POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Methods to exploit core-parallelism to improve configuration and results gathering phases of SNN simulations in a many-core neuromorphic platform

Relatori
Prof. Andrea Acquaviva
PhD Gianvito Urgese
Dr. Francesco Barchi

Luca Peres

Anno accademico 2017-2018

Summary

Nowadays, the neural computing approach is widespread and generally accepted. This kind of paradigm allows performing brain-related simulations. In this dissertation, all the work has been performed on the SpiNNaker system, an architecture developed by the APT Group of The University of Manchester as part of the Human Brain Project. These kinds of system are called *Neuromorphic platforms* and are used for simulating Spiking Neural Networks (SNNs).
This SpiNNaker platform is built as an asynchronous multicore architecture composed of nodes, each one containing 18 cores, connected to form a hexagonal toroid grid. These nodes communicate by means of packets exchanging managed by special routers which allow multicast transmissions. The information exchange with the SpiNNaker system can be performed through a 100 Mb/s Ethernet interface. Despite the advantages of the architecture, for example the high availability of hardware resources and the low power consumptions, some critical aspects still exist. One of the biggest issues is the time necessary to retrieve and send data to the board. This is due to the fact that only a chip is connected to the Ethernet interface.
This work tries to address this bottleneck by acting on two phases that can be collocated before and after the execution of the target application (the simulation of SNNs).
It is possible to split the work in two major sub-topics: the first part, performed during my traineeship at The University of Manchester, focuses on optimising the existing data extraction protocol. The second part, focuses on the design of an improved version of the data upload phase. In order to improve the extraction phase of simulation results, several protocols have been developed to extract in parallel the data coming from different sub-regions of the SpiNNaker system.
Regarding the simulation configuration phase, some solutions have been implemented in order to compress the information to be sent to the system and to efficiently use the multicast transmissions by exploiting the high parallelism of the machine.

Contents

Chapter 1

Introduction

Nowadays neural computing is widespread and globally accepted. Different types of neural networks have been defined and new computing approaches have been formulating in order to better understand how the human brain works and to build Artificial Intelligence. This work focuses on the SpiNNaker Neuromorphic platform and on its role of Spiking Neural Networks(SNN) simulator.
This first chapter provides an introduction to the Neuromorphic computing approach (presenting the current platforms) and to Neural Networks in general.
One of the biggest bottlenecks in Spiking Neural Networks simulations on the SpiNNaker system is the time necessary to exchange data with the environment. For large simulations, it is necessary to wait for hours until data is completely loaded. This work focused on the idea to reduce this time. It is possible to divide all the work into two steps: the data extraction and the data upload.

1.1 Neuromorphic Computing

The Neuromorphic computing concept was developed by Carver Mead during the late 1980s. A Neuromorphic system is a massively multicore system composed of simple processing units and memory elements communicating by means of message exchanging[20]. This type of approach aims at simulating the brain behaviour using design principles based on biological nervous systems.
Neuromorphic systems differentiate from traditional multicore systems in the way in which memory and processing are organised. Indeed, in this case, memory is distributed with processing units and not centralised and physically separated from the cores. Thanks to this, it is possible to avoid the traditional bottleneck of the access to memory present in the classical Von Neumann architectures.
The main idea behind this kind of systems is to process information in an event-driven way (the processing units remain idle until an event is presented, then a reaction is triggered and, after that the units return to the idle state), acting in an

asynchronous way. Thanks to this feature, Neuromorphic systems are much more energy-efficient than traditional multicore systems.
This idea comes from biology, indeed the human brain is composed of billions of neurons connected by synapses, working asynchronously with power consumptions lower than a lightbulb[7][10].
Another peculiarity of the Neuromorphic systems is the high number of interconnections between the processing units which speeds-up and simplifies the communications between the cores.

1.2 Biological background and Neural Networks

1.2.1 Neurons, Synapses and Neural Networks

The basic unit of a neural network is the neuron (FIG. 1.1). A neuron is a specialised type of cell able to change its configuration in response to external stimuli.
A neuron is composed of three parts:

- Cell Body (elaborates and produces signals)
- Axon (output line that branches off giving the chance to connect to thousands of neurons)
- Dendrites (input line to the neuron that can receive signals from other neurons)

The Cell Body sums up all the input signals, if the result is above a certain threshold, the neuron activates and produces an impulse increasing the membrane potential (measurable difference in electrical potential between the interior and exterior of a biological cell), this impulse is eventually propagated over the Axon to be sent to other neurons, otherwise the neuron remains dormant.
Axons are connected to other neurons' dendrites through synapses.

Synapses are special structures located on the axons which communicate with receptors on the dendrites (FIG. 1.1). The area between the axon and the connected dendrite is called Synaptic cleft, through this, the excited neuron sends the signal. This signal is carried out by neurotransmitters, which are fluids emitted by the movement of the electric signal through the axon. At this point, electrically charged atoms flow in or out the dendrite altering the charge of the receiver. The frequency of impulses to be sent is adjusted depending on the sum of all the input signals received on the dendrites[12].
There are two possible types of signals:

- excitatory (increases the frequency of impulses generated)
- inhibitory (decreases the frequency of impulses generated)

A Neural Network is a network composed of neurons. It describes a population of neurons physically interconnected to each other. The communication happens by means of an electrochemical process as described above.

An Artificial Neural Network(or ANN) is a mathematical model composed of

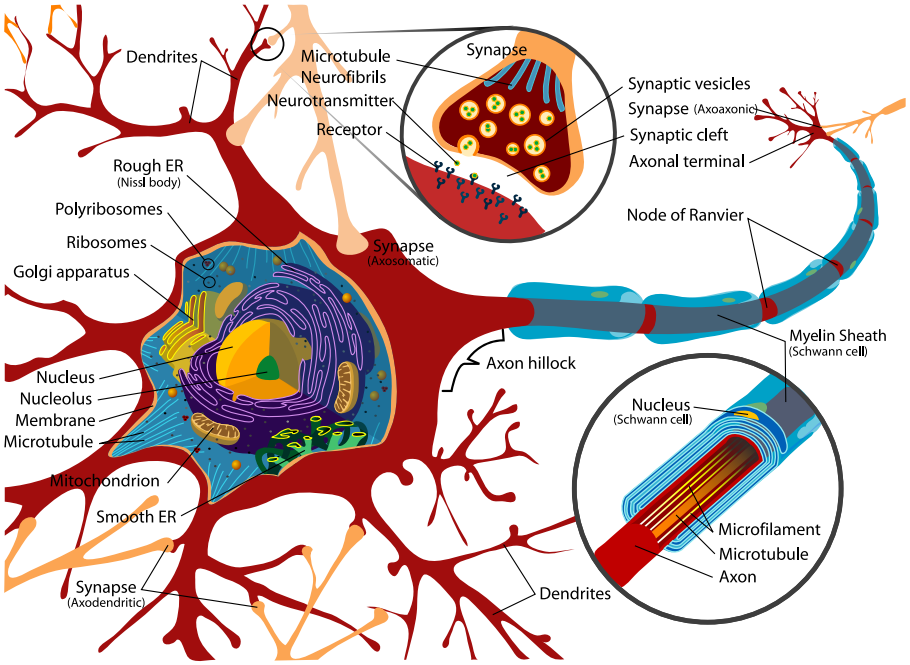


Figure 1.1: Neurons

artificial neurons. This type of network is an adaptive system which changes its structure depending on both external and internal information flowing through it during the learning phase (the initial configuration phase).

1.2.2 Spiking Neuron Model and SNN

A more accurate type of neural network is called Spiking Neural Network(or SNN). SNNs are based on the Spiking Neuron Model.
The Spiking Neuron Model is the mathematical description of cells which generate sudden electrical potentials a few milliseconds long through their membrane. This type of cell is the main signaling unit in the Nervous System.
The most common model is called Integrate-and-Fire (or IF). This type of model represents the neuron in the time domain as a current, defined as:

$$I(t) = C_m \frac{dV_m(t)}{dt}$$

It can be noticed that the behaviour is the same as a capacitor (whose capacitance is C_m and voltage $V_m(t)$).

By applying an input current, the membrane voltage increases up to a certain threshold. After that, an impulse is generated and then the voltage is reset to the rest potential. By increasing the input current the firing rate increases as well.
A more accurate description of this model considers a certain period of time called refractory time in which the neuron doesn't react to any stimuli. In this way, the frequency is limited[9].

$$f(I) = \frac{I}{C_m V_{th} + t_{refractory} I}$$

1.3 The Human Brain Project

The Human Brain Project (HBP) is an ambitious 10-years project launched in 2013 by the European Commission's Future and Emerging Technologies (FET) scheme, whose objective is to realise a complete simulation of the human brain. The idea behind the HBP it to build a collaborative ICT-based scientific research infrastructure to allow researchers across Europe to advance the knowledge in the fields of neuroscience, computing, and brain-related medicine.[1]
The main objectives are:

- To create an infrastructure for brain research, cognitive neuroscience and brain-inspired sciences
- To collect and organise data to describe the brain and diseases related to it
- To simulate the brain behaviour
- To build models for the brain
- To develop brain-inspired computing

The project is divided into 12 Subprojects (SPs). The topics of this work are related to the Subproject 9: *Neuromorphic Computing Platforms*.
This SP is composed of two different research projects: the BrainScale system (based in Heidelberg) and the SpiNNaker system (based in Manchester). The latter is the main topic of this dissertation.
Both the approaches are related to the development of a next technological generation and its integration. In addition, this Subproject focuses also on establishing principles for brain-like computation and for computational capabilities through learning.
The SP9 is organised in 5 Work Packages (WPs)[1] that are:

- Platform Software and Operations (focused on maintaining and developing software methods and tools for the neuromorphic hardware systems)

- Next-Generation Physical Model Implementation (focused on creating hardware systems to implement massively parallel, physical models of brain cells, circuits and networks)

- Next-Generation Many-Core Implementation (focused on creating hardware systems to implement massively parallel, many-core implementations of brain cells, circuits and networks)

- Computational Principles (focused on developing principles to enable brain-like computation, learning and cognition in neuromorphic systems by using brain activity and plasticity data)

- Platform Training and Coordination (focused on providing documentation and training for the neuromorphic systems developed inside the Subproject 9)

1.4 Current technologies

There are different approaches to neuromorphic computing both inside and outside the Human Brain Project. This section aims at describing the most relevant attempts in this area.

1.4.1 BrainScaleS

Besides the SpiNNaker system, inside the HBP, there is another Neuromorphic Computing Platform. This is called BrainScaleS and has been developed at the University of Heidelberg.
The main idea behind this project is to use above-threshold analogue circuits to implement physical models of neuronal processes, that exploit the analogy between electronic circuits and the ionic circuits in biological neurons.
BrainScaleS uses wafer-scale integration in order to deliver analogue neurons. On the same wafer, there are 48 copies of the same circuit (reticle) which are interconnected through an additional metal layer. Each reticle holds eight High-Count Analogue Neural Network (HiCANN) dies implementing 512 adaptive exponential Integrate-and-Fire neurons and over 100 000 synapses.
Communication happens in two layers: the first layer operates within a wafer connecting die-to-die through cross-bar switches, while the second one is used to connect different wafers through FPGAs.
BrainScaleS hardware is available in a portable form (a single HiCANN die) and as a 20-wafer platform incorporating a cluster server acting as host and supervising the operation of the machine (also executing network mapping functions)[8].

1.4.2 Intel Loihi

Loihi is an ambitious project for a neuromorphic many-core processor produced by Intel. It features a many-core mesh comprising 128 neuromorphic cores, 3 embedded x86 processor cores and off-chip communication interfaces that enlarge the mesh in 4 planar directions to other chips. Communication happens by means of packets that are conveyed through an asynchronous NoC. All the messages types come from an external host CPU or from the on-chip x86 cores.
Each neuromorphic core can implement 1024 spiking neural units grouped into sets of trees constituting neurons.
All the logic in the chip is digital and implemented in an asynchronous bundled data design style. This allows to generate, route and consume spikes in an event-driven way.
The neural network mapped to the Loihi architecture is a directed multigraph structure, each Loihi core includes a programmable learning engine able to evolve synaptic state variables over time as a function of historical spike activity. Loihi is able to autonomously modify the synaptic variables according to programmed learning rules. All the other network parameters remain constant unless they are modified by x86 core intervention[4].

Chapter 2

The SpiNNaker System

The SpiNNaker (Spiking Neural Network Architecture) system has been developed at The University of Manchester in the context of The Human Brain Project (inside the Subproject 9). The idea behind it is to create a massively parallel many-core computer inspired by the mammalian brain, trying to emulate its connectivity.
The global system is composed of 1,036,800 cores arranged in boards containing 800 processors each and of 7 TBs of RAM distributed throughout the system[5].
The communication happens by means of small data packets exchanged through an internal network.

2.1 The Hardware Configuration

2.1.1 The SpiNNaker chip

The basic building block of the SpiNNaker System is the SpiNNaker chip (FIG. 2.1), a SiP (System-in-Package) composed of 18 ARM968 cores (a 200 MHz clock frequency core without Floating Point Unit), an internal 32 KBs shared System RAM (SysRAM) and a 128 Mbyte off-die shared SDRAM.
Each core has two private memories: the ITCM (Instruction Tightly-Coupled Memory) and the DTCM (Data Tightly-Coupled Memory) whose sizes are 32 KBs and 64 KBs respectively. Finally, each chip is provided with a 32 KBs System ROM which contains the software necessary for the bootstrap.
Communications are managed by a concurrent hardware routing system incorporated in each chip. The SpiNNaker chips have six bidirectional links that allow connections in order to form a triangular lattice which is folded onto the surface of a toroid[6].
Inside a single chip the cores are organised as follows:

- 1 Monitor core (nominated at the start-up), which performs system management tasks

- 1 Fault tolerant/yield spare core, reserved for manufacturing yield-enhancement purposes
- 16 Application cores, used for application processing

The chips are then connected over boards of different sizes.
Currently, there exist two types of boards: The SpiNN-3 and the SpiNN-5.

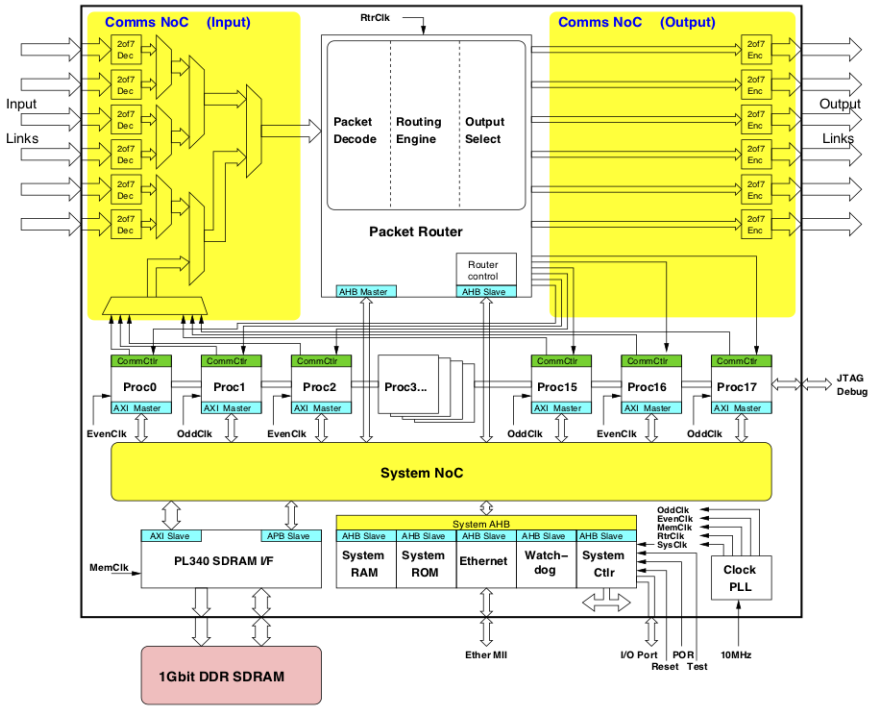


Figure 2.1: The SpiNNaker Chip.

2.1.2 The SpiNN-3 Board

The SpiNN-3 Board (FIG. 2.2) contains 4 SpiNNaker chips, one of them is called Master chip (labelled with the coordinates (0,0)), this one is connected to the Ethernet port of the board (standard RJ45 socket) and equipped with a serial ROM chip containing networking parameters. This external memory can be used to boot the system with specialised software (instead of using the internal ROM as boot source). The board can be connected to a host system through the Ethernet connection, that is used to upload and download data and code and to communicate with executing code[25].
The board is provided with a JTAG port for test purposes, an I/O port which brings out 8 GPIO lines from the master chip and a reset port to remotely reset the system.

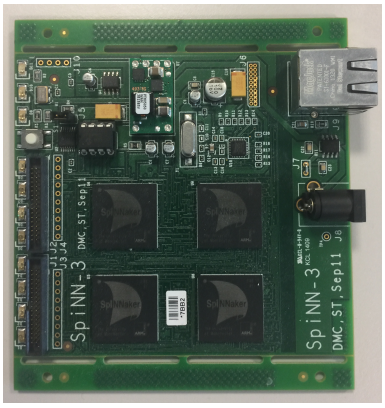


Figure 2.2: The SpiNN-3 Board.

2.1.3 The SpiNN-5 Board

The SpiNN-5 Board (FIG. 2.3) is more complex, it contains 48 SpiNNaker chips connected in a hexagonal arrangement, plus 3 Spartan6 FPGAs which can provide board-to-board interconnection through the SATA hardware.
A BMP (Board Management Processor) manages the power management, the resetting of FPGAs and chips and the clocking. Each SpiNN-5 has 2 100 Mbit Ethernet interfaces (each one with its own IP address), one connected to the chip (0,0) (the so-called Ethernet chip) and the other to the BMP[29].

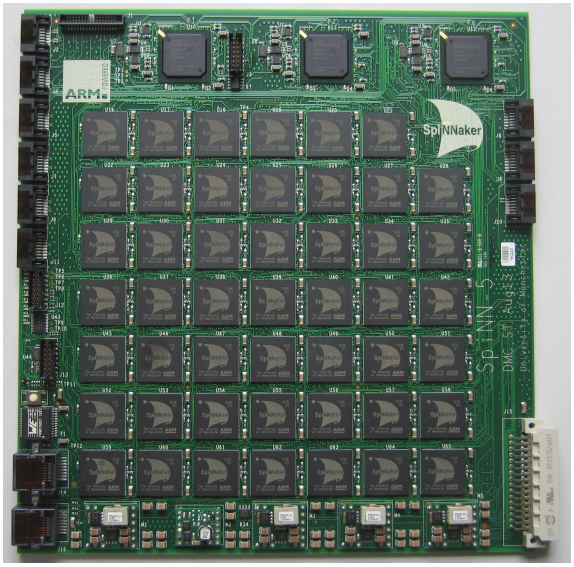


Figure 2.3: The SpiNN-5 Board.

2.1.4 The Hardware Routing System

The SpiNNaker system contains two different Networks-on-Chip (NoCs):

- The System NoC (Fig. 2.1), used for connecting the cores with the SDRAM interface and with system control and test functions
- The Communications NoC (Fig. 2.1), used for carrying packets between cores belonging to the same or different chips; the input is managed by a tree arbiter that merges the sources of packets (that can be both input links and the cores) into a single stream, while the outputs are sent over separate links for both on-chip and off-chip communications

All the packets are routed by the on-chip router following different rules depending on the type of the packet itself.
The router receives the packets one-at-a-time in the order decided by the Arbiter on the Communication NoC together with the identity of the receiver interface that the packet arrived through. The first performed operation is the error identification, then the packet, depending on its type, is passed to the specialised routing engine to be routed. The output of this stage will be a vector containing all the destinations for the packet. A final third stage, the emergency routing mechanism, handles the cases of failed or congested links. In this case, subsequent packets will be rerouted following a different path and the Monitor Processor will be notified, in order to detect if the fault is permanent or transient and to take proper counteractions.
The router contains a mechanism to check if a packet is out-of-date based on a 2-bit phase signal. Too old packets are dropped exactly like packets containing errors[13].

2.2 The Software Configuration

The software necessary to run a simulation can be split into two main groups: the software running host side, mostly written in Python, and the one running board side, written in C and Assembly code.

2.2.1 Host Side

The Host-side software, which is composed of several modules, is used to set the simulations and to map their structure on the SpiNNaker system. The simulations are firstly described using a high-level language called PyNN (which is based on the Python syntax) and then translated into structures ready to be loaded onto the SpiNNaker machine.
The modules used for SNN simulations running host side are:

- sPyNNaker: it contains the front-end specifications and implementation for the PyNN API. This module is a wrapper of the PyNN simulator and is responsible for translating the high-level description into populations of neurons to be loaded onto the SpiNNaker system. This module is independent of the PyNN version, there are two additional modules called sPyNNaker7 and sPyNNaker8 that provide support to specific versions of the PyNN tool (0.7 and 0.9 respectively)[19].

- SpiNNMachine: it is a Python abstraction of a SpiNNaker machine. Its functionality is to create a representation of the current state of the allocated SpiNNaker machine in terms of chips, cores, routable links, available routing entries and available SDRAM[17].

- SpiNNMan: this module is used to communicate with a SpiNNaker board, sending and receiving packets (using the UDP protocol) through the *Transceiver* class which is its main part. The SpiNNMan module allows to get the state of the machine, boot it with a specific version of the software, load application binaries and access the SDRAMs of the single chips[18].

- PACMAN: it is in charge of the *Partitioning and Placement* operation which is performed by creating a graph representing the Application that will be executed (called Application Graph) and to partition it in order to fit the SpiNNaker system (performing a correct distribution on the available resources and generating the routing information to implement the communication between vertices, obtaining the so-called Machine Graph). This module keeps track of the size of the available system and of the usable memory into each chip. PACMAN stands for Partition and Control Manager[15].

- Data Specification: this module is used to generate memory images (containing the Neuron Model Implementation configuration data needed during a simulation) for each SpiNNaker core involved in the simulation, starting from a specific set of instructions. The Data Specification tool is composed of two main parts: the Data Specification Generator (which is in charge of generating the Data Specification Language files, a set of text files containing the required instructions for generating the memory images) and the Data Specification Executor (which is in charge of executing the instructions contained in the generated files and of creating the memory images)[14].

- SpiNNFrontEndCommon: this module provides functionalities common to front ends translating application level programs into executables for the SpiNNaker system. The main implemented tools are the initialisation of the high-level machine object, the tool responsible to specify and write data into the chips' memories and the loader of executables onto the board[16].

2.2.2 Board Side

The software running board side is event-driven, this means that each core remains in *idle* state until an event (represented by an interrupt) triggers the execution. This approach allows achieving really low power consumptions. Indeed, when *idle*, the cores are in a low-power sleep mode sensitive to interrupts. As soon as an event is presented to the processors, they wake up, perform the associated task and then return in *idle* state. It is possible to configure different priorities for different events in order to set a hierarchy and to be sure to serve the most important requests first. Depending on the role of the core there can be different types of software running on it.

All the application processors, at the lowest level, run the SpiNNaker Application Runtime Kernel (SARK), which performs three main functions[30]:

- to provide a library for memory management, interrupt control and other low-level operations available to the application
- to provide mechanisms for the monitor core to communicate with application processors and for the host to control the application (with the possibility to read the memory)
- to initialise the core by setting the stack and some peripherals and then to call the main procedure of the application causing its start

The applications run on top of SARK and are composed of a set of callbacks that are triggered by the arrival of events which they are registered to. In support of this mechanism, a library called *spin1_api* is provided, containing all the necessary tools for event-driven programming on SpiNNaker.
Different types of events can trigger a callback, the most common are: the Timer tick, a packet reception, a DMA transfer completion and a configurable Custom User event.
The callbacks can have different priority levels, depending on the value of an associated ID. A negative value will associate the callback with a Fast Interrupt (FIQ), a zero value will correspond to a hardware IRQ and, finally, a positive value will indicate a queueable callback.
An application running on a specific core has the sole use of that core, usually many processors run the same application.
The board side software is built using a cross-compiler and converted into an APLX file (a format ready to be uploaded on SpiNNaker). Typically the monitor processor loads the APLX into an area of shared memory. At this point, the interested application cores load it and start the execution. The APLX file contains a which is used to initialise the data which is placed in DTCM and an executable part which

is stored at the bottom of the ITCM[26].

Regarding the monitor processors, during the bootstrap, a software called SC&MP (SpiNNaker Control & Monitor Program) is loaded. Its tasks are:

- to communicate with the Host system through SDP packets
- to supervise the operation of the chip
- to allow program loading on Application cores
- to act as a router for packets between two cores or with external internet endpoints

2.3 Communication Protocols

Each SpiNNaker chip has 6 bidirectional links which allow forming a triangular mesh topology inside the board. The internal communications happen through packet exchanging, mechanism managed by the routers. Currently, the information can be transmitted through 4 different protocols[13]:

- Point-to-Point
- Multicast
- Fixed-route
- Nearest Neighbour

2.3.1 The Multicast Protocol

The Multicast protocol is generally used to carry neural event information. This type of protocol allows reaching multiple destinations. A MC packet can be up to 72 bits long. The first 8 bits are called control field and contain the type of packet (2 bits), emergency routing and timestamp information (4 bits), a payload indicator (1 bit, it indicates whether the payload is present or not), and parity (1 bit, to identify errors). After that, there is a 32-bits field containing the routing key, which is inserted by the source and finally an optional 32-bits field used for the payload, depending on the value of the specific bit present in the control field. For the MC packets, the routing is performed through an associative subsystem. Here, the routing key is used to determine where to send the packet. The router keeps a table with 1024 look-up entries, each composed of a mask, a key and an output vector (containing the destinations for a given key). When a new MC packet is received from the router, this one compares the key inside the packet with each

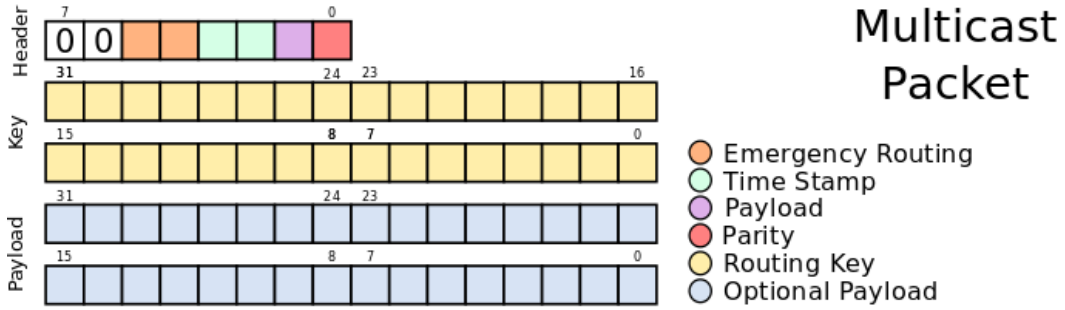


Figure 2.4: Multicast Packet Structure.

entry in the table (first filtering the packet key with the mask), in case of a match it sends the packets to the destinations specified in the output vector (could be one or more links towards other chips and/or one or more cores belonging to the same chip). These entries are stored inside a CAM present in the router of each chip. This memory must be set by software, as it is not initialised during reset. The matching operation is performed concurrently in order to improve performances and using a parallel ternary associative memory(the CAM mentioned above). In case of no match, a default routing is performed and the packet is sent to the output link opposite to the arrival input link[13].

2.3.2 The Point-to-Point Protocol

The Point-to-Point protocol is typically used for system management and control information purposes. This type of protocol is used to reach a single core on a given chip. A P2P packet can be up to 72 bits long, containing a control field 1 Byte long (2 bits for packet type, 2 bits for sequence code, 2 bits for timestamp, 1 bit as payload indicator and 1 bit for parity), source and destination ID (both 16-bits long, containing information about the two coordinates of the desired chip and the core number involved in the transmission) and an optional payload field 32 bits long. The P2P packets are routed using a look-up table. The router uses the destination ID field in the packet in order to understand which output link the packet should be sent to. For each possible destination ID, there is a 3-bit entry which is decoded and used to understand if the packet should be sent over one of the links (and in this case which one), or to the local Monitor Core or dropped[13].

2.3.3 The Nearest Neighbour Protocol

The Nearest Neighbour Protocol is used to support boot-time flood-fill and chip debug. A NN packet can be up to 72 bits long including an 8-bit control field (2

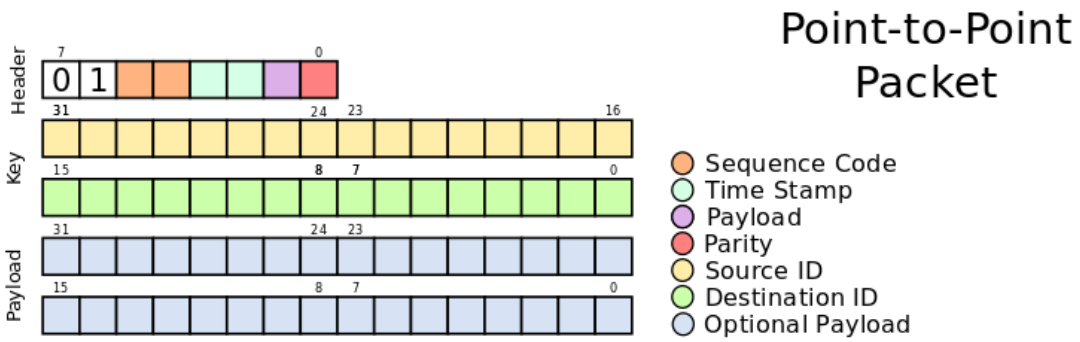


Figure 2.5: Point-to-Point Packet Structure.

bits for the packet type, 1 bit for type indicator, 3 bits for routing information, 1 bit for payload indicator and 1 bit for parity), a 32-bit address/operation field and an optional 32-bit payload field. The type indicator field can be set at *normal* (in this

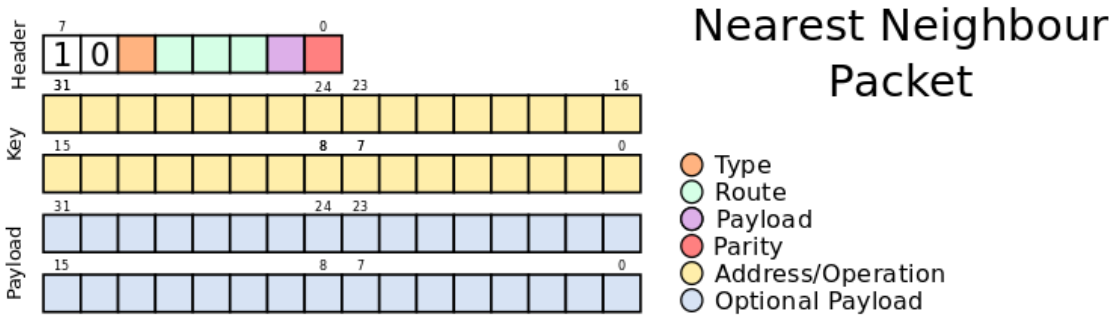


Figure 2.6: Nearest Neighbour Packet Structure.

case the packet will be sent to the Monitor Processor if it arrives from the outside or to the appropriate output link otherwise), or at *peek/poke* (in this case, the packet is used by neighbouring systems to access System NoC resources, this type of NN packet is generally used to check non-functioning chips, to re-assign the Monitor Processor or for test and debug purposes)[13].

2.3.4 The Fixed-Route Protocol

The Fixed-Route Protocol is used to convey application debug data back to the host computer, the destination is the Ethernet controller. Packets are routed by the content of a specific register. Each Packet can be up to 72 bits long. The first byte is reserved as control field (2 bits for package type, 2 bits for emergency routing, 2 bits for timestamp, 1 bit for the additional payload indicator and 1 bit for parity), 32 bits are used as payload and 32 additional bits can be used as optional payload

extension. FR packets are routed using the same mechanism as the MC ones. The

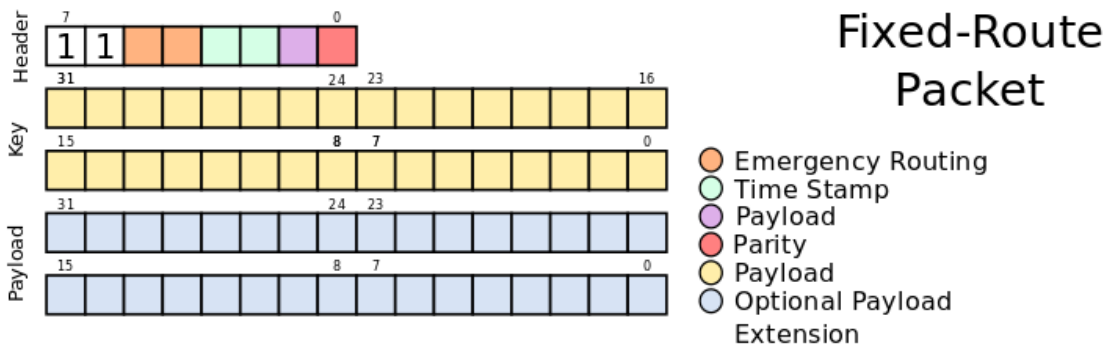


Figure 2.7: Fixed-Route Packet Structure.

only difference here is that packets do not have a key field, indeed they are always routed to the same output vector, ensuring a fixed path. The idea is to reduce at the minimum the distance towards the ethernet chip in order to facilitate data to go to the host system[13].

2.3.5 The SDP Protocol

An additional protocol can be used inside the SpiNNaker system, this is called SDP (SpiNNaker Datagram Protocol). SDP is a higher level protocol provided by the SpiNNaker APIs at the user level. This protocol can be used for both internal (core-to.core) and external (core-to-host) communications.
The current implementation of the SDP protocol is based on packets that can contain up to 256 bytes of data (to minimise the size of the buffer inside the SpiNNaker chips). Each packet is composed of a header used for addressing and control purposes and a data field which contains the payload. The length of the packet is managed by the agent conveying it in order to avoid the addition of a length field. The SDP Header is composed of 8 fields 1 byte long as shown in Figure 2.8. The Flag field is used to indicate if a response is expected or not, the Tag is used for external communication purposes, Source X, Y and Destination X, Y contain the coordinates of the source and destination chips respectively and D. and S. Port/CPU contain respectively the Virtual CPU ID and port of the source and destination cores.
Regarding the use of SDP for internal communications, the addressing mechanism is the same adopted by the SpiNNaker Point-to-Point protocol, the Source and Destination fields in the header indicate the X and Y coordinates of the involved chips, while the CPU fields indicate the desired core for the specified chip, and the Port bits are used to address a particular process (with 0 reserved for the kernel). The Monitor cores act as a Middleware between SDP and the NoC splitting each received SDP packet in fragments of 32 bits to be sent as P2P packets to the desired core.

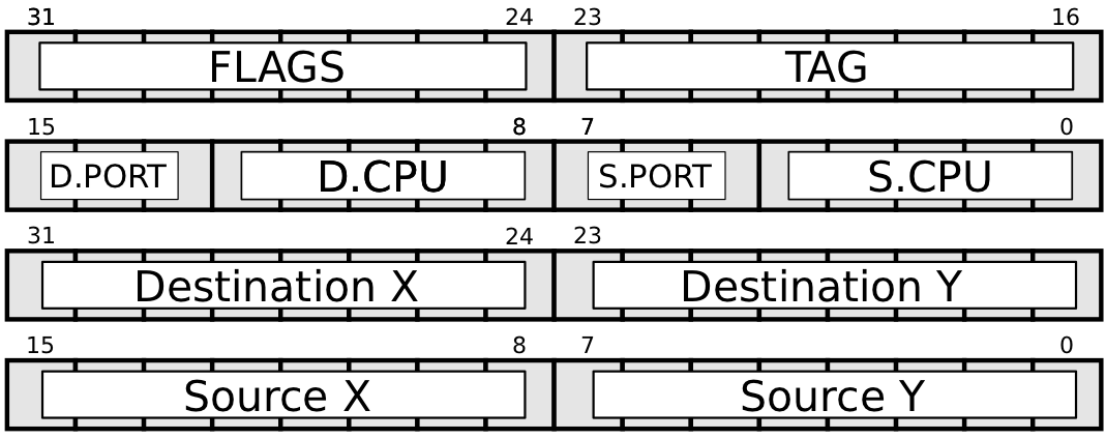


Figure 2.8: SDP Header Structure.

For reliability purposes, it is possible to implement an acknowledgement system, in which the sender Monitor Core enters in wait state (until the receiver one sends an ACK) after sending 16 P2P packets. When the transmission is completed the SDP packet is saved into the SysRAM and an interrupt on the interested core is triggered. In order to react to this interrupt the core must have a specific callback registered (for this reason it is necessary to manually register this callback in order to use SDP internally).
SDP can also be used for external communications. In this case, an SDP packet needs to be embedded into the data field of a UDP datagram (this field will also contain 2 bytes of padding used to align the start of the SDP packet to a 4-byte boundary, simplifying the processing inside the SpiNNaker system). To indicate that an Ethernet connected node is involved in the transmission, and that the packet will be routed onto the Internet, the Port/CPU field present in the SDP Header for that specific node will be set at the values Port=7 and CPU=3. This is necessary to specify that the Tag field contains a valid IPTag that has to be used.
The IPTags are 8-bit numbers used as indexes for a table available to Ethernet connected nodes, in which each of these Tags corresponds to an IP address and port. This mechanism is used to avoid to store the complete IP address in each SDP packet. IPTags can be permanent (created and removed using a *SC&MP* command) or transient (created when an SDP packet is received via UDP and a reply is expected and removed when the SDP reply packet is sent over the Ethernet interface)[27].

2.3.6 The SCP Protocol

The data field of an SDP packet could be formatted in order to follow the specifications of the SpiNNaker Command Protocol (SCP). This format consists of a

fixed-length header plus a variably sized data field.
This protocol is used for low-level interactions with the SpiNNaker system for debugging purposes and program loading. Commonly it is used to send a command to a specific core and to convey the response from that processor. The layout of SCP data is described in FIG. 2.9; for simplicity, in this case, each box represents a byte instead of a bit.
The first two bytes indicate the conveyed command in case of a command packet or, otherwise, the return code after the execution of that command in case of a response packet. The following two bytes are used to detect lost packets (indicating the sequence number of the command). The three *Arg* fields can be used as 32-bit arguments or return values. The remaining bytes (up to 256) are used for data. This protocol is used to initially control the SpiNNaker system from a host system

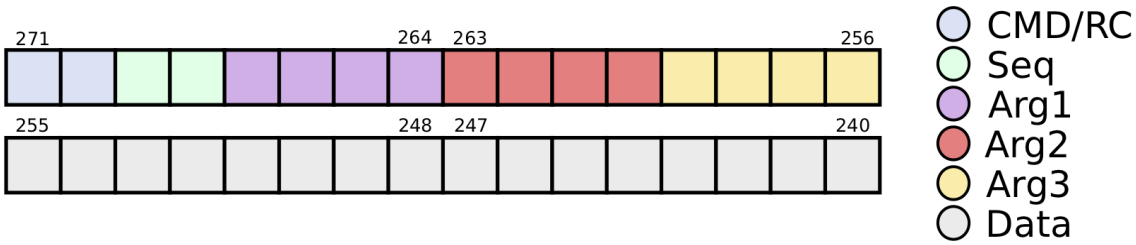


Figure 2.9: SCP Structure.

by sending commands to the Kernel running on every active core (that is SC&MP on Monitor and SARK on Application cores). These commands are typically used to download application programs and perform low-level functions[28].

Chapter 3

Current Implementations

In this chapter will be presented the current situation for both the data extraction and the data upload protocols in order to provide a general idea of the starting point of this work. Details about the required phases will be provided alongside with the existing weaknesses that have been addressed during this work.

3.1 The Data Extraction Phase

The data extraction phase is currently one of the biggest bottlenecks during SNN simulations on the SpiNNaker system. This phase is of paramount significance because it is necessary to retrieve the results of the execution for following elaboration or benchmarking.
A SpiNNaker system composed of half million cores can provide potentially 3.6 TB of data per run. Data extraction speed is limited to the Ethernet connection at 100 Mbps. Having each board a separate Ethernet interface, it is possible to extract data in parallel achieving theoretically a band of 60 Gbps on a 600 SpiNNaker boards machine.
Currently, there are three different approaches in order to extract data from the SpiNNaker system: the standard SDP Protocol, the Multi-packets in Flight (MPF) approach or the Direct Data Stream Process (DDS).

3.1.1 The SDP Protocol for Data Extraction

This approach is based on sending SDP messages from the host containing SCP requests for reading the SDRAM of the desired chip to the Monitor core on the Ethernet chip of the SpiNNaker board, in order to perform a data request. Data to be read will be fragmented in chunks up to 256 Bytes long.
As soon as the Monitor core of the Ethernet chip receives an SDP requiring data extraction, it forwards the SCP message to the Monitor processor of the interested

chip that starts reading the memory and fills P2P packets, which will be sent back to the Monitor core on the Ethernet chip. When the Monitor processor of the Ethernet chip has received 16 P2P packets, it builds an SDP packet and sends it to the host system. At the same time, an acknowledgment is sent to the Monitor core of the chip whose memory was read. This core, after sending the last P2P packet, goes in wait state. If an acknowledgment is received before a timer expires it will either start reading the following chunk of data or conclude the process in case no more data need to be read, otherwise it will send again the 16 P2P packets to the ethernet chip.
This protocol is reliable (because if some packets are lost they will be retransmitted thanks to the acknowledgment mechanism), but it is slow. It is, indeed, possible to achieve at most a data extraction speed of 8 Mb/s (and only on the Ethernet chip, while for the other chips the maximum speed registered is 1 Mb/s). This is due to the fact that acknowledgments and P2P packets slow down the transmissions inside the board and, furthermore, it is necessary to fragment memory requests in chunks 256 bytes long.
The advantages of this protocol are that it can operate while a simulation is running and the throughput remains stable regardless of the size of the read memory.

3.1.2 The Multi-Packets in Flight Approach

The MPF is built on top of the SDP protocol. The idea here is to send multiple SCP read request at the same time (instead of a single read command) and to pipeline the reading of large blocks of memory. The data extraction for the single blocks is performed in the same way as the SDP protocol described above.
The maximum number of parallel SCP messages that can be sent is 8 and new requests can be asserted (4 at a time) only after 4 messages have been responded.
Using this protocol it is possible to achieve up to 32 Mb/s as download speed when reading from the Ethernet chip (and 8 Mb/s for the other chips). This is due to the fact that, while a chip is being acknowledged, other messages are served.

These two protocols have drops in performances for all the chips not connected to the Ethernet interface and data requests need to be fragmented.

3.1.3 Direct Data Stream Process

The Direct Data Stream Process has been developed in order to overcome the limits highlighted so far. It uses the FR packets for internal SpiNNaker communications and reserves one core per chip.
This protocol exploits the possibility to set the timeouts of the SpiNNaker routers to a value that let them never drop a packet, implementing a reliable internal communication system, the drawback is that deadlocks become more likely if other packets

flow through the routers while DDS is running. For this reason, this protocol must run while nothing else is executing on the SpiNNaker system.
Thanks to this setting of the routers it is possible to avoid the acknowledgment phase increasing the performances. This implies a first phase for setting routers timeouts and a second one to restore them to the application values that need to be performed sequentially by using SCP messages.
At the beginning of the transmission, a single SDP packet is sent by the host system asking for data, this one contains the full size of data to be transferred and its starting position in memory. The destination core, which is the additional core reserved on the chip whose memory needs to be read will start a DMA cycle to retrieve data from memory. This processor is called ExtraMonitor core and, upon the DMA transfer completion, it generates the FR packets to be sent back to the Ethernet chip and sets a new DMA cycle. On this chip, an additional core has been reserved as Packet Gatherer. Its role is to collect all the FR packets and to generate the SDP that will be sent to the host machine.
The FR packets contain a key and a payload both 32 bits long, the key identifies the type of packet, while the payload assumes different meanings depending on the key:

- First packet of the stream: the key has a unique value indicating the beginning of a new stream, while the payload contains the maximum sequence number for the outgoing SDP packets
- Intermediate packet: the key has a unique value indicating a data packet, while the payload contains a chunk of data 32 bits long that will be added to the SDP packet which is being built
- Last packet of the stream: The key has a unique value indicating the end of the stream, while the payload is empty

The Packet gatherer fills SDP packets and sends them to the host system, a last FR packet key reception forces it to send the last SDP packet even if it is not complete. Each generated SDP packet contains a sequence number as the first 32-bit word in the payload indicating the position in the stream. The first SDP sent to the host contains the maximum sequence number.
The host system maintains a buffer whose size is the same of the data to be read, as soon as a UDP packet containing an SDP comes from the SpiNNaker system, the host extracts the payload and, depending on the sequence number, stores the data in the correct position of the buffer. After having received the packet with the maximum sequence number, the host system loops over the received sequence numbers looking for missing values and builds up SDP packets having as payload all the missing sequence numbers. In case no elements are missing, the host concludes the transmission and returns the data read (and router timeout are reset to

the application values). If the host system does not receive packets for at least one second it will start checking for missing sequences.
The retransmission phase begins with the identification of the missing sequence numbers. After this step, the host system starts sending SDP packets containing the numbers of missing sequences to the SpiNNaker system. each SDP packet can contain up to 67 sequence numbers (the first word of payload is reserved for indicating that the datagram contains a retransmission request), except for the first one that has to contain up to 66 elements (int this case the first word is reserved for the retransmission phase start ID and the second will contain the number of SDP packets that will be sent from the host).
Once it has received all the missing sequence numbers, the SpiNNaker system starts sending the requested data using the same procedure described above.
This process continues until no sequences are missing. In order to deal with errors, the host system has a limit in the number of retries for the same request, this is currently set to 20. In case this limit is reached, the host system will throw an exception indicating that an error has occurred.
This protocol outperforms the above-presented twos, provided that the extracted data is at least 100 KB long. The registered transfer speed for the DDS protocol is 38 Mb/s, regardless of the position of the chip containing the requested data (this is a great improvement with respect to the previous protocols which had a drop in performances for the non-Ethernet chips). This increase is mainly due to the fact that DMA requests are much more efficient than direct memory reading operations and acknowledgments are not necessary anymore.
However this gain has a cost, one core needs to be reserved as ExtraMonitor on each chip (and the Ethernet connected chip needs to additionally maintain the Packet Gatherer) and nothing else can be executed on the SpiNNaker board while this protocol is running.

Despite the good performances, this protocol is still presenting some problems: in case of lossy networks (like a Wi-fi one) or large data transfer requests the retransmission phase could become a real bottleneck, resulting in drops in performances due to the time wasted in requesting the missing sequences. Furthermore, the missing sequence research process is an operation whose complexity is $O(N)$ in the total number of sequences, because it requires to check all the received fragments. This can become a problem for simulations requiring exchanges of large amounts of data. At last, the performances of the Python code running host side for extracting data are far from the theoretically reachable ones (100 Mbps). For these reasons, further optimisations have become necessary.

3.2 The Data Specification Module

When sending data to the SpiNNaker system, the Data Specification phase is one of the most critical in terms of execution time and resource management. As described before, this module generates, for each core, a file containing a set of commands that will be executed in order to generate all the memory images for preparing the environment for the simulation.
The Data Specification phase is divided into two steps, both currently executed on host: the Data Specification Generation and the Data Specification Execution. Each of these steps is performed by a submodule.

3.2.1 Data Specification Generator

The role of the Data Specification Generator (DSG) is to generate the flow of instructions in a pre-defined meta-language for each core and to write them on a specific binary file that will be executed by the Data Specification Executor.
These instructions allow performing operations such as memory allocation, data writing and routine execution. Each instruction is coded on one or more 30-bit words as shown in FIG. 3.1.
The first field indicates the number of words the command will take (up to four), the second one represents the command code, while the third bit-field provides information about which of the following three register fields are used. If a particular register field is not in use, it can be rearranged for command-specific purposes[3].

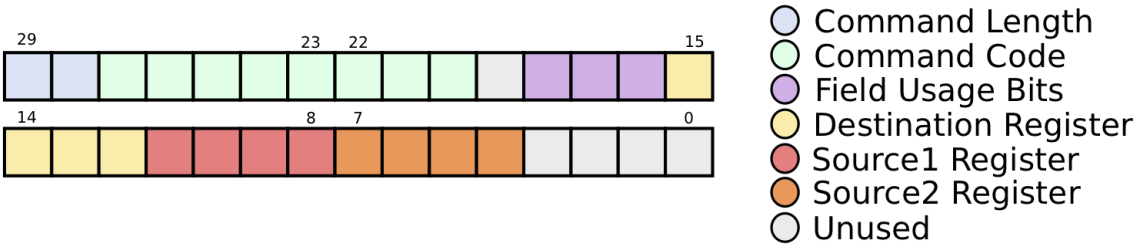


Figure 3.1: Command structure.

3.2.2 Data Specification Executor

The Data Specification Executor (DSE) receives all the files generated by the DSG for each core, unpacks each instruction and generates the memory images for the processors following the structure provided by the SpiNNMachine module (which gives a representation of the usable memory of the machine). All these operations are performed host side.
At this point, the memory images are saved into configuration files which are read

and sent through the Ethernet interface to the SpiNNaker system using the SCP protocol. All the data sent in this phase is saved into the SDRAM memory of the involved chips.

There are different drawbacks using this this protocol: first of all SCP is based on the SDP protocol which is built on top of UDP for the external communications which is an intrinsically unreliable protocol. Furthermore each SpiNNaker board has a single Ethernet interface which is directly accessible from a single chip only. For this reason, the available communication bandwidth is limited and sending memory images for all the cores, especially for large simulations, could become a really slow operation.
In addition, by executing the data specification host side, it is not possible to exploit the high parallelism and low-power nature of the SpiNNaker system, resulting into a sequential behaviour.
For these reasons some attempts at moving the DSE on the SpiNNaker system have been done[14][23].

3.2.3 Data Specification Executor on-board

In order to overcome the limitations of the DSE on host, a version of this module running on the SpiNNaker system has been implemented. This version comes as an executable loaded on each Application Processor. First of all, through the SCP protocol, all the commands generated by the DSG are sent to the SDRAM of each chip to be stores different regions depending on the core they have been generated for. After this phase, all the Application Processors involved are loaded with the DSE executable which will execute the commands stored in memory and generate the data structures directly on board.
Thanks to this approach, all the effort necessary for generating the data structures has been moved board side, better exploiting the parallelism of the system. However, the SDRAM memories are still loaded serially because of the single Ethernet interface accessible by a single chip and this phase is still too slow due to the DSG generated command streams sizes[23].

3.2.4 Data Specification Executor on-line

In order to overcome the remaining bottleneck, a modified version of the DSE on-board has been developed. This is called DSE on-line. In this protocol, the DSG commands are executed as soon as they are received on board instead of being stored before.
The main idea is to follow the producer-consumer paradigm. First of all, a slightly

modified version of the DSE on-board is loaded on the SpiNNaker system, this version exploits the event-driven nature of SpiNNaker. The DSE on-line is sensitive to two specific events: a hardware interrupt indicating an SDP reception and a software interrupt that notifies that an SDP packet is ready for being processed.
The *Producer* callback is in charge of moving a received SDP from the mailbox (the SysRAM) to the SDRAM area reserved to the core on which it is being executed. While the *Consumer* callback extracts the payload from the SDP packet, reconstructs the fragmented DSG commands and executes them. This second callback can be pre-empted by the *Producer* one in order to avoid to saturate the SysRAM.

The issues with this version are related to the loading phase on the root processor on the Ethernet chip, which is the one managing the SDP reception and their fragmentation in Point-to-Point packets to be spread over the system. Indeed, this single core has to forward the packets to all the chips. This phase can become critical especially for large simulations in which the number of communications increases consistently[23].

3.3 Support Libraries

In order to implement a more efficient communication system for SpiNNaker, there has been developed a set of APIs.
The three new libraries are called: Spin2, SpinACP and SpinMPI. Their positioning inside the software stack is shown in FIG. 3.2.

3.3.1 The Spin2 Library

The Spin2 library is an extension of the Spin1 API, it provides an interface that allows performing broadcast transmissions using the multicast message system and implements a Synchronisation mechanism.
Each MC packet is 72 bits long, the first 8 bits are the control field (the same as the one presented in FIG. 2.4), while the remaining 64 bits are detailed in FIG. 3.3. The first 20 bits are used to indicate the sender core (8 bits for the X and 8 for the Y coordinates and 4 bits for the core number), the following 2 are used as control field that indicates the position of the packet inside the MC stream, 3 bits are reserved for the Synchronisation protocol, 7 bits are used as packet ID and the last 32 bits are the payload field[2].
The Spin2 API is able to automatically generate the MC routing entries. Depending on the position of the router with respect to the sender there can be 4 scenarios:

- In case the router is on the same chip of the packet source, the rule will force a send on all the 6 output links

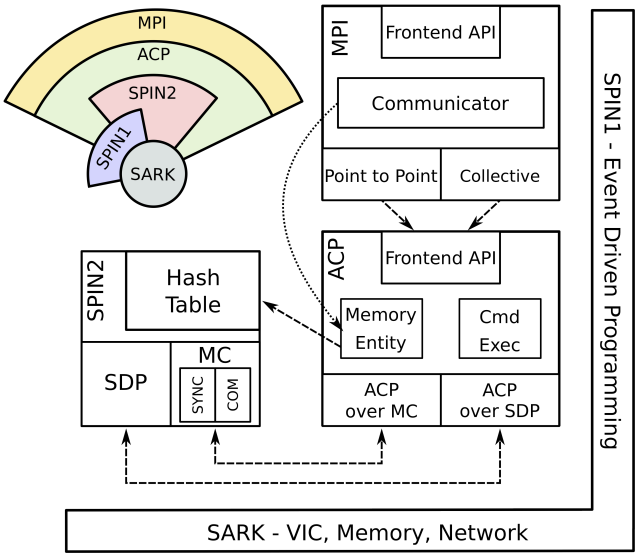


Figure 3.2: New APIs Hierarchy.

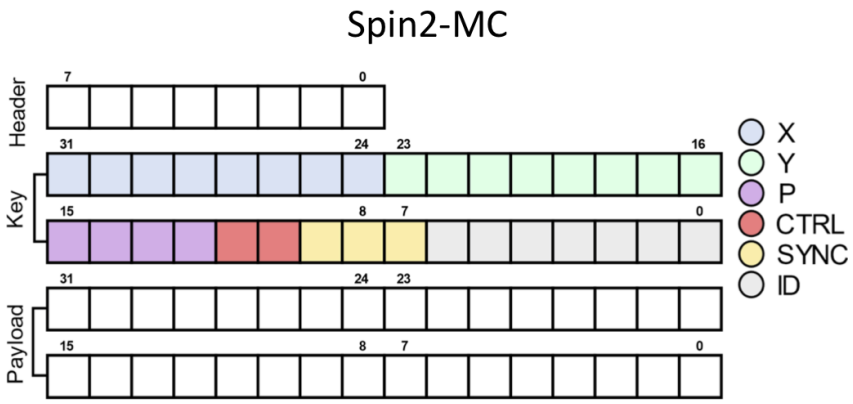


Figure 3.3: The Spin2 MC packet structure.

- All the chips on the x^+ axis will forward the packet on the E, NE and S links, while those on x^- axis will spread packets to W, SW and N links
- All the chips on the y^+ axis will use N and NE links, while those on y^- will send the packets on S and SW links
- Other: all the remaining chips will forward the packets on the opposite direction with respect to the reception link

The Synchronisation protocol is based on a layered structure. On a Spin5 there are three levels:

- SYNC1: Chip Level (implemented by all the chips), all the Application processors send a Spin2 MC Sync1 packet to the chip synchroniser (generally the core with ID 1)
- SYNC2: Ring Level, the synchronisers here are the chips whose X coordinate equals the Y coordinate. All the chips belonging to the same ring (in other terms sharing one coordinate with the ring synchroniser) will send a Spin2 MC Sync2 packet to the ring synchroniser
- SYNC3: Board Level, the board synchroniser is the chip (0,0), all the ring synchronisers send a Spin2 MC Sync3 to the board synchroniser after the SYNC2 level is completed

At the completion of the SYNC3 level, the board synchroniser send a $SYNC_{free}$ packet in broadcast to conclude the synchronisation process[2].
A section of a Spin5 indicating the Sync levels is shown in FIG. 3.4.

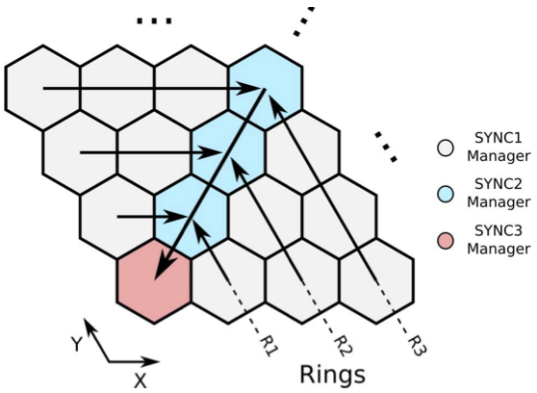


Figure 3.4: Spin5 Sync structure.

3.3.2 The SpinACP Library

The SpinACP library provides a framework for defining and executing commands throughout the Application Processors of the SpiNNaker system at the user level.
There are two types of commands which can be performed: built-in commands and user-defined commands. For these two purposes, SpinACP maintains two hash tables (structures defined and allocated inside the Spin2 library) in order to maximise the performances. When an ACP packet is received, the type of command is checked and the correct hash table is accessed.
The user-defined commands are configured as callback functions. In case a new command is sent using ACP, it is saved into the Command Callbacks hash table in order to be executed again in the future, otherwise, the hash of the index of the

existing command is computed and the associated callback is executed.
The built-in commands act on objects called *MemoryEntities* which are structures having an ID (used to get the access to the specific *MemoryEntity* stored in the hash table) and a pointer to a specific memory region on which the common CRUD (Create, Read, Update, Delete) operations can be performed.
ACP stands for Application Command Protocol, there are two ways to implement the communication in ACP.
The first one is ACPoverSDP, in this case, an ACP packet becomes the payload of an SDP packet (similarly to the SCP implementation) and it can be sent to the desired Application processor using the Point-to-Point protocol. This is also the way in which the host system acts on the *MemoryEntities*.
The other possible implementation is ACPoverMC, that exploits MC packets. In this case, an ACP packet is fragmented in 32-bit long chunks and sent to the desired Application cores through the Multicast protocol[2].

3.3.3 The SpinMPI Library

SpinMPI is the last support library and it is built on top of the previously described twos. The idea behind it is to implement a high-level Message Passing Interface (a programming model specification for distributed memory architectures). A host side runtime library loads the application on the SpiNNaker system and informs each processor about the whole context (identified by the number of involved rings) and the execution rank through ACP.
When launching MPI on SpiNNaker, first of all, a set of callbacks to manage SDP and MC packets are registered using ACP, after that the Spin2 API is used to register the routing entries for the Multicast transmissions and, finally, all the cores are checked to have been correctly configured.
Communications inside MPI can be Point-to-Point (with Blocking or Non-Blocking, Synchronous or Asynchronous, Buffered or Unbuffered features) or Collective (synchronism operations, data reduction or data movement). This second type is used for 1-to-All, All-to-1 or All-to-All communications[2].

Chapter 4

The Data Extraction Protocol

This chapter will provide an analysis of the attempts done, during this work, in order to improve the Direct Data Stream protocol, used for extracting the simulation results from the SpiNNaker system. All the work here described has been performed at The University of Manchester.

4.1 The Single-threaded C++ Version

The Python software running host side has demonstrated to be too slow for the DDS protocol, indeed the retransmission phase was engaged for every execution (and sometimes more than once), resulting in the slow down brought by computing the missing sequences and by performing their retransmission.
The idea to address this problem has been to rewrite the host side part of the DDS protocol in a more efficient language (possibly a faster compiled language instead of an interpreted one like Python), also with the idea of exploiting the possibility of downloading data using multiple parallel flows (impossible with Python because of its bad performances with multiple threads). The chosen one was the C++ language. The structure of the classes developed for this purpose is presented in FIG. 4.1.
The *SDPHeader* class contains all the methods necessary to define the header of an SDP packet, it maintains an internal structure with all the necessary fields on the correct number of bytes and has a function member that returns a char buffer containing the ordered fields.
The *SDPMessage* class internally stores an object of SDPHeader type plus a buffer for storing the payload, this class can return a char buffer containing the whole SDP message and also its total size.
The most important class is the *Host_data_receiver*, this one defines all the methods necessary to implement the data transfer.
An additional *main* file is used as the entry point to instantiate the *Host_data_receiver*

class and to start the transmission.
This protocol is a C++ translation of the Python version of the DDS protocol with the adaptation to the new language of all the data structures. This new version contains a library called *UDPConnection* (not shown in FIG. 4.1, because it is only a rewriting of standard socket functions) which allocates all the network resources (it creates a socket based on addresses and ports provided) and manages the data transfer at low level (creating specific wrappers for the functions used to send and receive data over the network provided by the standard C/C++).
Due to the changes brought by this translation (the creation and management of the socket and the transmission that is now performed externally by the C++ code and not by the SpiNNaker toolchain), an additional preliminary step has become necessary: the host system, before requesting the data, needs to set an IPTag on the SpiNNaker system in order to inform the board to respond to that specific address on the specified port (on which the allocated socket is listening).
Once the data transfer has been completed, data needs to be transferred to the software managing the simulation host side. To address this problem two possible approaches have been created.

4.1.1 The PyBind approach

The first possibility is to call the C++ code as an external library from the python code. This creates a perfect integration between the two languages and each C++ defined function can be directly called from the Python code as a Python function. This can be achieved by using the PyBind11 library, which is open source. This tool allows creating dynamic libraries starting from a C++ code which can be imported in Python. In order to do this, it is sufficient to add a specific function in the C++ code called *PYBIND11_MODULE* in which it is possible to specify the name of the class to be exported and the methods to be made visible and finally to compile the C++ code with the PyBind11 toolchain[11].
By importing the generated file, all the desired classes and methods are accessible from the Python code.
In this first approach, all the Python functions that were used for implementing the data extraction protocol have been changed with their C++ versions that, at the end of the transmission, returned the buffer containing the requested values.

4.1.2 External Process Call

The alternative approach to the PyBind Python binding is to call the C++ code as an external process. The code is compiled and the executable is called through the *Subprocess* Python library. Using this approach it is necessary to define an entry point that calls the library functions defined for the data transmission. This is performed by the *Main* file in FIG. 4.1 (not used in the PyBind approach). The

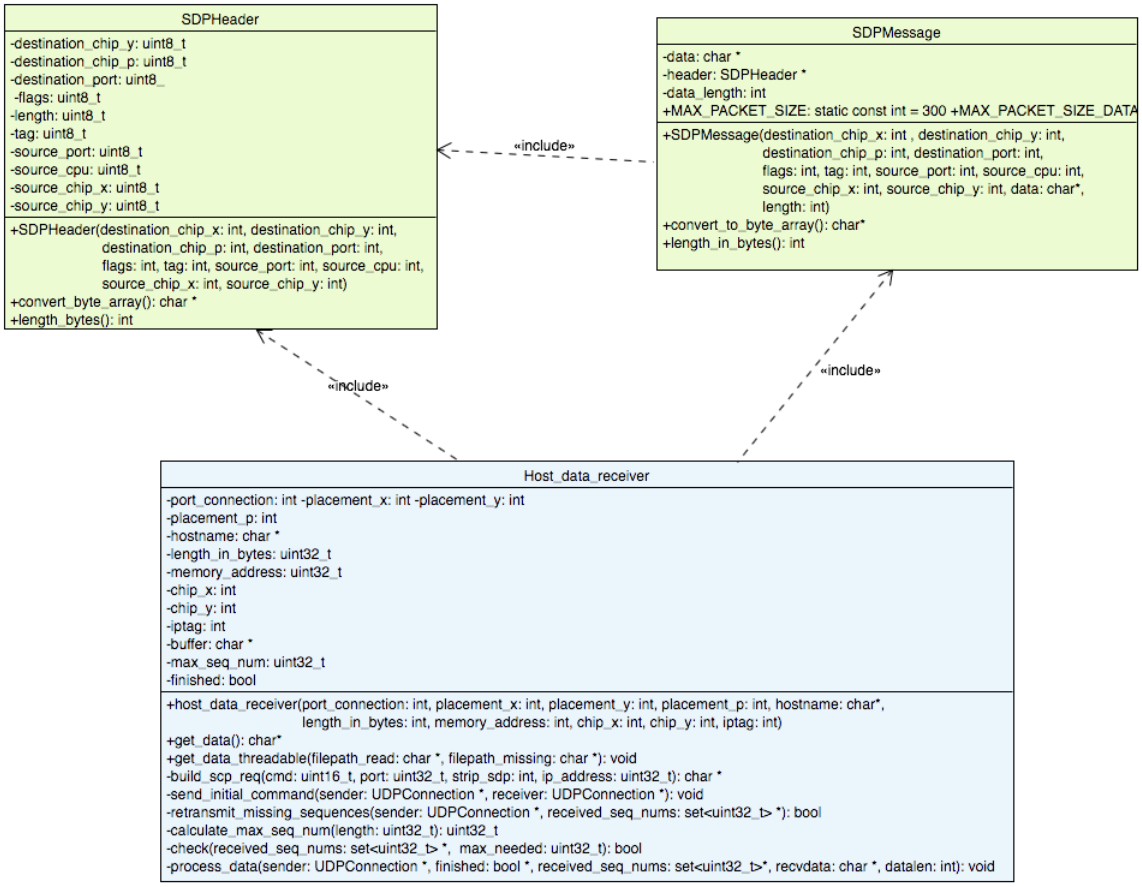


Figure 4.1: C++ classes structure.

code inside this file calls the same functions used for data download, but instead of returning a buffer containing the requested value, it prints it on a text file. This file is eventually read by the SpiNNaker toolchain (which is the caller of the external process) and data becomes available for the next elaboration phases.

4.2 The Parallel C++ Version

In several cases, the C++ version still presented some performances drops due to the fact that the data manipulation phase is too slow when compared to the packets arrival rate. The problem was that the host required too much time to extract the payload from an SDP packet and to save it into the buffer, and, during this period, UDP datagrams sent over the network went lost. This resulted in calls to the retransmission phase which caused performances drops.

In order to address this problem, an intrinsically parallel version of the C++ protocol has been developed. This approach exploits the producer-consumer paradigm and makes use of a queue.
The class structure is the same as the one presented in FIG. 4.1, with the addition of a queue as a shared resource. This new version, once sent the transmission request to the SpiNNaker system, creates two threads, named *ReaderThread* and *ProcessorThread*.
Its behaviour is summarised in FIG. 4.2.

The first one has the role of busy waiting on the socket for new UDP packets,

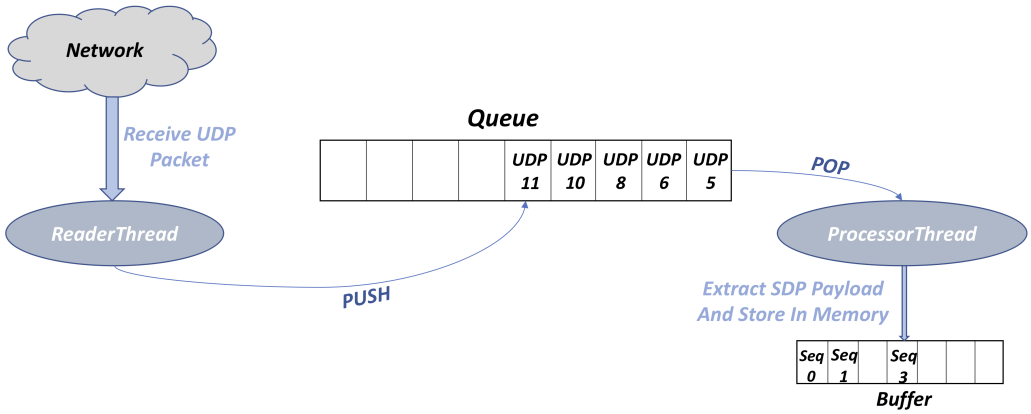


Figure 4.2: Parallel C++ Protocol.

as soon as a new datagram is received, this thread inserts it into the shared queue (each node of the queue is composed of the UDP packet plus an integer indicating its size).
The *ProcessorThread* instead, performs a busy waiting on the queue for new packets, as soon as the *ReaderThread* inserts a new element, the *ProcessorThread* extracts it and process the data in the same way as the single-threaded C++ version.
This approach allows separating the two phases of packet receiving and packet elaboration, consistently reducing the number of lost packets. Indeed now the software has not to wait for the completion of packet elaboration to check again on the network interface.
By adding the queue, the complexity in detecting the end of the transmission or an error identification has increased.
The error detection phase has been managed through a signaling system based on shared variables. Every time something goes wrong the executing thread raises an exception activating a flag which is visible to the other thread, that will terminate its execution. Both the busy waitings (the one on the empty queue and the one on the socket) have been provided with a timeout to avoid deadlocks, and, after a specific number of retries (currently set to 20) the exception is thrown.
To signal the end of the transmission, the *ReaderThread* sets a shared flag to *True*

(similarly to the error handling phase) and then returns. The *ProcessorThread* checks on this flag before starting waiting on the queue for new packets. As soon as it recognises the value of the flag as *True* it returns and the caller will be in charge of writing the read data on the output file.
The retransmission phase can only be triggered by the *ProcessorThread*, which is the only thread having access to the buffer and able to compute the missing sequences (for this reason the network interface is used in reading mode from the *ReaderThread* and in writing mode from the *ProcessorThread*).

4.3 The Windowed Protocol

The Parallel C++ version consistently reduced the calls to the retransmission phase, but even if triggered rarely, this phase would strongly impact the performances, especially when extracting high amounts of data.
In order to address this problem, an alternative version of the protocol has been developed. This version is written in C++ and it is built on top of the Parallel C++ protocol.
The main idea has been to develop a windowed version in order to manage the missing sequences, in this way it has been possible to remove the phase in which they are identified and requests are sent to the SpiNNaker system for the retransmission, at the cost of an increased network traffic.
A flowchart of this new protocol is represented if FIG. 4.3. This approach is based on a sliding window whose size is configurable (through the SpiNNaker toolchain), this window contains a certain number of consecutive sequences. As soon as all the sequences belonging to the current window are received, it slides on. The number of sequences the window slides on is configurable as well (again through the SpiNNaker toolchain).
In order to notify the SpiNNaker system that all the fragments belonging to a window have been received, the host system sends an acknowledgement to the board, this consists of an SDP packet containing a specific ACK code, plus the number of the window that is to be acknowledged (computed by dividing the number of the first sequence in the window by the number of sequences a window slides on). Through this system the board is able to verify that the acknowledgement is referred to the current window and not to old windows and to identify missing ACKs.

4.3.1 The Static Version

The first developed version of this protocol relies on a static sliding size, keeping the protocol simpler and with lower overhead board side.

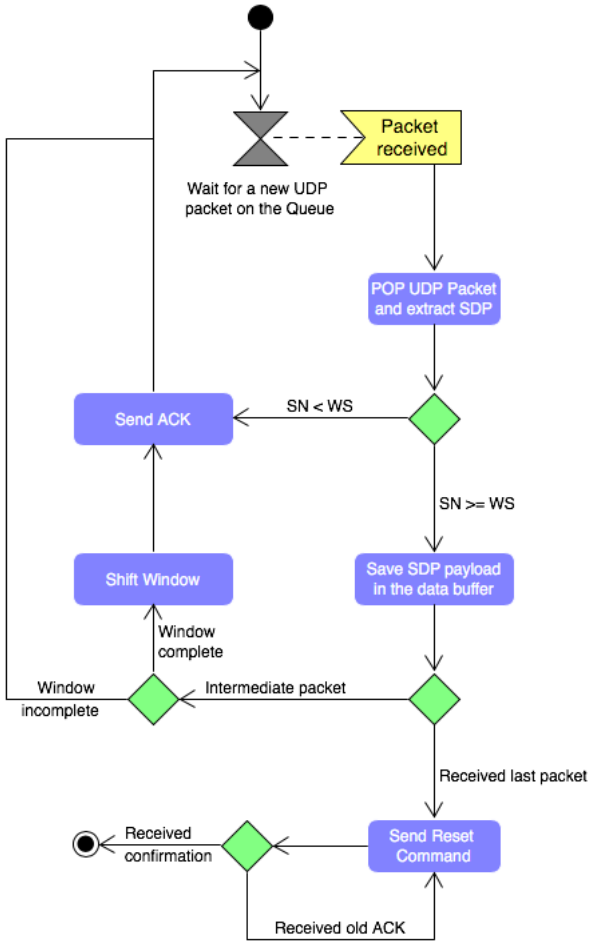


Figure 4.3: The flowchart of the Windowed protocol host side.

The code running host side is a modified version of the Parallel C++ version, but, in this case, the final retransmission phase have been eliminated. Every time a packet is received, firstly it is checked to belong to the current window, if it is an old packet it will be ignored, otherwise it will be saved and, if it was the last packet in the window and all the sequences in the window have been received, an ACK will be sent to the board and the window will shift. Otherwise the host side code will wait for another packet that will be a retransmitted one (indeed the board side code continues to send the same window until no ACKs are received). As soon as the missing packet (or the last one of the missing packets) in the window arrives, the host will send the ACK and slide the window on, avoiding to wait the window completion (the following packets will be ignored because their sequence number will be lower than the current window starting sequence, this mechanism speeds-up the retransmission for sequences close to the beginning of the window, because the board will receive

the ACK before finishing retransmitting the complete window and will move sooner to the next one). When the last window is completely received, the last ACK is sent. Then, to be sure that it has been received from the board (in order to avoid that the board starts sending continuously sequences without receiving ACKs, because the host has terminated its execution), the host will begin a cycle in which sends reset messages to the SpiNNaker system (an SDP packet containing a specific code) alternating them with the read of the packets coming from the board. As soon as the host system receives a packet containing the same reset code instead of a data packet, it terminates the execution.

Regarding the board side code, the Packet Gatherer has been modified as well in order to manage the windowed version of the DDS protocol.
This core now maintains a circular buffer containing all the packets of a single window (a circular buffer simplifies the management of the sliding of the window). Each SDP packet to be sent to the host system is also copied into the circular buffer (implemented as an array) and a counter indicating the number of sequences sent over he network without receiving ACKs is updated. As soon as this counter reaches the size of the window, the retransmission phase is triggered and the board starts resending the window. This process continues until either an ACK is received or a maximum number of retries is reached (in this case an error message is sent to the host system and the board goes in error state). The reception of an acknowledgement is managed by an high priority callback, in order to immediately stop the retransmission and maximise the performances. This function checks the correctness of the ACK (if it is related to a different window from the current one, it will be discarded) and correctly updates the indexes in the circular buffer.
The check for starting a new retransmission of the window is performed upon the reception of each FR packet from the system, this allows to reduce the latency of the retransmission.
The reset phase is managed through a callback having a lower priority than ACKs, this function checks the correctness of the code inside the message, sends an answer to the host and resets all the data structures for next transfer requests. The board system goes in *reset_waiting* phase after having sent the last sequence number. At this point it starts resending the last window either until a reset is received, or the maximum number of retries is reached.

4.3.2 The Dynamic Version

In order to increase the performances of this protocol, a dynamic version has been developed. In this approach, the number of sequences the window slides on is not constant, but stays in a predefined range (the maximum will be the value given to the SpiNNaker toolchain for the size of the shifts). This allows to partially shift the

window in case the first sequences of the window have been received at the cost of a little increase in complexity.
The idea behind this version is to send an ACK to the SpiNNaker system when an out-of-sequence SDP packet is received and at least one third of the window have been downloaded without losing sequences (one third has been chosen as value in order not to saturate the network of ACKs and, at the same time, to maximise the gain in performances brought by these modifications).
As soon as a situation of this type happens, the host system sends an acknowledgement to the board. This type of message is different from the ACKs seen so far, indeed, two fields have been added indicating whether the ACK is for a complete window or not and the last sequence received before the first missing packet.
The Packet Gatherer on the SpiNNaker system behaves in the same manner as the static version, but, in this case, if it receives an ACK indicating a partial shift it will slide the window of the correct number of sequences and accordingly adjust the indexes of the circular buffer. Otherwise the normal ACK flow described in the static version will be followed.
This new method, in principle, is able to reduce the number of retransmitted packets, starting the retransmission phase exactly from the first lost packet, instead of retransmitting all the window.

Chapter 5

The Data Upload Protocol

The second part of this work is focused on finding a way to improve the existing data upload protocol. This chapter will provide a description of the solutions adopted in order to address this problem. All the work here described has been performed at Politecnico di Torino.

5.1 Data Analysis

The main target of this part of the work is the transmission phase of the data specification generated commands to the board.
The first phase aimed at finding a way to reduce the number of messages sent to the board. For this purpose, the commands generated from the Data Specification Generator have been saved on text files separated by core. As a reference, the Cortical Microcircuit (a network representing a section of the human cerebral cortex having a base whose area is $1mm^2$ and composed of 4 Layers[22]) network has been used to generate the data specification commands.
An interesting feature that has been discovered is that, in different cores, some regions of commands are replicated and, furthermore, inside the commands streams generated for different cores it is possible to identify some regions in which the same command is replicated (even multiple times) and only data is changing.
Currently, all the commands are sent using the Point-to-Point protocol addressing the specific core. The idea in order to improve the transmission phase has been to identify common regions between the different cores' streams and to exploit the Multicast protocol to send a unique packet (containing that set of commands) directed to all the cores belonging to that area, instead of sending separated packets to all those processors.

5.2 The Multiple Sequences Alignment Phase

In order to identify the common areas between the different cores, it has been decided to use a Multiple Sequence Alignment algorithm. This type of software is used to compare different DNA sequences or proteins in biology.

5.2.1 Background on Sequence Alignment

The objective of an Alignment algorithm is, given two DNA sequences, to find their distance or, in other terms, their *Similarity*.
The *Similarity* between two sequences is defined as the *Score* of their highest-scoring alignment. The *Score* of an alignment is given by the sum of the *Substitution Scores* (given by aligned pairs of letters) and the *Gap Scores* (given by the alignment between a letter and a *Null* character). *Null* characters can be placed into the sequences and aligned with letters (but two *Null* characters cannot be aligned with each other). This operation can be either seen as an insertion of a letter into a sequence or the deletion from the other.
Typically, to a match score, is assigned a positive value while a negative score is assigned for mismatches and gaps.
The Multiple Sequence Alignment is an extension of the Pairwise Alignment (the one described so far) in order to incorporate more than two sequences at a time. This type of alignment has a high algorithmic complexity and generally belongs to the NP-complete class of problems.
One of the most common approaches to solve this problem is called *Tree-based* (or progressive) method. This strategy makes use of a tree (or hierarchical) alignment, in which, firstly, the most similar sequences are aligned in pairs and then less related groups are added to the alignment until the entire set (called *query*) has been incorporated into the solution. The first step is based on pairwise alignment to generate the initial tree that describes the sequence relatedness. The results of the *Tree-based* approach strongly depend on the choice of the most-related sequences, for this reason, often, the sequences are additionally weighted in the *query* set according to their relatedness in order to improve the accuracy[21].

5.2.2 Application of the MSA to the DSG Commands

For the purpose of this work, the Multiple Sequence Alignment has been used in order to identify commands (or regions of commands) common to the different cores in order to cluster them.
The idea was to group the cores by commands (in other terms each group contains all the cores that should receive a certain command) and then to generate Multicast packets containing the shared commands to be sent to that group instead of sending Point-to-Point packets to each core.

To address this problem the SeqAn library has been used.
SeqAn is a C++ template library used for analysing biological sequences. It provides a framework for manipulating and aligning sequences. The SeqAn library makes use of T-Coffee as alignment tool, a software which exploits the Tree-based alignment. A more detailed description of SeqAn and T-Coffee is beyond the scope of this work[24].

5.2.3 The MSA Tool

The tool has been completely written in C++. It takes as input the list of files corresponding to the streams of command relative to each core involved in the simulation and generates the list of groups and the Multicast stream of packets.
The first step performed is to extract the commands from each file. Each command is then associated with a unique integer in order to simplify the manipulation. At this point, the number of occurrences of each value is counted. The commands can either be manipulated as they are or fragmented into 32-bit words (thus separating the command words from the data words) depending on a parameter given to the tool (in case the fragmentation is performed, additional information indicating whether the single word is datum or command is attached in order to simplify the final reconstruction).
At this point, files are grouped by layer (or using a parameter provided by the user to the tool, in such a way to limit the maximum number of files grouped together) and a first Multiple Alignment is performed. This first level is performed in parallel in order to maximise the performances. Each layer (or user-defined group) is assigned to a hardware thread (if the number of cores per layer is too high, this one will be fragmented in one or more parts to achieve the highest speed possible by reducing the number of files to be aligned in a single step) and an alignment graph (structure containing the alignment results) is generated for each fragment. At this point, a second level of alignment is performed generating a graph that is the result of the MSA of all the graphs coming from the first level of alignment (that have been rearranged in order to be in the same form of the starting files).
The reconstruction phase is performed by iterating over the alignment graph and expanding each value in the final alignment recovering the corresponding values in the first-level alignment graphs. During this phase, the Multicast groups are generated depending on the alignment results. Each group will be identified by an integer and by the cores belonging to it.
The value *0* will never be used (this value is reserved for protocol purposes that will be explained later), while group *1* will always indicate the broadcast group.

In order to further reduce the latency due to the alignment phase (and to increase it precision), before the first alignment level, all the words (commands or data)

appearing only once will be converted into a *0* value (in case of sequential single-appearance words they will be collapsed into a single *0* value in order to reduce the number of sequences). This operation allows to align sequences of *0*s highly reducing the effort (and preserving the ordering).
When generating the output stream, whenever the MSA tool finds a *0* it checks the position in the original file and expands this value substituting it with the original ones that will have, for destination in the final stream, the core identified by that specific file.

5.2.4 Compression of the MC Stream

A further step can be performed in order to further reduce the number of packets to be sent to the SpiNNaker system. Indeed it is possible to notice that, inside the produced stream, there exist regions in which the same command is sent consequently multiple times to the same groups, the only difference is in the associated data.
For this reason, it has been decided to perform a compression while generating the packet stream. Whenever a command is equal to the previous one and needs to be sent to the same MC group, it is removed from the stream and only its relative data will be sent. A flag for the destinations stating that these data are relative to the last received command is also set (clearly this type of compression is possible only when the alignment is performed on the single 32-bit words, for this reason, this choice has been set as default in the tool).

The output of the MSA tool is composed of two text files, one containing all the Multicast groups and the other composed of the stream of packets.
The groups' file contains, in each line, the coordinates of a core (x, y and core number), the number of words that will be sent to that core (in order to simplify the allocation of the structures board side) and all the groups that core belongs to.
The stream file, instead, is composed of a series of lines containing 3 values: a 32-bit word (the command or data word), the group that has to receive that word (expressed as integer) and a string indicating whether the word is a datum or a command.

5.3 The Multicast Transmission Phases

Once the groups and the stream files have been generated they are passed to a program that implements the transmission. This software is composed of two parts, one running host side (written in Python) and another running board side (written in C).
The transmission protocol chosen for the on-board communications is the Multicast one. There are multiple reasons for this choice. First of all because of the nature of

the type of packets that will be sent, indeed most of them have multiple destinations, and the packets having only one destination are sent over the multicast network as well (The group list contains also groups composed of a single core). Furthermore, the Point-to-Point communication is slower because it involves the use of the Monitor processor of both the transmitter and receiver chips (when a P2P packet is generated by a core, it is copied into the SysRAM of the chip, then it is read by the Monitor processor of that chip that will send it to the Monitor core of the destination chip, eventually the packet will be copied into the SysRAM of the receiver and only at this point accessed by the destination core). The MC communication, instead, allows communicating directly between the two interested cores.
For simplicity all the transmission have been performed in Broadcast. When a core receives a packet, if it belongs to the group the packet is directed to, it will store it, otherwise, the packet will be ignored.
The developed protocol is built upon the two support libraries Spin2 and SpinACP.

5.3.1 Host Side Protocol

The host side software is composed of a main class called *Data_transfer_ACP* providing all the methods necessary to boot the system using ACP. This class contains a constructor which creates the connection with the SpiNNaker system, sets the execution context (the number of chips and cores involved in the simulation), loads the APLX file containing the board side software and starts the simulation. In addition to that, the *Data_transfer_ACP* class provides a method to send data to the board which builds ACP packets, embeds them into SDP packets and forward them to SpiNNaker and also a method to close the simulation that also clears the board.
Once the connection with the SpiNNaker system has been created and the context on the board has been defined, the software starts reading the groups' file. The first element on each line indicates the length of the stream of words that will be sent to a specific core whose coordinates are represented by the following three elements on the same line (in x, y, p format). This size is sent to the core as a single packet using ACP and through a specific *MemoryEntity*.
After this step, the list of groups (the remaining elements on the line of the file that is being currently read) is sent to the core using a different *MemoryEntity*. Currently, an ACP packet allows to send up to 255 Bytes of data, that means no more than 63 words 32 bits long. In order to maximise the performances, each group is stored on 15 bits, thus each word will contain two groups (each one occupying a 16 bits word with the first bit set to *0*). The host side software loops over the line packing the groups into the words and sending the packets as soon as they are completed. When it reaches the end of the line, the core adds a 32-bit word set to *0* at the top of the payload of the last packet (or, in case a packet is completed with the last group it creates a new packet containing only the word set to *0*) and sends

it to the core. During the alignment phase, the group *0* has been set as reserved exactly for this purpose, indeed, each core will be able to recognise the end of the group list when it receives a packet with the first two groups set to *0*.
This phase continues until all the groups have been sent to all the cores involved in the simulation. In order to reduce the load to each chip, the groups are sent interleaving the chips instead of filling the SpiNNaker system chip by chip (however, before moving to another core, it is necessary to complete the stream of the current one).
The second part consists of sending the stream of data aligned to the board. During this phase, packets are filled in groups of three 32-bit words. The first word contains two groups, while the following twos are filled with 2 words coming from the stream file generated so far.
The bit preceding each group this time will assume a particular meaning, indeed, in case that group is going to receive a command word, that bit will be set to *0*, otherwise to *1*.
The second and the third word of the triple represent the data (or command word) to be sent respectively to the first and to the second group stored inside the first word.
To signal the end of the stream, the same mechanism adopted for the groups is used and a packet having the first 32-bit word completely set to *0* is sent.

5.3.2 Board Side Protocol

The board side software acts in two phases. The first phase is composed of a first initialisation of a hash table (through the Spin2 library) that will be used to store all the groups that processor belongs to, this part is executed on all the cores (except for the Monitor processors).
Once this table has been allocated, the core waits on a specific *MemoryEntity* to receive the size of its stream of words and the list of groups. The first packet will only contain the size of the stream (the core will then allocate an array in SDRAM to store all the words), once received this, the processor will start a loop in which it will wait for the groups (another *MemoryEntity* has been used for this purpose). This loop will terminate as soon as the core receives a packet whose first word is set to *0* (this does not create conflicts since the group labelled with *0* has been reserved). The packets are read in words of 32 bits. As explained for the host side software, each word contains 2 groups (each group is stored on 15 bits). These two groups are extracted and stored into the hash table. When all the groups have been received the first phase terminates.
The second phase involves the use of the core labelled with *1* on the Ethernet chip as *ExtraMonitor* core. The role of this processor is to wait for packets on a third specific *MemoryEntity* and to forward them in Broadcast through another *MemoryEntity*.

The end of the packets stream is signaled in the same manner as the groups' stream. When a processor (*ExtraMonitor* core or not) receives a packet, it checks whether, inside the packet, are present words directed to one of its groups. In this case, it stores them inside the array in the SDRAM, otherwise the words inside the packet directed to other cores (or all the packet in case there are no groups containing the core inside the datagram) will be ignored.
A data packet can be seen as a series of groups of three 32-bit words, the first one will contain two group numbers, the following two will be (data or command) words to be sent to those groups respectively (as explained for the host side part). When a core receives a packet of this type, it will extract the two group numbers and will check inside the hash table for their presence. If a group is found, the respective word is saved. The bit preceding each group number indicates the type of the associated word (*1* will stand for data and *0* for command) for the reconstruction of the uncompressed stream.

5.4 The New Configuration Protocol

In order to improve the internal communications and to speed-up the data configuration phase, a new protocol has been developed. This new approach relies on optimised routing rules and on the Multicast transmission protocol.
The main idea behind it is to exploit all the cores belonging to the Ethernet chip when sending data for configuring the simulations and to use the Multicast protocol to convey both Point-to-Point and Broadcast packets.
The chance to send Point-to-Point packets over the Multicast network can theoretically bring great improvements, because, through this approach it is possible to implement a real core-to-core communication which does not imply the intermediation of the Monitor processors.

5.4.1 The new Routing Rules

All the Multicast routing entries have been rewritten in order to allow the transmission of the SYNC, Broadcast and Point-to-Point packets over the Multicast network. The transmission on the board can be either seen as a packet coming from a source having multiple destinations (Broadcast/Multicast communication) or as a packet directed to a specific destination coming from another source (Point-to-Point communication). The source in the first case and the destination in the second one will be called *Pivot*.
The new defined routing rules are built upon the assumption that a SpiNNaker board is divided in 8 regions relative to a specific *Pivot* that will be the center of the communication. The placement of these regions is shown in FIG. 5.1. On the right side of the picture are represented the labels for the 6 links of a SpiNNaker

chip. The eastern one is labelled with *0*, starting from it the others are labelled with anti-clockwise increasing values. This rule will be used from this point on to identify the output links of a SpiNNaker chip. Each Multicast routing rule is composed of

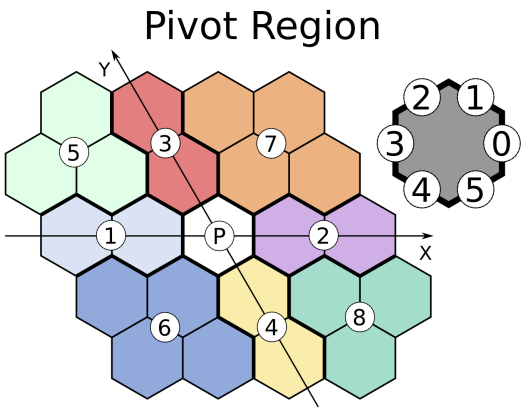
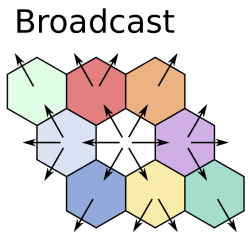


Figure 5.1: The Pivot Regions.

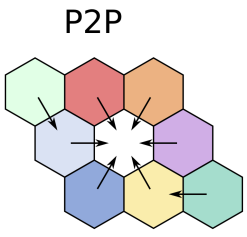
a key, a mask and an associated rule. When a packet needs to be sent over the Multicast network, its key is filtered with the mask of a routing entry and then, if it matches the key, the packet is sent following the rule. These steps are performed on all the routing entries until the correct one is found. If there are no matching keys, the default routing will be performed.
The new Broadcast rules are shown in FIG. 5.2a, the Pivot chip (the one which starts the communication in this case) is the one in the middle, while the arrows indicate, for each chip, the links on which the packet will be forwarded if that specific chip belongs to that area. The table shown in the picture provides a tabular representation of these rules. On the rows there are all the regions, while on the columns are placed the links. The presence of a x in a specific box indicates that if the chip is in the area on the row, it will have to forward the packet on the link indexed by the column.
The Point-to-Point rules are presented in FIG. 5.2b. Again the Pivot is in the middle. In this case, it is the destination of the communication. The table provided in this picture can be read in the same way as the one for the Broadcast protocol. These rules are generated statically and loaded while configuring the board.

This new data configuration approach is built on top of the Spin2 library, that has been updated in order to maximise the performances.
The mask and the key for each routing entry are built to be used with Spin2 Multicast packets (whose structure has for this new protocol). For this reason, in order to work, this approach needs to generate MC packets using Spin2.



	0	1	2	3	4	5
P	x	x	x	x	x	x
1			x	x	x	
2	x	x				x
3		x	x			
4					x	x
5			x			
6					x	
7		x				
8						x

(a) The New Broadcast Rules.



	0	1	2	3	4	5
P						
1	x					
2				x		
3						x
4			x			
5						x
6		x				
7					x	
8				x		

(b) The New Point-to-point Rules.

Figure 5.2: Broadcast and Point-to-Point routing rules

5.4.2 The New Spin2 Library

For this purpose, the Spin2 library has been rewritten and expanded. The structure of the Spin2 MC packets has been modified for the new rules and the information contained in the key field has been compressed. Thanks to this modification, now, the Broadcast packets can convey 48 bytes of payload instead of 32 as defined in the previous version of the Spin2 library (for this type of transmission the key field has been compressed to 16 bytes).
The new structure of the MC packets is presented in FIG. 5.3. The first two elements represent a Point-to-Point packet and a Broadcast one respectively. The Highest 16 bits of the key are really similar. The first bit represents the type of packet (it is set to *0* for P2P/BC packets), the second one indicates whether the packet is a P2P or BC, then there are two 3-bit fields representing respectively the x and y coordinates of the involved chip (the destination in case of the P2P packet and the source for the Broadcast one), which are followed by the core expressed on 4 bits. The last four bits are used for sequencing purposes: the first one indicates if this packet is the last of the sequence or not, while the following three assume different meanings depending on the value of the first. If the packet is the last in the sequence, these three bits will indicate how many bytes of payload this packet is carrying, otherwise, they will be used as a counter for the packet number.

If the packet is a P2P one, the following 16 bits of the key will be used for indicating the source of this packet (using the same approach adopted for the destination). In case of a BC packet the lower 16 bits of the key will be used as payload extending it to 48 bits.
The SYNC packet has a different structure, the first bit is set to *1* to indicate that the packet is a SYNC one, the second is forced to *0* (the value of this bit to *1* is reserved for future use). The SYNC level is provided through the last four bits of the second 16-bit word of the key using the One-Hot Encoding (the highest bit represents the SYNC Free value, the following one a Level 3 SYNC, followed by a Level2 SYNC bit and finally a Level1 bit). The bits in the middle are set as *Don't Care*
The Multicast routers contain all the routing rules (ordered as SYNC, Broadcast

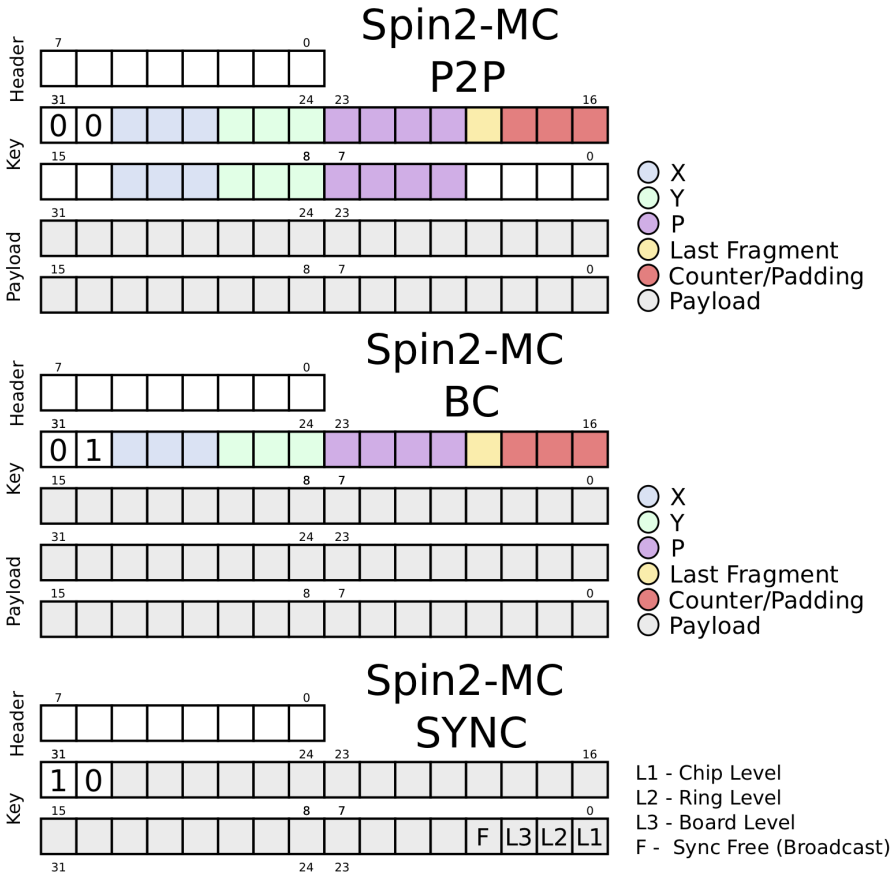


Figure 5.3: The New Spin2 MC packets.

and finally Point-to-Point) in order to manage these three types of packets.
The Spin2 library has been extended in order to manage the fragmentation and reconstruction of higher level packets in P2P/BC fragments, in order to hide these details to the applications.

Besides the implementation of the SYNC protocol (which has not changed with respect to the previous version), now the Spin2 library exposes functions in order to send and receive higher level packets.
The sending phase is performed by recognising the specific channel (Broadcast or Point-to-Point) and by executing the fragmentation of the higher level packet in Spin2 MC packets. This fragmentation will be different depending on the type of packet that needs to be sent, indeed each BC over MC packet can contain up to 6 Bytes of payload, while a P2P over MC has its limit at 4. As soon as they are created, the MC packets are sent over the network.
The receiving phase is more complex, it involves to maintain a number of buffers (currently 8) in SDRAM for each core. These buffers are used to reconstruct the packets starting from the fragment received through the Multicast network. Each buffer is 272 Bytes long (the same size of the payload of an SDP packet) and will be associated with the sender of a specific packet. This means that if all the buffers are reserved, packets coming from an additional cores will be discarded until at least one buffer is cleared.The association sender-buffer is managed through a circular buffer. The arrival of a new packet generates an event which triggers a specific callback that looks for the presence of the source core in the list of reconstruction buffers and, in case this is found, it appends the payload to the buffer. Otherwise, if there are available buffers, one of them will be allocated to that source core. If no buffers are free the packet will be discarded.
Once a packet has completely been reconstructed, the Spin2 library informs the upper level application of the completion of the transmission through a specific callback registered by the application itself (that will receive the packet, its size and the type of communication). At this point, when the application has finished managing the packet, the buffer used to store it is freed and it can be allocated for another transmission.

5.4.3 The New SpinACP Library

The SpinACP library has been modified as well in order to exploit the new protocol. The structure of an ACP packet has changed and the new version is shown in FIG. 5.4.
The first two Bytes are now used to store the type of command (in other terms the user-defined operation or the type of action to be performed on the selected *MemoryEntity*), the following 7 bits indicate the packet number (not used at the moment), then the size of the payload is represented on 9 bits. The last two fields of the ACP header are both 2 Bytes long and the first one is the offset (currently not used), while the last one represents the ID of the *MemoryEntity* which is being addressed. All the remaining Bytes (up to 256) are used for the payload.
The SpinACP Library is structured in two parts: the first one runs board side (it

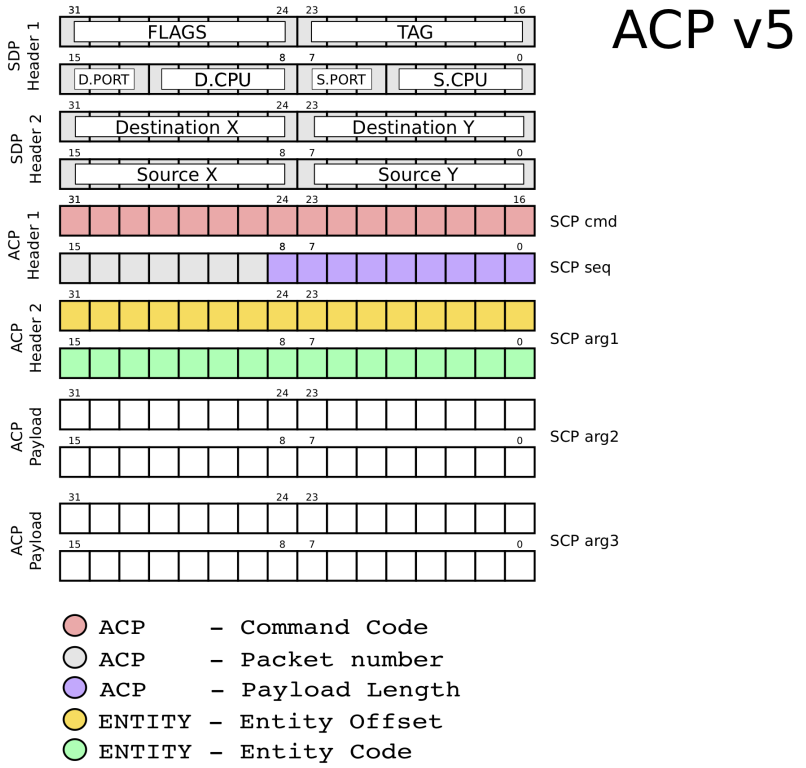


Figure 5.4: The New ACP packets.

is written in C) and its main modules are represented in FIG. 5.5, while the second one runs host side (and it is written in Python).

The entry point for the board side part of the library is the *acp* module. This one is in charge of initialising all the other modules and of exposing the functions necessary to act atomically on the *MemoryEntities*. The two modules *acp_mc* and *acp_sdp* define respectively the callbacks triggered when a MC or a SDP packet are received (the MC and SDP transmissions are implemented by the corresponding modules defined in Spin2). Both these callbacks, upon the reception of an ACP packet and its identification, call the *acp_exec* module. This one is in charge of recognising the type of command received by checking the *cmd* field inside the received ACP packet and, after this step, of calling the specific function to execute it, defined inside *acp_me*.

The *acp_me* module is the core of this software, it implements all the operations on the *MemoryEntities* called by *acp_exec* and *acp*.

These operations can be:

- *Init*: allocates the hash table containing the *MemoryEntities* in memory

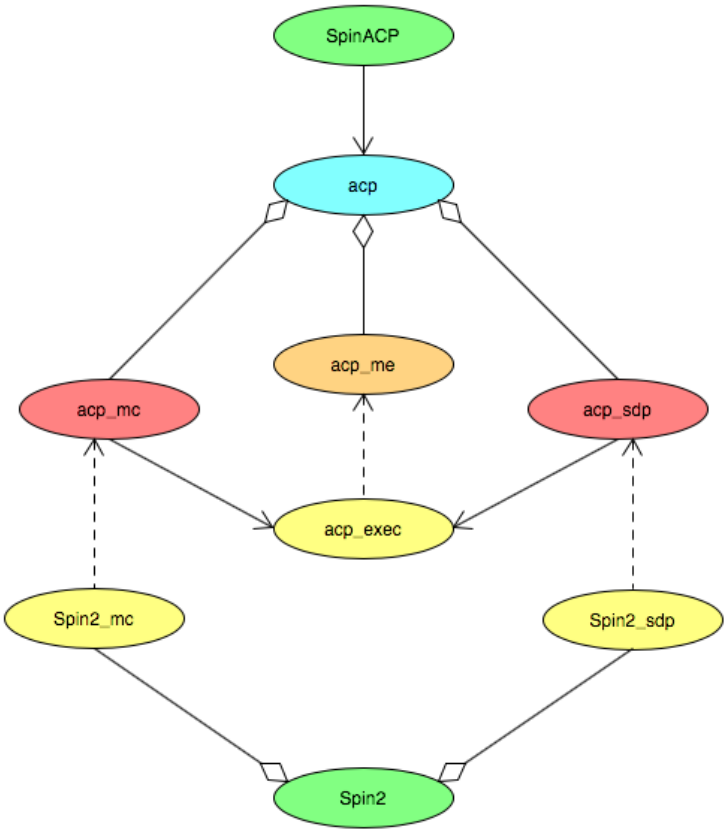


Figure 5.5: The SpinACP Library Structure Board Side.

- *Create*: creates a new *MemoryEntity* with a specific ID and gives the possibility to define callbacks triggered when this *MemoryEntity* is being read or written

- *Read*: reads a *MemoryEntity* given an ID and saves it in a byte array, it can be implemented as blocking (the read is performed only after the *MemoryEntity* has been updated)

- *Test*: checks the presence of a specific *MemoryEntity* given its ID inside the hash table

- *Update*: writes on a *MemoryEntity* given the ID and signals the update for any waiting read

- *Delete*: removes a specific *MemoryEntity* from the hash table

- *Clear*: deallocates the hash table

The host side software is a collection of Python modules used to interface with the board to implement the ACP communication.
The core of the host side implementation of ACP is the *ACPRuntime* class, this one implements the communication with the board, allows to start and stop the application and to send SDP packets to the SpiNNaker system. This class is also in charge of loading on the board the new routing keys defined for the Multicast protocol implemented by Spin2.
The execution context (the chips currently used) is provided by the *ACPContext* class that keeps track of the cores involved in the simulation and of their state.
The update of *MemoryEntities* remotely is performed through the *RemoteMemoryEntity* class which allows to implement the *Update* command on a specific *MemoryEntity* allocated on board.
The ACP packets are built through the *ACPPacket* class that creates a structure similar to the one defined in FIG. 5.4 and returns a bytes array containing the packet. This class is extended by a specialised module called *ACPPacketME* which creates an ACP packet specifically for managing *MemoryEntities*
The purpose of all these modules is to provide a middle-level interface for sending data and commands on SpiNNaker. Higher level applications are built on top of this library.

5.4.4 The Board Configuration

The new board configuration protocol exploits these two described libraries. The idea behind it is to send the DataSpecification commands using the Point-to-Point protocol over the Multicast network implemented by Spin2. This allows avoiding the complexity of the MSA phase and to incur in the time penalty caused by the intermediation of the Monitor processors that happens with the standard P2P protocol.
The commands are sent to the board as they are generated by the DSG, no manipulation is performed on them. The protocol is composed of two parts, a host side module and a board side application. Each SpiNN5 board allocated for the simulation is configured separately.
The stream is sent to the board using SDP packets (having ACP datagrams as payload) and forwarded to the specific cores through the Point-to-Point protocol. In order to maximise the performances, the Ethernet chip has been completely dedicated to the forwarding phase. Each core belonging to that chip has the role of filling a certain number of cores of the board (and will be called configurator from this point on).

The code running host side writes on a specific *MemoryEntity* allocated on each configurator defining all its destination cores. This is performed by sending, through

ACP, a list of coordinates each one associated with a specific *MemoryEntity*.
After this step, the stream of commands for each core is sent to the board through ACP packets having, as destination, the specific *MemoryEntity* defined on the configurator in charge of filling the core that needs to receive the stream.
By using all the Ethernet chip as board configurator, it is possible to generate multiple flows of data inside the board, speeding-up the forwarding phase. In order to avoid overloads, the streams of commands are sent interleaving the configurators. This means that a fragmentation operation is performed and each configurator receives one fragment of a stream at a time to be forwarded. Once all the configurators have received one fragment, the host side software starts sending the next chunk of the streams from the first configurator. This cycle continues until all the data are sent to the SpiNNaker system.

The board side software has two different behaviours depending on the core it is running on.
In case the processor is a configurator, an SDP packet is waited on a specific *MemoryEntity* through a blocking read operation. This packet contains the list of cores that need to be filled by this configurator associated with their specific *MemoryEntities*. This list is saved on a structure in memory and the necessary *MemoryEntities* are allocated. After this phase, the configurator goes in idle state, waiting for ACP packets to be forwarded to the core associated with the *MemoryEntity* on which they have been received.
In case the core is an Application processor, it registers a callback on the *MemoryEntity* with ID *0* (the one on which the configurators write) and waits for the data. Once an ACP packet is received, all the payload containing the DataSpecification commands is saved and the complete stream is reconstructed in order to be executed.

Chapter 6

Results

In this chapter will be presented the results of the executions of the developed solutions alongside with some benchmarks. The first section will provide comments on the data extraction protocols, while the second one will be focussed on the data upload.

6.1 Data Extraction Results

6.1.1 Comparisons Between the C++ and the Python Versions

The two pictures FIG. 6.1 and FIG. 6.2 provide benchmarks for the execution of the DDS protocol using the Python version and the C++ one respectively.
These tests have been executed on a single board, each execution demanded a different amount of data (from 1 MB up to 100 MBs). The downloaded data was an increasing sequence of numbers generated by the board. For both the tests the speed of the transfer (in Mb/s), the elapsed time (in s) and the CPU usage (in percentage) of the host machine (that was a dual-core machine) are provided.
Regarding the speed, it can be noticed that the average is around 34 Mb/s (slightly lower than the theoretical one because these tests were performed on a WiFi network and not via cable). The Python version registered some higher performances drops with respect to the C++ version, but the general behaviour is the same.
What is worth to be noticed is that the CPU usage is around a 10% lower for the C++ version with respect to the Python one. This means that, how expected, the C++ code had a lower impact on the performances, giving better chances to work in parallel.

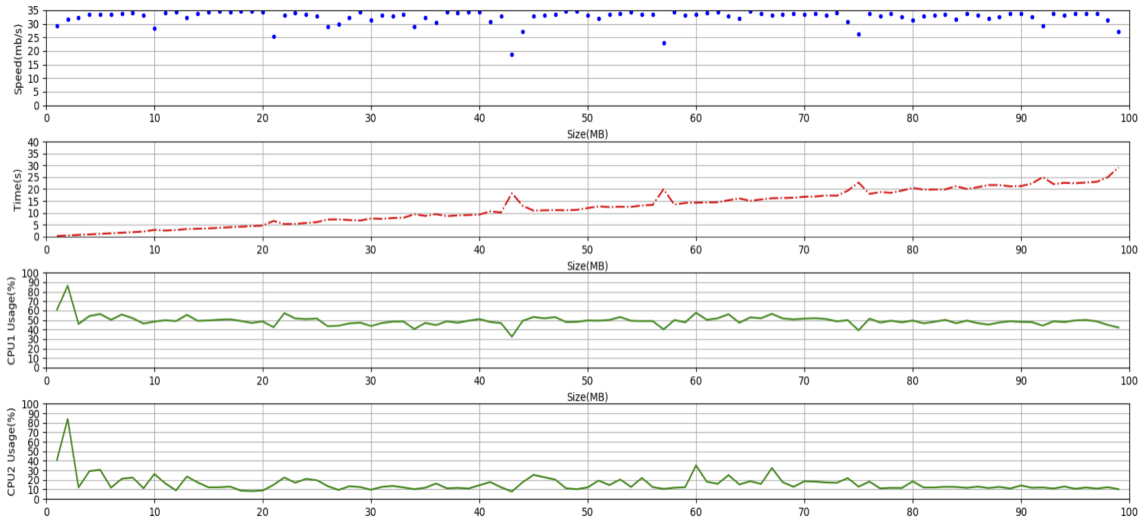


Figure 6.1: Python data extraction Benchmarks.

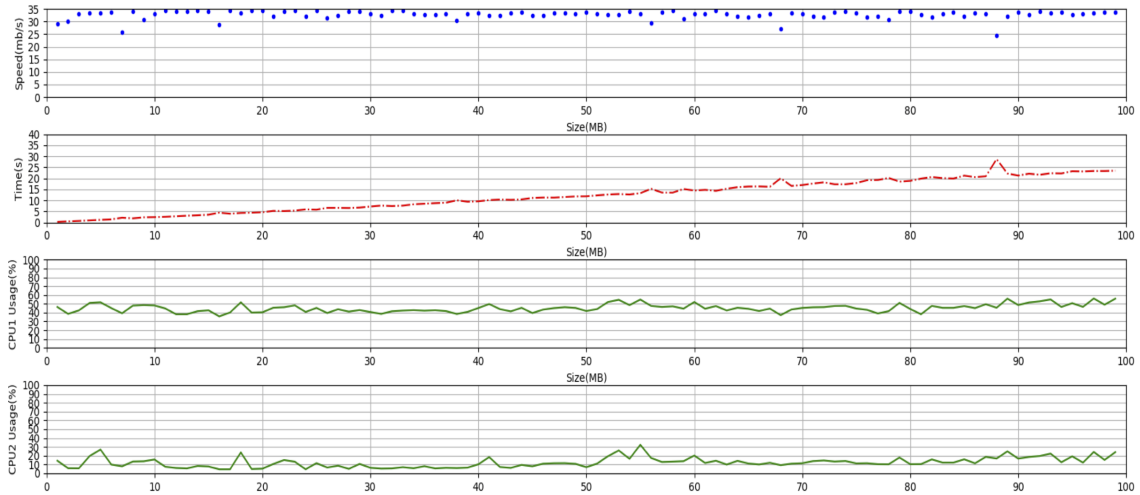


Figure 6.2: C++ data extraction Benchmarks.

6.1.2 Extraction of Parallels Flows of Data

Having demonstrated that the C++ performances are not lower than the Python one, it has been decided to test the new version with multiple concurrent flows of data. The Python code did not perform well in this case, giving the same results of the single flow version.
For this purpose, the same test code has been used, but, this time, the simulation required multiple boards. In order to verify the real performances of this code, these tests have been executed on a remote server that was connected via cable to the SpiNNaker machine and with no other jobs running on it.

The results can be seen in FIG. 6.3.
This chart presents error bars relative to the extraction of 30 MB from the board

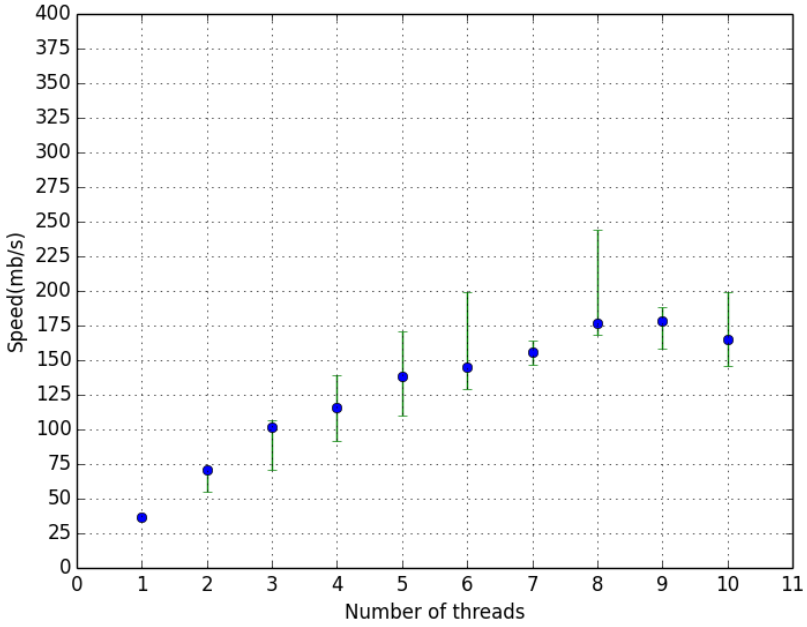


Figure 6.3: C++ Multiple flows Data Extraction Benchmark.

(given the similarity of the results, all the benchmarks from 1 to 29 and from 31 to 100 MBs have been omitted, and the 30 MBs charts have been chosen to represent all the stress tests executions). The X axis enumerates the number of threads used for that specific simulation (the number of threads indicates the number of parallel flows of data coming from the SpiNNaker board and so the number of boards allocated in that context). The Y axis indicates the speed (in Mb/s). Each bar is the result of 10 executions with the same number of threads, the blue dot indicates the average and the two bars provide the maximum and the minimum values gotten.
It can be seen that performances grow linearly up to 5 threads. Over this number, the average starts being in the lower part of the bar and, for executions with more than 8 threads, performances start decreasing. This is due to the fact that the code starts entering in the retransmission phase more frequently, because, as explained, the time required to save the data is too high and packets are lost.

6.1.3 Results for the Parallel Version

The Parallel C++ Version has been developed exactly to address this problem. The same stress test has been performed for this version as well. The results are presented in FIG. 6.4. This chart is, again, related to a data download of 30 MBs. It can be seen that performances are much better than the previous case. Here it is

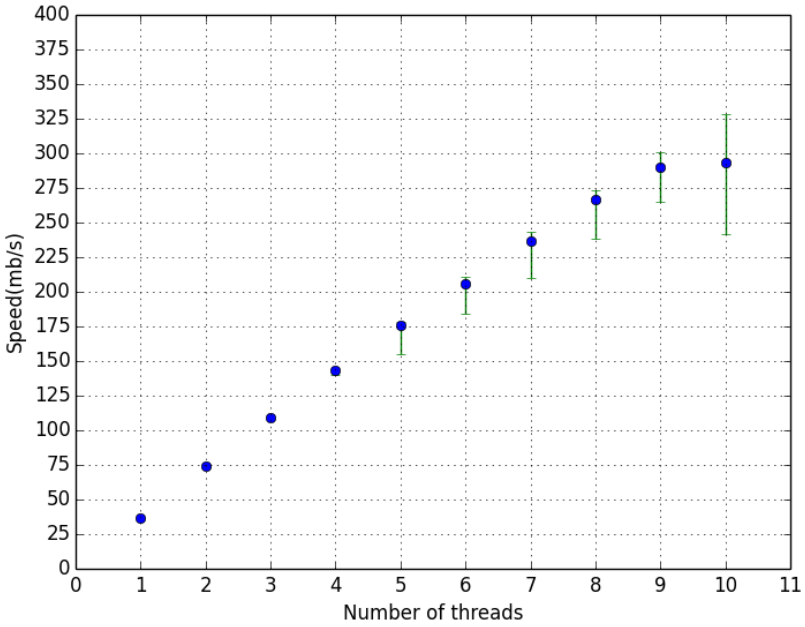


Figure 6.4: Parallel C++ Protocol Execution Results.

possible to see a linear growth in speed up to 9 parallel flows of data. Performances only start degrading with 10 threads. Another interesting result is that there is no drop in download speed up to 4 parallel threads (this means that the number of lost packets is really low). Furthermore, the average remains close to the top for all the executions, that indicates that for almost all the executions the retransmission phase was triggered only little times or never.
The maximum registered speed, in this case, is 329 Mb/s, which is much higher with respect to the previous case (242 Mb/s).
This version brought great improvements, both compared with the first C++ protocol and even more if compared with the Python one (which was able to reach no more than 38 Mb/s as total transfer speed).

6.1.4 Performances of the Windowed Protocol

The two pictures FIG. 6.5 and FIG. 6.6 show the results for the execution of the same test case, but using respectively the static and the dynamic version of the windowed protocol. The idea with this protocol was to reduce the overhead of the calls to the retransmission phase, obtaining a linear growth in the average speed regardless of the number of parallel flows (avoiding the decrease that happened in the first C++ version with more than 5 threads), paying with a small reduction in performances.
The results was not as good as expected and performances are much lower than the

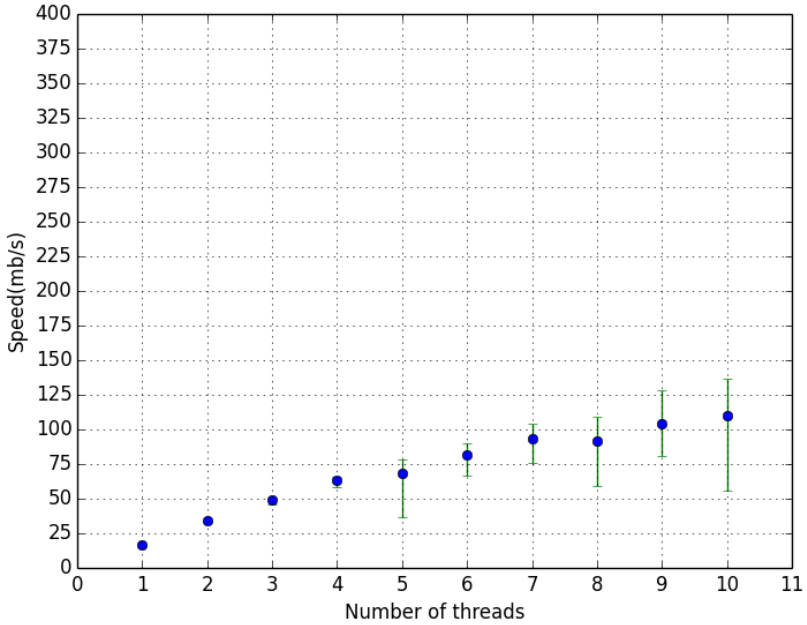


Figure 6.5: Windowed Protocol Static Verison Execution Results.

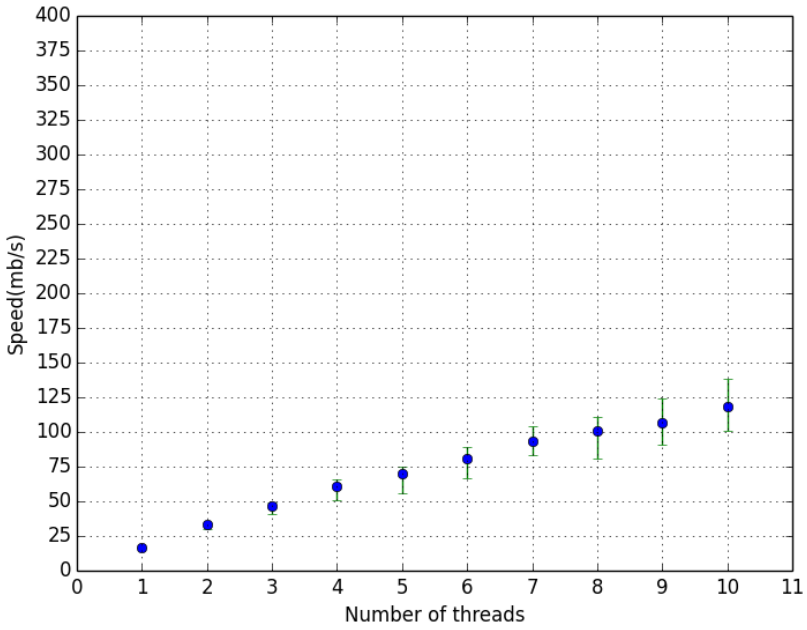


Figure 6.6: Windowed Protocol Dynamic Execution Results.

Parallel C++ case (here the maximum speed is at 139 Mb/s for both the versions). The behaviour is linear, but the protocol is not fast enough to become an alternative to the previous one. At least at the current situation.

The dynamic version, as foreseen, slightly outperforms the static one when the number of threads starts increasing, but the general trend is the same.
These results are due to the current implementation of SC&MP (the software running on the Monitor processors), indeed, when dealing with SDP packets to be sent over the network, the Monitor Processor on the Ethernet chip maintains a buffer whose size is limited (currently 16 packets). When this buffer becomes full, the system starts dropping packets, this causes performances to be so bad. Indeed, in the windowed versions, the SDP traffic is much higher than the previous versions. For this reason, the Monitor Processor on the Ethernet chip is not able to manage such a high quantity of packets and will implement an unreliable communication that will require many retransmissions in order to transfer all the data, causing these bad performances.

To prove this point, an experiment has been performed. The size of the buffer for the SDP packets managed by SC&MP has been increased (from 16 to 32, higher values could have been saturated the memory) and the same test on the Parallel C++ version has been performed (this version has been chosen because it was the best in terms of performances). The results are shown in FIG. 6.7. It is interesting

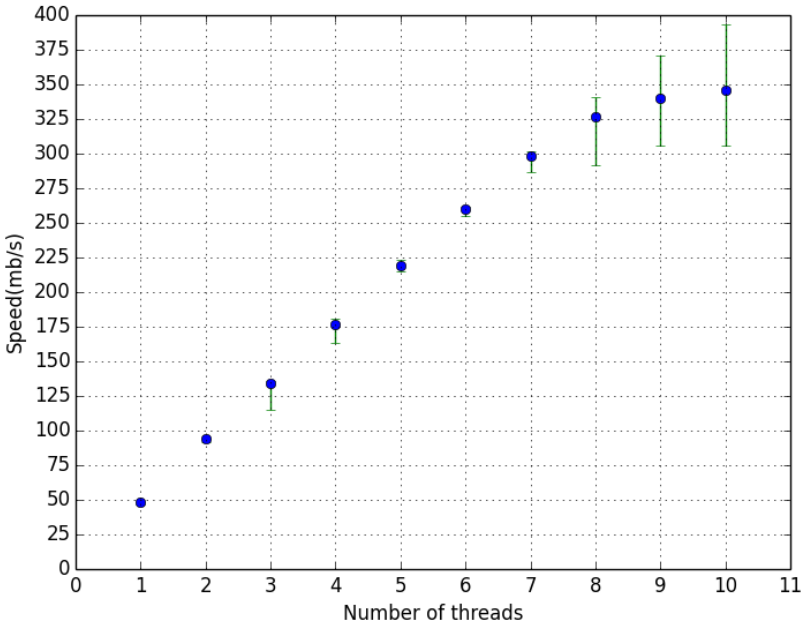


Figure 6.7: Parallel C++ Results with SC&MP Changes.

to notice that the performances with a single thread increased (reaching 48 Mb/s) and the general trend was very good up to 9 threads with a new maximum speed of 385 Mb/s. This means that the assumptions made were true and that, currently, the bottleneck is given by the implementations of SC&MP.

The table in FIG. 6.8 shows the average of total missing packets for each version of the protocol during the test case presented.
As expected the Parallel C++ version generated the lowest number of missing packets (the version with SC&MP changes was even better). The worst performances were given by the windowed versions, especially the dynamic one that resulted in the highest number of missing packets (again the reason is the high SDP traffic generated) even though performances equals the static version (the reason is that retransmission for the static one asks for a lower number of packets).

C++ not parallel	Parallel C++	Window static	Window dynamic	Parallel with SC&MP changes
1464	572	1663	1863	271

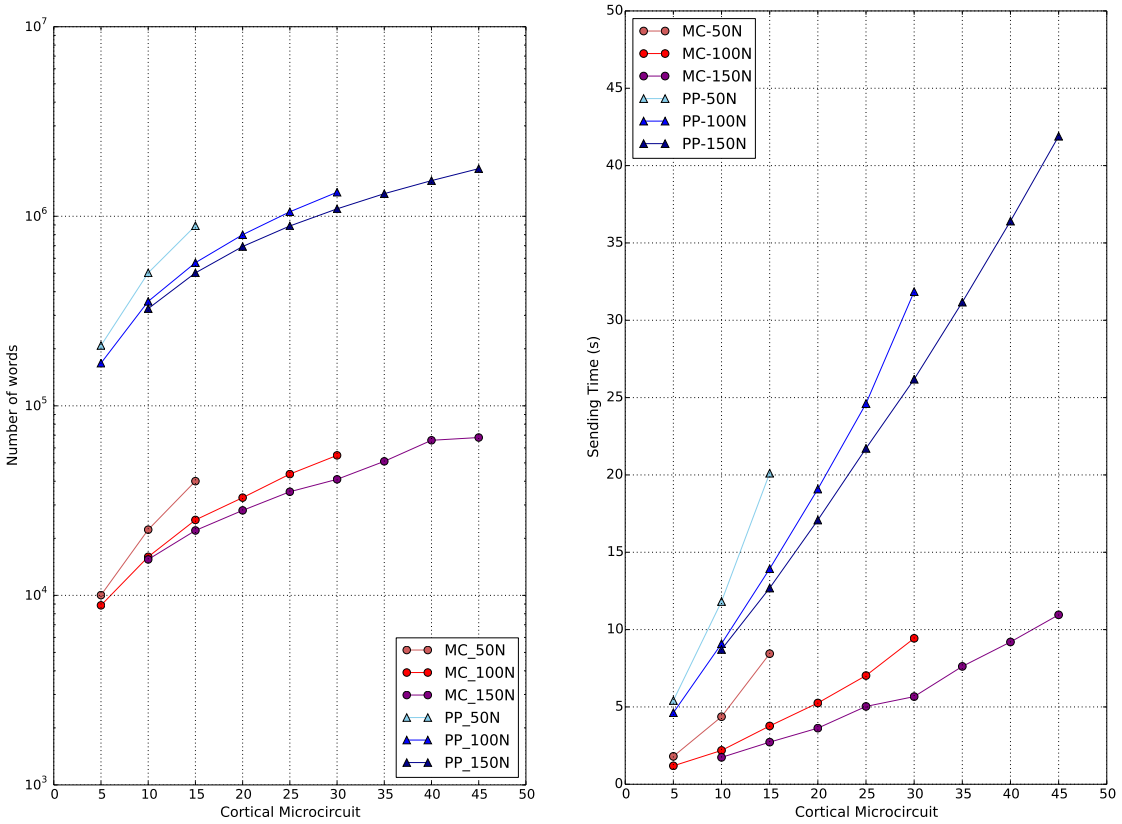
Figure 6.8: Comparison between missing packets for all the versions of the protocol.

6.2 Data Upload Results

The benchmarks for the two phases of the first version of the new Data Upload protocol (the MSA and the MC sending phase) are shown in FIG. 6.9.
The two images show the results of multiple executions of the Cortical Microcircuit network using different scaling factors. The distributions of neurons per core tested are 50, 100 and 150 respectively. Different scalings for the network have been tested (from 5% up to 45%). All the simulations performed fit inside a SpiNN-5 board.
In the two pictures are presented both the situation in which the MC Data Upload protocol is used (the lines with squares and labelled with MC in the legend) and the one in which the normal P2P data flow has been chosen (the lines with triangles and labelled with PP in the legend).

6.2.1 MSA results

The plot in FIG. 6.9a presents the total number of words of the stream to be sent to the SpiNNaker system both with and without the MSA step. The y-axis is in logarithmic scale.
It is possible to notice that the number of generated words differs of more than one order of magnitude between the two cases for all the three scaling.
By increasing the simulated percentage (x-axis), it is possible to see that the gap between these two approaches grows. This is visible especially for the case with 150 neurons per core (dark blue and purple lines in the charts). In this case, indeed, the number of packets generated without the MSA for a simulation at 10% is 13 times greater than the one generated if MSA is used. By scaling this network up to



(a) Comparison between the streams with and without the MSA + compression steps

(b) Comparison between sending times using the P2P protocol and the MC one

Figure 6.9: Data Upload phase benchmarks

the 45%(maximum percentage fitting inside a SpiNN-5), the stream without MSA becomes 30 times greater than the one after the Alignment operation. Similar values are generated for the other values of neurons per core (from 20 to 23 times is the gain with 50 neurons and from 17 to 30 times with 100).
With reference to FIG. 6.9a, it is possible to say that the MSA step brought, as foreseen, excellent results in this type of data compression.

6.2.2 Data transmission phase results

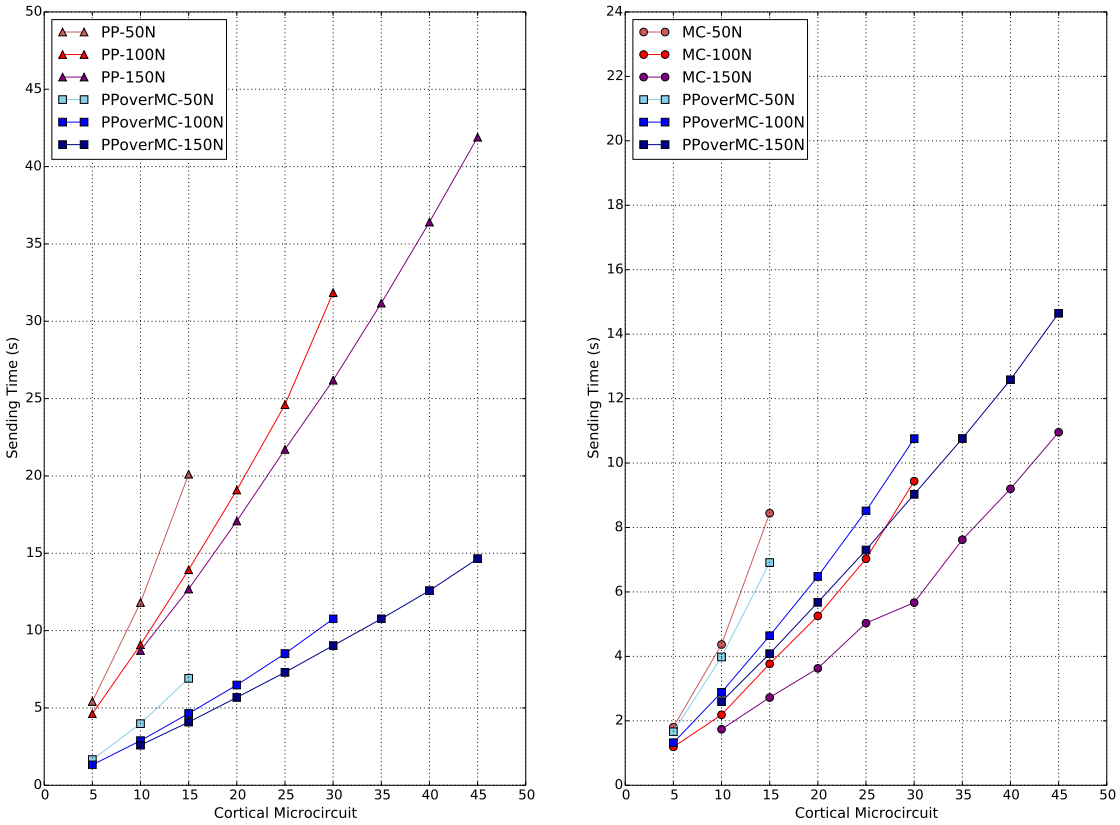
The results of the data upload phase are shown in FIG. 6.9b. Both the axes are in linear scale. In this case, the y-axis represents the sending times (measured in seconds). The color scheme is identical to the one used in FIG. 6.9a and the simulations performed are the same.

The Point-to-Point sending times grow exponentially with the percentage of the Cortical Microcircuit simulated. The same behaviour can be noticed for all the three different distributions of neurons per core. This is due to the fact that, increasing the scaling, the simulated network grows and, for this reason, the stream generated for each core becomes bigger.
On the other hand, by using the Multicast protocol for sending compressed data, only a linear increase in sending times is obtained. The most significant gain can be seen for the simulation at 45% using 150 neurons per core, in which, the time required for sending all the stream in Point-to-Point is 61 seconds while, using the Multicast approach, all the board is filled in only 12 seconds.

However, there is a drawback in this Data Upload protocol: the alignment times are not negligible and they negatively impact the performances of this approach highly increasing the total transmission time.
The obtained results are nevertheless significant because they show that improvements are possible if the MSA tool is correctly addressed using a full custom version instead of modifying an existing algorithm.

6.2.3 Performances of the New Data Configuration Protocol

In order to prove the efficiency of the new data configuration protocol, the same tests executed for the Multicast approach have been run for this version as well. The results are shown in FIG. 6.10.
FIG. 6.10a represents a comparison between the configuration phase performed using the new P2P over MC protocol and the standard Point-to-Point version, while FIG. 6.10b shows its performances with respect to the MC transmission of the aligned commands.
In the first case, the gain is evident. It is worth noticing that, besides having lower values, the curve indicating the upload times of the new protocol has a linear behaviour instead of the exponential one that can be seen for the standard P2P approach. These improvements are due to the fact that the transmission is now implemented on the Multicast network, although the packets have a single destination. This allows not to use the Monitor processors as intermediaries and to communicate directly with the destinations.
Another feature that increases the speed of the data transfer is the use of the whole Ethernet chip as configurator. This approach grants that the load is distributed over a whole chip instead on a single core, giving the chance to send packets to other configurators while one is receiving and manipulating.
During the performed tests, as can be seen in FIG. 6.10a, this new protocol allowed to send data up to 3 times faster than the standard P2P version. Again, by increasing the scaling of the network, the gain grows as well. To prove this point, with



(a) Comparison between sending times using the original P2P protocol and the new one.

(b) Comparison between sending times using the MC protocol and the new P2P one.

Figure 6.10: Performances of the new Data configuration protocol against the old versions.

a 5% Cortical Microcircuit the sending time is 2.5 times lower than the standard P2P, while when using a 45% scaling factor it becomes 3 times lower. This can be generalised for all the three distributions of neurons simulated as shown by the graph. This new protocol is moreover able to reach performances really close to the Multicast one, as shown in FIG. 6.10b. In this chart, the Multicast sending times are presented using the lines with triangles, while the new Point-to-Point ones are associated with the lines with boxes.
The case in which the Cortical Microcircuit simulated had 50 neurons per core is slightly different from the other two. This is because the length of the streams of data are quite small and the groups sending phase has a higher impact on the total time of the transmission.
By increasing the number of neurons per core and the scaling it can be noticed that both the Multicast and the new Point-to-Point transmission times grow linearly (and in the worst case scenario the new P2P is only 4 seconds slower).

The reasons why the new protocol is so close to the Multicast version in terms of performances are that this new version does not need the group definition phase which is computationally expensive and, besides, packets are sent by interleaving the cores on the Ethernet chip.
The assigning phase between configurators and Application processors in the new P2P is performed through a single ACP packet, while for sending the groups in the MC approach, it is necessary to fragment the information in several datagrams and to repeat this operation for all the cores, instead of setting only the Ethernet chip. Furthermore, the MC version needs to send SDP packets containing the stream to the Monitor processor of the Ethernet chip that will forward the ACP datagrams to the destination groups. In order not to lose packets, it is necessary to insert a *Throttling time* between two SDPs in such a way this Monitor core has the time to receive and forward the data. A similar delay is present for the new P2P approach as well, but it is lower because, while a configurator is receiving and forwarding, the host software is sending packets to another one.
What makes this results so interesting is that the total time to upload the board for the MC version is given by the sum of the Alignment time and the data upload phase, while for the new P2P version no additional steps are required, resulting in a final time lower than the Multicast version.

Chapter 7

Conclusions

The purpose of this work has been to improve the communication protocols for the SpiNNaker system. The communication speed is currently one of the most critical aspects that affect the performances for real-time simulations on the SpiNNaker neuromorphic platform.
During this work, different attempts have been performed in order to increase the data extraction speed, the most promising one is the Direct Data Stream in its parallel C++ version. However the current implementation of SC&MP, as evidenced by the tests, prevents it to reach the highest possible efficiency.
In order to achieve better performances, an optimisation of the SC&MP software is necessary, but this step would require to re-organise the Monitor Processors' work. For these reasons it is a candidate for future works.
Another step that can be made in order to improve the performances of this protocol is to try to completely exploit the Ethernet Chip (similarly to what is done during the data configuration phase). With the help of the 16 Application processors of the Ethernet chip, it could be possible to generate multiple flows of data coming from a single board. This modification would require first to test the performances of the Monitor core on the Ethernet chip in order evaluate its real capacity to generate packets to be sent over the Ethernet interface).
Other attempts are currently being performed at The University of Manchester in order to internally improve the DDS protocol. The idea is to use the key field of the intermediate FR packets as a payload extension (in order to double the data carried by each packet), in this way, in principle, it could be possible to double the speed of the data extraction.

Regarding the data uploading phase, the use of the Multicast network brought great improvements both if used to convey data with multiple destinations and for Point-to-Point packets. For the first case, currently, the real bottleneck has moved to the Multiple Sequence Alignment step, which remains too slow. The speed of

the alignment phase mostly depends on the number of different files aligned together, a first improvement could be to generalise the idea of aligning the streams on different levels increasing their depth in order to elaborate no more than two files at a time. However this approach would greatly increase the complexity of the reconstruction phase, for this reason, some further analysis would be necessary in order to find the correct balance. The SeqAn algorithm is optimised for biological sequences alignment in which the mismatch can be accepted and does not have to be totally excluded as in the case of this work. Furthermore, the type of data which is generally aligned is different with respect to the DataSpecification commands.
Another strategy could be to rewrite from scratch a tool to align the files.
A software written exactly for this purpose could give better performances. By combining these optimisations with a better exploitation of the whole Ethernet chip for the transmission, it could be possible to achieve much better improvements.
The new protocol based on P2P over MC is currently the best solution and could lead to extremely reduce the uploading times since it allows to configure multiple SpiNN5 concurrently. It could be possible to extend this protocol in order to configure large systems by acting in parallel on the different boards and achieving higher performances. This could be achieved by using a parallel host side software similar to the one developed for the data extraction phase.

This work demonstrated that improvements are possible not only for the external communications, but also for the internal protocols, providing new ways for achieving better results.
Given the similarities between the two systems, the ideas here developed could be theoretically used to improve the data exchanging protocols for the Intel Loihi platform as well.

Acknowledgements

First of all I would like to thank my supervisor Professor Andrea Acquaviva for giving me the chance to work on such an interesting project and for giving me the possibility to join the team at The University of Manchester. I would like also to thank Professor Steve Furber and Professor John V. Woods for inviting me in Manchester.
A special thank goes to the Human Brain Project Group in Manchester for welcoming me in their team and for their patience, especially I would like to thank Andrew Rowley, Alan Stokes, Andrew Gait, Christian Brenninkmajer, Oliver Rhodes and Donal Fellows.
I would like to thank Gianvito Urgese and Francesco Barchi for their help in every aspect of the project and for their continuous support, I wouldn't have done it without them.
I would like to thank my family for all the emotional and economic support during this project and all my university career.
Finally a special thank goes to my friend Alessandro that I consider as a brother, who is always there to support me, no matter what is the problem or the timing.

Bibliography

[1] The human brain project, 2016. Available at https://www.humanbrainproject.eu.

[2] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva. An efficient mpi implementation for multi-coreneuromorphic platforms. In *2017 New Generation of CAS (NGCAS)*, pages 273–276, Sept 2017.

[3] Simon Davidson. Pacman103-data structure generator encodings.

[4] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018.

[5] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, May 2014.

[6] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. Overview of the spinnaker system architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, Dec 2013.

[7] Steve Furber. To build a brain. 2012.

[8] Steve Furber. Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, 13(5):051001, 2016.

[9] W. Gerstner and W. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[10] G. Indiveri and S. C. Liu. Memory and information processing in neuromorphic systems. *Proceedings of the IEEE*, 103(8):1379–1397, Aug 2015.

[11] Wenzel Jakob. pybind11 seamless operability between c++11 and python. Available at http://pybind11.readthedocs.io/en/master/.

[12] Zipursky SL et al. Lodish H, Berk A. *Molecular Cell Biology*. W. H. Freeman, New York, 4th edition, 2000.

[13] The University of Manchester. Spinnaker data sheet v2.02, 2011.

[14] The University of Manchester. Dataspecification, 2015. Available at https://github.com/SpiNNakerManchester/DataSpecification.

[15] The University of Manchester. Pacman, 2015. Available at https://github.com/SpiNNakerManchester/PACMAN.

[16] The University of Manchester. Spinnfrontendcommon, 2015. Available at https://github.com/SpiNNakerManchester/SpiNNFrontEndCommon.

[17] The University of Manchester. Spinnmachine, 2015. Available at https://github.com/SpiNNakerManchester/SpiNNMachine.

[18] The University of Manchester. Spinnman, 2015. Available at https://github.com/SpiNNakerManchester/SpiNNMan.

[19] The University of Manchester. spynnaker, 2015. Available at https://github.com/SpiNNakerManchester/sPyNNaker.

[20] C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, Oct 1990.

[21] Walter Pirovano and Jaap Heringa. *Multiple Sequence Alignment*, pages 143–161. Humana Press, Totowa, NJ, 2008.

[22] Tobias C. Potjans and Markus Diesmann. The cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. In *Cerebral Cortex*, pages 785–806, March 2014.

[23] A. Siino, F. Barchi, S. Davies, G. Urgese, and A. Acquaviva. Data and commands communication protocol for neuromorphic platform configuration. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 23–30, Sept 2016.

[24] The SeqAn Team. Seqan manual. Available at http://seqan.readthedocs.io/en/master/#.

[25] Steve Temple. Appnote 1 - spinn-3 development board, 2011. Available at https://spinnakermanchester.github.io.

[26] Steve Temple. Appnote 3 - the aplx file format, 2011. Available at https://spinnakermanchester.github.io.

[27] Steve Temple. Appnote 4 - spinnaker datagram protocol (sdp) specification, 2011. Available at https://spinnakermanchester.github.io.

[28] Steve Temple. Appnote 5 - spinnaker command protocol (scp) specification, 2011. Available at https://spinnakermanchester.github.io.

[29] Steve Temple. Appnote 9 - spinn-5 quick start guide, 2011. Available at https://spinnakermanchester.github.io.

[30] Steve Temple. Sark - spinnaker application runtime kernel, 2011. Available at https://spinnakermanchester.github.io.