Elsevier Editorial System(tm) for Discrete Applied Mathematics
Manuscript Draft

Manuscript Number:

Title: The Mixed Binary Euclid Algorithm

Article Type: Special Issue: LAGOS 2009

Keywords: Greatest common divisor (GCD); Parallel Complexity; Algorithms.

Corresponding Author: Dr. Sidi Mohamed SEDJELMACI, Ph.D

Corresponding Author's Institution: LIPN, University of Paris 13

First Author: Sidi Mohamed SEDJELMACI, Ph.D

Order of Authors: Sidi Mohamed SEDJELMACI, Ph.D

Abstract: We present a new GCD algorithm for two integers that combines both the Euclidean and the binary gcd approaches.
We give its worst case time analysis and
we prove that its bit-time complexity is still
$O(n^2)$ for two $n$-bit integers in the worst case. Our experimental implementation shows a clear speedup for small integers. A parallel version
matches the best presently known time complexity, namely $O(n/\log n)$ time with
$O(n^{1+\epsilon})$ processors,
for any constant $\epsilon >0$.

Manuscript

# The Mixed Binary Euclid Algorithm

Sidi Mohamed Sedjelmaci

*LIPN CNRS UMR 7030*
*Université Paris-Nord*
*Av. J.-B. Clément, 93430, Villetaneuse, France*
*Email: sms@lipn.univ-paris13.fr*

Kenneth Weber

*Department of Computer Science and Information Systems*
*Mount Union College*
*Alliance, OH 44601, USA*
*Email: weberk@muc.edu*

**Abstract**

We present a new GCD algorithm for two integers that combines both the Euclidean and the binary gcd approaches. We give its worst case time analysis and we prove that its bit-time complexity is still $O(n^2)$ for two $n$-bit integers in the worst case. Our experimental implementation shows a clear speedup for small integers. A parallel version matches the best presently known time complexity, namely $O(n/\log n)$ time with $O(n^{1+\epsilon})$ processors, for any constant $\epsilon > 0$.

*Keywords:* Greatest common divisor (GCD); Parallel Complexity; Algorithms.

## 1. Introduction

Given two integers $a$ and $b$, the greatest common divisor (GCD) of $a$ and $b$, denoted $\gcd(a, b)$, is the largest integer which divides both $a$ and $b$. Applications for GCD algorithms include computer arithmetic, integer factoring, cryptology and symbolic computation.

Most GCD algorithms follow the same idea of reducing efficiently $u$ and $v$ to $u'$ and $v'$, so that $\gcd(u, v) = \gcd(u', v')$ [14]. These transformations are applied several times till a pair $(u', 0)$ is reached. Such transformations, also called *reductions*, are studied in a general framework in [14].

For very large integers, the fastest GCD algorithms [1, 13, 17, 18] are all based on a "half-gcd" procedure and compute the GCD in $O(n \log^2 n \log \log n)$ time, where $n$ is the size of the larger input. All these algorithms are recursive in nature and switch over to some other GCD algorithm that is more efficient for small inputs when the parameters in the recursive call become small enough.

In this paper, we are interested in small and medium size integers. Usually, the euclidean and the binary GCD algorithms work very well in practice for this range of integers. In Section 2, we present a new algorithm that alternates euclidean and binary reductions, obtaining a faster overall reduction to $\gcd(u', 0)$ than would be obtained by using either reduction exclusively. We give its worst case time complexity and a multi-precision version is suggested in Section 3. A parallel version is also suggested in Section 4. It matches the best presently known time complexity, namely $O(\frac{n}{\log n})$ time with $n^{1+\epsilon}$ processors, $\epsilon > 0$ (see [3, 16, 15]). Section 5 describes single, double and multi-precision implementations of the sequential algorithm; timings of these implementations for pseudorandomly generated input pairs of various sizes are also provided, supporting our conclusion that the new algorithm is a good choice for small inputs in many circumstances.

## 2. The Sequential Algorithm

### *2.1. Motivation*

Let us start with an illustrative example. Let $(u, v) = (5437, 2149)$. After one euclidean step, we obtain the quotient $q = 2$ and the remainder $r = 1139$. On the other hand, we observe that, in the same time, $u - v = 3288 = 2^3 \times 411$ and the binary algorithm gives $\frac{u-v}{8} = 411$ which is smaller and easy to compute (right-shift). The reverse is also true, Euclid algorithm step may perform much more than the binary algorithm with some other integers, especially when the quotients are large. So, the idea is that, instead of choosing one of them, one may take the most of both euclidean and binary steps and combine them in a same algorithm. Note that a similar idea was suggested by Harris (cited by Knuth [9]) with a different reduction step.

**Lemma 1.** *Let $u$ and $v$ be two integers such that $v$ odd, $u \geq v \geq 1$ and let $r = u \pmod v$. Then we have*

*i)* $\min \{ v - r,\ r,\ \frac{r}{2} \text{ or } \frac{v-r}{2} \} \leq \frac{v}{3}$

*ii)* $\gcd(r, \frac{v-r}{2}) = \gcd(u, v)$, *if $r$ is odd*

$\gcd(\frac{r}{2}, v - r) = \gcd(u, v)$, *if $r$ is even.*

Proof. Note that either $r$ or $v-r$ is even, so that either $\frac{r}{2}$ or $\frac{v-r}{2}$ is an integer. The basic gcd property is $\forall\lambda \geq 0,\ \gcd(u, v) = \gcd(v, u - \lambda v)$. Two cases arise:
**Case** 1: $r$ is even then $v - r$ is odd. If $r \leq \frac{2v}{3}$ then $\frac{r}{2} \leq \frac{v}{3}$, otherwise $r > 2v/3$ and $v - r < \frac{v}{3}$. Moreover, $\gcd(\frac{r}{2}, v - r) = \gcd(r, v - r) = \gcd(v, r) = \gcd(u, v)$.
**Case** 2: $r$ is odd then $v - r$ is even. If $v - r \leq \frac{2v}{3}$ then $\frac{v-r}{2} \leq \frac{v}{3}$, otherwise, $v - r > 2v/3$ and $r < \frac{v}{3}$. On the other hand, $\gcd(\frac{v-r}{2}, r) = \gcd(r, v - r) = \gcd(v, r) = \gcd(u, v)$.

We derive, from Lemma 1, the following algorithm.

Algorithm `MBE:   Mixed Binary  Euclid`

```
Input: u>=v>=1, with v odd
Output: gcd(u,v)
      while (v>1)
           r=u mod v; s=v-r;
           while (r>0 and r mod 2 =0 ) r=r/2;
           while (s>0 and s mod 2 =0 ) s=s/2;
           if (s<r) {u=r; v=s; }
           else     {u=s; v=r; };
       endwhile
       if (v=1) return 1 else return u.
```

**Example**: With Fibonacci numbers $u = F_{17} = 1597$ and $v = F_{16} = 987$, we obtain:

| $q$ | $r$ | *reduction* | |
|---|---|---|---|
| | 1597 | $u$ | |
| | 987 | $v$ | |
| 1 | 610 | $r = u - qv$ | |
| | 377 | $s = v - r$ | |
| | 305 | $r/2$ | |
| 1 | 72 | $r$ | |
| | 233 | $v - r$ | |
| | 9 | $r/8$ | |
| 25 | 8 | $r$ | |
| | 1 | $v - r$ | |
| | 1 | $r/8$ | STOP |

Note that Euclid algorithm gives the answer after 15 iterations, and its extended version gives: $-377\ u + 610\ v = 1 = \gcd(u, v)$, while MBE algorithm gives a modular relation $(-55\ u + 89\ v) = 8 = 2^3 \gcd(u, v)$, after 3 iterations. Moreover, we observe that the coefficients $-55$ and $89$ are smaller than $-377$ and $610$. We know that the cofactors of Bézout relation can be roughly as large as the size of the inputs (consider successive Fibonacci worst case inputs). So an interesting question is : What is the upper bound for the modular coefficients $a$ and $b$ in the relation $au + bv = 2^t\ \gcd(u, v)$ ?

*2.2. Complexity analysis*

First of all, thanks to Lemma 1, we have an upper bound for the number of iterations of the main loop. We have $(u, v) \to (u', v')$, such that $v' \leq v/3$, so after $k$ iterations, we obtain $1 \leq v/3^k < 2^n/3^k$ or, $3^k < 2^n$, hence a first upper bound $k \leq \lfloor (\log_3 2)\ n \rfloor$. So the algorithm is quadratic in bit complexity as the binary or Euclidean algorithms. However, the following lemma proves that the worst case provides a smaller upper bound for the number of iterations.

**Lemma 2.** *Let $k \geq 1$ and let us consider the sequence of vectors $\begin{pmatrix} r_k \\ s_k \end{pmatrix}$ defined by* $\forall k \geq 1,\ \begin{pmatrix} r_{k+1} \\ s_{k+1} \end{pmatrix} = \begin{pmatrix} 2r_k + 2s_k \\ 2r_k + s_k \end{pmatrix}$ and $\begin{pmatrix} r_1 \\ s_1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$.

*Then the worst case of algorithm* MBE *occurs when the inputs* $(u, v)$ *are equal to*

$$\begin{pmatrix} u_k \\ v_k \end{pmatrix} = \begin{pmatrix} 2r_k + s_k = s_{k+1} \\ r_k + s_k = r_{k+1}/2 \end{pmatrix},$$

*and the gcd is given after $k$ iterations.*

PROOF. Roughly speaking, the worst case is reached when, at each time, the quotient is 1 (the smallest), only one division by 2 occurs and the output is the smallest one. We can easily prove by induction that

$$\begin{cases} \forall k \geq 1,\ r_k \text{ is even, } s_k \text{ and } \frac{r_k}{2} \text{ are odd} \\ \forall k \geq 2,\ \frac{r_k}{2} < s_k < r_k \\ \forall k \geq 2,\ \lfloor \frac{u_k}{v_k} \rfloor = 1. \end{cases}$$

We call an *iteration*, each iteration of the (**while** $v > 1$) loop. We prove by induction that, at each iteration $k$, we have $q_k = 1$ and the triplets $(r_k, s_k, \frac{r_k}{2})$, for $k \geq 2$. After the first iteration with the inputs $(u_k = 2r_k + s_k, v_k = r_k + s_k)$, we obtain the triplet $(r_k, s_k, \frac{r_k}{2})$ since $r_k$ is even and $\frac{r_k}{2}$ is odd. The relation $\frac{r_k}{2} < s_k < r_k$ yields and the next quotient $q_{k-1}$ will be $q_{k-1} = \lfloor \frac{s_k}{r_k/2} \rfloor = 1$. We repeat the same process with the new triplet$(r_{k-1}, s_{k-1}, \frac{r_k}{2})$ until we reach the triplet $(r_1, s_1, \frac{r_1}{2}) = (2, 1, 1)$ which is the smallest output triplet possible.

EXAMPLE: For $k = 7$ we have $u_7 = 9805$ and $v_7 = 6279$. We obtain 7 iterations. Note that Euclid algorithm gives the answer after 12 iterations.

We give below the link between the maximum of iteration and the number of bits of the larger input integer.

**Proposition 1.** *Let $u \geq v \geq 11$ be two integers, where $u$ is an $n$-bit integer. If $k$ is the number of iterations when algorithm* MBE *is applied then*

$$k \leq \lceil \frac{n}{\log_2 \lambda} \rceil, \quad \text{with } \lambda = \frac{3 + \sqrt{17}}{2}.$$

PROOF. Let $u \geq v \geq 11$ be two integers, where $u$ is an $n$-bit integer, so that

$2^{n-1} \leq u < 2^n$. Let us denote $A = \begin{pmatrix} 2 & 2 \\ 2 & 1 \end{pmatrix}$, so, for each $k \geq 1$,

$$\begin{pmatrix} r_{k+1} \\ s_{k+1} \end{pmatrix} = A \begin{pmatrix} r_k \\ s_k \end{pmatrix}.$$

Let $\lambda_1 = \frac{3+\sqrt{17}}{2}$ and $\lambda_2 = \frac{3-\sqrt{17}}{2}$ be the enginevalues of $A$. Then the worst case occurs after $k$ iterations with $u \leq C\ (\lambda_1)^k < 2^n$, where $C$ is some positive constant. As a matter of fact we prove easily by induction or by diagonalization of matrix $A$, that $\forall k \geq 1$ :

$$\begin{cases} r_k = \frac{2}{\sqrt{17}}\ (\lambda_1^k - \lambda_2^k) \\ s_k = (\frac{\sqrt{17}-1}{2\sqrt{17}})\ \lambda_1^k\ +\ (\frac{\sqrt{17}+1}{2\sqrt{17}})\ \lambda_2^k\ . \end{cases}$$

Then, after a bit of calculation, we obtain

$$k = \lfloor \frac{n}{\log_2(\lambda_1)} \rfloor + 1.$$

REMARK: Note that $k \sim (\frac{\log 2}{\log \lambda})\ n \sim 0,54\ n$, while, when euclidean algorithm is applied to $n$-bit integers, the number of iterations is bounded by $k' \leq (\frac{\log 2}{\log \phi})\ n \sim 1,44\ n$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Indeed, a first experiment on 1000 pairs of 32-bit integers shows that our algorithm is about 3 time faster than Euclid algorithm. More careful experiments show a clear speed up for certain ranges of input size. These experiments are detailed in section 5.

## 3. The Multi-precision Algorithm

In order to avoid long divisions, we must consider some leading bits of the inputs $(u, v)$ for computing the quotients and some other last significant bits to know if either $r = u \bmod v$ or $s = v - r$ is even. The algorithm is based on the following multi-precision reduction step (sketch) called `MP-MBE`. The integer $m$ is a parameter choosen as in [14], it satisfies $m = O(\log n)$.

$M = Id$;
**Step 1:** Consider $u_1$ and $v_1$ the first $2m$ leading bits of respectively $u$ and $v$. Similarly, $u_2$ and $v_2$ are the last $2m$ significant bits of respectively $u$ and $v$.

**Step 2:** By Euclid algorithm, compute $q_1 = \lfloor u_1/v_1 \rfloor$. Compute $r_1 = |u_1 - q_1 v_1|$ and $s_1 = v_1 - r_1$. Similarly, compute $r_2 = |u_2 - q_1 v_2|$ and $s_2 = v_2 - r_2$ (see [14] for more details).

**Step 3:** Compute $t_1$ and $p_1$ such that $r_2/2^{t_1}$ and $s_2/2^{p_1}$ are both odd.

**Step 4:** Save the computations: $M \leftarrow M \times N$, where $N$ is defined by:
**Case 1:** $r_2$ is even. If $r_1/2^{t_1} \geq s_1$ then

$$N = \begin{pmatrix} 1/2^{t_1} & -q/2^{t_1} \\ -1 & q+1 \end{pmatrix}, \text{otherwise } N = \begin{pmatrix} -1 & q+1 \\ 1/2^{t_1} & -q/2^{t_1} \end{pmatrix}.$$

**Case 2:** $s_2$ is even. If $s_1/2^{p_1} \geq r_1$ then

$$N = \begin{pmatrix} -1/2^{p_1} & (q+1)/2^{p_1} \\ 1 & -q \end{pmatrix}, \text{otherwise } N = \begin{pmatrix} 1 & -q \\ -1/2^{p_1} & (q+1)/2^{p_1} \end{pmatrix}.$$

EXAMPLE: Let $u$ and $v$ be two odd integers such that: $u = 1617\ldots309$, and $v = 1045\ldots817$. We obtain, in turn, $N_1 = \begin{pmatrix} -1 & 2 \\ 1/4 & -1/4 \end{pmatrix}$ and $N_2 = \begin{pmatrix} -1 & 5 \\ 1/4 & -1 \end{pmatrix}$. Then the two steps are saved in the matrix $M = N_2 \times N_1 = \begin{pmatrix} 9/4 & -13/4 \\ -1/2 & 3/4 \end{pmatrix}$.

## 4. The Parallel Algorithm:

A parallel GCD algorithm can be designed based on the following `Par-MBE` reduction:

**Begin** ($k$ is a parameter such that $k = 2^m = O(n)$)
**Step** 1 **:** (in parallel)
  **For** $i = 1$ **to** $n$  $R[i] = v$, $S[i] = v$; $q_i = \lfloor (iu_1)/v_1 \rfloor$;  (see Step 2 of `MP-MBE`)
  **For** $i = 1$ **to** $n$  $r_i = |iu - q_i v|$ and $s_i = v - r_i$;  (see [15])
**Step** 2 **:**
  **While** ($r_i > 0$ and $r_i$ even)  **Do**  $r_i \leftarrow r_i/2$;
  **If** ($r_i < 2v/k$) **then** $R[i] = r_i$,  in parallel.
**Step** 3 **:**
  **While** ($s_i > 0$ and $s_i$ even)  **Do**  $s_i \leftarrow s_i/2$;
  **If** ($s_i < 2v/k$) **then** $S[i] = s_i$,  in parallel.
**Step** 4 **:**
  $r = \min\ \{R[i]\}$;  $s = \min\ \{S[i]\}$;  in $O(1)$ parallel time;
**If** $r \geq s$ **Return** $(r, s)$ **Else Return** $(s, r)$.
**End**.

### *4.1. Complexity Analysis*

The complexity analysis of the parallel GCD algorithm based on `Par-MBE` reduction is similar to that of `Par-ILE` in [15]. We compute in turn $q_i$, $r_i = |iu - q_i v|$, $s_i = v - r_i$ and test if $r_i < 2v/k$ or $s_i = v - r_i < 2v/k$ to select the index $i$. Note that there is no write concurrency. Recall that $k = 2^m$ is a parameter. All these computations can be done in $O(1)$ time with $O(n2^{2m}) + O(n \log\log n)$ processors. Indeed, precomputed table lookup can be used for multiplying two m-bit numbers in constant time with $O(n2^{2m})$ processors in CRCW PRAM model, providing that $m = O(\log n)$ (see [15, 16]).
Precomputed table lookup of size $O(m2^{2m})$ can be carried out in $O(\log m)$ time with $O(M(m)2^{2m})$ processors, where $M(m) = m \log m \log\log m$ (see [16] or [3] for more details). The computation of $r_i = |iu - q_i v|$ and $s_i = v - r_i$ require only two products $iu$ and $q_i v$ with the selected index $i$. Thus the reduction `Par-MBE` can be computed in parallel in $O(1)$ time with:

$$O(n2^{2m}) + O(n \log\log n) = O(n2^{2m}) \text{ processors.}$$

`Par-MBE` reduces the size of the smallest input $v$ by at least $m - 1$ bits. Hence the GCD algorithm based on `Par-MBE` runs in $O(n/m)$ iterations. For $m = 1/2\ \epsilon \log n$, ($\epsilon > 0$), this parallel GCD algorithm matches the best previous GCD algorithms in $O_\epsilon(n/\log n)$ time using only $n^{1+\epsilon}$ processors on a CRCW PRAM.

## 5. Experimental Sequential Implementation

The GNU MP Bignum Library (GMP) [4] is a highly optimized arbitrary precision integer arithmetic library, employing advanced algorithms for many integer operations, including greatest common divisor. Since its source code is freely available under the GNU public license, it is a natural environment for the development of new implementations of integer algorithms. In order to assess the performance of the MBE algorithm we decided to modify some of the low-level `mpn`-layer functions of GMP 4.3.1 [4, sect. 8] to use the MBE algorithm rather than the algorithms currently used by GMP. Thus we could make a head-to-head comparison of the new algorithm to the ones chosen by the GMP developers, avoiding the need to take into account differences in ancillary design issues such as integer representation and memory allocation.

This section is divided into three subsections. The first describes the three implementations of the algorithm, the second describes the actual timings of the three implementations on pseudorandomly generated input values in the appropriate range for the implementation, and the third presents some observations concerning the results.

### *5.1. Implementation description*

A GMP limb [4, sect. 3.2] is a block of bits from the base 2 representation of a nonnegative integer and is usually the same size as the architecture's word—32 bits or 64 bits. Three `mpn`-level functions were modified to create three separate implementations of the MBE algorithm, based on the size of the operands:

`mpn_lehmer_gcd` for multi-precision operands (more than two limbs)

`gcd_2`[1] for double-precision operands (two limbs)

`mpn_gcd_1` for single-precision operands (one limb)

The algorithms used by GMP in these functions are described in [4, sect. 16.3]. The binary algorithm is used for single and double-precision. Euclidean algorithm versions of `mpn_gcd_1` and `gcd_2` were also created, so that the MBE algorithm could be compared to both the binary and Euclidean algorithm in the single and double-precision ranges.

A variant of Lehmer's algorithm [8] is used at the low end the multi-precision range, which is similar in structure to the multi-precision MBE algorithm sketched above: the function `mpn_lehmer_gcd` calls on `mpn_hgcd2` to build a $2 \times 2$ matrix $M$ of single-precision integers until the quotient is too large to be incorporated into the matrix, at which time the main loop of `mpn_lehmer_gcd` uses $M$ to transform the old values of $u$ and $v$ to the new ones, using multi-precision integer operations. Above a certain threshold[2], a sublinear algorithm [10] is used. For multiprecision input, the MBE algorithm is compared to GMP's Lehmer-variant; the subquadratic algorithm is not included in the comparisons.

---

[1] `gcd_2` is actually accessed via the `mpn_lehmer_gcd` entry point.

[2] Denoted `GCD_DC_THRESHOLD` in GMP 4.3.1. On the machines used for experimentation, this value is 381 limbs for `i386`, 361 limbs for `ppc`, 691 limbs for `x86_64` and 242 limbs for `ppc64`.

*5.2. Timing results*

The comparisons were performed on two different computer architecture families, each comprising 32-bit and 64-bit versions: the Intel architectures `i386` and `x86_64`, and the PowerPC architectures `ppc` and `ppc64`. Timings for the Intel architectures were done on a Mac Pro with 2 GiB of 800 MHz memory, 12 MiB of L2 cache, 1.6 GHz bus speed, and one 2.8 GHz Quad-Core Intel Xeon processor, under OS X 10.5.8. Timings for the PowerPC architectures were done on a Power Mac G5 with 2 GiB of PC3200U-30330 memory, 512 KiB L2 cache, 600 MHz bus speed, and a 1.8 GHz PowerPC G5 (3.0) processor, under OS X 10.4.11.

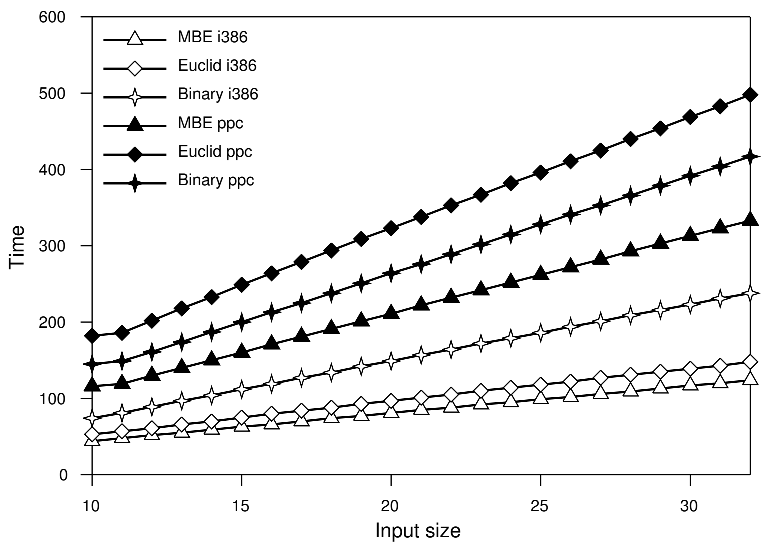
One important difference between these two processors is that the Enhanced Core 2 microarchitecture of the Xeon processor directly supports integer remainder [6] while the PowerPC architecture has an integer division instruction that returns only the quotient, so that the remainder must be computed using a division, multiplication, and subtraction [12, sect. 3.3.8]. Latency for division on the Xeon processor, which computes quotient and remainder, is 12-22 clock cycles and throughput is 5-14 clock cycles [6, Appendix C.3.1], varying with the number of significant bits in the quotient, while latency and throughput for both subtraction and right shift (fundamental operations in the binary algorithm) are 1 and 0.33 cycles, respectively. According to Noble and Papadopoulos [11], it takes 6 cycles for an 64-bit integer multiplication and roughly 60 cycles for a 64 bit integer divide on a PowerMac G5 processor, giving roughly 65 cycles for the remainder operation, assuming a subtraction costs at least one cycle. Thus the remainder operation on the Xeon processor is much closer in cost to subtraction and shifting than it is on the G5.

The compiler used was the Apple, Inc. implementation of `gcc version 4.0.1`. GMP 4.3.1 for these machines was obtained by using MacPorts to build `gmp @4.3.1_1+universal`, including object code for all four architectures. Modules from this library were statically linked into the comparison programs.

The BSD Unix system call `getrusage` [2] was used to query the operating system for time spent so far by the process executing user (i.e., non-privileged) instructions. Times reported below are computed by taking the difference in calls to `getrusage` before and after execution of a batch of one thousand calls to the particular function being timed. Any memory allocation required by GMP is performed before the start time is recorded. This system call appears to have an accuracy on the order of magnitude of one microsecond, under the operating systems used on the two machines.

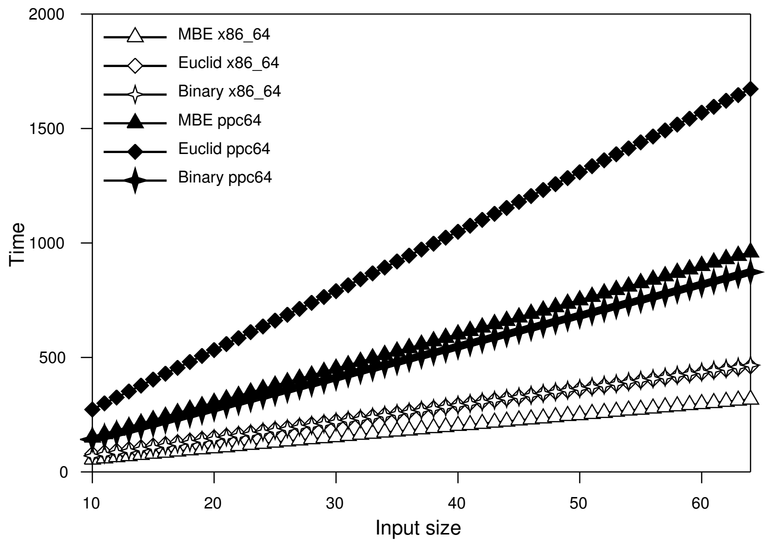
For single and double-precision tests, each data point for a given bit size represents average times, in nanoseconds, for one million pseudorandomly selected input pairs (grouped into 1,000 batches of 1,000 pairs). Only odd integers were selected for double-precision tests; single precision tests include even integers. The single and double-precision results are given in Figures 1 through 4.

For multiple-precision tests, each data point for a given size represents average times, in microseconds, for one batch of 1,000 pseudorandomly selected pairs. Only odd integers were selected for multi-precision tests. The multi-precision results are given in Figures 5 and 6. The columns in the table labeled



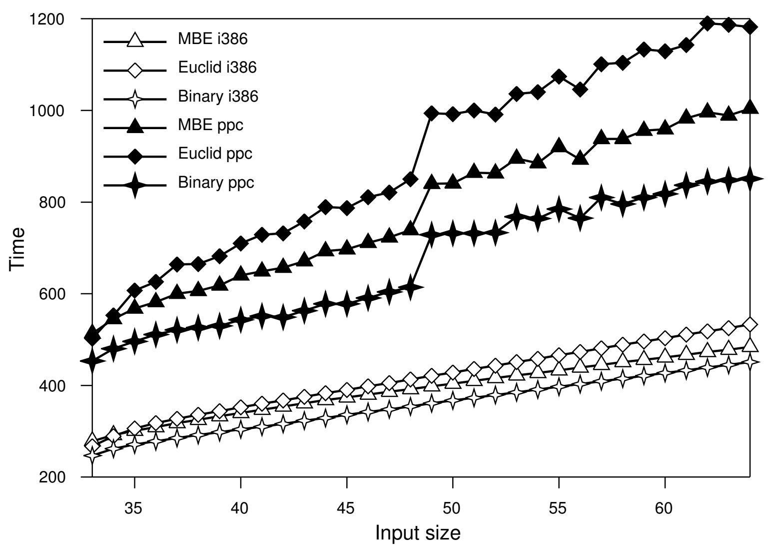

| Input size | i386 architecture | | | ppc architecture | | | Input | i386 architecture | | | ppc architecture | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (bits) | MBE | Euclid | Binary | MBE | Euclid | Binary | size | MBE | Euclid | Binary | MBE | Euclid | Binary |
| 10 | 44 | 53 | 74 | 116 | 182 | 145 | 22 | 88 | 105 | 164 | 232 | 353 | 289 |
| 11 | 48 | 57 | 81 | 119 | 186 | 149 | 23 | 92 | 110 | 172 | 242 | 367 | 302 |
| 12 | 52 | 61 | 89 | 130 | 202 | 161 | 24 | 95 | 114 | 179 | 252 | 382 | 315 |
| 13 | 55 | 66 | 97 | 140 | 218 | 174 | 25 | 99 | 118 | 186 | 262 | 396 | 328 |
| 14 | 59 | 70 | 104 | 150 | 233 | 187 | 26 | 102 | 122 | 194 | 272 | 411 | 341 |
| 15 | 63 | 75 | 112 | 160 | 249 | 200 | 27 | 106 | 127 | 201 | 282 | 425 | 353 |
| 16 | 66 | 80 | 119 | 171 | 264 | 213 | 28 | 109 | 131 | 209 | 293 | 440 | 366 |
| 17 | 70 | 84 | 127 | 181 | 279 | 225 | 29 | 113 | 135 | 216 | 303 | 454 | 379 |
| 18 | 74 | 88 | 134 | 191 | 294 | 238 | 30 | 117 | 139 | 223 | 313 | 469 | 392 |
| 19 | 77 | 93 | 142 | 201 | 309 | 251 | 31 | 120 | 143 | 231 | 323 | 483 | 404 |
| 20 | 81 | 97 | 149 | 211 | 323 | 264 | 32 | 124 | 148 | 238 | 333 | 498 | 417 |
| 21 | 85 | 101 | 157 | 222 | 338 | 276 | | | | | | | |

Figure 1: Times (ns) for 32-bit implementation—single precision range



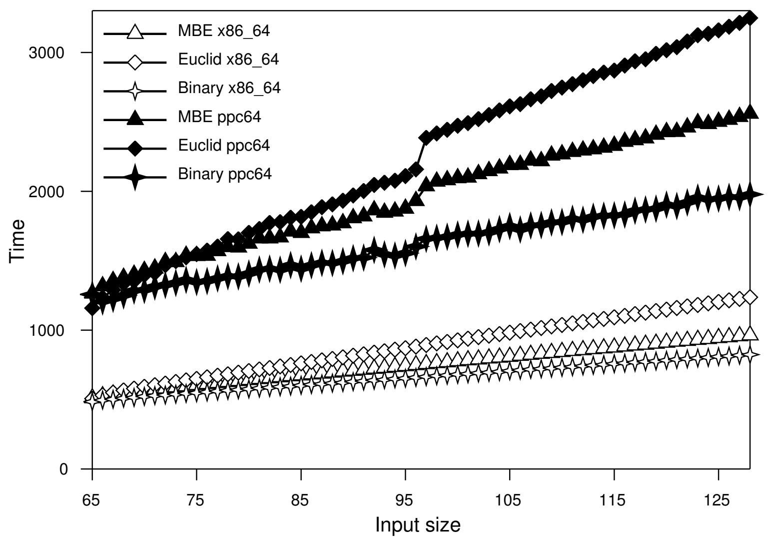

| Input size (bits) | x86_64 architecture MBE | x86_64 architecture Euclid | x86_64 architecture Binary | ppc64 architecture MBE | ppc64 architecture Euclid | ppc64 architecture Binary | Input size | x86_64 architecture MBE | x86_64 architecture Euclid | x86_64 architecture Binary | ppc64 architecture MBE | ppc64 architecture Euclid | ppc64 architecture Binary |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 55 | 79 | 71 | 151 | 273 | 142 | 38 | 192 | 279 | 277 | 572 | 998 | 521 |
| 11 | 60 | 86 | 78 | 166 | 300 | 155 | 39 | 197 | 286 | 284 | 587 | 1,024 | 534 |
| 12 | 65 | 93 | 86 | 181 | 326 | 169 | 40 | 202 | 293 | 292 | 602 | 1,050 | 548 |
| 13 | 70 | 100 | 93 | 197 | 352 | 182 | 41 | 206 | 300 | 299 | 617 | 1,076 | 562 |
| 14 | 75 | 108 | 101 | 212 | 378 | 196 | 42 | 211 | 307 | 306 | 632 | 1,102 | 575 |
| 15 | 80 | 115 | 108 | 227 | 404 | 209 | 43 | 216 | 314 | 313 | 647 | 1,128 | 589 |
| 16 | 85 | 122 | 116 | 242 | 430 | 223 | 44 | 221 | 321 | 321 | 661 | 1,154 | 602 |
| 17 | 90 | 130 | 123 | 258 | 455 | 236 | 45 | 226 | 328 | 328 | 676 | 1,180 | 616 |
| 18 | 95 | 137 | 130 | 273 | 481 | 250 | 46 | 230 | 335 | 335 | 692 | 1,206 | 629 |
| 19 | 100 | 144 | 138 | 288 | 507 | 263 | 47 | 235 | 342 | 343 | 706 | 1,232 | 643 |
| 20 | 104 | 151 | 145 | 303 | 533 | 277 | 48 | 240 | 349 | 350 | 721 | 1,258 | 657 |
| 21 | 109 | 158 | 153 | 318 | 559 | 291 | 49 | 245 | 356 | 357 | 736 | 1,284 | 670 |
| 22 | 114 | 165 | 160 | 333 | 585 | 304 | 50 | 249 | 363 | 364 | 751 | 1,310 | 683 |
| 23 | 119 | 172 | 167 | 348 | 611 | 318 | 51 | 254 | 370 | 372 | 766 | 1,336 | 697 |
| 24 | 124 | 179 | 175 | 363 | 636 | 331 | 52 | 259 | 377 | 379 | 781 | 1,362 | 711 |
| 25 | 129 | 186 | 182 | 378 | 662 | 345 | 53 | 264 | 384 | 386 | 796 | 1,388 | 724 |
| 26 | 134 | 193 | 189 | 393 | 687 | 358 | 54 | 269 | 391 | 394 | 811 | 1,414 | 738 |
| 27 | 139 | 201 | 197 | 408 | 713 | 372 | 55 | 273 | 398 | 401 | 826 | 1,440 | 751 |
| 28 | 144 | 208 | 204 | 422 | 739 | 385 | 56 | 278 | 405 | 408 | 841 | 1,466 | 765 |
| 29 | 148 | 215 | 211 | 438 | 765 | 399 | 57 | 283 | 412 | 415 | 856 | 1,492 | 778 |
| 30 | 153 | 222 | 219 | 452 | 790 | 412 | 58 | 288 | 419 | 423 | 871 | 1,518 | 792 |
| 31 | 158 | 229 | 226 | 467 | 816 | 426 | 59 | 293 | 426 | 430 | 886 | 1,544 | 806 |
| 32 | 163 | 236 | 233 | 482 | 842 | 440 | 60 | 297 | 433 | 437 | 901 | 1,570 | 819 |
| 33 | 168 | 243 | 240 | 497 | 868 | 453 | 61 | 302 | 440 | 445 | 916 | 1,596 | 833 |
| 34 | 173 | 250 | 248 | 512 | 894 | 467 | 62 | 307 | 447 | 452 | 931 | 1,622 | 846 |
| 35 | 177 | 257 | 255 | 527 | 920 | 480 | 63 | 312 | 454 | 459 | 945 | 1,647 | 860 |
| 36 | 182 | 265 | 262 | 542 | 946 | 494 | 64 | 317 | 461 | 467 | 960 | 1,673 | 873 |
| 37 | 187 | 272 | 270 | 557 | 972 | 507 | | | | | | | |

Figure 2: Times (ns) for 64-bit implementation—single precision range




| Input size (bits) | i386 architecture MBE | i386 architecture Euclid | i386 architecture Binary | ppc architecture MBE | ppc architecture Euclid | ppc architecture Binary | Input size | i386 architecture MBE | i386 architecture Euclid | i386 architecture Binary | ppc architecture MBE | ppc architecture Euclid | ppc architecture Binary |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 278 | 267 | 247 | 513 | 502 | 453 | 49 | 398 | 421 | 361 | 840 | 994 | 729 |
| 34 | 292 | 289 | 262 | 545 | 553 | 480 | 50 | 404 | 428 | 367 | 841 | 992 | 732 |
| 35 | 301 | 306 | 271 | 568 | 607 | 496 | 51 | 410 | 436 | 373 | 864 | 1,000 | 732 |
| 36 | 309 | 318 | 278 | 582 | 626 | 511 | 52 | 416 | 443 | 379 | 863 | 991 | 733 |
| 37 | 318 | 327 | 285 | 600 | 664 | 521 | 53 | 422 | 451 | 385 | 895 | 1,036 | 768 |
| 38 | 325 | 336 | 292 | 606 | 665 | 526 | 54 | 427 | 458 | 391 | 885 | 1,040 | 764 |
| 39 | 333 | 344 | 298 | 618 | 682 | 530 | 55 | 433 | 466 | 397 | 920 | 1,074 | 785 |
| 40 | 340 | 352 | 304 | 640 | 710 | 543 | 56 | 439 | 473 | 403 | 893 | 1,046 | 765 |
| 41 | 347 | 360 | 310 | 649 | 729 | 551 | 57 | 445 | 481 | 409 | 938 | 1,101 | 810 |
| 42 | 354 | 367 | 316 | 657 | 732 | 549 | 58 | 451 | 489 | 415 | 938 | 1,104 | 796 |
| 43 | 361 | 375 | 323 | 671 | 758 | 563 | 59 | 456 | 496 | 421 | 956 | 1,133 | 810 |
| 44 | 368 | 383 | 329 | 693 | 789 | 578 | 60 | 461 | 503 | 427 | 959 | 1,129 | 818 |
| 45 | 374 | 390 | 336 | 697 | 787 | 578 | 61 | 467 | 511 | 433 | 983 | 1,143 | 837 |
| 46 | 380 | 398 | 342 | 711 | 811 | 591 | 62 | 473 | 518 | 439 | 996 | 1,190 | 845 |
| 47 | 386 | 405 | 348 | 723 | 821 | 604 | 63 | 478 | 525 | 445 | 989 | 1,187 | 848 |
| 48 | 392 | 413 | 354 | 739 | 850 | 614 | 64 | 484 | 533 | 451 | 1,004 | 1,182 | 851 |

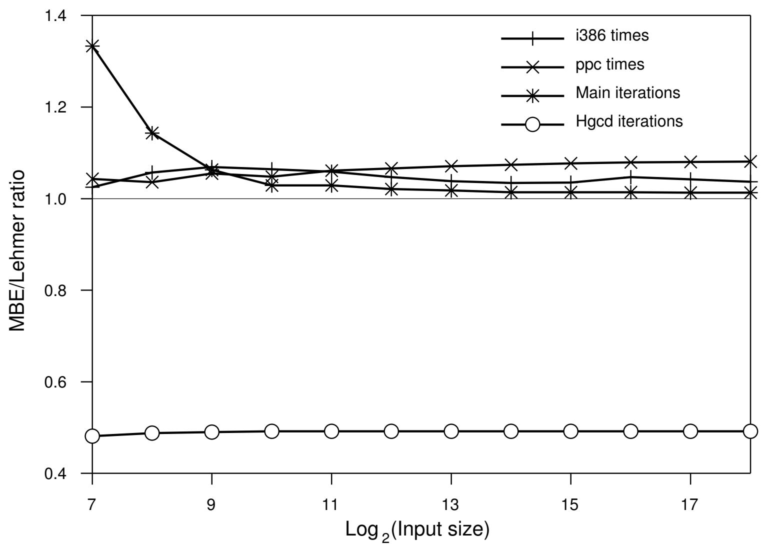
Figure 3: Times (ns) for 32-bit implementation—double precision range




| Input size (bits) | x86_64 architecture MBE | Euclid | Binary | ppc64 architecture MBE | Euclid | Binary | Input size | x86_64 architecture MBE | Euclid | Binary | ppc64 architecture MBE | Euclid | Binary |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 65 | 511 | 507 | 485 | 1,265 | 1,159 | 1,258 | 97 | 757 | 894 | 665 | 2,038 | 2,385 | 1,658 |
| 66 | 526 | 531 | 497 | 1,317 | 1,235 | 1,206 | 98 | 764 | 905 | 671 | 2,071 | 2,417 | 1,665 |
| 67 | 537 | 551 | 503 | 1,355 | 1,290 | 1,225 | 99 | 770 | 916 | 676 | 2,081 | 2,444 | 1,673 |
| 68 | 546 | 566 | 510 | 1,378 | 1,336 | 1,251 | 100 | 777 | 927 | 681 | 2,096 | 2,473 | 1,688 |
| 69 | 555 | 579 | 516 | 1,400 | 1,354 | 1,285 | 101 | 784 | 939 | 686 | 2,100 | 2,491 | 1,694 |
| 70 | 564 | 592 | 522 | 1,428 | 1,397 | 1,290 | 102 | 790 | 950 | 691 | 2,125 | 2,520 | 1,696 |
| 71 | 573 | 604 | 528 | 1,445 | 1,412 | 1,314 | 103 | 797 | 961 | 696 | 2,146 | 2,550 | 1,704 |
| 72 | 581 | 616 | 533 | 1,492 | 1,469 | 1,326 | 104 | 804 | 972 | 701 | 2,166 | 2,583 | 1,727 |
| 73 | 589 | 627 | 538 | 1,492 | 1,498 | 1,341 | 105 | 810 | 983 | 707 | 2,190 | 2,613 | 1,749 |
| 74 | 597 | 639 | 544 | 1,537 | 1,526 | 1,363 | 106 | 817 | 994 | 712 | 2,192 | 2,628 | 1,724 |
| 75 | 605 | 650 | 549 | 1,535 | 1,549 | 1,346 | 107 | 824 | 1,005 | 717 | 2,223 | 2,664 | 1,754 |
| 76 | 613 | 661 | 555 | 1,537 | 1,573 | 1,353 | 108 | 831 | 1,016 | 722 | 2,218 | 2,685 | 1,756 |
| 77 | 620 | 672 | 560 | 1,571 | 1,603 | 1,373 | 109 | 837 | 1,027 | 727 | 2,259 | 2,723 | 1,776 |
| 78 | 627 | 683 | 566 | 1,600 | 1,657 | 1,389 | 110 | 844 | 1,038 | 732 | 2,267 | 2,747 | 1,778 |
| 79 | 635 | 695 | 571 | 1,596 | 1,655 | 1,387 | 111 | 851 | 1,049 | 737 | 2,285 | 2,773 | 1,802 |
| 80 | 642 | 706 | 576 | 1,624 | 1,701 | 1,408 | 112 | 857 | 1,060 | 742 | 2,295 | 2,800 | 1,794 |
| 81 | 649 | 717 | 582 | 1,662 | 1,729 | 1,436 | 113 | 864 | 1,071 | 747 | 2,304 | 2,831 | 1,817 |
| 82 | 655 | 728 | 587 | 1,662 | 1,772 | 1,442 | 114 | 871 | 1,082 | 753 | 2,315 | 2,855 | 1,823 |
| 83 | 662 | 739 | 592 | 1,670 | 1,779 | 1,435 | 115 | 877 | 1,093 | 758 | 2,328 | 2,870 | 1,826 |
| 84 | 669 | 750 | 597 | 1,712 | 1,810 | 1,468 | 116 | 884 | 1,104 | 763 | 2,358 | 2,907 | 1,833 |
| 85 | 676 | 761 | 603 | 1,701 | 1,816 | 1,442 | 117 | 891 | 1,116 | 768 | 2,369 | 2,937 | 1,860 |
| 86 | 683 | 772 | 608 | 1,735 | 1,850 | 1,467 | 118 | 897 | 1,126 | 773 | 2,383 | 2,951 | 1,869 |
| 87 | 689 | 783 | 613 | 1,750 | 1,886 | 1,485 | 119 | 904 | 1,138 | 778 | 2,409 | 2,990 | 1,875 |
| 88 | 696 | 794 | 618 | 1,755 | 1,906 | 1,483 | 120 | 911 | 1,149 | 783 | 2,428 | 3,017 | 1,903 |
| 89 | 703 | 805 | 623 | 1,771 | 1,933 | 1,496 | 121 | 918 | 1,160 | 789 | 2,429 | 3,039 | 1,894 |
| 90 | 709 | 816 | 629 | 1,808 | 1,971 | 1,518 | 122 | 924 | 1,171 | 794 | 2,461 | 3,080 | 1,919 |
| 91 | 716 | 827 | 634 | 1,821 | 2,002 | 1,522 | 123 | 931 | 1,182 | 799 | 2,491 | 3,124 | 1,954 |
| 92 | 723 | 838 | 639 | 1,864 | 2,047 | 1,577 | 124 | 938 | 1,193 | 804 | 2,486 | 3,135 | 1,942 |
| 93 | 729 | 849 | 644 | 1,847 | 2,063 | 1,546 | 125 | 944 | 1,204 | 809 | 2,502 | 3,160 | 1,948 |
| 94 | 736 | 860 | 649 | 1,856 | 2,077 | 1,537 | 126 | 951 | 1,215 | 814 | 2,517 | 3,186 | 1,959 |
| 95 | 743 | 871 | 654 | 1,876 | 2,111 | 1,555 | 127 | 958 | 1,226 | 819 | 2,536 | 3,214 | 1,958 |
| 96 | 749 | 882 | 659 | 1,931 | 2,159 | 1,603 | 128 | 964 | 1,237 | 824 | 2,560 | 3,249 | 1,978 |

Figure 4: Times (ns) for 64-bit implementation—double precision range

*Hgcd iterations* refer to the average number of iterations required by the function `mpn_hgcd2` and the columns labeled *Main iterations* refer to the average number of iterations required by the main loop of `mpn_lehmer_gcd`. The graph displays the ratios of times and iteration counts for the MBE algorithm to times and iteration counts for the Lehmer variant.

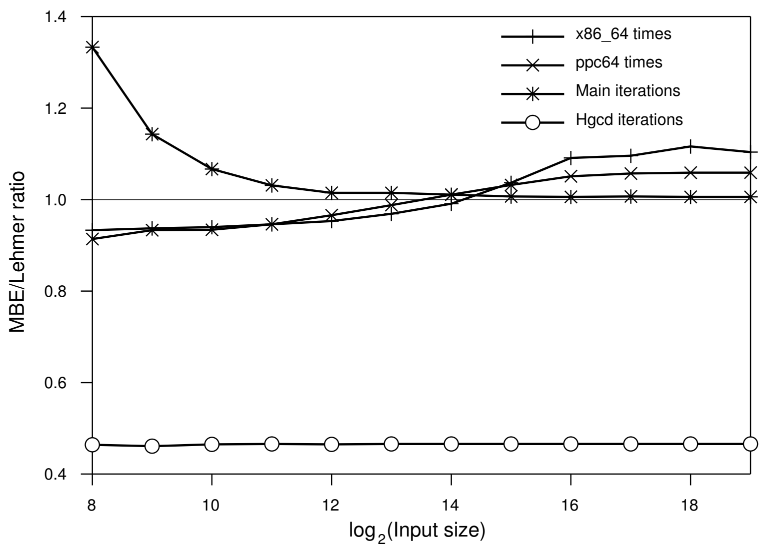



| Input size | i386 times | | | ppc times | | | Main iterations | | | Hgcd iterations | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (bits) | MBE | Lehmer | Ratio | MBE | Lehmer | Ratio | MBE | Lehmer | Ratio | MBE | Lehmer | Ratio |
| 128 | 1.904 | 1.858 | 1.025 | 2.968 | 2.847 | 1.043 | 4 | 3 | 1.333 | 26 | 54 | 0.481 |
| 256 | 4.324 | 4.090 | 1.057 | 6.743 | 6.510 | 1.036 | 8 | 7 | 1.143 | 61 | 125 | 0.488 |
| 512 | 9.822 | 9.190 | 1.069 | 15.339 | 14.539 | 1.055 | 17 | 16 | 1.063 | 140 | 286 | 0.490 |
| 1,024 | 22.117 | 20.777 | 1.064 | 35.534 | 33.913 | 1.048 | 35 | 34 | 1.029 | 297 | 604 | 0.492 |
| 2,048 | 52.148 | 49.241 | 1.059 | 89.185 | 84.059 | 1.061 | 72 | 70 | 1.029 | 608 | 1,236 | 0.492 |
| 4,096 | 132.203 | 126.267 | 1.047 | 248.031 | 232.654 | 1.066 | 144 | 141 | 1.021 | 1,233 | 2,506 | 0.492 |
| 8,192 | 377.072 | 363.095 | 1.038 | 773.362 | 722.300 | 1.071 | 289 | 284 | 1.018 | 2,480 | 5,041 | 0.492 |
| 16,384 | 1,206 | 1,167 | 1.034 | 2,657 | 2,473 | 1.074 | 578 | 570 | 1.014 | 4,976 | 10,109 | 0.492 |
| 32,768 | 4,226 | 4,082 | 1.035 | 9,760 | 9,061 | 1.077 | 1,158 | 1,142 | 1.014 | 9,966 | 20,254 | 0.492 |
| 65,536 | 15,846 | 15,131 | 1.047 | 37,316 | 34,577 | 1.079 | 2,316 | 2,285 | 1.014 | 19,950 | 40,552 | 0.492 |
| 131,072 | 60,500 | 58,089 | 1.042 | 145,947 | 135,124 | 1.080 | 4,632 | 4,572 | 1.013 | 39,917 | 81,116 | 0.492 |
| 262,144 | 236,105 | 227,700 | 1.037 | 577,589 | 534,433 | 1.081 | 9,266 | 9,147 | 1.013 | 79,848 | 162,248 | 0.492 |

Figure 5: Times ($\mu$s) and iterations for 32-bit implementation—multiple precision range

### *5.3. Observations*

The MBE algorithm is a clear winner for single precision on three of the four architectures; only on the `ppc64` architecture does it come in a close second to the binary algorithm. For double precision, it is better than the Euclidean algorithm but not as good as the binary algorithm on all four architectures. It seems that the level of support in hardware for integer remainder determines whether MBE or the binary algorithm is better, since the MBE algorithm clearly does a better job on the Xeon processor.




| Input size (bits) | x86_64 times MBE | x86_64 times Lehmer | x86_64 times Ratio | ppc64 times MBE | ppc64 times Lehmer | ppc64 times Ratio | Main iterations MBE | Main iterations Lehmer | Main iterations Ratio | Hgcd iterations MBE | Hgcd iterations Lehmer | Hgcd iterations Ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 256 | 2.900 | 3.109 | 0.933 | 5.675 | 6.206 | 0.914 | 4 | 3 | 1.333 | 51 | 110 | 0.464 |
| 512 | 6.336 | 6.761 | 0.937 | 12.212 | 13.082 | 0.933 | 8 | 7 | 1.143 | 118 | 256 | 0.461 |
| 1,024 | 13.511 | 14.366 | 0.940 | 25.891 | 27.723 | 0.934 | 16 | 15 | 1.067 | 255 | 548 | 0.465 |
| 2,048 | 29.709 | 31.409 | 0.946 | 59.592 | 61.909 | 0.946 | 33 | 32 | 1.031 | 543 | 1,166 | 0.466 |
| 4,096 | 67.071 | 70.413 | 0.953 | 140.367 | 145.298 | 0.966 | 67 | 66 | 1.015 | 1,118 | 2,405 | 0.465 |
| 8,192 | 162.770 | 168.054 | 0.969 | 369.651 | 374.014 | 0.988 | 135 | 133 | 1.015 | 2,263 | 4,861 | 0.466 |
| 16,384 | 437.674 | 441.462 | 0.991 | 1,095 | 1,083 | 1.011 | 271 | 268 | 1.011 | 4,556 | 9,780 | 0.466 |
| 32,768 | 1,344 | 1,296 | 1.037 | 3,615 | 3,503 | 1.032 | 543 | 539 | 1.007 | 9,140 | 19,623 | 0.466 |
| 65,536 | 4,627 | 4,242 | 1.091 | 12,995 | 12,367 | 1.051 | 1,086 | 1,079 | 1.006 | 18,306 | 39,318 | 0.466 |
| 131,072 | 16,810 | 15,342 | 1.096 | 48,934 | 46,305 | 1.057 | 2,174 | 2,159 | 1.007 | 36,644 | 78,675 | 0.466 |
| 262,144 | 64,172 | 57,511 | 1.116 | 189,389 | 178,917 | 1.059 | 4,348 | 4,320 | 1.006 | 73,309 | 157,395 | 0.466 |
| 524,288 | 249,806 | 226,234 | 1.104 | 746,704 | 704,828 | 1.059 | 8,696 | 8,642 | 1.006 | 146,651 | 314,855 | 0.466 |

Figure 6: Times ($\mu$s) and iterations for 64-bit implementations—multiple precision range

For multiprecision input the MBE algorithm doesn't perform as well as the Lehmer variant on the 32-bit architectures, but is marginally better than Lehmer on the 64-bit architectures, up to 16,384 bits on the `x86_64` machine and up to 8,192 bits on the `ppc64`. It is clear from the graph that MBE requires less than half of the iterations needed by the Lehmer variant in the Hgcd step.

Both GMP's version of `mpn_hgcd2` and the modified version used in the experimental implementation of MBE require several double-precision arithmetic operations per iteration to compute the $M$ matrix, so the advantage MBE has here is significant, but the cost of the multi-precision steps in the main loop of `mpn_lehmer_gcd` dominates the overall cost, and since MBE uses slightly more of these than Lehmer, MBE becomes more expensive for larger inputs. The experimental implementation could quite probably be improved so that more double-precision Hgcd steps could be combined into fewer main loop steps, but it is doubtful that MBE will be significantly faster than the Lehmer variant currently in use in GMP 4.3.1.

## 6. Conclusion

The Mixed Binary-Euclid algorithm has a sequential time complexity of $O(n^2)$, so it is not competitive asymptotically. However, a parallel version of the algorithm matches the best presently known time complexity. In addition, we provided experimental evidence that it has superior performance for single precision inputs when there is good hardware support for integer division. There is also some chance that the multiprecision version would be competitive, and we have identified some ideas to improve it. One of these ideas is the use of pseudo-quotients, called $\rho$-Euclid ([15], Section 5.1), to improve the computation of the Hgcd step.

## References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974

[2] BSD System Calls Manual, 4th Berkeley Distribution, June 4, 1993

[3] B. Chor and O. Goldreich, *An improved parallel algorithm for integer GCD*, Algorithmica, 5, 1990, 1-10

[4] The GMP Developers, GNU MP: The GNU Multiple Precision Arithmetic Library, Edition 4.3.0, Free Software Foundation, 14 April 2009

[5] IBM PowerPC 970FX RISC Microprocessor Users Manual, IBM Corp., Version 1.7, March 14, 2008

[6] Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number 248966-020, Intel Corporation, November 2009

[7] T. Jebelean, *A Generalization of the Binary GCD Algorithm*in Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'93), 1993, 111-116

[8] T. Jebelean, *A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers*, J. Symbolic Computation, 19, 1995, 145-157

[9] D.E. Knuth, The Art of Computer Programming, Vol. 2, 3rd ed., Addison Wesley, 1998

[10] N. Möller, *On Schönhage's algorithm and subquadratic integer GCD computation*, Mathematics of Computation, 77, 2008, 589-607

[11] S. Noble and J. Papadopoulos, *Special applications of 64-bit arithmetic: Acceleration on the Apple G5*, available through the Advanced Computation Group of Apple, Inc., 26 May 2006

[12] PowerPC User Instruction Set Architecture, IBM Corp., Book I, Version 2.02, January 28, 2005

[13] A. Schönhage, *Schnelle Berechnung von Kettenbruchentwicklugen*, Acta Informatica, 1, 1971, 139-144

[14] S.M. Sedjelmaci, *A Modular Reduction for GCD Computation*, Journal of Computational and Applied Mathematics, Vol. 162-I, 2004, 17-31

[15] S.M. Sedjelmaci, *A Parallel Extended GCD Algorithm*, Journal of Discrete Algorithms, 6, 2008, 526-538

[16] J. Sorenson, *Two Fast GCD Algorithms*, J. of Algorithms, 16, 1994, 110-144

[17] D. Stehlé, and P. Zimmermann, *A Binary Recursive Gcd Algorithm*, in Proc. of ANTS VI, University of Vermont, USA, June 13-18, 2004, 411-425

[18] J. von zur Gathen, and J. Gerhard, Modern Computer Algebra, 1st ed. Cambridge University Press, 1999