POETS: Distributed event-based computing - scaling behaviour

Andrew BROWN[a,1], Mark VOUSDEN[a], Alex RAST[a], Graeme BRAGG[a], David THOMAS[b], Jonny BEAUMONT[b], Matthew NAYLOR[c], and Andrey MOKHOV[d]
[a] *Electronics and Computer Science, University of Southampton, UK*
[b] *Electrical and Electronic Engineering, Imperial College London, UK*
[c] *Computer Laboratory, University of Cambridge, UK*
[d] *Electrical and Electronic Engineering, University of Newcastle, UK*

Abstract. POETS (Partially Ordered Event Triggered Systems) is a significantly different way of approaching large, compute intensive problems. The evolution of traditional computer technology has taken us from simple machines with tiny memory and (by todays standards) glacial clock speeds, to multi-gigabyte architectures running orders of magnitude faster, but with the same fundamental process at the heart: a central core doing one thing at a time. Over the past few years, architectures have appeared containing multiple cores, but exploiting these efficiently in the general case remains a 'holy grail' of computer science. POETS takes an alternative approach, made possible only today by the proliferation of cheap, small cores and massive reconfigurable platforms. Rather than program explicitly the behaviour of each core and each communication between them, as is done in conventional supercomputers, here the programmer defines a set of relatively small, simple behaviours for the set of cores, and leaves them to get on with it - with the right behavioural definitions, the system 'self-organises' to produce the desired results.

Keywords. Multicore/manycore systems, Heterogeneous systems, Accelerators

1. Introduction

Moores Law[1]: the number of transistors on a chip doubles every 18 months or so. Dennard scaling[2]: as transistors get smaller, the power density stays constant, so dissipated power remains proportional to area. Koomeys Law[3]: the number of computations per joule of energy dissipated increases in line with Moores Law.

These principles have guided commentary on the computing industry for a long while. Two are exponentials, (and no exponent is sustainable indefinitely in nature), and the other runs into trouble in the opposite direction: semiconductor device physics cannot avoid leakage and quantum effects forever. However, they are all - quite soundly - based on physical effects, and are the domain of the fabrication engineer.

A parallel problem is the continued absence of any general theory of parallel computing. Technology gives us a *new* Moores Law: the number of *cores* on a silicon platform rises exponentially and starts to push at the boundaries of manageability - a new roadblock, alongside the power wall, the memory wall, process spread....[7]. In a conventional parallel system, huge swathes of data are moved around to benefit from the compute capabilities afforded by multiple processors. Bubbles in pipelines must be filled. Every cycle of every thread must produce useful data: The Beast must be fed. The choreography of this dance is controlled - designed - by the software architect, and

[1] Corresponding author: Andrew Brown, Electronics and Computer Science, University of Southampton, , Hampshire, SO17 1BJ UK; adb@ecs.soton.ac.uk

in the vast majority of cases the *complexity* issue is side-stepped by making much of the compute functionality exact duplicates of some cornerstone behaviour. What cannot be side-stepped by this technique are the *relative* costs of communications and compute. As computation grows in size, so too do the necessary support datastructures, and the proportion of wallclock spent communicating increases unhelpfully at the expense of the time spent computing. Fabrication technology is realising exa-scale *compute*, but simultaneously exposing the problems intrinsic to exa-scale *communication*.

Concurrent **event-based computing** is an approach intended to address simultaneously the complexity and the communication problems. The foundation work in this space has been reported previously in this conference series[4] and elsewhere [5-6,8]. In essence, the idea is that vast numbers of tiny compute units, each with a small amount of state, interconnected by a narrow but fast (hardware brokered) communications fabric, carrying information in small, fixed size packets, can provide far superior performance in terms of cost and power dissipation - and in some cases, also compute capability. In this paper, we discuss firstly the concept in general terms, and then provide an outline of a prototype architecture, designed to exploit the idea of computation based around an *un*choreographed non-deterministic 'packet storm'. We then provide some initial physical scaling measurements derived from two application areas that have been implemented on the event-based architecture.

2. The concept

Without loss of generality, consider the numerical solution of some physical matrix-based (discrete grid) problem using an iterative process - Gauss-Seidl or Jacobi, for example. Note there is no requirement for regularity or any kind of dimensional planarity. The solution process will consist of some number of embedded loops, or some kind of traversal sequence, moving over the data points of the grid in some trajectory determined by the programmer. At each point, the local state is updated by a function of some set of physically adjacent states, and computation moves on. The solution trajectory is deterministic, and dictated by the programmer. It is not until the system reaches some form of numerical equilibrium that we assign physical meaning to the numerical results. If we have a single thread machine, we may use Gauss-Seidl for

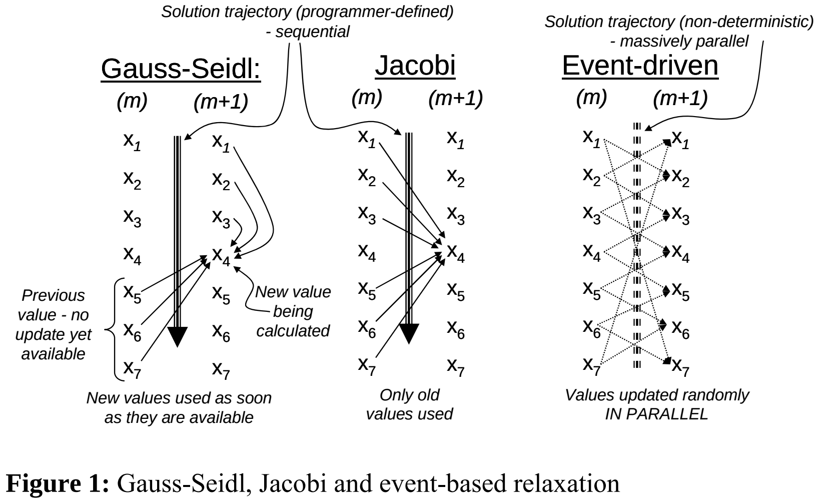
Solution trajectory (programmer-defined) - sequential
Solution trajectory (non-deterministic) - massively parallel
Gauss-Seidl:
Jacobi
Event-driven
(m)
(m+1)
x_1 x_2 x_3 x_4 x_5 x_6 x_7
Previous value - no update yet available
New value being calculated
New values used as soon as they are available
Only old values used
Values updated randomly IN PARALLEL

Figure 1: Gauss-Seidl, Jacobi and event-based relaxation

a fast convergence. If we have multiple cores available, we can use Jacobi (over) relaxation and double-buffer the data, achieving still faster overall performance. The numerical solution sequence *at each grid point* converges more slowly than the comparable trajectory in the Gauss-Seidl regime, because Jacobi is using data that is *less fresh* than Gauss-Seidl. However, both these approaches have controlled, deterministic solution trajectories, and this control is a waste of compute if all we are interested in is the asymptotic solution.

Consider an alternative approach: Each grid point - there may be millions - has a compute unit associated with it. Each compute unit maintains knowledge of its own state, plus ghosts of its logical neighbours. Leaving aside the starting and stopping problems (described later), the behaviour of each unit is almost trivial. It does nothing until it receives notification (a data packet) telling it that one of its logical neighbours has changed state. On receipt of such a notification, the unit recomputes its own state. *If* the state has not changed, the unit returns to quiescence. *If* it has, the unit asynchronously broadcasts this fact to its logical neighbours (unacknowledged data-push). It is easy to see that once this process starts, a packet storm will develop quite quickly, as each unit continually re-evaluates its own state and broadcasts the change. Some packets will be delayed: the design intention is that the wallclock cost of computing a state update is small, but it cannot be zero, and it certainly cannot be relied upon to be uniform over the system. The notion of simulated time across the compute fabric cannot be defined in any meaningful way whatsoever. How can we achieve useful compute in these circumstances? Some units will be computing with 'stale' data, but we don't mind, because 'fresher' values will be along in short (wallclock) order. We have wasted a (trivial) amount of compute, but this is the price for not having to impose (and pay for) high-level data choreography. The solution *trajectory* is non-deterministic, but has no physical meaning anyway in any compute regime; only the *asymptotic* numerical solution is stable and physically meaningful. This state of affairs obviously depends on the numerical properties of the equation set; some are wildly unstable and unsuitable for this technique. However, the size of the application space for which this approach *is* useful is large and growing.

3. A prototype architecture

Event-based computing is appropriate for systems that can be decomposed into a discrete mesh, albeit one with sometimes millions of nodes. Many important engineering problems[4] fall into this category. POETS introduces a system based on linking an event-based abstract problem definition to an event-based physical compute platform. From the perspective of abstract application definition, a problem consists of an *arbitrary* graph of **devices**. A device captures the behaviour of a vertex in the discrete distributed model of the physical system (it could be a point on a wire-mesh model of a thermal system, or a single CFD point). From the perspective of abstract *compute*, the system consists of a large number (O(millions)) of extremely small, cheap compute units. These are interconnected by a fixed, fast packet-based communication infrastructure. The packets are small (64 bytes) and entirely hardware-mediated. There is no MPI-like software message layer. The arbitrary application graph is mapped onto the fixed hardware graph by initialisation software (called the Orchestrator), and thereafter device can talk to *logical* neighbour devices logically transparently via hardware. Central points of this system:

- It is computationally asynchronous: there is no central 'overseer' clock.

- The state memory is distributed throughout the physical system, and devices 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, and *in extremis* it is possible for the communication stream to be non-transitive.

By far the most significant aspect of the system 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 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 (real) time.
- Modify the packet and try again.

Although (ultimately) guaranteeing data delivery, it is easy to see how this can contribute to the data shear that can lead to non-transitivity.

3.1 The hardware platform

The underlying system platform consists of a six-layer hierarchy - see figure 2 - not dissimilar to the GPGPU stack.

At the highest level, a POETS **system** consists of a set of physical **boxes**. Each box contains a **mothership** (an X86 conventional machine) and a set of **boards**. A board hosts a DE5 development system of 6 FPGAs Every subsequent layer in the system is synthesized on the FPGA, and so can easily be modified. The FPGA contains a fixed

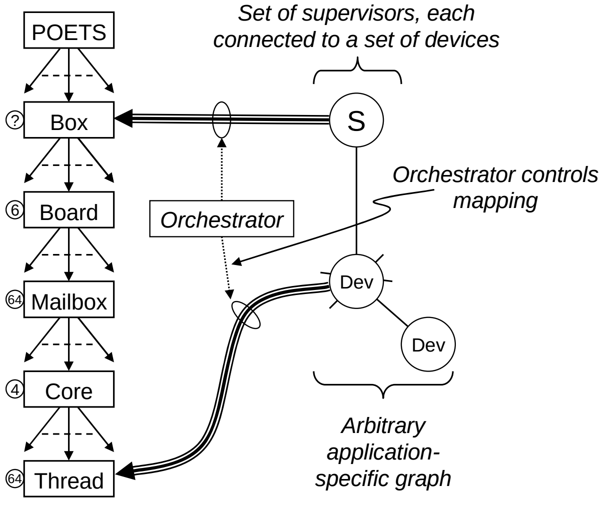
POETS
?
Box
6
Board
64
Mailbox
4
Core
64
Thread
Set of supervisors, each connected to a set of devices
S
Orchestrator
Orchestrator controls mapping
Dev
Dev
Arbitrary application-specific graph

Figure 2: The POETS hardware stack

(inasmuch as anything is fixed on an FPGA) graph of **mailboxes** and **ports**. The latter connect the cross-board mailboxes The former contains a number of **slots** (currently 4) that play host to a dynamic stream of 64 byte **packets**.
Each mailbox is connected (register mapped) to a synthesized RISC V core (250MHz), which is itself hyperthreaded. The current system (recall everything is synthesized) uses 32 bits to address the threads, limiting the maximum thread count to 4G.

3.2 The software stack

The computational problem, from the perspective of the domain-specific user, is of an *arbitrary* graph of application **devices**. The user defines the **application graph** in terms of *named* vertices (devices), each device presenting a set of *numbered* pins, and each pin may be connected to an arbitrary set of pins on other devices (and itself, if need be). The user may also define a **supervisor**. This is a kind of uber-device, the design intent of which is to oversee and facilitate command, control and data exfiltration. Figure 3 illustrates this. The important point here is that the mapping of devices to threads is decided by configuration software (the **Orchestrator**). Each mothership contains an instance of the supervisor (so the number of supervisor instances is dictated by the hardware). The mapping of supervisor instance to device subset is controlled by the Orchestrator. The supervisor behaviour must be defined by the user in the absence of hard knowledge of which device subset it will be overseeing - although the supervisor can always interrogate the device graph and find out.

3.3 Executing an application

What, then, constitutes the definition of an application graph? The application programmer defines the POETS graphs as two components: the graph topology and the device behaviour. The intent (hope?) is that the emergent behaviour of these components will produce the desired result - refer to the non-deterministic solution trajectory outlined in the previous section.
Graph topology is defined conventionally as a set of named, typed device instances with numbered (typed) pins, plus a set of pin-to-pin connections. Pins may only connect to pins of identical type.
Device behaviour is defined by a set of handlers. A hardware thread may play host to a number of (logical) devices (nominally 1024, but this figure is largely arbitrary). Multiple devices per thread represents an area of local temporal sequentialisation in the overall dataflow, so *prima facie* is to be avoided. Resident on each thread is a software skeleton (called the **softswitch**) which is effectively a spinner, interrogating the mailboxes attached to its host core and forwarding packets to the target device. (All the devices mapped to a specific thread share a hardware (32-bit) address. 1024 devices/thread gives a theoretical hard total system size of 4T devices.)
Each device contains a small state space (further subdivided into static properties and mutable state). Any incoming packets to a device are passed to the handler (invoked by the softswitch): the precise behaviour is domain-specific and user defined (the programmer embeds fragments of C into the device handler definitions), but in general the device handler - as a consequence of the incident packet - may (optionally) change the internal device state and/or emit packets of its own to its (logical) device neighbours and/or supervisor.

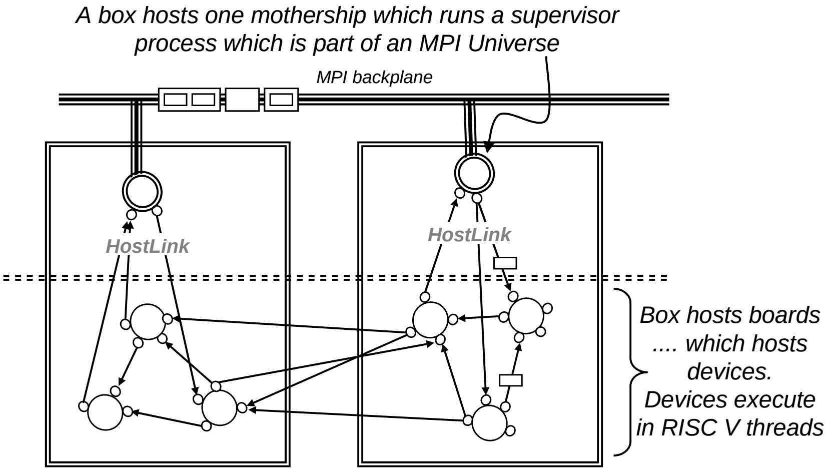
A box hosts one mothership which runs a supervisor process which is part of an MPI Universe
MPI backplane
HostLink
HostLink
Box hosts boards .... which hosts devices. Devices execute in RISC V threads

Figure 3: Supervisors and devices

Note that the user (or any external source) may inject packets into the device graph via the Orchestrator - (MPI) - supervisor path.

The Orchestrator is an asynchronous, heterogeneous MPI universe, resident on the set of motherships (plus any other processors connected to the MPI backbone). The Orchestrator controls the configuration of the system. Within its own datastructures, it contains

- A model of the available POETS hardware platform (vertex capacities, capabilities and connectivity).
- A model of the (abstract) application graph (devices, pins and types, device and supervisor behaviours).

It is responsible for

- Mapping the device graph to the thread set/graph (this single phase encapsulates the most numerically intensive functionality of the Orchestrator, and draws heavily from the world of IC placement, assignment and routing).
- Labeling the logical devices with a hardware address.
- Assembling the code fragments describing device behaviour and the device state space definitions with the softswitch skeleton, cross-compiling and linking the composite source with the low-level RISC-V library to produce the binary code (to be executed on the RISC-V threads), and downloading these binaries to the target cores.

Further details of note:

- The RISC-V has a Harvard architecture, and so the data space memory maps produced by the Orchestrator are obviously thread unique (and thus a function of the device:thread mapping), but the instruction space in each core is shared by all the threads on that core. This is not as restrictive as it might appear - in intended use, the vast majority of the devices will be of very few types, so the Orchestrator can ensure that all the devices on a core are of the same type without undue stress on the mapping penalty function. (This issue draws from the openMP GPU thread affinity problem).
- The Orchestrator part of the MPI universe is itself multi-threaded, and so can spin off the cross-compilers in a set of conventional X86 threads.

4. Performance: scaling behaviour

Two example application domains are presented here: solving the heat equation, and an example from computational chemistry.

4.1 The heat equation

The heat equation (section 2) $\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$ may be canonically discretised to give us $u_i^{(n+1)} = u_i^{(n)} + D\frac{\partial t}{\partial x_i^2}\left(u_{i-1}^{(n)} - 2u_i^{(n)} + u_{i+1}^{(n)}\right)$. A steady state solution of this equation has the temperature of each grid point with mutable state (temperature) as an average of the temperatures of its logical neighbours. (Time varying forcing heat sources necessitate the introduction of thermal *capacities* which complicate the point unnecessarily here.)

4.1.1 Knowing when to stop

Solving the equation numerically is an iterative process. In a conventional computing environment, some limit function looks to establish if the overall or average change in temperature value per iteration step has fallen below some pre-defined value; once this situation is detected, the system is deemed to have converged. In a packet-storm based system, this notion is less well defined, as individual packet latencies may vary wildly, and the time taken to notify the outside world of a putative convergence can be many times larger than an individual packet lifetime. Here we compromise:

Like the conventional approach, we ignore temperature changes below a pre-defined value, so the system eventually stops sending packets. However, the individual devices have no knowledge that this has occurred as they have no notion of time. We introduce the idea of a **heartbeat**: a software-implemented idle detection method that is fully defined by the application writer in the handlers that they provide. (We use the term "heartbeat" because there is no clock-like regularity implied.)

Heartbeats are a type of packet that is emitted frequently (see below); each device counts how many heartbeats it has received, the count being reset any time the device receives a packet from one of its logical neighbours. When this count reaches a pre-defined limit, the device emits an "end" packet to the supervisor. This packet also contains the device current temperature, fulfilling the role of data exfiltration. An end packet can be cancelled at any time prior to all the supervisors flagging finished, should a device receive any subsequent packets from its logical neighbours.

In our initial implementation, we generate heartbeats asynchronously at the thread level. Each device has a user-defined *OnIdle*" handler that may be executed by the softswitch when there is no other work to do (no packets to send or receive). We usurp the "first" device on each thread to count the number of times this softswitch handler is executed. When this reaches a pre-defined limit, a heartbeat is sent to each other device on the same thread, bypassing the mailbox. Two counters are required as an individual device has no knowledge of any packets received by other devices in the same thread.

4.1.2 Heat equation – performance

Figure 4 shows the wall-clock execution time a series of simulations of *n*-by-*n* two-dimensional heated plates on a POETS system and a single-threaded 3.8 GHz Intel i7 machine. On the POETS engine, a device calculates the temperature for a single point and convergence is detected using Heartbeats as described in 4.1.1. Devices on POETS are currently mapped to threads naïvely. Near-linear scaling is observed between 6,400 and 78,400 devices (with an anomaly at 16,900 devices). There is a discontinuity between 78,400 and 96,100 devices where the simulation fails to converge. We currently have no explanation for this. Near-linear scaling continues between 96,100 and 1,000,000 devices, albeit at a greater wallclock time.

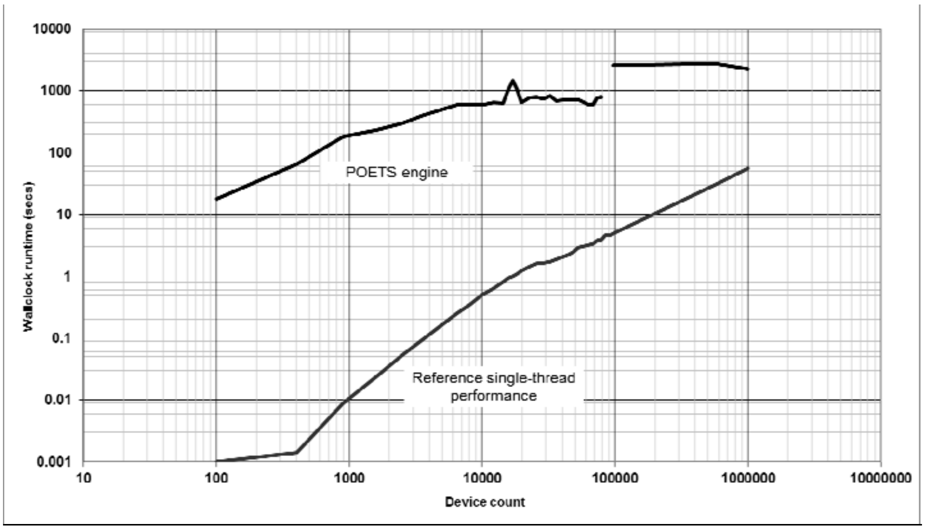

Figure 4: Heat plate simulation performance

4.2 Computational chemistry

The explosion of compute capability over the past decades has had a transformative effect on what may be achieved, and few fields have benefitted as much as computational chemistry: by solving the equations of motion of individual atoms and molecules, the demonstrated emergent behaviour is effectively that of a chemical reaction, with all the complexity that that implies. We live in interesting times: yes, we can compute the trajectories of individual atoms, and so simulate real chemical interactions, but to extract physically meaningful results requires the reaction trajectories of millions of particles followed over billions of timesteps. Even by the standards of the compute resources available today, such an undertaking is hugely expensive, and techniques are constantly being developed to make the undertaking less costly. Two strategies come together to provide a significant increase in what may be achieved in this area: Dissipative Particle Dynamics (DPD) and POETS.

4.2.1 Dissipative particle dynamics

Interesting chemistry usually (but not always) involves large organic molecules, where a carbon backbone folds in complex ways, depending on its surroundings and the ligands attached to side-chains. Usually, 'interesting' behaviour is a function of some gross stereochemical attribute of the system, not the detail: there is no point in following the behaviour of *each atom* in a -CH_3 group, because the relationship between the three hydrogens and the central carbon is unlikely to change significantly, no matter what happens to the rest of the molecule in the large. Without loss of (too much) generality, then, we can replace the four-atom subsystem with a single pseudo-particle - call it a **bead**. This idea of locally replacing relatively inflexible and internally uninteresting subgroups of atoms can be extended, sometimes cutting down the number of individual elements in a molecule by half an order of magnitude. As each individual atom in a bead itself contributes several degrees of freedom to any simulation, this represents a considerable decrease in the computational load.

The system under simulation usually consists of some number of large, complicated organic molecules, modeled by a set of beads. The beads are interconnected by Hookean and angular bonds, (representing chemical bonds), and usually immersed in some environment (water?) where each water molecule is represented by a single bead. (For reasons that are beyond the scope of this paper, systems incorporating electric charge do not analyze well in DPD). The simulation consists of integrating Newtons' equations of motion for each bead, marching forwards in discrete time steps. The forces acting on each bead at each time step are relatively simple:

- Some bead-bead repulsive force
- Some dissipative (damping) force
- Some random (thermal) force

Within 'sensible' limits, the gross behaviour of the overall system is quite insensitive to the exact numerical form of the force-fields.

4.2.2 The compute environment

Clearly this problem is amenable to parallelisation. The traditional supercomputer approach (using MPI) to this kind of simulation is to tile space with three-dimensional cuboids (wrapping round the boundaries to give a continuous periodic physical model), map each cuboid to a compute core, and to give each core responsibility for simulation of the interactions of the beads within that cuboid. Movement of beads across cuboid boundaries is handled by means of 'ghost' layers, and the simulation *rate* (the ratio of simulated time to wallclock time) is some function of the resources available to the core, the size of the system under simulation, and the number of beads per core. None of this is particularly novel, but the ideas map elegantly onto the POETS architecture, where we can easily and cheaply bring to bear many thousands of individual cores.

4.2.3 Dissipative particle dynamics - performance

Figure 5 below shows the computational cost of a simulation of two immiscible liquids. There is no termination configuration, the simulation is uninteresting and is simply allowed to run for the same number of timesteps for each point on the figure. For comparison, the sequential line is generated on a single thread, single core, 3GHz Intel i7 machine. The POETS line is generated from a small POETS system, containing 6144 threads. The wallclock cost of the simulation is (almost) flat up to 6144 devices, showing that the parallelism is (almost) perfect. The slight slope is due to the physical latency of moving packets about the system - communication costs. At 6144, the system is forced to start doubling up on the number of devices/thread - see earlier comments about serialization in the softswitch - and the runtime cost immediately doubles. Another discontinuity is visible at about 12000 devices, and thereafter the performance degenerates as network congestion starts to take its toll.

5. Final comments

These are small systems (the next system to be built is under construction - this will be an order of magnitude larger, and will move the inflections in figure 5 to the right correspondingly). Even though network congestion has an effect on the performance, in both examples, the system continues to function (section 3). However much traffic is injected into the communications fabric, the system waits locally until the network is drained by computation, and processing continues.

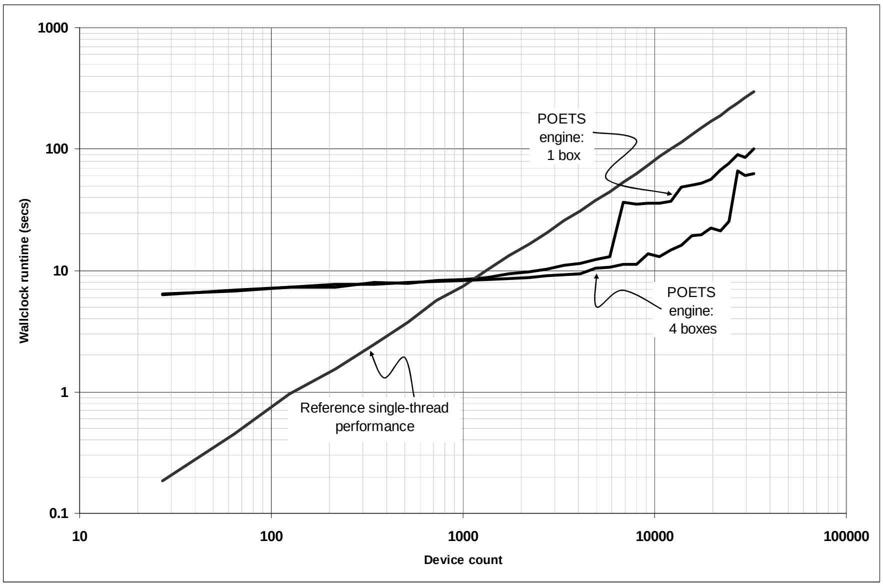

Figure 5: POETS DPD performance

References

[1] G.E. Moore, "Cramming more components onto integrated circuits", Electronics, **38** no 8, 1965
[2] R.H. Dennard, F. Gaensslen, H-N. Yu, L. Rideout, E. Bassous, A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions" IEEE Journal of Solid State Circuits. **SC-9** (5) 1974.
[3] J. Koomey et al "Implications of Historical Trends in the Electrical Efficiency of Computing", IEEE Annals of the History of Computing, **33** (3): 46-54, doi:10.1109/MAHC.2010.28
[4] A.D. Brown et al, "Distributed event-based computing", International conference on parallel computing, ParCo'17, Bologna, September 2017.
[5] S.B. Furber et al, "Overview of the SpiNNaker system architecture", IEEE T Computers, **62**, no 12, Dec 2013, pp2454-2467, doi 10.1109/TC.2012.142
[6] E. Painkras et al, "SpiNNaker: A 1W 18-core System-on-Chip for Massively-Parallel Neural Network Simulation", IEEE Journal of solid-state circuits, **48**, no 8, pp 1943-1953. doi:10.1109/JSSC.2013.2259038
[7] M. McCool, A.D. Robinson and J Reindeers, Elsevier, Structured Parallel Programming, ISBN 978-0-12-415993-8
[8] A.D. Brown et al "SpiNNaker: Neuromorphic simulation - quantitative behaviour", IEEE T Multi-Scale Computing, 4, no 3, July 2018, pp450-462, doi 10.1109/TMSCS.2017.2748122

Acknowledgements

This work was supported by the UK Engineering and Physical Sciences Research Council grant EP/N031768/1