Managing Big SpiNNaker Machines

Steve Temple - 18 May 2013 - Version 0.01

Introduction

This note describes some ideas I have had recently related to managing big SpiNNaker machines. I am referring to machines which can go all the way up to a maximal, 64K node, machine.

I have been thinking in terms of loading applications into the machine, controlling them to start and stop them and determining their state in an efficient way. One desirable feature is to be able to run multiple applications on the machine and I present some ideas on how to do that.

This is a proposal for how to do things and although I have prototyped most of the ideas here, I am happy to take on board suggestions for other ways to do things or to throw it all away if it's deemed to be of no use! Comments welcome...

SpiNNaker Applications

By application, I mean a program, such as a SNN simulation which occupies a number of cores in the machine. Such an application must be loaded into the requisite number of cores, provided with any data that it needs to do its job and then started. I don't address the issue of loading application data into the machine here!

I propose to define an application as a program image which is loaded onto a SpiNNaker core and has exclusive use of that core until it terminates. Currently, we build applications into APLX files. Many cores may be loaded with the same application and to identify that set of cores I propose that a given application be allocated an Application ID (AppID), which is an 8-bit number. Many different applications can be loaded into a SpiNNaker machine at a given time and they are distinguished by having different AppIDs. The allocation and management of AppIDs is a host-side function.

A particular computation may require several different applications to be loaded into SpiNNaker which then cooperate in some way. This is catered for though it will be necessary to allocate numerically related AppIDs to the applications in order to control the computation as a whole.

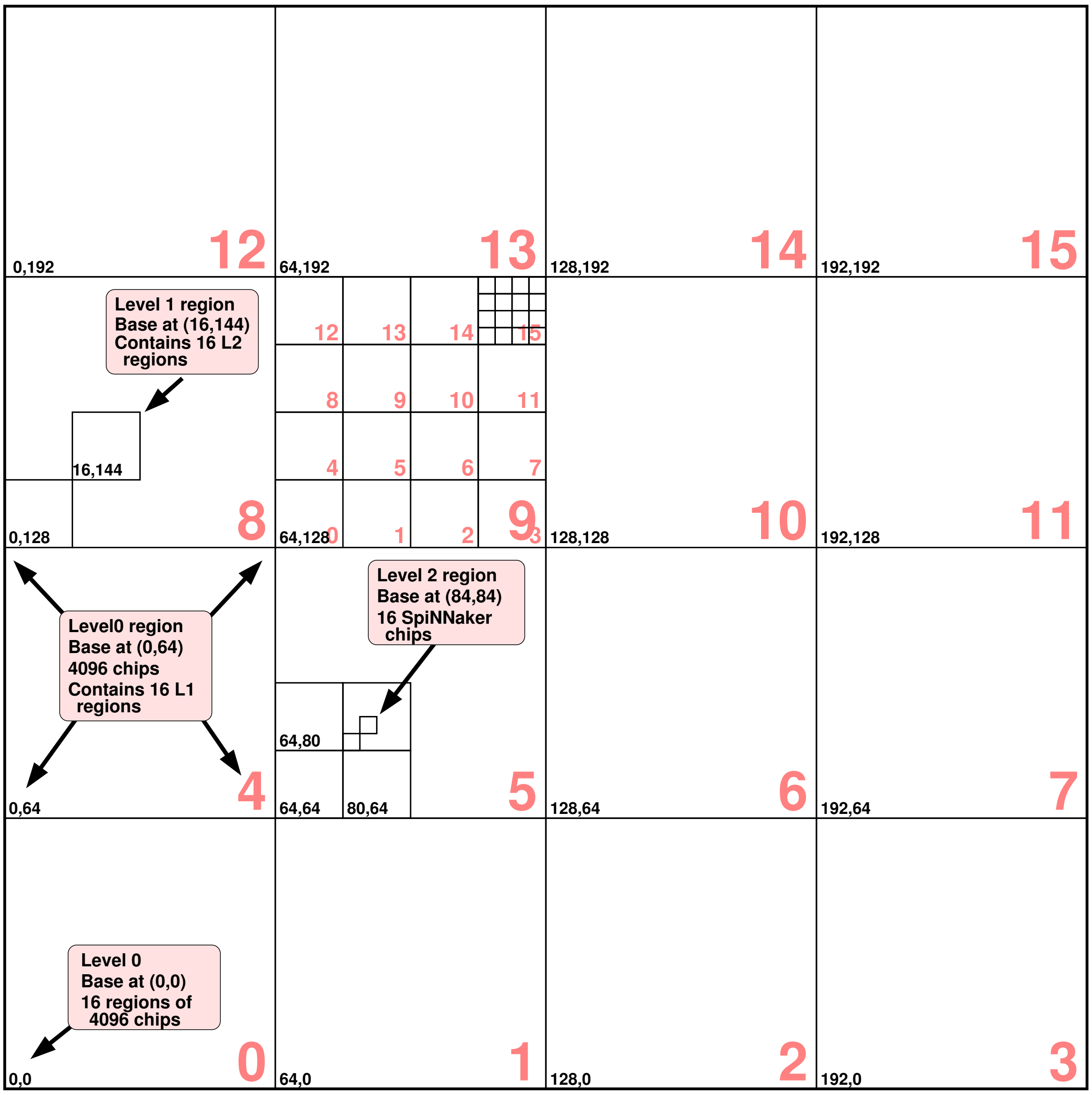
Core Allocation to Applications

A given application will need to be loaded onto many cores on the machine. I assume that this number is known at load time and will be static throughout the execution of the application. I propose a scheme of core allocation whereby a given application is loaded into a predefined number of chips and occupies a subset of the cores on each chip. For simplicity, the same subset of cores must be used on each chip. For example, an application might be loaded into cores 1 to 8 of 48 chips. Other applications could occupy cores 9 to 16 (or 17) of the same set of chips (or a sub- or super-set).

Machine Division for Core Allocation

In order to allocate chips and cores on the machine to many different applications simultaneously I propose a means to sub-divide the machine into addressable regions, based on a hierarchical tree where each parent in the tree has 16 children. I refer to this as an HD-tree (HD = hexadecimal).

I assume that the machine is a 2D mesh of SpiNNaker chips and that a maximal machine has 64K chips arranged as a 256x256 square. If we divide this machine up using an HD-tree we get 4 levels of hierarchy before we arrive at a single chip. I refer to the various levels as Level 0 through Level 3 and the square areas of chips at each level as Regions. Each level contains 16 regions and these are numbered 0 to 15 (see figure 1).



0,192
12
64,192
13
128,192
14
192,192
15
Level 1 region Base at (16,144) Contains 16 L2 regions
16,144
0,128
8
64,128
9
128,128
10
192,128
11
Level 2 region Base at (84,84) 16 SpiNNaker chips
Level0 region Base at (0,64) 4096 chips Contains 16 L1 regions
64,80
0,64
4
64,64
80,64
5
128,64
6
192,64
7
Level 0 Base at (0,0) 16 regions of 4096 chips
0,0
0
64,0
1
128,0
2
192,0
3

Figure 1: SpiNNaker Levels and Regions

Level 0 views the machine as 16 regions each containing 4096 chips. Each of these level 0 regions is subdivided into 16 level 1 regions of 256 chips. Each level 1 region is subdivided into 16 level 2 regions of 16 chips and each level 2 region is divided into 16 level 3 regions of 1 chip each. Note that each region can be very simply described in terms of point-to-point (P2P) addresses provided we stick with the current rectangular grid arrangement of these addresses.

Allocation of chips to a particular application is done at one level only. Up to 16 regions at a particular level may be allocated to an application. So to allocate half of the machine to an application, 8 regions would be allocated from level 0. An application which required 768 chips would be allocated 3 level 1 regions.

Note that the unit of allocation is the chip but an application can occupy a partial set of the cores on each chip so each chip can host several applications in disjoint sets of cores.

The allocation of applications to levels and regions is a moderately complex algorithm (if it is to be done optimally) and I would expect the controlling host to be responsible for this. The administration of levels and regions within the machine is controlled by the monitor processors.

Describing Region Allocations

I propose a hierarchical naming scheme for region allocations based on a dotted notation similar to IP addresses. An Allocation Descriptor consists of one to four numeric fields separated by ".". The value of each field is an integer in 0..15. The final field may be a range or list which denotes a set of regions at the same level. The first field denotes regions at level 0, the second (if present) at level 1, etc. Some examples

0-15	Level 0 regions 0-15 - the entire machine
0.0.0.0	Level 3 region 0 - the root chip
0.0.0.0-1,4-5	Level 3 regions 0-1, 4-5 - a 4 chip board
0.0.5	Level 2 region 5 - 'quarter' of a Spin4 board
6.0-5	Level 1 regions 0 to 5 of level 0 region 6 - 1536 chips

Because we can also allocate a specific set of cores on each chip in the allocation, the notation above is extended by having a second field, separated from the first by "/", which is a list or range of cores allocated to the application. The range of allowable values is integers from 1 to 17. Some examples

/1-16	Cores 1 to 16
/1-4,9-13	Cores 1 to 4 and 9 to 13
/1,3,5,7	Odd cores from 1 to 7

So to allocate the entire machine but using only cores 1 to 16 the allocation descriptor would be

0-15/1-16

Encoding Allocation Descriptors

Because it is necessary to pass allocation descriptions around the machine efficiently, I propose an encoding for chip allocation which fits in 32 bits and so can be carried in any of the SpiNNaker packet types. The location of a particular region is specified by the point-to-point address of its lower left corner (as an X,Y pair). The number of significant bits in the X and Y coordinates varies according to the level - higher numbered levels need more bits.

The format of the allocation word is as follows

Base X Addr	**Spare**	**Base Y Addr**	**Level**	**Region Mask**
(6 bits)	**(2)**	**(6 bits)**	**(2)**	**(16 bits)**

The Level specifies the level at which this allocation takes place. If it is 0, the unit of allocation is level 0 regions (4096 chips). If it is 3, the unit of allocation is level 3 regions or individual chips (etc).

The Base X and Y coordinates are the P2P address of the lower left corner of the square which encloses all of the regions at the specified level. This is (0, 0) for level 0 regions. Note that the lower 2 bits of the (8-bit) X and Y coordinates are never needed as the smallest region that needs to be addressed using P2P addresses is 16 (4x4) chips.

The Region Mask is a 16-bit mask which specifies which of the 16 possible regions at the given level are part of the allocation.

So an allocation of half the machine could be specified as

```
X = 0,  Y = 0, Level = 0, Mask = 0b0000000011111111
```

```
Allocation Descriptor: 0-7
```

and 768 chips as noted above (three level 1 regions) as

```
X = 64, Y = 128, Level = 1, Mask = 0b0000000011100000

Allocation Descriptor: 9.5-7
```

(X=64, Y=128) is the coordinate of the 9th (of 16) level 0 regions in the machine. Level 1 regions are always allocated from within a single level 0 region.

In addition to this word which describes the chip-level allocation, a bit-mask word will accompany it which describes the set of cores on each chip that the application occupies. This contains an 18 bit field where bit 0 is unused and bit 17 may not be used if any chip in the allocated regions has less than 18 good cores. The spare bits in this word can be used to hold the AppID for the application which occupies this allocation.

AppID **(8 bits)**	**Unused** **(6 bits)**	**Core Mask** **(18 bits – bit 0 unused)**

Loading Applications

At the moment, we have two ways of loading applications onto SpiNNaker. Flood-fill sends an application to every chip in the machine and starts it on a selected subset of cores. This is efficient but causes the application to start on all chips. The second method requires that applications are loaded individually to each chip or core but this can be very inefficient with the current hardware.

I propose that the flood-fill loader be modified so that it knows about region allocations. The application will still be sent to every chip but the Allocation Descriptor sent with the flood-fill data is used as a filter so that the application is only started on the desired set of chips and cores. The Application ID will also be sent with the flood-fill data so that the monitor processor on each chip can keep track of which applications are loaded.

The Application Life-cycle

Current applications are built by combining SARK, the Spin1 API and code which implements the specific functionality of the application. At present, this code is loaded into application cores by the monitor processor which places the code in a known location and then interrupts the code currently running on the application core to cause it to start the new code. This works quite well but I propose that in future a better way is for the monitor processor to soft-reset the application cores to ensure a 'clean start'. This means that an application core which has crashed can always be restarted with a new application.

I propose that we track the progress of each application running on a core by logging the various states that it passes through on the way from start-up to termination. This Core State should be visible to the monitor processor (eg in shared RAM) so that it is always accessible even if the application core is busy or has crashed for some reason. A maximum of 16 core states are allowed.

Refer to figure 2 in the following description. Following loading by the monitor processor, the application is started either by soft-reset (as above) or by the current (interrupt) method. At

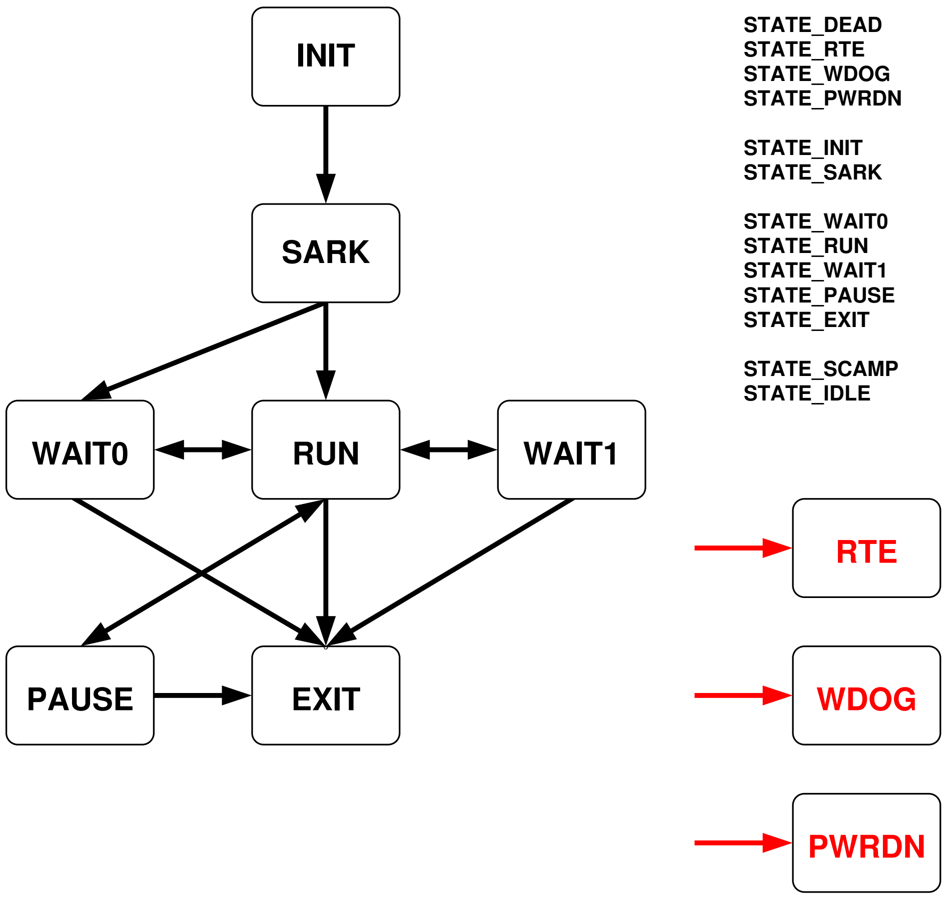


Figure 2: Application States

this point the core is in the (hopefully) transient INIT state. When the application starts to execute, it enters the SARK initialisation code and goes to state SARK. Simple applications which run without an API may stay in this state. More complex applications will probably run under an API and when they enter the API code they go to state RUN.

In order to synchronise the start-up of many cores spread across many chips we may want to wait until all of the cores have reached a given point in the code. This is barrier synchronisation, currently implemented in the API. If this synchronisation is required, the core enters state WAIT0. The signal to proceed past the barrier causes the state to change to RUN. When the program finishes whatever it needs to do it enters state EXIT and stays there until the core is reallocated. While in state RUN, the application may want to perform further barrier synchronisations and can enter states WAIT0 or WAIT1 to do this (I'm not sure whether or not we need 2 wait states!).

We have in the past discussed if it is possible to 'pause' an application in order to get the chance to inspect a running application. I propose a state PAUSE which denotes when an application has paused for this reason.

In addition, there are a number of states which can be reached from anywhere and denote some sort of problem. State RTE (run time error) is reached when an unrecoverable error occurs in the application (eg a data fetch abort). State WDOG occurs when the application core fails to respond to periodic pokes from the monitor processor and it is assumed that the core has died. State PWRDN is entered when the monitor processor turns off the clock to the application core because it was requested to do so (probably by the host).

Communicating with Applications

Running applications currently communicate with each other and with the host using either multicast (MC) packets or SDP. I propose another method of communication whereby a monitor processor (and hence the host) can send very short messages to specific applications, addressed by their Application ID. I refer to these messages as Signals. These messages need to be broadcast to all chips which host the application specified in the AppID. I have prototyped this mechanism using P2P packets but it could also be done less efficiently but possibly more robustly with NN packets.

I propose a small set of signals which may be sent in this way. These can be used to start, stop and pause applications or to cause them to pass through synchronisation barriers. In addition, the signals can be used to gather information from applications to query them about their state. Here are some suggested signals (some of which may resemble Unix signals!)

SIG_STOP	stop (pause) the application
SIG_CONT	continue (unpause) the application
SIG_KILL	exit the application
SIG_USR0	send user signal to application
SIG_USR1-USR3	(as above - 4 user signals)
SIG_GO	proceed through barrier
SIG_PWRDN	shut down core
SIG_INIT	re-initialise the core (with IDLE application)
SIG_RESET	reset core (restart application)
SIG_STAT	request statistics

All of these signals except the last don't need to return any information to the sender. SIG_STAT however allows three queries to be made

COUNT state	return number of cores in given state
AND	return AND mask of core states
OR	return OR mask of core states

The count operation simply counts the number of cores with given AppID which are in the state specified. This can be used, for example, to determine when all cores have reached a barrier. The AND operation returns a 16-bit mask with one bit for each possible state. The 1-hot encoding of the state of each core is ANDed and returned to the caller. This makes it possible to determine if all cores are in a particular state. The OR operation is similar but the 1-hot encodings are ORed together and this is used to determine if any core is in a particular state.

To allow cores with differing AppIDs to be targeted by signals, the AppID is accompanied by an 8-bit AppMask which is applied (as don't-cares) to the AppID when the signal is sent to each core. This allows ranges of AppIDs to be addressed or, with mask of zero, all cores regardless of AppID.

Conclusion

Draw your own!