

Parallel Computation in Spiking Neural Nets

Andrew Carnell and Daniel Richardson,
Computer Science, Bath University,
Bath BA2 7AY, UK
email: masdr@bath.ac.uk

March 13, 2007

Abstract

Numerical quantities can be represented as phase differences between equiperiodic oscillating subsystems in a spiking neural net. It is then possible to represent integer variables, and the increment and decrement operations, $X := X + 1$, $X := X - 1$. It is possible to represent the *if* construction, the *while* construction, and some other programming language constructions, including variants of the Seq, Par, and Alt constructors, which were used in Occam. We give a general purpose parallel programming language with integer variables which can be systematically implemented in spiking neural networks. Addition, subtraction and multiplication are done, albeit inefficiently, as examples.

1 Introduction

A great deal of work is currently going on to design and build spiking neural nets as nanostructures in silicon. See, for example, the recent paper by A. Bindal and S. Hamed-Hagh [1]. In connection with this, and the ongoing efforts to understand the computational powers of biological neurons, as reviewed, for example, in the paper by A. Herz et al [7], it seems important to understand the potential computational power of artificial spiking neural networks.

In her book of 1999, H. Siegelman shows that the family of analog neural nets with saturated sigmoid transition function, considered as computing devices, are Turing complete. See [10]. This means that any computation which can be carried out by a Turing machine with its potentially unbounded memory can also, in principle, be carried out by finite networks of these analog devices. This is possible because the neurons hold real numbers, each of which may contain an unbounded amount of information. The essence of Siegelmann's construction is to consider a real number as a stack of binary values, via the usual binary

decimal expansion. J. Neto, H. Siegelmann and J. F. Costa have also shown that a general purpose parallel programming language, similar to Occam, can be compiled into one of these sigmoid neural nets.

The case of spiking neural nets is somewhat different, because here the messages sent between neurons are series of stereotyped spikes. The only information carried by a spike is the time at which it occurs. We may, it is true, look at a spiking network over a long period of time and record the rate of firing, averaged over some time window, of each neuron at each instant. This rate coding transformation from one model to the other is not entirely convincing, since the number of possible values represented by a rate would depend on the size of the time window, but nevertheless it does suggest that spiking neural nets ought also to have the property of Turing completeness. This was actually shown by W. Maass in 2001. See [5]. In this case real numbers, within a certain interval, are represented as phase differences of oscillating systems. That is, the information is held in the dynamics of the system, not directly as in the work of Siegelmann. As before, however, the essential idea is to use the real numbers as binary stacks.

Both of these constructions give more than Turing completeness. They actually implement the real number machines of Blum, Smale and Shub. See [2]. Of course the implementations are in neural nets considered as dynamical systems, as mathematical abstractions, and not in actual networks of real neurons.

The W. Maass construction also depends on some special assumptions about the postsynaptic response function, the most important one being that it is linear on some interval.

The results of this article are closely related but the methods are different. Our main contribution is a way to represent integer variables. The natural number n is represented by taking two in-phase copies of oscillating subsystems and performing n standardised perturbations of one of them. To read the number back, the subsystem which was not perturbed is perturbed some number of times until the two subsystems are again in phase, as they were at the beginning. To support this idea, we require some device, made out of spiking neural nets, to detect whether or not two oscillators are in phase. This is called a coincidence detector. Hoping to make some concession to realism, we do not assume unbounded precision accuracy of the coincidence detector. And therefore we can only compute with integer values of a bounded size, as with ordinary computers. It is shown that a weak version of Occam can be implemented in this way. As an example, it is shown how addition and multiplication could be implemented, albeit inefficiently. We do not need the assumption of linearity in an interval of the post synaptic response function, as used by W. Maass. We do assume that the post synaptic response function is continuous, that it starts from a resting value, increases to a maximum and then decreases back to the resting value.

2 Oscillations and Spiking Neurons

The spiking neurons and spiking neural networks discussed in this article should be understood as members of the family of deterministic mathematical models, such as leaky integrate and fire, or the spike response model, discussed and described by Gerstner and Kistler. See [3].

Numerical quantities can be represented as phase differences between equiperiodic oscillating subsystems in a spiking neural net. It is then possible to represent integer variables, and the increment and decrement operations, $X := X + 1$, $X := X - 1$. It is also possible to represent some basic parallel programming constructions: if, while, seq, par, alt, as used in, for example, the Occam programming language. We do not claim that computations of this kind are actually done in this way in biological systems.

3 Terminology

We will say that a network of spiking neurons is an oscillator if it has some initial state so that, without any further input, every neuron in the network will fire periodically, possibly with different periods. It is quite possible for an oscillator to have other behaviours, for other initial states, even without any input. An oscillating orbit is only determined by a network and a designated initial state. We will say that an oscillating orbit is attractive if any initial state sufficiently near to the designated initial state results in an orbit which spirals in to the oscillating orbit.

Attractive oscillators are very common in spiking neural networks. For example, a single neuron with an excitatory link to itself and a time delay will oscillate, if the weight is large enough, and the oscillation is attractive, with rapid convergence, for most values of the parameters. Another example of this type would be a chorus, which is a symmetric network of identical neurons, totally connected by excitatory links, all with the same weights and time delays. For discussion, see [3]. Such oscillators can be started from the zero state by giving an initial excitatory impulse. The periods can be adjusted or altered, to some extent, by altering the weights and the time delays.

We will say two oscillating subsystems are matched if there is a one to one correspondence between the neurons and links of the subsystems, preserving all the parameters of the neurons and the links.

Two matched oscillating subsystems which are started together will be in phase forever. If one is started later than the other, the phase difference will be preserved forever. It is these phase differences which we will use as elements of memory. We note that in case there is noise on the links, the phase difference will drift at random over time. To some extent, such random drift can be slowed

by duplication of systems.

In the following, we will advance the phase of an oscillating system by sending an exciting impulse to it. This depends on the idea that if a neuron is about to fire in the near future, a small exciting impulse will cause the firing to occur earlier. This in turn depends on some assumptions about the post synaptic response. We assume, in fact, that this post synaptic response is continuous, that it starts from a resting state, increases to a maximum, and then decreases back to the resting state. (These assumptions are simpler and somewhat different from those used in [5], where strict continuity is not needed, but some intervals of linearity are required). The continuity is important for our construction, since it allows fine tuning of the response. We consider that this is a realistic assumption, but note that some event based simulation systems actually depend on the contrary assumption of an instantaneous and discontinuous initial jump of the response function in order to simplify their computations. See [6], [8].

In the section below we define a simple concurrent programming language. A program in such a language is just a statement. These statements will be computed in spiking neural nets.

A subsystem is a network of spiking neurons. We will assume that each subsystem has an initiating neuron and a terminating neuron. The subsystem is started by sending an exciting impulse to the initiating neuron. The subsystem finishes, if ever, at the first instant after it starts when the terminating neuron fires. The activity between the start and the finish is called a process. Of course the process depends on the initial state of the subsystem. We do not always assume that the initial state of a subsystem is zero. We intend to use these processes to implement statements in the programming language. Of course observing the process should tell us all we need to know about the computation which is supposed to be described by the original statement. At this point we will just make the minimum assumption, which is that the process should terminate, given some initial conditions, if and only if the statement in its usual denotational semantics describes a terminating computation. In case of termination, we also expect the terminating state of the variables (described below) to be correct, according to the denotational semantics, provided that none of the integral values which occur in the computation are too large.

4 A Simple Parallel Programming Language

The language described below is a very weak version of Occam , without channels, and only integer variables, and all variables global. See [9].

We will use strings of characters starting with upper case letters as variables for integers.

The language consists of statements.

If X is a variable, then $X := 0$, $X := X + 1$, and $X := X - 1$ are statements. If X and Y are variables, then $X := Y$ and $X := -Y$ are statements. These are all called assignment statements.

If X is a variable, then $X = 0$ and $X \neq 0$ and $X > 0$ are tests.

If T is a test and P and Q are statements then

If T then P else Q

is a statement.

If T is a test and P is a statement, then

While (T) P

is a statement.

We will say that two statements are independent if no variable which may be modified in one statement is referred to in another.

If P_1, \dots, P_n are statements then

$Seq\{P_1; P_2; \dots P_n; \}$ is a statement,

If P_1, \dots, P_n are independent statements then

$Par\{P_1; P_2; \dots P_n; \}$, and $Alt\{P_1; P_2; \dots P_n; \}$ are statements.

Statements in this language have a natural denotational semantics, defined as follows. We define a valuation to be an assignment of integral values to some subset of the variables. Then each statement denotes a possibly non deterministic, possibly non terminating, transformation from one valuation to another. These denotations can be defined by structural recursion on the statements, as usual. So, for example, the transformation denoted by $Par\{P_1; P_2; \dots P_n; \}$ is obtained by running the transformations denoted by P_1, P_2, \dots, P_n in parallel, and it only terminates when and if all of them terminate. On the other hand, the transformation denoted by $Alt\{P_1; P_2; \dots P_n; \}$ is obtained by running all of the denoted transformations in parallel, and it terminates just when one of its constituents terminates, the choice being non deterministic if several of them terminate. See [4] for discussion of nondeterministic choice.

We show below how the transformations denoted by statements in this programming language can be implemented in spiking neural networks.

5 Representation of Variables for Integers

We define a switch to be a pair of oscillators, each inhibiting the other. So if (A, B) is a switch, it has at least three steady states: A oscillating and B quiet; both A and B quiet; A quiet and B oscillating. There should be a large number

of time delayed inhibiting links between A and B to prevent them from both oscillating together. The simplest case of a switch would be a pair of neurons, A and B , each neuron exciting itself, and the pair connected by many time delayed inhibiting links. A switch can have one pole or the other turned on, or both turned off. We note that within the context of the deterministic model, there is no fading memory; whatever state a switch is in will persist until further input is given.

We define a variable X to be a switch, (A_0, B_0) together with an array of matched oscillators $(A_1, B_1), \dots, (A_n, B_n)$. For each i , the oscillators A_i and B_i are matched, and may oscillate independently, but the oscillators A_1, \dots, A_n are all distinct, with different periods.

Our convention will be that when the value of a variable is positive, the A_0 side of the switch will be on; and when the value of the variable is negative, the B_0 side of the switch will be on.

We will say that a variable $(A_0, B_0), (A_1, B_1), \dots, (A_n, B_n)$ is exactly zero if (A_0, B_0) are both off, and if, for each $i > 0$, A_i and B_i are in phase.

Very small phase differences may not be observable. Let us agree that we can observe whether or not a phase difference is below ϵ for some $\epsilon > 0$.

We will say that a variable $(A_0, B_0), (A_1, B_1), \dots, (A_n, B_n)$ is observably zero, with tolerance ϵ , if (A_0, B_0) are both off, and if, for all $i > 0$, A_i and B_i have phase difference below ϵ .

A variable is observably non zero with tolerance ϵ if it not observably zero with tolerance ϵ .

A subsystem which decides, with high probability, whether or not a variable is observably zero (with some tolerance ϵ) will be called a coincidence detector. As mentioned in [3], there are actual biological systems of this type, with quite small values of ϵ , in the barn owl for example.

Let X be a variable, represented by $(A_0, B_0), (A_1, B_1), \dots, (A_n, B_n)$. As mentioned above, we use the convention that when X is positive A_0 will be on, and when X is negative B_0 will be on, and when X is zero both A_0 and B_0 will be off, i.e. not firing. We will call A_1, \dots, A_n the positive part of the variable, and B_1, \dots, B_n the negative part of the variable. Incrementing the variable will mean incrementing all the positive parts, and decrementing the variable will mean incrementing all the negative parts, and then modifying the switch if necessary. One oscillator, A_i , will be incremented by sending it one exciting impulse, thereby advancing its phase.

It follows that, in order to ensure that each increment has the same effect, input spikes must be ‘injected’ into each neuron of the oscillator at the same point in the period of the neuron. Obviously another structure is required to ensure that this is the case. We shall call this structure the synchroniser. The task of the synchroniser is to ensure that the effect of an increment is independent of

the time when the increment is done. A synchroniser is described below for the simple case in which the oscillator is just one neuron connected to itself. The idea is to hold the incrementing impulse and to deliver it, with a small time delay, when the neuron itself fires.

The natural number n is represented by incrementing zero n times, and resetting the switch to indicate that n is positive. In general, after an increment or decrement, the switch may need to be reset appropriately.

Let (A, B) be the simplest matched pair of oscillators, each a single neuron self exciting. Let the weight for this exciting link be w , and the time delay d . The period τ of the spiking depends on w , decreasing as w increases. Suppose an increment advances the phase of A by δ . We must have $\delta > \epsilon$, since the result of one increment must observably change the phase. Assume $\delta > 2\epsilon$. After some number $N(\epsilon, \tau)$ of increments the matched pair will again appear in phase, with tolerance ϵ . After about τ/δ increments there is probability of about $2\epsilon/\delta$ of appearing to be in phase. So, roughly speaking, we could expect $N(\epsilon, \tau)$ to be about $\tau/(2\epsilon)$. That is the expected memory capacity of one matched pair. Since all the matched pairs in a variable have different parameters, and therefore different periods, we expect that the memory capacity of an array of matched pairs would be approximately the product of the memory capacities of the individual pairs.

Let X and Y be variables. The statement $X := Y$ could be implemented by a process which, when initiated, would first turn off all the oscillators of X , and then take input from all the oscillators of Y and pass it on, with the same time intervals, to the corresponding oscillators of X , and would then terminate. We will consider that the subsystem implementing $X := Y$ contains the neurons of X , and takes input from the neurons of Y , but does not contain them. So the subsystems implementing $X := Y$, and $Z := Y$ could be disjoint, even though both take input from the neurons of Y . So we would say that the independent statements $X := Y$ and $Z := Y$ could be done in parallel by disjoint subsystems.

6 Experiments in CSIM

6.1 An Introduction to CSIM

CSIM (neural Circuit SIMulator), is a neural network simulation tool that allows the simulation of networks of different models of neurons and synapses. It has a multitude of user modifiable properties for both synapses and neurons. These range from models such as the basic integrate and fire neuron, to more sophisticated models involving ion channel modeling. Similarly, synapses can be simple static spiking synapses or dynamic spiking synapses which allow the effects of synaptic plasticity to be accounted for.

All CSIM scripts are written and executed in Matlab. It has been used exten-

sively in work by Maass and is designed and written by the LSM Group of the Institute for Theoretical Computer Science at the University of Graz.

We have used CSIM to build examples of the structures described in this paper.

More detailed information concerning the usage and capabilities of CSIM can be found at WWW.LSM.TUGRAZ.AT/CSIM/INDEX.HTML.

6.2 Single Neuron Oscillator

First it was necessary to show that it is possible to create a stable oscillator which will oscillate with constant period forever if stimulated and then left undisturbed. The structure of a single neuron oscillator can be seen in figure 1 below.

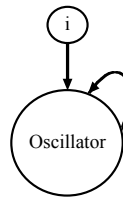


Figure 1: A Single neuron Oscillator with spiking input neuron

6.2.1 Results for a Single Neuron Oscillator in CSIM

Figure 2 shows the spiking output of a single neuron oscillator created with CSIM, using a leaky integrate and fire neuron.

6.3 The Synchroniser

Consider a matched pair of oscillators, (A, B) , as described earlier. A single input spike of sufficient magnitude to one of the oscillators would cause a phase advancement of that oscillator, after which the previous spiking rate is quickly resumed. However, there now exists a permanent phase difference between A and B .

The function of a synchroniser is to ensure that the introduction into an oscillator of a phase advancement spike will only ever occur at or very near to a specific point in the period of the oscillator. This ensures that every input into the oscillator facilitates a near-identical response - preserving the predictability of the system.

If we were allowed to introduce a phase advancement spike at any instant during the period of an oscillator then the instant of spiking of that oscillator would

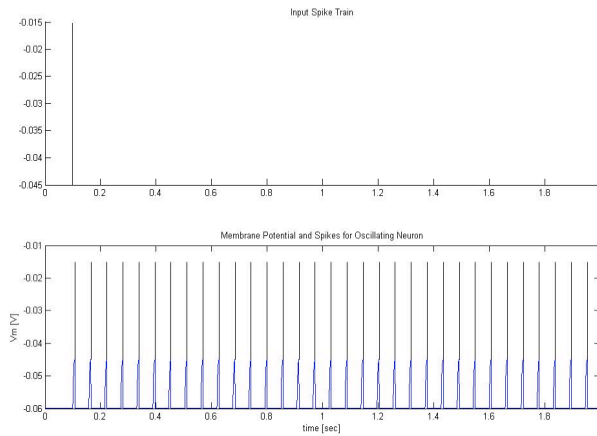


Figure 2: Oscillator Output

differ for each phase advancement spike. See figure 2 below - where the third input spike was ignored completely as it arrived while the oscillator was spiking and was consequently swamped.

6.3.1 Structure of a Synchroniser

Figure 4 illustrates a possible design for the synchroniser.

The input neuron i emits a spike which will produce a phase advancement of an oscillator A . This spike is relayed to the oscillator via a combination of two structures, the first of which is a switch, the second is known as the Accumulator. A precondition for the correct functioning of the synchroniser is that the switch has been in the 'off' state for a certain period of time before the spiking of the input neuron.

6.3.2 Elements of a Synchroniser: The Switch.

The function of the Switch is to indicate the presence of an input spike destined for the oscillator. The 'on' side of the switch has a single, time-delayed excitatory connection from the input neuron i , the weight of which is sufficient to cause spiking, and it has a connection to itself so that the spiking continues periodically. On the other hand, the 'off' side receives multiple inhibitory time delayed connections from the 'on' side - each of which have the same weight but

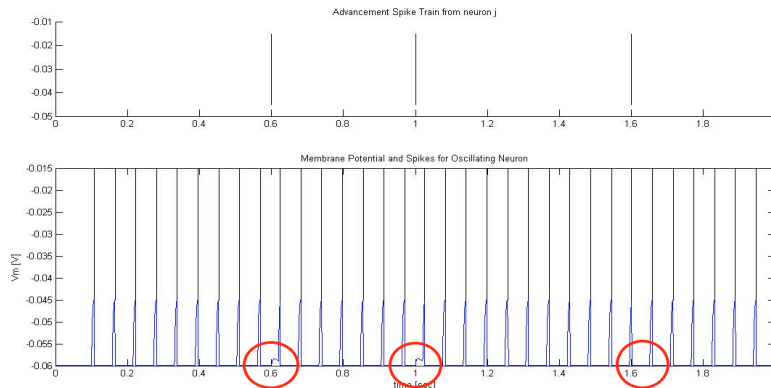


Figure 3: Spike injection without a synchroniser

where the modulus of one of these weights is much smaller than the weight of the excitatory connection to the ‘on’ side from the initiating neuron. The effect of these multiple inhibitory connections is to temporarily lower the internal state of the ‘off’ side oscillator by a certain amount so that it is unable to continue to spike.

When the Accumulator neuron eventually fires, the switch will also receive multiple inputs from the Accumulator neuron, with the ‘on’ side of the switch receiving multiple time delayed inhibitory connections and the ‘off’ side receiving a single, time-delayed excitatory connection, reversing the switch.

Once the ‘on’ side of the switch has been stimulated by i it will spike regularly and continually until the accumulator neuron spikes and sends it multiple inhibitory spikes to shut it down.

There are also mutual inhibitory links between both sides of the switch. This is to ensure that when one side is active the other is inhibited.

It will be necessary to prevent any scenario in which both the ‘on’ and ‘off’ sides of the switch are active simultaneously. This can be accomplished by further delaying the connections denoted in figure 4 as the ‘on’ and ‘off’ side excitatory links. Consider the case where the ‘off’ side of the switch is active and i emits a spike. We require that the ‘off’ side be inactive by the time ‘on’ is made active by the spike from i . Therefore we add a delay to the synaptic connection from i to ‘on’, while instantly relaying the multiple inhibitory spikes from i to ‘off’. The result is that the internal state of the ‘off’ oscillator will be lowered by an amount that will prevent it from its next firing so that when the excitatory pulse arrives at ‘on’, the ‘off’ side will be quiet. The delay is such that the single pulse arrives after the influence of the mutual inhibitory links has subsided.

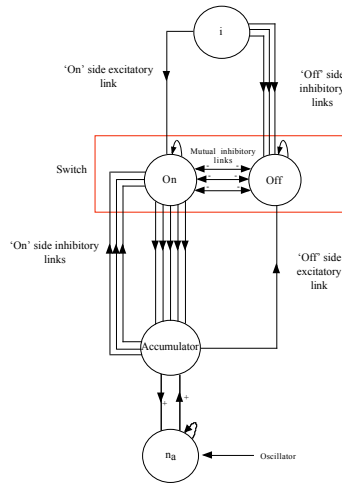


Figure 4: The Synchroniser

6.3.3 Elements of a Synchroniser: The Accumulator neuron.

Once the presence of an input destined for the oscillator has been detected by the switch, it will be necessary for the synchroniser not only to ‘remember’ that there is an outstanding input which needs to be injected into the target oscillator, but also be able to relay a pulse into the oscillator at the appropriate instant.

The accumulator neuron received input from the switch and also from the oscillator A , as shown in figure 4). Suppose that the internal state function of the accumulator is $N(s)$. We may write $N(s) = O(s) + A(s)$, where $O(s)$ is the response to the oscillator and $A(s)$ is the response to the inputs from the switch. Since we suppose that the oscillator has been nearly periodic for a while, it follows that $O(s)$ is nearly periodic with the same period. We suppose that the link or links between the oscillator and the accumulator have been set so that $O(s)$ is not only periodic with the same period as the oscillator, but also has just maximal point in between each firing of the oscillator. Let θ be the threshold of the accumulator. Pick the weights so that the maximum value of $O(s)$ is $\theta/2 + \delta$, where δ is some small value, well below $\theta/2$. $O(s)$ might have some other local maxima which are smaller than this but these must all be below $\theta/2$. The time interval in which $O(s) > \theta$ can be made as small as we like by choosing δ sufficiently small. When the switch is off, the accumulator will not fire, since $O(s)$ will not cross the threshold.

When the switch is on, we arrange so that $A(s)$ rises to a plateau of approximately $\theta/2 + \delta$ and oscillates around this with amplitude below δ .

This is accomplished by having n time delayed synapses which connect the ‘on’

side of the switch to the Accumulator neuron. The time delays are $j\tau/n$, for $j = 1 \dots n$, where τ is the period of the 'on' side of the switch.

The weights of these connections are all equal to some value w , which is determined so that the resulting cascade of pre-synaptic pulses has the effect of charging the Accumulator neuron so that $A(s)$ reaches the desired plateau. The more connections there are, the more closely will the course of $A(s)$ approximate to the ideal of perfect flatness.

The Accumulator will only spike if it has received a sequence of inputs from the 'On' side of the switch and it subsequently receives an input from the oscillator. At this point the Accumulator neuron 'injects' the spike into the oscillator, and sends inhibitory spikes to the 'On' oscillator followed by an excitatory spike to the 'Off' oscillator. The charging of the accumulator from the switch can be seen in figure 5.

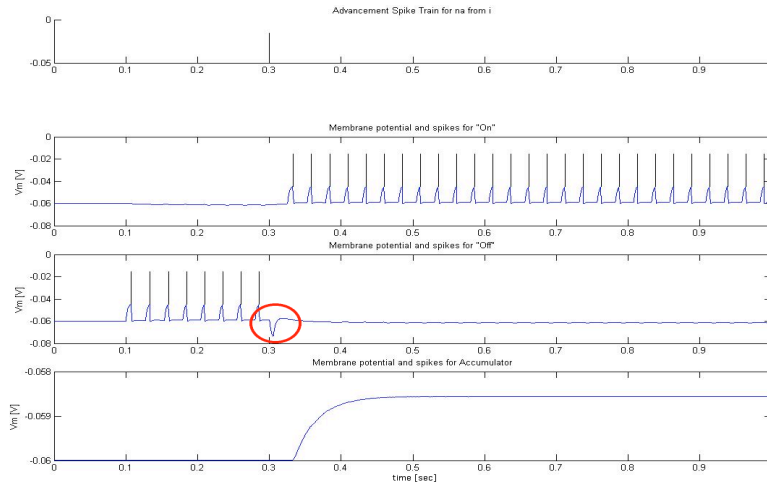


Figure 5: The Charging of the Accumulator

6.3.4 Results for the Synchroniser in CSIM

Figure 6 shows the behaviour of the synchroniser as a whole, while figure 7 shows the injection of the three spikes as shown before in figure 3, but this time utilising the synchroniser. It can be seen that with the synchroniser the spike that was ignored previously is now injected successfully.

Suppose that the synchroniser has been turned on, and that a spike from the oscillator occurs at time t , and a subsequent spike from the accumulator occurs at time $t + \Delta$. The value of Δ will depend on the phase of the oscillator.

We have that $A(t + \Delta) + O(t + \Delta) = \theta$. Let the maximum value of $O(s)$ be

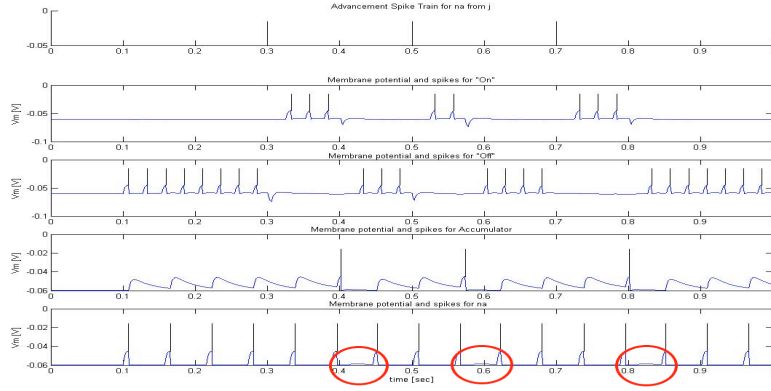


Figure 6: Synchroniser Behaviour

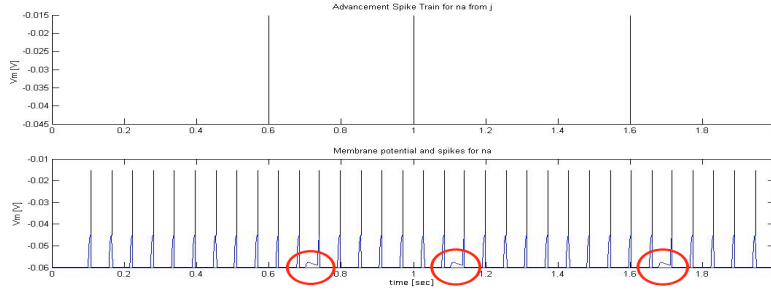


Figure 7: Spike Injection with a Synchroniser

$\theta/2 + \delta$. So $A(t + \Delta) \geq \theta/2 - \delta$. On the other hand, $A(s)$ is bounded from above by $\theta/2 + \delta$. We have

$$\theta/2 - \delta \leq A(t + \Delta) \leq \theta/2 + \delta$$

Let t^* be the maximum point of $O(s)$ which is nearest to $t + \Delta$.

The quantity $|t + \Delta - t^*|$ is bounded by the time it takes the value $O(s)$ to go from $\theta/2 - \delta$ to its maximum value $\theta/2 + \delta$ and then back down to $\theta/2 - \delta$; and this tends to zero with δ since $O(s)$ is continuous. Therefore the time between $t + \Delta$ and the next subsequent spike of the oscillator is approximately constant.

When we advance the phase of the oscillator, the phase of $O(s)$ will follow, keeping approximately constant the distances between maximum points of $O(s)$ and the time of the next subsequent spike of the oscillator.

6.4 The Coincidence Detector

The function of a coincidence detector is to indicate if two or more spikes, from two or more separate inputs, are ‘simultaneous’ events. In any real system it is expected that simultaneity can not be observed with perfect accuracy. Therefore, we define two spikes as being effectively simultaneous if they both arrive within a time interval of size ϵ , where ϵ is small.

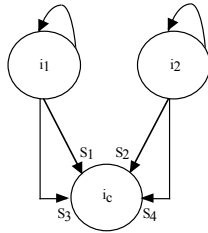


Figure 8: The Coincidence Detector

It is expected that the coincidence detector will be highly susceptible to variations in the synaptic weights of each input. It will be necessary to choose the weights with particular care, as they will determine how selective the detector is at classifying two inputs as effectively simultaneous.

Consider a coincidence detector which is a single spiking neuron i_c that has two excitatory synaptic inputs s_1 and s_2 and two (slightly) time delayed inhibitory synapses s_3 and s_4 , as shown in figure 8. Synapses s_3 and s_4 have the effect of sharpening the edge of the input spikes allowing the coincidence detector to be more selective.

If the threshold of i_c is given by θ , and the synaptic weights of the two inputs are given by w_1 and w_2 , where $w_1 = w_2 = w$ we set w so that

$$\theta = (2 \times w) - x$$

As x tends towards 0, ϵ will also tend towards 0. By adjusting the parameters, we can make ϵ as small as we desire.

6.4.1 Results for the Coincidence Detector in CSIM

In order to perform accurate operations with phase differences it is necessary that the integrity of a number stored in a pair of oscillators can be guaranteed.

Figure 9 shows the increment oscillator being perturbed 10 times by a phase advancement spike. This is effectively storing the number 10 in the pair. 10 spikes are then injected into the decrement neuron until both oscillators are

in phase once again. The number of spikes required for this is 10. Therefore, integrity has been demonstrated in this case.

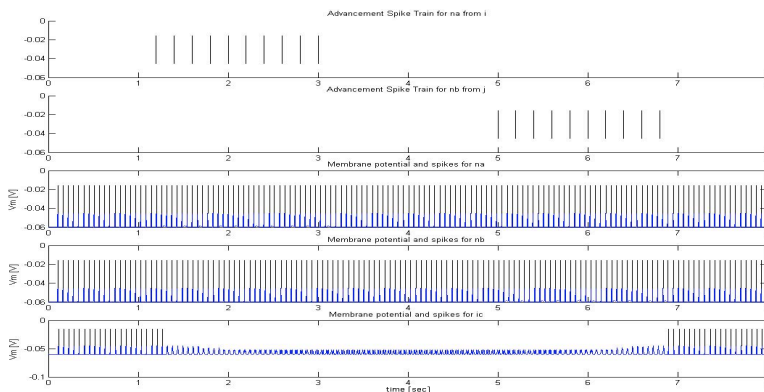


Figure 9: Integrity Test

7 IF, WHILE, SEQ, PAR, ALT constructions

Given structures such as the oscillator, synchroniser and coincidence detector it is possible, using combinations of these structures, to implement the basic constructions of the programming language described above.

7.1 The IF constructor

Consider some group of spiking neurons which perform some process P . We are not concerned with the function or the mechanics of the process, only that it is implemented using spiking neurons and synapses, and that it can be triggered by a spike arriving from an input neuron i and that upon completion it activates some termination neuron which, effectively passes control to the next constructor.

χ is a structure of spiking neurons which represents the condition of the IF statement which, must be satisfied if P is to be activated. χ has an internal switch, not shown in the diagram, which is normally off. When this switch is off, neither output of χ can fire. Upon firing, the neuron i sends a pulse to the input neuron of χ , which then turns on χ by resetting its switch. Whether or not the condition is satisfied will determine which output neuron of χ fires. If χ is satisfied neuron y is activated, if χ is not satisfied then neuron n is activated. In either case, when χ finishes, it is switched off. Neuron y will send a spike to the initiating neuron of the process P that, upon completion will activate the

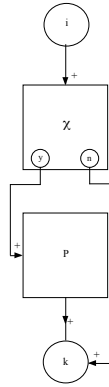


Figure 10: Example of the IF Constructor

termination neuron of the constructor k . Neuron n will immediately activate k , as the condition for P to be activated has not been met.

A precondition for this to work correctly is that χ is turned off before the initiating neuron fires.

7.2 The WHILE constructor

This is similar in structure to the IF statement. The WHILE constructor must first check if a condition χ is met and then, if it is, allow a process P to be activated. The cycle continues until the condition represented by χ is no longer met. If at any point χ is not met when tested, then the WHILE constructor passes control to the next process. A possible implementation is shown in figure 11.

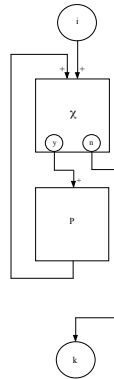


Figure 11: Example of the WHILE Constructor

Again χ is the condition of the WHILE statement, P is the process to be run while this condition is met.

Neuron i is an activation neuron. A spike emitted by i will activate χ . Upon completion P emits a spike to the input neuron of χ . If the condition χ is being met then neuron y is activated, otherwise neuron n is activated.

Neuron y is connected to the activation neuron of the process P . If neuron n is triggered it will activate the constructor termination neuron k , passing control to the next constructor.

7.3 The SEQ constructor

The function of the SEQ constructor is to allow a series of processes to activate sequentially, with the next process in the list only being activated once the current process has completed.

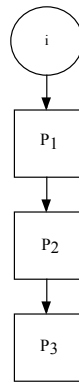


Figure 12: Example of the SEQ Constructor

Neuron i is the activation neuron of the constructor. $P_1, P_2, P_3 \dots P_n$ is the sequence of processes to be activated. Assume that each of these has been implemented by disjoint subsystems. After each process is completed only then will it send an activation spike to the next process in the sequence. As figure 12 shows, i sends an activation spike to P_1 , which in turn sends an activation spike to P_2 , which sends an activation spike to P_3 , which upon its completion will send a spike that activates the next constructor.

7.4 The PAR constructor

The PAR constructor activates a list of processes concurrently. Assume, as before, that each of these has been implemented by subsystems which are disjoint.

Control only passes from a PAR constructor once *all* of the listed processes have completed.

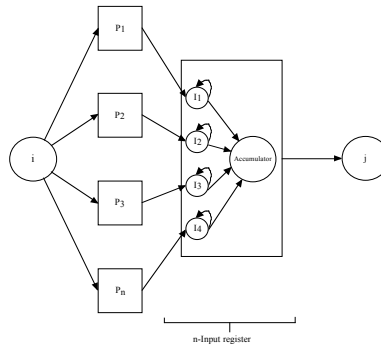


Figure 13: Example of the PAR Constructor

Once again i is the activation neuron of the constructor. When i spikes, the spike gets transmitted to the input of each of the processes to be activated. As each process completes it will send a spike to its specific Input neuron n within an n input register, see figure 13. The weights of this switch are set, such that a constant charging from each of the n inputs is required to cause activation of the accumulator. The mechanisms for charging the accumulator are similar to those described in the previous section on the synchroniser, the only difference being that we now utilise multiple accumulator input oscillators. The activation of the accumulator causes a spike to be sent to the constructor termination neuron j , which passes control to the next constructor.

7.5 The ALT constructor

The ALT constructor (essentially nondeterministic choice) is given a set of processes implemented on disjoint subsystems, runs them concurrently, and terminates when one of the processes terminates. So the initiating neuron of the ALT sends an initiating impulse to all of the initiating neurons of its constituent processes, and a spike from any of the terminating neurons of any of the constituent processes is sufficient to trigger a spike from the terminating neuron of the ALT. After the ALT terminates, it should turn itself off.

8 Examples: Addition, Subtraction and Multiplication

We can define addition, subtraction and multiplication over the integers in the language given earlier, using the recursive definitions starting from the successor

function.

```
Seq{
  Z:=Y;
  while Z \= 0
  Par{
    Z:= Z-1;
    X:= X+1;
  };
}
```

defines $X := X + Y$ if Y is non negative. Subtraction can be done similarly for Y non negative. Addition over the integers can be done as:

```
if Y > 0 then
Seq{
  Z:=Y;
  while Z \= 0
  Par{
    Z:= Z-1;
    X:= X+1;
  };
}
else
Seq{
  Z:= Y;
  while Z \= 0
  Par{
    Z:= Z+1;
    X:= X-1;
  };
}
```

To define $X := X * Y$, for Y non negative we can use

```

Seq{
  W:= Y;
  Wx:= X;
  X:= 0;
  while W \neq 0
  Par{
    X:= X+Wx;
    W:= W-1;
  };
}

```

and then we can extend to the integers as was done for addition.

The inner Par constructions in the above statements could be changed to Seq constructions.

9 Decidability and Complexity

If we had a perfect coincidence detector, i.e. sharp thresholds, and a perfect synchroniser, we could represent arithmetic for unbounded integers with a finite network. We could make such a network search for solutions to diophantine problems, which, as a class, are unsolvable. In the deterministic model therefore, long term behaviour would be expected to be undecidable. In fact even in very simple networks in the deterministic model, long term behaviour could be undecidable. This could be formalised using, for example, the Blum Shub Smale real number machine. See [2]. If we add some noise to the links so that the fading memory concept of Maass and Markram applies, we may lose the undecidability, but we would still expect that in a practical sense the long term behaviour of these systems would not be feasible to compute.

10 Other Types

10.1 Pointers and associations

It should be clear that it is also possible to represent pointers using networks of spiking neurons. Consider, for example, a pair of oscillators (A_p, B_p) . We can

say that this points to another pair (A_q, B_q) if B_p is in phase with A_q . In this way we can represent networks of associations. An attractive example of how such ideas can be used can be found in the work of D. Wang [12] where interactions of networks of oscillators are used to compute geometric and topological properties.

11 Conclusion and Discussion

The discussion above and the experiments we have done with CSIM suggest the possibility of a compiler which would take a statement in a simple parallel programming language such as the one given above and would write a script in for example the CSIM language which would construct a spiking neural network which would enact the computational meaning of the statement. Instead of training spiking neural nets to perform given tasks, we are constructing them, as is also done in [5]. There are already examples of such compilers from Occam-like languages into sigmoid neural networks. See [11].

The implementation of addition, subtraction and multiplication which we give, for purposes of exposition, is very inefficient computationally. However, using the same basic ideas it should be clear that we can implement other, more efficient, algorithms for arithmetic. For example, we could implement a variable as an array or pairs of oscillators in such a way that when one of the pairs has been advanced as far as possible it returns to zero and the next pair is advanced by one step, as in the usual decimal notation. We could then build in an addition and multiplication table, and proceed, as we did, with effort, in school.

References

- [1] A. Bindal, S. Hamedi-Hagh, The design of a new spiking neuron using dual work function nanowire transistors, *Nanotechnology* 18, 095201, pp 1-12, 2007
- [2] L Blum, F. Cucker, A. Shub, S. Smale, *Complexity and Real Computation*, Springer-Verlag, 1997
- [3] W. Gerstner, W. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*, Cambridge University Press, 2002.
- [4] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985
- [5] W. Maass, Lower Bounds for the Computational Power of Networks of Spiking Neurons, *Neural Computation*, 8(1), pp 1-40, 1996
- [6] Delorme A, Gautrais J, Van Rullen R, Thorpe S (1999) SpikeNET: A simulator for modelling large networks of integrate and fire neurons. *Neurocomputing*, 26-27, pp989-996.

- [7] A.V. M. Herz, T. Gollish, C. K. Machens, and D. Jaeger, Modelling Single-Neuron Dynamics and Computations: A Balance of Detail and Abstraction, *Science*, vol 314, 2006
- [8] M. Mattia and P. Del Giudice, Efficient event-driven simulation of large networks of spiking neurons and dynamical synapses, *Neural Computation*, 12, 2305-2330, 2000
- [9] Inmos Limited (1998), *Occam2 Reference Manual*, Prentice-Hall International, Hemel Hempstead
- [10] H. Siegelmann, *Neural Networks and Analog Computation, Beyond the Turing Limit*, Birkhauser 1999
- [11] J. P. Neto, H. Siegelmann, J. F. Costa, Symbolic processing in neural networks, *Journal of the Brazilian Computer Society*, vol 8, no 3, pp 58-70, 2003.
- [12] D.L Wang, On Connectedness, A Solution Based on Oscillatory Correlation, *IEEE Transactions on Neural Networks*, vol. 11, 935-947, 2000.