POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING
Master degree in Computer Engineering - Embedded Systems

Master Thesis

A graph oriented approach for implementing the MPI multicast communicator on a many-core GALS platform

Supervisors:
Gianvito Urgese
Evelina Forno
Francesco Barchi
Elisa Ficarra

Candidate:
Alessandro Salvato

October 2020

Summary

The binding between Computer Science and Biology is something much narrower than you normally may think. Emulation of human brain behaviour is one of the most hot topics let these two branches merge into. Nowadays, in the neuromorphic field, are available several SW and HW capable to simulate the behaviour of biological neural networks, identified in the biological studies. The research field of this thesis work is emulation of the brain by means of embedded applications. At this point we need to distinguish between software and hardware solutions. The formers, probably, are the most famous ones. This group counts artificial intelligence and artificial neural networks. Born in software, they may run on general purpose machines or on customized ones. Special purposes hardware and its own configuration is one of the target branches of Bioinformatics team at Politecnico di Torino. The machine, going to emulate a biologic brain, has been developed at University of Manchester, in collaboration with the University of Southampton, University of Cambridge, University of Sheffield, ARM Holdings and Silistix Limited. It's a **neuromorphic** hardware, from Wikipedia web-page: *Neuromorphic Engineering, or Computing, describes the use of multicore and analog circuits to mimic neuro-biological architectures present in the nervous system.*

This neuromorhic machine is called Spin5 (sometimes in this document we will also use more generically the stylized SpiNNaker or classic Spinnaker). From the logic point of view that's composed of 768 general purpose ARM cores, capable to simulate behaviour of brain neurons. It's easy to understand that all of them run in parallel, turning both throughput and memory availability as issues. Moreover, communications are allowed to be run in broadcast (1-to-all), multicast (1-to-more) and unicasting (1-to-1). The Bioinformatics' team has already developed libraries for both broadcasting and unicasting; this thesis work aims at integrating and making coexist the multicast communication scenario.

Spin5 is an hardware simulator running one subcategory of artificial neural networks: the **spiking neural networks**, SNN for short. They add the concept of timing to its own model, for which one neuron generates a signal, an electrical spike, after a certain number of events coming from the environment. The logic line dividing the neuron from the others has named **membrane potential**. That means that, in addition to developing the routing algorithms for the management of the multicast case, the implementation of a synchronization system has required, customized for multicasting, as well. All of these must be done taking into account the small memory availability for routing and synchronization, trying as well to minimize the throughput, no to forget that SpiNNaker is massively parallel computing oriented hardware.

MPI is one of the most used protocol when developers work with distributed machines. A lot of documentation comes from open-license works, making literature really widespread. Moreover, different MPI implementations fit on different machines. The MPI software stack for Spinnaker draws an environment made up of various layers of abstraction. However, as said above, only communicators for unicast and broadcast exist by now. Now, the main goal stands on multicast communicators realization.
Before focusing on that, a pre-processing step has to be performed. The solution space for any possible multicast group is really huge, turning any consideration on its manage-

ment not so trivial. Due to poverty on memory resources, the compression data phase, to be filled in routers, it's absolutely necessary.
MPI allows collecting data from parallel-processes, but Spinnaker is a Globally Asynchronous Locally Synchronous architecture-based. It means that misalignments, among hundreds of core, running in different clock domains, can occur very frequently. Therefore, some synchronization cycles must be performed in order to ensure the correctness of MPI flow. It will be presented a synchronization mechanism, fitting multicast scenarios, trying to satisfy all "special cases", but keeping algorithm as much as general. Synchronization follows same specification of broadcast implementation, but some variations have been applied, such as the number of hierarchical layers.
Evaluation of results passes through a customized simulator of the board. It acts as debug tool to collect information on the correctness for both synchronization and routing rules. Its specifications are described. Then, some simulations have been performed, trying executing 100 multicast groups at the same time. Slightly tuning some simulator's parameters and application model where MPI is running over, produces different results, that will be analyzed and commented in the end to enhance future publications and operational ideas for next works.

Acknowledgement

And I would like to acknowledge ...

Acknowledgements are mandatory only when people outside the academic institution supported the development of the research that was performed in order to reach the conclusion of the doctorate program.

La serenità è la più bella delle felicità

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 A bit of history about Neuromorphic engineering

Interests on both nervous system and human brain come from far away. First studies were up to Incas in Americas and to ancient peoples of Mesopotania since 10000 BCE. Other evidences are recurrent, Egyptians, Greeks and Romans have, over the centuries, laid the foundations for one of the most interesting branches of modern biology: Neurology.
In the XVII century Physiology was born; that's the branch dealing with *how* living beings work. But we have to wait for another century, in full Age of Enlightenment, to reap the rewards of another field that will help Neurology to evolve into a new layer of comprehension; I'm talking about Electrophysiology. Actually, early hypothesis were elaborated in Renaissance by Italian anatomists Costanzo Varolio and Bartolomeo Eustachi. They were among the first that had discredited the Galeno's millennial theory, strongly based on Ethics, Logics, Physics, Classical Philosophy and the conception that human emotions reside in organs. Electrophysiology tried to prove electrical activities in the brain, and more generically across the nervous system. Luigi Galvani was one of the most famous scientist of this branch. He could prove the deep link between electrical activity and life: Galvani had applied a differential of potential between two nodes, putting in the middle a dead frog; he had observed that the limbs of the animal twitched [5]. How was this possible was the big question of next years. The answer, nowadays obvious, is that the nervous system needs electricity to work. All the informations have transmitted from the brain to everywhere by means of electrical signals travelling all around that.
Within brain, the basic block predisposed to propagate electricity is neuron. Another italian scientist had a very important role in this path: the 1906 Medicine Nobel prize Camillo Golgi "Work on the structure of the nervous system"[6]. He was able to depict the neuron' structure, mainly composed of nucleus, dendrites and axon.
Despite World Wars began, both brain and computers studies pursued. In 1929 the first encephalogram was performed; it consists in measuring the current flowing through celebral cortex. In 1944 the first programmable electronics ever was built in Bletchley Park, England, by Alan Turing and Max Newman, having the main purpose to decrypt Nazi machine messages by Enigma. Probably, the most important year, leading to the first match between biology and computer science is 1943. "A logical calculus of the ideas immanent in nervous activity", McCulloch & Pitts, was published. This paper proposed

a mathematical and computational model to run algorithms by means of logical lattice schemes. At the beginning, the actual structure of neuron has depicted, describing how it can turn into an excited state depending on coming of external impulses. The theory description follows, providing some physical assumptions observable from real neurons [7].

1. *The activity of the neuron is an "all-or-none" process.* It means that a neuron/node can execute just a task at a time

2. *A certain fixed number of synapses must be excited within the period of latent addition in order to excite a neuron at any time, and this number is independent of previous activity and position of neuron.* It refers to the fact that any reaction has produced after a certain number of stimulus.

3. *The only significant delay within the nervous system is synaptic delay.*

4. *The activity of any inhibitory synapse absolutely prevents excitation of the neuron at that time*

5. *The structure of the net does not change with time*

Later in 1958, Von Neumann and Rosemblatt refine the previous theory. They proposed more complex models, much more similar to brain, and the first Artificial Neural Networks, perceptron based.
In 1990 another very important milestone was published: *Neuromorphic Electronic Systems* by Mead Carver. First, he compared performaces of a 90s microprocessor with respect to the brain's, stating that "*the brain is a factor of 1 billion more efficient than our present digital technology, and a factor of 10 million more efficient than the best digital technology that we can imagine*" [1]. Moreover, Carver treats the leakage power issue in modern digital electronics; he says that just a small fraction of energy has used to charge the capacitance of a transistor, but the higher amount of the total is used for internal wires and interconnections. These considerations laid to one question: how can nervous system be so more efficient than digital electronics? To enhance neuron synapse performance with respect to a MOS transistor, Garver advanced results of his observations by plotting the response of both as function of difference of potential applied on two terminals 1.1. His own conclusion bases on the fact that analog electronics fits better than digital one to mimic neurons. However that was not enough. Neurons own the capability to adapt themselfs, developing a kind of memory of the past. Again, Carver talks about analog memories, EPROMs and EEPROMs, as proof of the fact that analog components are compliant for this kind of implementation. He will refer to them as **neuromorphic systems**.
Apart from the hardware proposed by University of Manchester, that will be explained in next sections, let's have a look of current state of art of research and who are the main actors, as companies, that are investing in neuromorphic engineering. In 2014 IBM published a paper where their own 4096 cores and 1 million neurons architecture was proposed [8].
Intel designed Loihi in 2017. It's a self-learning neuromorphic research chip, composed of 128 cores, counting around 130.000 neurons. Actually Loihi cannot be accessed by general audience, since test phase is still ongoing [9].
Also mobile market hasn't been untouched from neuromorphic wave. Just think to

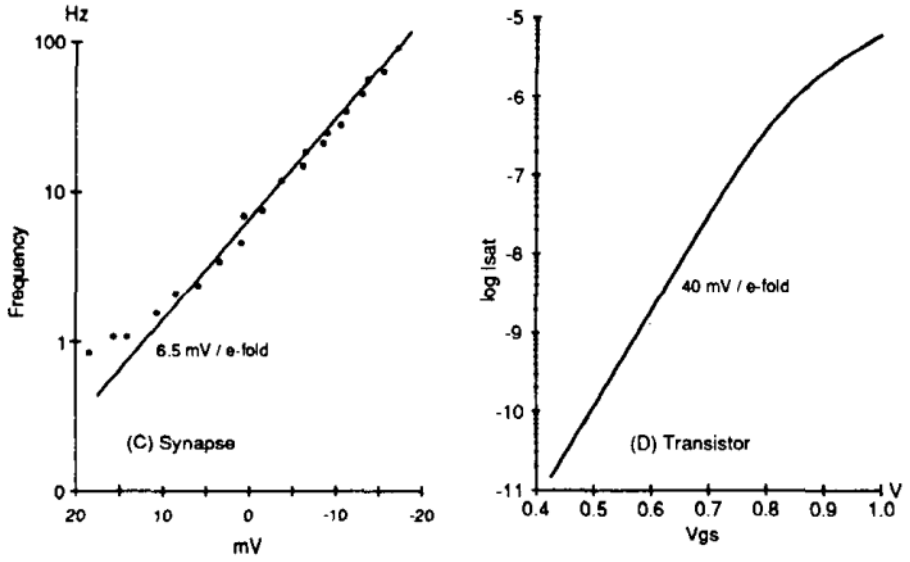

Figure 1.1: Curve C illustrates the exponential dependence of the arrival rate of packets of the neurotransmitter at the postsynaptic membrane on the presynaptyc membrane potentiale. Curve D shows the saturation current of a MOS transistor as a function of gate voltage. [1]

Google's TensorFlow Lite. It's a framework allowing developers to exploit this kind of technology without knowing machine characteristics. TensorFlow Lite is already available for Android, iOS and Raspberry Pi.

1.2 Neuron biology

Neuron is the basilar unit building nervous tissue. It allows electric impulses to be transmitted. A sketch of it is reported at figure 1.2.
Nucleus has contained in soma, that's the cell body basically. Within Golgi apparatus, mitochondrions, endosplamic reticulum rough and smooth stay. Neuron's nucleus, as the same matter of other type of cells, synthesizes RNA and proteins from mRNA. Two kind of branches depart from soma: dentrites and the axon. The formers work directing information from the environment to the center of cell body. Data may come from close neurons. Dentrites are identified as short-range projections [10]. The axon owns some outstanding feature. Any single neuron has only one axon. In opposition to dentrites, axon data direction is from the cell body to outside. Chemistry reactions due to some ions such as sodium, chlorine, calcium and potassium allow current to flow along. Action of myelin sheat and nodes of Ranvier (very short compared to the whole axon length) produces a very intense differential of potential between the inner membrane and the environment. This causes electrochemical events known as **spikes**. Speed of transmission can reach 120 m/s. Then, the axon tail is composed of several minor branches able to reach the remotest area of the brain, in fact these are considered as long-range-projections [10].

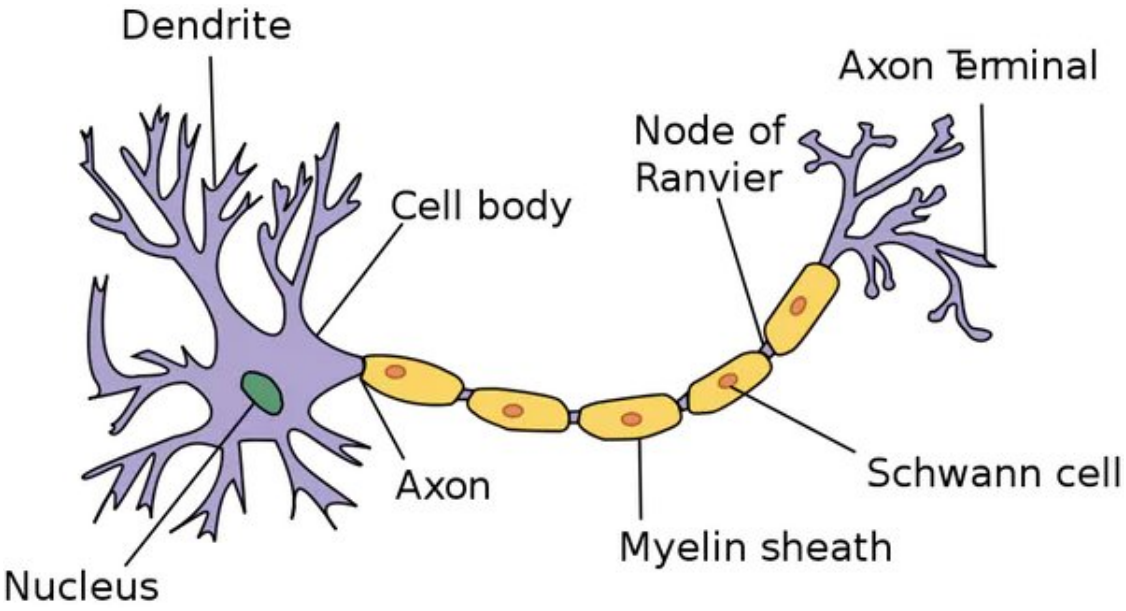

Figure 1.2: Neuron anatomy

1.3 Why Neuromorphic technologies?

Carver (1990) and some other else before, tried to explain digital electronics limits, meaning that scalability couldn't continue for ever. A platue would have been achieved in next years. Actually, that has been happening. First of all the problem stands over the transistor size. Although it's true, the smaller the transistor channel, the higher is density per area, it hasn't to be forgotten that a too tiny transistor cannot work. That's not only a geometrical issue, but interests also electrical quantities. As well as transistor channel, also voltage scaling is getting more and more hard.
New implementations are moving to parallel architectures mainly. Keeping the same transistor technology, but improving system organization, it's possible to get better performances. Obviously, splitting tasks may generate some hazards that did not exist in a single pipeline cortex. Modern microprocessors allow sub tasks to be executed in an out-of-order manner, exploiting the Tomasulo architecture or newer architectural paradigms. Strong parallelization fills also server management and web services, like Amazon Web Services, where any module has distributed in all around the world, but acting as a one. Following this philosofy, Neuromorphic engineering acts. New machines able to parallelize tasks, managing sub units which are able to exchange a lot of messages at minimum cost, both in terms of power consuming and in terms of delay introduced, reminds to neurological activity. Neuromorphic computers can be seen as cluster of neurons, able to parallelize a very high amount of tasks and to be feasible at high throughput scenarios. That's the reason why we can state that any neuromorphic machine is an hardware try to mimic human brain behaviour and structure from the morphology and electrical points of view.
In 2006 Georgia Tech released a field programmable neural array, mimic the channel-ion characteristics of neurons in the brain. In 2011 MIT produced an hardware able to replicate the synaptic behaviour. Both of them have been realized by MOSFET technology, but the emerging ones are getting more and more considered. In 2012 a neuromorphic

chip was fabricated using ferromagnetic units and memristors, these called also as *neuristors*.
It's not so hard to imagine what kind of applications may be interested by Neuromorphic machines. Graphics Processing Units of course, for floating points operations, but also to find out new oil wells or to determine stock option curves on the stock exchange. We can consider all those application machine learning based: vocal recognition, self-driven cars, astronomic simulations and assistant surgeons.

1.4 The Human Brain Project

In 2013 the European Union collected funds to realize software and hardware projects to move neuroscience and medicine on technological improvements. The Human Brain Project was born. To it, 500 scientists and researchers, from more than 100 universities, take part. Six main applicative branches [11] cover

- *Neuroinformatics*
- *Brain simulation*
- *High performance analytics and computing*
- *Medical informatics*
- *Neuromorphic computing*
- *Neurorobotics*

Neuromorphic computing and Brain simulation paths matched SpiNNaker ongoing development. It started in 2005 and the first prototype was released in 2009, ten years ago. Initial SpiNNaker ideas totally fit Human Brain Project purposes. Trying emulating brain activity, with millions of connected cores, acting as independent neurons, may lead outbreaks in Neuroscience [12] [13] [14].

The neuromorphic platform has thought for high-performance massively parallel processing. Politecnico di Torino university, in collaboration with University of Manchester, started in developing a MPI protocol feasible running of Spinnaker. The goal is to enlarge the algorithmic window to general purpose applications; in fact Spinnaker has mainly been developed to run brain simulations, it does not natively support a general purpose programming. That's the reason why researchers decided in MPI usage.

Chapter 2

Background

2.1 Some examples of neural networks

The union between Computer Science and Biology is something much narrower than you normally may think. A lot of algorithms and data structure derive from nature observation. The reference is quiet immediate: look at tree structures, for instance. They are a very understandable and easy study case, taught in any academic programming course. Names as roots and leafs come from nature. A much complex example regards genetic algorithms. This kind of strategies try solving optimization problems, where any single feature of the final possible solution is encoded in a gene. Combinations of gene lead to chromosomes. Population of individuals, owning certain chromosomes, are set of possible solutions. Evaluating these individuals allows to filter the bests, in the same manner natural selection makes the strongest survive and others become extinct. Moreover, genetic algorithms may apply random changes in chromosome solutions: these steps are called mutations and again, researchers took inspiration from biologic world. Mutations may lead to worst individuals or to better ones, with totally new features. For example, genetic algorithms are used to improve performances of inside processors, by fetching instructions not in the chronological order, but ensuring the normal behaviour [15].

Deep learning is a very recent subfield of neural network theory[16]. It started as machine learning system devoted at classification system for image processing. With respect to classic neural networks, deep neural networks expect a number of hidden layers higher than 2, typically. By increasing the number of middle layers, we increase the number of hierarchical feature extraction models that we can include in neural network. The brain has a deep architecture. For instance, the cortical area for visualization seems to be composed of some sequences, where images get processed step by step. Then, it acts like some kind of hierachy standing up; human, first, learns simpler concepts and then compose them to represent more complex ones.
Traditional machine learning system uses ad-hoc solutions depending on data nature and knowledge on those. Obviously, this kind of approach highly relies on a single application and it's really hard to make it portable, moreover requires a deep knowledge of data nature. Deep learning approach, by means of multiple layers, tries to extract information by each of them, from the rawest features and to the finest ones, stage by stage.

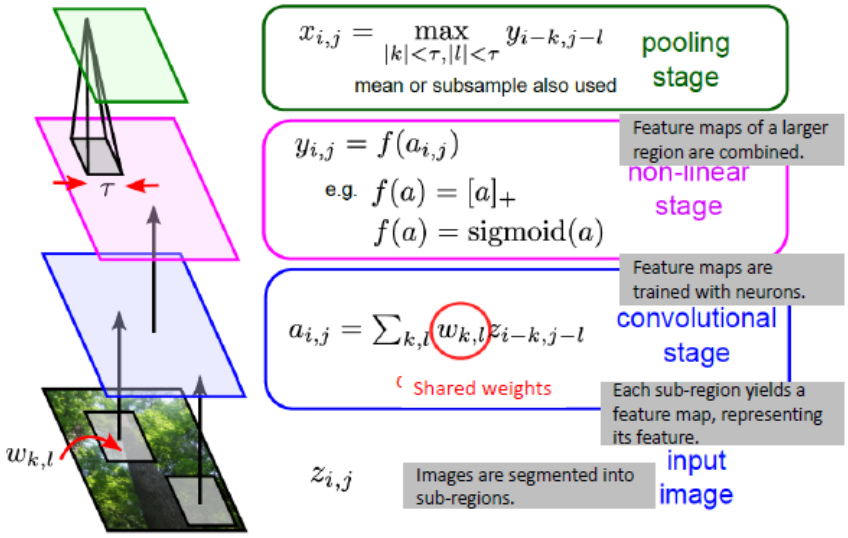

Figure 2.1: Building-blocks for CNN

2.1.1 Convolutional Neural Network

Convolutional Neural Network are mainly used for image recognition (Fig. 2.1), strongly implemented by graphical processing units [17]. In CNNs not all neurons are sensitive to the full data set, or, referring to images, the entire visual field, but only single subregions. Two kind of cells can be listed: simple neurons detecting patterns and complex ones having a larger receptive field.
With respect a classical network, a CNN takes as input a much lower number of inputs, because they are organized in a hierarchical manner and layers are not fully connected. Stages act on chuck of data, not on the whole set: lower layer extract local features, top-level layers extract global patterns.

The convolutional stage applies the convolutional operator between the input image, or a portion of that, to a kernel. The result is a multidimensional **feature map** for a specific visual pattern; obviously it can be done providing kernel hand-crafted. Convolutional operation is the base of filters for specific features.
The input is a $m \times n$ matrix, the kernel instead is a matrix $p \times q$, such that $p \leq m$ and $q \leq n$. The kernel moves over the input, without exceeding the boundaries, and performs a sum of product between elements of both having the same relative row and column indexes. For visualization look at Fig. 2.2. Mathematically, the elements of output s are so computed

$$s[x, y] = (input * kernel)(x, y) = \sum_{i=0}^{p} \sum_{j=0}^{q} input(x, y) \cdot g(x - i, y - j)$$

Actually, in a CNN we handle more kernels, the number of kernel has said **depth of CNN**.

As said before, CNN are locally connected, means that not all neuron of a layers are strongly connected to all neurons of previous and next layer. It means that nodes are unresponsive to variations outside of its receptive field. The architecture thus ensures

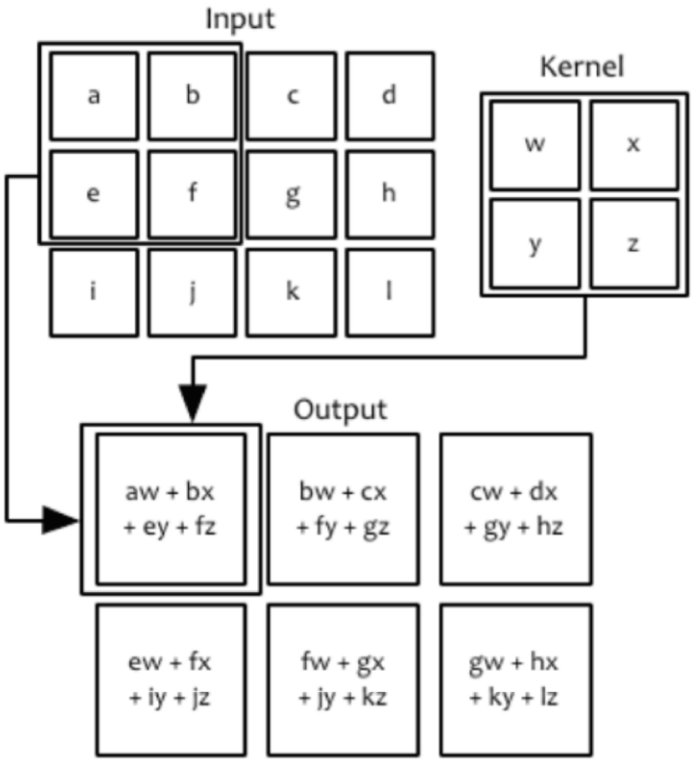

Figure 2.2: Example of convolution between two bidimensional objects

that the feature extractors produce the strongest response to a **spatially** local input pattern. Another characteristic regards **shared weights**.

Then, CNN have another layer performing some non linear function, in order to better mimic neurons.

In the end, to perform a kind of reduction on number of features obtained by convolution and non linear stages, **pooling** has performed, on a dedicated layer. It works as the same as a feature reduction algorithm, which in image processing can be considered as a compressor. The feature map has divided into sub non overlapped rectangles, then applying some grouping operation for each of them, such as maximum value search, mean computation, so on.
Convolution, non linear and pooling stages can be repeated in a hierarchical manner, as depicted in Fig. 2.3.

2.1.2 Recurrent Neural Networks

Classic neural networks considers fixed length inputs to produces fixed length vector of output. This leads that the number of computational step are known. Recurrent neural networks, RNN for short, try to overcome that constraint, allowing the management of sequences of vectors. RNN are often used in text processing because sentences and texts are sequences of words.
RNN extends kind of field of application, because each input feature has provided for any timestamp of the analysis, that's the reason why we talked about sequences. Actually

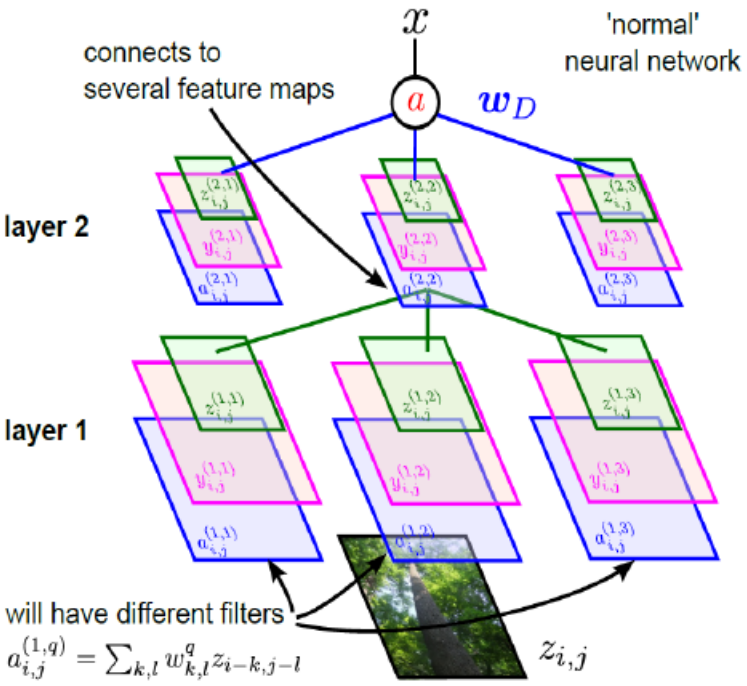

Figure 2.3: Full CNN: in blue the convolutional stages, in pink the non-linear function and in brown the pooling stage. At the top of the pyramid, each main branches produce a weight that will take as input of a classical neural network.

RNN, to accomplish a job, takes previous output or hidden states as next inputs. With respect to classic neural network, a recurrent one contains loop in its body; justifying a sequential behaviour Fig. 2.4.

Typically any RNN implements the following formula:

$$h_t = \tanh(W, x_t, h_{t-1})$$

Actually, the RNNs can be converted into a feed forward network by unfolding over time (Fig. 2.5), making optimization problem easier to solve, like the minimization of the error function. Acting in this manner any loop disappears. Then, a very common algorithm used in neural network analysis can be applied: the backpropagation. Talking about Recurrent Neural Network, it names **backpropagation through time**, BPTT for short. Grandient variation are applied at each time step, as you can see from Fig. 2.6. It's required the knowledge of initial states in order to apply BPTT, more or less in the same manner happened for classic backpropagation. Typically, initial states usually are set as random values.

2.1.3 Bidirectional RNN

For many sequence, labelling tasks would lead some benefits by accessing to future as well as past context. However, since standard RNNs process sequences in chronological order, they ignore future context. To solve this issue, there exist another kind of RNN, called BRNN, standing for **Bidirectional Recurrent Neural Network**. By doubling the hidden layer into a backward and a forward one, such that the former runs sequence from

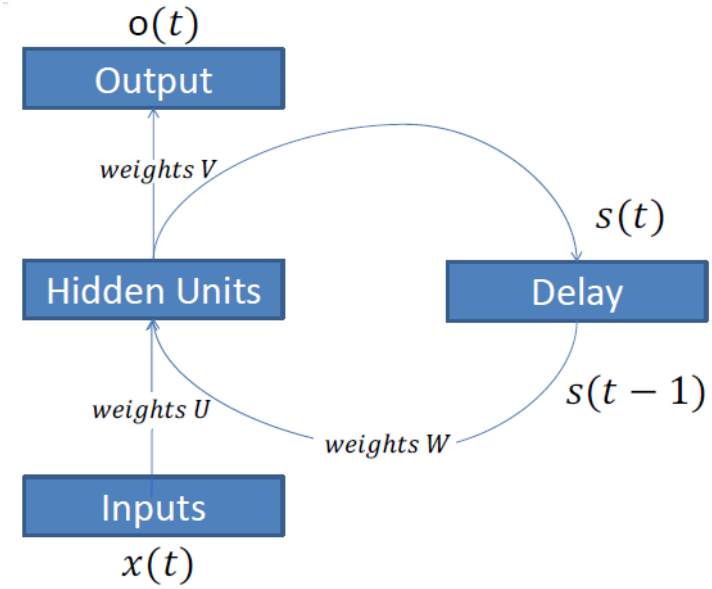

Figure 2.4: RNN architecture

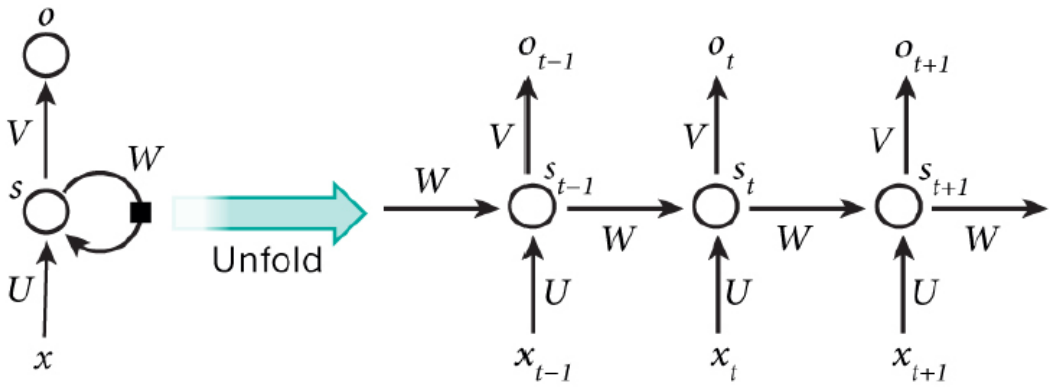

Figure 2.5: Unrolling of a recurrent neural network

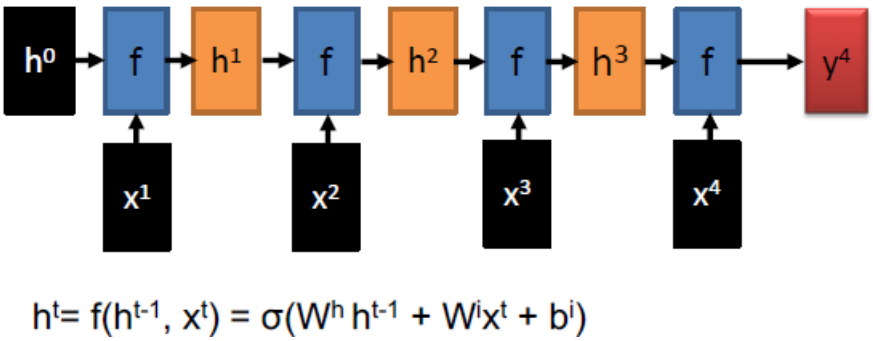

Figure 2.6: Solver schematic view of backpropation problem for a recurrent neural network after unfolding. h is the state vector, while f is the activation function. x is the vector of input applied at any timestamp t, producing in the end the output y. It comes that any state element h^t, produced by the current activation function f^t, depends on the previous step input x^{t-1} and previous state h^{t-1}. Don't forget to consider weights W in computation.

the first to the last, the latter works in opposite propagation, Fig. 2.7. The mechanism is pretty simple. The two middle layers act independently. Before, inputs are applied in order to run the forward chain. Then, inputs are applied, from the newest to the oldest, enabling the backward layer. Finally, the overall computation acts in producing output values, that has performed by considering storing activations from both the hidden layers.

It may happen that the action of gradient can be lost due to both the number of a lot of steps and middle computations, altering the right updating of weights, due to multiplication operation performed along the pipe. This issue is known as **vanishing or exploding gradient**. This can be incurred with 100 element sequences, so RNN and BRNN are hard to handle with large feature sets.
To fix vanishing gradient exists a solution called **Long Short Term Memory**, LSTM for short. By adding some other structure to a classical RNN it's possible to mantein the hidden states for long time. An LSTM block acts in three instant of time, $t-1, t, t+1$. The LSTM block receives the feature sample $x(t)$ and produces $y(t)$. Moreover, it takes as input the state of the previous step $s(t-1)$, then producing the step for the next timestamp $s(t+1)$. The LSTM node has called **memory cell**, where the logical structure has shown in Fig. 2.8. To keep the same math, within a memory cell exists an edge with unitary weight for state storing. The unitary weight is used to avoid impacting on gradient, this method has knowns as CEC, Constant, Error Carousel. From an architectural point of view, a memory cell has composed of four main elements:

- Input gate: controls whether input can be made passed
- Forget gate: manages the CEC, enabling it or not
- Output gate: modulate the signal coming out by net output
- Neuron itself with self-recurrent

Any gate may produces either 0 or 1, because they integrate sigmoidal function, then multiplied by a certain "internal" weight.

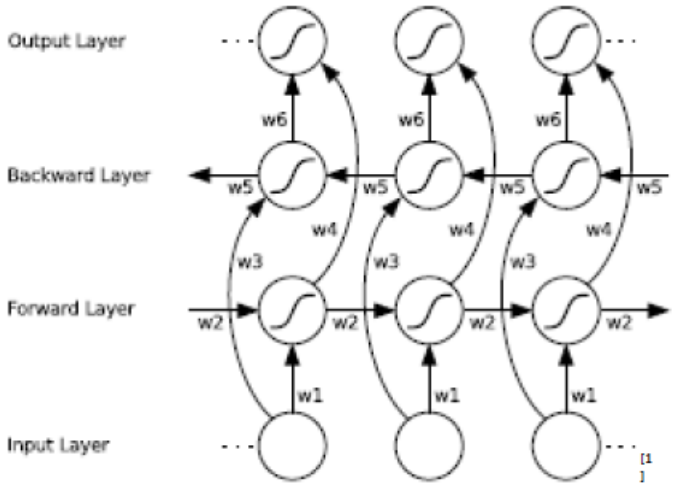

Figure 2.7: Bidirectional Recurrent Neural Network unfolded

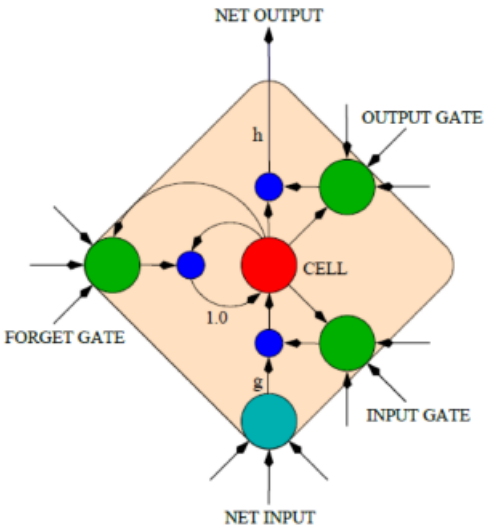

Figure 2.8: Neuron cell structure in a Long Short Term Memory context

2.2 SpiNNaker main architecture

In the last years, the Advanced Processor Technologies Research Group, based at the University of Manchester, has developed a new hardware architecture in order to emulate the human brain computation. This kind of technology is known as **neuromorphic**. The meaning of this term lies on the attempt to emulate the neuronal-biologic behaviour of the human nervous system: SpiNNaker project was born in 2005. Nowadays, after 15 years, SpiNNaker is a supercomputer made up of 1 million of ARM microprocessors. Its modelling tries to emulate an human brain counting 80000 neurons [18].
Another relevant SpiNNaker feature is that it acts as a GALS architecture, standing for Globally Asynchronous Locally Synchronous. Typically, this kind of hardware, detects some portion of it, called *clock domains*. Within a domain, everything follows the same clock, whose frequency may be different, a lot, than another one. However, to ensure connectivity and global communications, among domains are kind of asynchronous communicators. The big advantage, enhanced by GALS, stands on the fact that hardware can be massively pushed towards parallel computing. Moreover, it fits well the event-driven behaviour of spiking neural networks.
Supercomputer SpiNNaker has composed of several modules: the Spin5 boards (Fig. 2.9). In this thesis work the focus is on the Spin5 stand-alone. No considerations will made over Spin5 *inter* interconnections. SpiNNaker detects basic blocks, which are, basically 48 chips integrating 18 ARM processors each.
Spinnaker can be programmed via ethernet connection. What is required is to pass through two software programming steps: on the host side, a Python script generates all the configuration files for the board; on the SpiNNaker side, C user-defined applications have run. Main libraries will be presented later on this chapter.

Before going deeper in the introduction, let's have a look on the main components within a single SpiNNaker chip, keeping as reference Fig. 2.10. From the logic point of view, each chip has shaped hexagonally, since it makes easier the implementation of routing algorithms within SpiNNaker. As said before, 18 ARM processor cores are within. One of them, on the top in the picture, is the *Monitor Processor* (MP), used for internal chip management and communication operators. It's up to it running the SARK operating system. Other 16 cores are labeled as *Application Processors* (AP), which will run the real C application. The last core, actually doesn't work, since it's reserved for manufacturing yield-enhancement purposes. On the left bottom side is highlighted the *embedded router*. In the end a 128kB SRAM and a 128MB of SDRAM are integrated within each chip, shared by all Application Processors.

Each chip has labeled by a couple of number, which are integer values (x, y) detected by an hypothetical Cartesian plane. Then, it could be useful draw some logic subregions, at chip level. It has been made, assigning chips to different concentric ring-layers, where hierarchy policy depends on the distance of a node from the axis origin. The chip $(0, 0)$ is the one keeps ethernet communication with the host or other SpiNN5 boards. Around, there are those chips composing ring 1 (the origin is ring 0): $(1, 0)$, $(1, 1)$ and $(0, 1)$. You may have a look at figure 2.13, where all the rings are depicted.

Some attention should be given to the architecture and operation of the routers mounted in each chip (1 per chip). Understanding their key features have been fundamental to realize this project, since the focus is over multicast routing rule generation and their compression.

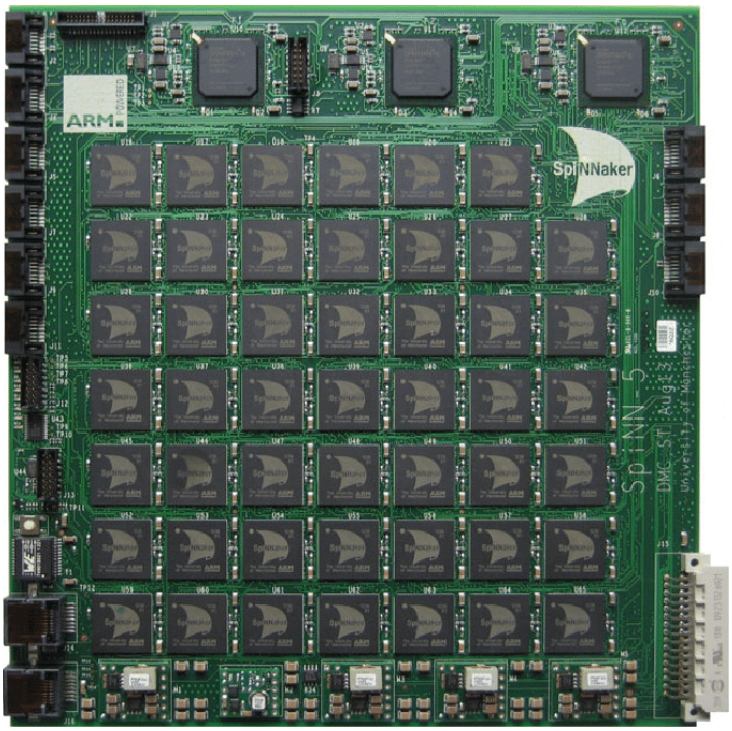

Figure 2.9: The Spin5 board. Here are well visible SpiNNaker chips, each of them composed of 18 ARM processors, for a total of 864 cores

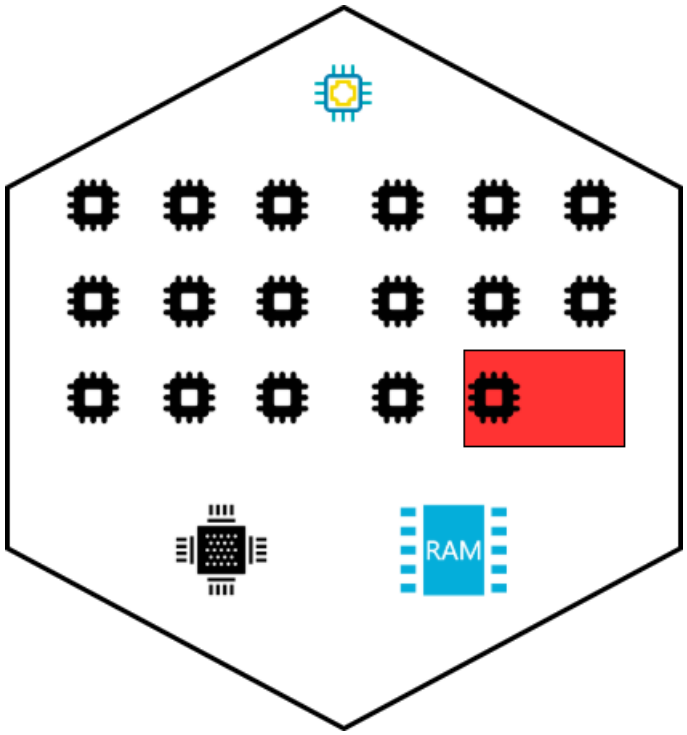

Figure 2.10: Logic overview of a SpiNNaker chip

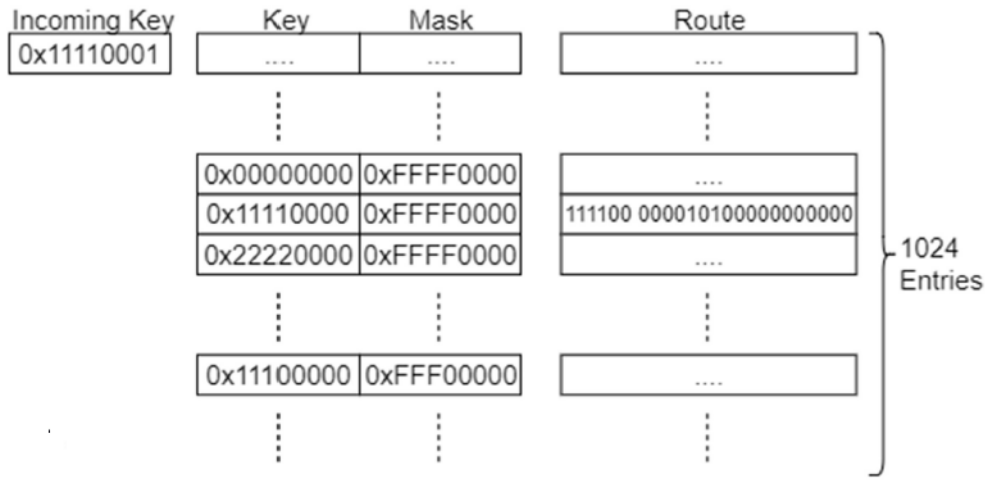

Figure 2.11: SpiNNaker routing table structure [2]

Mounted routers are CAM objects, acting as traditional caches. When they receive the routing key, a matching operation has performed, at the same time, all over the keys along the whole 1024 allocation matrix. Each router's row has composed of three parts: the key, the mask and the rule, Fig. 2.11. The presence of a mask, in a preprocessing phase, allows making kind of reducing operation among routing keys having common bits (mask bit set to 1), otherwise these bits are not considered (mask bit reset to 0). The incoming routing key passes through a bitwise-AND logic with the mask, whose result is compared with the key column. In case of equality, rule has enabled out, in order to drive the output signal decoders and multiplexers.

2.3 Message Passage Interface: the MPI

MPI has considered as a standard protocol to allow communication among clusters of computational units having distributed memory. It's very popular, mainly in the academic research world, since its own documentation has distributed in a free-license way. Actually, MPI is just a collection of rules and description of interfaces to be followed, in order to fit the standard. Being a specification, it is customizable by everyone, whatever the architecture taking part to communication. It's not needed that designers and engineers to provide architectural description of computation units, because MPI offers some high-level abstraction objects to avoid that step.

One of the most concept treated by MPI is the **communicator** object. It defines groups of units, or cores, such that to build a **world** of them. On the same hardware can run more communicators, each of them is totally independent from others. For each communicator some primitives are offered, such as: `MPI_Init`, `MPI_Comm_Rank`, `MPI_Send`, `MPI_Recv`.

To each communicator are assigned processes in the initialization phase, they are identified by a numerical **rank**. This value provides a kind of hierarchical order among processes of a communicator.

Last, but not least, MPI provides extension for C language, making the implementation very readable and easy to make. As matter of example, a very straightforward point-to-point MPI program [19] has listed next.

Listing 2.1: MPI P2P example

```
#include "mpi.h"
int main( int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if (myrank == 0) /* code for process zero */
  {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1,
             MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1) /* code for process one */
  {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99,
              MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

Lst. 2.1 describes a P2P communication MPI-based. This code has loaded into all the computational unit of the target universe, but it runs differently depending the assigned rank. That's a number assigned by MPI primitive, taking into account the communicator object, i.e. the "list" of all the computational units interested in. The unicast communicator has to assign just two ranks: zero and one. The code implements an unicast communication where the unit with rank zero produces the packet. If the unit's rank is one, then the packet is read and internally processed.
Lst 2.2 tries to present a classical P2P scenario between two processes in a Linux environment, where communication happens asynchronously due to the action of `writer` and `reader` blocking functions.

Listing 2.2: Linux Process P2P example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main () {
  FILE * stream;
  /* Create pipe place the two ends pipe file descriptors in fds */
  int fds[2]; pipe(fds);
  pid_t pid = fork();
  if(pid == (pid_t) 0) { /* Child process (consumer) */
    close(fds[1]);          /* Close the copy of the fds write end */
    stream = fdopen(fds[0], "r");
    reader(stream);
    close(fds[0]);
  }
```

```
  else {                    /* Parent process (producer)  */
    close(fds[0]);          /* Close the copy of the fds read end */
    stream = fdopen(fds[1], "w");
    writer("Hello, world.", 3, stream);
    close(fds[1]);
  }
  return 0;
}
```

2.4 Libraries for development

Some libraries are involved to guarantee both correct configuration and behaviour of SpiNNaker. They are organized in a hierarchical manner as depicted in Fig. 2.12.

- **SpinMPI**
- **Spin2API**
- **SpinACP**

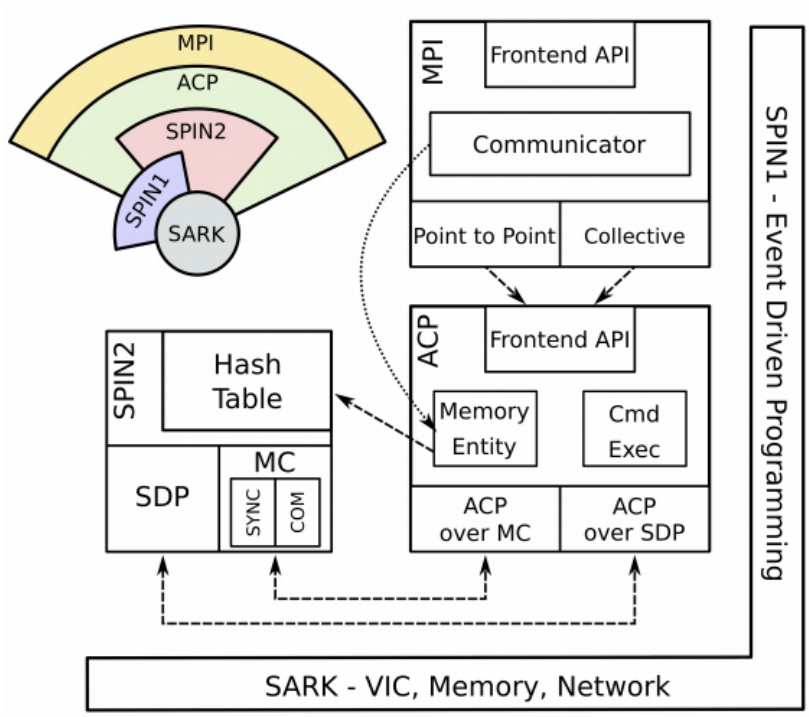

Figure 2.12: Hierarchical logic view of API libraries [3]

SpinMPI: Previous papers [20] have presented a customized implementation for the MPI algorithm, adapted to run on Spinnaker. It has been designed to emulate spiking neural networks, not to run general purpose algorithms. The Bioinformatic team tried to overcome this limitation, by doing that.

Let's spend some words about vanilla MPI. The Message Passing Interface is a standard aims at providing and easy-to-use software stack, to implement multi-threading algorithms among memory distributed clusters of computers. It offers portability extension for C and Fortran. In fact, SpinMPI is fully written by C language. Moreover, MPI is really widespread in the academic community, making it the golden protocol for research purposes. It deals with all communication models: unicast, broadcast and multicast. So far, SpinMPI offers full unicast and broadcast functionalities; multicast communication is still in working progress.

SpinMPI focuses on synchronization, in order to ensure correctness among multi-threading propagations. While point-to-point doesn't require it, both multicast and broadcast communications need that step. At the state-of-art, the synchronism mechanism exists for broadcast case; a possible realization filling multicast scenario will be proposed in next chapter. Broadcast synchronization relies on three level hierarchy. Each level is controlled by a manager, or a group of them. It's up to it (them, if more) collecting all synchronization packets. When it happens, the master generates a further synchronization message to the upper layer Figs 2.13 and 2.14.

1. **Chip level**: A $SYNC_1$ packet is sent to all cores within a SpiNNaker chip. When all synchronization packets are collected, a $SYNC_2$ packet has prepared and sent to the upper level.

2. **Ring level**: Chips having the same distance from the origin build a ring. The chips labeled with x-coordinate equal to the y one are the $SYNC_2$ managers, meaning they are the actors up at collecting the synchronization packets of the group. Each ring master knows how many $SYNC_1$ should be produced. When they receive all of them produce a $SYNC_3$.

3. **Board level**: All 2-level managers send $SYNC_3$ packets toward the third level manager, which is the chip on the origin. When all 2-level managers do that, the synchronizer of the upper level sends, broadcasting, a $SYNC_{unlock}$, or ACK. It means that synchronization phase is over.

Spin2API: it works as middleware between SpinMPI and lower layers. This library is an improvement of Spin1API, the native library directly designed by Manchester University. Spin2API provides frameworks for SpiNNaker Datagrams. It's up to this library the filling of routing table. Moreover, Spin2API implements broadcast connection and synchronization methods. Within this layer is also described format of messages. This concept will be recovered in the MCM section in the next chapter.

SpinACP: as well as Spin2API, SpinACP is a middleware between SpinMPI and lower layers. ACP stands for Application Command Protocol, exploited to send and to receive commands between the host and the SpiNNaker nodes. The offered functions cover: network command management, *ACPoverSPD* and *ACPoverMC* (dedicated at message reconstruction), memory entities management, which are described as structures within an hash table. Parsing of messages it's up to this library.

Libraries presented so far are all the ones running on the physical board, on Spinnaker itself, spread among all the cores. There exist another software stack allows configuring

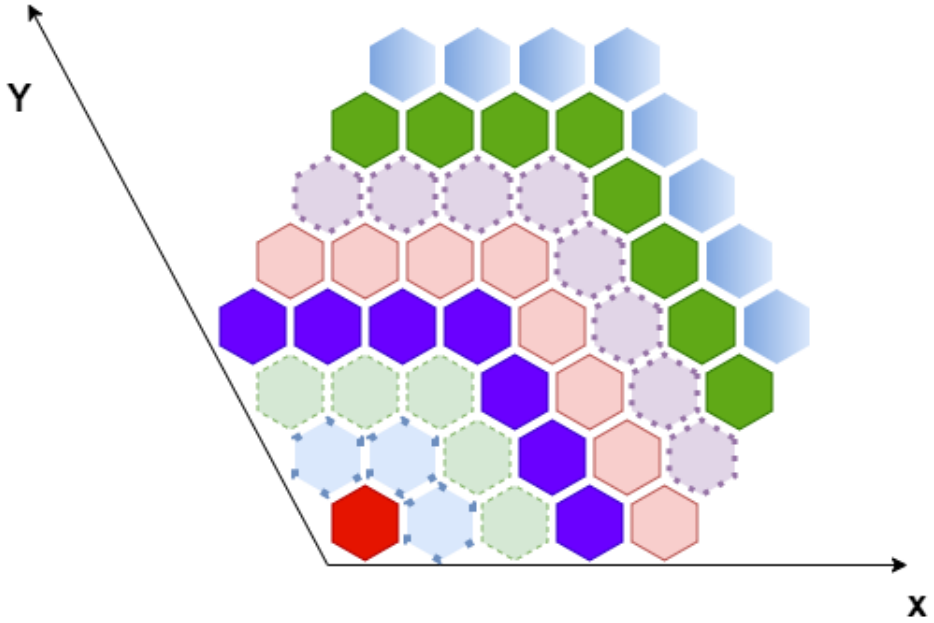

Figure 2.13: Logic overview of a SpiNN-5 board

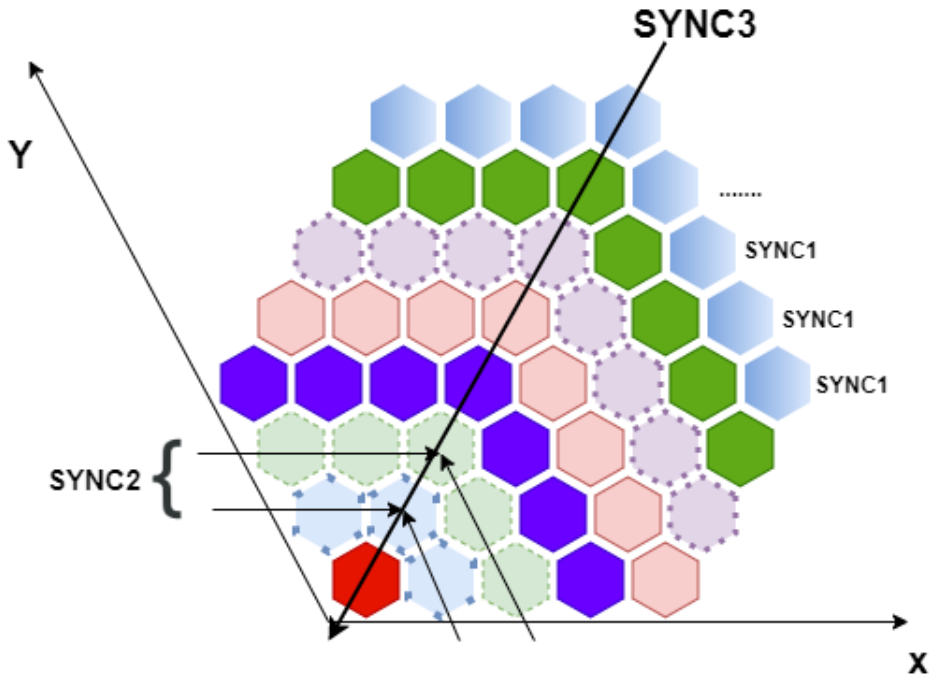

Figure 2.14: Synchronization mechanism: each single chip is a first level manager, any chip along the diagonal is a second level manager, finally, chip $(0,0)$ *is the third level manager*

it and provides an higher level management: **sPyNNaker**. Actually sPyNNaker is a front-end package offering a lot of sub-modules for Spinnaker configuration and/or simulation. Main integrated classes are: PyNN [21], SpiNNMachine [22], SpiNNMan [23], PACMAN [24], SpiNNFrontEndCommon [25].

Chapter 3

Methods

Nowadays, the SpinMPI stack can handle both unicast and broadcast communications, leaking of multicast scenario. Actually, looking for a mapping satisfying them it's not so hard; since unicast expects just a point-to-point path and broadcast's channel is always identified by the same ring-based structure.

This "hardwired" strategy cannot work for multicasting. First of all, choosing the same path to connect two nodes may lead to an excessive throughput both on the line and on the forwarding routers. Another reason, surely, regards the loss of the big advantage of a neuromorphic architecture, strongly parallel execution oriented. It would be better manage two non overlapped multicast groups, making their constructions dynamic depending on the previous mappings. This is the very basic idea behind methods will be presented in this chapter. The main goal aims at mapping, tens or hundreds, multicast groups in order to do makes distribution heterogeneous; bottleneck may occur among too stimulated edges.

In this chapter, the Fig. 3.1 chain will be described. As first, a kind of numerical analysis has presented, just to provide the reader a quantitative view about complexity of managed objects. Then, both all input and output files are described. At this point, the user owns the knowledge to understand implemented algorithms. Mapping deals with converting textual information in graph objects. From them, routing rules have being generated and then compressed to be stored in the routers. In the meantime, SpinMPI rules for multicast communicators have been performed. When all the information have been fitted in all the SpiNNaker routers, simulation starts and statistics have produced in output.

The other main task *mapping.py* has to accomplish deals with SpinMPI's barrier function by means of generating the binary synchronization rules to be filled in all the 48 routers in SpiNNaker. In order to do that, a synchronization mechanism has been designed, proposing a hierarchical organization in subgroups of nodes; in such a way master synchronizers at a certain level, turns into slaves nodes to be synchronized at the upper one.

Then, *mapping.py* aims at filling routers with routing and synchronization rules and performing compression where it's required, in order to avoid consuming the already limited memory space.

In the end the simulator (*main.py*) evaluates the correctness for each enabled group, checking both for the absence of loops in the multicast subgraph and for the consistency of built rules. In order to run, simulator requires of router contents, the graph-based description of the neuromorphic board, the (sub)set of multicast groups and a schedule

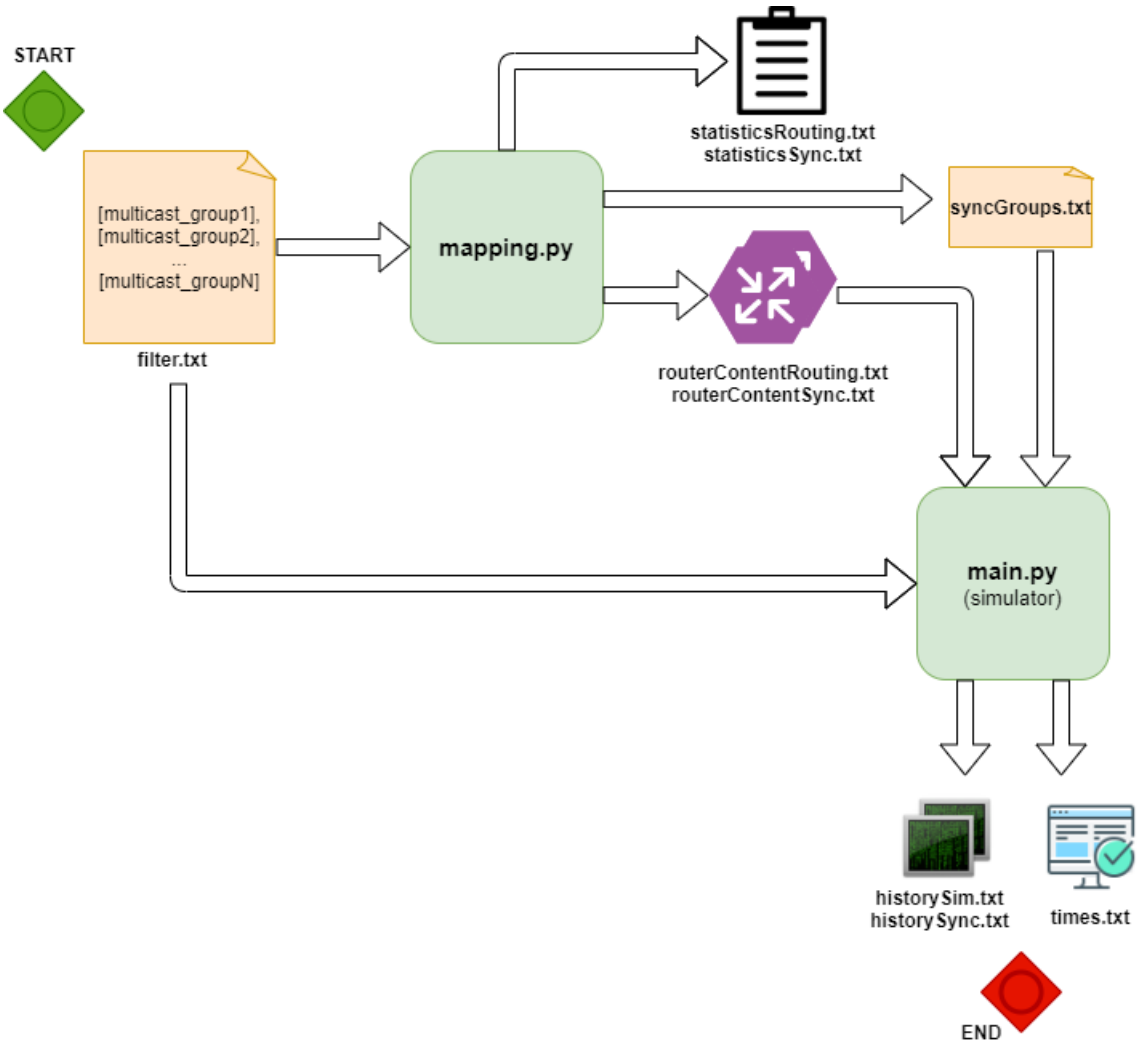

Figure 3.1: Inputs and outputs flow chart in the environment

of tasks that each core should perform. Simulator has integrated in *main.py*.

The whole project has been developed in Python3.7.5 on a VMware Workstation 15 Player virtual machine mounting Ubuntu 19.10.
To handle graphs object, networkx Python package has been really useful, offering ready-to-use methods. Modelling SpiNNaker got trivial by this one.
Rest of the code has been developed from scratch.

3.1 Mathematical issue

Before going on methods, it would be interesting to provide some quantitative number, making multicast grouping not so trivial. SpiNNaker counts 48 chips, each of them embedding 18 general purposes core: 864 as total. The minimum number of cores to draw a multicast group is 3, the upper bound has constrained by all of them minus 1 (otherwise we would fall in the broadcast case). When you build groups, you are allowed

to assign the same core to different ones, causing possible and very common overlaps. Mathematically you can model as follows. The total number of groups, having same size k, choosing from all n cores, comes from the binomial coefficient Eq. 3.1.

$$C(n,k) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \tag{3.1}$$

Obviously, saying that all multicast groups have the same size k is a very hard assumption, false in the majority of the cases. Actually k may vary in range $[3, 864-1]$. We can reformulate the previous equation in this manner:

$$\sum_{k=3}^{863} C(864,k) \tag{3.2}$$

Let's report some quantity to compare that.

Eq. 3.2 produces a number that is higher than

- total number of elementary particles in the known universe $\sim 10^{87}$
- the Googol number $= 10^{100}$
- the total number of possible chess games $\sim 10^{123}$

Relative plot has drawn at Fig. 3.2.

Each router can stores 1024 rules only! Obviously, it's just playing with Maths: feasible real solution satisfy some hundreds of multicast groups.
After the mapping step, routing rule compression comes. As said in chapter dedicated to SpiNNaker specifications, each router has composed of 1024 rows, one per rule, shared between unicast, broadcast and multicast. Moreover, routers must be filled with routing rules devoted for synchronism mechanism, which is another item of this chapter.

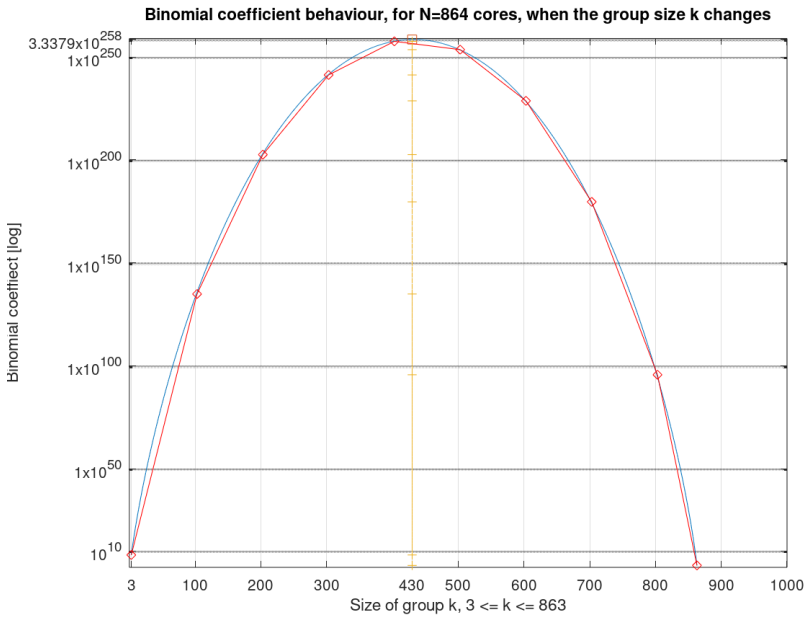

Figure 3.2: Plot of the function $\sum_{k=3}^{863} C(864,k)$

When routers have been filled, a simulation is performed. To do that, a software tries to emulate the hardware, trying to replicate delays and workload. The output returns both synchronization and simulation times; moreover, for both of them, it's possible to plot an heatmap of the cortical activity, router by router, cycle by cycle.

Before going into implementation details, it should be better to provide a description of kind of messages, and their compositions, travelling all around the board, both in synchronization phase and in normal communication mode.

3.2 Multicast Communication Middleware messages

Missing the description of unicast and broadcast messages, you can find in reference, in this section the multicast scenario has treated only.
Packets are basically composed of three parts:

- The **routing key** (32 bit)
- The **payload** (32 bit)
- Some additional options (8 bit)

The routing key is the part of the MCM message that it's used to match router lines, in order to identify directions packet has to take. Payload, as the name suggests, it's the really information content exchanged by neurons. The options are a set of flags enabling some functionalities on the packet; however they have not been considered here and have been mentioned just for completeness of information.
Then, you need of discriminating between "classic" multicast packet and the synchronization one. In fact, they have some small differences between, on the routing key (Fig. 3.3).

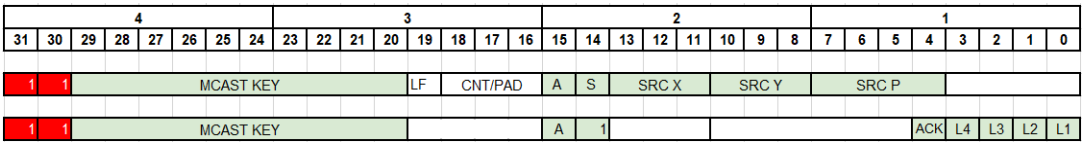

Figure 3.3: MCM format for multicast message on top and synchronization on bottom

Bit[31:30] set at "11" states that these messages are referred for multicasting purposes. Unicast and broadcast are identified by different encodings. Again, both of them share Bit[29:20], representing the multicast identification code. Looking at classic packet, you set LF and CNT/PAD bits. LF stands for Last Fragment, it's used by cores to reconstruct the full payload if only 32 bits are not sufficient and information has split into more messages. CNT/PAD are used as timeout. However, these bits are not interesting to determine routes, so they are simply avoided by putting at 0s in the routing mask. Bit[15:14] identify that message as Acknowledge or Synchronizer. In the multicast message, Bits[13:4] fill information about the source of this packet, where x and y are the cartesian coordinates of the chip on the chessboard and p is the logic identifier of the core in that chip, assuming value from 0 to 15. Looking at synchronization packet instead, Bits[4:0] assumes some meaning: L4 identifies this packet as a level 4 synchronizer, L3 identifies this packet as level 3 synchronizer and so on. You can find more information on how synchronism mechanism works in the relative section of the current chapter.

3.3 Inputs and outputs

Logically, the order of operations to be run are mapping before and simulation after. You need execute *mapping.py* and then *main.py* by the command line. Keeping as reference Fig. 3.1 and Fig. 3.4 the flow description from input to output has shown.

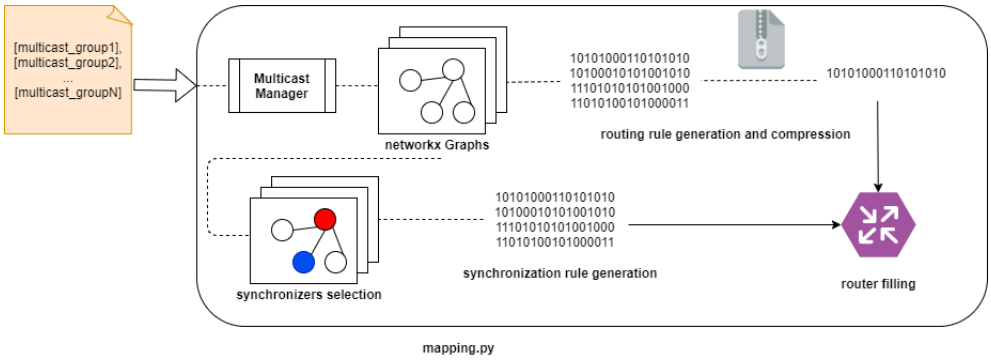

Figure 3.4: Main steps of mapping.py

Both of *mapping.py* and *main.py* need, as input, a textual file, here called *filter.txt*. It contains, on each line, the list of cores to build the multicast group. An example has reported in Lst. 3.1. The identification number for that group has assigned as the line index of that file.

Listing 3.1: Example of filter.txt where 10 multicast groups are described

```
[121, 123, 373, 402, 525, 536, 601, 673, 745]
[13, 31, 80, 103, 130, 281, 296, 313, 319, 321, 388, 501, 572, 628]
[42, 58, 66, 96, 196, 250, 280, 310, 331, 528, 545, 561, 651, 672]
[155, 207, 209, 328, 383, 529, 555, 849]
[42, 56, 135, 194, 238, 314, 345, 351, 366, 406, 567, 712]
[196, 341, 347, 430, 527, 543, 732, 778, 832]
[146, 161, 170, 197, 349, 386, 471, 579, 585, 604, 693, 840, 849]
[15, 31, 66, 88, 237, 371, 387, 388, 579, 593, 610, 725, 779, 812]
[12, 75, 98, 295, 337, 346, 412, 459, 524, 609, 653, 659, 744, 751]
[12, 672, 713, 857]
```

Actually, *filter.txt* is the only input *mapping.py* accepts. On the output side, you can set the boolean parameter to enable the plot for each multicast group on the SpiNNaker model graph.

Listing 3.2: Enable plot of multicast groups as images

```
...
MM = MulticastManager()
MM.loadGroups("filter.txt")
MM.generateMinimumSpanningTrees(True) # <---- False: no to plot
...
```

If *True*, for any multicast group, detected in *filter.txt*, will be produced a png image having the following name format *idGroup_c*[*center*], where idGroup is the identification number for the multicast set and center is the node in the subgraph having the lowest

eccentricity, it may be a list.
Let's have a look at the outputs:

- *routerContentRouting.txt*: router by router are listed, as tuples, the triplet routing key, mask and rule, for classic multicast routing (Lst. 3.3), Fig. 3.5
- *routerContentSync.txt*: router by router are listed, as tuples, the triplet routing key, mask and rule, for multicast synchronism (Lst. 3.3)
- *statisticsRouting.txt*: router by router the number of rules involved, for classic multicast routing (Lst. 3.4)
- *statisticsSync.txt*: router by router the number of rules involved, for synchronism (Lst. 3.4)
- *savingsSync.txt*: multicast group by multicast group, the percentage of savings, in term of rows in routers, between two synchronism mechanisms. The former, which takes into account geometrical properties of the mapped minimum spanning tree, the latter more efficient, which discriminates, when possible, avoiding the usage of forwarding-only nodes as synchronizers (Lst. 3.5)
- *syncs.txt*: multicast group by multicast group, all synchronization subset are reported. The format used is dictionary-like, where for each element the key is the synchronizer and the list of values are nodes to be synchronized at that level. (Lst. 3.6)

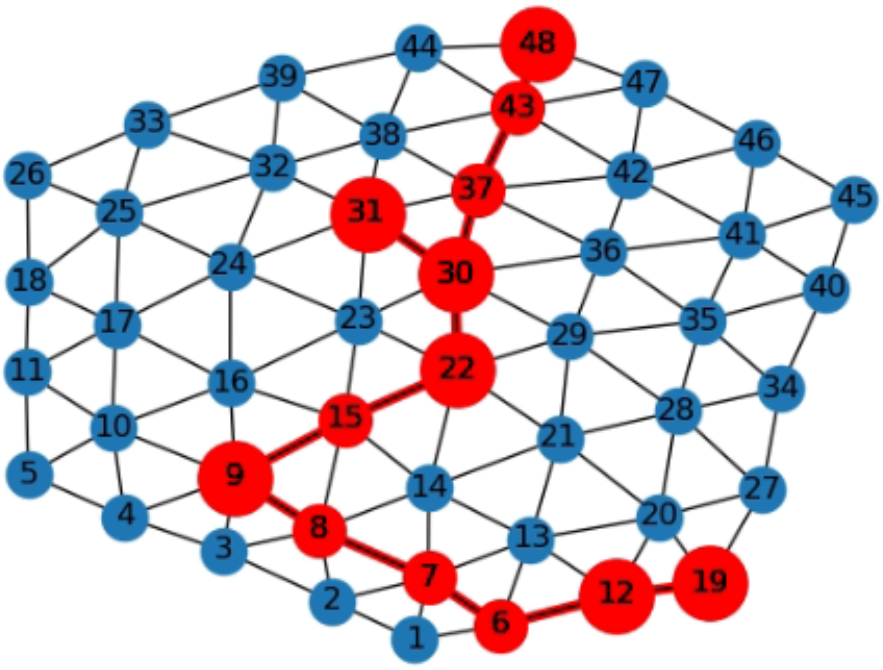

Figure 3.5: Plot generated from networkx. It names 3_[9,15].png. Bigger red nodes are the chips where are contained cores listed in filter.txt, smaller red nodes are chips inserted in the multicast groups in order to ensure connectivity among the bigger ones. Blue nodes are the ones that are not interested for the current configuration.

Listing 3.3: Format of routerContentRouting.txt and routerContentSync.txt

```
(0, 0)
(11000000000100000000000010110000, 11111111111100000011111111110000,
101000 - 00000000000000000)
(11000000000100000000000100000000, 11111111111100000011111100000000,
100000 - 00000000000100000)
(11000000000100000000000000000000, 11111111111100000001011100000000,
001000 - 00000000000100000)
(11000000000100000000000000000000, 11111111111100000000000000000000,
001000 - 00000000000100000)
(1, 0)
(11000000000100000000100010110000, 11111111111100000011111111110000,
010100 - 00000000000000000)
(11000000000100000000000000000000, 11111111111100000011111100000000,
010000 - 00000000000100000)
(11000000000100000000000100000000, 11111111111100000011111100000000,
010000 - 00000000000100000)
(11000000000100000010000000000000, 11111111111100000011111100000000,
000100 - 00000000000100000)
```

Listing 3.4: Format of statisticsRouting.txt and statisticsSync.txt

```
(0, 0) Size: 90
(1, 0) Size: 126
(2, 0) Size: 198
(3, 0) Size: 155
(4, 0) Size: 99
(0, 1) Size: 139
(1, 1) Size: 194
(2, 1) Size: 283
(3, 1) Size: 275
(4, 1) Size: 185
```

Listing 3.5: Format of savingSync.txt

```
0 Save: 38.5
1 Save: 26.1
2 Save: 33.3
3 Save: 30.8
4 Save: 20.0
5 Save: 45.5
6 Save: 23.8
7 Save: 21.7
8 Save: 41.7
9 Save: 43.8
10 Save: 53.3
```

Next listing reports *HighLevelSync*, *MainSynchLevel*, *CommLevelSync* and *LowwSyncLevel*, as levels of synchronization at 4 (the top), 3, 2 and 1 (the bottom), respectively.

Listing 3.6: "Format of syncs.txt"

```
IdGroup:0
HighLevelSynch:{23: [30, 22, 23]}
MainSynchLevel:{30: [30], 22: [21, 22], 23: [23]}
CommLevelSynch:{21: [7, 34, 21], 22: [42, 22], 23: [23],
                30: [38, 30]}
LowwSynchLevel:{7: [7], 21: [21], 23: [23], 30: [30], 34: [34],
                38: [38], 42: [42], 22: [22]}
IdGroup:1
HighLevelSynch:{16: [17, 9, 15, 16]}
MainSynchLevel:{17: [17], 9: [5, 8, 9], 15: [42, 28, 31, 22, 15],
                16: [16]}
CommLevelSynch:{28: [35, 28], 31: [38, 32, 31], 8: [6, 1, 2, 8],
                22: [22], 16: [16], 5: [5], 42: [42], 17: [18, 17],
                9: [9], 15: [15]}
LowwSynchLevel:{1: [1], 2: [2], 5: [5], 6: [6], 8: [8], 16: [16],
                17: [17], 18: [18], 22: [22], 28: [28], 32: [32],
                35: [35], 38: [38], 42: [42], 31: [31], 9: [9],
                15: [15]}
```

That's it on *mapping.py*. Some of its outputs are taken by *main.py* to fill simulation data, such as *routerContentRouting* and *routerContentSync*. To register information for synchronism, *syncs* has parsed as well.
Simulation produces three files:

- *histortSim.txt*: it contains a matrix where on the abscissa are listed the 48 chips, instead along ordinate axis the instant of simulation. The value in matrix is the amount of tasks in the queue on that chip at that moment. (Lst. 3.7)
- *historySync.txt*: it contains a matrix where on the abscissa are listed the 48 chips, instead along ordinate axis the instant of synchronization. The value in matrix is the amount of tasks in the queue on that chip at that moment. (Lst. 3.7)
- *times.txt*: it's the more relevant simulator output. It reports number of cores involved in, the synchronization time, the simulation time and the μ coefficient defined in Eq. 3.3. Format file in (Lst. 3.8)

$$\mu = \frac{T_{sim}}{T_{sync}} \tag{3.3}$$

Discussion on μ will be recalled later. However it's quite intuitive to guess that values higher than 1 are more convenient. The reason stands on the fact that the choice in using a platform like SpiNNaker depends on the need for a high-parallel computing platform. Since neurons must be synchronized, if the machine takes more time in that rather than running its own application, maybe it could be better either change application or change board.

Listing 3.7: Format of historySim.txt and historySync.txt

```
t 1 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 2 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
```

```
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 3 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 4 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 5 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 6 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 7 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 8 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 9 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16, 11, 11,
11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
t 10 [11, 10, 16, 13, 13, 13, 11, 9, 13, 10, 12, 14, 12, 13, 10,
10, 9, 12, 12, 12, 16, 12, 15, 10, 11, 14, 8, 15, 8, 16,
11, 11, 11, 8, 9, 7, 9, 18, 17, 11, 14, 14, 11, 7, 11, 12, 12, 17]
```

Listing 3.8: Format of times.txt

```
NCores:281  Syn T: 113  Sim T: 989  Mu: 8.8
```

3.4 Mapping algorithm

The Multicast manager is an ad-hoc class, designed to manage multicast groups. Its main purpose is the construction of a Graph object taking into account the mapping of previously laid out groups. In fact, two algorithms have been proposed for the tree generation:

- Minimum Spanning Tree method by *networkx Python package* [26]
- Heuristic iterative algorithm based

Quantity results will be presented in the next chapter.

The mapping algorithm is the entry point of the strategy. That is totally handled by **Multicast Manager** (MM) class, which is implemented in *multicast.py* and instantiated in *mapping.py*. Basically, it reads the *filter.txt* content and returns *networkx* graphs, each of them labeled by the identification group code. MM replicates Spinnaker's connections among nodes by means of a Graph object. Each node comes out with three attributes:

- Coefficient ς: total number of times that node has crossed considering all minimum paths for all possible couples.

- Eccentricity ϵ: after computing all possible paths starting from the current node to all the others, the eccentricity is a number equal to the maximum path length among those

- Effort ξ: is used as counter. It increases by an amount equal to the coefficient when the current node has been inserted in a multicast group either as interested node or as forwarding only

Actually, ς and ξ have been declared and used for a while, but no more dealt with in next version of this implementation. In fact, previously, the mapping algorithm took care about ς and ξ; currently it's used only one attribute stays on each edge:

- Weight ω: is used as counter. It's increased by 1 every time that edge/connection has inserted in a multicast graph representation.

At the Multicast Manager initialization, all ω are set to 1.

When you run the multicast group mapping methods, actually the MM allows to decide what kind of algorithm to enable. Two of them have been thought: one optimization based minimum spanning tree methods, one heuristic based.
The former exploits the minimum spanning tree method from networkx library. From documentation "*Return a minimum spanning tree or forest of an undirected weighted graph. A minimum spanning tree is a subgraph of the graph (a tree) with the minimum sum of edge weights.*"[26] It means that, given the current multicast group m_i, not yet mapped, and given Υ the mapping function returning as a mapped subgraph M_i, the behaviour of Υ depends on the history (Eq. 3.4). In the sense that the next mapping will depend on how the previous multicast groups, at very beginning, were placed.

$$M_i = \Upsilon(m_i) \qquad \Upsilon = f(M_{i-1}, M_{i-2}, \ldots, M_0) \tag{3.4}$$

Let's focus on the heuristic algorithm instead. It starts looking for the origin, or center. The *find_origin* φ is a function (Eq. 3.5) selecting the node, in the subset passed as input, with the minimum eccentricity, such that

$$\varphi(setOfNodes) = min\{\epsilon(n_i)\} \qquad i \in [0, size(setOfNodes) - 1] \tag{3.5}$$

The origin has assigned to *current_node* (CN) variable and a mask identifies it as "already taken". From CN, closest nodes not yet allocated are listed and, for each of that, are computed all minimum paths (from CN to closest node). Among these paths, the one having the minimum weight, as sum of weights along its own edges, is added to the multicast graph. Then the origin has recomputed among nodes of $setOfNodes$ not yet connected. This runs until $setOfNodes$ gets empty. Briefly the **pseudocode of**

heuristic mapping algorithm:

$connected = \{\}, G = Graph(), setOfNodes = multicast\_group$
while not($setOfNodes = \emptyset$)then
  $origin = \varphi(setOfNodes)$
  $CN = origin$
  $connected.add(CN)$
  $closests = find\_closests(CN, setOfNodes, connected)$
  $paths = find\_min\_paths(CN, closests)$
  $best = min\_weight(paths)$
  $fornode, edgeinbest\{G.add(node, edge)connected.add(node)\}$
  $setOfNodes = -connected$
endwhile
return G

3.5 Routing rule generation and compression algorithm

At this step, all multicast graphs have been generated and routing rules ready to be computed. This process works iteratively along all multicast groups, one at a time. We have to configure all chips being in the current subgraph, regardless of it's part of the original multicast vector as from *filter.txt* or it's just a forwarding node, used only to ensure connectivity. The current router, to be elaborated, has named *config_chip CC*. It should be to understand that CC has to be filled with information of all other nodes in the multicast group: these name *pivot_chips* (PC). Basically, a CC treats as PC any node which is able to act as a source of packets. It means that, in the multicast graph, all nodes must be configured, but not all act as pivot. The reason is quite simple. A forwarding node only serves to ensure the connected component: it will never generate packets, but will forward from others else.
A rule has composed of 24 bits. Top 6 deals with forwarding of the input packet toward the i-th chip port (if $p[i] == 1$), or not (if $p[i] == 0$). Other 18 bits regard the 18 cores within the CC, forwarding to the i-th core(if $c[i] == 1$), or not ($c[i] == 0$). Notice that for i_0 and i_{17} will be never set the bit to '1', as the former represents the monitor processor and the latter redirects to the shadow core.
For the current CC, are listed all possible PCs. For any PC two data are extracted

- The **source key**: it's a 6 bit string, encoding the cartesian coordinates of pivot logic id. Note that source key doesn't incorporate the logic id of the source core.
- The **encoding for port forwarding**: it's the 6 bits string encoding highlithing which ports are enabled in order to ensure packets propagation in the rest of the multicast group

When both of these information come out, source keys are grouped by port forwarding string. It means that the algorithm creates as many clusters of PC, as many are (distinct) port patterns in the current *config_chip*. Then, for each of these clusters, keys are again

distributed among pivot regions. As depicted in Fig. 3.6, these areas has to be thought as "relative", not absolute placed on the SpiNNaker. Pivot regions are geometrical rules, only. The number and the chip in each of them, depends on the CC; each of 48 chips detect 8 different neighboor areas, which are functions of the CC itself. It's possible that, given a CC, some of its own pivot regions are empty; this is the case of nodes along SpiNNaker boundaries, for instance.
Finally, compression runs on all over the pivot regions and produces the compressed source key k and its own mask m. For all the length of a generic source key, it's checked if the i-th bit, among all the keys, is identical ($k[i] \leftarrow 0/1$, $m[i] \leftarrow 1$) or not ($k[i] \leftarrow' X'$, $m[i] \leftarrow 0$). At most, 8 couple key/mask will be produced, one for each region. In case no *pivot_chips* falls into a region, the related couple has not generated and the algorithm, simply, continues.
Before inserting all the couples in the CC's router, they pass through a sorting step. Couples have sorted taking into account the number of "X" in k. They are sorted in a growing order manner. This choice depends on the physical working mode of routers. They are content addressable memory, CAM for short. The input of a CAM has matched with all the lines in the same moment, but in case of multiple hits, the upper in memory has extracted. A key k with no "X" leads to the less generic rule: that row matches exactly full key itself. The higher the number of don't care bits, the more generic and the higher the probability of matching. If the order was taken reversely, it would be almost zero the chance of a hit with the most accurate rule. Below has depicted the **pseudocode of routing rule compression algorithm**:

for CC in$multicast_graph$ then
  keys = {}, ports = {}, clusters = {}
  for$PC in multicast_group$then
    $keys.add(coord(PC))$
    $ports.add(forwardingPorts(PC))$
  endfor
  $clusters = keys.groupBy(ports)$
  for port in ports then
    $nodes = extractByRegion(clusters[port], 1)$
    $k_1, m_1 = compression(nodes)$
    $nodes = extractByRegion(clusters[port], 2)$
    $k_2, m_2 = compression(nodes)$
    ...
    $nodes = extractByRegion(clusters[port], 8)$
    $k_8, m_8 = compression(nodes)$
    $router[CC] = sort(\{(k_1, m_1), (k_2, m_2), \ldots (k_8, m_8)\})$
  endfor
endfor

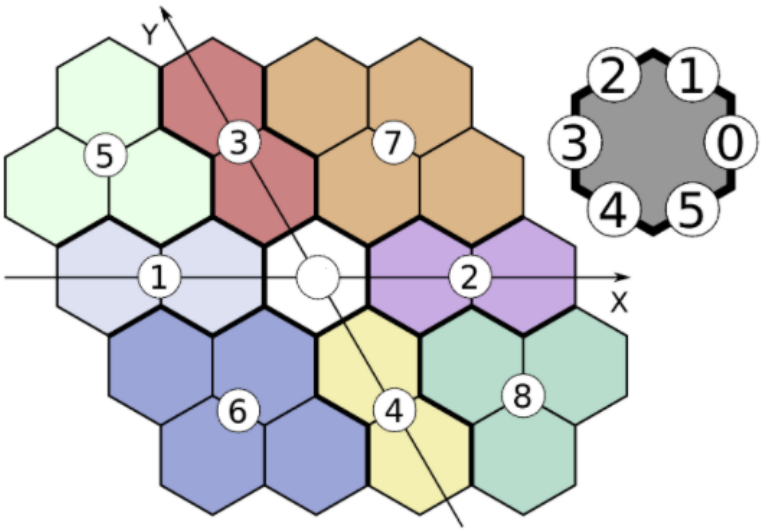

Figure 3.6: All possible pivot regions, depending on the config_chip. At right, a model of the six ports surrounding a router.

3.6 Synchronism mechanism

In parallel computing architectures, exploiting MPI, synchronism has used as a barrier. It means that, in order to avoid unbalanced growth of distributed processes, after a certain number of execution cycles, nodes have to wait for each other, before of going on running. Given a set of nodes, they exchange, with each other, some synchronization packets, where one has elected as master and all other else as slaves. This protocol works in a total asynchronous manner and spreads on all over the board. It's said **Globally Asynchronous**. In previous publications [20] have been explained reasons led to avoid the usage of a single synchronizer to enhance multi-synchronization layers, in order to distribute better the workload and the flow of data that would be sent to a single node. The MPI stack for SpiNNaker, developed at Politecnico di Torino, handles broadcast synchronization only. In this section, the multicast one solution has proposed.
Unfortunately, broadcast synchronization mechanism cannot be re-used for multicast groups. When you synchronizes all nodes, you can select in advance who are synchronizers of any level. Nodes layout doesn't change in time! This is not true for multicasting. At the beginning of the current chapter, the total number of possible combination had been computed. Two different multicast sets may be, geometrically, totally uncorrelated. So, the choice of who are the synchronizers and the list of sub-nodes treated as slaves must be done at run time, after mapping. The second big difference with respect to broadcast scenario, comes in the number of hierarchical layers: 4 in place of 3. Again, the reason stands on the irregular geometry and uncorrelated features among multicast groups; so an higher level of flexibility would be more reliable.
Let's explore which are criteria in selection of synchronizers, layer by layer. Layers are enumerated from 1 (lowest) to 4 (highest). Given a layer i, and one of synchronizers at i, called $sync_j i$, the total amount of signal $sync_j i$ has to receives before triggering the next level $i+1$ message, it's equal to the number of chips are allocated in the subgroup

where $sync_j i$ works. If $i = 1$ instead of dealing with chip, we deal with cores. If $i = 4$, the next level signal is the *all-free* message, sent by ACK to all members of multicast group.

Level 1: as well as in broadcast, first layer of synchronization works at core level. It means that no synchronization packets are sent out of the chip, but it's exploited the internal shared memory in order to implement the barrier among cores. Only cores building multicast group are considered, not the whole 16-vector.

Level 2: it treats no more cores but chips. Sync2 node and its own slaves come out from an iterative procedure. Substantially, multicast graph are always trees; it means that they own a root and some leafs. The goal is to divide the tree into some non overlapping sub-trees, where for each of these, the root is the sync2 and all the other else within are nodes to be synchronized by it. Before describing that algorithm, it should be repeated that nodes and edges can turn overloaded, leading to possible loss of messages. This scenario must be avoided. About edges, propagation of synchronization packets has always an upward propagation, to the root of each sub-tree; it means that you will never see "ping-pong" situation for synchronism mechanism. About nodes, the selector algorithm uses a soft-threshold to set an upper bound on the number of elements in a single sub tree. This value has set to 4, but often are accepted also 5 elements. However, as said, it's a soft threshold. It may be possible to tune that, in order to accept bigger sub groups; that is the case of long straightforward chains, that, with low thresholds, will be splitted, but are kept untouched setting higher values. About that, some results are reported in the dedicated chapter, looking at difference between small and more populated sub trees.
Let's have a look at the algorithm itself. At the beginning, all the nodes having *more than* 2 active ports are collected. Then, starting from the deepest, they build a sub tree where they are roots and leafs are the directed sons, Fig. 3.7. Remind that trees cannot be overlapped. At this point, all crosses have been removed and only straightforward chains stay to be allocated. Here, threshold is acting. Starting from the deepest non allocated node, the length of the chain has computed. If the sum between its size and the sub tree above is less than the threshold, then merge them, otherwise draw a new set equal to the chain, Fig. 3.8.
A very common case deals with chain long 1. It may be a leaf of the main minimum spanning tree or a node between two already allocated subsets. As design choice, instead of having trees made up of a single node, it has been preferred to merge, in any case regardless of threshold principle, with the closest group above, Fig. 3.9.
When the algorithm is over, groups have been defined, where sync2s are all the detected roots, Fig 3.10. You noticed that no distinctions have been done on the nature of each node, whether it's an interesting or a forwarding one. It would be better no to synchronize nodes which don't generate information. Hence, no matter about roots: they are kept, regardless of nature. For all other else, the filter has applied and forwarding nodes are simply excluded from synchronism mechanism.

Level 3: The rule to detect sync3 is quite simple. Any direct son of the root is a lever 3 synchronizer. To each of these converge signals coming from sync2s of that branch. The idea beyond this design choice comes from the fact that, in order to reduce the workload on the root, which is the level 4 synchronizer, an intermediate layer trying

cutting the tree in balanced ones could be useful. Both interesting and forwarding nodes may become sync3s.

Level 4: As anticipated, the sync4, only one per multicast group, is the root of the main tree. Given a set of n nodes in a minimum spanning tree, it's possible to draw it in n different manners, choosing at each iteration a different node as root. Here, what's done is looking for the most central node. This information is easy to extract, by applying the minimum eccentricity hunter φ, already treated when the mapping algorithm was presented. The only modification stands on the nature node. φ returns the object with minimum eccentricity which is in the multicast group. It cannot be just a forwarding point.

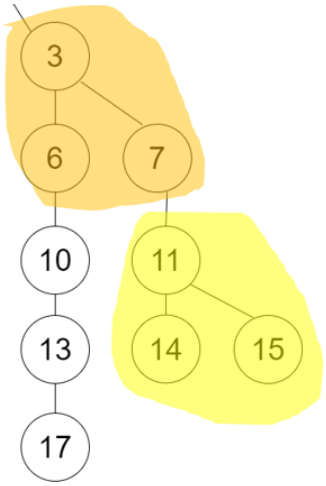

Figure 3.7: Sync2 detection algorithm: building proto-subtrees

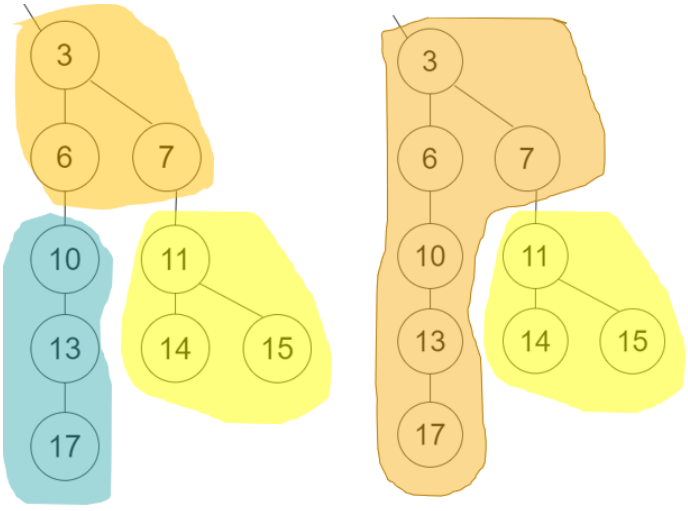

Figure 3.8: Sync2 detection algorithm: leftmost tree shows the case of a threshold lower than 4. Rightmost one shows what happens when threshold gets increased, allowing long chains to be merged with other groups.

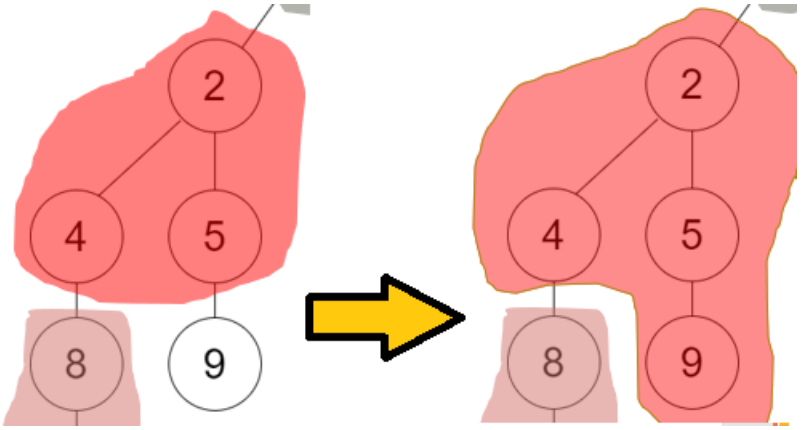

Figure 3.9: Sync2 detection algorithm: leafs are automatically merged up

Reported below the **pseudocode for level 2 synchronizers search.**

$nodes = \{orderByDepth(mst)\}, subTrees = \{dict\}$
for node in nodes then
  if nActivePorts(node) ≥ 3 then
    $subTrees[node] = \{\}$
    $childs = node.childs$
    for child in childs then
      if child in subTrees then childs.remove(child)
    endfor
    $subTrees[node] = childs$
  endif
endfor
for node in nodes and node not in subTrees then
  $chain = computeChain(node)$
  if size(chain) + size(aboveSubTree(chain)) $\leq threshold$ then
    $merge(chain, aboveSubTree(chain)))$
  else
    $subTrees[chain.root] = chain$
  endif
endfor
$subTrees = filter(subTrees)$

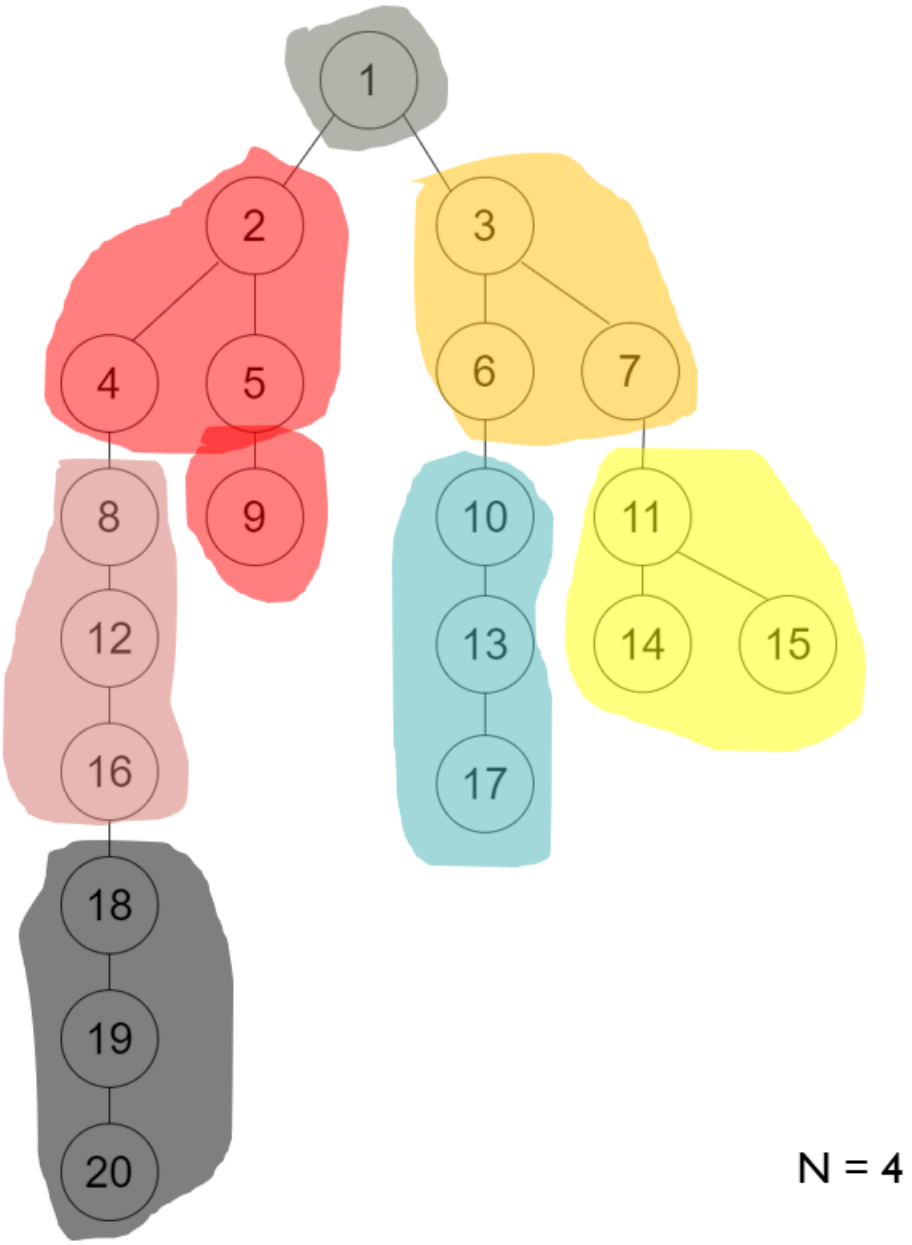

Figure 3.10: Sync2 detection algorithm: end of process. N is the threshold

In order to guarantee a regular hierarchy, a level i synchronizer, it's synchronizer of all inferior layers, from i downto 1. It means that sync4 must be also sync3, sync2 and sync1 at the same time. Then, all sync3s are sync2s and so on. The viceversa is not true. A lower level synchronizer is not required to be an higher one.
When synchronizers are done, routers are filled with the meaningful information. Moreover, to board (or simulator) are also passed information about the composition of sync groups at any level, providing also the number of signal has to wait before emitting.

3.7 Simulator specifications

Simulator, totally designed in Python, aims at emulating the SpiNNaker behaviour, in order to evaluate synchronism algorithm and the validity of routing rule compression.
A rough UML description of this module has provided in Fig 3.11: the simulator requires the SpiNNaker model, composed of *Board*, *Router* and *Core*. All of them are ad-hoc developed objects. Simulator works performing *Task* propagation all around the Board. Execution of a task allows the simulation time to continue. However, each node can switch only one Task per simulation cycle. Since Spinnaker is a high-parallel computation architecture, a lot of Task are on at the same time, also on the same node. So, for each node, a *QueueTask* has implemented, by a First In First Out strategy.

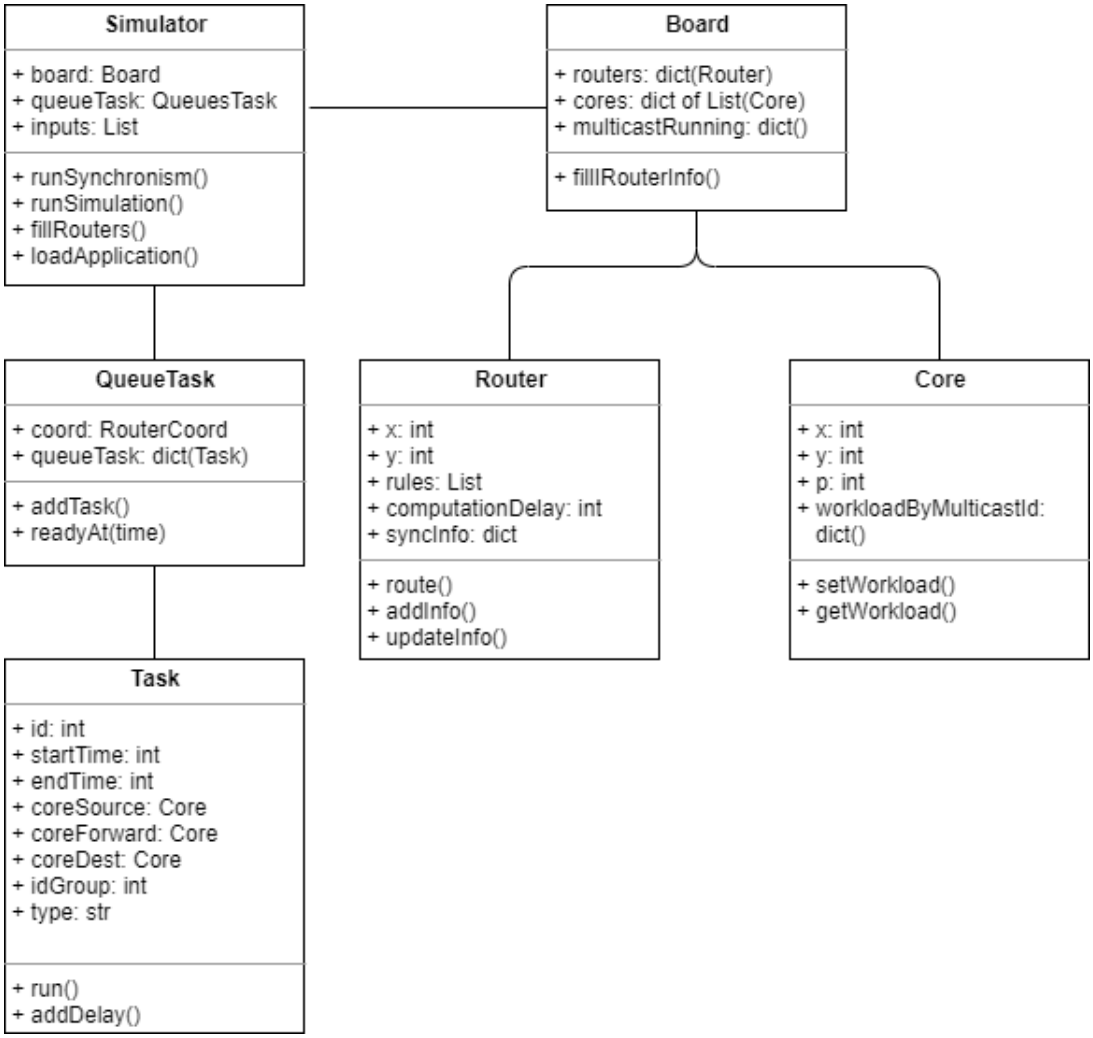

Figure 3.11: Simulator software implementation UML model

A Task has fetched when the simulation time matches its own start time; then the Task has popped out. If a new Task has emitted and its start and end time overlap with some other else in queue, it is simply delayed to the first fully free interval (FIFO approach).
Simulation/Synchronism is over, **succesfully**, when all QueueTask are empty and the

output has written down in the dedicated files.
Simulation/Synchronism fails when one of these two scenarios occur:

- An Exception raises in case, given a router and a incoming routing key, the CAM produces a miss.
- Program never ends due to presence of loops. Actually, minimum spanning tree doesn't form loops, but a too strong routing keys compression may lead to a loss of information, causing them

In order to customize simulation, two things must be provided: *testSet* and *applicationModel*. The *testSet* is the list of multicast groups you want to enable. It's a subset of the whole list of multicast groups provided as *filter.txt*. Formally, the *applicationModel* preloads the QueueTask with pre-built Tasks, in order to emulate some kind of algorithms; moreover, each Task so defined, must come together with the start time and the multicast group membership. For instance, the OneFire application, states that, for each enable multicast, a random core shots a message at $t = 0$. This model has realized by means of Lst. 3.9:

Listing 3.9: simulator.loadApplication method OneFire

```
for id, cores in self.multicastCores.items():
    index = rand.randint(0, len(cores)-1)
    self.inputs.append((0, id, cores[index]))
                        #(startTime, idGroup, Core)
```

Changing the content loaded into *self.inputs*, you change application behaviour.
The PingPong (Lst. 3.10) makes one core to fire, then, after a certain delay, the opposite on network responds.

Listing 3.10: simulator.loadApplication method PingPong

```
for id, cores in self.multicastCores.items():
    self.inputs.append((0, id, cores[0]))
                        #(startTime, idGroup, Core)
    self.inputs.append((delay, id, cores[len(cores)-1]))
                         #(startTime, idGroup, Core)
```

Before looking at how Task *run* method works, the concepts of **workload** and router's **computation delay** have to be explained. The former is a Core's property, the latter Router's. Workload has intended how the amount of time Core requires from starting running its own portion of algorithm to the MCM packet has emitted. It incorporates MPI procedures, the whole middleware software stack and the customized applications themselves. You can expect that the lower the workload, the more atomized the application had been designed. Since synchronism just perform one count and one comparison, all workloads, in that phase, are kept to 1. Router's computation delay is more hardware oriented, since it considers the access time to CAM, its own latency and the matching bitwise procedure. This parameter has set to the minimum possible value, 1, but it may be tuned in future.

Actually, the Task's run method is very easy to understand. Given the multicast id, the source coordinate of packets and type (common message or sync), the routing key has generated, in according to MCM specifications. The return run value is the 32 bit string itself. As said, the routing key has taken by Router performing bitwise operation to match the correct routing rule. The Router's method in charged of that is *routeM*. The sequence of operations are depicted in Fig. 3.12.

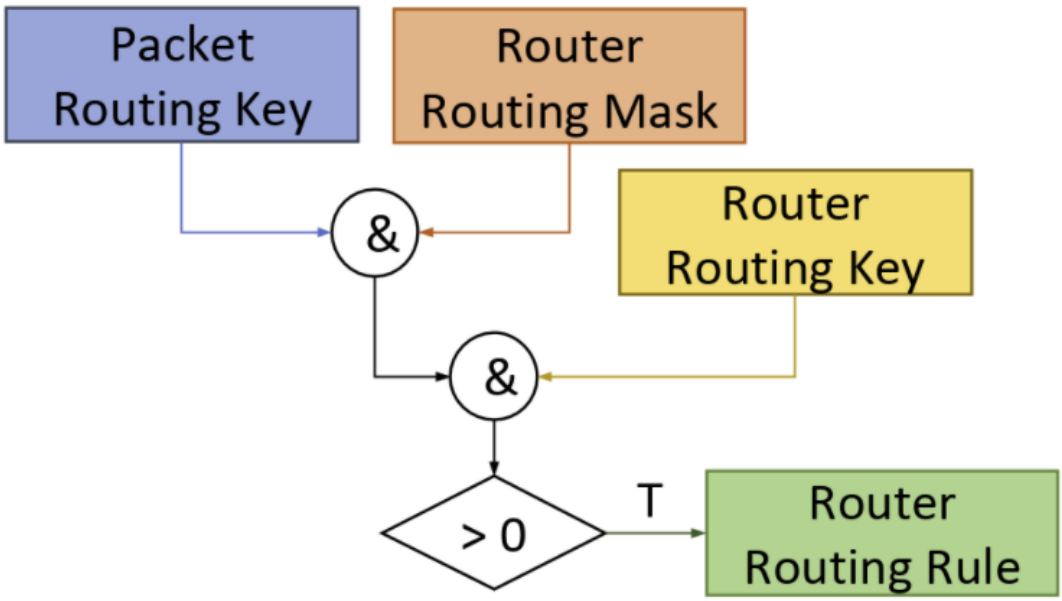

Figure 3.12: Router.routeM(packet_routing_key) flow

From the code point of view Lst. 3.11 describes how the route method works.

Listing 3.11: Router.routeM(key)

```
pk = int(key, 2)

for k, m, r in self.rules:

    tmp = (pk & int(m,2))
    # Apply the mask (m) to the packet key (pk)

    # and then check the routing key (k)


    if (pk & int(m,2)) == int(k,2):

    # Take the first match

        return r

return -1
```

When a Task has run, it simply dies and it's deallocated by simulator. A part invalid case, described above, after a single Task execution two things may happen.

- The packet, came with Task, has to be redirected to other branches, new Tasks are initialized.
- The packet, came with Task, hasn't to be redirected to other branches, just because the 6 forwarding port encoding is all 0s, nothing happens. No generation of further Task occurs.

Simulator returns *histortSim.txt*, *historySync.txt*, *times.txt* as output files. They are used to check correctness of all previous steps or to extract some kind of analytics, as will be presented in the next chapter.

Chapter 4

Results

This chapter aims at providing main results produced by this thesis work, proposed both in tabular and in graphical manner. First, mapping statistics are shown, to evaluate if the presented approach is feasible for Spin5. Then, varying the application model and the amount of loaded multicast groups, synchronism and simulation times get computed. It will come out that the much more paralleling oriented applications fit better this kind of architecture.

To discuss mapping results, two different algorithms have been applied, alternatively, in order to compare an optimum approach versus an heuristic one. The former relies on the *networkx* Minimum Spanning Tree method, the latter has been described deeper in the previous chapter, with the related pseudocode listed. For briefly it will be referred to them as MST and Heuristic.
Moreover, how it had been anticipated in methods chapter, size of synch sub groups may be tuned by a threshold value. MST algorithm uses '5' as that value, chosen from observing experiments on the physical boards. However, mechanism accepting long chains, working as pipeline, the overload issue may be never occur. By increasing threshold to '10', the MST_LongChain results are compared with.
Simulation runs on different scenarios. Five application models are applied. For each, results have been computed taking into account 5, 10, 25, 50 and 100 multicast groups concurrently running on the board. Then, considerations are made varying the Core's workload parameter, 1, 2, 4 and 10. Two out of five applicative are very basics, two other else come from real MPI situations, described in literature, while the latter fits a real SpiNNaker implementation for a massive parallelism and large-scale exchange of small messages. They are

- **OneFireAtBeginning**
- **PingPong**
- **MergeSort** [4], or BubbleSort [27]
- **Fast Encoded DNA (FED)** [28]
- **PageRank** [29]

Then, some heatmaps will be plotted. They come useful in order to see where bottlenecks occur and how the application model is able to distribute tasks among all the

nodes. Heatmaps depicts both synchronism only and then simulation.

Fig. 4.1 showsh the SpiNNaker logic overview, highlithing the 48 chips and their own interconnections. In SpiNNaker literature deals with chips by means of a cartesian representation, providing as identifier a couple of number (x, y). Just for simplicity, while running *networkx* module, I liked assigning incremental integer unique values (Fig. 4.2). Chip per chip mapping has reported in Tab. 4.1.

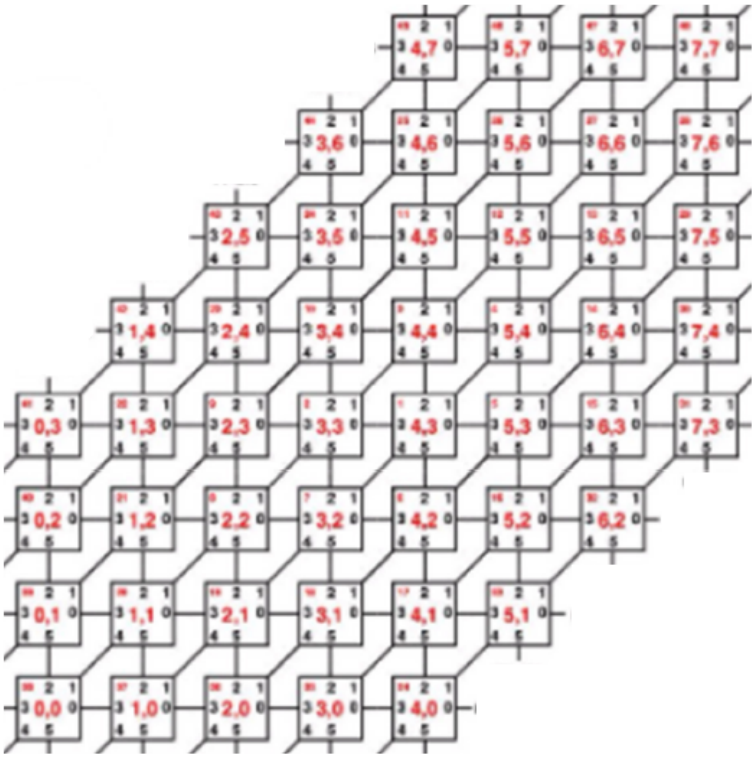

Figure 4.1: Spin5 map on cartesian coordinates

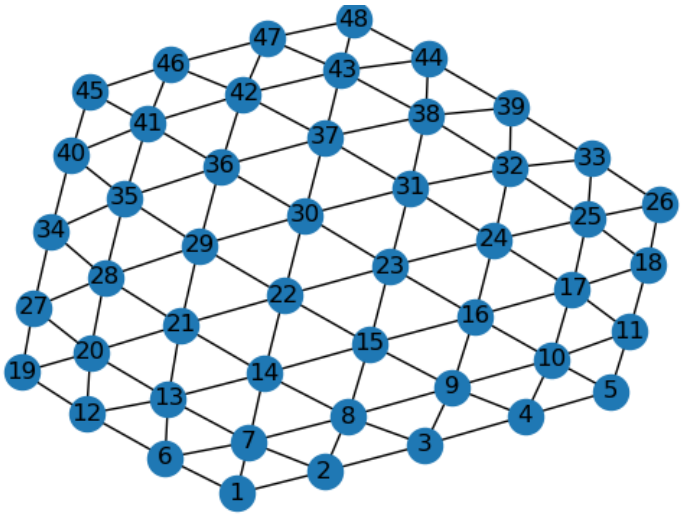

Figure 4.2: Spin5 map on logic id values

Logic ID	Cartesian Coordinates (x,y)
1	(0,0)
2	(1,0)
3	(2,0)
4	(3,0)
5	(4,0)
6	(0,1)
7	(1,1)
8	(2,1)
9	(3,1)
10	(4,1)
11	(5,1)
12	(0,2)
13	(1,2)
14	(2,2)
15	(3,2)
16	(4,2)
17	(5,2)
18	(6,2)
19	(0,3)
20	(1,3)
21	(2,3)
22	(3,3)
23	(4,3)
24	(5,3)
25	(6,3)
26	(7,3)
27	(1,4)
28	(2,4)
29	(3,4)
30	(4,4)
31	(5,4)
32	(6,4)
33	(7,4)
34	(2,5)
35	(3,5)
36	(4,5)
37	(5,5)
38	(6,5)
39	(7,5)
40	(3,6)
41	(4,6)
42	(5,6)
43	(6,6)
44	(7,6)
45	(4,7)
46	(5,7)
47	(6,7)
48	(7,7)

Table 4.1: Dictionary Logic Id - Cartesian coordinates for 48 chips on SpiNNaker

4.1 Router sizes

We are going to analyse the impact of different mapping algorithm on routers contents. The algorithms taken into accounts are: MST, Heuristic and MST_LongChain (MSTLC). Results of this section are obtained by a testset made up of 100 multicast groups, where each of them has composed of a number of cores lasting from 4 to 20, chosen randomly on uniform distribution.
Any router counts 1024 rows, and around 100 are already used for both unicasting and broadcasting. Actually MST and MSTLC produces the same result in mapping, so the only difference stays on how synchronization rules have been organized.

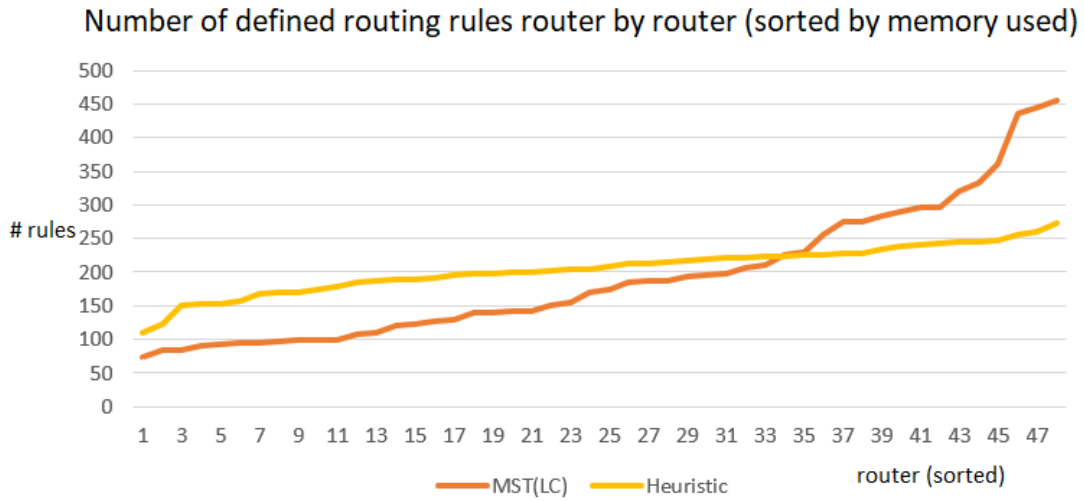

Figure 4.3: Memory size engaged for routing rules, on 100 multicast groups having size in range 4 - 20. Data sorted by number of rules.

Fig. 4.3 reports memory required space, router by router, after mapping algorithm. Note that along the x-axis is reported the logic id domain, not the actual correlation with corresponding number of rules.
Immediately, the big advantage of using MST over Heuristic appears. It's on the global memory saving. Although it increases a lot on few nodes (24.4%), both the global sum and the average rule per chip are lower. A brief summary has shown in Tab. 4.2.

Algorithm	Sum	Average
MST(LC)	9274	193,21
Heuristic	9808	204.33

Table 4.2: Total number of routing rules and average number of rules per router

Fig. 4.4 shows same data but in an unsorted manner, providing a clearer view on memory space consumed on the right chip. Highest consumption is up to chip 15, 22, 29 and 30, which are (3,2), (3,3), (3,4) and (4,4). Looking at the Spin5 map in cartesian coordinates, you can see that these nodes are laid out along the diagonal or very close to it. Geometrically speaking, it acts as symmetry axis, splitting the board into 2 uniform areas. Therefore, it shouldn't have to amaze that diagonal's nodes are the ones more interested in data traffic. Then, the test set has been generated fully randomly. Lowering

the probability of keeping diagonal's cores would drop those high values. Hence, the uniform distribution seems not to be the best choice in resource allocation. In future, allocator algorithm shall be implemented exploiting other probability distributions.

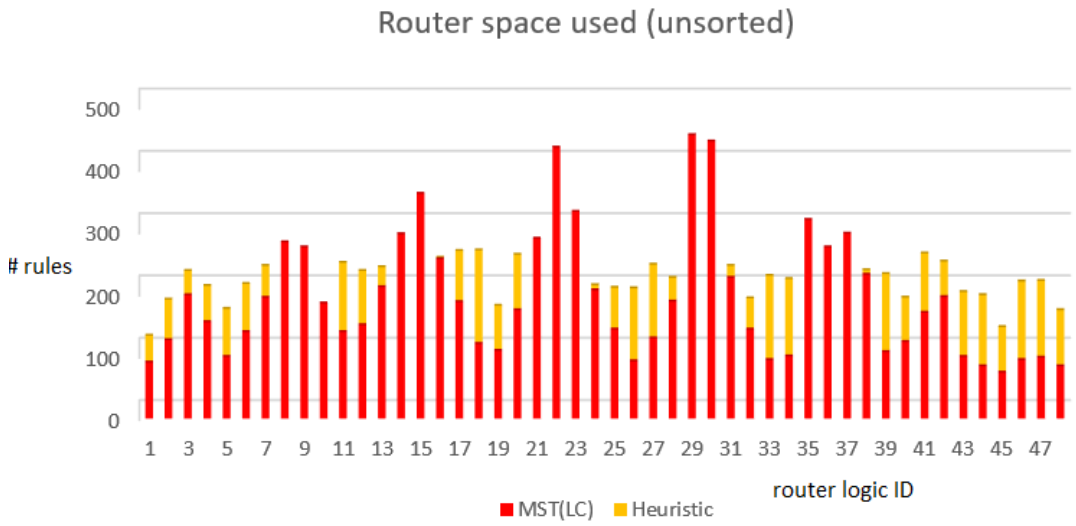

Figure 4.4: Memory size engaged for routing rules, on 100 multicast groups having size in range 4 - 20

Let's move on synchronism mechanism results. Here, MST and MSTLC works differently, producing different data series (Fig. 4.5).

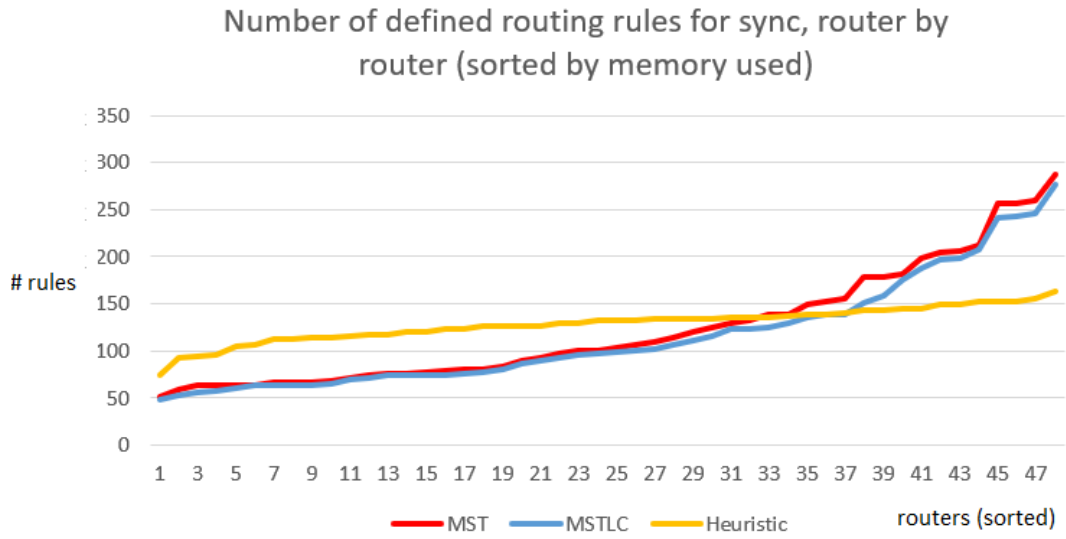

Figure 4.5: Memory size engaged for sync rules, sorted

Similarly as before, MST(LC) globally are cheaper than Heuristic, but getting more expensive on a small subset of nodes; 26% on MST and 15% on MSTLC. Heuristic acts as an horizontal axis. MST performances, with respect to MSTLC, seam to be worst in all points of the plot. Comparisons gathered in Tab 4.3

Algorithm	Sum	Average
MST	5905	123.02
MSTLC	5561	115.85
Heuristic	6181	128.77

Table 4.3: Total number of sync rules and average number of rules per router

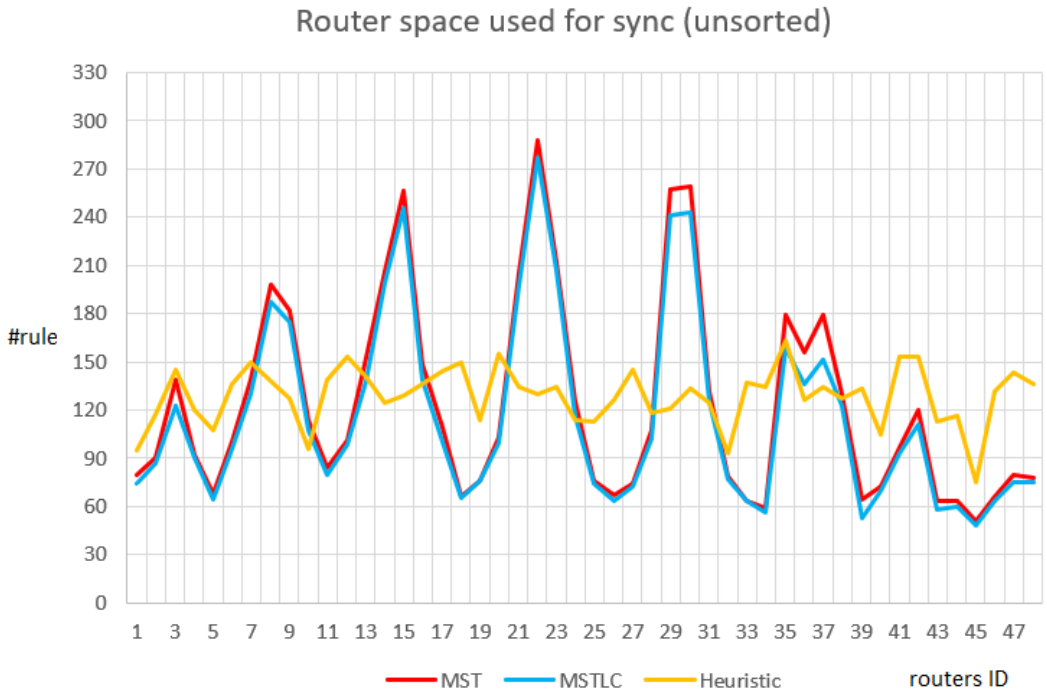

Figure 4.6: Memory size engaged for sync rules, chip ID on x-axis and number of router rows used along y-axis

From memory allocation, Fig. 4.6, peaks are detected over 15, 22, 29 and 30, the same as before. Again, it's suggested a change in allocation paradigm, or different weights setting to the edge in the initialization of mapping algorithms.

As last check, total quantities, i.e. routing + synch rules, are reported in tabular manner (Tab. 4.4).

Algorithm	Total	Global Memory Used [%]	Memory Used in Peak router [%]
MST	15179	30.8	70.6
MSTLC	14835	30.1	69.5
Heuristic	15989	32.5	38.6

Table 4.4: Total number of ALL rules and percentage of memory used

It seams that Heuristic provides better forecasts, but how will be presented in next

section, it is the worst among the three.

4.2 Simulator

4.2.1 Synchronism

Before tuning workload's values and application models, simulation can be conducted to evaluate synchronism mechanism. It's totally independent of the application, and all sync workloads are set by default to 1, just to have a strong metric next, while evaluating simulation results. Now results are presented, in terms of number of cycles required to synchronize everything, just using the mapping method from MST, MSTLC and Heuristic. Then, from them, one has picked up for next models. Just for notation, the test set taking part to simulation names $testSetX$, where X is equal to the number of multicast groups enabled at the same time on board. Results will be produced by running $X = \{5, 10, 25, 50, 100\}$

Let's plot results.

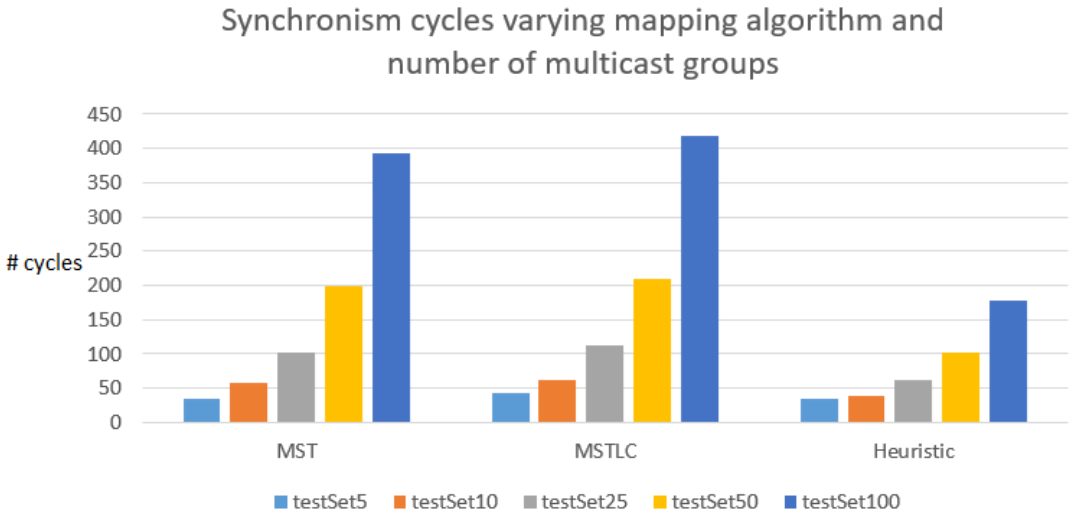

Figure 4.7: Synchronism cycles varying mapping algorithm and number (X) of multicast groups

As you can see in Fig. 4.7, MST and MSTLC are very similar again, but MST shows a little better behaviour. Then Heuristics leads a gain 100% around on the largest *testSet*. Everything redirect us to choice it to move on simulating applications, but it cannot be possible.

Although collected data are impressive, actually, Heuristic is poor of information. All numbers shown so far are after the compression algorithm action. Compression cuts so much, causing a very big issue on the routing rules especially. These are so compressed that a kind of **aliasing** occurs! This phenomena causes a lot of error and wrong configurations. If MST and MSTLC returns an error $\sim 3\%$, Heuristic leads simulator to raise exception at least in the 50% of cases.

MST needs a bit more memory to save a bit of synchronism cycles (simulation times are exactly identical because mapping is the same). MSTLC behaves in the opposite manner, less memory leads to bit of more sync time. Next simulation are conducted up to MST.

The reason stands on the fact that, the higher the threshold for sub tree splitting, the higher the probability more signals may come on the same node master. Issues about that had been experimentally observed. So, just to avoid this kind of trouble, MSTLC has discarded, as well as Heuristic.

4.2.2 Simulation

Simulator requires three objects to successfully run a simulation

- The **testSetX**, where $X = \{5, 10, 25, 50, 100\}$, i.e. the number of multicast group concurrently running on SpiNNaker
- The **workloadW** where $W = \{1, 2, 4, 10\}$. They are set heterogeneously widespread all over cores
- The **applicationModel**: OneFire, PingPong, MergeSort, FED and PageRank

All combination of them are performed. The mapping algorithm has fixed: MST is the best candidate.

OneFire: randomly, a core per multicast group, starts a communication at time 0.

PingPong: two opposite cores, owning to each multicast group, are detected. The former starts a communication at time 0; after a while the latter responds by triggering another multicast packet.

MergeSort: any multicast group performs a merge sort algorithm on a different subset of data. You know that merge sort is a recursive algorithm divide et conquer based (Fig. 4.8). Each operation of splitting and sorting is up to a different core. When a process ends, regardless of hierarchical level, the core generates a multicast packet.

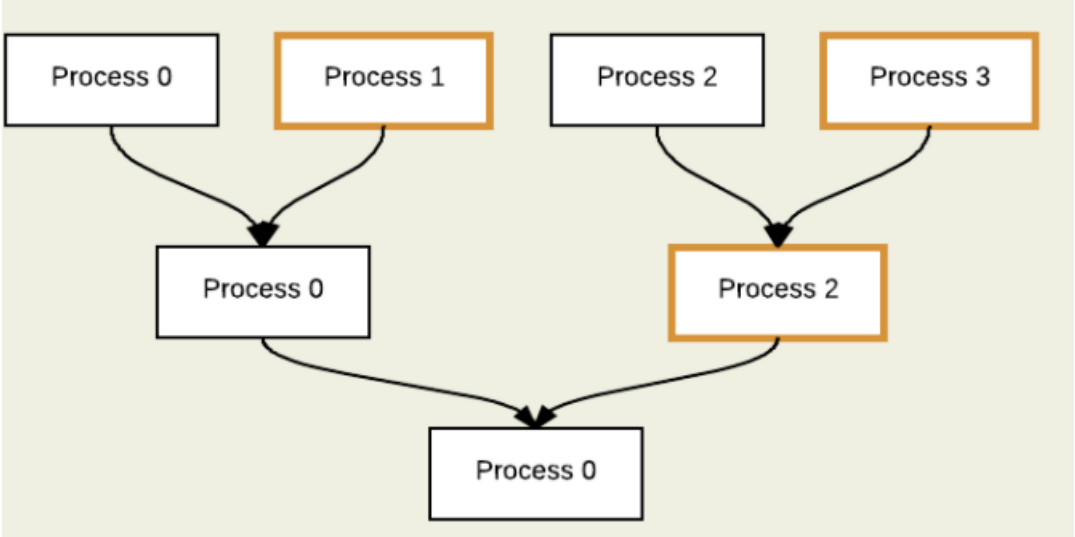

Figure 4.8: Merge Sort application model flow chart [4]

FED: this algorithm aims at performing search of subsequences on long DNA strings. Two phases are done: *preprocessing*, where distribution of information happens, *matching*, which is the core of the algorithm itself and a lot of shifting and comparisons are performed. Pattern matching fits a lot parallel application [30]. Preprocessing has not been modelled here, just because exploits unicast communication to spread data and fill cores. The interesting step is matching. When each core owns its own chunk of text, they run, internally, FED primitives and emit results by broadcast packets[28]. In the presented application model, broadcast concept is replaced by multicasting, imaging that different FED algorithms run at the same time, as many are the enabled multicast groups.

PageRank: PageRank is a very popular algorithm used to provide a quantitative classification on a set of objects (web pages at the beginning, by Sergey Brin and Larry Page, Google's founders), depending on number of links the single element builds with all the others. Just have a look at the PageRank equation [31]:

$$PR(A) = \frac{1-d}{N} + d \cdot \Big(\sum_{k=1}^{n} \frac{PR(P_k)}{C(P_k)} \Big) \tag{4.1}$$

where:

- $PR(A)$ is the PageRank value of page A
- N is the total number of pages
- n is the number of pages such that exists a link between page P_k and A
- $C(P_k)$ is the total number of links P_k owns
- d has known as dumping factor, just for tuning purposes. Its usual value is 0.85

From the equation you can see that, each page needs of knowing the PageRank value of all the others in the set. So each node, or vertex, sends a multicast communication with all its own main information. When all nodes collect information of each vertex, they are ready to apply the formula above. Actually, it's not enough. At this point the results are pretty raw and some further reiteration of the described steps are required, until some kind of convergence criteria has reached.
Multicast approach fits better than the broadcast one the PageRank algorithm. Not all pages in the universe must have a link in between. For all the page sets, the PageRank computation runs in parallel. Moreover, it's very common the case where a page is part of two, or more, different sets, and multiple PageRank algorithms can interest it in a totally independent manner. For simulation, as convergence criteria has been chosen the execution of 5 loops. For μ factor calculation, it gets more interesting deals with a single iteration rather than considering the algorithm running in a monolithic way. This choice has justified by literature on PageRank over SpiNNaker [29], describing how barriers have been put in the middle between two consecutive iterations.

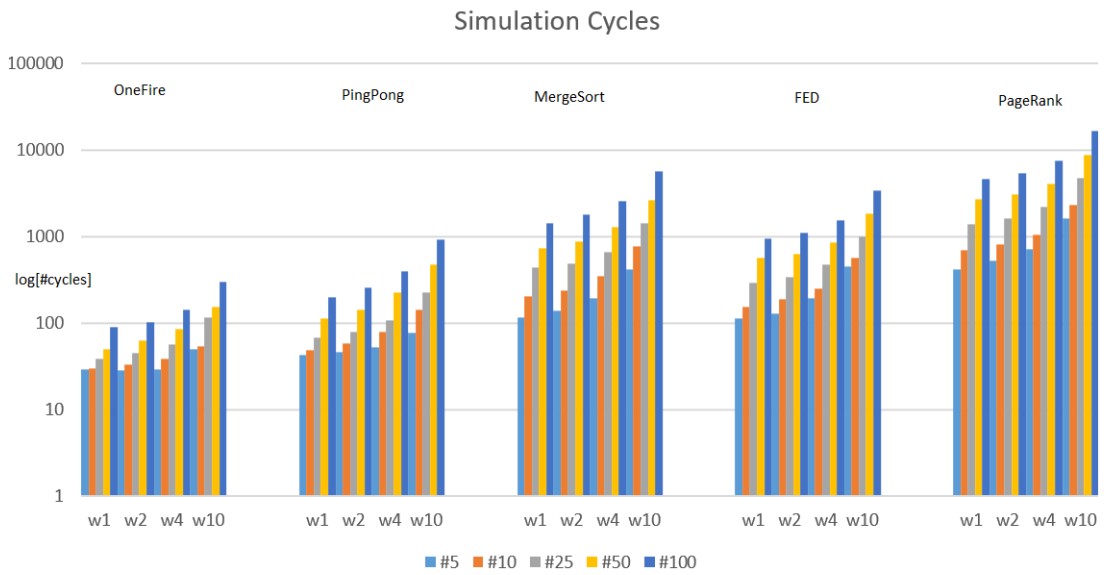

Figure 4.9: Simulation cycles varying the application model, varying the testSet and the workload

Looking at Fig. 4.9, we expected that the higher the workload, the higher the number of simulation cycles. That's obviously true, but it's interesting to notice one thing. Given any testSet, for any experiments, just pick up simulation cycles at $w1$ and $w10$. From 1 to 10 one order of magnitude stands. But it doesn't occur between the two relative numbers of simulation cycles, always kept under the order of magnitude.

Another relevant thing to be observed is how grow the simulation time as a function of the number of multicast groups. Now, keep the workload and application fixed. The way number of simulation cycles grows is sublinear with respect to that. It's amazing; this fact underlines the goodness of simulator, rewarding concurrent running application, rather than a stand alone scenario.

Next tables 4.5, 4.6, 4.7, 4.8, 4.9, report how many simulation cycle are needed to accomplish the application model, varying the set workload and the test set size.

Then, Tab. 4.10 shows the variation on the amount of multicast packets generated, varying the application model. If n is the number of cores making up a single multicast group, the total number of tasks emitted by each of them can be known in advanced. For PageRank, k is the number of iterations required to reach the convergence.

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	29	30	38	50	90
$w2$	28	33	45	63	101
$w4$	29	38	56	84	140
$w10$	49	53	114	151	298

Table 4.5: OneFire application model: simulation cycles

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	42	48	67	111	196
$w2$	46	58	79	140	254
$w4$	52	79	108	225	389
$w10$	77	141	227	468	921

Table 4.6: PingPong application model: simulation cycles

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	116	204	441	727	1428
$w2$	137	239	487	877	1800
$w4$	192	343	654	1266	2526
$w10$	419	773	1405	2590	5608

Table 4.7: MergeSort application model: simulation cycles

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	113	152	293	559	944
$w2$	128	186	336	628	1088
$w4$	194	249	473	856	1514
$w10$	448	562	989	1830	3377

Table 4.8: FED application model: simulation cycles

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	417	690	1393	2716	4633
$w2$	518	797	1604	3011	5294
$w4$	711	1037	2174	4096	7547
$w10$	1616	2315	4734	8648	16351

Table 4.9: PageRank application model: simulation cycles

Application Model	Complexity as Number of Tasks
OneFire	1
PingPong	2
MergeSort	$n \log n$
FED	n
PageRank	$n \cdot k$

Table 4.10: Multicast packets generated per application model

Computational cost of algorithms is something cannot be neglected, as you see from tables and plot. It affects the simulation time. Then, the higher amount of generated tasks, the longer are the queues in any chip, leading to possible bottlenecks. From a graphical point of view, it's easy to see where tasks pile up: plotting heatmaps. The reported heatmaps are just photography of the state in each chip queue, at any simulation

(or synchronization cycle). The brightness of color in each cell, determines number of tasks ready to be executed.

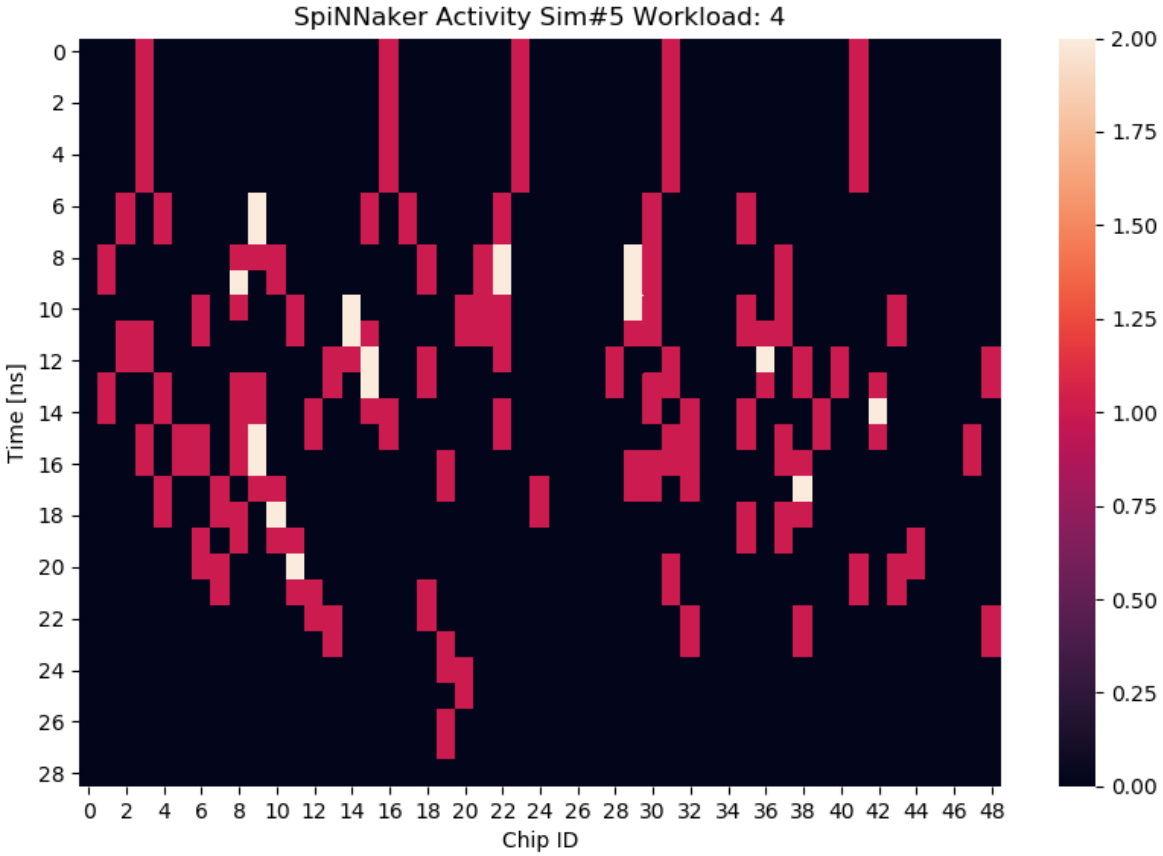

Figure 4.10: Heatmap of OneFire experiment. Workload set to 4, testSet5. On the y axis the simulation cycles, here treated as nanoseconds. On the x axis the chip id.

Fig. 4.10 shows how at the beginning 5 chips turn red. Being the workload set to 4, they requires at least 4 cycles to be committed. Then, at fifth cycle, first ramifications and overlaps appear.
Tuning the workload, the heatmap pattern appears totally different Fig. 4.11. Considering the same application OneFire, testSet5, but $w10$, it's like performing a zoom out on the board activity. In fact, increasing workload, and testSet size, many information get filtered and only dangerous bottleneck are left. These plots can be used as graphical tool to evaluate the "hottest" clusters, depending on multicast groups generated in the allocation phase. Looking at bright areas you may decide to remove cores on those chips, preferring unused hardware. Other cases are shown af Fig. 4.12, 4.13 and 4.14.

To generate heatmaps, *seaborn* Python package has been used. It takes as input a *numpy* matrix, which is one of the output, as textual file, by *main.py*. The same path can be conducted for synchronism mechanism matrix, as well.

Then, plotting synchronism's heatmap, you will see more regular structures, where the majority of activities has focused on synchronizer nodes, like in Figs. 4.15 and 4.16.

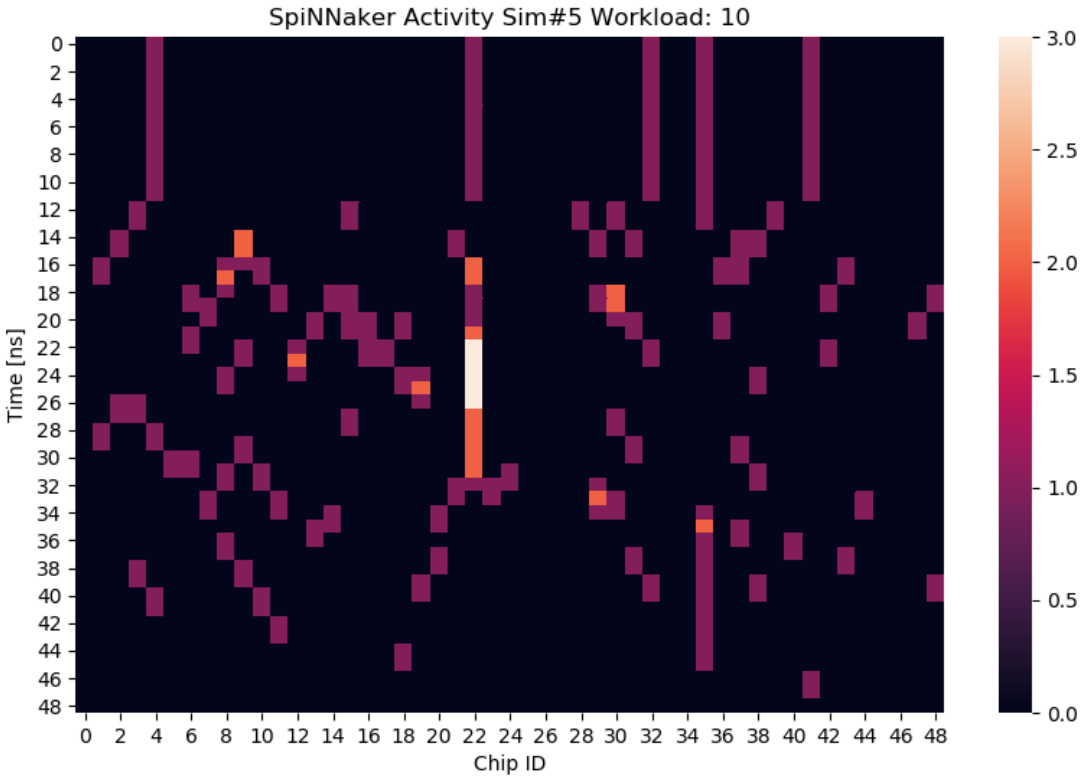

Figure 4.11: Heatmap of OneFire experiment. Workload set to 10, testSet5. On the y axis the simulation cycles, here treated as nanoseconds. On the x axis the chip id

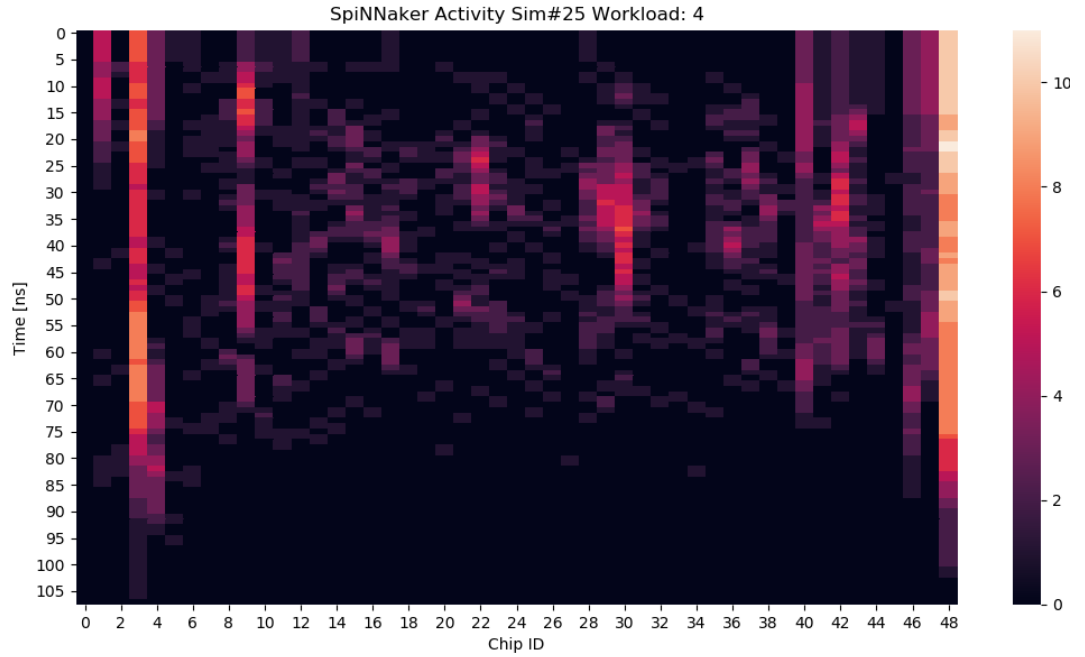

Figure 4.12: Heatmap of PingPong experiment. Workload set to 4, testSet25. On the y axis the simulation cycles, here treated as nanoseconds. On the x axis the chip id

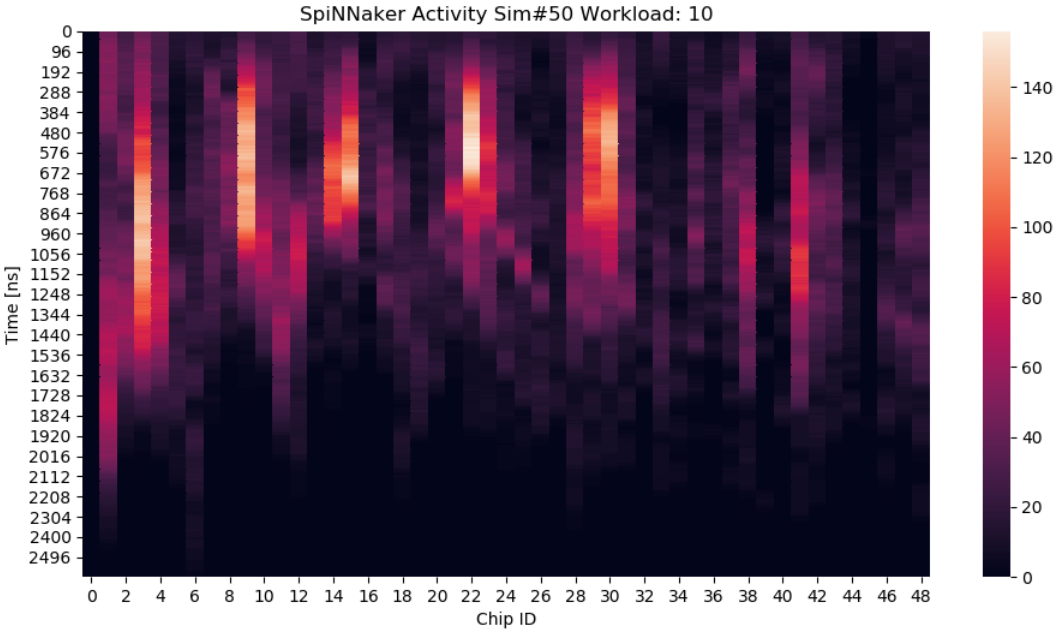

Figure 4.13: Heatmap of MergeSort experiment. Workload set to 10, testSet50. On the y axis the simulation cycles, here treated as nanoseconds. On the x axis the chip id

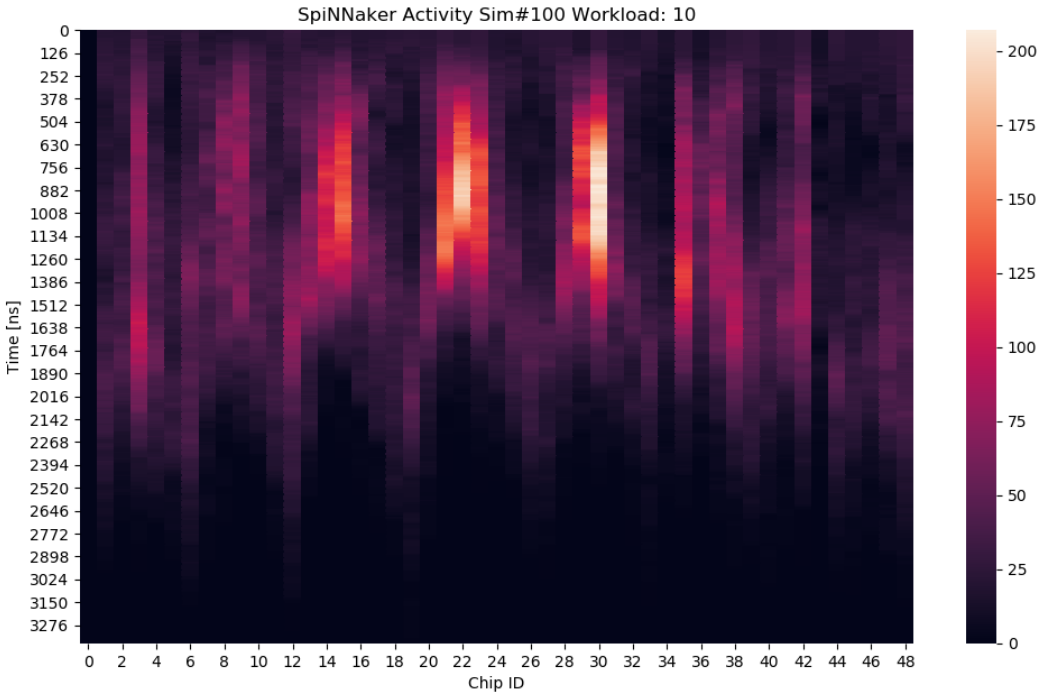

Figure 4.14: Heatmap of FED experiment. Workload set to 10, testSet100. On the y axis the simulation cycles, here treated as nanoseconds. On the x axis the chip id

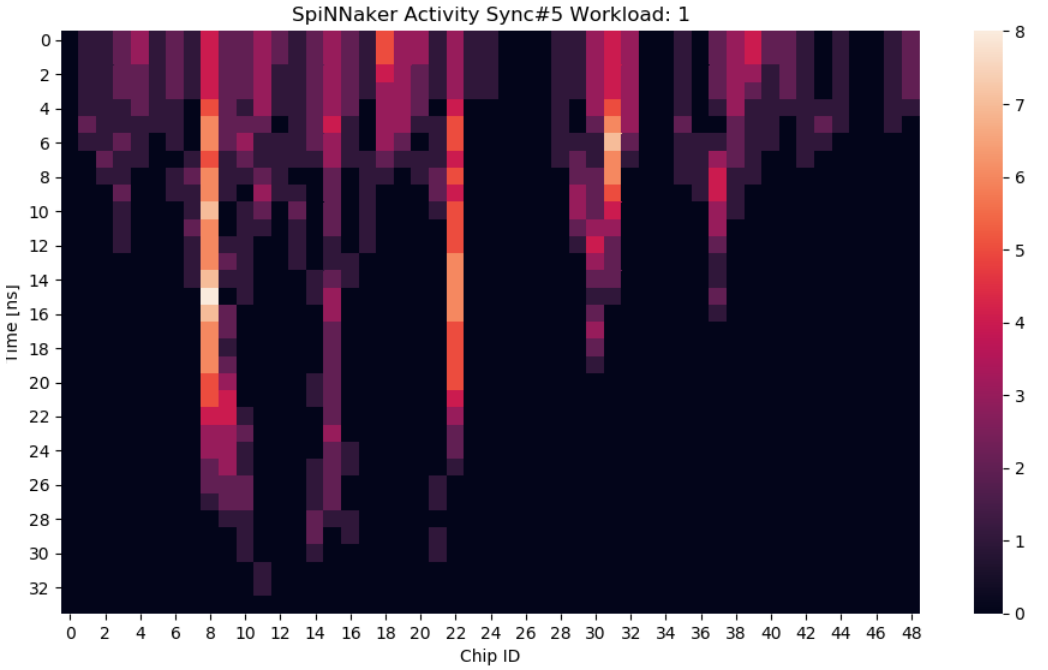

Figure 4.15: Heatmap of synchronism mechanism. Workload set to 1, testSet5. On the y axis the synchronization cycles, here treated as nanoseconds. On the x axis the chip id

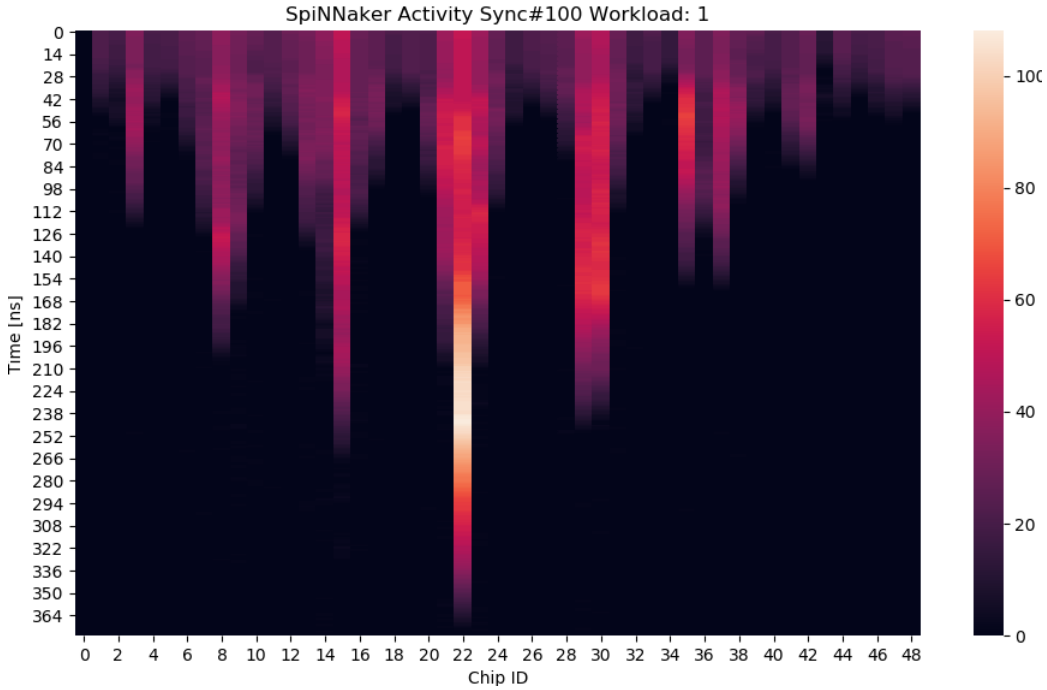

Figure 4.16: Heatmap of synchronism mechanism. Workload set to 1, testSet10. On the y axis the synchronization cycles, here treated as nanoseconds. On the x axis the chip id

The images above show the behaviour of SpiNNaker chips in time. The main target of these heatmaps is to provide a kind of focus on possible hard-to-manage situation on

the real hardware. The designer can use this tool as a debugger one. Figs 4.13 and 4.14 describe how the highest activities are in the first half of simulation process. Looking at this behaviour, the designer may decide to delay the execution associated to a subset of all allocated multicast groups. Moreover, these images depict possible bottlenecks on the column related to chip having the higher brightness, possibly suggesting avoiding those nodes to fit in multicast groups.
Same considerations can be carried out from Figs 4.15 and 4.16. Synchronization mechanism detects a hierarchical organization among nodes to be synchronized and synchronizers. This hierarchical pyramid reflects on how sync packets travel all around the board. From figures it's possible to see some "main" branches (in terms of length propagation in simulation time) come together with some minor ones.

4.2.3 The μ factor

This parameter measures the simulation activity, compared with synchronization time.

$$\mu = \frac{T_{sim}}{T_{sync}} \tag{4.2}$$

High values of μ say that the application model is strong parallelization oriented, hence fits better SpiNNaker behaviours. On the contrary, when $\mu < 1$, single thread models, keeps more time in synchronizing rather than running the application itself; that's the case at OneFire and PingPong models. It's shown just performing the ratio, experiment by experiment. Before plotting graphs, be careful about one more thing. Too high μ values generate some unwanted behaviour. As said in previous chapter, SpiNNaker is a GALS architecture. It means that, on the hardware, there exists some regions, the single chip, acting as totally independent clock domains. Due to that, cores owning to two different domains are possibly fed by different clock frequencies. It leads to need having asynchronous communication channels among these domains. However, the high parallel propagation of execution, may lead to some unbalancing among threads, eventually causing hazards or exception in software flow. If simulation takes too time, in according to MPI protocol, a sync barrier has to be performed, in order to allow balancing in computation propagating. It's not a golden rule, but from experiments, a satisfying μ range

$$1 \leq \mu \leq 10$$

In next pages are reported, application by application, all the μ factors extracted by varying the workload value and the size of test set applied. Let's depict some comparison histogram graph.

In almost all combination cases, the OneFire (Fig. **??**) takes more synchronization times than simulation ones. Actually, for this kind of model μ is consistent. Then, the lower the testSetSize, synchronism time grows faster than simulation. This is bad, but coherent with the stand alone model.

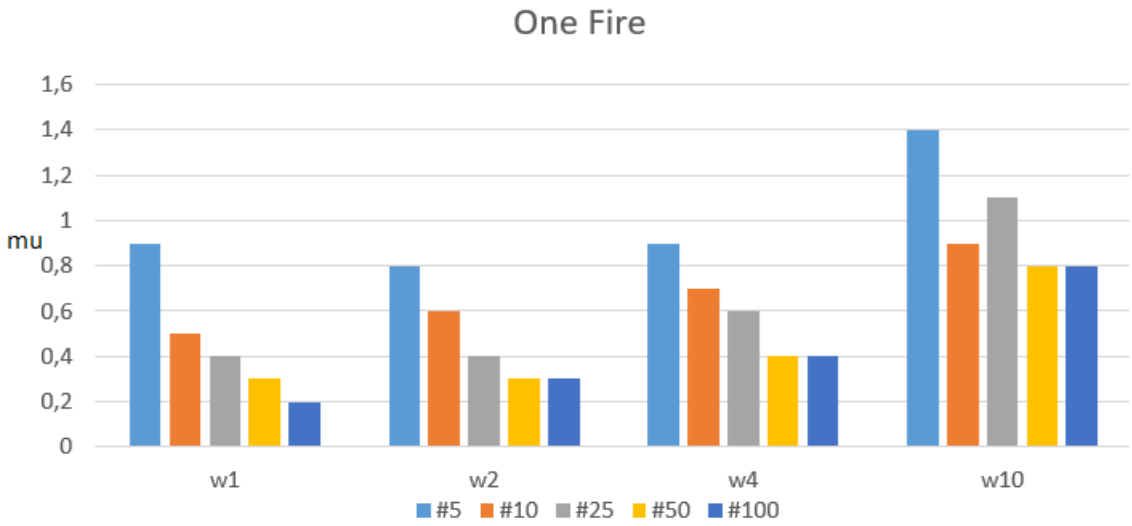

Figure 4.17: OneFire μ, varying workload and testSet size

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	0.9	0.5	0.4	0.3	0.2
$w2$	0.8	0.6	0.4	0.3	0.3
$w4$	0.9	0.7	0.6	0.4	0.4
$w10$	1.4	0.9	1.1	0.8	0.8

Table 4.11: OneFire μ, varying workload and testSet size

A few better values come out from PingPong (Fig. 4.18 and Tab 4.12), but are still under 1 in 10% of cases. However, some bad performances are again justified by single thread flow.

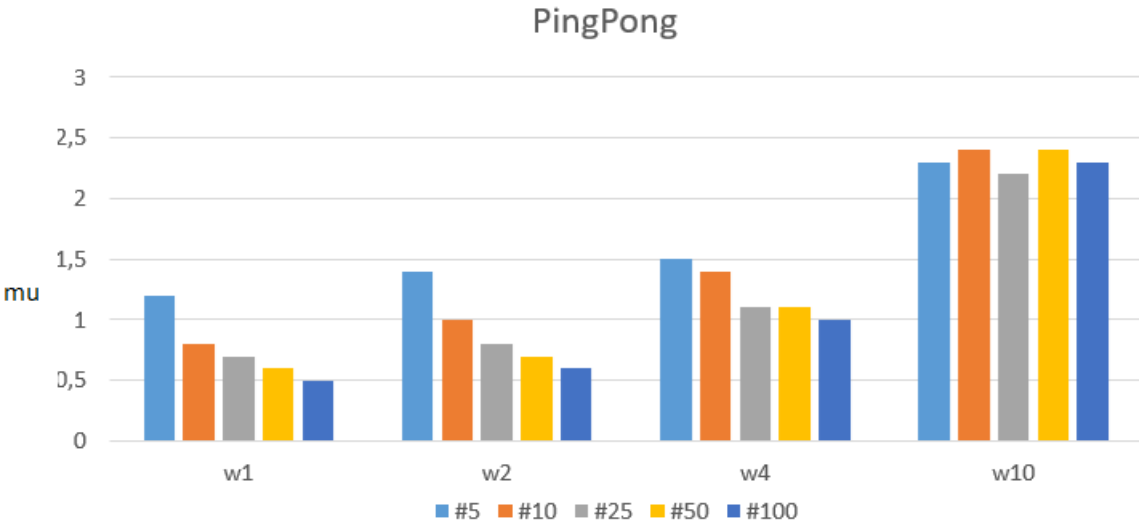

Figure 4.18: PingPong μ, varying workload and testSet size

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	1.2	0.8	0.7	0.6	0.5
$w2$	1.4	1.0	0.8	0.7	0.6
$w4$	1.5	1.4	1.1	1.1	1.1
$w10$	2.3	2.4	2.2	2.4	2.3

Table 4.12: PingPong application model: μ factors

When parallelized activity grows (in terms of simulation), μ factor grows and fits the satisfying range. Another difference stands on how μ behaves, fixed the workload, changing the testSet size. At OneFire and PingPong, μ tends to decrease while number of multicast groups increases. MergeSort, $O(n \log n)$, Fig. 4.19 and Tab. 4.13, relative μ growth is slower than size variation. It can be explained by the overlinear complexity of the algorithm. What's really interesting is the plot on FED, $O(n)$, Fig. 4.20 and Tab. 4.14. Fixing workload, μ stays on a kind of plateu, such that being constant when number of task grows.
It seems that linear algorithms produce more stable results, because both synchronization and simulation grow at the same rate. If μ gets trendly, like linearithmic $O(n \log n)$, when testSet grows too much, μ factor can generate to very high values.

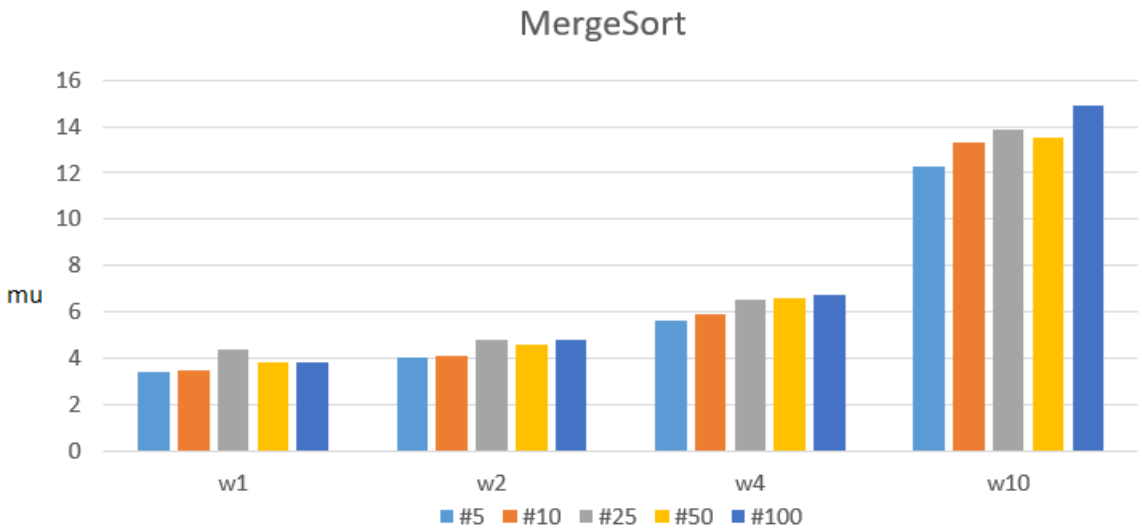

Figure 4.19: MergeSort μ, varying workload and testSet

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	3.4	3.5	4.4	3.8	3.8
$w2$	4.0	4.1	4.8	4.6	4.8
$w4$	5.6	5.9	6.5	6.6	6.7
$w10$	12.3	13.3	13.9	13.5	14.9

Table 4.13: MergeSort application model: μ factors

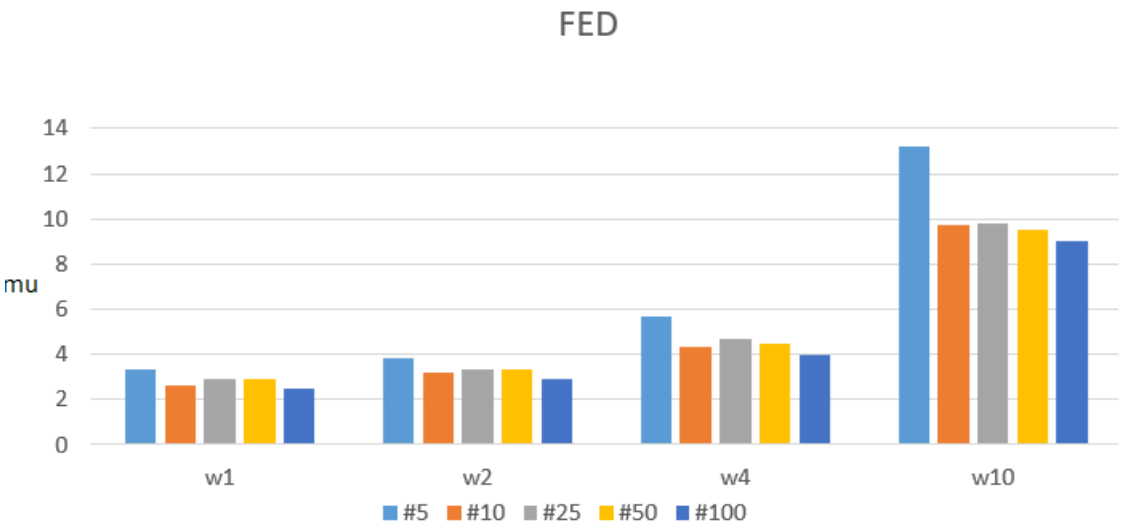

Figure 4.20: FED μ, varying workload and testSet

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	3.3	2.6	2.9	2.9	2.5
$w2$	3.8	3.2	3.3	3.3	2.9
$w4$	5.7	4.3	4.7	4.5	4.0
$w10$	13.2	9.7	9.8	9.5	9.0

Table 4.14: FED application model: μ factors

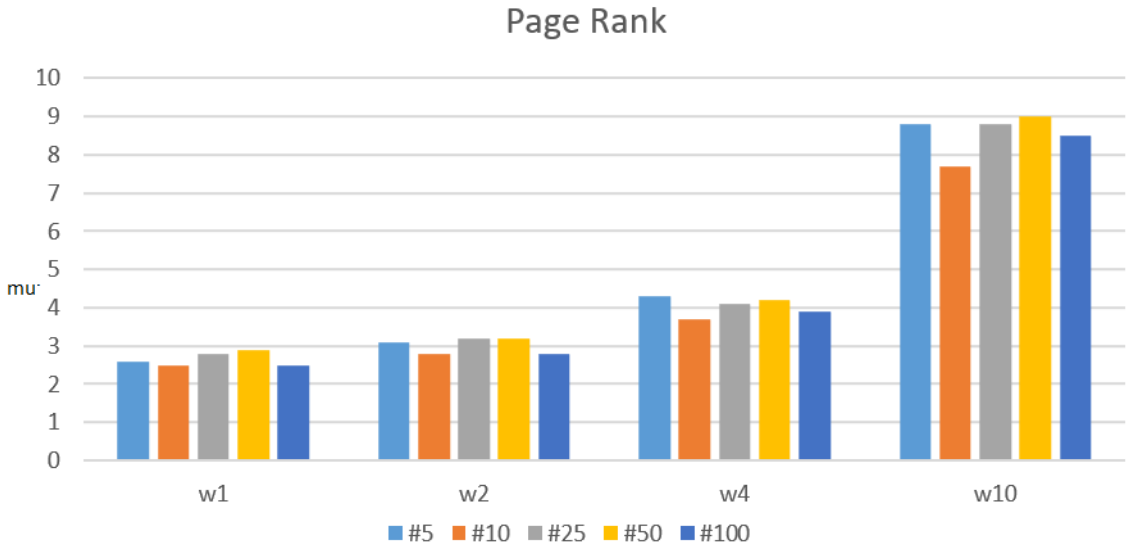

Figure 4.21: PageRank μ, varying workload and testSet

Workload	TS5	TS10	TS25	TS50	TS100
$w1$	2.6	2,5	2,8	2,9	2,5
$w2$	3,1	2,8	3,2	3,2	2,8
$w4$	4,3	3,7	4,1	4,2	3,9
$w10$	8,8	7,7	8,8	9,0	8,5

Table 4.15: PageRank application model: μ factors

However, by simulation, we see that PageRank results, Fig. 4.21 and Tab. 4.15, having complexity $O(n \cdot k)$ very close to linearithmic, improve μ factor, since introducing in the middle the synchronization cycles. **It means that, regardless of any discussion about computational complexity, exploiting the MPI barrier concept applied to SpiNNaker, allows developers to obtain the same result as well as FED model, in a context scenario much more massively oriented to multithreading and parallelism.** Moreover, all PageRank's mu values are in range $[1 - 10]$, considered as reliable.

Chapter 5

Conclusions

This thesis work aims at contributing the expansion of functionalities, released by the MPI software stack for Spinnaker, designed by Bioinformatics group at Politecnico di Torino.
At the end of this project, some modules and tools have been committed and made available for research field on SpiNNaker. First of all, the main contribution is all over on the extension of SpinMPI library, having as target the realization of MPI protocol running on SpiNNaker architecture. SpinMPI already offers facility to manage unicast and broadcast communication; this thesis work wants to enlarge it by providing methods and algorithms for multicasting. Before passing through a numerical analysis, trying to estimate the feasibility, then moving to the realization of a SW able to manage and gather multicast informations. That has been performed taking into account, as hard constraint, the poor memory space for each router on the board; trying to succefully fit onto the highest amount of groups.
Moreover, the SpinMPI's barrier algorithm has been produced, again taking into account routing memory space as well as avoiding useless loops of packet propagation. That's the reason why the synchronization mechanism, here presented, follows a hierarchical organization pyramidal-like.
To accompany these algorithms, a simulator tool has been designed, in order to prove their validity. Offering some degree of freedom, such as size of numbed of multicast groups concurrently running on the board, the workload for each SpiNNaker's core and the loaded algorithm, I have had the opportunity to extract some analytic for further analysis and comparisons among chosen parameters, taken as input by simulator.

The obtained results come from a testSet composed of 100 multicast groups, fully randomly built. This size tries emulating a very probably real scenario, although Spinnaker comes with some resource limitation in terms of memory space, especially on routers. The mapping algorithm is able to fit any chip, reaching 70% of total memory used on the worst case. Although, the full memory space is covered in the 30%. These considerations include synchronism rules as well. Anyway, it's clear that something can be done to improve memory sizes distribution. Therefore, from this analysis, a totally unexpected point raised up. Allocator algorithm was something we hadn't ever thought. But, a clever one (now, it's just a call to the uniform random function), for sure would turn low peaks on most traded node, at cost of a small increase on the average percentage. Probably, a Gaussian curve, where chips are laid out depending on their own geometrical features, may return an higher number of allocations from boundary-nodes, and lower ones on

nodes along the Spinnaker's diagonal.

Synchronism mechanism, mainly analysed in terms of simulator's cycles, has been proven having a slightly sub-linear behaviour as function of multicast groups engaged. It means that, eventually, the board might be connected to others Spin5, maybe running multicast *inter* them (not *intra*), so complexity could stay under control. Again, the idea of a smarter allocator might avoid too long groups, leading to a very big latency.

Talking about latency, here motivation of minimum spanning tree methods comes. Among almost all cases, no multicast groups have been drawn having a diameter longer than a dozen hops. Heuristic didn't be careful about chain sizes, producing in very bad cases, trees 30 hops sized.

Then, kind of algorithms family has been treated. Simple single-thread algorithms have been modelled, compared with multi processes ones coming from MPI literature. It shouldn't be surprising that Spinnaker, not being a general purpose machine, runs better some kind of application than others. Algorithms that don't require to be split, lose in performances due to useless spread of data among cores. The opposite side covers MPI-like application. MPI is, *de-facto*, the most common standard for asynchronous communication protocols on clusters of nodes; its literature is very vast, justifying also the choice in the usage for Spinnaker. Following MPI criteria, simulator offers chance on building application models, to get quantitative results before writing firmware, just by running an emulator. Depending on the results, designer may decide to tune some parameters, such as increasing the size of a group, or reduce it, update cores assigned to a job in order to avoid overloads situation, throw some deductions on number of application concurrently running, and so on.

Proposed results show how the MPI approach fits well with SpiNNaker architecture. Looking at FED, performances are independent on the testSet size have been obtained. Then, comparing MergeSort and PageRank, the former deals with monolithic propagation, the latter treats barrier cycle by cycle. PageRank, leads to an amount of multicast propagation higher than MergeSort, but returns better performances in terms of simulation cycles; showing relationship behaviours very similar to Fast-Encoded DNA. Fitting barriers has not to be scared, they are part of MPI and are useful to avoid non-homogeneous propagation among threads. Moreover, from these results, you can see that performances increases in according to this strategy, rather then avoiding them.

Spinnaker idea as neuromorphic machine was born in 2005; then the first prototype has been released in 2009. Ten years are passed, making it a still very young technology. However, a lot of publications have been released in the meantime, making it very popular and scalable in a lot of Computer Science problems. Researchers merit stands on the fact having understood, in advance, that a so popular protocol, like MPI, was missing on the Spinnaker architecture. Working on that, unicast and broadcast communicators have been published. Nowadays, the big goal deals with on multicast implementation. It leads to an higher level of complexity due to the vastness of possible "particular cases" to be managed, without violating the most general criteria. Reminds that number of combinations exceeds, a lot, the number of elementary particles in the known universe.

Bibliography

[1] Carver Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 1990.

[2] Andrew G. D. Rowley, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Andrew Gait, David R. Lester, Luis A. Plana, Oliver Rhodes, Alan B. Stokes, and Steve B. Furber. Spinntools: The execution engine for the spinnaker platform. *Frontiers in Neuroscience*, 13:231, 2019.

[3] Luca Peres. Methods to exploit core-parallelism to improve configuration and results gathering phases of snn simulations in a many-core neuromorphic platform. Master's thesis, Politecnico di Torino Corso di laurea magistrale in Ingegneria Informatica (Computer Engineering), 2018.

[4] Libby Shoop. Message passing with mpi - merge sort, 2018.

[5] *Storia delle discipline mediche.* 2001.

[6] Nomination Archive. Nobelprize.org. nobel media ab 2020, 26 Jun 2020.

[7] Walter McCulloch, Warren S.; Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 1943.

[8] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[9] IntelLabs. The emergent capabilities in artificial intelligence being driven by intel labs have more in common with human cognition than with conventional computer logic, 2018.

[10] Javier Navaridas, Mikel Luján, Luis A. Plana, Steve Temple, and Steve B. Furber. Spinnaker: Enhanced multicast routing. *Parallel Computing*, 45:49 – 66, 2015. Computing Frontiers 2014: Best Papers.

[11] Human Brain Project authors. Overview, 2017.

[12] Thomas Sharp; Francesco Galluppi; Alexander Rast; Steve Furber. Power-efficient simulation of detailed cortical microcircuits on spinnaker. *Journal of Neuroscience Methods*, 2012.

[13] Luis A. Plana, David Clark, Simon Davidson, Steve Furber, Jim Garside, Eustace Painkras, Jeffrey Pepper, Steve Temple, and John Bainbridge. Spinnaker: Design and implementation of a gals multicore system-on-chip. *J. Emerg. Technol. Comput. Syst.*, 7(4), December 2011.

[14] B. Sen Bhattacharya and S. B. Furber. Biologically inspired means for rank-order encoding images: A quantitative analysis. *IEEE Transactions on Neural Networks*, 21(7):1087–1099, 2010.

[15] A. Marcelli, E. Sanchez, G. Squillerò, M. U. Jamal, A. Imtiaz, S. Machetti, F. Mangani, P. Monti, D. Pola, A. Salvato, and M. Simili. Defeating hardware trojan in microprocessor cores through software obfuscation. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, 2018.

[16] Elisa Ficarra; Santa Di Cataldo. Bioinformatics course transparencies, 2019.

[17] P. Y. Simard, D. Steinkrau, and I. Buck. Using gpus for machine learning algorithms. In *Proceedings. Eighth International Conference on Document Analysis and Recognition*, pages 1115–1119, Los Alamitos, CA, USA, sep 2005. IEEE Computer Society.

[18] Manolo De Agostini. Il supercomputer neuromorfico con 1 milione di core che mima il cervello umano, 2018.

[19] Martin Schulz. Mpi: A message-passing interface standard version 3.1, 2015.

[20] Francesco Barchi; Gianvito Urgese; Enrico Maci; Andrea Acquaviva. An efficient mpi implementation for multi-core neuromorphic platforms. *New Generation of CAS (NGCAS)*, 2017.

[21] University of Manchester. Spynnaker, 2015.

[22] University of Manchester. Spinnmachine, 2015.

[23] University of Manchester. Spinnman, 2015.

[24] University of Manchester. Pacman, 2015.

[25] University of Manchester. Spinnfrontendcommon, 2015.

[26] NetworkX Developers. Networkx documentation v1.10, 2015.

[27] Zaid Alyasseri, Kadhim Al-Attar, and Mazin Nasser. Parallelize bubble and merge sort algorithms using message passing interface (mpi). 11 2014.

[28] Gianvito Urgese, Francesco Barchi, Emanuele Parisi, Evelina Forno, Andrea Acquaviva, and Enrico Macii. Benchmarking a many-core neuromorphic platform with an mpi-based dna sequence matching algorithm. *Electronics*, 8(11):1342, 2019.

[29] Louis Blin, Ahsan Javed Awan, and Thomas Heinis. Using neuromorphic hardware for the scalable execution of massively parallel, communication-intensive algorithms. pages 89–94, 12 2018.

[30] Q. Xue, J. Xie, J. Shu, H. Zhang, D. Dai, X. Wu, and W. Zhang. A parallel algorithm for dna sequences alignment based on mpi. In *2014 International Conference on Information Science, Electronics and Electrical Engineering*, volume 2, pages 786–789, 2014.

[31] Wikipedia. Pagerank — wikipedia, l'enciclopedia libera, 2020. [Online; in data 3-ottobre-2020].