Termination detection for fine-grained message-passing architectures

Matthew Naylor[1], Simon W. Moore[1], Andrey Mokhov[2], David Thomas[3], Jonathan R. Beaumont[3], Shane Fleming[4], A. Theodore Markettos[1], Thomas Bytheway[1], and Andrew Brown[5]
[1]University of Cambridge, [2]Newcastle University and Jane Street,
[3]Imperial College London, [4]Microsoft Research, [5]University of Southampton

***Abstract*—Barrier primitives provided by standard parallel programming APIs are the primary means by which applications implement global synchronisation. Typically these primitives are *fully-committed* to synchronisation in the sense that, once a barrier is entered, synchronisation is the only way out. For message-passing applications, this raises the question of what happens when a message arrives at a thread that already resides in a barrier. Without a satisfactory answer, barriers do not interact with message-passing in any useful way.**

In this paper, we propose a new *refutable* barrier primitive that combines with message-passing to form a simple, expressive, efficient, well-defined API. It has a clear semantics based on termination detection, and supports the development of both globally-synchronous and asynchronous parallel applications.

To evaluate the new primitive, we implement it in a prototype large-scale message-passing machine with 49,152 RISC-V threads distributed over 48 FPGAs. We show that *hardware support* for the primitive leads to a highly-efficient implementation, capable of synchronisation rates that are an order-of-magnitude higher than what is achievable in software. Using the primitive, we implement synchronous and asynchronous versions of a range of applications, observing that each version can have significant advantages over the other, depending on the application. Therefore, a barrier primitive supporting both styles can greatly assist the development of parallel programs.

I. Introduction

Message-passing and global synchronisation are powerful abstractions in parallel computing, especially when used in combination [1]. Both are supported by MPI, the standard API for message-passing applications in HPC domains. However, versions 1 and 2 of the MPI standard do not define any useful form of interaction between the two. In particular, a thread entering a synchronisation barrier becomes blocked and unable to react to further incoming messages. The major drawback of this approach is that threads must know how many messages they are going to receive *before* entering, but in general this is not easily predictable (without introducing overheads) because the decision of whether to send or not is made by the sender, not the receiver. Furthermore, if a thread does enter a barrier before receiving all the messages destined for it, the system is likely to deadlock.

To overcome this problem, MPI 3 introduces a new *non-blocking* barrier primitive which allows threads to remain active while present in a barrier. However, this leads to a complex semantics, as well as a number of limitations:

- Any attempt by a thread to send a message after entering a non-blocking barrier can lead to a race between the barrier and the send operation: it is not defined which will complete first.
- Since a non-blocking barrier cannot be cancelled, a thread cannot usefully send after entering, without introducing non-deterministic behaviour. This means there are restrictions on the kind of asynchronous computation that can occur within each synchronous time-step.
- To achieve a useful, deterministic interaction between message-passing and non-blocking barriers, *synchronous send* operations must be used. A synchronous send is one where the receipt of the message is implicitly acknowledged by the receiver. This allows a message to be sent by a thread not residing in a barrier to one that is, while ensuring the message is received before the barrier can possibly complete. However, synchronous sends are expensive, requiring a two-way exchange. For *fine-grained* parallel applications (consisting of large numbers of small processes), synchronous sends can increase communication traffic significantly, harming performance.

Whether blocking or non-blocking, MPI barriers are *fully-committed* to synchonisation in the sense that once a barrier is entered, synchronisation is the only way out. In this paper, we explore a different path and propose a *refutable* barrier primitive with a number of attractive properties: (1) it has a simple semantics based on termination detection [2]; (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 time-step of a parallel application. Our contributions are as follows.

- We present a hardware extension to an asynchronous message-passing machine that *detects termination*, i.e. the situation in which every thread has indicated a desire to terminate and there are no undelivered messages in-flight. It enables a globally-synchronous execution model in which a new time step may start every time termination of the previous time step is detected. It is also useful in asynchronous applications, to detect convergence. To our knowledge, custom hardware support for termination detection has not been explored before.
- We measure the hardware cost and performance of termination detection in a message-passing machine consisting of 49,152 RISC-V threads distributed over 48 FPGAs.

This includes a quantitative comparison of hardware and software approaches to termination detection.

- We show how our barrier primitive can be exploited by a high-level vertex-centric software API, generalising Google's Pregel model [3] to support development of both synchronous *and* asynchronous applications at a high level of abstraction.
- We run a range of standard vertex-centric benchmarks, demonstrating the strengths of the approach for fine-grained parallel applications, such as graph processing and spiking neural networks, where the cost of sending additional messages to assist synchronisation can be high.

II. Background

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 HPC domains, 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 (Partial Ordered Event Triggered Systems [4]), which forms the wider context for the work described in this paper. On the project, we have constructed a research platform consisting of a 48-FPGA cluster and a manycore RISC-V overlay called Tinsel [5] programmed on top. This serves both as a rapid prototyping environment for computer architecture research and, for certain applications, a genuine hardware accelerator. For example, in previous work [5] we have shown the potential for significant performance improvements over a standard Xeon cluster for HPC applications written using the vertex-centric programming model popularised by Google's Pregel [3]. Below, we outline the design of the research platform, and its asynchronous message-passing primitives, before presenting our termination-detection extension in the next section.

A. Research platform

The Tinsel **overlay** has 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 cores sharing an FPU, a data cache, and a mailbox:

- The **core** is heavily multithreaded, supporting 16 hardware threads by default. As a result, it can tolerate several cycles of latency, e.g. arising from deeply-pipelined FPGA floating-point operations, or cache misses that lead to off-chip memory accesses. Threads are barrel-scheduled (a context switch is performed on every clock cycle) so pipeline hazards are easily avoided, leading to a small, fast design.
- The **mailbox** contains a memory-mapped scratchpad storing up to 64KB 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 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 mailbox API is outlined in Section II-B.
- The **cache** is a non-blocking 16-way set-associative write-back cache that optimises access to the large off-chip memories available on each FPGA board. It is 128KB in size and is partitioned by thread id so that cache lines are not shared between threads. This means there are no hazards in the cache pipeline, which again leads to a small, fast design.
- The **FPU** supports IEEE single-precision floating-point operations. On the Stratix V FPGAs we are using, these are expensive in both area and latency, which we mitigate through sharing and multithreading respectively.

A single-FPGA view of the overlay is depicted in Figure 1. On the DE5-Net FPGA board, the overlay contains 64 RV32IMF cores (1,024 hardware threads), clocks at 240MHz, and utilises 67% of the board's Stratix V logic blocks.

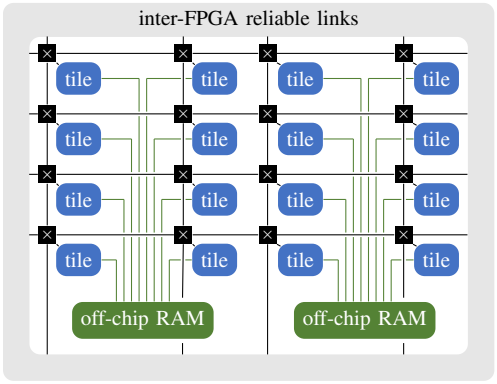


Fig. 1. Default configuration of our overlay on a single DE5-Net FPGA board. Mailboxes in tiles are connected together using dimension-ordered routers to form a NoC. Inter-FPGA links are connected to the NoC rim. A separate network is used to connect caches in tiles to off-chip memories. Each off-chip RAM component contains a DDR3 controller and two QDRII+ controllers.

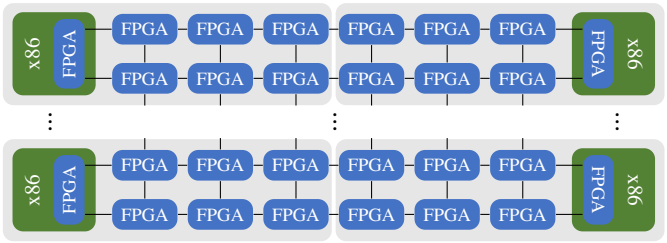


Fig. 2. Our FPGA cluster is composed of multiple *boxes* (shown in light gray). Each box contains an x86 server and 7× DE5-Net FPGA boards. One FPGA board in each box serves as a bridge between the x86 server (PCI Express) and the FPGA network (10G SFP+). The x86 servers provide command-and-control facilities, such as data injection and extraction. The full cluster contains a 2×4 mesh of boxes, and a 6×8 mesh of worker FPGAs.

The FPGA **cluster** comprises a 6×8 grid of DE5-Net boards connected together using 10G reliable links, as shown in Figure 2. The overlay distributes naturally over this cluster to yield a 3,072 core system (49,152 hardware threads), where any thread can send messages to any other thread.

B. Asynchronous message-passing API

Threads send and receive messages using custom RISC-V CSRs. These raw CSR accesses are abstracted by a very thin software API, which we outline below.

To send a message residing in the mailbox scratchpad, a thread first ensures that the network has capacity by calling

```
bool tinselCanSend();
```

and if the result is true, the thread can call

```
void tinselSend(uint32_t dest, void* msg);
```

where `dest` is a global thread identifier, and `msg` is a message-aligned address in the scratchpad. The message is not guaranteed to have left the mailbox until `tinselCanSend()` returns true again, at which point data pointed to by `msg` can safely be mutated, e.g. by writing a new message.

To receive a message, a thread must first allocate space in the scratchpad for an incoming message to be stored. Allocating space can be viewed as transferring ownership of that space from the software to the hardware. This is done by a call to

```
void tinselAlloc(void* msg);
```

where `msg` is a message-aligned address in the scratchpad. Space for multiple messages can be allocated in this way, creating a receive buffer of the desired size. Now, when a thread wishes to receive a message it can call

```
bool tinselCanRecv();
```

to see if a message is available and, if so, receive it by calling

```
void* tinselRecv();
```

which returns a pointer to the received message. Receiving a message can be viewed as transferring ownership of the space it occupies from the hardware back to the software.

A key property of this message-passing interface is that it is *non-blocking*. This is necessary for asynchronous applications to avoid deadlock and meet the *consumption assumption* [6], i.e. that threads are always willing to receive, and do not block on a send operation that prevents them from doing so. In return, the hardware guarantees delivery of all messages sent.

III. Hardware termination-detection

We now extend the hardware with a new refutable barrier primitive, with a semantics based on termination detection. This is achieved through the addition of single new custom RISC-V CSR. As above, the CSR access is abstracted by a thin API:

```
uint32_t tinselIdle(bool vote);
```

When called by a thread, this function blocks until 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 `tinselIdle` 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 > 1 denotes that all callers voted true. The voting mechanism is a simple form of aggregation that is useful for detecting termination in globally-synchronous applications, 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. After that, in Section IV, we look at the influence of this primitive on higher-level APIs.

A. Safra's algorithm

Safra's algorithm, as presented by Dijkstra [2], 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 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 basic operation of the algorithm is as follows:

1) Each machine maintains a local *count* of the number of messages it has sent minus the number it has received.
2) A termination *token* containing an accumulator, initially zero, is passed from machine to machine in a ring pattern.
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.
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:

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.
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:

7) When a machine forwards a token, it whitens itself.

B. Choosing the granularity

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 on a call to `tinselIdle`. 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.

Therefore, we implement the termination-detection algorithm in hardware at the granularity of FPGAs rather than threads, with the aim of achieving greater scalability. 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 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 `tinselIdle`. 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.

C. Scalable topology

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, i.e. $\mathcal{O}(L \cdot N)$. 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.

In our cluster, we use one of the FPGA bridge boards (see the FPGAs connected to the x86 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 $\mathcal{O}(4 \cdot L \cdot \sqrt{N})$ 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.

D. Barrier release

Safra's algorithm is only concerned with a *single* machine in the system detecting global termination. To implement the `tinselIdle`, 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 `tinselIdle` 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 `tinselIdle`. To remedy this, we disable the sending of messages when releasing the calls to `tinselIdle`, 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.

IV. High-level API

Termination detection is a key component of our high-level API that maps arbitrary *task graphs* onto the overlay. Behaviours of vertices in the graph are defined by *event handlers* that update the *vertex state* when a particular event occurs, e.g. when a message arrives on an incoming edge, or the network is ready to send a new message, or termination is detected. It is similar to the vertex-centric paradigm [3, 7], but supports both synchronous and asynchronous execution.

Using the API, vertex behaviour is defined by inheriting from the `PVertex` class:

```
template <typename S, typename E, typename M>
  struct PVertex {
    // Vertex state
    S* s;
    PPin* readyToSend;

    // Event handlers
    void init();
    void send(M* msg);
    void recv(M* msg, E* edge);
    bool step();
    bool finish(M* msg);
  };
```

Fig. 3. Essential structure of a task/vertex. It is parameterised by the task state type `S`, the edge weight type `E`, and the message type `M`.

Each vertex has access to local state `s`, and a `readyToSend` field whose value is one of:

- `No` – the vertex doesn't want to send.
- `Pin(p)` – the vertex wants to send on pin *p*.
- `HostPin` – the vertex wants to send to the host.

A pin is an set of outgoing edges, and sending a message on a pin means sending a message along all edges in the set. A vertex can have any number of pins. Vertices should initialise `*readyToSend` in the `init` handler, which runs once for every vertex when the application starts. After that, the other event handlers come into play:

- **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. The destination is deduced from the value of `*readyToSend` immediately before the `send` handler is called.
- **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 edge weight is passed to the `recv` handler rather than the `send` handler because it is associated with a particular edge, not a pin capturing multiple edges.
- **Step handler** The `step` handler is called when termination is detected, i.e. 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`.

To illustrate the API, Figure 4 shows an asynchronous solution to the single-source shortest paths problem. Each vertex maintains an `int` representing the shortest known path to it (initially the largest positive integer), and a read-only `bool` indicating whether or not it is the source vertex. When the application starts, only the source vertex requests to send, but this triggers further iterative sending until the shortest paths to the all vertices have been determined. Finally, when termination is detected, each vertex sends its distance back to the host. In this example, a single pin (pin 0) on each vertex, holding the set of neighbouring edges for that vertex, is sufficient to solve the problem.

```
// Vertex state
struct SSSPState {
  // Is this the source vertex?
  bool isSource;
  // The shortest known distance to this vertex
  int dist;
};

// Vertex behaviour
struct SSSPVertex : PVertex<SSSPState,int,int> {
  void init() {
    *readyToSend = s->isSource ? Pin(0) : No;
  }
  void send(int* msg) {
    *msg = s->dist;
    *readyToSend = No;
  }
  void recv(int* dist, int* weight) {
    int newDist = *dist + *weight;
    if (newDist < s->dist) {
      s->dist = newDist;
      *readyToSend = Pin(0);
    }
  }
  bool step() { return false; }
  bool finish(int* msg) {
    *msg = s->dist;
    return true;
  }
};
```

Fig. 4. Asynchronous single-source shortest paths using our high-level API.

	Before	After	Difference
Mean area (ALMs)	154,263	155,789	+1,526
Mean area (%)	65.7	66.3	+0.6
Mean Fmax (MHz)	226 (±8)	223 (±14)	-3
Best Fmax (MHz)	240	242	+2

Fig. 5. FPGA synthesis results for our overlay before and after enabling hardware termination-detection. Results were taken from a batch of 16 synthesis runs using Quartus Design Space Explorer.

V. Evaluation

A. Synthesis results

Figure 5 shows the FPGA synthesis results for our overlay with and without hardware termination-detection enabled. To counter natural variations in synthesis quality, we take averages from 16 separate synthesis runs using Quartus Design Space Explorer. The results show that hardware termination-detection requires only a small amount of logic (0.6% of the FPGA) and has no significant impact on the maximum clocking frequency of the design.

B. Round-trip time

The key factor limiting the performance of our termination detection procedure is the *round-trip time*. That is, the time taken for the master to send a token to every FPGA and receive back all acknowledgements. With the master at the origin of the FPGA mesh, the longest round-trip time (*rtt*) is via the FPGA in the far corner, which we estimate as

$$rtt = 2 * L * (x + y - 1)$$

where L is the inter-FPGA hop latency, and x and y are the dimensions of the FPGA mesh. We have measured the single-hop latency L, including the time to pass through our reliability layer, as 150 cycles at 240MHz, i.e. 625ns.

Figure 6 shows both the estimated and measured round-trip performance of the hardware for a range of FPGA mesh sizes. The measured time closely follows the estimated time (although the estimated time is slightly optimistic in assuming

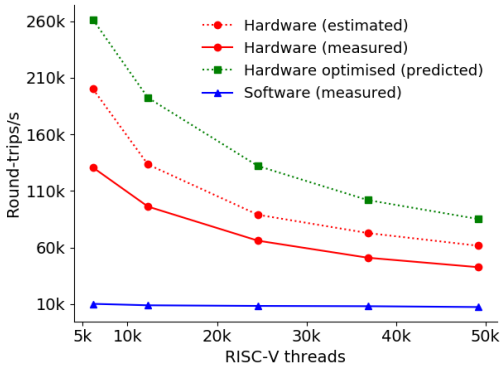


Fig. 6. Number of *round-trips* per second achieved using termination detection in hardware and an asynchronous clock tree in software. A round-trip consists of a master sending a token to every FPGA (hardware) or thread (software), and receiving back all acknowledgements.

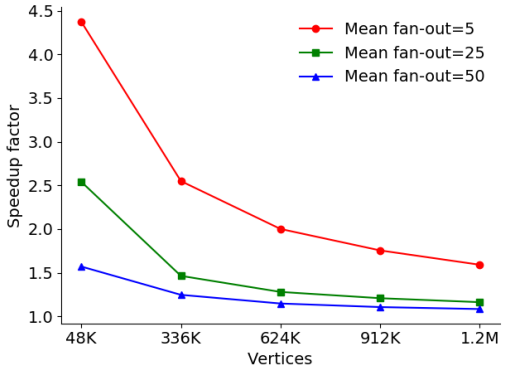


Fig. 7. Speedup of hardware termination-detection (measured) over software termination-detection (predicted) in the synchronous heat transfer application applied to a random graphs, and running on the 48-FPGA cluster.

that all tokens pass over a link simultaneously, rather than one-at-a-time). The plot shows that the round-trip performance of hardware termination-detection is 6 – 13 times faster than that of an asynchronous clock tree implemented in software. As discussed in Section III-C, the performance of the hardware could be doubled by placing the master at the centre of the mesh rather than the origin. The plot includes the predicted round-trip performance of this potential optimisation.

C. Benchmark applications and graphs

In the remainder of this section we evaluate termination-detection using benchmark applications written using our high-level vertex-centric API from Section IV. The five applications are: **PR** (page rank) for ranking webpages [8]; **SSSP** (single-source shortest paths) for weighted graphs; **MSSP** (multiple-source shortest paths) for unweighted graphs; **SNN** (spiking neural network) simulation using the Izhikevich model [9]; and **HT** (heat transfer) simulation using Newton's law of cooling.

All of the applications operate on arbitrary input graphs. We use a geometric random graph generator to produce graphs for benchmarking purposes. This generator gives us control over the amount of locality in the graphs, so we can explore the limits of the communication subsystem. To counter variations in run-time performance, we generate five versions of each graph (varying the random seed) and average the results.

We have three separate implementations of each application:

1) A **synchronous** version, in which each vertex sends at most one message to each neighbour in each time step. The beginning of a new time step is determined using hardware termination detection.
2) An **asynchronous** version, in which there is no global synchronisation barrier. For all applications except SSSP, we use a GALS approach (globally asynchronous, locally synchronous) whereby each vertex does not proceed to its next time step until it receives messages for the current time step from all its neighbours. This means that each vertex may be at most one time step ahead of any neighbour, but may be up to n time steps ahead of any vertex that is n hops away. For the SSSP application, we use a fully asynchronous approach where a vertex v sends a distance to its neighbours as soon as a shorter path to v has been found. Termination detection is used to determine completion in each asynchronous version.
3) A **single-threaded x86** version, running on an Intel i9-7940X PC. This is only intended as a simple baseline; here, we do not compare the performance of our research platform against conventional compute clusters (some such comparisons can be found in a previous paper [5]). For the PR application, we reuse the implementation from the GAP benchmark suite [11]. For the other applications, we use our own optimised C++ implementations.

All the graphs and application implementations that we use are available in the data package accompanying this paper [10].

D. Hardware versus software termination-detection

Having measured the round-trip times using both hardware (rtt_h) and software (rtt_s) approaches, we can predict the performance difference between using hardware and software termination-detection in an actual application. First, we take the run-time t of the application using hardware termination-detection, count the number of times n that termination is detected, and then subtract $3 \times n \times rtt_h$ from t and add $3 \times n \times rtt_s$ (to account for the three phases of our termination detection procedure).

Figure 7 shows such a comparison for the synchronous heat transfer application. The benefit of hardware termination-detection is significant for small graphs but reduces as the amount of compute per time step increases, either by mapping more vertices to each thread, or by increasing the fan-out and hence the number of messages processed by each thread. Vice-versa, if the amount of compute per time step decreases, for example by adding more cores, then we would expect the benefit of hardware termination-detection to increase.

E. Synchronous versus asynchronous execution

Figure 8 compares the runtime performances and message counts of synchronous and asynchronous implementations

of each application. All applications have been applied to geometric random graphs containing 6M vertices and 600M edges. Our main observations from these comparisons are outlined below.

In **PR** and **HT**, vertices exchange messages with their neighbours unconditionally on each time step. Comparing the synchronous and asynchronous versions, the number of messages sent is identical, but the runtime performance of the asynchronous version is slightly worse. This is because an additional buffer is required at each vertex to handle messages from a neighbour that could be one time step ahead.

In **MSSP** and **SNN**, vertices do not need to exchange information on every time-step, e.g. when the set of reaching vertices does not change (in MSSP), or when a neuron doesn't fire (in SNN). The synchronous version exploits this fact to significantly reduce the number of messages that need to be sent. The asynchronous version needs to send a far greater number of messages because it must maintain the invariant that neighbouring vertices are within one time step of each other. This is particularly problematic in SNN where neurons fire relatively rarely, and the directed neural graph must be augmented with additional zero-weight back-edges to maintain local synchronisation. This is a key strength of our barrier primitive: it avoids communication overheads when vertices interact optionally and dynamically on each time step.

In **SSSP**, each vertex in the synchronous version blocks until all vertices that wish to propagate new distances for the current time step have done so. In the asynchronous version, vertices can receive and propagate distances immediately, without blocking, leading to better runtime performance.

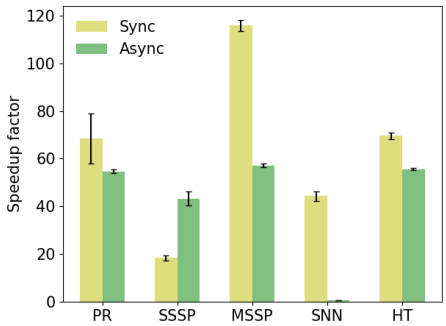
F. Worst-case versus average-case complexity

It is worth noting that the worst-case complexity of our asynchronous SSSP application is exponential. Indeed, if one assumes that the order of message arrival in the communication network is arbitrary, then it is possible to exhibit a graph comprising N identical components and a sequence of message arrivals such that the k-th component will receive 2^k distance updates. However, while the order of message arrival is not deterministic, it is also not entirely arbitrary: the closer the two nodes are topologically, the faster is the expected message delivery time from one to the other. This makes the *average-case* complexity of the asynchronous SSSP running on our platform polynomial.

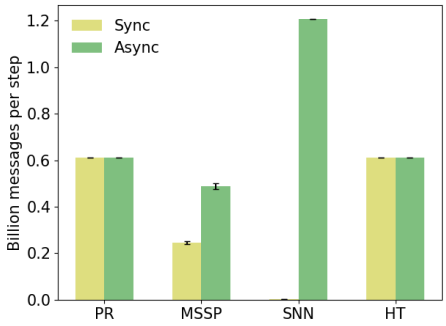
On the other hand, the worst-case complexity of our synchronous SSSP algorithm is linear in the size of the graph, yet it has worse runtime performance in the experiments we've conducted so far. This highlights the importance of studying the statistical properties of asynchronous algorithms, instead of relying on classic worst-case complexity bounds.

VI. Related work

SpiNNaker is a million ARM-core machine constructed at the University of Manchester [12]. It has been a great source of inspiration for our work. Similar to our overlay, it consists of a large number of small cores communicating purely by message-passing. Unlike our overlay, it is designed specifically for spiking neural network simulation and has no support for global synchronisation. Instead, SpiNNaker employs a per-chip real-time clock to advance the millisecond time-step of the neural simulation. The clocks on each chip are not synchronised and are therefore subject to drift. Furthermore, there is no guarantee that messages will reach their destinations before the end of each time step, and no guarantee that messages will not be dropped if the communication fabric is overwhelmed. As a result, emergent behaviours from the hardware can affect the correct operation of software in subtle ways, making it challenging to reason about the correctness of the simulation.



(a) Performance relative to single-threaded x86 baseline.



(b) Total number of messages sent per time step. This metric is not applicable for the fully-asynchronous SSSP application.

Fig. 8. Comparison of runtime performance and number of messages sent for synchronous and asynchronous implementations of each application.

A number of other architectures consisting of large numbers of small cores communicating explicitly have been proposed over the years. Some of the most notable are: RAMP Blue [13] (which is able to run off-the-shelf parallel applications expressed in the Unified Parallel C framework), GRVI Phalanx [14] (which packs a remarkable 1,680 RISC-V cores on a single Xilinx XCVU9P FPGA), and Adapteva Epiphany [15, 16] (which achieves high floating-point performance-per-watt). The trend in all these architectures is to support the partitioned global address space (PGAS) model of computation, where cores communicate by performing remote loads and stores to the memories of other cores. Compared to a pure message-passing machine, where a message is transferred in a single phase, PGAS can require multiple phases. For example,

sending an optional message in a synchronous application could require a request, a response, and then another request to implement a remote store where the address is specified dynamically by the receiver (three phase push). Another trend in these architectures is to provide barrier synchronisation as the means to support global synchronisation.

MPI (Message Passing Interface) is a standard API for writing parallel applications, and is supported by a wide range of architectures including the majority of supercomputers over the past few decades [17]. Version 3 of the standard introduces a *non-blocking* `MPI_Ibarrier()` function which, combined with *synchronous* sends (i.e. two-way send operations with an implicit acknowledgement), can express a limited form of termination detection [18]. In particular, it requires that threads, within a time step, do not decide to send a new message as a result of receiving a message (non-blocking barriers cannot be cancelled). Therefore, this approach would not be sufficient to capture our asynchronous SSSP application, which involves an fully-asynchronous computation occurring before the barrier. Furthermore, while it would be sufficient to capture our synchronous SNN application, the overhead of synchronous sends means that the approach is unlikely to be as efficient as true termination detection. MPI applications requiring true termination detection tend to implement ad-hoc versions of classic algorithms on top of MPI [19, 20]. Hardware support for the *blocking* `MPI_barrier()` function on an FPGA cluster has been explored by Gao [21].

VII. Conclusions

Message passing and global synchronisation are key concepts in parallel computing. In this paper, we have seen that termination detection is a powerful way to combine the two, through our proposed refutable barrier primitive. This has a number of advantages over standard barrier primitives provided by mainstream message-passing APIs such as MPI, with respect to ease-of-use, expressiveness, and efficiency. In particular, it allows an arbitrary asynchronous computation to occur within each synchronous time-step, and it avoids communication overheads that are often needed to assist synchronisation in applications with dynamically-changing communication patterns.

We have extended an asynchronous message-passing machine with hardware support for termination detection, adapting Safra's classic algorithm for improved performance on a multi-chip cluster. This requires very few hardware resources and offers significant speed-ups in cases where synchronisation time is the bottleneck (we measured a 6-13x improvement in synchronisation rate over software, with potential for twice that). For applications with large parallel slack, i.e. containing much more parallelism than the physical parallelism available, sufficient performance may be achievable from a pure software implementation of termination detection, if synchronisation is sufficiently infrequent. But such cases will become less common as machines scale up to ever-larger numbers of cores.

Finally, we have explored some of the trade-offs between asynchronous and globally-synchronous algorithms running on a fine-grained message-passing architecture. The results show that each approach can be more effective than the other, depending on the workload. Therefore, developers of message-passing APIs and architectures should consider support for *both* – and termination detection is a powerful way to do so.

A. Future work

Aggregators are a useful extension of barriers to support global communication [3]: values provided by each thread on entry to a barrier are reduced by an aggregator to a single value that is distributed to all threads on barrier release. In this paper, we have only considered a single aggregator for boolean conjunction. Furthermore, standard barrier primitives often have a configurable granularity: they can apply globally over all threads, or locally to a specified *group* of threads. In future, it would be desirable to extend our barrier primitive with support for thread groups and a greater range of aggregators.

VIII. Acknowledgments

Thanks to He Li and Mayhar Shahsavari. This work was supported by EPSRC grant EP/N031768/1 (POETS project).

References

[1] L. G. Valiant. *A bridging model for parallel computation*, Communications of the ACM 33(8), 1990.
[2] E. W. Dijkstra, *Shmuel Safra's version of termination detection*, EWD998, Online, accessed: 3 Feb 2020. Available: https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF.
[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: A System for Large-scale Graph Processing*, SIGMOD 2010.
[4] POETS project website. Online, accessed: 22 May 2020. Available: https://poets-project.org/.
[5] M. Naylor, S. W. Moore, and D. Thomas, *Tinsel: a manythread overlay for FPGA clusters*, FPL 2019.
[6] A. Hansson, et al. *Avoiding Message-Dependent Deadlock in Network-Based Systems on Chip*, Journal of VLSI Design, April 2007.
[7] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., and A. DeHon, *GraphStep: A System Architecture for Sparse-Graph Algorithms*, FCCM 2006.
[8] S. Brin and L. Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, 7th International Conference on the WWW, 1998.
[9] E. Izhikevich, *Simple model of spiking neurons*, IEEE Transactions on Neural Networks, 14(6), 2003.
[10] M. Naylor, Research data supporting this paper. Available: https://doi.org/10.17863/CAM.52819.
[11] S. Beamer, K. Asanovic, D. A. Patterson. *The GAP Benchmark Suite*, CoRR, http://arxiv.org/abs/1508.03619, 2015.
[12] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. *The SpiNNaker Project*, Proceedings of the IEEE, 102(5), 2014.
[13] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. Droz, *RAMP Blue: A Message-Passing Manycore System in FPGAs*, FPL 2007.
[14] J. Gray, *A 1680-core, 26 MB Parallel Processor Overlay for Xilinx UltraScale+ VU9P*, Hot Chips 29, 2017.
[15] L. Gwennap. *Adapteva: More flops, less watts*, Microprocessor Report, June 2011.
[16] Siddhartha and N. Kapre. *eBSP: Managing NoC traffic for BSP workloads on the 16-core Adapteva Epiphany-III processor*, DATE 2017.
[17] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*, June 2015.
[18] T. Hoefler, et al. *Scalable communication protocols for dynamic sparse data exchange*, SIGPLAN Notices 45(5), 2010.
[19] D. Ibanez, et al. *Hybrid MPI-thread parallelization of adaptive mesh operations*, Journal of Parallel Computing, vol. 52, 2016.
[20] A. Baker, S. Crivelli, E. Jessup. *An efficient parallel termination detection algorithm*, IJPEDS 2008.
[21] S. Gao, A. G. Schmidt and R. Sass. *Hardware implementation of MPI_Barrier on an FPGA cluster*, FPL 2009,