PROVING THE PRIMALITY OF VERY LARGE NUMBERS WITH FASTECPP

J. FRANKE, T. KLEINJUNG, F. MORAIN, AND T. WIRTH

ABSTRACT. The elliptic curve primality proving algorithm is one of the fastest practical algorithms for proving the primality of large numbers. Its fastest version, fastECPP, runs in heuristic time $\tilde{O}((\log N)^4)$. The aim of this article is to describe new ideas used when dealing with very large numbers. We illustrate these with the primality of the smallest number with 10,000 decimal digits.

1. Introduction

The work of Agrawal, Kayal and Saxena [1] on the existence of a deterministic polynomial time algorithm for deciding primality stimulated the field of primality proving at large. As a result, this caused the study and implementation of a fast version of the elliptic curve primality proving algorithm (ECPP). This faster version, originally due to J. O. Shallit, is a method running in heuristic time $\tilde{O}((\log N)^4)$ for proving the primality N. This method [13] turned out to be much faster than the ordinary one [2], proving the primality of numbers with more than $7,000$ decimal digits (with the distributed version, see below).

Independently of these efforts, three of us (JF, TK and TW) started to write a new implementation of ECPP in November 2002. This was motivated by the interest of TW in writing a Diploma Thesis about elliptic curves, and by the opinion of his advisor (JF) that it should be possible to improve upon the records established by the `PRIMO`* program of M. Martin, using a distributed implementation and faster arithmetic. A crude monoprocessor version of ECPP was available by December 2002, and this was improved step by step until the team working in Bonn came up with a set of programs which was used to prove the primality of $10^{9999} + 33603$ on August 19, 2003.

The two teams decided after this to exchange ideas and comparisons, forming the present article.

Our article is organized as follows. Section 2 gives a short description of fastECPP. Section 3 gives two strategies for distributing the computations. Section 4 deals with a faster way of looking for small prime factors of a bunch of numbers at the same time. In Section 5, an alternative to the method of [9, 5] for the root finding in the proving steps of ECPP is described. Section 6 deals with the use of fast multiplication beyond the GMP level in order to speed up all basic multiplications. In Section 7, we describe an *early abort strategy* for limiting the number of steps in ECPP. We conclude with timings on primality proofs for some very large numbers, obtained with either implementation.

2. The fast version of ECPP

Ordinary ECPP is described in [2] and fastECPP in [13]. We want to prove that N is prime. The algorithm builds a so-called *downrun* that is a sequence of decreasing probable primes $N_0, N_1, \ldots, N_k$ such that $N_0 = N$ and the primality of each N_i is enough to prove that of N_{i-1}. Theory tells us that we should anticipate a length of $k = O(\log N)$ for the sequence.

Date: December 31, 2003, version 1.8.

*`http://www.ellipsa.net/`

If q is an odd prime, put $q^* = (-1)^{(q-1)/2}q$; add to this special primes $-4, -8, 8$ as explained in [13].

The algorithm runs as follows:

[Step 1.]

1.1. Find the r smallest primes q^* such that $\left(\frac{q^*}{N}\right) = 1$, yielding $\mathcal{Q} = \{q_1^*, q_2^*, \ldots, q_r^*\}$.

1.2. Compute all $\sqrt{q^*}$ mod N for $q^* \in \mathcal{Q}$.

1.3. Try all subsets $\mathcal{S} = \{q_{i_1}^*, q_{i_2}^*, \ldots, q_{i_s}^*\}$ of $\mathcal{Q}$ for which $\prod_{q^* \in \mathcal{S}} q^* = -D < 0$, until a solution of the equation

$$4N = U^2 + DV^2 \tag{1}$$

in rational integers U and V is found. When this is the case, let $\{U_1, \ldots U_w\}$ be the different U-values (we have at most $w = 6$ and generally $w = 2$).

[Step 2.] For all U_i's, compute $m_i = N + 1 - U_i$; if none of these number can be written as cN' with c a B-smooth number and N' a probable prime, go to Step 1. It there is a good one, call it m.

[Step 3.] Build an elliptic curve $\overline{E}$ over $\overline{\mathbb{Q}}$ having complex multiplication by the ring of integers of $\mathbf{K}$.

[Step 4.] Reduce $\overline{E}$ modulo N to get a curve E of cardinality m.

[Step 5.] Find P on E such that $[N']P \neq O_E$. If this cannot be done, then N is composite, otherwise, it is prime.

[Step 6.] Set $N = N'$ and go back to Step 1.

Note that what differentiates ECPP from its fast version is Step 1. In Step 2., B-smooth numbers are to be identified. The number B is important in the actual running time, and its precise value must be set depending on the algorithm used. See section 4 below.

All algorithms and tricks [11, 12] developed over the years for ECPP apply *mutatis mutandis* to the new version. This includes the invariants developed in [6, 4] and the Galois approach for solving the equation $H_D[u](X)$ modulo p (see [9, 5]) needed in Step 3, which favors smooth class numbers. A new strategy (early abort) is described in the next section.

When dealing with very large numbers (10000 decimal digit numbers, say), a new look should be given to every part of the algorithm, which includes optimizing the basic routines beyond the current level of GMP. Also, Step 3-4 can be merged, as explained in section 5.

3. First strategies for distribution

Step 1-2 and Step 3-5 are easy to distribute over clusters of workstations. In this section, we give two distribution strategies.

3.1. **Strategy 1.** The following is easily implemented when all slaves have the same computing power, making it a parallel implementation.

S1. The master sends to each slave a batch of $\sqrt{q_i*}$ to compute.

S2. Each slave computes its batch and sends the results back to the master.

S3. The master sends all the squareroots to all the slaves.

S4. The master sends batches of D's to all the slaves. Each slave is responsible for the resolution of (1) and the factorization of the m's. If one is found, it is sent back to the master which checks the results and restart a new phase.

In S1, processors receive a number of squareroots proportional to their power, so that the load should be well balanced. S3 needs synchronization and communications.

In S4, load balancing is not easy, since the results are probabilistic in nature (for which D does N split?). A probabilistic answer is to compute beforehand the average number of splitting D's that can occur. By genus theory, each D with t prime factors has splitting probability $g(-D)/h(-D)$ with $g(-D) = 2^{t-1}$. We send to each slave discriminants $D_{i_1}, D_{i_2}, \ldots, D_{i_k}$ in such a way that

$$\sum_j g(D_{i_j})/h(D_{i_j}) \geq 5$$

(the value of 5 is somewhat arbitrary) which corresponds to the fact that on average, 5 values of D will be splitting values. Of course, this quantity should depend on the power of the slave.

3.2. **Strategy 2.** Another approach would be to set up a complicated system in which the master keeps trace of the work of each slave and decides what kind of work to do at some point. This is easily implemented on the side of the slaves: they wait for a task from the master, do it and send the result back. We now describe a possible implementation of the master.

There are 6 different tasks which the slaves work on:

T1. Check whether the class number for a discriminant D is good, i. e. is not too big and does not contain a very large prime factor.

T2. Compute a modular square root $\sqrt{q_i*}$ modulo N.

T3. Try to solve (1) for a given D.

T4. Do trial division for an interval of primes and a batch of m's and return the factored parts. See section...

T5. Do a Fermat test.

T6. Do a Miller-Rabin test.

The master keeps lists of tasks of these six types which at the beginning are all empty. If all task lists are empty the master creates new tasks of type T1. The tasks are sorted according to their priority, T1 having lowest and T6 having highest priority. If a slave request a new task the master selects from all available task one with the highest priority and passes it to this slave. A completed task will create a varying number of new task, e. g. a computed square root (T2) may create many task of type T3 whilst a Fermat test (T5) will only on success create a task of type T6. If a certain number of tasks T6 for the same integer are successful one reduction step is finished and pending tasks are cancelled.

4. An optimized method for the divisibility by small primes

Let us first analyze the effect of trial division on the number of pseudo-primality tests (the most time consuming part of our implementation). Suppose we do trial division up to B on a number $N = fR$ where f is only divisible by primes $\leq B$ and R only divisible by primes $> B$. Let us assume that $\log B \leq \sqrt[3]{\log N}$, a condition which is almost always satisfied in practice. Let $p_{\max}(n)$ be the largest prime divisor of n. One can combine the prime number theorem with Rankin's trick and Mertens' theorem [14, 9.1.5 and 9.1.8] and related facts to investigate the sums

$$s = \sum_{\substack{f<x \\ p_{\max}(f)<B}} \pi(x/f)$$
$$l = \sum_{\substack{f<x \\ p_{\max}(f)<B}} \log(f)\pi(x/f)$$

and, assuming $\log B \leq \sqrt[3]{\log x}$, finds them to be

$$s = \frac{x\exp(\gamma)\big(\log(B)+O(1)\big)}{\log(x)}$$
$$l = \frac{x\exp(\gamma)\big(\log(B)^2+O(\log B)\big)}{\log(x)},$$

where γ is Eulers constant. Since s counts the number of $N<x$ for which R is prime, while l is the sum of $\log(f)$ over such N, one concludes that for a randomly chosen $N \in [(1-\epsilon)x, x]$ with a fixed positive $\epsilon < 1$ and for $x \to \infty$, $z \to \infty$ subject to $\log B \leq \sqrt[3]{\log x}$, the probability that R is prime tends to $\exp(\gamma)\log(B)/\log(N)$ while the expectation value of $\log(l)$ is $\log(B)+O(1)$. By this heuristic argument, one expects the number of pseudo-primality tests for a reduction of fixed size to be proportional to $(\log B)^{-2}$.

We now describe how to do the trial division in Step 2 more efficiently by doing it on many numbers simultaneously. This is essentially a simplification of the algorithm in [3]. Let N be a (pseudo)prime for which we want to do a reduction step and let $m_1, \dots, m_t$ be the numbers computed in Step 2 (we may choose t of order $\frac{\log N}{2\log B}$). For simplicity we assume that $t = 2^u$ is a power of two. The following algorithm strips the primes up to B from the numbers m_i:

(1) Build the product $P = \prod_{p \leq B, p\,\mathrm{prime}} p$ using a binary tree. Unless the bound B is changed this has to be done only once.
(2) Compute the product $M = \prod_i m_i$ as follows: Set $m_i^{(0)} = m_i$ and for $a = 1, \dots, u$ successively compute $m_i^{(a)} = m_{2i-1}^{(a-1)} m_{2i}^{(a-1)}$, $1 \leq i \leq 2^{u-a}$. Set $M = m_1^{(u)}$.
(3) Compute $\tilde{M} = P \bmod M$.
(4) Compute $\tilde{m}_i = P \bmod m_i$ as follows: For $a = u, \dots, 1$ compute $\tilde{m}_{2i-1}^{(a-1)} = \tilde{m}_i^{(a)} \bmod m_{2i-1}^{(a-1)}$ and $\tilde{m}_{2i}^{(a-1)} = \tilde{m}_i^{(a)} \bmod m_{2i}^{(a-1)}$, $1 \leq i \leq 2^{u-a}$. Set $\tilde{m}_i = \tilde{m}_i^{(0)}$.
(5) For $1 \leq i \leq t$ replace repeatedly $\tilde{m}_i$ by $\gcd(m_i, \tilde{m}_i)$ and m_i by $\frac{m_i}{\tilde{m}_i}$ until $\tilde{m}_i = 1$.

Remarks:
Note that for $P < M$ we can save some of the top level computations and the application of the algorithm becomes equivalent to several applications with a smaller u, grouping the m_i appropriately. So we assume that u is chosen such that $P > M$ holds.

If we store the partial products

$$P^{(i)} = \prod_{p \leq \frac{B}{2^i}, p\,\mathrm{prime}} p$$

which are computed during the precomputation we can decrease the trial division bound by powers of 2 with no extra effort.

We now analyze the cost of the algorithm. Let $M(n)$ denote the cost of a multiplication of two numbers of size $\exp(n)$. We assume that FFT is used and set $M(n) = O(n(\log n)^{1+\epsilon})$. The first step is a precomputation whose cost is $O(B(\log B)^2)$. The cost for the second step is

$$\sum_{k=0}^{u-1} 2^{u-1-k} M(2^k \log N) = O(u2^u \log(N)(\log(2^u \log N))^{1+\epsilon})$$

since all m_i are of size N. In the fourth step the operation count is the same except that an $n \cdot n$-multiplication is replaced by an $2n : n$-division. Since the latter is asymptotically as fast as the former the cost for step 4 is the same as for step 2. Since $P \approx \exp(B)$ the cost of the third step is $O(B(\log(2^u \log N))^{1+\epsilon})$. The last step as described above has complexity $O(2^u(\log N)^2(\log\log N)^{2+\epsilon})$ since the iteration ends after at most $\log_2 N$ steps each consisting of a division and a gcd. This could be improved by modifying this step but we do not need it here. Note also that in practice this step consists mainly in the first $\gcd(m_i, \tilde{m}_i)$ which with high probability is very small and the number of iterations also is very small.

Assuming $2^u < \log N$ and neglecting $\log\log$-terms the time spent in pseudo-primality tests is $O(\frac{(\log N)^3}{\log B})$ for a reduction of size $\log B$ whereas the time for trial division is $O(B + (\log N)^3)$. So it is optimal to choose $B = O((\log N)^3)$ which also implies that the cost for the precomputation can be neglected.

Some remarks about storage and parallelization:

The algorithm above needs a lot of memory; most of it at the end of the computation of P. To reduce the memory requirement we may compute partial products P_j of the primes below B whose product is P and modify step 3 into computing the residues $P_j \bmod M$ and the modular product of these residues. For this to be efficient the partial products should be larger than M.

For a distributed implementation we propose to split P into as many pieces as slaves are present. Each slave executes steps 2-4 of the algorithm for its own P_j and passes at the end $\gcd(m_i, \tilde{m}_i) = \gcd(m_i, P_j)$ to the master. The master assembles these informations and executes step 5 which in practice is very fast.

For the number $10^{9999} + 33603$, the bound B was set to 2^{32}.

5. Calculating roots modulo p of the Hilbert polynomial

The run-down sequence contains, among other things, a list of pairs (p, D), where p is a pseudo-prime and where it is expected that an elliptic curve with complex multiplication by the ring $\mathcal{O}_K$ of integers in $K = \mathbb{Q}(\sqrt{-D})$ can be used to derive the primality of p from the primality of some smaller number. It is necessary to calculate an element j_p of $\mathbb{F}_p$ which is the j-invariant of an elliptic curve over $\mathbb{F}_p$ with complex multiplication by $\mathcal{O}_K$.

We outline the method which was used to perform this step for the run-down sequence of $10^{9999} + 33603$. As in [9], the idea is to split this task into several steps, each one involving the determination of the modular solution of an equation of degree ℓ_i, where the ℓ_i are the prime factors of the class number h of K. One difference is that [9] tries to calculate a sequence of polynomials which define a sequence of intermediate fields terminating in the Hilbert class field L of K. By contrast, the implementation which was used for $10^{9999} + 33603$ constructs methods to reduce elements x of the intermediate fields modulo an appropriate prime ideal over p, increasing the subfield in each step. The element x is given by floating point approximations to its conjugate algebraic numbers. Thus, the sequence of intermediate fields occurs in both methods but otherwise the language used is somewhat different, making it difficult to explain to what extent the methods are similar. Since the available space is not sufficient for are careful description of the new method, we give a short example explaining how it works in the case $p = 479$, $D = 335$.

The program chooses the modular invariant $x = x_{13}$ from [2, 2.7.1] and a precision of 32 bit is sufficient for the floating point calculations. The class group of K, and therefore $\mathrm{Gal}(L/K)$, is cyclic of order 18. The program has selected a generator σ of the Galois group, and has calculated the complex numbers $\sigma^i(x)$. It has then decided to choose the prime ideal $\mathfrak{p}_o \subset \mathcal{O}_K$ such that

$$(a + b\sqrt{-D}) \bmod \mathfrak{p}_o = \frac{1}{2}\Big(\big([2a] + 12[\frac{2b}{\sqrt{D}}]\big) \bmod p\Big), \tag{2}$$

where $\sqrt{D} = 18.30300521772312668$ is the positive square root of D, the subexpressions in square brackets are in real life floating point numbers which will be rounded to nearest integers, and the factor 12 in the second summand is of course a square root of $-D \bmod p$.

The Hilbert class field has unique subfields L_1 and L_2 of degrees 2 and 3 over K. The program knows that the genus field L_1, which is the largest subfield of L which is Abelian over $\mathbb{Q}$, is given by $L_1 = K(\sqrt{5})$. It decides to work with the prime ideal $\mathfrak{p}_1 \subseteq \mathcal{O}_{L_1}$ such that, for each element z of L_1 given by a complex float approximation to z and $\sigma(z)$, we have

$$z \bmod \mathfrak{p}_1 = \frac{1}{2}\Big(\big((z + \sigma(z)\big) \bmod \mathfrak{p}_o + 196\big((z - \sigma(z))\sqrt{5} \bmod \mathfrak{p}_o\big)\Big), \tag{3}$$

where $\sqrt{5} = 2.23606798$ and the factor 196 in the second summand is of course a modular square root of $1/5 \bmod p$. The reductions modulo $\mathfrak{p}_o$ occuring in this formula are calculated by the program using (2).

It is more difficult to describe a prime ideal $\mathfrak{p}_2 \subset \mathcal{O}_{L_2}$ with $\mathfrak{p}_2 \cap \mathcal{O}_K = \mathfrak{p}_o$ in a way which is suitable for calculations. The program considers

$$x_2 = \sum_{i=0}^{5} \sigma^{3i}(x),$$

which is an algebraic integer. We have

$$\begin{aligned} x_2 &= -60.2484307 + 78.0404771\sqrt{-1} \\ \sigma(x_2) &= -14.7805113 - 15.4588718\sqrt{-1} \\ \sigma^2(x_2) &= -10.4710580 + 1.47891293\sqrt{-1}. \end{aligned}$$

Note that $\sigma^i(x_2)$ depends only on $i \bmod 3$. The program calculates a complex float approximation to the minimal polynomial of x_2 over K and finds, using (2) to reduce polynomial coefficients modulo $\mathfrak{p}_o$, that this polynomial is congruent to $T^3 + 283T^2 + 226T + 108$ modulo $\mathfrak{p}_o$. It finds that 341, 395 and 418 are the roots of this polynomial modulo p and decides to work with the prime ideal $\mathfrak{p}_2 \subseteq L_2$ such that $x_2 \equiv 341 \pmod{\mathfrak{p}_2}$. It calculates coefficients $v_i \in \mathbb{F}_p$, $0 \le i < 3$, such that

$$z \bmod \mathfrak{p}_2 = \sum_{i=0}^{2} v_i\Big(\big(\sum_{j=0}^{2} \sigma^{i+j}(x_2)\sigma^j(z)\big) \bmod \mathfrak{p}_o\Big), \tag{4}$$

where in practice the reduction mod$\mathfrak{p}_o$ is carried out using (2). For this to be possible, it is necessary that x_2 generates a normal base of $\mathcal{O}_{L_2}$ over $\mathcal{O}_K$ after localisation at $\mathfrak{p}_o$. The program will abort if this assumption fails. This does not happen in this example, nor did it ever happen during the calculations for $10^{9999} + 33603$. But it should be possible to construct examples of failure of the program, although it is very unlikely for this to happen in practice. In order to determine the coefficients of (4), it is also necessary that the modular roots ξ_i of the minimal polynomial have been ordered in such a way that $\xi_i = \sigma^i(x_2) \bmod \mathfrak{p}_2$. The choice of ξ_0 is of course free, since a different ξ_0 only gives a different $\mathfrak{p}_2$. But the order of the other modular zeros is no longer free and the program has to calculate them in the correct order. We will describe in a different paper how this is done without a major increase of the required precision, where of course the order of the cyclic extension will often be larger than 3. Once the ξ_i have been calculated in the correct order, it is a linear algebra task to determine the v_i such that (4) holds for $z = \sigma^k(x_2)$, $0 \leq k < 3$. In the given example, the result is

$$v_0 = 417 \qquad v_1 = 170 \qquad v_2 = 393.$$

The compositum $L_3 = L_1 L_2$ has degree 6 over K, and there is a unique primeideal $\mathfrak{p}_3 \subset \mathcal{O}_{L_3}$ such that $\mathfrak{p}_3 \cap \mathcal{O}_{L_i} = \mathfrak{p}_i$ for $i \in \{1; 2\}$. If an element z of $\mathcal{O}_{L_3}$ is given by complex float approximations to $\sigma^i(z)$, where $0 \leq i < 6$, then

$$z \bmod \mathfrak{p}_3 = \frac{1}{2}\big(z_o \bmod \mathfrak{p}_2 + 196(z_1 \bmod \mathfrak{p}_o)\big), \tag{5}$$

where the $z_{0,1} \in L_2$ are given by

$$\begin{aligned} \sigma^i(z_0) &= \sigma^i(z) + \sigma^{i+3}(z) \\ \sigma^i(z_1) &= \sqrt{5}\big(\sigma^i(z) - \sigma^{i+3}(z)\big), \end{aligned}$$

and $z_i \bmod \mathfrak{p}_2$ is calculated by (4).

The program now calculates a complex float approximation to $\sigma^i(P)$ for $0 \leq i < 6$, where P is the minimal polynomial of x over L_3. Using (5), it finds P to be congruent to $T^3 + 151T^2 + 434T + 346$ modulo $\mathfrak{p}_3$. The largest rounding error was 0.000488281. One modular root of P is 153. This means that there exists a prime ideal $\mathfrak{p}$ dividing p in $\mathcal{O}_L$ such that $x \equiv 153 \bmod \mathfrak{p}$. From x, one can calculate the j-invariant of an elliptic curve with complex multiplication by $\mathcal{O}_K$ using the formula

$$j = \frac{(x^4 + 7x^3 + 20x^2 + 19x + 1)^3(x^2 + 5x + 13)}{x}.$$

With $x = 153 \bmod 479$, this gives $j = 307 \bmod 479$.

Calculating the minimal polynomial of x over K and reducing it modulo $\mathfrak{p}_o$, using (2), turns out to be impossible with 32 bit precision. If 48 bits are used, the largest rounding error is 0.0195312. Of course, this increase of the required precision is due to the fact that the theory of the genus field was not used.

The program used for $10^{9999} + 33603$ was a development version, with many possible optimisations not yet implemented. For instance, it is clear from the above example that not all Weber class invariants were implemented.

6. Use of the Fastest Fourier Transform in the West

For most of the calculations for $10^{9999} + 33603$, we used integer multiplication using the Fastest Fourier Transform in the West (see `http://www.fftw.org` and [7], [8]). To square a number of size 10^{9999}, it was broken into 1661 digits of size 20 bit. These digits were inserted into an array of 3600 `double` variables, which was then transformed using the functions provided by `libfftw3.a`, the result was squared and transformed back also using `libfftw3.a`. The same thing can be done for a product of two different factors, and if a factor occurs often then its Fourier transform may be precalculated and stored to reduce the time for a multiplication by this factor to the time for a squaring. It is easy to see that this choice of parameters does not guarantee exact results. Therefore, we also calculated a 32-bit checksum. If the checksum test indicates an error, the multiplication is recalculated using the `GMP` function. In the case of the p10000, these recalculations appear to be rare, if they occur at all. Of course, even the checksum does not make this multiplication method rigorous.

We used this fast multiplication both for primality tests and for the calculation of modular square roots in the calculation of the run down sequence. While a modular square root can be checked, and the calculation repeated if necessary, there appears to be no way to detect a false negative result of a Rabin-Miller test. Therefore, by using this method we accepted a small but probably positive probability of a prime number being declared composite by mistake.

The following benchmark results were obtained on a 800-MHz Athlon, using version 4.1.1 of `libgmp.a`, version 3.0 of `libfftw3.a`, and $10^{9999}+33603$ as the input number:

- One call to the `GMP` function `mpz_probab_prime_p` with second argument equal to 1, which means that one Fermat and one Rabin-Miller pseudo-primality test is carried out: 317 seconds user time.
- A Rabin-Miller test using 2 as base, and using Montgomery modular multiplication [10] and the `GMP` functions: 149 seconds.
- A similar program, but using `libfftw3.a` for multiplications: 56 seconds.

The advantage of using `libfftw3.a` could perhaps be reduced if `GMP` allowed for a way to precalculate Fourier and Toom-Cook transforms of frequently used factors. It is difficult to predict whether this is sufficient to achieve the speed of a multiplication subroutine which is optimized for speed at the cost of producing sometimes (albeit rarely) a false result.

7. The early abort strategy

The idea behind this strategy is to force the new candidate N' to satisfy N/N' greater than a certain bound, with the hope to decrease the number of steps and thereby the number of proving steps. Of course, this might slow down the search for N' a little bit and some optimization is necessary. Yet it appeared critical when used in the primality proof of $10^{9999}+33603$, when it was first implemented and used.

In FM's implementation, the following value for the lower bound on $\delta(N)=b(N)-b(N')$ was used, where $b(x)$ denotes the number of bits of integer x:

$b(N) \leq$	1000	2500	5000	7500	10000	15000	∞
$\delta(N)$	0	5	10	15	20	25	30

Using the MPI program, the following data were gathered. The first column contains the time without EAS, the second with. The line H_D stands for the time needed to compute the polynomials representing the class field, as in [9]; CZ is the time needed for the special variant of the Cantor-Zassenhaus algorithm using a trick of Atkin (using roots of unity modulo N); the number of steps in the *downrun* is then given, and the last lines contain the maximal value for h, before the mean value.

Number	$10^{2999}+1887$		$10^{3999}+4771$		$10^{4999}+22669$	
Steps 1-2	81 h	58 h	280 h	164 h	716 h	476 h
Steps 3-5	26 h	26 h	76 h	86 h	209h	261 h
H_D	1680 s	4880 s	4497 s	7317 s	3 h	5 h
CZ	22 h	22 h	63 h	75 h	179 h	234 h
# steps	436	358	597	437	734	512
$\max h$	1968	2336	2184	2432	3400	4000
$\overline{h}$	86	116	120	164	152	272

As was anticipated, any restriction one puts on m and thus on D tends to make D and h larger than in the plain case. This can have an impact on the time for computing H_D, and also on the proving part. EAS indeed decreases the number of steps, which tends to decrease the total time for the 1st phase, the 2nd being constant or increasing a little. In any case, getting a factor of 2 in the total running time is not to be wasted.

8. Some large primality proofs

8.1. **The first records of fastECPP.** We begin with some data from FM's implementation that uses MPI on top of his `ecpp` program, and implementing Strategy 1. All computations were done on a cluster of 6 biprocessors Xeon at 2.66 MHz. We took the following numbers from the tables of P. Leyland†. These are numbers of the form $x^y + y^x$. WCT stands for *wall clock time* and is a measure of the time wasted by the distribution process (waiting time of the slaves, typically). The line "Checking" indicates the time needed to check the certificate. Note that the time for this should be $\tilde{O}((\log N)^3)$ and this is compatible with the timings given.

x	2177	2589	2551	2438	3571
y	580	218	622	1995	648
#dd	6016	6055	7127	8046	10041
when	03:0513-0604	03:0606-0617	03:0618-0714	03:0715-0901	03:1001-1220
# steps	801	736	965	1128	947
Steps 1-2 (CPU)	164 days	103 days	235 days	355 days	531 days
Steps 1-2 (WCT)	164×1.2	103×1.1	235×1.3	355×1.13	531×1.2
$\sqrt{-q^*}$	81 days	30 days	74 days	138 days	204 days
Steps 3-5	28 days	21 days	55 days	77 days	138 days
H_D	2951 sec	1686 sec	18451 sec	22552 sec	20285 sec
CZ	26 days	20 days	50 days	69 days	124 days
Checking	25 hours	22 hours	45 hours	70 hours	85 hours
$\max h$	1980	2080	3312	3640	6176
$\max D$	7,749,263	19,076,479	52,396,648	87,949,348	95,895,480

Table 1. Some large numbers proven with fastECPP.

The first number was dealt with with an experimental program that turned out to spend too much time in the $\sqrt{-q^*}$ computations. As a matter of fact, a value of $r = 4000$ was used. This led to proceed by chunks of 400 squareroots from a total of 4000, adding 400 more if this was not enough. All discriminants with $D \leq 10^8$, $h \leq 6000$ and the largest prime of h not exceeding 200 were decided to be usable. A look at column 3 compared to 4 justifies the claim of complexity at $\tilde{O}((\log N)^4)$. The 8046*dd* number was done after the annoucement of the proof of $10^{9999} + 33603$ (see next section), and EAS was not used for this. The 10041dd number was finished on December 20, 2003, well after the one to be described. This was the first use of EAS for this implementation.

8.2. **A new frontier.** Let us turn our attention to the barrier-breaking number $10^{9999} + 33603$, whose primality was verified by JF, TK with the help of TW.

The calculations were done using two programs, a pvm program producing the sequence of discriminants, group orders and their prime number factorization (called a run down sequence in what follows), and the second program calculating the elliptic curves.

The calculation of a run down sequence was started on July 17, 2003 on six 900MHz PIII CPUs. On July 21, the computation was moved to 12 nodes of parnass2, the LINUX cluster built at the

†`http://research.microsoft.com/~pleyland/primes/xyyx.htm`

Scientific Computing Institute in Bonn. 4 of these node had two 800MHz CPUs, the other nodes were double PII/400MHz computers. At 8550 digits (on July 30) and 8286 digits (on July 31) we interrupted these calculations to replace the program by a faster version, using the Fastest Fourier Transform in the West in the way explained above. This improvement resulted by a speedup by a factor of about 2. On August 5, we reached 6574 digits. On August 8, we stopped the program at 3256 digits. The final calculations for the run down sequence took about 8 hours on eight 800MHz CPUs. The total CPU time to produce the run down (i.e, without calculating the elliptic curves) was estimated to 234.5 days on a 1GHz Pentium.

The CPU time spent for the actual certificates is more difficult to estimate, since the program for this step was still under development when the calculation of the run down sequence started, and since these calculations were done in heterogeneous environment. We estimate that it would have taken less than 140 days on a single 800MHz Athlon CPU.

The certificate is available as:
`ftp://ftp.math.uni-bonn.de:pub/people/franke/p10000.cert`

9. Conclusions

We have described new ideas to speed up practical primality proving of large numbers using fastECPP, some of which needing more testing and improvements. We hope that this will serve as benchmarks and motivations for the study of other primality proving algorithms as well.

Acknowledgments. Thanks to A. Enge for some useful discussions about this work. The authors of the certificate for $10^{9999} + 33603$ would like to thank the Scientific Computing Institute at Bonn University for providing the parallel computer which produced the downrun. One of us (J. F.) would also like to thank G. Zumbusch for point out the existence of libfftw3.a to him.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. Preprint; available at `http://www.cse.iitk.ac.in/primality.pdf`, August 2002.
[2] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61(203):29–68, July 1993.
[3] D. J. Bernstein. How to find small factors of integers. Available at `http://cr.yp.to/papers.html`, June 2002.
[4] A. Enge and F. Morain. Comparing invariants for class fields of imaginary quadratic fields. In C. Fieker and D. R. Kohel, editors, *Algorithmic Number Theory*, volume 2369 of *Lecture Notes in Comput. Sci.*, pages 252–266. Springer-Verlag, 2002. 5th International Symposium, ANTS-V, Sydney, Australia, July 2002, Proceedings.
[5] A. Enge and F. Morain. Fast decomposition of polynomials with known Galois group. In M. Fossorier, T. Høholdt, and A. Poli, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 2643 of *Lecture Notes in Comput. Sci.*, pages 254–264. Springer-Verlag, 2003. 15th International Symposium, AAECC-15, Toulouse, France, May 2003, Proceedings.
[6] A. Enge and R. Schertz. Constructing elliptic curves from modular curves of positive genus. Soumis, 2001.
[7] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.
[8] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
[9] G. Hanrot and F. Morain. Solvability by radicals from an algorithmic point of view. In B. Mourrain, editor, *Symbolic and algebraic computation*, pages 175–182. ACM, 2001. Proceedings ISSAC'2001, London, Ontario.
[10] Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.
[11] F. Morain. Primality proving using elliptic curves: an update. In J. P. Buhler, editor, *Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Comput. Sci.*, pages 111–127. Springer-Verlag, 1998. Third International Symposium, ANTS-III, Portland, Oregon, june 1998, Proceedings.
[12] F. Morain. Computing the cardinality of CM elliptic curves using torsion points. Submitted, October 2002.
[13] F. Morain. Implementing the asymptotically fast version of the elliptic curve primality proving algorithm. Available at `http://www.lix.polytechnique.fr/Labo/Francois.Morain/`, June 2003.

[14] M. Ram Murty. *Problems in Analytic Number Theory*, volume 206 of *Graduate Texts in Mathematics*. Springer-Verlag, 2001.

(J. Franke) Dept. of Math., Bonn University, Beringstr. 1, 53115 Bonn, Germany

(T. Kleinjung) Dept. of Math., Bonn University, Beringstr. 1, 53115 Bonn, Germany

(F. Morain) LIX École Polytechnique, CNRS/UMR 7650, INRIA/Futurs, F-91128 Palaiseau CEDEX, FRANCE

(T. Wirth) Dept. of Math., Bonn University, Beringstr. 1, 53115 Bonn, Germany
E-mail address, J. Franke: franke@math.uni-bonn.de
E-mail address, T. Kleinjung: thor@math.uni-bonn.de
E-mail address, F. Morain: morain@lix.polytechnique.fr
E-mail address, T. Wirth: wirth@math.uni-bonn.de