Pragmatic Many-Core Computing

Interim report

Steve Furber and Andrew Brown – 30th April 2012

Summary

This interim report presents the progress of the work at the 6-month point, which is primarily towards Goal 1 in the proposal, identifying exemplar applications and demonstrating programming methodologies using the SpiNNaker platform.

Table of Contents

1 Research Abstract and Goals (from the proposal project summary):

How do we use a million processor cores in a single application?

Many-core computer systems present several challenges to the programmer, among which are:

- Identifying promising applications and developing pragmatic programming methodologies;
- Deciding how best to map the components of a computation onto the available processor resources;
- Debugging the executing program.

The SpiNNaker project (funded by EPSRC, the UK funding agency for engineering and the physical sciences) has supported the development of an experimental multi-core platform that offers a unique environment for research into many-core programming. SpiNNaker employs an architecture inspired by the very high levels of connectivity found in the human brain, and the primary objective of the research has been to develop a generic platform for real-time brain modeling. The resulting machine is a mesh of Multi-Processor Systems-on-Chip (MPSoCs) that will be scaled up over 2011 to 2012 within the current funding to machines with up to a million processor cores.

Although designed for real-time neural modeling applications, the SpiNNaker machine is simply a massively-parallel energy-efficient computer with a bespoke inter-processor communications fabric. As such is it suited to applications outside the neural domain that have appropriate characteristics, principally that they can be broken down into a very large number of small processes with well-defined and slowly-changing interactions between the processes. Identifying suitable applications with these characteristics is the first objective.

> ***Goal 1:*** Identify exemplar applications for many-core systems, and demonstrate programming methodologies using the SpiNNaker platform.

The component mapping problem – deciding which process should run on which processor – is central to SpiNNaker and to many-core systems in general. The proposed research will investigate the mapping problem for a range of different applications, using SpiNNaker as the experimental test-bed, with the goal of minimizing the energy consumed by the computation. The work will proceed in three phases:

1. Extrinsic static mapping: The mapping problem is solved externally to the many-core machine, and the mapped program is loaded and run on the many-core machine with a fixed interconnect pattern.
2. Intrinsic static mapping: The problem is solved within the many-core machine itself, leading to a fixed interconnect pattern for the execution phase.
3. Intrinsic dynamic mapping: The problem is solved within the many-core machine, leading to an initial interconnect pattern, but the interconnect pattern may change as the program runs.

In the first year of the work, the first phase (extrinsic static mapping) will be investigated.

> ***Goal 2:*** An analysis, and experimental evaluation using SpiNNaker, of alternative algorithms for mapping fine-grain parallel applications onto a many-core computer system.

Debugging parallel applications is difficult; real-time massively-parallel applications on a system such as SpiNNaker present an extreme problem for debugging and visibility in general.

> ***Goal 3:*** Develop visualization and debugging methodologies for many-core systems, contribute to emerging standards for mega-core machines, and demonstrate these on SpiNNaker.

2 Introduction to the SpiNNaker machines

The SpiNNaker project's architecture mimics the human brain's biological structure and functionality. This offers the possibility of utilizing massive parallelism and redundancy to provide resilience in an environment of unreliability and failure of individual components.

In the human brain, communication between its computing elements, or neurons, is achieved by the transmission of electrical "spikes" along connecting axons. The biological processing of the neuron can be modeled by a digital processor and the axon connectivity can be represented by messages, or information packets, transmitted between a large number of processors which emulate the parallel operation of the billions of neurons comprising the brain.

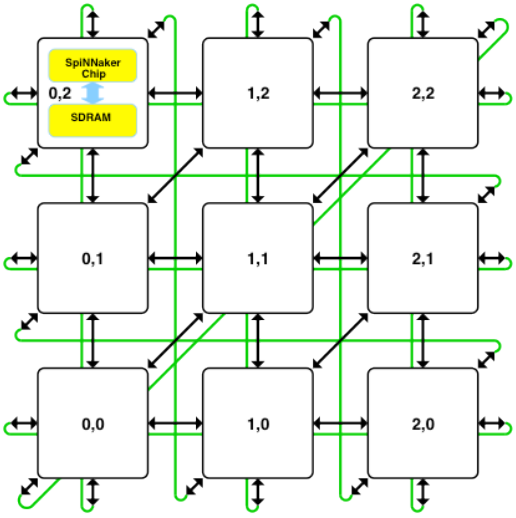


Figure 2.1: System diagram

The engineering of the SpiNNaker concept is illustrated in the figure 2.1 where the hierarchy of components can be identified. Each element of the toroidal interconnection mesh is a multi-core processor known as the "SpiNNaker Chip" comprising 18 processing cores. Each core is a complete processing sub-system with local memory and a DMA capability. It is connected to its local peers via a Network-on-Chip (NoC) which provides local high bandwidth communication and to other SpiNNaker chips via links between SpiNNaker chips. In this way the massive parallelism extending to thousands or millions of processors is possible.

The knowledge content and learning ability of the brain is embodied in its evolvable interconnection pattern; this routes a spike generated by one neuron to others which are interconnected with it by axons and these interconnections are modified and extended as a result of the learning and processes.

In SpiNNaker a packet Router within each multi-core processor controls the neural interconnection. Each transmitted packet representing a spike contains information which identifies its source neuron; this is used by a multi-core processor's Router to identify whether this packet should be routed to one of its contained application processors to respond, or should be routed on to one of the six adjacent multi-core processors connected to it as part of the overall SpiNNaker network.

2.1 The SpiNNaker chip

The basic building block of the SpiNNaker machine is the SpiNNaker multi-core System-on-Chip. The chip is a Globally Asynchronous Locally Synchronous (GALS) system with 18 ARM968 processor nodes residing in synchronous islands, surrounded by a lightweight, packet-switched asynchronous communications infrastructure.

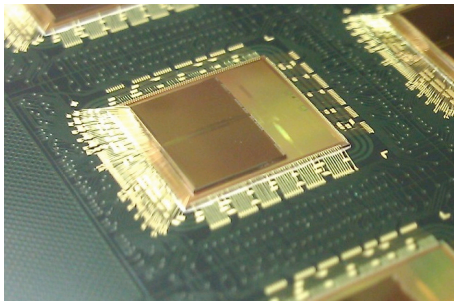


Figure 2.2: SpiNNaker package

Figure 2.2 shows that each SpiNNaker package contains two silicon dies: the SpiNNaker die itself and a 128 MByte SDRAM (Synchronous Dynamic Random Access Memory) die, which is physically mounted on top of the SpiNNaker die and stitch-bonded to it.

The micro-architecture assumes that processors are 'free': the real cost of computing is energy. This is why we use energy-efficient ARM9 embedded processors and Mobile DDR (Double Data Rate) SDRAM, in both cases sacrificing some performance for greatly enhanced power efficiency.

Figure 2.3 shows a plot of the SpiNNaker die, with the 18 identical processing subsystems located in the periphery, and the Network-on-Chip and shared components in the centre. At start-up, following self-test, one of the cores is elected to a special role as Monitor Core and thereafter performs system management tasks. Normally, 16 cores are used to support the application and one is reserved as a spare for fault tolerance and manufacturing yield-enhancement purposes.

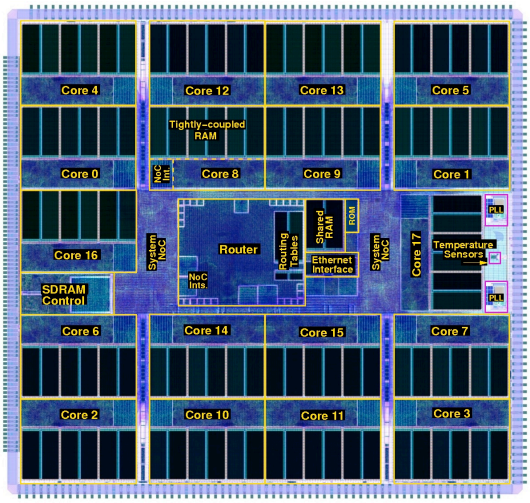


Figure 2.3: SpiNNaker die plot

Inter-processor communication is based on an efficient multicast infrastructure inspired by neurobiology. It uses a packet-switched network to emulate the very high connectivity of biological systems. The packets are source-routed, i.e., they only carry information about the issuer and the network infrastructure is responsible for delivering them to their destinations.

The heart of the communications infrastructure is a bespoke multicast router that is able to replicate packets where necessary to implement the multicast function associated with sending the same packet to several different destinations.

SpiNNaker chips have six bidirectional, inter-chip links that allow networks of various topologies. Inter-chip communication uses self-timed channels, which, although costly in wires, are significantly more power efficient than synchronous links of similar bandwidth.

The SpiNNaker die area is 102 mm^2 (10.386 mm × 9.786 mm). It was originally taped-out in December 2010. The first batch of fully functional packaged chips was delivered on May 20th, 2011.

2.2 SpiNNaker machines

SpiNNaker machines are classified by the (approximate) number of processor cores, thus the 10N machine has approximately 10^N processor cores. The 102 and 103 machines are single printed circuit boards, already available or in the final stages of design. The larger machines are racks or cabinets and specifications are subject to change.

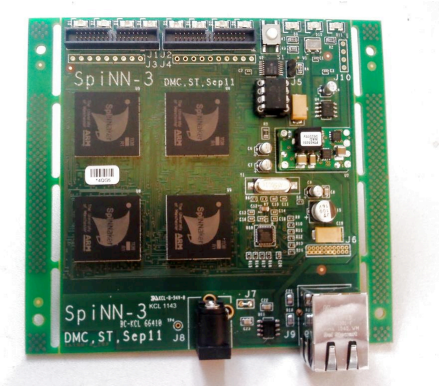


Figure 2.4: 102 machine PCB

102 machine. The 102 machine is the 4-node circuit board and hence has 72 ARM processor cores, which will typically be deployed as 64 application cores, 4 Monitor Processors and 4 spare cores. The 102 machine requires a 5V 1A supply, and can be powered from some USB2 ports. The control and I/O interface is a single 100Mbps Ethernet connection. There is limited provision for connecting cards together with SpiNNaker links to form larger systems.

103 machine. The 103 machine is the 48-node board and has 864 ARM processor cores, typically deployed as 768 application cores, 48 Monitor Processors and 48 spare cores. The 103 machine requires a 12V 6A supply. The control interface is two 100Mbps Ethernet connections, one for the Board Management Processor and the second for the SpiNNaker array. There are options to use the six on-board 3.1Gbps high-speed serial interfaces (using SATA cables, but not necessarily the SATA protocol) for I/O; this will require suitable configuration of the on-board FPGAs that provide the high-speed serial interface support. 103 boards can be connected together to form larger systems using the high-speed serial interfaces. This is the basis for the construction of the larger machines described below.

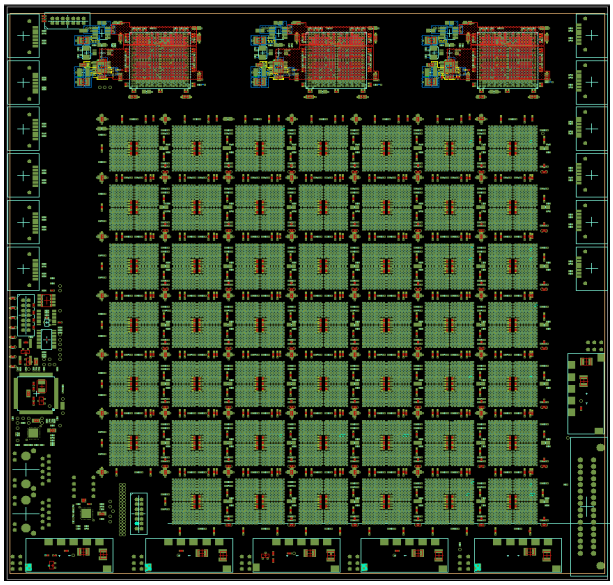


Figure 2.5: 103 machine PCB plot

104 machine. The 104 machine will be a single card frame incorporating 12 48-node cards. It will have 10,368 ARM processor cores, typically deployed as 9,216 application processors, 216 Monitor Processors and 216 spares. The 104 machine will be the largest portable SpiNNaker machine and is intended to operate in a desk-top environment from a standard mains supply. The card frame will incorporate a ~1kW power supply for the SpiNNaker cards and a 24-way 100Mbps Ethernet switch to offer a 1Gbps Ethernet interface to the host machine. The machine will include fan cooling for the power supply, the SpiNNaker boards and the Ethernet switch. Three of the six SATA interfaces on each board are used to interconnect the boards into a toroidal mesh. The remaining three per board are available for high-speed I/O, which will require suitable configuration of the on-board FPGAs.

105 machine. The 105 machine will be a 19" rack cabinet incorporating 5 card frames each with 24 48-node cards. It will have 103,680 ARM processor cores, typically deployed as 92,160 application processors, 2,160 Monitor Processors and 2,160 spares. This machine will require a 10kW (approx.) 240V supply and an air-conditioned environment. The control and I/O interfaces are to be defined.

106 machine. The 106 machine will comprise 10 19" rack cabinets each similar to the 105 machine. It will have 1,036,800 ARM processor cores, typically deployed as 921,600 application processors, 21,600 Monitor Processors and 21,600 spares. This machine will require a 100kW (approx.) 240V supply and an air-conditioned environment. The control and I/O interfaces are to be defined.

3 SpiNNaker software development flow

3.1 System Software Components

The SpiNNaker run-time software involves three different devices:

- **Host machine** – for application I/O and monitoring.
- **SpiNNaker monitor cores** – for application support and system monitoring. Additionally, one of them communicates with the host over Ethernet.
- **SpiNNaker application cores** – to run applications.

Figure 3.1 shows the SpiNNaker run-time software components used to support applications:

- **ybug** – [host software] interacts with scamp on the monitor cores and provides a simple command/debug interface to start applications, examine memory and on-chip resources.
- **scamp** – [monitor core software] interacts with ybug on the host and sark on the application cores. Supports system-wide inter-processor communication and communication with the host.
- **sark** – [application core software] interacts with scamp on the monitor core and provides the ability to use the core hardware/peripherals in an abstracted way. For example, starting a 1ms timer, setting an entry in the multicast routing table or installing a handler to deal with packet arrival.

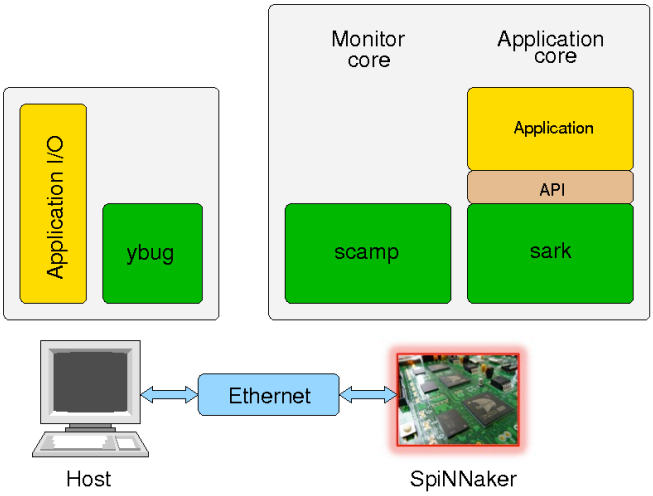


Figure 3.1: Run-time software

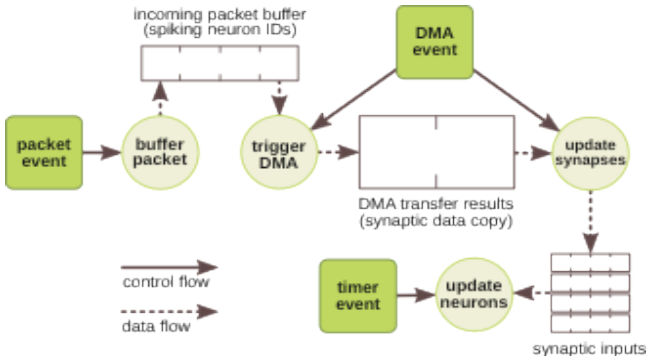


Figure 3.2: Event-driven software model

3.2 Application Programming Model

The SpiNNaker programming model is a simple, **event-driven model** (figure 3.2). Applications do not control execution flow, they can only indicate the functions, referred to as callbacks, to be executed when specific events occur, such as the arrival of a packet, the completion of a Direct Memory Access (DMA) transfer or the lapse of a periodic time interval. The Application Run-time Kernel (***sark***) controls the flow of execution and schedules/dispatches application callback functions.

Figure 3.3 shows the basic architecture of the

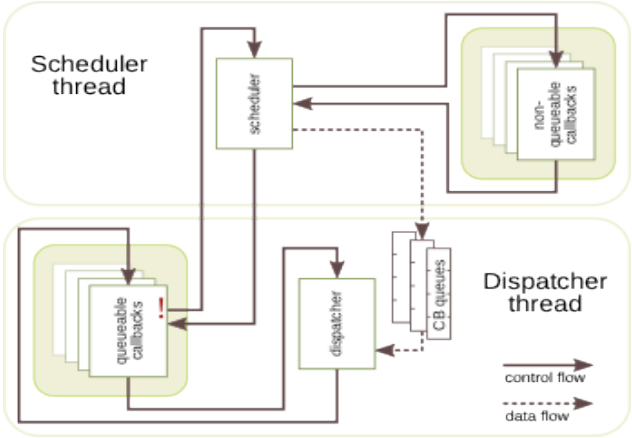


Figure 3.3: Event-driven software framework

event-driven framework. Application developers write callback routines that are associated with events of interest and register them at a certain priority with the kernel. When the corresponding event occurs the scheduler either executes the callback immediately and atomically (in the case of a non-queueable callback) or places it into a scheduling queue at a position according to its priority (in case of a queueable callback). When control is returned to the dispatcher (following the completion of a callback) the highest-priority queueable callback is executed. Queueable callbacks do not necessarily execute atomically: they may be pre-empted by non-queueable callbacks if a corresponding event occurs during their execution. The dispatcher goes to sleep (low-power consumption state) if the callback queue is empty and will be awakened by any event.

The SpiNNaker API supports the programming model providing functions to register callbacks, enter and exit critical sections, communicate with other cores and the host, trigger DMA operations and other useful tasks.

3.3 Neural application development

The primary objective of the neural application development framework is to enable non-expert users (such as neuroscientists and psychologists) to describe neural models in a familiar high-level network description language. The mapping of the high-level description onto the SpiNNaker machine will be automated, as far as is possible.

At present we have an automated flow for PyNN (figure 3.4), a high-level spiking neural network description language developed within the EU FACETS project. PyNN supports the description of networks in terms of populations (associated groups of neurons) and projections (connections from one population to another). The SpiNNaker Partitioning And Configuration MANager (PACMAN) maps populations to processors (dividing large populations across several processors) and constructs all of the tables and data structures required for SpiNNaker to run a real-time model of the network.

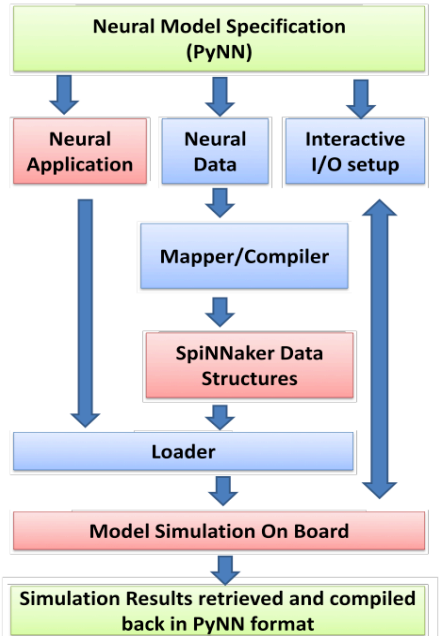


Figure 3.4: Neural application development

Future plans include developing a similar automated development flow for NEST, and possibly other neural network description languages such as NeuroML and NineML. These languages offer differing degrees of flexibility in the way neural networks may be described. NineML, for example, allows the neuron spiking behaviour to be specified by the user in the form of mathematical equations, which would have to be compiled into SpiNNaker fixed-point numerical functions.

4 Neural applications

4.1 Spiking neural applications

SpiNNaker has been optimized throughout for the efficient modeling of large-scale systems of spiking neural networks in biological real time. This optimization is manifest in the following principal aspects of the architecture:

- The SpiNNaker inter-processor communication fabric can handle very large numbers of small (40- and 72-bit) packets. The bisection bandwidth of the million-processor machine is 10 billion packets/s, which does not represent an especially high data rate for a supercomputer, but is an extremely high packet rate.
- The packet routing primitives include an efficient multicast protocol that enables the machine to support the very high connectivity found in the biological system
- The memory hierarchy is optimized for the data structures and access characteristics of the spiking neural model: frequently accessed neural data is held in tightly-coupled memory local to each processor; larger, less frequently-accessed synaptic data structures are held in the shared SDRAM.
- DMA engines are used to access the synaptic data structures in SDRAM, effectively hiding the latency of accesses to the SDRAM.

These mechanisms enable the ARM cores to operate at full performance, giving each SpiNNaker node (with 18 200MHz ARM cores) approximately the same compute performance on this task as an x86 PC processor, but with a power budget of 1W. Furthermore, the communications infrastructure yields a highly scalable architecture than can be extended up to 65k nodes and a million ARM processors in a single system.

The PyNN to SpiNNaker development flow is operational and has been used to develop small models (up to a few 10,000s of neurons) that run on the 4-node 102 machine – see for example, sample outputs from a small cortical model in figure 4.1. Similar models have been demonstrated in real-time robotics control, in simple vision applications, a control system for a virtual quad-rotor helicopter, and so on.

Developing the spiking neuron application portfolio in collaboration with partners who have suitably challenging models, and scaling the tools up to cope with the larger machines that will soon become available, is the central thrust of the SpiNNaker work over the next few years.

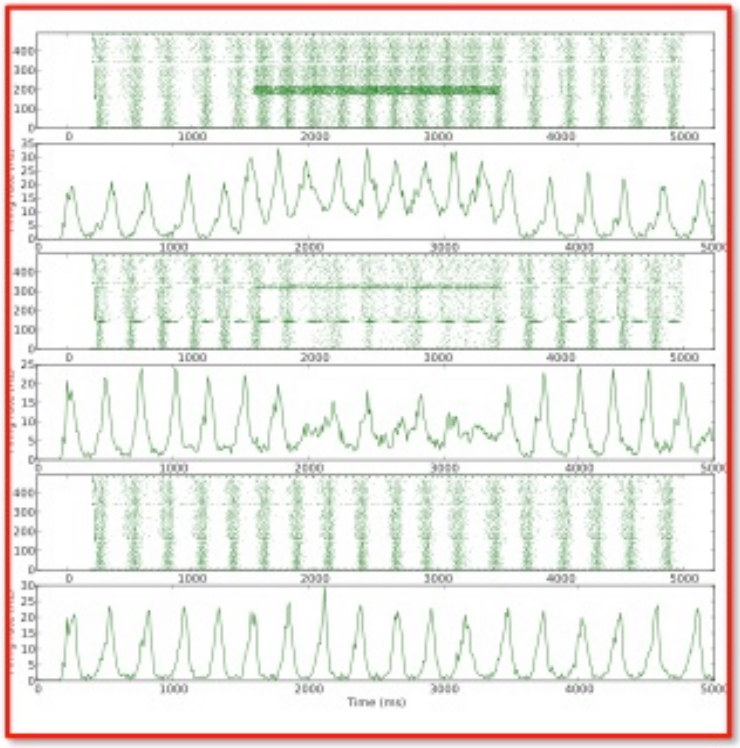


Figure 4.1: SpiNNaker spiking neuron outputs

4.2 MLP – matrix operations

The PDP2 project is implementing an entirely different kind of neural network: a non-spiking "classical" multilayer perceptron (MLP) model. By far the most popular type of computational neural network, the MLP consists of a number of neurons, (or "units") that compute some simple nonlinear threshold function on the sum of their inputs. Like spiking networks, connections between units are through synapses (or "weights") that are plastic and hence change weight while the data passes through the network. MLP models, however, have important differences. Typically weights are only plastic for a certain time, the "training phase" and then are fixed during the "test phase". Each phase consists of some number of examples, individual data items presented to the network. This in turn implies that the MLP has no time model, or to be exact, a discrete-time model. Contrasting strongly with spiking models, the MLP model thus has a synchronous dataflow. In addition, the most popular learning technique is back-propagation, a method that involves propagating errors from output units backwards through the network to update the weights. Both of these properties present challenges to the SpiNNaker architecture.

We have solved the synchronous dataflow problem using a new technique – "update-on-demand", that computes partial results and forwards them on through the network as soon as they are available. The back-propagation algorithm, which presents challenges for the source-routed SpiNNaker topology because it is effectively bidirectional, we solve using a matrix remapping method that distributes the processing between cores[1].

The SpiNNaker MLP implementation models networks using the Lens (http://tedlab.mit.edu/~dr/Lens) specification and will provide support for both feed-forward and recurrent networks. Initial support is provided for networks defined at a Group level; future plans may include support for Unit-level definitions. Full support is provided for Group-to-Unit and Unit-to-Unit connectivity. A SpiNNaker package plug-in for Lens allows Lens scripts to be implemented directly on SpiNNaker using the automated PACMAN path.

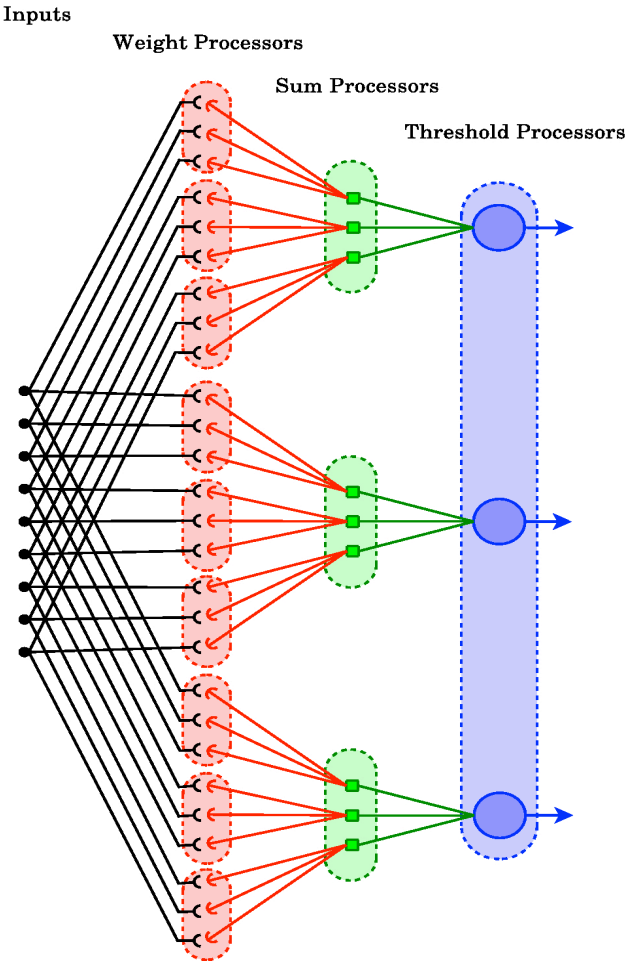


Figure 4.2: MLP matrix mapping

4.3 SpiNNaker MLP mapping.

In figure 4.2 each dotted oval is one processor. At each stage the unit sums the contributions from the previous stage. One processor may implement the input path for more than one final output neuron (shown here for the threshold stage).

[1] X. Jin, M. Lujan, M. M. Khan, L. A. Plana, A. D. Rast, S. Welbourne, and S. B. Furber, "Efficient Parallel Implementation of a Multi-Layer Back-propagation Network on Torus-connected CMPs", in Proc. ACM Conf. Computing Frontiers (CF'10), 2010, pp. 89-90.

5 Promising non-neural applications for many-core systems

Without exception, the contents of section 5 have not been quantitatively investigated. They are promising lines of enquiry, which are being pursued as resources permit, but have as yet to produced publishable results. Having said that, we are confident that SpiNNaker *is capable* of attacking these issues in a novel way; the only real research question is the execution time speedups that can be achieved.

5.1 SpiNNaker vs. conventional many-core systems

The anatomy of a *conventional* parallel program is shown in figure 5.1.

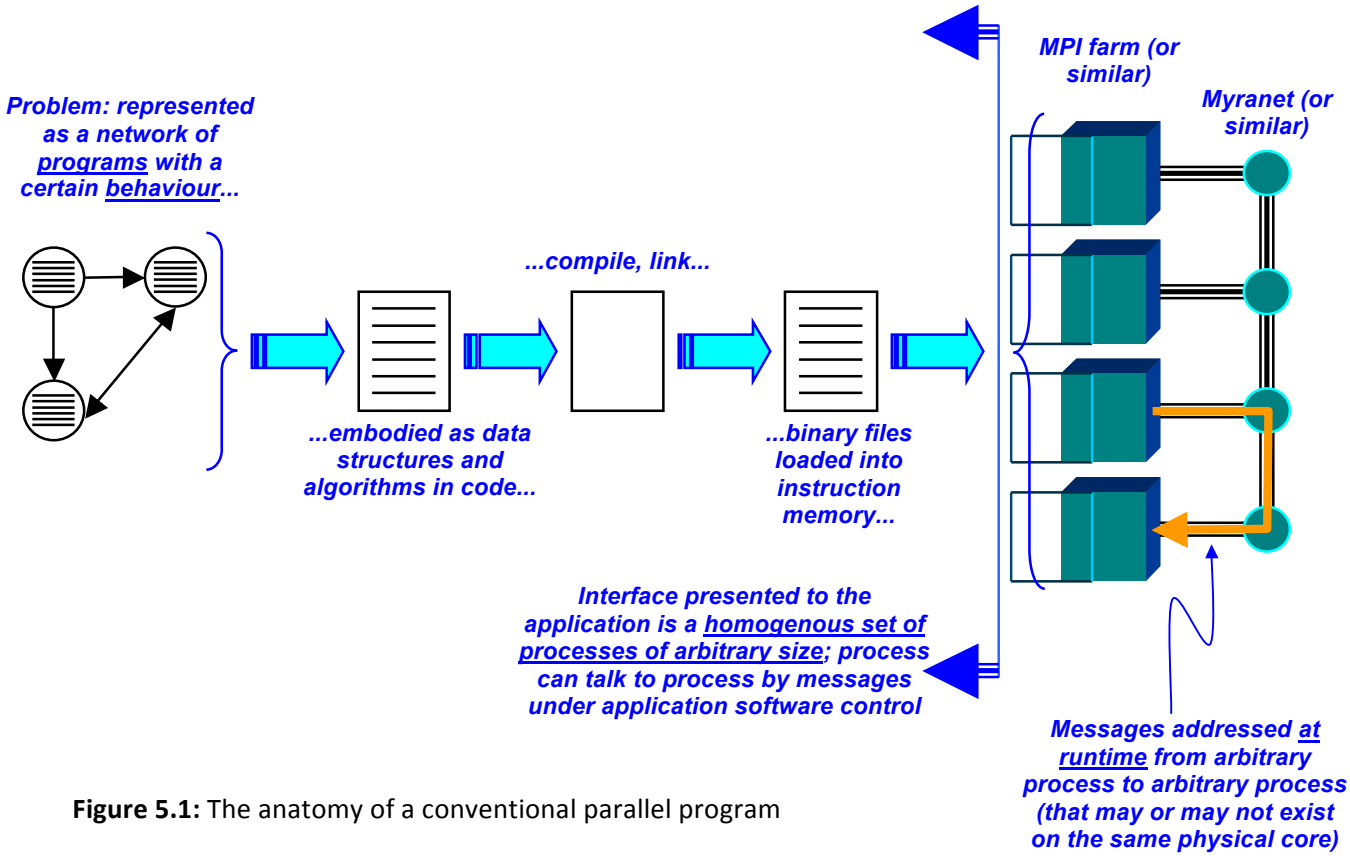


Figure 5.1: The anatomy of a conventional parallel program

The program designer can realistically expect a host of system level resources to be made available: blocking and non-blocking send/receive, message queue probing, message broadcasting, message scatter/gather, parallel I/O, remote process memory access, dynamic process management, console output, memory management, interactive debug, libraries, the notion of absolute time and barriers to name but a few.

The messages by which the independent process communicate are made up of an arbitrary number of units, which may themselves be a simple or compound chunk of data, the structure of which may be defined by the program designer.

The temporal cost of sending a message is usually a function of the message size: figure 5.2.

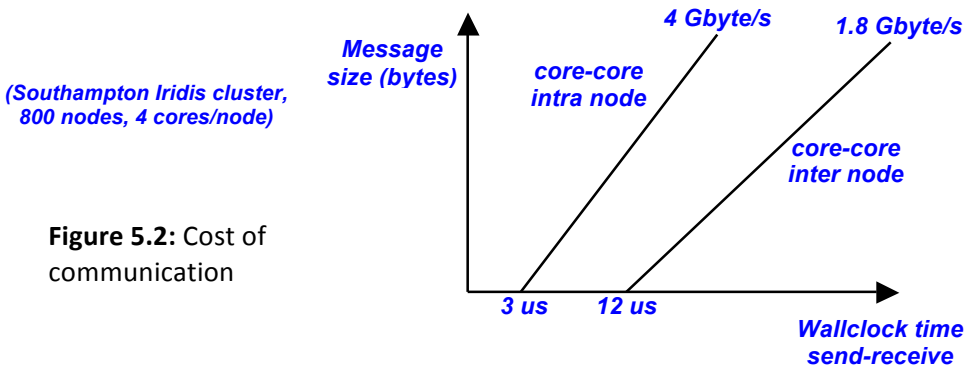


Figure 5.2: Cost of communication

The key point is that the user is running a set of arbitrarily complicated *programs*, the intercommunication of which may itself be arbitrarily complicated, and is choreographed by the program designer.

In contrast, the anatomy of a SpiNNaker-based parallel program is shown in figure 5.3. The structure of the problem is distributed throughout the route tables of the hardware – thereafter the potential routes of all the messages is considered fixed throughout the program execution. (It is possible to change the routing information during program execution, but this is – probably – an expensive task.)

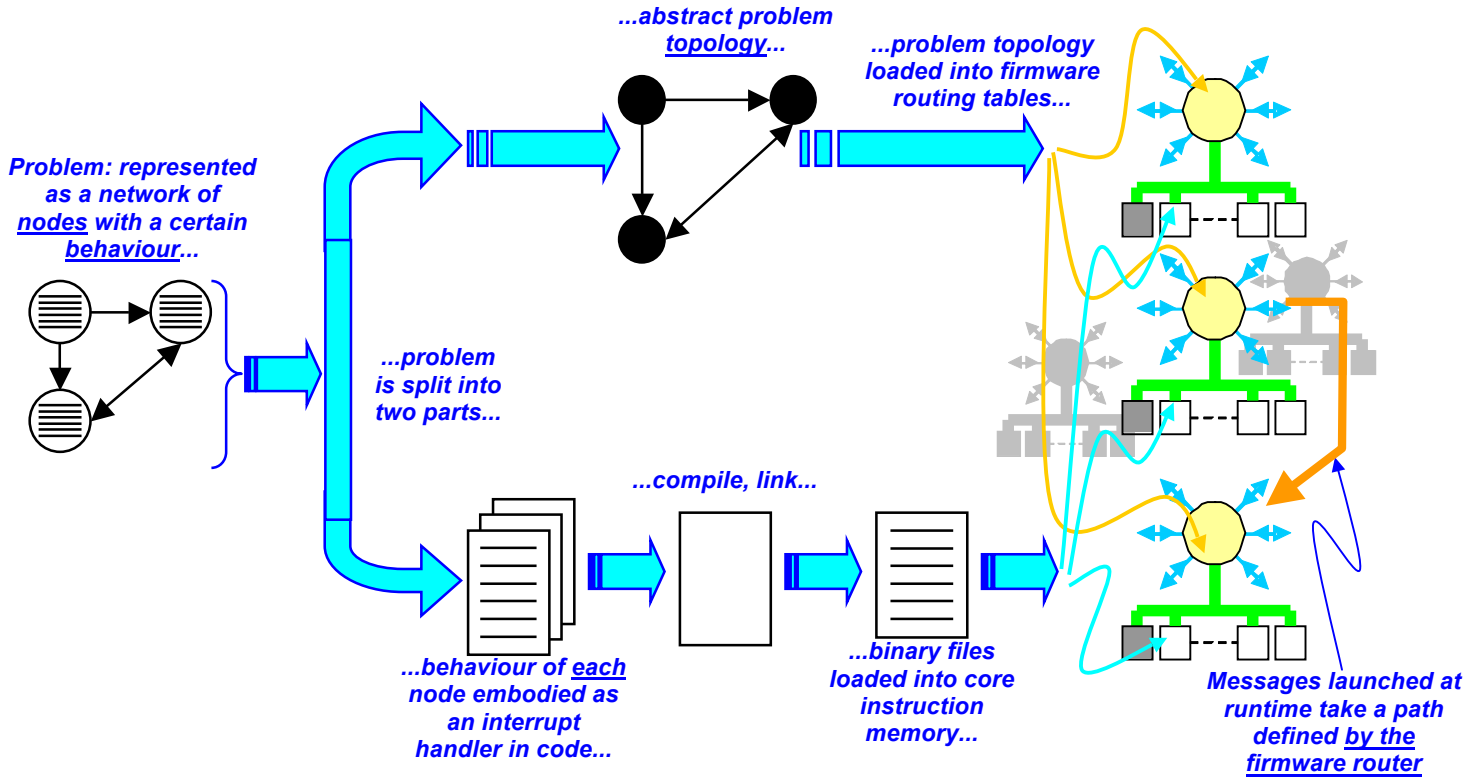


Figure 5.3: The anatomy of a SpiNNaker parallel program

As described earlier, an incoming message to a node causes an interrupt to be generated, which is handled by an appropriate (user supplied) fragment of code (the interrupt handler). This handler in turn may or may not cause consequent messages to be sent. The key point here is that a handler says "send message", but has no control where the output message goes (that information is distributed throughout the routing table) nor any idea where the message came from that woke up the handler.

Messages in SpiNNaker are small (a few bytes) and handed from node to node by the hardware routing engine. The approximate temporal cost of this is approximately 0.1us/hop, which gives an overall communication cost curve as shown in figure 5.4.

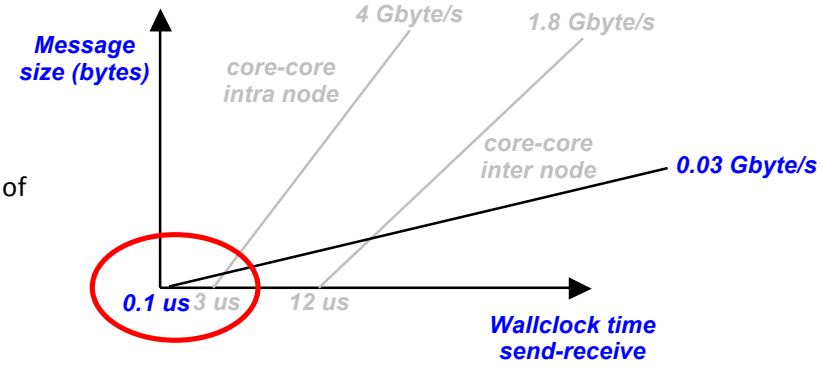


Figure 5.4: Cost of communication

SpiNNaker as a parallel engine has two significant advantages:

- The red area (highlighted) of figure 5.4 is where SpiNNaker wins in terms of message cost.
- A 'conventional' parallel supercomputer can cost of the order of £1-2k per core; SpiNNaker (to manufacture) costs around £1 per core.

5.1.1 Methodology (how promising applications are identified)

There is, as yet, no formal methodology for identifying applications whose solution might be amenable to the particular strengths of SpiNNaker. However, the informal thinking starts with consideration of the predicate graph of the problem, either in terms of the underlying physics, or the mathematical formulation of the solution.

5.1.2 SpiNNaker as a relaxation engine

Any physical system may be represented by a discrete graph. The vertices of the graph contain the physical attributes of the particular points they represent; the edges of the graph represent other physical quantities that flow between them. Table 5.1 contains a few examples of such systems:

Domain	Vertex	Vertex metric	Edge	Edge metric
Electronic	Electrical nodes	Voltage	Components	Current
Magnetic	*abstract*	Magnetomotive force	Magnetic path	Flux
Thermal	Physical point	Temperature	Physical distance	Heat flow
Translational	Force source	Displacement	Mechanical linkage	Force
Rotational	Torque source	Angle	Rotating shaft	Torque
Fluidic	Pressure source	Pressure	Pipe	Flow rate
Radiant	Radiation source	Illuminance	Physical distance	Radiation flux

Table 5.1: Decomposition of physical domains into nodes and flows

The accuracy of the approximation naturally depends on the granularity of the equivalent mesh. However, if the mesh is fine enough, the across/through relationships between adjacent vertices in the graph will become, to a suitable approximation, linear, and the interactions between them correspondingly simple. So simple, in fact, that the system solution may be obtained by allowing the vertex metrics to simply relax after any perturbation is imposed on the graph. Thus far, we could be using any conventional compute engine. A single thread machine would be spending all its time looping through pair-wise comparisons, and a conventional parallel engine would be spending ~30% of its wall-clock time handling messages.

SpiNNaker has compute nodes to spare, and a highly tuned message infrastructure that excels in passing small packets very fast. We can simply allocate *one processor to each problem grid point*, so the physical distribution of the problem graph amongst the SpiNNaker cores maps the physical reality of the actual problem.

As an example, consider the simple one-dimensional problem of figure 5.5. This could represent the temperature along a metal bar, or the voltage along a chain of resistors.

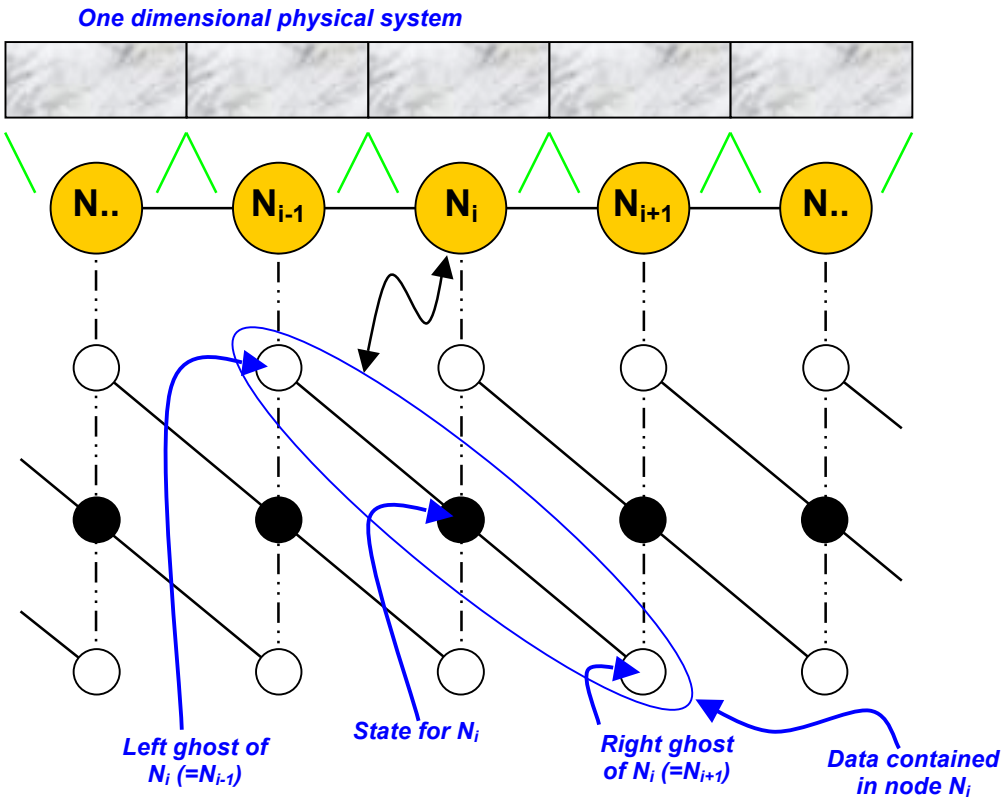


Figure 5.5: SpiNNaker as a relaxation engine

Each node contains a state (here a single scalar), plus the ghosts of the neighbouring node states. Each node is event-triggered; it sleeps until a message wakes it, it handles the message (which may or may

not cause it to emit messages of its own) and goes back to sleep. The key philosophy behind the "data-push" model is that a node only says something if it has something to say.

The event handler - running on every node - is something like this:

```
void ihr() {
                                    // Extract neighbour state
Recv(val,port);                     // and incoming stencil leg
if (port==LEFT) lghost = val;       // It WILL BE different, else
if (port==RIGHT) rghost = val;.......// the message wouldn't be here
oldtemp = mytemp;                   // Store current state
mytemp = (lghost+rghost)/2;         // Compute new value
if (oldtemp==mytemp) stop;          // If nothing changed, keep quiet
Send(mytemp,left);                  // Broadcast changed state
Send(mytemp,right);
}
```

This is the only code required to solve a million-point finite element representation of the one-dimensional bar; but it runs on a million processors *simultaneously*, exchanging messages that contain one variable value.

5.1.3 SpiNNaker as a dataflow engine

Behavioural synthesis is the process whereby a behavioural description of a system (typically, but not necessarily, a digital electronic system) is transformed automatically into a structural representation that exhibits the same functional behaviour.

As an example, consider figure 5.6. This shows a simple behavioural description of a system that multiplies two numbers together, and a direct translation of this description to structure. The figure contains details that are not germane to this discussion, but the essence is that the circuit is translated into a dataflow segment (left hand side) and control graph (right hand side). The control graph is effectively a one-hot clocked Petri net, the ***c2…*** labels referring to timeslots. As the controller cycles through a sequence of timeslots, the corresponding registers in the dataflow segment choreograph the flow of data through the functional units.

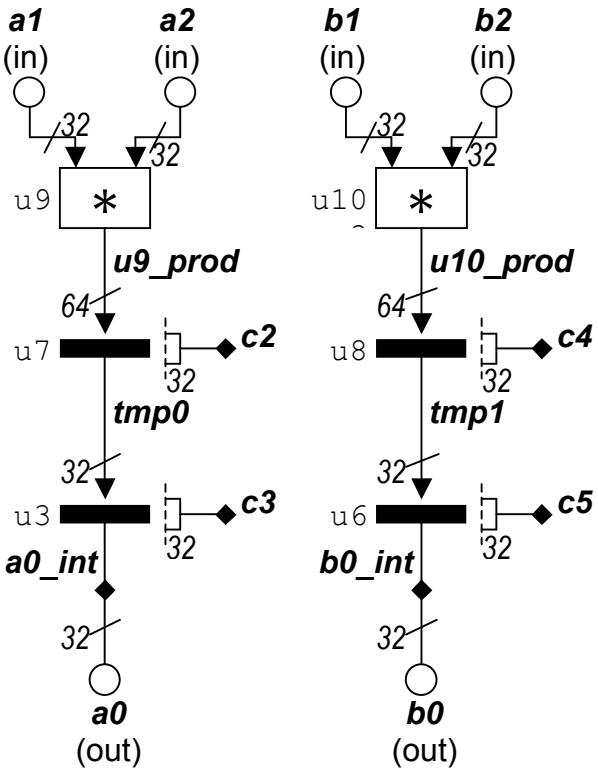
Figure 5.6 is a naive translation; many trivial optimizations are obvious by inspection:

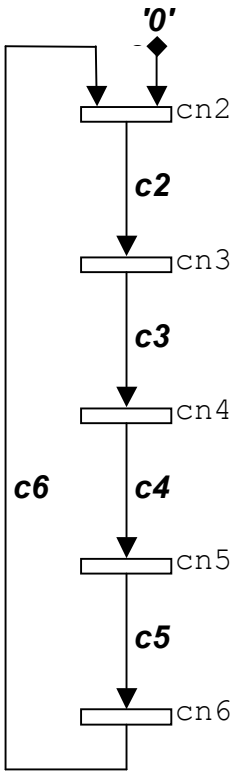
- To save time, the two multipliers can be clocked in parallel (discarding timeslots ***c2*** and ***c4***, and merging ***c3*** and ***c5***).
- To save area, the input signal pairs can be multiplexed onto a single physical multiplier, which takes twice the time but half the area of the previous optimization.

Whatever form the final circuit takes, it has to be realized in silicon, the two most obvious solutions being custom ASIC or FPGA.

FPGAs are becoming larger and more versatile, but are still fundamentally a sea of uncommitted functional units. The mapping of a circuit such as the one shown in figure 5.6 onto this sea of units is effected by programming the FPGA. Obviously the synthesis system requires knowledge of the portfolio of capabilities offered by the FPGA fabric, in order to make the mapping as sensible as possible.

SpiNNaker as a synthesis platform: If we treat SpiNNaker as an atomically programmable FPGA, we can break down the dataflow of our problem into much larger fragments, (typically single input/single output blocks of arbitrary complexity) and map each of these blocks onto a single SpiNNaker core.






```
entity optim is
  port(a1 : in  integer;
       a2 : in  integer;
       a0 : out integer;
       b1 : in  integer;
       b2 : in  integer;
       b0 : out integer);
end optim;

architecture behaviour of optim is

begin
doit: process
begin
  a0 <= a1*a2;
  b0 <= b1*b2;
  wait for 25 us;
end process doit;

end behaviour;
```

Figure 5.6: SpiNNaker as a simple dataflow engine

5.2 Promising applications

The following have been identified as potential non-neural application areas for SpiNNaker, although we have as yet no quantitative data on the performance of any of these. SpiNNaker is ultimately a collection of conventional cores; it can be *forced* to do anything that any ensemble of cores can. The issue we need to resolve is how *well* it can do these things.

5.2.1 Finite element systems

The relaxation example cited above showed a simple, highly idealized finite element system. In practice, of course, the mesh will be (at least) three dimensional and non-uniform - see figure 5.7.

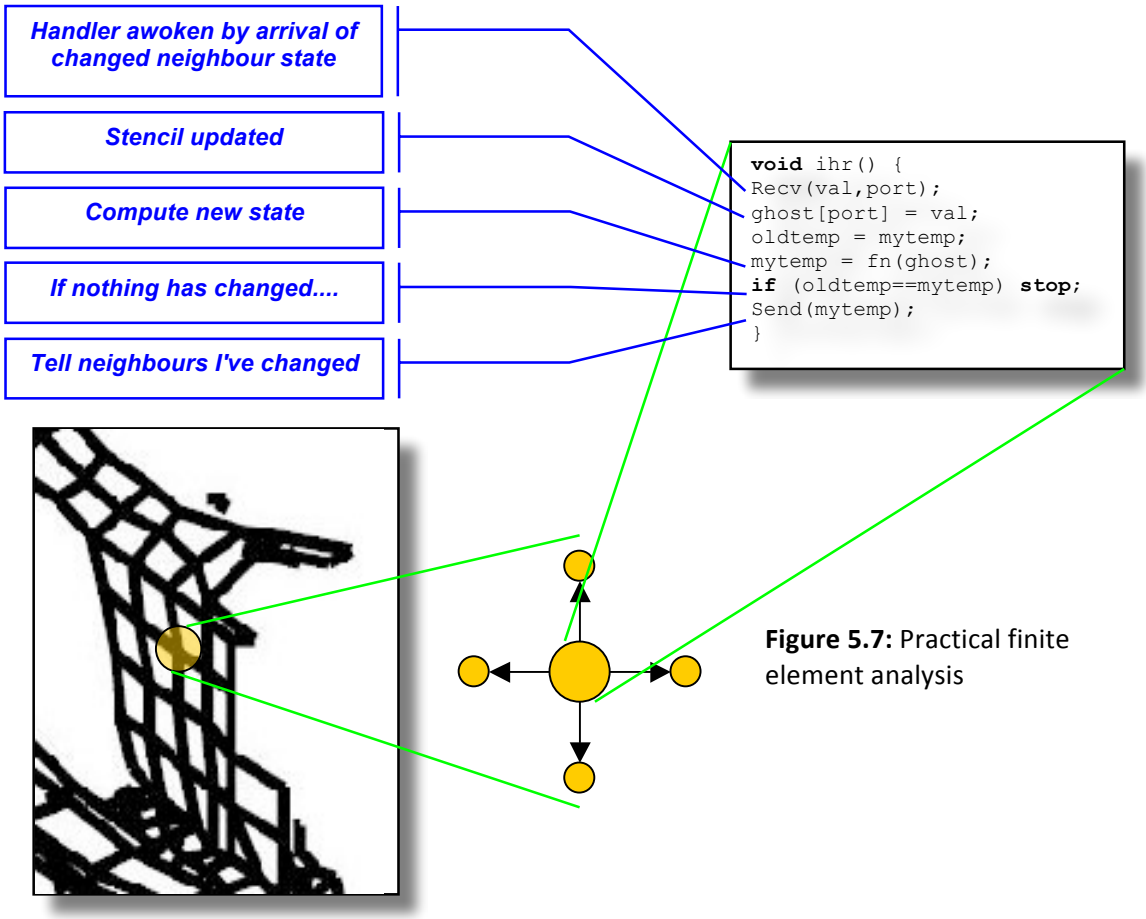


Figure 5.7: Practical finite element analysis

Pragmatically it may be sensible to allow it to change during the course of a calculation. Considering the heat equation used in the example:

$$\frac{\partial U}{\partial t} = D\frac{\partial^2 U}{\partial x^2} \approx \frac{\partial}{\partial x} D \frac{\partial U}{\partial x}$$

with a non-uniform spatial grid we must allow D = D(x), so the relationship becomes

$$\frac{\partial U}{\partial t} = D(x)\frac{\partial^2 U}{\partial x^2} + \frac{\partial U}{\partial x} \bullet \frac{\partial D(x)}{\partial x}$$

but we can store the $\frac{\partial D(x)}{\partial x}$ term as a synaptic weight and simply change the interrupt handler.

In précis:

- One handler (i.e. core) / mesh point
- Solution trajectories are non-deterministic
- Steady state solution is physically realistic
- *Detecting* convergence is hard

5.2.2 Discrete simulation

Discrete system simulation is probably one of the simplest system modeling algorithms in use today: an event queue is continually popped and consequently loaded, until all activity ceases. Counter-intuitively, parallel discrete system simulation is disproportionately complicated. The obvious distribution of the system under simulation onto an ensemble of parallel machines is deceptive, because of the need to maintain global causality across the machine ensemble.

However, in 1979 - long before the advent of desktop machines of any sort - Chandy and Misra published a seminal paper: *"... We propose a distributed solution where processes communicate only through messages with their neighbours; there are no shared variables and there is no central process for message routing or process scheduling. Deadlock is avoided in this system despite the absence of global control. Each process in the solution requires only a limited amount of memory....."*

They were effectively talking about SpiNNaker, three decades before its inception.

With SpiNNaker, we can map individual components (or small sub-graphs of components) onto each core – as in figure 5.8. It is fairly easy to see how logical events can be sent from device to device (core to core), possibly less so to see how temporal coherence may be maintained across the system. Chandy and Misra overcome this by the addition of another kind of signal, rather confusingly called a "null event". Each sub-graph knows (i.e. can statically compute) the fastest possible path through it. On receipt of a signal event (which will contain a simulation timestamp), a sub-graph will immediately broadcast to its successors a null event, effectively promising that nothing will happen until a set time in the future, allowing any downstream sub-graphs to compute up to the edge of the 'safe temporal horizon'. Whilst the practical implementation contains subtleties, the algorithm fits the SpiNNaker architecture so elegantly (logic events use MC packets, null events use p2p packets) it could have been designed with the SpiNNaker target in mind.

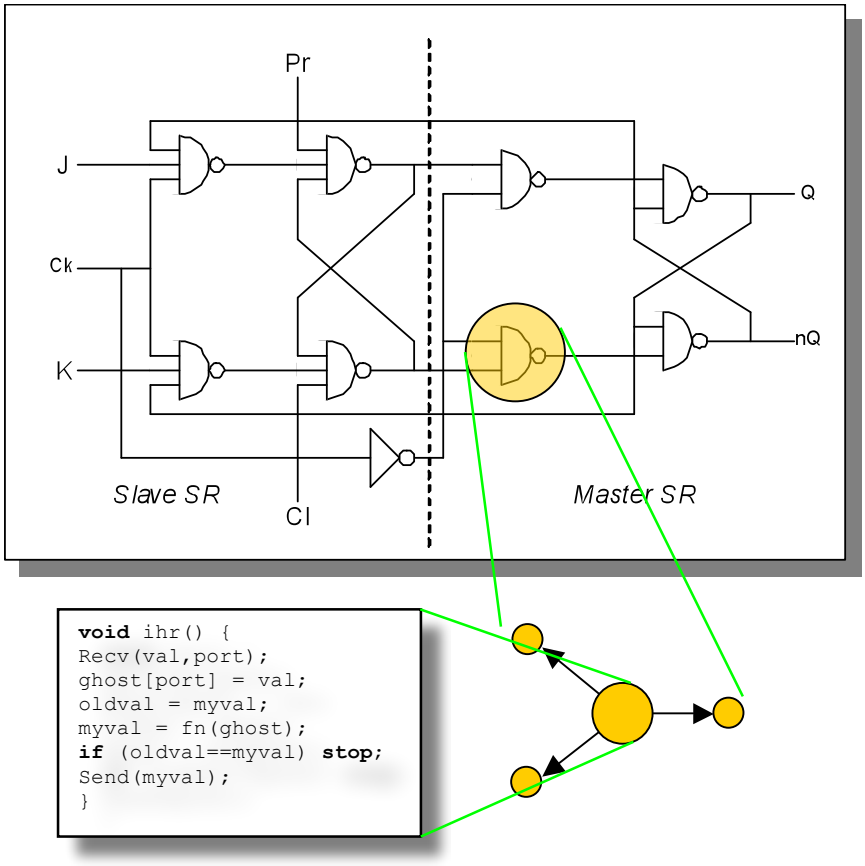


Figure 5.8: Discrete system simulation

5.2.3 Analogue simulation

(Electronic) analogue simulation is possibly the most heavily used and compute-intensive process used professionally, a position it has probably occupied since the advent of modern computers of any sort. Every nuance of the process has received close scrutiny over the years, attempting to make the algorithms more stable, more accurate, and above all, faster. One dominant technique has survived, the Newton-Raphson march-in-time algorithm, and this has resisted every attempt at parallelization since its initial development.

In précis, the algorithm takes as input a graph describing the circuit under simulation (CuS), where the vertices are instantaneous, multidirectional, isopotential electrical nodes, and the edges are the electronic components joining them, which in general will be non-linear and contain an interval state vector.

As the name suggests, the algorithm marches forwards in time – in non-uniform steps – and at each step, an adjacency matrix representing the connectivity of the problem graph is constructed, where the edges are represented by the linear approximation to the transfer characteristic of the actual component at that time and at that point in (V,I) space. This matrix must be inverted (which may or may not be possible – stiff problem systems tend to give rise to matrices that are almost singular). If inversion is not possible, the matrix is discarded, the time-step cut (typically by a factor of two) and the process restarted. If inversion is possible, the error associated with the linear approximation is computed, and if too great, the matrix is again discarded. If the solution is finally accepted, time is advanced, the last vertex solution extrapolated to the new time, and used as a starting point for the next matrix solution.

To condense the description even more, a simulation will typically consist of many thousand time-steps (and probably four times as many rejected ones), and at each time-step it is necessary to construct a matrix of rank equal to the number of vertices in the CuS, and invert it. Each term in each matrix will require the interrogation of the appropriate device model, which can easily be tens of thousands of lines of code in itself.

SpiNNaker can be used to attack the analogue circuit simulation problem in two ways. In the first, we map each component and each node onto its own SpiNNaker core, as in figure 5.9.

The fine-grained, asynchronous parallelism intrinsic to the SpiNNaker architecture presents a specific challenge – there can be no central control node. In this approach, we adopt a strategy entirely different from the march-in-time technique described above. Here, the conventional network matrix is not formed. Instead, the equations for the current at each node are formed independently. When the network equations have converged to the correct solution, the sum of currents at each node will be zero (Kirchhoff's Current Law). The hypothesis is that the circuit will eventually converge to a solution, as the contribution to each current is updated within each circuit element.

Conventionally, there are two ways to iteratively update a matrix equation: Gauss-Seidel, in which the matrix is updated as each new variable is calculated in a fixed sequence; and Jacobi, in which each circuit variable is re-calculated in order before the matrix is revised. The method adopted here is equivalent to updating the circuit variables and revising the matrix in a *random* order. In experiments to date, we have demonstrated (in simulation) that convergence is always reached, although the number of iterations is generally greater than needed for Jacobi updating. This is neither surprising nor alarming, because the simulations were carried out on a conventional (single thread) machine, and it has proved difficult to instrument the procedure to the point that useful timing estimates for SpiNNaker can be obtained.

The second approach involves treating time as simply another dimension in a solution space that already contains as many dimensions as there are nodes in the circuit.

Conventional algorithms rely upon predictor-corrector numerical integration algorithms, which can only progress monotonically with respect to time. In many other applications, predictor-corrector methods are regarded as inferior to newer methods, such as Burlisch-Stoer. In effect, using such methods would allow us to consider time as just another dimension to the solution space and allow the solution to proceed asynchronously in different parts of the network.

The main obstacle to using such methods is the formulation of circuit equations. Predictor-corrector methods allow circuit equations to be constructed by inspection (and hence are very suitable for the highly devolved structure of SpiNNaker). These other methods rely on state-space formulation and therefore require a transformation step. Initial work suggests that this transformation step can be automated and that accurate simulation results can be achieved. However, this thread of investigation is still very much in its infancy; the next stage will be to investigate whether asynchronous time advance is possible. It is currently unclear if this approach is compatible with the network relaxation solution method described above or with the fine-grained parallelism defined by SpiNNaker.

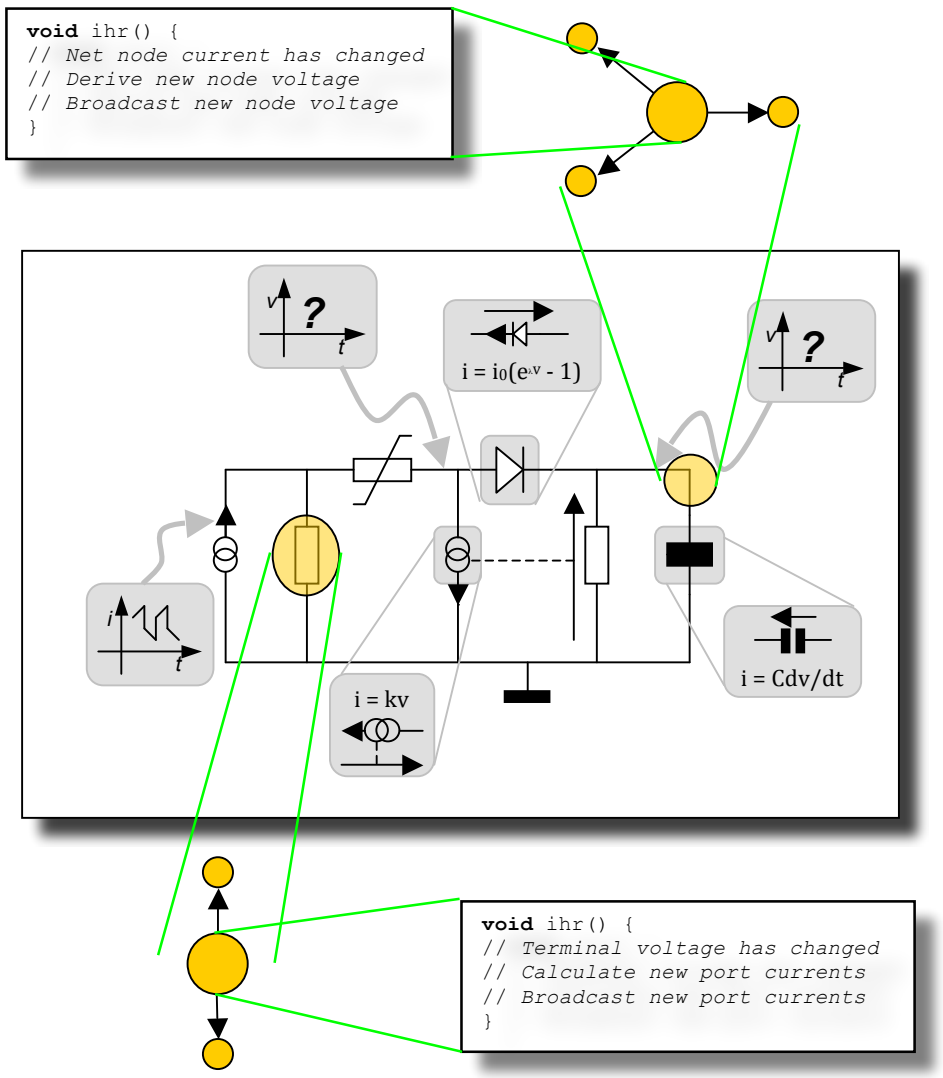


Figure 5.9: Continuous system simulation

5.2.4 Graph analysis for drug discover

Possibly the most ambitious and challenging non-neural application envisaged for SpiNNaker. Even the full-blown million-core machine will be nothing more than a demonstration of concept, but the potential payback of this application would be massive.

Almost every significant biological molecule is comprised of a chain (the backbone) of carbon atoms, of varying lengths, but 5.10^4 - 5.10^5 are not unreasonable numbers. Each carbon atom is capable of forming four bonds, two of which are used in linking adjacent atoms to form the backbone. The biological and chemical activity of the molecule is controlled by two things: the side-groups (ligands) attached to the other two bonds of the carbon atom, and the physical conformation of the molecule. Each backbone bond has effectively complete rotational freedom, and the number of ways the entire system can consequently fold is concomitantly massive.

We note, however:

- In nature, when such molecules are formed, they fold up into their stable conformation on a timescale of femtoseconds and with almost perfect repeatability. The number of energetic minima – even on the largest molecules – appear to be extremely small (usually one).
- It is the physical conformation, not the structural makeup, that dominates the biological behaviour. There is currently no known computationally efficient way of predicting the folded stereochemistry from the structural formula. Big Pharma routinely invest huge sums screening what are effectively random substances in order to discover their conformation. The most efficient computational technique currently in use is template matching against a library of known structures.

Each component (atom) of a molecule has associated with is an electrochemical field. In conventional physical electronics, charge carriers experience forces as a consequence of their charge and the ambient electric field, and any carrier concentration gradient. If the charge carrier moves as a consequence of the force, that movement may itself perturb the force surrounding it. From an electrochemical perspective, these two forces may be rolled into one and the forces on a particle are realized as the interaction of its *electrochemical* potential with an ambient *electrochemical* field.

SpiNNaker can model all this in terms of nearest-neighbour interactions:

- Space is represented as a mesh of points, each maintaining the local electrochemical potential. Each mesh point is assigned to an individual core.
- Each atom and each inter-atomic bond are also represented by individual cores.
- The system is allowed to relax: The curvature of the electrochemical field (represented by the mesh cores) tell the atoms how to move (subject to the constraints of the inter-atomic bonds), and the electrochemical potential of the atoms tells the field how to curve, as in figure 5.10.

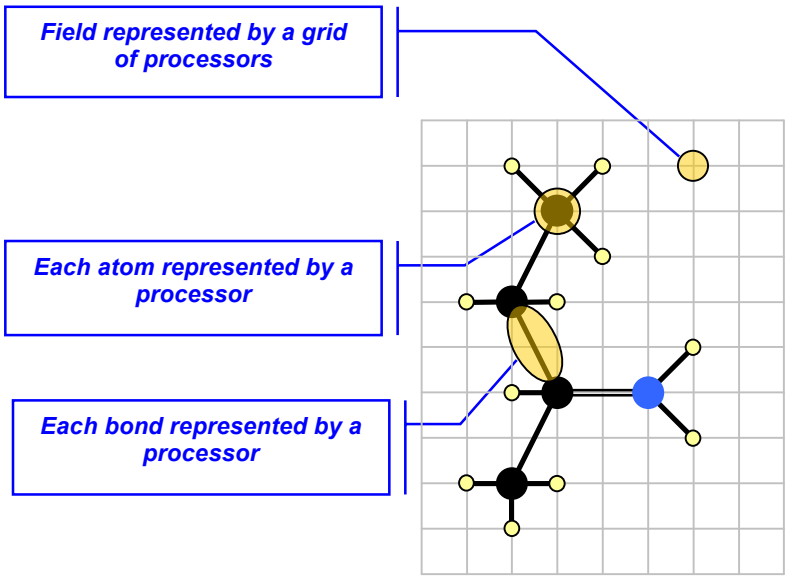


Figure 5.10: Particle and field simulation

Practical difficulties:

- In nature, biological molecules do not fold in isolation. Indeed, if you take a large molecule, isolate and purify it and allow it to fold in a vacuum, it generally assumes an unpredictable shape. Biological molecules fold in an environment that is usually polar (for example water) and usually containing what are known as 'chaperone' molecules, which are (usually simpler) molecules acting as conformational catalysts. None of this is conceptually difficult to model, it simply adds even more complexity to an already vast system.
- The sheer scale of the core array necessary to achieve anything like realism is way above the capabilities of the 106 machine. The necessity of chaperones hinders the use of dynamic meshing, so to model the space alone containing an interesting molecule is going to require of the order of $(10^5)^3$ threads.
- Alongside the electrochemical potential, transient van der Waals forces are known to play apart in the folding process.
- Plus, without question, a host of difficult issues we have not foreseen.

5.2.5 Ray tracing

In contrast to the previous application, ray-tracing is generally considered to be an 'embarrassingly parallel ' problem.

Animation and rendering of computer-generated images is a discipline that has advanced enormously over the last few decades, driven mainly by the entertainment industry and enabled by the exponential performance strides enabled by Moore's Law. The underlying algorithmic processes, however, have changed little. Some data structure is created, containing a representation of a three dimensional environment. (Which may itself be time-varying, but we ignore this as a minor complication). The goal of the rendering process is to create a two-dimensional image (the "vision plane") that represents what would be seen if the viewer looks at the three dimensional structure from a certain direction

with certain lighting, atmosphere and so on. This is achieved by mapping individual light rays between all the light sources and every point (pixel) on the vision plane. This is a massively detailed and compute intensive task; current state of the art renderings of, for example, fur, will model each individual hair in a pelt.

As a practical detail, to save unnecessary computation, the calculation is usually carried out backwards: A vector from the eye-point through each pixel in the vision plane is traced back through the three-dimensional virtual space to see where it ends up (it may of course split...). Any that terminate on a light source cause a contribution to the illumination of the pixel.

Although the calculations for each pixel are complex, there do not interact in any way; thus a 1000x1000 pixel image plane can be rendered by SpiNNaker in the same time it would take to render just one pixel with a conventional machine - figure 5.11.

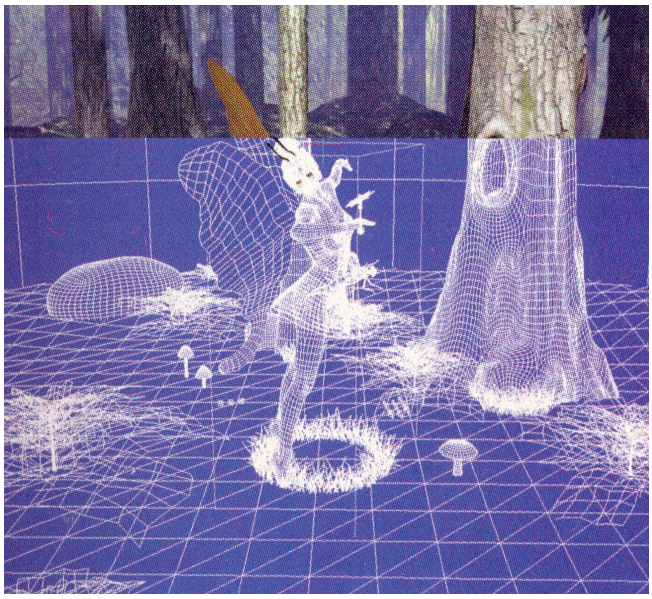


Figure 5.11: Ray tracing

5.2.6 Dataflow-driven applications

SpiNNaker as a computing engine sits somewhere between GPGPU architectures and “conventional” machines – figure 5.12. SpiNNaker is not a general-purpose machine. Whilst it is superbly good at some things, it is a wildly inappropriate architecture for others. At the other end of the scale, GPGPU architectures are much faster than SpiNNaker at what they do but are, in turn, much less versatile.

SpiNNaker occupies a position in another spectrum. In the world of IC design, the trade-offs between ASIC and FPGA implementation are widely accepted. ASICS are fast, small, efficient and have a massive NRE but very low quantity costs; FPGAs are large, slow, power-hungry and have no NRE. The cost crossover is currently generally accepted to be around 100,000 units.

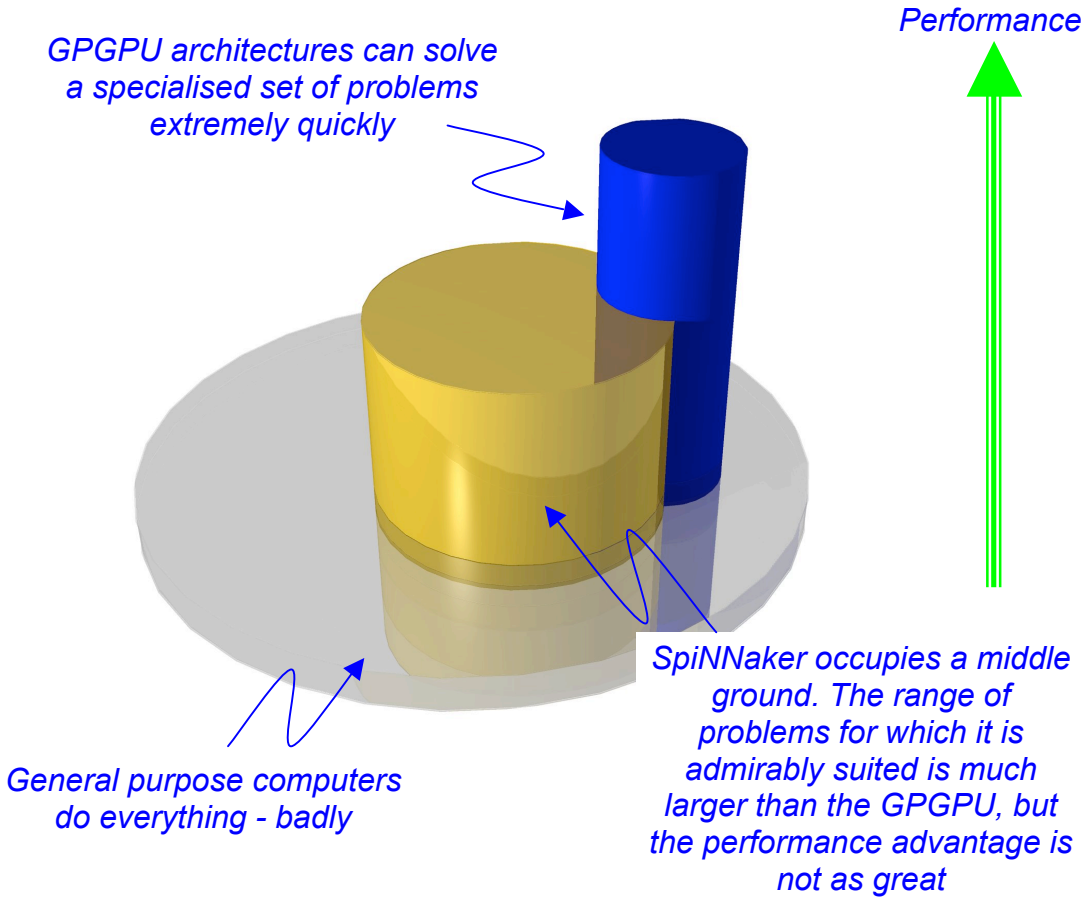


Figure 5.12: SpiNNaker in context

Aside from the usages outlined in the preceding sections, SpiNNaker can be viewed as an extremely versatile FPGA; the 106 is effectively a mesh of highly programmable functional units interconnected by an uncommitted (i.e. programmable) message fabric. In this context, it can be used – cheaply – for just about anything.

Whilst the conventional target functional unit in a synthesized system will be an arithmetic primitive – multiply, add, or maybe something a little more complex such as divide or multiply-accumulate, SpiNNaker can be used – programmed – to create functional units of much higher (arbitrary) complexity. Below we describe a simple matrix inversion system that operates in O(n) time. This is achieved by simply trading off processors for time – it uses $O(n^2)$ cores, but with a million available, this moves the boundaries of practicality quite significantly.

The triangular decomposition (or LU) method for inverting $\underline{\mathbf{A}}\mathbf{x} = \underline{\mathbf{b}}$ is well-known. Firstly $\underline{\mathbf{A}}$ is transformed into the product of two triangular matrices, $\underline{\mathbf{L}}$ (lower triangular) and $\underline{\mathbf{U}}$ (upper triangular).

Ax = **b** becomes **LU**x = **b**, which is equivalent to **L**(**U**x) = **b**. Inversion of this requires evaluation of **x** = **b****A**$^{-1}$, or (**b**.**L**$^{-1}$).**U**$^{-1}$ which is trivial, because of the specific shape of **L** and **U**.

As an example, consider the 4x4 matrix inversion problem below. For the sake of simplicity, we assume that all the matrices are well-behaved, i.e. all the u_{ij} and l_{ij} are not 0:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

Note that the specific shapes of **L** and **U** allow them to occupy – simultaneously – the same physical space as the original matrix **A**.

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21}u_{11} & l_{21}u_{12}+u_{22} & l_{21}u_{13}+u_{23} & l_{21}u_{14}+u_{24} \\ l_{31}u_{11} & l_{31}u_{12}+l_{32}u_{22} & l_{31}u_{13}+l_{32}u_{23}+u_{33} & l_{31}u_{14}+l_{32}u_{24}+u_{34} \\ l_{41}u_{11} & l_{41}u_{13}+l_{42}u_{22} & l_{41}u_{13}+l_{42}u_{23}+l_{43}u_{23} & l_{41}u_{14}+l_{42}u_{24}+l_{43}u_{34}+u_{44} \end{bmatrix}$$

We can create 4x4 processes (i,j) i,j = 1..4, and on each of these put a_{ij} and b_i such that process (i,j) contains a_{ij} and b_i. The dataflow to solve for l_{ij} and u_{ij} is shown in figure 5.13a. The arrows in figure 5.13a carry forward all the variables that have arrived from the same direction. For example, the message from (4,3) to (4,4) – labeled $l_{43}(6)$ in the figure – contains l_{41}, l_{42} and l_{43}. The process ends after u_{44} is evaluated after the seventh time-step. In general, the computational complexity is 2n-1, i.e. O(n). The computational cost complexity (the number of processes times the time complexity) is $n^2O(n)$, i.e. $O(n^3)$, the same as for the sequential algorithm. At the end of the LU stage (i,j) contains l_{ij} when i>j and u_{ij} when i<=j.

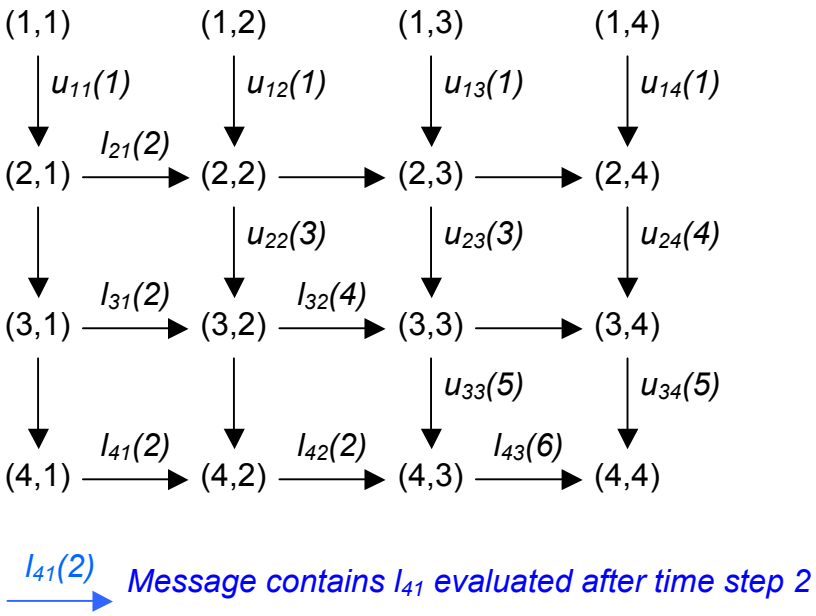


Figure 5.13a: SpiNNaker as a synthesis target

Now we need to perform the forward elimination to find the intermediate vector **y**:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \qquad \begin{aligned} y_1 &= b_1 \\ l_{21}y_1 + y_2 &= b_2 \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_3 \\ l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= b_4 \end{aligned}$$

So y_i is solved for on (i,j), as in figure 5.13b.

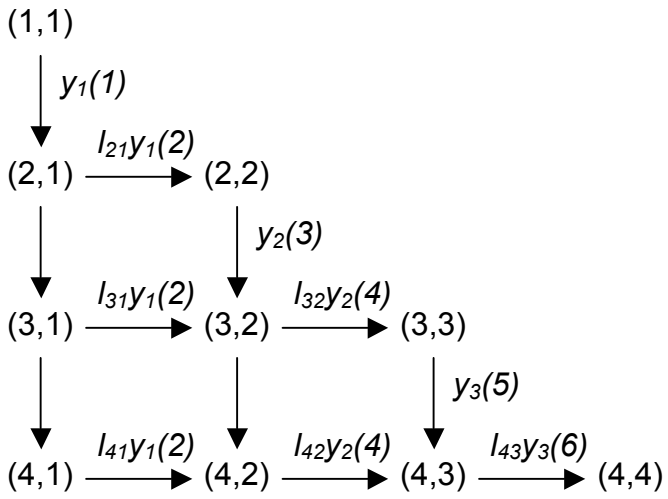


Figure 5.13b: SpiNNaker as a synthesis target

Again, this completes in 7(2n-1) steps, but when forward elimination immediately follows the LU decomposition the elimination stage completes 1 cycle after the decomposition stage and so has complexity O(1) in these circumstances.

Finally we perform the back substitution to solve for **x**:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \qquad \begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 &= y_1 \\ u_{22}x_2 + u_{23}x_3 + u_{24}x_4 &= y_2 \\ u_{33}x_3 + u_{34}x_4 &= y_3 \\ u_{44}x_4 &= y_4 \end{aligned}$$

so x_i is solved on (i,j), as in figure 5.13c.

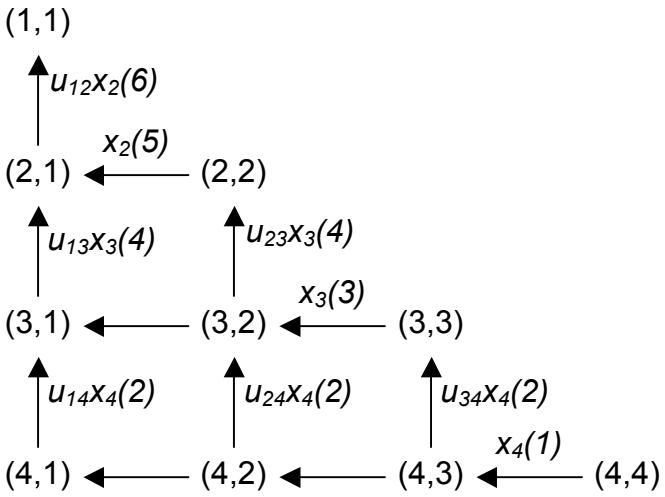


Figure 5.13c: SpiNNaker as a synthesis target

Once again, the solution requires seven steps. In general, the cost complexity is 2n-1, i.e. O(n).

5.2.7 Integer Linear Programming

Integer Linear Programming is an NP-hard problem to maximize a linear function under specified constraints. Advanced algorithms for solving ILP systems include "branch and bound" methods.

Future plans include investigating a parallel distributed "branch and bound" solution on SpiNNaker.

6 Evaluation methodology

6.1 Neural applications

As a major focus of the current work we are establishing collaborations with a number of leading research groups in computational neuroscience to evaluate the effectiveness of SpiNNaker as a modeling system for both real-time spiking neural networks and for Multi-Layer Perceptron neural systems.

6.2 Non-neural applications

Evaluating the performance of the non-neural application set is far simpler than it is for the neural counterpart. For each application, we simply need to:

- Implement it.
- Establish the behaviour is correct (and under what circumstances it might cease to be correct, considering potential traffic problems or other dynamic resource overloads).
- Measure the performance in terms of latency and throughput.

This is deceptive; there are a number of enabling components and processes that need to be created and put in place before any meaningful evaluation can occur.

- **Setup:** SpiNNaker can be an extremely powerful computing engine once configured. However, the configuration (physical loading and initialization) of the system is non-trivial, and the tool chain created to perform this is far from mature.
- **Problem pre-formatting:** SpiNNaker comes into its own when analyzing the behaviour of a system described in a very particular way – a large network of vertices and interconnect. This is not, in general, a manner of description sympathetic to the way humans manipulate information, and in general, each of the problem domains identified above – both neural and non-neural – require quite sophisticated and domain-specific pre-processing systems to translate problems from a high-level human-accessible description to a format accessible to SpiNNaker.
- **Instrumentation:** The communications infrastructure in SpiNNaker is highly tuned towards execution in the manner described previously. Command, control and debug requires a level of visibility and access that is currently only performed in an ad-hoc way. One of the main difficulties is that the system is essentially self-timed and non-deterministic – this means that any attempt to observe any quantitative timings perturbs the quantity we are trying to measure. This needs significant work before reliable metrics may be routinely obtained.
- **Simulation interfacing:** SpiNNaker is a system simulator of remarkable versatility. However, a simulator needs a simulation excitation if the results are to be non-trivial; some mechanism of providing this interaction (which can, of course, be two-way) needs to be engineered before meaningful performance evaluation can be obtained. In précis, the notion of a 'source device' needs to be defined.
- **SpiNNaker simulation:** A highly instrumented, event-accurate conventional SpiNNaker simulator needs to be designed and built.
- **Multi-packet protocols and meta-networks:** SpiNNaker was designed from the outset for the extremely rapid propagation of very small data packets. It is occasionally necessary to transfer larger tranches of information about the system; some form of higher-level transport mechanism needs to be put in place to facilitate this.

None of these issues are insurmountable, or even difficult; they simply require resource, and must be in place before any real impact can be made on the application space outlined in this report.

7 Conclusions and next steps

The SpiNNaker architecture is founded on fundamentally different principles from conventional computing engines. It is the first truly alternative architecture to the thread-dominated model introduced when computers as we now know them were first designed.

General-purpose computers do everything badly. They do everything, because they are general-purpose machines, and they do it badly, because by definition they are doing something they were not specifically designed to do.

SpiNNaker is *not* a general-purpose machine; there are a host of problems for which it is spectacularly ill-suited, and in these domains it can be easily outperformed by a $200 desktop machine. However, for problems that can be cast into a certain form (and the identified set is growing), SpiNNaker can approach almost perfect scalability (doubling the number of cores doubles the compute capabilities), although we still ultimately defer to Amdahl's Law.

Principal amongst these is spiking neural simulation. SpiNNaker was designed and optimized for this application.

The work to identify promising non-neural applications is at an earlier stage, and in each case considerable effort will be required to take the development far enough to demonstrate the effectiveness of the architecture for that case. The next step will be to identify one or two exemplars as proofs of concept and to undertake that development work.