Distributed Configuration of Massively-Parallel Simulation on SpiNNaker Neuromorphic Hardware

Thomas Sharp, Cameron Patterson and Steve Furber

***Abstract*— SpiNNaker is a massively-parallel neuromorphic computing architecture designed to model very large, biologically plausible spiking neural networks in real-time. A SpiNNaker machine consists of up to 2^{16} homogeneous eighteen-core multiprocessor chips, each with an on-board router which forms links with neighbouring chips for packet-switched interprocessor communications. The architecture is designed for dynamic reconfiguration and optimised for transmission of neural activity data, which presents a challenge for machine configuration, program loading and simulation monitoring given a lack of globally-shared memory resources, intrinsic addressing mode or sideband configuration channel. We propose distributed software mechanisms to address these problems and present experiments which demonstrate the necessity of this approach in contrast to centralised mechanisms.**

I. Introduction

Large-scale high-fidelity simulation of biological neural networks is a vision of research in computational neuroscience which has recently become feasible due to continuous fulfilment of Moore's law [3] and data resulting from progressive improvements to techniques in neuroanatomy [1], [16]. The inherent parallelism in neural information processing has allowed researchers to exploit multi-core computers with relatively little concern for the problems of, for example, shared data structures and synchronisation which arise in many computational problems. However, the huge scale of machines afforded by continually decreasing hardware costs presents a new set of infrastructure concerns quite different from those found with small-scale parallelism. In modelling systems on massively-parallel architectures we are concerned not only with the achievable ratio of real to simulated time but also the potentially significant overhead of simulation preparation, the limits of observable state during simulation and the presentation of data upon termination.

This paper presents software protocols for the initialisation, control and monitoring of a massively-parallel spiking neural network simulator, extending in part previous work on the subject [12]. In particular we address the problems of configuration of a flexible machine in the absence of fixed-topology command and control hardware typically found in large-scale computing architectures. We present experiments performed on novel neuromorphic hardware and infer from our results the necessity of a distributed approach to simulation configuration and observation in massively-parallel architectures.

The authors are with the School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK. Email: {thomas.sharp, cameron.patterson}@cs.man.ac.uk, steve.furber@manchester.ac.uk.

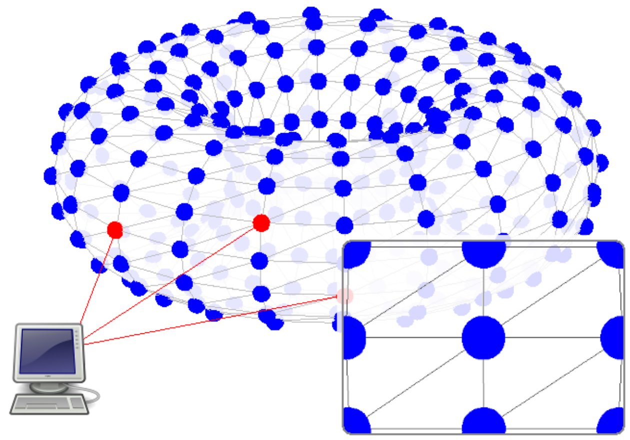

Fig. 1. A possible SpiNNaker machine topology. The host machine is represented by the graphic of a desktop computer. Processing nodes are denoted by blue dots, which are enmeshed in a triangular interprocessor communication fabric (inset). A subset of processing nodes, in red, form Ethernet links with a host machine. Communication between blue nodes and the host must traverse both the Ethernet and interprocessor communication channels.

II. Survey of Neuromorphic Hardware

The Blue Brain project at École Polytechnique Fédérale de Lausanne [13] is perhaps the most prominent project concerning large-scale neural simulation, having exploited some eight thousand processors of an IBM Blue Gene/L machine to model a cortical 'column' at ion-channel resolution. The scalability of the Blue Gene/L architecture to as many as 2^{16} processors [5] promises to support the project's intentions to model a number of laterally-connected columns in parallel and thereby simulate entire cortical hemispheres. Izhikevich and Edleman approach the same challenge using commodity hardware in a 60 processor Beowulf cluster and models of greatly lesser fidelity [9] supported by the argument that the computationally significant dynamics of neuron activity may be preserved in even relatively simple models [8], [7].

SpiNNaker is a massively-parallel neuromorphic architecture (figure 1) designed to model very large, biologically plausible spiking neural networks in real-time. The SpiNNaker project has typically followed Izhikevich and Edleman's approach in the development of neural simulation software [11] whilst sharing the design considerations with the hardware used by the former project of power consumption, scalability, reliability and efficient interprocessor communication [5]. The nodes of both Blue Gene/L and SpiNNaker are application specific integrated circuits, the former of which consists of two IBM Power PC cores running at 700MHz supported by dedicated floating-point units, whereas the latter incorporates eighteen ARM 9 processors (without hardware floating-point support) clocked at 200MHz. In both nodes, a processor may be selected to administrate chip functions such as communication, which in SpiNNaker we term the 'monitor' processor. Both architectures are designed to scale to 65,536 nodes arranged in a toroidal interconnect which supports both point-to-point and multicast communications and both architectures communicate with a host machine via a subset of nodes which have Ethernet interfaces. However, Blue Gene/L is distinguished by dedicated hardware which translates Ethernet frames into a native control protocol for direct administration of each node in the machine [6]. Conversely, nodes in a SpiNNaker machine which do not have an Ethernet interface may only communicate with the host via the interprocessor communication fabric to the nearest Ethernet-connected node.

This absence of a dedicated control infrastructure, motivated by considerations of design cost, reliability and flexibility, also ensure that at power-on a SpiNNaker machine is a mesh of truly homogeneous multiprocessor chips that are indistinguishable by any form of address. This presents challenges of how to break machine symmetry to create an address map, how to load heterogeneous programs and data to different processors and, once simulation is in progress, how to deliver monitoring information to the host without interfering with simulation computation or communication. The time required to complete these operations in a full-scale machine is also of concern, since the advantage of real-time model computation may be negated if the machine suffers from significant configuration overheads or if the observable state of a simulation is limited by 'downlink' bandwidth. A survey of the literature suggests these issues are common to neural simulation hardware: the control and I/O networks of Blue Gene/L address the problems; the addressing mode implicit in an Ethernet interconnect solves the first two problems for a Beowulf cluster and the third is mitigated by the operating system running on each node [15]; and an early proposal for massively-parallel neuromorphic hardware also included a dedicated control infrastructure to address these issues [10].

III. SpiNNaker Hardware

A SpiNNaker machine integrates as many as 2^{16} novel multiprocessor chips (see figure 2) each consisting of eighteen ARM 968 cores clocked at 200MHz. Each core has 32kB instruction and 64kB data tightly coupled memories (TCMs) and peripherals for controlling interrupts, interprocessor communication and DMA transfers. Each chip has a 32kB boot read-only memory (ROM), 32kB of shared RAM and is connected to a 128MB off-chip shared SDRAM used to store synaptic states. It is important to note that these memories comprise the only storage immediately available to processors on a SpiNNaker chip; the architecture does not have a hardware I/O subsystem with which to load

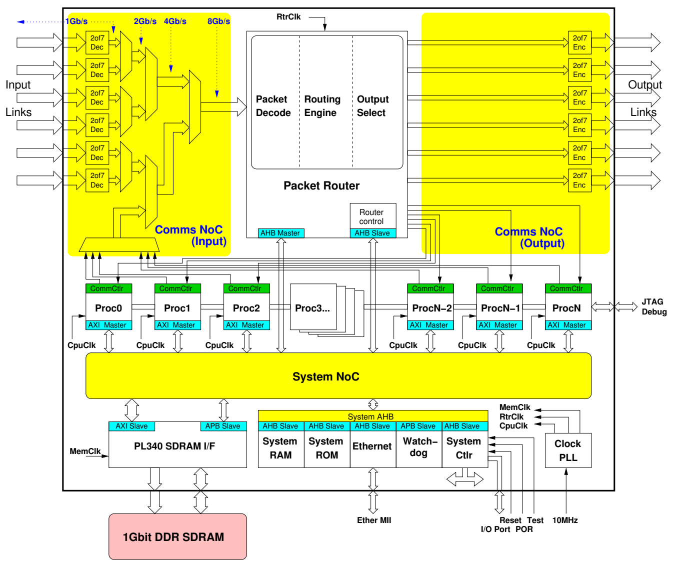

Fig. 2. The SpiNNaker chip. Processors read and write SDRAM and the memory-mapped registers of the system peripherals over the system Network on Chip (NoC). For interprocessor communication cores issue packets via their communications controllers to the router, which aggregates this input with that from the off-chip links in the communications NoC and routes packets to the other processors or the output links.

and store program code or simulation data from a globally-shared resource. A chip-shared system controller is used by the monitor processor to enable, disable and reset cores, configure clock signals and manage mutual exclusion operations. Each chip includes an Ethernet interface but only a proportion (depending on the size of the machine) are wired to a corresponding PHY controller and 8P8C connector.

Interprocessor communication is handled by an on-chip router, which aggregates packets from the six interchip links (see inset of figure 1) and the on-chip processors and forwards them to one or more of the same. Three routing subsystems exist for managing distinct patterns of traffic: the multicast routing system uses the ID of a spiking neuron as a lookup to a routing table to deliver neural event packets to one or more cores in the machine; point-to-point packets are routed from the monitor processor of a source chip to the monitor of a destination chip anywhere in the machine; and nearest-neighbour packets are communicated directly between the monitor processor of a source chip and a monitor processor on any one of the six immediately-neighbouring chips. The basic units of interprocessor communication in all routing subsystems are packets each consisting of a control byte, a four byte routing key and an optional four byte payload. Both the multicast and point-to-point subsystems require that routing tables are configured before use.

IV. Programming SpiNNaker

Given the problems listed above (created by the hardware constrains also discussed) programming SpiNNaker is a non-trivial task. The primary concern is distributing heterogeneous program code and simulation data to chips distant from an Ethernet interface with the host machine, which in turn necessitates distribution of system software to every node in order to establish an address map and administer communications. Furthermore, some method of monitoring and debugging simulation execution is required which does not interfere with the simulation itself.

We describe programming SpiNNaker in three phases:

- *node-boot*: power-on self-configuration
- *system-boot*: creation of a unified, addressable machine
- *simulation-boot*: distribution of simulation code and data

A. Node-Boot

At power-on each processor in a SpiNNaker chip immediately begins executing the program code stored in the boot ROM. Because this code is 'cast in silicon' at chip fabrication boot ROM correctness is critical to the function of a full scale SpiNNaker machine, given that we are only able to override the boot ROM using a mechanism which scales poorly to large numbers of chips. To guarantee program correctness we reduce the boot ROM function to hardware testing and reception (but not transmission) of more powerful software. The node-boot process may therefore be described by following the operation of a single chip in isolation.

Following power-on reset, each core copies the boot ROM into its instruction TCM and begins a series of tests on its peripherals such as the communications and DMA controllers. Failure of any of these tests (an event expected in smaller fraction than 0.01 of processors, depending on the quality of the fabrication process) causes a processor to record an error code in a designated location in system RAM and disable itself. The remaining processors then race to be the first to read from a system controller register which bestows monitor processor status on them. The 'losers' of this race become *fascicle* processors, so named for their impending role of simulating fascicles of neurons, and configure themselves to be interruptible only by the monitor processor before entering a power-saving wait-for-interrupt state. The monitor processor tests chip peripherals such as the router and shared memories and likewise self-disables on failure, thereby consigning all cores on-chip to indefinite wait-for-interrupt state. This approach follows from the assertion that a chip cannot take part in simulation if the chip peripherals are non-functional. Finally, the monitor processor tests for a PHY chip attached to the Ethernet interface (configuring it if found), enables interrupts from the Ethernet controller and router and enters wait-for-interrupt state.

Node-boot does not directly address any of the problems described in II but it does prepare the monitor processor of each chip to receive arbitrary programs of up to 32kB in size. A program is transmitted (by a mechanism discussed subsequently) to a chip as a series of nearest-neighbour packets over the inter-chip links, since this routing fabric is the only one of the three which does not require a preconfigured address map. Receipt of each packet raises an interrupt to the monitor which calls a handling function according to the contents the packet key. A `START` message precedes program loading (figure 3) and denotes in its payload the number of blocks in which the impending program is transmitted. Subsequently, a `BLOCK_START` message precedes one or more `BLOCK_DATA` messages, each of which carries a payload containing four bytes of program code. These payloads are stored by the monitor in system RAM and are marked off as 'received' in an array created for the purpose. A `BLOCK_END` message concludes a block and carries with it a payload containing a 32 bit cyclic redundancy checksum with which to validate the block. In response, the monitor uses the DMA controller to copy the block (simultaneously generating a checksum for comparison using the DMA controller hardware) from system RAM to a location in data TCM determined by the block ID. Finally, following a number of blocks depending on the image size, a `CONTROL` message prompts the monitor to run a routine in ROM which copies the program binary from data TCM to instruction TCM and begin its execution. For the sake of protocol simplicity, and by extension program correctness, acknowledgements are not transmitted by the boot ROM in response to received packets but instead the received program notifies the transmitter of its successful execution. In the absence of this notification transmission may be repeated.

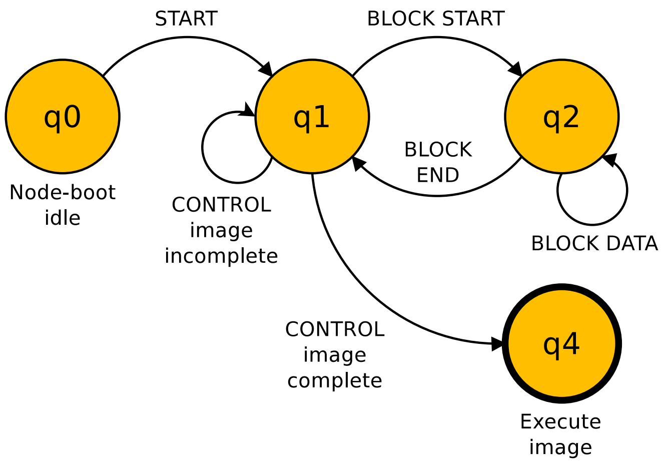

Fig. 3. Flow through the program loading procedure as prompted by the receipt of nearest neighbour packets.

B. System-Boot

It is intended that the first program transmitted to chips using this protocol is the system software which will administer the machine, so execution of the received code begins the phase we call system-boot.

1) Propagation: System software is transmitted by the host machine to Ethernet connected chips using a block-wise protocol similar to the one described previously with trivial adaptations to account for the difference in payload size between nearest-neighbour packets and Ethernet frames and automatic hardware validation of the latter. The first task of the system software upon its execution is to self-propagate to each of the neighbouring chips by transmitting the contents of the monitor processor's instruction TCM over the nearest-

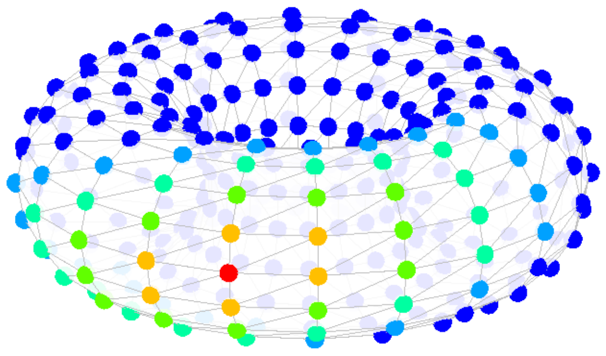

Fig. 4. A representation of a flood-fill process beginning with the single red seed chip. The wavefront at each distance from the seed chip is represented by evolving hues. Blue chips are yet to receive system software and remain in the wait-for-interrupt state following node-boot.

neighbour fabric, thereby fulfilling the role of the anonymous transmitter in the previous section. This process, which we call *flood-fill*, repeated iteratively and in parallel by each chip is the mechanism by which system software 'injected' into an Ethernet connected chip is distributed across a machine.

Figure 4 illustrates flood-fill commencing from an Ethernet-connect seed chip and proceeding across the nearest-neighbour network to a distance of four links from the seed chip. The wavefront of a flood-fill in the given topology touches upon $6n$ chips where n is the number of links between the seed chip and the wavefront. Consequently a flood-fill extending across n links from the seed chip affects

$$1 + \sum_{i=0}^{n} 6i = 6\frac{n(n+1)}{2} + 1 \tag{1}$$

chips. If we assume that transmission time across each link is invariant in the size of the wavefront (which we will subsequently demonstrate) then the time required for flood-fill is a root term of the number of chips in the machine. This suggests that flood-fill is an efficient mechanism for distribution of homogeneous binaries across large-scale SpiNNaker machines.

Since the boot ROM program does not contain routines for retransmission of received code, an entire binary must be received and executed by a chip before self-propagating to its neighbours. It may be argued that a pipelined approach to program distribution, in which each block received is forwarded to neighbours immediately following CRC validation, is more efficient. Although this is likely true, this approach requires a more complicated boot ROM program and it fixes the pattern of transmission. By keeping the flood-fill transmission (if not reception) mechanism in 'true' software, we are able to adapt the routines after chip fabrication to account for varying machine topologies, component failures and transmission rates which may vary with wire length. The same argument is applied to the other system software functions of address map creation and machine-wide communications.

2) Machine mapping: The symmetry of the SpiNNaker machines presented in figures 1 and 4 is broken only by the location of Ethernet connections to the host machine. These chips are assigned a socket into which a small read-only memory may be placed to provide the chips with unique MAC addresses. The host machine may then choose one chip as the origin of the 2D machine co-ordinate space (1 byte per dimension, hence the 2^{16} chip maximum) and send an Ethernet frame to that chip to inform it of the decision. The origin chip computes its neighbours' addresses relative to its own (for example the Northern neighbour is $(x,\, y+1)$ and the Southwestern neighbour is $(x-1,\, y-1)$ all modulo the x and y dimensions of the machine) and informs them of these co-ordinates via nearest-neighbour packet. Each chip repeats this process until every chip has received an address.

The point-to-point routing table contains a 3 bit entry for each of the 2^{16} chips which determines whether incoming packets should be forwarded on one of the six links or delivered to the local monitor processor. In a machine with correctly configured point-to-point routes a packet is transmitted from source to destination by comparison of the address in its key with the routing tables of each intermediate chip. Two approaches have been devised to configure the point-to-point router.

In the first approach the routing table is configured by the system software immediately upon receipt of a chip address, for example (x_0, y_0) configures the router to forward packets on the Northeast link for chip IDs

$$C_{ne} = \{(x,\, y) \mid x > x_0,\, y > y_0\} \tag{2}$$

Chips also ping their neighbours using nearest-neighbour packets to establish link status before configuring routing tables. Continuing the previous example, if the Northwest link was found to be dead then routing table entries in (x_0, y_0) to all chips C_{ne} would instead point to the North or East links. However, the absence of a global view of connectivity in this approach results in routing cycles under certain patterns of link failure. Furthermore, point-to-point packets transmitted from the origin to the chip to its immediate Southwest neighbour traverse the entire machine through North, East and Northeast links because of the modulo arithmetic used to compute chip addresses.

In the second approach each chip maintains a hop-count table with entries for every other chip initialised to infinity. At regular intervals every chip $(x_0,\, y_0)$ transmits a nearest-neighbour packet to its neighbours

$$C_{d=1} = \{(x,\, y) \mid D((x_0,\, y_0),\, (x,\, y)) = 1\} \tag{3}$$

containing the chip ID and a hop-count of one. Chips $C_{d=1}$ set the point-to-point route to $(x_0,\, y_0)$ to be the link upon which the nearest-neighbour packet was received, conditional on the hop-count being less than the entry in the hop-table for that chip. Upon the same condition, chips $C_{d=1}$ increment the hop-count and retransmit the packet to chips in $C_{d=2}$ which look up the hop-table entry to $(x_0,\, y_0)$ and undertake configuration and retransmission accordingly. This approach

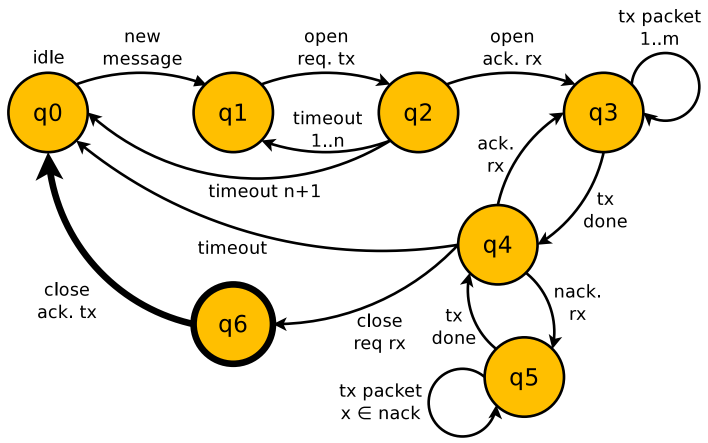

Fig. 5. SDP transmission control. State transitions are prompted by transmitter actions or receipt of acknowledgements and negative acknowledgements from the receiver. Message transmission occurs in sequences of m packets for which the receiver sends an 'ack' on successful receipt or a 'nack' denoting missing packets which are then retransmitted. The receiver sends a close request upon receipt of the last sequence. Failure of the receiver to respond at any stage results in timeouts which may terminate transmission. Entry into $q6$ denotes successful transmission.

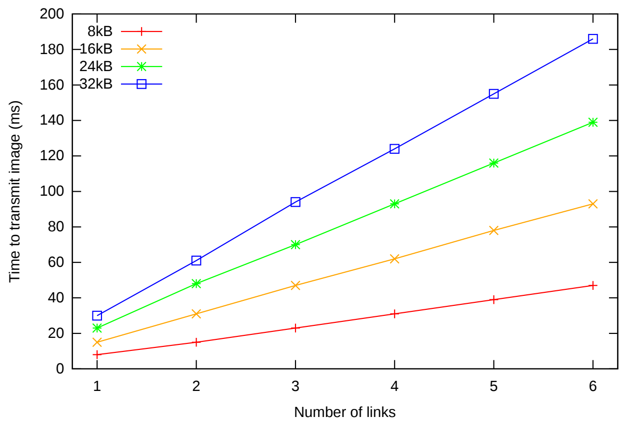

Fig. 6. Flood-fill times along a chain of chips. Repeated trials found no deviation in results hence the omission of error bars. Note that transmission time varies linearly with both transmission size and the number of links, supporting the earlier assumption in IV-B.1 that transmission time is invariant in the distance between seed chip and wavefront.

requires that information pertaining to each chip is flooded across the entire machine but ensures that routing cycles are obviated, the shortest point-to-point route between chips is established and irregular topologies are possible.

3) Point-to-point communication: A general command and control framework (notably absent in SpiNNaker hardware) is implemented by the system software in the form of the SpiNNaker datagram protocol (SDP). SpiNNaker datagrams are addressed to the monitor processors of specific chips or the host machine and contain a command, a number of arguments and a variable size payload in order to insert, manipulate and extract programs and data and to control simulation. The one kilobyte datagrams are comfortably contained within frames transmitted between the host machine and Ethernet connected chips but must be broken up for transmission across the point-to-point routing fabric.

SpiNNaker is not immune to the congestion and corresponding packet loss problems which typically affect packet-switched communication fabrics. Consequently, point-to-point packets are unreliable fire-and-forget transmissions which must be acknowledged in order to confirm successful receipt. This in turn calls for transmission of metadata within each packet and maintenance of state by the transmitter and receiver. A simplified representation of SDP transmission control over the point-to-point fabric is shown in figure 5; reception control is a mirror of this process. SDP communication begins with negotiation for a connection, proceeds with transmission of sequences of packets which are acknowledged by the receiver and is terminated by agreement upon complete reception. Acknowledgements of each sequence of packets guarantees completeness of transmission and an optional checksum transmitted at termination may guarantee correctness of the transmission contents.

C. Simulation-boot

Spiking neural networks to be modelled on SpiNNaker are specified by a researcher at the host machine and are compiled into data structures representing the neurons and synapses to be modelled by each processor and tables that represent their logical connectivity to the multicast routers [4]. This heterogeneous data ($\leq$128MB) is loaded into the SDRAM of each chip along with homogeneous simulation software ($\leq$32kB) using SDP messages sent by the host machine and propagated according to the protocols described above. The monitor processor commences simulation by prompting fascicle processors to copy the simulation software and data into local memory and begin execution, during which SDP messages are used for simulation monitoring such as reading the spike times or membrane potentials of subsets of neurons.

V. Experimental Results

A batch of SpiNNaker test chips containing two processors and all of the hardware discussed in III was produced prior to fabrication of the full SpiNNaker chip in order to verify the design. This test hardware was used to develop and debug the software and protocols described here and finally to address the concern of machine configuration time.

We arranged a number of test PCBs to create a chain of links spanning seven chips (the maximum number of serial connections were were able to fashion) with an Ethernet connection between the terminal chip and a conventional desktop computer acting as a host machine. The system software was seeded by the host machine in the terminal chip and self-propagated to its neighbour at run-time, as described in IV-B.1. The neighbouring chip then executed the system software, thereby propagating the code to its neighbour, and so on. Using a minimal version of the system software containing only the self-propagation routines and padding, we recorded the time required for transmissions of various sizes across each link. The results, presented in figure

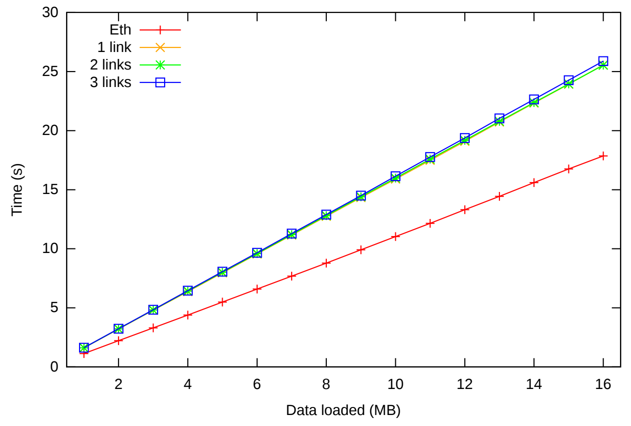

Fig. 7. Memory loading times using point-to-point SpiNNaker datagrams. Deviation in results over repeated trials was too small to plot. The disparity in transmission times to chips at varying distances from the Ethernet connected chip is almost unobservable in the given range.

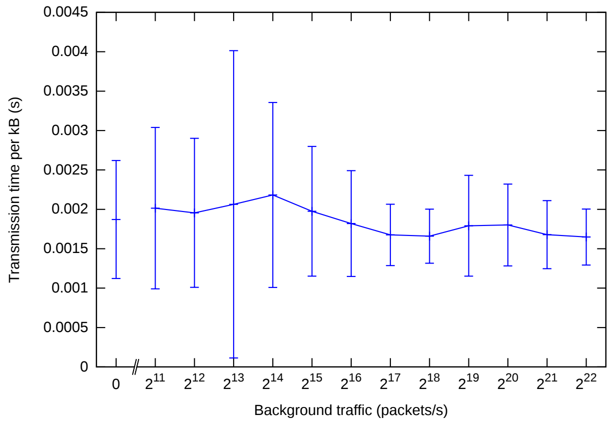

Fig. 8. Mean memory loading times using point-to-point SpiNNaker datagrams transmitted across three interchip links with varying background traffic. The standard deviation across one thousand transmissions exceeds the variance in transmission times with amount of background traffic.

6, support our assumption that transmission time for each link is invariant in the size of the wavefront.

Navaridas et al. [14] give the shortest path between any two chips in an n-by-n SpiNNaker machine of the given topology as $\lfloor \frac{2n}{3} \rfloor$. Taking $n = 256$ (the largest possible SpiNNaker machine) and the recorded transmission time over a single link of thirty milliseconds it is apparent that system software may be distributed to every chip in a full-scale SpiNNaker machine from a single Ethernet connection in around five seconds. Building multiple evenly-spaced Ethernet connected chips into a machine has the effect of dividing this result by the number of said chips, given that flood-fill may proceed in parallel from multiple points.

We rearranged the test PCBs for regular connectivity (the Northeast link, for example, of any chip would connect only to the Southwest of another if at all) in order to profile the point-to-point communications protocols described in IV-B.3. This limited the maximum distance between test chips to three hops. In successive trials we transmitted arbitrary binary files of 16MB in size as a series of 2^{14} 'load SDRAM' SDP messages to chips at incremental distances from the host. We were able to load one megabyte approximately every 1.1 seconds into the Ethernet connected chip (figure 7) and the same quantity approximately every 1.6 seconds into distal chips, largely invariant in the distance between the distal and Ethernet connected chip.

To investigate the transmission rates which might be expected during simulation, we employed unused processors to generate background traffic on the network. We then loaded 1kB SDP payloads one thousand times into chips at varying distances from the Ethernet connected chip and recorded the time taken for transmission of each datagram. The results of transmission to the most distant chip, three links from the Ethernet connected chip, are shown in figure 8. We provide a control measurement for which background traffic is absent to account for slight differences in methodology between this experiment and the last. We find that the maximum rates of background traffic which we are able to generate (2^{22} packets per second, corresponding to approximately 0.9 of the traffic on the channel during SDP transmission) has no observable effect on transmission times averaged over trials.

VI. Discussion

We have presented software protocols for the initialisation, control and monitoring of a massively-parallel spiking neural network simulator. In particular we have addressed the problems of configuration of a homogeneous machine in the absence of command and control hardware typically found in large-scale computing architectures.

We have approached the problem of rapid machine configuration using a flood-fill mechanism of program distribution. Initial experiments suggest that homogeneous system software may be distributed by flood-fill to the 2^{16} chips of a full-scale SpiNNaker machine particularly rapidly. Application-boot, however, necessitates linear distribution of heterogeneous data and presents a significant problem: to load the 8TB of SDRAM distributed across a machine using SpiNNaker datagrams would take almost a month, assuming the worst recorded transmission rate of a half kilobyte per millisecond, which is unacceptable for a real-time simulator.

The experiment summarised in figure 7 shows that loading time increased significantly between the Ethernet and distal target chips but was then largely invariant in the hop count to the latter, suggesting that transmission rates were primarily affected by the software overhead of the point-to-point SpiNNaker datagram protocol. The experiment summarised in figure 8 supports this argument by showing that background traffic had no observable effect (at least for the achievable degree of network utilisation) on transmission rates. We expect to improve transmission rates by optimising the SpiNNaker datagram protocol but the problem will remain that the distribution time of heterogeneous data scales linearly with the number of chips per Ethernet connection.

We propose to address this problem by describing spiking neural networks as populations and patterns of connectivity, rather than individual neurons and synapses, using a tool such as PyNN [2]. We hope to achieve sufficient compression in this approach that a single binary representing the network for the entire machine might be distributed to every chip via flood-fill. Should flood-fill proceed at the same rate as recorded in V we may expect a network description the size of SDRAM (128MB) to be distributed to 65,536 chips from a single Ethernet link in around six hours. On the reasonable assumption that a machine of this size may have many tens of Ethernet connected chips, this figure is reduced to the order of minutes. Furthermore, flood-fill across chips in node-boot transmits the entire binary across each link before proceeding to the next, improving reliability in the critical boot ROM software but leaving all chips not on the wavefront idle: subsequent implementations may use 'pipelined' transmission, wherein each chip propagates packets onwards immediately upon receipt in order to expedite flood-fill.

Configuring SpiNNaker using a high-level network representation necessitates a distributed 'compiler' running on each chip to unpack the description into local neuron and synapse data structures and to compute the routing tables. We expect that this self-configuration is essential in all massively-parallel neuromorphic architectures if a second super-computer is not to be required to compute the routing tables for the billion-neuron simulations of the first. Although we currently use PyNN for specification of networks for compilation on the host [4] we have yet to approach the problem of on-machine compilation.

Finally, in figure 8 we present the recorded transmission rates of SpiNNaker datagrams across routers loaded with an additional $\approx 4\cdot10^6$ packets per second. The SpiNNaker inter-chip links are capable of delivering thirty million packets to the router every second but it is intended that the average load during simulation should be around one tenth of this figure in order to prevent congestion during peaks in traffic. Assuming that we will model seventeen thousand neurons per chip spiking at around 10Hz [11] the average load on the router of any given chip will be $\approx 3.2\cdot10^6$ packets per second, suggesting that the experiment is a reasonable demonstration of the rate at which data may be dumped to the host machine during simulation. So, the simulation monitoring problem in which we are concerned about the amount of data we can extract from the machine in a given time is equivalent to the application-loading problem in which we address the amount of data we may insert into the machine. Referring again to the recorded transmission rate of a half kilobyte per millisecond, we conclude that (per Ethernet connection) we are able to observe in real-time the complete 32 bit state of 128 neurons or the spiking of 4,096 neurons assuming one bit per neuron indicating firing. This is clearly an extremely limited insight into a machine which may model millions of neurons per Ethernet connection but the problem is neither unique to SpiNNaker (BlueGene/L and Beowulf machines would only acheive an order of magnitude improvement on this figure under maximum utilisation of their 100Mbit/s Ethernet links to their hosts) or the primary concern: dumping the huge volumes of data produced by massively-parallel architectures to a host only moves the problem of post-processing to a less capable machine. Consequently, we argue that just as simulation-boot must be performed in a distributed manner, intermediate- and post-processing of data must make use of the same parallel resources required for modelling if the output of such models is to be useful.

VII. Acknowledgements

This work is funded by EPSRC and ARM Holdings. We would like to thank Mikel Lujan and Luis Plana for helpful suggestions.

References

[1] Tom Binzegger, Rodney J. Douglas, and Kevin A. C. Martin. A Quantitative Map of the Circuit of Cat Primary Visual Cortex. *The Journal of Neuroscience*, 24:8441–8453, 2004.

[2] Andrew P. Davidson, Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:1–10, 2009.

[3] Hugo de Garis, Chen Shuo, Ben Goertzel, and Lian Ruiting. A world survey of artificial brain projects, Part I: Large-scale brain simulations. *Neurocomputing*, 74:3–29, 2010.

[4] Francesco Galluppi, Alexander Rast, Sergio Davies, and Steve Furber. A General-Purpose Model Translation System for a Universal Neural Chip. In *Neural Information Processing, International Conference on*, pages 58–65, 2010.

[5] A. Gara, M. A. Blumrich, D. Chen, G. L.T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49:195–212, 2005.

[6] R. A. Haring, R. Bellofatto, A. A. Bright, P. G. Crumley, M. B. Dombrowa, S. M. Douskey, M. R. Ellavsky, B. Gopalsamy, D. Hoenicke, T. A. Liebsch, J. A. Marcella, and M. Ohmacht. Blue Gene/L compute chip: control, test, and bring-up infrastructure. *IBM Journal of Research and Development*, 49:289–301, 2005.

[7] Eugene M. Izhikevich. Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, 14:1569–1572, 2003.

[8] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. The MIT Press, 1st edition, 2006.

[9] Eugene M. Izhikevich and Gerald M. Edelman. Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences*, 105:3593–3598, 2008.

[10] Mark James and Doan Hoang. Design of Low-Cost, Real-Time Simulation Systems for Large Neural Networks. *Journal of Parallel Distributed Computing*, 14:221–235, 1992.

[11] Xin Jin, Steve B. Furber, and John V. Woods. Efficient Modelling of Spiking Neural Networks on a Scalable Chip Multiprocessor. In *Neural Networks, International Joint Conference on*, pages 2812–2819, 2008.

[12] M.M. Khan, J. Navaridas, A.D. Rast, X. Jin, L.A. Plana, M. Luján, J.V. Woods, J. Miguel-Alonso, and S.B. Furber. Event-Driven Configuration of a Neural Network CMP System over a Homogeneous Interconnect Fabric. In *Parallel and Distributed Computing, Eighth International Symposium on*, pages 54–61, 2009.

[13] Henry Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, pages 153–160, 2006.

[14] Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis A. Plana, and Steve Furber. Understanding the Interconnection Network of SpiNNaker. In *Supercomputing, ICS conference on*, pages 286–295, 2009.

[15] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. BEOWULF: A Parallel Workstation For Scientific Computation. In *Parallel Processing, International Conference on*, pages 11–14, 1995.

[16] Alex M. Thompson and Christophe Lamy. Functional maps of neocortical local circuitry. *Frontiers in Neuroscience*, 1:19–42, 2007.