Programming Research Group

A BSP PARALLEL MODEL OF THE GÖTTFERT ALGORITHM FOR POLYNOMIAL FACTORIZATION OVER F_2

Fatima Abu Salem

PRG-RR-03-14

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

In Niderreiter's factorization algorithm for univariate polynomials over finite fields, the factorization problem is reduced to solving a linear system over the finite field in question, and the solutions are used to produce the complete factorization of the polynomial into irreducibles. For fields of characteristic 2, a polynomial time algorithm for extracting the factors using the solutions of the linear system was developed by Göttfert, who showed that it is sufficient to use only a basis for the solution set. In this paper, we develop a new BSP parallel algorithm based on the approach of Göttfert over the binary field, one that improves upon the complexity and performance of the original algorithm for polynomials over F_2. We report on our implementation of the parallel algorithm and establish how it achieves very good efficiencies for many of the case studies. When combined with our work in [1] for solving large sparse Niederreiter linear systems over the binary field, this provides an efficient alternative to other implementations of the Niederreiter algorithm for the factorization of large sparse polynomials over F_2. [1]

1 Introduction

Symbolic computation has been the object of computer algebra systems for decades now, focusing on the use of computers to perform symbolic mathematics. Parallelism in computer algebra has also been an intriguing topic; see for instance [5], [21], [23]. One major problem in computer algebra is the factorization of univariate polynomials over finite fields. An important algorithm in this domain is the Niederreiter algorithm [16] which reduces the factorization problem to solving a linear system over the finite field in question. The solutions of the linear system can lead to a complete factorization in a variety of ways, one of which was presented by Göttfert [8] for fields of characteristic 2, leading to a simple and polynomial time algorithm for extracting the factors using only the basis elements of the solution set of the Niederreiter linear system.

In this paper, we develop a new BSP parallel approach to the Göttfert algorithm over F_2. The BSP model offers simplicity in terms of its cost analysis and its clear distinction between the three important phases of computation, communication, and synchronization. Its also has the advantage of being independent of the underlying architecture of the machine, thus providing portable software that can be used efficiently in a variety of applications (see [2], [9]). Our algorithm achieves high efficiency in many of our test cases and can thus be used efficiently to factorize very large polynomials over F_2 provided a basis of the solution set is given. In Sections 2 and 3, we give a brief survey of the algorithms underlying our work and some background information describing the BSP parallel model. In Section 4 we present our parallel algorithm, prove its correctness and discuss its BSP cost analysis. In Section 5 we report on our experimental results and discuss the scalability of the algorithm.

[1]Key-words: symbolic computation, parallelism in computer algebra, finite fields, polynomial factorization, BSP model.

2 Göttfert's Refinement of Niederreiter's Algorithm over The Binary Field

For complete proofs and results related to the Niederreiter algorithm we refer the reader to [8], [11], [15], [16], [17], [18], [19]. Let F_2 be the binary field of order 2 consisting only of the elements $\{0, 1\}$; it is thus understood that all polynomials described in this paper are monic. Let f be a polynomial of degree d over F_2, and

$$f = g_1^{e_1} ... g_m^{e_m}$$

be its canonical factorization of over the field. Let N_f be the Niederreiter matrix of coefficients of f (e.g. see [8], [16]). The Niederreiter linear system can be expressed as

$$(N_f - I_d)\mathbf{h}^T = \mathbf{0}, \tag{1}$$

where I_d is the $d \times d$ identity matrix over F_2 and $\mathbf{h} = (h_0, ..., h_{d-1})$ is the coefficient row vector of an unknown polynomial $\mathbf{h}$ over F_2 of degree less than d. In [16], Niederreiter establishes that the solutions $\mathbf{h}$ of the system (1) form a linear subspace of the vector space $F_2[x]$ of dimension m over F_2 and that they are given by

$$\mathbf{h} = \frac{f}{b} b'$$

where b denotes a factor of $g_1 ... g_m$. Since b is square-free (so that $\gcd(b, b') = 1$), we also have

$$\gcd(f, \mathbf{h}) = \frac{f}{b} \gcd(b, b') = \frac{f}{b}.$$

Suppose that $m > 1$ so that the factorization is non-trivial. Several methods can be used to produce the complete factorization of f once a solution of the linear system is determined. In the first method, one chooses a solution $\mathbf{h}$ of the linear system such that the corresponding polynomial whose coefficients form the row vector $\mathbf{h}$ is not equal to zero of f'. If $\mathbf{h} \neq \mathbf{0}$, then $\gcd(f, \mathbf{h}) \neq f$ and if $\mathbf{h} \neq f'$, then $b \neq f$ and so $\gcd(f, \mathbf{h}) \neq 1$. One then iterates the algorithm using the non-constant factor $\gcd(f, \mathbf{h})$ of f using the Neiderreiter algorithm. Although this would lead to a polynomial time algorithm for extracting all the factors of f, it is highly inconvenient in practice, as it requires setting up a new matrix and solving the associated system once again. In the second method, one can produce all 2^m solutions of the linear system, which in turn produce all 2^m factors of the squarefree part of f given by $b = f / \gcd(f, \mathbf{h})$. This does not have the disadvantage associated with setting up and solving a linear system several times during one factorization process, but it does not lead to a polynomial time algorithm for extracting the factors, as it requires 2^m gcd computations. A third method was suggested by Göttfert [8], who showed that it is enough to use the m polynomials corresponding to any basis of the linear system in order to determine a complete factorization into irreducibles using at most m^2 gcd calculations and m^2 divisions, thus leading to another polynomial time algorithm for extracting the factors, without having to solve other than the original linear system associated with f.

We describe this approach briefly here and refer to the original paper [8] for details and proofs of the following results.

Let $\{\mathbf{h_1}, ..., \mathbf{h_m}\}$ be a basis spanning the solution set of the system (1). The corresponding polynomials

$$b_i = \frac{f}{\gcd(f, \mathbf{h_i})} \in F_2[x] \quad \text{for } i = 1, ..., m$$

are square-free factors of f. The factors are listed in a collection of rows as follows. The first row contains only b_1. The second row consists of at most three polynomials, specifically, the non-constant polynomials among

$$\gcd(b_2, b_1), \quad \frac{b_1}{\gcd(b_2, b_1)}, \quad \frac{b_2}{\gcd(b_2, b_1)}.$$

In general, the polynomials of the n-th row for $n = 3, ..., m$ consist of the non-constant polynomials among

$$d_1, \frac{r_1}{d_1}, ..., d_s, \frac{r_s}{d_s}, \frac{b_n}{d_1 ... d_s}$$

where $r_1, ..., r_s$ are the polynomials in row $n-1$ and $d_j = \gcd(b_n, r_j)$ for $j = 1, ..., s$. In [8], it is shown that the polynomials in any row are pairwise relatively prime square-free factors of f, that the polynomial b_n appears in row n, either in its original form or split up into some non-constant factors, and that every polynomial in the row $n-1$ also appears in row n, either in its original form or split up into two non-constant factors. More importantly, it is shown that this process ends successfully by setting up at most m rows, as a consequence of the following theorems:

Theorem 1 *The irreducible square-free factors of f are determined once a row containing m non-constant polynomials has been reached.*

Theorem 2 *A row of index at most m contains the polynomials $g_1, ..., g_m$, the distinct irreducible factors of f.*

As a result, and since at most m rows have to be set up, each containing at most m polynomials, we need at most m^2 gcd operations and at most m^2 division operations to find all irreducibles, thus bringing the total cost of such computations to $O(m^2 M(d) \log d)$ field operations, where $\log d$ is the binary logarithm of d, $M(d)$ is the time to multiply (or divide) two polynomials of degree at most d over F_2, and $O(M(d) \log d)$ is the time to perform the gcd of two such polynomials over F_2.

3 The Bulk Synchronous Parallel Model (BSP)

The *bulk synchronous parallel model* is a model for parallel programming which provides a simple framework to achieve portable parallel algorithms independent of the architecture of the computer on which the parallel work is carried out. The model is attractive because of its simple cost function which helps predict the running time of parallel algorithms before implementing them. For a detailed description of the BSP model, we refer the reader to [22]. In our implementation of the BSP model, we use the standard BSP library [12], [13]. An alternative is the Paderborn University BSP (PUB) library [4] which has the extra feature of allowing subset synchronization and hence, very importantly, nested parallelism.

A BSP computer consists of a set of p processors each with its own private memory, and having remote access to other processors' private memories through a communication network. A BSP algorithm consists of a sequence of parallel steps, denoted by *supersteps*. A computation superstep is a series of computations performed on local data available to the processor before the superstep. A communication superstep is a series of communications in the form of sending or receiving a number of non-local data between processors that are needed to perform local computations. Communication supersteps are followed by *synchronization barriers*, whereby all transferred data is updated.

Our algorithm is designed as a SPMD model, in which a single program with multiple data is encountered by all processors, which then execute their own version of the program, distinguished by their own identification number, id, which takes values from zero to $p-1$, where p is the total number of processors operating in parallel. A BSP computer can be described by the following four parameters:

- p, the number of processors available;
- s, the processor speed in flop/sec;
- g, the time (in flop time units) it takes to communicate (send or receive) a data element;
- ℓ, the time (in flop time units) it takes all processors to synchronize.

We distinguish between the BSP cost of an algorithm and its expected running time. The BSP cost is established using the parameters g and ℓ and the estimate of the execution time is obtained by dividing the BSP cost in flop time units by s, the single processor speed. The BSP cost of an algorithm is simply the sum of the BSP costs of its supersteps. The complexity of a superstep is defined as

$$w_{max} + g.h_{max} + \ell,$$

where w_{max} is the maximum number of flops performed, and h_{max} is the maximum number of messages sent or received by any one processor during that superstep. In the present paper, all field operations are considered as flops, since we are working over the binary field.

4 A Parallel Approach to Göttfert's Refinement of the Niederreiter Algorithm

Let $\#r_n$ denote the maximum number of non-constant polynomials P_i, for $i = 1, ..., \#r_n$, that can appear in any row n described in the construction above. Each P_i can be either the result of a gcd or a division operation, in which case we denote it by a D-polynomial or an R-polynomial respectively. It is easy to see that that there are at most $(\#r_n - 1)/2$ non-constant D-polynomials and at most $(\#r_n + 1)/2$ non-constant R-polynomials in each row n. We denote D and R-polynomials in row n by $n; D_j$ and $n; R_{j'}$ respectively, where j and j' are the polynomials' indices along row n. For consistency throughout the text, we can arrange the computations along rows so that all the D polynomials are computed first, their corresponding R polynomials next, and the polynomial $b_n / \prod_j n; D_j$ (where the product is over non-constant polynomials $n; D_j$) last. With this notation, it is also easy to see that, if the polynomials in row $n-1$ are written as

$$(n-1); D_i, \text{ for some } i = 1, ..., (\#r_{n-1} - 1)/2,$$

and

$$(n-1); R_i, \text{ for some } i = (\#r_{n-1} + 1)/2, ..., \#r_{n-1},$$

then row n consists of

$$n; D_i = \begin{cases} \gcd(b_n, (n-1); D_i), & \text{if } 1 \le i \le (\#r_{n-1} - 1)/2, \\ \gcd(b_n, (n-1); R_{i-(\#r_{n-1}-1)/2}), & \text{if } (\#r_{n-1} + 1)/2 \le i \le \#r_{n-1}, \end{cases}$$

$$n; R_i = \begin{cases} (n-1); D_i / n; D_i, & \text{if } 1 \le i \le (\#r_{n-1} - 1)/2, \\ (n-1); R_{i-(\#r_{n-1}-1)/2} / n; D_i, & \text{if } (\#r_{n-1} + 1)/2 \le i \le \#r_{n-1}, \end{cases}$$

and

$$n; R_{\#r_{n-1}+1} = \frac{b_n}{\prod\limits_{\substack{i=1 \\ n; D_i \neq 1}}^{\#r_{n-1}} n; D_i}.$$

Furthermore, we assert the following:

Claim 1 $\#r_n = 2^n - 1$ *for* $n = 1, ..., m$.

Proof:
We prove the claim by inducion on n. For $n = 1$, we know that row 1 consists of the polynomial b_1 only. Suppose the assertion is true for n. We know that any row $n+1$ has at most one plus twice the number of non-constant polynomials in row n so that

$$\#r_{n+1} = 2\#r_n + 1 = 2.(2^n - 1) + 1 = 2^{n+1} - 1.$$

$\square$

The first step in our parallel approach consists of studying the dependencies between the gcd and division computations and structuring these dependencies in a parallel hierarchy. Without loss of generality we may assume that the number of threads coincides with the number of processors available. We introduce the concept of a parallel queue, which consists of a list of polynomials that can be computed independently by a number of p processors using a number of supersteps. The queue comprises a set of jobs that are not necessarily performed in the same parallel superstep; however, the jobs are entirely independent and hence can be performed in any order and do not require that the processors synchronize at any point before the queue is fully tackled. The parallel queue consists of the polynomials b_i, for $i = 1, ..., m$, where m polynomials can be computed simultaneously in parallel. The second parallel queue consists of the polynomial $2; D_1$ only, since all other polynomials (in its row or in following rows) depend on it. This constitutes the only queue where not enough distinct tasks are available to engage all processors. In fact, the ensuing queues start filling up immediately according to an iterative formula derived from the dependencies that we describe in the following algorithm:

Algorithm 1 *Set_Queues*($queue_k, queue_{k'}$)
Input: $queue_k = \{P_1, ..., P_s\}$, *a list of non-constant polynomials from the Göttfert setting computed in a parallel queue* $k \geq 2$.
Output: $queue_{k'}$, *a list of polynomials that can be computed in the parallel queue* $k' > k$.
1. $queue_{k'} = ()$;
for $j \in \{1, ..., s\}$ **do**
  if $P_j = n; D_i$ *for some* $n = 2, ..., m$ *and some* $i = 1, ..., \#r_{n-1}$ **do**
2. $queue_{k'} \leftarrow queue_{k'} \cup n; R_i \cup (n+1); D_i$.
  end;
  if $P_j = n; D_i$ *for some* $n = 2, ..., m$ *and* $i = \#r_{n-1}$ **do**
3. $queue_{k'} \leftarrow queue_{k'} \cup n; R_{\#r_{n-1}+1}$.
  end;
  if $P_j = n; R_i$ *for some* $n = 2, ..., m$ *and some* $i = 1, ..., \#r_{n-1} + 1$ **do**
4. $queue_{k'} \leftarrow queue_{k'} \cup (n+1); D_{i+((\#r_n - 1)/2)}$.
  end;
end.

Theorem 3 *The algorithm works correctly as specified, producing all the rows in the Göttfert algorithm required to achieve a complete factorization. As a result, the algorithm requires at most* $3s$ *steps for a list of size* s.

Proof:
We assume that the polynomials $b_1, ..., b_m$ are alread computed. Correctness of the algorithm follows as a result of justifying the steps 2-5:

Step 2: Suppose $P_j = n; D_i$ for some $n = 2, ..., m$ and some $i = 1, ..., \#r_{n-1}$. If $1 \leq i \leq (\#r_{n-1} - 1)/2$, then

$$P_j = \gcd(b_n, (n-1); D_i) \text{ and } n; R_i = \frac{(n-1); D_i}{P_j}.$$

Since P_j has already been computed in queue k, we know that $(n-1); D_i$ must be a non-constant polynomial computed in queue $j < k$. As such, $n; R_i$ has both its components available and can be assigned to $queue_{k'}$. Else, if $(\#r_{n-1} + 1)/2 \leq i \leq \#r_{n-1}$, then

$$P_j = \gcd(b_n, (n-1); R_{i-((\#r_{n-1}-1)/2)}) \text{ and } n; R_i = \frac{(n-1); R_{i-((\#r_{n-1}-1)/2)}}{P_j}.$$

Again, $(n-1); R_{i-((\#r_{n-1}-1)/2)}$ must be a non-constant polynomial computed in queue $j < k$, so that $n; R_i$ can be assigned to $queue_{k'}$. On the other hand, for $i = 1, ..., \#r_{n-1}$ (or $i = 1, ..., (\#r_n - 1)/2$) we know that

$$(n+1); D_i = \gcd(b_{n+1}, n; D_i).$$

The proof now follows as above.

Step 3: Suppose $P_j = n; D_i$ for some $n = 2, ..., m$ and $i = \#r_{n-1}$. Since $n; D_{\#r_{n-1}}$ is the D-polynomial to be computed last in row n, and since

$$n; R_{\#r_{n-1}+1} = \frac{b_n}{\prod\limits_{\substack{i=1 \\ n;D_i \neq 1}}^{\#r_{n-1}} n; D_i},$$

the proof follows as above.

Step 4: Suppose that $P_j = n; R_i$ for some $n = 2, ..., m$ and some $i = 1, ..., \#r_{n-1} + 1$. We know that i also satisfies $i = 1, ..., (\#r_n + 1)/2$ or

$$\frac{(\#r_n + 1)}{2} \leq i + \frac{(\#r_n - 1)}{2} \leq \#r_n.$$

Moreover, we have

$$(n+1); D_i = \gcd(b_{n+1}, n; R_{i-((\#r_n-1)/2)}) \text{ if } \frac{(\#r_n + 1)}{2} \leq i \leq \#r_n,$$

or equivalently

$$(n+1); D_{i+\frac{\#r_n-1}{2}} = \gcd(b_{n+1}, n; R_i) \text{ if } \frac{(\#r_n + 1)}{2} \leq i + \frac{(\#r_n - 1)}{2} \leq \#r_n.$$

The proof now follows as above.

With the correctness of the algorithm now established, it becomes immediate to see that any polynomial in $queue_k$ can lead to at most 3 polynomials in $queue_{k'}$ (e.g. the polynomial satisfying the conditions in steps 2 and 3 above). □

4.1 The Parallel Göttfert Algorithm

One major characteristic of the algorithm is that it consists of task parallelism, since distributing the data would require much more synchronization between processors in the inner loops than would be the case in our present algorithm. To minimize the number of synchronization barriers, we choose to make all initial data available globally at the beginning of the algorithm and all recently computed data available to all processors once they are obtained. The following details are crucial in following up on the algorithm and describe some of the data structures as well as the notations we adopt throughout this section. The polynomials are represented by integer arrays whose entries are either zero or one. The coefficients are packed into bit-words (where w_l is the bit-size of the computer word being used). This not only speeds up the polynomial arithmetic sub-routines but also minimizes the number of messages to be transmitted among processors, and hence the BSP cost of the algorithm. We describe several arrays that store either the values of the polynomials or information about them. Unless otherwise stated, all arrays are global.

We first define two copies of three integer arrays, $Type_{j,j'}$, $Row_{j,j'}$, and $Index_{j,j'}$, each of size m. Those serve to hold temporarily information about a polynomial P_i being computed in some parallel queue. In particular, $Type_{j,j'}[i]$ denotes the type of the polynomial (whether a D or an R polynomial), $Row_{j,j'}[i]$ the row to which it belongs, and $Index_{j,j'}[i]$ its index within that row. Those arrays are embedded within two queues $queue_j$ and $queue'_j$ such that $queue_j$ is a sequence of triples $(Type_j[i], Row_j[i], Index_j[i])$, for $i = 0, ..., \#(queue_j) - 1$, and each such triple describes a polynomial already computed in some parallel queue. On the other hand, $queue_{j'}$ consists of similar triples describing polynomials to be computed in a forthcoming parallel queue.

By a simple abuse of notation we also define what we call an array of polynomials $Poly_k$ of size $\lceil d/w_l \rceil$ (where w_l is the bit-size of the computer word being used). By this we simply mean a double array of integers $Poly$ such that the k'th row of the double array contains the coefficients of the polynomial $Poly_k$. The array is used to store permanently the values of all non-constant polynomials computed in the parallel process. Similary, we define an array of polynomials B_k of size $\lceil d/w_l \rceil$ containing the coefficients (in bit-words) of the polynomials $b_i = f/gcd(f, \mathbf{h_i})$, for $i = 0, ..., m-1$. We note that the polynomial indices are global variables indicating that $Poly_k$ and B_k are global polynomials whose individual values are computed by one particular processor then broadcasted to all at one fixed location k independent of the processor id. To keep track of the number of non-constant polynomials in each row, we define the integer array $Length$ of size m such that $Length[i]$ denotes the total number of non-constant polynomials located along row i during any phase of the parallel process. We also define two double arrays of pointers, D and R, of approximate sizes $m \times (2^m - 1)$. $D[n][i]$ points to null if the polynomial $n; D_i$ is constant; else, it contains the address of the row in $Poly$ where the polynomial $n; D_i$ is stored. A similar description holds for the array R. Finally, we define two integer arrays sum, and $local\_sums$, of sizes m and $p \times m$ respectively, that are used to update the lengths of rows individually by each processor, as will be described later on.

All communication between processors is achieved through the bsp_put command ([4], [12], [13]). We use the short-hand of the function call as in

$$y \leftarrow BSP\_Put(s, N, x).$$

where N is an integer greater than or equal to zero. If $N = 0$, this indicates that the processor meeting the command is sending its individual value of variable x onto variable y found on processor s. Otherwise, x denotes a polynomial whose coefficients form the first N entries of array x and are being sent to processors s at the same corresponding locations in the global array y. $Signal$ is an indicator which controls the flow of the loops in that, if the length of any row becomes equal to m, we set $Signal$ to $Stop$, indicating that all irreducibles have been found (see Theorem 1); else, we set it equal to OK in which case all loops continue to operate. $queue\_length$ always designates the number of polynomials to be computed in a new parallel queue, and $total\_poly$ designates the total number of non-constant factors determined during any stage of the parallel algorithm. Our parallel algorithm now takes the following form:

Algorithm 2 $Parallel\_G\ddot{o}ttfert(f, d, m, \{\mathbf{h_0}, ...\mathbf{h_{m-1}}\}, p, id)$

Input: f a polynomial of degree d over F_2, $m > 1$ the number of irreducible factors of f, $\{\mathbf{h_0}, ..., \mathbf{h_{m-1}}\}$ a basis for the solution space of (1), p the total number of processors operating in parallel, and id the processor identification number ranging from $0, ..., p-1$.

Output: the m irreducible factors of f and their multiplicities in f.

1. $Signal \leftarrow OK$, $k \leftarrow id$;

While $(k < m)$ **do**

2. $b_k \leftarrow f/\gcd(f, \mathbf{h_k})$, $degree \leftarrow \deg b_k$;

for $y \in \{0, ..., p-1\}$ **do**

3. $b_k \leftarrow BSP\_Put(y, degree, b_k)$;

end;

4. $k \leftarrow k + p$;

end.

5. $BSP\_synchronize()$.

6. $P_0 \leftarrow \gcd(b_0, b_1)$;

if $(P_0 \neq 1)$ **do**

7. $Poly_0 \leftarrow P_0$, $D[2][1] \leftarrow \&Poly_0$, $total\_poly \leftarrow 1$, $Length[2] \leftarrow 1$;

else do

8. $total\_poly \leftarrow 0$;

end;

9. $queue_j \leftarrow \{P_0\}$, $Set\_Queues(queue_j, queue_{j'})$, $queue\_length \leftarrow \#queue_{j'}$;

While $(Signal = OK)$ **do**

10. $k \leftarrow id$, $Set\_to\_zero(sum, local\_sums)$;

While $(k < queue\_length)$ **do**

11. $Type \leftarrow Type_{j'}[k]$, $n \leftarrow Row_{j'}[k]$, $i \leftarrow Index_{j'}[k]$,

$P_k \leftarrow Compute\_Polynomial(Type, n, i)$;

if $(P_k \neq 1)$ **do**

12. $Poly_{(k+total\_poly)} \leftarrow P_k$;

if $(T = D\_type)$ **do**

13. $D[n][i] \leftarrow \&Poly_{(k+total\_poly)}$;

else do

14. $R[n][i] \leftarrow \&Poly_{(k+total\_poly)}$;

end;

15. $sum[n] \leftarrow sum[n] + 1$, $degree \leftarrow \deg(Poly_k)$;

for $y \in \{0, ..., p-1\}$ **do**

16. $Poly_{(k+total\_poly)} \leftarrow BSP\_Put(y, degree, Poly_{(k+total\_poly)})$;

end;

if $(Type = D\_type)$ **do**

for $y \in \{0, ..., p-1\}$ **do**

17. $D[n][i] \leftarrow BSP\_Put(y, 0, D[n][i])$;

```
                    end;
                else do
                    for y ∈ {0, ..., p − 1} do
18.                     R[n][i] ← BSP_Put(y, 0, R[n][i]);
                    end;
                end;
            end;
19.         k ← k + p;
        end;
        for y ∈ {0, ..., p − 1} do
            for w ∈ {2, ..., m} do
20.             local_sums[y][w] ← BSP_Put(y, 0, sum[w]);
            end;
        end;
21.     BSP_synchronize();
        for (y ∈ {0, ..., p − 1}) do
            for (w ∈ {2, ..., m}) do
22.             Length[w] ← Length[w] + local_sums[y][w],
                total_poly ← total_poly + local_sums[y][w];
                if (Length[w] = m) do
23.                 Signal ← Stop, last_row ← w;
                end;
            end;
        end;
        if (Signal = OK) do;
24.         queue_j ← queue_j', queue_length ← Sort(queue_j), queue_j' ← (),
            Set_Queues(queue_j, queue_j');
        end;
end;
25. i ← id + 1;
While (i ≤ 2^(last_row−1)) do
        if (D[last_row][i] ≠ NULL) do
26.         factor ← *D[last_row][i], exp ← Multiplicity(f, factor),
            return (factor, exp);
        end;
        if (R[last_row][i] ≠ NULL) do
27.         factor ← *R[last_row][i], exp ← Multiplicity(f, factor),
            return (factor, exp);
28.     i ← i + p;
end.
```

The algorithm is called by all processors which implement the same copy of it for various data (conforming to the SPMD model described above). In step 1 of the algorithm, few initializations are set. The first **While** loop is a parallel loop met by all processors which compute the square-free factors $b_k = f/\gcd(f, \mathbf{h_k})$, for $k = 0, ..., m-1$. k is a global variable which when first set to id and then incremented by p guarantees that all processors compute almost an equal number of polynomials b_k. This constitutes the first parallel queue according to Algorithm 1. Every processor then broadcasts its own value of b_k to all other processors, but no synchronization barrier is met until all the b_k's are computed, since they are not needed in any loop computation. A synchronization point in the loop as such would only incur an extra cost of synchronization wihtout actually being required.

The second parallel queue consists of the polynomial $2; D_1$ (see Algorithm 1) which is computed by all processors. Although this constitutes a sequential step, the processors start to engage in distinct computations soon after the second queue is set up. If $P_0 = 2; D_1$ is not trivial, it is stored in a permanent location in $Poly_0$, $D[2][1]$ is set to point to the location of $Poly_0$ (which we denote by $\&Poly_0$), and the length of row 2 and the total number of non-constant factors computed so far are updated. We set the second queue $queue_j = \{P_0\}$, and we call Algorithm 1 to set up the ensuing $queue_{j'}$ of polynomials to be computed in parallel. $queue\_length$ denotes $\#queue_{j'}$.

Thereafter, the main loop of the algorithm is iterated so long as $Signal$ is not set to $Stop$ (indicating that none of the rows has attained m non-constant polynomials). The global variable k loops over indices in $queue_{j'}$. Each processor's copy of k is intialized to its id and is incremented by p so that processors compute almost an equal number of polynomials P_k in $queue_{j'}$. The processors receive information about the polynomials they should compute through the global data found in $Type = Type_{j'}[k]$, $n = Row_{j'}[k]$, and $i = Index_{j'}[k]$, and call the sub-routine $Compute\_Poly$ which determines the polynomial P_k as defined in the Göttfert setting. If P_k is non-constant, processor id stores it permanently in $Poly_{(k+total\_poly)}$, and sets $D[n][i]$ (or $R[n][i]$) to point to the address of $Poly_{(k+total\_poly)}$. Because $total\_poly$ represents the total number of non-constant factors computed so far, this particular index of $Poly$ is such that all new polynomials do not overwrite previous ones, and no two processors store their results in the same location. The local value of the polynomial and its pointer are then broadcasted by processor id to all processors, and the total number of non-constant factors found along row n by processor id during the set up of $queue_{j'}$ is increased by 1 in the processor's local copy of $sum[n]$ (it is assumed that sum and $local\_sums$ are initalized to zero before every new iteration of the main loop of step 10). When all polynomials in $queue_{j'}$ have been computed, each processor id places its own copies of $sum[n]$, for $n = 2, ..., m$ in global locations at $local\_sums[id][n]$. A synchronization barrier is now met, which updates the values of the non-constant polynomials, their pointers, and the partial lengths of rows as computed by every individual processor. We note the absence of a synchronization point immediately after the broadcasting of $Poly_{(k+total\_poly)}$ and the pointer to it, due to the fact that they were not needed in any computation within the loop of step 11. We also note that, although updating the total row lengths inside the loop of step 11 (i.e. while processors are still operating within the same parallel queue) definitely discards any unnecessary gcd or division operations remaining in the queue, our choice not to perform accordingly can be justified by the fact that this will require a synchronization point within the innermost loop, one whose repeated application could prove to be costly. Each processor now has all the partial sums available to it globally in $local\_sums[id][n]$ and can thus sum them all up into one global quantity in $Length[n]$. The total number of non-constant factors is also updated as being the sum of all row lengths. If any row length becomes equal to m, all processors are signalled to stop. Else, $queue_{j'}$ is transferred onto $queue_j$ (so that the most recent polynomials can help determine what the new parallel queue will be), and $queue_j$ is sorted through a call to $Sort$. Since some processors compute constant polynomials whose index k leaves the corresponding location in the array $Poly$ empty, the $Sort$ subroutine re-arranges the elements stored in $Poly$ (and their corresponding pointers in the arrays D or R) so that the non-constant factors are stored consecutively after each other. $Sort$ also returns the length of the sorted list. Finally, a new $queue_{j'}$ is set according to Algorithm 1. The loop of step 10 can be shown to end, since we are bound to reach a row containing m non-constant polynomials which constitute all the irreducible factors of f (Theorem 3). At this point, $last\_row$ contains the index to that row. All processors scan in parallel the non-constant D and R pointers to the polynomials found along $last\_row$ (using our notation, the polynomials are accessed by applying $*$ to a particular location in the D or R arrays). By Claim 1, there is a maximum of $\#r_{last\_row-1} = (2^{last\_row-1} - 1)$ D-polynomials and $\#r_{last\_row-1} + 1 = 2^{last\_row-1}$ R-polynomials in $last\_row$, which by Theorems 1 and 2 constitute the m non-constant factors of f. Each processor then determines the multiplicity of that factor in f (by a call to the sub-routine $Multiplicity$). At this stage, we can choose not to distribute the results globally so that each processor outputs its own set of $(factor, exp)$ pairs. The algorithm terminates with the last iteration of this loop.

4.2 The BSP Cost of the Algorithm

In this section we establish the parallel complexity of our algorithm. To this end, we first state and prove several preliminary results.

Theorem 4 *In the parallel setting described in Algorithm 1, every row n has its first element $n;D_1$ computed in the parallel queue n and its last element $n;R_{\#r_{n-1}+1}$ computed in the parallel queue $2n-1$.*

Proof:
We prove the result by induction on n. For $n = 2$, we know that queue 2 starts with $2;D_1$, and by Algorithm 1, queue 3 contains the polynomials $3;D_1$, $2;R_1$ and $2;R_2$, where $2;R_2$ is the last polynomial to be computed in row 2. Suppose that $n;D_1$ can be first computed in queue n. Since $(n+1);D_1 = \gcd(b_{n+1}, n;D_1)$, the first queue which assigns the computation of $(n+1);D_1$ is $n+1$. Furthermore, suppose that queue $2n-1$ contains the polynomial $n;R_{\#r_{n-1}+1}$ which is computed last in row n and is determined at the latest when $(n+1);D_{\#r_n}$ is computed. Since

$$(n+1);D_{\#r_n} = \gcd(b_{n+1}, n;R_{\#r_{n-1}+1}),$$

this polynomial can be determined at the earliest in the parallel queue $2n$, so that $(n+1);R_{\#r_n+1}$ can be at computed at the earliest in the parallel queue $2n+1$. ☐

Corollary 1 *It takes at most $2m - 1$ parallel queues for a complete factorization into irreducibles to be established.*

Proof:
By Theorem 2 it takes at most m rows to compute all irreducible factors of f (see [8] for proof). By Theorem 4, row m requires at most $2m - 1$ parallel queues before all non-constant polynomials appearing in it are computed. This concludes the proof. $\square$

Theorem 5 *If n is odd, then queue n contains polynomials belonging only to rows $(n+1)/2 + j$, for $j = 0, ..., (n-1)/2$, if $2 \leq n \leq m$, and for $j = 0, ..., m - (n+1)/2$, if $m < n \leq 2m - 1$. Else, if n is even, then queue n contains polynomials belonging only to rows $n/2 + 1 + j$, for $j = 0, ..., n/2 - 1$, if $2 \leq n \leq m$, and for $j = 0, ..., m - (n/2 + 1)$, if $m < n \leq 2m - 1$.*

Proof:
For all queues n appearing in the parallel set-up, Corollary 1 maintains that $1 \leq n \leq 2m-1$. Suppose now that n is odd. Let k be a row occupying queue n. By Theorem 4, we must have $n \leq 2k - 1$ (or $k \geq (n+1)/2$). Write $k = (n+1)/2 + j$ for $j \geq 0$ (since n is odd, this expression is an integer). For $n = 2, ..., 2m - 1$, we must have $k \leq n$ (so that $j \leq (n-1)/2$); otherwise, row k starts appearing in queues $n + 1$ onwards, a contradiction. If $m < n \leq 2m - 1$, the upper bound on k can be strengthened to satisfy $k \leq m$ (or $j \leq m - (n+1)/2$), since the last row to be set up in the Göttfert representation is row m.

If n is even, $n \leq 2k-1$ implies that $n \leq 2k-2$ since $2k-1$ is odd, or that $k \geq (n/2)+1$. Write $k = n/2 + 1 + j$ for $j \geq 0$ (again, this expression is an integer since n is even). A similar argument as above follows to establish the upper bounds on j, and we leave the straightforward proof to the reader. $\square$

Theorem 6 *Each parallel queue consists of at most $2m$ gcd and division operations and contributes to at most m non-constant polynomials.*

Proof:
First, we note that, since each row k in the Göttfert representation requires at most $2m$ gcd and division computations and has at most m non-constant polynomials appearing in it, and since each such row requires a number of at most k queues to be fully set up, we would expect, roughly and on average, each parallel queue n to require a number of $2m/k$ gcd and division operations leading to about m/k non-constant polynomials for each row k assigned to queue n. Let n be odd (the case when n is even can be proven similarly and hence can be omitted). If $2 \leq n \leq m$, Theorem 5 implies that queue n has polynomials belonging to rows $k = (n+1)/2 + j$, for $j = 0, ..., (n-1)/2$, where each row k has roughly m/k non-constant polynomials appearing in queue n. Thus, the total number of gcd and division operations to be performed in queue n is approximately

$$\sum_{j=0}^{(n-1)/2} \frac{2m}{\frac{n+1}{2} + j} \lessapprox \left(\frac{n-1}{2} + 1 \right) \frac{2m}{(n+1)/2} = 2m.$$

Furthermore, if $m < n \leq 2m - 1$, this number is approximately

$$\sum_{j=0}^{m-(n+1)/2} \frac{2m}{\frac{n+1}{2}+j} \lessapprox \left(m - \frac{n+1}{2} + 1\right) \frac{2m}{(n+1)/2}$$
$$< (m - \frac{m+1}{2} + 1)\frac{2m}{(m+1)/2} = 2m.$$

Using a very similar calculation, the total number of non-constant polynomials produced is easily shown to be at most m. $\square$

Theorem 7 *The BSP cost of Algorithm 2 is of the order*

$$O\left(\frac{m^2}{p}M(d)\log d + gm^2\frac{d}{w_l} + m\ell\right) \textit{flops.} \tag{2}$$

Proof:
In our proof, we note the following remarks. Since all polynomials appearing in the course of the algorithm are factors of f, their degrees are at most equal to $d = \deg(f)$. It is also understood that any computational complexity is the maximum work load achieved by any one processor. Sorting a list of size at most k can be achieved in $O(k \log k)$ flops (e.g. see [3]). Computing the multiplicity of a factor of f requires at most d polynomial multiplications over F_2 (and hence in our case, $M(d)$ flops). We also assume that accessing an array entry requires almost as much time as one flop. A *message* denotes one computer word (i.e. a message of size 1). The total cost of the algorithm is the summation of the costs of its supersteps. The individual BSP costs of the main supersteps (i.e. those whose cost is not constant) can be detailed as follows:

Supersteps 2-4: we have seen earlier that this loop is divided almost equally among all processors. As a result, the parallel loop is accessed at most $\lceil m/p \rceil$ times. Each iteration of the outer loop involves mainly one gcd and one division computation, and an inner loop consisting of a *bsp_put* operation, whereby each processor sends $\lceil degree/w_l \rceil \leq \lceil d/w_l \rceil$ messages to all processors (and hence p copies of these) and receives $degree \leq d$ messages. Thus, $h_{max} = p.\lceil degree/w_l \rceil < p.\lceil d/w_l \rceil$. Superstep 5 is a synchronization point, so that the total BSP cost of supersteps 2-4 is at most

$$O\left(\left\lceil \frac{m}{p} \right\rceil \left[M(d)\log d + g.p\frac{d}{w_l}\right] + \ell\right) \text{ flops.} \tag{3}$$

Supersteps 6-9: consist mainly of one gcd operation and a call to *Set_Queues* with input $queue_j = \{P_0\}$, which according to Theorem 3 requires at most 3 steps. Thus, the BSP cost of these supersteps is of the order

$$O(M(d)\log(d)) \text{ flops.} \tag{4}$$

Supersteps 10-28: constitute the main body of the parallel algorithm. The outer–most loop designates the total number of times the processors set up parallel queues before a complete factorization is achieved. This number has been shown in Corollary 1 to be at most $2m-1$. Supersteps 11-19 are embedded within an inner loop ranging over all polynomials along $queue_{j'}$. The tasks within the queue, consisting mainly of gcd and division operations, are divided almost equally among processors (check the initial value of the loop variable k and the increment it receives). By Theorem 6, each queue consists of at most $2m$ gcd and division operations to perform equally among all processors, and hence the loop is accessed at most $\lceil 2m/p \rceil$ times. The bulk of the work appears in the following: step 11 consists of either a gcd or a division operation, step 16 consists of sending at most $p.\lceil d/w_l \rceil$ messages and receiving at most $\lceil d/w_l \rceil$ messages (resulting in $h_{max} = p.\lceil d/p \rceil$), and step 17 or 18 consists of sending p messages and receiving one message (resulting in $h_{max} = p$). Within each inner loop iteration, the BSP cost of supersteps 11-19 is at most

$$O\left(M(d)\log(d) + gp\left\lceil \frac{d}{w_l} \right\rceil\right)$$

which across both the outer and inner loops becomes of the order

$$O\left(m\left\lceil \frac{m}{p} \right\rceil \left[M(d)\log(d) + gp\left\lceil \frac{d}{w_l} \right\rceil\right]\right) \text{ flops.} \tag{5}$$

Supersteps 20-28 are found outside the inner loop of step 11 but inside the loop of step 10. In the communication superstep 20 each processor sends pm messages and receives m messages (so that $h_{max} = pm$). Superstep 21 is a synchronization point, and superstep 22 consists of about $2pm$ additions. Superstep 24 consists mainly of a call to $Sort$ with input list of size at most $2m$ (producing a list of size at most m by Theorem 6), as well as a call to Set_Queues with input list of size at most m (this requiring at most $3m$ flops). Summing up, a single application of supersteps 20-28 requires at most

$$O\left(m(p + \log m) + gpm + \ell\right)$$

flops which when iterated across the outer loop of step 10 becomes of the order

$$O(m^2(p + \log m) + gpm^2 + m\ell) \text{ flops.} \tag{6}$$

Supersteps 25-28 consist of computing the multiplicity of all m factors in parallel, where each processor takes up almost an equal number of D and R-factors. If each call to $Multiplicity$ requires about $M(d)$ flops, the supersteps will require

$$O\left(\left\lceil \frac{m}{p} \right\rceil M(d)\right) \text{ flops.} \tag{7}$$

Summing up the individual costs 3, 4, 5, 6, and 7, the total BSP cost can be found to be of the order

$$O\left(m\left\lceil \frac{m}{p} \right\rceil M(d)\log d + m^2(p + \log m) + g.p.(m\left\lceil \frac{d}{w_l} \right\rceil \left\lceil \frac{m}{p} \right\rceil + m^2) + m\ell\right)$$

flops. Since

$$\left\lceil \frac{m}{p} \right\rceil < \frac{m}{p} + 1, p << d, M(d) = d^2 \text{ and } m \approx \log d \text{ [14]},$$

(where the second inequality easily holds in implementations involving large polynomial degrees, the third estimate holds in our implementations of classical polynomial arithmetic [7], and the last estimate is an average for the number of irreducible factors of a polynomial of degree d [14]), it follows that $m^2(p + \log m) << m^2 d$ and $m^2 p < M(d) \log d$, from which the total BSP cost (2) can be derived. □

Corollary 2 *Algorithm 2 has low synchronization and communication requirements.*

Proof:
We claim that our algorithm has very good synchronization and communication requirements, in that the number of flops required by both can be negligible compared to the computation cost. In particular, and based on the values of the BSP parameters in table 3, it can be easily attained that

$$g << \frac{d \log d}{p} \text{ and } \ell << \frac{m}{p} M(d) \log d$$

for large values of d, so that

$$gm^2 \left\lceil \frac{d}{w_l} \right\rceil + m\ell << \frac{m^2}{p} M(d) \log d.$$

□

4.3 Reduction of the Memory Requirements of the Algorithm

As defined in Algorithm 2, the two double arrays D and R of size $m \times (2^m - 1)$ each can be easily seen to constitute an infeasible (exponential) space requirement unless m is very small. To this end, we describe how a linked-list structure can be adopted which reduces the memory requirements to four integer arrays of size m each and four integer arrays of size $2m^2$ each, requiring only a polynomial order space complexity.

Recall that, for $n = 2, ..., m$ and $i = 1, ..., 2^m - 1$, $D[n][i]$ represents a pointer which is NULL if $n; D_i$ is a constant polynomial, and which points to the location of $n; D_i$ in the array *Poly*, otherwise. Similarly for $R[n][i]$. We also note that there are many more constant polynomials than there are non-constant ones, and hence, the distribution of non-constant polynomials among all possibilities is a sparse one. Our algorithm in its present form contains many NULL pointers, and a candidate for a more efficient method has to replace this structure with one which locates only non-constant polynomials without any reference to the others.

The improvement can be described as follows. We first order the non-constant factors in a list $\mathcal{F}$, in which each factor occupies a index corresponding to its position in the array $Poly$. For instance, if a non-constant factor is stored at location k in $Poly$, it would appear as the $k+1$ polynomial in $\mathcal{F}$. In this way, a non-constant factor in the entire collection can be compared to a non-zero entry in a collection of sparse row vectors (or columns) representing a sparse matrix.

We define four global integer arrays, D_header, D_tail, R_header and R_tail, each of size m, and four global integer arrays, D_fwd_link, R_fwd_link, D_index and R_index, each of size $2m^2$. The aim would be to arrange polynomials in $\mathcal{F}$ in lists of polynomials of the same type and row. The description below concerns only D-arrays but a very similar one holds for R-arrays. We leave this for the reader to verify.

Suppose we want to link all D- polynomials in $\mathcal{F}$ appearing in row n. For $n = 2, ..., m$, $D\_header[n]$ represents the index in $\mathcal{F}$ of the first (non-constant) D-polynomial appearing in row n. All ensuing non-constant D-polynomials in row n are adjoined to the list, and as such, we need to preserve various information describing each one of them as they appear together. For each row n, we keep track of the index in $\mathcal{F}$ of the last D-polynomial appearing in the row by storing it in $D\_tail[n]$. Thus, for a start, if one non-constant D-polynomial of row n appears in $\mathcal{F}$ at location k, we set $D\_header[n] = k$ and $D\_tail[n] = k$. We also store its index along row n in $D\_index[k]$, which completes all information about the polynomial. For each non-constant D-polynomial in row n and of index k in $\mathcal{F}$, we maintain a pointer to the index in $\mathcal{F}$ of the next such polynomial and store it in $D\_fwd\_link[k]$. If $D\_fwd\_link[k] = 0$, polynomial k is the last one in the list of non-constant D-polynomials belonging to row n, which can be generated completely as follows:

Algorithm 3 *Input: A non-constant D-polynomial of index i in some row $n = 2, ..., m$ and index k in $\mathcal{F}$.*
Output: The polynomial adjoined to the end of the list of non-constant D-polynomials belonging to row n.
if $(D\_header[n] = 0)$ **do**
1. $D\_header[n] \leftarrow k$, $D\_tail[n] \leftarrow k$, $D\_index[k] \leftarrow i$,
$D\_fwd\_link[k] \leftarrow 0$;
else do
2. $D\_fwd\_link[D\_tail[n]] \leftarrow k$, $D\_tail[n] \leftarrow k$, $D\_index[k] \leftarrow i$;
end.

It is trivial to see that the call to this algorithm comes at a very negligible constant cost and hence can be embedded within the estimate of (2). We now illustrate the use of this structure in locating non-constant factors whenever required for new computations. Suppose, for instance, that we need to compute $n; D_i$ for some $n = 3, .., m$ and $i = 1, ..., (\#r_{n-1} - 1)/2$ (so that $n; D_i = \gcd(b_n, (n-1); D_i)$). Using the double array D, and if $D[n-1][i]$ is NULL, one concludes that $n; D_i$ is constant; otherwise, the polynomial pointed to by $D[n-1][i]$ is used to compute the required gcd. Using the linked list structure, the process can be described as follows:

Algorithm 4 *Input: A D-polynomial of index i in some row $n-1$, $n = 3, ..., m$.*
Output: The location of $(n-1); D_i$ in the array Poly if it is non-constant or FAIL

otherwise.
1. $t \leftarrow FAIL$, $x \leftarrow D\_header[n-1]$;
if $(D\_header[n-1] \neq D\_tail[n-1])$ **do**
    While $(x \neq 0$ *and* $t = FAIL)$ **do**
        if $(D\_index[x] = i)$ **do**
2. $t \leftarrow PASS$;
3. $k \leftarrow x$, $x \leftarrow D\_fwd\_link[x]$;
        end;
    end;
end;
else do
    if $((D\_index[x] = i)$ **do**
4. $t \leftarrow PASS$;
    end;
5. $k \leftarrow x$;
end;
if $(t = PASS)$ **return** $k-1$;
else return $FAIL$.

If $(D\_header[n-1] \neq D\_tail[n-1])$, the list of D-polynomials belonging to row n contains more than one polynomial and hence has to be scanned entirely; else, the list contains only one element. x takes up indices in this list, starting with its header, and moving across the forward links. t is an indicator which when set to $PASS$ indicates that a non-constant polynomial $(n-1); D_i$ has been found whose location in $\mathcal{F}$ is k (or location in $Poly$ is $k-1$). If x becomes zero, this signals the end of the list. Finally, the loop across the list ends either when t becomes PASS or $x = 0$. The algorithm demonstrates how checking $D[n][i]$ can be substituted with scanning a list of size at most m. The increase in the total computational cost as a result of using the improved data structre can be realized as follows. We have seen that the call to *Compute_Polynomial* in step 11 of algorithm 2 is issued at most $2m \cdot \lceil m/p \rceil$ times. Each such polynomial computation will require two calls to algorithms 3 and 4, thus increasing the total cost of Algorithm 2 by $O(2m^2 \cdot \lceil m/p \rceil)$ flops. Since $m << M(d) \log d$, the upper bound estimate given in (2) remains of the same order, and as such, the new improvement can be introduced at little cost to the worst case analysis initially provided.

5 Implementation and Run Times

All programs were written in C and extended using the standard BSP library [12], [13]. The work was carried out at the Oxford University Supercomputing Centre (OSC) using the Oswell machine. Oswell is a Sun cluster of 84 processors and a shared memory system where the processors are arranged in three groups of 24 processors (having 48 GBytes of memory each) and a group of 12 processors (having 24 GBytes of memory each). Each of the UltraSPARC III processors is capable of performing an average of 122.2 Mflop/s. With this scheme, any work submitted onto the machine is queued onto one of those

Table 1: Parallel run times.

				Processors				
d	m	max	seq.	1	2	4	8	16
8000	6	7446	2.37	2.32 (1)	1 (1.2)	0.8 (0.7)	0.2 (1.5)	0.6 (0.2)
8000	7	3200	2.62	2.6 (1)	0.9 (1.5)	0.7 (0.9)	1.4 (0.2)	1.1 (0.1)
8000	30	600	48.98	48.36 (1)	27 (0.9)	17.4 (0.7)	6.8 (0.9)	6.6 (0.5)
16000	10	10224	29.9	29.9 (1)	14.2 (1.1)	10.4 (0.7)	10.4 (0.4)	3.7 (0.5)
16000	14	5600	8.78	8.74 (1)	4.7 (0.9)	2.4 (0.9)	2.6 (0.4)	1.4 (0.4)
32000	31	19360	145.8	144.89 (1)	76.4 (1)	31.5 (1.2)	13.8 (1.3)	10.4 (0.9)
64000	10	26400	82.6	82.01 (1)	64.2 (0.6)	29.3 (0.7)	27 (0.4)	29.05 (0.2)
128000	14	44800	2629.62	2609 (1)	794.8 (1.7)	761.4 (0.9)	760.6 (0.4)	468.8 (0.4)
256000	14	92800	6819.12	6761.89 (1)	2696.3 (1.3)	1420.4 (1.2)	1417 (0.6)	325.1 (1.3)
300000	11	120000	621.8	621.32 (1)	408.68 (0.8)	242.2 (0.6)	51.2 (1.5)	57.6 (0.7)

Table 2: Worst case analysis BSP estimates.

				Processors				
d	m	max	seq.	1	2	4	8	16
8000	6	7446	244.5	244.5	122.3	61.1	30.6	15.3
8000	7	3200	332.8	332.8	166.4	83.2	41.6	20.8
8000	30	600	23346	23346	11673	5836.5	2918.3	1495.1
16000	10	10224	2926.6	2926.6	1463.3	731.7	365.8	182.9
16000	14	5600	5736.2	5736.2	2868.1	1434.1	717	358.5
32000	31	19360	120551.9	120551.9	60276	30138	15069	7534.5
64000	10	26400	53529.7	53529.7	26764.9	13382.4	6691.2	3345.6
128000	14	44800	445950.5	445950.5	222975.3	111487.7	55743.9	27871.9
256000	14	92800	1888917	1888917	944458.5	472229.6	236114.8	118057.4
300000	11	120000	1621914.9	1621914.9	810957.5	405478.9	202739.5	101369.7

four boxes, and the number of processors available for use by any one job is at most 24. The various input data were taken from the results of [1] where the Niederreiter linear system for a trinomial over F_2 is solved and a basis for the solution set is produced. We remark that, in spite of the input polynomial being sparse, the intermediary polynomials in the Göttfert representation are not necessarily so, which renders such a case study for trinomials not so special a case as it appears. The run times in table 1 represent the times in seconds for producing the entire factorization using our BSP parallel algorithm given a particular basis set. The timings correspond to the sequential as well as the

Table 3: BSP parameters.

p	g	ℓ
1	0.34	55
2	1.6	1496
4	2.475	1683
8	2.34	2562
16	4.83	3431

parallel results. max represents the maximum over all $i = 1, ..., m$ of $deg(g_i^{e_i})$, and d and m are as defined previously. The absolute efficiencies are shown in parentheses. All polynomial arithmetic was performed using classical algorithms so that the multiplication time for polynomials of degree at most d is $O(d^2)$ [7]. In tables 2 and 3, we present the theoretical bounds obtained using our BSP cost estimate as well as the machine values of the BSP parameters[2], which are in flops for ℓ and in flops/word for g. We stress that our theoretical bounds are not close estimates but rather higher ones than the actual timing results. This is a result of the fact that our calculations as in the proof of Theorem 7 were a worst case analysis; in particular, a major component of our theoretical estimate for the BSP cost depended on $O(M(d)\log d)$ as a cost for polynomial gcd computations, when many gcd's computed in the parallel set-up might have very likely had input degrees that were considerably smaller than d.

Let T_p and T_s denote the parallel run time using p processors and the sequential run time respectively. Our run times suggest a speed gain in almost all cases, an outcome that is to be expected according to our BSP cost 2, which roughly suggests that

$$T_p < T_s \text{ iff } \frac{m^2}{p} < m^2 \text{ iff } p > 1,$$

ignoring the negligible communication and synchronization requirements of our algorithm as well as any other parallel overheads associated with the creation and management of threads. To measure the scalability of our parallel algorithm, we calculate the absolute efficiency E_p [10] for all cases, where

$$E_p = \frac{T_s}{p.T_p},$$

which when achieving values close to one indicates a good parallel performance. Accordingly, and upon examining our efficiencies, we note that almost all our experiments scale very well for up to 8 processors. Thereafter, the efficiency remains very good for a fixed d either as m increases (e.g. compare trinomials with $m > 30$ to others), or as max increases (for $d = 8000$, compare the trinomials with $max = 7446$ and $max = 3200$). We also note that efficiency remains almost constant around 1 for $d \geq 256000$. We remark the absence of a sharp fluctuation in the efficiency levels mainly because our algorithm does

[2]See http://www.bsp-worldwide.org/implmnts/oxtool/params.html for the benchmark tests.

not involve data partitioning (but only task parallelism), which results in the computation being either entirely *in cache* or *out of cache* across all processors for the same d. This has the advantage of revealing the real scalability of the algorithm and avoiding cache effects. With results as such we expect our algorithm to continue scaling well as d increases more considerably than what is actually reported in this paper, and experiments related to the output of the algorithm in [1] for solving large Niederreiter linear systems for trinomials over F_2 are currently under-way.

6 Conclusion

In this paper we presented and analyzed a complete BSP algorithm for extracting the factors of a polynomial over F_2 using the Göttfert refinement of the Niederreiter algorithm, which, given a basis for the solution set of the Niederreiter linear system, performs the last phase of the factorization algorithm in polynomial time. Our BSP theoretical model resulted in an efficient BSP cost requiring relatively small communication and synchronization costs. The parallel algorithm not only achieves considerable speed gains as the number of processors increases up to 16, but maintains a moderate to very good efficiency that is better maintained as the degree of the polynomial, the number of its factors or the maximum over its irreducible factors' degrees increases. The algorithm can be applied over fields of characteristic 2 in general, provided an input basis is available. When combined with our work in [1] which exploits sparsity in the Niederreiter linear system, the hybrid algorithm provides a cheaper and more memory efficient alternative to the factorization of trinomials over F_2 than the implementation in [20], which uses dense explicit linear algebra and a maximum of 256 nodes to achieve a polynomial record of degree 300000. When compared with the Black Box Niederreiter algorithm of [6], the hybrid algorithm is a simpler approach that promises high record factorizations of trinomials over F_2 as those allowed by the use of implicit linear algebra, requiring reasonable running times [1]. Apart from the significance of its experimental results, our algorithm provides a good model of how parallelism in general, and the BSP model in particular, can be incorporated elegantly and successfully into problems in symbolic computation.

Acknowledgements

The author is grateful to the Oxford Supercomputing Centre for allowing the use of its facilities to generate the reported experiments.

References

[1] F. Abu Salem, "A New Sparse Gaussian Elimination Algorithm and the Niederreiter Linear System for Trinomials over F_2", *to appear*.

[2] R. H. Bisseling and W. F. McColl, "Scientific Computing on Bulk Synchronous Parallel Architectures", *Proc. IFIP 13th World Computer Congress*, vol. I, North-Holland, 1994, pp. 509-514.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[4] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping, "The Paderborn University BSP (PUB) library - design, implementation, and performance", 13^{th} *International Parallel Processing Symposium* & 10^{th} *Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San Juan, Puerto Rico, 1999.

[5] J. Della Dora and J. Fitch, *Computer Algebra and Parallelism*, Academic Press, London, 1989.

[6] P. Fleischmann, M. Holder, and P. Roelse, "The Black-Box Niederreiter Algorithm and its Implementation over the Binary Field", *Mathematics of Computation*, vol. 72, 2003.

[7] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, Cambridge, 1999.

[8] R. Göttfert, "An Acceleration of the Niederreiter Factorization Algorithm in Characteristic 2", *Mathematics of Computation*, vol. 62, no. 206, 1994, pp. 831-839.

[9] M. A. Inda and R. H. Bisseling, "A Simple and Efficient Parallel FFT Algorithm Using the BSP Model", *Parallel Computing*, vol. 27, no. 14, 2001, pp. 1847-1878.

[10] VOL. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[11] T.C.Y. Lee and S. A. Vanstone, "Subspaces and polynomial factorizations over finite fields", *AAECC 6*, Springer-Verlag LNCS 357, 1995, pp. 147-157.

[12] J. M. D. Hill, W. F. McColl, and D. B. Skillicorn, "Questions and Answers about BSP", *Report PRG-TR-15-96*, Oxford University Computing Laboratory, 1996.

[13] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudrea, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, R. H. Bisseling, "BSPlib: The BSP Programming Library", *Parallel Computing*, vol. 24, 1998, pp. 1947-1980.

[14] M. Mignotte, *Mathematics for Computer Algebra*, Springer-Verlag, New York, 1992.

[15] V.S. Miller, "On the Factorization Method of Niederreiter", IBM T. J. Watson Research Center, Yorktown Heights, N.Y., preprint, 1992.

[16] H. Niederreiter, "A New Efficient Factorization Algorithm for Polynomials over Small Finite Fields", *AAECC*, vol. 4, 1993, pp. 81-87.

[17] H. Niederreiter, "Factorization of Polynomials and Some Linear Algebra Problems over Finite Fields", *Linear Algebra and its Applications*, vol. 192, 1993, pp. 301-328.

[18] H. Niedrreiter, "Factoring Polynomials over Finite Fields Using Differential Equations and Normal Bases", *Mathematics of Computation*, vol. 62, 1994, pp. 819-830.

[19] H. Niederreiter and R. Göttfert, "Factorization of Polynomials over Finite Fields and Characteristic Sequences", *J. Symbolic Computation*, vol. 16, no. 5, 1993, pp. 401-412.

[20] P. Roelse, "Factoring High-Degree Polynomials over F_2 with Niederreiter's Algorithm on the IBM SP2", *Mathematics of Computation*, vol. 68, no. 226, 1999, pp. 869-880.

[21] W. Schreiner, "Parallelizing the Big Prime Berlekamp Algorithm with Distributed Maple", *Technical Report 02-07*, RISC-Linz, Johannes Kepler University, Linz, Austria, July 2001.

[22] L. G. Valiant, "A Bridging Model for Parallel Computation", *Communications of the ACM*, vol. 33, no. 8, 1990, pp. 103-111.

[23] P. S. Wang, "Symbolic Computation and Parallel Software", *Proceedings of First International Conference of the Austrian Center for Parallel Computation*, Springer-Verlag LNCS, Parallel Computation, vol. 591, pp. 316-337, 1991.