A Hierachical Configuration System for a Massively Parallel Neural Hardware Platform

Francesco Galluppi*
APT group
School of Computer Science
The University of Manchester
Manchester, U.K.

Sergio Davies
APT group
School of Computer Science
The University of Manchester
Manchester, U.K.

Alexander D. Rast
APT group
School of Computer Science
The University of Manchester
Manchester, U.K.

Thomas Sharp
APT group
School of Computer Science
The University of Manchester
Manchester, U.K.

Luis A. Plana
APT group
School of Computer Science
The University of Manchester
Manchester, U.K.

Steve B. Furber
APT group
School of Computer Science
The University of Manchester
Manchester, U.K.

ABSTRACT

Simulation of large networks of neurons is a powerful and increasingly prominent methodology for investigate brain functions and structures. Dedicated parallel hardware is a natural candidate for simulating the dynamic activity of many non-linear units communicating asynchronously. It is only scientifically useful, however, if the simulation tools can be configured and run easily and quickly. We present a method to map network models to computational nodes on the SpiNNaker system, a programmable parallel neurally-inspired hardware architecture, by exploiting the hierarchies built in the model. This PArtitioning and Configuration MANager (PACMAN) system supports arbitrary network topologies and arbitrary membrane potential and synapse dynamics, and (most importantly) decouples the model from the device, allowing a variety of languages (PyNN, Nengo, etc.) to drive the simulation hardware. Model representation operates on a *Population/Projection* level rather than a single-neuron and connection level, exploiting hierarchical properties to lower the complexity of allocating resources and mapping the model onto the system. PACMAN can be thus be used to generate structures coming from different models and front-ends, either with a host-based process, or by parallelising it on the SpiNNaker machine itself to speed up the generation process greatly. We describe the approach with a first implementation of the framework used to configure the current generation of SpiNNaker machines and present results from a set of key benchmarks. The system allows researchers to exploit dedicated simulation hardware which may otherwise be difficult to program. In effect, PACMAN provides automated hardware acceleration for some commonly used network simulators while also pointing towards the advantages of hierarchical configuration for large, domain-specific hardware systems.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Neural networks*; D.2.11 [**General**]: Software Architectures—*Domain-specific architectures*

Keywords

Spiking Neural Networks, Neuromorphic Hardware, Parallel Hardware, SpiNNaker

1. INTRODUCTION

Large-scale models of spiking neural networks are an essential tool to test hypotheses regarding the mechanisms of information processing in the brain [33] [12] or to exploit the computational capabilities of networks of neurons [15] [31]. An improved knowledge in this are would lead to more accurate understanding of brain-related diseases and of the functions that robotics and machine learning are trying to mimic with cognitive computing [2]. The brain is composed of numerous non-linear units (neurons) communicating asynchronously and organised in a hierarchical fashion [28]. This makes simulation on standard parallel hardware challenging due to synchronization and communication overheads [39]; dedicated hardware emerges then as a natural candidate simulation substrate. While a variety of modelling tools make neural modelling easier from a user point of view, they introduce technological challenges [5]. Study of efficient modelling technologies is therefore a key requirement for computational neuroscientists [35]. Various hardware architectures try to exploit parallelism to circumvent software limitations [34] [2]. However, to be a generic platform for neural exploration, both atomic elements (neurons and synapses) and connection topologies must be easily configurable and extensible so some approaches focus on a software abstraction layer which can easily reconfigure the hardware and make it accessible to non-hardware experts [36] [14] [6] [19].

In this context we present an approach to decouple the

*francesco.galluppi@cs.man.ac.uk

CF'12, May 15–17, 2012, Cagliari, Italy.

model specification stage and the generation of data structures representing the model on a parallel system. This approach can be used to configure different types of computational nodes (neurons) connected in an arbitrary way and map them to a dedicated architecture. By organising neurons into homogeneous populations, the system maintains hierarchical information about the model, using it to drive the allocation on the system in a simpler way than by considering single neurons. We begin by reviewing most popular neural simulation strategies in software and hardware in section 2, in order to introduce the SpiNNaker system, a massively parallel digital architecture oriented to the simulation of large networks of spiking neurons in section 3. To demonstrate the proposed approach we present an implementation of the abstraction layer which translates a model from different front-ends and allocates it on the SpiNNaker machine in section 4. Results obtained using this configuration approach are in 5. Finally, we discuss the challenges and potential in such an approach and overall implications in the last two sections.

2. NEURAL NETWORK SIMULATORS

Simulation of neural network models can be done either on dedicated hardware or with general purpose hardware running software simulators. While the first offers more efficiency and higher speeds, the latter offers more flexibility in the neural model and the interconnectivity.

2.1 Hardware

The large number of nodes (neurons) and interconnections (synapses) required to simulate neural networks has led to different hardware solutions. Each tries to exploit the intrinsic parallelism of neural networks directly in hardware. Efficiently mapping a neural network onto a hardware substrate is a nontrivial task, due to the massive parallelism and communication bandwidth required by the high connectivity. Different approaches have been proposed to circumvent this problem:

Supercomputers, notably the IBM Blue Gene, have been used in brain modelling at different levels of abstraction, from the ion-channel level [33] to point neurons [2]. Such a use of supercomputers is generally tightly optimised to the model simulated, and accessibility to such computational resources is not widespread. Finally, the power requirements make scaling of the model challenging.

Graphical Processing Units (GPUs) are more mass-market devices, and with their inherent parallelism they offer a reasonable alternative for medium-scale neural modelling. GPUs can also easily be configured to run arbitrary models by interfacing them with PyNN [14] or PyNN-like interfaces [36], making them accessible to non hardware experts. As the limiting factor is the access to memory [36], however, GPUs are primarily suited for high computation-to-communication-ratio models [3]; in fact the maximum available bandwidth can be accessed only using highly idealised memory access patterns [14]. The other major limiting factor is power, which affects scalability, a typical GPU consuming $\sim$250W/chip.

Neuromorphic Hardware: ASICs (Application Specific Integrated Circuits) can be used to effect a specific neural model in circuitry [27]. While such an approach is very power- and compute- efficient [41] [26], it comes at the price of reconfigurability, since the neural model or the connectivity [9] is hard-wired. Some systems [34] [6] use an FPGA-like lookup to overcome connectivity limitations, combining analogue neural simulation with digital packet-based spike communication to support more general connectivity patterns, albeit at lower density. Another way to enhance reconfigurability of such hardware is to use an FPGA for storing connectivity information in multi-chip analogue SNN system [49]. Nonetheless flexibility remains the principal challenge.

Field-Programmable Gate Arrays (FPGA) are used standalone as exploratory tools since their reconfigurability softens the low flexibility of neural models cast into silicon [8]. However mapping very high connection densities in FPGAs is challenging due to the circuit-switched architecture, and cost, power consumption and performance scales poorly [32].

2.2 Software

Simulation on dedicated hardware offers an efficient way of modelling, but data exchange between hardware models is difficult since lack of standards means that the data organisation is typically proprietary. Simulation in software makes flexible and exploratory modelling accessible to a wide community. According to a survey conduced by NeuroDebian [24] the most popular tools for neural modelling are[1]:

NEURON [7], a modelling and simulation tool oriented to the simulation of finely detailed neurons which can be connected in complex networks. It is a very popular instrument for neuroscientists as it helps model the physical properties of the neurons, but also for network modelling by expansion through python bindings. As of today more than 1000 publications have used NEURON[2].

Brian [22], a neural network simulator entirely written in Python, oriented to the simulation of large networks of point neurons. In addition to standard neuron models, powerfully, using Brian the user can input the equations to be solved by the simulator directly in a standard notation. The choice of Python as a native language, which has "turnkey" packages for numeric and vector-based computation as Scipy and Numpy, rather than using C/C++ like GENESIS and NEST, lets the user run the simulations, plot and analyse data all in the same environment.

PyNN [11], a Python-based standard description language able to run a model on a chosen supported back-end simulator (most of the ones mentioned here). The declared objective of the language is to *"write the code for a model once, run it on any supported simulator without modification"* [11]. It is composed of a standardised set of neuron, synaptic, plasticity and connector models.

Nengo [44], the tool used to implement the Neural Engineering Framework principles [12]. In this respect it is a tool which is specific to a modelling framework rather then a general purpose system, requiring specific implementation strategies. It affords the user the possibility to map a wide range of neuro-computational dynamics to spiking neural networks.

[1]http://neuro.debian.net/survey/2011/results.html
[2]http://www.neuron.yale.edu/neuron/static/bib/usednrn.html

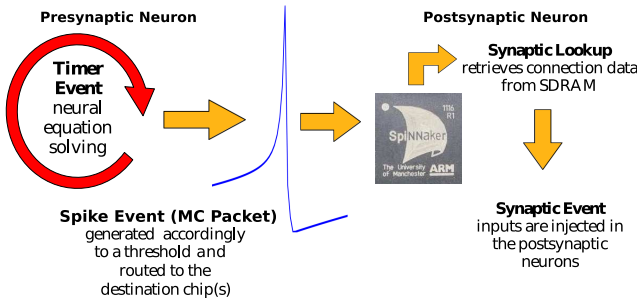

Figure 1: Neural Simulation Events

GENESIS (the GEneral NEural SImulation System) [4], a tool for realistic simulation based on physical structures and biological properties. Using an object oriented scripting language, users configure abstract objects and can interact with the simulation through a GUI, while the computation is carried out in the underlying Script Language Interpreter (SLI) written in C.

NEST (NEural Simulation Tool) [20] can be used to simulate large networks composed of standard neuron models efficiently. It is able to model Leaky Integrate-and-Fire and Hudgkin-Huxley neurons, along with various synaptic models and plasticity methods. Initially based on a proprietary core language, it now supports interaction with Python through PyNest [13]. NEST supports parallelisation through multithreading and message passing [39].

3. SPINNAKER SYSTEM

3.1 Architecture

The SpiNNaker System is a programmable asynchronous massively parallel multi-core system oriented to the simulation of heterogeneous large-scale models of spiking neural networks [17]. Each SpiNNaker chip contains 18 ARM968 cores embedded in a programmable, packet based, network on chip [38]. Spikes are encoded as source-based AER [30] event packets and transmitted through a Multicast (MC) Router capable of handling one packet per clock cycle. Every core has a local Tightly-Coupled Memory (TCM - 32Kb for instruction and 64Kb for data), while each core has access through a dedicated DMA controller to a 1Gb SDRAM shared by the 18 ARM cores within a single chip, with an aggregate memory bandwidth of 5.6 Gb/s. SDRAM is partitioned into regions containing core-specific synaptic information, eliminating the issue of memory sharing across the system. The memory system is thus local to every chip, circumventing the challenges needed on, e.g. GPUs to access memory [3] and maintain process coherency [36], while keeping power consumption lower than on such systems. Each chip can be connected to 6 adjacent neighbours in a toroidal mesh using bi-directional asynchronous links for an aggregate spiking bandwidth of 1.5 Gb/s [37], supporting reconfigurable arbitrary connectivity [16]. Arguably a MC mesh NoC is the most suitable interconnect architecture for reconfigurable neural network implementations [46]. The custom packet-switching network [38] is easily reconfigurable by changing the MC routing tables and more wire-efficient than a circuit-switched architecture [32] [8].

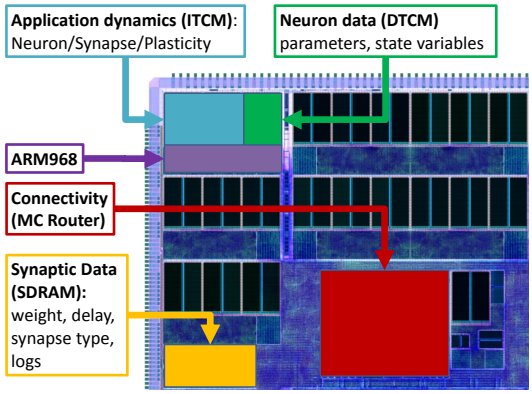

Figure 2: Functional blocks on a section of a SpiNNaker Chip

The SpiNNaker System offers an alternative set of trade-offs to neuromorphic chips [26] and programmable, standard computing systems [2]. While being outperformed in absolute power consumption [27] or speed [41] by dedicated analog hardware, and in representational flexibility by general-purpose computers, SpiNNaker offers a scalable and completely reconfigurable platform for exploration of a wide range of network models, within a low power budget of 1 W/Chip [15]. The full system is designed to contain up to 65,536 chips and more than a million cores [15].

3.2 Hardware Reconfigurability

From a computational and communication point of view each ARM core offers flexibility in the complexity and number of neurons and the way they are embedded in a configurable network based on the Multicast Router. The number of neurons which can be modelled on a single core depends on factors including the activity of the neurons, the computational power needed to solve the neurodynamic equations in real time, the number of synapses and the memory occupancy. The software that allocates neurons on board must facilitate exploring such trade-offs so as to be able to experiment with different configurations. As introduced in the previous section, the SpiNNaker system is an architecture with multiple dimensions of configurability, in particular:

Application reconfigurability: single cores in the system can be configured to run different applications, for instance different neural, synaptic or plasticity kernels, or non-neural applications like data collection and on-board analysis. Each application needs to be configured with its parameters (eg. governing the neural equations). Seamless integration of new applications into the existing framework is critical, so to have them as available resources in system deployment.

Connection reconfigurability: the MC router system permits arbitrary network topologies to be mapped through a 2-stage routing and lookup system which is able to route a spike efficiently to many different destination neurons placed anywhere in the system. Spikes are encoded as source-based AER events: The MC routers manage the first stage through the leading field of the routing key (processor coordinates in the system), while the lookup phase deals with the second field of the routing key (neuron id within a processor). The

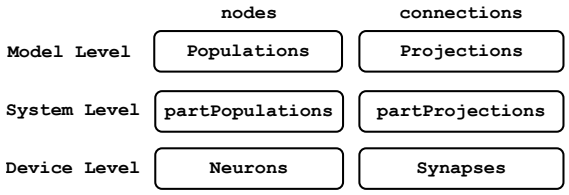

Figure 3: Hierarchical Approach

field definitions themselves are not fixed but programmable through mask bits in the router.

3.3 Neural Applications

While the SpiNNaker system is a general purpose scalable parallel machine, its architecture is designed for simulation of large networks of spiking neurons, as they can be modelled efficiently by representing spikes as stereotypical events encoded in an MC packet. Neural applications (or kernels) are based on a set of APIs [43] written in C which expose the event-driven nature of the system. From a user perspective we distinguish 2 events (fig. 1):

Timer Event (Neural Event): each core contains a programmable timer generating interrupts at configurable time steps (eg. every msec). At each time step, for each neuron, the equations are solved and spikes generated as required.

DMA Done (Synaptic Event): once a spike arrives to its destinations, DMA retrieves the relative synaptic information from SDRAM and the input is injected into the post-synaptic neurons.

Figure 1 shows how events interact: each time step produces a timer event. Neural equations are solved (according to the neural application, parameters and state variables) and a Spike Event is produced if a condition is met (eg. the membrane potential crosses a threshold). Spikes are encoded as source-based AER packets and are routed to their destination cores by the MC routers. On the post-synaptic chip synapses are retrieved locally using a source-based lookup to access SDRAM; synaptic inputs are then injected into the target neurons according to the synaptic model used.

Neural applications use run-time, model-dependent data to configure neural dynamics and connectivity. These need to be compiled for the system, translating the model (neurons and synapses with associated parameters) into SpiNNaker data structures. Such data structures are presented in fig. 2 and include the individual neural parameters, the routing tables to route spikes from source to destinations, the lookup table to retrieve synaptic data from memory and the synaptic data itself. To configure the entire system, several data structures need to be created after having allocated and mapped the model on the architecture.

To reconfigure and expand a system of thousands of chips and up to about a million cores rapidly, data structure creation needs to be as flexible and efficient as possible. The next section presents the proposed approach.

4. MODEL TO HARDWARE

By introducing a translation level (*map* or system level) from one to the other layer, the approach effectively decouples the device from the neural network model level. This

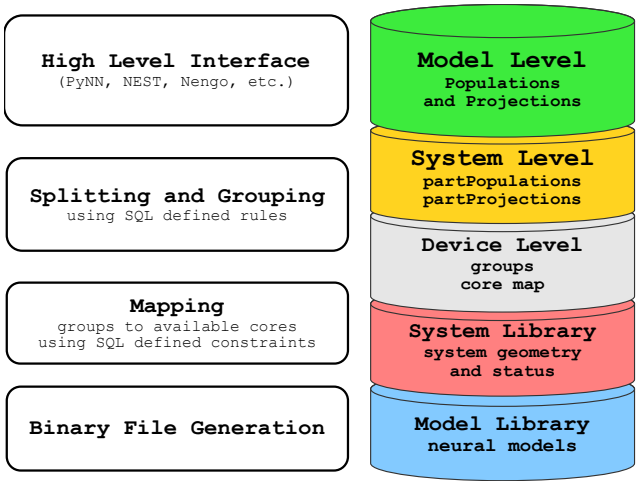

Figure 4: PACMAN structure

layer needs to reflect the availability of a variety of models (neural, synaptic, plasticity), their allocation on the system, and the configuration of the connections between neurons, in order to transform the high-level representation into an on-chip representation. It also needs to be able to represent the resources present in the system, both from a software and hardware point of view. This layer then becomes central to any model-to-machine interaction: at boot time for configuring the system, at runtime for real-time interaction and for data analysis. Therefore it needs to be efficient, as the potential of a real-time configurable device is wasted if the time or complexity to configure and interact with it becomes unmanageable. It also needs to scale up to configure networks with millions of neurons and billions of connections. Finally, it must be compatible with different programming languages or modelling front-ends and approaches to leverage the hardware reconfigurability fully.

We represent information at the *Population* (group of neurons) and *Projection* (bundle of single connections between Populations) level, as most of the languages considered use this representation natively [11] [44] [20]. This greatly simplifies the allocation, mapping and generation processes as against flattening out the whole network at a neuron level, using hierarchical information to effect the translation from the model and place it on the device. Atomic reconfigurability (single neurons and synapses) is maintained, but is pushed to the data file generation step in a way which can easily be ported on-chip and parallelised. The 2-stage routing/lookup system lets the router deal only with Populations, greatly saving routing entries and lowering the complexity through interval routing [47] (where a routing interval can be considered as either the set of routing keys in a Population or in a core, depending on the granularity of the mapping algorithm). This approach simplifies reconfigurability both for arbitrary connectivity between Populations and for easy integration of new cell types constituting Populations. To maximise system scalability, we choose to represent all the data of this layer in SQL format, taking advantage of a "language agnostic" representation which can be easily be accessed from different languages with high performance database technologies. The following sections explain the structure and function of this layer by presenting an im-

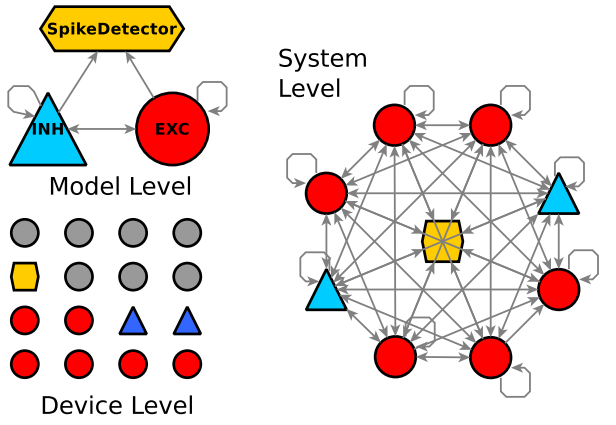

Figure 5: Example network composed of 2 Populations and a Spike Detector interconnected

plementation which is able to translate models from different front-ends and drive data structure generation.

4.1 General Structure

The function of PACMAN - the PArtitioning and Configuration MANager, is to transform the high-level representation of a neural network into a physical on-chip implementation. PACMAN is based on a database that holds three representations of the neural network (fig.4): **Model Level**, the network as specified in the high-level language (PyNN, Nengo, etc.); **System Level**, the network as partitioned into *partPopulations* that can be fit into a single computing core, Projections, probes and inputs being split accordingly; and **Device Level**, a map linking groups of *partPopulations* that can fit into a single core with a particular core (identified by its coordinates) in the system. These representations are specific for every network instantiated, while the *Model and System Libraries* are imported to represent information about the system resources and geometry and the models available. Such translations enable the network to be mapped and deployed on the SpiNNaker system, by generating the binaries needed to configure the simulation components and topology, implementing the hierarchical approach described in figure 3. PACMAN itself (fig.4) is divided into 4 different steps:

Splitting, responsible for splitting neural *Populations* which will not fit in a single core (because of memory or computational complexity limitations) into *partPopulations* that will fit in a core.

Grouping, responsible for collating *partPopulations* which can be run using the same application code in order to fit more of them onto a single core. The result of this operation is a *group*, a collection of neurons which may belong to one or more *partPopulations* and that can fit in a single core.

Mapping, responsible for allocating neural groups to processors and calculating routing. At this stage the model is placed onto the device.

Binary file generation, which creates the actual data binaries from the partitioned and mapped network.

Figure 5 illustrates an example, showing a common scenario where one excitatory (red circle) and one inhibitory (blue triangle) population of neurons is interconnected [5]. A spike detector device is placed to record spike activity. PACMAN splits Populations into *partPopulations* according to their size (in the example the result is 6 excitatory and 2 inhibitory *partPopulations*), and *Projections* between them into *partProjections*, maintaining connectivity unaltered. Finally, it maps *partPopulations* to physical cores at the Device level.

Rules for Splitting and Grouping can be defined in SQL format so as to represent arbitrary rules, which can be easily incorporated in the framework. For example, splitting can be done on the basis of the maximum number of a certain neural type a core can model in real time, while grouping can be applied to neurons using the same neural/plasticity model and having compatible mapping constraints. This information can be passed to the Mapper stage along with model-specific data provided by the high-level generation tool, giving it all the information needed to generate each portion of the network locally. The Mapping stage can take into account constraints, also defined in SQL on a Population basis. This makes interaction with external tools and user-entered constraints simple; if none are specified, groups are assigned to available cores on a first-come first-serve basis. The *System Library* (fig. 4) represents available resources of the system, holding a representation of the system size, geometry and functional status so to map around malfunctioning resources. This process can also differentiate between multiple users accessing separate parts of the machine independently. When a user requests resources, PACMAN will grant only chips allocated to him.

Output from the Mapper is a hierarchical physical description of the entire network. At this point the description still contains abstract objects rather than single neurons or synapses. The mapper organises this information in a way such that binary file generation can be easily parallelised, evaluating the table produced by the mapper core by core. This in turn passed to an Object File generator (which for the moment resides on the Host but could eventually be migrated to an on-SpiNNaker implementation) which flattens the network and generates the actual data binaries for the system. The neural and connectivity structures are computed by retrieving the translation, size and position of the parameters in the neural structure from the Model Library. Parameters can be defined as single values, random distributions or lists explicitly defining the parameter values for each neuron or synapse.

4.2 Introducing new types

The previous section presented how cells can be divided into populations and placed on the system according to their connectivity. Within this framework, new neural, synaptic, plasticity and connectors can easily be implemented. All these elements have a representation in the *Model Library* which contains the translation methods for those objects into SpiNNaker data structure. Using this approach, new models based on the event-driven API can easily be incorporated in a neural application framework [43] and generated using the approach described. PACMAN can be used to specify the translations (order, data types, size) of the parameters describing the neural model. This makes integration of new models onto the machine rapid, exploiting the ARM cores' reconfigurability [18].

4.3 Compatible Frontends

Typically dedicated hardware platforms have their own proprietary font-end [36], or offer compatibility with a single standard one [6] [14] or don't address compatibility with a standard description language [3] [8]. We have proposed an intermediate model-to-system translation layer which effectively decouples the front-end model language from the hardware back-end configuration. As a result of this operation it is possible to plug different front-ends, analysis or representation tools to the system by interfacing with PACMAN, as the process of configuring and compiling the data structures is the same regardless of the language used. The Population/Projection abstraction is already built into many of the front-ends considered, making the integration seamless. The *Model Library* stores the neuron, synapse, plasticity and connector models, containing language-neutral translation methods for the models available on the system. Most of the tools also make native use of Python or Python bindings [13] [11] [1] [44] [25], motivating the choice of Python as an implementation language.

PyNN: as an emerging standard, PyNN is a central tool that can be used to exchange and validate models and results between different groups or compare neuromorphic hardware [14] [36] [6]. We implement the High Level API as a natural extension of the *Population* and *Projection* objects used in the language. *OneToOne*, *AllToAll*, *FixedProbability* and *FromList* connectors are supported (which subsumes the remaining connector types, e.g. *DistanceDependent*). As discussed PyNN supports standard cells and plasticity methods; we extended it to incorporate the Izhikevich neural model and to test novel plasticity algorithms [10].

Nengo: in contrast to all the other front-ends considered, Nengo [44] is specific to the Neural Engineering Framework, which computes connection and weights between groups of neurons to achieve a desired neural computation [12]. It demonstrates the flexibility of the system, in that it proves possible to translate a model from Nengo to PACMAN and generate the correct structures for the model, by implementing an *encoding* and a *decoding* cell type [18].

NEST: a preliminary NEST plugin based on PyNEST [13] has been developed which can be used to create neural populations and connections. The *Device* object in NEST nicely map to a generic application core doing parallel data collection and analysis rather than neural simulation.

Different type of tools can also been integrated within the current framework:

I/O mapping: other than being used at a model generation time, PACMAN can be used to translate information flowing from/to the system at run time. It also makes the mapping of AER devices [30] in the system manageable as another system resource. This lets the end user work exclusively with the model level, PACMAN managing the translation to the Device level when sending/retrieving information to specific populations or neurons when, for example, interfacing with a data visualisation software or with a peripheral.

Network Analysis tools: it is possible to consider *Populations* and *Projections* (or their *part* versions) as nodes and edges of a graph. A weight can then be introduced on the edges, representing the connection density between two populations. It is also possible to represent the Device level in this way, considering cores and routes between cores as nodes and edges, and using the connection density metric to optimize the placement of the cores on the system for example by clustering the most connected cores or by using existing graph representation/analysis tools (as Networkx or GraphML). Those can directly be interfaced with the Mapping stage at SQL level in order to improve placement of the model on the device.

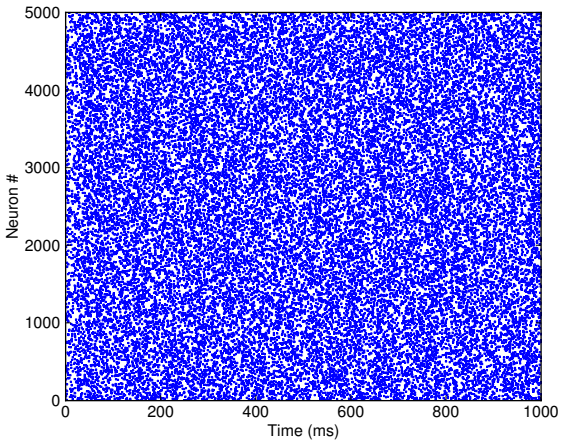

Figure 6: Example Benchmark Network Dynamics

5. RESULTS

We present a set of models having different structures and characteristics. All models run in real-time on a 2x2-chip SpiNNaker board; each core models 500 LIF neurons; all models are written in PyNN. Results have been obtained running PACMAN on an Intel Core 2 Duo T6600 with 4Gb of RAM. We first present results for the configuration of existing SpiNNaker machines (4 chip boards equipped with a total of 72 cores) and we then explore the challenges of configuring larger machines.

5.1 4 chip board results

5.1.1 Example Benchmark Network

The network presented as an example in section 4 (fig. 5) can be used to study the balance of excitation and inhibition in a neural network, and is commonly considered a spiking neural network benchmark [5], as it can easily be scaled up while maintaining the network dynamics intact. The model [48] consists of a network of excitatory and inhibitory neurons, connected via current-based first order synapses (injected current with instantaneous rise and exponential decay) with a 2% probability of interconnection. Figure 6 presents the dynamics for a 5000-neuron instantiation of the network: neurons are driven by their internal dynamics and settle to a state in which they fire in the 5-10Hz range. Figures 9 and 8 presents the performance of PACMAN with different network sizes. This network represents a worst-case scenario, as every partPopulation (and consequently every core in the system) is interconnected and every partProjection is probabilistic. The overhead of creating partProjections is maximal as they are then utilised with only 2% connection density.

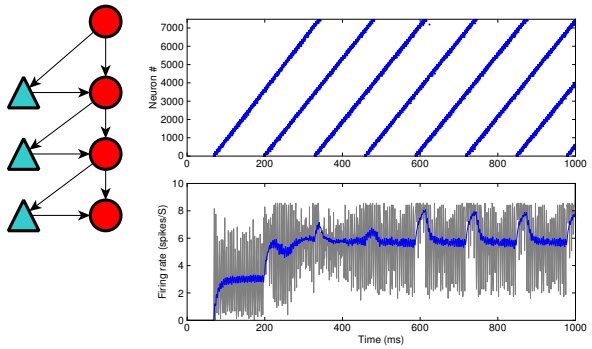

Figure 7: Results from a synfire chain of 60 nodes of 100+25 neurons each

5.1.2 *Synfire Chain*

Synfire chains are well-studied systems of accurate signal propagation in networks of spiking neurons [48] and they offer an ideal mapping benchmark, since scaling the number of units while keeping the network dynamics intact is a relatively easy task. As the network is scaled up the interconnection probability is scaled down so as not to change the dynamics [6]. Figure 7 shows the general connectivity pattern of the synfire chain: each layer is composed of a population of excitatory neurons (red circles) and a population of inhibitory neurons (blue triangles); each excitatory population is interconnected with both the inhibitory and excitatory population in the next layer in a feed-forward inhibition fashion [29] to make it stable; the inhibitory population suppresses the corresponding excitatory population. Figure 7 presents the results for a run of a synfire chain composed of 60 nodes of 100 excitatory + 25 inhibitory neurons each. The first population stimulates the single propagation through the chain. Figure 9 summarises build times for different nodes/network sizes. The placing process (splitting, grouping and mapping populations and projections to the system) is fast for every experiment, with a maximum in Synfire #2 since it contains more nodes and populations.

5.1.3 *Randomly Connected Network*

The previous two examples have dealt with very different interconnectivity patterns: in the first any given neuron has finite probability to connect to any other neuron in the network. In the second there is a fixed feed-forward connection topology, where each population projects to the next level or receives projections from the inhibitory cells in the same layer. In order to study intermediate classes of topologies we introduce a network where populations are randomly interconnected; neurons are injected with random currents so to sustain activity. While neural dynamics are of limited interest, this model can be used to explore different mappings on the system as the number of populations and their interconnections varies, and it can, for instance, represent how signals from a group of functionally specialized neural populations are integrated [45]. This network topology takes advantage of fewer (but denser) partProjections between cores and can therefore be considered as an intermediate case. Results in figure 8 and 9 demonstrate that as the indexing overhead is reduced (few partProjections) and

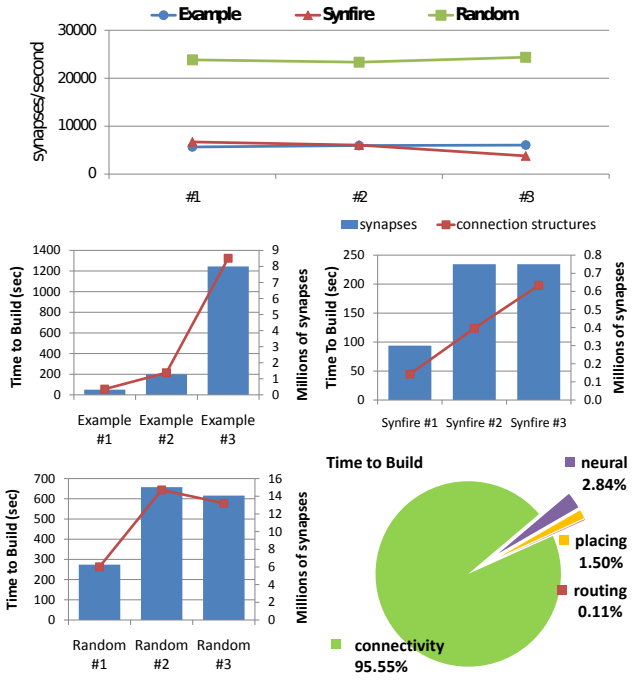

Figure 8: 2x2 results analysis

the probability of synaptic connection is high (50%, making dense Projections), synapse generation is considerably faster than the Example Benchmark Network. It is worth noting that for a given model topology the number of synapses created per second is constant (see fig. 9), leading to a linear scaling up of execution time when adding more synapses.

5.2 PACMAN for larger systems

In the previous sections we have used an approach for configuring 2x2 systems that shows that the limiting factor for large systems is the generation of complex data structures (see fig. 8) that need to be indexed by every core in every chip. We can use the same approach to test bigger systems without simulating the data structure creation. As we have chosen models that maintain their dynamics under scaling, we can simply increase the number of neurons per population so as to fill a much larger system. System size and geometry can be easily modified by adding entries in the *Model Library* representing new chips: PACMAN will then consider them in the mapping phase as available resources. In order to test how the population-based placing approach works on bigger systems, we simulate a network of 16x16 SpiNNaker chips; assuming 16 cores available for neural modelling per chip we model a total of 4,096 cores available for neural simulation.

5.2.1 *Results*

Figure 10 presents build times for each PACMAN stage and model specifications. As in the previous section, cores model 500 LIF neurons each. The Splitting stage is the one which is in charge of re-indexing Projections into partProjections, and as their number increases the process takes longer, as does the routing which needs to route such partProjections. This is evident from last plot: the time to build is directly proportional to the number of partProjections in

	Model Specifications			PACMAN			
	neurons	synapses	cores	placing	routing	connectivity	neural
Example Network #1	4,000	320,000	8	1.98	0.04	56.43	1.56
Example Network #2	8,000	1,280,000	16	2.13	0.14	215	3.22
Example Network #3	20,000	8,000,000	40	2.46	0.91	1323	9.6
Synfire #1 40 nodes, 250 neurons/node	10,000	292,500	20	2.3	0.14	44.7	3.4
Synfire #2 100 nodes, 250 neurons/node	25,000	742,500	50	2.5	0.37	123.79	8.35
Synfire #3 10 nodes, 2500 neurons/node	25,000	675,000	50	2.36	0.16	198	8.34
Random Network #1 50 populations, 5 projections	12,500	6,262,500	25	2.17	0.17	262.9	4.25
Random Network #2 120 populations, 5 projections	30,000	15,030,000	60	2.29	0.43	643.9	10.29
Random Network #3 50 populations, 10 projections	12,500	14,075,000	25	2.28	0.3	577.7	4.26

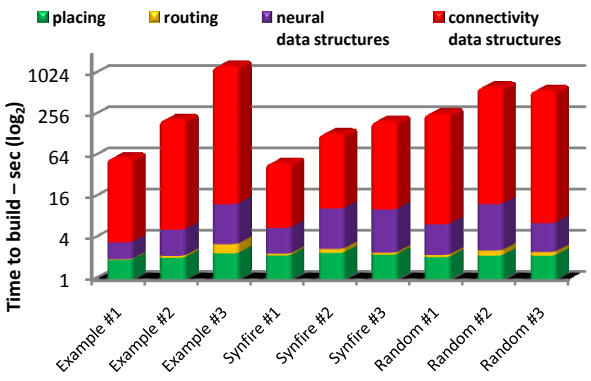

Figure 9: Time building results for models run on a 2x2 system

the system. As the previous section discusses, the example benchmark network is worst-case, both for the placing strategy and for the system itself. It is no surprise then that, regardless of the neuron and synapse count, this is the model that takes most time to build. A model with fewer but more dense Projections (such as the Random Network) takes less time to be built despite having 4× more neurons and twice as many synapses as the Example Network. The Synfire Chain Network is the fastest to build, as it has fewer Projections/synapses, but stresses the Grouper, since inhibitory Populations can be grouped in pairs together in a core. The Mapping stage is trivial in all cases as no constraints are provided, and hence it just needs to allocate groups to cores serially. This results show the potential of the system with different connectivity patterns, and hint at the ones that are easier to model on the machine.

	cores	part populations	part projections	splitting	grouping	mapping	routing
Example Network 500K neurons 5.000M synapses (2%interconn probability)	1,000	1,000	1,000,000	337	24	0.66	1378
Synfire Chain 1600 nodes 1000+250 neurons each 60M synapses	4,000	4,800	12,792	17	53	0.91	40
Random Network 2000 Populations 1000 neurons/10 Projections 2M neurons 9.000M synapses	4,000	4,000	80,000	72	38	0.92	213

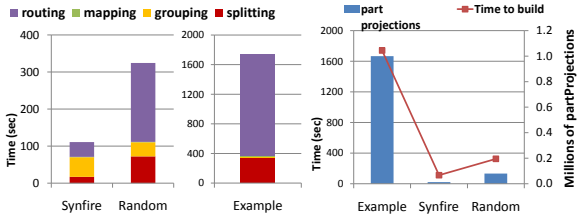

Figure 10: Results from models on a 16x16 system

6. DISCUSSION

The software infrastructure presented can be used to configure parallel digital hardware system for simulation of spiking neural networks by exploiting its arbitrary remapping capability, both on a single computational node and a connectivity level. The approach takes advantage of hierarchical information embedded in the model at a neural *Population* level to drive the configuration of the system in an efficient way, by deferring the data file compilation stage until after the resource mapping on the system, simplifying its parallelisation (and hence the generation of the configuration data structures on the machine itself). This is a key result of the paper: by using a hierarchical approach and considering Populations and Projections rather than flattening single neurons and synapses all the mapping and placing algorithms have lower complexity, while still maintaining a full low-level representation at the device level. The results presented in the previous section show interesting aspects of the problem and of the approach proposed to solve it. It is evident that for the scale of systems considered in this section the most complex task is compiling the data structures, especially for synaptic data. Indeed, the allocation and mapping on this system sizes is trivial, as is routing, since they work on a core/partPopulation basis rather than on a single neuron/synapse level. We have identified the bottleneck of the actual system - the generation of complex synaptic structures that must be indexed by every core in every chip. While optimization techniques at a DB level (proper indexing, using prepared statements, partitioning of large tables) or at a Python level (using fast numerical libraries as Numpy) could be used to improve performance, data structure creation is an "embarrassingly parallel" problem (each core needs only to configure is local portion of memory independently) and thus we propose to implement this stage of PACMAN on-chip.

Another key result of this approach is that at the end of the Mapping stage information is already represented in a local, core-based way and can therefore be parallelised and distribute on the machine itself, greatly speeding up the most critical processes and exploiting the system architecture in the configuration phase. At this point each neuron has been allocated to a core, and it is possible to route the network, associating neurons to routing keys, even though the data structures do not as yet exist. Therefore, this part of the process can be done on the host machine, and the hierarchical representation permits the use of interval routing [47], simplifying the problem greatly. Allocating neurons to system resources can be done by sending the Population parameters to the corresponding core. Each core can then receive information about the incoming partProjections (parameters, range of routing keys considered, connectivity pattern) from PACMAN, and initialise, compile, organize and index its own synaptic data structures in SDRAM, having all the necessary information represented locally. Hence the host

system only needs to send abstract information to the machine which self-configures by populating its local portion of memory.

The proposed approach has already been used to map models which explore new plasticity models [10] or structural biological models of the cortex micro-circuitry [42]. Creating new neural applications which can then be seamlessly used as computational resource nodes is an approach that greatly exploits the reconfigurability of a digital architecture [43] and can be used to introduce new computational frameworks rapidly for the architecture [18]. For general purpose use, a user will in future be able to create his own neural model specifying dynamical equations as done with Brian [1]. As the system is completely event-driven, this neural model creation tool will map dynamics to events [40], making use of code generation techniques [23] either from the modelling language or from a standard XML format [21].

7. CONCLUSIONS

We have presented a hierarchical approach for configuring large parallel systems. The layer proposed effectively decouples the model, written in a user-selected front-end language, and the device level, abstracting the hardware to the end user and letting non-hardware experts exploit the reconfigurability of the architecture in a seamless way. We have tested the proposed method by configuring the current generation of SpiNNaker systems in a variety of networks of thousands of neurons and millions of synapses. We indicate the current limitations and propose solutions, discussing the capabilities of an approach which promises to be able to scale up to configure systems of millions of neurons and thousands of processors. A system which combines off-hardware hierarchical decomposition of the application with on-board generation of the data binaries is a compelling and efficient model for very large-scale parallel applications that maximises the utilisation of available hardware resources. This approach is a natural one whenever the application, like networks of neurons, has its own internal structure which naturally suggests the on-hardware mapping.

8. ACKNOWLEDGEMENTS

We would like to thank Chris Eliasmith and Terry Stewart for their contribution in the Nengo/SpiNNaker interface. The SpiNNaker project is supported by the Engineering and Physical Science Research Council (EPSRC), grant EP/4015740/1, and also by ARM and Silistix. We appreciate the support of these sponsors and industrial partners.

9. REFERENCES

[1] http://www.briansimulator.org/.

[2] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha. The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 63:1–63:12, New York, NY, USA, 2009. ACM.

[3] M. Bhuiyan, V. Pallipuram, and M. Smith. Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[4] J. M. Bower and D. Beeman. The book of GENESIS (2nd ed.): exploring realistic neural models with the GEneral NEural SImulation System. 1998.

[5] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. Bower, M. Diesmann, A. Morrison, P. Goodman, F. Harris, and Others. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398, Dec. 2007.

[6] D. Brüderle, M. Petrovici, B. Vogginger, M. Ehrlich, T. Pfeil, S. Millner, A. Grübl, K. Wendt, E. Müller, M. Schwartz, and Others. A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. *Biological cybernetics*, 104(4-5):1–34, May 2011.

[7] T. Carnevale. Neuron simulation environment. *Scholarpedia*, 2(6):1378, 2007.

[8] A. Cassidy, A. Andreou, and J. Georgiou. Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis. In *Information Sciences and Systems (CISS), 2011 45th Annual Conference on*, pages 1–6. IEEE, 2011.

[9] T. Choi, B. Shi, and K. Boahen. An ON-OFF Orientation Selective Address Event Representation Image Transceiver Chip. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 51(2):342–353, Feb. 2004.

[10] S. Davies, F. Galluppi, and A. Rast. A forecast-based STDP rule suitable for neuromorphic implementation. *Neural Networks*, 2012.

[11] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: A Common Interface for Neuronal Network Simulators. *Front Neuroinformatics*, 2:11, 2008.

[12] C. Eliasmith and C. H. Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press, Cambridge, MA, 2003.

[13] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig. PyNEST: A Convenient Interface to the NEST Simulator. *Front Neuroinformatics*, 2:12, 2008.

[14] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk. NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 137–144. IEEE, July 2009.

[15] S. Furber and A. Brown. Biologically-Inspired Massively-Parallel Architectures - Computing Beyond a Million Processors. In *Proceedings of the 2009 Ninth International Conference on Application of Concurrency to System Design*, ACSD '09, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society.

[16] S. B. Furber, S. Temple, and A. D. Brown. High-Performance Computing for Systems of Spiking Neurons. *The AISB06 workshop on GC5: Architecture of Brain and Mind*, 2006.

[17] S. B. Furber, S. Temple, and A. D. Brown. On-chip and Inter-Chip Networks for Modelling Large-Scale Neural Systems, 2006.

[18] F. Galluppi, S. Davies, T. Stewart, C. Eliasmith, and

S. Furber. Real Time On-Chip Implementation of Dynamical Systems with Spiking Neurons. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*. IEEE, 2012.

[19] F. Galluppi, A. Rast, S. Davies, and S. Furber. A general-purpose model translation system for a universal neural chip. In *Neural Information Processing. Theory and Algorithms*, pages 58–65. Springer, 2010.

[20] O. Gewaltig, M. Diesmann, and M.-O. Gewaltig. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[21] N. H. Goddard, M. Hucka, F. Howell, H. Cornelis, K. Shankar, and D. Beeman. Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philos Trans R Soc Lond B Biol Sci*, 356(1412):1209–1228, Aug. 2001.

[22] D. Goodman. Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2, 2008.

[23] D. Goodman. Code generation: a strategy for neural network simulators. *Neuroinformatics*, 8(3):183–96, Oct. 2010.

[24] M. Hanke and Y. Halchenko. Neuroscience runs on GNU/Linux. *Frontiers in neuroinformatics*, 5, 2011.

[25] M. L. Hines and N. T. Carnevale. The NEURON Simulation Environment. *Neural Computation*, 9(6):1179–1209, Aug. 1997.

[26] K. M. Hynna and K. Boahen. Neuronal Ion-Channel Dynamics in Silicon. In *Proc. 2006 Int'l Symp. Circuits and Systems (ISCAS 2006)*, pages 3614–3617, 2006.

[27] G. Indiveri, E. Chicca, and R. Douglas. A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *IEEE Transactions on Neural Networks*, 17:211–221, 2006.

[28] C. Johansson and A. Lansner. Towards cortex sized artificial neural systems. *Neural Networks*, 20(1):48–61, 2007.

[29] J. Kremkow, A. Aertsen, and A. Kumar. Gating of Signal Propagation in Spiking Neural Networks by Balanced and Correlated Excitation and Inhibition. *Journal of Neuroscience*, 30(47):15760–15768, 2010.

[30] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Silviotti, and D. Gillespie. Silicon Auditory Processors as Computer Peripherals. *IEEE Transactions on Neural Networks*, 4(3):523–528, May 1993.

[31] W. Maass, T. Natschläger, T. U. Graz, and H. Markram. On the computational power of circuits of spiking neurons. *J. of Physiology (Paris*, 2005, 2003.

[32] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for Large-Scale Implementations of Spiking Neural Networks on FPGAs. *Neurocomputing*, 71, Dec. 2007.

[33] H. Markram. The blue brain project. *Nat Rev Neurosci.*, 7:153–160, 2006.

[34] P. A. Merolla, J. V. Arthur, B. E. Shi, and K. A. Boahen. Expandable Networks for Neuromorphic Chips. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(2):301–311, Feb. 2007.

[35] A. Morrison, C. Mehring, T. Geisel, A. D. Aertsen, and M. Diesmann. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural computation*, 17(8):1776–801, Aug. 2005.

[36] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5–6):791–800, 2009.

[37] C. Patterson, J. Garside, E. Painkras, S. Temple, L. Plana, J. Navaridas, T. Sharp, and S. Furber. Scalable Communications for a Million-Core Neural Processing Architecture. *Accepted by the Journal of Distributed Computing*.

[38] L. Plana, S. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design & Test of Computers*, 24(5):454–463, 2007.

[39] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M. Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. *Euro-Par 2007 parallel processing*, pages 672–681, 2007.

[40] A. Rast, F. Galluppi, X. Jin, and S. Furber. The leaky integrate-and-fire neuron: A platform for synaptic model exploration on the spinnaker chip. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[41] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proc. 2010 Int'l Symp. Circuits and Systems (ISCAS 2010)*, pages 1947–1950, 2010.

[42] T. Sharp, F. Galluppi, A. Rast, and S. Furber. Power-efficient simulation of detailed cortical microcircuits on SpiNNaker. *Journal of Neuroscience Methods*, 2012.

[43] T. Sharp, L. A. Plana, F. Galluppi, and S. Furber. Event-Driven Simulation of Arbitrary Spiking Neural Networks on SpiNNaker. In *ICONIP 2011*, volume 2011, pages 424–430. Springer, 2011.

[44] T. C. Stewart, B. Tripp, and C. Eliasmith. Python scripting in the Nengo simulator. *Frontiers in Neuroinformatics*, 3(0), 2009.

[45] G. Tononi, G. M. Edelman, and O. Sporns. Complexity and coherency: integrating information in the brain. *Trends in cognivitve sciences*, 2(12):474–484, 1998.

[46] D. Vainbrand and R. Ginosar. Scalable network-on-chip architecture for configurable neural networks. *Microprocessors and Microsystems*, 35(2):152–166, Mar. 2011.

[47] J. Van Leeuwen. Interval Routing. *The Computer Journal*, 30(4):298–307, Apr. 1987.

[48] T. P. Vogels and L. F. Abbott. Signal propagation and logic gating in networks of integrate-and-fire neurons. *J Neurosci*, 25(46):10786–10795, Nov. 2005.

[49] R. J. Vogelstein, U. Mallik, E. Culurciello, G. Cauwenberghs, and R. Etienne-Cummings. A multichip neuromorphic system for spike-based visual information processing. *Neural computation*, 19(9):2281–300, Sept. 2007.