Deadlock-avoidance for Streaming Applications with Split-Join Structure: Two Case Studies

Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain, Joseph M. Lancaster
Department of Computer Science and Engineering
Washington Univeristy in St. Louis
St. Louis, Missouri, United States
Email: {pengli, kunal, jbuhler, roger, lancaster}@wustl.edu

***Abstract*—Streaming is a highly effective paradigm for expressing parallelism in high-throughput applications. A streaming computation is a network of compute nodes connected by unidirectional FIFO channels. When these computations are mapped onto real parallel platforms, however, some computations, especially ones in which some nodes act as filters, can deadlock the system due to finite buffering on channels. In this paper, we focus on streaming computations which contain a commonly used structure called split-join. Based on our previous work, we propose two correct deadlock-avoidance algorithms, named the Propagating Algorithm and the Non-propagating Algorithm. Our evaluation of two representative applications, biological sequence alignment and random number generation, shows that the Non-propagating Algorithm has very small communication overhead. For systems with large buffers or a low filtering ratio, the communication overhead of the Non-propagating Algorithm is negligible.**

***Keywords*-Streaming Computation, Deadlock Avoidance, BLAST, Pseudorandom Number Generation**

I. Introduction

Streaming is becoming an increasingly popular computing paradigm since it can be used to conveniently express throughput-oriented parallel computation. A streaming application is essentially a network of compute nodes connected by unidirectional communication channels. Each compute node reads data items from its input channels and places data items on its output channels. From the programmer's viewpoint, the compute nodes can perform arbitrary computation and channels provide FIFO guarantees. Here, we assume that computation on each node is bounded and deterministic for a particular set of input values.

In this paper, we focus on split-join topologies in which asynchronous computations on each path must synchronize their results at the sink node. To permit the synchronization, we add a non-negative timestamp (a.k.a. "index") to each data item. The second feature of the streaming computations considered in this paper is that some nodes are *filters*. If a computation at node v does not result in an output data d being emitted on an outgoing channel q, we say that v *filters* data d. In our applications, filtering is a data-dependent behavior that cannot be predicted at compilation time.

Streaming computations with split-join structures and

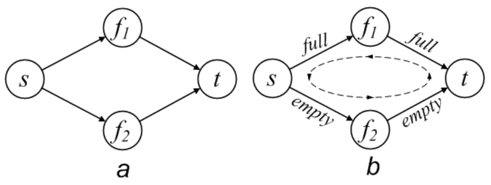

Figure 1. A deadlock example

filtering nodes may *deadlock* during execution. Consider, for example, the system of Figure 1a, in which a uniform random number generator s sends pairs of random deviates to nodes f_1 and f_2, each of which implements a transform with rejection to produce non-uniform random deviates. To ensure repeatable output, we index the uniform deviates sent from source s and demand that t emits the combined output streams from f_1 and f_2 in index order. Suppose that node f_2 rejects a number of successive inputs, and so emits nothing on channel f_2t, while node f_1 continues to emit outputs on f_1t. Since channels may have arbitrary delays, node t cannot immediately consume data from f_1t because it does not know whether data with an earlier index might subsequently appear on f_2t. If channels sf_2 and f_2t become empty due to filtering while sf_1 and f_2t fill, the system may reach the situation shown in Figure 1b, where node t is unable to consume data since it is waiting to receive data that must be sent from s, but s is unable to generate any more output because it is blocked trying to emit onto the full path from s to f_1. This situation is a deadlock since no node can make progress for an infinite amount of time.

In previous work [1], we showed that in asynchronous streaming systems with a variety of topologies, including split-join systems, the combination of synchronization and filtering can cause deadlock. That work proposes algorithms for deadlock avoidance in such systems. In this paper, we adapt those algorithms for the special case of split-join structures. In addition, we empirically evaluate these algorithms on two real-world streaming applications that fit the split-join model: the Mercury BLAST system [2][3], and the non-uniform random number generation.

II. Deadlock-avoidance Algorithms

As mentioned earlier, filtering nodes can lead to deadlocks in streaming applications. The general idea behind our deadlock-avoidance algorithms is the concept of dummy messages. A *dummy* is a distinguished class of messages with an index but no content of its own. Its purpose is to notify receivers that a data item in the stream with this index has been filtered by an upstream sender. The algorithms described in [1] work for general network topologies including, but not limited to, split-join structures. In this section, we describe these algorithms, slightly simplified to only handle split-join structures.

The first algorithm is a simple algorithm that sends a dummy message each time a data item is filtered, referred to as *Naive Algorithm* later. In effect, this approach converts a filtering node to a non-filtering node, but will send many unnecessary dummy messages.

The other two algorithms try to send fewer dummy messages than the simple naive algorithm. The first of these two algorithms, called *Propagating Algorithm*, is a algorithm in which only the source (split) node sends dummy messages and these messages are propagated until they get to the sink (join) node. That is, if a node receives a dummy message with a particular index, it must propagate it on its output links. The source node sends a dummy to output channel q if the computing index has increased by $[q]$ since the last dummy sent to this channel. $[q]$ is computed as the minimum total buffer size of the other paths between the source and the sink. The last algorithm is called *Non-propagating Algorithm*. Here, each filtering node can generate dummy messages, but there is no requirement to propagate them. In this algorithm, a node sends a dummy message on channel q if its computing index has increased by $[q]$ since the last data item or dummy message was sent on this channel, as shown in Algorithm 1. $[q]$ is computed as the minimum total buffer size of other paths divided by the number of filtering nodes on this path, as shown in Algorithm 2. Due to space limitations, we are unable to provide algorithmic descriptions for the Propagating Algorithm here. Interested users may find [1] a useful reference.

Algorithm 1: The Non-propagating Algorithm: Single-node Behavior

```
ComputeIndex $\leftarrow 0$ ;
foreach output port $q$ do
    LastOutputIndex$_q \leftarrow 0$ ;
while $ComputeIndex \neq Index\ of\ EOS$ do
    if not source node then
        wait until each input channel has a token ;
        let $T$ be minimum index of any pending token;
        consume tokens with index $T$ from input;
    else
        $T \leftarrow$ ComputeIndex $+ 1$ ;
    ComputeIndex $\leftarrow T$ ;
    perform computation on data tokens with index $T$ ;
    foreach output channel $q$ do
        if a token with index $T$ will be emitted on $q$
        then
            schedule a token with index $T$ for $q$ ;
            LastOutputIndex$_q \leftarrow T$ ;
        else if $T - LastOutputIndex_q \geq [q]$ then
            schedule a dummy with index $T$ for $q$ ;
            LastOutputIndex$_q \leftarrow T$ ;
    if not sink node then
        emit tokens with index $T$, including any dummy
```

Algorithm 2: The Non-propagating Algorithm: Dummy Interval Calculation

```
Input: A system abstracted as graph $G = \{V, E\}$
Output: Dummy intervals for each channel
foreach edge $q \in E$ do $[q] \leftarrow \infty$ ;
foreach path $p$ between the source $s$ and the sink $t$ do
    let $b$ be the minimum total buffer size of other
        paths between $s$ and $t$;
    let $m$ be the number of filtering channels on $p$;
    foreach filtering channel $q$ on $p$ do
        $[q]$ = $\lceil b/m \rceil$;
```

III. Applications and Experimental Result

This section describes our two representative applications which both have a split-join structure with filtering nodes. Therefore, both of these applications are vulnerable to deadlocks. Mercury BLAST, our biological application, is much more vulnerable than the random number generator and, in fact, this work was motivated by our observation of actual deadlocks while running Mercury BLAST. For each application, we evaluate the overhead associated with deadlock-avoidance algorithms. The dummy message scheduling causes very little computation, so our performance evaluation pays attention to the number of dummy messages sent by nodes across communications links.

A. Mercury BLAST

Mercury BLAST [2][3] is an FPGA-accelerated implementation of the Basic Local Alignment Search Tool (BLAST), a bioinformatics tool for comparing DNA or protein sequences to discover regions of biologically meaningful similarities.

BLAST uses filtering heuristics to quickly discard large portions of the database that are unlikely to match the query sequence. The principal heuristic, *seed matching*, divides the database into overlapping sequences of a short, fixed length w, then tests whether each such w*-mer* appears in the query. If a w-mer is present at position x in the database and position y in the query, this test generates a *seed match* (x, y). The portions of the database and query near these coordinates are then subjected to further testing to confirm or reject the presence of biologically meaningful similarity. Mercury BLASTN implements BLASTN's filters as a streaming computation network, with a split-join topology as shown in Figure 2. The query is preprocessed into a lookup table stored in seed matching module 1b. The database is then streamed into module 1a, which both divides it into w-mers that are sent to 1b for matching and forwards it unmodified to later stages of the application (represented in the diagram by module 2). Because the database input channel to module 2 has a finite buffer (on the order of 64 Kchars), there is a risk of deadlock if 1b happens not to find any seed matches in a long enough piece of the database.

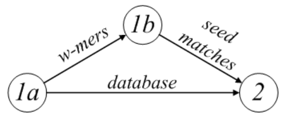

Figure 2. The first two stages of Mercury BLAST

To investigate the number of dummy messages sent in Mercury BLAST, we ran Mercury BLASTN to search a set of 1000 queries sampled from human messenger RNA (mRNA) sequences against all other vertebrate mRNA as the database. This represents 787 billion input elements. We monitor the number of dummy messages out of module 1a. We run the Non-propagating Algorithm and use the Time-Trial platform described in [4] to count the actual dummy messages. The amount of dummy messages generated by the Naive Algorithm can be estimated by the multiplication of input data volume and the filtering rate and dummy messages generated by the Propagating Algorithm can be calculated by dividing the number of input data items by the dummy interval, which is a fixed value. We set the buffer size of the database channel to 32, 256, and 2048, which stands for the database items the channel can hold. The corresponding dummy intervals are 32, 256, and 2048 for the Propagating Algorithm and 16, 128, and 1024 for the Non-propagating Algorithm. Our results (shown in Table I) indicate that the Non-Propagating Algorithm has, by far, the smallest message overhead. The dummy messages it sends can be reduced exponentially as the buffer size increases.

Table I
MEASURED DUMMY MESSAGE COUNTS FROM MODULE 1A FOR MERCURY BLASTN

	Dummy message count		
Total Buffer Size (msgs)	32	256	2048
Naive Algo.	787×10^9	787×10^9	787×10^9
Prop. Algo.	25×10^9	3×10^9	0.4×10^9
Non-prop.	36×10^9	36×10^6	72,000

B. Pseudorandom Number Generation

Pseudorandom number generators (PRNGs) are widely used in applications, such as Monte Carlo simulation, that require a long stream of input values that appear "random" but can be generated repeatably. Some applications need numbers that follow some non-uniform distributions, such as a Gaussian or exponential. A common strategy employed by non-uniform PRNGs is *rejection sampling*. Rejection-based PRNGs use k-tuples of uniform deviates, for some fixed $k \geq 1$, to drive a second sampling process that sometimes produces a sample from the desired target distribution and sometimes produces nothing. Classic examples include the Marsaglia polar method [5] and the ziggurat algorithm [6], each of which have $k = 2$.

When an application has a high demand for pseudorandom numbers, and the necessary transform is computationally demanding, the generator may be parallelized using a pipeline stream and has some stages replicated to speedup the bottleneck, as Figure 3 shows. Node s generates a sequence of uniform deviates, which are transformed by filters f_i. To ensure that we can produce the same stream of values given the same seed as the sequential program, t must implement some form of predictable synchronization over all filters.

One approach to synchronization is for s to assign a monotonically increasing index to each k-tuple sent from s to a filter. Filters emit samples with the same index as the k-tuples that produced them, and the sink t emits samples in order of their indices. There are also other alternative approaches, but for whatever approach, deadlock can occur if any filter happens to discard a long sequence of inputs while another continues to produce outputs. We choose the latter it is more efficient for our deadlock algorithms.

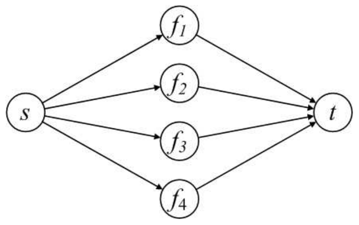

Figure 3. A split-join pipeline for PRNG

To assess the performance impact of our algorithms on PRNGs, we simulate the Marsaglia polar method, which has a rejection rate of 21.46%. We replicate four filters between the source and the sink. The total buffer size of each path is set to 10, 100, and 1,000 in different runs. The source generates 1 million uniformly distributed random numbers and distributes them evenly to four replicated filters, each of which runs the Marsaglia polar method independently. We apply the three deadlock avoidance algorithms and count the total number of dummy messages each of them generates. From the data in Table II, the Non-propagating Algorithm is again the most efficient. Sometimes it sends no dummy message at all because it can guarantee no deadlock based on the filtering history.

Table II
SIMULATION RESULTS FOR MARSAGLIA POLAR ALGORITHM

	Dummy message count		
Total Buffer Size (msgs)	10	100	1000
Naive Algo.	215,030	215,030	215,030
Prop. Algo.	200,000	20,000	2,000
Non-prop.	1	0	0

The Marsaglia polar method represents a category of applications in which an arbitrary number of filters can be replicated between the source and the sink if the filter is a performance bottleneck of the system. Due to the conservative nature of the three algorithms, the number of dummy messages is decided by the channel buffer size configuration and data filtering history rather than the actual instances of deadlock. Hence we can use alternative buffer size and filtering ratio configurations to evaluate the performance of our algorithms in potential applications. We executed another two simulations with different filtering ratio and buffer size configurations. We conclude that the Non-propagating Algorithm has the least overhead. Even with a high filtering ratio (95%) and small channel buffer size (only 10 data items), the communication overhead is less than 10%. In low filtering ratio (5%) or large buffer size (1000 data items) cases, the overhead is negligible.

IV. RELATED WORK

In [1], we formalized a streaming computation model based on directed acyclic multigraphs, which are more general than the split-join structure. For that model, we proposed three deadlock-avoidance algorithms and proved the correctness of each of them. They are the basis of the algorithms adopted in this work.

There has been considerable prior research on dealing with deadlocks in distributed systems. Chandy et al. developed algorithms to detect both resource deadlock and communication deadlock [7]. Mitchell et al. designed a deadlock detection algorithm using public and private labels [8]. There has been some recent work on deadlock avoidance using Kahn Process Networks [9], which is a theoretical model close to streaming computation, such as [10] by Geilen et al. and [11] by Allen et al. These deadlock avoidance and resolution algorithms require runtime changes to channel capacities, while our algorithms do not. Furthermore, some applications may have difficulty in dynamic resource allocation, such as those deployed on FPGAs.

V. CONCLUSION

In this work, we have analyzed the split-join streaming computation model and demonstrated how applications like BLAST and PRNG can be efficiently parallelized within this model. The PRNG actually represents a category of applications in which some pipeline stages can be arbitrarily replicated to satisfy performance requirements. To solve the deadlock problem in these applications, we proposed two deadlock-avoidance algorithms. Experiments show that the Non-propagating Algorithm is more efficient, which adds negligible overhead while correctly avoids deadlocks.

ACKNOWLEDGMENT

This work was supported by NSF awards CNS-0751212 and CNS-0905368 and by NIH award R42 HG003225 (through BECS Technology, Inc.). R.D. Chamberlain is a principal in BECS Technology, Inc.

REFERENCES

[1] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Deadlock avoidance for streaming computations with filtering," in *ACM Symposium on Parallelism in Algorithms and Architectures*, (Santorini, Greece), June, 2010.

[2] P. Krishnamurthy, J. Buhler, R. D. Chamberlain, M. A. Franklin, K. Gyang, A. C. Jacob, and J. M. Lancaster, "Biosequence similarity search on the Mercury system," *VLSI Signal Processing*, vol. 49, no. 1, pp. 101–121, 2007.

[3] J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "Acceleration of ungapped extension in Mercury BLAST," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 33, no. 4, pp. 281–289, 2009.

[4] J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *Proc. of 22nd IEEE Int'l System-on-Chip Conf. (SoCC)*, (Belfast, Northern Ireland, UK), pp. 23–28, Semptember 2009.

[5] G. Marsaglia, "Improving the polar method for generating a pair of normal random variables," Tech. Rep. D1-82-0203, Boeing Sci. Res. Labs., Seattle, WA, Sept. 1964.

[6] G. Marsaglia and W. W. Tsang, "The ziggurat method for generating random variables," *Journal of Statistical Software*, vol. 5, pp. 1–7, 10 2000.

[7] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, 1983.

[8] D. P. Mitchell and M. J. Merritt, "A distributed algorithm for deadlock detection and resolution," in *Proc. of ACM Symp. on Principles of Distributed Computing*, pp. 282–284, 1984.

[9] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, pp. 471–475, 1974.

[10] M. Geilen and T. Basten, "Requirements on the execution of Kahn process networks," in *Proc. of 12th European Symposium on Programming*, pp. 319–334, 2003.

[11] G. Allen, P. Zucknick, and B. Evans, "A distributed deadlock detection and resolution algorithm for process networks," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, pp. 33–36, Apr. 2007.