SpiNNaker Graph Front End
New Application Lab Manual

This document informs a end user of the SpiNNakerGraphFrontEnd's interfaces and requirements needed to allow a new application to run via the SpiNNakerGraphFrontEnd Interface.
Note: for developers of this document, a lot of this document can be removed once ReadTheDocs is back up and running and the python code docs are updated with the words in this document.

Contents

Sections

Tables

Figures

Code

Requirements

To use the SpiNNakerGraphFrontEnd (GFE), any developer needs to be able to describe their problem within a graph, referred to as an application graph, representation where nodes represent computational load and edges represent communication between these computational nodes.

The GFE supports two types of application graph, referred to as Partitionable and Partitioned graphs. The vertices represented within the *partitioned graph* reside on one spinnaker core only, whereas vertices within the *partitionable graph* can reside on multiple cores and need to partitioned.

During the development of the application code to run on SpiNNaker through the GFE, the developer will need to write a c programme, at least one python class and a python script. This is broken down as follows:

1. The c code represents the code that will actually run on the SpiNNaker machine, discussed in Section SpiNNaker Graph Front End: c Code Interface.
2. The python class which represents the computational node, referred to as a vertex, which can be either a *partitioned vertex* if the computation node needs to reside on one core, or both a *partitioned vertex* and a *partitionable vertex* if the node can be split over multiple cores. This is discussed in more detail in Section SpiNNaker Graph Front End: Computational Node Interface.
3. If there is special context needed for the communication between the computational nodes, then the end user may be required to construct a Edge class. This is discussed in more detail in Section SpiNNaker Graph Front End: Edges Interfaces.
4. The python script that contains the construction of the application graph, the commands to setup, execute, and end the GFE. This is discussed in Section SpiNNaker Graph Front End: Python Script Interface.

The developer is expected to have installed the GFE in developer mode, which can be done by following:

http://spinnakermanchester.github.io/2016.001.AnotherFineProductFromTheNonsenseFactory/spinnaker_graph_pages/SpiNNakerGraphFrontEndDeveloperInstall.html

Throughout this document, we will be comparing what we learn with the example code called the HeatDemo. The code can be found in your example folder for the GFE installation, but in case you cannot find this, the link is below:

https://github.com/SpiNNakerManchester/SpiNNakerGraphFrontEnd/tree/master/examples/heat_demo

SpiNNaker Graph Front End: Python Script Interface

The GFE interface supports a collection of calls and ways of instantiating itself. The GFE can be used in python by either importing the module directly or by instantiating the main class as shown below:

```
import spinnaker_graph_front_end as front_end
front_end.setup(
        hostname=None, graph_label=None, model_binary_module=None,
        model_binary_folder=None, database_socket_addresses=None,
        user_dsg_algorithm=None)
```

Code 1: Importing the Graph Front End directly.

```
from spinnaker_graph_front_end.spinnaker import SpiNNaker
front_end = Spinnaker(
        host_name=None, timestep=None, min_delay=None, max_delay=None,
        graph_label=None, database_socket_addresses=None)
```

Code 2: Instantiating the Graph Front End.

As you can see from these two forms of initialization, there are a lot of parameters which you can add to the constructor, but none of them are essentially required and are only exposed at this level for end users who wish to operate in a more advanced fashion than what's intended. During this document, we will cover each of these parameters individually when they become apparent.

We recommend importing the module directly (as shown in **Code 1**), as it then adds some helpful objects and variables users will need without needing to know where they are stored. For example, by using **Code 1**, you can import the LivePacketGather, ReverseIpTagMultiCastSource and MultiCastPartitionedEdge directly through the usage of **Code 3**, otherwise you need to know the import paths shown in **Code4**.

```
Front_end.LivePacketGather
Front_end.ReverseIpTagMultiCastSource
Front_end.MultiCastPartitionedEdge
Front_end.MultiCastPartitionableEdge
Front_end.LivePacketGatherPartitionedVertex
Front_end.ReverseIPTagMulticastSourcePartitionedVertex
```

Code 3: Direct importing of useful classes.

```
from spinn_front_end_common.utility_models.live_packet_gather import LivePacketGather
from spinn_front_end_common.utility_models.reverse_ip_tag_multi_cast_source import ReverseIpTagMultiCastSource
from pacman.model.partitioned_graph.multi_cast_partitioned_edge import MultiCastPartitionedEdge
from pacman.model.partitionable_graph.multi_cast_partitionable_edge  import MultiCastPartitionableEdge
from spinn_front_end_common.utility_models.live_packet_gather_partitioned_vertex import  LivePacketGatherPartitionedVertex
from spinn_front_end_common.utility_models.reverse_ip_tag_multicast_source_partitioned_vertex import ReverseIPTagMulticastSourcePartitionedVertex
```

Code 4: Importing via source.

The main spinnaker object (no matter how you import it) provides a collection of functions as a simple interface to help end users to operate the SpiNnaker software. this interface is described in **Table 1**.

Name	Inputs	Outputs	Definition
setup()	**hostname**: (string) Ipaddress of the machine **graph_label**: (string) human readable name of the application **executale_finder**: instance of spinn_front_end_common.utilities.utility_objs.executable_finder.ExecutableFinder **database_socket_addresses**:	None	This initializes the tool chain and reads the .spinnakerGraphFrontEnd.cfg file and sets up basic variables.

	iterable of spinn_front_end_common.utilities.notification_protocol.socket_address.SocketAddress **extra_algorithms_for_auto_pause_and_resume**: iterable of string.		
add_partitionable_vertex_instance()	**vertex_to_add**: instance of an application vertex **constraints**: iterable of pacman.model.constraints.abstract_constraints.abstractConstraint.AbstractConstraint	None	Adds a instainted partitionable vertex into the partitionable graph.
add_partitionable_vertex()	**cellclass:**(class) the class to instantiate **cellparams:** (dict) of parameter name to value used by the cell class **label:** (string) human readable name of the vertex. **constraints:** iterable of pacman.model.constraints.abstract_constraints.abstractConstraint.AbstractConstraint	The instantiated vertex	Instantiates and adds the partitionable vertex into the partitionable graph.
add_vertex()	**cellclass:**(class) the class to instantiate **cellparams:** (dict) of parameter name to value used by the cell class **label:** (string) human readable name of the vertex. **constraints:** iterable of pacman.model.constraints.abstract_constraints.abstractConstraint.AbstractConstraint	The instantiated vertex	instantiates and adds the partitioned vertex into the partitioned graph.
add_vertex_instance()	**vertex_to_add**: instance of an application vertex	None	Adds a instainted partitioned vertex into the partitioned graph.
add_partitionable_edge()	**cellclass:**(class) the class to instantiate **cellparams:** (dict) of parameter name to value used by the cell class **label:** (string) human readable name of the edge. **constraints:** iterable of pacman.model.constraints.abstract_constraints.abstractConstraint.AbstractConstraint **partition_id**: (string) Label of the partition to consider for graph routing.	The instantiated Edge	instantiates and adds the partitionable edge into the partitionable graph.
add_partitionable_edge_instance()	**edge**: instance of an application edge **partition_id**: (string) Label of the partition to consider for graph routing.	None	Adds a instantiated partitionable edge into the partitionable graph.
add_edge()	**cellclass:**(class) the class to instantiate	The	instantiates and adds the partitioned

	cellparams: (dict) of parameter name to edge used by the cell class **label:** (string) human readable name of the edge. **constraints:** iterable of pacman.model.constraints.abstract_constraints.abstractConstraint.AbstractConstraint	instantiated Edge	edge into the partitioned graph.
add_edge_instance()	**edge**: instance of an application edge **partition_id**: (string) Label of the partition to consider for graph routing.	None	Adds a instantiated partitioned edge into the partitioned graph.
read_partitionable_graph_xml_file()	**file_path**: (string) to a xml file	None	Adds the vertices and edges described in the xml file into the partitionable graph.
read_partitioned_graph_xml_file()	**file_path**: (string) to a xml file	None	Adds the vertices and edges described in the xml file into the partitioned graph.
get_machine_dimensions()	None	dict with keys ‘x’, ‘y’	Either returns the dimensions of the virtual machine or tries to power on, boot and discover a real SpiNNaker machine.
machine()	None	SpinnMachine instance	Returns the python representation of the SpiNNaker machine (real or virtual).
transceiver()	None	SpiNNMan instance	returns the python interface for communicating with a real SpiNNaker machine.
run()	**runtime**: (int) in milliseconds	None	maps, loads, runs and validates an executing application on a SpiNNaker machine. If using a virtual machine, only runs mapping.
reset()	None	None	turns off the executable code running on the SpiNNaker machine, but keeps router tables, ip_tags etc loaded.
stop()	**turn_off_machine**: (boolean) overrides the default behaviour in powering off	None	turns off the executable code

	the SpiNNaker machine. **clear_routing_tables**: (boolean) overrides the default behaviour of removing the routing tables from the SpiNNaker machine. **clear_tags**: (boolean) overrides the default behaviour of removing the tags from the SpiNNaker machine.		running on the SpiNNaker machine and removes the router tables, ip_tags etc and powers off the machine if requested.

Table 1: Standard functions provided by the SpiNNakerGraphFrontEnd interface.

It is worth noting that the collection of edges which leave a vertex can be split into partitions, as described in the interface functions *add_edge_instance()*, and *add_partitionable_edge_instance()*. These partitions represent separate communication paths where the edges within each partition share the same keys and route. Figure 1 shows this in a visual one.

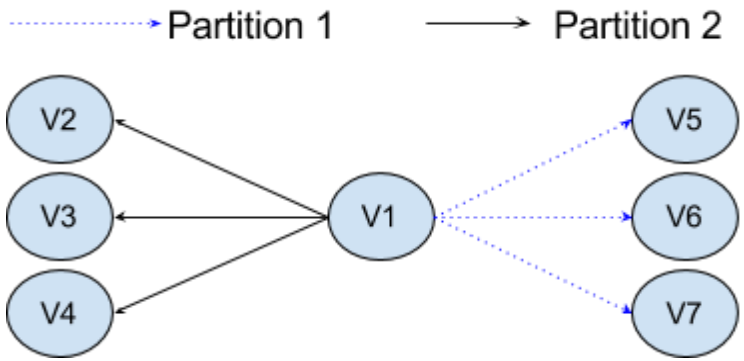

Figure 1: Representation of Edge partitions

At this point, it should be worth defining some general terms which we use throughout the document. These terms have usually been defined in other presentations within the workshop, but we cover them here for completeness.

Parameter	**Definition**
machine_time_step	The real time in microseconds between timer ticks
timescale_factor	The multiplying factor used to slow down the simulation whilst keeping the simulation in a given time step mentality.

Table 2: basic parameters definitions.

Heat Demo Example

We will now go through the heat demo example and show how it utilizes the spinnaker interface. If we look at the HeatDemo.py located in https://github.com/SpiNNakerManchester/SpiNNakerGraphFrontEnd/blob/master/examples/heat_demo/heat_demo.py we will find a python script which on line 29, as shown in **Code 5**, calls the setup() command with the parameters:

```
graph_label="heat_demo_graph"

model_binary_module=model_binaries
```

Code 5: inputs to front_end.setup()

From **Code 5** we can determine that the human readable graph is called “heat_demo_graph” and from the line 22, as shown in **Code 6**, the model binaries folder is where the binaries for the spinnaker_graph_front_end are stored. This does not need to be the case, but is more a convention for easy access.

```
from examples import model_binaries
```

Code 6: How to import the location of the binaries

On line 31, as shown in **Code 7**, the script asks the front end for the dimensions of the machine, as shown below:

```
dimensions = front_end.get_machine_dimensions()
```

Code 7: How to get the dimensions of the SpiNNaker machine

Note that even though the script uses this to deduce the size of the application space it's going to construct, its not essentially required, on lines 47 and 48, shown in **Code 8**, where there are two commented lines to override the dimensions of the application space.

```
47.    max_x_element_id = 2

48.    max_y_element_id = 2
```

Code 8: Overriding the SpiNNaker machine dimensions for application graph generation.

Lines 52 to 62, as shown in **Code 9**, build a utility model used by scripts that want to do live output of data from SpiNNaker machine to host efficiently. The *LivePacketGatherPartitionedVertex* receives messages in multicast form (both with and without payload) and every timer tick (note that these are programmable from the parameters '*machine_time_step*' and *'timescale_factor*') and combines whatever messages, it has received within that period, into as few SDP packets as possible and forwards them over to the host. This is more efficient that each core sending its own SDP message due to the overhead of duplicate SDP headers going over the ethernet socket (which has rather limited bandwidth). The format used by the *LivePacketGatherPartitionedVertex* is a EIEIO packet, as defined within the EIEIO lab manual. This protocol is the reason for the parameters "message_type".

```
live_gatherer = front_end.add_partitioned_vertex(
        LivePacketGatherPartitionedVertex,
                                {'machine_time_step': machine_time_step,
                            'timescale_factor': time_scale_factor,
                                'label': "gatherer from heat elements",
                                'ip_address': machine_host,
                                'port': machine_receive_port,
                                'message_type': EIEIOType.KEY_32_BIT})
```

Code 9: Instantiating a Live Packet Gather Partitioned Vertex.

Lines 65 to 75, as shown in **Code 10**, instantiate a collection of computational node, each of which represent an element in a conductive material, and therefore requires computation to deduce its next temperature at given periods. Each computational node, now referred to as vertices, is represented by a HeatDemoVertexPartitioned which will be explained in Section SpiNNaker Graph Front End: Computational Node Interface.

During the instantiation of the application graph the **heat_demo** uses the call add_partitioned_vertex(), instead of the call add_partitionable_vertex(). This informs the GFE that this application graph will contain vertices where every vertex resides on one core. The **LivePacketGatherPartitionedVertex** defined in Lines 52 to 62 was also added with a call to add_partitioned_vertex(); the **LivePacketGatherPartitionedVertex** and the **LivePacketGather** represent the same functionality, the only difference being that a **LivePacketGather** can only be inserted into a *partitionable graph* whereas the **LivePacketGatherPartitionedVertex** can only be placed within a *partitioned graph*.

Unfortunately, the GFE does not support the mixing of both a *partitionable graph* and a *partitioned graph* and therefore if the utility model you want to use is not of the same type, it will require you to find the version that corresponds to your application graph.

```
                                    for x_position in range(0, max_x_element_id):
                                        for y_position in range(0, max_y_element_id):
                                            element = front_end.add_partitioned_vertex(
HeatDemoVertexPartitioned,
                                                {'machine_time_step': machine_time_step,
                                                 'time_scale_factor': time_scale_factor},
                                                label="heat_element at coords {}:{}".format(
                                                    x_position, y_position))
```

Code 10: Instantiating a collection of HeatDemoVertexPartitioned

Lines 78 to 136 build the communication links between the set of different **HeatDemoVertexPartitioned** vertices. **Code 11** shows a subset of this code as an example. In this application, each heat element of the conductive material can transfer heat to its nearest neighbours in the directions of North, South, East, West. Therefore these links need to be defined here. Because each Edge has a given direction, which the

HeatDemoVertexPartitioned uses during its loading process in python, a new Edge Type has been created, which is called HeatDemoEdge. This will be discussed in more detail in Section SpiNNaker Graph Front End: Edges Interface.

```
front_end.add_partitioned_edge(
        MultiCastPartitionedEdge,
        {'pre_subvertex': vertices[x_position][y_position],
         'post_subvertex': live_gatherer},
         label="gatherer edge from vertex {} to live packet gatherer"
                .format(vertices[x_position][y_position].label),
         partition_id="TRANSMISSION")
```

Code 11: Example code for creating an edge between computational nodes.

In this case the **HeatDemoVertexPartitioned** uses the **HeatDemoEdge** to help when transferring to the executable code that will run on a SpiNNaker machine (in this case, defined in heat_demo.c) what keys it's expecting to receive from its different directional neighbours.

Take close note of the partition id used when defining the Edges, these have direct interaction with the PACMAN routing key generation, as all edges within the same partition share the same key set, and different partitions have different keys.

Line 148, as shown in **Code 12**, informs the GFE that the application graph has finished being constructed and needs to be mapped, loaded, and ran on a SpINNaker machine for 10 microseconds.

```
front_end.run(10)
```

Code 12: Code to map, load, execute, the application graph on a SpiNNaker machine.

Line 149, as shown in **Code 13**, informs the GFE that the model has finished being ran and it should clean the SpiNNaker machine of all data structures relating to this model, so that a new application can run cleanly on the SpiNNaker machine.

```
front_end.stop()
```

Code 13: call to clear the SpiNNaker machine for the next application.

SpiNNaker Graph Front End: Computational Node Interface

This section describes the different interfaces available for end users to use to get the most out of the tool chain.

The methodology of the SpiNNaker tool chain as a whole is to use classes and inherit/extend as needed to represent functionality. This means that to get access to certain parts of the SpiNNaker tool chain, e.g., the ability to buffer data in and out of the SpiNNaker machine during the execution of the simulation, requires the Computational nodes to inherit from specific classes. **Table 2** provides a list of the classes that a end user's vertex could helpfully inherit from and the functionality they represent, and **Table 3** provides the import lines for each interface.

Class	Description
Graph Focused Interfaces	
PartitionedVertex	This informs the GFE that this vertex does not need to be partitioned, as it already requires only one core.
AbstractPartitionableVertex	This informs the GFE that this vertex needs to be partitioned into a collection of smaller vertices which can reside on one core each. This causes the GFE to engage the PACMAN partitioning algorithm.
Data transmission Interfaces	
AbstractPartitionedDataSpecableVertex	This tells the GFE that this partitioned vertex generates data in the Data specification language defined in the Data Specification data sheet.
AbstractDataSpecableVertex	This tells the GFE that this partitionable vertex generates data in the Data specification language defined in the Data Specification data sheet.
Multi run interfaces	
AbstractChangeableAfterRun	This allows the GFE to deduce between calls to run() if the application graph has changed, and therefore avoid the mapping process if it has not been changed.
AbstractHasFirstMachineTimeStep	This informs the GFE that this vertex needs to be informed of the next machine time step to be ran on the SpiNNaker machine. This is usually 0, but during multiple calls to run, this will be updated. Usually this is

	used in conjunction with the **SendsBuffersFromHostPartitionedVertexPreBufferedImpl** interface.
Buffer management	
AbstractReceiveBuffersToHost	This defines the interface for informing the GFE and buffer manager that this vertex, when executing on the SpiNNaker machine, will expect buffers to be extracted in real time.
ReceiveBuffersToHostBasicImpl	This implements the **AbstractReceiveBuffersToHost** defined above, and is the recommended class to inherit from.
AbstractSendsBuffersFromHost	This defines the interface for informing the GFE and buffer manager that this vertex, when executing on the SpiNNaker machine, will be forwarding buffers from host to the c code.
SendsBuffersFromHostPreBufferedImpl	This implements the **AbstractSendsBuffersFromHostPartitionedVertex** defined above, and is the recommended class to inherit from.
Mapping support interfaces	
AbstractProvidesNKeysForPartition	This informs the GFE that when executing key allocation, the vertex already knows how many keys it requires per outgoing partition, or that this number if different to the number of atoms that this vertex represents.
Constraint based interfaces	
AbstractProvidesOutgoingPartitionConstraints	This informs the GFE and PACMAN algorithms that during routing / routing key generation, that there are constraints that need to be considered for the edges coming out of this vertex. The list of available constraints can be found in **Table 5**.
AbstractProvidesIncomingPartitionConstraints	This informs the GFE and PACMAN algorithms that during routing / routing key generation, that there are constraints that need to be considered for the edges coming into this vertex. The list of available constraints can be found in **Table 5**.
Provenance based interfaces	
AbstractProvidesProvenanceFromMachine	This informs the GFE that during provenance extraction, that this partitioned vertex contains its own provenance data.
ProvidesProvenanceFromMachineImpl	The implementation of the AbstractProvidesProvenanceFromMachine that most end users will want to use.

Table 2: The different types of interfaces provided by the SpiNNaker tools for a vertex

Class	Import line
Graph Focused Interfaces	
PartitionedVertex	**from** pacman.model.partitionable_graph.abstract_partitionable_vertex **import** AbstractPartitionableVertex
AbstractPartitionableVertex	**from** pacman.model.partitioned_graph.partitioned_vertex **import** PartitionedVertex
Data transmission Interfaces	
AbstractPartitionedDataSpecableVertex	**from** spinn_front_end_common.abstract_models.abstract_partitioned_data_specable_vertex **import** AbstractPartitionedDataSpecableVertex
AbstractDataSpecableVertex	**from** spinn_front_end_common.abstract_models.abstract_data_specable_vertex **import** AbstractDataSpecableVertex
Multi run interfaces	
AbstractChangeableAfterRun	**from** spinn_front_end_common.abstract_models.abstract_changable_after_run **import** AbstractChangableAfterRun
AbstractHasFirstMachineTimeStep	from spinn_front_end_common.abstract_models.abstract_has_first_machine_time_step import AbstractHasFirstMachineTimeStep
Buffer Management	
AbstractReceiveBuffersToHost	**from** spinn_front_end_common.interface.buffer_management.buffer_models.abstract_receive_buffers_to_host **import** AbstractReceiveBuffersToHost
ReceiveBuffersToHostBasicImpl	**from** spinn_front_end_common.interface.buffer_management.buffer_models.receives_buffers_to_host_basic_impl **import** ReceiveBuffersToHostBasicImpl
AbstractSendsBuffersFromHost	**from** spinn_front_end_common.interface.buffer_management.buffer_models.abstract_sends_buffers_from_host **import** AbstractSendsBuffersFromHost
SendsBuffersFromHostPreBufferedImpl	**from** spinn_front_end_common.interface.buffer_management.buffer_models.sends_buffers_from_host_pre_buffered_impl **import** SendsBuffersFromHostPreBufferedImpl

Mapping support interfaces	
AbstractProvidesNKeysForPartition	**from** spinn_front_end_common.abstract_models.abstract_provides_n_keys_for_partition **import** AbstractProvidesNKeysForPartition
Constraint based interfaces	
AbstractProvidesOutgoingPartitionConstraints	**from** spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints **import** AbstractProvidesOutgoingPartitionConstraints
AbstractProvidesIncomingPartitionConstraints	**from** spinn_front_end_common.abstract_models.abstract_provides_incoming_partition_constraints **import** AbstractProvidesIncomingPartitionConstraints
Provenance based interfaces	
AbstractProvidesProvenanceFromMachine	**from** spinn_front_end_common.interface.provenance.abstract_provides_provenance_data_from_machine **import** AbstractProvidesProvenanceDataFromMachine
ProvidesProvenanceFromMachineImpl	**from** spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl **import** ProvidesProvenanceDataFromMachineImpl

Table 3: The import lines for the different interfaces provided by the SpiNNaker tool chain.

Note that for all future interfaces, the __init__ function is defined as forced to be implemented function, but this is only due to python objects should calls its parent constructor as good practice.

Graph Interfaces

At this point, it is worth considering what type of computational node you are going to require. As discussed previously in Section **SpiNNaker Front End: Python Script Interface** there are two types of graph represented by the GFE, referred to as *partitionable* and *partitioned graph*.

The decision of which type of graph your computational node resides needs to be decided up front, as the number of python classes the end user needs to implement and what interfaces these require is decided here. A *partitioned vertex* only needs one python class to be implemented (and this is how the Heat Demo operates), if we look in the **heat_demo.py** on lines 53 and 68, the script adds a partitioned_vertex() and if we look at the **HeatDemoVertexPartitioned** on line 39 the class is stated to be inheriting from **PartitionedVertex**. This informs the GFE and PACMAN that this vertex can only be placed in a *partitioned graph* and can only reside on one core and therefore the PACMAN partitioning algorithm is not needed.

Each interface usually requires specific functions to be implemented for it to work correctly. Below is a breakdown of each interface and what methods/parameters are provided/needed.

AbstractPartitionableVertex

The AbstractPartitionableVertex interface is one of two mutually exclusive interfaces that is needed by end users to represent their application graphs. By being *partitionable*, this type of vertex has to contain a number of 'atoms' where an atom is the lowest atomic bit of computation that this application can represent. Because the contains multiple atoms, it is possible that the resources required by all the atoms this partitionable vertex represents is greater than what is provided by a SpiNNaker core. A SpiNNaker core contains 32 kb of instruction memory, 64 kb of ram memory, referred to as DTCM, and each chip has 120 Mb of hard disk memory, referred to as SDRAM, which has to be shared between the 16/17 cores on the chip.

Because it is more likely that a partitionable vertex is going to require more resources than what a core provides, or more SDRAM than is approx equal for all cores (8 Mb), then this partitionable vertex is going to need to be broken down into chunks which do fit on one core. To do this, the partitionable vertex needs to talk to the PACMAN Partitioner algorithms that decide upon this partitioning. **Figure 2** shows how a partitionable vertex can be split into multiple cores. To support the ease of developing both vertices and partitioning algorithms the

AbstractPartitionableVertex provides a collection of abstract functions which any vertex needs to implement. Each one defines how much of a given resource each atom requires. These functions are defined in Table 4. Other functions which can be overloaded are defined in Table 5 and parameters provided are defined in Table 6.

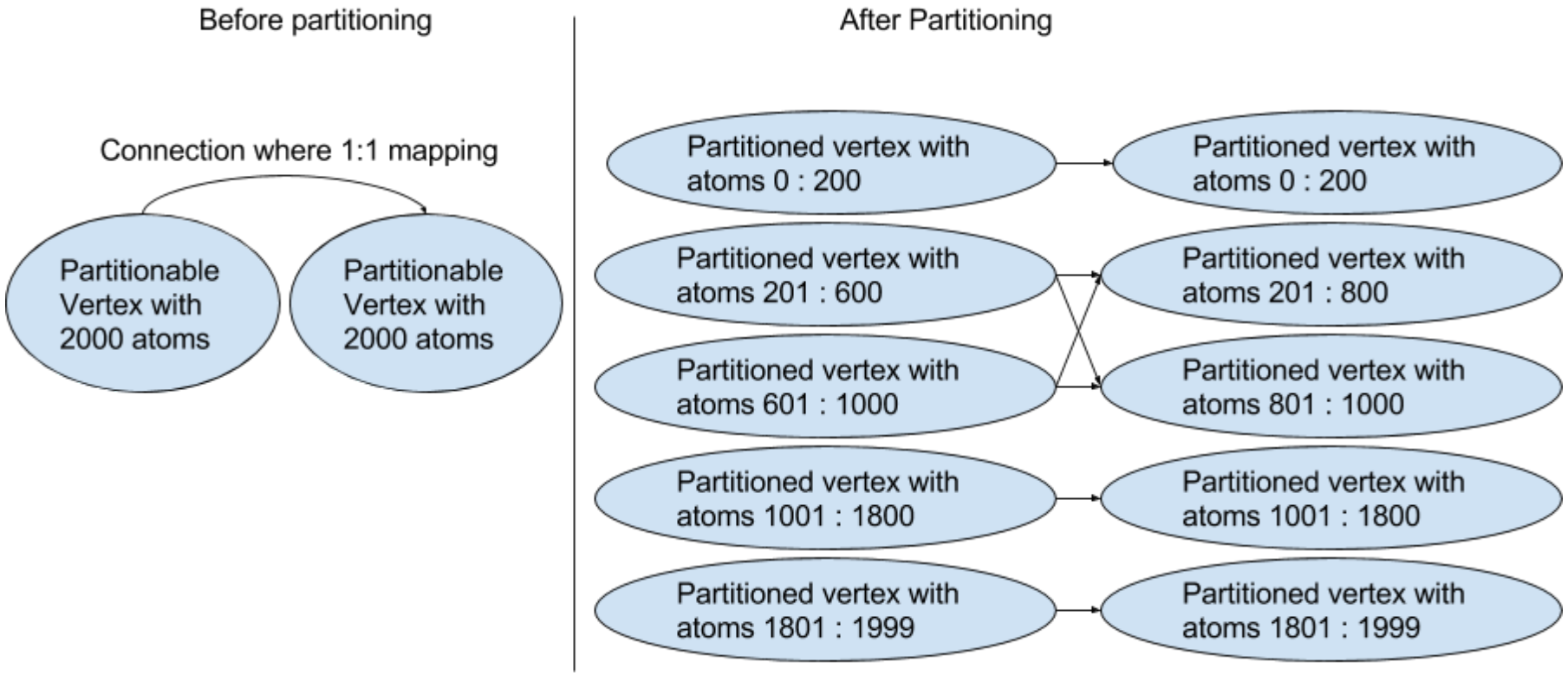

Figure 2: The behaviour of a possible partitioning where atoms of a vertex have different resource requirements.

Method call	Inputs	Outputs	Description
__init__()	**n_atoms:** (int) The number of atoms this vertex represents. **label:** (str) a human readable label for this vertex **max_atoms_per_core:** (int) the max size of a chunk of atoms for this vertex **constraints:** (iterable of pacman.model.constraints.abstract_constraints.abstract_constraint.AbstractConstraint) list of constraints to put on this vertex.	N/A	The constructor that all classes that inherit this class must call.
get_sdram_usage_for_atoms()	**vertex_slice**: (pacman.model.graph_mapper.slice.Slice) A selection of atoms with lo-atom, hi_atom. **partitionable_graph**: (pacman.model.partitionable_graph.partitionable_graph.PartitionableGraph) The partitionable graph instance.	int	A method that given a slice of atoms from the partitionable vertex, returns the amount of SDRAM that slice would require in bytes.
get_dtcm_usage_for_atoms()	**vertex_slice**: (pacman.model.graph_mapper.slice.Slice) A selection of atoms with lo-atom, hi_atom. **partitionable_graph**: (pacman.model.partitionable_graph.partitionable_graph.PartitionableGraph) The partitionable graph instance.	int	A method that given a slice of atoms from the partitionable vertex, returns the amount of DTCM that slice would require in bytes.
get_cpu_usage_for_atoms()	**vertex_slice**: (pacman.model.graph_mapper.slice.Slice) A selection of atoms with lo-atom, hi_atom. **partitionable_graph**: (pacman.model.partitionable_graph.partitionable_graph.PartitionableGraph) The partitionable graph instance.	int	A method that given a slice of atoms from the partitionable vertex, returns the amount of CPU that slice would require in bytes.
model_name()	**None**	str	A human readable form of this partitioning vertex.

Table 4: Methods expected from the AbstractPartitionableVertex

Method call	Inputs	Outputs	Description
get_resources_used_by_atoms()	**vertex_slice**: (pacman.model.graph_mapper.slice.Slice) A selection of atoms with lo-atom, hi_atom. **partitionable_graph**: (pacman.model.partitionable_graph.partitionable_graph.PartitionableGraph) The partitionable graph instance.	An instance of pacman.model.resoruces.resoruce_container.ResourceContainer	A function that provides a resource container which encapsulates the the outputs from: 1. get_cpu_usage_for_atoms() 2. get_dtcm_usage_for_atoms() 3. get_sdram_usage_for_atoms()
get_max_atoms_per_core()	None	int	searches through the PartitionerMaximumSizeConstraint of the vertex to find the one with the smallest value.
create_subvertex()	**vertex_slice**: (pacman.model.graph_mapper.slice.Slice) the slice of atoms to consider **resources_required:** (pacman.model.resources.resource_container.ResourceContainer) The container for cpu, dtcm, sdram usage for this partitioned vertex. **label**: (str) a human readable label for this vertex **constraints**: (iterable of pacman.model.constraints.abstract_constraints.abstract_constraint.AbstractConstraint) list of constraints to put on this partitioned vertex.	An instance of pacman.model.partitioned_graph.PartitionedVertex	Function that created a partitioned_vertex for the partitionable vertex.
n_atoms()	None	int	Property method for finding the number of atoms this partitionable vertex represents.

Table 5: Methods provided by the AbstractPartitionableVertex

Parameter	Description
_n_atoms	how many atoms this partitionable vertex contains.
_label	Human readable form of this partitionable vertex.
_constraints	list of constraints that this partitionable vertex has.

Table 6: The variables provided by the abstract partitionable vertex interface.

An example of a Partitionable vertex would be the **LivePacketGatherer** where:
Line 60 calls the **AbstractPartitionableVertex** constructor.
Line 135 instantiates the **AbstractPartitionableVertex** *model_name()* method.
Line 111 instantiates the **AbstractPartitionableVertex** *get_cpu_usage_for_atoms()* method.
Line 115 instantiates the **AbstractPartitionableVertex** *get_sdram_usage_for_atoms()* method.
Line 119 instantiates the **AbstractPartitionableVertex** *get_dtcm_usage_for_atoms()* method.
Line 123 overrides the provided **AbstractPartitionableVertex** *create_subvertex()* method.

PartitionedVertex

The PartitionedVertex interface is the other mutually exclusive main interfaces that is needed by end users to represent their application graphs. By already being partitioned, the interface needed by the PACMAN partitioner algorithms are not needed, but the data that would have been passed down to the ***PartitionedVertex*** during partitioning needs to be given up front now so that the PACMAN placer can deduce where to place it on the SpiNNaker machine in the most efficient way.

Luckily for a partitioned vertex, there are no abstract methods that require the end user to implement, but there is one method which can be overloaded. **Table 7** shows what is required for the __init__() method, as well as the only available method to overload, and **Table 8** shows exposes the parameters provided by the ***PartitionedVertex*** class.

Method call	Inputs	Outputs	Description
__init__()	**resources_required**: (pacman.model.resources.rewsource_container.ResourceContainer) dtcm, sdram, cpu cycles used by this partitioned vertex **label**: (str) human readable form of this partitioned vertex **constraints**: (iterable of pacman.model.constraints.abstract_constraints.abstract_constraint.AbstractConstraint) list of constraints to put on this partitioned vertex.	None	Constructor for the partitioned vertex. Requires the resources for informing the PACMAN placer algorithms for efficient placements.
resources_required()	None	**resources_required**: (pacman.model.resources.rewsource_container.ResourceContainer) dtcm, sdram, cpu cycles used by this partitioned vertex	property method that returns a pacman.model.resources.rewsource_container.ResourceContainer that contains the SDRAM, DTCM and CPU cycles used by this partitioned vertex.

Table 7: partitioned vertex provided methods.

Parameter	Description
_resources_required	Container for SDRAM, DTCM and CPU cycles used by this partitioned vertex.
_label	Human readable form of this partitionable vertex.
_constraints	list of constraints that this partitionable vertex has.

Table 8: The variables provided by the partitioned vertex interface.

An example of a Partitioned vertex would also be the ***LivePacketGatherPartitionedVertex*** where on line 65 the ***PartitionedVertex*** constructor is called.

It is worth noting that for vertices that start off as partitioned vertices, for example the ***heat_demo_vertex,*** they still require to build a resource container with:

1. A CPU value (encapsulated within a *CPUCyclesPerTickResource* object) which represents how many CPU cycles this vertex needs per timer tick,
2. a DTCM value (encapsulated within a DTCMResource object) which represents how much DTCM this vertex requires overall,
3. and a SDRAM value (encapsulated within a SDRAMResource object) which represents how much SDRAM this vertex requires overall.
4. These 3 objects are encapsulated within a ResourceContainer object.

The lines to import these objects can be found in **Code 14**.

```
from pacman.model.resources.cpu_cycles_per_tick_resource import CPUCyclesPerTickResource
from pacman.model.resources.dtcm_resource import DTCMResource
from pacman.model.resources.resource_container import ResourceContainer
from pacman.model.resources.sdram_resource import SDRAMResource
```

Code14: Import lines for the different resource objects

Data Transmission Interfaces

The data transmission interfaces are designed to support compressing data from host for communicating it into the SpiNNaker machine. The current working interface's use the **Data Specification Language**, whose definitions can be found here.

If the end user does not wish to use the Data Specification Language, they are completely free to design their own interface and algorithms, but to be able to be tied into the toolchain, the end user will need to provide an algorithm which works within the mapping framework described in the **Adding new mapping algorithms to the SpiNNaker tool-chain tutorial,** which can be found here, and which produces the *LoadedApplicationData* token required by the FrontEndCommonApplciationRunner algortihm.

Note that this should only be done in extreme cases with the **Another Fine Product From The Nonsense Factory** release. This is because the c interfaces contained within the files *simulation.h, recording.h* and *data_specification.h* are currently designed to accept DSG regions, and therefore the end user will need to reengineer these. A future release may remove this requirement.

Assuming the end user intends to use the Data Specification Language, then the end user needs to know which graph vertex they are instantiating, as discussed in **Section** Graph Interfaces. Currently the two types of vertex are treated differently, due to the main method, which is required for vertices which inherit from, having different parameter requirements.

Note that In a future release there are plans to reduce the parameters required by this method to None, and expose a set of provider interfaces in some form to support easier operations, but to date this has not been implemented yet.

If the end user has created a partitionable vertex, then they need to inherit from *AbstractDataSpecableVertex* whereas if the end user has created a partitioned vertex, then they need to inherit from *AbstractPartitionedDataSpecableVertex*. The next two subsections describe each interface in detail.

AbstractPartitionedDataSpecableVertex

The *AbstractPartitionedDataSpecableVertex* supports the generation of application data for a *partitioned vertex*. Table 9 describes the methods and their parameters needed to be implemented by the end user. Table 10 describes the functions provided by the *AbstractPartitionedDataSpecableVertex* and Table 11 describes parameters provided by the *AbstractPartitionedDataSpecableVertex*.

Method call	Inputs	Outputs	Description
__init__()	**machine_time_step:** (int) The period between timer tick callbacks considered by the simulation code.. **timescale_factory:** (int) The real time multiplication effect for the period between timer tick for the SpiNNaker machine.	N/A	The constructor that all classes that inherit this class must call.
generate_data_spec()	**placement**: (pacman.model.placements.placement.Placement) the location of this partitioned vertex on the SpiNNaker machine. **partitioned_graph**: (pacman.model.partitioned_graph.partitioned_graph.PartitionedGraph) the partitioned graph that this partitioned vertex is apart of. **routing_info**: (pacman.model.routing_info.routing_info.RoutingInfo) the object that contains keys and masks for the edges of the partitioned graph. **hostname**: (str) the machine hostname / ip_address of the machine **report_folder**: (path) file path to where reports should be written, such as dsg report. **ip_tags**: (iterable of spinnMachine.tags.iptag.Iptag) collection of tags used by the partitioned graph. **reverse_ip_tags**: (iterable of spinnMachine.tags.reverse_iptag.ReverseIptag) collection of tags used by the partitioned graph. **write_text_specs**: (bool) flag which states to write the text based spec (a report) or not. **application_run_time_folder**: (path) the path for where the application data should be stored.	file_path	The main method that generates a filepath for where the Data Specification file has been written.
get_binary_file_name()	**None**	name of the binary	Property method for telling the tools what the c binary is called.
is_data_specable()	**None**	None	Needed by the tools to recognize that this vertex is an AbstractDataSpecableVertex.

Table 9: Methods expected from the AbstractPartitionedDataSpecableVertex

Method call	Inputs	Outputs	Description
_write_basic_setup_info()	**spec**:(data_specification.data_specification_generator.DataSpecificationGenerator) the specification generator used by the partitioned vertex to write its application data. **region_id**: (int) the region id defined for system data.	None	Adds generic data needed for the c code's simulation.c and data_specification.c interfaces to work correctly.
machine_time_step()	None	int	property method for returning the machine_time_step
no_machine_time_steps()	None	int	property method which returns the total number of machine time steps to run currently.
set_no_machine_time_steps()	**new_no_machine_time_steps**: the new number of machine time steps. This is handed over from the GFE interface once the runtime is given.	None	sets the total number of machine time steps to run currently
get_data_spec_file_writers()	**processor_chip_x**:(int) x coord the chip this partitioned vertex is placed. **processor_chip_y**:(int) y coord the chip this partitioned vertex is placed. **processor_id**:(int) the processor id that this partitioned vertex is placed. **hostname**: (str) the ip_address of the SpiNNaker machine. **report_directory**: (file_path) the file path for where reports should be stored. **write_text_specs**:(bool) flag which states to write the text based spec (a report) or not. **application_run_time_folder**: (path) the path for where the application data should be stored.	data writer, report_writer	generates writers for both the data spec and the report writer(if required)
get_data_spec_file_path()	**processor_chip_x**:(int) x coord the chip this partitioned vertex is placed. **processor_chip_y**:(int) y coord the chip this partitioned vertex is placed. **processor_id**:(int) the processor id that this partitioned vertex is placed. **hostname**: (str) the ip_address of the SpiNNaker machine. **application_run_time_folder**: (path) the path for where the application data should be stored.	file_path	returns the file path needed for the data spec writer (report writer)
get_application_data_file_path()	**processor_chip_x**:(int) x coord the chip this partitioned vertex is placed. **processor_chip_y**:(int) y coord the chip this partitioned vertex is placed. **processor_id**:(int) the processor id that this partitioned vertex is placed.	file_path	returns the file path for the data spec writer (application data writer).

	hostname: (str) the ip_address of the SpiNNaker machine. **application_run_time_folder**: (path) the path for where the application data should be stored.		

Table 10: Methods provided from the AbstractPartitionedDataSpecableVertex

Parameter	Description
_machine_time_step	The period between timer tick callbacks considered by the simulation code
_timescale_factor	The real time multiplication effect for the period between timer tick for the SpiNNaker machine.
_no_machine_time_steps	the total number of machine time steps. This is handed over from the GFE interface once the runtime is given.

Table 11: The variables provided by the AbstractPartitionedDataSpecableVertex.

The *generate_data_spec* method is the entrance method for generating all the data your application requires for this given partitioned vertex. If we look back at the **heat_demo_vertex.py** where**:**

1. On line 112 the *geneate_data_spec* is implemented.
2. On line 98 the *get_binary_file_name* is implemented which links the python vertex to the compiled c code aplx.
3. On line 375 the *is_partitioned_data_specable* is implemented.

We do not plan to explore the definition of a DSG file here, and forward you to the DataSpecification tutorial here.

AbstractDataSpecableVertex

The *AbstractDataSpecableVertex* supports the generation of application data for a *partitionable vertex*, because the application data is core based, the *generate_data_spec()* method includes the *partitionable_graph, graph_mapper*, and the *partitioned_vertex* to create the data for. It is recommended to push this code down into your *partitioned_vertex*, but is not required. **Table 12** describes the methods and their parameters needed to be implemented by the end user. **Table 13** describes the functions provided by the *AbstractDataSpecableVertex* and **Table 14** describes parameters provided by the *AbstractDataSpecableVertex*.

Method call	Inputs	Outputs	Description
__init__()	**machine_time_step:** (int) The period between timer tick callbacks considered by the simulation code.. **timescale_factory:** (int) The real time multiplication effect for the period between timer tick for the SpiNNaker machine.	N/A	The constructor that all classes that inherit this class must call.
generate_data_spec()	**partitioned_vertex**: (pacman.model.partitioned_graph.partitioned_vertex.parttiioned_vertex.PartitionedVertex) the partitioned vertex to generate the data for. **placement**: (pacman.model.placements.placement.Placement) the location of this partitioned vertex on the SpiNNaker machine. **partitioned_graph**: (pacman.model.partitioned_graph.partitioned_graph.PartitionedGraph) the partitioned graph that this partitioned vertex is apart of. **partitionable_graph**: (pacman.model.partitionable_graph.partitionable_graph.PartitionableGraph) the partitionable graph that this partitioned vertex's partitionable vertex is apart of. **routing_info**: (pacman.model.routing_info.routing_info.RoutingInfo) the object that contains keys and masks for the edges of the partitioned graph. **hostname**: (str) the machine hostname / ip_address of the machine **graph_mapper**: (pacman.model.graph_mapper.graph_mapper.GraphMapper) the mapping between partitionable and partitioned graphs. **report_folder**: (path) file path to where reports should be written, such as dsg report. **ip_tags**: (iterable of spinnMachine.tags.iptag.Iptag) collection of tags used by the partitioned graph. **reverse_ip_tags**: (iterable of spinnMachine.tags.reverse_iptag.ReverseIptag) collection of tags used by the partitioned graph.	file_path	The main method that generates a filepath for where the Data Specification file has been written.

	write_text_specs: (bool) flag which states to write the text based spec (a report) or not. **application_run_time_folder**: (path) the path for where the application data should be stored.		
get_binary_file_name()	**None**	name of the binary	Property method for telling the tools what the c binary is called.
is_data_specable()	**None**	None	Needed by the tools to recognize that this vertex is an AbstractDataSpecableVertex.

Table 12: Methods expected from the AbstractDataSpecableVertex

Method call	Inputs	Outputs	Description
_write_basic_setup_info()	**spec**:(data_specification.data_specification_generator.DataSpecificationGenerator) the specification generator used by the partitioned vertex to write its application data. **region_id**: (int) the region id defined for system data.	None	Adds generic data needed for the c code's simulation.c and data_specification.c interfaces to work correctly.
machine_time_step()	None	int	property method for returning the machine_time_step
no_machine_time_steps()	None	int	property method which returns the total number of machine time steps to run currently.
set_no_machine_time_steps()	**new_no_machine_time_steps**: the new number of machine time steps. This is handed over from the GFE interface once the runtime is given.	None	sets the total number of machine time steps to run currently
get_data_spec_file_writers()	**processor_chip_x**:(int) x coord the chip this partitioned vertex is placed. **processor_chip_y**:(int) y coord the chip this partitioned vertex is placed. **processor_id**:(int) the processor id that this partitioned vertex is placed. **hostname**: (str) the ip_address of the SpiNNaker machine. **report_directory**: (file_path) the file path for where reports should be stored. **write_text_specs**:(bool) flag which states to write the text based spec (a report) or not. **application_run_time_folder**: (path) the path for where the application data should be stored.	data writer, report_writer	generates writers for both the data spec and the report writer(if required)
get_data_spec_file_path()	**processor_chip_x**:(int) x coord the chip this partitioned vertex is placed. **processor_chip_y**:(int) y coord the chip this partitioned vertex is placed. **processor_id**:(int) the processor id that this partitioned vertex is placed. **hostname**: (str) the ip_address of the SpiNNaker machine. **application_run_time_folder**: (path) the path for where the application data should be stored.	file_path	returns the file path needed for the data spec writer (report writer)
get_application_data_file_path()	**processor_chip_x**:(int) x coord the chip this partitioned vertex is placed. **processor_chip_y**:(int) y coord the chip this partitioned vertex is placed. **processor_id**:(int) the processor id that this partitioned vertex is placed. **hostname**: (str) the ip_address of the SpiNNaker machine. **application_run_time_folder**: (path) the path for where the application data	file_path	returns the file path for the data spec writer (application data writer).

	should be stored.		

Table 13: Methods provided from the AbstracDataSpecableVertex

Parameter	Description
_machine_time_step	The period between timer tick callbacks considered by the simulation code
_timescale_factor	The real time multiplication effect for the period between timer tick for the SpiNNaker machine.
_no_machine_time_steps	the total number of machine time steps. This is handed over from the GFE interface once the runtime is given.

Table 14: The variables provided by the AbstractDataSpecableVertex.

The *generate_data_spec* method is the entrance method for generating all the data your application requires for this given *partitionable vertex*. If we look back at the **LivePacketGather.py** where**:**

1. Line 100 the *geneate_data_spec* is implemented.
2. On line 140 the *get_binary_file_name* is implemented which links this python vertex to the compiled c code aplx.
3. On line 143 the *is_data_specable* is implemented.

Multi Run Interfaces

The main interface provided by the GFE shares a lot of functionality from the PyNN front end (sPyNNaker) and therefore functionality that maps between the two is provided for no cost. One of these functionalities is the ability to support the ability to call run() multiple times. To support this efficiently, the graph's need to be able to recognise that there have been changes to their vertices / edges between two calls to run. This is supported by the *AbstractChangeableAfterRun*. For vertices which contain data which is time stamped (either for injection or retrieval), when multiple calls to run are executed, they need to be informed of what the current timestep they are on, and the *AbstractHasFirstMachineTimeStep* provides that interface.

Note that chunking your data to be loaded into your binary, running on the SpiNNaker machine, per run cycle has a direct interaction with the functionality that is designed to remove the SDRAM limitation problem for recording state etc on the SpiNNaker chips. This functionality is called auto_pause_and_resume and can be read in more detail **here**.

AbstractChangeableAfterRun

The *AbstractChangeableAfterRun* requires any object that inherits from it to implement two methods, as described in **Table 15**. The interface does not provide any extra functions or parameters to objects that inherit from it.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
requires_mapping()	None	bool	Asks the object if since the last time mark_no_changes was called, if there's been changes to themselves which will require to go through the mapping process again.
mark_no_changes()	None	None	Informs the object that from now on, record new changes and ignore old changes.

Table 15: methods needed to be implemented by objects inheriting from the AbstractMappableInterface.

AbstractHasFirstMachineTimeStep

The *AbstractHasFirstMachineTimeStep* requires objects that inherit from it to implement one method which the GFE will call during the call to run with the new starting point of where the time is going to run for. **Table 16** describes this function and the interface does not provide any extra functionality nor parameters.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
set_first_machine_time_step()	first_machine_time_step	None	The GFE will call this function during run with the next initial timer tick value. This is often used in conjunction with multi-run functionality.

Table 16: methods needed to be implemented by objects inheriting from the AbstractHasFirstMachineTimeStep

Neither of these interfaces are currently used within the HeatDemo or the GFE at large. If we look at the sPyNNaker front end, we can see that the *SpikeSourceArray* uses the *AbstractHasFirstMachineTimeStep* interface on Line 243 and uses this data via the *ReverseIpTagMultiCastSourcePartitionedVertex* on lines 246 to 259 to deduce what spikes it needs to stream into the SpiNNaker machine for this run period. The *AbstractChangeableAfterRun* interface is also used by the *SpikeSourceArray* on lines 155 to 160, it uses its set methods to determine if the parameter's changed requires a remapping process to be engaged.

Buffer Management

As mentioned previously in Section **Graph Interfaces**, a SpiNNaker chip contains 120 Mg of SDRAM which is used for both application setup data (used by the *AbstractDataSpecableVertex* and *AbstractPartitionedDataSpecableVertex*) and storing recorded state. Because this space is shared between 16 to 17 cores on the SpiNNaker chip, it often reduces down to approximately 8mg per core.

If we use the sPyNNaker front end interface for an working example, a core can record 3 states (spikes, v, gsyn). Let's assume that each partitioned vertex in a sPyNNaker application contains 256 atoms (which in sPyNNaker represent neurons). Now let's assume that every timer tick, all the neurons fire, and that each neuron uses 4 bytes for gsyn, 4 bytes for v and 4 bytes to record it spikes and let's assume that it requires no configuration data (a invalid assumption, but will simplify the calculations). This means for every neuron it requires 12 bytes to record, and therefore 3072 bytes per timer tick to record its state. This means that the core can run for 2730 timer ticks before running out of memory to record its state. If the simulation runs at 1ms time steps, this means you can run the simulation for a total of 2.7 seconds before running out of memory on a chip. This obviously is a problem and therefore to allow simulations to run for longer periods, 3 separate pieces of functionality have been devised, which in summary are:

1. Injection of data into a SpiNNaker machine during runtime in small chunks, referred to as input buffers, to reduce playback SDRAM footprint.
2. Live extraction of data from the SDRAM of SpiNNaker chips in chunks, referred to as output buffers, to reduce recording SDRAM footprint.
3. Auto pause and resume functionality which calculates how long the simulation can run before filling the SDRAM of a chip, and uses this as a basis to create multiple runs where in between these runs the recorded data is extracted and amalgamated with whatever data was actively extracted through 2.

Functions 1 and 2 use the next 2 interfaces and their implementations to support the interaction of vertices and these buffers and is described in greater detail here. Function 3 builds on top of the functionality which functions 1 and 2 use, but only for convenience. Function 3 is mainly implemented through the use of 2 cfg parameters that vertices are expected to read to realize if they are in deed in auto_puase_and_resume mode. These parameters are shown in **Code 15**.

```
[buffers] use_auto_pause_and_resume
[buffers] minimum_buffer_sdram
```

Code 15: cfg Parameters needed for detect auto_pause_and_resume

If the system is in the Function 3. The Vertex is expected to adjust its sdram usage to just be the static SDRAM usage needed to configure the vertex, and include the minimum_buffer_sdram to the value to stop the partitioner/placer from partitioning in such a way as to leave 0 SDRAM for recorded data.

Function 1 and Function 3 is covered through the *AbstractReceiveBuffersToHost* interface, whereas function 2 is covered through the *AbstractSendsBuffersFromHost* interface.

It is worth noting that the live injection interface is generalised in python to allow the injection of any type of data, but the there is no clean c interface to date, and therefore it is not recommended to extend the *AbstractSendsBuffersFromHost* vertex unless you're prepared to reimplement the c code of the the ReverseIptagMultiCastSource.c into your own system. Only for completeness do we discuss the python interface for live injection in Section **SendsBuffersFromHostPreBufferedImpl.**

AbstractReceiveBuffersToHost

The *AbstractReceiveBuffersToHost* interface defines functions used by the Buffer Manager (which can be read about in **The SpiNNaker software Stack Buffer Functionality** for recording retrieval and *auto_pause_and_resume*. These ask for if the object is using buffered output, the size that this vertex expects to have (given if its using *auto_pause_and_resume* or not), and how many buffers can be held in the predefined space for the buffers. These functions are expected to be implemented by any class that inherits from the *AbstractReceiveBuffersToHost* class and are described in **Table 17**. It does not provide any extra methods, nor any variables. It is worth noting, that we provide an implementation version of this interface, which we recommend you extend from instead, this is discussed in **Section ReceiveBuffersToHostBasicImpl.**

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
buffering_output()	None	Bool	Returns True if the buffering output has been engaged. False otherwise.
get_pre_reserved_buffer_sdram_size()	None	int	Returns a minimum size the partitioned vertex expects to have no matter what. This is mainly used in auto_pause_and_resume to ensure that there is some space allocated for recording during partitioning.
get_n_timesteps_in_buffer_space()	buffer_space	int	Determines how many timer ticks can be executed before this buffer space is filled in. floored if it's a fractional value.

Table 17: Methods that need to be implemented by objects that inherit from AbstractRedieveBuffersToHost.

ReceiveBuffersToHostBasicImpl

The *ReceiveBuffersToHostBasicImpl* class implements the *AbstractReceiveBuffersToHost* interface as much as possible, but still exposes the get_pre_reserved_buffer_sdram_size() function to the end user, as this cannot be decided up front. The cfg file provides a helpful default value under the field [Buffers] minimum_buffer_sdram, which can be used if the end user does not know what to use here. **Table 18** shows the methods that need to be implemented by the end user and **Table 19** shows the extra method provided by the implementation. **Table 20** shows the parameters provided by the interface.

Unfortunately, there is quite a lot of constraints in using the *ReceiveBuffersToHostBasicImpl* interface currently. This may be fixed in a future release, but to date any user of the *ReceiveBuffersToHostBasicImpl* interface needs to do the following:

1. Allocate an extra region that the BufferManager (a block of code used to manage both injection and retrieval of buffers) will use for storing state.
2. During a vertex's generate_data_spec, when the vertex reserves memory regions (as done so by the *spec.reserve_memory_region()* function) the vertex is not to reserve the regions which are expected to be managed by the BufferManager and instead, you should call the *reserve_buffer_regions()* with the correct inputs as defined in **Table 19**.
3. After reserving the vertex's data regions the vertex must call *write_recording_data()* with the correct inputs as defined in **Table 19.** This code writes data to wherever the spec is currently pointing, so be aware of this, as the recording.h interface requires you to give it the memory address where this data block is written during the recording initialise function discussed in **Section Buffered recording interface**.

If the vertex does not do this, the buffered regions will not be extracted properly and the errors produced by the tool chain in this functionality are rather vague.

Note that the *recording.h* interface uses this behind the scenes, and therefore if you plan to use the recording interface in c, your python object MUST inherit from this class, and execute the correct procedure defined above.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
add_constraint	**constraint**	None	Method to add a constraint to the list of constraints applied to this object. Assuming that your object is a vertex, it gets this method by default, as all vertices are constrained and therefore get this method from AbstractConstrainedVertex

Table 18: Methods that need to be implemented by objects that inherit from the ReceiveBuffersToHostBasicImpl interface.

Method call	Inputs	Outputs	Description
buffering_output()	None	Bool	Returns True if the buffering output has been engaged. False otherwise. Is a implementation of the AbstractReceiveBuffersToHost interface function.
activate_buffering_output()	**buffering_ip_address**:(str) ip_address of the board where the buffering is going through **buffering_port**:(int) the port number of the socket for where the buffering is going to. **board_address**:(str) the board address is the ip address of the board within the SpiNNaker machine. **notification_tag**:(int) the tag id used for sending notifications (used during the buffering interface). **pre_reserved_buffer_sdram_size**:(int) amount of memory in bytes, that the object expects to use no matter what. **buffered_sdram_per_timestep**:(int) number of bytes this object uses during each timer tick	None	Sets up state and adds buffering constraints to the object. Configurable through the parameters.
get_buffer_state_region_size()	**n_buffered_regions**: (int) the number of DSG regions that are under buffered control for this object.	int	Gets the amount of memory, in bytes, used by the state needed by the the buffer code for the number of buffered regions.

get_recording_data_size()	**n_buffered_regions**: (int) the number of DSG regions that are under buffered control for this object.	int	Gets the amount of memory, in bytes, used by the recording functionality for its own state.
reserve_buffer_regions()	**spec**: (data_specification.data_specification_generator.DataSpecificationGenerator) the specification generator used by the partitioned vertex to write its application data. **state_region**: (int) the DSG region which the buffer manager can use to contain its state data. **buffer_regions**: (iterable of int) iterable of dsg regions which are all buffered. **region_sizes**: (iterable on int) iterable of dsg region sizes.	None	allocates the buffer regions to the data specification generator with adjusted sizes to deal with recording initial states etc. buffer_regions and region_sizes are mapped 1 to 1 for each other.
get_tag()	**tags**: (pacman.model.tags.tags.Tags) the collection of ip and reverseip tags generated by PACMAN.	tag	The IPTAG used for buffering.
write_recording_data()	**spec**:(data_specification.data_specification_generator.DataSpecificationGenerator) the specification generator used by the partitioned vertex to write its application data. **ip_tags**: (iterable of spinnMachine.tags.iptag.Iptag) the iptags assigned to this object by the PACMAN algorithm. **region_sizes**:(iterable of int) An ordered list of the sizes of the regions in which buffered recording will take place **buffer_size_before_receive**:(int) The amount of data that can be stored in the buffer before a message is sent requesting the data be read **time_between_requests**: (int) The amount of time between requests for more data	None	Writes configuration data used by the buffer interface during execution.
get_pre_reserved_buffer_sdram_size()	None	int	returns the amount of memory, in bytes, that this objec t expects no matter what.
get_n_timesteps_in_buffer_space()	**buffer_space**: (int) the amount of space in bytes	int	Determines how many timer ticks can be executed before

	allocated to the buffer.		this buffer space is filled in. floored if it's a fractional value. Is a implementation of the AbstractReceiveBuffersToHost interface function. If _buffered_sdram_per_timestep is not constant, then this method needs to be overloaded.

Table 19: functions provided by the ReceiveBuffersToHostBasicImpl interface.

Parameter	Description
_buffering_output	Bool that states if its doing buffering.
_buffering_ip_address	The ip address of the machine its buffering to.
_buffering_port	The port from the socket used for doing buffering.
_pre_reserved_buffer_sdram_size	The amount of SDRAM that the vertex expects to be allocated no matter what for recording state.
_buffered_sdram_per_timestep	The amount of sdram used per time tick(used by the auto_pause_and_resume functionality). If this is not constant, then this will need to be overloaded.

Table 20: parameters provided by the ReceiveBuffersToHostBasicImpl interface.

An example of a model that uses the ReceiveBuffersToHostBasicImpl interface in anger can be found in the SpyNNaker front end under the is the **PopulationPartitionedVertex** and the **AbstractPopulationVertex** where:

1. Line 21 from **PopulationPartitionedVertex** inherits from the ReceiveBuffersToHostBasicImpl interface.
2. Line 38 from **PopulationPartitionedVertex** initiates the ReceiveBuffersToHostBasicImpl interface.
3. If you look at the spynnaker **constants** file you will see a enum from line 48 to 61 which defines the data specification regions for a spynnaker population vertex. Line 59 is the inclusion of the Buffer Manager's state region.
4. Line 344 from **AbstractPopulationVertex** calls the **PopulationPartitionedVertex's** *reserve_buffer_regions()* function. Note that the functions used by the *generate_data_spec()* between lines 423 and 510 could have been pushed down to the **PopulationPartitionedVertex** but has not yet been done so.
5. Line 366 from the **AbstractPopulationVertex** calls the *write_recoridng_data*() function so that the data is placed within the SYSTEM data specification region after the standard *basic_setup_info* data is stored.

AbstractSendsBuffersFromHost

The *AbstractSendsBuffersFromHost* interface defines functions used by the Buffer Manager (which can be read about in **The SpiNNaker software Stack Buffer Functionality** for injection purposes. These ask for if the object is using buffered output, the size that this vertex expects to have no matter what, and how many buffers can be held in the predefined space for the buffers. These functions are expected to be implemented by any class that inherits from the *AbstractSendsBuffersFromHost* class and are described in **Table 21**. It does not provide any extra methods, nor any variables. It is worth noting, that we provide an implementation version of this interface, which we recommend you extend from instead, this is discussed in **Section SendsBuffersFromHostPreBufferedImpl.**

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
buffering_input()	None	bool	Returns True if the object has engaged buffering input.
get_regions()	None	iterable of int	returns the list of DSG region ids, which are all buffered input regions.
get_region_buffer_size()	**region**: (int) the DSG region id to find the buffer size of	int	returns the size of the buffer which is stored within the dsg region.
get_max_buffer_size_possible()	**region**: (int) the DSG region id to find the buffer size of	int	returns the max possible size of a buffered region
is_next_timestamp()	**region**: (int) the DSG region id to find the buffer size of	bool	Determine if there is another timestamp with data to be sent
get_next_timestamp()	**region**: (int) the DSG region id to find the buffer size of	int	The timestamp of the next available events
is_next_key()	**region**: (int) the DSG region id to find the buffer size of **timestamp**: (int) The timestamp to determine if there are	bool	True if there are more keys to send for the parameters, False otherwise

	more keys for		
get_next_key()	**region**: (int) the DSG region id to find the buffer size of	int	Get the next key in the given region
is_empty()	**region**: (int) the DSG region id to find the buffer size of	bool	Return true if there are no spikes to be buffered for the specified region
rewind()	**region**: (int) the DSG region id to find the buffer size of	int	Rewinds the internal buffer in preparation of re-sending the spikes. Used during the GFE reset functionality.

Table 21: Methods that need to be implemented by objects that inherit from the AbstractSendsBuffersFromHost interface.

SendsBuffersFromHostPreBufferedImpl

The *SendsBuffersFromHostPreBufferedImpl* class implements the *AbstractSendsBuffersFromHost* interface. Table 22 shows the methods that are provided by the SendsBuffersFromHostPreBufferedImpl, including the __init__ function beyond what is supplied from the *AbstractSendsBuffersFromHost* interface. Table 23 shows the parameters provided by the interface.

Method call	Inputs	Outputs	Description
__init__()	**send_buffers**: dict(int -> :py:class:`spinnaker.pyNN.buffer_management.storage_objects.buffered_sending_region.BufferedSendingRegion`) A dictionary of the buffers of events to send, indexed by the regions	None	Constructor for this interface.
send_buffers()	None	send buffers	return the dict(int -> :py:class:`spinnaker.pyNN.buffer_management.storage_objects.buffered_sending_region.BufferedSendingRegion`) A dictionary of the buffers of events to send,

			indexed by the regions
send_buffers()	**value**: dict(int -> :py:class:`spinnaker.pyNN.buffer_management.storage_objects.buffered_sending_region.BufferedSendingRegion`) A dictionary of the buffers of events to send, indexed by the regions	None	sets the send buffer parameter which stores the buffered data that's going to be transmitted.

Table 22: Methods provided by the SendsBuffersFromHostPreBufferedImpl interface above the methods provided by the AbstractSendsBuffersFromHost interface.

Parameter	Description
_send_buffers	dict(int ->:py:class:`spinnaker.pyNN.buffer_management.storage_objects.buffered_sending_region.BufferedSendingRegion`) A dictionary of the buffers of events to send, indexed by the regions

Table 23: Parameters provided by the SendsBuffersFromHostPreBufferedImpl interface

An example model which uses the SendsBuffersFromHostPreBufferedImpl interface would be the ReverseIptagMultiCastSource. Even though there are no methods visible here, on Line 42 and 144 are where the interface is inherited and instantiated from.

Mapping Support Interfaces

The interface is used by models which know how many keys they need for each outgoing edge partition (I remind the reader that an edge partition is a subset of the total number of edges which have the vertex as the source point, which will use the same number and key values for routing purposes) and where the number of keys is neither 1 nor equal to the number of atoms the vertex contains. Take for example a computational atom has numerous states, where each state needs to be transmitted to another computational node, this would require multiple keys per atom.

AbstractProvidesNKeysForPartition

This class could easily be encapsulated as a constraint generator, or the next constraint interface, but due to the way certain bits of functionality were created, this interface came to be. In future releases, this may be removed. **Table 24** covers methods that need to be implemented by any object that inherits from the *AbstractProvidesNKeysForPartition*.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
get_n_keys_for_partition()	**partition:**(pacman.utilities.utility_objs.outgoing_partition.OutgoingPartition) the subset of edges that will share the routing and routing keys. **graph_mapper:**(pacman.model.graph_mapper.graph_mapper.GraphMapper) the mapping interface between partitionable and partitioned graphs.	iterable of constraints.	generates constraints for each partition that leaves this object. Returns a iterable of pacman.model.constraints.abstract_constraint.AbstractConstraint`

Table 24: The method needed to be implemented for objects that inherit from the AbstractProvidesNKeysForPartition class.

Constraint Based Interfaces

These next two interfaces support the addition of constraints onto the edges partitions based on which side of the edge the vertex is on. For example, edges that come into a vertex that represents multiple atoms (a PyNN neural model) requires a filterer / translator to interpret the event id into application based data. These translators / filterers can have restrictions on the types of keys that they can handle. The 2D master pop table requires that keys fit into the mapping process shown in **Figure 3**. This requires a constraint to be placed on that partition, and therefore to the keys given to the vertices which use the partition.

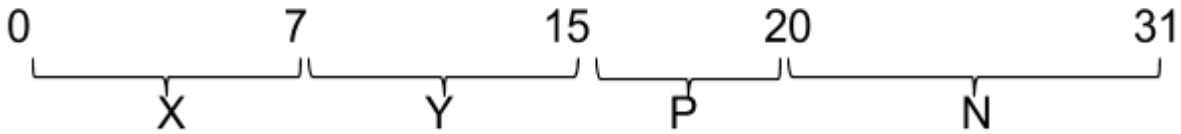

Figure 3: The Key mapping process needed by the 2D master pop table.

AbstractProvidesOutgoingPartitionConstraints

The AbstractProvidesOutgoingPartitionConstraints class supports adding constraints to partitions that use the given vertex as its source. **Table 25** shows the method that the GFE will use to add the constraints to during the mapping process. It does not provide any extra methods, nor any parameters.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
get_outgoing_partition_constraints()	**partition:**(pacman.utilities.utility_objs.outgoing_partition.OutgoingPartition) the subset of edges that will share the routing and routing keys.	iterable of constraints.	generates constraints for each partition that leaves this object. Returns an iterable of pacman.model.constraints.abstract_constraint.A

	graph_mapper:(pacman.model.graph_mapper.graph_mapper.GraphMapper) the mapping interface between partitionable and partitioned graphs.		bstractConstraint`

Table 25: The method needed to be implemented for objects that inherit from the AbstractProvidesOutgoingPartitionConstraints

An example of the usage of this interface, can be found in the SpyNNaker interface under the **AbstractPopulationVertex** where:

1. On line 668 the vertex adds the constraint that all its keys must be in a continuous range (reducing the data needed to be sent to the cores).

AbstractProvidesIncomingPartitionConstraints

The AbstractProvidesIncomingPartitionConstraints class supports adding constraints to partitions that use the given vertex as its destination. **Table 26** shows the method that the GFE will use to add the constraints to during the mapping process. It does not provide any extra methods, nor any parameters.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
get_incoming_partition_constraints()	**partition:**(pacman.utilities.utility_objs.outgoing_partition.OutgoingPartition) the subset of edges that will share the routing and routing keys. **graph_mapper:**(pacman.model.graph_mapper.graph_mapper.GraphMapper) the mapping interface between partitionable and partitioned graphs.	iterable of constraints.	generates constraints for each partition that have this vertex as a destination. this object. Returns a iterable of pacman.model.constraints.abstract_constraint.AbstractConstraint`

Table 26: The method needed to be implemented for objects that inherit from the AbstractProvidesIncomingPartitionConstraints

An example of the usage of this interface, can be found in the SpyNNaker interface under the **AbstractPopulationVertex** where:

2. On line 658 the vertex calls its synapse_manager which in turn asks the population table for any constraints. If we look at the master-pop_table_as_2d_array object, on line 294 to 311 iit states that all edges coming into it must have a given key format and a fixed mask.

Provenance Based Interfaces

The provenance interface supports the extraction of data that can be used to verify if 2 runs of the same simulation can be considered to have been executed the same way. There are a collection of data that is acquired from the SpINNaker machine already, ranging from the amount of time it takes for the PACMAN algorithms to execute, to the number of packets that have been dropped during the execution of the application code.

Due to the event based nature of the SpiNNaker machine, there is a collection of provenance data that can be collected from the vertex. For example, the number of timer tick callbacks that were queued at any given time point, which can reflect that your c code is becoming overloaded and cannot handle the current demands put on it. Due to the general nature of these provenance data items, we have built the *ProvidesProvenanceFromMachineImpl* and *AbstractProvidesProvenanceFromMachine* interface where general vertices allocate a Data Specification Region for storing/extracting general core based provenance data and also for storing/extracting application specific provenance data.

AbstractProvidesProvenanceFromMachine

The *AbstractProvidesProvenanceFromMachine* interface provides one abstract method that needs to be implemented by the end user's vertex, as shown in **Table 27**. No extra methods nor variables are given. This is the definition of the interface, and we recommend you inherit from one of the implementations of this interface for ease of use.

Method call	Inputs	Outputs	Description
__init__()	None	None	Constructor for this interface.
get_provenance_data_from_machine()	**transceiver**: (spinnMan.transceiver.Transceiver) the python interface to the SpiNNaker machine. **placement**: (pacman.model.placements.placement.Placement) the location of this object on the SpiNNaker machine.	iterable of spinnFrontEndCommon.utilities.utility_objs.provenance_data_item.ProvenanceDataItem	Is the method that will be called by the GFE to extract provenance data during *reset()* and *end()* functions described in Section **SpiNNaker Graph Front End: Python Script Interface**.

Table 27: The method needed to be implemented for objects that inherit from the AbstractProvidesProvenanceFromMachine

There are no examples of vertices using this interface directly, as all our vertices use the *ProvidesProvenanceFromMachineImpl* class discussed in **Section ProvidesProvenanceFromMachineImpl.**

ProvidesProvenanceFromMachineImpl

The *ProvidesProvenanceFromMachineImpl* class provides the implementation for extracting the standard provenance data items that can be extracted from the core. In future releases, this might expand and so it is wise to follow this interface closely. **Table 28** describes the extra functions above the ones defined in the *AbstractProvidesProvenanceFromMachine* interface. **Table 29** defines the parameters provided by this interface as well.

It is worth noting that the provenance data items support the ability to be nested in a structure through the use of its *names* parameter. This is exploited in the tools, as we by default write provenance data as a set of XML files. The *ProvenanceDataItem* also supports reporting when the data looks like something has gone wrong during the execution of the simulation through the use of the *report* flag and the *message* field.

Method call	Inputs	Outputs	Description
__init__()	**provenance_region_id**: (int) The Data specification region id where the provenance data will be stored. **n_additional_data_items**: (int) the number of application based provenance data items which will be put into the region by the application.	None	Constructor for this interface.
reserve_provenance_data_region()	**spec**: (data_specification.data_specification_generator.DataSpecificationGenerator) the data specification writer for the application vertex	None	reserve the provenance region, taking into account the size of the standard and application provenance data elements.
get_provenance_data_size()	**n_additional_data_items**:(int) the number of additional items being provided by the application vertex.	int	returns the total size, in bytes, of the provenance data region
_get_provenance_region_address()	**transceiver**: (spinnMan.transceiver.Transceiver) the python interface to the SpiNNaker machine. **placement**: (pacman.model.placements.placement.Placement) the location of this object on the SpiNNaker	int	returns the memory address on the chip within the SpiNNaker machine where the provenance data region resides.

	machine.		
_read_provenance_data()	**transceiver**: (spinnMan.transceiver.Transceiver) the python interface to the SpiNNaker machine. **placement**: (pacman.model.placements.placement.Placement) the location of this object on the SpiNNaker machine.	bytearray of ints	Returns the bytearray, of ints, of the provenance data region from the machine.
_get_placement_details()	**placement**: (pacman.model.placements.placement.Placement) the location of this object on the SpiNNaker machine.	str, int, int, int, array[str]	Returns a collection of data, which are: 1. the partitioned vertex label, 2. x coord of the chip where this partitioned vertex is located, 3. y coord of the chip where this partitioned vertex is located, 4. processor id of the core. 5. array with the human readable form of of the placement for file writer
_add_name()	**names:** (array of str) current hierarchy structure as a list of names. **name:** (str) the current name of this provenance data item	iterable of str	Returns the new hierarchy of the provenance data item.
_read_basic_provenance_items()	**provenance_data:** (bytearray of ints) The byte array from the spiNNaker machine. **placement**: (pacman.model.placements.placement.Placement) the location of this object on the SpiNNaker machine.	iterable of SpinnFrontEndCommon.utilities.utility_objs.provenance_data_item.ProvenanceDataItem	reads the provenance data items that are standard to all cores. returns them as a colelction of PorvenanceDataItem which can be appended upon by end user provenance data items.
_get_remaining_provenance_data_items()	**provenance_data**	byte array	Returns the byte array that only contains end user provenance data items.

Table 28: Overloadable methods provided by the ProvidesProvenanceFromMachineImpl interface.

Parameter	Description
_provenance_region_id	The Data specification region id where provenance data is expected to be stored.
_n_additional_data_items	The number of data items that the application wants to put into the provenance region.

Table 29: Parameters provided by the ProvidesProvenanceFromMachineImpl interface

An example of this provenance interface at work is the LivePacketGatherer.py and LivePacketGatherPartitionedVertex, where the following lines are used.

Live Packet Gatherer

1. Line 168: The live packet gatherer calls the partitioned vertex to create the provenance data regions.

LivePackertGathererPartitionedVertex

1. Line 13: Declares that it is inheriting from from the *ProvidesProvenanceFromMachineImpl*
2. Line 26: instances the *ProvidesProvenanceFromMachineImpl* interface with 2 extra provenance elements.
3. Lines 30 to 61: overloads the ProvidesProvenanceFromMachineImpl.get_provenance_data_from_machine() method
    a. Line 31: asks for the entire provenance data region data.
    b. Line 32-33: calls the *ProvidesProvenanceFromMachineImpl* to get its provenance data items.
    c. Line 34: filters the raw byte array to just contain application based data.
    d. Lines 35 to 59 generates and adds the 2 elements of provenance data items generated by the Live packet gatherer to the list genertated by the *ProvidesProvenanceFromMachineImpl* interface.

Vertex provided constraints

This section describes the constraints that a vertex can use. Table 30 describes these in detail whereas Table 31 defines which type of vertex each constraint can be connected with and what the import line is.

Constraint Name	Inputs	Description
PartitionerMaximumSizeConstraint	**size**: (int) The max size that this partitionable vertex can be broken down into.	Tells the PACMAN partitioner that at a max, it can break this apart by chunks of 'size'. Only used on partitionable vertices.
PartitionerSameSizeAsVertexConstraint	**vertex**: (pacman.model.partitionable_graph.abstract_partitionable_vertex.AbstractPartitionableVertex) The vertex to partition in the same fashion as.	Tells the PACMAN partitioner that the vertex that contained this constraint needs to be partitioned in the same way as 'vertex'.
PlacerChipAndCoreConstraint	**x**: (int) The x coord of a chip in the SpiNNaker machine. **y**: (int) The y coord of a chip in the SpiNNaker machine. **p**: (int / None) Either the processor id or None (meaning any core).	Tells the PACMAN placer that this partitioned vertex needs to reside on a specific chip/core on the SpiNNaker machine with coords X, Y, P.
PlacerRadialPlacementFromChipConstraint	**x**: (int) The x coord of a chip in the SpiNNaker machine. **y**: (int) The y coord of a chip in the SpiNNaker machine.	Tells the PACMAN placer that the partitioned vertices which come from this partitionable vertex needs to placed as close as possible from chip at coords X, Y
TagAllocatorRequireIptagConstraint	**ip_address**: (str) The IP address that the tag will cause data to be sent to. **port**: (int) The UDP port that the tag will cause data to be sent to **strip_sdp**: (bool) Whether the tag requires that SDP headers are stripped before transmission of data **board_address**: (str) Optional fixed board ip address **tag_id**: (int) optional fixed tag id required.	Tells the PACMAN tag allocator that this vertex needs a IPTAG (used for sending data out of the machine) which a given ip_address, port, tag_id etc.
TagAllocatorRequireReverseIptagConstraint	**port**: (int) The UDP port to listen to. **sdp_port**: (int) Optional SDP port number to be used when constructing SDP packets from the received UDP packets. **board_address**: (str) Optional fixed board ip address.	Tells the PACMAN tag allocator that this vertex needs a ReverseIPTAG (used for injecting packets to a specific core from host to machine) with a given UDP port.

	tag_id: (int) Optional fixed tag id required.	

Table 30: Constraints available to be put on vertices.

Constraint Name	Partitioned Vertex usage	Partitionable Vertex usage	Import lines
PartitionerMaximumSizeConstraint	Not relevant	Valid	**from** pacman.model.constraints.partitioner_constraints.partitioner_maximum_size_constraint **import** PartitionerMaximumSizeConstraint
PartitionerSameSizeAsVertexConstraint	Not relevant	Valid	**from** pacman.model.constraints.partitioner_constraints.partitioner_same_size_as_vertex_constraint **import** PartitionerSameSizeAsVertexConstraint
PlacerChipAndCoreConstraint	Valid	Not Valid[1]	**from** pacman.model.constraints.placer_constraints.placer_chip_and_core_constraint **import** PlacerChipAndCoreConstraint
PlacerRadialPlacementFromChipConstraint	Not relevant	Valid	**from** pacman.model.constraints.placer_constraints.placer_radial_placement_from_chip_constraint **import** PlacerRadialPlacementFromChipConstraint
TagAllocatorRequireIptagConstraint	Valid	Valid	**from** pacman.model.constraints.tag_allocator_constraints.tag_allocator_require_iptag_constraint **import** TagAllocatorRequireIptagConstraint
TagAllocatorRequireReverseIptagConstraint	Valid	Not valid[2]	**from** pacman.model.constraints.tag_allocator_constraints.tag_allocator_require_reverse_iptag_constraint **import** TagAllocatorRequireReverseIptagConstraint

Table 31: when each constraint is useful, and its import line.

[1] Unless there is a 1:1 mapping between partitionable and partitioned vertices (such as with a Live Packet Gatherer)
[2] Unless there is a 1:1 mapping between partitionable and partitioned vertices (such as with a Live Packet Gatherer)

SpiNNaker Graph Front End: Edges Interface

Within the SpiNnaker machine, edges represent communication between computational nodes, and therefore there are not any functional interfaces that can be applied to edges. This said, there are numerous types of Edge, each of which is treated differently within the tool chain.

To explain these types thoroughly, you need to understand the 5 different types of communication available by the SpiNNaker machine. These are:

1. Multicast packets.
2. fixed route packets.
3. nearest neighbour packets.
4. point to point packets.
5. SDP packets.

Of these 5 different types of message passing, nearest neighbour and point to point packets are reserved by the SpiNNaker low level software, and therefore are not available directly for end user usage. SDP packets are broken down into Point to Point packets during transmission through the SpiNNaker communication fabric and rebuilt into a SDP packet at the destination core and so the end user can utilize point to point messages via SDP packets.

This leaves the user with Multicast and fixed route packets for communication and the tool chain distinguishes between these communication paths through the use of Edge class types. There is currently a set of 5 different edges, as described in **Table 32**, and how to import them is described in **Table 33**.

Edge Type	inputs	Description
MultiCastPartitionableEdge()	**pre_partitionable_vertex**:(pacman.model.partitionable_graph.partitionable_vertex.PartitionableVertex) the partitionable vertex at the start of the edge **post_partitionable_vertex**:(pacman.model.partitionable_graph.partitionable_vertex.PartitionableVertex) the partitionable vertex at the end of the edge **label**:(str) human readable version of this edge	Represents the communication via multi-cast packets between 2 partitionable vertices. During the PACMAN partitioning algorithm, this edge will be broken down into a collection of MultiCastPartitionedEdge()
MultiCastPartitionedEdge()	**pre_partitioned_vertex**:(pacman.model.partitioned_graph.partitioned_vertex.PartitionedVertex) the partitioned vertex at the start of the edge **post_partitioned_vertex**:(pacman.model.partitioned_graph.partitioned_vertex.PartitionedVertex) the partitioned vertex at the end of the edge **label**:(str) human readable version of this edge	Represents the communication via multi-cast packets between 2 partitioned vertices.
MultiCastPartitionedEdgeWithNKeys()	**pre_partitioned_vertex**:(pacman.model.partitioned_graph.partitioned_vertex.PartitionedVertex) the partitioned vertex at the start of the edge **post_partitioned_vertex**:(pacman.model.partitioned_graph.partitioned_vertex.PartitionedVertex) the partitioned vertex at the end of the edge **n_keys**: (int) the number of keys this edge needs. **label**:(str) human readable version of this edge	Represents the communication via multi-cast packets between 2 partitioned vertices where the number of keys needed are not tied to the number of atoms the source vertex contains. Encapsulates the *AbstractProvidesNKeysForPartition* described in Section **AbstractProvidesNKeysForPartition**
FixedRoutePartitionableEdge()	**pre_partitionable_vertex**:(pacman.model.partitionable_graph.partitionable_vertex.PartitionableVertex) the partitionable vertex at the start of the edge **post_partitionable_vertex**:(pacman.model.partitionable_graph.partitionable_vertex.PartitionableVertex) the partitionable vertex at the end of the edge **label**:(str) human readable version of this edge	Represents the communication via fixed route packets between 2 partitionable vertices. During the PACMAN partitioning algorithm, this edge will be broken down into a collection of FixedRoutePartitionedEdge()
FixedRoutePartitionedEdge()	**pre_partitioned_vertex**:(pacman.model.partitioned_graph.partitioned_vertex.PartitionedVertex) the partitioned vertex at the start of the edge **post_partitioned_vertex**:(pacman.model.partitioned_graph.partitioned_vertex.PartitionedVertex) the partitioned vertex at the end of the edge **label**:(str) human readable version of this edge	Represents the communication via fixed route packets between 2 partitioned vertices.

Table 32: The different types of Edges with their inputs

Edge Type	import line
MultiCastPartitionableEdge()	**from** pacman.model.partitionable_graph.multi_cast_partitionable_edge **import** MultiCastPartitionableEdge
MultiCastPartitionedEdge()	**from** pacman.model.partitioned_graph.multi_cast_partitioned_edge **import** MultiCastPartitionedEdge
MultiCastPartitionedEdgeWithNKeys()	**from** spinnaker_graph_front_end.models.mutli_cast_partitioned_edge_with_n_keys **import** MultiCastPartitionedEdgeWithNKeys
FixedRoutePartitionableEdge()	**from** pacman.model.partitionable_graph.fixed_route_partitionable_edge **import** FixedRoutePartitionableEdge
FixedRoutePartitionedEdge()	**from** pacman.model.partitioned_graph.fixed_route_partitioned_edge **import** FixedRoutePartitionedEdge

Table 33: The different types of Edges and their import lines.

The edges that represent multicast communication are routed through the routers that reside on each SpiNNaker chip. These routers require Routing Tables and the PACMAN tools generate these automatically. This is not currently the same for the FixedRoute based communication, as this requires a different type of routing algorithm and different data to be loaded onto each chip's router. This can be added through the use of the **mapping_alogrithms** tutorial.

The constraints requested to be applied by the interfaces *AbstractProvidesOutgoingPartitionConstraints* and *AbstractProvidesIncomingPartitionConstraints* is applied to the edges during the mapping between partitions and number of keys. This is the only time when the constraints from vertices are applied to the edges during normal operation.

At this point, you should be able to build the first 2/3 components needed to generate new applications through the SpiNNaker tool chain. The next section talks about the executable code that will run on the SpiNNaker machine, and the interfaces which they contain.

SpiNNaker Graph Front End: c Code Interface

At this point, you should have both a python script that represents your application graph, and the python vertices and edges required to represent and run through the SpiNNaker tool chain. But we have not yet covered anything that actually runs on the SpiNNaker machine.

The code that runs on the SpINNaker machine is event based c, and we recommend reading the lab manual and slides on programming in event c located here to learn how to avoid mistakes that often catch new programmers out when dealing with the SpiNNaker programming model.

The rest of this document covers the 3 basic interfaces that are currently supported for the c code. These are currently all tied to the Data Specification language described in **Data specification language tutuorial**, and so if you plan to not use these interfaces, please take this in mind.

This in mind, these three interfaces are:

1. **data_specification.h** : This allows easy access to the memory address of the different Data Specification regions defined in the python code.
2. **simulation.h**: This supports the starting, provenance gathering and rerunning of code via the python *run()* and *reset()* commands.
3. **recording.h**: This supports the buffering out of recorded data through the buffered out process.

These three interfaces are described in **Sections Data Transmission Interface**, **Multi-run interface**, and **Buffered Recording Interface.**

Data transmission interface

The data_specifiation.h is designed to simplify the accessing of data through the Data Specification region structures, as well as verifying that the data specification file loaded is actually the one the c code was configured for. **Table 34** describes these functions.

Method call	Inputs	Outputs	Description
data_specification_get_data_address()	**None**	address_t	Returns the app pointer table base address for this core. This is stored in the user0 register.
data_specification_read_header()	**data_address:** (address_t) the base address for the app pointer table.	bool	Verifies if the Data Specification that was written into memory is the same version as the one the core expects.
data_specification_get_region()	**region:** (int) the data specification region id **data_address:** (address_t) the base address for the app pointer table.	address_t	returns the address of where the data starts for the given region.

Table 34: Methods provided by the data_specification.h interface.

If we look at the heat_demo.c file:

1. Line 410, the code requests the memory address for the App pointer table for this core,
2. Line 413 asks to verify the data specification data is in the right version,
3. Line 419 asks to get the memory address for the first region in the App Pointer table, most commonly known as the system region.

Multi-run interface

The multirun interface ties in all the expected state changes etc when it comes to updating the runtime and synchronizing all application cores so that they start running “near synchronously”. **Table 35** describes the different functions provided by the simulation.h interface.

Method call	Inputs	Outputs	Description
simulation_read_timing_details()	**address**: (address_t) the base address of the system region **expected_application_magic_number**: (uint32_t) the hash code of the vertex binary. **timer_period**: (uint32_t*) pointer to the variable that the application code uses for setting the time between timer ticks **n_simulation_ticks:** (uint32_t*) pointer to the variable used by the application code to identify when to stop. **infinite_run:** (uint32_t*) pointer to the flag that states if this application core is to run forever.	bool	This method extracts the configuration data stored by the *AbstractDataSpecableVertex* and *AbstractPArtitionedDataSpecableVertex's _write_basic_setup_info()* method defined in Section **Data Transmission interfaces**
simulation_handle_pause_resume()	None	None	places cores into a sync state and extracts provenance data. Supports updating runtime's.
simulation_run()	**timer_function:**(callback_t) the function to call every timer tick event. **timer_function_priority:** (int) the priority that this callback will require when setting the callback up.	None	places cores into a sync state, and sets off the timer tick callback. Causes a blocking action till the python interface informs them to start running.
simulation_register_simulation_sdp_callback()	**simulation_ticks_pointer:**(uint32_t*) pointer to the variable the application code uses for identifying when to stop **infinite_run_pointer:** (uint32_t*) pointer to the	None	Supports the end user setting off a SDP callback when linked to system sdp messages.

	flag that states if this application core is to run forever. **sdp_packet_callback_priority:** (int) the priority that the sdp callback should take.		
simulation_register_provenance_function_call()	**provenance_function:**(prov_callback_t) the function used by the application to store its own provenance data. **provenance_data_region_id:** (uint32_t) the data specification region id where provenance data is stored.	None	Registers functions for calling during simulation_handle_pause_and_resume for extracting application based provenance data..

Table 35: methods provided by the simulation.h

If we take the heat_demo.c as the example code, we see that:

1. Line 576 we register the SDP callback priority.
2. Line 597 we call the simulation run function, to lock into a synchronisation barrier.
3. Line 316 we call the simualtion_handle_pause_and_resume function, as we have finished running for the given timer period and now need to await new commands.
4. Lines 87 to 89 build control variables that we hand over to the simulation functions.

Buffered recording interface

The buffered recording interface supports the recording of state during execution. This interface encapsulates the buffered output interface, but can easily be used without the buffered output functionality. **Table 36** describes the methods provided by the interface.

Method call	Inputs	Outputs	Description
recording_initialize()	**n_regions:** (uint8_t) the number of regions to be recorded, one per type of data **region_ids**: (uint8_t *) the ids of the regions to be recorded to **recording_data**:(uint32_t*) The start of the data about the recording. Data is {uint32_t tag; uint32_t buffer_size_before_request; uint32_t size_of_region[n_regions]} **state_region**: (uint8_t) The region in which to store the end of recording state information **buffering_priority**: (uint32_t) The priority of the callback related to the buffering of the recording **recording_flags**: (uint32_t*) Output of flags which can be used to check if a channel is enabled for recording	bool	initialises the recording of data. returns True if it was able to initialize and False otherwise.
recording_record()	**channel**: (uint8_t)the channel to store the data into. **data**: (void *)the data to store into the channel. **size_bytes**: (uint32_t) the number of bytes that this data will take up.	bool	records some data into a specific recording channel.
recording_finalise()	None	None	Finishes recording - should only be called if there is a region being recorded.

recording_is_channel_enabled()	**recording_flags**: (uint32_t) The flags as returned by recording_initialize **channel**: (uint8_t) The channel to check	bool	Determines if the given channel has space assigned for recording, returns True if it is, false otherwise.
recording_do_timestep_update()	**time**: (uint32_t) the current timer tick count	void	Call once per timestep to ensure buffering is done - should only be called if some recording is actually being done.

Table 36: methods provided by the recording.h

If we take the heat_demo.c as the working example, we can see that:

1. Lines 525 to 529 creates a list of one region where it records its temperature each timer tick.
2. Lines 531 to 535 informs the recording interface that its recording one region which is region stores the Buffered output state field.
3. Line 310 to 313 stops the recording when it has finished running for the given time period.
4. Line 384 records the current temperature into the recorded region.

By using the simulation.c functionality at a minimum, then you should be able to generate and execute a programme through the GFE now. Good luck.