MATHEMATICS OF COMPUTATION
Volume 72, Number 243, Pages 1555–1572
S 0025-5718(02)01479-5
Article electronically published on December 6, 2002

THE TWENTY-FOURTH FERMAT NUMBER IS COMPOSITE

RICHARD E. CRANDALL, ERNST W. MAYER, AND JASON S. PAPADOPOULOS

ABSTRACT. We have shown by machine proof that $F_{24} = 2^{2^{24}} + 1$ is composite. The rigorous Pépin primality test was performed using independently developed programs running simultaneously on two different, physically separated processors. Each program employed a floating-point, FFT-based discrete weighted transform (DWT) to effect multiplication modulo F_{24}. The final, respective Pépin residues obtained by these two machines were in complete agreement. Using intermediate residues stored periodically during one of the floating-point runs, a separate algorithm for pure-integer negacyclic convolution verified the result in a "wavefront" paradigm, by running simultaneously on numerous additional machines, to effect piecewise verification of a saturating set of deterministic links for the Pépin chain. We deposited a final Pépin residue for possible use by future investigators in the event that a proper factor of F_{24} should be discovered; herein we report the more compact, traditional Selfridge-Hurwitz residues. For the sake of completeness, we also generated a Pépin residue for F_{23}, and via the Suyama test determined that the known cofactor of this number is composite.

1. Computational history of Fermat numbers

It is well known that P. Fermat, in the early part of the 17th century, described the numbers

$$F_n = 2^{2^n} + 1.$$

Fermat noted that for $n = 0, 1, 2, 3, 4$ these are all primes, and claimed that the primality property surely must hold for *all* subsequent $n > 4$. In a remarkable oversight, Fermat did not go on to test the status of F_5, even though he could have done so quite easily using the compositeness test that now bears his name, and whose later refinement by Euler yielded the key to the rigorous Pépin test for the F_n. After the discovery of certain small factors of various F_n through the ensuing centuries, and after the machine-aided work of Selfridge and Hurwitz [28] with the spectacular resolution of F_{14} as composite, it was known by the early 1980s that F_n is composite for all $5 \le n \le 32$, except for the five cases $n =$20, 22, 24, 28 and 31, for which the character of F_n remained unresolved [26]. Many of the compositeness proofs have simply involved direct sieving to find small factors, and there seems to be no end to such discoveries. For example, in 1997 Taura found a small factor of F_{28} [19]. More recently (in fact during preparation of this manuscript,) A. Kruppa discovered the first known factor of F_{31}, $p = 46931635677864055013377$, using a sieving program developed by T. Forbes. (This factor has $p - 1 = 2^{33} \cdot 3 \cdot 13 \cdot 140091319777$ and $p + 1 = 2 \cdot 7 \cdot 3352259691276003929527$, so could not have been

Received by the editor October 14, 1999 and, in revised form, September 5, 2001.
2000 *Mathematics Subject Classification.* Primary 11Y11, 11Y16, 68Q25, 11A51.

found via the $p-1$ or $p+1$ factorization algorithms, which methods have only recently come into the realm of feasibility for numbers of this size, as discussed further in §5.)

Between the early 1980s and the present, massive compositeness tests have established that F_{20} [31], F_{22} [9, 29], and as reported herein F_{24} are all composite. Thus we now know that every F_n with $5 \le n \le 32$ is composite, and the smallest Fermat number of unknown status is, at the time of this writing, the 8-gigabit colossus F_{33}. With the rise in distributed computing via the Internet, the pace of discovery (especially in the area of factoring) has recently accelerated; accordingly, the current status of known factors and the like is available on the World Wide Web [19]. More details on the computational issues relevant to Fermat numbers are presented in [12].

There are at the moment four F_n which are known to be composite but for which not a single factor is known, namely those with $n = 14, 20, 22, 24$. The discovery rate exhibited by these four indices n appears to follow a rule of thumb: To get from the character-resolution of F_m to that of F_n takes about $3(n-m)$ years. Using modern transform-based large-integer multiply algorithms, each unit increase in index yields an approximately fourfold increase in the number of machine computations needed for the Pépin test. This scaling thus implies a doubling in effective computing power approximately every 18 months, and as such appears to be a corollary to Moore's Law [23] (which strictly speaking only predicts an increase in areal complexity of integrated circuitry, although a concomitant speed increase can be inferred from this.) However, this is misleading, since it neglects significant algorithmic improvements which have occurred since the resolution of F_{14}. These are such that whereas the Pépin tests for F_{20} and F_{22} required vector supercomputer hardware in order to be completed in under a year, the two separate floating-point tests of F_{24} described in this paper were accomplished on quite modest commodity microprocessor hardware, and each running on just a single processor. In view of the enormous leap in size between F_{24} (5050446 decimal digits, something like a "book" of digits) and F_{33} (2585827973 digits, more like a "small library"), it is difficult to guess what manner of algorithms and machinery will be needed for Pépin tests on numbers of this and larger size. The rule of thumb of $3(n-m)$ if taken seriously would imply a resolution of F_{33} by about 2025 A.D. We believe that for such a resolution to occur that soon (barring the discovery of a small factor, which seems at least equally likely), machinery and algorithms must continue to improve at a rate comparable to that of the past several decades. As microprocessor device sizes are currently approaching scales where quantum effects begin to become appreciable, it seems doubtful that silicon-based hardware will be able to keep pace with Moore's Law for much longer, so it may well prove to be the case that F_{24} will remain the largest Fermat number resolved by the Pépin test on single-processor silicon computer hardware. In fact, since it is also reasonable to expect concomitant improvements in factoring algorithms and implementations thereof, it is not out of the realm of reasonable possibility that F_{24} will prove to be the largest Fermat number whose character is established via Pépin test on *any* kind of hardware (although larger numbers will certainly be subjected to Pépin test and Suyama step in order to test the status of known cofactors, once it becomes feasible to do so in a reasonably short time frame).

Regarding our computation, we paraphrase the claim of Young and Buell [31] regarding their calculation for F_{20}, that this is the deepest ever performed for what

(future cofactor tests aside) amounts to a "one-bit answer."[1] And the bit can be made explicit: we shall see that the high bit of the final Pépin residue can be interpreted as a Boolean bit rigorously signaling prime or composite. This is not to say that F_{24} is the largest number ever subjected to a direct (nonfactoring) compositeness test per se: since the conclusion of our Pépin test, several larger Mersenne numbers have been subjected to two independent Lucas-Lehmer tests—with the conclusion of composite character—by the GIMPS project [30]; however, none of the Mersennes was subjected to the much more computationally intensive additional step of an all-integer verification, as was F_{24}.

It is also the case that at the time of proof for F_{22}, some of the best machinery available was pressed hard for a good fraction of a year, to achieve the proof. Incidentally in the case of F_{22}, an entirely independent team of J. Carvalho and V. Trevisan [29] surprised the authors of [9] by announcing shortly after the latter group's run a machine proof of the same result, together with identical Selfridge-Hurwitz residues. The two proofs used different software and machinery, and so there can be no reasonable doubt that F_{22} is composite. When we say "no reasonable doubt" here, we mean certainty up to, say, what might be called "unthinkable" happenstance, such as space-time-separated coincidence of cosmic rays impingent on machines causing accidental false proof *and* accidentally verified false proof. For example, the probability of two final Pépin residues in the F_{22} case agreeing because of random bit-flips at both sites independently is say $2^{-2^{22}}$, or about $1/10^{1000000}$.

During the preparations for our runs of F_{24} we again reproduced the previous Selfridge-Hurwitz residues for F_{22}. Because we were unaware of any independent team also testing F_{24}, we took extreme care to verify our proof, as explained below. Again (in 1999) the machinery was hard-pressed for months, algorithm refinement required substantial labor but was well worth the effort, and so on.

To convey an idea of scale, we note that F_{24} is a number of nearly 17 million binary bits. Thus it is larger even than the square of the currently largest known explicit prime $2^{6972593}-1$, found in June of 1999 by N. Hajratwala, G. Woltman, and S. Kurowski (and verified by D. Willmore using a program written by one of the authors [22]).[2] If F_{24} had turned out to be prime, it would therefore have dwarfed all other known primes. But such was not the case. In fact, what appears to be a reasonable estimate of the "probability" that F_n be prime is attainable via a straightforward sieve-based argument [21]. Since for $n>1$ every prime factor p of F_n must be of the form

$$p = 1 + k2^{n+2}, \tag{1}$$

one may deduce approximate formulae such as

$$\text{Prob}\{F_n \text{ is prime}\} \sim \frac{e^\gamma \log_2(B)}{\log_2(F_n)} \sim \frac{e^\gamma \log_2(B)}{2^n}, \tag{2}$$

[1]This was true at the time the computation was completed, but this record has recently been broken by the PiHex project [25], which used idle time on over 2000 networked personal computers in order to compute the one quadrillionth bit of π, a computation which needed an order of magnitude more computational effort than did the resolution of F_{24}.

[2]Note added in proof: this record was broken on November 14, 2001, when a GIMPS participant's PC flagged $2^{13466917}-1$ as prime; the primality of this number of over 4 million decimal digits was verified 3 weeks later, again using the same author's program [17].

where $\gamma \sim 0.577...$ is the Euler constant and B is the current lower bound (e.g., for $n > 20$ or so, the upper bound for sieve-based factoring) on prime factors of F_n.[3] Since F_{24} has been sieved by various investigators, in regard to the constraint (1), up to roughly $B \sim 10^{20}$ [20], we obtain a probability less than 10^{-5} for F_{24} to be prime. Incidentally, no matter how one twists and turns, any manner of rough summation of the given probability on F_n leads us to the conclusion that there are probably not *any* more prime Fermat numbers beyond F_4, although this argument, even if probabilistically accurate, of course does not rule out the happenstance that there may yet be a prime Fermat number lurking in the as-yet untested reaches beyond our current computational capabilities. We note that Mersenne candidates $M_q = 2^q - 1$ with q itself prime also have a special form $p = 1 + 2kq$ for possible factors, but even though this leads to a qualitatively similar estimate for the probability of an individual M_q being prime, we expect to find Mersenne primes with some regularity, since in a given exponent range there are simply so many more primality candidates among the numbers for which no explicit factor has been found (all primes p_j in the given exponent range for Mersennes, versus only integer powers of 2 for the Fermats). More simply, the different expectations for Fermat and Mersenne numbers result from the fact that $\sum 1/2^n$ converges whereas $\sum 1/p_j$ diverges.

2. Method of proof

The classical Pépin test for primality asserts that if $F = F_n > 3$ is a quadratic nonresidue of an odd prime q, then F is prime if and only if

$$q^{(F-1)/2} \equiv -1 \pmod{F}. \tag{3}$$

This is really just an Euler pseudoprime test—Pépin's contribution was to show that such a test in fact constitutes a rigorous primality test for the F_n. Whereas Pépin used the seed $q = 5$, in modern times it is most typical to choose $q = 3$ (and in fact the reporting of the Selfridge-Hurwitz residues assumes this is the case), and compute the final Pépin residue R_n which we define:

$$R_n = 3^{(F_n-1)/2} \bmod F_n,$$

where by this notation we mean R_n is the least nonnegative residue modulo F_n. Now the Pépin criterion says that F_n is prime if and only if $R_n = F_n - 1$. In the case of primality the final Pépin residue R_n is the largest possible binary value 10....0 modulo F_n, so the highest bit position for residues—namely the 2^{2^n} position—has a '1' if and only if F_n is prime. We continue herein the tradition of Selfridge and Hurwitz by reporting the numbers

$$R_n \pmod{2^{35} - 1, 2^{36}, 2^{36} - 1},$$

for use by future investigators in matters of verification. Note that as a quick check, $F_n,\ n > 5$, is composite if the second Selfridge-Hurwitz residue is nonvanishing. We also stored the complete residue, which will be of use whenever a new factor

[3]Note that these simple sieve-based arguments can sometimes mislead, as in the case of the Riesel-Sierpinski problems, where one asks: How many primes occur in the sequences $R_k = \{k2^n - 1 | n > 0\}$ or $S_k = \{k2^n + 1 | n > 0\}$? The probabilistic argument suggests an infinite number for all nonzero k, but it is well-known that there are in fact no primes at all for some k. However, the Fermat and Mersenne numbers can be considered subclasses with $k = 1$ of this problem where we have sufficient data to justify statistical inferences.

of F_{24} is discovered. The value of such permanent storage is based on the Suyama method for checking compositeness of cofactors [26]. If we can decompose

$$F_n = fG,$$

where f is a (not necessarily prime) known factor, then the cofactor G is prime only if

$$(R_n^2 \bmod F_n) \bmod G \equiv (3^{f-1} \bmod F_n) \bmod G;$$

otherwise, G is composite. The point of having the nested mod operations is that when the product of known prime factors f is a relatively small part of F_n, the number of modular multiplications needed to calculate $3^{f-1} \bmod F_n$ is much less than would be required to perform a direct Fermat or Euler pseudoprime test of the large cofactor G. Additionally, due to the availability of fast discrete weighted transform (DWT) arithmetic [10], each Fermat-mod multiplication with respect to F_n is much more efficient than an operation modulo G. It was in just this way that the hardest cofactors to date, namely of F_{19} and F_{21}, were established as composite [9]. Incidentally there is yet another practical use for a permanently stored residue R_n. First observe that no Fermat number can be a prime power p^k, $k \geq 2$, because, as is easily proven, the Diophantine equation

$$p^k - 4^n = 1$$

for $k > 1$ has no solutions [9]. When a new factor f of an $F_n = fG$ is discovered, one can further use the stored final Pépin residue R_n to determine whether the cofactor G is a prime power. As explained in [9], one approach is to calculate

$$\gcd(3^{fG} - 3^f, G) = \gcd(R_n^2 - 3^{f-1}, G),$$

for if this should be 1, then G is neither prime nor a prime power, i.e., one neatly combines both the Suyama and the prime-power tests into evaluation of a single gcd.

3. Algorithms

Now we turn to the algorithmic issues, the main idea being to calculate (and store some subset of) the Pépin residues $3^{2^k} \bmod F_{24}$. (Readers less interested in the algorithmic and implementational details of the computations may wish to skip ahead to §4 at this point.) It is evident that we need to square essentially random residues of about 2^{24} bits each, a total of $2^{24} - 1$ times.

Let us adopt some nomenclature. Denote a residue x modulo F_n by its digits (in some treatments called a "signal" [10]) $\{x_0, x_1, ..., x_{N-1}\}$ in the sense that we have a generic (i.e. possibly variable-base) expansion

$$x = \sum_{j=0}^{N-1} \left(x_j \prod_{k=1}^{j} W_k \right),$$

where the product is taken to be unity if $j = 0$. Ignoring for the moment the possibility of $x \equiv -1$, which case can easily be handled separately in actual implementation, we see that for the particular case of convolution modulo F_n where the convolution length N is a power of 2, a constant wordsize

$$W = 2^{n/N}, \quad 0 \leq x_j < W,$$

is the most obvious choice, but, as described below, other convolution lengths are also possible. (These require variable wordsizes, but it will suffice to consider the

individual wordsizes W_k to be positive-integer powers of two.) Note that floating-point-based implementations have better controlled error behavior if balanced digits are used; i.e., one forces[4] $x_j \in [-W/2, W/2]$, this being just one of many enhancements discovered in recent times [10]. Now it is an important fact that integer multiplication modulo a Fermat number is equivalent to negacyclic convolution with proper carry [8, 10]. What this means is that for two digit decompositions on residues x, y, the value of $xy \bmod F_n$ can be obtained by adjusting the carries in the negacyclic convolution $(x \times_- y)$ defined elementwise as:

$$(x \times_- y)_m = \sum_{j+k=m} x_j y_k - \sum_{j+k=N+m} x_j y_k.$$

In the case of generating Pépin residues, we only need an "autonegacyclic" convolution, meaning $(x \times_- x)$, and this leads to important simplifications.

Perhaps surprisingly, there are a good many available methods for effecting a negacyclic convolution. For our proof, two basic methods were used, the first based on convolution via floating-point-based fast Fourier transform (FFT), the second via an all-integer recursive convolution algorithm. Though obvious, it should be said right off that all these methods have the same computational goal, and only the pure-integer methods are devoid of round-off problems, so the floating-point methods, while currently the fastest of the lot, are not rigorous until rendered so, either by a rigorous worst-case analysis of floating-point roundoff errors (which is made difficult by the fact that roundoff errors can depend on input, operation, hardware, and rounding mode), or by all-integer verification, the method used here. In any event, *some* kind of independent verification is necessary, irrespective of the type of arithmetic used, simply to rule out hardware error, which can corrupt both floating-point and integer computations, especially when such a long chain of dependent calculations is performed. Briefly, the algorithms we used are 1) floating-point discrete weighted transform (DWT), and 2) Nussbaumer recursive autonegacyclic all-integer convolution. To complete our compositeness proof for F_{24} we used essentially two variants of (1) for the floating-point machines, and two variants of (2) for the integer-based "drones." The key algorithmic and implementational issues arising with these two methods are described next.

3.1. **Floating-point discrete weighted transform (DWT).** The basic method of calculating an autonegacyclic via DFT (and therefore employing FFT methods) is well-known [24]: for digit expansion $\{x_j\}$ one can form a DWT via premultiplication of the jth input digit by g^j, with g a primitive N-th root of (-1), then calculating the usual DFT-based convolution of the weighted signal and scaling the outputs by the inverse weights. This is essentially Schönhage-Strassen multiplication via floating-point FFT [27], but with the signal "weighted" (some authors say "twisted") with an appropriate root of (-1) so that the intended convolution is negacyclic. It is important to note that there are various ways to implement such a DWT, such as real-signal and folded-complex techniques that effectively halve complex signal lengths [8]. Note that the original paper describing DWT-based arithmetic [10] claimed that Fermat-mod DWT requires the base W to be a power

[4]We allow equality at both extremes, since the IEEE floating-point standard [18] allows a signed floating-point number $\pm(x + 0.5)$ to be rounded either to $\pm x$ or $\pm(x + 1)$, depending on whether the nonnegative integer x is even or odd, respectively, in order to prevent subtle systematic errors from creeping into roudoff-sensitive computations.

of two with an exponent that is itself a power of two, e.g., $W = 2^{16}$, but this in not in fact necessary. The key insight here is to understand the fundamentally different nature of Fermat-mod and Mersenne-mod DWT as described in the aforementioned reference: as already mentioned, Fermat-mod DWT is simply the weighting of the convolution inputs by appropriate roots of unity so as to effect a negacyclic result. "Mersenne-mod" DWT, perhaps better referred to as "irrational base" DWT (IBDWT), on the other hand, uses one of two wordsizes for each digit:

$$W_k = 2^{\lceil nk/N \rceil - \lceil n(k-1)/N \rceil},\ k = 1, \ldots, N-1,$$

and premultiplies the jth term of the input signal by a weight factor which is a fractional power of 2:

$$a_j = 2^{\lceil nj/N \rceil - nj/N},\ j = 0, \ldots, N-1.$$

Performing a convolution of this weighted signal and multiplying the outputs by the inverse weights causes the "folding" which occurs naturally in a (cyclic or negacyclic) convolution to occur at the nth bit of the underlying multiword integer without n needing to be a power of two or even to be composite. If the above weighting is used with a cyclic convolution, then the "folded" terms have positive sign, and thus one achieves a convolution modulo $2^n - 1$ without any need for zero-padding of the input vectors or explicit modular reduction, as are needed for large-integer arithmetic with respect to a generic modulus. However, there is no reason the variable-base IBDWT cannot be combined with the negacyclic-convolution input weighting, and thus we can also effect arithmetic modulo $2^n + 1$ for general n without the requirement of a specific form for the vector length. Since the maximum allowable input wordsize for mod-F_{24} convolution using 64-bit floating-point arithmetic is only slightly larger than 16 bits, the above general-length convolution technique was not used in our proof, but it will become useful for convolution modulo larger Fermat numbers, specifically those for which input wordsizes must be smaller than 16 bits for accuracy reasons.

For the floating-point scenario, it is useful to embark on a rough derivation of expected convolution output wordsizes and roundoff error, in order to provide useful estimates of optimal transform lengths. There have been over the years interesting attempts at rigorous error bounding, but such bounds tend to be over-conservative in practice, as one might expect. So for the present we shall not attempt rigorous error bounds, but rather sketch some plausible heuristics which lead to accuracy estimates that in fact show close agreement with numerical experiment over a wide range of transform lengths, and which allow one to make useful predictions, especially regarding accuracy degradation of floating-point convolutions at very long vector lengths.

3.1.1. *Optimal floating-point convolution length.* To estimate how the maximum allowable wordsize (and hence minimum transform length) depends on the algorithm and the underlying precision, we can make use of random-walk statistics to quantify the magnitude both of convolution terms and of floating-point errors. We consider a residue modulo F_{24} (say) to be a signal of N digits, each digit (word) of size W; that is, $NW \approx 2^{24}$. Let a unit-stepsize (± 1) random walk have position $x(s)$ after s steps. By the Fisher theorem [13],

$$\limsup_{s \to \infty} \frac{x(s)}{\sqrt{\frac{s}{2} \log \log s}} = 1, \tag{4}$$

where the limit is attained not in some most-probable sense, but in fact with probability one. The obvious question is, in what sense is an FFT-based autoconvolution (squaring) for such a signal like a random walk? Using balanced-digit representation is crucial here, since the inputs to the transform are then essentially random digits in [−W/2, +W/2]. (The various DWT scale factors and variable-base representation of the IBDWT do not change these properties qualitatively.) Now it might seem simplest to model the $O(\log_2 N)$ passes of the FFT as the discrete steps of a random walk, but this is problematic because the "stepsize" then increases with each pass. But since the FFT is mathematically equivalent to a DFT, we can simply consider the simpler DFT, each of whose outputs is the weighted sum of the N input digits, the weights being simply roots of unity, which (up to constants of proportionality) do not alter the random-walk statistics, at least as far as deviation magnitude is concerned. After the forward FFT, using (4), the most-probable maximum displacement has complex magnitude of order

$$\frac{W}{2} \cdot A(N) = \frac{W}{2}\left(\frac{N}{2}\log\log N\right)^{1/2}, \tag{5}$$

where the function $A(N)$ can be viewed as an amplification factor acting on the input signal. We then do a pointwise squaring, which squares the above displacement, followed by an inverse transform which amplifies the magnitude of the largest dyadic squaring output by the same factor A as above, but also divides by the complex transform length $N/2$. Thus, we obtain a maximum convolution output of order

$$\frac{W^2}{4}\left(\frac{N}{2}\right)^{1/2}\left(\log\log N\right)^{3/2}. \tag{6}$$

The *average* output, on the other hand, has order $\frac{W^2}{4}\sqrt{\frac{2N}{\pi}}$.

Roundoff error sizes, in contrast to convolution outputs, depend crucially on the structure of the computation. Were we to do a matrix-multiply DFT, rounding errors would behave like the convolution terms, i.e., the average error would scale as $O(\sqrt{N})$. Using forward-FFT/dyadic-square/inverse-FFT, on the other hand, behaves with respect to relative error like a random walk of $O(2\log_2 N)$ steps; hence the average relative error scales as $O(\sqrt{\log_2 N})$ times machine epsilon. This grows very slowly with increasing N, as does the expected maximum error, which is $O(\epsilon\sqrt{\log_2 N \log\log\log_2 N})$. Since the maximum error is an asymptotic quantity which converges extremely slowly (cf. Exercise VIII.7.9 of [13]) and the "number of steps" $2\log_2 N$ here is generally quite small, it is actually better to use the average relative error (which settles down much more quickly to its asymptotic behavior) in estimating maximum wordsize; we thus use the simpler estimate that the maximum relative error is asymptotic to $C\sqrt{\log_2 N}$), where C is larger than unity but still expected to be order of unity as long as N is not extremely large.

The input wordsize to the convolution is then limited by the requirement that the accumulated round-off error must remain small enough to permit one, during the round-and-propagate-carries phase of each convolution-based multiply, to confidently round each output digit to the nearest integer. For example, if a typical output digit x_j has fractional part (which we define as $\mathrm{frac}(x_j) = |x_j - \mathrm{nint}(x_j)|$, which by definition is in the interval $[0, 0.5]$) no larger than 0.1 we are safe, but when fractional parts approach 0.3-0.4 we are dangerously close to an incorrect

rounding, and a fractional part of 0.5 (especially if it occurs repeatedly) virtually guarantees that a catastrophic loss of precision has occurred. That is, we require the expected maximum absolute error in the convolution outputs to be appreciably less than one, let us say less than one-fourth:

$$\epsilon C \frac{W^2}{4} \left(N \log_2 N \right)^{1/2} \left(\log\log N \right)^{3/2} < \frac{1}{4}. \tag{7}$$

Letting $B_{mant} = -\log_2 \epsilon$ be the number of mantissa bits of our floating-point representation, we have, upon taking the base-2 logarithm of (7),

$$2\log_2 W + \log_2 C + \frac{1}{2}\left[\log_2 N + \log_2 \log_2 N\right] + \frac{3}{2}\left[\log_2 \log\log N\right] < B_{mant}. \tag{8}$$

Using IEEE 64-bit precision with $B_{mant} = 53$, for a vector length of 2^{20} this formula predicts a maximum input wordsize of roughly $(20 - \log_2 C)$ bits, and taking $C = 2$ gives a result closely matching the upper limit of roughly 19 bits yielded by our computational tests. We also tested how well the formula predicts trends at even longer runlengths (e.g., those relevant to the current smallest F_n of unknown status); for F_{31}, for example, it predicts that a runlength of $15 \cdot 2^{23}$ might just be feasible, and a numerical test consisting of 10000 modular squarings of an F_{31}-sized number indeed produced no warnings about roundoff errors near 0.5 and a residue which matched that of a run at the slightly larger vector length of 2^{27}, whereas a run at length $14 \cdot 2^{23}$ failed in the first 100 squarings due to autodetection of fatal roundoff errors. Of course F_{31} is now known to be composite, so of greater current relevance is that a runlength of 2^{29} is predicted to be more than adequate for performing mod-F_{33} floating-point convolutions. Also of interest is that maximum allowable wordsize is predicted to drop below 16 bits per input digit at or just beyond F_{35}, at which point the ability to combine a Fermat-mod DWT and variable-base IBDWT, and thus to use an average input wordsize only slightly less than 16 bits, will become very relevant indeed. Neglecting the higher-order logarithmic terms (and independently of the choice of C), the above estimate also predicts that for each doubling of the runlength we lose roughly 0.25 bits from the maximum allowable wordsize, also very close to what is observed in practice, e.g., from the vast computational experience of the GIMPS project [15].

We thus see that for F_{24} work a wordsize $W = 2^{16}$ is in fact somewhat smaller than what is allowable, but the savings resulting from a more aggressive setting of the runlength (say $N = 7 \cdot 2^{17}$) are approximately offset by the increased complexity of the combined negacyclic DWT/IBDWT needed to effect such a nonpower-of-2 runlength convolution. Thus, both of our two independent floating-point proofs used a runlength of 2^{20}. The average fractional error for the code used for wavefront 1 (see §3.1) was $\approx 3.55 \times 10^{-4}$ (and after climbing from zero to this value during the first few dozen squarings, deviated from this mean value by less than 5% for the rest of the run), so a fatal fractional error ≤ 0.5 should, in the absence of hardware error, occur with a probability corresponding to an event lying some 1400 standard deviations outside the mean. While not rigorous, this estimate indicates that if there is to be a fatal error in such a computation, it is overwhelmingly more likely to be due to a hardware (or software) error than to actual floating-point convolution roundoff errors.

3.2. **FFT Algorithms.** From an algorithmic perspective, we shall be considering the two major approaches to the FFT, the Cooley-Tukey or decimation-in-time

(DIT) transform, which begins with bit-reversal-reordered input data and outputs an ordered transform vector, and the Gentleman-Sande or decimation-in-frequency (DIF) transform, which begins with ordered input data and outputs a bit-reversed transform vector [7].

In deciding on one's particular implementation of option (1), one must consider the important recent discoveries in regard to large-signal FFTs, in particular, the "parallel" or "four-step" FFT [1, 2, 3, 12], which essentially allows the mapping of a one-dimensional FFT to a row-column matrix FFT. For various reasons, neither of the two independently developed floating-point codes used herein does a standard four-step FFT, although they do have the same aim, namely to allow for excellent performance of the fundamentally data-nonlocal FFT algorithm on modern, multilevel-cache-based microprocessor architectures. In this regard, the most crucial levels of the memory hierarchy are the two closest to the FPU itself, namely the registers (which can be viewed as a level-zero "L0 cache") and the on-chip data (L1) cache. It is desirable to do as many operations as possible on the data while they are in registers, which naturally leads to higher-radix FFTs, and to move data, especially ones that are widely separated in terms of their indices, in and out of memory with as few L1 cache conflicts as possible, i.e., to minimize thrashing. This requires above all that one carefully consider data access patterns, not only in terms of absolute locations in linear memory, but also how these map to the much smaller high-speed data caches of the processor.

FFT Algorithm 1, used by the second author (EWM) in his proof,[5] is essentially the so-called "ping-pong" FFT algorithm described in [12], with several enhancements, the most crucial being (i) padding of the data arrays to prevent large power-of-two strides, which are known to lead to cache thrashing, and (ii) higher-radix FFT passes to reduce the number of passes through the data. Beyond these basic measures, several additional optimization opportunities were exploited. In our test of F_{24}, the complex transform length was $N = 2^{19}$, and each squaring began with ordered input data, did a forward DIF transform using the set of radices $\{8, 16, 16, 16, 16\}$, a dyadic squaring of the (now bit-reversed) output data, followed by an inverse DIT transform using radices $\{16, 16, 16, 16, 8\}$ and a rounding-and-carry-propagation step. The combination of DIF and DIT transforms avoids the need for any explicit bit-reversal reordering, which, although not overwhelming, can consume 10-20% of the execution time in a single-FFT (i.e., DIF or DIT used for both the forward and inverse transform) implementation. Moreover, this time fraction tends to increase rather than decrease with N (even though the relative arithmetic operation count decreases), as bit-reversal is difficult to do in a cache-friendly way, even with array padding. The reason we reverse the order of the radices (which is necessary for coprime radices, but not for power-of-two transform lengths) in doing the inverse transform is related to one final optimization stategy permitted by the above-described data movement scheme. Whenever a radix-R DIT pass is preceded by a radix-R DIF pass (or vice versa) with few or no intervening operations, one can eschew the gather-scatter phases of the two passes (which do impose a cost in terms of loads and stores, and thus potential cache misses) and instead roll the two passes and the intervening operations into a single in-place pass through the data. During each Pépin squaring, there are two opportunities

[5]The source code for this implementation is available at `http://www.perfsci.com/F24/-F24floatem.zip`.

for such streamlining: the two passes (final DIF pass, initial DIT pass) bracketing the dyadic squaring, and the two (final DIT pass, initial DIF pass) bracketing the carry propagation step. For this to work, however, the radices of the two adjacent passes must be identical; hence our reversal of the order of radices during the inverse transform. We also used the right-angle-transform idea described in [10], because it eliminates the need for real-complex wrapper passes at the end of the forward and prior to the inverse transform, and thus makes the dyadic squaring step trivial, even in the presence of bit-reversed data. By exploiting all of the aforementioned optimization opportunities, the per-squaring machine time on Wavefront machine 1 (a Silicon Graphics Octane workstation with a 250MHz MIPS R10000 microprocessor) was a mere 0.85 seconds, representing a reduction in overall runtime by a factor of roughly five over a naive (but unfortunately widely used) radix-2-pass FFT implementation.

FFT Algorithm 2, used by the third author (JSP) in his proof,[6] also uses a discrete weighted transform to square numbers modulo F_{24}, and also uses the fully complex right-angle variant to keep the pointwise squaring simple and avoid complicated real-valued FFTs. The primary goal here is to efficiently use the limited amount of high-speed storage available to a modern microprocessor. Algorithm 2 begins with Bailey's 4-step FFT [3, 7], simplified for in-place convolution. As with Algorithm 1, both DIF and DIT transforms are used, allowing the data to be processed in-place and in order, and making unnecessary the matrix transposes of Bailey's algorithm.

Although both wavefronts use the same style of DWT to effect Fermat-mod squaring, FFT Algorithm 1 is quite different from FFT Algorithm 2. The former emphasizes uniform but highly nonlocal data access, whereas the latter tries to load blocks of data into high-speed memory and keep them there as long as possible. Whereas Algorithm 1, in its final form, performs 14 passes through its data and requires an eight-megabyte array for scratch space, Algorithm 2 performs the convolution in-place, makes exactly three passes through the residue per squaring and needs little storage beyond the residue itself. F_{24}-mod squaring requires an FFT multiply of size 2^{19}; Algorithm 2 used a single radix-32 pass to break the problem into chunks somewhat smaller than the external cache, then processed each chunk recursively.

Wavefront 2 was a 167-MHz Sun UltraSPARC-1. While the off-chip secondary cache on this machine can deliver data at high speed, the C compilers available had a difficult time scheduling code to use it effectively. This limitation meant that none of the time-critical portions of the code could be entrusted to a C compiler, and the final version includes large amounts of hand-written assembly code. The program used the FFTW library [14] for the first half of the Pépin test and managing a modular squaring in 1.1 seconds; switching finally to hand-coded FFTs which overcame the aforementioned limitations in the compilers available for the UltraSPARC brought that down to just 0.885 seconds per squaring.

3.3. **Nussbaumer convolution.** This method involves no actual numerical FFTs, only "symbolic" ones [8]. A negacyclic convolution is built recursively, using smaller negacyclic ones. The digit size W is entirely flexible; for example, $W = 2^{512}$ yields

[6]The source code for this implementation is available at `http://www.perfsci.com/F24/-F24floatjp.zip`.

a signal length of only $N = 2^{15}$, at the expense of size-W' multiplications and additions, where W' is slightly larger than W.

Our implementation of option (2), Nussbaumer convolution, is an adaptation of established software for general integer convolution [8]. Such software has been used in many disparate domains, ranging from number theory to signal processing, but was in fact pioneered by J. P. Buhler for the purpose of numerical investigations on Fermat's "last theorem" (FLT) [4, 5, 6]. It has been known to Buhler and colleagues for about a decade that integer convolutions of lengths into the millions are indeed possible in error-free fashion, on conventional (even by 1980s standards) machinery, as no floating-point arithmetic is involved. Thus the scheme was a natural choice for checking the Pépin residues via "drone" machines, as described in Section 4.

Many optimizations are possible beyond the original description of Nussbaumer, and even beyond the variant so successful for FLT computations. Let us name a few optimizations, to convey the kind of thinking that improved the speed of proof. First, the small negacyclics at the bottom recursive level of Nussbaumer can be done especially fast because they turn out to be autonegacyclics. For example, a length-4 autonegacyclic can be done—amazingly enough—in 3 multiplies and 4 squares, for an equivalent count of just 5 multiplies, which is very much faster than the naïve 16 multiplies. Second, special FFT structure as discussed next, but as applied to the symbolic FFTs within Nussbaumer recursion, results in substantial improvement. Third, there are special ways to combine (or even remove) many transposition operations and memory motion, to exploit the known cache behavior on certain machinery. All of these together resulted in an implementation of option (2) that required around 5 CPU-seconds per squaring on a 500MHz Apple G4 processor, and comparable but somewhat longer times on similar-frequency Pentium II machinery. Note that either manifestation on one "wavefront" machine would have required several years years to complete the proof. As it was, the floating-point option implementation detailed in the foregoing section needed just 6 CPU-months on hardware significantly less than cutting-edge. Owing to this order-of-magnitude speed advantage over the best available all-integer algorithms, the floating-point machinery emerged as the natural "wavefront" candidate.

4. "Wavefront" generation

In our proof of compositeness we used a method pioneered by C. Norrie [9] on F_{22}. Elegant in its simplicity, this method is to use fast machines to deposit—at convenient intervals—the Pépin residues $3^{2^k} \bmod F_{24}$. Having these in storage for various k, one may check the deterministic link between, say, the k_1-th square and the k_2-th square; the point being that this checking can be done on a relatively slow (for whatever reason) machine. Thus, a fast "wavefront" machine is expected to deposit Pépin residues at the highest possible rate, with a host of slower machines, or "drones," acting in parallel fashion to test each link in the resulting Pépin chain.

We chose to divide the labors cleanly into: two wavefront machines running the independently developed floating-point DWT squarers described in the preceding section and squaring at similar speeds (and thus making fairly frequent cross-checks convenient to do), and a set of drones each of which ran an *integer* squarer via Nussbaumer convolution, each drone beginning with a selected squaring residue—call it the a-th square—previously deposited by one of the wavefront machines,

then performing a total of $(b-a)$ squarings modulo F_{24}, eventually comparing the resulting residue with a previously-deposited b-th square.[7]

Incidentally we cannot resist here a cautionary anecdote about extremely long computations, which fortunately has a happy ending. The very first complete Pépin test of F_{24} was actually carried out by one of us (EWM) in 1998-99, with a residue that ultimately was shown erroneous by both an alternative floating-point run and an integer-convolution run. In fact this faulty run was a primary motive for continued optimization and deployment of our pure-integer algorithm. This first run was begun in June 1998 and finished in early February 1999, and (not surprisingly) indicated that F_{24} was composite. The reason a second run was not performed in parallel is that this author had access to one fast dedicated machine, but not to another which could even come close to keeping up with the first. There was reason to expect that a second fast engine might become available during the course of the first test, but this did not come to pass, and so a second run (using an improved version of the code) on the same machine was planned.

During the initial run, the bottom 64 residue bits were saved every 2000th squaring, and when the other two authors of the present joint paper (whose own floating-point run was then about one-eighth of the way to completion) got the rather surprising news that someone unknown to them had completed an initial test of F_{24}, they naturally wanted to do some cross-checking against their early residues. This cross-checking turned up a discrepancy around the 150000-th squaring, and was eventually traced to a faulty residue file which was subsequently used to restart the initial run following a power outage during a thunderstorm. The precise source of the error was a flawed hardware conversion of eight-byte floating residue data to the 2-byte signed integer form used for the savefile. A previous test run of F_{22} had been deliberately interrupted and restarted several times, yielding a correct final result, but that run had used default 4-byte signed integers for the savefile data. During preparations for the assault on F_{24}, the 4-byte save format was changed to 2-byte almost as an afterthought, in order to halve the size of the (then) four-megabyte savefiles. The resulting code passed all the 1000-squaring self-tests used for program validation but, in a crucial oversight, was not subjected to a validation test involving a restart-from-interrupt. The savefiles also did not contain the simple expedient of a checksum, and the kind of error that occurred (an incorrect 2-byte integer resulting from conversion of an 8-byte float) was of a kind not detectable by the extensive round-off-error-related consistency checking built into the program.

In any event, as soon as it became clear that the first result was undoubtedly erroneous, it was decided to join forces: two floating runs using the separate codes would execute independently, but with frequent cross-checking of residues mod 2^{64}, and all-integer verification hopefully keeping pace after sufficient hardware was assembled for that purpose. A second run by EWM using a revised version of his code was launched in late February, which caught up with JSP's test in early summer 1999. Further improvements by JSP to his code meant that, as of late June 1999 the two floating runs were dispatching squarings at roughly the same rate, and in fact they finished within days of each other, on 27 and 31 August 1999, respectively, with exactly matching final residues. The third and final link, the integer verification, was completed about a month after the second floating run.

[7]The source code for this implementation is available at `http://www.perfsci.com/F24/-F24int.rcjp`.

5. Conclusions, and comments on the future

We hereby report that F_{24} is composite, the Selfridge-Hurwitz residues for $n = 24$ being, in decimal,

$$R_{24} \equiv \begin{cases} 32898231088 & \mod\ 2^{35}-1 \\ 60450279532 & \mod\ 2^{36} \\ 68627742455 & \mod\ 2^{36}-1. \end{cases}$$

Furthermore, the lower 64-bit word of the final residue is, in hexadecimal and decimal respectively:

$$R_{24} \mod\ 2^{64} = 17311\text{B}7\text{E}131\text{E}106\text{C}_{16} = 16711471650317313 08_{10}.$$

The total elapsed time for the floating-point wavefront machines was about 200 days (either implementation), while as expected about $5-10$ drone machines running the integer autonegacyclic convolver kept up essentially with the wavefront speed, and as we imply established integrity for every link in the Pépin chain. We submit that there can be no doubt that F_{24} is composite.[8]

What about the next Fermat number of unknown character? At the time of this writing that would be F_{33}. It is fair to say that the only hope for resolving the character of F_{33} in the near term would be sieving with respect to the form (1) (and of course that has been tried roughly up to 20 decimal digits for possible factors). Given the 75-bit factor recently found for F_{31}, which required several CPU-months, together with the fact that sieving to a given depth is 4 times cheaper for F_{33} than it is for F_{31}, it seems one should sieve to a limit of 2^{80} or slightly greater before expending CPU time on other options. However, it is interesting to consider what recourse one may have if sieving fails, as it well may, since even a fairly modest 30-digit factor will be out of reach via this route without inordinate computational effort.

Factors of F_m have the special form $k \cdot 2^{m+2} + 1$, so Pollard's $p-1$ method [26] is the obvious next choice, since even for fairly large factors, k can be quite modest in size – a 75-bit factor of F_{33} would have only a 40-bit k, which itself would have a reasonable likelihood of having no factors larger than 20 bits, in which case it would be quite likely to be discovered by a concerted $p-1$ effort, as discussed below. The odds also are boosted by the fact that for $p-1$ we do not need the smallest factor to have a smooth k; rather, we merely need *any* factor p to have a sufficiently smooth k. The elliptic curve method (ECM) relies on a similar smoothness property, but the memory requirements for doing ECM (in particular stage 2) on numbers of this size are at present too large to make the algorithm practical.[9]

Here is a brief summary of how the $p-1$ method might be applied to a number the size of F_{33}. First off, we do not consider disk-based transforms for the large-integer multiplies, since these would simply be too slow, and reliability also is much lower, as one is no longer dealing with solid-state components as is the case

[8]For the sake of completeness, we also generated two matching Pépin residues for F_{23}, and via the Suyama test determined that the known cofactor of this number, $F_{23}/(5 \cdot 2^{25}+1)$, is composite.

[9]One should not rule out any approach until such is known for sure to be inefficient. There is always the possibility of a parallel Pollard-rho scheme, as enunciated in [11]. In that work it is explained how the new McIntosh–Tardif factor of F_{18} could have been found with a number of machines all effecting a parallel Pollard "rho" scheme.

when the dataset fits into RAM memory. We also prefer not to deal with massively parallel implementation, since interprocessor communication then becomes the dominant factor limiting speed, and efficiency (as measured in total CPU-time) drops accordingly, especially for fundamentally data-nonlocal algorithms such as the FFT. Rather, our aim is to establish whether the needed arithmetic can be performed in the fast RAM memory and a single processor of a high-end workstation system. If this proves true, one has immediately gained a large boost in stage 2 of the algorithm, since various stage 2 intervals can be split up among as many uniprocessor machines as are available with sufficient memory and speed for the computation. The maximum wordsize estimate (8) indicates that an FFT-based convolution modulo F_{33} would permit a wordsize of 16 bits, i.e., would require an input vector of $N = 2^{29}$ 64-bit floating-point elements. Using an in-place FFT scheme, this yields 4 gigabytes (GB) of storage for the convolution data, plus a small additional amount (on the order of several megabytes) for array padding elements and small tables storing precomputed DWT weights and complex roots of unity. For the modest stage 1 bounds which would be feasible with a number this large, an efficient stage 1 implementation could use a precomputed product of all the stage 1 prime powers, permitting a left-to-right binary exponentiation scheme. The advantage of the latter is that it can be done entirely in-place, as it needs no secondary accumulated power to be stored, and uses only modular squarings and multiplies by the initial seed, the latter being a small scalar. Thus, enough memory for stage 1 is readily available on workstation-class machinery. Regarding the processing time needed, our tests of an advanced-prototype in-place FFT indicate a time on the order of one minute per modular squaring on a Gigahertz-class processor. Using the precomputed-primes-product and exponentiation scheme mentioned previously, this will permit a stage 1 to a prime limit of 500,000 to be run in around one year of dedicated computation. Even though stage 1 is not parallelizable in the same fashion as stage 2, this time estimate could be cut appreciably (or a deeper stage 1 run) by exploiting the symmetric multithreading (SMT) supported by an increasing percentage of microprocessors, which gains the benefit of a parallel implementation when multiple processors are available, while requiring relatively little additional code.[10] Stage 2, on the other hand, is rather more memory-intensive. We estimate that the minimum memory needed to perform a reasonably efficient stage 2 is eight times that needed for stage 1; this permits us to store the current residue, plus seven small even powers of the stage 1 residue r, e.g., r^2, r^4, r^6, r^8, r^{10}, r^{12} and r^{14}. Any prime gaps not covered by these precomputed powers would need additional multiplies, but such gaps (for the stage limits which are feasible for such large numbers) occur with sufficiently low frequency that they would slow the computation by only a small amount. For primes on the order of one million the maximal gap is slightly larger than 100, so even in this worst-case scenario we need no more than eight multiplies by the above precomputed powers to cover the gap. Storage of eight F_{33}-length residues in floating-point form needs 32 GB of memory, an amount which will be available in the near future on high-end compute servers. Note that even stage 1 of ECM requires on the order of 10 to 15 full-length vectors

[10]Some early protoypes of such a large-integer multiply implementation appear quite promising; for example an FFT-based modular multiply of an F_{24}-sized number executed on a 4-processor workstation in less than one-third the time needed on a single processor of the same type and speed [16].

to be stored [32], so to do any ECM work whatsoever will need roughly double this amount of memory.

Finally, if over the years factoring continues to prove fruitless, interest may again turn to direct nonfactoring resolution. The fact is, a single FFT-based convolution modulo F_{33} would, on a state-of-the-art workstation, consume around one CPU-minute, so that the complete Pépin test on our current brands of machinery would require many millennia. But we have already remarked on the effective rate of resolution for the F_n, and noted that one cannot estimate such achievements based on *current* resources. As gigabit-per-second networking technology becomes commonplace, massively parallel (MP) computation using large clusters of cheap commodity-microprocessor-based computers will become the norm for many large computational tasks, and a clever MP implementation of the Pépin test, perhaps using many discrete moduli and CRT reconstruction, might bring F_{33} into reach. On the basis of these vague heuristics, but more in keeping with the way that computational history has actually evolved, one might expect F_{33} to be resolved within the next two or three decades, or at worst, well before the year 2100 A.D.

6. Acknowledgments

We heartily thank J. Buhler (Mathematical Sciences Research Institute, Berkeley) for his original (1990) implementation of Nussbaumer convolution; P. Montgomery (Microsoft Research), H. Lenstra (University of California, Berkeley), J. Selfridge (Northern Illinois University), and C. Pomerance (Bell Laboratories) for their theoretical and heuristic insights, J. Klivington (Apple Computer), for his fast Apple G4 convolution, and G. Woltman for his engineering expertise. We are grateful to R. Knapp, P. Wellin, S. Wolfram (Wolfram Research, Inc.) for their algorithm support of various kinds. We thank C. Curry (University of Southern Mississippi), K. Dilcher and R. Milson (Dalhousie University) and especially A. Kruppa and the staff of the Infohalle der Fakultät für Informatik an der Technischen Universität München for their tireless integer convolution runs. EWM thanks J. Alexander of Case Western Reserve University for generous access to the SGI Octane used for wavefront 1. JSP thanks the management at 3S Group Incorporated for the extended use of the UltraSPARC fileserver used for wavefront 2. We also acknowledge a grant from the Number Theory Foundation in the later stages of our work, which grant significantly enhanced the all-integer effort. We are indebted to Apple Computer for resources pertinent to the new G4 processor. A Reed College team of staff and students: N. Essy, B. Hanson, C. Chen, J. Dodson, R. Richter, W. Cooley, J. Heilman, D. Turner and (from the University of Georgia) C. Gunning finished in glorious and selfless fashion the last stages of the all-integer proof. Lastly, we acknowledge the many insightful and constructive criticisms of one of the anonymous reviewers of the initial manuscript.

References

1. R. C. Agarwal and J. W. Cooley,"Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility," *IBM Journal of Research and Development* **30** (1986), 145 - 162. CMP 18:12
2. M. Ashworth and A. G. Lyne, "A Segmented FFT Algorithm for Vector Computers," *Parallel Computing* **6** (1988), 217-224. CMP 20:07
3. D. Bailey, "FFTs in External or Hierarchical Memory," (1989) manuscript.

4. J. Buhler, R. Crandall, R. Ernvall, T. Metsankyla, "Irregular Primes to Four Million," *Math. Comp.* **61**, 151–153, (1993). MR **93k:**11014
5. J. Buhler, R. Crandall, R. W. Sompolski, "Irregular Primes to One Million," *Math. Comp.* **59**, 717–722 (1992). MR **93a:**11106
6. J. Buhler, R. Crandall, R. Ernvall, T. Metsankyla, A. Shokrollahi, "Irregular Primes and Cyclotomic Invariants to Eight Million," manuscript (1996).*
7. C. Burrus, *DFT/FFT and Convolution Algorithms: Theory and Implementation*, Wiley, New York, 1985.
8. R. E. Crandall, *Topics in Advanced Scientific Computation*, Springer, New York, 1996. MR **97g:**65005
9. R. Crandall, J. Doenias, C. Norrie, and J. Young, "The Twenty-Second Fermat Number is Composite," *Math. Comp.* **64** (1995), 863-868. MR **95f:**11104
10. R. Crandall and B. Fagin, "Discrete Weighted Transforms and Large-Integer Arithmetic," *Math. Comp.* **62** (1994), 305-324. MR **94c:**11123
11. R. Crandall, "Parallelization of Pollard-rho factorization," manuscript, `http://www.-perfsci.com`, (1999).
12. R. Crandall and C. Pomerance, *Prime numbers: a computational perspective*, Springer, New York, (2001) MR **2002a:**11007
13. W. Feller, "An Introduction to Probability Theory and Its Applications," Vol. I, 3rd ed., Wiley, New York, 1968. MR **37:**3604
14. M. Frigo and S. Johnson, "The fastest Fourier transform in the west," `http://theory.-lcs.mit.edu/fftw`.
15. E. W. Mayer, GIMPS Source Code Timings Page, `http://hogranch.com/mayer/-gimps_timings.html#accuracy`.
16. G. B. Valor, private communication (2001).
17. GIMPS homepage, `http://www.mersenne.org`.
18. *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, IEEE (1985).
19. W. Keller, Fermat-number website data: `http://www.prothsearch.net/fermat.html`.
20. W. Keller, private communication (1999).
21. H. Lenstra, private communication (1999).
22. E. Mayer, Mlucas: an open-source program for testing the character of Mersenne numbers. `http://hogranch.com/mayer/README.html`.
23. R. R. Schaller, *Moore's law: past, present and future*, *IEEE Spectrum* **34** (1997), 52-59. (Also cf. `http://www.intel.com/research/silicon/mooreslaw.htm`.)
24. H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, 2nd ed., Volume 2 of Springer Series in Information Sciences, Springer, New York, 1982. MR **83e:**65219
25. C. Percival, PiHex: A distributed effort to calculate Pi. `http://www.cecm.sfu.ca/-projects/pihex/index.html`.
26. H. Riesel, *Prime Numbers and Computer Methods for Factorization*, 2nd ed., Birkhäuser, Boston, 1994. MR **95h:**11142
27. A. Schönhage 1971, Schnelle Multiplikation grosser Zahlen, *Computing* **7** (1971) 282-292. MR **45:**1431
28. J. L. Selfridge and A. Hurwitz, "Fermat numbers and Mersenne numbers," *Math. Comp.* **18** (1964), 146-148. MR **28:**2991
29. V. Trevisan and J. B. Carvalho, "The composite character of the twenty-second Fermat number," *J. Supercomputing* **9** (1995), 179-182.
30. G. Woltman, private communication (1999).
31. J. Young amd D. Buell, "The Twentieth Fermat Number is Composite," *Math. Comp.* **50** (1988), 261-263. MR **89b:**11012
32. P. Zimmerman, private communication (2001).

*Added in proof. Their latest paper, going up to 12 million, appeared in J. Symbolic Comput. **31** (2001), 89–96. MR **2001m:**11220

Center for Advanced Computation, Reed College, Portland, Oregon 97202
E-mail address: crandall@reed.edu

Department of Mech. & Aerospace Engineering, Case Western Reserve University, Cleveland, Ohio 44106
E-mail address: ewmayer@aol.com
Current address: 10190 Parkwood Dr. Apt. 1, Cupertino, CA 95014

Department of Elec. & Comp. Engineering, University of Maryland, College Park, Maryland 20742
E-mail address: jasonp@boo.net