Bachelorarbeit

im Studiengang Medieninformatik

JUMP-BASED PARALLEL PSEUDO RANDOM NUMBER GENERATORS

vorgelegt von

Patrick Baumann

an der **Hochschule der Medien Stuttgart** am **28. Februar 2024**

zur Erlangung des akademischen Grades eines

Bachelor of Science

Erstprüfer:	**Prof. Dr. Roland Schmitz**
Zweitprüfer:	**Dipl.-Inf. Christian Siebert**

Ehrenwörtliche Erklärung

Hiermit versichere ich, Patrick Baumann, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Jump-based parallel pseudo random number generators“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO, § 23 Abs. 2 Master-SPO (Vollzeit)) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

[illegible]

Leinfelden, 28. Februar 2024

Zusammenfassung

Viele Anwendungen im Bereich des High-Performance Computing benötigen große Mengen an Zufallszahlen, welche mit sogenannten Pseudozufallszahlengeneratoren erzeugt werden. Eines der Hauptprobleme in diesem Bereich ist das der Skalierbarkeit: Das Erzeugen von Zufallszahlenströmen scheint zunächst eine grundsätzlich sequentielle Operation zu sein. Diese Arbeit beschäftigt sich mit dem Thema, wie durch die Implementierung einer Sprung Funktionalität, welche es erlaubt, zu einem beliebigen Punkt im Zufallszahlenstrom zu springen, bisher bestehende Zufallszahlengeneratoren parallelisiert werden können.

Abstract

Many applications in high-performance computing require the generation of large streams of random numbers using pseudo-random number generators. One of the main issues arising in this area is a lack of scalability, since generating streams of numbers at first glance seems to be an inherent sequential operation. This thesis deals with the subject of parallelizing existing random number generators, with the implementation of a jump functionality, which allows 'jumping' to a specific point in the stream of random numbers.

Nomenclature

List of Acronyms

CPU	central processing unit
GB	gigabyte
GCC	GNU compiler collection
GHz	gigahertz
GMP	GNU Multiple Precision Arithmetic library
HLRS	Höchstleistungsrechenzentrum Stuttgart
HPE	Hewlett Packard Enterprise
LCG	linear congruential generator
LFSR	linear-feedback shift register
MPI	message passing interface
PB	petabyte
PRNG	pseudo-random number generator
RNG	random number generator
SW	sliding window algorithm
SWD	sliding window algorithm with polynomial decomposition
TinyMT	Tiny Mersenne Twister

Definitions

α_i	the ith coefficient in a characteristic polynomial
$\prod$	product of a sequence
$\sum$	summation of a sequence
a	multiplier of a linear congruential generator

a_i	ith coefficient of the jump polynomial
b_i	ith coefficient of a polynomial in $\mathbb{F}_2$, or the ith bit of a binary number, depending on the context
c	increment of a linear congruential generator
$\mathrm{c}_{\mathrm{swd}}$	initialisation time of SWD
d	exponent, used in sliding window decomposition algorithm
$\det(\mathbf{M})$	determinant of a matrix $\mathbf{M}$
$\mathrm{ex}_{\mathrm{swd}}$	execution time of SWD
$\mathrm{ex}_{\mathrm{sw}}$	exectution time of SW
$GF(2)$	Galois field with two elements, synonym for $\mathbb{F}_2$
i	index variable
j	index variable
k	positive integer, denoting the degree of a polynomial, an exponent, or the width of a binary number, depending on the context
l	positive integer, often denoting an exponent
m	modulus of an LCG index of second to last polynomial in SW depending on the context
mad	median absolute deviation
med	median
n	arbitrary integer, or jump size in an LCG, depending on the context
p	number of processes in Amdahl's law
q	small integer in the range from 2 to 10
r	positive integer, usually denoting a remainder
r_p	fraction of parallel computation in Amdahl's law
r_s	fraction of sequential computation in Amdahl's law
s	shift size of a shift matrix
s_{rng}	state size of an RNG in bits
$T(p)$	execution time of a program with p processes
u_n	output of an $\mathbb{F}_2$-linear generator after n steps in the sequence

v	exponent denoting the jump size in an $\mathbb{F}_2$-linear generator
X_n	state of a LCG after n steps
z	indeterminate of a polynomial
$g(z)$	jump polynomial of an $\mathbb{F}_2$-linear generator
$h(z)$	polynomial in $\mathbb{T}_q$
$P(z)$	polynomial such that $g(z) = z^v + P(z)p(z)$
$p(z)$	characteristic polynomial of matrix $\mathbf{A}$
$Q(z)$	polynomial in the form of $c\sum_{i=0}^{n-1} z^i$
$q(z)$	minimal polynomial of matrix $\mathbf{A}$ or linear-feedback shift register
$r_i(z)$	polynomial in the form of $c\sum_{j=0}^{2^i-1} z^j$, with $r_0 = c$
$s_i(z)$	polynomial in the form of $\prod_{j=i+1}^{k-1} b_j z^{2^j}$, with $s_{k-1} = 1$
$t_i(z)$	polynomial in $\mathbb{T}_q$, enumerated through gray code
$\mathbb{F}_2$	finite field with two elements
$\mathbb{F}_2^k$	vector space with dimension k over $\mathbb{F}_2$
$\mathbb{N}$	the natural numbers
$\mathbb{R}$	the real numbers
$\mathbb{T}_q$	set of polynomials of exactly degree q
$\mathbf{A}$	state transformation matrix for $\mathbb{F}_2$-linear generators
$\mathbf{B}$	output transformation matrix for $\mathbb{F}_2$-linear generators
$\mathbf{I}$	identity matrix
$\mathbf{L}^s$	left-shift matrix with shift size s
$\mathbf{R}^s$	right-shift matrix with shift size s
$\mathbf{x}_n$	state vector of an $\mathbb{F}_2$-linear generator after n steps in the sequence
$\mathbf{y}_n$	output vector after applying the output transformation matrix in an $\mathbb{F}_2$-Linear generator after n steps in the sequence

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Many applications in high-performance computing require the generation of large streams of random numbers. One of the main issues arising in this area is the lack of scalability, by existing random number generators having no or very limited functionality to be parallelized. Since generating a stream of numbers at first appears to be an inherent sequential operation, it becomes problematic when considering Amdahl's Law, which states that the speedup of an application is limited by the fraction of time spent on sequential computation.

But what does parallelization of a random number generator mean? Many random number generators are based on a linear recurrence, meaning the successive numbers in a stream depend on one or more of the previously generated numbers. These recurrences start with an initial value, called the *seed*. Therefore, parallelization is transforming a linear recurrence into an explicit formula, which can then be evaluated in order jump ahead in the generated stream. The restriction here is that the explicit formula needs to be evaluated in less than linear time; otherwise, scaling is impossible. In a parallel application it is then possible to divide a single large stream into several smaller substreams, which can be generated by individual threads or processes.

One of the common approaches is to simply provide each thread or process with an unique seed. This might work for generators with a very large period, for example the Mersenne Twister, but there is always some risk that generated substreams may overlap. In scientific applications, this may introduce a bias to the generated random state, making the results unusable. Jumping ahead completely solves this issue, since all the threads or processes generate numbers from separate substreams of a large stream. Some applications also require to create substreams, which overlap by a small, predefined amount. This is completely impossible to scale without a jump functionality. Furthermore, it makes it easier to reproduce previous results, since all numbers are generated from the same seed. This allows for better fine tuning of parameters to create a stream of numbers suitable for a specific application.

In this thesis, the parallelization of existing random number generators is explored. Two libraries were implemented, Prand48 and F2Lin, written in the C programming language. Prand48 is based on a linear congruential sequence, while F2Lin is based on $\mathbb{F}_2$-linear transformations. Both libraries implement a 'jump-ahead' functionality. To show the efficiency of a jump functionality the libraries were benchmarked in that regard, proving that a jump can be faster than sequential computation by several orders of magnitude. The scaling capacities of the libraries in a parallel computation environment were also measured.

The following chapters are structured as follows: In chapter 2, the mathematical properties of linear congruential and $\mathbb{F}_2$-linear generators are explained. Chapter 3 provides an overview of how the libraries were benchmarked, explaining common terminology used in this context. Both chapter 4 and 5 afterwards give an overview on both of the libraries, with a performance evaluation of both of them.

Chapter 2

Types of PRNGs

This chapter deals with the mathematical properties behind the random number generation of the two implemented libraries. For each type of generator, its mathematical properties and the theory behind jumping ahead in the sequence will be given. Section 2.1 introduces the linear congruential generator, implemented in PRAND48. Section 2.2 deals with $\mathbb{F}_2$-Linear random number generators, implemented in F2LIN.

2.1 Linear congruential generator

A linear congruential generator (LCG) is based on the sequence:

$$X_{n+1} = aX_n + c \pmod{m}, \tag{2.1}$$

where m is called the modulus, a is the multiplier, c is the increment and X_0 the seed. [Knu81, P.10] It was originally proposed by H.D. Lehmer in 1949, but without the increment [Leh49]. This form of LCG is also called the *Lehmer random number generator*.

Choosing good values of a, c and m is of great importance of the speed of the implementation, the distribution properties and the period of a LCG.

2.1.1 Choice of modulus

The modulus gives an upper bound on the period of an LCG, as well as an influence on the speed of an implementation. Regarding the choice of m, it is preferable to choose it as some number 2^k. By choosing powers of two, the modulus can be computed simply by truncating a binary number to k digits, which can be done by a single binary 'and' operation with a bit mask of $2^k - 1$. Also, if a common integer width is chosen for k, such as 32 or 64, the modulo is automatically calculated due to integer overflow.

Another property of the modulus being 2^k is, that in an implementation it is possible to do all computations with the word size of a CPU (32 or 64 bit) and then truncate the result to k bits, since $(n \bmod 2^k) \bmod 2^l = n \bmod 2^l$, with $l < k$ and $l, k \in \mathbb{N}$.

2.1.2 Choice of multiplier and increment

The choice of multiplier and increment directly affects the period and the distribution of an LCG. The choices of interest for a and c are those that produce a maximum period. It should be noted, however, that choosing a value for a with a maximum period does not necessarily produce a generator with a good distribution. [Knu81, P. 17] gives the example of $a = c = 1$, which gives a generator that simply produces the sequence x, $x + 1$, $x + 2$, $\dots$.

The conditions for an LCG to have a maximum period are given by the following theorem [Knu81, P. 17]:

Theorem A 2.1.1 *The linear congruential sequence defined by m, a, c, and X_0 has period length m if and only if*

1. *c is relatively prime to m;*
2. *$b = a - 1$ is a multiple of p, for every prime p dividing m;*
3. *b is a multiple of 4, if m is a multiple of 4.*

In the case of choosing powers of 2 as the modulus, for the second condition, the only prime that divides m is 2. Also, m is usually greater than 2^4, so m is at least a multiple of 4. So b must also be a multiple of 4. This leaves values for a being in the form of $a \bmod 8 = 5$ or 3 [Knu81, P. 21]. Choosing a value for c is simply choosing an odd number.

Finding good values then mainly depends on the requirements for the generated distribution. [L'E99] lists a table of good values for a.

2.1.3 Criticisms of LCGs

LCGs are often criticised for producing a random sequence with undesirable properties, failing statistical tests such as the 'Die Harder Test Suite' [Bro] or 'TestU01' [LS07].

According to [L'E99], one of the main drawbacks of LCGs is a small period in the low order bits, which is why it is often recommended to avoid this type of generator. By choosing a value for m like $m = 2^k$ with $k >= 128$, and for example using only the 53 most significant bits of a generated number to create a floating point number, this drawback becomes less severe. [O'N14] also shows examples of LCGs with a period of 2^{92} and 2^{128} that pass all the tests of TestU01's "BigCrush" test suite.

Nowadays, many compilers have built-in support for 128-bit arithmetic, so calculating such large numbers is less of a problem.

2.1.4 Jumping Ahead

In order to jump ahead in the stream of random numbers generated by an LCG, the equation 2.1 must be transformed from a sequence to an explicit formula.

Writing out the sequence for the first few numbers gives the following (the modulus is omitted for brevity):

$$\begin{aligned}
X_1 &= aX_0 + c \\
X_2 &= aX_1 + c = a(aX_0 + c) + c = a^2X_0 + ac + c \\
X_3 &= aX_2 + c = a(a^2X_0 + ac + c) + c = a^3X_0 + a^2c + ac + c
\end{aligned}$$

Generalizing this formula for X_n is:

$$\begin{aligned}
X_n &= (a^nX_0 + a^{n-1}c + \cdots + a^1c + a^0c) \\
&= (a^nX_0 + c\sum_{i=0}^{n-1} a^i)
\end{aligned}$$

And finally, transforming the sum into its explicit form results in the final equation:

$$X_n = (a^nX_0 + c\frac{a^n - 1}{a - 1}) \pmod{m} \tag{2.2}$$

The implementation of this formula will be described in section 4.1

2.2 $\mathbb{F}_2$-Linear Random Number Generators

The term $\mathbb{F}_2$-Linear Random Number Generators was established by Pierre L'Ecuyer and Francois Panneton [Pan07]. It is a class of generators that are all based on linear recurrences modulo 2. In a nutshell, a linear recurrence is a sequence of numbers, where successive elements depend on one or more numbers previously generated in the sequence. One of the most famous examples of a linear recurrence is the Fibonacci Sequence, which has the form of $x_n = x_{n-1} + x_{n-2}$, with $x_0 = 0, x_1 = 1$. Types of generators in this class include linear feedback shift registers (LFSR), XORSHIFT generators [Mar03] or the MERSENNE TWISTER [Nis98]. It should be noted that all the following equations operate on the field $\mathbb{F}_2$ where addition (+) and and subtraction (−) are identical operations and can be implemented by a bitwise 'xor' operation.

2.2.1 General Outline

In general, the state of a random number generator is represented by a k-bit vector $\mathbf{x}_n \in \mathbb{F}_2^k$. The transformation from one state to the next is then usually implemented by a series of 'xor', 'shift' and 'rotation' operations of the state with some constants. This transformation can be implemented by a so-called transition matrix $\mathbf{A}$. A state can also be transformed by an $w \times k$ output transformation matrix $\mathbf{B}$. The resulting vector $\mathbf{y}_n$ is not part of the state, but can further improve the quality of a generator [Nis98] [Bla18]. Finally, $\mathbf{y}_n$ is mapped to an output $u_n \in [0, 1)$ via a function $f : \mathbb{F}_2^k \to \mathbb{R}, \mathbf{y}_n \mapsto u_n$.

[Pan07] summarizes these steps with the three equations:

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1}, \tag{2.3}$$

$$\mathbf{y}_n = \mathbf{B}\mathbf{x}_n, \tag{2.4}$$

$$f(\mathbf{y}_n) = u_n = \sum_{l=1}^{w} y_{n,l-1} 2^{-l} \tag{2.5}$$

The period of such a random number generator can never exceed $2^k - 1$, since this is the maximal number that can be represented by a k-bit binary number.

When testing for maximum periodicity of a generator, [Pan07] describes the following method to do so:

Consider the characteristic polynomial of $\mathbf{A}$:

$$p(z) = det(\mathbf{A} - z\mathbf{I}) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k, \quad \alpha_n \in \mathbb{F}_2 \tag{2.6}$$

$p(z)$ is also the characteristic polynomial of the linear recurrence:

$$x_n = (\alpha_1 x_{n-1} + \cdots + \alpha_k x_{n-k}). \tag{2.7}$$

The minimal polynomial of $\mathbf{A}$ $q(z)$ is the polynomial with the smallest degree for which $q(\mathbf{A}) = 0$, which implies that $p(z)$ is a multiple of $q(z)$.

Since the sequence defined in 2.3 satisfies the same recurrence that corresponds to the minimal polynomial of $\mathbf{A}$, this polynomial can be calculated with the *Berlekamp-Massey algorithm* ([Mas69]).

[Pan07] gives the following conditions to test if a generator is to have a maximum period. A generator has a maximum period if $p(z)$ is primitive, meaning it is irreducible (it has no divisor other than one and itself) and for all prime divisors p_i of $r = 2^k - 1, z^{\frac{r}{p_i}} \not\equiv 1 \mod p(z)$. If r is prime, the second condition is always true. Then r is called a Mersenne prime. One example of such a number is 19937, which is the state size of the MERSENNE TWISTER [Nis98].

For the rest of this thesis, when talking about the polynomial $p(z)$ of $\mathbf{A}$, it will be referred to as the minimal polynomial.

2.2.2 Xorshift

XORSHIFT only operates on bitwise 'shift' and 'xor' operations. In all of the following code examples, `a << b` and `a >> b` will refer to left and right shifts of `a` by `b` respectively. `a ^ b` denotes a binary 'xor' operation of `a` and `b`. It was first introduced by George Marsaglia in 2003 in [Mar03]. Since they only use very basic CPU instructions, they are easy to implement.

One thing to note about the XORSHIFT generator is, that it doesn't have an output transformation, as described in 2.4.

The state size k of an Xorshift random number generator is usually a multiple of the width of CPU words, e.g. 32, 64, ... with theoretical maximum periods of $2^k - 1$.

The most basic form of an Xorshift random number generator is `y ^ (y << a)` or `y ^ (y >> a)`.

As mentioned in 2.2.1, a state with size k is represented as a binary vector $\mathbf{x}$ in $\mathbb{F}_2^k$, with a $k \times k$ transition matrix $\mathbf{A}$. For example, an 'xor' operation on a vector $\mathbf{x}$ can be represented by multiplication with the identity matrix $\mathbf{I}$. A shift operation is represented by a shift matrix. To shift a vector n bits to the right, the diagonal row of the identity matrix gets shifted n positions to the left and vice versa for a right shift operation. These matrices are denoted by $\mathbf{L}^s$ and $\mathbf{R}^s$, with a shift size of s. For example, an operation like `y ^ (y << 3)` with a state of size 5 can be represented by the equation $\mathbf{A} = (\mathbf{I} + \mathbf{L}^3)$. The resulting transition matrix $\mathbf{A}$ would then be:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

A state transition is now represented as: $\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i$. Applying the same transition twice is: $x_{i+2} = \mathbf{A}\mathbf{A}x_i = \mathbf{A}^2\mathbf{x}_i$. And n state transitions are: $\mathbf{x}_{i+n} = \mathbf{A}^n\mathbf{x}_i$. A more complex operation like `y ^= (y << s_1); y ^= (y >> s_2); y ^= (y << s_3)` would be: $(\mathbf{I} + \mathbf{L}^{s_1})(\mathbf{I} + \mathbf{R}^{s_2})(\mathbf{I} + \mathbf{L}^{s_3})$. With $s_1 = 1$, $s_2 = 3$, $s_3 = 2$ the transition matrix $\mathbf{A}$ in $\mathbb{F}_2^5$ would be:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{2.9}$$

The period of a Xorshift generator depends on the choice of s_1, s_2, s_3. A list of combinations of tuples that produce a maximum period can be found in [Mar03].

2.2.3 Jumping Ahead

As mentioned before, a state transition can be represented by equation 2.3. Implementing a jump functionality can be done by computing $\mathbf{A}^v$, where v is the jump size. A jump of v then corresponds to the equation $\mathbf{A}^v\mathbf{x}$. Taking the vth power of $\mathbf{A}$, can be calculated with the 'Right-to-left binary method' as described in [Knu81, P. 462], which has a runtime of $O(k^3 \log v)$ for matrix exponentiation. This method has a high memory footprint, though. For example, if a single component takes up n byte of memory, then $\mathbf{A}$ will take k^2n bytes of memory. This becomes impractical for large state random number generators. [L'E08] lists an example of the Mersenne Twister, where $k = 19937$, whose transition matrix required 47.4MB of memory.

[L'E08] thus presents a polynomial based approach, which requires less memory and is also faster. First, the characteristic polynomial of $\mathbf{A}$ is determined:

$$p(z) = \det(\mathbf{A} - z\mathbf{I}) = z^k + \alpha_1 z^{k-1} + ... + \alpha_{k-1} z + \alpha_k \tag{2.10}$$

An important property of the characteristic polynomial of $\mathbf{A}$ is

$$p(\mathbf{A}) = 0, \tag{2.11}$$

since $p(\mathbf{A}) = det(\mathbf{A} - \mathbf{A}I) = det(0) = \mathbf{A}^k + \alpha_1 \mathbf{A}^{k-1} + ... + \alpha_{k-1}\mathbf{A} + \alpha_k \mathbf{I} = 0$

Next, the following polynomial is defined (where j is the jump size):

$$g(z) = z^v \bmod p(z) = a_1 z^{k-1} + .. + a_{k-1} z + a_k \tag{2.12}$$

$g(z)$ will be called the jump polynomial for the remainder of this thesis. Now there exists a polynomial $P(z)$ such that $g(z) = z^v + P(z)p(z)$. This means $g(\mathbf{A}) = \mathbf{A}^v + P(\mathbf{A})0 = \mathbf{A}^v$ which than can be used to calculate $\mathbf{A}^v$:

$$\mathbf{A}^v = g(\mathbf{A}) = a_1 \mathbf{A}^{k-1} + ... + a_{k-1}\mathbf{A} + a_k \mathbf{I} \tag{2.13}$$

The jump can now be written as:

$$\mathbf{A}^v \mathbf{x} = g(\mathbf{A})\mathbf{x}, \tag{2.14}$$

where $\mathbf{x}$ is the seed vector.

There are several techniques in order to evaluate this equation, which will be explained in detail in 5.2.

Chapter 3

Benchmarks and Performance

This chapter provides an overview of performing benchmarks and common methodologies for performance evaluation.

3.1 Hardware

The benchmarks were conducted at the High Performance Computing Center Stuttgart (HLRS) in Germany using the HPE Apollo, also known as 'Hawk'.

Hawk is one of Europe's fastest computing systems and is optimized to provide large scale computing power for complex simulations. Hawk has 5632 compute nodes with a peak performance of 26 Petaflops. Each node has two CPUs, an AMD EPYC 7742, with 64 cores per CPU running at 2.25 GHz. Its total memory consists of approximately 1.44PB, with 256 GB of memory per node [Stu].

3.2 Execution of benchmarks

Benchmarking assesses the performance of different algorithms and the application as a whole. It is done on either single functions or parts of the code that represent common operations, such as jumping ahead in the stream and then calculating numbers.

To benchmark a single functions, it is called in a loop of n iterations using a 'for' loop. The time is measured from the start to the end of the loop, dividing the resulting duration by n.

The execution time of an empty loop was also measured. However, the results showed that the loop execution time was so insignificant (in the order of 10^{-8} seconds) that it can be ignored.

This process is then repeated k times, with each measurement stored in an array of length k. Thus, a single function is called a total of $n \times k$ times.

As the execution time of a function or program can vary due to external factors, for example the operating system, taking an average of the results may lead to bias caused by outliers. To avoid this, the results must be sanitized. This is achieved by sorting the results and calculating the median *med* and the median absolute deviation *mad*. Outliers that deviate too much from med $\pm$ $(c \times \text{mad})$, where c is some constant, are removed from the list. The benchmark result is then calculated as the average of this sanitized list.

Benchmarking a common operation is performed in a similar manner, by executing the operation repeatedly in a loop.

All benchmarks utilise the MESSAGE PASSING INTERFACE (MPI), which allows for a larger sample size and investigation of the scalability of an application. MPI is a standard for the message-passing parallel programming model, designed to implement and run parallel applications. It does so by spawning several sub-processes, which move data from their address space to that of another process through cooperative operations on each process. [for]

To create n samples with p processes, each process creates n/p samples. n is always chosen to evenly divide the number of processes. The samples are then gathered in a single process, which calculates the benchmark result and saves it in a .csv file. The .csv files are then analyzed in Python using data analysis and plotting tools such as PANDAS [M+10] and MATPLOTLIB [Hun07].

Some of the benchmarks required the usage of external libraries. These were: GMP version 6.3.0, GF2X version 1.3.0, NTL version 11.5.1, and FLINT version 3.0.1.

The benchmarks for large strong scaling were compiled and run using the Hewlett Packard Enterprise implementation of MPI, included in the MESSAGE PASSING TOOLKIT (MPT), version 2.23, with the following compilers: The MPI C and C++ compilers, both using the GNU compiler collection (GCC), version 10.2.0.

All other benchmarks other benchmarks were compiled and run with the Open MPI implementation, version 4.1.4. It uses the MPI C and C++ compilers, with GCC version 10.2.0.

3.3 Strong scaling

Strong scaling refers to the scalability of an application. The principle behind this is to measure how much faster an application runs with an increased number of processes for a given problem size. This is called the 'speedup'.

However, before strong scaling is explained further, one must first consider Amdahl's Law, which states:

$$\text{Speedup} = 1/(\mathrm{r}_s + \frac{\mathrm{r}_p}{p}), \tag{3.1}$$

where r_s is the proportion of time spent in the serial part, meaning it cannot be parallelized, and r_p is the proportion of time spent in the parallel part of a program. So $\mathrm{r}_s + \mathrm{r}_p = 1$. p denotes the number of processes used.

To determine the theoretical *maximum speedup* of an application, $\lim_{p\to\infty} 1/(\mathrm{r}_s + \frac{\mathrm{r}_p}{p})$ is calculated. This shows that even if only a small fraction of the execution time of a program is spent in the serial part, the speedup is severely limited, even with an infinite number of processes. For example, if only 5 percent of an application is spent in the serial part, the maximum speedup possible is only 20.

As mentioned above, strong scaling keeps the problem size n constant. To determine the speedup, first measure the *serial execution time* $T(1)$, which is the time it takes a single process to complete execution. The speedup can then be calculated by comparing the serial execution time to the time $T(p)$ it takes for p processes to execute which is:

$$\text{Speedup} = \frac{T(1)}{T(p)}. \tag{3.2}$$

In the case of creating parallel substreams of random numbers using a jump ahead, the maximum speedup is mainly limited by the time it takes to initialise and execute a jump (representing the serial part of Amdahl's law).

For the strong scaling experiments in this thesis, it was measured how long it takes to calculate n random numbers, with p processes ranging from 1 to 128. To avoid load imbalance, meaning one processor having do more work than the others, the problem size for a single process is calculated by $\lfloor \frac{n}{p} \rfloor$, with the remainder r of the division being evenly distributed among the first r processes.

As a quick side note, when the results are displayed, the *ideal speedup* is also given to make it easier to grasp the scaling of an application. An ideal speedup would be a linear increase of the speedup with respect to the number of processes.

Chapter 4

Prand48

PRAND48 is a linear congruential generator. Its design is closely tied to the family of `rand48` functions from the C standard library.

`rand48` functions all generate 48-bit wide pseudo-random numbers based on a linear congruential sequence. In order to generate random numbers there are a total of six interfaces, which are categorized into three subsections, depending on the type of number to be generated. For each subsection there is one interface where the user supplies a seed and one where the seed is set automatically:

1. `double` in range 0.0 to 1.0: `drand48()` and `erand48(short seed[3])`
2. `long` in range 0 to $2^{31} - 1$: `lrand48()` and `nrand48(short seed[3])`
3. `long` in range -2^{31} to $2^{31} - 1$: `mrand48()` and `jrand48(short seed[3])`

Furthermore, there are three functions which let the user customize `drand48()`, `lrand48()` and `mrand48()`:

1. `srand48(long seed)`: Copies the argument `seed` into the upper 32 bits of the internal seed. The lower 8 bits of the seed are initialized to `0x330e`. This also resets the increment and multiplier of the algorithm to the default value.
2. `seed48(short seed[3])`: Allows the user to specify all 48 bits of the seed.
3. `lcong48(unsigned short param[7])`: Allows the user to customize the multiplier and increment of **all** six functions, where the first three shorts are the seed, the second three the multiplier and the last the increment.

The following section provides a brief overview of the algorithms required to implement a jump ahead , based on equation 2.2 from chapter 2. This is followed by an introduction of the interface of PRAND48, and a section on the implementation of the library. The last section evaluates the performance of the library.

4.1 Theory and algorithms

To implement equation 2.2, it is split into two parts:

$$a^n r_0 \mod m \tag{4.1}$$

$$c\frac{a^n - 1}{a - 1} \mod m \tag{4.2}$$

4.1 can be evaluated using the *right-to-left binary method for exponentiation.* The exponent n first gets written in binary representation:

$$n = \sum_{i=0}^{k-1} b_i 2^i \tag{4.3}$$

with $b \in \{0, 1\}$ and $k = \lceil \log_2 n \rceil$, the length of n in bits. Now, a^n is written as:

$$a^{\sum_{i=0}^{k-1} b_i 2^i} = \prod_{i=0}^{k-1} a^{b_i 2^i} \tag{4.4}$$

The algorithm starts with the right-most bit of n and shifts it one bit to the right on each iteration. Furthermore, it squares the base a on each iteration. The result is accumulated by adding the current base to the accumulator `ret`, if the least significant bit of n is 1 at the current iteration step. The algorithm works the same when executed in modulo m, by calculating each of the products in the modulo.

The implementation of the algorithm, shown in listing 4.1 is taken from [Sch05, P.344] and has a runtime of $O(k)$.

```
right_to_left_binary_method(a, n, m):
    base = a, exp = n
    if m = 1 return 0

    ret = 1
    while exp > 0
        if (exp mod 2 == 1)
            ret = (ret * base) mod m
        exp = exp >> 1
        base = (base * base) mod m
    return ret
```

Listing 4.1: Right-to-left binary method

Theoretically, equation 4.2 should be easy to evaluate by finding the inverse of $a - 1 \pmod m$. However, a problem arises, when considering the second condition of theorem 2.1.1. It states that $a - 1$ is a multiple of every prime that divides the modulo. Since m is chosen to be an arbitrary power of 2, this means that a must be odd, if the generator is to have a maximum period. So the number $a - 1$ is a zero divisor in modulo m.

The other option is to evaluate the sum $c\frac{a^n-1}{a-1} = c\sum_{i=0}^{n-1} a^i \pmod m$ itself. [Bro94] gives the algorithm in listing 4.2, which runs in $O(\log m)$ time:

```
1   algorithm_C(a, n, c, m):
2       C = 0               // result (Q(z))
3       r = c               // increment
4       z = a               // multiplier
5       i = n mod m         // jump size
6
7       while i > 0
8           if (i mod 2 == 1)
9               C = (C * z + r) mod m
10          r = (r * (z + 1)) mod m
11          z = (z * z) mod m
12          i = i >> 1
13      return C
```

Listing 4.2: Algorithm C

In following explanation, the modulus is omitted for brevity. Also, note that the variable `i` is different from the index i used in the explanation. The general idea behind the algorithm is to evaluate the polynomial $Q(z) = c\sum_{i=0}^{n-1} z^i$ by decomposing it into sub-polynomials, each of the form $r_i(z) = c\sum_{j=0}^{2^i-1} z^j$, except for $r_0(z) = c$. The number of components of a sub-polynomial is therefore always a power of 2. Each of the sub-polynomials can also be written as the product $r_i(z) = c\prod_{j=0}^{i-1}(z^{2^j}+1)$. Note that the sum has an upper limit of 2^i-1, whereas the upper limit of the product is only $i-1$. This means that evaluating the product runs in logarithmic time compared to evaluating the sum. Since $Q(z)$ has n components, the length and amount of the sub-polynomials is determined by the binary representation of $n = \sum_{i=0}^{k-1} b_i 2^i$, with $b_i \in \{0,1\}$ and $k = \lceil \log_2(n) \rceil$, which is the width of n in bits. In order for each sub-polynomial to have the correct degree, they get multiplied with $s_i(z) = \prod_{j=i+1}^{k-1} b_j z^{2^j}$, with $s_{k-1}(z) = 1$.

Therefore, $Q(z)$ can be rewritten by the binary representation of n as:

$$\sum_{i=0}^{k-1} b_i \cdot r_i(z) \cdot s_i(z) = \sum_{i=0}^{k-1} b_i \cdot \left(c\prod_{j=0}^{i-1}(z^{2^j}+1)\right) \cdot \left(\prod_{j=i+1}^{k-1} b_j z^{2^j}\right) \tag{4.5}$$

Referring to the listing 4.2, line 10 computes $r_i(z)$ and line 11 computes higher powers of z^{2^i}. Line 9 then iteratively calculates the sum by adding $r_i(z)$ if $b_i = 1$. Note also that the algorithm applies $s_i(z)$ through the distributive property in line 9 by multiplication with the variable `z`.

The way the algorithm works is best illustrated by an example. Suppose $n = 26$. First, $n = 2^4 + 2^3 + 2$ is written in binary, which is 11010_b. The following table shows the evaluation of $Q(z)$ by the algorithm, performing the steps for each i from left to right. Note, that $r_i(z)$ is actually precalculated for the next iteration, meaning in the first line with $i = 0$, r_{i+1} is calculated. $r_0(z)$ is initialised by setting it to c.

i	b_i	$Q(z)$	r_{i+1}	z
0	0	0	$c(z+1)$	z^2
1	1	$c(z+1)$	$c(z+1)(z^2+1)$	z^4
2	0	$c(z+1)$	$c(z+1)(z^2+1)(z^4+1)$	z^8
3	1	$c(z+1)z^8+c(z+1)(z^2+1)(z^4+1)$	$c(z+1)(z^2+1)(z^4+1)(z^8+1)$	z^{16}
4	1	$(c(z+1)z^8+c(z+1)(z^2+1)(z^4+1))z^{16}$ $+c(z+1)(z^2+1)(z^4+1)(z^8+1)$	(omitted)	z^{32}

Table 4.1: Evaluation of Algorithm C

Writing the result of the algorithm in polynomial form is, with c was omitted for clarity:

$$\underbrace{(z+1)(z^2+1)(z^4+1)(z^8+1)}_{1+z^1+\cdots+z^{15}}+\underbrace{(z+1)(z^2+1)(z^4+1)z^{16}}_{z^{16}+\cdots+z^{23}}+\underbrace{(z+1)z^8\cdot z^{16}}_{z^{24}+z^{25}} \tag{4.6}$$

Multiplying the polynomial with the constant c is simply done by initialising the variable `r` to c. Note also, that the algorithm determines b_i by initialising the variable `i` to n, which then gets shifted right by 1 at each iteration, and testing if the rightmost bit is 1. This is a small difference from the way the algorithm is explained.

4.2 User Interface

The user interface of the library is exposed via the header file `prand48.h`. Figure 4.1 gives a visual representation of it.

Using the library consists of four steps: Initialising the global state of the library, getting a copy of the state, jumping ahead in the stream and finally generating numbers from the sequence.

4.2.1 Global Initialisation

Before using pdrand, the global state needs to be initialised. There are four options to do so.

```c
void prand_init(void);

void prand_init48(uint_16 seed[3]);

void prand_init32(uint_32 seed);

void prand_init_man(uint_16 seed[3], uint_64 a, uint_16 c);
```

Listing 4.3: prand48.h - global initialisation routines

`prand_init` initialises the generator with default values for a, c, and the seed, taken from `rand48`. These are: $a = 25214903917$, $c = 11$, with the seed set to 20017429951246.

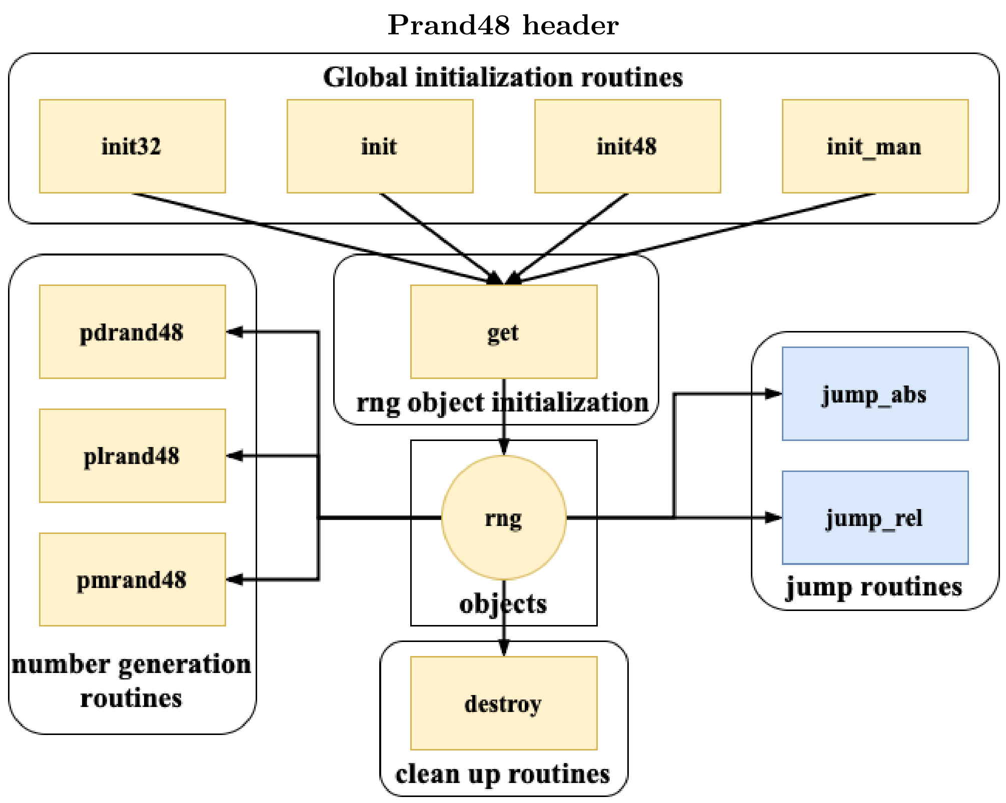

Figure 4.1: An overview of the header file of PRAND48. The color yellow categorizes routines/objects for random number generation. Blue denotes jump routines. Arrows are dependencies between functions, meaning in order to obtain an 'rng' object through a call to 'get', first, one of the global initialization routines has to be called.

`prand_init48` initialises the seed to a 48 bit number, stored in an array of 16 bit numbers of size three. It uses the default values for a and c. `prand_init32` initialises the upper 32 bits of the seed to the user provided 32 bit integer and sets the lower 16 bits to `0x330e`. `prand_init_man` lets the user manually specify the seed, a and c.

4.2.2 Obtaining a copy of the generator

After initialising the global state, the user can obtain a copy of it with the routine from listing 4.4.

```
Prand48* prand_get();

void prand_destroy(Prand48* prand);
```

Listing 4.4: prand48.h obtaining a copy of the rng state

`prand_get` returns a heap allocated pointer of the initial state of the generator. `prand_destroy` frees all the memory used by the `prand`.

4.2.3 Jumping ahead

Jumping ahead in the stream can be done with the routines shown in listing 4.5.

```c
void prand48_jump_abs(Prand48* prand, uint64_t n);

void prand48_jump_rel(Prand48* prand, uint64_t n);
```

Listing 4.5: prand48.h jump routines

`prand48_jump_abs` jumps ahead to the nth number in the sequence. This jump is absolute, meaning it starts from the beginning of the sequence. The result of the jump is stored in `prand`. Note that each thread needs to provide an unique instance of `prand`. `prand48_jump_rel` jumps ahead to the nth number in the sequence, relative to the current point in the sequence, stored in `prand`.

4.2.4 Number generation

Lastly, number generation is handled by the functions in listing 4.6.

```c
double pdrand(Prand48* prand);

uint32_t plrand(Prand48* prand);

int32_t pmrand(Prand48* prand);
```

Listing 4.6: prand48.h number generation

All of the functions take a pointer to a `Prand48` struct as input, which previously has to have been obtained with `prand48_get`. They then generate the next number and advance their state by one step. `pdrand` returns a double precision floating point number in the range of $[0.0, 1.0)$. `plrand` returns an integer in the range from 0 to $2^{31} - 1$. `pmrand` returns an integer in the range from -2^{31} to $2^{31} - 1$.

4.2.5 Example Usage

The following code listing 4.7 shows an example C program of how to use the library.

```c
#include "prand48.h"
#include <stdlib.h>

#define SIZE 10000ull

// forward declaration of some function to do some computation
void compute(unsigned long long len, double random_state[len]);

int main(void) {
    Prand48* rng;
    size_t jump_size = 10000;
    double random_state[SIZE];

    // initialise with default seed, do this once for a shared memory model
    prand48_init();
    // get a generator object
    rng = prand48_get();

    // perform an absolute jump
    prand48_jump_abs(rng, jump_size);

    // generate numbers to create a random state
    for (size_t i = 0; i < SIZE; ++i) {
        random_state[i] = pdrand48(rng);
    }

    // do some computation with the random_state
    compute(SIZE, random_state);

    return EXIT_SUCCESS;
}
```

Listing 4.7: Example usage of PRAND48

First, the global state is initialised with the default seed. Then the random number generator is obtained by calling `prand48_get`. Next, the random number generator performs an absolute jump. From that point on, a random state is created by storing random numbers in an array through a 'for' loop. Said array is then used to perform a computation.

4.3 Implementation

4.3.1 Global State

The global state is a struct, which stores the 48 bit seed X_0, and the values for the multiplier and increment.

```
struct PrandState {
    uint16_t seed[3];
    bool init;
    uint16_t c;     // increment
    uint64_t a;     // multiplier
}

static PrandState state = { 0 };
```

Listing 4.8: Global state

It also has a boolean flag, indicating if the state was already initialised. The state should be initialised only by one thread, if the runtime environment uses a shared data model and multiple threads are used. For a runtime using an distributed memory model like MPI, each process needs to initialise the global state separately. The global state is stored in the static variable `state`. The opaque pointer through which the user then interacts with the library is simply a wrapper around an 48 bit buffer:

```
struct Prand48 {
    uint16_t buf[3];
}
```

Listing 4.9: Prand48 struct

4.3.2 Jumping ahead

When `prand48_jump_abs` or `prand48_jump_rel` are called, they use the internal function `__jump_intern` to execute the jump. This function implements the algorithms described in listings 4.1 and 4.2. For the case that the increment is 0, only the right-to-left binary method is executed. The difference between performing an absolute jump and a relative jump simply is, that for an absolute jump, the seed stored in the global state is copied into the user provided buffer before executing the jump.

4.3.3 Outputting numbers

All of the routines generating random numbers, described in listing 4.6, use an internal function called `__prand_next`, which takes a pointer to `Prand48` as an input and evaluates the linear congruential equation, updating the current state stored in the pointer. Since the state is stored via an array of length three of 16 bit integers, the output functions which convert the 48 bit state into a signed and unsigned integer do this via bitwise 'or' and 'shift' operations.

Creating a floating point number is a bit more involved. Internally, in order to convert the the buffer to a float, a floating point number is represented via a union:

```
union IEEE754Double {
    double d;
    char b[sizeof(double)];
};
```

Listing 4.10: Internal floating point representation

This union represents a floating point number according to the IEEE754 double precision standard as bytes. It has to be noted that this is currently only implemented for little-endian machines.

The state buffer is then converted with the following routine:

```
void IEEE754Double_new(union IEEE754Double * n,
                       uint8_t sign, uint16_t exp, uint16_t mantissa[3]);

```

Listing 4.11: Floating point conversion routine

It takes all the parts of a floating point number, a sign, an exponent and a mantissa, and stores them into the argument `n`, which is the union from listing 4.10. Since `pdrand` returns floating point numbers in the range of $[0.0, 1.0)$, the sign is always set to 0, which indicates a positive number. The exponent is set to the value `0x3ff`, which is the binary representation of 2^0, for 11-bit excess-1023 numbers, where an exponent of 1023 represents the actual number 0. Since the mantissa of a double precision floating point number is 52 bits wide, the 48 upper bits of the mantissa are set to the state buffer and bits 49 through 52 are set to 0. After all the bytes have been written into the union, the floating point number from `pmrand` is returned via the double `d` field in the union.

Figure 4.2 shows a visualization of that process.

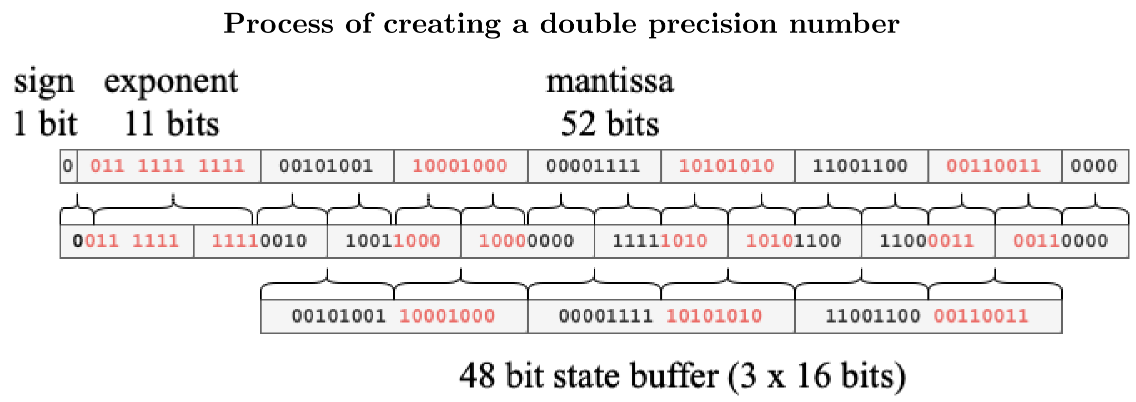

Figure 4.2: Visualization of how the 48 bit state buffer gets stored into a IEEE754 double precision floating point. The sign is always set to 0 or positive. The exponent is set to 2^0 in 11-bit excess-1023 encoding and the state buffer is stored in the 48 upper bits of the mantissa.

4.4 Performance and Benchmarks

The last section of this chapter assesses the performance of PRAND48. Two types of benchmarks were performed: Comparing the execution time of jumps to sequentially computing all numbers and testing the generator in regards to strong scaling.

4.4.1 Sequential computation versus jumping

The results of comparing the jump execution to calculating the actual sequence are shown in figure 4.3.

It can be seen, that jumping ahead is faster than sequential computation by several orders of magnitude, even for smaller jump sizes in the range of 10^1. The reason for this becomes clear when considering the time complexity of the two methods. 'Jumping' by computing numbers sequentially obviously runs in linear time with respect to the amount of numbers generated. On the other hand, when performing a jump, the two algorithms as described in the listings 4.1 and 4.2 are used, both of which have logarithmic time complexities. Therefore, executing a jump has a time complexity $O(\log n)$, meaning it increases linearly as the problem size doubles. Also, there are no additional pre-computations required when executing a jump.

Furthermore, since everything is computed internally with 64 bit unsigned integers, there is no need for any modulo operations, since the numbers are automatically wrapped with a modulo of 2^{64}, and get truncated to 48 bits for the output. It might also be a little

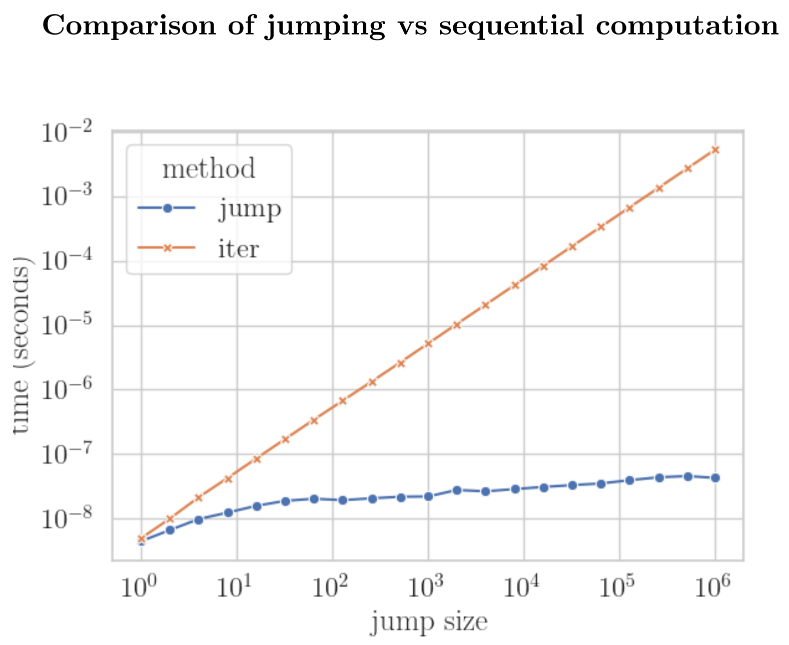

Figure 4.3: Comparison of the execution time between calculating numbers sequentially to jumping ahead in the stream. The line "iter" denotes sequential computation, "jump" jumping ahead in the stream.

bit faster to just store the internal state as a 64 bit unsigned integer, but since the library is intended to be a 'parallel' version of `rand48` its internal structure has been made to mimic that of `rand48`.

In a reference implementation though, it would be advised to just use 64 bit numbers, or even an arbitrary sizable array to create large period random number generators.

4.4.2 Strong scaling

This section measures the library in regards to strong scaling, meaning the speedup of the library for a constant problem size n. Here, the generation of n random numbers, with an increasing number of processes p is determined. The speedup was measured two times: The first one was in regards to small to medium problem sizes ranging from 128 to 10^6, with the number of processes ranging from 1 to 128. The second one measured large problem sizes ranging from 10^8 to 2.5×10^{10}, with process counts in the range of 1 to 65536.

Small to medium problem sizes

Figure 4.4 shows the measurements for small to medium problem sizes. Even for a tiny amount of 128 generated numbers, there is some form of speedup, up to about 15, with scaling stopping at about 50 processes. For problem sizes larger than 10^4, the library shows excellent scaling capabilities, with a speedup of almost 100 when running with 128 processes.

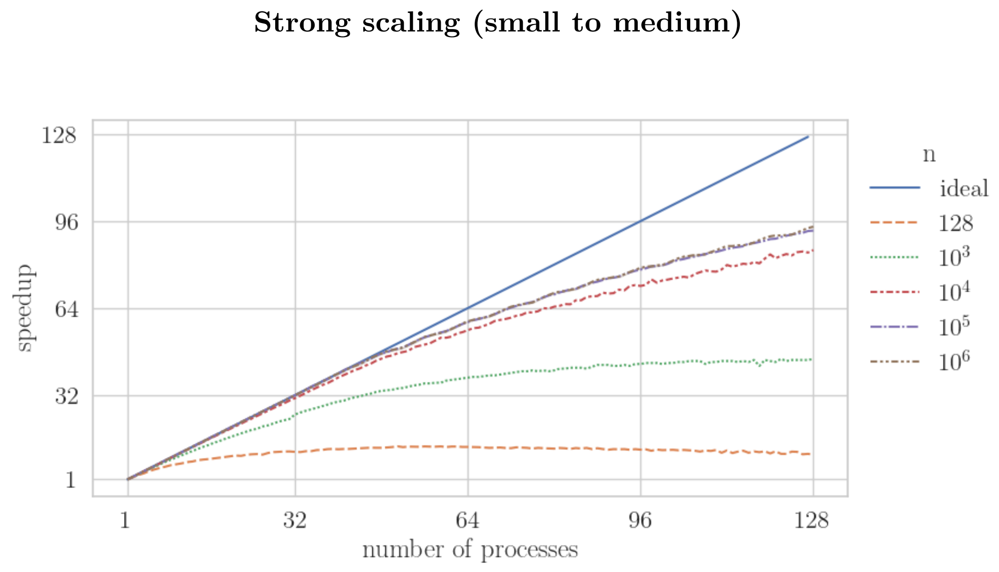

Figure 4.4: Strong scaling for small to medium problem sizes, when calculating n random numbers with 1 to 128 processes. The line "ideal" shows the ideal speedup, which would be linear to the number of processes.

Large problem sizes

The results for large problem sizes are shown in figure 4.5. Here, Prand48 demonstrates

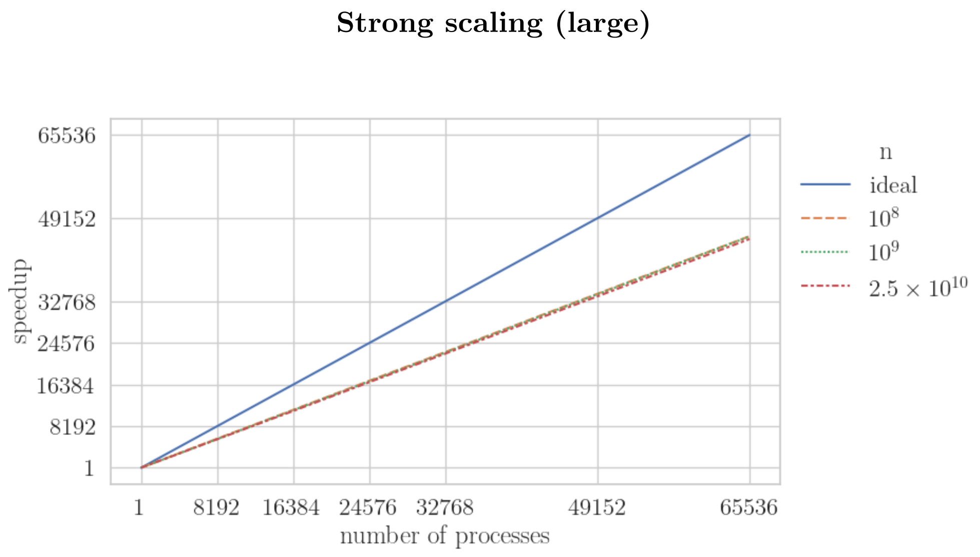

Figure 4.5: Strong scaling, when calculating n random numbers with 1 to 65536 processes. The line "ideal" shows the ideal speedup, which would be linear to the number of processes.

linear speedup, albeit with a smaller slope than the ideal speedup. Another observation is that there is a small dip in the line at approximately 500 processes. Nevertheless, for such large problem sizes, an LCG exhibits excellent scaling capabilities.

In general, jumping in a linear congruential generator gives an excellent performance when run in a parallel environment, making it a good choice of generator, if it meets a user's requirements for the properties of the generated random number stream.

Chapter 5

F2Lin

F2Lin is a random number library written in the C programming language based on $\mathbb{F}_2$-linear random number generators. F2Lin provides a generic interface, that can be implemented by any kind of random number generator of the above mentioned kind, which makes it easy to extend. Currently, the following random number generators are implemented:

- Xorshift, as described in section 2.2.2
- Mersenne Twister, a generator with a very large state size of 2^{19937}. It was presented by Makoto Matsumoto and Takuli Nishimura in 1998 in the paper [Nis98]. It is based on a mersenne prime, hence the name.
- Tiny Mersenne Twister, a smaller version of Mersenne Twister, with a state size of 2^{127}.
- Xoshiro256, with a state size of 2^{256}. It is based on bitwise xor, shifts and rotations. It was introduced by Sebastian Vigna in the paper "Scrambled Linear Pseudorandom Number Generators" [Bla18].

Since the implementation of this type of generator requires a lot of polynomial calculation, the first section 5.1 explains the reasons for the external library chosen, based on a performance comparison. Section 5.2 explains the algorithms and theory behind the implementation of the jump functionality, based on section 2.2.3. Next, in section 5.3, the user interface of the library is introduced. Section 5.4 shows the architecture of the library, with the implementation details in section 5.5. Afterwards, section 5.6 will give an example on how to add a new random number generator to F2Lin. Finally, section 5.7 discusses the performance of the library.

5.1 Polynomial arithmetic libraries

Since the random number generator requires polynomial arithmetic over $\mathbb{F}2$, and an own implementation would be prone to errors, it was decided to choose an external library to handle this part. Two libraries were considered: NTL found at https://libntl.org, which is a high performance library, implementing many mathematical functionalities. It is implemented in C++. The other one is called FLINT, a fast library for number theory, found at https://flintlib.org. It is written in C. For the choice of library, the implementation language was not as important as the performance.

As mentioned in 2.2.3, in principle two polynomials have to be determined: One is the minimal polynomial $p(x)$ of the transition matrix A. The other one is the jump polynomial $g(x) = x^v \mod p(x)$.

The minimal polynomial is determined by the *Berlecamp-Massey algorithm*, of which both NTL and FLINT provide an implementation. To apply the algorithm to a generator, a sequence of numbers two times the state size must be generated. Since the minimal polynomial of the transition matrix is the same as the minimal polynomial of the corresponding LFSR, it is sufficient to generate a sequence of bits [Pan07]. Listing 5.1 shows the implementation in pseudocode.

```
function min_poly(x):
    // find the minimal polynomial of the rng
    seq_len = 2 * state_size(x)
    seq[seq_len]
    for i in 0..seq_len:
        seq[i] = next_state(x) & 1 // extract the rightmost bit
                                   // from the next state
    return berlecamp_massey(seq)
```

Listing 5.1: Calculation of the minimal polynomial via Berlecamp-Massey

Next $g(z) = z^v \mod p(z)$ needs to be calculated, which can be done via modular right-to-left binary exponentiation, which is also implemented by both libraries.

To compare the performance of the two libraries, it was measured how long it takes to compute the minimal polynomial of each of the implemented generators, and to calculate the jump polynomial for various jump sizes. The benchmarks were performed as described in chapter 3. The results are displayed in the two figures 5.1 and 5.2.

In figure 5.1, showing the results for calculating $g(z)$, it can be seen, that NTL is faster than FLINT roughly by a factor of 10. One of the reasons for this might be that NTL has a separate module for polynomials over $GF(2)$, called GF2X. Another reason for NTL being faster, might be that it uses the library gf2x (capital letters are used to refer to the NTL class, and lowercase to the separate library), found at https://gitlab.inria.fr/gf2x/gf2x under the hood to do polynomial multiplication, which is needed to calculate the jump polynomial. gf2x implements refinements of several algorithms proposed in [Zim08], which improve the performance of polynomial multiplication.

Furthermore, when comparing the time needed to compute the minimal polynomial, as seen in figure 5.2, NTL is also faster than FLINT by about a factor of 10. The reasons why NTL is faster remain unclear, however, since both libraries implement the same algorithm.

Since NTL was faster in both cases, it was chosen as the external library for polynomial arithmetic over $GF(2)$.

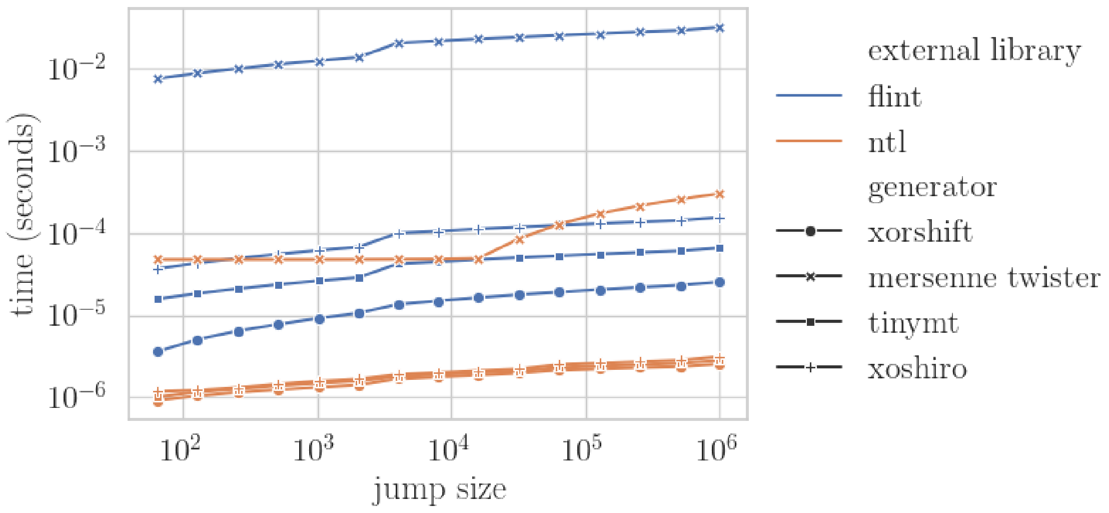

Figure 5.1: Time Comparison between flint and NTL in order to calculate the jump polynomial for different jump sizes and random number generators. The sizes of the jumps where in the range from 64 to 10^6, doubling the size at every step. Both axis use the logarithmic scale.

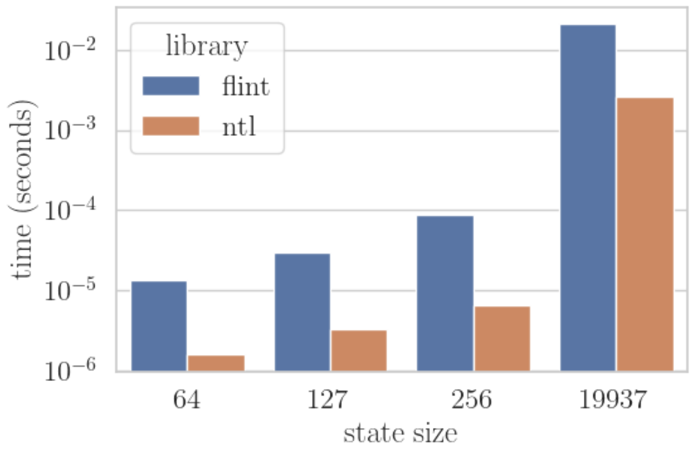

Figure 5.2: Time comparison between FLINT and NTL in order to calculate the characteristic polynomial of a random number generator. The state sizes correspond to Xorshift, TinyMT, Xoshiro and Mersenne Twister, in that order.

5.2 Theory and Algorithms

This section explains how to implement the jump functionality of the random number generators based on the background of section 2.2.

First, the evaluation of equation $g(\mathbf{A})\mathbf{x}$ from 2.14 with Horner's scheme is explained, then an improvement of this algorithm is presented, which requires fewer vector additions via polynomial decomposition and a sliding window method.

5.2.1 Evaluation with Horner's Scheme

Recall from section 2.2.3 that at the heart of the implementation, equation 2.14 has to be evaluated. One way to do this efficiently, is to apply Horner's scheme to the equation. For a polynomial of degree k, Horner's scheme requires exactly k multiplications and additions. It is as follows, applied to equation 2.14:

$$\begin{aligned}\mathbf{A}^v\mathbf{x} &= g(\mathbf{A})\mathbf{x} \\ &= (a_1\mathbf{A}^{k-1} + ... + a_{k-1}\mathbf{A} + a_k\mathbf{I})\mathbf{x} \\ &= \mathbf{A}(...(\mathbf{A}(\mathbf{A}(\mathbf{A}a_1\mathbf{x} + a_2\mathbf{x}) + a_3\mathbf{x}) + ... + a_{k-1}\mathbf{x}) + a_k\mathbf{x}\end{aligned} \tag{5.1}$$

The multiplication $\mathbf{Ax}$ can be calculated by advancing the generator's state $\mathbf{x}$ by one step. The addition can be calculated by a single binary 'xor' operation. The implementation is shown in listing 5.2.

```
algorithm horner(x, g):
    // x is the initial state, g the jump polyomial
    // coeff(i, g) is a function which returns the coefficient at x^i of g
    i = degree(g) // i is the coefficient at x^i
    tmp = x

    while i > 0:
        if coeff(i, g) == 1:
            tmp = tmp ^ x
        next_state(tmp)
        --i;

    if coeff(0, g) == 1:
        tmp = tmp ^ x

    return tmp;
```

Listing 5.2: Horner's scheme implementation

5.2.2 Sliding Window and Polynomial Decomposition

According to [L'E08] the amount of required 'xor' operations can be reduced by polynomial decomposition and a sliding window algorithm and gives the following explanation for it:

The main goal is to represent the jump polynomial $g(z)$ of degree k by several smaller polynomials $h_n(z) = z^q + b_1 z^{q-1} + ... + b_q$, $h_n(z) \in \mathbb{T}_q$. $\mathbb{T}_q$ is the set of polynomials with coefficients in $\mathbb{F}_2$ and polynomials exactly of degree q, where q is a small integer in the range of $3 - 10$. Therefore the cardinality of the set $\mathbb{T}_q$ is 2^q.

The decomposed form of $g(z)$ then is:

$$g(z) = h_1(z)z^{d_1} + h_2(z)z^{d_2} + \cdots + h_m(z)z^{d_m} + h_{m+1}(z) + z^q \tag{5.2}$$

The exponents d_j are calculated with: $d_j = k - q - i_j$. i_j is the index of the first non-zero coefficient a_{i_j} of $g(z)$, with $i_j > i_{j-1} + q + 1$, and $i_1 >= 0$. m is the last index for which $d_j = k - q - i_j > 0$. The last of the decomposition polynomials then is: $h_{m+1}(z) = z^q + a_{i_{m+1}} z^{k-i_{m+1}} + a_{1+i_{m+1}} z^{k-1-i_{m+1}} + \cdots + a_k$

Applying this to Horner's scheme is:

$$\begin{aligned} \mathbf{A}^v\mathbf{x} &= g(\mathbf{A})\mathbf{x} \\ &= \mathbf{A}^{d_m}(\ldots(\mathbf{A}^{d_2-d_3}(\mathbf{A}^{d_1-d_2}h_1(\mathbf{A})x + h_2(\mathbf{A})\mathbf{x}) + h_3(\mathbf{A})\mathbf{x}) + \cdots + h_m(\mathbf{A})\mathbf{x}) \\ &\quad + h_{m+1}(\mathbf{A})\mathbf{x} + \mathbf{A}^q\mathbf{x} \end{aligned} \tag{5.3}$$

The decomposition of a polynomial is best explained with an example. Suppose the coefficients of a polynomial of degree 15 are as follows:

$$\begin{array}{cccccccccccccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ 0 & \underbrace{1 \quad 0 \quad 0 \quad 1}_{h_1(z)} & & & & 0 & 0 & \underbrace{1 \quad 0 \quad 1 \quad 1}_{h_2(z)} & & & & 0 & 0 & 0 & \underbrace{1 \quad 1}_{h_3(z)+z^3} & \end{array} \tag{5.4}$$

Here a a_i denotes the coefficient of z^{k-i} in a polynomial of degree k, with the subpolynomials shown in brackets below the coefficients. For this is example q is set to 3. This means the decomposition polynomials h_n will be in the form of:

$$h(z) = z^3 + b_1 z^2 + b_2 z^1 + b_3 \tag{5.5}$$

First, the index i_1 of the first non-zero coefficient is determined. In the example, this is $i_1 = 1$. The resulting polynomial thus is $h_1(z) = z^3 + 1$, with $d_1 = 15 - 3 - 1 = 11$. Next, start looking for the index i_2 of the next non-zero coefficient from $i_1 + q + 1 = 5$. It is found at $i_2 = 7$ with $h_2(z) = z^3 + z + 1$ and $d_2 = 15 - 3 - 7 = 5$. Continuing at $i_2 + q + 1 = 11$, the index i_3 is 14. But since $d_3 = 15 - 3 - 14 <= 0$, m is set to 2. Therefore, $i_{m+1} = i_3 = 14$, $h_{m+1} = h_3$, and it is set to $h_3(z) = z^3 + z + 1$. When fully written out, it is:

$$\begin{aligned} g(z) &= z^{14}+z^{11}+ \quad z^8 + z^6+z^5+ \quad x+1 && (5.6) \\ &= h_1(z)z^{11}+ \quad h_2(z)z^5+ \quad h_3(z)+z^3 && (5.7) \\ &= (z^3+1)z^{11}+ \quad (z^3+z+1)z^5+ \quad (z^3+z+1)+z^3 && (5.8) \end{aligned}$$

Note, that $z^3 + z^3 = 0$ in $\mathbb{F}_2$, which is the reason for the addition with z^3 at the end of the equation.

Next, equation (5.3) is applied:

$$\begin{aligned} \mathbf{A}^v\mathbf{x} &= g(\mathbf{A})x \\ &= \mathbf{A}^{d_2}(\mathbf{A}^{d_1-d_2}h_1(\mathbf{A})x + h_2(\mathbf{A})x) + h_3(\mathbf{A})x + \mathbf{A}^q x \\ &= \mathbf{A}^5(\mathbf{A}^6 h_1(\mathbf{A})x + h_2(\mathbf{A})x)h_3(\mathbf{A})x + \mathbf{A}^3 x \end{aligned} \tag{5.9}$$

In the implementation, all the different vectors of each $h(A)x$ are precomputed for a given value of $\mathbf{x}$, the state of the random number generator before executing a jump. Also, since all the coefficients are in $\mathbb{F}_2$, they can be represented by a binary digit. In order to do this efficiently, [L'E08] suggests using gray code to represent each of the polynomials in $\mathbb{T}_q$. Gray code is a different ordering of the binary system, such that each consecutive number only differs by one bit. For example, to encode the numbers from 0 to 7, the corresponding gray codes are:

n	0	1	2	3	4	5	6	7
gray	000	001	011	010	110	111	101	100

Table 5.1: The first 8 gray codes

Next, all the elements of $\mathbb{T}_q$ are enumerated as $t_0(z), t_1(z), \ldots t_{2^q-1}(z)$, so that they correspond to their gray code enumeration. Looking at the table above, the first few elements of $\mathbb{T}_3$ would be: $t_0(z) = z^3, t_1(z) = z^3 + 1, t_2(z) = z^3 + z + 1$ and so forth. Also note, that the z^3 component is not encoded, since it is always included in $\mathbb{T}_3$.

Therefore, the first element is always set to $t_0(\mathbf{A})\mathbf{x} = \mathbf{A}^q\mathbf{x}$. The following elements can then be computed with a single binary 'xor' operation: Calculating $t_1(\mathbf{A})\mathbf{x}$ would be $t_1(\mathbf{A})\mathbf{x} = t_0(\mathbf{A})\mathbf{x} + \mathbf{A}\mathbf{x}$, and next, $t_2(\mathbf{A})\mathbf{x} = t_1(\mathbf{A})x + \mathbf{A}^2\mathbf{x}$ etc. It doesn't matter if $\mathbf{A}^n\mathbf{x}$ needs to be added or subtracted, since these are identical operations in $\mathbb{F}_2$.

Additionally, $\mathbf{A}^1, \mathbf{A}^2, \ldots \mathbf{A}^q$ need to be computed, which can be done by advancing the generator q steps.

The pseudocode in listing 5.3 demonstrates how the precomputation would be implemented.

```
algorithm precompute(q, x, gray):
    // x is the current rng state,
    // gray is an indirection array, to map a number n
    // to its gray code

    A[q + 1]        // array of size q + 1 holding A^0 * x to A^q * x
    y[2^q - 1]      // array of size 2^q - 1, storing t_n(A)x

    for i in 0 to q:
        A[i] = x
        next_state(x)

    y[0] = A[q]      // set t_0(A)x = A^q * x

    for i in 1 to 2^q:
        // determine the bit, which flips in the next gray code
        flipped_bit = 0
        diff = gray[i] ^ gray[i - 1]

        // this loop implements log2(n)
        while (diff > 1):
            diff >>= 1
            flipped_bit += 1

        y[gray[i]] = y[gray[i - 1]] ^ A[flipped_bit]

    return y
```

Listing 5.3: Precompute sliding window

After precomputing the values needed to calculate the jump, the decomposition polynomials $h_1(z)$ to h_{m+1} and the values for d_1 to d_m and m need to be determined. In the implementation, this is handled by the functions in `poly_decomp.h`, which will be explained in the next section.

In order to create the decomposition of the polynomial the following algorithm is used. Since arrays are usually zero indexed, the polynomials of $h_n(z)$ start at index 0 in the implementation.

```
algorithm decompose_g(g, q, h, d):
    // q is the degree of the smaller polynomials
    // h is an array holding the coefficients of the smaller polynomials
    // encoded in a binary digit
    // d holds the exponents needed to step the algorithm forward
    i = deg(d) // i holds the current coefficient of g
    m = 0

    while i >= q:
        if coeff(g, i) == 0:
            continue
        h[m] = determine_sub_poly(q, i, g)
        i -= q
        d[m] = i
        m += 1
        i -= 1

    h[m] = determine_sub_poly(i + 1, i + 1, g)
    return h, d, m - 1
```

Listing 5.4: Polynomial decomposition

The algorithm `determine_sub_poly` determines the coefficients of the sub polynomials. It is shown in listing 5.5

```
algorithm determine_sub_poly(q, i, g):
    // encodes the sub polynomial starting at the x^i coefficient
    // this is done with bit-shifts and xoring
    sub_poly = 0
    for j in 0 to q:
        sub_poly = (sub_poly << 1) ^ coeff(g, i - j - 1)
    return sub_poly
```

Listing 5.5: Determine sub-polynomial

The last step that remains is to evaluate Horner's scheme, similar to listing 5.2, with some slight modifications.

```
algorithm horner_sw(h, d, m, y):
    // h and d are arrays of length m + 1,
    // containing the decomposed jump polynomial
    // y are the precomputed polynomyals
    // x is the current state of the rng
    tmp = h[0]

    for i in 1 to m + 1:
        for j in 0 to d[i - 1] - d[i]:
            next_state(tmp) // calculate A^(di-1 - di)
        tmp = tmp ^ y[h[i]]

    for j in 0 to d[m]:
        next_state(tmp)

    return tmp ^ h[m + 1] ^ h[0]
```

Listing 5.6: Horner's scheme with sliding window

One thing to note, is that the decomposition can also be done on the fly. Choosing the jump algorithm and value for q depends on the size of the jump polynomial, which will be discussed in the performance evaluation in 5.7.

5.3 User Interface

This section gives an overview of the functionality exposed by `f2lin.h`. There are two main objects for using the random number generator: A pointer to the generator itself, called `F2LinRngGeneric` and a pointer to the jump polynomial, called `F2LinJump`.

Figure 5.3 gives a visual representation of the user interface.

5.3.1 Random Number Generator Routines

The routines for the actual generator are are shown in listing 5.7.

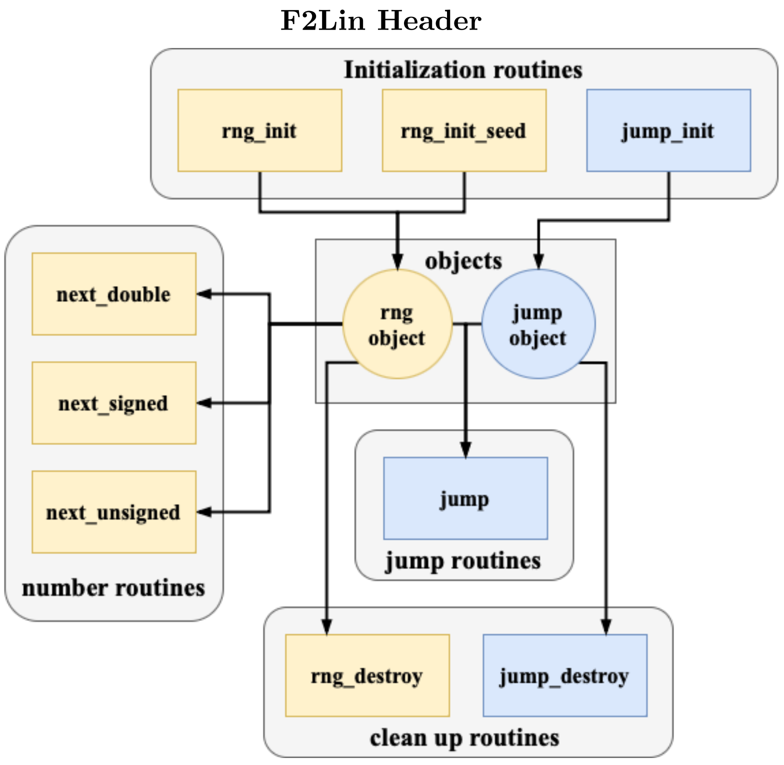

Figure 5.3: An overview of the header file exposing the user interface of F2Lin. The color yellow denotes routines/objects, which belong to random number generation, while blue denotes routines/objects belonging to jumping. Arrows are dependencies, for example, in order to obtain an rng_object, one of the initialization routines has to be called first.

```c
F2LinRngGeneric* f2lin_rng_init();
F2LinRngGeneric* f2lin_rng_init_seed(const uint64_t seed);
void f2lin_rng_destroy(F2LinRngGeneric* rng);
uint64_t f2lin_next_unsigned(F2LinRngGeneric* rng);
int64_t f2lin_next_signed(F2LinRngGeneric* rng);
double f2lin_next_double(F2LinRngGeneric* rng);
```

Listing 5.7: f2lin.h random number generator routines

To get a pointer to `F2LinRngGeneric` the user can either call `f2lin_rng_init`, which initialises the random number generator with a predefined seed, or `f2lin_rng_init_seed`, where the user can provide a seed in the form of a 64-bit unsigned integer. Since the pointer to the random number generator is heap-allocated, each one must be deallocated by calling `f2lin_rng_destroy` when it is no longer needed, to prevent memory leaks. The three functions `f2lin_next_unsigned`, `f2lin_next_signed` and `f2lin_next_double` are used in order to generate random numbers.

- `f2lin_next_unsigned` creates an unsigned integer in the range from 0 to $2^{64} - 1$.
- `f2lin_next_signed` calculates a signed integer in the range from -2^{32} to $2^{31} - 1$.
- `f2lin_next_double` creates a real number in the range of $[0 - 1)$

5.3.2 Jump Configuration routines

The functions described next are used to initialize the jump polynomial:

```c
F2LinJump* f2lin_jump_init(const size_t jump_size, F2LinConfig* cfg);
void f2lin_jump_destroy(F2LinJump* jump);
```

Listing 5.8: f2lin.h jump initialization and deallocation routines

`f2lin_jump_init` is used to initialise a new jump polynomial. This must be done separately for each jump size. Also, initialising a jump is independent of the current state of the random number generator, meaning a single `F2LinJump` can be used for multiple `F2LinRngGenerics` to perform a jump. The parameters can also be configured by providing a pointer to a `F2LinConfig`. `F2LinConfig` is a struct, found in `config.h` with the following fields:

1. `jump_algorithm: enum JumpAlgorithm`: The algorithm used to calculate the jump. There are three variants for this:

   `HORNER`, `SLIDING_WINDOW` and `SLIDING_WINDOW_DECOMPOSITION`. `HORNER` evaluates the jump polynomial using Horner's scheme. `SLIDING_WINDOW` uses the sliding window algorithm, where the decomposition of the jump polynomial is done on the fly. `SLIDING_WINDOW_DECOMP` precomputes the decomposition polynomial for the sliding window algorithm. Which version of the algorithms to use is discussed in section 5.7.

2. `q: size_t` The degree of the decomposition polynomials, when using `SLIDING_WINDOW` or `SLIDING_WINDOW_DECOMP`. Needs to be in the range from $1 - 10$.

To use defaults, `cfg` can be set to `0`. The default values set the jump algorithm to `SLIDING_WINDOW_DECOMP` with a q value of 6.

When the jump parameters are no longer needed, they can be destroyed by calling `f2lin_jump_destroy` to prevent memory leaks.

5.3.3 Jump routine

To jump ahead in the random number stream, the header exposes the routine from listing 5.9.

```c
void f2lin_jump(F2LinRngGeneric* rng, F2LinJump* jump)
```

Listing 5.9: f2lin.h jump routine

A call to this function advances the state of the random number generator `rng` by the number of steps specified in the initialisation of `jump`.

5.3.4 Example Usage

Listing 5.10 shows an example C program using the library. It initialises the random number generator and jump, then jumps forward in the generated stream. After jumping, it generates numbers which are stored in an array to set up a random state, which can then be used to do some calculations.

```c
#include "f2lin.h"
#include <stdlib.h>

#define SIZE 1000ull

// forward declaration some function to do some computation
void compute(unsigned long long len, double random_state[len]);

int main(void) {
    double random_state[SIZE];
    size_t jump_size = 10000000;

    // Custom configuration with sliding window algorithm with a q value of 4
    F2LinConfig cfg = { .q = 4, .algorithm = SLIDING_WINDOW };
    // initialize the rng with an own seed
    F2LinRngGeneric* rng = f2lin_rng_init_seed(123456ull);
    // initialize the jump with a custom configuration
    F2LinJump* = f2lin_jump_init(jump_size, &cfg);

    // perform the jump
    f2lin_jump(rng, jump);

    // generate numbers to create some random state
    for (size_t i = 0; i < SIZE; ++i) {
        random_state[i] = f2lin_next_double(rng);
    }

    // do some computation with the random state
    compute(SIZE, random_state);

    return EXIT_SUCCESS;
}
```

Listing 5.10: Example usage of F2Lin

5.4 Architecture

Figure 5.4 shows a diagram of the library's architecture.

The library is divided into four layers. The first layer (from the top) is the user interface, explained in the previous section. The next two layers represent the internal parts of the library. The second layer represents components accessed by the user interface, with some of its objects exposed to the user through opaque pointers. The third layer consists of components that are only needed internally and are never exposed to the user. The fourth layer represents additional components: One is NTL, the other, minpoly is used to compute minimal polynomials for a given generator, usually at compile time.

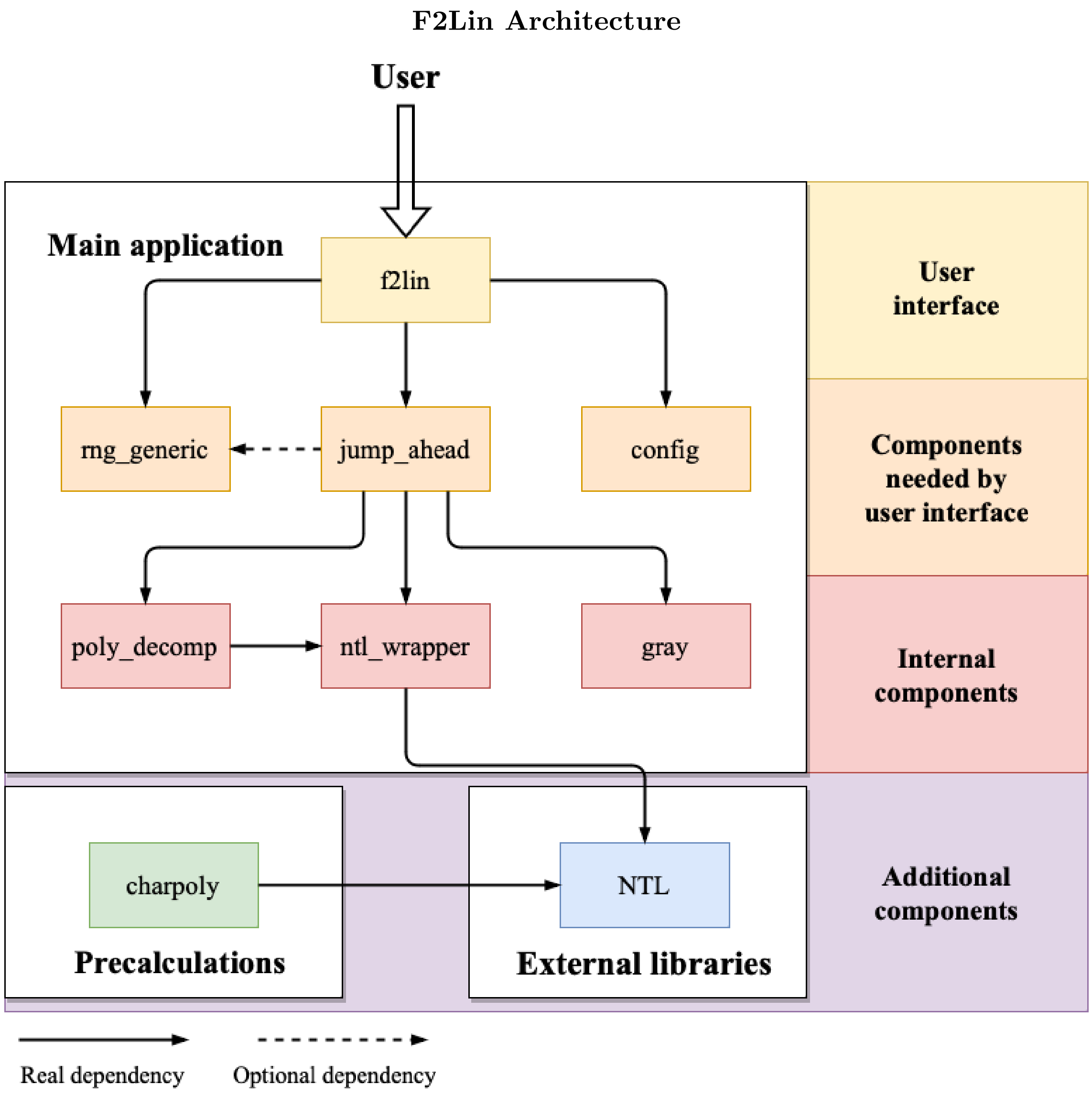

Figure 5.4: An overview of the architecture of F2Lin, showing all of its components. An arrow from component A to B means, B is a dependency of A. The architecture is divided into four layers, with the components of the lowest layer not being part of the actual library itself.

5.5 Implementation

5.5.1 jump_ahead

The component `jump_ahead` consists of the header file `jump_ahead.h` and the implementation file `jump_ahead.c`. As there are three algorithms implemented for calculating the jump, the necessary parameters for each algorithm are stored via a tagged union:

```
struct F2LinJump {
    enum F2LinJumpAlgorithm algorithm;
    union F2LinJumpPoly jp;
};
```

Listing 5.11: jump_ahead.h struct F2LinJump tagged union

The tag is given by the enum `F2LinJumAlgorithm`, which is declared in `config.h` (see 5.5.6 for details). The union `F2LinJumpPoly` is implemented as follows:

```
union F2LinJumpPoly {
    GF2X* horner;
    F2LinJumpSW sw;
    F2LinJumpSWD swd;
};
```

Listing 5.12: jump_ahead.h union F2LinJumpPoly

The field `horner` stores only the jump polynomial, as this is the only thing needed to evaluate Horner's Scheme.

The field `sw` stores the data needed to implement the sliding window algorithm, which is stored in the struct `F2LinJumpSW`:

```
struct F2LinJumpSW {
    int q;
    F2LinRngGeneric** y;
    GF2X* jp;
};
```

Listing 5.13: jump_ahead.h struct F2LinJumpSW

To evaluate via the sliding window algorithm the following fields are required: The degree of the decomposition polynomials `q`, the values for the state of the random number generator `y`, after multiplying it with $t(\mathbf{A})$ for each $t(z) \in \mathbb{T}_q$, and the jump polynomial $g(z)$ itself, called `jp`. This version of the sliding window algorithm computes the decomposition polynomials on the fly, so the jump polynomial is stored in its raw form.

Finally, the field `swd` stores all the parameters needed to evaluate the sliding window algorithm, but this time with the precomputed decomposition polynomial `pd`. Otherwise, it is the same as `F2LinJumpSW`:

```
struct F2LinJumpSWD {
    int q;
    F2LinRngGeneric** y;
    F2LinPolyDecomp* pd;
};
```

Listing 5.14: jump_ahead.h struct F2LinJumpSWD

The header file also exposes three functions:

```
F2LinJump* f2lin_jump_ahead_init(const size_t jump_size, F2LinConfig* c);

F2LinRngGeneric* f2lin_jump_ahead_jump(F2LinJump* jump_params,
                                     F2LinRngGeneric* rng);

void f2lin_jump_ahead_destroy(F2LinJump* jump_params);
```

Listing 5.15: jump_ahead.h functions

`f2lin_jump_ahead_init` initialises a jump for the given jump size `jump_size`. It also takes an optional pointer to the configuration. It first validates the given configuration, then calculates the jump polynomial $p(z)$. It then checks whether the sliding window algorithm is being used, and allocates all the space necessary to store the parameters required for it. Finally, if the algorithm is set to `SLIDING_WINDOW_DECOMP`, it also initialises the decomposition polynomial according to listing 5.4. `f2lin_jump_ahead_destroy` deallocates all the parameters used in `F2LinJump`.

`f2lin_jump_ahead_jump` jumps ahead the state of the random number generator `rng`. This is implemented as follows, see also figure 5.5:

First, it checks which jump algorithm should be used. If it is `HORNER`, the function `horner` is called, which implements Horner's scheme according to listing 5.2. Otherwise, the sliding window algorithm must first be initialised as described in listing 5.3. Then, depending on whether the jump algorithm is set to `SLIDING_WINDOW` or `SLIDING_WINDOW_DECOMP`, `horner_sliding_window` or `horner_sliding_window_decomp` is called.

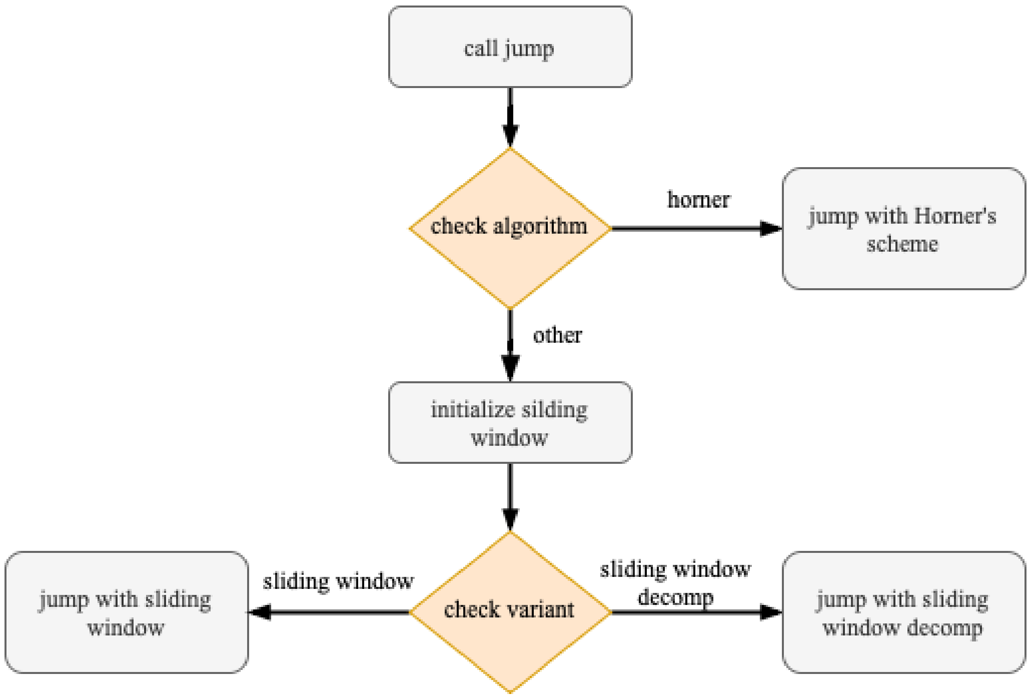

Figure 5.5: An overview of how a jump gets executed.

5.5.2 gray

This component, which consists of the single header file `gray.h`, statically stores the first 1024 gray values in an array called `GRAY`, which are needed when computing the decomposition polynomials. Since the maximum value of `q` is 10, there is no need to store gray values for higher numbers. It also exposes the following function:

```c
static inline
size_t f2lin_determine_gray_enumeration(const size_t q, const size_t i,
                                        const GF2X* poly);
```

Listing 5.16: gray.h functions

It implements the algorithm described in listing 5.5, which finds the subpolynomial of length $q-1$, starting from the coefficient at z^{i-1} in the argument `poly`.

5.5.3 minpoly

The component `minpoly` is implemented by a `c++` program called `minpoly.h`. It is not an actual part of the application itself, but is used to create a header file containing the minimal polynomial $g(z)$ of a random number generator. Since the calculation of the minimal polynomial is quite CPU intensive (see the benchmarks in section 5.1) and only needs to be done once for a given generator, it is best to calculate it in advance and include it as a string in a header file in the implementation of a given generator. Then it can be read in at runtime when calculating the jump polynomial.

5.5.4 poly_decomp

`poly_decomp` consists of `poly_decomp.h` and `poly_decomp.c`. It implements the functionality to decompose the jump polynomial $p(z)$ into smaller sub-polynomials $h(z)$, which are then stored in a struct called `F2LinPolyDecomp`. It is only needed when the algorithm is set to `SLIDING_WINDOW_DECOMP`. The implementation is shown in listing 5.17.

```c
struct F2LinPolyDecomp {
    uint16_t* h;
    size_t* d;
    size_t m;
    uint16_t hm1;
    size_t cap;
};
```

Listing 5.17: poly_decomp.h representation of a decomposed polynomial

It stores all the components needed to apply the sliding window algorithm as described in section 5.2.2. One small caveat is that the subpolynomial h_{m+1} is stored in a separate field `hm1`. The field `cap` is the capacity of the arrays `h` and `d`, set to $\lfloor \frac{k}{q} \rfloor$, where k is the degree of the jump polynomial. The field `m` is the length of the said arrays. Note, that compared to the previous explanation in section 5.2.2, the indices of h_n start at 0 instead of 1, since C arrays are zero indexed. The header then exposes the following two functions:

```c
F2LinPolyDecomp* f2lin_poly_decomp_init_from_gf2x(const GF2X* jump_poly,
                                                  const int q);

void f2lin_poly_decomp_destroy(F2LinPolyDecomp* decomp_poly);
```

Listing 5.18: poly_decomp.h routines

- `f2lin_poly_decomp_init_from_gf2x` returns a heap-allocated pointer to the decomposition polynomial. It does this as described in listing 5.4.
- `f2lin_poly_decomp_destroy` frees all the memory used by the decomposition polynomial.

5.5.5 rng_generic

As mentioned at the beginning of chapter 2, F2Lin is designed to be easily extended with other random number generators. This is done by implementing the header file `rng_generic.h` from listing 5.19.

```
1  typedef struct F2LinRngGeneric F2LinRngGeneric;
2  
3  F2LinRngGeneric* f2lin_rng_generic_init_zero();
4  
5  F2LinRngGeneric* f2lin_rng_generic_init();
6  
7  F2LinRngGeneric* f2lin_rng_generic_init_seed(uint64_t seed);
8  
9  void f2lin_rng_generic_destroy(F2LinRngGeneric* rng);
10 
11 F2LinRngGeneric* f2lin_rng_generic_copy(F2LinRngGeneric* dest,
12                                         const F2LinRngGeneric* source);
13 
14 F2LinRngGeneric* f2lin_rng_generic_add(F2LinRngGeneric* lhs,
15                                        const F2LinRngGeneric* rhs);
16 
17 uint64_t f2lin_rng_generic_gen64(F2LinRngGeneric* rng);
18 
19 uint64_t f2lin_rng_generic_next_state(F2LinRngGeneric* rng);
20 
21 long f2lin_rng_generic_state_size();
22 
23 void f2lin_rng_generic_gen_n_numbers(F2LinRngGeneric* rng, size_t N,
24                                      uint64_t buf[N]);
25 
26 int f2lin_rng_generic_compare_state(F2LinRngGeneric* lhs, F2LinRngGeneric* rhs);
27 
28 #ifndef CALC_MIN_POLY
29 char* f2lin_rng_generic_min_poly();
30 #endif
```

Listing 5.19: rng_generic.h

Line 1 of the listing 5.19 is the forward declaration of the struct `F2LinRngGeneric`, which is the state of a random number generator.

- `f2lin_rng_generic_init_zero` returns a pointer to a random number generator whose state is initialised to zero.
- `f2lin_rng_generic_init` returns a pointer to a random number generator with its state initialised to a predefined seed.
- `f2lin_rng_generic_init_seed` initialises a random number generator with a seed given by the user, stored as a 64-bit unsigned integer.
- `f2lin_rng_generic_destroy` deallocates a random number generator.
- `f2lin_rng_generic_copy` copies the state of the generator `source` to `dest`.
- `f2lin_rng_generic_add` adds the state of `rhs` to `lhs` and stores the result in `lhs`.
- `f2lin_rng_generic_gen64` generates a 64-bit unsigned integer random number. From a more theoretical point of view, this function combines the state transformation matrix A with the output transformation matrix B, as shown in equations 2.3 and 2.4 of section 2.2.1 of chapter 2.
- `f2lin_rng_generic_next_state` advances the state of the random number generator by applying the state transformation matrix A.
- `f2lin_rng_generic_state_size` returns the state size of the random number generator in bits.
- `f2lin_rng_generic_gen_n_numbers` generates `N` numbers at once and stores them in the user supplied buffer `buf`.
- `f2lin_rng_generic_compare_state` compares the state of lhs and rhs, and returns a non-zero value if they are equal and 0 if they are not. This function is mainly used to test if a random number generator is implemented correctly.
- `f2lin_rng_generic_min_poly` returns the minimal polynomial of the generator, if `CALC_MIN_POLY` is not defined. Usually `CALC_MIN_POLY` is set when calculating the minimal polynomial of a random number generator by running `minpoly.cpp` (5.5.3).

In general, all random number generators implementing the header should advance their state by 64 bits at a time.

5.5.6 config

This component consists of the header file `config.h`, shown in listing 5.20. It contains the definitions of the available jump algorithms, and the struct containing the configuration, which is used when initialising a jump with `f2lin_jump_init` (5.5.1).

```
#define Q_MAX 10
#define Q_MIN 1
#define Q_DEFAULT 6
```

```c
#define ALGORITHM_DEFAULT SLIDING_WINDOW_DECOMP

enum F2LinJumpAlgorithm {
    HORNER = 0, SLIDING_WINDOW = 1, SLIDING_WINDOW_DECOMP = 2,
};

typedef struct F2LinConfig F2LinConfig;
struct F2LinConfig {
    enum F2LinJumpAlgorithm algorithm;
    int q;
};
```

Listing 5.20: config.h

As mentioned earlier, the field `algorithm` in `F2LinConfig` specifies the jump algorithm, and `q` specifies the size of the sub-polynomials when using the sliding window algorithm.

5.5.7 gf2x_wrapper

gf2x_wrapper is a small `C` wrapper around NTL. Since only functionality from NTL's `GF2X` class is needed, it only wraps around that. The reason for writing a wrapper rather than using the library itself was that F2Lin should be written in the `C` programming language.

5.6 Extending F2Lin

This section gives an example of how to add a new random number generator to F2Lin.

The generator implemented here is Xorshift, as briefly described at the beginning of this chapter. The generator will have a state size of 64 bits and the implementation file will be called `rng_generic64.c`.

First we implement the struct `F2LinRngGeneric`, which wraps around a 64-bit number:

```c
struct F2LinRngGeneric {
    uint64_t state;
};
```

Listing 5.21: struct F2LinRngGeneric

Next, the initialisation and deallocation functions are implemented:

```c
F2LinRngGeneric* f2lin_rng_generic_init_zero() {
    return (F2LinRngGeneric*) calloc(1, sizeof(F2LinRngGeneric));
}

F2LinRngGeneric* f2lin_rng_generic_init() {
    F2LinRngGeneric* rng = (F2LinRngGeneric* )calloc(1, sizeof(F2LinRngGeneric));
    rng->state = 12323456ull;
    return rng;
}

```

```c
F2LinRngGeneric* f2lin_rng_generic_init_seed(uint64_t seed) {
    F2LinRngGeneric* rng = f2lin_rng_generic_init_zero();
    rng->state = seed;
    return rng;
}

void f2lin_rng_generic_destroy(F2LinRngGeneric* rng) {
    free(rng);
}
```

Listing 5.22: F2LinRngGeneric initialization and deallocation

The copy and add functions are also easy to implement, as only a single bitwise 'xor' operation is required to add two states:

```c
F2LinRngGeneric* f2lin_rng_generic_copy(F2LinRngGeneric *dest,
        const F2LinRngGeneric *source) {
    dest->state = source->state;
    return dest;
}

F2LinRngGeneric* f2lin_rng_generic_add(F2LinRngGeneric* lhs,
        const F2LinRngGeneric* rhs) {
    lhs->state ^= rhs->state;
    return lhs;
}
```

Listing 5.23: F2linRngGeneric copying and adding

Then, the state transformation and output functions are implemented, as well as the generation of a batch of numbers. Since this is a very simple generator, there is no output transformation. See section 2.2.2 for more information on choosing shift values.

```c
uint64_t f2lin_rng_generic_next_state(F2LinRngGeneric* rng) {
    rng->state ^= (rng->state << 13);
    rng->state ^= (rng->state >> 17);
    rng->state ^= (rng->state << 5);
    return rng->state;
}

uint64_t f2lin_rng_generic_gen64(F2LinRngGeneric* rng) {
    return f2lin_rng_generic_next_state(rng);
}

void f2lin_reng_generic_gen_n_numbers(F2LinRngGeneric* rng, size_t N,
                                      uint64_t buf[N]) {
    for (size_t = i; i < N; ++i) {
        buf[i] = f2lin_rng_generic_gen64(rng);
    }
}
```

Listing 5.24: F2LinRngGeneric state transformation and output

The last function to implement is `f2lin_rng_generic_state_size` which simply returns the integer 64.

This completes the basic implementation of the random number generator. All that remains is to compute its minimal polynomial $p(z)$ by running the program `minpoly.cpp`. First, the implementation file of the generator must be prepared by including the header to be generated and the function that returns $p(z)$:

```
#ifndef CALC_MIN_POLY
#include "minpoly64.h"
#endif
.
.
.
#ifndef CALC_MIN_POLY
char* f2lin_rng_generic_min_poly() {
    return MIN_POLY;
}
#endif
```

Listing 5.25: Minimal polynomial implementation

These functions are wrapped in a preprocessor directive, since the implementation needs to be bootstrapped by calculating $p(z)$, which would not be possible otherwise.

The generated header file will be of the form:

```
#define MIN_POLY = "101....01"
```

Listing 5.26: minpoly.h

Next `minpoly.cpp` is compiled with the generator's implementation (assuming that all files are in the same directory):

```
$ g++ -O3 -std=c++11 minpoly.cpp rng_generic64.c -o minpoly64 \
      -lntl -lgmp -lgf2x
$ ./minpoly64
```

Listing 5.27: Generating the header file for a generators minimal polynomial

Then, the program that creates the header file `minpoly.h` is executed. The only thing left to do is to rename it to `minpoly64.h`.

Now the library can be built with the new generator by including its translation unit during the build process. One thing to note is that in a production implementation, the creation and renaming of the header file for a generator would be done automatically in a build script.

5.7 Performance and Benchmarks

This last section evaluates the performance of F2Lin. Three types of benchmarks have been run on the library: First the performance of the initialisation routines of the decomposition polynomials was measured and a comparison of the three jump algorithms was made. Second, it measured the performance of jumping ahead in the stream compared to calculating everything sequentially. Finally, the library was analyzed in respect to strong scaling.

5.7.1 Jump Algorithms

This benchmark measured the performance of the three different jumping algorithms, Horner, Sliding Window and Sliding Window with the Decomposition Polynomial, for each of the four random number generators.

It was measured how long it took to evaluate a polynomial with degree k, ranging from $s_{\text{rng}}/8$ to $s_{\text{rng}} - 1$, where s_{rng} is the state size of a random number generator. Since the degree k of the jump polynomial $g(z)$ is bounded by the degree l of the minimal polynomial $p(z)$ of a random number generator through the modulo operation, it can never be greater than $l - 1$. The polynomials themselves were generated randomly, but were the same for each algorithm and generator. In addition, for both sliding window variants, it was measured how they perform with different values of q, ranging from 2 to 10. $q = 1$ was not tested, as it basically makes the algorithm behave as if it were just evaluating Horner's scheme. Next, the two sliding window algorithms were compared for values of q ranging from 4 to 8.

First, it is necessary to determine which values of q result in good performance for a random number generator. This is done by looking only at the data from the sliding window decomposition algorithm. According to [L'E08], the main goal of the sliding window algorithm is to reduce the number of bitwise 'xor' operations needed to evaluate the jump polynomial. The cost of this operation depends on the state size of a random number generator, so it is important to evaluate it for each of them.

It is to sufficient check this only for the sliding window with decomposition variant, since both algorithms showed exactly the same behaviour for each value of q in preliminary benchmarks. The results of this are shown in figure 5.6.

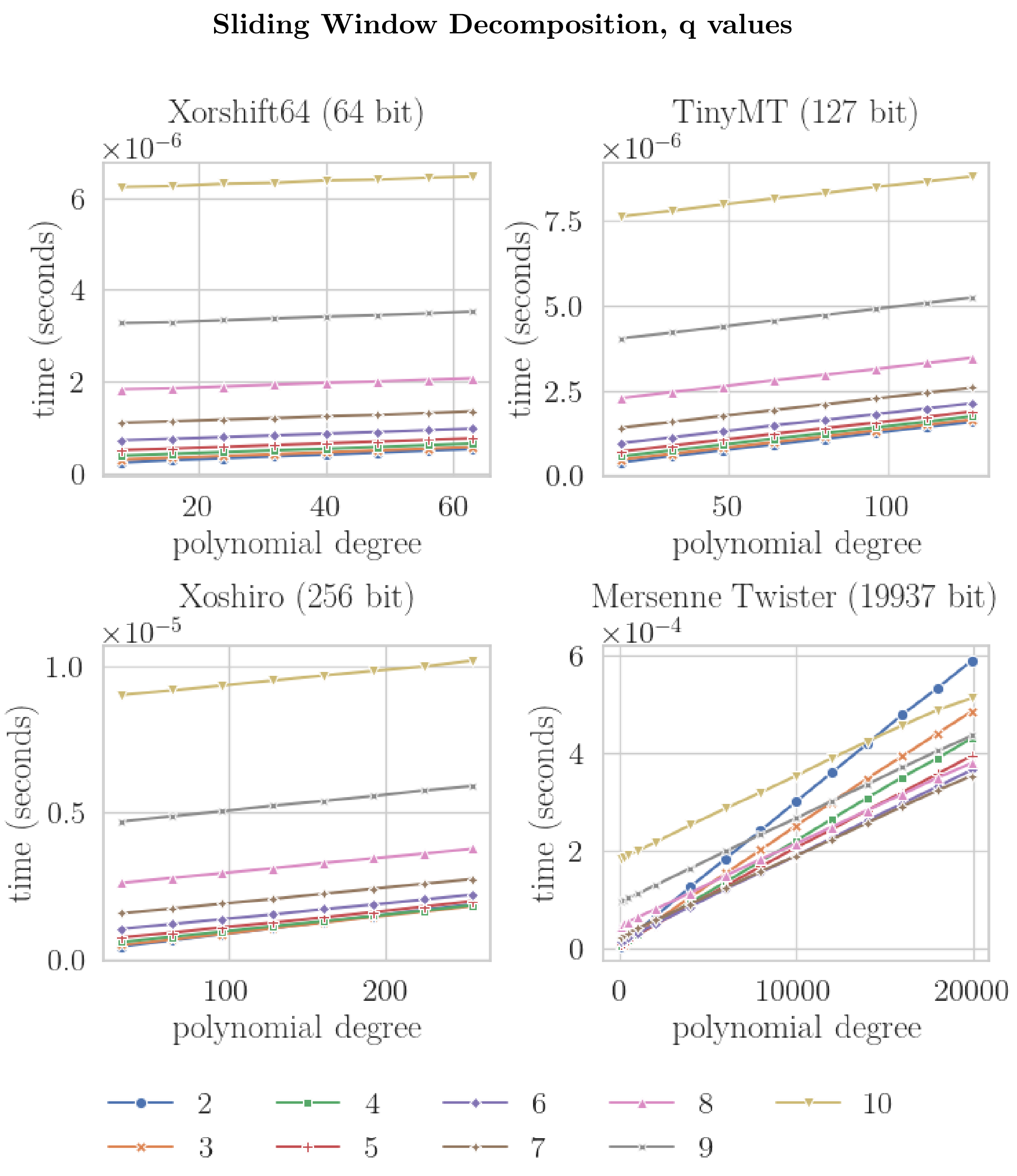

Figure 5.6: Time comparison of the Sliding Window decomposition algorithm, for values of q (the degree of the sub polynomials), ranging from 2 to 10, for each of the four random number generators .

It is immediately apparent from the results that the influence of each value for q is only really noticeable for large state random number generators. Therefore, the following discussion will be limited to the results of Mersenne Twister. While for degrees k $k < 1000$ the values $q < 8$ outperform the higher values, from about $k = 2000$ the smaller values $q < 4$ start to have a steeper slope, compared to values of $q > 4$. Two excellent default candidates seem to be $q = 7$ or $q = 6$, which perform very well for larger polynomials with $k >= 6000$ and are comparatively fast for smaller polynomials with $k < 2000$. Values between $q > 3$ and $q < 8$ seem to be a good default choice. In general, it seems that the larger the polynomial, the larger the value for q should be.

Next, the sliding window variants were compared, with q ranging from 4 to 8, for the Mersenne Twister. It is important to note that the initialisation of the decomposition polynomial for the sliding window decomposition algorithm also takes time, which must be taken into account. This calculation only happens once, during the initialisation of the jump, but it can still be noticeable if jumps of the same size don't occur often. Therefore, in the next figure 5.7, there is an additional plot for the time of the sliding window algorithm plus the initialisation of the decomposition polynomial.

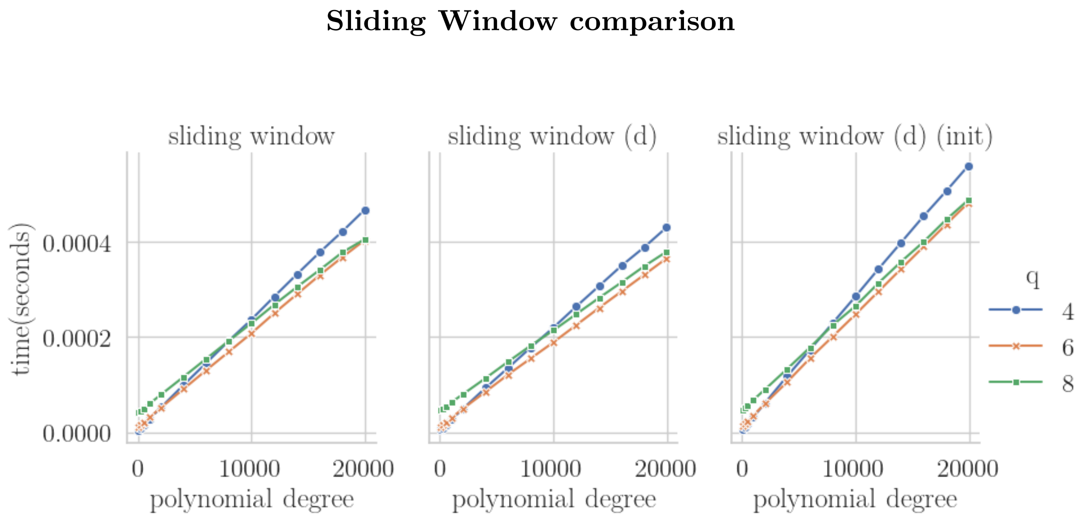

Figure 5.7: Comparison of the Sliding window algorithms for q = 4, 6 and 8. "sliding window (d)" is the sliding window decomposition algorithm without initialization, "sliding window (d) (init)" is the same algorithm plus the initialization time of the decomposition polynomial. The random number generator used is Mersenne Twister.

As expected, the algorithm that pre-calculates the decomposition polynomial is faster and the resulting slope is flatter. However, looking at the third plot, which adds the initialisation times, it is significantly slower than computing the sliding window algorithm on the fly.

It might be interesting to get a general idea of after how many jumps the sliding window decomposition algorithm outperforms the in-place variant. To do this, the execution time for each q and polynomial degree to is transformed into a line where the x values represent a jump, and the y values the execution time. For sliding window decomposition, the initialisation time is added as a constant factor c_{swd}. This results in the following equation:

$$x = \frac{c_{\text{swd}_{q,k}}}{\text{ex}_{\text{sw}_{q,k}} - \text{ex}_{\text{swd}_{q,k}}}, \tag{5.10}$$

It needs to be calculated for each q and k (the degree of the polynomial), with ex_{sw} being the execution time of the sliding window and ex_{swd} the execution time of the sliding window decomposition algorithm.

The results are shown in figure 5.8.

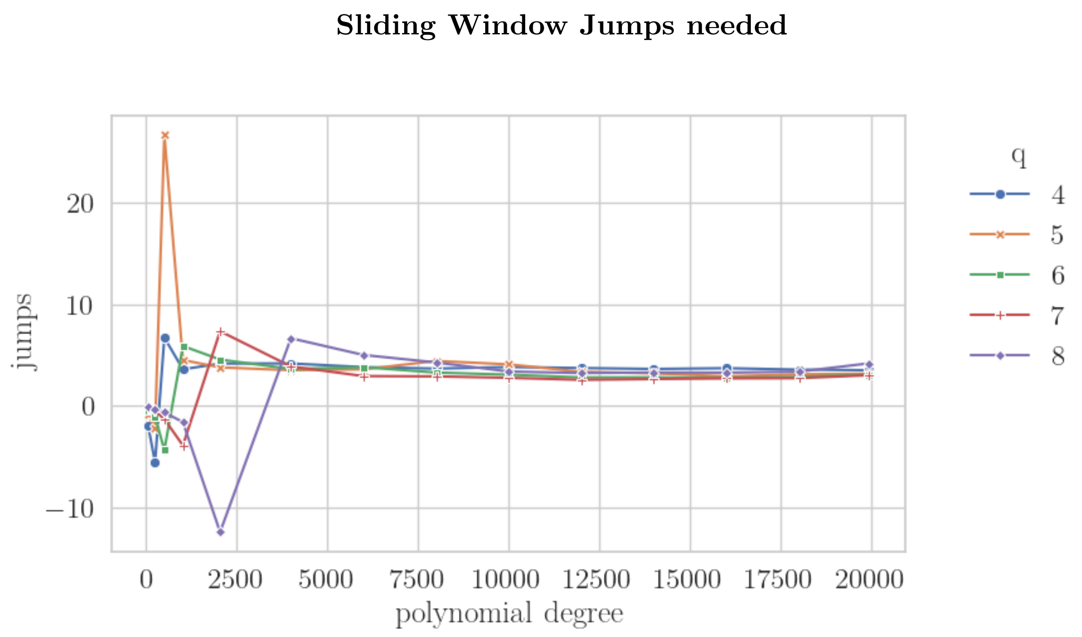

Figure 5.8: Analysis of how many jumps are needed so that sliding window decomposition might be preferred over sliding window.

From the figure it can be seen that for larger polynomials three to four jumps need to be performed in order for precomputing the decomposition polynomial to be faster. A negative jump value means that the normal sliding window algorithm is actually faster, which happens for smaller degree polynomials, perhaps due to noise in the data. Therefore, precomputing the decomposition polynomial will only give better performance if multiple jumps of the same size need to be performed. However, this difference may be so small that it may simply be better to always use the in-place variant.

Finally, the three jump algorithms were compared. The q values were chosen from the results of the previous benchmarks. The results and choices for q are shown in figure 5.9.

It shows that Horner's scheme works very well for generators with small state sizes. Remember that the sliding window is mainly used to reduce the number of state additions performed, which only seems to be noticeable for generators with larger states. XORSHIFT and TINYMT have such a small state size that the overhead required for a sliding window algorithm seems too great to gain any performance benefit.

However, for XOSHIRO, for jump polynomials of degree $k > 80$, sliding window decomposition becomes faster with a good choice of q. The in-place variant basically gives no performance advantage. But the difference between the jump algorithms is so small that it may simply be best to always use Horner's scheme, since sliding window decomposition only gives a performance boost for multiple jumps of the same size, due to the overhead of precomputing the decomposition polynomial.

For MERSENNE TWISTER, using any sliding window variant will give a good performance boost.

It should also be mentioned that it is more likely to encounter jump polynomials with higher degrees. This is due to the fact that a polynomial with degree k has 2^{k-1} variations, which is the same number of variations possible for degrees less than or equal to $k-1$.

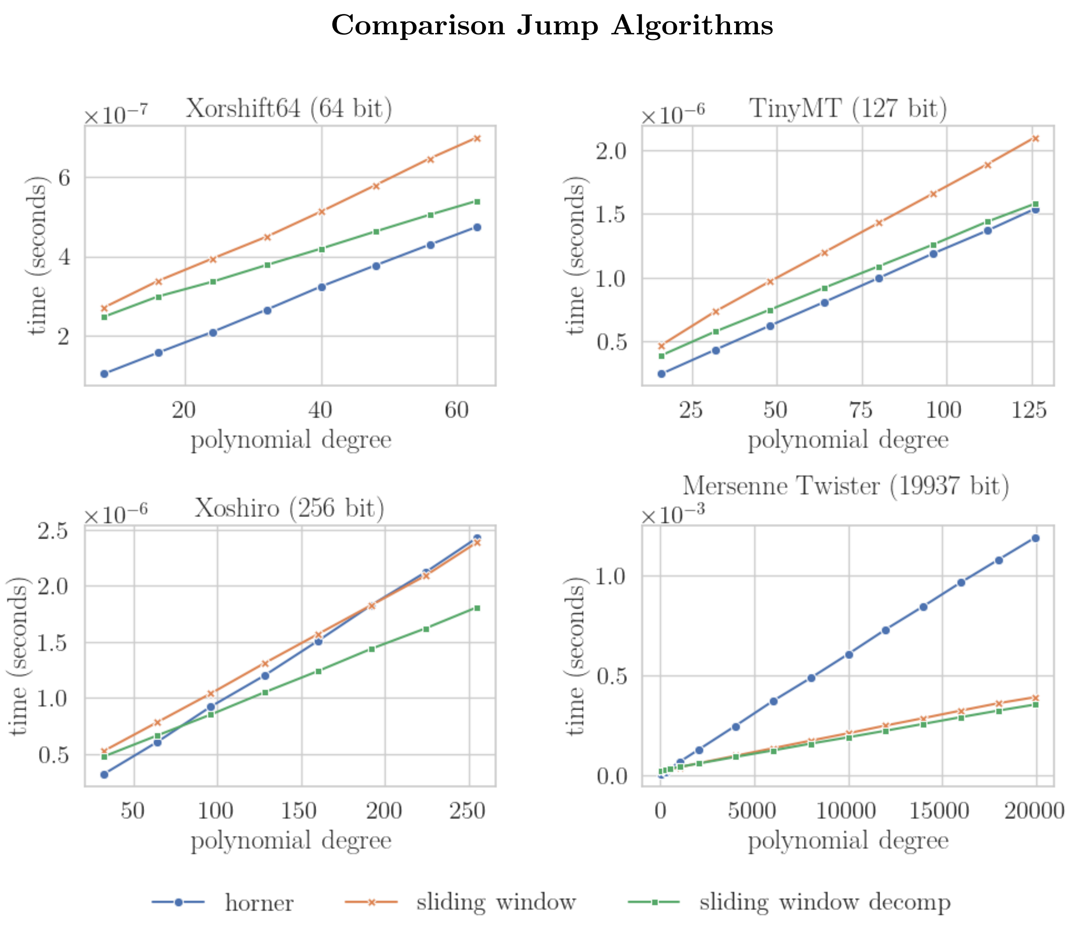

Figure 5.9: Comparison of the jump algorithms for the four random number generators. q is set to 2, 2, 3 and 7 for Xorshift, TinyMt, Xoshiro and Mersenne Twister, respectively.

So 50 percent of all possible polynomials will have a degree of k. Therefore, the results of higher degree polynomials should be given more weight.

In conclusion, the choice of algorithm depends mainly on the size of the state and the degree of the jump polynomial.

It might also be very useful to initialise the jumps first and store them somewhere, in case an application needs to perform the same jumps over and over again. The current implementation doesn't allow this, but in an actual release it would be very beneficial to provide this kind of functionality. Furthermore, an automatic configuration could simply set the algorithm to `HORNER` if the state size is equal to or less than 256, and to one of the sliding window algorithms otherwise, with a q in the range 5 to 7.

The last thing to note for generators with a state size greater than 256, more testing would be required to determine really good values for q. Unfortunately, the only generator currently implemented with a large state is Mersenne Twister, which leaves a huge gap for state sizes in the range 256 to 19937.

5.7.2 Sequential computation versus jumping

The purpose of this benchmark was to determine which jump sizes give a real performance advantage over sequential number generation. It was measured how long it takes to generate n numbers sequentially and perform a jump of size n. Additionally, a third measurement was made where the jump polynomial was initialised each time before performing a jump. This gives an indication of when it is beneficial to jump, in case an application needs to perform jumps with many different jump sizes that are not repeated very often.

As usual, all four random number generators were measured individually. The algorithm used for jumps was sliding window decomposition. Figure 5.10 shows the results of these measurements.

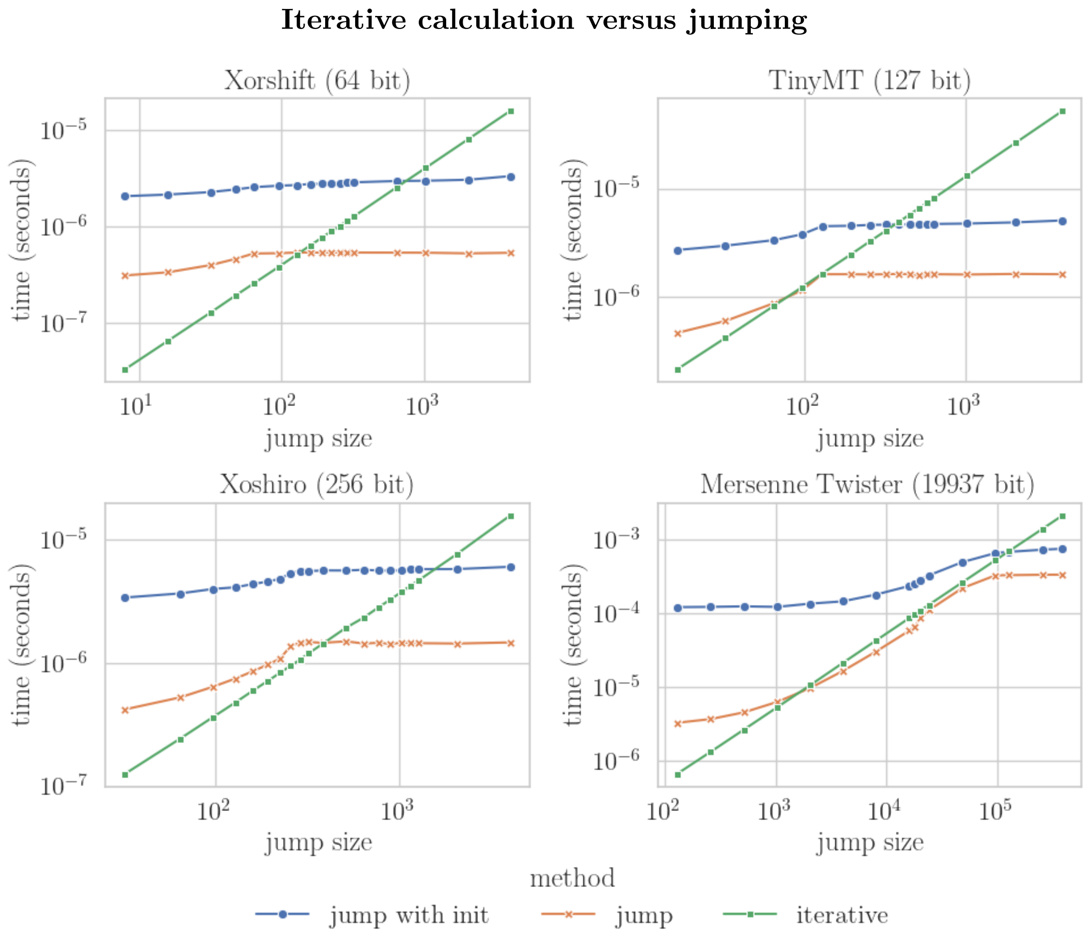

Figure 5.10: Comparison of iterative/sequential calculation of n numbers to perform a jump of n, compared to actual jumping. 'jump with init' refers to jumping with initialization of the jump polynomial, 'jump' to jumping without initialization by only evaluating the jump polynomial. 'iterative' performs a jump by computing the required about of numbers sequentially. Both axis use the logarithmic scale.

Computing each jump sequentially has, as expected, a linear time complexity. For XORSHIFT the jumps become faster from a jump size of about 200. For initialisation, a jump size of 1000 is preferable over an iterative calculation.

For TINYMT, which has a state about twice as large as XORSHIFT, a minimum jump size of 200 is also faster than sequential computation. Interestingly, the overhead of initialising a jump is less noticeable here, since starting from a jump size of about 500, jumping is faster even with initialisation.

Jumping with Xoshiro is advantageous from a jump size of about 500, and, including initialisation, from a jump size of about 2000.

The results for Mersenne Twister are very interesting, as a jump seems to be a bit faster for a jump size of only 4000. However, the difference is really noticeable from a jump size of 100000 and 200000 with initialisation.

One thing that is also apparent is that it is never advantageous to jump for an amount less than the state size of a generator. The reason for this can be seen by considering two parameters: the jump polynomial $g(z)$ and the algorithm for evaluating this polynomial.

Recall that the jump polynomial is given by $g(z) = z^v \bmod p(z)$. The degree of $p(z)$ is bounded by the state size of a generator, meaning a generator of state size k with maximum period will have a minimal polynomial of degree k. So for jumps with $v <= k$, $q(z)$ will simply be z^v. Next, the two algorithms for evaluating a jump polynomial are considered, starting with Horner's scheme. The main work of this algorithm is done in the loop (taken from listing 5.2):

```
i = v
while i > 0:
    if coeff(i, g) == 1:
        tmp = tmp ^ x
    next_state(tmp)
    --i;
```

Listing 5.28: While loop in Horner's scheme

This means that in the simple case of z^v only one 'xor' operation is performed. Otherwise, `next_state` is simply called v times, which is basically identical to generating numbers sequentially. The same is true for the sliding window algorithm, where at the end of the algorithm the random number generator is advanced by d_m steps (see 5.2.2), with $m = 1$ and $h_{m+1} = 0$ in this case. Sliding window will also have the overhead of having to do some initialisation first, before jumping.

This is also the reason why the evaluation of the jump polynomial becomes basically constant as soon as a jump size of k is reached. For the initialisation of the jump polynomial, the runtime is $k^2 \log(v)$, since it is usually implemented with the right-to-left binary squaring method, which can be seen in the plots. The factor of $\log(v)$ is barely visible, but a very slight increase in runtime can be seen between the last two data points for each generator.

In summary this means that, starting from a jump size of k, the computation of a jump can basically be interpreted as an operation with logarithmic time complexity.

Therefore, in a reference implementation, it would be advisable to check the jump size during initialisation. If it is less than k plus some constant factor, which would need to be determined for a random number generator, then the calculation of the jump polynomial could simply be skipped and the jump performed by simply calculating v numbers. The user wouldn't need to know the details of why a jump is slow, and would not get worse performance by using the jump. It was observed that for random number generators with more complex transition functions, such as TinyMt and Mersenne Twister, the time difference between initialising before jumping and only jumping was less. This is likely due to the reduced significance of the overhead involved in calculating the jump polynomial with the time required to perform the necessary state transitions to evaluate the jump polynomial.

5.7.3 Strong Scaling

The last benchmarks we performed were in regards to strong scaling.

Strong scaling measurements where done for small to medium problem sizes ranging from 128 to 10 million, with 1 to 128 processes. For large problem sizes in the range of 100 million to 25 billion, measurements were done with 1 to 65536 processes. This meant that the amount of numbers k generated by one processor was $\frac{n}{p}$ with the jumps ranging from 0 to $n - k$, in steps of k. Subsequently, the library's speedup was calculated.

Small to medium problem sizes

Figure 5.11 displays the results for values for n of 10^4, 10^5, 10^6 and 10^7 and 1 to 128 processes. Values below 10^4 were excluded as they did not show any scaling.

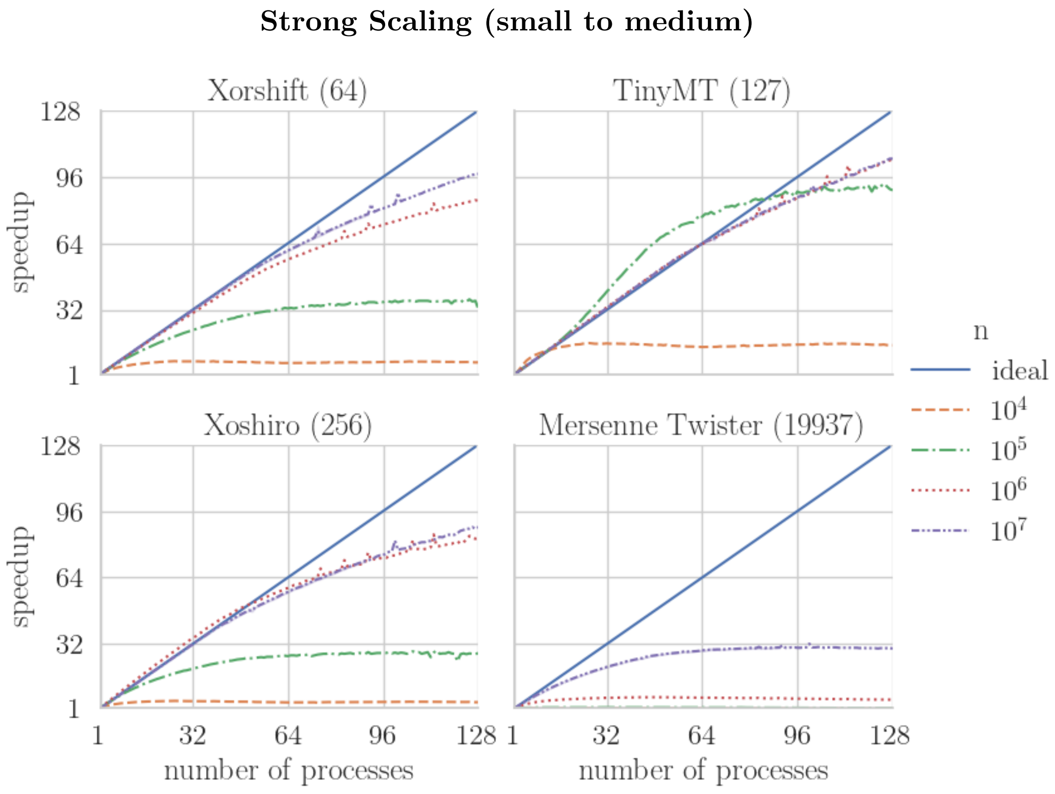

Figure 5.11: Results for strong scaling measurements. The values in brackets next to each generator denote its state size. The "ideal" line shows ideal strong scaling. Number of processes ranges from 1 to 128. n denotes the problem size (the amount of generated numbers).

The correlation between the scaling capacity and the state size of a random number generator, as well as the problem size, is immediately noticeable. Even for a very small state random number generator like Xorshift, there is only a small amount of scaling for a problem size of 10^4.

Another curious observation is that TinyMT exhibits super linear scaling for $n = 10^5$. However, the reason for this remains unclear. It is also worth noting, that TinyMT has superior scaling capacities compared to other generators, including Xorshift, which has only half the state size. On the other hand, Mersenne Twister shows the poorest scaling for problem sizes less than or equal than 10^7. When selecting a generator for calculating parallel substreams of random numbers, it is important to consider its state size. A very large generator such as Mersenne Twister requires a very high amount of numbers to be generated to really scale effectively. If a user intents to use Mersenne Twister for its random number stream properties, it may be a beneficial to consider TinyMT. When determining the number of processes, it is best to select a number that still demonstrates some scaling. For instance, when trying to calculate 10^5 numbers using Xoshiro it would not make any sense to use process count larger than approximately 50.

Large problem sizes

Figure 5.12 shows results for n of 10^8, 10^9 and 2.5×10^{10} and p from 1 to 65536.

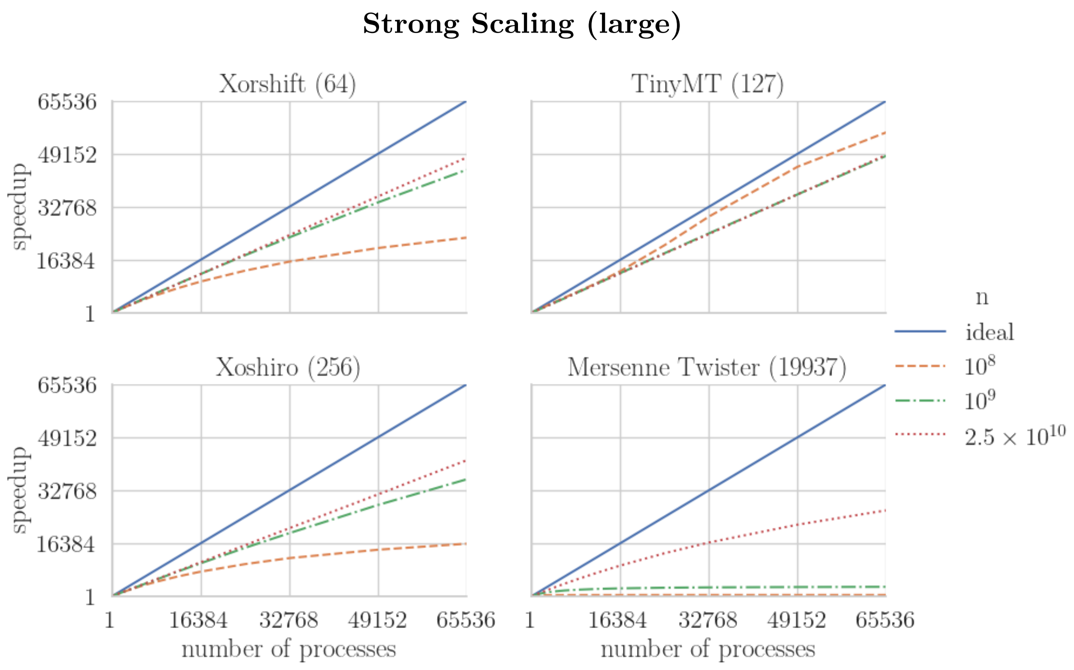

Figure 5.12: Results for large strong scaling measurements. The values in brackets next to each generator denote its state size. The ”ideal” line shows ideal strong scaling. Number of processes ranges from 1 to 65536. n denotes the problem size (the amount of generated numbers).

For large problem sizes, TinyMT again scales best, exhibiting almost super-linear scaling for a problem size of 10^8, which is better than Xorshift, despite the larger state size. Additionally, TinyMT shows better scaling for a smaller problem size of 10^8 compared to the other two problem sizes, which is unexpected. Though there is a drop off after 49152 processes. When comparing scaling for $n = 10^9$ and $n = 2.5 \times 10^{10}$, there is no significant difference in speedup. Moreover, the speedup for large problem sizes is nearly linear, although with a smaller slope compared to the ideal speedup. XORSHIFT and XOSHIRO exhibits a drop-off for $n = 10^8$ at around 16384 processes. However, they show a linear speedup for the other two problem sizes, with XORSHIFT scaling a slightly better. MERSENNE TWISTER has the weakest scalability. Even for $n = 10^8$, scaling stops almost immediately, rendering this problem size unsuitable for computations with a large number of processes. The speedup is also very low for $n = 10^9$, being significantly less when compared to the other generators. For a problem size of 2.5×10^{10} it shows a better speedup. For process counts less than 16384, its speedup is comparable to that of XOSHIRO, though it drops off afterwards.

In summary, the results for large problem sizes exhibit excellent scaling in general, with the exception of MERSENNE TWISTER, which only shows an average speedup for the largest problem size.

5.7.4 Conclusion

For generators of the type $\mathbb{F}_2$-Linear, giving a general recommendation on what algorithm to use is more involved and really depends on the generator itself. However, when it comes to selecting a method for evaluating a jump polynomial, Horner's scheme is the most suitable for small state generators with a state size of up to 256. For a large state generators such as the MERSENNE TWISTER, additional tweaking and testing is required. Choosing one of the sliding window variants may result in a noticeable increase in performance. It is important to determine a suitable value for q with a good default being between 4 and 8. The choice between the two variants depends on the frequency of a specific jump that needs to be performed. In general, choosing sliding window over sliding window decomposition may be better, except when precalculation is possible.

The use of a jump depends on the state size and state transition function of a generator. For the generators with state sizes less than or equal to 256, a jump can improve performance by a factor of 10^2 to 10^3. For larger state generators, a jump size of around 10^5 is required to see a performance gain.

These results are closely related to those of the strong scaling experiments performed. Small state generators demonstrate good scaling for small to medium problem sizes starting from approximately 10^5, where a large state generator would require a problem size of over 10^7 to achieve similar scalability. For large problem sizes, all generators scale well with a large number of processes, with the exception being MERSENNE TWISTER, who only shows good scaling at problem sizes greater than 2.5×10^{10}.

Chapter 6

Conclusion

It is possible to parallelize a random number generator if its underlying sequence can be transformed into an explicit formula. This can be done without further problems for a linear congruential generator (see chapter 2, equation 2.2). The main challenge is to find a way to evaluate the formula in less than linear time. This can be achieved using algorithms like the "right-to-left binary exponentiation" and "Algorithm C" (Chapter 2, listings 4.1 and 4.2), both of which have a logarithmic runtime. This thesis also addresses some of the shortcomings of the LCG, demonstrating that its statistical weaknesses become less significant for larger state sizes (starting from about 128 bit) when only considering the higher bits (Chapter 2, section 2.1.3).

For $\mathbb{F}_2$-linear generators, it is also possible to transform the sequence into an explicit formula, although it is bit more involved. In general, a series of bitwise 'shifts' and 'xor' operations can be represented through a transition matrix $\mathbf{A}$ in $\mathbb{F}_2$ (Chapter 2, section 2.2.1). The sequence can then described as a matrix vector product: $\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1}$ where $\mathbf{x}$ is the vector representation of a series of bits, representing the state of a generator (Chapter 2, equation 2.3). With the characteristic polynomial $p(z)$ of $\mathbf{A}$ it is then possible to calculate a jump polynomial $g(z) = z^v \bmod p(z)$, which can be used to jump ahead in the sequence by v steps (Chapter 2, section 2.2.3). The algorithm runs faster than linear time because the calculation of $g(z)$ takes only $O(k^2 \log v)$. k has an upper bound through the state size of a random number generator.

All of the above was verified in practice by the two implemented libraries, PRAND48 and F2LIN (chapters 4 and 5). The benchmarks of PRAND48 showed that it has excellent scaling capabilities, and confirmed the theoretical runtime of $O(\log n)$ when jumping. It was shown that jumping ahead is faster than sequential calculation by several orders of magnitude. It showed a good speedup for very small small problem sizes, and a significant speedup for larger problem sizes (Chapter 4, section 4.4)

Measuring the performance of F2LIN also provided valuable insights: The comparison between sequential computation and jumping ahead revealed that the performance benefit of a jump is linked to the jump size itself and the state size of a given generator. Nevertheless, once a certain threshold was reached, jumping became faster than sequential computation confirming the logarithmic runtime of calculating the jump polynomial. Moreover, if the algorithm performs multiple jumps of the same size, it runs constant time to the size of a generator(see Chapter 5, Section 5.7.2).

By measuring its strong scaling capabilities, it was shown that in order for a $\mathbb{F}_2$ generator to show a good speedup, it requires a larger problem size than compared to PRAND48. Additionally, scaling was limited by the state size of a generator, with a larger state also requiring a larger problem size to exhibit a good speedup (see Chapter 5, Section 5.7.3). When selecting a generator, a general rule of thumb is to use a smaller state for smaller problem sizes. Furthermore, F2LIN can be optimized by selecting different algorithms for polynomial evaluation to further improve its performance, which is particularly useful for generators with a large state (see chapter 5, section 5.7.1).

In the future, it might be interesting to research a combination of generators with jump functionality, since this may help increase the distributional properties of a generator. [Zim08] provides an example of the 'KISS' generator, which combines three types of random numbers generators: a linear congruential, an $\mathbb{F}_2$-linear (XORSHIFT), and a 'multiply-with-carry' generator. For this thesis, the third option was also considered, though no resources could be found on how to jump ahead in this type of generator. It is possible though to transform its underlying sequence into an explicit formula, suggesting the possibility of implementing a jump ahead for it.

In conclusion, this thesis has demonstrated efficient methods for parallelising common pseudo-random number generators, such as RAND48 and MERSENNE TWISTER.

Bibliography

[Bla18] Sebasitan Vigna David Blackman. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software*, 47:1–32, 2018.

[Bro] Robert G. Brown. Die harder test suite. https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[Bro94] Forrest B. Brown. Random number generation with arbitrary strides. *Transactions of the American Nuclear Society*, 71, 1994.

[for] The MPI forum. Mpi: A message-passing interface standard. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[Knu81] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 2 edition, 1981.

[L'E99] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[L'E08] Hiroshi Haramoto Makoto Matsumoto Takuji Nishimura Francois Panneton Pierre L'Ecuyer. Efficient jump ahead for f2-linear random number generators. *Journal on Computing*, 20(3):385–390, 2008.

[Leh49] D.H. Lehmer. Mathematical methods in large-scale computing units. *Proceedings of a Second Symposium on Large Scale Digital Calculating Machinery*, 26:141–146, 1949.

[LS07] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), 2007.

[M$^+$10] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[Mar03] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14), 2003.

[Mas69] James L. Massey. Shift-register synthesis and bch decoding. *IEEE Transactions on oinformation theory*, 15(1):122–127, 1969.

[Nis98] Makoto Matsumoto Takuju Nishimura. Mersenne twister: a 623-dimensionally equidirstibuted uniform pseudo-random number generator. *ACM Transactions on Moteling and Computer Simulation*, 8(1):3–30, 1998.

[O'N14] Melissa E. O'Neil. Pcg : A family of simple fast space-efficient statistically good algorithms for random number generation. *Harvey Mudd College Computer Science Department Technical Report*, 2014.

[Pan07] Pierre L'Ecuyer Francois Panneton. F2-linear random number generators. *Advancing the Frontiers of Simulation*, pages 169–193, 2007.

[Sch05] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley, 2 edition, 2005.

[Stu] High Perfomance Computing Center Stuttgart. Hpe apollo (hawk). `https://www.hlrs.de/solutions/systems/hpe-apollo-hawk`.

[Zim08] Richard P. Brent Pierrick Gaudry Emmanuel Thomé Paul Zimmermann. Faster multiplication in gf(2)[x]. *ANTS-VIII - Algorithmic Number Theory*, pages 153–166, 2008.