Approaches for Implementing Persistent Queues within Data-Intensive Scientific Workflows

Michael Agun, Shawn Bowers
Department of Computer Science
Gonzaga University
Overview

Data-intensive scientific workflows

– Different “components” to wrap external programs/scripts
– Components handle large numbers of data “tokens”
  • Tokens can hold a variety of data objects
  • E.g., numerical data, gene sequences, images, phylogenetic trees

Dataflow model of computation

– Used in many workflow systems (e.g., Taverna & Kepler)
– Simple and formal semantics
– Supports branch/merge, conditional execution, iteration
– As well as streaming data and pipelined execution …
Overview (cont.)

In dataflow models (with streaming/pipelining)

- Workflow components can execute *concurrently*
- Can provide more efficient workflow execution
- But requires *token buffering* on dataflow channels
- Current systems buffer using *in-memory queues*
- Can lead to *performance issues* (buffer overflow, paging)

A outputs a *sequence* of tokens $x_1, x_2, \ldots, x_n$
Overview – Process Networks

We consider dataflow *process networks*

– Generalizes most dataflow-oriented workflow systems

• Workflow graphs (networks)
  – *Actors* … processes that read/write data tokens
  – *Channels* … first-in first-out communication edges

• Actor input and output
  – Actors *read* from channels using a *get* operation
  – Actors *write* to channels using a *put* operation
Overview – Scheduling Process Networks

Actors can be executed *sequentially or in parallel*

– Same result regardless of scheduling approach

**Data-driven** (eager) scheduling

– Each actor runs in a *separate thread*

– Max concurrency, but potentially *unbounded* buffers

**Demand-driven** (lazy) scheduling

– Defer actor execution until output needed

– Work by sequentially “pulling” data through workflow

– Can reduce data buffering, but also reduces concurrency

Many *hybrid* algorithms (more concurrency/more buffering)
Overview (cont.)

Unbounded buffers …

– E.g., B could be *twice as slow* as A
– or A could output more tokens per invocation

During each $B_i$ invocation $A_{2i}$ and $A_{2i+1}$ execute

The buffer must increase by 1 on each B invocation
Overview (cont.)

The goals of this work

1). Develop strategies to use external storage for buffering
   - Allow maximum amount of (pipeline) concurrency …
   - … while avoiding out of memory “crashes”
   - … and performance penalties of high memory utilization

2). Reduce overhead (reads/writes) of external buffering

3). Store queues persistently (for recovery & provenance)
Persistent Queue Strategy I

Basic Persistent Queue
(data driven / eager)

Replace channel with external store (e.g., relational DB)

- A put call adds a token to the store with a “put order”
- A get call obtains the next token token … without removal
- A “last put” variable to hold last token added (initially 0)
- A “next get” variable to hold next token to get (initially 1)
- Block get call whenever last put < next get
Persistent Queue Strategy II

Cached Persistent Queue
(data driven / eager)

Reduce put/get overhead

– put/get calls require waiting for token read/write into store
  • Depending on storage technology, can add significant overhead

– Add intermediate fixed-size caches
  • put adds token to in-memory put cache
  • Separate thread flushes cache to external queue
  • Works similarly for get calls

– Can also be batched (bulk reads/writes)
Persistent Queue Strategy III

**Fast-tracked Persistent Queue**
(data driven / eager)

Further reduce put/get overhead

- get calls must still read from storage to obtain tokens
- Add a “fast track” pipe between caches
  - Tokens in put cache can be immediately added to get cache
  - Fast-tracked tokens still written to persistent queue (remain in cache)
  - Fast-track implemented as a separate thread
- Actor B no longer has to wait for all tokens to be serialized, stored, retrieved, and deserialized
Implementation

Dataflow engine in Java
- Primarily to verify/evaluate channel strategies
- General API for specifying actors and workflows
- Workflows configured with different channel strategies
- Supports both data and demand driven scheduling

Incorporated 3 external storage technologies
- Relational storage using MySQL
- “Non-relational” storage using MongoDB (fast read/write)
- Specialized file-based token storage (sequential read/write)
Experimental Evaluation

Source (D) and Sink (S) actors
- Read tokens from a file … dictionary of unsorted words
- Write resulting tokens to a file

Simple token passthrough actor (P) w/ 20K tokens
- A delay variable (ranging from 0 to 2 ms)
Experimental Evaluation

Select (F) words starting with ‘c’
  – Run over increasing numbers of words
  – MySQL approaches not shown (similar to previous)

---

![Filter Workflow (Zoomed)](image)

- **MongoDB w/ Fast-Track (batch)**
- **File storage (seq r/w)**
- **In memory**
Experimental Evaluation

Mergesort

- All state (temp lists) “stored” on workflow channels (A, B, T)
- Places heavy strain on channel queues

In memory starts better, begins to slow down, then overflows

File (w/ fast-track) grows linearly, without buffer overflow
Ongoing Work

Combining \textit{data} and \textit{pipeline} “parallelism”

– Extended current system with data parallel support
– Based on fork/join tasks (Java concurrency)
– Uses “naive” scheduling approach
– Exploring different scheduling approaches

Extensions for \textit{distributed} settings

– E.g., cluster of nodes
– Support data streaming both within and across nodes
Overview (cont.)

Simple case (no buffering needed)

– Each execution of A writes (produces) **one token**
– Each execution of B reads (consumes) **one token**
– B is **no slower** than A

B\(_i\) and A\(_{i+1}\) execute concurrently