Mapping Spiking Neural Networks on Multi-Core Neuromorphic Platforms: Problem Formulation and Performance Analysis

Francesco Barchi, Gianvito Urgese, Enrico Macii, and Andrea Acquaviva

Politecnico di Torino, DAUIN,
Corso Duca degli Abruzzi, 24, 10129 Torino, Italy
{francesco.barchi,gianvito.urgese,
enrico.macii,andrea.acquaviva}@polito.it

Abstract. In this paper, we propose a methodology for efficiently mapping concurrent applications over a globally asynchronous locally synchronous (GALS) multi-core architecture designed for simulating a Spiking Neural Network (SNN) in real-time. The problem of neuron-to-core mapping is relevant as a non-efficient allocation may impact real-time and reliability of the SNN execution. We designed a task placement pipeline capable of analysing the network of neurons and producing a placement configuration that enables a reduction of communication between computational nodes. We compared four Placement techniques by evaluating the overall post-placement synaptic elongation that represents the cumulative distance that spikes generated by neurons running on a core have to travel to reach their destination core. Results point out that mapping solutions taking into account the directionality of the SNN application provide an improved placement with respect to the random placement.

Keywords: Graph Mapping, Multicore Neuromorphic Architectures, Spiking Neural Netwoks

1 Introduction

Finding the best way to map tasks to processor cores in multi and many-core systems is a relevant optimisation problem, with significant impact on application reliability, performance, and energy consumption.

The solution of this particular problem in many cases can only be computed employing heuristic methodologies capable of providing approximated or sub-optimal solutions.

The task placement problem is common in many fields of applications that go from the mapping of parallel applications on stream-oriented MPSoCs [23] to the placement of virtual machines in cloud data centres with parallel nodes [17]. Programming such architectures efficiently is a challenge because numerous hardware characteristics have to be taken into account, especially the memory hierarchy. One appealing idea to improve the performance of parallel applications

is to decrease their communication costs by matching the communication pattern to the underlying hardware architecture. Such a method can be performed with the design of a strategy capable of partitioning the main application in several independent tasks with computation/communication load compatible with the capability of the parallel cores available on the architecture.

A good example of a solution for this problem has been proposed in [23] where they defined a placement system that split the full original application in a directed acyclic task graph where each node is an independent atomic task that communicates with the other nodes in discrete time. Moreover, they considered the number and capability of available cores and the communication infrastructure of the targeted platform for reducing the placement problem on N tasks to M processors.

In the Neuromorphic domain, we explored the task placement problem in the case of a globally asynchronous locally synchronous (GALS) multicore architecture called SpiNNaker. However, the same type of analysis can be customised for Intel Loihi [4], IBM TrueNorth [2] and SpiNNaker2 [15], representing the future of the neuromorphic multi-core platforms, for discovery what are the best practices to be adopted for efficiently mapping running highly parallel tasks.

SpiNNaker has been designed mainly for running neuromorphic applications. Here, tasks to be executed are physical neuron models running in parallel on the platform and communicating through messages. These messages represent signals, called spikes, which biological neurons exchange through their physical (neural) connections inside the brain.

The overall purpose of this application is to execute a Spiking Neural Network (SNN) in real-time. In this case, real-time means that the timings of the spikes generated by the neurons should be compliant with the one of the real human brain. Thus opening the way for the use of neuromorphic platforms to interface external physical systems and elaborates their signals (e.g. images, sounds) in the same way as the brain does. Being the neurons executed as concurrent tasks by the general purpose cores, how to efficiently mapping neurons-to-cores is an issue that must be addressed for optimising the communication between cores.

Generalising, the problem we faced concerns the mapping of a large number of light parallel tasks with intensive communication to a many-core architecture. A non-efficient communication, in the specific case of SNN execution, may impact real-time capabilities as well as the reliability of the application. Indeed, spikes can be lost due to congestion problems. In general, a possible approach to face the mapping problem is to model the tasks and their communication as a graph to be mapped over the underlying hardware architecture, represented by another graph.

Sugiarto et al. [26] presented an approach for improving the overall performance of general-purpose applications running as a task graph on the same many-core neuromorphic supercomputer. Whereas in a recent paper, we have used the cortical microcircuit application as a test case for demonstrating that an enhanced partitioning and placement system studied for the SNN topology

can produce a more reliable and stable configuration for the simulation on the SpiNNaker system [27, 28].

In this document, we present a methodology for mapping a task graph representing the SNN computation on a multi-chip many-core architecture with communication awareness. To achieve this target, we designed a task mapping framework capable of analysing the network of neurons to find a configuration with the goal of reducing the communication between computational nodes. The neuron-to-core mapping problem has been formalised as a problem of minimisation of synaptic elongation. Intuitively, this metric represents the cumulative distance that spikes generated by neurons running on a specific core have to travel to reach their destination core.

The framework starts by extracting a graph of independent processes from a neural network description. In the case of SNN, the direction of a communication path is also to be represented using a directed graph. On the platform side, the interconnect structure is described as a graph where nodes represent on-chip cores while edges represent physical communication links between them. In this way, we formalised a neuron-to-core mapping as a graph-matching problem solvable through the exploitation of various algorithms available in the literature. The specific formulation we devised for SNN mapping takes into account the typical organisation of these type of neural networks into neuron populations, sharing similar characteristics as well as the neuron model.

The results obtained by comparing four mapping algorithms points out and quantify the relevance of the communication direction information to achieve a better mapping if compared with non-directional algorithms.

2 Background

In this section, we will introduce the application and the MCSoC board selected as a target for demonstrating the advantages of adopting our task-placement communication aware framework.

2.1 Target Application: Neural Network Simulation

Spiking Neural Network (SNN) is a particular neural model used by neuroscientist for simulating biologically plausible brain activity. Two of the most adopted neuron models are the *leaky Integrate and Fire (IF)* [1] and *Izhikevich (IZK)* [10], because they can ensure a plausible picture of the biological behaviours with reduced computational costs. During SNN simulations neurons and their synapses are modelled as differential equations capable of emulating the behaviours observed in biological networks [16]. An SNN can be described as a graph where each vertex is called *Population* containing a homogeneous group of neurons sharing the same model and parameters. Whereas, each edge (*Projection*) represents the rule used to generate synaptic connections between the neurons of two *Populations*. Using PyNN [5] scientist can describe many neurons/synapses

models and configurations that can be exploited on different backends such as software simulators and neuromorphic platforms.

Using this SNN description system, Van Albada et al. [29] designed an SNN application implementing the cell-type specific cortical microcircuit (CM) model created by Potjans et al. [20]. Then they simulated this SNN on a neuromorphic multi-chip many-core platform called SpiNNaker [7] using the standard application partitioning and placement system for setting up the simulation on the board.

2.2 Target Architecture: Neuromorphic MPSoCs Board

For validating our placement methodology framework we took as target a GALS Neuromorphic many-core architecture and used its native application such as an example case. We used the SpiNNaker architecture, which is a general-purpose real-time many-core platform mainly used for simulating neural networks following an event-driven computational approach [7]. This system mimics the features of a biological neural network through the implementation of several features:

- Native parallelism: Each biological neuron is a fundamental computational element within a massively parallel system. Likewise, SpiNNaker uses parallel computation.
- Spiking communications: In biology, neurons communicate through spikes. The SpiNNaker architecture uses source-based Address Event Representation (AER) packets to transmit the equivalent of neural signals (i.e. action potentials) [21]. Each AER packet identifies the event source through an addressing scheme.
- Event-driven behaviour: Neurons are very power efficient, and consume much less power than other modern hardware, in fact to reduce power consumption, the hardware is put into "idle" state until an interrupt event doesn't trigger an action [11].
- Distributed memory: In biology, neurons use only local information to process incoming stimuli. The SpiNNaker architecture features a hierarchy of memories: memory local to each of the cores and an SDRAM local to each chip.

The SpiNNaker chip (Figure 1) has 18 ARM 968 cores running at 200MHz with no floating point units[1], a full customized router for intra/inter-chip communications, and an SDRAM external to the chip and accessible through the PL340 interface [8]. Each core of a chip can access three four memory levels: i) a 64KB Tightly Coupled Memory (TCM) that is part of each ARM core. It is divided into ITCM containing instructions and DTCM containing application data. ii) a 32KB System RAM integrated into the chip and shared between all the core. iii) a 128MB SDRAM shared between all cores of a chip. iv) a 32KB System ROM shared between all processors that contains the bootstrap software.

[1] the SpiNNaker simulator applies a mechanism of rescaling that allows working with only integers, even if in the equations of the neural models are in the domains of the Real numbers

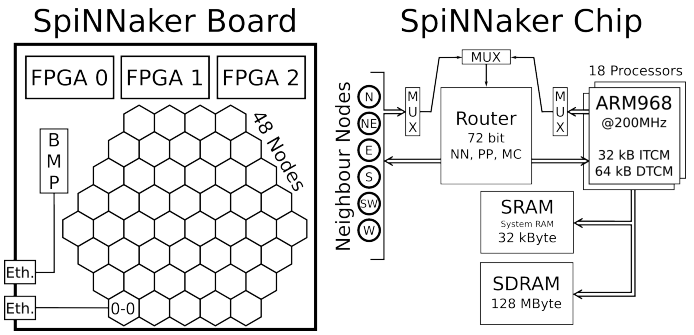


Fig. 1: **SpiNNaker Architecture.** On the left the Board with 48 chips connected in exagonal mesh. On the right the chip architecture

The SpiNNaker system is built with boards of 48 chips (Figure 1) interconnected for forming a toroidal shaped triangular mesh where each chip is connected to six neighbours chips. Each router is in charge to dispatch packets along intra and inter-chip cores and it is designed for managing transmission of four types of packets:

- Multi-Cast (MC) packets are used for reaching many cores across the board. They are widely used during neural simulations for spreading neural potentials to multiple destinations (emulating synapses potential transmission). These packets are routed using a routing table of 1024 entries, stored in a ternary CAM with three values per entry: routing entry, mask and direction. The routing key of a multicast packet is compared with all the entries and then redirected. The length of those packets could be up to 72 bits.
- Point-to-Point (PP) packets are used for reaching an exact core of the board uniquely identified by the coordinates of the belonging chip and its relative number (from 0 to 17). These packets are routed using a dedicate routing table. If the destination is within the local chip, the packet is delivered to the monitor processor. This type of packet can transport a payload of 32 bits over the available 72 bits.
- Nearest Neighbour (NN) packets are used for initialising the board and for implementing a keep-alive mechanism useful for understanding if there are broken links and calculate different paths in this case.
- Fixed Route (FR) packets are used for reaching a fixed destination (the Ethernet controller). The advantage of this type of packet is that it provides 64 bits of payload with 8 bits of overhead only.

Configuration and management of this new neuromorphic architecture need a set of software tools for translating the applications to be executed in executables to be processed on the many available cores. The official software for configuring and running a simulation on the system involves the board-side C/Assembly code and the host-side code, mostly written in Python [22]. We will briefly discuss the Host-Side module currently in charge of partitioning and placing

SNN populations, details on the full configuration software stack can be found in [12]. The current implementation of SNN Partition and Placement Manager computes the partitioning of the graph of the SNN population and calculate a radial placement of the partitioned population on the cores of the board. These two steps are necessary for configuring the board when the application to be executed on the SpiNNaker is the simulation of an SNN represented as a graph of populations of neurons interconnected in a biological-inspired network.

The host side software that allows running simulations, loading modules and managing connections and data transfer is made of five main modules:

- sPyNNaker: a module that provides a wrapper of the PyNN [5] neural network software simulator implementing the neural models (implemented both in python and machine executable .aplx code), population models, connectors and projections. The tool parses all the parameters from the configuration files and translates the PyNN data into populations to be loaded into SpiNNaker.
- SpiNNMachine: implements a set of classes that represents a high level all the features of a SpiNNaker board, there are many classes each one that represents a component (e.g. *Processor*, *SDRAM*, *Router*). The main class, *Machine*, is the software representation of the board with its chips and all their components.
- PACMAN: The Partition and Control Manager performs the partitioning of the SNN graph splitting each Population in a set of vertex (partial-population) and performs the placement of the partial-population on the SpiNNaker processors.
- SpiNNMan: is a tool that implements at a lower level the communication with the board allowing to send and receive messages to it (SDP packets, SCP commands, EIEIO Packets) using UDP protocol. It is widely used by the other modules for load files and data.
- DataSpecification: contains a tool that allows specifying data (synaptic matrices, data structures for doing reports to the host or register spikes) for the neurons in each core.

This full software stack can be combined with a new protocol [25] designed for simplifying configuration and execution of applications by enabling: i) A more efficient generation of data structures during the configuration phase, ii) An on-the-fly reconfiguration of specific parameters, avoiding the re-load of simulation data, and iii) The possibility of embedding alternative computational flows in the applications, allowing users to switch between predefined tasks.

3 Problem Formulation

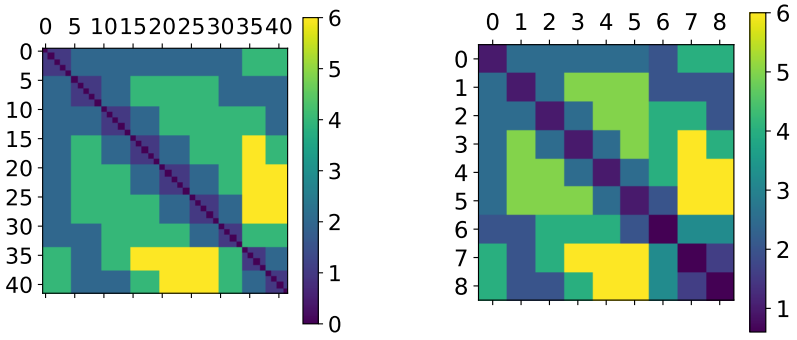
The SNN placement into the neuromorphic architecture can be view as an optimisation problem that involves two graphs: $\mathcal{G}_{\mathcal{N}}$ and $\mathcal{G}_{\mathrm{CPU}}$.

A graph $\mathcal{G} = (V, E, \mathcal{W})$ is a mathematical representation for describing a set of elements V and a set of relations $E \subseteq \{(v_i, v_j) : v_i, v_j \in V\}$ among them.

The elements are called *nodes* of the graph and the relations are called *edges* of the graph. An edge $e_{ij} \in E$ binds two nodes $v_i, v_j \in V$ to each other. A graph can have a $\mathcal{W} : E \to W$ function that associates an edge $e_{ij} \in E$ to a value $w_{ij} \in W$. The value $w_{ij} = \mathcal{W}(e_{ij})$ is called edge weight. A graph can be categorised according to two properties: i) If the nodes on edges form unordered pairs $e_{ij} : \{v_i, v_j\}$ the graph is said *undirected* otherwise it is said *directed* and the nodes on edges form ordered pairs $e_{ij} : (v_i, v_j)$. ii) If the weight set W is empty the graph is said *unweighted*, otherwise it is said *weighed*.

A Spiking Neural Network (SNN) can be represented using a directed and weighted graph called *neuron graph* $\mathcal{G}_\mathcal{N}$. In $\mathcal{G}_\mathcal{N}$ the nodes are the SNN neurons and the edges are the SNN synapses. Taking into account a synapse $e_{ij} : (v_i, v_j)$, the neuron v_i is called pre-synaptic neuron and the neuron v_j is called post-synaptic neuron. The edge weight w_{ij} represents the synapse contribution to injected current into the post-synaptic neuron after a stimulus received by the pre-synaptic neuron and is called synaptic weight.

The neuromorphic architecture can be represented using an undirected and weighed graph, called *target graph* $\mathcal{G}_\mathcal{T}$.



(a) Distance matrix obtained using a fine-grain target graph

(b) Distance matrix obtained using a coarse-grain target graph

Fig. 2: The distance matrix of the same placement area using a fine-grain and a coarse-grain target graph.

The target graph can be more or less detailed. If the graph nodes are the SpiNNaker Chip, we define the target graph as *coarse-grain*. If the graph nodes are the ARM processors, we define the target graph as *fine-grain*.

If $\mathcal{G}_\mathcal{T}$ is coarse-grain, all edges have a weight of 2 and represent the inter-chip communication links. If $\mathcal{G}_\mathcal{T}$ is fine-grain, all edges between two processors located on the same chip have a weight of 1, while all edges between two processors belonging to adjacent chips have a weight of 2.

This choice is determined by the structure of arbiter which feeds the SpiNNaker chip routers. The router has two branches for introducing packets according to their origin: the 18 internal processors and the six neighbouring chips. It

has been demonstrated in [27] that the arbiter does not correctly manage some traffic configurations coming from the six external links. It was therefore decided to disadvantage all inter-chip communications with twice the weight of intra-chip communications. The Fig.2 shows the differences between the fine-grain model and the coarse-grain model through the distance matrices obtained from the graphs of the target nodes.

We can define the placement problem $\Pi : \mathcal{G}_\mathcal{N} \rightarrow \mathcal{G}_\mathcal{T}$ as a minimization problem (1).

$$\underset{f(\pi)}{\text{minimize}} \qquad f : \sum_{e_{ij} \in E_\mathcal{N}} d(\pi(v_i), \pi(v_j)) \tag{1a}$$

$$\text{subject to} \qquad \pi(i) = \pi(j) \rightarrow \mathcal{M}(i) = \mathcal{M}(j),\ i, j \in V_\mathcal{N} \tag{1b}$$

$$|\pi(i) = p| \leq \mathcal{S}(\mathcal{M}(i)),\ i \in V_\mathcal{N}, p \in V_\mathcal{T} \tag{1c}$$

The goal of a placement procedure is to minimise the *overall synaptic stretching* (1a) to reduce the communication along the network nodes. The *synaptic stretching* is the distance between the nodes where two adjacent neurons are placed. Where $\pi : V_\mathcal{N} \rightarrow V_\mathcal{T}$ is the placement rule, $\mathcal{M} : V_\mathcal{N} \rightarrow M$ is the neuron-model association rule and, $\mathcal{S} : M \rightarrow \mathbb{N}$ is the association rule between a neuron model and the maximum number of neurons per node. The constraints of the placement problem are two: i) All neurons mapped into a target node must be of the same model (1b). ii) Each node can simulate only a certain number of neurons, and the quantity depends on the complexity of the neuron model (1c).

3.1 Problem Relaxation

A SNN is almost never described in $\mathcal{G}_\mathcal{N}$ form, due the high complexity in manage all neurons and synapses, but is normally described in terms of Population and Projection. A Population $\mathcal{P}$ is a set of neurons that share the same model and the same properties. A Projection between two Population $\mathcal{P}^{(a)}$ and $\mathcal{P}^{(b)}$ defines a rule for create a set of synapses where the pre-synaptic neurons are in $\mathcal{P}^{(a)}$ and the post-synaptic neurons are in $\mathcal{P}^{(b)}$. We will refer to the *Population-Projection graph* using the notation $\mathrm{G}_\mathcal{P}$.

We can eliminate the two constrains (1b, 1c) redefining the problem Π working from the graph $\mathcal{G}_\mathcal{P}$. The first step is splitting each population $\mathcal{P}^{(i)}$ into a set of partial populations $\left\{\mathcal{P}_1^{(i)}, \mathcal{P}_2^{(i)}, \ldots, \mathcal{P}_z^{(i)}\right\}$. All partial populations must contains at most a number of neurons equal to the maximum number of neurons allowed to be simulated in a target node: $|\mathcal{P}_j^{(i)}| \leq n^{(i)}\ \forall j = 1, \ldots, z$, with $n^{(i)} = \mathcal{S}(\mathcal{M}(\mathcal{P}^{(i)}))$.

In this way we obtain the *partial population graph* $\mathcal{G}_{\mathrm{pp}}$. The edges of the partial population graph are weighed and ordered. Given an edge $e_{ij} \in E_{\mathrm{pp}}$ between two partial population, its weight w_{ij} is equal to the number of synapses shared between the neurons belonging the two partial populations.

We can redefine (1) using the placement rule $\pi : V_{\text{pp}} \rightarrow V_{\mathcal{T}}$ that map a partial population into a processor (2).

$$\underset{f(\pi)}{\text{minimize}} \qquad \sum_{e_{ij} \in E_{\text{pp}}} d(\pi(v_i), \pi(v_j)) * w_{ij} \tag{2a}$$

$$\text{subject to} \qquad |\pi(i) = p| \leq 1,\ i \in V_{\text{pp}}, p \in V_{\mathcal{T}} \tag{2b}$$

In (2a) we modify the cost function to take into account the number of synapses shared between the target nodes. The rule in (2b) describes the single constraint of the problem: a target node may contain only one partial population.

3.2 Graph Partitioning

The partition problem of $\mathcal{G}_{\mathcal{P}}$ can be solved in different ways. In [28] it was treated as a problem of clustering. The provided solution was divided into three step:

- Graph expansion: $\mathcal{G}_{\mathcal{P}} \rightarrow \mathcal{G}_{\mathcal{N}}$
- Spectral clustering: $\mathcal{G}_{\mathcal{N}} \rightarrow \mathbb{R}^{|V_{\mathcal{N}}|}$
- Legalization and clusters fusion: $\mathbb{R}^{|V_{\mathcal{N}}|} \rightarrow \mathcal{G}_{\text{pp}}$.

The first step is to create the neuron graph $\mathcal{G}_{\mathcal{N}}$ by applying the synaptic generation rules defined into the Population-Projections graph $\mathcal{G}_{\mathcal{P}}$. In the second step, a spectral clustering procedure is applied to the neuron graph.

The Spectral Clustering involves the eigendecomposition of a representative matrix of the graph. In the case of $\mathcal{G}_{\mathcal{N}}$, a directed graph, it was used a Laplacian Matrix (3) obtained throught a transition matrix induced by a random walk [3].

$$L = I - \frac{(\Phi^{\frac{1}{2}} P \Phi^{-\frac{1}{2}} + \Phi^{-\frac{1}{2}} P^T \Phi^{\frac{1}{2}})}{2} \tag{3}$$

The results of the Spectral Clustering is the $\mathcal{G}_{\mathcal{N}}$ rapresentation into the eigenspace of $\mathbf{L}$, a space belonging to $\mathbb{R}^{|V_{\mathcal{N}}|}$. The neurons can be clustered into the eigenspace using the KMeans algorithm. After the clustering, a legalisation phase gathers in groups all neurons belonging to the same cluster and the same population. Finally, a second legalisation phase, called Fusion, builds the partial populations putting together the nearby groups of neurons until reach the maximum number of neurons that a processor can simulate.

Other techniques of graph clustering are Multilevel Graph Partitioning and Markov Cluster Algorithm [13, 30]. These techniques, like the Spectral Cluster, was born for undirected graph and their usage should be analysed using different symmetrisation techniques if applied to a directed graph.

4 Placement

As seen in section 3 our goal is placing $\mathcal{G}_\mathcal{N}$ into a set of nodes $\mathcal{G}_\mathcal{T}$. In subsection 3.1 we have relaxed the constraints of the problem separating it into two sub-problems: i) Clustering $\mathcal{G}_\mathcal{N}$ (or partitioning if consider $\mathcal{G}_\mathcal{P}$ as a starting point) into the partial population graph. ii) Placement of $\mathcal{G}_{\text{pp}}$ into $\mathcal{G}_\mathcal{T}$. We have briefly described the clustering (or partitioning problem) in the section 3.2. In this section, we independently explore the placement problem (2) by comparing different techniques: Naïve, Spectral Embedding, Scotch and Simulated Annealing.

4.1 Naïve Placement

The Naïve approach is the standard mapping procedure adopted in the SpiNNaker toolchain for assigning populations of neurons to be simulated on the cores available in the SpiNNaker Platform. It is a simple and computationally light method to perform the graph placement without taking into account neither source and target graph connectivity.

The target graph was ordered following a polar coordinate system (ρ, φ) starting from a chip of choice. The radius $\rho = \max(|x|, |y|, |x - y|)$ has been calculated using the hexagonal distance. The angle $\varphi \in [0, 2\pi)$ is expressed in radians. The procedure starts to place a partial population into each processor and change the chip when all processors inside a chip are used. As the ρ increases, the sub-populations will be distributed along the chip on the circumference and will be separated by a greater and greater distance.

4.2 Spectral Embedding

The Spectral Embedding placement was partially used in a previous work described in [28]. The procedure involves the spectral analysis of the graph and a dimension reduction procedure to obtain a planar representation of it. By doing so, the target graph can be directly superimposed on the graph of the partial populations. Contrary to previous work, in which a greedy heuristic was used, the association of partial populations with processors was finally described through an *Integer Linear Programming* (ILP) problem.

The procedure starts with the extraction of the first five eigenvalues, and the relative eigenvectors, from the matrix $\mathbf{L}$. The eigenvectors form a matrix $\mathbf{\Lambda}$ that represents the partial populations in a $\mathbb{R}^5$ space. We apply a non-linear dimension reduction procedure using *Sammon Mapping* obtaining a space in $\mathbb{R}^2$.

The Sammon Mapping algorithm minimise the error function in (4) where d_{ij} is the distance in the high-dimensional space (eigenspace) and d^*_{ij} is the distance in the low-dimensional space (placement space) [24].

$$E = \frac{1}{\sum_{i<j} d_{ij}} \sum_{i<j} \frac{(d_{ij} - d^*_{ij})^2}{d_{ij}} \tag{4}$$

Each chip, in the chip mesh, is represented as a point (x, y) in an axial coordinate system. We superimpose the graph $\mathcal{G}_{\mathcal{T}}$ on $\mathcal{G}_{\text{pp}}$ projecting the chip mesh in the placement space (5).

$$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = \sqrt{\frac{2A_h}{3\sqrt{3}}} \begin{pmatrix} \sqrt{3} & -\frac{\sqrt{3}}{2} \\ 0 & \frac{3}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{5}$$

Where (x, y) is the chip coordinate in the hex mesh, and (x^*, y^*) is the chip coordinate in the placement space. The side length of the hex is used as a normalising factor and calculated using the area $A_h = \frac{A}{m}$ occupied by each chip. The normalising factor allows scaling the chip mesh concerning the area A occupied by the partial populations.

In the case where the target graph is fine-grain, we need to introduce the processors in the placement space. To ensuring spatial coherence, it was decided to place them equidistant along a circumference centred on the chip coordinate. The radius is chosen in such a way that it is smaller than the distance between two processors belonging to different chips.

After projecting the points into the placement space, they are translated to centre them on the median of the points representing the partial populations. Now we can describe the placement problem using the ILP formulation (6).

$$\underset{f(\mathbf{X})}{\text{minimize}} \quad f : \sum_{i=1}^{n}\sum_{j=1}^{m} x_{i,j} d_{i,j} \tag{6a}$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_{i,j} \leq k \quad \forall j \in \{1, \ldots, m\} \tag{6b}$$

$$\sum_{j=1}^{m} x_{i,j} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{6c}$$

Where the $\mathbf{X} = (x_{ij})$, $x_{ij} \in \{0, 1\}$ matrix is the placement matrix. An entry $x_{ij} = 1$ means that partial population i is mapped on the target node j. The problem constraints are two: i) Each target node can host at most k partial populations (6b). ii) Each partial population can be associated to only one target node (6c). The ILP problem was modelled using PuLP Python library and solved with *COIN-OR branch and cut* (CBC) solver.

4.3 Scotch

The Scotch mapping procedure makes use of the programs available in the homonym software suite (SCOTCH). The Dual Recursive Bipartitioning (DRB) is the primary procedure used by this tool [18]. The DRB can use a plethora

of other bi-partitioning methods according to a strategy defined by the user or deducted by graph properties. The main available methods are: Gibbs-Poole-Stockmeyer [9], Fiduccia-Mattheyses [6], Greedy Graph Growing [13] and Diffusion [19].

The mapping workflow with SCOTCH plans to pre-partition the target graph through the *amk_grf* program. The *amk_grf* program take in input a graph in *grf* format and create a target file (*tgt* format) which contains a decomposition-defined target architecture of same topology as the input graph.

Once a decomposition of the target graph has been obtained, the graph of the partial populations is placed on the target graph using the *gmap* program. The program *gmap* take in input the partial population graph in *grf* format and the target graph in *tgt* format and perform the DRB procedure minimising the communication cost function[2]. The *gmap* output file is a mapping file (*map* format) that contains the association between the Source and the Target nodes.

We had developed a Python module able to exporting a NetworkX graph to a file according to the *grf* format used by SCOTCH and capable of automating the procedures described above.

4.4 Simulated Annealing

The Simulated Annealing is a well know procedure used to find a good solution to an optimisation problem [14]. Given the problem in (2a), it is convenient to express the overall synaptic stretching in a matrix form and define a cost function to minimise. Given the partial population graph $\mathcal{G}_{\text{pp}}$ we build its Adjacency matrix $\mathbf{A} = (a_{ij})$ as described in (7).

$$a_{ij} = \begin{cases} w_{ij} & \text{if } \exists (v_i, v_j) \in E_{\text{pp}} \\ 0 & \text{otherwise} \end{cases} \quad \forall i, j \in \{1, \dots, n\} \tag{7}$$

Given the target graph $\mathcal{G}_{\mathcal{T}}$ we build its distance matrix $\mathbf{D} = (d_{ij})$ where each entry d_{ij} is the lenght of the mimimum path between two target nodes cpu_i and cpu_j. The distance matrix can be build using the Floyd–Warshall algorithms or repeating Dijkstra's algorithms if $|E_{\mathcal{T}}| \ll |V_{\mathcal{T}}|^2$.

Assuming to have as many subpopulations as target nodes and a placement rule $\Pi : \{v_1, \dots, v_n\} \to \{\text{cpu}_1, \dots, \text{cpu}_n\}$ we construct the *permutation vector* $\pi : (\Pi(v_1), \dots, \Pi(v_n))$ and the *permutation matrix* $\mathbf{P}_\pi = (p_{ij})$ in row form (8).

$$p_{ij} = \begin{cases} 1 & \text{if } i = \pi_j \\ 0 & \text{otherwise} \end{cases} \quad \forall i, j \in \{1, \dots, n\} \tag{8}$$

The permutation matrix is applied to $\mathbf{D}$ to permutate its rows and columns. We obtain the matrix $\mathbf{D}_\pi = \mathbf{P}_\pi \mathbf{D} \mathbf{P}_\pi$. The overall synaptic stretching can be

[2] The SCOTCH cost function is similar to our Synaptic Stretching

expressed in a matrix form and used as the cost function for the simulated annealing algorithm (9).

$$f : \mathbf{e}^T(\mathbf{A} \odot \mathbf{D}_\pi)\mathbf{e} = \sum_{i,j} a_{ij} * d_{ij}^{(\pi)} \tag{9}$$

Where $\odot$ is an element-wise multiplication and $\mathbf{e}$ is a column vector whose all elements are equal to one. In the case of a fine-grain $\mathcal{G}_\mathcal{T}$, before perform the synaptic stretching evaluation, the matrix $\mathbf{A} \odot \mathbf{D}_\pi$ should be collapsed in order to aggregate the processors belonging to the same chip. We used the Simulated Annealing implementation provided in the SciPy ecosystem using the *temperature* to decide how many elements of the permutation vector π to swap.

5 Results

In this section, we present the exploration experiments using the methods described in Section 4.

We use the Cortical Microcircuit (CM) as benchmark network, [20]. This network represents the connectivity of neurons inside a slice of the cerebral cortex with an area of $1\,mm^2$. The CM has been chosen because it is a rapresentative biological model with a relativly high global connectivity (5%) and natural clusters defined by the four cerebral cortex layers $\{L_{23}, L_4, L_5, L_6\}$. The CM is described in terms of Population and Projection with two populations for each layer, for a total of 8 Population and 64 Projections.

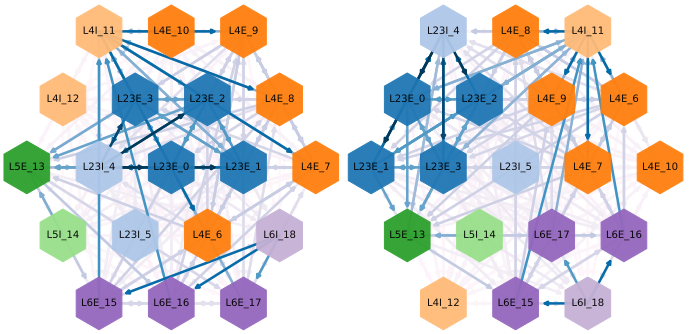
The network is composed of Integrate and Fire (LIF) and Spike Source (SRC) neuron models. The LIF neurons are models that mimic the biological neurons behaviour. The SRC neurons are simple programmable applications for outputting signals when desired. In this network, the SRC neurons are used to simulate the background activity of cortical neurons not presents in the model. Each SRC neuron is connected to only one LIF neuron, so they can be excluded by the $\mathcal{G}_\mathrm{N}$ provided that processors are reserved for their execution.

The CM model has 7.72e+4 LIF neurons and 2.99e+8 synapses. The network can be down-scaled to a percentage CM_p, for example:

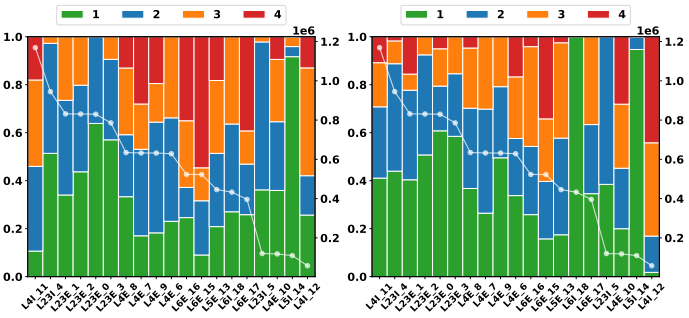
- $\mathrm{CM}_{5\%}$ has 3.86e+3 neurons and 7.47e+5 synapses.
- $\mathrm{CM}_{10\%}$ has 7.72e+3 neurons and 2.99e+6 synapses.
- $\mathrm{CM}_{50\%}$ has 3.86e+4 neurons and 7.47e+7 synapses.

For each processor in charge of simulating a LIF partial population, we must reserve two further processors. A processor is reserved for the simulation of paired SRC neurons. A further processor is reserved to host a special application necessary to manage synapses with delays greater than 10 ms, as described in [27]. Taking into account a set of 16 processors belonging to the same chip, we can place 5 partial population per chip for a total of a thousand neurons per chip.

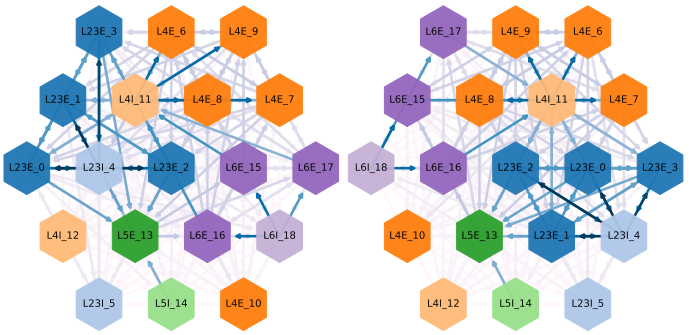
For simplifying the problem we perform a sequential slicing of each population in order to obtain partial populations with at most 1 000 neurons. In this way,



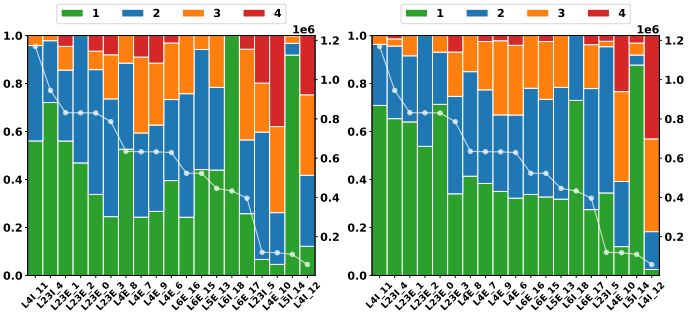
(a) Naïve Placement (+8%) (b) Spectral Placement (+18%)



(c) Naïve Placement (d) Spectral Placement



(e) Scotch Placement (+23%) (f) Simulated Annealing (+28%)



(g) Scotch Placement (h) Simulated Annealing

Fig. 3: The figures in the first row represent the placement of the partial population graph build from a $CM_{20\%}$ with 1000 neurons per chip on 19 chip (5 processors per chip). The figures in the second-row represent for each partial population the number of synapses (white line) and the percentage of synapse stretching.

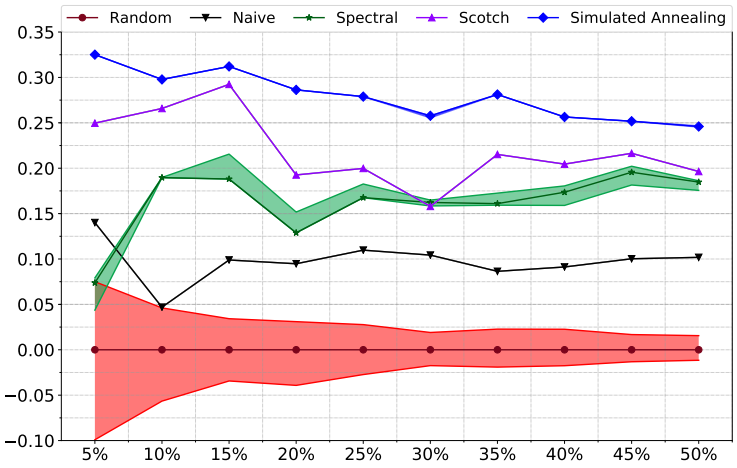


Fig. 4: The graph represents the improvement of a mapping technique with respect to the median of the results obtained with a random placement using a constraint of 1000 neurons per chip. The x-axis shows the CM scale factor. The areas represent the first and third quartile of the results obtained on 100 samples.

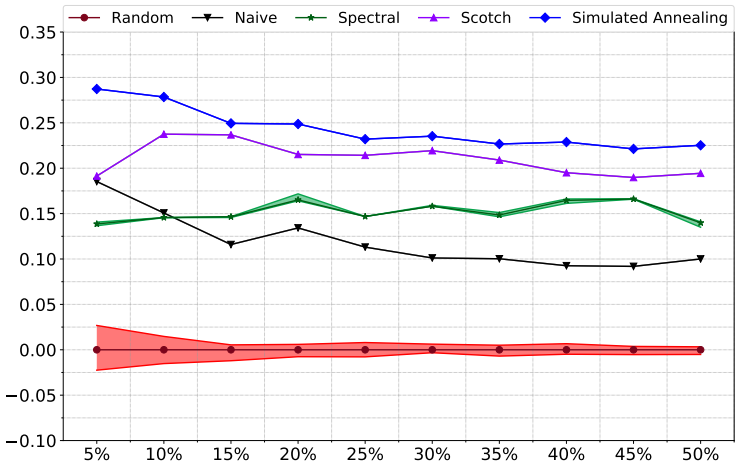
we can use a coarse-grain target graph where the nodes are the spinnaker chips (each chip with 5 processors and 200 LIF neurons per processor).

The experiment environment is composed of four different mapping procedures: Naïve, Spectral, Scotch and Simulated Annealing. We had generated 5 CM networks for 10 different scale factors, from 5% to 50%, for a total of 50 networks. For each network, we applied all mapping procedures 20 times. We evaluate the performance of each mapping procedure for each scale factor, using the fitness function (9). As a result, we obtain a distribution of 100 different placement results concerning overall synaptic stretching.

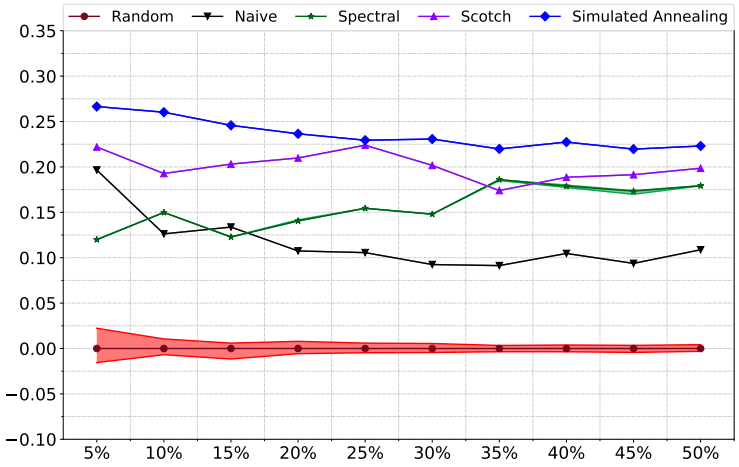
The performance of mapping procedures is compared to the performance of random placement. The median value of the results obtained with the Random procedure is used to compute the percentage improvement of the results obtained with other techniques.

In Fig.4 is depicted a chart that summarize all the experiments. On the x-axis, there are the network scale factors, on the y-axis the percentage placement improvements versus random. The data series are represented by polylines of different colours representing the medians of the results set. Each polyline is drawn within an area whose extremes delimit the first and third quartile of the results set.

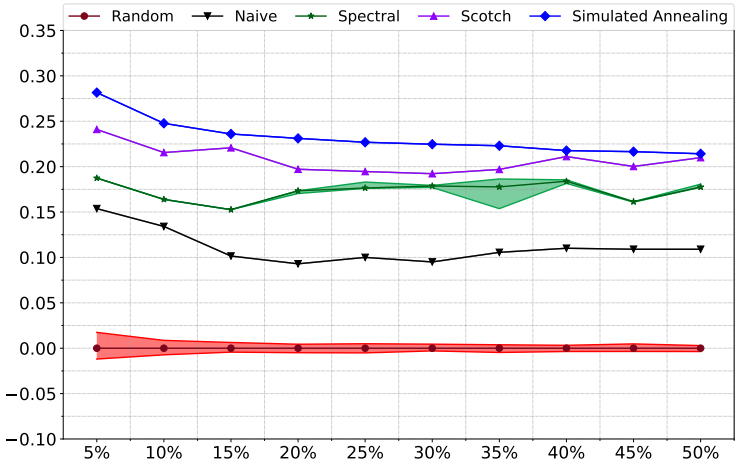
In Fig.3 are depicted the mapping results of a $\text{CM}_{20\%}$ into a target graph of 19 chip using the four placement techniques. Each hex represents a SpiNNaker chip connected with six neighbours. The colour of the hex area points out the belonging of the neurons, mapped on the chip, to one of the eight populations of the CM. The number of synapses shared between two partial populations



(a) Exploration with 200 neurons per node and fine-grain target graph



(b) Exploration with 150 neurons per node and fine-grain target graph



(c) Exploration with 100 neurons per node and fine-grain target graph

Fig. 5: The three graphs represent the improvement of a mapping technique with respect to the median of the results obtained with a random placement using three different constraints for the number of neurons in a processor. The x-axis shows the CM scale factor. The areas represent the first and third quartile of the results obtained on 100 samples.

is highlighted with the colour intensity of the edge that connects them. The different concentration of the connections with more synapses can be appreciated qualitatively from the figures 3a to 3f and quantitatively from the figures 3c to 3h.

In Fig.3a can be seen how the Näive method does not consider the connectivity but place each partial population sequentially following the polar ordering of the chip. Indeed there are many connections with a large number of synapses directed towards distant chips. This not happens in Fig.3f where the Simulated Annealing can localise in a defined area all partial population with a high number of shared synapses. In figures 3c and 3h the same information can be appreciated quantitatively. The chart has a bar for each partial population. Each bar represents the overall outgoing synapses of a partial population and shows the percentage of synapses at different levels of elongation. The white line depicts the number of synapses belonging to each partial population. The partial populations are sorted in descending order according to the total number of synapses.

We can see how better methods improve the percentage of synapses at a distance of 1 chip (Green) and decrease the percentage of synapses at a distance of 4 chips (Red).

While the results of the coarse-grain model were obtained by imposing a maximum of 1000 neurons per chip belonging to the same population, the results obtained with the fine-grain model were evaluated using three different values that limit the neurons per processor: 100, 150, 200. The results obtained using 200 neurons per processor are shown in Fig.5a. In the Table1, the results are shown in terms of processors involved. Where possible, a maximum of 5 processors per chip was used, because the network CM, in addition to the *lif* neurons considered there, makes use of other applications including a manager for synapses for high delays and a manager of external stimuli [src]. For each processor that simulates *lif* neurons, two other processors are required for a total of 15 processors per chip. In any case, in this exploration some configurations required more processors than theoretically available, so we ignored this constraint where necessary. Each chip, therefore, hosts from 500 to 1000 neurons belonging to different populations.

As the Fig.5 shows, the results show a profile similar to the one obtained with the coarse-grain model. However, it is noted, mainly for problems with many processors are involved, that the use of the methodology based on SCOTCH obtains results slightly inferior to Simulated Annealing. Considering the high efficiency of the solution offered by the SCOTCH suite and the simple $A + A_T$ symmetrisation necessary to use the tool, it is possible to renounce to the 2% of improvement but obtain a fast and acceptable solution.

6 Conclusions

In this paper, we described a mapping problem that involves a complex directed graph to be placed in a mesh of processors. We have modelled the mapping problem of SNN into SpiNNaker processor-mesh and split the problem into 3

	Neurons per Node								
	200			150			100		
CM	Processors	Chips	Ratio	Processors	Chips	Ratio	Processors	Chips	Ratio
5%	24	5	4.8	28	6	4.7	42	9	4.7
10%	42	9	4.7	54	11	4.9	80	16	5.0
15%	62	13	4.8	80	16	5.0	120	24	5.0
20%	80	16	5.0	107	22	4.9	157	32	4.9
25%	100	20	5.0	132	27	4.9	196	40	4.9
30%	120	24	5.0	157	32	4.9	236	48	4.9
35%	140	28	5.0	184	37	5.0	274	46	6.0*
40%	157	32	4.9	209	42	5.0	312	45	6.9*
45%	178	36	4.9	236	48	4.9	351	44	8.0*
50%	196	40	4.9	261	44	5.9*	390	44	8.9*

Table 1: Size of the fine-grain target graph used to position the CM. We have always tried to keep 5 processors per chip with different constraints for the number of neurons per chip. Some configurations marked with (*) could not meet the first constraint.

phases: the expansion, clustering, and mapping. Focusing on the mapping phase, we have identified and test 4 methodologies to solve the problem. The *Naïve* method maintains the proximity of clusters but does not take into account their connectivity. The *Spectral* method uses the graph eigendecomposition to obtain a planar representation of it and perform the node association with the chip mesh through an ILP formulation. The *Scotch* method uses the Dual Recursive Bipartitioning heuristic for fast mapping of a source graph into a target graph. The *Simulated Annealing* method uses the well-known procedure to minimise a cost function.

We are redefining the cost function of the placement problem bringing it into matrix form as a function of a permutation vector. We have chosen the cortical microcircuit at different scale factors as our benchmark network, preferring it for its high connectivity and the presence of clusters. After performing several tests on the chosen benchmark network, the results highlight the superiority of the Simulated Annealing method that works natively on direct graphs. Using a fine-grain model, the gap between the SA and SCOTCH based method has narrowed, especially when dealing with particularly large graphs. In these cases, the (more efficient) SCOTCH-based method has the advantage of providing an acceptable solution in a shorter time.

This modelling system for SNN placement problems can be adapted to other architectures such as Intel Loihi and SpiNNaker 2 for investigating new mapping techniques to be adopted for improving the usability of these emerging architectures. In the next works, we will implement these techniques within the placement pipeline of the SpiNNaker neuromorphic architecture, to offer an alternative to the currently implemented method (Naïve) and evaluating experimentally the reduction of communications between the chips involved.

References

1. Abbott, L.F.: Lapicque's introduction of the integrate-and-fire model neuron (1907). Brain research bulletin 50(5), 303–304 (1999)
2. Cassidy, A.S., Alvarez-Icaza, R., Akopyan, F., Sawada, J., Arthur, J.V., Merolla, P.A., Datta, P., Tallada, M.G., Taba, B., Andreopoulos, A., Amir, A., Esser, S.K., Kusnitz, J., Appuswamy, R., Haymes, C., Brezzo, B., Moussalli, R., Bellofatto, R., Baks, C., Mastro, M., Schleupen, K., Cox, C.E., Inoue, K., Millman, S., Imam, N., Mcquinn, E., Nakamura, Y.Y., Vo, I., Guok, C., Nguyen, D., Lekuch, S., Asaad, S., Friedman, D., Jackson, B.L., Flickner, M.D., Risk, W.P., Manohar, R., Modha, D.S.: Real-time scalable cortical computing at 46 giga-synaptic ops/watt with 100x speedup in time-to-solution and 100,000x reduction in energy-to-solution. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 27–38 (Nov 2014)
3. Chung, F.: Laplacians and the cheeger inequality for directed graphs. Annals of Combinatorics 9(1), 1–19 (2005)
4. Davies, M.e.a.: Loihi: A neuromorphic manycore processor with on-chip learning. IEEE Micro 38(1), 82–99 (2018)
5. Davison, A.P.e.a.: Pynn: a common interface for neuronal network simulators. Frontiers in neuroinformatics 2 (2008)

6. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Papers on Twenty-five years of electronic design automation. pp. 241–247. ACM (1988)
7. Furber, S.B.e.a.: The spinnaker project. Proceedings of the IEEE 102(5), 652–665 (2014)
8. Furber, S.e.a.: On-chip and inter-chip networks for modeling large-scale neural systems. In: Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on. pp. 4–pp. IEEE (2006)
9. Gibbs, N.E.e.a.: A comparison of several bandwidth and profile reduction algorithms. ACM Transactions on Mathematical Software (TOMS) 2(4), 322–330 (1976)
10. Izhikevich, E.M.: Simple model of spiking neurons. IEEE Transactions on neural networks 14(6), 1569–1572 (2003)
11. Jin, X., Furber, S., Woods, J.: Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In: Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on. pp. 2812–2819 (June 2008)
12. Jin, X.e.a.: Algorithm and software for simulation of spiking neural networks on the multi-chip spinnaker system. In: Neural Networks (IJCNN), The 2010 International Joint Conference on. pp. 1–8. IEEE (2010)
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing 20(1), 359–392 (1998)
14. Kirkpatrick, S.e.a.: Optimization by simulated annealing. science 220(4598), 671–680 (1983)
15. Liu, C., Bellec, G., Vogginger, B., Kappel, D., Partzsch, J., Neumärker, F., Höppner, S., Maass, W., Furber, S.B., Legenstein, R., et al.: Memory-efficient deep learning on a spinnaker 2 prototype. Frontiers in neuroscience 12 (2018)
16. Maass, W.: Networks of spiking neurons: the third generation of neural network models. Neural networks 10(9), 1659–1671 (1997)
17. Mann, Z.Á.: Multicore-aware virtual machine placement in cloud data centers. IEEE Transactions on Computers 65(11), 3357–3369 (2016)
18. Pellegrini, F.: Static mapping by dual recursive bipartitioning of process architecture graphs. In: Scalable High-Performance Computing Conference, 1994., Proceedings of the. pp. 486–493. IEEE (1994)
19. Pellegrini, F.: A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In: European Conference on Parallel Processing. pp. 195–204. Springer (2007)
20. Potjans, T.C.e.a.: The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. Cerebral cortex 24(3), 785–806 (2014)
21. Rast, A., Stokes, A., Rowley, A., Davies, S., Lester, D., Furber, S., Whatley, A., Luck, C., Iakymchuk, T., Ros, P., Partzsch, J., Reza, A., Bi, S., Neil, D., Stefanini, F., Binas, J., Scott, N., Waniek, N.and Celiker, O., Isaacs, P., George, R., Urgese, G.: Aerie-p: Aer intersystem exchange protocol (2015)
22. Rhodes, O., Bogdan, P.A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., Lester, D.R., Mikaitis, M., Plana, L.A., Rowley, A.G., et al.: spynnaker: A software package for running pynn simulations on spinnaker. Frontiers in neuroscience 12 (2018)
23. Ruggiero, M.e.a.: A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness. International Journal of Parallel Programming 36(1), 3–36 (2008)

24. Sammon, J.W.: A nonlinear mapping for data structure analysis. IEEE Transactions on computers 100(5), 401–409 (1969)
25. Siino, A., Barchi, F., Davies, S., Urgese, G., Acquaviva, A.: Data and commands communication protocol for neuromorphic platform configuration. In: 2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC). pp. 23–30 (Sept 2016)
26. Sugiarto, I.e.a.: Optimized task graph mapping on a many-core neuromorphic supercomputer. In: High Performance Extreme Computing Conference (HPEC), 2017 IEEE. pp. 1–7. IEEE (2017)
27. Urgese, G., Barchi, F., Macii, E.: Top-down profiling of application specific many-core neuromorphic platforms. In: IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-15) (IEEE MCSoC-15). Turin, Italy (sep 2015)
28. Urgese, G., Barchi, F., Macii, E., Acquaviva, A.: Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms. IEEE Transactions on Emerging Topics in Computing pp(99) (2016)
29. Van Albada, S.J.e.a.: Full-scale simulation of a cortical microcircuit on spinnaker. In: Front. Neuroinform. Conference Abstract: Neuroinformatics. vol. 10 (2016)
30. Van Dongen, S.: Graph clustering via a discrete uncoupling process. SIAM Journal on Matrix Analysis and Applications 30(1), 121–141 (2008)