

Parallel Computation of π Using High Precision Arithmetics

Author: Maryna Babayeva
Student-ID: F100185
maryna.babayeva@gmail.com

Course: Parallel Algorithms
Department of Mathematics
Utrecht University

Lecturer: Rob Bisseling
January 8, 2011

Abstract

During the last decades estimating π has become a competition and a benchmarking tool. Many different algorithms have been developed and with the advent of high performance architectures it is possible to calculate very large number of π decimal digits. Within the scope of the Parallel Algorithms lecture a parallel library for high precision arithmetics will be introduced and its performance on the Huygens¹ supercomputer using Message Passing Interface (MPI) will be analyzed. This library will contain parallel implementations for addition and subtraction. For parallel multiplication an efficient algorithm by Schönhage and Strassen will be implemented, which uses complex Fast Fourier Transform. In addition to that Newton's division and square root algorithms will be also used to finally calculate up to 1 million digits of π .

1 Introduction

The high precision computation of π has a long history and many numerical methods have been developed so far. In the beginning of the 20th century, there appeared some new and very interesting theoretical result by Srinivasa Ramanujan on π computation. In 1976, an innovative quadratically convergent formula, based on the method of algebraic-geometric mean, was published independently by Brent and Salamin. This approach was taken even further by Jonathan and Peter Borwein [4]. They developed a fast algorithm to approximate π in the 1980's, which has a quartical convergence.

A very interesting formula for calculating π was discovered in 1995 by Simon Plouffe and is called the Bailey-Borwein-Plouffe (BBP). This formula computes π in a hexadecimal base without the need to compute the previous digits. In 1997, Fabrice Bellard improved Plouffe's algorithm for digit-extraction in an arbitrary base to reduce the runtime to $O(n^2)$ [2].

¹<http://huygens.supercomputer.nl/SARA/>

Besides the theoretical development it was the introduction of digital computers, which made it possible to calculate an unbelievable large number of decimals of π . Such an example is the y-cruncher by Alexander Yee. This program holds the current world record for calculating 5,000,000,000,000 digits since August 2nd, 2010. This a record for both super computers as well as home-built computers [1]. It uses checkpointing and efficient disk swapping to facilitate extremely long run times and memory-expensive computations.

Although, many ways for π calculation exist, the Brent-Salamin algorithm has been chosen here to approximate up to 1 million digits of π due to its mathematical simplicity. Here, I will explain the representation of high precision numbers and its data structure, which will serve for parallel implementations of basic algebraic operations and MPI computing model will be used. The choice of block data distribution will be explained on the example of parallel addition. Further, an efficient approach for high precision multiplication by means of the Fast Fourier Transform will be described. This will be the starting point for parallel division and square root operations using Newton's numerical root finding algorithm. Finally, in the results section the run times and the speed-up of the described parallel algorithms will be measured.

1.1 Parallel π Estimation

The Gauss-Legendre algorithm or Brent-Salamin algorithm can compute π up to n digits in time proportional to $n \log n \log \log n$. It was used to compute the first 206,158,430,000 decimal digits of π on September 18 to 20 in 1999, and the results were checked with Borwein's algorithm. The Brent-Salamin method converges quadratically. Therefore, $\log_2 n$ iterations according to equation 2 need to be performed to obtain n digits of π . Initial values are set according to equation 1.

$$a_0 = 1, b_0 = \frac{1}{\sqrt{2}}, t_0 = \frac{1}{4}, p_0 = 1 \quad (1)$$

$$\begin{aligned} a_{i+1} &= \frac{a_i + b_i}{2} \\ b_{i+1} &= \sqrt{a_i b_i} \\ t_{i+1} &= t_i - p_i (a_i - a_{i+1})^2 \\ p_{i+1} &= 2p_i \\ \pi &\approx \frac{(a_i + b_i)^2}{4t_i} \end{aligned} \quad (2)$$

In order to parallelize this algorithm for high precision numbers, basic mathematical operations as addition, subtraction, multiplication, division, and square root operations need to be implemented in parallel first. This will be explained by the following chapters.

2 Parallel Implementation of Basic High precision Mathematical Operations

2.1 High Precision Number Representation

To represent an n -digit number a base- B representation of a floating point number x is the shortest sequence of the digits x_i such that each digit satisfies $0 \leq x_i < B$. The floating point number can thus be stored according to equation 3 as a vector $(x_0, x_1, \dots, x_{n-1})$

$$x = sB^e \sum_{i=0}^{n-1} x_i B^i \quad (3)$$

with an exponential shift e and a sign s .

2.2 Data Distribution

To explain the choice for the block data distribution the example of high precision addition will be used. The sequential algorithm 2.1 shows a basic addition of two high precision n -digit numbers x and y with the complexity $O(n)$.

Algorithm 2.1: SEQADD(x, y, res)

```

input :  $x$  as 1st  $n$ -digit summand
        $y$  as 2nd  $n$ -digit summand
output :  $n$ -digit  $res$  as sum of  $x$  and  $y$ 

carry  $\leftarrow 0$ 
for  $i \leftarrow n - 1$  to 0
  do  $\begin{cases} temp \leftarrow x_i + y_i + carry \\ carry \leftarrow temp \div B \\ res_i \leftarrow temp \bmod B \end{cases}$ 
 $res_{n-1} \leftarrow x_{n-1} + y_{n-1} + carry$ 
return ( $res$ )

```

Figure 1 shows an approach to parallelize the addition. To ensure an efficient implementation the communication between the processors, owing different parts of the high precision number, has to be minimized. The addition routine has (a) to communicate the carry over and (b) in case the first digit x_1 exceeds base B , the whole array needs to be shifted to the right by one to create room for the new significant digit. In the following the communication costs of block and cyclic data distributions are compared.

Block Distribution: The most common and straight forward way of distributing data for parallel computations is a block distribution. The array x_i containing an n -digit number x can be distributed over p processors. This distribution can be described by

$$distr(x_i) = \phi_b \quad (4)$$

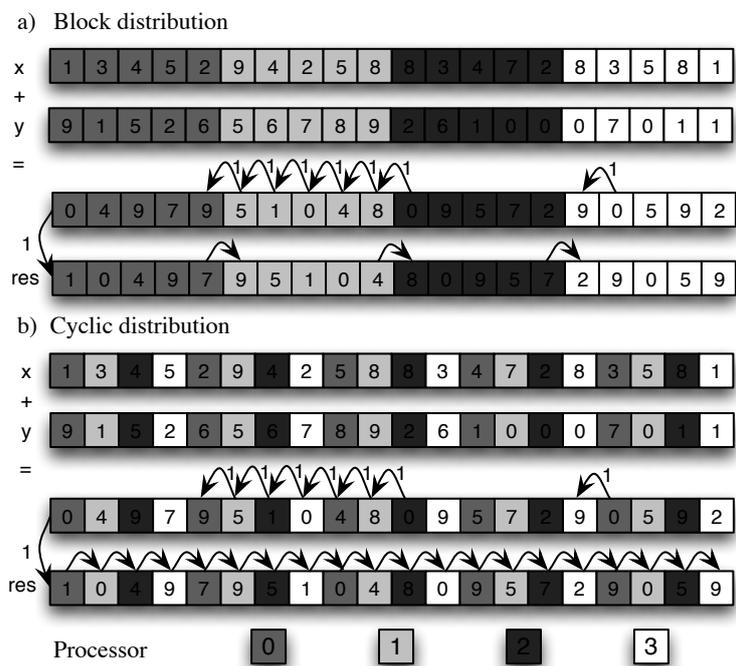


Figure 1: An example of different distributions is shown here. The grey shade denotes the processor ID, whose private memory owns the cell. It can be seen that the communication heavily depends on the distribution for the propagation of the carry over. Additionally, the last step requires a shift to the right, which has the best behavior in terms of communication for block distribution. Here two 16-digit integer x and y with base $B = 10$ are added.

with $\phi_b(i) = \lfloor i/p \rfloor$, for $0 \leq i \leq n$.

The block distribution is easy to implement and leads to the maximum communication cost of $b_{block} = 2p - 1$. This cost consists of $p - 1$ data, which has to be communicated in case of a shift. The p data words have to be send around for the carry over propagation in the worst case. This cost apply provided the fact that only one carry over propagation is needed.

Cyclic Distribution: In contrary the cyclic distribution can be considered. It distributes the data as stated by

$$distr(x_i) = \phi_c \tag{5}$$

with $\phi_c(i) = i \bmod p$, for $0 \leq i \leq n$.

Unfortunately, applying a cyclic distribution as defined in equation 5 leads to a higher communication cost than block distribution as shown in Figure 1. An amount of $b_{cyclic} = 2n/p$ data has to be send in worst case, which is clearly higher than in case of block distribution. Thus, a block data distribution has been chosen for the implementation.

2.3 Parallel Addition

As already introduced in previous section 2.2 on data distribution, the basic addition is done by adding two numbers in a per-digit fashion and ensuring each digit to be within the defined base B .

For a correct implementation few things have to be taken into account. Both numbers have to have the same exponential shift B^e and also the signs have to be considered. Theses steps are explained by the steps (0) and (1) of the algorithm 2.2 PARAADD. To align the two summands and to ensure that they both have the same B^e the algorithm MAKESAMEEXP is introduced. It also uses the algorithm 5.1 ADDWITHCARRY to determine a carry over, which will be then propagated to all other processors . The pseudo code for both routines can be found in the appendix 5.

In step (2) of the algorithm 2.2 PARAADD the routine ADDWITHCARRY is being called as long there exists a non-zero carry over. For random inputs this process is repeated only few times. In a rare worst case this approach leads to p iterations of this communication step. This occurs when $0.99999\dots9$ is added to $0.00000\dots1$, here the carry over has to be propagated all the way to the processor owing the first part of x_i thus, leading to p repeats.

The last step (3) is to shift the final result to the right in case of the first digit x_1 exceeds

base B and to create room for the carry over, which becomes the new significant digit.

Algorithm 2.2: $\text{PARAADD}(x, y, res, s, p)$

input : x 1st n -digit summand

y 2nd n -digit summand

s for the processor ID

p for the number of processors

output : res as n -digit sum of x and y

external : $\text{PARASUB}(x, y, res, s, p)$, $\text{SHIFTRIGHT}(x, n, s, p)$,

$\text{MAKESAMEEXP}(min, max, s, p)$, $\text{MINMAX}(min, max, x, y, s, p)$,

$\text{ADDWITHCARRY}(x, y, res)$

(0) *Determine correct operation according to the signs*

if $y.sign > x.sign$

then $\left\{ \begin{array}{l} \text{PARASUB}(y, x, res, s, p) \\ \text{return } (res) \end{array} \right.$

if $y.sign < x.sign$

then $\left\{ \begin{array}{l} \text{PARASUB}(x, y, res, s, p) \\ \text{return } (res) \end{array} \right.$

(1) *Adjust the exponents of x and y if necessary*

$\text{MINMAX}(min, max, x, y, s, p)$

$\text{MAKESAMEEXP}(min, max, s, p)$

(2) *Add x and y in parallel*

$carry_s \leftarrow \text{ADDWITHCARRY}(x, y, res)$

$\text{BROADCAST}(carry_s, p)$

$maxCarry \leftarrow \max\{carry_s | 0 \leq s < p\}$

while $maxCarry > 0$ $\left\{ \begin{array}{l} carry_s \leftarrow \text{ADDWITHCARRY}(res, carry_{s+1}, res) \\ \text{BROADCAST}(carry_s, p) \\ maxCarry \leftarrow \max\{carry_s | 0 \leq s < p\} \end{array} \right.$

(3) *Shift res by 1 if necessary*

if $carry_0 \neq 0$

then $\text{SHIFTRIGHT}(res, s, p)$

if $s == 0$

then $res_0 \leftarrow carry_s$

return (res)

2.4 Parallel Subtraction

For the high precision subtraction the subtrahend will be rewritten as its radix complement and then added to the minuend. Let us consider an example $1024-456=568$. The

9-complement of 456 is 543. To obtain the final result of the subtraction the result of the addition $1024+543+1=1568$ will be modified by removing the most significant digit, thus obtaining the correct result for the subtraction 568.

Of course, in case the subtrahend is bigger than the minuend the difference will have a negative sign and the complement of the smaller number has to be taken instead. This happens in step (0) of the PARASUB algorithm 2.3.

It is to mention that the shift to the right inside the parallel addition routine has to be skipped in order to omit the most significant digit.

In the final step (3) the leading zeros, which may have been introduced by the addition have to be removed to ensure that the most significant digit is not equal zero. This is done by the algorithm 5.6 REMZEROS, which is shown in the appendix 5.

Algorithm 2.3: PARASUB(x, y, res, s, p)

input : x 1st n -digit minuend

y 2nd n -digit subtrahend

s for the processor ID

p for the number of processors

output : res as n -digit difference of x and y

external : PARAADD(x, y, res, s, p), REMZEROS(x, p, s), PARACOMPL(a, p, s)

MAKESAMEEXP(min, max, s, p), MINMAX(min, max, x, y, s, p),

(0) *Get the smaller number to obtain the sign of the result*

MINMAX(min, max, x, y, s, p)

(1) *Calculate the complement of the smaller number*

PARACOMPL(min, p, s)

(2) *Add compl(min) and max in parallel*

PARAADD(min, max, res, s, p)

(3) *Remove leading zeros of the result and adjust the sign accordingly*

REMZEROS(res, p, s)

$res.sign \leftarrow min.sign$

return (res)

2.5 Parallel Multiplication

Here a high precision multiplication based on floating-point complex Fourier Transform will be used. This approach was suggested by Arnold Schönage and Volker Strassen in 1971 [8]. They proposed a method, which made it possible to calculate a product of two big integers with a complexity of $O(n \log n)$. To show the theory in brief let x and y be high precision

n -digit integer numbers with $x = (x_0, x_1, \dots, x_{n-1})$ and $y = (y_0, y_1, \dots, y_{n-1})$. The product sequence $z = (z_0, z_1, \dots, z_{2n-1})$ of x and y is the discrete convolution $C(x, y)$ as shown by equation 6.

$$\begin{aligned}
z &= \left(\sum_{i=0}^{n-1} x_i \right) \left(\sum_{j=0}^{n-1} y_j \right) \\
&= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j \\
&= \sum_{k=0}^{2n-1} \sum_{i=0}^k x_i y_{k-i} \\
&= \sum_{k=0}^{2n-1} C_k
\end{aligned} \tag{6}$$

The discrete convolution C_k can be calculated fast using the Fourier transform. Let $F(x)$ denote the Fourier transform of vector x and $F^{-1}(x)$ the inverse Fourier transform of x and extend x and y to length $N = 2n$ by appending zeros at the end of each.

$$F_k(x) = \sum_{j=0}^{N-1} x_j e^{-2\pi i j k / N}, \quad F_k^{-1} = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{2\pi i j k / N} \tag{7}$$

with i being the complex number and $N = 2n$

The convolution theorem states, that the Fourier transform of a convolution product is the ordinary product of the Fourier transforms as shown by equation 8 .

$$\begin{aligned}
F(C(x, y)) &= F(x)F(y) \\
C(x, y) &= F^{-1}(F(x)F(y))
\end{aligned} \tag{8}$$

Thus, the product z can be obtained by performing two forward discrete Fourier transforms, one vector complex multiplication and one inverse transform, each of length $N = 2n$.

To exploit the advantage of parallel computation the parallel fast Fourier transform (FFT) algorithm as proposed by Bisseling [3] has been used to implement a parallel multiplication algorithm 2.4 PARAMULT. The FFT algorithm works on cyclically distributed arrays, thus, two routines had to be implemented for data redistribution from block to cyclic and back. The last step of the parallel multiplication is the normalization. After complex multiplication the obtained digits of the resulting array may be bigger than the chosen base B . Therefore a

parallel addition is performed.

Algorithm 2.4: PARAMULT(x, y, res, s, p)

input : x 1st n -digit multiplicand, y 2nd n -digit multiplicand
 s for the processor ID, p for the number of processors

output : res as n -digit product of x and y

external : BLOCKSIZE(p, s, n), BLOCKTOCYCLIC(a, b, s, p), PARAADD(x, y, res, s, p),
CYCLICTOBLOCK(a, b, s, p), MPI_FFT(x, n, s, p, dir), INITCOMPLEX($compX, s, p$)

$w \leftarrow$ BLOCKSIZE(p, s, n)

(0) *redistribute data from block to cyclic for FFT*
BLOCKTOCYCLIC($x, compX, s, p$)
BLOCKTOCYCLIC($y, compY, s, p$)
INITCOMPLEX($compX, s, p$)
INITCOMPLEX($compY, s, p$)

(1) *Perform FFT on complex data*
MPI_FFT($compX, 2n, s, p, 1$)
MPI_FFT($compY, 2n, s, p, 1$)

(2) *Perform complex multiplication*
for $i \leftarrow 1$ **to** $2w$
 do $\begin{cases} reX \leftarrow compX_{2i} \\ imX \leftarrow compX_{2i+1} \\ reY \leftarrow compY_{2i} \\ imY \leftarrow compY_{2i+1} \\ compX_{2i} \leftarrow reX * reY - imX * imY \\ compX_{2i+1} \leftarrow reX * imY + reY * imX \end{cases}$

(3) *Perform inverse FFT on complex result*
MPI_FFT($compX, 2n, s, p, -1$)

(4) *Redistribute the data back from cyclic to block*
CYCLICTOBLOCK($compRES, compX, s, p$)

(5) *Convert from double to int and round up*
for $i \leftarrow 0$ **to** w
 do $res_i \leftarrow \lfloor compRES_i + 0.5 \rfloor$

(6) *Normalize the result and set exponent and sign accordingly*
PARAADD($res, 0, res, s, p$)
 $res.e \leftarrow res.e + x.e + y.e$
 $res.sign \leftarrow x.sign + y.sign$
return (res)

2.6 Parallel Division and Square Root

Although, many methods exist to perform a division and square root operations, Newton's method is known to be the fastest for large integers [6, 5]. By using Newton's approximation high precision division and square root can be reduced to addition subtraction and multiplication [9]. Newton's method aims to find the root of a function $f(x)$ by repeatedly evaluating equation 9.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (9)$$

The proposed algorithm has quadratical convergence given a good initial guess. Several techniques have been developed for obtaining the initial approximation for the Newton's algorithm. As suggested by Kornerup and Muller [7] the natural starting point would be the arithmetic mean as shown by equation 10 for division and 11 for square root operations.

$$x_0 = \frac{1}{2} \left(\frac{1}{a_{min}} + \frac{1}{a_{max}} \right) \quad (10)$$

$$x_0 = \frac{1}{2} (\sqrt{a_{min}} + \sqrt{a_{max}}) \quad (11)$$

To calculate a_{min} and a_{max} the first digits of the high precision number a are used. In division, the quotient of a and b is computed as follows. First, equation 12 is evaluated that converges to $1/b$ by setting $f(x) = 1/x - b$ and $f'(x) = -1/x^2$. Subsequently the result is multiplied by a . Note that there is no division in the equation any longer.

$$\begin{aligned} x_{i+1} &= x_i - \frac{1/x_i - b}{-1/x_i^2} \\ &= x_i + x_i(1 - bx_i) \\ &\approx 1/b \end{aligned} \quad (12)$$

To calculate a square root of a high precision number b $1/\sqrt{b}$ is being approximated first by using the Newton's method as shown in equation 13 and then multiplied by b .

$$\begin{aligned} x_{i+1} &= x_i - \frac{1/x_i^2 - b}{-2/x_i^3} \\ &= x_i + \frac{x_i}{2}(1 - bx_i^2) \\ &\approx 1/\sqrt{b} \end{aligned} \quad (13)$$

3 Theoretical and Experimental Results

The theoretical complexity of the implemented algorithms can be described in a following way as proposed by Bisseling for the BSP model in [3].

The **parallel addition** needs to align two high precision numbers of length n if they have different exponents. Here, the communication costs in worst case $O(n/p)$ 32-bit data words in addition to $O(n/p)$ copy-shift operations per processor. Also a maximum of p iterations of the `ADDWITHCARRY` routine might be needed, which sends p messages per processor to communicate the carry. Thus, a maximum communication cost of $O(p^2)$ applies and in that case maximum p times $O(n/p)$ additions and also p synchronizations per processor. At the end one extra shift to the right may be needed. As it can be assumed that the worst case scenario happens quite rarely the total cost can be approximated by $T_{add} = n/p + p(p-1)g + 2l$.

For the **parallel subtraction** the same cost approximations apply but with an additional step for removing leading zeros with a maximum communication cost of $O(n/p)$ per processor plus $O(n/p)$ copy-shift operations, which adds an extra synchronization, and without the final shift to the right. Therefore the same costs for best case scenario as for the addition may be assumed $T_{sub} = n/p + p(p-1)g + 2l$.

To perform a **parallel multiplication** the algorithm needs to redistribute the data three times. These are block-to-cyclic and cyclic-to-block redistributions with a communication cost of n/p data words each per processor. Also $4n/p$ additions are needed for complex addition of two $2n$ -digit numbers and of coarse tree FFTs on $2n$ -long arrays with a total cost of $T_{FFT} = (5n \log_2 n)/p + 2ng/p + 3l$ for $p > 1$. The accumulated cost for a high precision multiplication is $T_{mult} = T_{add} + 3T_{FFT} + 3n/pg + 3l$.

The costliest operations are **parallel division** and **square root** operations with the following costs for division of $T_{div} = \log_2 n(2T_{mult} + T_{add} + T_{sub}) + T_{mult}$ and $T_{sqrt} = \log_2 n(4T_{mult} + T_{add} + T_{sub}) + T_{mult}$ for performing a square root of an n -digit high precision number.

Finally, the cost for approximating n digits of π **in parallel** with the Brent-Salamin method accumulate to $T_\pi = T_{sqrt} + \log_2 n(4T_{mult} + 2T_{add} + 2T_{sub} + T_{sqrt}) + 2T_{mult} + T_{add} + T_{div}$.

In order to compare the measured timings with theoretically predicted complexity the characteristic parameters of the Huygens supercomputer were measured by running a benchmark. A benchmark from the *BSPedupack*² by Rob Bisseling was used and the parameters as shown in table 1 were obtained.

Table 1: Characteristics of the Huygens supercomputer

# of processors	r [flops]	g [flop units]	l [flop units]
1	195	58	570
2	194	56	2007
4	187	62	4288
8	195	57	8316
16	194	60	39138
32	195	59	42821

²<http://www.staff.science.uu.nl/~bisse101/Software/software.html>

First, I would like to discuss the performance of the parallel addition. As shown by figure 2 the relative speed-up is linear but only for up to 16 processors. When 32 processors are used the speed-up drops significantly. This is a good example when the gain in computation speed-up cannot make up for the increased communication cost anymore. Other performance measurements for parallel multiplication, division, and square root operation can be found in the appendix 5.1 (figure 4) and show a good speed-up behavior as can be also seen in figure 3a.

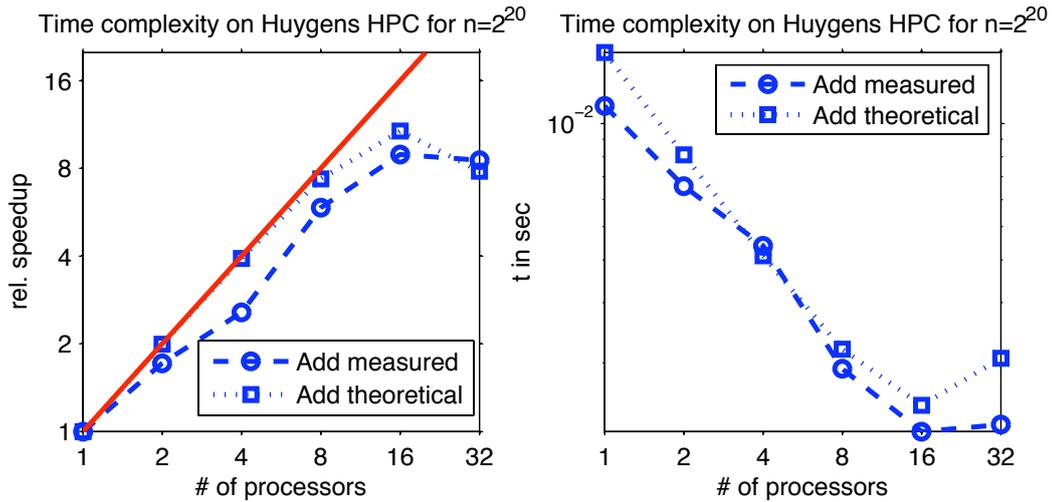


Figure 2: Left figure shows the relative speed-up for adding two 2^{20} -digits numbers ($\pi + \pi$) in parallel. The figure on the right shows theoretical time complexity compared to measured timings for different number of processors.

Also some time measurements have been done to observe the performance of the parallel Brent-Salamin method for calculating up to $n = 2^{20}$ digits of π . For $n = 2^{20}$ and $n = 2^{18}$ measurements not for all numbers of processors could be performed due to high memory consumption per processor.

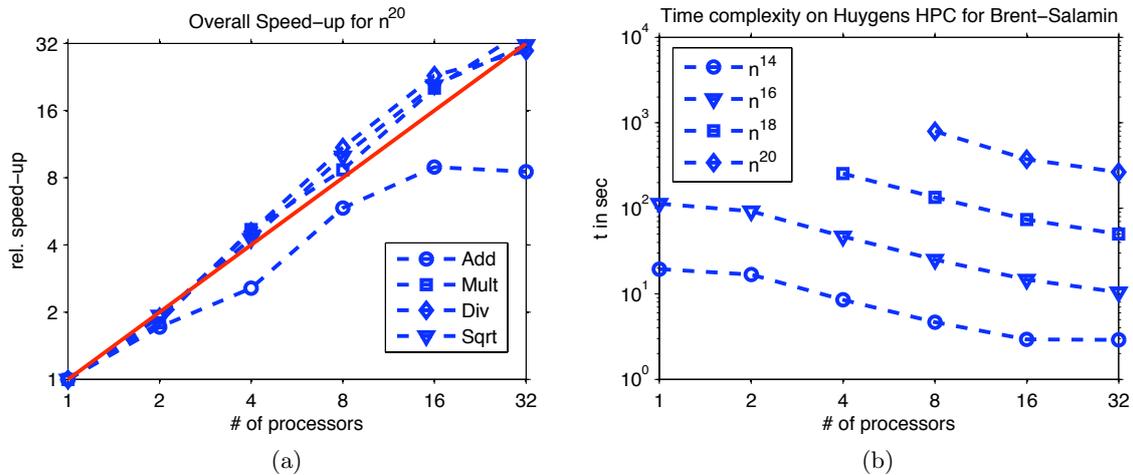


Figure 3: Figure 3a shows an overall speed-up for parallel addition, multiplication, division, and square root for $n = 2^{20}$ decimal digits. Figure 3b shows time measurements for different numbers of π digits on the Huygens HPC.

4 Discussion

To parallelize the Brent-Salamin algorithm in order to calculate 1 million digits of π , the basic mathematical routines have been implemented in parallel first using the MPI model and programming language C. It has been shown that block data distribution is the best choice to keep the communication cost low. The calculated digits have been verified by comparing the result to pre-computed π file from <http://pi.is.online.fr/>.

Although, the proposed approach already makes it possible to calculate up to 1 million digits of π , several optimization have not been applied, due to time limitations for the project. First, a different method with better convergence rate than the Brent-Salamin method could be used, which would eventually speed up the computation time.

Further, better initial values for Newton's division and Square root operation may reduce the number of iterations. This could be done by providing a look-up table filled with suitable starting values for different input data.

Also, the accuracy of parallel multiplication can be increased by using number-theoretic transforms instead of discrete Fourier transforms, thus avoiding rounding error by using modular arithmetic instead of complex numbers. By applying this method all computations would be done in integer arithmetic. This would avoid the necessary cast from float to integer and rounding errors.

To enhance the performance of parallel addition an efficient carrySkip method to reduce the number of iterations when propagating the carry could be used, as suggested by Takahashi [10].

Last, but not least a hybrid programming model with MPI and OpenMP could be used. MPI could take the obvious role of inter-node communication, whereas OpenMP could be used to parallelize the intra-node computation *e.g.* parallelizing FOR LOOPS over blocks of the high

precision number.

References

- [1] <http://www.numberworld.org/y-cruncher/>, accessed 7 Jan., 2011.
- [2] David Bailey, Peter Borwein, and Simon Plouffe. On the rapid computation of various polylogarithmic constants. *Math. Comput.*, 66:903–913, April 1997.
- [3] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004.
- [4] 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. *Am. Math. Monthly*, 96:201–219, March 1989.
- [5] Karl Hasselström. Fast division of large integers - a comparison of algorithms. Master's thesis, Royal Institute of Technology Stockholm, February 2003.
- [6] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Trans. Math. Softw.*, 23:561–589, December 1997.
- [7] Square root Reciprocals, Square root Reciprocals, Peter Kornerup, Peter Kornerup, Jean-Michel Muller, Jean michel Muller, Thme Gnie Logiciel, and Projet Arnaire. Choosing starting values for newton-raphson computation of reciprocals, square-roots and square-root reciprocals, 2003.
- [8] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281-292, 1971.
- [9] Daisuke Takahashi. Implementation of multiple-precision parallel division and square root on distributed-memory parallel computers. In *Proceedings of the 2000 International Workshop on Parallel Processing, ICPP '00*, pages 229–, Washington, DC, USA, 2000. IEEE Computer Society.
- [10] Daisuke Takahashi. Parallel implementation of multiple-precision arithmetic and 2,576,980,370,000 decimal digits of π ; calculation. *Parallel Comput.*, 36:439–448, August 2010.

5 Appendix

5.1 Performance evaluation

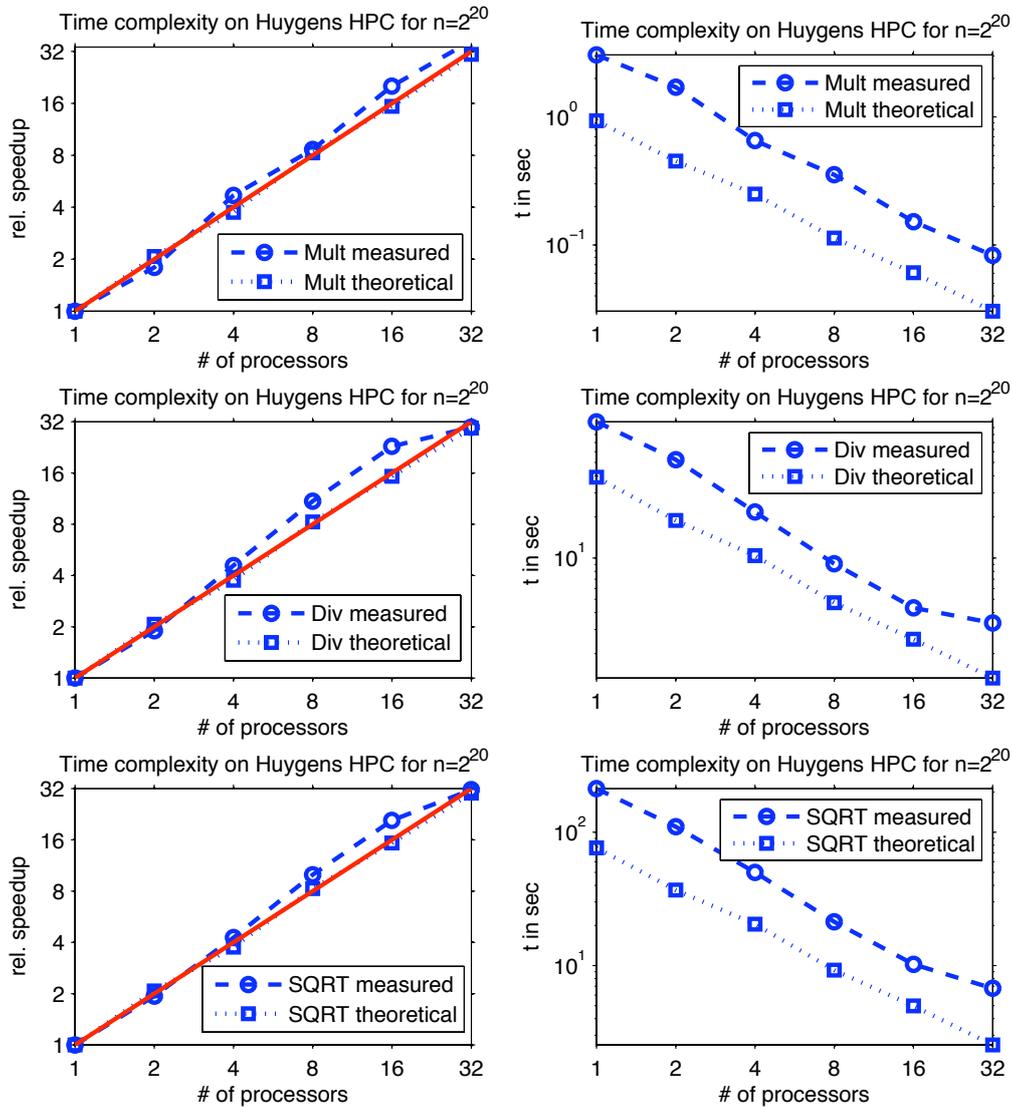


Figure 4: Time measurements and relative speed-ups for parallel multiplication (xy), division ($1/y$) and square root (\sqrt{y}) operations for $n = 2^{20}$ decimal digits and $x = y = \pi$.

5.2 addWithCarry

Algorithm 5.1: ADDWITHCARRY(x, y, res, s, p)

input : x : significant of the 1st summand
 y : significant of the 2nd summand
 s for the processor ID
 p for the number of processors
output : res : significant of the sum of x and y significants
 $carry$ the carry over
external : BLOCKSIZE(p, s, n)

$w \leftarrow \text{BLOCKSIZE}(p, s, res.precision)$
 $carry \leftarrow 0$
 $temp \leftarrow 0$
for $i \leftarrow w - 1$ **to** 0
 do $\begin{cases} temp \leftarrow x_i + y_i + carry \\ carry \leftarrow temp \div B \\ res_i \leftarrow temp \bmod B \end{cases}$
return ($carry, res$)

5.3 initComplex

Algorithm 5.2: INITCOMPLEX($compX, s, p$)

$w \leftarrow \text{BLOCKSIZE}(p, s, 2n)$
for $i \leftarrow 1$ **to** $w - 1$
 do $\begin{cases} compX_{2(w-i)} \leftarrow compX_{w-i} \\ compX_{2(w-i)+1} \leftarrow 0 \\ compX_{2(i-1+w)} \leftarrow 0 \\ compX_{2(i-1+w)+1} \leftarrow 0 \end{cases}$
return ($compX$)

5.4 BlockToCyclic

Algorithm 5.3: BLOCKTOCYCLIC(b, c, p, s)

input : b : local array of length n with part of block distributed data
 c : array of length n
 s for the processor ID
 p for the number of processors
output : c : local array with part of cyclically distributed data from c

```
packet  $\leftarrow$   $n/p$ 
for  $i \leftarrow 0$  to  $n$ 
  do  $\left\{ \begin{array}{l} m \leftarrow i + sn \\ destID \leftarrow m \bmod p \\ l \leftarrow (m - destID)/p \\ offset \leftarrow packet(destID - s) \\ temp_{offset+l} \leftarrow b_i \end{array} \right.$ 
  ALLTOALL( $temp, c, packet$ )
return ( $c$ )
```

5.5 CyclicToBlock

Algorithm 5.4: CYCLICTOBLOCK(c, b, p, s)

input : c : local array of length n with part of cyclically distributed data
 b : array of length n
 s for the processor ID
 p for the number of processors
output : b : local array with part of block distributed data from c

```
packet  $\leftarrow$   $n/p$ 
ALLTOALL( $c, temp, packet$ )
for  $i \leftarrow 0$  to  $n$ 
  do  $\left\{ \begin{array}{l} m \leftarrow ip + s \\ destID \leftarrow m/n \\ l \leftarrow m \bmod n \\ b_{l+destID-s} \leftarrow temp_i \end{array} \right.$ 
return ( $b$ )
```

5.6 makeSameExp

Algorithm 5.5: MAKE_SAME_EXP(a, b, s, p)

input : a : high precision number
 b : high precision number with $b > a$
 s for the processor ID
 p for the number of processors
output : a shifted to the right and filled with zeros
external : BLOCK_SIZE(p, s, n)

$w \leftarrow \text{BLOCK_SIZE}(p, s, n)$

(0) Determine amount of shifts to the right

$shift \leftarrow \text{abs}(a.e - b.e)$

(1) If shifts exceed the amount of digits all digits will be replaced by 0

if $shift \geq n$
 then $a \leftarrow 0$

(2) In the other case the data inside the array will be shifted to the right

else $\left\{ \begin{array}{l} len \leftarrow n - shift \\ k \leftarrow 0 \\ \text{for } i \leftarrow 0 \text{ to } i < p \text{ and } k < shift \\ \quad \text{do } \left\{ \begin{array}{l} \text{if } k < (shift - shift \bmod w) \\ \quad \text{then } n_i \leftarrow w \\ \quad \quad k \leftarrow k + w \\ \quad \text{else } n_i \leftarrow shift \bmod w \\ \quad \quad k \leftarrow shift \bmod w \end{array} \right. \\ \text{for } i \leftarrow s * w, j \leftarrow 0 \text{ to } i < len \text{ and } j < w \\ \quad \text{do } array_i \leftarrow a_j \end{array} \right.$

ALLREDUCE($array_s, array, SUM$)

$a \leftarrow 0$

for $j \leftarrow 0$ to $p - 1$

do $l_{j+1} \leftarrow w - n_j + l_j$

for $i \leftarrow n_s j \leftarrow l_s$ to $i < w$ and $j < len$

do $a_i \leftarrow array_j$

(3) Adjust the exponent accordingly

$a.e \leftarrow a.e + shift$

return (a)

5.7 remZeros

Algorithm 5.6: REMZEROS(x, s, p)

input : x : high precision number

s for the processor ID

p for the number of processors

output : a shifted to the left by amount of leading zeros

external : BLOCKSIZE(p, s, n)

$w \leftarrow \text{BLOCKSIZE}(p, s, n)$

(0) Determine amount of leading zeros for each processor and broadcast it

while $x_i == 0$ **and** $i < w$

do $\begin{cases} n_s \leftarrow n_s + 1 \\ i \leftarrow i + 1 \end{cases}$

ALLGATHER(n_s, n)

if $n_0 > 0$

(1) Determine total amount of leading zeros

$i \leftarrow 0$

while $n_i == w$ **and** $i < p - 1$

do $i \leftarrow i + 1$

$zeros \leftarrow n_i + i * w$

(2) reset n_j for $j \geq i$

if $i \neq p - 1$ **and** $p > 1$

then $\begin{cases} i \leftarrow i + 1 \\ \text{for } i \text{ to } p - 1 \\ \text{do } n_i \leftarrow 0 \end{cases}$

then (3) Prepare an array for non-zero values

$len \leftarrow n - zeros$

for $j \leftarrow 0$ **to** $p - 2$

do $l_{j+1} = w - n_j + l_j$

for $j \leftarrow n_s, i \leftarrow l_s$ **to** $j < w$ **and** $i < len$

do $array_i \leftarrow x_j$

ALLREDUCE($array_s, array, SUM$)

(4) Place the data shifted to the left inside the x_i array

$x \leftarrow 0$

for $i \leftarrow 0, j \leftarrow s * w$ **to** $i < w$ **and** $j < len$

do $x_i \leftarrow array_j$

(5) Adjust the exponent accordingly

$x.e \leftarrow x.e - zeros$

return (x)