A BSP Parallel Model for the Göttfert Algorithm over F_2

Fatima Abu Salem

Oxford University Computing Laboratory,
Wolfson Building,
Parks Road,
OX1 3QD, UK
fkas@comlab.ox.ac.uk

Abstract. In Niederreiter'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.

1 Introduction

One important problem in symbolic computation is the factorization of univariate polynomials over finite fields, where the Niederreiter algorithm has been introduced [1]. A major feature of the algorithm consists in its linearization of the factorization problem, or reducing it to solving a linear system over the finite field in question, where the solutions lead to a complete factorization in a variety of ways. One such method was presented by Göttfert [2] for fields of characteristic 2, and is best featured as a simple and polynomial time algorithm for extracting the factors using only the basis elements of the solution set of the Niederreiter system. In this paper, we develop a new BSP parallel approach to the Göttfert algorithm over F_2. 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 Section 2, we give a brief survey of the algorithms underlying our work and some background information describing the BSP parallel model. In Section 3 we present our parallel algorithm and discuss its BSP cost analysis. In Section 4 we report on our experimental results and discuss the scalability of the algorithm.

R. Wyrzykowski et al. (Eds.): PPAM 2003, LNCS 3019, pp. 217–224, 2004.

2 Göttfert's Refinement of the Niederreiter Algorithm over F_2 and the BSP Model

For complete proofs and results related to the Niederreiter and Göttfert algorithms we refer the reader to [1], [2], [3], [4]. 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 ([1], [2]). The Niederreiter linear system can be expressed as $(N_f - I_d)h^T = 0$, where I_d is the $d \times d$ identity matrix over F_2 and $h = (h_0, ..., h_{d-1})$ is the coefficient row vector of an unknown polynomial h over F_2 of degree less than d. A crucial result establishes that there are 2^m solutions h of the system, and that any of them can eventually produce a complete factorization into irreducibles. In particular, the solutions h of the linear system form a linear subspace of $F_2[x]$ of dimension m over F_2. For full details relating to the Niederreiter algorithm, we refer the reader to [1], [3], [4]. Suppose that $m > 1$ so that the factorization is non-trivial. Several methods have been suggested to achieve a factorization using solutions of the sytem, but have the disadvantage of either requiring all 2^m solutions to be found or several linear systems to be solved before a complete factorization is established. In a new method, Göttfert showed that it is enough to use the m polynomials corresponding to any basis of the linear system, without having to solve other than the original linear system associated with f. To illustrate, let $\{h_1, ..., h_m\}$ be a basis spanning the solution set of the linear system. For $i = 1, ..., m$, the corresponding polynomials $b_i = f/\gcd(f, h_i)$ are square-free factors of f [1]. 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 $\gcd(b_2, b_1)$, $b_1/\gcd(b_2, b_1)$, and $b_2/\gcd(b_2, b_1)$. In general, the polynomials of row n, for $n = 3, ..., m$, consist of the non-constant polynomials among d_1, r_1/d_1, ..., d_s, r_s/d_s, $b_n/d_1...d_s$, 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 [2], 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 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 [2]:

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.*

It can also be shown that the total cost of this algorithm is at most $O(m^2 M(d) \log d)$ field operations, where $\log d$ is the binary logarithm of d, and

$M(d)$ is the arithmetic complexity to multiply (or divide) two polynomials of degree at most d over F_2.

The *bulk synchronous parallel model* (BSP) [5], [6], [7], is a model for 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. 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*. Communication supersteps are followed by *synchronization barriers*, whereby all transferred data is updated. Processors are distinguished by their own identification number, $id = 0, ..., p-1$. A BSP computer can be described by machine dependent parameters: s, the processor speed (in flop/sec), g, the time (in flop time units) it takes to communicate (send or receive) a data element, and ℓ, the time (in flop time units) it takes all processors to synchronize. The BSP cost is established using the parameters g and ℓ and the cost of an algorithm is simply the sum of the BSP costs of its supersteps. The estimate of the execution time is obtained by dividing the BSP cost in flop time units by s. 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 F_2.

3 A Parallel Approach to Göttfert's 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 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 $(n-1); D_i$, for some $i = 1, ..., (\#r_{n-1} - 1)/2$, and $(n-1); R_i$, for some $i = (\#r_{n-1} + 1)/2, ..., \#r_{n-1}$, then row n consists of

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

$$n; R_i = \begin{cases} (n-1); D_i / n; D_i, & \text{if } 1 \le i \le \frac{\#r_{n-1}-1}{2} \quad (3) \\ (n-1); R_{i-(\#r_{n-1}-1)/2} / n; D_i, & \text{if } \frac{\#r_{n-1}+1}{2} \le i \le \#r_{n-1} \quad (4) \end{cases}$$

and $n; R_{\#r_{n-1}+1} = b_n / \prod_{i=1}^{\#r_{n-1}} n; D_i$ for non-constant $n; D_i$. Furthermore, we assert the following (see [8] for proof):

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

The first step in our parallel approach consists of studying the dependencies between the gcd and division computations and structuring them in a parallel hierarchy. Without loss of generality we may assume that the number of threads coincides with the number of processors available. We define a parallel queue as one which consists of a list of polynomials that can be computed independently by a number of p processors using a number of parallel supersteps, such that the supersteps can be executed without requiring a synchronization point throughout the queue. The first parallel queue consists of the polynomials b_i, for $i = 1, ..., m$. 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: a list queue$_{k'}$ *of polynomials to be computed in the parallel queue* $k' > k$.
1. $queue_{k'} \leftarrow ()$*; 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: see [8].

3.1 The Parallel Göttfert Algorithm

One major characteristic of the algorithm is that it consists mainly 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. The polynomials are represented by integer arrays whose entries are either zero or one and where the coefficients are packed into bit-words (where w_l is the bit-size of the computer word being used). We refer the reader to our report in [8] for a detailed description of the data structures used. Unless otherwise stated, all arrays are global. The arrays $Type$, Row, and $Index$ are embedded within two queues $queue_j$ and $queue'_j$; $queue_j$ 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.

Algorithm 2 *Parallel_Göttfert*
Input: f a polynomial of degree d over F_2, $m > 1$ the number of irreducible factors of f, $\{h_0, ..., h_{m-1}\}$ a basis for the solution set of the linear system, $\{b_0, ..., b_{m-1}\}$ the corresponding set of squarefree factors of f defined by $b_i = f/\gcd(f, h_i)$ for $i = 0, ..., m-1$, 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.

```
1. P_0 ← gcd(b_0, b_1);
if (P_0 ≠ 1) do
2.    Store_value(P_0), Update(length(2));
end;
3. queue_j ← {P_0}, Set_Queues(queue_j, queue_j');
while not all m irreducible factors have been found do
4.    k ← id;
      while (k < queue_length) do
5.          P_k ← Compute_Polynomial(Type, row, index, k);
            if (P_k ≠ 1) do
6.                Store_value(P_k), n ← row(P_k),
                  Update(partial_length(n)), Broadcast_value(P_k),
                  Broadcast(partial_length(n));
7.          k ← k + p;
            end;
8.    BSP_synchronize();
9.    for i ∈ {2, ..., m} do
            Assemble(partial_lengths(i));
      end;
      if (not all irreducible factors have been found) do
10.         queue_j ← queue_j', Sort(queue_j),
            Set_Queues(queue_j, queue_j');
      end;
end.
```

For elaborate details describing the algorithm we refer the reader to [8]. In this paper, we give a brief description of the main steps comprising the process above. The algorithm is called by all processors which implement the same copy

of it for various data . The second parallel queue consists of the polynomial $2; D_1$ (see Algorithm 1) which is computed by all processors. If $P_0 = 2; D_1$ is not trivial, it is stored permanently, and the length of row 2 is updated. We call Algorithm 1 to set up the ensuing $queue_{j'}$ of polynomials to be computed in parallel. Thereafter, the main loop of the algorithm is iterated so long as none of the rows has attained m non-constant polynomials. The variable k, which loops over indices in $queue_{j'}$, 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. 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 embedded within $queue_{j'}$. They then call the sub-routine $Compute\_Polynomial$ which determines the polynomial P_k as defined in the Göttfert setting. If P_k is non-constant, processor id stores it permanently and updates its own local value of $length(n)$. When all computations in $queue_{j'}$ have been performed, a synchronization barrier is met, which updates the values of the non-constant polynomials 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 the non-constant polynomials due to the fact that they were not needed in any computation within the innermost loop of the algorithm. Also, although updating the total row lengths inside the innermost loop definitely discards any unnecessary gcd or division computations 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 expensive. All processors then assemble the partial lengths of all rows as computed by the relevant processors which have contributed in non-constant polynomials. 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 memory empty, the $Sort$ sub-routine re-arranges them (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 outermost loop can be shown to end, since we are bound to reach a row containing all m non-constant irreducible factors.

3.2 The BSP Cost of the Algorithm

Before discussing the parallel complexity of our algorithm, we derive several preliminary results, whose complete proofs can be found in [8].

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 $2n - 1$.*

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

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* $m<n\leq 2m-1$*.*

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

Theorem 7. *The BSP cost of algorithm 2 is of the order* $O\left(\frac{m^2}{p}M(d)\log d + gm^2\left\lceil\frac{d}{w_l}\right\rceil + m\ell\right)$ *flops.*

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

4 Implementation and Run Times

For a full report on our parallel performance the reader can refer to [8]. Our run times in table 1 of [8] suggest a speed gain in almost all cases, an outcome that is to be expected as a result of the negligible communication and synchronization requirements of our algorithm. The efficiencies demonstrate that almost all our experiments scale very well up to 8 processors. Thereafter, the efficiency remains very good either as d increases or as m increases. Efficiency also remains almost constant around 1 for $256000 \leq d \leq 400000$. We remark the absence of a sharp fluctuation in the efficiency levels, mainly because our algorithm does 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. 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 [9] for solving large Niederreiter linear systems for trinomials over F_2 are currently under-way.

5 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 high efficiency that is better maintained as the degree of the polynomial or the number of its irreducible factors 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 [9] 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 [10], 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 [11], the hybrid algorithm is a simpler approach for moderately high record factorizations of sparse polynomials over F_2, requiring reasonable running times [9]. 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. Niederreiter, H.: "A New Efficient Factorization Algorithm for Polynomials over Small Finite Fields", *AAECC*, Vol. 4, 1993, pp. 81-87.
2. Göttfert, G.: "An Acceleration of the Niederreiter Factorization Algorithm in Characteristic 2", *Math. Comp.*, Vol. 62, 1994, pp. 831-839.
3. Niederreiter, H.: "Factorization of Polynomials and some Linear Algebra Problems over Finite Fields", *Lin. Alg. and its App.*, Vol. 192, 1993, pp. 301-328.
4. Niedrreiter, H.: "Factoring Polynomials over Finite Fields Using Differential Equations and Normal Bases", *Mathematics of Computation*, vol. 62, 1994, pp. 819-830.
5. Valiant, L. G.: "A Bridging Model for Parallel Computation", *Comm. of the ACM*, Vol. 33, 1990, pp. 103-111.
6. Hill, J. M. D., McColl, W. F., and Skillicorn, D. B.: "Questions and Answers about BSP", *Report PRG-TR-15-96*, Oxford University Computing Laboratory, 1996.
7. Hill, J. M. D., McColl, W. F., Stefanescu, D. C., Goudrea, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T., Bisseling, R. H.: "BSPlib: The BSP Programming Library", *Parallel Computing*, Vol. 24, 1998, pp. 1947-1980.
8. Abu Salem, F: "A BSP Parallel Model of the Göttfert Algorithm for Polynomial Factorization over F_2", Report PRG-RR-03-14, Oxford University Computing Laboratory, July 2003.
9. Abu Salem, F: "A New Sparse Gaussian Elimination Algorithm and the Niederreiter Linear System for Trinomials over F_2", Report PRG-RR-03-18, Oxford University Computing Laboratory, August 2003.
10. Roelse, P.: "Factoring High-Degree Polynomials over F_2 with Niederreiter's Algorithm on the IBM SP2", *Math. Comp.*, Vol. 68, 1999, pp. 869-880.
11. Fleischmann, P., Holder, M., and Roelse, P.: "The Black-Box Niederreiter Algorithm and its Implementation over the Binary Field", *Math. Comp.*, Vol. 72, 2003, pp. 1887-1899.