FAST MULTIPLE-PRECISION ARITHMETIC ON DISTRIBUTED MEMORY PARALLEL COMPUTERS AND ITS APPLICATIONS

分散メモリ型並列計算機における高速多倍長計算とその応用

高 橋 大 介

(1)

FAST MULTIPLE-PRECISION ARITHMETIC ON DISTRIBUTED MEMORY PARALLEL COMPUTERS AND ITS APPLICATIONS

分散メモリ型並列計算機における高速多倍長計算とその応用

by

Daisuke Takahashi

高橋 大介

A Dissertation

Submitted to

The Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for The Degree of Doctor of Science
in Information Science

December 1998

Abstract

This thesis proposes highly efficient parallel algorithms for the multiple-precision addition, subtraction, multiplication, division and square root operation on distributed memory parallel computers.

It is well known that the fast Fourier transform (FFT) based multiplication can be used to implement multiplication of n-digit numbers in $O(n \log n \log \log n)$ operations. The conventional FFT-based algorithms multiply two n-digit numbers to obtain a $2n$-digit result. In the multiple-precision floating point multiplication, we need only the returned result whose precision is equal to the multiple-precision floating point number. This fact is exploited in our "dividing method" which is faster than the conventional FFT-based multiplication algorithm for the multiple-precision floating point numbers.

For an arbitrary-precision FFT-based multiplication, the number of points N in FFT is not necessarily 2^m. This thesis proposes high performance radix-2, 3 and 5 parallel 1-D FFT algorithms for $N = 2^p 3^q 5^r$ on distributed memory parallel computers. Experimental results of $2^p 3^q 5^r$ point parallel 1-D FFTs on distributed memory parallel computers are reported.

A parallel implementation of the real FFT-based multiplication is presented. This thesis proposes a parallelization of releasing propagated carries and borrows in the multiple-precision addition, subtraction and multiplication. In the parallel implementation of the Newton iteration based multiple-precision division and square root operation, there is a trade-off between load balance and communication overhead on distributed memory parallel computers. An efficient data distribution for the multiple-precision division and square root operation, is presented. A multiple-precision parallel division by single-precision integer, which is much faster than the multiple-precision division by a multiple-precision number, is proposed.

Moreover, improvements of the Gauss-Legendre algorithm and Borweins' quartically convergent algorithm for π calculation are proposed. The improved Gauss-Legendre algorithm is up to 1.08 times faster than the original Gauss-Legendre algorithm, and the improved Borweins' quartically convergent algorithm is up to 1.78 times faster than the original Borweins' quartically convergent algorithm.

Finally, this thesis shows how more than 137 billion decimal digits of the square root of 2 and more than 51.5 billion decimal digits of π are computed on the distributed memory parallel computer HITACHI SR2201 (1024 PEs). According to these results for the calculation of the mathematical constants, it is shown that our multiple-precision parallel arithmetic algorithms are quite useful for computing highly accurate mathematical constants.

Acknowledgments

First of all, I would like to express my appreciation to Professor Yasumasa Kanada for his support and discussions on this research.

I am grateful to Associate Professor Hiroyuki Sato for quite good comments about the manuscript. I appreciate discussions with members of Kanada laboratory.

I would like to thank Dr. Arthur Norman at Cambridge University. He gave me appropriate comments about multiple-precision arithmetic algorithms. I am pleased to acknowledge Dr. Aad van der Steen at Utrecht University for valuable comments about parallel FFTs. I wish to thank Dr. Yasunobu Torii at Fujitsu Ltd. He gave me suggestive comments and advised me to investigate the parallel division by short integer.

Contents

1 Introduction 8

1.1 Overview . . . 8

1.2 Sequential Multiple-Precision Arithmetic . . . 9

1.3 Parallelization of Multiple-Precision Arithmetic . . . 10

1.4 Organization . . . 11

2 Fast Multiple-Precision Multiplication Based on Dividing Method 13

2.1 Introduction . . . 13

2.2 Multiple-Precision Multiplication Based on the FFT . . . 14

2.3 The Dividing Multiple-Precision FFT-based Multiplication . . . 15

2.3.1 Multiple-Precision Multiplication Based on Dividing Method . . . 15

2.3.2 Multiple-Precision Square Operation Based on Dividing Method . . . 18

2.4 Experimental Results . . . 20

3 Implementation of Radix-2, 3 and 5 1-D FFT on Distributed Memory Parallel Computers 24

3.1 Introduction . . . 24

3.2 The Four-Step and Six-Step FFT Algorithms . . . 25

3.2.1 The Four-Step FFT . . . 25

3.2.2 The Six-Step FFT . . . 26

3.2.3 An Extended Three-Dimensional Four-Step FFT . . . 26

3.3 Parallel FFT Algorithm . . . 27

3.3.1 Algorithm (1) . . . 27

3.3.2 Algorithm (2) . . . 30

3.3.3 Adaptability of Parallel FFT Algorithms to Processor Architecture . . . 31

3.4 Radix-2, 3, 4 and 5 FFT Algorithm on a Single Processor . . . 31

3.4.1 The Radix-2 FFT . . . 32

3.4.2 The Radix-3 FFT . . . 32

3.4.3 The Radix-4 FFT . . . 33

3.4.4 The Radix-5 FFT . . . 34
3.4.5 Arithmetic Operation Counts . . . 35
3.5 Experimental Results of the Parallel FFT . . . 35
3.5.1 Experimental Results on the HITACHI SR2201 . . . 35
3.5.2 Experimental Results on the IBM SP2 . . . 37

4 Fast Multiple-Precision Addition, Subtraction and Multiplication on Distributed Memory Parallel Computers **39**
4.1 Introduction . . . 39
4.2 Parallelization of the Multiple-Precision Addition, Subtraction and Multiplication by Single-Precision Integer . . . 40
4.3 Parallelization of the Multiple-Precision Multiplication . . . 42
4.3.1 Multiple-Precision Multiplication Algorithm . . . 42
4.3.2 Parallelization of the Multiple-Precision Multiplication . . . 44
4.4 Experimental Results . . . 44

5 A Multiple-Precision Division by Single-Precision Integers on Distributed Memory Parallel Computers **48**
5.1 Introduction . . . 48
5.2 Algorithm . . . 49
5.3 Experimental Results . . . 52

6 Fast Multiple-Precision Calculation of Division and Square Root on Distributed Memory Parallel Computers **56**
6.1 Introduction . . . 56
6.2 Newton Iteration . . . 57
6.3 Parallelization of the Multiple-Precision Addition, Subtraction and Multiplication 57
6.4 Parallelization of the Multiple-Precision Division and Square Root Operation . . 58
6.4.1 Arithmetic Operation Counts . . . 58
6.4.2 Communication Time on Parallel Processing (Normalization) . . . 59
6.4.3 Total Computational Time . . . 60
6.5 Experimental Results . . . 60

7 Calculation of $\sqrt{2}$ to 137,438,950,000 Decimal Digits on the Distributed Memory Parallel Computer **66**
7.1 Introduction . . . 66
7.2 The Newton Iteration for Square Roots . . . 67
7.3 Multiple-Precision Arithmetic . . . 67

7.4 Results 68

8 Improvement of Algorithms for π Calculation **70**

8.1 Introduction 70

8.2 The Gauss-Legendre Algorithm 70

8.2.1 Improvement of the Gauss-Legendre Algorithm 72

8.3 Borweins' Quartically Convergent Algorithm 73

8.3.1 Improvement of Borweins' Quartically Convergent Algorithm 73

8.3.2 Improvement in the Final Iteration of Borweins' Quartically Convergent Algorithm 75

8.4 Experimental Results 77

9 Calculation of π to 51,539,600,000 Decimal Digits on the Distributed Memory Parallel Computer **79**

9.1 Introduction 79

9.2 Multiple-Precision Arithmetic 81

9.2.1 Multiple-Precision Addition, Subtraction and Multiplication 81

9.2.2 Multiple-Precision Reciprocal 82

9.2.3 Multiple-Precision Square Root 82

9.2.4 Multiple-Precision Reciprocal 4-th Root 82

9.3 Results of π 51,539,600,000 Decimal Digit Calculation 83

10 Conclusion **86**

List of Tables

2.1 Execution time of $\pi \times \sqrt{2}$ (in seconds). Underscored results are the minimum calculation time for each division. 22

2.2 Execution time of $\pi \times \pi$ (in seconds). Underscored results are the minimum calculation time for each division. 22

2.3 Execution time of real FFT (CPU TIME, $N = 2^m$). 23

3.1 Number of real operations for small-n transforms [83]. 31

3.2 Performance of parallel FFT algorithm (1) on the HITACHI SR2201 36

3.3 Performance of parallel FFT algorithm (2) on the HITACHI SR2201 36

3.4 All-to-all communication performance on the HITACHI SR2201 37

3.5 Performance of parallel FFT algorithm (1) on the IBM SP2 37

3.6 Performance of parallel FFT algorithm (2) on the IBM SP2 38

3.7 All-to-all communication performance on the IBM SP2 38

4.1 Execution time of multiple-precision parallel addition ($\pi+\sqrt{2}$) (in seconds), $N =$ number of decimal digits. 46

4.2 Execution time of multiple-precision parallel multiplication ($\pi \times \sqrt{2}$) (in seconds), $N =$ number of decimal digits. 47

5.1 Execution time of multiple-precision parallel division by a single-precision integer ($\pi/2$) (in seconds), $N =$ number of decimal digits. 54

5.2 Execution time of multiple-precision parallel division by a single-precision integer ($\pi/3$) (in seconds), $N =$ number of decimal digits. 55

6.1 Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, block distribution) (in seconds), $N =$ number of decimal digits. 62

6.2 Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, cyclic distribution) (in seconds), $N =$ number of decimal digits. 63

6.3 Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, block distribution) (in seconds), $N =$ number of decimal digits. 64

6.4 Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, cyclic distribution) (in seconds), N = number of decimal digits. 65

7.1 Frequency distribution for $\sqrt{2}-1$ up to 100,000,000,000 decimal digits. 68

8.1 Comparison with the number of operations in each iteration of the Gauss-Legendre algorithm. 73

8.2 Comparison with the number of operations in each iteration of Borweins' quartically convergent algorithm. 75

8.3 Comparison with the performance of the Gauss-Legendre algorithm (in seconds). 78

8.4 Comparison with the performance of Borweins' quartically convergent algorithm (in seconds). 78

9.1 Historical records of the π calculation by computers. 80

9.2 Frequency distribution for $\pi - 3$ up to 50,000,000,000 decimal digits. 85

9.3 Frequency distribution for $1/\pi$ up to 50,000,000,000 decimal digits. 85

List of Figures

2.1 Illustration of the dividing method. 15
2.2 Efficiency of the dividing method (multiplication). 19
2.3 Efficiency of the dividing method (square). 21

3.1 Performance of FFT kernel (radix-4) (HITACHI SR2201 1PE). 29

4.1 Multiple-precision sequential addition. 41
4.2 Multiple-precision parallel addition. 42
4.3 Parallel normalization with the carry skip method. 43
4.4 Execution time of multiple-precision parallel addition ($\pi + \sqrt{2}$), N = number of decimal digits. 46
4.5 Execution time of multiple-precision parallel multiplication ($\pi \times \sqrt{2}$), N = number of decimal digits. 47

5.1 The communication diagram for equation (5.14). 51
5.2 Execution time of multiple-precision parallel division by a single-precision integer ($\pi/2$), N = number of decimal digits. 54
5.3 Execution time of multiple-precision parallel division by a single-precision integer ($\pi/3$), N = number of decimal digits. 55

6.1 Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, block distribution), N = number of decimal digits. 62
6.2 Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, cyclic distribution), N = number of decimal digits. 63
6.3 Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, block distribution), N = number of decimal digits. 64
6.4 Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, cyclic distribution), N = number of decimal digits. 65

Chapter 1

Introduction

1.1 Overview

Numerical computations on modern computers are generally performed with arithmetic operations of *int*, *float* and *double* datatypes. 32-bit *int* datatype can only express numbers in the range of $-2^{31} \sim 2^{31} - 1$. The precision of 32-bit *float* datatype is about 6 decimal digits and the precision of 64-bit *double* datatype is about 15 decimal digits in IEEE 754 representation. Although quad precision is supported on several computers with software, its precision is at most 33 decimal digits.

In most scientific computations, these precisions are sufficient enough. However, for an arbitrary-precision computation, we need an efficient software of the multiple-precision arithmetic. This thesis proposes parallel algorithms for the multiple-precision arithmetic on distributed memory parallel computers.

Application fields of the multiple-precision arithmetic include

- RSA cryptosystem [64, 46],
- Integer factorization [23, 63, 80],
- Mersenne prime search [53, 62],
- Highly accurate calculation of mathematical constants (π, e, $\sqrt{2}$, etc.) [61, 67, 32, 82, 7, 8, 41, 76, 78], and
- Symbolic and algebraic computation [37, 31].

Two factors may set a limit to the very high precision with which mathematical constants can be calculated:

(a) The efficiency of the algorithms used.

(b) The total amount of memory available.

The typical CPU-intensive calculations that have been performed are integer factorization and Mersenne prime search which are performed on large clusters of workstations or personal computers. For 10 to 100 billion digit mathematical constants calculation, above factor (b) is critical, since so the machines with the order of 0.1 TB (Tera Bytes) to 1 TB are needed.

To perform the multiple-precision calculation at high speed, vector processing oriented implementations have been proposed [41, 24, 11]. The processing speed and main memory size of the vector computers are becoming saturated.

Therefore a parallel processing by a distributed memory parallel computer is one of the solutions for the very high precision calculation. However, because of communication overhead and load imbalance, it is not easy to obtain its peak performance. Thus it is quite important to develop parallel algorithms that can be implemented efficiently on distributed memory parallel computers.

1.2 Sequential Multiple-Precision Arithmetic

The multiple-precision sequential algorithms are described in references [5, 45]. The arithmetic operation count of n-digit multiple-precision sequential addition, subtraction and multiplication by single-precision integer is clearly $O(n)$.

A key operation in the fast multiple-precision arithmetic is the multiplication, by which significant time in the total computation is spent. The multiple-precision multiplication of n-digit numbers requires $O(n^2)$ operations by using ordinary multiplication algorithm [45]. Karatsuba's algorithm [43, 45] is known to reduce the number of operations to $O(n^{\log_2 3})$.

On the other hand, it is well known that the multiple-precision multiplication of n-digit numbers can be performed in $O(n \log n \log \log n)$ operations by using the Schönhage-Strassen algorithm [66, 5, 45] which is the algorithm based on the fast Fourier transform (FFT) [27, 22, 58, 86]. In the multiple-precision multiplication of several thousand decimal digits or more, the FFT-based multiplication is the fastest.

These conventional FFT-based algorithms multiply two n-digit numbers to obtain a $2n$-digit result [82, 41, 8, 19]. However, in the multiple-precision floating point multiplication, we need only the returned result whose precision is equal to the multiple-precision floating point number.

This fact is exploited in our "dividing method" which is faster than the conventional FFT-based multiplication algorithm for the multiple-precision floating point numbers. In references [47, 48, 56], algorithms for the multiple-precision floating point multiplication are shown. However, they use the ordinary $O(n^2)$ multiplication algorithm or Karatsuba's $O(n^{\log_2 3})$ algorithm which is asymptotically slower than the FFT-based multiplication algorithm. We show that the overall arithmetic operations for the multiple-precision FFT-based multiplication are reduced by decomposition of the full length FFT-based multiplication into shorter length FFT-based

multiplication.

The multiple-precision division and square root operation take considerably longer time to compute than the addition, subtraction and multiplication. There are a number of ways to perform division and square root operation [45]. It is well known that the multiple-precision division and square root operation can be reduced to the multiple-precision addition, subtraction and multiplication by using the Newton iteration [45]. This scheme requires $O(M(n))$ operations, where $M(n)$ is the number of operations for n-digit multiplication.

Several software packages are available for the multiple-precision computation [21, 24, 9, 36, 69, 11, 12]. Brent's MP multiple-precision package [21] is probably the most widely used of these packages at present, due to its greater functionality and efficiency. D. M. Smith made a similar package which features improved performance for certain transcendental functions [69]. Another package available at some sites is MPFUN made by D. H. Bailey [11]. One of the key features in the MPFUN package is that package is optimized for vector supercomputers and RISC processors.

1.3 Parallelization of Multiple-Precision Arithmetic

Parallel implementation of the multiple-precision arithmetic on a shared memory machine was reported by K. Weber [87]. Weber modified the MPFUN package [11] to run in parallel on a shared memory multiprocessor. However, a complete solution of the sequential bottleneck in the normalization of the result (carry/borrow propagation) is not presented, and the multiple-precision division is also not discussed. This thesis shows that these propagation operations can be parallelized by the carry skip method [52].

Parallel implementation of Karatsuba's multiplication algorithm was proposed by G. Cesari and R. Maeder [25] on a distributed memory parallel computer. In the multiple-precision multiplication of several thousand decimal digits or more, the FFT-based multiplication is the fastest. FFT-based multiplication algorithms are known to be good candidates for the parallel implementation. B. S. Fagin [33, 35] used the Fermat number transform (FNT) [60, 1, 2, 58] for large integer multiplication on the Connection Machine CM-2. However, FNT uses many modulo operations which are slow because of integer division.

This thesis proposes the real FFT-based multiple-precision parallel multiplication on distributed memory parallel computers. In this scheme, a high performance parallel 1-D FFT routine is needed. Many papers have proposed parallel 1-D complex and real FFT algorithms with radix-2 [73, 6, 40, 4, 38]. However, for an arbitrary-precision FFT-based multiplication, the number of points N in FFT is not necessarily 2^m. This thesis also proposes the radix-2, 3 and 5 parallel 1-D FFT algorithms for $N = 2^p 3^q 5^r$ on distributed memory parallel computers. By using the radix-2, 3 and 5 parallel 1-D FFT, we can reduce the arithmetic operations and

memory size of the multiple-precision FFT-based multiplications.

Parallel computation of $\sqrt{2}$ up to 1 million decimal digits has been performed by B. Char et al. [26] on a network of workstations in 1994. However, they did not propose the multiple-precision parallel division and they implemented a parallel version of Karatsuba's multiplication algorithm.

This thesis also proposes the FFT-based multiple-precision parallel division and square root operation. In the parallel implementation of the Newton iteration based multiple-precision division and square root operation, there is a trade-off between load balance and communication overhead on distributed memory parallel computers [75]. This is because the Newton iteration is performed by doubling the precision for each iteration. This thesis proposes an efficient data distribution for the multiple-precision division and square root operation.

On the other hand, the multiple-precision division by single-precision integer is also used in the multiple-precision arithmetic, which is much faster than the division by a multiple-precision number. Although several multiple-precision arithmetic packages [21, 24, 11] include a routine of the multiple-precision division by single-precision integer, the multiple-precision parallel division by single-precision integer has not been presented so far.

This thesis proposes the multiple-precision numbers can be divided by single-precision integer in parallel. Although this "division by short integer" includes a first-order recurrence, we can apply the parallel cyclic reduction method [39].

1.4 Organization

This thesis is organized as follows. Chapter 2 studies a fast multiple-precision multiplication based on the dividing method.

Chapter 3 gives an implementation of radix-2, 3 and 5 1-D FFTs on distributed memory parallel computers. Experimental results of $N = 2^p 3^q 5^r$ point FFTs on the distributed memory parallel computers HITACHI SR2201 and IBM SP2 are described.

Chapter 4 studies a fast multiple-precision addition, subtraction and multiplication on distributed memory parallel computers. A parallel implementation of the real FFT-based multiplication is presented, because a key operation in the fast multiple-precision arithmetic is the multiplication. We also propose a parallelization of releasing propagated carries and borrows in the multiple-precision addition, subtraction and multiplication.

In Chapter 5, a multiple-precision parallel division by single-precision integer is presented.

Chapter 6 studies a fast multiple-precision calculation of division and square root operation on distributed memory parallel computers. An efficient data distribution for the multiple-precision division and square root operation by using the Newton iteration is discussed.

Chapter 7 describes the calculation of $\sqrt{2}$ up to 137,438,950,000 decimal digits have been

computed and verified on a distributed memory parallel computer. The calculation is based on the classical Newton iteration for the reciprocal of the square root. Since the Newton iteration has the second order convergence nature, it can be performed by doubling the precision for each iteration.

Chapter 8 gives improvements of the Gauss-Legendre algorithm and Borweins' quartically convergent algorithm for π calculation, in which the number of the multiple-precision multiplication and square operation is reduced.

Chapter 9 focuses on the calculation of π up to 51,539,600,000 decimal digits. Calculation is based on Borweins' quartically convergent formula. Correctness of the calculation was verified through arithmetic-geometric mean formula discovered independently by R. P. Brent and E. Salamin in 1976. Details of the computation and statistical tests on the first 50 billion digits of π are explained.

Chapter 10 gives conclusion of this thesis.

Chapter 2

Fast Multiple-Precision Multiplication Based on Dividing Method

2.1 Introduction

Many multiple-precision multiplication algorithms have been proposed [45]. The multiple-precision multiplication of n-digit numbers requires $O(n^2)$ operations by using ordinary multiplication algorithm [45]. Karatsuba's algorithm [43] is known to reduce the number of operations to $O(n^{\log_2 3})$.

Schönhage-Strassen's $O(n \log n \log \log n)$ algorithm [66, 45] is known as the fastest multiple-precision algorithm. However, this algorithm may not be able to exploit computers with fast floating point hardware and needs binary to decimal radix conversion for the final result. D. H. Bailey also used discrete Fourier transform with three prime modulo computation followed by the reconstruction through Chinese Remainder Theorem for his 29 million decimal digit π calculation [8].

On the other hand, the multiple-precision multiplication algorithm using the real fast Fourier transform (FFT) [14] is known as another fast multiplication algorithm [82, 41, 19]. In this chapter, we discuss this algorithm.

These conventional FFT-based algorithms multiply two n-digit numbers to obtain a $2n$-digit result. In the multiple-precision floating point multiplication, we need only the returned result whose precision is equal to the multiple-precision floating point number. We will call it "short product" here. In [47, 48, 56], algorithms for multiple-precision floating point multiplication are shown. They used the ordinary $O(n^2)$ multiplication algorithm or Karatsuba's $O(n^{\log_2 3})$ algorithm. However, in the multiple-precision multiplication of several thousand decimal digits

or more, the FFT-based multiplication is the fastest. We show that the overall arithmetic operations for the multiple-precision FFT-based multiplication are reduced by decomposition of the full length FFT-based multiplication into shorter length multiple-precision FFT-based multiplication.

For simplicity, we use the short product which do not provide exact rounding. It is not hard to extend the multiple-precision multiplication with exact rounding [47, 48, 56].

2.2 Multiple-Precision Multiplication Based on the FFT

Throughout the chapter, we discusses a case of the multiple-precision multiplication based on the FFT.

Let us consider the product Z of two integers X and Y with length n and radix B. Note that the radix B need not to be a power of 2.

$$X = \sum_{i=0}^{2n-1} x_i B^i, \quad Y = \sum_{i=0}^{2n-1} y_i B^i,$$

where $0 \le x_i < B$, $0 \le y_i < B$ for $0 \le i < n-1$, $0 < x_{n-1} < B$, $0 < y_{n-1} < B$ and $x_i = y_i = 0$, for $n \le i$.

Then,

$$Z \equiv X \cdot Y = \Big(\sum_{i=0}^{2n-1} x_i B^i\Big) \cdot \Big(\sum_{j=0}^{2n-1} y_j B^j\Big) = \sum_{i=0}^{2n-1} B^i \Big(\sum_{j=0}^{2n-1} x_j y_{i-j}\Big) \equiv \sum_{k=0}^{2n-1} z_k B^k.$$

Thus, z_k can be written as follows:

$$z_k = C_k(x, y) = \sum_{j=0}^{2n-1} x_j y_{k-j}, \quad \text{for } k = 0, \cdots, 2n-2 \text{ and } z_{2n-1} = 0.$$

where the subscript $k-j$ is to be interpreted as $k-j+2n$ if $k-j$ is negative.

Now, the discrete Fourier transform (DFT) and the inverse DFT are given by

$$F_k(x) = \sum_{j=0}^{N-1} x_j e^{-2\pi i jk/N}, \tag{2.1}$$

$$F_k^{-1}(x) = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{2\pi i jk/N}, \tag{2.2}$$

where $N = 2n$.

Then the convolution theorem for discrete sequences states that

$$F[C(x, y)] = F(x)F(y).$$

Let $C(x, y)$ denotes the convolution of the sequences x and y:

$$C(x, y) = F^{-1}[F(x)F(y)].$$

x_0	x_1	x_2	x_3			
y_0	y_1	y_2	y_3			
x_0y_0	x_0y_1	x_0y_2	x_0y_3			
	x_1y_0	x_1y_1	x_1y_2	x_1y_3		
		x_2y_0	x_2y_1	x_2y_2	x_2y_3	
			x_3y_0	x_3y_1	x_3y_2	x_3y_3
z_0	z_1	z_2	z_3			

Figure 2.1: Illustration of the dividing method.

We can use the FFT to calculate the DFT in (2.1) and (2.2). The arithmetic operation count of N point FFT is $O(N \log N)$ [27].

2.3 The Dividing Multiple-Precision FFT-based Multiplication

2.3.1 Multiple-Precision Multiplication Based on Dividing Method

Let us consider the multiple-precision multiplication of n-digit numbers based on dividing method.

We show the multiple-precision FFT-based multiplication based on 4-way division between X and Y, both of which are n-digit numbers with radix B. Let us consider the multiple-precision multiplication, such as (n-digit number) $\times$ (n-digit number) $\rightarrow$ (n-digit number):

$$X = x_0 \cdot B^{3n/4} + x_1 \cdot B^{n/2} + x_2 \cdot B^{n/4} + x_3, \qquad (0 \le x_i < B^{n/4}),$$
$$Y = y_0 \cdot B^{3n/4} + y_1 \cdot B^{n/2} + y_2 \cdot B^{n/4} + y_3, \qquad (0 \le y_i < B^{n/4}).$$

The returned results of upper half n-digits between X and Y are:

$$\begin{aligned} Z &\equiv X \cdot Y \\ &= F^{-1}[F(x_0)F(y_0)] \cdot B^{3n/2} + F^{-1}[F(x_0)F(y_1) + F(x_1)F(y_0)] \cdot B^{5n/4} \\ &\quad + F^{-1}[F(x_0)F(y_2) + F(x_1)F(y_1) + F(x_2)F(y_0)] \cdot B^{n} \\ &\quad + F^{-1}[F(x_0)F(y_3) + F(x_1)F(y_2) + F(x_2)F(y_1) + F(x_3)F(y_0)] \cdot B^{3n/4} \\ &= z_0 \cdot B^{3n/2} + z_1 \cdot B^{5n/4} + z_2 \cdot B^{n} + z_3 \cdot B^{3n/4}. \end{aligned}$$

The illustration of the dividing method is shown in Figure 2.1.

When each partial multiplication x_iy_j $(0 \le i \le 3,\ 0 \le j \le 3,\ 0 \le i+j \le 3)$ is computed, it is necessary to compute the forward Fourier transform of x_i $(0 \le i \le 3)$ and y_j $(0 \le j \le 3)$.

If we preserve the Fourier coefficients $F(x_i)$ $(0 \le i \le 3)$ and $F(y_j)$ $(0 \le j \le 3)$, total number of the forward Fourier transform is 8.

Since the Fourier coefficients are added up at the same position after the inverse Fourier transforms are computed, total number of the inverse Fourier transforms is 10 ($= \sum_{i=1}^{4} i$). However, we can utilize the linearity of the Fourier transform. First, we compute the addition of the Fourier coefficients at the same position. Since we compute the inverse Fourier transform of these data, we can reduce the number of the inverse Fourier transform to 4.

We assume that the arithmetic operation count of $N(= 2n)$ point FFT is to be $c_{fft} \cdot N \log_2 N$ and the arithmetic operation count of N point Fourier coefficient product is to be $c_{mult} \cdot N$, and that of N point Fourier coefficient addition is to be $c_{add} \cdot N$.

If we do not divide the multiple-precision number (i.e. the conventional FFT-based multiplication algorithm), the arithmetic operation count T_{nodiv}^{mult} is as follows:

$$T_{nodiv}^{mult} = 3c_{fft} \cdot N \log_2 N + c_{mult} \cdot N.$$

The total arithmetic operation count of the multiplication with 4-way dividing, e.g. T_{fft}, is as follows:

$$\begin{aligned} T_{fft} &= 3 \cdot \left(4 \cdot c_{fft} \cdot \frac{N}{4} \log_2 \frac{N}{4} \right) \\ &= 3c_{fft} \cdot N(\log_2 N - 2). \end{aligned}$$

The total arithmetic operation count T_{mult} for the product of the Fourier coefficient is:

$$\begin{aligned} T_{mult} &= \left(\sum_{i=1}^{4} i \right) \cdot c_{mult} \cdot \frac{N}{4} \\ &= \frac{5}{2} c_{mult} \cdot N. \end{aligned}$$

Hence, the total arithmetic operation count T_{add} for the addition of the Fourier coefficient at the same position is as follows:

$$\begin{aligned} T_{add} &= \left(\sum_{i=1}^{3} i \right) \cdot c_{add} \cdot \frac{N}{4} \\ &= \frac{3}{2} c_{add} \cdot N. \end{aligned}$$

The total arithmetic operation count $T_{overlap}$ for the addition of the inverse Fourier transformed data at the same position which is overlapped is as follows:

$$T_{overlap} = 3 \cdot c_{add} \cdot \frac{N}{8}.$$

Hence, the total arithmetic operation count of multiplication with 4-way dividing, T_{div4}^{mult} is as follows:

$$T_{div4}^{mult} = T_{fft} + T_{mult} + T_{add} + T_{overlap}$$

$$
\begin{aligned}
&= 3c_{fft} \cdot N(\log_2 N - 2) + \frac{5}{2} c_{mult} \cdot N \\
&\quad + \frac{3}{2} c_{add} \cdot N + \frac{3}{8} c_{add} \cdot N \\
&= T_{modi}^{mult} + \left(-6c_{fft} + \frac{5}{2} c_{mult} + \frac{15}{8} c_{add} \right) N.
\end{aligned}
$$

In the same way, we show the multiple-precision FFT-based multiplication based on d-way division.

$$
\begin{aligned}
&X = \sum_{i=0}^{d-1} x_i B^{\frac{d-1-i}{d} \cdot n}, \quad Y = \sum_{i=0}^{d-1} y_i B^{\frac{d-1-i}{d} \cdot n}, \\
&U_i = F(x_i), \quad V_i = F(y_i), \quad 0 \le i \le d-1, \\
&z_i = F^{-1} \left[\sum_{j=0}^{i} U_j V_{i-j} \right], \quad 0 \le i \le d-1, \\
&Z \equiv X \cdot Y = \sum_{i=0}^{d-1} z_i B^{\frac{2d-2-i}{d} \cdot n}.
\end{aligned}
$$

The total arithmetic operation count of multiplication with d-way dividing, e.g. T_{fft}, is as follows:

$$
\begin{aligned}
T_{fft} &= 3 \cdot \left(d \cdot c_{fft} \cdot \frac{N}{d} \log_2 \frac{N}{d} \right) \\
&= 3c_{fft} \cdot N(\log_2 N - \log_2 d).
\end{aligned}
$$

The total arithmetic operation count T_{mult} for the product of the Fourier coefficient is:

$$
\begin{aligned}
T_{mult} &= \left(\sum_{i=1}^{d} i \right) \cdot c_{mult} \cdot \frac{N}{d} \\
&= \frac{d+1}{2} \cdot c_{mult} \cdot N.
\end{aligned}
$$

Hence, the total arithmetic operation count T_{add} for the addition of the Fourier coefficient at the same position is as follows:

$$
\begin{aligned}
T_{add} &= \left(\sum_{i=1}^{d-1} i \right) \cdot c_{add} \cdot \frac{N}{d} \\
&= \frac{d-1}{2} \cdot c_{add} \cdot N.
\end{aligned}
$$

The total arithmetic operation count $T_{overlap}$ for the addition of the inverse Fourier transformed data at the same position which is overlapped is as follows:

$$
\begin{aligned}
T_{overlap} &= (d-1) \cdot c_{add} \cdot \frac{N}{2d} \\
&= \frac{d-1}{2d} \cdot c_{add} \cdot N.
\end{aligned}
$$

Hence, the total arithmetic operation count for the multiplication with d-way dividing, e.g. T_{div}^{mult}, is as follows:

$$\begin{aligned} T_{div}^{mult} &= T_{fft} + T_{mult} + T_{add} + T_{overlap} \\ &= 3c_{fft} \cdot N(\log_2 N - \log_2 d) + \frac{d+1}{2} \cdot c_{mult} \cdot N \\ &\quad + \frac{d-1}{2} \cdot c_{add} \cdot N + \frac{d-1}{2d} \cdot c_{add} \cdot N \\ &= T_{nodiv}^{mult} + \left(-3c_{fft} \cdot \log_2 d + \frac{d-1}{2} \cdot c_{mult} + \frac{d^2-1}{2d} \cdot c_{add} \right) N. \end{aligned}$$

Then, we can compute the optimal d, to minimize T_{div}^{mult}, i.e.

$$\begin{aligned} (T_{div}^{mult})' &\approx -3c_{fft} \cdot N \cdot \frac{\log_2 e}{d} + \frac{1}{2} c_{mult} \cdot N + \frac{1}{2} c_{add} \cdot N \\ &\equiv 0. \end{aligned}$$

It shows that

$$d \approx \frac{6c_{fft} \log_2 e}{c_{mult} + c_{add}},$$

where d is an integer value and it is independent of N.

The arithmetic operation count of N point real FFT is $(5/2)N \log_2 N$ [14]. Thus, we assume that c_{fft} is 2.5.

To multiply a complex number, in general, 4 times real number multiplications and 2 times real number additions are necessary [45]. Thus, we assume that c_{mult} is 3. The optimal value d is $d \approx 5.41$ when assuming c_{add} is 1. Thus, when we use radix-2 FFT, the optimal value d is 4 or 8, because d is an integer.

We show the ratio of $T_{nodiv}^{mult}/T_{div}^{mult}$ when d and data point $N(=2n)$ are varied in Figure 2.2. The ratio $T_{nodiv}^{mult}/T_{div}^{mult}$ shows the improvement that the dividing method provides over the conventional FFT-based multiplication algorithm. For $d = 4$ or $d = 8$ and $N = 2^9$, we can see that the dividing method is better than conventional FFT-based multiplication algorithm.

2.3.2 Multiple-Precision Square Operation Based on Dividing Method

Let us assume that the arithmetic operation count of $N(=2n)$ point FFT is to be $c_{fft} \cdot N \log_2 N$.

We assume the arithmetic operation count of N point Fourier coefficient product is to be $c_{square} \cdot N$ and assume that of N point Fourier coefficient addition is to be $c_{add} \cdot N$.

If we do not divide the multiple-precision number (i.e. the conventional FFT-based multiplication algorithm), the total arithmetic operation count T_{nodiv}^{square} is as follows:

$$T_{nodiv}^{square} = 2c_{fft} \cdot N \log_2 N + c_{square} \cdot N.$$

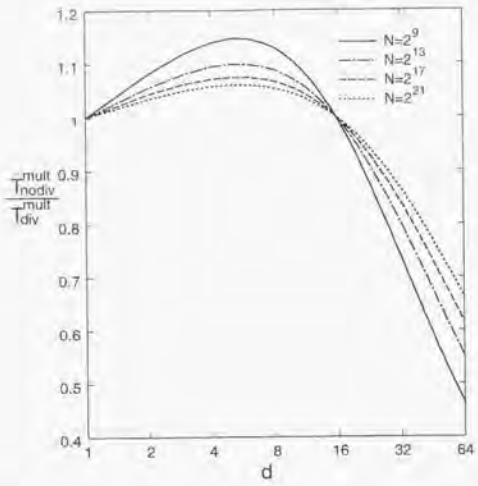


Figure 2.2: Efficiency of the dividing method (multiplication).

Next, we compute the case of square operation with d-way dividing. The total arithmetic operation count of the multiplication with d-way dividing, e.g. T_{fft}, is as follows:

$$\begin{aligned} T_{fft} &= 2d \cdot c_{fft} \cdot \frac{N}{d} \log_2 \frac{N}{d} \\ &= 2c_{fft} \cdot N(\log_2 N - \log_2 d). \end{aligned}$$

Here, we have to take care of the symmetry of the square operation. The arithmetic operation count T_{mult} for the product of the Fourier coefficient is half of that in the case of multiplication. Thus, the total arithmetic operation count T_{mult} is as follows:

$$\begin{aligned} T_{mult} &= \frac{1}{2} \left\{ \left(\sum_{i=1}^{d} i \right) - \frac{d}{2} \right\} \cdot c_{mult} \cdot \frac{N}{d} + \frac{d}{2} \cdot c_{square} \cdot \frac{N}{d} \\ &= \frac{d}{4} \cdot c_{mult} \cdot N + \frac{1}{2} c_{square} \cdot N. \end{aligned}$$

Hence, the total arithmetic operation count T_{add} for the addition of the Fourier coefficient at the same position is as follows:

$$\begin{aligned} T_{add} &= \left(\sum_{i=1}^{d-1} i \right) \cdot c_{add} \cdot \frac{N}{d} \\ &= \frac{d-1}{2} \cdot c_{add} \cdot N. \end{aligned}$$

The total arithmetic operation count $T_{overlap}$ for the addition of the inverse Fourier transformed data at the same position which is overlapped is as follows:

$$\begin{aligned} T_{overlap} &= (d-1) \cdot c_{add} \cdot \frac{N}{2d} \\ &= \frac{d-1}{2d} \cdot c_{add} \cdot N. \end{aligned}$$

Hence, the total arithmetic operation count in the square operation with d-way dividing, e.g. T_{div}^{square}, is as follows:

$$\begin{aligned} T_{div}^{square} &= T_{fft} + T_{mult} + T_{add} + T_{overlap} \\ &= 2c_{fft} \cdot N(\log_2 N - \log_2 d) + \frac{d}{4} \cdot c_{mult} \cdot N + \frac{1}{2} c_{square} \cdot N \\ &\quad + \frac{d-1}{2} \cdot c_{add} \cdot N + \frac{d-1}{2d} \cdot c_{add} \cdot N \\ &= T_{nodiv}^{square} + \left(-2c_{fft} \cdot \log_2 d + \frac{d-4}{4} \cdot c_{mult} + \frac{1}{2} c_{square} + \frac{d^2-1}{2d} \cdot c_{add} \right) N. \end{aligned}$$

Then, we can compute the optimal d from

$$\begin{aligned} (T_{div}^{square})' &\approx -2c_{fft} \cdot N \cdot \frac{\log_2 e}{d} + \frac{1}{4} c_{mult} \cdot N + \frac{1}{2} c_{add} \cdot N \\ &\equiv 0. \end{aligned}$$

The formula shows that the optimal value of d is as follows:

$$d \approx \frac{8c_{fft} \log_2 e}{c_{mult} + 2c_{add}}.$$

Then, the optimal value d is 5.77. Thus, when we use radix-2 FFT, the optimal value of d is 4 or 8 because d is an integer.

We show the ratio of $T_{nodiv}^{square} / T_{div}^{square}$ when d and data point $N(= 2n)$ are varied in Figure 2.3. The ratio $T_{nodiv}^{square} / T_{div}^{square}$ shows the improvement that the dividing method provides over the conventional FFT-based multiplication algorithm. For $d = 4$ or $d = 8$ and $N = 2^9$, we can see that the dividing method is better than conventional FFT-based multiplication algorithm.

2.4 Experimental Results

To evaluate our algorithms, the calculation of $\pi \times \sqrt{2}$ and $\pi \times \pi$ were performed on a main frame machine of HITACHI MP-5800.

All routines were written in FORTRAN. The compiler used was optimized FORTRAN 77 of Hitachi Ltd. As for a optimization option, `-W0,'opt(o(s),approx(0))'` was specified.

The execution times when number d of division and n are varied are shown in Tables 2.1 and 2.2, respectively.

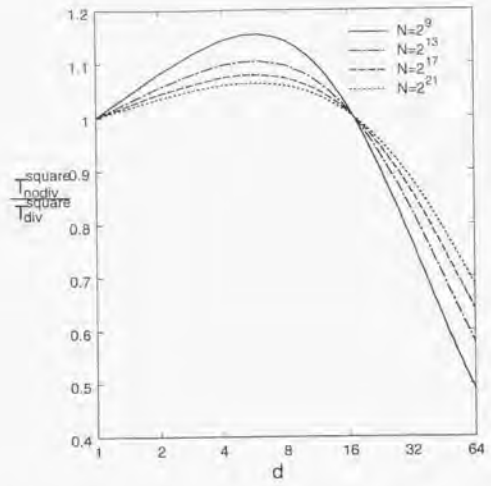


Figure 2.3: Efficiency of the dividing method (square).

In the cases of $n = 2^{12}$ and $n = 2^{16}$, the tendency of the observed values in Tables 2.1 and 2.2 is almost the same as the theoretical one. However, in the cases of $n = 2^8$ and $n = 2^{20}$, the tendency of the observed values in Tables 2.1 and 2.2 is different from the theoretical one. In the case of $d = 1$ (not dividing) and $n = 2^8$, the execution times are reduced in Tables 2.1 and 2.2. By using the dividing method, in the cases of $d = 16$ and $n = 2^{20}$, the execution time is reduced to $5.242/9.995 \approx 1/1.91$ in Table 2.1. and $4.197/6.342 \approx 1/1.51$ in Table 2.2.

Hereafter, we consider these causes. In the case of $n = 2^8$, the overhead of DO-loop is larger than in the case of $d = 1$ (because the loop length of innermost loop is shortened). In the case of $n = 2^{20}$, the cache misses occur frequently since the memory size of the FFT is 32 MB. However, the cache misses do not occur easily in the dividing method because the multiple-precision numbers are divided, and working set size becomes small.

The execution time of real FFT is shown in Table 2.3 which shows that the performance decreases especially for $N \geq 2^{19}$. For computing the MFLOPS rate in Table 2.3, we used the theoretical arithmetic operation count of $2.5N \log_2 N$ for N point real FFT.

In the dividing method, the arithmetic operations can be reduced and the cache miss is easily reduced for large digit processing.

Table 2.1: Execution time of $\pi \times \sqrt{2}$ (in seconds). Underscored results are the minimum calculation time for each division.

d	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
1	7.754×10^{-4}	0.01268	0.2466	9.995
2	7.920×10^{-4}	0.01091	0.2149	9.047
4	9.667×10^{-4}	0.01011	0.1988	7.246
8	1.313×10^{-3}	0.01037	0.2047	6.266
16	2.005×10^{-3}	0.01094	0.2165	5.242
32	3.557×10^{-3}	0.01379	0.2394	5.404
64	7.104×10^{-3}	0.02048	0.3232	6.901

Table 2.2: Execution time of $\pi \times \pi$ (in seconds). Underscored results are the minimum calculation time for each division.

d	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
1	4.122×10^{-4}	7.234×10^{-3}	0.1435	6.342
2	4.312×10^{-4}	7.068×10^{-3}	0.1416	6.198
4	4.900×10^{-4}	6.851×10^{-3}	0.1382	5.320
8	5.865×10^{-4}	6.926×10^{-3}	0.1408	5.004
16	7.996×10^{-4}	7.560×10^{-3}	0.1460	4.197
32	1.341×10^{-3}	0.01073	0.1916	4.466
64	2.713×10^{-3}	0.01641	0.2714	5.894

Table 2.3: Execution time of real FFT (CPU TIME, $N = 2^m$).

m	TIME (sec)	MFLOPS
9	1.085×10^{-4}	105.98
10	2.338×10^{-4}	109.45
11	5.110×10^{-4}	110.22
12	1.075×10^{-3}	114.31
13	2.331×10^{-3}	114.22
14	5.051×10^{-3}	113.53
15	0.01078	113.99
16	0.02284	114.76
17	0.04977	111.93
18	0.1198	98.43
19	0.3929	63.38
20	1.247	42.05
21	2.899	37.98

Chapter 3

Implementation of Radix-2, 3 and 5 1-D FFT on Distributed Memory Parallel Computers

3.1 Introduction

The fast Fourier transform (FFT) [27] is an algorithm widely used today in science and engineering. Parallel FFT algorithms have intensively been studied [73, 6, 40, 4, 38].

For almost all scalar and vector computers, FFT algorithms with radix-2, 3 and 5 are proposed [68, 84, 3]. Many vendors support parallel 1-D complex and real FFT algorithms with radix-2, but few vendor support radix-2, 3 and 5 parallel 1-D complex FFT on distributed memory parallel computers.

The parallel FFT algorithm can be derived from the four-step or six-step FFT algorithms [86]. These ideas can be adopted not only for the radix-2 parallel FFT but also for the radix-2, 3 and 5 parallel FFT. We succeeded to implement a radix-2, 3 and 5 parallel 1-D complex FFT algorithm on the HITACHI SR2201 and the IBM SP2, and we report their performance in this chapter.

According to theoretical analysis, we can show that the suitability of the parallel FFT algorithm differs between machines because of the variety of the CPU architecture for the processor elements of distributed memory parallel computers.

3.2 The Four-Step and Six-Step FFT Algorithms

3.2.1 The Four-Step FFT

The discrete Fourier transform (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 the indices j and k can be expressed as:

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

We can define x and y as two-dimensional arrays (in FORTRAN notation):

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

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

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

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{3.5}$$

This derivation leads to the following four-step FFT algorithm [86, 10]:

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

Step 2: $x_2(j_1, k_2) = x_1(j_1, k_2) \omega_{n_1 n_2}^{j_1 k_2}$.

Step 3: $x_3(k_2, j_1) = x_2(j_1, k_2)$.

Step 4: $y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} x_3(k_2, j_1) \omega_{n_1}^{j_1 k_1}$.

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

- 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).
- Two multirow FFTs are performed in steps 1 and 4. In this case the locality of the memory references is low, resulting in many cache misses. The four-step FFT is therefore not suitable for the RISC processors which depend on high cache hit rates to obtain high performance.

3.2.2 The Six-Step FFT

There is an algorithm known as the six-step FFT algorithm which is an extension of the four-step FFT algorithm [86, 10] in the following sense:

$$
\begin{aligned}
&\textit{Step 1:} \quad x_1(j_2, j_1) = x(j_1, j_2).\\
&\textit{Step 2:} \quad x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1)\omega_{n_2}^{j_2 k_2}.\\
&\textit{Step 3:} \quad x_3(k_2, j_1) = x_2(k_2, j_1)\omega_{n_1 n_2}^{j_1 k_2}.\\
&\textit{Step 4:} \quad x_4(j_1, k_2) = x_3(k_2, j_1).\\
&\textit{Step 5:} \quad x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1 k_1}.\\
&\textit{Step 6:} \quad y(k_2, k_1) = x_5(k_1, k_2).
\end{aligned}
$$

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

- Two multicolumn FFTs are performed in steps 2 and 5. The locality of the memory reference in the multicolumn FFT is high. Therefore, the six-step FFT is suitable for RISC processors because of the high performance which can be obtained with high hit rates in the cache memory.
- The matrix transposition takes place three times.

3.2.3 An Extended Three-Dimensional Four-Step FFT

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 ($n = n_1 n_2 n_3$), then the indices j and k can be expressed as:

$$
\begin{aligned}
j &= j_1 + j_2 n_1 + j_3 n_1 n_2,\\
k &= k_3 + k_2 n_3 + k_1 n_2 n_3.
\end{aligned}
\tag{3.6}
$$

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

$$
\begin{aligned}
x_j = x(j_1, j_2, j_3), \quad & 0 \le j_1 \le n_1 - 1,\\
0 \le j_2 \le n_2 - 1, \quad & 0 \le j_3 \le n_3 - 1,
\end{aligned}
\tag{3.7}
$$

$$
\begin{aligned}
y_k = y(k_3, k_2, k_1), \quad & 0 \le k_1 \le n_1 - 1,\\
0 \le k_2 \le n_2 - 1, \quad & 0 \le k_3 \le n_3 - 1.
\end{aligned}
\tag{3.8}
$$

Substituting the indices j and k in equation (3.1) by those in equation (3.6) and using the relation of $n = n_1 n_2 n_3$, we can derive the following equation:

$$
y(k_3, k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3)\omega_{n_3}^{j_3 k_3}\omega_{n_2 n_3}^{j_2 k_3}\omega_{n_2}^{j_2 k_2}\omega_{n}^{j_1 k_3}\omega_{n_1 n_2}^{j_1 k_2}\omega_{n_1}^{j_1 k_1}. \tag{3.9}
$$

This derivation leads to the following extended three-dimensional four-step FFT:

$$
\begin{aligned}
&\textit{Step 1:} \quad x_1(j_1, j_2, k_3) = \sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3)\omega_{n_3}^{j_3 k_3}. \\
&\textit{Step 2:} \quad x_2(j_1, j_2, k_3) = x_1(j_1, j_2, k_3)\omega_{n_2 n_3}^{j_2 k_3}. \\
&\textit{Step 3:} \quad x_3(k_3, j_1, j_2) = x_2(j_1, j_2, k_3). \\
&\textit{Step 4:} \quad x_4(k_3, j_1, k_2) = \sum_{j_2=0}^{n_2-1} x_3(k_3, j_1, j_2)\omega_{n_2}^{j_2 k_2}. \\
&\textit{Step 5:} \quad x_5(k_3, j_1, k_2) = x_4(k_3, j_1, k_2)\omega_n^{j_1 k_3}\omega_{n_1 n_2}^{j_1 k_2}. \\
&\textit{Step 6:} \quad x_6(k_3, k_2, j_1) = x_5(k_3, j_1, k_2). \\
&\textit{Step 7:} \quad y(k_3, k_2, k_1) = \sum_{j_3=0}^{n_3-1} x_6(k_3, k_2, k_1)\omega_{n_1}^{j_1 k_1}.
\end{aligned}
$$

The distinctive features of the extended three-dimensional four-step FFT can be summarized as:

- 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 $n^{2/3}$. So, the three-dimensional four-step FFT algorithm is more suitable for vector processors than the "original" four-step FFT algorithm.
- The matrix transposition takes place twice.
- Three multirow-like FFTs are performed in each step, the locality of the memory references by multirow-like FFT is again low. So, the three-dimensional four-step FFT algorithm is not suitable for RISC processors as they depend on a high cache utilization to obtain high performance.

3.3 Parallel FFT Algorithm

3.3.1 Algorithm (1)

The first parallel FFT algorithm we implemented is based on the six-step FFT algorithm. We will call it algorithm (1) hereafter.

Let N has 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 FORTRAN notation). On a distributed memory parallel computer which has P processors, the array $x(N_1, N_2)$ is distributed along the first dimension N_1. If N_1 is divisible by P, each processor has distributed data of size N/P. We introduce the notation $\hat{N}_r \equiv N_r/P$ and we denote the corresponding index as $\hat{J}_r$ which is indicating that the data along J_r is distributed across all P processors. Here, we use the subscript r to indicate that this index belongs to dimension r. The distributed array is represented as

$\hat{x}(\hat{N}_1, N_2)$. At processor m, the local index $\hat{J}_r(m)$ corresponds to the global index as the *cyclic* distribution:

$$J_r = \hat{J}_r(m) \times P + m, \quad 0 \le m \le P-1, \quad 1 \le r \le 2. \tag{3.10}$$

To illustrate the all-to-all communication it is convenient to decompose N_i into two dimensions $\hat{N}_i$ and P_i. Although P_i is same as P, we are using the subscript i to indicate that this index belongs to dimension i.

Starting with the initial data $\hat{x}(\hat{N}_1, N_2)$, the parallel FFT can be performed according to the following steps:

Step 1: Transpose

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

Step 2: Multicolumn FFTs

$$\hat{x}_2(K_2, \hat{J}_1) = \sum_{J_2=0}^{N_2-1} \hat{x}_1(J_2, \hat{J}_1)\omega_{N_2}^{J_2 K_2}.$$

Step 3: Twiddle factor multiplication and transpose

$$\hat{x}_3(\hat{J}_1, P_2, \hat{K}_2) \equiv \hat{x}_3(\hat{J}_1, K_2) = \hat{x}_2(K_2, \hat{J}_1)\omega_{N_1 N_2}^{\hat{J}_1 K_2}.$$

Step 4: Rearrangement

$$\hat{x}_4(\hat{J}_1, \hat{K}_2, P_2) = \hat{x}_3(\hat{J}_1, P_2, \hat{K}_2).$$

Step 5: All-to-all communication

$$\hat{x}_5(\hat{J}_1, \hat{K}_2, P_1) = \hat{x}_4(\hat{J}_1, \hat{K}_2, P_2).$$

Step 6: Transpose

$$\hat{x}_6(J_1, \hat{K}_2) \equiv \hat{x}_6(P_1, \hat{J}_1, \hat{K}_2) = \hat{x}_5(\hat{J}_1, \hat{K}_2, P_1).$$

Step 7: Multicolumn FFTs

$$\hat{x}_7(K_1, \hat{K}_2) = \sum_{J_1=0}^{N_1-1} \hat{x}_6(J_1, \hat{K}_2)\omega_{N_1}^{J_1 K_1}.$$

Step 8: Transpose

$$\hat{y}(\hat{K}_2, K_1) = \hat{x}_7(K_1, \hat{K}_2).$$

In steps 2 and 7, multicolumn FFTs are performed along the local dimensions. Computation in step 3 is accompanied with a transposition and twiddle factor multiplication. Step 4 is a local transposition for data rearrangement.

We note that we combined some of the operations with data movements as in step 3 to gain efficiency in utilizing the memory bandwidth.

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

- Independent $\sqrt{N}$ point FFT is repeated $\sqrt{N}/P$ times in steps 2 and 7 for the case of $N_1 = N_2 = \sqrt{N}$.

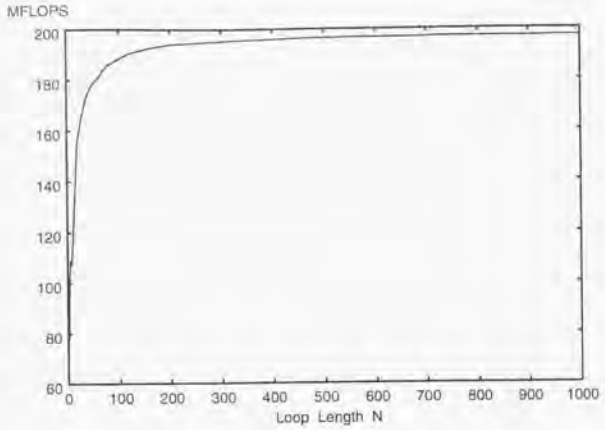


Figure 3.1: Performance of FFT kernel (radix-4) (HITACHI SR2201 1PE).

- The all-to-all communication occurs just once. Moreover, the input data x and the output data y are both *natural order*.

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

For $N = 2^{20}$ point FFT, the working set size is in the order of $\sqrt{N}(= 2^{10})$ and working set fits entirely into the cache. Thus, the multicolumn FFTs can be performed at high speed on cache-based RISC processors like the Power 2 processor as employed in the IBM SP2.

We now discuss the case of a (pseudo) vector processor processing element, e.g. HITACHI SR2201.

When an n point FFT is performed on a vector processor, the innermost loop length is $1, 2, \cdots, N/2$ or $N/2, N/4, \cdots, 1$. By interchanging the loop index, the average loop length can be in the order of $\sqrt{N}$.

Even if the innermost loop is interchanged for speed, the average loop length in the parallel algorithm is in the order of $N^{1/4}$ for an N point FFT because each processor performs an $N_1(= N_2 = \sqrt{N})$ point FFT repeatedly in this algorithm. So, even for a large N of 2^{32} the average loop length is 256 $(= 2^{32/4} = 2^8)$ which is too short and inefficient for vector processing.

Even though pipeline startup time is very short for the processing element of the HITACHI SR2201 as shown in Figure 3.1 because of the pseudo-vector processing [57] feature compared to other vector processors, the minimum loop length to obtain peak performance is more than 200. So, the algorithm (1) is not suitable for the vector parallel architecture processors.

3.3.2 Algorithm (2)

Let us consider how to perform long-vector FFTs on the processing elements of vector-parallel processors.

We can adopt the idea of an extended three-dimensional four-step FFT as described in Section 2.

Let N has factors N_1, N_2 and N_3 ($N = N_1 \times N_2 \times N_3$). Starting with the initial data $\hat{x}(\hat{N}_1, N_2, N_3)$, the FFT can be performed according to the following steps:

Step 1: Multirow-like FFTs

$$\hat{x_1}(\hat{J}_1, J_2, K_3) = \sum_{J_3=0}^{N_3-1} \hat{x}(\hat{J}_1, J_2, J_3)\omega_{N_3}^{J_3K_3}.$$

Step 2: Twiddle factor multiplication and transpose

$$\hat{x_2}(K_3, \hat{J}_1, J_2) = \hat{x_1}(\hat{J}_1, J_2, K_3)\omega_{N_2N_3}^{J_2K_3}.$$

Step 3: Multirow-like FFTs

$$\hat{x_3}(K_3, \hat{J}_1, K_2) = \sum_{J_2=0}^{N_2-1} \hat{x_2}(K_3, \hat{J}_1, J_2)\omega_{N_2}^{J_2K_2}.$$

Step 4: Twiddle factor multiplication and rearrangement

$$\begin{aligned}\hat{x_4}(P_3, \hat{K}_3, K_2, \hat{J}_1) &\equiv \hat{x_4}(K_3, K_2, \hat{J}_1) \\ &= \hat{x_3}(K_3, \hat{J}_1, K_2)\omega_N^{\hat{J}_1(K_3+K_2N_3)}.\end{aligned}$$

Step 5: Transpose

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

Step 6: All-to-all communication

$$\hat{x_6}(\hat{K}_3, K_2, \hat{J}_1, P_1) = \hat{x_5}(\hat{K}_3, K_2, \hat{J}_1, P_3).$$

Step 7: Rearrangement

$$\begin{aligned}\hat{x_7}(\hat{K}_3, K_2, J_1) &\equiv \hat{x_7}(\hat{K}_3, K_2, P_1, \hat{J}_1) \\ &= \hat{x_6}(\hat{K}_3, K_2, \hat{J}_1, P_1).\end{aligned}$$

Step 8: Multirow-like FFTs

$$\hat{y}(\hat{K}_3, K_2, K_1) = \sum_{J_1=0}^{N_1-1} \hat{x_7}(\hat{K}_3, K_2, J_1)\omega_{N_1}^{J_1K_1}.$$

The distinctive features of this second algorithm, which we call algorithm (2) from now on, can be summarized as:

- $N^{2/3}/P$ simultaneous $N^{1/3}$ point multirow-like FFTs are performed in steps 1, 3 and 8 for the case of $N_1 = N_2 = N_3 = N^{1/3}$.
- Only one all-to-all communication is required. Moreover, the input data x and the output data y are both in *natural order*.

Table 3.1: Number of real operations for small-n transforms [83].

n	Rader		Winograd	
	Adds	Mults	Adds	Mults
2	4	0	4	0
3	12	4	12	4
4	16	0	16	0
5	32	12	34	10

3.3.3 Adaptability of Parallel FFT Algorithms to Processor Architecture

In this section we want to analyze the adaptability of algorithm (1) and algorithm (2) to the type of processing element in parallel computers, e.g., processing elements of the vector processor type or of the cache-based scalar RISC processor type. The average inner loop length is particularly important. For the ease of analysis, we assume N_1, N_2 and N_3 are equal ($N_1 = N_2 = N_3 = N^{1/3}$) in algorithm (2). The average loop length in the FFTs are $N^{2/3}/P$ in the algorithm (2), and $N^{1/4}$ in the algorithm (1). P is about 2^{10} at most and N is in the order of 2^{24} or more. The expression $N^{5/12} > P$ follows from the inequality $N^{2/3}/P > N^{1/4}$. This relation means that algorithm (2) is suitable for vector-parallel architectures with the values given above for P and N.

Next, we focus on the working set size of the processing element of the cache-based RISC processor type. The working set size for the floating point operations in algorithms (1) and (2) is to be analyzed.

In algorithm (2), the working set size is N/P because $N^{2/3}/P$ simultaneous $N^{1/3}$-point multirow-like FFTs are performed. The working set size is $\sqrt{N}$ in algorithm (1), because $\sqrt{N}/P$ $\sqrt{N}$ point FFTs are performed independently in algorithm (1). $P \leq 2^{10}$ and $N \geq 2^{24}$. Therefore the expression $\sqrt{N} > P$ derived from the comparison of $N/P > \sqrt{N}$ holds.

Under these conditions, algorithm (2) is suitable for parallel computers with cache-based RISC processor processing elements.

3.4 Radix-2, 3, 4 and 5 FFT Algorithm on a Single Processor

As for a single processor algorithm we used radix-2, 3, 4 and 5 FFT algorithm based on the mixed-radix FFT algorithms of Temperton [84]. The Stockham FFT algorithm [72] was used for radix-2 FFT transforms. We modified the Stockham algorithm by including Rader's "small-n" transform [59] for radix-3 and radix-5.

The "small-n" transform, based on the WFTA (Winograd Fourier transform algorithm) [88] by S. Winograd, has two more additions as compared to Rader's radix-5 algorithm. By contrast, Rader's transform uses two more multiplications (see Table 3.1).

Therefore, Rader's "small-n" transform is more efficient when the CPU time for multiplication operation is equal to that of addition operation and the multiplication operation and addition operation can be performed simultaneously on the processing element as is the case on the HITACHI SR2201 and IBM SP2.

When performing a 2^p point FFT, a radix-4, or radix-8 FFT is faster than a radix-2 FFT [15] because of less memory access and a reduced number of floating point operations. In the same way, a radix-6 ($= 2 \times 3$) FFT and a radix-9 ($= 3 \times 3$) FFT, can be applied to $2^p 3^q$ point FFTs. These higher radix FFTs reduce the number of multiplies and the total floating point operation count in the algorithm. However, higher radix FFTs require more registers to hold intermediate results. Present day most CPUs has insufficient registers for high radix operation. For this reason, we only implemented the radix-2, 3, 4 and 5 FFT algorithms for the evaluation.

3.4.1 The Radix-2 FFT

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

$l = n/2;\ m = 1$
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

Here the variables c_0 and c_1 are temporary variables.

3.4.2 The Radix-3 FFT

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.

3.4.3 The Radix-4 FFT

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.

3.4.4 The Radix-5 FFT

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.

3.4.5 Arithmetic Operation Counts

Analysis of the operation count for the mixed-radix Cooley-Tukey FFT algorithm is explained in reference [84]. Here we adapt the formula given there to the case of $N = 2^p3^q5^r$.

The number of real additions $A(N)$ and multiplications $M(N)$ are given by:

$$A(N) = 2N\left(\frac{3}{2}p + \frac{8}{3}q + 4r - 1\right) + 2,$$
$$M(N) = 2N\left(p + 2q + \frac{14}{5}r - 2\right) + 4.$$

So, the total operation count is:

$$A(N) + M(N) = 2N\left(\frac{5}{2}p + \frac{14}{3}q + \frac{34}{5}r - 3\right) + 6. \tag{3.11}$$

3.5 Experimental Results of the Parallel FFT

To evaluate our radix-2, 3 and 5 parallel 1-D complex FFT, p, q, r of $N = 2^p3^q5^r$ and the number of processors P were varied. We averaged the elapsed times obtained from 10 executions of complex forward FFTs. The parallel FFTs were performed on double precision complex data and the table for twiddle factors was prepared in advance.

A HITACHI SR2201 (1024 PEs, 256 MB per PE, 300 MFLOPS per PE, 256 GB total main memory size, communication bandwidth 300 MB/sec both way per link, and 307.2 GFLOPS peak performance) and an IBM SP2 thin-node system (32 PEs, 256 MB per PE, 266 MFLOPS per PE, 8 GB total main memory size, communication bandwidth 40 MB/sec per link, and 8.5 GFLOPS peak performance) were used as distributed memory parallel computers in the experiment.

3.5.1 Experimental Results on the HITACHI SR2201

Remote Direct Memory Access (RDMA) message transfer protocol [17] without memory copy was used as a communication library on the HITACHI SR2201. All routines were written in

Table 3.2: Performance of parallel FFT algorithm (1) on the HITACHI SR2201
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

P	$N = 2^{20} \cdot 3 \cdot 5$		$N = 2^{21} \cdot 3^2$		$N = 2^{25} \cdot 3$		$N = 2^{22} \cdot 5^2$		$N = 2^{30}$	
	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS
8	3.6857	0.50	*	*	*	*	*	*	*	*
16	1.6233	1.13	2.1084	1.05	*	*	*	*	*	*
32	0.8178	2.25	1.0615	2.09	*	*	*	*	*	*
64	0.4165	4.42	0.5401	4.11	3.0333	4.26	3.3616	4.09	*	*
128	0.2218	8.29	0.3012	7.37	1.5341	8.42	1.6971	8.11	*	*
256	0.1295	14.20	0.2347	9.46	0.8433	15.32	0.8744	15.73	*	*
512	0.1013	18.16	0.1232	18.03	0.6775	19.07	0.5038	27.31	4.6271	33.42
1024	0.0525	35.06	0.0630	35.28	0.3406	37.93	0.3741	36.77	2.3158	66.77

Table 3.3: Performance of parallel FFT algorithm (2) on the HITACHI SR2201
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

P	$N = 2^{20} \cdot 3 \cdot 5$		$N = 2^{21} \cdot 3^2$		$N = 2^{25} \cdot 3$		$N = 2^{22} \cdot 5^2$		$N = 2^{30}$	
	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS
8	3.1892	0.58	*	*	*	*	*	*	*	*
16	0.9153	2.01	1.0794	2.06	*	*	*	*	*	*
32	0.4788	3.84	0.5420	4.10	*	*	*	*	*	*
64	0.2466	7.46	0.2792	7.96	1.5621	8.27	1.5624	8.81	*	*
128	0.1298	14.17	0.1638	13.56	0.7975	16.20	0.8952	15.37	*	*
256	0.0720	25.55	0.0860	25.81	0.4274	30.22	0.4160	33.07	*	*
512	0.0406	45.27	0.0517	42.98	0.2541	50.85	0.2779	49.50	2.3436	65.98
1024	0.0379	48.53	0.0465	47.76	0.1359	95.04	0.1358	101.34	1.1912	129.80

FORTRAN. The compiler used was optimized FORTRAN77 V02-05-/B of Hitachi Ltd. The optimization option, `-W0,'opt(o(ss),split(2))'` was specified.

Tables 3.2 and 3.3 show the results of the average execution times of algorithm (1) and algorithm (2). The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average execution performance in GFLOPS. The GFLOPS value is based on equation (3.11) for a transform of size $N = 2^p 3^q 5^r$.

Algorithm (2) is better than algorithm (1) on the HITACHI SR2201 as is clear from Tables 3.2 and 3.3. The innermost loop length of the algorithm (2) is larger than that of the algorithm (1). The (pseudo) vector processor architecture of the HITACHI SR2201 processing element is able to take advantage of this fact.

We note that on the HITACHI SR2201 with 1024 PEs, about 130 GFLOPS was realized with size $N = 2^{30}$ in algorithm (2) as in Table 3.3.

Table 3.4: All-to-all communication performance on the HITACHI SR2201
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

P	$N = 2^{20} \cdot 3 \cdot 5$		$N = 2^{21} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{22} \cdot 5^2$		$N = 2^{30}$	
	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec
8	0.1321	238.08	*	*	*	*	*	*	*	*
16	0.0647	243.00	0.0773	244.32	*	*	*	*	*	*
32	0.0353	222.64	0.0415	227.42	*	*	*	*	*	*
64	0.0235	167.51	0.0266	177.40	0.1078	233.37	0.1120	234.05	*	*
128	0.0242	81.24	0.0258	91.40	0.0677	185.82	0.0699	187.61	*	*
256	0.0182	53.89	0.0217	54.37	0.0584	107.67	0.0596	109.98	*	*
512	0.0122	40.39	0.0144	40.97	0.0723	43.52	0.0753	43.54	0.2278	147.27
1024	0.0081	30.35	0.0094	31.31	0.0437	35.97	0.0455	36.02	0.2439	68.80

Table 3.5: Performance of parallel FFT algorithm (1) on the IBM SP2
(* means that we were not able to execute because the maximum available memory size of 256 MB per PE was insufficient).

P	$N = 2^{17} \cdot 3 \cdot 5$		$N = 2^{18} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{19} \cdot 5^2$		$N = 2^{25}$	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
4	2.1675	92.46	2.8684	84.44	*	*	*	*	*	*
8	0.7222	277.50	0.9169	264.17	9.0235	158.04	9.2553	164.56	*	*
16	0.3133	639.67	0.3794	638.43	3.9170	364.07	3.5936	423.82	12.9452	308.45
32	0.1631	1228.75	0.1942	1247.28	1.7596	810.45	1.4650	1039.63	6.5209	612.34

Table 3.4 shows the results of the all-to-all communication timings on the HITACHI SR2201. The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average bandwidth in MB/sec.

3.5.2 Experimental Results on the IBM SP2

MPI [54] was used as a communication library on IBM SP2. All routines were written in FORTRAN as on the HITACHI SR2201. The compiler used was IBM XL Fortran version 3.2. As a optimization option, `-O3 -qarch=pwr2 -qhot -qtune=pwr2` was specified. Tables 3.5 and 3.6 show the result of the average execution times of algorithm (1) and algorithm (2).

The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average execution performance in MFLOPS.

We can see that algorithm (1) is better than algorithm (2) on the IBM SP2. This is because the working set size of algorithm (1) is smaller than that of algorithm (2). Thus, the algorithm (1) is suitable for the parallel computers with cache-based scalar RISC processors as processing

Table 3.6: Performance of parallel FFT algorithm (2) on the IBM SP2
(* means that we were not able to execute because the maximum available memory size of 256 MB per PE was insufficient).

P	$N = 2^{17} \cdot 3 \cdot 5$		$N = 2^{18} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{19} \cdot 5^2$		$N = 2^{25}$	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
4	3.6755	54.53	4.7684	50.80	*	*	*	*	*	*
8	1.7492	114.57	2.2489	107.71	19.0635	74.81	15.8133	96.31	*	*
16	0.7383	271.45	1.0034	241.40	8.4410	168.94	7.4366	204.81	30.4703	131.05
32	0.2431	824.39	0.3972	609.82	3.7436	380.93	3.3936	448.80	15.0385	265.52

Table 3.7: All-to-all communication performance on the IBM SP2
(* means that we were not able to execute because the maximum available memory size of 256 MB per PE was insufficient).

P	$N = 2^{17} \cdot 3 \cdot 5$		$N = 2^{18} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{19} \cdot 5^2$		$N = 2^{25}$	
	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec
4	0.2655	29.62	0.3184	29.64	*	*	*	*	*	*
8	0.1532	25.66	0.1819	25.94	0.9645	26.09	0.9963	26.31	*	*
16	0.0923	21.29	0.1094	21.57	0.5680	22.15	0.5876	22.31	1.4952	22.44
32	0.0611	16.08	0.0742	15.90	0.3616	17.40	0.3770	17.38	0.9589	17.50

elements.

We note that on the IBM SP2 with 32 PEs, about 1.25 GFLOPS was realized with size $N = 2^{18} \cdot 3^2$ in algorithm (1) as shown in Table 3.5.

Table 3.7 shows the results of the all-to-all communication timings on the IBM SP2. The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average bandwidth in MB/sec.

Chapter 4

Fast Multiple-Precision Addition, Subtraction and Multiplication on Distributed Memory Parallel Computers

4.1 Introduction

In this chapter, we present efficient parallel algorithms for the multiple-precision addition, subtraction and multiplication of more than several million decimal digits on distributed memory parallel computers.

Several software packages are available for the multiple-precision computation [21, 24, 69, 11]. Brent's MP multiple-precision package [21] is probably the most widely used of these packages at present, due to its greater functionality and efficiency. D. M. Smith made a similar package that features improved performance for certain transcendental functions [69].

Another available package at some sites is MPFUN made by D. H. Bailey [11]. One of the key features in the MPFUN package is that package is optimized for vector supercomputers and RISC processors.

To perform the multiple-precision calculation at high speed, vector processing oriented implementations have been proposed [41, 24, 11]. The processing speed and main memory size of the vector computers are becoming saturated. Therefore a parallel processing by a distributed memory parallel computer is one of the solutions for the fast multiple-precision calculation.

As for the related works, parallel implementations of the multiple-precision arithmetic on a shared memory machine have been reported by K. Weber [87]. Weber modified the MPFUN package [11] to run in parallel on a shared memory multiprocessor. However, he did not present

a complete parallel solution in the normalization of the result, e.g. carry/borrow propagation.

Parallel implementation of Karatsuba's multiplication algorithm [43, 45] was proposed by G. Cesari and R. Maeder [25] on a distributed memory parallel computer. Karatsuba's algorithm is known as $O(n^{\log_2 3})$ multiple-precision multiplication algorithm. However, the multiple-precision multiplication of n-digit numbers can be performed in $O(n \log n \log\log n)$ operations by using the Schönhage-Strassen algorithm [66, 5, 45] which is the algorithm based on the fast Fourier transform (FFT) [27].

In the multiple-precision multiplication of several thousand decimal digits or more, the FFT-based multiplication is the fastest. FFT-based multiplication algorithms are known to be good candidates for parallel implementation.

B. S. Fagin [33, 35] used the Fermat number transform (FNT) [60, 1, 2, 58] for large integer multiplication on the Connection Machine CM-2. FNT uses many modulo operations which are slow because of integer division. Thus, we use the real FFT-based multiplication for the multiple-precision multiplication on distributed memory parallel computers.

In the multiple-precision parallel addition, subtraction and multiplication by single-precision integer, parallelization of releasing propagated carries and borrows is the key component in the processing speed. Similarly to the multiple-precision addition and subtraction, a part of normalization of results in the multiple-precision multiplication can be parallelized.

For simplicity, this chapter discusses the calculation of the multiple-precision fixed point numbers. However, it is not hard to extend the proposed algorithm to the calculation of the multiple-precision floating point numbers [74].

4.2 Parallelization of the Multiple-Precision Addition, Subtraction and Multiplication by Single-Precision Integer

The arithmetic operation counts for n-digit multiple-precision sequential addition, subtraction and multiplication by single-precision integer is clearly $O(n)$. However, a major factor to obstruct parallelization is releasing the carries and borrows in these operations.

For example, a FORTRAN 77 program of the multiple-precision sequential addition is shown in Figure 4.1. Here, `ICARRY` is a variable to store carry and `ITEMP` is a temporary variable. We assume that the input data is normalized as `0` $\sim$ `IRADIX-1` and stored in arrays `IA` and `IB`.

In this program, the value of `ICARRY` recurrently decides the value of `ITEMP` at line 5. Then, the program cannot be parallelized because of data dependency.

An algorithm shown in Figure 4.2 enables us to parallelize releasing the carries and borrows. We assume that the input data is normalized to `0` $\sim$ `IRADIX-1`, and input data is stored in arrays `IA` and `IB`.

We perform the multiple-precision addition without propagation of carries at line 3. Results

```
1     SUBROUTINE SEQADD(IA,IB,IRADIX,N)
2     INTEGER IA(N),IB(N)
3     ICARRY=0
4     DO I=N,2,-1
5       ITEMP=IA(I)+IB(I)+ICARRY
6       ICARRY=ITEMP/IRADIX
7       IA(I)=ITEMP-ICARRY*IRADIX
8     END DO
9     IA(1)=IA(1)+IB(1)+ICARRY
10    RETURN
11    END
```

Figure 4.1: Multiple-precision sequential addition.

are checked with (IA(2:N) $\geq$ IRADIX) at line 4. If the value of the each element of the array IA(2:N) is greater than or equal to IRADIX, we have to compute the carries in line 8 and release the carries in lines 7 and 8. Here, IC is a working array to store carries.

At the time of releasing carries, we do not care about the propagation of the carries. Since carries are not corrected completely, DO WHILE loop is repeated until each element in the array IA(2:N) is less than IRADIX.

We are assuming that the value of each element of the array IA has been normalized as 0 $\sim$ IRADIX-1. In the case of radix $= 10^8$, a probability of having two consecutive carries is $0.5 \times (1/10^8)^2 = 5 \times 10^{-17}$. Thus, this algorithm performs the propagation operations successfully. When the propagation of carries repeats like in the case of $0.99999999\cdots 9 + 0.00000000\cdots 1$, we have to use the carry skip method [52].

A Fortran 90 program of the multiple-precision parallel normalization with the carry skip method is shown in Figure 4.3. We are also assuming that the input data is normalized to 0 $\sim$ IRADIX-1 and stored in array IX. We perform *incomplete* normalization as 0 $\sim$ IRADIX in lines 3 $\sim$ 8. The range for carry skipping is decided from line 10 to line 17. Note that DO WHILE loop in line 5 is repeated twice at most. Finally, we perform carry skipping from line 18 to line 20. Array IC is a working array to store carries.

The same methods can be applied to the multiple-precision parallel subtraction and multiplication by single-precision integer.

```
1       SUBROUTINE PARAADD(IA,IB,IC,IRADIX,N)
2       INTEGER,DIMENSION(N) :: IA,IB,IC
3       IA(1:N)=IA(1:N)+IB(1:N)
4       DO WHILE (ANY(IA(2:N) .GE. IRADIX))
5         IC(N)=0
6         IC(1:N-1)=IA(2:N)/IRADIX
7         IA(2:N)=IA(2:N)+IC(2:N)-IC(1:N-1)*IRADIX
8         IA(1)=IA(1)+IC(1)
9       END DO
10      RETURN
11      END
```

Figure 4.2: Multiple-precision parallel addition.

4.3 Parallelization of the Multiple-Precision Multiplication

4.3.1 Multiple-Precision Multiplication Algorithm

A key operation in the fast multiple-precision arithmetic is the multiplication, by which significant time in the total computation is spent. Many multiple-precision multiplication algorithms have been proposed [45]. In this section, we discuss the multiple-precision multiplication based on the *floating point real* FFT [82, 41].

The following is the multiple-precision multiplication algorithm [82, 41]. Here, let us consider the multiplication of two $m \times 2^n$ bit ($= m \times (\log_{10} 2) \times 2^n$ decimal digit) integers A and B.

Step 1: Prepare two double-precision floating point arrays with 2×2^n entries.

Step 2: Convert both of $m \times 2^n$ bit integers into double-precision floating point numbers. (The first half of 2×2^n entries contain information for $m \times 2^n$ bits, namely, m bit information per double-precision floating point array entry.)

Step 3: Initialize the second half of 2×2^n entries to double-precision floating point zero.

Step 4: Apply 2^{n+1} point forward FFT operations to A and B, giving A' and B', respectively.

Step 5: Perform the convolution product operations between A' and B', giving a new 2×2^n entry double-precision array C'.

Step 6: Apply 2^{n+1} point inverse FFT operations to C', giving C. (Now, C is the double-precision floating point array of 2×2^n entries. If operations of forward FFT, inverse FFT and convolution product are performed in infinite precision.

```
1     SUBROUTINE SKIPNORM(IX,IC,IRADIX,N)
2     INTEGER,DIMENSION(N) :: IX
3     DO WHILE (ANY(IX(2:N) .GT. IRADIX))
4       IC(N)=0
5       IC(1:N-1)=IX(2:N)/IRADIX
6       IX(2:N)=IX(2:N)+IC(2:N)-IC(1:N-1)*IRADIX
7       IX(1)=IX(1)+IC(1)
8     END DO
9     DO WHILE (ANY(IX(2:N) .EQ. IRADIX))
10      IE=1
11      DO I=2,N
12        IF (IX(I) .EQ. IRADIX) IE=I
13      END DO
14      IS=1
15      DO I=2,IE-1
16        IF (IX(I) .LT. IRADIX-1) IS=I
17      END DO
18      IX(IS)=IX(IS)+1
19      IX(IS+1:IE-1)=IX(IS+1:IE-1)-(IRADIX-1)
20      IX(IE)=IX(IE)-IRADIX
21    END DO
22    RETURN
23    END
```

Figure 4.3: Parallel normalization with the carry skip method.

each entry of C should be the exact double-precision floating point representation for an integer with maximum value of $2^n \times (2^m - 1)^2$. However, these representations slightly deviate from exact values in the actual operation, because infinite precision operations are impossible.)

Step 7: Convert each entry of C into integer representation, denoted by `X`. (Conversion should be done with `DNINT` operation in FORTRAN. If the absolute value of `(X-DNINT(X))` is near `0.5D0`, the multiplication is considered to be incorrect.)

Step 8: Normalize C under the suitable radix. The radix of 2^m or $10^{m(\log_{10} 2)}$ is better for binary or decimal representation. Final result is the result of multiplication between A and B.

For the *floating point real* FFT-based multiplication, we can use the "balanced representation" [29] which tend to yield reduced errors for the convolutions we intend to perform.

4.3.2 Parallelization of the Multiple-Precision Multiplication

In a parallelization of the multiple-precision multiplication, steps $1 \sim 3$, step 5 and step 7 of the multiplication algorithm given in subsection 4.3.1 can be parallelized with ease.

Since many parallel FFT algorithms are proposed [86, 38, 79], we can use an efficient parallel FFT algorithm for steps 4 and 6.

The normalization of step 8 is essentially the same as the parallel processing of carry in the multiple-precision addition, subtraction and multiplication by single-precision integer. Thus, this normalization can also be parallelized with ease.

4.4 Experimental Results

To evaluate our parallel multiple-precision arithmetic algorithm, decimal digits n and the number of processors P were varied. We averaged the elapsed times obtained from 10 executions of the multiple-precision parallel addition $(\pi + \sqrt{2})$, multiple-precision parallel multiplication $(\pi \times \sqrt{2})$. We note that the value of n-digit π and $\sqrt{2}$ were prepared in advance. The choice of these values has no particular significance here, but was convenient as definite test cases for which the results were able to be checked for randomized test data.

A HITACHI SR2201 was used as the distributed memory parallel computer. In the experiment, we used 4 PEs $\sim$ 256 PEs on the HITACHI SR2201.

MPI [54] on the HITACHI SR2201 was used as a communication library. All routines were written in FORTRAN. The compiler used was optimized FORTRAN77 V02-06-/A of Hitachi Ltd. The optimization option, `-WO,'opt(o(ss),approx(0))'` was specified.

The radix of the multiple-precision number is 10^8. The multiple-precision number is stored with cyclic distribution in the array of 32-bit integers. Each input data word is split into two words upon entry to the FFT-based multiplication.

Table 4.1 shows the result of the averaged execution times of multiple-precision parallel addition $(\pi + \sqrt{2})$. The column headed by P shows the number of processors. The next six columns contain the average execution times in seconds. In Figure 4.4 we compare the average execution times of multiple-precision parallel addition. For small digits of $N = 2^{20}$ and $P > 64$, we can clearly see that communication overhead dominates the execution time.

Table 4.2 shows the result of the average execution times of the multiple-precision parallel multiplication $(\pi \times \sqrt{2})$. The column headed by P shows the number of processors. The next six columns contain the average execution times in seconds. Figure 4.5 is the comparison of the average execution times of multiple-precision parallel multiplication. We can see that our

multiple-precision parallel multiplication is scalable on the HITACHI SR2201 as is clear from Figure 4.5.

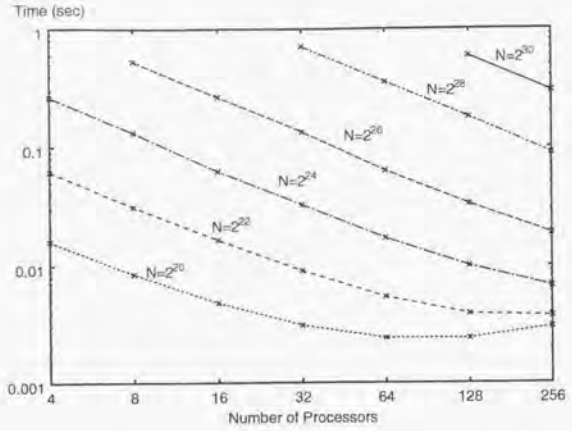


Figure 4.4: Execution time of multiple-precision parallel addition $(\pi + \sqrt{2})$, N = number of decimal digits.

Table 4.1: Execution time of multiple-precision parallel addition $(\pi + \sqrt{2})$ (in seconds), N = number of decimal digits.

(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	0.0159	0.0620	0.2638	*	*	*
8	0.0085	0.0316	0.1329	0.5316	*	*
16	0.0048	0.0164	0.0628	0.2665	*	*
32	0.0031	0.0090	0.0324	0.1345	0.7092	*
64	0.0024	0.0054	0.0171	0.0638	0.3550	*
128	0.0024	0.0039	0.0100	0.0335	0.1800	0.5907
256	0.0030	0.0037	0.0067	0.0187	0.0898	0.2996

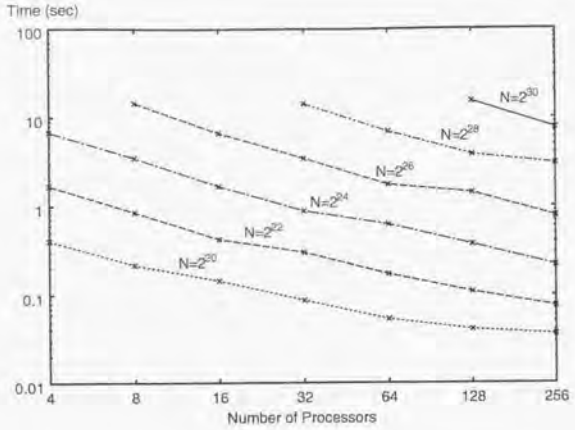


Figure 4.5: Execution time of multiple-precision parallel multiplication ($\pi \times \sqrt{2}$), N = number of decimal digits.

Table 4.2: Execution time of multiple-precision parallel multiplication ($\pi \times \sqrt{2}$) (in seconds), N = number of decimal digits.
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	0.4070	1.6910	6.7453	*	*	*
8	0.2178	0.8612	3.5138	14.4982	*	*
16	0.1451	0.4232	1.6907	6.6718	*	*
32	0.0861	0.3036	0.8833	3.4489	14.4165	*
64	0.0532	0.1730	0.6333	1.7678	6.9537	*
128	0.0406	0.1088	0.3748	1.4373	3.8784	15.2900
256	0.0358	0.0747	0.2175	0.7897	3.0539	7.6423

Chapter 5

A Multiple-Precision Division by Single-Precision Integers on Distributed Memory Parallel Computers

5.1 Introduction

Many multiple-precision division algorithms have intensively been studied [71, 55, 70, 45]. D. E. Knuth [45] described classical algorithms for n-digit multiplication and division. These methods require $O(n^2)$ operations.

It is well known that the division of two multiple-precision numbers can be performed using the Newton iteration [11, 45]. This scheme requires $O(M(n))$ operations, where $M(n)$ is the number of operations for n-digit multiplication. Multiple-precision multiplication of n-digit numbers can be performed in $M(n) = n \log n \log \log n$ operations by using the Schönhage-Strassen algorithm [66, 5, 45] which is the algorithm based on the fast Fourier transform (FFT) [27].

On the other hand, the multiple-precision division by single-precision integer is also used in the multiple-precision arithmetic, which is much faster than the division by a multiple-precision number. Although several multiple-precision arithmetic packages [21, 24, 11] include a routine of the multiple-precision division by single-precision integer, the multiple-precision parallel division by single-precision integer has not been presented so far.

Parallel implementation of the multiple-precision arithmetic on a shared memory machine have been reported by K. Weber [87]. Weber modified the MPFUN multiple-precision arithmetic package [11] to run in parallel on a shared memory multiprocessor. B. S. Fagin also implemented the multiple-precision addition [34] and multiplication [33, 35] on the Connection Machine CM-2.

However, they did not discuss the multiple-precision division.

In this chapter, a multiple-precision parallel division by single-precision integer is presented.

5.2 Algorithm

In this chapter, we discuss the multiple-precision arithmetic with radix b for the division of an n-digit integer by a 1-digit integer, giving an n-digit quotient and a 1-digit remainder. For simplicity, we assume that we are working with nonnegative integer.

Let us define an n-digit dividend $u = \sum_{i=1}^{n} u_i b^{n-i}$ and a 1-digit divisor v in radix b notation, where $0 \leq u_i < b$ and $1 \leq v < b$.

The quotient q can be expressed as follows:

$$q = \lfloor u/v \rfloor = \sum_{i=1}^{n} q_i b^{n-i}, \tag{5.1}$$

where $0 \leq q_i < b$.

The partial remainder r_i, and the overall remainder r can be expressed as follows:

$$r_i = u_i - vq_i, \qquad i = 1, 2, \cdots, n, \tag{5.2}$$

$$r = u - vq \equiv r_n, \tag{5.3}$$

where $0 \leq r_i < v$.

Then, the partial quotient q_i and the partial remainder r_i can be expressed as follows:

$$q_i = \lfloor (br_{i-1} + u_i)/v \rfloor, \qquad i = 1, 2, \cdots, n, \tag{5.4}$$

$$r_i = (br_{i-1} + u_i) \bmod v, \qquad i = 1, 2, \cdots, n. \tag{5.5}$$

Note that equation (5.5) includes the first-order recurrence.

The first-order recurrence can be evaluated sequentially from the definition of the recurrence by the following serial FORTRAN 77 code:

```
R(1)=MOD(U(1),V)
DO 10 I=2,N
  R(I)=MOD(B*R(I-1)+U(I),V)
10 CONTINUE
```

where R and U have been declared as arrays.

This requires

$$(3n - 2) \text{ arithmetic operations with parallelism 1,} \tag{5.6}$$

and

$$(n - 1) \text{ communications with parallelism 1.} \tag{5.7}$$

We note that a unit parallel communication operation is defined as a shift of all elements of an array in parallel to a set of other PEs.

We will apply the parallel cyclic reduction method [39] to the equation (5.5). Let us write the original recurrence relation for two successive terms as:

$$r_i = (br_{i-1} + u_i) \bmod v, \tag{5.8}$$

and

$$r_{i-1} = (br_{i-2} + u_{i-1}) \bmod v. \tag{5.9}$$

By substituting equation (5.9) into equation (5.8), we obtain

$$\begin{aligned} r_i &= (b^2 r_{i-2} + bu_{i-1} + u_i) \bmod v \\ &= (b^{(1)} r_{i-2} + u_i^{(1)}) \bmod v, \end{aligned} \tag{5.10}$$

where equation (5.10) is a first-order recurrence between alternate terms of the sequence with a new set of coefficients given by

$$b^{(1)} = b^2 \bmod v, \quad u_i^{(1)} = (bu_{i-1} + u_i) \bmod v. \tag{5.11}$$

The repeated application of the above process can be summarized as follows:

$$r_i = (b^{(l)} r_{i-2^l} + u_i^{(l)}) \bmod v, \quad \begin{cases} l = 0, 1, \cdots, \lceil \log_2 n \rceil \\ i = 1, 2, \cdots, n \end{cases} \tag{5.12}$$

where superscripts denote the level number.

$$b^{(l)} = (b^{(l-1)})^2 \bmod v, \tag{5.13}$$

$$u_i^{(l)} = (b^{(l-1)} u_{i-2^{l-1}}^{(l-1)} + u_i^{(l-1)}) \bmod v. \tag{5.14}$$

and initially

$$b^{(0)} = b \bmod v, \quad u_i^{(0)} = u_i \bmod v. \tag{5.15}$$

If the subscript of any r_i and u_i is outside the defined range $1 \le i \le n$, the correct result is obtained by taking its value as zero. When $l = \lceil \log_2 n \rceil$, all references to $r_{i-2^l} = r_{i-2^{\lceil \log_2 n \rceil}}$ in equation (5.12) are outside the defined range, hence the solution to the recurrence is given by

$$r_i = u_i^{(\lceil \log_2 n \rceil)} \bmod v. \tag{5.16}$$

The method is therefore to generate successively the coefficients $b^{(l)}$ and $u_i^{(l)}$ defined by equation (5.13) and (5.14) until $u_i^{\lceil \log_2 n \rceil}$ is obtained. Figure 5.1 shows the communication diagram for the evaluation of r_i.

The average parallelism is

$$[(n-1) + (n-2) + \cdots + (n-2^m) + \cdots + n/2] / \lceil \log_2 n \rceil = n[1 - (1 - n^{-1}) / \lceil \log_2 n \rceil]. \tag{5.17}$$

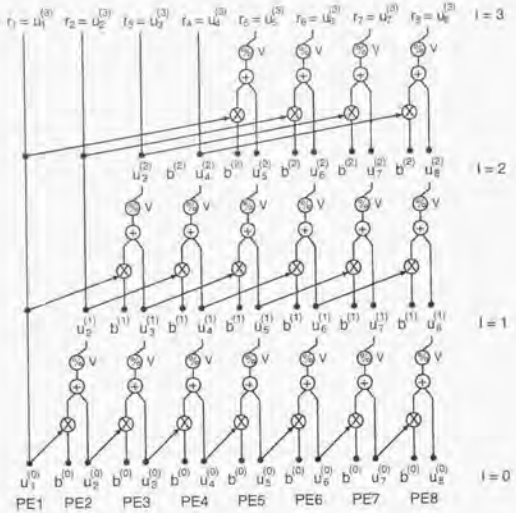


Figure 5.1: The communication diagram for equation (5.14).

hence, asymptotically for large n, we have:

$$3\lceil \log_2 n \rceil \text{ arithmetic operations with parallelism } n. \tag{5.18}$$

and

$$\lceil \log_2 n \rceil \text{ communications with parallelism } n. \tag{5.19}$$

The parallel cyclic reduction algorithm can be implemented in a parallel form of Fortran 90 as the following:

```
BMOD=MOD(B,V)
R(1:N)=MOD(U(1:N),V)
DO L=1,CEILING(LOG2(N))
  IF (BMOD .EQ. 0) EXIT
  R(1:N)=MOD(BMOD*EOSHIFT(R(1:N),-(2**(L-1)))+R(1:N),V)
  BMOD=MOD(BMOD**2,V)
END DO
```

Where R and U have been declared as arrays.

When $b^{(m)} \bmod v$ $(m = 0, 1, \cdots, \lceil \log_2 n \rceil)$ is zero, all references to $b^{(l)} \bmod v$ $(l = m+1, m+2, \cdots, \lceil \log_2 n \rceil)$ are zero in equation (5.13) and $r_i = u_i^{(m)} \bmod v$ in equation (5.12). Thus, the `DO` loop of the above program can be interrupted when $b^{(l)} \bmod v = 0$.

In particular, when b is multiple of v, $b^{(0)} \bmod v = 0$. Thus, in this case, the arithmetic operation count of this algorithm is $O(n/P)$ on parallel computers which have P processors. Also, when b is not multiple of v, an upper bound of the arithmetic operation count of this algorithm is $O((n/P)\log n)$.

Finally, we can obtain the quotient q in parallel by the following Fortran 90 program:

```
Q(1:N)=INT((B*EOSHIFT(R(1:N),-1)+U(1:N))/V)
```

where `Q`, `R` and `U` have been declared as arrays.

5.3 Experimental Results

To evaluate our multiple-precision parallel division by single-precision integer, decimal digits N and the number of processors P were varied. We averaged the elapsed times obtained from 10 executions of the multiple-precision parallel division by single-precision integer, $\pi/2$ and $\pi/3$. We note that the value of n-digit π was prepared in advance.

A HITACHI SR2201 was used as distributed memory parallel computer. In the experiment, we used $4 \sim 256$ PEs on the HITACHI SR2201. MPI [54] was used as a communication library on the HITACHI SR2201. All routines were written in FORTRAN.

The radix of the multiple-precision number is 10^8. The multiple-precision number is stored with cyclic distribution in the array of 32-bit integers.

Table 5.1 shows the result of the average execution times of multiple-precision parallel division by the single-precision integer, $\pi/2$. The column headed by P shows the number of processors. The next six columns contain the average execution times in seconds. In Figure 5.2 we compare the average execution times of $\pi/2$. For $N = 2^{20}$ and $P > 64$, we can clearly see that communication overhead dominates the execution time. We note that the arithmetic operation count is $O(N/P)$ in the division of $\pi/2$, since the radix $(= 10^8)$ is multiple of the divisor $(= 2)$.

Table 5.2 shows the result of the average execution times of the multiple-precision parallel division by a single-precision integer, $\pi/3$. The column headed by P shows the number of processors. The next six columns contain the average execution times in seconds. In Figure 5.3 we compare the average execution times of $\pi/3$. For $N = 2^{20}$ and $P > 64$, we can clearly see that communication overhead dominates the execution time. We note that the arithmetic operation count is $O((N/P)\log N)$ in the division of $\pi/3$, since the radix $(= 10^8)$ is not multiple of the divisor $(= 3)$.

The calculation of $\pi/2$ is up to about 8.44 times faster than that of $\pi/3$ when $N = 2^{30}$ and $P = 256$. This is because the computation of $\pi/2$ has less arithmetic operations $O(N/P)$ compared with the case of $\pi/3$.

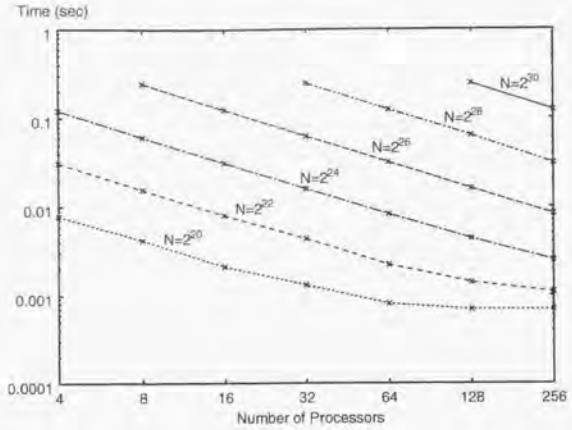


Figure 5.2: Execution time of multiple-precision parallel division by a single-precision integer $(\pi/2)$, N = number of decimal digits.

Table 5.1: Execution time of multiple-precision parallel division by a single-precision integer $(\pi/2)$ (in seconds), N = number of decimal digits.
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	0.0078	0.0311	0.1214	*	*	*
8	0.0042	0.0157	0.0613	0.2442	*	*
16	0.0021	0.0080	0.0309	0.1228	*	*
32	0.0013	0.0043	0.0158	0.0615	0.2451	*
64	0.0008	0.0022	0.0082	0.0314	0.1222	*
128	0.0007	0.0014	0.0044	0.0162	0.0639	0.2456
256	0.0007	0.0011	0.0025	0.0083	0.0313	0.1231

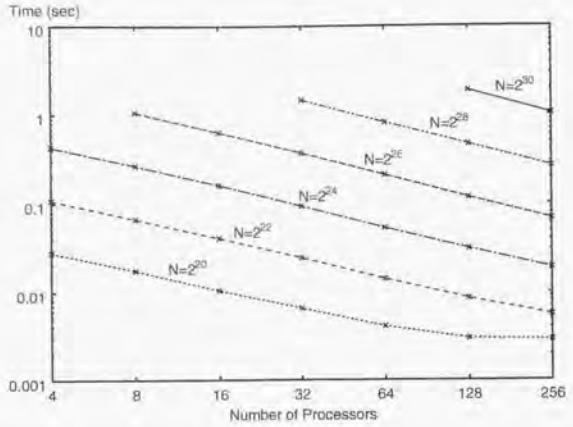


Figure 5.3: Execution time of multiple-precision parallel division by a single-precision integer ($\pi/3$), N = number of decimal digits.

Table 5.2: Execution time of multiple-precision parallel division by a single-precision integer ($\pi/3$) (in seconds). N = number of decimal digits.
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	0.0278	0.1096	0.4347	*	*	*
8	0.0176	0.0674	0.2679	1.0707	*	*
16	0.0105	0.0405	0.1599	0.6303	*	*
32	0.0065	0.0243	0.0930	0.3662	1.4537	*
64	0.0041	0.0140	0.0534	0.2111	0.8237	*
128	0.0030	0.0085	0.0310	0.1180	0.4647	1.8647
256	0.0029	0.0056	0.0190	0.0687	0.2637	1.0387

Chapter 6

Fast Multiple-Precision Calculation of Division and Square Root on Distributed Memory Parallel Computers

6.1 Introduction

In this chapter, we present efficient parallel algorithms for the multiple-precision division and square root operation of more than several million decimal digits on distributed memory parallel computers.

The multiple-precision division and square root operation take considerably longer time to compute than the addition, subtraction and multiplication. There are a number of ways to perform division and square root operation [45]. It is well known that the multiple-precision division and square root operation can be reduced to the multiple-precision addition, subtraction and multiplication by using the Newton iteration [45]. This scheme requires $O(M(n))$ operations, where $M(n)$ is the number of operations for an n-digit multiplication.

Multiple-precision multiplication of n-digit numbers can be performed in $M(n) = n \log n \log \log n$ operations by using the Schönhage-Strassen algorithm [66, 5, 45] which is the algorithm based on the fast Fourier transform (FFT) [27].

Parallel computation of $\sqrt{2}$ up to 1 million decimal digits has been performed by B. Char et al. [26] on a network of workstations in 1994. They used Karatsuba's multiplication algorithm [43, 45] which is the algorithm of $O(n^{\log_2 3})$.

However, in the multiple-precision multiplication of several thousand decimal digits or more, the FFT-based multiplication is the fastest. Thus, a parallelization of the multiple-precision

FFT-based multiplication algorithm is discussed in this chapter.

6.2 Newton Iteration

In division, the quotient of a and b is computed as follows. First the following Newton iteration is employed, which converges to $1/b$:

$$x_{k+1} = x_k(2 - bx_k). \tag{6.1}$$

To obtain the value of a/b, we have to calculate $a \cdot (1/b) = a/b$. These iterations are performed by doubling the precision for each iteration.

Square roots are computed by the following Newton iteration, which converges to $1/\sqrt{a}$:

$$x_{k+1} = \frac{x_k}{2}(3 - ax_k^2). \tag{6.2}$$

To obtain the value of $\sqrt{a}$, we have to calculate $(1/\sqrt{a}) \cdot a = \sqrt{a}$. These iterations are also performed by doubling the precision for each iteration.

Here, we discuss the sequential computation time of n-digit division and square root operation. Let us consider an n-digit number X with radix B.

$$X = \sum_{i=0}^{n-1} x_i B^i, \tag{6.3}$$

where $0 \leq x_i < B$.

In each iteration of the multiple-precision division includes two multiple-precision multiplications and that of the multiple-precision square root operation includes three multiple-precision multiplications. At the i-th iteration, it is sufficient to work with accuracy of $O(2^i)$. In this sense, the Newton iteration is self-correcting. Thus, the computation time of sequential processing $T_s(n)$ is as follows:

$$\begin{aligned} T_s(n) &= \sum_{i=0}^{\log_2 n} c_{mult} M(n/2^i) + M(n) \\ &\approx (2c_{mult} + 1) \cdot M(n), \end{aligned}$$

where $c_{mult} = 2$ (in division) or $c_{mult} = 3$ (in square root), and $M(n)$ is the number of operations for an n-digit FFT-based multiplication. Throughout this paper, we assume $M(n) = n \log n \log \log n$.

6.3 Parallelization of the Multiple-Precision Addition, Subtraction and Multiplication

In the multiple-precision parallel addition, subtraction and multiplication by single-precision integer, parallelization of releasing propagated carries and borrows is the key component in the

process speed. These propagation operations can be parallelized by the carry skip method [52].

Many multiple-precision multiplication algorithms have been proposed [45]. In this chapter, we discuss the multiple-precision multiplication by using the *floating point real* FFT [82, 41, 19].

Because many parallel FFT algorithms are proposed [86, 38, 79], we can use an efficient parallel FFT algorithm.

The normalization is essentially the same as the parallel processing of carry in the multiple-precision parallel addition, subtraction and multiplication by single-precision integer. Thus, this normalization can be parallelized with ease.

6.4 Parallelization of the Multiple-Precision Division and Square Root Operation

To compute the n-digit multiple-precision arithmetic in the Newton iteration of (6.1) and (6.2), it is necessary to perform the multiple-precision parallel addition, subtraction and multiplication on parallel computers which have P processors. In these operations, we can apply the parallel algorithm given in section 6.3.

6.4.1 Arithmetic Operation Counts

Arithmetic Operation Counts for Block Distribution

Since the arithmetic operation count of n-digit multiple-precision parallel addition, subtraction and multiplication by single-precision integer is $O(n/P)$, no consideration is done in this chapter.

In the case of the block distribution, n-digit multiple-precision numbers are distributed across all P processors. We denote the corresponding index at processor m $(0 \leq m \leq P-1)$ as i $(m = \lfloor i/\lceil n/P \rceil \rfloor)$ in (6.3).

Thus, the arithmetic operation count of the block distribution $T_{calc}^{block}(n, P)$ is as follows:

$$\begin{aligned} T_{calc}^{block}(n, P) &= \sum_{i=0}^{\log_2 \frac{n}{P}} c_{mult} M(2^i) + \sum_{i=1}^{\log_2 P} c_{mult} \cdot \frac{1}{2^i} M(n \cdot 2^i/P) + \frac{1}{P} M(n) \\ &\approx \frac{1}{P}(c_{mult} \log_2 P + 2c_{mult} + 1) M(n). \end{aligned} \tag{6.4}$$

Arithmetic Operation Counts for Cyclic Distribution

In the case of the cyclic distribution, n-digit multiple-precision numbers are also distributed across all P processors. We denote the corresponding index at processor m $(0 \leq m \leq P-1)$ as i $(m = i \bmod P)$ in (6.3).

Thus, the arithmetic operation count of the cyclic distribution $T_{calc}^{cyclic}(n,\ P)$ is as follows:

$$T_{calc}^{cyclic}(n,\ P) = \sum_{i=0}^{\log_2 P} c_{mult} \cdot \frac{1}{2^i} M(2^i) + \sum_{i=1}^{\log_2 P} c_{mult} \cdot \frac{1}{P} M(n \cdot 2^i / P) + \frac{1}{P} M(n)$$
$$\approx \frac{1}{P}(2c_{mult} + 1) M(n). \tag{6.5}$$

By comparing (6.4) and (6.5), we can conclude that the arithmetic operation count of the cyclic distribution is less than that of the block distribution.

6.4.2 Communication Time on Parallel Processing (Normalization)

We consider the communication time of the part of normalization (see Figure 4.3) in the multiple-precision multiplication. We assume a message passing model of computation.

Communication Time of Block Distribution

In the normalization of the block distribution, the processor m has to send the carry of one digit to the neighboring processor $(P + m - 1) \bmod P$.

Let us assume the latency of communication is L, the bandwidth is W, and the number of iterations at `DO WHILE` loop of Figure 4.3 is c_{norm}. Then the communication time of the block distribution $T_{comm}^{block}(n,\ P)$ is as follows:

$$T_{comm}^{block}(n,\ P) = c_{norm}\left\{(c_{mult}\log_2 P)\left(L + \frac{1}{W}\right) + \left(L + \frac{1}{W}\right)\right\}$$
$$= L \cdot \{c_{norm}(c_{mult}\log_2 P + 1)\} + \frac{1}{W} \cdot \{c_{norm}(c_{mult}\log_2 P + 1)\}. \tag{6.6}$$

Communication Time of Cyclic Distribution

In the cyclic distribution, to reduce the communication time, we realign the distributed data from the cyclic distribution to the block distribution before normalization. Then, we also realign the distributed data from the block distribution to the cyclic distribution after normalization. In the normalization of the cyclic distribution with realignment, the processor number m has to send the carry of one digit to the neighboring processor $(P + m - 1) \bmod P$.

Let us assume the latency of communication is L, the bandwidth is W, and the number of iterations at `DO WHILE` loop of Figure 4.3 is c_{norm}. Then, the communication time of the realignment (cyclic $\leftrightarrow$ block) with all-to-all communication $T_{comm}^{align}(n,\ P)$ is as follows:

$$T_{comm}^{align}(n,\ P) = (P-1)\left(L + \frac{1}{W} \cdot \frac{n}{P^2}\right).$$

In each iteration of (6.1) and (6.2), the realignment with all-to-all communication of twice (i.e., cyclic $\rightarrow$ block and block $\rightarrow$ cyclic) are necessary. The communication time of the normalization with realignment from the cyclic distribution to the block distribution $T_{comm}^{cyclic}(n,\ P)$ is as

follows:

$$T_{comm}^{cyclic}(n,\ P) = T_{comm}^{block}(n,\ P) + c_{mult} \sum_{i=0}^{\log_2 n} 2T_{comm}^{align}(2^i,\ P) + 2T_{comm}^{align}(n,\ P). \tag{6.7}$$

By comparing (6.6) and (6.7), we can see that the communication time of the block distribution is less than that of the cyclic distribution.

6.4.3 Total Computational Time

Here, we consider the total computational time of the multiple-precision parallel division and square root operation.

Total Computational Time of Block Distribution

Total computational time of the block distribution $T_{total}^{block}(n,\ P)$ is as follows:

$$T_{total}^{block}(n,\ P) = T_{calc}^{block}(n,\ P) + T_{comm}^{block}(n,\ P). \tag{6.8}$$

Total Computational Time of Cyclic Distribution

Total computational time of the cyclic distribution $T_{total}^{cyclic}(n,\ P)$ is as follows:

$$\begin{aligned} T_{total}^{cyclic}(n,\ P) = {} & T_{calc}^{cyclic}(n,\ P) + T_{comm}^{block}(n,\ P) \\ & + c_{mult} \sum_{i=0}^{\log_2 n} 2T_{comm}^{align}(2^i,\ P) + 2T_{comm}^{align}(n,\ P), \end{aligned} \tag{6.9}$$

By comparing (6.8) and (6.9), we can conclude that the total computational time of the cyclic distribution is less than that of the block distribution when

$$M(n) > \frac{2}{\log_2 P}\left(LP^2 \log_2 n + \frac{2n}{W}\right).$$

6.5 Experimental Results

To evaluate our parallel multiple-precision division and square root algorithms, decimal digit n of $\sqrt{2}/\pi$ and $\sqrt{\pi}$ and the number of processors P were varied. We averaged the elapsed times obtained from 10 executions of the multiple-precision parallel division $\sqrt{2}/\pi$ and multiple-precision parallel square root $\sqrt{\pi}$. We note that the value of n-digit π and $\sqrt{2}$ were prepared in advance. The choice of these values has no particular significance here, but was convenient as definite test cases for which the results could be checked for randomized test data.

A HITACHI SR2201 was used as distributed memory parallel computer. In the experiment, we used 4 PEs $\sim$ 256 PEs on the HITACHI SR2201.

MPI [54] on the HITACHI SR2201 was used as a communication library. All routines were written in FORTRAN. The compiler used was optimized FORTRAN77 V02-06-/A of Hitachi Ltd. The optimization option, `-W0,'opt(o(ss),approx(0))'` was specified.

The radix of the multiple-precision number is 10^8. The multiple-precision number is stored in the array of 32-bit integers. Each input data word is split into two words upon entry to the FFT-based multiplication.

Tables 6.1 and 6.2 show the averaged execution times of multiple-precision division ($\sqrt{2}/\pi$). The column headed by P shows the number of processors. The next six columns contain the average elapsed time in seconds. In Figures 6.1 and 6.2, we compare the average execution times of multiple-precision square division ($\sqrt{2}/\pi$). We can see that the performance of the cyclic distribution is better than that of the block distribution. This is mainly because the arithmetic operation time in the case of the cyclic distribution is shorter. In Figure 6.2, for small digits $N = 2^{20} \sim 2^{22}$ and $P > 64$, we can clearly see that communication overhead dominates the execution time.

Tables 6.3 and 6.4 show the result of the averaged execution times of multiple-precision square root ($\sqrt{\pi}$). The column headed by P shows the number of processors. The next six columns contain the average elapsed time in seconds. We can see that the tendency of the observed results in Figures 6.3 and 6.4 is almost the same as the results of Figures 6.1 and 6.2.

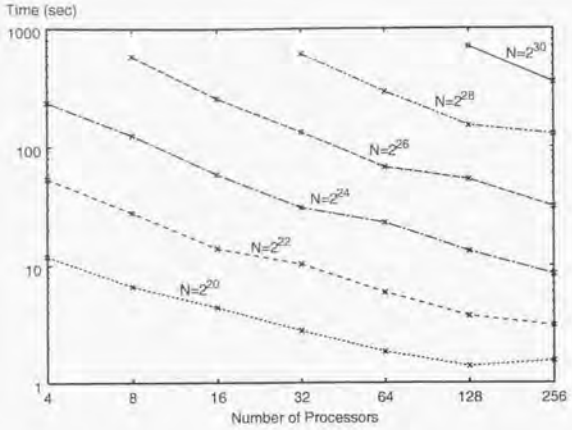


Figure 6.1: Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, block distribution). N = number of decimal digits.

Table 6.1: Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, block distribution) (in seconds), N = number of decimal digits.

(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	11.9369	53.8865	236.8914	*	*	*
8	6.5816	27.8718	124.9462	581.3678	*	*
16	4.3910	13.8523	58.3941	256.8172	*	*
32	2.7804	10.1804	30.5462	133.3248	608.5208	*
64	1.8298	5.8997	22.9757	67.7558	291.3710	*
128	1.3968	3.7244	13.1759	53.1426	151.6683	699.9246
256	1.5397	3.0976	8.4494	31.6179	128.7817	351.9944

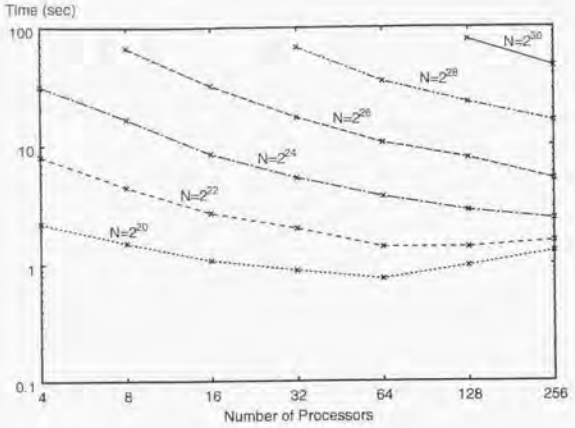


Figure 6.2: Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, cyclic distribution). N = number of decimal digits.

Table 6.2: Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$, cyclic distribution) (in seconds), N = number of decimal digits.
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	2.2126	8.1177	31.8555	*	*	*
8	1.4949	4.4671	16.7328	67.2696	*	*
16	1.0720	2.7101	8.5042	31.9142	*	*
32	0.8757	1.9826	5.3389	17.3196	67.6652	*
64	0.7423	1.3893	3.6850	10.5190	34.7479	*
128	0.9658	1.3860	2.8254	7.8244	23.0717	78.1338
256	1.2730	1.5582	2.3920	5.2297	15.9626	46.6842

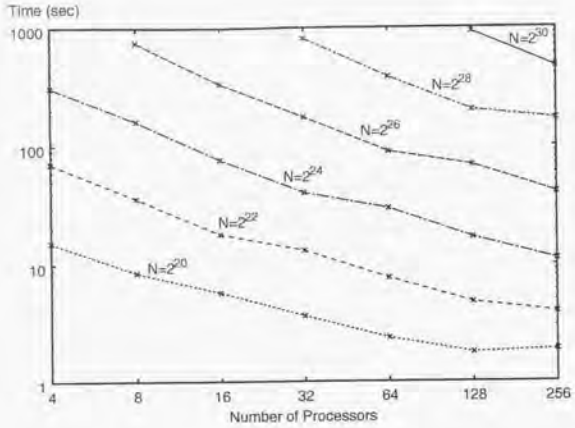


Figure 6.3: Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, block distribution), N = number of decimal digits.

Table 6.3: Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, block distribution) (in seconds), N = number of decimal digits.
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	15.2758	70.4181	310.4497	*	*	*
8	8.4737	36.3030	162.9026	759.0732	*	*
16	5.7397	17.9397	77.0243	336.8013	*	*
32	3.6156	13.0498	40.4459	174.3698	806.1861	*
64	2.3532	7.6094	29.7985	89.1936	383.0885	*
128	1.7700	4.7694	16.9661	69.1920	200.4155	918.5154
256	1.8889	3.9253	11.0589	40.6694	169.5797	464.0280

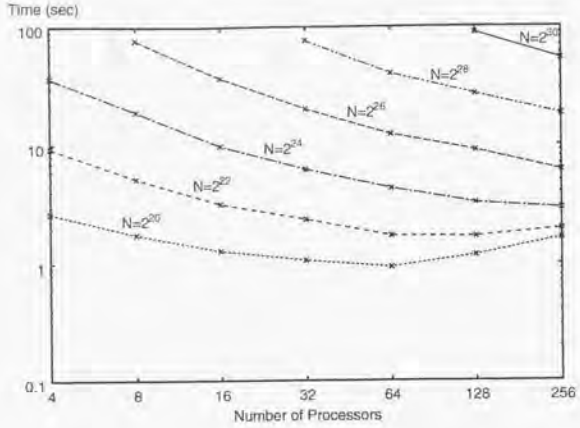


Figure 6.4: Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, cyclic distribution). N = number of decimal digits.

Table 6.4: Execution time of multiple-precision parallel square root ($\sqrt{\pi}$, cyclic distribution) (in seconds). N = number of decimal digits.
(* means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient).

$P \setminus N$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
4	2.6253	9.3042	36.3102	*	*	*
8	1.7562	5.1890	19.2272	77.3929	*	*
16	1.2796	3.1780	9.8195	36.7118	*	*
32	1.0581	2.3440	6.2100	19.9579	76.4990	*
64	0.9245	1.6986	4.3203	12.2624	39.8957	*
128	1.1724	1.6942	3.2651	9.0219	26.7400	89.2394
256	1.6455	1.9596	2.9304	6.1577	18.3057	53.9143

Chapter 7

Calculation of $\sqrt{2}$ to 137,438,950,000 Decimal Digits on the Distributed Memory Parallel Computer

7.1 Introduction

In this chapter, we compute more than 137 billion decimal digits of the square root of 2 to experiment with the multiple-precision parallel square root algorithm described in Chapter 6 on the distributed memory parallel computer.

The computation of the square root of 2 to high precision has a long history. R. Coustal [28] and H. S. Uhler [85] made use of binomial series expansions. K. Takahashi and M. Sibuya [81] employed an iterative method based on the formula

$$x_{k+1} = x_k(1.5 - 0.5ax_k^2) \tag{7.1}$$

which requires only multiple-precision multiplications and additions, and x_k converges to $1/\sqrt{a}$.

M. Lal [49] employed a special method which yields one digit at a time. In the later his calculations [51, 50] the Newton iteration was employed to extend the original result.

This Newton iteration for $\sqrt{a}$ is as follows:

$$x_{k+1} = \frac{x_k + a/x_k}{2}, \tag{7.2}$$

where x_0 is an initial approximation to $\sqrt{a}$.

J. Dutka [32] made use of a quadratically converging algorithm which derived from the Pell (Fermat) equation $P^2 - aQ^2 = 4$ (where a is a nonsquare positive integer) by means of recurrence relations involving multiplication. The approximation of $\sqrt{a}$ is represented by a suitable ratio of P/Q.

For calculating more than 137 billion decimal digits of $\sqrt{2}$, we used the Newton iteration for the reciprocal of the square root. This algorithm is considerably better than the formula based on (7.2), because of no full precision divisions are involved.

7.2 The Newton Iteration for Square Roots

Formula (7.1) can be represented as:

$$x_{k+1} = x_k + x_k \cdot (1 - a \cdot x_k^2)/2, \tag{7.3}$$

where the multiplication between x_k and $(1 - a \cdot x_k^2)/2$ can be performed with only half of the normal level of precision [11] and shorten computing time. Multiplying the final approximation to $1/\sqrt{a}$ by a gives the square root of a.

These iterations are performed by doubling the precision for each iteration.

7.3 Multiple-Precision Arithmetic

The reciprocal of the square root operation can be reduced to the multiple-precision addition, subtraction and multiplication by using the Newton iteration [11, 44]. We can use the multiple-precision parallel addition, subtraction and multiplication algorithms described in Chapter 4.

We used the *floating point real* FFT-based multiplication. For the *floating point real* FFT-based multiplication, we can use the "balanced representation" [29, 30] which tend to yield reduced errors for the convolutions we intend to perform.

A multiple-precision number is represented in the array of 32-bit integers. The radix selected for the multiple-precision numbers is 10^8. Each input data word is split into two words upon entry to the FFT-based multiplication.

Memory size of the multiple-precision FFT-based multiplication is much larger than the ordinary $O(n^2)$ multiplication method. For example, for performing the FFT-based multiplication between $2^{37} \approx 137$ billion decimal digit numbers, at least 1.7 TB of main memory should be available under the ideal conditions. It was impossible to obtain 137 billion decimal digits through in-core (on main memory) operations because of the maximum available main memory size of 224 GB which we were able to use on the distributed memory parallel computer HITACHI SR2201.

Thus, we performed 2^{29} point FFT for $2^{30} \approx 1.07$ billion decimal digit multiplications on main memory. Then, we used Karatsuba's algorithm which requires $O(n^{\log_2 3})$ operations [43, 45] for $2^{37} \approx 137$ billion decimal digit multiplications. These schemes needed about 204 GB of main memory for the working storage.

We can use the parallel algorithm for the multiple-precision square root operation given in Chapter 6.

Table 7.1: Frequency distribution for $\sqrt{2}-1$ up to 100,000,000,000 decimal digits.

Digit	Count
0	9999946091
1	10000062987
2	9999903614
3	9999996931
4	9999963242
5	9999985234
6	9999930492
7	10000091438
8	10000105868
9	10000014103

7.4 Results

The calculation of the square root of 2 by the Newton iteration was carried out on the distributed memory parallel computer HITACHI SR2201 (1024 PEs, total main memory 256 GB).

In (7.3), the initial value is given by $x_0 \approx \sqrt{0.5}$ with a 52-bit mantissa in IEEE 754 double-precision arithmetic. Then, x_k in (7.3) is converged to $1/\sqrt{0.5} = \sqrt{2}$.

The calculation of the square root of 2 was completed in 3rd of August 1997, which took 7 hours and 31 minutes which include the time for the verification.

The verification method is squaring the value of $\sqrt{2}$ and comparing it with $2 = 1.999\cdots$. The number obtained by squaring the approximation in the verification was one and decimal point followed by $2^{37} - 30 = 137{,}438{,}953{,}442$ nines. The results of square root of 2, which have been stored in the disk file, is in the form of 1024 different files, each containing $2^{27} = 134{,}217{,}728$ decimal digits and size of each file is 64 MB, and a last file on which the first 134,217,698 decimal digits are correct.

The results of tabulated frequencies for one digit string are listed in Table 7.1.

The decimal numbers of $\sqrt{2}$ from 137,438,953,217-th to 137,438,953,266-th digits are:

```
8913458017 7391236935 4900286855 3714574742 2009047472.
```

Analysis of digit sequences for 137,438,950,000 decimal digits of $\sqrt{2}-1$ gives some interesting features:

1. The longest ascending sequences are 45678901234 (from 4,027,971,080), 78901234567 (from

21,932,314,878, 51,177,313,690), 89012345678 (from 28,522,096,911, 56,308,436,119, 88,773,299,248 and 121,646,429,299), 01234567890 (from 88,055,854,279), 23456789012 (from 33,960,124,767, 41,669,414,929, 101,708,237,670 and 104,668,656,044) and 56789012345 (from 128,693,866,283, 132,288,691,729). The next longest ascending sequence of length 10 appears 97 times.

2. The longest descending sequence is 321098765432 (from 31,561,102,674). The next longest descending sequence of length 11 appears 13 times.

3. The sequence of maximum multiplicity (of 12) appears only once. This is 0 (from 64,678,262,264). The next longest sequence of multiplicity (of 11) appears 13 times.

4. The longest sequence of 2718281828 appears from 810,443,250, 10,855,468,698, 13,529,335,768, 14,656,415,520, 16,095,198,868, 28,958,822,656, 64,152,793,518 and 67,861,907,796. The next longest sequence of 271828182 appears 114 times.

5. The longest sequence of 14142135623 appears from 8,197,850,925 only once. The next longest sequence of 141421356 appears 11 times.

6. The longest sequence of 31415926535 appears 3 times. These are from 35,921,168,408, 65,099,003,919 and 110,305,459,937. The next longest sequence of 314159265 appears 17 times.

Chapter 8

Improvement of Algorithms for π Calculation

8.1 Introduction

The Gauss-Legendre algorithm [20, 65] and Borweins' quartically convergent algorithm [18] are often used for multiple-precision π calculation.

Although two algorithms include many multiple-precision multiplications and square operations, these operations can be reduced by transformation of expressions.

In general, it is known that the arithmetic operation count of the multiple-precision square operation is less than that of the multiple-precision multiplication. Thus, the arithmetic operations of the calculation of π can be reduced by replacing the multiple-precision multiplication by the multiple-precision square operation as much as possible.

Since the arithmetic operation count of n-digit multiple-precision addition, subtraction and multiplication by single-precision integer is $O(n)$, no consideration is done in this chapter.

Multiple-precision multiplication of n-digit numbers can be performed in $O(n \log n \log \log n)$ operations by using the fast Fourier transform (FFT) [66, 45]. In the multiple-precision multiplication of several thousand decimal digits or more, the FFT-based multiplication is the fastest. In this chapter, we use the FFT-based multiplication algorithm.

8.2 The Gauss-Legendre Algorithm

In 1976 R. P. Brent [20] and E. Salamin [65] independently discovered an approximation algorithm based on elliptic integrals that yields quadratic convergence to π.

We first define the arithmetic-geometric mean $\mathrm{agm}(a_0, b_0)$. Let a_0, b_0 and c_0 be positive numbers satisfying $a_0^2 = b_0^2 + c_0^2$. Define a_i, the sequence of arithmetic means, and b_i, the

sequence of geometric means, by

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1}), \quad b_i = \sqrt{a_{i-1} b_{i-1}}. \tag{8.1}$$

Also, define a positive sequence c_i:

$$c_i^2 = a_i^2 - b_i^2. \tag{8.2}$$

We note that two relations (8.1), (8.2) easily follow from the following:

$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1}) = a_{i-1} - a_i, \quad c_i^2 = 4a_{i+1}c_{i+1}. \tag{8.3}$$

After i iterations, π can be approximated by π_i:

$$\pi_i = \frac{4a_{i+1}^2}{1 - \sum_{j=1}^{i} 2^{j+1} c_j^2} = \frac{a_{i+1}^2}{\frac{1}{4} - \sum_{j=1}^{i} 2^{j-1} c_j^2}. \tag{8.4}$$

The formula (8.4) has the second order convergence nature. Then the sequences of agm and agm related π to decimal precision n are performed by the following algorithm [20]:

$A := 1;\ B := 2^{-1/2};\ T := 1/4;\ X := 1;$
while $A - B > 10^{-n}$ **do begin**
$Y := A;\ A := (A+B)/2;\ B := \sqrt{B * Y};$
$T := T - X * (Y - A)^2;\ X := 2 * X$
end;
return $(A+B)^2/(4 * T)$.

Here, A, B, T and Y are full-precision variables and X is a double-precision variable.

Although the Gauss-Legendre algorithm has the operations of square root and reciprocal calculation, these calculations can be reduced to the multiple-precision addition, subtraction and multiplication by using the Newton iteration. The arithmetic operation count of n-digit multiple-precision addition, subtraction and multiplication by single-precision integer is clearly $O(n)$. Let the arithmetic operations of n-digit multiple-precision multiplication is $M(n)$, then π can be performed in $O(M(n) \log n)$ steps [20].

In each iteration of above algorithm, we have to calculate the following values:

i) $a_{i-1} b_{i-1}$,

ii) $b_i := \sqrt{a_{i-1} b_{i-1}}$,

iii) $c_i^2 := (a_{i-1} - a_i)^2$.

In total, the multiplication of once, the square operation of once and the square root operation of once are needed in the Gauss-Legendre algorithm. Since the arithmetic operation count of n-digit multiple-precision addition, subtraction and multiplication by single-precision integer is $O(n)$, no further consideration is needed here.

We note that square roots are computed by employing the following Newton iteration, which converges to $1/\sqrt{a}$:

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - ax_k^2). \tag{8.5}$$

Then, the final iteration is performed as follows [44]:

$$\sqrt{a} \approx (ax_k) + \frac{x_k}{2}(a - (ax_k)^2). \tag{8.6}$$

8.2.1 Improvement of the Gauss-Legendre Algorithm

In general, it is known that the arithmetic operation count of the multiple-precision square operation is less than that of the multiple-precision multiplication. Thus, the arithmetic operations of the calculation of π can be reduced by replacing the multiple-precision multiplication by the multiple-precision square operation as much as possible.

First, $b_i^2 = a_i^2 - c_i^2$ is obtained from (8.2). Furthermore, $b_i^2 = a_{i-1}b_{i-1}$ is also obtained from (8.1). Hence, the multiple-precision multiplication $(a_{i-1}b_{i-1})$ can be replaced to $(a_i^2 - c_i^2)$. However, $c_i^2 = (a_i - b_{i-1})^2$ is a value which should be computed in any case, the multiple-precision multiplication by $a_{i-1}b_{i-1}$ will be substantially obtained from the calculation of a_i^2.

In the first iteration, a_1, b_1, c_1 are respectively defined as the following:

$$a_1 = \frac{a_0 + b_0}{2} = \frac{2+\sqrt{2}}{4}, \quad b_1 = \sqrt{a_0 b_0} = 2^{-1/4}, \quad c_1^2 = a_1^2 - b_1^2 = \frac{3-2\sqrt{2}}{8}. \tag{8.7}$$

We show the improved Gauss-Legendre algorithm for π is as follows:

$A := (2+\sqrt{2})/4$; $B := 2^{-1/4}$; $T := (2\sqrt{2}-1)/8$; $X := 2$;
while $A - B > 10^{-n}$ **do begin**
  $A := (A+B)/2$; $B := (A-B)^2$;
  $T := T - X * B$; $B := \sqrt{A^2 - B}$; $X := 2 * X$
  end;
return $(A+B)^2/(4*T)$.

Here, A, B and T are full-precision variables and X is a double-precision variable.

We summarized the comparison of the number of operations in each iteration of the Gauss-Legendre algorithm in Table 8.1. Improved algorithm has no multiplication operation.

Table 8.1: Comparison with the number of operations in each iteration of the Gauss-Legendre algorithm.

	Original algorithm	Improved algorithm
Multiplication	1	0
Square	1	2
Square root	1	1

8.3 Borweins' Quartically Convergent Algorithm

Borweins' quartically convergent algorithm [18] is explained as the following scheme. Let $a_0 = 6 - 4\sqrt{2}$ and $y_0 = \sqrt{2} - 1$. Iterate the following calculations:

$$y_{k+1} = \frac{1 - (1 - y_k^4)^{1/4}}{1 + (1 - y_k^4)^{1/4}}, \tag{8.8}$$

$$a_{k+1} = a_k(1 + y_{k+1})^4 - 2^{2k+3} y_{k+1}(1 + y_{k+1} + y_{k+1}^2). \tag{8.9}$$

Then a_k converges quartically to $1/\pi$. Here, precisions for a_k and y_k must be more than the desired digits. This algorithm was used for the main run of the 29 million decimal digit calculation done by D. H. Bailey [8].

We note that reciprocals are computed by employing the following Newton iteration, which converges to $1/a$:

$$x_{k+1} = x_k + x_k(1 - ax_k). \tag{8.10}$$

4-th roots are computed by the following Newton iteration, which converges to $a^{-1/4}$:

$$x_{k+1} = x_k + \frac{x_k}{4}(1 - ax_k^4). \tag{8.11}$$

To obtain the value of $a^{1/4}$, we have to calculate $a^{1/4} = (a^{-1/4})^3 \cdot a$.

8.3.1 Improvement of Borweins' Quartically Convergent Algorithm

To obtain y_{k+1} in formula (8.8), we have to calculate the following values:

i) y_k^2,

ii) $y_k^4 = (y_k^2)^2$,

iii) $(1 - y_k^4)^{-1/4}$,

iv) $(1 - y_k^4)^{1/4} = ((1 - y_k^4)^{-1/4})^2 \times (1 - y_k^4)^{-1/4} \times (1 - y_k^4)$,

v) $(1 + (1 - y_k^4)^{1/4})^{-1}$,

vi) $(1+(1-y_k^4)^{1/4})^{-1} \times (1-(1-y_k^4)^{1/4})$.

In total, the multiplication of three times, the square operation of three times, the reciprocal operation of once and the reciprocal 4-th root operation of once are necessary.

However, it has already done by the calculation of π for more than 201 million decimal digits in 1988 which Y. Kanada [41] did and the frequency of the multiple-precision multiplication and square operation can be reduced by doing following improvement algorithm.

First, (8.8) can be transformed as follows:

$$\begin{aligned} y_{k+1} &= \frac{1-(1-y_k^4)^{1/4}}{1+(1-y_k^4)^{1/4}} \\ &= 1 - \frac{2}{1+(1-y_k^4)^{-1/4}}. \end{aligned} \tag{8.12}$$

Since (8.8) is transformed into (8.12), the multiple-precision multiplication with $(1-(1-y_k)^{1/4})$ can be reduced.

Moreover, (8.12) has the reciprocal 4-th root, we can reduce the operation of $(1-y_k^4)^{1/4} = ((1-y_k^4)^{-1/4})^2 \times (1-y_k^4)^{-1/4} \times (1-y_k^4)$. Thus, we can reduced the multiplication of twice and the square operation of once.

Finally, to obtain y_{k+1}, we can reduce the multiple-precision multiplication of three times and the multiple-precision square operation of once.

To obtain a_{k+1} in formula (8.9), we have to calculate the following values:

i) $(1+y_{k+1})^2$,

ii) $(1+y_{k+1})^4 = ((1+y_{k+1})^2)^2$,

iii) $a_k \times (1+y_{k+1})^4$,

iv) y_{k+1}^2,

v) $y_{k+1} \times (1+y_{k+1}+y_{k+1}^2)$.

In total, the multiplication of twice and the square operation of three times are necessary.

However, the calculation of $y_{k+1}(1+y_{k+1}+y_{k+1}^2)$ can be transformed to:

$$\begin{aligned} y_{k+1}(1+y_{k+1}+y_{k+1}^2) &= \frac{(1+y_{k+1})^4 - (1+2y_{k+1}^2+y_{k+1}^4)}{4} \\ &= \frac{(1+2y_{k+1}+y_{k+1}^2)^2 - (1+2y_{k+1}^2+y_{k+1}^4)}{4}. \end{aligned} \tag{8.13}$$

In (8.13), thus, we can calculate the following values:

i) y_{k+1}^2,

ii) $y_{k+1}^4 = (y_{k+1}^2)^2$,

Table 8.2: Comparison with the number of operations in each iteration of Borweins' quartically convergent algorithm.

	Original algorithm	Improved algorithm
Multiplication	5	1
Square	6	3
Reciprocal 4-th Root	1	1
Reciprocal	1	1

iii) $(1+y_{k+1})^4 = (1+2y_{k+1}+y_{k+1}^2)^2$,

iv) $a_k \times (1+y_{k+1})^4$.

This improved scheme has only the multiplication of once, the square operation of three times. Thus, we can reduce the multiplication of once.

Furthermore, if the value of y_{k+1}^4 is preserved in (8.13), the calculation of y_{k+1}^4 is unnecessary in each iteration when assuming $k \to k+1$ in (8.8). Hence this algorithm shows the 4-th power calculation can be reduced, that is, the twice square operation can be reduced.

As for the iteration of once, the multiplication of four times and square operation of three times can be reduced by improving Borweins' quartically convergent algorithm (refer to Table 8.2).

8.3.2 Improvement in the Final Iteration of Borweins' Quartically Convergent Algorithm

In the final iteration, the reciprocal 4-th root and reciprocal can be omitted as follows in (8.8) and (8.9).

First, we obtain Taylor series expansion to $(1-y_k^4)^{-1/4}$ in (8.12) as follows:

$$(1-y_k^4)^{-1/4} = 1 + \frac{1}{4}y_k^4 + \frac{5}{32}y_k^8 + \frac{15}{128}y_k^{12} + \cdots. \tag{8.14}$$

Then,

$$y_{k+1} = \frac{1}{8}y_k^4 + \frac{1}{16}y_k^8 + \frac{21}{512}y_k^{12} + \cdots, \tag{8.15}$$

$$(1+y_{k+1})^4 = 1 + \frac{1}{2}y_k^4 + \frac{11}{32}y_k^8 + \frac{17}{64}y_k^{12} + \cdots \tag{8.16}$$

and

$$y_{k+1}(1+y_{k+1}+y_{k+1}^2) = \frac{1}{8}y_k^4 + \frac{5}{64}y_k^8 + \frac{15}{256}y_k^{12} + \cdots. \tag{8.17}$$

To obtain the value of π, only a_{k+1} in (8.9) is necessary. Thus, in (8.8) we only have to calculate the value of a_k from (8.16) and (8.17) at the final iteration.

(i) Case of $10^{-n} \leq y_k^4 < 10^{-n/2}$

Since $y_k^8 < 10^{-n}$, (8.16) and (8.17) can be approximated with:

$$(1 + y_{k+1})^4 \approx 1 + \frac{1}{2}y_k^4, \tag{8.18}$$

$$y_{k+1}(1 + y_{k+1} + y_{k+1}^2) \approx \frac{1}{8}y_k^4. \tag{8.19}$$

Thus, the multiple-precision multiplication, square operation, 4-th root and reciprocal calculation are unnecessary.

(ii) Case of $10^{-n/2} \leq y_k^4 < 10^{-n/3}$

Equations (8.16) and (8.17) can be approximated with:

$$(1 + y_{k+1})^4 \approx 1 + \frac{1}{2}y_k^4 + \frac{11}{32}y_k^8, \tag{8.20}$$

$$y_{k+1}(1 + y_{k+1} + y_{k+1}^2) \approx \frac{1}{8}y_k^4 + \frac{5}{64}y_k^8. \tag{8.21}$$

Thus, the 4-th root and reciprocal calculation are unnecessary.

(iii) Case of $10^{-n/3} \leq y_k^4 < 10^{-n/4}$

Equations (8.16) and (8.17) can be approximated with:

$$(1 + y_{k+1})^4 \approx 1 + \frac{1}{2}y_k^4 + \frac{11}{32}y_k^8 + \frac{17}{64}y_k^{12}, \tag{8.22}$$

$$y_{k+1}(1 + y_{k+1} + y_{k+1}^2) \approx \frac{1}{8}y_k^4 + \frac{5}{64}y_k^8 + \frac{15}{256}y_k^{12}. \tag{8.23}$$

Furthermore, the 4-th root and reciprocal calculation are unnecessary.

We summarize the improvement of Borweins' quartically convergent algorithm for π is as follows:

$A := 6 - 4\sqrt{2}$; $Y := 17 - 12\sqrt{2}$; $X := 2$;
while $Y > 10^{-n/4}$ **do begin**
$Y := 1 - \dfrac{2}{1 + (1 - Y)^{-1/4}}$; $B := Y^2$;
$W := (1 + 2 * Y + B)^2$; $Y := B^2$;
$A := A * W - X * (W - (1 + 2 * B + Y))$;
$X := 4 * X$
end;
if $Y < 10^{-n/2}$ **then begin**
$W := 1 + Y/2$; $A := A * W - X * (Y/8)$

```
        end
    else if Y < 10^{-n/3} then begin
        B := Y^2; W := 1 + Y/2 + 11 * B/32;
        A := A * W - X * (Y/8 + 5 * B/64)
        end
    else begin
        B := Y^2; W := 1 + Y/2 + 11 * B/32 + 17 * B * Y/64;
        A := A * W - X * (Y/8 + 5 * B/64 + 15 * B * Y/256)
        end;
    return 1/A.
```

Here, A, B, W and Y are full-precision variables and X is a double-precision variable.

8.4 Experimental Results

For evaluating the improved two algorithms, m were changed and averaged CPU time as obtained with execution of calculation of $n = 2^m$ decimal digits of π. We implemented the algorithm based on the FFT-based arithmetic. IBM RS6000/590 workstation was used in the experiment.

Tables 8.3 and 8.4 show the results of each execution time of the Gauss-Legendre algorithm and Borweins' quartically convergent algorithm.

We note that the improved Gauss-Legendre algorithm is up to 1.08 times faster than the original Gauss-Legendre algorithm. This is because the arithmetic operations of the calculation of π can be reduced by replacing the multiple-precision multiplication with the multiple-precision square operation.

Furthermore, we note that the improved Borweins' quartically convergent algorithm is up to 1.78 times faster than the original Borweins' quartically convergent algorithm. This is because the improved Borweins' quartically convergent algorithm has less multiplication and square operation compared with the original Borweins' quartically convergent algorithm.

Table 8.3: Comparison with the performance of the Gauss-Legendre algorithm (in seconds).

m	$n = 2^m$ (digits)	Original algorithm	Improved algorithm	ratio
10	1024	0.14	0.13	1.077
11	2048	0.29	0.27	1.074
12	4096	0.67	0.64	1.047
13	8192	1.33	1.30	1.023
14	16384	2.86	2.75	1.040
15	32768	6.29	6.09	1.033
16	65536	15.99	14.96	1.069
17	131072	39.19	36.71	1.068
18	262144	89.98	84.78	1.061
19	524288	217.62	203.27	1.071
20	1048576	517.55	484.09	1.069

Table 8.4: Comparison with the performance of Borweins' quartically convergent algorithm (in seconds).

m	$n = 2^m$ (digits)	Original algorithm	Improved algorithm	ratio
10	1024	0.20	0.12	1.667
11	2048	0.42	0.25	1.680
12	4096	0.91	0.53	1.717
13	8192	1.83	1.16	1.578
14	16384	4.25	2.53	1.680
15	32768	8.84	5.39	1.640
16	65536	25.36	14.23	1.782
17	131072	58.28	33.48	1.741
18	262144	137.20	81.19	1.690
19	524288	332.06	191.17	1.737
20	1048576	795.77	469.63	1.694

Chapter 9

Calculation of π to 51,539,600,000 Decimal Digits on the Distributed Memory Parallel Computer

9.1 Introduction

The computation of π with high precision has a long history [13]. Several computations have been performed as in Table 9.1. The development of new programs suited to the calculation of π and new high speed computers with large memory have thrown more light on this fascinating number.

There are many arctangent relations for π [82]. In particular, all the computations until 1981 and verification for 10,000,000 decimal digit calculation by Y. Ushiro and Y. Kanada [42] used arctangent formulae such as:

$$\begin{aligned}
\pi &= 16\arctan\frac{1}{5} - 4\arctan\frac{1}{239}, && \text{Machin}\\
&= 24\arctan\frac{1}{8} + 8\arctan\frac{1}{57} + 4\arctan\frac{1}{239}, && \text{Störmer}\\
&= 48\arctan\frac{1}{18} + 32\arctan\frac{1}{57} - 20\arctan\frac{1}{239}, && \text{Gauss}\\
&= 32\arctan\frac{1}{10} - 4\arctan\frac{1}{239} - 16\arctan\frac{1}{515}. && \text{Klingenstierna}
\end{aligned}$$

On the other hand, D. V. Chudnovsky and G. V. Chudnovsky computed π up to over 8 billion decimal digits by using the following algorithm which they found:

$$\frac{1}{\pi} = \frac{6541681608}{640320^{3/2}} \sum_{k=0}^{\infty} \left(\frac{13591409}{545140134} + k\right) \frac{(6k)!}{(3k)!(k!)^3} \frac{(-1)^k}{(640320)^{3k}}. \tag{9.1}$$

and the computer made by themselves in 1996 [16].

Table 9.1: Historical records of the π calculation by computers.

Calculated by	Machine used	Date	Precision (calculated) declared	Precision correct	Time (check)	Formula (check)
Reitwiesner et al.	ENIAC	1949	(2040)	2037	$\sim$ 70h ($\sim$ 70h)	M(M)
Nicholson, Jeenel	NORC	1954	(3093)	3092	13 m (13 m)	M(M)
Felton	Pegasus	1957	(10021)	7480	33 h (33 h)	K(G)
Genuys	IBM 704	1958	(10000)	10000	1 h 40 m (1 h 40 m)	M(M)
Felton	Pegasus	1958	(10021)	10020	33 h (33 h)	K(G)
Guilloud	IBM 704	1959	(16167)	16167	4 h 18 m (4 h 18 m)	M(M)
Shanks, Wrench	IBM 7090	1961	(100265)	100265	8 h 43 m (4 h 22 m)	S(G)
Guilloud, Filliatre	IBM 7030	1966	(250000)	250000	41 h 55 m (24 h 35 m)	G(S)
Guilloud, Dichampt	CDC 6600	1967	(500000)	500000	28 h 10 m (16 h 35 m)	G(S)
Guilloud, Bouyer	CDC 7600	1973	(1001250)	1001250	23 h 18 m (13 h 40 m)	G(S)
Miyoshi and Kanada	FACOM M-200	1981	(2000040) 2000000	2000030	137 h 18 m (143 h 18 m)	K(M)
Guilloud	x	1981-82	(x) 2000050	2000050	x (x)	x (x)
Tamura	MELCOM 900II	1982	(2097152)	2097144	7 h 14 m (2 h 21 m)	L(L)
Tamura and Kanada	HITAC M-280H	1982	(4194304)	4194288	2 h 21 m (6 h 52 m)	L(L)
Tamura and Kanada	HITAC M-280H	1982	(8388608)	8388576	6 h 52 m ($\geq$ 30 h)	L(L)
Kanada, Yoshino and Tamura	HITAC M-280H	1982	(16777216)	16777206	$\geq$ 30 h (6 h 36 m)	L(L)
Ushiro and Kanada	HITAC S-810/20	1983.10	(10013400) 10013395	10000000	$\leq$ 24 h ($\geq$ 30 h)	G(L)
Gosper	Symbolics 3670	1985.10	($\geq$ 17526200)	17526200	x (28 h)	R(B4)
Bailey	CRAY-2	1986.1	(29360128) 29360000	29360111	28 h (40 h)	B4(B2)
Kanada and Tamura	HITAC S-810/20	1986.9	(33554432) 33554400	33554414	6 h 36 m (23 h)	L(L)
Kanada and Tamura	HITAC S-810/20	1986.10	(67108864)	67108839	23 h (35 h 15 m)	L(L)
Kanada, Tamura, Kubo, et al.	NEC SX-2	1987.1	(134217728) 133554400	134217700	35 h 15 m (48 h 2 m)	L(B4)
Kanada and Tamura	HITAC S-820/80	1988.1	(201326572) 201326000	201326551	5 h 57 m (7 h 30 m)	L(B4)
Chudnovskys	CRAY-2 IBM-3090/VF	1989.5	($\geq$ 480000000) 480000000	4800000007	$\geq$ 6 month? (x)	C(C)
Chudnovskys	IBM-3090	1989.6	($\geq$ 525229270) 525229270	525229270	$\geq$ 1 month? (x)	C(x)
Kanada and Tamura	HITAC S-820/80	1989.7	(536870912) 536870000	536870898	67 h 13 m (80 h 39 m)	L(B4)
Chudnovskys	IBM-3090	1989.8	($\geq$ 1011196691) 1011196691	1011196691?	$\geq$ 2 month? (x)	C[C)
Kanada and Tamura	HITAC S-820/80	1989.11	(1073741824) 1073740000	1073741799	74 h 30 m (85 h 57 m)	L(B4)
Chudnovskys	m zero	1991.5	($\geq$ 2260000000) 2260000000?	22600000007	250 h? (x)	C(C)
Chudnovskys	x	1994.5	($\geq$ 4044000000) 4044000000?	40440000007	x (x)	C(C)
Takahashi and Kanada	HITAC S-3800/480	1995.6	(3221225472) 3221220000	3221225466	36 h 32 m (53 h 43 m)	B4(L)
Takahashi and Kanada	HITAC S-3800/480	1995.8	(4294967296) 4294960000	4294967286	113 h 41 m (130 h 20 m)	B4(L)
Takahashi and Kanada	HITAC S-3800/480	1995.10	(6442450944) 6442450000	6442450938	116 h 38 m (131 h 40 m)	B4(L)
Chudnovskys	x	1996.3	($\geq$ 8000000000) 8000000000?	80000000007	1 week? (x)	C(C)
Takahashi and Kanada	HITACHI SR2201	1997.4	(17179869184)	17179869142	5 h 11 m (5 h 26 m)	L(B4)
Takahashi and Kanada	HITACHI SR2201	1997.5	(34359738368)	34359738327	15 h 19 m (20 h 34 m)	B4(L)
Takahashi and Kanada	HITACHI SR2201	1997.7	(51539607552) 51539600000	51539607510	29 h 3 m (37 h 8 m)	B4(L)

M, K, G, S, L, R, B4, B2, C are formulae of Machin, Klingenstierna, Gauss, Störmer, and Gauss-Legendre, Ramanujan, Borweins' quartic convergent, Borweins' quadratic convergent, and Chudnovskys' formula, respectively. Symbol 'x' means 'unknown'. Check time means the additional time for the calculated value checking.

In 1976 R. P. Brent [20] and E. Salamin [65] independently discovered an approximation algorithm based on elliptic integrals that yields quadratic convergence to π (hereafter called the Gauss-Legendre algorithm). Later in 1983, quadratic, cubic, quadruple and septet convergent product expansion for π, which are competitive with Brent's and Salamin's formula, were also discovered by Borweins [18]. These new formulae are based on the arithmetic-geometric mean, a process whose rapid convergence doubles, triples, quadruples and septates the number of significant digits at each step.

The author and Y. Kanada have computed π up to more than 51.5 billion decimal digits by using the formula of improved Borweins' quartically convergent algorithm [77] and verified the results through the improved Gauss-Legendre algorithm [77] for π.

To attain more speed than before, parallelization schemes to the multiple-precision addition, subtraction and multiplication [74] were performed on a distributed memory parallel computer.

Multiple-precision arithmetic algorithms are described in Section 9.2. The results of the calculation of π to more than 51.5 billion decimal digits are described in Section 9.3.

9.2 Multiple-Precision Arithmetic

9.2.1 Multiple-Precision Addition, Subtraction and Multiplication

The improved Borweins' quartically convergent algorithm and the improved Gauss-Legendre algorithm have the reciprocal, square root and reciprocal 4-th root calculations, respectively. These calculations can be reduced to the multiple-precision addition, subtraction and multiplication by using the Newton iteration [11, 44]. We can use the multiple-precision parallel addition, subtraction and multiplication algorithms described in Chapter 4.

We used the *floating point real* FFT-based multiplication. Similarly to the calculation of the square root 2, we can use the "balanced representation" for the *floating point real* FFT-based multiplication [29, 30] which tend to yield reduced errors for the convolutions we intend to perform.

A multiple-precision number is represented in the array of 32-bit integers. The radix selected for the multiple-precision numbers is 10^8. Each input data word is split into two words upon entry to the FFT-based multiplication.

Memory size of the multiple-precision FFT-based multiplication is much larger than the ordinary $O(n^2)$ multiplication method. To perform the FFT-based multiplication of $3 \times 2^{34} \approx 51.5$ billion decimal digit numbers, at least 648 GB of main memory should be available under the ideal conditions. It was impossible to obtain 51.5 billion decimal digits through in-core (on main memory) operations because of the maximum available main memory size of 224 GB which we were able to use on the distributed memory parallel computer of HITACHI SR2201.

Thus, in Borweins' quartically convergent algorithm, we performed 3×2^{30} point FFT for

$3 \times 2^{31} \approx 6.4$ billion decimal digit multiplications on main memory. Then, we used Karatsuba's algorithm which requires $O(n^{\log_2 3})$ operations [43, 45] for $3 \times 2^{34} \approx 51.5$ billion decimal digit multiplications. These schemes needed about 212 GB of main memory for the working storage.

On the other hand, in the Gauss-Legendre algorithm, we performed 3×2^{29} point FFT for $3 \times 2^{30} \approx 3.2$ billion decimal digit multiplications on main memory. This is because the Gauss-Legendre algorithm requires more multiple-precision variables than Borweins' quartically convergent algorithm. These schemes needed about 188 GB of main memory as for the working storage.

9.2.2 Multiple-Precision Reciprocal

Reciprocals are computed using the following Newton iteration, which converges to $1/a$:

$$x_{k+1} = x_k + x_k(1 - ax_k), \tag{9.2}$$

where the multiplication between x_k and $(1 - ax_k)$ can be performed with only half of the normal level of precision [11]. These iterations are performed with a dynamic precision level.

9.2.3 Multiple-Precision Square Root

Square roots are computed by the following Newton iteration, which converges to $1/\sqrt{a}$:

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - ax_k^2), \tag{9.3}$$

where the multiplication between x_k and $(1 - ax_k^2)/2$ can be performed with only half of the normal level of precision.

Then, the final iteration is performed as follows [44]:

$$\sqrt{a} \approx (ax_k) + \frac{x_k}{2}(a - (ax_k)^2), \tag{9.4}$$

where the multiplications to ax_k and x_k are performed with only half of the final level of precision.

9.2.4 Multiple-Precision Reciprocal 4-th Root

Reciprocal 4-th roots are computed by the following Newton iteration, which converges to $a^{-1/4}$:

$$x_{k+1} = x_k + \frac{x_k}{4}(1 - ax_k^4), \tag{9.5}$$

where the multiplication between x_k and $(1 - ax_k^4)/4$ can be performed with only half of the normal level of precision.

9.3 Results of π 51,539,600,000 Decimal Digit Calculation

The calculations of π by the Borweins' quartically convergent algorithm and Gauss-Legendre algorithm were carried out on the distributed memory parallel computer HITACHI SR2201 (1024 PEs, main memory 256 GB). The original program is written in FORTRAN 77 with MPI [54]. To reduce the communication overhead, a Remote Direct Memory Access (RDMA) message transfer protocol [17] without memory copy was used as a communication library in optimized main/verification program.

Main program run:

Job start	: 6th June 1997 22:29:06
Job end	: 8th June 1997 03:32:17
Elapsed time	: 29:03:11
Main memory	: 212 GB
Algorithm	: Borweins' quartically convergent algorithm

Optimized main program run:

Job start	: 1st August 1997 23:04:15
Job end	: 3rd August 1997 00:18:47
Elapsed time	: 25:14:32
Main memory	: 212 GB
Algorithm	: Borweins' quartically convergent algorithm

Optimized verification program run:

Job start	: 4th July 1997 22:11:42
Job end	: 6th July 1997 11:19:58
Elapsed time	: 37:08:16
Main memory	: 188 GB
Algorithm	: Gauss-Legendre algorithm

The decimal numbers of π and $1/\pi$ from 51,539,599,951-st to 51,539,600,000-th digits are:

π : 1900691944 0299999207 6824359555 7053246569 8614212904

$1/\pi$: 4531204418 2539535923 9327200920 6008150624 6219272973.

Furthermore, the distribution of the figures of 0–9 up to the decimal point 50,000,000,000 digits of π and $1/\pi$ is shown in Tables 9.2 and 9.3.

Main computation took 18 iterations of Borweins' quartically convergent algorithm for $1/\pi$, followed by a reciprocal operation, to yield $3 \times 2^{34} = 51,539,607,552$ digits of π. This computation was checked using 35 iterations of the Gauss-Legendre algorithm for π. A comparison of these output results gave no discrepancies except for the last 42 digits due to the normal

truncation errors.

Analysis of digit sequences for 51,539,600,000 decimal digits of $\pi - 3$ gives some interesting features:

1. The longest ascending sequences are 45678901234 (from 2,401,798,228), 23456789012 (from 4,055,974,863), 12345678901 (from 7,997,135,197, 47,404,247,915), 56789012345 (from 17,664,375,855) and 89012345678 (from 29,085,092,351). The next longest ascending sequence of length 10 appears 51 times.

2. The longest descending sequence is 76543210987 (from 17,223,851,531), 09876543210 (from 42,321,758,803). The next longest descending sequence of length 10 appears 33 times.

3. The sequences of maximum multiplicity (of 11) appear 4 times. These are 1 (from 15,647,738,228), 9 (from 27,014,073,304) and 6 (from 32,104,158,792, 40,863,606,404). The next longest sequence of multiplicity (of 10) appears 39 times.

4. The longest sequence of 27182818284 appears (from 45,111,908,393) only once. The next longest sequence of 2718281828 appears 5 times.

5. The longest sequence of 1414213562 appears from 10,037,891,176, 12,888,529,951, 17,404,920,660, 24,149,232,165, 31,170,773,565, 40,081,788,717, 46,156,779,825, 47,945,472,360 and 48,610,722,512. The next longest sequence of 141421356 appears 49 times.

6. The longest sequence of 3141592653 appears 4 times. These are from 7,902,183,159, 13,381,905,334, 17,387,932,788 and 45,531,531,119. The next longest sequence of 314159265 appears 50 times.

Table 9.2: Frequency distribution for $\pi - 3$ up to 50,000,000,000 decimal digits.

Digit	Count
0	5000012647
1	4999986263
2	5000020237
3	4999914405
4	5000023598
5	4999991499
6	4999928368
7	5000014860
8	5000117637
9	4999990486

Table 9.3: Frequency distribution for $1/\pi$ up to 50,000,000,000 decimal digits.

Digit	Count
0	4999969955
1	5000113699
2	4999987893
3	5000040906
4	4999985863
5	4999977583
6	4999990916
7	4999985552
8	4999881183
9	5000066450

Chapter 10

Conclusion

This thesis has proposed parallel multiple-precision arithmetic algorithms which include the multiple-precision addition, subtraction, multiplication, division and square root operation.

For the very high precision calculation of mathematical constants, multiple-precision arithmetic algorithms are the key to reducing the execution time and increasing the number of calculation digits. We have succeeded for parallelizing basic routines of the addition, subtraction, multiplication, division and square root operation in the multiple-precision arithmetic on distributed memory parallel computers.

The presented multiple-precision parallel arithmetic algorithms make it possible to compute more than 137 billion decimal digits of $\sqrt{2}$ and more than 51.5 billion decimal digits of π computed on the distributed memory parallel computer, HITACHI SR2201 (1024 PEs, total main memory 256 GB).

Contributions by this thesis are summarized as follows:

- The conventional FFT-based algorithms multiply two n-digit numbers to obtain a $2n$-digit result. In the multiple-precision floating point multiplication, we need only the returned result whose precision is equal to the multiple-precision floating point number. This fact is exploited in our "dividing method" which is faster than the conventional FFT-based multiplication algorithm for the multiple-precision floating point numbers. According to the experimental results of the multiple-precision multiplication and square operations on cache effective processors, timings were about 1/1.91 and 1/1.51 compared to the conventional method, respectively. These results show that the overall arithmetic operations can be reduced and the cache miss is easily reduced by dividing the multiple-precision number.

- For an arbitrary-precision FFT-based multiplication, the number of points N in FFT is not necessarily 2^m. This thesis presented the radix-2, 3 and 5 parallel 1-D FFT algorithms on distributed memory parallel computers. In our parallel FFT algorithms, since we use cyclic distribution, all-to-all communication takes place only once. Moreover, the input

data and output data are both in natural order. We were able to show that the suitability of the parallel FFT algorithm depends on the CPU architecture of the processing elements of parallel computers. It was found that the four-step FFT-based parallel FFT algorithm is suitable for vector-parallel architectures and the six-step FFT-based parallel FFT algorithm is suitable for cache-based RISC processor processing elements. Our algorithms have resulted in high performance 1-D parallel complex FFTs suitable for distributed memory parallel computers. We succeeded to attain performances of about 130 GFLOPS on the 1024 PEs of HITACHI SR2201 and about 1.25 GFLOPS on the 32 PEs of IBM SP2. These parallel FFT algorithms are not only efficient for computing multiple-precision multiplication but also quite useful for other numerical computations.

- A key operation in the fast multiple-precision arithmetic is the multiplication, by which significant time in the total computation is spent. A parallel implementation of the real FFT-based multiplication has been presented, because the *floating point real* FFT is faster than the FNT (Fermat number transform) for the latest distributed memory parallel computers. By using the radix-2, 3 and 5 parallel 1-D FFT, we can reduce the arithmetic operations and memory size of the arbitrary-precision FFT-based parallel multiplication. In particular, we can use the radix-2, 3 and 5 parallel FFT-based multiplication for computing more than 51.5 billion ($\approx 3 \times 2^{34}$) decimal digits of π.

- The arithmetic operation counts for n-digit multiple-precision sequential addition, subtraction and multiplication by single-precision integer is clearly $O(n)$. However, a major factor to obstruct parallelization is releasing the carries and borrows in the multiple-precision addition, subtraction and multiplication by single-precision integer. This thesis proposed a parallelization of releasing these propagation operations by using the carry skip method. Similarly to the multiple-precision addition and subtraction, a part of normalization of results in the multiple-precision multiplication can be parallelized. It is concluded that the carry skip method is quite efficient for parallelizing the normalization of the multiple-precision addition, subtraction and multiplication.

- In the parallel implementation of the Newton iteration based multiple-precision division and square root operation, there is a trade-off between load balance and communication overhead on distributed memory parallel computers. This is because the Newton iteration is performed by doubling the precision for each iteration. It was found that the cyclic distribution with realignment is quite efficient for the Newton iteration based multiple-precision division and square root operation.

- This thesis has been presented a multiple-precision parallel division by single-precision integer, which is much faster than the multiple-precision division by a multiple-precision

number. In particular, when a radix b is multiple of a divisor v, the arithmetic operation of the n-digit multiple-precision division by single-precision integer is $O(n/P)$ on parallel computers which have P processors. When a radix b is not multiple of a divisor v, an upper bound of the arithmetic operation of this algorithm is $O((n/P)\log n)$. It is concluded that the multiple-precision parallel division by single-precision integer can be derived from the first-order recurrence which is parallelized by the parallel cyclic reduction method.

- Improvements of the Gauss-Legendre algorithm and Borweins' quartically convergent algorithm for π calculation are proposed. The improved Gauss-Legendre algorithm is up to 1.08 times faster than the original Gauss-Legendre algorithm, and the improved Borweins' quartically convergent algorithm is up to 1.78 times faster than the original Borweins' quartically convergent algorithm. We can conclude that these improved algorithms are quite efficient for computing highly accurate π.

These results have contributed to an innovation for computing highly accurate mathematical constants. In 1995, more than 6.4 billion decimal digits of π were computed on a vector supercomputer HITAC S-3800/480 (32 GFLOPS peak performance in which 16 GFLOPS were used) within elapsed time of 116 hours and 38 minutes. However, by using our multiple-precision parallel arithmetic algorithms and the distributed memory parallel computer HITACHI SR2201 (1024 PEs, 307.2 GFLOPS peak performance), more than 6.4 billion decimal digits of π are computed within only 1 hour 30 minutes. It follows from this that our multiple-precision parallel arithmetic algorithms are quite efficient for computing highly accurate mathematical constants.

As Table 9.1 shows, it took 12 years for extending the length of known π value from 100,000 to 1,000,000, 10 years from 1,000,000 to 10,000,000, 4 years from 10,000,000 to 100,000,000, 2 years from 100,000,000 to 1,000,000,000 and 8 years from 1,000,000,000 to the order of 10,000,000,000.

A computation of π to up 100,000,000,000 decimal digits will not be difficult within the 20th century, if we consider the trend of the program for the parallel computers.

Bibliography

[1] R. C. Agarwal and C. S. Burrus, *Fast Convolution Using Fermat Number Transforms with Applications to Digital Filtering*, IEEE Trans. Acoust., Speech, Signal Processing, ASSP-22 (1974), pp. 87–97.

[2] ——, *Number Theoretic Transforms to Implement Fast Digital Convolution*, in Proc. IEEE, vol. 63, 1975, pp. 550–560.

[3] R. C. Agarwal and J. W. Cooley, *Vectorized Mixed Radix Discrete Fourier Transform Algorithms*, in Proc. IEEE, vol. 75, 1987, pp. 1283–1292.

[4] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *A High Performance Parallel Algorithm for 1-D FFT*, in Proc. Supercomputing '94, 1994, pp. 34–40.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[6] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan, *A Parallel FFT on a MIMD Machine*, Parallel Computing, 15 (1990), pp. 61–74.

[7] D. H. Bailey, *Numerical Results on the Transcendence of Constants Involving π, e, and Euler's Constant*, Math. Comp., 50 (1987), pp. 275–281.

[8] ——, *The Computation of π to 29,360,000 Decimal Digits Using Borweins' Quartically Convergent Algorithm*, Math. Comp., 50 (1988), pp. 283–296.

[9] ——, *A Portable High Performance Multiprecision Package*, NASA Ames RNR Technical Report, RNR-90-022, NASA Ames Research Center, Moffett Field, CA 94035, 1990.

[10] ——, *FFTs in External or Hierarchical Memory*, J. Supercomputing, 4 (1990), pp. 23–35.

[11] ——, *Algorithm 719: Multiprecision Translation and Execution of FORTRAN Programs*, ACM Trans. Math. Softw., 19 (1993), pp. 288–319.

[12] ——, *A Fortran 90-Based Multiprecision System*, ACM Trans. Math. Softw., 21 (1995), pp. 379–387.

[13] L. Berggren, J. Borwein, and P. Borwein, eds., *Pi: A Source Book*, Springer-Verlag, New York, 1997.

[14] G. D. Bergland, *A Fast Fourier Transform Algorithm for Real-Valued Series*, Comm. ACM, 11 (1968), pp. 703–710.

[15] ——, *A Fast Fourier Transform Algorithm Using Base 8 Iterations*, Math. Comp., 22 (1968), pp. 275–279.

[16] D. Blatner, *The Joy of Pi*, Walkerbooks, New York, 1997.

[17] T. Boku, K. Itakura, H. Nakamura, and K. Nakazawa, *CP-PACS: A massively parallel processor for large scale scientific calculations*, in Proc. 1997 International Conference on Supercomputing, 1997, pp. 108–115.

[18] J. M. Borwein and P. B. Borwein, *Pi and the AGM — A Study in Analytic Number Theory and Computational Complexity*, Wiley, New York, 1987.

[19] J. M. Borwein, P. B. Borwein, and D. H. Bailey, *Ramanujan, Modular Equations, and Approximations to Pi or How to Compute One Billion Digits of Pi*, American Mathematical Monthly, 96 (1989), pp. 201–219.

[20] R. P. Brent, *Fast Multiple-Precision Evaluation of Elementary Functions*, J. ACM, 23 (1976), pp. 242–251.

[21] ——, *A Fortran Multiple-Precision Arithmetic Package*, ACM Trans. Math. Softw., 4 (1978), pp. 57–70.

[22] E. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[23] J. Brillhart, D. H. Lehmer, J. L. Selfridge, and B. Tuckerman, *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to High Powers*, American Mathematical Society, Rhode Island, 2nd ed., 1988.

[24] D. A. Buell and R. L. Ward, *A Multiprecise Integer Arithmetic Package*, J. Supercomputing, 3 (1989), pp. 89–107.

[25] G. Cesari and R. Maeder, *Performance Analysis of the Parallel Karatsuba Multiplication Algorithm for Distributed Memory Architectures*, J. Symbolic Computation, 21 (1996), pp. 467–473.

[26] B. Char, J. Johnson, D. Saunders, and A. P. Wack, *Some Experiments with Parallel Bignum Arithmetic*, in Proc. 1st International Symposium on Parallel Symbolic Computation, 1994, pp. 94–103.

[27] J. W. Cooley and J. W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. Comp., 19 (1965), pp. 297–301.

[28] R. Coustal, *Calcul de $\sqrt{2}$ et réflexion sur une espérance*, C. R. Acad. Sci. Paris, 230 (1950), pp. 431–432. MR 11, 402.

[29] R. Crandall and B. Fagin, *Discrete Weighted Transforms and Large-Integer Arithmetic*, Math. Comp., 62 (1994), pp. 305–324.

[30] R. E. Crandall, *Topics in Advanced Scientific Computation*, TELOS/Springer-Verlag, New York, 1995.

[31] J. H. Davenport, Y. Siret, and E. Tournier, *Computer Algebra: Systems and Algorithms for Algebraic Computation*, Academic Press, 2nd ed., 1993.

[32] J. Dutka, *The Square Root of 2 to 1,000,000 Decimals*, Math. Comp., 25 (1971), pp. 927–930.

[33] B. S. Fagin, *Large Integer Multiplication on Massively Parallel Processors*, in Proc. Third Symposium on the Frontiers of Massively Parallel Computation, 1990, pp. 38–42.

[34] ——, *Fast Addition of Large Integers*, IEEE Trans. Comput., 41 (1992), pp. 1069–1077.

[35] ——, *Large Integer Multiplication on Hypercubes*, J. Parallel and Distributed Computing, 14 (1992), pp. 426–430.

[36] T. Granlund, *GNU MP: The GNU Multiple Precision Arithmetic Library*, Free Software Foundation, 1991.

[37] R. Grossman, ed., *Symbolic Computation: Applications to Scientific Computing*, SIAM Press, Philadelphia, PA, 1989.

[38] M. Hegland, *Real and Complex Fast Fourier Transforms on the Fujitsu VPP 500*, Parallel Computing, 22 (1996), pp. 539–553.

[39] R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam-Hilger, Bristol, 1981.

[40] S. L. Johnsson and R. L. Krawitz, *Cooley-Tukey FFT on the Connection Machine*, Parallel Computing, 18 (1992), pp. 1201–1221.

[41] Y. Kanada, *Vectorization of Multiple-Precision Arithmetic Program and 201,326,000 Decimal Digits of π Calculation*, in Proc. Supercomputing '88, vol. 2, 1988, pp. 117–128.

[42] Y. Kanada, Y. Tamura, S. Yoshino, and Y. Ushiro, *Calculation of π to 10,013,395 Decimal Places Based on the Gauss-Legendre Algorithm and Gauss Arctangent Relation.* CCUT-TR-84-01, Computer Centre, University of Tokyo, Bunkyo-ku, Yayoi 2-11-16, Tokyo 113, Japan, 1983.

[43] A. Karatsuba and Y. Ofman, *Multiplication of multidigit numbers on automata*, Doklady Akad. Nauk SSSR, 145 (1962), pp. 293–294.

[44] A. H. Karp and P. Markstein, *High-Precision Division and Square Root*, ACM Trans. Math. Softw., 23 (1997), pp. 561–589.

[45] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 3rd ed., 1997.

[46] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, New York, 2nd ed., 1994.

[47] W. Krandick and J. R. Johnson, *Efficient Multiprecision Floating Point Multiplication with Exact Rounding*, Technical Report RISC-Linz Report Series Number 93-76, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.

[48] ——, *Efficient Multiprecision Floating Point Multiplication with Optimal Directional Rounding*, in Proc. 11th IEEE Symposium on Computer Arithmetic, 1993, pp. 228–233.

[49] M. Lal, *Expansion of $\sqrt{2}$ to 19600 Decimals*, Math. Comp., 21 (1967), pp. 258–259.

[50] ——, *Expansion of $\sqrt{2}$ to 100,000 Decimals*, Math. Comp., 22 (1968), pp. 899–900.

[51] ——, *First 39000 Decimal Digits of $\sqrt{2}$*, Math. Comp., 22 (1968), p. 226.

[52] M. Lehman and N. Burla, *Skip Techniques for High-Speed Carry Propagation in Binary Arithmetic Units*, IRE Trans. Elec. Comput., EC-10 (1961), pp. 691–698.

[53] D. H. Lehmer, *On Lucas's test for the primality of Mersenne's numbers*, J. London Math. Soc., 10 (1935), pp. 162–165.

[54] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, 1995.

[55] C. J. Mifsud, *A Multiple-Precision Division Algorithm*, Comm. ACM, 13 (1970), pp. 666–668.

[56] T. Mulders, *On Computing Short Products*, Technical Report No. 276, Department of Computer Science, ETH Zurich, 1997.

[57] K. NAKAZAWA, H. NAKAMURA, H. IMORI, AND S. KAWABE, *Pseudo Vector Processor based on Register-Windowed Superscalar Pipeline*, in Proc. Supercomputing '92, 1992, pp. 642–651.

[58] H. J. NUSSBAUMER, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, New York, second corrected and updated ed., 1982.

[59] C. M. RADER, *Discrete Fourier transforms when the number of data samples is prime*, in Proc. IEEE, vol. 56, 1968, pp. 1107–1108.

[60] ——, *Discrete Convolutions via Mersenne Transforms*, IEEE Trans. Comput., C-21 (1972), pp. 1269–1273.

[61] G. W. REITWIESNER, *An ENIAC Determination of π and e to more than 2000 Decimal Places*, Mathematical Tables and Other Aids to Computation, 4 (1950), pp. 11–15.

[62] P. RIBENBOIM, *The Little Book of Big Primes*, Springer-Verlag, New York, 1991.

[63] H. RIESEL, *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, 2nd ed., 1994.

[64] R. L. RIVEST, A. SHAMIR, AND L. ADLEMAN, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Comm. ACM, 21 (1978), pp. 120–126.

[65] E. SALAMIN, *Computation of π Using Arithmetic-Geometric Mean*, Math. Comp., 30 (1976), pp. 565–570.

[66] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Zahlen*, Computing (Arch. Elektron. Rechnen), 7 (1971), pp. 281–292.

[67] D. SHANKS AND J. W. WRENCH, JR., *Calculation of π to 100,000 Decimals*, Math. Comp., 16 (1962), pp. 76–99.

[68] R. C. SINGLETON, *An Algorithm for Computing the Mixed Radix Fast Fourier Transform*, IEEE Trans. Audio Electroacoust., 17 (1969), pp. 93–103.

[69] D. M. SMITH, *Algorithm 693: A FORTRAN Package for Floating-Point Multiple-Precision Arithmetic*, ACM Trans. Math. Softw., 17 (1991), pp. 273–283.

[70] ——, *A Multiple-Precision Division Algorithm*, Math. Comp., 65 (1996), pp. 157–163.

[71] M. L. STEIN, *Divide-and-Correct Methods for Multiple Precision Division*, Comm. ACM, 7 (1964), pp. 472–474.

[72] P. N. SWARZTRAUBER, *FFT Algorithms for Vector Computers*, Parallel Computing, 1 (1984), pp. 45–63.

[73] ——, *Multiprocessor FFTs*, Parallel Computing, 5 (1987), pp. 197–210.

[74] D. TAKAHASHI AND Y. KANADA, *Fast High-Precision Arithmetic on Distributed Memory Parallel Machines*, in Proc. Ninth SIAM Conference on Parallel Processing for Scientific Computing. (to appear).

[75] ——, *Fast Multiple-Precision Calculation on Distributed Memory Parallel Computers*, in IPSJ SIG Notes, 96-HPC-60, Information Processing Society of Japan, 1996, pp. 31–36. (in Japanese).

[76] ——, *π — Fast Calculation and Statistical Testing (3)*, in Proc. 37th Programming Symposium IPSJ, Information Processing Society of Japan, 1996, pp. 73–84. (in Japanese).

[77] ——, *Improvement of Algorithms for π Calculation: the Gauss-Legendre Algorithm and the Borwein's Quartically Convergent Algorithm*, Trans. IPS. Japan, 38 (1997), pp. 2406–2409. (in Japanese).

[78] ——, *Calculation of π to 51.5 Billion Decimal Digits on Distributed Memory Parallel Processors*, Trans. IPS. Japan, 39 (1998), pp. 2074–2083. (in Japanese).

[79] ——, *Implementation and Evaluation of Radix-2, 3 and 5 1-D FFT on Distributed Memory Parallel Computers*, Trans. IPS. Japan, 39 (1998), pp. 519–528. (in Japanese).

[80] D. TAKAHASHI, Y. TORII, AND T. YUASA, *An Implementation of Factorization on Massively Parallel SIMD Computers*, Trans. IPS. Japan, 36 (1995), pp. 2521–2530. (in Japanese).

[81] K. TAKAHASHI AND M. SIBUYA, *The Decimal and Octal digits of $\sqrt{n}$*, Math. Comp., 21 (1967), pp. 259–260.

[82] Y. TAMURA AND Y. KANADA, *Calculation of π to 4,194,293 Decimals Based on the Gauss-Legendre Algorithm*, CCUT-TR-83-01, Computer Centre, University of Tokyo, Bunkyo-ku, Yayoi 2-11-16, Tokyo 113, Japan, 1983.

[83] C. TEMPERTON, *A Note on Prime Factor FFT Algorithms*, J. Comput. Phys., 52 (1983), pp. 198–204.

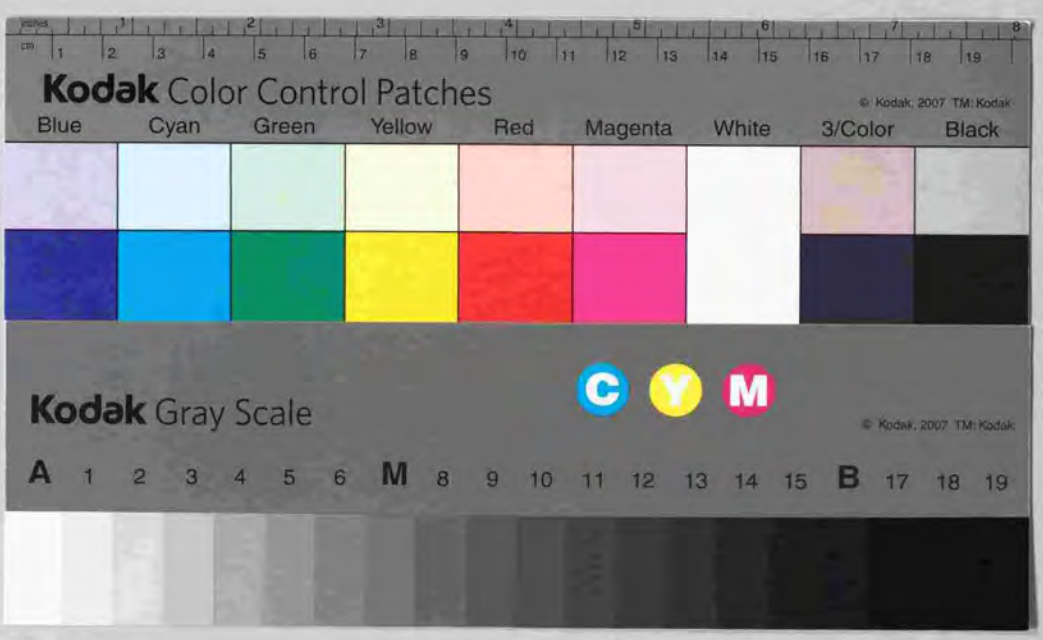
[84] ——, *Self-Sorting Mixed-Radix Fast Fourier Transforms*, J. Comput. Phys., 52 (1983), pp. 1–23.

[85] H. S. Uhler, *Many-figure approximations to $\sqrt{2}$, and distribution of digits in $\sqrt{2}$ and $1/\sqrt{2}$*, in Proc. Nat. Acad. Sci. U.S.A., vol. 37, 1951, pp. 63–67. MR 12, 444.

[86] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA, 1992.

[87] K. Weber, *An Experiment in High-precision Arithmetic on Shared Memory Multiprocessors*, SIGSAM Bulletin, 24 (1990), pp. 22–40.

[88] S. Winograd, *On Computing the Discrete Fourier Transform*, Math. Comp., 32 (1978), pp. 175–199.


Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19