8-26-2020 11:30 AM

A Generic Implementation of Fast Fourier Transforms for the BPAS Library

Colin S. Costello, *The University of Western Ontario*

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

In this thesis we seek to realize an efficient, generic and parallel implementation of fast Fourier transforms (FFTs) over finite fields. These FFTs will be used in support of fast multiplication of polynomials. Our goal is to obtain a relatively high performing parallel implementation that will run over a variety of finite fields, with characteristics ranging from small primes (say of a machine-word size) to arbitrarily large primes. To this end, we implement and compare two Cooley-Tukey Six-Step fast Fourier transforms and a Cooley-Tukey Four-Step variant, against a high performing specialized FFT already implemented in the Basic Polynomial Algebra Subprograms (BPAS) library. We use optimization techniques found in modern day high performance FFT implementations like FFTW by Matteo Frigo and Steven G. Johnson as well as SPIRAL by [16] . We start with a Six Step parallel algorithm suggested by Franz Franchetti and Markus Puschel in the Encyclopedia of Parallel Computing and derive two FFT variants, a Six-Step loop-merged variant and a Four-Step loop-merged variant. We implement and compare these FFTs in both C and C++ programming languages and we compare our BPAS finite field C++ implementation against a GNU multiple precision (GMP) implementation. We compare both serial and parallel versions over finite fields with characteristic primes of size 32 bits and larger. In addition to providing a fair comparison between FFT implementations with a varying degree of specialization, we optimistically hope to achieve a relative degree of performance with C++ as compared to C.and we hope to show how well the different FFTs parallelize and if that relates to the degree of specialization.

Keywords: FFT, Fast Fourier Transform, DFT, Discrete Fourier Transform, DFT over Finite Fields, Finite Fields, BPAS, C, C++, Gnu MP, Parallel

Summary for lay audience

The fast Fourier transform (FFT) is used by most scientific disciplines. High performance implementations of the fast Fourier transform play a crucial role in research areas like cryptography, signal processing, and polynomial system solving. On contemporary parallel computing platforms it is difficult to obtain high-performance FFT implementations. In this paper we derive and implement parallel Cooley-Tukey general radix decimation-in-time six step FFTs that we can match to various target platforms. Our goal is a competitive parallel FFT over Finite Fields. We detail and discuss our implementation and optimization effort. Then we analyze the performance of our implementation and present our findings.

Acknowlegements

I would like to acknowledge and kindly thank my supervisor Dr. Marc Moreno Maza.
I would also like to thank my colleagues at the Ontario Research Centre for Computer Algebra.
Linxiao Wang, thank you for all your help and for the use of your benchmark FFT.
Davood Mohajerani, thank you for all your help and guidance w.r.t. FFT implementations.
Svyatoslav Covanov, thank you for the use of your benchmark FFT implementation.
Alexander Brandt, thank you for all your time and for help in the lab.
Robert Moir, thank you for you support and guidance.

Contents

Abstract i

Summary for lay audience ii

Acknowlegements iii

List of Figures vii

1 Introduction 1
- 1.1 Background and objectives . . . 1
- 1.2 Methodology . . . 2
- 1.3 Results . . . 2
- 1.4 Document outline . . . 3

2 Finite Field Arithmetic 4
- 2.1 Symbol usage . . . 4
- 2.2 Modular arithmetic . . . 4
- 2.3 Primitive n^{th} root of unity . . . 5
- 2.4 Prime fields in the BPAS library . . . 5
- 2.5 General modular arithmetic algorithms . . . 6

3 Tensor product and the stride permutation 11
- 3.1 Tensor product . . . 11
- 3.2 Stride permutations . . . 14
  - 3.2.1 Matrix transposition as a permutation of an indexing set . . . 14
  - 3.2.2 Radix-2 stride permutations . . . 15
  - 3.2.3 Radix-P stride permutations . . . 16
- 3.3 Stride permutation subprogram . . . 17

4 Twiddle factors 20
- 4.1 Direct sum . . . 20
- 4.2 Diagonal scaling . . . 21
- 4.3 $D_{K,J}$ twiddle matrices . . . 21
- 4.4 The $D_{2,K}$ twiddle matrix . . . 21
- 4.5 $D_{K,J}$ twiddle matrix . . . 22
- 4.6 Twiddle factors subprogram . . . 23
- 4.7 Online twiddle function . . . 24

4.8 Offline twiddle function . . . 24
4.9 Twiddle pre-computation . . . 25
4.10 Twiddle function comparison . . . 25

5 The Fourier Transform **27**
5.1 The DFT matrix . . . 27
5.2 DFT_4 in terms of DFT_2 . . . 28
5.3 The Fast Fourier Transform . . . 29
5.3.1 Radix-2 FFTs . . . 29
5.3.2 Radix-2 twiddle matrix . . . 29
5.3.3 The Cooley-Tukey factorization . . . 30
5.4 Radix-P FFTs . . . 32
5.5 Cooley-Tukey Four-Step FFT . . . 33
5.6 Cooley-Tukey Six-Step FFT . . . 33

6 The Generic Fast Fourier Transform **34**
6.1 FFT base cases . . . 34
6.1.1 DFT_4 . . . 34
6.1.2 DFT_8 . . . 34
6.1.3 DFT_{16} . . . 35
6.1.4 DFT_{32} . . . 35
6.1.5 DFT_{64} . . . 36
6.1.6 Four-Step FFT base cases . . . 36
6.1.7 Four-Step DFT_4 . . . 36
6.1.8 Four-Step DFT_8 . . . 37
6.1.9 Four-Step DFT_{16} . . . 37
6.1.10 Four-Step DFT_{32} . . . 37
6.1.11 Four-Step DFT_{64} . . . 38
6.2 Loop unrolling a FFT base case . . . 38
6.3 General FFT function . . . 43
6.4 General FFT function optimization . . . 47
6.4.1 Pre-shuffling the input vector . . . 48
6.4.2 Loop merging . . . 49
6.4.3 Loop merging example . . . 49
6.4.4 Loop merged Six-Step FFT . . . 52
6.4.5 Four-Step FFT . . . 53
6.4.6 Dead code elimination . . . 54
6.4.7 Parallelization . . . 56

7 Experimentation **61**
7.1 Genericity and context switching . . . 61
7.2 Experimental setup . . . 64
7.3 Serial small prime field C: generic vs specialized benchmark . . . 64
7.4 Serial VS parallel C++ big prime experiment (generic) . . . 66
7.5 Serial VS parallel GMP . . . 72

7.6 C VS C++ small prime field experiment 76
7.7 C VS C++ big prime field experiment 80
7.8 Generalized Fermat prime field C serial benchmarks 84
7.9 Generalized Fermat prime field C parallel benchmarks 87
7.10 Memory Profile 90
7.11 Observations 92

Bibliography **93**

Curriculum Vitae **94**

List of Figures

3.1 Illustration of lemma 2.2.1 . . . 17
3.2 Blocked transpose (red) vs recursive transpose (blue) . . . 19

4.1 Offline twiddle blue vs online twiddle red . . . 26

6.1 DFT_{16} right-expanded using radix-2 splitting . . . 43
6.2 Recursive pre-shuffle . . . 48
6.3 Iterative pre-shuffle . . . 49
6.4 Six-Step FFT before loop merge code listing . . . 52
6.5 Six-Step FFT after loop merge code listing . . . 53
6.6 Four-Step FFT after loop merge code listing . . . 54
6.7 Twiddle function code listing . . . 55
6.8 Twiddle function parallel code listing . . . 55
6.9 Four-Step FFT code listing . . . 56
6.10 Parallel macro example . . . 57
6.11 Parallel grain size macro example . . . 57
6.12 Transpose parallel code listing . . . 57
6.13 Six-Step FFT parallel code listing . . . 58
6.14 Six-Step merged FFT parallel code listing . . . 59
6.15 Four-Step FFT parallel code listing . . . 60

7.1 C++ prime field macros . . . 62
7.2 Conditional inclusion code snippet . . . 62
7.3 C small prime field macros . . . 62
7.4 C big prime field macros . . . 63
7.5 C generalized Fermat prime field macros . . . 63
7.6 Serial small prime field C benchmark. . . . 65
7.7 Serial vs parallel C++, $K = 2$ $P2$ and $K = 4$ $P2$ . . . 67
7.8 Serial vs parallel C++, $K = 8$ $P2$ and $K = 16$ $P2$ . . . 69
7.9 Serial vs parallel C++, $K = 32$ $P2$ and $K = 64$ $P2$ . . . 71
7.10 Serial vs parallel GMP, $K = 2$ $P2$ and $K = 4$ $P2$ . . . 73
7.11 Serial vs parallel GMP, $K = 8$ $P2$ and $K = 16$ $P2$ . . . 74
7.12 Serial vs parallel GMP, $K = 32$ $P2$ and $K = 64$ $P2$ . . . 75
7.13 SPF C vs C++,$K = 2$ $P1$ and $K = 4$ $P1$ . . . 77
7.14 SPF C vs C++,$K = 8$ $P1$ and $K = 16$ $P1$ . . . 78
7.15 SPF C vs C++,$K = 32$ $P1$ and $K = 64$ $P1$ . . . 79
7.16 BPF C vs C++, $K = 2$ $P2$ and $K = 4$ $P2$ . . . 81

7.17 BPF C vs C++,$K = 8$ P2 and $K = 16$ $P2$ 82
7.18 BPF C vs C++,$K = 32$ $P2$ and $K = 64$ $P2$ 83
7.19 GFPF vs benchmark $K = 8$ $P3$ and $K = 16$ $P4$ 85
7.20 GFPF vs benchmark $K = 32$ $P5$ and $K = 64$ $P6$ 86
7.21 GFPF vs benchmark parallel $K = 8$ $P3$ and $K = 16$ $P4$ 88
7.22 GFPF vs benchmark parallel $K = 32$ $P5$ and $K = 64$ $P6$ 89

Chapter 1

Introduction

1.1 Background and objectives

In this thesis we seek to realize an efficient and generic implementation of fast Fourier transforms (FFTs) over finite fields. These FFTs aim at supporting fast multiplication of polynomials. Our goal is to obtain a relatively high performing implementation that will run over a variety of finite fields, with different sized characteristic primes, and offer a variety of FFT schemes, either running sequentially or in parallel fashion. This contrasts with the common practice in the literature to focus on a particular type of finite field and a specific FFT scheme. Of course, under those constraints, one expects that the resulting code would reach even higher performance. This is the case in the paper of Victor Shoup [20], or that of Akpodigha Filatei, Xin Li, Marc Moreno Maza, Éric Schost [10], or that of Joris van der Hoeven [21], or that of David Harvey, Daniel S. Roche [14], where a fixed FFT scheme is run serially modulo 32-bit prime numbers. This is also the case in the papers by Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza, Lin-Xiao Wan where a fixed FFT scheme is run in parallel (on GPUs in [3] and on multi-core architecture [7]) modulo Generalized Fermat Prime numbers.

To achieve our goal, we implement and compare two Cooley-Tukey Six-Step fast Fourier transforms and a Cooley-Tukey Four-Step variant against high performing specialized FFTs already implemented in the Basic Polynomial Algebra Subprograms (BPAS) library [1]. We use optimization techniques found in modern day high performance FFT implementations over the field of complex numbers (a context in which computations are performed with floating point number arithmetic whereas our study relies on symbolic computations) like FFTW by Matteo Frigo and Steven G. Johnson [12] and SPIRAL [18] by Markus Pschel, Jos M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson and Nicholas Rizzolo. We start with a Six Step parallel algorithm suggested by Franz Franchetti and Markus Puschel in the Encyclopedia of Parallel Computing [11] and derive two FFT variants, a Six-Step loop-merged variant and a Four-Step loop-merged variant. We implement and compare the FFTs in both C and C++ programming languages and we compare our BPAS finite field C++ implementation against a GNU multiple precision (GMP) implementation. We compare both serial and parallel versions over finite fields with characteristic primes of size 32 bits and larger. In addition to providing a fair comparison between FFT implementations with a varying

degree of specialization, we optimistically hope to achieve a relative degree of performance with C++ as compared to C, and we hope to show how well the different FFTs parallelize and how that relates to the degree of specialization.

1.2 Methodology

The Six-Step Cooley-Tukey decimation-in-time fast Fourier transform

$$DFT_N = L^N_K(I_M \otimes DFT_K)L^N_M D_{K,M}(I_K \otimes DFT_M)L^N_K. \quad N = KM. \tag{1.1}$$

uses explicit transpositions throughout the computation and for each recursion step it passes over the data six times. These data passes make competition against highly optimized serial fast Fourier transforms like FFTW or specialized prime field FFTs like the one by Svyatoslav Covanov [6] challenging. To see if we can affect performance vs fast serial FFTs, we use loop-merging techniques to create a version that we call the loop-merged Six-Step FFT. Finally, we reverse the initial Four-Step FFT to Six-Step FFT manipulation that was used to create the Six-Step FFT in [11] and we create a loop-merged Four-Step FFT version. Using the FFT definitions, we create and optimize three FFT schemes and use them across our FFTs to keep the playing field as level as possible. We loop-unroll, perform code elimination, and minimize data movement, for base case FFT kernels of sizes: 2, 4, 8, 16, 32, and 64. Then, we use the recursive definition to create the three FFTs that reduce a large problem of size $N = N_1N_2$ into N_1 problems of size N_2 and N_2 problems of size N_1. We use a radix-2 split to accommodate two power N. We follow a blocking strategy with respect to data locality and create various block sizes to accommodate the different sized finite field characteristic primes. Using varying sized prime field characteristics we run an experiment to compare the serial performance of our FFTs and we run an experiment to compare the parallel performance of our FFTs. We run FFTs of different sizes using each of the base case kernels.

1.3 Results

We found that the Six-Step explicit FFT realized the most speedup. Serially, the Six-Step FFTs couldn't compete with the serial Four-Step FFTs. We found that the Four-Step Loop-merged variant performed better than both the Six-Step explicit and loop merged versions. In terms of parallelism, the Six-Step explicit FFT on average realized the most speedup. Looking at the overall computation, the areas that benefit the most from parallelism are the areas with more work. First, the twiddle factor subprogram (consisting of point-wise multiplications of finite field elements) is the most work intensive area due to the prime field arithmetic followed by the base case kernels, followed by the stride permutations. In terms of work intensity, we found the stride permutation subprogram to be light on work and we wanted to see what kind of parallel speedup we might achieve by using the loop merging technique to push the stride permutations into the other loops for each recursion step during a computation. Unfortunately, the act of loop merging introduces an additional copy back loop which reduces the overall effectiveness of the loop merge by adding another pass through the data. We found that loop merging adds another level of complexity to the code as well as the parallel computation. Also, it introduces

large two power stride memory accesses in the loop merged versions. A local buffer described in [11] needs to be used to help mitigate the two power stride access issue. Realizing speed up from parallelism for the loop merged variants was more difficult due to cache penalties incurred during the computation. We found that our C++ versions were not very competitive against a high performance serial FFT written in pure C. Nor were they competitive when compared to our own C versions. We found that we could get the best performance from our C code. We found the comparison between the C++ and GMP C implementations to be closer in performance and we attribute the relative closeness of measured time difference to the memory management going on as GMP field elements grow and shrink in size during the computation. The GMP manual states, "mpz_t variables represent integers using sign and magnitude, in space dynamically allocated and reallocated" [13]. The increased complexity of the loop merged versions made parallelism difficult to achieve using the parallel for loop construct and the associated overhead doesn't appear to provide as much benefit for small prime sized transforms. When analyzing the result, the added complexity and implementation time to loop merge and block copy the parallel looped merged versions may have out-weighed the end result. Lastly, due to the blocked nature of our FFTs, a poor choice of block size for the base case can incur significant performance penalties.

1.4 Document outline

This thesis follows a logical progression that introduces topics and includes relevant background information for the reader. It is critical to note that, in Chapters 2, 3, 4, and 5, the background material is taken from the literature. Chapter 6 describes our implementation techniques but we also employ techniques taken from the literature. Chapter 2 includes background on Finite Fields and Modular arithmetic. Chapter 3 includes background on the Tensor Product and introduces the Stride Permutation and we detail its implementation. Chapter 4 describes the Direct Sum and defines the diagonal scaling matrix $D_{k,m}$ or T^n_m *twiddle matrix* and we detail its implementation. Chapter 5 discusses Fast Fourier Transforms. Chapter 6 describes our implementation techniques as well as the loop-merging techniques taken from the literature which we employ. Chapter 7 details our experiment and in Chapter 8 we present our results.

Chapter 2

Finite Field Arithmetic

In this chapter, we define provide some relevant background information about modular arithmetic, finite fields, and primitive roots of unity.

2.1 Symbol usage

The following is the description of the syntax used in the description of algorithms.

- $a = b$ assigns value b to a
- $a == b$ returns true is a is equal to b
- $\vec{x}$ is a vector
- $x[i]$ is the i-th element in $\vec{x}$
- x_i is the i-th element in $\vec{x}$

2.2 Modular arithmetic

In this paper we focus on optimizing the components of the fast Fourier transform which will be used over the different prime fields. In the prime field context, the size of the field characteristic dictates the per-element storage requirements as well as the relative cost of the modular arithmetic. Naturally, the faster we can perform our basic operations, the faster our overall computation. A fast Fourier transform over a prime field, in terms of basic operations, consists of modular addition, modular multiplication, and the movement of data. Of these three, the modular multiplications are the most expensive. The underlying modular arithmetic is discussed more thoroughly in Putting Furer's Algorithm into Practice with the BPAS Library by Linxiao Wang [24].

2.3 Primitive n^{th} root of unity

The primitive n^{th} root of unity is used in a variety of mathematical algorithms and is central to the fast Fourier transform. We represent a primitive n^{th} root of unity with the symbol ω_n. The subscript indicates that ω is an n^{th} primitive root.
Formally, ω is a root of unity if

$$\omega^n = 1$$

and, it is primitive if

$$\omega^x \neq 1 \forall 1 <= x < n.$$

We know from Lagrange's theorem that if n divides $p-1$ then the prime field $\mathbb{Z}/p\mathbb{Z}$ admits a n^{th} primitive root of unity. In [24] the following algorithm is derived and will be used in this paper for finding ω_n. We now have a way to calculate ω_n for use in our fast Fourier transform.

Procedure 1 Primitive Root Of Unity (p, n)

Input: p: a prime number, n: a two-power integer where n divides $p-1$.
Output: ω_n

```
q = (p − 1)/n
d = q(n/2)
c = 0
while c^d ≠ −1 mod p do
    c = randomnumber()
end while
return c^d
```

2.4 Prime fields in the BPAS library

In algebra, a non-empty set $\mathbb{A}$ is a ring whenever $\mathbb{A}$ is endowed with two binary operations denoted $+$ and $\times$ such that, both addition and multiplication are associative and both binary operations admit neutral elements, 0 and 1 respectively. Here, addition must be commutative and every $a \in \mathbb{A}$ admit a symmetrical element $-a$ with respect to addition. Additionally, multiplication is distributive with respect to the addition. The ring $\mathbb{A}$ is commutative if the multiplication of elements $\in \mathbb{A}$ is commutative and that multiplication of a non-zero element $a \in \mathbb{A}$ admits a symmetrical element a^{-1}. A commutative ring $\mathbb{A}$ is called a field.
The residue classes modulo a prime number, p, form a field called a prime field. Prime fields have p elements and are denoted $\mathbb{Z}/p\mathbb{Z}$. Here, the notation $\mathbb{Z}/p\mathbb{Z}$ is used because this is the quotient ring of $\mathbb{Z}$ by the ideal $p\mathbb{Z}$ containing all integers divisible by p where $0\mathbb{Z}$ is the $\{null\}$ singleton. Arithmetic over a prime field is called modular arithmetic.
In modular arithmetic, we use integer representatives of the residue class $\mod p$. To perform a modular arithmetic operation, arithmetic is performed on the integer representatives and the output determines a residue class, which is returned in representative form congruent $\mod p$ [2]. In this paper, we distinguish between two types of prime fields, "Small" prime fields and "Big" prime fields. Small prime fields have as a characteristic, primes represented by ≤ 32

bits and Big prime fields with a characteristic ¡ 32 bits. The following finite fields have been implemented by [24] and are found in the BPAS library.

- SmallPrimeField in C A Set of C functions in the BPAS library implementing arithmetic operations in a prime field of the form GF(p) where p is of machine word size.
- SmallPrimeField C++ Class C++ implementation in the BPAS library of a prime field of the form GF(p) where p is of machine word size.
- BigPrimeField C++ Class C++ implementation in the BPAS library of a prime field of the form GF(p) where p is an arbitrary prime number.
- BigPrimeField GMP Class Set of C functions provided by the GNU Multiple Precision library implementing arithmetic operations in a prime field of the form GF(p) where p is an arbitrary prime number.

2.5 General modular arithmetic algorithms

The following algorithms is reproduced from the Handbook of Applied Cryptography [15]. The following algorithm converts multiple precision numbers to radix b representation.

Procedure 2 Radix b representation

Input:
a: integer $a \geq 0$
b: integer $b \geq 2$
Output: the base b representation $(a_n a_{n-1} \cdots a_1 a_0)_b$, where $n \geq 0$ and $a_n \neq 0$ if $n \geq 1$.
$i = 0$
$x = a$
$q = \lfloor x/b \rfloor$
$a_i = x - qb$
while $a > 0$ **do**
  $i = i + 1$
  $x = q$
  $q = \lfloor x/b \rfloor$
  $a_i = x - qb$
end while
return $(a_i a_{i-1} \cdots a_1 a_0)$

The following algorithm handle multiple precision modular addition and is found in [15].

Procedure 3 Multiple-precision addition

Input:
x: positive integer having $n+1$ base b digits.
y: positive integer having $n+1$ base b digits.
Output: the sum $x + y = (z_{n+1}z_n \cdots z_1z_0)$ in radix b representation

```
c = 0
for i in 0 to n do
    z_i = (x_i + y_i + c) mod b
    if ((x_i + y_i + c) < b) then
        c = 0
    else
        c = 1
    end if
end for
z_{n+1} = c
if z ≥ n then
    z = z − (n)
end if
return (z_{n+1} z_n ··· z_1 z_0)
```

The following algorithm handles subtraction and is found in [15].

Procedure 4 Multiple-precision subtraction

Input:
x: positive integer having $n+1$ base b digits.
y: positive integer having $n+1$ base b digits.
$x \geq y$
Output: the difference $x - y = (z_{n+1}z_n \cdots z_1z_0)$ in radix b representation

```
c = 0
for i in 0 to n do
    z_i = (x_i − y_i + c) mod b
    if ((x_i − y_i + c) ≥ 0) then
        c = 0
    else
        c = −1
    end if
end for
return (z_n z_{n−1} ··· z_1 z_0)
```

The following is a multiple precision multiplication algorithm also found in [15].

Procedure 5 Multiple-precision multiplication

Input:
x: positive integer having $n + 1$ base b digits.
y: positive integer having $m + 1$ base b digits.
Output: the product $x \cdot y = (z_{n+m+1} \cdots z_1 z_0)$ in radix b representation

```
for i from 0 to n + m + 1 do
    z_i = 0
end for
for i from 0 to m do
    c = 0
    for j from 0 to n do
        (uv)_b = z_{i+j} + x_j · y_i + c
        z_{i+j} = v
        c = u
    end for
    z_{i+n+1} = u
end for
return (z_{n+m+1} ··· z_1 z_0)
```

followed by a division algorithm from [15].

Procedure 6 Multiple-precision division

Input:
x positive integer $x = (x_n \cdots x_1 x_0)_b$
y positive integer $y = (y_m \cdots y_1 y_0)_b$
with $n \geq m \geq 1$ and $y_m \neq 0$
Output: the quotient $q = (q_{n-m} \cdots q_1 q_0)_b$ and remainder $r = (r_m \cdots r_1 r_0)_b$ such that $x = qy + r$, $0 \leq r < y$

```
for j from 0 to (n − m) do
    q_j = 0
end for
while (x ≥ y b^{n−m}) do
    q_{n−m} = q_{n−m} + 1
    x = x − y b^{n−m}
end while
for i from n to m − 1 do
    if x_i == y_m then
        q_{i−m−1} = b − 1
    else
        q_{i−m−1} = ⌊(x_i b + x_{i−1})/y_m⌋
    end if
    while (q_{i−m−1}(y_m b + y_{m−1}) > x_i b^2 + x_{i−1} b + x_{i−2}) do
        q_{i−m−1} = q_{i−m−1} − 1
    end while
    x = x − q_{i−m−1} y b^{i−m−1}
    if x < 0 then
        x = x + y b^{i−m−1}
        q_{i−m−1} = q_{i−m−1} − 1
    end if
end for
r = x
return (q, r)
```

Division is an expensive operation and is avoided whenever possible. Next is a classical multiplication algorithm

Procedure 7 Classical modular multiplication

Input:
x positive integer in radix b representation
y positive integer in radix b representation
m the modulus a positive integer in radix b representation
Output: $x \cdot y \bmod m$

```
compute x · y using multiple-precision multiplication
compute the remainder r when x · y is divided by m
return r
```

followed by the Montgomery Multiplication algorithm from [17] but this algorithmic description is from [15].

Procedure 8 Montgomery Multiplication

Input:
x positive integer in radix b representation $x = (x_{n-1} \cdots x_1 x_0)_b$
y positive integer in radix b representation $y = (y_{n-1} \cdots y_1 y_0)_b$
the modulus, a positive integer in radix b representation $m = (m_{n-1} \cdots m_1 m_0)_b$
where $0 \leq x, y < m$, $R = b^n$
where $\gcd(m, b) = 1$, $m' = -m^{-1} \mod b$.
Output: $xyR^{-1} \mod m$
$A = 0$
for i from 0 to $n - 1$ do: **do**
$u_i = (a_0 + x_i y_0)m' \mod b$
$A = (A + x_i y + u_i m)/b$
end for
if $A \geq m$ **then**
$A = A - m$
end if
return A

Modular multiplication, Montgomery form, and the REDC algorithm are discussed thoroughly in [24].

Now that we have a way to initialize our prime fields and our modular arithmetic defined, we can begin the development of our FFT subprograms. We organize our optimization approach into the breaking down of our computation into cache efficient subprograms that can be executed in parallel. We start by deriving subprograms for each of the three major components involved in the computation. The major components we identify as the Stride permutation $L^k_k m$, the Twiddle function $D_{k,m}$, and the DFT_k base case. Each of these components are developed following a cache-oblivious blocking strategy which we analyze in terms of work and cache complexity. Then, we derive a Cooley-Tukey recursive algorithm that lends itself to blocking. It follows a blocking strategy because we can choose the size of the base case or computational 'block' used to terminate the recursion. Blocking size is selected such that the size of the block or working set of elements fit entirely in the cache. Using a blocked recursive algorithm we create an iterative version of the algorithm that oversees the parallel computation. Our goal is to perform the computation in a cache-efficient manner. If each of our three subprograms are cache-efficient then our overall computation should also be cache-efficient.

Chapter 3

Tensor product and the stride permutation

In this chapter, we first review the tensor product which is ubiquitous in FFT literature and fundamental to the description of FFT algorithms. Next, we relate the idea of matrix transposition to a permutation of an indexing set. Then, we describe a few key index permutations that play a key role connecting large Fourier transforms to smaller Fourier transforms.. Finally, we highlight the significance of the stride permutation and we discuss the cache optimal matrix transposition subprogram implemented to perform it.

3.1 Tensor product

In this section we discuss the tensor product which is used when discussing FFT algorithms. Note that I do not claim the proofs and theorems about the tensor product to be mine, they are found in every modern algebra textbook. These are reproduced here slightly altered to suit our discussion from [23], [22], and [19], and are included to aid in the development of the discussion.
The following is modified from [19] to suit our discussion.
Let C^N be a N-dimensional vector space of N-tuples of complex numbers.
A typical point $a \in C^N$ is a column vector

$$\vec{a} = \begin{bmatrix} a_0 \\ . \\ . \\ . \\ a_{N-1} \end{bmatrix}.$$

We say that $\vec{a}$ has size N. If the size of $\vec{a} \in C^N$ is important, we denote $\vec{a}$ as $\vec{a}^N$.
The *tensor product* of two vectors $\vec{a} \in C^M$ and $\vec{b} \in C^L$ is the vector $\vec{a} \otimes \vec{b} \in C^N$, $N = ML$ defined by

$$\vec{a} = \begin{bmatrix} a_0\vec{b} \\ . \\ . \\ . \\ a_{M-1}\vec{b} \end{bmatrix}.$$

Example 2.1 *Tensor product of vectors $\vec{a} \in C^M$ and $\vec{b} \in C^L$*

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \otimes \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \\ a_2 b_0 \\ a_2 b_1 \end{bmatrix}.$$

For vectors $\vec{a}, \vec{b}, \vec{c}$ of appropriate sizes

$$(\vec{a} + \vec{b}) \otimes \vec{c} = \vec{a} \otimes \vec{c} + \vec{b} \otimes \vec{c} \tag{3.1}$$

$$\vec{a} \otimes (\vec{b} + \vec{c}) = \vec{a} \otimes \vec{b} + \vec{a} \otimes \vec{c} \tag{3.2}$$

The tensor product is not commutative. In general,

$$\vec{a} \otimes \vec{b} \neq \vec{b} \otimes \vec{a}.$$

Consider tensor products $\vec{a} \otimes \vec{b}$ and $\vec{b} \otimes \vec{a}$ with $\vec{a} \in C^M$ and $\vec{b} \in C^L$. Identify $\vec{a} \otimes \vec{b}$ with

$$Mat_{M \times L}(\vec{a} \otimes \vec{b}) = \left[a_0 \vec{b}, \ldots, a_{M-1} \vec{b}\right],$$

and $\vec{b} \otimes \vec{a}$ with

$$Mat_{L \times M}(\vec{b} \otimes \vec{a}) = \left[b_0 \vec{a}, \ldots, b_{L-1} \vec{a}\right],$$

we see that interchanging the order of the tensor product corresponds to a matrix transposition.

$$Mat_{L \times M}(\vec{b} \otimes \vec{a}) = (Mat_{M \times L}(\vec{a} \otimes \vec{b}))^T.$$

In example 2.1,

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \otimes \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

corresponds to the 3×2 matrix

$$\begin{bmatrix} a_0 b_0 & a_1 b_0 & a_2 b_0 \\ a_0 b_1 & a_1 b_1 & a_2 b_1 \end{bmatrix}.$$

and

$$\begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \otimes \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

corresponds to the 2×3 matrix

$$\begin{bmatrix} a_0 b_0 & a_0 b_1 \\ a_1 b_0 & a_1 b_1 \\ a_2 b_0 & a_2 b_1 \end{bmatrix}.$$

and

$$\begin{bmatrix} a_0b_0 & a_1b_0 & a_2b_0 \\ a_0b_1 & a_1b_1 & a_2b_1 \end{bmatrix}^t = \begin{bmatrix} a_0b_0 & a_0b_1 \\ a_1b_0 & a_1b_1 \\ a_2b_0 & a_2b_1 \end{bmatrix}.$$

Denote by $e_m^M, 0 \leq m < M$, the vector of size M with 0 everywhere except the m-th spot which contains a 1. The resulting set of vectors

$$\{e_m^M : 0 \leq m < M\}$$

is the standard basis of C^M.
Set $N = ML$ and form the tensor products $e_m^M \otimes e_l^L, 0 \leq m < M, 0 \leq l < L$. Since

$$e_{l+mL}^N = e_m^M \otimes e_l^L, 0 \leq m < M, 0 \leq l < L,$$

the set

$$\{e_m^M \otimes e_l^L : 0 \leq m < M, 0 \leq l < L\}$$

is the standard basis of C^N. As $\vec{a}$ runs over all vectors of size M and $\vec{b}$ runs over all vectors of size L, the *tensor products* $\vec{a} \otimes \vec{b}$ span the space C^N.

Theorem 3.1.1 *From [22]*
If A is a $R \times S$ matrix and B an $M \times L$ matrix, then

$$(A \otimes B)(\vec{a} \otimes \vec{b}) = A\vec{a} \otimes B\vec{b},$$

for any vectors $\vec{a}$ and $\vec{b}$ of sizes S and L.

Proof The vector $\vec{a} \otimes \vec{b}$,

$$\begin{bmatrix} a_0\vec{b} \\ a_1\vec{b} \\ . \\ . \\ . \\ a_{s-1}\vec{b} \end{bmatrix} \tag{3.3}$$

consists of consecutive segments

$$a_0\vec{b}, a_1\vec{b}, ..., a_{s-1}\vec{b},$$

each of size L. Since the $M \times SL$ matrix formed by the M rows of $A \otimes B$ is

$$\begin{bmatrix} a_{0,0}B & a_{0,1}B & \cdots & a_{0,L-1}B \end{bmatrix},$$

we have that the vector of size M formed from the first M components of $(A \otimes B)(\vec{a} \otimes \vec{b})$ is

$$(a_{0,0}a_0 + a_{0,1}a_1 + \cdots + a_{0,s-1}a_{s-1})B\vec{b}.$$

continuing in this way proves the theorem.

Theorem 3.1.2 *From [22]*
If A and C are $M \times M$ matrices and B and D are $L \times L$ matrices, then

$$(A \otimes B)(C \otimes D) = AC \otimes BD.$$

Proof Take vectors $\vec{a}$ and $\vec{b}$ of sizes M and L. By theorem 2.1 and, in light of the preceding discussion,

$$(A \otimes B)(C \otimes D)(\vec{a} \otimes \vec{b}) = (A \otimes B)(C\vec{a} \otimes D\vec{b}) = AC\vec{a} \otimes BD\vec{b}, \tag{3.4}$$

proving the theorem.

3.2 Stride permutations

The Stride permutation is a major component of the Cooley-Tukey six-step FFT. It is represented by the symbols L^N_K and L^N_M in

$$DFT_N = L^N_K(I_M \otimes DFT_K)L^N_M D_{K,M}(I_K \otimes DFT_M)L^N_K. \quad N = KM.$$

A Stride permutation is simply another name for a transposition. We call this type of transposition a *stride permutation* because it permutes the indexing set of an input vector at a given stride.

3.2.1 Matrix transposition as a permutation of an indexing set

A matrix transpose can be thought of as a permutation of the indexing set. Consider a $M \times L$ matrix

$$A = [a_{m,l}]_{0 \le m < M, 0 \le l < L}.$$

for any $0 \le x, y < N$, where $N = ML$, we can write uniquely

$$x = m + lM, \quad 0 \le m < M, 0 \le l < L.$$

$$y = l + mL, \quad 0 \le l < L, 0 \le m < M,$$

The vector $\vec{a}$ formed by the matrix A has components given by

$$a_x = a_{m,l}, \quad x = l + mL, \quad 0 \le m < M, 0 \le l < L.,$$

while the vector $\vec{b}$ formed by the matrix A^t has components given by

$$b_y = a_{m,l}, \quad y = m + lM, \quad 0 \le m < M, 0 \le l < L.$$

These vectors correspond to permutations of the indexing set,

$$\phi(l + mL) = (m + lM)$$

and we have

$$a_x = b_{\phi(x)}, \quad 0 \le x < N.$$

Example Take $M = 2, L = 4$ and

$$\vec{z} = \begin{bmatrix} z_0 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 \end{bmatrix}.$$

We have

$$\phi = (\ 0 \quad 2 \quad 4 \quad 6 \quad 1 \quad 3 \quad 5 \quad 7\ )$$

and

$$\vec{a} = \begin{bmatrix} z_0 & z_2 & z_4 & z_6 & z_1 & z_3 & z_5 & z_7 \end{bmatrix}$$

To form $\vec{a}$ from $\vec{z}$, we 'stride' through $\vec{z}$ with length two. In general, to form $\vec{a}$ from $\vec{z}$, we first initialize at z_0, the 0^{th} component of $\vec{z}$, and then stride through $\vec{z}$ with length the size of M. After a pass through $\vec{z}$, we re-initialize at the 1^{st} component of $\vec{z}$, and again stride through $\vec{z}$ with length of size M. After another pass through $\vec{z}$, we re-initialize now at the 2^{nd} component of $\vec{z}$. We repeat this permutation of data until we form $\vec{a}$.

Example 2.2 Take $M = 4, L = 2$ and

$$\vec{z} = \begin{bmatrix} z_0 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 \end{bmatrix}.$$

We have

$$\phi = (\ 0 \quad 4 \quad 1 \quad 5 \quad 2 \quad 6 \quad 3 \quad 7\ )$$

and

$$\vec{a} = \begin{bmatrix} z_0 & z_4 & z_1 & z_5 & z_2 & z_6 & z_3 & z_7 \end{bmatrix}$$

To form $\vec{a}$ from $\vec{z}$, we 'stride' through $\vec{z}$ with length 4.

This procedure is an example of a *stride permutation*.

3.2.2 Radix-2 stride permutations

A radix-2 stride permutation is a permutation with a stride of 2. In general, the term radix-2 means to relate a problem of size n to a problem of half the original size. The radix-2 stride permutation is appropriately named as it is part of the key to relating a Fourier transform of size N to a Fourier transform of size $N/2$. The radix-2 stride permutation is sometimes called the *even-odd* sort permutation. Applying the radix-2 stride permutation to $\vec{x}$ results in the vector being arranged with the even-indexed columns first followed by the odd-indexed columns second.

$$y = L_2^N[x_0, x_1, \ldots, x_{n-1}] \leftrightarrow y = \begin{bmatrix} [x_0, x_2, \ldots, x_{n-1}] \\ [x_1, x_3, \ldots, x_{n-1}] \end{bmatrix}. \tag{3.5}$$

The action of the radix-2 stride permutation on a matrix $A \in \mathbb{C}^{n\times n}$ is simply the matrix A arranged with its even-indexed columns grouped first followed by the odd-indexed columns second.

$$AL_2^N = \left[\ A_{even} \ \middle|\ A_{odd}\ \right].$$

If the radix-2 stride permutation transposes $\vec{x}$ as a $2 \times M$ array, then the inverse of the radix-2 stride permutation transposes $\vec{x}$ as an $M \times 2$ array. The inverse of the radix-2 stride permutation

is sometimes called the *perfect shuffle*.
if $N = 8, N = 2M$ and $\vec{x} = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]$ then,

$$y = L_2^8 x = [x_0, x_2, x_4, x_6, x_1, x_3, x_5, x_7]$$

and

$$x = \left[L_2^8\right]^T y = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7].$$

3.2.3 Radix-P stride permutations

The radix-p stride permutation is key to expressing a Fourier transform of size N as P Fourier transforms of size N/P. The radix-p stride permutation is sometimes called the *Mod P Sort permutation* and it is the generalization of the radix-2 stride permutation above. Continuing with Van Loan's approach,
If $N = PM$ and $x \in \mathbb{A}^N$ then the radix-p stride permutation is defined as:

$$L_P^N x = \begin{bmatrix} \begin{bmatrix} x_0, x_{0+p}, \ldots, x_{n-1} \end{bmatrix} \\ \begin{bmatrix} x_1, x_{1+p}, \ldots, x_{n-1} \end{bmatrix} \\ \cdots \\ \begin{bmatrix} x_{p-1}, x_{p-1+p}, \ldots, x_{n-1} \end{bmatrix}. \end{bmatrix} \tag{3.6}$$

Lemma 3.2.1 *Suppose $N = PM$ and that $x, y \in \mathbb{A}^n$. If $y = L_P^N x$, then*

$$y_{M\times P} = x_{P\times M}.$$

Put another way,

$$y_{\beta M+\alpha} = x_{\alpha P+\beta}$$

for all $0 \le \alpha < M$ and $0 \le \beta < P$.
Thus, if

$$x_{P\times M} \equiv \begin{bmatrix} x_{0,0}, \ldots, x_{p-1,0} \\ \cdots \\ x_{0,m-1}, \ldots, x_{p-1,m-1} \end{bmatrix}$$

and

$$y_{M\times P} \equiv \begin{bmatrix} y_{0,0}, \ldots, y_{m-1,0} \\ \cdots \\ y_{0,p-1}, \ldots, y_{m-1,p-1} \end{bmatrix},$$

then

$$X_{\beta,\alpha} = Y_{\alpha,\beta}.$$

Proof *If $y = L_P^{PM} x$ then from* 2.6 *we have*

$$y_{M\times P} = \left[\; x_0, x_{0+p}, \ldots, x_{n-1} \;\middle|\; x_1, x_{1+p}, \ldots, x_{n-1} \;\middle|\; \cdots \;\middle|\; x_{p-1}, x_{p-1+p}, \ldots, x_{n-1} \;\right] = x_{P\times M}.$$

Since

$$y_{\beta m+\alpha} = \left[y_{m\times p}\right]_{\beta,\alpha} = \left[x_{p\times m}\right]_{\beta,\alpha} = x_{\beta+\alpha p},$$

the lemma is established.

Mod P Stride Permutation Example

$$L_P^N x = \left[\begin{array}{cccccccccccc} x_0 & x_3 & x_6 & x_9 & x_1 & x_4 & x_7 & x_{10} & x_2 & x_5 & x_8 & x_{11} \end{array} \right], \quad N = 12, \quad P = 3, \quad M = 4.$$

The *Mod P shuffle permutation* is the inverse of the Mod P sort permutation. If $N = PM$ and $L_P^{P\times M}$ transposes x as an $P \times M$ matrix then $\left[L_P^{P\times M}\right]^T$ transposes x as an $M \times P$ matrix.

3.3 Stride permutation subprogram

Now that we have seen how the stride permutation works, we will look at how we implement the function as a cache-optimal subprogram. Recall that the stride permutation is a major component of the Cooley-Tukey six-step FFT. It is represented by the symbol L in

$$DFT_N = L_K^N(I_J \otimes DFT_K)L_J^N D_{K,J}(I_K \otimes DFT_J)L_K^N.$$

Proper handling of this function is critical to realizing an efficient FFT. The best suited matrix transpose as of the writing of this paper is the one described in Cache-Oblivious Algorithms by Matteo Frigo et al [5]. It is the cache-oblivious rectangular transpose. It uses $\mathrm{O}(MN)$ work and incurs $\mathrm{O}(1 + MN/L)$ cache misses, which they show to be optimal. It is a recursive function that takes a large rectangular matrix and divides the matrix along the larger of the dimensions calling itself on the subsequent halves. The recursion halts at a base case where all the elements being transposed as well as the destination auxiliary array fit in the cache. Given a rectangular matrix A of size $M \times N$ and an auxiliary $L \times M$ matrix B, If $M \geq N$ then $Transpose(A, B)$ partitions A vertically and B horizontally

$$A = (A_1 A_2), \quad B = \left(\begin{array}{c} B_1 \\ B_2 \end{array} \right)$$

and recursively executes $Transpose(A_1, B_1)$ and $Transpose(A_2, B_2)$. Otherwise, if $L > M$ then it partitions A horizontally and B vertically

$$A = \left(\begin{array}{c} A_1 \\ A_2 \end{array} \right), \quad B = (B_1 B_2)$$

and recursively executes the transpositions. The following was proven by Matteo Frigo et al. in [5] and is reproduced here for convenience.

Lemma 3.3.1 *The cache-oblivious matrix transpose algorithm involves* $\mathrm{O}(MN)$ *work and incurs* $\mathrm{O}(1 + mn/L)$ *cache misses for an* $M \times N$ *matrix.*

$$\left[L_3^{12} x\right]_{4\times 3} = \left[\begin{array}{ccc} x_0 & x_1 & x_2 \\ x_3 & x_4 & x_5 \\ x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} \end{array} \right] = x_{3\times 4}.$$

Figure 3.1: Illustration of lemma 2.2.1

Proof To transpose a matrix we have to visit each element at least once. There are MN elements. Let $Q(m,n)$ be the cache complexity for transposing a $M \times N$ matrix.
Let α be a constant sufficiently small such that sub-matrices of sizes $m \times n$ and $n \times m$ where $max\{m,n\} \leq \alpha L$ fit entirely in cache even if each row is stored in a different cache line. Distinguishing between three cases:

Case 1: $max\{m,n\} \leq \alpha L$. Both matrices fit in $\mathrm{O}(1) + 2mn/L$ lines. From the choice of α, the number of lines is at most Z/L. Therefore $Q(m,n) = \mathrm{O}(1 + mn/L)$.

Case 2: $m \leq \alpha L < n$ or $n \leq \alpha L < m$. Suppose first that $m \leq \alpha L < n$, The transposition algorithm partitions the larger dimension by 2 and recursively calls itself until at some point n falls into the range $\alpha L/2 \leq n < \alpha L$ and the whole problem fits in the cache. At this point, the input array is n rows $\times$ m columns in size, it is laid out in row major order and occupies contiguous memory locations requiring at most $\mathrm{O}(1 + nm/L)$ cache misses to be read. Writing to the output array, which has nm elements in m rows, where in the worst case every row lies on a different cache line, incurs a cost of at most $\mathrm{O}(m + nm/L)$. Since $n \geq \alpha L/2$, the total cache complexity for this base case is $\mathrm{O}(1 + m)$. These observations produce the following recurrence

$$Q(m,n) \leq \begin{cases} \mathrm{O}(1+m) & if n \in [\alpha L/2, \alpha L], \\ 2Q(m,n/2) + \mathrm{O}(1) & otherwise; \end{cases}$$

whose solution is $Q(m,n) = \mathrm{O}(1 + mn/L)$. The case $n \leq \alpha L < m$ is analogous.

Case 3: $m,n > \alpha L$. As in Case 2, at some point in the recursion, both n and m will fall in the range $[\alpha L/2, \alpha L]$. The whole problem then fits in cache and can be solved in at most $\mathrm{O}(m + n + mn/L)$ cache misses. The cache complexity is described with the following recurrence

$$Q(m,n) \leq \begin{cases} \mathrm{O}(m+n+mn/L) & if m,n \in [\alpha L/2, \alpha L], \\ 2Q(m/2,n) + \mathrm{O}(()\,1) & if m \geq n, \\ 2Q(m,n/2) + \mathrm{O}(()\,1) & otherwise; \end{cases}$$

whose solution is $Q(m,n) = \mathrm{O}(1 + mn/L)$.

Theorem 3.3.2 *The cache-oblivious matrix-transpose algorithm is asymptotically optimal.*

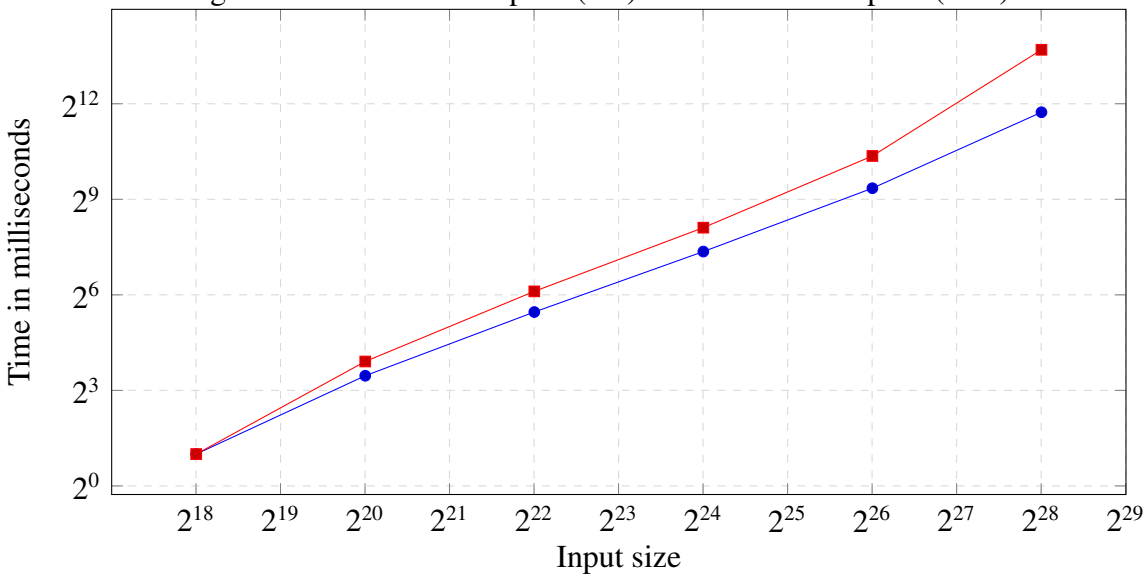
Proof For an $M \times N$ matrix, the matrix transposition algorithm must write to mn distinct elements,which occupy at least $[mn/L] = \Omega(1 + mn/L)$ cache lines.

Below we implement and compare the optimal cache oblivious recursive transpose algorithm to the well known iterative blocked transpose algorithm[5]. The following listing contains a comparison of both the iterative blocked transpose and the recursive transpose described in [5] implemented in C. Following the above result, we use the cache-oblivious recursive transpose algorithm as a subprogram to perform the stride permutations found in the Cooley-Tukey six-step FFT.

```
void blocked_transpose(long int* A, int m, int n, long int* B, int
    blocksize){
     for (int i = 0; i < n; i += blocksize)
       for (int j = 0; j < m; j += blocksize)
         for (int k = i; k < i + blocksize; k++)
           for (int l = j; l < j + blocksize; l++)
             B[k+l*n] = A[l+k*m];
     for (long int i=0;i<m*n;i++)
       A[i]=B[i];
}
```

```
void rec_transpose(long int* A, int rowstart, int numrows,int rowsize, int
    colstart, int numcols,int colsize, long int* B, int basecase){
    if (numrows*numcols<=2*basecase){
       transpose(&A[colsize*rowstart+colstart], numrows, numcols, &B[colsize
    *rowstart+colstart]);
    }else if (numrows>=numcols){
       rec_transpose(A, rowstart, numrows/2, rowsize, colstart, numcols,
    colsize, B, basecase);
       rec_transpose(A, rowstart + numrows/2, numrows/2, rowsize, colstart,
    numcols, colsize, B, basecase);
    }else{
       rec_transpose(A, rowstart, numrows, rowsize, colstart, numcols/2,
    colsize, B, basecase);
       rec_transpose(A, rowstart, numrows, rowsize, colstart+numcols/2,
    numcols/2, colsize, B, basecase);
    }
}
```

Figure 3.2: Blocked transpose (red) vs recursive transpose (blue)



Chapter 4

Twiddle factors

In this section we discuss the twiddle factors that appear in Cooley-Tukey FFT algorithms. The twiddle factors matrix is represented by the symbol $D_{K,J}$ in

$$DFT_N = L^N_K(I_J \otimes DFT_K)L^N_J D_{K,J}(I_K \otimes DFT_J)L^N_K.$$

In order to understand the structure of the twiddle factors matrix we need to first define the diagonal scaling matrix $D_{K,J}$ and for that we need the direct sum operator $\oplus$. Then, we will look at the structure of the $D_{K,J}$ twiddle factor matrix for different values of K,J and show how we use them in practice. Then we detail our implementation and discuss the performance of our twiddle factor function.

4.1 Direct sum

First we need to define the direct sum operator $\oplus$. For matrices A and B, operator $\oplus$ is defined as follows

$$A \oplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}.$$

Example 3.1 For n matrices $A_0 \ldots A_{n-1}$, the $\oplus$ sum of them is defined as

$$\oplus_{i=0}^{n-1} A_i = A_0 \oplus A_1 \oplus \cdots \oplus A_{n-1} = \begin{bmatrix} A_0 & & & & \\ & A_1 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & A_{n-1} \end{bmatrix}$$

where omitted values are zeros.

4.2 Diagonal scaling

Next we need to define the diagonal scaling matrix $D = diag(d) = diag(d_0, d_1, \ldots, d_{n-1})$.
The diagonal scaling matrix D is the $n \times n$ diagonal matrix with diagonal entries $d_0, d_1, \ldots, d_{n-1}$.

$$D = diag(d_0, d_1, \ldots, d_{n-1}) = d_0 \oplus d_1 \oplus \cdots \oplus d_{n-1} = \begin{bmatrix} d_0 & & & & & \\ & d_1 & & & & \\ & & \cdot & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & & d_{n-1} \end{bmatrix}.$$

The application of D to a vector $\vec{x}$ of size n is the point-wise multiplication of the vector $\vec{d}$ and the vector $\vec{x}$.

$$D\vec{x} = \begin{bmatrix} d_0 x_0 & & & & & \\ & d_1 x_1 & & & & \\ & & \cdot & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & & d_{n-1} x_{n-1} \end{bmatrix}.$$

4.3 $D_{K,J}$ twiddle matrices

The $D_{K,J}$ matrix or twiddle matrix is defined as

$$D_{K,J} = \oplus_{k=0}^{K-1} diag(I_j, \Omega_{K,J}, \ldots, \Omega_{K,J}^{K-1}) \tag{4.1}$$

where $\Omega_{K,J} = diag(1, \omega_n, \ldots, \omega_n^{J-1})$ and ω_n is a primitive n^{th} root of unity.

Now, we look at two types of twiddle matrices relevant to our factorization strategy for the generic six-step FFT. First is the $D_{2,J}$ twiddle matrix which is otherwise known as the radix-2 butterfly operator, and the second is the $D_{K,J}$ twiddle matrix which is otherwise known as the radix-k butterfly operator.

4.4 The $D_{2,K}$ twiddle matrix

The $D_{2,2}$ twiddle matrix is a 4×4 diagonal matrix. By definition, the non-diagonal elements in these matrices are zeros. So we know that the action of this $n \times n$ diagonal matrix on an input vector with n elements results in a point-wise multiplication operation. So, for practical purposes, we store these sparse $n \times n$ matrices as contiguous element vectors of length n. To illustrate, the 16 element $D_{2,2}$ twiddle matrix

$$D_{2,2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \omega^2 \end{bmatrix}, \tag{4.2}$$

in practice, it is represented as a 4 element contiguous vector

$$D_{2,2} \equiv \vec{d} = [1, 1, 1, \omega^2]. \tag{4.3}$$

Similarly by definition, the $D_{2,4}$ twiddle matrix is a 8×8 diagonal matrix. It also has 8 non-zero elements. So

$$D_{2,4} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^3 \end{bmatrix} \tag{4.4}$$

in practice becomes

$$D_{2,4} \equiv \vec{d} = [1, 1, 1, 1, 1, \omega, \omega^2, \omega^3]. \tag{4.5}$$

Following the pattern, the $D_{2,8}$ twiddle matrix is a 16×16 diagonal matrix

$$D_{2,8} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^7 \end{bmatrix} \tag{4.6}$$

and has the non-zero entries represented in vector form as

$$D_{2,8} = \left[1, 1, 1, 1, 1, 1, 1, 1, 1, \omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6, \omega^7\right]. \tag{4.7}$$

4.5 $D_{K,J}$ twiddle matrix

The $D_{K,J}$ twiddle matrix is built from the same definition as $D_{2,K}$ and has a similar structure. The obvious difference is that instead of building the diagonal from 2 sub-vectors like the

radix-2 twiddle, we build it from K sub-vectors. Large values of K make depiction impractical so we show a small value K example followed by larger twiddle matrices in contiguous vector form.

Example $D_{4,4}$ Twiddle Matrix

$$D_{4,4} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \omega & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \omega^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^6 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^6 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^9
\end{bmatrix}$$

$D_{4,4}$ is represented in vector form as

$$D_{4,4} = \vec{d} = \left[1, 1, 1, 1, 1, \omega, \omega^2, \omega^3, 1, \omega^2, \omega^4, \omega^6, 1, \omega^3, \omega^6, \omega^9\right].$$

Example $D_{4,8}$ Twiddle Matrix $D_{4,8}$ is represented in contiguous vector form as

$$D_{4,8} = \vec{d} = \begin{matrix}
[1, & 1, & 1, & 1, & 1, & 1, & 1, & 1, \\
1, & \omega^1, & \omega^2, & \omega^3, & \omega^4, & \omega^5, & \omega^6, & \omega^7, \\
1, & \omega^2, & \omega^4, & \omega^6, & \omega^8, & \omega^{10}, & \omega^{12}, & \omega^{14}, \\
1, & \omega^3, & \omega^6, & \omega^9, & \omega^{12}, & \omega^{15}, & \omega^{18}, & \omega^{21}]
\end{matrix}.$$

4.6 Twiddle factors subprogram

The action of the twiddle matrix on an input vector is a point-wise multiplication.

In order to follow the blocking strategy, we start by breaking up the point-wise multiplication into the same size as the chosen base case. For if we design all our subprograms to be cache-oblivious then the resulting FFT should also be cache-oblivious. We look at three versions that follow the same blocking strategy. The first is an online version that computes or scales the twiddle factor mid-computation. The other two, one recursive and one iterative, use a precomputed twiddle vector.

4.7 Online twiddle function

The function loops over the $n \times m$ elements of the input vector multiplying each element by the appropriately scaled value of ω. To minimize cache misses we follow the blocking strategy. Considering the overall blocking strategy, this version of the twiddle function requires m such that at least $2m$ elements fit entirely in cache. This way, each iteration of the outer loop results in the inner loop requesting the first sub-vector of size $2m$ elements. This results in the first access of each inner loop causing a cache miss as the required line is not yet loaded into cache. The next $2m - 1$ accesses are hits. One cache miss to fetch the $2m$ elements of the inner loop each time results in a $O(1 + n/2)$ cache miss rate.

```
void online_twiddle(ELEMENTS* vector, int m, int n, ELEMENTS w,struct
    p_data P){
    ELEMENTS t;
    for (int i=0;i<n;i++){
        for (int j=0;j<m;j++){
        t = POW(w,(i*j),P);
        vector[i*m+j]=MULTIPLY(&vector[i*m+j],&t,P);
        }
    }
}
```

4.8 Offline twiddle function

The next approach we look at is the offline twiddle function. We take an iterative approach. Here we afford ourselves an auxiliary array to store precomputed values of ω (twiddle factors). We now need both the input sub-vector elements and the twiddle factors sub-vector elements to fit in cache for each iteration of the inner loop. So this function iterates over the $J \times K$ elements in J K-sized blocks.

Lets look at cache misses. During each iteration of the outer loop we perform operations on 2K elements, K from the input vector and K from the twiddle factors vector. We will have a cache miss for each of the K element sub vector at J iterations of the inner for loop as neither of them are in cache. So for this twiddle function we have a O(()1 + 2(J)) cache miss rate.

```
void offline_twiddle(ELEMENTS* vector, int m, int n, ELEMENTS* omegas,
    struct p_data P){
    int i,j,idx;
    for (i=0;i<n;i++){
        for (j=0;j<m;j++){
            idx = i*m+j;
            vector[idx]=MULTIPLY(&vector[idx],&omegas[idx],P);
        }
    }
}
```

4.9 Twiddle pre-computation

If $n = k^q$ and $m = n/k$ then the following algorithm computes the vector of vectors representing each diagonal twiddle matrix $D_{k,m}$ required for each level of the computation tree.

```
ELEMENTS** precomputeOmegas(int N, int radix, ELEMENTS w){
    ELEMENTS **D;
    int i,j,k,levels,m;
    ELEMENTS omega;
    levels = (int)(log(N)/log(radix));
    D = malloc(sizeof(ELEMENTS*)*(levels));
    m = N;
    omega=w;
    for (j=1;j<=levels;j++){
        ELEMENTS *Dj = D[j-1] = malloc(sizeof(ELEMENTS)*m);
        for (i=0;i<radix;i++)
            for (k=0;k<m/radix;k++)
                *(Dj++) = pow(omega,k*i));

        omega *= omega;
        m /= radix;
    }
    return D;
}
```

If $n = k^q$ and $m = n/k$ then the following algorithm computes the vector of vectors representing each diagonal twiddle matrix $D_{2,k}$ required for each level of base case DFTs in the computation tree.

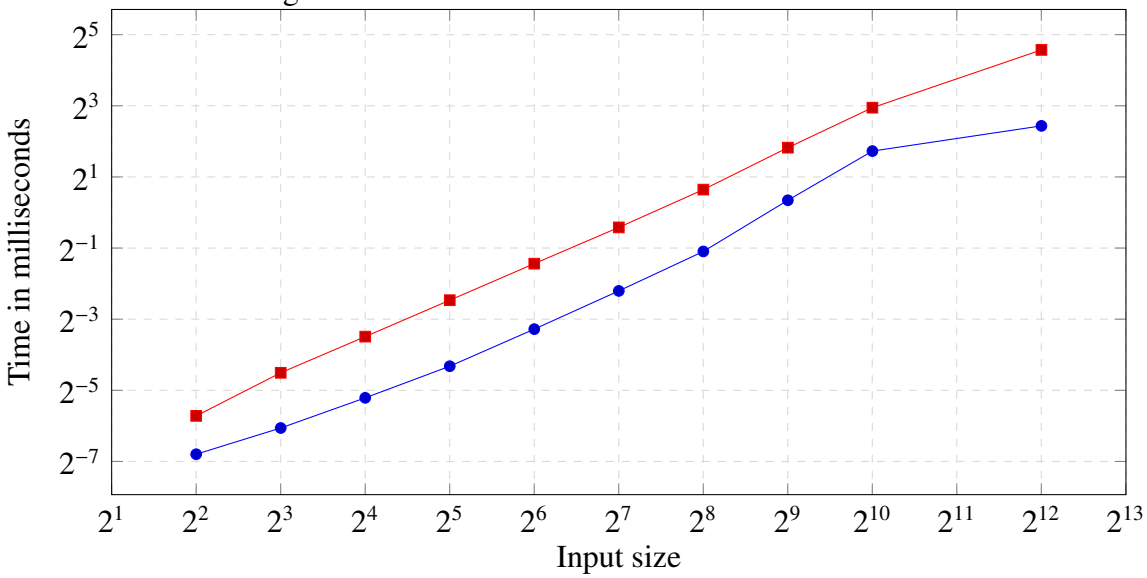
```
ELEMENTS** precomputeBasecaseOmegas(int N, int radix, ELEMENTS w){
    ELEMENTS **D;
    int i,j,k,levels,m;
    ELEMENTS omega;
    levels = (int)(log(N)/log(radix));
    D = malloc(sizeof(ELEMENTS*)*(levels));
    m = N;
    omega=w;
    for (j=1;j<=levels;j++){
        ELEMENTS *Dj = D[j-1] = malloc(sizeof(ELEMENTS)*radix);
        for (i=0;i<2;i++)
            for (k=0;k<radix;k++)
                *(Dj++) = pow(omega,k*i));

        omega *= omega;
        m /= radix;
    }
    return D;
}
```

4.10 Twiddle function comparison

Below is a graph of their comparison.

Figure 4.1: Offline twiddle blue vs online twiddle red



Chapter 5

The Fourier Transform

In this chapter, we first recall the definition of the Fourier transform. Then, we introduce the DFT matrix and show how we can relate a DFT of size n to a DFT of size $n/2$. After, we discuss the idea behind the Cooley-Tukey factorization and introduce relevant factorizations of the FFT.

5.1 The DFT matrix

Let $\mathbb{A}$ be a ring, and $\omega \in \mathbb{A}$ be an n^{th} primitive root of unity; $\omega_n^n = 1$. Let n=jk and recall that the n-point discrete Fourier transform or (DFT_n) on a vector $\vec{x}$ is a matrix-vector product. If $y = [y_0, y_1, \ldots, y_{n-1}]$ is the DFT of $x = [x_0, x_1, \ldots, x_{n-1}]$ for k = 0, 1, ..., n-1 then we have

$$y_k = \sum_{j=0}^{n-1} \omega_n^{jk} x_j. \tag{5.1}$$

So

$$y = DFT_n x,$$

where $DFT_n = \left[DFT_{jk}\right]$ and $\left[DFT_{jk}\right] = \omega_n^{jk}$ is the $n \times n$ DFT matrix. Thus,

$$DFT_1 = [1], \qquad DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \text{ and } DFT_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}$$

We can see that the straightforward computation of a N-point Fourier transform requires a number of arithmetic operations proportional to N^2.

5.2 DFT_4 in terms of DFT_2

The following example is borrowed from Van Loan's book [22].
If we look at the DFT matrix when $n = 4$,

$$DFT_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}.$$

where $\omega = \omega_4 = exp(-2\pi i/4) = -i$. Since $\omega^4 = 1$, it follows that

$$DFT_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

Now, if we let the even-odd sort 4×4 permutation L_2 be

$$L_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and note that

$$DFT_4 L_2 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega & \omega^3 \\ \hline 1 & 1 & \omega^2 & \omega^2 \\ 1 & \omega^2 & \omega^3 & \omega \end{array}\right] = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array}\right]$$

is just DFT_4 with its even-indexed columns grouped first. As I learned in [22], the key to connecting a DFT_4 to a DFT_2 is to consider this permutation of DFT_4 as a 2×2 block matrix, for if we define

$$D_2 = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix} = diag(1, \omega_4)$$

and recall that

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

then

$$DFT_4 L_2 = \left[\begin{array}{c|c} DFT_2 & D_2 DFT_2 \\ \hline DFT_2 & -D_2 DFT_2 \end{array}\right].$$

Which means that each block of DFT_4 is either DFT_2 or a diagonal scaling of DFT_2.
We see that the stride permutation that grouped even-indexed columns first followed by the odd-indexed columns was one part of the mechanism for connecting a DFT_4 to a DFT_2 and the diagonal scaling matrix or "twiddle" matrix

$$D_2 = \begin{bmatrix} 1 & 0 \\ 0 & \omega_4 \end{bmatrix} = diag(1, \omega_4)$$

the other. These operations are used to establish a general version, assuming n is even, that connects DFT_n to $DFT_{n/2}$.

5.3 The Fast Fourier Transform

In the fundamental work of J.W. Cooley and J.W. Tukey [4], the two authors describe a fast Fourier transform (FFT) algorithm that significantly reduces the computational cost from N^2 to an operational count proportional to $N \log N$. The idea was to exploit sparse factorizations of the DFT matrix to quickly compute a two-power n-point DFT by reducing it to a pair of half sized $n/2$-point DFTs. Repetition of the reduction process is called a radix-2 FFT.

5.3.1 Radix-2 FFTs

To best illustrate, we will continue following Van Loan's approach. Using Theorem 1.2.1 from his book,
if $n = 2m$ and $D_m = diag(1, \omega_n, \ldots, \omega_n^{m-1})$ then

$$DFT_n L_n = \begin{bmatrix} DFT_m & D_m DFT_m \\ DFT_m & -D_m DFT_m \end{bmatrix} = \begin{bmatrix} I_m & D_m \\ I_m & -D_m \end{bmatrix} (I_2 \otimes DFT_m).$$

If p and q satisfy $0 \le p < m$ and $0 \le q < m$, then

$$\begin{aligned} [DFT_n L_n]_{pq} &= \omega_n^{p(2q)} &&= \omega_m^{pq} &&= [DFT_m]_{pq}, \\ [DFT_n L_n]_{p+m,q} &= \omega_n^{(p+m)(2q)} &&= \omega_m^{(p+m)q} &&= [DFT_m]_{pq}, \\ [DFT_n L_n]_{p,q+m} &= \omega_n^{p(2q+1)} &&= \omega_n^p \omega_m^{pq} &&= [D_m DFT_m]_{pq}, \\ [DFT_n L_n]_{p+m,q+m} &= \omega_n^{(p+m)(2q+1)} &&= -\omega_n^{p(2q+1)} &&= [-D_m DFT_m]_{pq}. \end{aligned}$$

Using the verifiable facts that $\omega_n^2 = \omega_m$ and $\omega_n^m = -1$, the above four equations confirm the 4 m by m blocks of DFT_n have the desired structure. The following is proof of Van Loan's corollary 1.2.2 [22]
If n is even then the splitting can be applied recursively and we divide and conquer our way down to DFTs of size 1.

Corollary 5.3.1 (Van Loan's corollary 1.2.2) *if $n = 2m$ and $x \in \mathbb{A}$ then*

$$DFT_n \vec{x} = \begin{bmatrix} I_m & D_m \\ I_m & -D_m \end{bmatrix} \begin{bmatrix} DFT_m [x_0, x_2, \ldots, x_{n-1}] \\ DFT_m [x_1, x_3, \ldots, x_{n-1}] \end{bmatrix}$$

Proof *From Van Loan's theorem 1.2.1 above we have*

$$DFT_n = \begin{bmatrix} I_m & D_m \\ I_m & -D_m \end{bmatrix} \begin{bmatrix} DFT_m & 0 \\ 0 & DFT_m \end{bmatrix} L_n.$$

The corollary follows by applying both sides to x.

5.3.2 Radix-2 twiddle matrix

For $n = 2m$ define the $D_{2,m}$ twiddle matrix as $T \in \mathbb{A}$

$$T_n = \begin{bmatrix} I_m & D_m \\ I_m & -D_m \end{bmatrix},$$

where

$$D_m = diag(I, \omega_n, \ldots, \omega_n^{m-1})$$

and ω_n a primitive n-th root of unity. The synthesis of a DFT_n from a pair of $DFT_{n/2}$ proceeds as follows:

$$DFT_n[x_0, x_1, \ldots, x_n - 1] = I_1 \otimes T_n \begin{bmatrix} DFT_{n/2}[x_0, x_2, \ldots, x_{n-1}] \\ DFT_{n/2}[x_1, x_3, \ldots, x_{n-1}] \end{bmatrix}$$

descending one level in the computation tree we find a pair of butterfly operators

$$DFT_{n/2}[x_0, x_2, \ldots, x_{n-1}] = I_2 \otimes T_{n/2} \begin{bmatrix} DFT_{n/4}[x_0, x_4, \ldots, x_{n-1}] \\ DFT_{n/4}[x_2, x_6, \ldots, x_{n-1}] \end{bmatrix}$$

and

$$DFT_{n/2}[x_1, x_3, \ldots, x_{n-1}] = I_2 \otimes T_{n/2} \begin{bmatrix} DFT_{n/4}[x_1, x_5, \ldots, x_{n-1}] \\ DFT_{n/4}[x_3, x_7, \ldots, x_{n-1}] \end{bmatrix}$$

it follows that

$$DFT_n[x_0, x_1, \ldots, x_{n-1}] = T_n \begin{bmatrix} T_{n/2} & 0 \\ 0 & T_{n/2} \end{bmatrix} \begin{bmatrix} DFT_{n/4}[x_0, x_4, \ldots, x_{n-1}] \\ DFT_{n/4}[x_2, x_6, \ldots, x_{n-1}] \\ DFT_{n/4}[x_1, x_5, \ldots, x_{n-1}] \\ DFT_{n/4}[x_3, x_7, \ldots, x_{n-1}] \end{bmatrix}.$$

Recognizing a pattern, Cooley-Tukey sought an expression of the form

$$DFT_n \vec{x} = A_t \cdots A_1 L_n x,$$

where L_n is some permutation and A_q is a direct sum $\oplus$ of the butterfly operators we know as a twiddle matrix.

$$A_q = diag(T_L, \ldots, T_L) = I_r \otimes T_L$$

where $L = 2^q$ and $r = n/L$.

5.3.3 The Cooley-Tukey factorization

Lemma 5.3.2 *[22] Suppose $n = 2^q$ and $m = n/2$. If $DFT_m P_m = C_{t-1} \cdots C_1$ and*

$$P_n = L_n(I_2 \otimes P_m),$$

then

$$DFT_n P_n = (I_1 \otimes T_n)(I_2 \otimes C_{t-1}) \cdots (I_2 \otimes C_1).$$

Proof *Recall that*

$$DFT_n P_n = T_n(I_2 \otimes DFT_m) \tag{5.2}$$

and so

$$DFT_n P_n = T_n(I_2 \otimes C_{t-1} \cdots C_1 P_m^T).$$

but by the tensor product theorem that $(A \otimes B)(C \otimes D) = (AC \otimes BD)$

$$I_2 \otimes (C_{t-1} \cdots C_1 P_m^T) = (I_2 \otimes C_{t-1}) \cdots (I_2 \otimes C_1)(I_2 \otimes P_m^T)$$

and so

$$DFT_n P_n = T_n (I_2 \otimes C_{t-1}) \cdots (I_2 \otimes C_1)(I_2 \otimes P_m^T).$$

Since

$$(I_2 \otimes P_m^T)^-1 = (I_2 \otimes P_m^T)^T = (I_2 \otimes P_m). \tag{5.3}$$

the lemma follows.

Next, a non-recursive specification of P_n follows.

Lemma 5.3.3 *[22]*
Suppose $n = 2^t$, $P_n = L_n(I_2 \otimes P_m)$ *and* $R_q = I_{2^t-q} \otimes L_2^q$, *then*

$$P_m = R_{t-1} \cdots R_1.$$

Proof *If* $t = 1$, $P_2 = L_2 = I_2$ *the lemma holds. By induction, assume if* $m = 2^{t-1} = n/2$, *then*

$$P_m = \bar{R}_{t-1} \cdots \bar{R}_1.$$

where $\bar{R}_q = I_{2^{t-1-q}} \otimes L_{2q}$. *But from the previous lemma and* $(A \otimes B)(C \otimes D) = (AC \otimes BD)$ *we have*

$$P_n = L_n(I_2 \otimes P_m) = L_n(I_2 \otimes \bar{R}_{t-1} \cdots \bar{R}_1) = L_n(I_2 \otimes \bar{R}_{t-1}) \cdots (I_2 \otimes \bar{R}_1). \tag{5.4}$$

Using the fact that $I_P \otimes (I_q \otimes A) = I_{pq} \otimes A$,

$$I_2 \otimes \bar{R}_q = I_2 \otimes (I_{2^{t-1-q}} \otimes L_{2q}) = I_{2^{t-q}} \otimes L_{2q} = R_q \quad q = 1, \ldots, t-1.$$

Observe $R_t = L_n$ *Completing the proof.*

Theorem 5.3.4 *Cooley-Tukey Radix-2 Factorization Originally, [4], but, this example is borrowed from [22].*
If $n = 2^q$, *then*

$$DFT_n = A_t \cdots A_1 P_n$$

where P_n *is defined as* $R_t \cdots R - 1$ *and* $A_q = I_r \otimes T_n$, $L = 2^q, r = n/L, m = L/2$ T_n *is the* $D_{2,m}$ *radix-2 twiddle matrix, defined in the twiddle factor chapter.*

Proof *if* $n = 2$ *then* $P_n = L_n = I_n$, $DFT_n = T_2$ *and the theorem holds by lemma 4.3.2:*

$$DFT_2 P_2 = DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = I_2 \otimes T_2.$$

For $n = 2^t$ *it can be seen that,*
if we define $A_t = I_1 \otimes T_n$ *and* $A_q = I_2 \otimes C_q$ *for* $q = 1$ *to* $t - 1$ *then lemma 4.3.2 provides the necessary inductive step.*

5.4 Radix-*P* FFTs

We just saw how to factor a FFT into 2 smaller FFTs. We can now discuss how to factor a FFT into p smaller FFTs.

Theorem 5.4.1 *Radix-P splitting [22] If $n = pm$ then*

$$DFT_n L_p^n = DFT_p \otimes I_m diag(I_m, \Omega_{p,m}, \ldots, \Omega_{p,m}^{p-1})(I_p \otimes DFT_m),$$

where $diag(I_m, \Omega_{p,m}, \ldots, \Omega_{p,m}^{p-1})$ is the $D_{p,m}$ twiddle matrix from the twiddle matrix chapter.

Proof *since L_p^N sorts the matrix by columns mod$p$ we have*

$$DFT_n L_p^N = \left[\; DFT_n[x_0, x_p, \ldots, x_{n-1}] \;\middle|\; DFT_n[x_1, x_{1+p}, \ldots, x_{n-1}] \;\middle|\; \cdots \;\middle|\; DFT_n[x_{p-1}, x_{p-1+p}, \ldots, x_{n-1}] \; \right].$$

Now think of $DFT_n L_p^n$ as a $p \times p$ block matrix with $m \times m$ blocks

$$DFT_n L_p^n = (G_{qr}), G \in \mathbb{A}^{>\times>}, o \leq q, r, < p. \tag{5.5}$$

it follows that $G_{qr} = DFT_n X[qm, (q+1)m - 1]$, for r from 0 to $n-1$ with stride p and thus,

$$\left[G_{qr}\right]_{kj} = \omega_n^{(qm+k)(r+jp)} = \omega_p^{qr} \omega_n^{kr} \omega_m^{kj}.$$

using the identities $\omega_n^{mqr} = \omega_p^{qr}, \omega_n^{pkj} = \omega_m^{kj}$, and $\omega_n^{qmjp} = (\omega_n^n)^{qj} = 1$. if by the right-hand side of the radix-p splitting theorem,

$$(DFT_p \otimes I_m) diag(I_m, \Omega_{p,m}, \ldots, \Omega_{p,m}^{p-1})(I_p \otimes DFT_m) = H_{qr}, H \in \mathbb{A}^{m \times m},$$

then $\left[H_{qr}\right]_{qr} = \omega_p^{qr} \Omega_{p,m}^r DFT_m$. Since

$$\left[H_{qr}\right]_{kj} = \omega_p^{qr} \left[\Omega_{p,m}^r\right]_{kk} [DFT_m]_{kj} = \omega_p^{qr} \omega_n^{kr} \omega_m^{kj} = \left[G_{qr}\right]_{kj}$$

holds for all k and j, the theorem follows.

Now, if we set $p = 2$, in the preceding theorem, we get

$$DFT_n L_2^n = (DFT_2 \otimes I_{n/2}) diag(I_{n/2}, D_{n/2})(I_2 \otimes DFT_{n/2}).$$

where

$$(DFT_2 \otimes I_{n/2}) diag(I_{n/2}, D_{n/2}) = \begin{bmatrix} I_{n/2} & D_{n/2} \\ I_{n/2} & -D_{n/2} \end{bmatrix}.$$

is the radix-2 twiddle matrix used in radix-2 splitting. So we now have a way to relate a DFT_n to $DFT_{n/p}$.

5.5 Cooley-Tukey Four-Step FFT

If we look at the preceding theorem and arrange it into four parts we get

$$DFT_n = (DFT_k \otimes I_m)D_{k,m}(I_k \otimes DFT_m)L^n_k \tag{5.6}$$

what is known as the Four-Step Cooley-Tukey general-radix decimation-in-time fast Fourier transform. It is built from a pair of DFTs, the twiddle matrix and the stride permutation. We can manipulate this equation from [11] with

$$A \otimes B \quad = L^{mn}_n(B \otimes A)L^{mn}_m. \tag{5.7}$$

We apply the manipulation (5.11) to the equation (5.10) above, substituting $(DFT_k \otimes I_m)$ for $(A \otimes B)$. This gives us

$$\begin{aligned} DFT_n &= (DFT_k \otimes I_m)D_{k,m}(I_k \otimes DFT_m)L^n_k \\ &= L^{mk}_k(I_m \otimes DFT_k)L^{mk}_m D_{k,m}(I_k \otimes DFT_m)L^{mk}_k \end{aligned}$$

which results in the Cooley-Tukey general-radix decimation-in-time Six-Step fast Fourier transform.

5.6 Cooley-Tukey Six-Step FFT

The Cooley-Tukey Six-Step fast Fourier transform is given by

$$DFT_n = L^n_k(I_m \otimes DFT_k)L^n_m D^n_{k,m}(I_k \otimes DFT_m)L^n_k, \quad n = km. \tag{5.8}$$

It is built from two stages of parallel DFTs, a twiddle diagonal, and three global transpositions.

We use the definitions 4.10 and 4.12 to create generic FFT functions that we can use with a range of DFT_k base case kernels that work over a variety of finite fields. The algorithm performs a right side expansion of the Cooley-Tukey six-step FFT. We do this by first creating loop-unrolled base case DFTs that compute DFT_k. Then we expand the right side DFT_ms until the DFT_k base case is reached and computed directly. The base case kernel k is selected following our overall blocking strategy.

Chapter 6

The Generic Fast Fourier Transform

Our overall strategy is to break down a large FFT into small FFTs that can be computed in an cache-oblivious manner. To this end, all the elements of each base case FFT input vector as well as the twiddle vector must fit in the cache to avoid expensive memory accesses. Our FFT will work over a variety of finite fields with different size characteristics so we want a variety of sizes to help meet this restriction. We have chosen to implement base case DFTs of size 2, 4, 8, 16, 32, and 64 for both Six-Step and Four-Step variants.

6.1 FFT base cases

For our base case FFTs we use radix-2 splitting and reduce each base case to a base case of size 2 which is then computed directly.
Given

$$DFT_N = L^{km}_k(I_m \otimes DFT_k)L^{km}_m D_{k,m}(I_k \otimes DFT_m)L^{km}_k$$

we reduce DFT_N where $N = km$ by recursively applying their definitions. We will now create our base case equations for $N = 4, 8, 16, 32, 64$.

6.1.1 DFT_4

We know

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

So to begin, we set

$$DFT_4 = L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2$$

6.1.2 DFT_8

Next, we set

$$DFT_8 = L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes DFT_4)L^8_2.$$

We have DFT_4

$$DFT_4 = L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting it all together we have

$$DFT_8 = L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes (L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2. \tag{6.1}$$

6.1.3 DFT_{16}

Recall that

$$DFT_N = L^N_K(I_J \otimes DFT_K)L^N_J D_{K,J}(I_K \otimes DFT_J)L^N_K.$$

Set

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes DFT_8)L^{16}_2.$$

Where

$$DFT_8 = L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes DFT_4)L^8_2,$$
$$DFT_4 = L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2,$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting that together we have DFT_{16} unrolled as

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes (L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes (L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2))L^{16}_2. \tag{6.2}$$

6.1.4 DFT_{32}

Recall that

$$DFT_N = L^N_K(I_J \otimes DFT_K)L^N_J D_{K,J}(I_K \otimes DFT_J)L^N_K.$$

Set

$$DFT_{32} = L^{32}_2(I_{16} \otimes DFT_2)L^{32}_{16} D_{2,16}(I_2 \otimes DFT_{16})L^{32}_2.$$

Where

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes DFT_8)L^{16}_2,$$
$$DFT_8 = L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes DFT_4)L^8_2,$$
$$DFT_4 = L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2,$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting the pieces together we have DFT_{32} unrolled as

$$DFT_{32} = L^{32}_2(I_{16} \otimes DFT_2)L^{32}_{16} D_{2,16}(I_2 \otimes (L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes (L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes (L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2))L^{16}_2))L^{32}_2. \tag{6.3}$$

6.1.5 DFT_{64}

Recall that

$$DFT_N = L^N_K(I_J \otimes DFT_K)L^N_J D_{K,J}(I_K \otimes DFT_J)L^N_K.$$

Set

$$DFT_{64} = L^{64}_2(I_{32} \otimes DFT_2)L^{64}_{32} D_{2,32}(I_2 \otimes DFT_{32})L^{64}_2.$$

Where

$$DFT_{32} = L^{32}_2(I_{16} \otimes DFT_2)L^{32}_{16} D_{2,16}(I_2 \otimes DFT_{16})L^{32}_2,$$

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes DFT_8)L^{16}_2,$$

$$DFT_8 = L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes DFT_4)L^8_2,$$

$$DFT_4 = L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2,$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting all the pieces together we have the Six-Step DFT_{64} unrolled as

$$L^{64}_2(I_{32}\otimes DFT_2)L^{64}_{32}D_{2,32}(I_2\otimes(L^{32}_2(I_{16}\otimes DFT_2)L^{32}_{16}D_{2,16}(I_2\otimes(L^{16}_2(I_8\otimes DFT_2)L^{16}_8D_{2,8}(I_2\otimes(L^8_2(I_4\otimes DFT_2)L^8_4D_{2,4}(I_2\otimes(L^4_2(I_2\otimes DFT_2)L^4_2D_{2,2}(I_2\otimes DFT_2)L^4_2))L^8_2))L^{16}_2))L^{32}_2))L^{64}_2. \tag{6.4}$$

6.1.6 Four-Step FFT base cases

For our base case FFTs we use radix-2 splitting and reduce each base case to a base case of size 2 which is then computed directly.
Given

$$DFT_N = (DFT_k \otimes I_m)D_{k,m}(I_k \otimes DFT_m)L^{km}_k$$

we reduce DFT_N where $N = km$ by recursively applying their definitions. We will now create our base case equations for $N = 4, 8, 16, 32, 64$.

6.1.7 Four-Step DFT_4

We know

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

So to begin, we set

$$DFT_4 = (DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L^4_2.$$

6.1.8 Four-Step DFT_8

Next, we set

$$DFT_8 = (DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes DFT_4)L_2^8$$

where

$$DFT_4 = (DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^4.$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting it all together we have

$$DFT_8 = (DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes ((DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^4))L_2^8.$$

6.1.9 Four-Step DFT_{16}

Next, we set

$$DFT_{16} = (DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes DFT_8)L_2^{16}$$

where

$$DFT_8 = (DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes DFT_4)L_2^8,$$
$$DFT_4 = (DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^4,$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting it all together we have the Four-Step DFT_{16} unrolled as

$$DFT_{16} = (DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes ((DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes ((DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^4))L_2^8))L_2^{16}$$

6.1.10 Four-Step DFT_{32}

Repeating the process for DFT_{32}, Set

$$DFT_{32} = (DFT_2 \otimes I_{16})D_{2,16}(I_2 \otimes DFT_{16})L_2^{32}$$

where

$$DFT_{16} = (DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes DFT_8)L_2^{16},$$
$$DFT_8 = (DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes DFT_4)L_2^8,$$
$$DFT_4 = (DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^4,$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting it all together we have the Four-Step DFT_{32} unrolled as

$$DFT_{32} = (DFT_2 \otimes I_{16})D_{2,16}(I_2 \otimes ((DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes ((DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes ((DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^4))L_2^8))L_2^{1}$$

6.1.11 Four-Step DFT_{64}

Repeating the process for DFT_{64}, Set

$$DFT_{64} = (DFT_2 \otimes I_{32})D_{2,32}(I_2 \otimes DFT_{32})L_2^{64}$$

where

$$DFT_{32} = (DFT_2 \otimes I_{16})D_{2,16}(I_2 \otimes DFT_{16})L_2^{32},$$
$$DFT_{16} = (DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes DFT_8)L_2^{16},$$
$$DFT_8 = (DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes DFT_4)L_2^{8},$$
$$DFT_4 = (DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L_2^{4},$$

and

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Putting it all together we have the Four-Step DFT_{64} unrolled as

$$DFT_{64} = (DFT_2 \otimes I_{32})D_{2,32}(I_2 \otimes ((DFT_2 \otimes I_{16})D_{2,16}(I_2 \otimes ((DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes ((DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes ((DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes$$

6.2 Loop unrolling a FFT base case

We will begin our first example by taking the Six-Step DFT_4 and implementing it in the C language.
To maximize efficiency, we first pre-compute and pass in to our function all of the different ω_n twiddle factors required during the base case computation. Next, we minimize data movement by keeping track of the indices during the stride permutations of the base case FFTs and then apply operations to the appropriate elements rather than actually moving around the data. Finally, we eliminate any unnecessary or "dead" code.
We know DFT_2 as

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

We implement the function DFT2() as

```
void fft2(ELEMENTS* x,ELEMENTS* y,struct Prime_ptr* P){
   ELEMENTS tmp = 0;
   tmp = ADD(x, y, P);
   *y = SUB(x, y, P);
   *x = tmp;
}
```

We also need a way to swap two elements of a vector so we use the swap() function listed below.

```
void swap(ELEMENTS* x, ELEMENTS* y){
    ELEMENTS tmp = 0;
    tmp = *y;
    *y = *x;
    *x = tmp;
```

```
}

fast_bpas_swap(long int* a,long int* b){
__asm__ volatile (
      "xchg \%0,\%1\n\n":
      "+g"(*a),"+g"(*b):
      :);
}
```

We start with an input vector of size 4

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix},$$

and we begin the loop-unroll by applying the first stride permutation.

$$L^4_2 A = \begin{bmatrix} a_0 & a_2 & a_1 & a_3 \end{bmatrix} \tag{6.5}$$

Next, we perform our two base case DFT_2s on the two tuples

$$DFT_2(a_0, a_2) \quad DFT_2(a_1, a_3). \tag{6.6}$$

Now we need the $D_{2,2}$ twiddle matrix.
So we have

$$D_{2,2} = \begin{bmatrix} 1 & 1 \\ 1 & \omega^2 \end{bmatrix}. \tag{6.7}$$

We do not compute any unnecessary multiplications and apply our twiddle factors to only the affected elements.

Ignore	Compute
$a_0 = a_0 \times 1$	
$a_2 = a_2 \times 1$	
$a_1 = a_1 \times 1$	
	$a_3 = a_3 \times omega^2$

After the twiddle we perform an L^4_2 stride permutation

$$L^4_2 A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix}$$

and then the second round of DFT_2s on the given tuples

$$DFT_2(a_0, a_1) \quad DFT_2(a_2, a_3) \tag{6.8}$$

and finish with the last L^4_2 stride permutation

$$\begin{bmatrix} a_0 & a_2 & a_1 & a_3. \end{bmatrix}. \tag{6.9}$$

Putting this into C code we have

```
void fft4(ELEMENTS* x, ELEMENTS* omegas, struct Prime_ptr* P){
  //L_2^4 (I_2 otimes fft2) L_2^4 T_2^4 (I_2 otimes fft2)L_2^4
  // 0 1 2 3
  //L_2^4 <--x
```

```
  //0 2 1 3
  //I_2 otimes DFT_2
  fft2(&x[0],&x[2],P);
  fft2(&x[1],&x[3],P);
  //T_2^4
  //{w^0 w^1}^0 oplus {w^0 w^1}^1
  //1 1 oplus 1 w_4
  x[3] = MULTIPLY(&x[3],&omegas[0],P);
  //L_2^4
  //0 1 2 3
  //I_2 otimes DFT_2
  fft2(&x[0],&x[1],P);
  fft2(&x[2],&x[3],P);
  //L_2^4
  //0,2,1,3 <-- need to actually swap 2 and 1
  swap(&x[1],&x[2]);
}
```

.

We can do the same with the Four-Step DFT_4. The same code results.

```
void four_step_fft_4(ELEMENTS* x,ELEMENTS* omegas,struct Prime_ptr* P){
  //L_2^4
  //0,2,1,3
  //I_2 \otimes DFT_2
  fft2(x[0],x[2]);
  fft2(x[1],x[3]);
  //D_{2,2}
  x[3] = MULTIPLY(x[3],omegas[0],P);
  //(DFT_2 \otimes I_2)
  fft2(x[0],x[1]);
  fft2(x[2],x[3]);
  // perform minimum data movement
  swap(x[2],x[1]);
  // same as six-step
}
```

.

Now we unroll the Six-Step DFT_8. We start with an input vector of size 8

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \end{bmatrix},$$

and we begin by applying the first stride permutation.

$$L^8_2 A = \begin{bmatrix} a_0 & a_2 & a_4 & a_6 & a_1 & a_3 & a_5 & a_7 \end{bmatrix} \tag{6.10}$$

next, we apply the first stride permutation from the first term of the substituted DFT_4

$$L^4_2 A = \begin{bmatrix} a_0 & a_4 & a_2 & a_6 & a_1 & a_5 & a_3 & a_7 \end{bmatrix} \tag{6.11}$$

we then apply the DFT_2s on the twice stride permuted matrix A

$$I_2 \otimes (...(I_2 \otimes DFT_2))A. \tag{6.12}$$

which results in DFT_2s on the elements given in the following DFT_2 tuples

$$DFT_2(a_0, a_4) \quad DFT_2(a_2, a_6) \quad DFT_2(a_1, a_5) \quad DFT_2(a_3, a_7). \tag{6.13}$$

Now we need the $D_{2,2}$ twiddle matrix.
So we have

$$D_{2,2} = \begin{bmatrix} 1 & 1 \\ 1 & \omega^2 \end{bmatrix}. \tag{6.14}$$

We do not compute any unnecessary multiplications and apply our twiddle factors to only the affected elements.

Ignore	Compute
$a_0 = a_0 \times 1$	
$a_4 = a_4 \times 1$	
$a_2 = a_2 \times 1$	
	$a_6 = a_6 \times omega^2$.
$a_1 = a_1 \times 1$	
$a_5 = a_5 \times 1$	
$a_3 = a_3 \times 1$	
	$a_7 = a_7 \times omega^2$

We follow with another L^4_2 stride permutation

$$L^4_2 A = \begin{bmatrix} a_0 & a_2 & a_4 & a_6 & a_1 & a_3 & a_5 & a_7 \end{bmatrix}$$

and follow with another round of DFT_2s on the given tuples

$$DFT_2(a_0, a_2) \quad DFT_2(a_4, a_6) \quad DFT_2(a_1, a_3) \quad DFT_2(a_5, a_7) \tag{6.15}$$

and another L^4_2 stride permutation

$$\begin{bmatrix} a_0 & a_4 & a_2 & a_6 & a_1 & a_5 & a_3 & a_7 \end{bmatrix} \tag{6.16}$$

followed by a multiplication by the $D_{2,4}$ twiddle factor matrix

$$D_{2,4} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \omega \\ \omega^2 \\ \omega^3 \end{bmatrix}. \tag{6.17}$$

again we apply our twiddle factors to only the affected elements.

Ignore	Compute
$a_0 = a_0 \times 1$	
$a_4 = a_4 \times 1$	
$a_2 = a_2 \times 1$	
$a_6 = a_6 \times 1$	
$a_1 = a_1 \times 1$	
	$a_5 = a_5 \times omega$
	$a_3 = a_3 \times omega^2$
	$a_7 = a_7 \times omega^3$

Next comes an L^8_4 stride permutation

$$\left[\begin{array}{cccccccc} a_0 & a_1 & a_4 & a_5 & a_2 & a_3 & a_6 & a_7 \end{array} \right]. \tag{6.18}$$

Followed by $I_4 \otimes DFT_2$s on the following tuples

$$DFT_2(a_0, a_1) \quad DFT_2(a_4, a_5) \quad DFT_2(a_2, a_3) \quad DFT_2(a_6, a_7) \tag{6.19}$$

Ending with an L^8_2 stride permutation

$$\left[\begin{array}{cccccccc} a_0 & a_4 & a_2 & a_6 & a_1 & a_5 & a_3 & a_7 \end{array} \right]. \tag{6.20}$$

Putting this into C code we have

```
void DFT8(ELEMENTS *a,ELEMENTS * omega){

  //DFT_2s on permuted indices
  DFT2(a[0], a[4]);
  DFT2(a[2], a[6]);
  DFT2(a[1], a[5]);
  DFT2(a[3], a[7]);

  //D_(2,2) twiddle factor
  a[6] = a[6] * omega[2];
  a[7] = a[7] * omega[2];

  //DFT_2s on permuted indices
  DFT2(a[0], a[2]);
  DFT2(a[4], a[6]);
  DFT2(a[1], a[3]);
  DFT2(a[5], a[7]);

  // D_(2,4) twiddle factor
  a[5] = a[5] * omega[1];
  a[3] = a[3] * omega[2];
  a[7] = a[7] * omega[3];

  //DFT_2s on permuted indices
  DFT2(a[0], a[1]);
```

```
26    DFT2(a[4], a[5]);
27    DFT2(a[2], a[3]);
28    DFT2(a[6], a[7]);
29
30    // final permutation
31    swap(a[1],a[4]);
32    swap(a[3],a[6]);
33  }
```

Where *a* points to the input vector and *omega* points to a precomputed vector containing the powers of ω.

Figure 6.1: DFT_{16} right-expanded using radix-2 splitting

$$L_2^{16}(I_8 \otimes DFT_2)L_8^{16}D_{2,8}(I_2 \otimes (L_2^8(I_4 \otimes DFT_2)L_4^8D_{2,4}(I_2 \otimes (L_2^4(I_2 \otimes DFT_2)L_2^4D_{2,2}(I_2 \otimes (DFT_2)L_2^4)L_2^8)L_2^8)L_2^{16} \tag{6.21}$$

We repeat the above process for DFT_{16}, DFT_{32}, and DFT_{64}. We use our stride permutation function to automate index calculations. The code for the both the six-step and four-step base case FFTs when loop-unrolled and optimized is identical.

Next, we look at the operation count in our loop-unrolled DFT_8. We count 24 modular additions from the twelve DFT_2 butterflies, and only 5 modular multiplications by twiddle factors. We see a reduction in the number of modular multiplications in the loop-unrolled versions due to the removal of unnecessary multiplications by twiddle factors where $\omega^{kj} == 1$. The reduction in modular multiplications increases as the size of the DFT_k loop-unrolled base case increases. At some point the working set no longer fits in cache, causing cache capacity misses resulting in heavy performance penalties. In terms of cache misses, the base case input vector and the twiddle vector fit entirely in cache. The first access results in a miss but all subsequent accesses are hits.

6.3 General FFT function

While loop-unrolling the different sized base cases, a recognizable pattern emerges.
First we need to perform a series of stride permutations. L_k^x for $x = k^n, \ldots, k^2$.
This series of stride permutations is followed by a round of base case DFTs. Next, we perform a repeating group of operations. The repeating group represents a level of recursion and, in the case of the Six-Step FFT, consists of a twiddle, followed by a stride permutation, followed by base case DFTs, finishing with a stride permutation. Below is DFT_{16} colour-coded to help identify the components. The series of stride permutations (in red), the first set of base case DFTs (in black) followed by the repeating group of a twiddle, a shuffle stride permutation, base case DFTs, followed by a sort stride permutation.

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes (L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes (L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2))L^{16}_2.$$

We see that the Four-Step FFT follows the same pattern. The Four-Step starts with the same series of stride permutations followed by the first round of base case DFTs and the it repeats a $D_{2,m}$ twiddle operation and a $(DFT_2 \otimes I_m)$ operation for each level of recursion.

$$DFT_{16} = (DFT_2 \otimes I_8)D_{2,8}(I_2 \otimes ((DFT_2 \otimes I_4)D_{2,4}(I_2 \otimes ((DFT_2 \otimes I_2)D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2))L^{16}_2$$

The pattern described above is used to derive our general FFT function. It falls out of the right-side recursive expansion of a Cooley-Tukey decimation-in-time FFT. In both cases, we begin by recursing down the computation tree performing the right-side "even/odd sort" stride permutations and making recursive calls on the sorted halves for each level of the descent from k^n to k^2. The recursion terminates after performing the $n = 2 \cdot k$ iteration containing the L^4_2 stride permutation because the general FFT function calls the $DFT_{basecase}$ kernel for $n = k^1$ and the $DFT_{basecase}$ kernel performs its own permutations.

The next step in the right-side expansion is to perform the right-side base case DFT kernels (coloured in black). Since $N = km$, there will be m of the right-side base case DFT_k kernel calls.

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes (L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes (L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2))L^{16}_2.$$

We perform these DFT_k kernels in parallel and in cache-sized blocks. After the right-side base case DFT_k kernels, each leaf node computes its repeating group of operations (coloured first in green, second in teal, and blue) for that level of recursion in the tree. For each level of the recursion tree, at each leaf node, we perform a twiddle, followed by inner "shuffle" stride permutations. Then, the left side base case FFT kernels are performed followed by the outer "sort" stride permutations. When a leaf node finishes the last of its group operations, the result is returned to the parent. When we get back to the top of the recursion tree, the parent finishes its group of operations and the computation is complete.

If $n = 2^q$ and $n == b$ and the ω_n diagonal $D_{k,j}$ twiddle matrices are available then, the general function for the Six-Step FFT computes DFT_n by calling the base case DFT_b kernel–the DFT_b kernel computes the DFT_b directly using the radix-2 loop-unrolled base cases detailed above. Otherwise, if $n >= b \cdot m$ then the general FFT function right expands the DFT_n, $n = bm$ into m DFT_bs and b DFT_ms.

Procedure 9 General function for the Six-Step FFT in pseudo code

Input: X input vector of size N
Input: n $log_2(N)$
Input: w primitive n^{th} root of unity
Input: b size of basecase
Input: DN** pointers to arrays of precomputed powers of w_n for the twiddle matrix at each level of recursion
Input: DB* pointer to precomputed powers of omega for the chosen basecase size
Output: DFT_nX

```
for (i = 0; i < n − log2(b); i + +) do
    for (j = 0; j < pow(2, i); j + +) do
        L_2^{pow(2,n−i)}(X[j ∗ pow(2, n − i)])
    end for
end for
for (i = 0; i < N; i+ = b) do
    DFT_b(X[i], DB)
end for
for (i = n − log2(b) − 1; i >= 0; i − −) do
    m = pow(2, n − i);
    for (j = 0; j < pow(2, i); j + +) do
        index = j ∗ m
        D_{2,m/2}(X[index], DN[i])
        L^m_{m/2}(X[index])
    end for
    for (j = 0; j < pow(2, i); j + +) do
        index = j ∗ m
        for (z = 0; z < m/2; z + +) do
            idx = index + z ∗ 2
            DFT_2(x[idx])
        end for
    end for
    for (j = 0; j < pow(2, i); j + +) do
        index = j ∗ m
        L^m_2(X[index])
    end for
end for
```

Similarly, If $n = 2^q$ and $n == b$ and the ω_n diagonal $D_{k,j}$ twiddle matrices are available then, the general function for the Six-Step Merged FFT computes DFT_n by calling the base case DFT_b kernel–the DFT_b kernel computes the DFT_b directly using the radix-2 loop-unrolled base cases detailed above. Otherwise, if $n >= b \cdot m$ then the general Six-Step Merged FFT function right expands the DFT_n, $n = bm$ into m DFT_bs and b DFT_ms.

Procedure 10 General function for the Loop Merged Six-Step FFT in pseudo code

Input: X input vector of size N
Input: n $log_2(N)$
Input: w primitive n^{th} root of unity
Input: b size of basecase
Input: DN** pointers to arrays of precomputed powers of w_n for the twiddle matrix at each level of recursion
Input: DB* pointer to precomputed powers of omega for the chosen basecase size
Input: Y* scratch array of size N
Output: DFT_nX

```
for (i = 0; i < n − log2(b); i + +) do
  for (j = 0; j < pow(2, i); j + +) do
    L_2^{pow(2,n−i)}(X[j ∗ pow(2, n − i)])
  end for
end for
for (i = 0; i < N; i+ = b) do
  DFT_b(X[i], DB)
end for
for (i = n − log2(b) − 1; i >= 0; i − −) do
  m = pow(2, n − i);
  for (j = 0; j < pow(2, i); j + +) do
    idx = j ∗ m
    for (z = 0; z < m/2); z + + do
      tmp = X[(idx + z) + m/2] ∗ DN[i][z + m/2]
      Y[idx + z] = X[idx + z] + tmp
      Y[idx + z + m/2] = X[idx + z] − tmp
    end for
    for (z = 0; z < m; z + +) do
      X[idx + z] = Y[idx + z]
    end for
  end for
end for
```

Similarly, If $n = 2^q$ and $n == b$ and the ω_n diagonal $D_{k,j}$ twiddle matrices are available then, the general function for the Four-Step FFT computes DFT_n by calling the base case DFT_b kernel–the DFT_b kernel computes the DFT_b directly using the radix-2 loop-unrolled base cases detailed above. Otherwise, if $n >= b \cdot m$ then the general Four-Step FFT function right expands the DFT_n, $n = bm$ into m DFT_bs and b DFT_ms.

Procedure 11 General function for the Four-Step FFT in pseudo code

Input: X input vector of size N
Input: n $log_2(N)$
Input: b size of basecase
Input: DN** pointers to arrays of precomputed powers of w_n for the twiddle matrix at each level of recursion
Input: DB* pointer to precomputed powers of omega for the chosen basecase size
Output: DFT_nX

```
for (i = 0; i < n − log2(b); i + +) do
  for (j = 0; j < pow(2, i); j + +) do
    L_2^{pow(2,n−i)}(X[j ∗ pow(2, n − i)])
  end for
end for
for (i = 0; i < N; i+ = b) do
  DFT_b(X[i], DB)
end for
for (i = n − log2(b) − 1; i >= 0; i − −) do
  current_n = 1 << (log2(N) − i);
  m = current_n >> 1;
  for (int j = 0; j < (1 << i); j + +) do
    index = j ∗ current_n;
    for (z = 0; i < m; i + +) do
      a = X[index + z] ∗ DN[i][index + z]
      b = X[index + z + m] ∗ DN[i][index + z + m]
      X[index + z] = a + b;
      X[index + z + m] = a − b;
    end for
  end for
end for
```

6.4 General FFT function optimization

We look at ways to optimize the general FFT algorithm. First, we pre-compute our twiddle matrices and pass in respective pointers. Next, we look at reducing the number of times our algorithms pass through the data set during the computation using loop-merging and by treating the initial permutations as data access into the smaller block-parallel DFTs by pre-shuffling the

input array. Then we look at reducing our operation count using dead code elimination. Finally, we parallelize our code using Cilk.

6.4.1 Pre-shuffling the input vector

The pre-shuffle function is a recursive function that follows the same factorization path as the FFT computation. The same radix is used. The function propagates the permutations as data access into the smaller DFTs. We allow ourselves a degree of freedom as to whether or not these initial permutations (coloured in red) are performed ”online” or ”offline”. We have implemented versions with both but since we are already pre-computing our twiddle factors, we will pre-shuffle the input vector.

$$DFT_{16} = L^{16}_2(I_8 \otimes DFT_2)L^{16}_8 D_{2,8}(I_2 \otimes (L^8_2(I_4 \otimes DFT_2)L^8_4 D_{2,4}(I_2 \otimes (L^4_2(I_2 \otimes DFT_2)L^4_2 D_{2,2}(I_2 \otimes DFT_2)L^4_2))L^8_2))L^{16}_2.$$

The input to the pre-shuffle function is a pointer to the vector to be pre-shuffled, the size of the vector, and the size of base case to halt the recursion. The function performs the ”even-odd sort” stride permutation (coloured in red) and then makes a recursive call on each of the permuted halves. The recursion terminates when the size of the input vector is the same size as the base case. The Four-Step and Six-Step FFT equations both start with the L^{km}_k stride permutation and both FFTs make the same recursive calls on each of the permuted halves so we create the recursive pre-shuffle function for both FFTs.

Figure 6.2: Recursive pre-shuffle

```
1  void rec_preshuffle_vector(ELEMENTS* X, int n, int k){
2    if (n==k){
3      return;  // do nothing and return
4    }else{
5      int m = (n>>1);
6      stride_permutation(&X[0],2,m);
7      spawn rec_preshuffle_vector(&X[0],m,k); // cilk_spawn this call
8      rec_preshuffle_vector(&X[m],m,k); // then cilk_sync after this call
9      sync;
10   }
11 }
```

An iterative version follows.

Figure 6.3: Iterative pre-shuffle

```
void preshuffle_vector(ELEMENTS* vector, int N, int basecase){
  int basecase_log_2 = (int)(log(basecase)/log(2));
  int n_log_2 = (int)(log(N)/log(2));
  for (int i=0;i<n_log_2-basecase_log_2;i++){ // for each level of
   recursion
   #pragma cilk_grainsize = GRAINSIZE_1
    p_for (int j=0;j<pow(2,i);j++){ // cilk_for each dft node at this level
      stride_permutation(&vector[j*(pow(2,n_log_2-i))],2,pow(2,(n_log_2-i)
   -1));
      %// stride_permutation(&vector[j*(1<<(n_log_2-i))],2,1<<((n_log_2-i)
   -1));
    }
  }
}
```

We can see that by pre-shuffling the input vector we eliminate a pass through the data set at each level of recursion during the computation.

6.4.2 Loop merging

We use loop merging to minimize the number of times the algorithm passes through the data set. If we look at the Six-Step FFT we see that for each recursion step, we pass through the data six times. Now we minimize the number of times we pass through the data after our pre-shuffle and first block of base case DFTs.

6.4.3 Loop merging example

We start with a simple example, this equation contains a L_8^{16} stride permutation and a block of DFT_2s.

$$(I_8 \otimes DFT_2)L_8^{16}$$

What happens when we apply this equation to input $\vec{x}$ is a loop to perform the shuffle stride permutation followed by a loop to execute a round of DFT_2 that happen in order on the elements of $\vec{x}$. To merge these loops we can perform a single loop over the elements of x and execute the round of DFT_2s at the same stride as the L_8^{16} stride permutation.

$$x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_9 \quad x_{10} \quad x_{11} \quad x_{12} \quad x_{13} \quad x_{14} \quad x_{15}$$

$\vec{x}$ after L_8^{16} becomes

$$x_0 \quad x_8 \quad x_1 \quad x_9 \quad x_2 \quad x_{10} \quad x_3 \quad x_{11} \quad x_4 \quad x_{12} \quad x_5 \quad x_{13} \quad x_6 \quad x_{14} \quad x_7 \quad x_{15}$$

and the DFT_2s get performed on the following tuples

$$DFT_2(x_0, x_8) \quad DFT_2(x_1, x_9) \quad DFT_2(x_2, x_{10}) \quad DFT_2(x_3, x_{11}) \quad DFT_2(x_4, x_{12}) \quad DFT_2(x_5, x_{13}) \quad DFT_2(x_6, x_{14})$$

Now we combine the two in a for loop

```
//x[16];
//y[16];
for (i=0 i<8;i++){
  y[i]= x[i] + x[i+8];
  y[i+1]=x[i]-x[i+8];
}
```

and we see that the result is the same. DFT_2s are performed on the following tuples.

Loop iteration	$DFT_2 Tuple$
0	$DFT_2(x_0, x_8)$
1	$DFT_2(x_1, x_9)$
2	$DFT_2(x_2, x_{10})$
3	$DFT_2(x_3, x_{11})$
4	$DFT_2(x_4, x_{12})$
5	$DFT_2(x_5, x_{13})$
6	$DFT_2(x_6, x_{14})$
7	$DFT_2(x_7, x_{15})$

Now, lets add a $D_{2,8}$ twiddle matrix to our example.

$$(I_8 \otimes DFT_2)L_8^{16}D_{2,8}$$

The action of a $D_{2,8}$ twiddle matrix on a vector $\vec{x}$ is a point-wise multiplication.

$$D_{2,8} = [\ 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \omega_n \quad \omega_n^2 \quad \omega_n^3 \quad \omega_n^4 \quad \omega_n^5 \quad \omega_n^6 \quad \omega_n^7\ ]$$

$$D_{2,8} = D[16] = [\ d_0 \quad d_1 \quad d_2 \quad d_3 \quad d_4 \quad d_5 \quad d_6 \quad d_7 \quad d_8 \quad d_9 \quad d_{10} \quad d_{11} \quad d_{12} \quad d_{13} \quad d_{14} \quad d_{15}\ ]$$

and

$$\vec{x} = [\ x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_9 \quad x_{10} \quad x_{11} \quad x_{12} \quad x_{13} \quad x_{14} \quad x_{15}\ ]$$

Our twiddle function simply loops over the elements and multiplies $x[i]$ with $D[i]$. So, the way to merge these loops is to simply perform the $D_{2,8}$ twiddle while we have our vector elements in cache. The $D_{2,8}$ twiddle matrix contains 16 powers of ω_n. We simply point-wise multiply the elements with their twiddle counterparts right before we perform the DFT_2. Integrating this change into our previous loop results in the following code

```
x[16];
y[16];
a;
b;
for (i=0 i<8;i++){
  a = x[i] * D[i]; // point-wise multiplication
  b = x[i+8] * D[i+8]; //
  y[i]= a + b;
  y[i+1]= a - b;
}
```

which produces exactly the same result as if we multiplied the elements using a loop first. To finish our loop merging example we add one last L_2^8 stride permutation to our equation

$$L_2^8(I_8 \otimes DFT_2)L_8^{16}D_{2,8}.$$

Without loop merging, after the DFT_2s, we simply loop over the elements of x and perform an even/odd sort permutation. To merge this loop we need to modify our loop output so that it writes to our result array the same way it would be after we perform the newly added L_2^8 stride permutation. We see that the un-merged code writes its output in order and then the sort takes and arranges the even elements first and odd elements second. This means that the even elements get stored in the first 8 indexes and the odd indexed elements get moved to the second half of the vector and are now at a stride of 8 from their original locations.
First, original x_0 stays in the same spot but original x_1 moves to x_8,
next, original x_2 moves to x_1 and original x_3 moves to x_9,
next, original x_4 moves to x_2 and original x_5 moves to x_{10},
next, original x_6 moves to x_3 and original x_7 moves to x_{11},
next, original x_8 moves to x_4 and original x_9 moves to x_{12},
next, original x_{10} moves to x_5 and original x_{11} moves to x_{13},
next, original x_{12} moves to x_6 and original x_{13} moves to x_{14},
next, original x_{14} moves to x_7 and original x_{15} stays at x_{15},
We can clearly see that the first elements of the previous DFT_2 tuples get written in order they were processed to the first half of the output array, while the second elements of the DFT_2 tuples get written in the order they were processed to the second half of the output array. To achieve this we simply write our output to match our pattern above. The first DFT_2 tuple element is at output $y[i]$ while the second output is written to $y[i+8]$. Integrating the changes into our code example

```
x[16];
y[16];
a;
b;
for (i=0 i<8;i++){
  a = x[i] * D[i]; // point-wise multiplication
  b = x[i+8] * D[i+8]; //
  y[i]= a + b;
  y[i+8]= a - b;
}
```

again produces the exact same result as the un-merged code. We worked through this with a specific value for $m = 8$ but this generalizes when matrices are of appropriate sizes for n=2m. To make this generalize we can simply substitute a general m for the value 8 in our examples which produces the following code.

$$L_2^n(I_m \otimes DFT_2)L_m^n D_{2,m}, \text{ where } n = 2m.$$

```
x[n];
y[n];
a;
b;
```

```
for (i=0 i<m;i++){
  a = x[i] * D[i]; // point-wise twiddle multiplication
  b = x[i+m] * D[i+m]; //
  y[i]= a + b;
  y[i+m]= a - b;
}
```

6.4.4 Loop merged Six-Step FFT

The example we worked through produced roughly the same code as we use in our loop merged version of the Six-Step FFT. In figure 6.4 there are three main for loops to perform the twiddles,shuffle permutations,l.h.s.DFT_2s and sort permutations, while in figure 6.5 these three main loops are merged into one loop. Merging these for loops reduced the number of times we pass through the working data set.

Figure 6.4: Six-Step FFT before loop merge code listing

```
void SIX_STEP_FFT_K_2_N(ELEMENTS* vector,int N,int n_log_2,ELEMENTS**
    twiddle,struct Prime_ptr* P){
  for (int i=0;i<(1<<(n_log_2-1));i++){
    fft2(&vector[i*2],&vector[i*2+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){
    int m = 1<<(n_log_2-i);
    for (int j=0;j<(1<<i);j++){
      int index = j*m;
      twiddle_pc(&vector[index],m/2,2,twiddle[i],P);
      stride_permutation(&vector[index],m/2,2);
    }
    for (int j=0;j<(1<<i);j++){
      int index = j*m;
      for (int z=0;z<m/2;z++){
         int idx = index+z*2;
         fft2(&vector[idx],&vector[idx+1],P);
      }
    }
    for (int j=0;j<(1<<i);j++){
         stride_permutation(&vector[j*m],2,m/2);
    }
  }//end recursion step
}//end SIX_STEP_FFT_K_2_N
```

becomes

Figure 6.5: Six-Step FFT after loop merge code listing

```
void LOOP_MERGED_DFT_K_2_N(ELEMENTS* vector,int N,int n_log_2,ELEMENTS** DN
    ,struct Prime_ptr* P,ELEMENTS* Y){
  for (int i=0;i<(1<<(n_log_2-1));i++){ // basecase rhs fft2s
    int idx = i*2;
    fft2(&vector[idx],&vector[idx+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){ // for each level of recursion starting
    from the basecase to n
    int n = 1<<(n_log_2-i);
    int m = n>>1;
    // loop merge L_2^n (I_m otimes fft2) L_m^n T_m^n
    for (int j=0;j<(1<<i);j++){ // # of dft nodes for this level of
   recursion can be done concurrently
      int idx = j*n;
      for (int z=0;z<m;z++){
        ELEMENTS t1,t2;
        t1 = t2 = 0;
        t1 = MULTIPLY(&vector[idx+z],&DN[i][z],P);
        t2 = MULTIPLY(&vector[(idx+z)+m],&DN[i][z+m],P);
        Y[idx+z] = ADD(&t1,&t2,P);
        Y[idx+z+m] = SUB(&t1,&t2,P);
      }//end L_2(I_m otimes fft2)L_m T_m
      for (int z=0;z<n;z++){ // need to copy result back
        vector[idx+z]=Y[z];
      }//end copy for loop
    }// end merged loop
  }// end level loop
}//end merged_DFT_N_2
```

6.4.5 Four-Step FFT

We repeat the same process for the Four-Step FFT. We treat the initial L_2^{2m} stride permutation as we do in the SIx-Step FFT and perform our base case DFTs in a blocked fashion. This Four-Step FFT loop merge is straightforward, we merge a $D_{2,m}$ twiddle with a $DFT_2 \otimes I_m$. We simply point-wise multiply our elements with their respective twiddles before we do our l.h.s DFT_2s and while we have them in the cache.

Figure 6.6: Four-Step FFT after loop merge code listing

```
void LOOP_MERGED_FOUR_STEP_FFT_K_2_N(ELEMENTS *vector,int N, int n_log_2,
    ELEMENTS** twiddle,struct Prime_ptr* P){
  for (int i=0;i<(1<<(n_log_2-1));i++){ //pow(2,n_log_2-1)
    int idx = i * 2;
    fft2(&vector[idx],&vector[idx+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){ // for each level of recursion
    int n=1<<(n_log_2-i); //pow(2,n_log_2-i)
    int m=n>>1; // m=n/2
    for (int j=0;j<(1<<i);j++){ // for each node at current level of
    recursion
      int index = j*n;
      for (int z=0;z<m;z++){ // DFT_2 \otimes I_m D_{2,m}
        ELEMENTS tmp = 0;
        // MULTIPLY(&vector[index+z],&twiddle[i][z],P); // multiplication
    by one in a D_{2,m} Twiddle
        tmp = MULTIPLY(&vector[index+z+m],&twiddle[i][z+m],P);
        vector[index+z+m]=SUB(&vector[index+z],&tmp,P);
        vector[index+z]=ADD(&vector[index+z],&tmp,P);
      }
    }
  }
}
```

6.4.6 Dead code elimination

We want to reduce our operation count in our general FFTs. We have places where we can do this. The twiddle function is an area where we can eliminate unnecessary multiplications. If we look at the structure of the radix-2 twiddle matrix,

$$D_{2,4} \equiv \vec{d} = [1, 1, 1, 1, 1, \omega, \omega^2, \omega^3], \tag{6.22}$$

or

$$D_{2,8} = \left[1, 1, 1, 1, 1, 1, 1, 1, 1, \omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6, \omega^7\right]. \tag{6.23}$$

we see that we have $m + 1$ multiplications by 1 in every multiplication of a twiddle matrix. We can eliminate these operations and their storage requirements. First, we remove the m ones by simply adjusting our twiddle functions and our twiddle pre-computation routines. Then, we can remove the last unnecessary one in $m + 1$ by removing the unnecessary multiplication by adjusting the loop to start at 1 rather than 0.

Figure 6.7: Twiddle function code listing

```
void twiddle_pc(ELEMENTS* vector, int m, int n, ELEMENTS* omegas, struct
    Prime_ptr* P){
   int i,j,idx;
   for (i=0;i<n;i++){
      for (j=0;j<m;j++){
         idx = i*m+j;
         vector[idx]=MULTIPLY(&vector[idx],&omegas[idx],P);
      }
   }
}
```

becomes

Figure 6.8: Twiddle function parallel code listing

```
void twiddle_pc(ELEMENTS* vector, int m, int n, ELEMENTS* omegas, struct
    Prime_ptr* P){
   int j,idx;
   #pragma cilk_grainsize = GRAINSIZE_1
   p_for (j=1;j<m;j++){
         idx = m+j;
         vector[idx]=MULTIPLY(&vector[idx],&omegas[j-1],P);
   }
}
```

Now we look at our loop merged versions. We can see that the first element in the second round of base case DFT_2 is always multiplied by a twiddle with a value of one. We can take any 2-power value for n, construct the $D_{2,m}$ twiddle matrix and see that it is always the case. We can remove this multiplication from our routine. We follow the same plan and pull the first DFT_2 out in front of the loop, we remove the unnecessary multiplication, and adjust the loop counter accordingly.

Figure 6.9: Four-Step FFT code listing

```
void FOUR_STEP_FFT_K_2_N(ELEMENTS *vector,int N, int n_log_2, ELEMENTS**
    twiddle,struct Prime_ptr* P){
  for (int i=0;i<(1<<(n_log_2-1));i++){ //pow(2,n_log_2-1)
    int idx = i * 2;
    fft2(&vector[idx],&vector[idx+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){
    int n=1<<(n_log_2-i); //pow(2,n_log_2-i)
    int m=n>>1; // m=n/2
    for (int j=0;j<(1<<i);j++){
      int index = j*n;
      ELEMENTS tmp = 0;
      // First DFT pulled out front and performed without the
    multiplication by 1
      tmp = ADD(&vector[index+0],&vector[index+m],P);
      vector[index+m]=SUB(&vector[index+0],&vector[index+m],P);
      vector[index+0]=tmp;
      // loop adjusted accordingly
      for (int z=1;z<m;z++){
        // removed unnecessary MULTIPLY(&vector[index+z],&twiddle[i][z],P)
    by one of the first DFT_2 element
        tmp = MULTIPLY(&vector[index+z+m],&twiddle[i][z+m],P);
        vector[index+z+m]=SUB(&vector[index+z],&tmp,P);
        vector[index+z]=ADD(&vector[index+z],&tmp,P);
      }
    }
  }
}
```

At this point, all redundant operations have been pruned from our computation tree and we can parallelize our algorithm.

6.4.7 Parallelization

We chose our DFTs because they each express exploitable forms of parallelism. The Six-Step FFT is built from two stages of block-parallel DFTs coloured in blue

$$DFT_N = L^{km}_k(I_m \otimes DFT_k)L^{km}_m D_{k,m}(I_k \otimes DFT_m)L^{km}_k$$

and the Four-Step FFT has a block-parallel DFT, in blue, and a vector-parallel DFT in red

$$DFT_N = (DFT_k \otimes I_m)D_{k,m}(I_k \otimes DFT_m)L^{km}_k.$$

In our loop merged versions we run both types of DFT loops in parallel using Cilk (see figures 6.14 and 6.15 on pages 59 and 60). Cilk uses the keyword cilk_for to express a parallel for loop as well as the cilk_spawn and cilk_sync constructs used to express the fork-join idiom. In our Six-Step FFT (figure 6.13 on 58) we parallelize both block-parallel DFTs and parallelize our twiddle function (see figure 6.8 on page 55) and the stride-permutation function (see figure

6.12 on page 57). We use Cilk with a macro switch to quickly change between parallel and serial versions (see figure 6.10) and we also use macros to control the grain size of the parallel for loops (see figure 6.11).

Figure 6.10: Parallel macro example

```
#define PARALLEL 0
#ifdef PARALLEL
#define p_for cilk_for
#define spawn cilk_spawn
#define sync cilk_sync
#else
#define p_for for
#define spawn
#define sync
#endif
```

Figure 6.11: Parallel grain size macro example

```
#define GRAINSIZE_1 8
#define GRAINSIZE_2 4
#define GRAINSIZE_3 2
```

Figure 6.12: Transpose parallel code listing

```
void transpose(ELEMENTS* A, int m, int n,ELEMENTS* B){
  long int N=0;
  N = m*n;
  #pragma cilk_grainsize = GRAINSIZE_3
  p_for (int i=0;i<n;i++){//0,1
    for (int j=0;j<m;j++){//0..8
      B[j*n+i] = A[i*m+j]; //B[idx_b]=A[idx_a]
    }
  }
#pragma cilk_grainsize = GRAINSIZE_1
  p_for (int i=0;i<N;i++){
    A[i]=B[i];
  }
}
```

Figure 6.13: Six-Step FFT parallel code listing

```
void SIX_STEP_FFT_N_2(ELEMENTS* vector,int N,int n_log_2,ELEMENTS** twiddle
    ,Prime_ptr* P, ELEMENTS* B){
  int idx,m;
#pragma cilk_grainsize = GRAINSIZE_1
  p_for (int i=0;i<(1<<(n_log_2-1));i++){
    fft2(&vector[i*2],&vector[i*2+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){ // for each level of recursion
    m = 1<<(n_log_2-i); // size of current dft_n
    #pragma cilk_grainsize = GRAINSIZE_2
    p_for (int j=0;j<(1<<i);j++){ // loop through each node of L_2^m (I_m/2
    \otimes DFT_2)L_m/2^m D_{2,m/2}^m and do L_m/2^m D_{2,m/2}^m part
      idx = j*m;
      twiddle_pc(&vector[idx],m/2,2,twiddle[i],P);
      stride_permutation(&vector[idx],m/2,2,&B[idx],2);
    }
    #pragma cilk_grainsize = GRAINSIZE_2
    p_for (int j=0;j<(1<<i);j++){ // loop through each node of L_2^m (I_m/2
    \otimes DFT_2)L_m/2^m D_{2,m/2}^m and do (I_m/2 \otimes DFT_2) part
        idx = j*m;
      #pragma cilk_grainsize = GRAINSIZE_2
        p_for (int z=0;z<m/2;z++){ // at each node do (I_m/2 \otimes DFT_2)
          int index = idx+z*2;
          fft2(&vector[index],&vector[index+1],P);
        }
    }
    #pragma cilk_grainsize = GRAINSIZE_1
    p_for (int j=0;j<(1<<i);j++){ // loop through each node of L_2^m (I_m/2
    \otimes DFT_2)L_m/2^m D_{2,m/2}^m and do  L_2^m part
      idx=j*m;
      stride_permutation(&vector[idx],2,m/2,&B[idx],2);
    }
  }
}//end DFT_N_2
```

Figure 6.14: Six-Step merged FFT parallel code listing

```
void SIX_STEP_MERGED_FFT_N_2(ELEMENTS* vector,const int N,const int n_log_2
    , ELEMENTS** DN, Prime_ptr* P, ELEMENTS *Y){
#pragma cilk_grainsize = GRAINSIZE_1
  p_for (int i=0;i<(1<<(n_log_2-1));i++){ // parallel basecase FFT_2s
    int idx = i*2;
    fft2(&vector[idx],&vector[idx+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){ // for each recursion step starting
    from the basecase to n
    int n = 1<<(n_log_2-i); //n_log 2=6 i=0 n=64 m=32
    int m = n>>1;
    #pragma cilk_grainsize = GRAINSIZE_2
    p_for (int j=0;j<(1<<i);j++){ //for each dft node at this recursion
    step
      int idx = j*n;
      #pragma cilk_grainsize = GRAINSIZE_3
      p_for (int z=0;z<m;z++){ // do L_2^n (I_m otimes fft2) L_m^n T_m^n
          ELEMENTS t2;
          t2 = 0;
          t2=MULTIPLY(vector[(idx+z)+m],DN[i][z+m],P);
          Y[idx+z]=ADD(vector[idx+z],t2,P);
          Y[idx+z+m]=SUB(vector[idx+z],t2,P);
      }//end L_2(I_m otimes fft2)L_m T_m
      #pragma cilk_grainsize = GRAINSIZE_1
      p_for (int z=0;z<n;z++){ // copy result back
         vector[idx+z]=Y[idx+z];
      }//end copy for loop
    }// end merged loop
  }// end recursion step loop
}//end DFT_N
```

Figure 6.15: Four-Step FFT parallel code listing

```
void FOUR_STEP_FFT_K_2_N(ELEMENTS *vector,int N, int n_log_2, ELEMENTS**
    twiddle, Prime_ptr* P){
 #pragma cilk_grainsize = GRAINSIZE_1
  cilk_for (int i=0;i<(1<<(n_log_2-1));i++){ //pow(2,n_log_2-1)
    int idx = i * 2;
    fft2(&vector[idx],&vector[idx+1], P);
  }
  for (int i=n_log_2-1-1;i>=0;i--){ // for each recursion step
    int n=1<<(n_log_2-i); //pow(2,n_log_2-i)
    int m=n>>1; // m=n/2
 #pragma cilk_grainsize = GRAINSIZE_2
    cilk_for (int j=0;j<(1<<i);j++){  // do (DFT_2 \otimes I_m) D_{2,m} for
    each FFT node at recursion step
      int index = j*n;
      ELEMENTS a;
      // First FFT_2 pulled out front and performed without the
    multiplication by 1
      a=ADD(&vector[index],&vector[index+m],P);
      vector[index+m]=SUB(&vector[index],&vector[index+m],P);
      vector[index]=a;
      #pragma cilk_grainsize = GRAINSIZE_3
      p_for (int z=1;z<m;z++){    // loop adjusted accordingly
        ELEMENTS tmp = 0;
        // removed unnecessary MULTIPLY(&vector[index+z],&twiddle[i][z],P)
    by one of the first DFT_2 element
        tmp = MULTIPLY(&vector[index+z+m],&twiddle[i][z+m],P);
        vector[index+z+m]=SUB(&vector[index+z],&tmp,P);
        vector[index+z]=ADD(&vector[index+z],&tmp,P);
      }
    }
  }
}
```

We have our three general functions. An explicit Six-Step FFT 6.13, a loop-merged Six-Step FFT 6.14 and a Four-Step FFT 6.15. After optimization, all FFTs require that the input vector be pre-shuffled and the pre-computed twiddle matrices passed in and a base case size selected. The BPAS library wrapper can perform the pre-computations and the pre-shuffle when we have the size of the transform and the field characteristic. Given the size of the field characteristic we can experimentally find or compute the best-performing size of base case *DFT* block.

Chapter 7

Experimentation

In this chapter we first discuss the generality of our FFT schemes and how we switch between contexts. Next, we discuss the experimental setup. Then, we benchmark our FFTs in a serial fashion against the high performance FFT already implemented in the BPAS library by Svyatoslav Covanov [6]. Then, we run a serial vs parallel experiment for the arbitrary big prime field. Next, we run an experiment using a generalized Fermat prime field version. Lastly, we measure our C++ implementation against a C version to measure the performance penalty and conclude with observations.

7.1 Genericity and context switching

We have our three generic FFT schemes: Six-Step FFT from Procedure 9 on Page 45, Six-Step Merged FFT from Procedure 10 on Page 46, and the Four-Step FFT from Procedure 11 on Page 47. We call these three FFT schemes generic because the underlying finite field arithmetic is not specialized, on the contrary of the highly optimized FFT codes [6, 7] against which we compare our more generic FFT codes. Properties such as: problem size, base case size, type of prime number, and implementation language, are generally selected beforehand and with a particular FFT scheme in mind. In this context, we want to compare these three generic schemes against the specialized FFTs from [6] and [7].

All three of the generic FFT schemes use the same loop-unrolled base cases and the base case size selection is the basis for an overall blocking strategy with respect to data locality. Experimental data is collected for each base case size for each size of input vector. Templates can be used for the loop-unrolled base cases, the stride permutation related functions, and the twiddle related functions. We can define macros as in figure 7.1 on page 62 and use the same code for each of the three FFT schemes to generate code for all three C++ Prime Fields. The C++ Fields have a common interface. The field multiplication scheme is encapsulated by the class and the field elements are created in the correct representation for their respective fields.

Figure 7.1: C++ prime field macros

```
#define ADD(X,Y) X+Y
#define SUB(X,Y) X-Y
#define MULTIPLY(X,Y) X*Y
#define POW(X,Y) X^Y
#define SWAP(Y) swap(X,Y)
```

When we switch from C++ to C or when we want to take advantage of special field properties, we lose some generality. Switching to the C classes requires different initialization schemes and involves the passing around of pointers to initialized structures containing field properties. Field properties include the characteristic prime as well as data that is required for the multiplication algorithms. These differences are handled using the preprocessor and conditional inclusions (see figure 7.2 on page 62). For example, when switching from the small prime field C++ to the small prime field C version, we need to convert elements in and out of Montgomery representation before and after the computation and it uses a slightly different interface (because of a pointer to a data structure) which the code needs to accommodate (see figure 7.3).

Figure 7.2: Conditional inclusion code snippet

```
#ifdef SMALLPRIMEFIELD_C
  for (int i = 0; i < n; i++){
        a[i]=CONVERT_IN(a[i], P);
        b[i]=CONVERT_IN(b[i], P);
  #ifdef DEBUG
            cout<<"a["<<i<<"]="<<a[i]<<endl;
            cout<<"b["<<i<<"]="<<b[i]<<endl;
  #endif
  }
#endif
```

Figure 7.3: C small prime field macros

```
#define ADD(X,Y,P) smallprimefield_add(X,Y,P)
#define SUB(X,Y,P) smallprimefield_sub(X,Y,P)
#define MULTIPLY(X,Y,P) smallprimefield_mul(X,Y,P)
#define POW(X,Y,P) smallprimefield_exp(X,Y,P)
#define SWAP(X,Y) swap(X,Y)
#define CONVERT_IN(X,P) smallprimefield_convert_in(X,P)
#define CONVERT_OUT(X,P) smallprimefield_convert_out(X,P)
```

When we switch from the C++ big prime field to the big prime field C (GMP) version, we need a pointer to the modulus and the interface changes by the addition of a result operand that the code needs to accommodate (see Figure 7.4). Additionally, the size of the characteristic prime can affect the algorithms used by the GMP library. For this reason, in the big prime field C (GMP) experiment we use the same 128-bit prime named P2 in the table 7.1 on page 64

throughout to avoid any algorithm changes during the experiment. GMP breaks down multiple precision numbers into limbs which in the case of a 128 bit number like P2 would be 2 limbs per field element using a 64-bit limb size. It selects arithmetic algorithms based on the number of limbs. Since the number of limbs per field element is dictated by the size of the field characteristic, the underlying GMP algorithms remain consistent for each FFT scheme at each particular input size throughout the experiment.

Figure 7.4: C big prime field macros

```
#define ADD(R,X,Y,P) mpz_add(R,X,Y);mpz_mod(R,R,P)
#define SUB(R,X,Y,P) mpz_sub(R,X,Y);mpz_mod(R,R,P)
#define MULTIPLY(R,X,Y,P) mpz_mul(R,X,Y);mpz_mod(R,R,P)
#define POW(R,X,Y,P) mpz_powm(R,X,Y,P)
#define SWAP(X,Y) mpz_swap(X,Y)
```

When we switch from arbitrary big prime field C to a generalized Fermat prime field C, the interface for the arithmetic functions changes. Now, the result of each type of arithmetic operation overwrites the first of the input operands with the result. This difference requires the use of a facade to modify the behaviour so that the operations no longer overwrite the first operand. The facade has to make a copy of the first operand, perform the operation using the copy (to preserve the original operand), and then return the result. Next, as in [24] the base case twiddle multiplications get replaced by multiplications by powers of R. This multiplication algorithm is discussed in [24] and the specialized FFT from [24] is used as the benchmark FFT in our generalized Fermat prime field experiment. The arbitrary inter-element multiplication function is also discussed in [24] and replaces the generic multiplication algorithm during the experiment (see Figure 7.5 on Page 63). As before, any necessary data structures need to be initialized and accommodated using the preprocessor and conditional inclusions.

Figure 7.5: C generalized Fermat prime field macros

```
#define ADD(R,X,Y,K,P) R=addition_big_elements (X, Y, K, P.radix)
#define SUB(R,X,Y,K,P) R=subtraction_big_elements (X, Y, K, P.radix)
#define MULTIPLY(R,X,Y,K,P) R=gfpf_multiplication(X, Y, K, &
    t_crt_data_global , &t_lhc_data_global)
#define MULPOWR(R,X,Y,K,P) R=mult_pow_R(X,Y,K,P.radix)
#define POW(R,X,Y,K,P) mpz_powm(R,X,Y,K,P)
#define SWAP(X,Y) mpz_swap(X,Y)
```

Table 7.1: Primes table

Label	Prime
P1	655360001
P2	$923237923610951680^2 + 1$
P3	$8646911284551372^4 + 1$
P4	$7205764901350932^8 + 1$
P5	$3242943575427645^{16} + 1$
P6	$3242591731708067^{32} + 1$

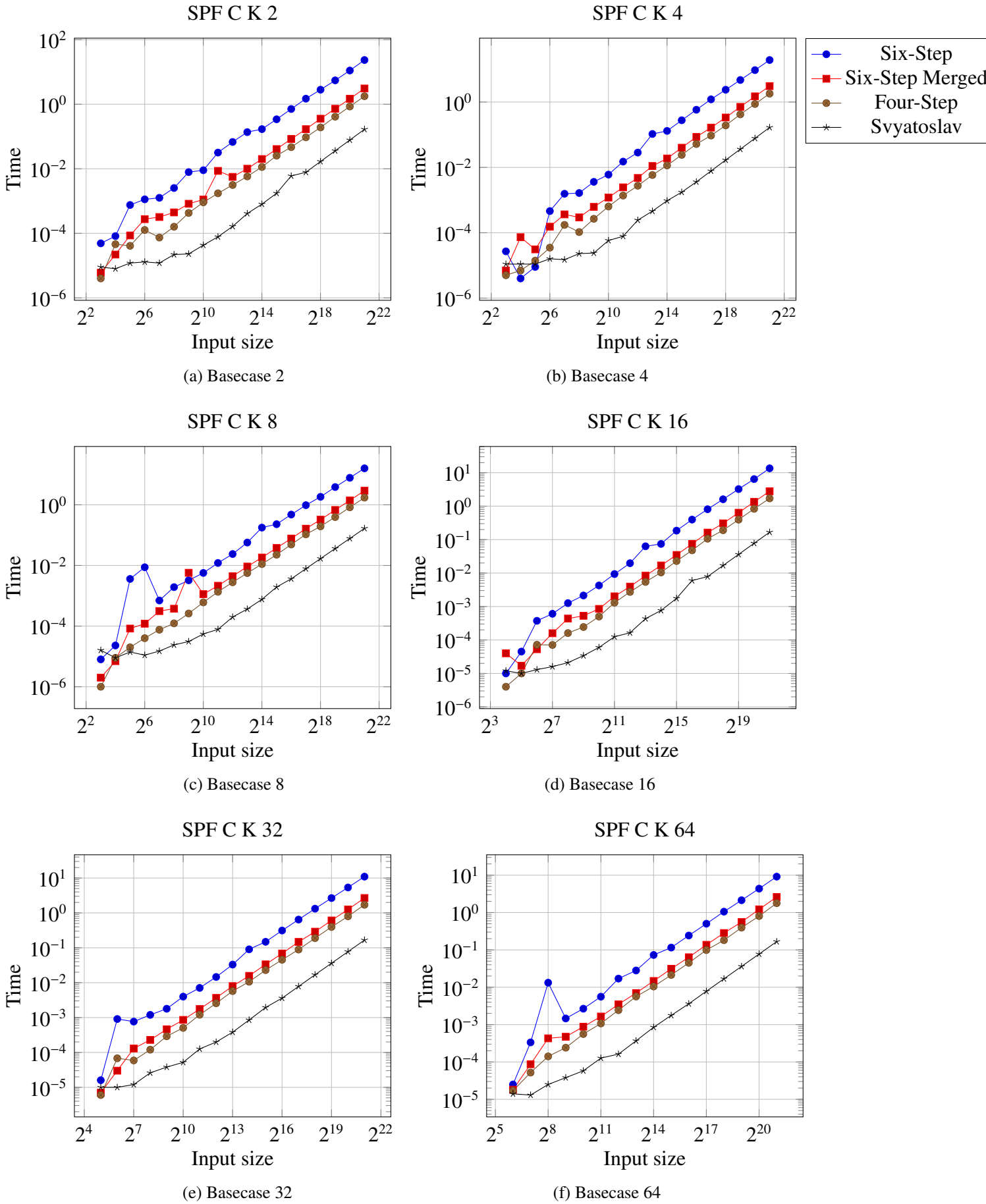
7.2 Experimental setup

We generate a randomized input vector and use the same input data for the various FFTs and compare the runtimes of our various base case block sizes. Each FFT version gets the same field characteristic and value of ω_n as input for each of the problem sizes. The serial experiment is run on a single core of a 2.66 GHz Intel Xeon CPU, while the parallel experiment is run on the same 2.66 GHz Intel Xeon CPU but with 12 cores. We use macros to switch between serial and parallel versions. The preprocessor macro switch adds/removes parallel code to/from the serial code. Following the recommendations in [11], our test harness initializes our data structures including our twiddle matrices and kernel twiddle vectors. Then we perform a warm-up run in advance of the experiment. Next, we read the current time, call our kernel multiple times and then read the time again. We then divide the time difference by the number of kernel calls. We use the same test harness structure and collect data in the same fashion for all experiments. Various bash scripts are used for the collation of data. Table 7.1 on Page 64 lists the prime numbers used in the following experiments.

7.3 Serial small prime field C: generic vs specialized benchmark

In this section we benchmark our code against a specialized high performance FFT in the BPAS library created by Svyatoslav Covanov. The graphs from figure 7.6 on page 65 use a log-log axis. The experiment uses input sizes ranging from 2^1 to 2^{21} and base case sizes: 2,4,8,16,32, and 64. The generic FFTs use an arbitrary prime $P1$ and use the same arithmetic as found in the small prime field c++ class. The specialized benchmark FFT uses a specific prime, 4179340454199820289, and specialized arithmetic. Looking at figure 7.6 on page 65, we can see from the results that serial execution of the Six-Step explicit FFT is slowest for all serial runs with all the different size K, in every benchmark test. We attribute this to the 3 additional passes through the data for each recursion step when compared to the other FFT schemes. The Six-Step merged and Four-step merged FFTs perform better due to having to pass through the data less times at each recursion step. Finally, we see that, out of the three FFTs, The Four-step loop merged FFT is closest to the benchmark, next the Six-Step loop-merged FFT, followed by the Six-Step explicit.

Figure 7.6: Serial small prime field C benchmark.



(a) Basecase 2

(b) Basecase 4

(c) Basecase 8

(d) Basecase 16

(e) Basecase 32

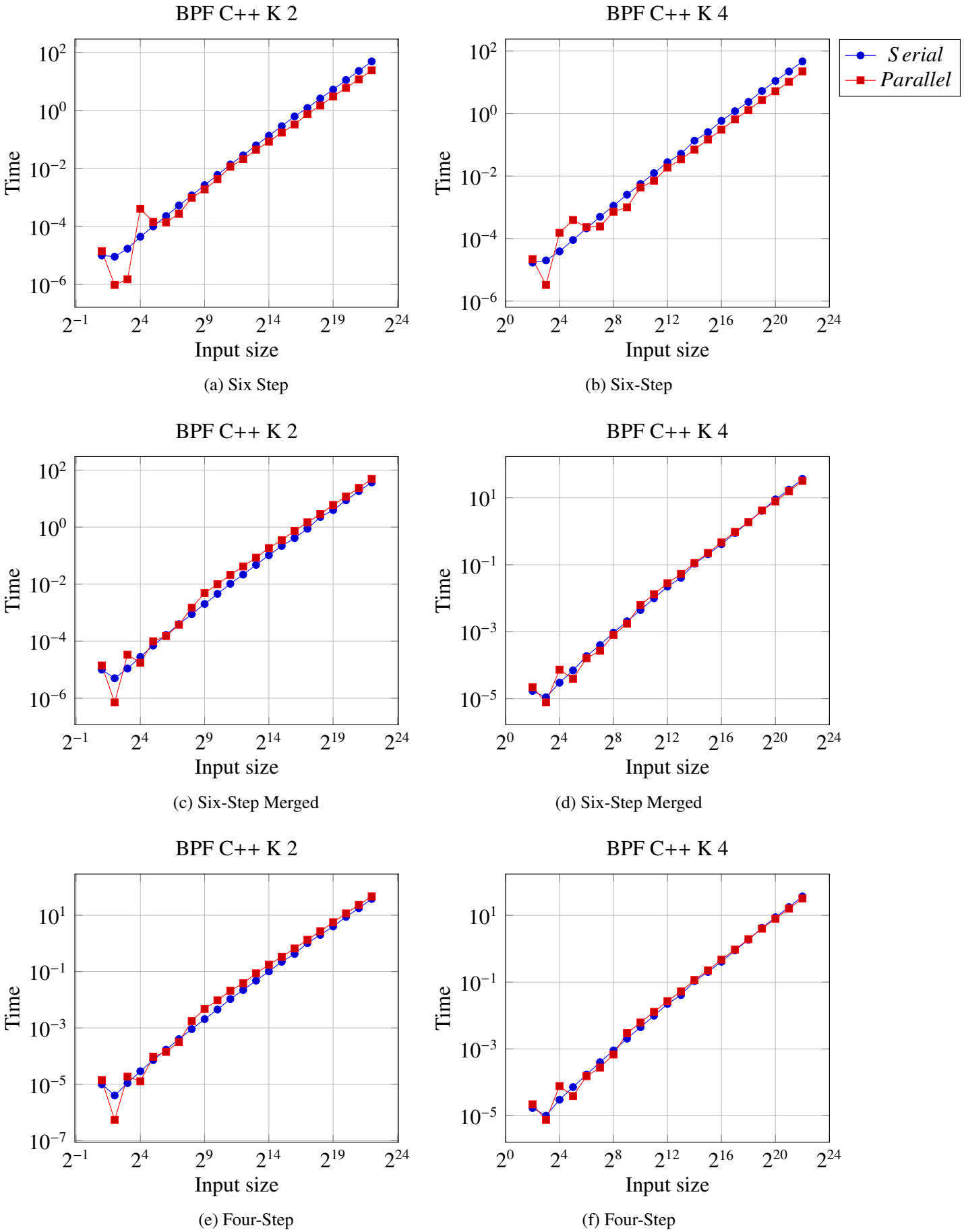
(f) Basecase 64

7.4 Serial VS parallel C++ big prime experiment (generic)

In this section we compare the C++ versions of the arbitrary big prime field in a serial vs parallel experiment. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400 01. The experiment uses input sizes that range from 2^1 to 2^{20} and base case sizes: 2,4,8,16,32, and 64. The generic FFTs all use an arbitrary prime $P2$ and the same arithmetic. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. The graphs in figure 7.7 use a log-log axis.

Looking at figure 7.7 on page 67 the largest input size for base case 2, we realize a speedup of 2.03 for the Six-Step explicit FFT while the Six-Step loop merged has a measured speedup of 0.75 and Four-Step loop merged version has a measured speedup of 0.82. For the basecase kernels four in size, at the largest input size, we see that the Six-Step explicit shows a measured speed up of 2.08 while the Six-Step loop merged version measured a speed up of 1.15 and the Four-Step speed up was measured at 1.16.

Figure 7.7: Serial vs parallel C++, $K = 2$ $P2$ and $K = 4$ $P2$



(a) Six Step

(b) Six-Step

(c) Six-Step Merged

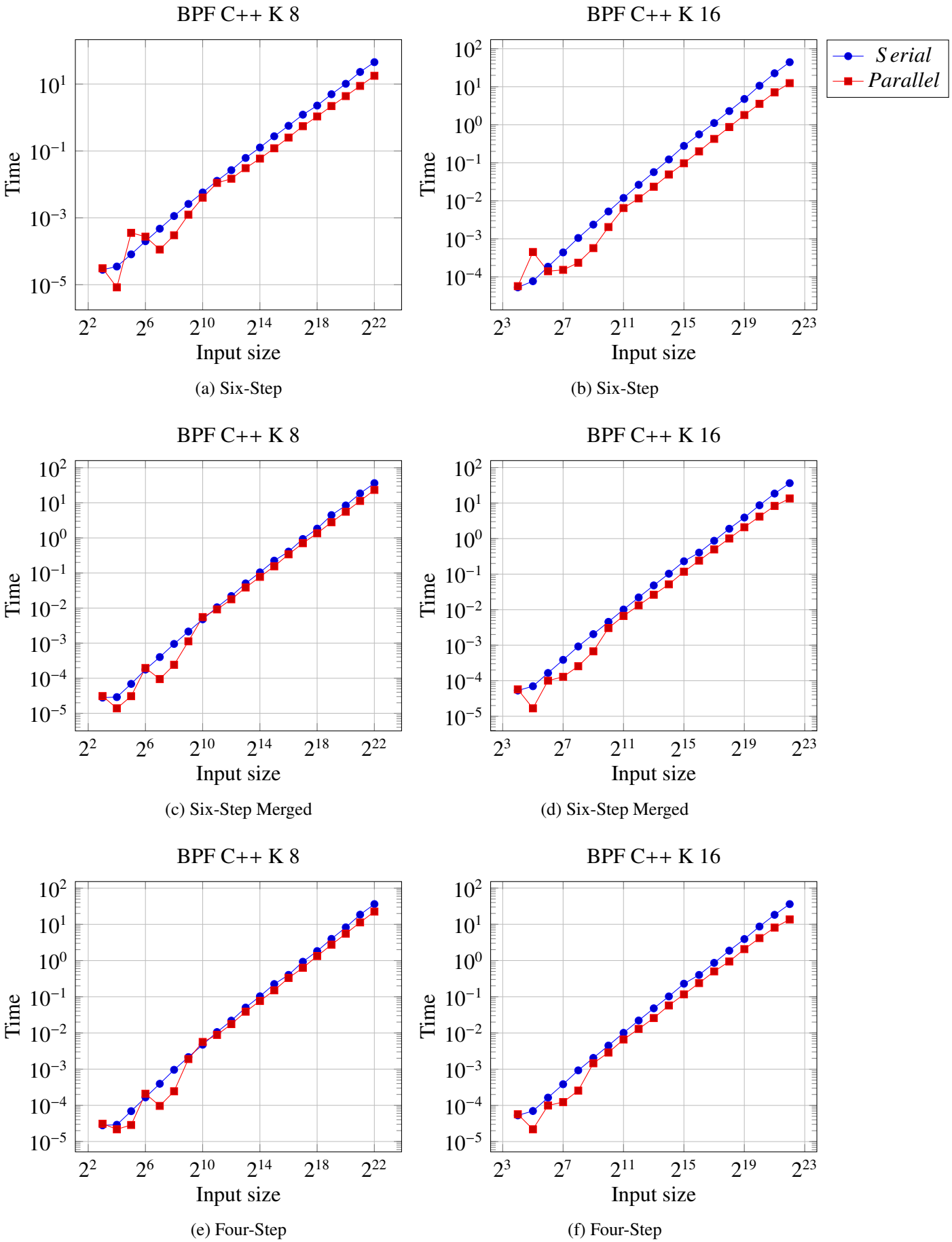
(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Next, we compare the C++ versions of the arbitrary big prime field in a serial vs parallel experiment with base case values 8 and 16. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400 01. The experiment uses input sizes that range from 2^1 to 2^{20}. The generic FFTs all use an arbitrary prime $P2$ and the same arithmetic. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. The graphs from figure 7.8 on page 69 use a log-log axis.

Figure 7.8: Serial vs parallel C++, $K = 8$ $P2$ and $K = 16$ $P2$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

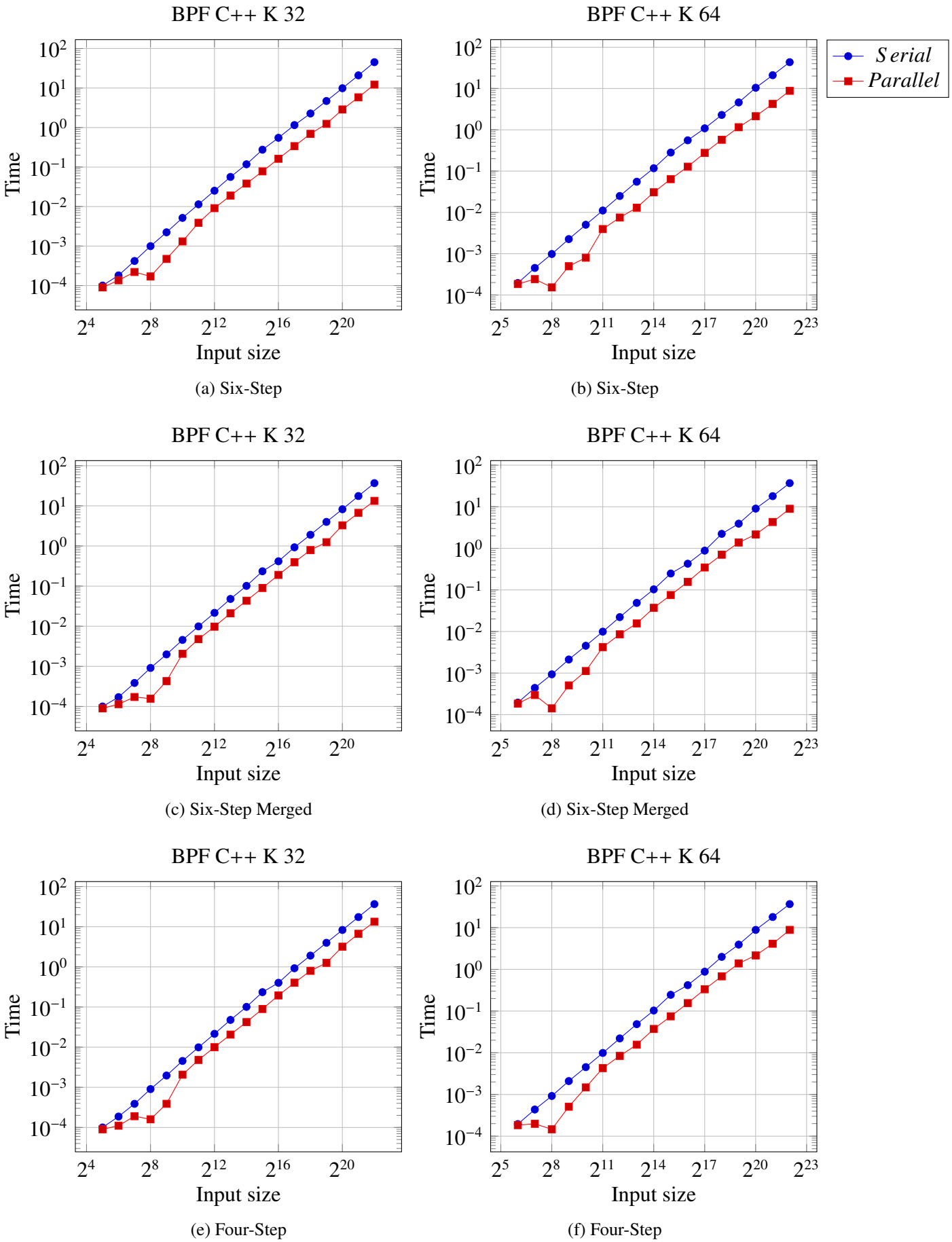
Looking at figure 7.8 on page 69, for our base case kernels of size 8, at the largest transform size we measure a speedup of 2.55 for the Six-Step, 1.56 speed up for the Six-Step loop merged, and, a speed up of 1.61 for the Four-Step. From the same figure 7.8 on page 69, looking now at the base case kernels of size 16, at the largest transform size we measure a speedup of 3.56 for the Six-Step, 2.7 speed up for the Six-Step loop merged, and, a speed up of 2.65 for the Four-Step.

Next, we compare the C++ versions of the arbitrary big prime field in a serial vs parallel experiment with base case values 32 and 64. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400001. The experiment uses input sizes that range from 2^1 to 2^{20}. The generic FFTs all use an arbitrary prime $P2$ and the same arithmetic.

We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. The graphs from figure 7.9 on page 71 use a log-log axis.

Looking at figure 7.9 on page 71, for our base case kernels of size 32, at the largest transform size we measure a speedup of 3.7 for the Six-Step, 2.76 speed up for the Six-Step loop merged, and, a speed up of 2.74 for the Four-Step. Looking at the base case kernels of size 64 from figure 7.9 on page 71, we see better speedup when using the larger kernel sizes.

Figure 7.9: Serial vs parallel C++, $K = 32$ $P2$ and $K = 64$ $P2$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

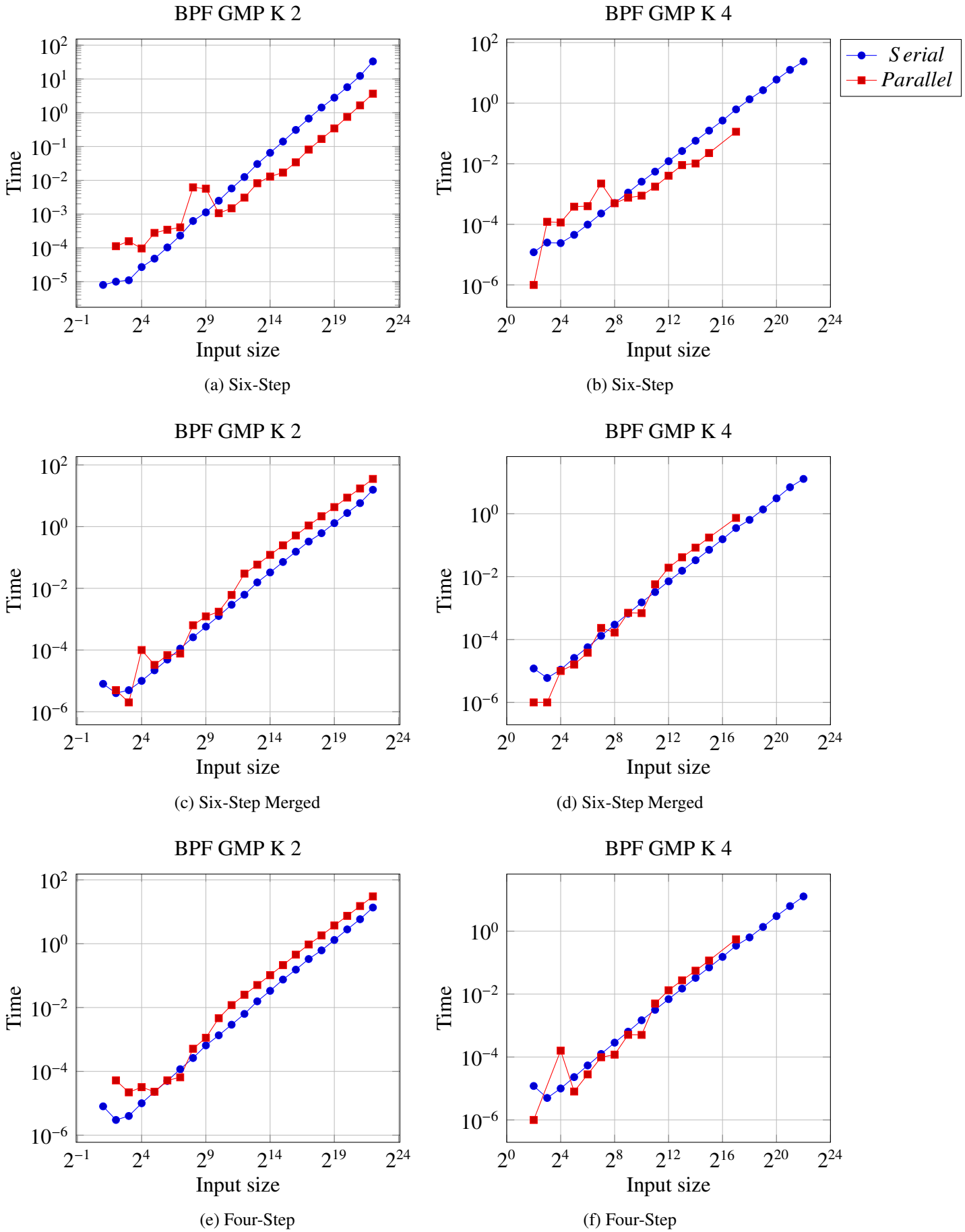
(f) Four-Step

7.5 Serial VS parallel GMP

In this section we compare the C versions of the arbitrary big prime field in a serial vs parallel experiment. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400 01. The experiment uses input sizes that range from 2^1 to 2^{22} and base case sizes: 2,4,8,16,32, and 64. The generic FFTs all use an arbitrary prime $P2$ and the same arithmetic for both serial and parallel runs. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. The graphs 7.10 on page 73 use a log-log axis and uses base case sizes 2 and 4. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400 01. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. Figure 7.10 on page 73 shows that for our GMP experiment using base case kernels of size 2 at the same transform size measured a speed up of 9.08 for the Six-Step, 0.3 speed up for the Six-Step loop merged, and a speed up of 0.34 for the Four-Step. Figure 7.10 on page 73 shows that for our GMP experiment using base case kernels of size 4 at the same transform size measured a speedup of 4.93 for the Six-Step, 0.21 speed up for the Six-Step loop merged, and a speed up of 0.28 for the Four-Step. The graphs 7.11 on page 74 use a log-log axis and uses base case sizes 8 and 16. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400 01. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. We can see from the graph in figure 7.11 on page 74 that for our GMP experiment using base case of size 8, at the same input size, we measure a speedup of 1.62 for the Six-Step, 0.28 speed up for the Six-Step loop merged, and 0.26 speed up for the Four-Step. Also in figure 7.11 on page 74, using base case size 16, we measure the greatest speedup of 0.64 for the Six-Step, 0.19 speed up for the Six-Step loop merged, and a speed up of 0.21 for the Four-Step.

The graphs 7.12 on page 75 use a log-log axis and use base case sizes 32 and 64. The field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 852368263593461449566383235294822400 01. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization. Looking at figure 7.12 on page 75, for our GMP experiment using base case kernels of size 32, we measure the greatest speedup as 1.19 for the Six-Step, 0.46 speed up for the Six-Step loop merged, and a speed up of 9.41 for the Four-Step. Also from figure 7.12 on page 75, for our GMP experiment using base case kernels of size 64, we measure the greatest speedup of 1.47 for the Six-Step, 0.96, speed up for the Six-Step loop merged, and a speed up of 0.31 for the Four-Step.

Figure 7.10: Serial vs parallel GMP, $K = 2$ $P2$ and $K = 4$ $P2$



(a) Six-Step

(b) Six-Step

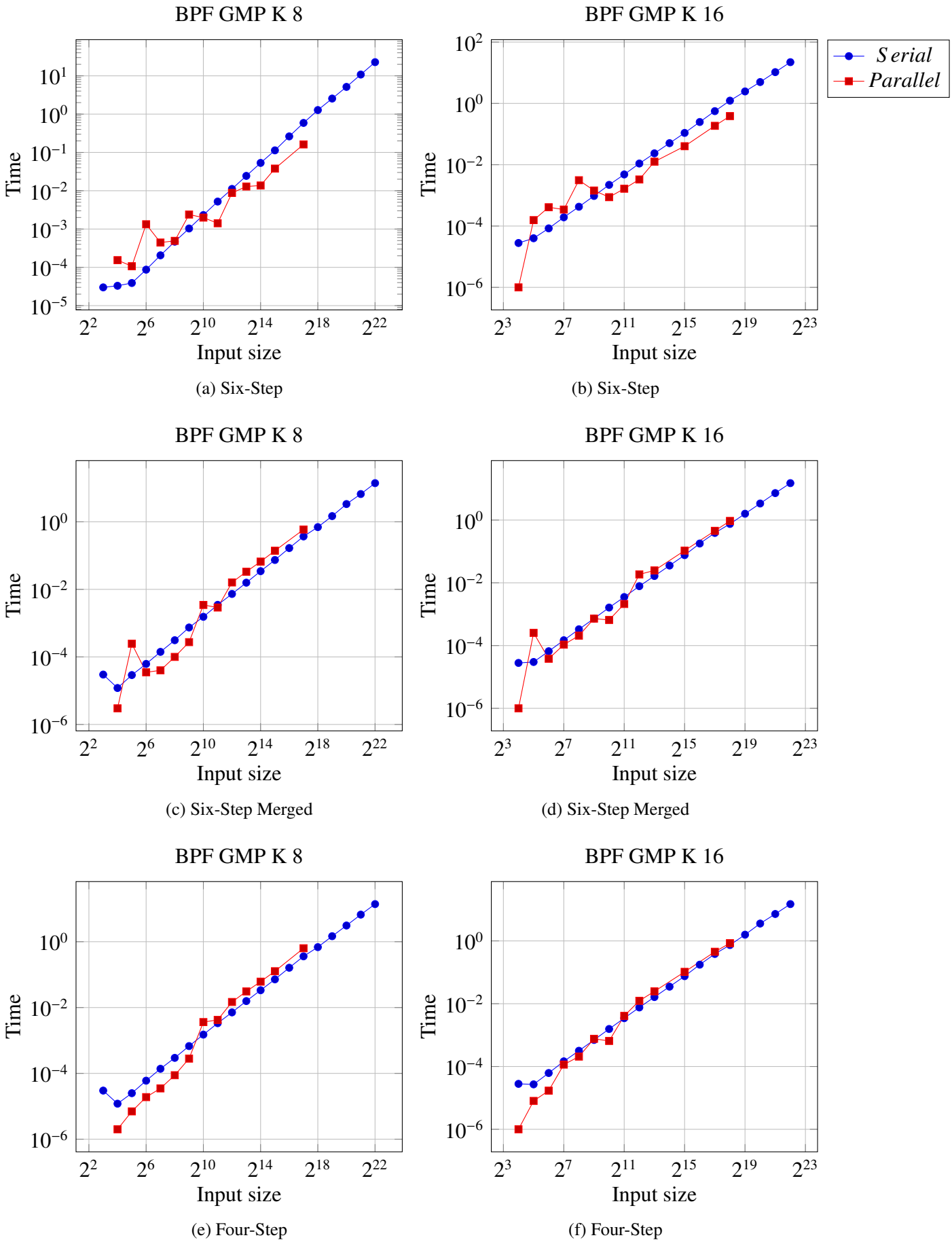
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.11: Serial vs parallel GMP, $K = 8$ $P2$ and $K = 16$ $P2$



(a) Six-Step

(b) Six-Step

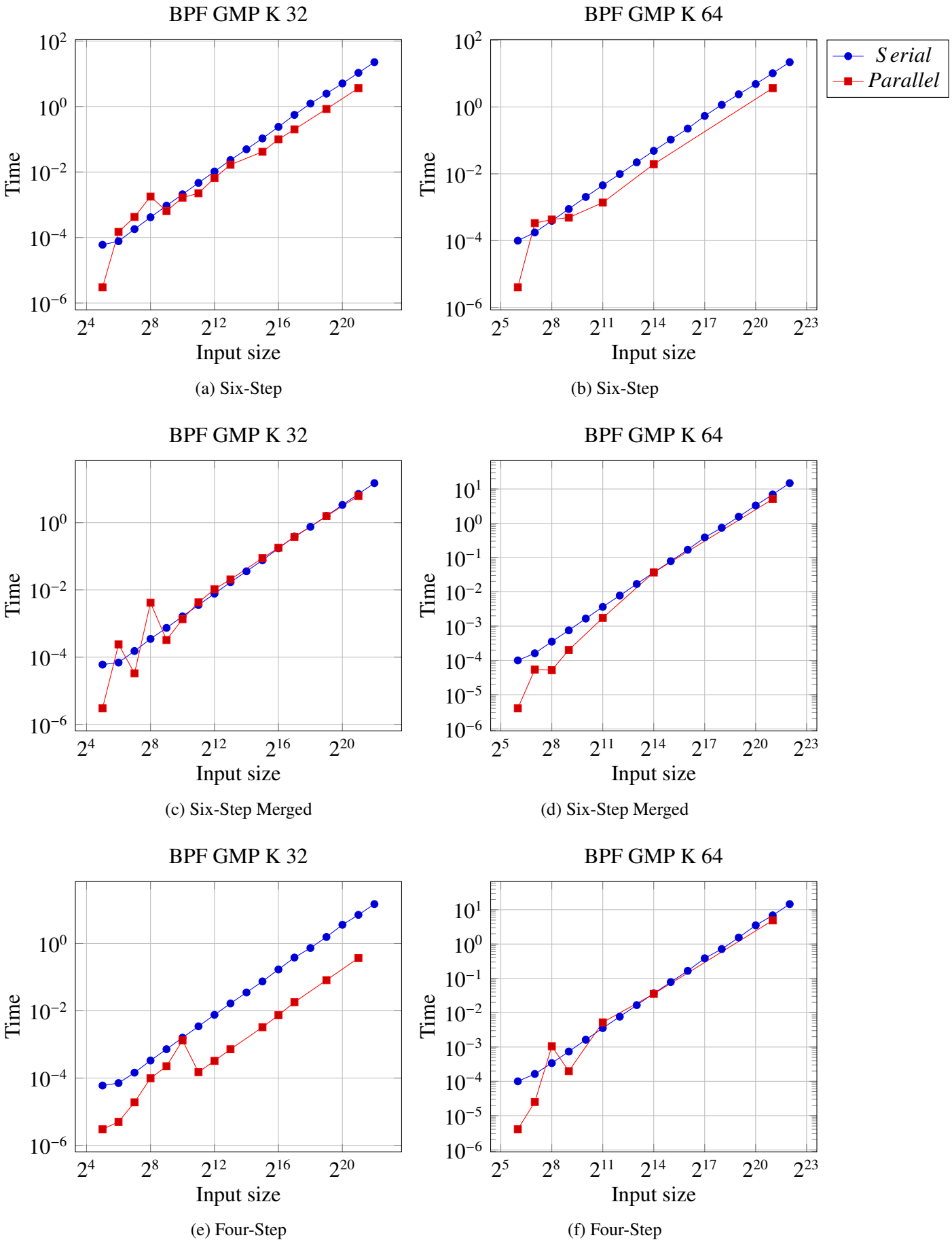
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.12: Serial vs parallel GMP, $K = 32$ $P2$ and $K = 64$ $P2$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

7.6 C VS C++ small prime field experiment

In this section we compare the serial C and C++ versions of the small prime field. The field characteristic is labeled P1 in the table 7.1 on page 64 and has a value of 655360001. The experiment uses input sizes that range from 2^1 to 2^{21} and base case sizes: 2,4,8,16,32, and 64. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU. We compile all implementations with option -o2 to obtain further optimization.

The figure 7.13 on page 77 uses base case values 2 and 4. It has a log-log axis and the field characteristic is P1.

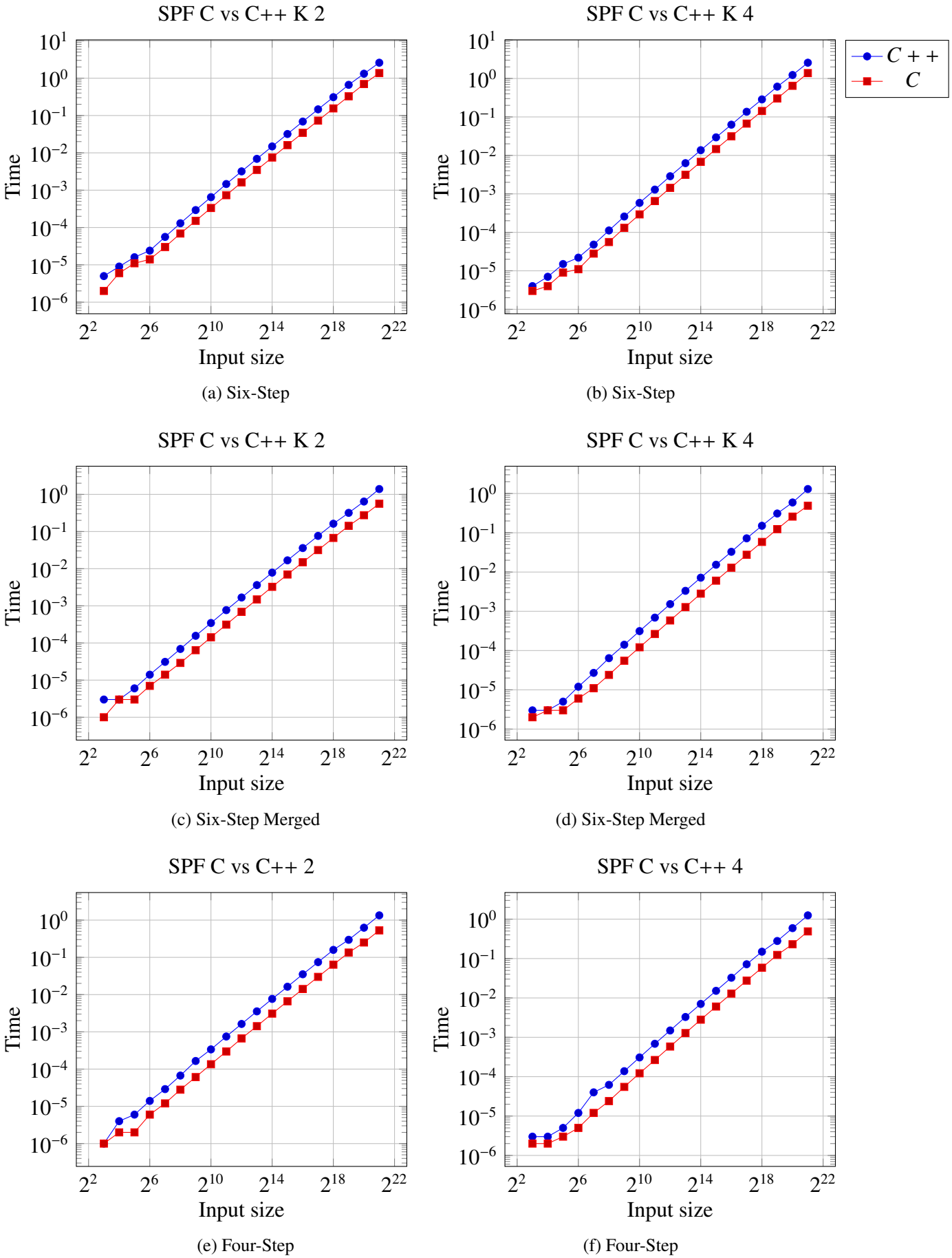
We see from figure 7.13 on page 77 that our C++ versions are not as competitive against our C version. The overhead appears to be consistent. We see that we get the best performance from our C code.

The next experiment uses input sizes that range from 2^1 to 2^{21} and base case sizes 8 and 16. We run this serial experiment on a single core of a 2.66 GHz Intel Xeon CPU. We compile all implementations with option -o2 to obtain further optimization. The figure 7.14 on page 78 uses a log-log axis and the field characteristic is labeled P1 in the table 7.1 on page 64 with a value of 655360001. We see from figure 7.14 on page 78 that our C++ versions are not as competitive against our C version. The overhead appears to be consistent. We see that we get the best performance from our C code.

The next experiment uses input sizes that range from 2^1 to 2^{21} and base case sizes 32 and 64. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU. We compile all implementations with option -o2 to obtain further optimization. The figure 7.15 on page 79 uses a log-log axis and the field characteristic is labeled P1 in the table 7.1 on page 64 with a value of 655360001. Again we see from figure 7.15 on page 79 that our C++ versions are not as competitive against our C version. The overhead appears to be consistent. We see that we get the best performance from our C code.

In the small prime field C vs C++ experiments, the C++ classes have layers of inheritance affecting performance. If classes lower down in inheritance have data members then offsets are required to maintain pointer arithmetic and obviously this isn't without performance cost. By carrying around additional pointers and other inherited data [5], the actual field element data isn't necessarily contiguously laid out in memory, as in the C version with direct function calls [9]. Also, calling virtual functions is more expensive than calling non-virtual functions because C++ uses the virtual pointers to get to the appropriate virtual table. Then it has to index the virtual table to find the appropriate function to call. With at least 8 layers of inheritance for the field classes there is non-trivial overhead that is associated with the measured slow down.

Figure 7.13: SPF C vs C++,$K = 2$ $P1$ and $K = 4$ $P1$



(a) Six-Step

(b) Six-Step

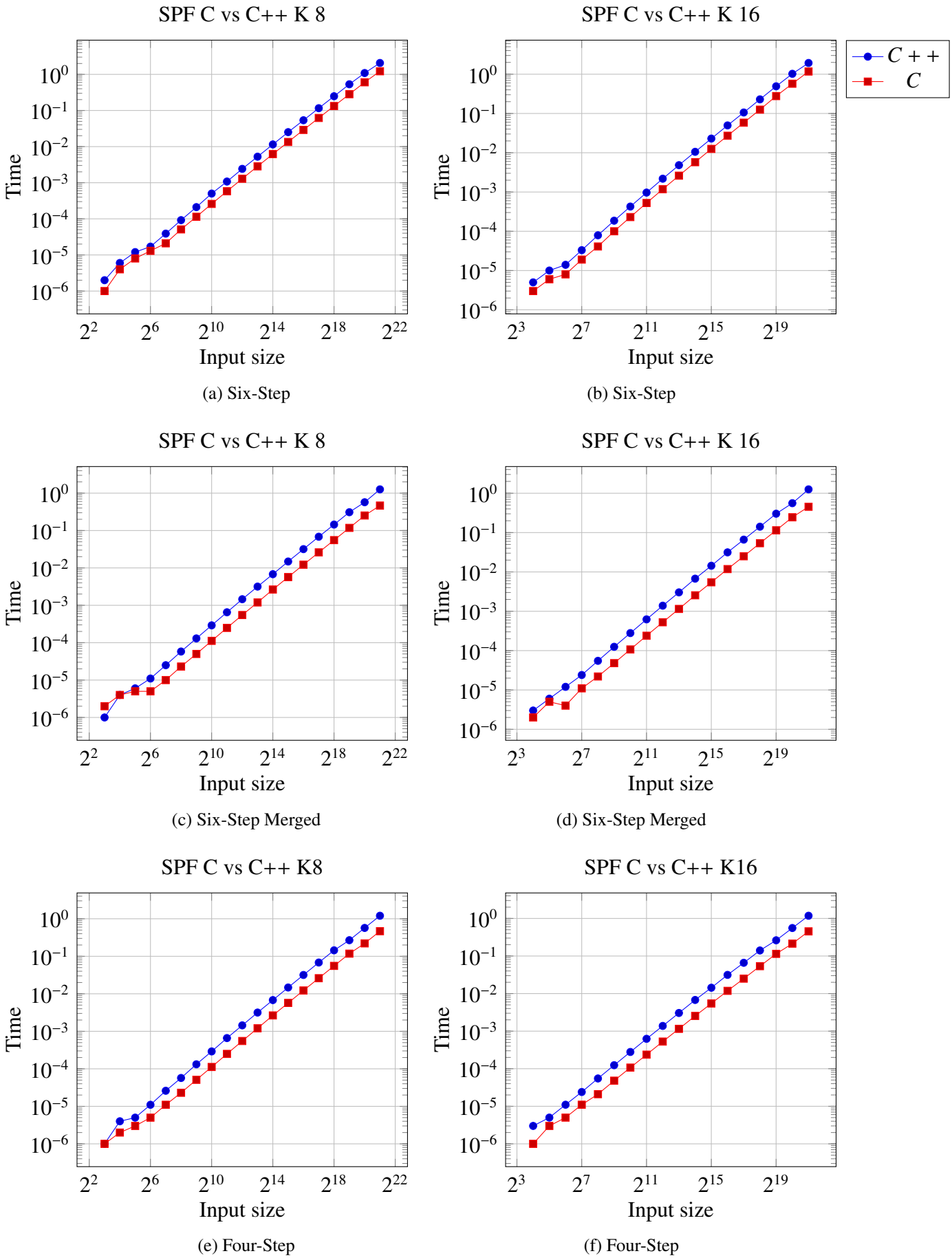
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.14: SPF C vs C++,$K = 8$ $P1$ and $K = 16$ $P1$



(a) Six-Step

(b) Six-Step

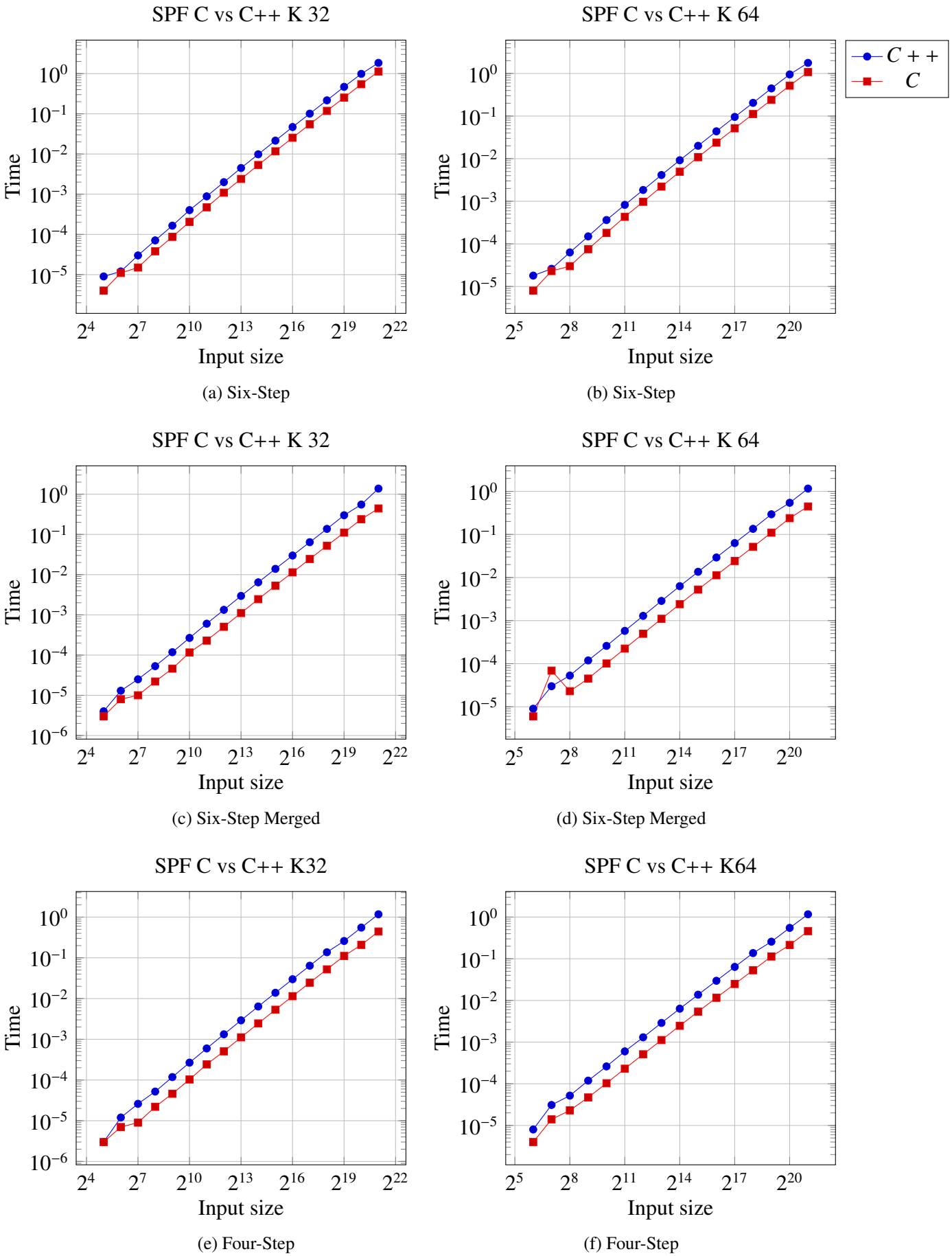
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.15: SPF C vs C++,$K = 32$ $P1$ and $K = 64$ $P1$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

7.7 C VS C++ big prime field experiment

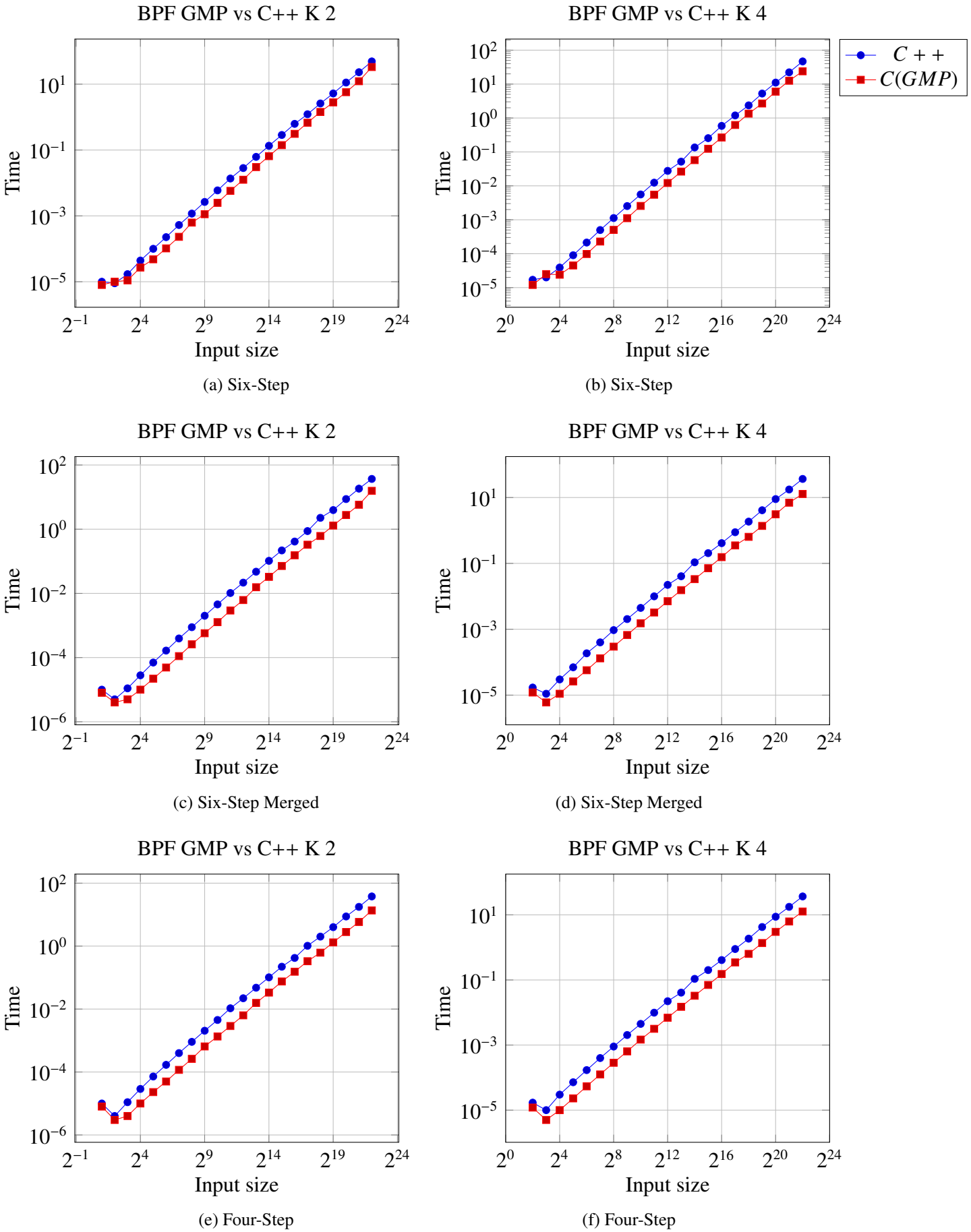
In this section we look at the C vs C++ comparison of the generic FFTs over a big prime field. The next experiment uses input sizes that range from 2^1 to 2^{20} and base case sizes 2 and 4. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU. We compile all implementations with option -o2 to obtain further optimization. The figure 7.16 on page 81 uses a log-log axis and the field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 85236826359346144956638323529482240001. We see from figure 7.16 on page 81 that our C++ versions are not as fast as our GMP version. The difference appears consistent. We see that we get the best performance from our C based GMP code.

The next experiment uses input sizes that range from 2^1 to 2^{20} and base case sizes 8 and 16. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU. We compile all implementations with option -o2 to obtain further optimization. The figure 7.17 on page 82 uses a log-log axis and the field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of 85236826359346144956638323529482240001. We see from figure 7.17 on page 82 that our C++ versions are not as fast as our GMP version. The difference appears consistent. We see that we get the best performance from our C based GMP code. The next experiment uses input sizes that range from 2^1 to 2^{20} and base case sizes 32 and 64. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU. We compile all implementations with option -o2 to obtain further optimization.

The figure 7.18 on page 83 uses a log-log axis and the field characteristic is labeled P2 in the table 7.1 on page 64 and has a value of
85236826359346144956638323529482240001.

We see from figure 7.18 on page 83 that our C++ versions are not as fast as our GMP version. The difference appears consistent. We see that we get the best performance from our C based GMP code.

Figure 7.16: BPF C vs C++, $K = 2$ $P2$ and $K = 4$ $P2$



(a) Six-Step

(b) Six-Step

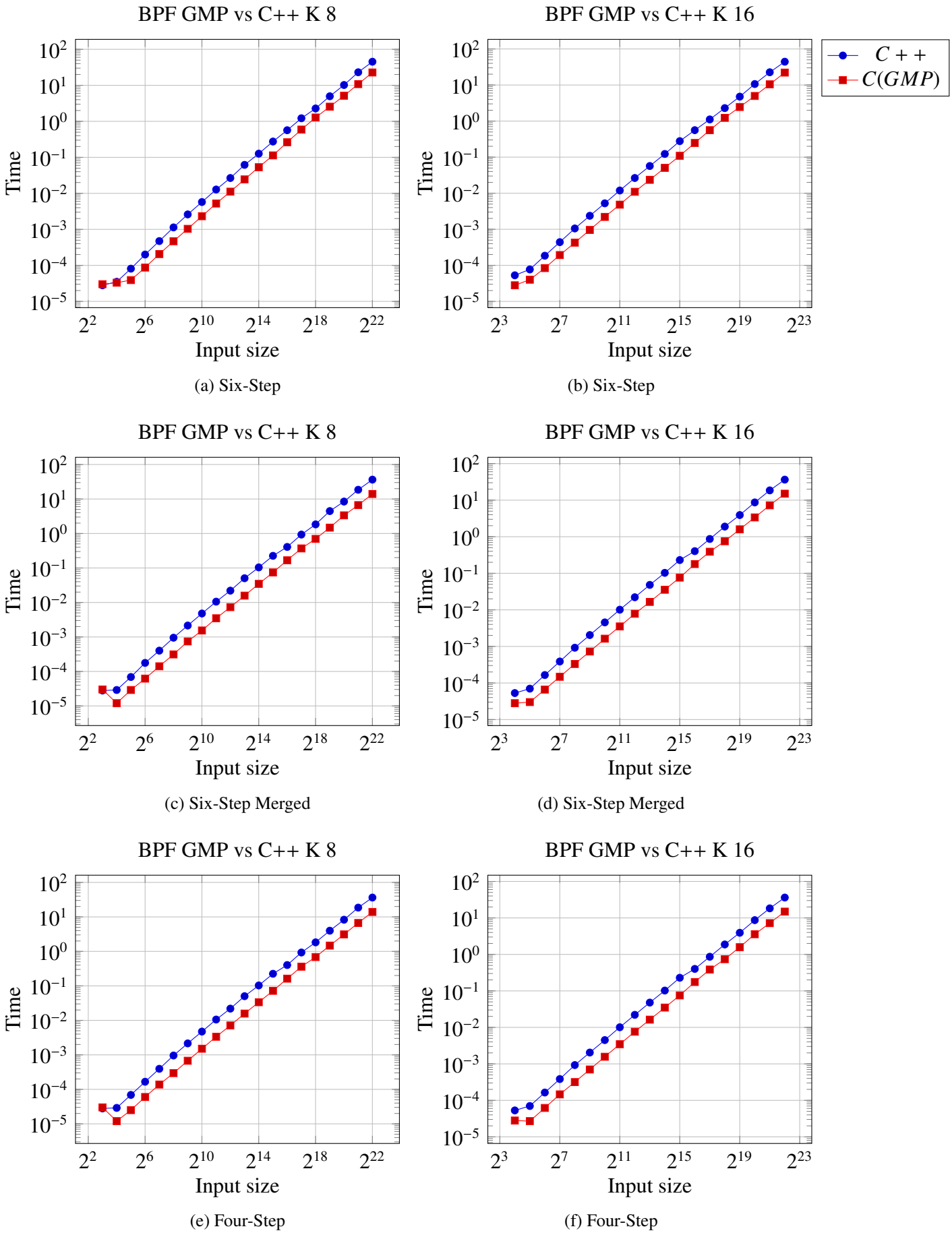
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.17: BPF C vs C++,$K = 8$ P2 and $K = 16$ $P2$



(a) Six-Step

(b) Six-Step

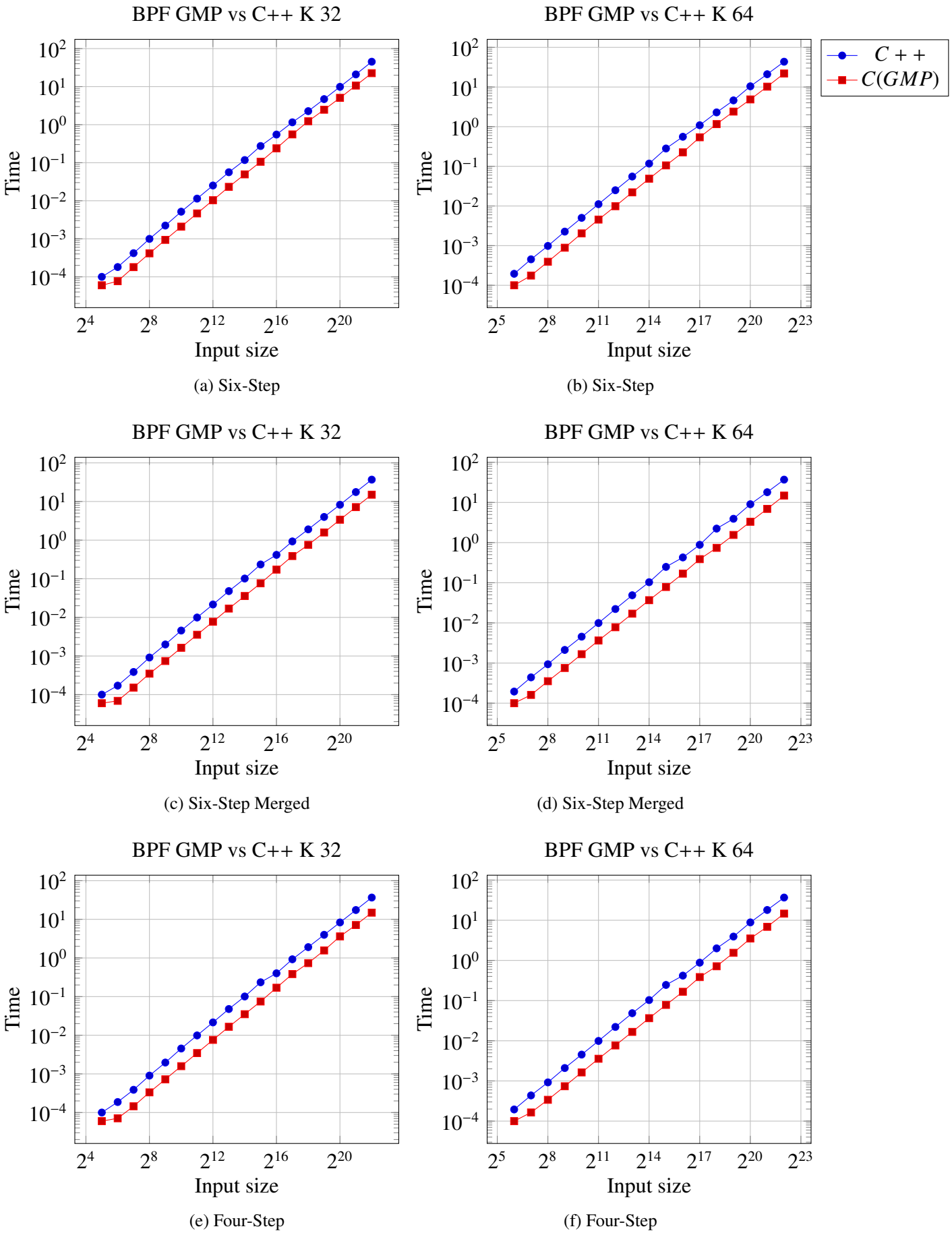
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.18: BPF C vs C++,$K = 32$ $P2$ and $K = 64$ $P2$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

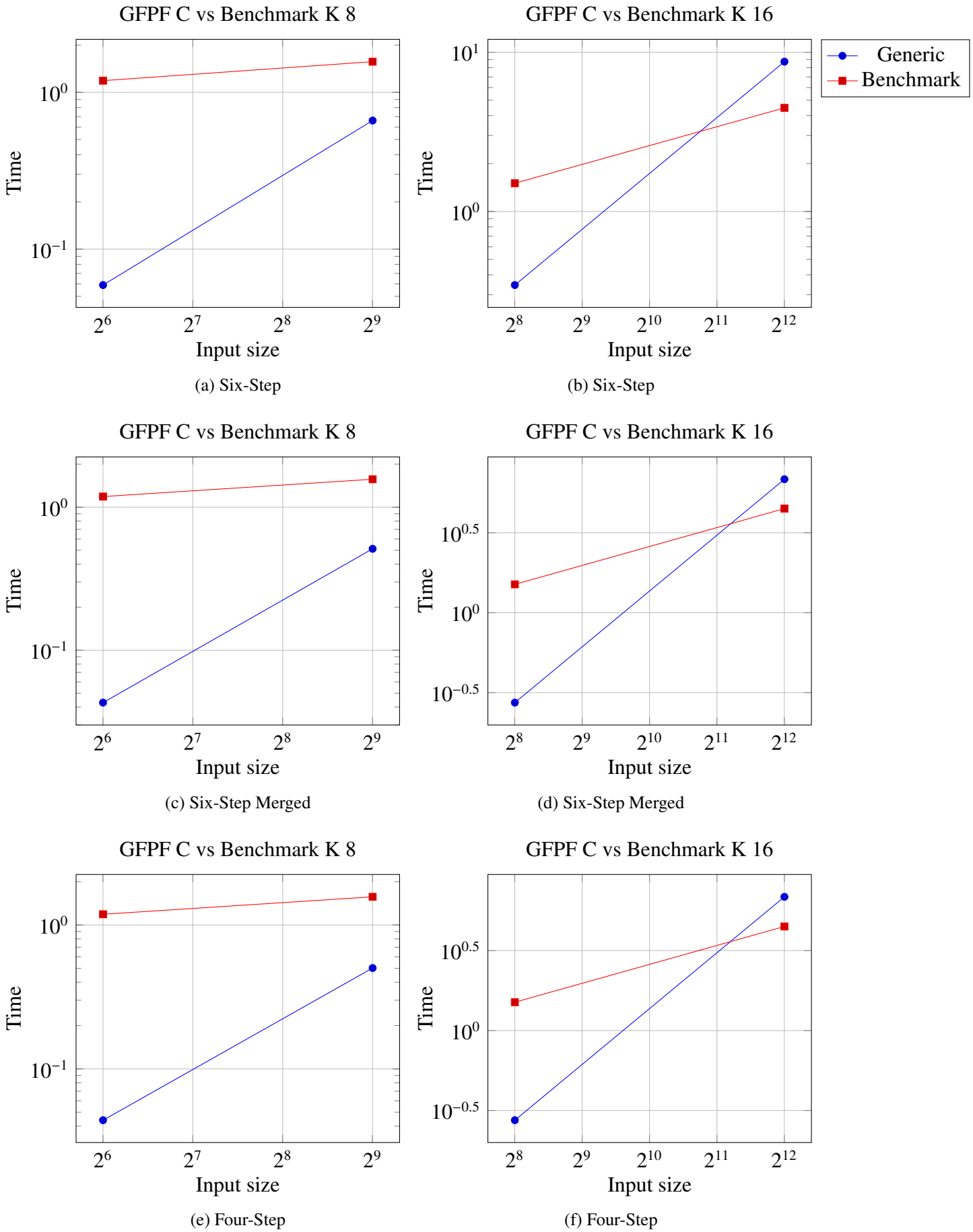
7.8 Generalized Fermat prime field C serial benchmarks

In this section we compare the C versions of the generic FFTs over a generalized Fermat prime field against the FFT specifically designed for generalized Fermat prime fields from [24] in a serial experiment. The field characteristics change based on the input size and radix. The primes range from P3 to P6 from the table 7.1 on page 64. The experiment uses input sizes that range from K^2 to K^3 for $K = 8$, $K = 16$, $K = 32$, and $K = 64$. The generic FFTs all use the same prime as the one required by the specialized FFT. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization.

The graphs from figure 7.19 on page 85 use a log-log axis. It has input sizes that range from K^2 to K^3 for $K = 8$, $K = 16$. When $K = 8$, $P3 = 864691128455137280^4 + 1$ and when $K = 16$, $P4 = 720576490135093248^8 + 1$. Looking at the figure 7.19 on 85 we can see that the generic code is slower when $K = 8$ and $e = 2$ when $K = 8$ and $e = 3$. However, it performs better when $K = 16$ and $e = 3$ then when $K = 16$ and $e = 2$ in this serial experiment.

The graphs from figure 7.20 on page 86 use a log-log axis. The experiment has input sizes that range from K^2 to K^3 for $K = 32$, $K = 64$.When $K = 32$, $P5 = 324294357542764544^{16} + 1$ and when $K = 64$, $P6 = 324259173170806784^{32} + 1$. Serially, when looking at the figure 7.20 on 86 we can see that the generic code is close when $K = 32$ and $e = 2$ when $K = 64$ and $e = 2$. However, it doesn't keep up to the specialized benchmark FFT when $K = 32$ and $e = 3$ and $K = 64$ and $e = 3$ in this serial experiment.

Figure 7.19: GFPF vs benchmark $K = 8$ $P3$ and $K = 16$ $P4$



(a) Six-Step

(b) Six-Step

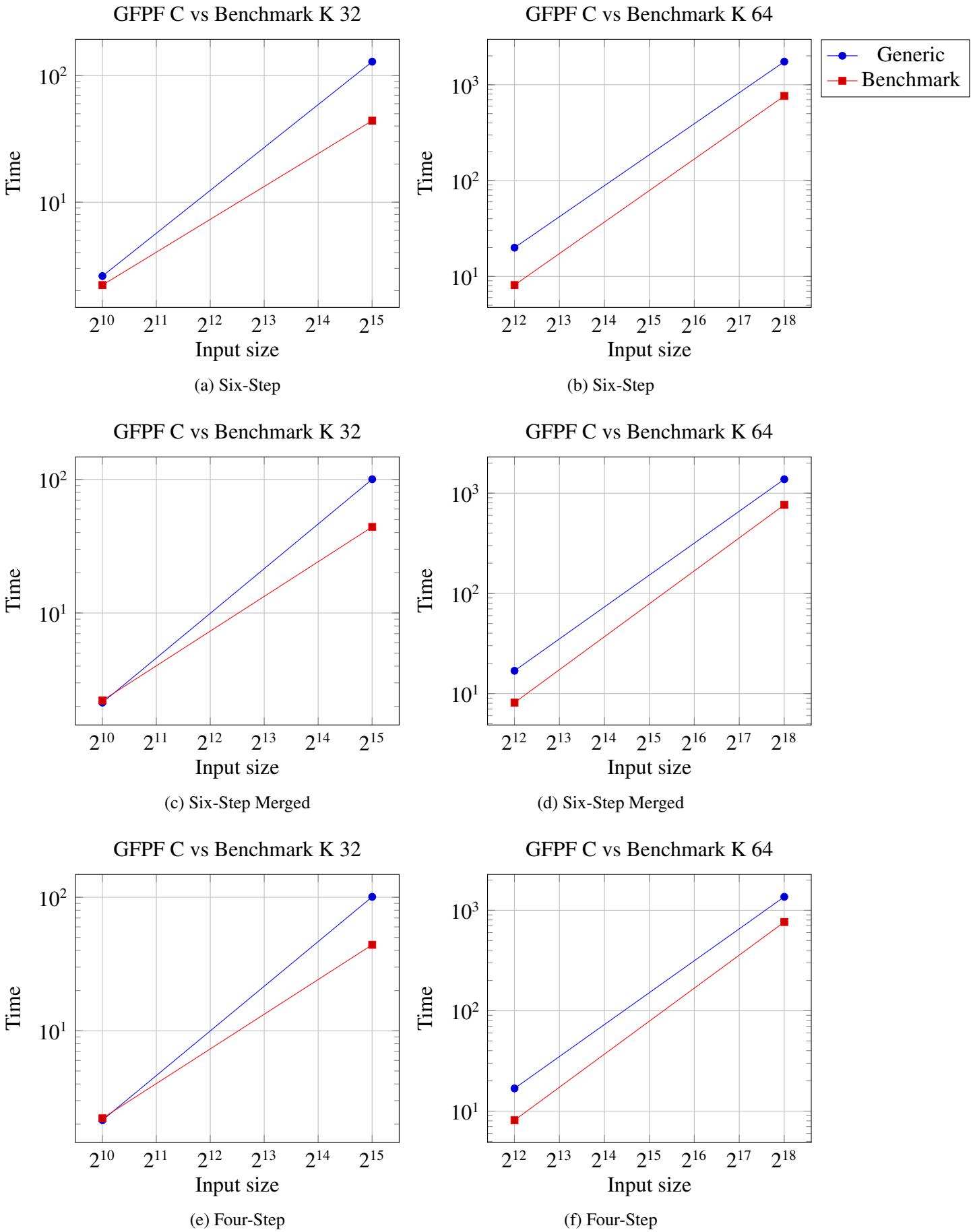
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.20: GFPF vs benchmark $K = 32$ $P5$ and $K = 64$ $P6$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

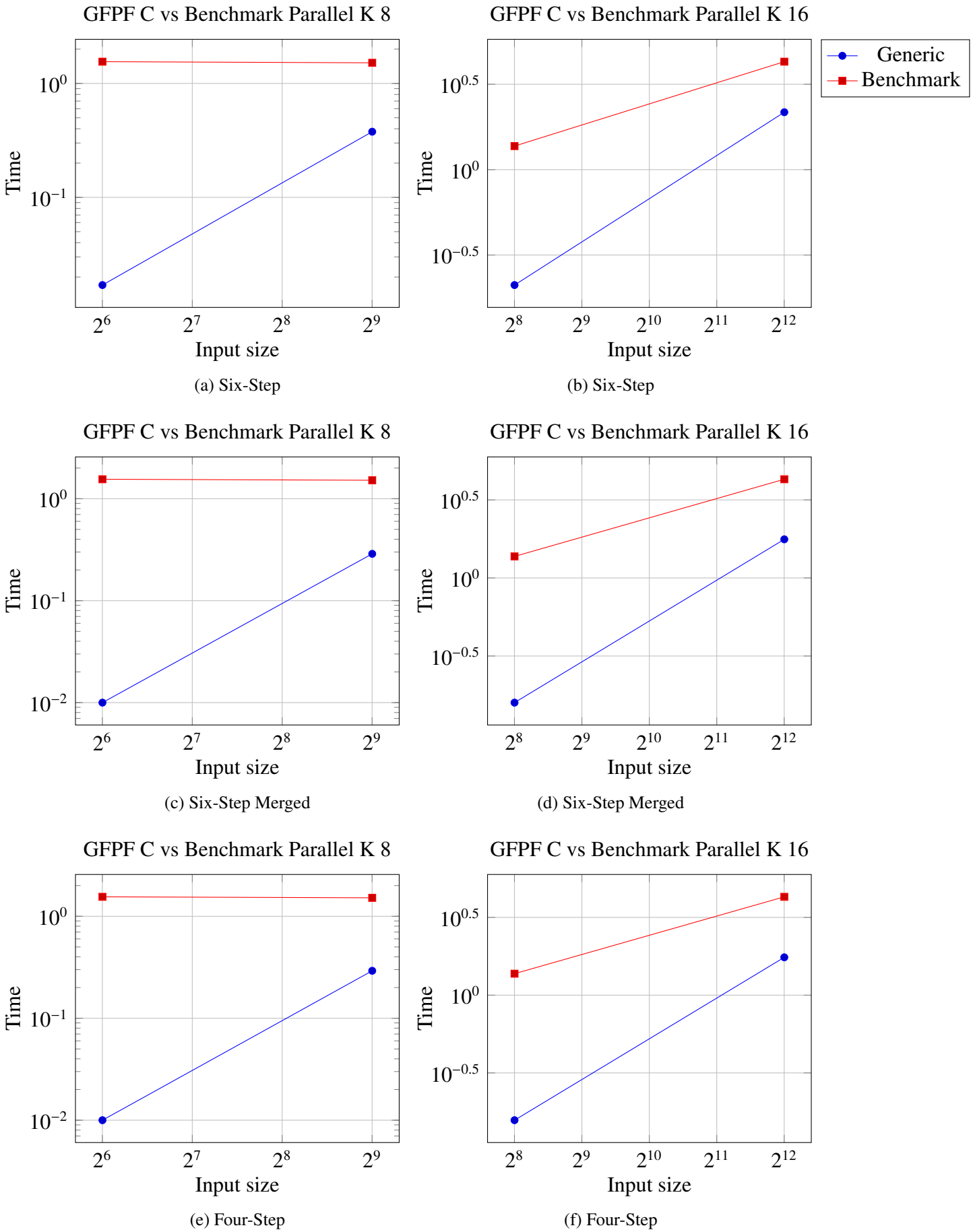
7.9 Generalized Fermat prime field C parallel benchmarks

In this section we compare the C versions of the generic FFTs over a generalized Fermat prime field against the FFT specifically designed for generalized Fermat prime fields from [24] in a parallel experiment. The field characteristics change based on the input size and radix. The primes range from P3 to P6 from the table 7.1 on page 64. The experiment uses input sizes that range from K^2 to K^3 for $K = 8$, $K = 16$, $K = 32$, and $K = 64$. The generic FFTs all use the same prime as the one required by the specialized FFT. We run our serial experiment on a single core of a 2.66 GHz Intel Xeon CPU and the parallel experiment on a 2.66 GHz Intel Xeon CPU but with 12 cores. We compile all implementations with option -o2 to obtain further optimization.

The graphs from figure 7.21 on page 88 use a log-log axis. It has input sizes that range from K^2 to K^3 for $K = 8$, $K = 16$. When $K = 8$, $P3 = 8646911284551372804 + 1$ and when $K = 16$, $P4 = 720576490135093248^8 + 1$. Looking at the figure 7.21 on 88 we can see that the generic code is faster during this parallel experiment.

The graphs from figure 7.22 on page 89 use a log-log axis. The experiment has input sizes that range from K^2 to K^3 for $K = 32$, $K = 64$.When $K = 32$, $P5 = 3242943575427645^{16} + 1$ and when $K = 64$, $P6 = 32425917317080678^{32} + 1$. Looking at the figure 7.22 on 89 we can see that the generic code is close when $K = 32$ and $e = 2$ when $K = 64$ and $e = 2$. However, it doesn't keep up to the specialized benchmark FFT when $K = 32$ and $e = 3$ and $K = 64$ and $e = 3$ in this parallel experiment.

Figure 7.21: GFPF vs benchmark parallel $K = 8$ $P3$ and $K = 16$ $P4$



(a) Six-Step

(b) Six-Step

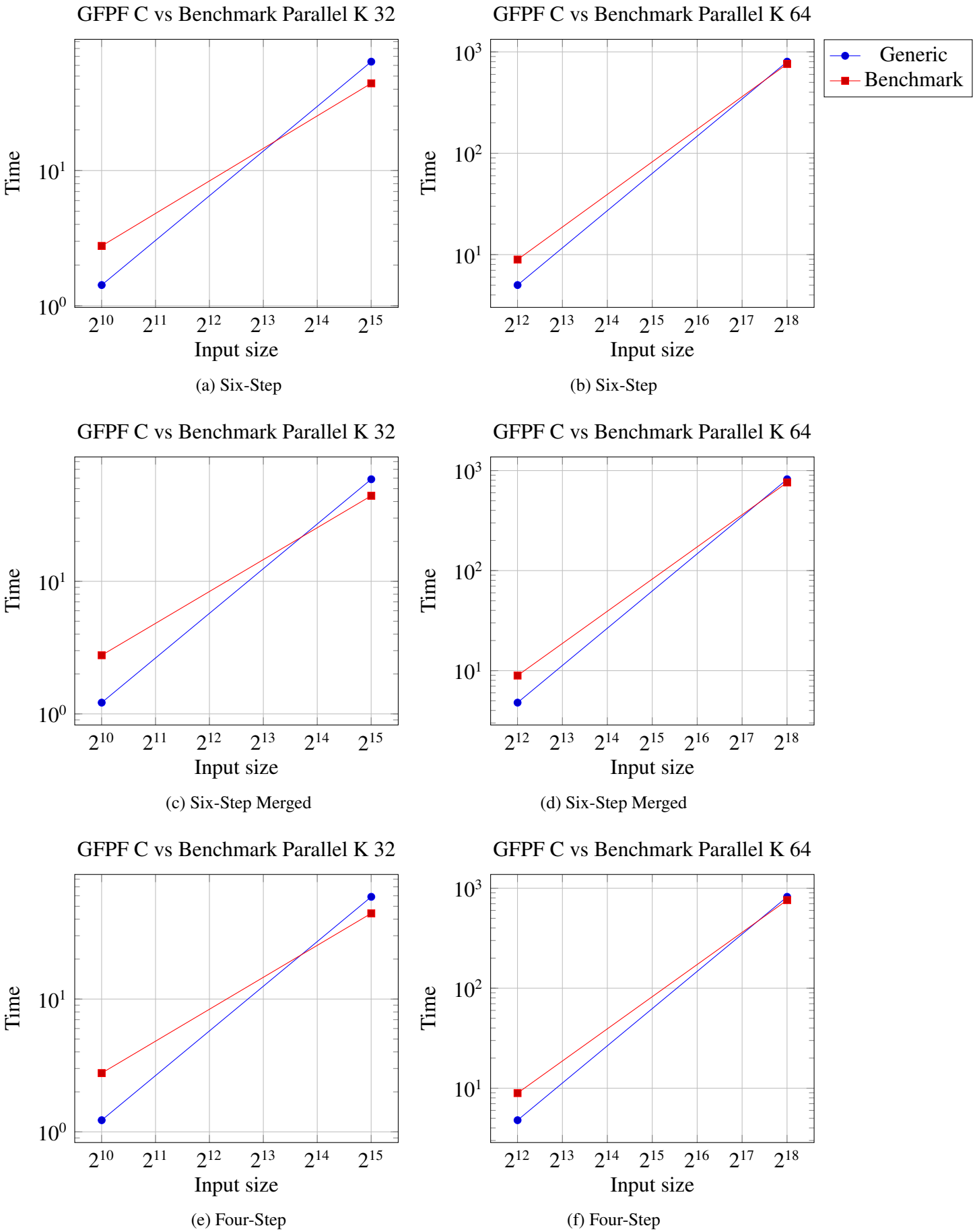
(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Figure 7.22: GFPF vs benchmark parallel $K = 32$ $P5$ and $K = 64$ $P6$



(a) Six-Step

(b) Six-Step

(c) Six-Step Merged

(d) Six-Step Merged

(e) Four-Step

(f) Four-Step

Table 7.2: Cachegrind Summary Legend

I refs	I cache reads
I1 misses	I1 cache read misses
LLi misses	LL cache instruction read misses
D refs	D cache reads
D1 misses	D1 cache read misses
LLd misses	L2 cache data read misses
LL ref	D cache writes
LL misses	D1 cache write misses

Table 7.3: Six-Step FFT Cachegrind Summary K=32

I refs:	9,747,963	
I1 misses:	13,769	
LLi misses:	2,881	
I1 miss rate:	0.14%	
LLi miss rate:	0.03%	
D refs:	4,020,200	(2,578,150 rd + 1,442,050 wr)
D1 misses:	17,312	(12,627 rd + 4,685 wr)
LLd misses:	8,447	(4,697 rd + 3,750 wr)
D1 miss rate:	0.4 %	(0.5% + 0.3%)
LLd miss rate:	0.2%	(0.2% + 0.3%)
LL refs:	31,081	(26,396 rd + 4,685 wr)
LL misses:	11,328	(7,578 rd + 3,750 wr)
LL miss rate:	0.1%	(0.1% + 0.3%)

7.10 Memory Profile

Tables 7.3 , 7.4 , and 7.5 on pages 90 and 91 are the summary statistics of a cache-miss profiler called Cachegrind [8]. Cache accesses for instructions come first, followed by cache accesses for data, followed by combined instruction and data figures for the L2 cache.

Table 7.4: Six-Step Merged FFT Cachegrind Summary K=32

I refs:	9,747,963	
I1 misses:	13,769	
LLi misses:	2,881	
I1 miss rate:	0.14%	
LLi miss rate:	0.03%	
D refs:	4,020,200	(2,578,150 rd + 1,442,050 wr)
D1 misses:	17,312	(12,627 rd + 4,685 wr)
LLd misses:	8,447	(4,697 rd + 3,750 wr)
D1 miss rate:	0.4%	(0.5% + 0.3%)
LLd miss rate:	0.2%	(0.2% + 0.3%)
LL refs:	31,081	(26,396 rd + 4,685 wr)
LL misses:	11,328	(7,578 rd + 3,750 wr)
LL miss rate:	0.1%	(0.1% + 0.3%)

Table 7.5: Four Step FFT Cachegrind Summary K=32

I refs:	9,747,963	
I1 misses:	13,769	
LLi misses:	2,881	
I1 miss rate:	0.14%	
LLi miss rate:	0.03%	
D refs:	4,020,200	(2,578,150 rd + 1,442,050 wr)
D1 misses:	17,312	(12,627 rd + 4,685 wr)
LLd misses:	8,447	(4,697 rd + 3,750 wr)
D1 miss rate:	0.4%	(0.5% + 0.3%)
LLd miss rate:	0.2%	(0.2% + 0.3%)
LL refs:	31,081	(26,396 rd + 4,685 wr)
LL misses:	11,328	(7,578 rd + 3,750 wr)
LL miss rate:	0.1%	(0.1% + 0.3%)

7.11 Observations

We found that the Six-Step explicit FFT realized the most speedup. Serially, the Six-Step FFTs couldn't compete with the serial Four-Step FFTs. We found that the Four-Step Loop-merged variant performed better than both the Six-Step explicit and loop merged versions. In terms of parallelism, the Six-Step explicit FFT on average realized the most speedup.

Looking at the overall computation, the areas that benefit the most from parallelism are the areas with more work. First, the twiddle factor subprogram (consisting of point-wise multiplications of finite field elements) is the most work intensive area due to the multiplications followed by the base case kernels, followed by the stride permutations.

In terms of work intensity, we found the stride permutation subprogram to be lightest on work and we wanted to see what kind of parallel speedup we might achieve by using the loop merging technique to push the stride permutations into the other loops for each recursion step during a computation.

Unfortunately, the act of loop merging introduces an additional copy back loop which reduces the overall effectiveness of the loop merge by adding another pass through the data. We found that loop merging adds another level of complexity to the code as well as the parallel computation. Also, it introduces large two power stride memory accesses in the loop merged versions.

The difference in performance between our C and C++ versions is not inconsequential. The additional weight of the pointers and the pointer offset calculations are still not negligible on modern architectures. For competitive code, it is better to build specialized subprograms using C rather than C++. The difference in measured performance between the C vs C++ experiment is larger than the difference between the GMP vs C++.

We found that our C++ versions were not competitive against a high performance serial FFT. Nor were they competitive when compared to our own C versions. We found that we could get the best performance from our C code. We found the comparison between the C++ and GMP C implementations to be closer in performance. The GMP class performs memory management online, the manual states, "mpz_t variables represent integers using sign and magnitude, in space dynamically allocated and reallocated" [13].

The increased complexity of the loop merged versions made parallelism difficult to achieve using the parallel for loop construct. The effort taken to merge the loops was not proportional to the result and becomes hard to justify when the very nature of the Six-Step explicit stride permutations align the field elements in a less complicated and more cache friendly way for the FFTs that follow.

Finally, it is worthy to note that due to the blocked nature of our FFTs, a poor choice of block size for the base case can incur significant performance penalties and it was difficult to get accurate measurements for the smallest values for N for the generic FFTs.

Bibliography

[1] M. Asadi, A. Brandt, C. Chen, S. Covanov, M. Kazemi, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, Linxiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS), 2020. `www.bpaslib.org`.

[2] Richard P. Brent and Paul Zimmermann. Modern computer arithmetic (version 0.5.1). *CoRR*, abs/1004.4710, 2010.

[3] Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, and Marc Moreno Maza. Big prime field FFT on the GPU. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC*, pages 85–92, 2017.

[4] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[5] Intel Corporation. The itanium C++ application binary interface, 2020. `http://itanium-cxx-abi.github.io/cxx-abi`.

[6] Svyatoslav Covanov. Putting Fürer's Algorithm into practice. Technical report, ORCCA Lab, London, 2014.

[7] Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza, and Lin-Xiao Wang. Big prime field FFT on multi-core processors. In James H. Davenport, Dongming Wang, Manuel Kauers, and Russell J. Bradford, editors, *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, July 15-18, 2019*, pages 106–113. ACM, 2019.

[8] Valgrind developers. Valgrind user manual, 2020. `https://www.valgrind.org/docs/manual/manual.html`.

[9] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In Lougie Anderson and James Coplien, editors, *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996*, pages 306–323. ACM, 1996.

[10] Akpodigha Filatei, Xin Li, Marc Moreno Maza, and Éric Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings*, pages 93–100, 2006.

[11] Franz Franchetti and Markus Püschel. FFT (fast Fourier transform). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 658–671. Springer, 2011.

[12] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[13] "Torbjörn Granlund and the GMP development team". *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2020. `https://gmplib.org/`.

[14] David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2010, Munich, Germany, July 25-28, 2010, Proceedings*, pages 325–329, 2010.

[15] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996.

[16] Lingchuan Meng, Yevgen Voronenko, Jeremy R. Johnson, Marc Moreno Maza, Franz Franchetti, and Yuzhen Xie. Spiral-generated modular FFT algorithms. In *PASCO*, pages 169–170, 2010.

[17] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[18] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[19] C. Lu. R. Tolimieri, M. An. *Algorithms for Discrete Fourier Transform and Convolution Second Edition.* Springer, 1997.

[20] Victor Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comput.*, 20(4):363–397, 1995.

[21] Joris van der Hoeven. The truncated Fourier transform and applications. In Jaime Gutierrez, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC 2004, Santander, Spain, July 4-7, 2004, Proceedings*, pages 290–296. ACM, 2004.

[22] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[23] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.

[24] Lin-Xiao Wang. Putting Fürer's algorithm into practice with the BPAS library. Master's thesis, University of Western Ontario, London, ON, Canada, 2018.

Curriculum Vitae

Name:	Colin Costello
Post-Secondary Education and Degrees:	The University of Western Ontario Major Specialization in Computer Science, Minor in Software Engineering University of Western Ontario London, ON 2014 - 2018 B.Sc.
Honours and Awards:	Dean's List Award 2018
Related Work Experience:	Teaching Assistant The University of Western Ontario 2018 - 2019

Publications: