MATHEMATICS OF COMPUTATION
Volume 68, Number 226, April 1999, Pages 869–880
S 0025-5718(99)01008-X

FACTORING HIGH-DEGREE POLYNOMIALS OVER $\mathbf{F}_2$ WITH NIEDERREITER'S ALGORITHM ON THE IBM SP2

PETER ROELSE

ABSTRACT. A *C* implementation of Niederreiter's algorithm for factoring polynomials over $\mathbf{F}_2$ is described. The most time-consuming part of this algorithm, which consists of setting up and solving a certain system of linear equations, is performed in parallel. Once a basis for the solution space is found, all irreducible factors of the polynomial can be extracted by suitable gcd-computations. For this purpose, asymptotically fast polynomial arithmetic algorithms are implemented. These include Karatsuba & Ofman multiplication, Cantor multiplication and Newton inversion. In addition, a new efficient version of the half-gcd algorithm is presented. Sequential run times for the polynomial arithmetic and parallel run times for the factorization are given. A new "world record" for polynomial factorization over the binary field is set by showing that a pseudo-randomly selected polynomial of degree 300000 can be factored in about 10 hours on 256 nodes of the IBM SP2 at the Cornell Theory Center.

1. INTRODUCTION

Recently, the implementation of polynomial factorization algorithms over finite fields has attracted much attention; see e.g. [6], [7], [9], [11] and [16]. The paper of von zur Gathen and Gerhard is of special interest here since it also describes an implementation for the factorization of high-degree univariate polynomials over $\mathbf{F}_2$. They use techniques based on a distinct/equal degree factorization. In this paper a *C* implementation of Niederreiter's factorization algorithm for factoring polynomials over $\mathbf{F}_2$ is described. In this algorithm, a system of linear equations has to be set up and solved. To solve this system of equations an "explicit" approach is used, i.e. storing the corresponding matrix over $\mathbf{F}_2$ in memory and performing Gaussian elimination on it. An advantage of this approach is that it is highly suited for parallelization. The last stage of the algorithm consists of extracting the factors from a basis for the solution space with suitable gcd-computations. This is performed sequentially, but is usually very fast in comparison with the Gaussian elimination. The implementation was optimized for dense polynomials. It will be shown that it's competitive with the implementations in [7], which seem to be harder to parallelize, provided that enough processors are available.

In Section 2, Niederreiter's algorithm for factoring polynomials in $\mathbf{F}_2[x]$ is described. Section 3 explains how the linear system of equations is set up and solved. The distribution of the matrix over the different processors and some tricks to speed

Received by the editor May 19, 1997 and, in revised form, August 11, 1997.
1991 *Mathematics Subject Classification.* Primary 11–04, 11T06, 11Y16.
Key words and phrases. Finite fields, parallel computing, polynomial factorization.

up the computations will be discussed. Section 4 outlines the different fast polynomial arithmetic algorithms which were implemented. A new efficient version of the half-gcd algorithm is presented. In Section 5, run times on the IBM SP2 are given. An IBM SP2 is basically a collection of RISC System/6000 workstations connected by a high-performance switch, which allows message passing with low latency and high bandwidth. Sequential run times for the fast polynomial arithmetic and parallel run times for the complete factorization are presented.

2. Niederreiter's algorithm

In this section, a brief description of Niederreiter's algorithm for factoring polynomials over $\mathbf{F}_2$ is given. For more detailed information, refer to [13]. Let $f \in \mathbf{F}_2[x]$ be a nonconstant polynomial of degree d with

$$f = \sum_{i=0}^{d} f_i x^i = g_1^{e_1} \cdot g_2^{e_2} \dots g_m^{e_m},$$

where $g_i \in \mathbf{F}_2[x]$ are pairwise different irreducible factors occurring with multiplicities $e_1, e_2, \dots, e_m$. The starting point of Niederreiter's algorithm is the differential equation

$$(fh)' = h^2, \tag{2.1}$$

with $h = \sum_{i=0}^{d-1} h_i x^i \in \mathbf{F}_2[x]$ unknown. The following theorem gives the link to the factorization problem.

Theorem 2.1 (Niederreiter)**.** *The set of solutions $h \in \mathbf{F}_2[x]$ of equation 2.1 is an $\mathbf{F}_2$-space N:*

$$N = \{\ h = \sum_{i=1}^{m} c_i g_i' \frac{f}{g_i} \mid c_1, ..., c_m \in \mathbf{F}_2\ \}.$$

In the following we shall restrict ourselves to square-free polynomials. This is not a serious restriction, since efficient methods exist to reduce the problem to the square-free case (see [19]). The elements of the Niederreiter subspace N can be used to factor f. Observe that for a square-free polynomial f the following equality holds:

$$\gcd(f, h) = \prod_{\{1 \le i \le m \mid c_i = 0\}} g_i,$$

leading to a nontrivial factorization if $h \neq 0$ and $h \neq f'$. If the Niederreiter matrix $\mathcal{N}_f$ is defined as

$$\mathcal{N}_f^T := \begin{pmatrix} f_1 & f_3 & f_5 & \cdots & \cdots & 0 & 0 \\ f_0 & f_2 & f_4 & \cdots & \cdots & 0 & 0 \\ 0 & f_1 & f_3 & \cdots & \cdots & 0 & 0 \\ 0 & f_0 & f_2 & \cdots & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \cdots & f_d & 0 \\ 0 & 0 & 0 & \cdots & \cdots & f_{d-1} & 0 \\ 0 & 0 & 0 & \cdots & \cdots & f_{d-2} & f_d \end{pmatrix},$$

then comparing coefficients in Equation 2.1 gives

$$h \in N \Leftrightarrow \mathbf{h}(\mathcal{N}_f^T - \mathcal{I}_d) = \mathbf{0}, \tag{2.2}$$

where $\mathbf{h} := (h_0 h_1 \cdots h_{d-1})$. So a basis $\{v_1, v_2, \ldots, v_m | v_i \in \mathbf{F}_2[x]\}$ for N can be found by setting up the matrix $\mathcal{N}_f^T - \mathcal{I}_d$ and computing a basis for its *right* nullspace (this is the reason that the transpose of the Niederreiter matrix is used; the nullspace algorithm described in the next section only computes right nullspaces). Note that the dimension of the nullspace coincides with the number of distinct irreducible factors. Assuming this number is larger than one, all irreducible factors of f can be found with the elements of the basis. First compute $j = \gcd(f, v_1)$. If this gcd is trivial, proceed with the computation of $\gcd(f, v_2)$; otherwise compute both $\gcd(j, v_2)$ and $\gcd(f/j, v_2)$. Repeat this procedure of computing the gcd with the next v_i and all factors already found until the number of factors equals m.

3. Computation of the nullspace

The coefficients of the rows of the matrix $\mathcal{N}_f^T - \mathcal{I}_d$ are packed in 32 bit computer words. The operations in $\mathbf{F}_2$ can then be performed by using boolean arithmetic on these words, e.g. addition can be done by an *exclusive or* operation adding 32 field elements in one turn. The matrix is distributed in columns, where the number of columns per processor (which is a multiple of 32) is the same for all processors. An exception is the first processor, which usually gets a bit less columns since this one also handles the IO. To set up the matrix, all processors first precompute all possible 32 bit words that occur in the matrix $\mathcal{N}_f^T$, which is an $O(d)$ process. After this, only assignments are needed. Finally, each processor subtracts its part of the identity matrix.

For the parallel computation of the nullspace of the matrix, a program developed by Michael Weller is used [18]. This program transforms the matrix into full echelon form. The reason that it computes *right* nullspaces is to maintain compatibility with older programs. A problem arising when the right nullspace has to be computed is that column operations have to be performed. Since the rows are packed in computer words, this would mean that operations within these words are needed. This is inefficient, as also transposing the matrix would be. Therefore the column operations are replaced by row operations. The idea for this was developed by Reiner Staszewski and works as follows. A row that equals the pivot row with one minor change is stored. The modification is that the pivot is replaced by a zero. For all rows the element in the pivot column is checked. Whenever this equals one, the modified row is added to this row. This process is illustrated in Example 3.1. With slight extensions this can be generalized to any finite field; see [18].

Example 3.1. Consider the following small example of a 3×3 matrix over $\mathbf{F}_2$. The pivot is chosen to be the first element in the first row. Then the pivot row is $\begin{pmatrix} \mathbf{1} & 1 & 0 \end{pmatrix}$, so the modified row will be $\begin{pmatrix} \mathbf{0} & 1 & 0 \end{pmatrix}$.

$$\begin{pmatrix} \mathbf{1} & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

One can easily see that instead of doing column operations using the pivot column, we can perform row operations using the modified row. Note that we have to perform the row operations on all rows of the matrix, i.e. *including* the pivot row.

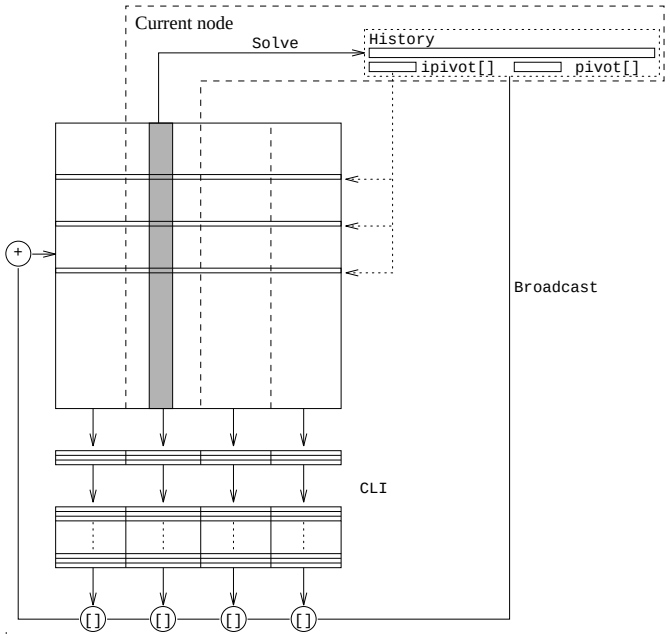

FIGURE 1. Data flow and distribution of the matrix

The elimination process is speeded up by eliminating multiple rows in one turn. For a detailed description of this process, see [18]. In each step a small stripe of columns is selected on the current node (see also Figure 1). This is then transformed by this node to column echelon form by a special column solver. Although this makes the program more sequential, it does speed up the nullspace computation considerably. The indices of the pivot rows are stored in *ipivot*, while *History* contains which linear combination of the pivot rows should be added to each row of the matrix. This is then broadcast to all nodes. Each node then computes all linear combinations of the pivot rows using a highly optimized submodule (CLI) before the appropriate linear combinations are added to the rows of the matrix. An important advantage of the distribution in columns is that the calculation of the linear combinations is fully parallelized.

Sorting the columns after the Gaussian elimination is not done; the program only constructs a table containing the corresponding permutation. However, if the result has to be written to a file, the permutation is performed. This usually is no problem, since only a few basis vectors are expected.

4. Fast polynomial arithmetic

In order to compute the gcd's in the final stage of the algorithm efficiently, several (fast) polynomial arithmetic algorithms were implemented. A polynomial is represented as an array of 32 bit computer words. The bits in these words correspond to the coefficients of the polynomial. Figure 2 shows how the different

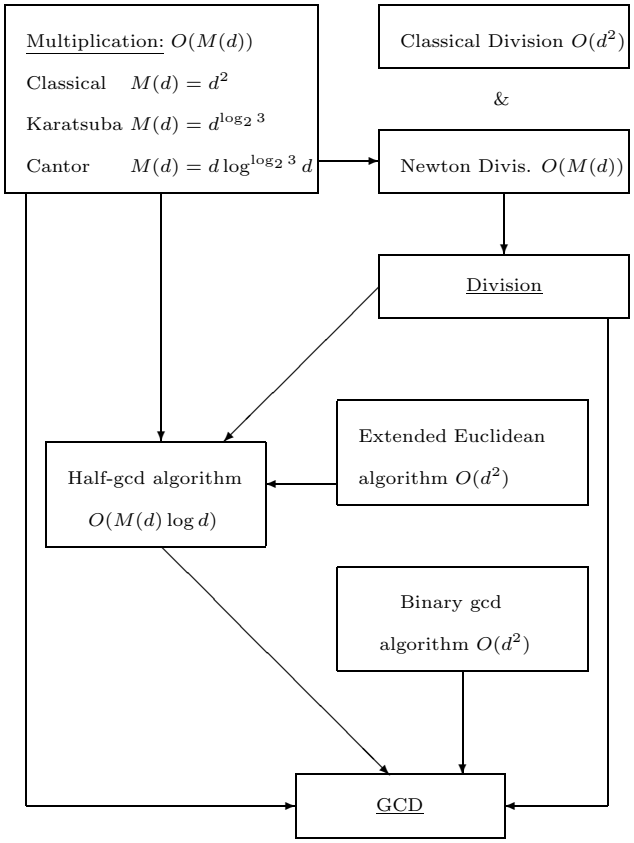

FIGURE 2. Polynomial arithmetic

algorithms depend on each other together with their complexities. Note that efficient multiplication of two polynomials is crucial for the efficient implementation of division and gcd-computations. Observe also that multiplication, division and gcd-computations all have a complexity that is *almost* linear, i.e. they can be computed with $O(d^{1+\varepsilon})$ field operations for any fixed $\varepsilon > 0$.

For a description of Karatsuba & Ofman multiplication and division based on Newton inversion, refer to [1], [8] or [12]. The multiplication method of Cantor, which is based on evaluation and interpolation at additive subspaces of $\mathbf{F}_{2^{2^n}}$, is presented in [3]. In this implementation the field $\mathbf{F}_{2^{16}}$ is used, which allows multiplication of two polynomials over $\mathbf{F}_2$ as long as the sum of the degrees is smaller than 524288. For a detailed description of Cantor's algorithm for multiplication in $\mathbf{F}_2[x]$, including an iterative version, refer to [14].

A description of the half-gcd algorithm, the fast version of the Euclidean algorithm, can be found in [1] and [17]. However, the version in [1] is incorrect while the version in [17] is inefficient, since the polynomials involved are too large. Therefore an efficient and correct version will be given below. Let a_1 and a_2 be two nonzero polynomials for which the gcd has to be computed. Without loss of generality we assume that $\deg(a_1) > \deg(a_2)$. Then the Euclidean algorithm computes the

remainder sequence of a_1 and a_2 such that

$$\begin{aligned} a_1 &= q_1 a_2 + a_3, \\ a_2 &= q_2 a_3 + a_4, \\ &\vdots \\ a_{t-1} &= q_{t-1} a_t, \end{aligned}$$

with $a_{t+1} = 0$ and $a_t = \gcd(a_1, a_2)$. However, this algorithm is not used in this version in the factorization algorithms. For small polynomials the binary gcd algorithm is used (see [2]). For two polynomials of approximately the same degree, the implementation of the binary gcd algorithm is about 30% faster than the implementation of the Euclidean algorithm. For large polynomials the half-gcd algorithm is used, which uses the following fact. If the matrices $\mathcal{M}_i$ are defined as

$$\mathcal{M}_i := \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix},$$

then

$$\begin{pmatrix} a_k \\ a_{k+1} \end{pmatrix} = \mathcal{M}_{k-1}\mathcal{M}_{k-2}\cdots\mathcal{M}_1 \begin{pmatrix} a_1 \\ a_2 \end{pmatrix},$$

with $k \le t$. The half-gcd algorithm computes the product $\mathcal{M}_{k-1}\mathcal{M}_{k-2}\cdots\mathcal{M}_1$ such that the degree of a_k is larger than or equal to half the degree of f, and either $k = t$ (i.e. $a_{k+1} = 0$) or a_{k+1} is the first term in the remainder sequence of a_1 and a_2 for which the degree is smaller than half the degree of a_1. It returns $\deg(f) - \deg(a_k)$. The product of matrices is computed recursively and with polynomials of smaller degree than a_1 and a_2. The following definitions will be needed for the description of the algorithm and a proof of its correctness. If $f = \sum_{i=0}^{d} f_i x^i \neq 0$, then define

$$f \mid l := f_d x^l + f_{d-1} x^{l-1} + \cdots + f_{d-l} \text{ for } 0 \le l \le d.$$

Two pairs of polynomials (f, g) and (h, j) with $\deg(f) > \deg(g) \ge 0$ and $\deg(h) > \deg(j) \ge 0$ are said to coincide up to l if and only if

$$f \mid l = h \mid l,$$

$$g \mid (l - \deg(f) + \deg(g)) = j \mid (l - \deg(h) + \deg(j)).$$

This is only defined if $l \le \min\{\deg(f), \deg(h)\}$, $0 \le l - \deg(f) + \deg(g) \le \deg(g)$ and $0 \le l - \deg(h) + \deg(j) \le \deg(j)$. The following lemma can be found in [17].

Lemma 4.1 (Lehmer/Strassen)**.** *Let $l \ge 0$ and $1 \le k \le t$ be such that $\deg(a_1) - \deg(a_k)\left(= \sum_{i=1}^{k-1} \deg(q_i)\right) \le l$ and either $k = t$ or $\left(\sum_{i=1}^{k} \deg(q_i) =\right) \deg(a_1) - \deg(a_{k+1}) > l$. Consider the Euclidean algorithm for another pair of nonzero polynomials (a_1^*, a_2^*) with $\deg(a_1^*) > \deg(a_2^*)$:*

$$\begin{aligned} a_1^* &= q_1^* a_2^* + a_3^*, \\ a_2^* &= q_2^* a_3^* + a_4^*, \\ &\vdots \\ a_{t^*-1}^* &= q_{t^*-1}^* a_{t^*}^*. \end{aligned}$$

Define k^* similarly. If (a_1, a_2) and (a_1^*, a_2^*) coincide up to $2l$, then $k = k^*$ and $q_i = q_i^*$ for $1 \le i \le k-1$.

The half-gcd algorithm

```
     int half-gcd(R, f, g)
     {
1.       u ← ⌊deg(f)/2⌋
2.       if (u < deg(f) − deg(g))
         {
3.         R ← ( 1  0 )
               ( 0  1 )
4.         return 0
         }
5.       if (deg(f) < C) return half-ext-Euclid(R, f, g)
6.       s ← half-gcd(T, f | u, g | (u − deg(f) + deg(g)))
7.       ( h ) ← T ( f )
         ( j )     ( g )
8.       if (j = 0 or u < deg(f) − deg(j))
         {
9.         R ← T
10.        return s
         }
11.      quotient-remainder(q, r, h, j)
12.      s ← s + deg(q)
13.      if (r = 0 or u < deg(f) − deg(r))
         {
14.        R ← ( 0   1 ) T
               ( 1  −q )
15.        return s
         }
16.      v ← 2(u − s)
17.      s ← s+ half-gcd(S, j | v, r | (v − deg(j) + deg(r)))
18.      R ← S ( 0   1 ) T
               ( 1  −q )
19.      return s
     }
```

Theorem 4.2. *Let $f \neq 0$ and $g \neq 0$ with $\deg(f) > \deg(g)$. The half-gcd algorithm computes the matrix $\mathcal{R}$, such that $\mathcal{R}(f,g)^T = (a_k, a_{k+1})^T$ with the property that $\deg(f) - \deg(a_k) \leq \lfloor \deg(f)/2 \rfloor$ and either $a_{k+1} = 0$ or a_{k+1} is the first term in the remainder sequence of $a_1 := f$ and $a_2 := g$ for which $\deg(f) - \deg(a_{i+1}) > \lfloor \deg(f)/2 \rfloor$. The function returns $\deg(f) - \deg(a_k)$.*

Proof. Assume that there exist functions satisfying the properties for the input for which the algorithm produces a wrong output. Among these functions, there exist an f and a g with $\deg(f) + \deg(g)$ minimal. We call the function with these arguments. The condition in line 2 cannot hold, since the output would be correct in that case. So we have $\deg(f) > \deg(g)$ and $u \geq \deg(f) - \deg(g)$. From this we obtain the following inequalities for $u = \deg(f \mid u)$ and $u - \deg(f) + \deg(g) =$

$\deg(g \mid (u - \deg(f) + \deg(g))$:

$$\deg(f) > \lfloor \deg(f)/2 \rfloor = u > u - \deg(f) + \deg(g),$$

$$\deg(g) > u - \deg(f) + \deg(g) \geq 0.$$

Line 5 of the algorithm is for efficiency reasons only; if the arguments are of small degree the matrix is computed by the classical extended Euclidean algorithm. It is assumed that the result of this call is correct. The output of the recursive call to the half-gcd function in line 6 has to be correct, since the arguments satisfy the input conditions and have smaller degrees than f and g respectively. Note that (f, g) and $(fu, g|(u - \deg(f) + \deg(g)))$ coincide up to $2\lfloor \deg(f)/4 \rfloor$. If $(a^*_{k^*}, a^*_{k^*+1})^T = \mathcal{T}(f|u, g|(u - \deg(f) + \deg(g)))^T$, then by Lemma 4.1 and induction we see that h and j in line 7 are two polynomials in the remainder sequence of f and g with $s = \deg(f|u) - \deg(a^*_{k^*}) = \deg(f) - \deg(h) \leq \lfloor \deg(f)/4 \rfloor$. If the condition in line 8 would hold, the output would be correct, contradicting the assumption. So we continue with line 11. Right after line 12 it holds that $s = \deg(f) - \deg(h) + \deg(q) = \deg(f) - \deg(j) \leq u$. So the condition in line 13 also cannot hold, since the output would then be correct. After line 15 we have $\deg(j) > \deg(r)$ and $u \geq \deg(f) - \deg(r)$. For $v = \deg(j|v)$ and $v - \deg(j) + \deg(r) = \deg(r|(v - \deg(j) + \deg(r))$ it follows that:

$$\deg(f) > \deg(j) \geq 2u - s > 2(u - s) = v > v - \deg(j) + \deg(r),$$

$$\deg(g) > \deg(r) > v - \deg(j) + \deg(r) \geq \deg(j) - \deg(r) > 0.$$

So again the recursive call to the function must give a correct output. Next observe that (j, r) and $(j|v, r|(v - \deg(j) + \deg(r)))$ coincide up to $2(u - s)$. Let

$$(a_k, a_{k+1})^T := \mathcal{S}(j, r)^T \left(= \mathcal{R}(f, g)^T\right)$$

and

$$(\tilde{a}_{\tilde{k}}, \tilde{a}_{\tilde{k}+1})^T = \mathcal{S}(j|v, r|(v - \deg(j) + \deg(r)))^T.$$

Then by Lemma 4.1 and induction we have

$$\deg(j) - \deg(a_k) = \deg(j|v) - \deg(\tilde{a}_{\tilde{k}}) \leq v/2$$

and either $\tilde{a}_{\tilde{k}+1} = 0$ or $\deg(j \mid v) - \deg(\tilde{a}_{\tilde{k}+1}) > v/2$. If $a_{k+1} = 0$, then the output would be correct, so we can assume that $a_{k+1} \neq 0$. If $\deg(j) - \deg(a_{k+1}) \leq v/2$, then by Lemma 4.1 $q_k = \tilde{q}_{\tilde{k}}$. This would imply that also $\deg(j \mid v) - \deg(\tilde{a}_{\tilde{k}+1}) \leq v/2$. Since this is not true, we know that $\deg(j) - \deg(a_{k+1}) > v/2$. From this it follows that

$$\begin{aligned}\deg(f) - \deg(a_k) &= \deg(f) - \deg(j) + \deg(j) - \deg(a_k)\\ &\quad (= \deg(f) - \deg(j) + \deg(j|v) - \deg(\tilde{a}_{\tilde{k}}) = s)\\ &\leq \deg(f) - \deg(j) + v/2 = \lfloor \deg(f)/2 \rfloor\end{aligned}$$

and

$$\begin{aligned}\deg(f) - \deg(a_{k+1}) &= \deg(f) - \deg(j) + \deg(j) - \deg(a_{k+1})\\ &> \deg(f) - \deg(j) + v/2 = \lfloor \deg(f)/2 \rfloor,\end{aligned}$$

which contradicts the assumption that the output is wrong. □

gcd algorithm

```
        pol gcd(f, g)
        {
1.          h ← f
2.          j ← g
3.          for (;;)
            {
4.              quotient-remainder(q, r, h, j)
5.              if (r = 0) return j
6.              if (deg(j) < K) return binary-gcd(j, r)
7.              s ← half-gcd(R, j, r)
8.              ( h ) ← R ( j )
                ( j )     ( r )
9.              if (j = 0) return h
            }
        }
```

In order to compute the gcd of two polynomials the half gcd algorithm usually has to be iterated. For completeness, a gcd algorithm is given. The arguments should be non-zero, and $\deg(f) \geq \deg(g)$. For polynomials over fields other than $\mathbf{F}_2$, the binary gcd algorithm can be replaced by the Euclidean algorithm if necessary. The value returned by the half-gcd algorithm is used to compute the matrix-vector products in line 7 of the half-gcd algorithm and in line 8 of the gcd algorithm more efficiently. If

$$\begin{pmatrix} h \\ j \end{pmatrix} \leftarrow \mathcal{T} \begin{pmatrix} f \\ g \end{pmatrix},$$

then only the first $\lfloor(\deg(f) - s)/32\rfloor + 1$ computer words (32 bits) of f and g are needed to compute the matrix-vector product, since we know that $\deg(h) = \deg(f) - s < \deg(g)$ and either $j = 0$ or $\deg(j) < \deg(f) - s$ This speeds up the gcd computation considerably.

5. Run times

This section presents run times of the implementations. All programs were written in *C* and were optimized for speed. Techniques for improving a program's performance can be found in [4] and [10]. The IBM AIX XL C Compiler/6000 with optimization option -O3 was used to compile the programs.

The sequential run times (in seconds) for the fast polynomial arithmetic in Table 1 were obtained using an IBM RS6000 - 590 with a POWER2 processor, rated at 67 Mhz. The times represent average running times for multiplying two polynomials of degree d, computing the quotient and remainder when dividing a degree $2d$ polynomial by a degree d polynomial, and computing the gcd of two degree d polynomials, respectively. All polynomials were selected pseudo-randomly, using the standard C `rand()` function. It should be mentioned that the times for the division based on Newton inversion and the half-gcd algorithm can be improved by saving some multi-point evaluations of polynomials, e.g. in the matrix product in

TABLE 1. Sequential CPU times for polynomial arithmetic in $\mathbf{F}_2[x]$

Degree	Multiplication		Division		GCD's	
d	Karats.	Cantor	Classical	Newton	Bin.-gcd	gcd-alg.
8000	0.02	0.03	0.05	0.05	0.17	0.17
16000	0.05	0.06	0.19	0.15	0.64	0.64
32000	0.14	0.13	0.76	0.42	2.52	1.94
64000	0.41	0.26	2.93	0.98	10.17	5.72
128000	1.22	0.61	12.35	2.26	40.42	15.22
256000	3.76	1.34	49.86	4.98	162.62	39.82

line 18 of the half-gcd algorithm. This was also pointed out in [15]. At the moment this is not implemented.

The parallel run times for the complete factorization were taken at IBM SP2 installations. The ones in Table 2 were obtained in Bonn (Germany). The times in Table 3 were obtained in Cornell (USA), where more nodes are available. One node corresponds to an IBM RS6000 as described above with 128 MB main memory. The library used for the communications between the nodes is Parallel Operating Environment (POE) with the Userspace (US) protocol. This was specially designed for the IBM SP and uses its high performance switch. The run times represent the time for factoring one pseudo-randomly selected (dense) polynomial of degree d. They are split into three parts: the time to set up the matrix, the time for the Gaussian elimination and the (sequential) time to extract the irreducible factors from the polynomial using the basis for the nullspace. These times are all in seconds, while the total run time is in hours-minutes-seconds. The parameter λ represents the amount of precomputation, i.e. the number of rows that are eliminated in one single turn. The missing times in Table 2 are due to a lack of main memory. Note that storing the $d \times d$ matrix needs $d^2/8$ bytes of main memory. For the largest example, the degree 300000 polynomial, this corresponds to 11.25 GB. This is about 44 MB for one workstation. The precomputation needs $2^\lambda d/8$ bytes of memory. For the degree 300000 polynomial, this equals 2.4 MB for one node.

To the best of our knowledge, the largest instances of polynomial factorization over the binary field in the literature are a trinomial of degree 216091 (see [5]) and a pseudo-randomly selected polynomial of degree 262143 (see [7]). With the factorization of the pseudo-randomly selected polynomial of degree 300000 a new "world record" is set. This shows that Niederreiter's algorithm for polynomial factorization over $\mathbf{F}_2$ (for which the algorithm looked particularly promising from the very beginning) is the fastest factorization algorithm currently available inpractice.

TABLE 2. Parallel CPU times for polynomial factorization in $\mathbf{F}_2[x]$ (Bonn)

Degree d	λ	Task	Number of processors 8	16	32	Degrees of factors
8000	10	setup	0.03	0.02	0.02	1,5,10,16,29,59,
		nullspace	26.0	21.3	19.4	152,234,256,697,
		extract	0.76	0.76	0.76	725,1433,1587,
		total	$0^h00'27''$	$0^h00'22''$	$0^h00'20''$	2796
16000	11	setup	0.10	0.06	0.04	8,12,13,14,19,42,
		nullspace	134	103	89.2	519,7374,7999
		extract	2.09	2.09	2.09	
		total	$0^h02'16''$	$0^h01'45''$	$0^h01'31''$	
32000	11	setup	0.35	0.20	0.12	7,11,96,381,436,
		nullspace	718	548	436	777,912,2235,
		extract	8.79	8.79	8.79	11485,15660
		total	$0^h12'07''$	$0^h09'17''$	$0^h07'25''$	
64000	12	setup	1.36	0.75	0.41	1,1,5,25,30,125,
		nullspace	4396	2941	2107	340,668,2369,
		extract	29.0	29.0	29.0	5388,16244,
		total	$1^h13'46''$	$0^h49'30''$	$0^h35'36''$	38804
128000	12	setup	—	—	1.47	6,36,83,92,133,
		nullspace	—	—	11377	140,368,958,
		extract	—	—	67.0	2213,4849,9051,
		total	—	—	$3^h10'45''$	17211,92860

TABLE 3. Parallel CPU times for polynomial factorization in $\mathbf{F}_2[x]$ (Cornell)

Degree d	#procs	λ	Task	CPU time	Degrees of factors
75000	64	12	setup	0.27	1,17,35,40,81,
			nullspace	2641	311,622,12225,
			extract	26.9	21515,40153
			total	$0^h44'28''$	
150000	128	13	setup	0.54	2,5,27,36,47,66,276,
			nullspace	9623	284,316,948,3690,
			extract	106	5066,8536,14534,
			total	$2^h42'10''$	27871,88296
300000	256	14	setup	1.25	2,4,165,233,275,
			nullspace	36540	10965,18709,
			extract	236	41770,100948,
			total	$10^h12'57''$	126929

ACKNOWLEDGMENTS

The author is very grateful to the Cornell Theory Center, Cornell University, Ithaca, NY and the GMD, St. Augustin, Germany for offering us the CPU time on their IBM SP2 installations. Without their support these computations would

not have been possible. Special thanks go to Michael Weller for permission to modify and use his parallel nullspace algorithm. He also wants to thank Reiner Staszewski, who implemented the classical multiplication of polynomials and several subroutines of the nullspace program. Helpful discussions with Peter Fleischmann are also highly appreciated. He thanks the referee for valuable remarks and for bringing the article [5] to his attention.

References

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974. MR **54:**1706
2. E. Bach and J. Shallit, *Algorithmic Number Theory - Volume 1: Efficient Algorithms*, The MIT Press, 1996. MR **97e:**11157
3. D.G. Cantor, *On arithmetical algorithms over finite fields*, J. Combin. Theory Ser. A **50**, (1989), 285–300. MR **90f:**11100
4. K. Dowd, *High Performance Computing*, O'Reilly & Associates Inc., 1993.
5. A. Diaz, E. Kaltofen and V. Pan, *Algebraic algorithms*, The Computer Science and Engineering Handbook (A.B. Tucker, ed.), CRC Press, 1997, pp. 226–249.
6. P. Fleischmann and P. Roelse, *Comparative implementations of Berlekamp's and Niederreiter's polynomial factorization algorithms*, Finite Fields and their Applications (S. Cohen and H. Niederreiter, eds.), Cambridge University Press, 1996, pp. 73–84. MR **98a:**12009
7. J. von zur Gathen and J. Gerhard, *Arithmetic and factorization of polynomials over* F_2, Proc. ISSAC '96, ACM Press, 1996, pp. 1-9.
8. K.O. Geddes, S.R. Czapor and G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, 1992. MR **96a:**68049
9. J. Gerhard, *Faktorisieren von Polynomen über* $\mathbf{F}_q$*: ein Vergleich neuerer Verfahren*, Master's thesis, University of Erlangen-Nürnberg, 1994.
10. IBM, *AIX Version 3.2 for RISC System/6000: Optimization and Tuning Guide for Fortran, C, and C++*, IBM Manual, 1993.
11. E. Kaltofen and A. Lobo, *Factoring high-degree polynomials by the black box Berlekamp algorithm*, Proc. ISSAC '94 (J. von zur Gathen and M. Giesbrecht, eds.), ACM Press, 1994, pp. 90–98.
12. A. Karatsuba and Y. Ofman, *Multiplication of multidigit numbers on automata*, Soviet Phys. Dokl. **7** (1963), 595–596.
13. H. Niederreiter, *New deterministic factorization algorithms for polynomials over finite fields*, Contemp. Math. **168** (1994), 251–268. MR **95f:**11100
14. D. Reischert, *Schnelle Multiplikation von Polynomen über* $GF(2)$ *und Anwendungen*, Master's thesis, University of Bonn, 1995.
15. D. Reischert, *Multiplication by a Square is cheap over* F_2, Preprint, 1996.
16. V. Shoup, *A new polynomial factorization algorithm and its implementation*, J. Symbolic Comput. **20** (1995), 363–397. MR **97d:**12011
17. V. Strassen, *The computational complexity of continued fractions*, SIAM J. Comput. **12** (1983), 1–27. MR **84b:**12004
18. M. Weller, *Parallel Gaussian elimination over small finite fields*, Parallel and Distributed Computing Systems, Proc. of the ISCA International Conference (K. Yetongnon and S. Hariri, eds.), 1996, 56–63.
19. D.Y.Y. Yun, *On square-free decomposition algorithms*, Proc. ACM Symp. Symbolic and Algebraic Comput. (R.D. Jenks, ed.), 1976, pp. 26–35.

Institute for Experimental Mathematics, University of Essen, Ellernstrasse 29, 45326 Essen, Germany

Current address: Philips Crypto B.V., De Witbogt 2, 5652 AG Eindhoven, The Netherlands

E-mail address: roelse@exp-math.uni-essen.de, roelse@crypto.philips.com