8-28-2024 10:30 AM

Efficient Algorithms and Parallel Implementations for Power Series Multiplication

Seyed Abdol Hamid Fathi, *Western University*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Abstract

Power series play an important role in solving differential equations and approximating functions. A key operation in manipulating power series is their multiplication. Power series multiplication algorithms working based on a prescribed precision, say n (where n is a natural number), take the first n coefficients of the two power series as input, multiply them, and return the first n coefficients of the product. While these algorithms can be fast, they incur the overhead of recomputing known terms to enhance the product precision. On the other hand, lazy or relaxed multiplication algorithms compute the product terms incrementally. This allows for dynamic updates of product precision without the need to recompute the already known terms. In this thesis, we discuss efficient multiplication algorithms for univariate and multivariate power series, based on various schemes, including the Karatsuba algorithm, a novel partition multiplication technique using FFT, and an evaluation-interpolation strategy, along with their complexity analyses and parallelization opportunities. These algorithms and methods are implemented in C++ and integrated in the BPAS (Basic Polynomial Algebra Subprograms) library. To parallelize the implementations, we use a thread pool with a work-stealing scheduler using modern C++ multithreading techniques. The performance results, comparing the execution times of various algorithms in both serial and parallel modes, are presented and analyzed.

Keywords: Power series multiplication, Karatsuba, Fast Fourier Transform (FFT), Parallel programming, C++, Multithreading, Thread pool, Divide-and-conquer, Partition multiplication, Evaluation-interpolation.

Lay Summary

Power series are polynomial-like objects with potentially infinite terms and are utilized in various mathematical areas, such as function approximation and solving differential equations. For example, computers approximate transcendental functions like $\sin(x)$ by summing the terms of a power series. Our focus is on the multiplication of power series, which forms the basis for other mathematical operations. There are two primary approaches to multiplying two formal power series, f and g. Methods working in prescribed precision involve expanding f and g up to a predetermined order, multiplying these expansions, and then truncating the product. While these methods can use fast multiplication algorithms, such as the Fast Fourier Transform (FFT), they lack the flexibility to reuse previous computations when higher precision is required. In contrast, lazy and relaxed methods compute the product terms incrementally, allowing for dynamic precision improvements. In this thesis, we explore and implement various static and relaxed algorithms and strategies for multiplying both univariate and multivariate power series. We identify opportunities for concurrency within these algorithms and parallelize them using appropriate parallel programming techniques. The performance results of these implementations are presented and discussed.

Acknowlegements

I would like to thank my supervisor, Professor Marc Moreno Maza, for his continuous guidance, significant contributions, and constant support throughout all stages of this research work. I am thankful to Dr. Alexander Brandt and Haoze Yuan for their valuable discussions and assistance with the implementations. I am also grateful to every one at the Ontario Research Centre for Computer Algebra (ORCCA) for their time and kind help in the lab.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Since the early days of scientific computing, numerical analysis has been the main provider of algorithms for solving problems from the mathematical field of *functional analysis*, such as continuous optimization and solving differential equations. Indeed, in the field of functional analysis, the basic objects, such as real numbers and limits of functions, are not always computable and, thus, must be be approximated.

While symbolic computation is the realm of exact methods, this latter field has developed exact solutions to approximate problems, which can be used to provide approximate values to non-computable objects. A well-known example is the so-called *symbolic Newton iteration method* for approximating solutions of algebraic equations, see the landmark textbook, *Modern Computer Algebra* [vzGG03].

At the heart of these methods is the manipulation of formal multivariate power series, including Laurent series and Puiseux series.

Power series multiplication algorithms can be divided into two groups. First, there are algorithms which set up a truncated degree in advance, say n, so that each power series is "represented" by a polynomial of degree n which collects all the terms, of that power series, with degree at most n, see [Knu97]. In particular, the product of two such truncated power series is also represented by a truncated power series of the same degree n. These power series multiplication algorithms are sometimes termed *static*. Note that properly setting the degree n can be difficult, which can lead to either extraneous or repeated computations.

A second strategy uses *lazy evaluation*. In this approach, the terms of a power series are computed only when they are needed. Moreover, they are cached and no prescribed degree n needs to be set ahead of time. The obvious advantages of this approach are that: (1) unnecessary computations can be avoided, and (2) no prescribed degree n needs to be guessed ahead of time. However, there is an inconvenience: a typical implementation of power series arithmetic based on lazy evaluation tend to compute terms incrementing their total degree by 1, instead of by a larger gap, which would open the door to use faster algorithms for polynomial arithmetic, such as those based on Karatsuba's trick or Fast Fourier Transforms (FFTs). This inconvenience was solved by the scheme of *relaxed algorithms* proposed by Joris van der Hoeven and that we review below.

Power series are polynomial-like objects, with potentially infinitely many terms. As such, it is natural to represent a power series h as a function $n \longmapsto h^{(n)}$ returning all the

terms of a given degree n. One may also store the terms of h that have already been returned by this function. Indeed, computing $h^{(n)}$ may have a cost, for instance if h is the product of two other power series f and g, in which case one uses the well-known formula:

$$h^{(n)} = \Sigma_{i=o}^{i=n} f^{(i)} g^{(n-i)}. \tag{1.1}$$

This raises the following question. Suppose that $h^{(0)}, h^{(1)}, \ldots, h^{(n-1)}$ have been computed and stored, while $h^{(n)}$ has not been computed but is requested by other calculation. Should we simply compute just $h^{(n)}$, or should we compute $h^{(n)}$ together with a few more terms, say $h^{(n+1)}, \ldots, h^{(2n)}$. The former approach, advocating for the least effort, is that lazy evaluation, while the latter, depending on the context, can be called relaxed evaluation or eager evaluation.

The advantage of lazy evaluation is to minimize the amount of computations and stored data. Algorithms for arithmetic operations on power series based on lazy evaluation are studied in [BM21, ABK+21]. The advantage of the alternative approach is that it offers the possibility to use faster algorithms for computing $h^{(n)}, h^{(n+1)}, \ldots$. To illustrate this observation, let us assume that h, f, g are univariate power series in the variable X. We can write:

$$\begin{array}{rcl} h^{(2n)} & = & h_0 + h_1 X + \cdots + h_n X^n + h_{n+1} X^{n+1} + \cdots + h_{2n} X^{2n} \\ f^{(2n)} & = & f_0 + f_1 X + \cdots + f_n X^n + f_{n+1} X^{n+1} + \cdots + f_{2n} X^{2n} \\ g^{(2n)} & = & g_0 + g_1 X + \cdots + g_n X^n + g_{n+1} X^{n+1} + \cdots + g_{2n} X^{2n}, \end{array} \tag{1.2}$$

where h_i (resp. f_i) (resp. g_i) is the coefficient of degree i of h (resp. f) (resp. g), for $0 \leq i \leq 2n$. We can re-arrange $h^{(2n)}, f^{(2n)}, g^{(2n)}$ as follows:

$$\begin{array}{rcl} h^{(2n)} & = & h^{(n)} + X^n C_n \\ f^{(2n)} & = & f^{(n)} + X^n A_n \\ g^{(2n)} & = & g^{(n)} + X^n B_n, \end{array} \tag{1.3}$$

where each of C_n, A_n, B_n is a polynomial which is either zero, or non-zero with degree at most n and at least 1. Indeed, if one of C_n, A_n, B_n has a term of degree n, then it belongs to $h^{(n)}, f^{(n)}, g^{(n)}$ respectively. Suppose that $h^{(n)}$ and $f^{(n)} \cdot g^{(n)}$ are known and that the goal is to compute all the terms of $X^n C_n$, so that $h^{(2n)}$ becomes known, as well as $f^{(2n)} g^{(2n)}$ in order to, later on, increase the precision from $2n$ to $4n$. Of course, we are also assuming that $f^{(2n)}$ and $g^{(2n)}$ are known since f and g are our input power series. Observe that we have:

$$f^{(2n)} g^{(2n)} = D_{2n} + X^n E_{2n} + X^{2n} F_{2n}, \tag{1.4}$$

where:

$$\begin{array}{rcl} D_{2n} & = & f^{(n)} g^{(n)} \\ E_{2n} & = & A_n g^{(n)} + B_n f^{(n)} \\ F_{2n} & = & A_n B_n. \end{array} \tag{1.5}$$

Assume that both $A_n \neq 0$ and $B_n \neq 0$ hold. Then the degree of $A_n B_n$ is at least 1, thus the degree of $X^{2n} A_n B_n$ is at least $2n+1$. It follows that all the terms of $h^{(2n)}$ appear in $D_{2n} + X^n E_{2n}$. To be more precise, we have:

$$f^{(2n)} \equiv D_{2n} + X^n E_{2n} \mod X^{2n+1}. \tag{1.6}$$

In other words, it suffices to compute $D_{2n} + X^n E_{2n}$ and remove every term there which is a multiple of X^{2n+1}.

Using Karatsuba's multiplication trick [KO63] we can re-arrange the computations of D_{2n}, E_{2n}, F_{2n} as follows:

$$\begin{array}{rcl} D_{2n} & = & f^{(n)} g^{(n)} \\ F_{2n} & = & A_n B_n \\ E_{2n} & = & (A_n - B_n)(f^{(n)} - g^{(n)}) + D_{2n} + F_{2n}. \end{array} \tag{1.7}$$

Hence, by computing the two products $A_n B_n$ and $(A_n - B_n)(g^{(n)} - h^{(n)})$ we deduce both $h^{(2n)}$ and $f^{(2n)} g^{(2n)}$. This is done at the cost of computing two products of polynomials of degree at most n, which can be done via Karatsuba's multiplication trick or Toom-Cook algorithms [Too63, Coo66, BZ07, Zan09] or an algorithm based on FFT, see [vzGG03].

This implies that $h^{(2n)}$ and $f^{(2n)} g^{(2n)}$ can be deduced from $h^{(n)}$ and $f^{(n)} \cdot g^{(n)}$ at the cost of two "fast" multiplications in degree at most n. If instead we compute $f^{(n)}, f^{(n+1)}, \ldots$, one after another in a lazy evaluation manner, using Equation (1.1) the cost of those successive computations is necessarily quadratic in n, that is, equivalent to doing a plain "non-fast" multiplications in degree n.

The scheme that we just presented was proposed for univariate power series multiplication by Joris van der Hoeven in his landmark article *Relax, but Don't be Too Lazy* [vdH02] with follow up works in [vdHL13, vdH14, vdH19].

It is highly desirable to adapt the relaxed multiplication presented above for univariate power series to multivariate power series. Unfortunately, a major difficulty arises, which is attributed to the dimensionality. Formula (1.3) no longer applies in a multivariate setting. And, even if we could solve that issue, the number of terms in A_n, B_n, C_n would be much larger than in $f^{(n)}$, $g^{(n)}$, $h^{(n)}$. Hence, the potential savings due to the memorization of the products $g^{(n)} h^{(n)}$ would be negligible.

In this thesis, we explore alternative solutions to obtain efficient multiplication for multivariate power series. The contributions of this thesis are the following ones.

In Chapter 2, we implement the relaxed multiplication method for univariate power series using the Karatsuba's algorithm and compare its performance results with the static Karatsuba and lazy multiplication methods.

In Chapter 3, we report on the implementation of a thread pool using the language constructs of C++ Standard 14. The intention is to build multithreading support for our power series multiplication relying only on the features of the C++ language. We use this support in our parallelization of Karatsuba's method and our experimental results suggest that we obtain some improvement w.r.t. to our implementation based on Cilk. We note that the parallelization of Karatsuba's trick has also been studied in [CMPS10] and [LZMQ19]. A study of the parallelization of Karatsuba's method targeting distributed memory architectures is presented in [CM96].

In Chapter 4, we consider different schemes of multivariate power series multiplications based on distributivity of multiplication over addition. Thus, these schemes are, to some sense, in the spirit of Karatsuba. One of them, which combines two types of truncation (in total degree and in partial degrees) outperforms the other studied schemes experimentally.

In Chapter 5 we consider the evaluation-interpolation scheme based on [Sch05] for computing the product of two multivariate power series modulo a monomial ideal. We observe that, while this method has good theoretical performance for truncation in partial degrees, it is not optimal for truncation in total degree. As a future work, for truncation in total degree we would consider the work of Lecerf and Schost in [LS03]. We report on an implementation and present the performance results.

Chapter 2

Efficient Univariate Power Series Multiplication

In this chapter, we show that the Karatsuba multiplication method can be effectively used as a relaxed method for univariate power series multiplication.

Power series are mathematical constructs similar to polynomials but with potentially infinitely many terms. One approach of multiplying power series is to treat them statically as polynomials by truncating them at a specific precision and then multiplying these truncated power series (TPS) using standard polynomial multiplication techniques such as the naive method (with quadratic time complexity), Karatsuba, or FFT. We refer to this approach as *static* (or *fixed precision*) power series multiplication. Whenever we need to achieve a higher product precision, we start over by truncating input power series at a higher precision and then multiply larger TPS. This method incurs the overhead of recomputing the already known product terms.

Another multiplication approach is to consider the dynamic nature of power series and the possibility of increasing the precision. This approach tries to reuse the currently computed product terms when trying to increase the product precision for the next round. We call this approach *dynamic* (or *varying precision*) power series multiplication.

A power series multiplication method is referred to as *lazy* if it is dynamic, meaning it reuses previous computations and applies the naive multiplication algorithm to compute the product. On the other hand, a power series multiplication method is referred to as *relaxed* if it is dynamic and uses the Karatsuba multiplication algorithm to compute the product. The relaxed power series multiplication method reuses previous computations in two ways. First, similar to the lazy method, the known product terms does not need to be recalculated. Second, as will be detailed in this chapter, the previously computed product is used as one of the three sub-products required by the Karatsuba algorithm to compute the higher precision product.

Starting with the theory, we explain the Karatsuba multiplication algorithm used in relaxed univariate power series along with its time and space complexities and how the already computed terms of the product power series can be recycled and used to compute the next number of terms of the series. Furthermore, we identify independent parts of the Karatsuba algorithm that can be executed in parallel.

Moving on to the implementation, we first demonstrate how we represent power series

as instances of classes in an object-oriented programming language with data members to store the already known terms and member functions to compute additional terms of the series. Regular power series, product power series using naive multiplication, and product power series using divide-and-conquer (DnC) Karatsuba multiplication have different implementations of the functions used to compute more terms of the series.

Finally, we perform experiments on our product power series implementation and compare the running times of the static naive, lazy, static Karatsuba, and relaxed algorithms with different problem input sizes and DnC threshold values and present and analyze the results. We parallelize the static Karatsuba and the relaxed methods in Chapter 3.

2.1 Relaxed Karatsuba Multiplication Algorithm

Karatsuba's method for multiplying polynomials is based on the observation that the product $(a_1X + a_0) \times (b_1X + b_0) = a_1b_1X^2 + (a_1b_0 + a_0b_1)X + a_0b_0$ can be calculated using $a_1b_1X^2 + ((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0)X + a_0b_0$, therefore requiring only three multiplications instead of four, and each of the three multiplications can be computed by recursively applying Karatsuba's method.

Given two truncated univariate power series $f = f_0 + f_1z + ... + f_{n-1}z^{n-1}$ and $g = g_0 + g_1z + ... + g_{n-1}z^{n-1}$, we define $f_* = f_0 + ... + f_{\lceil n/2 \rceil - 1}z^{\lceil n/2 \rceil - 1}$ and $f^* = f_{\lceil n/2 \rceil} + ... + f_{n-1}z^{\lceil n/2 \rceil - 1}$. Similarly, we define $g_* = g_0 + ... + g_{\lceil n/2 \rceil - 1}z^{\lceil n/2 \rceil - 1}$ and $g^* = g_{\lceil n/2 \rceil} + ... + g_{n-1}z^{\lceil n/2 \rceil - 1}$. Hence, we can write f and g as

$$\begin{aligned} f &= f_* + f^*z^{\lceil n/2 \rceil} \\ g &= g_* + g^*z^{\lceil n/2 \rceil} \end{aligned} \tag{2.1}$$

Then, similar to what Karatsuba and Ofman [KO63] found, we can write

$$fg = f_*g_* + ((f_* + f^*)(g_* + g^*) - f_*g_* - f^*g^*)z^{\lceil n/2 \rceil} + f^*g^*z^{2\lceil n/2 \rceil} \tag{2.2}$$

Applying the above formula recursively for calculating f_*g_*, $(f_*+f^*)(g_*+g^*)$, and f^*g^* multiplications, we can devise a divide-and-conquer algorithm as shown in Algorithm 1. In this algorithm the product f_*g_* is denoted by the array variable *low*, the product $(f_* + f^*)(g_* + g^*)$ is denoted by the array variable *mid*, and the product f^*g^* is denoted by the array variable *high*.

The DnC Karatsuba algorithm can be a relaxed method of univariate power series multiplication. We recall that relaxed methods for power series multiplication share the properties of zealous and lazy methods. That is the time complexity of the relaxed power series multiplication is less than $O(n^2)$ as in zealous methods, and increasing the product precision does not involve recomputing the previous product coefficients which is a feature of lazy methods. If one wants to increase the product precision from $h^{(2N-1)} = f^{(N)} \times g^{(N)}$ to $h^{(4N-1)} = f^{(2N)} \times g^{(2N)}$, ${f_*}^{(2N)}{g_*}^{(2N)}$ is actually $h^{(2N-1)}$ which has already been computed.

It is noted that in Algorithm 1, there are three recursive function calls that are independent and hence can be parallelized. The parallelization of the Karatsuba method using multithreading and the implementation results are shown in Chapter 3.

2.2 Complexity Analysis of Karatsuba Method

In this section, we analyze the time and space complexities of the Karatsuba method. Since the Karatsuba method is a recursive divide-and-conquer algorithm, we use master theorem for its analysis.

Based on master theorem [CLRS22], if $a > 0$ and $b > 1$ are constants, $f(n)$ is a nonnegative function, and we have the following recurrence for $T(n)$ for $n \in \mathbb{N}$.

$$T(n) = aT(n/b) + f(n), \tag{2.3}$$

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

2.2.1 Time Complexity

If we denote by T_1 the serial execution time of the Karatsuba algorithm, and n the input size, here is the recurrence relation for the time complexity of the algorithm:

$$T_1(n) = 3T_1(n/2) + O(n) \tag{2.4}$$

This recurrence relation falls within the first case of the master theorem, and hence, the asymptotic time complexity of the serial Karatsuba algoirthm is $O(n^{\log_2 3})$.

In parallel execution, assuming that there are an infinite number of threads, the three recursive multiplications can be performed in parallel, and hence, the execution time is reduced to the time required for one multiplication plus the linear combining time. If we denote the parallel execution time by T_∞, we will have:

$$T_\infty(n) = T_\infty(n/2) + O(n) \tag{2.5}$$

The recurrence relation above falls within the third case of the master theorem, and therefore the parallel time complexity of the algorithm is $O(n)$.

2.2.2 Space Complexity

If we denote by S the memory storage required for the Karatsuba algorithm execution, we will have the following recurrence relation:

$$S(n) = 3S(n/2) + O(n) \tag{2.6}$$

which is the first case of master theorem, and thus, the space complexity of the Karatsuba algorithm is $O(n^{\log_2 3})$.

We note that there exist versions of Karatsuba and Toom-Cook algorithms working in place, that is, working in space $O(n)$, see the paper [CMPS10], or nearly in-place (that is, with an extra storage in $O(\log(n))$, see [Roc09].

Algorithm 1 Karatsuba

Input: Univariate polynomials $f = f_0 + ... + f_{n-1}z^{n-1}$ and $g = g_0 + ... + g_{n-1}z^{n-1}$.
Output: The product fg.

function DNCMULTIPLY(f, g, n)
  if $n \leq Threshold$ **then**
    return $\sum_{i=0}^{2n-2}(\sum_{j=\max(0,i+1-n)}^{\min(n-1,i)} f_j g_{i-j})z^i$
  else
    $low =$ DNCMULTIPLY$(f_*, g_*, n/2)$
    $mid =$ DNCMULTIPLY$((f_* + f^*), (g_* + g^*), n/2))$
    $high =$ DNCMULTIPLY$(f^*, g^*, n/2)$
    return $low + mid \times z^{\lceil n/2 \rceil} + high \times z^{2\lceil n/2 \rceil}$
  end if
end function

2.3 Implementation in C++

We have implemented the lazy and relaxed multiplication of univariate power series in the fast and object-oriented `C++` programming language following van der Hoeven [vdH02].

Basically, a power series is implemented as a class that stores the already computed terms representing a truncated power series or polynomial and has the functionality to compute more terms when needed.

The polynomial part of a power series is represented by an array. In this array, each element stores the coefficient of a polynomial term, and the index of the element is the exponent of that term. Since the coefficients can be arbitrarily large, we use the GMP Library to handle them.

2.3.1 The GMP Library

The The GNU Multiple Precision Arithmetic Library (GMP) [GtGdt21] is a portable library in C language for arithmetic operations on integers, rational numbers, and floating-point numbers that require higher precision than what standard C types can support.

GMP aims to provide the fastest possible arithmetic on high precision numbers by using fullwords as the basic arithmetic type, sophisticated algorithms with an emphasis on speed, and including carefully optimized assembly code for many different CPUs.

All GMP types and functions are declared in the `<gmp.h>` header file, and programs using GMP must link against the `libgmp` library (`-lgmp` on Unix-like systems). The GMP types for storing integers, rational numbers, and floating-point numbers are `mpz_t`, `mpq_t`, and `mpf_t`. The corresponding arithmetic functions start with the prefixes `mpz_`, `mpq_`, and `mpf_` respectively.

2.3.2 Description of Classes and Methods

The truncated power series are represented by instances of the class `TPS`. As shown in Listing 2.1 This class has a pointer to an array of elements of rational numbers and the

length of the array which is the truncation order. This array of rational numbers are the coefficients of the terms of the univariate truncated power series, and the the index of each element is the degree of the corresponding term. Clearly, since all elements of the truncated power series are stored, this implementation is most efficient in handling dense power series.

Power series are implemented as objects of the `Series` class that are to be used directly by the user. The `Series` class has a series representation class `SeriesRep`. The `SeriesRep` class in turn has a `TPS` class and a virtual method `next()` which yields the coefficients of a series one by one. The `Series` and `SeriesRep` classes are shown in Listing 2.2.

Whenever the coefficient of a term of degree k of a series class with known terms up to degree n is requested, the series class first checks if $k \leq n$ in which case the element k of the current coefficient array of the truncated power series is returned; otherwise, the `next()` method is called repeatedly until the coefficients of degree $n+1, ..., k$ are computed and the coefficient array of the truncated power series is updated, and then the coefficient of the element k is returned.

The product series representation classes, `LazyProdSeriesRep` and `RelaxedProdSeriesRep`, shown in Listings 2.3 and 2.4 respectively, inherit from the `SeriesRep` class and override the `next()` method. In the `LazyProdSeriesRep` class, the `next()` method calculates the term(s) of the product series of the next required degree using the naive multiplication method, as shown in Listing 2.6. Whereas in the `RelaxedProdSeriesRep` class, the `next()` method computes a group of terms of the product series using the relaxed Karatsuba multiplication method, as shown in Listing 2.5. It is noted that the naive multiplication method is also used in the relaxed Karatsuba multiplication method when the problem input size is less than a threshold.

The `LazyProdSeries` and the `RelaxedProdSeries` classes, intended for direct user interaction, inherent from the `Series` class and have an instance of the `LazyProdSeriesRep` class and the `RelaxedProdSeriesRep` class respectively, as shown in Listing 2.7.

```
1  class TPS {
2    int arraySize;
3    mpq_t *coefficientArray;
4  };
```

Listing 2.1: The TPS class.

```
1  class SeriesRep {
2    public:
3    TPS* getTPS() const {
4      return phi;
5    }
6    int getSize() const {
7      return n;
8    }
9    void setSize(const int size) {
10     n = size;
11   }
```

```
12    virtual void next();
13    private:
14    TPS *phi;
15    int n;
16  };
17  class Series {
18    SeriesRep sRep;
19    virtual void getCoefficient(int index, mpq_t coefficient);
20  };
```

Listing 2.2: The SeriesRep and Series classes.

```
1   class LazyProdSeriesRep : public SeriesRep {
2     public:
3       LazyProdSeriesRep(Series, Series);
4       void next() override;
5     private:
6       void naiveMultiply(int arraySize, mpq_t *fArray, mpq_t *
          gArray, mpq_t *prodArray);
7       Series f, g;
8   };
```

Listing 2.3: The LazyProdSeriesRep class.

```
1   class RelaxedProdSeriesRep : public SeriesRep {
2     public:
3       RelaxedProdSeriesRep(Series, Series);
4       void DnCMultiply(int arraySize, mpq_t *fArray, mpq_t *gArray,
            mpq_t *prodArray);
5       void next() override;
6     private:
7       void naiveMultiply(int arraySize, mpq_t *fArray, mpq_t *
          gArray, mpq_t *prodArray);
8       Series f, g;
9       int DnCThreshold;
10  };
```

Listing 2.4: The RelaxedProdSeriesRep class.

```
1   void RelaxedProdSeriesRep::DnCMultiply(int arraySize, mpq_t *
      fArray, mpq_t *gArray, mpq_t *prodArray) {
2     if (arraySize <= DnCThreshold) {
3       naiveMultiply(arraySize, fArray, gArray, prodArray);
4     } else {
5       mpq_t *fLowStar = fArray;
6       mpq_t *gLowStar = gArray;
7       mpq_t *fUpStar = fArray + arraySize / 2;
8       mpq_t *gUpStar = gArray + arraySize / 2;
9       mpq_t *fLowPlusUp = new mpq_t[arraySize / 2];
```

```
10      mpq_t *gLowPlusUp = new mpq_t[arraySize / 2];
11
12      for (int i = 0; i < arraySize / 2; ++i) {
13        mpq_init(fLowPlusUp[i]);
14        mpq_init(gLowPlusUp[i]);
15        mpq_add(fLowPlusUp[i], fArray[i], fArray[arraySize / 2 + i
              ]);
16        mpq_add(gLowPlusUp[i], gArray[i], gArray[arraySize / 2 + i
              ]);
17      }
18
19      // low = (f_*)(g_*)
20      // mid = (f_* + f^*)(g_* + g^*)
21      // high = (f^*)(g^*)
22      // size (# of elements) of these = arraySize - 1
23      mpq_t *lowArray = new mpq_t[arraySize - 1];
24      mpq_t *midArray = new mpq_t[arraySize - 1];
25      mpq_t *highArray = new mpq_t[arraySize - 1];
26      for (int i = 0; i < arraySize - 1; ++i) {
27        mpq_init(lowArray[i]);
28        mpq_init(midArray[i]);
29        mpq_init(highArray[i]);
30      }
31
32      // calculation of low, mid, high
33      DnCMultiply(arraySize / 2, fLowStar, gLowStar, lowArray);
34      DnCMultiply(arraySize / 2, fUpStar, gUpStar, highArray);
35      DnCMultiply(arraySize / 2, fLowPlusUp, gLowPlusUp, midArray);
36
37      // mid = mid - low - high
38      for (int i = 0; i < arraySize - 1; ++i) {
39        mpq_sub(midArray[i], midArray[i], lowArray[i]);
40        mpq_sub(midArray[i], midArray[i], highArray[i]);
41      }
42
43      // assemble product array from low, high, mid - low - high
44      for (int i = 0; i < arraySize - 1; ++i)
45        mpq_add(prodArray[i], prodArray[i], lowArray[i]);
46      for (int i = 0; i < arraySize - 1; ++i)
47        mpq_add(prodArray[arraySize / 2 + i],
48        prodArray[arraySize / 2 + i], midArray[i]);
49      for (int i = 0; i < arraySize - 1; ++i)
50        mpq_add(prodArray[arraySize + i],
51          prodArray[arraySize + i], highArray[i]);
52
53      // memory deallocation
54      for (int i = 0; i < arraySize / 2; ++i) {
```

```
55        mpq_clear(fLowPlusUp[i]);
56        mpq_clear(gLowPlusUp[i]);
57      }
58      for (int i = 0; i < arraySize - 1; ++i) {
59        mpq_clear(lowArray[i]);
60        mpq_clear(midArray[i]);
61        mpq_clear(highArray[i]);
62      }
63      delete[] fLowPlusUp;
64      fLowPlusUp = nullptr;
65      delete[] gLowPlusUp;
66      gLowPlusUp = nullptr;
67      delete[] lowArray;
68      lowArray = nullptr;
69      delete[] midArray;
70      midArray = nullptr;
71      delete[] highArray;
72      highArray = nullptr;
73    }
74  }
```

Listing 2.5: The DnCMultiply method.

```
1  void RelaxedProdSeriesRep::naiveMultiply(int arraySize, mpq_t *
       fArray, mpq_t *gArray, mpq_t *prodArray) {
2    mpq_t partialProduct;
3    mpq_init(partialProduct);
4    for (int i = 0; i < arraySize; ++i) {
5      for (int j = 0; j < arraySize; ++j) {
6        mpq_mul(partialProduct, fArray[i], gArray[j]);
7        mpq_add(prodArray[i + j], prodArray[i + j], partialProduct)
             ;
8      }
9    }
10   mpq_clear(partialProduct);
11 }
```

Listing 2.6: The naiveMultiply method.

```
1  class LazyProdSeries : public Series{LazyProdSeriesRep lPSR;};
2  class RelaxedProdSeries : public Series{RelaxedProdSeriesRep rPSR
       ;};
```

Listing 2.7: The LazyProdSeries and RelaxedProdSeries classes.

Power Series Multiplication Algorithm	**Description**	**Polynomial Multiplication Used**
Fixed precision Naive	Computes $(f^{(d)}g^{(d)} \mod x^d)$ for a given d without reusing previous results.	Plain (Quadratic)
Lazy (varying precision)	Computes the terms of degrees $d/2, \ldots, d-1$ of fg and stores them, assuming results up to $d/2-1$.	Plain
Fixed precision Karatsuba	Computes $(f^{(d)}g^{(d)} \mod x^d)$ for a given d without reusing previous results.	Karatsuba
Relaxed (varying precision)	Computes the terms of degree $d-1$ of fg, reusing previous results, and stores those terms.	Karatsuba

Table 2.1: Descriptions of the implemented power series multiplication algorithms.

2.4 Experimental Results

In this section, we present the experimental results from testing our implementations of the static naive, lazy, static Karatsuba, and relaxed multiplication methods. For further clarity, given two univariate power series f and g, Table 2.1 specifies the characteristics of the computation of their product (as a truncated power series) for the 4 methods mentioned in Table 2.4.

We note that, in the literature, there exist versions of the plain multiplication (for dense univariate polynomials) which are optimized in terms of cache complexity, see [CMPS10], by means of a divide-and-conquer scheme. However, the version that we are using is a purely iterative one based on a 2-loop nest.

We measure and compare the time required to compute the product of power series using these methods at varying levels of precision.

All tests were run on a system with the specifications listed in Table 2.2.

The execution times for the static naive multiplication method and the static Karatsuba multiplication method with different divide-and-conquer thresholds are presented in Table 2.3 and shown graphically in Figures 2.1 and 2.2. These experimental results show that the static Karatsuba method outperforms the static naive method, and the threshold value of $T = 16$ gives the optimal performance in the static Karatsuba method.

The lazy power series multiplication algorithm, which uses naive polynomial multiplication, and the relaxed power series multiplication algorithm, which uses Karatsuba polynomial multiplication, both reuse previously computed products when an increase in product precision is required. Table 2.4 and Figures 2.3 and 2.4 present comparisons of

Component	**Specifications**
CPU	Intel Core i7-4710HQ @ 2.50GHz
L1 Data Cache (L1d)	128 KiB (4 instances)
L1 Instruction Cache (L1i)	128 KiB (4 instances)
L2 Cache	1 MiB (4 instances)
L3 Cache	6 MiB (1 instance)
Hyperthreading	Enabled
RAM	8GB
Operating System	Ubuntu 22.04.1 LTS
Compiler	clang++ 16.0.6
Compiler Optimization Level	Default (`-O0`)

Table 2.2: System Specifications.

Product	**Static**	**Static Karatsuba**						
Precision	**Naive**	**T = 2**	**T = 4**	**T = 8**	**T = 16**	**T = 32**	**T = 64**	**T = 128**
2	5.47e-5	2.10e-5	2.50e-5	4.50e-5	9.00e-5	0.0003	0.0009	0.0036
4	8.81e-5	1.50e-5	6.00e-6	2.60e-5	6.40e-5	0.0003	0.0009	0.0035
8	0.0001	3.90e-5	2.70e-5	2.00e-5	6.00e-5	0.0003	0.0009	0.0033
16	0.0002	0.0001	8.50e-5	7.60e-5	5.90e-5	0.0003	0.0009	0.0032
32	0.0005	0.0004	0.0003	0.0002	0.0002	0.0002	0.0009	0.0031
64	0.0014	0.0012	0.0009	0.0008	0.0007	0.0008	0.0009	0.0030
128	0.0026	0.0035	0.0027	0.0024	0.0022	0.0025	0.0029	0.0035
256	0.0064	0.0102	0.0080	0.0072	0.0069	0.0082	0.0080	0.0083
512	0.0221	0.0254	0.0216	0.0190	0.0181	0.0203	0.0214	0.0240
1024	0.0848	0.0677	0.0547	0.0488	0.0468	0.0516	0.0566	0.0703
2^{11}	0.3400	0.2040	0.1641	0.1426	0.1366	0.1525	0.1697	0.2100
2^{12}	1.3647	0.6150	0.4961	0.4328	0.4126	0.4628	0.5142	0.6383
2^{13}	5.5781	1.8566	1.5011	1.3127	1.2566	1.4128	1.5612	1.9354
2^{14}	22.2973	5.6013	4.5192	3.9757	3.8064	4.2768	4.7292	5.8319
2^{15}	90.3683	16.8652	13.6735	12.0067	11.5153	12.9148	14.2982	17.6861
2^{16}	373.7040	50.8692	41.3140	36.2653	34.8404	39.0014	43.2828	53.5939
2^{17}	1512.7900	153.4520	124.8990	109.5690	105.4290	117.8210	130.8930	162.2700
2^{18}	5995.5200	459.6320	373.1990	327.3920	316.3050	351.9760	389.8980	483.2490

Table 2.3: Execution time (second) of the static naive and static Karatsuba with different threshold values.

the results from these lazy and relaxed algorithms with those obtained using the naive and Karatsuba methods without reuse of previous computations. It can be observed that the lazy and relaxed methods are more efficient than the static naive and static Karatsuba methods, respectively.

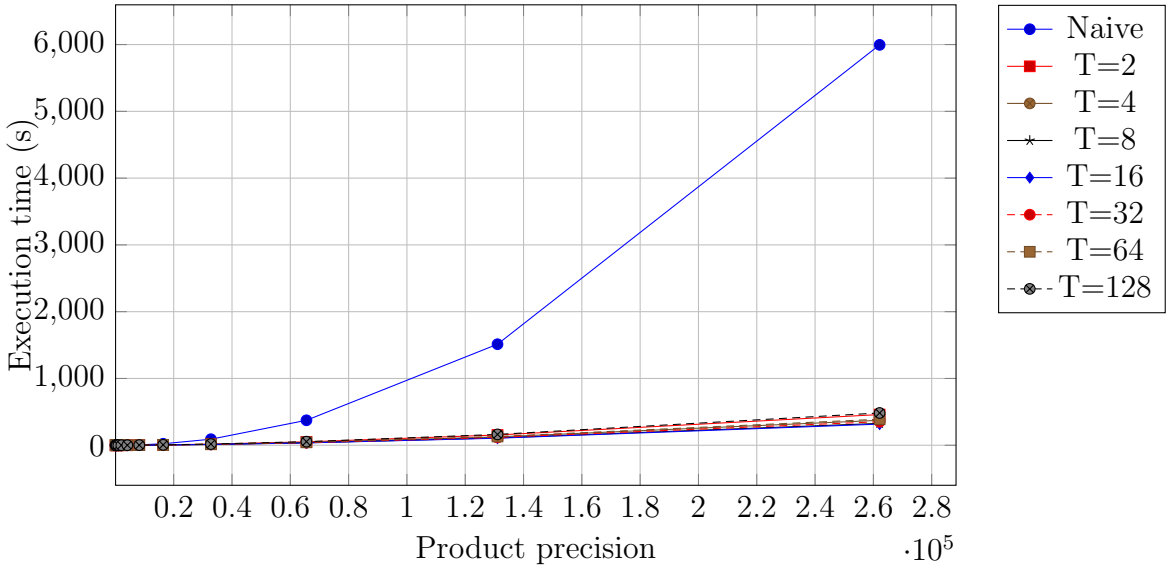

Figure 2.1: Execution time of static naive multiplication and static Karatsuba multiplication with different threshold values versus product precision.

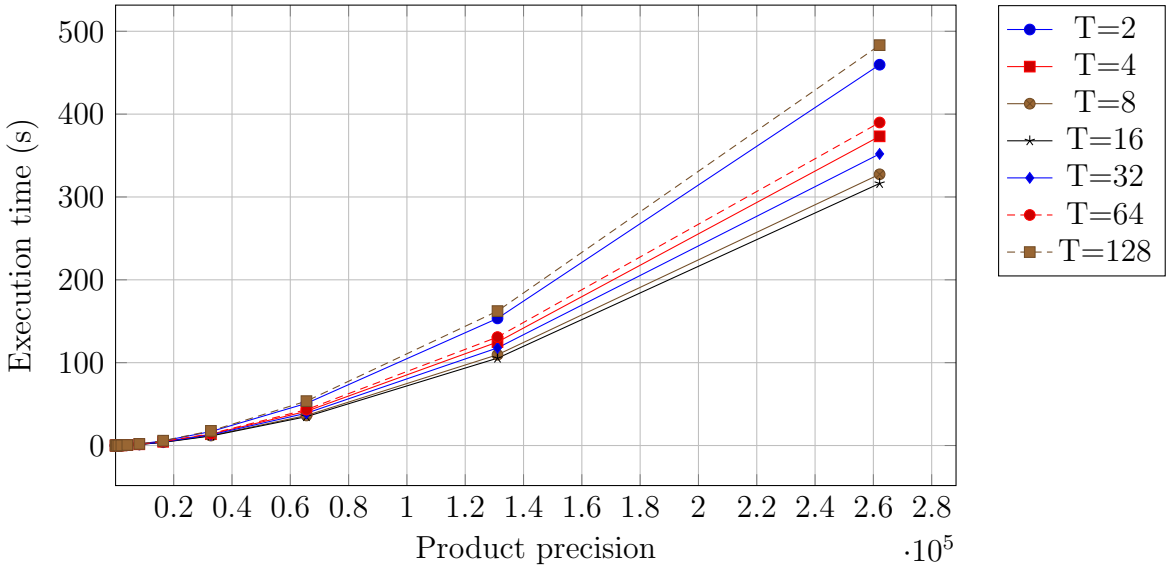

Figure 2.2: Execution time of static Karatsuba multiplication method with different threshold values versus product precision.

Product Precision	Static Naive Multiplication	Lazy Multiplication	Static Karatsuba Multiplication	Relaxed Multiplication
2	0.0001	0.0001	0.0001	0.0001
4	0.0001	0.0001	0.0001	0.0001
8	0.0001	0.0001	0.0001	0.0001
16	0.0002	0.0001	0.0001	0.0001
32	0.0005	0.0002	0.0002	0.0002
64	0.0014	0.0003	0.0007	0.0006
128	0.0026	0.0011	0.0022	0.0017
256	0.0064	0.0042	0.0069	0.0047
512	0.0221	0.0167	0.0181	0.0131
1024	0.08480	0.0667	0.0468	0.0319
2^{11}	0.3400	0.2614	0.1366	0.0910
2^{12}	1.3647	1.0560	0.4126	0.2736
2^{13}	5.5781	4.2607	1.2566	0.8177
2^{14}	22.2973	17.2254	3.8064	2.4597
2^{15}	90.3683	69.6232	11.5153	7.4391
2^{16}	373.704	282.392	34.8404	22.4544
2^{17}	1512.79	1149.76	105.429	68.3184
2^{18}	5995.52	4601.09	316.305	202.588

Table 2.4: Execution times (seconds) for lazy and relaxed power series multiplication methods, which reuse previous computations, compared to naive and Karatsuba methods, which do not reuse previous computations.

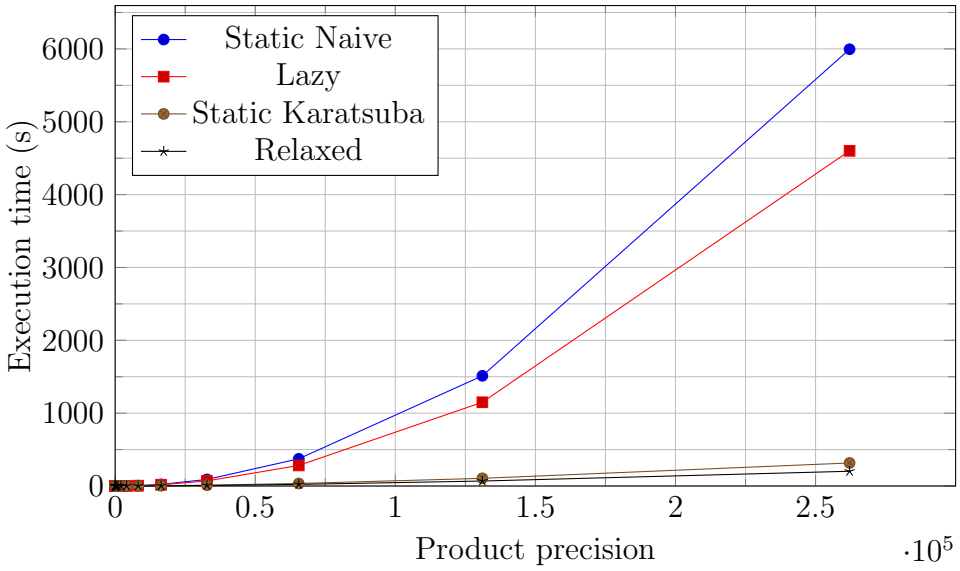

Figure 2.3: Execution times (seconds) for lazy and relaxed power series multiplication methods, compared to static naive and Karatsuba methods.

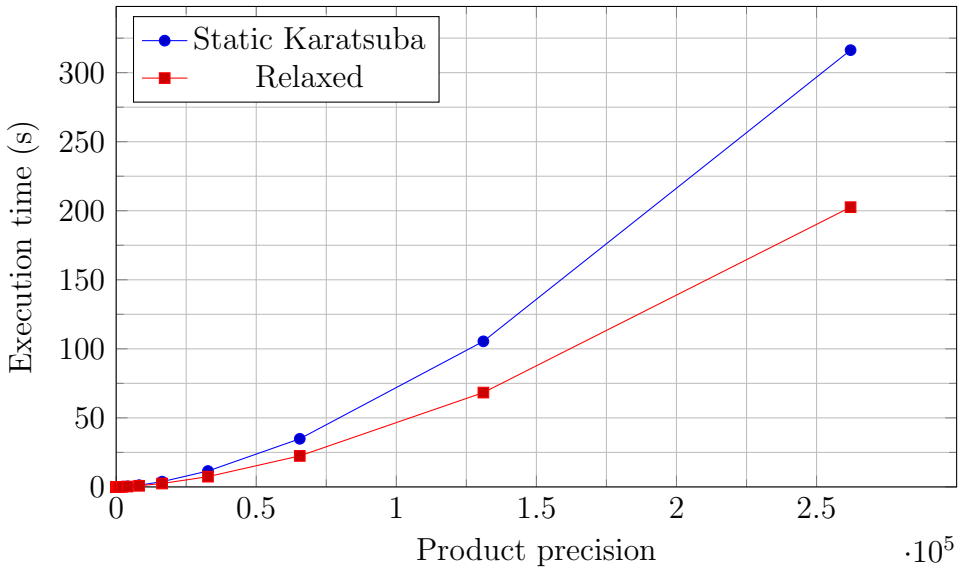

Figure 2.4: Execution times (seconds) for the relaxed and static Karatsuba power series multiplication methods.

Chapter 3

Parallelizing Algorithms with Multithreading

In designing fast and high-performance algorithms, it is essential to consider the capabilities and constraints of contemporary hardware architectures. Hardware manufacturers have shifted focus towards parallel architectures due to the physical limitations of increasing clock speeds. In this chapter, we begin by reviewing the theoretical performance measures of parallel algorithms and explaining various parallel programming patterns. Next, we discuss the fundamentals of multithreading using C++ and develop a thread pool with a work-stealing scheduler. We parallelize the static Karatsuba and relaxed power series multiplication methods discussed in Chapter 2 using this thread pool and compare the results with those obtained using Cilk. Although using Cilk for multithreading is relatively effortless and efficient, it introduces an external dependency to the project, which can lead to compatibility and portability issues. Additionally, Cilk does not provide the programmer with low-level control over thread management, synchronization, and error handling. Our multithreaded implementation does not rely on external libraries and achieves execution results that outperform those of Cilk.

3.1 Theoretical Analysis of Parallel Algorithms

This section reviews analytic performance measures of parallel programs from a theoretical perspective. These measures enable us to compare parallelization strategies and algorithms. First, we define the relevant concepts and terminology. Then, we discuss theorems related to the analysis of parallel algorithms.

Work, denoted by T_1, is the total amount of time it takes to run a program on a single processor. It is essentially the sum of the execution times of all operations of the algorithm performed sequentially.

Span, denoted by T_∞, is the minimum amount of time it takes to execute the program on a machine with an infinite number of processors. Span is also referred to as the *critical path length* and represents the longest sequence of dependent computations in the algorithm.

The ratio of work to span (T_1/T_∞) is called *parallelism*.

If we denote by T_P the total amount of time it takes to execute the program on P processors, the *speedup* S is defined as T_1/T_P.

Efficiency E is speedup divided by the number of processors, $E = S/P = \frac{T_1}{PT_P}$, with an ideal value of 1.

An algorithm that runs P times faster on P processors, i.e., $S_P = T_1/T_P = P$, is said to have a *linear speedup*. Although linear speedup is considered optimal, achieving that in practice is challenging due to factors such as task dependencies, communication overhead, and resource contention. Generally, we have the following lower-bound for T_P:

$$\frac{T_1}{P} \leq T_P \tag{3.1}$$

It is natural to represent the execution order and dependencies between operations using a *directed acyclic graph* (DAG). A DAG consists of nodes and directed edges without cycles. Each node represents a computational task, and each edge represents a dependency between tasks. An example of a DAG is shown in Figure 3.1.

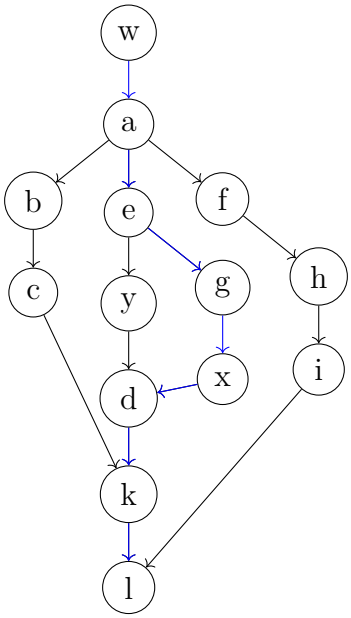

Figure 3.1: A DAG representing the dependencies and execution order of tasks. The critical path is shown in blue.

3.1.1 Graham-Brent Theorem

The Graham-Brent Theorem [Bre74] states that the execution time T_P of a parallel algorithm on P processors is bounded by:

$$T_P \leq T_1/P + T_\infty \tag{3.2}$$

To prove the theorem, we denote by m_i the number of operations on level i of the algorithm DAG with n levels. From the definition of T_1, we can write:

$$T_1 = \sum_{i=1}^{n} m_i. \tag{3.3}$$

Also $T_\infty = n$. The time it takes for P processors to execute level i of the DAG is:

$$T_P^i = \lceil \frac{m_i}{P} \rceil \leq \frac{m_i}{P} + 1. \tag{3.4}$$

Therefore,

$$T_P = \sum_{i=1}^{n} T_P^i \leq \sum_{i=1}^{n} \left(\frac{m_i}{P} + 1 \right) = T_1/P + T_\infty. \tag{3.5}$$

3.2 Concurrency Versus Parallelism

Concurrency refers to handling multiple tasks together, but not necessarily executing them simultaneously. It can involve interleaving the execution of these tasks. For example, when the processor encounters long-latency operations, such as memory reads that result in cache misses, it can switch to other tasks instead of idly waiting [MRR12a].

Parallelism, on the other hand, refers to executing multiple tasks simultaneously, typically on multiple processors or cores. Concurrency is a more general term that includes actual parallelism. Here, we are concerned with concurrency that comes from parallelism.

3.3 Parallel Programming Patterns

Parallel programming patterns identify reusable structures and themes frequently encountered in parallel computing [MRR12a]. These patterns apply to parallel programming systems regardless of the hardware architecture or programming language being used. Here, we describe the fork-join, map, stencil, reduction, and pipeline patterns. The fork-join pattern is used to parallelize the recursive divide-and-conquer Karatsuba method from Chapter 2, and the map pattern is used to parallelize the partition method discussed in Chapter 4.

3.3.1 Map

The map parallel pattern is a foundational concept in parallel computing, often used in functional programming and data parallelism. In a map pattern, a function is applied to every element in a collection, such as a list or an array, independently and concurrently to produce a new collection of elements. This pattern is particularly powerful because it is naturally parallelizable, meaning that the operations on each element do not depend on one another. Algorithm 2 and Figure 3.2 respectively show pseudo-code and an illustration of the map pattern.

Algorithm 2 Map Pattern

Input: Collection, FUNCTION
Output: Results

```
1: if Collection is empty then return Empty collection
2: else
3:     Results = Empty list
4:     for all Element in Collection parallel do
5:         result = FUNCTION(Element)
6:         APPEND(Results, result)
7:     end for
8:     return Results
9: end if
```

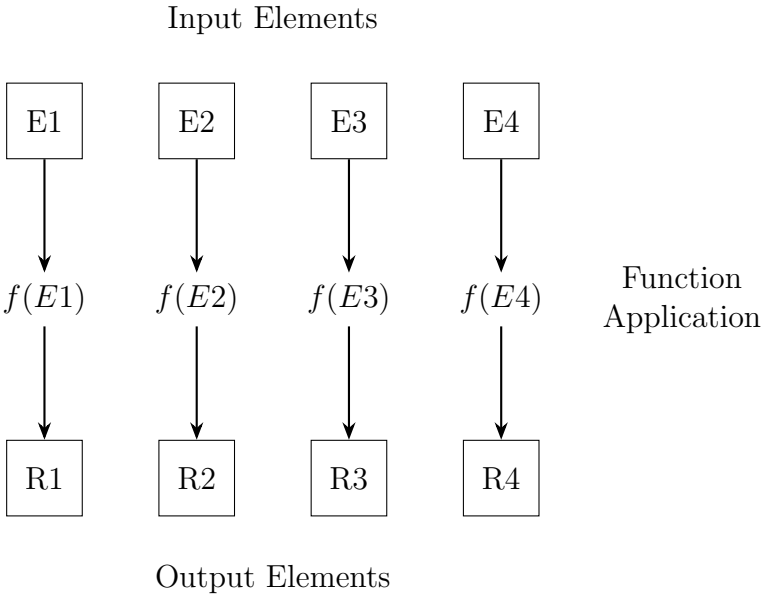

Figure 3.2: A diagram of the map pattern. Function f is applied to input elements in parallel producing output results.

3.3.2 Pipeline

Pipeline is a parallel programming pattern that processes tasks in a linear sequence of stages, where each stage performs a part of the overall computation on the data. Two consecutive stages of a pipeline form a producer-consumer pair. Pipelines with a fixed number of stages are not scalable with more processors; however, they multiply the speedup by a constant factor, with the maximum being equal to the number of stages. A pipeline pseudo-code and diagram are shown in Algorithm 3 and Figure 3.3, respectively.

Algorithm 3 Pipeline Pattern

```
1: Initialize pipeline stages
2: while more data do
3:     Receive data element from previous stage
4:     Perform operation on data element
5:     Send data element to next stage
6: end while
```

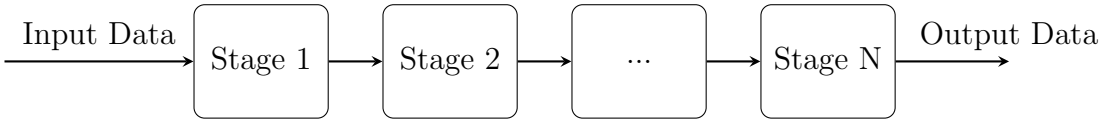

Figure 3.3: Diagram illustrating the pipeline pattern with N sequential stages. Each stage can be executed by a separate processor or thread.

3.3.3 Reduction

The reduction pattern involves combining all elements in a collection into a single result using an associative combiner function, which ensures that the function produces the same result regardless of the grouping of operations. This pattern is efficient for parallel execution as it allows the problem to be broken down into smaller subproblems that can be processed concurrently. Each processor performs the combiner function on its subset of elements independently, and the partial results are then combined in a hierarchical manner until a single final result is obtained. Examples of reduction operations include summing an array of numbers, finding the maximum value in a set, or performing a logical AND across a series of boolean values. Algorithm 4 presents the pseudocode for performing summation using parallel reduction. Figure 3.4 shows a diagram of the reduction pattern.

3.3.4 Fork-Join

The fork-join model is a parallel computing design pattern in which the program execution branches off (forks) at certain points in the program, the branches run in parallel, and they merge (join) at a later point of the program execution [MRR12b, Con63, NL16].

Algorithm 4 Parallel Reduction Pattern

```
1: function ParallelReduction(array, n)
2:     Input: array of n elements
3:     Output: result of reduction operation
4:     Initialize: threadCount = GetNumberOfThreads
5:     Parallel:
6:     localSum[threadId] = 0
7:     for i = threadId; i < n; i += threadCount do
8:         localSum[threadId] += array[i]
9:     end for
10:    Synchronize threads
11:    Reduction:
12:    globalSum = 0
13:    for i = 0; i < threadCount; i++ do
14:        globalSum += localSum[i]
15:    end for
16:    return globalSum
17: end function
```

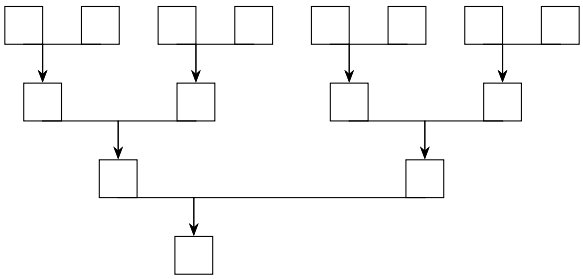

Figure 3.4: A diagram illustrating the reduction pattern.

Algorithm 5 shows the fork-join model pseudo-code, and Figure 3.5 shows a diagram of the fork-join model applied to a recursive divide and conquer problem.

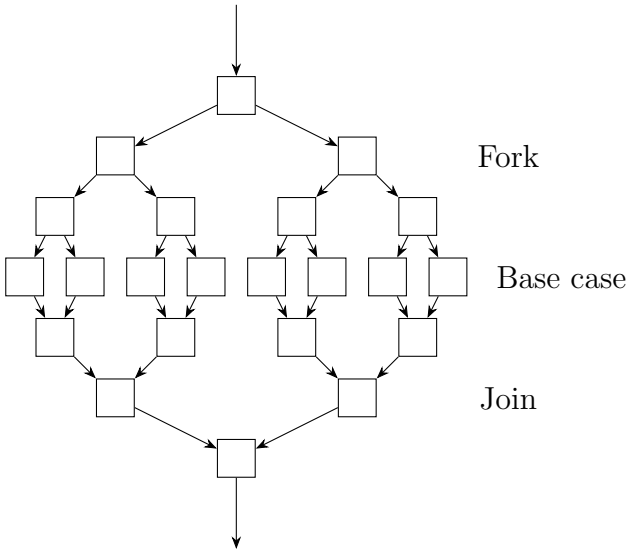

Figure 3.5: Illustration of the fork-join model applied to a recursive divide-and-conquer problem.

Algorithm 5 Fork-Join Pattern

```
1: function forkJoin(Problem)
2:     if Problem is base case then return Solve sequentially
3:     else
4:         Divide Problem into Sub-problems and spawn a thread for each Sub-problem
5:         for all Sub-problems do forkJoin(Sub-problem)
6:         end for
7:         Join all threads spawned for Sub-problems
8:         Return combined results
9:     end if
10: end function
```

Recursive divide and conquer problems can be parallelized by nested fork-join model.

The fork-join model is typically implemented using a thread pool paradigm to avoid oversubscription with a work stealing scheduler to ensure proper load balancing [MRR12b, Lea00].

3.4 Why Multithreading

Until around 2005, the general trend of improving program execution speed was to increase the CPU frequency made possible by improvements in semiconductor manufacturing. After that, it became clear that simply increasing clock speeds was no longer

sustainable due to the physical limitations such as the exponential increase in power consumption and heat generation at higher CPU frequencies. As a result, the computer architecture industry shifted focus from increasing clock speeds to multicore processors to improve performance [Bor07, HP11].

Maximum utilization of multicore processors requires concurrency where the program performs multiple independent tasks in parallel rather than sequentially.

Concurrency can be achieved with creating multiple single-threaded processes or multiple threads in one process.

Compared to multithreading, multiprocessing is more resource-intensive and incurs higher overhead. This is because the operating system devotes extra internal resources to manage the processes and prevent them from accidentally modifying data belonging to other processes. In multithreading, however, multiple threads within a process can run independently while sharing the same address space, resulting in lower overhead. The challenge with multithreading is that the programmer must carefully manage shared data and synchronize threads to prevent issues such as race conditions, deadlocks, and data corruption.

3.5 Threads

A thread is a basic unit of CPU utilization. A thread has a thread ID, a program counter (PC), a set of registers, and a stack. Threads within the same process share the same address space, which includes the code segment, data segment, and other operating system resources such as open files and signal handlers.

Thread support can be provided either at the user level, by *user threads*, or at the operating system level, by *kernel threads*. User threads are handled by a user-level library, whereas kernel threads are created and managed directly by the operating system. User threads are mapped to kernel threads in a many-to-one, one-to-one, or many-to-many model [SGG18]. The threads in the C++ Standard Library are intended to map one-to-one with the operating system's threads [Str13].

3.6 Multithreading and Synchronization in C++

3.6.1 C++ Support for Multithreading

In the C++ programming language, support for multithreading was introduced in the C++11 Standard and enhanced by the C++14, C++17, and C++20 Standards to provide robust, portable, and efficient concurrency features for modern software development [Com11, Com14, Com17, Com20].

Currently, the C++ Standard Thread Library has a thread-aware memory model, and includes classes for managing threads, protecting shared data, synchronizing operations between threads, and low-level atomic operations. Notably, the Resource Acquisition Is Initialization (RAII) technique is used with locks to ensure that mutexes are unlocked when the relevant scope is exited [Wil19].

3.6.2 C++ Memory Model

An important feature of the current C++ Standard is its multithreading-aware memory model. This model describes how different threads interact with memory.

The building blocks of data in C++ are objects. According to the C++ Standard [ISO20], an object has a type, it occupies a region of storage, and it can have a name. Some objects are simply values of fundamental types such as `int` while others are instances of user-defined classes. Some objects such as arrays or instances of derived classes contain sub-objects. Each memory location, that is the smallest addressable unit of storage, is either an object (or a sub-object), or a series of adjacent bit fields.

In a C++ program, every object has a modification order that includes all writes to that object by all threads, beginning with its initialization. While this order can differ between program runs, all threads must agree on the order within any single execution.

When two or more threads access the same memory location and at least one of them is modifying the data, there is a potential for a race condition and undefined behavoir. A race condition occurs when the result of operations depends on the order in which two or more threads execute those operations.

The race condition is avoided by enforcing an ordering among the accesses of the threads. One way to ensure there is an ordering is to use mutexes. Another way is to use atomic operations.

3.6.3 Launching Threads

Every C++ program starts with a single thread running the `main()` function. This thread can launch additional threads that run concurrently with itself and each other.

The functions and classes for managing threads are declared in the `<thread>` header. Threads are launched by constructing a `std::thread` object. Every thread has to be given a callable object in the constructor of the `std::thread` object that specifies the task to be run by that thread. The callable object can be a regular function, a function pointer, a lambda function, a function object (functor), a `std::function` object or any other callable type.

A started thread needs to be joined or detached before the `std::thread` object is destroyed, otherwise, the `std::thread` destructor calls `std::terminate()` and the program terminates. Joining a thread means waiting for it to finish and detaching it means leaving it to run in the background and passing its ownership over to the C++ Runtime Library.

Creating more threads than the hardware can support is known as oversubscription and can result in excessive context switching and memory waste which significantly decreases the performance. The maximum number of threads that can run concurrently can be determined using the `std::thread::hardware_concurrency()` function from the C++ Standard Library.

A naive approach to avoid the overhead of creating too many threads in recursive divide-and-conquer programs is to execute recursive calls serially after reaching a maximum recursion depth. Listing 3.1 illustrates this approach. This maximum recursion depth is problem-dependent, and one needs to experiment with different values to find

the one that gives the best performance. A better approach is to use a pool of threads, which is explained in section 3.9.

```
if (recursionDepth < maxRecursionDepth) {
  // creating a thread for each recursive call
  // for computing low, high, and mid.
  std::thread t1([&]() {
    DnCMultiply(arraySize / 2, fLowStar, gLowStar, lowArray);
  });
  std::thread t2([&]() {
    DnCMultiply(arraySize / 2, fUpStar, gUpStar, highArray);
  });
  std::thread t3([&]() {
    DnCMultiply(arraySize / 2, fLowPlusUp, gLowPlusUp, midArray);
  });
  // wait for all threads to complete
  t1.join();
  t2.join();
  t3.join();
} else {
  // calculation of low, high, and mid sequentially.
  DnCMultiply(arraySize / 2, fLowStar, gLowStar, lowArray);
  DnCMultiply(arraySize / 2, fUpStar, gUpStar, highArray);
  DnCMultiply(arraySize / 2, fLowPlusUp, gLowPlusUp, midArray);
}
```

Listing 3.1: Threads are created in recursive divide-and-conquer if recursion depth is below a maximum value; otherwise, execution is sequential.

3.6.4 Protecting Shared Data Using Mutexes

If the shared data among threads is read-only, it does not need protection since multiple threads can read it simultaneously without affecting the data. However, if one or more threads start modifying the shared data, it could lead to a race condition and cause undefined behavior.

To avoid race conditions, a simple approach is to wrap the data structure with a protection mechanism that ensures only one thread making modifications can see the intermediate states of the data. For all other threads trying to access the data structure, these modifications either have not started yet or are completed. The C++ Standard Library offers several such mechanisms. The most basic and general one is the mutex.

A mutex (mutual exclusion) is a synchronization primitive used to protect shared data. A thread locks the mutex before accessing a shared data structure and unlocks it once it is done. The C++ Standard Thread Library ensures that when a thread locks a mutex, other threads attempting to lock the same mutex must wait until the mutex is unlocked.

A mutex can be created in C++ by constructing an instance of `std::mutex`. This mutex can be locked with a call to the `lock()` member function, and unlocked with a call

to the `unlock()` member function. It is, however, recommended to use `std::lock_guard` from the C++ Standard Library, which is based on the RAII idiom and is more exception-safe, instead of directly accessing a mutex. The `std::lock_guard` locks the supplied mutex on construction and unlocks it on destruction, ensuring that a locked mutex is unlocked when the scoped block is exited.

`std::unique_lock` is another locking mechanism provided by the C++ Standard Library that is more flexible than `std::lock_guard`. It allows for manual locking and unlocking, deferred locking, transfer of lock ownership, and use with condition variables. However, this flexibility comes with a slightly higher overhead.

`std::mutex`, `std::lock_guard`, and `std::unique_lock` are declared in the `<mutex>` library header.

It is noted that simply locking a mutex before accessing shared data and unlocking it afterwards does not guarantee data protection. If a member function returns a pointer or reference to the protected data, then any code that has access to that pointer or reference can modify the shared data even without acquiring the lock. Therefore, a careful interface design is needed to ensure that no uncontrolled code can modify the shared data.

3.6.5 Thread Synchronization Tools: Condition Variables, Futures, and Promises

Sometimes we want a thread to wait for a specific event to occur or a condition to be met. One approach is to have the waiting thread keep checking the state of a shared variable to see if it has been changed by another thread. In this approach, the waiting thread consumes a lot of processing resources by constantly checking the state of the variable. Also, when the waiting thread checks the state of the variable, it has to lock a mutex which keeps other threads from accessing the shared data. Another approach is to have the waiting thread sleep for specific intervals and then check the state of the variable upon waking. Although this method consumes less processing power, the waiting thread might wake up late, reducing the program's speed. The best way is that the thread that has made the condition satisfied, notify the thread(s) waiting for that event.

A basic synchronization primitive offered by the C++ Standard Library for waiting for an event to be triggered by another thread is the condition variable.

`std::condition_variable` is an implementation of a condition variable declared in the `<condition_variable>` header that is used with a `std::mutex` to block one or more threads until notified by another thread.

`wait()`, `notify_one()`, and `notify_all()` are member functions of the `std::condition_variable`. `wait()` blocks the current thread until notified. `notify_one()` unblocks one of the waiting threads, and `notify_all()` unblocks all of the waiting threads.

Sometimes a waiting thread might be unblocked by a spurious wakeup. To handle this, the `wait()` function can accept a second optional parameter which is a predicate that rechecks the condition to ensure it is truly met. This predicate can be in the form of a lambda function.

Let us say we want a thread `t1` to access a shared resource as soon as notified by

another thread `t2`. This scenario can happen in a parallel producer-consumer model, where a producer thread notifies a consumer thread(s) that a data item is ready to be consumed. Here are the steps involved:

1. Thread `t1` locks the mutex using a `std::unique_lock` (so that it can be unlocked and relocked later).
2. The `wait()` function blocks thread `t1` and unlocks the mutex (so that other thread(s) can acquire the lock).
3. Thread `t2` locks the mutex (typically using a `std::lock_guard`), modifies the shared data, unlocks the mutex (by exiting the scope), and notifies the condition variable using `notify_one()` or `notify_all()` functions.
4. The `wait()` function unblocks thread `t1` once it is notified and the condition is met.
5. Thread `t1` locks the mutex again and accesses the shared resource.

Listing 3.2 shows an implementation of the producer-consumer model which allows the producer to add data to a queue and the consumer to process data from the queue concurrently using `std::condition_variable`.

```
// producer-consumer model
std::mutex theMutex;
std::queue<dataType> dataQueue;
std::condition_variable condVar;
void dataProducer() {
  while(isDataAvailable()) {
    dataType data = prepareData();
    {
      std::lock_guard<std::mutex> lock(theMutex);
      dataQueue.push(data);
    }
    condVar.notify_one();
  }
}
void dataConsumer() {
  while(!isLastData(data)) {
    std::unique_lock<std::mutex> lock(theMutex);
    condVar.wait(lock, []{return !dataQueue.empty();});
    dataType data = dataQueue.front();
    dataQueue.pop();
    lock.unlock();
    process(data);
  }
}
```

Listing 3.2: A concurrent producer-consumer model implementation.

A more abstracted approach to handle asynchronous operations is to use `std::future` and `std::async`.

`std::future` is a class template that allows a program to access the result of an asynchronous operation. It acts as a placeholder for the eventual result of a computation that

may not be available immediately. The `std::future::get()` function waits until the future object has a valid result and retrieves it. The `std::future::wait_for(duration)` function blocks until either the specified `duration` has elapsed or the result becomes available. It returns an `enum` of type `std::future_status` indicating the state of the result:

- `std::future_status::ready` means that the asynchronous result is ready.
- `std::future_status::timeout` indicates that the specified duration has passed, and the result is not ready.
- `std::future_status::deferred` specifies that the asynchronous result contains a deferred function that has not started yet.

`std::async` is a function template that runs a function asynchronously (potentially with a new thread) in the background and returns a `std::future` object that can be used to retrieve the result. When the result is needed, a call to `std::future::get()` blocks the thread until the result is available, and then returns it. `std::future` and `std::async` are declared in the `<future>` header.

A simple usage of `std::async` with `std::future` is shown in Listing 3.3, where `std::async` launches a task in the background and returns a `std::future<int>`.

```
auto future = std::async(std::launch::async, [] {
    // perform some computation
    return result;
});

int result = future.get();
```

Listing 3.3: A simple usage of `std::async` with `std::future`.

A lower level mechanism for decoupling the execution of tasks from their eventual result is to use `std::promise` which is a class template that allows a value to be set in the future and accessed via an associated `std::future` object. `std::promise` and `std::future` provide a means of communication between threads in a producer-consumer manner. One thread creates a `std::promise` and obtains a `std::future` from it. The `std::promise` and function arguments are then moved into another thread, where the function executes and fulfills the promise by setting its value. The original thread can wait for and retrieve the result using the `std::future` object.

Listing 3.4 shows an example where `std::promise` is used to set a value from another thread, which is then retrieved via the associated `std::future` object.

```
std::promise<int> promise;
std::future<int> future = promise.get_future();

// a thread that sets the value
std::thread([&promise] {
    // perform some computation
    promise.set_value(someValue);
}).detach();

// another thread that blocks until the value is set
```

```
11 int result = future.get();
```

Listing 3.4: Synchronizing two threads, where one thread sets a value and the other one waits for the value to be set using `std::promise` and `std::future`.

3.6.6 Atomic Operations and Atomic Types

Atomic operations and atomic types provide facilities for low-level and lockless synchronization operations.

An atomic operation on an object is an indivisible operation, i.e. other threads can only see the state of the object either before the operation starts or after it finishes and cannot see any intermediate states. Atomic types are defined in `<atomic>` library header. `std::atomic<>` is the template class for such types. All operations on these types are atomic.

Listing 3.5 shows a simple example of using atomic operations to increment a counter. Multiple threads can concurrently execute the `incrementCounter()` function without the need for explicit locks.

```
1 std::atomic<int> counter(0);
2
3 void incrementCounter() {
4   ++counter;
5 }
```

Listing 3.5: Incrementing a shared counter using atomic operations.

3.7 Data Structures in C++ Containers: Stack, Queue, Deque

The `std::stack` class, defined in the `<stack>` library header, is a container adapter providing the functionality of a stack. A stack is a data structure to store a collection of elements in a LIFO (last-in, first-out) manner, that is, the last element added is the first one to be removed. Figure 3.6 shows a stack data structure.

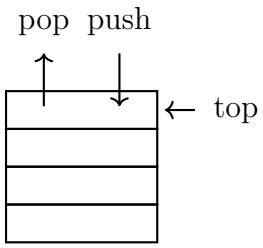

Figure 3.6: Illustration of a stack data structure.

The `std::queue` class, defined in the `<queue>` library header, is a container adapter providing the functionality of a queue, that is, a FIFO (first-in, first-out) data structure. The `push()` function inserts an element at the back of the queue, and the `pop()` function removes the front element from the queue. The `front()` and `back()` functions access the front and last element respectively without removing them. The `empty()` function checks if the queue is empty. A schematic diagram of a queue data structure is shown in Figure 3.7.

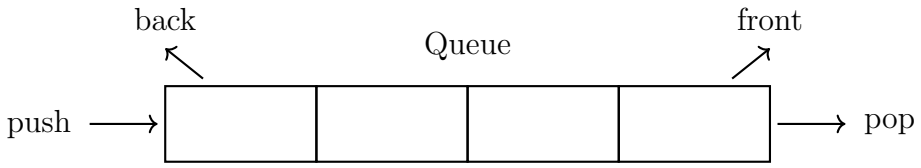

Figure 3.7: Illustration of a queue data structure.

The `std::deque` (double-ended queue), defined in the `<deque>` header, is a sequence container with a dynamic size that can be expanded or contracted on both ends (front or back). A deque allows insertion and removal of elements from both ends efficiently. The `push_front()` function adds an element to the front of the deque, and the `push_back()` function adds an element to the back of the deque. The `pop_front()` and `pop_back()` functions remove the first and last elements of the deque respectively. The `front()` and `back()` functions access the first and last elements of the deque respectively without removing it. The `empty()` function checks if the deque is empty. Figure 3.8 shows a deque data structure.

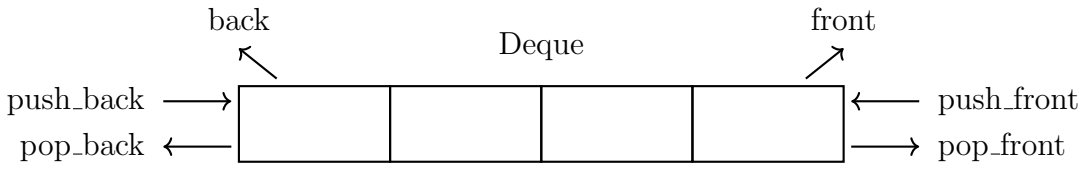

Figure 3.8: Illustration of a deque data structure.

It is noted that the `std::stack`, `std::queue`, and `std::deque` from the C++ Standard Library are not thread-safe, and to achieve thread-safety, we need to manage synchronization ourselves.

3.8 Cache Complexity, Data Locality, and Data Contention

Cache complexity and data locality play a crucial role in the performance of multithreaded programs.

Cache memory is smaller and faster than main memory and is located closer to the CPU. It stores copies of frequently accessed data from main memory to speed up data retrieval. Modern processors have multiple levels of cache (L1, L2, L3), with L1 being the smallest and fastest, and L3 being the largest and slowest. Memory is divided into cache lines, and when data is accessed, the entire cache line containing the data is loaded into the cache. When the CPU needs to access a data item, it first checks the cache. If the requested data item can be found in the cache, a cache hit occurs, otherwise, a cache miss occurs, and the CPU must retrieve the data from the slower main memory.

Cache misses can be categorized into several types: cold miss (when the first time data is accessed), capacity miss (when data previously accessed is evicted because the working data set is too large), and conflict miss (when multiple data items are mapped to the same cache location, leading to eviction before the cache is full). Generally, data locality improves cache performance by keeping frequently accessed data close together (spatial locality) and reusing data within short periods (temporal locality). In multithreaded programs, true sharing and false sharing misses are also introduced.

If two or more threads from different processors try to read the same data, the data will be copied into their respective caches, and the processors can work on the data. However, if at least one of the threads modifies the data, this change has to be propagated to the cache of the other processors before the processors can proceed. This scenario, where multiple threads access and modify the same data, is known as true sharing. For example, if two threads keep accessing a global counter variable and incrementing it, the value of the counter must be passed back and forth between the two processors' caches so that the thread from each processor has the most recent value of the counter variable before it increments. This back and forth transfer of data between caches, which is called "cache ping-pong", can significantly impact the performance of the application since a processor waiting for a cache transfer cannot do any work in the meantime [Wil19]. One way to mitigate this issue is by minimizing data sharing among threads. If too many processors want to increment the counter variable, they might find themselves mostly waiting for each other to update the counter variable and propagating this change. This situation is referred to as "high contention".

False sharing happens when threads from different processors access different variables on the same cache line. When a thread modifies a variable in its cache line, the entire cache line is updated, even if the other variables in that cache line remain unchanged. This is because the cache hardware only operates in cache-line-sized blocks of memory.

False sharing leads to unnecessary cache traffic and overhead. This problem can be reduced by padding data structures so that each thread works on variables apart from each other in different cache lines, preventing interference.

3.9 Thread Pool Design Pattern

A thread pool is a concurrency design pattern used to avoid the overhead of creating and destroying threads for each task and prevent oversubscription [GS01, Hol00].

In a thread pool, the number of created pool threads are fixed and the same threads are reused to perform different tasks throughout the program lifetime. By setting the

number of pool threads created at the beginning of the program to a fixed value less than or equal to the maximum number of threads the hardware can support, oversubscription is avoided. Choosing the optimal number of pool threads which is referred to as the pool size is crucial for program performance [LML00].

3.10 Work Stealing Strategy for Scheduling

Two dynamic load balancing strategies can be used to schedule multithreaded computations: work sharing and work stealing. In work sharing, whenever new work items are created on a thread, the scheduler attempts to migrate some of them to other threads to distribute the work. In work stealing, however, the idle threads, i.e., the threads with no work to do, take the initiative and attempt to "steal" work from other threads. Work stealing involves less work migration compared to work sharing, because contrary to a a work stealing scheduler, a work sharing scheduler still does work migration even if all threads have work to do [BL99].

The idea of work stealing dates back to the implementation of Multilisp and parallel execution of functional programs [BL99, BS81, 10.84]. Now, work stealing is employed in the scheduler for the Cilk programming language [BJK+95a], the OpenMP [DM98], the Java fork/join framework [Lea00], the .NET Task Parallel Library [LSB09], and the Intel's Threading Building Blocks (TBB) [MRR12b].

The basic notion of work stealing is as follows. Each thread maintains a queue of work items, i.e. a set of instructions, to execute. A work item might create new work items during its execution which are initially put on the queue of the same thread. When a thread runs out of work, it checks the queues of other threads and steals a piece of work from them. In an advanced design, each thread maintains a double-ended queue or a deque which allows for insertion and removal of work items from both ends, i.e. top and bottom. New work items are added to the top of the deque, and threads execute tasks by removing them from the top. A thread that has run out of work steals from the bottom of other threads' deques. The work stealing algorithm distributes the scheduling work over the threads, and there will be no scheduling overhead if all threads have work to do [KLJ14]. Figure 3.9 shows an illustration of a thread pool with work stealing mechanism.

3.11 Thread Pool Implementation

3.11.1 Basic Thread Pool Functionalities

At its most basic implementation, a thread pool class launches a fixed number of worker threads in its constructor to process work. The callable object given to the threads upon creation takes off work from a queue of tasks, executes the task, and goes back to the queue for more work. The thread pool class also has a function to enqueue new tasks. The destructor ensures that all threads are joined before the thread pool object is destroyed.

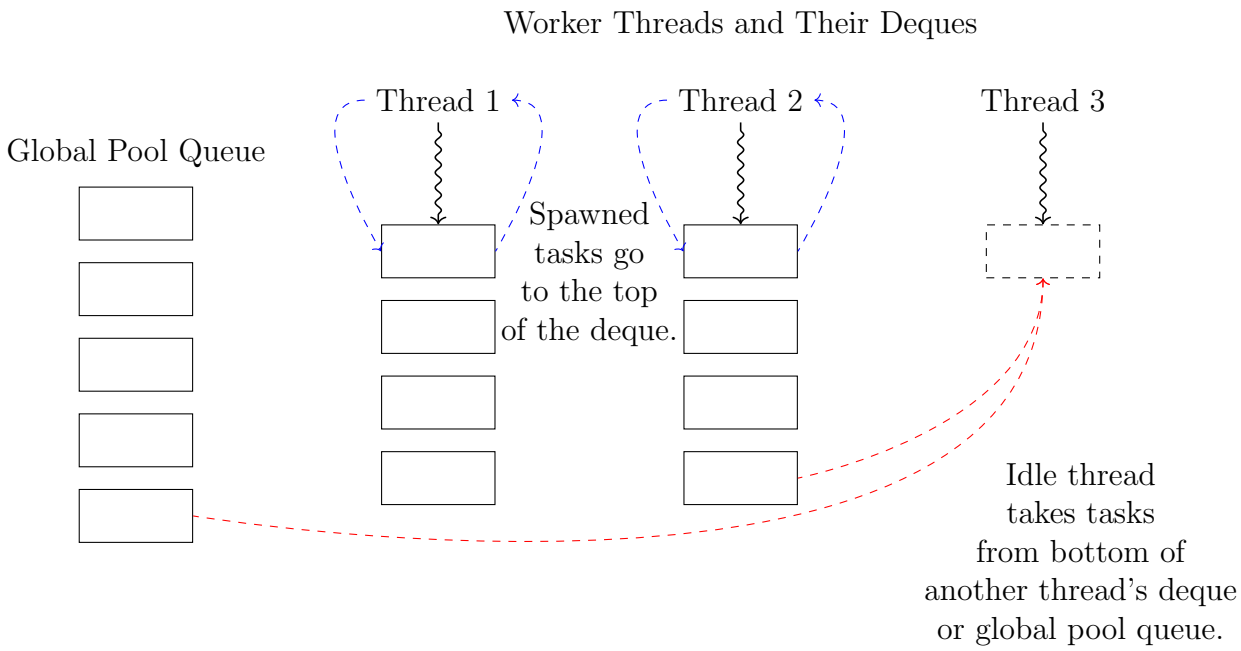

Figure 3.9: A thread pool with work stealing mechanism. Rectangles represent tasks. Each worker thread has its own double-ended queue (deque) for storing tasks. Workers execute tasks in LIFO order, adding new tasks to the top of their deque. When a worker's deque is empty, it steals tasks from the bottom of another worker's deque or from a global queue.

3.11.2 Thread-Safe Queue Data Structure

A fundamental part of implementing a thread pool is to construct a thread-safe queue.

Listing 3.6 shows an implementation of a thread-safe queue that uses a `std::mutex` to protect the `std::queue<T>` from concurrent access. It also uses a `std::condition_variable` to manage the synchronization between threads.

```
#include <queue>
#include <mutex>
#include <condition_variable>
#include <memory>

template<typename T>
class ThreadSafeQueue {
private:
  mutable std::mutex theMutex;
  std::queue<T> dataQueue;
  std::condition_variable condVar;
public:
  ThreadSafeQueue() {}
  void push(T newItem) {
    std::lock_guard<std::mutex> lk(theMutex);
    dataQueue.push(std::move(newItem));
    condVar.notify_one();
  }
  void waitAndPop(T& value) {
    std::unique_lock<std::mutex> lk(theMutex);
    condVar.wait(lk,[this]{return !dataQueue.empty();});
    value=std::move(dataQueue.front());
    dataQueue.pop();
  }
  std::shared_ptr<T> waitAndPop() {
    std::unique_lock<std::mutex> lk(theMutex);
    condVar.wait(lk,[this]{return !dataQueue.empty();});
    std::shared_ptr<T> result(
      std::make_shared<T>(std::move(dataQueue.front())));
    dataQueue.pop();
    return result;
  }
  bool tryPop(T& value) {
    std::lock_guard<std::mutex> lk(theMutex);
    if(dataQueue.empty())
      return false;
    value=std::move(dataQueue.front());
    dataQueue.pop();
  }
  std::shared_ptr<T> tryPop() {
    std::lock_guard<std::mutex> lk(theMutex);
```

```
42      if(dataQueue.empty())
43        return std::shared_ptr<T>();
44      std::shared_ptr<T> result(
45        std::make_shared<T>(std::move(dataQueue.front())));
46      dataQueue.pop();
47      return result;
48    }
49    bool empty() const {
50      std::lock_guard<std::mutex> lk(theMutex);
51      return dataQueue.empty();
52    }
53  };
```

Listing 3.6: A thread-safe queue implementation.

In Listing 3.6, the default constructor `ThreadSafeQueue()` initializes an empty queue.

The `push()` method adds a new element to the queue. This method locks the mutex to ensure exclusive access to the queue while the new element is added. After pushing the element, it calls `notify_one()` on the condition variable to wake up one of the waiting threads, signaling that new data is available.

The `waitAndPop()` methods provide a mechanism for threads to wait until an element is available in the queue before removing and returning it.

The `tryPop()` methods attempt to remove and return an element from the queue without blocking.

The `empty()` method checks if the queue is empty.

It it noted that in this implementation, the `waitAndPop()` methods are useful in scenarios where the consumer thread should wait for data to become available rather than repeatedly checking the queue. Since the `waitAndPop()` methods use a condition variable, they avoid busy-waiting and save CPU cycles by putting the thread to sleep until data is available. On the other hand, the `tryPop()` methods provide a non-blocking behavior and are useful in scenarios where the consumer thread has other tasks to perform and does not block if no data is available.

The `std::shared_ptr<T>` overloads of the `waitAndPop()` and the `tryPop()` methods use a shared pointer, which handles memory management automatically, and do not require the caller to manage the lifetime of the popped value. However, they incur slightly more overhead.

The `std::move` in the `waitAndPop()` and `tryPop()` methods transfers ownership of resources (such as dynamically allocated memory) directly to the new object without duplicating them.

3.11.3 A Basic Thread Pool Implementation

Listing 3.8 shows an implementation of a basic thread pool.

```
1  #include <thread>
2  #include <vector>
3  #include <atomic>
```

```
#include <functional>

class JoinThreads {
  std::vector<std::thread>& threads;
public:
explicit JoinThreads(std::vector<std::thread>& threadsParam):
  threads(threadsParam) {}
  ~JoinThreads() {
    for(unsigned long i = 0; i < threads.size(); ++i) {
      if(threads[i].joinable())
        threads[i].join();
    }
  }
};

class ThreadPool {
  std::atomic_bool done;
  ThreadSafeQueue<std::function<void()> > workQueue;
  std::vector<std::thread> threads;
  JoinThreads joiner;
  void workerThread() {
    while (!done) {
      std::function<void()> task;
      if (workQueue.tryPop(task)) {
        task();
      } else {
        std::this_thread::yield();
      }
    }
  }
public:
  ThreadPool() : done(false), joiner(threads) {
        unsigned const threadCount = std::thread::
           hardware_concurrency();
    try {
      for (unsigned i = 0; i < threadCount; ++i) {
        threads.push_back(
          std::thread(&ThreadPool::workerThread, this));
      }
    } catch(...) {
      done = true;
      throw;
    }
  }
  ~ThreadPool() {
    done = true;
  }
```

```
50    template<typename FunctionType>
51    void submitTask(FunctionType f) {
52      workQueue.push(std::function<void()>(f));
53    }
54  };
```

Listing 3.7: A basic thread pool implementation.

This thread pool uses the thread-safe queue from Listing 3.6 to store tasks. The tasks are encapsulated using the `std::function<void()>`. The `submitTask()` function enables users to add new tasks to the queue.

The thread objects are stored in the `std::vector<std::thread>` container. The `JoinThreads` class ensures that all threads are properly joined before the thread pool is destroyed.

`done` is an atomic boolean flag which indicates when the thread pool should stop processing tasks.

The constructor initializes the done flag to false, sets up the joiner, and starts the worker threads.
The number of threads is determined by `std::thread::hardware_concurrency()`.

The destructor sets the `done` flag to `true`, signaling all worker threads to stop processing tasks.

The `workerThread()` function is the main loop for each worker thread. It continuously checks the task queue for new tasks and executes them. If no tasks are available, the thread yields to allow other threads to run.

3.11.4 Waiting on Tasks Completion

In parallel divide and conquer algorithms, the results of computations done by worker threads are needed by the main thread. Generally, when threads are explicitly spawned, the main thread waits for them to finish before returning to the caller. With thread pools, the main thread waits for submitted tasks to complete rather than waiting for individual worker threads to finish. The thread pool implementation in Listing 3.8 does not support waiting for tasks to complete or retrieving return values. To enable this functionality, one can use futures to wait for a task to complete and pass the resulting value to the waiting thread.

The class template `std::packaged_task` from the `<future>` header provides a means to associate a `std::future` with a task. An instance of the `std::packaged_task` wraps a callable target and allows it to be executed asynchronously. The return value is stored in a shared state that can be accessed through `std::packaged` objects. Listing 3.8 shows an example of using `std::packaged_task` with `std::future`.

```
1  int add(int a, int b) {
2      return a + b;
3  }
4  int main() {
5    std::packaged_task<int(int, int)> task(add);
6    std::future<int> future = task.get_future();
```

```
  std::thread t(std::move(task), 1, 1);
  int result = future.get();
  t.join();
  return 0;
}
```

Listing 3.8: An example of using `std::packaged_task` to execute a function asynchronously and retrieve the result using `std::future`.

Instances of the `std::packaged_task` are only movable and not copyable. If we want to use `std::packaged_task` in a thread pool queue, we cannot use `std::function` to hold queue elements because `std::function` requires that the stored functions are copy-constructible. Hence, we need to use a custom function wrapper that handles move-only types. Listing 3.9 shows an example of such a wrapper [Wil19].

```
#include <future>
#include <memory>
#include <functional>

class FunctionWrapper {
  struct implBase {
    virtual void call() = 0;
    virtual ~implBase() {}
  };
  std::unique_ptr<implBase> impl;
  template <typename F>
  struct implType : implBase {
    F f;
    implType(F&& fParam) : f(std::move(fParam)) {}
    void call() {f();}
  };
public:
  template <typename F>
  FunctionWrapper(F&& f):
    impl(new implType<F>(std::move(f))) {}

  void call() {impl->call();}

  FunctionWrapper(FunctionWrapper&& other):
    impl(std::move(other.impl)) {}

  FunctionWrapper& operator=(FunctionWrapper&& other) {
    impl = std::move(other.impl);
    return *this;
  }
  FunctionWrapper(const FunctionWrapper&) = delete;
  FunctionWrapper(FunctionWrapper&) = delete;
  FunctionWrapper& operator = (const FunctionWrapper&) = delete;
};
```

Listing 3.9: An example of a wrapper for movable-only callable objects.

Listing 3.10 presents a modified version of the basic thread pool in Listing 3.8. This modified version waits for tasks to complete and passes the return values from the tasks to the waiting thread.

```
1  class ThreadPool {
2    std::atomic_bool done;
3    ThreadSafeQueue<FunctionWrapper> workQueue;
4    std::vector<std::thread> threads;
5    JoinThreads joiner;
6
7    void workerThread() {
8      while (!done) {
9        FunctionWrapper task;
10       if (workQueue.tryPop(task)) {
11         task();
12       }
13       else {
14         std::this_thread::yield();
15       }
16     }
17   }
18 public:
19   ThreadPool() : done(false), joiner(threads) {
20     unsigned const threadCount = std::thread::
          hardware_concurrency();
21     try {
22       for (unsigned i = 0; i < threadCount; ++i) {
23         threads.push_back(
24           std::thread(&ThreadPool::workerThread, this));
25       }
26     }
27     catch (...) {
28       done = true;
29       throw;
30     }
31   }
32   ~ThreadPool() {
33     done = true;
34   }
35   template<typename FunctionType>
36   std::future<typename std::result_of<FunctionType()>::type>
37     submitTask(FunctionType f) {
38     typedef typename std::result_of<FunctionType()>::type
          resultType;
39
```

```
    std::packaged_task<resultType()> task(std::move(f));
    std::future<resultType> result(task.get_future());
    workQueue.push(FunctionWrapper(std::move(task)));
    return result;
  }
};
```

Listing 3.10: A thread pool with waitable tasks.

The `submitTask()` function has been modified so that it returns a `std::future` to hold the return value of the task and enable the caller to wait for the task to complete. The task is wrapped in an instance of the `std::packaged_task` and the future is obtained before the task is added to the queue. The queue elements are of type `FunctionWrapper` presented in Listing 3.9 instead of `std::function` because the instances of the `std::packaged_task` are not copyable.

3.11.5 A Separate Queue for Every Thread

If a thread pool has only one queue for tasks, such as the thread pool in Listing 3.10, then the threads that submit the tasks to the queue and the threads that pop off the tasks from the queue and run them access and modify the same data structure. This will create contention on the queue and cache ping-pong. To avoid this, we create a separate queue per thread and one global queue. Each worker thread of the pool adds new tasks to its own queue and takes off tasks from its own queue. A worker thread only removes tasks from the global queue if its own queue is empty. Threads that do not belong to the pool add tasks to the global queue. An implementation of such a thread pool is shown in Listing 3.11.

```
class ThreadPool {
  std::atomic_bool done;
  ThreadSafeQueue<FunctionWrapper> poolQueue;
  typedef std::queue<FunctionWrapper> localQueueType;
  static thread_local std::unique_ptr<localQueueType> localQueue;
  std::vector<std::thread> threads;
  JoinThreads joiner;

  void workerThread() {
    localQueue.reset(new localQueueType);
    while (!done) {
      runPendingTask();
    }
  }
public:
  ThreadPool() : done(false), joiner(threads) {
    unsigned const threadCount = std::thread::
       hardware_concurrency();
    try {
      for (unsigned i = 0; i < threadCount; ++i) {
```

```
20          threads.push_back(
21            std::thread(&ThreadPool::workerThread, this));
22        }
23      }
24      catch (...) {
25        done = true;
26        throw;
27      }
28    }
29    ~ThreadPool() {
30      done = true;
31    }
32    template<typename FunctionType>
33    std::future<typename std::result_of<FunctionType()>::type>
34      submitTask(FunctionType f) {
35      typedef typename std::result_of<FunctionType()>::type
           resultType;
36
37      std::packaged_task<resultType()> task(std::move(f));
38      std::future<resultType> result(task.get_future());
39      if(localQueue) {
40        localQueue->push(std::move(task));
41      } else {
42        poolQueue.push(std::move(task));
43      }
44      return result;
45    }
46    void runPendingTask() {
47      FunctionWrapper task;
48      if(localQueue && !localQueue->empty()) {
49        task = std::move(localQueue->front());
50        localQueue->pop();
51        task();
52      } else if(poolQueue.tryPop(task)) {
53        task();
54      } else {
55        std::this_thread::yield();
56      }
57    }
58  };
59
60  thread_local std::unique_ptr<ThreadPool::localQueueType>
       ThreadPool::localQueue = nullptr;
```

Listing 3.11: A thread pool with local queues for every pool thread and one global queue.

Although the thread pool presented in Listing 3.11 minimizes contention, it lacks a system for efficient load balancing among threads. This can lead to a scenario where one

thread is overwhelmed with tasks while another remains idle. To achieve proper load balancing, a work-stealing algorithm can be implemented, allowing threads to dynamically redistribute tasks by taking work from each other.

3.11.6 Work Stealing Using a Double-Ended Queue

Work stealing is an efficient way of load balancing and dynamic distribution of work in a thread pool. To enable this technique, each worker thread maintains its own double-ended queue (deque) of tasks. Worker threads push tasks to the front and pop tasks from the front of their deque. When a thread runs out of tasks in its own deque, instead of becoming idle, it tries to take tasks from the back of the deques of other threads. This means that the deque works as a LIFO stack for the thread it belongs to, where the most recently added task is the first to be executed. This LIFO behavior can enhance performance due to cache locality since the data associated with the most recently pushed task is more likely to remain in the cache.

To allow threads to take tasks from each other, their deque must be accessible to each other, and the data in the deque must be protected with a mechanism such as a mutex. Listing 3.12 shows an implementation of a thread-safe deque suitable for work stealing.

```
class WorkStealingQueue {
private:
  typedef FunctionWrapper dataType;
  std::deque<dataType> theQueue;
  mutable std::mutex theMutex;
public:
  WorkStealingQueue() {}
  WorkStealingQueue(const WorkStealingQueue& other) = delete;
  WorkStealingQueue& operator=(const WorkStealingQueue& other) =
     delete;
  void push(dataType data) {
    std::lock_guard<std::mutex> lock(theMutex);
    theQueue.push_front(std::move(data));
  }
  bool empty() const {
    std::lock_guard<std::mutex> lock(theMutex);
    return theQueue.empty();
  }
  bool tryPop(dataType& result) {
    std::lock_guard<std::mutex> lock(theMutex);
    if(theQueue.empty()) {
      return false;
    }
    result = std::move(theQueue.front());
    theQueue.pop_front();
    return true;
  }
  bool trySteal(dataType& result) {
```

```
    std::lock_guard<std::mutex> lock(theMutex);
    if(theQueue.empty()) {
      return false;
    }
    result = std::move(theQueue.back());
    theQueue.pop_back();
    return true;
  }
};
```

Listing 3.12: An implementation of a thread-safe deque.

3.11.7 Thread Pool With Work Stealing Implementation

Listing 3.13 shows an implementation of a thread pool that uses work stealing mechanism for load balancing. This thread pool uses thread-safe deques from Listing 3.12.

```
class ThreadPool {
  typedef FunctionWrapper taskType;
  std::atomic_bool done;
  ThreadSafeQueue<taskType> poolQueue;
  std::vector<std::unique_ptr<WorkStealingQueue>> queues;
  std::vector<std::thread> threads;
  JoinThreads joiner;

  static thread_local WorkStealingQueue* localWorkQueue;
  static thread_local unsigned myIndex;

  void workerThread(unsigned myIndex_) {
    myIndex = myIndex_;
    localWorkQueue = queues[myIndex].get();
    while (!done) {
      runPendingTask();
    }
  }

  bool popTaskFromLocalQueue(taskType& task) {
    return localWorkQueue && localWorkQueue->tryPop(task);
  }

  bool popTaskFromPoolQueue(taskType& task) {
    return poolQueue.tryPop(task);
  }

  bool popTaskFromOtherThreadQueue(taskType& task) {
    for (unsigned i = 0; i < queues.size(); ++i) {
      unsigned const index = (myIndex + i + 1) % queues.size();
      if (queues[index]->trySteal(task)) {
```

```
32          return true;
33        }
34      }
35      return false;
36    }
37
38  public:
39    ThreadPool():
40      joiner(threads), done(false) {
41      unsigned const threadCount = std::thread::
           hardware_concurrency();
42
43      try {
44        for (unsigned i = 0; i < threadCount; ++i) {
45          queues.push_back(std::unique_ptr<WorkStealingQueue>(
46            new WorkStealingQueue));
47        }
48        for (unsigned i = 0; i < threadCount; ++i) {
49          threads.push_back(
50            std::thread(&ThreadPool::workerThread, this, i));
51        }
52      } catch (...) {
53        done = true;
54        throw;
55      }
56    }
57
58    ~ThreadPool() {
59      done = true;
60    }
61
62    template<typename FunctionType>
63    std::future<typename std::result_of<FunctionType()>::type>
         submitTask(FunctionType f) {
64      typedef typename std::result_of<FunctionType()>::type
           resultType;
65      std::packaged_task<resultType()> task(f);
66      std::future<resultType> result(task.get_future());
67      if (localWorkQueue) {
68        localWorkQueue->push(std::move(task));
69      } else {
70        poolQueue.push(std::move(task));
71      }
72      return result;
73    }
74
75    void runPendingTask() {
```

```
76      taskType task;
77      if (popTaskFromLocalQueue(task) ||
78        popTaskFromPoolQueue(task) ||
79        popTaskFromOtherThreadQueue(task)) {
80        task();
81      } else {
82        std::this_thread::yield();
83      }
84    }
85  };
86
87  thread_local WorkStealingQueue* ThreadPool::localWorkQueue =
       nullptr;
88  thread_local unsigned ThreadPool::myIndex = 0;
```

Listing 3.13: A thread pool implementation with work stealing mechanism for load balancing.

3.12 Multithreaded Implementation of Karatsuba Algorithm

The Karatsuba method for multiplying univariate polynomials, explained in section 2.1, is a recursive divide-and-conquer algorithm that can be parallelized using the fork-join model. In this section, we present how we can parallelize the Karatsuba algorithm implemented in Listing 2.5 using C++ multithreading and the created thread pool in Listing 3.13.

We note that other ways of parallelizing Karatsuba algorithm in the fork-join model have been reported in the literature. For instance, in [CMPS10], the authors report on a parallel and in-place versions of Karatsuba and Toom-Cook algorithms for multiplying univariate polynomials. In the case of Karatsuba, this is done by slightly increasing the span from $O(n)$ to $O(n\log(n))$ while reducing the space usage from $O(n^{\log_2(3)})$ to $O(n)$. In the case of Toom-Cook, this is done without increasing the span, while reducing the space usage from $O(n^{\log_3(5)})$ to $O(n)$.

Listing 3.14 shows a multithreaded implementation of the Karatsuba algorithm. A thread pool instance is created in the caller function and passed by reference to the recursive divide-and-conquer `DnCMultiply()` function, which calls itself three times.

```
1
2  void DnCMultiply(int indSeriesSize, mpq_t *f, mpq_t *g, mpq_t *
       product, ThreadPool &pool) {
3    // if indSeriesSize <= Threshold, we don't apply DnCMultiply,
4    // we perform ordinary multiplication
5    if (indSeriesSize <= DnCThreshold) {
6      ordinaryMultiplication(indSeriesSize, f, g, product);
7    } else {
8      mpq_t *fLowStar = f;
```

```
    mpq_t *gLowStar = g;
    mpq_t *fUpStar = f + indSeriesSize / 2;
    mpq_t *gUpStar = g + indSeriesSize / 2;
    mpq_t *fLowPlusUp = new mpq_t[indSeriesSize / 2];
    mpq_t *gLowPlusUp = new mpq_t[indSeriesSize / 2];
    for (int i = 0; i < indSeriesSize / 2; ++i) {
      mpq_init(fLowPlusUp[i]);
      mpq_init(gLowPlusUp[i]);

      mpq_add(fLowPlusUp[i], f[i], f[indSeriesSize / 2 + i]);
      mpq_add(gLowPlusUp[i], g[i], g[indSeriesSize / 2 + i]);
    }
    mpq_t *lowArray = new mpq_t[indSeriesSize - 1];
    mpq_t *midArray = new mpq_t[indSeriesSize - 1];
    mpq_t *highArray = new mpq_t[indSeriesSize - 1];
    for (int i = 0; i < indSeriesSize - 1; ++i) {
      mpq_init(lowArray[i]);
      mpq_init(midArray[i]);
      mpq_init(highArray[i]);
    }

    // Parallel execution
    std::future<void> lowFuture = pool.submitTask([&] {
      DAC(indSeriesSize / 2, fLowStar, gLowStar, lowArray, pool,
          depth + 1);
    });
    std::future<void> highFuture = pool.submitTask([&] {
      DAC(indSeriesSize / 2, fUpStar, gUpStar, highArray, pool,
          depth + 1);
    });
    std::future<void> midFuture = pool.submitTask([&] {
      DAC(indSeriesSize / 2, fLowPlusUp, gLowPlusUp, midArray,
          pool, depth + 1);
    });

    while ((lowFuture.wait_for(std::chrono::seconds(0)) ==
      std::future_status::timeout) ||
      (highFuture.wait_for(std::chrono::seconds(0)) ==
      std::future_status::timeout) ||
      (midFuture.wait_for(std::chrono::seconds(0)) ==
      std::future_status::timeout)) {
      pool.runPendingTask();
    }

    // Wait for tasks to complete
    lowFuture.get();
    highFuture.get();
```

```
53      midFuture.get();
54
55      for (int i = 0; i < indSeriesSize - 1; ++i) {
56        mpq_sub(midArray[i], midArray[i], lowArray[i]);
57        mpq_sub(midArray[i], midArray[i], highArray[i]);
58      }
59      // assemble product array from low, high, mid - low - high
60      for (int i = 0; i < indSeriesSize - 1; ++i)
61        mpq_add(product[i], product[i], lowArray[i]);
62      for (int i = 0; i < indSeriesSize - 1; ++i)
63        mpq_add(product[indSeriesSize / 2 + i],
64        product[indSeriesSize / 2 + i], midArray[i]);
65      for (int i = 0; i < indSeriesSize - 1; ++i)
66        mpq_add(product[indSeriesSize + i],
67          product[indSeriesSize + i], highArray[i]);
68
69      // memory deallocation
70      for (int i = 0; i < indSeriesSize / 2; ++i) {
71        mpq_clear(fLowPlusUp[i]);
72        mpq_clear(gLowPlusUp[i]);
73      }
74      for (int i = 0; i < indSeriesSize - 1; ++i) {
75        mpq_clear(lowArray[i]);
76        mpq_clear(midArray[i]);
77        mpq_clear(highArray[i]);
78      }
79      delete[] fLowPlusUp;
80      fLowPlusUp = nullptr;
81      delete[] gLowPlusUp;
82      gLowPlusUp = nullptr;
83      delete[] lowArray;
84      lowArray = nullptr;
85      delete[] midArray;
86      midArray = nullptr;
87      delete[] highArray;
88      highArray = nullptr;
89    }
90  }
```

Listing 3.14: Multithreaded implementation of the Karatsuba polynomial multiplication.

3.13 Multithreaded Implementation of Inverting a Lower Triangular Matrix

Another recursive divide-and-conquer problem encountered in the multiplication of power series, which can be parallelized using the fork-join model, is the inversion of a lower

triangular matrix.

In Section 5.1, we introduce an evaluation-interpolation strategy for multiplying power series. This strategy requires computing the interpolation matrix (M_{interp}), which is the inverse of the evaluation matrix (M_{eval}). We do the inversion in the following steps. First, we perform LU factorization on M_{eval} to decompose it into a lower triangular matrix L and an upper triangular matrix U as shown in Equation 3.6. Then, we transpose U to make it a lower triangular matrix and apply our recursive divide-and-conquer matrix inversion function to L and U^T. Finally, we use Equation 3.7 to obtain M_{interp}.

$$M_{\text{eval}} = L \times U \tag{3.6}$$

$$M_{\text{interp}} = \left((U^T)^{-1}\right)^T \times L^{-1} \tag{3.7}$$

Now, we describe the method to invert a lower triangular matrix. If A is an $n \times n$ lower triangular matrix with all non-zero diagonal elements, it is invertible. Assuming that n is a power of 2, computing the inverse A^{-1} of A can be done using a divide-and-conquer strategy as follows. Let A be partitioned into $n/2 \times n/2$ blocks as follows:

$$A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}. \tag{3.8}$$

Then, the matrix A^{-1} is given by:

$$A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix}. \tag{3.9}$$

Algorithm 6 shows the process of computing A^{-1}. This is a recursive divide-and-conquer algorithm with two independent inner calls that can be parallelized using the fork-join model.

Listing 1 in the Appendix shows a multithreaded implementation of inverting a lower triangular matrix described in Algorithm 6.

3.14 Performance Analysis of Multithreading

In this section, we evaluate our multithreaded implementations of the static Karatsuba and relaxed multiplication methods using the developed thread pool with a work-stealing scheduler in Subsection 3.11.7 (see Listing 3.14) and compare the performance results with the serial implementation (see Listing 2.5) and multithreading with Cilk (see Listing 3.15). These tests were run on a system with the specifications listed in Table 2.2. It can be observed that for large values of the product precision, the speedup is around 3.

3.14.1 Parallelization with Cilk

Cilk, originally developed at MIT, is a language extension to the C and C++ programming languages that enables multithreaded parallel computing by adding simple constructs. The `cilk_spawn` keyword preceding a function call signals that the function can

Algorithm 6 Computing A^{-1} Using the Fork-Join Model

Input: Lower Triangular Matrix A of size $n \times n$, Threshold B.
Output: Matrix A^{-1}.

1: **function** MatrixInverse(A, n)
2: **if** $n > B$ **then**
3: Partition A into A_1 (top-left), A_2 (bottom-left), and A_3 (bottom-right) quarters.
4: Spawn threads.
5: $A_{InvTopLeft} = \text{MatrixInverse}(A_1, n/2)$
6: $A_{InvBottomRight} = \text{MatrixInverse}(A_3, n/2)$
7: Synchronize threads.
8: $A_{InvBottomLeft} = -A_{InvBottomRight} \times A_2 \times A_{InvTopLeft}$
9: Assemble A^{-1} from $A_{InvTopLeft}$, $A_{InvBottomLeft}$, $A_{InvBottomRight}$.
10: **else**
11: Compute A^{-1} using forward substitution.
12: **end if**
13: **end function**

execute concurrently with the statements that follow it, without mandating the scheduler to run them in parallel. The `cilk_sync` statement acts as a synchronization barrier, ensuring that the program cannot continue until all previously spawned threads have completed. The runtime environment uses a work-stealing strategy to distribute the tasks among processors [BJK+95b].

To parallelize the implementation of the static Karatsuba method using Cilk, we only need to change Lines 33-35 in Listing 2.5, as shown in Listing 3.15.

```
1  cilk_spawn DnCMultiply(arraySize/2, fLowStar, gLowStar,
       lowArray);
2  cilk_spawn DnCMultiply(arraySize/2, fUpStar, gUpStar, highArray
       );
3  DnCMultiply(arraySize/2, fLowPlusUp, gLowPlusUp, midArray);
4  cilk_sync;
```

Listing 3.15: Adding Cilk constructs to parallelize the recursive calls in the Karatsuba method.

3.14.2 Performance Results of Multithreaded Static Karatsuba Multiplication Method

In this section, we present the performance results of the multithreaded static Karatsuba multiplication using the developed thread pool in Subsection 3.11.7 and compare them with the results of the serial and Cilk parallel executions. These tests were run on a system with the specifications listed in Table 2.2. The test results are shown in Table 3.1

and Figure 3.10. It can be observed that the parallel execution time is about one-third of the serial execution time.

Product Precision	Serial Karatsuba	Parallel Karatsuba with Cilk	Parallel Karatsuba with Developed Thread Pool	Developed Thread Pool Speedup
2	7.24e-5	0.0001	0.0001	1.8166
4	8.08e-5	7.44e-5	0.0001	1.6717
8	9.28e-5	7.15e-5	0.0001	1.4927
16	9.13e-5	7.66e-5	0.0001	1.5029
32	0.0002	0.0011	0.0003	1.5298
64	0.0005	0.0009	0.0007	1.3520
128	0.0016	0.0016	0.0014	1.1552
256	0.0048	0.0049	0.0027	1.7779
512	0.0179	0.0087	0.0067	2.6720
1024	0.0468	0.0225	0.0182	2.5698
2^{11}	0.1374	0.0600	0.0513	2.6802
2^{12}	0.4159	0.1589	0.1537	2.7066
2^{13}	1.2632	0.4436	0.4223	2.9920
2^{14}	3.8125	1.2809	1.2045	3.1640
2^{15}	11.5335	3.7492	3.6231	3.1821
2^{16}	34.8514	11.5012	10.6497	3.2736
2^{17}	105.8640	36.4011	33.6434	3.1465
2^{18}	316.8210	125.5320	110.7680	2.8603

Table 3.1: Execution time (seconds) for the serial and parallel (using Cilk and the developed thread pool) Karatsuba multiplication methods with a divide-and-conquer threshold of 16.

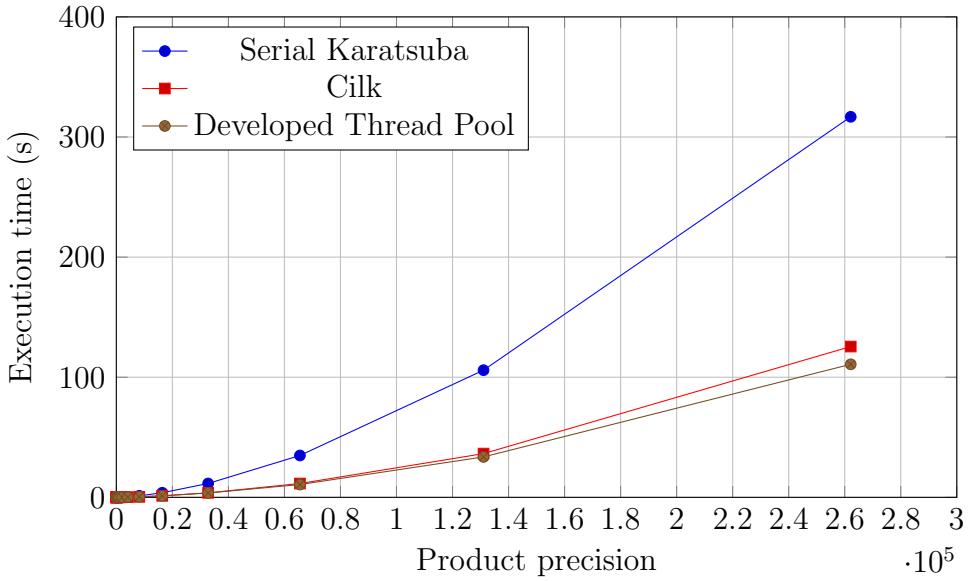

Figure 3.10: Execution time (seconds) for the serial and parallel (using Cilk and the developed thread pool) Karatsuba multiplication methods with a divide-and-conquer threshold of 16.

The cache performance statistics with no compiler optimization and with the compiler optimization flag `-O3` for one of the cases (product precision = 2^{12}) are shown in Table 3.3. Valgrind [NS07, Val24] is a programming framework used to build dynamic analysis tools for memory debugging, leak detection, and profiling. We use the Cachegrind tool in Valgrind to measure cache reads and misses.

Abbreviation	Meaning
I	Instruction cache
I1	Level 1 instruction cache
LLi	Last level instruction cache
D	Data cache
D1	Level 1 data cache
LLd	Last level data cache
LL	Last level cache (instruction and data)

Table 3.2: Cachegrind abbreviations and their meanings.

Cache Utilization Metrics	Without Optimization	With Optimization (`-O3`)
I refs	4,233,333,251	4,081,865,658
I1 misses	1,824,934	4,284
LLi misses	5,417	3,365
I1 miss rate	0.04%	0.00%
LLi miss rate	0.00%	0.00%
D refs	1,652,343,301 (1,035,349,659 rd + 616,993,642 wr)	1,568,869,497 (975,983,709 rd + 592,885,788 wr)
D1 misses	6,741,114 (5,523,511 rd + 1,217,603 wr)	6,661,397 (5,526,429 rd + 1,134,968 wr)
LLd misses	4,472,521 (3,276,345 rd + 1,196,176 wr)	4,406,618 (3,294,083 rd + 1,112,535 wr)
D1 miss rate	0.4% (0.5% + 0.2%)	0.4% (0.6% + 0.2%)
LLd miss rate	0.3% (0.3% + 0.2%)	0.3% (0.3% + 0.2%)
LL refs	8,566,048 (7,348,445 rd + 1,217,603 wr)	6,665,681 (5,530,713 rd + 1,134,968 wr)
LL misses	4,477,938 (3,281,762 rd + 1,196,176 wr)	4,409,983 (3,297,448 rd + 1,112,535 wr)
LL miss rate	0.1% (0.1% + 0.2%)	0.1% (0.1% + 0.2%)

Table 3.3: Memory access statistics for the case of product precision = 2^{12} in Table 3.1 using the developed thread pool, with and without compiler optimization (`-O3`).

3.14.3 Performance Results of Multithreaded Relaxed Multiplication Method

In this section, we present the performance results of the multithreaded relaxed multiplication using the developed thread pool in Subsection 3.11.7 and compare them with the results of the serial and Cilk parallel executions. These tests were run on a system with the specifications listed in Table 2.2. The test results are shown in Table 3.4 and Figure 3.11.

Product Precision	Serial Relaxed	Parallel Relaxed with Cilk	Parallel Relaxed with Developed Thread Pool	Developed Thread Pool Speedup
2	0.0001	0.0007	0.0008	0.125
4	0.0001	0.0001	0.0001	1.000
8	0.0001	0.0001	0.0001	1.000
16	0.0001	0.0001	0.0001	1.000
32	0.0002	0.0003	0.0006	0.333
64	0.0006	0.0007	0.0011	0.545
128	0.0017	0.0011	0.0013	1.308
256	0.0047	0.0027	0.0044	1.068
512	0.0131	0.0069	0.0054	2.426
1024	0.0319	0.0191	0.0155	2.058
2^{11}	0.0910	0.0493	0.0374	2.433
2^{12}	0.2736	0.1381	0.0985	2.777
2^{13}	0.8177	0.3981	0.2937	2.784
2^{14}	2.4597	1.1558	0.8202	2.999
2^{15}	7.4391	3.4227	2.3974	3.102
2^{16}	22.4544	10.4685	7.1766	3.129
2^{17}	68.3184	33.5267	21.9230	3.116
2^{18}	202.588	116.4900	74.9096	2.705

Table 3.4: Execution time (seconds) of the serial and parallel (using Cilk and the developed thread pool) relaxed multiplication methods.

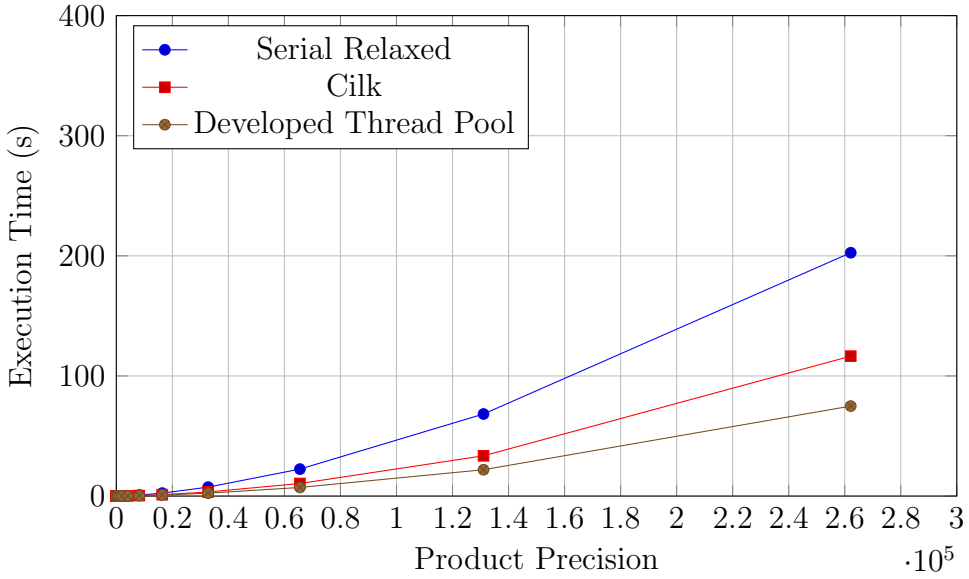

Figure 3.11: Execution time (seconds) of the serial and parallel (using Cilk and the developed thread pool) relaxed multiplication methods across varying product precision.

Chapter 4

Multivariate Power Series Multiplication

Section 4.1 is a review of the basic notions on multivariate power series. For more details, see [BKM20, ABK+20] and the references therein.

Since multiplying multivariate power series requires multiplying polynomials, we review in Section 4.2 some useful constructions which are used to reduce computational costs in polynomial multiplication.

In Section 4.3, we discuss various schemes for computed truncated products of multivariate power series. These schemes are based on the distributivity of multiplication over addition, and they are related in spirit to multi-way Karatsuba multiplication. We consider two truncations: w.r.t. total degree and w.r.t partial degree. In Chapter 5 we shall consider another scheme based on evaluation-interpolation.

Section 4.3.5 concludes this chapter with experimental results and observations.

4.1 Multivariate Power Series

Let $\mathbb{K}$ be a field. We assume that $\mathbb{K}$ is either of characteristic zero, or has a characteristic which is "large enough" to provide interpolation points, when needed, see Section 5.3. We denote by $\mathbb{K}[[X_1, \ldots, X_n]]$ the ring of formal power series with coefficients in $\mathbb{K}$ and with ordered variables $X_1 < \cdots < X_n$. For $f \in \mathbb{K}[[X_1, \ldots, X_n]]$, we write

$$f = \sum\nolimits_{e \in \mathbb{N}^n} a_e X^e,$$

where $a_e \in \mathbb{K}$, $X^e = X_1^{e_1} \cdots X_n^{e_n}$, $e = (e_1, \ldots, e_n) \in \mathbb{N}^n$, and $|e| = e_1 + \cdots + e_n$.

Let k be a non-negative integer. The *homogeneous part* and *polynomial part* of f in degree k are denoted $f_{(k)}$ and $f^{(k)}$, and are defined by

$$f_{(k)} = \sum\nolimits_{|e|=k} a_e X^e \quad \text{and} \quad f^{(k)} = \sum\nolimits_{i \leq k} f_{(i)}.$$

The *order* of f, denoted $\mathrm{ord}(f)$, is defined as $\min\{i \mid f_{(i)} \neq 0\}$, if $f \neq 0$, and as ∞ otherwise.

Recall several properties regarding power series. First, $\mathbb{K}[[X_1, \ldots, X_n]]$ is an integral domain. Second, the set $\mathcal{M} = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \text{ord}(f) \geq 1\}$ is the only maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$. Third, for all $k \in \mathbb{N}$, we have $\mathcal{M}^k = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \text{ord}(f) \geq k\}$. Note that for $n = 0$ we have $\mathcal{M} = \langle 0 \rangle$. Further, note that $f_{(k)} \in \mathcal{M}^k \setminus \mathcal{M}^{k+1}$ and $f_{(0)} \in \mathbb{K}$. Fourth, a unit $u \in \mathbb{K}[[X_1, \ldots, X_n]]$ has $\text{ord}(u) = 0$ or, equivalently, $u \notin \mathcal{M}$.

Let $f, g, h, p \in \mathbb{K}[[X_1, \ldots, X_n]]$. The *sum* and *difference* $h = f \pm g$ is given by $\sum_{k \in \mathbb{N}} (f_{(k)} \pm g_{(k)})$. The product $p = f\,g$ is given by $\sum_{k \in \mathbb{N}} \left(\Sigma_{i+j=k}\, f_{(i)} g_{(j)} \right)$. Notice that the these formulas naturally suggest a *lazy evaluation* scheme, where the result of an arithmetic operation can be incrementally computed for increasing *precision*. A power series f is said to be known to precision $k \in \mathbb{N}$, when $f_{(i)}$ is known for all $0 \leq i \leq k$.

4.2 Algorithms for Multiplying Multivariate Polynomials

We review in this section two basic schemes for multiplying dense multivariate polynomials. The first one, Multi-way Karatsuba, reduces the cost by taking advantage of the distributivity of multiplication over addition. It is suitable for relatively small input polynomials. The second one, multi-dimensional FFTs, offers additional advantages in terms of data locality and should be considered for those large problems not fitting in L3 cache.

4.2.1 Multi-way Karatsuba

Let R be a polynomial ring over the field $\mathbb{K}$ and n be a positive integer. Let $A_0, A_1, \ldots, A_n, B_0, B_1, \ldots, B_n$ be arbitrary elements of R, and let $X_1, \ldots, X_n$ be monomials of R. We define:

$$A := A_0 + A_1 X_1 + \cdots + A_n X_n, \quad \text{and } B := B_0 + B_1 X_1 + \cdots + B_n X_n. \tag{4.1}$$

We have:

$$AB = A_0 B_0 + \textstyle\sum_{i=1}^{i=n} A_i B_i X_i^2 + \sum_{1 \leq i < j \leq n} (A_i B_j + A_J B_i) X_i X_j. \tag{4.2}$$

This can be re-arranged to:

$$= A_0 B_0 + \textstyle\sum_{i=1}^{i=n} A_i B_i X_i^2 + \sum_{1 \leq i < j \leq n} \left((A_i + A_j)(B_i + B_j) - A_i B_i - A_j B_j \right) X_i X_j. \tag{4.3}$$

Equation (4.3) can be seen as a generalization of Karatsuba's trick for multiplying big integers [KO63] and univariate polynomials. Assume that the polynomials $A_0, A_1, \ldots, A_n$, $B_0, B_1, \ldots, B_n$ are dense of total degree d. Computing AB using Equation (4.2) leads to compute $(n+1)^2$ products of two polynomials taken from $A_0, A_1, \ldots, A_n, B_0, B_1, \ldots, B_n$. Using Equation (4.3) reduces this number of products to $\frac{1}{2}n^2 + \frac{1}{2}n + 1$ thus realizing of saving of $\frac{1}{2}n^2 - \frac{3}{2}n$, thus roughly (and only) dividing by 2 the number of products. Moreover, this reduction has a cost which is $\frac{3}{2}n^2 + \frac{3}{2}n + 3$ sums of two polynomials.

If $X_1, \ldots, X_n$ are algebraically independent over $\mathbb{K}$, then we beleive that one cannot reduce the number of products further. However, if $X_1, \ldots, X_n$ are powers of a given variable, we can realize more savings, by considering the family of algorithms known as Toom-Cook algorithms [Too63, Coo66, BZ07, Zan09]

4.2.2 Multiplicaiton Based on Multi-dimension FFTs

We recall a result of [MY22] on the multiplication of dense multivariate polynomial based on multi-dimensional FFT. For algorithm details, we refer to the paper [MY22].

We call *TFT time* any function $\ell \longmapsto \mathsf{F}(\ell)$

- giving an upper bound for the number of operations in $\mathbb{K}$ necessary for computing either the TFT or the inverse TFT of a vector of size ℓ and using at most $2^{\lceil \log_2(\ell) \rceil}$ elements of $\mathbb{K}$ for storage, and
- satisfying the following inequality for any finite sequence $z_1, \ldots, z_m \geq 2$ of positive integers:

$$\sum_{i=1}^{m} (z_1 \cdots z_{i-1} z_{i+1} \cdots z_m) \mathsf{F}(z_i) \leq \mathsf{F}(z_1 \cdots z_m). \tag{4.4}$$

Let $a, b \in \mathbb{K}[X_1, \ldots, X_n]$ be multivariate polynomials. Let i be an integer satisfying $1 \leq i \leq n$. We denote by d_i (resp. d'_i) the degree in X_i of a (resp. b). We assume the existence in $\mathbb{K}$ of a primitive s_i-th root of unity ω_i, where s_i is the smallest power of 2 satisfying $s_i > d_i + d'_i$. We write $s := s_1 \cdots s_n$. Then, Proposition 6 of [MY22] states that, in the fork-join model, the product ab can be computed in time

$$W(n, s) \leq 3\mathsf{F}(s) + s. \tag{4.5}$$

Assume that a and b are the polynomial parts $f^{(k)}$ and $g^{(k')}$, where k, k' are two positive integers. Then, for $1 \leq i \leq n$, we have

$$d_i = k \quad \text{and} \quad d'_i = k'. \tag{4.6}$$

We have:

$$s_i = s(k, k') \quad \text{where} \quad s(k, k') := 2^{\lceil \log_2(k+k'+1) \rceil} \tag{4.7}$$

Therefore, the product of $f^{(k)}$ and $g^{(k')}$ can be computed in time

$$T(n, k, k') \leq 3\mathsf{F}(s^n(k, k')) + s^n(k, k'). \tag{4.8}$$

4.3 Multiplication Schemes for Multivariate Power Series

Let $f, g \in \mathbb{K}[[X_1, \ldots, X_n]]$. Let k be a non-negative integer. Our goal is to compute $(fg)^{(2k)}$

1. using either a divide-and-conquer approach (based on the ideas of Karatsuba or Toom-Cook algorithms) or a modular method (based on an evaluation-interpolation method, if possible FFT-based), and

2. recycling as much as possible the computations that were performed to obtain $(fg)^{(k)}$.

In Section 4.3.1, we make a few observations on the polynomial part of the product of two power series. In Section 4.3.2, we discuss a first scheme computing $(fg)^{(2k)}$ from $f^{(2k)}$, $g^{(2k)}$ and $f^{(k)}g^{(k)}$. While this scheme seems theoretically attractive, its is of limited interest in practice, in particular when polynomial multiplication is done via multi-dimensional FFT. To overcome this difficulty in Section 4.3.4, we combine truncation in total degree and truncation in partial degrees. We discuss this latter truncation in Section 4.3.3.

4.3.1 The Polynomial Part of the Product of Two Power Series

Recall that $f^{(k)}$ and $g^{(k)}$ denote the polynomial parts of f and g in degree k. Thus, the polynomial $f^{(k)}$ (resp. $g^{(k)}$) consists of all the terms in f (resp. g) of (total) degree equal or less than k. We denote by $T(n,k)$ the maximum number of those terms and we have:

$$T(n,k) = \sum_{0 \leq i \leq k} \binom{n+i-1}{i} = \binom{n+k}{k}. \tag{4.9}$$

Considering the product fg of the formal power series f and g, the following clearly holds

$$f^{(k)}g^{(k)} \equiv (fg)^{(k)} \mod \mathcal{M}^{k+1}. \tag{4.10}$$

Indeed, we have:

$$\begin{array}{rcl} (fg)^{(k)} & = & \Sigma_{\ell=0}^{\ell=k} (fg)_{(\ell)} \\ & = & \Sigma_{\ell=0}^{\ell=k} \left(\Sigma_{i+j=\ell} f_{(i)} g_{(j)}\right) \\ & = & \Sigma_{i+j \leq k} f_{(i)} g_{(j)} \\ & \equiv & \Sigma_{i+j \leq 2k} f_{(i)} g_{(j)} \mod \mathcal{M}^{k+1} \\ & \equiv & f^{(k)} g^{(k)} \mod \mathcal{M}^{k+1} \end{array}$$

It follows from Equation (4.10) that $(fg)^{(k)}$ is the normal form of $f^{(k)}g^{(k)}$ w.r.t. any Gröbner basis of $\mathcal{M}^{k+1}$ for any graded term order.

In order to better exploit the theory of Gröbner bases in the sequel of this work, we introduce some notations.

Notation 1. *Let $I \subseteq \mathbb{K}[X_1, \ldots, X_n]$ be an ideal and let τ an admissible monomial ordering on $\mathbb{K}[[X_1, \ldots, X_n]]$. Let $G = \{G_1, \ldots, G_m\}$ be the reduced Gröbner basis for I w.r.t. τ. Let $A \in \mathbb{K}[X_1, \ldots, X_n]$ be a polynomial. We denote by $\mathsf{Rem}(A, G)$ the normal form of A w.r.t. G and by $\mathsf{Quo}(A, G, G_i)$ the quotient of A w.r.t. the generator G_i of G. Therefore, we have:*

$$A = \mathsf{Rem}(A,G) + \mathsf{Quo}(A,G,G_1)\, G_1 + \cdots + \mathsf{Quo}(A,G,G_m)\, G_m. \tag{4.11}$$

Following up on Notation 1, let $B \in \mathbb{K}[X_1, \ldots, X_n]$ be a second polynomial. Similarly to Equation (4.11), we have:

$$B = \mathsf{Rem}(B,G) + \mathsf{Quo}(B,G,G_1) G_1 + \cdots + \mathsf{Quo}(B,G,G_m) G_m. \tag{4.12}$$

Obviously, we have:

$$\mathsf{Rem}(AB, G) \equiv \mathsf{Rem}(A, G)\,\mathsf{Rem}(B, G) \mod \langle G \rangle. \tag{4.13}$$

For the purpose of increasing the precision of the polynomial part of the product of two power series, we are interested in $\mathsf{Rem}(AB, \langle G \rangle^2)$ where $\langle G \rangle = I$ is a monomial ideal, typically $\mathcal{M}^{k'}$ for some positive integer k'.

Since $G_i G_j \in I^2$ for all $1 \leq i \leq j \leq m$, we have:

$$\begin{array}{rcl} AB & \equiv & \mathsf{Rem}(A, G)\mathsf{Rem}(B, G) + \\ & & \mathsf{Rem}(A, G)\,(\Sigma_{i=1}^{m} \mathsf{Quo}(B, G, G_i) G_i) + \\ & & \mathsf{Rem}(B, G)\,(\Sigma_{i=1}^{m} \mathsf{Quo}(A, G, G_i) G_i) \mod I^2. \end{array} \tag{4.14}$$

4.3.2 Computing $(fg)^{(2k)}$ from $f^{(2k)}$, $g^{(2k)}$ and $f^{(k)} g^{(k)}$

With Equation (4.10), we have:

$$f^{(2k)} g^{(2k)} \equiv (fg)^{(2k)} \mod \mathcal{M}^{2k+1}. \tag{4.15}$$

Let H be the degree-reverse-lexicographic reduced Gröbner basis of $\mathcal{M}^{k+1}$. With Equation (4.11), we have:

$$\begin{array}{rcl} f^{(2k)} & = & \mathsf{Rem}(f^{(2k)}, H) + \Sigma_{h \in H}\, \mathsf{Quo}(f^{(2k)}, H, h)\, h \\ & = & f^{(k)} + \Sigma_{h \in H}\, \mathsf{Quo}(f^{(2k)}, H, h)\, h. \end{array} \tag{4.16}$$

Similarly, we have:

$$g^{(2k)} = g^{(k)} + \Sigma_{h \in H}\, \mathsf{Quo}(g^{(2k)}, H, h)\, h. \tag{4.17}$$

Since for all $h, h' \in H$, we have:

$$h\, h' \equiv 0 \mod \mathcal{M}^{2k+1} \tag{4.18}$$

we deduce the following.

Proposition 1. *We have:*

$$\begin{array}{rcl} (fg)^{(2k)} & \equiv & f^{(k)} g^{(k)} + \\ & & f^{(k)} \Sigma_{h \in H}\, \mathsf{Quo}(g^{(2k)}, H, h)\, h + \\ & & g^{(k)} \Sigma_{h \in H}\, \mathsf{Quo}(f^{(2k)}, H, h)\, h \mod \mathcal{M}^{2k+1}. \end{array} \tag{4.19}$$

Proposition 1 provides a way to save on computations w.r.t. directly computing $(fg)^{(2k)}$ as

$$\mathsf{Rem}((f)^{(2k)}(g)^{(2k)}, \mathcal{M}^{2k+1}).$$

One should also note that the decompositions given by Equation (4.16) and Equation (4.17) are made essentially at no cost since H is a monomial basis.

However, after computing

$$\Sigma_{h \in H}\, \mathsf{Quo}(g^{(2k)}, H, h)\, h + g^{(k)} \Sigma_{h \in H}\, \mathsf{Quo}(f^{(2k)}, H, h)\, h \tag{4.20}$$

one still needs to compute the remainder of that polynomial w.r.t. $\mathcal{M}^{2k+1}$. Therefore, Proposition 1 does not provide a fully satisfactory solution to the question of computing $(fg)^{(2k)}$ from $f^{(2k)}$, $g^{(2k)}$, and $f^{(k)}g^{(k)}$, since many computed terms will be discarded.

Moreover, in each of the products of Equation 4.20, one factor has total degree $2k$ and the other has degree k. If these two factors are dense, which will often be the case in the context of power series, and are multiplied a multi-dimensional FFT algorithm, then their product will be computed as a polynomial of degree

$$2^{\lceil \log_2(3k+1) \rceil} \tag{4.21}$$

If $k = 2^e$, the above formula evaluates to 2^{e+2}. Therefore, in this context, the fact that one factor has degree k brings the same benefits as if it had degree $2k-1$, due the padding done by FFT computations.

In order to make a better use of FFT, one should rather use it for multiplying dense polynomials of the same total degree k where k is of the form $2^e - 1$ so that their product has total degree $2^{e+1} - 2$, thus minimizing padding.

4.3.3 The Truncation in Partial Degrees of the Product of Two Power Series

We denote by $f^{[k]}$ (resp. $g^{[k]}$) and call the k-th *cubic truncation* or *truncation in partial degrees* of f (resp. g), that is, the sum of all the terms in f (resp. g) where the partial degree w.r.t. X_i does not exceed k, for all $i = 1 \cdots n$.

We are curious about the ratio between the number of terms in $f^{(k)}$ and the number of terms in $f^{[k]}$. We denote by $C(n,k)$ the maximum number of the terms in $f^{[k]}$. Clearly, we have:

$$C(n,k) = (k+1)^n. \tag{4.22}$$

For $n = 1$, we have $T(n,k) = k + 1 = C(n,k)$. For $n = 2$, we have $T(n,k) = \frac{(k+2)(k+1)}{2}$ and $C(n,k) = (k+1)^2$, thus

$$\frac{C(2,k)}{T(2,k)} = \frac{1}{2}\,\frac{k+1}{k+2}. \tag{4.23}$$

For $n = 3$, we have $T(n,k) = \frac{(k+3)(k+2)(k+1)}{6}$ and $C(n,k) = (k+1)^3$, thus

$$\frac{C(2,k)}{T(2,k)} = \frac{1}{6}\,\frac{(k+1)^2}{(k+3)(k+2)}. \tag{4.24}$$

It is easy to check that, asymptotically, the ratio $\frac{C(n,k)}{T(n,k)}$ is equivalent to $\frac{1}{(n+1)!}$.

Now, we are curious about the relation between $f^{[k]}, g^{[k]}$ and $(fg)^{[k]}$. We denote by $\mathcal{C}_k$ the polynomial ideal $\langle X_1^K, X_2^K, \ldots, X_n^K \rangle$. We note that its generators are a Gröbner basis (for any term order) of that ideal.

Similarly to Equation (4.10), we have the following identity:

$$f^{[k]}\ g^{[k]} \quad \equiv \quad (fg)^{[k]} \quad \mod \langle X_1^K, X_2^K, \ldots, X_n^K \rangle, \tag{4.25}$$

with $K = k + 1$. Indeed, any term of $(fg)^{[k]}$ is necessarily a term of $f^{[k]}\ g^{[k]}$. Conversely, any term of $f^{[k]}\ g^{[k]}$ with no partial degree exceeding k is also a term of $(fg)^{[k]}$. Therefore, $(fg)^{[k]}$ is the normal form of $f^{[k]}g^{[k]}$ w.r.t. any Gröbner basis of $\mathcal{M}^{k+1}$ for a graded term order.

4.3.4 Computing $(fg)^{(2k)}$ from $f^{[k]}g^{[k]}$

Using the same notations as in the previus sections, we observe that there exist polynomials $Q_1^f, \ldots, Q_n^f, Q_1^g, \ldots, Q_n^g$ such that we have:

$$f^{(2k)} \ = \ f^{[k]} + Q_1^f X_1^k + \cdots + Q_n^f X_n^k \ \text{ and } \ g^{(2k)} \ = \ g^{[k]} + Q_1^g X_1^k + \cdots + Q_n^g X_n^k, \quad (4.26)$$

so that none of the $Q_1^f, \ldots, Q_n^f, Q_1^g, \ldots, Q_n^g$ has a constant term, and each of them has a total degree of at most k.

It follows from Equation 4.10 that:

Proposition 2. *We have:*

$$(fg)^{(2k)} \ \equiv \ f^{[k]}g^{[k]} + (Q_1^f g + Q_1^g f)X_1^k + \cdots + (Q_n^f g + Q_n^g f)X_n^k. \mod \mathcal{M}^{2k+1}. \quad (4.27)$$

Proposition 2 shows that $(fg)^{(2k)} \mod \mathcal{M}^{2k+1}$ can be computed by computing $2n+1$ products (namely $f^{[k]}g^{[k]}$, $Q_1^f g + Q_1^g f$, …, $Q_n^f g + Q_n^g f$) in partial degrees all eqaul to k, n shifts (namely the multiplications by X_1^k, …, X_n^k) and 1 truncation (namely modulo $\mathcal{M}^{2k+1}$). Since the n shifts and the truncation only involve operations on the exponent vectors, we deduce that Proposition 2 shows that $(fg)^{(2k)} \mod \mathcal{M}^{2k+1}$ can be computed within

$$(2n + 1)(3\mathsf{F}(s^n) + s^n) \quad (4.28)$$

coefficient operations, where $s = 2^{\lceil \log_2(2k+1) \rceil}$.

A graphical illustration for the case of decomposing and multiplying power series $f(X_1, X_2)$ and $g(X_1, X_2)$ using 4.27 (hereafter referred to as the partition method) is shown in Figure 4.1, where we have:

$$\begin{aligned} (fg)^{(2k)} \ &= \ \left[f^{[k]} + C_k(f)X_1^k + D_k(f)X_2^k\right] \cdot \left[g^{[k]} + C_k(g)X_1^k + D_k(g)X_2^k\right] \mod \mathcal{M}^{2k+1} \\ &= \ f^{[k]}\left(g^{[k]} + C_k(g)X_1^k + D_k(g)X_2^k\right) + g^{[k]}\left(C_k(f)X_1^k + D_k(f)X_2^k\right) \end{aligned} \quad (4.29)$$

4.3.5 Experimental Results

In this section, we present the performance comparison between the direct and partition power series multiplication methods with varying numbers of variables and degrees in both serial and parallel implementations. Notably, the multiplications in the partition method are independent, making it suitable for parallelization. We parallelize them using the map pattern in [Bra22]. The tests were conducted on a system with the specifications listed in Table 4.1.

The execution times for both the serial and parallel implementations of the direct and partition methods and the partition method speedup for 2, 3, 4, and 5 variables with

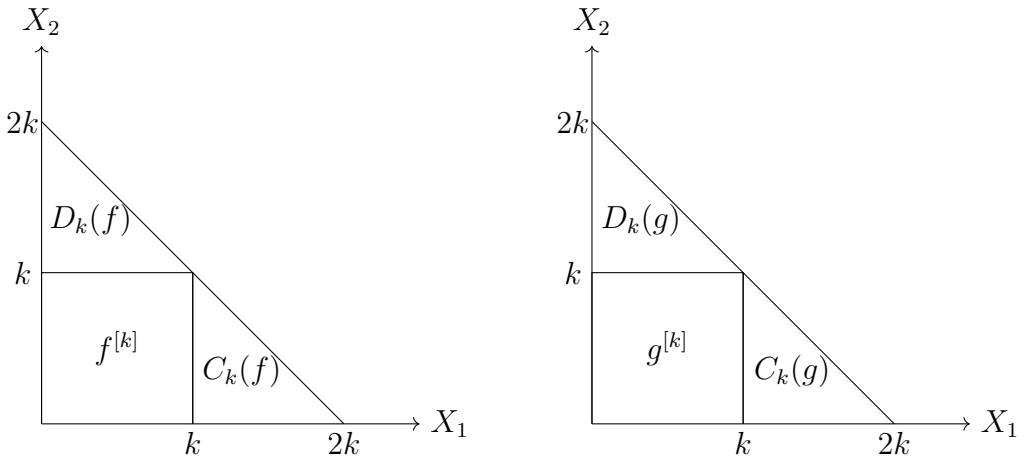

Figure 4.1: A graphical illustration of decomposing $f(X_1, X_2)$ and $g(X_1, X_2)$ in the partition method.

Component	**Specifications**
CPU	Intel Xeon X5650 @ 2.67GHz
Architecture	x86_64
CPU(s)	24
On-line CPU(s) list	0-23
Thread(s) per core	2
Core(s) per socket	6
Socket(s)	2
Stepping	2
Frequency boost	Enabled
Hyperthreading	Enabled
L1 Data Cache (L1d)	384 KiB (12 instances)
L1 Instruction Cache (L1i)	384 KiB (12 instances)
L2 Cache	3 MiB (12 instances)
L3 Cache	24 MiB (2 instances)

Table 4.1: System Specifications.

varying degrees are shown in Tables 4.2, 4.3, 4.4, and 4.5 and Figures 4.2, 4.3, 4.4, and 4.5. As the number of variables increases, the partition method increasingly outperforms the direct method.

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0001	0.0002	0.0023	0.0002	0.9016
4	0.0003	0.0005	0.0025	0.0005	0.9765
8	0.0011	0.0015	0.0047	0.0005	2.8048
16	0.0042	0.0052	0.0068	0.0015	3.4791
32	0.0128	0.0179	0.0139	0.0050	3.5881
64	0.0494	0.0579	0.0298	0.0142	4.0757
128	0.2116	0.2355	0.0815	0.0656	3.5903
256	0.8285	1.0217	0.3018	0.2308	4.4278
512	3.3755	3.9346	0.9162	0.9589	4.1033
1024	14.4911	16.8674	3.6976	4.0522	4.1629
2^{11}	61.8196	73.0748	15.6075	16.4796	4.4356
2^{12}	273.0004	309.6342	67.2966	71.7569	4.3162

Table 4.2: Performance comparison for 2 variables in seconds. The Parallel Partition Method is 5 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0008	0.0009	0.0040	0.0006	1.4761
4	0.0054	0.0050	0.0075	0.0018	2.8185
8	0.0366	0.0328	0.0271	0.0062	5.3004
16	0.2902	0.2292	0.0905	0.0417	5.4897
32	2.4415	1.9020	0.5981	0.3285	5.7904
64	18.5132	17.0732	3.6049	3.0133	5.6647
128	194.3451	129.6254	35.0982	21.7583	5.9590
160	355.3166	259.5845	61.8392	41.6352	6.2358

Table 4.3: Performance comparison for 3 variables in seconds. The Parallel Partition Method is 7 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0052	0.0036	0.0073	0.0015	2.4711
4	0.0541	0.0360	0.0284	0.0080	4.5106
8	0.9321	0.4569	0.2197	0.0961	4.7554
16	14.6326	7.3545	2.7641	1.5972	4.6050
32	265.8875	132.6068	37.8152	17.7673	7.4638
40	421.1498	204.8680	70.5869	27.0816	7.5664

Table 4.4: Performance comparison for 4 variables in seconds. The Parallel Partition Method is 9 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0465	0.0192	0.0325	0.0047	4.1256
4	1.1803	0.4159	0.2401	0.0871	4.7768
6	9.4207	2.4694	1.6607	0.3709	6.6585
8	45.8923	11.5437	6.9359	1.3823	8.3512
10	148.4728	43.2585	20.4864	5.2195	8.2857
12	336.4207	104.9664	43.8913	12.0742	8.6923

Table 4.5: Performance comparison for 5 variables in seconds. The Parallel Partition Method is 11 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

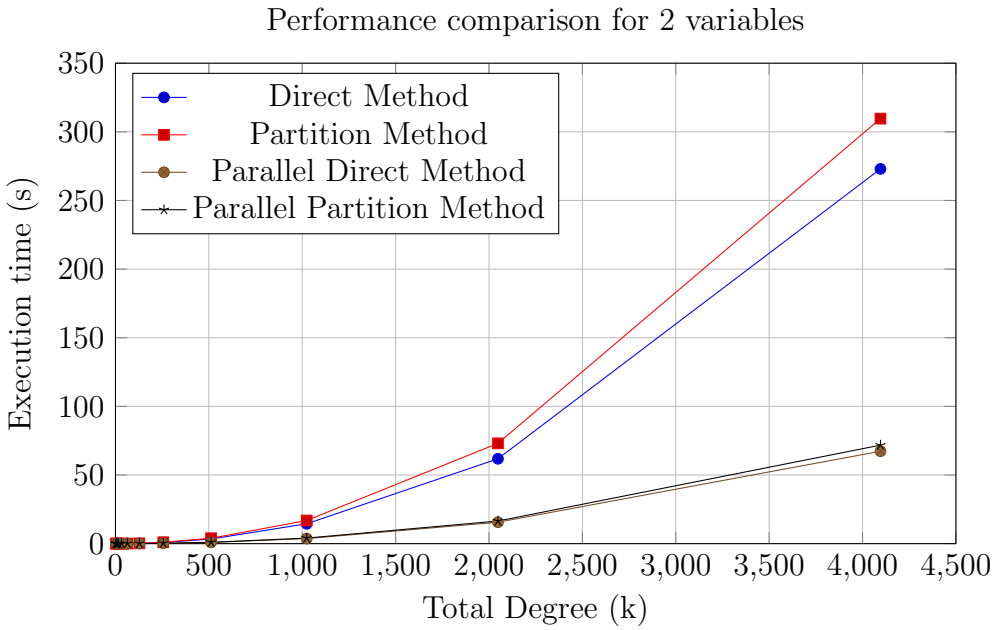

Figure 4.2: Execution time comparison for 2 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

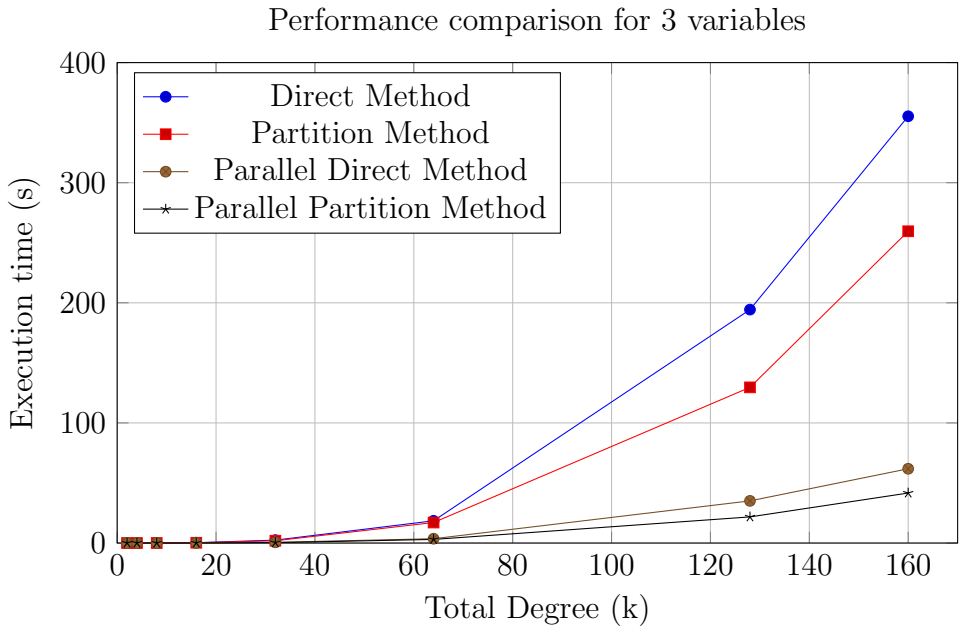

Figure 4.3: Execution time comparison for 3 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

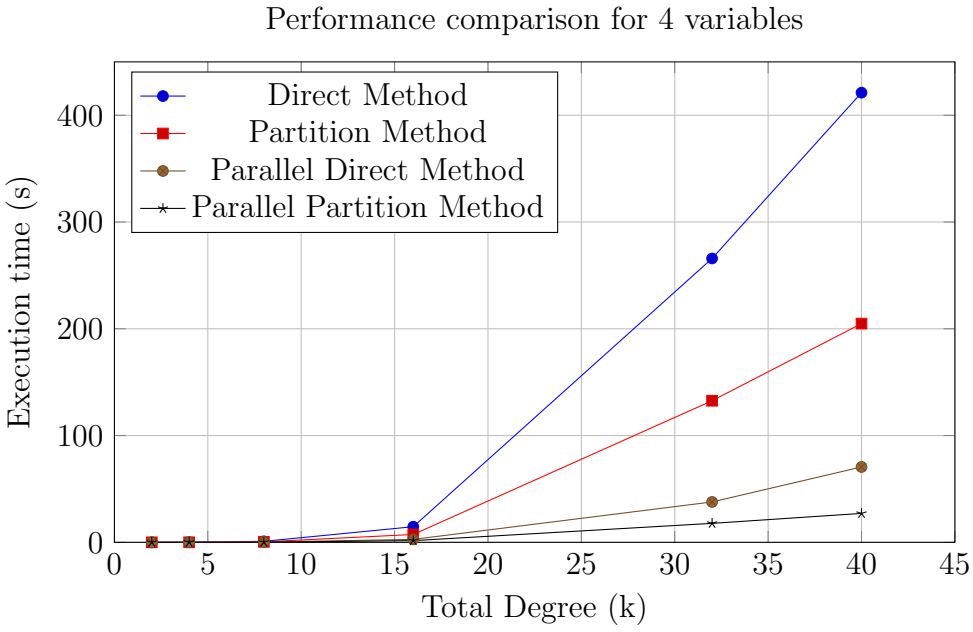

Figure 4.4: Execution time comparison for 4 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

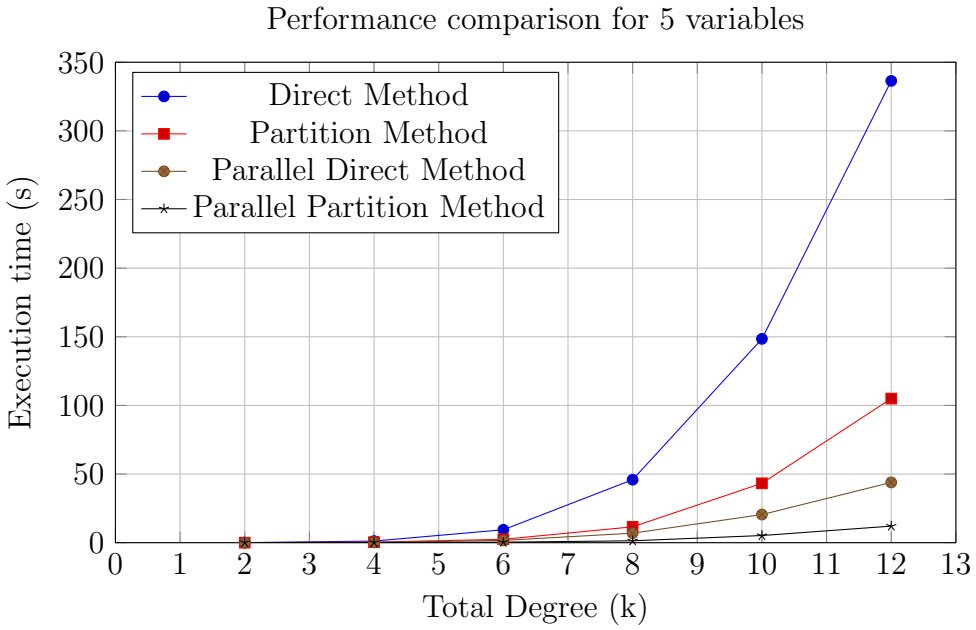

Figure 4.5: Execution time comparison for 5 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

Chapter 5

Modular Multiplication for Multivariate Power Series

In this chapter, we describe a method for computing the product of two multivariate power series modulo a monomial ideal. This is a more general problem than the one discussed in Chapter 4. We follow the deflation technique presented in [Sch05]. We have implemented the algorithms of that paper in their full generality, that is, for an arbitrary monomial ideal, whereas the author's implementation reported in [Sch05] deals only with univariate power series.

In Section 5.1, we set up the notations. In Section 5.2, we go through an introductory example. In Section 5.3, we explain how to choose the evaluation points. The evaluation and interpolation phases are detailed in Sections 5.4 and 5.5. Cost analysis is discussed 5.6 and some experimentation is reported in Section 5.7.

5.1 An Evaluation-Interpolation Strategy

We will be using standard results from the theory of Gröbner bases, for which we refer to the landmark textbooks [CLO97, CLO05].

Let $G = \{g_1, \ldots, g_m\}$ be the reduced minimal Gröbner basis of a monomial ideal $\mathcal{I}$ of $\mathbb{K}[X_1, \ldots, X_n]$, w.r.t. some admissible monomial order τ, which refines the partial order comparing total degrees of monomials. We assume that $\mathcal{I}$ is zero-dimensional, thus the residue class ring $\mathbb{K}[X_1, \ldots, X_n]/\mathcal{I}$ is a vector space Q over $\mathbb{K}$ of dimension d, where d is the degree of the ideal $\mathcal{I}$. Moreover, for every i such that $1 \leq i \leq n$ there exists j such that $1 \leq j \leq m$ and a positive integer $\delta_{i,j}$ so that we have:

$$g_j = X_i^{\delta_{i,j}}.$$

More generally, we write every polynomial $g_j \in G$, for $1 \leq j \leq m$ as:

$$g_j = X_1^{\delta_{1,j}} \cdots X_n^{\delta_{n,j}}. \tag{5.1}$$

We denote by T the set of the monomials that do not belong to $\mathcal{I}$; those monomials form a basis of the vector space Q. In practice, the ideal $\mathcal{I}$ would often be $\mathcal{M}^k$ or

$\langle X_1^k, X_2^k, \ldots, X_n^k \rangle$ for some positive integer k. Let $A, B \in \mathbb{K}[X_1, \ldots, X_n]$. Our goal is to compute

$$C := A \cdot B \mod \mathcal{I} \tag{5.2}$$

For that goal, we can view A and B as elements of Q. There exists a positive integer r (called the regularity of the $\mathbb{K}$-algebra Q) so that we can decompose A as:

$$A = A_0 + A_1 + \cdots + A_r, \tag{5.3}$$

where A_i is the homogeneous component of A of degree i, for $0 \leq i \leq r$. Similarly, we write B as:

$$B = B_0 + B_1 + \cdots + B_r, \tag{5.4}$$

where B_i is the homogeneous component of B of degree i, for $0 \leq i \leq r$.

5.2 Introductory Example

We first consider an example with $n = 1$ and $\mathcal{I} = \langle X_1^2 \rangle$. Thus we can write:

$$A = a_1 X_1 + a_0 \quad \text{and} \quad B = b_1 X_1 + b_0, \tag{5.5}$$

for some a_1, a_0, b_1, b_0 in $\mathbb{K}$. We have $T = \{1, X_1\}$.

One would like to use an evaluation-interpolation scheme. If $\mathcal{I}$ wold be $\langle X_1(X_1 - 1) \rangle$, then we could do it by means of the Chinese Remaindering Theorem. To reduce to this case, we replace $\mathcal{I}$ with $\mathcal{I}_Z := \langle X_1(X_1 - Z) \rangle$, where Z is a new variable, which is meant to be specialized to zero in order to retrieve the ideal $\mathcal{I}$.

Now we evaluate A and B at $X_1 = 0$ and $X_1 = Z$. We build the corresponding evaluation matrix, by evaluating each of the basis elements of $T = \{1, X_1\}$ at each of the points of $\{0, Z\}$, yielding to:

$$M_{\text{eval}} = \begin{pmatrix} 1 & 0 \\ 1 & Z \end{pmatrix} \tag{5.6}$$

Next, we compute the coordinates of A and B in the basis given by T:

$$A = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \tag{5.7}$$

Hence the values of A and B at the points of $P_Z = \{0, Z\}$ are given by:

$$A_{\text{eval}} = \begin{bmatrix} a_0 \\ Z a_1 + a_0 \end{bmatrix} \quad \text{and} \quad B_{\text{eval}} = \begin{bmatrix} b_0 \\ Z b_1 + b_0 \end{bmatrix} \tag{5.8}$$

In order to obtain $A \cdot B \mod \mathcal{I}_Z$, we compute the product of the above vectors, which produces:

$$(AB)_{\text{eval}} = \begin{bmatrix} a_0 b_0 \\ (Z a_1 + a_0)(Z b_1 + b_0) \end{bmatrix} \tag{5.9}$$

Next, we prepare for interpolating AB by computing the interpolation matrix, that is, the inverse of the evaluation matrix:

$$M_{\text{interp}} \;=\; \begin{bmatrix} 1 & 0 \\ -\frac{1}{\_Z} & \frac{1}{\_Z} \end{bmatrix} \tag{5.10}$$

Computing the matrix-vector product $M_{\text{interp}}(AB)_{\text{eval}}$ produces:

$$AB := a_0\, b_0 + \left(-\frac{a_0\, b_0}{\_Z} + \frac{(\_Z a_1 + a_0)\,(\_Z b_1 + b_0)}{\_Z} \right) x1. \tag{5.11}$$

Finally, we evaluate this latter at $Z = 0$ which yields:

$$a_0\, b_1\, x1 + a_1\, b_0 x1 + a_0\, b_0, \tag{5.12}$$

that is, $A \cdot B \mod \mathcal{I}$.

5.3 Choosing the Evaluation Points

The previous example can be generalized to an algorithmic procedure but one needs to specify how to choose the set of evaluation points. Let us continue with the univariate case, considering now the ideal $\mathcal{I} = \langle X_1^d \rangle$. Assume that $\mathbb{K}$ admits a d-th primitive root of unity ω. Then, we choose:

$$\mathcal{I}_Z \;= \langle X_1^d - Z^d \rangle \tag{5.13}$$

and we have

$$\mathbb{K}[X_1]/\mathcal{I}_Z \;\simeq\; \Pi_{i=0}^{d-1}\, \mathbb{K}[X_1]/\langle X_1 - \omega^i Z \rangle. \tag{5.14}$$

Now consider the multivariate case. Recall that, since the ideal $\mathcal{I}$ is zero-dimensional, for each $1 \leq i \leq n$ there exists a positive integer δ_i so that $X_i^{\delta_i}$ belongs to the Gröbner basis G. In fact δ_i is $\delta_{i,j}$ (defined earlier) if $X_i^{\delta_i}$ is the polynomial g_j.

Now for each $1 \leq i \leq n$, for each $0 \leq j < \delta_i$ choose $a_{i,j} \in \mathbb{K}$ so that the collection $a_{i,0}, a_{i,1}, \ldots, a_{i,\delta_i}$ consists of pairwise different values of $\mathbb{K}$. If there exists a δ_i-th primitive root of unity ω_i, then one can choose $a_{i,j} = \omega_i^j$. Next, for $1 \leq i \leq n$, we define the set:

$$\Omega_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,\delta_i - 1}\}. \tag{5.15}$$

Finally we define the sets

$$V_T \;=\; \{(a_{1,e_1}, a_{2,e_2}, \ldots, a_{n,e_n}) \;\mid\; X_1^{e_1} X_2^{e_2} \cdots X_n^{e_n} \in T\}, \tag{5.16}$$

and

$$P_Z \;=\; \{(a_{1,e_1} Z, a_{2,e_2} Z, \ldots, a_{n,e_n} Z) \;\mid\; X_1^{e_1} X_2^{e_2} \cdots X_n^{e_n} \in T\}, \tag{5.17}$$

where Z is our new variable. We note that the points of V_T (resp. P_Z) are pairwise different and the number of those points is equal to the cardinality of T. We note that every point p of P_Z writes $v \cdot Z$ where v is a point of V_T.

5.4 The Evaluation Phase

In order to evaluate the polynomials A and B at every point p of P_Z, we use their homogeneous decompositions given by Equations (5.3) and (5.4). For every point p of P_Z, we clearly have:

$$A(p) = A_0(v) + A_1(v)Z + \cdots + A_r(v)Z^r, \tag{5.18}$$

and,

$$B(p) \;=\; B_0(v) + B_1(v)Z + \cdots + B_r(v)Z^r, \tag{5.19}$$

where v is such that $p = v \cdot Z$. It follows that evaluating A and B at every point of P_Z reduces to evaluating A and B at every point of V_T.

Note that the above evaluation at every point of P_Z defines a linear map $E_{Q,W}$ from Q to the vector space W of dimension $r+1$ over the field $\mathbb{K}$ consisting of the univariate polynomials over $\mathbb{K}$ of degree at most r.

Recall that Q is a vector space over $\mathbb{K}$ with T as a basis. We denote by Q_i the sub-vector space of Q generated by T_i, where T_i consists of all terms of T with total degree i. We denote by d_i the dimension of Q_i over $\mathbb{K}$, that is, the number of elements in T_i. Hence, Q, and the direct sum $Q_0 \oplus \cdots \oplus Q_r$ of the vector spaces $Q_0, \ldots, Q_r$, are isomorphic. For $0 \leq j \leq r$, we order the elements of Ti using the lexicographical order induced by $X_1 < \cdots < X_n$. Then, we concatenate these ordered sets (in the order $Q_0, \ldots, Q_r$) so as to order the elements of T.

Correspondingly, we decompose the set V_T as:

$$V_T \;=\; V_{T_0} \cup \cdots \cup V_{T_r}, \tag{5.20}$$

where:

$$V_{T_i} \;=\; \{(a_{1,e_1}, a_{2,e_2}, \ldots, a_{n,e_n}) \;\mid\; X_1^{e_1} X_2^{e_2} \cdots X_n^{e_n} \in T_i\}, \tag{5.21}$$

and we order the elements of V_T using the order induced by T.

Similarly, we decompose the set P_Z as:

$$P_Z \;=\; P_Z^{(0)} \cup \cdots \cup P_Z^{(r)}, \tag{5.22}$$

where:

$$P_Z^{(i)} \;=\; \{(a_{1,e_1}Z, a_{2,e_2}Z, \ldots, a_{n,e_n}Z) \;\mid\; X_1^{e_1} X_2^{e_2} \cdots X_n^{e_n} \in T_i\}, \tag{5.23}$$

and we order the elements of P_Z using the order induced by T.

Next, we form two square matrices of order d that we denote by $M_{\text{eval}}(T, V_T)$ and $M_{\text{eval}}(T, P_Z)$. For $1 \leq i \leq d$ and $1 \leq j \leq d$, the element $c_{i,j}(T, V_T)$ (resp. $c_{i,j}(T, P_Z)$) of $M_{\text{eval}}(T, V_T)$ (resp. $M_{\text{eval}}(T, P_Z)$) at the intersection of the i-th row and j-th column is the value of the j-th element of T on the j-th element of V_T (resp. P_Z). By construction, we have:

$$c_{i,j}(T, P_Z) \;=\; c_{i,j}(T, V_T)\, Z^e, \tag{5.24}$$

where e is the positive integer defined by:

$$d_0 + \cdots + d_{e-1} < j \leq d_0 + \cdots + d_e. \tag{5.25}$$

Consider the diagonal matrix D_i of order d_i where every diagonal entry is equal to Z^i. Then, we define the matrix D as the diagonal matrix whose diagonal elements consist of the diagonal elements of D_1, followed by those of D_2, ..., followed by those of D_r. With Equation (5.24) we obtain the following result.

Theorem 1. *We have:*

$$M_{\text{eval}}(T, P_Z) \ = M_{\text{eval}}(T, V_T) \cdot D \tag{5.26}$$

In practice, evaluating A and B at every point $p = v \cdot Z$ of P_Z can be done simply by

1. evaluating their homogeneous components $A_0, A_1, \ldots, A_r$ and $B_0, B_1, \ldots, B_r$ at every point v of V_T,
2. forming the vector C of $(\mathbb{K}(Z))^d$ given, for every $p = v \cdot Z$ of P_Z, by:

$$C(p) \ = \ A(p)B(p) \tag{5.27}$$

We stress the fact that the product given by Equation (5.27) should not be expanded. The reason will be clear in the next section.

5.5 The Interpolation Phase

It follows from Theorem 1 that the matrix of the interpolation map (with the choices of bases used in Section 5.4) is given by:

$$M_{\text{interp}}(T, P_Z) \ = \ D^{-1} \cdot M_{\text{eval}}(T, V_T)^{-1}. \tag{5.28}$$

We observe that D^{-1} is the diagonal matrix whose diagonal elements consist of the inverses of the diagonal elements of D_1, followed by those of D_2, ..., followed by those of D_r.

Let i be a positive integer such that $i \leq d$. We note that the the i-th diagonal element of the matrix D^{-1} is Z^{-e}, where e is the positive integer defined by:

$$d_0 + \cdots + d_{e-1} < i \leq d_0 + \cdots + d_e. \tag{5.29}$$

It follows that for computing the i-th coordinate of C, where C was defined by Equation (5.2), we proceed as follows:

1. we determine e with Equation (5.29),
2. we compute and return the coefficient of Z^e in $A(p)B(p)$, where p is the i-th point of P_Z.

Indeed, the term of degree k in the i-th coefficient of $A(p)B(p)$, for all $k < e$ (resp. $e < k$) is necessarily zero (resp. specializes to zero at $Z = 0$).

5.6 Cost analysis

As explained in [Sch05], the proposed deflation technique runs in $\Omega(rd)$. In practice, the ideal $\mathcal{I}$ is often $\mathcal{M}^{k+1}$ or $\langle X_1^{k+1}, X_2^{k+1}, \ldots, X_n^{k+1} \rangle$ for some non-negative integer k. In

the former case, the regularity r and the degree d are $k+1$ and $\binom{n+k}{k}$, while they are $nk+n-1$ and $(k+1)^n$ in the latter case.

For the case of the ideal $\mathcal{M}^{k+1}$, the deflation technique of [Sch05] is not optimal, essentially due to the factor r in the lower bound $\Omega(rd)$. In fact, for that ideal and for the case of characterisitic zero, an algorithm running in $O\ (d)$, that is, in time proportional to d up to log factors, is presented in [LS03].

5.7 Experimentation

In this section we present the experimentation results from executing our evaluation-interpolation implementation in the BPAS library. The running times for multiplication of power series with different number of variables (n) and partial degrees (d_i) are shown in Table 5.1. Since the matrix inversion step is independent of polynomial coefficients and is the same for any polynomial with the same number of variables and degrees, its timing is not taken into account.

$n=2$		$n=3$	
d_i	Time (s)	d_i	Time (s)
2	0.0001	2	0.0003
4	0.0010	3	0.0017
6	0.0077	4	0.0091
8	0.0213	5	0.0380
16	0.3622	6	0.1121

Table 5.1: Execution time (s) of the evaluation-interpolation scheme for power series of different number of variables and partial degrees.

Bibliography

[10.84] *Lfp '84: Proceedings of the 1984 acm symposium on lisp and functional programming*, New York, NY, USA, Association for Computing Machinery, 1984.

[ABK+20] Mohammadali Asadi, Alexander Brandt, Mahsa Kazemi, Marc Moreno Maza, and Erik J. Postma, *Multivariate power series in maple*, Maple in Mathematics Education and Research - 4th Maple Conference, MC 2020, Waterloo, Ontario, Canada, November 2-6, 2020, Revised Selected Papers (Robert M. Corless, Jürgen Gerhard, and Ilias S. Kotsireas, eds.), Communications in Computer and Information Science, vol. 1414, Springer, 2020, pp. 48–66.

[ABK+21] Mohammadali Asadi, Alexander Brandt, Mahsa Kazemi, Marc Moreno Maza, and Eric Postma, *Multivariate power series in Maple*, Proc. of MC 2020, 2021.

[BJK+95a] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, *Cilk: an efficient multithreaded runtime system*, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA), PPOPP '95, Association for Computing Machinery, 1995, p. 207–216.

[BJK+95b] ———, *Cilk: an efficient multithreaded runtime system*, SIGPLAN Not. **30** (1995), no. 8, 207–216.

[BK78] Richard P Brent and Hsiang T Kung, *Fast algorithms for manipulating formal power series*, Journal of the ACM (JACM) **25** (1978), no. 4, 581–595.

[BKM20] Alexander Brandt, Mahsa Kazemi, and Marc Moreno Maza, *Power series arithmetic with the BPAS library*, Proc. of CASC 2020, LNCS, vol. 12291, Springer, 2020, pp. 108–128.

[BL99] Robert D. Blumofe and Charles E. Leiserson, *Scheduling multithreaded computations by work stealing*, J. ACM **46** (1999), no. 5, 720–748.

[BM21] Alexander Brandt and Marc Moreno Maza, *On the complexity and parallel implementation of hensel's lemma and weierstrass preparation*, Computer Algebra in Scientific Computing - 23rd International Workshop, CASC 2021, Sochi, Russia, September 13-17, 2021, Proceedings (François Boulier,

Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, eds.), Lecture Notes in Computer Science, vol. 12865, Springer, 2021, pp. 78–99.

[Bor07] Shekhar Borkar, *Thousand core chips: A technology perspective*, Proceedings of the 44th Annual Design Automation Conference (DAC), 2007, pp. 746–749.

[Bra22] Alexander Brandt, *The Design and Implementation of a High-Performance Polynomial System Solver*, Phd dissertation, The University of Western Ontario, 2022, Electronic Thesis and Dissertation Repository. 8733.

[Bre74] Richard P. Brent, *The parallel evaluation of general arithmetic expressions*, J. ACM **21** (1974), no. 2, 201–206.

[BS81] F. Warren Burton and M. Ronan Sleep, *Executing functional programs on a virtual tree of processors*, Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (New York, NY, USA), FPCA '81, Association for Computing Machinery, 1981, p. 187–194.

[BZ07] M. Bodrato and A. Zanoni, *Integer and polynomial multiplication: towards optimal Toom-Cook matrices*, ISSAC, 2007, pp. 17–24.

[CLO97] David A. Cox, John Little, and Donal O'Shea, *Ideals, varieties, and algorithms - an introduction to computational algebraic geometry and commutative algebra (2. ed.)*, Undergraduate texts in mathematics, Springer, 1997.

[CLO05] David A Cox, John Little, and Donal O'shea, *Using algebraic geometry*, vol. 185, Springer Science & Business Media, 2005.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, 4th ed., MIT Press, Cambridge, MA, 2022.

[CM96] GIOVANNI CESARI and ROMAN MAEDER, *Performance analysis of the parallel karatsuba multiplication algorithm for distributed memory architectures*, Journal of Symbolic Computation **21** (1996), no. 4, 467–473.

[CMPS10] Muhammad F. I. Chowdhury, Marc Moreno Maza, Wei Pan, and Éric Schost, *Complexity and performance results for non fft-based univariate polynomial multiplication*, ACM Commun. Comput. Algebra **44** (2010), no. 3/4, 99–100.

[Com11] C++ Standards Committee, *C++11 standard*, 2011, `https://isocpp.org/std/the-standard`.

[Com14] ———, *C++14 standard*, 2014, `https://isocpp.org/std/the-standard`.

[Com17] ———, *C++17 standard*, 2017, `https://isocpp.org/std/the-standard`.

[Com20] ———, *C++20 standard*, 2020, `https://isocpp.org/std/the-standard`.

[Con63] Melvin E. Conway, *A multiprocessor system design*, Proceedings of the November 12-14, 1963, Fall Joint Computer Conference (New York, NY, USA), AFIPS '63 (Fall), Association for Computing Machinery, 1963, p. 139–146.

[Coo66] S. A. Cook, *On the minimum computation time of functions*, Ph.D. thesis, 1966, URL: `http://cr.yp.to/bib/entries.html#1966/cook`.

[DM98] Leonardo Dagum and Ramesh Menon, *Openmp: An industry-standard api for shared-memory programming*, IEEE Comput. Sci. Eng. **5** (1998), no. 1, 46–55.

[GS01] Rajat P. Garg and I.A. Sharapov, *Techniques for optimizing applications: High performance computing*, 2001.

[GtGdt21] Torbjörn Granlund and the GMP development team, *Gnu mp: The gnu multiple precision arithmetic library*, GNU Project, 2021, `https://gmplib.org/`.

[Hol00] Allen Holub, *Taming java threads*, Apress, 2000.

[HP11] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, Elsevier, 2011.

[ISO20] *ISO/IEC 14882:2020: Information technology – programming languages – C++*, 6th ed., International Organization for Standardization, Geneva, 2020.

[KLJ14] Hsien-Kai Kuo, Bo-Cheng Charles Lai, and Jing-Yang Jou, *Reducing contention in shared last-level cache for throughput processors*, ACM Trans. Des. Autom. Electron. Syst. **20** (2014), no. 1.

[Knu97] Donald E. Knuth, *The art of computer programming*, 3 ed., vol. 1-4, Addison-Wesley, Boston, MA, 1997.

[KO63] A. A. Karatsuba and Y. Ofman, *Multiplication of multidigit numbers on automata*, Soviet Physics Doklady **7** (1963), 595–596, URL: `http://cr.yp.to/bib/entries.html#1963/karatsuba`.

[Lea00] Doug Lea, *A java fork/join framework*, Proceedings of the ACM 2000 Conference on Java Grande (New York, NY, USA), JAVA '00, Association for Computing Machinery, 2000, p. 36–43.

[LML00] Yibei Ling, Tracy Mullen, and Xiaola Lin, *Analysis of optimal thread pool size*, SIGOPS Oper. Syst. Rev. **34** (2000), no. 2, 42–55.

[LS03] Grégoire Lecerf and É Schost, *Fast multivariate power series multiplication in characteristic zero*, Electronic Journal of SADIO **5** (2003).

[LSB09] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt, *The design of a task parallel library*, vol. 44, 10 2009, pp. 227–242.

[LZMQ19] Yin Li, Yu Zhang, Xingpo Ma, and Chuanda Qi, *On the complexity of non-recursive n-term karatsuba multiplier for trinomials*, IACR Cryptol. ePrint Arch. (2019), 111.

[MRR12a] Michael McCool, Arch D. Robison, and James Reinders, *Chapter 3 - patterns*, Structured Parallel Programming (Michael McCool, Arch D. Robison, and James Reinders, eds.), Morgan Kaufmann, Boston, 2012, pp. 79–119.

[MRR12b] Michael D. McCool, Arch D. Robison, and James Reinders, *Structured parallel programming patterns for efficient computation*, 2012.

[MY22] Marc Moreno Maza and Haoze Yuan, *Balanced dense multivariate multiplication: The general case*, 24th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2022, Hagenberg / Linz, Austria, September 12-15, 2022, IEEE, 2022, pp. 35–42.

[NL16] Linus Nyman and Mikael Laakso, *Anecdotes: Notes on the history of fork and join*, IEEE Annals of the History of Computing **38** (2016), 84–87.

[NS07] Nicholas Nethercote and Julian Seward, *Valgrind: A framework for heavy-weight dynamic binary instrumentation*, Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), ACM, 2007, pp. 89–100.

[Roc09] Daniel S. Roche, *Space- and time-efficient polynomial multiplication*, Symbolic and Algebraic Computation, International Symposium, ISSAC 2009, Seoul, Republic of Korea, July 29-31, 2009, Proceedings (Jeremy R. Johnson, Hyungju Park, and Erich L. Kaltofen, eds.), ACM, 2009, pp. 295–302.

[Sch05] Éric Schost, *Multivariate power series multiplication*, Symbolic and Algebraic Computation, International Symposium ISSAC 2005, Beijing, China, July 24-27, 2005, Proceedings, ACM, 2005, pp. 293–300.

[SGG18] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating system concepts, 10th edition*, Wiley, 2018.

[Str13] Bjarne Stroustrup, *The c++ programming language*, 4th ed., Addison-Wesley Professional, 2013.

[Too63] A. L. Toom, *The complexity of a scheme of functional elements realizing the multiplication of integers*, Soviet Mathematics Doklady **3** (1963), 714–716.

[Val24] Valgrind Developers, *Valgrind: Instrumentation framework for building dynamic analysis tools*, 2024, Available at `http://www.valgrind.org/`.

[vdH02] Joris van der Hoeven, *Relax, but don't be too lazy*, J. Symb. Comput. **34** (2002), no. 6, 479–542.

[vdH14] _____, *Faster relaxed multiplication*, Proc. of ISSAC 2014, ACM, 2014, pp. 405–412.

[vdH19] _____, *Effective power series computations*, Found. Comput. Math. **19** (2019), no. 3, 623–651.

[vdHL13] Joris van der Hoeven and Gregoire Lecerf, *On the bit-complexity of sparse polynomial and series multiplication*, J. Symb. Comput. **50** (2013), 227–254.

[vzGG03] J. von zur Gathen and J. Gerhard, *Modern computer algebra*, 2 ed., Cambridge University Press, NY, USA, 2003.

[Wil19] Anthony Williams, *C++ concurrency in action*, second ed., Manning Publications, Shelter Island, NY, 2019.

[Zan09] Alberto Zanoni, *Toom-cook 8-way for long integers multiplication*, SYNASC (Stephen M. Watt, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, and Daniela Zaharie, eds.), IEEE Computer Society, 2009, pp. 54–57.

Appendices

.1 Fast Fourier Transform (FFT) in Polynomial Multiplication

The Fast Fourier Transform (FFT) is an efficient method for polynomial multiplication with a time complexity of $O(n \log n)$. The idea is that instead of directly multiplying two polynomials, as in the naive or Karatsuba methods with time complexities of $O(n^2)$ and $O(n^{\log_2 3})$, respectively, we can evaluate the two polynomials at a set of points, multiply these points pairwise, and then interpolate the product points to get the product polynomial. This approach uses the Polynomial Interpolation Theorem, which states that for any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n points where all x_k values are distinct, there is a unique polynomial $A(x)$ of degree at most $n-1$ such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$. Such a set of points is referred to as the value representation of the polynomial.

Although it is enough to represent polynomials A and B of degree $n-1$ by n unique points, when these points are multiplied, the resulting n points are not enough to represent the product polynomial $C = A \times B$ which has a degree of $2n-2$. Therefore, extended value representations for A and for B are needed which consist of at least $2n-1$ points. The FFT procedure represents the input polynomials by $2n$ points and assumes that n is a power of 2. If this is not the case, one can use zero-padding, which is adding higher-order zero coefficients.

The pairwise points multiplication has a time complexity of $O(n)$. One would expect that the evaluation and interpolation steps have a time complexity of $O(n^2)$; however, if the evaluation points are chosen to be complex roots of unity, the evaluation and interpolation time complexities will reduce to $O(n \log n)$, and therefore, the whole polynomial multiplication process using FFT will have a time complexity of $O(n \log n)$.

A number $\omega \in \mathbb{C}$ is an *nth root of unity* if $\omega^n = 1$. There are exactly n complex nth roots of unity: $\omega_n^k = e^{2\pi i k/n}$ for $k = 0, 1, \ldots, n-1$. Euler's formula, $e^{ix} = \cos(x) + i \sin(x)$, is used to represent the complex exponential function as trigonometric functions.

The FFT method uses a divide-and-conquer strategy to separate the even-indexed and odd-indexed coefficients of each of the two input polynomials into two new polynomials, each with $\frac{n}{2}$ terms. For a given coefficients sequence $a = (a_0, \ldots, a_{n-1}) \in \mathbb{C}^n$, we can write:

$$\begin{aligned} A_{even}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1}, \\ A_{odd}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}. \end{aligned} \tag{30}$$

And, the polynomial A can be evaluated at any $x \in \mathbb{C}$ using the formula:

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2). \tag{31}$$

Therefore, the problem of evaluating A at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to evaluating two polynomials of degree $\frac{n}{2} - 1$ at the points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$ and combining the results using Equation 31. According to the *Halving lemma*, if $n > 0$ is even, then the squares of the n complex n-th roots of unity are the $\frac{n}{2}$ complex $(\frac{n}{2})$-th roots of unity. Therefore, the list of values $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$ consists of only $\frac{n}{2}$ complex $(\frac{n}{2})$-th roots of unity.

The FFT algorithm recursively evaluates the polynomials A_{even} and A_{odd} of degree $\frac{n}{2} - 1$ at the $\frac{n}{2}$ complex $(\frac{n}{2})$-th roots of unity. With each recursion, the size of the problem is halved. The inverse FFT algorithm is used to convert the product of pointwise multiplications back to the coefficient form of the product polynomial.

The polynomial multiplication process using FFT is illustrated in Figure .1. Algorithms 7 and 8 show the FFT and inverse FFT in pseudo-code [CLRS22].

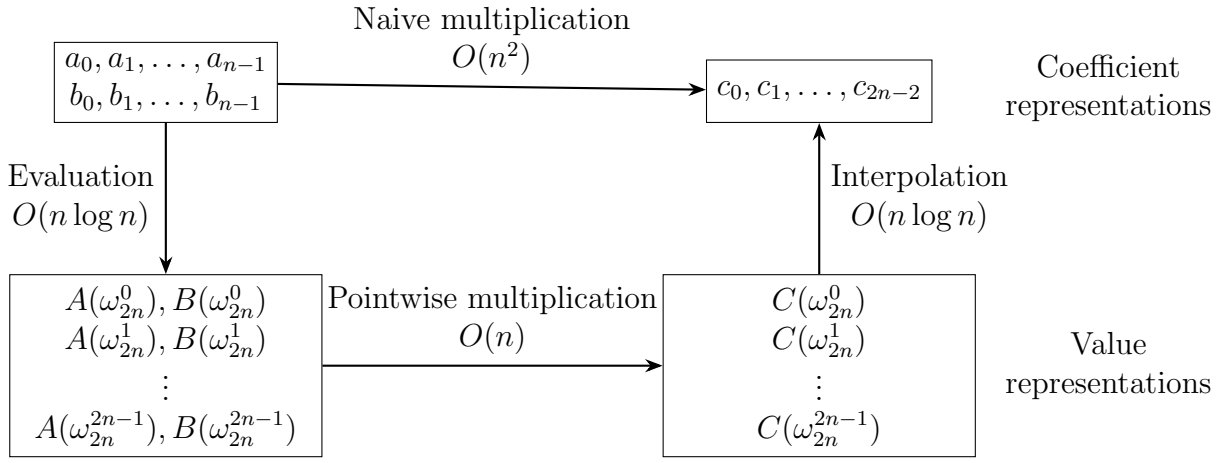

Figure .1: The process of polynomial multiplication using FFT. Evaluation and interpolation refer to converting a polynomial from coefficient representation to value representation and vice versa. The ω_{2n} terms are complex $2n$th roots of unity.

Algorithm 7 FFT

Input: Polynomial $A = [a_0, a_1, \ldots, a_{n-1}]$ of degree $n-1$, where n is a power of 2.
Output: The FFT of A.

function FFT(A, n)
  if $n = 1$ **then**
    return A
  end if
  $\omega := e^{2\pi i/n}$
  $A_{\text{even}} := [a_0, a_2, \ldots, a_{n-2}]$
  $A_{\text{odd}} := [a_1, a_3, \ldots, a_{n-1}]$
  $y_{\text{even}} := \text{FFT}(A_{\text{even}}, n/2)$
  $y_{\text{odd}} := \text{FFT}(A_{\text{odd}}, n/2)$
  $y := [0, 0, \ldots, 0]$ ▷ Initialize y with n zeros
  for $k = 0$ to $n/2 - 1$ **do**
    $y[k] := y_{\text{even}}[k] + \omega^k y_{\text{odd}}[k]$
    $y[k + n/2] := y_{\text{even}}[k] - \omega^k y_{\text{odd}}[k]$
  end for
  return y
end function

Algorithm 8 Inverse FFT

Input: $Y = [y_0, y_1, \ldots, y_{n-1}]$, where n is a power of 2.
Output: The inverse FFT of Y, i.e., the polynomial coefficients.

function IFFT(Y, n)
  if $n = 1$ **then**
    return Y
  end if
  $\omega := (1/n) \cdot e^{-2\pi i/n}$
  $Y_{\text{even}} := [y_0, y_2, \ldots, y_{n-2}]$
  $Y_{\text{odd}} := [y_1, y_3, \ldots, y_{n-1}]$
  $a_{\text{even}} := \text{IFFT}(Y_{\text{even}}, n/2)$
  $a_{\text{odd}} := \text{IFFT}(Y_{\text{odd}}, n/2)$
  $a := [0, 0, \ldots, 0]$ ▷ Initialize a with n zeros
  for $k = 0$ to $n/2 - 1$ **do**
    $a[k] := a_{\text{even}}[k] + \omega^k a_{\text{odd}}[k]$
    $a[k + n/2] := a_{\text{even}}[k] - \omega^k a_{\text{odd}}[k]$
  end for
  return a
end function

.2 Implementation of Inverting a Lower Triangular Matrix

```
void findMatrixInverse(mpq_t *AInv, mpq_t *A, int n, int B, ThreadPool
   &pool) {
  if (n > B) { // partition A into blocks of (n/2) * (n/2).
    mpq_t *A1, *A2, *A3;
    A1 = new mpq_t[(n/2)*(n/2)];
    A2 = new mpq_t[(n/2)*(n/2)];
    A3 = new mpq_t[(n/2)*(n/2)];
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        if (i < n/2 && j < n/2) {
          mpq_init(A1[i * n/2 + j]);
          mpq_set(A1[i * n/2 + j], A[i * n + j]);
        } else if (j < n/2) {
          mpq_init(A2[(i - n/2) * n/2 + j]);
          mpq_set(A2[(i - n/2) * n/2 + j], A[i * n + j]);
        } else  if (i >= n/2) {
          mpq_init(A3[(i - n/2) * n/2 + (j - n/2)]);
          mpq_set(A3[(i - n/2) * n/2 + (j - n/2)], A[i * n + j]);
        }
      }
    }
    // find inverse of A1 and A3.
    mpq_t *A1Inv, *A3Inv;
    A1Inv = new mpq_t[(n/2)*(n/2)];
    A3Inv = new mpq_t[(n/2)*(n/2)];

    std::future<void> fut1 = pool.submitTask([&] {
      findMatrixInverse(A1Inv, A1, n/2, B, pool);
    });
    std::future<void> fut2 = pool.submitTask([&] {
      findMatrixInverse(A3Inv, A3, n/2, B, pool);
    });
    while ((fut1.wait_for(std::chrono::seconds(0)) ==
      std::future_status::timeout) ||
      (fut2.wait_for(std::chrono::seconds(0)) ==
      std::future_status::timeout)) {
      pool.runPendingTask();
    }
    fut1.get();
    fut2.get();

    // ABLInv = -(A3^(-1))*A2*(A1^(-1)) = - A3Inv * A2 * A1Inv
    mpq_t *ABLInv = new mpq_t[(n/2)*(n/2)];
    mpq_t *tempMatrix = new mpq_t[(n/2)*(n/2)];

    multiplyMatrix(tempMatrix, A3Inv, A2, n/2);
    multiplyMatrix(ABLInv, tempMatrix, A1Inv, n/2);
    negateMatrix(ABLInv, n/2);

```

```
49      // assembling four quarters of A^(-1)
50      for (int i = 0; i < n; ++i) {
51        for (int j = 0; j < n; ++j) {
52          mpq_init(AInv[i * n + j]);
53          if (i < n/2 && j < n/2) {
54            mpq_set(AInv[i * n + j], A1Inv[i * n/2 + j]);
55          } else if (j < n/2) {
56            mpq_set(AInv[i * n + j], ABLInv[(i - n/2) * n/2 + j]);
57          } else if (i >= n/2) {
58            mpq_set(AInv[i * n + j], A3Inv[(i - n/2) * n/2 + (j - n/2)]);
59          } else {
60            mpq_set_ui(AInv[i * n + j], 0, 1);
61          }
62        }
63      }
64
65      // memory clean-up.
66      for (int i = 0; i < (n/2)*(n/2); ++i) {
67        mpq_clear(A1[i]);
68        mpq_clear(A2[i]);
69        mpq_clear(A3[i]);
70        mpq_clear(A1Inv[i]);
71        mpq_clear(A3Inv[i]);
72        mpq_clear(ABLInv[i]);
73        mpq_clear(tempMatrix[i]);
74      }
75      delete[] A1;
76      delete[] A2;
77      delete[] A3;
78      delete[] A1Inv;
79      delete[] A3Inv;
80      delete[] ABLInv;
81      delete[] tempMatrix;
82
83    } else { // when n <= B, no recursion, use forward substitution.
84      // create identity matrix
85      mpq_t tmp;
86      mpq_init(tmp);
87      mpq_t one;
88      mpq_init(one);
89      mpq_set_ui(one, 1, 1);
90      for (int i = 0; i < n; ++i) {
91        for (int j = 0; j < n; ++j) {
92          mpq_init(AInv[i * n + j]);
93          if (i == j) {
94            mpq_div(tmp, one, A[i * n + j]);
95            mpq_set(AInv[i * n + j], tmp);
96          } else {
97            mpq_set_ui(AInv[i * n + j], 0, 1);
98          }
99        }
100     }
101     mpq_clear(one);
102
```

```
103     mpq_t sum;
104     mpq_init(sum);
105     // forward substitution
106     for (int i = 1; i < n; ++i) {
107       for (int j = 0; j < i; ++j) {
108         mpq_set_ui(sum, 0, 1);
109         for (int k = j; k < i; ++k) {
110           mpq_mul(tmp, A[i * n + k], AInv[k * n + j]);
111           mpq_add(sum, sum, tmp);
112         }
113         mpq_div(tmp, sum, A[i*n + i]);
114         mpq_neg(tmp, tmp);
115         mpq_set(AInv[i * n + j], tmp);
116       }
117     }
118     mpq_clear(tmp);
119     mpq_clear(sum);
120   }
121 }
```

Listing 1: A multithreaded implementation of inverting a lower triangular matrix.

Curriculum Vitae

Name:	Hamid Fathi
Post-Secondary Education and Degrees:	Western University London, ON, Canada M.Sc. in Computer Science 2022 - 2024 Sharif University of Technology Tehran, Iran M.Sc. in Engineering 2018 - 2020
Related Work Experience:	Teaching Assistant Western University 2022 - 2024 Research Assistant Ontario Research Center for Computer Algebra Western University 2022 - 2024 R&D Intern Maplesoft, Waterloo, Canada Summer 2024 C++ Software Developer MTC, Tehran, Iran 2020-2022

Software:

- Basic Polynomial Algebra Subprograms (BPAS), `https://bpaslib.org/`.