Approaches for Implementing Persistent Queues within Data-Intensive Scientific Workflows

Michael Agun, Shawn Bowers

Deptartment of Computer Science, Gonzaga University
dagun@gonzaga.edu, bowers@gonzaga.edu

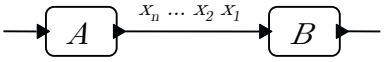
***Abstract*—Many scientific workflow systems are built on dataflow-based models of computation in which data drives the execution of workflow components. An advantage of using dataflow models is their straightforward semantics (which includes support for branching, merging, and looping) and their ability to concurrently execute workflow steps. However, for many data-intensive workflows the dataflow model often requires data buffering. Current systems largely perform buffering through in-memory queues which can lead to buffer overflow and performance degradation as queues reach capacity (e.g., because of paging). We describe an alternative framework that leverages external storage to implement buffers (which we refer to as *persistent queues*) within data-intensive scientific workflows. Our framework can easily be used with different underlying storage technologies, and we consider and evaluate three distinct approaches: a traditional relational database implementation, a non-relational implementation designed for fast reads and writes, and a specialized approach that can further reduce external buffering overhead. In addition, the use of persistent queues can provide detailed provenance information "for free" by capturing the input and output information of each workflow component during workflow execution. Although many systems provide such provenance information, we show how this information can be captured both efficiently and can be used to improve overall workflow performance through persistent queues.**

I. Introduction

Scientific workflows from a wide range of disciplines are often data-intensive, requiring many different tools working together to manipulate, analyze, and visualize large data sets. For example, workflows for earth and environmental science [1], molecular biology [2], [3], phylogenetics [4], image processing [5]–[7], and financial analyses [8] often involve the use of multiple tools to combine different datasets (often ranging from tabular data to streaming data), perform one or more statistical or specialized analyses, and then generate and display visualizations of analytical results. In such workflows, workflow components (that wrap and call external applications) often receive and produce large numbers of data "*tokens*" containing either fine-grain or course-grain objects (e.g., numerical data, gene sequences, images, or more complex structures such as phylogenetic trees [2], [7]).

Workflow systems that support data-intensive analyses (e.g., [9]–[12]) are often based on dataflow models of computation [13]. Dataflow has a number of advantages for designing and executing data-intensive workflows, including a simple and formal semantics, and the ability to leverage a variety of constructs such as branching, merging, conditional execution, and iteration (e.g., to implement while loops or fixed-point computations [4]). Another significant advantage of dataflow is its inherent ability to allow data tokens to be streamed between components that are executed concurrently (so called "*pipeline parallelism*"). Similar to data parallelism [3], [14], [15], pipeline parallelism can also provide speedup of scientific workflows by allowing different components within a pipeline to be executed concurrently. Pipeline parallelism, however, requires token buffering, and current workflow systems supporting pipeline parallelism typically employ in-memory buffering approaches. In-memory buffering can lead to performance problems including running out of memory (buffer "overflow") and slow-down caused by paging.

For example, consider the following simple "workflow" with two components A and B connected by a single dataflow channel. As shown, A and B may also be connected to "upstream" and "downstream" components, respectively. In



this example, the set of invocations of A produce an overall sequence $x_1, x_2, \ldots, x_n$ of data tokens that are consumed by invocations of B. If each invocation of A produces one data token, each invocation of B consumes one token, and A and B execute in the same amount of time, then no additional buffering is required on the channel (i.e., on the "pipe" connecting A to B). In this case, each invocation of A produces a data token that is then immediately used by an invocation of B. Note that as B executes, A can execute again concurrently, producing the next data token for the subsequent invocation of B. This concurrency decreases overall execution time of the workflow since A does not have to wait until B finishes its execution. However, if B is slower than A (or alternatively, if an invocation of A produces more tokens than B consumes in a single invocation), then data passed on the channel must be buffered. For instance, if A is twice as fast as B, two invocations of A can occur for each invocation of B, which means two data tokens are produced by A for each invocation of B. In this case, a buffer (implemented as a queue) must be used to hold these "extra" tokens produced by A.

As shown in Fig. 1, the first invocation of A (denoted A_1) produces a token x_1, which is read by B_1 (the first invocation of B). While B_1 executes, two invocations A_2 and A_3 of A occur, producing data tokens x_2 and x_3, respectively, which

978-0-7695-4461-8/11 $26.00 © 2011 IEEE
DOI 10.1109/SERVICES.2011.57

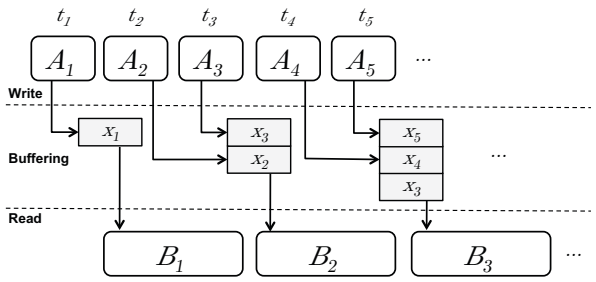


Fig. 1. Workflow trace showing pipeline parallelism and buffering, where the size of the buffer for B increases over time and, in general, without a fixed bound.

are stored in the channel's queue. The first of these, x_2 is then read by the second invocation of B, during which two more invocations of A occur, increasing the buffer size needed by one token. Thus, as the workflow executes, the queue used for buffering the channel between A and B continues to grow with time.

Even in this simple case we can quickly run into memory problems, e.g., if the size of data tokens output by A is large, if A and B are executed many times, or if there are a large number of channels within the workfow (each of which often occur in data-intensive workflows [4], [9], [14]). Thus, using in-memory queues for buffering, we can quickly reach memory limitations (and corresponding performance issues) or worse, we may run out of memory altogether causing the entire workflow to halt.

In this paper, we propose a set of strategies for leveraging external storage to implement channel buffers in scientific workflows. Our goal is twofold: (i) we want to avoid the above issues with in-memory queues while reducing the amount of overhead required to use external storage for buffering; and (ii) as a by-product of external buffering, we want to use these "*persistent queues*" to efficiently record detailed provenance information on the inputs and outputs of each component invocation. Although many systems provide such provenance information (e.g., [4], [10], [12]), we show how this information can be captured efficiently using the external buffering approaches presented here, especially compared to the conventional approaches employed within many workflow systems. Specifically, we show that pipeline parallelism can provide performance increases over sequential scheduling, that external buffering overcomes issues with in-memory buffering, and that traditional approaches for provenance storage (based on relational database technologies) introduce significant overhead within data-intensive workflows that can be reduced by our framework and alternative implementations presented here.

The rest of this paper is organized as follows. In Sec. II we describe preliminaries and related work with respect to dataflow process networks and their common scheduling and buffering approaches. In Sec. III we present our framework for enabling persistent queues within scientific workflow systems (based on the process network model of computation). In Sec. IV we describe three different implementations of the framework presented in Sec. III, which include a standard relational database approach, an approach based on MongoDB [16] for supporting fast reads and writes from persistent queues, and a simple, specialized file-based approach for leveraging sequential reads and writes. In Sec. IV we also describe our initial experimental evaluation of the framework and the different implementation approaches we consider. In Sec. V we summarize our contributions and briefly discuss future work.

II. Preliminaries and Related Work

Dataflow process networks. In this paper, we adopt *process networks* [13] as a model of computation for representing dataflow-oriented workflow systems. Process networks form the basis of the dataflow models used within Kepler [11], and are quite general in nature: a process network is a set of processes (also called *actors*) that execute either sequentially or in parallel, and are connected by first-in, first-out (FIFO) communication channels to form a network. Actors produce data elements (called *tokens*) and send them along communication channels, where they are consumed by the destination actor. Each actor can have zero or more input and output channels, and actors in a workflow may only communicate through these channels. When executed in parallel, each actor runs within a separate thread, placing data onto its output channels (via a `put` operation) and reading data from its input channels (via a `get` operation). If an actor attempts to get data from an input channel that is empty, it is blocked until data is placed on the channel by the corresponding upstream actor. It is typical to view channels as token streams (or token sequences) in which actors represent functions that map input streams to output streams.[1] A key property of process networks is that the number of tokens produced and their values are determined by the definition of the actors and the structure of the process network, and not by the specific scheduling of individual actors [13] (assuming the basic channel dependencies of the network are maintained). This means that a workflow modeled as a process network will produce the same results regardless of whether it is executed sequentially (i.e., one actor at-a-time) or in parallel.

Scheduling. There are two standard types of scheduling approaches used in process networks. In *data-driven* (or "eager") execution, each actor runs in a separate thread (and thus in parallel), and each actor executes (or is invoked) as soon as a sufficient number of data tokens become available. This approach provides the maximum (pipeline) parallelism possible within the network since each actor executes as soon as possible. However, data-driven scheduling can also result in workflow executions requiring *unbounded* queues (as in Fig. 1), in which no guaranteed maximum queue size exists for all possible runs of the workflow. In *demand-driven* (or "lazy") execution, the number of tokens to buffer can be

[1] Many workflow systems (e.g., [4], [10]) support incremental, streaming computation in this way, often using explicit lists or sequences of tokens.

reduced by deferring the execution of an actor until its output is needed by another actor. Demand driven schedules work by sequentially "pulling" data through the workflow: the last actor is activated first, which causes it to request tokens on its input channels; this activates the corresponding upstream actors, which causes these actors to request tokens on their input channels; this in turn causes their upstream actors to become activated; and so on, until a source (or token-producing) actor is reached. Depending on the token consumption-production rates of actors (i.e., the number of tokens required to invoke an actor and the resulting number of tokens produced by the invocation) and the structure of the network, a demand-driven execution may still require queues of unbounded size. Alternatively, for many cases where a fixed buffer size is possible, synchronous dataflow (SDF) [17] can be used to statically determine (via the workflow structure and actor consumption-production rates) a fixed, sequential schedule that guarantees bounded channels. A number of "hybrid" approaches have also been developed that combine these different approaches. These algorithms aim at helping minimize queue sizes while allowing some actors to run concurrently [13] (e.g., such an approach is used instead of pure data-driven execution within Kepler).

However, these hybrid approaches can be complex to implement (often leading to deadlock issues), cannot guarantee unbounded queues, and can only minimize queue sizes in certain cases at the cost of reducing (or even eliminating) pipeline parallelism. Because scientific workflows often employ non-trivial, computationally expensive actors, increasing parallelism becomes much more important than decreasing queue sizes. This is especially true within data-intensive workflows where the amount and size of data to be processed is already large and difficult to manage using in-memory approaches [2], [14]. For instance, in Fig. 1, a sequential schedule would add 4 times the cost of executing actor A to complete the workflow. This cost increases with the number of actors in the workflow and the time required to execute each actor. Our goal is to fully embrace data-driven execution to maximize pipeline parallelism, while using external storage to minimize the problems associated with in-memory queues (where boundedness is less of an issue).

Data and pipeline parallelism. Pipeline parallelism is only one type of concurrency that can be used to increase efficiency in scientific workflows. For instance, a number of approaches have explored the use of data parallelism by executing and scheduling workflow pipelines using the MapReduce framework [14], [18], [19] or using specialized approaches [3]. While in many cases these approaches can significantly improve workflow execution by concurrently executing individual actors, there is still a need within these systems to incorporate pipeline parallelism. As discussed in [15] and [14], the standard approach used to implement pipelines in MapReduce is to serially execute one MapReduce "program" per actor, i.e., as a sequential schedule without any data streaming between actor invocations. We see the work presented here as complementary to data parallel approaches as well as being beneficial within conventional execution approaches typically used within systems such as Kepler and Taverna.

Provenance. A typical approach for collecting provenance of scientific workflows is to layer provenance recording on top of existing workflow execution (e.g., [9], [10], [12], [20]). This means in-memory buffering is used together with a separate step that detects and stores the inputs and outputs of actor invocations, where storage of provenance information is often within a relational database system. (Similar approaches are also used to facilitate fault-tolerance in workflow systems [21].) In general, these approaches incur both the negative effects of in-memory buffering and the overhead caused by external storage. In contrast, with efficient implementations of persistent queues (i.e., external storage of channel buffers), these separate steps can be combined to help minimize the overhead of storing provenance information during workflow execution.

III. Persistent Queues for Scientific Workflows

Fig. 2 shows the different buffering strategies we consider. Fig. 2(a) and 2(b) represent the conventional approaches for implementing channels in demand and data-driven scheduling, respectively, where in-memory queues are used to buffer data tokens; whereas Fig. 2(c–e) represent the different strategies we consider for implementing persistent queues within data-driven (i.e., pipeline parallel) scheduling. We discuss advantages and disadvantages of each persistent-queue strategy below, and in the following section describe the external storage approaches we have implemented for the different strategies of Fig. 2.

Basic persistent queues. Fig. 2(c) represents a straightforward approach for persistent queues in which the channel is effectively replaced by external storage (e.g., a relational database, as shown in the gray box denoting the modified channel). In this case, a `put` operation made by an invocation of the source actor A inserts a data token into the underlying external queue. Each data token is assigned a *put order*, which is a unique value denoting the order the token was added (via the `put` operation) to the queue. When a `get` operation is made by an invocation of the destination actor B, the channel retrieves from external storage the data token corresponding to the next item in the queue (e.g., in the case of a relational database implementation, the token would be retrieved by issuing an appropriate SQL query). Once a data token is retrieved from external storage, it is not removed from the queue. Instead, all data tokens added by a `put` operation are maintained within external storage, e.g., as part of the provenance record of the workflow. Thus, the channel maintains a *last put* and a *next get* variable that stores the put order of the last token added to the database and the put order of the next token to retrieve from the database, respectively. Once a token is returned by the `get` operation, the *next get* variable is incremented. This means that an invocation of B is blocked whenever the *next get* value is larger than the *last put* value. The channel is initialized such that *last put* has the value 0 and *next get* has the value

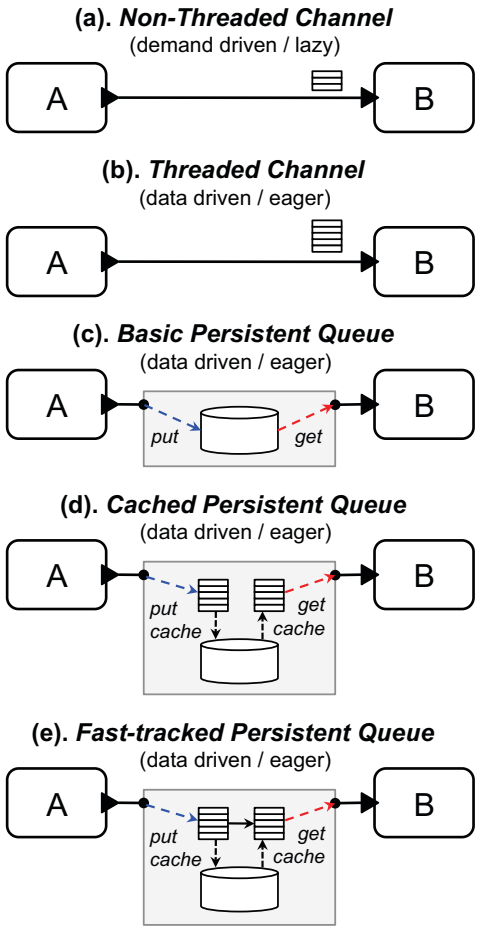


Fig. 2. Strategies for implementing channel buffers: (a-b) standard in-memory buffers used in demand-driven and data-driven execution; (c) a straightforward persistent queue implementation where external storage completely replaces the channel; (d) extension of (c) where read/write tokens are cached in memory prior to being placed in external storage; and (d) the use of a "fast track" within the channel to help further reduce overhead of using external storage.

1 (when the first token added to the queue is assigned a put order value of 1).

Cached Persistent Queue. The basic persistent queue of Fig. 2(c) adds significant overhead (depending on the external storage implementation used) onto the `put` and `get` operations, since each call has the additional cost of accessing external storage as well as serializing and deserializing data tokens, respectively. Compared to an in-memory queue, this can considerably slow down each `put` and `get` operation. To help avoid some of this cost we add in-memory put and get caches to the channel, as shown in Fig. 2(d). When a `put` operation is made by A, the data token is immediately stored within the put cache. The put cache is controlled by a separate thread that removes tokens from the cache (e.g., once the cache is close to filling) and adds them into the external queue. In this case, unless the put cache reaches capacity, A does not have to wait until a data token is serialized and added to the external queue to continue its execution. A get cache is used in a similar way, namely, a separate thread loads data tokens into the get cache (e.g., once the cache is close to empty), allowing calls to `get` to simply retrieve and remove data tokens from the cache. Thus, assuming there are tokens available in the get cache, B does not have to wait for tokens to be retrieved and deserialized from external storage to continue its execution. The use of put and get caches can additionally help reduce overhead by using "bulk" reads and writes to external storage, i.e., by batching updates and retrieving multiple tokens within a single query, which for many systems can be much faster than inserting and retrieving single tokens at a time. Unlike the in-memory queues of Fig. 2(a) and 2(b), put and get caches are only used here for caching: the size of the caches can be specified prior to workflow execution (as a tuning parameter) and are guaranteed not to overflow since they are always backed by external storage.

Fast-tracked Persistent Queue. In Fig. 2(e) we introduce a special *fast-track* pipe between the put and get caches on a channel to further reduce the overhead of external storage. The basic idea of a fast-track is to allow tokens to move directly from a put cache to a get cache without B having to wait until the token is sent through external storage. Assuming the put cache has not been written out to external storage before B requests tokens stored within it, the fast track allows B to use tokens produced by A without having to wait for them to be serialized, added to external storage, retrieved from external storage, and deserialized. Once a token has been fast-tracked, it continues to remain in the put cache until it is written to external storage. Once used by B, the *next get* value on the channel is still incremented (thus, with fast-tracking, the *next get* value can increase well beyond the *last put* value, depending on cache sizes). Similar to the put and get caches, implementing a fast-track requires adding an additional thread to each channel. While the put and get cache can allow A and B to execute faster independently (e.g., by not requiring A to wait for the token to be placed in external storage before calling `put` again), the fast-track allows A and B to execute faster "in concert" (i.e., during direct communication on the channel).

IV. Implementation and Evaluation

To verify and evaluate the different approaches for implementing channels discussed in the previous section, we have developed a simple workflow engine based on the dataflow process network model. The engine was developed in Java, and provides an API for specifying actors and workflows, which largely follows the approach used within the Kepler system (although our implementation contains many fewer features). While our system was developed primarily to experiment with different channel implementation strategies, it can easily be incorporated within Kepler to provide more flexible queuing strategies. A main feature of our system is that it was designed to easily support different external storage approaches for implementing persistent queues. Below we describe the different underlying storage systems we use, and also present

the results of our initial experimental evaluation comparing the different strategies for implementing channels (including the use of persistent queues).

A. Underlying Storage Systems

As mentioned above, the workflow engine we developed allows different backend storage technologies to be selected independently from the specific strategy used to realize persistent queues. Thus it is possible to select, e.g., a fast-tracked persistent queue strategy independently of the external storage system to use. The primary reason for adding this flexibility was to test how different types of external storage technologies performed. However, this approach also allows for workflow developers (e.g., using Kepler) to be able to configure a workflow to use a specific external storage system of their choice, which may be beneficial in some situations, e.g., for provenance applications that rely on specific external database systems. The system currently supports the following three external storage approaches.

Relational storage using MySQL. In this approach, each channel is represented by a table within a relational database. We used MySQL for our tests, however, any relational system supporting JDBC can be used. Storing and retrieving tokens from the database is performed using SQL statements. Besides being an obvious choice for external, persistent storage, relational databases are used in many workflow systems as a mechanism for storing provenance information (although some systems also store provenance using semantic web technologies, while a few offer a choice between relational and file-based storage [20]).

Non-relational database system using MongoDB. One potential issue with using a backend relational database for persistent queues is the overhead that can occur with database reads and writes. To help minimize this overhead, in this approach each channel is represented as a separate collection within a MongoDB database. MongoDB [16] is one of a number of non-relational (essentially nested key-value pair) database systems that optimizes for fast reads and writes (and scalability). MongoDB also has an advantage in that it is extremely easy to setup and use, and provides rich query support.

File-based external token storage. As discussed in the previous section, persistent queues access external storage using a simple pattern in which tokens are *sequentially* read and written from the channel. To further minimize the overhead of channel reads and writes, we developed a custom, file-based storage approach. In particular, each channel is associated with a separate file. Instead of using *last put* and *next get* variables, a file output stream and input stream is used, respectively. Tokens are added to the queue by appending them to the end of the file using the file output stream. The file input stream reads tokens from the queue, and is always positioned at the associated *next get* token (if there is one). Reads are especially fast using this approach since we do not have to "search" for a token with a given put order.

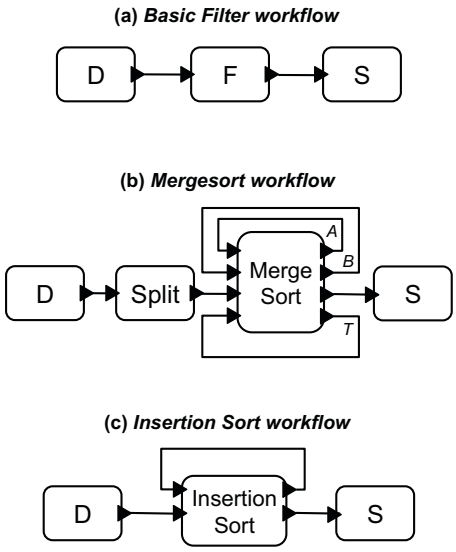


Fig. 3. Workflows used to evaluate the different channel strategies: (a) a simple workflow for filtering token streams; (b) a workflow for sorting a token stream using mergesort, where all intermediate lists are stored in channels; and (c) a workflow for sorting using insertion sort, again with all intermediate lists stored in channels.

Additionally, for MySQL and MongoDB, we also implemented versions that perform batch reads and writes (see Sec. III).

B. Experimental Evaluation

To evaluate the effectiveness of using persistent queues we performed a number of tests comparing the above strategies for implementation channels and using different backend storage systems. Our main goal in evaluating persistent queues was to gain a better understanding of the overhead of these different approaches, with the overall aim of minimizing the overhead of using persistent queues as much as possible in data-intensive workflows.

To explore the overhead of persistent queues, we used the following simple workflows within our prototype workflow engine (see Fig. 3). Each workflow contains a data source and sink actor. The data source actor reads an unordered sequence of dictionary words line-by-line from a file, converts each word into a data token, and passes the token on to downstream actors. The data sink actor simply takes a token containing a word as input, and appends the word to an output text file (which is also used to verify the workflow output is correct). To make the workflows somewhat more realistic in terms of processing input and output, the source and sink actors have small delays to better simulate work being done.

Simple Filter. As shown in Fig. 3(a), this workflow consists of a source actor, a simple filter actor, and a sink actor. The filter actor selects tokens (based on a given condition) received by the source actor and passes them on to the sink actor. Below we consider two cases for the filter workflow. In the first case, the filter actor acts a simple pass-through that sends

all tokens received to the sink actor. In this case, the filter actor has a configurable delay that can be used to simulate processing. The delay is implemented as a simple busy loop. In the second case, the filter actor checks if the word in a given token begins with the letter 'c' (to reduce the number of tokens sent to the sink actor). This workflow does not put strain on channel queues, i.e., queue sizes are kept small since the source, filter, and sink actors take close to the same amount of time to execute and send and receive one token at a time. Instead, this workflow is largely used to evaluate the overhead of the different strategies when a small amount of queuing is required. This workflow, which is similar to the example in Fig. 1 (but with one additional actor), is also used to compare data-driven (pipeline parallel) execution to demand-driven (sequential) execution.

Mergesort. As shown in Fig. 3(b), this workflow consists of a source actor, a split actor, a mergesort actor, and a sink actor. The workflow sorts the word list alphabetically and stores the result in a new word list file. The source actor connects to a split actor that inserts a special split token between every data token. The split token is used to break the input list into initial sublists of length one. The resulting list is then passed to the mergesort actor, which sorts the list using three channels (shown as A, B, and T in the figure) that each connect back to the merge actor. The mergesort actor reads one sublist into the temporary channel T. It then merges the sublist on T with the first sublist on channel A, sending the merged list onto the third channel B. When the entire list has been merged, the mergesort actor switches "directions" and merges into the channel that has just been emptied (in this case A). When there is only one sublist left, the resulting sorted list is sent to the sink actor. In terms of the use of channels, this workflow primarily reads from one channel while writing to another channel (as opposed to switching between reads and writes on a single channel). However, although extremely simple, this workflow requires a large number of tokens to be placed within channel queues since the intermediate results of the sort are being "stored" in the channels. Thus, the mergesort actor simply performs comparisons and token routing, relying on the channels to manage data tokens.

Insertion sort. As shown in Fig. 3(c), this workflow consists of a source actor, an insertion sort actor, and a sink actor. Like mergesort, this workflow sorts the wordlist received by the source actor. Each time the insertion sort actor receives a new (unsorted) token, it spins through its self-connecting channel until it finds the location where the token should be inserted by getting a token off the channel, performing a comparison, and then putting a token back onto the channel. When it receives an end token from the source actor, it sends the entire sorted list to the sink actor. In contrast to the mergesort workflow, this workflow interleaves reads and writes on its self-connecting channel. However, like mergesort, the insertion sort workflow requires a large number of tokens to be stored within channel queues.

Figures 4 through 6 summarize the results of our tests for

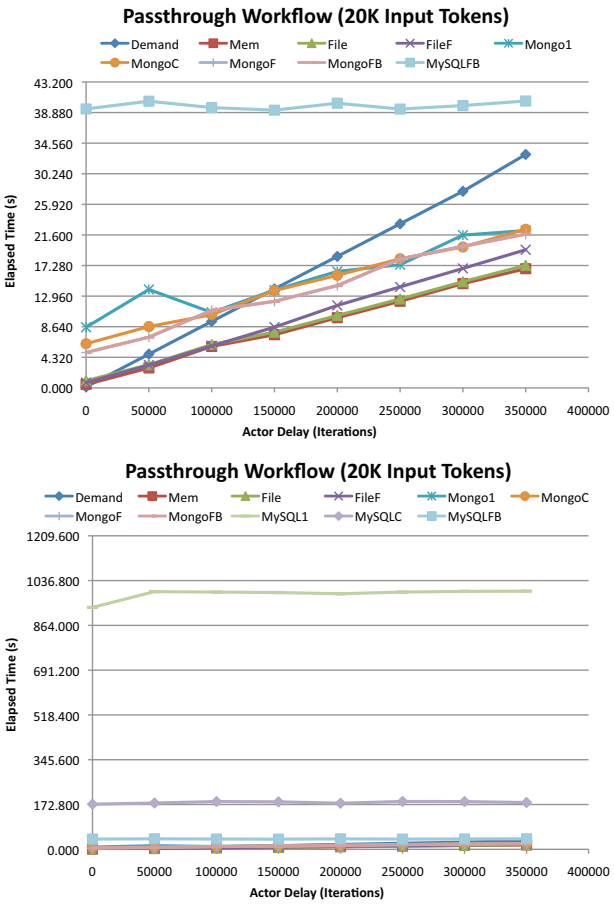


Fig. 4. Total execution times for a "passthrough" filter workflow run over 20,000 input words using different delay settings (as shown, loop iterations). The top graph only shows one of the MySQL approaches, whereas the bottom graph shows each approach.

each of the above workflows. Each experiment was performed using a dual-core 3.0GHz Intel Core 2 vPro PC running Windows 7 with 3GB of RAM and 20GB of free disk space. We used MongoDB version 1.6.3 with version 2.1 of the Java driver, and MySQL server version 5.1 with version 5.1.13 of the MySQL JDBC driver.

The top of Fig. 4 shows the results of running the first version of the filter workflow (where the filter simply passes along received tokens) over a fixed number of input words (20,000). Each approach was run over different delay settings in the workflow given by the number of iterations in a busy loop (for each actor). On average, the busy loop creates a delay of between (approximately) 0 and 2 ms. As shown in Fig. 4, after only a small delay is created, the execution time of the demand-driven workflow quickly surpasses the data-driven approaches, except for the case of MySQL. In addition, shortly after this delay, each of the non-MySQL data-driven approaches begin to converge, with in-memory queues (Mem) being slightly faster than the file-based storage, followed by MongoDB. Thus for even a very

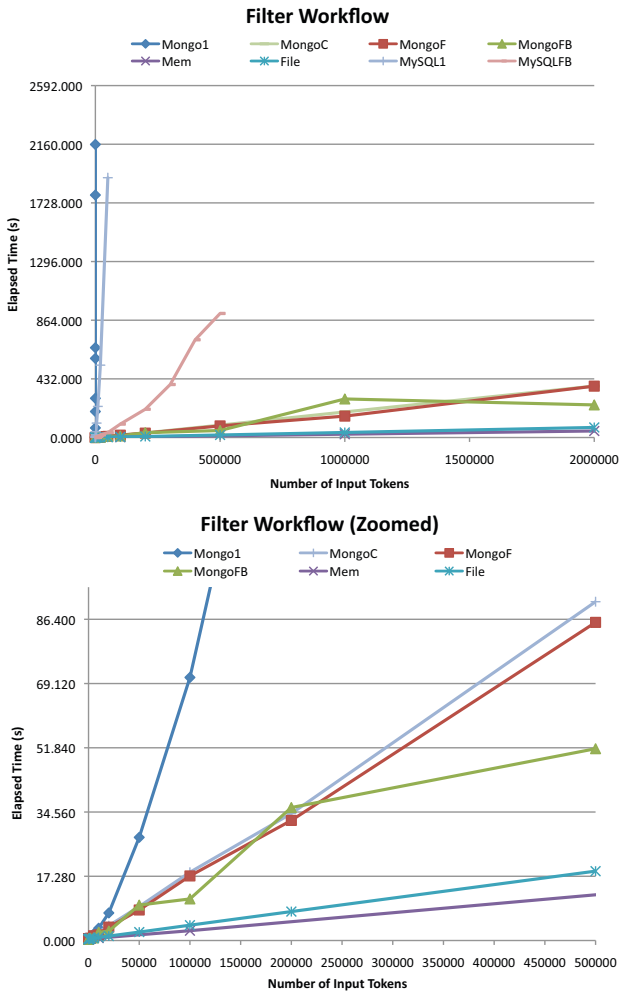


Fig. 5. Results of the different strategies and storage approaches for the filter workflow (top) with a detailed view showing the first 500K tokens (bottom).

simple workflow with a relatively small input data set, we can quickly see the advantages of pipeline parallelism over sequential scheduling. With MySQL, however, the overhead of reads and writes eliminates the performance gains of pipeline parallelism, even when batching and a fast-tracking are used. The bottom of Fig. 4 shows the MySQL approaches in relation to the other approaches. Here we can see that caching (MySQLC) provides a significant benefit over the simple buffer approach (MySQL1), which is still considerably slower than using MySQL with batch reads and writes and the fast-track (MySQLFB). Alternatively, in the top of Fig. 4, the file-based storage approach (File) has very little overhead compared to using in-memory buffers.

The top of Fig. 5 shows the overall results of running the second version of the filter workflow (where words beginning with the letter 'c' are passed along) over increasing numbers of input words, ranging from 50 to 1 million. The bottom of Fig. 5 shows the same graph, but only for the first 500,000 tokens (and without the MySQL results). This graph shows similar results as those of Fig. 4, except that the basic MongoDB channel (Mongo1) performs slightly worse than the basic MySQL channel (MySQL1) and significantly worse than the MySQL channel with batch reads and writes and the fast-track. However, MongoDB used with caching (MongoC), fast-track (MongoF), and fast-track with batch reads and writes (MongoFB) significantly outperforms the MySQLFB approach (where MongoFB appears to be the most efficient of the four MongoDB channels shown). As in Fig. 4, in-memory data-driven execution is the fastest, but is only slightly faster than the customized, file-based external storage approach.

Fig. 6 shows the results (both summarized and zoomed in) of the mergesort and insertion sort workflows for MongoDB, the custom file, and in-memory data-driven approaches. As above, MySQL is considerably slower than these approaches, and so not shown in the graphs. In both workflows, the file-based approach using a fast-track (FileF) outperforms the simple file approach (which was not the case with the passthrough and filter workflows), and is very close to the in-memory approach. In fact, for insertion sort, FileF does slightly better than in-memory queues. For the mergesort workflow, we see the two issues arise with using in-memory buffering for data-intensive workflows: (1) after approximately 7.5 million input tokens, we begin to see a sharp slowdown (due to paging), and (2) after 10 million input tokens, the system crashes (because of memory limitations). However, in both file-based approaches (as well as the MongoDB approaches) neither of these cases occur.

Taken together, these tests (although limited in scope) demonstrate that using persistent queues is both feasible, and especially for data-intensive workflows, incurs only a small amount overhead (especially when caching, fast tracked channels, read/write batching, and optimized storage approaches are used), while gaining the benefits of using external storage for channel queues. These benefits can include provenance storage "for free" as well as the ability to run data-intensive workflows without in-memory performance issues due to paging or overflowing buffers, as was the case in the mergesort workflow. While the tests above were targeted at isolating certain behavior (e.g., relative actor delays, actors that process large numbers of data items but with small to large queuing needs), many real-world scientific workflows will have a large number of actors and connections that mix these general patterns. In more complex cases, the overall time required to execute actors, the amount of time each actor requires in comparison to other actors, and the amount of data both being passed and that must be queued within the workflow will each contribute to the overhead and performance benefits of using external storage for implementing workflow channels.

V. Summary

We have described approaches for using external storage to implement persistent queues in dataflow process networks. The goal of this work is to improve and extend current support for data-intensive scientific workflows in two primary

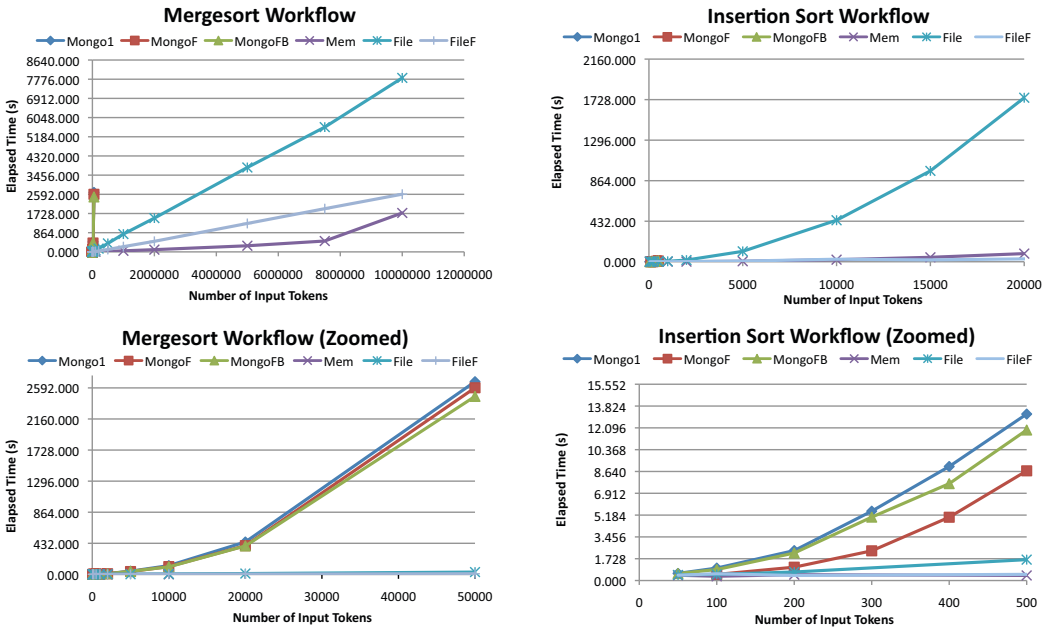


Fig. 6. Result of mergesort (top left) with first 50K tokens (bottom left) and insertion sort (top right) with first 500 tokens (bottom right).

ways: to overcome issues with in-memory based queues for enabling pipeline parallelism; and to reduce the overhead of using external storage of data tokens during workflow execution. We presented a general framework for reducing overhead in channels using external storage, compared different external storage strategies including relational and non-relational database systems and a custom file-based approach (for fast sequential reads and writes). Our results show that the framework is feasible and can reduce overhead when using external storage for implementation channels, making it close to (and sometimes more efficient than) in-memory queues. Our results can also be used to improve the performance of existing provenance recording techniques by integrating persistent queues with provenance collection and storage. As future work we intend to add persistent queues to Kepler and explore ways to combine our approaches with data parallelism.

Acknowledgements. This work was funded in part by the Gonzaga University Research Council through a Faculty Research Award.

REFERENCES

[1] D. Barseghian, *et al.*, "Workflows and extensions to the kepler scientific workflow system to support environmental sensor data access and analysis," *Ecological Informatics*, vol. 5, no. 1, pp. 42–50, 2010.

[2] A. L. Hartman, S. Riddle, T. M. McPhillips, B. Ludäscher, and J. A. Eisen, "Waters: a workflow for the alignment, taxonomy, and ecology of ribosomal sequences," *BMC Bioinformatics*, vol. 11, p. 317, 2010.

[3] F. Coutinho, *et al.*, "Data parallelism in bioinformatics workflows using hydra," in *HPDC*, 2010, pp. 507–515.

[4] S. Bowers, T. M. McPhillips, S. Riddle, M. K. Anand, and B. Ludäscher, "Kepler/ppod: Scientific workflow and provenance support for assembling the tree of life," in *IPAW*, 2008, pp. 70–77.

[5] K. Wiley, A. Connolly, J. P. Gardner, S. Krughof, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu, "Astronomy in the cloud: Using mapreduce for image coaddition," *CoRR*, vol. abs/1010.1015, 2010.

[6] K. Maheshwari, C. A. Goble, P. Missier, and J. Montagnat, "Medical image processing workflow support on the egee grid with taverna," in *CBMS*, 2009, pp. 1–7.

[7] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 519–529, 2008.

[8] A. Lerner and D. Shasha, "The virtues and challenges of ad hoc + streams querying in finance," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 49–56, 2003.

[9] E. Deelman, D. Gannon, M. S. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Comp. Syst.*, vol. 25, no. 5, pp. 528–540, 2009.

[10] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. A. Goble, "Taverna, reloaded," in *SSDBM*, 2010, pp. 471–481.

[11] B. Ludäscher, *et al.*, "Scientific workflow management and the kepler system," *Conc. and Comp.: Pract. & Exp.*, vol. 18, no. 10, 2006.

[12] B. Bavoil, *et al.*, "VisTrails: Enabling interactive multiple-view visualizations," in *IEEE Visualization*, 2005.

[13] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, 1995.

[14] D. Zinn, S. Bowers, S. Köhler, and B. Ludäscher, "Parallelizing XML data-streaming workflows via mapreduce," *J. Comput. Syst. Sci.*, vol. 76, no. 6, pp. 447–463, 2010.

[15] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "Haloop: Efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1, pp. 285–296, 2010.

[16] "MongoDB," http://www.mongodb.org/.

[17] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[18] X. Fei, S. Lu, and C. Lin, "A mapreduce-enabled scientific workflow composition framework," in *ICWS*, 2009, pp. 663–670.

[19] J. Wang, D. Crawl, and I. Altintas, "Kepler + hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems," in *SC-WORKS*, 2009.

[20] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the kepler scientific workflow system," in *IPAW*, 2006, pp. 118–132.

[21] D. Crawl and I. Altintas, "A provenance-based fault tolerance mechanism for scientific workflows," in *IPAW*, 2008, pp. 152–159.