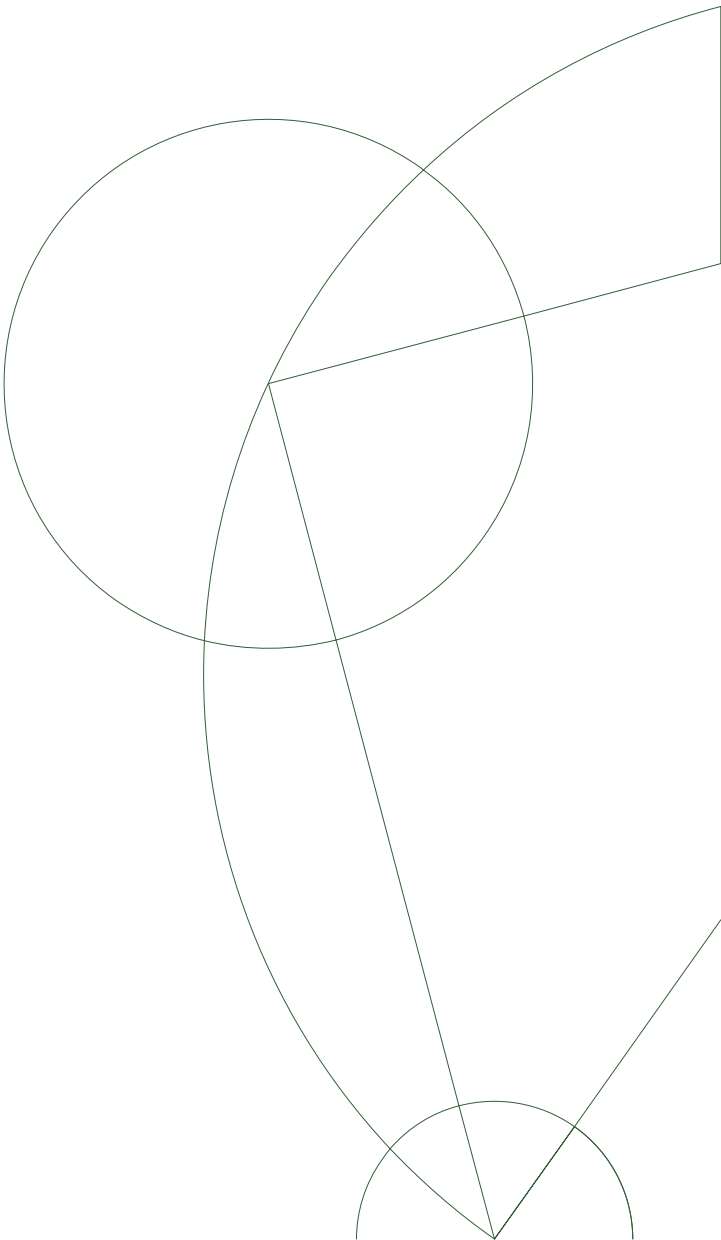
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Asbjørn Thegler - asbjoern@thegler.dk

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 threading synchronization primitives made available in C++11. 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 allows programmers 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`. The current version is tagged as *v0.1*.

Changelog

The thesis study was handed in on the 19th of October 2015. The following is a list of changes that has been done after this deadline.

- Changed email address on title page.

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 invest large amounts of money into this new trend[2], 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[7]. He defines them as **Velocity**, **Volume** and **Variety**.

Velocity refers to the speed at which data is produced. Previously, most data streams could be stored 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 2014, Mark van Rijmenam from Datafloq[10] 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 findings that can be found in such huge datasets. New mechanisms will have to be created, which can be used to visualize interesting parts, such that decisions can be made on an informed basis.

Value refers to the potentially lucrativeness of the Big Data industry. While utilizing huge sets of information about customers can 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 study 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*. It will be done by creating a library that uses several buffers to swap around, often referred to as *Triple Buffering*.

1.1 Terminology

The first well known occurrences of the term *Triple Buffering* stems from the computer graphics industry[11]. This is a technique where the graphics card renders images into three different buffers. *Triple Buffering* is a solution to a problem with the *Double Buffering* technique. In *Double Buffering*, a two buffers are used; a *front-buffer* and a *back-buffer*.

In *Double Buffering*, the graphics card renders a frame into the back buffer, and the buffers swaps to allow the screen to show the picture from the front buffer. Since no synchronization exists between the graphics card and the screen, a buffer swap can happen as the screen is reading from the front buffer, resulting in what is known as *screen tearing* where the screen shows two different frames at once. An attempt to fix this problem is know as *vertical sync*. This includes adding artificial delays to the graphics cards renderer, to match the frame rate of the consuming screen, effectively slowing down the graphics card.

To better solve this problem, a third buffer is employed, effectively making it *Triple Buffering* rather than *Double Buffering*. The graphics card can now switch between two back-buffers, and always have a free buffer to write a new frame to. If the screen is too slow, frames will simply be lost, with no greater loss to the viewer. This extra buffer obviously require extra memory, but modern graphics cards have plenty of such.

1.2 Triple Buffering Big Data

Triple Buffering within computer graphics is a different concept than *Triple Buffering* within data processing. There are many similarities, however, and we can translate parts of the solution to increase data processing capabilities. The bottlenecks of a graphics card are namely the rate at which the graphics card can render images, and the rate at which the screen can show those images. If the screen has a low frame rate, then a very fast graphics card is useless, and vice versa.

This translates to the I/O problems we encounter when working with data processing. If we cannot process and store data at the same velocity they are being generated in, then we have to throw some data away, or slow down the data generator. The computer graphics industry increased throughput by utilizing more space, by adding an extra buffer. This concept can solve, or at least mitigate, some of the I/O problems related to processing data at high velocity.

The focus of this study is to produce a 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 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 study in no

way introduces new technology or uncovers scientific ground. This is a study in combining existing technology and knowledge to create a highly optimized and effective library.

1.3 Motivation

This study did not manifest from thin air. Many people have contemplated building such a library. *Triple Buffering* has been used many places, many times before, and it is a well-known technique. The reason why this study was undertaken now, and not 10 years ago or in 10 years, is a combination and collision between several factors:

- **Data Growth** - According to CSC, the amount of data will increase by an astounding 4300% by 2020, since 2012[3].
- **Data Bus Speed** - The growth in data transfer rates has not followed the growth in data volume, and as such, it is necessary to build tools to better utilize the available transfer rates.
- **Open Source Popularity** - Open Source has gained traction, and becomes more and more popular, according to Wired[8].

Having a library that exerts these capabilities, that can be scrutinized and improved by everyone, will help organizations all over the world when they have to process fast data streams.

There are some existing solutions to this problem already. The Apache Foundation has both *Apache Hadoop*[1] and *Apache Storm*[2], which both deploy a distributed mapreduce cluster[4]. This study will separate itself from these solutions by being an library rather than a service. It will require a programmer to include the library in software, rather than configuring a distributed cluster.

1.3.1 The I/O Problem

Moving data to and from computational resources inherently results in I/O operations. Depending on the transfer rate of the data storage, these operations quickly becomes a bottleneck for the entire process. If we then proceed to not extract data from these streams at all time, then a lot of time time is spent on waiting. In reality, the process could pre-load data during processing and reduce the time spent on waiting for I/O. To process data as fast as possible, we want to fully utilize the I/O resource.

When processing data, the traditional method consists of the following steps:

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

[1] Apache Hadoop, `https://hadoop.apache.org/`
[2] Apache Storm, `http://storm.apache.org/`

If the computational task of processing or transforming the data is very large, the I/O becomes negligible. In this case, most time is spent on step 2, and interlacing the I/O operations will not give a significant improvement. Parallelizing step 2, however, will allow for very significant improvements. This will include adding more workers that can process the data in parallel.

If the computational task of processing or transforming the data is very small, most of the execution time will be spent waiting for I/O. An example could be a simple copy of a file from one disk to another. In this case, there are two I/O resources, which are being used in step 1 and step 3, namely the *input stream* and the *output stream*. This means that only one of the resources are being used at any given moment. To improve this mechanism, we could have two concurrent workers, each occupying one resource, to utilize both of the resources at any given time. In case of a computational task, a third worker could then perform step 2, which is exactly how *Triple Buffering* works.

This study will focus on the case where neither the execution, nor the I/O operations are negligible. In this case, it is important to be able to parallelize step 2, while having additional workers occupying the I/O resources at all times. The library will be most useful in these cases, but will also be applicable in other cases.

1.3.2 The Generic Problem

When programmers develop software, they are generally encouraged to utilize established libraries whenever possible, instead of relying on their own ability to create both complex and correct code. Often, a programmer encounters a specific problem, that 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 utilizing widely used and established libraries, these problems are often already addressed by many other programmers.

Within the Open Source community, it is common practice to make ones code available for others to use. In theory, when many organizations use the same tool or library, errors are found, reported and corrected much faster than when code is only used privately or within a single organization. Over time, this often result in libraries that are used globally, and has many contributors. This is known as crowd-sourcing.

When a general library that solves a general problem does not exist, programmers must write custom code that solves the problem. This will result in many programmers solving their specific problem over and over again. They cannot share their solutions, because they have all solved their own specific problem, and every problem has slight differences. At some point, a programmer will realize that there is a pattern and pick up the task to attempt to build a generally applicable library.

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 irrelevant to the task at hand. Reading the manual of many tool can be a daunting task, while often simple problems can be solved faster with custom code. Attempting to identify what library might fit the current problem can be a sizeable project in itself.

Open Source projects tend to have organic growth. Without tight steering from some small group of committed programmers, a project will become monolithic and it will suffer from *software bloat*. A tool or a library is usually created to solve a problem. Over time, the problem-body might be extended to include more features. In these cases, many people using the tool or library does not need the newer features, and to them, the features are merely an annoyance. In 1996, Jamie Zawinski wrote the following about *software bloat*[12]:

> Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.
> – Jamie Zawinski, 1996

This is clearly put very comically, but the reality is that this often happens. Software develops until it becomes too slow and cumbersome. At this point, it is replaced by new and lean software, which overtime grows until it becomes almost identical to the software it replaced. To avoid this tendency, it is important to keep a very strict process, when deciding what features to include.

1.3.3 Use Cases

The library that is built over the course of this study could be used for several purposes. Large amounts of data are being processed in many places. Following are some use cases where using such a library could be a good solution.

Modern hash algorithms are designed to ensure the integrity of a collection of data. As an example, when large files are transported from one place to another, it is important to ensure that the data is intact upon arrival. A hash-value can be calculated before and after transport, to be compared afterwards. In this case, the tool used for transmitting the data could use a buffering mechanism, which calculates the hash value, as it is read from disk, and as it is written to disk on the receiving end. This would be far more efficient, than to read the file from disk after transport, only to calculate the hash-value. The data is already passing through memory on both sides of the transportation, and utilizing the data here would be beneficial.

Additionally, the command line hashing tools *md5sum* and *sha512sum* both have implementations that read 512 bytes at a time, which results in many I/O operations, in case of large files. Many of those operations could be avoided, if more memory is available for multiple larger buffers. Further, using a concurrent approach to this problem, could introduce some latency hiding, which will improve the runtime of the tools.

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. An example could be a network of rain water gauges. When there is no rain, it is unnecessary to store the information.

The *ESS project*[3] 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, for situations where questions arise about data integrity. 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 interesting. 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 up time to perform other tasks. With the suggested library, including such functionality would be technically trivial.

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

2 Theory and Analysis

This section will explain the ideas and concepts that are applied during the design and implementation of the *nbuf* library.

The first section will explain how concurrency is handled and how correctness is ensured. This also entails how to ensure that the library will always terminate when used correctly.

The second section will explain how the study handles aspects related to data, I/O and how to handle the enormous amounts of data, and how different data streams affect the resulting library.

The third section will reason about what technical results can be expected from this study. This will mainly focus on benchmarking, and how different configurations can affect the measured results.

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[6].

Deadlocks and deadlock prevention is paramount when working with concurrency, and anyone reading this should know exactly why. If the reader does not know how deadlocks can happen, I will suggest reading Concurrent Systems, by Jean Bacon[1]. 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 understand 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 a process reacts and interacts as expected. Finite-state diagrams are trivial to both create and understand, and can be used to reason about a process. They can be used as a development tool and as documentation about a certain system or process. It gives an abstract overview of a concrete process.

Figure 1 is an example finite state diagram. This diagram is quite simple, and shows how a door with a lock could 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,

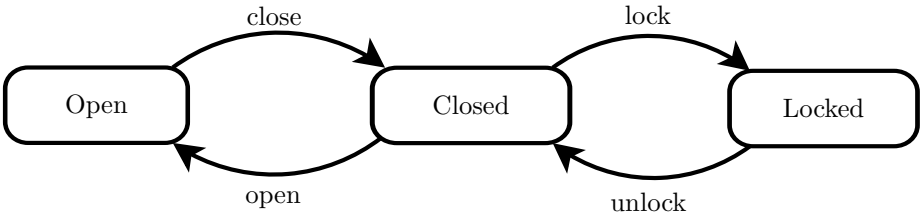


Figure 1: This is an example of a finite-state diagram. It represents a door which can be in 3 states, and there are 4 different transitions.

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 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* library works, finite-state diagrams will be used. These help understand how the worker 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 can be done by using multiple threads which can be scheduled individually on different cores. This allows multiple threads to run simultaneously, 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 are not used to concurrency. Those problems are often known as race-conditions, which happens when two threads use the same section of memory at the same time. To solve these problems, it is necessary to use some kind of synchronization, to ensure *thread-safety*.

Most synchronization mechanisms are based on the hardware instruction *test-and-set*[4]. 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 and abstract synchronization mechanisms in systems. The primitives that are relevant to this study are the following:

- **Condition Variable** - A condition variable is a primitive that can be used to make threads wait for specific events.
- **Mutex** - A mutex is merely an advanced locking primitive. The name is derived from *mutual exclusion*. It is intended to give one thread exclusive access to a critical section of code.
- **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*.

[4]The *test-and-set*-instruction is an atomic operation that tests some condition, and changes a values if the condition was true.

- **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 **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 **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 a taken mutex. When the mutex is released, only one waiting thread is notified, if there are any. This does not guarantee that the notified thread is the one that has been waiting for the longest time. This can result in *starvation*, meaning that one thread can risk not gaining the mutex, even though it has been waiting for a long time. The thread is then *starved*.

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 that is created by the calling thread, and passed off to a worker 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 terminates, 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 does not 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 requires very complex mechanisms, 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 data to and from a stream, it is important to interpret the data correctly. 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 semi-colon. This solution saves a lot of space, but makes parsing harder, since it is necessary to inspect the data during demarshalling. 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 in other cases 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 concept they call *Protocol Buffers*[5]. 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 to keep 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 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 programmer uses the *nbuf* library, it is recommended to use a protocol buffer, when processing the data. However, since the *nbuf* library 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. If *Protocol Buffers* is not used, it is important to be extra careful when interpreting the data. A solution could be to include some sort of sanity-check, that could alert the programmer if parsing did not work is not as expected.

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 it has been played to the viewer.

When using the *nbuf* library, using buffers that allows a single thread to contain the entire file in its local buffer, will make the library behave inherently sequential. This reduces the multiprocessing capabilities of the library. 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* library seeks to take an agnostic approach to how data streams behave. The library should yield 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 abstract methods, on the streams that are supplied. As an example, it cannot check if a file is open, since there is no guarantee that the stream comes from a file at all.

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

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[5], namely a *network stream*.

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

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 connection, 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 yet be sent. 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 platter 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 continue 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 library. 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* library, we want to ensure that optimal I/O has been achieved. If the library can handle fast streams concurrency, slower streams should be trivial to handle.

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

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

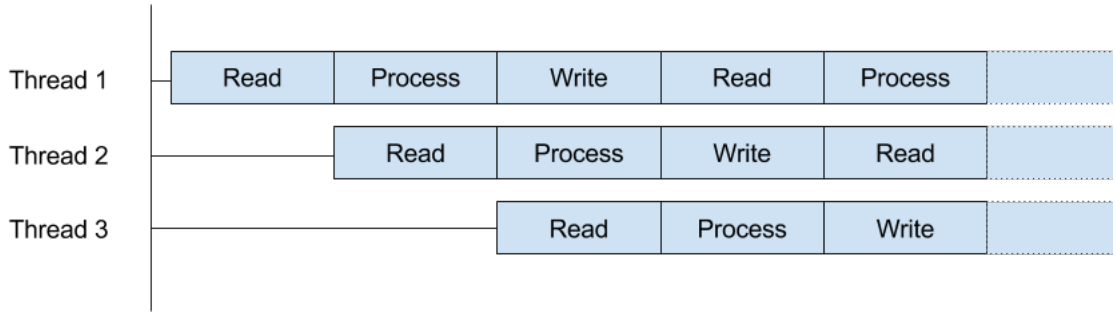


Figure 2: This is an example of how the *nbuf* library could interlace, if all three tasks are similar in time taken.

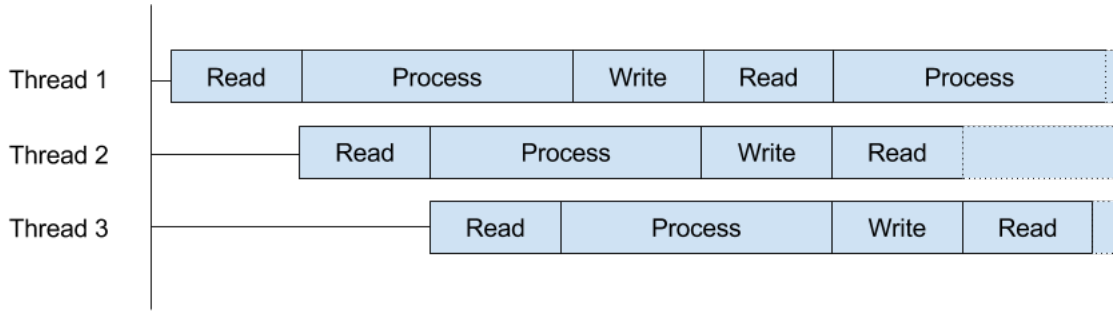


Figure 3: This is an example of how the *nbuf* library could interlace, if the processing task takes more time than the read and write tasks, and the processing task can be done in parallel.

systems, multiple threads can execute in parallel. If every process only contains a single thread, no memory is shared between threads, unless explicit process memory sharing is used.

If a process contains several threads, the process can have several threads executing simultaneously on multiple cores. When tasks can be done in parallel, the complete task can be finished faster than if they were performed in sequence.

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

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

Figure 3 is an example of how the library 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.

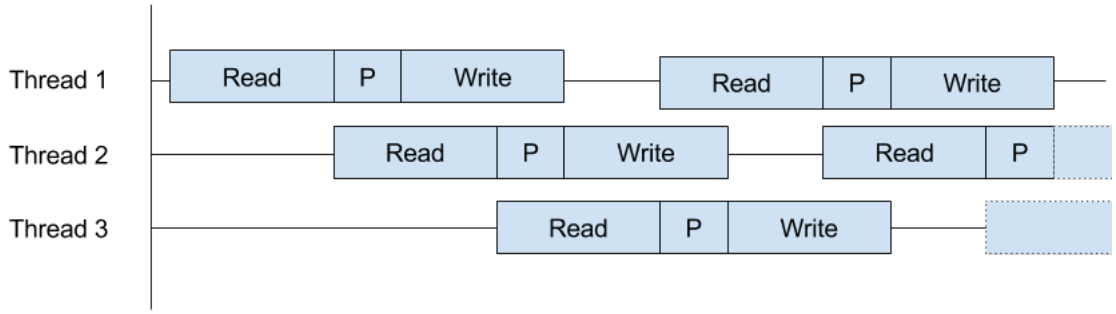


Figure 4: This is an example of how the *nbuf* library could interlace, if the processing task takes less time than the read and write tasks.

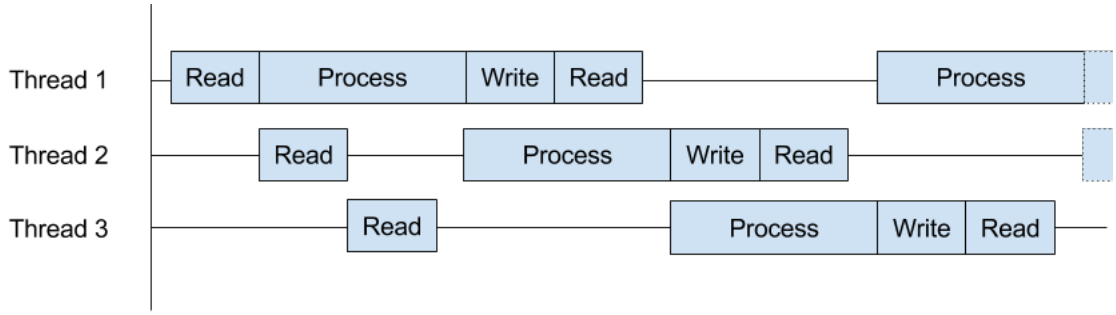


Figure 5: This is an example of how the *nbuf* library could interlace, if the processing task takes more time than the read and write tasks, but the processing task can only be performed by one thread at a time

This can be seen in the figure, since after the third thread has completed reading, it takes some time before the first thread reads again. In this case, more threads are needed to fully utilize the read- and write-resources.

Figure 4 is an example of how the library 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, since the first thread would be available to read after the second thread has finished reading.

Figure 5 is an example of how the library 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 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 a domain specific language, in order to obtain the desired results. While the details of GPU programming is not part of this study, the concept of a worker thread scheduling 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. In this case, the *nbuf* library 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 study, but it indeed important to keep this option open, should it become a requirement in the future.

3 Design and Implementation

This section will explain how the *nbuf* library has been designed and implemented. First I will identify and elaborate on the inherent requirements of such library. Secondly, I will elaborate on the abstract idea of how the library handles concurrency. Then I will elaborate on how the library is to be used, and finally a technical description of how it has been built.

3.1 Technical Requirements

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

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

- **Efficient and Parallel I/O** - The *nbuf* library must prioritize keeping the I/O resources busy as often as possible.
- **Limited Space Guarantee** - When processing data, the library must be able to keep a strict upper limit on allocated memory.
- **Custom Processing** - The programmer 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 programmer 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 library must terminate if and only if the input stream is finite.

These features are all important, if this library 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 study attempts to maximize the use of the available I/O resources on a system, when processing incoming data. If there are solutions which yields a better or larger rate of processing data, then the *nbuf* library 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 benchmarking the library.

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 one thread can not wait for two resources at the same time.

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

In a new library, it would be prudent to evaluate and rethink this *Triple Buffering* mechanism. In cases where there are is a high amount of processing work, a *Triple Buffering* mechanism will be waiting for the thread performing the real work, before the buffers could be swapped around. A good computer scientist might be able to optimize the processing work, but this has a limited usefulness, and would only improve on a single implementation.

On 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 processing work can be parallelized, the extra processing power can be utilized, up 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 library must be able to use multiple processing cores, and support the programmers in performing parallel work.

3.1.2 Limited Space Guarantee

The available main memory on a computer system is rarely a hard limit. Usually, when a process starts allocating more main memory that are available, some data in main memory is moved to a swap-partition on the systems disk drive. This results in very high delays when allocating more memory, and when trying to read data which has been moved to the swap-partition. When trying to work efficiently, this is a performance killer, and any processes causing 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 should not be used in critical system set-ups. It is a minimum requirement for all software to deallocate unused 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 introduced the term and programming idiom *RAII*, short for *Resource Acquisition Is Initialization* [9]. 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* library 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 programmer using the *nbuf* library 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 *nbuf* library.

The worker performing the real processing must be programmable by the programmer. When the worker has a buffer full of data, it must be up to the programmer 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 programmer intends to do with the data in the buffer is to no concern of the library, but the library should support the intentions of the programmer.

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

- **Stream-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 library. To ensure that the programmer can solve the task at hand as an isolated task, it is important to split the data in a sensitive way. This will be done by defining a *stride*, which is the size of *independent data chunks*. The worker threads will only read entire *strides* into its working buffer. This way, there is no risk that one stride is split across multiple workers.

3.1.4 Input/Output Sensitivity

In an example case, we might want to find the minimum value, the maximum value and the average value in a file. The naive implementation 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 library.

Some other cases, 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 an md5-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 the *nbuf* library, since this could help performing latency hiding.

To accustom to this need, it must be possible to decide that the library 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 programmer 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 library 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 retrieved from the library.

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 outputted data such that it does not occupy more memory than necessary. This is indeed relevant in the use case where the programmer receives data from sensors, which in periods doesn't measure interesting data. This could be a rain water gauge, which reports 0 millimetres for many months a year.

3.1.7 Guaranteed Termination

The library must terminate when it reads from a finite stream of data. The library can never be fool-proof, but it should always terminate when it receives an indication that there are no more data, often an EOF[6].

Futher, if the programmer introduces an infinite loop during data processing, this promise cannot be kept, but the library should never result in a deadlock or a spinlock.

These are the technical requirements that I have decided the *nbuf* library 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.

[6]End Of File. This is the common indication that there are no more data to be fetched from a data source.

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 framework. 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* library entails a few tasks:

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

After processing all 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, the library 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 it is possible to establish a method where less memory sharing is used, it will make the library 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.

3.2.2 The *nbuf* workers

Each worker has allocated its own buffer which it will use for reading into, processing in, 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 library 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 moves through the states in the diagram. Remember that there are a finite amount of workers, and that the input resource will be empty, at some point. In cases where the input will never empty, the framework can not be expected to terminate.

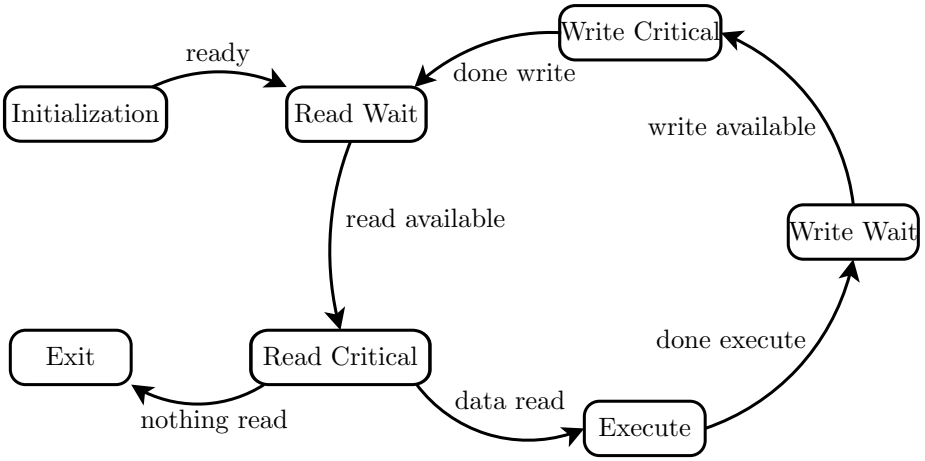


Figure 6: *nbuf* worker state diagram. This finite-state diagram shows how the workers can change states, based on events. Further explanation on how the transitions happens can be found in the related transition table which can be seen in Table 1.

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.

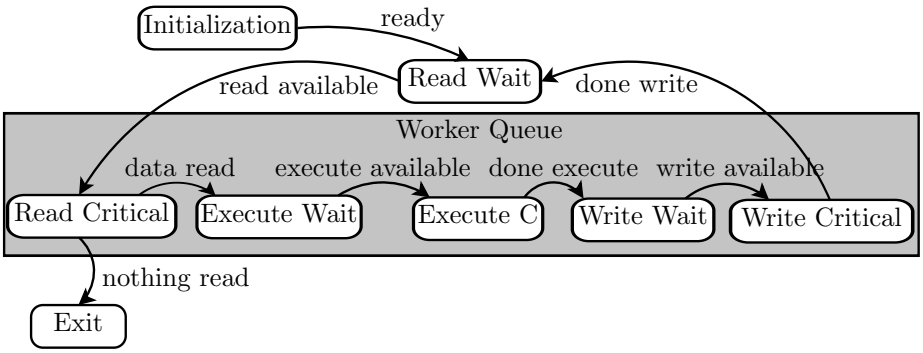


Figure 7: *nbuf* worker state diagram with a queue. This is an evolution from Figure 6. The workers cannot overtake each other when they are inside the queue.

All workers begins in the *Initialization*-state, and will move into the *Read Wait*-state. When the read resource is available, one 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 one of the workers which are in the *Read Wait*-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 programmer 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, if another worker is occupying the next state. When the next state 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 the **Input/Output Sensitivity**-requirement can not be supported in this way. To comply with this requirement, the library must support utilizing an altered state diagram which can be seen in Figure 7. In this diagram a FIFO[7]-type queue has been added. This means that the worker thread that entered the

[7]First In, First Out. The alternative type of queue is a LIFO, Last In, First Out.

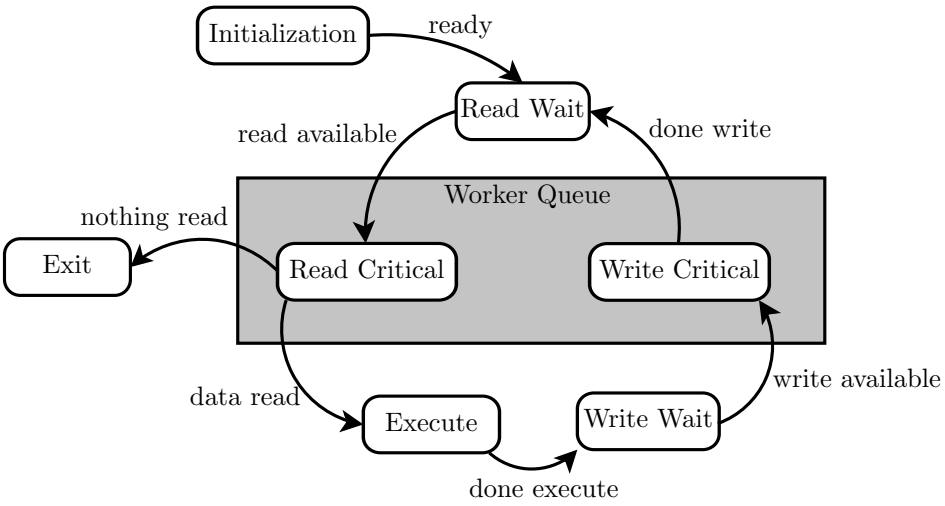


Figure 8: *nbuf* worker state diagram with a smaller queue. This is an evolution from Figure 7. The workers cannot overtake each other when they are inside the queue, but the execution step can be performed and finished in parallel. This means that workers entering the queue at the *Read Critical*-state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

queue first will reach every new state before every other thread. Further, an *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 completely 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.

If we want execution to be sequential, but the output ordering does not matter, we can use a different kind of queue. The related finite-state diagram can be seen in Figure 9. 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

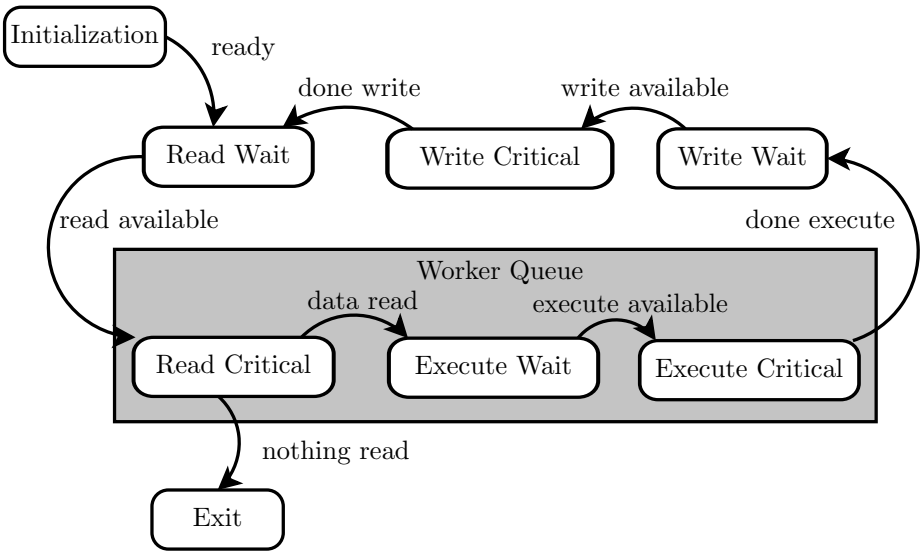


Figure 9: *nbuf* worker state diagram with a smaller queue. This is an evolution from Figure 7. The workers cannot overtake each other when they are inside the queue, but the write step can be performed and finished in parallel.

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* library. A library 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* library interacts.

```
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 master thread can be seen in algorithm 1. It is a very simple algorithm; it creates the worker threads, waits for them to finish, and gathers the results, combining them into an aggregate result. In the case of a sequential execution, the result is stored in shared memory, and no combination is required.

```
Worker thread method;
begin
    allocate promise struct;
    calculate and allocate buffer memory;
    more data = true;
    while more data do
        lock input resource;
        read to buffer 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 from buffer 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.

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 number of strides that can fit in in the buffer. This is done to avoid splitting a stride across two workers.

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 programmer 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. At this point, the promise will consist of aggregate data from this single worker.

In case of a sequential execution, the workers will be writing to a promise shared between all workers. An additional lock protecting this section will have to be locked and unlocked.

3.4 Library Interface

When building a library 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 tools tend to require a ton of configuration to do the simplest of things [8], while others merely have sane defaults, and requires configuration to use the more advanced features[9].

First, I need to identify what input is required to configure the *nbuf* library. Here is a run-through of the configuration that could be done with the library.

The library will need an **Input Stream**. The sensible default would be to read from *stdin* [10]. This stream should inherit from a generic type, to allow many different kinds of input streams, as mentioned earlier.

The programmer must supply a **Processing Method**. This is the method used to process a single stride. The library can default to doing nothing with the input data, which would result in a simple copy or move of data.

If **Sequential Execution** is desired, this bool should be set to true. The default should be to perform parallel execution.

In cases where the data processed by the library is already stored on the file system, the programmer can disable the output with an **Output Stream Enabler**. This is a bool, and can default to true, meaning that the library will enable output to the stream.

In cases where output from the worker buffer is desired, an **Output Stream** must be supplied. A sensible default would be to use *stdout*. This stream should inherit from a generic type, to allow many different kinds of input streams, as mentioned earlier.

If the programmer desires to filter the output data, a filtering method can be enabled. This requires an **Output Filter Enabler**. This is a bool, and can default to false.

When a programmer has decided to filter the output, the filtering mechanism must be built. An **Output Filter Method** can be supplied to the library. It can default to terminating the program with a friendly message, since enabling filtering, but not implementing a filter would signify a user error.

When both the entire output, and filtered output is desired, an alternative **Output Filter Stream** can be supplied. It can default to the regular output stream. This stream should inherit from a generic type, to allow many different kinds of input streams, as mentioned earlier.

The programmer should be able to test what amount of threads would yield the best performance. The **Number of Threads** can be set, and it can default to an arbitrary number, e.g. 3 threads.

The library will adhere to a strict memory limit that should be configurable. The programmer can supply a numeric **Memory Limit**. The library can default

[8] 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.

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

[10] 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*.

to an arbitrary number, e.g. 100 megabyte, which most systems should have readily available.

The stream of data will be structured in some way. The processing will be done in data chunks aligned with a certain **Stride Size**. The programmer can supply this, and it can default to 1KB of data.

If the output should be ordered in the same way as it was read by the library, the **Input/Output Sensitivity** must be enabled. This is a bool, and can default to false.

These are the configuration parameters that the library 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 programmer configures the library.

3.5 Multithreading with *std::thread*

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

The new thread library included in C++11 is referred to as std::thread. It offers a much more abstact API than pthreads, which allows the programmer to ignore certain aspects of multithreading. This allows for higher productivity and less complex code. This, in turn, makes the code easier to maintain.

When the *nbuf* library is initialized, 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, when they have finished processing data. 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 library. A combined, accumulated set of data gathered across the entire input stream.

The library has been designed to require as little synchronization as possible. While this has decreased the complexity of the library, 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 required for a thread at the same time. These two mutexes will manage each of the critical sections, related to the I/O resources. The fact that no thread requires both resources at the same time completely eliminates the risk of deadlocks, since no thread will be waiting for a resource, while occupying another resource.

[11] The POSIX thread library is also known as *pthreads*. POSIX threads have implementations for nearly every platform.

When a critical section is required in the execution-step, it will also require a mutex. In this case, more synchronization is necessary, since the workers will also have to 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* library handles concurrency with the *std::thread*-library.

3.6 Current Limitations

The current[12] implementation of the *nbuf* library 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 added in the future.

- **No Queue** - The library 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.
- **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 technically trivial to implement, but requires fairly much code, thus they did not make it to the final result from the study.

[12] As of the 19th of October '15.

4 Experimentation and Benchmarking

The *nbuf* library 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 library can work with system I/O.

4.1 Experimental Setup

The library solves a practical problem; it attempts 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.

The *throughput* of the library under different conditions would be a good quantifier. 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, and more on the bandwidth of the I/O resources.

The benchmarking calculation is a simple search for minimum- and maximum-value and calculating an average value. The naive implementation is simply a sequential loop which sums the values, while keeping a current minimum- and maximum-value and counting the amount of values in the data. This implementation can be used as a baseline to test how much faster the *nbuf* library can perform these calculations. If the library is slower than this naive implementation, then the library 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 machine 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 library 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 recently used files in memory, as long as the memory is not needed for anything else. This has been a

mystery to many new Linux-users, and has sprouted many helping words[13]. This easily results in wrong experimental results, when benchmarking throughput from disk.

Imagine doing an experiment two times, 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 result for the second run, for the wrong reasons.

Second, the operating systems does not write your file to disk, just because you tell it to. It has to schedule all disk I/O and therefore it keeps 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 operating system decides to write it. The size of this buffer can vary, depending on how much memory the operating system keeps to itself. When the buffer is full, the operating system is forced to write to disk, and let the program wait, until the operating system has enough memory to copy the data from the program. To show the large wait on disk input and disk output, a simple experiment has been conducted.

The I/O experiment is meant to show the difference between return times when performing disk input and disk output, when the operating systems schedules the I/O. First, a file with 1GB of data is located on a file, on a consumer-grade hard disk drive. The operating system file cache is emptied, a timer is started, and the program starts reading the entire file into main memory. When returning from the read operation, 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 operating system 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, along with the respective RSD[14].

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

4.1.2 From memory

In comparison, main memory on a system suffers from much lower latency than a hard 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 location to another. It is not necessary to

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

[14] The *Relative Standard Deviation* is used, to make it easier to compare the results from different experiments. This is also known as the *Coefficient of Variance*.

Action	Average Time Taken	RSD
Reading 1GB of data from disk	13411522 μs	2.81 %
Writing 1GB of data to disk	541844 μs	1.28 %
Copying 1GB in memory	184583 μs	0.46 %
Sequential Processing 1GB in memory	4757252 μs	0.68 %

Table 3: The results from the disk I/O experiment on the actual hardware, along with their RSD. 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.

test reads and writes, since they are essentially the same, when performed 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 are not working with a spinning metal platter that has to seek and find.

We can use these results to gain a higher confidence, when benchmarking the *nbuf* library. When testing the library 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 library 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 RSD. 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 library performs much worse when using more than 8 threads. This is probably due to many

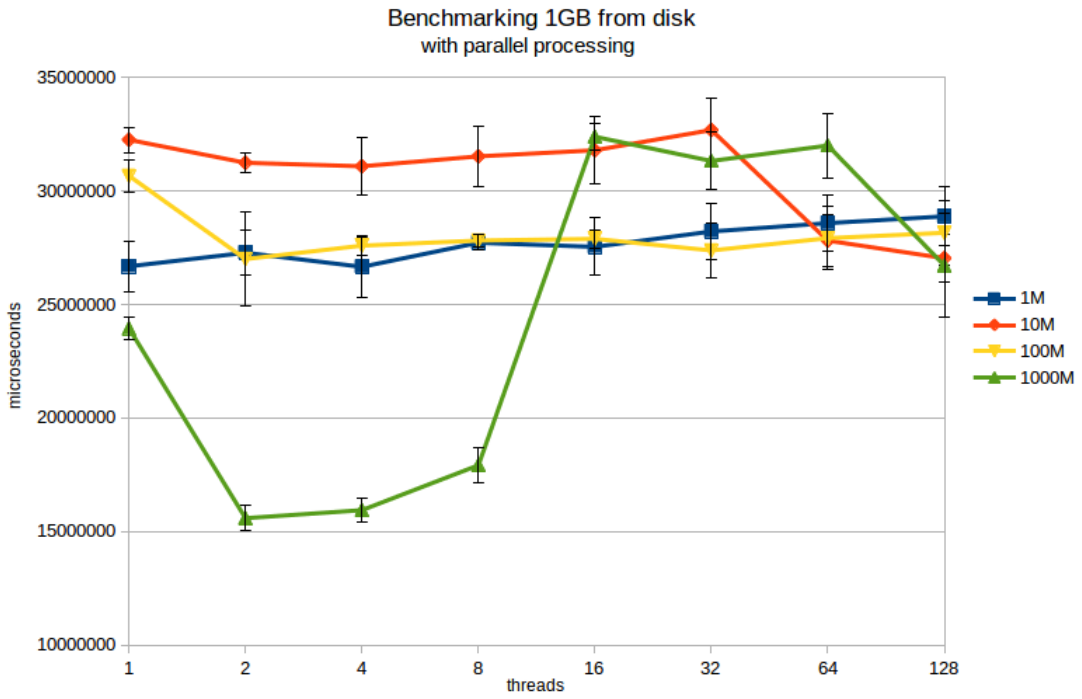


Figure 10: These are the results from running the benchmarking method with 1GB data from a file stream, with varying amounts of available memory, from 1 megabyte to 1000 megabyte. Processing of the data was done in parallel. The RSD resembles the RSD from the disk-benchmarking and lies from 1% to 8%.

unnecessary context-switches, since we can only schedule up to 8 threads on the system. This is known as *thread-thrashing*.

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.

The key points are to let the library have as much memory as possible, and not running more threads than there are hardware support for. It is not necessary to give the library more memory, than the size of the data it has to process.

4.2.2 From memory

The results from running the library 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 uses 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.

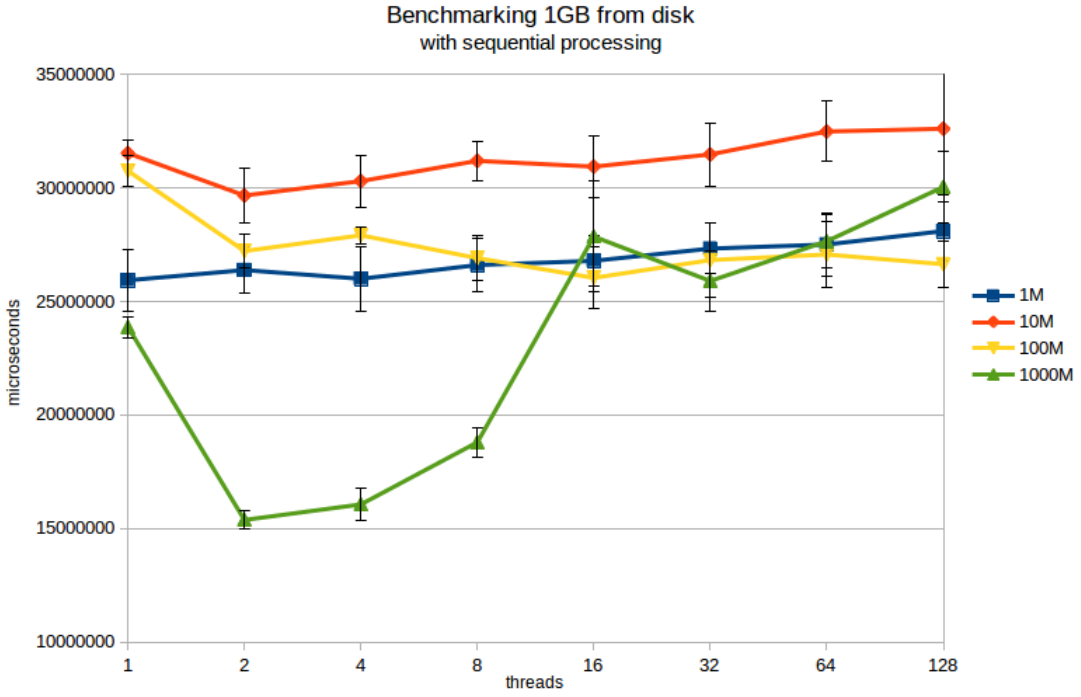


Figure 11: These are the results from running the benchmarking method with 1GB data from a file stream, with varying amounts of available memory, from 1 megabyte to 1000 megabyte. Processing of the data was done sequentially. The RSD resembles the RSD from the disk-benchmarking and lies from 1% to 8%.

The practical timings in Table 3 shows that copying 1GB of data takes approximately 0.2 seconds, this will be done twice in the benchmarking operations. 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 parallel processing, the buffer-size does not make much of a difference. Figure 12 clearly has very identical results for all four sizes of memory. With sequential processing, however, using the most memory clearly makes the library faster. Larger buffers results in more processing per iteration, 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 3.5 times faster than the sequential processing. There is some inherent overhead from multithreading, which is why we can not 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 and multiple threads, the locking becomes more expensive. With 1 megabyte 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, but it might have to do with how the read operations from the input stream aligns with the buffer sizes. However, with plenty of memory, the sequential system performs best at 8 threads.

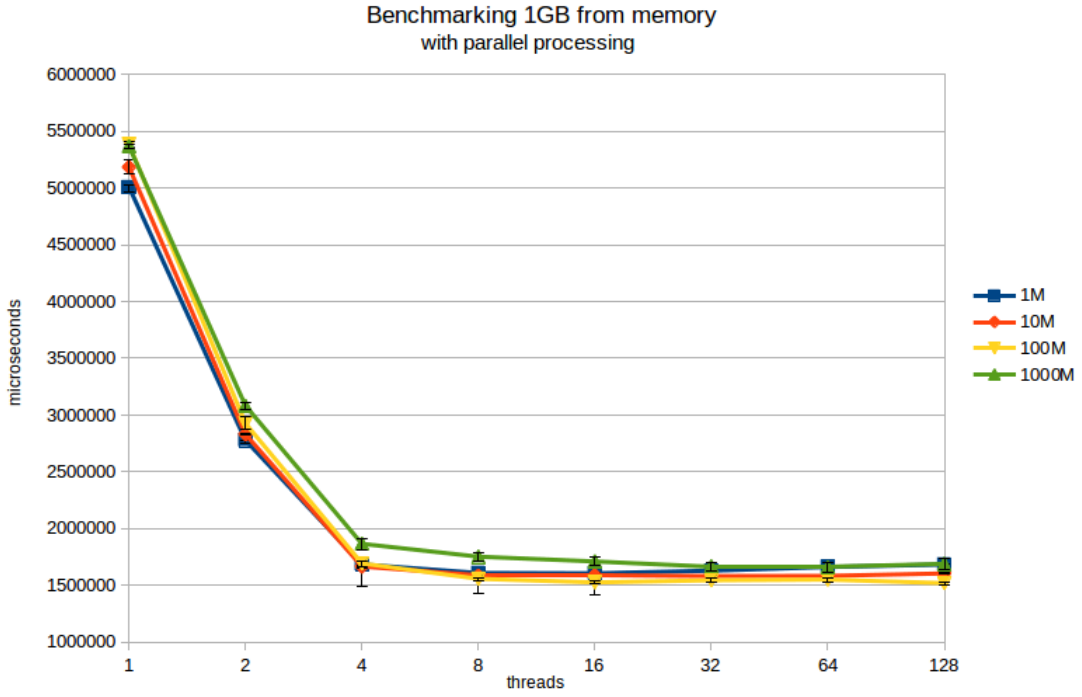


Figure 12: These are the results from running the benchmarking method with 1GB data from main memory, with varying amounts of available memory, from 1 megabyte to 1000 megabyte. Processing of the data was done in parallel. The RSD resembles the RSD from the memory-benchmarking and lies from 1% to 2%.

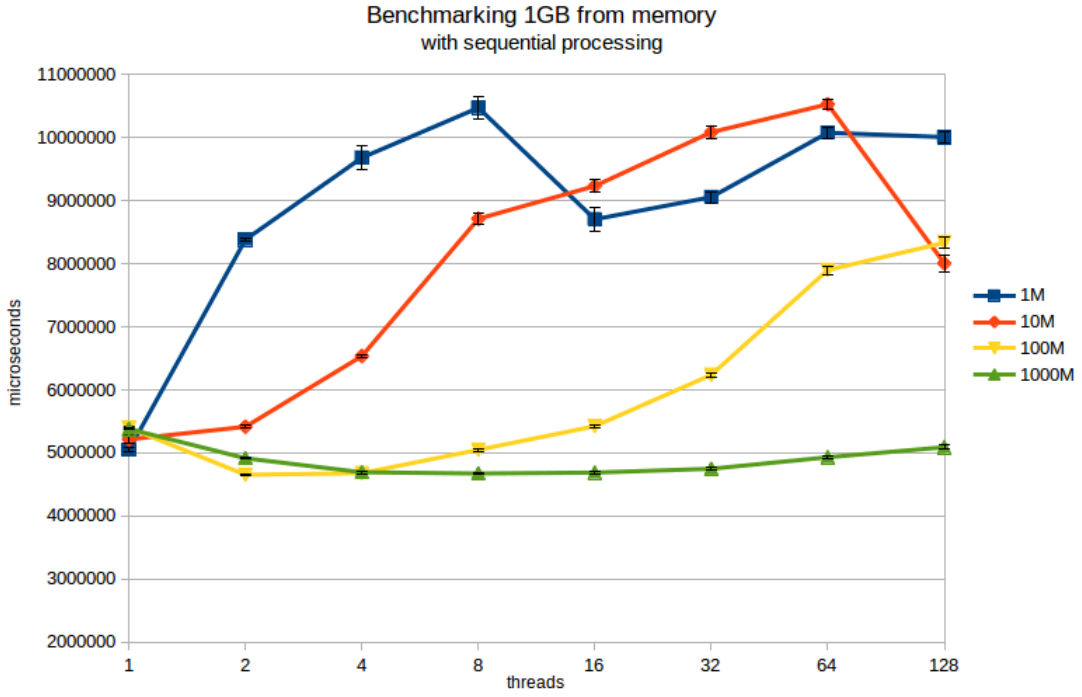


Figure 13: These are the results from running the benchmarking method with 1GB data from main memory, with varying amounts of available memory, from 1 megabyte to 1000 megabyte. Processing of the data was done sequentially. The RSD resembles the RSD from the memory-benchmarking and lies from 1% to 2%.

5 Conclusion

The *nbuf* library 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 library 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* library has been done with a benchmarking 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 library to performing I/O. This is a good thing, because it lets us see directly, how the bandwidth of the I/O resources affects the performance of the library.

When working with a slow stream, the library benefits from having a large amount of memory. Further, having more threads than there are hardware threads for, only reduces the efficiency, due to *thread-thrashing*.

When using a slow hard disk drive, the sequential operation takes approximately 18.2 seconds, where the optimal configuration of the library only takes approximately 15.1 seconds. The practical limit is approximately 13.4 seconds. The library has shaved off approximately 3.1 seconds, by performing the calculations during loading. Here, the execution has been *hidden*. This is independent of whether we perform sequential or parallel execution.

When working with a fast stream, the library 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 does not make a difference. In both cases, the optimal amount of threads matches the number of hardware threads.

The benchmarking task can be done sequentially in approximately 5.2 seconds, and the library performs the benchmarking task in parallel approximately 1.6 seconds. This is a factor 3.5 speed-up, which aligns with the 4 hardware cores on the system at hand. When performing the benchmarking sequentially, the library manages to finish the task in 4.8 seconds, effectively hiding all I/O latency.

To sum up, with both 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* library has resulted in a C++11 library that can be included by programmers 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 library 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 library 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 library solves the problem.

The library does not yet support the queue mechanics, which is necessary for computational tasks which require some degree of 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* library is already available for interested people to work with through GitHub[15], and I expect to submit it to Boost[16] if the code reaches maturity levels beyond this master's thesis. The current implementation has been tagged as *v0.1*.

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

[16]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* library, and some interesting topics to explore.

6.1 Missing Features

The library is missing support for sequential processing and in-order output. It can be implemented in the current library, as described previously.

6.2 Further Experimentation

While the experimentation clearly shows the effectiveness of the *nbuf* library, 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 library reacts in different environment can easily be an entire 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 library behaves.
- **Additional 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 study 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 iteration. This will allow the threads to start processing quicker. If plenty of computational power is available, then this will result in a faster overall runtime.

6.4 Multiple Input Streams

If identical data is generated from multiple sources, the library could be used as a method for unifying the information into a single stream, while gathering statistics. This would require that the library accepted multiple input streams, and some sort of balancing scheme, to ensure that all streams would be read from.

6.5 Multiple Output Streams

If a data steam consists of multiple different information, that should be output in different places, then it could be useful to be able to specify multiple output streams, along with a scheme for how to distribute the data during the writing-state.

6.6 Error Handling

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

6.7 Logging

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 programmer has to insert debugging statements directly into the library.

6.8 Use C++11/C++15 Features

The current library does not use many of the newer features from C++11. Using smart pointers and modern allocators would help deallocating memory, when it is no longer needed. This will decrease the risk of memory leaks.

References

[1] J. Bacon. *Concurrent systems : operating systems, database and distributed systems : an integrated approach*. International computer science series. Addison-Wesley, Harlow (England), Reading (Mass.), 1998.

[2] L. Columbus. 56the next three years. `http://www.forbes.com/sites/louiscolumbus/2015/03/22/56-of-enterprises-will-increase-their-investment-in-big-data-over-the-next-three-year` Accessed: 2015-10-17.

[3] CSC. Big data universe beginning to explode. `http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode`. Accessed: 2015-10-17.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[5] G. Developers. Protocol buffers project page. `https://developers.google.com/protocol-buffers/`. Accessed: 2015-09-22.

[6] E. W. Dijkstra. The origin of concurrent programming. chapter Hierarchical Ordering of Sequential Processes, pages 198–227. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[7] D. Laney. 3d data management: Controlling data volume, velocity and variety. `http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf`. Accessed: 2015-09-30.

[8] W. Magazine. Github's top coding languages show open source has won. `http://www.wired.com/2015/08/github-data-shows-changing-software-landscape/`. Accessed: 2015-10-17.

[9] B. Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[10] M. van Rijmenam. Why the 3v's are not sufficient to describe big data. `https://datafloq.com/read/3vs-sufficient-describe-big-data/166`. Accessed: 2015-09-30.

[11] D. Wilson. Triple buffering: Why we love it. `http://www.anandtech.com/show/2794`. Accessed: 2015-09-25.

[12] J. Zawinski. Law of software envelopment. `https://www.jwz.org/hacks/`. Accessed: 2015-09-26.