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 effi-

cient 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.

2.1 Concurrency

Concurrency has proven to be hard for the human mind to keep track of. 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. (Dining philosophers?)

2.1.2 State machine diagram

When working with multiple objects, state machine diagrams are brilliant for ensuring that the objects at hand react and interact as expected. The UML state diagrams are easy to both create and understand, and can be used both as a development tool and as documentation.

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.

2.2 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.

3 Design and Implementation

This section will explain how the framework has been designed and implemented. First, I will elaborate on the

3.1 Multithreading with POSIX

3.2 Multiprocessing with OpenCL

4 Experimentation and Benchmarking

5 Conclusion

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.
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