1 Introduction

1.1 Aim

The aim of this project is to solve the Conjugate Gradient Method[**?**] on a SpiNNaker chip, thus using the massive parallelism that this architecture provides to reduce the time complexity of the aforementioned algorithm. This is accomplished by reducing the time complexity of the most expensive operations of the algorithm which are matrix-vector multiplication and scalar products between vectors. The complexity is reduced dramatically, due to the abundant number of cores provided from the architecture. This report provides a detailed analysis of this project and its constitutes, the background research done to launch this project, along with design and implementation choices.

1.2 Reasons and Justification

SpiNNaker is an architecture inspired by the biology of the human brain. Its optimal configuration has over a million cores[**?**]. This architecture has mainly been used to simulate the neurons of the human brain and in some cases, robotics.

However little work had been done into putting the SpiNNaker architecture to solve classic computer science problems. That is why a problem such as the Conjugate Gradient Method has been proposed, which is a very common solution to optimization problems. In addition to that, the SpiNNaker architecture offers new parallel programming paradigms, that escape some common parallel programming pitfalls such as race conditions, deadlocks, mutual exclusion etc[**?**].

1.3 Overview

This report starts off by giving a detailed analysis of the SpiNNaker architecture and its constitutes, and the Conjugate Gradient Method. It continues by outlining the design and the tools used to solve the given problem and provides an in-depth examination of the implementation issues and solutions. It involves a short description of the testing that was completed and it ends with a critical evaluation and conclusion.

2 Background

2.1 The neuron

To make the explanation of the SpiNNaker architecture smoother, the design from which the SpiNNaker chip was inspired will be outlined. This is no other than the human neuron.

The human neuron is an electrically excitable cell that processes and transmits information through electrical and chemical signals. Its basic constitutes are the the soma, the dendrites and the axon. The soma is the body of the neuron. A dendrite receives signals from the soma of the neuron that it belongs to, or other neurons. It extends for hundreds of micrometers and branches multiple times throughout the body, thus forming a dendritic tree, which connects with other neurons' axons. The axon is used to transmit signals to other neurons and it extends from the soma of the neuron to a dendrite. All human neurons have only one axon. Given the above analysis the dendrites could be described as the inputs of a neuron.

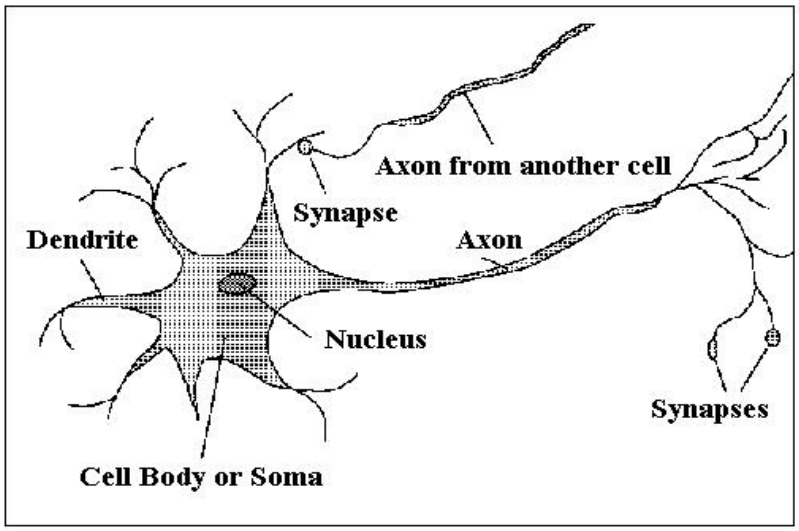

Figure 1: A neuron

One of the most important parts of the neuron structure is the synapse. The synapse is where the contact between the axon of a neuron and the dendrite or

the soma of another neuron happens. It is where information from one neuron is transmitted to the other. When a set of neurons are connected with each other through synapses they create a neural network.

Finally, the communication between neurons is accomplished through spikes, which are either chemical or electrical[**?**].

Given the terms in this section, the name of the SpiNNaker chip is deductible. It stands for Spi(king)N(eural)N(etwork) architecture.

2.2 The SpiNNaker architecture

As mentioned before the SpiNNaker architecture is inspired by the biology of the human brain, and more specifically, neurons. However, its architecture is not constrained by the biology of the brain, but many techniques to speed up computation are used. It differs from other supercomputers, which usually have a number of strong processors with slow network capabilities. The design of the architecture was made with two concerns as its main priority. The first one being MIPS(millions of instructions per-second) per mm^2. Namely how many instructions can be performed in an area of silicon. The second one was MIPS per watt, which means how many instructions per second can be performed given a fixed amount of energy. Its most optimal configuration will use a million cores and will be able to simulate over a billion neurons[**?**].

2.2.1 SpiNNaker chip Overview

The heart of the SpiNNaker architecture is the SpiNNaker chip. Its main components are 20 identical ARM cores that run at around 100MHz each, a router, a system NoC(Network-on-Chip) which connects to the router of the chip and a 128 MB SDRAM. At start up, in each chip a processor is selected to act as a Monitor processor, with the task of performing the management of the chip's system. The remaining ARM cores perform the calculations for the program, with the exception of one which is reserved as a spare. The spare processor exists in the emergency of another core having a malfunction, so in the end 18 cores are available for computation in each chip[**?**]. Each core has the ability to hold 32KB of instructions and 64KB of data. Since this is not enough space, all other data is saved in the SDRAM[**?**].

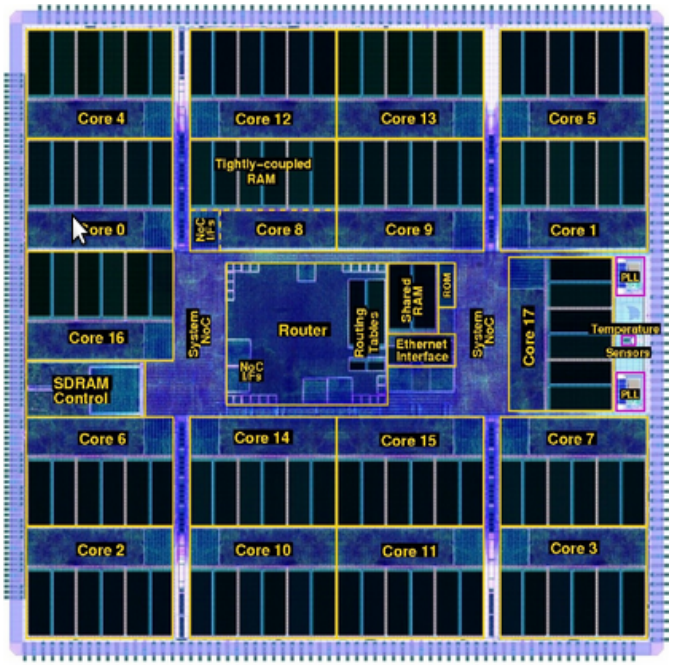

Figure 2: The SpiNNaker chip[**?**]

2.2.2 The router

As mentioned before the architecture supports low-delay communication, with less powerful processors, than those used in most supercomputers. To accomplish that, the system uses asynchronous multicast packets to communicate between cores. This is done through the router which lies in each chip.

The router exists as the part of the NoC(Network-on-Chip) and its primary role is to direct packets that arrive and packets that are sent or need to be forwarded to other neighbouring chips. The router has 20 ports for the ARM cores that are located in the chip and is able to forward one packet at a time. The router works faster than a transmission port, which results into the router being most of the time lightly used. It is designed to support point-to-point communication, using small packets. The use of multicast packets helps to reduce the amount of packets that exist in the network at any given moment. To even further enhance the quickness of the network, the architecture supports default routing. This means that not all connections need to sit in the routing tables, in order to have the packets forwarded to their eventual destination. This concept will be explained in more depth in the next section.

The functionality of the router is pretty simple. When a packet arrives from

the input port, then the router will try to send it to an output port. If there is a problem during transmission, then the router will keep trying to send it and after a while it will try the emergency route(also explained in the next section). If the emergency route still does not work and a certain amount of time has passed, then the packet will be dropped. To implement the time feature, the time of delivery of a packet is stored in the header of each packet.

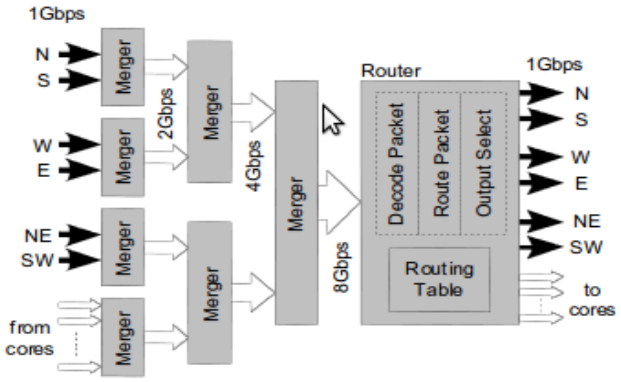

Figure 3: The router[**?**]

To further understand these concepts, the topology of the system needs to be considered.

2.2.3 Topology

In order to have a million cores available for processing a number of SpiNNaker chips need to be connected and work efficiently as a whole. To reach that goal an effective topology is important.

Considering Figure 4, it is visible that the topology is viewed by the system as a 2D mesh, with chips having a fixed amount of neighbours and specific base connections. Any chip is able to communicate directly to 6 other chips. This is also depicted in Figure 3, where the positions of these chips are shown. In general each chip can directly communicate with chips that are located North, South, West, East ,Northeast, and Southwest of its respective position. This however does not apply for chips that lie in the perimeter of the mesh, since in some of the positions described above, some neighbouring chips will not exist[**?**].

Returning now to the default routing mentioned in the previous section, if a packet arrives to a router, and the router does not have an entry for it, then it

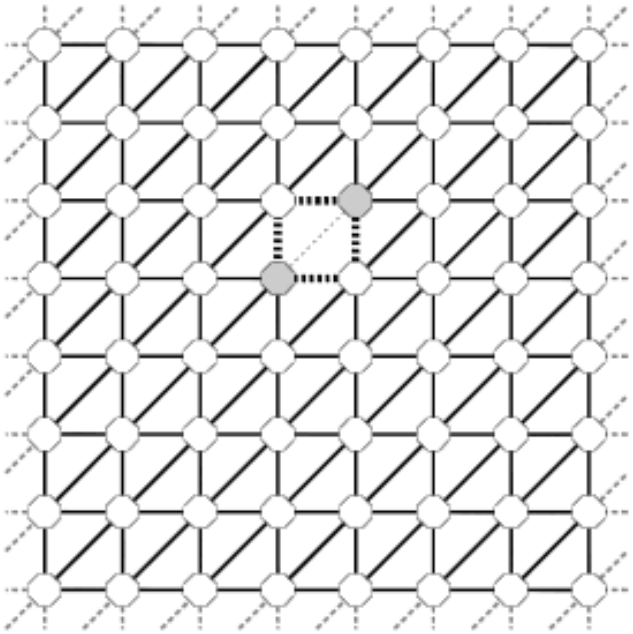

Figure 4: *The SpiNNaker topology[**?**]. This is an example of what an 8x8 board would look like.Each circle is a SpiNNaker chip and the lines between them are the connectivity.*

will be forwarded to the chip opposite to the one it came from. For example, if a packet comes from the East and the aforementioned rule is applied, then it will be forwarded to the West. This helps reduce the size of the routing tables in each chip[**?**].

In addition to the default routing, the topology also provides two-hop routes among neighbor chips, as shown in Figure 4. These routes are named emergency routes and their functionality is to bypass any faulty connections that might exist between connections[**?**][**?**].

2.2.4 Packets

The packets(spikes) used in the SpiNNaker architecture are all multicast. They are asynchronous and relatively small. Having multicast packets means that one packet might have many different destinations, and with potentially a million cores, they can have thousands of destinations. For that reason, the packets do not contain the destination(s) they are supposed to reach, but rather only the source. The transmission of the packets is done completely by the hardware of the system, thus achieving high bandwidth[**?**][**?**]. It is also notable that,

when simulating a billion neurons, unique addresses are needed to acknowledge each neuron. Hence 32-bit addresses are used to represent different elements in the cores. In this architecture, the main packets used can be divided in two types.

- Multicast packets(Type 0). Consists of 32-bit source address, 8 bits of control and 32-bit payload. These types of packets are delivered to processors and would usually contain information for computation
- Point-to-point packets(Type 1). Consists of 16-bit source *chip* address, 16-bit target *chip* address, 8-bit control payload and 32-bit payload. These types of packets are command/control packets and are used when the Monitor Processor communicates with the target chip[**?**]

In this project Type 0 packets have mainly been used.

2.2.5 Routing Tables

A mention of the routing tables is also important in understanding the architecture. Optimally, one would want for all cores to communicate with all the other cores in the system. However, that would be impractical, since the data structure to hold the entries would have to be enormous.

Instead, this architecture supports routing tables of 1024 entries. Each entry is of 32-bit size. Notice that the size of each entry is the same size of the addresses of elements in cores. The nature of the most significant bit of the source determines if a particular packet is for distributing locally to the cores, or if it should be linked and forwarded to other chips[**?**].

In addition to the routing tables, another trait of the SpiNNaker architecture are the Target Table entries. These tables exist for each element in a core and contain information about that element. They are customizable for every program and they help the programmer since they are readable by the cores during run time.

2.2.6 Bandwidth

Having discussed the architecture and its network, a mention of the bandwidth bisection that is available will be outlined here.

Assuming the optimal configuration of the SpiNNaker architecture with a million cores, then it is safe to assume that the number of chips that will be used,

would be around 63.000(63K). If we would split the topology in two parts(a right and a left one) then the optimal goal would be to have all the neurons of one half connected to at least one neuron in the other half, so as to achieve more connections. That way, the bandwidth in the border would be 6.4G packets/sec. Another safe assumption is that the 2D mesh for this machine would be 256x256.[**?**][**?**].

Allocating the board in the fashion that was just described, the SpiNNaker machine is able to achieve an abundance of connections, with fast travelling packets and high bandwidth.

2.3 Software

There is a lot that comes when trying to run any kind of software in a SpiNNaker board. The basic devices that are used to run a program are three.

- Host Machine. This machine is mainly used for input and output, and it is the component that the user interacts with to start an application
- Monitor Cores. These are the same Monitor cores that were discussed previously. Their main use is monitoring and managing the system and the application cores. One of them also communicates with the host, in order to start up the program.
- Application Cores. These cores are used to carry out the computation of the application.[**?**]

At this point it is noteworthy to mention that the programming model of the machine is event-driven, meaning computations occur when specific events take place. More on that will be covered in later sections

2.3.1 Start up

In order to do the start up and interact with the devices mentioned in the previous section some kind of software needs to be used.

For the Host machine to interact with the main Monitor Core *ybug* is used. The same software is used to start applications. Of course ybug does not communicate only with the hardware of the monitor core, but also with other pieces of software in the system.

There is another software design called *scamp*, that works on top of the Monitor Core. It interacts with the *ybug* and the *sark*, which is a software that is used by the application cores.

Sark lets the user to manipulate some hardware parts of a SpiNNaker chip.[**?**]

2.3.2 Local Data Structures

Before the user program starts running, start up applications need to be run first. These include finding the state of the machine, finding a communications tree for the cores and chips and sketching out the point-to-point tables.

The state of the machine To discover the state of the machine, each chip checks its cores and the connections between them to check whether any broken links exist. The algorithm starts by selecting a root node and marking it with a Request(R) token. Each time a core has an R token it sends an Acknowledge(A) label down to the incoming port and labels it Good. After that it finds all unlabelled ports that are connected to it and sends them an R token while labelling them Broken. If the port that just received the R token does not go through the process that the root node went through it remains Broken and the system knows it will not work. Otherwise it will send an A token and will be deemed Good.[**?**].

Building a communications tree In order to build a communication tree a breadth first tree traversal is used. To achieve that, a root core is selected and given the token Forward(F) at first. The root core communicates with the neighbouring cores and after giving them the label Child it instructs them to also send F tokens to all their neighbouring cores, thus making them both a Parent and a Child. Once there are not any more unlabelled cores, the nodes are instructed to return a Backwards(B) token to their parent nodes. That way the system becomes aware of which nodes communicate with each other. If by any chance two cores send to each other a B token through the same port, then that port is deemed non communicational and is labelled Unused. The reason for that is to avoid any loops in the system[**?**]. Considering that the communication between cores is built as a tree it is deductible that the theoretical complexity of sending a packet from one core to another one is $log_2(N)$. This is the same complexity that a tree has when searching for a node inside its structure. It should also be pointed out that the tree is not saved somewhere in the system, but rather that is how the routing tables and the P2P tables view the connections.

Creating the P2P Tables Cores need ID's in order to have packets sent correctly to certain destinations. Point-to-Point tables are used to list the ID's of nodes, so the packets know which ports they should follow. At first all nodes are undefined(U). If we were to examine one entry for the P2P Table, then the algorithm would go as follows. If a core is undefined and does not have an ID yet, then an incoming port is assigned to it. Then it sends its ID down to all the other ports, except the incoming one, so other cores will know where packets arrive from. If the core has already been given an ID, then all that the core does is send its ID to other cores. If that algorithm is carried out for all the cores in the system, then all the cores in the SpiNNaker board are recognizable[**?**].

2.3.3 Program Loading

In order to begin running a program some input needs to be provided to the system. The way this is currently done is by using the static model.

With the static model the network topology, the connections between nodes and the allocation of the hardware is provided explicitly and externally by the user before the execution of the program. Moreover, data and additional execution information needs to be defined before the program is loaded in the machine. This information is read from an external machine and written again in a fashion that is readable by the machine in the form of binary files. The Host starts the execution of the program, after the readable by the machine version has been loaded to the machine. When the program terminates, or the user wishes to terminate it, then the Host issues a halt command to stop all operations[**?**].

There is an additional model for loading data in the system called the dynamic model, which would enable the machine to figure the network topology by itself, while running the application. However, this kind of a model is still in primary stages[**?**].

2.3.4 Event Handling

As mentioned before the SpiNNaker architecture's programming model is event-driven. For a function to be executed by a an application core, a certain event needs to occur beforehand. In the context of SpiNNaker there are three events that can cause a function to run.

- Delivery of a packet.

- Completion of a DMA transfer.
- After a time interval.

Each of these bullet points qualifies as an event. The programmer cannot control when these specific events will take place(except in the case of the time interval). What the programmer can control is what happens when such an event occurs. The functions, in this context referred to as callbacks, are written by the programmers, which after a specific event occurs, are given a priority with the kernel(sark) in order to be executed.

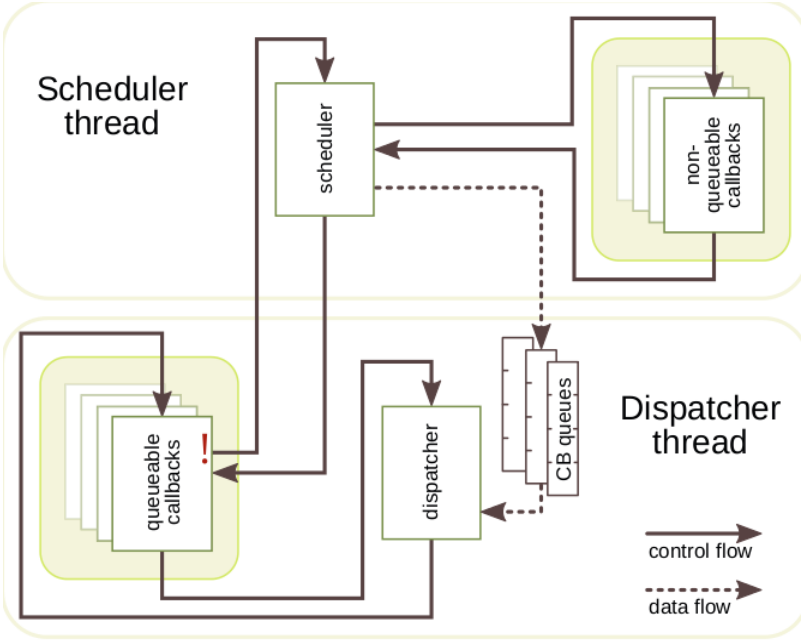

Figure 5: Event-handling. The control and data[**?**].

The kernel is responsible for allocating resources and processors to the tasks that are triggered by the events. Following events, control over the processors is given to the scheduler that queues tasks. When the scheduler finishes it gives processor time to the dispatcher that dequeues tasks and instructs specific cores to execute them. This whole process is depicted in Figure 5.

The types of callbacks can be separated in two categories. Queueable and non-queueable. In the case of a non-queuable callback, the callback is executed immediately by the system, having the top priority of execution. A queueable callback, which is most common, will be put on an execution queue along with other queueable callbacks. In the case that the queue empties, no callbacks will

be executed. It will continue executing callbacks when another event occurs at some point[**?**][**?**].

2.4 Notable work using the SpiNNaker architecture

Most of the notable work done in SpiNNaker machines is simulation of neurons. Various applications would simulate different kinds of neurons[**?**][**?**][**?**][**?**]. One of the most interesting examples is simulating the neurons of the retina, so as to make it possible for a robot to calculate its place in a room[**?**]. Another one would be simulating thousands of spiking neurons using four million synapses[**?**]. Finally, another intriguing application is teaching the neurons temporal sequences of discrete symbols by using a test data set, thus making fast and accurate predictions of new sequences[**?**].

2.5 Conjugate Gradient Method

The Conjugate Gradient Method(CGM) is an algorithm for the numerical solution of systems of linear equations of the type Ax=b; those whose matrix is symmetric and positive definite.

A *symmetric* matrix is a matrix which is equal to its transpose. If A is a symmetric matrix then $A = A^T$. The entries of the matrix are symmetric with respect to the main diagonal, so if an element of the matrix A is a, then $a_{ij} = a_{ji}$.

A *positive definite* matrix M is a matrix which when multiplied by any non-zero vector z and its transpose z^T, is always positive. In short the relationship that needs to be satisfied is zMz^T>0

The CGM is an iterative method, which means it can be applied to sparse systems. A *sparse matrix* is a matrix which is populated primarily with zeros. Its opposite would be a *dense matrix*. The algorithm was developed by Magnus Hestenes and Eduard Stiefel and can be used to solve optimization problems[**?**].

2.5.1 The quadratic form

In order to explain why the Conjugate Gradient Method can solve problems whose matrix can only be positive-definite and symmetric an explanation of the quadratic form needs to be presented. The following explanation will also outline the reason why this algorithm is used in optimization problems too.

$$f(x) = \frac{1}{2}x^T Ax - b^T + c \tag{1}$$

The gradient $\nabla f(x)$ of the quadratic form(1) is a vector field that points in the direction of the greatest increase of $f(x)$. If we were to take into account the ith component of ∇f, then.

$$\begin{aligned} f(x + \psi e_i) &= \frac{1}{2}(x + \psi e_i)^T A(x + \psi e_i) - b^T(x + \psi e_i) + c \\ &= \frac{1}{2}(x^T Ax + \psi e_i^T Ax + x^T A\psi e_i) - b^T(x + \psi e_i) + c \end{aligned} \tag{2}$$

Note that e$_i$ is the error affiliated with taking the ith component of ∇f. Now from the definition of a derivative $f(x)$=$\frac{f(a+h)-f(a)}{h}$ and (2) we have.

$$\frac{f(x + \psi e_i) - f(x)}{\psi} = \frac{\frac{1}{2}(\psi e_i^T Ax + x^T A\psi e_i) - \psi e_i^T b}{\psi} \tag{3}$$

Equation (3) is the same as the ith component of $\frac{1}{2}$(Ax+A^{T}x)-b So we have

$$\nabla f = \frac{1}{2}A^T x + \frac{1}{2}Ax - b \tag{4}$$

But if A is symmetric, then it is apparent that we have $Ax = b$ at the minimum of f. In addition if A is positive negative then f is concave up.

2.5.2 Conjugate Gradients[?][?][?]

If we were to consider the original function of the quadratic form(1), then in order to apply optimization, what needs to be done is to find the minimum of Equation (1).If the process and results are correct then minima can be found for quadratic functions with n variables.

As it is for most optimization techniques(for example the method Steepest Descent[**?**]) the algorithm searches for a direction p_k for the next step of x in each iteration. The same goes for he CGM.

$$x_{k+1} = x_k + \alpha p_k \tag{5}$$

It would be useful to point out that since a A is symmetric and positive definite, then if we have two search directions p_f and p_l which are conjugate with respect to A, then the following equation is satisfied.[**?**]

$$p_f A p_l = 0, f \neq l \tag{6}$$

Consider (5), a number of conjugate direction $p_1, p_2, ..., p_k$ and an initial x_1. By minimizing $f(x)$ with p_1, a new point will be obtained which will be x_2. If this procedure is followed until x_k and p_k are reached, then the point x_k will be the minimum of $f(x)$. This would denote:

$$\nabla f_k = \nabla f(x_k) = x_k^T A + b \tag{7}$$

Equation (7) is deduced by taking the derivative from Equation (1). Again by considering Equation (5), it is clear that the factor α cannot be random, but rather should be optimized, in order to achieve results faster. More specifically, it should optimize Equation (5). It should also be mentioned that α is a constant for each iteration and not for the entirety of the algorithm i.e. has a different value for each iteration. Given the above information it is deductible that α must minimize $f(\alpha)$, which means the following.

$$min\alpha \rightarrow f(x_k + \alpha p_k) \tag{8}$$

If Equation (8) is applied to (1), then

$$f(x_k + \alpha p_k) = \frac{1}{2}(x_k + \alpha p_k)^T A(x_k + \alpha p_k) + b^T(x_k + \alpha p_k) + c \tag{9}$$

In order to minimize α the derivative with respect to α needs to be derived

$$\frac{df(x_k + \alpha p_k)}{d\alpha} = \frac{1}{2}(x_k + \alpha p_k)^T A p_k + \frac{1}{2}[p_k^T A(x_k + \alpha p_k)]^T + b^T p_k = (x_k + \alpha p_k)^T A p_k + b^T p_k \tag{10}$$

To deduce the optimal(minimal) vale for α, the derivative calculated in Equation (10) needs to be equated to 0.

$$\begin{aligned} (x_k + \alpha p_k)^T A p_k + b^T p_k &= 0 \\ x_k^T A p_k + \alpha p_k^T A p_k + b^T p_k &= 0 \\ \alpha p_k^T A p_k &= -x_k^T A p_k - b^T p_k \end{aligned} \tag{11}$$

It was mentioned before that for the needs of this algorithm A is a symmetric matrix among other properties. If this is and Equation (7) are taken into account, then Equation 11 becomes

$$\alpha p_k^T A p_k = -p_k^T \nabla f_k \tag{12}$$

Solving for α yields the following equation

$$\alpha = -\frac{p_k^T \nabla f_K}{p_k^T A p_k} \tag{13}$$

Equation (13) forms the value for α that helps towards finding the the next step for x_k. However a valid step is needed for the search direction of p_{k+1} as well. In the Conjugate Gradient Method this step is represented as follows.

$$p_{k+1} = -\nabla f_{k+1} + \beta p_k \tag{14}$$

In Equation (14) βp_k shows the deflection of the search direction with respect to the steepest descent direction. It is also noteworthy to mention that p_{k+1} and p_k are conjugate in respect to A. Combining the process to find α described above and Equation (6) we get the following equation for β.

$$\begin{aligned} (-\nabla f_{k+1} + \beta p_k)^T A p_k &= 0 \\ \beta &= \frac{\nabla f_{k+1}^T A p_k}{p_k^T A p_k} \end{aligned} \tag{15}$$

Having completed the equations for α and β the most important calculations for the algorithm are completed. However their equations are not really handy for the final algorithm, which means they could be improved in order to have easier and faster calculations. In order to do that consider Equations (5) and (7).

$$\begin{aligned} p_k &= \frac{x_k + 1 - x_k}{\alpha}, Equation(5) altered \\ A p_k &= \frac{A x_k + 1 - A x_k}{\alpha}, multiplying A \\ A p_k &= \frac{\nabla f_{k+1} - \nabla f_k}{\alpha}, Equation(7) applied \end{aligned} \tag{16}$$

Multiplying ∇f_k with the previous step of Equation (14) the following equation is derived.

$$p_k^T \nabla f_k = -\nabla f_k^T \nabla f_k + \beta p_{k-1}^T \nabla f_k \tag{17}$$

One condition that was made previously was that $\frac{df}{d\alpha} = 0$ make p_k and ∇f_{k+1} orthogonal. If this is applied to (17), then the following equation is deduced.

$$p_k^T \nabla f_k = -\nabla f_k^T \nabla f_k \tag{18}$$

With Equation (18) the final form of α and β can be deduced

$$\begin{aligned} \alpha &= \frac{\nabla f_k^T \nabla f_k}{p_k^T A p_k} \\ \beta &= \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_K^T \nabla f_k} \end{aligned} \tag{19}$$

As far as β's final equation is concerned, its value is deduced by considering the fact that $\nabla f_{k+1}^T \nabla f_k = 0$

2.5.3 The algorithm

Combining the optimal value of α(8) and β(9) and a step direction for p_k the Conjugate Gradient Method is formed as follows

r_0=b-Ax_0
p_0=r_0
k=0
for k to 1000000 **do**
  $\alpha_k = \frac{r_k^T * r_k}{p_k^T * A * p_k}$
  x_{k+1}=x_k+α_k*p_k
  r_{k+1}=r_k-α_k*p_k*A
  if r_{k+1}^T*r_{k+1} is small enough **then**
    break
  end if
  $\beta_k = \frac{r_{k+1}^T * r_{k+1}}{r_k^T * r_k}$
  p_{k+1}=r_{k+1}+β_k*p_k
  k=k+1
end for

The difference between the values of α and β in the algorithm and the ones in the previous section is that ∇f_n has been replaced with r_n. In this case these two values are equal, since the r vector is deduced by evaluating the first derivative of the quadratic form.

It should also be noted that the analysis given in the quadratic form section enables the Conjugate Gradient method to solve systems of linear equations, but it is from analyzing the quadratic form's properties that we get the values of α and β[**?**][**?**][**?**]

Another interesting application of the Conjugate Gradient Method is that you can apply precondition on it in order to find the solution faster. However, preconditioning is outside of the scope of this project, since it adds a lot more calculation and would need a less general solution.

In general, the time complexity of the CGM is $\sqrt{n}$, where n is the size of the input matrix. Performing this algorithm sequentially would make the matrix-vector multiplication finish in $O(N^2)$ steps and the inner products in $O(N)$ steps.

2.5.4 Parallel solutions of the CGM

Some parallel implementations of the CGM use blocks(parts) of the input vectors and matrix and assign them carefully to specific cores, thus letting each core produce output that will be used in the next iteration. These kinds of solutions work better with vector processors of high processing capabilities, since they are customized for the algorithm's specification. Depending on how many processors a machine has, parts of the algorithm can continue being sliced to blocks, until the optimal implementation is reached[**?**].

Some other parallel implementations try to use more specific preconditioning to achieve faster results, but again for the parallel part, they split the input into blocks to distribute it to various cores[**?**][**?**].

One of the most interesting implementations of the CGM, suggests an improved algorithm named ICGS(Improved Conjugate Gradient Squared), which is based in another already altered version of the CGM[**?**]. This implementation computes all vector-matrix multiplications and inner products concurrently for each iteration. The communication time between these operations and vector updates has been organized efficiently, so that global communication drops significantly, thus dropping the run time of the algorithm as well[**?**].

Finally, there are many recent implementations which have used GPU's in order to solve the Conjugate Gradient Method efficiently. These techniques use CUDA as their main tool and various other techniques to accomplish their goal while achieving very good results. Solutions in different GPU models exist but that just enhances the generality of the solution. Generally,as is for most paral-

lel implementations of the CGM, the most used technique is reducing the time it takes to compute the matrix-vector multiplication and inner products, thus reducing the time of the algorithm overall[**?**][**?**].

3 Design

As it was stated in the introduction the purpose of this project was to significantly reduce the time that the CGM takes to run, by reducing the time complexity of its most expensive computations. These computations were matrix-vector multiplication and inner-products. In this section the basic communication, the matrix-vector multiplication and the flow of the program is considered. In addition to that the contents of the Target Tables are shown, along with the way the event handlers decide what operation to do, namely the Operation Codes. The section finishes, by discussing some of the design choices that were made.

Throughout this section, an element of the matrix,vectors and constants in the algorithm which will be placed into a core is going to be referred to as a *node*. This way the algorithm will be viewed easier as a graph in the SpiNNaker system, which might help with the visualization of the problem.

3.1 Basic communication

Using the massive parallelism that the SpiNNaker architecture provides with over a million cores, it was a clear step that each element of the input matrix(A) and vectors(b,r) would be put into a different node. Considering that choice, it

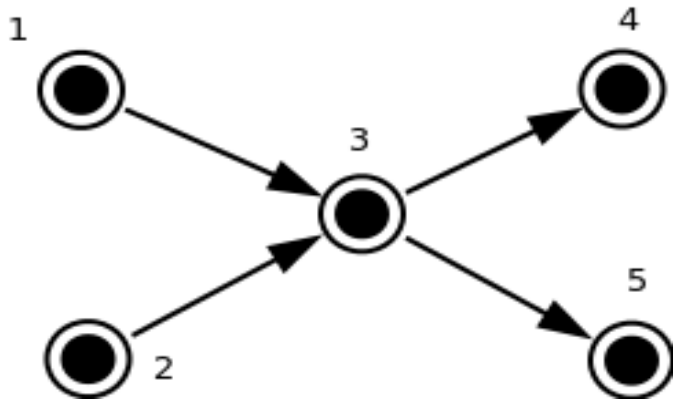

Figure 6: Basic communication between nodes.

is apparent that each node would need to receive and send spikes to a number of other nodes at any given time, in the same fashion that Figure 6 shows. In Figure 6 node 3 receives packets from nodes 1 and 2 and after some manipulation of the incoming data it sends its results to nodes 4 and 5.

If this model is to be used each computation would be completed atomically and if only one computation is needed in each node then the complexity at each core would be $O(1)$.

3.2 Matrix-Vector Multiplication

Using the above model a matrix-vector multiplication would look like Figure 7.

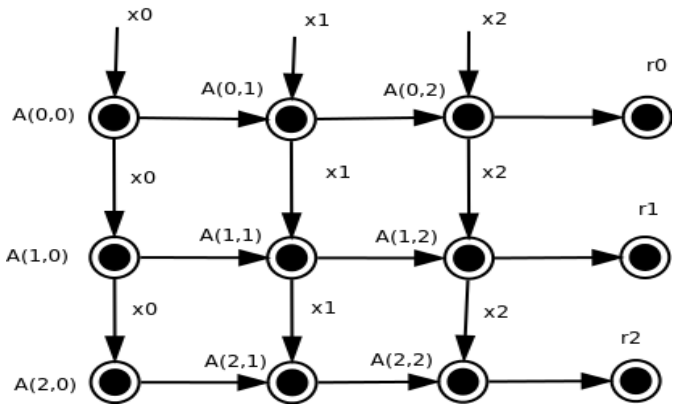

Figure 7: Matrix-vector multiplication for A*x. A nodes send the result to the respective r node, not the next A node[**?**]

In Figure 7 the first matrix-vector multiplication of the algorithm is used($Ax = r$) as an example to display the general overview of the multiplication operation. In this figure the example of a 3x3 A matrix and 3x1 vectors is used for simplicity. Also assume that the respective element of the vector b has already been placed in the respective r node, waiting to complete the calculation for r, once the multiplication is over.

To do the calculation, the x-nodes send a packet with their value to the respective A-nodes, where the multiplication part is completed. The A nodes know which packets they should receive, because the connections between nodes have been provided externally by the user. The packets from x reach the A nodes because they are multicast. The column of the element in the matrix is

considered to identify which x nodes should be received by the A-nodes. The figure shows without precision that A-nodes send their value to the next A-node in the column. This has been done for means of clarity, but what the graph means to show is x nodes sending their value down the line. The same applies for when a A-node, after having completed the multiplication part of the calculation, sends the result to its respective r-node. Each time a packet arrives in the r-node an addition occurs, so as to complete the additive part of the multiplication.

In Figure 7 the node x_0 sends its value to the A nodes with column 0($A_{0,0}A_{1,0}A_{2,0}$). Once the multiplication in each of these nodes is completed, the result is sent to the r_0 node. The same holds for the other two sets of A nodes and their respective r nodes.

Given Figure 7, it is clear that the multiplication in each A-node is done in $O(1)$. The same applies for the addition in the r-nodes, since there is some delay from the time the packet is sent and the time it arrives. Considering the communication tree built and discussed in section 2.3.2, the overall time-complexity for a matrix-vector multiplication to complete is $O(1) + log_2(N) + O(1)$. However, this is an optimal time-complexity that requires only one computation in each node. In practice as it will be explained later, the complexity is larger.

3.3 Program Flow

Given what has been discussed so far consider the following figure. Figure 8

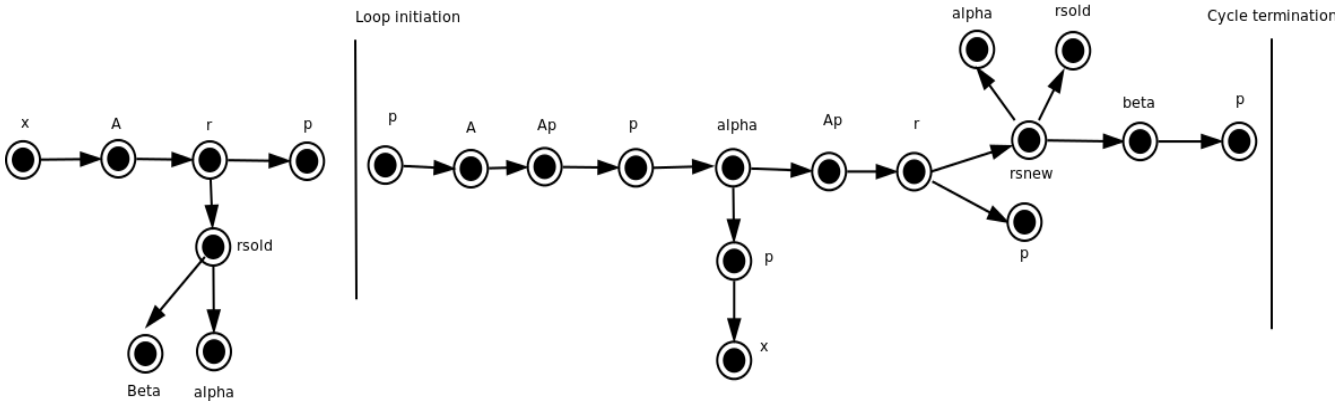

Figure 8: The program flow in class diagram form.

demonstrates the program flow of the CGM through the SpiNNaker machine.

Each node in the figure represents a number of nodes during run time. If the example from the previous section is continued(a 3x3 matrix) the x node in the beginning of Figure 8 represents 3 actual nodes. The A node in Figure 8, represents N^2(9 in the example) nodes, since it has NxN dimensions.

In addition to the input nodes x, A, b it is visible that there are some additional nodes that exist throughout the algorithm. These nodes are helper nodes, without whom the algorithm would not be solvable in this context. A set of nodes that is not found in the CGM algorithm explained previously are Ap nodes. These nodes are used to save the result of the $A * p$ computation and help towards running the algorithm.

Some constant nodes that can be found in Figure 8 are also noteworthy. These are α(alpha in Figure 8),β,(beta in Figure 8),$rsold$ and $rsnew$. These nodes were used to compute and save constant values that are needed for the computation of the algorithm. The constants α and β can be found in the original algorithm, but this is not the case for $rsold$ and $rsnew$. These constants are used to save the result of the inner product $r^T r$ in each iteration. Since both r_k and r_{k+1} are needed in each computation, two variables are used to save their respective result.

Notice that the program flow is quite sequential in the way it executes, so as to avoid bad timing, which will result in wrong computations. Even though the flow seems sequential, the algorithm runs in parallel for each calculation. Some cases where the program flow seems to be more parallel is when rsold and rsnew send to many different kinds of nodes at once.

Another point of interest would be that there is not a node of type b in Figure 8, which from the definition of the algorithm is seemingly an input node. It was decided that it would be best to place the b node values immediately to the r nodes, so as to speedup the calculation, save space and cores, and reduce the overall complexity. This step is allowed by the calculation, since r nodes at the beginning of the algorithm are initialized to 0.

Finally, it should also be mentioned that after the first iteration of the algorithm whatever operation exists before the loop initiation is not needed.

3.4 Target Table

As mentioned in the Background section each node in the system has a Target Table entry, which helps each node complete various computations by reading

its contents. For the Conjugate Gradient Method the Target Table entry is as follows.

- unsigned Kd
- vector<queue<unsigned» OpCodes
- char Name
- unsigned YD
- unsigned X
- unsigned Y
- float Value
- float Temp
- unsigned counter
- unsigned V

The Kd identifies the key of the node, which is individual to each node in the system.

The Name identifies the type of which the node belongs to. The YD variable is the Y dimension of the element. For example, if the algorithm could take as input a 4x3 matrix, then its Y dimension would be 4 and its X dimension 3. However, in the CGM the matrices are symmetric and the vectors have as an X dimension 1, so XD would be redundant to add in the table.

The X and Y variables are there to identify the position of the node in the matrix or vector that it belongs to.

The floats Value,Temp and unsigned counter are the original values of Value,Temp and counter, which are mainly there to show that these kinds of variables exist and are used for each node. Their position in the table is not vital, since their values cannot be changed throughout the program in the Target Table.

Finally, there is the variable V, which is used to identify the position of the variables Value,Temp,counter and OpCodes in each core's data depending on the node that it belongs.

3.5 OpCodes

OpCodes in this context stands for operation codes, as in what operation needs to be done at any point in a node. Throughout the algorithm different kinds of operations are done. Be it addition,multiplication, subtraction etc., different operations need to be completed at different points of the program. Thus an unsigned integer is used to identify these operations upon the arrival of a packet. So for example, 1 would be used to identify addition, 2 for multiplication etc. The operation would be carried out after the node went through a series of cases in a switch statement that would be checked against the OpCode.

If the program flow is looked thoroughly, it is apparent that some nodes need to do different computations at different points of the algorithm. For example, a p node needs to either store the value of its respective r node, or do a computation such as $\alpha * p$. These are two completely different operations that need to be accounted for. Given that, the easy solution would be to use a vector to save all the operations that need to be done and after reading the first element of the vector and completing the respective operation , put it in the back of the vector, since CGM is an iterative algorithm and some computations need to be repeated.

However, just a vector would not work. Consider the following. Based on the general connectivity between nodes the α node can receive packets from both r-nodes and p-nodes. That means that just before the loop initiates on Figure 8, the α node will receive packets from both r-nodes and p nodes. In addition to that, r nodes send their values to p nodes. There is a chance that packets from p nodes will reach the α node sooner than r nodes do. In the case of p nodes arriving, the operation would be to do nothing, since that information is not needed at that point and in the case of r nodes arriving to compute the inner product. This sequence of events cannot be foreseen and with just a vector to represent the OpCodes the wrong calculation is to be expected. This behaviour is observed for most of the nodes in the algorithm.

Thus the choice of a vector of queues was made. With this data structure, the index of the vector is selected to act as a key to find the type of the node that the packet came from. Each type of node has a distinguished offset in the source address that will make it identifiable to the vector. For example, nodes from matrix A have an offset of 0. Once the offset is read by the vector, a queue will come up. The front integer of the queue will be the operation that needs to be carried out at that point for the packet that had just arrived. Afterwards, in the same fashion as with the vector, the front OpCode of the queue will be placed in the back of the respective queue for the next iteration.

By using this data structure it is apparent that the time complexity at each core has increased, because more operations to handle the queue take place. However, using this specific data structure does not increase the complexity dramatically. The operator for accessing a vector is $O(1)$. The same applies for each of the following actions: looking at the front value of the queue, popping it from the front of the queue and pushing it in the back of the queue. Considering this the time complexity at each core has a small increase totalling $O(4)$.

3.6 Justification for Design Choices

At this point a justification for some design decisions would be appropriate.

The SpiNNaker architecture gives you the choice of having global variables that would be accessible by all nodes of the system. That feature would be usable considering constant values such as α, β, $rsold$ and $rsnew$. Instead nodes were selected to contain and compute these values. The reason for that was that better control over computation on certain constants was provided when using constant nodes. This is because there was certainty that the value would change only when it was supposed to(when a packet would arrive in that constant node), whether with a global value reading a wrong value was a possibility.

Another important design decision was the use of the vector of queues data structure. Given the iterative nature of the algorithm, following a model such as the one described has proved be very usable and simple to program. There existed an option of using a graph of nodes with multiple connections used, but that option was deemed too complicated, with the possibility of not even being implementable given the nature of this algorithm. However, it might have worked for other algorithms that do not require iterations to find solutions and do not involve many steps.

4 Implementation

In this section some implementation details are going to be outlined. More specifically a description of the simulators used and the process that the keys for the vector of queues were generated. In addition, the iterations after the first iteration will be described as well as the general specification of the OpCodes for each node.

4.1 Simulators

To implement and test this algorithm simulators of the SpiNNaker chip were used. These were supplied by the supervisor of this project and were written in the C++ language.

The main development of this algorithm was made in the first simulator. In this simulator the programmer had the ability to use STL(Standard Template Library), which helped towards abstracting the problem and leaving out some implementation details. The same simulator would also write out the binary files that the second simulator and the actual machine would use. This simulator was configured to comply better with the specification of this project.

The second simulator was much less flexible towards using data structures, since it had to simulate much better the functionality of a SpiNNaker chip, when data and a program was loaded to it. It read the same binary files that the SpiNNaker machine would read and the event handlers written for it resembled the event handlers for the SpiNNaker machine much more.

4.2 Offset

As mentioned in the Background section, each node has an 32-bit address-key. In order to generate the key 4 elements are used. These are

- X position of the chip in the mesh
- Y position of the chip in the mesh
- Core number
- Offset which is the ID of the node in the core.

In addition, it was mentioned in the Design section that the index of the vector acted as a key to access the queues of OpCodes, in order to select the correct operation at any given time. The key for the vector is the offset of the source key of the incoming spike(packet).

The offset for each element is allocated at the time that all keys are generated. A switch statement was used to distinguish the nodes by name and then alter the offset according to the name.

Since each core can theoretically have more than one nodes allocated to it, redefining the offset might cause some problems. For example, if elements of the

same type and offset are allocated to the same core then one of them might be overwritten.

However, if that practice is avoided then this problem will not occur. In addition to that, the solution to this problem has been designed so as to allocate one node of whatever type to only *one* core, so as to achieve the highest speedup possible. Hence putting more than one elements to a core would not comply with the overall design of this project.

4.3 Second Iteration

As mentioned before, after the first iteration some operations must not be completed again, so as to not disrupt the completion of the algorithm. These operations are the ones that are left from the "Loop Initiation" line in Figure 7. To accomplish that a special callback(function) was written. This function would be called only once throughout the run time of the program once all remaining packets had arrived at the p nodes(See last node in Figure 8) after the end of the first iteration. That function would issue commands to all the nodes in the system, making them remove at least one OpCode from one of the queues, based on the type of element it is. The name of the element was considered to make that decision.

4.4 Input files

In order to simulate and run a program an input file would have to be put on the first simulator, so as to process the data and write the binary files. Hence a file format was invented to comply with the specification of this project. An example of the input file is shown in Appendix(8.1). At the Values section, the elements of matrices, vectors and constant nodes are described. The names, dimensions inside the brackets and actual values are specified. The OpCodes for each type of element are also visible. They are the integers next to the dimensions of each matrix/vector/constant. The first integer is the key and all the others are actual OpCodes. Different queues are separated with a semicolon. In the Connections section, the connections between nodes are specified. The general layout for the OpCodes for any size matrix is shown in Appendix 8.2.

4.5 Implementation OpCodes

As explained before, the SpiNNaker architecture does not support any libraries that would help form a queue data structure which is pivotal to the design analyzed previously. In addition to that, the contents of the binary files are read one at a time, so an STL queue would not be readable from them. Finally, SpiNNaker cannot read serialized data, so serialization was not a valid option either.

Considering all the aforementioned parameters, it was decided that the vector of queues of unsigned integers, would be implemented using only an array. To successfully translate the vector of queues of integers data structure the following format was selected.

1. Number of queues.
2. Size of next queue.
3. Head
4. Tail
5. Key.
6. Data.

The number of queues(1) would exist only once in each array, so as to know how many different queues exist. The information from 2-6 is the information that the array would need to simulate one queue. This information would be repeated in each array, depending on how many queues existed. The size and the key would appear only once in each simulated queue. The data consists of the actual OpCodes(integers) that each node would look into to execute the next operation. The Key is the index that the vector would look using the offset of the incoming packet. Finally, the Head and Tail would act as pointers to the first and the last element of the queue respectively, so as to execute queue like operations.

Given the above format, three functions were also implemented to do operations on that array and more precisely in the simulated queues in that array. These were front(),pop() and pushandpop(). Depending on the offset of the source key of the arrived packet the front() function would show the first integer on the simulated queue, pop() would remove the OpCode in the front position from the queue and pushandpop() would put the OpCode in the first position in the back of the queue. When using pop(), the OpCodes is not actually removed, but rather the tail is reduce, so as to make the queue unable to

access it again.

4.6 Connections and OpCodes programs

In addition to the implementation of the algorithm, two additional programs were constructed in order to create the input files quickly and without errors.

A Connections C++ program was written to create the connections between nodes and add them in the Connections part of the input .mx file. For the connections between the matrix and the vectors, the rows and columns of the elements were taken into account. For the nodes containing constant values, the connections with other input nodes were more constant, but would also vary given the size of the input matrix and vectors.

An OpCodes C++ program was written to produce the operation codes for each node and also add it to the .mx input file. To create this program the general layout of OpCodes in Appendix 8.2 was used.

Combining these two programs, most of the input file is written automatically. All the user needs to do is insert the values and dimensions of the input vectors and matrix correctly.

4.7 SpiNNaker Event Handler

Towards the end of the project the event handler for a SpiNNaker machine was created. This event handler resembled highly the ones created for the simulators, with the difference of having only one timer event function instead of two. The reason for that is that only one timer event handler is allowed. The handler was written in C and it used functions from the SpiNNaker API to send messages, trigger the timer function and start off the program in the machine. A 2x2 SpiNNaker board containing 64 cores was used to test the algorithm and according binary files were produce to match it.

5 Testing

The testing phase of the algorithm took place in both of the simulators. This needed to be done because not only did the result of the algorithm had to be verified, but also because corruption of binary files was also a possibility. In

addition to that, the functionality of the implementation queue needed to be tested.

The actual testing was made by using different sets of matrices, with different values and different sizes. This was done to test whether the solution of the algorithm was scalable and robust. The results from the simulators were cross referenced with the results from a sequential implementation of the algorithm to check their validity. An example of the test made and their results are shown in Appendix 8.3. More examples can be found in the Git-hub repository where most implementation files are saved[**?**].

All of the tests compiled in both of the simulators, while also producing the correct output. Throughout the testing phase there were no problems with corrupted binary files.

6 Evaluation

In general the design and implementation that has been proposed and tested throughout this project has been proven to be robust and scalable. The data structure of vector of queues could provide a solution not only for the Conjugate Gradient Method, but also for other algorithms that might be similar to this one. Using the proposed design, a programmer could implement other algorithms by sketching out a program flow similar to Figure 8, and then finding the connections between the nodes. In addition, using the proposed design offers the advantage of few and understandable connections between nodes that can be easily produced using a simple program.

One disadvantage of the design is the additional complexity that it adds in each computation from $O(1)$ to $O(4)$. However this is not a dramatic increase and will not reduce the speedup dramatically. In addition, creating a customized vector of queues of integers using an array might prove to be complex if used for some other algorithms. However, if the design proposed in the Implementation section of this project is followed, then the task might prove to be easier.

One improvement that this project could have benefited from is running the algorithm in a SpiNNaker chip to see if it would work in the actual machine as well as it did in the simulators. Having tested that, by using the SpiNNaker chips, the practical speedup that the architecture provides could have been found and compared to the sequential implementation of the algorithm, as well as other parallel implementations. Unfortunately, the time available with

a SpiNNaker board was limited and in addition to that there was a malfunction to the board during testing time.

Moreover, more testing could have been carried out in the simulators throughout this project to make sure whether the algorithm could scale to matrices with very high dimensions i.e. a 400x400 sparse matrix. The tests that were carried out do indicate that a correct solution would have been produced, but doing the actual testing would improve upon the validity of the solution.

In the event handler it is visible that there are around 13 different operations that can be carried out by all the nodes. This number could be reduced from 13 to 11 or 10, by optimizing some of the operations that each case does. During the implementation stage, a higher number of operations was chosen in order to test each operation's validity more atomically.

The hardest part of this project was coming up with an efficient way of doing operations in each node, while preserving a low time complexity in each computation and also producing the correct result. The vector of queues of integers that was selected to represent OpCodes, while using the Offset of each incoming packet's source address as a key to the OpCodes. One of the easier tasks of this project was the creation of the programs to automatically produce the OpCodes for each node and the connections between them.

6.1 Future Work

In the future, attempting experiments to verify the complexity of the packet sending would be very useful. In this project $log(N)$ is assumed because of the way the connections are built. However, experiments could indicate the actual time complexity, which would make the validity of the complexity achieved in this project much more accurate.

In addition, it would be very interesting to test the design proposed in this report applied to other algorithms and applications. It would be intriguing to discover if it scales to other computer science problems, or even neural simulations.

Finally, it would be possible to see the queue data structure that was implemented by using an array have even more optimal operations.

7 Conclusion

In conclusion, this project achieved the goal that was outlined in the project brief by finding a solution to the Conjugate Gradient Method algorithm using the SpiNNaker architecture. It took advantage of the massive parallelism that the architecture provides and was able to dramatically reduce the time complexity of the most expensive operations of the algorithm. The matrix-vector multiplication was reduced from $O(N^2)$ to $O(4) + log_2(N) + O(4)$ and the inner product from $O(N)$ to $O(4)$.