Distributed event-based computing

Andrew BROWN[a1], David THOMAS[b], Jeff REEVE[a], Ghaith TARAWNEH[c], Alessandro DE GENNARO[c], Andrey MOKHOV[c], Matthew NAYLOR[d] and Tom KAZMIERSKI[a]

[a] *Department of Electronics, University of Southampton, SO17 1BJ, UK*
[b] *Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, UK*
[c] *Electrical and Electronic Engineering, University of Newcastle, NE1 7RU, UK*
[d] *Computer Laboratory, University of Cambridge, CB3 0FD, UK*

Abstract. As computing systems get larger in capability - a good thing - they also get larger in ways less desirable: cost, volume, power requirements and so on. Further, as the datastructures necessary to support large computations grow physically, the proportion of wallclock time spent communicating increases dramatically at the expense of the time spent calculating. This state of affairs is currently unacceptable and will only get worse as exa-scale machines move from the esoteric to the commonplace. As the unit cost of non-trivial cores continues to fall, one powerful approach is to build systems that have immense numbers of relatively small cores embedded (both geometrically and topologically) in a vast distributed network of stored state data: take the compute to the data, rather than the other way round. In this paper, we describe POETS - **P**artially **O**rdered **E**vent **T**riggered **S**ystems. This is a novel kind of computing architecture, built upon the neuromorphic concept that has inspired such machines as SpiNNaker[1,2] and BrainScaleS[3]. The central idea is that a problem is broken down into a large set of interacting *devices*, which communicate asynchronously via small, hardware brokered *packets* (the arrival of which is an *event*). The set of devices is the **task graph**. You cannot take a conventional codebase and port it to a POETS architecture; it is necessary to strip the application back to the underlying mathematics and reconstruct the algorithm in a manner sympathetic to the solution capabilities of the machine. However, for the class of problems for which this approach is suitable, POETS has already demonstrated solution speedups of a factor of 200 over conventional techniques.

Keywords. Multicore/manycore systems, heterogeneous systems, accelerators

Introduction

Previously in this conference series[4] we have described the SpiNNaker system[1,2] consisting (when fully assembled) of just over a million ARM-9 cores, interconnected via a bespoke, high-speed, asynchronous network. The network is implemented as a flat triangular mesh of nodes (with 16 application cores at each node) with opposite edges identified such that the entire structure configures as a torus. SpiNNaker is a neural network simulator: the design intent is that each core hosts ~1000 neuron models, which communicate with each other using short (32 bit) hardware messages. The mapping of the (problem specific) neural network topology onto the *a priori* fixed communications network is achieved by a pre-processing stage

[1] Corresponding Author.

that programs hardware message routers resident on each node. At the time of writing, the machine is half assembled; when complete, the intention is that it should be able to simulate ensembles of up to 10^9 neurons (1% of the human brain) in real time. It achieves this by disregarding a number of the historical axioms of computer design:

- It is computationally asynchronous: there is no central overseer clock.
- The state memory is distributed throughout the physical system, and cores have no visibility of any memory other than that which is local to them.
- Communication is via short, hardware brokered packets. Packet transits are non-deterministic (once launched, the sender loses visibility of the packet, and until it physically arrives, the receiver has no visibility or knowledge of the impending arrival. Packets can take an unpredictable amount of time to arrive (although the nearest-neighbour node-node transit delay is O(0.1)us, the overall multi-node-node trajectory may change dynamically), and *in extremis* it is possible for the communication stream to be non-transitive.

This is a computational model of immense power and potential: if a distributed model of a system can be constructed, a distributed computation engine can simulate or operate on it, exploiting the physical parallelism of the architecture (in principle, SpiNNaker can do 10^6 operations simultaneously), unthrottled by concerns about communication bandwidth.

Work reported previously[5,6] described the utilisation of event-based concepts in non-neural application domains. Whilst exploring these possibilities, a number of hard constraints of the SpiNNaker architecture were identified that made the utility of SpiNNaker for non-neural simulation not as well suited as may have been wished. As SpiNNaker is an ASIC, there is little possibility for meaningful workarounds.

This paper is about a next-generation event-based system: POETS (**P**artially **O**rdered **E**vent **T**riggered **S**ystems). Architecturally, POETS is derived from and contains several similarities to the progenitor SpiNNaker system:

- Information is conveyed via packets (SpiNNaker uses 32 *bits*, POETS 64 *bytes*); the asynchronous arrival of a packet at a handler causes the receiving hardware to react to the packet arrival (akin to a process `wait` in a HDL). The increased size of the POETS packets makes possible (relatively) complicated communications protocols impossible to realise in SpiNNaker.
- State data is distributed: SpiNNaker has ~64 bits/neuron, POETS has ~1kbyte/task graph node.

At other points of comparison, the systems start to diverge:

- SpiNNaker is composed of ASICs, each node containing 16 ARM 9 cores (1 monitor/local overseer, 16 workers and 1 spare). There is no floating point capability. POETS is designed around FPGA technology; each node (FPGA) has available to it a hard local mothercore plus several thousand RISC-V application threads. All of these have floating point capability.
- Each SpiNNaker core has 64kbytes of data and 32kbytes of close coupled instruction memory available to it; each node (16 application cores) has shared access to 2GByte of DDR RAM. Each POETS core has 1M data memory and 32kbyte of instruction memory. Almost all of the POETS DRAM is offboard, but fast burst access and aggressive cache design makes this (almost) transparent to the user/software architect.

By far the most significant difference lies in the way packets are communicated. In any packet-based communications system with finite internal buffering, if material is injected into the infrastructure faster than it is removed, something must give: either the communications system must refuse to accept further packet injections, or packets must be dropped.

In SpiNNaker, once a packet has been injected into the communication system (which can never fail to happen), it is entirely controlled by the hardware: neither the sender nor putative receiver has any control or visibility over the packet. This has the advantage of a clean and simple programming model: send and receive are completely independent - senders *send* and forget, receivers *react* when a packet is delivered. The packet trajectory is unknown to the task graph (and may change in real time). Herein lies a disadvantage: if local traffic density overwhelms a node router, the only option is to drop the packet (or buffer it for later re-injection: this just moves the problem). Neither the putative receiver nor the sender can have knowledge of this event; to tell either would require more packets, exacerbating the traffic density problem.

In POETS, packet launch is proscribed until and unless the hardware can guarantee (at least part of) the route is open. Whilst this does not *solve* the problem of local congestion, it *moves* it to the point at which it can be most responsibly addressed: the sending component. The sender can

- Abandon the send attempt.
- Repeat the attempt at some future controlled (real) time.
- Modify the packet and try again.

The central point is that the *task* knows there is a transient communication difficulty, and can react accordingly.

In this paper, we describe

- The hardware POETS architecture.
- The software middleware necessary to map an (arbitrary topology) task graph to the underlying fixed processor graph.
- The methodology designed to describe and define POETS applications.
- A typical POETS application. You cannot take a conventional codebase - no matter how carefully and professionally crafted - and port it to POETS. It is necessary to strip the application back to the underlying mathematics and reconstruct the algorithms in a manner sympathetic to the solution capabilities of POETS.
- Some simulation data showing the potential power of the technique. SpiNNaker exists - the POETS system is still in the early phases of construction. These simulations - conservatively - indicate a wallclock acceleration for the problems considered of approaching three orders of magnitude.

This last point is of fundamental importance: it is not an incremental speed increase, it is a step-change in capability, and brings with it a significant impact. Problems once the domain of supercomputer clusters are bought within range of commodity hardware. Bringing machine response times down to the same timescales as human thought allows different working methodologies. Users no longer require supercomputer access and budgets - with a POETS machine, engineers can play around with ideas, rather than plan computational strategies weeks in advance. This is a double-edged sword: used sensibly it can provide immense advantages to the user base.

1. System overview

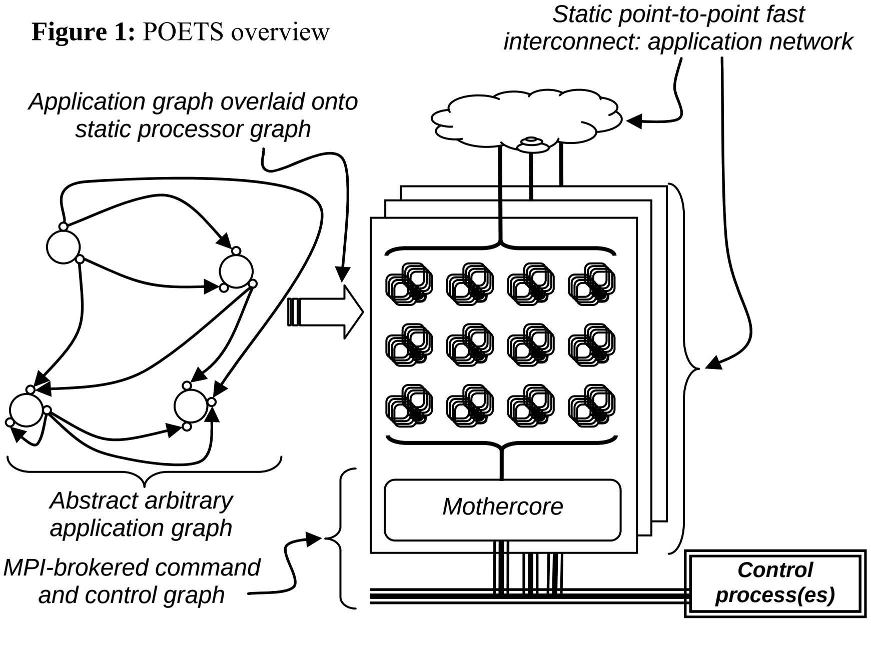
POETS consists of three graphs, two physical, one abstract, as in figure 1:

- A fixed physical graph composed of many (tens of thousands) of compact cores (in our initial implementation, these are hyperthreaded RISC-V cores, realised on a bank of FPGAs). These are interconnected by a fast bespoke communication fabric, described in section 2.
- An arbitrary abstract application (task) graph, which embodies the graphical representation of the domain-specific problem to be addressed. This is mapped onto the physical graph by a layer of middleware (section 3). The vertices of the application graph are represented as small, self-contained state automata, and the connectivity of the graph absorbed into the routing of the physical graph, so for all intents and purposes, *from the perspective of the application graph*, the physical graph effectively disappears - the only connectivity seen by the application vertices is that of the application edges.
- Sitting alongside the fixed physical RISC-V graph is a higher-level graph structure. The RISC-V cores are implemented on FPGAs; associated with each physical FPGA is a hard mothercore. These mothercores run a conventional operating system, have significant hardware resources, and are interconnected via a conventional MPI universe. This component of the system provides conventional access to the compute nodes: it is used for command and control, and data in- and exfiltration.

2. Machine hardware architecture

Efficient communication and low power consumption are two key goals in the construction of distributed systems. Both these requirements can be met using commodity hardware components: FPGA development boards. These boards combine state-of-the-art networking facilities with reconfigurable logic which, when customised

Figure 1: POETS overview


Static point-to-point fast interconnect: application network
Application graph overlaid onto static processor graph
Abstract arbitrary application graph
MPI-brokered command and control graph
Mothercore
Control process(es)

to a particular application or application domain, can offer better performance-per-watt than other commodity devices such as CPUs and GPUs.

FPGA-based systems have drawbacks: low-level hardware description languages and long synthesis times are barriers to productivity for application developers. An attractive approach is to provide a **soft-core overlay architecture** on top of the FPGA logic that can be programmed quickly and easily using standard software languages and tools. While this overlay is not customised to a particular POETS *application*, it is at least customised to the POETS application *domain*. This is a natural first step because higher levels of customisation, using high-level behavioural synthesis, are still insufficiently developed.

Here is described the soft-core overlay architecture of POETS and the initial machine built using a network of commodity development boards.

2.1. Compute subsystem

A customised 32-bit RISC-V core forms the basis of POETS. This is heavily multithreaded, supporting up to 32 threads. Crucially, this enables the core to tolerate the inherent latencies of deeply-pipelined FPGA floating-point operations (execution of a multi-cycle operation blocks only the issuing thread; other threads remain runnable). In the same way, multithreading also tolerates the latency of arbitration logic, allowing aggressive sharing of large components such as FPUs and caches between cores. This kind of sharing reduces FPGA area significantly, allowing more cores per FPGA, without compromising system throughput.

At most one instruction per thread is allowed in a core pipeline at any time, eliminating all control and data hazards. This leads to a small, simple, high-frequency design that is able to execute one instruction per cycle provided there are sufficient parallel threads (which is the case for POETS).

Custom instructions are provided for sending and receiving messages. Threads are automatically suspended when they become blocked on an event, for example waiting to receive a message, and are automatically resumed when the event is triggered. This results in a simple and efficient programming model, avoiding the low-level interrupt handlers that are required in the SpiNNaker system.

2.2. Memory subsystem

One of the design requirements is that each thread has access to a generously-sized private data segment, around 1MB in capacity. Assuming a large number of threads per FPGA, this kind of memory demand can only be met using off-chip DRAM.

FPGA boards typically provide a number of high-bandwidth DRAMs and it is essential to exploit spatial locality for efficient access. One way to achieve this (employed by the SpiNNaker system) is to require the programmer to use a DMA unit to explicitly transfer regions of data between DRAM and a small, core-local SRAM.

A different data cache philosophy has been developed for POETS. The cache is partitioned by thread id (the hash function combines this with some number of address bits) and permits at most one request per thread to be present in the cache pipeline at any time. Consequently, there is no aliasing between threads and all data hazards are eliminated, yielding a simple, non-blocking design that can consume a request on every cycle, even if it is a miss. This full-throughput cache can usefully be shared by up to four cores, based on the observation that an average RISC workload will access data memory once every four instructions.

2.3. Communication subsystem

One of the challenges in supporting 64-byte POETS messages is the amount of serialisation/deserialisation required to get a message into/out-of a 32-bit core. The POETS solution is to use a dual-ported memory-mapped *mailbox* with a 32-bit port connected to the core and a much wider port connected to the on-chip network. The mailbox stores both incoming and outgoing messages. A message can be forwarded (received and sent) in a single instruction, which is useful to implement efficient multcasting in software. A single mailbox can be shared between several cores, reducing the size of the on-chip network needed connect the mailboxes together.

Fundamental to POETS is the ability to scale the hardware to an arbitrary number of cores, and hence multiple FPGAs. The inter-board serial communication links available on modern FPGAs are both numerous and fast but, like all serial links, some errors are inevitable. Even if there is only one bit error in every thousand billion bits (10^12), that is still an error every ten seconds for a 10Gbps link. Therefore, on top of a raw link is placed a 10Gbps Ethernet MAC, which automatically detects and drops packets containing CRC errors. On top of the MAC is placed a POETS-specific window-based reliability layer that retransmits dropped packets. The use of Ethernet allows the use of mostly standard (and free) IP cores for inter-board communication. Since these links are used point-to-point, almost all of the Ethernet packet fields are be used for POETS purposes, resulting in very little overhead on the wire.

2.4. Initial POETS machine

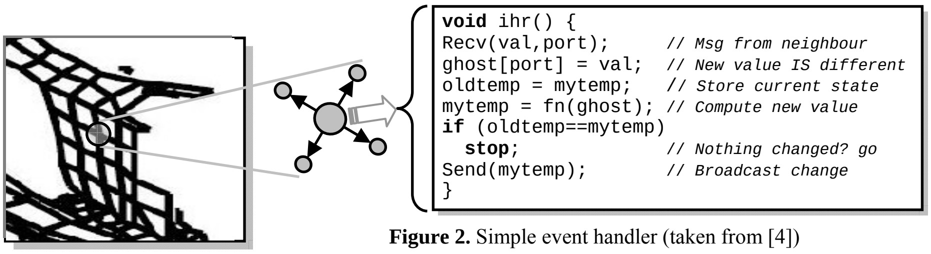
The first POETS prototype is based around the DE5-Net FPGA board. It consists of four multi-FPGA **boxes**, each containing ten FPGA boards and a modern PC acting as a mothercore in each box. The ten FPGAs are connected together as a mesh and also to the mothercore via a PCIe link. Four boxes when connected together will provide a mesh of 40 FPGAs and 4 mothercores. The intention is that each FPGA will host around a hundred RISC-V cores (thousands of RISC-V threads). The prototype will provide around 4000 cores in total, supporting the overall design intent that each RISC-V thread will host ~1000 POETS devices.

3. The software middleware

Controlling - orchestrating - the initialisation, layout and overall flow of the data through the machine is a heterogeneous multi-process program called the **Orchestrator**. This executes processes on conventional compute clusters attached to the MPI backplane, *and* the FPGA mothercores. The principal tasks for which the Orchestrator is responsible are

- ***Machine discovery:*** The MPI Universe is obviously discovered by the MPI subsystem, and the POETS engine relies on the continued integrity of this system throughout the execution cycle. Less predictable, however, is the array of RISC-V cores. A headline goal of all our event-based research is to build systems with core counts that are huge even by the standards of today: when processing sizes exceed a million cores, it is unreasonable - or at least uneconomic - to expect everything to function perfectly 'out of the box'. We assume the integrity of certain set of core components (quis custodiet ipsos custodes), and allow as much of the rest of the system to be dynamically discovered. The MPI network sends interrogating discovery packets into and through the application network that enable the discovery of the functioning topology. The outcome of this phase is that the Orchestrator builds within it a model of the available **application network**.

- ***Application graph construction:*** The domain specific application graph is handed into the Orchestrator from the domain-specific front end; the Orchestrator builds within it a model of the **application graph**.
- ***Application mapping:*** The principle initialisation task of the Orchestrator is to create a mapping of *application graph* to *application network*. The algorithms employed here borrow heavily from the world of automated place and route in electronic design automation. Prima facie, the aims here are to allocate application graph vertices (devices) to physical cores such that the overall communication cost is minimised, but dynamic network congestion and balancing must also be taken into account.
- ***Data in/exfiltration:*** Any software system - and POETS is no exception - needs interaction with the outside world if it is to produce useful behaviour. The application graph - set up by the domain-specific user (section 4) is described in terms of a local namespace; the task of this component of the Orchestrator is to support the maintenance of IO channels from the outside world through to the individual RISC-V threads supporting the application devices. POETS offers massive internal parallelisation - which is how it delivers the speedups it can - but has *relatively* low bandwidth connecting to the 'conventional' compute environment. The Orchestrator is responsible for overseeing a sensible triage/compression step in the datapath to the outside world.



```
void ihr() {
Recv(val,port);      // Msg from neighbour
ghost[port] = val;   // New value IS different
oldtemp = mytemp;    // Store current state
mytemp = fn(ghost);  // Compute new value
if (oldtemp==mytemp)
  stop;              // Nothing changed? go
Send(mytemp);        // Broadcast change
}
```

Figure 2. Simple event handler (taken from [4])

4. Applications: description methodology

4.1. Canonical heat diffusion

To illustrate the concepts underlying POETS programming, we re-visit an example from[4]: the canonical steady-state heat diffusion equation. The traditional (Gauss-Seidl or Jacobi) approach to this is to iterate over some discrete mesh, 'relaxing' the state of each grid point. Numerically, there is no reason why this loop should be processed in any particular order, and in an event-based system, there *is* no outer loop. Each grid point (application device) is represented by a state value (the 'temperature') and an event handler - see figure 2. When any point changes its state, it sends an asynchronous event to its neighbours in the graph; they react, each sending out further packets as described in the code fragment in figure 2. In principle, each thread can be executing a state update in parallel.

4.2. Heat diffusion - performance

Figure 3 shows the execution time of the technique (run on SpiNNaker) compared to a conventional sequential program. The key point is not specifically how much faster the event-based approach is, but the scaling implied by the curves - as the problem gets larger, event-based technology simply uses more and more threads to deliver flat scaling (ultimately the machine capacity will be exceeded, and the scaling will change accordingly).

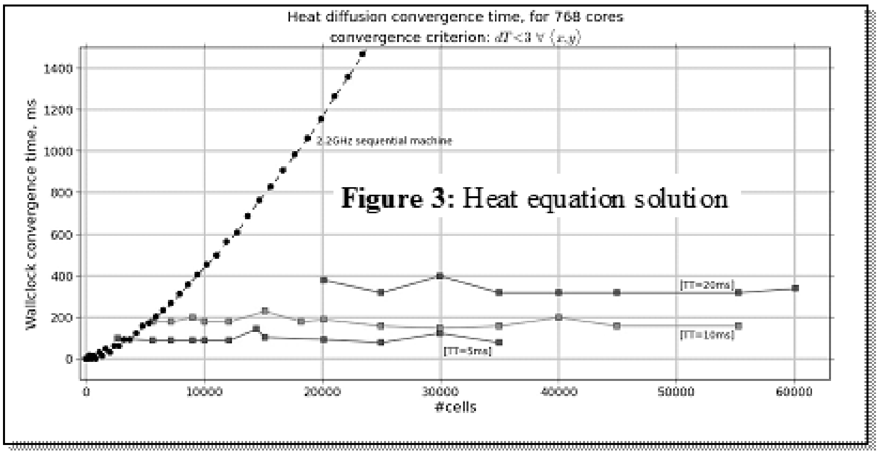


Figure 3: Heat equation solution

4.3. Analysis of Protein Interaction Networks for Drug Discovery

Unusual complex networks are becoming increasingly common nowadays given the explosion in data scale and availability. Recent medical advances, for example, have made it possible to collect vast amounts of data and construct interaction networks or **interactomes** for genes, proteins and other building blocks at various levels of biological organization. These networks are used in a number of scientific/medical applications and being able to analyze and process them efficiently is driving development in these fields. Here is discussed how POETS can be used for analyzing protein interaction networks for *drug discovery*.

In this domain, *protein interaction networks* are used to evaluate the impact of many components (drug candidates) on a cellular system. Each drug candidate has known effects on some protein interactions and its impact can be evaluated by disconnecting the network edges corresponding to these interactions then traversing the network to calculate certain impact metrics (for example, average shortest path. Why do we care? This metric is an indication of how robust a cause-effect sequence may be

Figure 4: A typical interaction network

in a drug therapy). This computation can be performed on a POETS machine: (1) the network is mapped to the core mesh, allocating one thread per protein (or a group of proteins). (2) A classic breadth-first traversal via message passing is performed, simulating a node discovery process, and starting from a selected node S. During the traversal, each visited node retains the distance from S (which is included in the incoming message) as part of its state. Once traversal is complete, the average shortest path from S is calculated by aggregating these distances. Note that the time taken to complete the traversal is proportional to the *diameter* (the largest possible separation of any two vertices) of the network, *not* to the number of its vertices or edges. Impact metrics are typically averaged over starting nodes, and traversal is repeated using different start nodes. Traversal operations can be performed in parallel by including S in each message and maintaining a table of shortest path lengths/node (with entries for the minimum distance to every other node).

This approach is fundamentally different from conventional (shared-memory) parallel architectures where network representation and traversal state are kept in a centralized memory and accessed by all worker cores. POETS offers better scalability and the prospect of constant execution time (up to the capacity of the architecture) since larger networks can be traversed just as fast simply by recruiting more cores to the computation. This is possible because POETS takes the compute to the data, rather than the other way around.

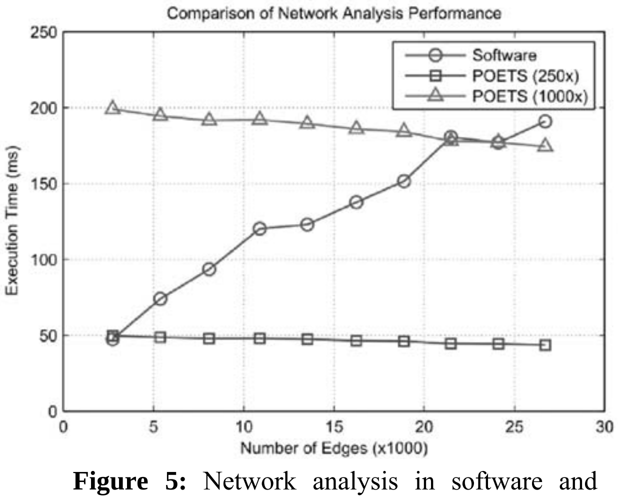


Figure 5: Network analysis in software and POETS. *Note that POETS execution times are multiplied by 250 and 1000 for ease of comparison.*

4.4. Drug discovery - performance

The scalability of the approach is illustrated by a cut-down POETS machine on a single FPGA[4]. This implementation uses individual flip-flops as devices that are either *visited* or not. On each clock cycle, any devices with at least one visited neighbour get marked as visited themselves. Network traversal is conducted by initially resetting all devices, setting one device (the starting node S) to visited then letting this value (the visited state) propagate through the network. The number of nodes discovered on each clock cycle is counted and this is used to calculate average shortest path. While substantially simplified, this implementation retains enough of the qualitative characteristics of the event-based paradigm to be demonstrated.

Figure 5 shows the scalability of POETS-based network traversal versus a single-threaded software implementation in C++. The number of active edges in a realistic protein interaction network is varied and calculated execution time for both POETS and the software implementation (for fairness, we exclude graph parsing, allocation and other initialization costs). The plot highlights two important aspects:

- In terms of absolute performance, POETS achieves a considerable speedup; between 2~3 orders of magnitude depending on the number of network edges.
- While software execution time scales linearly with the number of network edges, the execution time of POETS is constant. Enabling more network edges tends to reduce the network diameter and therefore has the counter-intuitive effect of reducing execution time.

5. Closing comments

Event-based (atomic) computing is a powerful new computing paradigm. Application domains typically the preserve of powerful supercomputers can potentially migrate to the desktop, with the changes in working practice that that implies. Identified domains that are amenable to event-based processing include simulation (continuous and discrete, particle and mesh), deep data mining, modelling (weather, plasma, financial, image processing), and the list is growing.

References

[1] S.B. Furber, D.R. Lester, L. Plana, J.D. Garside, E. Painkras, S. Temple, and A.D. Brown, "Overview of the SpiNNaker system architecture", IEEE T Computers, **62**, no 12, Dec 2013, pp2454-2467, doi 10.1109/TC.2012.142

[2] E. Painkras, L.A. Plana, J.D. Garside, S. Temple, F. Galluppi, C. Patterson, D.R. Lester, A.D. Brown, and S.B. Furber, "SpiNNaker: A 1W 18-core system-on-chip for massively-parallel neural network simulation", IEEE J Solid-State Circuits, **48**, no 8, pp 1943-1953. doi:10.1109/JSSC.2013.2259038

[3] M. Ehrlich, C. Mayr, H. Eisenreich, S. Henker, A. Srowig, A. Grübl, J. Schemmel, R. Schüffny. (2007). "Wafer-scale VLSI implementations of pulse coupled neural networks." Proc IEEE Conf on Sensors, Circuits and Instrumentation Systems.

[4] A. Mokhov, A. de Gennaro, G. Tarawneh, J. Wray, G. Lukyanov, S. Mileiko, J. Scott, A. Yakovlev, A.D. Brown. "Language and hardware acceleration backend for graph processing" Forum Design Languages (FDL), 2017, IEEE.

[5] A.D. Brown, J.S. Reeve, K.J. Dugan and S.B. Furber, "Atomic computing - a different perspective on massively parallel problems", ParCo'13, Munich

[6] K.J. Dugan, A.D. Brown, J.S. Reeve, R.M. Mills and S.B. Furber "Reliable computation with unreliable computers", DATE'15, Grenoble