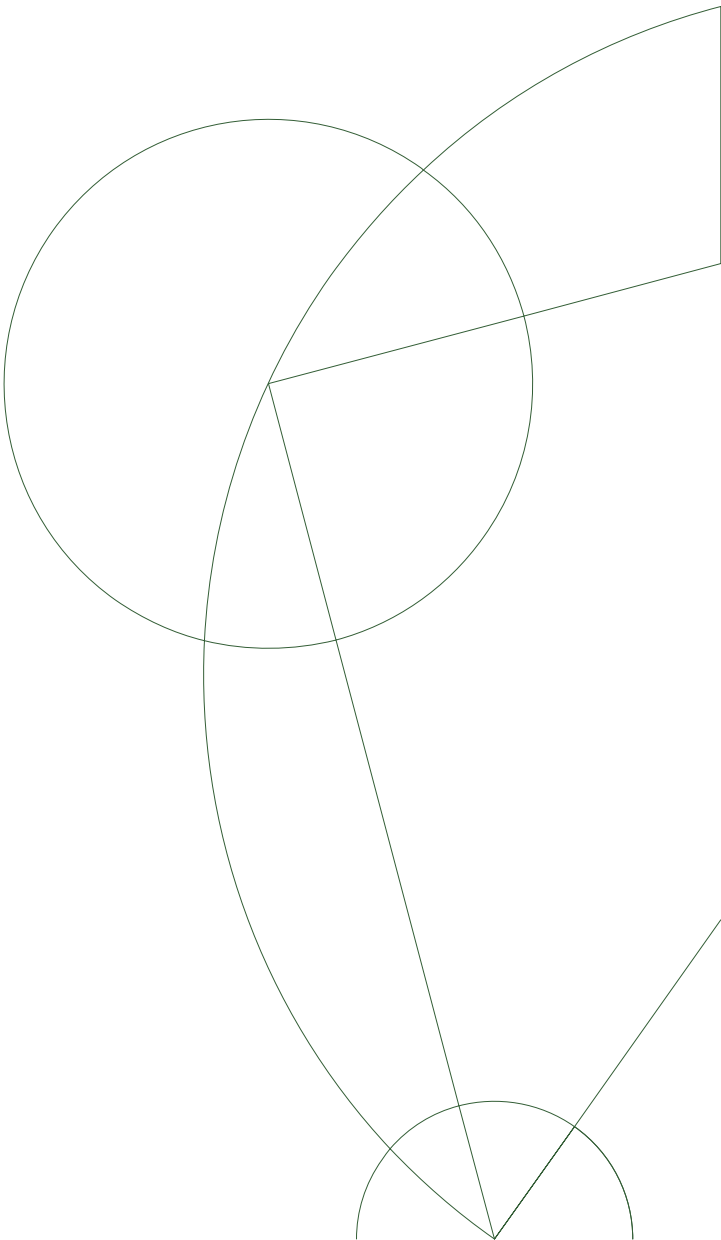
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Asbjørn Thegler - asbjoern@gmail.com

Automatic n-buffering for Big Data processing

Department of Computer Science



Professor Brian Vinter

October 2015

Abstract

This study attempts to construct a generally applicable library for efficient and parallel processing of data from generic data streams.

The *nbuf* library is intended to be used mainly for live processing of very voracious streams. This is useful when huge amounts of data is being generated and when processing those data must be done before archiving them in slow storage solutions.

The library has been built with the abstract synchronization primitives made available in C++11, which is based on POSIX threads. Further care was placed in making the library as flexible as possible for every thinkable scenario.

Benchmarking results shows that the library can fully utilize the rate of a data stream, when sufficient computational powers are available. In cases where there are additional constraints on the level of parallelism, the library exhibit brilliant latency hiding capabilities.

The library successfully enables developers to efficiently process streams of data in a concurrent manner, without extensive knowledge about multiprocessing and inter-process communication.

The *nbuf* library can be found at `http://github.com/ath88/nbuf`.

Contents

1 Introduction

Big Data has become a huge topic over the last few years. Google Trends reveal that the interest for 'big data' has exploded since 2012. Huge companies worldwide speculate on how to profit from this new trend, and many widely respected conferences include entire tracks on how to work with Big Data.

There are many definitions to what Big Data really is. In 2001, an analyst from Gartner, Doug Laney, defined big data to consist of 3 V's. He defines them as **Velocity**, **Volume** and **Variety**.

Velocity refers to the speed at which data is produced. Previously, most dataset could be processed and analysed with primitive mechanisms, without much thought into architecture and algorithms. With the new velocity of data generation, it is necessary to design solutions that can handle and utilize very fast streams of information.

Volume refers to the huge volumes of data that are being gathered in recent times. Modern scientific equipment will generate several terabytes per day, which all have to be processed and stored. To support these growing amounts of data, it is necessary to keep producing larger and cheaper storage solutions.

Variety refers to the endless amounts of different formats that data is stored in. To be able to interpret such unstructured data, it is necessary to have a wide array of tools and mechanisms to analyse these different sorts of information in a way that enables the usefulness of the resulting information.

In 2014, Mark van Rijmenam from Datafloq argued that the definition of Big Data could be extended with 4 additional V's. He defines them as **Veracity**, **Variability**, **Visualisation** and **Value**

Veracity refers to the validity of the data. Even though data is being generated, they are worthless if they are not correct. Smaller amounts of data can be sanity-checked manually, but these huge amounts of data cannot be checked for errors in any sensible way. The data generation mechanisms must be precise and correct.

Variability refers to the changing meaning of a subject. Languages and meanings change over time, and as such, the interpretation of data must adhere to the current context, and not a context that has been established over many years.

Visualization refers to the need for presenting the results from analysing Big Data in a sensible way. Two dimensional graphs can no longer express the multitude of finding that can be found in such huge datasets. New mechanisms will have to be create, which can be used to visualize interesting parts, such that decisions can be made on a clear basis.

Value refers to the potentially lucrativeness of the Big Data industry. While using those huge datasets can indeed help increase profit for many companies, these techniques are not cheap to use. Done wrong, a Big Data venture can cost a lot in storage and processing power.

This project does not aim to solve all problems encountered in the field of Big Data. The main focus of this study is to attempt to mitigate some of the

difficulties that arise from having data streams with *extreme velocity*.

1.1 Terminology

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.

1.2 Triple Buffering Big Data

'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.

1.3 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.

1.3.1 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 I/O 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 I/O.

1.3.2 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'.

[cite Zawinski's law of software envelopment] Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.

1.3.3 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, in case of large files, 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 as it arrives, instead of delaying until after is has been written to disk. This can include gathering statistics, calculating hash values or filtering irrelevant data.

The ESS[1] project aims to create a state of the art super microscope. Such microscope is bound to produce tons of information which many different peers will be interested in analysing. The raw data will have to be stored, in case questions arise about some data. Typically, the data will be stored on inexpensive tape or cheap hard disks, which are very slow storage solutions.

Whenever data is generated from the microscope, there will be a set of different statistical results which will always be necessary. If these statistics could be gathered as the data is in-route to the storage solution, it would not be necessary to load the data back into memory to perform these statistical calculations. This would save a lot of waiting time and free time to perform other tasks.

[1] European Spallation Source, `https://europeanspallationsource.se`.

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 spinning out of control, or not running at all. The widest known example used in teaching about deadlocks 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. When working with concurrency, it is important to have knowledge of how deadlocks can happen, and what measures can be deployed to avoid them, both theoretical and practical.

2.1.2 Finite-state Diagrams

Any process can be interpreted 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 machine, a finite-state diagram can be created. It is a tool that can be used to ensure that the process at hand reacts and interacts as expected. Finite-state diagrams are trivial to both create and understand, and can be used to reason 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 progresses.

In figure Figure 1 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*. The process must be in either state at any one time. Further, there are 4 different events that can happen, *open*, *close*, *lock* and *unlock*. These can happen at any time, and the door will transition to another state. There are implicit transitions on every state, which

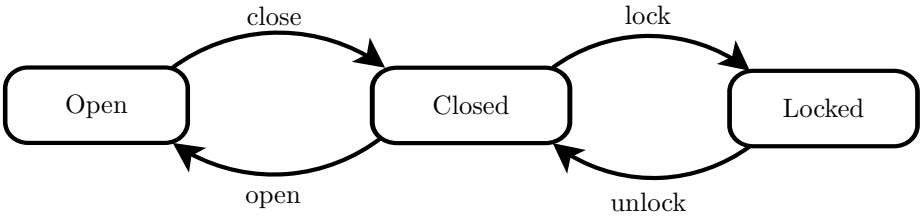


Figure 1: This is an example of a finite-state diagram. It represents a door which can be in 3 states, and there are 4 different transitions.

lead to itself on events that there are not explicit transitions for. For example, locking an open door will make the door transition to the same state, while locking a closed door will make the door transition to the *Locked*-state.

When explaining how the *nbuf* framework works, I will use finite-state diagrams to show how the worker threads behaves. This helps understand how the threads are initialized, used and terminated, and more importantly, how they interact with each other through synchronization primitives.

2.1.3 Synchronization Primitives

Concurrent programming entails using some kind of multiprocessing. This is usually done by using multiple threads which can be scheduled individually on cores. This allows multiple thread to run simultaneous, using the same memory space. While sharing memory is very practical, it also brings several pitfalls that can seem in-obvious to a programmer who aren't used to concurrency. To solve these problems, it is necessary to use some kind of synchronization, to ensure *thread-safety*.

All synchronization mechanisms are based on the hardware instruction *test-and-set*. Using this instruction, locks, semaphores and queues can be implemented. These are all known as *Synchronization Primitives*. These are abstractions, and are intended to make it easier to make more elaborate synchronization in systems. The primitives that are relevant to this project are the following:

- **Mutex** - A mutex is merely an advanced locking primitive. The name is derived from 'mutual exclusion'.
- **Condition Variable** - A condition variable is a primitive that can be used to make threads wait for specific events.
- **Future** - A future is a primitive that encapsulates the result of an asynchronous calculation for the calling thread. The naming is intended to be understood as; *In the future I have this*.
- **Promise** - A promise is a primitive that encapsulates the result of an asynchronous calculation for the executing thread. The naming is intended to be understood as; *I will have to fulfil the promise*.

A **mutex** is basically a lock. It can be used to protect critical section of code. It is a higher-level construct, and automatically makes a thread wait,

when it attempts to obtain the mutex while it is already taken.

A **Condition Variable** is a primitive which can be used to block threads on purpose. A thread can be set to wait on a specific condition variable, which means that it will block until another thread has notified one or all threads that are waiting on that specific condition variable. Most condition variables have a mechanism for notifying just one thread, or all threads. This is practical, since there might be cases where many threads are waiting, but only one can continue, and the other threads will have to wait for some time. In other cases, all threads might be able to continue working after this event.

Normally, using a condition variable requires that some condition has to be true, before continuing, since many implementations allow threads to wake up spuriously, even though no threads has notified. In case of a spurious wakeup, the thread should be able to detect if the wakeup was due to a notification, or if it was spurious.

A **Future** is a primitive which is created by the calling thread, sometimes referred to as the master thread. This thread creates the future primitive and pairs it with a task that can be performed asynchronously. The task can be started, and the master thread can perform other calculations. When the master thread needs the result from the task, it can wait for the future to be finished, such that the result can be retrieved from the future.

A **Promise** is a primitive which is created by the calling thread, and passed off to another thread. From the promise, the calling thread can create a future. The worker thread which received the promise as a parameter can set the content of the promise when it exits, such that it can be retrieved.

When working with multiple threads, it is historically very hard to handle exceptions that happens in other threads than the master thread. When a thread encountered an exception, it would terminate, and throw away the exception. A solution was to define a shared exception variable prior to starting the task, which could then be used to inform about what happened to the thread.

When using the Future-Promise constructs, exceptions are stored in the promise and re-thrown when the master thread attempts to retrieve the result through the future. This makes debugging concurrency a lot easier, since the exceptions are no longer thrown away, and doesn't require the programmer to declare exception variables before every asynchronous call.

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 Data Marshalling

When receiving and sending data to and from a stream, care should be taken when interpreting the data. The act of turning an object into a string representation is known as *marshalling* or *serialization*. The opposite act, turning a string representation into an object is called *demarshalling* or *de-serialization*.

There are many ways to serialize an object, depending on what data the object contains. If the object contains variable length strings, there are several ways to design the string representation. One solution is to decide on the maximum length of the string, and then allocate that same size in the string representation. This can result in a lot of unnecessary white space, if the string turns out to always be very small, compared to the allocated size. De-marshalling becomes trivial, because you know exactly where and how long the string is. Also, the size of the string representation can be predicted precisely.

Another solution is to use some kind of delimiters. Depending on the content of the string, we can use a symbol such as the pipe or a simple semi-colon. This solution saves a lot of space, but makes parsing harder, since it is necessary to inspect the data, before parsing it. Further, you cannot reason about the maximum size of the string representation, since the strings can have arbitrary length.

When using a very simple object structure, it may be easier to simply use the binary representation of the object. In some cases this will work, and others it will not. Different platforms and compilers may differ in how the memory layout of an object is constructed. This will lead to inconsistencies and ultimately in-correctness.

Google has given a solution to the marshalling problem with a mechanism they call Protocol Buffers [2]. They allow you to specify a data structure in a platform- and language-independent way, and have constructed cross-platform libraries for C++, Java and Python. These libraries can be used to create a protocol buffer object, from the specified structure. When desired, one such object can be marshalled, and de-marshalled by another program, on another platform or in another language. This helps keeping a consistent view of how data is interpreted and altered across multiple systems and implementations.

The Protocol Buffer implementations inspect the string representation when de-marshalling the data, and it is impossible to predict the length of an objects string representation, unless you yourself enforce fixed width-strings. While

[2] reference google project page

inspecting is slower than parsing a fixed-width string representation, Google has promised that the implementations are highly optimized. Further, they recommend keeping protocol buffer objects smaller than 1 megabyte.

When a developer uses the *nbuf* framework, it is recommended to use a protocol buffer, when processing the data. However, since the *nbuf* framework only allows fixed-width parsing, care must be taken to ensure that the width of the string representation is constant, or ensure that proper padding is added.

2.2.2 Optimal Buffer Size

Buffers are used everywhere. The optimal size is a ubiquitous definition, and all depends on what you want to *optimize for*. In some cases, you want speed, in other cases, you want a low memory footprint.

Typically, you want the system to perform well, within some given constraints. Sometimes using too large buffers merely slows down a system, since other processes won't be able to allocate the desired memory.

When processing data, a buffer that is larger than the amount of data it needs to buffer is unnecessary. As an example, a video-buffer is large enough to ensure that connection fallouts of a certain size can be mitigated. If the buffer is so large that it can contain the entire video file, most of the buffer is wasted, since memory could easily be reused, when playing the video.

When using the *nbuf* framework, using buffers that allows a single thread to contain the entire file in its local buffer, will make the framework behave inherently sequential. This reduces the multiprocessing capabilities of the framework. However, using too small buffers will induce extra overhead, since the amount of I/O operations will be increasing. Finding the right balance is essential, but it is hard to generalize what size will fit most projects.

2.2.3 System Data Steams

The *nbuf* framework seeks to give an agnostic approach to how the streams behave. The framework should give the same results, when given the same data, whether the data comes from a network stream, a file stream, or a stream from a memory location. This means that it has to use only slightly abstract methods, on the streams that are supplied.

It is important to know how different streams behave, even though the framework must have agnostic behaviour. When benchmarking, it is indeed important, to be able to identify where and why certain results are attained.

In C++11, there are two different kinds of basic streams, which both inherit from a general stream type. These streams are known as *string streams* and *file streams*. A third kind of stream can be gained from third-party libraries, such as the Boost.Asio library[3], namely a *network stream*.

[3] http://www.boost.org/doc/libs/1_59_0/doc/html/boost_asio.html

These three kinds of streams can be used in the *nbuf* framework, which gives a very versatile system. Each kind of stream, however, has limitations which in general will limit the throughput of the framework.

A *network stream* used as input is limited by the bandwidth of the connection to the data source. If the data source is on a local wired network, it is common to have a large bandwidth, while a remote data source, transmitting via a modem will have a drastically reduced bandwidth. The same limit applies when sending data to a network stream. However, most operating systems keep a buffer, such that the program can quickly return, while data may not be sent yet. The data may stay in the buffer for some time, until the operating system decides to transmit the data. If large amounts of data is transmitted at once, the buffer might be too small, and the operating system is forced to transmit some data, before returning control to the calling process.

A *file stream* used as an input stream is limited by the bandwidth of the disk and the bus that connects the disk to the system. When traditional hard disks or tapes are used, locating data can be a slow task, and benchmarking such tasks will result in very inconsistent results. The disk has to seek to the right disk and location, before it is able to read the file. Newer technology, such as a solid state disk or raid configurations can mitigate or eliminate these inconsistencies. When a file stream is used as output, the operating system again buffers the data, which allows the calling program to return rather quickly. Similarly, with output data that is too large to fit in the buffer, the operating system will have to save some data to disk, before returning to the calling process.

A *string stream* is a stream that streams directly from data that is already located in main memory. The bandwidth of such stream is only limited by the transfer rate of the main memory technology and the bus that connects the memory to the CPU. This is, without doubt, the fastest stream that can be used in the framework. When used as an output stream, it has the same fast transfer rate. The downside to this kind of stream is that it is very impractical, since you will hardly ever be able to fit all your data into a string stream, and if you do, you might as well have processed directly from the other stream, instead of putting data into the string stream.

When benchmarking the *nbuf* system, we want to ensure that optimal I/O has been achieved. If the framework can handle concurrency from fast streams, slower streams should be trivial to handle concurrently.

2.3 Theoretical Speed-up with Threads

Most modern CPUs include multiple cores with hyper threading. On a single-core systems, only one thread can execute at any one time. On multi core systems, multiple threads can execute in parallel. If every process only contains a single thread, no memory is shared between multiple threads, unless explicit process memory sharing is used.

If a process contains several threads, the process can have several threads

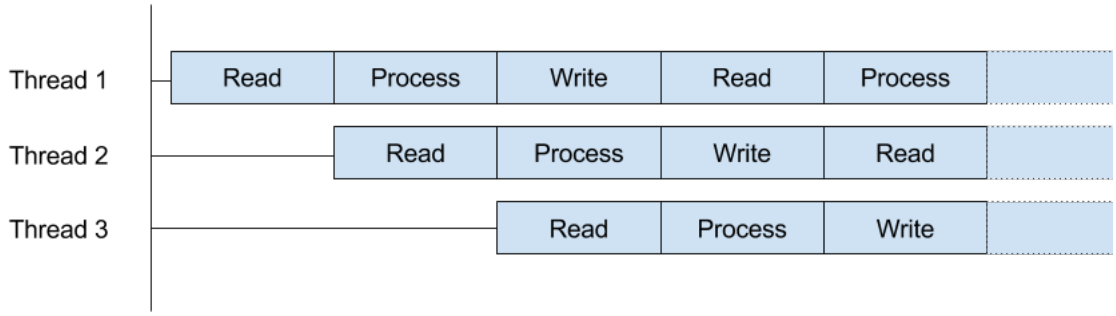


Figure 2: This is an example of how the *nbuf* framework could interlace, if all three tasks are similar in time taken.

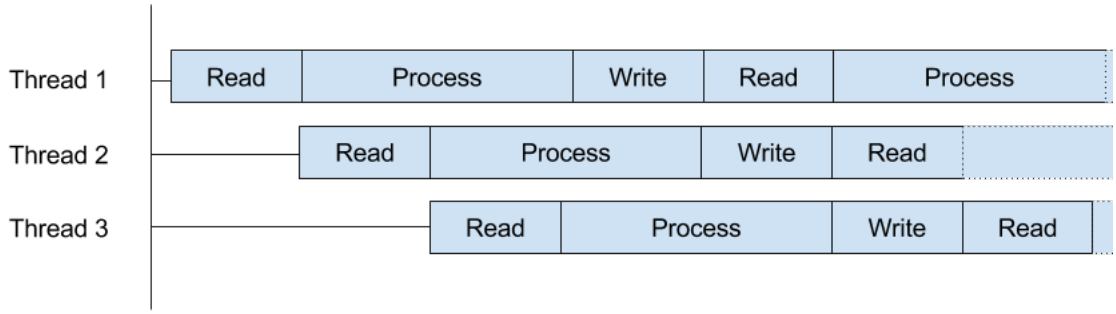


Figure 3: This is an example of how the *nbuf* framework could interlace, if the processing task takes more time than the read and write tasks.

executing simultaneous on multiple cores. When tasks can be done in parallel, the entire bunch of tasks can be finished faster than if they were performed in sequence.

The *nbuf* framework allows developers to interlace the tasks of *reading*, *processing* and *writing* data. Following are a few examples of how the three tasks at hand can be interlaced.

In Figure 2 is an example of how the framework could interlace if both the read-, the process- and the write-task took an identical amount of time. In this case, three threads would be able to completely occupy both the read and the write resource at all times. If another thread was added, it would only introduce idle times, while waiting for the resources to become available.

In Figure 3 is an example of how the framework could interlace if the processing task takes longer time than the read and the write task. The processing task can be done in parallel, but after the third thread has read, the read-resource is available for some time, until the first thread is done processing and writing. In this case, more threads are needed to fully utilize the read- and write-resources.

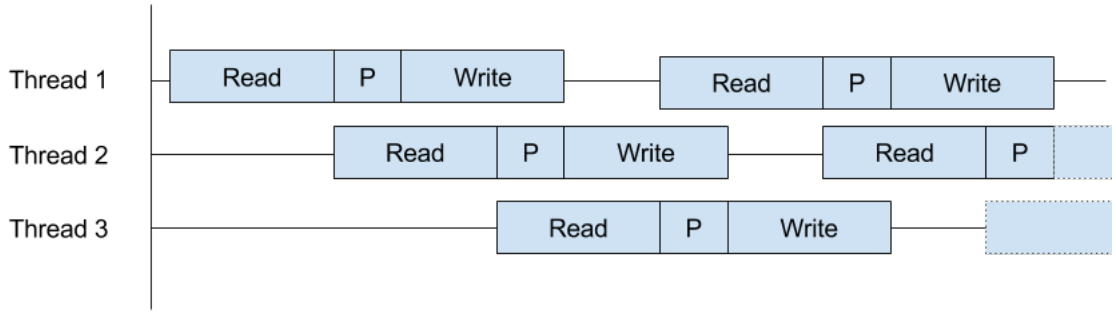


Figure 4: This is an example of how the *nbuf* framework could interlace, if the processing task takes less time than the read and write tasks.

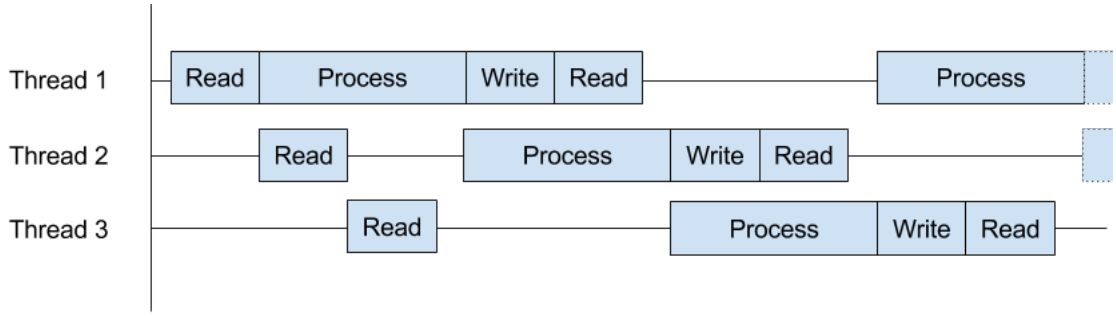


Figure 5: This is an example of how the *nbuf* framework could interlace, if the processing task takes more time than the read and write tasks, but the processing task can only be performed by one thread at a time

In Figure 4 is an example of how the framewould could interlace if the processing task takes less time than the read and the write task. The read- and write-resources are occupied at all time, but threads are merely waiting idle, for resources to become available. In this case, two threads would not be able to fully utilize the read- and write-resources, but the smaller the processing task becomes, the more idle time there will be. In the case where there is no processing at all, only two threads would be able to fully utilize the resources.

In Figure 5 is an example of how the framework could interlace if the processing task takes more time than the read and the write task, and the processing step can only be done sequentially. The read- and write-resources are not fully utilized, but processing happens as often as possible. Adding or removing threads would not increase the utilization of the read- and write-resources, so in this case, the utilization cannot become any higher.

From this, we can conclude that full utilization always can be achieved when we have an unlimited amounts of threads. With four threads, we can, theoretically, obtain close to a quarter of the execution time, including some latency hiding, compared to only having one thread. In cases where the processing task is very small, more than three threads does not increase the throughput.

When the processing step has to be done in sequence, it might not be possible to achieve full utilization of the I/O resources.

2.4 Theoretical Speed-up with Devices

Using a single CPU on the machine at hand severely limits the obtainable speed-ups. Modern CPUs can include up to 16 cores with hyper threading. Fully parallelizable algorithms can hope to reduce the calculation time to around a tenth, compared to a sequential version. When using multiple CPUs, it should be possible to reduce the execution time even further. The impressive speed-up will not be found when using CPUs, however.

Modern GPUs easily has more than a thousand computational cores. While this sounds alluring, they are far more difficult to use, than traditional CPU cores. They are bound by being in the SIMD-class of devices, which is short for *Single Instruction, Multiple Data*. All cores on a SIMD-device can only execute the same instructions in parallel, but they can do so on different sets of data. In some cases, this is useless, but in other cases, this can greatly increase the performance of an algorithm.

Coding for a massively parallel device requires extensive knowledge of how to arrange data, and code in the domain specific language, in order to obtain the desired results. While the details of GPU programming is not part of this project, the idea that a thread can schedule the task on a GPU device is indeed interesting.

A single thread can schedule multiple operations on several GPU devices. This requires handling of the stream of data, split it into suitable sizes and scheduling the tasks. Then the thread must retrieve the results, combine them back into an aggregated result and ensure that the results are properly written to the right stream.

The *nbuf* framework can easily be used, to allow threads to use external devices during the processing step. If this is done right, the execution times can go as low as a thousandth of the sequential version. This is outside the scope of this project, but it indeed important to keep this option open.

3 Design and Implementation

This section will explain how the *nbuf* framework has been designed and implemented. First I will identify and elaborate on the inherent requirements of such framework. Secondly, 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 a technical description of how it has been built.

3.1 Technical Requirements

The *nbuf* framework, being generic, should enable developers to create a wide array of software, which solves different problems. This can easily result in *software bloat*, when software gets too much functionality. *nbuf* should include features which is necessary to solve basic tasks, but a balance is important, when it comes to deciding what is necessary and what is not.

Follow are a list of functional and non-functional requirements which I deem important enough, to warrant further discussion:

- **Efficient and Parallel I/O** - The *nbuf* framework must prioritize keeping the I/O resources busy as often as possible.
- **Limited Space Guarantee** - When processing data, the framework must be able to keep a strict upper limit on allocated memory.
- **Custom Processing** - The developer must have freedom in deciding how to process or transform the data.
- **Input/Output Sensitivity** - The order of input and output must be identical, if the developer desires this.
- **Output Selecting** - It must be possible to only output statistics, or only output transformed data.
- **Output Filtering** - It must be possible to only output parts of the transformed data.
- **Guaranteed Termination** - The framework must terminate if and only if it works with a finite amount of data.

These features are all items that are important, if this framework should be generally applicable to most sorts of projects working with any kind of I/O. In the next few sections, I will explain what these requirements entail, and how they have been solved.

3.1.1 Efficient and Parallel I/O

The entire project attempts to maximize the use of the available I/O resources on a system, when processing incoming data. If there are solutions which allows for a larger rate of processing data, then the *nbuf* framework does not perform as it should. This is a primary goal, and should at all times be respected. This non-functional requirement will only be observable when running the system.

To achieve this goal, it is necessary to use some kind of concurrent programming methods. It is impossible to keep all system I/O resources busy, using

only a single processing thread, since there will often be at least two resources, and there will be real processing work to be done as well.

As mentioned earlier, this has been done multiple times with a triple buffer system, where three processing threads would perform all three tasks in parallel. One thread would keep the input resource busy, would thread would keep the output resource busy, and one thread would perform the processing work.

In a new framework, we can evaluate and rethink this triple-buffer-idea. In cases where there are is a high amount of processing work, a triple buffer system will be waiting for the thread performing the real work. On a single core system, this is not a solvable problem, since we can't speed up the execution speed. A good computer scientist might be able to optimize the program, but this has a limited usefulness.

On the other hand, if we have a system with multiple execution cores, it would be possible, and a very good idea too, to utilize the extra computational powers to speed up the execution time. This requires that the work can be done in parallel, which is not always the case. In cases where the work can be parallelized, the extra processing can be utilized, until the rate at which data can be processed matches the lowest rate at which data can be read or written. This will, in turn, maximise the use of the available I/O resources on the system.

The framework must be able to use multiple processing cores, and support the developers in performing parallel work.

3.1.2 Limited Space Guarantee

The main memory available on a computer system is rarely a hard limit. Usually, when a process starts allocating more main memory that are available, some data is moved to a swap-partition on the systems disk drive. This results in very high delays when allocating more memory, and trying to read data which has been moved to the swap-partition. When trying to work efficiently, this is a killer, and any processes doing this, will completely stall the entire system.

For this reason, it is important to be aware of the memory usage of all processes on a system. Software which greedily allocates huge amounts of memory, or simply forgets to deallocate memory (resulting in memory leaks) is to be considered buggy, and will not be used in critical systems. It is a minimum requirement for all software to deallocate memory, and many programming languages employs different methods such as garbage-collectors or scoped variables to ensure that memory is deallocated when it is no longer used.

Back in the 1994, Bjarne Stroustrup [The Design and Evolution of C++] introduced the term and programming idiom *RAII*, short for *Resource Acquisition Is Initialization*. This has become a widely used technique which gives several advantages. In RAII all memory required for an object to exist, or a process to run, will be allocated during initialization. This gives the advantage of ensuring that the process will not slowly allocate additional memory, and over time exhaust the main memory resources.

The *nbuf* framework will require memory and as such, it should be possible to give an upper limit on how much memory it will consume. If the process consumes more memory than it has been given, then there is a risk that the

system will start using the swap-partition, which is bad. The given upper limit should be respected at all times.

3.1.3 Custom Processing

The developer using the *nbuf* framework will only use it, if it gives the freedom to solve whatever task he desires. Some cases warrant only gathering statistics about the data at hand, others require a slight transformation or reorganisation of data. These cases should be solvable using the framework.

The worker performing the real processing must be programmable by the developer. When the worker has a buffer full of data, it must be up to the developer to decide how to process the buffer. There should be allocated some memory for gathering statistics, and it should be possible to alter the data in the buffer. What the developer intends to do with the data in the buffer is to no concern of the framework, but the framework should support the intentions of the developer.

Here are the three identified types of data that the developer might want to extract and save when processing data:

- **Input-wide Accumulated Data** - This data is to be accumulated across the entire input data.
- **Buffer-wide Accumulated Data** - This data is to be accumulated across one buffer, or any smaller amount of data.
- **Transformed Data** - This data is an in-memory transformation of the data in the buffer.

These three types of data must be creatable and extractable, when using the framework.

3.1.4 Input/Output Sensitivity

The case where we want to find the minimum value, the maximum value and the average value in a file is a very simple example. It is of linear complexity and simply requires a single run through, in no specific order. This is a highly parallelizable algorithm and is a brilliant example case for this kind of framework.

Another case, creating an md5-sum for example, requires that the data will always be parsed in the same order, every time. For simplicity and practicality, it has been decided that the sum must be calculated chronologically, meaning that the first data in the file must be hashed first. This is an algorithm which cannot easily be parallelized to any extend, however, it does heavily rely on I/O, and would benefit from using a framework such as *nbuf* .

To accustom to this need, it must be possible to decide that the framework should always parse the data in-order and not parallelize the calculations. This can, of course result in not gaining a maximum utilization of the I/O resources, but this is always the choice of the developer and a requirement at hand, related to the task.

3.1.5 Output Selecting

When calculating the md5-sum of a file from disk, there is no need to write the content of the file back to disk. In this case, it must be possible to not occupy the output resource with needless work. If the framework occupies resources it does not need, it can lead to other processes waiting for the resources which will result in a slower system. Therefore, it should be possible to decide what parts of the data that should be retrievable from the framework.

3.1.6 Output Filtering

In cases where parts of the data might not be interesting, or the transformed data is smaller than the original data, it would be ideal to filter data such that it does not occupy more memory than necessary. This is indeed relevant in the use case where the developer receives data from sensors, which in periods doesn't measure interesting data. This could be a rain gauge, which reports 0 millimetres for many months a year.

3.1.7 Guaranteed Termination

The framework must terminate when it works with a finite amount of data. The framework can never be fool-proof, but it should always terminate when it receives an indication that there are no more data, often an EOF. If the developer employs an infinite loop when processing data, this promise cannot be kept, but the framework should never result in a deadlock or a spinlock.

These are the technical requirements that I have decided the *nbuf* framework will have to live up to, to be generally useful. During the remainder of this section, I will discuss what actions I have taken to ensure that these requirements are fulfilled.

3.2 Abstract Overview

When performing concurrent programming, it is custom to have a master thread which prepares all the communication channels, the worker threads and allocates all the resources required to perform the concurrent work. This is part of the RAII idiom, and will be a central part of the design of the system. In this section I will give an abstract overview of how the master thread will prepare the environment to enable the worker threads to run, and how the workers synchronize with each other, to ensure correctness and to avoid race conditions.

3.2.1 The *nbuf* master

The initialization of the *nbuf* framework entails a few tasks:

- Sanity checking settings
- Allocating system resources
- Initialize worker synchronization mechanisms

After parsing the data, the master thread will be responsible for:

- Deallocating system resources
- Termination of worker threads
- Returning the required data

While allocating system resources should be a job for the master thread, due to RAII, I will deviate slightly when it comes to allocating the worker buffers. Sharing memory between threads require careful coordination. This is part of why concurrent programming is inherently hard. If I can establish a method where less memory sharing is used, it will make the framework less complex, easier to understand and more maintainable. For this reason, some allocation will be left to the worker threads. This will include the worker buffers and the memory for the buffer-wide accumulated data, which are the largest part of the required memory.

3.2.2 The *nbuf* workers

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 6 is a finite-state diagram which shows how each worker in the system transfers from state to state, and 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.

To clarify the intention behind how the workers interact, I will here explain how a worker move through the states in the diagram. Remember that there are a finite amount of workers, and that the input resource will be empty, at

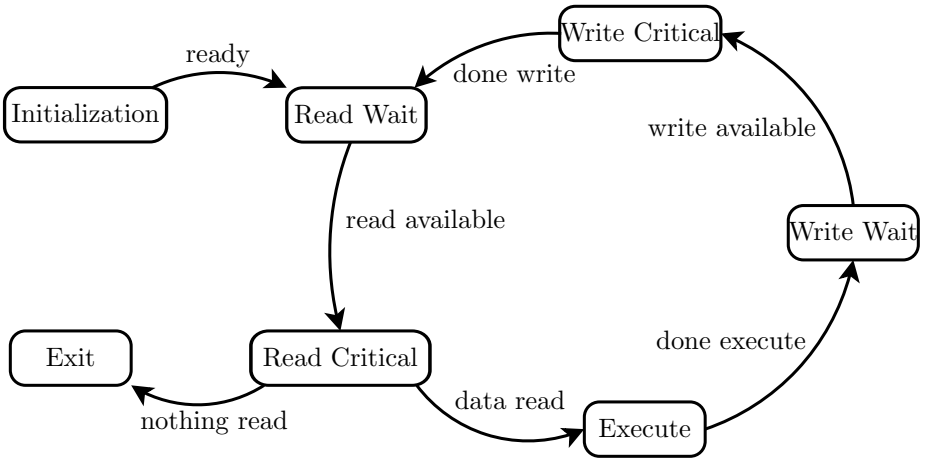


Figure 6: *nbuf* worker state diagram. This finite-state diagram shows how the workers can change states, based on events. Further explanation on how the transitions happens can be found in the related transition table which can be seen in Table 1.

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 Critical	Worker can now write, which blocks other workers from this state.
Write Critical	*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. This table shows the states, acceptable input and transitions within the *nbuf* worker state diagram, which can be seen in Figure 6.

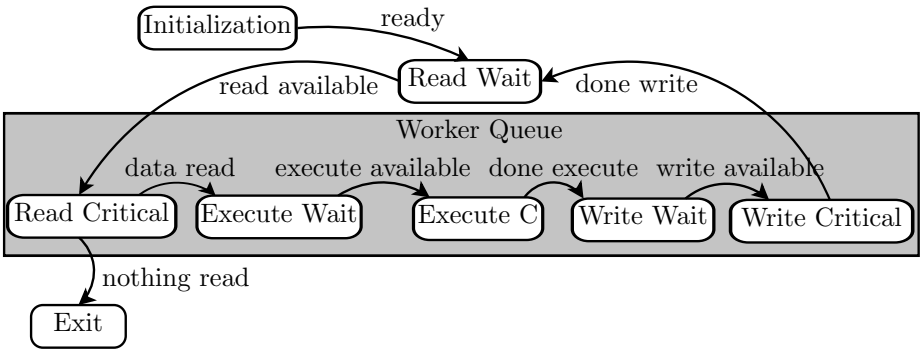


Figure 7: *nbuf* worker state diagram with a queue. This is an evolution from Figure 6. The workers cannot overtake each other when they are inside the queue.

some point. In cases where the input will never empty, we will not expect the program to terminate.

First, all workers begins in the *Initialization*-state and, it will move into the "Read Wait"-state. When the single read resource is available, a worker will move into the *Read Critical*-state. This state is exclusive, since only one worker can read at a time. This may be any of the workers which are in the *Initialization*-state.

We now follow the worker in the *Read Critical*-state. At this point, two things can happen. Either, the worker receives data from the resource, or it does not receive data. If it does not receive data, it will be because there is nothing to receive from the input-resource. If it receives data, the amount of data it receives does not matter, the buffer may be almost empty, or it may be full.

With data in the buffer, the worker will move to the *Execute*-state, and another worker can enter the *Read Critical*-state. Note that the *Execute*-state is not exclusive, and that multiple workers can perform this step in parallel.

Now, the worker will process the data located in the buffer, and produce whatever data the developer has decided. When the worker has finished processing, it will move into the *Write Wait*-state. In this state, the worker will wait for the single output resource to become available. When it becomes available, the worker will move to the *Write Critical*-state and occupy the output resource. When the worker has written the content of its buffer to the output-resource, it moves to the *Read Wait*-state, since it has finished the 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, which will be initiated by the master thread.

It is clear that this state diagram will serve the general purpose, but **Input/Output Sensitivity**-requirement can not be supported this way. To comply with this requirement, the framework must support utilizing an altered state diagram which can be seen in Figure 7. In this diagram a FIFO-type queue has been added. This means that the worker thread that entered the

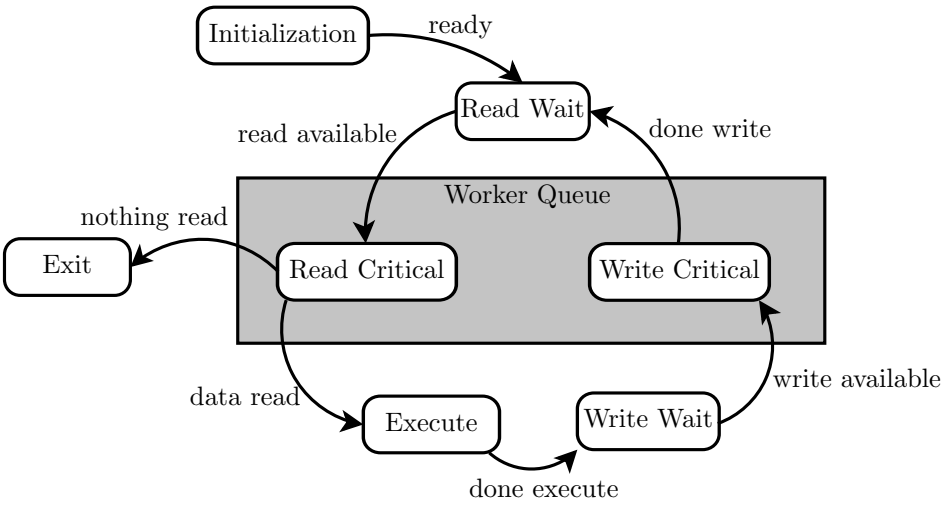


Figure 8: *nbuf* worker state diagram with a smaller queue. This is an evolution from Figure 7. The workers cannot overtake each other when they are inside the queue, but the execution step can be performed and finished in parallel. This means that workers entering the queue at the *Read Critical*-state has to go to the *Write Critical*-state in the same order.

queue first will reach every new state before every other thread. Further, a *Execute Wait*-state has been introduced along with introducing an *Execute Critical*-state, which replaces the original state.

Figure 6 and Figure 7 are two extremes. One entails full parallelization, the other is inherent sequential. There are two middle-way solutions.

If we want the output to be sequential, but the execute step to be performed in parallel we will have to introduce a different kind of queue. This queue will merely ensure that the order of threads entering the *Write Critical*-state matches the order they arrived at the *Read Critical*-state. This will still allow parallel processing, but threads will not be able to overtake each other in the cycle, outside of the *Read Wait*-state. The third alternative can be seen in Figure 8.

The other middle-way solution can be seen in Figure 9. This entails having a sequential execute-step, but the order of the output does not matter. In this setup, the threads will have to arrive in the *Execute Critical*-state in the same order they were in the *Read Critical*-state.

Considering these four configurations, we can reason about how they relate. In Table 2, it is clear that they must all exist, due to combinations of different requirements.

But, there are even more possible configurations. There are cases where we want to ignore the *Write Wait*-state and the *Write Critical*-state, since the **Output Selecting**-requirement requires that we can turn off outputting the content of the buffers. This voids some of the configurations, but for simplicity, we will still consider these four configurations, and merely perceive that the

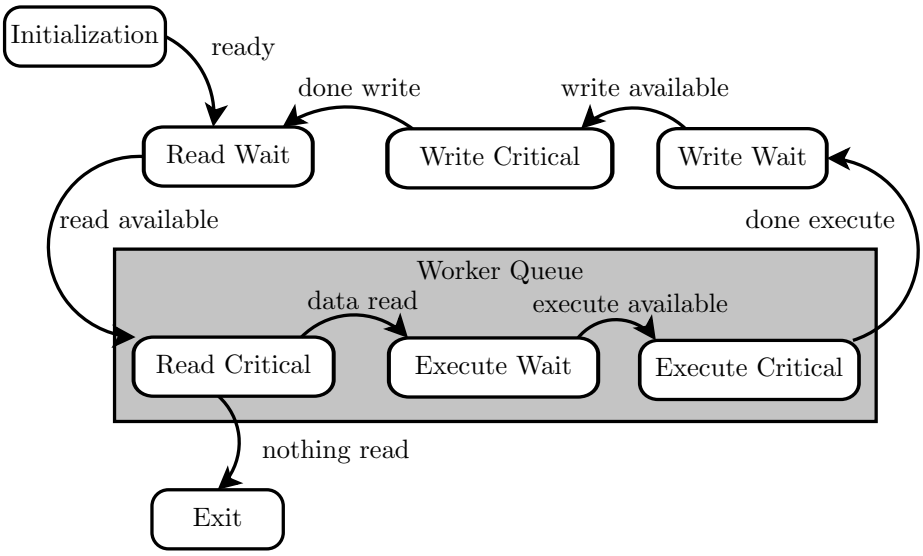


Figure 9: *nbuf* worker state diagram with a smaller queue. This is an evolution from Figure 7. The workers cannot overtake each other when they are inside the queue, but the write step can be performed and finished in parallel.

Finite-State Diagram	**Sequential Execution**	**Input/Output Sensitivity**
Figure 6	No	No
Figure 7	Yes	Yes
Figure 8	No	Yes
Figure 9	Yes	No

Table 2: Configurations of the requirements, related to the finite-state diagrams. The figures have all been explained in-depth previously.

work to be done in these two steps is non-existent.

This concludes the requirements to the the *nbuf* framework. A framework which supports all of these items, should be generic enough to be useful, while still being practical.

3.3 Algorithmic Overview

After establishing an abstract overview of how the master thread and the worker threads interact, I will now give the algorithms in pseudocode, to further explain how the *nbuf* framework interacts.

The algorithm performed by the master thread can be seen in algorithm 1. It is a very simple algorithm, and merely creates the worker threads, and gathers the results. In the case of a sequential execution, the result is stored in shared memory, and no combination is required.

```
Master thread method;
begin
    allocate futures matching # of threads;
    for 1 .. # of threads do
        spawn and launch worker thread with future;
    end
    for 1 .. # of threads do
        wait and retrieve future result from worker thread;
    end
    combine list of results;
    return combined result;
end
```

Algorithm 1: The master thread spawns the desired amount of worker threads, waits for them all to finish and then returns a combined result.

The algorithm performed by the worker thread can be seen in algorithm 2. The worker calculates the size of the buffer. This is done by dividing the available memory out onto the number of threads, and then aligning with the highest amount of strides that can fit in in the buffer.

When the buffer has been allocated, the worker locks the input resource and reads data into its locally allocated buffer. After unlocking the input resource, the worker processes the data, one stride at a time, as the developer has decided. Then the worker locks the output resource, outputs the transformed data, and unlocks the output resource. The while loop repeats until the input resource is

emptied, where after the worker thread returns the resulting promise.

```
Worker thread method;
begin
    allocate promise struct;
    calculate and allocate buffer memory;
    more data = true;
    while more data do
        lock input resource;
        read from input resource;
        unlock input resource;
        if data was read then
            get count of how much data;
        else
            more data = false;
            break;
        end
        process data, store result in promise;
        lock output resource;
        write to output resource; unlock output resource;
    end
    return promise;
end
```

Algorithm 2: The worker thread allocates a buffer, where after it reads, processes and writes. When no more data is available, it returns the resulting promise.

3.4 Framework Interface

When building a framework for other people to use, it is important to keep a clean, intuitive and usable interface. While man-pages are very useful, they can be dreading to read, for the non-guru. There are several ways to design a clean interface. Some developers tend to require a ton of configuration to do the simplest of things [4], while others merely have sane defaults, and requires configuration to use the more advanced features[5].

First, I need to identify what input is required to configure the *nbuf* framework. Here is a list of input, along with example defaults:

- **Input Stream** - The framework can default to read from *stdin* [6].
- **Processing Method** - This is the method used to process a single stride. The framework can default to doing nothing with the input data.
- **Sequential Execution** - This is a bool, and the framework can default to performing parallel execution.
- **Output Stream Enabler** - This is a bool, and can default to true.
- **Output Stream** - In cases where output from the worker buffer is desired, this must can be supplied. It can also default to *stdout*.
- **Output Filter Enabler** - This is a bool, and can default to false.
- **Output Filter Method** - In cases where the output should be filtered, a filtering method must be supplied. It can default to terminate the program, since not filtering would signify a user error.
- **Output Filter Stream** - In cases where filtered output from the worker buffer is desired, this must can be supplied. It can default to *stdout*.
- **Number of Threads** - The framework can default to an arbitrary number, e.g. 3 threads.
- **Memory Limit** - The framework can default to an arbitrary number, e.g. 100 megabyte, which most systems should have readily available.
- **Data Stride Size** - The amount of data that should be processed at once, by the processing method. This can default to 1KB.
- **Input/Output Sensitivity** - This is a bool, and can default to false.

This is the configuration that the framework must accept. They are, however, different types, and as such, they should be handled differently. Two of the input parameters are code references. There are several ways to insert code to be executed into programs. Many languages support method-overwriting, when working with objects. This could entail subclassing an object, when a developer requires to use the system.

[4] The linux-command *find* is a good example of non-intuitive CLI. You would expect the first argument to be the filename you want to find, but you need to specify the filename with -name.

[5] The linux-command *locate* does what you would expect, it locates files with the name you supply as the first argument.

[6] Standard input is a standard stream used on most *NIX systems, along with standard error and standard out. They are known as *stdin*, *stderr* and *stdout*.

3.5 Multithreading with *std::thread*

The *nbuf* framework has been built using C++11. A new feature in C++11 is the support for native multithreading, and not depend on external libraries to execute parallel threads. Before C++11, it was necessary to call the POSIX thread library[7] directly, which did not offer many abstraction layers.

The new thread library included in C++11 is referred to as std::thread. This library uses POSIX threads, but implements a much nicer abstraction layer, which allows the developer to ignore certain aspects. This allows for higher productivity and less complex code. This, in turn, makes the code easier to maintain.

When the *nbuf* framework is started, the master thread will create *futures* matching the number of worker threads. The master thread will expect each future to contain a pointer to the accumulated data from a single worker thread. Each worker thread will be initialized with a promise, which they are expected to fulfil. When they have reached the *Exit*-state, they return the accumulated data to the promise, and the master thread will gather all results. The results will be combined, and this will be the result returned from the framework. A combined, accumulated set of data.

The framework has been designed to require as little synchronization as possible. While this has decreased the complexity of the framework, it has not completely eliminated the need for synchronization. The default case, without *Input/Output Sensitivity* and without *Sequential Execution* requires only two mutexes, which will never be locked at the same time. These two mutexes will manage each of the critical sections, related to the I/O resources.

When a critical section is required near the execution-step, it will also require a mutex. In this case, more synchronization is necessary, since the workers will also have to keep stay in the same order, at some point (all but the first configuration in Table 2. A worker-queue must be used, to ensure that threads are kept in order. There is no native synchronization mechanism for this kind of problem, but one can be built using *condition variables*.

These are the technical details, related to how the *nbuf* framework handles concurrency with the *std::thread*-library.

3.6 Current Limitations

The current[8] implementation of the *nbuf* framework does not include all the features that are listed in the *Technical Requirements*-section. Here is a list of features that are not yet implemented, but can be supported by the framework, in the future.

- **No Queue** - The framework supports sequential execution of the user-supplied processing method, but there are still race conditions that can result in processing the data out of order.

[7] The POSIX thread library is also known as pthreads.
[8] As of the 19th of October '15.

- **Missing output selector** - Currently, only one output stream can be supplied. If both filtering and the full output are desired, then the user is out of luck. The current implementation allows the usage of a filtering method, but this will override the non-filtered output.

These limitations are trivial to implement, but did not make it to the final result from the thesis project.

4 Experimentation and Benchmarking

The *nbuf* framework has two primary goals. It has to be efficient, and it has to be usable. The first goal can be measured quantitatively, the second cannot. This section will focus on deciding how efficiently this framework can work with system I/O.

4.1 Experimental Setup

The framework solves a practical problem, namely it tries to fully utilize the speed of disks, bandwidth of networks, etc., and therefore it must be benchmarked in a practical environment to prove its effectiveness. This however, will add a lot of uncontrollable factors, to the results of the benchmarking. This section will elaborate on how different factors can affect the benchmarking, and how to eliminate these factors.

What I would like to measure is *throughput*. The throughput can be measured as bytes processed per second. This clearly depends on how much execution work is required, and how much computational resources are available. If enough computing power is available though, the throughput depends less on the computation task.

I have decided that the benchmarking calculation is as simple as a search for minimum and maximum and calculating an average value. The naive implementation is simply a sequential loop which sums the values, while looking for minimum and maximum values and counting the amount of values. This implementation can be used as a baseline to test how much faster the *nbuf* framework can perform these calculations. If the framework is slower than this naive implementation, then the framework is useless.

The benchmarking will be done on a machine with a Intel Core i7-2600K CPU 3.40GHz x 8. This is a processor with 4 cores, all with hyperthreading. On a system, it will appear to have 8 logical cores, but it can only run 4 hardware threads at a time. Futher, the system has 4 * 4GB of DDR3 1333MHz as main memory. This will sum up to 16 GB of main memory. Finally, the machines has a Samsung 500GB hard disk drive, with 16MB cache and 7200 RPM. The machine runs Ubuntu 14.04 64-bit as operating system.

4.1.1 From disk

In many cases, the framework will have to work with a stream of data from a disk drive of some sort. In this case, the maximum throughput is limited by the rate at which we can read data from the disk, and write back to disk. Here we have to keep two things in mind, when experimenting.

First, most operating systems keep a cached version of files in memory, as long as the memory isn't used for anything else. This has been a mystery to many new Linux-users, and has sprouted many helping words[9]. This easily

[9] http://linuxatemyram.com explains very nicely how Linux caches files when there is available main memory.

Action	Average time taken	Standard Deviation / Average
Reading 1GB of data from disk	13411522 μs	0.0281
Writing 1GB of data to disk	541844 μs	0.0128
Copying 1GB in memory	184583 μs	0.0046
Processing 1GB in memory	4757252 μs	0.0068

Table 3: The results from the disk I/O experiment on the actual hardware. Note that when writing data to disk, the operating system performs *latency hiding*, which is why we observe much faster *writing* that *reading*. These are the timings we should expect to see when reading or writing 1GB of data to disk.

results in wrong experimental results, when benchmarking throughput from disk.

Imagine performing an experiment twice, which includes timing how fast a file is read from disk. The first run will fetch the file from disk, and the experiment will return expected results. When starting the second run, there is a good chance the file is still in memory, and the operating system will therefore not fetch the file from disk, but merely return the cached version. This will result in a much faster benchmark for the second run, for the wrong reasons.

Second, the operating systems doesn't write your file to disk, just because you tell it to. It has to schedule all disk I/O and therefore it keep an internal buffer of data to be written to disk. This means that when some code writes to a file-stream, it will return before the data is actually written to the disk. The data is copied to another part of the system memory, and kept there until the OS decides to write it. The size of this buffer can vary, depending on how much memory the OS keeps to itself. When the buffer is full, the OS is forced to wait to disk, and let the program wait, until the OS has enough memory to copy the data from the program. To show how large the wait on disk input and disk output is, a simple experiment has been conducted.

The I/O experiment is meant to show the different between return times when performing disk input and disk output, when the operating systems manages the I/O. First, a file with 1GB of data is located on a file, on a consumer-grade hard disk drive. The OS file cache is emptied, a timer is started, and the program starts reading the entire file into main memory. When returning from the read, the timer is stopped and the resulting time duration is printed. Then a timer starts, and the program starts writing the 1GB data to a new file on the disk drive. When the OS returns to the user-code, the timer is stopped and the duration is printed. The experiment is repeated 20 times, after letting the operating system flush the buffer to disk, to ensure consistent results. The results can be seen in Table 3.

When this has been concluded, we know that the throughput of the *nbuf* framework, is limited by the rate at which it can read data from the disk. The throughput of the framework should be as close to this as possible, to exhibit efficient I/O. If the framework can parse data faster than the disk loading times, it is a clear warning that something doesn't work as expected.

4.1.2 From memory

In comparison, main memory on a system suffers from much lower latency, than disk. This means that the potential throughput is much larger, when processing data that are already in memory. The experiment conducted here, merely copies 1GB of ram from one allocation to another. It is not necessary to test reads and writes, since they are essentially the same, in memory.

The results in Table 3 clearly shows that working with data that is already in-memory is much faster than when it has to be fetched from disk, to no surprise. Also, the timings are much more consistent, since we aren't working with a spinning metal disk that has to seek and find.

We can use these results to gain a higher confidence, when benchmarking the *nbuf* framework. When testing the framework with a disk file-stream, the results are subject to these long load times. If we pre-load data to memory, it means that we can actually benchmark the framework to its limit, be it due to software or hardware, instead of merely testing the loading times from disk.

4.2 Experimental Results

Every experiment has been run 20 times, averaging the results and calculating the standard deviation. In the experiments where data is read from disk, the file cache has been cleared just prior to starting the experiment. Further, all non-critical software running on the experiment machine was terminated, to ensure that there was as few other processes running on the system. While this was possible to a large extend, it is still necessary to keep in mind that this is a live and running Linux system, so some inconsistency between experimental results are to be expected.

4.2.1 From disk

The results from running the experiment with parallel execution can be seen in Figure 10, and with sequential execution in Figure 11. The practical timings from Table 3 tell us that the rate at which we can read 1GB data from the hard disk on the actual system is approximately 13.4 seconds. Further, the processing takes approximately 4.8 seconds. Performed sequentially, we can add the timings, this becomes 18.2 seconds.

The best result we get is approximately 15.1 seconds, which was achieved with 2 threads and using 1GB of memory. The result is quite close to the practical limit of 13.4 seconds. It is worth noticing that the system performs much worse when using more than 8 threads. This is probably due to many unnecessary context-switches, since we can only schedule up to 8 threads on the system.

It it difficult to say why the system performs so much worse with less available memory. The smaller amounts of memory will result in less memory allocated per thread, and in turn result in many smaller I/O operations. However, the experiments with 10M available memory performed even worse than the 1M experiments. I am unable to explain why.

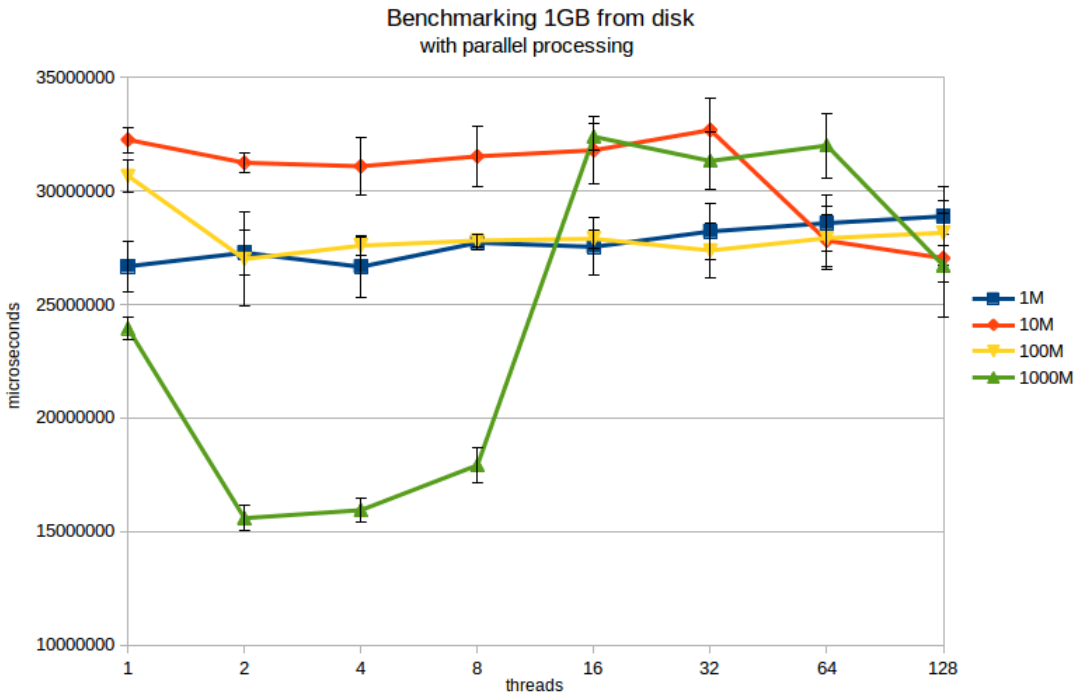


Figure 10: These are the results from running the benchmarking method on a 1GB data from a file stream, with varying amounts of available memory, from 1 megabyte to 1 gigabyte. Processing of the data was done in parallel. The standard deviation resembles the standard deviation from the disk-benchmarking and lies from 1% to 8%.

The key points are to let the framework have as much memory as possible, and not running more threads than there are hardware support for.

4.2.2 From memory

The results from running the framework on streams of data that are already loaded into memory can be seen in Figure 12 and in Figure 13. The first one shows the results from running the system with parallel processing, where the second exhibits sequential processing. This is the ideal case, where we have a fast input stream, and we have a somewhat large amount of processing power available.

The practical timings in Table 3 shows that copying 1GB of memory takes approximately 0.2 seconds, this will be done twice in the benchmarking. First, when copying data to the worker buffers, and then to the output buffer. Further, the processing itself takes around 4.8 seconds. In total, that sums up to 5.2 seconds, if it were to be done sequentially.

The results shows that when we can perform sequential processing, the buffer-size does not make much of a difference. Figure 12 clearly has very

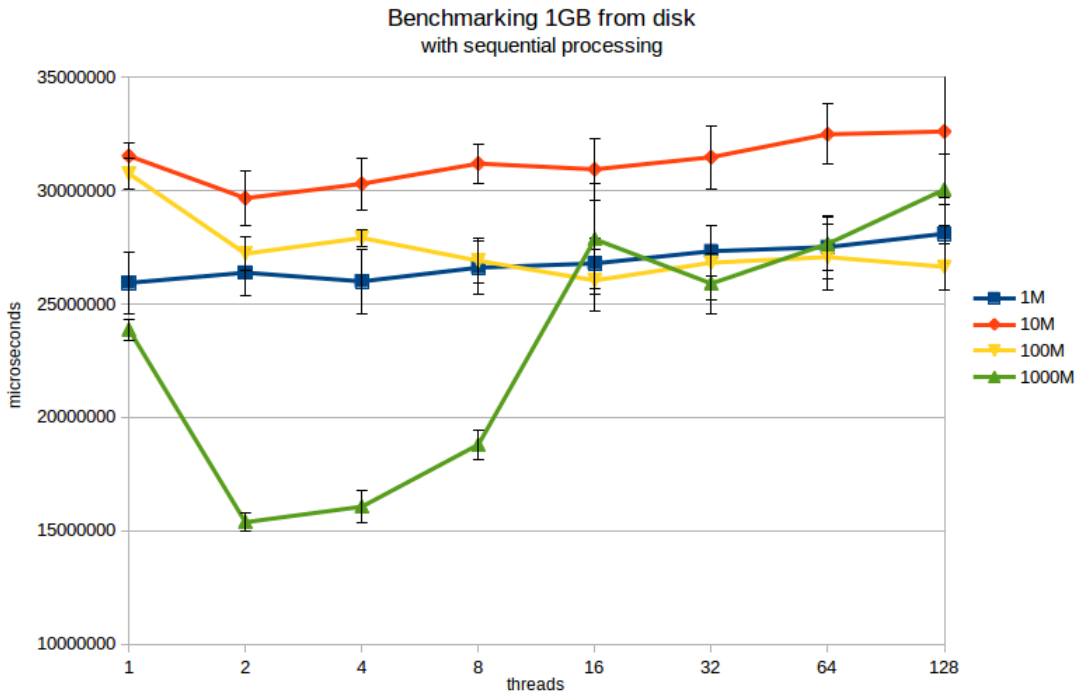


Figure 11: These are the results from running the benchmarking method on a 1GB data from a file stream, with varying amounts of available memory, from 1 megabyte to 1 gigabyte. Processing of the data was done sequentially. The standard deviation resembles the standard deviation from the disk-benchmarking and lies from 1% to 8%.

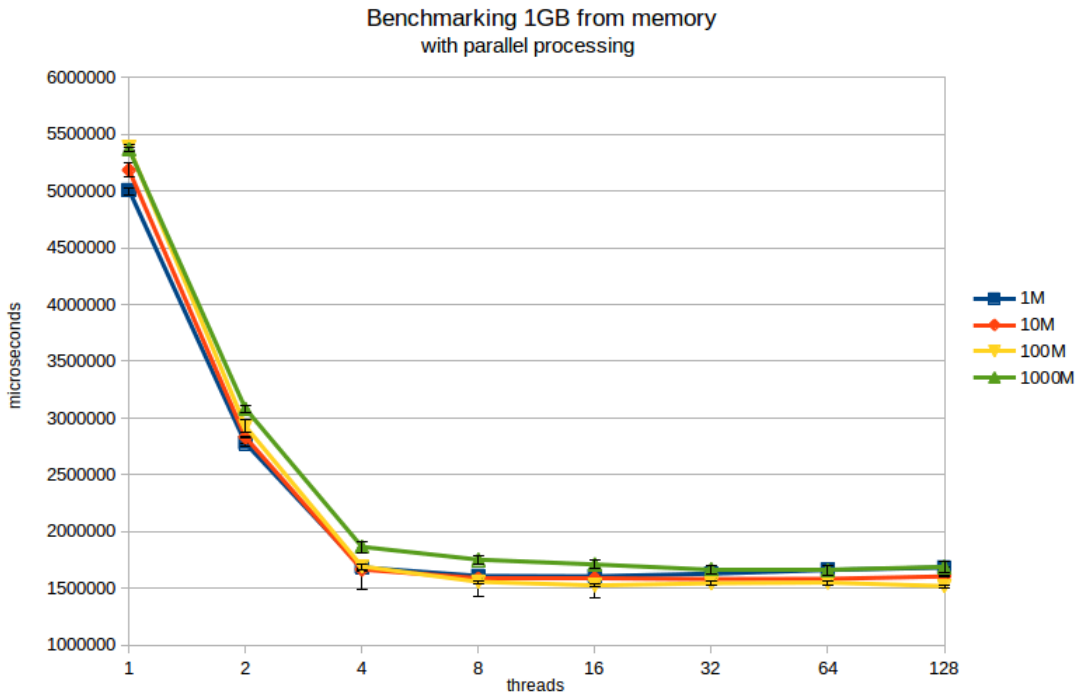


Figure 12: These are the results from running the benchmarking method on a 1GB data from main memory, with varying amounts of available memory, from 1 megabyte to 1 gigabyte. Processing of the data was done in parallel. The standard deviation resembles the standard deviation from the memory-benchmarking and lies from 1% to 2%.

identical results for all four sizes of memory. With sequential processing, however, using the most memory clearly makes the framework faster. Larger buffers results in more processing, before switching threads, which in turn results in less overall locking and unlocking. The sequential processing manages to perform the benchmarks in approximately 4.8 seconds. This shows that the I/O has been hidden completely, and only the processing itself takes time. The parallel processing performs the benchmarking in approximately 1.6 seconds, effectively making it approximately 3.5 times faster thant the sequential processing. There is some inherent overhead from multithreading, which is why we can't achieve a 4 times speed-up with 4 cores.

The results also shows that with 1 thread, the timings match with performing the calculations in sequence. Figure 13 shows that with smaller amounts of memory, the locking becomes more expensive, with multiple threads. With 1M memory and 16 threads, there is a speed-up from 8 threads. The same tendency can be seen with 10M from 64 threads to 128 threads. This behaviour is hard to explain, and I have no apparent explanation. However, with plenty of memory, the sequential system performs best at around 8 threads.

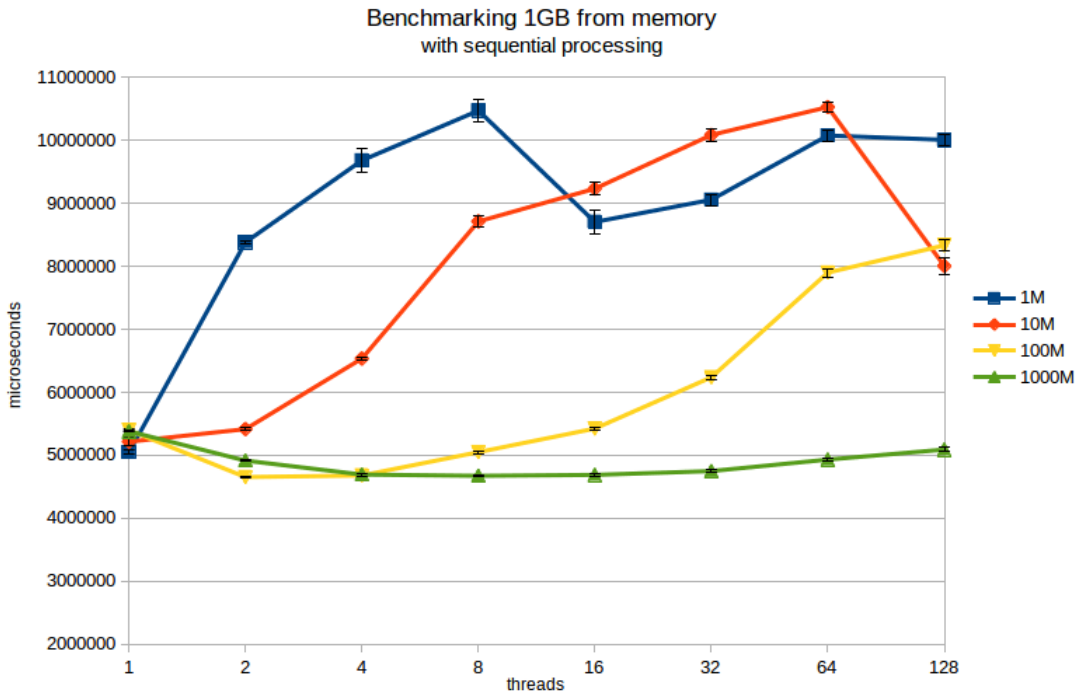


Figure 13: These are the results from running the benchmarking method on a 1GB data from main memory, with varying amounts of available memory, from 1 megabyte to 1 gigabyte. Processing of the data was done sequentially. standard deviation resembles the standard deviation from the memory-benchmarking and lies from 1% to 2%.

5 Conclusion

The *nbuf* framework has been built with one main purpose. It should allow users with less experience in writing multithreaded applications to perform optimal and parallel I/O. This entails two different criteria.

The first criteria depends on whether the framework really exhibits optimal I/O. The second criteria depends on whether the task of abstracting away the concurrency has created an interface that requires less multithreading experience to use.

5.1 Performance

Benchmarking the *nbuf* framework has been done with a method that finds minimum and maximum values, and calculates an average value. This is a very light computation, and as such leaves most of the execution time of the framework to performing I/O. This is a good thing, because it lets us see directly, how the speed of the I/O resources affects the performance of the framework.

When working with a slow stream, the framework benefits from having a large amount of memory. Further, having more threads than there are hardware threads for, only reduces the efficiency.

When using a slow hard disk drive, the sequential operation takes approximately 18.2 seconds, where the optimal configuration of the framework only takes approximately 15.1 seconds. The practical limit is approximately 13.4 seconds. This is independent of whether we perform sequential or parallel execution.

When working with a fast stream, the framework exhibits different behaviours when performing parallel processing and sequential processing. With sequential processing, having a large amount of memory helps. With parallel processing, the amount of memory doesn't make a large difference. In both cases, the optimal amount of threads matches with the amount of hardware threads.

The the benchmarking task can be done sequentially in approximately 5.2 seconds, and the framework performs the benchmarking task in approximately 1.6 seconds. This is a factor 3.5 speed-up, which aligns with the 4 hardware cores on the system at hand.

To sum up, both with slow and fast I/O streams, speed-ups were achieved as expected. The highest throughput was obtained when having a fast stream, and parallel processing. When working with a slow stream, allocating extra memory gives a higher throughput. Using more threads than the hardware supports never gives higher throughput.

5.2 Usefulness

Producing on the *nbuf* framework has resulted in a C++11 library that can be included by people with less experience with multi threaded applications. The library has been proven to increase performance whenever data is too large to

fit in main memory, or when a small memory footprint is desired and larger amounts of data needs to be processed.

The framework gives access to native threading on most *NIX, while exposing an interface where no locks or synchronization has to be handled. The user of the library still has to know to what extend he can parallelize the task at hand, so that the task is solved correctly. This does require some amount of knowledge about how concurrency works.

Concurrency is inherently hard, and the framework performs correct concurrency. Users who decide to include this library will be able to perform tasks with efficient and parallel I/O, without having to debug for race conditions. This was the intention, and the framework solves the problem.

The framework does not yet support the queue mechanics, which is necessary for computational tasks which require more synchronization. This limitation decreases the usefulness of the system. Implementing such queue mechanic can be done rather easily, and therefore this is *not a big problem*. The design has already been laid out.

The *nbuf* framework is already available for people to work with through GitHub[10], and I expect to submit it to Boost[11] if the code reaches maturity levels beyond this master's thesis.

[10]The library is available on http://github.com/ath88/nbuf.

[11]Boost is a repository for peer-reviewed C++ libraries. It can be found at http://www.boost.org.

6 Future Work

The thesis work has been concluded, but there are still some missing features in the *nbuf* framework, and some interesting topics to explored.

6.1 Missing features

The framework misses complete support for sequential processing and in-order output. It can be implemented in the current framework, as described previously.

6.2 Further experimentation

While the experimentation clearly shows the effectiveness of the *nbuf* framework, the experiment results also shows some odd behaviour which is not directly explainable, without more experimentation. This requires more time constructing test cases, and possibly multiple hardware setups.

Figuring out how the framework reacts in different environment can easily be a project in itself. The settings that could be varied in new experiments could be;

- **Different memory footprints** - Currently only 1MB, 10MB, 100MB and 1000MB were tested. I would suggest trying with a *power of two*-range instead of the tenth power. This will hopefully give a more gradient view of how the framework behaves.
- **Other disk setups** - Testing with more advanced disk set-ups would be interesting. It could be SSDs, raid configurations or network file systems.
- **Other data sizes** - In this project I only ever tested with 1GB of data. It was a balanced amount of data which could still fit in local memory for string streams, and didn't take too long to complete the experiments. With more time, larger files could be used. This could also help strain the operating systems output buffers.

6.3 I/O Throttling

When large buffers are used, it takes some time before the first worker gets to processing. This will, at some point, result in lower performance than necessary. If the first thread reading would only fill a fraction of its buffer, it could start processing a lot quicker, and in turn, the entire task could be finished earlier, since more latency would be hidden.

A method could be to let threads fill up 10%, then 20%, then 40%, then 80% and finally 100% for each reading operation. This will allow the threads to start processing quicker. If plenty of computation power is available, then this will result in a faster overall runtime-

6.4 Logging and Error handling

If an error happens in the *nbuf* framework, an exception is thrown, and it is up to the user to handle it. There are some exceptions which could be handled by the framework, and the framework could in turn react by correcting, or throwing a different exception.

When including a third-party library, it is nice to have some kind of output, either on *stdout* or *stderr*, or to a stream, to be able to figure out what happens, or for debugging purposes. Currently, no such mechanism is available, and the developer has to insert debugging statements directly into the library.