ΕΘΝΙΚΌ ΜΕΤΣΌΒΙΟ ΠΟΛΥΤΕΧΝΕΊΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΌΓΩΝ ΜΗΧΑΝΙΚΏΝ ΚΑΙ ΜΗΧΑΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΤΗΜΑΤΩΝ

Asynchronous Inter-Core Communication in the Inferior Olive Simulator for the Intel SCC

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Παντελόπουλου Ανδρέα

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Οκτώβριος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Τμημα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΤΗΜΑΤΩΝ

Asynchronous Inter-Core Communication in the Inferior Olive Simulator for the Intel SCC

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Παντελόπουλου Ανδρέα

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή επιτροπή την ημερομηνία εξέτασης

....	....	....
Δημήτριος Ι. Σούντρης	Νεκτάριος Κοζύρης	Κιαμάλ Πεκμεστζή
Αναπληρωτής Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Οκτώβριος 2014

....
Παντελόπουλος Ανδρέας
Διπλωματούχος Φοιτητής του Εθνικού Μετσόβιου Πολυτεχνείου

Περίληψη

Το Single-Chip Cloud Computer(SCC) είναι μία πειραματική πλατφόρμα, των εργαστηρίων της Intel, αποτελούμενη από 48 P54C Πέντιουμ πυρήνες. Η διπλωματική εργασία αυτή ασχολείται με την μεταφορά στην πλατφόρμα του προσομοιωτή δικτύων των εγκεφαλικών κυττάρων πυρήνα κάτω ελαίας, δίνοντας έμφαση στην επικοινωνία μεταξύ των πυρήνων κατά την εκτέλεση της εφαρμογής. Η συγγεκριμένη εφαρμογή ασχολείται με ένα πολύ σημαντικό σύνολο εγκεφαλικών κυττάρων, τα κύτταρα πυρήνα της κάτω ελαίας. Διαφέρει από τα συνήθη νευρωνικά δίκτυα, καθώς προσομοιώνει ένα βιολογικά ακριβές μοντέλο, με τέτοιο τρόπο ώστε να αποτυπώνεται όλα τα βιολογικά χαρακτηριστικά των κυττάρων κατά την διάρκεια της προσομοίωσης.

Η εργασία αυτή εστιάζει στα παρακάτω. Αρχικά εξετάζεται σε βάθος η συνδετικότητα μεταξύ των νευρώνων της Κάτω Ελαίας.Έπειτα εφαρμόζονται βελτιώσεις στον αρχικό κώδικα του προσομοιωτή. Τέλος εφαρμόζονται βελτιώσεις σχετικές με την επικοινωνία μεταξύ των πυρήνων της πλατφόρμας.

Ένα μοναδικό χαρακτηριστικό του SCC είναι ο Message Passing Buffer (MPB), ο οποίος αποτελείται από μικρές SRAM caches, μία για κάθε πυρήνα, για την επικοινωνία μεταξύ των πυρήνων. Με την χρήση αυτής της προσθήκης καταφέραμε να επιταχύνουμε την προσομοίωση.

Λέξεις κλειδιά: SCC, Intel, RCCE, HPC, Επιτάχυνση, Επικοινωνία, Κύτταρα Κάτω Ελαίας, Εγκέφαλος, Δίκτυο Κυττάρων, Προσομοιωτής

Abstract

The Single-Chip Cloud Computer (SCC) is an experimental platform, created by Intel labs, with 48 P54C cores. This thesis covers the optimized porting of the Inferior Olive Cell Simulator on the SCC, especially from an inter-core communication perspective. The targeted application involves a very important set of brain cells, namely the Inferior Olive Cells. It differs from usual neural networks, in a way that a biologically acurate model is used, thus exploitting full biological information concerning the cells.

The contributions of this Thesis lie in the following domains. First neuron cell interconnectivity of the Inferior Olive Model (InfOli) is explored. Then source Code optimizations of the original InfOli Model are achieved. Finally inter-core communication optimizations, regarding the application, are introduced.

One of the distinguishing features of the SCC is the Message Passing Buffer (MPB), whereby small SRAM caches are assigned to each core for communication purposes. With the use of this feature, thus impressive speedups are achieved.

Keywords: SCC, Intel, RCCE, HPC, , Acceleration, Communication, Inferior Olive, Power, Brain, Cell Network, Simulator

Contents

List of Figures

Chapter 1

Introduction

Current thesis focuses on the optimized porting of the Inferior Olive (InfOli) Simulator on the experimental Intel Single-Chip Cloud Computer. The target application simulates a very important set of brain cells, the InfOli cells. The InfOli cells are associated with brain functions involving motor skills and space perception. In this work a biologically accurate neuron simulation is utilized, thus full biological functionality of the neurons is exposed. Different communication schemes and connectivity schemes on existing porting options have been explored, and great emphasis is given in reducing CPU times and measuring the platform power consumption.

The motivation behind this thesis, was the writers interest in understanding part of the human brain functionallity, through the perspective of a computer scientist. The Cerebullum is one of the most important human brain parts, and one of the most complex. Time demanding simulations were already performed for the targeted simulator, but there were needs for further acceleration, such as the ones achieved in this thesis. After the study of the platform and the application involved, we managed to harvest the best out of the SCC board, while adding novelty to brain related research.

Chapter 2 holds a state-of-the-art analysis regarding the modelling and engineering aspects of the simulation, as well as an overview of the SCC platform. Related work is explored, with emphasis on alternative hardware targets used in the modelling of the InfOli Neurons, comparing them to our contribution. Different hierarchical models regarding neuron simulations are also discussed. Afterwards, the SCC board is explored in depth, along with the utilities that are provided by Intel to facilitate programming of the chip. Special emphasis is given to unique memory characteristics of the chip, that enabled impressive simulation times, reducing the inter-core communication overhead of the simulator.

Chapter 3 examines closer the InfOli simulator. The biological model is introduced here, and InfOli neurons are analysed. Simulation runtime and porting options are described, allong with various connectivity schemes between the neurons. Optimizations are inroduced, on the basis of previous code versions of the InfOli simulator. Connectivity schemes are examined with respect to CPU time and Power Consumption. Related work results are also outlined.

Chapter 4 explores the applicability of unique memory mechanisms of the SCC platform on the optimization of the InfOli runtime. Having used those underlying mechanisms and the unique on-chip RAM of the platform, this thesis presents an efficient asynchronous communication scheme based on memory polling between the SCC cores. Libraries provided by Intel developers were efficiently used towards this direction. Thus, it was feasible to exploit in depth software/hardware specific mechanisms of the SCC, while achieving impressive simulation speedups, which are also presented at the end of this Chapter.

This thesis concludes in Chapter 5 which summarizes results and respective observations. Remarks about the application and the platform are made, in addition to future work references.

Chapter 2

State of The Art and Platform Description

2.1 Prior Art Overview

2.1.1 Biological Aspects

The understanding of human brain mechanisms and functions is of a great importance. Scientists strive to map the human brain for years. Such explorations would lead to great achievements, including the understanding of the human cognition and behaviour, as well treat many brain-related diseases. Several large scale projects focus on those goals today [4], untreatable diseases are also believed to connect with the brain functions and mechanisms [5].

The Cerebullum, is involved in the learning and timing of movements, and comparing intended movement decisions with achieved movements [6]. The olivary nucleus has been proved to interact with the above functions of the human brain [7]. As the functionality of the Cerebellum becomes increasingly clearer, targeted experiments and models can be produced, in order to further understand the mechanisms of human movement and control.

Brain modelling is distinguished from other model procedures, due to its complexity. Construction and refinement of qualitative and quantitative models for the brain observed phenomena, is the standard in neuroscience, and other scientfic fields. However the building of neural models consists of different hierarchical levels of analysis [8], in contrast to usual modelling techniques. Several threads of neural modelling have arrised in the past years. The differences between them relate to the different abstraction maintained towards brain simulation.

As we can see in [8], neural modelling is is divided in two main threads :

- Conventional Reductive Models

  This type consists of the traditional reductive modeling. These types of models are usefull in describing the neural phenomena as well as providing explanations of them, with the use of mechanisms that may generate them. Those mechanisms are also described in terms of models, so the whole process is recursive. It is common for those models to be expressed with a set of mathematical equations, which provide the formulation needed for quantifying the phenomena involved.

- Computational Interpretive Models

  The idea behind these models resides in the interpreting of the brain mechanisms in terms of the underlying computations that are involved in order for a particular task to be completed. The main aspects of such modelling, is the representation, storage and transformation of information in the neural network [9]. Since the above models involve computational and algorithmic planes, they can be represented with the help of traditional programming languages. Those models offer a high degree of detail and simulations conducted with them are very computationally intensive. That is why HPC architectures prove extremelly usefull in neural modelling. The level of analysis of Computational Models may vary in terms of abstraction. Further classification is possible, exposing Conductance-based, Integrate-and-fire models and Firing-rate models. The InfOli Model falls under this category, as full biological information is exposed during the simulation, due to the full modelling of the underlying mechanisms.

The InfOli model that has been used in the contect of this thesis, is an extended version of the Hodgkin-Huxley classical neuron model [10]. Other approaches in neuron simulations, like neural networks [11] involve methods from the Artificial Intelligence field. They are really abstract in terms of the underlying actual mechanisms, thus no biological information of the neurons states during the simulation is extracted.

The Hodgin-Huxley model treats each cell component as an electrical element, which is described with a set of non-linear differential equations. The differences in the compartment's voltages result in current flowing through cells, thus producing output for Purkinje-Cell (PC) through the so-called Climbing Fibers (CF) [2]. In this thesis, we focus on the InfOli neurons, rather than the other parts of the Cerebullum.

Biological parameters are described in terms of the electrical characteristics of the cell compartments, and the conductive channels associated with them. The model falls under the Spiking-Neural Network class of models , since each simulation step is driven by spike inputs flowing in the cell network.

2.1.2 Engineering Aspects

In order for the brain models to prove usefull, they have to be easily portable to platforms, thus enabling simulations of sufficient detail and duration. This way, a subset of the brain's functionallity could be simulated, in some level of detail, providing usefull results. Simulations also give neuro scientists the alternative needed to partially avoid in-vitro and in-vivo experiments, which are costly, time-consuming and may require testing on animals, [3]. In general, simulating brain parts consisting of large cell inventories calls for significant computational resources. Applications like these prove also to be of high inherent parallelism, but consist of elaborate communication schemes, since neurons are usually inter-connected in non-regular ways.

Exposing parallelism and reducing communication is critical in every neuron simulation, thus using the appropriate platform is of paramount importance. Platforms with different characteristics have been used for neuron simulations, such as Field-Programmable Gate Arrays (FPGA), Graphical Processor Units (GPU), multiprocessors, Network-on-Chip's, High Performance Computing (HPC) platforms and more. The advantages and drawbacks of each platform play a major role in the usability and the results that neuro-scientists will gain. Simulations can be devided in real-time simulations and large-scale simulations. Real-time simulations usually involve low-populated networks of cells, in order to reach real time speeds, while large-scale simulations sacrifice real-time characteristics in favor of larger cell populations. Both approaches play a major role in understanding the human brain.The challenge here for engineers is to achieve hybrid solutions in order to achieve near optimal platform utillization, conducting real-time simulations of large-scale networks. Contributions toward the above direction would be very useful in biological applications running on robots [12] and other embedded approaches [13].

2.1.3 FPGAs

FPGAs are typically used to speed up applications that exhibit bit-level inherent parallelism [14], and accelerate HPC applications [15]. A typical drawback of this is the need to design a differenct coprocessor for each application involved. Neuron modelling simulations, are considered to be fine grained parallelized applications, thus hardware implementations seem to fit those models. It is true that impressive simulation speedups are achieved with FPGA implementations [16]. Those implementations offer a compromise between the hardware efficiency of ASICs and the flexibility of software-based solutions. Especially in real-time simulations, and on-chip solutions, FPGAs could be really usefull for creating specific devices such as those used in robots. However, FPGAs show some limitations in neural modelling. For one, FPGAs are not easilly programmable implementations, so they provide limited flexibility, while neural modelling is a constantly evolving field with changing requirements. This results in design complexity exceeding that of other platforms [14]. Ideally one should only manipulate the software aspects of the simulation, while achieving high performance.

However, in the FPGA case, the alterations are constrained by the synthesis feasibility. In many cases, complex and elaborate data structures that are required from a neuroscientific standpoint, may not be synthsizable on an FPGA. Hence, FPGAs come with the optional overhead of data type refinement.

2.1.4 GPUs

Simulations on Graphical Processor Units (GPUs) seem to have gained considerable attention in the recent years. Graphic cards are cheapper and easier maintained in comparison to clusters, while prove to be more software reconfigurable than FPGA implementations [17] involving lower time-to-solution implementations. However, producing code for GPUs could be a tricky procedure. All GPU implementations should take into account the memory bottleneck rather than pure computation speed [18]. However for neural models like the one implemented in this thesis, which includes managing structures of linked lists in a non-uniform manner, GPUs would not be efficient. The reason behind this is that linked lists are not comprised by consecutive memory, thus GPU acceleration is inefficient. Another drawback in GPU implementations is the lack of communication mechanisms between the majority of running threads. In the InfOli model, the communication phase inititating each simulation step, would involve calling seperate GPU kernels, rather calling one kernel for the entire simulation, thus resulting in major performance drawbacks.

2.1.5 HPC platforms

Since the brain consists of very large neuron inventories, it is only natural for super-computer architectures to play a major role in neural simulations. Supercomputers come at a high cost, and with demanding adminsitration issues. Nowadays, it is possible to simulate parts of the brain on supercomputers, and supra-linear speedups are achieved [19]. Recently, the largest brain simulation ever achieved was performed in Japan, with the use of K Computer, which lasted 40 minutes and completed a simulation of 1 second of brain activity [20]. It is widely believed that exascale super-computers, that are likely to be manufactured in the next decades [21], will manage to simualte the whole brain in logical CPU times. However, large-scale simulations with super-computers, may not be quite usefull for real-time applications or devices, which should interact constantly with the patient, or the system, and of course be portable.

2.1.6 Manycore / Multicore Platforms

In the recent years there has been an outburst in multiprocessor desktops. While their pure computing capabilities are significantly lower to that of the GPUs, they are easily programmable and maintainable, and they require no time consuming Peripheral Compnent Interconnect (PCI) transfers [22]. The use of multichip processors lead to increasing research and production of many-core chips. Those many-core chips are currently used as accelerators, connected on the PCI bus, resembling GPUs, but research is being conducted towards the production of many-core CPUs. The cores of those chips are superior of GPU cores in terms of logical complexity and branch-prediction mechanisms, and are programmed like cores on a Desktop. We can deduct from the Intel Roadmap [23] that in the recent years, all architectures will include a significant amount of cores and threads for use.

Although the work in this thesis involves non-real time large scale simulation, the research in the field will make feasible for more accurate and real time simulations to be achieved by single many-core chips, like the SCC. This will make possible the produce of implantable devices, that will maybe diagnose brain problems, or even correct them at real-time, while being easily reconfigurable [24]. Progress towards this scope is already achieved in small animals [25].

Usually single multicore chips, consist of network of cores residing close to each other, withe their own unique characteristics. This results to low inter-core communication times. As a result, the production of low-latency communication schemes and the speeding communication demanding applications such as neuron modelling is fully enabled. One major drawback of many-core chips is that there is no available core expansion, since all cores reside on the same chip, thus the platform is restrictrive from any application scaling.

A distinction among many-core chips is the location of the memory. They are divided mainly in two categories : shared memory chips, and distributed memory chips. In fact most platforms implement a hybrid memory model, combining both approaches. A platform like this is the SCC board, since it both has off-chip shared memory, as well distributed memory residing on every tile.

While each target platform has its advantages and disadvantages, none is the ideal for neuron modelling. It is possible that hybrid solutions will emerge in the future years [26] [27], targetted in neural simulation, which will combine characteristics from various platforms. Research on the field of brain modelling and brain exploration is constantly expanding, as organizations fund researchers all over the world towards this scope [28].

2.2 Single-Chip Cloud Computer Platform description

2.2.1 Chip description

The Single Cloud Chip is a second generation processor design developed by Intel in order to advance research on multi-core chip architectures. It is the predecessor of a 80-core chip, Teraflops Research Chip [29].

The Intel SCC chip consists of 24 tiles, each tile containing two cores. The SCC core is a full IA-P54C core, thus it can support the compilers and Operating Systems technology required for full application development. The 24 tiles create a 6×4 mesh network.

The SCC platform is connected over a PCIe bus with the Management Console PC (MCPC). The MCPC runs a Linux distribution and is used for easily managing the SCC chip and porting applications. The MCPC directory `/shared` is NFS-mounted on the cores. Interraction with the SCC is done through this shared directory

2.2.2 Tile description

Each individual core uses an L1 Instruction cache and an L1 Data cache, with 16 KB capacity each. Every core also has an L2 cache of 256 KB capacity. Both the L1 caches are on the core, while the L2 caches are placed on the tile next to the cores.

Located next to the cores is the Mesh Interface Unit (MIU), which connects the tile to the mesh. It is responsible for packetizing/depacketizing the data in and out from the mesh. It is the component controlling the flow of data on the mesh with a credit-based protocol, while using a round-robin scheme to serve core requests.

A small 16KB SRAM buffer is also located on each tile, namely the Message Passing Buffer (MPB). This buffer is shared between two cores residing on the same tile. The MPB serves as the only on-die inter-core communication aid. The MPB is accessible by all SCC cores, thus with the right software libraries message passing mechanisms can be produced. A more detailed elaboration of the MPB and its use will be performed in the next subsection. Finally, a traffic generator is located on the tile, used to test the performance capabilities of the mesh.

Platform layout is illustrated at figure 2.1

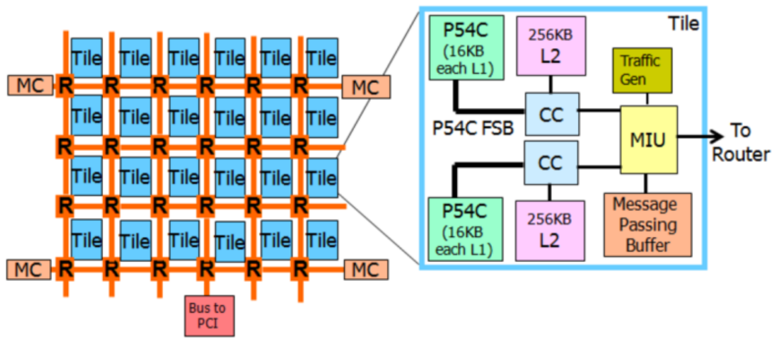

FIGURE 2.1: Single Cloud Chip layout from [1]

2.2.3 Memory organization and basic I/O

The SCC memory consists of the global off-chip DRAM and on-chip SRAM per tile. The off-chip DRAM is accessed through four on-die memory controllers. The DRAM has a capacity of 64GB, which is in the default setting, divided or shared between all cores. The physical memory assignment is perfomed at boot, and can be changed by the programmer by modified the Lookup Tables (LUT) for each core.

The platform is divided into four distinct regions. Each memory controller is responsible for a particular region, serving requests from and to the memory. Memory controllers also support shared memory between the cores, thus access to shared memory region may pass through any memory controller. Since the core addresses are 32-bit and total memory of the chip is 64GB, a translation occurs in order for a core to access the off-chip memory. This translation is made possible with the use of look up tables, private for each core, mapping each virtual address of a process running on a core, to a physical address at the off-chip DRAM. The MIU performs this translation, before issuing core requests at the mesh.

In addition to the off-chip memory, each tile has an on-chip SRAM, which is accesible by all other cores and is called Message Passing Buffer (MPB). This memory is also addressed with the help of the translation process mentioned above. The size of the MPB per core is 8KB, thus resulting in a 384KB message passing buffer available for all the cores. This buffer is mapped on the virtuall adress space of some process running on a core, thus every process running can modify its contents by simple read/write memory operations. Typically, users do not modify directly this memory, instead libraries provided by Intel Labs are used. That way users do not need to concern with cache invalidation and other memory specific issues.

As mentioned before, the MCPC local `/shared` directory is NFS-mounted on the cores. All files in that directory are directly accesible by all the SCC cores. There reside all usefull configuration files needed for the applications (for example simulation parameters for the Inferior Olive simulator) as well as all output files created by the cores.

2.2.4 Programming Model of the SCC

2.2.4.1 Programming Model

Application developers usually work on the MCPC, where the SCC is connected via PCIe bus. Residing there are all the utilities and developing tools needed to compile, port and execute applications on the SCC. Users develope source code according to SCC standards, compile it using the cross-compiling tools provided by Intel and port it on the chip.

All cores of the SCC are accesible through TCP/IP protocolls and thus can be reached with ssh sessions. Intel Labs provide developers with a bash script, namely `rccerun`, which has the role to port applications on the board. Developers make use of this in order to conduct experiments on the platform.

The proccess of porting an application is the following :

- Develop code for the application at the MCPC.
- Cross-compile code for the P54C architecture.
- Port application on the SCC, using `rccerun`, with the appropriate arguments
- Retrieve output files, residing at the local mounted directory

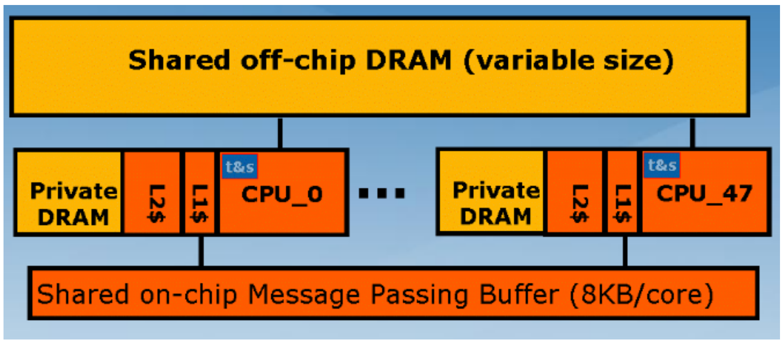
Shared off-chip DRAM (variable size)
Private DRAM
L2$
L1$
t&s
CPU_0
...
Private DRAM
L2$
L1$
t&s
CPU_47
Shared on-chip Message Passing Buffer (8KB/core)

FIGURE 2.2: Programmer's point of view from [1]

2.2.4.2 Underlying message passing mechanisms

Communication between the SCC cores is achieved with the use of the Message Passing Buffer (MPB). In order for a core to pass a message , it has to write that message in the target core's MPB, and the target core has to read it accordingly. Since the MPBs reside on the chip, communication overhead is very small [30].

Intel has developed the `RCCE` library, which consists of a set of functions in C langugage. These functions take care of the internal mechanisms of message passing, and also help configure the platform. Thus the user need not explicitely write to the MPBs in order to achieve communication, but use the `RCCE` library functions.

`RCCE` library has two main modes, `gory` and `non-gory` mode. The main difference between those two modes is the level of detail that is provided to the developer concerning message passing. The `RCCE` library was mainly used for message parsing between the cores. The gory mode will be covered in more detail, since extensive use of it was necesary in order to perform the asynchronous communication, described in Chapter 4. Both versions provide a runtime similar to that of the MPI runtime system. Each core is assigned a unique ID, distinct from all other cores running and defines Communicators in order to communicate with other SCC cores. This programming model is similar to the classic model provided by MPI runtime system, which is used in most parallel platforms.

- Non-gory RCCE library

  Non-gory library mode provides two basic synchronous functions for Message Passing, `RCCE_send()` and `RCCE_receive()`. These two library functions preserve the usuall semantics of the MPI send() and receive(). They involve sender and receiver cores, as well as pointers to the virtual source and target memory. The library also provides functions for collective communication between the cores such as `RCCE_scatter()` and `RCCE_gather()`, as well as the alternative asynchronous send and receive functions. Barriers are also provided with the function `RCCE_barrier()`, in order for synchronization to be achieved between running cores. The Message Passing interface is straightforward to any user with prior experience with MPI runtime system [31]

- Gory RCCE library

  Gory library mode provides explicit detail of the underlying hardware and message passing mechanisms of the platform. It allows the user to allocate and deallocate memory on the MPBs, as read and write data from every MPB on the platform. Using this, an experienced platform user is able to further accelerate a ported application, as also reduce communication overheads. Each core uses functions `RCCE_malloc() and` `RCCE_free()` in order to allocate and de-allocate memory in its MPB. In addition, data can be written to an MPB by a core with the use of the `RCCE_put() and RCCE_get()` library functions accordingly. This function receive as an argument the core's ID that will access its memory. [32] The Gory communication model adopts a shared name space or symmetric memory model. This programming model allows a programmer to reference data structures stored within the MPBs of every core.

This model makes the assumption that RCCE calls are encountered jointly by all cores of the platform. This way, all cores would call `RCCE_malloc()` jointly, thus reciveing memory at the same offset of their MPBs. Consider an example where each core `RCCE_malloc()` memory for an object `X`. Since all cores malloc the same object X jointly, all will receive memory for this object at the same offset in their MPBs. Thus since each core can access other core's MPBs it can access this object using this offset. When Non-Gory library mode is used all the above details concerning the MPB memory operations remain hidden.

In figure 2.3 the above collective logic is illustrated :

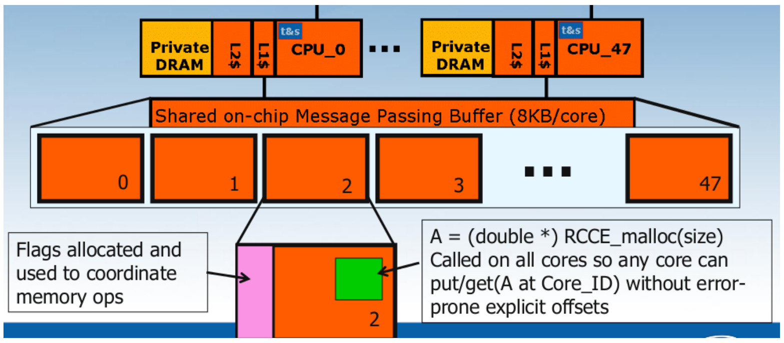
Private DRAM
L2$
L1$
t&s
CPU_0
...
Private DRAM
L2$
L1$
t&s
CPU_47
Shared on-chip Message Passing Buffer (8KB/core)
0
1
2
3
...
47
Flags allocated and used to coordinate memory ops
2
A = (double *) RCCE_malloc(size) Called on all cores so any core can put/get(A at Core_ID) without error-prone explicit offsets

FIGURE 2.3: Collective allocation calls from [1]

After the collective memory allocation, cores use `RCCE_put()` and `RCCE_get()` in order to manipulate data in the MPBs. Figure 2.4 illustrates this concept:

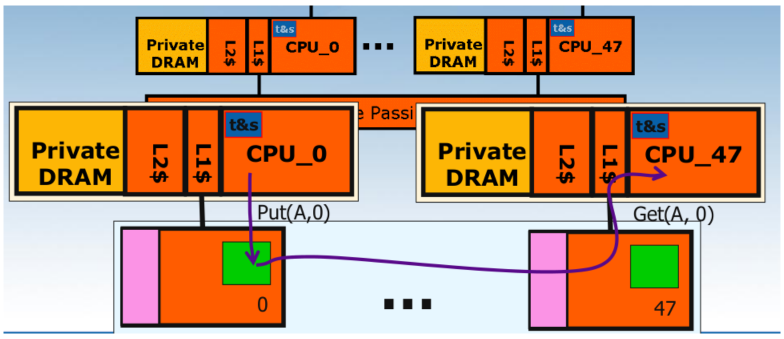

FIGURE 2.4: Message Passing through MPBs from [1]

The collective communication model is restrictive, in a way that memory should be allocated in every MPB of the chip, while only a subset of the cores will communicate. This restriction was avoided in the communication scheme presented in Chapter 4.

Chapter 3

Model Description

3.1 Application and Porting

3.1.1 Application Description

InfOli Neurons provide a major input to the cerebulum. They receive their input from Deep-Cerebellar-Nuclei (DCN) cells and transmit their output to Purkinje-Cells [7]. Each InfOli cell is divided in three compartments, the dendrite, the soma and the axon. Each compartment has its own unique characteristics and serves a different biological role. In the contect of this thesis, InfOli simulation proceeds in descrete time steps equal to 50ms. At every step the biological behaviour of the three compartments is simulated, as well as the communication between InfOli cells.

- Dendrite

  Dendrite represents the input connections to the cell's body . They receive input current from DCN cells as well as neighbouring InfOli cells and forward their output signal to the soma compartment.The interfaces with neighbouring InfOli cells are called gap-junctions. In general, dendrites is the compartment that deals with cell communication.

- Soma

  Soma is the neuron cell's body. Through electrochemical reactions, the soma transforms the input signal from the dendrite into a response signal through the axon. It is the most elaborate and time consuming compartment that deals with inter-cell communication.

- Axon

  Axon can be considered as the output connection of the IO cell. It transmits the electrochemical output signal to other brain cells, Purkinje-Cells, through the so called Climbing Fiber.

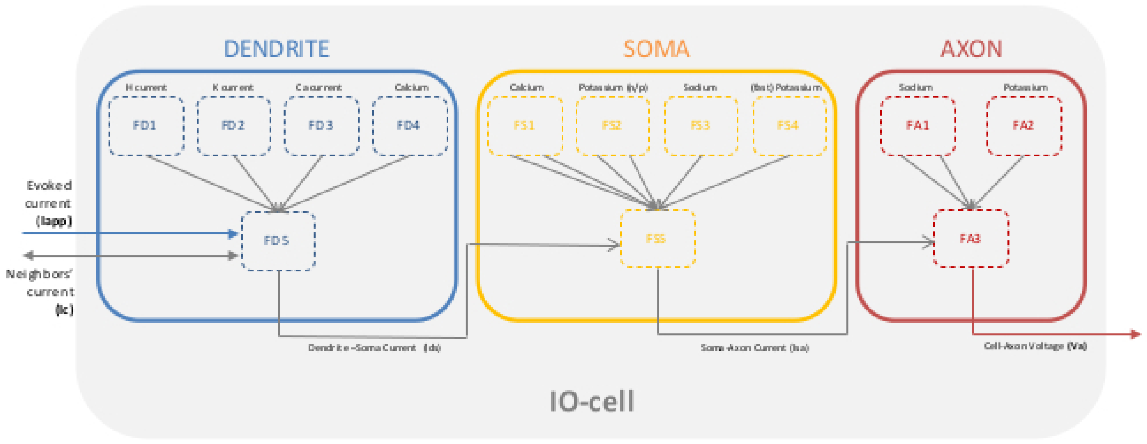

FIGURE 3.1: Illustrates an InfOli Cell, from [2]

In the context of this thesis the network of InfOli cells can be viewed as a two-dimensional mesh. Every mesh node (cell) is fed with an electric current from DCN and, in turn, signals an output value through its axon compartment and the CF. The computational model works in a synchronous way, in every step: all mesh nodes compute their output and their new state based on the current inputs and their previous internal state. The time flow of the simulation is illustrated in Figure 3.2.

3.1.2 Application Porting

Prior to the simulation step loop the application allocates memory for all the structures involving the neurons information as well as all the structures participating in the communication between InfOli cells. Structures involving the biological data of the neurons are initialized with random data.

Axons are fed input currents in every simulation step. Those currents may be derived from user-defined files, or alternatively a hard-coded spike can be

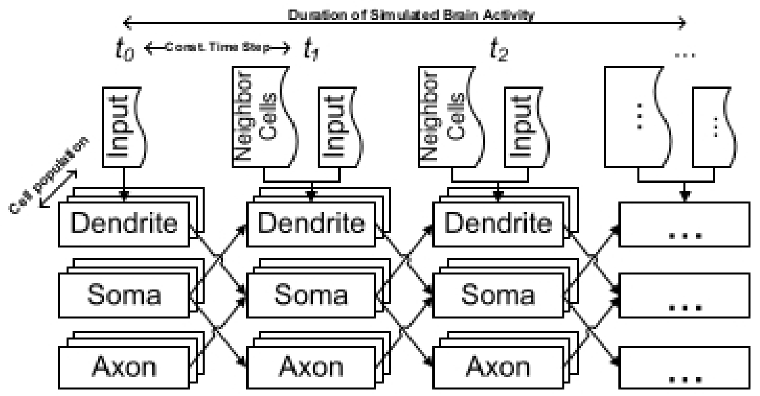

FIGURE 3.2: Simulator Runtime from [3]

simulated.Throughout the work presented herein, the second method is used for debugging and experimental purposes. Originally, 8-way connectivity was assumed as the inter-connectivity scheme of the InfOli cells, meaning that only neighbouring cells were forming gap junctions. In the context of this work, a different model o fcell inter-connectivity has beed devised, which is more realistic, being of a more stochastic nature. This will be covered extensively later on the chapter.

After the communication of input currents and neighbouring IO voltages, the compartments share the voltage levels of the previous simulation step. Every compartment then has aquired all the information it needs, and compartment simulation begins. Specifically for the axon compartment, it's new voltage level is recorded to output files and the simulation proceeds to the next step.

This process is repeated until the input current file ends, or 120.000 steps of the hard-coded input spike are performed.

Previous work [33] involved two major porting options of the simulator.

- Data Partitioning

  This porting option distributes InfOli cells to the cores. We refer to data as the cells that make up the neuron mesh. Each core simulates all three compartments of all InfOli cells that are distributed to it. This is the most intuitive mapping on the SCC. Simulations involve grid sizes that are multiple of 48, in order to avoid unballanced workloads accross the SCC cores. Each core will simulate a total of $N \div 48$ InfOli cells, where N is the number of InfOli cells in the grid.

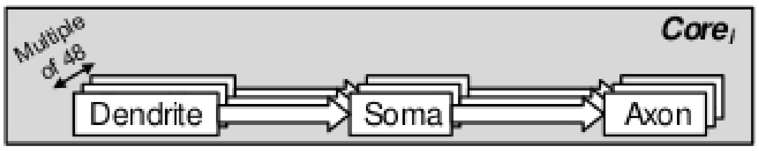

Figure 3.3: Data partitioning, from [3]

•Task and Data Partitioning

This porting option differs from the previous in the way that each core simulates a specific compartment for many InfOli cells. We consider a single task to be the simulation of a cell compartment. Since SCC consists of 48 cores then we can assign every compartment to a total 16 cores. Each core simulates a total of $N \div 16$ InfOli cells.

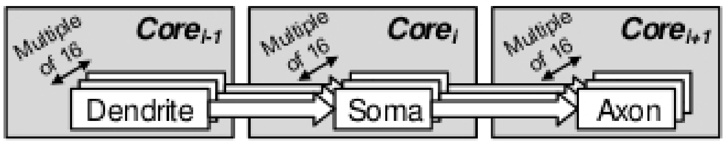

Figure 3.4: Task and Data Patitioning, from [3]

The evaluation of the above porting options in [3] showed that the first porting option is the most efficient, in terms of energy and CPU time. Since SCC cores are of the same nature it is not natural to enforce unballance between their workloads. Also increasing communication overhead decreases the inherit parallelism of the application, thus resulting in higher simulation time [34].

3.2 Inter-Connectitity

In general, networks of neurons for a specific brain part, do not receive solely input from other brain parts, but in addition have a constant interaction with each other. They receive signals from thousand other neighbouring neurons, that are propagated through the dendrite and integrated in the soma.

As mentioned before, the InfOli cells can be represented as a 2D mesh, where each node is a cell. Cells form connections between them, which are named gap-junctions. A gap junction between two cells represents a voltage ecxhange in the communication phase of the simulation. To be more specific, for a given gap-junction between two cells, current will flow from one to another because of the voltage difference betweem them. This results in neighbour current flowing through the dendrites, apart from the usual current channels that are specific for each cell compartment.

3.2.1 Normal Distribution

Considering how important the interaction between neurons in the same network is, it seems crucial to devise a stochastic connectivity model which describes the actual connectivity schemes of the neurons. While we can not always specify how two distinct neurons will interact, we can impose general models that will give the probability of two neurons forming a gap junction.

The nature of the connectivity between cells, plays a major role in the application behaviour and the CPU times of the simulation. In the previous work [33], cells were connected in an 8-way scheme. This meant that each cell interacted only with the neighbouring cells directly and no other, thus imposing a maximum of 8 connections per cell. This communication scheme does not fully comply with the neuron nature. It appears that neurons can form various connections with non-neighbouring cells. In fact, it is likely that nearby neurons will share more common inputs than distant neurons, so interaction is expected to decrease with interneuronal distance [35].

3.2.2 Stochastic Neuron Connectivity

Consider two neurons, at a distance r between them. The probability to form a gap-junction is given in Equation 3.1

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(r)^2/2\sigma^2} \tag{3.1}$$

This equation represents the stochastic connectivity model, and the Normal distribution of the connection probability. The connections file is generated before the simulation, and is given as an input to the program running at the platform. This connectivity file is parsed before the main simulation loop, and the according structures that are specific for communication are initialized accordingly. We should note here that the creation of the connectivity file is generic. This means that the user can create connectivity schemes with different probability equations easilly.

The distance r can be any distance metric the user finds appropriate e.g. Manhattan, Euclidean, Euclidean Squared etc. The simulation time varies depending on the above parameters, since different gap-junctions will be formed every time. It is important to note that simuation with stochastic communication schemes are more time elaborate than simulations conducted with the previous 8-way connectivity scheme, due to the larger average number of gap junctions formed, thus larger communication simulation steps.

3.2.3 Analysis

In order to have a high-level view of the connections per cell, as well as a time estimation per simulation, 3D plots were generated. Furthermore, since probability values, derived from equation 3.1 will determine the number of connections per cell, a 3D diagram has been plotted, representing the possibility of two cells to form a gap-junction for various for various network sizes.

Figures 3.5 and 3.6 from [3] illustrate the above :

In figure 3.5 we see that probability is highest, when inter-neuron distance is small. This is only natural, since the Normal distribution is used in determining

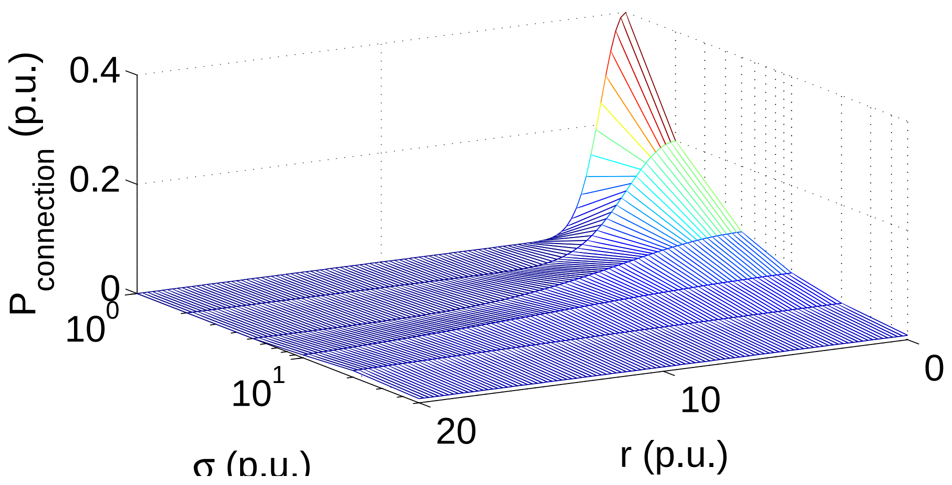

Figure 3.5: Connection Probability according to equation 3.1

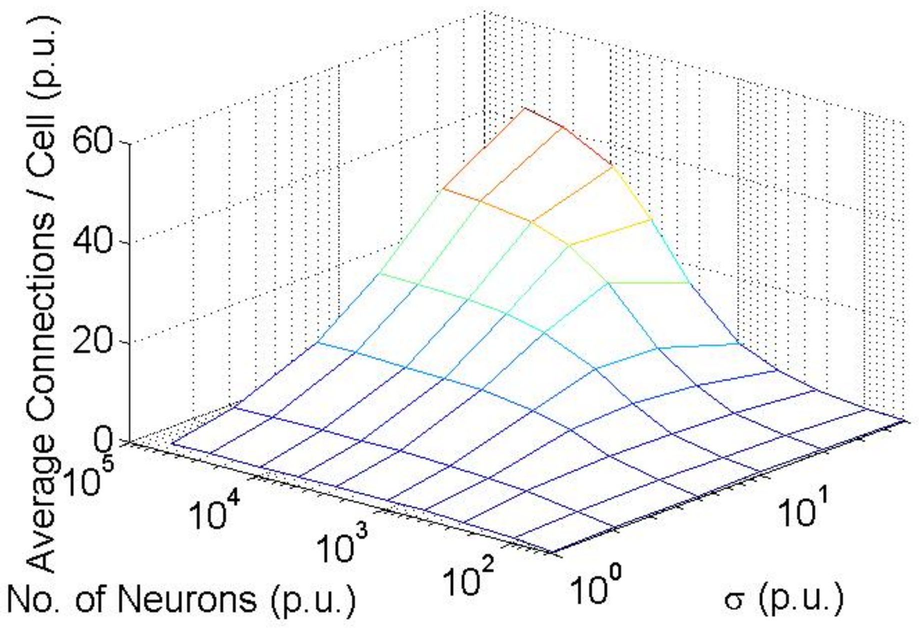

Figure 3.6: Average number of connections for various parameters

the connections between the neurons. In figure 3.5 we notice that increasing network sizes the peak of average connections change accordingly. It is also noted that for a specific network size, average connections per cell does not increase with the σ

of the distribution equation.

We can also see that for a specific value, average connections per cell increases, however after a specific point the number of connections decreases. This is happening because the neuron mesh is finite, while the connection spread advances, [3].

Up to a specific point, the connection spread overcomes the max distance between two neurons, thus average connection per cells is decreasing. Plots like the above are of a major significance, since the CPU time of the simulation can be evaluated, as far as the user-defined simulations parameters are concerned. For different probability equations similar plots could be extracted

3.3 Code optimizations

In this thesis major optimizations concerning the original implementation of the Infoli Simulator were performed, that resulted in larger simulated IO networks, as well as speedups in the CPU time, which resulted in more power friendly simulations. In this section will we present the basic structrues involved in the simulation, and the general optimizations that were imposed on the original porting.

3.3.1 Major Structures

Before the start of the simulation, each SCC core determines the count of cells that will simulate.

`int cellCount` : Holds the total number of cells simulated by each core.

`cellPtr` : Array with pointers to cellState arrays. Different array is used in each simulation step. For example, if cellParams[0] has the values of the previous simulation step, then in cellParams[1] will be stored the next step's values. In the next step, cellParams[0] will store the new voltage values.

`cellParamsPtr` : Array with structures involving cell information, currrents flowing, and neighbour voltages. It stores pointers to cellPtr structure for determining next and previous cell states for each cell simulating. Before each compartmental simuation this structure is filled with appropriate values.

`communication_list_head` : Pointer to a linked-list containing a node for each cell that this core will communicate with.

`cores_table` : Array of pointers to intra-core communication structures. A NULL pointer indicates that no communication is performed with a specific core. This array is filled according to the communicating cells from the above linked list.

`conFile` : Filename of the connectivity file. Parsed from all cores in order to determine the neurons connections throughout the grid.

After the initilization of the above strutures, cores engage the simulation loop, perform the following two tasks until the end of the simulation :

- *Communication* : Communication is performed here.

  Function `Pefrorm_Gory_Communication` handles all core communication, through the on-tile Message Passing Buffers.

- *Compartments Simulation* : Functions `CompDend, CompSoma and CompAxon` handle the compartmental simulation for each core cell. They receive as parameters a cell specific cell structure from `cellParamsPtr`.

3.3.2 Optimizations

Original structures concerning the states of the InfOli cells were arrays. Those arrays were changed with linked lists, thus better memory management of the application was achieved. Original porting could simulate a maximum 96.000 InfOli cells. After that network size applications running on the cores were excessive in memory demanding, resulting in being terminated by the Linux operating system running on each core.

After the changes in the data structures, applications were running with no problem for network sizes up to 500.000 InfOli cells. No obsollete memory allocations were performed. In addition, algorithmic changes were also introduced in the handling of all above structures. The connectivity file was changed in a way, that it was parsed twice as fast, than before.

Originally, for each cell, connectivity file contained the cells that would transmit voltages to this cell, and the cells that that would receive voltage from this cell. This was obsollete, and instead, for each cell communicating we placed a character representing the type of communication in the connectivity file. Those characters are used in the communication schemes described later. Since for large sizes connectivity files are relatively large and are located in the local mounted file system, parsing them required a lot of CPU time, as well as excessive data transfer through the memory controllers of the board. Changing them enabled us the reduce of the pre-simulation CPU time.

With all the above optimizations the InfOli simulator performed 30% better than before and larger cell sizes could also be simulated, figure The major optimizations are introduced in Chapter 4 though, were the MPBs are used.

Chapter 4

Asynchronous Communication

4.1 Communication Overhead in InfOli

In parallel applications, communication overhead plays a major role in the design and the application porting. The higher the communication needs of the application the less speedup we can achieve through concurrent labour [34]. The inherent parallelism in the Infoli simulator in a specific simulation step is obvious, since in each simulation step InfOli cells perform arithmetic computations independently. However, InfOli cells need their neighbours' previous values in order to simulate next cell states, through the interactions formed between them (gap junctions). SCC cores simulating a specific part of the cell inventory must determine their neighbours and communicate through high level functions provided by `RCCE` library [32].

It is obvious that the higher the average connections per cell, the higher the impact of communication overhead for the InfOli simulation will be. This is heavily dependent of the distribution that the user will choose in order to spread the connection probability across the network. Every core must transmit to participating cores all InfOli voltages from cells that form gap junctions, before proceeding to the computation part of the simulation step.

Originally, each core participating transmitted messages that consisted of the voltage (type `modrprec`) value as well the source cell (`sender_cell_id`) and the target cell (`receiver_cell_id`). Cores created linked lists of type `struct` `communication_list` that contained all the required inofrmation. A major optimization implemented here was the reduction of the message size to contain only the participating voltage that was transmitted. That was made possible due to the sorting of the linked lists for each core, that in turn enabled communications with the blocking `RCCE` calls in a proper manner.

Figure 4.1 presents measurements for various grid sizes, concerning communication and computation CPU times, for fixed $\sigma = 10$. This figure concerns the InfOli simulator before any communication related optimizations.

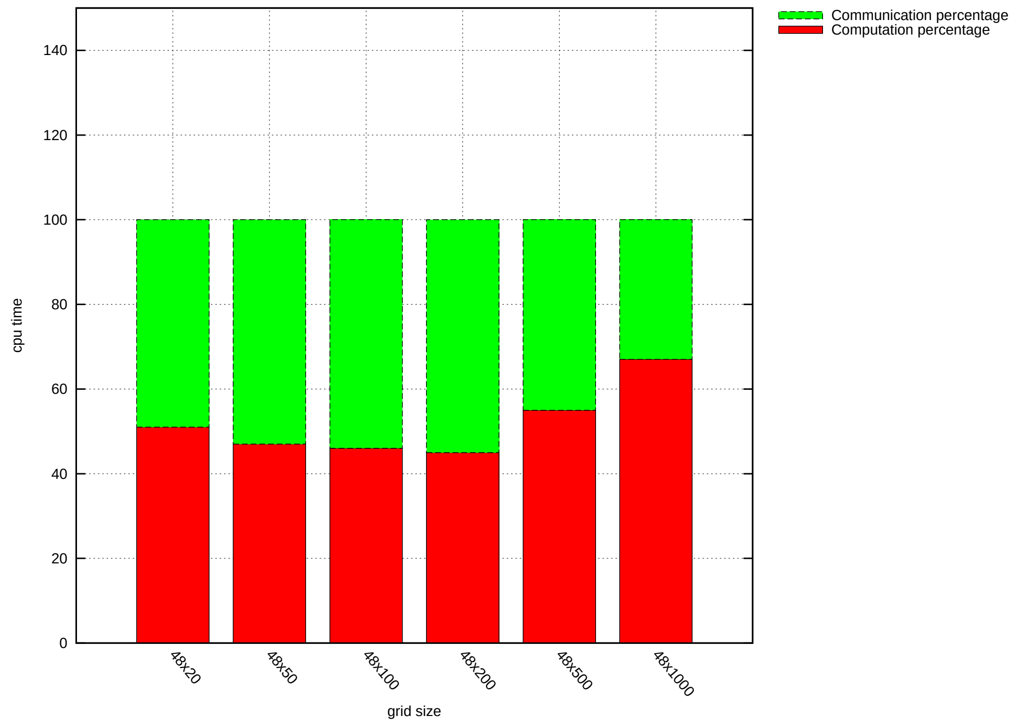

FIGURE 4.1: Communication overhead for sample grid sizes

We measured time with the UNIX function `gettimeoftheday()`, before and after each `RCCE_send()` and `RCCE_receive()`. It is important to note that communication times were not measured while performing dynamic voltage frequency scaling (dvfs) for the application on the fly, since `gettimeoftheday()` would not work correctly in that case. For this reason, we kept voltage and frequency constant throughout our simulation, and we presented the normalized values in our figure.

We can see that 50% of the cpu time is spent for the communication involved through interacting cells for small network sizes. While we increase the network size total communicatoin overhead seems to decrease. This is happening because the max value of the average connections per cell is achieved for different σ for various network sizes, thus the point of highest communication cost is not the same while we sweep the network size. For example, if we measured communication time for large network sizes with higher σ values , we would see that this would still acount for the 50% and more of the simulation time.

Communication costs are high even though the high speed interconnection network of the SCC. That indicates that more elaborate communication schemes should be created, using platform low-level mechanisms, and specifically the Message Passing Buffer (MPB). It is true that fine tuned parallelisation can be achieved in platforms like this, which are highly experimental.

In reality, communication overhead could be even higher than the one illustrated in our figure. In the way communication scheme was original implemented in InfOli, in order for blocking communications to work without deadlocks, cores communicated from higher to lower core ids.

For example, suppose core with id 0, wants to send a voltage to cores with ids 1,2,3. Core with id 0 will execute `RCCE_send(voltage)` for every other core and the other cores will execute `RCCE_receive(voltage)` from core with id 0. We see here that cores 2 and 3 will wait for core 0 to send the voltage to core 1 and so on. Supposing asynchronous communication between cores, core 0 would send asynchronously the voltages to the other cores, and would proceed to the computation phase.

4.2 Collective allocation programming model

As mentioned in Chapter 2, in order to have full control and utilization of the platform developers can use `RCCE_gory` mode of the `RCCE` library. In addition, in Chapter 2 the on-tile Message Passing Buffers (MPB) were described in detail. Functions of `RCCE_gory` use the MPBs in order to convey messages asyncrhonously from one core to another.

Ideally we would like cores to communicate in an asyncrhonous non-blocking manner. This way subsets of cores communicating only with each other will be able to proceed further to the simulation, since no cell voltage is exchanged. Each SCC core has an on-chip RAM, the MPB. All the MPBs in the SCC platform are accessible for read and write operations from every SCC core. Though all cores can access the MPBs, only the core on which the MPB resides can allocate memory on them for use. This implies that before the use of memory allocations the ”owner core” should perform the proper allocations, resulting in an allocation phase before every simulation. All MPBs are mapped in the virtual memory of processes running on the cores, with the use of the translation that is performed with the Mesh Interface Unit (MIU), which happens before a memory request leaves the tile. Requests concernning a memory operation for a specific MPB will be routed through the mesh interface to the tile in which the MPB resides.

Assume that a core allocates memory on its MPB for others core to use. The question arrises : *How will the other cores know the start and end of the allocated addreses in order to use them*? The SCC developers solved this obstacle by assuming collective memory allocation calls, through the symmetric name space of the MPBs. In this way, programming model assumes that each `RCCE_malloc()` function call will be performed collectively by all SCC cores. For example, if i-th SCC core allocates memory of 1 byte in its MPB, all cores in the platform will allocate 1 byte memory in their MPBs accordingly, and that applies for every SCC core.

In this way, cores will be aware of memory boundaries across the MPBs, without having to explicitely transmit them to the cores participating in the communication. This is happening since all the cores allocate the same objects at the same order in time, thus they receive the same absolute memory offsets in their MPBs. SCC developers claim that `gory` and `non-gory` versions of the libraries should work together, but throughout our experience with the board we could not achieve the above.

For example, assume j-core wants to transmit object A to i-core. Both cores would perform the `RCCE_malloc( sizeof(A) )` function call and they would receive the same start and end memory relative offsets for this object in their MPBs, since no prior memory allocation has been performed. Now j-core knows the absolute offset of the memory that i-core has allocated in its MPB, and it can explicitely write to it the value of A.

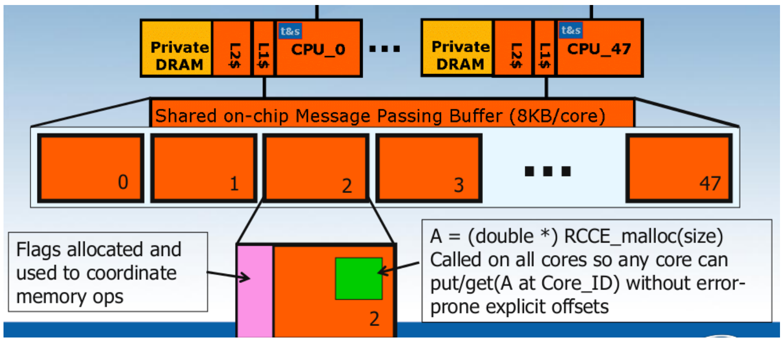
Private DRAM
L2$
L1$
t&s
CPU_0
Private DRAM
L2$
L1$
t&s
CPU_47
Shared on-chip Message Passing Buffer (8KB/core)
0
1
2
3
47
Flags allocated and used to coordinate memory ops
2
A = (double *) RCCE_malloc(size) Called on all cores so any core can put/get(A at Core_ID) without error-prone explicit offsets

FIGURE 4.2: Collective allocation calls from [1]

The use of this model is reflected in the functions of the `RCCE_gory` library, `RCCE_get()` and `RCCE_put()` which we use in our work. Internally those functions contain the following line :

```
final_address = target_address - RCCE_comm[x] + RCCE_comm[j] ,
```

where `RCCE_comm[48]` is a table holding the start addresses of all the MPBs, `RCCE_comm[i]` holds the start address of the MPB of i-core. Variable `x` is the id of the core performing the call and j is the id of the core that holds the target MPB. This line of code translates a local MPB address to a remote MPB address, with the same absolute offset value in the buffers.

It is obvious that this collective model results in *non-optimal* communication schemes in terms of memory footprint. The reason is that all cores must perform allocations, even though they are not necessarily participating in the communication.

4.3 Non-collective allocation communication model

In our work we avoid the use of collective memory allocations. The reason behind this is that we want to maximize the utilization of the platform targeting our application. We would like sets of communicating cores to progress independently from other disjoint sets of cores, with on-tile memory resources not shared between them.

4.3.1 Initilization phase

Consider i-core which communicates with a subset S of the SCC cores. The nature of communication comes from the formed gap junctions between InfOli cells, thus communication between cores in this set with i-core will be bidirectional. SCC i-core will split its MPB in equal slots for every communicating core, with length L bytes. Each j-core, j ∈ S will put data in this slot located in i-core's MPB, and i-core will eventually read them at some point *asyncrhonously*.

The same operation will be performed for every core in the platform. The smaller the number of communicating cores in S the bigger the slots in the MPBs that will be allocated for them, which results in high throughput communication.

The slot of each core will be splitted in `struct message` subslots, and will have a specific number of subslots, specifically :

$$length = \frac{L\ bytes}{\texttt{sizeof (struct message)}} \tag{4.1}$$

`Struct message` will be defined later on this chapter, and its fields will be explained in detail. Cores will transmit struct messages to other cores in order to exchange the appropriate voltages.

Since we now broke the collective model and i-core allocates memory independently for communicating cores in S, *how will those cores know the exact possition of their slots in other cores MPBs* ?

Each core will write to a file located in the mounted filesystem at the `shared` directory which is accessible by all the cores. This file will consist of 48 lines, one for each platform core. If i-core is not communicating with j-core, then the according lines will be empty in both files concerning those cores, otherwise they will contain the absolute start address of the memory in the MPB and the length of the MPB slot *length*. For example assume core with id = 5 needs to communicate with core with id = 6. In 5-th core's file, at the sixth line, will be stored the offset reserved for the 6-th core as well as the length of the slot. This will be the same in the 6-th core's file, with the difference that information for 5-th core will be stored in 5-th line. All cores will read from the filesystem the files concerning communicating cores and determine their slots in the platform MPBs.

Suppose that we have four cores participating in the simulation, cores 0,1,2,3. Connectivity file indicates that core 0 communicates with cores 1,2,3. Core 1 communicates with 0,2 and 3, core 2 communicates with 0 and 1, and finally core 3 communicates with 0 and 1.

The initilization phase of those cores would be the following :

Initialization phase

- Before the initilization phase, the MPBs are empty and without partitions. This is illustrated in Figure 4.3. We can see that the MPBs are empty and no memory allocation has been performed at this point.
- Each core determines communicating cores through parsing the connectivity file. This information is used afterwards in order to allocate the slots.
- Allocate memory for communicating cores with `RCCE_malloc()`. Figure 4.4 illustrates this step. We can see that the MPBs are partitioned in slots according to the number of the communicating cores, for each core. We should mention here that slots need not to be consecutive memory segments, as shown in the figure.
- Write to a special file the memory addresses and the length of the slots in the MPB.

For example a snapshot of the file created by 0-th core would be the following:

```

              Address for slot—1    length of slot
              Address for slot—2    length of slot
              Address for slot—3    length of slot

```

The file created by 3th-core would contain:

```
              Address for slot—0    length of slot
              Address for slot—1    length of slot

```

This step also is illustrated in 4.4, where we see the transmission of the appropriate values through the MCPC.

- Barrier for all SCC cores. We should wait until all cores allocate and transmit the information through the `shared` directory.
- Read absolute offsets for slots created in other MPBs for this core.
- Store adresses relative to the own core MPB, in order for `RCCE_put()` and `RCCE_get()` to work correctly.
- Barrier for all SCC cores. Wait until all cores finish the initilization phase before procceeding to the simulation steps.

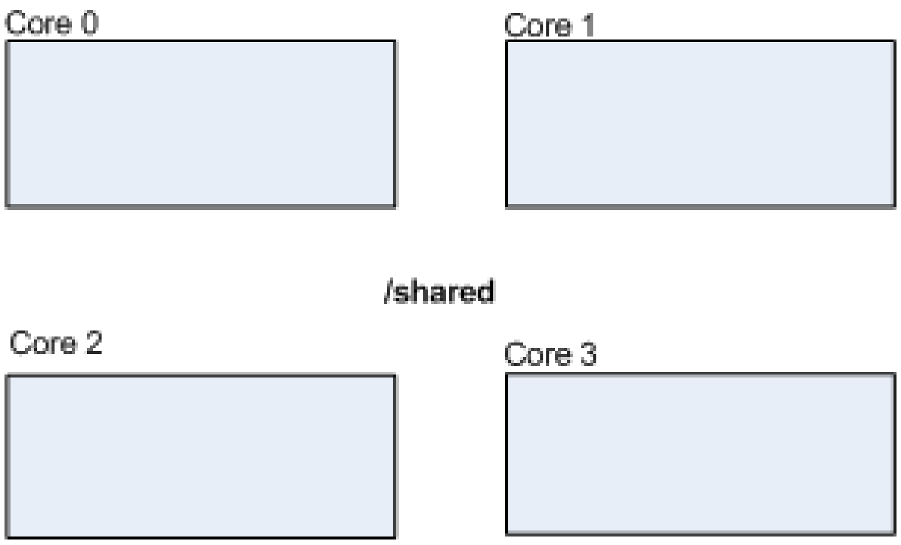

FIGURE 4.3: MPBs prior to any memory allocation

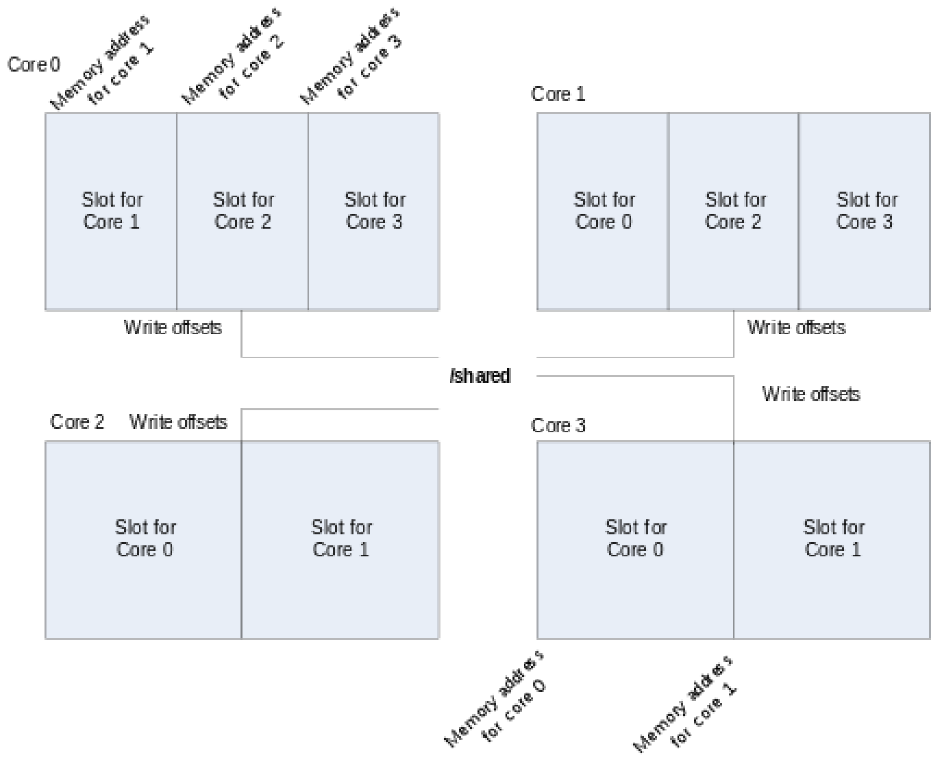

FIGURE 4.4: MPBs after the proper memory allocations

4.3.2 Necessary data structures

We examine here the basic structures that play major role in the InfOli communication :

```
struct message {
        bool flag;
        int cell;
        mod_prec voltage;
}
struct core_packet {

        int bidirectional_cells;
        int receive_only_cells;
        int send_only_cells;

        int offset;

        int counter_B_send;
        int counter_B_receive;
        int counter_R;
        int counter_S;

        int communicating_core_slot_length;

        volatile message *same_core_slot_address;
        int same_core_slot_length;

        couple_cells *bidirectional_list;
        couple_cells *receive_list;
        couple_cells *send_list;

}
```

Struct message is the basic message that cores transmit to each other in order to exchange voltages. `int cell` represents the cell to the receiver core and `mod_prec voltage` is the voltage of a neighbour of this cell. We must mention here that receiver cores do not concern with the cell that transmits the voltage (the neighbour) since we are only concerned with the difference in the voltage between two neighbouring cells.

`Bool flag` is a field that makes sense only for the first subslot of the slot reserved in an MPB. `Bool flag = false` indicates that data in the MPB slot have been read by the owner core, and the slot is available for use. `Bool flag = true` indicates that the owner of the MPB slot has not yet read all the data in the slot, so the core that wants to write to it must wait. Since only the fist subslot is used for controlling the access to the slot, `Bool flag` is concerned only for this subslot.

Since each core splits its MPB in slots for the other communicating cores, every MPB slot is a pool of voltages that foreign cores(producers) write in, and the MPB owner (consumer) read those votlages at some point. This results in an one-to-one producer-consumer scheme, and thus synchronization is not needed between the consumer and the producer.

Struct core_packet is the structure holding all the logistic information concerning information with other SCC cores. Each core maintains an array of pointers `to core_packet structs`. If an array element has the `NULL` value then no communication is performed with this core. For example if in i-core `core_packet[j] == NULL` then no communication is performed with j-core. `Bidirectional_cells, receive_only_cells and send_only_cells` variables hold the number of different connections with cells simulated by the other core.

The counter variables count how many cells have been transmitted with the other core at the current communication phase of the simulation step. `Couple_cells` lists hold the specific cells that must be sent or received by this core. The struct also holds information regarding the MPB address of the slot reserved for the communicating core, as well as the slot reserved for this core in the foreign's core MPB.

4.3.3 Communication phase

Having defined the above the following communication scheme emerges :

Consider i-core which communicates with a subset S of the platform cores. Platform's i-core will repeat the following process until all communication is finished for the simulation step.

```
Communication algorithm executed in every simulation step , for every core

communication phase
  For every j in [0..47] :
        if communication with j is not marked as finished:
                Try to read data from j—core , lcoated in i—core's  MPB.
                Try to write data to the slot reserved in the MPB of j—core.
                If i—core has no more voltages to send or receive from j—core , mark communication as finiished.
                Proceed to examine communication with next core in the list.
end procedure
```

The above algorithm uses information contained in `struct core_packet`. We should note here the asynchronous nature of the communication. A core desiring to put data in a full MPB slot owned by another core, will not block waiting for the previous data to be read. Instead it will try to write to other MPB slots, concerning other communicating cores, or read from its own slots and eventually will try again to write data to the previous slot, when the loop above reaches again the desired core.

The above communication implementation is both asynchronous and blocking-free. In addition no deadlock can be achieved since we have a one-to-one consumer-producer scheme. This is happening because every reader updates `Bool flag` only when it has aquired all the data from the MPB slot, and every writer only after it has written all the desired data in the core.

The proof of correctness is performed by running the InfOli simulator and comparing results of axon potentials at every step of the simulation with results of previous versions of the source code.

Figure 4.5 illustrates the communication phase between the SCC cores.

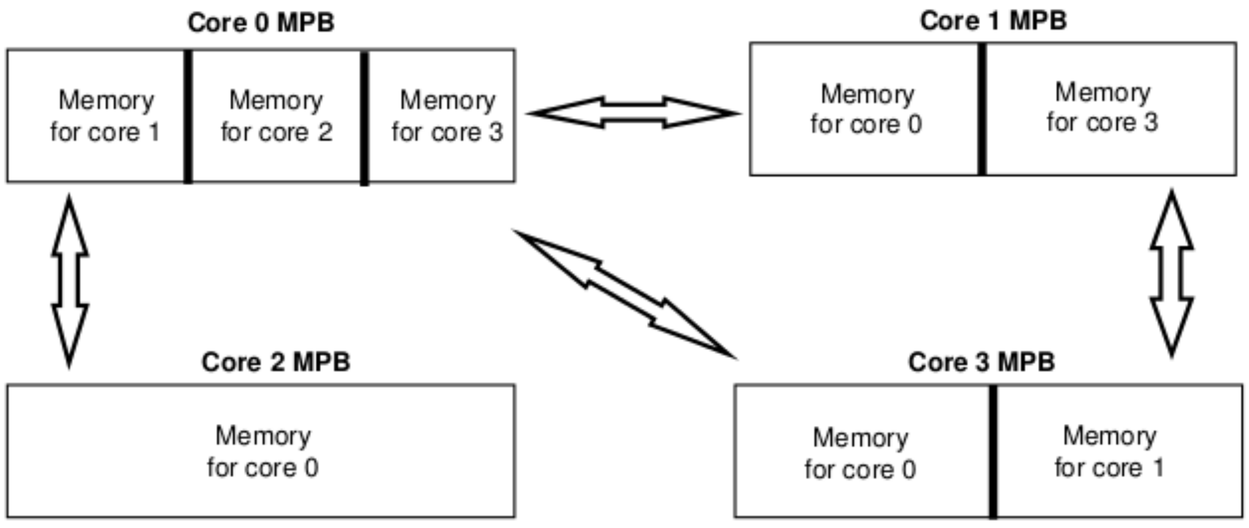

FIGURE 4.5: Communication phase

4.4 Experimental Setup Description

The frequency of all cores throughout the board was kept at 800MHz, while the voltage was kept at 1.1 V. We were not interested for on the fly adjustement of those parameters, because we would like to measure the performance gain of our implementation.

Simulations for the InfOli network have been performed for a variety of the parameters involved : σ and grid size. The average of the Normal distribution (which models neuron inter-connectivity) was kept in zero for all the simulations. We increased the size of those parameters with a logarithmic sweep. We performed experiments with every possible combination deriving from the following values and created respected figures for the measurements involved.

Simulations for higher cell inventories were performed, however energy gains as well cpu speedups are presented for network size up to 48000. This happens because simulations higher than that point are very time demanding.

We measured CPU time of the total simulation and the energy consumption of the SCC board during the simulations. CPU time was measured with the linux program `time`, which measured the platform occupation throughout the simulation. The command line program `time` was executed in the MCPC platformm, that initiated the simulation, through proper scripts provided by Intel developers. In order to measure energy consumption, we initiated a thread reading appropriate values from configuration files, at specific time intervals, throughout the whole simulation.

The management of the threads and the initiation of the simulation was achieved through Python scripts. The power monitoring thread was not terminated exactly after the end of the simulation, rather than kept monitoring for 5 more seconds.

Figure 4.6 demonstrates a power pulse. We should note that we measured the power during the initilization and de-initilization of the cores that would run each simulation.

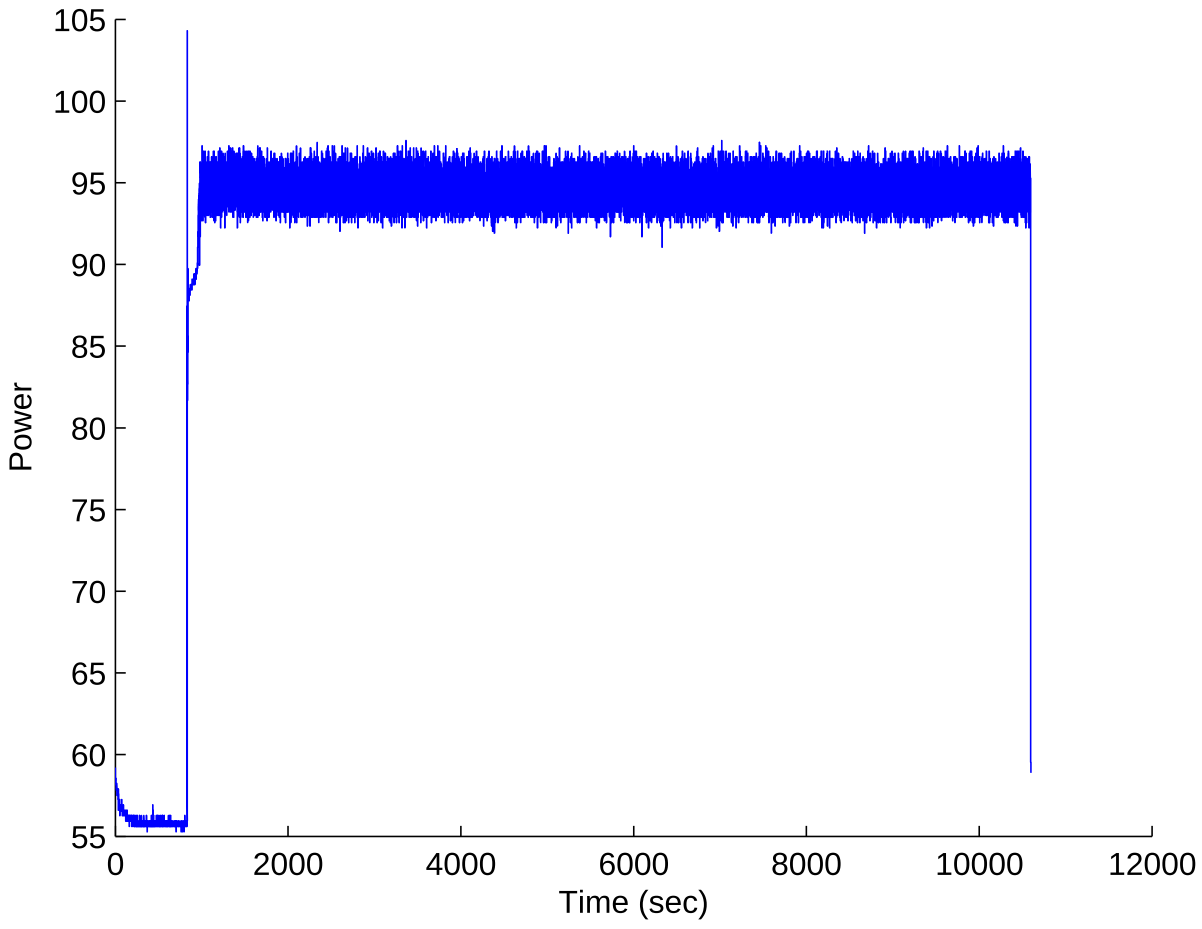

Figure 4.6: Power pulse for a simulation involved in Infoli

The total energy of the simulation is calculated as the integral of the power pulse regardng the total simulation time.

4.5 Results and Discussion

Plots have been created regarding the measurements involved. Those figures are presented below :

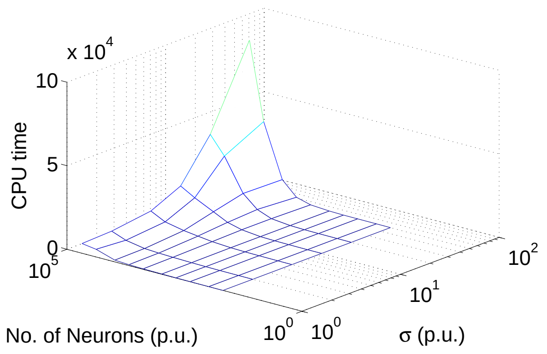

Figure 4.7: CPU times before the communication optimization

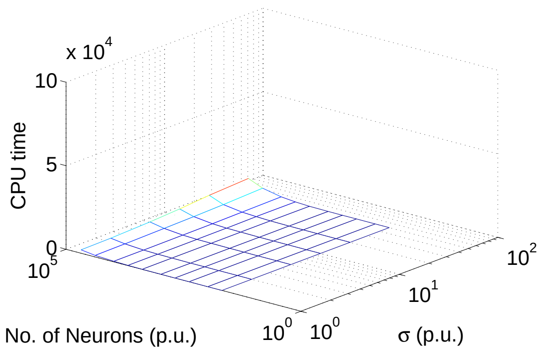

Figure 4.8: CPU times after the communication optimization

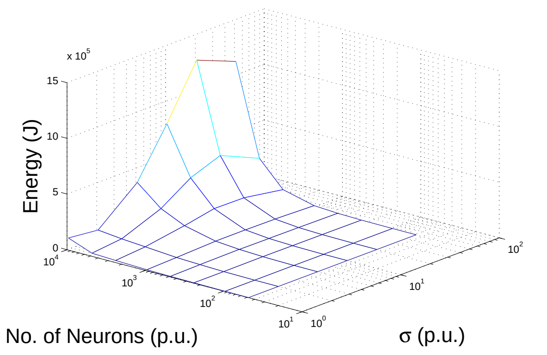

Figure 4.9: Energy consumption before the communication optimization

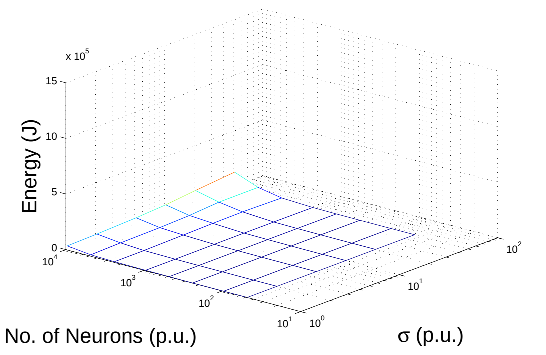

Figure 4.10: Energy consumption after the communication optimization

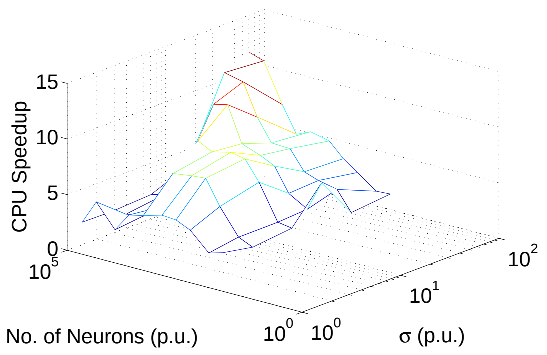

Figure 4.11: CPU speedup

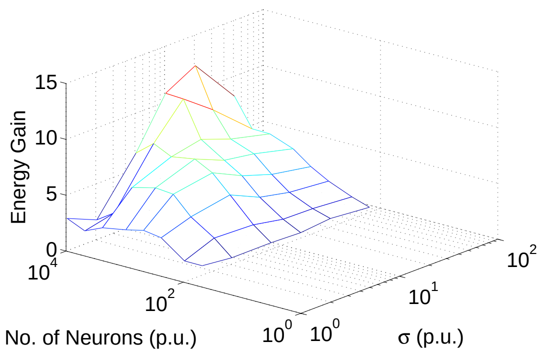

Figure 4.12: Total Energy Gain

The above figures represent the absolute values of the measurements involved, as well the gains of communication optimizations for those measurements. Figures 4.7 and 4.8 illustrate overall simulation times before and after the optimizations. Figures 4.9 and 4.10 illustrate the total energy consumption before and after the optimization. We can see at those figures that CPU time and total energy dropped significantly. We impose the same scale on the Z axis on those figures, in order to best compare the experiments.

However, in order to have a clearer view of the benefits of communication optimizations in our work, we created Figures 4.11 and 4.12. Equation 4.2 and 4.3 define the metrics used:

$$CPU\ speedup\ =\ \frac{CPU\ before}{CPU\ after} \tag{4.2}$$

$$Energy\ Gain\ =\ \frac{Energy\ before}{Energy\ after} \tag{4.3}$$

The values before our implementation refer to previous parallel implementations of InfOli simulator. The performance gains of our implementation are obvious. The maximun cpu speedup achieved is 11, while the maximun energy gain is near to 15. Those are the absolute values of our performance improvements. The interesting fact here is that in 4.11, speedup is increasing with network size. From that we deduce that the communication scheme performs better for bigger network sizes. That is happening due to the increase in the average connections per cell for each size. More connections introduce bulk loading in the communication scheme, so we observe higher speedups in comparison to previous code versions.

The total energy is also very important. In long lasting simulations like those performed, energy is a major simulation factor. While the platform is not of an HPC nature, since all cores reside on the same chip, energy gains should be noted. One major benefit is that we reduced the total use of the platform. Core failures often happen on many-cores chips, and by conducting the same simulations in less CPU time, we decrease the risk of platform failure.

Chapter 5

Thesis Conclusion

5.1 General Remarks

The current thesis aims at further accelerating very time consuming applications, such as the InfOli simulator. We could see how that was made possible, through the understanding and utilization of the SCC platform. Indeed, the InfOli application was accelerated, and bigger neuron simulations were conducted in less time, reducing the platform usage, and thus reducing the risk of platform failures. Core failure on experimental platforms like the SCC is a major consideration, since cores can not be replaced, beacuse they reside on the same chip.

The fact that the SCC is a highly experimental platform, was reflected on the use of it. In fact, SCC never meant to be an original product, in contrast with the next many-core Intel platform, the Xeon Phi co-processor. This meant more difficult application porting on the platform, since many tools were not available. Another remark that must be made is that the platform's hardware is out of date, in contrast with a modern desktop.

However, the experimental nature of the platform, enabled the SCC designers to include unique characteristics, like the MPBs, which were a hybrid implementation of shared and distributed memory. This gave us the chance to further experiment with communication schemes, and finally accelerate the application. We should note here that the SCC designers choice for uniform `RCCE_malloc()` calls should have been avoided, as it is constraining for the developer, and is resource demanding. The designers should provide non-uniform allocation calls for the distributed memory, as well a mechanism for cores to determine the memory boundaries in the MPBs.

Though the SCC enabled us to perform large-scale simulations, we should make clear that it is not an HPC platform in any way. HPC platforms demand higher communication costs, since computing units are inter-connected through typical networks, in contrast with the SCC, where computing units reside on the same chip. For applications like InfOli, an HPC platform may be a better fit.

The uniformity of the SCC cores indicates that the platform does not fit applications that require load unballancing between the computing units involved. InfOli portings that involved unbalanced loads between SCC cores, like the one presented in [3], proved to be the most inefficient. Also voltage and frequency adjustment on the fly, seemed to hamper the simulation performance. Indeed, applications involving load inballance do not fit the SCC board and thus porting them reduces the gains of parallelization.

5.2 Future Work

As for future work, an interesting approach would be to develop an rcce-style library for asynchronous communication, that would exploit the MPB buffers in the most efficient way. This way applications would take full advantage of the MPBs, and developers would achieve near to optimal solutions, without being concerned for the underlying mechanisms of the platform. That would drastically reduce the time-to-solution, which is a factor of paramount importance, concerning parallel applications.

Regarding the InfOli application, an interesting scenario would be to explore different connectivity schemes, like the one described in this thesis. Those schemes could be a result of different probability distributions, or could derive from brain-related research that would shed light on the actual nature of the neurons. Simulation times and energy consumption could be measured, while changing on the fly the neuron connections, as well the impact of those changes on the overall simulation. In addition, those changes could be triggered by an external event, like some input from other brain parts, or an event on the enviroment of the subject. The development of combined models that would interact between them and dynamically adjust their behaviour would be very interesting.

In this thesis, we developed an asynchronous communication scheme on top of distributed memory buffers, that were accessed through a common address space by the application running on the cores. Another approach would be to test models like the above, in pure shared memory architectures, where the memory access is uniform for sets of cores, or all the platform cores.

Finally, since the InfOli applications is very time-demanding, a way forward would be to port it on an pure HPC platform, like a cluster. This way a bigger neuron network could be simulated, for longer brain time, and would provide more usefull input for neuro scientists towards the understanding of the brain functionallity. In addition the scaling of the InfOli application could be studied, and new communication schemes could emerge from these results.

Bibliography

[1] "The scc platform overview, revision 0.75," Intel Labs, Tech. Rep., 2010, September 1.

[2] "Olivocerebellar hardware modelling," Erasmus MC, Neurasmus, Tech. Rep.

[3] *Optimal Mapping of the Inferior Olive Neuron Simulations on the Single-Chip Cloud Computer*, 2014.

[4] H. Markram, "The blue brain project," EPFL, Tech. Rep.

[5] D. S. V.Shah, *Diseases of the Brain and the Neurous System*.

[6] "Consensus paper: Roles of the cerebellum in motor control—the diversity of ideas on cerebellar involvement in movement," 13 December 2011.

[7] D. Z. CI, S. JI, H. CC, G. N, K. SK, and R. TJ., "Microcircuitry and function of the inferior olive," *Trends in Neuroscience*, 1998.

[8] P. Dayan, "Levels of analysis in neural modeling," University of College London, Tech. Rep., 2001.

[9] T. J. Senjowski, "Computational methods," Salk institute for BIological Studies, Tech. Rep., 2009.

[10] A.L.Hodgkin and A.F.Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, 1952.

[11] H. S. Seung and H. Hughes, "Learning in spiking neural networks by reinforcement of stochastic synaptic transmission," *Neuron*, 2003.

[12] *Live Demo: Spiking ratSLAM: Rat hippocampus cells in spiking neural hardware*, 2012.

[13] *The Application Research of Neural Network in Embedded Intelligent Detection*, 2011.

[14] G. Stitt, "Are field-programmable gate arrays ready for the mainstream?" *Micro, IEEE (Volume:31 , Issue: 6)*, 2011.

[15] "Accelerating high-performance computing with fpgas," ALTERA, Tech. Rep., 2010.

[16] *Neural networks on FPGAs: a survey*, 2000.

[17] *GPU versus FPGA for high productivity computing*.

[18] *Effect of Instruction Fetch and Memory Scheduling on GPU Performance*, 2010.

[19] A. Morrison, C. Mehring, T. Geisel, A. Aertsen, and M. Diesmann, "Advancing the boundaries of high-connectivity network simulation with distributed computing," 2005.

[20] "Largest neuronal network simulation to date achieved using japanese supercomputer."

[21] J. M. John Shalf, Sudip Dosanjh, "Exascale computer technology challenges," Lawrence Berkeley National Laborator, Tech. Rep., 2011.

[22] *Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization*, ser. HPCA '13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2013.6522332

[23] A. Duran, "The intel many core integrated archtecture," Intel corp., Tech. Rep., 2010.

[24] T. W.Berger and D. L.Glanzman, "Toward replacement parts for the brain," Massachusetts Institute of Technology, Tech. Rep., 2005.

[25] "http://www.livescience.com/41808-neural-prosthesis-bypasses-destroyed-area.html."

[26] N. Francisco, L. N. R., G. J. A., C. R. R., and R. Eduardo, "Cpu-gpu hybrid platform for efficient spiking neural-network simulation," *BMC Neuroscience;2013, Vol. 14 Issue Suppl 1, p1*, 2013.

[27] E. Ros, E. M. Ortigosa, R. Ag R. R. Carrillo, and M. Arnold, “Real-time computing platform for spiking neurons (rt-spike),” *IEEE Transactions on Neural Networks*, pp. 1050–1063, 2006.

[28] “Brain research eu funding.”

[29] “Teraflop research chip overview.”

[30] R. Rotta, T. Prescher, J. Traue, and J. örg Nolte, “In-memory communication mechanisms for many-cores – experiences with the intel scc.”

[31] *Recent Advances and Future Prospects in iRCCE and SCCMPICH*, 2011.

[32] “Rcce:a small library for many-core communication,” Intel corp., Tech. Rep., 2010.

[33] G. Chantzicwnstantis, “Energy aware mapping of a biologically accurate inferior olive cell model on the single-chip cloud computer,” Master’s thesis, NTUA, 2013.

[34] *Amdahl’s Law in the Multicore Era*, 2007.

[35] *Effect of lateral connections on the accuracy of the population code for a network of spiking neurons*, 2001.