Daisuke Takahashi

Fast Fourier Transform Algorithms for Parallel Computers

Springer

High-Performance Computing Series

Volume 2

Series Editor

Satoshi Matsuoka, RIKEN Center for Computational Science, Kobe, Hyogo, Japan

The series publishes authored monographs, textbooks, and edited state-of-the-art collections covering the whole spectrum of technologies for supercomputers and high-performance computing, and computational science and engineering enabled by high-performance computing series (HPC).

Areas of interest include, but are not limited to, relevant topics on HPC:

- Advanced algorithms for HPC
- Large-scale applications involving massive-scale HPC
- Innovative HPC processor and machine architectures
- High-performance / low-latency system interconnects
- HPC in the Cloud
- Data science / Big Data / visualization driven by HPC
- Convergence of machine learning / artificial intelligence with HPC
- Performance modeling, measurement, and tools for HPC
- Programing languages, compilers, and runtime systems for HPC
- Operating system and other system software for HPC
- HPC systems operations and management

More information about this series at http://www.springer.com/series/16381

Daisuke Takahashi

Fast Fourier Transform Algorithms for Parallel Computers

Springer

Daisuke Takahashi
University of Tsukuba
Tsukuba, Japan

ISSN 2662-3420 ISSN 2662-3439 (electronic)
High-Performance Computing Series
ISBN 978-981-13-9964-0 ISBN 978-981-13-9965-7 (eBook)
https://doi.org/10.1007/978-981-13-9965-7

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

The fast Fourier transform (FFT) is an efficient implementation of the discrete Fourier transform (DFT). The FFT is widely used in numerous applications in engineering, science, and mathematics. This book is an introduction to the basis of the FFT and its implementation in parallel computing. Parallel computation is becoming indispensable in solving the large-scale problems that arise in a wide variety of applications. Since there are many excellent books on FFT, this book focuses on the implementation details of FFTs for parallel computers. This book provides a thorough and detailed explanation of FFTs for parallel computers. The algorithms are presented in pseudocode, and a complexity analysis is provided.

The performance of parallel supercomputers is steadily improving, and it is expected that a massively parallel system with more than hundreds of thousands of compute nodes equipped with manycore processors and accelerators will be exascale supercomputers in the near future. This book also provides up-to-date computational techniques relevant to the FFT in state-of-the-art parallel computers.

This book is designed for graduate students, faculty, engineers, and scientists in the field. The design of this book intends for readers who have some knowledge about DFT and parallel computing. For several implementations of FFTs described in this book, you can download the source code from www.ffte.jp.

This book is organized as follows. Chapter 2 introduces the definition of the DFT and the basic idea of the FFT. Chapter 3 explains mixed-radix FFT algorithms. Chapter 4 describes split-radix FFT algorithms. Chapter 5 explains multidimensional FFT algorithms. Chapter 6 presents high-performance FFT algorithms. Chapter 7 explains parallel FFT algorithms for shared-memory parallel computers. Finally, Chap. 8 describes parallel FFT algorithms for distributed-memory parallel computers.

I express appreciation to all those who helped in the preparation of this book.

Tsukuba, Japan
March 2019

Daisuke Takahashi

Contents

Chapter 1
Introduction

Abstract The fast Fourier transform (FFT) is an efficient implementation of the discrete Fourier transform (DFT). The FFT is widely used in numerous applications in engineering, science, and mathematics. This chapter describes the history of the FFT briefly and presents an introduction to this book.

Keywords Discrete Fourier transform (DFT) · Fast Fourier transform (FFT) · Parallel processing

The fast Fourier transform (FFT) is a fast algorithm for computing the discrete Fourier transform (DFT).

The fast algorithm for DFT can be traced back to Gauss's unpublished work in 1805 [11]. Since the paper by Cooley and Tukey [6] was published in 1965, the FFT has become to be widely known. Then, Gentleman and Sande presented a variant of the Cooley-Tukey FFT algorithm [10]. In the Cooley-Tukey FFT algorithm, the input data is overwritten with the output data (i.e., in-place algorithm), but bit-reversal permutation is required. It is also possible to construct an out-of-place algorithm that stores input data and output data in separate arrays. Stockham algorithm [5] is known as an out-of-place algorithm and it does not require bit-reversal permutation. Bergland proposed an FFT algorithm for real-valued series [4]. Yavne [18] presented a method that is currently known as the split-radix FFT algorithm [7]. Singleton proposed a mixed-radix FFT algorithm [16]. As FFT algorithms of a different approach from the Cooley-Tukey FFT algorithm, Rader proposed an FFT algorithm that computes the DFTs when the number of data samples is prime [15]. Kolba and Parks proposed a prime factor FFT algorithm (PFA) [13]. Winograd extended Rader's algorithm and proposed a Winograd Fourier transform algorithm (WFTA) that can be applied to the DFTs of the powers of prime numbers [17]. Bailey proposed a four-step FFT algorithm and a six-step FFT algorithm [2]. Johnson and Frigo proposed a modified split-radix FFT algorithm [12], which is known as the FFT algorithm with the lowest number of arithmetic operations.

As early studies of parallel FFTs, Pease proposed an adaptation of the FFT for parallel processing [14]. Moreover, Ackins [1] implemented an FFT for the ILLIAC IV parallel computer [3]. Since then, various parallel FFT algorithms and

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_1

implementations have been proposed. Frigo and Johnson developed the FFTW (The fastest Fourier transform in the west), which is known as the fastest free software implementation of the FFT [8, 9].

Examples of FFT applications in the field of science are the following:

- Solving partial differential equations,
- Convolution and correlation calculations, and
- Density functional theory in first principles calculations.

Examples of FFT applications in the field of engineering are the following:

- Spectrum analyzers,
- Image processing, for example, in CT scanning and MRI, and
- Modulation and demodulation processing in orthogonal frequency multiplex modulation (OFDM) used in wireless LAN and terrestrial digital radio and television broadcasting.

The rest of this book is organized as follows. Chapter 2 introduces the definition of the DFT and the basic idea of the FFT. Chapter 3 explains mixed-radix FFT algorithms. Chapter 4 describes split-radix FFT algorithms. Chapter 5 explains multidimensional FFT algorithms. Chapter 6 presents high-performance FFT algorithms. Chapter 7 explains parallel FFT algorithms for shared-memory parallel computers. Finally, Chap. 8 describes parallel FFT algorithms for distributed-memory parallel computers. Performance results of parallel FFT algorithms on parallel computers are also presented.

References

1. Ackins, G.M.: Fast Fourier transform via ILLIAC IV. Illiac IV Document 198, University of Illinois, Urbana (1968)
2. Bailey, D.H.: FFTs in external or hierarchical memory. J. Supercomput. **4**, 23–35 (1990)
3. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A.: The ILLIAC IV computer. IEEE Trans. Comput. **C-17**, 746–757 (1968)
4. Bergland, G.D.: A fast Fourier transform algorithm for real-valued series. Commun. ACM **11**, 703–710 (1968)
5. Cochran, W.T., Cooley, J.W., Favin, D.L., Helms, H.D., Kaenel, R.A., Lang, W.W., Maling, G.C., Nelson, D.E., Rader, C.M., Welch, P.D.: What is the fast Fourier transform? IEEE Trans. Audio Electroacoust. **15**, 45–55 (1967)
6. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)
7. Duhamel, P., Hollmann, H.: Split radix FFT algorithm. Electron. Lett. **20**, 14–16 (1984)
8. Frigo, M., Johnson, S.G.: FFTW: an adaptive software architecture for the FFT. In: Proceedings of 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '98), vol. 3, pp. 1381–1384 (1998)
9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**, 216–231 (2005)
10. Gentleman, W.M., Sande, G.: Fast Fourier transforms: for fun and profit. In: Proceedings of AFIPS '66 Fall Joint Computer Conference, pp. 563–578 (1966)

11. Heideman, M.T., Johnson, D.H., Burrus, C.S.: Gauss and the history of the fast Fourier transform. IEEE ASSP Mag. **1**, 14–21 (1984)
12. Johnson, S.G., Frigo, M.: A modified split-radix FFT with fewer arithmetic operations. IEEE Trans. Signal Process. **55**, 111–119 (2007)
13. Kolba, D.P., Parks, T.W.: A prime factor FFT algorithm using high-speed convolution. IEEE Trans. Acoust. Speech Signal Process **ASSP-25**, 281–294 (1977)
14. Pease, M.C.: An adaptation of the fast Fourier transform for parallel processing. J. ACM **15**, 252–264 (1968)
15. Rader, C.M.: Discrete Fourier transforms when the number of data samples is prime. Proc. IEEE **56**, 1107–1108 (1968)
16. Singleton, R.C.: An algorithm for computing the mixed radix fast Fourier transform. IEEE Trans. Audio Electroacoust. **17**, 93–103 (1969)
17. Winograd, S.: On computing the discrete Fourier transform. Math. Comput. **32**, 175–199 (1978)
18. Yavne, R.: An economical method for calculating the discrete Fourier transform. In: Proceedings of AFIPS '68 Fall Joint Computer Conference, Part I, pp. 115–125 (1968)

Chapter 2
Fast Fourier Transform

Abstract This chapter introduces the definition of the DFT and the basic idea of the FFT. Then, the Cooley–Tukey FFT algorithm, bit-reversal permutation, and Stockham FFT algorithm are explained. Finally, FFT algorithm for real data is described.

Keywords Decimation-in-time · Decimation-in-frequency · Butterfly operation

2.1 Definitions of DFT

The DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1. \tag{2.1}$$

Moreover, the inverse DFT is given by

$$x(j) = \frac{1}{n}\sum_{k=0}^{n-1} y(k)\omega_n^{-jk}, \quad 0 \le j \le n-1, \tag{2.2}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

2.2 Basic Idea of FFT

In Eq. (2.1), for example, when $n = 4$, the DFT can be calculated as follows:

$$\begin{aligned} y(0) &= x(0)\omega^0 + x(1)\omega^0 + x(2)\omega^0 + x(3)\omega^0, \\ y(1) &= x(0)\omega^0 + x(1)\omega^1 + x(2)\omega^2 + x(3)\omega^3, \\ y(2) &= x(0)\omega^0 + x(1)\omega^2 + x(2)\omega^4 + x(3)\omega^6, \\ y(3) &= x(0)\omega^0 + x(1)\omega^3 + x(2)\omega^6 + x(3)\omega^9. \end{aligned} \tag{2.3}$$

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_2

Equation (2.3) can be expressed more simply in the form of a matrix–vector product using a matrix as follows:

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{2.4}$$

In Eq. (2.1), $x(j)$, $y(k)$, and ω are complex values. Therefore, in order to calculate the matrix–vector product of Eq. (2.4), n^2 complex multiplications and $n(n-1)$ complex additions are required.

When we use the relation $\omega_n^{jk} = \omega_n^{jk \bmod n}$, Eq. (2.4) can be written as follows:

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^0 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{2.5}$$

The following decomposition of the matrix allows the number of complex multiplications to be reduced:

$$\begin{bmatrix} y(0) \\ y(2) \\ y(1) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & \omega^0 & 0 & 0 \\ 1 & \omega^2 & 0 & 0 \\ 0 & 0 & 1 & \omega^1 \\ 0 & 0 & 1 & \omega^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & \omega^0 & 0 \\ 0 & 1 & 0 & \omega^0 \\ 1 & 0 & \omega^2 & 0 \\ 0 & 1 & 0 & \omega^2 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{2.6}$$

Performing this decomposition recursively, the number of arithmetic operations can be reduced to $O(n \log n)$.

Let us consider a generalization of the decomposition. When n is divisible by two in Eq. (2.1), by decomposing the n-point data into the first half of the $n/2$-point data and the second half of the $n/2$-point data, the n-point DFT can be expressed as follows:

$$\begin{aligned} y(k) &= \sum_{j=0}^{n/2-1} \left\{ x(j)\omega_n^{jk} + x(j+n/2)\omega_n^{(j+n/2)k} \right\} \\ &= \sum_{j=0}^{n/2-1} \left\{ x(j) + x(j+n/2)\omega_n^{(n/2)k} \right\} \omega_n^{jk}, \quad 0 \le k \le n-1. \end{aligned} \tag{2.7}$$

Here, $\omega_n^{(n/2)k}$ in Eq. (2.7) can be expressed in terms of $\omega_n = e^{-2\pi i/n}$, as follows:

$$\omega_n^{(n/2)k} = e^{-\pi ik} = \begin{cases} 1 & \text{if } k \text{ is even} \\ -1 & \text{if } k \text{ is odd} \end{cases}. \tag{2.8}$$

Therefore, if k is divided into even and odd numbers, the n-point DFT can be decomposed into two $n/2$-point DFTs, as follows:

$$\begin{aligned} y(2k) &= \sum_{j=0}^{n/2-1} \{x(j) + x(j+n/2)\}\, \omega_n^{2jk} \\ &= \sum_{j=0}^{n/2-1} \{x(j) + x(j+n/2)\}\, \omega_{n/2}^{jk}, \quad 0 \le k \le n/2-1, \qquad (2.9) \\ y(2k+1) &= \sum_{j=0}^{n/2-1} \{x(j) - x(j+n/2)\} \omega_n^{j(2k+1)} \\ &= \sum_{j=0}^{n/2-1} \left\{\{x(j) - x(j+n/2)\}\omega_n^j\right\} \omega_{n/2}^{jk}, \quad 0 \le k \le n/2-1. (2.10) \end{aligned}$$

The $n/2$-point DFT in Eqs. (2.9) and (2.10) is calculated by $n^2/4$ complex multiplications and $(n/2)(n/2-1)$ complex additions. By this decomposition, the arithmetic operations are reduced to approximately 1/2. Furthermore, when n is a power of two, by recursively performing this decomposition, the n-point DFT can finally be reduced to a two-point DFT and the arithmetic operations can be reduced to $O(n \log n)$.

There are two methods for FFT decomposition, namely, the decimation-in-time method and the decimation-in-frequency method. In the decimation-in-time method, n-point data are decomposed into the even-numbered $n/2$-point data and the odd-numbered $n/2$-point data, whereas in the decimation-in-frequency method, n-point data are divided into the first half of the $n/2$-point data and the second half of the $n/2$-point data, as shown in Eq. (2.7). Both methods require the same number of arithmetic operations.

Listing 2.1 Decimation-in-frequency FFT routine by recursive call

```
1        recursive subroutine fft(x,temp,n)
2        implicit real*8 (a-h,o-z)
3        complex*16 x(*),temp(*)
4  !
5        if (n .le. 1) return
6  !
7        pi=4.0d0*atan(1.0d0)
8        px=-2.0d0*pi/dble(n)
9  !
10       do j=1,n/2
11         w=px*dble(j-1)
12         temp(j)=x(j)+x(j+n/2)
13         temp(j+n/2)=(x(j)-x(j+n/2))*dcmplx(cos(w),
               sin(w))
14       end do
15 !
16       call fft(temp,x,n/2)
17       call fft(temp(n/2+1),x,n/2)
```

```
!
      do j=1,n/2
        x(2*j-1)=temp(j)
        x(2*j)=temp(j+n/2)
      end do
      return
      end
```

By straightforwardly coding the concept behind Eqs. (2.9) and (2.10), we can write the decimation-in-frequency FFT routine by recursive call, as shown in Listing 2.1. In this routine, $\log_2 n$ recursive calls are made. Since $n/2$ complex multiplications and n complex additions and subtractions are performed on each call, the number of arithmetic operations is $O(n \log n)$. In this routine, the values of trigonometric functions are calculated by calling the functions `sin` and `cos` each time in the innermost loop, but these values are calculated in advance and are stored in the table for speeding up. In order to obtain the inverse FFT, we first invert the sign of `w=px*dble(j-1)` and calculate `w=-px*dble(j-1)` and then multiply the calculation result by $1/n$.

Listing 2.2 Cooley-Tukey FFT algorithm

```
      subroutine fft(x,n,m)
! Number of data n = 2**m
      implicit real*8 (a-h,o-z)
      complex*16 x(*),temp
!
      pi=4.0d0*atan(1.0d0)
!
      l=n
      do k=1,m
        px=-2.0d0*pi/dble(l)
        l=l/2
        do j=1,l
          w=px*dble(j-1)
          do i=j,n,l*2
            temp=x(i)-x(i+l)
            x(i)=x(i)+x(i+l)
            x(i+l)=temp*dcmplx(cos(w),sin(w))
          end do
        end do
      end do
! Rearrange output data in bit-reversed order
      j=1
      do i=1,n-1
        if (i .lt. j) then
          temp=x(i)
          x(i)=x(j)
          x(j)=temp
        end if
        k=n/2
   10   if (k .lt. j) then
          j=j-k
```

```
            k=k/2
            go to 10
          end if
          j=j+k
        end do
        return
        end
```

2.3 Cooley–Tukey FFT Algorithm

By rewriting the recursive call in the FFT routine of Listing 2.1 into a loop, the FFT algorithm of Listing 2.2, which is known as the Cooley–Tukey FFT algorithm [4], is derived. Note that the input data is overwritten with the output data (in-place). Figure 2.1 shows the data flow graph of a decimation-in-frequency Cooley–Tukey FFT. As shown in Fig. 2.1, the operation in each stage is composed of two data operations and does not affect other operations.

These two data operations are calculated as shown in Fig. 2.2, and the calculations can be written as follows:

$$\begin{aligned} X &= x + y, \\ Y &= (x - y)\omega^j. \end{aligned} \tag{2.11}$$

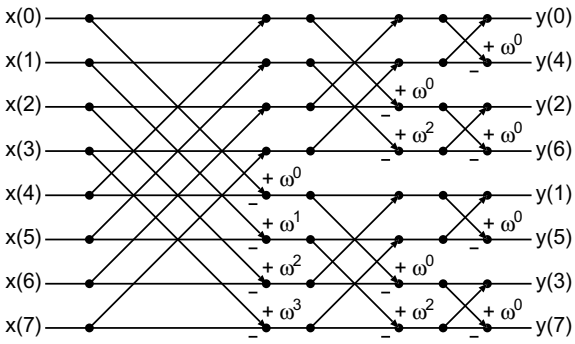

Fig. 2.1 Data flow graph of a decimation-in-frequency Cooley–Tukey FFT ($n = 8$)

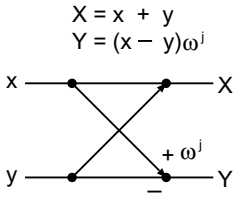

Fig. 2.2 Basic operation of decimation-in-frequency

Table 2.1 Bit-reversal permutation for $n = 8$

Input order	Binary notation	Bit reversal	Output order
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

This equation is the basic operation of decimation-in-frequency, or butterfly operation.

2.4 Bit-Reversal Permutation

In the Cooley–Tukey FFT algorithm, the input data is overwritten with the output data, and, as shown in Fig. 2.1, the order of the output data is by bit reversal. Table 2.1 shows the bit-reversal permutation for $n = 8$.

In order to make the order of the output data the same as the input data, it is necessary to rearrange the order obtained by bit reversal. Cache misses frequently occur because this permutation is not continuous access and the spatial locality is low.

Furthermore, the innermost loop `do i=j,n,l*2`, which computes the FFT, is a stride access of power of two. In order to obtain high performance, it is necessary to perform computation using as much as possible the data of one line (often 64 bytes or 128 bytes) loaded from the cache memory. However, in the case of a stride access of power of two, a situation occurs whereby only one data point among the data of one line can be used. Therefore, when considering hierarchical memory, the Cooley–Tukey FFT algorithm is not necessarily a preferable algorithm.

The FFT routine based on the recursive call shown in Listing 2.1 has the same effect as hierarchically cache blocking, so the performance often increases.

2.5 Stockham FFT Algorithm

The Cooley–Tukey FFT algorithm is an in-place algorithm, in which the input data is overwritten with the output data, but it is also possible to construct an out-of-place algorithm that stores input data and output data in separate arrays. Stockham FFT algorithm [3], which is known as an out-of-place algorithm, is shown in Fig. 2.3.

Fig. 2.3 Stockham FFT algorithm

$n = 2^p$, $X_0(j) = x(j)$, $0 \le j < n$, and $\omega_n = e^{-2\pi i/n}$
$l = n/2$; $m = 1$
do $t = 1, p$
  do $j = 0, l-1$
    do $k = 0, m-1$
      $c_0 = X_{t-1}(k + jm)$
      $c_1 = X_{t-1}(k + jm + lm)$
      $X_t(k + 2jm) = c_0 + c_1$
      $X_t(k + 2jm + m) = \omega_{2l}^{j}(c_0 - c_1)$
    end do
  end do
  $l = l/2$; $m = m * 2$
end do

With the Stockham FFT algorithm, since the input data cannot be overwritten with the output data, the required memory capacity is doubled as compared with the Cooley–Tukey FFT algorithm. However, the innermost loop is continuous access, and bit-reversal permutation is unnecessary. Therefore, the Stockham FFT algorithm is more advantageous than the Cooley–Tukey FFT algorithm from the viewpoint of adaptability to the hierarchical memory.

2.6 FFT Algorithm for Real Data

2.6.1 FFT of Two Real Data Simultaneously

When the input data of the DFT are real, two n-point real DFTs can be computed using an n-point complex DFT [2]. Perform complex FFT by putting one real data in the real part of complex input data and another real data in the imaginary part.

Let

$$x(j) = a(j) + ib(j), \quad 0 \le j \le n-1, \tag{2.12}$$

where $a(j)$ $(0 \le j \le n-1)$ and $b(j)$ $(0 \le j \le n-1)$ are two n-point real input data.

The n-point complex DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1. \tag{2.13}$$

We obtain the following output of complex DFT:

$$y(k) = c(k) + id(k), \quad 0 \le k \le n-1, \tag{2.14}$$

$$\overline{y}(n-k) = c(k) - id(k), \quad 0 \le k \le n-1, \tag{2.15}$$

where $c(k)$ $(0 \le k \le n-1)$ and $d(k)$ $(0 \le k \le n-1)$ are complex DFTs of $a(j)$ $(0 \le j \le n-1)$ and $b(j)$ $(0 \le j \le n-1)$, respectively.

Then, we obtain two n-point real DFTs from the n-point complex DFT as follows:

$$c(k) = \frac{1}{2}\{y(k) + \overline{y}(n-k)\}, \quad 0 \le k \le n-1, \tag{2.16}$$

$$d(k) = -\frac{i}{2}\{y(k) - \overline{y}(n-k)\}, \quad 0 \le k \le n-1. \tag{2.17}$$

2.6.2 n-Point Real FFT Using n/2-Point Complex FFT

When the input data of the DFT are real, an n-point real DFT can be computed using an $n/2$-point complex DFT [2].

Let

$$x(j) = a(2j) + ia(2j+1), \quad 0 \le j \le n/2-1, \tag{2.18}$$

where $a(j)$ $(0 \le j \le n-1)$ are n-point real input data.

The $n/2$-point complex DFT is given by

$$y(k) = \sum_{j=0}^{n/2-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n/2-1. \tag{2.19}$$

We obtain the n-point real DFT from the $n/2$-point complex DFT as follows:

$$b(k) = y(k) - \frac{1}{2}\{y(k) - \overline{y}(n/2-k)\}(1 + i\omega_n^k), \quad 1 \le k \le n/4-1, \tag{2.20}$$

$$\overline{b}(n/2-k) = \overline{y}(n/2-k) + \frac{1}{2}\{y(k) - \overline{y}(n/2-k)\}(1 + i\omega_n^k), \quad 1 \le k \le n/4-1, \tag{2.21}$$

$$b(0) = \mathrm{Re}\{y(0)\} + \mathrm{Im}\{y(0)\}, \tag{2.22}$$

$$b(n/2) = \mathrm{Re}\{y(0)\} - \mathrm{Im}\{y(0)\}, \tag{2.23}$$

$$b(n/4) = y(n/4), \tag{2.24}$$

where $b(j)$ $(0 \le j \le n/2)$ are $(n/2+1)$-point complex output data. Note that $b(k)$ $(n/2 < k \le n-1)$ can be reconstructed using the symmetry $b(k) = \overline{b}(n-k)$.

In the n-point real DFT, one-half of the DFT outputs are redundant. Bergland's real FFT algorithm [1] is based on this property. The complex-to-real FFT routine is shown in Listing 2.3. Note that the array `a(n)` is real input data, and the array `a(2,n/2+1)` is overwritten with the complex output data (in-place).

Listing 2.3 Complex-to-real FFT routine

```
      subroutine rfft(a,n)
      implicit real*8 (a-h,o-z)
      real*8 a(2,n/2+1)
!
      pi=4.0d0*atan(1.0d0)
      px=-2.0d0*pi/dble(n)
!
! n/2-point complex FFT
      call fft(a,n/2)
!
      temp=a(1,1)-a(2,1)
      a(1,1)=a(1,1)+a(2,1)
      a(2,1)=0.0d0
      a(2,n/4+1)=-a(2,n/4+1)
      a(1,n/2+1)=temp
      a(2,n/2+1)=0.0d0
      do i=2,n/4
        w=px*dble(i-1)
        ar=0.5d0*(a(1,i)-a(1,n/2-i+2))
        ai=0.5d0*(a(2,i)+a(2,n/2-i+2))
        wr=1.0d0-sin(w)
        wi=cos(w)
        temp=ar*wi+ai*wr
        ar=ar*wr-ai*wi
        ai=temp
        a(1,i)=a(1,i)-ar
        a(2,i)=a(2,i)-ai
        a(1,n/2-i+2)=a(1,n/2-i+2)+ar
        a(2,n/2-i+2)=a(2,n/2-i+2)-ai
      end do
      return
      end
```

References

1. Bergland, G.D.: A fast Fourier transform algorithm for real-valued series. Commun. ACM **11**, 703–710 (1968)
2. Brigham, E.O.: The Fast Fourier Transform and Its Applications. Prentice-Hall, Englewood Cliffs, NJ (1988)
3. Cochran, W.T., Cooley, J.W., Favin, D.L., Helms, H.D., Kaenel, R.A., Lang, W.W., Maling, G.C., Nelson, D.E., Rader, C.M., Welch, P.D.: What is the fast Fourier transform? IEEE Trans. Audio Electroacoust. **15**, 45–55 (1967)
4. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)

Chapter 3
Mixed-Radix FFT Algorithms

Abstract This chapter presents Mixed-Radix FFT Algorithms. First, two-dimensional formulation of DFT is given. Next, radix-3, 4, 5, and 8 FFT algorithms are described.

Keywords Two-dimensional formulation · Mixed-radix FFT algorithm

3.1 Two-Dimensional Formulation of DFT

In the FFT algorithms up to Chap. 2, we have assumed that the number of data in the discrete Fourier transform n is a power of two. In this section, we explain the FFT algorithm when this assumption is excluded. In order to derive the FFT algorithm for an arbitrary number of data points, let us first consider the case of $n = n_1 n_2$.

The DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1, \tag{3.1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n has factors n_1 and n_2 ($n = n_1 \times n_2$), then indices j and k in Eq. (3.1) can be expressed as follows:

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{3.2}$$

We can define x and y in Eq. (3.1) as two-dimensional arrays (in column-major order):

$$x(j) = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{3.3}$$

$$y(k) = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{3.4}$$

Substituting the indices j and k in Eq. (3.1) with those in Eq. (3.2) and using the relation $n = n_1 \times n_2$, we derive the following equation:

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_3

$$
\begin{aligned}
y(k_2, k_1) &= \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_n^{(j_1+j_2 n_1)(k_2+k_1 n_2)} \\
&= \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_n^{j_1 k_2} \omega_n^{j_1 k_1 n_2} \omega_n^{j_2 k_2 n_1} \omega_n^{j_2 k_1 n_1 n_2} \\
&= \sum_{j_1=0}^{n_1-1} \left[\sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \right] \omega_{n_1}^{j_1 k_1}. \qquad (3.5)
\end{aligned}
$$

Here, the relation $\omega_n^{n_1 n_2} = \omega_n^n = 1$ is used.

Equation (3.5) shows that the n-point DFT can be decomposed into an n_1-point DFT and an n_2-point DFT. In other words, even if n is not a power of two, the number of arithmetic operations can be reduced. If we decompose the n-point DFT into an $n_1 = n/2$-point DFT and an $n_2 = 2$-point DFT recursively, the decimation-in-frequency Cooley–Tukey FFT algorithm is derived. Moreover, if we decompose the n-point DFT into an $n_1 = 2$-point DFT and an $n_2 = n/2$-point DFT recursively, the decimation-in-time Cooley–Tukey FFT algorithm is derived.

The number of data points to which the DFT is finally reduced in the FFT decomposition is called the radix. In Chap. 2, we explained the case in which the radix is 2, but the FFT algorithm can be constructed in the same manner, even if the radix is 3, 5, or 7. An algorithm for the radix-3 FFT [2], a radix-6 FFT algorithm [4], and an FFT algorithm of radix-3, 6, and 12 have been proposed [6]. It is also possible to construct a mixed-radix FFT algorithm such that the radices are 2 and 4 [5, 7].

Prime factor FFT algorithms have been proposed [1, 3, 8]. In the prime factor FFT algorithm, n_1 and n_2 are coprime for $n = n_1 \times n_2$-point DFT. Winograd proposed a Winograd Fourier transform algorithm (WFTA) that can be applied to the DFTs of the powers of prime numbers [9].

3.2 Radix-3 FFT Algorithm

Let $n = 3^p$, $X_0(j) = x(j)$, $0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-3 FFT algorithm can be expressed as follows:

$l = n/3;\ m = 1$
do $t = 1,\ p$
  do $j = 0,\ l-1$
    do $k = 0,\ m-1$
      $c_0 = X_{t-1}(k + jm)$
      $c_1 = X_{t-1}(k + jm + lm)$
      $c_2 = X_{t-1}(k + jm + 2lm)$
      $d_0 = c_1 + c_2$
      $d_1 = c_0 - \frac{1}{2} d_0$
      $d_2 = -i \left(\sin \frac{\pi}{3} \right) (c_1 - c_2)$

$X_t(k+3jm) = c_0 + d_0$
$X_t(k+3jm+m) = \omega_{3l}^{j}(d_1 + d_2)$
$X_t(k+3jm+2m) = \omega_{3l}^{2j}(d_1 - d_2)$
end do
end do
$l = l/3;\ m = m*3$
end do

Here the variables c_0–c_2 and d_0–d_2 are temporary variables.

3.3 Radix-4 FFT Algorithm

Let $n = 4^p$, $X_0(j) = x(j),\ 0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-4 FFT algorithm can be expressed as follows:

$l = n/4;\ m = 1$
do $t = 1,\ p$
do $j = 0,\ l-1$
do $k = 0,\ m-1$
$c_0 = X_{t-1}(k+jm)$
$c_1 = X_{t-1}(k+jm+lm)$
$c_2 = X_{t-1}(k+jm+2lm)$
$c_3 = X_{t-1}(k+jm+3lm)$
$d_0 = c_0 + c_2$
$d_1 = c_0 - c_2$
$d_2 = c_1 + c_3$
$d_3 = -i(c_1 - c_3)$
$X_t(k+4jm) = d_0 + d_2$
$X_t(k+4jm+m) = \omega_{4l}^{j}(d_1 + d_3)$
$X_t(k+4jm+2m) = \omega_{4l}^{2j}(d_0 - d_2)$
$X_t(k+4jm+3m) = \omega_{4l}^{3j}(d_1 - d_3)$
end do
end do
$l = l/4;\ m = m*4$
end do

Here the variables c_0–c_3 and d_0–d_3 are temporary variables.

3.4 Radix-5 FFT Algorithm

Let $n = 5^p$, $X_0(j) = x(j),\ 0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-5 FFT algorithm can be expressed as follows:

$l = n/5;\ m = 1$
do $t = 1,\ p$
do $j = 0,\ l-1$

do $k = 0,\ m-1$
$c_0 = X_{t-1}(k + jm)$
$c_1 = X_{t-1}(k + jm + lm)$
$c_2 = X_{t-1}(k + jm + 2lm)$
$c_3 = X_{t-1}(k + jm + 3lm)$
$c_4 = X_{t-1}(k + jm + 4lm)$
$d_0 = c_1 + c_4$
$d_1 = c_2 + c_3$
$d_2 = \left(\sin\dfrac{2\pi}{5}\right)(c_1 - c_4)$
$d_3 = \left(\sin\dfrac{2\pi}{5}\right)(c_2 - c_3)$
$d_4 = d_0 + d_1$
$d_5 = \dfrac{\sqrt{5}}{4}(d_0 - d_1)$
$d_6 = c_0 - \dfrac{1}{4}d_4$
$d_7 = d_6 + d_5$
$d_8 = d_6 - d_5$
$d_9 = -i\left(d_2 + \dfrac{\sin(\pi/5)}{\sin(2\pi/5)}d_3\right)$
$d_{10} = -i\left(\dfrac{\sin(\pi/5)}{\sin(2\pi/5)}d_2 - d_3\right)$
$X_t(k + 5jm) = c_0 + d_4$
$X_t(k + 5jm + m) = \omega_{5l}^{j}(d_7 + d_9)$
$X_t(k + 5jm + 2m) = \omega_{5l}^{2j}(d_8 + d_{10})$
$X_t(k + 5jm + 3m) = \omega_{5l}^{3j}(d_8 - d_{10})$
$X_t(k + 5jm + 4m) = \omega_{5l}^{4j}(d_7 - d_9)$
end do
end do
$l = l/5;\ m = m * 5$
end do

Here the variables c_0–c_4 and d_0–d_{10} are temporary variables.

3.5 Radix-8 FFT Algorithm

Let $n = 8^p$, $X_0(j) = x(j),\ 0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-8 FFT algorithm can be expressed as follows:

$l = n/8;\ m = 1$
do $t = 1,\ p$
do $j = 0,\ l-1$
do $k = 0,\ m-1$
$c_0 = X_{t-1}(k + jm)$
$c_1 = X_{t-1}(k + jm + lm)$
$c_2 = X_{t-1}(k + jm + 2lm)$
$c_3 = X_{t-1}(k + jm + 3lm)$
$c_4 = X_{t-1}(k + jm + 4lm)$
$c_5 = X_{t-1}(k + jm + 5lm)$

$$
\begin{aligned}
&c_6 = X_{t-1}(k + jm + 6lm)\\
&c_7 = X_{t-1}(k + jm + 7lm)\\
&d_0 = c_0 + c_4\\
&d_1 = c_0 - c_4\\
&d_2 = c_2 + c_6\\
&d_3 = -i(c_2 - c_6)\\
&d_4 = c_1 + c_5\\
&d_5 = c_1 - c_5\\
&d_6 = c_3 + c_7\\
&d_7 = c_3 - c_7\\
&e_0 = d_0 + d_2\\
&e_1 = d_0 - d_2\\
&e_2 = d_4 + d_6\\
&e_3 = -i(d_4 - d_6)\\
&e_4 = \frac{\sqrt{2}}{2}(d_5 - d_7)\\
&e_5 = -\frac{\sqrt{2}}{2}i(d_5 + d_7)\\
&e_6 = d_1 + e_4\\
&e_7 = d_1 - e_4\\
&e_8 = d_3 + e_5\\
&e_9 = d_3 - e_5\\
&X_t(k + 8jm) = e_0 + e_2\\
&X_t(k + 8jm + m) = \omega_{8l}^{j}(e_6 + e_8)\\
&X_t(k + 8jm + 2m) = \omega_{8l}^{2j}(e_1 + e_3)\\
&X_t(k + 8jm + 3m) = \omega_{8l}^{3j}(e_7 - e_9)\\
&X_t(k + 8jm + 4m) = \omega_{8l}^{4j}(e_0 - e_2)\\
&X_t(k + 8jm + 5m) = \omega_{8l}^{5j}(e_7 + e_9)\\
&X_t(k + 8jm + 6m) = \omega_{8l}^{6j}(e_1 - e_3)\\
&X_t(k + 8jm + 7m) = \omega_{8l}^{7j}(e_6 - e_8)
\end{aligned}
$$

end do
end do
$l = l/8;\ m = m * 8$
end do

Here the variables c_0–c_7, d_0–d_7 and e_0–e_9 are temporary variables.

References

1. Burrus, C.S., Eschenbacher, P.W.: An in-place, in-order prime factor FFT algorithm. IEEE Trans. Acoust. Speech Signal Process. **ASSP-29**, 806–817 (1981)
2. Dubois, E., Venetsanopoulos, A.N.: A new algorithm for the radix-3 FFT. IEEE Trans. Acoust. Speech Signal Process. **ASSP-26**, 222–225 (1978)
3. Kolba, D.P., Parks, T.W.: A prime factor FFT algorithm using high-speed convolution. IEEE Trans. Acoust. Speech Signal Process. **ASSP-25**, 281–294 (1977)
4. Prakash, S., Rao, V.V.: A new radix-6 FFT algorithm. IEEE Trans. Acoust. Speech Signal Process. **ASSP-29**, 939–941 (1981)
5. Singleton, R.C.: An algorithm for computing the mixed radix fast Fourier transform. IEEE Trans. Audio Electroacoust. **17**, 93–103 (1969)

6. Suzuki, Y., Sone, T., Kido, K.: A new FFT algorithm of radix 3, 6, and 12. IEEE Trans. Acoust. Speech Signal Process. **ASSP-34**, 380–383 (1986)
7. Temperton, C.: Self-sorting mixed-radix fast Fourier transforms. J. Comput. Phys. **52**, 1–23 (1983)
8. Temperton, C.: A generalized prime factor FFT algorithm for any $n = 2^p 3^q 5^r$. SIAM J. Sci. Stat. Comput. **13**, 676–686 (1992)
9. Winograd, S.: On computing the discrete Fourier transform. Math. Comput. **32**, 175–199 (1978)

Chapter 4
Split-Radix FFT Algorithms

Abstract This chapter presents split-radix FFT algorithms. First, split-radix FFT algorithm is given. Next, extended split-radix FFT algorithm is described.

Keywords Split-radix FFT algorithm · Extended split-radix FFT algorithm · Modified split-radix FFT algorithm

4.1 Split-Radix FFT Algorithm

The split-radix FFT algorithm [6] is a variant of the Cooley–Tukey FFT algorithm. Yavne [15] presented a method that is currently known as the split-radix FFT algorithm. Many implementations of the split-radix FFT have been proposed [2, 3, 5, 11, 14]. Johnson and Frigo proposed a modified split-radix FFT algorithm [10], which is known as the FFT algorithm with the lowest number of arithmetic operations.

The split-radix FFT algorithm focuses on the fact that when the radix-4 decomposition is applied to the even-indexed terms in the radix-4 FFT algorithm, the number of multiplications of the twiddle factor [4, 9] is not reduced compared to the radix-2 decomposition. The basic idea behind the split-radix FFT is the application of a radix-2 index map to the even-indexed terms and a radix-4 index map to the odd-indexed terms.

The DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1, \tag{4.1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n is a power of two, then

$$y(2k) = \sum_{j=0}^{n/2-1} \{x(j) + x(j+n/2)\}\, \omega_n^{2jk} \tag{4.2}$$

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_4

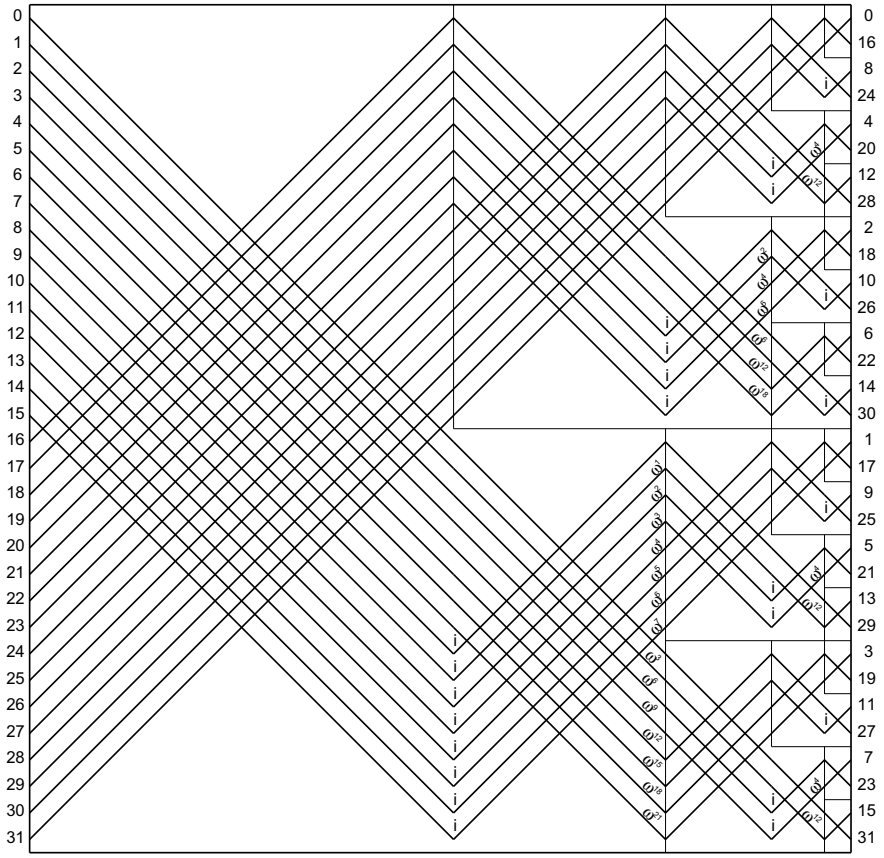

Fig. 4.1 Data flow graph of a decimation-in-frequency split-radix FFT ($n = 32$)

for the even-indexed terms, and

$$y(4k+1) = \sum_{j=0}^{n/4-1} \Big[\{x(j) - x(j+n/2)\} - i\,\{x(j+n/4) - x(j+3n/4)\} \Big] \omega_n^j \omega_n^{4jk} \quad (4.3)$$

$$y(4k+3) = \sum_{j=0}^{n/4-1} \Big[\{x(j) - x(j+n/2)\} + i\,\{x(j+n/4) - x(j+3n/4)\} \Big] \omega_n^{3j} \omega_n^{4jk} \quad (4.4)$$

for the odd-indexed terms. This results in an L-shaped butterfly which relates an n-point DFT to one $n/2$-point DFT and two $n/4$-point DFTs with twiddle factors. The n-point DFT is then obtained by the successive use of such butterflies, up to the last stage, where some usual radix-2 butterflies (without twiddle factors) are also needed. The L-shaped split-radix FFT butterfly advances the calculation of the top

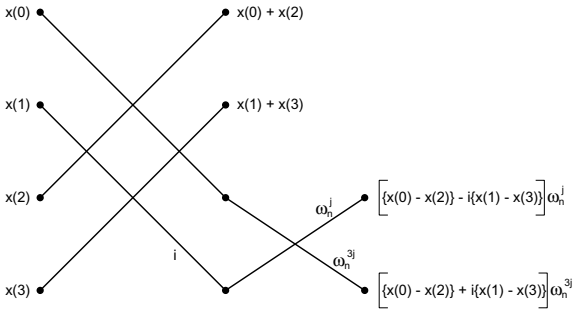

Fig. 4.2 Butterfly used in the data flow graph of a decimation-in-frequency split-radix FFT

half by one of the $\log_2 n$ stages, while the lower half, much as a radix-4 butterfly, calculates two stages at once.

A data flow graph of a decimation-in-frequency split-radix FFT is shown in Fig. 4.1, and the butterfly used in the data flow graph is illustrated in detail in Fig. 4.2.

4.2 Extended Split-Radix FFT Algorithm

The split-radix idea can be extended to other radix pairs [14]. Vetterli and Duhamel [14] have shown that the radix-p/p^2 split-radix approach is generalized to p^m-point DFTs. The extended split-radix FFT algorithm [12] is based on a radix-2/8 split-radix decomposition. The algorithm has the same asymptotic arithmetic complexity as the radix-2/4 split-radix FFT algorithm, but this algorithm has fewer loads and stores than the radix-2/4 split-radix FFT algorithm [12]. Bouguezel et al. proposed a radix-2/8 FFT algorithm for $q \times 2^m$-point DFTs [2] which has fewer arithmetic operations than the extended split-radix FFT algorithm.

The basic idea behind the extended split-radix FFT [12] is the application of a radix-2 index map to the even-indexed terms and a radix-8 [1] index map to the odd-indexed terms.

If n is a power of two in Eq. (4.1), then

$$y(2k) = \sum_{j=0}^{n/2-1} \{x(j) + x(j + n/2)\}\, \omega_n^{2jk} \tag{4.5}$$

for the even-indexed terms, and

$$y(8k+1) = \sum_{j=0}^{n/8-1} \Big[\big\{ \{x(j) - x(j+n/2)\} - i\,\{x(j+n/4) - x(j+3n/4)\} \big\} + \frac{1}{\sqrt{2}} \big\{ (1-i)\,\{x(j+n/8) - x(j+5n/8)\} - (1+i)\,\{x(j+3n/8) - x(j+7n/8)\} \big\} \Big] \omega_n^j \omega_n^{8jk} \tag{4.6}$$

$$y(8k+3) = \sum_{j=0}^{n/8-1} \Big[\big\{ \{x(j) - x(j+n/2)\} + i\,\{x(j+n/4) - x(j+3n/4)\} \big\} - \frac{1}{\sqrt{2}} \big\{ (1+i)\,\{x(j+n/8) - x(j+5n/8)\} - (1-i)\,\{x(j+3n/8) - x(j+7n/8)\} \big\} \Big] \omega_n^{3j} \omega_n^{8jk} \tag{4.7}$$

$$y(8k+5) = \sum_{j=0}^{n/8-1} \Big[\big\{ \{x(j) - x(j+n/2)\} - i\,\{x(j+n/4) - x(j+3n/4)\} \big\} - \frac{1}{\sqrt{2}} \big\{ (1-i)\,\{x(j+n/8) - x(j+5n/8)\} - (1+i)\,\{x(j+3n/8) - x(j+7n/8)\} \big\} \Big] \omega_n^{5j} \omega_n^{8jk} \tag{4.8}$$

$$y(8k+7) = \sum_{j=0}^{n/8-1} \Big[\big\{ \{x(j) - x(j+n/2)\} + i\,\{x(j+n/4) - x(j+3n/4)\} \big\} + \frac{1}{\sqrt{2}} \big\{ (1+i)\,\{x(j+n/8) - x(j+5n/8)\} - (1-i)\,\{x(j+3n/8) - x(j+7n/8)\} \big\} \Big] \omega_n^{7j} \omega_n^{8jk} \tag{4.9}$$

for the odd-indexed terms. This results in an L-shaped butterfly which relates an n-point DFT to one $n/2$-point DFT and four $n/8$-point DFTs with twiddle factors. The n-point DFT is then obtained by the successive use of such butterflies, up to the last two stages, where some radix-2/4 split-radix butterflies (without twiddle factors) are needed, and to the last stage, where some usual radix-2 butterflies (without twiddle factors) are also needed. The L-shaped extended split-radix FFT butterfly advances the calculation of the top half by one of the $\log_2 n$ stages, while the lower half, much as a radix-8 butterfly, calculates three stages at once.

A data flow graph of a decimation-in-frequency extended split-radix FFT is shown in Fig. 4.3 and the butterfly used in the data flow graph is illustrated in detail in Fig. 4.4. The extended split-radix FFT algorithm can be performed in place by repetitive use of the butterfly type of structure [12].

The implementation of the algorithm is considerably simplified by noting that at each stage m ($= 0, \ldots, \log_2 n - 1$) of the algorithm, the butterflies are always applied in a repetitive manner to blocks of length $n/2^m$. Then, only one test is needed

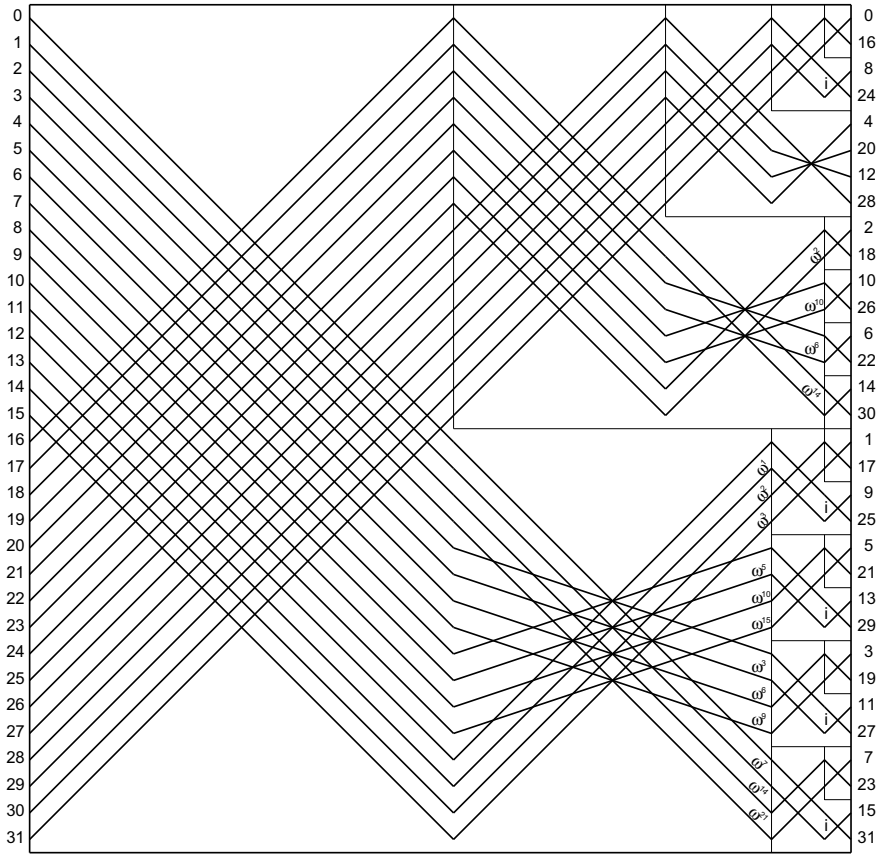

Fig. 4.3 Data flow graph of a decimation-in-frequency extended split-radix FFT ($n = 32$)

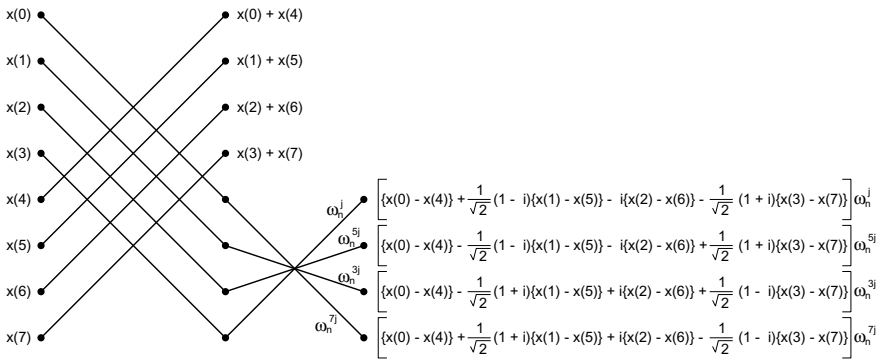

Fig. 4.4 Butterfly used in the data flow graph of a decimation-in-frequency extended split-radix FFT

to decide whether the butterflies are applied to the block (2^m tests at each stage m). We can use a β-vector scheme [13] for the test.

We define a bit vector $\beta_l[0{:}l-1]$. Here, $l = 2^m$, $r = n/l$ and for $k = 0{:}l-1$ we set

$$\beta_l(k) = \begin{cases} 1 & \text{if the butterflies are applied to the blocks} \\ & \quad x[kr : (k+1)r - 1] \text{ at each stage } m, \\ 0 & \text{if the butterflies are not applied to the blocks} \\ & \quad x[kr : (k+1)r - 1] \text{ at each stage } m. \end{cases}$$

The β-vector for the extended split-radix FFT is given by

$$\beta_l = [\, \beta_{l/2} \,|\, \beta_{l/8} \,|\, \beta_{l/8} \,|\, \beta_{l/8} \,|\, \beta_{l/8} \,], \quad l \geq 8.$$

In the $n = 32$ case, the following bit vectors arise:

$$\begin{aligned} \beta_1 &= [\,1\,], \\ \beta_2 &= [\,1\,0\,], \\ \beta_4 &= [\,1\,0\,0\,0\,], \\ \beta_8 &= [\,1\,0\,0\,0\,1\,1\,1\,1\,], \\ \beta_{16} &= [\,1\,0\,0\,0\,1\,1\,1\,1\,1\,0\,1\,0\,1\,0\,1\,0\,]. \end{aligned}$$

The basic one-butterfly program is shown in Listing 4.1. In the basic one-butterfly program, the array `IBETA` is the length-$n/2$ bit vector $\beta_{n/2}$. The `DO 10`, `DO 20`, and `DO 30` loops in Listing 4.1 calculate the β-vector table. In case one does multi-dimensional FFTs, the overhead of the `IF` statement evaluations in the `DO 50`, `DO 80` and `DO 90` loops is negligible. For the inverse FFT, we can use the property of complex conjugate input/output [7].

It should be noted that the Fortran program given in Listing 4.1 was written to be as compact as possible. The basic program does not remove extraneous multiplications and additions. When the exponent of ω_n is zero, a reduction of complex multiplications by one is possible. A special butterfly can be inserted just before the `DO 60` statement that uses fewer multiplications and additions. The `DO 60` loop should then start at `J=2`. This program is called a two-butterfly algorithm and uses significantly fewer operations than the basic one-butterfly algorithm. The two-butterfly algorithm needs the fewest arithmetic operations possible for the extended split-radix FFT.

Table 4.1 gives the number of real multiplications and additions required by the extended split-radix FFT to calculate an n-point DFT of complex data using four real multiplications and two real additions for each complex multiplication by a twiddle factor. Table 4.2 contains the same values when using three real multiplications and three real additions for each complex multiplication.

Table 4.3 compares the operation count of two-butterfly radix-2, radix-4, radix-8, the split-radix, and the extended split-radix FFTs. We note that the extended split-radix FFT uses the lowest total number of arithmetic operations among the

Table 4.1 The number of real multiplications and additions for two extended split-radix FFT programs using four real multiplications and two real additions per complex multiplication for n complex data points

n	One-butterfly		Two-butterfly	
	Mults	Adds	Mults	Adds
8	20	60	4	52
16	60	164	28	148
32	140	404	92	380
64	380	996	252	932
128	940	2356	668	2220
256	2140	5380	1660	5140
512	4940	12180	3932	11676
1024	11260	27236	9148	26180
2048	24940	60020	20892	57996
4096	54940	131268	46844	127220
8192	120460	285268	103900	276988
16384	261180	615460	228412	599076

Table 4.2 The number of real multiplications and additions for two extended split-radix FFT programs using three real multiplications and three real additions per complex multiplication for n complex data points

n	One-butterfly		Two-butterfly	
	Mults	Adds	Mults	Adds
8	16	64	4	52
16	48	176	24	152
32	112	432	76	396
64	304	1072	208	976
128	752	2544	548	2340
256	1712	5808	1352	5448
512	3952	13168	3196	12412
1024	9008	29488	7424	27904
2048	19952	65008	16916	61972
4096	43952	142256	37880	136184
8192	96368	309360	83948	296940
16384	208944	667696	184368	643120

two-butterfly algorithms. In Table 4.4, the asymptotic number of loads, stores, multiplications, and additions used by each algorithm is given.

Let $M(n)$ (respectively, $A(n)$) be the number of real multiplications (respectively, additions) needed to perform an n-point DFT with the extended split-radix FFT

Table 4.3 The number of real multiplications and additions for five FFT programs using four real multiplications and two real additions per complex multiplication for n complex data points

n	Radix-2		Radix-4		Radix-8		Split-radix		Extended split-radix	
	Two-butterfly		Two-butterfly		Two-butterfly		Two-butterfly [11]		Two-butterfly	
	Mults	Adds	Mults	Adds	Mults	Adds	Mults	Adds	Mults	Adds
8	20	58			4	52	8	52	4	52
16	68	162	36	146			32	144	28	148
32	196	418					104	372	92	380
64	516	1026	324	930	260	930	288	912	252	932
128	1284	2434					744	2164	668	2220
256	3076	5634	2052	5122			1824	5008	1660	5140
512	7172	12802			4100	11650	4328	11380	3932	11676
1024	16388	28674	11268	26114			10016	25488	9148	26180
2048	36868	63490					22760	56436	20892	57996
4096	81924	139266	57348	126978	49156	126978	50976	123792	46844	127220
8192	180228	303106					112872	269428	103900	276988
16384	393220	655362	278532	598018			247584	582544	228412	599076

Table 4.4 Number of loads, stores, multiplications, and additions divided by $n \log_2 n$ used by FFT algorithms to compute an n-point DFT. Lower order terms have been omitted [12]

Algorithm	Loads	Stores	Mults	Adds
Radix-2	3	2	2	3
Radix-4	7/4	1	3/2	11/4
Radix-8	5/4	2/3	4/3	11/4
Split-radix	2	4/3	4/3	8/3
Extended split-radix	3/2	1	5/4	11/4

algorithm. This DFT requires 4 real multiplications, 36 real additions, and 4 complex multiplications to evaluate Eqs. (4.5)–(4.9).

The following formulas can be used to generate counts for longer lengths that use the four-multiply, two-add scheme.

One-butterfly:

$$M(n) = M(n/2) + 4M(n/8) + (5/2)n,$$
$$A(n) = A(n/2) + 4A(n/8) + (11/2)n$$

and, with the initial conditions $M(2) = 0$, $M(4) = 0$, $M(8) = 20$, and $A(2) = 4$, $A(4) = 16$, $A(8) = 60$.

We obtain

$$
\begin{aligned}
M(n) &= (5/4)n\log_2 n - (25/16)n + (10/n)(\alpha^{\log_2 n-3} + \beta^{\log_2 n-3}) \\
&\quad - \big(50\sqrt{7}i/(7n)\big)(\alpha^{\log_2 n-3} - \beta^{\log_2 n-3}), \\
A(n) &= (11/4)n\log_2 n - (15/16)n + \big(15/(32n)\big)(\alpha^{\log_2 n} + \beta^{\log_2 n}) \\
&\quad - \big(27\sqrt{7}i/(224n)\big)(\alpha^{\log_2 n} - \beta^{\log_2 n}),
\end{aligned}
$$

where $\alpha = -1 + \sqrt{7}i$ and $\beta = -1 - \sqrt{7}i$.

Two-butterfly:

$$
\begin{aligned}
M(n) &= M(n/2) + 4M(n/8) + (5/2)n - 16, \\
A(n) &= A(n/2) + 4A(n/8) + (11/2)n - 8
\end{aligned}
$$

and, with the initial conditions $M(2) = 0$, $M(4) = 0$, $M(8) = 4$, and $A(2) = 4$, $A(4) = 16$, $A(8) = 52$.

We obtain

$$
\begin{aligned}
M(n) &= (5/4)n\log_2 n - (57/16)n + 4 - \big(7/(32n)\big)(\alpha^{\log_2 n} + \beta^{\log_2 n}) \\
&\quad - \big(13\sqrt{7}i/(224N)\big)(\alpha^{\log_2 n} - \beta^{\log_2 n}), \\
A(n) &= (11/4)n\log_2 n - (31/16)n + 2 - \big(1/(32n)\big)(\alpha^{\log_2 n} + \beta^{\log_2 n}) \\
&\quad - \big(11\sqrt{7}i/(224n)\big)(\alpha^{\log_2 n} - \beta^{\log_2 n}),
\end{aligned}
$$

where $\alpha = -1 + \sqrt{7}i$ and $\beta = -1 - \sqrt{7}i$.

On the other hand, the following gives the same counts for the three-multiply, three-add scheme.

One-butterfly:

$$
\begin{aligned}
M(n) &= M(n/2) + 4M(n/8) + 2n, \\
A(n) &= A(n/2) + 4A(n/8) + 6n
\end{aligned}
$$

and, with the initial conditions $M(2) = 0$, $M(4) = 0$, $M(8) = 16$, and $A(2) = 4$, $A(4) = 16$, $A(8) = 64$.

We obtain

$$
\begin{aligned}
M(n) &= n\log_2 n - (5/4)n + (8/n)(\alpha^{\log_2 n-3} + \beta^{\log_2 n-3}) \\
&\quad - \big(40\sqrt{7}i/(7n)\big)(\alpha^{\log_2 n-3} - \beta^{\log_2 n-3}), \\
A(n) &= 3n\log_2 n - (5/4)n + (5/(8n))(\alpha^{\log_2 n} + \beta^{\log_2 n}) \\
&\quad - \big(9\sqrt{7}i/(56n)\big)(\alpha^{\log_2 n} - \beta^{\log_2 n}),
\end{aligned}
$$

where $\alpha = -1 + \sqrt{7}i$ and $\beta = -1 - \sqrt{7}i$.

Two-butterfly:

$$M(n) = M(n/2) + 4M(n/8) + 2n - 12,$$
$$A(n) = A(n/2) + 4A(n/8) + 6n - 12$$

and, with the initial conditions $M(2) = 0$, $M(4) = 0$, $M(8) = 4$, and $A(2) = 4$, $A(4) = 16$, $A(8) = 52$.

We obtain

$$\begin{aligned} M(n) &= n\log_2 n - (11/4)n + 3 - \big(1/(8n)\big)(\alpha^{\log_2 n} + \beta^{\log_2 n}) \\ &\quad - \big(3\sqrt{7}i/(56n)\big)(\alpha^{\log_2 n} - \beta^{\log_2 n}), \\ A(n) &= 3n\log_2 n - (11/4)n + 3 - \big(1/(8n)\big)(\alpha^{\log_2 n} + \beta^{\log_2 n}) \\ &\quad - \big(3\sqrt{7}i/(56n)\big)(\alpha^{\log_2 n} - \beta^{\log_2 n}), \end{aligned}$$

where $\alpha = -1 + \sqrt{7}i$ and $\beta = -1 - \sqrt{7}i$.

It is interesting to note that the split-radix and extended split-radix FFT algorithms need the same asymptotic arithmetic complexity. The superiority of the extended split-radix FFT algorithm comes from its lower loads and stores [12].

Because the extended split-radix FFT algorithm is a combination of radix-2 and radix-8 decomposition (at each step), one could expect that its loads and stores would be somewhere between the loads and stores of radix-2 and radix-8. In fact, the number of loads involved is equal to that of radix-4, and the number of stores involved is lower than that of radix-4, as is shown in Table 4.4.

Listing 4.1 An extended split-radix decimation-in-frequency FFT [8]

```
C-------------------------------------------------------------C
C      An Extended Split-Radix DIF FFT                        C
C      Complex input and output in data arrays X and Y        C
C      Length is N = 2**M                                     C
C                                                             C
C      D. Takahashi, University of Tsukuba, Sep. 2002         C
C                                                             C
C      Reference:                                             C
C          D. Takahashi, "An extended split-radix FFT         C
C          algorithm," IEEE Signal Processing Lett.,          C
C          vol. 8, pp. 145-147, May 2001.                     C
C-------------------------------------------------------------C
       SUBROUTINE ESRFFT(X,Y,IBETA,N,M,IV)
       IMPLICIT REAL*8 (A-H,O-Z)
       REAL*8 X(N),Y(N)
       INTEGER*4 IBETA(N/2)
       PARAMETER (C21=0.70710678118654752D0,
      +           TWOPI=6.28318530717958647D0)
C-------Beta vector table-------
       IBETA(1)=1
       IBETA(2)=0
       IBETA(3)=0
       IBETA(4)=0
       ID=1
       DO 30 K=3,M-1
         DO 20 J=1,4
           IS=(J+3)*ID+1
           DO 10 I=IS,IS+ID-1
             IBETA(I)=IBETA(I-IS+1)
   10      CONTINUE
   20    CONTINUE
```

```
        ID=2*ID
   30 CONTINUE
C
      IF (IV .EQ. -1) THEN
        DO 40 I=1,N
          Y(I)=-Y(I)
   40   CONTINUE
      END IF
C-------L shaped butterflies-------
      L=1
      N2=N
      DO 70 K=1,M-2
        N8=N2/8
        E=-TWOPI/DBLE(N2)
        A=0.0D0
        DO 60 J=1,N8
          A3=3.0D0*A
          A5=5.0D0*A
          A7=7.0D0*A
          CC1=DCOS(A)
          SS1=DSIN(A)
          CC3=DCOS(A3)
          SS3=DSIN(A3)
          CC5=DCOS(A5)
          SS5=DSIN(A5)
          CC7=DCOS(A7)
          SS7=DSIN(A7)
          A=DBLE(J)*E
          DO 50 I=1,L
            IF (IBETA(I) .EQ. 1) THEN
              I0=(I-1)*N2+J
              I1=I0+N8
              I2=I1+N8
              I3=I2+N8
              I4=I3+N8
              I5=I4+N8
              I6=I5+N8
              I7=I6+N8
              X0=X(I0)-X(I4)
              Y0=Y(I0)-Y(I4)
              X1=X(I1)-X(I5)
              Y1=Y(I1)-Y(I5)
              X2=Y(I2)-Y(I6)
              Y2=X(I6)-X(I2)
              X3=X(I3)-X(I7)
              Y3=Y(I3)-Y(I7)
              X(I0)=X(I0)+X(I4)
              Y(I0)=Y(I0)+Y(I4)
              X(I1)=X(I1)+X(I5)
              Y(I1)=Y(I1)+Y(I5)
              X(I2)=X(I2)+X(I6)
              Y(I2)=Y(I2)+Y(I6)
              X(I3)=X(I3)+X(I7)
              Y(I3)=Y(I3)+Y(I7)
              U0=X0+C21*(X1-X3)
              V0=Y0+C21*(Y1-Y3)
              U1=X0-C21*(X1-X3)
              V1=Y0-C21*(Y1-Y3)
              U2=X2+C21*(Y1+Y3)
              V2=Y2-C21*(X1+X3)
              U3=X2-C21*(Y1+Y3)
              V3=Y2+C21*(X1+X3)
              X(I4)=CC1*(U0+U2)-SS1*(V0+V2)
              Y(I4)=CC1*(V0+V2)+SS1*(U0+U2)
              X(I5)=CC5*(U1+U3)-SS5*(V1+V3)
              Y(I5)=CC5*(V1+V3)+SS5*(U1+U3)
              X(I6)=CC3*(U1-U3)-SS3*(V1-V3)
```

```
              Y(I6)=CC3*(V1-V3)+SS3*(U1-U3)
              X(I7)=CC7*(U0-U2)-SS7*(V0-V2)
              Y(I7)=CC7*(V0-V2)+SS7*(U0-U2)
            END IF
 50       CONTINUE
 60     CONTINUE
        L=2*L
        N2=N2/2
 70   CONTINUE
C-------Length four butterflies-------
      DO 80 I=1,N/4
        IF (IBETA(I) .EQ. 1) THEN
          I0=4*I-3
          I1=I0+1
          I2=I1+1
          I3=I2+1
          X0=X(I0)-X(I2)
          Y0=Y(I0)-Y(I2)
          X1=Y(I1)-Y(I3)
          Y1=X(I3)-X(I1)
          X(I0)=X(I0)+X(I2)
          Y(I0)=Y(I0)+Y(I2)
          X(I1)=X(I1)+X(I3)
          Y(I1)=Y(I1)+Y(I3)
          X(I2)=X0+X1
          Y(I2)=Y0+Y1
          X(I3)=X0-X1
          Y(I3)=Y0-Y1
        END IF
 80   CONTINUE
C-------Length two butterflies-------
      DO 90 I=1,N/2
        IF (IBETA(I) .EQ. 1) THEN
          I0=2*I-1
          I1=I0+1
          X0=X(I0)-X(I1)
          Y0=Y(I0)-Y(I1)
          X(I0)=X(I0)+X(I1)
          Y(I0)=Y(I0)+Y(I1)
          X(I1)=X0
          Y(I1)=Y0
        END IF
 90   CONTINUE
C-------Digit reverse counter-------
      J=1
      DO 110 I=1,N-1
        IF (I .LT. J) THEN
          X0=X(I)
          Y0=Y(I)
          X(I)=X(J)
          Y(I)=Y(J)
          X(J)=X0
          Y(J)=Y0
        END IF
        K=N/2
100     IF (K .LT. J) THEN
          J=J-K
          K=K/2
          GO TO 100
        END IF
        J=J+K
110   CONTINUE
C
      IF (IV .EQ. -1) THEN
        DO 120 I=1,N
          X(I)=X(I)/DBLE(N)
          Y(I)=-Y(I)/DBLE(N)
```

```
120    CONTINUE
     END IF
     RETURN
     END
```

References

1. Bergland, G.D.: A fast Fourier transform algorithm using base 8 iterations. Math. Comput. **22**, 275–279 (1968)
2. Bouguezel, S., Ahmad, M.O., Swamy, M.N.S.: A new radix-2/8 FFT algorithm for length-$q \times 2^m$ DFTs. IEEE Trans. Circuits Syst. I Reg. Papers **51**, 1723–1732 (2004)
3. Bouguezel, S., Ahmad, M.O., Swamy, M.N.S.: A general class of split-radix FFT algorithms for the computation of the DFT of length-2^m. IEEE Trans. Signal Process. **55**, 4127–4138 (2007)
4. Brigham, E.O.: The Fast Fourier Transform and Its Applications. Prentice-Hall, Englewood Cliffs, NJ (1988)
5. Duhamel, P.: Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data. IEEE Trans. Acoust. Speech Signal Process. **ASSP-34**, 285–295 (1986)
6. Duhamel, P., Hollmann, H.: Split radix FFT algorithm. Electron. Lett. **20**, 14–16 (1984)
7. Duhamel, P., Piron, B., Etcheto, J.M.: On computing the inverse DFT. IEEE Trans. Acoust. Speech Signal Process. **36**, 285–286 (1988)
8. FFTE: A Fast Fourier Transform Package. http://www.ffte.jp/
9. Gentleman, W.M., Sande, G.: Fast Fourier transforms: for fun and profit. In: Proceedings of AFIPS '66 Fall Joint Computer Conference, pp. 563–578 (1966)
10. Johnson, S.G., Frigo, M.: A modified split-radix FFT with fewer arithmetic operations. IEEE Trans. Signal Process. **55**, 111–119 (2007)
11. Sorensen, H.V., Heideman, M.T., Burrus, C.S.: On computing the split-radix FFT. IEEE Trans. Acoust. Speech Signal Process. **ASSP-34**, 152–156 (1986)
12. Takahashi, D.: An extended split-radix FFT algorithm. IEEE Trans. Signal Process. **8**, 145–147 (2001)
13. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia, PA (1992)
14. Vetterli, M., Duhamel, P.: Split-radix algorithms for length-p^m DFT's. IEEE Trans. Acoust. Speech Signal Process. **37**, 57–64 (1989)
15. Yavne, R.: An economical method for calculating the discrete Fourier transform. In: Proceedings of AFIPS '68 Fall Joint Computer Conference, Part I, pp. 115–125 (1968)

Chapter 5
Multidimensional FFT Algorithms

Abstract In this chapter, two- and three-dimensional FFT algorithms are explained as examples of multidimensional FFT algorithms. As multidimensional FFT algorithms, there are a row–column algorithm and a vector-radix FFT algorithm (Rivard, IEEE Trans. Acoust. Speech Signal Process. **25**(3), 250–252, 1977 [1]). We describe multidimensional FFT algorithms based on the row–column algorithm.

Keywords Row–column algorithm · Two-dimensional FFT algorithm · Three-dimensional FFT algorithm

5.1 Definition of Two-Dimensional DFT

The two-dimensional DFT is given by

$$y(k_1, k_2) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2}, \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{5.1}$$

Moreover, the inverse two-dimensional DFT is given by

$$x(j_1, j_2) = \frac{1}{n_1 n_2} \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} y(k_1, k_2) \omega_{n_1}^{-j_1 k_1} \omega_{n_2}^{-j_2 k_2}, \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{5.2}$$

where $\omega_{n_r} = e^{-2\pi i / n_r}$ $(1 \le r \le 2)$ and $i = \sqrt{-1}$.

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_5

5.2 Two-Dimensional FFT Algorithm

Equation (5.1) can be modified as follows:

$$y(k_1, k_2) = \sum_{j_2=0}^{n_2-1} \left[\sum_{j_1=0}^{n_1-1} x(j_1, j_2) \omega_{n_1}^{j_1 k_1} \right] \omega_{n_2}^{j_2 k_2}. \tag{5.3}$$

Furthermore, Eq. (5.3) can be calculated by applying the one-dimensional DFT twice, as follows:

$$t(k_1, j_2) = \sum_{j_1=0}^{n_1-1} x(j_1, j_2) \omega_{n_1}^{j_1 k_1}, \tag{5.4}$$

$$y(k_1, k_2) = \sum_{j_2=0}^{n_2-1} t(k_1, j_2) \omega_{n_2}^{j_2 k_2}. \tag{5.5}$$

In other words, first, n_1-point DFTs are performed n_2 times, and n_2-point DFTs are then performed n_1 times. This method is referred to as the row–column algorithm.

The following two-dimensional FFT based on the row–column algorithm is derived from Eq. (5.3):

Step 1: n_2 individual n_1-point multicolumn FFTs

$$x_1(k_1, j_2) = \sum_{j_1=0}^{n_1-1} x(j_1, j_2) \omega_{n_1}^{j_1 k_1}.$$

Step 2: Transposition

$$x_2(j_2, k_1) = x_1(k_1, j_2).$$

Step 3: n_1 individual n_2-point multicolumn FFTs

$$x_3(k_2, k_1) = \sum_{j_2=0}^{n_2-1} x_2(j_2, k_1) \omega_{n_2}^{j_2 k_2}.$$

Step 4: Transposition

$$y(k_1, k_2) = x_3(k_2, k_1).$$

The program of the two-dimensional FFT based on the row–column algorithm is shown in Listing 5.1.

Listing 5.1 Two-dimensional FFT based on row–column algorithm

```
      subroutine fft2d(x,n1,n2)
      implicit real*8 (a-h,o-z)
      complex*16 x(n1,n2),y(n2,n1)
! Step 1: n2 individual n1-point multicolumn FFTs
      do j=1,n2
        call fft(x(1,j),n1)
```

```
      end do
! Step 2: Transposition
      do i=1,n1
        do j=1,n2
          y(j,i)=x(i,j)
        end do
      end do
! Step 3: n1 individual n2-point multicolumn FFTs
      do i=1,n1
        call fft(y(1,i),n2)
      end do
! Step 4: Transposition
      do j=1,n2
        do i=1,n1
          x(i,j)=y(j,i)
        end do
      end do
      return
      end
```

5.3 Definition of Three-Dimensional DFT

The three-dimensional DFT is given by

$$y(k_1, k_2, k_3) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3) \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \omega_{n_3}^{j_3 k_3},$$
$$0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1, \quad 0 \le k_3 \le n_3 - 1. \tag{5.6}$$

Moreover, the inverse three-dimensional DFT is given by

$$x(j_1, j_2, j_3) = \frac{1}{n_1 n_2 n_3} \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_3-1} y(k_1, k_2, k_3) \omega_{n_1}^{-j_1 k_1} \omega_{n_2}^{-j_2 k_2} \omega_{n_3}^{-j_3 k_3},$$
$$0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \quad 0 \le j_3 \le n_3 - 1, \tag{5.7}$$

where $\omega_{n_r} = e^{-2\pi i / n_r}$ $(1 \le r \le 3)$ and $i = \sqrt{-1}$.

5.4 Three-Dimensional FFT Algorithm

Equation (5.6) can be modified as follows:

$$y(k_1, k_2, k_3) = \sum_{j_3=0}^{n_3-1} \left[\sum_{j_2=0}^{n_2-1} \left[\sum_{j_1=0}^{n_1-1} x(j_1, j_2, j_3) \omega_{n_1}^{j_1 k_1} \right] \omega_{n_2}^{j_2 k_2} \right] \omega_{n_3}^{j_3 k_3}. \tag{5.8}$$

Furthermore, Eq. (5.8) can be calculated by applying the one-dimensional DFT three times, as follows:

$$t(k_1, j_2, j_3) = \sum_{j_1=0}^{n_1-1} x(j_1, j_2, j_3)\omega_{n_1}^{j_1 k_1}, \tag{5.9}$$

$$u(k_1, k_2, j_3) = \sum_{j_2=0}^{n_2-1} t(k_1, j_2, j_3)\omega_{n_2}^{j_2 k_2}, \tag{5.10}$$

$$y(k_1, k_2, k_3) = \sum_{j_3=0}^{n_3-1} u(k_1, k_2, j_3)\omega_{n_3}^{j_3 k_3}. \tag{5.11}$$

Similar to the two-dimensional FFT based on the row–column algorithm, n_1-point DFTs are performed $n_2 n_3$ times, and n_2-point DFTs are performed $n_3 n_1$ times, then n_3-point DFTs are performed $n_1 n_2$ times.

The following three-dimensional FFT based on the row–column algorithm is derived from Eq. (5.8):

Step 1: $n_2 n_3$ individual n_1-point multicolumn FFTs

$$x_1(k_1, j_2, j_3) = \sum_{j_1=0}^{n_1-1} x(j_1, j_2, j_3)\omega_{n_1}^{j_1 k_1}.$$

Step 2: Transposition

$$x_2(j_2, j_3, k_1) = x_1(k_1, j_2, j_3).$$

Step 3: $n_3 n_1$ individual n_2-point multicolumn FFTs

$$x_3(k_2, j_3, k_1) = \sum_{j_2=0}^{n_2-1} x_2(j_2, j_3, k_1)\omega_{n_2}^{j_2 k_2}.$$

Step 4: Transposition

$$x_4(j_3, k_1, k_2) = x_3(k_2, j_3, k_1).$$

Step 5: $n_1 n_2$ individual n_3-point multicolumn FFTs

$$x_5(k_3, k_1, k_2) = \sum_{j_3=0}^{n_3-1} x_4(j_3, k_1, k_2)\omega_{n_3}^{j_3 k_3}.$$

Step 6: Transposition

$$y(k_1, k_2, k_3) = x_5(k_3, k_1, k_2).$$

The program of the three-dimensional FFT based on the row–column algorithm is shown in Listing 5.2.

Listing 5.2 Three-dimensional FFT based on row–column algorithm

```
      subroutine fft3d(x,n1,n2,n3)
      implicit real*8 (a-h,o-z)
      complex*16 x(n1,n2,n3),y(n2,n3,n1),z(n3,n1,n2)
! Step 1: n2*n3 individual n1-point multicolumn FFTs
      do k=1,n3
        do j=1,n2
          call fft(x(1,j,k),n1)
        end do
      end do
! Step 2: Transposition
      do i=1,n1
        do k=1,n3
          do j=1,n2
            y(j,k,i)=x(i,j,k)
          end do
        end do
      end do
! Step 3: n3*n1 individual n2-point multicolumn FFTs
      do i=1,n1
        do k=1,n3
          call fft(y(1,k,i),n2)
        end do
      end do
! Step 4: Transposition
      do j=1,n2
        do i=1,n1
          do k=1,n3
            z(k,i,j)=y(j,k,i)
          end do
        end do
      end do
! Step 5: n1*n2 individual n3-point multicolumn FFTs
      do j=1,n2
        do i=1,n1
          call fft(z(1,i,j),n3)
        end do
      end do
! Step 6: Transposition
      do k=1,n3
        do j=1,n2
          do i=1,n1
            x(i,j,k)=z(k,i,j)
          end do
        end do
      end do
      return
      end
```

Reference

1. Rivard, G.E.: Direct fast Fourier transform of bivariate functions. IEEE Trans. Acoust. Speech Signal Process. **25**(3), 250–252 (1977). ASSP-25

Chapter 6
High-Performance FFT Algorithms

Abstract This chapter presents high-performance FFT algorithms. First, the four-step FFT algorithm and five-step FFT algorithm are described. Next, the six-step FFT algorithm and blocked six-step FFT algorithm are explained. Then, nine-step FFT algorithm and recursive six-step FFT, and blocked multidimensional FFT algorithms are described. Finally, FFT algorithms suitable for fused multiply–add instructions and FFT algorithms for SIMD instructions are explained.

Keywords Four-step FFT algorithm · Six-step FFT algorithm · Fused multiply–add (FMA) instruction · SIMD instruction

6.1 Four-Step FFT Algorithm

The DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1, \tag{6.1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n has factors n_1 and n_2 ($n = n_1 \times n_2$), then the indices j and k can be expressed as

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{6.2}$$

We can define x and y in Eq. (6.1) as two-dimensional arrays (in column-major order):

$$x(j) = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{6.3}$$

$$y(k) = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{6.4}$$

Substituting the indices j and k in Eq. (6.1) with those in Eq. (6.2), and using the relation of $n = n_1 \times n_2$, we can derive the following equation:

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_6

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2)\omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{6.5}$$

This derivation leads to the following four-step FFT algorithm [1]:

Step 1: n_1 simultaneous n_2-point multirow FFTs

$$x_1(j_1, k_2) = \sum_{j_2=0}^{n_2-1} x(j_1, j_2)\omega_{n_2}^{j_2 k_2}.$$

Step 2: Twiddle factor multiplication

$$x_2(j_1, k_2) = x_1(j_1, k_2)\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 3: Transposition

$$x_3(k_2, j_1) = x_2(j_1, k_2).$$

Step 4: n_2 simultaneous n_1-point multirow FFTs

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} x_3(k_2, j_1)\omega_{n_1}^{j_1 k_1}.$$

The distinctive features of the four-step FFT algorithm can be summarized as follows:

- If n_1 is equal to n_2 ($n_1 = n_2 \equiv \sqrt{n}$), the innermost loop length can be fixed to $\sqrt{n}$. This feature makes the algorithm suitable for vector processors.
- A matrix transposition takes place just once (Step 3).

An ns simultaneous n-point radix-2 multirow FFT based on the Stockham FFT algorithm is shown in Fig. 6.1.

Fig. 6.1 Multirow FFT based on Stockham FFT algorithm

```
n = 2^p and ω_n = e^{-2πi/n}
l = n/2; m = 1
do t = 1, p
  complex*16 X_{t-1}(0 : ns − 1, 0 : n − 1), X_t(0 : ns − 1, 0 : n − 1)
  do j = 0, l − 1
    do k = 0, m − 1
      do row = 0, ns − 1
        c_0 = X_{t-1}(row, k + jm)
        c_1 = X_{t-1}(row, k + jm + lm)
        X_t(row, k + 2jm) = c_0 + c_1
        X_t(row, k + 2jm + m) = ω_{2l}^j (c_0 − c_1)
      end do
    end do
  end do
  l = l/2; m = m ∗ 2
end do
```

6.2 Five-Step FFT Algorithm

We can extend the four-step FFT algorithm in another way into a three-dimensional formulation. If n has factors n_1, n_2 and n_3 ($n = n_1 n_2 n_3$) in Eq. (6.1), then the indices j and k can be expressed as

$$j = j_1 + j_2 n_1 + j_3 n_1 n_2, \quad k = k_3 + k_2 n_3 + k_1 n_2 n_3. \tag{6.6}$$

We can define x and y as three-dimensional arrays (in Fortran notation), e.g.,

$$\begin{aligned} x(j) &= x(j_1, j_2, j_3), \\ &\quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \quad 0 \le j_3 \le n_3 - 1, \end{aligned} \tag{6.7}$$

$$\begin{aligned} y(k) &= y(k_3, k_2, k_1), \\ &\quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1, \quad 0 \le k_3 \le n_3 - 1. \end{aligned} \tag{6.8}$$

Substituting the indices j and k in Eq. (6.1) by those in Eq. (6.6) and using the relation of $n = n_1 n_2 n_3$, we can derive the following equation:

$$y(k_3, k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3) \omega_{n_3}^{j_3 k_3} \omega_{n_2 n_3}^{j_2 k_3} \omega_{n_2}^{j_2 k_2} \omega_n^{j_1 k_3} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{6.9}$$

This derivation leads to the following five-step FFT algorithm [23]:

Step 1: $n_1 n_2$ simultaneous n_3-point multirow FFTs

$$x_1(j_1, j_2, k_3) = \sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3) \omega_{n_3}^{j_3 k_3}.$$

Step 2: Twiddle factor multiplication and transposition

$$x_2(k_3, j_1, j_2) = x_1(j_1, j_2, k_3) \omega_{n_2 n_3}^{j_2 k_3}.$$

Step 3: $n_3 n_1$ simultaneous n_2-point multirow FFTs

$$x_3(k_3, j_1, k_2) = \sum_{j_2=0}^{n_2-1} x_2(k_3, j_1, j_2) \omega_{n_2}^{j_2 k_2}.$$

Step 4: Twiddle factor multiplication and rearrangement

$$x_4(k_3, k_2, j_1) = x_3(k_3, j_1, k_2) \omega_n^{j_1 k_3} \omega_{n_1 n_2}^{j_1 k_2}.$$

Step 5: $n_3 n_2$ simultaneous n_1-point multirow FFTs

$$y(k_3, k_2, k_1) = \sum_{j_1=0}^{n_1-1} x_4(k_3, k_2, j_1) \omega_{n_1}^{j_1 k_1}.$$

We note that we combined some of the operations with data movements as in Steps 2 and 4 to gain efficiency in utilizing the memory bandwidth.

The distinctive features of the five-step FFT algorithm can be summarized as follows:

- If n_1, n_2, and n_3 are equal ($n_1 = n_2 = n_3 \equiv n^{1/3}$), the innermost loop length can be fixed to $n^{2/3}$. Thus, the five-step FFT algorithm is more suitable for vector processors than the four-step FFT algorithm.
- A matrix transposition with twiddle factor multiplication takes place twice (Steps 2 and 4).

The five-step FFT algorithm is based on representing the one-dimensional array $x(n)$ as a three-dimensional array $x(n_1, n_2, n_3)$. This idea can also be found in Hegland's approach [7]. One difference between the five-step FFT algorithm and Hegland's approach is in the choice of the size of the n_i $(1 \leq i \leq 3)$. Both Hegland's approach and the five-step FFT algorithm choose $n_1 = n_3$. Hegland chooses n_2 to be of minimal size [7].

Taking the opposite approach, we suggest that all the n_i $(1 \leq i \leq 3)$ should be of equal size. This is because the minimum innermost loop length (i.e., $\min(n_1 n_2, n_2 n_3, n_3 n_1)$) can be maximized. In view of the innermost loop length, the five-step FFT algorithm is more advantageous than Hegland's approach. While on the other hand, Hegland's approach requires one transpose step, the five-step FFT algorithm requires two transpose steps.

6.3 Six-Step FFT Algorithm

In this section, we explain the six-step FFT algorithm [1] that can effectively use the cache memory. In the six-step FFT algorithm, it is possible to reduce cache misses compared to the Stockham FFT algorithm by computing a one-dimensional FFT represented in the two-dimensional formulation described in Eq. (6.5).

The following six-step FFT algorithm [1] is derived from Eq. (6.5):

Step 1: Transposition

$$x_1(j_2, j_1) = x(j_1, j_2).$$

Step 2: n_1 individual n_2-point multicolumn FFTs

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1)\omega_{n_2}^{j_2 k_2}.$$

Step 3: Twiddle factor multiplication

$$x_3(k_2, j_1) = x_2(k_2, j_1)\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 4: Transposition

$$x_4(j_1, k_2) = x_3(k_2, j_1).$$

Step 5: n_2 individual n_1-point multicolumn FFTs

$$x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1 k_1}.$$

Step 6: Transposition

$$y(k_2, k_1) = x_5(k_1, k_2).$$

In Step 3, $\omega_{n_1 n_2}^{j_1 k_2} = e^{-2\pi i j_1 k_2/(n_1 n_2)}$ is a root of unity called a twiddle factor and is a complex number. Moreover, the transposition of the matrix in Steps 1 and 4 is carried out in order to make the memory access of the multicolumn FFT in Steps 2 and 5 continuous. Furthermore, transposing the matrix of Step 6 is necessary in order to make the order of the input data and the order of the output data the same.

Listing 6.1 Six-step FFT algorithm

```
      complex*16 x(n1,n2),y(n2,n1),w(n1,n2)
!
! Step 1: Transposition
      do i=1,n1
        do j=1,n2
          y(j,i)=x(i,j)
        end do
      end do
! Step 2: n1 individual n2-point multicolumn FFTs
      do i=1,n1
        call fft(y(1,i),n2)
      end do
! Steps 3-4: Twiddle factor multiplication and transposition
      do j=1,n2
        do i=1,n1
          x(i,j)=y(j,i)*w(i,j)
        end do
      end do
! Step 5: n2 individual n1-point multicolumn FFTs
      do j=1,n2
        call fft(x(1,j),n1)
      end do
! Step 6: Transposition
      do i=1,n1
        do j=1,n2
          y(j,i)=x(i,j)
        end do
      end do
```

The features of the six-step FFT algorithm are as follows:

- When $n_1 = n_2 = \sqrt{n}$, $\sqrt{n}$ individual $\sqrt{n}$-point multicolumn FFTs [29] are performed in Steps 2 and 5. The $\sqrt{n}$-point multicolumn FFTs have high locality of memory access and are suitable for a processor equipped with a cache memory.
- It is necessary to transpose the matrix three times. These three matrix transpositions are a bottleneck in the processor equipped with the cache memory.

Cache misses can be reduced by cache blocking the matrix transposition in Steps 1, 4, and 6. However, even when cache blocking is performed, the multicolumn FFT and the matrix transposition are separated. Therefore, there is a problem in that the

data placed in the cache memory in the multicolumn FFT cannot be effectively reused when transposing the matrix.

The program of the six-step FFT algorithm is shown in Listing 6.1.

6.4 Blocked Six-Step FFT Algorithm

In this section, in order to further effectively reuse the data in the cache and reduce the number of cache misses, we explain a blocked six-step FFT algorithm [22] that combines the separated multicolumn FFT and the matrix transposition in the six-step FFT. In the six-step FFT described in Sect. 6.3, let $n = n_1 n_2$, and let n_b be the block size. Here, it is assumed that the processor is equipped with a multilevel cache memory. The blocked six-step FFT algorithm is as follows:

1. Assume that the input data is contained in a complex array `x` of size $n_1 \times n_2$. At this time, while transferring n_b rows of data from $n_1 \times n_2$ array `x`, transfer the data to a work array `work` of size $n_2 \times n_b$. Here, the block size n_b is set such that the array `work` fits into the L2 cache.
2. Perform n_b individual n_2-point multicolumn FFTs on the $n_2 \times n_b$ array in the L2 cache. Here, it is assumed that each column FFT can be performed substantially in the L2 cache.
3. After multicolumn FFTs, multiply each element of the $n_2 \times n_b$ array `work` remaining in the L2 cache by the twiddle factor `w`. The data of this $n_2 \times n_b$ array `work` is then stored again in the same place as the original $n_1 \times n_2$ array `x` while transposing n_b rows at a time.
4. Perform n_2 individual n_1-point multicolumn FFTs on the $n_1 \times n_2$ array. Here, each column FFT can almost be performed in the L1 cache.
5. Finally, this $n_1 \times n_2$ array `x` is transposed by n_b rows and is stored in the $n_2 \times n_1$ array `y`.

Listing 6.2 Blocked six-step FFT algorithm

```
      complex*16 x(n1,n2),y(n2,n1),w(n1,n2),work(n2+np,nb)
!
      do ii=1,n1,nb
! Step 1: Blocked transposition
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              work(j,i-ii+1)=x(i,j)
            end do
          end do
        end do
! Step 2: n1 individual n2-point multicolumn FFTs
        do i=ii,min(ii+nb-1,n1)
          call fft(work(1,i-ii+1),n2)
        end do
! Steps 3-4: Blocked twiddle factor multiplication and
!            transposition
```

```
        do j=1,n2
          do i=ii,min(ii+nb-1,n1)
            x(i,j)=work(j,i-ii+1)*w(i,j)
          end do
        end do
      end do
      do jj=1,n2,nb
!  Step 5: n2 individual n1-point multicolumn FFTs
        do j=jj,min(jj+nb-1,n2)
          call fft(x(1,j),n1)
        end do
!  Step 6: Blocked transposition
        do i=1,n1
          do j=jj,min(jj+nb-1,n2)
            y(j,i)=x(i,j)
          end do
        end do
      end do
```

The program of the blocked six-step FFT algorithm is shown in Listing 6.2. Here, the parameters `nb` and `np` are the blocking parameter and the padding parameter, respectively. The array `work` is the work array. Figure 6.2 shows the memory layout of the blocked six-step FFT algorithm. In Fig. 6.2, numbers 1 through 8 in the arrays `x`, `work`, and `y` indicate the sequence of accessing the array. By padding the work array `work`, it is possible to minimize the occurrence of cache line conflict when transferring data from array `work` to array `x` or to perform multicolumn FFTs on array `work` whenever possible. Note that this algorithm is a *two-pass* algorithm

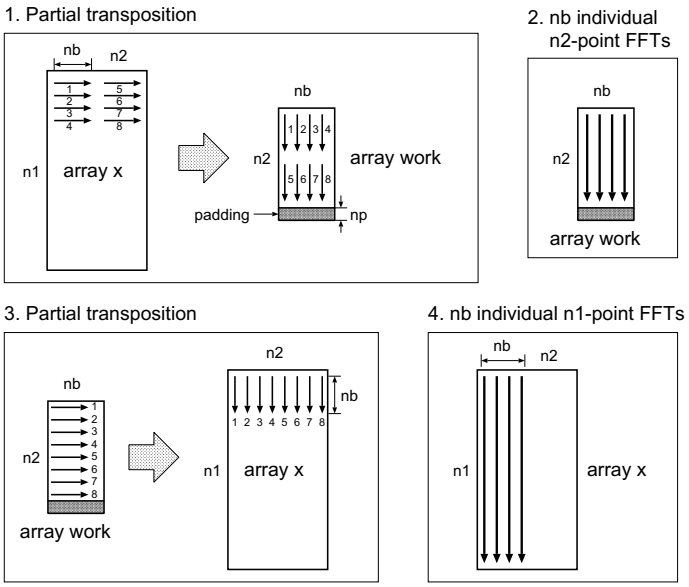

Fig. 6.2 Memory layout of the blocked six-step FFT algorithm

[1]. Here, the *two-pass* algorithm reads and writes each element of the array twice. In other words, in the blocked six-step FFT algorithm, the number of arithmetic operations of the n-point FFT is $O(n \log n)$, whereas the number of accesses to the main memory is ideally $O(n)$.

In this section, it is assumed that each column FFT in Steps 2 and 4 fits into the L1 cache. However, when the problem size n is very large, it is expected that each column FFT may not fit into the L1 cache. In such a case, it is possible to calculate each column FFT in the L1 cache by reducing the problem size of each column FFT using multidimensional formulation instead of the two-dimensional formulation. However, when using a multidimensional formulation of more than three dimensions, a *two-pass* algorithm is not be used. For example, in the case of using a three-dimensional representation, a *three-pass* algorithm is used. Thus, as the number of dimensions of the multidimensional formulation increases, FFTs of larger problem sizes can be performed. On the other hand, the number of accesses to main memory increases, indicating that the performance also depends on the capacity of the cache memory in the blocked six-step FFT.

Even if we use an out-of-place algorithm (e.g., Stockham FFT algorithm) for the multicolumn FFTs of Steps 2 and 4, the additional array size is only $O(\sqrt{n})$. Moreover, if the output of the one-dimensional FFT is the transposed output, the transposition of the matrix of Step 5 can be omitted. In this case, the size of the array `work` needs only be $O(\sqrt{n})$. In general, the array of twiddle factors is the same size as the input array. We can use Bailey's technique [1] to reduce the array size of twiddle factors in Step 3.

Bailey's technique is as follows:

$$\begin{aligned} w(j + ra, k + sb) &= \omega_n^{(j+ra)(k+sb)} \\ &= \omega_n^{jk} \omega_n^{jsb} \omega_n^{rak} \omega_n^{rasb} \\ &= w_0(j, k) w_1(j, sb) w_2(ra, k) w_3(ra, sb), \end{aligned} \tag{6.10}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If we assume that $n_1 = n_2 = a^2 = b^2$, then the total array size of twiddle factors is only $O(\sqrt{n})$. Moreover, these reduced twiddle factors fit into the data cache.

6.5 Nine-Step FFT Algorithm

We can extend the six-step FFT algorithm in another way into a three-dimensional formulation.

From Eq. (6.9), we can derive the following nine-step FFT algorithm [27]:

Step 1: Transposition

$$x_1(j_3, j_1, j_2) = x(j_1, j_2, j_3).$$

Step 2: $n_1 n_2$ individual n_3-point multicolumn FFTs

$$x_2(k_3, j_1, j_2) = \sum_{j_3=0}^{n_3-1} x_1(j_3, j_1, j_2)\omega_{n_3}^{j_3 k_3}.$$

Step 3: Twiddle factor multiplication

$$x_3(k_3, j_1, j_2) = x_2(k_3, j_1, j_2)\omega_{n_2 n_3}^{j_2 k_3}.$$

Step 4: Transposition

$$x_4(j_2, j_1, k_3) = x_3(k_3, j_1, j_2).$$

Step 5: $n_1 n_3$ individual n_2-point multicolumn FFTs

$$x_5(k_2, j_1, k_3) = \sum_{j_2=0}^{n_2-1} x_4(j_2, j_1, k_3)\omega_{n_2}^{j_2 k_2}.$$

Step 6: Twiddle factor multiplication

$$x_6(k_2, j_1, k_3) = x_5(k_2, j_1, k_3)\omega_n^{j_1 k_3}\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 7: Transposition

$$x_7(j_1, k_2, k_3) = x_6(k_2, j_1, k_3).$$

Step 8: $n_2 n_3$ individual n_1-point multicolumn FFTs

$$x_8(k_1, k_2, k_3) = \sum_{j_1=0}^{n_1-1} x_7(j_1, k_2, k_3)\omega_{n_1}^{j_1 k_1}.$$

Step 9: Transposition

$$y(k_3, k_2, k_1) = x_8(k_1, k_2, k_3).$$

The distinctive features of the nine-step FFT algorithm can be summarized as follows:

- Three multicolumn FFTs are performed in Steps 2, 5, and 8. The locality of the memory reference in the multicolumn FFT is high. Therefore, the nine-step FFT is suitable for cache-based processors because of the high performance which can be obtained with high hit rates in the cache memory.
- The matrix transposition takes place four times.

The program of the blocked nine-step FFT algorithm is shown in Listing 6.3. For extremely large FFTs, we should switch to a four-dimensional formulation and higher approaches.

Listing 6.3 Blocked nine-step FFT algorithm

```
      complex*16 x(n1,n2,n3),y(n3,n2,n1),u2(n3,n2),u3(n1,n2,n3)
      complex*16 ywork(n2+np,nb),zwork(n3+np,nb)
!
      do j=1,n2
        do ii=1,n1,nb
! Step 1: Blocked transposition
          do kk=1,n3,nb
            do i=ii,min(ii+nb-1,n1)
              do k=kk,min(kk+nb-1,n3)
```

```
                zwork(k,i-ii+1)=x(i,j,k)
              end do
            end do
          end do
! Step 2: n1*n2 individual n3-point multicolumn FFTs
          do i=ii,min(ii+nb-1,n1)
            call fft(zwork(1,i-ii+1),n3)
          end do
! Steps 3-4: Blocked twiddle factor multiplication and
!             transposition
          do k=1,n3
            do i=ii,min(ii+nb-1,n1)
              x(i,j,k)=zwork(k,i-ii+1)*u2(k,j)
            end do
          end do
        end do
      end do
      do k=1,n3
        do ii=1,n1,nb
          do jj=1,n2,nb
            do i=ii,min(ii+nb-1,n1)
              do j=jj,min(jj+nb-1,n2)
                ywork(j,i-ii+1)=x(i,j,k)
              end do
            end do
          end do
! Step 5: n1*n3 individual n2-point multicolumn FFTs
          do i=ii,min(ii+nb-1,n1)
            call fft(ywork(1,i-ii+1),n2)
          end do
! Steps 6-7: Blocked twiddle factor multiplication and
!             transposition
          do j=1,n2
            do i=ii,min(ii+nb-1,n1)
              x(i,j,k)=ywork(j,i-ii+1)*u3(i,j,k)
            end do
          end do
        end do
! Step 8: n2*n3 individual n1-point multicolumn FFTs
        do j=1,n2
          call fft(x(1,j,k),n1)
        end do
      end do
! Step 9: Blocked transposition
      do ii=1,n1,nb
        do jj=1,n2,nb
          do kk=1,n3,nb
            do i=ii,min(ii+nb-1,n1)
              do j=jj,min(jj+nb-1,n2)
                do k=kk,min(kk+nb-1,n3)
                  y(k,j,i)=x(i,j,k)
                end do
              end do
            end do
          end do
        end do
      end do
```

6.6 Recursive Six-Step FFT Algorithm

With the multicolumn FFTs in the six-step FFT algorithm described in Sect. 6.3, the Stockham FFT algorithm can be used. The Stockham FFT algorithm works well until the $\sqrt{n}$-point each column FFT exceeds the cache size. However, for extremely large FFTs (e.g., $n = 2^{40}$-point FFT), each $\sqrt{n}$-point column FFT is not small enough to fit into the L2 cache.

When each $\sqrt{n}$-point column FFT exceeds the cache size, the blocked six-step FFT should be used. This means that we can recursively use the blocked six-step FFT for each column FFT. We call the algorithm the recursive six-step FFT algorithm [28].

Listing 6.4 gives the pseudocode for the recursive six-step FFT algorithm. Here, a and b are the input array and the output array, respectively. The twiddle factors $\omega_{n_1 n_2}^{j_1 k_2}$ in Eq. (6.5) are stored in the array w. The parameter nb is the blocking parameter for the matrix transpositions. The Stockham FFT algorithm is used as the in-cache FFT in Chap. 2.

Listing 6.4 Recursive six-step FFT algorithm

```
      recursive subroutine recursive_fft(a,b,w,n)
      implicit real*8 (a-h,o-z)
      complex*16 a(*),b(*),w(*)
!
      if (n .le. cachesize) then
        call fft(a,b,n)
        return
      end if
! Step 0: Decompose n into n1 and n2
      call getn1n2(n,n1,n2)
! Step 1: Blocked transposition
      do ii=1,n1,nb
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              b(j+(i-1)*n2)=a(i+(j-1)*n1)
            end do
          end do
        end do
      end do
! Step 2: n1 individual n2-point multicolumn FFTs
      do i=1,n1
        call recursive_fft(b((i-1)*n2+1),a,w,n2)
      end do
! Step 3: Twiddle factor multiplication
      do i=1,n1*n2
        a(i)=a(i)*w(i)
      end do
! Step 4: Blocked transposition
      do jj=1,n2,nb
        do ii=1,n1,nb
          do j=jj,min(jj+nb-1,n2)
            do i=ii,min(ii+nb-1,n1)
              b(i+(j-1)*n1)=a(j+(i-1)*n2)
```

```
            end do
          end do
        end do
      end do
! Step 5: n2 individual n1-point multicolumn FFTs
      do j=1,n2
        call recursive_fft(b((j-1)*n1+1),a,w,n1)
      end do
! Step 6: Blocked transposition
      do ii=1,n1,nb
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              b(j+(i-1)*n2)=a(i+(j-1)*n1)
            end do
          end do
        end do
      end do
      return
      end
```

Listing 6.5 Blocked two-dimensional FFT based on row–column algorithm

```
      subroutine fft2d(x,n1,n2)
      implicit real*8 (a-h,o-z)
      complex*16 x(n1,n2),work(n2+np,nb)
! Step 1: n2 individual n1-point multicolumn FFTs
      do j=1,n2
        call fft(x(1,j),n1)
      end do
! Step 2: Blocked transposition
      do ii=1,n1,nb
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              work(j,i-ii+1)=x(i,j)
            end do
          end do
        end do
! Step 3: n1 individual n2-point multicolumn FFTs
        do i=ii,min(ii+nb-1,n1)
          call fft(work(1,i-ii+1),n2)
        end do
! Step 4: Blocked transposition
        do j=1,n2
          do i=ii,min(ii+nb-1,n1)
            x(i,j)=work(j,i-ii+1)
          end do
        end do
      end do
      return
      end
```

6.7 Blocked Multidimensional FFT Algorithms

6.7.1 Blocked Two-Dimensional FFT Algorithm

In the two-dimensional FFT algorithm based on the row–column algorithm, it is necessary to transpose the matrix twice, but the algorithm can be blocked in the same manner as the blocked six-step FFT. Listing 6.5 shows a blocked two-dimensional FFT algorithm based on the row–column algorithm.

Here, the parameters `nb` and `np` are the blocking parameter and padding parameter, respectively. The array `work` is the work array. In the blocked two-dimensional FFT algorithm, the array `work` of size $O(n_2)$ is sufficient for the work array. In other words, the input data is overwritten with the output data, which is an in-place algorithm.

6.7.2 Blocked Three-Dimensional FFT Algorithm

In the three-dimensional FFT algorithm, it is necessary to transpose the matrix three times, but the algorithm can be blocked in the same manner as the blocked six-step FFT and the blocked two-dimensional FFT. Listing 6.6 shows a blocked three-dimensional FFT algorithm.

Listing 6.6 Blocked three-dimensional FFT algorithm

```
      subroutine fft3d(x,n1,n2,n3)
      implicit real*8 (a-h,o-z)
      complex*16 x(n1,n2,n3),ywork(n2+np,nb),zwork(n3+np,nb)
! Step 1: n2*n3 individual n1-point multicolumn FFTs
      do k=1,n3
        do j=1,n2
          call fft(x(1,j,k),n1)
        end do
! Step 2: Blocked transposition
        do ii=1,n1,nb
          do i=ii,min(ii+nb-1,n1)
            do j=1,n2
              ywork(j,i-ii+1)=x(i,j,k)
            end do
          end do
! Step 3: n3*n1 individual n2-point multicolumn FFTs
          do i=ii,min(ii+nb-1,n1)
            call fft(ywork(1,i-ii+1),n2)
          end do
! Step 4: Blocked transposition
          do j=1,n2
            do i=ii,min(ii+nb-1,n1)
              x(i,j,k)=ywork(j,i-ii+1)
            end do
          end do
```

```
        end do
      end do
      do j=1,n2
        do ii=1,n1,nb
          do i=ii,min(ii+nb-1,n1)
            do k=1,n3
              zwork(k,i-ii+1)=x(i,j,k)
            end do
          end do
! Step 5: n1*n2 individual n3-point multicolumn FFTs
          do i=ii,min(ii+nb-1,n1)
            call fft(zwork(1,i-ii+1),n3)
          end do
! Step 6: Blocked transposition
          do k=1,n3
            do i=ii,min(ii+nb-1,n1)
              x(i,j,k)=zwork(k,i-ii+1)
            end do
          end do
        end do
      end do
      return
      end
```

Here, the parameters `nb` and `np` are the blocking parameter and padding parameter, respectively. The arrays `ywork` and `zwork` are the work arrays. In the blocked three-dimensional FFT algorithm, the array `ywork` of size $O(n_2)$ and the array `zwork` of size $O(n_3)$ are sufficient for the work arrays. In other words, the input data is overwritten with the output data, which is an in-place algorithm.

6.8 FFT Algorithms Suitable for Fused Multiply–Add (FMA) Instructions

6.8.1 Introduction

Until 1990s, floating-point addition was faster than floating-point multiplication on most processors. For this reason, FFT algorithms that reduced real number multiplication, e.g., the Winograd Fourier transform algorithm (WFTA) [30] and the prime factor algorithm (PFA) [3, 11], have been intensively studied. These algorithms show an advantage over processors that require more time for multiplication than addition. Today, floating-point multiplication is as fast as floating-point addition on the latest processors. Moreover, many processors have fused multiply–add (FMA) instructions.

Linzer and Feig [13, 14] have shown power-of-two FFT algorithms for FMA instructions. These FFT algorithms are based on the Cooley–Tukey FFT algorithm and the split-radix FFT algorithm. On the other hand, radix-2, 3, 4, and 5 FFT algorithms on computers with overlapping multiply–add instructions have been proposed

by Goedecker [6] and Karner et al. [10]. An implementation of a radix-6 FFT algorithm suitable for FMA instructions has been presented [21].

In this section, a radix-16 FFT algorithm suitable for FMA instructions [24] is described. Throughout this section, we use an FMA instruction, which computes $d = \pm a \pm bc$, where a, b, c, and d are floating-point registers. Also, we assume that an addition, a multiplication, or a multiply–add each requires one machine cycle on processors that have FMA instructions. We will call any of these computations a floating-point instruction, and assign a unit cost to each.

6.8.2 FFT Kernel

An FFT kernel [2, 6, 29] calculates the innermost part in a transformation, which has the form [6]

$$Z_{out}(k) = \sum_{j=0}^{P-1} Z_{in}(j)\Omega^j \omega_P^{jk} \tag{6.11}$$

for $0 \leq k \leq P-1$. The radix of the kernel is given by the prime factor P. Ω^j is the twiddle factor and $\omega_P = e^{-2\pi i/P}$.

In the radix-P FFT kernel, an input data $Z_{in}(j)$ multiplied by the twiddle factor Ω^j is performed with "short DFT" [17].

6.8.3 Goedecker's Technique

In this subsection, Goedecker's technique [6] is explained. To simplify the explanation, in the case of $P = 2$ in Eq. (6.11), that is, a radix-2 FFT kernel is shown as an example. Hereinafter, the real part of the array Z_{in} is denoted by zinr, the imaginary part by zini, and correspondingly for Z_{out}. The real part and imaginary part of the twiddle factor Ω^j are cr_j and ci_j, respectively.

The radix-2 FFT algorithm (decimation-in-time) is shown in Fig. 6.3. When executing this radix-2 FFT algorithm on processors with FMA instructions, it is necessary to perform two FMA instructions and two multiplications to compute u1 and v1, and is necessary to perform four additions to compute zoutr(0), zouti(0), zoutr(1), and zouti(1). As a result, the radix-2 FFT algorithm has eight floating-point instructions on processors with FMA instructions.

Goedecker's technique consists of repeated transformations of the expression:

$$ax + by \rightarrow a(x + (b/a)y), \tag{6.12}$$

where $a \neq 0$.

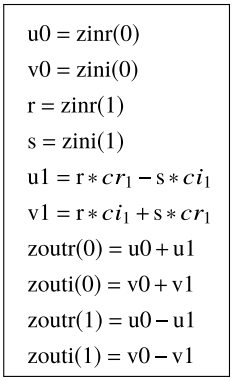

Fig. 6.3 Radix-2 FFT algorithm (decimation-in-time)

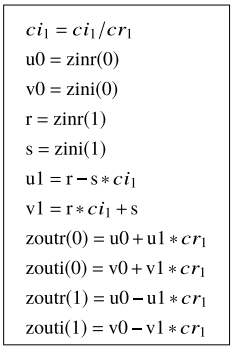

Fig. 6.4 Radix-2 FFT algorithm suitable for FMA instructions

Note that $cr_1 \neq 0$ in the radix-2 FFT algorithm. Applying repeated transformations of Eq. (6.12) to the radix-2 FFT algorithm, a radix-2 FFT algorithm suitable for FMA instructions is obtained.

The radix-2 FFT algorithm suitable for FMA instructions is shown in Fig. 6.4. In the radix-2 FFT algorithm suitable for FMA instructions, a table for twiddle factors of cr_1 and ci_1 is needed. Since these values can be computed in advance, the overhead of making the table is negligible. The radix-2 FFT algorithm suitable for FMA instructions has only six floating-point instructions on processors with FMA instruction. We can see that the multiply–add unit can be exploited in the radix-2 FFT algorithm suitable for FMA instructions.

6.8.4 Radix-16 FFT Algorithm

The radix-16 FFT algorithm (decimation-in-time) is shown in Figs. 6.5 and 6.6. The radix-16 FFT is split into the first step and the remaining part. The first step is the complex multiplication of $Z_{in}(j) \times \Omega^j$ $(0 \leq j \leq 15)$. Then 15 complex multiplications are necessary. We assume that a complex multiplication is done with

```
cos81 = cos(π/8)
cos82 = cos(2π/8) = √2/2
cos83 = cos(3π/8)
/* 15 complex multiplications
   by twiddle factors */
u0 = zinr(0)
v0 = zini(0)
r = zinr(1)
s = zini(1)
u1 = r*cr1 − s*ci1
v1 = r*ci1 + s*cr1
r = zinr(2)
s = zini(2)
u2 = r*cr2 − s*ci2
v2 = r*ci2 + s*cr2
r = zinr(3)
s = zini(3)
u3 = r*cr3 − s*ci3
v3 = r*ci3 + s*cr3
r = zinr(4)
s = zini(4)
u4 = r*cr4 − s*ci4
v4 = r*ci4 + s*cr4
r = zinr(5)
s = zini(5)
u5 = r*cr5 − s*ci5
v5 = r*ci5 + s*cr5
r = zinr(6)
s = zini(6)
u6 = r*cr6 − s*ci6
v6 = r*ci6 + s*cr6
r = zinr(7)
s = zini(7)
u7 = r*cr7 − s*ci7
v7 = r*ci7 + s*cr7
r = zinr(8)
s = zini(8)
u8 = r*cr8 − s*ci8
v8 = r*ci8 + s*cr8
r = zinr(9)
s = zini(9)
u9 = r*cr9 − s*ci9
v9 = r*ci9 + s*cr9
r = zinr(10)
s = zini(10)
u10 = r*cr10 − s*ci10
v10 = r*ci10 + s*cr10
r = zinr(11)
s = zini(11)
u11 = r*cr11 − s*ci11
v11 = r*ci11 + s*cr11
r = zinr(12)
s = zini(12)
u12 = r*cr12 − s*ci12
v12 = r*ci12 + s*cr12
r = zinr(13)
s = zini(13)
u13 = r*cr13 − s*ci13
v13 = r*ci13 + s*cr13
r = zinr(14)
s = zini(14)
u14 = r*cr14 − s*ci14
v14 = r*ci14 + s*cr14
r = zinr(15)
s = zini(15)
u15 = r*cr15 − s*ci15
v15 = r*ci15 + s*cr15
/* 16-point DFT */
r0 = u0 + u8
s0 = v0 + v8
r1 = u0 − u8
s1 = v0 − v8
r2 = u4 + u12
s2 = v4 + v12
r3 = v4 − v12
s3 = u12 − u4
r4 = u1 + u9
s4 = v1 + v9
r5 = u1 − u9
s5 = v1 − v9
r6 = u5 + u13
s6 = v5 + v13
r7 = v5 − v13
s7 = u13 − u5
r8 = u2 + u10
s8 = v2 + v10
r9 = u2 − u10
s9 = v2 − v10
r10 = u6 + u14
s10 = v6 + v14
r11 = v6 − v14
s11 = u14 − u6
r12 = u3 + u11
s12 = v3 + v11
r13 = u3 − u11
s13 = v3 − v11
r14 = u7 + u15
s14 = v7 + v15
r15 = v7 − v15
s15 = u15 − u7
u0 = r0 + r2
v0 = s0 + s2
u1 = r1 + r3
v1 = s1 + s3
```

Fig. 6.5 Radix-16 FFT algorithm (decimation-in-time)

u2 = r0 − r2	r = r12 − r14	zoutr(9) = r0 − r2
v2 = s0 − s2	s = s12 − s14	zouti(9) = s0 − s2
u3 = r1 − r3	u14 = $-cos_{82} * (r - s)$	zoutr(13) = r1 − r3
v3 = s1 − s3	v14 = $-cos_{82} * (r + s)$	zouti(13) = s1 − s3
u4 = r4 + r6	r = r13 − r15	r0 = u2 + u10
v4 = s4 + s6	s = s13 − s15	s0 = v2 + v10
r = r5 + r7	u15 = $-r * cos_{81} - s * cos_{83}$	r1 = u2 − u10
s = s5 + s7	v15 = $r * cos_{83} - s * cos_{81}$	s1 = v2 − v10
u5 = $r * cos_{81} + s * cos_{83}$	r0 = u0 + u8	r2 = u6 + u14
v5 = $-r * cos_{83} + s * cos_{81}$	s0 = v0 + v8	s2 = v6 + v14
r = r4 − r6	r1 = u0 − u8	r3 = v6 − v14
s = s4 − s6	s1 = v0 − v8	s3 = u14 − u6
u6 = $cos_{82} * (r + s)$	r2 = u4 + u12	zoutr(2) = r0 + r2
v6 = $cos_{82} * (s - r)$	s2 = v4 + v12	zouti(2) = s0 + s2
r = r5 − r7	r3 = v4 − v12	zoutr(6) = r1 + r3
s = s5 − s7	s3 = u12 − u4	zouti(6) = s1 + s3
u7 = $r * cos_{83} + s * cos_{81}$	zoutr(0) = r0 + r2	zoutr(10) = r0 − r2
v7 = $-r * cos_{81} + s * cos_{83}$	zouti(0) = s0 + s2	zouti(10) = s0 − s2
u8 = r8 + r10	zoutr(4) = r1 + r3	zoutr(14) = r1 − r3
v8 = s8 + s10	zouti(4) = s1 + s3	zouti(14) = s1 − s3
r = r9 + r11	zoutr(8) = r0 − r2	r0 = u3 + u11
s = s9 + s11	zouti(8) = s0 − s2	s0 = v3 + v11
u9 = $cos_{82} * (r + s)$	zoutr(12) = r1 − r3	r1 = u3 − u11
v9 = $cos_{82} * (s - r)$	zouti(12) = s1 − s3	s1 = v3 − v11
u10 = s8 − s10	r0 = u1 + u9	r2 = u7 + u15
v10 = r10 − r8	s0 = v1 + v9	s2 = v7 + v15
r = r9 − r11	r1 = u1 − u9	r3 = v7 − v15
s = s9 − s11	s1 = v1 − v9	s3 = u15 − u7
u11 = $-cos_{82} * (r - s)$	r2 = u5 + u13	zoutr(3) = r0 + r2
v11 = $-cos_{82} * (r + s)$	s2 = v5 + v13	zouti(3) = s0 + s2
u12 = r12 + r14	r3 = v5 − v13	zoutr(7) = r1 + r3
v12 = s12 + s14	s3 = u13 − u5	zouti(7) = s1 + s3
r = r13 + r15	zoutr(1) = r0 + r2	zoutr(11) = r0 − r2
s = s13 + s15	zouti(1) = s0 + s2	zouti(11) = s0 − s2
u13 = $r * cos_{83} + s * cos_{81}$	zoutr(5) = r1 + r3	zoutr(15) = r1 − r3
v13 = $-r * cos_{81} + s * cos_{83}$	zouti(5) = s1 + s3	zouti(15) = s1 − s3

Fig. 6.6 Radix-16 FFT algorithm (decimation-in-time, continued)

four real multiplications and two real additions. In the first step, since the ratio of multiplications to additions is two to one, the addition unit cannot be exploited on processors with FMA instructions. Then a 16-point DFT is performed in the remaining part. Since the radix-16 FFT has 24 real multiplications and 144 real additions in the remaining part, the multiply unit also cannot be exploited. As a result, the radix-16 FFT algorithm has 220 floating-point instructions on processors with FMA instructions.

We conclude that the radix-16 FFT algorithm is therefore not suitable for FMA instructions.

6.8.5 *Radix-16 FFT Algorithm Suitable for Fused Multiply–Add Instructions*

As we mentioned in the above section, the FMA unit cannot be exploited in the radix-16 FFT algorithm. We will make full use of the FMA unit to transform the radix-16 FFT.

Applying repeated transformations of Eq. (6.12) to the radix-16 FFT in Figs. 6.5 and 6.6, a radix-16 FFT algorithm suitable for FMA instructions is obtained. The radix-16 FFT algorithm suitable for FMA instructions is shown in Figs. 6.7 and 6.8. In the radix-16 FFT algorithm suitable for FMA instructions, a table for twiddle factors of $ci_1 \sim ci_{15}$, $cr_{31} \sim cr_{157}$, $cos_{81} \sim cos_{8381}$, and $cr_{181} \sim cr_{282}$ is needed. Since these values can be computed in advance, the overhead of making the table is negligible.

The radix-16 FFT algorithm suitable for FMA instructions has only 174 floating-point instructions on processors with FMA instructions. We can see that the FMA unit can be exploited in the radix-16 FFT algorithm from Figs. 6.7 and 6.8.

6.8.6 *Evaluation*

In order to evaluate the effectiveness of power-of-two FFT algorithms, we compare the number of floating-point instructions, loads, and stores.

We summarized the comparison of the number of floating-point instructions for power-of-two FFT algorithms with FMA instructions in Tables 6.1 and 6.2. The radix-16 FFT algorithm suitable for FMA instructions asymptotically saves approximately 21% of the floating-point instructions over the radix-16 FFT on processors that have FMA instructions. In comparison with the Linzer and Feig radix-4 and radix-8 FFT algorithm, the radix-16 FFT algorithm suitable for FMA instructions asymptotically saves approximately 1% of the floating-point instructions. On the other hand, the radix-16 FFT algorithm suitable for FMA instructions asymptotically increases approximately 2% of the floating-point instructions over the Linzer and Feig split-radix FFT algorithm.

```
/* twiddle factors can be
    computed in advance */
ci1 = ci1/cr1
ci2 = ci2/cr2
ci3 = ci3/cr3
ci4 = ci4/cr4
ci5 = ci5/cr5
ci6 = ci6/cr6
ci7 = ci7/cr7
ci8 = ci8/cr8
ci9 = ci9/cr9
ci10 = ci10/cr10
ci11 = ci11/cr11
ci12 = ci12/cr12
ci13 = ci13/cr13
ci14 = ci14/cr14
ci15 = ci15/cr15
cr31 = cr3/cr1
cr51 = cr5/cr1
cr62 = cr6/cr2
cr73 = cr7/cr3
cr91 = cr9/cr1
cr102 = cr10/cr2
cr113 = cr11/cr3
cr124 = cr12/cr4
cr135 = cr13/cr5
cr146 = cr14/cr6
cr157 = cr15/cr7
cos81 = cos(π/8)
cos82 = cos(2π/8) = √2/2
cos83 = cos(3π/8)
cos8381 = cos83/cos81
cr181 = cr1 * cos81
cr182 = cr1 * cos82
cr282 = cr2 * cos82
/* 15 complex multiplications
    by twiddle factors */
u0 = zinr(0)
v0 = zini(0)
r = zinr(1)
s = zini(1)
u1 = r − s * ci1
v1 = r * ci1 + s
r = zinr(2)
s = zini(2)
u2 = r − s * ci2
v2 = r * ci2 + s
r = zinr(3)
s = zini(3)
u3 = r − s * ci3
v3 = r * ci3 + s
r = zinr(4)
s = zini(4)
u4 = r − s * ci4
v4 = r * ci4 + s
r = zinr(5)
s = zini(5)
u5 = r − s * ci5
v5 = r * ci5 + s
r = zinr(6)
s = zini(6)
u6 = r − s * ci6
v6 = r * ci6 + s
r = zinr(7)
s = zini(7)
u7 = r − s * ci7
v7 = r * ci7 + s
r = zinr(8)
s = zini(8)
u8 = r − s * ci8
v8 = r * ci8 + s
r = zinr(9)
s = zini(9)
u9 = r − s * ci9
v9 = r * ci9 + s
r = zinr(10)
s = zini(10)
u10 = r − s * ci10
v10 = r * ci10 + s
r = zinr(11)
s = zini(11)
u11 = r − s * ci11
v11 = r * ci11 + s
r = zinr(12)
s = zini(12)
u12 = r − s * ci12
v12 = r * ci12 + s
r = zinr(13)
s = zini(13)
u13 = r − s * ci13
v13 = r * ci13 + s
r = zinr(14)
s = zini(14)
u14 = r − s * ci14
v14 = r * ci14 + s
r = zinr(15)
s = zini(15)
u15 = r − s * ci15
v15 = r * ci15 + s
/* 16-point DFT */
r0 = u0 + u8 * cr8
s0 = v0 + v8 * cr8
r1 = u0 − u8 * cr8
s1 = v0 − v8 * cr8
r2 = u4 + u12 * cr124
s2 = v4 + v12 * cr124
r3 = v4 − v12 * cr124
s3 = u12 * cr124 − u4
r4 = u1 + u9 * cr91
s4 = v1 + v9 * cr91
r5 = u1 − u9 * cr91
s5 = v1 − v9 * cr91
r6 = u5 + u13 * cr135
s6 = v5 + v13 * cr135
r7 = v5 − v13 * cr135
s7 = u13 * cr135 − u5
r8 = u2 + u10 * cr102
s8 = v2 + v10 * cr102
r9 = u2 − u10 * cr102
s9 = v2 − v10 * cr102
r10 = u6 + u14 * cr146
s10 = v6 + v14 * cr146
```

Fig. 6.7 Radix-16 FFT algorithm suitable for FMA instructions

$r11 = v6 - v14 * cr_{146}$
$s11 = u14 * cr_{146} - u6$
$r12 = u3 + u11 * cr_{113}$
$s12 = v3 + v11 * cr_{113}$
$r13 = u3 - u11 * cr_{113}$
$s13 = v3 - v11 * cr_{113}$
$r14 = u7 + u15 * cr_{157}$
$s14 = v7 + v15 * cr_{157}$
$r15 = v7 - v15 * cr_{157}$
$s15 = u15 * cr_{157} - u7$
$u0 = r0 + r2 * cr_4$
$v0 = s0 + s2 * cr_4$
$u1 = r1 + r3 * cr_4$
$v1 = s1 + s3 * cr_4$
$u2 = r0 - r2 * cr_4$
$v2 = s0 - s2 * cr_4$
$u3 = r1 - r3 * cr_4$
$v3 = s1 - s3 * cr_4$
$u4 = r4 + r6 * cr_{51}$
$v4 = s4 + s6 * cr_{51}$
$r = r5 + r7 * cr_{51}$
$s = s5 + s7 * cr_{51}$
$u5 = r + s * cos_{8381}$
$v5 = -r * cos_{8381} + s$
$r = r4 - r6 * cr_{51}$
$s = s4 - s6 * cr_{51}$
$u6 = r + s$
$v6 = s - r$
$r = r5 - r7 * cr_{51}$
$s = s5 - s7 * cr_{51}$
$u7 = r * cos_{8381} + s$
$v7 = -r + s * cos_{8381}$
$u8 = r8 + r10 * cr_{62}$
$v8 = s8 + s10 * cr_{62}$
$r = r9 + r11 * cr_{62}$
$s = s9 + s11 * cr_{62}$
$u9 = r + s$
$v9 = s - r$
$u10 = s8 - s10 * cr_{62}$
$v10 = r10 * cr_{62} - r8$
$r = r9 - r11 * cr_{62}$
$s = s9 - s11 * cr_{62}$
$u11 = -(r - s)$
$v11 = -(r + s)$
$u12 = r12 + r14 * cr_{73}$
$v12 = s12 + s14 * cr_{73}$
$r = r13 + r15 * cr_{73}$
$s = s13 + s15 * cr_{73}$
$u13 = r * cos_{8381} + s$
$v13 = -r + s * cos_{8381}$
$r = r12 - r14 * cr_{73}$
$s = s12 - s14 * cr_{73}$
$u14 = -(r - s)$
$v14 = -(r + s)$
$r = r13 - r15 * cr_{73}$
$s = s13 - s15 * cr_{73}$
$u15 = -r - s * cos_{8381}$
$v15 = r * cos_{8381} - s$
$r0 = u0 + u8 * cr_2$
$s0 = v0 + v8 * cr_2$
$r1 = u0 - u8 * cr_2$
$s1 = v0 - v8 * cr_2$
$r2 = u4 + u12 * cr_{31}$
$s2 = v4 + v12 * cr_{31}$
$r3 = v4 - v12 * cr_{31}$
$s3 = u12 * cr_{31} - u4$
$zoutr(0) = r0 + r2 * cr_1$
$zouti(0) = s0 + s2 * cr_1$
$zoutr(4) = r1 + r3 * cr_1$
$zouti(4) = s1 + s3 * cr_1$
$zoutr(8) = r0 - r2 * cr_1$
$zouti(8) = s0 - s2 * cr_1$
$zoutr(12) = r1 - r3 * cr_1$
$zouti(12) = s1 - s3 * cr_1$
$r0 = u1 + u9 * cr_{282}$
$s0 = v1 + v9 * cr_{282}$
$r1 = u1 - u9 * cr_{282}$
$s1 = v1 - v9 * cr_{282}$
$r2 = u5 + u13 * cr_{31}$
$s2 = v5 + v13 * cr_{31}$
$r3 = v5 - v13 * cr_{31}$
$s3 = u13 * cr_{31} - u5$
$zoutr(1) = r0 + r2 * cr_{181}$
$zouti(1) = s0 + s2 * cr_{181}$
$zoutr(5) = r1 + r3 * cr_{181}$
$zouti(5) = s1 + s3 * cr_{181}$
$zoutr(9) = r0 - r2 * cr_{181}$
$zouti(9) = s0 - s2 * cr_{181}$
$zoutr(13) = r1 - r3 * cr_{181}$
$zouti(13) = s1 - s3 * cr_{181}$
$r0 = u2 + u10 * cr_2$
$s0 = v2 + v10 * cr_2$
$r1 = u2 - u10 * cr_2$
$s1 = v2 - v10 * cr_2$
$r2 = u6 + u14 * cr_{31}$
$s2 = v6 + v14 * cr_{31}$
$r3 = v6 - v14 * cr_{31}$
$s3 = u14 * cr_{31} - u6$
$zoutr(2) = r0 + r2 * cr_{182}$
$zouti(2) = s0 + s2 * cr_{182}$
$zoutr(6) = r1 + r3 * cr_{182}$
$zouti(6) = s1 + s3 * cr_{182}$
$zoutr(10) = r0 - r2 * cr_{182}$
$zouti(10) = s0 - s2 * cr_{182}$
$zoutr(14) = r1 - r3 * cr_{182}$
$zouti(14) = s1 - s3 * cr_{182}$
$r0 = u3 + u11 * cr_{282}$
$s0 = v3 + v11 * cr_{282}$
$r1 = u3 - u11 * cr_{282}$
$s1 = v3 - v11 * cr_{282}$
$r2 = u7 + u15 * cr_{31}$
$s2 = v7 + v15 * cr_{31}$
$r3 = v7 - v15 * cr_{31}$
$s3 = u15 * cr_{31} - u7$
$zoutr(3) = r0 + r2 * cr_{181}$
$zouti(3) = s0 + s2 * cr_{181}$
$zoutr(7) = r1 + r3 * cr_{181}$
$zouti(7) = s1 + s3 * cr_{181}$
$zoutr(11) = r0 - r2 * cr_{181}$
$zouti(11) = s0 - s2 * cr_{181}$
$zoutr(15) = r1 - r3 * cr_{181}$
$zouti(15) = s1 - s3 * cr_{181}$

Fig. 6.8 Radix-16 FFT algorithm suitable for FMA instructions (continued)

Table 6.1 Number of floating-point instructions for FFT algorithms of n points with FMA instructions

Algorithm	Floating-point instructions
Linzer and Feig radix-4 [14]	$(11/4)n\log_2 n - (13/6)n + (8/3)$
Linzer and Feig radix-8 [14]	$(11/4)n\log_2 n - (57/28)n + (16/7)$
Linzer and Feig split-radix [14]	$(8/3)n\log_2 n - (16/9)n + 2 - (2/9)(-1)^{\log_2 n}$
Radix-16	$(55/16)n\log_2 n - (241/60)n + (64/15)$
Radix-16 for FMA instructions	$(87/32)n\log_2 n - (241/120)n + (32/15)$

Table 6.2 Number of floating-point instructions for FFT algorithms of n points with FMA instructions

n	Linzer and Feig radix-4 [14]	Linzer and Feig radix-8 [14]	Linzer and Feig split-radix [14]	Radix-16	Radix-16 for FMA instructions
8		52	52		
16	144		144	160	144
32			372		
64	920	928	912		
128			2164		
256	5080		5008	6016	5056
512		11632	11380		
1024	25944		25488		
2048			56436		
4096	126296	126832	123792	152512	125408

Table 6.3 shows the comparison of the number of loads, stores, and floating-point instructions for FFT kernels with FMA instructions. In calculating the number of loads and stores, we assume that enough registers are available to perform an entire FFT kernel in registers without using any intermediate stores and loads.

In Table 6.4, the asymptotic number of loads, stores, and floating-point instructions for FFT algorithms is given. If the radix-16 FFT suitable for FMA instructions is being computed on a machine that has enough floating-point registers to perform an entire radix-16 kernel in registers, then the radix-16 FFT algorithm suitable for FMA instructions will use fewer loads and stores than the Linzer and Feig split-radix FFT algorithm. In view of a ratio of floating-point instructions to memory operations, the radix-16 FFT algorithm suitable for FMA instructions is more advantageous than the Linzer and Feig split-radix FFT algorithm.

Table 6.3 Loads, stores, and floating-point instructions for FFT kernels with FMA instructions. The number of loads does not include loading twiddle factors

Algorithm	Loads	Stores	Floating-point instructions
Linzer and Feig radix-4 [14]	8	8	22
Linzer and Feig radix-8 [14]	16	16	66
Linzer and Feig split-radix [14]	8	8	16
Radix-16	32	32	220
Radix-16 for FMA instructions	32	32	174

Table 6.4 Number of loads, stores, and floating-point instructions divided by $n \log_2 n$ used by FFT algorithms to compute an n-point DFT

Algorithm	Loads	Stores	Floating-point instructions
Linzer and Feig radix-4 [14]	1	1	11/4
Linzer and Feig radix-8 [14]	2/3	2/3	11/4
Linzer and Feig split-radix [14]	4/3	4/3	8/3
Radix-16	1/2	1/2	55/16
Radix-16 for FMA instructions	1/2	1/2	87/32

6.9 FFT Algorithms for SIMD Instructions

6.9.1 Introduction

Today, a number of processors have SIMD instructions. These instructions provide substantial speedup for digital signal processing applications. Efficient FFT implementations with SIMD instructions have also been investigated thoroughly [4, 5, 12, 15, 16, 19, 20, 25, 26]. A difference between the different implementations in complex multiplication using SIMD instructions has been shown [18].

In this section, we describe an implementation of parallel one-dimensional FFT using SIMD instructions.

6.9.2 Vectorization of FFT Kernels Using Intel SSE3 Instructions

The Intel streaming SIMD extensions 3 (SSE3) were introduced into the IA-32 architecture [8]. These extensions are designed to enhance the performance of IA-32 processors.

The most direct way to use the SSE3 is to insert the assembly language instructions inline into the source code. However, this can be time-consuming and tedious. Instead, Intel provides for easy implementation through the use of API extension sets referred to as intrinsics [9]. We used the SSE3 intrinsics to access SIMD hardware. An example of complex multiplication using the SSE3 intrinsics [25] is shown in Listing 6.7.

Listing 6.7 An example of complex multiplication using SSE3 intrinsics [25]

```
#include <pmmintrin.h>

static __inline __m128d ZMUL(__m128d a, __m128d b)
{
    __m128d ar, ai;

    ar = _mm_movedup_pd(a);        /* ar = [a.r a.r] */
    ar = _mm_mul_pd(ar, b);        /* ar = [a.r*b.r a.r*b.i] */
    ai = _mm_unpackhi_pd(a, a);    /* ai = [a.i a.i] */
    b = _mm_shuffle_pd(b, b, 1);   /* b = [b.i b.r] */
    ai = _mm_mul_pd(ai, b);        /* ai = [a.i*b.i a.i*b.r] */

    return _mm_addsub_pd(ar, ai); /* [a.r*b.r-a.i*b.i
                                     a.r*b.i+a.i*b.r] */
}
```

The `__m128d` data type in Listing 6.7 is supported by the SSE3 intrinsics. The `__m128d` data type holds two packed double-precision floating-point values. In the complex multiplication, the `__m128d` data type is used as a double-precision complex data type.

The inline function `ZMUL` in Listing 6.7 can be used to multiply two double-precision complex values. To add two double-precision complex values, we can use the intrinsic function `__mm_add_pd` in Listing 6.7. To vectorize FFT kernels, the SSE3 intrinsics and the inline function `ZMUL` can be used. An example of vectorized radix-2 FFT kernel using the SSE3 intrinsics [25] is shown in Listing 6.8.

Listing 6.8 An example of vectorized radix-2 FFT kernel using SSE3 intrinsics [25]

```
#include <pmmintrin.h>

__m128d ZMUL(__m128d a, __m128d b);

void fft_vec(double *a, double *b, double *w, int m, int l)
{
  int i, i0, i1, i2, i3, j;
  __m128d t0, t1, w0;
```

```
  for (j = 0; j < l; j++) {
    w0 = _mm_load_pd(&w[j << 1]);
    for (i = 0; i < m; i++) {
      i0 = (i << 1) + (j * m << 1); i1 = i0 + (m * l << 1);
      i2 = (i << 1) + (j * m << 2); i3 = i2 + (m << 1);
      t0 = _mm_load_pd(&a[i0]); t1 = _mm_load_pd(&a[i1]);
      _mm_store_pd(&b[i2], _mm_add_pd(t0, t1));
      _mm_store_pd(&b[i3], ZMUL(w0, _mm_sub_pd(t0, t1)));
    }
  }
}
```

6.9.3 *Vectorization of FFT Kernels Using Intel AVX-512 Instructions*

The Intel Advanced Vector Extensions 512 (AVX-512) [8] is a 512-bit single-instruction multiple data (SIMD) instruction sets on the Intel Xeon Phi processor. AVX-512 supports 512-bit wide SIMD registers (ZMM0–ZMM31).

The Intel C/C++ and Fortran compilers also use Intel AVX-512 to support automatic vectorization. In this subsection, we use automatic vectorization.

An example of a vectorizable radix-2 FFT kernel [26] is shown in Listing 6.9. Here, arrays `a` and `b` are the input array and the output array, respectively. The twiddle factors [2] are stored in array `w`. The problem size n corresponds to $\mathtt{m} \times \mathtt{l} \times 2$. For the Intel Xeon Phi processor, memory movement is optimal when the data starting address lies on 64-byte boundaries. Thus, we specified the directive "`!dir$ attributed align: 64`" for the arrays.

In the radix-2 FFT kernel, the innermost loop lengths are varied from 1 to $n/2$ for n-point FFTs during $\log_2 n$ stages. For the first stage of the radix-2 FFT kernel in Listing 6.9, the innermost loop length is 1. In this case, the double-nested loop can

Table 6.5 Real inner-loop operations for radix-2, 4, 8, and 16 double-precision complex FFT kernels based on the Stockham FFT algorithm

	Radix-2	Radix-4	Radix-8	Radix-16
Loads	4	8	16	32
Stores	4	8	16	32
Multiplications	4	12	32	84
Additions	6	22	66	174
Byte/Flop ratio	6.400	3.765	2.612	1.984

be collapsed into a single-nested loop to expand the innermost loop length, as shown in Listing 6.10.

Listing 6.9 An example of a vectorizable radix-2 FFT kernel [26]

```
      subroutine fft(a,b,w,m,l)
      complex*16 a(m,l,*),b(m,2,*),w(*)
      complex*16 c0,c1
      do j=1,l
        do i=1,m
          c0=a(i,j,1)
          c1=a(i,j,2)
          b(i,1,j)=c0+c1
          b(i,2,j)=w(j)*(c0-c1)
        end do
      end do
      return
      end
```

Listing 6.10 First stage of a vectorizable radix-2 FFT kernel [26]

```
      subroutine fft1st(a,b,w,l)
      complex*16 a(l,*),b(2,*),w(*)
      complex*16 c0,c1
      do j=1,l
        c0=a(j,1)
        c1=a(j,2)
        b(1,j)=c0+c1
        b(2,j)=w(j)*(c0-c1)
      end do
      return
      end
```

We use the radix-2, 4, 8, and 16 Stockham FFT algorithm for in-cache FFTs. Table 6.5 shows the real inner-loop operations for radix-2, 4, 8, and 16 double-precision complex FFT kernels. In view of the Byte/Flop ratio, the radix-16 FFT is preferable to the radix-2, 4, and 8 FFTs [24]. Although higher radix FFTs require more floating-point registers to hold intermediate results, the Intel Xeon Phi processor has 32 ZMM 512-bit registers. A power-of-two-point FFT (more than or equal to 64-point FFT) can be performed by a combination of radix-8 and radix-16 steps containing at most three radix-8 steps. In other words, power-of-two FFTs can be performed as a length $n = 2^p = 8^q 16^r$ ($p \geq 6,\ 0 \leq q \leq 3,\ r \geq 0$).

References

1. Bailey, D.H.: FFTs in external or hierarchical memory. J. Supercomput. **4**, 23–35 (1990)
2. Brigham, E.O.: The Fast Fourier Transform and Its Applications. Prentice-Hall, Englewood Cliffs, NJ (1988)
3. Burrus, C.S., Eschenbacher, P.W.: An in-place, in-order prime factor FFT algorithm. IEEE Trans. Acoust. Speech Signal Process. **ASSP-29**, 806–817 (1981)

4. Franchetti, F., Karner, H., Kral, S., Ueberhuber, C.W.: Architecture independent short vector FFTs. In: Proceedings of 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001), vol. 2, pp. 1109–1112 (2001)
5. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**, 216–231 (2005)
6. Goedecker, S.: Fast radix 2, 3, 4, and 5 kernels for fast Fourier transformations on computers with overlapping multiply-add instructions. SIAM J. Sci. Comput. **18**, 1605–1611 (1997)
7. Hegland, M.: A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. Numer. Math. **68**, 507–547 (1994)
8. Intel Corporation: Intel architecture instruction set extensions and future features programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf (2018)
9. Intel Corporation: Intel C++ compiler 19.0 developer guide and reference. https://software.intel.com/sites/default/files/19.0_U3_CPP_Compiler_DGR.pdf (2019)
10. Karner, H., Auer, M., Ueberhuber, C.W.: Multiply-add optimized FFT kernels. Math. Models Methods Appl. Sci. **11**, 105–117 (2001)
11. Kolba, D.P., Parks, T.W.: A prime factor FFT algorithm using high-speed convolution. IEEE Trans. Acoust. Speech Signal Process. **ASSP-25**, 281–294 (1977)
12. Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C.W.: SIMD vectorization of straight line FFT code. In: Proceedings of 9th International Euro-Par Conference (Euro-Par 2003). Lecture Notes in Computer Science, vol. 2790, pp. 251–260. Springer, Berlin (2003)
13. Linzer, E., Feig, E.: Modified FFTs for fused multiply-add architectures. Math. Comput. **60**, 347–361 (1993)
14. Linzer, E.N., Feig, E.: Implementation of efficient FFT algorithms on fused multiply-add architectures. IEEE Trans. Signal Process. **41**, 93–107 (1993)
15. McFarlin, D.S., Arbatov, V., Franchetti, F., Püschel, M.: Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets. In: Proceedings of 25th International Conference on Supercomputing (ICS'11), pp. 265–274 (2011)
16. Nadehara, K., Miyazaki, T., Kuroda, I.: Radix-4 FFT implementation using SIMD multimedia instructions. In: Proceedings of 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'99), vol. 4, pp. 2131–2134 (1999)
17. Nussbaumer, H.J.: Fast Fourier Transform and Convolution Algorithms, updated edn. Springer-Verlag, New York (1982). Second corrected and updated edition
18. Popovici, D.T., Franchetti, F., Low, T.M.: Mixed data layout kernels for vectorized complex arithmetic. In: Proceedings of 2017 IEEE High Performance Extreme Computing Conference (HPEC 2017) (2017)
19. Püschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: code generation for DSP transforms. Proc. IEEE **93**, 232–275 (2005)
20. Rodriguez, P.: A radix-2 FFT algorithm for modern single instruction multiple data (SIMD) architectures. In: Proceedings of 2002 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2002), vol. 3, pp. 3220–3223 (2002)
21. Takahashi, D.: A new radix-6 FFT algorithm suitable for multiply-add instruction. In: Proceedings of 2000 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2000), vol. 6, pp. 3343–3346 (2000)
22. Takahashi, D.: A blocking algorithm for parallel 1-D FFT on shared-memory parallel computers. In: Proceedings of 6th International Conference on Applied Parallel Computing (PARA 2002). Lecture Notes in Computer Science, vol. 2367, pp. 380–389. Springer, Berlin (2002)
23. Takahashi, D.: A parallel 1-D FFT algorithm for the Hitachi SR8000. Parallel Comput. **29**, 679–690 (2003)
24. Takahashi, D.: A radix-16 FFT algorithm suitable for multiply-add instruction based on Goedecker method. In: Proceedings of 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003), vol. 2, pp. 665–668 (2003)

25. Takahashi, D.: An implementation of parallel 1-D FFT using SSE3 instructions on dual-core processors. In: Proceedings of 8th International Workshop on State of the Art in Scientific Computing (PARA 2006). Lecture Notes in Computer Science, vol. 4699, pp. 1178–1187. Springer, Berlin (2007)
26. Takahashi, D.: An implementation of parallel 2-D FFT using Intel AVX instructions on multi-core processors. In: Proceedings of 12th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2012), Part II. Lecture Notes in Computer Science, vol. 7440, pp. 197–205. Springer, Berlin (2012)
27. Takahashi, D., Boku, T., Sato, M.: A blocking algorithm for parallel 1-D FFT on clusters of PCs. In: Proceedings of 8th International Euro-Par Conference (Euro-Par 2002). Lecture Notes in Computer Science, vol. 2400, pp. 691–700. Springer, Berlin (2002)
28. Takahashi, D., Uno, A., Yokokawa, M.: An implementation of parallel 1-D FFT on the K computer. In: Proceedings of 2012 IEEE 14th International Conference on High Performance Computing and Communications (HPCC-2012), pp. 344–350 (2012)
29. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia, PA (1992)
30. Winograd, S.: On computing the discrete Fourier transform. Math. Comput. **32**, 175–199 (1978)

Chapter 7
Parallel FFT Algorithms for Shared-Memory Parallel Computers

Abstract This chapter presents parallel FFT algorithms for shared-memory parallel computers. First, the implementation of parallel one-dimensional FFT on shared-memory parallel computers is described. Next, optimizing parallel FFTs for many-core processors and its performance are explained.

Keywords Shared-memory parallel computers · OpenMP · Manycore processors

7.1 Implementation of Parallel One-Dimensional FFT on Shared-Memory Parallel Computers

7.1.1 Introduction

OpenMP [9] has emerged as the standard for shared-memory parallel programming. The OpenMP Application Program Interface (API) provides programmers with a simple way to develop parallel applications for shared-memory parallel computers. Parallel FFT algorithms on shared-memory parallel computers have been well studied [1, 12, 16, 17].

Many FFT algorithms work well when data sets fit into a cache. When a problem exceeds the cache size, however, the performance of these FFT algorithms decreases dramatically. One goal for large FFTs is to minimize the number of cache misses. A recursive algorithm is very good at improving the use of the cache. Thus, some previously proposed FFT algorithms [3, 6] use a recursive approach.

In this section, we describe an OpenMP implementation of a recursive algorithm for computing large one-dimensional FFTs on shared-memory parallel computers.

Listing 7.1 A recursive three-step FFT algorithm

```
      recursive subroutine recursive_fft(a,temp,w,n)
      complex*16 a(*),temp(*),w(*)
      if (n .le. cachesize) then
        call fft(a,temp,w,n)
        return
      end if
! Step 1: n1 simultaneous n2-point multirow FFTs with
```


D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_7

```
!          twiddle factor multiplication
      do i=1,n/2
        temp(i)=a(i)+a(i+n/2)
        temp(i+n/2)=(a(i)-a(i+n/2))*w(i)
      end do
! Step 2: n2 individual n1-point multicolumn FFTs
      do j=1,2
        call recursive_fft(temp((j-1)*(n/2)+1),a,w(n/2+1),n/2)
      end do
! Step 3: Transposition
      do i=1,n/2
        a(2*i-1)=temp(i)
        a(2*i)=temp(i+n/2)
      end do
      return
      end
```

Listing 7.2 An OpenMP implementation of a recursive three-step FFT algorithm

```
      subroutine parallel_fft(a,temp,w,n)
      complex*16 a(*),temp(*),w(*)
      if (n .le. cachesize) then
        call fft(a,temp,w,n)
        return
      end if
!$omp parallel
!$omp do
      do i=1,n/2
        temp(i)=a(i)+a(i+n/2)
        temp(i+n/2)=(a(i)-a(i+n/2))*w(i)
      end do
!$omp do
      do j=1,2
        call recursive_fft(temp((j-1)*(n/2)+1),a((j-1)*(n/2)+1),
     &                     w(n/2+1),n/2)
      end do
!$omp do
      do i=1,n/2
        a(2*i-1)=temp(i)
        a(2*i)=temp(i+n/2)
      end do
!$omp end parallel
      return
      end
```

7.1.2 A Recursive Three-Step FFT Algorithm

The DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1, \tag{7.1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n has factors n_1 and n_2 ($n = n_1 \times n_2$), then the indices j and k can be expressed as

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{7.2}$$

We can define x and y as two-dimensional arrays (in column-major order):

$$x(j) = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{7.3}$$

$$y(k) = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{7.4}$$

Substituting the indices j and k in Eq. (7.1) with those in Eq. (7.2), and using the relation of $n = n_1 \times n_2$, we can derive the following equation:

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{7.5}$$

This derivation leads to a following three-step FFT algorithm [14]:

Step 1: n_1 simultaneous n_2-point multirow FFTs with twiddle factor multiplication

$$x_1(j_1, k_2) = \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2}.$$

Step 2: n_2 individual n_1-point multicolumn FFTs

$$x_2(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_1(j_1, k_2) \omega_{n_1}^{j_1 k_1}.$$

Step 3: Transposition

$$y(k_2, k_1) = x_2(k_1, k_2).$$

The distinctive features of the three-step FFT algorithm can be summarized as follows:

- One multirow FFT and one multicolumn FFT are performed in Steps 1 and 2, respectively.
- A matrix transposition takes place just once in Step 3.

For each n_1-point multicolumn FFT in Step 2 can be performed recursively. We will call it a recursive three-step FFT algorithm. Listing 7.1 gives the pseudocode for the recursive three-step FFT algorithm where $n_1 = n/2$ and $n_2 = 2$. Here the arrays `a` and `temp` are the input/output array and work array, respectively. The twiddle factors $\omega_{n_1 n_2}^{j_1 k_2}$ in Step 1 are stored in array `w`. We can use the padding technique [10] for each recursive step.

7.1.3 Parallelization of Recursive Three-Step FFT

We parallelized the recursive three-step FFT by using OpenMP directives. Since the recursive three-step FFT has enough outermost parallelism, it is not necessary to use the nested parallelism. An OpenMP implementation of the recursive three-step FFT algorithm is shown in Listing 7.2. The parallelized subroutine `parallel_fft()` shown in Listing 7.2 calls the recursive subroutine `recursive_fft()` shown in Listing 7.1.

Although the range of `do j=1,2` loop shown in Listing 7.2 is two, the loop length can be extended to $n_2 \geq 2$ shown in Eq. (7.5). Since we use the radix-8 FFT algorithm [2] for the OpenMP implementation of the recursive three-step FFT, the range of `j` is 8.

Each directive of OpenMP may cause overhead. In order to reduce fork/join overhead, three parallel regions can be fused shown in Listing 7.2. Thus, the parallelized subroutine `parallel_fft()` has only one `parallel` directive.

7.2 Optimizing Parallel FFTs for Manycore Processors

7.2.1 Introduction

The Intel Xeon Phi processor has emerged as an important computational accelerator in high-performance computing systems. Knights Landing processor [11] is the second-generation Intel Xeon Phi product. Cori, a system equipped with the Knights Landing processor, placed the 12th in the TOP500 list of November 2018 [15].

Parallel FFTs on manycore processors have been implemented [5, 7, 13]. Both vectorization and parallelization are of particular importance with respect to Intel Xeon Phi processors. We vectorized FFT kernels using the Intel Advanced Vector Extensions 512 (AVX-512) instructions. Furthermore, the proposed approach makes use of the parallelism of the Intel Xeon Phi processor by loop collapse.

In this section, we describe an implementation of a parallel one-dimensional real fast Fourier transform (FFT) on Intel Xeon Phi processors. We implemented the parallel one-dimensional real FFT on an Intel Xeon Phi processor.

Listing 7.3 Example of matrix transposition with cache blocking

```
      complex*16 x(n1,n2),y(n2,n1)
!$omp parallel do private(i,j,jj)
      do ii=1,n1,nb
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              y(j,i)=x(i,j)
            end do
          end do
        end do
      end do
```

Listing 7.4 Parallelization of six-step FFT with loop collapse

```
      complex*16 x(n1,n2),y(n2,n1),w(n1,n2)
!$omp parallel
! Step 1: Transposition
!$omp do collapse(2)private(i,j,jj)
      do ii=1,n1,nb
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              y(j,i)=x(i,j)
            end do
          end do
        end do
      end do
! Step 2: n1 individual n2-point multicolumn FFTs
!$omp do
      do i=1,n1
        call fft(y(1,i),n2)
      end do
! Steps 3-4: Twiddle factor multiplication and
! transposition
!$omp do collapse(2) private(i,ii,j)
      do jj=1,n2,nb
        do ii=1,n1,nb
          do j=jj,min(jj+nb-1,n2)
            do i=ii,min(ii+nb-1,n1)
              x(i,j)=y(j,i)*w(i,j)
            end do
          end do
        end do
      end do
! Step 5: n2 individual n1-point multicolumn FFTs
!$omp do
      do j=1,n2
        call fft(x(1,j),n1)
      end do
! Step 6: Transposition
!$omp do collapse(2) private(i,j,jj)
      do ii=1,n1,nb
        do jj=1,n2,nb
          do i=ii,min(ii+nb-1,n1)
            do j=jj,min(jj+nb-1,n2)
              y(j,i)=x(i,j)
            end do
          end do
        end do
      end do
!$omp end parallel
```

7.2.2 Parallelization of Six-Step FFT

When we parallelize the six-step FFT by using OpenMP, the outermost loop of each FFT step is distributed across the cores. In Listing 7.3, the outermost loop length

Table 7.1 Specifications of the platform

Platform	Intel Xeon Phi processor
Number of cores	68
CPU type	Intel Xeon Phi 7250 Knights Landing 1.4 GHz
L1 Cache (per core)	I-Cache: 32 KB D-Cache: 32 KB
L2 Cache	1 MB (shared between two cores)
Main Memory	MCDRAM 16 GB + DDR4-2400 96 GB
OS	Linux 3.10.0-327.22.2.el7.xppsl_1.4.1.3272.x86_64

may not have sufficient parallelism for manycore processors. For an $n = 2^{18}$-point FFT, we assume `n1=n2=512` and `nb=8`. In this case, the outermost loop length is 64 (=512/8), which is less than the number of cores (68) on the Intel Xeon Phi 7250 processor. Thus, insufficient parallelism always leads to load imbalance.

A loop collapsing makes the length of a loop long by collapsing nested loops into a single-nested loop. When using the OpenMP collapse clause, which is supported from OpenMP 3.0 [9], the loops must be perfectly nested. Since the outermost nested loop in Listing 7.3 is a perfectly nested loop, it can be collapsed into a single-nested loop. Listing 7.4 shows the parallelization of six-step FFT with loop collapse. Arrays `x` and `y` are the input array and the output array, respectively. The twiddle factors ($\omega_{n_1 n_2}^{j_1 k_2}$) in Eq. (6.5) are stored in the array `w`. By using the OpenMP collapse clause, the parallelism of the outermost loop can be expanded from 64 to 4,096 (=64 × 64) for the $n = 2^{18}$-point FFT.

The blocked six-step FFT algorithm described in section 6.4 improves performance by utilizing the cache memory more effectively. In the blocked six-step FFT, the multicolumn FFTs and the transpositions are combined to further reuse data in the cache memory by strip mining. Since the loops in the blocked six-step FFT are not perfectly nested, the loops cannot be collapsed. We emphasize parallelism rather than cache utilization and use the implementation of Listing 7.4.

7.2.3 Performance Results

To evaluate the implemented parallel one-dimensional real FFT, referred to as FFTE (version 6.2alpha), we compared its performance with that of the FFTW (version 3.3.6-pl1) [4] and the Intel Math Kernel Library (MKL, version 2017 Update 1) [7]. The FFTW and the MKL support the AVX-512 instructions.

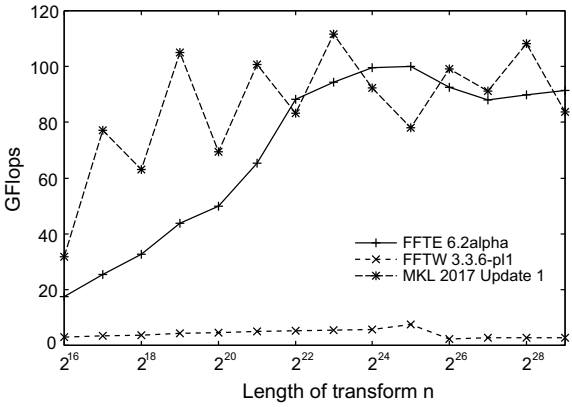

Fig. 7.1 Performance of one-dimensional real FFTs (Intel Xeon Phi 7250, 272 threads)

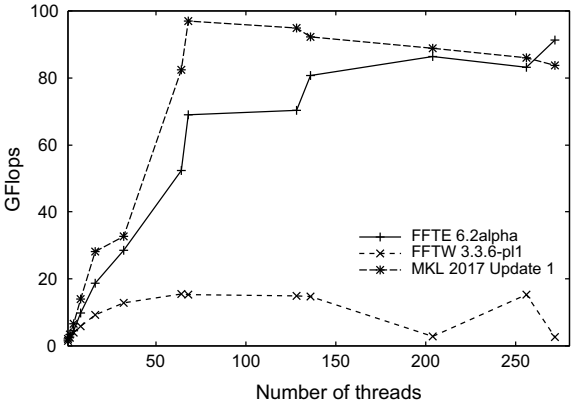

Fig. 7.2 Performance of one-dimensional real FFTs (Intel Xeon Phi 7250, $n = 2^{29}$)

The elapsed times obtained from 10 executions of real-to-complex FFTs were averaged. The input is in normal order, and the output is in a conjugate-even order. The real FFTs were performed on double-precision real data, and the table for twiddle factors was prepared in advance. In the FFTW, the "measure" planner was used. The specifications of the platform used herein are shown in Table 7.1. We note that hyper-threading (HT) [8] was enabled on the Intel Xeon Phi 7250.

The compiler used was the Intel Fortran Compiler (`ifort`, version 17.0.1.132) for the FFTE and the MKL. The compiler options used were specified as `ifort -O3 -xMIC-AVX512 -qopenmp`. The compiler used was the Intel C Compiler (`icc`, version 17.0.1.132) for the FFTW, with the compiler options `icc -O3 -xMIC-AVX512 -qopenmp`. The compiler option `-O3` specifies to optimize for maximum speed and enable more aggressive optimizations. The compiler option `-xMIC-AVX512` specifies to generate AVX-512 Foundation instructions, AVX-512 Conflict Detection instructions, AVX-512 Exponential and Reciprocal instructions, and AVX-512 Prefetch instructions. The compiler option `-qopenmp` specifies to enable the compiler to generate multi-threaded code based on the OpenMP direc-

tives. The executions were performed in "flat mode" and "quadrant mode". The environment variable `KMP_AFFINITY=granularity=fine,balanced` was specified. All programs were run in MCDRAM.

Figures 7.1 and 7.2 compare GFlops of the FFTE, the FFTW, and the MKL. The GFlops values are each based on $5n \log_2 n$ for a transform of size $n = 2^m$. As shown in Figs. 7.1 and 7.2, the FFTE is faster than the FFTW. As shown in Fig. 7.1, the FFTE is faster than the MKL for the cases of $n = 2^{22}$, $2^{24} \leq n \leq 2^{25}$ and $n = 2^{29}$ on 272 threads. Figure 7.2 indicates that hyper-threading is effective for the FFTE.

References

1. Bailey, D.H.: FFTs in external or hierarchical memory. J. Supercomput. **4**, 23–35 (1990)
2. Bergland, G.D.: A fast Fourier transform algorithm using base 8 iterations. Math. Comput. **22**, 275–279 (1968)
3. Frigo, M., Johnson, S.G.: FFTW: an adaptive software architecture for the FFT. In: Proceedings of 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '98), vol. 3, pp. 1381–1384 (1998)
4. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**, 216–231 (2005)
5. Hascoet, J., Nezan, J.F., Ensor, A., de Dinechin, B.D.: Implementation of a fast Fourier transform algorithm onto a manycore processor. In: Proceedings of 2015 Conference on Design and Architectures for Signal and Image Processing (DASIP 2015) (2015)
6. Hegland, M.: A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. Numer. Math. **68**, 507–547 (1994)
7. Intel math kernel library developer reference (2017). https://software.intel.com/sites/default/files/managed/ff/c8/mkl-2017-developer-reference-c_0.pdf
8. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-threading technology architecture and microarchitecture. Intel Technol. J. **6**, 1–11 (2002)
9. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf
10. Panda, P.R., Nakamura, H., Dutt, N.D., Nicolau, A.: Augmenting loop tiling with data alignment for improved cache performance. IEEE Trans. Comput. **48**, 142–149 (1999)
11. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights Landing: second-generation Intel Xeon Phi product. IEEE Micro **36**, 34–46 (2016)
12. Swarztrauber, P.N.: Multiprocessor FFTs. Parallel Comput. **5**, 197–210 (1987)
13. Takahashi, D.: An implementation of parallel 1-D real FFT on Intel Xeon Phi processors. In: Proceedings of 17th International Conference on Computational Science and Its Applications (ICCSA 2017), Part I, Lecture Notes in Computer Science, vol. 10404, pp. 401–410. Springer International Publishing (2017)
14. Takahashi, D., Sato, M., Boku, T.: An OpenMP implementation of parallel FFT and its performance on IA-64 processors. In: Proceedings of International Workshop on OpenMP Applications and Tools (WOMPAT 2003), Lecture Notes in Computer Science, vol. 2716, pp. 99–108. Springer (2003)
15. TOP500 Supercomputer Sites. http://www.top500.org/
16. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia, PA (1992)
17. Wadleigh, K.R.: High performance FFT algorithms for cache-coherent multiprocessors. Int. J. High Perform. Comput. Appl. **13**, 163–171 (1999)

Chapter 8
Parallel FFT Algorithms for Distributed-Memory Parallel Computers

Abstract This chapter presents parallel FFT algorithms for distributed-memory parallel computers. First, implementation of parallel FFTs in distributed-memory parallel computers and computation–communication overlap for parallel one-dimensional FFT are explained. Next, parallel three-dimensional FFT using two-dimensional decomposition is described. Then, the optimization of all-to-all communication on multicore cluster systems is explained. Finally, parallel one-dimensional FFT in a GPU cluster is described.

Keywords Distributed-memory parallel computers · Computation–communication overlap · Two-dimensional decomposition

8.1 Implementation of Parallel FFTs in Distributed-Memory Parallel Computers

In this section, we describe implementations of parallel FFTs in distributed-memory parallel computers. When computing FFTs in distributed-memory parallel computers, it is necessary to distribute data among MPI processes. There are roughly two methods of data distribution, block distribution, and cyclic distribution. In most parallel FFT libraries [1–4, 7, 9], the block distribution is used. Since it is possible to reduce the number of all-to-all communication by using the cyclic distribution in parallel one-dimensional FFT [44, 49, 50], a parallel one-dimensional FFT using cyclic distribution is described in Sect. 8.1.2.

8.1.1 Parallel One-Dimensional FFT Using Block Distribution

Parallel one-dimensional FFT algorithms in distributed-memory parallel computers have been investigated extensively [12, 17, 19, 20, 27, 29, 36, 39, 48, 51, 52].

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7_8

Let N have factors N_1 and N_2 ($N = N_1 \times N_2$). The original one-dimensional array $x(N)$ can be defined as a two-dimensional array $x(N_1, N_2)$ (in column-major order). On a distributed-memory parallel computer having P MPI processes, the array $x(N_1, N_2)$ is distributed along the second dimension N_2. If N_2 is divisible by P, then each MPI process has distributed data of size N/P. We introduce the notation $\hat{N}_r \equiv N_r/P$ and denote the corresponding index as $\hat{J}_r$, which indicates that the data along J_r are distributed across all P MPI processes. The subscript r specifies that this index belongs to dimension r. The distributed array is represented as $\hat{x}(N_1, \hat{N}_2)$. At MPI process m, the local index $\hat{J}_r(m)$ corresponds to the global index as the block distribution:

$$J_r = m \times \hat{N}_r + \hat{J}_r(m), \quad 0 \le m \le P-1, \quad 1 \le r \le 2. \tag{8.1}$$

In order to illustrate all-to-all communication, it is convenient to decompose N_i into the two dimensions $\tilde{N}_i$ and P_i, where $\tilde{N}_i \equiv N_i/P_i$. Although P_i is the same as P, we use the subscript i to specify that these indices belong to dimension i.

Starting with the initial data $\hat{x}(N_1, \hat{N}_2)$, the parallel one-dimensional FFT algorithm based on the six-step FFT can be performed in the following steps:

Step 1: Rearrangement

$$\begin{aligned}\hat{x_1}(\tilde{J}_1, \hat{J}_2, P_1) &= \hat{x}(\tilde{J}_1, P_1, \hat{J}_2)\\ &\equiv \hat{x}(J_1, \hat{J}_2).\end{aligned}$$

Step 2: All-to-all communication

$$\hat{x_2}(\hat{J}_1, \tilde{J}_2, P_2) = \hat{x_1}(\tilde{J}_1, \hat{J}_2, P_1).$$

Step 3: Transposition

$$\begin{aligned}\hat{x_3}(J_2, \hat{J}_1) &= \hat{x_2}(\hat{J}_1, J_2)\\ &\equiv \hat{x_2}(\hat{J}_1, \tilde{J}_2, P_2).\end{aligned}$$

Step 4: (N_1/P) individual N_2-point multicolumn FFTs

$$\hat{x_4}(K_2, \hat{J}_1) = \sum_{J_2=0}^{N_2-1} \hat{x_3}(J_2, \hat{J}_1)\omega_{N_2}^{J_2K_2}.$$

Step 5: Rearrangement and twiddle factor multiplication

$$\begin{aligned}\hat{x_5}(\tilde{K}_2, \hat{J}_1, P_2) &= \hat{x_4}(\tilde{K}_2, P_2, \hat{J}_1)\omega_N^{\hat{J}_1}(\tilde{K}_2 + \tilde{N}_2P_2)\\ &\equiv \hat{x_4}(K_2, \hat{J}_1)\omega_N^{\hat{J}_1K_2}.\end{aligned}$$

Step 6: All-to-all communication

$$\hat{x_6}(\hat{K}_2, \tilde{J}_1, P_1) = \hat{x_5}(\tilde{K}_2, \hat{J}_1, P_2).$$

Step 7: Transposition

$$\begin{aligned}\hat{x_7}(J_1, \hat{K}_2) &= \hat{x_6}(\hat{K}_2, J_1)\\ &\equiv \hat{x_6}(\hat{K}_2, \tilde{J}_1, P_1).\end{aligned}$$

Step 8: (N_2/P) individual N_1-point multicolumn FFTs

$$\hat{x_8}(K_1, \hat{K_2}) = \sum_{J_1=0}^{N_1-1} \hat{x_7}(J_1, \hat{K_2}) \omega_{N_1}^{J_1 K_1}.$$

Step 9: Rearrangement

$$\hat{x_9}(\tilde{K_1}, \hat{K_2}, P_1) = \hat{x_8}(\tilde{K_1}, P_1, \hat{K_2}) \equiv \hat{x_8}(K_1, \hat{K_2}).$$

Step 10: All-to-all communication

$$\hat{x_{10}}(\hat{K_1}, \tilde{K_2}, P_2) = \hat{x_9}(\tilde{K_1}, \hat{K_2}, P_1).$$

Step 11: Transposition

$$\hat{y}(K_2, \hat{K_1}) = \hat{x_{10}}(\hat{K_1}, K_2) \equiv \hat{x_{10}}(\hat{K_1}, \tilde{K_2}, P_2).$$

In Steps 4 and 8, multicolumn FFTs are performed along the local dimensions. In Step 5, the computation is accompanied by a local transposition for data rearrangement and a twiddle factor multiplication. We combined some of the operations with data movement, as in Step 5, in order to increase the efficiency in using the memory bandwidth.

The distinctive features of the parallel one-dimensional FFT algorithm based on the six-step FFT are summarized as follows:

- $\sqrt{N}/P$ individual $\sqrt{N}$-point multicolumn FFTs are performed in Steps 4 and 8 for the case of $N_1 = N_2 = \sqrt{N}$.
- All-to-all communications occur three times.

If both N_1 and N_2 are divisible by P, then the workload on each MPI process is uniform.

8.1.2 Parallel One-Dimensional FFT Using Cyclic Distribution

Let N have two factors N_1 and N_2 ($N = N_1 \times N_2$). The original one-dimensional array $x(N)$ can be defined as a two-dimensional array $x(N_1, N_2)$ (in column-major order). On a distributed-memory parallel computer having P MPI processes, the array $x(N_1, N_2)$ is distributed along the first dimension N_1. If N_1 is divisible by P, then each MPI process has distributed data of size N/P. We introduce the notation $\hat{N_r} \equiv N_r/P$ and denote the corresponding index as $\hat{J_r}$ which indicates that the data along J_r are distributed across all P MPI processes. The subscript r specifies that this index belongs to dimension r. The distributed array is represented as $\hat{x}(\hat{N_1}, N_2)$. At MPI process m, the local index $\hat{J_r}(m)$ corresponds to the global index as the cyclic distribution:

$$J_r = \hat{J_r}(m) \times P + m, \quad 0 \le m \le P-1, \quad 1 \le r \le 2. \tag{8.2}$$

In order to illustrate the all-to-all communication it is convenient to decompose N_i into two dimensions $\tilde{N}_i$ and P_i, where $\tilde{N}_i \equiv N_i/P_i$. Although P_i is the same as P, we use the subscript i to specify that these indices belong to dimension i.

Starting with the initial data $\hat{x}(\hat{N}_1, N_2)$, the parallel one-dimensional FFT algorithm based on the six-step FFT can be performed in the following steps:

Step 1: Transposition

$$\hat{x_1}(J_2, \hat{J}_1) = \hat{x}(\hat{J}_1, J_2).$$

Step 2: (N_1/P) individual N_2-point multicolumn FFTs

$$\hat{x_2}(K_2, \hat{J}_1) = \sum_{J_2=0}^{N_2-1} \hat{x_1}(J_2, \hat{J}_1)\omega_{N_2}^{J_2K_2}.$$

Step 3: Twiddle factor multiplication and transposition

$$\begin{aligned}\hat{x_3}(\hat{J}_1, P_2, \tilde{K}_2) &\equiv \hat{x_3}(\hat{J}_1, K_2)\\ &= \hat{x_2}(K_2, \hat{J}_1)\omega_{N_1N_2}^{\hat{J}_1K_2}.\end{aligned}$$

Step 4: Rearrangement

$$\hat{x_4}(\hat{J}_1, \tilde{K}_2, P_2) = \hat{x_3}(\hat{J}_1, P_2, \tilde{K}_2).$$

Step 5: All-to-all communication

$$\hat{x_5}(\tilde{J}_1, \hat{K}_2, P_1) = \hat{x_4}(\hat{J}_1, \tilde{K}_2, P_2).$$

Step 6: Transposition

$$\begin{aligned}\hat{x_6}(J_1, \hat{K}_2) &\equiv \hat{x_6}(P_1, \tilde{J}_1, \hat{K}_2)\\ &= \hat{x_5}(\tilde{J}_1, \hat{K}_2, P_1).\end{aligned}$$

Step 7: (N_2/P) individual N_1-point multicolumn FFTs

$$\hat{x_7}(K_1, \hat{K}_2) = \sum_{J_1=0}^{N_1-1} \hat{x_6}(J_1, \hat{K}_2)\omega_{N_1}^{J_1K_1}.$$

Step 8: Transposition

$$\hat{y}(\hat{K}_2, K_1) = \hat{x_7}(K_1, \hat{K}_2).$$

In Steps 2 and 7, multicolumn FFTs are performed along the local dimensions. In Step 3, the computation is accompanied by a local transposition for data rearrangement and a twiddle factor multiplication. Step 4 is a local transposition for data rearrangement. We combined some of the operations with data movement, as in Step 3, in order to increase the efficiency in using the memory bandwidth.

The distinctive features of the first parallel algorithm can be summarized as follows:

- $\sqrt{N}/P$ individual $\sqrt{N}$-point multicolumn FFTs are performed in Steps 2 and 7 for the case of $N_1 = N_2 = \sqrt{N}$.
- All-to-all communication occurs just once. Moreover, the input data x and the output data y are both normal order.

If both N_1 and N_2 are divisible by P, the workload on each MPI process is uniform.

8.1.3 Parallel Two-Dimensional FFT in Distributed-Memory Parallel Computers

Parallel two-dimensional FFT algorithms in distributed-memory parallel computers have been proposed [16, 17].

Let $N = N_1 \times N_2$. On a distributed-memory parallel computer having P MPI processes, a two-dimensional array $x(N_1, N_2)$ is distributed along the second dimension N_2. If N_2 is divisible by P, then the distributed array is represented as $\hat{x}(N_1, \hat{N}_2)$. At MPI process m, the local index $\hat{J}_r(m)$ corresponds to the global index as the block distribution:

$$J_r = m \times \hat{N}_r + \hat{J}_r(m), \quad 0 \le m \le P-1, \quad 1 \le r \le 2. \tag{8.3}$$

Starting with the initial data $\hat{x}(N_1, \hat{N}_2)$, the parallel two-dimensional FFT can be performed according to the following steps:

Step 1: (N_2/P) individual N_1-point multicolumn FFTs

$$\hat{x}_1(K_1, \hat{J}_2) = \sum_{J_1=0}^{N_1-1} \hat{x}(J_1, \hat{J}_2)\omega_{N_1}^{J_1 K_1}.$$

Step 2: Transposition

$$\begin{aligned}\hat{x}_2(\hat{J}_2, \tilde{K}_1, P_1) &\equiv \hat{x}_2(\hat{J}_2, K_1) \\ &= \hat{x}_1(K_1, \hat{J}_2).\end{aligned}$$

Step 3: All-to-all communication

$$\hat{x}_3(\tilde{J}_2, \hat{K}_1, P_2) = \hat{x}_2(\hat{J}_2, \tilde{K}_1, P_1).$$

Step 4: Rearrangement

$$\begin{aligned}\hat{x}_4(J_2, \hat{K}_1) &\equiv \hat{x}_4(\tilde{J}_2, P_2, \hat{K}_1) \\ &= \hat{x}_3(\tilde{J}_2, \hat{K}_1, P_2).\end{aligned}$$

Step 5: (N_1/P) individual N_2-point multicolumn FFTs

$$\hat{x}_5(K_2, \hat{K}_1) = \sum_{J_2=0}^{N_2-1} \hat{x}_4(J_2, \hat{K}_1)\omega_{N_2}^{J_2 K_2}.$$

Step 6: Transposition

$$\begin{aligned}\hat{x}_6(\hat{K}_1, \tilde{K}_2, P_2) &\equiv \hat{x}_6(\hat{K}_1, K_2) \\ &= \hat{x}_5(K_2, \hat{K}_1).\end{aligned}$$

Step 7: All-to-all communication

$$\hat{x}_7(\tilde{K}_1, \hat{K}_2, P_1) = \hat{x}_6(\hat{K}_1, \tilde{K}_2, P_2).$$

Step 8: Rearrangement

$$\begin{aligned}\hat{y}(K_1, \hat{K}_2) &\equiv \hat{y}(\tilde{K}_1, P_1, \hat{K}_2) \\ &= \hat{x}_7(\tilde{K}_1, \hat{K}_2, P_1).\end{aligned}$$

8.1.4 Parallel Three-Dimensional FFT in Distributed-Memory Parallel Computers

Parallel three-dimensional FFT algorithms in distributed-memory parallel computers have been proposed [14, 16, 37, 43, 46].

Let $N = N_1 \times N_2 \times N_3$. On a distributed-memory parallel computer having P MPI processes, a three-dimensional array $x(N_1, N_2, N_3)$ is distributed along the third dimension N_3. If N_3 is divisible by P, then the distributed array is represented as $\hat{x}(N_1, N_2, \hat{N}_3)$. At MPI process m, the local index $\hat{J}_r(m)$ corresponds to the global index as the block distribution:

$$J_r = m \times \hat{N}_r + \hat{J}_r(m), \quad 0 \le m \le P-1, \quad 1 \le r \le 3. \tag{8.4}$$

Starting with the initial data $\hat{x}(N_1, N_2, \hat{N}_3)$, the parallel three-dimensional FFT can be performed according to the following steps:

Step 1: $N_2 \cdot (N_3/P)$ individual N_1-point multicolumn FFTs

$$\hat{x_1}(K_1, J_2, \hat{J}_3) = \sum_{J_1=0}^{N_1-1} \hat{x}(J_1, J_2, \hat{J}_3)\omega_{N_1}^{J_1 K_1}.$$

Step 2: Transposition

$$\hat{x_2}(J_2, K_1, \hat{J}_3) = \hat{x_1}(K_1, J_2, \hat{J}_3).$$

Step 3: $N_1 \cdot (N_3/P)$ individual N_2-point multicolumn FFTs

$$\hat{x_3}(K_2, K_1, \hat{J}_3) = \sum_{J_2=0}^{N_2-1} \hat{x_2}(J_2, K_1, \hat{J}_3)\omega_{N_2}^{J_2 K_2}.$$

Step 4: Rearrangement

$$\begin{aligned}\hat{x_4}(\tilde{K}_1, K_2, \hat{J}_3, P_1) &= \hat{x_3}(K_2, \tilde{K}_1, P_1, \hat{J}_3) \\ &\equiv \hat{x_3}(K_2, K_1, \hat{J}_3).\end{aligned}$$

Step 5: All-to-all communication

$$\hat{x_5}(\hat{K}_1, K_2, \tilde{J}_3, P_3) = \hat{x_4}(\tilde{K}_1, K_2, \hat{J}_3, P_1).$$

Step 6: Transposition

$$\begin{aligned}\hat{x_6}(J_3, \hat{K}_1, K_2) &= \hat{x_5}(\hat{K}_1, K_2, J_3) \\ &\equiv \hat{x_5}(\hat{K}_1, K_2, \tilde{J}_3, P_3).\end{aligned}$$

Step 7: $(N_1/P) \cdot N_2$ individual N_3-point multicolumn FFTs

$$\hat{x_7}(K_3, \hat{K}_1, K_2) = \sum_{J_3=0}^{N_3-1} \hat{x_6}(J_3, \hat{K}_1, K_2)\omega_{N_3}^{J_3 K_3}.$$

Step 8: Transposition

$$\begin{aligned}\hat{x_8}(\hat{K}_1, K_2, \tilde{K}_3, P_3) &\equiv \hat{x_8}(\hat{K}_1, K_2, K_3) \\ &= \hat{x_7}(K_3, \hat{K}_1, K_2).\end{aligned}$$

Step 9: All-to-all communication

$$\hat{x_9}(\tilde{K_1}, K_2, \hat{K_3}, P_1) = \hat{x_8}(\hat{K_1}, K_2, \tilde{K_3}, P_3).$$

Step 10: Rearrangement

$$\begin{aligned}\hat{y}(K_1, K_2, \hat{K_3}) &\equiv \hat{y}(\tilde{K_1}, P_1, K_2, \hat{K_3}) \\ &= \hat{x_9}(\tilde{K_1}, K_2, \hat{K_3}, P_1).\end{aligned}$$

8.2 Computation–Communication Overlap for Parallel One-Dimensional FFT

8.2.1 Introduction

Several FFT libraries with automatic tuning have been proposed, for example, FFTW [27], SPIRAL [26, 39], and UHFFT [33]. FFTW uses a planner to adapt to the hardware to maximize performance and generates highly optimized straight-line FFT code blocks called codelets. SPIRAL generates optimized codes for digital signal processing (DSP) transforms and uses a formal framework for generating efficient MPI algorithms [15].

Further, we need to select optimal parameters according to the computational environment and problem size. Parallel FFTs in distributed-memory parallel computers require intensive all-to-all communication, which affects the performance of parallel FFTs. Overlapping the computation and all-to-all communication is one of the open issues in parallel FFTs. Parallel FFT algorithms in distributed-memory parallel computers with computation–communication overlap have been proposed [17, 19, 42].

Doi and Negishi presented overlapping methods of all-to-all communication and FFT algorithms for torus-connected massively parallel supercomputers [19]. However, their implementation does not optimize computation–communication overlap automatically. Song and Hollingsworth implemented an auto-tuning of parallel three-dimensional FFT for computation–communication overlap [42]. Their approach requires the non-blocking `MPI_Ialltoall` operation described in the MPI-3.0 standard [5].

On the other hand, a computation–communication overlap method using OpenMP based on introducing a communication thread has been presented [28, 32]. This method does not require non-blocking collective operations in the MPI-3.0 standard. We use this method for computation–communication overlap. The same method has been shown in the context of shared memory with multiple threads, cores, and sockets [38].

In this section, we describe an automatic tuning of computation–communication overlap for parallel one-dimensional FFT. The implementation is based on the parallel one-dimensional FFT described in Sect. 8.1.1. We have implemented an

automatic tuning facility for selecting the optimal parameters of the computation–communication overlap, the radices, and the block size.

Listing 8.1 Computation-communication overlap [28]

```
 1: !$omp parallel
 2: !$omp master
 3:       MPI communication
 4: !$omp end master
 5: !$omp do schedule(dynamic)
 6:       do i=1,n
 7:         Computation
 8:       end do
 9: !$omp do
10:       do i=1,n
11:         Computation using the result of communication
12:       end do
13: !$omp end parallel
```

8.2.2 Computation–Communication Overlap

MPI asynchronous communication [5] is widely used for computation–communication overlap. This approach requires the non-blocking `MPI_Ialltoall` operation in the MPI-3.0 standard for computation–communication overlap on parallel FFTs. On the other hand, a computation–communication overlap method by introducing a communication thread with OpenMP has been presented [28, 32]. We use this method for computation–communication overlap.

Computation–communication overlap [28] is shown in Listing 8.1. MPI communication is performed on the master thread in line 3. There is no barrier synchronization between lines 4 and 5. Thus, computation in lines 6–8 can be overlapped with MPI communication in line 3. In this case, the computation in lines 6–8 are performed on threads other than the master thread by using the OpenMP clause `!$omp do schedule(dynamic)`. If the MPI communication in line 3 finishes before the computation in lines 6–8, the master thread can join the computation. Implicit barrier synchronization is performed between lines 8 and 9. Then, computation in lines 10–12 is performed after completion of the MPI communication in line 3.

By dividing the computation and communication in Listing 8.1, it is possible to overlap in a pipelined manner [32]. Figure 8.1 shows the pipelined computation–communication overlap. In Fig. 8.1, `NDIV` denotes the number of divisions for computation–communication overlap.

8.2.3 Automatic Tuning of Parallel One-Dimensional FFT

When tuning a parallel one-dimensional FFT, the following are the three main performance parameters:

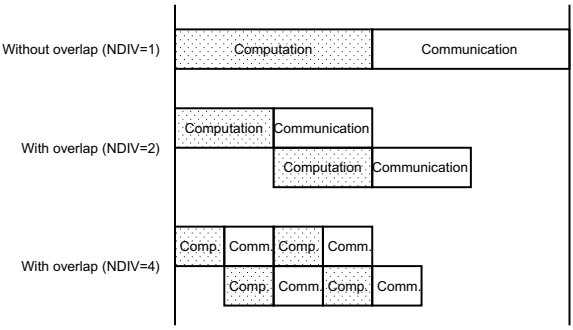

Fig. 8.1 Pipelined computation–communication overlap [48]

(1) the number of divisions `NDIV` for computation–communication overlap,
(2) radices (N_1 and N_2), and
(3) block sizes (`NB`).

By searching for these performance parameters, it is possible to further improve the performance of the parallel one-dimensional FFT. Note that (1) is a parameter related to the MPI inter-process communication, and (2) and (3) are parameters related to performance within the MPI process.

8.2.3.1 The Number of Divisions for Computation–Communication Overlap

When we increase the number of divisions for computation–communication overlap, the overlap ratio also increases. On the other hand, the performance of all-to-all communication decreases owing to the splitting of the message size. Thus, a trade-off exists between the overlap ratio and performance of all-to-all communication. The default overlapping parameter of the original FFTE 6.1alpha is `NDIV=4`. In our implementation, the overlapping parameter `NDIV` is varied with 1, 2, 4, 8, and 16.

8.2.3.2 Radices

In the six-step FFT algorithm, the number of data N is decomposed into $N = N_1 \times N_2$ to calculate N_1 individual N_2-point FFTs and N_2 individual N_1-point FFTs, respectively. Let us call the radices N_1 and N_2. The values of N_1 and N_2 can be chosen arbitrarily as long as $N = N_1 \times N_2$ is satisfied. However, in GPU clusters, when P is the number of MPI processes, it is necessary to satisfy $N_1,\ N_2 \geq P$.

Usually, N_1 and N_2 are chosen to be $N_1 \approx N_2 \approx \sqrt{N}$, but we can choose N_1 and N_2 so that the performance is the highest. If N and P are powers of two, N_1 is varied with $P,\ 2P,\ \ldots,\ \sqrt{N}$, and then $N_2 = N/N_1$. In this case, even if all combinations of N_1 and N_2 are tried, the search space is $\log_2(\sqrt{N}/P)$.

8.2.3.3 Block Sizes

In the six-step FFT algorithm, it is necessary to transpose the matrix, but it is known that transposition of this matrix can be performed efficiently by performing cache blocking. At this time, the optimum block size `NB` depends on the problem size, cache size, and other factors.

In this implementation, the block size `NB` is limited to powers of two and is varied as 4, 8, 16, 32, and 64. Although the optimum block size `NB` is not necessarily a power of two, the automatic tuning method is considered to be effective even when the block size `NB` is other than a power of two. An automatic tuning method of parallel one-dimensional FFT is shown in Fig. 8.2. In this code, the `GCD` function calculates the greatest common divisor. We assume that both N_1/P and N_2/P are divisible by `NDIV`. Otherwise, the divided message sizes become unequal.

Fig. 8.2 Automatic tuning of parallel one-dimensional FFT [48]

```
min_time = DBL_MAX;
for (k = 2; k <= 6; k++) {
  NB = 2^k;
  for (j = log2(P); j <= log2(√N); j++) {
    N1 = 2^j;
    N2 = N/N1;
    for (i = 1; i <= 16; i++) {
      NDIV=GCD(i, GCD(N1/P, N2/P));
      MPI_Barrier(MPI_COMM_WORLD);
      start = MPI_Wtime();
      for (count = 0; count < ITER_NUM; count++) {
        Parallel_1D_FFT(...);
      }
      MPI_Barrier(MPI_COMM_WORLD);
      end = MPI_Wtime();
    }
    if (end − start < min_time) {
      M1 = N1;
      M2 = N2;
      MB = NB;
      MDIV = NDIV;
    }
  }
}
N1 = M1;
N2 = M2;
NB = MB;
NDIV = MDIV;
```

8.2.4 Performance Results

In order to evaluate the parallel 1-D FFT with automatic tuning, we compared its performance against that of the FFTE 6.1alpha [3] and that of the FFTE 6.1alpha with automatic tuning (AT). We averaged the elapsed times obtained from 10 executions of complex forward FFTs.

The performance was measured on the Fujitsu PRIMEHPC FX100 at Nagoya University. The specifications for the Fujitsu PRIMEHPC FX100 are listed in Table 8.1.

In the experiment, we used from 1 to 512 nodes. The parallel one-dimensional FFT programs were run on 1 to 512 MPI processes, i.e., each node has 1 MPI process. For FFTE 6.1alpha and FFTE 6.1alpha with AT, each MPI process has 32 threads. For FFTE 6.1alpha and FFTE 6.1alpha with AT, all routines were written in Fortran. A Tofu2-optimized Message Passing Interface based on the Open MPI library was used as the communication library.

The compiler used was the Fujitsu Fortran compiler (`frtpx`, version 1.2.1). The compiler options used were specified as "`frtpx -Kfast,ocl,preex, openmp`". The compiler option "`-Kfast`" creates an object program that runs at high speed on the target machine, and "`ocl`" allows optimization control lines. When the compiler option "`preex`" is specified, optimizations are applied to move invariant expressions evaluated in advance. The compiler option "`openmp`" allows OpenMP directives.

Table 8.2 lists the run times and GFlops of FFTE 6.1alpha and FFTE 6.1alpha with AT. The GFlops value is based on $5N \log_2 N$ for an N-point FFT. Table 8.3 shows the results of automatic tuning of parallel one-dimensional FFTs on the Fujitsu PRIMEHPC FX100 (512 nodes).

On the other hand, for $N \geq 2^{29}$, the FFTE 6.1alpha with AT is faster than the FFTE 6.1alpha (without overlap) due to overlapping computation and communication. As shown in Table 8.3, when the problem size N increases, the number of divisions `NDIV` also increases on the FFTE 6.1alpha with AT. Thus, the default parameters of the FFTE 6.1alpha are not always optimal according to the results of the automatic tuning.

Table 8.1 Specifications of the Fujitsu PRIMEHPC FX100

CPU	Processor	SPARC64 XIfx
	Architecture	SPARC V9 + HPC-ACE2
	Number of cores	32 compute cores + 2 assistant cores
	Theoretical peak performance	Over 1 TFlops (double precision)
Node	Architecture	1 CPU per node
	Memory capacity	32 GB (HMC)
	Memory bandwidth	240 GB/s (read) + 240 GB/s (write)
	Interconnect	Tofu Interconnect 2
	Interconnect link bandwidth	12.5 GB/s×2 (bidirectional) per link

Table 8.2 Performance of parallel one-dimensional FFTs on the Fujitsu PRIMEHPC FX100

Number of nodes	N	FFTE 6.1alpha		FFTE 6.1alpha with AT	
		Time (s)	GFlops	Time (s)	GFlops
1	2^{26}	1.10796	7.874	0.93758	9.305
2	2^{27}	1.07253	16.894	0.91897	19.717
4	2^{28}	0.99461	37.785	0.72559	51.794
8	2^{29}	0.87085	89.392	0.64637	120.436
16	2^{30}	0.83650	192.541	0.66617	241.770
32	2^{31}	0.87016	382.529	0.65455	508.530
64	2^{32}	1.13935	603.148	0.93854	732.193
128	2^{33}	1.15303	1229.227	0.99648	1422.349
256	2^{34}	1.22166	2390.660	0.96346	3031.333
512	2^{35}	1.15052	5226.314	1.08218	5556.349

Table 8.3 Results of automatic tuning of parallel one-dimensional FFTs on the Fujitsu PRIMEHPC FX100 (512 nodes)

N	FFTE 6.1alpha					FFTE 6.1alpha with AT				
	N_1	N_2	NB	NDIV	GFlops	N_1	N_2	NB	NDIV	GFlops
2^{26}	8192	8192	32	4	534.291	8192	8192	16	1	1888.453
2^{27}	8192	16384	32	4	957.086	8192	16384	64	1	2274.347
2^{28}	16384	16384	32	4	1791.444	16384	16384	64	1	2863.900
2^{29}	16384	32768	32	4	2336.193	32768	16384	64	2	3193.509
2^{30}	32768	32768	32	4	3471.513	16384	65536	64	2	3651.177
2^{31}	32768	65536	32	4	3973.254	32768	65536	64	4	4177.870
2^{32}	65536	65536	32	4	4411.628	65536	65536	64	4	4689.423
2^{33}	65536	131072	32	4	4495.392	131072	65536	32	8	4864.679
2^{34}	131072	131072	32	4	5118.963	131072	131072	64	8	5304.136
2^{35}	131072	262144	32	4	5226.314	262144	131072	64	8	5556.349

We note that the performance was more than 5.55 TFlops with size $N = 2^{35}$ in the FFTE 6.1alpha with AT on the Fujitsu PRIMEHPC FX100 (512 nodes), as shown in Table 8.3.

8.3 Parallel Three-Dimensional FFT Using Two-Dimensional Decomposition

8.3.1 Introduction

As a typical array distribution method in parallel three-dimensional FFTs, thus far, only one dimension (for example, z-axis) among three dimensions (the x-, y-, and

z-axes) is divided. In this case, the number of data in the z-axis must be greater than or equal to the number of MPI processes.

In the recent massively parallel cluster, the number of cores and the number of processors tend to increase in order to improve the performance. For example, Sunway TaihuLight, which is a massively parallel system, was ranked third in the TOP500 list of November 2018 [8], and the number of cores exceeds 10 million. In such a system, reducing the number of MPI processes by hybrid MPI and OpenMP parallel programming is effective in reducing the communication time. Even in that case, the maximum number of MPI processes is more than 10,000. Therefore, when one-dimensional decomposition is used in the z-axis, the number of data in the z-axis for such a system must be more than 10,000, and the problem size of the three-dimensional FFT is restricted.

As a method by which to solve this problem, a method of three-dimensional decomposition in the x-, y-, and z-axes has been proposed [21, 22]. In the three-dimensional decomposition, when performing FFTs in the x-, y-, and z-axes, it is necessary to exchange data by all-to-all communication each time. On the other hand, in the two-dimensional decomposition, since there is one undecomposed axis among the x-, y-, and z-axes, there is an advantage in that the number of all-to-all communications can be reduced [46]. Parallel three-dimensional FFT algorithms using two-dimensional decomposition have been proposed [1, 14, 37, 46].

In this section, we describe a parallel three-dimensional FFT algorithm using two-dimensional decomposition in the y- and z-axes. Using the parallel three-dimensional FFT algorithm, it is possible to obtain high scalability, even with a relatively small number of data.

8.3.2 *Implementation of Parallel Three-Dimensional FFT Using Two-Dimensional Decomposition*

Figure 8.3 shows the parallel three-dimensional FFT using one-dimensional block decomposition of initial data in the z-axis, which is a typical array distribution method in parallel three-dimensional FFT. In this section, a parallel three-dimensional FFT algorithm using two-dimensional decomposition in the y- and z-axes is described.

Assume that the number of data N is $N = N_1 \times N_2 \times N_3$, and the MPI process is mapped in two dimensions $P \times Q$. In a distributed-memory parallel computer having $P \times Q$ MPI processes, the three-dimensional array $x(N_1, N_2, N_3)$ is distributed along the second dimension (N_2) and the third dimension (N_3). If N_2 is divisible by P and N_3 is also divisible by Q, $N_1 \times (N_2/P) \times (N_3/Q)$ pieces of data are distributed to each MPI process. Although somewhat complicated, we next introduce the notations $\hat{N}_r = N_r/P$ and $\hat{\hat{N}}_r = N_r/Q$. Moreover, $\hat{J}_r$ denotes that data along the index J_r is distributed to P MPI processes and $\hat{\hat{J}}_r$ denotes that data along the index J_r is distributed to Q MPI processes. Note that r means that the index is of dimension

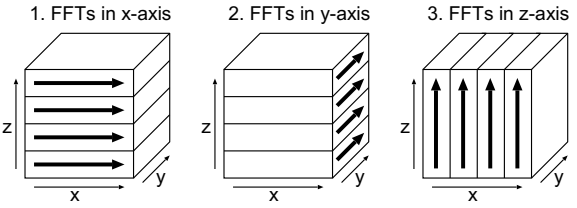

Fig. 8.3 One-dimensional decomposition for a three-dimensional FFT

r. Thus, the distributed three-dimensional array can be expressed as $\hat{x}(N_1, \hat{N}_2, \hat{\hat{N}}_3)$. According to the block distribution, the local index $\hat{J}_r(l)$ in the l-th MPI process in the y-axis corresponds to the following global index, J_r:

$$J_r = l \times \hat{N}_r + \hat{J}_r(l), \quad 0 \le l \le P-1, \quad 1 \le r \le 3. \tag{8.5}$$

Moreover, the local index $\hat{\hat{J}}_r(m)$ in the m-th MPI process in the z-axis corresponds to the global index J_r as follows:

$$J_r = m \times \hat{\hat{N}}_r + \hat{\hat{J}}_r(m), \quad 0 \le m \le Q-1, \quad 1 \le r \le 3. \tag{8.6}$$

Here, to show all-to-all communication, the notations $\tilde{N}_i \equiv N_i/P_i$ and $\tilde{\tilde{N}}_i \equiv N_i/Q_i$ are introduced. Here, N_i is decomposed into two-dimensional representations of $\tilde{N}_i$ and P_i and $\tilde{\tilde{N}}_i$ and Q_i. Note that P_i and Q_i are the same as P and Q, respectively, but this indicates that these indices are of dimension i.

Letting the initial data be $\hat{x}(N_1, \hat{N}_2, \hat{\hat{N}}_3)$, the parallel three-dimensional FFT using two-dimensional decomposition [46] is as follows:

Step 1: $(N_2/P) \cdot (N_3/Q)$ individual N_1-point multicolumn FFTs

$$\hat{x_1}(K_1, \hat{J}_2, \hat{\hat{J}}_3) = \sum_{J_1=0}^{N_1-1} \hat{x}(J_1, \hat{J}_2, \hat{\hat{J}}_3)\omega_{N_1}^{J_1 K_1}.$$

Step 2: Transposition

$$\begin{aligned}\hat{x_2}(\hat{J}_2, \hat{\hat{J}}_3, \tilde{K}_1, P_1) &\equiv \hat{x_2}(\hat{J}_2, \hat{\hat{J}}_3, K_1) \\ &= \hat{x_1}(K_1, \hat{J}_2, \hat{\hat{J}}_3).\end{aligned}$$

Step 3: Q individual all-to-all communications across P MPI processes in the y -axis

$$\hat{x_3}(\tilde{J}_2, \hat{\hat{J}}_3, \hat{K}_1, P_2) = \hat{x_2}(\hat{J}_2, \hat{\hat{J}}_3, \tilde{K}_1, P_1).$$

Step 4: Rearrangement

$$\begin{aligned}\hat{x_4}(J_2, \hat{\hat{J}}_3, \hat{K}_1) &\equiv \hat{x_4}(\tilde{J}_2, P_2, \hat{\hat{J}}_3, \hat{K}_1) \\ &= \hat{x_3}(\tilde{J}_2, \hat{\hat{J}}_3, \hat{K}_1, P_2).\end{aligned}$$

Step 5: $(N_3/Q) \cdot (N_1/P)$ individual N_2-point multicolumn FFTs

$$\hat{x_5}(K_2, \hat{\hat{J}}_3, \hat{K}_1) = \sum_{J_2=0}^{N_2-1} \hat{x_4}(J_2, \hat{\hat{J}}_3, \hat{K}_1)\omega_{N_2}^{J_2 K_2}.$$

Step 6: Transposition

$$\begin{aligned}\hat{x_6}(\hat{\hat{J}}_3, \hat{K}_1, \tilde{\tilde{K}}_2, Q_2) &\equiv \hat{x_6}(\hat{\hat{J}}_3, \hat{K}_1, K_2) \\ &= \hat{x_5}(K_2, \hat{\hat{J}}_3, \hat{K}_1).\end{aligned}$$

Step 7: P individual all-to-all communications across Q MPI processes in the z -axis

$$\hat{x_7}(\tilde{\tilde{J}}_3, \hat{K}_1, \hat{\hat{K}}_2, Q_3) = \hat{x_6}(\hat{\hat{J}}_3, \hat{K}_1, \tilde{\tilde{K}}_2, Q_2).$$

Step 8: Rearrangement

$$\begin{aligned}\hat{x_8}(J_3, \hat{K}_1, \hat{\hat{K}}_2) &\equiv \hat{x_8}(\tilde{\tilde{J}}_3, Q_3, \hat{K}_1, \hat{\hat{K}}_2) \\ &= \hat{x_7}(\tilde{\tilde{J}}_3, \hat{K}_1, \hat{\hat{K}}_2, Q_3).\end{aligned}$$

Step 9: $(N_1/P) \cdot (N_2/Q)$ individual N_3-point multicolumn FFTs

$$\hat{x_9}(K_3, \hat{K}_1, \hat{\hat{K}}_2) = \sum_{J_3=0}^{N_3-1} \hat{x_8}(J_3, \hat{K}_1, \hat{\hat{K}}_2)\omega_{N_3}^{J_3 K_3}.$$

Step 10: Transposition

$$\hat{y}(\hat{K}_1, \hat{\hat{K}}_2, K_3) = \hat{x_9}(K_3, \hat{K}_1, \hat{\hat{K}}_2).$$

Figure 8.4 shows the parallel three-dimensional FFT when the initial data is decomposed into two dimensions in the y- and z-axes. In this parallel three-dimensional FFT, when $N_1 = N_2 = N_3 = N^{1/3}$, $(N^{1/3}/P) \cdot (N^{1/3}/Q)$ individual $N^{1/3}$-point multicolumn FFTs are performed in Steps 1, 5, and 9. Note that while the input data $\hat{x}(J_1, \hat{J}_2, \hat{\hat{J}}_3)$ is two-dimensionally decomposed in the y- and z-axes, the Fourier transformed output data $\hat{y}(\hat{K}_1, \hat{\hat{K}}_2, K_3)$ is two-dimensionally decomposed in the x- and y-axes. In this manner, by using different data distributions for input and output, it is sufficient to perform all-to-all communications twice in the y- and z-axes in Steps 3 and 7. When the same data distribution is used for input and output, it is necessary to perform all-to-all communications in the y- and z-axes once more.

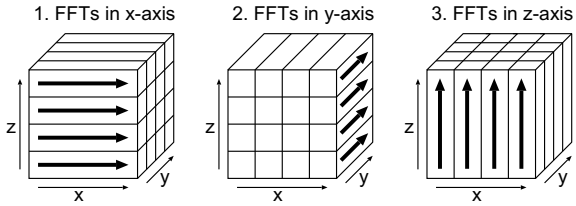

Fig. 8.4 Two-dimensional decomposition for a three-dimensional FFT

8.3.3 Communication Time in One-Dimensional Decomposition and Two-Dimensional Decomposition

Let N be the total number of data, and let $P \times Q$ be the number of MPI processes. Moreover, let W be the communication bandwidth (byte/s), and let L be the communication latency (s). Hereinafter, the communication time in the cases of one-dimensional decomposition and two-dimensional decomposition will be examined. For simplicity, we assume that there is no communication contention in all-to-all communications.

8.3.3.1 Communication Time in One-Dimensional Decomposition

In the case of one-dimensional decomposition, each MPI process will send $N/(PQ)^2$ pieces of double-precision complex data to $PQ - 1$ MPI process other than itself.

Therefore, the communication time $T_{1\text{dim}}$ in the one-dimensional decomposition is expressed as follows:

$$\begin{aligned} T_{1\text{dim}} &= (PQ - 1)\left(L + \frac{16N}{(PQ)^2 \cdot W}\right) \\ &\approx PQ \cdot L + \frac{16N}{PQ \cdot W}. \end{aligned} \tag{8.7}$$

8.3.3.2 Communication Time in Two-Dimensional Decomposition

In the case of two-dimensional decomposition, since Q pairs of all-to-all communications are performed among P MPI processes in the y-axis, each MPI process in the y-axis sends $N/(P^2Q)$ double-precision complex data to $P - 1$ MPI processes in the y-axis. Moreover, since P pairs of all-to-all communications are performed among Q MPI processes in the z-axis, each MPI process in the z-axis sends $N/(PQ^2)$ double-precision complex data to $Q - 1$ MPI processes in the z-axis.

Therefore, the communication time $T_{2\text{dim}}$ in the two-dimensional decomposition is expressed as follows:

$$\begin{aligned} T_{2\text{dim}} &= (P - 1)\left(L + \frac{16N}{P^2Q \cdot W}\right) + (Q - 1)\left(L + \frac{16N}{PQ^2 \cdot W}\right) \\ &\approx (P + Q) \cdot L + \frac{32N}{PQ \cdot W}. \end{aligned} \tag{8.8}$$

8.3.3.3 Comparison of Communication Time in One-Dimensional Decomposition and Two-Dimensional Decomposition

The communication times in the cases of one-dimensional decomposition and two-dimensional decomposition are expressed by Eqs. (8.7) and (8.8), respectively.

In comparing these two equations, the one-dimensional decomposition of Eq. (8.7) becomes approximately half of two-dimensional decomposition of Eq. (8.8) as the total communication amount. However, when the total number of MPI processes $P \times Q$ is large and the latency L is large, the communication time is shorter in the two-dimensional decomposition of Eq. (8.8). Here, a condition is obtained whereby the communication time $T_{2\text{dim}}$ represented by Eq. (8.8) is smaller than the communication time $T_{1\text{dim}}$ represented by Eq. (8.7).

From Eqs. (8.7) and (8.8),

$$(P+Q)\cdot L + \frac{32N}{PQ\cdot W} < PQ\cdot L + \frac{16N}{PQ\cdot W}. \tag{8.9}$$

We have

$$N < \frac{(LW\cdot PQ)(PQ-P-Q)}{16}. \tag{8.10}$$

For example, substituting $L = 10^{-5}$ (s), $W = 10^9$ (byte/s), and $P = Q = 64$ into Eq. (8.10), in the range of $N < 10^{10}$, the communication time of two-dimensional decomposition is found to be smaller than that of one-dimensional decomposition.

8.3.4 Performance Results

In the performance evaluation, we compare the performances of parallel three-dimensional FFT using two-dimensional decomposition and parallel three-dimensional FFT using one-dimensional decomposition. For the measurement, 32^3, 64^3, 128^3, and 256^3-point forward FFTs were performed 10 times and the average elapsed time was measured. The calculation of the FFT is performed with double-precision complex numbers and the trigonometric function table is prepared in advance. In multicolumn FFTs, the Stockham FFT algorithm was used for in-cache FFT when each column FFT fits into the cache. The in-cache FFT is implemented by combinations of radices 2, 3, 4, 5, and 8, and the same program was used for one-dimensional decomposition and two-dimensional decomposition.

Moreover, in order to reduce the number of all-to-all communications, different data distribution formats are used for the input and output of the three-dimensional FFT. In the one-dimensional decomposition, the input is decomposed in the z-axis and the output is decomposed in the x-axis. In the two-dimensional decomposition,

the input is decomposed in the y- and z-axes, and the output is decomposed in the x- and y-axes.

An Appro Xtreme-X3 (648 nodes, 32 GB per node, 147.2 GFlops per node, total main memory: 20 TB, communication bandwidth: 8 GB/s per node, and peak performance: 95.4 TFlops) was used as a multicore massively parallel cluster. Each computation node of the Xtreme-X3 has a four-socket quad-core AMD Opteron 8356 (Barcelona, 2.3 GHz) in a 16-core shared-memory configuration with a performance of 147.2 GFlops. All of the nodes in the system are connected through a full-bisectional fat tree network with DDR InfiniBand.

MVAPICH 1.2.0 [6] was used as a communication library. In this performance evaluation, one MPI process per core was used and the flat MPI programming model was used because this evaluation focuses on scalability when the number of MPI processes is large. The Intel Fortran Compiler 10.1 was used with the compiler option "`ifort -O3 -xO`".

Figure 8.5 shows the performance of parallel three-dimensional FFTs using two-dimensional decomposition. Here, the GFlops value of $N = 2^m$-point FFT is calculated from $5N \log_2 N$. As shown in Fig. 8.5, good scalability is not obtained using 32^3-point FFT. This is because the problem size is small (1 MB), which is why all-to-all communication dominates most of the total execution time. On the other hand, the performance is improved up to 4,096 cores for 256^3-point FFT. The performance for 4,096 cores was approximately 401.3 GFlops.

The performances of one-dimensional decomposition and two-dimensional decomposition in 256^3-point parallel three-dimensional FFT are shown in Fig. 8.6. As shown in Fig. 8.6, in the case of 64 or fewer cores, the one-dimensional decomposition with a small communication amount has higher performance than the two-dimensional decomposition. On the other hand, for 128 or more cores, the two-dimensional decomposition, which can reduce the communication time, has higher performance than the one-dimensional decomposition.

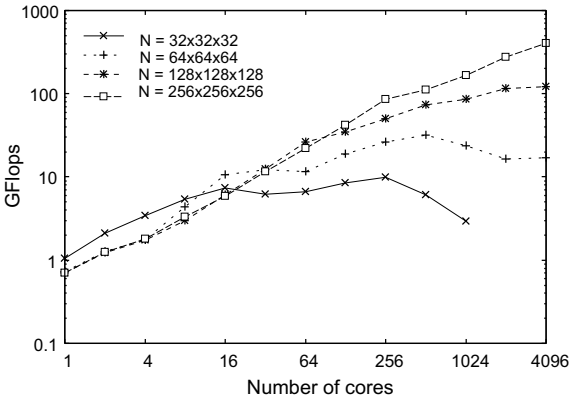

Fig. 8.5 Performance of parallel three-dimensional FFTs using two-dimensional decomposition

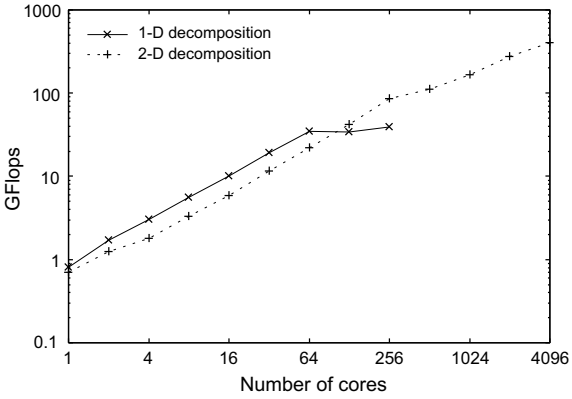

Fig. 8.6 Performance of parallel three-dimensional 256^3-point FFTs

Table 8.4 Execution and communication times of parallel 256^3-point FFTs using one-dimensional decomposition

Number of cores	Execution time (s)	Communication time (s)	% of Communication
1	2.49814	0.17032	6.82
2	1.17380	0.15196	12.95
4	0.66563	0.14482	21.76
8	0.35875	0.10306	28.73
16	0.19917	0.07495	37.63
32	0.10570	0.04313	40.81
64	0.05839	0.02584	44.26
128	0.05900	0.03552	60.19
256	0.05136	0.03622	70.53

In the case of one-dimensional decomposition, 256^3-point FFT can be performed in 256 MPI processes or fewer, but 256^3-point FFT can be performed in up to 65,536 MPI processes when two-dimensional decomposition is used.

Tables 8.4 and 8.5 show the ratio of the communication time to the execution time for 256^3-point parallel three-dimensional FFT using one-dimensional decomposition and two-dimensional decomposition, respectively. As shown in Table 8.5, for 4,096 cores, more than 96% of the total time is taken up by communication. The reason for this is considered to be that latency is dominant in the communication time because the amount of communications data sent by each MPI process at one time in all-to-all communication is only 1 KB.

In order to further reduce the communication time, it is necessary to reduce the latency by using a lower level communication function without using the `MPI_Alltoall` function for all-to-all communication, as in reference [21].

Table 8.5 Execution and communication times of parallel 256^3-point FFTs using two-dimensional decomposition

Number of cores	Execution time (s)	Communication time (s)	% of Communication
1	2.84448	0.33756	11.87
2	1.60708	0.25705	15.99
4	1.11583	0.26579	23.82
8	0.61128	0.16514	27.02
16	0.34474	0.11832	34.32
32	0.17320	0.06372	36.79
64	0.09039	0.03737	41.34
128	0.04823	0.02176	45.11
256	0.02346	0.01320	56.28
512	0.01793	0.01461	81.48
1024	0.01199	0.01079	89.98
2048	0.00736	0.00652	88.64
4096	0.00502	0.00482	96.07

8.4 Optimization of All-to-All Communication on Multicore Cluster Systems

8.4.1 Introduction

All-to-all communication is widely used in parallel algorithms. For example, matrix transposition and parallel FFT require intensive all-to-all communication. However, all-to-all communication is one of the performance bottlenecks for many parallel applications. Thus, it is important to optimize the all-to-all communication to achieve good scalability for massively parallel computers.

Effective implementations of all-to-all communication have been well studied [13, 19, 23–25, 30, 31, 41, 53]. An empirical approach for generating efficient all-to-all communication routines for Ethernet switched clusters is presented in [24].

A performance model and measurement of all-to-all on Blue Gene/L are presented in [31]. Several optimization algorithms for MPI collective communication on Blue Gene/P are presented in [23]. An overlapping method for all-to-all communications to reduce the network contention in asymmetric torus network is also presented in [19].

In this section, we describe an optimization method for all-to-all communication on multicore cluster systems.

8.4.2 Two-Step All-to-All Communication Algorithm

Kumar et al. proposed an optimized all-to-all collective algorithm for multicore systems connected using modern InfiniBand network interfaces [30]. All-to-all communication comprises two steps: intra-node exchange and inter-node exchange.

We extend this algorithm to the general case of $P = P_x \times P_y$ MPI processes, where P_x and P_y denote the number of MPI processes along the x-axis and y-axis, respectively.

In the all-to-all communication algorithm for multicore systems [30], P_x and P_y denote the number of cores on each node and the number of nodes, respectively. However, the parameters P_x and P_y in the all-to-all communication algorithm for multicore systems may not always be optimal.

A generalized two-step all-to-all communication algorithm [45] can be derived as follows. Let N be the total number of elements of the array of all MPI processes.

1. In each MPI process, copy the array subscript order so that it changes from $(N/P^2, P_x, P_y)$ to $(N/P^2, P_y, P_x)$. Then, P_x pairs of all-to-all communications among P_y MPI processes are performed.
2. In each MPI process, copy the array subscript order so that it changes from $(N/P^2, P_y, P_x)$ to $(N/P^2, P_x, P_y)$. Then, P_y pairs of all-to-all communications among P_x MPI processes are performed.

Figure 8.7 shows a simplified code for the two-step all-to-all communication algorithm. In this code, the MPI_Comm_Split routine splits the MPI_COMM_WORLD communicator into sub-communicators. The sub-communicator for step 1 contains the P_x MPI processes, and the sub-communicator for step 2 contains the P_y MPI processes.

In the two-step all-to-all communication algorithm, the inter-node all-to-all communication takes place twice. Thus, the two-step all-to-all communication algorithm requires twice the total amount of communications as the ring algorithm [24].

However, for small to medium messages, the two-step all-to-all communication algorithm is better than the ring algorithm owing to the smaller startup time.

We can extend the two-step all-to-all communication algorithm in another way to a three-step all-to-all communication algorithm. In the three-step all-to-all commu-

Fig. 8.7 Two-step all-to-all communication

```
Two-Step-Alltoall(sendbuf, ..., recvbuf, ..., Px, Py, ...)
{
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_split(MPI_COMM_WORLD, rank/Px, 0, &comm_x);
  MPI_Comm_split(MPI_COMM_WORLD, rank%Px, 0, &comm_y);
  Transpose from sendbuf(n/P, Px, Py) to tmpbuf(n/P, Py, Px).
  MPI_Alltoall(tmpbuf, ..., recvbuf, ..., comm_x);
  Transpose from recvbuf(n/P, Py, Px) to tmpbuf(n/P, Px, Py).
  MPI_Alltoall(tmpbuf, ..., recvbuf, ..., comm_y);
  MPI_Comm_free(comm_x);
  MPI_Comm_free(comm_y);
}
```

nication algorithm, the inter-node all-to-all communication takes place three times. Thus, the three-step all-to-all communication algorithm requires three times the total amount of communications as that of the ring algorithm. This is a trade-off between the startup time and the total amount of communications.

8.4.3 Communication Times of All-to-All Communication Algorithms

We will compare the communication times of the ring algorithm and the two-step all-to-all communication algorithm. For simplicity, we assume that no communication contention in all-to-all communications.

Let us assume the following parameters for all-to-all communication.

- Data size per MPI process: N (byte),
- Startup time for each message: L (s),
- Communication bandwidth: W (byte/s), and
- Number of MPI processes: $P = P_x \times P_y$.

In the ring algorithm, each MPI process sends N/P bytes of data to $P - 1$ MPI processes.

Thus, the communication time of the ring algorithm T_{ring} can be estimated as follows:

$$\begin{aligned} T_{\text{ring}} &= (P-1)\left(L + \frac{N}{P \cdot W}\right) \\ &\approx P \cdot L + \frac{N}{W}. \end{aligned} \tag{8.11}$$

From Eq. (8.11), for small messages, the communication time of the ring algorithm T_{ring} is dominated by startup time.

In the two-step all-to-all communication algorithm, each MPI process in the x-axis sends N/P_x bytes of data to $P_x - 1$ MPI processes in the x-axis. Also, each MPI process in the y-axis sends N/P_y bytes of data to $P_y - 1$ MPI processes in the y-axis.

Thus, the communication time of the two-step all-to-all communication algorithm T_{2step} can be estimated as follows:

$$\begin{aligned} T_{\text{2step}} &= (P_x-1)\left(L + \frac{N}{P_x \cdot W}\right) + (P_y-1)\left(L + \frac{N}{P_y \cdot W}\right) \\ &\approx (P_x + P_y) \cdot L + \frac{2N}{W}. \end{aligned} \tag{8.12}$$

From Eq. (8.12), the two-step all-to-all communication algorithm requires twice the total amount of communications compared with the ring algorithm. However, by

```
min_time = DBL_MAX;
for (i = 0; i <= log2(P); i++) {
  Px = 2^i;
  Py = P/Px;
  MPI_Barrier(MPI_COMM_WORLD);
  start = MPI_Wtime();
  for (count = 0; count < ITER_NUM; count++) {
    if (Px == 1 || Py == 1)
      MPI_Alltoall(sendbuf, ..., recvbuf, ...);
    else
      Two-Step-Alltoall(sendbuf, ..., recvbuf, ...,Px, Py, ...);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  end = MPI_Wtime();
  if (end − start < min_time) {
    min_time = end - start;
    Qx = Px;
    Qy = Py;
  }
}
Px = Qx;
Py = Qy;
```

Fig. 8.8 Automatic tuning of all-to-all communication [45]

comparing Eqs. (8.11) and (8.12), the communication time of the two-step all-to-all communication algorithm is less than that of the ring algorithm for a larger number of MPI processes $P = P_x \times P_y$ and startup time for each message L.

Automatic tuning of all-to-all communication can be accomplished by performing a search over the parameters of all of P_x and P_y.

If $P = P_x \times P_y$ is a power of two, the size of the search space is $\log_2 P$. When P_x or P_y is equal to one, the MPI_Alltoall routine is used. Figure 8.8 shows a simplified code for the automatic tuning of all-to-all communication.

8.4.4 Performance Results

We evaluate and compare the performance of the proposed scheme for all-to-all communication on the multicore cluster system. We averaged the elapsed times obtained from 10 iterations.

An Appro Xtreme-X3 (648 nodes, 32 GB per node, 147.2 GFlops per node, total main memory: 20 TB, communication bandwidth: 8 GB/s per node, and peak performance: 95.4 TFlops) was used as a multicore cluster system. Each computation node of the Xtreme-X3 has a four-socket quad-core AMD Opteron 8356 (Barcelona, 2.3 GHz) in a 16-core shared-memory configuration with a performance of 147.2 GFlops. All of the nodes in the system are connected through a full-bisectional fat tree network with DDR InfiniBand.

Table 8.6 Performance of all-to-all communication on the Appro Xtreme-X3 (flat MPI, 64 nodes, 1024 cores)

Message size (bytes)	MPI_Alltoall			Automatically tuned all-to-all				
	P	Time (s)	Bandwidth (MB/s)	P_x	P_y	Time (s)	Bandwidth (MB/s)	Tuning time (s)
16	1024	0.00126	13.045	1024	1	0.00144	11.402	1.883
32	1024	0.00308	10.649	1024	1	0.00347	9.443	0.305
64	1024	0.00518	12.644	1	1024	0.00548	11.955	0.425
128	1024	0.01052	12.463	1024	1	0.01085	12.076	0.754
256	1024	0.01104	23.740	1024	1	0.01137	23.060	0.794
512	1024	0.02458	21.330	1024	1	0.02488	21.069	1.590
1024	1024	0.04917	21.327	128	8	0.04650	22.552	2.957
2048	1024	0.09663	21.704	64	16	0.05824	36.006	5.339
4096	1024	0.20447	20.513	32	32	0.07651	54.821	9.643
8192	1024	0.40991	20.464	16	64	0.14334	58.520	15.362
16384	1024	1.48561	11.293	8	128	0.27483	61.046	22.345
32768	1024	1.58972	21.107	4	256	0.50260	66.762	35.701
65536	1024	1.99518	33.635	2	512	0.95924	69.960	62.639
131072	1024	1.91880	69.949	1024	1	0.89231	150.416	108.232
262144	1024	2.56603	104.611	1024	1	2.31248	116.081	432.128

MVAPICH2 1.5.1 [6] was used as a communication library. The Intel Fortran Compiler (ifort, version 11.1) was used on the Appro Xtreme-X3. The compiler options used were specified as "`ifort -O3 -xO`" for the flat MPI programming model and "`ifort -O3 -xO -openmp`" for hybrid MPI/OpenMP programming model, respectively.

8.4.4.1 Performance Results on Flat MPI Programming Model

In the experiment, we used 1024 cores (64 nodes) with flat MPI programming model. Each message size of the all-to-all communication was varied from 16 bytes to 262144 bytes.

Table 8.6 shows the performance of all-to-all communication on the Appro Xtreme-X3 (flat MPI, 64 nodes, 1024 MPI processes). For the message ranges 1–64 KB, the two-step all-to-all communication algorithm is selected. When $P_x \leq 16$, the all-to-all communication across P_x cores are performed using shared memory. The time required for a tuning depends on the message size and the number of cores.

Figure 8.9 shows the performance of all-to-all communication on the Appro Xtreme-X3 (flat MPI, 64 nodes, 1024 MPI processes). Since the bandwidth of the Appro Xtreme-X3 is 8 GB/s per node, the peak bandwidth per core is 500 MB/s. However, the available bandwidth is reduced due to network contention and the startup overhead.

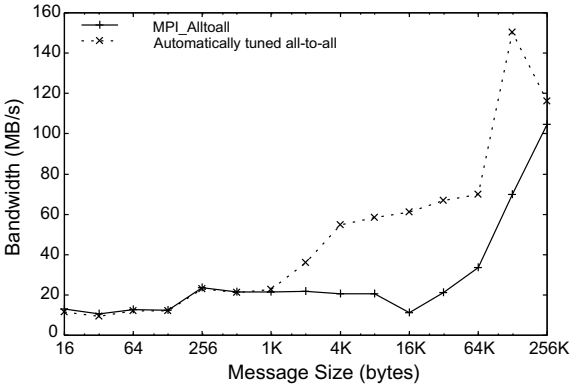

Fig. 8.9 Performance of all-to-all communication on the Appro Xtreme-X3 (flat MPI, 64 nodes, 1024 cores)

For medium messages, the automatically tuned all-to-all communication offers much higher performance than MPI_Alltoall routine. When the message size is 16 KB, the communication time for the automatically tuned all-to-all communication is 0.27483 (s), and the time for MPI_Alltoall routine is 1.48561 (s), which is approximately 5.4x speedup. This is because the two-step all-to-all communication algorithm is better than the ring algorithm due to the smaller startup time.

As shown in Table 8.6, the cases of $P_x = 16$ which corresponds to the all-to-all communication algorithm for multicore systems [30] are not always optimal according to the results of the automatic tuning of all-to-all communication.

8.4.4.2 Performance Results on Hybrid MPI/OpenMP Programming Model

In the experiment, we used 256 sockets (64 nodes) with the hybrid MPI/OpenMP programming model, i.e., each node has 4 MPI processes. Each message size of the all-to-all communication was varied from 16 bytes to 4194304 bytes.

Table 8.7 shows the performance of all-to-all communication on the Appro Xtreme-X3 (64 nodes, 256 sockets). For the message ranges 2–8 KB, the two-step all-to-all communication algorithm is selected. When $P_x \leq 4$, the all-to-all communication across P_x sockets are performed using shared memory.

Figure 8.10 shows the performance of all-to-all communication on the Appro Xtreme-X3 (64 nodes, 256 sockets). Since the bandwidth of the Appro Xtreme-X3 is 8 GB/s per node, the peak bandwidth per socket is 2 GB/s. When the message size is 8 KB, the communication time for the automatically tuned all-to-all communication is 0.01748 (s), and the time for MPI_Alltoall routine is 0.04596 (s), which is approximately 2.6x speedup.

Table 8.7 Performance of all-to-all communication on the Appro Xtreme-X3 (hybrid MPI/OpenMP, 64 nodes, 256 sockets)

Message size	MPI_Alltoall			Automatically tuned all-to-all				
(bytes)	P	Time (s)	Bandwidth (MB/s)	P_x	P_y	Time (s)	Bandwidth (MB/s)	Tuning time (s)
16	256	0.00024	16.860	256	1	0.00023	18.199	0.184
32	256	0.00032	25.757	256	1	0.00026	31.756	0.034
64	256	0.00040	40.542	1	256	0.00046	35.386	0.045
128	256	0.00091	35.847	256	1	0.00082	39.814	0.101
256	256	0.00173	37.951	256	1	0.00156	41.877	0.108
512	256	0.00338	38.814	1	256	0.00308	42.515	0.214
1024	256	0.00338	77.490	256	1	0.00334	78.413	0.223
2048	256	0.00803	65.315	16	16	0.00736	71.274	0.444
4096	256	0.02147	48.834	32	8	0.01258	83.326	0.903
8192	256	0.04596	45.625	16	16	0.01748	119.954	1.592
16384	256	0.02850	147.163	256	1	0.03006	139.531	2.179
32768	256	0.04918	170.566	256	1	0.05317	157.755	3.375
65536	256	0.09630	174.224	256	1	0.09707	172.843	5.960
131072	256	0.06274	534.800	256	1	0.06558	511.633	9.960
262144	256	0.12216	549.340	1	256	0.13064	513.700	20.549
524288	256	0.23043	582.454	256	1	0.25844	519.336	39.786
1048576	256	0.45239	593.371	256	1	0.46494	577.360	77.790
2097152	256	0.89151	602.205	256	1	0.91232	588.468	251.538
4194304	256	1.96136	547.448	256	1	2.12298	505.771	1130.119

8.5 Parallel One-Dimensional FFT in a GPU Cluster

8.5.1 Introduction

In recent years, the high performance and memory bandwidth of the graphics processing unit (GPU) have attracted attention, and attempts have been made to apply the GPU to various HPC applications. Moreover, GPU clusters, which connect a large number of computation nodes equipped with GPUs, are also in widespread use, and Titan, which is a GPU cluster equipped with NVIDIA Tesla V100 GPUs, was ranked first in the TOP500 list of November 2018 [8].

In a GPU cluster, GPUs are connected to individual nodes as PCI Express devices. Many FFT implementations work well within GPU cards [11, 34]. However, PCI Express transfer is often a performance bottleneck in FFT because FFT requires a large number of memory accesses per arithmetic operation. One goal for parallel FFTs on GPU clusters is to minimize the PCI Express transfer time and the MPI communication time.

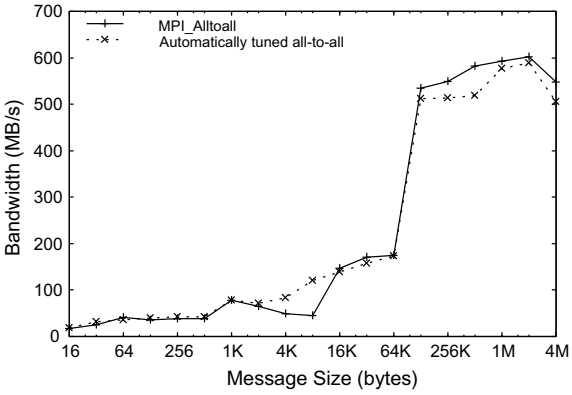

Fig. 8.10 Performance of all-to-all communication on the Appro Xtreme-X3 (hybrid MPI/OpenMP, 64 nodes, 256 sockets)

Parallel three-dimensional FFT implementations on GPU clusters have previously been presented [18, 35]. Also, parallel one-dimensional FFT algorithm on Xeon Phi cluster has been proposed [36].

In this section, we describe an implementation of a parallel one-dimensional FFT in a GPU cluster. The implementation is based on the parallel one-dimensional FFT described in Sect. 8.1.1.

8.5.2 *Implementation of Parallel One-Dimensional FFT in a GPU Cluster*

Implementation of parallel one-dimensional FFT in a GPU cluster has been proposed [47], but all-to-all communication takes place three times when performing parallel one-dimensional FFT, so most of the computation time is dominated by all-to-all communication. Furthermore, since the theoretical bandwidth of the PCI Express bus, which is the interface connecting the CPU and the GPU, is 8 GB/s per direction in the PCI Express Gen2 × 16 lanes. Thus, it is also important to reduce the amount of data transfer between the CPU and the GPU.

The parallel one-dimensional FFT was implemented using a CUDA Fortran [10] program with MPI. CUDA Fortran allows automatic kernel generation and invocation from a region of host code containing one or more tightly nested loops [10]. Therefore, this CUDA Fortran extension was used.

Listing 8.2 CUDA Fortran code for complex vector addition [47]

```
      complex(8),device :: A_d(N),B_d(N),C_d(N)
!$cuf kernel <<<*,*>>>
      do i=1,N
        C_d(i)=A_d(i)+B_d(i)
      enddo
```

Listing 8.2 shows a CUDA Fortran code for complex vector addition. The kernel loop directive "`!$ cuf kernel do «< *, * »>`" in Listing 8.2 instructs the compiler to automatically create and call the CUDA kernel.

In the parallel one-dimensional FFT described in Sect. 8.1.1, coalesced memory access can be performed in the rearrange steps (Steps 1, 5, and 9). On the other hand, the transpose steps (Steps 3, 7 and 11) contain a stride memory access. To avoid the stride memory access, we can use the tiled transpose for GPUs [40]. When the data sets fit into GPU device memory, the rearrange steps and the transpose steps can be performed within the device memory.

On a typical implementation for performing one-dimensional FFT, the table of twiddle factors $\omega_{n_1 n_2}^{j_1 k_2}$ in Eq. (6.5) is precomputed in advance. In general, the array of twiddle factors has the same size as the input array. Since the device memory of NVIDIA Tesla M2090 is 6 GB, the array of twiddle factors should be allocated in the host memory. In this case, PCI Express transfer is the chief bottleneck because the bandwidth of PCI Express 2.0 × 16 is only 8 GB/s, whereas the memory bandwidth of NVIDIA Tesla M2090 is 177.6 GB/s. Thus, the twiddle factors are computed on the fly within each GPU by using trigonometric functions.

Listing 8.3 Naive MPI and CUDA Fortran code for all-to-all communication between GPU and GPU [47]

```
      complex(8) :: A(N),B(N)
      complex(8),device :: A_d(N),B_d(N)
! Copy data from device memory to host memory
      A=A_d
! All-to-all communication between CPU and CPU
      call MPI_ALLTOALL(A,N,MPI_DOUBLE_COMPLEX,B,N,
     &                  MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
! Copy data from host memory to device memory
      B_d=B
```

Naive MPI and CUDA Fortran code for all-to-all communication between GPU and GPU is shown in Listing 8.3. When transferring the memory on the GPU by MPI, basically, the following procedure is basically necessary:

(1) Copy data from the device memory on the GPU to the host memory on the CPU.
(2) Transfer data using the MPI communication function.
(3) Copy data from the host memory on the CPU to the device memory on the GPU.

In this case, there is a problem in that MPI communication is not performed while data is transferred between the CPU and the GPU. Therefore, this problem was solved using the MVAPICH2-GPU [54], which is an MPI library that can pipeline data transfer between a CPU and a GPU and MPI communication between nodes and overlap.

Listing 8.4 MVAPICH2-GPU and CUDA Fortran code for all-to-all communication between GPU and GPU [47]

```
      complex(8) :: A(N),B(N)
      complex(8),device :: A_d(N),B_d(N)
! All-to-all communication between GPU and GPU
      call MPI_ALLTOALL(A_d,N,MPI_DOUBLE_COMPLEX,B_d,N,
     &                  MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
```

Listing 8.4 shows MVAPICH2-GPU and CUDA Fortran code for all-to-all communication between GPU and GPU. We use advanced features of MVAPICH2-GPU for this communication.

Listing 8.5 CUDA Fortran program of parallel one-dimensional FFT [47]

```
      complex(8) :: A(N/P),B(N/P)
      complex(8),device :: A_d(N/P),B_d(N/P)
      integer :: plan1,plan2
! Decompose N into N1 and N2
      call GETN1N2(N,N1,N2)
! Create 1-D FFT plans
      istat=cufftPlan1D(plan1,N1,CUFFT_Z2Z,N2/P)
      istat=cufftPlan1D(plan2,N2,CUFFT_Z2Z,N1/P)
! Copy data from host memory to device memory
      A_d=A
! Step 1: Rearrange (N1/P)*P*(N2/P) to (N1/P)*(N2/P)*P
      call REARRANGE(A_d,B_d,N1/P,P,N2/P)
! Step 2: All-to-all communication between GPU and GPU
      call MPI_ALLTOALL(B_d,N/P,MPI_DOUBLE_COMPLEX,A_d,N/P,
     &                  MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
! Step 3: Transpose (N1/P)*N2 to N2*(N1/P)
      call TRANSPOSE(A_d,B_d,N1/P,N2)
! Step 4: (N1/P) individual N2-point multicolumn FFTs
      istat=cufftExecZ2Z(plan2,B_d,B_d,CUFFT_FORWARD)
! Step 5: Rearrange (N2/P)*P*(N1/P) to (N2/P)*(N1/P)*P and
!         twiddle factor multiplication
      call REARRANGE_TWIDDLE(B_d,A_d,N2/P,P,N1/P)
! Step 6: All-to-all communication between GPU and GPU
      call MPI_ALLTOALL(A_d,N/P,MPI_DOUBLE_COMPLEX,B_d,N/P,
     &                  MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
! Step 7: Transpose (N2/P)*N1 to N1*(N2/P)
      call TRANSPOSE(B_d,A_d,N2/P,N1)
! Step 8: (N2/P) individual N1-point multicolumn FFTs
      istat=cufftExecZ2Z(plan1,A_d,A_d,CUFFT_FORWARD)
! Step 9: Rearrange (N1/P)*P*(N2/P) to (N1/P)*(N2/P)*P
      call REARRANGE(A_d,B_d,N1/P,P,N2/P)
! Step 10: All-to-all communication between GPU and GPU
      call MPI_ALLTOALL(B_d,N/P,MPI_DOUBLE_COMPLEX,A_d,N/P,
     &                  MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
! Step 11: Transpose (N1/P)*N2 to N2*(N1/P)
      call TRANSPOSE(A_d,B_d,N1/P,N2)
! Copy data from device memory to host memory
      B=B_d
! Destroy the 1-D FFT plans
      istat=cufftDestroy(plan1)
      istat=cufftDestroy(plan2)
```

Listing 8.5 shows a CUDA Fortran program of parallel one-dimensional FFT using MVAPICH2-GPU. Here, the arrays `A` and `B` are the input array and the output array, respectively. These arrays are allocated in host memory. The arrays `A_d` and `B_d` are work arrays allocated in device memory. We note that the CUDA Fortran program uses GPUs only for performing parallel one-dimensional FFT.

The NVIDIA CUDA FFT library (CUFFT) [11] is called by the CUDA Fortran program to perform Steps 4 and 8.

8.5.3 Performance Results

In the performance evaluation, we compare the performances of the implemented parallel one-dimensional FFT, named FFTE 6.0 [3], and the FFT library of FFTW 3.3.3 [4]. We changed m of $N = 2^m$ and then executed forward FFTs 10 times and measured the average elapsed time. The calculation of the FFT was performed with double-precision complex numbers, and NVIDIA CUFFT [11] was used as an FFT routine for a single GPU. In FFTW 3.3.3, the "measure" planner was used.

An HA-PACS base cluster was used as a GPU cluster. The specifications for the HA-PACS base cluster are shown in Table 8.8.

In the experiment, we used from 1 to 128 nodes. The parallel one-dimensional FFT programs were run on from 1 to 512 MPI processes, i.e., each node has 4 MPI processes. For FFTE 6.0 (GPU), each MPI process has one CPU thread and one GPU. On the other hand, for FFTE 6.0 (CPU) and FFTW 3.3.3 (CPU), each MPI process has four CPU threads.

For FFTE 6.0 (GPU), all routines were written in CUDA Fortran. The compiler options used were specified as "`pgf90 -fast -Mcuda=cc20,cuda5.5`". The compiler option "`-fast`" chooses generally optimal flags for the target platform, and "`-Mcuda=cc20,cuda5.5`" generates code for a device with compute capability 2.0 and uses the CUDA 5.5 toolkit to build the GPU code. For FFTE 6.0 (CPU), the compiler options used were specified as "`pgf90 -fast -mp`". The compiler option "`-mp`" interprets OpenMP pragmas to explicitly parallelize regions of code for execution by multiple threads on a multiprocessor system. For FFTW 3.3.3 (CPU), the compiler options used were specified as "`pgcc -fast -mp`".

Table 8.9 compares FFTE 6.0 (GPU), FFTE 6.0 (CPU), and FFTW 3.3.3 (CPU) in terms of their run times and GFlops. In the table, the first column states the number of MPI processes, and the second column gives the problem size. The last six columns

Table 8.8 Specification of the HA-PACS base cluster

Number of nodes	268
CPU	Intel Xeon E5-2670 (8-core, 2.6 GHz)× 2
CPU memory	128 GB (DDR3 1600 MHz)
GPU	NVIDIA Tesla M2090 × 4
GPU memory	6 GB (GDDR5 ECC on) per GPU
Interconnect	InfiniBand QDR × 2 rails
Network topology	Fat tree
OS	Linux 2.6.32-358.el6.x86_64
C compiler	PGI C compiler 13.10
Fortran compiler	PGI Fortran compiler 13.10
MPI library	MVAPICH2 2.0b
CUDA toolkit	5.5

Table 8.9 Performance of parallel one-dimensional FFTs on the HA-PACS base cluster

Number of MPI processes	N	FFTE 6.0 (GPU)		FFTE 6.0 (CPU)		FFTW 3.3.3 (CPU)	
		Time (s)	GFlops	Time (s)	GFlops	Time (s)	GFlops
1	2^{25}	0.33955	12.353	1.02498	4.092	0.44576	9.409
2	2^{26}	0.52161	16.726	1.22595	7.116	1.69025	5.161
4	2^{27}	0.70781	25.599	1.80267	10.051	2.06936	8.756
8	2^{28}	1.06053	35.436	2.13668	17.588	2.80648	13.391
16	2^{29}	1.41816	54.892	2.47753	31.421	3.06965	25.360
32	2^{30}	1.60425	100.396	2.67592	60.189	3.28646	49.008
64	2^{31}	1.82508	182.381	3.24729	102.504	3.55636	93.596
128	2^{32}	1.92941	356.168	3.58899	191.473	3.80902	180.413
256	2^{33}	2.01236	704.318	3.81578	371.441	3.90983	362.506
512	2^{34}	2.21854	1316.440	4.18234	698.312	4.35183	671.116

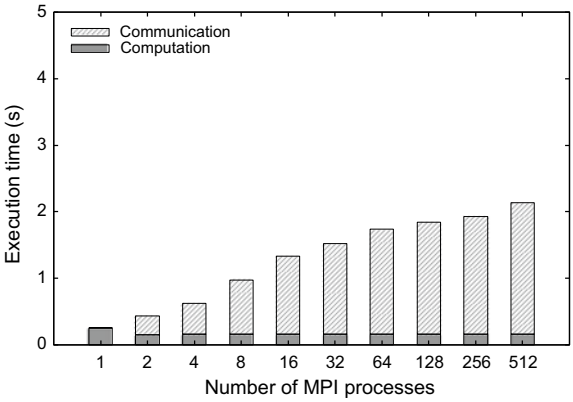

Fig. 8.11 Breakdown of execution time in FFTE 6.0 (GPU) on the HA-PACS base cluster, $N = 2^{25} \times$ number of MPI processes

provide the average elapsed time in seconds and the average execution performance in GFlops for each implementation. The GFlops value is based on $5N \log_2 N$ for an N-point FFT.

FFTE 6.0 (GPU) achieves a speedup of up to 3.02x and 3.24x compared to FFTE 6.0 (CPU) and FFTW 3.3.3 (CPU), respectively, as shown in Table 8.9. Figures 8.11 and 8.12 show a breakdown of the execution time in FFTE 6.0 (GPU) and FFTE 6.0 (CPU) on the HA-PACS base cluster, respectively. In Fig. 8.11, the communication time is the MPI communication time of the all-to-all communication between GPU and GPU.

In the case of using GPUs, the computation time is reduced as compared with the case of using CPUs only, whereas the communication time becomes longer. This

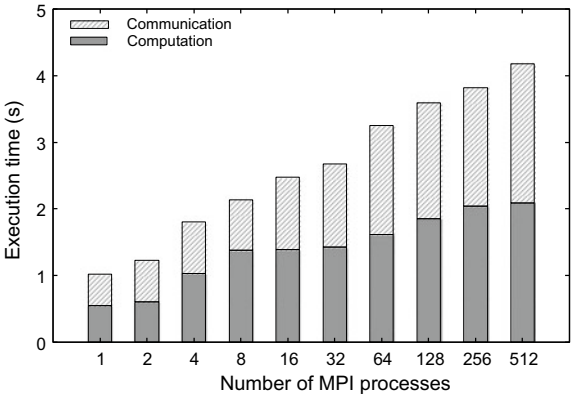

Fig. 8.12 Breakdown of execution time in FFTE 6.0 (CPU) on the HA-PACS base cluster, $N = 2^{25} \times$ number of MPI processes

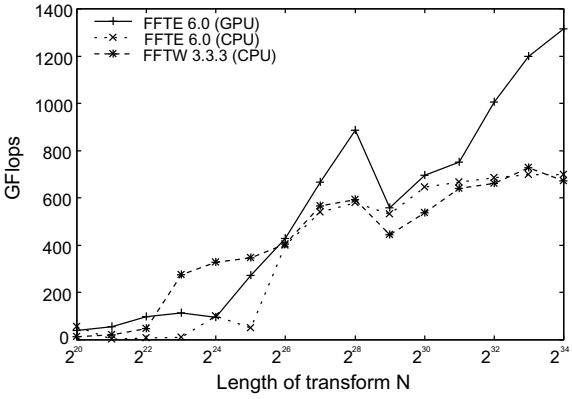

Fig. 8.13 Performance of parallel one-dimensional FFTs on the HA-PACS base cluster (128 nodes, 512 MPI processes)

difference in communication time is thought to be due to the overhead of copying between host memory and device memory.

In Fig. 8.11, we can clearly see that the all-to-all communication overhead contributes significantly to the execution time. For this reason, the difference in performance among FFTE 6.0 (GPU), FFTE 6.0 (CPU) and FFTW 3.3.3 (CPU) decrease as the number of MPI process increases.

Figure 8.13 shows the parallel one-dimensional FFT performance comparison among FFTE 6.0 (GPU), FFTE 6.0 (CPU) and FFTW 3.3.3 (CPU) on the HA-PACS base cluster. Figure 8.14 shows the performance of the all-to-all communication on the HA-PACS base cluster. Since the peak communication bandwidth of the HA-PACS base cluster is 8 GB/s per node, the peak communication bandwidth per MPI process is 2 GB/s. However, the available bandwidth is reduced due to network contention and the startup overhead.

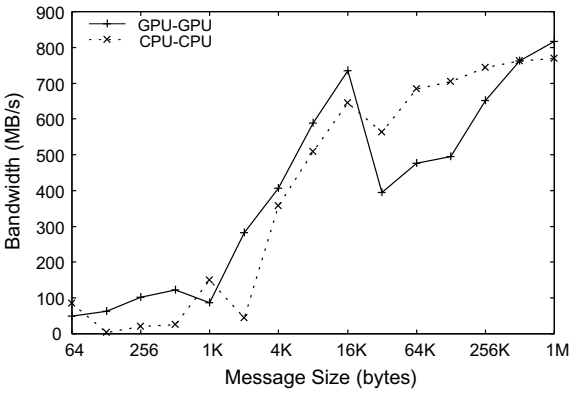

Fig. 8.14 Performance of all-to-all communication on the HA-PACS base cluster (128 nodes, 512 MPI processes)

As can be seen from Figs. 8.13 and 8.14, all-to-all communication significantly affects the performance of the parallel one-dimensional FFTs. We note that the performance was more than 1.31 TFlops with size $N = 2^{34}$ in the six-step FFT-based parallel FFT on 128 nodes of the HA-PACS base cluster, as shown in Table 8.9.

References

1. 2DECOMP & FFT—Library for 2D Pencil Decomposition and Distributed FFTs. http://www.2decomp.org/
2. AccFFT. http://accfft.org/
3. FFTE: A Fast Fourier Transform Package. http://www.ffte.jp/
4. FFTW Home Page. http://www.fftw.org/
5. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
6. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/
7. P3DFFT Home Page: Scalable Spectral transforms in 3D - P3DFFT. https://www.p3dfft.net/
8. TOP500 Supercomputer Sites. https://www.top500.org/
9. Intel math kernel library developer reference. https://software.intel.com/sites/default/files/managed/ff/c8/mkl-2017-developer-reference-c_0.pdf (2017)
10. CUDA Fortran Programming Guide and Reference. https://www.pgroup.com/resources/docs/19.1/pdf/pgi19cudaforug.pdf (2019)
11. CUFFT Library User's Guide. https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf (2019)
12. Agarwal, R.C., Gustavson, F.G., Zubair, M.: A high performance parallel algorithm for 1-D FFT. In: Proceedings of Supercomputing '94, pp. 34–40 (1994)
13. Almási, G., Archer, C.J., Erway, C.C., Heidelberger, P., Martorell, X., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of MPI collective communication on BlueGene/L systems. In: Proceedings of 19th ACM International Conference on Supercomputing (ICS'05), pp. 253–262 (2005)

14. Ayala, O., Wang, L.P.: Parallel implementation and scalability analysis of 3D fast Fourier transform using 2D domain decomposition. Parallel Comput. **39**, 58–77 (2013)
15. Bonelli, A., Franchetti, F., Lorenz, J., Püschel, M., Ueberhuber, C.W.: Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: Proceedings of 4th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2006). Lecture Notes in Computer Science, vol. 4330, pp. 818–832. Springer-Verlag, Berlin (2006)
16. Brass, A., Pawley, G.S.: Two and three dimensional FFTs on highly parallel computers. Parallel Comput. **3**, 167–184 (1986)
17. Calvin, C.: Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication. Parallel Comput. **22**, 1255–1279 (1986)
18. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: Proceedings of 24th International Conference on Supercomputing (ICS'10) , pp. 315–324 (2010)
19. Doi, J., Negishi, Y.: Overlapping methods of all-to-all communication and FFT algorithms for torus-connected massively parallel supercomputers. In: Proceedings of 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10) (2010)
20. Edelman, A., McCorquodale, P., Toledo, S.: The future fast Fourier transform? SIAM J. Sci. Comput. **20**, 1094–1114 (1999)
21. Eleftheriou, M., Fitch, B.G., Rayshubskiy, A., Ward, T.J.C., Germain, R.S.: Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: implementation and early performance measurements. IBM J. Res. Dev. **49**, 457–464 (2005)
22. Fang, B., Deng, Y., Martyna, G.: Performance of the 3D FFT on the 6D network torus QCDOC parallel supercomputer. Comput. Phys. Commun. **176**, 531–538 (2007)
23. Faraj, A., Kumar, S., Smith, B., Mamidala, A., Gunnels, J.: MPI collective communications on the Blue Gene/P supercomputer: algorithms and optimizations. In: Proceedings of 17th IEEE Symposium on High Performance Interconnects (HOTI'09), pp. 63–72 (2009)
24. Faraj, A., Yuan, X.: Automatic generation and tuning of MPI collective communication routines. In: Proceedings of 19th ACM International Conference on Supercomputing (ICS'05), pp. 393–402 (2005)
25. Faraj, A., Yuan, X., Lowenthal, D.: STAR-MPI: self tuned adaptive routines for MPI collective operations. In: Proceedings of 20th ACM International Conference on Supercomputing (ICS'06), pp. 199–208 (2006)
26. Franchetti, F., Low, T.M., Popovici, D.T., Veras, R.M., Spampinato, D.G., Johnson, J.R., Püschel, M., Hoe, J.C., Moura, J.M.F.: SPIRAL: extreme performance portability. Proc. IEEE **106**, 1935–1968 (2018)
27. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**, 216–231 (2005)
28. Idomura, Y., Nakata, M., Yamada, S., Machida, M., Imamura, T., Watanabe, T., Nunami, M., Inoue, H., Tsutsumi, S., Miyoshi, I., Shida, N.: Communication-overlap techniques for improved strong scaling of gyrokinetic eulerian code beyond 100k cores on the K-computer. Int. J. High Perform. Comput. Appl. **28**, 73–86 (2014)
29. Johnsson, S.L., Krawitz, R.L.: Cooley-Tukey FFT on the Connection Machine. Parallel Comput. **18**, 1201–1221 (1992)
30. Kumar, R., Mamidala, A., Panda, D.K.: Scaling alltoall collective on multi-core systems. In: Proceedings of 2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008) (2008)
31. Kumar, S., Sabharwal, Y., Garg, R., Heidelberger, P.: Optimization of all-to-all communication on the Blue Gene/L supercomputer. In: Proceedings of 37th International Conference on Parallel Processing (ICPP 2008), pp. 320–329 (2008)
32. Maeyama, S., Watanabe, T., Idomura, Y., Nakata, M., Nunami, M., Ishizawa, A.: Improved strong scaling of a spectral/finite difference gyrokinetic code for multi-scale plasma turbulence. Parallel Comput. **49**, 1–12 (2015)

33. Mirković, D., Johnsson, S.L.: Automatic performance tuning in the UHFFT library. In: Proceedings of 2001 International Conference on Computational Science (ICCS 2001). Lecture Notes in Computer Science, vol. 2073, pp. 71–80. Springer, Berlin (2001)
34. Nukada, A., Matsuoka, S.: Auto-tuning 3-D FFT library for CUDA GPUs. In: Proceedings of Conference on High Performance Computing Networking, Storage and Analysis (SC'09) (2009)
35. Nukada, A., Sato, K., Matsuoka, S.: Scalable multi-GPU 3-D FFT for TSUBAME 2.0 supercomputer. In: Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12) (2012)
36. Park, J., Bikshandi, G., Vaidyanathan, K., Tang, P.T.P., Dubey, P., Kim, D.: Tera-scale 1D FFT with low-communication algorithm and Intel Xeon Phi coprocessors. In: Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13) (2013)
37. Pekurovsky, D.: P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions. SIAM J. Sci. Comput. **34**, C192–C209 (2012)
38. Popovici, D.T., Low, T.M., Franchetti, F.: Large bandwidth-efficient FFTs on multicore and multi-socket systems. In: Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018), pp. 379–388 (2018)
39. Püschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: code generation for DSP transforms. Proc. IEEE **93**, 232–275 (2005)
40. Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in CUDA.http://www.cs.colostate.edu/~,cs675/MatrixTranspose.pdf (2009)
41. Sanders, P., Träff, J.L.: The hierarchical factor algorithm for all-to-all communication. In: Proceedings of 8th International Euro-Par Conference (Euro-Par 2002). Lecture Notes in Computer Science, vol. 2400, pp. 799–803. Springer, Berlin (2002)
42. Song, S., Hollingsworth, J.K.: Computation-communication overlap and parameter auto-tuning for scalable parallel 3-D FFT. Parallel Comput. **14**, 38–50 (2016)
43. Takahashi, D.: Efficient implementation of parallel three-dimensional FFT on clusters of PCs. Comput. Phys. Commun. **152**, 144–150 (2003)
44. Takahashi, D.: A parallel 1-D FFT algorithm for the Hitachi SR8000. Parallel Comput. **29**, 679–690 (2003)
45. Takahashi, D.: Automatic Tuning for Parallel FFTs, pp. 49–67. Springer, New York (2010)
46. Takahashi, D.: An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors. In: Proceedings of 8th International Conference on Parallel Processing and Applied Mathematics (PPAM 2009), Part I, Workshop on Memory Issues on Multi- and Manycore Platforms. Lecture Notes in Computer Science, vol. 6067, pp. 606–614. Springer, Berlin (2010)
47. Takahashi, D.: Implementation of parallel 1-D FFT on GPU clusters. In: Proceedings of 2013 IEEE 16th International Conference on Computational Science and Engineering (CSE 2013), pp. 174–180 (2013)
48. Takahashi, D.: Automatic tuning of computation-communication overlap for parallel 1-D FFT. In: Proceedings of 2016 IEEE 19th International Conference on Computational Science and Engineering (CSE 2016), pp. 253–256 (2016)
49. Takahashi, D., Boku, T., Sato, M.: A blocking algorithm for parallel 1-D FFT on clusters of PCs. In: Proceedings of 8th International Euro-Par Conference (Euro-Par 2002). Lecture Notes in Computer Science, vol. 2400, pp. 691–700. Springer, Berlin (2002)
50. Takahashi, D., Kanada, Y.: High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. J. Supercomput. **15**, 207–228 (2000)
51. Takahashi, D., Uno, A., Yokokawa, M.: An implementation of parallel 1-D FFT on the K computer. In: Proceedings of 2012 IEEE 14th International Conference on High Performance Computing and Communications (HPCC-2012), pp. 344–350 (2012)
52. Tang, P.T.P., Park, J., Kim, D., Petrov, V.: A framework for low-communication 1-D FFT. In: Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12) (2012)

53. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Automatically tuned collective communications. In: Proceedings of 2000 ACM/IEEE Conference on Supercomputing (SC'00) (2000)
54. Wang, H., Potluri, S., Luo, M., Singh, A.K., Sur, S., Panda, D.K.: MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. Comput. Sci. Res. Dev. **26**, 257–266 (2011)

Index

D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*,
High-Performance Computing Series 2, https://doi.org/10.1007/978-981-13-9965-7