Event-Driven Simulation of Arbitrary Spiking Neural Networks on SpiNNaker

Thomas Sharp, Luis A. Plana, Francesco Galluppi, and Steve Furber

School of Computer Science, The University of Manchester,
Oxford Road, Manchester, UK
thomas.sharp@cs.man.ac.uk

Abstract. Programming supercomputers correctly and optimally is non-trivial, which presents a problem for scientists simulating large areas of the brain. Researchers face the challenges of learning how to fully exploit hardware whilst avoiding the numerous pitfalls of parallel programming such as race conditions, deadlock and poor scaling. The SpiNNaker architecture is designed to exploit up to a million processors in modelling as many as one billion neurons in real-time. We present a programming interface for the architecture to allow modelling of arbitrary neuron and synapse dynamics using standard sequential C code, without concern for parallel-programming techniques or interprocessor communication mechanisms. An example is presented in which SpiNNaker is programmed to model multiple synaptic dynamics that are exchanged on the fly and the results of the different synaptic efficacies are shown.

Keywords: neural simulation, event driven, parallel programming, SpiNNaker, kernel, tasks, callbacks, C.

1 Introduction

Parallel computers are powerful tools for modelling large-scale, biologically-plausible Spiking Neural Networks (SNNs) [4]. However, the energy requirements of real-time brain-scale simulations in digital circuits [8] are prohibitively expensive and are expected to remain so for some decades [6]. *Neuromorphic engineering* [11] has produced analogue circuits which simulate neuron and synapse dynamics with great speed and energy efficiency, but at the expense of model adaptation in light of discoveries in 'wet' and computational neurosciences.

SpiNNaker is a digital many-core hardware architecture that aims to address these issues using low-power, general-purpose processors. The SpiNNaker operating system kernel must abstract the details of the hardware to allow researchers to easily simulate arbitrary neuron and synapse models whilst maintaining real-time performance and power efficiency. To this end, inspiration is taken from microcontroller kernels which operate under strict timing and energy requirements [5]. This paper describes event-driven computation as a solution to the problem of large-scale parallel-programming, presents the SpiNNaker Application Run-Time Kernel (ARK) and Application Programming Interface (API) and shows their use in modelling a number of synaptic dynamics.

B.-L. Lu, L. Zhang, and J. Kwok (Eds.): ICONIP 2011, Part III, LNCS 7064, pp. 424–430, 2011.
© Springer-Verlag Berlin Heidelberg 2011

2 SpiNNaker

SpiNNaker is a massively-parallel computing architecture designed to model billion-neuron, trillion-synapse SNNs in real-time. A SpiNNaker machine consists of up to 2^{16} multiprocessor chips (figure 1) containing eighteen low-power ARM processor cores dedicated to simulation of up to 10^3 neurons and 10^6 synapses each [10]. Processors communicate neural spikes and other simulation data via an on-chip router that also forms links with six neighbouring chips so that any processor may communicate with any other in the machine.

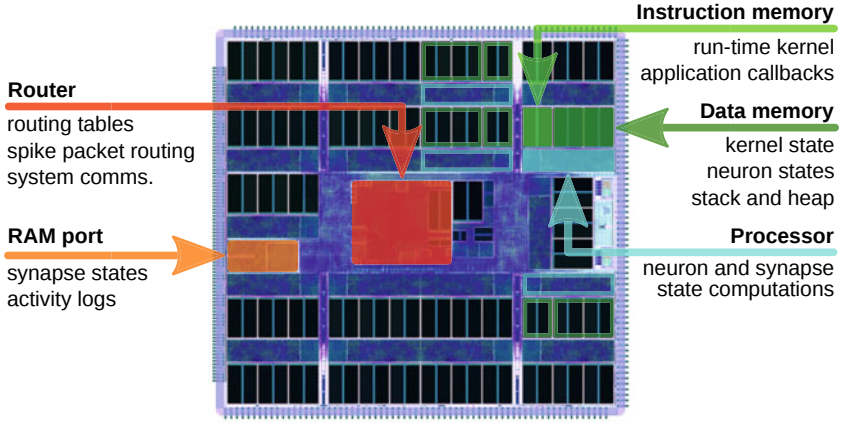

Fig. 1. A SpiNNaker chip. Each processor has interrupt control and timer peripherals and 32kB instruction and 64kB data memories. A shared 128MB off-chip memory is accessed by DMA transfers. A packet-switched router handles communications between processors on local and remote chips.

3 Event-Driven Neural Simulation

SpiNNaker applications are *event-driven* (figure 2) in that all computational *tasks* follow from events in hardware. Neuron states are computed in discrete timesteps initiated in each processor by a local periodic *timer event*. At each timestep processors evaluate the membrane potentials of all of their neurons given prior synaptic inputs and deliver a packet to the router for each neuron that spikes. Spike packets are routed to all processors that model neurons efferent to the spiking neuron. Receipt raises a *packet event* that prompts the efferent processor to retrieve the appropriate synaptic weights from off-chip RAM using a background Direct Memory Access transfer. The processor is then free to perform other computations during the DMA transfer and is notified of its completion by a *DMA done event* that prompts calculation of the sizes of synaptic inputs to subsequent membrane potential evaluations.

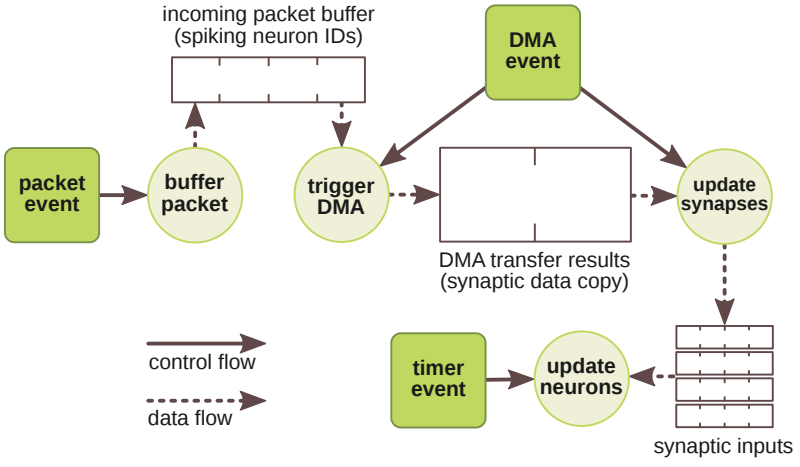

Fig. 2. Events and corresponding tasks in a typical neural simulation

Each SpiNNaker processor executes an instance of the Application Run-Time Kernel (ARK) which is responsible for providing computational resources to the tasks arising from events. The ARK has two threads of execution (figure 3) that share processor time: following events, control of the processor is given to the *scheduler* thread that queues tasks; upon its completion, the scheduler returns control to the *dispatcher* thread that dequeues tasks and executes them. In terms of figure 2, for example, a timer event schedules a neuron update task that is dispatched upon returning from the event.

Tasks have *priorities* that dictate the order in which they are executed by the dispatcher. The scheduler places each task at the end of the queue corresponding to its priority and the dispatcher continually executes tasks from the highest-priority non-empty queue. To facilitate immediate execution, priority zero tasks are *non-queueable* and are executed by the scheduler directly, precluding any further scheduling or dispatching until the task is complete.

The SpiNNaker Application Programming Interface (API) allows a user to specify the tasks that are executed following an event. The user writes *callback* functions in C that encode the desired tasks and then registers them with the scheduler against particular events. The following example lists callbacks to compute the Izhikevich equations (see [9] and [10] for details) on the timer event, to buffer packets and kickstart DMA transfers on a packet event and to start subsequent DMA transfers (conditional on receipt of further packets) and process synaptic inputs on the DMA done event. Three variants of the timer callback are provided which compute different dynamics of synaptic efficacy, namely, current-based instantaneous spike response synapses and current- and conductance-based synapses with first-order response dynamics [3, 1]. It should be noted that these models serve only to demonstrate the API; readers are referred to the citations for an explanation of the neural activity itself.

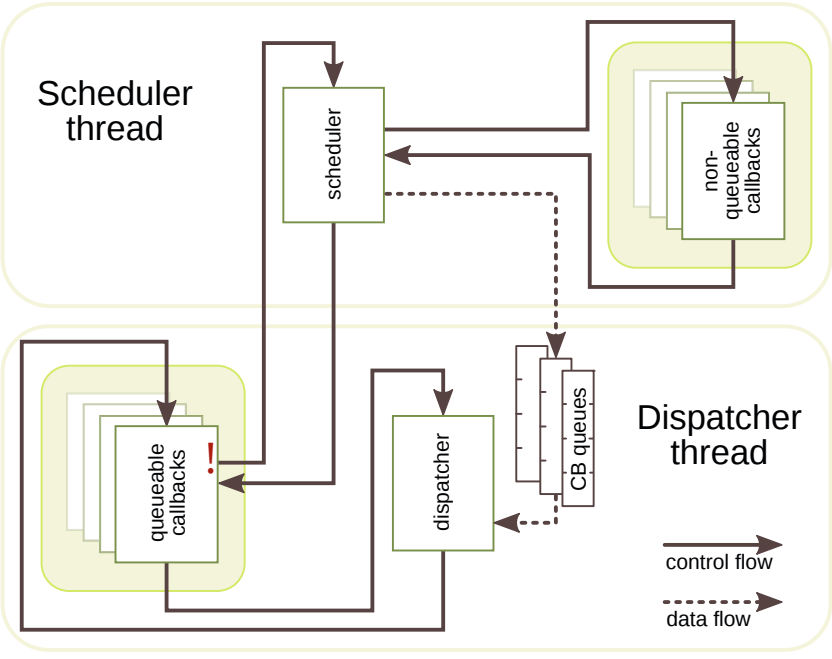

Fig. 3. Control and data flow between the scheduler and dispatcher threads

In the `main` function the timer callback of the first synapse model is registered along with the packet and DMA done callbacks. A simulation of a single neuron receiving input spikes every 16 milliseconds is run for 800 milliseconds. At $t = 400$ms and $t = 600$ms the timer callback registers a new callback in the scheduler, which causes a new set of synaptic dynamics to be computed on subsequent timer events. The code for functions provided by the API is not listed and for brevity only excitatory synaptic inputs (buffers denoted `epsp`) are shown.

```
int main() {
  // Call hardware and simulation configuration functions
  ...
  // Register callbacks and run simulation
  callback_on(PACKET_EVENT, packet_callback, PRIORITY_1);
  callback_on(DMA_DONE_EVENT, dma_done_callback, PRIORITY_2);
  callback_on(TIMER_EVENT, timer_callback_0, PRIORITY_3);
  start(800);
}

void feed_dma_pipeline() {
  // Start engine if idle and transfers pending
  if(!dma_busy() && !dma_queue_empty()) {
    void *source = lookup_synapses(packet_queue_get());
    dma_transfer(..., source, ...);
  }
}

void buffer_post_synaptic_potentials(synapse_row_t *synapse_row) {
  for(uint i = 0; i < synapse_row_length; i++) {
    // Get neuron ID, connection delay and weight for each synapse
```

```
    ...
    // Store synaptic inputs
    neuron[neuron_id].epsp[connection_delay] += synaptic_weight;
  }
}

void dma_done_callback(uint synapse_row, uint unused) {
  // Restart DMA engine if transfers pending
  feed_dma_pipeline();
  // Deliver synaptic inputs to neurons
  buffer_post_synaptic_potentials((synapse_row_t *) synapse_row);
}

void packet_callback(uint key, uint payload) {
  // Queue DMA transfer and start engine if idle
  packet_queue_put(key);
  feed_dma_pipeline();
}

void timer_callback_0(uint time, uint null) {
  if(time >= 400) callback_on(TIMER_EVENT, timer_callback_1, PRIORITY_3);
  for(int i = 0; i < num_neurons; i++) {
    uint current = neuron[i].epsp[time];
    // Compute neuron state given input and deliver spikes. See Jin et al.
    ...
    if(neuron[i].v > THRESHOLD){
      send_mc_packet(neuron[i].id);
    }
  }
}

void timer_callback_1(uint time, uint null) {
  if(time >= 600) callback_on(TIMER_EVENT, timer_callback_2, PRIORITY_3);
  for(int i = 0; i < num_neurons; i++) {
    uint current = neuron[i].epsp[time];
    // Compute neuron state given input and deliver spikes.
    ...
    if(neuron[i].v > THRESHOLD){
      send_mc_packet(neuron[i].id);
    }
    // Add an exponentially decaying quantity to next timestep's input
    neuron[i].epsp[time + 1] += current * neuron[i].decay;
  }
}

void timer_callback_2(uint time, uint null) {
  for(int i = 0; i < num_neurons; i++) {
    // Get synaptically induced conductance (scaled down)
    int conductance = neuron[i].epsp[time] / 64;
    // Compute current from conductance and membrane and eq. potentials
    int current = conductance * (neuron[i].v - EQUILIBRIUM_POTENTIAL);
    // Compute neuron state given input and deliver spikes.
    ...
    if(neuron[i].v > THRESHOLD){
      send_mc_packet(neuron[i].id);
    }
    // Add an exponentially decaying quantity to input at time + 1
    neuron[i].epsp[time + 1] += current * neuron[i].decay;
  }
}
```

The effect of each of the synapse models on the input terms and neuron membrane potential is shown in figure 4. Spikes are clipped from the top of the figure and synaptic currents are negatively offset for clarity. The instantaneous synapse response elicits a brief spike in the membrane potential of the neuron but does not provide enough drive to cause a spike. The first-order dynamics of

the next synapse model cause the neuron to integrate significantly more current and thus to spike. The neuron also spikes under the third synapse model and interactions between the membrane potential and input current are visible, such as where the membrane potential approaches 0 and the current term decreases correspondingly.

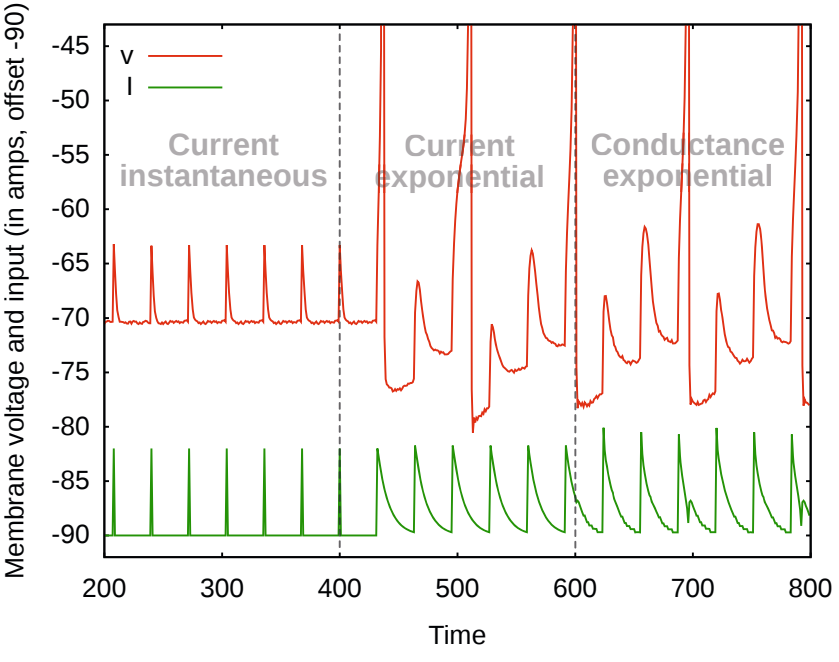

Fig. 4. Membrane potential and input current traces for three synapse models

4 Conclusion

Parallel computers are powerful tools in neuroscientific research but programming them is non-trivial. Researchers face two particular challenges to exploiting supercomputers: the difficulty of parallel programming, in terms of both program optimality and correctness; and knowledge of the numerous capabilities and communication mechanisms of parallel computing hardware. The Application Run-Time Kernel and Application Programming Interface presented in this paper largely addresses these concerns; researchers use template programs and implement just the sequential, standard C code required to compute neuron and synapse dynamics or borrow from a repository of established models. Network structure (the number of neurons and their connectivity) is then specified at a desktop computer using PyNN [2] from which the simulation data structures are compiled and loaded into the machine [7]. Parallelism is achieved transparently by executing sequential programs in numerous processors simultaneously and interprocessor communication is handled in terms of neural spikes, which are

transmitted by an API function call and received by a callback registered against the packet received event in the ARK. Thus, a model is provided for exploiting a massively-parallel computer for real-time simulation of large-scale spiking neural networks without the otherwise major challenges of ensuring correctness and optimality of large parallel programs.

References

1. Brette, R.: Exact simulation of integrate-and-fire models with synaptic conductances. Neural Computation 18, 2004–2027 (2006)
2. Davidson, A.P., Brüderle, D., Eppler, J.M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., Yger, P.: PyNN: a common interface for neuronal network simulators. Frontiers in Neuroinformatics 2, 1–10 (2009)
3. Dayan, P., Abbott, L.F.: Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems, 1st edn. The MIT Press (2001)
4. de Garis, H., Shuo, C., Goertzel, B., Ruiting, L.: A world survey of artificial brain projects, part I: Large-scale brain simulations. Neurocomputing 74, 3–29 (2010)
5. Emerson, M., Neema, S., Sztipanovits, J.: Handbook of Real-Time and Embedded Systems. CRC Press (2006)
6. Furber, S., Temple, S.: Neural systems engineering. Journal of the Royal Society Interface 4, 193–206 (2006)
7. Galluppi, F., Rast, A., Davies, S., Furber, S.: A General-Purpose Model Translation System for a Universal Neural Chip. In: Wong, K.W., Mendis, B.S.U., Bouzerdoum, A. (eds.) ICONIP 2010, Part I. LNCS, vol. 6443, pp. 58–65. Springer, Heidelberg (2010)
8. Gara, A., Blumrich, M.A., Chen, D., Chiu, G.L.T., Coteus, P., Giampapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Liebsch, T.A., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vranas, P.: Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development 49, 195–212 (2005)
9. Izhikevich, E.M.: Simple model of spiking neurons. IEEE Transactions on Neural Networks 14, 1569–1572 (2003)
10. Jin, X., Furber, S.B., Woods, J.V.: Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In: International Joint Conference on Neural Networks, pp. 2812–2819 (2008)
11. Mead, C.A.: Analog VLSI and neural systems. Addison-Wesley (1989)