Parallel Implementation of Multiple-Precision Arithmetic and $2,576,980,370,000$ Decimal Digits of π Calculation

Daisuke Takahashi

Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan

Abstract

We present efficient parallel algorithms for multiple-precision arithmetic operations of more than several million decimal digits on distributed-memory parallel computers. A parallel implementation of floating-point real FFT-based multiplication is used, since the key operation for fast multiple-precision arithmetic is multiplication. The operation for releasing propagated carries and borrows in multiple-precision addition, subtraction and multiplication was also parallelized. More than 2.576 trillion decimal digits of π were computed on 640 nodes of Appro Xtreme-X3 (648 nodes, 147.2 GFlops/node, 95.4 TFlops peak performance) with a computing elapsed time of 73 hours 36 minutes which includes the time required for verification.

Key words: multiple-precision arithmetic, fast Fourier transform, distributed-memory parallel computer

1 Introduction

Several software packages are available for multiple-precision computation [1–4]. At present, GNU MP [3] is probably the most widely used package due to its greater functionality and efficiency. Using GMP 4.2 with sufficient memory, it should be possible to compute up to 41 billion digits [3]. In 2009, Bellard computed π up to about 2.7 trillion digits in about 131 days using an Intel Core i7 PC [5]. However, parallel processing using a distributed-memory parallel computer is required to calculate further digits in a reasonable amount of time.

Email address: daisuke@cs.tsukuba.ac.jp (Daisuke Takahashi).

Parallel implementation of Karatsuba's multiplication algorithm [6,7] on a distributed-memory parallel computer was proposed [8]. Karatsuba's algorithm is known as the $O(n^{\log_2 3})$ multiplication algorithm.

However, 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 [9], which is based on the fast Fourier transform (FFT).

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

The Fermat number transform (FNT) for large integer multiplication was performed on the Connection Machine CM-2 [10]. On the other hand, the number theoretic transform (NTT) uses many modulo operations, which are slow due to the integer division process. Thus, *floating-point real* FFT-based multiplication was used for multiple-precision multiplication on distributed-memory parallel computers.

Parallel computation of $\sqrt{2}$ up to 1 million decimal digits was performed on a network of workstations [11]. However, multiple-precision parallel division was not presented in this paper, and a parallel version of Karatsuba's multiplication algorithm was used.

Section 2 describes the parallelization of multiple-precision addition, subtraction and multiplication. Section 3 describes the parallelization of multiple-precision division and square root operations. Section 4 presents the experimental results. Section 5 describes the calculation of π to $2,576,980,370,000$ decimal digits on a distributed-memory parallel computer. Finally, section 6 presents some concluding remarks.

2 Parallelization of Multiple-Precision Addition, Subtraction, and Multiplication

2.1 Parallelization of Multiple-Precision Addition, Subtraction

Let us consider an n-digit number X with radix-B.

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

where $0 \leq x_i < B$.

Algorithm 1 Sequential Addition

Input: $X = \sum_{i=0}^{n-1} x_i B^i,\ Y = \sum_{i=0}^{n-1} y_i B^i$
Output: $Z = X + Y := \sum_{i=0}^{n-1} z_i B^i$

```
1: c ← 0
2: for i = 0 to n − 2 do
3:     w ← x_i + y_i + c
4:     c ← w div B
5:     z_i ← w mod B
6: z_{n−1} ← x_{n−1} + y_{n−1} + c
7: return Z.
```

Fig. 1. Multiple-precision sequential addition

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

The arithmetic operation counts for n-digit multiple-precision sequential additions, subtractions is clearly $O(n)$. Also, the arithmetic operation counts for n-digit multiple-precision sequential multiplication by a single-precision integer is $O(n)$. However, releasing the carries and borrows in these operations is a major factor that can prevent parallelization.

For example, a pseudo code of multiple-precision sequential addition is shown in Fig. 1. Here, c is a variable to store the carry, w is a temporary variable and B is the radix of the multiple-precision numbers. We assume that the input data has been normalized from 0 to $B-1$ and is stored in arrays X and Y.

Since in this program, at line 3, the value of c is used recurrently to determine the value of w, the algorithm can not be parallelized because of data dependency.

Fig. 2 shows an algorithm that allows the parallelization of addition by creating a method for the releasing of the carries. We assume that the input data has been normalized from 0 to $B-1$ and is stored in arrays X and Y.

Multiple-precision addition without the propagation of carries is performed at lines 1 and 2. While the maximum value of z_i $(0 \leq i \leq n-2)$ is greater than or equal to B, the carries are computed in lines 5 and 6, then the carries are released in lines 7 to 9. The maximum value can be evaluated in parallel easily. Here, c_i $(0 \leq i \leq n-1)$ is a working array to store carries.

Algorithm 2 Parallel Addition

Input: $X = \sum_{i=0}^{n-1} x_i B^i,\ Y = \sum_{i=0}^{n-1} y_i B^i$
Output: $Z = X + Y := \sum_{i=0}^{n-1} z_i B^i$

```
 1: for i = 0 to n − 1 do in parallel
 2:     z_i ← x_i + y_i
 3: while max_{0≤i≤n−2}(z_i) ≥ B do
 4:     c_0 ← 0
 5:     for i = 0 to n − 2 do in parallel
 6:         c_{i+1} ← z_i div B
 7:     for i = 0 to n − 2 do in parallel
 8:         z_i ← (z_i mod B) + c_i
 9:     z_{n−1} ← z_{n−1} + c_{n−1}
10: return Z.
```

Fig. 2. Multiple-precision parallel addition

In line 6, communication is required to send the carries to the neighbor processor. The amount of communication is $O(1)$ in the block distribution, whereas the amount of communication is $O(n/P)$ in the cyclic distribution on parallel computers that have P processors.

In this algorithm, it is necessary to repeat the normalization until all carries have been released. For random inputs, this process is performed usually few times. When the propagation of carries repeats, as in the case of $0.99999999\cdots9 + 0.00000000\cdots1$, we have to use the carry skip method [12].

A pseudo code for multiple-precision parallel addition with the carry skip method is shown in Fig. 3. We assume that the input data has been normalized from 0 to $B-1$ and is stored in array X and Y. Here, c_i $(0 \le i \le n-1)$ is a working array to store carries.

Multiple-precision addition without the propagation of carries is performed at lines 1 and 2. Then, *incomplete* normalization is performed from 0 to B in lines 3 to 9. Note that the while loop in lines 3 to 9 is repeated at most twice. The range for carry skipping is determined in lines 11 to 22. Finally, carry skipping is performed in lines 23 to 26.

Because carry skips occur intermittently in the case of $0.998998998\cdots + 0.0100100100\cdots$, this is the one of the worst case of the carry skip method. For the worst case, the carry look-ahead method is effective. However, the carry look-ahead method requires $O(\log P)$ steps on parallel computers that have P processors. Since we assumed that such a worst case occur rarely, the carry skip method was used for multiple-precision parallel addition and subtraction.

Algorithm 3 Parallel Addition with Carry Skip Method

Input: $X = \sum_{i=0}^{n-1} x_i B^i$, $Y = \sum_{i=0}^{n-1} y_i B^i$
Output: $Z = X + Y := \sum_{i=0}^{n-1} z_i B^i$

1: **for** $i = 0$ to $n-1$ **do in parallel**
2: $z_i \leftarrow x_i + y_i$
3: **while** $\max_{0 \le i \le n-2}(z_i) > B$ **do**
4: $c_0 \leftarrow 0$
5: **for** $i = 0$ to $n-2$ **do in parallel**
6: $c_{i+1} \leftarrow z_i \text{ div } B$
7: **for** $i = 0$ to $n-2$ **do in parallel**
8: $z_i \leftarrow (z_i \bmod B) + c_i$
9: $z_{n-1} \leftarrow z_{n-1} + c_{n-1}$
10: **while** $\max_{0 \le i \le n-2}(z_i) = B$ **do**
11: **for** $i = 0$ to $n-1$ **do in parallel**
12: **if** $(z_i = B)$ **then**
13: $p_i \leftarrow i$
14: **else**
15: $p_i \leftarrow n-1$
16: $l \leftarrow \min_{0 \le i \le n-1}(p_i)$
17: **for** $i = l+1$ to $n-1$ **do in parallel**
18: **if** $(z_i < B-1)$ **then**
19: $p_i \leftarrow i$
20: **else**
21: $p_i \leftarrow n-1$
22: $m \leftarrow \min_{l+1 \le i \le n-1}(p_i)$
23: $z_l \leftarrow z_l - B$
24: **for** $i = l+1$ to $m-1$ **do in parallel**
25: $z_i \leftarrow z_i - (B-1)$
26: $z_m \leftarrow z_m + 1$
27: **return** Z.

Fig. 3. Multiple-precision parallel addition with the carry skip method

The carry skip method can be applied to the normalization process of multiple-precision parallel multiplication by a single-precision integer.

2.2 Parallelization of Multiple-Precision Multiplication

In this paper, we use multiple-precision multiplication based on the *floating-point real* FFT.

The main weakness of the floating-point FFT multiplication is that it is very difficult to guaranty the result due to rounding errors. If a 2^{20}-point FFT is performed, the maximal number of bits is 13 per IEEE 754 double-precision floating point number in worst case [13].

For *floating-point real* FFT-based multiplication, we can use the "balanced representation" [14], which tends to yield reduced errors for the convolutions we intended to perform. In this technique, an n-digit multiplicand $X = \sum_{i=0}^{n-1} x_i B^i$ with radix B is represented as follows:

$$X = x'_0 + x'_1 B + x'_2 B^2 + \cdots + x'_{n-1} B^{n-1},\ |x'_i| \leq \left\lfloor \frac{B}{2} \right\rfloor. \tag{2}$$

By using the "balanced representation", we expect the error to be reduced due to cancellation. Since each x_i may be random in the π computation, this is one of the best case in the floating-point FFT multiplication.

We checked whether the error for the convolutions is less than 0.1 in each FFT multiplication step. The criterion of 0.1 is sufficient enough in the actual $3 \times 5^2 \times 2^{33} \approx 644$ billion decimal digit FFT-based multiplication when four digits are stored in each double-precision floating-point number. To guaranty the correctness of the results, we computed π using the improved Gauss-Legendre algorithm and verified using the improved Borweins' quartically convergent algorithm.

A key operation in FFT-based multiple-precision parallel multiplication is the FFT and normalization, on which a significant proportion of the total computation time is spent. An efficient parallel FFT algorithm can be used for the FFT-based multiple-precision parallel multiplication.

We used a block nine-step FFT-based parallel one-dimensional FFT algorithm [15], which is extended from the six-step FFT algorithm [16]. The block nine-step FFT algorithm improves performance by utilizing the cache memory effectively. Although this FFT algorithm performs an n-point complex FFT, a result of $2n$-point real FFT can be easily from the n-point complex FFT.

The normalization of results in FFT-based multiple-precision parallel multiplication is essentially the same as the parallel processing of carries in multiple-precision addition. Thus, the normalization can also be parallelized.

3 Parallelization of Multiple-Precision Division and Square Root Operations

The computation time of multiple-precision division and square root operations is considerably longer than that of addition, subtraction, or multiplication. There are a number of methods to perform division and square root operations [17,7]. It is well known that multiple-precision division and square root operations can be reduced to multiple-precision addition, subtraction, and multiplication by using the Newton iteration [7].

3.1 Newton Iteration

We denote the number of operations used to multiply two n-digit numbers as $M(n)$, and the number of operations used to square an n-digit as $S(n)$. The Newton iteration requires $O(M(n))$ operations to perform n-digit division and square root operations.

In FFT-based multiplication and square operations, computation of FFTs consumes the most computation time. Although three FFTs are needed to multiply two n-digit numbers, only two FFTs are needed to square an n-digit number. Thus, we assume $S(n) \approx (2/3)M(n)$ in FFT-based multiplication and square operations.

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

$$x_{k+1} = x_k + x_k(1 - bx_k), \tag{3}$$

where the multiplication of x_k and $(1 - bx_k)$ can be performed with only half of the precision of x_{k+1} [2].

The final iteration is performed as follows [17]:

$$a/b \approx (ax_k) + x_k(a - b(ax_k)), \tag{4}$$

where the multiplication of a and x_k, and the multiplication of x_k and $(a - b(ax_k))$ can be performed with only half of the final level of precision. This scheme requires $(7/2)M(n)$ operations if we do not reuse the results of FFTs.

Square roots are computed by the following Newton iteration that converges

to $1/\sqrt{a}$:

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - ax_k^2), \tag{5}$$

where the multiplication of x_k and $(1 - ax_k^2)/2$ can be performed with only half of the precision of x_{k+1}.

The final iteration is performed as follows [17]:

$$\sqrt{a} \approx (ax_k) + \frac{x_k}{2}(a - (ax_k)^2), \tag{6}$$

where the multiplication of a and x_k, and the multiplication of x_k and $(a - (ax_k)^2)/2$ can be performed with only half of the final level of precision. This scheme requires $(19/6)M(n)$ operations if we do not reuse the results of FFTs.

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

3.2 Parallelization

To perform the Newton iteration in parallel, it is necessary to parallelize the multiple-precision parallel addition, subtraction and multiplication. For these operations, the parallel algorithms given in section 2 can be applied.

In this section, the distribution method of the data to each processor is considered. Since the number of calculated digits can be doubled by the Newton iteration for the division and square root operations, the computational volume changes gradually. For example, in the first iteration, both additions and multiplications can be performed with double-precision. Then, the second iteration can be performed with quad-precision. In the block distribution, processor 0 has the first $\lceil n/P \rceil$-digit multiple-precision numbers. Thus, in the first several iterations, processors except the processor 0 are idling. Only the final iteration is performed with full-precision on all P processors. This causes a load imbalance in computation and communication.

In the block distribution, because of the load imbalance, the parallel computation time for the operations of the multiple-precision parallel division and square root operations is $O((M(n)/P) \log P)$ [18].

On the other hand, in the cyclic distribution, the parallel computation time for operations of the multiple-precision parallel division and square root operations is $O(M(n)/P)$ [18].

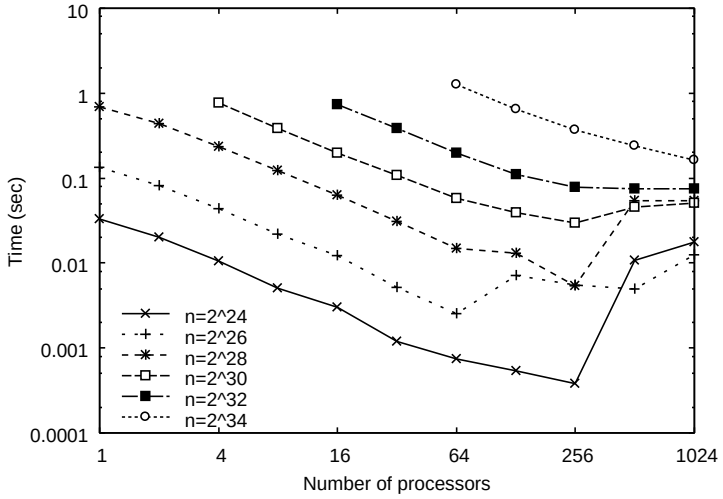

Fig. 4. Execution time for multiple-precision parallel addition ($\pi+\sqrt{2}$), n = number of decimal digits

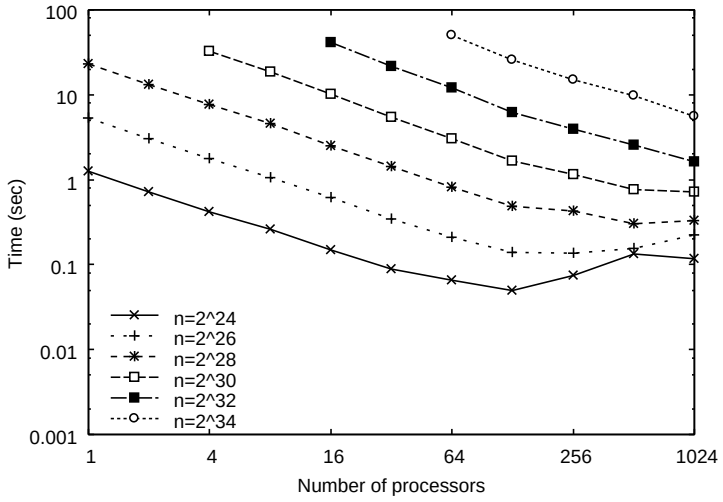

Fig. 5. Execution time for multiple-precision parallel multiplication ($\pi \times \sqrt{2}$), n = number of decimal digits

Thus, the parallel computation time of the cyclic distribution is less than that of the block distribution. However, the cyclic distribution has a large amount of communication for the normalization process described in section 2. Although there is a trade-off between the load imbalance and the communication overhead, the effect of the load imbalance is larger than the communication overhead.

Thus, the cyclic distribution was used for multiple-precision parallel arithmetic.

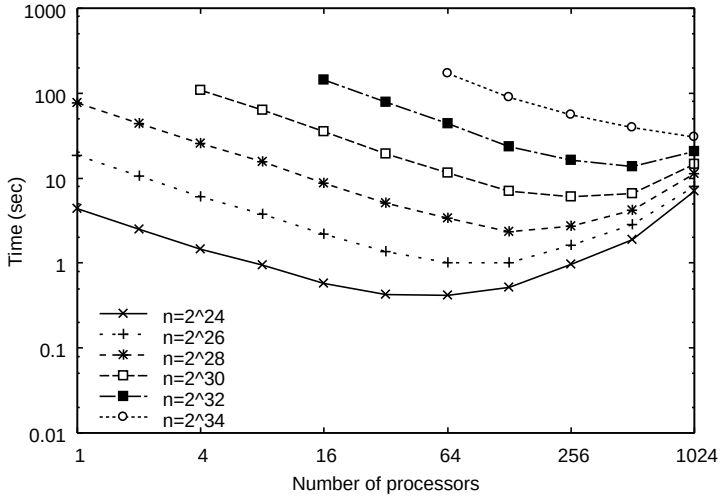

Fig. 6. Execution time for multiple-precision parallel division ($\sqrt{2}/\pi$), n = number of decimal digits

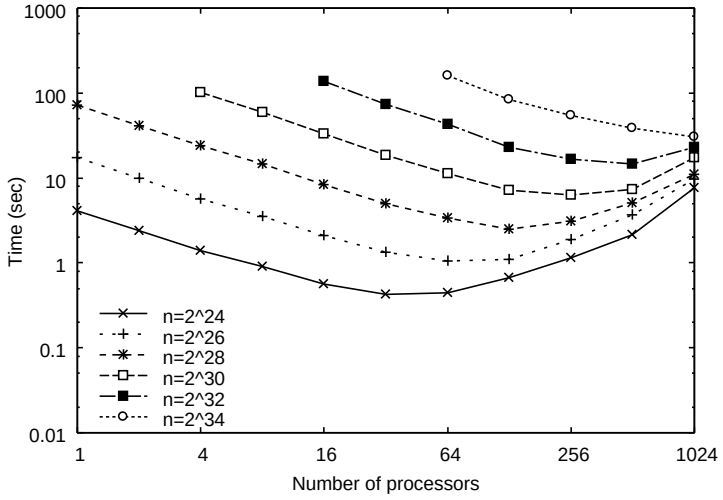

Fig. 7. Execution time for multiple-precision parallel square root ($\sqrt{\pi}$), n = number of decimal digits

4 Experimental Results

In order to evaluate the multiple-precision parallel arithmetic algorithms, the number of 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 ($\pi+\sqrt{2}$), multiplication ($\pi\times\sqrt{2}$), division ($\sqrt{2}/\pi$) and square root ($\sqrt{\pi}$). We note that the respective values of n-digit π and $\sqrt{2}$ were prepared in advance. The selection of these values has no particular significance here, but was convenient to establish definite test cases, the results of which were used as randomized test data.

An Appro Xtreme-X3 (648 nodes, 32 GB per node, 147.2 GFlops per node, 20

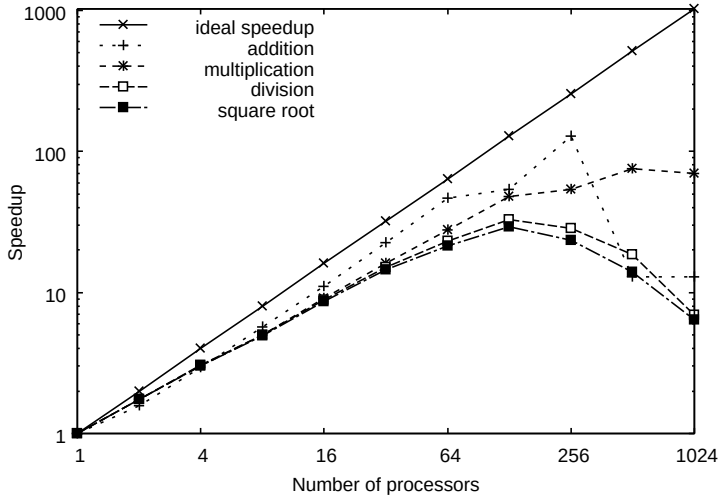

Fig. 8. Speedup for multiple-precision parallel addition, multiplication, division and square root, $n = 2^{28}$ decimal digits

TB total main memory size, communication bandwidth 8 GB/sec per node, and 95.4 TFlops peak performance) was used as the distributed-memory parallel computer.

Each computation node of the Xtreme-X3 is equipped with a 4-socket of quad-core AMD Opteron (2.3 GHz) in a 16-core shared-memory configuration with 147.2 GFlops of performance.

In the experiment, 1 processor (4 cores) to 1,024 processors (4,096 cores) were used. The original program was written in FORTRAN 77 with MPI and OpenMP. Open MPI 1.3 [19] was used as the communication library. OpenMP was used to parallelize intra-socket processing, and MPI was used to communicate between sockets.

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

Fig. 4 shows the averaged execution time for multiple-precision parallel addition ($\pi+\sqrt{2}$). For $n = 2^{24}$ and $P > 256$, we can clearly see that communication overhead dominates the execution time. This can be attributed to the fact that the arithmetic operation count for n-digit multiple-precision parallel addition is only $O(n/P)$.

Fig. 5 shows the averaged execution time for multiple-precision parallel multiplication ($\pi \times \sqrt{2}$).

Fig. 5 shows that multiple-precision parallel multiplication is scalable for $n > 2^{30}$ on 1,024 processors.

Figs. 6 and 7 show the averaged execution time for multiple-precision parallel division ($\sqrt{2}/\pi$) and square root operations ($\sqrt{\pi}$). For $n = 2^{24}$ and $P > 64$, we can clearly see that communication overhead dominates the execution time. This is because that the division and square root operations include small digit additions and multiplications.

Fig. 8 shows the speedup for multiple-precision parallel addition ($\pi + \sqrt{2}$), multiplication ($\pi \times \sqrt{2}$), division ($\sqrt{2}/\pi$) and square root operations ($\sqrt{\pi}$) of $n = 2^{28}$ decimal digits. In the "strong scaling" case, the multiple-precision parallel arithmetic does not scale well for large number of processors.

5 The Calculation of $2,576,980,370,000$ Decimal Digits of π

The high precision computation of π has a long history, and many computations have been performed [20]. The development of new programs suited to the calculation of π and high speed computers with a large memory have revealed interesting information about this ubiquitous number. In 2002, Kanada et al. computed π to over 1.241 trillion decimal digits by using the arctangent formulae [21].

We have computed π to more than 2.576 trillion decimal digits by using the formula of the improved Gauss-Legendre algorithm and verified the results through improved Borweins' quartically convergent algorithm for π on the Appro Xtreme-X3 supercomputer at the Center for Computational Sciences, University of Tsukuba.

5.1 Algorithms for π

5.1.1 The Gauss-Legendre Algorithm: Main Algorithm

The theoretical basis for the Gauss-Legendre algorithm for π is explained in the references [22–24].

The following improved Gauss-Legendre algorithm for π was used [25]:

$A := 1;\ B := 1/2;\ T := 1/2;\ X := 2;$
while $A - B > 2^{-n}$ **do begin**
    $W := A * B;\ A := (A + B)/2;$
    $B := \sqrt{W};\ A := (A + B)/2;$
    $T := T - X * (A - B);\ X := 2 * X$

end;
return $(A+B)/T$.

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

At each iteration, multiple-precision multiplication of once can be reduced by improving the original Gauss-Legendre algorithm.

5.1.2 Borweins' Quartically Convergent Algorithm: Verification Algorithm

Borweins' quartically convergent algorithm [24] can be explained as follows: Let $a_0 = 6 - 4\sqrt{2}$ and $y_0 = \sqrt{2} - 1$. Iterate the following calculations:

$$y_{k+1} = \frac{1 - (1 - y_k^4)^{1/4}}{1 + (1 - y_k^4)^{1/4}}, \tag{7}$$

$$a_{k+1} = a_k(1 + y_{k+1})^4 - 2^{2k+3} y_{k+1}(1 + y_{k+1} + y_{k+1}^2). \tag{8}$$

Then a_k converges quartically to $1/\pi$. Here, the precisions for a_k and y_k must be more than the desired number of digits.

The following improved Borweins' quartically convergent algorithm for π was used:

$A := 6 - 4\sqrt{2}$; $Y := 17 - 12\sqrt{2}$; $X := 2$;
while $Y > 2^{-n/4}$ **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 $Y < 2^{-n/2}$ **then begin**
$W := 1 + Y/2$; $A := A * W - X * (Y/8)$
end
else if $Y < 2^{-n/3}$ **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.

At each iteration, four multiple-precision multiplications and three square operations can be reduced by improving the original Borweins' quartically convergent algorithm.

5.2 Layout of Storage

A multiple-precision number is stored with cyclic distribution in the array of 32-bit integers. A data format of the multiple-precision number is the non-negative fixed-point representation. The radix selected for the multiple-precision numbers is 10^8. For example, the value of π is stored as `x[0] = 314159265`, `x[1] = 35897932`, and so on. In this format, `x[0]` is allowed to exceed 10^8 exceptionally. Thus, the range of the fixed-point representation is from 0 to $(2^{31} - 1)/10^8 + 0.999\cdots = 21.47483647999\cdots$. This range is enough to compute the value of π.

Each input data word is split into two words upon entry to the FFT-based multiplication. This means four digits are stored in each 64-bit floating-point number for the FFT-based multiplication.

To perform FFT-based multiplication of $3 \times 5^2 \times 2^{35} \approx 2.576$ trillion decimal digit numbers, at least 31.6 TB of main memory should be available, as shown in Table 1.

In addition, several working arrays are required to compute the division and the square root. Thus, these schemes needed approximately 33.4 TB of main memory for the Gauss-Legendre algorithm and approximately 32.8 TB of main memory for Borweins' quartically convergent algorithm.

It was impossible to obtain 2.576 trillion decimal digits through in-core (on main memory) operations because of the 17.5 TB main memory limit on 640 nodes of Appro Xtreme-X3 supercomputer.

Thus, $3 \times 5^2 \times 2^{32}$-point real FFT for $3 \times 5^2 \times 2^{33} \approx 644$ billion decimal digit multiplications was performed on the main memory. Then, Karatsuba's multiplication algorithm was recursively used twice to obtain $3 \times 5^2 \times 2^{35} \approx$

Table 1
Memory size and computation time ratio for $3 \times 5^2 \times 2^{35} \approx 2.576$ trillion decimal digits multiplication ($Z = X \times Y$) with combination of FFT-based multiplication and Karatsuba's multiplication.

Decimal digit of FFT multiplication		$3 \times 5^2 \times 2^{32}$	$3 \times 5^2 \times 2^{33}$	$3 \times 5^2 \times 2^{34}$	$3 \times 5^2 \times 2^{35}$
memory size (TB)	inputs X and Y	2.344	2.344	2.344	2.344
	output Z	1.172	1.172	1.172	1.172
	FFT work	3.516	7.031	14.063	28.125
	Karatsuba work	1.465	1.172	0.586	0
	Total	8.496	11.719	18.164	31.641
Ratio of computation time		3.138	2.145	1.465	1.000

2.576 trillion decimal digit multiplications. The combination of FFT-based multiplication and Karatsuba's multiplication requires approximately 11.7 TB of main memory, as shown in Table 1. These schemes needed approximately 13.5 TB of main memory for the Gauss-Legendre algorithm and 12.9 TB of main memory for Borweins' quartically convergent algorithm.

In order to increase speed or reduce elapsed time, the amount of main memory available is crucial. To perform $N = 3 \times 5^2 \times 2^m$ point FFTs, we used a radix-2, 3 and 5 parallel FFT, named FFTE [26] written by the author.

5.3 Results of Calculating $2,576,980,370,000$ *Decimal Digits of* π

The calculations of π by Gauss-Legendre algorithm and Borweins' quartically convergent algorithm were performed on 640 nodes of the Appro Xtreme-X3 supercomputer.

The original program was written in FORTRAN 77 with MPI and OpenMP. Open MPI 1.3 [19] was used as the communication library. OpenMP was used to parallelize intra-socket processing, and MPI was used to communicate between sockets. The main program and the verification program were run on 2,560 MPI processes, i.e. each node has 4 MPI processes.

Main program run:

Job start	: 9th April 2009 07:37:32 (JST)
Job end	: 10th April 2009 12:43:21 (JST)
Total elapsed time	: 29 hours 5 minutes
Main memory	: 13.5 TB
Algorithm	: Gauss-Legendre algorithm

Verification program run:

Job start	: 27th April 2009 21:35:36 (JST)
Job end	: 29th April 2009 18:06:09 (JST)
Total elapsed time	: 44 hours 30 minutes
Main memory	: 12.9 TB
Algorithm	: Borweins' quartically convergent algorithm

The decimal numbers of π and $1/\pi$ from the $2,576,980,369,951$-st to the $2,576,980,370,000$-th digits are:

π : `3616276346 5152343138 0598550567 3249553206 9855284552`

$1/\pi$: `1787760186 8492477551 0458174294 3077949861 4582392945`.

Furthermore, Tables 2 and 3 give the distribution of the digits of π and $1/\pi$ from 0 to 9 up to the $2,500,000,000,000$-th decimal digit.

The main computation took 40 iterations of the improved Gauss-Legendre algorithm for π, to yield $3 \times 5^2 \times 2^{35} = 2,576,980,377,600$ digits of π. This computation was checked with 20 iterations of improved Borweins' quartically convergent algorithm for $1/\pi$, followed by a reciprocal operation.

A comparison of these output results gave no discrepancies except for the last 76 digits due to the normal truncation errors.

6 Conclusion

We presented efficient parallel algorithms for multiple-precision arithmetic operations of more than several million decimal digits on distributed-memory parallel computers. The operation for releasing propagated carries and bor-

Table 2
Frequency distribution for $\pi - 3$ up to the $2,500,000,000,000$-th decimal digit.

Digit	Count
0	249999192826
1	249999959334
2	250000751269
3	249999904969
4	250000455856
5	249999721513
6	249999564178
7	249999660121
8	250001040584
9	249999749350

Table 3
Frequency distribution for $1/\pi$ up to the $2,500,000,000,000$-th decimal digit.

Digit	Count
0	249999622924
1	250000603011
2	249999024748
3	249999886945
4	250000566113
5	250000389148
6	250000066227
7	249999751301
8	250000370807
9	249999718776

rows in multiple-precision addition, subtraction and multiplication was parallelized using the carry skip method. Similar to multiple-precision addition and subtraction, some part of the normalization of results in the multiple-precision multiplication can be parallelized. It was concluded that the carry skip method is quite efficient for parallelizing normalization of multiple-precision addition, subtraction and multiplication.

Thus, the presented multiple-precision parallel arithmetic algorithms allow for more than 2.576 trillion decimal digits of π to be calculated on 640 nodes of

Appro Xtreme-X3 (648 nodes, 147.2 GFlops/node, 95.4 TFlops peak performance).

References

[1] D. M. Smith, Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic, ACM Trans. Math. Softw. 17 (1991) 273–283.

[2] D. H. Bailey, Algorithm 719: Multiprecision translation and execution of FORTRAN programs, ACM Trans. Math. Softw. 19 (1993) 288–319.

[3] The GNU MP Bignum Library, http://gmplib.org/.

[4] The MPFR Library, http://www.mpfr.org/.

[5] F. Bellard, Computation of 2700 billion decimal digits of pi using a desktop computer (2010).

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

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

[8] G. Cesari, R. Maeder, Performance analysis of the parallel Karatsuba multiplication algorithm for distributed memory architectures, Journal of Symbolic Computation 21 (1996) 467–473.

[9] A. Schönhage, V. Strassen, Schnelle Multiplikation grosser Zahlen, Computing (Arch. Elektron. Rechnen) 7 (1971) 281–292.

[10] B. S. Fagin, Large integer multiplication on hypercubes, Journal of Parallel and Distributed Computing 14 (1992) 426–430.

[11] B. Char, J. Johnson, D. Saunders, A. P. Wack, Some experiments with parallel bignum arithmetic, in: Proc. 1st International Symposium on Parallel Symbolic Computation, 1994, pp. 94–103.

[12] M. Lehman, N. Burla, Skip techniques for high-speed carry propagation in binary arithmetic units, IRE Trans. Elec. Comput. EC-10 (1961) 691–698.

[13] R. P. Brent, P. Zimmermann, Modern Computer Arithmetic, 2009, version 0.4.

[14] R. Crandall, B. Fagin, Discrete weighted transforms and large-integer arithmetic, Math. Comput. 62 (1994) 305–324.

[15] D. Takahashi, T. Boku, M. Sato, A blocking algorithm for parallel 1-D FFT on clusters of PCs, in: Proc. 8th International Euro-Par Conference (Euro-Par 2002), Vol. 2400 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 691–700.

[16] D. H. Bailey, FFTs in external or hierarchical memory, The Journal of Supercomputing 4 (1990) 23–35.

[17] A. H. Karp, P. Markstein, High-precision division and square root, ACM Trans. Math. Softw. 23 (1997) 561–589.

[18] D. Takahashi, Implementation of multiple-precision parallel division and square root on distributed-memory parallel computers, in: Proc. 2000 International Conference on Parallel Processing (ICPP-2000) Workshops, 2000, pp. 229–235.

[19] OpenMPI: Open Source High Performance Computing, `http://www.open-mpi.org/`.

[20] L. Berggren, J. Borwein, P. Borwein (Eds.), Pi: A Source Book, 3rd Edition, Springer-Verlag, New York, 2004.

[21] Kanada Laboratory home page, `http://www.super-computing.org/`.

[22] R. P. Brent, Fast multiple-precision evaluation of elementary functions, J. ACM 23 (1976) 242–251.

[23] E. Salamin, Computation of π using arithmetic-geometric mean, Math. Comput. 30 (1976) 565–570.

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

[25] T. Ooura, Improvement of the π calculation algorithm and implementation of fast multiple-precision computation, Transactions of the Japan Society for Industrial and Applied Mathematics 9 (1999) 165–172, (in Japanese).

[26] FFTE: A Fast Fourier Transform Package, `http://www.ffte.jp/`.