Electronics and Computer Science Faculty
of Physical and Applied Sciences
University of Southampton

Alexandros-Pana Oikonomou
1^{st} May 2013

Solving the Conjugate Gradient Method in a SpiNNaker Machine

Project Supervisor: Jeff Reeve
Second Examiner: Marcus Breede

A project report submitted for the award of
BSc Computer Science

Abstract

SpiNNaker is an asynchronous, event-driven parallel architecture designed to simulate the human brain. It has been designed to operate as a large scale neural network in real-time using a System-on-Chip multi core system. Its architecture is different from usual parallel computers, since cores use spikes to communicate with each other. That way usual pitfalls of parallel computing, such as race conditions and deadlocks are avoided. So far the most prominent uses of this architecture have been in neuroscience and robotics. The aim of this project is to put into use SpiNNaker's architecture and bring it closer to classic computer science problems, while solving them optimally. The given algorithm to solve in this project is the conjugate gradient method, an iterative way of solving systems linear equations. The algorithm successfully runs on the simulator and reduces the time complexity of the most expensive operations of the algorithm.

Contents

Acknowledgments and Statement of Originality

I would like to thank my supervisor Jeff Reeve for his help and support throughout this project.

1 Introduction

1.1 Aim

The aim of this project is to correctly solve the Conjugate Gradient Method[10] on a SpiNNaker chip, thus using the massive parallelism that this machine offers 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 the scalar product of vectors. The complexity is reduced dramatically, due to the abundant number of cores provided from the architecture. This report explains this project and its constituents, any background research done to launch this project, along with designe 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[14], which have mainly been used to simulate the neurons of the human brain and in robotics.

However little work had been done into using the SpiNNaker architecture to solve classic computer science problems. That is why a problem such as the Conjugate Gradient Method had 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[23].

1.3 Overview

Give a brief overveiw of what each section contatins

2 Background

2.1 The neuron

To make the explanation of the SpiNNaker architecture easier, 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 costitutes 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. The dendrite extends for hundreds of micrometers and branches multiple times, 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

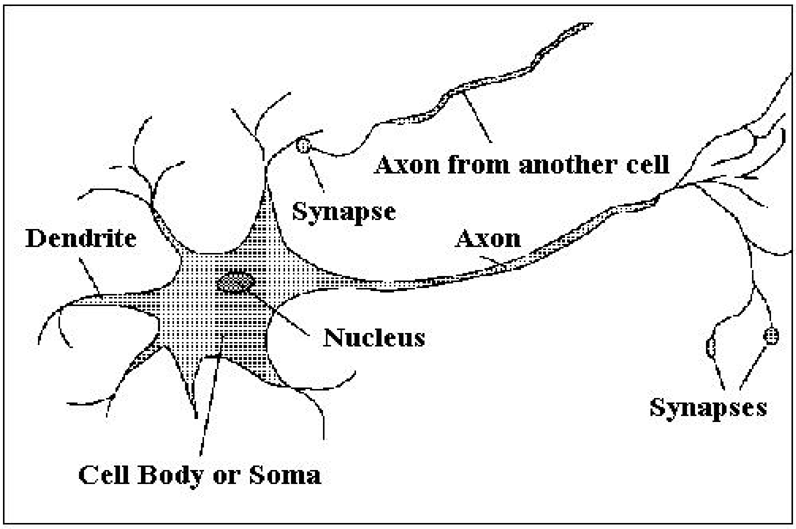

Figure 1: A neuron

One of the most important parts of the neuron structure is the synapse. The synapse is the contact between the axon of a neuron and the dendrite or the soma of another. It is where a information from one neuron is transmitted to the other. When a set of neurons are connected with each other through synapses, then they create a neural network.

Finally, the communication between neurons is accomplihed through spikes, which are either chemichal or electrical[9].

Given the terms in this section, the name of the SpiNNaker chip is detuctable. 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 brain, and more specifically, neurons. However, its architecture is not constrained by the biology of the brain, but many techniques to speed up computations are used. It differs from other supercomputers, which usually have a lot of strong processors with slow network capabilities. The design of the architecture was made with having as priority two concerns. 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[7].

2.2.1 SpiNNaker chip Overview

The heart of the SpiNNaker architecture is the SpiNNaker chip. Its main components are 18 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 startup, in each chip a processor is selected to act as a Monitor processor, with the functionality of performing the management of the given chip's system. The remaining ARM cores perform the calculations for the given problem, with the exception of one which is reserved as a spare, in the emergency of another core having a malfunction[7]. 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 which can hold up to 1GB of data[14].

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 exists in each chip.

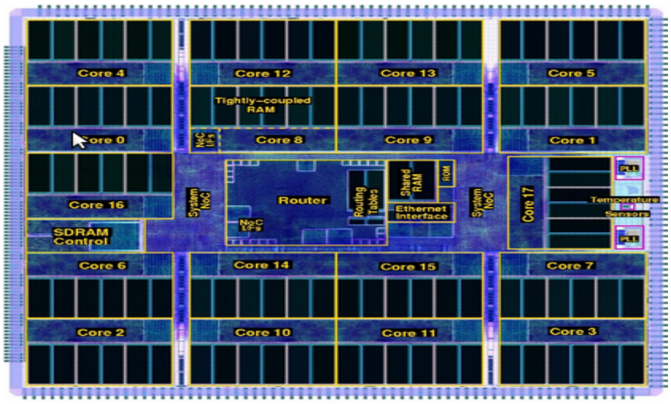

Figure 2: The SpiNNaker chip[11]

The router exists as the part of the NoC and its primary role is to direct packets that arrive and packets that are sent or need to be forwarded. The router has 20 ports for the ARM cores that are located in the given 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. Using multicast packets helps reduce the amount of packets that exist in any givne moment in the network. To help with that, the architecture supports default routing, which means that some connections do not need to be in some routing tables for them to be forwarded to their eventual destination. This concept will be explained 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 an amount of time is passed, then the packet will be dropped. To help with that, each packet has the time it was transmitted in their header.

To further understand this concept, 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 get connected and work efficiently as an architecture. To reach that goal an effective topology is important. Considering Fig-

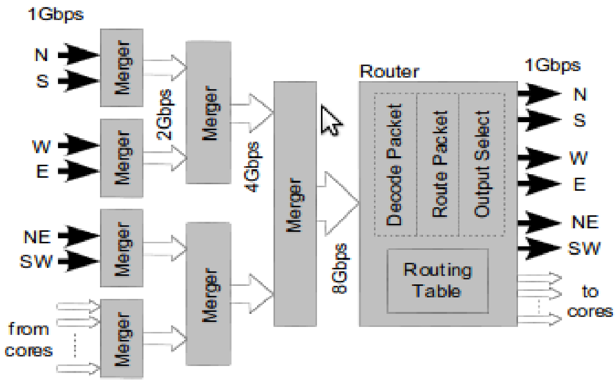

Figure 3: The router[14]

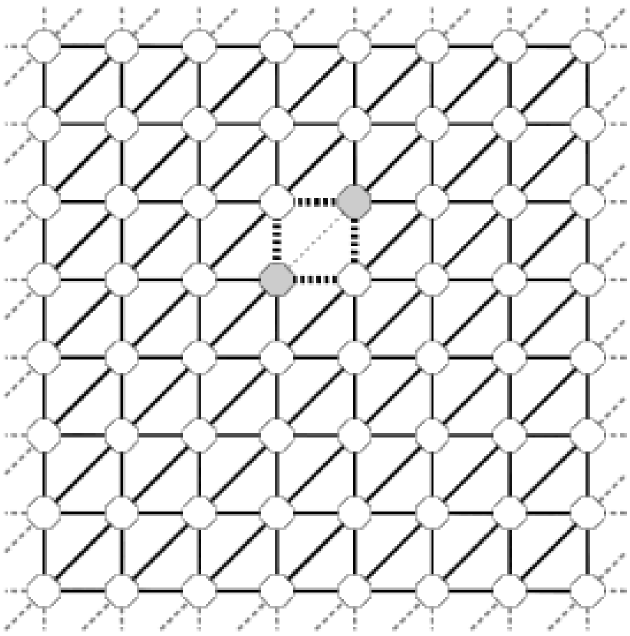

Figure 4: *The SpiNNaker topology[14]. This is an example of what an 8x8 would look like.Each circle is a SpiNNake chip and the lines between them are the connectivity.*

ure 4, it is visible that the topology is viewed by the system as a 2D mesh, with chips having a specific amount of neighbours and specific 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 directly communicates with chips that are North, South, West, East ,NorthEast, and Southwest of its respective position. This however does not apply for chips that are in the perimeter of the mesh, since they in some of the positions described above chips will not exist[6].

Returning now to the default routhing mentioned in the previous section, if a packet arrives to a router, and the router does not have an entry for it, a then it 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 condition is met, then it will be forwarded to the West. This helps reduce the size of the routing tables in each chip[12].

In addition to the default routing, the topology also provides for two-hop routes aming 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[14][17].

2.2.4 Packets

The packets used in the architecture are all multicast packets. They 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. Their size is of around 40 or 70 bits and their transmission is done completely by the hardware of the system, thus having high bandwidth[11][14]. It is also notable that, when simulating a billion neurons, unique addresses are needed to acknowledge them. Hence 32-bit addresses are used to represent items 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 usually would 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. It is a command/control packet and is used when the Monitor Processor communicates with the target chip[5]

in this project mainly the Type 0 packets have 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 that would exist in the routing table would have to be enormous.

This architecture supports routing tables of 1024 entries. Each entry is of 32-bit size, since it needs to hold addresses for the routing. The nature the most siginificant bit of the source is written determines if a particular packet is for distributing locally to the cores, or if it should be linked and forwarded to other chips[5].

2.2.6 Bandwidth

Having talked about how the architecture works, it would be a good point to talk about the bandwidth bisection that it offers. 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 63K. If we would split the topology in two then the optimal goal would be to have all the neurons of one half connected to at least one neuron in the other half. 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. Given that each node should carry 25M packets/sec[5][14].

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 software are three.

- Host Machine, which is mainly used for input and output, and it is the component that the user interacts with to start an application
- Monitor Cores. This is the same Monitor core that was discussed previously. Its 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, which are used to carry out the computation of the application.[6]

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 late sections

2.3.1 Startup

In order to do the startup the devices mentioned above some kind of software needs to be used. For the Host machine to interact with the main Monitor Core *ybug* is used. It is also used to start applications. Ofcourse ybug does not communicate only with the hardware of the monitor core. There is another software 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.[11]

2.3.2 Local Data Structures

Before the application starts running, some other applications need to be comlpeted first. These include finding the state of the machine, finding a communications tree 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. How the algorithm works is that firstly a root node is selected and is marked with a Request(R) token. Each time a core has a R token it sends an Acknowledge(A) label to all the incoming ports. If the labeling is successful, then that incoming port is deemed Good. Otherwise it tries to send an R token to that port and it marks it as Broken[20].

Building a communications tree In order to build a communication tree a breadth first tree traversal is used. To achieve that firstly a root node is selected and given the token Forward(F). The root node speaks to neighbour nodes and after giving them the label Child it says to them to also send F tokens to all their neighbours, thus making them parents. Once there are not any more unlabelled nodes, 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 nodes 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[20].

Creating the P2P Tables In order to know to which nodes a packet needs to be sent, the nodes need an ID. Point-to-Point tables are used to list the ID's of nodes, so it knows which port it should follow. At first all nodes are undefined(U). If we were to find the entry for one undefined node then the table would mark its own incoming port and send its port to all other ports. That way all the other chips in the board would know what port to use to communicate with that node. For the nodes that are on the same chip with that node...........If that is done for the all of the nodes in the system, then the P2P table is built[20].

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 explicitely 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. This is usually done through binary files. the Host starts the execution of the program, after the 'readable' version is has been loaded to the machine. When the program terminates, or the user wishes to terminate it, then the Host issues a halt command that stops all operations[5].

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 low stages[5].

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 beforehands. 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 really control when these specific events will take place(except in the case of the time interval). What the programmer can control is the what happens when such an event occurs. The functions, in this context referred as callbacks, are written by the programmers, which after a specific event occurs, are given a priority with the kernel(sark). The types of callbacks

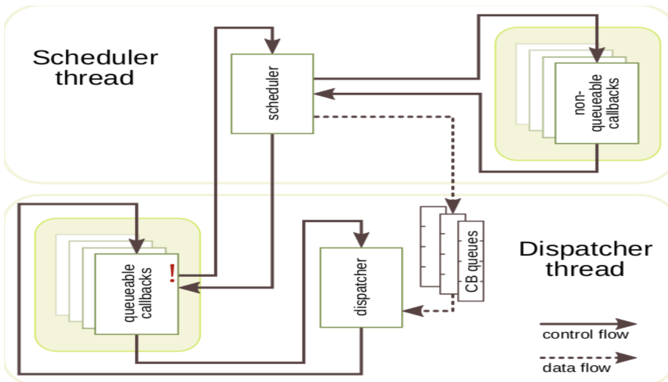

Figure 5: Event-handling. The control and data flow is shown[23]

can be seperated 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[23][19].

2.4 Notable work using the SpiNNaker architecture

Most of the notable work done in SpiNNaker machines is simulation of neurons. Each time different sets of neurons would be used to be simulated. An example would be simulating the neurons of the retina, so as to make it possible for a robot to calculate its place[4]. Another one would be simulating thousands of spiking neurons using four million synapses[22]. Another interesting application would be teaching temporal sequences of discrete symbols, thus making fast and accurate predictions of new sequnces[3]. Most of the notable work done in SpiNNaker machines is simulation of neurons. e

2.5 Conjugate Gradient Method

The conjugate gradient method is an algorithm for the numerical solution of systems of linear eqations 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 multipled 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

It is an iterative method, which means it can be applied to sparse systems. A *sparse matrix* is a matrix which is populated primirily with zeros. Its opposite would be a *dense matrix*. It was developed by Magnus Hestenes and Eduard Stiefel and can be used to solve optimization problems[18].

2.5.1 The quadratic form

In order to explain why the Conjugate Gradient Method can solve problems whose matrix is positive-definite and symmetric an explanation of the quadratic form needs to be presented.

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

The gradient $\nabla f(x)$ of the quadratic form 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.

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

(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 obvious 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

As mentioned before the matrix A must be symmetric and positive definite for the Conjugate Gradient Method to work. Given a quadratic function as defined in (1), then it turns out that $-\nabla f = b - Ax = r$.

The CGM, as well as most optimization methods(for example the method of the Steepest Descent[21]), look towards a specific direction everytime they are to move to a correct position and try to look for the best step given their gradient and direction

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

Given (5), α can be deduced from the fact that e_{k+1}(the error vector of each iteration) must be orthogonal to p$_k$, which also means that p$_k$ will never be met again as the solution progresses. Using this we can dedcuce

$$\begin{aligned} p_k^T e_{k+1} &= 0 \\ p_k^T(e_k + \alpha p_k) &= 0 \\ \alpha &= -\frac{p_k^T e_k}{p_k^T p_k} \end{aligned} \tag{6}$$

The equation at (6) however is not solvable. The value of e_k is needed in each iteration, which is not computable at that time. To help with that the search directions in each iteration(p_k) is defined as A-orthogonal in respect to the matrix A, instead of just orthogonal. The definition of A-orthogonal vectors is, given a matrix A, then:

$$p_k^T A p_k = 0 \tag{7}$$

Given (7), e_{k+1} and p_k must be A-orthogonal. If we wish to find the optimal value for α,then the gradient must be set to 0. This leads to the following equation.

$$0 = \frac{d}{d\alpha} f(x_{k+1}) = \nabla f(x_{k+1})^T \frac{d}{d\alpha} x_{k+1} = -r_{k+1}^T \frac{d}{d\alpha}(x_k + \alpha p_k) = -r_{k+1}^T p_k = p_k^T A e_{k+1} \tag{8}$$

From (6),(7) the equation for α when e_k and p_k are A-orthogonal is

$$\begin{aligned} \alpha &= -\frac{p_k^T A e_k}{p_k^T A p_k} \\ &= \frac{p_k^T r_k}{p_k^T A p_k} \end{aligned} \tag{9}$$

Another important equation that comes from analysing the error term is

$$p_k^T r_k = u_k^T r_k \tag{10}$$

In (10) u_k^T are u vectors that span the vector subspace of the vector field of p_k. They are also orthogonal to r_k and u_{k+1} is also orthogonal to all the previous u vectors.

Using (9) and (10) we are able to reach a more accurate value for α

$$\alpha = \frac{r_k^T r_k}{p_k^T A p_k} \tag{11}$$

Using (9) and (10) and a Gram-Schmidt conjugation we are able to reach and a value for β

$$\beta = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \tag{12}$$

[19]

2.5.3 The algorithm

Combining the optimal value of α(8) and β(9) and a good 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

Some notes that can be made about this algorithm are that the axis is defined by the eigenvectors of A and that the algorithm takes a conjugate step closest to r. Finally, the algorithm guarantees convergeance in at most n steps.[18][24][16]

Ofcourse there is an error that exists in every step, as mentioned before, but it is minimized in each iteration. The error function would be:

$$\left|e_k\right| \leq 2(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^k \left|e_0\right|_A \tag{13}$$

In Equation 9 κ is the condition number.

Another interesting applicatoin 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.

2.5.4 Parallel solutions of the CGM

Some parallel implementations of the CGM use blocks 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, since it allows a processor to complete large vector operations. Depending on how many processors a machine has, parts of the algorithm can continue being sliced to blocks, until the optimal implementation is reached[15].

Some other parallel implementations try to use more specific preconditionings to achieve faster results, but again for the parallel part, they split the input into blocks to distribute it to various cores[2][1].

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[13]. 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 organised efficiently, so that global communication drops significantly, thus dropping the run time of the algorithm as well[26].

Finally, there are many recent implementations which have used GPU's in order to solve the Conjugate Gradient Method efficiently. These pieces of work use CUDA and various other techniques to accomplish their goal, with some very good results. Ofcourse different GPU models are used, but that just enhances the generality of the solution. Generally though,as is for most parallel 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[8][25].

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 multplication and inner-product. In this section the basic communication, the matrix-vector multiplication and the flow of the program is considered.

Throughout this section, each core is going to be referred as a node, so as to think the system as a graph. This might help visualising 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

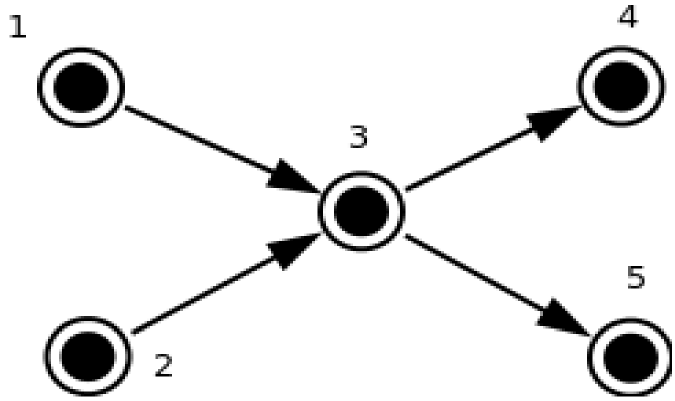

Figure 6: Basic communication between nodes.

that choice, it is easy to see that each node would need to receive and send spikes to different node at each time in the same fashion that Figure 6 shows. In that figure node 3 receives packets from nodes 1 and 2 and after some manipulation of the incoming data it sends its results to nodes 4,5.

If this model is to be used each computation would be completed atomically and if only one computation is needed every time 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.

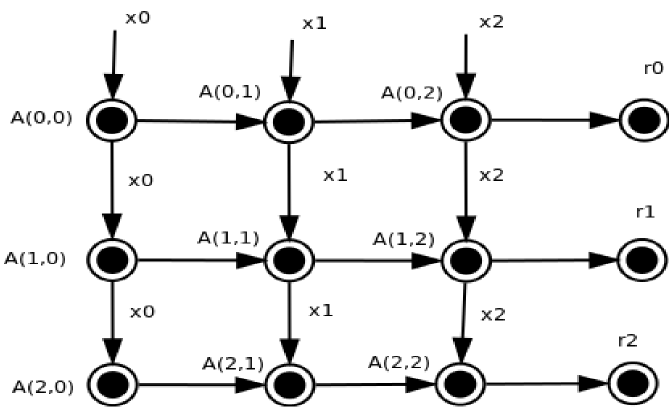

Figure 7: Matrix-vector multiplicatio for A*x.

In Figure 7 the first matrix-vector multiplication of the algorithm is used($Ax = r$) as an examplee to display the general overview of this operation. In this figure the example of a 3x3 matrix A matix and 3x1 vectors is used for simplicity. 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 x-nodes know to which A-nodes they should send based on the connections that were provided beforehands by the user. The column of the element in the matrix is considered to identify which x nodes should be received by the A-nodes. Even though the figure shows that A-nodes send their value to the next A-node in the column, the graph means to show that the x-nodes are the ones that are sent. The figure is shown that way for means of clarity. The same applies for when a A-node, after having completed the multiplication, 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.

To demonstrate the above property with an example, 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 muliplication in each of these nodes is completed, the result is sent to the respective r node. Again this connection has been provided by the user, but this time the row of the A element is considered.

Given the figure above, 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 it arrives. Considering the communication tree built and discussed in section 2.3.2, the overall time-complexity for a matrix-vector multiplication to occur 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 8. Figure 8 demonstrates roughly the program flow of the CGM through the SpiNNaker machine. Each node in the figure represents a number of nodes in the machine. Based on the previous section and this diagram the x node in the beginning of the diagram represents 3 actual nodes, if the example of the previous section is to be continued. In the case of A nodes and Ap nodes where there exist nxn nodes, then the one node representds n^2 nodes.

In addition to the input nodes x, A, b it is viewable that there are some

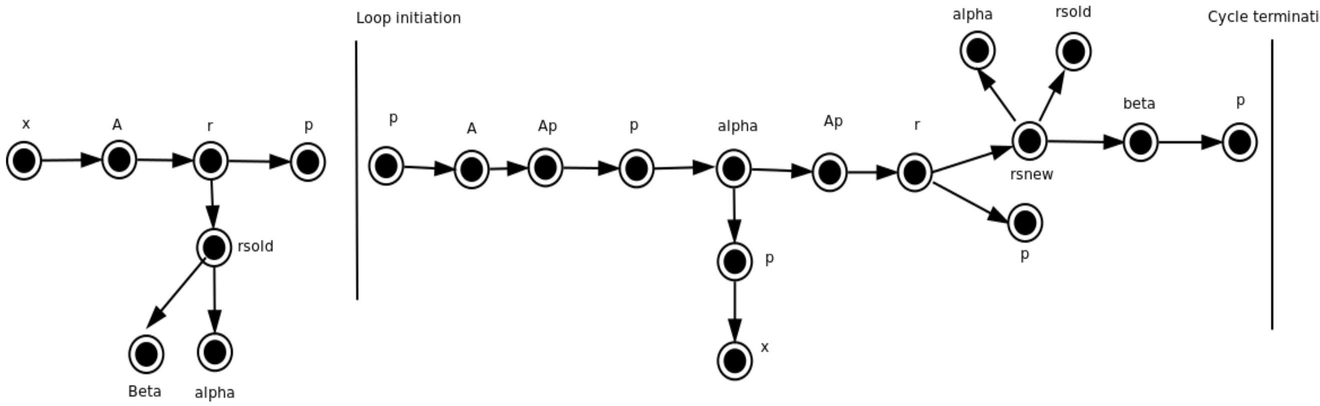

Figure 8: The program flow in class diagram.

more nodes that exist throughout the algorithm. These nodes are helper nodes, without whom the algorithm would not be solvable. 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.

Also consider some constant nodes that can be found in Figure 8. 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 constant α and β can be found in the original algorithm too, 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 $r_k^T r_k$ and $r_{k+1}^T r_{k+1}$ 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 timed and wron calculation. 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 nodes at once. There is no problem of wrong calculation there, because assignement is done in these cases and the nodes invlolved are not used again for a considerable amount of time. In the rsold node this operation is not completed again after the first iteration.

Another point of interest would be that there is not a node of type b in Figure 8, which is an input node. It was decided that it would be best to place the b node values immediately to r nodes, so as to speedup the calculation, save space and cores, and reduce complexity.

Finally, it should also be mentioned that after the first iteration whatever is done before the loop initiation is not needed, hence any operations existed to handle these events should be removed.

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. The Target Table entry is customizable, which means it is different for each application. For the Conjugate Gradient Method the Target Tabel entry is as follows.

- unsigned Kd
- vector<queue<unsigned» OpCodes
- char Name
- usigned 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 the Y dimension would be 3 and the X dimension 4. However, in the CGM the matrixes 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 they are also saved in main memory so as to have the program change their values when needed. Finally, there is the variable V, which is used to identify the position of the variables Value,Temp,counter and OpCodes in main memory.

3.4.1 OpCodes

OpCodes in this context stands for operation codes, as in what operation needs to be done at some point. 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 has been 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.

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 in its value, or do a computation such as $\alpha * p$. These are two completely different operations that need to be accounted for. Given the above argument the easy solution would be to use a vector to save all the operations that need to be done and after the current operation is completed, 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 connectivity α can receive packets from both r-nodes and p-nodes. Also r nodes send their values to p nodes. There is a chance that packets from p nodes reach the α node sooner than r nodes do. In the case of p nodes arriving, the operation would be do nothing, since that information is not needed at that point and in the case r nodes arriving to find the inner product. This sequence of events cannot be forseen and with just a vector to represent the OpCodes wrong calculation will be completed.

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 has come 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 the matrix A have an offset of 0. Once the offset is read by the vector queue will come up. The front integer of the queue will be the operation that needs to be carried out at that point for that packet that just arrived. Afterwards, in the same fashion as with the vector, the front of the queue will be placed in the back of the queue for the next iteration.

By using this data structure the time complexity at each core has increased. However, by using this particular data structure the increase is not that big. 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 $O(4)$.

3.5 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 some kind of global variables that would be accessible by all nodes of the system. That feature would be usable considering constant values such as α and β, but instead nodes were selected to contain and compute these values. The reason for that was the algorithm itself. Better control over computation on certain nodes was provided when having constant nodes. This phenomenon occured because there was certainty that the value would change only when it was supposed to(when a packet would arrive in the 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 really easy to use and program. There existed an option of using a graph of nodes with multiple connections used, but that option was deemed too complicated, with the option of not even being manageable given this algorithm. However, it might have worked for other algorithms that do not require iterations to find solutions and do not have many steps.

4 Implementation

4.1 The Simulator

4.2 Offset

4.3 Second Iteration

References

[1] Loyce Adams. M-step preconditioned conjugate gradient method for parallel computation. 1983.

[2] Loyce Adams. M-step preconditioned conjugate gradient methods. *SIAM Journal on Scientific and Statistical Computing*, 6(2):452–463, 1985.

[3] Joy Bose, Steve B Furber, and Jonathan L Shapiro. A spiking neural sparse distributed memory implementation for learning and predicting temporal sequences. In *Artificial Neural Networks: Biological Inspirations–ICANN 2005*, pages 115–120. Springer, 2005.

[4] Sergio Davies, Cameron Patterson, Francesco Galluppi, Alexander D Rast, David Lester, and Steve B Furber. Interfacing real-time spiking i/o with the spinnaker neuromimetic architecture. In *Proceedings 17th International Conference, ICONIP 2010.*, pages 7–11. Australian Journal of Intelligent Information Processing Systems, Vol. 11, No. 1,, 2010.

[5] Andrew Brown Steve Furber and a host of others... Spinnaker..., 2008.

[6] S Furber, D Lester, L Plana, J Garside, Eustace Painkras, Steve Temple, and A Brown. Overview of the spinnaker system architecture. 2012.

[7] Steve Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society interface*, 4(13):193–206, 2007.

[8] V Galiano, H Migallón, V Migallón, and J Penadés. Gpu-based parallel algorithms for sparse nonlinear systems. *Journal of Parallel and Distributed Computing*, 72(9):1098–1105, 2012.

[9] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[10] Magnus Rudolph Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems, 1952.

[11] http://apt.cs.man.ac.uk/projects/SpiNNaker/. Official website of the spinnaker project.

[12] MM Khan, DR Lester, Luis A Plana, A Rast, X Jin, E Painkras, and Stephen B Furber. Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856. IEEE, 2008.

[13] Muthucumaru Maheswaran, Kevin J Webb, and Howard Jay Siegel. Mcgs: a modified conjugate gradient squared algorithm for nonsymmetric linear systems. *The Journal of Supercomputing*, 14(3):257–280, 1999.

[14] Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis A Plana, and Steve Furber. Understanding the interconnection network of spinnaker. In *Proceedings of the 23rd international conference on Supercomputing*, pages 286–295. ACM, 2009.

[15] Dianne P O'Leary. Parallel implementation of the block conjugate gradient algorithm. *Parallel Computing*, 5(1):127–139, 1987.

[16] L. Olson. Lecture 21 better iterative methods; google and markov chains, 2009.

[17] Luis A Plana, Steve B Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. A gals infrastructure for a massively parallel multiprocessor. *Design & Test of Computers, IEEE*, 24(5):454–463, 2007.

[18] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[19] Alexander D Rast, Javier Navaridas, Xin Jin, Francesco Galluppi, Luis A Plana, Jose Miguel-Alonso, Cameron Patterson, Mikel Luján, and Steve Furber. Managing burstiness and scalability in event-driven models on the spinnaker neuromimetic system. *International Journal of Parallel Programming*, 40(6):553–582, 2012.

[20] Jeff Reeve. Jeffs lectures, 2012.

[21] PC Rosenbloom. The method of steepest descent. In *Proc. of Symp. in Applied Math*, volume 6, pages 127–176, 1956.

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

[23] Thomas Sharp, Luis A Plana, Francesco Galluppi, and Steve Furber. Event-driven simulation of arbitrary spiking neural networks on spinnaker. In *Neural Information Processing*, pages 424–430. Springer, 2011.

[24] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[25] Marcin Wozniak, Tomasz Olas, and Roman Wyrzykowski. Parallel implementation of conjugate gradient method on graphics processors. In *Parallel Processing and Applied Mathematics*, pages 125–135. Springer, 2010.

[26] Laurence Tianruo Yang and Richard P Brent. The improved conjugate gradient squared (icgs) method on parallel distributed memory architectures. In *Parallel Processing Workshops, 2001. International Conference on*, pages 161–165. IEEE, 2001.

[14] [22] [4] [23] [9] [18] [24] [16] [15] [2] [13] [26] [1] [8] [25] [10] [21] [7] [11] [12] [5] [17] [6] [3] [20] [19]