A platform independent approach to multi-threaded encoding with Random Linear Network Coding

Master Thesis
Software
Aalborg University
29th May 2017

AALBORG UNIVERSITY
STUDENT REPORT

Summary

Purpose: The purpose of this master thesis is to investigate the possibility of creating a multi-thread Random Linear Network Coding Encoder which is ignorant to specific system resources, such as size of main memory and cache, and the amount of available CPU cores. This is done with the goal of significantly decreasing the latency introduced by encoding when using Random Linear Network Coding.

Method: We present three possible encoding schemes which can utilise parallel computing and multi-threaded programming, and thereby have the potential to decrease encoding latency. We select one of these schemes for implementation. With the implemented encoder we conduct an empirical study and compare the results with the latency of a state-of-the-art single threaded Random Linear Network Coding encoder.

Results: We concluded that the implemented scheme does not give a decrease to encoding latency, and that this is the effect of ignoring cache size. Additionally, we also concluded that it is unlikely, that an efficient parallel multi-threaded encoding scheme can be designed, without taking system resources, such as cache size, into account.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Software
Selma Lagerlöfs Vej 300
9220 Aalborg East
https://www.cs.aau.dk

Title:

A platform independent approach to multi-threaded encoding with Random Linear Network Coding

Project:

Master Thesis

Project period:

February 2018 - June 2018

Project group:

deis1020f18

Members:

Lars Nielsen

Supervisor:

René Rydhof Hansen - Aalborg University, department of Computer Science Group for Distributed and Embedded Systems

Daniel E. Lucani - Aarhus University, Department of Engineering

Prints: 1
Pages: 39
Ended: 06-06-2018

Synopsis:

Purpose: The purpose of this master thesis is to investigate the possibility of creating a multi-thread Random Linear Network Coding Encoder which is ignorant to specific system resources, such as size of main memory and cache, and the amount of available CPU cores. This is done with the goal of significantly decreasing the latency introduced by encoding when using Random Linear Network Coding.

Method: We present three possible encoding schemes which can utilise parallel computing and multi-threaded programming, and thereby have the potential to decrease encoding latency. We select one of these schemes for implementation. With the implemented encoder we conduct an empirical study and compare the results with the latency of a state-of-the-art single threaded Random Linear Network Coding encoder.

Results: We concluded that the implemented scheme does not give a decrease to encoding latency, and that this is the effect of ignoring cache size. Additionally, we also concluded that it is unlikely, that an efficient parallel multi-threaded encoding scheme can be designed, without taking system resources, such as cache size, into account.

Preface

Special Thanks and Recognition

I would like to thank my supervisors Associate Professors Daniel E. Lucani and René Rydhof Hansen for their guidance and advice during my 9^{th} semester project and master thesis, and for keeping me on track.

I would like to thank PhD Jane Billestrup, my first supervisor at Aalborg University. Firstly for continuous guidance throughout both my bachelor- and master degree. But also for listening when things were difficult.

I would like to thank Chocolate Cloud ApS for providing the hardware used for experiments and accommodations for me to work.

I would like to thank my mother, father and younger sister for their support during my bachelor and master degree. They believed in me, even when I did not.

I would like to thank Jes Andersen for being my rubber duck, when I had problems with C++ code.

Lastly I would like to give a special thanks to Jens Christian Laursen for his invaluable help proofreading both this and my 9^{th} semester report, and for his unfaltering support throughout my masters degree.

Reading Guidance

This master thesis is aimed ad graduate students, academics, and professionals with and understanding of software benchmarks and requires a basic knowledge of Network Coding.

Code

For this project the software library Kodo [1] has been used for Random Linear Network Coding operations. The usage of Kodo requires a license which can be acquired here: `http://steinwurf.com/license.html`

The source code and results for this master thesis is available on github `https://github.com/looopTools/master-thesis-src` under the MIT Software license [2]. The source code is written in C++ version 14 and basic knowledge of this C++ version is needed to follow the source code.

Lars Nielsen

Table of Content

1 Introduction

In recent years the usage of cloud based systems has become an increasingly more integrated part of our everyday technology usage, for instance cloud based storage solution such as DropBox [3], Google Cloud [4], Microsoft One Drive [5], Apple iCloud [6] and others have found their way into both the consumer and enterprise markets. However, with cases such as *"the fappening"* [7] an increased demand for secured service have risen from the customer base. Based on this, Chocolate Cloud ApS [8] have developed the a distributed cloud storage solution called SkyFlok [9], which focus on high security and availability, which in both cases utilise Random Linear Network Coding (RLNC) [10]. However, the latency penalty of using RLNC for data encoding was unknown to Chocolate Cloud ApS, but they wanted a maximum latency limit of one second. In advance it was known that the configuration used for RLNC had an impact on latency and what elements of the configuration had an influence was also known, but the relationship between these elements was unknown, therefore did we create a way to investigate the impact of a configuration on latency and analysed relationship between configuration changes an latency as presented in Nielsen [11]. The study showed that for a data size of 512MB the latency limitation of one second was broken. Thus, Chocolate Cloud ApS would like to investigate a scheme which could reduce the latency of encoding, such that it decreases to below the one second limitation.

Chocolate Cloud ApS have two main reasons for investigating the potential of decreasing latency for RLNC encoding, both will be explained using Figure 1.1. The first reason is based on the satisfaction of clients, as stated above we know that there is a latency penalty using RLNC thus extending the time spent uploading or downloading files, and from the clients perspective the upload time in SkyFlok will seem longer than that of competing cloud storage solutions, even though the preprocessing of data using RLNC encoding is a key essential of SkyFlok. Therefore is it essential to decrease the latency of RLNC encoding such that the latency penalty becomes less evident for the clients. In [11] it is described how Chocolate Cloud ApS would like to decrease the management overhead when utilising RLNC, which is the second reason for investigating a way to decrease latency. In Figure 1.1, we see meta data stored in SkyFlok during the upload, the meta data stored contains information such as *file name*, *location of file fragments*, *RLNC generation information*, and more. The major issue with this is RLNC generation information, because for each generation we need to store the *symbol-* and *generation size*, and what symbols belong to what generation, see Section 2.1 for a description of RLNC. The management of this information can be dramatically lessened by using a single generation, this will however result in a large encoding latency as shown in [11]. Therefore decreasing encoding latency will be a beneficial tool for lessening the management issue.

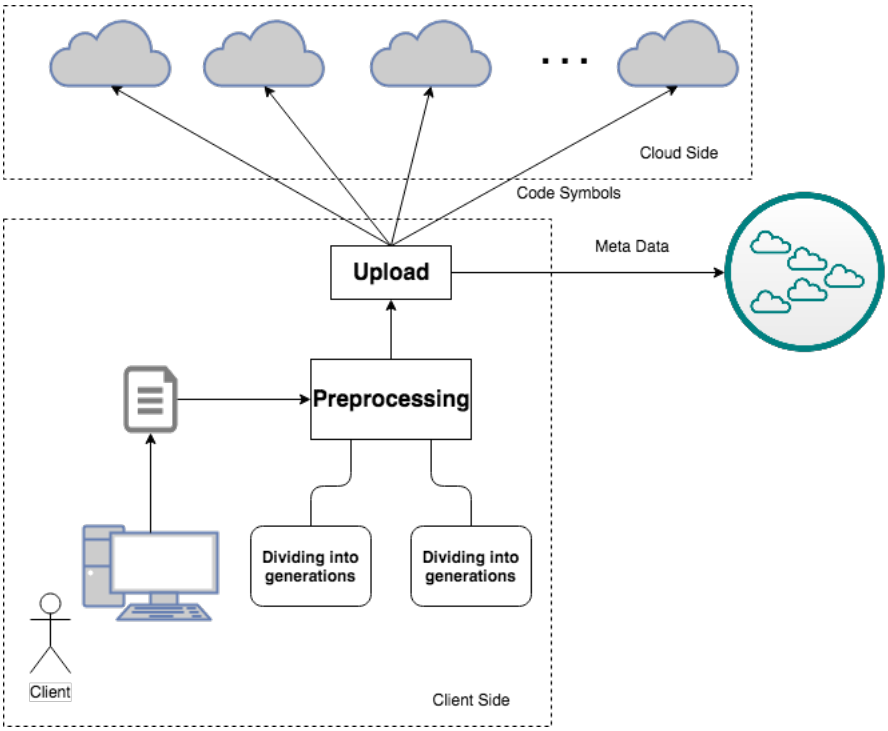

Figure 1.1: Data upload in the multi cloud storage system by Chocolate Cloud ApS

The problem is then; *"How can we decrease encoding latency?"* A possible approach to a solution is to utilise the multi-threaded capabilities of modern Central Process Units (CPUs) [12, p. 180-181] and distribute the computations of RLNC encoding between the cores of the CPUs. However, for Chocolate Cloud ApS this raises a concern, because it is unlikely that clients machines provide a uniform set of capabilities, i.e amount of main memory, size of cache (L1, L2, and L3), and amount of CPU cores. Therefore, Chocolate Cloud ApS would like such a solution to be platform independent and not be designed around a system specific configuration. Thus, a multi-thread solution must be tested with multiple amount of threads to ensure that the latency will be below the limitation of one second.

Thus, the purpose for the project can be formulated as follows; *We seek to design, implement, and through experiments investigate if it is possible to create a platform independent multi-thread RLNC encoder which decreases the latency of encoding, such that a latency limitation of one second can be achieved for larger generation size.*

1.1 Related Work

Research already conducted in the field of using parallelism / multi-threading have focused either on RLNC decoding alone or the utilisation of a specific system resource. Examples of this is the usage of General Purpose Graphical Processing Units as presented by Choi et al. [13], who shows an approach utilising the extreme parallel capabilities of GPUs for decoding data using RLNC. Another example is presented by Wunderlich et al. [14], this approach utilise a direct acycled graph to determine which part of data can be parsed to a thread for computation and which is currently either blocking or being blocked. The

data in this approach is essentially divided into sub-matrices which can fit into L1 cache. A thing both approaches have in common is also that they focus on throughput ($\frac{MB}{s}$), rather than latency itself.

Additionally, Morten V. Pedersen Chief Technical Officer of Steinwurf ApS [15] have played with a multi-threaded implementation of RLNC encoding and decoding. This work resulted in a working solution, however it was slower than the single threaded implementation provided with the RLNC software library Kodo [1].

Preliminaries 2

In this chapter we present the preliminary knowledge base, needed to design a parallel RLNC encoder. The topics discussed are Network Coding (NC)/RLNC, multi-threading, and thread-pooling. This includes the advantages and disadvantages of using multi-threading for solving latency issues in general.

2.1 Network Coding

Ahlswede et al. [16] introduce NC, as a technique which could be applied in networks for outperforming routing, by allowing coding of data at the source and in the network nodes. Ho et al. [10] showed that performing random linear combinations of incoming packets in the intermediate nodes of a network, achieves multicast capacity asymptotically. This process is called random linear network coding (RLNC). RLNC has since been adapted for usage in data storage by applications such as SkyFlok. In this section we describe how operations on data are conducted, from the source, in network nodes, and at the destination. We will also give a brief introduction to Finite Field (GF), as they provide the key arithmetic operations for NC and RLNC. Finally we will give an introduction of the RLNC C++ library Kodo.

NC can be divide into the following step *Encoding*, *Decoding*, and *Recoding*. In Figure 2.1, we give an illustration of where the three steps is located in a network utilising NC.

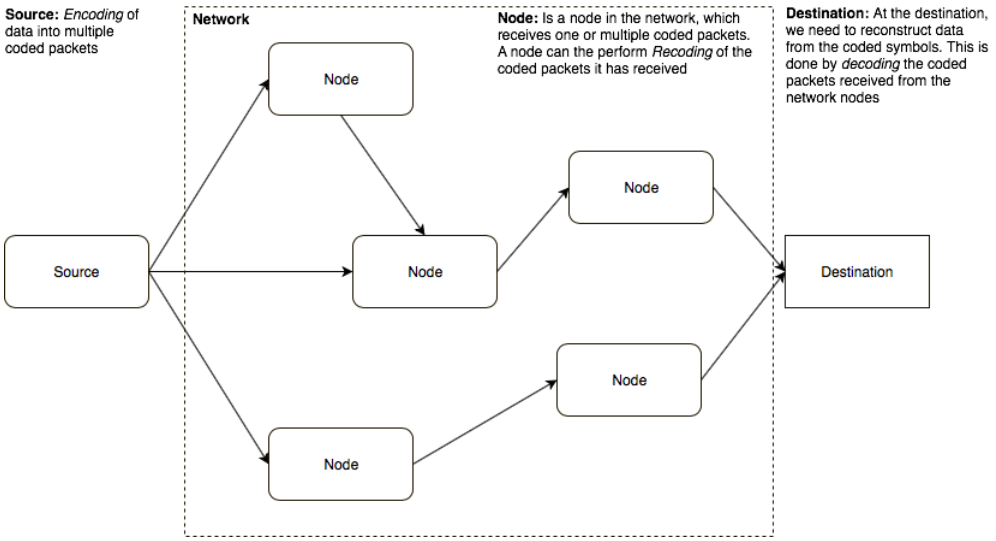

Figure 2.1: Illustration of where in a network encoding, recoding, and decoding takes places

At the source we wish to transfer data to the destination and part of this process is to produce *coded packets* using RLNC encoding. To do this, we first divide the data into a set of blocs called *generations*, these blocks are not required to have the same size. Wen the divide each generation further into smaller data chunks called symbols, where each symbol with in a generation have the same size, called *symbol size (k)*. The number of symbols with in a generation is referred to as the *generation size (g)*, and $g \cdot k$ is the total data size of the generation, also called the block size. For each generation we create a matrix, called the *symbol matrix (𝕊)*. $\mathbb{S}$ is created by viewing each symbol in the generation as a row in $\mathbb{S}$, as illustrated in Equation (2.1)

$$\begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1k} \\ S_{21} & S_{22} & \cdots & S_{2k} \\ \vdots & \ddots & \dots & \vdots \\ S_{g1} & S_{g2} & \cdots & S_{gk} \end{bmatrix} \tag{2.1}$$

The next part of encoding is to generate a matrix, called the *coefficient matrix (ℂ)*. Each row in $\mathbb{C}$ consist of g coefficients, all drawn uniformly at random from the elements of a Finite Field (GF) on the form $GF(2^n)$, see section Section 2.1.1 for a brief introduction to finite fields, and are called a *coefficient vector (c)*. $\forall c \in \mathbb{C}$ we then construct a *coded symbol* by multiplying c with $\mathbb{S}$ as shown in Equation (2.2)

$$\mathbb{C}_i \cdot \mathbb{S} = \begin{bmatrix} C_{i,0} \cdot (S_{11} & S_{12} & \cdots & S_{1k}) \\ & + & & \\ C_{i,1} \cdot (S_{21} & S_{22} & \cdots & S_{2k}) \\ & + & & \\ \vdots & \ddots & \dots & \vdots \\ & + & & \\ C_{i,g} \cdot (S_{g1} & S_{g2} & \cdots & S_{gk}) \end{bmatrix} = \mathbb{CS}_i \tag{2.2}$$

From the coded symbols, we construct a new matrix called the *coded symbol matrix* $\mathbb{CS}$. We then append the coefficient used to generate the coded symbol, such that we have a augmented matrix on the form $[\mathbb{CS}|\mathbb{C}]$, each row is now referred to as a *coded packet*. Though this concludes encoding itself, there is an additional step. This step is linear combination of n coded packets as illustrated in Figure 2.2. What happens is that from $[\mathbb{CS}|\mathbb{C}]$ we take n coded packets at random and combine them using the `XOR` ($\oplus$) operation, and we then transmit these combined coded packets along side the coded packets in the network. The advantage of this is best explained by example; We assume that we have two coded packets a and b, and we create the combined coded packet $a + b$. We then transmit all three packets through the network to the destination, but due to packet loss the destination never receives b. This not a problem, due to the properties of $\oplus$, it is possible for us to reconstruct b from a and $a + b$. For, if $a \oplus b = a + b$, then $a \oplus a + b = b$. This decrease the charge of having to retransmit a packet through the network.

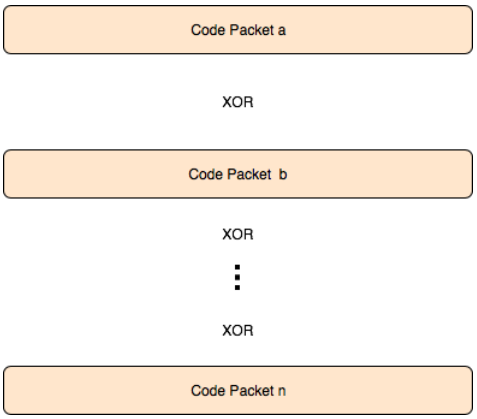

Figure 2.2: Linear combination of n packets using the $\oplus$ operation

In each network node we receive a set of both coded- and combined coded packets. We then construct new combined coded packets from the set of received coded packets, following the same method as shown in Figure 2.2, this is *Recoding*. The node then transmit these newly combined packets through the network along side those it received, either from the source or another node in the network.

As both the combined coded- and coded packets arrive at the destination, the foundation of the decoding process takes place. Using the coded packets, we start reconstructing other coded packets from the combined coded packets, and we do this until we have g unique coded packets. We order the coded packets as an augmented matrix $[\mathbb{CS}|\mathbb{C}]$. Then, we decode $\mathbb{CS}$ by applying Gauss-Jordan reduction [17, p. 18] on $\mathbb{C}$, such that it is brought to its identify. We apply all row operations performed in $\mathbb{C}$ in $\mathbb{CS}$ as well. By doing so, will the row operations bring $\mathbb{CS}$ back to $\mathbb{S}$ and this concludes decoding. After decoding each generation, we order them in the correct sequence and thus, we have reconstructed the original data from the source at the destination. Next we will give a brief introduction to finite fields.

2.1.1 Finite Fields

Finite Field (GF), also known as Galois Field, is a class of numerical fields which are used in NC and RLNC for providing essential arithmetic operations and provided additional beneficial properties. Furthermore are the content of $\mathbb{C}$ drawn uniform randomly from the same Finite Field (GF). Here we briefly will introduce the arithmetic operations used by RLNC and the beneficial properties, for a in-depth description see [18] section 4.5 and 4.7, with 4.5 focusing on explaining the general principals of GFs on the form $GF(p)$, and 4.7 focuses on the GFs applied in NC and RLNC, which is on the form $GF(2^n)$.

A GF is a field which contains a finite number of elements and is closed under the field, meaning that if an arithmetic operation is perform on a GF will map to a GF in the same field. Thus if x is a number in $GF(2^8)$ and we add y, then the result will be in $GF(2^8)$. This property, has a beneficial side effect in terms of memory in a computer, because if we use $GF(2^8)$, we then know that we can store the result in a contain which has the size of a byte, i.e `uint8_t` from C++, meaning that we have a large control over memory usage.

The second property, is interesting in relation to computations, as GFs utilised polynomial arithmetic for computations, this includes addition and multiplication. Furthermore, for any $GF(p^q)$ it follows that the addition rule for the base field p is applied, when utilising addition in a larger field. For $Gf(2^n)$ this means the addition rule for $GF(2)$ is applied, which is an `XOR`($\oplus$) operation. This means that instead of having to implement multiple version of addition, is possible to only implement one. The case for multiplication is a bit more complex, as the multiplication of two polynomials does not necessary map to the same field domain, but that is a rule of GF and this is solved as described in Stallings [18, p]: If the multiplication of two polynomials of a GF exceeds the field, the resulting polynomial must be reduced by modulo with a irreducible polynomial from the GF and the remainder is kept. Thus will the multiplication of two polynomials stay within a GF.

We will now give a brief introduction to the RLNC C++ library Kodo.

2.1.2 Kodo

Kodo is a C++ software library designed for data encoding and decoding and it provides multiple codes for this process, amongst these are RLNC codes, for a full list of Kodo supported RLNC codes see [11, p.10-13]. Kodo was first introduced by Pedersen et al. [19] in 2011 as a research project under the department of Electronic Systems at Aalborg University and is currently maintained and further developed by Steinwurf ApS [15].

The encoder and decoder components provided by Kodo for RLNC operations all provided a similar interface for interaction, by expecting generation size (g) and symbol size (k), and data input, with extra parameters for special codes. This makes Kodo easy to use when switching RLNC code. Furthermore Kodo provides memory shallow encoders and decoders, this means that instead of copying the data when initialising an encoder or decoder is a reference passed, through which data is used. For multi-threading this is ideal as it minimise the amount om memory need by the individual encoder.

Encoding in using Kodo encoders, is executed by invoking the `write_symbol` method, which takes a `uint8_t` pointer pointer as destination ad fill the pointer with a coded symbol. When `write_symbol`, the Kodo encoder will generate the coefficient vector for us and append it, to the destination.

Since the study presented in [11] was conducted, have 4 new major versions of Kodo been released, and Kodo version 11.0.0 is the newest major version. However, to ensure comparability with the study presented in former study, will we continue to use Kodo version 7.0.0.

This concludes the section on RLNC, we will follow this with a description of multi-threading.

2.2 Multi-threading

In this section we present the principles of multi-threading. We include a explanation of the difference between *concurrent* and *parallel* execution, the disadvantage of multi-threading, and the usage thread-pools. We base most of this chapter on Stallings [12, p. 177 - 217]

A program is executed within a process, where a process can be seen as an environment which contains a program as machine instructions and associated data, as illustrated in Figure 2.3. A process allows a program to be executed in a protected environment, such that other programs is unable to change neither machine instructions nor the data of the program. However, a process only allows for sequential execution of machine instructions, to enable concurrent execution of machine instructions we use *threads*.

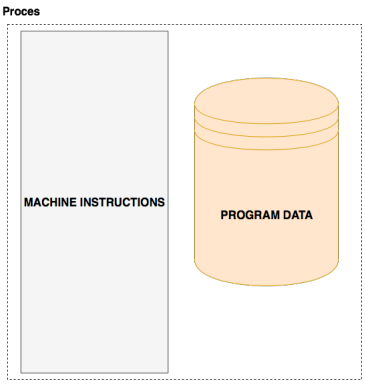

Figure 2.3: A process and its content, all program machine instruction and associated data

The purpose of threads is to allow concurrent execution of tasks within a process. Thus, a thread is an environment which contains an executable sub-part of a process machine instructions. As illustrated in Figure 2.4.

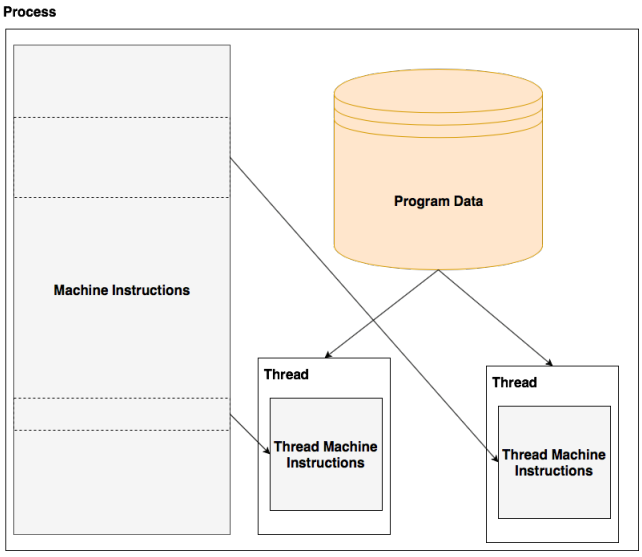

Figure 2.4: A process containing multiple threads, and distribution of machine instructions to the threads

However, threads can be executed in two different ways, *concurrent* and *parallel*. Concurrent thread execution, allows for the execution of multiple threads on a single CPU by interweaving the execution of threads, Figure 2.5 illustrates concurrent thread execution. What we see is that first a part of the machine instructions of T_1 is executed, then a part of T_2 This continues until both threads have finished execution. In Figure 2.5,

the total execution time for T_1 and T_2 is the same, but if we assume T_2 execution time would only match two boxes in the figure, then it would complete execution quickly, allowing T_1 to complete execution afterwards. This would allow following tasks to not wait for T_2 when T_1 completes its execution. This is how, multi-threading is handled in most single CPU core systems.

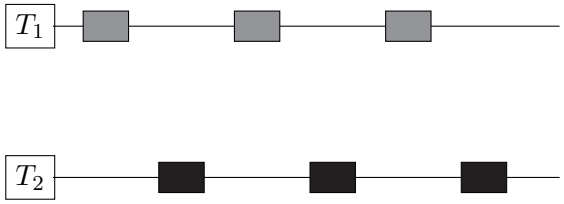

Figure 2.5: Example of concurrent multi-threaded execution tow thread. Coloured boxes indicate active thread execution, grey for T_1 and black for T_2.

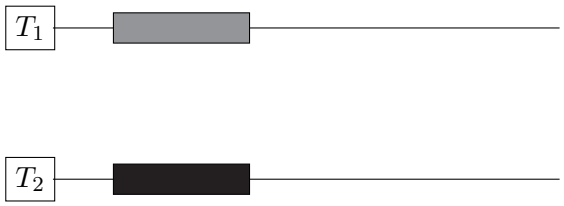

Figure 2.6: Example of Parallel multi-thread execution for two threads. Coloured boxes indicate thread execution, grey for T_1 and black for T_2.

To take advantage of the more modern multi core CPU systems, parallel execution is used. Here T_1 and T_2 are executed at the same time as illustrated in Figure 2.6. But they are executed on different CPU cores and does not interfere with each others execution. However, if we make the assumption that we have a CPU with n cores, we are then able to execute n threads in parallel and we cannot exceed this. Does this mean that we are only able to have n threads in a multi-threading system? No, because each CPU core allows for x threads to run concurrent, and though they will not run true parallel, the total execution time will be shortened and we end out with $\#threads = n \cdot x$. Note that while we are able to create more threads than can be executed concurrent on a single core, they will just be in a waiting state until they can be executed.

Thus in theory, we can assume that if we have a single process running on a single CPU core and divide it between multiple cores, then the speed gain should be as shown in Equation (2.3) [12, p. 192], where $(1-f)$ is the part of the code which cannot be executed in parallel, with the fraction f which can be run in parallel infinitely.

$$speedup = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on N parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}} \tag{2.3}$$

This means, that if we assume, that 10% ($f = 0.9$) of the program needs to be run sequentially and that we have an eight core CPU, the performance gain will , in theory, be $\frac{1}{(1-0.9)\frac{0.9}{8}} = 4.7x$.

However, though there is a performance gain using multi-threading, it also introduces a set of issues. Firstly, recall that data is protected between different process. This, however, is not the case for threads and this alone creates two major issues; *race conditions* and *deadlocks*. Race conditions is a situation where two or more threads work on the same piece of data, and depends on the order of which the data is updated. But, as we have no control over the order in which threads are executed, the threads can reach a situation where data is changed out of order and resulting in program bugs. Race conditions can be avoided by locking data such that only one thread can access it at a time and when done it releases the data. However, this creates the possibility of deadlocks, a situation where, for instance, T_1 has a lock on data resource d_1 and will release the lock, when it can access data resource d_2, and T_2 has a lock on d_2 and will release the lock, when it has access to d_1. Thus, the two threads a blocking each other and will never continue executing, resulting in a deadlock. As with race conditions, deadlock can be avoided using locks, by letting a thread lock all the data it needs, thus putting all other threads on hold in the period it has lock, and then release the lock when done with data, letting the next thread do the same. However, this does create bottlenecks in thread execution and result in further decrease to the theoretical performance gain.

The last issue is the time cost of spawning threads. The time cost of spawning a thread depends on the CPU, which can be influenced by environmental conditions, and the thread system implementation used. Therefore, spawning new threads for each task is a costly strategy. To solve this, we utilise the ability to reuse a thread, by giving it new instructions to process and avoid the cost of spawning a new thread. The next step in this solution is to create a pool of n threads, which are spawned at the beginning of program execution, then when a thread is need we request it from the pool, and when the thread completes execution, we return it to the pool. This final approach is known as a thread-pool.

With the preliminaries covered in this chapter, will we continue with the design of the parallel encoder.

Parallel Encoder Design 3

In this chapter we present different parallel encoding schemes and select which one to move forward with, and based on this we design a parallel RLNC encoder.

In Chapter 1 we state that Chocolate Cloud ApS seeks a solution which can be used on any platform. This presents a set of limitations for designing a parallel encoder. Firstly, we cannot use system specific libraries, as this will limit the encoder designed to a single system. Secondly, the designed encoder cannot depend om specific hardware configurations such as cache size or size of main memory. Finally, we need to utilize an abstraction layer over the thread system used in the host system, which will allow usage of threads on any system. With this in mind will we look at possible parallel encoding schemes.

3.1 Parallel Encoding Schemes

Before designing the encoder is it needed to investigate the potential schemes which can be utilised to parallelize the process of encoding with RLNC. With the encoding process being $\mathbb{C} \times \mathbb{S}$ there exists at least three schemes of dividing encoding into multiple tasks. The first scheme is to divide a symbol into multiple chunks or sub-symbols, where for each sub-symbol a thread is spawned where a coefficient from the coefficient vector is multiplied with the sub-symbols in each thread, as illustrated in Figure 3.1.

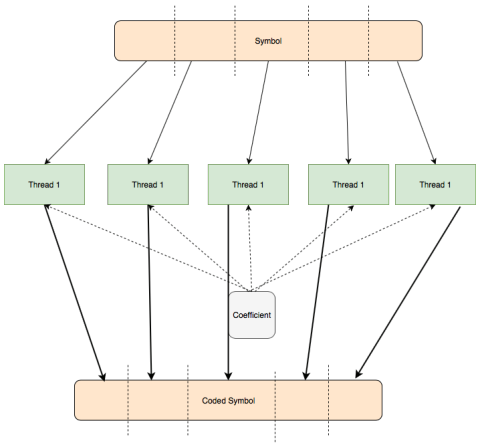

Figure 3.1: Dividing a symbol into multiple sub-symbols and multiplying each sub-symbol with a coefficient within multiple threads

This scheme will essentially reduce the symbol size (k), such that $k' = \frac{k}{m}$ where m is the number of sub-symbols created. Though this will decrease the symbol size, it will also increase the generation size (g) and from [11, 20] we know that decreasing k will result

in lower latency, but that increasing g will result in an higher latency. Therefore, if this scheme is chosen, the benefit of multi-threading will depend on the final choice of k and g.

The second scheme is to parallelize the addition of symbols during the matrix multiplication, as illustrated in Figure 3.2. This approach takes effect after a coefficient vector has been multiplied on $\mathbb{S}$, by distributing n symbols to m threads, and let each thread handle the addition of n symbols, Figure 3.2 illustrates this with $n = 2$ and $m = \frac{g}{n}$. Then when the result of each thread is returned, we divide n' partial coded symbol per thread, where we have m' threads. This is repeated until only the fully coded symbol is constructed. This is repeated for each coefficient vector $\in \mathbb{C}$.

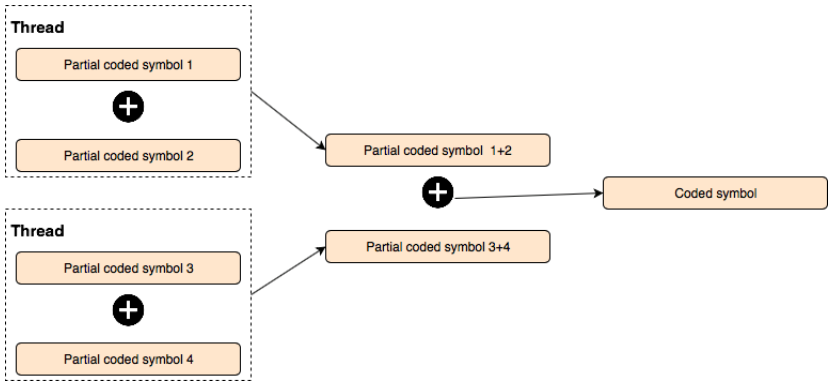

Figure 3.2: Multi-threaded approach to partial symbol addition for four symbols

This scheme will alter neither g nor k in any way but it should result in a speedup for encoding and thus decrease latency.

The final scheme is to split the multiplication of $\mathbb{C}$ and $\mathbb{S}$ by letting a thread handle the multiplication of n coefficient vectors from $\mathbb{C}$ with $\mathbb{S}$, as illustrated in Figure 3.3. This means that each thread produces n coded symbols.

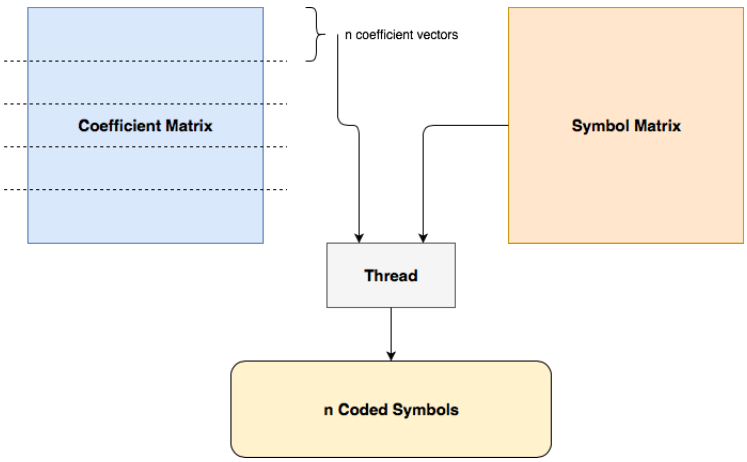

Figure 3.3: Multi-threadded approach to coefficient vector multiplication with $\mathbb{S}$, where n coefficient vectors are parsed to a thread to construct n coded symbols.

This scheme effectively splits the original problem of g coefficients into $\frac{g}{n}$ problems, thus reducing the amount of product operations performed per thread and thus should drastically decrease the latency for encoding, based on the Equation 5.4 [11, p. 18].

If we compare the three encoding schemes in relation to Equation 5.4 [11, p. 18], we see that decreasing g, whilst not altering k has the highest potential of decreasing latency. For this reason we chose to continue with the last presented encoding scheme and in the following we present a structure for an encoder based on this scheme.

3.2 Encoder Structure

With the parallel encoding scheme selected, we will outline the components of the encoder and how it facilitates RLNC encoding using the selected parallel scheme.

First we have to determine how to handle the threads used for encoding. In Section 2.2 we explain how spawning threads are costly in relation to latency and a way to circumvent this issue is the usage of a thread pool. For this reason we will be using a thread pool in the encoder and let it handle threads. We will be using the thread pool library *ThreadPool* by Jakob Progsch [21]. We are using this library because we have experience using it and therefore, know that it is a working library.

Next, we need to figure out how to implement the selected encoding scheme and for this we have chosen to implement two different approaches, and for the second approach we also present an smart approach. The first approach creates a thread pool with g threads, each thread then produce a single coded symbol. This implementation of the encoding scheme will likely result in a high level of context switching, which is disadvantageous for performance, we will reference this implementation as *simple encoder*. For this reason it is expected that a decrease in latency will be seen, but that the improvement will be limited. The second approach is implemented such that it is possible to state the number of threads *(t)* to use for encoding and number of coded symbols produced per thread is $\frac{g+r}{t}$, we will reference this implementation as *complex encoder*. This approach will have a smaller amount of context switching and should in the ideal world result in latency decrease of $\frac{\text{single threaded encoding latency}}{t}$. The smart approach is to utilise a language feature of C++ called `std::thread::hardware_concurrency` [22], which allows for automatic reading of how many concurrent threads a system supports, we will reference this implementation as *smart encoder*. However, [22] state, that this language feature is only available on supported systems. We have tested the support of `std::thread::hardware_concurrency` on the experiment machine and the two machines used for developing the experiment source code with the coded presented in Listing 3.1 and the result of `std::thread::hardware_concurrency()` for all three machines is 8.

```
1  #include <thread>
2  #include <iostream>
3  #include <string>
4  int main() {
5      std::string result = "";
6
7      if (std::thread::hardware_concurrency() == 0) {
8          result = "Unsupported/Not computable";
9      }
10     else { result = "Supported"; }
```

```
11
12      std::cout << "Hardware concurrency is: " << result << std::endl;
13
14      return 0;
15  }
```

Listing 3.1: Simple program for identifying if the system support the usage of `std::thread::hardware_concurrency`

Furthermore, we have also tested machines with Windows 10, MacOS High Sierra and Linux[1] and have not found a system, not supporting the language feature amongst these and also the none-int??? version of this encoder can be used, when `std::thread::hardware_concurrency` is not supported. Therefore, we will still implement a solution using this language feature, though it may limit the amount of supported systems. It must also be noted that `std::thread::hardware_concurrency`, only gives the amount of concurrent threads and not parallel threads.

To produce coded symbols n Kodo Full Vector encoders will be used, where each encoder will be assigned to a thread and based on the number of coded symbols produced per thread, the method `write_payload` will be invoked and all the resulting payloads will be stored in a shared result vector. To ensure minimal memory usage for the encoders, we will use a version of the Kodo Full Vector encoder, known as a shallow Full Vector encoder which avoids deep copy of data to the encoder, by only working on a reference to the original data.

Additionally we need to handle the case where $(g + r)\ mod\ t \neq 0$, as this means that the number coded symbols created per thread in the second implemented approach cannot be divide equally between threads, and for this reason we will need to handle this case. For this experimental implementation we handled it by assigning the encoding of the remaining $(g + r)\ mod\ t$ coded symbols, to the last thread. This is not necessarily the most efficient approach with which a solution can be found, but for this initial implementation, this will be the solution used. Later it should be investigated, what the most efficient division of the remaining symbols is.

Finally we need a way to check if the encoding process is completed. The reason for this is, that unlike the single-thread RLNC encoder, we do not know when encoding has concluded unless we keep track of the process. We will do this with an unsigned integer counter of the type `uint32_t` [23]. We initialise the counter to `0U` and each time a thread has completed its run, we increment the counter with `1U`, using the compound operator `++`. We guard it with an atomic lock provided by C++'s *Atomic operations library* [24], which provide a atomic lock free version of the compound operation $++$ on integers.

Based on this design we have implemented three different encoders, a simple encoder which spawns a thread per coded symbol, a complex encoder where the number of threads used can be adjusted, and finally a smart encoder which utilise the capabilities of `std::thread::hardware_concurrency`. With these encoders we have conducted the experiments presented in the following chapter.

[1] Ubuntu 17.11, Fedora 26/27, and Debian 9

Experiment Setup 4

In this chapter we describe the experiment setup, the configurations which will be used, and present an experiment protocol. The intention of these experiments is meant to measure the encoding latency for the three parallel encoders and compare the recorded latency values, with that presented in [20] and [11] for a single threaded full vector encoder, with generation size 32 for data size 512MB. We compare with the result for generation size 32, because it was this configuration which broke the 1 second limitation defined by Chocolate Cloud ApS.

4.1 Experiment Setup

We have conducted the experiments with the same setup presented in [11]. This is done to create a comparison foundation, between the results presented in [11, 20] and those recorded for the parallel encoders. We have used a *HP ProDesk 490 G3 MT Business PC* [25], with an Intel® Core™i7-6700 which has a clock rate of *3.40GHz*, and *16GB* of main memory, divide into two blocks of *8GB*, running a clock rate of *2133HZ*. For operating system we are using Fedora 21. [26]

We have enabled Secure Shell (SSH) [27] access to the machine, enabling remote execution of experiments and supervision of experiments. We also utilise the Linux system command `nohup` [28], which makes it possible to ignore user hangups and log out. This allows us to terminate an SSH connection without terminating execution of the experiments.

Finally, we have created a setup of scripts which are used for execution of the experiments with different configurations. The scripts are structured with a ten second break period between each experiment, thereby allowing the system to return to an idle state.

4.2 Experiment Configuration

The experiments will be conducted for three data sizes *512MB*, *1GB*, and *2GB*. We know from Nielsen [11], that a single-thread approach reaches the one second latency limit generation size 32 for data size 512MB, and we also know that for the same data size with generation size 8 and 16 is below the latency limit. Therefore, these configurations will be used as control configurations, to confirm if there is a performance gain when switching to a multi-threaded encoding approach. Additionally, to investigate if it is possible to increase the generation size, whilst still keeping latency below the limit, we will include experiments configured with generation size 64, 128, and 256. Thus, the configurations for generation- and symbol size presented in Table 4.1 will be used

g	k for ds = 512MB	k for ds = 1GB	k for ds = 2GB
8	67108864	134217728	268435456
16	33554432	67108864	134217728
32	16777216	33554432	67108864
64	8388608	16777216	33554432
128	4194304	8388608	16777216
256	2097152	4194304	8388608

Table 4.1: Experiment configurations used for 512MB, 1GB, and 2GB. g = generation size, k = symbol size, and ds = data size.

Furthermore, for the complex encoder we include an additional configuration parameter representing the amount of threads used. The experiments for the complex encoder have been conducted with *1, 4, 8, 16* and *32* threads. Also, as presented in Nielsen et al. [20], each experiment will be executed *1000* times. Next we present the experiment protocol used to conduct the experiments.

4.3 Experiment Protocol

Here we describe the experiment protocol. We do this to ensure reproducibility of the results, but also to ensure that future experiments are conducted in the same manner as the first. For all encoder types the following steps apply:

Step zero: Ensure that the multi-threaded RLNC encoder you wish to conduct experiments for is implemented and that a benchmark have been created.

First step: Select the multi-threaded RLNC encoder for which the experiment will be conducted.

Second step: Confirm that the necessary configuration file for the experiment exists. Generate a configuration file if none is available. Configuration files can be generated using the `generate_config.py` script, included with the project source code.

Third step: If a hypothesis for the experiment does not exists, formulate one and append it to the list of hypothesis.

Fourth step: If a script to execute the experiment does not exists, create it and remember to utilise the system command `sleep`. Additionally, if a result folder for the experiment does not exists in the directory `./results`, create one.

Fifth step: Reboot system before experiment execution, to ensure a clean system.

Sixth step: Start the experiment using the `nohup` command and the experiment script.

Seventh step: Compare results to hypothesis, to either confirm or debunk the hypothesis.

Eighth step: Compare results with that of other multi-threaded encoders.

Ninth step: Compare the results with those presented for the Full Vector algorithm presented in Nielsen et al. [20].

With the experiment setup and protocol defined, we can define hypothesis for the experiments, which we present in the following chapter.

Hypothesis 5

In this chapter we present the hypothesis which will be utilise during experimtation. The hypothesis will focus on the changes in latency based on generation size configuration and with regards to the latency record for data size 512MB for generation size 8, 16, and 32 presented in [11, p.21].

5.1 Change in Latency with Regards to Increases in Generation Size

In [11] we present Equation (5.1) as the base calculation for RLNC encoding latency, which can be used to calculate the effect of changing either generation- or symbol size.

$$\text{latency} = \frac{\alpha \cdot \Sigma_{i=1}^{g}(k(g+r)) + \beta((k \cdot (g-1))(g+r))}{\text{Hz}} \tag{5.1}$$

To accommodated for the usage of threads when calculating latency, we have to divide latency for a single-threaded encoder with the number of threads (t), but add the cost of spawning t threads (t_c), thus Equation (5.1) becomes Equation (5.2).

$$\text{latency} = \frac{\frac{\alpha \cdot \Sigma_{i=1}^{g}(k(g+r)) + \beta((k \cdot (g-1))(g+r))}{\text{Hz}}}{t} + (t \cdot t_c) \tag{5.2}$$

However, it is not possible to define a value for t_c, as it varies for all CPU units and are influenced by environmental factors, such as temperature [29] and for this reason is it decided to ignore the cost of spawning threads. This will make the expected latency better than what it will be in reality and will be kept in mind when results are analysed. But, in Chapter 3 we present a multi-threaded encoding scheme which changes generation size based on t, in the following way; $\frac{g}{t}$ and for this reason we have to adapt Equation (5.2) such that it is on the form shown in Equation (5.3)

$$\text{latency} = \frac{\alpha \cdot \Sigma_{i=1}^{g}(k(\frac{g}{t}+r)) + \beta((k \cdot (g-1))(\frac{g}{t}+r))}{Hz} + (t \cdot t_c) \tag{5.3}$$

And in [11], we show that the difference in latency between generation size, can be calculated as shown in Equation (5.4), if we ignore the clock rate of the CPU

$$h(g,k,g',k') = \frac{(2g'-1)\cdot k'\cdot(g'+r)}{(2g-1)\cdot k\cdot(g+r)} \tag{5.4}$$

We adapt this to take into account for t modifying generation size, such that the difference in latency ΔL can be calculated as presented in Equation (5.5), g' and k' is the increased generation size and the decreased symbol size.

$$\Delta L = h'(g,k,g',k',t) = \frac{(2g'-1)\cdot k'\cdot(\frac{g'}{t}+r)}{(2g-1)\cdot k\cdot(\frac{g}{t}+r)} \tag{5.5}$$

By adapting Equation (5.4) to Equation (5.5), we can calculate the latency for the complex- and smart encoder, where the calculation for eight threads, represent that of the smart encoder. In Table 5.1 is the expected ΔL values presented, we have filled the row generation size 8 with X's, as no experiments with smaller generation sizes has been conducted.

g/t	4	8	16	32
8	X	X	X	X
16	2.06x	2.06x	2.06x	2.06x
32	2.03x	2.03x	2.03x	2.03x
64	2.01x	2.01x	2.01x	2.01x
128	2x	2x	2x	2x
256	2x	2x	2x	2x

Table 5.1: Expected changes in latency (ΔL) for the complex- and smart encoder with regards to generation size (g) and threads (t) used

From Table 5.1 and Equation (5.5) we get that the number of threads used, will not change the expected ΔL when increasing generation size. Furthermore, if we compare the expected ΔL presented here with ΔL values found in [11, p.21], see Table 5.2, we see that they follow the same ΔL pattern. Thus, we can hypothesis, that using more threads will not change ΔL. Therefor, will the benefit only be a decreased latency.

Code / generation sizes	8 vs 16	16 vs 32
Full Vector	2.06	2.03
On-The-Fly	2.14	2.6

Table 5.2: Estimated x differences in latency between generation sizes for The Full Vector and On-The-Fly codes [11, p.21]

Next, we will state hypothesis for the changes to latency, when going from a single-threaded encoder to the multi-threaded encoder

5.2 Difference in Latency betwen Single-threaded and Multi-threaded Encoders

Next to calculate the gain of utilising a multi-threaded encoder, we utilise Equation (5.3). Again as we cannot provide an accurate $(t \cdot t_c)$ will it not be include in the calculation.

$$f(g,k,r,t) = \frac{\alpha \cdot \Sigma_{i=1}^{g}(k(g+r)) + \beta((k\cdot(g-1))(g+r))}{\alpha \cdot \Sigma_{i=1}^{g}(k(\frac{g}{t}+r)) + \beta((k\cdot(g-1))(\frac{g}{t}+r))} \tag{5.6}$$

Based on Equation (5.6) can it be hypothesised that t is equal to $n\times$ decrease in latency which should be observed. However, this is not correct, because if $g < t$ will the complex- and smart encoder not utilise all the spawned threads, as there will be fewer symbols to encoded than t, leaving threads without work. We therefore, use X to mark where no further performance improvement is expected. For the generation size 128 and 256 we should still see a ΔL which follows Equation (5.6) as the encoders can efficiently divide the multiplication of coefficient vector between threads. The expected values are presented in Table 5.3

g/t	**4**	**8**	**16**	**32**
8	4x	8x	X	X
16	4x	8x	16x	X
32	4x	8x	16x	32x
64	4x	8x	16x	32x
128	4x	8x	16x	32x
256	4x	8x	16x	32x

Table 5.3: ΔL between single-threaded- and multi-threaded encoders with regards to genration size (g) and amount of threads (t) used

However, there is an issue with the presented expected values for ΔL. Above we noted how, if $g < t$, then we wouldn't be able to fully utilise the spawned threads, which will result in something inconvenient. Which is; Though $\frac{n}{m}$ threads can execute multiplication of coefficient vectors with $\mathbb{S}$, will a number threads do nothing. The problem is that a thread pool keeps the thread in a "busy waiting"-state, meaning that a thread will check if a task is available for it to work. This results in a context switch which has a negative effect on latency and thus, we are unable to accurately predict the latency for these experiment cases. But we must take it under consideration when analysing the results.

With the experiment procedure defined and hypothesis formulated, we will conducted the experiments and analyse the results.

Results 6

Here we presented the results for the experiments we have conducted, we evaluate the result, and evaluate our approach as part of this process.

6.1 Results for Complex, Smart, and Simple Encoder

Here we present the results and analysis of the empirical study conducted for data size 512MB, 1GB, and 2GB with the complex-, smart-, and simple encoder.

In Table 6.1 we present the average observed ΔL values for experiments conduct for the complex encoder with 512MB, for all number of threads (t) and generation size (g) configurations. By comparing the *latency values* for the complex encoder run with t being either 4, 8, 16, or 32, with the single-threaded encoder, as presented in Table 6.2. We observer something highly unexpected.

# of Threads / Generation Size		**8**	**16**	**32**	**64**	**128**	**256**
4	Latency	700.02	1036.09	1796.62	3312.44	5780.05	7908.37
	ΔL	X	1.48	1.73	1.84	1.74	1.37
8	Latency	831.07	1153.76	1896.27	3584.45	6825.26	11660.70
	ΔL	X	1.39	1.64	1.89	1.9	1.7
16	Latency	432.17	1329.55	2114.77	3844.64	7304.92	13626.84
	ΔL	X	3.07	1.59	1.82	1.9	1.86
32	Latency	901.79	460.19	2215.72	3909.08	7417.16	14308.14
	ΔL	X	0.51	4.81	1.76	1.89	1.93

Table 6.1: The average results for complex encoder in milliseconds for data size 512MB with 4, 8, 16, and 32 threads, for generations size 8 to 32, with the differences (Δ) between previous and current generation size.

Generation Size	8	16	32
Latency	534.29	1036.49	1972.12

Table 6.2: Latency for Single-threaded Encoder for Generation Size 8, 16, and 32 for data size 512MB in milliseconds

We observer that the complex encoder, only in certain cases outperform the single-threaded encoder. These case are g = 8 with t = 16 and g = 16 with t = 32, for the later the

improvement is $\sim 2.25x$. Compared to the other ΔL values, we deem ΔL an extreme data point. For this reason we decided to conduct the experiments again to verify our findings, a part of this process involved validating the configuration files for the experiment. We found no fault in the configuration files and we where able to reproduce similar results to those presented in Table 6.1. Therefore, we cannot disregard this ΔL value. We also see that for $g = 32$ and $t = 4 \vee t = 8$, we see small performance gain of $1.09x$ and 1.04 respectively. Such a minimal gain can be attributed to system or environmental interference and we therefore will not conclude that for these configurations are the complex encoder better. Additionally, we observer that for $g = 16$ that for all configurations but one, $g = 16 \wedge t = 32$, is the latency values above the 1 second limit. Thereby, we can conclude that the complex encoder is not a general purpose solution for the latency issue, and that it will not make sense to further investigate the latency values for the complex encoder with data size 1GB and 2GB.

Another interesting observation, is that if we recall Table 5.1, we expected a $\sim 2x$ ΔL when increasing generation size, as was also expected in [11, p. 21]. But, like observed in [11, p. 25 - 28] we do not see $\sim 2x$ ΔL when increasing generation size. With the exception of the difference between generation size 16 and 32, we observer $\Delta L < 2x$, with ΔL for $g' = 32 \wedge g = 8$ for $t = 2$ being 0.51. This indicates that the single-threaded- and complex encoder does behave alike in terms of latency.

Based on what we have observer, we would like to investigate further, why the performance of the complex encoder is so poor. If we follow 1) the theory of multi-threading, 2) the fact that Kodo utilise Single Instruction Multiple Data (SIMD), and 3) we know that for some relationships of t and g, will some threads not be utilised properly. Then, we can state three hypothesis as too why the complex encoder performance poorly. Starting with utilisation of threads, we stated that if $g < t$ then some threads would not be utilised and they would enter a *"bussy wait"*-state, and when they request a context switch to check if a task is available, this will result in increased latency. However, though this might be the case for $g = 8 \wedge g = 16$, it should not be the case for $g \geq 32$ and therefore, is it not the likely cause for the poor performance of the complex encoder.

The second hypothesis is SIMD, in [11, p. 26] we state that SIMD might be a factor for the lower than expected ΔL values observed, but in a multi-threaded scenario the benefits of SIMD can be turned into something harmful. The reason for this, is that SIMD utilise dedicated registries for SIMD-operations [30] and depending on the CPU will there either be SIMD-registers available per core or a shared between all cores, with the later being the most common. This means that if each core has its own set of SIMD-registers, then if $t \geq$ number of cores, then when one thread is using the cores SIMD-register, then other threads one the core will not be able to continue execution until SIMD-registers are free. This becomes an even more harmful problem if the SIMD-registers are shared between all cores. This issue is fairly easy to investigate, as the part of Kodo, which utilise SIMD can be compiled without support for SIMD and will fallback to a non-SIMD approach.

The third hypothesis, is in regards to multi-threading and how memory swapping is done when two threads switching state. Such that one goes from running to waiting and the other goes from waiting to running. When two threads switch state, the data from the one going to waiting, will also be swapped out of cache and possible main memory. Then the

data of the thread, now in the running state, will be read into main memory and cache, and preferably all in cache as this is a faster medium. However, what if *symbol size (k)* ≥ cache size? Then we will first swap the current running threads data out of cache and write it either to main memory or disk. Next the former waiting thread will now be running and when it will access data it will likely get a cache miss. This will lead to a read from main memory, and if we are unlucky this might lead to a page fault [12, p. 370] which will lead to a read from secondary storage, which is costly. We compared k for all g with the cache size of the experiment machine which is 32kB for the L1 cache, and we could conclude, that $\forall g\ k \geq$ cache size. This makes it a likely performance issue candidate and for this reason, we have decided to create an encoder which is cache size aware and compare it to the performance of single-threaded encoder.

Before investigating these issues further, will we also analyse, the results for the simple- and smart encoder with data size 512MB, to confirm that the latency issues seen with the complex encoder is present with the two other encoders, and we will start with the smart encoder.

In Table 6.3 we show the observed average latency and ΔL values for the smart encoder, where the $t = 8$. What we see is that the differences between the complex- and smart encoder are insignificant and can be attributed to system interference.

Generation Size	**8**	**16**	**32**	**64**	**128**	**256**
Latency	774.59	1083.56	1905.92	3592.45	6799.44	11629.59
ΔL	X	1.40	1.76	1.88	1.89	1.71

Table 6.3: The average results for smart encoder for generation size 8 to 256 in milliseconds, with the differences (Δ) between previous and current generation size.

This becomes even more evident if we plot the results for the two encoders side by side as shown in Figure 6.1, where it can bee seen that the data points for the two encoders are close to overlapping. This means that as with the complex encoder, can we state the smart encoder is not a general purpose solution to the latency problem. This is to be expected as the smart encoder, is a trivial modified version of the complex encoder. It also means that we have a continues reason to investigate the hypothesis presented above.

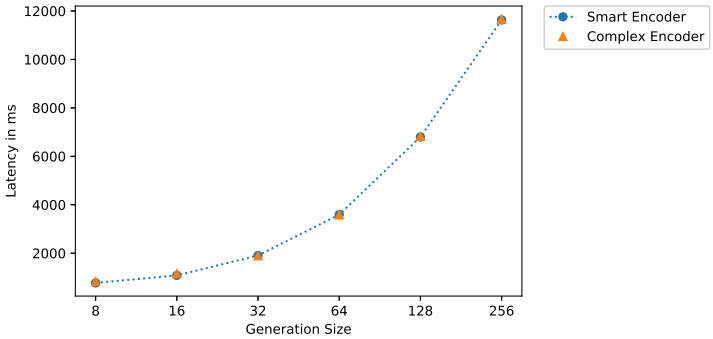

Figure 6.1: Overlay of the results for the complex encoder running with 8 threads and the smart encoder

Next, we analyse the result for the simple encoder. In Table 6.4 we show the recorded average latency for the simple encoder, and we observer something interesting. What we observer is that the ΔL between the simple encoder and the single-thread encoder is minuscule. Which follows the our expectation presented in Chapter 5. We also observer that only for $g = 8 \wedge g = 16$ is the latency below the 1 second limit, and for this reason, we can conclude that the simple encoder is not a solution to the latency problem

Generation Size	**8**	**16**	**32**	**64**	**128**	**256**
Latency	499.50	997.87	1880.42	3591.80	7032.30	13968.92
ΔL	X	1.96	1.92	1.91	1.96	1.99

Table 6.4: The average results for simple encoder for generation size 8 to 256 in milliseconds, with the differences (Δ) between previous and current generation size.

However, we can make two interesting observations. First, the ΔL values observed for the simple encoder is much more uniform, than that of the single-threaded-, smart-, and complex encoder, and the ΔL values are much closer to the expected $\sim 2x$. The second observation was made when comparing the simple-encoder with the complex encoder configured with 8 threads and the smart encoder, as illustrated in Figure 6.2. What we observer is that for $g = 256$ there is a clear benefit of having $t = 8$ as is the case with the complex- and smart encoder, where $t = 256$ for the simple encoder. This observation furthers the suspicion, that something is interfering with the execution of threads.

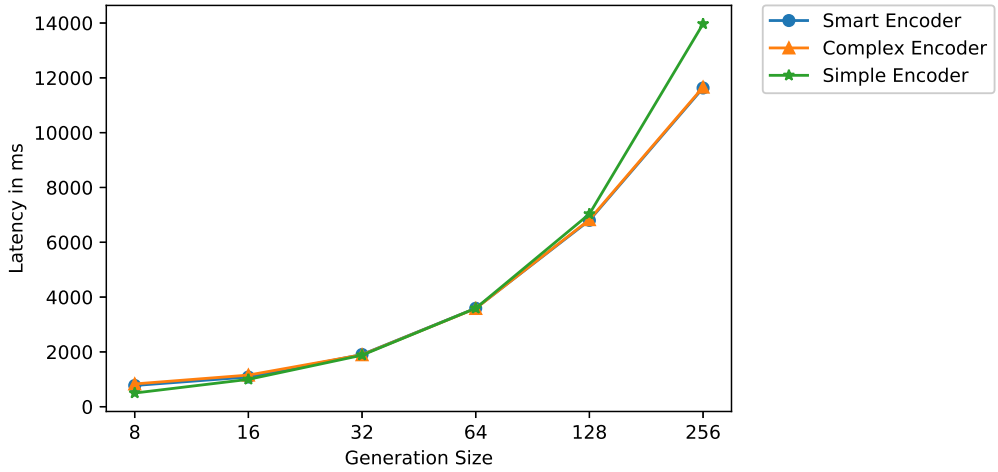

Figure 6.2: Comparison of the Simple Encoder with Complex Encoder configured with $t = 8$ and the Smart Encoder

With the results for the three encoders analysed will we investigate the two hypothesis presented above, starting with the SIMD-operations hypothesis.

6.2 Results for Simple, Complex, and Smart Encoder without SIMD

To investigate if SIMD-instructions have a negative influence on latency, we need to disable SIMD-instructions for the encoder to determine if the latency is effect in a positive or

negative way. As, the influence of SIMD-instructions should effect all three encoders in same way, either decreasing or increasing latency, have we decided to only conduct this experiment for the smart encoder.

In the source code repository, we have created *no-simd* folder, its content is a replica of source code for the encoders presented above, with a change to the file *resolved_ dependencies/fifi-f85dcd/27.0.0-0d5bf9/wscript*, where we have removed the following compiler flags `-mssse3`, `-msse4.2`, and `-mavx2`. These compiler flags are used for enabling different types of SIMD-instructions and to verify if the platform supports SIMD-instructions and this disables SIMD-instruction usage in Kodo.

Table 6.5 shows the results for these experiments, along side the ΔL values for the smart encoder run with, see Table 6.3, and without SIMD-instructions.

Generation Size	**8**	**16**	**32**	**64**	**128**	**256**
Latency	924.54	1489.08	2726.40	5318.04	10426.15	20956.72
ΔL	1.19x	1.37x	1.43x	1.48x	1.53x	1.8x

Table 6.5: Latnecy for the Smart Encoder, when no utilising SIMD-instructions, and ΔL values compared against the smart encoder utilisng SIMD

Based on the latency values we observer and the ΔL values, can we confidently conclude that SIMD-instructions is not negative factor when it comes to latency. As all cases of smart encoder run without SIMD-instructions have a worse latency, than the encoder run with SIMD-instructions. With the worst case increase in latency being $1.8x$. Based on this, we conclude that further investigation into a multi-threaded encoder should apply the usage of SIMD-instruction. We will continue investigating the potential of a cache aware multi-threaded encoder.

6.3 Cache Size Aware Encoder

Here we present a new design for multi-threaded cache size aware encoder, with the intent decreasing the latency of RLNC encoding. We also present the results for experiments conducted with cache aware encoder.

6.3.1 Cache Aware Encoder Design

There are multiple ways to make an encoder cache aware, one would simply be to say generation size (g) multiplied by the symbol size (k) must not be larger than cache size. However, this will restrict the data size which can be encoded, which is highly undesirably.

The scheme we propose is inspired by the first encoder design presented in Section 3.1 and is similar to that presented by Choi et al. [13] without the usage of graphical processing unit and intend for encoding instead of decoding. In single-threaded RLNC, we can describe the data size as presented in Equation (6.1)

$$\text{Data Size}_{\text{Bytes}} = g \cdot k \tag{6.1}$$

What we suggest for a cache aware encoder is to alter Equation (6.1) to accommodate splitting symbols into multiple fragments of length l. Each symbol is split into $\frac{k}{l}$ fragments, where each fragment is placed into a group, representing its placement in the symbol. This is illustrated in Figure 6.3, where the first fragment of each symbol is placed in *group 0*. The next fragment of each symbol will the be placed in group 1, and so on until *group n*, where $n = \frac{k}{l} - 1$.

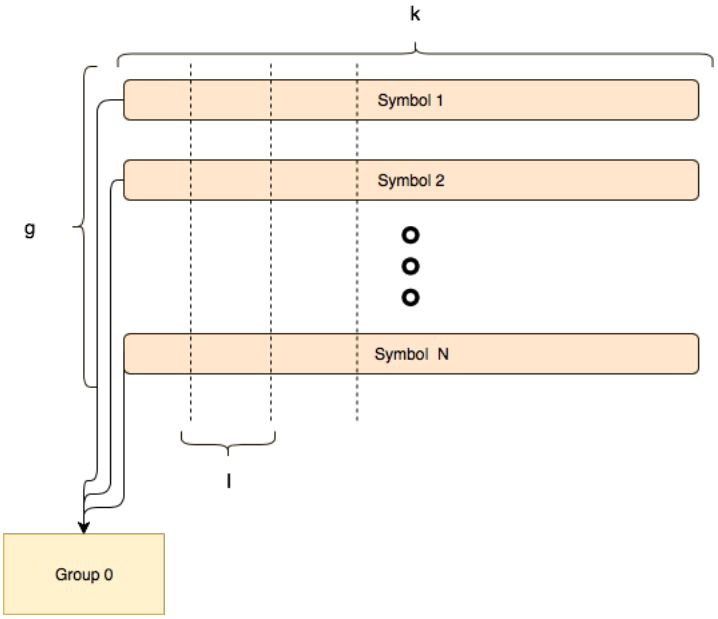

Figure 6.3: Symbol splitting for Cache Aware Encoder, with placement of fragments in Group 0Placement of fragements in group 0 after symbol splitting

To decide l, we say that for the number of threads (t) available, will the total data size operated on not exceed cache size, as shown in Equation (6.1). Then l can be decided as shown in Equation (6.2)

$$\text{cache size} \geq t \cdot g \cdot l \tag{6.2}$$

$$l = \frac{\text{cache size}}{t \cdot g} \tag{6.3}$$

However, we know that the data access pattern illustrate in Figure 6.3 is flawed, as we cannot take advantage of spatial memory location [31, p. 582-584]. A solution for this is to transpose the symbol matrix ($\mathbb{S}$) before encoding, such that we have $\mathbb{S}^T$ on the form $[k \times g]$. This allows us to use a data access pattern, which allows for the usage of spatial memory access, as the fragments of a group now are located sequential in memory. There, however, is another issue with the solution, which is that $g \cdot l \cdot k > k \cdot g$, meaning that we have to alter k and how we calculate Data $\text{Size}_{\text{Bytes}}$. We alter Equation (6.1), such that Data $\text{Size}_{\text{Bytes}}$ is calculated as shown in Equation (6.4) and adapt k to be calculated as shown in Equation (6.5).

$$\text{Data Size}_{Bytes} = g \cdot l \cdot k \tag{6.4}$$

$$k = \frac{\text{Data Size}_{Bytes}}{g \cdot l} \tag{6.5}$$

This means that for this encoder, we do not allow the user to decide k and l, and thereby restrict the configuration of RLNC parameters, however, for a *proff-of-concept* encoder, we will accept this limitation. We will also have the encoder determine the amount of threads it use by replicating the usage of `std::thread::hardware_concurrency` [22] from the smart encoder.

The next step is determining cache size, which is very difficult to do cross platform. The reason for this is that most operating systems has individual ways of determining cache size and is not an integrate part of C++. For this reason have we decided provided the cache size in bytes as a parameter to the encoder constructor.

Encoding for this encoder is performed by creating k shallow Full Vector encoders each assigned there own fragment group, and configured with $k = l$ and generation size is still the g from above. We generate the coefficient matrix ($\mathbb{C}$). The we create a task queue which contains k tasks, where each task generates g coded fragments and places them in the coded symbol matrix ($\mathbb{CS}$). When a thread is free, it will take task from the queue and execute. In Figure 6.4 we illustrate how coded fragments are placed into $\sim$ for each fragment group multiplied with the first coefficient vector (c_1) from $\mathbb{C}$.

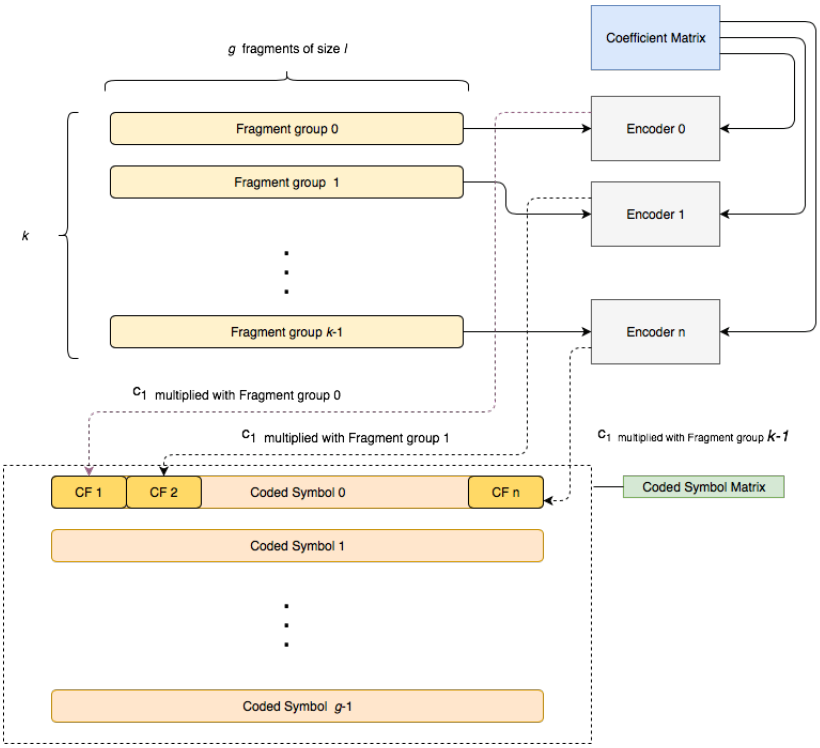

Figure 6.4: Encoding approach for the cache aware encoder. Illustrating how fragment groups are utilised to create coded fragments

The approach allows us to avoid usage of a mutex or atomic locks on $\mathbb{CS}$ as no-two threads will be writing to the same memory. Allowing for a lower latency, as threads will not have to wait for each other to finish writing to $\mathbb{CS}$.

We will implement a version of this encoder design and conduct experiments to see if performance of this encoder is better than that of the other multi-threaded encoders and the single-threaded encoder.

6.3.2 Experiment Results for the Cache Aware Encoder

Before presenting the results for the cache aware encoder, we must state that due to technical issues with the experiment machine used for the experiment presented above, have another machine been used for the experiments. The machine is an Apple MacBook Pro with an Intel Core i7-6920HQ CPU, running with a clock rate of 2.9GHz and 16GB of RAM with a clock rate of 2133MHz. Thus, the CPU used for these experiments is slower, than that of the original experiment machine. The cache size of the new experiment machine have been found by using the sytem command `sysctl` [32] with the parameter `hw.l1icachesize` which gives the size of L1 cache in bytes, and the cache size of the experiment machine is 32768 Bytes. Furthermore, due to time constraints of the project, will we reduce the amount of iterations from 1000 to 10 for each experiment.

We will also calculate the values for l and k respectively based on cache size. We determined t, by executing the code presented in Listing 3.1, on the experiment machine and the value was 8. Thus l can be calculated as shown in Equation (6.6) and the result of the calculations is presented in Table 6.6.

$$l = \frac{32768}{8 \cdot g} \tag{6.6}$$

g	8	16	32	64	128	256
l	512	256	128	64	32	16

Table 6.6: l values for cache aware encoder running with 8 threads, generation size (g), and cache size 32768 Bytes

The value of k is then calculated as illustrated in Equation (6.5), with Data $\text{Size}_{\text{Bytes}}$ = 536870912 and with respective l and g values from Table 6.6. k is calculated to 131072 for all generation size. For the same reason as stated in Chapter 5, is it difficult to state hypothesis for the cache aware encoder and as see above, is the hypothesis far from reality. Furthermore, we know the context switching will cost less than it did before, due to it know being adapted to the cache size, but the cost of writing to $\mathbb{CS}$ has increased and we cannot say if this will result in a latency high enough to undermined the decrease in latency from context switching. Therefore, we will not present hypothesis for the cache aware encoder, but we will make the assumption that it perform better than the other encoders.

In Table 6.7 we presented the results for the cache aware encoder and can state that the results are interesting. Firstly, the difference in ΔL between generation size are

insignificant including ΔL for generation size 8 and 32. Due to these minuscule difference in latency, we decided to verify the configuration files used and conduct the experiments again. The result of this was a confirmation of the findings found in Table 6.7. Based on these result, we observer that for generation sizes 8, 16, and 32 is latency increased with 3.16x, 1.63x, and 1.17 respectively, compared to the single-threaded encoder. However, we also observer that compared to smart encoder, we see gains for generation size 32, 64, 128, and 256. Where we for generation size 256 achieve a gain of $6.87x$.

g	8	16	32	64	128	256
Latency	1690.6	1697.16	1685.54	1672.17	1705.03	1692.54

Table 6.7: Observed latency values for the cache aware encoder, with regards to generation size (g)

The increase in latency for generation size 8, 16, and 32 compare to the single-threaded encoder indicates that there is an added cost when writing data to $\mathbb{CS}$. The result also shows that adapting the cache aware encoder scheme, does provided a gains for large generation size. However, none of the recorded latency values are below the one second limit and for this reason, we cannot concluded that the cache aware encoder is a solution to the latency problem. But, we will state, that further optimisation of writes to $\mathbb{CS}$ could have the potential to bring the latency further down. For this reason we suggest further investigation of the cache aware encoder approach.

This concludes our experiments, we will now switch focus to discussion of choice made during the project, conclusion, and future work.

Discussion 7

In this chapter we discus decision made through the project and the influence of theses choice on the final state of the project.

7.1 Choosing to Ingore Cache Size

Through the project we have ignored cache size until the result analysis, as it was a requirement state by Chocolate Cloud ApS as explained in the Chapter 1. However, as presented in Chapter 6 this meant that the latency for the parallel encoders increased instead of decreased as expected, resulting in an attempt to investigate if it was SIMD or cache missed which resulted in the increased latency.

Was it good decision to ignore cache size? We will argue both yes and no. Firstly, if we from the beginning had taken into account cache size and the symbol size, we would have known that context switching would be an issue, and therefore, had been able to dismiss Chocolate Cloud ApS requirement from the get go. But, this would have limited us such that the approach without taking cache size into consideration, might never had been research. Leading to the a potential simple solution to the latency issue would have been left unexplored. Thereby, we can state; Though taking cache size into consideration would have saved time, it would also have limited the process of investigating a cache size ignoring solution, leading to a gap in the research of utilising parallel computation in RLNC. For this reason will we state that though the decision might not have been the best, it was still the correct decision.

7.2 Selected Encoding Scheme

In Section 3.1 we described and selected the encoding scheme used for the project. Was it the correct scheme or would one of the two other schemes have been better suited for the project? Based on the reality, that the first scheme would result in a higher level of context switching, than the selected scheme. Will we say that the latency would have been higher for this scheme and thus, was it correct to disregard it. If we compare to the second encoding scheme, we see that though no changes to generation- nor symbol size would have occur, would an increased amount of context switching be introduced. But would it have performed better or worse than selected scheme? As the selected scheme decreased generation size, and thus the amount of operations performed within a single thread, will we say that it is more plausible that the selected scheme will have better performance, and thus we see it as the correct choice. However, we will also state that we highly recommend

investigation of the second scheme to compare the performance of that with the complex and smart encoder, and potentially design a new cache aware encoder scheme.

Furthermore, if we consider the adapted encoder presented in Section 6.3.1, we designed a more complex version of the first disregarded encoding scheme, which includes being aware of cache size. We show with this scheme decrease latency of encoding for certain generation sizes. We also suggest that further optimisation of the implementation could lead to even greater decreases in latency. Thus, we see it as the correct choice to implement a cache aware encoding scheme.

This concludes the discussion of choices made during this master thesis, we will now continue with the conclusion.

Conclusion 8

We have constructed three RLNC encoders adapted to a multi-threaded approach based on already implemented RLNC encoders provided through Kodo. We attempted to produce an encoder which seamlessly could be ported across systems without taking into account system specific configurations, such as cache size. Based on the result presented in Sections 6.1 and 6.2 we can state that the approach used for platform independent multi-threading encoding cannot fulfil the latency limitation requirement set by Chocolate Cloud ApS and the approach is outperformed by single-threaded RLNC encoders provided through Kodo. Therefore, we concluded that this approach should not be investigated further without taking cache size into consideration when encoding, as it results in a high level of context switching based on the symbol size.

We also conclude that we have designed, implemented, and empirically analysed an RLNC encoder which is aware of the cache size. We conclude that this encoder has the potential to solve Chocolate Cloud ApS latency issues, if further efforts are put into optimisation of the encoder.

Furture Work 9

Here we present potential furture work for improving a multi-threaded RLNC encoder.

In Section 6.3.2, we explain how the new approach for writing coded symbols to coded symbol matrix is suspected of increasing latency. We would like to investigate if this is true and find ways to reduce the latency if it is an issue. One thing we already are aware of is that at the moment, is the cache aware encoder using a copy operation to write to the coded symbol matrix. This is more costly than a move operation and therefore, should all copy operations, where possible, be replaced with a move operation.

In Section 6.3.2, we state that C++ does not provided a built-in abstraction for determining cache size for a machine. We would therefore like to investigate, if there exists a third-party library for this, or if none exists create a library for C++ for determining cache size.

In Section 6.1, we disregard the encoders schemes, as they are not a general solution to the latency problem. However, we wonder what would happen if one increases generation size such that $k = \frac{\text{Data Size}_{\text{Bytes}}}{g} \leq \frac{\text{Cache Size}}{n}, n \in \mathbb{N}$. Do we still see the same latency punishment as seen in Section 6.1 for the complex and smart encoder? We suggest that this should be investigate further.

In Section 3.2, we describe the need for continuously increment a counter to track how many threads have completed there run. Then when the condition counter == number of threads − 1 we know that all threads have completed and we can work with the encoded data. However, in the benchmark code, we utilise a busy `while`-loop to check if the condition is true. This is bad for performance as it constantly request a function call to counter, which is enclosed in `std::atomic`, which will halt other threads from updating the counter whilst it is being read. But there is a solution to this issue, which is to utilise a construct from C++ called `std::condition_variable` [33]. The purpose of this construct is to create a lock for one thread, which then can be unlock in another part of the program, even within another thread. Thus, when after a thread increments the counter, can we check if the condition is true and if yes release the lock. This should decrease latency even further and therefore, we would like to implement this in the encoder and investigate the change impact on latency. Furthermore, this can be applied in all four of the designed and implemented encoders.

Lastly, we would like to conduct all experiments again on the same machine, such that the foundation for comparison between the cache aware encoder and a single-thread encoder.

Glossary

CPU	Central Process Unit. 2, 9–11, 22, 26, 32
GF	Finite Field. 5–8
NC	Network Coding. 5, 7
RLNC	Random Linear Network Coding. 1–3, 5–8, 13, 15, 16, 18, 21, 29, 31, 35, 37, 39
SIMD	Single Instruction Multiple Data. 26, 28, 29, 35, 42
SSH	Secure Shell. 17

List of Figures

List of Tables

Listings

Bibliography

[1] Steinwurf Aps. *Kodo*. http://steinwurf.com/products/kodo.html. visited: 13/09-2017. 2017.

[2] Massachusetts Institute of Technology. *The MIT License*. https://opensource.org/licenses/MIT. visited: 13/03-2017. 2017.

[3] Dropbox. *Dropbox*. https://www.dropbox.com/about. visited: 4/06-2018.

[4] Google Inc. *Google Drive*. https://www.google.com/drive/. visited: 4/06-2018.

[5] Microsoft. *OneDrive*. https://onedrive.live.com/about/en-us/. visited: 4/06-2018.

[6] Apple Inc. *iCloud - The best place for all your photos, files, and more.* https://www.apple.com/lae/icloud/. visited: 4/06-2018.

[7] James Kosur. *What Is The Fappening? It's A Dirty Moment Captured In Time*. https://www.business2community.com/social-buzz/fappening-diving-deep-scandal-0995227. visited: 4/06-2018.

[8] Chocolate Cloud ApS. *Chocolate Cloud ApS*. https://www.chocolate-cloud.cc/. visited: 28/02-2018. 2017.

[9] Chocolate Cloud ApS. *skyflok*. https://www.skyflok.com/. visited: 13/02-2018. 2018.

[10] Tracey Ho, Muriel Mèdard, Jun Shi, Michelle Effros, and David R. Karger. "On Randomized Network Coding". In: *In Proceedings of 41st Annual Allerton Conference on Communication, Control, and Computing*. 2003.

[11] Lars Nielsen. *Performance Study of Encoding in Random Linear Network Coding*. Tech. rep. Aalobrg University, 2018.

[12] William Stallings. *Operating Systems Internals and Design Principles*. 7nd. Pearson, 2012. ISBN: 9780273751502.

[13] Seong-Min Choi and J. S. Park. "Massive parallelization technique for random linear network coding". In: *2014 International Conference on Big Data and Smart Computing (BIGCOMP)*. Jan. 2014, pp. 296–299. DOI: 10.1109/BIGCOMP.2014.6741456.

[14] S. Wunderlich, J. Cabrera, F. H. P. Fitzek, and M. V. Pedersen. "Network Coding Parallelization Based on Matrix Operations for Multicore Architectures". In: *2015 IEEE International Conference on Ubiquitous Wireless Broadband (ICUWB)*. Oct. 2015, pp. 1–5. DOI: 10.1109/ICUWB.2015.7324482.

[15] Steinwurf ApS. *Steinwurf ApS*. http://steinwurf.com/about.html. visited: 13/09-2017. 2017.

[16] R. Ahlswede, Ning Cai, S. Y. R. Li, and R. W. Yeung. "Network information flow". In: *IEEE Transactions on Information Theory* 46.4 (July 2000), pp. 1204–1216. ISSN: 0018-9448. DOI: 10.1109/18.850663.

[17] Steven J. Leon. *Linear Algebra with Applications*. 7nd. Pearson, 2006. ISBN: 0132003066.

[18] William Stallings. *Cryptography and Network Security: Principles and Practice*. 5th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010. ISBN: 0136097049, 9780136097044.

[19] Morten V. Pedersen, Janus Heide, and Frank H. P. Fitzek. "Kodo: An Open and Research Oriented Network Coding Library". In: *NETWORKING 2011 Workshops: International IFIP TC 6 Workshops, PE-CRN, NC-Pro, WCNS, and SUNSET 2011, Held at NETWORKING 2011, Valencia, Spain, May 13, 2011, Revised Selected Papers*. Ed. by Vicente Casares-Giner, Pietro Manzoni, and Ana Pont. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 145–152. ISBN: 978-3-642-23041-7. DOI: 10.1007/978-3-642-23041-7_15. URL: http://dx.doi.org/10.1007/978-3-642-23041-7_15.

[20] Lars Nielsen, René Rydhof Hansen, and Daniel Enrique Lucani Rötter. "Latency Performance of Encoding with Random Linear Network Coding". English. In: *European Wireless*. IEEE Press, 2018.

[21] Jakob Progsch. *progschj/ThreadPool*. https://github.com/progschj/ThreadPool. visited: 20/05-2018. 2018.

[22] cppreference.com. *std::thread::hardware_ concurrency*. http://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency. visited: 08/05-2018. 2018.

[23] cppreference.com. *Fixed width integer types*. http://en.cppreference.com/w/cpp/types/integer. visited: 08/05-2018. 2018.

[24] cppreference.com. *Atomic operations library*. http://en.cppreference.com/w/cpp/atomic. visited: 08/05-2018. 2018.

[25] Hewlett-Packard. *HP ProDesk 490 G3 Microtower Business PC - Specifications*. https://support.hp.com/lt-en/document/c04835039. visited: 12/01-2018. 2017.

[26] Redhat Inc. *Fedora Project*. https://getfedora.org/en/. visited: 13/04-2018. 2017.

[27] Inc. SSH Communications Security. *SSH (Secure Shell)*. https://www.ssh.com/ssh/. visited: 13/04-2018. 2017.

[28] Jim Meyering. *nohup(1) - Linux man page*. https://linux.die.net/man/1/nohup. visited: 13/03-2018. 2017.

[29] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number: 248966-033, June 2016*. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. visited: 27-11-2017.

[30] Intel. *Intel Processor Architecture: SIMD Instructions*. https://www.intel.com/content/dam/www/public/apac/xa/en/pdfs/ssg/Intel_Processor_Architecture_SIMD_Instructions.pdf. visited: 5/06-2018.

[31] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd. Pearson, 2014. ISBN: 9781292025841.

[32] Apple Inc. *sysctl.h*. https://opensource.apple.com/source/xnu/xnu-792.12.6/libkern/libkern/sysctl.h. visited: 3/06-2018.

[33] cppreference.com. *std::condition_ variable*. http://en.cppreference.com/w/cpp/thread/condition_variable. visited: 08/05-2018. 2018.