Received: 22 November 2021 | Revised: 2 September 2022 | Accepted: 5 December 2022

DOI: 10.1049/cdt2.12051

ORIGINAL RESEARCH

Event-based high throughput computing: A series of case studies on a massively parallel softcore machine

Mark Vousden[1] | Jordan Morris[2] | Graeme McLachlan Bragg[1] | Jonathan Beaumont[3] | Ashur Rafiev[2] | Wayne Luk[3] | David Thomas[1] | Andrew Brown[1]

[1]School of Electronics and Computer Science, University of Southampton, Southampton, UK

[2]School of Engineering, Newcastle University, Newcastle Upon Tyne, UK

[3]Department of Electrical and Electronic Engineering, Imperial College London, London, UK

Correspondence

Mark Vousden, School of Electronics and Computer Science, Faculty of Engineering and Physical Sciences, University of Southampton, Highfield Campus, Mountbatten Building, 53/4044, Southampton SO171BJ, UK.
Email: m.vousden@soton.ac.uk

Funding information

EPSRC, Grant/Award Number: EP/N031768/1

Abstract

This paper introduces an event-based computing paradigm, where workers only perform computation in response to external stimuli (events). This approach is best employed on hardware with many thousands of smaller compute cores with a fast, low-latency interconnect, as opposed to traditional computers with fewer and faster cores. Event-based computing is timely because it provides an alternative to traditional big computing, which suffers from immense infrastructural and power costs. This paper presents four case study applications, where an event-based computing approach finds solutions to orders of magnitude more quickly than the equivalent traditional big compute approach, including problems in computational chemistry and condensed matter physics.

KEYWORDS

computer architecture, distributed shared memory systems, electronic engineering computing, message passing, multi-threading, network-on-chip, parallel architectures, parallel memories, parallel processing

1 | INTRODUCTION

Conventional von Neumann machines exhibit a bottleneck when fetching data: processors have become faster, while memory access speeds have stagnated. The efforts of optimising compilers aside, this bottleneck causes processors to idle, waiting for instructions and data to be retrieved. This idling inhibits conventional machines from keeping up with humanity's increasing demand to solve increasingly massive problems increasingly quickly. One consequence is a tremendous energy inefficiency, which is particularly detrimental when one considers the increasing demand for energy on a global scale. Massive engineering problems solved on conventional architectures consume millions of core-hours and watts, at a huge cost, and the problems humanity needs to be solved are only getting larger. This severely limits the potential for exascale computing: **Conventional architectures do not scale.** It is critical to address these issues, otherwise humanity will not be able to keep pace with the scale of computational problems it needs to solve.

Power dissipation is a major challenge for conventional computers in a heavily loaded computing system (the dark-silicon problem): It is not possible to power the entirety of a state-of-the-art chip at once, because it is not possible to get the electrical power in and consequent heat out fast enough without the chip failing [1]. Systems with multiple smaller processors have demonstrated greater power efficiency than equivalent systems with larger processors [2, 3]. However, the traditional argument against using a system with many smaller cores is the relative cost of communication and computation—moving data is traditionally orders of magnitude more expensive than computing it.

To combat these issues, there is an increasing demand for alternative hardware designs and architectures for tackling

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2022 The Authors. *IET Computers & Digital Techniques* published by John Wiley & Sons Ltd on behalf of The Institution of Engineering and Technology.

massive engineering problems. Field-Programmable Gate Arrays (FPGAs) are uniquely positioned to facilitate rapid exploration of the space of alternative hardware architectures. FPGAs and other programmable logic devices are commonly used to deploy a static configuration of hardware and have been used to create diverse application-specific flexible architectures. Such configurations may be adapted to include soft microprocessors (softcores) that facilitate more general computation. Clusters of softcores have been deployed in FPGAs to rapidly solve large problems that are amenable to parallelisation. These multiple softcore approaches have been shown to alleviate the power-dissipation problem [2], and improve intercore communication speed by using a parallel hardware communications fabric. In approaches like this, where both communication and computation could reasonably bottleneck, it is no longer feasible to separate the computation from the communications—the computer becomes the network.

This paper introduces event-based computing as a computing paradigm in Section 2. Section 2 also introduces event storms as a common computing pattern that occurs in event-based systems, discusses how recent developments in hardware can be exploited to efficiently perform event-based computing, and introduces an example trivial application to illustrate the concepts. **The key point of this paper is to demonstrate the efficacy of event-based computing for large, industrially important high-throughput computing problems**: Section 3 introduces a portfolio of these problems, shows how event-based computing **dramatically improves performance scaling** in those problems, and explains how it does so. Lastly, the paper concludes with discussion of the next steps for event-based computing, and how future developments might adapt it to solve further problems.

2 | EVENT-BASED COMPUTING

In the event-based computing paradigm, event consumers perform work in response to external stimuli (events). Each consumer (worker) views and operates on a subspace of the problem, conditionally modulating the local substate as a consequence and communicates in a local neighbourhood. Event-based computing describes a family of approaches across the hardware/software computing stack, which can be classified taxonomically:

- Implemented at the software level using traditional compute platforms (examples: [4, 5]): Where powerful, high-clock rate (GHz) cores use massive (≈100 Gb) memory resources to perform local computing and communicate using expensive choreographed communication stacks.
- Implemented on Graphics Processing Units (examples: [6, 7]): Where a wide Single Instruction Multiple Data (SIMD) computing supports massively-parallel homogeneous computing. While MIMD GPU computing is also possible, GPU architectures generally perform better in a SIMD context [8].
- Implemented on softcore machines (examples: Anton [9], SpiNNaker [10], POETS [2]): Where many small (hundreds of MHz) softcores are networked into a fast bespoke communications fabric. Each softcore is connected to its own memory, resulting in a distributed memory architecture. Compared with GPU processing, these softcores tend to be more flexible in the instructions they support. These architectures exist on a scale, where the application specificity is traded off against performance, and are typically realised either on Application-Specific Integrated Circuits (ASICs, fast, and low-energy) or Field-Programmable Gate Arrays (FPGAs, easier to modify, and cheaper design/deployment cycle).

This paper is not about general event-based computing, but focuses on the last entry in this taxonomy, which uses a massive quantity ($>10^4$) of compute cores with a limited instruction and data footprint relative to traditional computing platforms. Under such architectures, compute workers have no concept of the global state or synchronisation, and each worker views and operates only on a tiny problem subspace. For certain classes of problems, these workers can operate concurrently, computing with 'stale' data with no impact on the final result. In particular cases, it is more performant to send state updates to all neighbours than to explicitly choreograph data flow such that all data is up to date all the time—the receivers compute whether the state update is relevant to them in parallel on the receipt.

Event-based computing on bespoke hardware has primarily been driven by neuromorphic compute systems and brain-inspired computing, which itself has arisen from early innovations in bespoke parallel hardware [11]. Despite the availability of traditional parallel architectures, neuromorphic systems have demanded numerous smaller processing elements (ala neurons) over the few complex processing elements provided by these architectures, hence the push for bespoke hardware solutions. Execution time was the primary motivation for this decision [12], which continues to be a design driver in modern neuromorphic computing. More recent research is broad, encompassing a series of different chip design ideas. These include: the development of mixed-mode VLSI devices to model integrate-and-fire neurons in real-time [13] and neuromorphic bio-hybrid systems where hardware directly interacts with cell cultures [14]. The first ASIC-based deep learning processing architecture, DianNao, emerged in 2014 and accelerated both deep neural network and convolutional neural network inference [15]. More recent chip developments have integrated random number generation and function accelerators to speed up the computation of oft-complex brain-inspired algorithms used in neuromorphic computing [16].

Larger-scale projects have recently emerged due to advances in chip research, coupled with advances in fabrication and reconfigurable hardware. The SpiNNaker project [10] is one such project, as it has produced a high-performance massively parallel platform for the simulation of neural networks in real time. The architecture of SpiNNaker is a network

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

of ARM9 processors whose communication is brokered by neurobiology-inspired packet-switched routers. These processors host the 'behaviours' of neurons; sending 'spikes' to other associated neurons when sufficiently stimulated, and these neurons have no concept of the global state—an appropriate fit for event-based computing. The success of SpiNNaker has spawned a series of projects adapting this concept in the neuromorphic computing space. These include research projects like the direct successor, SpiNNaker 2 [10, 17], as well as industry projects like the development of Intel's Loihi 2 [18], which boasts a simulation capacity of one million neurons per chip. As with neuromorphic computing, the fields of vision and image processing have undergone a similar path with bespoke hardware solutions [19, 20].

Note that, throughout these research trajectories, the emphasis has been on architectures with many simple compute units over few complex ones—we posit that such architectures can be used to solve other similar classes of problems. This paper demonstrates that the event-based computing paradigm works well on applications beyond the targets of neuromorphic compute systems. In particular, large problems are amenable to parallelisation, but have critical communication components to them, including Dissipative Particle Dynamics (DPD) and micromagnetics, amongst others. It does not perform well for small problems, as dividing workload into many small workers itself takes time. It also does not perform well for problems that are massively parallel, but with components that do not communicate often, as the low communication requirement means the computing power of a traditional approach is unfettered. A useful way to visualise suitable applications is to think of a wire mesh model with a computing unit associated with every vertex of the mesh. The individual computing load is light, but there are many thousands/millions of them, and they have a complex, nuanced communication pattern. The 'solution' is an emergent property of the entire mesh.

2.1 | Hardware for event-based computing

Event-based computing fundamentally requires a large number (millions) of computing cores connected by a fast network. Early attempts at event-based computing architectures manifested as Massively Parallel Processor Arrays, including Ambric's Am2045 (for video processing) [21, 22] and PicoChip's picoArray (for wireless telecommunication infrastructure). More recent event-based computing hardware maintains this application-specific nature, like the neuromorphic machines SpiNNaker 2 [10, 17] and Loihi 2 [18]. Demonstrating the efficacy of event-based computing on a variety of problems, using a single hardware platform, requires a machine that is more general-purpose than these while still possessing architectural attributes that align it closely with the numerical structure of the underlying application.

While traditional desktop machines are less suited to these requirements, general-purpose graphics processing units (GPGPUs) are a natural choice. However, when problems exceed the number of computing cores available on a GPGPU, multiple GPGPUs must be networked together, which introduces undesirable networking bottlenecks. Furthermore, while GPGPUs excel when problems can be discretised into regular two-dimensional or three-dimensional grids, they generally perform less well when the application graph is irregular and is at a higher dimension (e.g. 3D finite-element analysis), particularly when nodes in the application graph have widely varying degrees (e.g. neuromorphic simulation [23]).

Field-Programmable Gate Arrays (FPGAs) provide an alternative platform on which to rapidly prototype alternative hardware architectures. The use of FPGAs gives event-based computing clusters more flexibility in the applications that can be run:

- If the application will not fit onto a single FPGA, then multiple FPGAs can be networked together more effectively than the equivalent GPGPU system [24].
- GPGPU approaches are significantly less energy-efficient than equivalent FPGA solutions [25]. The interaction between processors and off-chip memory is an energy-expensive process in most FPGA solutions, so the design of memory systems is particularly performance-critical [26].
- If an application is particularly floating-point intensive, the architecture can be adapted to accommodate that, by trading footprint between compute cores and floating-point units [2].
- The softcores can be programmed for more general computing, or streamlined to operate based ofn the needs of a specific application.

Neuromorphic compute solutions are commonly based around Application-Specific Integrated Circuits (ASICs), which are fundamentally integrated circuits (ICs) designed with a particular end use in mind [27], as opposed to ICs like memory systems. ASICs, when compared with FPGA solutions have a smaller physical size and power consumption [25], but greater turnaround times and economic design and manufacture costs at smaller scales [28]. The rapid rate of technology evolution makes it difficult to pin down these tradeoffs into hard numbers, but recent studies in a neuromorphic computing context find a factor 2.8–6.3 performance improvement, and an 8.7 area improvement in favour of ASICs [29]. FPGAs are naturally more attractive in research contexts, where rapid prototyping is important and economically-constrained. Smaller-scale deployments may be tolerated, whereas ASICs are favoured in large-scale deployments and where power dissipation is a crucial issue.

Lastly, it is worth noting that hybrid ASIC-FPGA solutions explore tradeoffs across this spectrum. By embedding an FPGA into an ASIC directly, communication between the two can be greatly improved compared to having them on separate cards [28]. The flexibility of the FPGA contributes to the design process, as it allows system designers to define more 'frozen' areas of the architecture (ASIC), with the ability to adapt more flexible FPGA components. Economic considerations (e.g. semiconductor processing, assembly of multiple

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

separate chips, and power dissipation), while nuanced in its own right, can be traded off in a hybrid design [25, 28].

The studies in this paper use the Partially-Ordered Event-Triggered Systems (POETS) platform [2], which consists of a network-optimised FPGA cluster containing thousands of RISC-V cores and is capable of solving different types of problems in an event-based manner. The development of POETS is ongoing; custom enhancements to these cores, with a view to improving performance on certain applications (inspired by research in other fields [16]), are being explored [30]. We emphasise that the development of event-based computing hardware is not the key point of this paper - our focus is on demonstrating its potential for solving a variety of pertinent big computing problems.

2.2 | How does it work?

Event-based computing is different from other message-based systems. It is ideally suited and intended for simulation problems based on large graphs (**application graphs**), where small, independent tasks (**devices**) are connected by edges carrying small, atomic, and asynchronous messages (**events**)—two small examples of this are shown in Figure 1. Problems like this include neural simulations and state changes in finite element methods—other examples are discussed later. The application graph may represent the physical topology of the system under simulation (in which case the topology is arbitrary, like an electronic circuit), but this is not always the case. In this representation, a device operates on a particularly small, non-overlapping region of that space. Compute workers (hardware) typically host multiple devices (software).

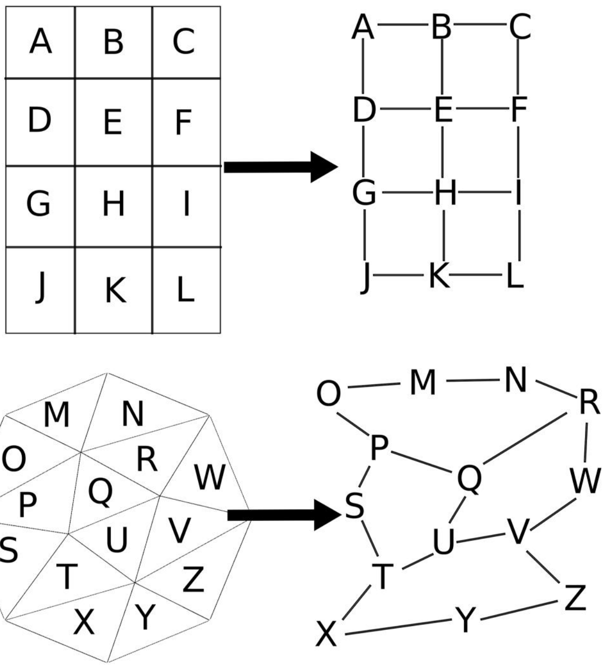

FIGURE 1 Two application graphs, showing how different problem domains can be represented as application graphs. Event-based approaches can be applied to regular and irregular problem geometries. Top: A regular two-dimensional geometry discretised into devices 'A' to 'L' and represented as an application graph with connecting edges, along which events travel. Bottom: An irregular discretisation represented as another application graph. A planar example is shown here; planarity is not required in the general case.

For example, consider a two-dimensional space tiled with identical squares, where each square can interact with each of the four neighbours in its immediate vicinity. Each square could be a device in the application graph, which in this case is highly regular (Figure 1 top), but this is not necessary in the general case (Figure 1 bottom). The event-based computing paradigm is not constrained by dimensionality. The arbitrary topology is problem specific, and application graphs are mapped onto the fixed hardware platform by initialisation software and routing infrastructure.

Each device maintains a small state independent of its neighbours, and the behaviour of each device is defined by a set of small sections of an executable code (**handlers**), which react to events. On receipt of an event, the device executes the appropriate handler. During the course of this execution, the device may modify its own state and/or emit events of its own. Thereafter it returns to quiescence, awaiting the arrival of another event. Consequently, the entire computing trajectory of this event-based computing paradigm is **asynchronous**—no communication choreography is imposed on the problem, and the solution is an **emergent property** of the system, once all devices return to quiescence. This is unlike the traditional big computing systems, which require human-designed choreography (computers are deterministic), and it may not be necessary in some problems (see the Heat Equation example below).

The operation of an event-based computing system may be described given the following definitions:

```
define Id: Set # Used to identify a single
device
define DeviceState: Set
define Devices: Map(Id -> (DeviceState, Set
(Pin)))
define Edge: Set(Id, Pin, Id, Pin)
define Events: Set
routine OnRecv: DeviceState, Pin, Event ->
DeviceState
routine CalcRTS: DeviceState -> Bool
routine OnSend: DeviceState -> DeviceState,
Pin, Event
```

Each device is a member of the devices set; it has an Id, a DeviceState, and a set of pins connected to it, where a pin either sends events to other devices along an edge (acting as an output), or receives events along an edge (acting as an input). The three routines define behaviours executed by application-level handlers:

OnRecv: When an event is received on a pin, the handler transforms the state of the device; the handler defines how the state is transformed.

CalcRTS: The handler determines whether the device should send (and to whom) based on the state of the device.

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

OnSend: Sends an event to a device using a pin. The handler defines the content of the event and the pin to send on from the device state. The handler might also modify this state.

A model of the interaction of devices in an event-based system given these definitions may be described thus:

Colloquially, OnSend is called when an event is received by a device on a pin—this may change the state of the device. That device may wish to send data: This decision can be computed by its state. If the device does wish to send data, OnRecv is called, sending events along each edge connected to

```
define InFlight: Multiset(Id, Pin, Event) = {}
define ReadyToSend: Set(Id) = {}
define Computing: Set(Id) = {}
define Action: Set = {"Send", "Recv", "Finish"}
define UpdateCandidates: Set(Action, Id, Pin, Event) = {}
while |InFlight| > 0 or |ReadyToSend| > 0 or |Busy| > 0:
    # Determine which devices may be updated:
    UpdateCandidates = {}
    for Id in ReadyToSend where Id not in Busy:
        UpdateCandidates.insert("Send", Id, NULL, NULL)
    end for
    for (Id, Pin, Event) in InFlight where Id not in Busy:
        UpdateCandidates.insert("Recv", Id, Pin, Event)
    end for
    for Id in Busy:
        UpdateCandidates.insert("Finish", Id)
    end for

    # Choose one candidate action to take
    transition = stochastic_choice(UpdateCandidates)

    # Perform that action
    match transition with:
        ("Send", Id):
            DeviceState, Pin, Event = OnSend(Devices[Id])
            Devices[Id] = DeviceState
            for (Source, SPin, Destination, DPin) in Edges \
            where Source == Id and SPin == Pin:
                InFlight.insert(Destination, DPin, Event)
            Busy.insert(Id)
            ReadyToSend.remove(Id)
        ("Recv", Id, Pin, Event):
            DeviceState = OnRecv(Devices[Id], Pin, Event)
            Devices[Id] = DeviceState
            Busy.insert(Id)
        ("Finish", Id):
            Busy.remove(Id)
    end match

    if CalcRTS(Devices[Id]):
        ReadyToSend.insert(Id)
    else:
        ReadyToSend.remove(Id)
    end if
end while
```

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the output pin. Once the event arrives at its destination, OnSend is called by the recipient. On the sending device, the OnRecv logic may further modify its state, which may result in further events being sent to other devices. Using this framework, it is possible to describe a variety of communication patterns and behaviours to define the logic for different applications.

The algorithm presented above is a serialised model of the behaviour of our event-based computing approach. Critically, in the real system:

- The behaviour of each device is asynchronous with every other device, and
- Events must traverse the hardware network, from the sending device to the receiving device. Consequently, they take time to arrive at their destination, which is subject to network conjestion and receiver behaviour.

It is also worth emphasising that the stochastic_choice is determined by the hardware. Section 2.4 describes how this approach is used to find the temperature distribution in a plate, and Section 3 describes a series of applications implemented using this paradigm.

2.3 | A storm of events

Events triggering handlers, which in turn emit more events, can quickly lead to an **event storm**—a bounded exponential growth of events. Counter-intuitively, this storm is not dangerous, as most event-based computing architectures contain a throttling mechanism to prevent the network between workers from being overloaded. Consequently, the system goes as fast as the physical network permits, all the time, as a function of local congestion.

To introduce this, consider a problem where space is discretised into devices such as in Figure 1 (top), and each device is initialised with the same scalar state value. A small change is introduced to each corner device of the space ('A', 'C', 'J', and 'L': four devices in all), slightly altering their states. In this example problem, these perturbed devices send their updated state to their neighbour, each of which then adjust its own state to the local average. Those devices then send their new state to their neighbours, if it has changed significantly, beyond a small tolerance value that is specific to the application. All devices act asynchronously and simultaneously, and each event may spawn more events when it is received by a device. Eventually, all devices hold the same, averaged state value.

Figure 2 illustrates the transient behaviour of an event storm—it shows the rate of outgoing events, for each device, for the aforementioned problem on a 512 by 512 grid. A storm of events emerges abruptly because each device acts as fast as it can, when it is notified of a state change from one of its neighbours. As abruptly as it starts, the grid of state values discovers, on the basis of local information alone, that it is in equilibrium, so all event traffic ceases (every incoming event causes no state change and therefore no consequent emission of events). This sudden stop is shown by the sudden 'silence' of traffic at the end of Figure 2.

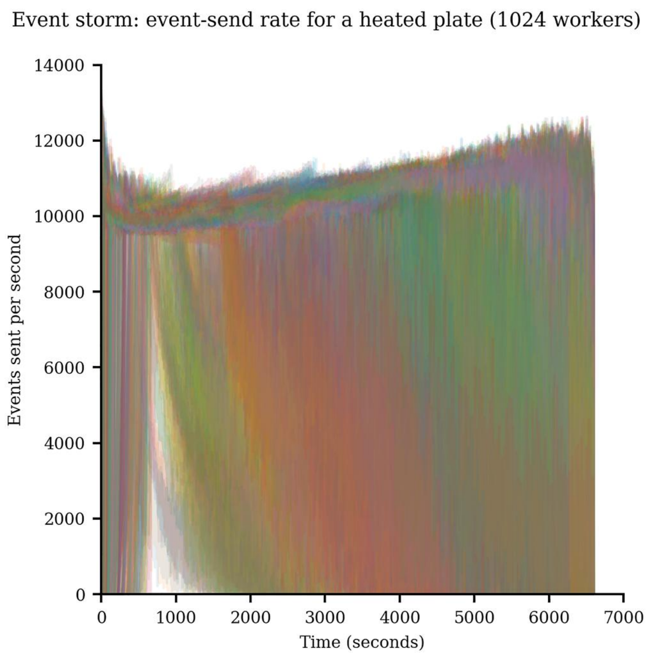

FIGURE 2 Network activity during an event storm, caused by the application described in the 'Storm of Events' Section. Data available at reference [31]. Shows the event send rate for all 1024 workers as a function of time—each translucent line corresponds to one worker. After the brief compute-bound startup, a storm of events emerges as devices share their state with their neighbours. All workers are active at the start of the problem, as the packet storm develops and taxes the inter worker network (0–2000 s). As the system tends to equilibrium (2000–6250 s), the local event traffic becomes more irregular. The system converges suddenly, shown by the sudden 'silence' of activity at the end of the computation.

As with any application on any compute system, there is always a performance bottleneck. An event storm occurs when events are, on the whole, created and sent faster than they are processed. In this case, event-based architectures typically throttle the network in such a way that events are not dropped, though some domain-specific event-based architectures permit dropping of events, and have mechanisms for recovering dropped events [3, 10]. In an application that is more compute-intensive, an event storm does not emerge because some aspect of the computing system, for example, the floating point unit or the memory subsystem, is the bottleneck.

2.4 | Heat equation example

Consider the problem of using the heat equation to find the temperature distribution in a plate, where symbols have their usual meanings:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right). \qquad (1)$$

Applying a central differencing scheme to discretised space and an Euler integration method to this equation yields:

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

$$u_{i,j}^{m+1} \approx u_{i,j}^{m} + \frac{\alpha\,\delta\,t}{\delta x\,\delta\,y}\left(u_{i+1,j}^{m} + u_{i-1,j}^{m} + u_{i,j+1}^{m} + u_{i,j-1}^{m} - 4u_{i,j}^{m}\right), \tag{2}$$

where i, j, and m denote discretisation in the x, y, and t dimensions, respectively.

In a traditional computing approach, one can solve this using Gauss–Seidl, Jacobi, or other iterative linear algebra solution methods (shown in Figure 3, left). Whichever we choose, the software must choose an order to update elements in (with an associated cost), ideally in a manner sympathetic to the architecture. In the problem we have constructed, however, the order of updates is entirely irrelevant, as **different numerical dynamics always converge to the same result—the temperature across the plate will be constant**.

Now consider an event-based approach that creates an event storm (Figure 3, right). Each of these discrete points (i, j) is mapped onto its own thread. Since updates are performed *asynchronously* (because order does not matter here), there are fewer synchronisation overheads. Fundamentally, the software engineer needs to neither know nor care about updating the ordering. When the event storm described in the previous section ends, each thread (i, j) holds the physically realistic solution at the point of concern.

An event-based computing approach to solve this problem would first break up the plate into a wire mesh. This mesh is the **application graph**, where the nodes (**devices**) represent discrete sections of the problem, and the edges represent the communication between those nodes. In the general case, the mesh is arbitrary—the degree is low for spatial discretisation problems like this one, but is high for problems like neuromorphic simulation. Using the model introduced in Section 2.2, devices implemented in the following way would solve the heat equation problem:

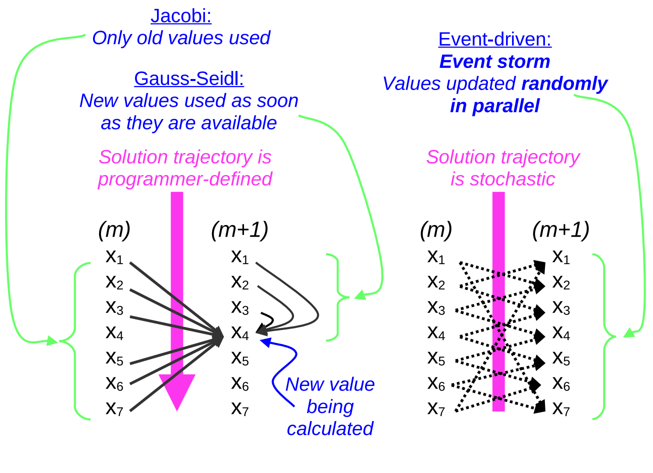
Jacobi:
Only old values used
Gauss-Seidl:
New values used as soon as they are available
Solution trajectory is programmer-defined
(m)
(m+1)
x1
x2
x3
x4
x5
x6
x7
New value being calculated
Event-driven:
Event storm
Values updated randomly in parallel
Solution trajectory is stochastic

FIGURE 3 Three different methods that can be used to iteratively solve the heat equation. Left: Common traditional computing methods where the solution trajectory is defined by the programmer—Jacobi (only values from m are used to find values for m+1) and Gauss–Seidl methods (values from m or m+1 can be used to find values for m+1, depending on what is available). Right: An event storm, where the stochastic solution trajectory is defined to go as fast as the hardware can support.

Device State: Each device stores its local temperature, as well as the temperature of each of its neighbouring devices. It would also store a boolean, update, showing whether the last update was significant with respect to some threshold.

Events: Events would each contain the temperature of the device that sends it.

OnRecv: The temperature of the message is stored in the device state under the appropriate neighbour. Then, the local temperature is updated based on an average of the temperatures of its neighbours. If the change from this update was significant, set update to true.

CalcRTS: Just returns the value of update from the device state.

OnSend: Sends an event with the current temperature to all neighbours, and sets the update to false (since no further event needs to be sent, until more temperature data is passed to the device).

Given the code descriptions of these behaviours, an initial state and a mapping of the application graph to the hardware compute as described in Section 2.5, these descriptions are compiled into a set of binaries to be deployed onto the hardware and are executed to solve the defined problem. Further information about event-based computing and the POETS approach can be found at https://poets-project.org.

2.5 | Mapping an application to hardware

The event-based computing hardware, described in Section 2.1, distributes compute workers (e.g. cores) over a network. Such a network can be modelled as a **hardware graph**: a graph $G_H(N_H, E_H)$, where nodes (N_H) are compute workers connected by edges (E_H). Application graphs can also be modelled similarly: G_A (N_A, E_A), where nodes (N_A) are devices, connected by weighted edges (E_A), which carry events.

As compute workers typically host multiple devices, one key challenge in event-based computing is to map the devices N_A onto workers N_H efficiently. A balance must be reached between loading the compute workers with as few devices as possible (balancing the quantity of N_A mapped to each N_H), while also placing devices as close together as possible to minimise communication delay (having as few E_A overlapping E_H as possible). The former encourages devices to be spread apart over the hardware graph, whereas the latter encourages clustering.

Figure 4 shows a mapping of an example of the application graph with 16 devices onto a hardware graph with four compute workers. The mapping balances the load of each worker while reducing the number of edges in the application graph that travel between hardware nodes. In the general case, where application graphs can be highly irregular, effective solutions require numerical optimisation techniques. Simulated annealing [32, 33], an iterative optimisation method, is commonly used to obtain effective solutions to modern placement problems [34–36], though graph partitioning approaches are also effective.

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

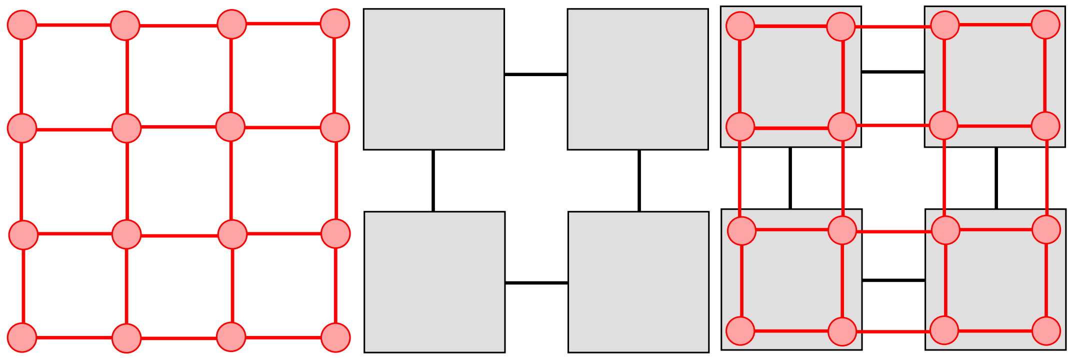

FIGURE 4 A regular 4 × 4 device graph (left), a regular 2 × 2 hardware graph (middle), and one possible mapping of this device graph onto this hardware graph (right). In this mapping, each hardware node is loaded with four devices, and eight edges in the application graph will send events over edges in the hardware graph. Application and hardware graphs may be irregular in practice.

2.6 | Developing applications for event-based hardware

To create an efficient application to run on event-based hardware, it is not sufficient to port the code of a traditional computing application. Much of the design behind event-based applications follow common principles in distributed and asynchronous computing [37], though each architecture has its own approach for handling events, typically defined by the needs of the applications that are to be executed upon it.

Data choreography patterns, other than the event storm shown in Section 2.3, exist in event-based computing. Some applications benefit from (very infrequent) communication with a global overseer process to collect system telemetry. Globally-Asynchronous, Locally-Synchronous (GALS) approaches are sometimes employed when problems are stiff [38], such as the DPD example presented in Section 3. With reference to the model presented in Section 2.2, GALS synchronisation can be implemented by augmenting the device state with the following fields:

`value_present` The 'value' of the device (e.g. temperature in the heat equation example in Section 2.4).

`t_present` The 'time step' corresponding to `value`.

`value_present_adj[]` The values of all neighbouring devices at `t_present`, if known.

`value_future_adj[]` The values of all neighbouring devices at the 'next time step', if known.

where 'time' is a discrete construct used to step an iterative process forward. Handlers can be defined as follows (events now hold a value and a time step):

OnRecv: Store the incoming neighbour value in either `value_present_adj[]` if the incoming time step equals `t_present`, or in `value_future_adj[]` if it is one greater. In this scheme, it is not possible for the incoming time value to be any other value.

CalcRTS: If `value_present_adj[]` is fully populated, this device will send to its neighbours (and update its own state).

OnSend: In order:

1. Compute a new value for `value_present` at the next time step, using neighbour values in `value_present_adj[]`.
2. Update `t_present` to the next time step.
3. Replace `value_present_adj[]` with `value_future_adj[]`, and clear `value_future_adj[]`.
4. Send one event to all neighbours with `value_present` and `t_present`.
5. Run **CalcRTS** again.

This approach increases the per-device memory footprint, but prevents neighbouring devices from drifting apart during execution.

3 | EXEMPLAR APPLICATIONS

Machine architecture design efforts are typically focussed on pre-emptively identifying potential bottlenecks and accelerating or avoiding them. This section presents four examples where bottlenecks constrain performance on the traditional big computing systems are to be linear with respect to problem size, but where different event-based computing approaches result in superior and sub-linear scaling behaviour. Each example presents a graph, which compares the measured application performance against problem size for both event-based and traditional computing approaches.

3.1 | Dissipative particle dynamics

Dissipative Particle Dynamics (DPD) is a particle-based Newtonian march-in-time simulation scheme that was developed to allow simulations of complex fluids on near-macroscopic length scales, while retaining near-molecular fidelity [39]. In DPD, particles do not represent atoms, but groups of atoms or molecules—a particle might consist of several water molecules or several atomic groups within a larger molecule. Compared with molecular dynamics, this

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

choice sacrifices some atomistic details, for example, H-bonds of molecules and ligands, but significantly reduces compute load. DPD is best suited to exploring the properties and dynamics of complex fluids and soft matter on length scales much larger than the atomic, for example, supramolecular aggregates such as phospholipid membranes and vesicles, polymeric mixtures, and soft surfaces. On these scales, the fine details of intermolecular interactions are hidden within material constants and fluid viscosity. While short-range interactions are of critical importance, the forces between particles must be soft (capped as the distance between particles tends to zero), which increases the size and time scales of systems that can be simulated using reasonable computational resources. DPD simulations are most commonly performed in a simulation box with a constant volume, constant number of particles, and constant temperature. The simulation box is subject to periodic boundary conditions to avoid artefacts due to hard walls; a particle that moves beyond any boundary of the simulation box instantly re-appears at the opposite face.

DPD is commonly used to simulate dynamic chemical and biological systems whose interesting behaviour emerges on micrometer length scales and over timescales of hundreds of microseconds. This requires following the evolution of tens of millions of particles for millions of integration steps. This leads naturally to the use of a parallel version of DPD and the communication cost involved in calculating the inter-particle interactions constitutes the main bottleneck in applying DPD. Figure 5 shows how event-based and traditional computing approaches scale with bead count (and consequently box size) in an oil-water mixing problem. At 2^{17} beads, the event-based approach takes two orders of magnitude less wall clock time to achieve relaxation. Problems larger than this require that multiple devices are mapped onto compute workers, because the number of devices in the problem is greater than the number of cores. Between 2^{17} and 2^{20} beads, each core hosts either one or two devices. Another step exists just after 2^{20}, where some cores have to host three devices in order for the application to fit onto the hardware.

The time-to-relaxation data for the traditional approach, presented in Figure 5, was obtained from the Open Source Polymer Research Engine-Dissipative Particle Dynamics (OSPREY-DPD) simulation engine [40] on the traditional high-performance computing hardware. The event-based implementation is described in Ref. [41], without angle bonds. The three-dimensional application graph is mapped onto the two-dimensional hardware graph using a graph partitioning approach [42].

3.2 | Petri net simulation

The standard form of Petri nets (bipartite graphs containing places and transitions, annotated with tokens and directed arcs, respectively) is principally used for modelling concurrency and choice in distributed systems. Most Petri net models are descriptions of systems/communication mechanisms condensed

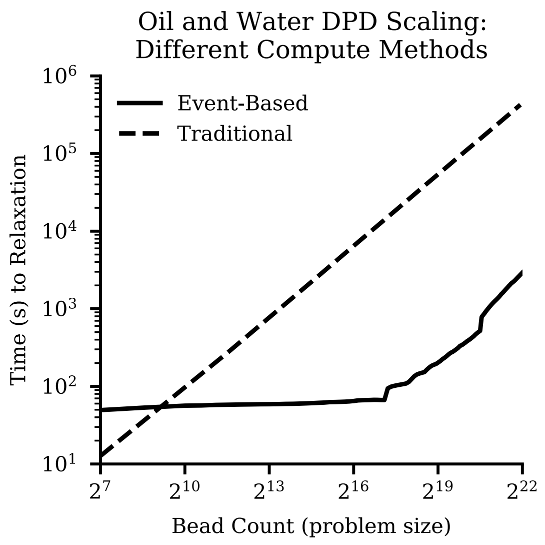

FIGURE 5 Run-time versus problem size for event-based (solid) and traditional computing (dashed) dissipative-particle dynamics implementations of oil-water mixing.

into a small number of places and transitions. Static dataflow structures are one form of the Petri net model that have gained popularity within the field [43]; these models with behavioural equivalence of registers, combinational logic and communications channels. Once generated, simulation and verification of logic are considerably simplified due to the relatively simple nature of the Petri net formalism. Figure 6 shows a fragment of two Muller pipelines, with a synchroniser. This design is an eloquent benchmark for an activity, as exactly half of the pipeline resources are occupied and half vacant, leading to half of the transitions being active at the initial marking and a large amount of continual activity during simulation—a significant simulation load.

Petri nets with maximum firing execution policies show significant simulation speed improvements; specifically, simulation speeds of standard Petri nets with single firing execution policies dramatically improved in applications requiring maintenance of local causality only. Figure 7 shows how event-based and traditional compute approaches scale with the number of places for the Muller pipeline problem. Note the scaling behaviour is similar to the DPD example, but the problem does not serialise for the problem sizes considered here. The order-of-magnitude runtime improvement with event-based computing allows for much faster design iteration.

Both the traditional approach and event-based version run virtually identical C code to update the state. The code is written in a runtime environment/Petri net model breakdown. Thus, the code is essentially a runtime environment that knows the execution policies of Petri net models, and the actual Petri net is then provided as a model instance (the Petri net itself is not hardcoded and can be easily swapped out for a different model). The ‘traditional’ approach uses one worker, which handles all places and transitions in the Petri net model. In the event-based computing approach, the runtime environment is

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

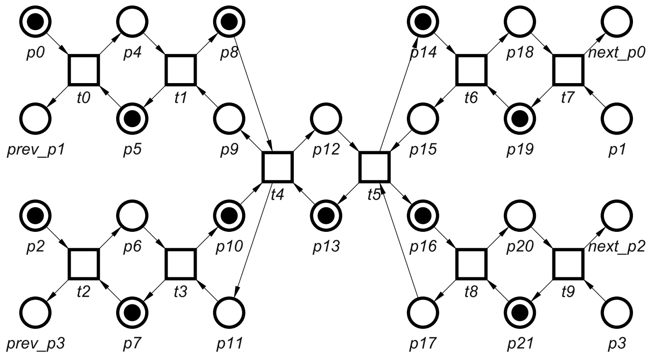

FIGURE 6 A fragment of two Muller pipeline Petri Nets with a synchroniser.

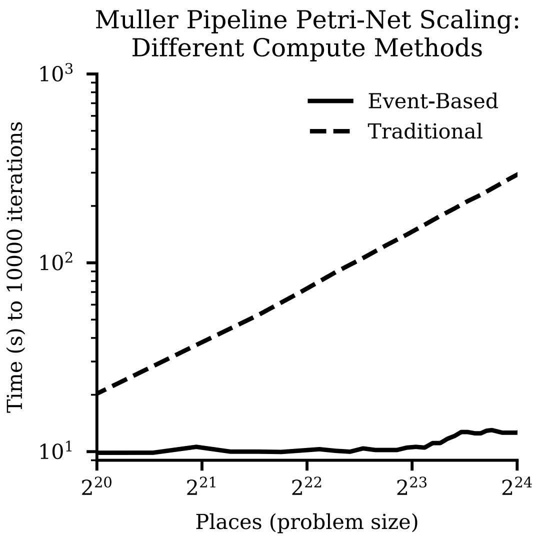

FIGURE 7 Run-time versus problem size for event-based (solid line) and traditional computing (dashed line) implementations of Muller pipeline Petri net problems.

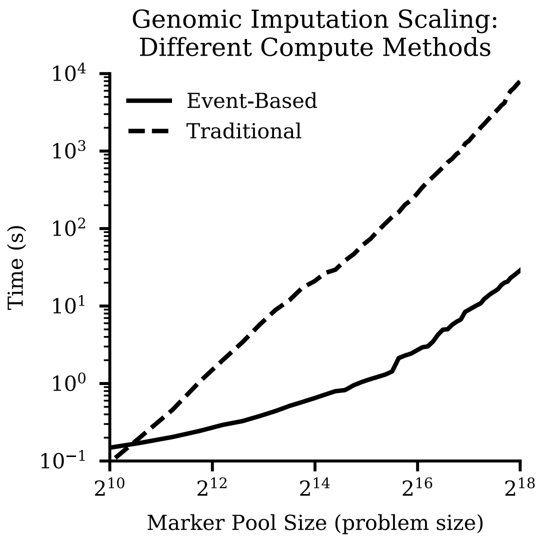

FIGURE 8 Run-time versus problem size for event-based (solid-line) and traditional computing (dashed-line) implementations for a Genomic Imputation problem.

written to all devices, but the Petri net model is discretised such that each device governs a small section of the Petri net. Events are used to transport tokens (used by transitions) from one subsection (held by a device) of the Petri net to another (held by another device). GALS synchronisation, as described in Section 2.6, is employed here to prevent tokens from arriving out of sequence. The one-dimensional application graph is mapped onto the two-dimensional hardware graph in such a way that places adjacent devices together in the hardware graph.

3.3 | Genomic imputation

The first and perhaps most important stage in understanding heritable medical conditions is the process of identifying the underlying causal genes. Whilst DNA sequencing was the first technology used to map the human genome at the turn of the millennium [44, 45], whole genome sequencing remains prohibitively expensive for experiments requiring samples from many participants. As such, a secondary technology was developed called Genotyping-By-Chip. This technology samples markers distributed across the entire genome and critically, is an order of magnitude cheaper than the next generation sequencing.

Genotyping-By-Chip has enabled experiments such as Genome Wide Association Studies [46], a methodology to genetically sample populations as a whole. By separating samples based on observable traits, genetic loci with significant statistical correlation to those traits are obtained. The accuracy of the results is inherently tied to the number of haplotypes used in the study, H (each participant provides two haplotypes - a group of alleles - to pass on), and the number of markers in the technology, M. This has driven exponential trends in both values, with studies a decade ago consisting of 10^3 haplotypes and 2.1×10^6 markers [47]. Current studies consist of 2.4×10^8 M [48] markers and within the next decade, studies are expected to contain 10^6 haplotypes. A challenge for Genome Wide Association Studies is that they are rapidly superseded by newer studies with higher participation on better technology. Moreover, two major companies provide the technology and the markers locations chosen by each are proprietary.

To address these issues, a technique known as Genomic Imputation was developed to statistically infer new markers from samples taken using older technology [49]. This improves the relevance of study results and allows for meta-analyses between studies. The method uses a customised Hidden Markov Modelling typically implemented as a forward/backward dynamic programming algorithm. This scales as $\mathcal{O}(H^2M)$ in traditional computing, presenting a significant challenge to the field due to the exponential trends in H and M. Figure 8 shows how event-based and traditional computing approaches scale with marker pool size. By considerably improving the runtime of large genomic

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

imputation problems, as event-based computing has done here with a three-orders-of-magnitude decrease, greatly facilitates the incorporation of earlier studies and 'future-proofing' of today's research.

As with the Petri net example, both approaches run virtually identical C code to update the state. In the event-based computing approach, each cell in the reference table is mapped to a single device, which performs the forward/backward dynamic programming algorithm ([50]) to calculate the cell value. The device then broadcasts its value to all cells in the next column. In the 'traditional' implementation, cells are calculated one after another via iteration. As the application graph is a two-dimensional array of cells, it maps directly onto the regular two-dimensional hardware graph.

3.4 | Magnetic simulation

To store data in computers, data systems magnetise regions of material either 'up' or 'down', into small (50 nm) magnetised domains. One alternative to these magnetic domains are magnetic skyrmions [51], which are particle-like structures in the magnetic moment field of a helimagnetic material. Since skyrmions are of the order of nanometres in size (<10 nm, potentially atomic-spacing) [52], they are a promising replacement for magnetic domains to increase storage density. The process of flipping a skyrmion from 'up' to 'down' takes orders of magnitude less energy than with magnetic domains, resulting in a significant energy saving.

Skyrmions and other small particle-like structures like magnons also motivate the design of small logic-gate devices [53]. However, these structures are only stable in limited regions of the parameter space, defined by the strength of an external magnetic field and the properties of the material.

A micromagnetic model is one approach for analysing skyrmions in a magnetic field [54]: The model discretises space (using a finite-element scheme) to solve problems on different materials and geometries. A numerical approach is fundamental in solving micromagnetic problems of value, as race-track and logic systems both have a complicated geometry and an irregular field distribution over that geometry. Presenting computing technology imposes a practical restriction on the size of these models, as problems of interest, most of which are large, often take weeks to solve on conventional computers. Figure 9 shows that event-based computing significantly reduces the execution time of these models, enabling rapid development of spintronic devices, eventually enabling next-generation exascale computing systems.

The time-to-relaxation data for the traditional approach, presented in Figure 9, was obtained from Fidimag: a finite difference atomistic and micromagnetic simulation engine [55] on the traditional high-performance computing hardware. The event-based implementation uses GALS synchronisation, as described in Section 2.6, as the governing differential equations are stiff. As with the genomic imputation example, the two-dimensional application graph maps directly onto the two-dimensional hardware graph.

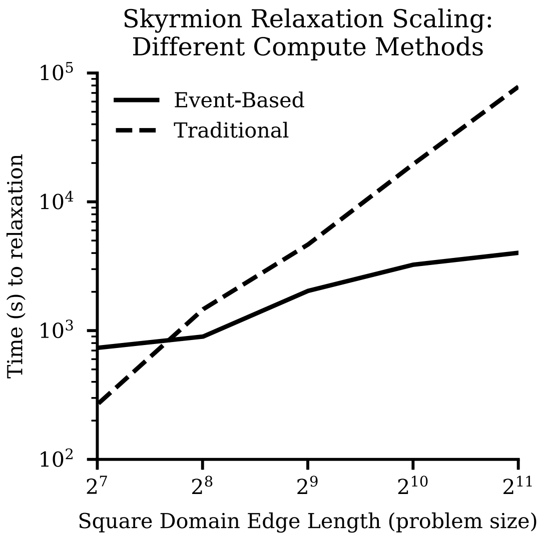

FIGURE 9 Run-time versus problem size for event-based (solid-line) and traditional compute (dashed-line) implementations of a skyrmion relaxation micromagnetics problem.

3.5 | Comments on scaling behaviour

Each of the Figures 5 and 7–9 showcase an application where execution time scales linearly with problem size using traditional computing methods (the dashed line). The scaling of event-based approaches shown by those figures (solid line) starts sub-linear and linearises as the hardware is overloaded. Figure 5 in particular demonstrates this—the scaling is flat up to 2^{17} beads, but linearises as the problem size increases because each compute worker holds multiple devices. The hardware of the event-based system is best improved by increasing its size, effectively increasing the problem size at which the linear scaling occurs. Unlike with traditional computing (where expensive advancements in chip design and thermal regulation are required), event-based computing solutions simply need 'more of the same' hardware in order to scale, allowing for incremental improvement without further architectural exploration.

In each of the four applications above, the event-based solution demonstrates orders-of-magnitude reduction in execution times—this is incredibly significant for addressing the power and execution issues that inhibit exascale computing efforts. Event-based computing is not about shaving milliseconds off second-long problems; it is about orders-of-magnitude improvements in week or month-long runtimes.

3.5.1 | Comments on traditional parallel approaches

Figure 10 shows the time-to-relaxation data presented in Figure 9, Section 3.4, with up to eight compute workers operating in parallel on a traditional computing platform. As the number of compute workers doubles, the time to relaxation halves. The execution times of these traditional approaches can

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

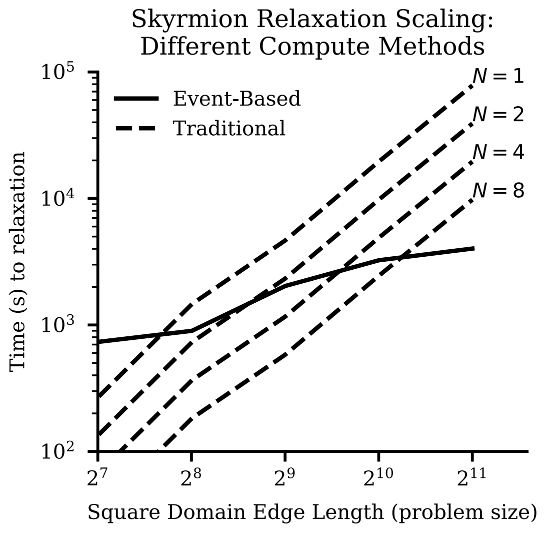

FIGURE 10 As with Figure 9: Run-time versus problem size for event-based (solid-line) and traditional compute (dashed-line) implementations of a skyrmion relaxation micromagnetics problem. This figure is extended to show scaling behaviour from traditional parallel approaches, for N compute cores.

be further decreased by using faster processors with more parallel workers, up to a saturation point. **However, unlike with the event-based computing approach which scales sub-linearly, the execution time of this traditional parallel approach continues to scale at best, linearly with the size of the problem.** If the problem is large enough, the time to solution remains a major issue for traditional parallel approaches.

4 | NEXT STEPS

The exemplar applications exhibit properties that make them amenable to event-based computing—they all decompose the problem domain into a large mesh of devices with simple handler logic driving them and do meaningful work by allowing these devices to rapidly communicate asynchronously. Event-based computing is less suitable for problems with narrow and deep datapaths graphs, for example, problems requiring significant serial computation. The space of problems amenable to event-based computing is broader than the DPD, Petri net simulation, genomic imputation, and magnetic simulation applications we have introduced in this paper: More **exploration** will broaden the portfolio of event-based computing applications. Doing this will provide a more precise set of characteristics, to allow computer engineers and software engineers to more accurately predict whether or not event-based computing is of value in their scenario. In particular, further exploration of systems with irregular graphs like finite element models will considerably broaden the event-based computing application space.

Many tried-and-failed innovations in computer architecture demonstrate great promise, but they fall into obscurity as research focusses solely on exploring potential applications. Hence, one of the key next steps for event-based computing is **exploitation**—taking an application domain known to be amenable to event-based computing and use it to generate meaningful results and to conduct meaningful scientific investigations on problems too big or too slow to explore using traditional methods. It is important to note that one can no longer develop software, port it to a set of architectures, and expect efficient results. Software design at the most intimate level, now more so than ever, is a function of the computing paradigm on which the target hardware architectures operate, so exploitation of event-based computing is a non-trivial endeavour.

Much like the development trajectory behind GPGPUs, event-based computing can be further generalised by employing ideas from heterogeneous computing. Combining event-based computing architectures with conventional compute architectures results in a platform that can solve more general problems than event-based architectures alone, at the cost of increased implementation complexity.

4.1 | Closing thoughts

An ever-growing number of research areas demand more computational resources. We have provided a miniscule slice of what event-based computing can be applied to in the Exemplar Applications section; there is great potential for event-based artifical intelligence solutions and digital twins for complex engineering systems—both of these topics have recently gained considerable traction and are certainly not the only topics to do so. Modelling the passage of a virus through a cell wall is an immense computation challenge, and is currently well of reach for conventional architectures, but event-based computing technology brings solutions within the reach of further study. Computational demands for research are greater than ever, so computer-architectural and compute-paradigm solutions must be transformative enough to overcome these demands.

Event-based computing is transforming computational approaches for solving certain classes of problems, but it is important to note that this paradigm is not a general-purpose compute approach. There are many problem domains for which it is not well suited. For those it is well suited to, the algorithms and approaches used to traditionally solve the problem that needs to be reformulated—a significant up-front cost. Such an investment, if made to a suitable problem, will yield similar remarkable gains similar to those we have demonstrated in this study. Aside from the monetary and energy costs associated with the traditional big computing saved with lightweight event-based computing architectures, eliminating weeks off month-long runtimes allows bigger parameter space exploration. This enables researchers to more closely explore their system of interest—facilitating more efficient science across a broad spectrum of disciplines.

AUTHOR CONTRIBUTIONS

Mark Vousden: Conceptualization; Data curation; Methodology; Software; Writing – original draft; Writing – review & editing. **Jordan Morris**: Investigation; Software; Writing – review &

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

editing. **Graeme Bragg**: Investigation; Software; Writing – review & editing. **Jonathan Beaumont**: Investigation; Software. **Ashur Rafiev**: Investigation; Software; Writing – review & editing. **Wayne Luk**: Funding acquisition; Writing – review & editing. **David Thomas**: Funding acquisition; Investigation; Software; Writing – original draft; Writing – review & editing. **Andrew Brown**: Funding acquisition; Writing – original draft; Writing – review & editing.

ACKNOWLEDGEMENTS

We acknowledge financial support from EPSRC (EP/N031768/1). We also acknowledge Dr. Julian Charles Shillcock for his Dissipative Particle Dynamics expertise, and his simulation software at https://github.com/Osprey-DPD/osprey-dpd [40].

CONFLICT OF INTEREST

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

The data supporting the packet storm discussion (Figure 2) is available at https://dx.doi.org/10.5281/zenodo.5215832, reference number 31. The other data that support the findings of this study (Figures 5 and 7–10) are available from the corresponding author upon reasonable request.

ORCID

Mark Vousden https://orcid.org/0000-0002-6552-5831

REFERENCES

1. Taylor, M.B.: Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In: DAC Design Automation Conference 2012, pp. 1131–1136. IEEE (2012)
2. Naylor, M., Moore, S.W., Thomas, D.: Tinsel: a manythread overlay for FPGA clusters. In: 2019 29th International Conference on Field Programmable Logic and Applications (FPL), pp. 375–383. IEEE (2019)
3. Brown, A.D., et al.: SpiNNaker—programming model. IEEE Trans. Comput. 64(6), 1769–1782 (2014)
4. Rashti, M.J., et al.: Multi-core and network aware MPI topology functions. In: European MPI Users' Group Meeting, pp. 50–60. Springer (2011)
5. Alexander, P., Scott, B., Bader, M.: A UPC++ actor library and its evaluation on a shallow water proxy application. In: 2019 IEEE/ACM Parallel Applications Workshop, Alternatives to MPI (PAW-ATM), pp. 11–24. IEEE (2019)
6. Rybacki, S., Himmelspach, J., Uhrmacher, A.M.: Experiments with single core, multi-core, and GPU based computation of cellular automata. In: 2009 First International Conference on Advances in System Simulation, pp. 62–67. IEEE (2009)
7. Lee, H., Faruque, M.A.Al: GPU-EvR: run-time event based real-time scheduling framework on GPGPU platform. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6. IEEE (2014)
8. Dietz, H.G., Young, B.D.: MIMD interpretation on a GPU. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 65–79. Springer (2009)
9. Shaw, D.E., et al.: Anton, a special-purpose machine for molecular dynamics simulation. Commun. ACM 51(7), 91–97 (2008). https://doi.org/10.1145/1364782.1364802
10. Furber, S.B., et al.: Overview of the SpiNNaker system architecture. IEEE Trans. Comput. 62(12), 2454–2467 (2012). https://doi.org/10.1109/tc.2012.142
11. Murray, A.F., Smith, A.V.W.: Asynchronous VLSI neural networks using pulse-stream arithmetic. IEEE J. Solid State Circ. 23(3), 688–697 (1988). https://doi.org/10.1109/4.307
12. Burr, J.B.: Digital neural network implementations. Neural Networks Concepts Appl. Implement. 3, 237–285 (1991)
13. Chicca, E., Indiveri, G., Douglas, R.J.: An event-based VLSI network of integrate-and-fire neurons. In: 2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No. 04CH37512), vol. 5. IEEE (2004). V–357
14. George, R., et al.: Event-based softcore processor in a biohybrid setup applied to structural plasticity. In: In 2015 International Conference on Event-Based Control, Communication, and Signal Processing (EBCCSP), pp. 1–4. IEEE (2015)
15. Chen, T., et al.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. Comput. Architect. News 42(1), 269–284 (2014). https://doi.org/10.1145/2644865.2541967
16. Yan, Y., et al.: Efficient reward-based structural plasticity on a SpiNNaker 2 prototype. IEEE Trans. Biomed. Circuits Syst. 13(3), 579–591 (2019). https://doi.org/10.1109/tbcas.2019.2906401
17. Mayr, C., Hoeppner, S., and Furber, S.: SpiNNaker 2: a 10 million core processor system for brain simulation and machine learning. arXiv preprint arXiv:1911.02385 (2019)
18. Davies, M., et al.: Loihi: a neuromorphic manycore processor with on-chip learning. IEEE Micro 38(1), 82–99 (2018). https://doi.org/10.1109/mm.2018.112130359
19. Amiri, M., et al.: FPGA-based soft-core processors for image processing applications. J. Signal Processing Syst. 87(1), 139–156 (2017). https://doi.org/10.1007/s11265-016-1185-7
20. Stumpp, D.C., et al.: hARMS: a hardware acceleration architecture for real-time event-based optical flow. IEEE Access 10, 58181–58198 (2022). https://doi.org/10.1109/access.2022.3172396
21. Butts, M., Jones, A.M., Paul, W.: A structural object programming model, architecture, chip and tools for reconfigurable computing. In: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), pp. 55–64. IEEE (2007)
22. Jones, A.M., Butts, M.: TeraOPS hardware: a new massively-parallel MIMD computing fabric IC. In: 2006 IEEE Hot Chips 18 Symposium (HCS), pp. 1–15. IEEE (2006)
23. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 141–151. IEEE (2012)
24. Kobayashi, R., et al.: GPU-FPGA heterogeneous computing with OpenCL-enabled direct memory access. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 489–498. IEEE (2019)
25. Hu, Y., Liu, Y., Liu, Z.: A survey on convolutional neural network accelerators: GPU, FPGA and ASIC. In: 2022 14th International Conference on Computer Research and Development (ICCRD), pp. 100–107. IEEE (2022)
26. Villasenor, J., Mangione-Smith, W.H.: Configurable computing. Sci. Am. 276(6), 66–71 (1997). https://doi.org/10.1038/scientificamerican0697-66
27. Hilbert, J.L.: Introduction to ASIC technology. Appl. Specif. Integrated Circ. Technol. 23(12), 1–6 (1991). https://doi.org/10.1016/b978-0-12-234123-6.50006-8
28. Zuchowski, P.S., et al.: A hybrid ASIC and FPGA architecture. In: IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002, pp. 187–194. IEEE (2002)
29. Boutros, A., Yazdanshenas, S., Vaughn, B.: You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference. ACM Trans. Reconfigurable Technol. Syst. (TRETS) 11(3), 1–23 (2018). https://doi.org/10.1145/3242898
30. Todman, T., Wayne, L.: Custom enhancements to networked processor templates. In: 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 224–229. IEEE (2021)
31. Bragg, G.M., Vousden, M.: Event-based computing packet storm dataset - 512 square heated plate (2021). https://doi.org/10.5281/zenodo.5215832

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

32. Scott, K., Gelatt, D.C., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983). https://doi.org/10.1126/science.220.4598.671
33. Černỳ, V.: Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. J. Optim. Theor. Appl. 45(1), 41–51 (1985). https://doi.org/10.1007/bf00940812
34. Chen, C., Tiong, L.K.: Using queuing theory and simulated annealing to design the facility layout in an AGV-based modular manufacturing system. Int. J. Prod. Res. 57(17), 5538–5555 (2019). https://doi.org/10.1080/00207543.2018.1533654
35. Sung, W.P., Kim, S., de Weck, O.: Optimization of reconfigurable satellite constellations using simulated annealing and genetic algorithm. Sensors 19(4), 765 (2019). https://doi.org/10.3390/s19040765
36. Sergey, G., Daniil, Z., Rustam, C.: Simulated annealing based placement optimization for reconfigurable systems-on-chip. In: 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), pp. 1597–1600. IEEE (2019)
37. Ben-Ari, M., Ben-Arî, M.: Principles of Concurrent and Distributed Programming. Pearson Education (2006)
38. Chapiro, D.M.: Globally-asynchronous Locally-Synchronous Systems. Technical Report. Stanford Univ CA Dept of Computer Science (1984)
39. Groot, R.D., Warren, P.B.: Dissipative particle dynamics: bridging the gap between atomistic and mesoscopic simulation. J. Chem. Phys. 107(11), 4423–4435 (1997). https://doi.org/10.1063/1.474784
40. Shillcock, J.C., et al.: Phase behaviour and structure of a model biomolecular condensate. Soft Matter 16(27), 6413–6423 (2020). https://doi.org/10.1039/d0sm00813c
41. Shillcock, J.C., et al.: Coupling bulk phase separation of disordered proteins to membrane domain formation in molecular simulations on a bespoke compute fabric. Membranes 12(1), 17 (2021). https://doi.org/10.3390/membranes12010017
42. George, K., Kumar, V.: METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices (1997)
43. Poliakov, I., Sokolov, D., Mokhov, A.: Workcraft: a static data flow structure editing, visualisation and analysis tool. In: International Conference on Application and Theory of Petri Nets, pp. 505–514. Springer (2007)
44. Lancet, D., Human Genome Sequencing Consortium, et al.: Initial sequencing and analysis of the human genome. Nature 409(6822), 860–921 (2001)
45. Venter, J.C., et al.: The sequence of the human genome. Science 291(5507), 1304–1351 (2001)
46. MacArthur, J., et al.: The new NHGRI-EBI catalog of published genome-wide association studies (GWAS Catalog). Nucleic Acids Res. 45(D1), D896–D901 (2017). https://doi.org/10.1093/nar/gkw1133
47. International HapMap 3 Consortium, et al.: Integrating common and rare genetic variation in diverse human populations. Nature 467(7311), 52–58 (2010). https://doi.org/10.1038/nature09298
48. Brody, J.A., et al., The Cohorts for Heart and Aging Research in Genomic Epidemiology CHARGE Consortium, TOPMed Hematology and Hemostasis Working Group, CHARGE Analysis and Bioinformatics Working Group: Analysis commons, a team approach to discovery in a big-data environment for genetic epidemiology. Nat. Genet. 49(11), 1560–1563 (2017). https://doi.org/10.1038/ng.3968
49. Li, Na, Stephens, M.: Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. Genetics 165(4), 2213–2233 (2003). https://doi.org/10.1093/genetics/165.4.2213
50. Collins, M.: The Forward-Backward Algorithm. Columbia Columbia Univ (2013)
51. Tony Hilton Royle Skyrme: A non-linear field theory. In: Selected Papers, with Commentary, of Tony Hilton Royle Skyrme, pp. 195–206. World Scientific (1994)
52. Heinze, S., et al.: Spontaneous atomic-scale magnetic skyrmion lattice in two dimensions. Nat. Phys. 7(9), 713–718 (2011). https://doi.org/10.1038/nphys2045
53. Zhang, X., Ezawa, M., Zhou, Y.: Magnetic skyrmion logic gates: conversion, duplication and merging of skyrmions. Sci. Rep. 5(1), 1–8 (2015). https://doi.org/10.1038/srep09400
54. Brown, W.F.: Micromagnetics. Number 18. Interscience Publishers (1963)
55. Bisotti, M.-A., et al.: Fidimag–a finite difference atomistic and micromagnetic simulation package. arXiv preprint arXiv:2002.04318, (2020)

How to cite this article: Vousden, M., et al.: Event-based high throughput computing: A series of case studies on a massively parallel softcore machine. IET Comput. Digit. Tech. 17(1), 29–42 (2023). https://doi.org/10.1049/cdt2.12051

1751861x, 2023, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cdt2.12051 by University Of Southampton, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License