Received: 18 June 2021 | Revised: 23 February 2022 | Accepted: 21 March 2022

DOI: 10.1049/cdt2.12041

ORIGINAL RESEARCH

Synchronization in graph analysis algorithms on the Partially Ordered Event-Triggered Systems many-core architecture

Ashur Rafiev[1] | **Alex Yakovlev**[1] | **Ghaith Tarawneh**[1] | **Matthew F. Naylor**[2] | **Simon W. Moore**[2] | **David B. Thomas**[3] | **Graeme M. Bragg**[4] | **Mark L. Vousden**[4] | **Andrew D. Brown**[4]

[1]School of Engineering, Newcastle University, Newcastle upon Tyne, UK

[2]Computer Architecture Group, Cambridge University, Cambridge, UK

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

[4]Electronics and Computer Science, University of Southampton, Southampton, UK

Correspondence

Ashur Rafiev, School of Engineering, Newcastle University, Newcastle upon Tyne NE1 7RU, UK.
Email: ashur.rafiev@ncl.ac.uk

Funding information

Engineering and Physical Sciences Research Council, Grant/Award Number: EP/N031768/1

Abstract

One of the key problems in designing and implementing graph analysis algorithms for distributed platforms is to find an optimal way of managing communication flows in the massively parallel processing network. Message-passing and global synchronization are powerful abstractions in this regard, especially when used in combination. This paper studies the use of a hardware-implemented refutable global barrier as a design optimization technique aimed at unifying these abstractions at the API level. The paper explores the trade-offs between the related overheads and performance factors on a message-passing prototype machine with 49,152 RISC-V threads distributed over 48 FPGAs (called the Partially Ordered Event-Triggered Systems platform). Our experiments show that some graph applications favour synchronized communication, but the effect is hard to predict in general because of the interplay between multiple hardware and software factors. A classifier model is therefore proposed and implemented to perform such a prediction based on the application graph topology parameters: graph diameter, degree of connectivity, and reconvergence metric. The presented experimental results demonstrate that the correct choice of communication mode, granted by the new model-driven approach, helps to achieve 3.22 times faster computation time on average compared to the baseline platform operation.

1 | INTRODUCTION

Many practical applications use graphs to study complex relationships between entities as diverse as proteins, bank accounts, and social profiles. The introduction of the small-world network model almost 2 decades ago [1] sparked immense research interest in graphs when it showed that graphs of many real-world systems are underpinned by a common set of organising principles [2]. This discovery led to the emergence of *network science*, a multidisciplinary field dedicated to the study of complex large-scale networks [3]. Network science draws heavily on our ability to compute graph properties using techniques that build on elementary operations such as graph (network) traversal. This is becoming more challenging as growth in the underlying application area datasets is making it possible to construct graphs of unprecedented scales, several orders of magnitude beyond what can be accommodated and analysed efficiently on a single commodity computer.

Great interest is therefore invested in scalable graph processing methods and platforms, including distributed architectures and massively parallel supercomputers such as CPU and GPU-based clusters [4] and FPGAs [5]. Application development at that scale requires solving an important problem of execution policies and semantics as well as scheduling resources in time and in space. The *event-based programming model* addresses this problem by abstracting computation units as independent actors that work within their local memory segments (*states*) and communicate via small messages (*events*). Extending this model with global events introduces new trade-offs and has the potential to solve the problem of synchronization of execution flows and the supervision of communications, which calls for an investigation.

We explore these trade-offs using the Partially Ordered Event-Triggered Systems (POETS) platform [6, 7]: a prototype hardware-software stack specifically suited for computing problems that can be decomposed into a large number of inter-communicating processes, and whose performance relies critically on communication efficiency. It provides a massively-parallel multi-NoC hardware architecture comprised of a large number of small RISC-V cores [8] communicating via small (up to 64 bytes) packets. The FPGA-based platform implementation offers an additional benefit of configurability by providing an opportunity to implement application-specific experimental features in hardware and exposing these features to the programming model through a flexible API. The POETS programming model has been expanded in [9] by introducing a custom hardware-implemented termination detection feature that can also be used as a *refutable* global synchronization barrier.

Graph analysis algorithms pose an additional challenge to this type of exploration as they display high variability in their behaviour depending on application input graphs. Different levels of synergy between an application graph topology and a hardware topology may result in an affinity for a specific mode of communication, as illustrated in Figure 1. While in some graphs a well-behaved communication pattern emerges naturally, others require supervision of the message traffic to prevent congestion and related performance penalties such as the *reconvergence* effect to be discussed in Section 4. This shows that the preference for one type of communication over the other is application-dependent, hence the decision on the protocol has to be made per application in order to gain the maximum improvement of overall computation time.

The work presented in this paper addresses this challenge by building a classifier model capable of categorising applications by their synchronization preference based solely on the application graph topology. Figure 1 outlines the proposed model-based workflow for application development with configurable communication capabilities. The core of the workflow is the model fitting that uses experimental data to refine the performance model, which is then used for determining optimal communication protocol based on the properties of the application graph.

In summary, the paper makes the following **contributions**:

- It evaluates the efficiency of global synchronization based on hardware-implemented termination detection and enhances the POETS hardware-software stack with the choice between synchronized and asynchronous communication protocols.
- It builds experimental evidence and analyzes a variety of graph topology classes for the effect of communication types on performance and scalability.
- It develops a classifier model using the experimentally obtained performance characteristics. The model-based workflow expands the application design space and unlocks greater performance potential by efficiently exploiting new configuration options.

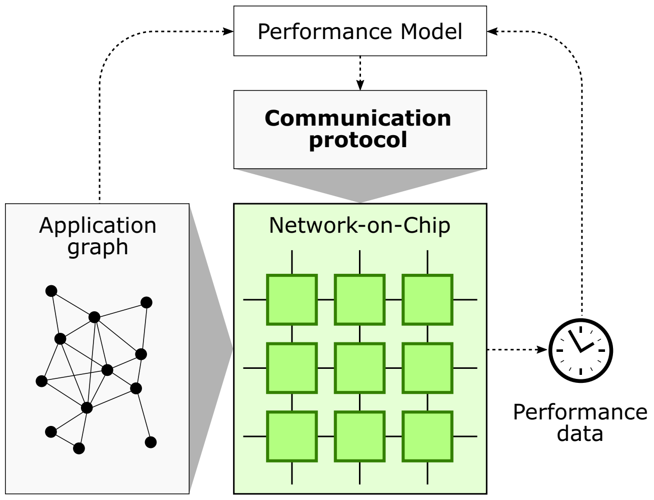

FIGURE 1 An overview of model-driven application development for graph analytic applications on network-on-chips

The paper is organised as follows. Section 2 motivates interest in distributed graph processing by presenting an overview of several application areas where large-scale network processing is used. Section 3 presents the POETS hardware architecture and discusses its features. Section 4 presents the description of network traversal algorithms, while Section 5 presents benchmarking results and analysis. Section 6 concludes the work.

2 | APPLICATIONS AND MOTIVATION

Network traversal is a subclass of graph analytic algorithms used for analysing graph properties such as, for example, shortest paths. In this work, we focus on network traversal applications that can be described with a combination of the following requirements:

- type of analysis, for example, *single-source shortest paths* to all nodes (single source shortest path (SSSP)) or *all-pair shortest paths* (APSP);
- graph topology: irregular graphs or regular grids;
- edge weights: weighted versus unweighted/unit edges.

This section motivates the work with practical examples from the fields of biology and geology, which represent drastically different graph types and form a foundation for our benchmarks in Section 5.

2.1 | Drug discovery

Biological systems can be modelled as networks of protein interaction representing normal cellular functions. Disease mechanisms can be considered as emerging from collections of pathological interactions over such interaction networks. Identification and perturbation of those systems aimed at combating the disease is an underlying principle of drug discovery. The robustness and resilience to failure or functional perturbation of a small subset of constituents in complex biological systems imply that the substantial levels of change require multiple elements to be perturbed simultaneously [10]. e-Therapeutics [11] has developed a practical, in silico, systems-based approach to drug discovery based on the above principles [12]. Numerous measures have been used in studies of network percolation and robustness with two commonly used measures being network diameter and the average of APSP. Practical percolation experiments used in drug discovery at e-Therapeutics involve calculating the impact of removing various protein sets on networks spanning up to 20,000 proteins and over half a million interactions. There can be hundreds of thousands of potential drug candidates (protein sets) whose removal impact must be evaluated, resulting in a very large number of shortest path calculations. As such, performance improvements in network analysis can shorten drug discovery time dramatically. For benchmarking purposes, this use case represents *APSP calculation on irregular unweighted graphs*.

2.2 | Seismic raytracing

Marine geologists use seismic tomography to construct 3D models of the seismic velocity at points in the crust and underlying mantle in order to study the geophysical properties and understand magmatic, hydrothermal, and tectonic processes involved [13, 14]. Seismic tomography works by measuring the arrival times of seismic waves from a source of seismic radiation and comparing the observed arrival times to the arrival times predicted by a model. The model is then perturbed to minimise the misfit between the predicted and measured arrivals times using a variation of the least-squares approach [15]. At each iteration in the tomographic process, ray paths and their travel times from sources to receivers must be determined for the updated model. A volume of Earth's crust is represented as a spatially regular 3D grid overlayed with an acoustic velocity model represented by edge weights. The problem reduces to an *SSSP calculation over a regular weighted grid*. Due to the volumetric nature of the data, the grid sizes can vary from millions to tens of billions of points depending on the target accuracy of the model.

2.3 | Parallel platforms

Some of the earliest work on parallel graph processing was based on the *Parallel Random Access Machine* (PRAM) model where multiple processors access shared memory and perform the visiting `step` of graph traversal in parallel [16]. This idealised view of parallel graph traversal ignores communication, synchronization, and load-balancing costs/performance, each a key performance determinant in practice. Work on multi-threaded systems has therefore attempted to approach the ideal traversal algorithm complexities based on PRAM by addressing these factors through a combination of architectural and algorithmic solutions targeting specific platforms [17].

The range of available hardware platforms creates a multi-dimensional spectrum with respect to the degree of distribution of computation and memory, as well as the aspects of reconfigurability. *GPUs* are commodity units with performant off-the-shelf communication and synchronization primitives and are therefore a popular platform for researching parallel graph traversal. Studies on GPUs have investigated the performance impact of thread placement [18], memory representations and their tradeoffs [4], as well as the implementation of particular algorithms (e.g. directional search [19]) and load-balancing techniques (e.g. partitioning [20]). *Massively parallel machines* continue to assume the top positions in GRAPH500 through a combination of hardware scale and architectural/algorithmic innovation. For clusters of commodity processors, techniques to improve scalability have focussed on data representation, compression, and communication [4], partitioning and task mapping [21] as well as algorithmic techniques such as delegate vertices [22]. One of the main limitations hindering the use of commodity hardware (CPUs and GPUs) is to traverse very large graphs is RAM size. Studies have therefore examined ways to off-load graph data to larger external memories while mitigating the associated performance penalty using *in-memory computing* [23, 24].

FPGAs offer greater flexibility in architectural design and are therefore another common medium for investigating parallel graph traversal. Studies on FPGAs presented various techniques to optimise graph data structures [25, 26] and memory access patterns [27, 28]. FPGA platforms also offer a novel approach to solving graph problems by compiling and mapping graph topologies directly onto the hardware communication fabric. The previous study [5] successfully demonstrated such use of FPGAs; however, the method was constrained by the size of the FPGA fabric and in particular by the number of available connections. To overcome this problem, we use an FPGA-based architecture consisting of an extremely large number of very simple CPU cores, embedded in an efficient message-exchange fabric, such as the POETS platform presented in the following section.

2.4 | Related modelling methodologies

The classical methods of representing computational complexity for single-processor applications, like the big-O notation, are not suitable for many-core platforms where the multiple agents interact asynchronously creating a so-called globally asynchronous locally synchronous system [29].

Formal modelling techniques can be applied at the level of communication protocols and circuit designs where all components are well-understood. Timed Colour Petri Nets have been used to model dataflows and communication protocols at the hardware level in order to predict the performance of asynchronous systems [30]. Stochastic methods like Stochastic Timed Petri-Nets have also been applied to the circuit-level design to estimate upper and lower bounds for asynchronous performance [31]. The benefit of using a Petri Net model is that it can represent the system structure and interactions as a graph and can be simulated at large scale [32].

Empirical models use a different approach by focussing on the observation rather than the description of a system. These models are well-suited for detecting unknown or hidden interactions within a system and are typically built by applying statistical analysis and machine learning techniques to the experimental data. These include model-fitting techniques like multivariate linear regression (MVLR) [33], which is often used in conjunction with the principal component analysis (PCA) [34]. The method of gradient descent can be applied to non-linear model hypotheses as long as they have derivatives [35]. Compared to neural-network machine learning, model fitting techniques like MVLR have the important benefit of providing insights into the physical interplays and implications. This is because the analytical model parameters can usually be traced to physical properties within a system.

A *classifier* is a type of empirical model that separates experimental data into different categories (classes) based on given criteria [36]. The novelty of the POETS platform is that it allows choosing between synchronized and asynchronous execution modes, therefore calling for a model that would classify the workload depending on which mode benefits best to the computation time. Building such a model is the main focus of the experimental part of this paper.

3 | PARTIALLY ORDERED EVENT-TRIGGERED SYSTEMS HARDWARE PLATFORM

Today's general-purpose processors rely on elaborate hardware features such as superscalar execution and cache coherency to automatically infer parallelism and communication from general workloads. But for inherently parallel workloads with explicit communication patterns, which are common in the high-performance computing domain, these costly hardware features become much less valuable. Instead, processors consisting of larger numbers of far simpler cores, communicating by message-passing, can potentially achieve more performance from a single chip and scale more easily to large numbers of chips.

This is the hypothesis of the POETS project (POETS [6, 7]), which forms the wider context for the work described in this paper. On the project, researchers have constructed a prototype platform consisting of a 48-FPGA cluster and a many-core RISC-V overlay called Tinsel [37] programmed on top. The Tinsel overlay has a regular structure, consisting of a scalable grid of *tiles* connected by a reliable communication fabric that extends both within each FPGA and throughout the FPGA cluster. By default, a tile consists of four RV32IMF multithreaded cores [8], clocked at 240 MHz, sharing an floating-point unit, 128 KB thread-partitioned data cache, and a *mailbox*. Each core supports 16 barrel-scheduled hardware threads. A separate network is used to connect caches in tiles to off-chip memories. A single-FPGA view of the overlay is depicted in Figure 2b, and a single tile is shown in Figure 2c.

Each POETS mailbox serves four cores and contains a memory-mapped scratchpad storing up to 64 KB of incoming and outgoing messages, which can also be used as a small general-purpose local memory. Messages are variable-length, containing up to four flits, with each flit holding 128 bits of payload. The mailbox allows threads to trigger the transmission of outgoing messages, to allocate space for incoming messages, and to consume those messages when they arrive, all via custom RISC-V CSRs (control/status registers).

The FPGA cluster comprises a grid of 48 DE5-Net boards (8 server boxes, 6 boards per box) connected together using 10G reliable links, as shown in Figure 2a. Each box also contains an ×86 host with an additional bridge FPGA board that connects to the Tinsel mesh. The overlay distributes naturally over this cluster to yield a 3072 core system (49,152 hardware threads), where any thread can send messages to any other thread.

3.1 | Hardware-level termination detection

A crucial feature added to the POETS platform in [9] is a *refutable* barrier primitive with a number of attractive properties: (1) it has a simple semantics based on termination detection; (2) it does not introduce race conditions or non-determinism; (3) it does not depend on the use of costly synchronous send operations; and (4) it allows an arbitrary asynchronous computation to occur within each synchronous `step` of a parallel application (*self-timed* synchronization).

At the level of the Tinsel overlay, the feature is represented with a blocking function that is released when either (1) a message is available for that thread to receive, or (2) all threads in the entire system are blocked on a call to this function and there are no undelivered messages in the system. The function returns zero in the former case and non-zero in the latter. A return value greater than one denotes that all callers voted true. The voting mechanism is a simple form of aggregation that is useful for globally detecting termination, e.g. all threads agreeing they are stable since the previous time `step`.

In the remainder of this section, we present the implementation details underpinning the above feature. We start by looking at a classic termination-detection algorithm and then describe how we refine this algorithm to an efficient hardware implementation.

3.1.1 | Safra's algorithm

Safra's algorithm, as presented by Dijkstra [38], is a classic solution to the problem of detecting termination in distributed systems. It considers a set of *machines*, each of which is either *passive*, if it has indicated that it has no further messages to send, or *active*, otherwise. A machine in the passive state

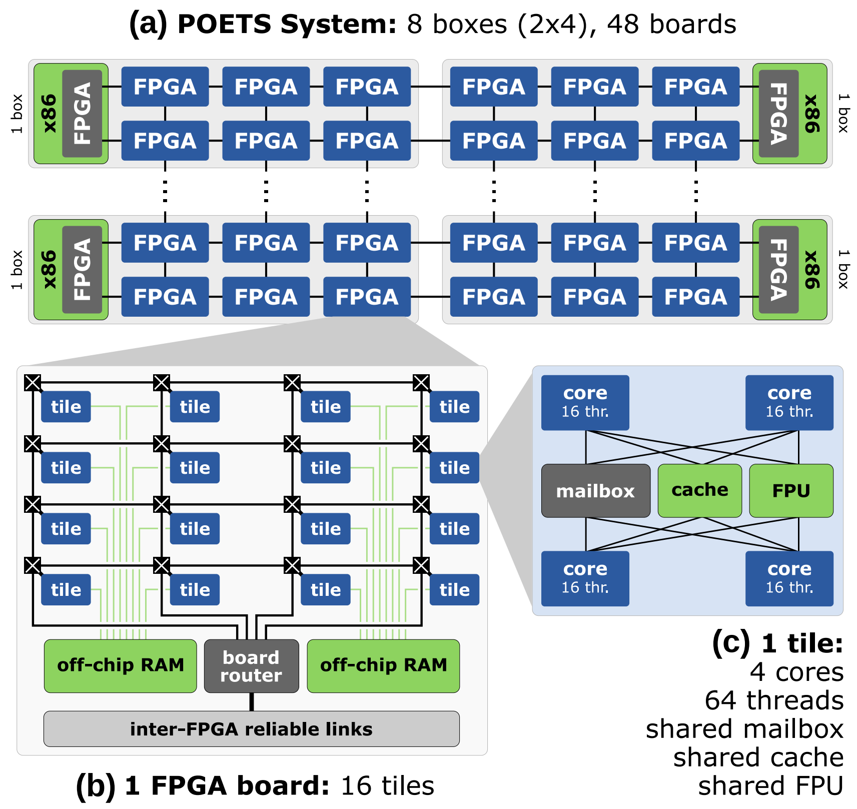

FIGURE 2 Current version of the Partially Ordered Event-Triggered Systems (POETS) cluster architecture (a) with a zoom on a single FPGA board (b) and a tile (c)

automatically transitions to the active state upon receipt of a message. The algorithm detects the case in which all machines are passive and there are no undelivered messages.

The operation of the algorithm is described as a set of rules:

Rule 1: Each machine maintains a local *count* of the number of messages it has sent minus the number it has received.
Rule 2: A termination *token* containing an accumulator, initially zero, is passed from machine to machine in a ring pattern.
Rule 3: Each machine holds on to the token until it becomes passive, at which point it adds its local count to the accumulator in the token and *forwards* the token to the next machine in the ring.
Rule 4: When the token completes a full iteration of the ring, and the final accumulator is zero (i.e. all the local counts sum to zero) then the conditions for termination *may be met.*

The case in which the final accumulator is zero, but termination has not occurred, is as follows. Suppose a machine M, which has already forwarded the token, receives a message and transitions to the active state. On its own, this is acceptable because the final accumulator will exceed zero: M's count has already been sampled before receiving this latest message, and the sender's count is still to be sampled. However, the now-active machine M can send a new message to a machine that has not yet forwarded the token, meaning that the final accumulator may well be zero for the opposite reason: M's count has already been sampled before sending this latest message, and the receiver's count is yet to be sampled. This situation is remedied as follows:

Rule 5: Each machine, and the token, are initially coloured white. On receipt of a message, a machine turns black. When a black machine forwards a token, the token is blackened.
Rule 6: Termination is detected when the token completes a full iteration, and its final accumulator is zero, and its final colour is white.

This remedy catches the situation in which a machine receives a message before its count is sampled. When termination is not detected, a new iteration of the ring is started. Of course, a new iteration can only succeed if black machines are somehow whitened again, leading to one final case:

Rule 7: When a machine forwards a token, it whitens itself.

3.1.2 | Scalable topology

Safra's algorithm is a natural fit for our platform, with machines corresponding to RISC-V threads, and the passive state corresponding to a thread blocked by a barrier. However, there are two main efficiency concerns when using the algorithm at such a fine granularity: (1) we have tens of thousands of threads in our cluster, which is thousands of times more than the number of FPGAs; and (2) if implemented in software, the token would incur the latency of passing through the software stack running on each thread.

In order to achieve greater scalability, we implement the hierarchical termination-detection in hardware with Safra's algorithm working at the granularity of FPGAs rather than threads and synchronous pipelined trees at the intra-FPGA level, as illustrated in Figure 3. To determine a machine's message count and passive/active status at the FPGA level (cumulatively with respect to the individual threads) we use the following hardware structures:

- Each core outputs a pair of wires: one that is pulsed when a thread on that core sends a message, and one that is pulsed when a thread on that core receives a message. A pipelined adder tree reduces these wires to a single signed number that is added to the FPGA's message count on every clock cycle.
- Each core also emits a wire indicating whether all threads on that core are in a call to the barrier function. A pipelined conjunction tree reduces these wires to a single active/passive wire for the whole FPGA.

These reduction networks are non-blocking and have the same depth, which means that the states of all the threads are always sampled at a consistent point in time. If this were not the case, and the count was sampled at a different time to the active/passive status, then a token could be forwarded with an invalid count.

The best-case run-time performance of a single iteration of Safra's algorithm is proportional to the number of machines N multiplied by the inter-machine latency L, that is $O(L \cdot N)$. For efficiency, we exploit parallelism and use a *star* topology instead of a ring: a single *master* sends a token to each machine in parallel, and each machine forwards its token directly back to the master, which sums the individual counts and combines the individual colours accordingly. With all other aspects remaining the same, the algorithm continues to function correctly: the order in which the machines are sampled is not important, and a black token will be produced by any machine that receives a message before it is sampled.

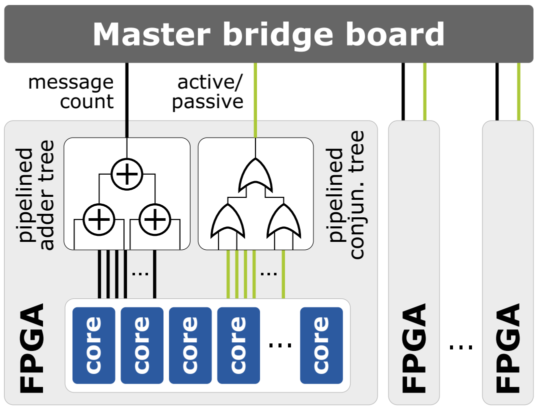

FIGURE 3 Hierarchical structure of the dedicated termination detection hardware

In our cluster, we use one of the FPGA bridge boards (see the FPGAs connected to the ×86 servers in Figure 2) at the origin of the FPGA mesh, as the master. In the worst case, a token from the master will travel along each dimension of the mesh, to reach the FPGA at the far corner, and return back again. For an almost-square mesh like ours, this will result in a best-case run-time of $O\left(4 \cdot L \cdot \sqrt{N}\right)$ for a single iteration of the algorithm. This could be halved by placing the master in the centre of the mesh, but for now, we have chosen to avoid putting the master logic on the homogeneous worker FPGAs.

3.1.3 | Barrier release

Safra's algorithm is only concerned with a *single* machine in the system detecting global termination. To implement the barrier, all threads need to be notified. Therefore we introduce an additional phase to the algorithm that is triggered when termination is detected at the master:

- Once termination is detected, the master sends a "termination detected" notification to each FPGA. Each FPGA releases all its threads from the barrier call (with each call returning non-zero) and responds to the master with an acknowledgement.

Unfortunately, this new phase introduces a race: a released thread can potentially send a message to another thread that has not yet been released, which would result in the receiving thread returning zero from the barrier function. To remedy this, we disable the sending of messages when releasing the calls to the function, and introduce a third and final phase:

- Once the master has received all acknowledgements to the release phase, it sends a "re-enable sending" notification to each FPGA. Each FPGA responds to the master with an acknowledgement.

The end result is a three-phase procedure, where each phase involves a round-trip from the master to the FPGAs (in parallel) and back again. The final two phases only come into play when the first phase successfully detects termination.

It is important to note that the refutable nature of the termination detection barrier means that all other events have higher priority. Holding a token by a machine does not prevent it or any other machine from becoming active. The communication fabric is not blocked and can continue passing messages if there are any; reception of a message on any machine wakes it up from the barrier state and interrupts (cancels) the termination detection algorithm until the device becomes passive again. Therefore, Safra's algorithm does not cause system starvation.

3.2 | High-level API

The POETS hardware-software stack provides a high-level API that accommodates event-based models by creating a relation between logical devices (graph nodes) and hardware threads, and by routing arbitrary application-level edges on top of the regular Tinsel communication mesh. This section outlines the API basics required for a better understanding of the algorithm description in Section 4.

Behaviours of vertices in the graph are defined by *event handlers* that update the node state when a particular event occurs, for example when a message arrives on an incoming edge, or the network is ready to send a new message, or termination is detected. Multiple graph nodes in a single thread are managed by the *soft-switch*. Partially Ordered Event-Triggered Systems infrastructure provides a selection of node-to-core mapping algorithms including mapping based on *METIS* partitioning [39] (prioritising adjacency) and a fast *bucket-fill* (direct) mapping that simply maps nodes in the order they are declared in the graph specification.

Each vertex has access to an application-specific local state $\mathcal{S}$, defined using C++ `struct`, and a *readyToSend* flag indicating that the vertex requests sending a message to the communication fabric. For the purposes of this paper, we assume it takes a Boolean value, and the message is dispatched to all outgoing edges. The message content $\mathcal{M}$ can also be customised but is limited to 56 bytes in the current version of the API. Edges can have associated data of any type; in this paper, we use 16-bit integers as edge weights w.

Initialisation handler The `init` handler is invoked once for every vertex when the application starts, before any other events. FPGA boards and POETS boxes are configured to start asynchronously by default, hence the initialisation may happen at slightly different wall-clock times. Vertices should initialise *readyToSend* in the `init` handler.

Send handler Any vertex indicating that it wishes to send will eventually have its `send` handler called. When called, the `send` handler is provided with a message buffer, to which the outgoing message should be written. It is an application's responsibility to clear the *readyToSend* flag, otherwise the system assumes the vertex wants to send more messages (the latter case is useful for implementing large multi-message protocols).

Receive handler A message arriving at a vertex causes the `recv` handler of the vertex to be called with a pointer to the message and a pointer to the weight associated with the incoming edge along which the message has arrived. The API prioritises receiving messages before sending to drain the network as fast as possible. Therefore a vertex may get multiple `recv` events before it is allowed to `send`.

In order to provide support for the new termination-detection feature, this paper extends the API with the following events:

Step handler The `step` handler is called when termination is detected, that is no vertex in the entire graph wishes to send, and there are no messages in-flight. The return value indicates whether or not the vertex wishes to continue executing. Typically, an asynchronous application will simply return "false", while a synchronous one will do some compute, perhaps requesting to send again, and return "true" to start a new time `step`.

Finish handler If the conditions for calling the `step` handler are met, but the previous call of the `step` handler returned "false" at every vertex, then the finish handler is called. At this stage, each vertex may optionally send a message to the host by writing to the provided buffer and returning "true".

At this abstraction level, there is no direct control over the underlying hardware features. However, the API is flexible enough to provide the following control over the communication and layout: controlling the message traffic by using *readyToSend* flag and API events; choosing to use or not to use the `step` handler for termination detection or global synchronization; choosing the device mapping, including the number of graph nodes per core; controlling the memory allocation by changing the vertex state size. The next section focuses on the shortest path calculation algorithms implemented at the API level.

4 | ALGORITHMS

We use SSSP as an example of a communication-heavy graph analysis application to evaluate the effectiveness of synchronization in a practical setting. In the distributed SSSP, the messages propagate along graph edges accumulating edge weights along the way. The algorithm terminates when all nodes are reached. The result can be an array of path lengths, an aggregate sum, or an average. This section describes the high-level POETS API implementation of the SSSP algorithm. The event-based view provides a hardware-independent model, where logical devices directly correspond to graph nodes, and the graph topology forms a communication fabric.

Let's first consider an unweighted network where all edges have unit lengths and the shortest path between two nodes is defined as a number of *hops* between them.

Synchronized: The synchronization-based approach has been proposed in [5] for a different kind of hardware (clocked circuit). This approach separates execution into distinct steps. We adapt it to an event-based model, using synchronization barrier in place of the clock: the barrier ensures all messages sent during the `step` have been received, that is there are no in-flight messages before the start of the next `step`. An important distinction is that, in the case of POETS, the stepping is *self-timed*, thanks to termination detection, so the computation between the steps can be arbitrarily long. Figure 4 shows a `step`-by-step example. In Step 0 the source node sends token messages to its neighbours. On each subsequent `step`, if any given node receives a message for the first time, it

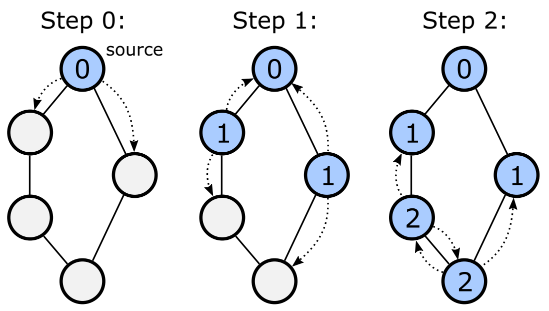

FIGURE 4 Synchronized single source shortest path (SSSP) step-by-step example

updates its state to the step counter value and propagates token messages to its neighbours.

With the synchronized message-passing approach to SSSP, the propagation of messages forms a "front" moving one hop per step of the algorithm, hence the step counter directly corresponds to the shortest path length. The algorithm is finished when all nodes have been reached and therefore no messages to be sent. Messages carrying new information are called the *head traffic*, and the messages that are reflected back and discarded without causing state updates are called the *tail traffic*. The tail traffic has minimal overhead as it is discarded immediately; for the POETS platform, it is a more efficient approach than "pre-filtering" messages on the sender side as the latter requires larger vertex state and would cause a memory bottleneck. We are primarily interested in the head traffic as it forms a wave of updates radiating from the source node and, in the case of the synchronized SSSP, travelling at a constant speed. The behaviour of the head traffic is different in asynchronous SSSP or weighted graphs, as will be discussed later in this section.

Asynchronous (packet storm): In the packet storm algorithm, nodes send updates as soon as they receive incoming messages without the need for global synchronization. There is no notion of a global step counter, therefore each node has to remember its distance to the source in the local memory, and the updated distance is propagated via messages. Event-triggered nature of the asynchronous approach introduces automatic load balancing as congested parts of the network become slower while other parts keep up the speed.

Algorithm 1 shows SSSP implementation details based on the high level API described in Section 3.2. The implementation supports edge weights w (and unweighted graphs as a special case with $w = 1$) and both types of synchronization, as controlled by the compile-time switch *async*. Node state is defined as a tuple $\mathcal{S} := \langle dist, changed \rangle$, where *dist* stores the distance to the source (or 0 for the source node) and *changed* indicates that the node has an update to send. The message type $\mathcal{M}$ carries a single value: the latest distance update *dist*. Most of the algorithm is the same for both synchronization types with the notable differences in recv and step event handlers. In recv, when the state is updated, the asynchronous version immediately flags the readiness to send an update message, whereas the synchronized version only flags the update but does not send (lines 17–21). The step function is invoked by the termination detection hardware, which for asynchronous SSSP signals is the end of the computation (line 31). In the synchronized SSSP, it means the end of a computation step, so the algorithm needs to decide whether to progress or terminate depending on if there are any updates to be sent (lines 33–38). The result returned from the step function is passed to the voting hardware, hence the whole system keeps running as long as there is at least one non-terminated vertex.

Intuitively, it may appear that, for distributed computation, asynchronous algorithms must be better than globally synchronized. However, there are two caveats: (1) uncontrolled bursts of messages may cause congestion and slow down communication fabric, and (2) a distributed asynchronous communication may not guarantee the ordering of arriving messages, so the first received message cannot be used as an indicator of the shortest path. In fact, due to out-of-order arrival, some messages may carry data that does not contribute to the final result and therefore reduce the overall utility of the communication fabric. Head traffic from the source travels along all possible paths with different speeds, and when it *reconverges*, the following interaction takes place: if the "true" shortest path message arrives first, all later updates will be discarded, and only the valid head traffic will continue further; however, if an alternative path arrives earlier, it will still cause a state update (temporary, until the "true" shortest path arrives) and will keep propagating as "false" head traffic. In the worst case, reconvergence may cause an exponential explosion of message traffic.

In practice, the probability of the worst-case scenario is low as the degree of connectivity is not the only determining factor. Figure 5 shows an example of regular grids demonstrating the case. In a 2D square, if only orthogonal edges are allowed, there are only two paths between A and D: ABD and ACD. Both paths are of length two and are valid solutions for the shortest path, so there is no negative effect from path reconvergence. If we allow diagonal edges, three other paths also become possible: ABCD, ACBD, and AD. The latter is the only shortest path solution in this case, but there are four other valid paths that have a probability of arriving early. Similarly, in a 3D cube with only orthogonal edges, there are six possible A to H paths (ABDH …AEGH), all can be valid solutions with the length three. Diagonal edges transform the 3D cube into a

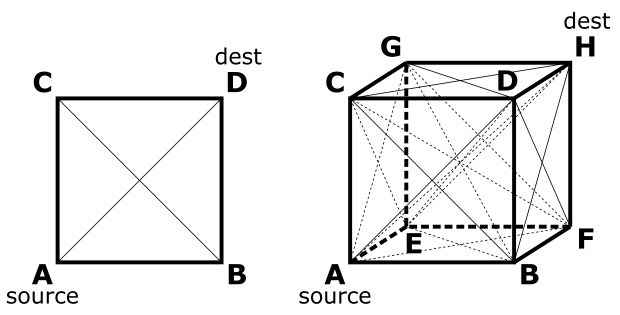

FIGURE 5 Example of reconvergence in regular topologies: 2D square and 3D cube, discussed in Section 4

Algorithm 1 Node-level event-based implementation of SSSP

```
1: $\mathcal{S} := \langle dist, changed \rangle$                ▷ node state
2: $\mathcal{M} := \langle dist \rangle$                          ▷ message type
3: $async$                                                        ▷ compile-time switch

4: procedure init                                                 ▷ initialize node
5:     if source node then
6:         $\mathcal{S}.dist \leftarrow 0$
7:         $readyToSend \leftarrow$ true
8:     else
9:         $\mathcal{S}.dist \leftarrow \infty$
10:        $readyToSend \leftarrow$ false
11:    end if
12:    $\mathcal{S}.changed \leftarrow$ false
13: end procedure

14: procedure recv($\mathcal{M}, w$) ▷ on receiving a message $\mathcal{M}$ over an
    edge with the weight $w$
15:    if $\mathcal{M}.dist + w < \mathcal{S}.dist$ then             ▷ if shorter
16:        $\mathcal{S}.dist \leftarrow \mathcal{M}.dist + w$         ▷ update solution
17:        if $async$ then                                          ▷ for asynchronous:
18:            $readyToSend \leftarrow$ true                          ▷ send update
19:        else                                                     ▷ for synchronized:
20:            $\mathcal{S}.changed \leftarrow$ true                  ▷ mark updated
21:        end if
22:    end if
23: end procedure

24: procedure send($\mathcal{M}$)                                   ▷ on sending a message
25:    $\mathcal{M}.dist \leftarrow \mathcal{S}.dist$                 ▷ create message
26:    $\mathcal{S}.changed \leftarrow$ false                         ▷ clear updated
27:    $readyToSend \leftarrow$ false                                 ▷ clear send trigger
28: end procedure

29: function step                                                   ▷ on detecting idle network
30:    if $async$ then                                              ▷ for asynchronous:
31:        return false                                             ▷ terminate
32:    else                                                         ▷ for synchronized:
33:        if $\mathcal{S}.changed$ then                            ▷ if update pending:
34:            $readyToSend \leftarrow$ true                          ▷ send update
35:            return true                                          ▷ continue execution
36:        else                                                     ▷ not updated:
37:            return false                                         ▷ terminate
38:        end if
39:    end if
40: end function
```

fully connected graph with 1956 paths of lengths two to seven between A and H, and only one true shortest path of length one (AH).

Performance overheads due to path reconvergence can be estimated by analysing the difference in length across all possible paths weighted by their probabilities. This approach resembles the *path integration* technique from quantum mechanics, but unfortunately, it is not practical as the probability distribution is difficult to obtain accurately from the experiments. Instead, we aim at finding alternative heuristic methods. There is no predictive metric that would clearly stand out because these effects manifest from interactions between multiple hardware and application effects. The choice of communication (synchronization) mode requires a classifier model capable of predicting a general trend from a set of topology-specific parameters (i.e., parameters that can be obtained directly from the application graph alone). Section 5 presents the experiments and describes a methodology for building such a classifier model from the experimental data.

In weighted graphs, the path length is calculated as the sum of weights of constituent edges, so it is independent of the number of hops. Therefore, even with synchronized traversal, there is no guarantee that the first arrived message would correspond to the shortest path, and the effect of reconvergence emerges again. Thus, we differentiate two sources of path reconvergence: (1) *communication-driven* reconvergence caused by asynchronous communication and (2) *application-driven* reconvergence caused by the data (edge weights). Asynchronous traversal in weighed graphs is impacted by both types at the same time.

The relation between these effects and the graph topology will be experimentally explored in the next section. In addition to the parameters like graph size and fanout (degree) distribution, we consider the *step count*, which is defined as the largest number of edges in a shortest path. For unweighted graphs, the `step` count equals to the graph diameter D. If edge weights are taken into account, the shortest path may be larger than the graph diameter. The ratio between the weighted `step` count D_w and the graph diameter (unweighted `step` count) D may serve as an indicator of application-driven reconvergence overheads because it depends not only on the topology but also on the distribution of edge weight values. Although in this metric it does not capture the effects of asynchronous execution directly, it still has the potential to serve as a heuristic for hardware-driven reconvergence as well. The next section provides the evidence for this correlation.

5 | EXPERIMENTAL RESULTS AND ANALYSIS

Tinsel overlay provides a set of performance counters inside POETS threads, which allow cycle-accurate time measurements; however, these measurements are not synchronized between the threads, hence cannot represent the wall-clock time. During termination detection, the finish event is issued from the supervisor FPGA board and arrives at all threads within the inter-machine latency L. For the purposes of processing the performance results after the computation is finished, we can align all collected performance counter readings to the finish event to obtain a valid approximation for the wall-clock time with $\pm L$ uncertainty, which is measured as 150 cycles at 240 MHz (including the time to pass through our reliability layer), that is 625 ns. Using this technique, we can calculate that the average delay for termination detection is roughly 5000 cycles (21 μs at 240 MHz). This delay represents the actual application-level overhead of global synchronization.

This section uses collected performance measurements to explore the system behaviour in different communication modes in relation to the application graph parameters. The data is then used to build performance models and evaluate the platform's capabilities.

5.1 | Initial experiments

We perform a set of exploration experiments on the following graph topologies, with and without edge weights; graph size $|V|$ ranging from $2^{10} = 1024$ to 2^{20} (over a million) nodes:

- ***k*-connected 2D and 3D grids** represent spatially uniform structures with a given degree of connectivity k. 2D grids can have orthogonal ($k = 4$) and diagonal ($k = 8$) connections between neighbours; similar connections in 3D grids give $k = 6$ and $k = 26$ respectively. 3D grids relate to the seismic raytracing use case (Section 2.1).
- Irregular graphs with **normal** degree distribution. The number of edges $|E|$ is specified approximately in a fixed proportion to the number of nodes $|V|$.
- Irregular graphs with exponential degree distribution ($\lambda = |V|/|E|$), are also known as **scale-free graphs.** Scale-free graph benchmarks represent drug discovery use case examples (Section 2.2).
- For the corner case scenarios, we use **binary tree** and **ring** topologies representing the best case and the worst case respectively.

Graphs of size 2^{20} are still considered relatively small for the SSSP problem. We limit our benchmarks to this scale because our goal is to explore the design space by running hundreds of different configurations. The results from real-life application graphs are presented in Section 5.4.

Scale-free graphs naturally contain a small number of nodes with very large fanout, over 9000 edges in our larger examples. Since the POETS system is built on small interconnected devices, such hub nodes break the programming model, so the solution is to split those hub nodes into multiple smaller nodes connected with zero-weight edges. For this reason, truly unweighted graphs are not possible for scale-free topologies and are labelled as graphs with *binary weights* (zero or one) in the following discussions.

Table 1 shows the relation between D and the graph size $|V|$, as well as the ratio D_w/D for uniformly distributed edge weights in the range $[10, 1000]$. The table also shows other graph characteristics, for example fanout (also called *node degree*) and edge count; the numbers represent asymptotic values for $|V| \to \infty$.

In the initial experiments, we observed the interplay between various topological parameters and communication protocols and their effect on performance. In our experiments, we tested two different node-to-thread mappings (direct and METIS) described in Section 3. The METIS mapping provides 20%–60% improved computation time for irregular graphs but gives no apparent benefit when applied to regular grids. For consistency, all results presented in this section are from the direct mapping.

TABLE 1 Characteristics of graph topologies

Topology	$\frac{\lvert E\rvert}{\lvert V\rvert}$	Fanout c	Diameter D	$\frac{D_w}{D}$
4-con. 2D grid	2	4	$\lvert V\rvert^{\frac{1}{2}}$	1.06
8-con. 2D grid	4	8	$\lvert V\rvert^{\frac{1}{2}}$	1.73
6-con. 3D grid	3	6	$\lvert V\rvert^{\frac{1}{3}}$	1.22
26-con. 3D grid	13	26	$\lvert V\rvert^{\frac{1}{3}}$	2.51
Normal	8	16 (mean)	$\approx\lvert V\rvert^{0.136}$	4.33
		$\sigma = 4.00$		
Normal	16	32 (mean)	$\approx\lvert V\rvert^{0.164}$	3.58
		$\sigma = 5.66$		
Scale-free	8	16 (mean)	$\approx 0.533 \cdot \ln\lvert V\rvert$	2.05
Scale-free	16	32 (mean)	$\approx 0.481 \cdot \ln\lvert V\rvert$	2.14
Binary tree	1	3	$\log_2\lvert V\rvert$	1
Ring	1	2	$\frac{1}{2}\lvert V\rvert$	1

Figure 6 shows on a log-to-log scale the performance results for SSSP. We implement and compare the algorithms described in Section 4, namely synchronized (*sync*) and packet storm (*async*). Even though the asynchronous execution is non-deterministic, any noise in the data is found insignificant because all local non-deterministic effects are averaged over a large number of graph nodes. Table 2 summarises the results from Figure 6 and highlights the link between different topology classes and the preference towards a certain type of communication using the speedup metric: speedup for the asynchronous mode is defined as $S_a = t_s/t_a$, where t_s is the computation time of the synchronized version and t_a is the computation time of the asynchronous version on identical input graphs; synchronized mode speedup is defined as its reciprocal: $S_s = 1/S_a = t_a/t_s$. The results show that the choice of the synchronization mode is crucial for the performance: for a 4-connected unweighted 2D grid, asynchronous execution can be 18.25 times faster than synchronized; however, the effect is reversed in some topologies, where the synchronized mode is faster (up to 3.17 times in a 26-connected weighted 3D grid). Across all graph types, six

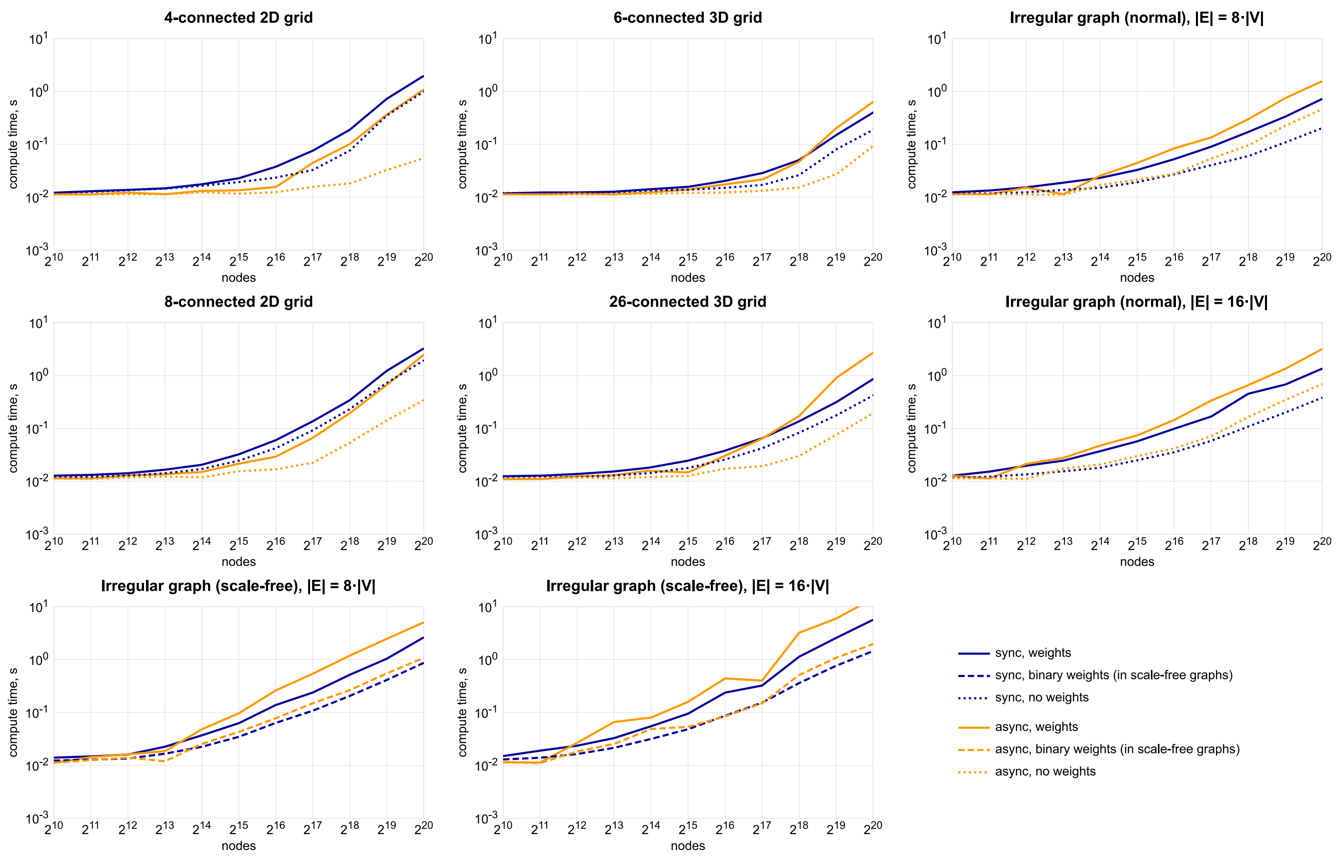

FIGURE 6 SSSP performance results for different graph topologies and sizes

topologies prefer the asynchronous mode with the average speedup of 5.2 and 10 prefer the synchronous mode with the average speedup of 2.4.

For weighted graphs, there is an observable link between the preference for synchronous mode over asynchronous and the proposed reconvergence heuristic: the correlation coefficient between D_w/D and the asynchronous speedup S_a is moderately positive at 0.61. This link can be explained by the effect of communication-driven reconvergence amplified by the application-driven reconvergence, causing a combined negative impact on the performance. For unweighted graphs, the effect of reconvergence is not as pronounced, and the diameter of the graph becomes a more significant factor.

The results show that the irregular graphs favour synchronization, whereas regular grids favour asynchronous execution. It is important to understand at which point the change in behaviour occurs, so we designed an additional experiment to explore the impact of graph regularity on the performance in different synchronization modes. We start the experiment with a regular 1024 × 1024 2D grid and randomly mutate some $r \cdot |E|$ edges to connect arbitrary nodes instead of adjacent, where $0 \leq r \leq 1$ is the ratio of randomized edges. For $r = 1$, the graph transforms into a fully irregular graph with normal degree distribution. Figure 7 presents the results for a range of r values in 0.1 steps; the results show that even the small number of random edges in a regular grid create a strong shift towards synchronized execution. We believe that the main reason for this is the graph diameter: any random edge creates a "shortcut" in a grid greatly reducing its diameter. For example, for $r = 0$, the diameter equals 1024, whereas $r = 0.1$ reduces the diameter to 11. Adding more random edges further reduces the diameter, but at a much lower rate.

The extreme case of the graph diameter hindering the performance is clearly visible in Figure 8 showing statistics for binary tree and ring topologies (here, weighted and unweighted graphs display exactly the same behaviour, hence are not differentiated in the plots). These topologies make less practical sense for SSSP but demonstrate the system behaviour if there is no or very limited reconvergence of traversal paths. Self-timed nature of the POETS synchronization mechanism means that the `step` time can vary depending on the network and core loads. The results for the `step` time show that it remains approximately constant for different graph sizes and only changes due to the differences in topology and communication modes. Therefore, the computation time grows proportionally to the graph diameter for this set of results.

5.2 | Additional experiments

Message traffic density is a crucial factor for asynchronous SSSP. Partially Ordered Event-Triggered Systems system guarantees the arrival of messages, but it does not guarantee the order of arrival. In the extreme cases of congestion, a "starvation" effect may occur when messages from the beginning of the computation get delayed almost until the very end. Figure 9 shows a snapshot from the asynchronous SSSP for an 8-connected 2D grid running on one POETS box (7 FPGA boards, 6144 hardware threads). The size of the grid determines the number of graph nodes per hardware thread, and therefore impacts the traffic density: for 256 × 256 grid, each thread processes 11 graph nodes, whereas for 1024 × 1024 grid, the number of nodes per thread increases to 171. The image captures the arrival of the very first wave of updates coming from the source. In the smaller grid, the update front is almost smooth with only local fluctuations due to asynchronous communication. In the larger example, some of the messages get stalled causing the front to split into multiple wavelets creating a "foam" effect. This does not affect the correctness of computation since the algorithm is delay insensitive by design. All gaps eventually get filled during the execution. However, the effect increases the reconvergence of the update wavefronts and therefore has a negative impact on the performance.

To confirm the hypothesis of the effect of reconvergence on performance, we performed a set of additional SSSP experiments while also counting the number of times the state of the node (i.e., the most recently known value of

TABLE 2 Measured *async* (S_a) versus *sync* (S_s) speedup

	Unweighted		Weighted	
Topology	S_a	S_s	S_a	S_s
Normal, $\|E\| = 8 \cdot \|V\|$	0.44	**2.27**	0.46	**2.15**
Normal, $\|E\| = 16 \cdot \|V\|$	0.56	**1.80**	0.42	**2.35**
Scale-free, $\|E\| = 8 \cdot \|V\|$	0.81	**1.24**	0.52	**1.93**
Scale-free, $\|E\| = 16 \cdot \|V\|$	0.74	**1.35**	0.40	**2.52**
4-con. 2D grid	**18.25**	0.05	**1.81**	0.55
8-con. 2D grid	**5.65**	0.18	**1.32**	0.76
6-con. 3D grid	**2.01**	0.50	0.63	**1.60**
26-con. 3D grid	**2.16**	0.46	0.32	**3.17**

Note: Superior communication mode is highlighted in bold.

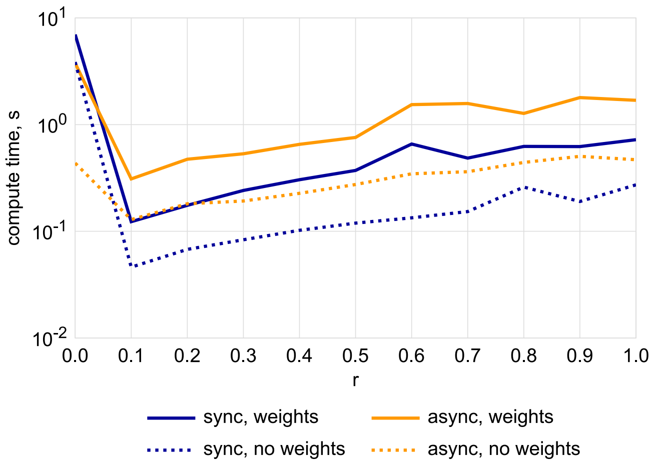

FIGURE 7 Randomized grid performance results; r is the ratio of random edges within a grid

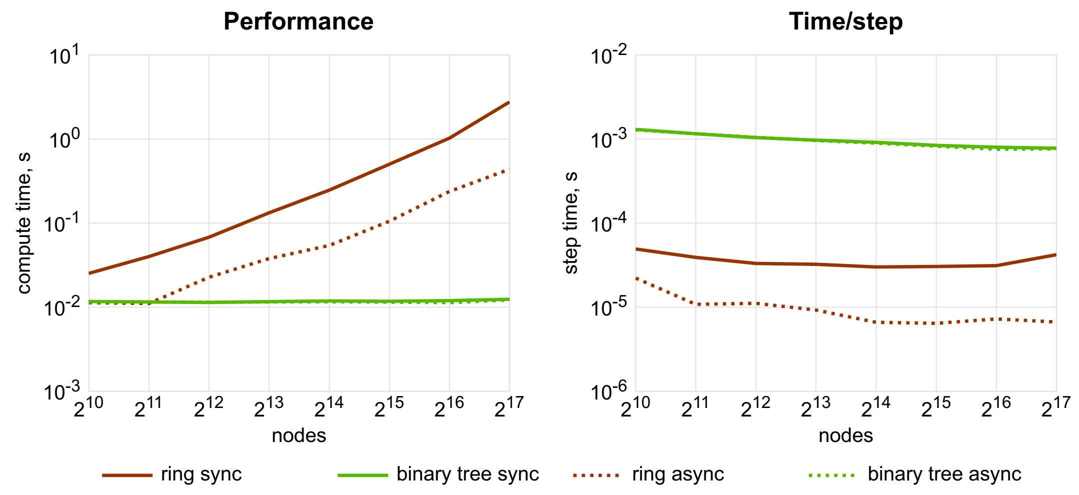

FIGURE 8 SSSP performance and step time for binary tree and ring topologies; for *async*, the step time equivalent is calculated as the total computation time divided by D

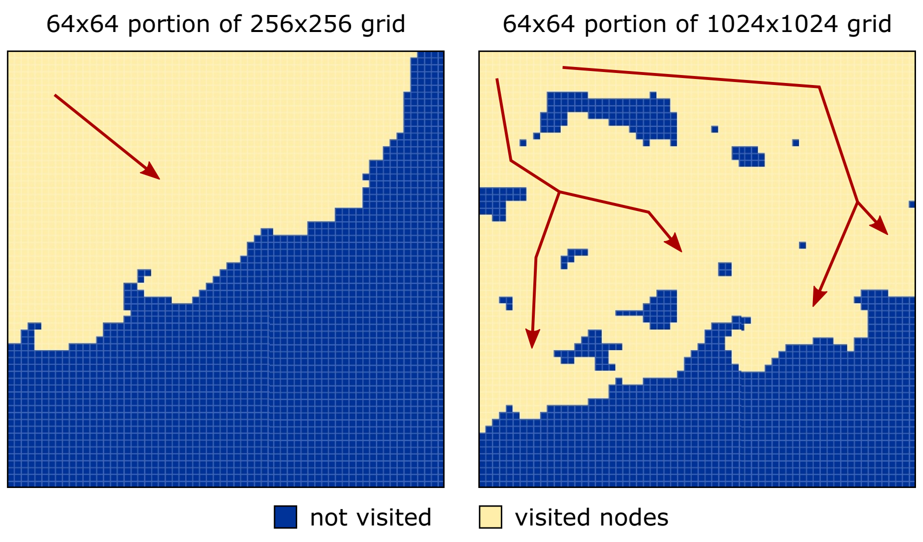
64x64 portion of 256x256 grid
64x64 portion of 1024x1024 grid
not visited
visited nodes

FIGURE 9 A snapshot of the asynchronous single source shortest path (SSSP) progress showing the change in update propagation between different graph sizes due to traffic density. Arrows show the general direction of travel

the shortest path) is updated. Figure 10 shows the spatial distribution of the update count (also called *visits*) in a regular 8-connected grid. Asynchronous SSSP on an unweighted grid has up to 69 visits per node caused by communication-driven reconvergence. Synchronous SSSP on a weighted grid has only up to 7 visits per node, and it can be seen that the pattern of higher visits resembles the differences in weights on the weight map, which demonstrates the effect of application-driven reconvergence. Notably, this type of reconvergence has a considerably smaller impact as the synchronization of packets has a high chance to force message waves to interact. In asynchronous SSSP on a weighted graph, both effects interfere and amplify each other to the maximum number of visits of 190. In comparison, 4-connected grid (not shown) has the maximum visits of 6, 18, and 128 respectively. Horizontal bands visible on asynchronous plots in Figure 10 are caused by the fact that FPGA boards start asynchronously (within 2 ms): if a board has a delayed start relative to its neighbour, some messages have to queue before the inter-FPGA connection; the effect manifests as an unwanted synchronization barrier at that location.

5.3 | Model fitting

Collected experimental data display complex interplays between multiple effects where some topologies show a preference towards asynchronous communication and others clearly benefit from synchronization. We apply the curve fitting technique to these data to build a performance model that can be used as a classification heuristic. The goal is to predict communication mode preference from the properties of the application graph at design time.

Control variables $X = (x_1, \ldots, x_4)$ are selected as shown in Table 3. The graph diameter parameter represents the asymptotic theoretical value, shown in Table 1, because finding the actual graph diameter would require running APSP, which defeats the purpose of the method. Additional data points can be used to improve the model during runtime operation; however, for the purposes of the demonstration only the initial characteristics are used for the fitting.

Since the speedup is defined as a ratio, it may not be well-suited for linear model fitting. Instead, we separately fit performance models for t_s and t_a, and then use those for calculating the speedup. We select quasi-linear model hypothesis capturing the interplay between pairwise combinations of parameters as it keeps partial derivatives constant for model fitting purposes. The hypothesis represents linear-interpolated performance trend for $2^{18} \leq |V| \leq 2^{20}$, which is sufficient for the purpose of the final classifier model. The major drawback of a linear hypothesis is that it may give negative answers that are nonsensical for a performance model. Hypothesis general form is set to:

$$\begin{aligned} t(X) = \alpha_0 \quad & +\alpha_1 x_1 \\ & +\alpha_2 x_2 + \alpha_3 x_1 x_2 \\ & +\alpha_4 x_3 + \alpha_5 x_1 x_3 + \alpha_6 x_2 x_3 \\ & +\alpha_7 x_4 + \alpha_8 x_1 x_4 + \alpha_9 x_2 x_4 + \alpha_{10} x_3 x_4, \end{aligned} \tag{1}$$

where some terms are eliminated using PCA separately for each model. In fact, we considered more parameters than listed $x_1, \ldots, x_4$, including max fanout, fanout variance, edge weight variance, and others, but those got eliminated during the PCA phase. We know from the initial observations, presented earlier in this section, that there is a significant difference in the type of interplays in unweighted graphs compared to weighted graphs, hence we build separate models for each of these cases. Experiments on scale-free graphs use binary weights because of node splitting, and the performance data are not similar enough to fit into the same model as unweighted, hence cannot be used for model fitting in our case (shown as "no model" in the results).

Table 4 presents the models produced by the least-squares model fitting; R^2 values are in the acceptable range, although weighted data are considerably noisier and more difficult to fit. From the methodological point, we try to avoid performing a larger number of initial experiments with the idea that the future performance data can still be used to refine the model during the exploitation phase. Typically, real-world applications work with specific and fairly narrow topology classes that can be studied in advance, therefore it is possible to perform an

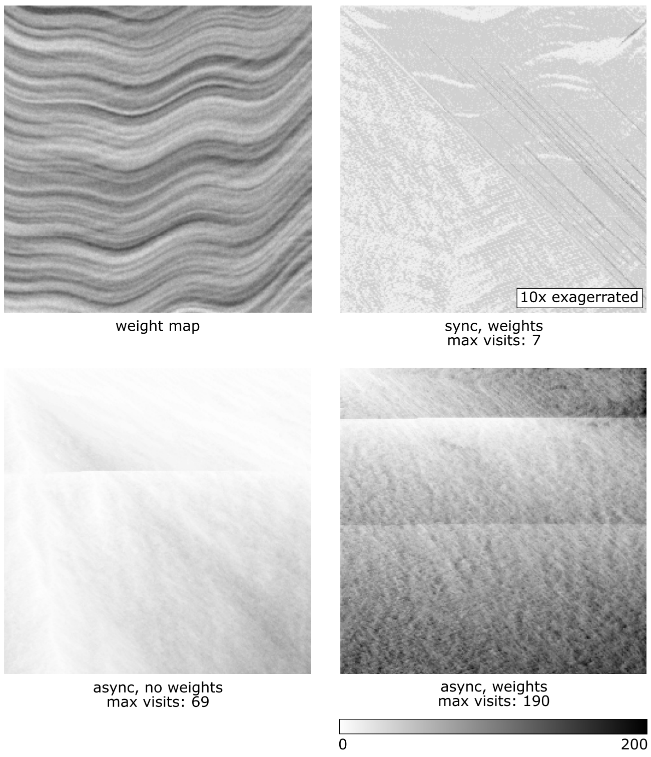

FIGURE 10 Comparison between the effects of reconvergence on the number of state updates (visits) in an 8-connected regular grid. Fewer visits (lighter areas) represent a more efficient computation. The source is in the top-left corner

TABLE 3 Performance model parameters

X	Description
$x_1 = \lvert V\rvert/2^{20}$	Graph size (normalized to 2^{20})
$x_2 = \bar{c}$	Average fanout
$x_3 = D$	Graph diameter
$x_4 = D_w/D$	Reconvergence metric

TABLE 4 Performance model fitting results

Edge weights	Type	Performance model	R^2
Unweighted	Sync	$t_\mathrm{s}(X) = -1.0641 \cdot 10^{-1} + 2.8554 \cdot 10^{-1} \cdot x_1 - 3.8612 \cdot 10^{-3} \cdot x_3 + 1.2842 \cdot 10^{-3} \cdot x_1 x_3 + 3.3268 \cdot 10^{-3} \cdot x_3 x_4$	0.9254
Unweighted	Async	$t_\mathrm{a}(X) = 3.8145 \cdot 10^{-2} - 3.1503 \cdot 10^{-2} \cdot x_1 - 1.0566 \cdot 10^{-2} \cdot x_1 x_2 - 5.8989 \cdot 10^{-2} \cdot x_4 + 2.0707 \cdot 10^{-1} \cdot x_1 x_4 + 1.9132 \cdot 10^{-3} \cdot x_2 x_4$	0.9258
Weighted	Sync	$t_\mathrm{s}(X) = -5.6585 \cdot 10^{-1} + 8.6366 \cdot 10^{-1} \cdot x_1 + 2.4097 \cdot 10^{-2} \cdot x_1 x_2 + 1.0416 \cdot 10^{-3} \cdot x_1 x_3 + 1.1998 \cdot 10^{-3} \cdot x_2 x_3 - 3.9211 \cdot 10^{-3} \cdot x_3 x_4$	0.8760
Weighted	Async	$t_\mathrm{a}(X) = -4.1262 \cdot 10^{-1} + 9.4450 \cdot 10^{-1} \cdot x_1 - 2.7867 \cdot 10^{-2} \cdot x_2 + 4.1067 \cdot 10^{-1} \cdot x_1 x_2 + 4.2625 \cdot 10^{-4} \cdot x_2 x_3 - 1.8203 \cdot x_1 x_4$	0.8131

exploratory experimental study on a small number of graphs and extrapolate configuration strategy to a larger number of use cases from the same category.

Figure 11 compares the speedup values calculated from the models to the measured speedup from Table 2. S_a and S_s are reciprocal to each other, hence their value axes are symmetrical around the value of one on a log scale. Communication preference is therefore represented with the *async* value bars aligned to the right and *sync* aligned to the left. Since the models are intended to be used as a heuristic classifier, we are not interested in the quantitative accuracy of the predicted performance, but only in a correct prediction of the "direction" of the bars (i.e., the model correctly predicts the cases of $t_a > t_s$ and vice versa). Table 5 displays the classification results. Apart from a single instance of model error (linear model returned a negative value), all predictions are correct: the preferences for the synchronization modes are the same as in the experimental observations. By following these model predictions, we are able to achieve the average speedup of 3.22 across the investigated graph topologies.

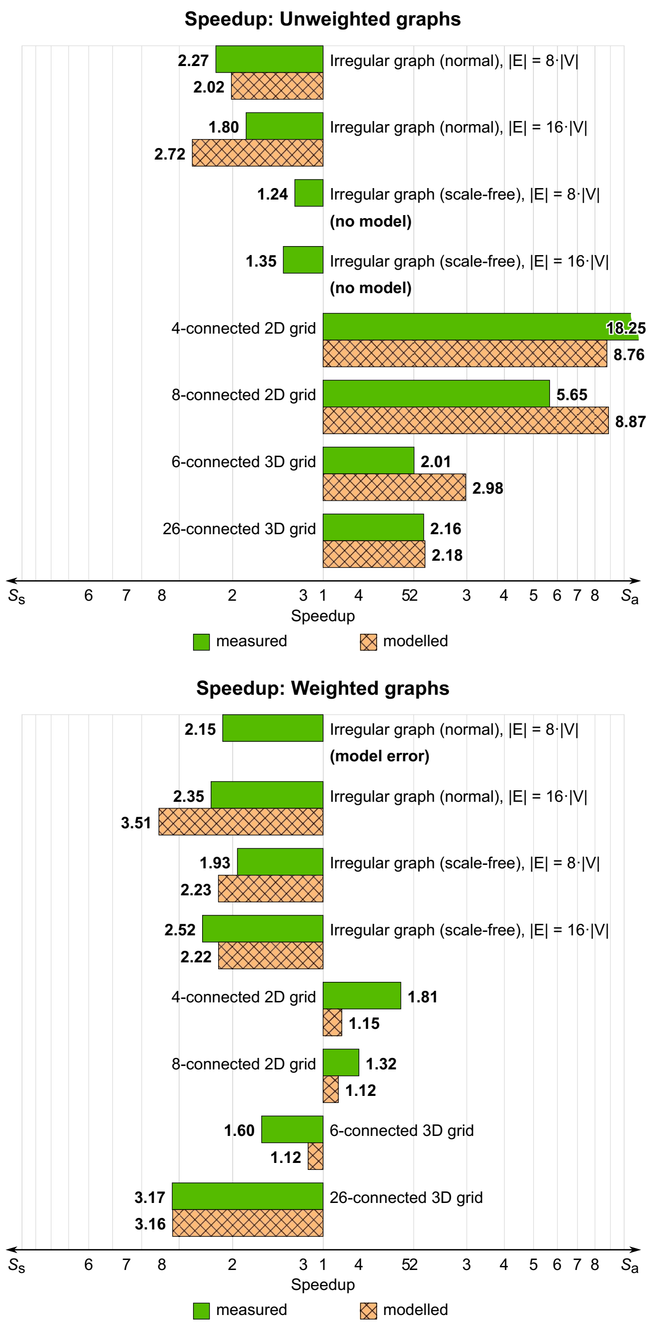

FIGURE 11 Experimental evidence for synchronization preferences in different topologies shown as *async* (on the right) and *sync* (on the left) speedup values and compared to model predictions. We are interested in predicting trends, not actual values. Note that $S_s = 1/S_a$

5.4 | Real-life use case comparison

The developed classifier models have been used to create a decision flow for selecting the POETS prototype platform's mode of communication. This section demonstrates the achieved improvement in computation time in comparison with the same platform's operation without this feature in place. All results are obtained from the experiments on the actual hardware running real-life applications introduced in Section 2.

Figure 12 shows the results for the *seismic raytracing* application. The model classified this application for synchronized communication preference. The results show better performance scaling to larger graph sizes in comparison to the asynchronous mode.

The *drug discovery* use case results, shown in Figure 13, demonstrate the APSP variant of the network traversal application. Fundamentally, APSP calculation is a scaled-up version of SSSP where each node is a source node. For a given network consisting of $|V|$ nodes, scaling up SSSP can be done in the following ways:(1) *time scaling* repeats SSSP calculation $|V|$ times for each source node; this approach is typical for single-core execution; (2) in *data scaling*, each node acts as a source node and also keeps track of $(|V|-1)$ shortest path lengths to all other sources; messages include additional information on which source calculation they belong to, and the number of messages is

TABLE 5 Classifier model prediction results

Topology	Unweighted	Weighted
Normal, $\lvert E\rvert = 8 \cdot \lvert V\rvert$	Sync	*model error*
Normal, $\lvert E\rvert = 16 \cdot \lvert V\rvert$	Sync	Sync
Scale-free, $\lvert E\rvert = 8 \cdot \lvert V\rvert$	*no model*	Sync
Scale-free, $\lvert E\rvert = 16 \cdot \lvert V\rvert$	*no model*	Sync
4-con. 2D grid	Async	Async
8-con. 2D grid	Async	Async
6-con. 3D grid	Async	Sync
26-con. 3D grid	Async	Sync

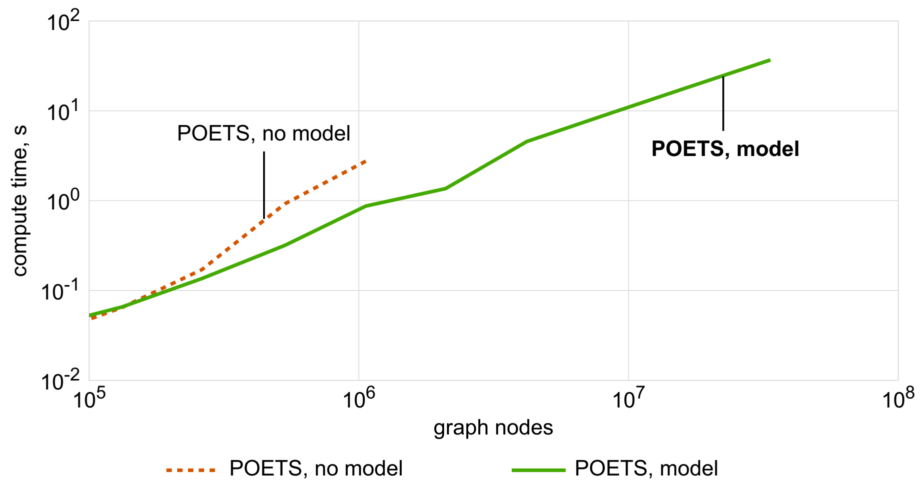

FIGURE 12 Impact of model-driven approach for the *seismic raytracing* use case: single source shortest path (SSSP) on a regular 3D grid with weighted edges

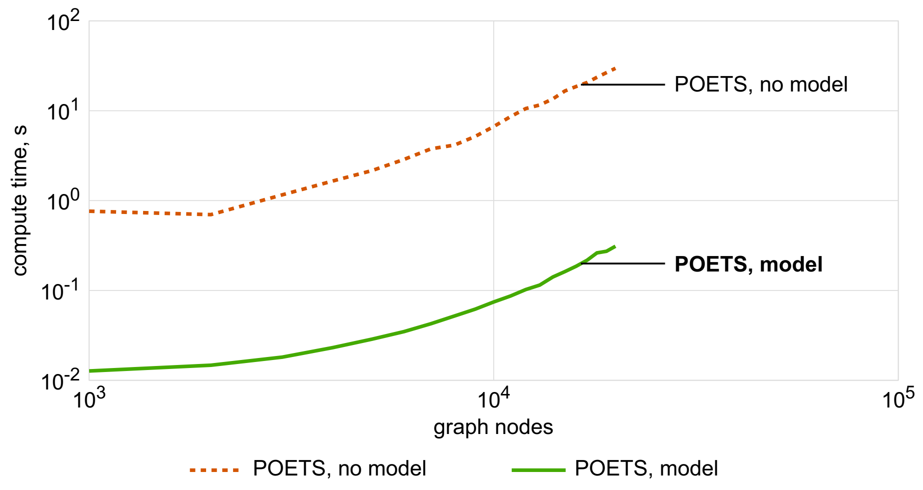

FIGURE 13 Impact of model-driven approach for the *drug discovery* use case: all-pair shortest paths (APSP) on an irregular unweighted graph

increased by $|V|$; (3) *spatial scaling* or *cloning* creates $|V|$ copies of the original network as island graphs in a large meta-network, and each island calculates SSSP for a given source node; the total number of nodes is increased to $|V|^2$, but the amount of memory per node and the traffic density stay the same. It is possible to do a combination of different scaling types: for example, one can use data scaling of a factor $k < |V|$ to utilise all available core memory and/or network throughput and then clone the network $\lceil |V|/k \rceil$ times to cover all source nodes.

Asynchronous implementation of APSP turned out to be considerably less efficient than synchronized, although the rate of scaling is similar. The main reason is that asynchronous APSP requires 16 times more device memory and 16 times more message data as discussed earlier. Adding the fact that the message traffic comes in unrestricted bursts, asynchronous APSP appears on average 80–100 times slower than synchronized.

6 | CONCLUSIONS

This paper described an event-based programming model for graph analytic applications and proposed a methodological case for model-based decisions on communication protocols. From an extensive experimental exploration of the design space, we determined that the major contributing factors from the graph topology are the graph size, average fanout, graph diameter, and reconvergence. These factors can be estimated at design-time from the static graph analysis. We applied the least-squares fitting technique to create a classifier model based on these factors and achieved the improvement in computation time up to 18.25 times, or 3.22 times on average in comparison to the results without the model support.

The study was performed in the context of POETS: a massively parallel prototype compute platform based on FPGAs. A high-level event-based API formed an additional abstraction level and provided control over the crucial features like node-to-core mapping, memory distribution, and communication model. Hardware-implemented termination detection experimentally proved to be fast and efficient, so it can also serve as a synchronization barrier. Therefore, globally synchronized communication protocols were confirmed as a viable alternative to asynchronous communications and were shown to improve computation time in a number of graph topologies, including real-life use cases from the fields of biology and geology.

We are currently undertaking the process of upgrading the POETS platform to the next-generation FPGAs (Intel Stratix 10) and faster inter-board connections. The future work will also include the investigation of whether the performance models still produce valid predictions for the upgraded system, and if not, generalise the modelling methodology to support multiple platforms.

ACKNOWLEDGEMENT

This work is supported by EPSRC/UK as a part of the POETS project EP/N031768/1.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Ashur Rafiev https://orcid.org/0000-0002-7387-5970
Alex Yakovlev https://orcid.org/0000-0003-0826-9330
Matthew F. Naylor https://orcid.org/0000-0001-9827-8497
Simon W. Moore https://orcid.org/0000-0002-2806-495X
David B. Thomas https://orcid.org/0000-0002-9671-0917
Graeme M. Bragg https://orcid.org/0000-0002-5201-7977
Mark L. Vousden https://orcid.org/0000-0002-6552-5831

REFERENCES

1. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature. 393(6684), 440 (1998)
2. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science. 286(5439), 509–512 (1999)
3. Barabási, A.L., et al.: Network Science. Cambridge university press (2016)
4. Satish, N., et al.: Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In: Proc. To SC'12, pp. 1–11. IEEE (2012)
5. Mokhov, A., et al.: Language and hardware acceleration backend for graph processing. In: Languages, Design Methods, and Tools for Electronic System Design, pp. 71–88. Springer (2019)
6. Brown, A., et al.: Distributed Event-Based Computing. IOS Press (2018). https://gow.epsrc.ukri.org/NGBOViewGrant.aspx?GrantRef=EP/N031768/1
7. POETS Project Website, (2021). https://poets-project.org
8. RISC-V: The Free and Open RISC Instruction Set Architecture (2021). https://riscv.org
9. Naylor, M., et al.: Termination detection for fine-grained message-passing architectures. In: Proc. To ASAP. IEEE (2020)
10. Kitano, H.: A robustness-based approach to systems-oriented drug design. Nat. Rev. Drug Discov. 6(3), 202–210 (2007)
11. E-Therapeutics Website, (2021). https://www.etherapeutics.co.uk
12. Young, M.P., Zimmer, S., Whitmore, A.V.: Chapter 3. Drug molecules and biology: network and systems aspects. In: RSC Drug Discovery, pp. 32–49. Royal Society of Chemistry (2012)
13. Malony, A.D., et al.: Towards scaling parallel seismic raytracing. In: CSE/EUC/DCABES 2016, pp. 225–233. IEEE (2016)
14. Monil, M.A.H., et al.: A scalable parallel seismic raytracing system. In: Proc. To PDP, pp. 204–213. IEEE (2018)
15. Nollet, G.: A Breviary of Seismic Tomography: Imaging the Interior of the Earth and Sun. Cambridge University Press, (2008). Cambridge University Press
16. Quinn, M.J., Deo, N.: Parallel graph algorithms. ACM Comput. Surv. 16(3), 319–348 (1984)
17. Bader, D.A., Madduri, K.: Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In: 2006 International Conference on Parallel Processing (ICPP'06), pp. 523–530. IEEE (2006)
18. Mastrostefano, E., Bernaschi, M.: Efficient breadth first search on multi-gpu systems. J. Parallel Distr. Comput. 73(9), 1292–1305 (2013)
19. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. Sci. Program. 21(3-4), 137–148 (2013)
20. Bernaschi, M., et al.: Enhanced gpu-based distributed breadth first search. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, pp. 1–8 (2015)
21. Yoo, A., et al.: A scalable distributed parallel breadth-first search algorithm on bluegene/l. In: SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 25, IEEE, (2005)
22. Pearce, R., Gokhale, M., Amato, N.M.: Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 549–559. IEEE (2014)
23. Pearce, R., Gokhale, M., Amato, N.M.: Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2010)
24. Pearce, R., Gokhale, M., Amato, N.M.: Scaling techniques for massive scale-free graphs in distributed (external) memory. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 825–836. IEEE (2013)
25. Wang, Y., et al.: Gunrock: gpu graph analytics. ACM Trans. Parallel Comput. (TOPC). 4(1), 1–49 (2017)
26. Xu, C., et al.: Graph processing services on energy-efficient hardware accelerator. In: 2018 IEEE International Conference on Web Services (ICWS), pp. 274–281. IEEE (2018)
27. Lei, G., et al.: Torusbfs: a novel message-passing parallel breadth-first search architecture on fpgas. EngineerSci. Technol. Int. J. 5(5), 10 (2015)
28. Finnerty, E., et al.: Dr. bfs: data centric breadth-first search on fpgas. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2019)
29. Plana, L.A., et al.: A GALS infrastructure for a massively parallel multiprocessor. IEEE Des.Test. Comput. 24(5), 454–463 (2007)
30. Witlox, B.R.T.M., et al.: Performance analysis of dataflow architectures using timed coloured petri nets. In: Hardware Design and Petri Nets. Springer (2000)
31. Xie, A., Beerel, P.A.: Performance analysis of asynchronous circuits and systems using stochastic timed petri nets. In: Hardware Design and Petri Nets. Springer (2000)
32. Rafiev, A., et al.: Practical Distributed Implementation of Very Large Scale Petri Net Simulations. Petri Nets and Other Models of Concurrency (ToPNoC) (2022). (in print)
33. Rencher, A.C., Christensen, W.F.: Methods of Multivariate Analysis. Wiley, Hoboken, NJ (2012)
34. Jolliffe, I.T., Cadima, J.: Principal component analysis: a review and recent developments. Philos Trans A Math Phys Eng Sci. 374(2065) (2016)
35. Darken, C., Chang, J., Moody, J.: Learning rate schedules for faster stochastic gradient search. In: Neural Networks for Signal Processing II Proceedings of the 1992, pp. 3–12. IEEE Workshop (1992)
36. Aalsaud, A., et al.: Power–aware Performance Adaptation of Concurrent Applications in Heterogeneous Many-Core Systems. ISLPED 16, pp. 368–373. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2934583.2934612

37. Naylor, M., Moore, S.W., Thomas, D.: Tinsel: a manythread overlay for FPGA clusters. In: Proc. To FPL (2019)
38. Dijkstra, E.W.: Shmuel Safra's version of termination detection. EWD998 (1987). https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF
39. Karypis, G., Kumar, V.: METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. University of Minnesota (1998)

How to cite this article: Rafiev, A., et al.: Synchronization in graph analysis algorithms on the Partially Ordered Event-Triggered Systems many-core architecture. IET Comput. Digit. Tech. 16(2-3), 71–88 (2022). https://doi.org/10.1049/cdt2.12041