Massively Parallel Computation of Discrete Logarithms *

Daniel M. Gordon†
Kevin S. McCurley‡

Abstract

Numerous cryptosystems have been designed to be secure under the assumption that the computation of discrete logarithms is infeasible. This paper reports on an aggressive attempt to discover the size of fields of characteristic two for which the computation of discrete logarithms is feasible. We discover several things that were previously overlooked in the implementation of Coppersmith's algorithm, some positive, and some negative. As a result of this work we have shown that fields as large as $GF(2^{503})$ can definitely be attacked.

Keywords: Discrete Logarithms, Cryptography.

1 Introduction

The difficulty of computing discrete logarithms was first proposed as the basis of security for cryptographic algorithms in the seminal paper of Diffie and Hellman [4]. The discrete logarithm problem in a finite group is the following: given group elements g and a, find an integer x such that $g^x = a$. We shall write $x = \log_g a$, keeping in mind that $\log_g a$ is only determined modulo the multiplicative order of g. For general information on the discrete logarithm problem and its cryptographic applications, the reader may consult [9] and [11]. In this paper we shall report on some computations done for calculating discrete logarithms in the multiplicative group of a finite field GF(2^n), and the lessons we learned from the computations. The computations that we carried out used a massively parallel implementation of Coppersmith's algorithm [2], combined with a new method of smoothness testing. Coppersmith's algorithm will be described in section 2, and our new method of smoothness testing will be described in section 2.2. The results of our calculations will be presented in section 3.

*This research was supported in part by the U.S. Department of Energy under contract number DE-AC04-76DP00789

†Department of Computer Science, University of Georgia, Athens, GA 30602. This work was begun while visiting Sandia National Laboratories

‡Sandia National Laboratories, Albuquerque, NM 87185

A great deal of effort (and CPU time!) has been expended on the cryptographically relevant problem of factoring integers, but comparatively little effort has gone into implementing discrete logarithm algorithms. The only published reports on computations of discrete logarithms in GF(2^n) are in [1] and [2, 3]. Both papers report on the calculation of discrete logarithms in the field GF(2^{127}).

Odlyzko [11] has carried out an extensive analysis on Coppersmith's algorithm and projected the number of 32-bit operations required to deal with a field of a given size. A similar analysis was made by van Oorschot [13]. Many of their predictions are consistent with our experience, but there were some surprising discoveries that show their analysis to be quite optimistic. We were able to complete most of the computation to compute discrete logarithms for fields of size up to GF(2^{503}), and can probably go at least a little bit further with our existing machines. The major limitation at this point seems to lie as much in the linear algebra as the equation generation, due to the large amount of computation time and storage needed to process equations for a large factor base.

Analyses of the type made by van Oorschot and Oldyzko can be extremely useful to chart the *increase* in difficulty of computing discrete logarithms as the field size increases. It is however almost impossible to get exact operation counts to within anything better than an order of magnitude using such an analysis. Among the reasons for this are:

- if a high-level language is used, then compilers vary widely in their ability to efficiently translate the code into machine instructions.
- even counting 32-bit operations is not enough, since the number of clock cycles may vary widely. On the nCUBE-2 that was used for most of our computation, 32-bit integer instructions take between 2 and 38 machine cycles.
- data cache misses can cost many operations (as many as 10 cycles on the Intel i860).

For these and other reasons, it is impossible to get very accurate estimates from analytic methods alone. The only reliable method is to actually implement the algorithms with careful attention to details, and measure the running time.

In the course of this work, we used a variety of machines for the computations. The parallel machines were all MIMD (multiple instruction, multiple data), and included

- a 1024 processor nCUBE-2, with four megabytes per processor,
- a 64 processor Intel iPSC/860, with 8-32 megabytes per processor,
- the 512-processor Intel Touchstone Delta, with 16 megabytes per processor.

We started out with the intention of using a Thinking Machines CM-2, but for technical reasons associated with the SIMD hardware and the system software, we found this to be uncompetitive. It also had the disadvantage that it required using a language specific to the machine, whereas the other machines could all accept standard C, with a few minor changes to accomodate differences in message passing syntax.

2 Coppersmith's algorithm

Coppersmith's algorithm belongs to a class of algorithms that are usually referred to as index calculus methods, and has three stages. In the first stage, we collect a system of linear equations (called relations) that are satisfied by the discrete logarithms of certain group elements belonging to a set called a factor base. In our case, the equations are really congruences modulo the order of the group, or modulo $2^n - 1$. In the second stage, we solve the set of equations to determine the discrete logarithms of the elements of our factor base. In the third stage, we compute any desired logarithm from our precomputed library of logarithms for the factor base.

For the Coppersmith algorithm, it is convenient that we construct our finite field $\mathrm{GF}(2^n)$ as $\mathrm{GF}(2)[x]/(f(x))$, where f is an irreducible polynomial of the form $x^n + f_1(x)$, with f_1 of small degree. Heuristic arguments suggest that this should be possible, and a search that we made confirms this, since it is possible to find an f_1 of degree at most 11 for all n up to 600, and it it is usually possible to find one of degree at most 7. For the construction of fields, it is also convenient to choose f so that the element $x \pmod{f(x)}$ is primitive, i.e. of multiplicative order $2^n - 1$. As we shall explain later, there are other factors to be considered in the choice of f_1.

For a given polynomial f that describes the field, there is an obvious projection from elements of the field to the set of polynomials over $\mathrm{GF}(2)$ of degree at most n. In our case, we shall take as our factor base the set of field elements that correspond to the irreducible polynomials of degree at most B for some integer B to be determined later. Call a polynomial B*−smooth* if all its irreducible factors have degrees not exceeding B. Let m be the cardinality of the factor base, and write g_i for an element of the factor base. We note that an equation of the form

$$\prod_{i=1}^{m} g_i^{e_i} \equiv x^t \pmod{f(x)}$$

implies a linear relationship of the form

$$\sum_{i=1}^{m} e_i \log_x g_i \equiv t \pmod{2^n - 1}.$$

In order to describe the first stage in the Coppersmith method, we shall require further notation. Let r be an integer, and define $h = \lfloor n2^{-r} \rfloor + 1$. To generate a relation, we first choose random relatively prime polynomials $u_1(x)$ and $u_2(x)$ of degrees at most d_1 and d_2, respectively. We then set $w_1(x) = u_1(x)x^h + u_2(x)$ and

$$w_2(x) = w_1(x)^{2^r} \pmod{f(x)}. \tag{1}$$

It follows from our special choice of $f(x)$ that we can take

$$w_2(x) = u_1(x^{2^r})x^{h2^r - n} f_1(x) + u_2(x^{2^r}), \tag{2}$$

so that $\deg(w_2) \leq \max(2^r d_1 + h2^r - n + \deg(f_1), 2^r d_2)$. If we choose d_1, d_2, and 2^r to be of order $n^{1/3}$, then the degrees of w_1 and w_2 will be of order $n^{2/3}$. If they behave as random polynomials of that degree (as we might expect), then there is a good chance that they will be B-smooth. If so, then from (1) we obtain a linear equation involving the logarithms of polynomials of degree $\leq B$.

An asymptotic analysis of the algorithm suggests that it is possible to choose the parameters so that the asymptotic running time of the first stage of the algorithm is of the form in such a way that the expected running time to complete stage one is of the form

$$\exp((c_2 + o(1))n^{1/3} \log^{2/3} n), \quad \text{where } c_2 < 1.405.$$

The system of equations generated by the first phase is relatively sparse, and there exist algorithms to solve the system that have an asymptotic running time of $O(m^{2+\epsilon})$ (see section 2.4). If such algorithms are used, then the asymptotic running time of the algorithm turns out to be the same as the first phase.

An analysis of the running time for the third stage (which we do not describe in detail here) suggest a running time of

$$\exp((c_3 + o(1))n^{1/3} \log^{2/3} n),$$

where $c_3 < 1.098$, so it takes less time than the first two stages.

The preceding statements pertain to the asymptotic running time, but give only a rough estimate of the time required in practice for actual cases.

2.1 Refinements of Stage 1.

Odlyzko has suggested several ways to speed up the performance of stage 1. None of these affect the asymptotic running time, but each of them may have some practical significance by speeding up the implementation by a factor of two or three. We shall not discuss these methods in great detail, but merely report on some of them.

Forcing a Factor Into w_1 and w_2 One method that was suggested by Odlyzko for improving the probability that w_1 and w_2 were smooth was by forcing them to contain at least one small degree factor. The method is described in complete detail in [11] and [13], but roughly speaking we fix polynomials v_1 and v_2 of degree at most B, and consider those (u_1, u_2) pairs for which w_1 and w_2 are divisible by v_1 and v_2 respectively. The (u_1, u_2) pairs with this property are described by a rather small set of linear equations modulo 2, and we can easily find such pairs by Gaussian elimination. For the size fields that we considered, the linear systems had fewer than 50 rows and equations, and a special purpose routine to solve these systems proved to be extremely efficient (rows could be added together by using two xor operations on 32-bit integers). One problem with this method is different v_1, v_2 pairs can lead to the same u_1, u_2 pairs, making it rather difficult to avoid duplication of effort. As far as we can tell, we were the first to implement this method, and our experience with it seemed to agree with the predictions made by Odlyzko.

Large Prime Variation One well known method for speeding up the generation of equations is to also use equations that involve one irreducible polynomial of degree only slightly larger than B. The rationale for this is that these equations can be discovered essentially for free, and two such equations involving the same "large prime" can be combined to produce an equation involving only the irreducibles of degree at most B. Many such equations can be discovered by checking whether after removing the smooth part from a polynomial, the residual factor has small degree. After combining two such equations, the equations produced are on average twice as dense as the other equations, so they complicate the linear algebra in stage 2. Many of these equations can however be generated more or less for free, so we chose to use them in the calculations.

Double Large Prime Variation Just as we can use equations involving only a single irreducible of degree slightly larger than B, we can also use equations having two "large prime" factors. This has been used to speed up the quadratic sieve integer factoring algorithm [8], and we might expect the same sort of benefit when it is applied to the Coppersmith algorithm. Many such equations can be produced from reporting those u_1, u_2 pairs that produced a w_1 and w_2 both of which contained a large prime factor.

Smoothness Testing The most time-consuming part of the Coppersmith algorithm is the testing of polynomials for smoothness. At least two methods have been suggested for doing this, both of which are outlined in [11]. Of the two methods, we found the one used by Coppersmith to work faster for our implementation, and this was initially what we used. For this method, a polynomial $w(x)$ is tested for m-smoothness by computing

$$w'(x) \prod_{i=\lceil m/2 \rceil}^{m} (x^{2^i} + x) \pmod{w(x)}. \tag{3}$$

A faster method, using a polynomial sieve, will be outlined in Section 2.2.

Early Abort Strategy One strategy that has been suggested for locating smooth integers is to search through random integers, initially dividing by small primes. At a certain point, we then check to see if the residual factor has moderate size, and abort the testing if it fails. It so happens that a random integer is more likely to be B-smooth from having many very small prime factors than it is from having just a few factors near B, and it follows that we should not spend a lot of time dividing by moderately large primes to test for smoothness. This strategy has come to be known as the "early abort" strategy, and the same heuristic reasoning carries over to the smoothness testing part of Coppersmith's algorithm. Odlyzko predicted that this may result in a speedup of a factor of two in the algorithm, but we never got around to implementing it. The major reason for this is that there seems to be no obvious way to combine this idea with sieving, and the latter gave a somewhat better speedup.

2.2 A Polynomial Sieve

Our first implementation of Coppersmith's algorithm used methods suggested previously by Odlyzko and Coppersmith to test polynomials for smoothness. After having carried out the computation for the case $n = 313$, we looked around for any variations that would speed up the smoothness testing. Drawing on the knowledge that sieving can be exploited to great advantage in integer factoring algorithms, we sought a way to use sieving to test many polynomials simultaneously for smoothness. Sieving over the integers is relatively efficient due to the fact that integers that belong to a fixed residue class modulo a prime lie a fixed distance apart, and it is very easy to increment a counter by this quantity and perform a calculation on some memory location corresponding to the set element.

For polynomials, the problem is slightly different, since we saw no obvious way of representing polynomials in such a way that representatives of a given residue class are a fixed distance apart. It turns out that this is not a great deterrent, since what is important is the ability to quickly move through the representatives, and for the data structures that we used, this can be done using the notion of a Gray code.

Polynomials over GF(2) of degree less than d can be thought of as the vertices of a d-dimensional hypercube, with the coefficient of x^i in a polynomial corresponding to the ith coordinate of a vertex. A Gray code gives a natural way to efficiently step through all such polynomials. The same applies to all polynomials that are divisible by a fixed polynomial g.

Let $G_1, G_2, \ldots, G_{2^d}$ be the standard binary reflected Gray code of dimension d. For any positive integer x, let $l(x)$ be the low-order bit of x, i.e. the integer i such that $2^i \parallel x$. Then we have (see, for example, [10]):

PROPOSITION 1. *The bit that differs in G_x and G_{x+1} is $l(x)$.*

This allows us to efficiently step through the Gray code. Let $s[0], \ldots, s[2^t - 1]$ be 8-bit memory locations corresponding to the u_2 of degree less than t in the obvious way (mapping $u_2(x)$ to $u_2(2)$). Figure 1 describes an algorithm which takes u_1, and finds all u_2 of degree less than t such that $w_1 = u_1x^h + u_2$ is B-smooth.

Note that the inner loop consists of only two 32-bit operations, a shift to multiply g by x^i, and an exclusive-or to add gx^i to u_2, and one 8-bit add.

The actual implementation has a few additions. It checks for large primes, by reporting any pair for which $s[u_2] \geq (\text{degree}(w_1) + h - LP)$, where LP is the maximum degree of a large prime. A sieve by powers of irreducibles up to degree B is also done. Instead of calculating $u_1x^h \bmod g$ each time to start sieving, $x^h \bmod g$ is saved for each g. Then to step from one u_1 to another, we only have to add a shift of $x^h \bmod g$ to the starting sieve location.

A sieve over polynomials w_2 would work similarly; the main difference is that initializing u_2 requires taking a fourth root, which slows things down. It turned out to be more efficient to test smoothness of each w_2 corresponding to a smooth w_1 individually, since only a small number of pairs u_1, u_2 survive the w_1 sieve (w_1 has much higher degree than w_2).

One reason that sieving works so well for the quadratic sieve algorithm is that it replaces multiple precision integer calculations with simple addition operations. We gain the same sort of advantage in Coppersmith's algorithm, by eliminating the need for many modular

```
for i = 0 to 2^t − 1
        s[i] ← 0                                  /* initialize sieve locations */
for d = 1 to B
        dim ← max(t − d, 0)                       /* dimension of Gray code */
        for each irreducible g of degree d
                u2 ← u1 x^h mod g
                if degree(u2) < t then
                for i = 1 to 2^dim
                        s[u2] ← s[u2] + d
                        u2 ← u2 + g x^l(i)        /* u2 = u1 x^h mod g + g G_i */
for i = 0 to 2^t − 1
        if s[i] ≥ (degree(u1) + h − B) then print u1, u2
```

Figure 1: Pseudocode for sieve algorithm

multiplications involving polynomials. The actual operation counts for sieving come out rather close to the operation counts given in [11] and [13], but in the case of sieving the operations are somewhat simpler, and the speedup is substantial.

The number of 32-bit operations to sieve a range of u_1, u_2 pairs is proportional to $\log B$ times the size of the range. This is because there are about $2^d/d$ irreducible polynomials of degree d, so the number of steps to sieve a range of l pairs is:

$$\sum_{d=1}^{B} \sum_{\substack{g \text{ irreducible} \\ \deg(g)=d}} \left(c + \frac{l}{2^d}\right) \approx \sum_{d=1}^{B} \left(c + \frac{l}{2^d}\right) \frac{2^d}{d} \approx l \log B + \frac{c 2^{B+1}}{B},$$

where c represents the startup time for each irreducible. Each of these steps uses a fixed number of 32-bit operations (typically between 2 and 12, depending on the machine, compiler, and source code used). If l is sufficiently large, then the c operations performed for each irreducible become inconsequential. The time spent on finding the initial locations for sieving by each polynomial in the factor base can be made inconsequential by amortizing it over several sieving runs.

In comparison, the number of 32-bit operations needed to test a polynomial for smoothness using Coppersmith's method is at least $3Bh^2/32$ (see [13]), where $h = \lfloor n2^{-r} \rfloor + 1$ is the approximate degree of w_1. As n (and therefore B and h as well) become large, the advantage of using a polynomial sieve becomes overwhelming.

Note that the memory access patterns for the array $s[\cdot]$ in the sieving algorithm are somewhat chaotic, since the indices of consecutive values for u_2 are widely and irregularly dispersed. For processors such as the Intel i860 whose performance is heavily dependent on using memory caches, this severely limits the performance improvement gained from sieving. By contrast, the nCUBE processor is not so dependent on memory access patterns, and the improvement from sieving was more pronounced.

2.3 The choice of f_1

Once we were quite sure that our sieving code was giving completely reliable results, we were unpleasantly surprised that the number of relations discovered was not in agreement with the heuristic arguments given in [11] and [13], but was instead considerably smaller. This led us to reconsider the arguments there, in an attempt to produce more accurate predictions on the number of equations produced by examining a certain range of u_1 and u_2.

The assumption made in both [11] and [13] that w_1 and w_2 are smooth as often as a random polynomial of the same degree is not quite accurate. We shall provide several justifications for this statement, based on heuristic arguments showing ways that w_1 and w_2 (particularly w_2) deviate from behaviour of random polynomials. We have been unable to combine all of the effects we know of into an analytical method for accurately predicting these probabilities. Luckily, it is relatively simple to make random trials to estimate the actual probabilities.

For the cases that we shall be most interested in, w_2 has the form

$$x^T u_1(x)^4 f_1(x) + u_2(x)^4 \tag{4}$$

where $T = 4h - n$ is 1 or 3, and $\gcd(u_1, u_2) = 1$. In the following discussion, g will be an irreducible polynomial of degree d.

First, note that if $g \mid u_1$, then $g \nmid u_2$, and therefore $g \nmid w_1$ and $g \nmid w_2$. Hence if $g \mid w_1$ or $g \mid w_2$, then $g \nmid u_1$. It follows that if if $g^e \mid w_2$ for some integer e, then

$$x^T f_1(x) \equiv (u_1^{-1} u_2)^4 \pmod{g^e}. \tag{5}$$

Note that if $e \geq 2$ and $de > (T + \deg(f_1))$, then (5) is clearly impossible, since the right side reduces to a polynomial with only even exponents modulo g^2, whereas the left side will have odd powers since T is odd and $f_1(0) = 1$. Hence if $d \geq (T + deg(f_1))/2$, it follows that g^2 cannot divide w_2. This shows that w_2 is much more likely to be squarefree than a random polynomial, and therefore somewhat less likely to be smooth.

Another example of nonrandom behaviour from w_2 can be seen from examining the expected value of the degree of the power of an irreducible that divides w_2, compared to the expected power that divides a random polynomial. One can easily show that in some sense, a truly random polynomial will be divisible by an irreducible factor g to the e'th power with probability $1/2^{de}$, and will be exactly divisible by the e'th power with probability $(2^d - 1)/2^{d(e+1)}$. Hence the expected value of the degree of the power of g that divides a random polynomial is $d/(2^d - 1)$.

f_1	factorization	probability
$x^8+x^5+x^4+x^2+x+1$	$(1+x)^2(1+x+x^3+x^4+x^6)$	0.002468
$x^8+x^7+x^5+x^2+x+1$	$(1+x)(1+x^2+x^3+x^4+x^7)$	0.002366
$x^9+x^8+x^5+1$	$(1+x)^4(1+x+x^2)(1+x+x^3)$	0.002607
$x^{10}+x^7+x^6+x^3+x^2+1$	$(1+x)^2(1+x^3+x^5+x^6+x^8)$	0.001956
$x^{10}+x^9+x^8+x^2+x+1$	$(1+x)^8(1+x+x^2)$	0.002383

Table 1: Empirical probabilities that a (u_1, u_2) pair will produce a smooth w_2, for $n = 593$ and different choices of f_1. Tests based on examination of over five million random relatively prime pairs (u_1, u_2) of degrees 22 and 24, respectively.

The expected contribution to a polynomial w_2 is somewhat different. For the case where $g \nmid x^T f_1(x)$, an easy counting argument on residue classes modulo g shows that the probability that g divides w_2 is $(2^d - 1)/(2^{2d} - 1) = 1/(2^d + 1)$, so that the expected degree of the power of g dividing w_2 is $d/(2^d + 1)$, somewhat smaller than for a random polynomial. If $g^e \mid x^T f_1(x)$ for some integer $e \leq 4$, then g^e is automatically guaranteed to divide w_2 whenever $g \mid u_2$. If e is large for a small degree g, then this helps w_2 to be smooth, but if $e = 1$, then it makes w_2 less likely to be smooth.

A complete analysis of this situation is probably not worth the effort. In this paper, it suffices to illustrate the effects by considering the example of $n = 593$. The only f_1's of degree up to 10 for which $x^{593} + f_1$ is irreducible are in Table 1. Clearly the first two f_1's in the table have an advantage from having the smallest degrees, but the third and fifth have an advantage from the large power of $1 + x$ that divides them. The tradeoffs between these effects are not at all clear, but the results of the experiments show that the third f_1 gives a slight advantage, in spite of its larger degree. For the case of $n = 503$, it turned out that $f_1 = x^3 + 1$ was the best choice.

2.4 Linear Algebra

The solution of sparse linear systems over finite fields have received much less attention than the corresponding problem of solving sparse linear systems over the field of real numbers. The fundamental difference between these two problems is that issues involving numerical stability problems arising from finite precision arithmetic do not arise when working over a finite field. The only pivoting that is required is to avoid division by zero. Algorithms for the solution of sparse linear systems over finite fields include:

- standard Gaussian elimination.
- structured Gaussian elimination.
- Wiedemann's algorithm.

n	sparse matrix			dense matrix		reduction
	equations	unknowns	nonzeros	size	nonzeros	
313	108736	58636	1615469	9195	633987	84%
401	117164	58636	2068707	16139	1203414	72%
503	434197	210871	10828595	78394	6394049	63%

Table 2: Results of structured Gaussian elimination for various n.

- Conjugate Gradient.
- Lanczos methods.

A description of these methods can be found in the paper by LaMacchia and Odlyzko [7], where they describe their experience in solving systems that arise from integer factoring algorithms and the computation of discrete logarithms over fields $GF(p)$ for a prime p. We chose to implement three of these algorithms: conjugate gradient, Wiedemann, and structured Gaussian elimination. For handling multiple precision integers we used the Lenstra-Manasse package. The original systems were reduced in size using the structured Gaussian elimination algorithm, after which the conjugate gradient or Wiedemann algorithm was applied to solve the smaller (and still fairly sparse) system.

This approach was used by LaMacchia and Odlyzko in [7] with great success. The structured Gaussian elimination reduced their systems by as much as 95%, leaving a small system that could easily be solved on a single processor. We were not as successful, due to a feature of the equations that Coppersmith's method produces. For the equations in [7], almost all the coefficients are ± 1, and so during the Gaussian elimination most operations involve adding or subtracting one row from another. For our systems, half of the coefficients are multiples of 4, and so it is often necessary to multiply a row by ± 4 before adding it to another. This caused the coefficients in the dense part of the matrix to grow rapidly.

This presented a dilemma. If the matrix coefficients are allowed to become large integers, then the arithmetic operations take considerably more time (and require considerable more complicated code). The alternative is to restrict which rows can be added to others, to keep the coefficients down to 32 bits. This results in a larger matrix, which also slows down stage 2. We elected to deal with the larger matrices. Table 2 gives results for partial gaussian elimination on several systems.

For the 127, 227, and 313 systems, we were able to solve the systems on a workstation (the last one took approximately ten days). The other systems were clearly too large to be solved on a single processor workstation, and the algorithm requires too much communication to effectively run on a network of workstations. We therefore wrote a parallel version (MIMD) of the conjugate gradient code. A single source program was written in C that would compile for Suns, the Intel iPSC/860, the Intel Delta Touchstone, and the nCUBE-2.

Parallelization of the algorithm was accomplished by distributing the matrix rows and columns across the processors. A matrix-vector multiply is then done by multiplying the rows held by the processor times the entire vector. After this operation, each processor communicates to every other processor (in a logarithmic manner) its contribution to the vector result. The distribution of the matrix rows was done by simply assigning the same number of rows to each processor. The structure of the matrix is such that each processor then gets essentially the same number of nonzero entries. For the distribution of the columns, this is certainly *not* the case, as the first few columns contain far more nonzeros than the last few columns. The columns of the matrix were then permuted in order to approximately balance the number of nonzeros assigned to each processor, and some processors ended up getting far more columns. This creates a slight imbalance in the communication phase, but is better than an imbalance in the computation phase.

Unfortunately, this approach suffered from a severe problem when scaled to a large number of processors, since the first column of the reduced 503 matrix contained 61166 nonzero entries, but a perfect load balance on 1024 processors would place $6394049/1024 \approx 6244$ nonzeros on each processor. Proper load balancing of the matrix multiplication would therefore have required that we divide columns between processors, and we were reluctant to modify the code for this due to the added complexity.

Instead, we chose to implement the Wiedemann method. This had the advantage that it required only multiplications of the coefficient matrix times a vector, not the multiplication of the transpose of the matrix. Once again, however, we discovered that there were scaling problems in moving to a large number of processors, since the amount of communication required for sharing results at the end of the distributed matrix-vector multiply increased at least with the logarithm of the number of processors, whereas the amount of computation decreases linearly with the number of processors. Hence when this code was run on 1024 processors of the nCUBE, it ran only slightly faster than it would run on 512 processors. For more dense matrices, the speedup would be larger, but so would the total runtime. This problem was even worse on the 512 processor Delta, where the bisection bandwidth of the machine is about 16% of that of the nCUBE, but the peak processor speed is about 10 times faster.

The communication that we used in each matrix-vector multiplication is often called an all-to-all broadcast, or global concatenation. For machines such as the nCUBE-2 and iPSC/860 that use a hypercube topology for their communications network, there is a fairly obvious algorithm for accomplishing the all-to-all broadcast in $\log(p)$ phases on p processors, passing a minimal amount of information, with no contention for communication channels. The Intel Delta Touchstone uses instead a 16×32 two-dimensional mesh topology. When we first ported the code from the iPSC/860 to the Delta, we were using an Intel-supplied library routine for the communication, but we found that the performance of the Intel routine was far from optimal on the Delta, and the result was that the Delta showed almost no speedup in moving to more processors. Subsequent to this, the second author worked with David Greenberg to develop code and algorithms that improved the performance of the all-to-all broadcast library routine (`gcolx()`) by a factor of 21. This work is reported in [5].

The Wiedemann algorithm requires the use of the Berlekamp-Massey algorithm for computing the minimal polynomial of the matrix. In contrast to the matrix-vector multiplications, this turned out to be quite easy to parallelize, since the core operations required are polynomial additions that are easily parallelized. The only difficulty arises from the fact that the degree steadily increases through the computation, requiring continual load balancing. Eventually the degree of the polynomials becomes large enough that this communication becomes insignificant, and all communication is between nearest-neighbor processors in the network topology, giving very good scalability to large parallel machines. In practice, the Berlekamp-Massey algorithm turned out to consume much less time than the matrix-vector multiplications.

To summarize, after we had invested a substantial amount of time in writing code for the various algorithms, we became aware that communication would be a severely limiting factor in the use of distributed memory parallel machines for solving the linear systems. Since then we have learned of other methods [6],[12] that might dramatically improve the performance. We believe that there remains substantial room for improvement in this area, using these and other ideas.

3 Results

We have completed the precomputation step required to compute discrete logarithms for the fields GF(2^n) for $n = 227$, $n = 313$, and $n = 401$. Once this step has been completed, individual logarithms can be found comparatively easily. We have not bothered to implement the third phase yet, as we expect the running time for this to be substantially less than the first two phases.

The code for producing equations has gone through many revisions and removal of bugs. As a result, we ended up using much more computer time for producing the equations for 401 and 503 than would be required with our current version of the code. Moreover, most of our computations were carried out on the nCUBE-2, which has no queueing of jobs, and no priority system. We therefore wrote our own queueing system, and wrote some code for other users to kill our jobs. This extremely crude approach allowed us to aggressively consume computer time while at the same time allow other users to carry on their normal development activities. The unfortunate result is that many ranges of u_1, u_2 pairs were only partially completed before they were killed, so that very accurate statistics on the completed ranges are difficult to keep. After running the code for 503 for several months, we decided to go back and redo 401 with more care, to keep more accurate records and make an accurate measurement of the amount of calculation required.

For the case of $GF(2^{401})$, we chose to search through all u_1 of degree up to 20, and all u_2 of degree up to 22. The nCUBE-2 was able to process approximately 1.5×10^8 u_1, u_2 pairs per hour on a single processor. Using the full 1024 processors of our nCUBE-2, we could therefore carry out this calculation in approximately 111 hours, or just under 5 days. For comparison, a Sparcstation 2 is able to process approximately 6×10^8 u_1, u_2 pairs per hour, so a single Sun workstation would take approximately 19,000 days (or more realistically, 500 workstations would take just over a month).

Searching this range of u_1, u_2 pairs produced a total of 117,164 equations from a factor base of 58,636 polynomials (all irreducibles of degree up to 19). It also produced approximately 700,000 equations each of which involved only one "large prime" polynomial of degree 20 or 21, which we ended up ignoring due to previously mentioned difficulties with solving the linear system. Clearly there is a tradeoff to be made between producing more equations with a longer sieving phase, or spending more time on solving a harder system of equations. Since the sieving can be carried out in a trivially parallel manner, we opted to spend more time on this rather than claim the whole machine for a long dedicated period to solve a larger system of equations.

For the case of $n = 503$, we attempted to search all u_1 of degree up to 22 and all u_2 of degree up to 25 (again, some of this range was missed by killed jobs, but the percentage should be small). This range produced 165,260 equations over the factor base of 210,871 polynomials of degree up to 21. Combining pairs of equations involving a single irreducible of degree 22 or 23 brought the total up to 361,246 equations. We estimate that repeating this calculation would take approximately 44 days on the full 1024-processor nCUBE. In practice it took us several months due to the fact that we were trying to use idle time, and we never used the full machine. We later extended this calculation to produce a total of 434,197 equations, by running over some u_1 polynomials of degree 23.

The parallel conjugate-gradient code was able to solve the system of equations for $n = 313$ in 8.3 hours on 16 processors of a 64-processor Intel iPSC/860. The equations for $n = 401$ took approximately 33 hours on 32 processors.

Note that $2^{503} - 1$ factors as

$$\begin{aligned} 2^{503} - 1 &= 3213684984979279 \cdot 12158987054135300783 \\ &\quad \cdot 18730306650610808942263 \cdot p_4 \\ &= p_1 \cdot p_2 \cdot p_3 \cdot p_4, \end{aligned}$$

where p_4 is a prime of 96 decimal digits. Solution of the system modulo $2^{503} - 1$ can thus be accomplished by solving four separate systems modulo these prime factors, and combined afterwards using the chinese remainder theorem. The only truly hard part is solving the system modulo p_4, since the individual operations are much slower and the amount of data to be communicated is also larger.

Our original projections for the solution of the 503 equations were too optimistic, since we underestimated the cost of communication. We have still not completed the solution of the 503 equations, but have now at least made timings of individual iterations to estimate the amount of time required. Timings that we have made on the Delta Touchstone and nCUBE-2 show that solution of the system modulo p_1 using the Wiedemann algorithm would take approximately 106 hours on 256 processors of the nCUBE for the matrix multiplications, and 38.4 hours on 512 processors of the Delta. The Berlekamp-Massey calculation would require less than two hours on each of these. For the prime p_4, we are unable to run the matrix-vector multiplications on the nCUBE with our current code due to memory limitations, but the time for matrix multiplications on the Delta is estimated at approximately 105 hours. Logistics have simply prevented us from reserving enough time on the machine to solve the equations in a single run (after all, the purpose of our project was to investigate the effectiveness of massively parallel computers and better algorithms, not to do real cryptanalysis).

4 Conclusion

We started out by repeating Coppersmith's calculation of discrete logarithms for GF(2^{127}). Our original goal was to determine whether it was possible to compute discrete logarithms for the field GF(2^{593}), which has been suggested for possible use in at least one existing cryptosystem. Odlyzko predicted that fields of size up to 521 should be tractable using the fastest computers available within a few years (exact predictions are difficult to make without actually carrying out an implementation). van Oorschot predicted that computing discrete logarithms in GF(2^{401}) should be about as difficult as factoring 100 digit numbers. Both predictions turned out to be reasonable.

We believe that 521 should now be possible to complete, albeit with the consumption of massive amounts of computing time. Discrete logarithms in GF(2^{593}) still seem to be out of reach. Sandia National Laboratories is scheduled to take delivery of an Intel Paragon machine in July 1993 whose peak speed is approximately 50 times the speed of the nCUBE-2 used for this work. Massively parallel machines are expected to be built in the next five years that will reach peak performance levels approximately 500 times faster than the 1024 processor nCUBE-2 that was our primary machine. Unfortunately, this peak speed will be harder to attain in future architectures, so the actual increase in speed for a given application is difficult to project. With a concerted effort on one of these faster machines, or further algorithmic improvements, computing discrete logarithms in GF(2^{593}) might be possible within the next 5-10 years. It would require a much larger factor base (we estimate at least the irreducibles up to degree 23, or 766150 polynomials). It would also be a computation of enormous proportions, and is not likely to be completed in the near future without further innovations.

Acknowledgment The authors wish to thank A.M. Odlyzko, Bruce Hendrickson, and Peter Montgomery for helpful comments in the course of this research.

References

[1] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone. Computing logarithms in fields of characteristic two. *SIAM Journal of Algebraic and Discrete Methods*, 5:276–285, 1984.

[2] D. Coppersmith. Fast evaluation of discrete logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30:587–594, 1984.

[3] D. Coppersmith and J. H. Davenport. An application of factoring. *Journal of Symbolic Computation*, 1:241–243, 1985.

[4] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:472–492, 1976.

[5] David S. Greenberg and Kevin S. McCurley. Bringing theory to practice: The reality of interprocessor communication. unpublished manuscript, 1993.

[6] B. Hendrickson, Robert Leland, and Steve Plimpton. An efficient parallel algorithm for matrix-vector multiplication. Technical Report SAND92-2765, Sandia National Laboratories, 1992.

[7] B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. In *Advances in Cryptology - Proceedings of Crypto '90*, volume 537 of *Lecture Notes in Computer Science*, pages 109–133, New York, 1991. Springer-Verlag.

[8] A. K. Lenstra and Mark Manasse. Factoring with two large primes. In *Advances in Cryptology - Proceedings of Eurocrypt '90*, volume 473 of *Lecture Notes in Computer Science*, pages 72–82, New York, 1991. Springer-Verlag.

[9] Kevin S. McCurley. *The Discrete Logarithm Problem*, volume 42 of *Proceedings of Symposia in Applied Mathematics*, pages 49–74. American Mathematical Society, Providence, 1990.

[10] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms.* Academic Press, New York, second edition, 1978.

[11] A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology (Proceedings of Eurocrypt 84)*, number 209 in Lecture Notes in Computer Science, pages 224–314, Berlin, 1985. Springer-Verlag.

[12] A.T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal of Scientific and Statistical Computing*, 14:??–??, 1993.

[13] Paul C. van Oorschot. A comparison of practical public-key cryptosystems based on integer factorization and discrete logarithms. In Gustavus J. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, chapter 5, pages 289–322. IEEE Press, Piscataway, 1992.