

Parallel Implementation of Multiple-Precision Arithmetic and 1, 649, 267, 440, 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
daisuke@cs.tsukuba.ac.jp

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 because a key operation in fast multiple-precision arithmetic is multiplication. We also parallelized an operation of releasing propagated carries and borrows in multiple-precision addition, subtraction and multiplication. More than 1.6 trillion decimal digits of π were computed on 256 nodes of Appro Xtreme-X3 (648 nodes, 147.2 GFlops/node, 95.4 TFlops peak performance) with a computing elapsed time of 137 hours 42 minutes which includes the time for verification.

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]. However, parallel processing by a distributed-memory parallel computer is indispensable to calculate the further number of digits in realistic time.

Parallel implementation of Karatsuba’s multiplication algorithm [5, 6] on a distributed-memory parallel computer was proposed [7]. 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 [8], which is based on the fast Fourier transform (FFT).

In multiple-precision multiplication of several thousand decimal digits or more, 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 [9]. FNT uses many modulo operations, which are slow due to the integer division process. Thus, we used

```

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

```

Fig. 1. Multiple-precision sequential addition

floating-point real FFT-based multiplication 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 [10]. 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 1,649,267,440,000 decimal digits on a distributed-memory parallel computer. We provide some concluding remarks in section 6.

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

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

The arithmetic operation counts for n -digit multiple-precision sequential additions, subtractions, and multiplications by a single-precision integers is clearly $O(n)$. However, releasing the carries and borrows in these operations is a major factor potentially obstructing parallelization.

For example, a FORTRAN 77 program of multiple-precision sequential addition is shown in Figure 1. Here, ICARRY is a variable to store carry, ITEMP is a temporary variable and IRADIX is a radix of multiple-precision numbers. We assume that the input data has been normalized as $0 \sim \text{IRADIX}-1$, and is 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 can not be parallelized because of data dependency.

```

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

```

Fig. 2. Multiple-precision parallel addition

The algorithm shown in Figure 2 is one that enables us to parallelize the releasing of the carries and borrows. We assume that the input data has been normalized to $0 \sim \text{IRADIX}-1$, and stored in arrays **IA** and **IB**.

We perform multiple-precision addition without the propagation of carries at line 3. Results are checked with $(\text{IA}(2:N) \geq \text{IRADIX})$ at line 4. If the value of each element of the array **IA**(2:N) is greater than or equal to **IRADIX**, we have to compute the carries in line 6 and release the carries in lines 7 and 8. Here, **IC** is a working array to store carries.

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

We assume that the input data has been normalized to $0 \sim \text{IRADIX}-1$, and stored in arrays **IA** and **IB**. In the case of $\text{IRADIX} = 10^8$, the 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, as in the case of $0.99999999 \dots 9 + 0.00000000 \dots 1$, we have to use the carry skip method [11].

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

The same methods can be applied to multiple-precision parallel subtraction and 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.

```

1  SUBROUTINE SKIPNORM(IX,IC,IRADIX,N)
2  INTEGER,DIMENSION(N) :: IX,IC
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

```

Fig. 3. Parallel normalization with the carry skip method

For *floating-point real* FFT-based multiplication, we can use the “balanced representation” [12], 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:

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

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.

We can use an efficient parallel FFT algorithm in the FFT-based multiple-precision parallel multiplication.

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, subtraction, and multiplication by a single-precision integer. 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 [13, 6]. It is well known that multiple-precision division and square root operations can be reduced to multiple-precision addition, subtraction, and multiplication by using Newton iteration [6]. This scheme requires $O(M(n))$ operations, where $M(n)$ is the number of operations for an n -digit multiplication.

3.1 Newton Iteration

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

$$x_{k+1} = x_k + x_k(1 - bx_k), \quad (2)$$

where the multiplication between x_k and $(1 - bx_k)$ can be performed with only half of the normal level of precision [2].

Then the final iteration is performed as follows [13]:

$$a/b \approx (ax_k) + x_k(a - b(ax_k)), \quad (3)$$

where the multiplication between a and x_k , and the multiplication between x_k and $(a - b(ax_k))$ can be performed with only half of the final level of precision.

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

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - ax_k^2), \quad (4)$$

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 [13]:

$$\sqrt{a} \approx (ax_k) + \frac{x_k}{2}(a - (ax_k)^2), \quad (5)$$

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

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

3.2 Parallelization

To compute the n -digit multiple-precision arithmetic by Newton iteration of equations (2) ~ (5), it is necessary to perform multiple-precision parallel addition, subtraction and multiplication on parallel computers which have P processors. In these operations, we can apply the parallel algorithms given in section 2.

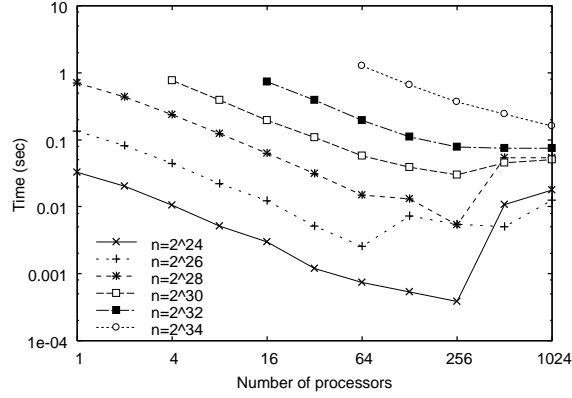


Fig. 4. Execution time of multiple-precision parallel addition $(\pi + \sqrt{2})$, $n =$ number of decimal digits

Here, we discuss the distribution method of the data to each processor. Let us consider an n -digit number x with radix- B .

$$x = \sum_{i=0}^{n-1} x_i B^i, \quad (6)$$

where $0 \leq x_i < B$.

In the case of 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 equation (6). In the case of 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 equation (6).

Since the number of calculated digits can be doubled by the Newton iteration for the division and square root operations, the computation area changes gradually. In the block distribution, because of load imbalancing, the parallel computation time for the operations of the multiple-precision parallel division and square root operations is $O((M(n)/P) \log P)$ [14]. Note that $M(n)$ is the number of operations of multiplying two n -digit numbers.

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 $O(M(n)/P)$ [14].

Thus, the parallel computation time of the cyclic distribution is less than that of the block distribution. Therefore, we use the cyclic distribution for the multiple-precision parallel arithmetic.

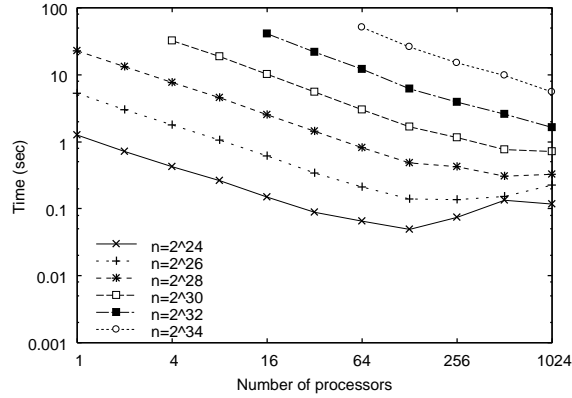


Fig. 5. Execution time of multiple-precision parallel multiplication ($\pi \times \sqrt{2}$), $n =$ number of decimal digits

4 Experimental Results

In order to evaluate our multiple-precision parallel arithmetic algorithms, the decimal digit 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 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 4-socket of quad-core AMD Opteron (2.3 GHz) in 16-core shared-memory configuration with 147.2 GFlops of performance.

In the experiment, we used 1 processor (4 cores) \sim 1,024 processors (4,096 cores). The original program was written in FORTRAN 77 with MPI and OpenMP. Open MPI 1.3 [15] was used as a communication library.

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 FFT-based multiplication.

Figure 4 shows the averaged execution times of 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 is due to the arithmetic operation count for n -digit multiple-precision parallel addition is only $O(n/P)$.

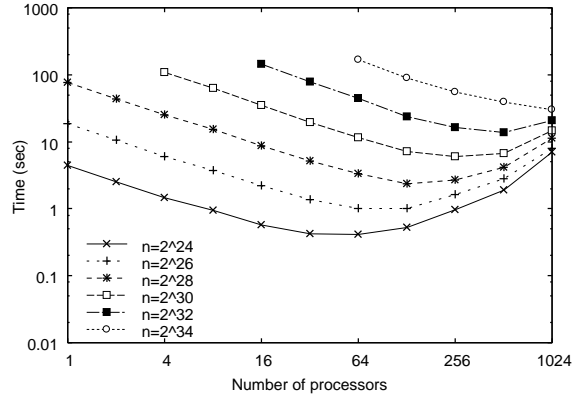


Fig. 6. Execution time of multiple-precision parallel division ($\sqrt{2}/\pi$), $n =$ number of decimal digits

Figure 5 shows the averaged execution times of multiple-precision parallel multiplication ($\pi \times \sqrt{2}$). We can see that our multiple-precision parallel multiplication is scalable for $n > 2^{30}$ on 1,024 processors from Figure 5.

Figures 6 and 7 show the averaged execution times of 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.

5 1, 649, 267, 440, 000 Decimal Digits of π Calculation

The computation of π to a high precision has a long history and many computations have been performed [16]. The development of new programs suited to the calculation of π and high speed computers with a large memory have thrown more light on this fascinating number. In 2002, Kanada et al. computed π to over 1.24 trillion decimal digits by using arctangent formulae [17].

We have computed π to more than 1.64 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

The Gauss-Legendre Algorithm: Main Algorithm

The theoretical basis of the Gauss-Legendre algorithm for π is explained in the references [18–20].

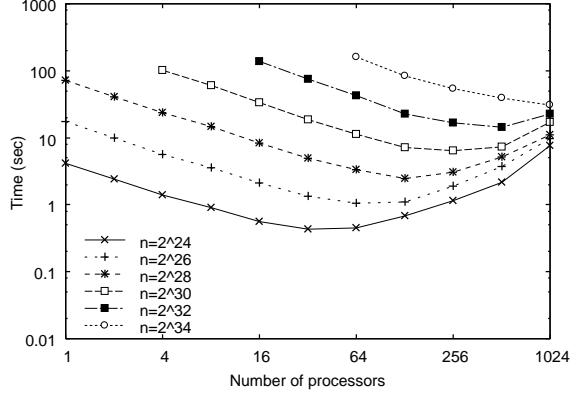


Fig. 7. Execution time of multiple-precision parallel square root ($\sqrt{\pi}$), n = number of decimal digits

We employed the following improved Gauss-Legendre algorithm for π [21]:

```

A := 1; B := 1/2; T := 1/2; X := 2;
while A - B > 10-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.

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

Borweins’ Quartically Convergent Algorithm: Verification Algorithm

Borweins’ quartically convergent algorithm [20] is explained as following scheme. Let $a_0 = 6 - 4\sqrt{2}$ and $y_0 = \sqrt{2} - 1$. Iterate the following calculations:

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

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

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

We employed the following improved Borweins’ quartically convergent algorithm for π :

```

A := 6 - 4 $\sqrt{2}$ ; Y := 17 - 12 $\sqrt{2}$ ; X := 2;
    
```

```

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

In each iteration, the 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. 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. To perform FFT-based multiplication of $3 \times 2^{39} \approx 1.649$ trillion decimal digit numbers, at least 20736 GB of main memory should be available. It was impossible to obtain 1.649 trillion decimal digits through in-core (on main memory) operations because of the 7168 GB main memory limit on 256 nodes of Appro Xtreme-X3 supercomputer.

Thus, we performed 3×2^{35} point real FFT for $3 \times 2^{36} \approx 206$ billion decimal digit multiplications on main memory. Then, we used Karatsuba's multiplication algorithm for $3 \times 2^{39} \approx 1.649$ trillion decimal digit multiplications. These schemes needed approximately 6732 GB and 6348 GB of main memory for working storage for the Gauss-Legendre and Borweins' quartically convergent algorithms, respectively.

5.3 Results of π 1, 649, 267, 440, 000 Decimal Digit Calculation

The calculations of π by Gauss-Legendre algorithm and Borweins' quartically convergent algorithm were performed on 256 nodes of Appro Xtreme-X3 super-computer.

All routines were written in FORTRAN 77 with MPI and OpenMP. Main program and verification program were run on 1,024 MPI processes, i.e. each node has 4 MPI processes. Due to the time limit of a job, two programs were performed in 10 steps, respectively.

Main program run:

Job start : 2nd January 2009 21:17:26 (JST)
 Job end : 5th January 2009 23:56:23 (JST)
 Total elapsed time : 64 hours 14 minutes
 Main memory : 6732 GB
 Algorithm : Gauss-Legendre algorithm

Verification program run:

Job start : 6th January 2009 01:36:45 (JST)
 Job end : 26th January 2009 08:45:58 (JST)
 Total elapsed time : 73 hours 28 minutes
 Main memory : 6348 GB
 Algorithm : Borweins' quartically convergent algorithm

The decimal numbers of π and $1/\pi$ from 1, 649, 267, 439, 951-st to 1, 649, 267, 440, 000-th digits are:

π : 7712856414 0105560548 9805732574 3212539317 0912654849
 $1/\pi$: 7726694296 8436590719 4549360485 5555663940 4302590248.

The main computation took 40 iterations of the improved Gauss-Legendre algorithm for π , to yield $3 \times 2^{39} = 1, 649, 267, 441, 664$ 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 139 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. We also parallelized an operation of releasing propagated carries and borrows in multiple-precision addition, subtraction and multiplication.

We proposed a parallelization of releasing these propagation operations by using the carry skip method. Similarly to 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 normalization of multiple-precision addition, subtraction and multiplication.

The presented multiple-precision parallel arithmetic algorithms make it possible to compute more than 1.6 trillion decimal digits of π computed on 256 nodes of Appro Xtreme-X3 (648 nodes, 147.2 GFlops/node, 95.4 TFlops peak performance).

References

1. Smith, D.M.: Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw.* **17** (1991) 273–283
2. Bailey, D.H.: 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. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. *Doklady Akad. Nauk SSSR* **145** (1962) 293–294
6. Knuth, D.E.: *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. 3rd edn. Addison-Wesley, Reading, MA (1997)
7. Cesari, G., Maeder, R.: Performance analysis of the parallel Karatsuba multiplication algorithm for distributed memory architectures. *Journal of Symbolic Computation* **21** (1996) 467–473
8. Schönhage, A., Strassen, V.: Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)* **7** (1971) 281–292
9. Fagin, B.S.: Large integer multiplication on hypercubes. *Journal of Parallel and Distributed Computing* **14** (1992) 426–430
10. Char, B., Johnson, J., Saunders, D., Wack, A.P.: Some experiments with parallel bignum arithmetic. In: *Proc. 1st International Symposium on Parallel Symbolic Computation*. (1994) 94–103
11. Lehman, M., Burla, N.: Skip techniques for high-speed carry propagation in binary arithmetic units. *IRE Trans. Elec. Comput.* **EC-10** (1961) 691–698
12. Crandall, R., Fagin, B.: Discrete weighted transforms and large-integer arithmetic. *Math. Comput.* **62** (1994) 305–324
13. Karp, A.H., Markstein, P.: High-precision division and square root. *ACM Trans. Math. Softw.* **23** (1997) 561–589
14. Takahashi, D.: 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) 229–235
15. OpenMPI: Open Source High Performance Computing. <http://www.open-mpi.org/>.
16. Berggren, L., Borwein, J., Borwein, P., eds.: *Pi: A Source Book*. 3rd edn. Springer-Verlag, New York (2004)
17. Kanada Laboratory home page. <http://www.super-computing.org/>.
18. Brent, R.P.: Fast multiple-precision evaluation of elementary functions. *J. ACM* **23** (1976) 242–251
19. Salamin, E.: Computation of π using arithmetic-geometric mean. *Math. Comput.* **30** (1976) 565–570
20. Borwein, J.M., Borwein, P.B.: *Pi and the AGM — A Study in Analytic Number Theory and Computational Complexity*. Wiley, New York (1987)
21. Oura, T.: 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).