reset a simulation to the state at t=0.

How Auto Pause and Resume works.



Auto Pause and Resume - Python

```
class AbstractPopulationVertex(..., AbstractChangableAfterRun):
...
def __init__(......):
    AbstractChangableAfterRun.__init__(self)

    # bool for if state has changed.
    self._change_requires_mapping = True

@property
def requires_mapping(self):
    # determine if there are changes within which require a remapping
    return self._change_requires_mapping

def mark_no_changes(self):
    # restart the tracking of changes
    self._change_requires_mapping = False

def set_recording_spikes(self):
   self._change_requires_mapping = not self._spike_recorder.record
   self._spike_recorder.record = True
```


Auto Pause and Resume - C

```
...
void timer_callback(uint unused0, uint unused1) {
    ...

    if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
        simulation_exit();
        simulation_handle_pause_resume(resume_callback);
        ...
    }
}

void resume_callback() {

    // restart the recording just before resuming
    if (!initialise_recording()) {
        rt_error(RTE_SWERR);
    }
}
```

Provenance data gatherers

1. Data that can be used to prove 2 simulations are equivalent to each other.
2. Data that can also be used for debug purposes.
3. Is stored in XML and searched through for errors by the main tools.
4. Every vertex can provide its own provenance data.

Provenance example output

```xml
<provenance_data_items name="my_object">
  <provenance_data_items name="my_category">
    <provenance_data_item name="my_item">0</provenance_data_item>
    <provenance_data_item name="my_other_item">0</provenance_data_item>
  </provenance_data_items>
</provenance_data_items>

<provenance_data_items name="0_0_5_my_vertex">
  <provenance_data_items name="my_category">
    <provenance_data_item name="my_machine_value">0</provenance_data_item>
  </provenance_data_items>
</provenance_data_items>
```


Local Provenance Data - Python

```python
class MyVertex(..., AbstractProvidesLocalProvenanceData):
    ...

    def get_local_provenance_data(self)
        self._data_items = list()

        # store data in a provenance data item
        self._data_items.append(
            ProvenanceDataItem(
                ["my_object" "my_category", "my_item"], my_value))
        self._data_items.append(
            ProvenanceDataItem(
                ["my_object", "my_category", "my_other_item"], my_other_value,
                report=(my_other_value > error_value),
                message="value {} was bigger than expected ({})".format(
                    my_value, error_value))
        ...

        # return provenance items
        return self._data_items
```

Hierarchy of categories and names used to group items in XML

debug arguments

Simulation Provenance Data - Python

```python
class MyVertex(..., ProvidesProvenanceDataFromMachineImpl):
    ...
    def get_provenance_data_from_machine(self, transceiver, placement):
        provenance_data = self._read_provenance_data(transceiver, placement)

        # translate system specific provenance data items
        provenance_items = self._read_basic_provenance_items(
            provenance_data, placement)

        # translate application specific provenance data items
        provenance_data = self._get_remaining_provenance_data_items(
                provenance_data)
        my_value = provenance_data[0]
        label, x, y, p, names = self._get_placement_details(placement)

        # translate into provenance data items
        provenance_items.append(
            ProvenanceDataItem(
                self._add_names(names, ["my_category", "my_machine_value"]),
                my_value))

        return provenance_items
```

Simulation Provenance Data - Python

```
class MyVertex(..., ProvidesProvenanceDataFromMachineImpl):
    ...

    def __init__(self, ...)
        ProvidesProveanceDataFromMachineImpl.__init__(self, 9, 1)
        ...

    def generate_data_spec(...):
        ...
        self.reserve_provenance_data_region(spec)
```

Provenance Region

Number of custom provenance data items

Simulation Provenance Data - C

```
static my_value = 0;

void c_main(void) {

    ...
    if (!simulation_initialise(
            system_region, APPLICATION_NAME_HASH,
            &timer_period, &simulation_ticks,
            &infinite_run, SDP,
            get_provenance_data,
            data_specification_get_region(9, address))) {
        log_error("Error in initialisation - exiting!");
        rt_error(RTE_SWERR);
    }
    ...
}

void get_provenance_data(address_t provenance_data_address) {
    provenance_data_address[0] = my_value;
}
```

Provenance Region

Summary

1. Application graphs
2. Buffered recording
3. Auto pause and resume
4. Provenance data gathering