Computation of 2700 billion decimal digits of Pi using a Desktop Computer

Fabrice Bellard

Jan 6, 2010

This article describes some of the methods used to get the world record of the computation of the digits of π digits using an inexpensive desktop computer.

1 Notations

We assume that numbers are represented in base B with $B = 2^{64}$. A digit in base B is called a *limb*. $M(n)$ is the time needed to multiply n limb numbers. We assume that $M(Cn)$ is approximately $CM(n)$, which means $M(n)$ is mostly linear, which is the case when handling very large numbers with the Schönhage-Strassen multiplication [5].

$\log(n)$ means the natural logarithm of n. $\log_2(n)$ is $\log(n)/\log(2)$.

SI and binary prefixes are used (i.e. 1 TB = 10^{12} bytes, 1 GiB = 2^{30} bytes).

2 Evaluation of the Chudnovsky series

2.1 Introduction

The digits of π were computed using the Chudnovsky series [10]

$$\frac{1}{\pi} = \frac{12}{C^{3/2}} \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(U+Vn)}{(n!)^3(3n)!C^{3n}}$$

with

$$U = 13591409 \quad V = 545140134 \quad C = 640320\ .$$

It was evaluated with the binary splitting algorithm. The asymptotic running time is $O(\log(n)^2 M(n))$ for a n limb result. It is worst than the asymptotic running time of the Arithmetic-Geometric Mean algorithms of $O(\log(n)M(n))$ but it has better locality and many improvements can reduce its constant factor.

Let $S(a, b)$ defined as

$$\sum_{n=a}^{b-1} a_n \frac{\prod_{k=a}^{n} g(k)}{\prod_{k=a}^{n} q(k)}$$

If we define

$$G(a,b) = \prod_{k=a}^{b-1} g(k)$$

$$Q(a,b) = \prod_{k=a}^{b-1} q(k)$$

and $P(a,b)$ such as

$$S(a,b) = \frac{P(a,b)}{Q(a,b)}$$

then if $a < c < b$, we have:

$$\frac{P(a,b)}{Q(a,b)} = \frac{P(a,c)}{Q(a,c)} + \frac{G(a,c)P(c,b)}{Q(a,c)Q(c,b)}$$

hence

$$P(a,b) = P(a,c)Q(c,b) + P(c,b)G(a,c)$$

which gives a way to compute P recursively. Moreover, both Q and G can be computed recursively by seeing that:

$$\begin{aligned} G(a,b) &= G(a,c)G(c,b) \\ Q(a,b) &= Q(a,c)Q(c,b) \end{aligned}$$

The binary splitting algorithm allows to compute $S(a,b)$ recursively:
If $a = b - 1$ then

$$\begin{aligned} G(b-1,b) &\leftarrow g(b) \\ Q(b-1,b) &\leftarrow q(b) \\ P(b-1,b) &\leftarrow a(b)g(b) \end{aligned}$$

else choose $c = \lfloor (a+b)/2 \rfloor$ and evaluate:

$$\begin{aligned} G(a,b) &\leftarrow G(a,c)G(c,b) \\ Q(a,b) &\leftarrow Q(a,c)Q(c,b) \\ P(a,b) &\leftarrow P(a,c)Q(c,b) + P(c,b)G(a,c) \end{aligned}$$

For the Chudnovsky series we take:

$$\begin{aligned} g(n) &= (2n-1)(6n-5)(6n-1) \\ a(n) &= (-1)^n(U+Vn) \\ q(n) &= n^3C^3/24 \end{aligned}$$

We get then

$$\pi \approx \frac{Q(0,N)}{12P(0,N)+12AQ(0,N)}C^{3/2}$$

2.2 Improvements

2.2.1 Common factor reduction

The result is not modified if common factors are removed from the numerator and the denominator. The recursion becomes:

$$\begin{array}{rcl} D & \leftarrow & \gcd(G_1(a,c), Q_1(c,b)) \\ G_D & \leftarrow & G_1(a,c)/D \\ Q_D & \leftarrow & Q_1(c,b)/D \\ G_1(a,b) & \leftarrow & G_D G_1(c,b) \\ Q_1(a,b) & \leftarrow & Q_1(a,c)Q_D \\ P_1(a,b) & \leftarrow & P_1(a,c)Q_D + P_1(c,b)G_D \end{array}$$

The gcd (Greatest Common Divisor) is computed by explicitly storing the factorization of G and Q in small prime factors at every step. The factorization is computed by using a sieve [3]. In [3] the sieve takes a memory of $O(N\log(N))$ which is very large. Our implementation only uses a memory of $O(N^{1/2}\log(N))$, which is negligible compared to the other memory use.

As

$$\frac{(6n)!}{(n!)^3(3n)!} = \prod_{k=1}^{n} \frac{24(2k-1)(6k-5)(6k-1)}{k^3}$$

is an integer, it is useful to note that the final denominator Q(0, N) of the sum divides C^{3N} when all the common factors have been removed.

So the maximum size of the final $P(0,N)$ can easily be evaluated. Here it is $\log(C)/\log(C/12) \approx 1.22$ times the size of the final result.

2.2.2 Precomputed powers of C^3

It is not efficient to recompute the powers of $C^3/24$ at every step of the recursion. So we precompute them separately.

Another point is that the final value of Q is known to divide C^{3N}. So in order to have a bounded size result, it is interesting to precompute the powers of C^3 instead of $C^3/24$. Some space is lost for the storage of G, but the overall memory usage is smaller.

The recursion becomes:

$$\begin{array}{rcl} g_2(n) & = & 24(2n-1)(6n-5)(6n-1) \\ a_2(n) & = & (-1)^n(U+Vn) \\ q_2(n) & = & n^3 \end{array}$$

$$\begin{aligned}
D &\leftarrow \gcd(G_2(a,c), Q_2(c,b)) \\
G_D &\leftarrow G_2(a,c)/D \\
Q_D &\leftarrow Q_2(c,b)/D \\
G_2(a,b) &\leftarrow G_D G(c,b) \\
Q_2(a,b) &\leftarrow Q_2(a,c) Q_D \\
P_2(a,b) &\leftarrow P_2(a,c) Q_D C^{3(b-c)} + P_2(c,b) G_D.
\end{aligned}$$

C^{3n} needs only be computed for $\log_2(N)$ exponents if N is a power of two. If N is not a power of two, it is also possible to use $\lceil \log_2(N) \rceil$ exponents by choosing c correctly at every step.

2.3 Multi-threading

When the operands are small enough, different parts of the binary splitting recursion are executed in different threads to optimally use the CPU cores. When the operands get bigger, a single thread is used, but the large multiplications are multi-threaded.

2.4 Restartability

When operands are stored on disk, each step of the computation is implemented so that it is restartable. It means that if the program is stopped (for example because of a power loss or a hardware failure), it can be restarted from a known synchronization point.

3 Verification

3.1 Binary result

The usual way is to get the same result using a different formula. Then it is highly unlikely that a software or hardware bug modifies the result the same way.

Our original plan was to use the Ramanujan formula which is very similar to the Chudnovsky one, but less efficient (8 digits per term instead of 14 digits per term). However, a hard disk failure on a second PC doing the verification delayed it and we did not wanted to spend more time on it.

Instead of redoing the full computation, we decided to only compute the last binary digits of the result using the Borwein-Bailey-Plouffe algorithm [8] with a formula we found which is slightly faster (and easier to prove) than the original BBP formula [9].

Of course, verifying the last digits only demonstrates that it is very unlikely that the overall algorithm is incorrect, but hardware errors can still have modified the previous digits. So we used a second proof relying on the fact that the last computation step of the Chudnovsky formula is implemented as a floating point multiplication of two N limbs floating point numbers C_1 and C_2. We suppose here that the floating point results are represented as size N integers. Then:

$$\begin{array}{rcl} R & = & C_1 C_2 \quad (R \text{ has } 2N \text{ limbs}) \\ \pi B^N & \approx & \lfloor R/B^N \rfloor \quad (\text{truncation to take the high } N \text{ limbs}) \end{array}$$

where $C_2/B^N \approx C^{3/2}$ and C_1/B^N is the result of the binary splitting.

If we assume that C_1 and C_2 are random like, then the last digits of the result of the multiplication depend on all the input digits of C_1 and C_2.

More precisely, if there is an additive error eB^k in C_1 with $-B < e < B$ and $0 \leq k < N$, then

$$R = (C_1 + eB^k)C_2 = C_1 C_2 + eC_2.B^k$$

The fact that $C2$ has random like digits ensures that digits near the position N in R are modified (multiplying eC_2 by B^k can be seen as a left shift by k limbs of eC_2).

So it remains to prove that the multiplication itself was done without error. To do it, we check the $2N$ limb result R (before doing the truncation) modulo a small number p :

$$\begin{array}{rcl} c_1 & \leftarrow & C_1 \bmod p \\ c_2 & \leftarrow & C_2 \bmod p \\ r_1 & \leftarrow & c_1 c_2 \bmod p \\ r_2 & \leftarrow & R \bmod p \end{array}$$

The check consists in verifying that $r1 = r2$. p is chosen as a prime number so that the modulus depends on all the limbs of the input numbers. Its size was fixed to 64 bit which gives a negligible probability of not catching an error.

3.2 Base conversion

The binary to base 10 conversion also needs to be verified. The common way is to convert back the result from base 10 to binary and to verify that it matches the original binary result.

We implemented this method, but it would have doubled the computation time. Instead we did a more efficient (but less direct) verification:

Let R be the N limb floating point binary result and R_2 be the base 10 result expressed as a N_2 limb floating point number in base B_2 ($B_2 = 10^{19}$ in our case).

Assuming that R_2 is normalized between $1/B_2$ and 1, we want to verify that

$$\lfloor R B_2^{N_2} \rfloor = \lfloor R_2 B_2^{N_2} \rfloor$$

neglecting possible rounding errors. Doing the full computation would need converting back $R_2 B_2^{N_2}$ to integer, which does not save time. So we check the equality modulo a small prime p.

We compute $r_1 = \lfloor R B_2^{N_2} \rfloor \bmod p$ the usual way (cost of $O(M(N))$).

$r_2 = \lfloor R_2 B_2^{N_2} \rfloor \bmod p$ is evaluated by considering $\lfloor R_2 B_2^{N_2} \rfloor$ as a polynomial in B_2 evaluated modulo p using the Horner algorithm. It has a cost of $O(N)$ assuming that single word operations have a running time of $O(1)$.

Then the check consists in verifying that $r_1 = r_2$.

4 Fast multiplication algorithm

4.1 Small sizes

For small sizes (a few limbs), the naive $O(n^2)$ method is used. For larger sizes (up to a few tens of limbs), the Karatsuba method is used ($O(n^{\log_2(3)})$ running time).

4.2 Large sizes (operands fit in RAM)

For larger sizes, the multiplication is implemented as a cyclic convolution of polynomials. The cyclic convolution of polynomials is implemented with floating point Discrete Fourier Transforms (DFT) [5].

Instead of doing more complicated real valued DFTs, we do complex DFTs and use a dyadic multiplication instead of the point-wise multiplication. Negacyclic convolutions use a similar trick [2].

To reduce the memory usage, large multiplications giving a result of a size of $2N$ limbs are splitted in two multiplications modulo $B^N - 1$ (cyclic convolution) and modulo $B^N + 1$ (negacyclic convolution). The two results are combined using the Chinese Remainder Theorem (CRT) to get a multiplication modulo $B^{2N} - 1$.

Although it is possible to design the floating point DFT convolution with guaranteed error bounds [4] [1], it severely reduces its performance. Instead of doing that, we use the fact that the input numbers are random like and we use a DFT limb size found empirically. We systematically use a fast modulo $(B - 1)$ checksum for the cyclic convolution to detect most rounding errors. We made the assumption that possible rounding error left would be detected by the verification steps.

A DFT of size $N = n_1 n_2$ is evaluated by doing n_2 DFT of size n_1, N multiplications by complex exponentials (they are usually called *twiddle factors*) and n_1 DFT of size n_2. The method can be applied recursively [6].

When N is small, we use n_1 or $n_2 = 2, 3, 5$ or 7 which give well known *Fast Fourier Transform* algorithms using *decimation in time* or *decimation in frequency.* Using other factors than 2 allows N to be chosen very close to the actual size of the result, which is important to reduce the memory usage and to increase the speed.

When N is large, a better CPU cache usage is obtained by choosing factors so that each DFT fits in the L1 cache. The smaller DFTs are evaluated using several threads to use all the CPU cores. Since the task is mostly I/O bound, it does not scale linearly with the number of CPUs.

Another important detail is how to compute the complex exponentials. For the small DFTs fitting the L1 cache, a cache is used to store all the possible values. The twiddle factors of the large DFTs are evaluated on the fly because it would use too much memory to save them. It comes from the fact that the binary splitting uses multiplications of very different sizes. The twiddle factors are evaluated using multiplication trees ensuring there is little error propagation. The intermediate complex values are stored using 80 bit floating point numbers which are natively supported by x86 CPUs.

All the other floating point computations are achieved using 64 bit floating point numbers. All the operations are vectorized using the x86 SSE3 instruction set.

4.3 Huge sizes (operands are on disk)

4.3.1 Main parameters

For huge sizes (hundreds gigabytes as of year 2009), the main goal was to reduce the amount of disk I/O. A key parameter is what we call the *DFT expansion ratio* r, which is the ratio between the DFT word size and the DFT limb size.

For example, for DFTs of 2^{27} complex samples, the expansion ratio is about 4.5.

The DFT expansion ratio cannot be smaller than two because the result of the convolution needs at least twice the size of the DFT limb size. A ratio close to 2 can be obtained by using large DFT word sizes. Using the hardware of year 2009, it is easier to do it using a Number Theoretic Transform (NTT) which is a DFT in $\mathbb{Z}/n\mathbb{Z}$ where n is chosen so that there are primitive roots of unity of order N, where N is the size of the NTT.

A possible choice is to choose $n = 2^{2^k} + 1$, which gives the Schönhage-Strassen multiplication algorithm. However, it is difficult to make it efficient for all the result sizes N, although it can be improved by using the Chinese Remainder Theorem (CRT) as described in [7].

Another choice is to use several small primes n and to combine them with the CRT. The advantage is that the same multi-threading and cache optimizations as for the floating point DFT are reused and that it is possible to find a NTT size N very close to the needed result size. The drawback is that the computation involves modular arithmetic on small numbers which is not as efficient as floating point arithmetic on x86 CPUs of the year 2009.

However, it turns out that latest x86 CPUs are very fast at doing 64 bit by 64 bit multiplications giving a 64 bit result. We improved the NTT speed by using the fact that most modular multiplications are modular multiplications by a constant. This trick is attributed to Victor Shoup and is described in section 4.2 of [11].

We choose 8 64 bit moduli, which gives a DFT expansion ratio of about 2.2 for NTTs of a few terabytes. Using more moduli is possible to get closer to 2, but it complicates the final CRT step.

4.3.2 NTT convolution

The NTT size N is split in two factors n_1 and n_2. n_2 is chosen as large as possible with the constraint that a NTT of size n_2 can be done in memory. The resulting convolution in split in 3 steps:

1. n_2 NTTs of size n_1 on both operands, multiplication by the twiddle factors
2. n_1 NTTs of size n_2 on both operands, pointwise multiplication, n_1 inverse NTT of size n_2
3. multiplication by the twiddle factors, n_2 NTTs of size n_1 on the result.

Step (2) involves only linear disk access, which are fast. (1) and (3) involves more disk seeks as n_1 gets bigger because the NTTs must be done using samples separated by n_2 samples.

If n_1 is too big, it is split again in two steps, which gives a 5 step convolution algorithm.

For the 3 step algorithm, the amount of disk I/O is $12r + 4$ assuming two input operands of unit size and a DFT expansion ratio of r.

The needed disk space, assuming the operands are destroyed is $4r + 1$.

If the disk space is limited, we use the same CRT trick as for the floating point DFT by doing two multiplications modulo $B^N - 1$ and $B^N + 1$. The amount disk I/O is $12r + 10$ but the needed disk space is lowered to $2r + 2$.

The disk space can be further reduced at the expense of more I/Os by using more than two moduli.

5 Other operations

5.1 Division

The evaluation of the Chudnovsky formula involves a lot of integer divisions because of the GCD trick. So having a good division algorithm is critical. We implemented the one described in [1] using the concept of *middle product*. Then the floating point inversion has a cost of 2 multiplications and the division a cost of 2.5 multiplications.

5.2 Base conversion

The usual base conversion algorithm for huge numbers involves recursively dividing by powers of the new base. The running time is $0.5D(n/2)\log_2(n)$ [1] when n is a power of two and $D(n)$ is the time needed to divide a $2n$ limb number by a n limb number.

We implemented a better algorithm by using the Bernstein scaled remainder tree and the middle product concept, as suggested in exercise 1.32 of [1]. The running time becomes $0.5M(n/2)\log_2(n)$.

6 Pi computation parameters

We describe the various parameters to compute 2700 billion decimal digits of π.

6.1 Global memory use

The full π computation, including the base conversion uses a total memory of 6.42 times the size of the final result. Here the result is 1.12 TB big, hence at least 7.2 TB of mass storage is needed.

It is possible to further reduce the memory usage by using more moduli in the large multiplications (see section 4.2).

6.2 Hardware

A standard desktop PC was used, using a Core i7 CPU at 2.93 GHz. This CPU contains 4 physical cores.

We put only 6 GiB of RAM to reduce the cost. It means the amount of RAM is about 170 times smaller than the size of the final result, so the I/O performance of the mass storage is critical. Unfortunately, the RAM had no ECC (Error Correcting Code), so random bit errors could not be corrected nor detected. Since the computation lasted more than 100 days, such errors were likely [12]. Hopefully, the computations included verification steps which could detect such errors.

For the mass storage, five 1.5 TB hard disks were used. The aggregated peak I/O speed is about 500 MB/s.

6.3 Computation time

- Computation of the binary digits: 103 days
- Verification of the binary digits: 13 days
- Conversion to base 10: 12 days
- Verification of the conversion: 3 days

7 Ideas for future work

Here are some potentially interesting topics:

- Study the theoretical running time and memory usage of the evaluation of the Chudnovsky series using the binary splitting and the mentioned improvements.
- See if the Schönhage-Strassen multiplication can be faster than the NTT with small moduli when using mass storage.
- Speed up the computations using a network of PCs and/or many-core CPUs.
- Lower the memory usage of the π computation.

8 Conclusion

We showed that a careful implementation of the latest arbitrary-precision arithmetic algorithms allows computations on numbers in the terabyte range with inexpensive hardware resources.

References

[1] Richard Brent and Paul Zimmermann, *Modern Computer Arithmetic*, version 0.4, November 2009,
`http://www.loria.fr/~zimmerma/mca/mca-0.4.pdf` .

[2] Richard Crandall, *Integer convolution via split-radix fast Galois transform*, Feb 1999,
`http://people.reed.edu/~crandall/papers/confgt.pdf` .

[3] Hanhong Xue, `gmp-chudnovsky.c` program to compute the digits of Pi using the GMP library, `http://gmplib.org/pi-with-gmp.html` .

[4] Richard P. Brent, Colin Percival, and Paul Zimmermann, *Error bounds on complex floating-point multiplication. Mathematics of Computation*, 76(259):1469-1481, 2007.

[5] Arnold Schönhage and Volker Strassen, *Schnelle Multiplikation großer Zahlen*, Computing 7:281-292, 1971.

[6] Cooley, James W., and John W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput. 19, 297-301 (1965).

[7] Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann, *A GMP-based Implementation of Schönhage-Strassen's Large Integer Multiplication Algorithm*, ISSAC'07.

[8] Bailey David H., Borwein Peter B., and Plouffe Simon, *On the Rapid Computation of Various Polylogarithmic Constants*, April 1997, Mathematics of Computation 66 (218): 903-913.

[9] Fabrice Bellard, *A new formula to compute the n'th binary digit of Pi*, 1997, `http://bellard.org/pi/pi_bin.pdf` .

[10] D. V. Chudnovsky and G. V. Chudnovsky, *Approximations and complex multiplication according to Ramanujan*, in Ramanujan Revisited, Academic Press Inc., Boston, (1988), p. 375-396 & p. 468-472.

[11] Tommy Färnqvist, *Number Theory Meets Cache Locality - Efficient Implementation of a Small Prime FFT for the GNU Multiple Precision Arithmetic Library*, 2005,
`http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2005/rapporter05/farnqvist_tommy_05091.pdf` .

[12] Bianca Schroeder, Eduardo Pinheiro, Wolf-Dietrich Weber, *DRAM Errors in the Wild: A Large-Scale Field Study*, SIGMETRICS/Performance'09, 2009.