A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm

Paul Zimmermann
INRIA Lorraine/LORIA, Nancy, France
(joint work with Torbjörn Granlund, Alexander Kruppa and Pierrick Gaudry)

Second Workshop on High-Dimensional Approximation (HDA07), 19-22 February 2007

Context

Question: Given two N-bit integers, how fast can we multiply them?

- complex floating-point FFT: $O(n \log^* n)$ where
  $\log^* n = \log(n) \log\log(n) \log\log\log(n) \ldots$
- FFT mod $2^N + 1$ (called SSA here): $O(n \log(n) \log\log(n))$

Schnelle Multiplikation großer Zahlen, A. Schönhage and V. Strassen, Computing, 1971.

	transform length	coeff size	transform cost	pointwise cost
	K	ℓ	$K \log K M(\ell)$	$K M(\ell)$
complex FFT	$\frac{n}{\log n}$	$\log n$	$nM(\log n)$	$\frac{n}{\log n} M(\log n)$
SSA	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n} \log(n) O(\sqrt{n})$	$\sqrt{n} M(\sqrt{n})$

Complex FFT

Efficient implementations in Prime95 (G. Woltman, GIMPS), Glucas (G. Ballester Valor).

Used to find/check the 44th (known) Mersenne prime:

$$2^{32,582,657} - 1 \qquad \text{(9,808,358 digits)}$$

The Electronic Frontier Foundation (EFF) offers $100,000 to the first individual or group who discovers a prime number with at least 10,000,000 decimal digits.

If coefficients are represented in signed-digit notations:

$$A = \sum_{i=0}^{n-1} a_i \beta^i,$$

where $-\beta/2 < a_i \leq \beta/2$, then a product coefficient:

$$c_k = \sum_{i+j=k} a_i b_j$$

exceed $\alpha n \beta^2$ with small probability.

Motivation

SSA is implemented in GNU MP since version 3.1 (released in August 2000).

In July 2005, Allan Steel published a web page `http://magma.maths.usyd.edu.au/users/allan/intmult.html`:

> *Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication*

Visits of Torbjörn Granlund in March-April, November-December 2006.

Schönhage-Strassen's Algorithm

$$\mathbb{Z}$$

$$\Downarrow$$

$$R_N := \mathbb{Z}/(2^N + 1)\mathbb{Z}$$

$$\Downarrow$$

$$\mathbb{Z}[x] \bmod (x^K + 1)$$

$$\Downarrow$$

$$R_n[x] \bmod (x^K + 1)$$

$$\Downarrow$$

$$R_n$$

From R_N to $\mathbb{Z}[x] \bmod (x^K + 1)$

Write $N = K \cdot \ell$ where $K = 2^k$ (transform length).

Interpret $a \in [0, 2^N]$ as $A(2^\ell)$ where:

$$A(x) = \sum_{i=0}^{K-1} a_i x^i.$$

Idem for $b \in [0, 2^N]$:

$$B(x) = \sum_{i=0}^{K-1} b_i x^i.$$

$a = A(2^\ell)$ and $b = B(2^\ell)$ thus $ab \equiv C(2^\ell) \bmod (2^N + 1)$ where $C(x) = A(x)B(x) \bmod (x^N + 1)$.

From $\mathbb{Z}[x] \bmod (x^K+1)$ to $R_n[x] \bmod (x^K+1)$

$$C(x) := A(x)B(x) \bmod (x^K+1)$$

$$= (c_0 - c_K) + (c_1 - c_{K+1})x + \cdots + (c_{K-2} - c_{2K-2})x^{K-2} + c_{K-1}x^{K-1}$$

where:

$$c_m = \sum_{i+j=m} a_i b_j$$

The coefficients of $C(x)$ take at most $2^k \cdot 2^{2\ell}$ values: it suffices to compute them mod 2^n+1 with:

$$n \geq 2\ell + k.$$

Arithmetic modulo $2^n + 1$

A residue modulo $2^n + 1$ is represented by:

$$a = (a_m, a_{m-1}, \ldots, a_0),$$

with $0 \leq a_i < 2^w$ for $0 \leq i < m$, and $0 \leq a_m \leq 1$ ($w = 32$ or $w = 64$).

GMP syntax:

```
c = a[m] + b[m] + mpn_add_n (r, a, b, m);
r[m] = (r[0] < c);
MPN_DECR_U (r, m + 1, c - r[m]);

c = a[m] - b[m] - mpn_sub_n (r, a, b, m);
r[m] = (c == 1);
MPN_INCR_U (r, m + 1, r[m] - c);
```

Cache locality in the Fourier transforms

The basic operation is the *butterfly*:

$$\begin{cases} a & \leftarrow & a + \omega b \\ b & \leftarrow & a - \omega b \end{cases} \quad \text{or} \quad \begin{cases} a & \leftarrow & a + b \\ b & \leftarrow & (a - b)\omega \end{cases}$$

- the Belgian transform
- higher radix transform
- Bailey's 4-step algorithm

The Belgian Transform

Parametrizable behavioral IP module for a data-localized low-power FFT, E. Brockmeyer, C. Ghez, J. D'Eer, F. Catthoor and H. De Man, IEEE Workshop on Signal Processing Systems, 1999. (thanks Markus!)

Main idea: when we perform a butterfly, we reuse at least one of the two outputs in the next butterfly.

Guarantees less than 50% cache misses.

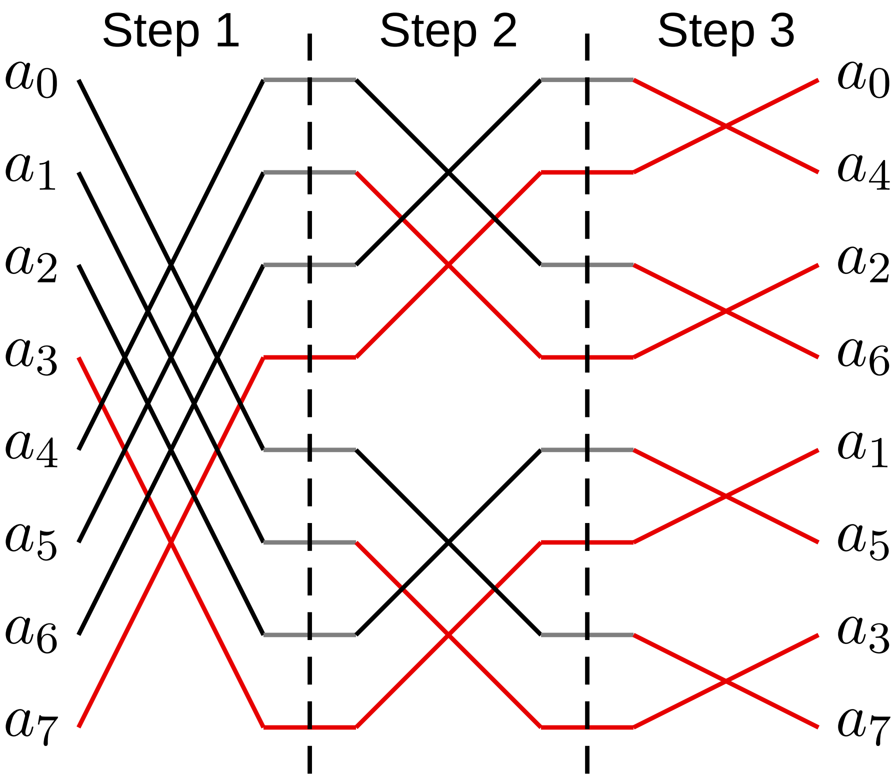
The Belgian Transform

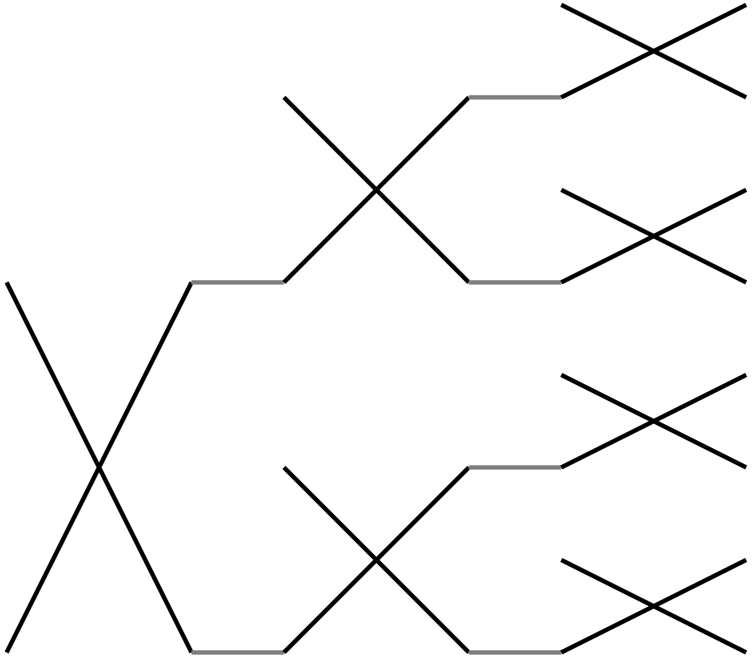
```
void fft(int n_stage) {
  int size = 1<<(n_stage-1);
  for (int i=0; i<size/2; i++) { /* initial 2 stage 0 butterflies */
    int stage0_bf = bitrev(i, n_stage-2);
    Radix2Butterfly(&mem[stage0_bf], &mem[stage0_bf+size]);
    stage0_bf += (size/2);
    Radix2Butterfly(&mem[stage0_bf], &mem[stage0_bf+size]);
    if ((stage0_bf-size/2) >= 0) {
      unsigned int branch_ref = 1;
      int offset = 0, upper = stage0_bf;
      for (;branch_reg != 0;){
                                        /* upper branches */
        for (;((size>1)&&((upper-size/2)>=0));){
          size /= 2;
          Radix2Butterfly(&mem[offset+upper-size],
                          &mem[offset+upper]);
          upper -= size;
          branch_reg = (branch_reg << 1)|1;
        }
        for (;((branch_reg&1)==0);){        /* trace back */
          branch_reg = branch_reg >> 1;
          upper += size;
          size *= 2;
          offset -= (size);
        }
        branch_reg ^= 1;                /* lower branches */
        if (branch_reg != 0){
           offset += size*2;
           Radix2Butterfly(&mem[offset+upper],
                           &mem[offset+upper+size]);
        }}}}}
```

The Belgian Transform (recursive version)

```
BelgianFFT(A, k)
    K = 2^{k-1}
    for i := 0 to K-1
        TreeBfy(A, BitReverse(i, k-1), 1+ord_2(i+1), K)

TreeBfy(A, index, depth, stride)
    Bfy(A[index], A[index+stride])
    if depth > 1
        TreeBfy(A, index-stride/2, depth-1, stride/2)
        TreeBfy(A, index+stride/2, depth-1, stride/2)
```

The FFT circuit of length 8



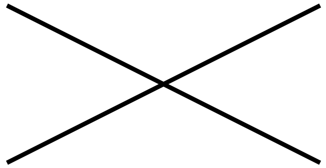


Radix 4

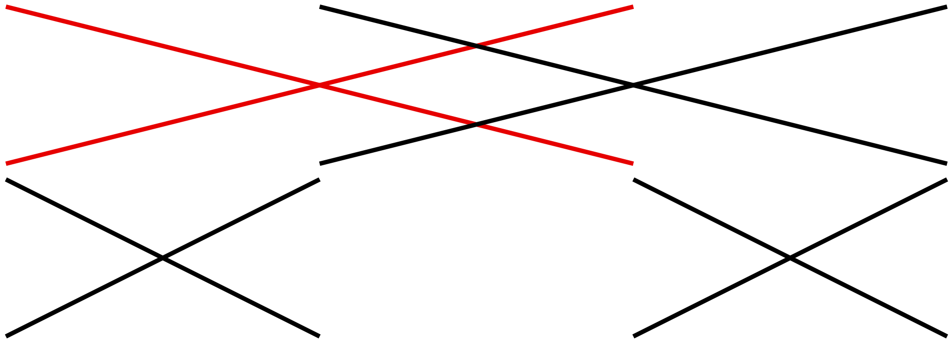
```
Radix4FFT(A, index, k, omega)
    if k == 0
        return;
    if k == 1
        Bfy(A[index], A[index+1], 1);
        return;

    K1 = 2^{k-1}
    K2 = 2^{k-2}

    for j = 0 to K2-1 do
        Bfy(A, index+j,    index+j+K1,    omega^j);
        Bfy(A, index+j+K2, index+j+K1+K2, omega^(j+K2));
        Bfy(A, index+j,    index+j+K2,    omega^(2*j));
        Bfy(A, index+j+K1, index+j+K1+K2, omega^(2*j));
    end for;

    Radix4FFTrec(A, index,       k-2, omega^4);
    Radix4FFTrec(A, index+K2,    k-2, omega^4);
    Radix4FFTrec(A, index+K1,    k-2, omega^4);
    Radix4FFTrec(A, index+K1+K2, k-2, omega^4);
```

Higher Radix Transform

Classical FFT: radix 2, 2 inputs/outputs, 1 butterfly:



Radix 4: 4 inputs/outputs, 2×2 butterflies.



Radix 2^t: 2^t inputs/outputs, $t \times 2^{t-1}$ butterflies.

Bailey's 4-step Algorithm

Let $K = 2^k$ be the FFT length, where $k = k_1 + k_2$:

1. Perform 2^{k_2} transforms of length 2^{k_1};
2. Multiply the data by weights;
3. Perform 2^{k_1} transforms of length 2^{k_2}.

a_0	a_1	a_2	a_3
a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}
a_{12}	a_{13}	a_{14}	a_{15}

Bailey's 4-step Algorithm

```
Bailey(A, k, k1, k2, omega)
    K1 = 2^k1;
    K2 = 2^k2;

    // Phase 1:
    for i = 0 to K2-1 do
        for j := 0 to K1-1 do
            B[j] = A[i+K2*j]
        twistedFFT(B, i, k1, k, omega);
        for j = 0 to K1-1 do
            A[i+K2*j] = B[j];

    // No Phase 2!

    // Phase 3:
    for j := 0 to K1-1 do
        for i = 0 to K2-1 do
            B[i] = A[i+K2*j]
        FFT(B, k2, omega^K1);
        for i = 0 to K2-1 do
            A[i+K2*j] = B[i];
```

Fermat and Mersenne Transforms

Fermat Transform: modulo $2^N + 1$ (negacyclic convolution).

$\ominus$ weighted transform: slightly more expensive.

Mersenne Transform: modulo $2^N - 1$ (cyclic convolution).

$\ominus$ does not work recursively.

$\oplus$ can use twice the FFT length because no $2K$-th root of unity is needed

The $\sqrt{2}$ Trick

Credited to Schönhage by Bernstein.

A product modulo $2^N \pm 1$ reduces to $K = 2^k$ products modulo $2^{N'} + 1$.

$\omega = 2^{2N'/K}$ is the primitive Kth root of unity.

$\theta = 2^{N'/K}$ is the weight signal (Discrete Weighted Transform).

Fermat transform: K must divide N'.

Mersenne transform: K must divide $2N'$.

$$\left(2^{3N'/4} - 2^{N'/4}\right)^2 \equiv 2 \pmod{2^{N'} + 1}.$$

Fermat transform: K must divide $2N'$.

Mersenne transform: K must divide $4N'$.

$\Longrightarrow$ smaller pointwise products.

Integer Multiplication

Problem: multiply two m-bit numbers.

Original SSA: multiply modulo $2^N + 1$ for $N \geq 2m$ (GMP 4.1.4).

GMP 4.2.1: $2^{2N} + 1$ and $2^{3N} + 1$ for $5N \geq 2m$, and reconstruct by CRT.

Generalization: $2^{aN} + 1$ and $2^{bN} - 1$.

Lemma. Let a, b be two positive integers. Then at least one of $\gcd(2^a + 1, 2^b - 1)$ and $\gcd(2^a - 1, 2^b + 1)$ is 1.

Example: $\gcd(2^{17} + 1, 2^{10} - 1) = 3$, $\gcd(2^{17} - 1, 2^{10} = 1) = 1$.

Proof: study the length mod 3 of the subtractive-Euclidean sequence of (a, b).

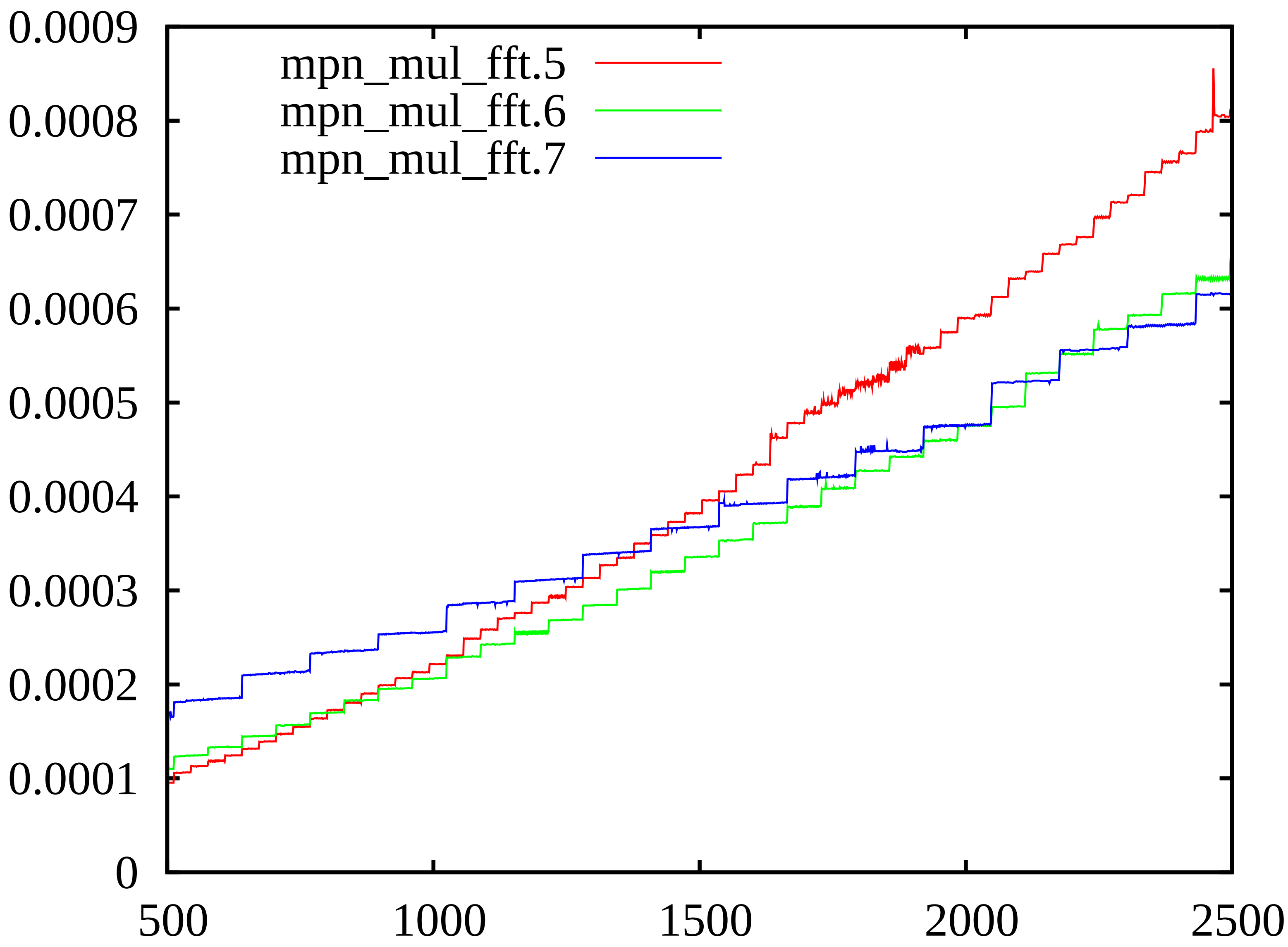
Current code uses $1 \leq a \leq 7$, and $b = 1$.

Improved Tuning mod $2^N + 1$

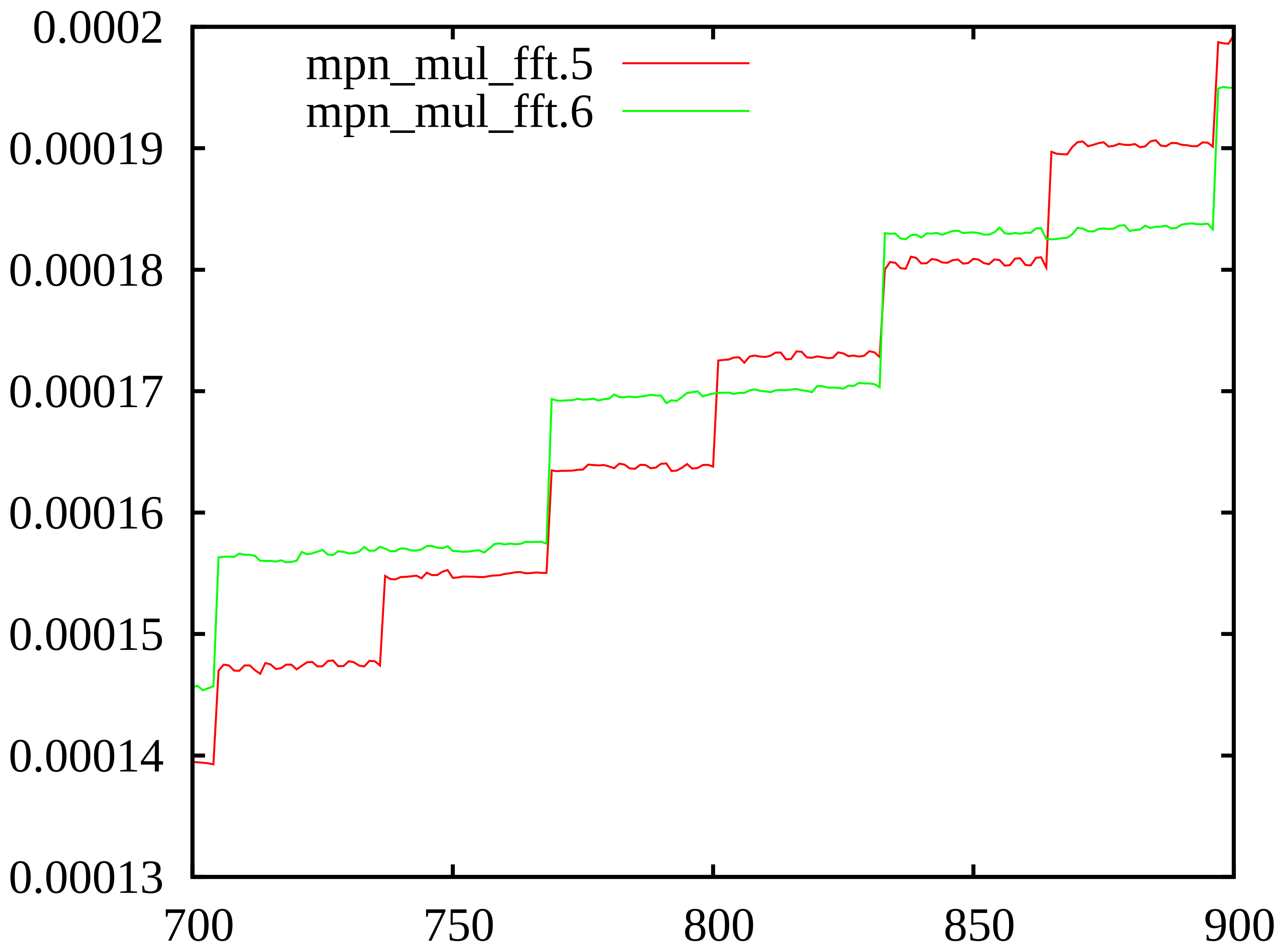
GMP 4.2.1:

```
#define MUL_FFT_TABLE  { 528, 1184, 2880, 5376, 11264, 36864, 114688,
                         327680, 1310720, 3145728, 12582912, 0 }
```

Improved Tuning mod $2^N + 1$



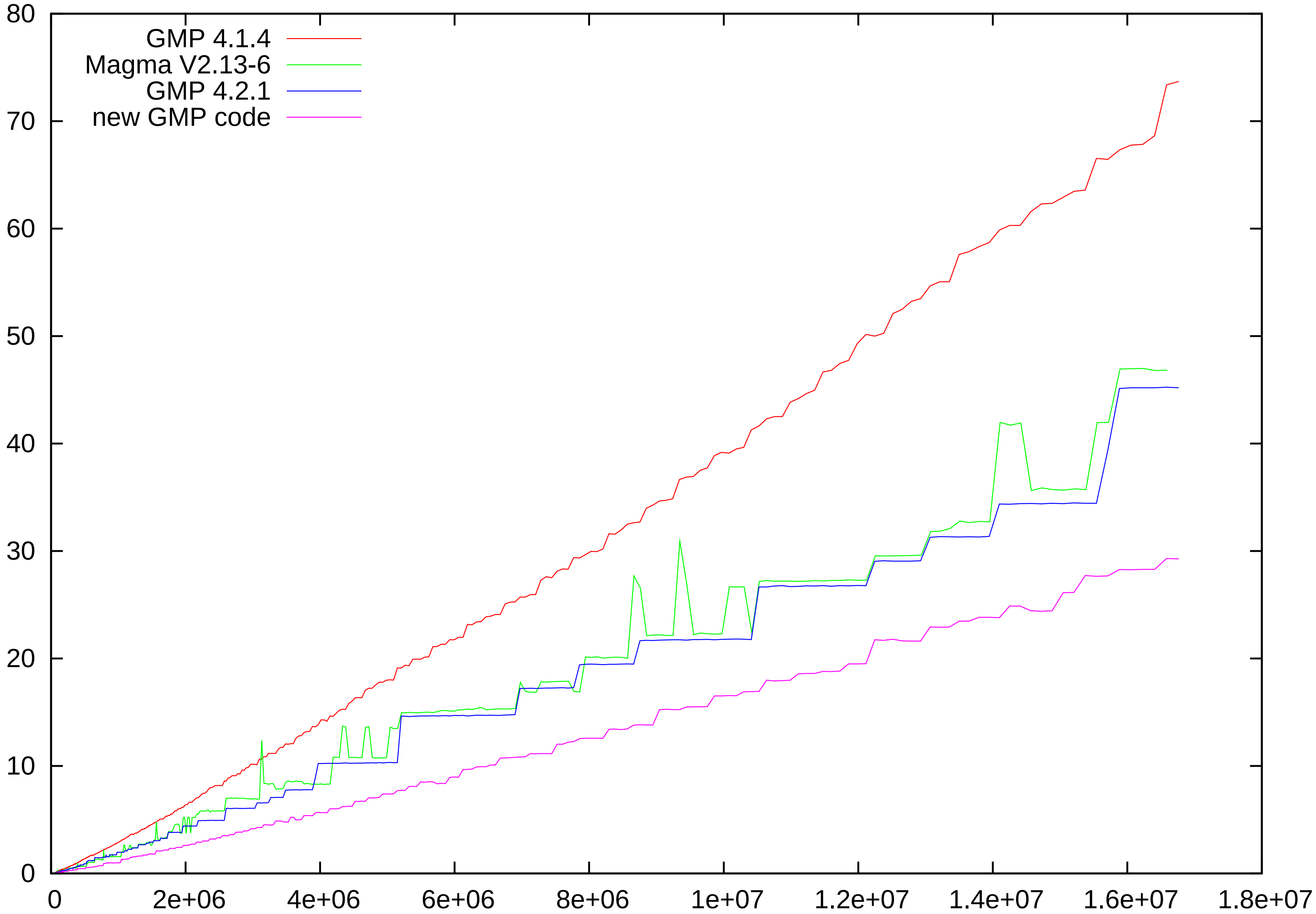
Improved Tuning mod $2^N + 1$



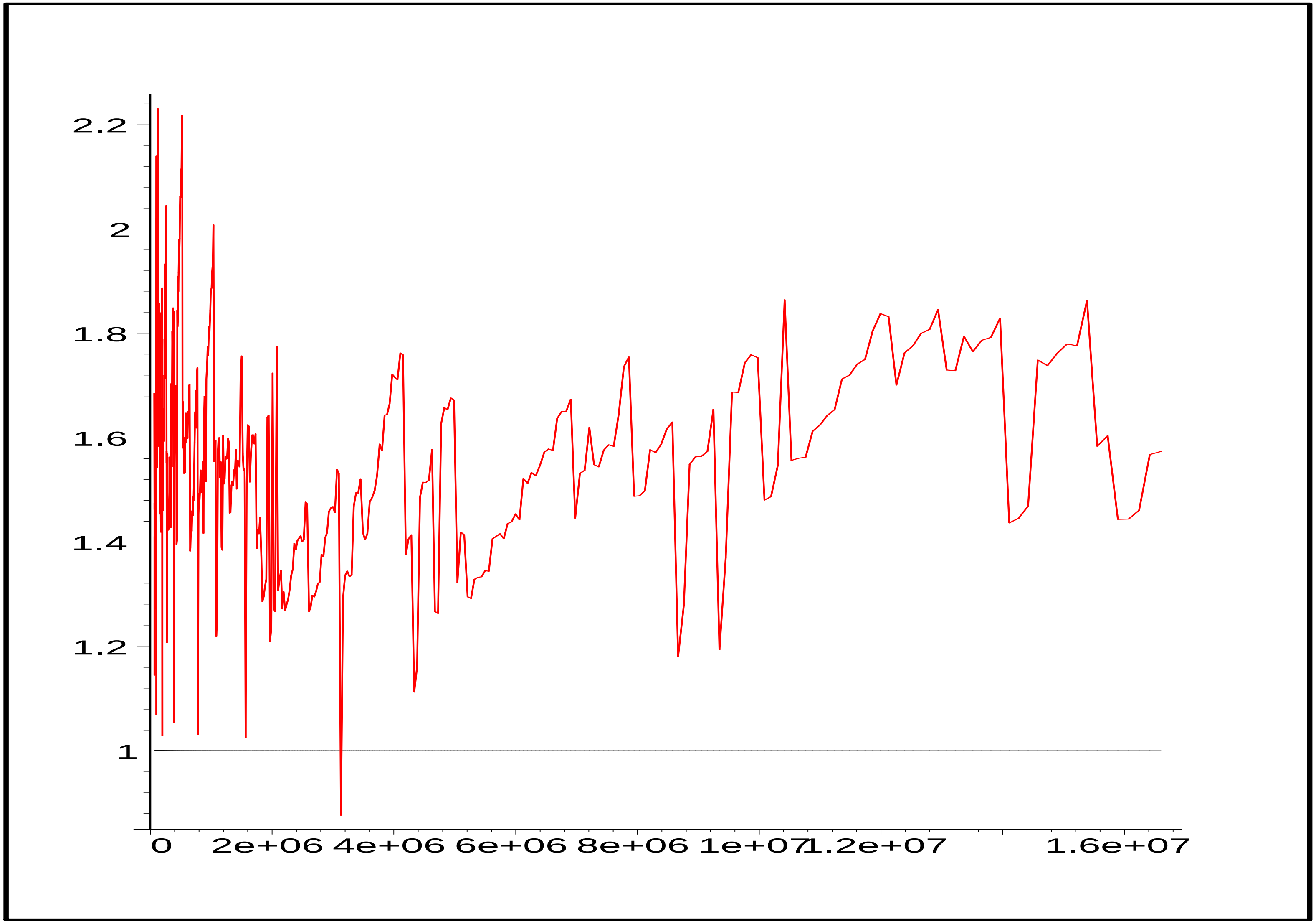
Improved Tuning mod $2^N + 1$

```
#define MUL_FFT_TABLE2 {{1, 4 /*66*/}, {401, 5 /*96*/},
     {417, 4 /*98*/}, {433, 5 /*96*/}, {865, 6 /*96*/},
     {897, 5 /*98*/}, {929, 6 /*96*/}, {2113, 7 /*97*/},
     {2177, 6 /*98*/}, {2241, 7 /*97*/}, {2305, 6 /*98*/},
     {2369, 7 /*97*/}, {3713, 8 /*93*/}, ...
```

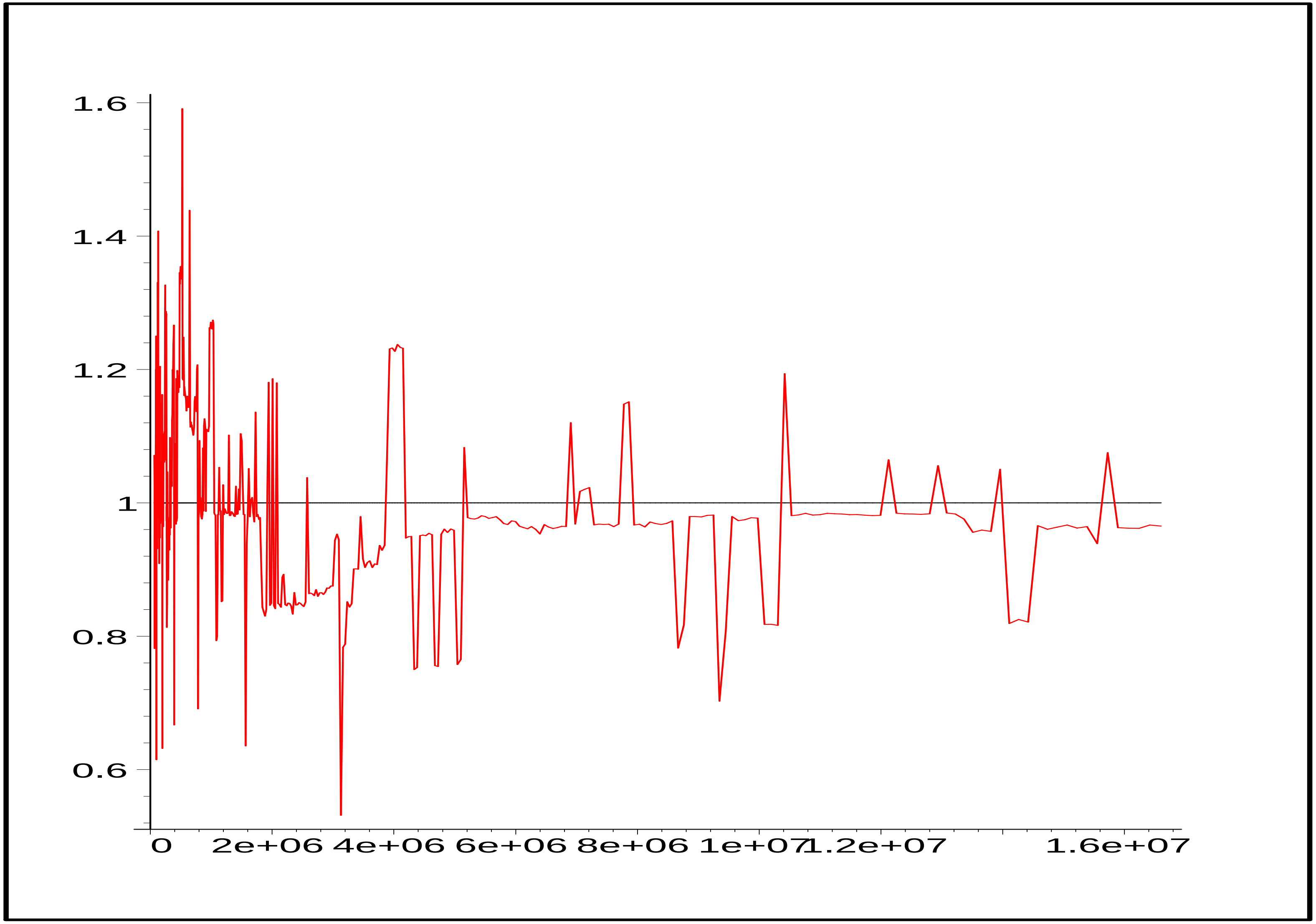
Current Timings up to 2^{30} bits, 2.4Ghz Opteron



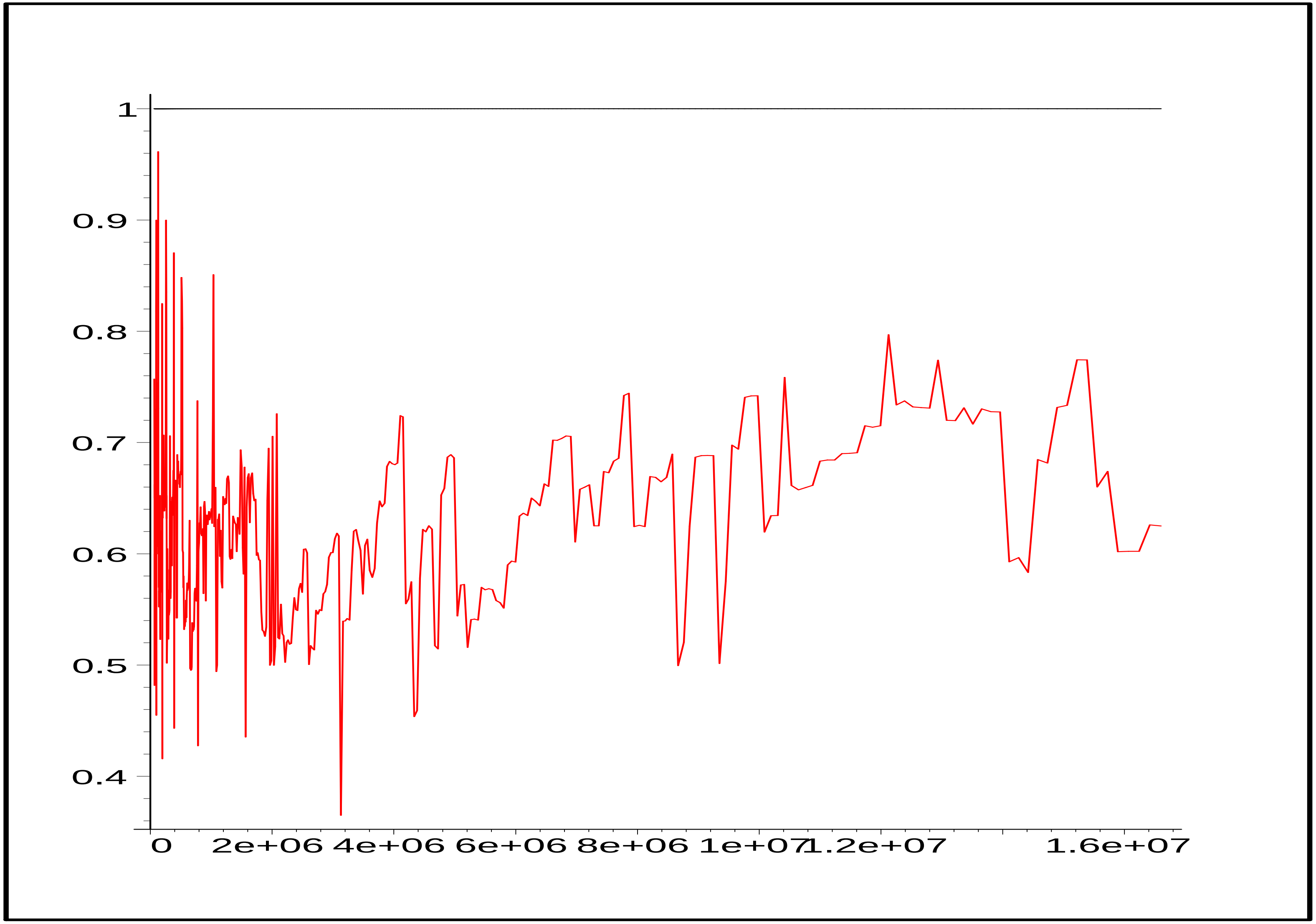
Relative Timings GMP 4.1.4 vs Magma V2.13-6



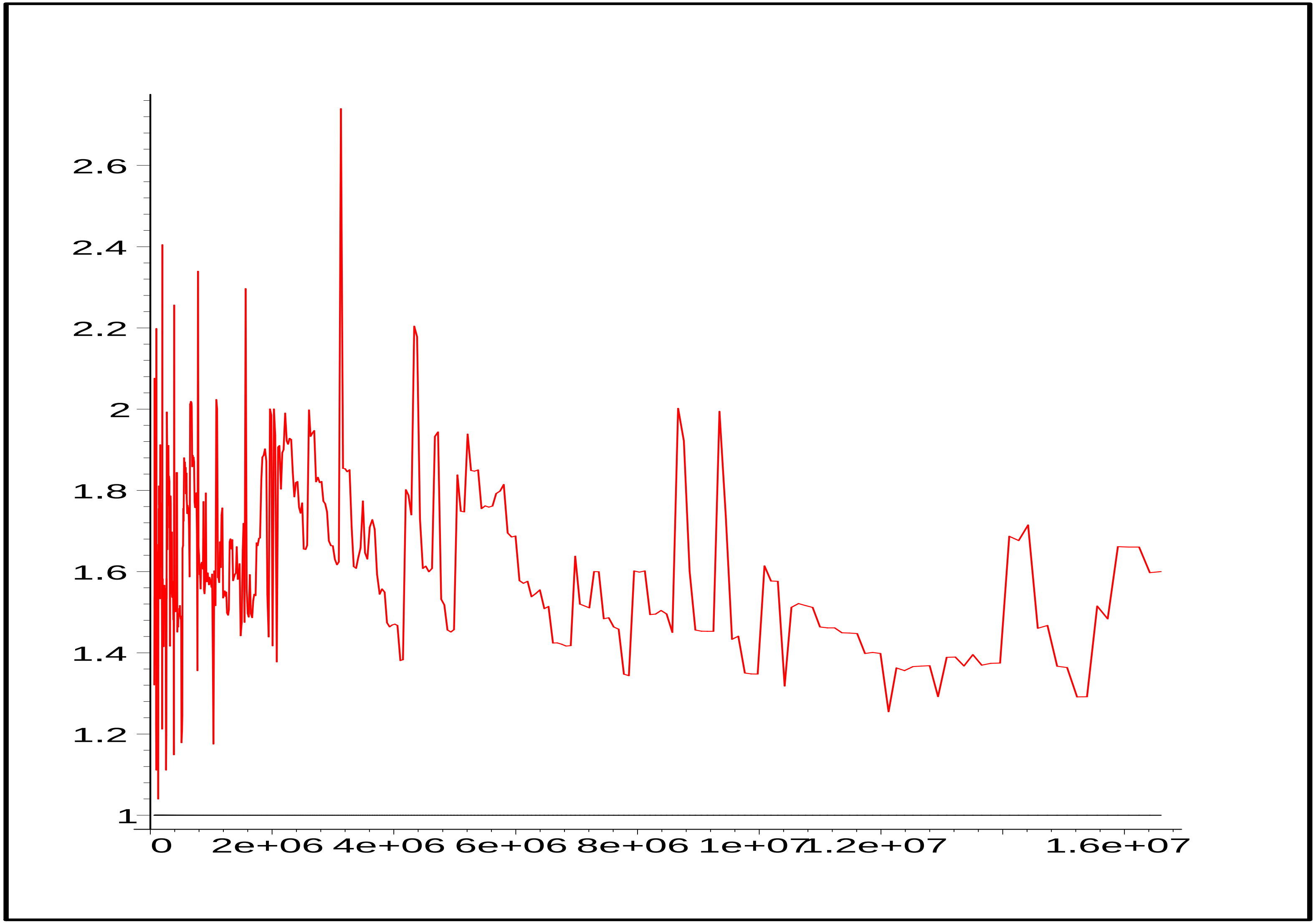
Relative Timings GMP 4.2.1 vs Magma V2.13-6



Relative Timings new GMP code vs Magma V2.13-6



Relative Timings Magma V2.13-6 vs new GMP code



Conclusion

- every 5% gain is worthwhile

Conclusion

- every 5% gain is worthwhile
- surely not the end of the story . . .

Conclusion

- every 5% gain is worthwhile
- surely not the end of the story ...
- give challenges to your colleagues!