Congruent number theta coefficients to 10^{12}

Robert Bradshaw, William B. Hart [*], David Harvey,
Gonzalo Tornaria [†], Mark Watkins [‡]

September 23, 2009

Abstract

We report on a computation of congruent numbers, which subject to the Birch and Swinnerton-Dyer conjecture is an accurate list up to 10^{12}. The computation involves multiplying large theta series as per Tunnell (1983). The first method, which we describe in some detail, uses a multimodular disk based technique for multiplying polynomials out-of-core which minimises expensive disk access by keeping data truncated. The second technique uses "Bailey's four-step" Fast Fourier method in combination with compression of the data to disk in intermediate stages.

1 Introduction

1.1 History

The congruent number problem first makes its appearance in the literature of the classical Islamic period, e.g. in al-Karaji's text the al-Fakhri, written in honour of the ruler of Baghdad at that time. Dickson [11] states that an anonymous Arab manuscript written before 972 A.D. also contains reference to the problem.

The problem was initially studied in terms of squares of rational numbers. It can be stated as follows: a natural number n is *congruent* iff there exist rational numbers x, y, z, w such that

$$x^2 + ny^2 = z^2 \text{ and } x^2 - ny^2 = w^2.$$

In other words n is congruent iff there exist three rational squares in arithmetic progression with common difference n.

Bachet, in translating Diophantus' Arithmetica, wrote an appendix of problems on right triangles. Problem 20 was "to find a right-angled triangle such that its area is equal to a given number". This equivalent problem refers to right triangles with rational sides whose area n is a natural number.

[*]Supported by EPSRC grant number EP/G004870/1

[†]supported by duct tape, string and grant numbers blahdiblah

[‡]All authors were supported at workshops administered by AIMath under NSF Grant number DMS-0757627

The problem was studied by Fermat and Fibonacci the latter of which referred to a common difference of squares in arithmetic progression as a *congruum*. Euler referred to such numbers as *congruere* meaning to "come together".

Many other authors contributed to the study of the properties of and computation of congruent numbers, including Alter, Curtz and Kubota [1] who conjectured that if n is congruent to 5, 6 or 7 modulo 8 then n is a congruent number. This was shown to be true, subject to the weak Birch and Swinnerton-Dyer conjecture by Stephens [35] in 1975.

The earliest computations of congruent numbers are again due to the classical Islamic mathematicians, the congruent numbers 5, 6, 14, 15, 21, 30, 34, 65, 70, 110, 154, 190, 210, 221, 231, 246, 290, 390, 429, 546 and ten other substantially larger congruent numbers being known to them.

Fibonacci, Genocchi and Gérardin added 7, 22, 41, 69, 77 and forty-three other values below 1000.

Fermat showed that 1 is not congruent in 1659, something which had been stated but not proved by Fibonacci in 1225. By scaling this is equivalent to the fact that no square number can be congruent.

Bastien [5] observed that numbers which are prime and 3 modulo 8, products of two such primes, twice a prime which is 5 modulo 8, twice a product of two such primes or twice a prime which is 9 modulo 16 are not congruent.

Numerous congruent numbers were demonstrated by Alter, Curtz and Kubota [1] and by Jean Lagrange in his thesis [22]. See Guy [16] for further details on the history of the computation of congruent numbers.

More recently Monsky showed that, for example, two times the product of primes $p \equiv 1 \pmod 8$ and $q \equiv 7 \pmod 8$ with $(p/q) = -1$ is a congruent number. For a history of results along these lines see Feng [13]. Also see [26].

By 1980 there were still numerous values below 1000 which were not decided either way.

Congruent numbers up to 2000 had been computed by 1986 [25] and online tables exist for congruent numbers to 10000 [28]. Subject to a conjecture of Birch and Swinnerton-Dyer (see below), Mike Rubinstein had computed all congruent numbers up to 10^9 in ???? [32] and the current authors had raised the limit to 2×10^{10} by 2008. With this paper, the current plateau is now 10^{12}.

1.2 Relating congruent numbers to elliptic curves

If three rational squares in arithmetic progression have common difference n, their product is a square:

$$v^2 = (u^2 - n)u^2(u^2 + n) = (u^2)^3 - n^2(u^2).$$

This shows immediately that if n is congruent then it corresponds to a point (u^2, v) on the elliptic curve $E_n : y^2 = x^3 - n^2x$.

Along similar lines, in 1877 Lucas showed that n is congruent iff $y^2 = x^4 - n^2$ has a positive rational solution.

The group of points on the curve E_n is isomorphic to $(\mathbb{Z}/2\mathbb{Z}\times\mathbb{Z}/2\mathbb{Z})\times\mathbb{Z}^r$ where r is the rank. The three non-trivial 2-torsion points do not yield congruent numbers and so n is congruent iff E_n has positive rank.

There has been considerable interest in verifying that the curves E_n for which n is thought to be congruent do in fact have positive rank. See for example the tables of Elkies [12].

As the sign of the functional equation of $L(E_n/\mathbb{Q},s)$ is $+1$ for $n\equiv 1,2,3 \pmod 8$ and -1 for $n\equiv 5,6,7 \pmod 8$ [6] then by the Parity Conjecture (a special case of the Birch and Swinnerton-Dyer Conjecture) we expect that the rank of E_n is even in the $+1$ case and odd in the -1 case. This again is an interesting test of the Birch and Swinnerton-Dyer Conjecture.

1.3 Tunnell's criterion

In 1983 Jerrold Tunnell gave the following criterion:

Theorem (Tunnell) 1.1 *Let n be a squarefree positive integer. Set*

$$f(n) = \#\{(x,y,z)\in\mathbb{Z}^3 | x^2+2y^2+8z^2 = n\},$$
$$g(n) = \#\{(x,y,z)\in\mathbb{Z}^3 | x^2+2y^2+32z^2 = n\},$$
$$h(n) = \#\{(x,y,z)\in\mathbb{Z}^3 | x^2+4y^2+8z^2 = n/2\},$$
$$k(n) = \#\{(x,y,z)\in\mathbb{Z}^3 | x^2+4y^2+32z^2 = n/2\}.$$

For odd n if n is congruent then $f(n) = 2g(n)$. For even n, if n is congruent then $h(n) = 2k(n)$.

Moreover, if the weak Birch and Swinnerton-Dyer conjecture is true for the curve $y^2 = x^3 - n^2x$ then the converse of both implications is true: $f(n) = 2g(n)$ implies n is congrunent when n is odd and $h(n) = 2k(n)$ implies n is congruent when n is even.

Tunnell's criterion can be used to make an effective test for congruent numbers, subject to the BSD conjecture. For example Rogers used it for n up to $N = 2\times 10^7$ [31]. We note that the asymptotic running time for such a computation up to a limit N, when relying on counting representations of n or $n/2$ by binary quadratic forms, is $O(N^{\frac{3}{2}})$. Rubinstein's computations used a similar method.

1.3.1 Relation to modular forms

We explain briefly the connection between the curves E_n and the criterion of Tunnell.

The curve E_n is a quadratic twist of the curve $E : y^2 = x^3 - x$.

Now associated to E is a weight 2 newform $F(z) = \eta(4z)^2\eta(8z)^2 \in S_2^{\text{new}}(\Gamma_0(32))$ such that $L(E,s) = L(F,s)$, where $L(E,s)$ is the Hasse-Weil L-series of the elliptic curve E and $L(F,s)$ is the Mellin transform of the modular form F.

If we write $L(E,s) = \sum b_m m^{-s}$ then $L(E_n,s) = L_F(\chi_D,s) = \sum \chi_D(m) b_m m^{-s}$, where $D = -n$ if $n\equiv 1 \pmod 4$ and $D = -4n$ if $-n \equiv 2,3 \pmod 4$.

The importance of this fact is that the conjecture of Birch and Swinnerton-Dyer (applied to the elliptic curve E_n) then gives a condition on when n can be congruent:

Conjecture (Birch and Swinnerton-Dyer) 1.2 *If E is an elliptic curve defined over $\mathbb{Q}$ then $L(E,1) = 0$ iff E has positive rank.*

The following theorem of Shimura gives a link between modular forms of half integer weight $k/2$ and forms of integer weight $k-1$. The correspondence is called a *Shimura lift*. We are interested in this theorem in the case $k = 3$.

Theorem (Shimura) 1.3 *Let $f(z) = \sum_{m=1}^{\infty} a(m)q^m \in S_{k/2}(4N, \chi)$ be a modular form of weight $k/2$ for $\Gamma_0(4N)$ (actually $\Delta_0(4N)$) with χ a Dirichlet character modulo $4N$ and suppose that $T_p^2(f) = \omega_p f$ for all primes p, where T_p^2 are the Hecke operators. Define $F(z) = \sum_{m=1}^{\infty} A(m)q^m$ where the values $A(m)$ are given by the identity*

$$\sum_{m=1}^{\infty} A(m)m^{-s} = \prod_p (1 - \omega^p p^s + \chi(p)^2 p^{k-2-2s})^{-1}.$$

Then for some integer N_0 divisible by the conductor of χ^2 we have that $F(z) \in M_{k-1}(N_0, \chi^2)$, i.e. $F(z)$ is an integer weight modular form of weight $k-1$.

As mentioned above, we are interested in whether or not the L-series $L(E_n, s)$ vanishes at $s = 1$, this corresponding to E_n having positive rank, assuming the Birch and Swinnerton-Dyer conjecture.

Tunnell made use of a result of Waldspurger to access information about the value of these L-series at $s = 1$. The basic idea behind Waldspurger's Theorem and related results is that if $F(z)$ is the Shimura lift of $f(z)$ as per the previous theorem, then the value of $L(F_n, s)$ at $s = (k-1)/2$ for squarefree n, is proportional to the n-th Fourier coefficient of $f(z)$. In particular if suitable forms $f(z)$ can be identified then it is possible to determine when $L(F_n, s)$ vanishes at the centre of the critical strip, $s = (k-1)/2$.

The following result (which is a reformulation of the theorem of Waldspurger, see [29]) formulates this more precisely.

Theorem (Waldspurger) 1.4 *If $F(z) = \sum_{m=1}^{\infty} a(m)q^m \in S_{k-1}^{new}(\Gamma_0(M))$ and $\delta = \pm 1$ is the sign of the functional equation of $L(F, s)$ then there is a Dirichlet character χ modulo $4N$, a positive integer $M|N$, a nonzero complex number Ω_F and a nonzero Hecke eigenform*

$$f(z) = \sum_{m=1}^{\infty} b_F(m)q^m \in S_{k/2}(\Gamma_0(4N), \chi)$$

such that there are fundamental disciminants n, coprime to $4N$ and with the same sign as δ that lie in arithmetic progressions and for which

$$b_F(n_0)^2 = \varepsilon_n \cdot \frac{L(F_n, (k-1)/2) n_0^{k/2}}{\Omega_F},$$

where ε_n is algebraic and $n_0 = |n|$ if n is odd, otherwise $n_0 = |n|/4$. For all other n with the same sign as δ the Fourier coefficients $b_F(n_0)$ vanish.

By careful examination of the conditions of Waldspurger's Theorem, Tunnell was able to identify modular forms which allowed for identification of the values of n for which $L(E_n, s)$ vanishes at $s = 1$. Even better yet, he was able to write these weight $3/2$ modular forms as the product of explicit theta series.

If we define $f_1(z) = \eta(8z)\eta(16z)$ then

$$f_1(z)\theta_0(2z) = \sum_{m=1}^{\infty} a(m)q^m \in S_{\frac{3}{2}}(\Gamma_0(128)),$$

$$f_1(z)\theta_0(4z) = \sum_{m=1}^{\infty} b(m)q^m \in S_{\frac{3}{2}}(\Gamma_0(128), \chi),$$

where $\chi(r) = \left(\frac{8}{r}\right)$ and $\theta_0(z) = \sum_{m=-\infty}^{\infty} q^{m^2}\exp(2\pi imz)$ is the Jacobi theta function.

Tunnell proved that these were Hecke eigenforms whose Shimura lift was $F(z)$. He then showed that if n is an odd positive squarefree integer then

$$L(E_n, 1) = a(n)^2 \cdot \frac{\Omega}{4\sqrt{n}},$$

$$L(E_{2n}, 1) = b(n)^2 \cdot \frac{\Omega}{2\sqrt{2n}},$$

for a certain real period Ω.

For further information on Tunnell's approach, see Tunnell's original paper [39] and the books by Ono [29] and Koblitz [24].

The above result of Tunnell allows us to determine congruent numbers, subject to the BSD conjecture, simply by checking whether the Fourier coefficients $a(n)$ and $b(n)$ are zero.

Thus the entire problem of determining congruent numbers is reduced to computing the theta series f_1 and θ_0 and performing power series multiplications. We actually use slight modifications of these Θ-functions, which allow us to exploit additional information on arithmetic progressions.

1.4 Our work

In this paper we present methods for performing both parts of the computation efficiently in practice. In particular the computation of theta functions is done in a cache efficient manner, and asymptotically fast truncated polynomial multiplication techniques are used to perform the power series multiplications.

As the computations that we performed were too large to fit into the main memory of the computers used, our method made use of disk based techniques which allowed each phase of the computation to be done in blocks.

2 "Out-of-core" Fast Fourier Transform methods

In order for our computation to complete in a reasonable time the large power series multiplication had to be completed using Fast Fourier Transform (FFT) techniques.

The complex FFT algorithm was essentially known to Gauss in 1805 (see [18]) but developed in its current form by Cooley and Tukey in 1965 [9].

In 1971 Schönhage and Strassen presented two algorithms for multiplication of large integers based on the FFT [33]. One of these methods, where the field of complex numbers is replaced by a finite ring $\mathbb{Z}/p\mathbb{Z}$ containing sufficiently many roots of unity, has become known as the Schönhage-Strassen method. It can multiply two n bit numbers in asymptotic time $O(n\log n\log\log n)$.

The technique of Schönhage and Strassen can be viewed as either a polynomial multiplication or a large integer multiplication technique.

The standard technique for converting a polynomial multiplication to a large integer multiplication is known as Kronecker segmentation, where the polynomials to be multiplied are first evaluated at some power of 2 chosen sufficiently large that the coefficients of the product polynomial can be identified by their binary representation in the output of the large integer multiplication.

Power series multiplication can be effected by simply truncating a full polynomial multiplication of two n term polynomials to length n.

In the literature, FFT computations which are larger than the available memory of the machine (and which necessarily use disk as an extension) are referred to as *out-of-core* FFT methods.

Many of the FFT methods described in the literature have been for the now defunct vector architectures, or for distributed memory systems, including those with tree, mesh or hypercube architectures (see [2], [8], [23], [36] and [38] for examples), where the emphasis has often been on minimising interprocess communication.

In our case, the machine used was a shared memory system and indeed one where the quantity of memory was a limiting factor for the computation, forcing the computation to be done "out-of-core".

The principal issue with using standard FFT algorithms in a hierarchical memory system (including one where disk is one level of the hierarchy) is that at least n complete passes over the dataset must be performed to compute a convolution of length 2^n. As disk access is typically a couple of orders of magnitude slower than memory access this makes standard FFT algorithms prohibitively slow for computations which do not fit in memory.

The first description of a technique to deal with memory hierarchy in this context is the paper of Gentleman and Sande [19]. The method has become known as Bailey's Four Step method (in the context of complex FFT's), see [3]. The basic idea is to break the data into a two dimensional array and perform multiple small FFT's in the horizontal and then in the vertical directions. The second method used in the present computation is essentially a version of Bailey's method adapted for large integer multiplication.

Bailey's method can be extended to a six (or five) step three dimensional method and beyond. See the above cited paper of Bailey's for older references, or [30] for a more recent reference. For application to integer multiplication, see for example [20].

Some other algorithms for out-of-core FFT's include the algorithm of Cormen [10] based on the in-core method of Swarztrauber, the method of Takahashi [37] for the Parallel Disk Model (PDM) of Vitter and Shriver and the parallel FFT method of Vitter and Shriver [40] for a two level memory system.

Apart from Bailey's method there is another commonly used method for out-of-core FFT computations involving exact arithmetic with integers. This is the method of using Number Theoretic Transforms (NTTs) with Chinese Remainder Theorem reconstitution.

A Number Theoretic Transform is an FFT in the ring $R = \mathbb{Z}/p\mathbb{Z}$ for a specially chosen prime p usually called an "FFT prime". Usually p is chosen to fit into a single machine word, i.e. 32 or 64 bits. The value p is chosen so that R has sufficiently many roots of unity to support the chosen length of convolution.

FFT primes p can be chosen to be of the form $p = 2^k m + 1$ for some small value m. Let x be a primitive root modulo p, i.e. a value x such that $x^{p-1} \equiv 1 \pmod{p}$, but such that x^a is not $1 \pmod{p}$ for any value of a dividing $p - 1$. Clearly x is then a $(p-1)$-th root of unity in R and x^m is a 2^k-th root of unity, supporting convolutions of length 2^k.

In order to multiply two large integers A and B out-of-core using NTT's, the integers are first broken down into chunks so that the problem becomes a polynomial multiplication problem $h(x) = f_A(x) \times g_B(x)$. The coefficients of the two polynomials are then reduced modulo a number of FFT primes. Then the Chinese Remainder Theorem can be used to reconstitute the product of the original polynomials from the products modulo the various FFT primes p.

If r FFT primes p are used then the memory usage of each NTT is r times less than doing an FFT directly with the polynomials f_A and g_B.

The first method which we used in our computation is essentially a variant of the NTT method, but with its own set of advantages. We give a complete description in a later section.

The NTT transform method, in combination with Chinese Remaindering has been used frequently in the computation of digits of π. See for example the paper of Bailey, [4] where two FFT primes were used, in that case to avoid the necessity of quad-precision arithmetic in a complex FFT. The same paper also mentions a proposal to use three FFT primes, avoiding even the requirement for double precision arithmetic in the NTT's, but imposing a severe restriction on the length of convolution supported at that time.

More recently Carey Bloodworth's prime digit program used eight NTTs and CRT to compute π in 32 bit mode, [7]. There is also the program of Xavier Gourdon [15] which as of 2004 held the record for the greatest number of digits of π computed on a home computer (it has since been beaten by Steve Pagliarulo's QuickPi program). Gourdon's program uses an unspecified number of NTTs and CRT.

3 Cache efficient computation of theta functions

3.1 Our Θ-functions

Rather than use the modular forms of Tunnell given above, we note that (mentioned to us by N. D. Elkies) we can split the problem(s) up by a factor of two, upon noting that $\eta(8z)\eta(16z)\theta_0(2z)$ and $\eta(8z)\eta(16z)\theta_0(4z)$ can each be split into a sum of two similar products, each of which is supported on (approximately) half as many coefficients.

In particular, we have the Shimura lifts as

$$\eta(q^8)^{-1}\eta(q^{16})^6\eta(q^{32})^{-2} \quad \text{for } n \equiv 1 \pmod 8, \tag{1}$$

$$\eta(q^8)\eta(q^{32})^2 \quad \text{for } n \equiv 3 \pmod 8, \tag{2}$$

$$\eta(q^{16})^{-1}\eta(q^{32})^2\eta(q^{64})^4\eta(q^{128})^{-2} \quad \text{for } n \equiv 2 \pmod{16}, \tag{3}$$

$$\eta(q^{16})^{-1}\eta(q^{32})^4\eta(q^{64})^{-2}\eta(q^{128})^2 \quad \text{for } n \equiv 10 \pmod{16}. \tag{4}$$

As each is a series in q^8 or q^{16}, our complexity reduces by a factor of 8 or 16.

The ability to compute each of these lifts as a multiplicative convolution is obtained from the following expansions:

$$A(q) = \eta(q)^{-2}\eta(q^2)^5\eta(q^4)^{-2} = \sum_n q^{n^2}, \tag{5}$$

$$B(q) = \eta(q)^{-1}\eta(q^2)^2 = q^{1/8}\sum_n q^{n(2n+1)}, \tag{6}$$

$$C(q) = \eta(q) = q^{1/24}\sum_n (-1)^n q^{n(3n+1)/2}, \tag{7}$$

$$D(q) = \eta(q)^2\eta(q^2)^{-1} = \sum_n (-1)^n q^{n^2}, \tag{8}$$

$$E(q) = \eta(q)\eta(q^2)^{-1}\eta(q^4) = q^{1/8}\sum_n (-1)^n q^{n(2n+1)}. \tag{9}$$

We can then obtain the four Shimura lifts of the above respectively as $A(q^8)C(q^8)C(q^{16})$, $C(q^8)C(q^{32})^2$, $B(q^{16})D(q^{64})^2$, and $B(q^{16})E(q^{32})^2$.

One can compute (say) $E(q^{32})^2$ directly – that is, not as a convolution – via iterating over lattice points in 2 dimensions, taking approximately linear time. So we only need one convolution for each of the four cases.

4 The first polynomial method

Our first method can be viewed as a variation of the Number Theoretic Transform (NTT) method mentioned above in that numerous products were computed modulo primes which fit into a single machine word and the result reconstituted using the Chinese Remainder Theorem. However we did not use NTTs.

The main obvious advantage of such an approach for a computation of this size is that a much larger computation over $\mathbb{Z}[x]$ is decomposed into numerous smaller polynomial products over $\mathbb{Z}/p\mathbb{Z}[x]$ for various "primes" p. This means

that the entire computation does not need to fit into memory. Instead memory only has to accomodate one of the products in $\mathbb{Z}/p\mathbb{Z}[x]$. However note that this benefit does not require the use of NTTs per se. Any method for multiplying polynomials over $\mathbb{Z}/p\mathbb{Z}[x]$ will result in the same advantage.

One advantage of using NTTs is that the primes p can be chosen in such a way that reduction modulo p can be performed very efficiently. For example "primes" p of the form $2^K + 1$ can be used, in which case reduction modulo p is reduced to subtractions rather than expensive divisions. Variations on this theme allow for a number of special primes which reduce the cost of reduction modulo p. The main problem with this approach is that not very many primes with special form exist below 64 bits (the size of a machine word). Whilst it may have been possible to perform our 10^{12} computation using this approach, future larger computations are ruled out on account of the lack of special primes.

In the general case, primes of the form $p = 2^k m + 1$ for small values of m can be used, as explained above. Reduction modulo p can still be computed relatively efficiently for primes of this form. There is an implicit limit in that on a 64 bit architecture the length of convolution multiplied by the largest value of m must be less than 2^{64}. In practice this is not such a significant restriction as convolutions of length 10^{12} could be performed with many thousands of primes p of this kind.

For our computation we chose to use general word sized primes p and an alternative method of performing polynomial multiplications over $\mathbb{Z}/p\mathbb{Z}$. The main reason for this choice was the existence of well-tested, high performance packages for doing such computations, such as FLINT [17] and zn_poly [21]. In fact, Victor Shoup's well-tested NTL package [34] was the only library we were aware of with asymptotically fast NTTs. Unfortunately NTL is not threadsafe and numerous recent improvements in polynomial arithmetic are not reflected in NTL, which is no longer under active development. There was also an advantage in having two separate implementations of arithmetic in $\mathbb{Z}/p\mathbb{Z}[x]$ in that comparisons could be made between the two implementations whilst testing. In the final computation zn_poly was used for the multiplications in $\mathbb{Z}/p\mathbb{Z}[x]$.

Another minor consideration when using FFT primes and NTTs is that on 32 bit architectures the largest m times the maximum length of convolution must be less than 2^{32}. This severely restricts the maximum length of convolution, even on 32 bit machines with sufficient memory to hold larger convolutions. Although this can be worked around by making use of two word primes, this comes with a performance deficit and complicates code.

The implementation of multiplication in $\mathbb{Z}/p\mathbb{Z}[x]$ in zn_poly is highly optimised. It offers a cache-efficient, truncated, Schönhage-Nussbaumer convolution [20], which performs significantly better than other implementations for general primes p, which are usually based on Kronecker Segmentation (or methods which are not even asymptotically fast).

Another feature of our implementation is that we use a large number of primes p. For the largest polynomial multiplications, in the 1 (mod 8) and 3 (mod 8) cases, just over 500 primes were used. The main reason for this is due to the parallel (threaded) nature of our implementation. It was convenient to deal with multiple primes at the same time with each thread performing a multiplication modulo a different prime. We worked with 16 threads (the number of

CPU cores on the machine used). All the data for all 16 threads must be in memory simultaneously, so to realise any memory savings through the use of multimodular arithmetic the total number of primes must be significantly larger than this. Another consideration is that each multiplication of polynomials f and g in $\mathbb{Z}/p\mathbb{Z}[x]$ requires a considerable amount of temporary storage. In fact the total memory usage of the zn_poly routine used to multiply f and g is about six times the memory required to store the polynomials f and g themselves.

One disadvantage of using so many primes is that reduction and Chinese Remainder Theorem reconstruction require a significant portion of the run time. The naive approach is to reduce the large coefficients of the polynomials in $\mathbb{Z}[x]$ modulo each of the primes p in turn and to similarly reconstruct one prime at a time. However for m coefficients in $\mathbb{Z}$ of n bits, reconstruction using this approach will take time $O(n^2m)$. This is asymptotically much worse than the time required to do the actual polynomial multiplications over $\mathbb{Z}/p\mathbb{Z}$.

In order to avoid this situation a divide-and-conquer approach is used for the multimodular reduction and recombination phases. The divide-and-conquer approach completes the CRT recombination in time $O(n \log^2 nm)$ ignoring smaller loglog factors. Although this is still asymptotically a log factor greater than the time for the multiplications, the implied constant was small enough in our implementation that the CRT recombination did not dominate the computation time at the scale we were working.

For a straightforward description of the divide-and-conquer approach to the Chinese Remainder algorithm see [41], pages 57–58. Similar preconditioning and a divide-and-conquer approach can of course be applied to the multimodular reduction phase. A slight adjustment needs to be made to both the reduction and CRT phases to cope with a number of primes which is not a power of 2.

An advantage of this multimodular approach to multiplying theta series, which we discuss in more detail below, is that the amount of data written to disk at intermediate stages of the computation can be kept very low, minimising expensive disk read/write operations.

We now describe our algorithm in full. Throughout the algorithm we make use of two sets of disk files, $\mathcal{F} = \{F_i : i = 0, 1, .., FILES - 1\}$ and $\mathcal{G} = \{G_j : j = 0, 1, .., FILES - 1\}$. The number of files, FILES, in each set can be adjusted. In our implementation we used FILES = 500 for the 1 (mod 8) and 3 (mod 8) computations and FILES = 250 for the 2 (mod 16) and 10 (mod 16) computations.

A number of other constants were also set before the computation started. The length of the theta functions being multiplied, LIMIT, and the optimal block size, BLOCK, for the computation of the theta coefficients were set (see the section on cache efficient computation of theta series). We also set a constant BUNDLE which specified how many small theta function coefficients would be bundled together at a time, using Kronecker Segmentation, to make each large coefficient of the intermediate polynomials in $\mathbb{Z}[x]$ that were multiplied using the multimodular technique. Various values of BUNDLE were tried, including 500, 1000, 2000. Throughout the computation THEADS = 16 threads were used.

To simplify the computation, the number of primes, PRIMES, was rounded up to a multiple of 16, i.e. a multiple of the maximum number of threads,

THREADS. The length of the theta functions, LIMIT, ($10^{12}/8$ in the $1, 3 \pmod 8$ cases and $10^{12}/16$ in the $2, 10 \pmod{16}$ cases), was chosen to be a multiple of both the number of files, FILES, times BUNDLE, and of the number of files, FILES, times the theta function block size, BLOCK.

We determined that the largest coefficient of the product of our theta series would comfortably fit into 16 signed bits. Thus in the Kronecker Segmentation phase, zero padded fields of 16 bits were used.

The algorithm is presented in three stages, corresponding to file read/write phases. Conceptually speaking, the first phase bundles groups of coefficients of the first theta function theta_A together using Kronecker Segmentation, to compile a polynomial $f_A \in \mathbb{Z}[x]$ with multiprecision coefficients, then reduces each of the coefficients of f_A modulo each of the PRIMES word sized primes, then the transpose of the matrix of values produced is written to disk. The entire process is multithreaded and numerous files are used so that the entire computation does not have to be done in memory at once. The entire process is repeated for the second theta function theta_B.

Naturally functions theta_A and theta_B are assumed to exist which can supply BLOCK coefficients of θ_A and θ_B at a time from an arbitrary starting point.

Algorithm 1 : Phase 1

PRIMES $\leftarrow$ceil$(2 \times 16 \times$BUNDLE$/62) + 1$
PRIMES $\leftarrow$ ceil(PRIMES$/16) \times 16$
primes[0] $\leftarrow$ nextprime(2^{62})
for $k = 1$ to PRIMES **do**
  primes[k] $\leftarrow$ nextprime(primes[$k-1$])
end for
blocksize $\leftarrow LIMIT/FILES$
for $i = 0$ to $FILES - 1$ **do**
  for $l = 0$ to blocksize/BLOCK **do**
    for $m = 0$ to BLOCK **do**
      theta[$l\times$BLOCK+m] $=$ $\text{theta}_A(i\times\text{blocksize}+l\times\text{BLOCK}+m)$, (using THREADS threads)
    end for
  end for
  for $j = 0$ to blocksize/BUNDLE (using THREADS threads) **do**
    for $r = 0$ to BUNDLE **do**
      a_r =theta[$j\times$BUNDLE+r]
    end for
    $B \leftarrow 2^{16}$
    $c_j \leftarrow a_0 + a_1 B + a_2 B^2 + \cdots + a_{s-1} B^{s-1}$, $s =$ BUNDLE
  end for
  $f_i \leftarrow c_0 + c_1 x + \cdots + c_{t-1} x^{t-1} \in \mathbb{Z}[x]$, $t =$ blocksize/BUNDLE
  for $j = 0$ to t **do**
    for $k = 0$ to PRIMES, (using THREADS threads) **do**
      $M_1[j][k] \leftarrow c_j \pmod{\text{primes}[k]}$
    end for
  end for
  Transpose M_1 and write to file F_i
end for
Repeat above for theta function θ_B, writing transposes of M_2 to files G_i

The second phase of the algorithm reads the data stored in the files F_i and G_j and multiplies the polynomials in $\mathbb{Z}/p\mathbb{Z}$ for each of the PRIMES primes p, truncating the results and storing them back in the files F_i.

Algorithm 1 : Phase 2

for $i = 0$ to PRIMES, (using THREADS threads) **do**
  for $j = 0$ to FILES **do**
    Read block j of $M_1[i]$ from line i of file F_j
    for $k = 0$ to blocksize/BUNDLE **do**
      $a_{k+j*t} \leftarrow M_1[i][k + j * t]$, where $t =$ blocksize/BUNDLE
    end for
  end for
  $fp(x) \leftarrow a_0 + a_1x + \cdots + a_{t-1}x^{t-1}$
  for $j = 0$ to FILES **do**
    Read block j of $M_2[i]$ from line i of file G_j
    for $k = 0$ to blocksize/BUNDLE **do**
      $b_{k+j*t} \leftarrow M_2[i][k + j * t]$, where $t =$ blocksize/BUNDLE
    end for
  end for
  $gp(x) \leftarrow b_0 + b_1x + \cdots + b_{t-1}x^{t-1}$
  $hp(x) = c_0 + c_1x + c_2x^2 \ldots \leftarrow gp(x) \times hp(x)$
  Truncate $hp(x)$ to length blocksize/BUNDLE
  for $j = 0$ to FILES **do**
    for $k = 0$ to blocksize/BUNDLE **do**
      $M_1[i][k + j * t] \leftarrow c_{k+j*t}$, where $t =$ blocksize/BUNDLE
    end for
    Write block j of $M_1[i]$ to line i of file F_j
    Delete file G_j
  end for
end for

In the statement of phase 2 we have not explicitly mentioned the fact that multiplications for THREADS primes were dealt with at the same time in memory. However it is straightforward to modify the algorithm to handle this subtlety.

The final phase of the algorithm reconstitutes the product polynomial $H = f_A \times f_B \in \mathbb{Z}[x]$ using the preconditioned, divide-and-conquer CRT mentioned above, overlaps and adds the coefficients of H to make a large integer (not all stored in memory at once), extracts the theta function product coefficients from bit fields of this integer and counts zeroes and performs other statistical computations on these small product coefficients.

Note that coefficients of f_A and f_B are at most $D = 2 \times 8 \times BUNDLE$ bits in size. Thus coefficients of the product $G = f_A \times f_B$ are at most $2D + l$ bits in size, where $l = \text{ceil}(log_2(\text{length}(f_A)))$. Conceptually we turn G into a large integer by evaluating it at 2^D. As the coefficients of G are at most $2D + l$ bits, at most three coefficients of G are overlapped and added when this evaluation takes place.

Although it is possible to parallelise the reconstruction of the large integer in blocks that fit in memory, we chose not to use multithreaded code for this part of the computation, for the sake of simplicity.

In counting zeroes and computing the statistics in the final phase of the algorithm we first sieved out non-squarefree indices so that we were counting primitive congruent numbers.

Algorithm 1 : Phase 3

Let $t_1 = a_0 + a_1 2^D + a_2 * 2^{2D}$,
Let $t_2 = b_0 + b_1 2^D + b_2 * 2^{2D}$,
Let $t_3 = c_0 + c_1 2^D + c_2 * 2^{2D}$, {w}here a_i, b_i, c_i are field of D bits initialised to 0
Read block 0 of M from file F_0
Transpose M
for $i = 0$ to blocksize/BUNDLE, (using THREADS threads) **do**
  $d_i \leftarrow$ CRT($M[i][0] \pmod{primes[0]}, \ldots, M[i][t-1] \pmod{primes[t-1]}$)
end for
$t_1 \leftarrow d_0$
$v \leftarrow 0$
carry $\leftarrow 0$
$j \leftarrow 0$
while v ¡ LIMIT **do**
  carry, $T \leftarrow a_0 + b_1 + c_2 +$ carry, where T is D bits
  Extract BUNDLE coefficients from T, count zeroes, compute stats
  $t_3 \leftarrow t_2$
  $t_2 \leftarrow t_1$
  $v \leftarrow v$+BUNDLE
  if $v \equiv 0 \pmod{\text{blocksize}}$ **and** $v <$ limit **then**
    $s \leftarrow v$/blocksize
    Read block s of M from file F_s
    Transpose M
    for $i = 0$ to blocksize/BUNDLE, (using THREADS threads) **do**
      $d_i \leftarrow$ CRT($M[i][0] \pmod{primes[0]}, \ldots, M[i][t-1] \pmod{primes[t-1]}$)
    end for
    $j \leftarrow 0$
  end if
  $t_1 \leftarrow d_j$
  $j \leftarrow j + 1$
end while

In order to manage the multithreaded parts of the code, very straightforward OpenMP pragmas were used. OpenMP is now a standard component of the gcc C compiler and version 4.4.1 of the compiler was used to compile the code. To manage the disk access, the mmap kernel service was used. This allows a block of process address space to be mapped directly to a file. The kernel then schedules reading and writing of the files automatically.

Each small coefficient of the final theta product fit into $B = 2$ signed bytes. Thus each coefficient of the polynomial $H \in \mathbb{Z}[x]$ is just over $2 \times B \times$BUNDLE bytes, and so a little over $2 \times B \times$BUNDLE/8 word sized primes must be used on a 64 bit machine, i.e. a little over BUNDLE/2 primes were used in our implementation. As each of the polynomials f_A, f_B had LIMIT/BUNDLE coefficients, the space required to store the multimodular reductions of f_A and f_B on disk is a little over $4 \times$LIMIT bytes per polynomial. As LIMIT was $10^{12}/8$ for the largest of

our polynomials, this meant a little over 0.5Tb were used per polynomial.

The second polynomial used in each case is sparse and this property is retained after reduction modulo the primes. The data for the second polynomial could therefore be compressed to disk with a very high compression ratio, however this was not attempted in our implementation.

In the second phase of the computation the files F_i are overwritten with the result of the multimodular products and no further files are created. Thus in total a little over 0.5Tb were written and read for each of the polynomials f_A and f_B and 0.5Tb were written and read for the polynomial H.

When performing a product using an FFT directly, one must zero pad each of the input coefficients out to the size of the output coefficients and also zero pad the polynomials from the input length up to the output length. In the multimodular method of Algorithm 1, only the first padding is necessary, the data remaining truncated at all stages of the algorithm. This results in approximately half the data being written to disk and read again.

5 The second polynomial method

6 The results and analysis

Each of the congruent number series computations was performed twice, once with each of the two algorithms described in this paper. The number of zeroes found in each case was compared as were the statistics that were collected on the frequency of other values of the product series.

In addition to these checks, in the computation using the first algorithm, the theta series were set up in such a way that the output coefficients were all divisible by 2 or 4 (with the possible exception of a single -1 value). Each value that the theta function could take, from -2^{15} to 2^{15} was counted to see what the frequency of its occurrence was. Thus if any coefficient was off by 1 this would be detected by the code as a nonzero count for a value that was not divisible by 2 or 4.

This check is particularly relevant as we needed to be sure that no overflows had occurred. Indeed if an overflow occurred, the overflowed value would have the wrong sign. Thus an extra borrow would propagate to the next coefficient (or not propagate when it should) as the coefficient extraction was being conducted at the final stage of the algorithm. This would be indicated by a value that was out by 1.

In the computation using the second algorithm, a different check was performed. The computation was done modulo a small prime which acted as a checksum to see that the products had been performed correctly. In both cases numerous zeroes were checked to verify that they represented elliptic curves with nonzero rank. Furthermore one of the authors constructed code which computed the value of the theta function product at any specific index from first principles, so that numerous values could be computed by a completely independent means.

The first computation was done on a 4 $\times$ Quad Core AMD Opteron server running at 2.4GHz. The memory was 128GB of registered ECC memory, capable

of detecting and correcting single bit errors. The disk array consisted of four 500GB drives in Raid 5 arrangement, resulting in about 1.3TB of available space after system and swap partitions were allocated and 500GB was committed to parity checking. All cores were used in the multithreaded parts of the computation, the result being that the computation was essentially I/O bound in most parts of the computation.

Each of the 1 (mod 8) and 3 (mod 8) computations could be performed by the first algorithm in about 30 hours real time, on this machine. Each of the 2 (mod 16) and 10 (mod 16) computations took around 9 hours.

In the table below we present some statistics from the computation, namely the number of zeroes in bins from 0 to 10^{12}. The results are presented per residue class. Note that only primitive, i.e. squarefree, congruent numbers are counted. Each zero is only counted in one bin, e.g. the 10^{10} bin counts all zeroes in $(10^9, 10^{10}]$.

10^9	10^{10}	10^{11}	2×10^{11}	3×10^{11}	4×10^{11}
3801661	21768969	142778019	127475330	115249740	107930081
5×10^{11}	6×10^{11}	7×10^{11}	8×10^{11}	9×10^{11}	10^{12}
102774355	98817294	95656907	93030373	90748990	88803354

Table 1 : Congruent numbers in the 1 (mod 8) class.

10^9	10^{10}	10^{11}	2×10^{11}	3×10^{11}	4×10^{11}
2921535	17019170	112979066	101436853	91949066	86213764
5×10^{11}	6×10^{11}	7×10^{11}	8×10^{11}	9×10^{11}	10^{12}
82196846	79106503	76626341	74546400	72781203	71239101

Table 2 : Congruent numbers in the 3 (mod 8) class.

10^9	10^{10}	10^{11}	2×10^{11}	3×10^{11}	4×10^{11}
2110645	12294626	81759844	73445274	66579936	62455317
5×10^{11}	6×10^{11}	7×10^{11}	8×10^{11}	9×10^{11}	10^{12}
59536672	57282587	55504389	53993974	52728711	51619397

Table 3 : Congruent numbers in the 2 (mod 16) class.

10^9	10^{10}	10^{11}	2×10^{11}	3×10^{11}	4×10^{11}
1842072	10842882	72556705	65378932	59347550	55720114
5×10^{11}	6×10^{11}	7×10^{11}	8×10^{11}	9×10^{11}	10^{12}
53152609	51190025	49599296	48268971	47158661	46159584

Table 4 : Congruent numbers in the 10 (mod 16) class.

7 Future improvements

The implementation of the first polynomial method could be improved in future in a number of ways.

The matrix transpose operations are currently carried out in a very naive way. No attention has been paid to doing this part of the computation in a cache friendly manner.

The second polynomial can be compressed to disk in a manner similar to that described for the second polynomial method above. This would save substantial disk access and time. The first polynomial can also be compressed on disk with a compression factor of roughly 2. Whether this could save time has not been investigated. Although it would save disk space in the first stage of the computation, there is no reason to expect the data written in the second phase to have low entropy.

At times the disk access is quite random. The mmap service does not guarantee reading or writing of the data sequentially. Instead a paging algorithm is used which reads or writes pages on demand. Currently no attempt has been made in the code to ensure that pages are accessed sequentially. In fact, in the final phase of the computation, reading of the data from disk was sped up substantially by performing CRC checksums of the files in order so that the kernel would cache them in memory before they were needed. Other parts of the implementation could be sped up by similarly ensuring that disk access is not random.

It would be interesting to try number theoretic transforms to see if an optimised implementation would perform better than the current zn_poly code for polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$. This would require substantial work, as zn_poly is highly optimised.

Currently the reconstruction phase at the end of the algorithm is not parallelised. As demonstrated in the other implementation, it is possible to parallelise this with appropriate data structures to take care of carries and borrows.

Another important optimisation for the first polynomial method is speeding up of the division code used in the CRT recombination. Efforts are underway to improve the speed of division in the integer library MPIR [27], that was used. Further improvements in the FLINT library, in which the CRT code is implemented, are also possible. One suggestion was to use Montgomery's REDC algorithm instead of making use of division functions.

At present no attempt is made to handle I/O operations in parallel with computation. Instead, the I/O phase is done by single threaded code after the relevant computations are done. By working in parallel, files could be read in advance. It would be more efficient to allocate a single thread for I/O and use only 15 threads for computation instead of 16.

An interesting question is which other theta functions have decompositions such as the ones we used for the congruent number problem. The authors are interested in computing other theta functions using the methods of this paper. Some experiments with L-series of symmetric powers of elliptic curves have been performed by the authors.

Numerous other interesting theta series and modular forms await investigation, e.g. the Mordell curve, or the congruent number-like series of Shin-ichi Yoshida [42], [43].

8 Acknowledgements

Thanks to Mike Rubinstein for challenging us with the problem of computing the congruent number theta function to 10^{12} in 2008.

We thank Noam D. Elkies for pointing out that the theta functions of Tunnell could be decomposed, making the theta products more manageable.

Many thanks to the American Institute of Mathematics for their involement in the workshops at which much of the collaboration occurred. A special thanks to David Farmer, Estelle Basor, Kent Morrison, Sally Koutsoliotas and Brian Conrey of AIMath for their careful preparation of a web page providing details of our computation for the general public.

Thanks to William Stein for allowing us to use the Sage cluster of machines. These machines are funded by a National Science Foundation grant No. DMS-0821725.

Thanks to J. B. Tunnell for pointing out a careless mistake in an earlier draft of this manuscript.

References

[1] R. Alter, T. B. Curtz, K. K, Kubota, *Remarks and results on congruent numbers*, Proc. Third Southeastern Conf. on Combinatorics, Graph Theory and Computing (1972), pp. 27–35.

[2] Francisco Argüello, Margarita Amor, Emilio L. Zapata, *Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication*, Parallel Comput. 22 (1996), no. 9, pp. 1255–1279.

[3] David H. Bailey, *FFTs in external or hierarchical memory*, J. Supercomput. 4 (1990), pp. 23-35.

[4] David H. Bailey, *The computation of π to $29,360,000$ decimal digits using Borweins' quartically convergent algorithm*, Math. Comp. 50 (1988), no. 181, pp. 283–296.

[5] L. Bastien, *Nombres congruents*, Intermédiaire Math., 22 (1915), pp. 231–232.

[6] Bryan J. Birch, Nelson M. Stephens, *The parity of the rank of the Mordell-Weil group*, Topology, 5 (1966), pp. 295–299.

[7] Carey Bloodworth, *Prime digit program*, available at http://www.boo.net/~jasonp/pipage.html (1999).

[8] C. Calvin, *Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication.*, Parallel Comput. 22 (1996), no. 9, pp. 1255–1279.

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

[10] Thomas H. Cormen, *Determining an out-of-core FFT decomposition strategy for parallel disks by dynamic programming*, Algorithms for parallel processing IMA Vol. Math. Appl., 105, Springer, (1999), pp. 307–320.

[11] Leonard E. Dickson, *History of the Theory of Numbers II*, Carnegie Intitute of Washington, (1920), reprinted Chelsea (1966).

[12] Online tables: http://www.math.harvard.edu/~elkies/compnt.html

[13] Keqin Feng, *Non-congruent Numbers, Odd graphs and the B-S-D Conjecture*, Acta Arith., LXXV, 1, (1996), pp. 71–83.

[14] Keqin Feng, Yan Xue, *New series of odd non-congruent numbers*, Science in China Series A: Mathematics, vol. 49, No. 11 (2006), pp. 1642–1654.

[15] Xavier Gourdon, *PiFast prime digit program*, available at http://numbers.computation.free.fr/Constants/PiProgram/pifast.html, (2004).

[16] Richard K. Guy, *Unsolved Problems in Number Theory*. Springer (2004).

[17] William B. Hart, *Fast Library for Number Theory (FLINT)*, http://www.flintlib.org/.

[18] M. T. Heideman, D. H. Johnson, and C. S. Burrus, *Gauss and the history of the fast Fourier transform*, IEEE ASSP Magazine, 1, (4) (1984), pp. 14-21.

[19] W. M. Gentleman, G. Sande, *Fast Fourier Transforms - For Fun and Profit*, AFIPS Proceedings, vol. 29 (1966), pp. 563–578.

[20] David Harvey, *A cache-friendly truncated FFT*, Theor. Comput. Sci. 410 (2009), pp. 2649-2658.

[21] David Harvey, *zn_poly*, http://www.cims.nyu.edu/~harvey/zn_poly/index.html.

[22] Jean Lagrange, *Thèse d'Etat de l'Unicersité de Reims*, (1976).

[23] S. Lennart Johnsson, Michel Jacquemin, Robert L. Krawitz, *Communication efficient multi-processor FFT*, J. Comput. Phys. 102 (1992), no. 2, pp. 381–397.

[24] Neal Koblitz, *Introduction to Elliptic Curves and Modular Forms*, 2nd edition (1993), Springer-Verlag.

[25] G. Kramarz, *All congruent numbers less than 2000*, Math. Annalen, 273 (1986), pp. 337–340.

[26] F. Lemmermeyer, *Some families of non-congruent numbers*, Acta. Arith., 110 (2003), pp. 15–36.

[27] The MPIR Group, *MPIR : Multiple Precision Integers and Rationals*, http://www.mpir.org/

[28] T. D. Noe, *Congruent numbers up to 10000*, http://www.research.att.com/~njas/sequences/b003273.txt

[29] K. Ono, *The web of modularity: Arithmetic of the coefficients of modular forms and q-series*, CBMS Conference Series, 102 (2004), Amer. Math. Soc.

[30] Rami Al Na'mneh, David W. Pan, *Five-step FFT algorithm with reduced computational complexity*, Inform. Process. Lett. 101 (2007), no. 6, pp. 262–267.

[31] Nicholas F. Rogers, *Rank computations for the congruent number elliptic curves*, Eperiment. Math, vol. 9, Issue 4 (2000), pp. 591–594.

[32] Michael O. Rubinstein, *Personal communication*, (2008).

[33] Arnold Schönhage, Volker Strassen, *Schnelle Multiplikation grosser Zahlen*, Computing 7(3-4) (1971), pp. 281–292.

[34] Victor Shoup, *NTL : Number Theory Library*, http://www.shoup.net/ntl/.

[35] Nelson M. Stephens, *Congruence properties of congruent numbers*, Bull. London Math. Soc. 7 (1975), pp. 182–184.

[36] Paul Swarztrauber, *Multiprocessor FFTs. Proceedings of the international conference on vector and parallel computing—issues in applied research and development (Loen, 1986)*, Parallel Comput. 5 (1987), no. 1-2, pp. 197–210.

[37] Daisuke Takahashi, *Calculation of π to 51.5 billion decimal digits on distributed memory parallel processors*, Trans. Inform. Process. Soc. Japan 39 (1998), no. 7, pp. 2074–2083.

[38] Clive Temperton, *Implementation of a prime factor FFT algorithm on CRAY-1*, Parallel Comput. 6 (1988), no. 1, pp. 99–108.

[39] Jerrold B. Tunnell, *A classical diophantine problem and modular forms of weight 3/2*, Invent. Math. 72 (1983) pp. 323–334.

[40] J. S. Vitter, E. A. M. Shriver, *Algorithms for parallel memory. I. Two-level memories*, Algorithmica 12 (1994), no. 2-3, pp. 110–147.

[41] Franz Winkler, *Polynomial Algorithms in Computer Algebra*, Springer, N.Y. (1996).

[42] Shin-ichi Yoshida, *Some variants of the congruent number problem, I*, Kyushu J. Math. 55:2 (2001), pp. 387-404.

[43] Shin-ichi Yoshida, *Some variants of the congruent number problem, II*, Kyushu J. Math. 56:1 (2002), pp. 147-165.