Automatic n-Buffering for Big Data processing

Asbjørn Thegler

October 2015

Abstract

WRITE LAST

During the research, it was discovered that using three buffers was an arbitrary constraint, and concluded that many use cases would benefit from a different number of buffers.

Contents

1 Introduction

It is hardly a secret that Big Data has become a huge topic over the last few years. Huge companies worldwide compete on this new trend, and searching for 'big data trend' on Google reveals how popular this topic is, and is predicted to be, for years to come. Forbes, CIO, ComputerWorld and Gartner all predict increases in the Big Data industry. (More)

There are many definitions to what Big Data really is. The truly new aspects of Big Data is to have large enough datasets to be able to find and recognize patterns that previously were too vague or noisy to find, using only smaller datasets. When traditional data processing techniques grow inapplicable, we must gain new knowledge to find methods to work with Big Data.

(Moore's Law for data growth? Harddisk space?) Having data doesn't make any interesting results. The data must be processed, analyzed, and scrutinized. This is no small task, and given that many new measurement tools generate tera-bytes of data per hour, back in 2013, we need to have mechanisms in place for analyzing the data in a similar rate.

(Should this be past tense?) The first popularly known occurrences of the term 'Triple Buffering' stems from the computer graphics industry. This is a technique where the graphics card renders images into 3 different buffers. Previous to this technique, a double buffer was used. A 'front-buffer' and a 'back-buffer'. The renderer would render a frame into the back buffer, and the buffers would swap. Since no synchronization exists between the renderer and the consumer, a buffer swap could happen in the middle of consumption, resulting in what is known as 'screen tearing'. Some attempts at fixing this problem is know as 'vertical sync' or 'vSync'. This included adding artificial delays to the renderer, to match the frame rate of the consuming screen.

To better solve this problem, a third buffer is employed, effectively making it 'Triple Buffering'. The renderer could now switch between two back-buffers, and always have a free buffer to write a new frame to. If the consumer was too slow, frames would simply be lost, with no greater loss to the viewer. This obviously require extra memory on the graphics card.

'Triple Buffering' within computer graphics is a different thing, though there are many similarities. We can translate some of the solution to the field of Big Data. The bottlenecks of a graphics card are namely the rate of the renderer and the rate of the consumer. This translates to the IO problems we encounter when working with Big Data. The graphics industry solved the problem by utilizing more space, in the shape of an extra buffer and in theory, this can solve, or at least mitigate, some of the IO problems related to Big Data.

The focus of this thesis project is to produce a framework or library that enables programmers to process or transform large amounts of data in an efficient and concurrent way, without having to worry about concurrency issues and memory management. The framework or library should be generic, such that it is as generally applicable as possible, while still being useful and simple enough to understand for people who aren't familiar with multiprogramming. It is important to note that this project in no way introduces new technology or

uncovers scientific ground. This is a study in working with existing technology to create a highly optimized and effective library.

1.1 Motivation

This project didn't manifest from thin air. Many people have probably seen it coming from a long distance. Triple Buffering has been used many places, many times before, and it is a well-known term. The reason why this project was started now, and not 10 years ago or in 10 years, is a combination and collision between Moore's Law, data bus speed and the growing Open Source community, both within academia, but also within established industries.

1.1.1 The IO Problem

When processing data, it is relatively easy to increase the amount of computational resources, but moving data to and from the computational resources results in IO, which quickly becomes a bottleneck. To process data as fast as possible, we want to squeeze as much effect out of every available resource. When processing data sequentially, the traditional method is to;

1. Load data into buffer from disk
2. Process or transform data
3. Write data back to disk
4. If there is more data; go to 1

If the computational task of processing or transforming is large enough, the IO becomes negligible. If the computational task is very small, most of the execution time will be spent waiting for IO. In the latter case, there are two notable resources, which are being used in turn, namely the *input stream* and the *output stream*. This means that only half of the resources are being used at any given moment. In this case, we could have two concurrent workers, each swapping what resource they use, to utilize all of the resources at any given time.

When the IO becomes negligible, we have a very compute-heavy task. In this case, we can attempt to add more compute resources, which in turn, will shorten the execution time. This means that we could have 5, 10 or 50 buffers, depending on how large the computation task is, in comparison to IO.

(More?)

1.1.2 The Generic Problem

When programmers and developers write software, they are generally encouraged to utilize established libraries as much as possible, instead of relying on their own ability to create elaborate and correct code. Often, a programmer has to solve a specific problem, which can be translated into a general problem which has already been solved multiple times. The productivity of the programmer can increase greatly, when using tested and accomplished libraries.

Some topics are inherently difficult for programmers, such as memory management and concurrency, often leading to memory leaks and race conditions. When using established libraries, these problems are often already addressed.

Within the Open Source community, it is common practice to make ones code available for others to use. When multiple entities utilize the same code or library, bugs and race conditions are found, reported and corrected much faster than when code is only used privately (link?). Over time, this often result in libraries that are used globally, and has many contributors.

When a library does not exist for the specific problem, programmers must solve the problem themselves. This will result in many programmers solving the same problem over and over again. At some point, someone will see the pattern and pick up the task, and attempt to build a generally applicable library.

There are some pitfalls to using libraries to solve tasks. When a problem is simple, using a complex library might be too much work, since many libraries has a ton of options that might be relevant. Reading the 'man'-page of any Linux tool can be a daunting task, while often simple problems can be solved faster in other ways. Also, Open Source projects tend to have organic growth. Without tight steering from some small group of committed developers, a project will be monolithic and many people will classify it as 'bloatware'.

1.1.3 Use Cases

The intended library can be used for several specific purposes. Many places, large amounts of data are being processed. Following are a few use cases where using such a library is indeed a good solution.

Hash algorithms are designed to be compute-heavy. When hash-values are needed on very large local files, it would be more efficient to use a n-buffering mechanism, and add buffers until the IO again becomes the bottleneck. The command line tools md5sum and sha512sum both have implementations that read 512 bytes at a time, which results in many IO operations which could be avoided, if more memory is available for multiple buffers. Gathering statistics on sensor data is also a brilliant use case.

When large amounts of sensor data are received via a network, they are usually written directly to disk, before they get processed. In cases where much of the data is merely noise it could be good to have an option to process the data the instant it arrives, instead of waiting until after is has been written to disk. This can include gathering statistics, calculating hash values or filtering irrelevant data.

(More use cases?)

1.1.4 Big Data and Complexity

Complexity of algorithms doesn't matter with small data-sizes. Big Data makes complexity really important.

2 Theory and Analysis

This section will explain the ideas and thoughts that are used during the design and implementation of the framework. The project has two main topics.

First there will be reasoning about concurrency and correctness. How to ensure that the library will always terminate when used correctly.

Second, the project entails a lot of aspects related to data, IO and how to handle the enormous amounts of data.

Finally, I will reason on what results I expect to get from this project, in relation to solving some of the problems touched upon in my motivation.

2.1 Concurrency

Concurrency has proven to be hard for the human mind to understand, design and work with. When done wrong, software can easily include deadlocks or other race conditions. This section will explain some of the pitfalls of concurrency and how to avoid them.

2.1.1 Flow and Deadlocks

Concurrency done wrong can result in processes running out of control, or not running at all. The school-example used in teaching deadlocks to classes is known as the Dining Philosophers Problem, which was presented by E. W. Dijkstra in 1971. [Dijkstra, Hierarchical ordering of sequential processes]

Deadlocks and deadlock prevention is paramount when working with concurrency, but explaining why should be trivial at this point. I will suggest reading Concurrent Systems, by Jean Bacon [reference!], if you want to learn more on this topic. This report will assume thorough knowledge of how deadlocks can happen, and what measures can be deployed to avoid them, both theoretical and practical, but I will not explain further here.

2.1.2 Finite-state Diagram

Any process can be viewed as a finite-state machine. Doing so will help understanding the process, its possible states, and the triggers that will change the internal state of the process. This is known as the scientific body of "Automata Theory" and what I will elaborate on here, is a subset of this field.

To gain a better understanding of a finite-state machines, finite-state diagrams are brilliant for ensuring that the process at hand reacts and interacts as expected. The finite-state diagrams are trivial to both create and understand, and can be used for reasoning about a process, as a development tool and as documentation about a certain system or process. It gives an abstract idea of how a concrete process works.

In figure [DOOR EXAMPLE] there is an example finite state diagram. This diagram is quite simple, and shows how a door with a lock will behave. There are 3 states, "open", "closed" and "locked", and this finite set is often denoted as Q. This means that the door must be either open, closed or locked (in reality the door is both closed and locked at the same time, but I leave this out for

brevity). Further, there are 4 different events that can happen, ”open door”, ”close door”, ”lock door” and ”unlock door”. This set is called the ”alphabet” and is denoted $\sum$.

2.1.3 CSP

CSP is a formal language to describe patterns of interactions in concurrent systems. This can be used to prove that a system will react as expected.

2.1.4 Synchronization Primitives

future-promise mutexes

2.2 Data Handling

Working efficiently with data is no small task. There are many physical limits to what results we can obtain, but getting to these limits often requires a lot of thought, since there are many abstraction layers between hardware and software. This section will elaborate on how to work with these sizes of data in a correct and efficient manner.

2.2.1 Optimal Buffer Size

The size of a buffer

Buffer size should be less than the file size. If buffer sizes are 3*300M but file is 100M, then it is inherently sequential.

2.2.2 Data Marshalling

When receiving and sending data to and from a stream, care should be taken to correctly de-serialize and serialize data. (Google Proto-buffers)

2.3 Theoretical Speedup with threads

Using multiple threads will still only use one process, and can only run on one processor at a time. This limits the potential speedup to only include latency hiding.

2.4 Theoretical Speedup with processes

Using multiple processes, it will be possible to utilize all cores on a system. This greatly increases the potential speedup, in relation to only using threads.

2.5 Theoretical Speedup with devices

Using multiple devices, such as GPUs can greatly affect the computational power. When performing the same task on many pieces of data, one should always consider using a GPU, since it is massively parallel.

Current State	Input	Next State	Result
Initialization	*ready*	Read Wait	Worker is ready for reading, but has to wait for the read-resource.
Read Wait	*read available*	Read Critical	Worker can now read, which blocks other workers from this state.
Read Critical	*data read*	Execute	Worker read some data, and can now process it.
	nothing read	Exit	Worker read nothing, and the work is finished.
Execute	*done execute*	Write Wait	Worker has processed its data, but has to wait for the write-resource.
Write Wait	*write available*	Write Critial	Worker can now write, which blocks other workers from this state.
Write Critial	*done write*	Read Wait	Worker is ready for reading again, but has to wait for the read-resource.
Exit			No transition exists from the exit state.

Table 1: NBUF worker State Transition Table

3 Design and Implementation

This section will explain how the NBUF framework has been designed and implemented. First, I will elaborate on the abstract idea of how the framework handles concurrency. Then I will elaborate on how the framework is to be used, and finally how it has been built.

3.1 Abstract Overview

I will here give an abstract overview of how the workers in the NBUF framework will interact. Each worker has its own buffer which it will use for reading into, processing and writing from. This buffer is not shared with any other worker.

In [Figure NBUF worker State Diagram] is a finite-state diagram which shows how each worker in the system transfers from state to state. In Table 1 the related transition table can be seen. It is important to note that there are two critical states. These states are intended to mimic the importance of a critical section, as known from concurrent programming

A worker begins in the ”Initialization”-state and, it will move into the ”Read Wait”-state. When the single read resource is available, the worker will move into the critical ”Read Critial”-state. This state is exclusive, since only one worker can read at a time. At this point, two things can happen. Either, the worker receives data from the resource, or it does not receive data. The amount of data it receives does not matter, the buffer may be almost empty, or it may be full. In cases where it receives data, it will move to the ”Execute”-

state, and another worker can enter the ”Read Critical”-state. Now, the worker will crunch the data located in the buffer, and produce whatever output is desired. When the worker has finished processing, it will move into the ”Write Wait”-state. In this state, the worker will wait for the single write resource to become available. When it becomes available, the worker will move to the ”Write Critial”-state. When the worker has written the content of the buffer, it is moves to the ”Read Wait”-state, since it has finished its cycle, and can read new data into the buffer. At some point, the read resource has no more data, and the worker will not receive data during the ”Read Critical”-state. At this point, it will move to the ”Exit”-state, and stay there until thread termination.

3.2 API

3.3 Multithreading with *std::thread*

3.4 Multiprocessing with OpenCL

4 Experimentation and Benchmarking

5 Conclusion

Bringing knowledge together to create something useful

6 Future Work

6.0.1 IO throttling

not reading 1gb of data as a start, but starting low, and slowly increasing the buffer sizes.

6.0.2 Variable buffer sizes

When reading small files, allocating large buffers is unnecessary.

6.0.3 Slow network? How does istream handle that

The End

The learning goals for reference:

- Programming for and with Big Data // DONE
- Thoroughly understand and design generic synchronization // Eh..
- Reason about complexity in relation to Big Data // Scheduled under 'Motivation'
- Reason about IO problems // Scheduled under motivation and theory
- Design correct benchmarking // Scheduled under Experimentation and benchmarking