Orchestrating Safe Streaming Computations with Precise Control

Peng Li
Kunal Agrawal
Jeremy Buhler
Roger D. Chamberlain

Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain, “Orchestrating Safe Streaming Computations with Precise Control,” in *Proc. of International Workshop on Extreme Scale Computing Application Enablement - Modeling and Tools* (ESCAPE), December 2014.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Orchestrating Safe Streaming Computations with Precise Control

Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain
Department of Computer Science and Engineering
Washington University in St. Louis
{*pengli, kunal, jbuhler, roger*}*@wustl.edu*

***Abstract*—Streaming computing is a paradigm of distributed computing that features networked nodes connected by first-in-first-out data channels. Communication between nodes may include not only high-volume *data tokens* but also infrequent and unpredictable *control messages* carrying control information, such as data set boundaries, exceptions, or reconfiguration requests. In many applications, it is necessary to order delivery of control messages *precisely* relative to data tokens, which can be especially challenging when nodes can *filter* data tokens. Existing approaches, mainly data serialization protocols, do not exploit the low-volume nature of control messages and may not guarantee that synchronization of these messages with data will be free of deadlock.**

In this paper, we propose an efficient messaging system for adding precisely ordered control messages to streaming applications. We use a credit-based protocol to avoid the need to tag data tokens and control messages. For potential deadlocks caused by filtering behavior and global synchronization, we propose deadlock avoidance solutions and prove their correctness.

I. Introduction

Streaming computing is a paradigm for parallel and distributed computing. A streaming application is a network of computing nodes connected by first-in-first-out (FIFO) data channels. Each node processes incoming data in streaming (equivalently, online or one-pass) fashion. Streaming can exploit common types of parallelism in applications, such as task parallelism, data parallelism, and pipeline parallelism.

If a node emits no data on an output channel in response to some input, we say that the node has *filtered* the input on that channel. Filtering is a natural behavior in applications such as machine learning [1] and biological sequence comparison [2]. Other applications do not naturally filter data but can be implemented in a filtering way for higher performance. We consider a classic statistics problem, computing variance of pixel intensities in an image, as a compelling example.

The canonical formula for population variance, denoted by σ^2, is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(z_i - \overline{z})^2, \tag{1}$$

where $\overline{z}$ is the average of the N values. Equation 1 seems to requires a two-pass calculation process: one pass to compute the mean, and the second to compute the variance using the mean. However, we can convert this computation to a one-pass algorithm [3], [4] that is more streaming-friendly:

$$\sigma^2 = \overline{z^2} - \overline{z}^2 \tag{2}$$

We can implement Equation 2 as a streaming computation as in Figure 1. The source node u duplicates input data to v and w, which compute $\overline{z}^2$ and $\overline{z^2}$ respectively. These quantities are then merged at node x to compute variance values.

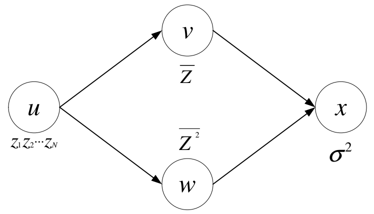

Figure 1. A streaming computation for variance. It occurs as part of large streaming computing systems, including the next generation of VERITAS [5], a ground-based gamma-ray observatory system.

A typical way of computing variances for a stream of images is to process every pixel value until an image boundary is reached, then emit the image's variance. For sparse images, a lot of pixel values are zero. There is no need for node u to send those zeroes, which consume communication bandwidth and processing time at v and w. Instead, u can filter out all zeroes; however, this means that the number of values received by v and w varies from image to image, and u must promptly notify v and w when an image boundary is reached.

Notifications of image boundaries are a type of *control messages* that are distinct from the stream of pixel values. They are inserted by a node into the filtered stream unpredictably and infrequently, and they impact the behavior of downstream nodes when they arrive. Importantly, control messages must be *precisely ordered* relative to the data stream – it is incorrect for a node to group a pixel from before a boundary with the image after the boundary, or vice versa.

Precisely ordered control messages arise in many streaming computations to ensure correctness and/or to boost performance. The variance example is one case of communicating boundaries between finite-length streams. Exceptions in out-of-order CPUs are another common case where precise ordering is needed relative to an instruction stream, only parts of which are sent to each functional unit. In streaming applications with filtering and synchronization, control messages can also be used to avoid deadlocks [6]. These examples involve nodes that can filter their inputs, though precise ordering is also useful for non-filtering paradigms like SDF.

In this work, we describe an efficient strategy for adding precisely ordered control messages to streaming applications with filtering behavior. We pay particular attention to applications in which the communication channels connecting compute nodes have small, statically determined buffer sizes, and in which control messages are kept separate from the data stream for reasons of performance or ease of implementation. Under these circumstances, careful attention must be paid to preserve the desired semantics and to avoid the possibility of deadlocks if buffers become full. We give protocols to ensure precise ordering and deadlock freedom. Due to space constraint, we omit proofs in this paper. The interested reader can find them in [7].

II. Background

This section describes the *synchronized filtering dataflow* (SFDF) computing model that forms the basis of our work. We introduced SFDF in [6]; this work extends that model to accommodate separate control and data paths between nodes.

A. Application Topology

An application consists of compute nodes organized in a directed acyclic multigraph. Nodes are connected by one-way *channels*, each of which reliably delivers data from a sender to a receiver in FIFO order. However, channels have no timing guarantee. Each channel has a known, finite buffer capacity that does not change at runtime. We denote by $|q|$ the buffer size of channel q.

There are two types of channels: *data* and *control*, which carry *data tokens* and *control messages* respectively. For each data channel q connecting two nodes, there is a parallel control channel q'. (We refer to this pair of channels as the *edge* between the nodes.) A node can listen for input on at most one channel at a time; once a channel is chosen for listening, the node can take no further action until input appears on that channel.

B. Filtering and Data Channel Synchronization

When a node of an SFDF application receives and processes input data, it may produce zero or one data token on each of its output data channels. If no token is produced for some input on a given output channel, we say that the node has *filtered* its input on that channel.

When a node takes as input two or more data streams, each of which may be subject to filtering by upstream nodes, the semantics of joining these multiple streams must be clearly defined.

In SFDF, data tokens emitted into a channel bear *strictly increasing* integer indices. In a single computation, a node may consume only data tokens with a common index i, and any output tokens produced by this computation will also have index i. Moreover, the node may not begin computing on data tokens with index i until, for each of its input channels, either the next token on that input has index i, or it is known that *no* token with index i will ever appear. These semantics ensure that, even though different channels are not synchronized, all tokens with a common index, and *only* such tokens, are processed together by a single node. In other words, in a single computation, a node should only consume data tokens with the same index, and all data tokens with the same index must be consumed by the node in only one computation.

Note that, if it is possible for a node to receive inputs on only a subset of its input channels due to filtering, then the application designer must specify the meaning of the node's computation for all such possible subsets.

C. Control Channel Behavior

Control channels carry *control messages*, which have one of a finite set of types and can contain arbitrary content. The order in which control and data are processed is *precise*: if a node sends a data token with index i on data channel q of an edge, followed immediately by a control message on the associated control channel q', then this message should be processed by the receiving node after computing on all input data with index i but before consuming data with any index $> i$. A node may send multiple control messages on an edge between two consecutive data indices.

Intuitively, control messages are sent only rarely compared to data tokens. By splitting these messages out into their own channels, we avoid multiplexing them with the data tokens in the higher-volume data channels. This separation permits strong typing assumptions about data channels, which may lead to more efficient implementation; moreover, it simplifies the common case of sending and receiving data between nodes, which may benefit the application's latency and throughput. Unfortunately, while multiplexing data and control in one channel trivially guarantees precise ordering, the same is not true for separate, unsynchronized control and data channels.

In what follows, we first give a protocol to ensure precise ordering of control messages and data tokens on a single edge. We then show how to extend our protocol to ensure that SFDF applications with control channels execute safely without global deadlock.

III. Ensuring Precise Control-Data Ordering

Consider an edge e consisting of two nodes connected by data and control channels q and q'. We will enforce precise ordering of control messages and data tokens on this edge through the use of *credits*. The sender and receiver each maintain internal credit balances, which are integer values that are initially zero. When a receiver receives some number c of credits on e, its credit balance RCB_e is incremented by c; when it *consumes* a data token on e, RCB_e is decremented by one. The sender's credit balance SCB_e is incremented by one whenever it *sends* a data token; when it sends c credits to the receiver on e, SCB_e is decremented by c.

Credits can be attached to any control message. If credits must be sent but no other control message is pending, the sender may send a *credit message* with no intrinsic content but its attached credit. When the receiver sees a control message, it immediately increments its credit balance and may then switch to the data channel and attempt to read data tokens without first processing the control message itself.

A. Credit Balance Protocols

Intuitively, a credit represents permission from the sender for the receiver to consume a data token. It implies that *there are no pending control messages that must be processed before consuming the next data token.* The receiver may consume data tokens as long as its credit balance is positive, but when the balance goes to zero, it must wait for the sender either to supply more credits or to send control messages that should be processed before the next data token. The formal protocol followed by the receiver is given in Algorithm 1.

Algorithm 1: Receiver Credit Balance Protocol

```
while $RCB = 0$ do
    wait for a control message on $q'$
    let $c$ be credit value carried by message
    if $c = 0$ then
        consume message
    else
        Detach $c$ credits from message
        $RCB \leftarrow RCB + c$
wait for a data token on $q$
consume token
$RCB \leftarrow RCB - 1$
```

The sender, for its part, must issue credit to consume a pending data token only after it knows that no control message should precede that token. Algorithm 2 gives a sender's protocol parametrized by a threshold T, which should be set less than the buffer size of the outgoing data channel. When the threshold is exceeded with no intervening control messages, the sender issues credit to drain the data channel's buffer. Note that, in this and all following protocols, all **emit** operations block until the output channel is not full.

Algorithm 2: Sender Credit Balance Protocol

```
if token is ready then
    emit token on $q$
    $SCB \leftarrow SCB + 1$
while control message is ready OR
  $SCB > T$ do
    emit message on $q'$ with $SCB$ credits
    $SCB \leftarrow 0$
```

B. Correctness and Safety

We argue that the sender and receiver protocols ensure precise ordering of control messages vs. data tokens. Please see [7] for proofs.

Theorem III.1. *If a receiver and sender are connected by an edge and behave as in Algorithms 1 and 2, and the sender issues a data token d followed by a control message m, then the receiver will process m after d but before the next token following d.*

The above argument assumes that the sender and receiver are always able to make progress. Because the data and control channels have finite buffers, the sender could at some point be blocked trying to send a data token or control message into a channel with a full buffer, or the receiver could be blocked waiting for tokens or messages when none are yet visible to it. If both the sender and the receiver are blocked indefinitely, the system is deadlocked. We now verify that our protocol makes such a deadlock impossible.

Theorem III.2. *If a receiver and sender are connected by an edge and behave as in Algorithms 1 and 2, this pair of nodes will never deadlock.*

IV. Combining SFDF with Precise Control

We now explore how to combine SFDF's synchronization of multiple, possibly filtered input streams with the use of separate data and control channels. Recall that an SFDF application is a directed acyclic multigraph. Each edge e of this multigraph now consists of two channels: a data channel q_e, and a control channel q'_e. Each edge also holds variables sufficient to implement the credit protocols of the previous section, including sending and receiving credit balances SCB_e and RCB_e

and a threshold T_e that is smaller than data channel buffer size $|q_e|$.

Algorithm 3 describes how to combine SFDF with control channels. To ensure precise data and control ordering, each node implements Algorithm 1 on each of its input edges and Algorithm 2 on each of its output edges. Edges are processed sequentially in an arbitrary order. To synchronize across data channels, the receiving protocol is split into two parts: part one ensures that data tokens are available on all input edges' data channels, while part two decides which tokens to read (based on their indices) in order to start the next computation. The common index i of tokens consumed by a computation at a node is called the *computation index*. Note that no attempt is made to synchronize control messages across edges.

Not every node in an application may have inputs or outputs. In particular, *source nodes* have no inputs but rather generate tokens and messages spontaneously, following only the output part of the protocol.

Algorithm 3: Single-node behavior in SFDF with control messages.

```
foreach input edge $e$ do
    while $RCB_e = 0$ do
        wait for a control message on $q'_e$
        let $c$ be credit value carried by message
        if $c = 0$ then
            consume message
        else
            Detach $c$ credits from message
            $RCB_e \leftarrow RCB_e + c$
    wait for a data token on $q_e$
let $i$ be least index among data tokens on all edges
foreach input edge $e$ do
    if token on $q_e$ has index $i$ then
        consume token
        $RCB_e \leftarrow RCB_e - 1$
perform computation for index $i$
foreach output edge $e$ do
    if token is ready on $e$ then
        emit token on $q_e$ with index $i$
        $SCB_e \leftarrow SCB_e + 1$
    while control message is ready on $e$ OR
      $SCB_e > T_e$ do
        emit message with $SCB_e$ credits
        $SCB_e \leftarrow 0$
```

Unfortunately, this straightforward combination of SFDF and the credit protocols is prone to deadlock. We explore this issue and its remediation next.

A. Deadlocks Due to Full Data Channels

To further focus the discussion, we make two simplifications. First, we will assume until otherwise stated that *no control channel ever becomes full* during a computation. This is intuitively reasonable if control messages are sent much less frequently than data tokens. Second, we observe that, because a node always sends the credit to receive a data token after the token itself, *a node cannot block indefinitely on an empty data channel*. Indeed, if a node is waiting on a data channel, then it has unexpended credit, which means the corresponding data token is already in flight.

With the above simplifications, a blocking cycle must contain only two types of edges: *full* data channels and *empty* control channels. The following example shows that such a deadlock is possible. Consider four nodes connected as in Figure 1 above, with edges uv, vx, uw, and wx. Every computation of u produces data tokens on q_{uv} and q_{uw}, and every computation of v produces a data token on q_{vx}; however, w filters more than half of its inputs on q_{wx}. Assume the data channels on all four edges have the same buffer size 32, the threshold for scheduling credit messages is $T = 31$ (recall that a credit message is prompted if buffered tokens are more than T), and that no control messages are sent other than credit messages. After some computations, the system reaches the state shown in Figure 2.

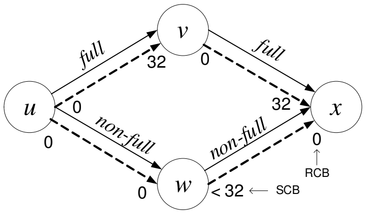

Figure 2. A deadlock example. w filters 46 of 64 consumed data tokens and no other node filters data. Now data channels uv and vx are full, blocking u and v; SCB values for uw and wx are not big enough to prompt credit messages, blocking w and x.

At this point, if u does one more computation (and w filters the resulting data token), then we have that (1) u is blocked by v on a full q_{uv}; (2) v is blocked by x on a full q_{vx}; (3) x is blocked by w waiting for credit on an *empty* control channel q'_{wx}; and (4) w, which has no pending tokens and hence no credit, is blocked by u waiting for credit on the empty control channel q'_{uw}. Hence, the system is deadlocked with a blocking cycle.

This example actually illustrates two related but distinct causes of deadlock. If w sends *no* data on q_{wx}, then deadlock occurs because x has no input on this channel but does not know that none will arrive. This kind of deadlock also occurs in SFDF networks without control channels [6]. If, however, w sends *some* data on q_{wx}, x has enough data to make progress, but in the absence of control messages, nothing prompts w to send its stored credit to enable x to use the data. This kind of deadlock is a side effect of the credit protocols. Below, we propose a modified protocol to avoid both causes of deadlock.

B. Avoiding Deadlocks with Periodic Communication

To avoid deadlock, we modify our protocol in two ways. First, we periodically flush pending credit from the sender to the receiver, so that data tokens cannot linger indefinitely at the receiver with no credit. Second, we periodically notify the receiver if tokens with consecutive indices have been filtered by upstream senders, using a new kind of control message called a *dummy message* that carries the index of the sender's most recent computation.

The augmented protocol is shown in Algorithm 4. The receiver's protocol is essentially unchanged, except that, instead of a data token with index i, an edge may present a dummy message with index $j \geq i$, which implies that no token with index i will ever be received on that edge, and it should therefore not block computation i from proceeding.

Algorithm 4: Adding dummy messages to SFDF with control.

```
foreach input edge $e$ do
    while $RCB_e = 0$ do
        wait for a control message on $q'_e$
        let $c$ be credit value carried by message
        if $c = 0$ then
            if message is a dummy then
                break
            consume message
        else
            Detach $c$ credits from message
            $RCB_e \leftarrow RCB_e + c$
    if $RCB_e > 0$ then
        wait for a data token on $q_e$
let $i$ be least index among tokens on edges with
  $RCB > 0$ and dummies on edges with $RCB = 0$
foreach input edge $e$ do
    if $RCB_e > 0$ AND token on $q_e$ has index $i$ then
        consume token
        $RCB_e \leftarrow RCB_e - 1$
    else if dummy on $q'_e$ has index $i$ then
        discard dummy
perform computation for index $i$
foreach output edge $e$ do
    if token is ready on $e$ then
        emit token on $q_e$ with index $i$
        $SCB_e \leftarrow SCB_e + 1$
        $\text{LastSentIdx}_e \leftarrow i$
    if $SCB_e = 0$ AND $i - LastRecvIdx_e > [e]$ then
        emit dummy on $q'_e$ with index $i$
        $\text{LastRecvIdx}_e \leftarrow i$
    while control message is ready on $e$ OR
      $i - LastRecvIdx_e > [e]$ do
        emit message on $q'_e$ with $SCB_e$ credits
        $SCB_e \leftarrow 0$
        $\text{LastRecvIdx}_e \leftarrow \text{LastSentIdx}_e$
```

The sender's protocol is augmented with two state variables: LastSentIdx_e, which tracks the index of the last data token *actually sent* by the sender, and LastRecvIdx_e, which tracks the last index for which the receiver has *permission* to consume inputs with that index from e. If the sender does too much work (as measured by the size of the gap between the index i of the most recent computation and LastRecvIdx_e) without enabling the receiver to proceed, then it either flushes its pending credit for any data tokens sent in this gap, or, if no tokens were sent, transmits a dummy message with index i to tell the receiver not to expect them. The largest permissible gap size for an edge e is called its *heartbeat interval*, denoted in the protocol by $[e]$.

The remaining question is how large to make the heartbeat interval for each edge. It would be trivially safe to set $[e] = 0$ for every e, but doing so would flush credit or send a dummy after every computation, which would add excessive communication overhead. Instead, we utilize the following scheme adapted from [8]. Given a dataflow graph G, for each *undirected cycle* C of G, suppose the set of clockwise edges is H_1 and the set of counterclockwise edges is H_2. We enforce the following inequality constraints for cycle C:

$$\Sigma_{e \in H_1}[e] \quad < \quad \Sigma_{e \in H_2}|q_e| \qquad (3)$$

$$\Sigma_{e \in H_2}[e] \quad < \quad \Sigma_{e \in H_1}|q_e|. \qquad (4)$$

An application graph may have more than one undirected cycle, in which case each such cycle generates a pair of constraints as described. We also need to avoid local deadlocks, so the following constraint, which we specified for the sender's protocol of Section III, is added for each edge e:

$$[e] < |q_e|. \qquad (5)$$

The union of all these constraints defines a feasible polyhedron of heartbeat intervals for the application, and we select a set of intervals from this feasible region.

Theorem IV.1. *Assuming that control channels never become full, if every node in an SFDF application behaves as Algorithm 4 with heartbeat intervals constrained by Inequalities (3), (4), and (5), then the application cannot deadlock.*

V. Related Work

The control messaging system proposed in this paper is based on synchronized filtering dataflow (SFDF), which can be viewed as Homogeneous Synchronous Dataflow (HSDF) [9] with the addition of node filtering. HSDF is a special type of Synchronous Dataflow (SDF) [10] where the data rate (the number of items a node reads/writes from its input/output channels) is 1 for all channels. SFDF applications are vulnerable to deadlocks with finite buffer capacity due to filtering and synchronization. We previously described the use of dummy messages, a special type of control message, sent in-band with data streams to avoid deadlock [6], [11], [12], [8]. The system in this paper incorporates dummy messages but instead delivers them through dedicated control channels.

Synchronizing data streams and control messages is also common in network protocols. The Internet Control Message Protocol (ICMP) is designed to exchange control messages between two Internet devices during data transmission [13]. The work most closely related to ours is StreamIt's Teleport Messaging system [14]. Both their work and our work address the problem of sending infrequent and irregular control messages for streaming applications computing on regular data streams. The key difference is that Teleport Messaging is based on the SDF model and uses dependence analysis for precise event handling, while our precise control mechanism, the Credit Balance Protocol, does not rely on any specific model.

VI. Conclusion and Future Work

Precisely ordered control messages are important to the correctness and performance of streaming applications. In this work, we have designed a messaging system that works for streaming computations in which nodes can filter their inputs. It can work even in streaming pipelines that require global synchronization, as in SFDF. We have given protocols and sufficient design constraints to avoid deadlocks while delivering precise control even in the presence of filtering.

In the future, we plan to investigate how to efficiently find the highest-throughput set of heartbeat intervals that satisfy the constraints of Inequalities (3), (4), and (5). Currently, computing a satisfactory (not necessarily maximal) set of intervals given the buffer sizes requires superpolynomial time in the application size. We will investigate faster interval selection algorithms with stronger guarantees.

References

[1] P. Viola and M. Jones, "Robust real-time object detection," *Int'l J. Computer Vision*, vol. 57, no. 2, 2002.

[2] A. C. Jacob, J. M. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain, "Mercury BLASTP: Accelerating protein sequence alignment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 2, 2008.

[3] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[4] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Algorithms for computing the sample variance: Analysis and recommendations," *The American Statistician*, vol. 37, no. 3, pp. 242–247, 1983.

[5] E. J. Tyson, J. Buckley, M. A. Franklin, and R. D. Chamberlain, "Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 595, no. 2, pp. 474–479, 2008.

[6] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Deadlock avoidance for streaming computations with filtering," in *Proc. 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, 2010, pp. 243–252.

[7] P. Li, K. Agrawal, J. Buhler, and R. Chamberlain, "Streaming computations with precise control," Washington University in St. Louis, All Computer Science and Engineering Research, Tech. Rep. 14-1, October 2014. [Online]. Available: http://openscholarship.wustl.edu/cse_research/1

[8] P. Li and J. Buhler, "Polyhedral constraints for bounded-memory execution of synchronized filtering dataflow," *Workshop on Data-Flow Execution Models for Extreme Scale Computing*, Sep. 2013.

[9] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

[10] ——, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, 1987.

[11] P. Li, K. Agrawal, J. Buhler, R. D. Chamberlain, and J. M. Lancaster, "Deadlock-avoidance for streaming applications with split-join structure: Two case studies," in *IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, Jul. 2010, pp. 333–336.

[12] J. D. Buhler, K. Agrawal, P. Li, and R. D. Chamberlain, "Efficient deadlock avoidance for streaming computation with filtering," in *Proc. 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 235–246.

[13] J. Postel, "Internet control message protocol," *RFC-792*.

[14] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, "Teleport messaging for distributed stream programs," in *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, 2005, pp. 224–235.