Self-discovery Algorithms for a Massively-Parallel Computer

Kier J. Dugan, Jeff S. Reeve, Andrew D. Brown
Electronics and Computer Science
University of Southampton
Southampton, UK
{kjd1v07, jsr, adb}@ecs.soton.ac.uk

***Abstract*—SpiNNaker is a biologically-inspired massively-parallel computer design that will contain over a million processors, distributed over more than 60,000 chips. The system bootstrap must discover how they are connected for the machine to enter a usable state. In this paper we describe a set of algorithms for discovering missing or malfunctioning inter-chip links, assigning unique identifiers to each chip, and building point-to-point network routing tables. All of the algorithms have been simulated, and will be implemented into SpiNNaker after further investigation. Our goal is to design an autonomic bootstrap stage that can operate on arbitrary machine geometries.**

***Keywords*—SpiNNaker; self-discovering networks; parallel computer bootstrap procedures; self-configuration.**

I. Introduction

SpiNNaker [1] is a biologically-inspired massively-parallel computer that will contain over a million processors, distributed across more than sixty-thousand Multi-Processor System-on-Chip (MPSoC) devices. The flagship application for this machine is to model large neural-networks containing biologically-realistic numbers of neurons and synapses in biological real-time [2]. Each MPSoC contains 18 ARM processors with the intention of using 16 for simulation, one as a *monitor* processor that manages communications for the chip, and one as a spare for reliability purposes.

The network fabric of SpiNNaker follows a globally asynchronous, locally synchronous (GALS) methodology which allows each processor to exist in its own clock domain [3]. Inside each MPSoC, an asynchronous network-on-chip (NoC) connects all of the processors to the router. These routers communicate with each other using six inter-chip ports which have also been inspired by NoC designs. Both networks use m-of-n codes to provide reliable, low-latency, self-timed communications using compact transmit/receive logic.

Four routing methods, each optimised for a specific task, operate in parallel throughout the machine [4]. MC (multicast) traffic is used to carry address-event representation (AER) simulation data in a one-to-many fashion inspired by neural connectivity patterns; FR (fixed-route) packets are a specialisation of this, where the source-addressed routing has been sacrificed in favour of a larger payload. P2P (point-to-point) packets carry command and system information between two chips of the machine in a one-to-one mapping. Finally, NN (nearest-neighbour) packets provide a one-to-one link between a chip and any one of its six immediate neighbours.

An ideal SpiNNaker network is an isotropic 3D torus with extra diagonal links to facilitate triangular routing around problematic links. Due to the scale of the final machine, there can be no guarantee that all processors and chips will be functional on start-up. Assigning labels and routes statically is therefore not viable, nor can any assumptions be made about the regularity of the structure of the machine.

Other MPSoCs avoid this issue by taking more self-contained approach, acting as either master- or co-processors. In the Centip3De [5] MPSoC, a 3D NoC is used to maintain cache-coherency throughout the chip so that all 64 ARM Cortex-M3 processors can communicate using shared memory. A similar approach has been used by Intel in their prototype data-center-on-a-die, which reserves a small region of shared memory as a *message-passing buffer* that allows the 48 Pentium-class IA-32 processors to communicate [6]. TILE64™ [7] uses several software-controlled networks (one also being software-routed) to connect a regular grid of 64 VLIW processors to the system RAM, on-chip peripherals, and each other. PCI-express and Ethernet controllers provide the inter-chip communications instead of allowing the processor network to bridge chip boundaries as it does in SpiNNaker.

These devices either assign processor labels statically or are structured such that they can be derived at runtime. Similarly, conventional cluster machines assembled from commodity computer hardware may make use of the vendor-assigned MAC address of the network interface card as a machine label. Higher level protocols, such as the Dynamic Host Configuration Protocol (DHCP), can be used to automatically assign system-wide labels from a central source.

SpiNNaker chips are, nominally, identical and hence there is no equivalent of a MAC address available. It follows that only NN packets may be used during the system boot because the higher-level networks require both chip labels and at least partial knowledge of the machine geometry. In this paper we present a set of algorithms that will discover missing/malfunctioning inter-chip links, assign each chip a unique label, and then build the P2P tables. All of these algorithms have been prototyped in simulations that mimic the distributed interrupt-driven nature of SpiNNaker.

The rest of this paper is structured as follows: in Section II

 ISBN: 978-1-61208-274-5

we introduce the bootstrap algorithms that eventually lead to a constructed P2P table on each chip; Section III briefly describes some early-stage work towards building the MC tables; and Section IV concludes with a summary of this paper and introduces planned future work.

II. Bootstrapping Algorithms

The SpiNNaker machine does not power up with any implied knowledge of its structure and must discover this as part of the bootstrap procedure. Each chip has an integrated ROM that stores a small boot program capable of initialising the NN routing mechanism and the Ethernet controller if there is an active connection. All cores enter the wait-for-interrupt (WFI) state once these resources are ready, and may only process interrupts from these sub-systems.

System software, which must be loaded into the machine from an external host, is then distributed to all chips using a flood-fill mechanism [8]. Existing algorithms for providing each SpiNNaker chip with a unique label and building the point-to-point tables are semi-automatic and require prior knowledge. The algorithms presented in this paper aim to remove this constraint and hence provide a more autonomic self-discovering bootstrap process that may be applied to *arbitrary* machine geometries.

A. The α-ping

Malfunctioning links are not detected during the initial flood-fill process primarily due to the small size of the boot ROM. After control is passed from the boot-loader to the system software, higher-level detection algorithms may be applied to the machine to detect faults. Completion of the flood-fill cannot easily be detected without making assumptions about the machine geometry. We propose the α-ping as a process that will be incorporated into the system software and then executed after an appropriate time-out to allow for completion of the flood-fill.

Two tokens are passed between the monitor processors of adjacent chips—the request token, $\alpha_{\mathbf{R}}$, and the acknowledgement token, $\alpha_{\mathbf{A}}$. Each chip labels all local ports as `undefined` immediately after executing the system software. The host machine starts the process by injecting $\alpha_{\mathbf{R}}$ into the Ethernet-connected chip. $\alpha_{\mathbf{R}}$ is then broadcast to all neighbouring chips and every local port is assigned the `requested` label. Chips respond to incoming $\alpha_{\mathbf{R}}$ with $\alpha_{\mathbf{A}}$ and label the appropriate port as `active`; further $\alpha_{\mathbf{R}}$ are broadcast to all other unlabelled ports and the `requested` label will be attached as before. After a predetermined chip-local time-out (for the same reasons as with the flood-fill) all ports that are still labelled `requested` are assumed to have malfunctioned and will hence be `inactive`.

B. Assigning Chip Labels

Each chip of the SpiNNaker machine must be assigned a unique identifier so that P2P routes can be established. The existing method assumes a grid topology and requires extents in X and Y to be specified *a priori* by the operator [9]. A chip will be assigned a label from a predecessor (which is an Ethernet-connected host in the case of the root chip) and then geometric assumptions and simple arithmetic ($x \bmod X$ and $y \bmod Y$) are used to calculate the labels for the surrounding chips. Two clear pros of this method are that a) chip labels are entirely deterministic so there cannot be any conflicts during the set-up; and b) it will operate as a wave front of parallel computation emanating from the root chip. However, the geometric assumptions constrain the SpiNNaker machine to a grid which may not always be an appropriate geometry.

A solution is to build a spanning tree with its root at the Ethernet-connected chip. The generated tree structure may be derived from an *arbitrary* connected graph and provides a simple method for building hierarchical barrier, scatter and gather constructs common in parallel computing. Chips can also be uniquely labelled as part of the traversal process that builds the tree.

The SpiNNaker programming model is based on *events* (i.e., hardware interrupts) that are triggered either by a regular timer tick or by a packet arrival. Deriving the spanning tree must be performed within SpiNNaker and can only make use of NN packets to raise events on neighbouring chips. Only the monitor processors can take part in this process, and the algorithm must be defined on a per-node basis rather than on the machine-graph as a whole.

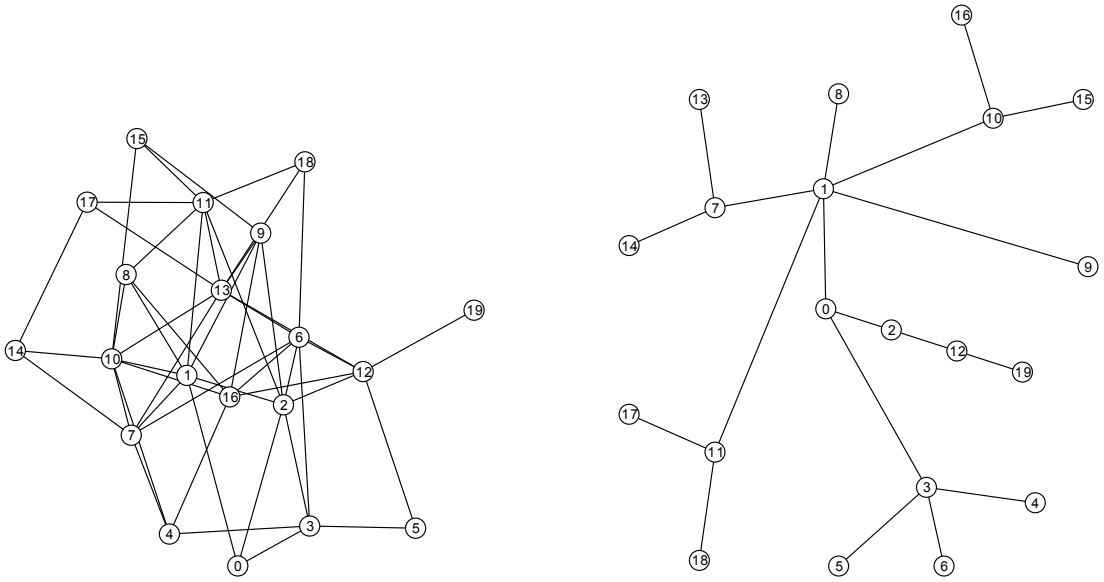
An interrupt-driven breadth-first search (BFS) is used because it should extract a wide, shallow tree from the SpiNNaker grid. This is desirable as it allows a large volume of the barrier, scatter and gather communication to occur in parallel. A sequential BFS is presently used because it ensures that labels are generated contiguously and will therefore be unique across the machine as illustrated by Figure 1.

1) General Algorithm Description: A node is represented as a finite-state machine and will be in one of the following states during the algorithm: `IDLE`, `LABELLED`, `PARENT`, or `BARRIER`. All nodes begin in the `IDLE` state and enter the `BARRIER` state once the tree has been successfully built. Query-events, $Q(L)$, and reply-events, $R(L, A)$, are used to transmit labels, L, between parent and child nodes and to report the number of nodes affected by an operation, A.

For a given node, V, in the `IDLE` state, a label will be attached upon the reception of an event, $Q(L)$, containing the new label value to use. V will then emit a reply-event, $R(L, 1)$, to the originator of Q to report that the label has been accepted (i.e., $A = 1$ because V was the only node affected by Q). V will then advance to the `LABELLED` state and hence a parent-child relationship has been established.

A node in the `LABELLED` state will receive an event, $Q(L)$, after the parent, V_P, has finished labelling its neighbour nodes. L will be the first value that may be used as a label. V will iterate over all neighbouring nodes (except V_P) sending $Q(L+n)$ where n is initialised to 0 and incremented for each $R(L+n, 1)$ reply. Neighbours that already have been assigned a label will respond with $R(L+n, 0)$ and hence n will not be incremented. This process will continue until all neighbours of V have been visited, causing V to respond to V_P with

 ISBN: 978-1-61208-274-5



(a) Arbitrary machine graph (b) Extracted tree

Fig. 1. Input graph and resulting tree for the proposed event-driven breadth-first algorithm.

$R(L+n, n)$ before entering the PARENT state. Using Figure 1 as an example and assuming that node 0 is in the LABELLED state (i.e., the host machine has assigned it a label of 0), it will pass $Q(1)$ to one of its neighbours which will then respond with $R(1, 1)$ to indicate that the label, 1, has been accepted. If the target neighbour already has a label, the reply will instead be $R(1, 0)$ because the label has not been accepted. The next neighbour receives $Q(2)$ and will reply with $R(2, 1)$, and so on until all neighbours have been labelled.

Once in this state, V performs similar behaviour but instead computes a running total of affected nodes, n_T, which is first initialised to 0 and then increased by n_R for each reply event $R(L + n_T + n_R, n_R)$ from a child node. V will respond to its parent, V_P, with $R(L + n_T, n_T)$ after all child nodes have been visited. Following from the previous example, node 0 would respond to its parent (the host, in this case) with $R(3, 3)$ because $L = 3$ was the highest label assigned to a neighbour and $A = 3$ nodes were affected.

A node may, at any time, receive a barrier event, $B(L)$, which immediately causes the node to perform the following actions: 1) store L as the number of nodes in the machine graph, 2) propagate $B(L)$ to all child nodes, and 3) transition into the BARRIER state. No BFS events will be processed in this state, hence this marks the completion of the algorithm.

2) Duty of the Root Node: The description in the previous section is valid for all nodes of the graph and of the derived tree, but the root node is required to behave slightly differently. Its parent is the *host* of the simulation and will not be part of the machine graph (an Ethernet-connected PC is the host of a SpiNNaker-based simulation). The host will issue $Q(0)$ to a node V_R of the target system to start the algorithm. V_R will assert itself as the root node of the machine because $L = 0$.

V_R will progress through the states in the same manner as any other node except that it does not require the permission of its parent to raise new events. It therefore issues new events, calculating appropriate values for L, and computes a running total of the number of affected nodes, A_R, for each pass. When $A_R = 0$, all nodes of the machine graph have been assigned a unique label and have progressed through all the required states, which triggers V_R to perform the following:

- $B(L_{max})$ is issued to all child nodes of V_R;
- $R(L_{max}, 0)$ is issued to the *host* of the simulation;
- V_R enters the BARRIER state.

Following the running example one final time with node 0 having labelled its neighbouring nodes and entered the PARENT state, a random child is chosen (node 3 in the case of Figure 1) and issued with $Q(4)$. Node 3 follows the same procedure as before by labelling its neighbours 4, 5 and 6. Once complete, node 3 replies to node 0 with $R(6, 3)$ because $L = 6$ is the highest label used and $A = 3$ nodes were affected. Next, node 1 is randomly chosen and issued $Q(7)$; $R(11, 5)$ is raised in response after each neighbour has been visited. Finally node 2 is issued $Q(12)$ and responds with $R(12, 1)$. Node 0 can now calculate the total number of affected nodes for this pass as $A_R = 3 + 5 + 1 = 9$. As $A_R \neq 0$, a random child node is passed $Q(13)$ and the process continues until nodes 1, 2, and 3 all respond with $R(L_{max}, 0)$, which causes node 0 to complete the actions described above. Figure 1a shows a random graph that has been used as a machine model for this algorithm. Each node represents a SpiNNaker chip and each edge is a nearest-neighbour (NN) connection. Figure 1b is the tree structure that has been built.

C. P2P Table Generation

Building the P2P routing tables currently uses the same machine geometry assumptions as the labelling process [9],

and must be replaced to use the BFS-assigned chip labels.

Assuming that all nodes of an arbitrary connected graph (i.e., all chips of a SpiNNaker machine) are in the BARRIER state, a node, V_i, transmits its label, i, to *all* neighbouring nodes including those that are not child nodes. A receiving node, V_k, with network ports $E(V_k)$ links the incoming port, $E_j \in E(V_k)$, to V_i by setting the ith entry of its routing table accordingly (i.e., $P_k[i] = E_j$). The message is then forwarded through all ports other than E_j to continue the process.

This bootstrap stage begins with the root node broadcasting its own label, 0, to its neighbours after all graph nodes have entered the BARRIER state. Nodes receiving label messages for the *first* time update their P2P table, propagate the message to all neighbours except the source, and then broadcast their own label. If a P2P table entry is already present then the node will not broadcast further messages. A wave front of these messages will propagate across the graph until all nodes have a complete P2P table, which will contain L entries as reported by the $B(L)$ message of the labelling stage.

A second barrier condition is required to conclude the bootstrap. Messages may now be routed between any two nodes of the machine graph dynamically by following the appropriate ports mapped in the P2P tables.

III. Mapping Problem Graphs to Machine Geometries

A SpiNNaker application is represented as a connected graph that describes how data flows through a set of behaviours. Mapping these *problem graphs* onto the machine is essentially a combination of assignment and path-finding problems. SpiNNaker is optimised for simulating large-scale neural networks in biological real-time, and hence existing methods exploit the hierarchy of these problem graphs to simplify allocation and routing [10][11].

These assumptions do not hold for general-purpose applications because there can be no guarantee of the structure of the problem graph. We are developing a physically-inspired approach that treats each node of the graph as a charged particle contained within a volume. The field interactions between nodes will distribute them evenly across the machine geometry. Additional forces acting in place of the edges will keep heavily connected areas local. Graph drawing and chip-layout algorithms have served as two key inspirations.

Our goal is to produce an algorithm that can be solved locally at each node without requiring any global knowledge of the machine or problem graphs. This cannot be included in the bootstrap because the application may change during runtime. An ongoing supervisory process may be able to adjust node and edge weights (i.e., their field contributions) to facilitate dynamic load balancing without requiring global knowledge.

IV. Conclusion and Future Work

We have presented a brief review of bootstrapping algorithms that we are developing for use on the SpiNNaker massively parallel computer. A breadth-first search based on node-local information and event driven interactions is used to assign unique labels to nodes (chips) and to support a barrier tree structure. At present, all algorithms have been tested in a simulation environment that accurately mimics the NN network. The BFS is sequential to guarantee unique labels across the machine but this leads to a computational complexity of $O(n)$. This is somewhat wasteful of the massively parallel resources of SpiNNaker, and further work will be conducted to parallelise this algorithm as much as is practicable.

Section III presents an idea we aim to develop that will allow arbitrary problem graphs to be mapped onto arbitrary machine geometries using only locally available knowledge. Our longer term goal is to couple these algorithms to design a system capable of reacting to system-level changes (such as a processor or chip malfunctioning) without using global knowledge or a central overseer. Additionally, they will allow a problem graph to be streamed into SpiNNaker and the allocation of problem nodes to cores, and the derivation of network routes, will be an automatic process.

Acknowledgment

This work is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grants EP/G015775/1 and EP/G015740/1, and with industrial support from ARM Ltd.

References

[1] SpiNNaker home page. University of Manchester. Last Accessed: May 2013. [Online]. Available: http://apt.cs.man.ac.uk/projects/SpiNNaker/

[2] S. Furber and A. Brown, "Biologically-Inspired Massively-Parallel Architectures - Computing Beyond a Million Processors," in *Int. Conf. on Application of Concurrency to System Design*. IEEE, 2009, pp. 3–12.

[3] L. Plana, S. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A GALS Infrastructure for a Massively Parallel Multiprocessor," *Design & Test of Computers*, vol. 24, no. 5, pp. 454–463, Sep. 2007.

[4] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown, "Overview of the SpiNNaker System Architecture," *IEEE Transactions on Computers*, pp. 1–14, 2012.

[5] D. Fick, R. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Sylvester, and D. Blaauw, "Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores," in *ISSCC*. IEEE, Feb. 2012, pp. 190–192.

[6] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, and Others, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *ISSCC*, vol. 9, no. 2. IEEE, Feb. 2010, pp. 108–109.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-c. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64? Processor: A 64-Core SoC with Mesh Interconnect," in *ISSCC*. IEEE, Feb. 2008, pp. 88–89.

[8] M. Khan, J. Navaridas, A. Rast, X. Jin, L. Plana, M. Lujan, J. Woods, J. Miguel-Alonso, and S. Furber, "Event-Driven Configuration of a Neural Network CMP System over a Homogeneous Interconnect Fabric," in *8th Int. Symp. on Parallel and Distributed Computing*. IEEE, 2009, pp. 54–61.

[9] T. Sharp, C. Patterson, and S. Furber, "Distributed configuration of massively-parallel simulation on SpiNNaker neuromorphic hardware," in *Int. Joint Conf. on Neural Networks*. IEEE, 2011, pp. 1099–1105.

[10] F. Galluppi, S. Davies, A. Rast, T. Sharp, L. Plana, and S. Furber, "A hierachical configuration system for a massively parallel neural hardware platform," in *Computing Frontiers*. ACM Press, 2012, pp. 183–192.

[11] S. Davies, J. Navaridas, F. Galluppi, and S. Furber, "Population-based routing in the SpiNNaker neuromorphic architecture," in *Int. Joint Conf. on Neural Networks*. IEEE, 2012, pp. 1–8.

 ISBN: 978-1-61208-274-5