Inversion of Several Field Elements: A New Parallel Algorithm

Pradeep Kumar Mishra and Palash Sarkar

Cryptology Research Group,
Applied Statistics Unit,
Indian Statistical Institute,
203 B T Road,
Kolkata-700108, INDIA.
e-mail {*pradeep_t, palash*}*@isical.ac.in*

Abstract. In many crypographic hardware or software packages, a considerable part is devoted to finite field arithmetic. The finite field arithmetic is a very costly operation in comparison to other finite field operations. Taming the complexity of this operation has been a major challenge for researchers and implementers. One approach for the purpose is accumulate all the elements to be inverted and to compute several inversions simultaneously at the cost of one inversion and some multiplictions. One such algorithm is known as Montgomery's trick. However Montgomery's trick does not allow much parallelism. In [5] an algorithm for computation of inverses of several field elements simultaneously in parallel has been proposed. The algorithm allows ample scope for parallelism and performes well if there is no restriction on the number of processors used. In the current work, we present an algorithm, which is same in complexity as Montgomery's trick but suitable for a parallel implementation. In parallel implementation, it computes inverse of n elements in $2 \log n$ parallel rounds. It performs better than both the previous algorithms under the circumstances where the restricted number of multipliers is used.

keywords Parallel algorithm, Montgomery's trick, inversion.

1 Introduction

In coding theory and cryptography, finite fields have a lot of applications. In the implementation of many cryptographic primitives a non-trivial part is the implementation of finite field arithmetic. In a finite field the basic operations are addition, multiplication and inversion. Of these operations addition is generally the cheapest one and inversion is the costliest. An addition takes negligible time in comparison to time taken by a multiplication. The cost of inversion varies a lot depending upon the underlying field. For prime fields, the cost of inversion can be 30 to 50 times that of an multiplication [1], [2].

Suppose we wish to invert α and β. Instead of computing two inverses we can compute the product $\alpha\beta$ and compute its inverse. Then we can compute inverse of α and β by two more multiplications: $\alpha^{-1} = (\alpha\beta)^{-1}\beta$ and $\beta^{-1} = (\alpha\beta)^{-1}\alpha$. Thus inverses of two field elements can be computed at the cost of one inversion and 3 multiplications. This trick and its elegant generalisation to n elements is called Montgomary's trick. It takes one inversion and $3(n-1)$ multiplications to compute the inverse of n elements using the Montgomery's trick.

However, Montgomery's trick does not allow much scope for parallel implementation. In [5], a parallel algorithm for simultaneous computation of inverses has been proposed. It performs better than Montgomery's trick if the number of multipliers is more than 2. With 2 multipliers Montgomery's trick is a better alternative. Besides, the algorithm computes higher number of multiplications than required by Montgomery's trick.

In the current work we present a new algorithm which performs as good as Montgomery's trick in sequential execution. In parallel implementation it is much better than Montgomery's trick. If the number of multipliers is restricted, the algorithm performs better than the parallel algorithm proposed in [5]. The algorithm in [5] performs the best when the number of multipliers employed is $3/2$ times n for inverting n elements. For n or lesser number of multipliers, the performance of our new algorithm is much better. In the new algorithm requires as much memory as Montgomery's trick.

2 Background

2.1 Montgomery's Trick

In Montgomery's trick (see for example [4]) inverses are computed as follows. Let $x_1, \cdots, x_n$ be the elements to be inverted. Set $a_1 = x_1$ and for $i = 2, \ldots, n$ compute $a_i = a_{i-1}x_i$. Then invert a_n and compute $x_n^{-1} = a_{n-1}a_n^{-1}$. Now, for $i = n-1, n-2, \ldots, 2$, compute $a_i^{-1} = x_{i+1}a_{i+1}^{-1}$ and $x_i^{-1} = a_{i-1}a_i^{-1}$. Finally compute $x_1^{-1} = a_1^{-1} = x_2a_2^{-1}$. This procedure provides $x_1^{-1}, \ldots, x_n^{-1}$ using a total of $3(n-1)$ multiplications and one inversion. However, there is very little scope for parallel implementation of this algorithm. The first part of the algorithm computing the a_i's is inherently sequential. One can not compute a_j before computing a_{j-1}. In the next section we describe an algorithm, which computes inverses of several field elements in parallel.

3 Computing Simultaneous Inverses

Let $x_1, x_2, \cdots, x_n$ be the field elements to be inverted. Let $A[1, \cdots, (2n-1)]$ be an array of $(2n-1)$ elements, each capable of storing one field element. The following algorithm computes the inverses simultaneously.

Algorithm 1
Input: Field elements $x_1, x_2, \cdots, x_n$.
Output: $x_1^{-1}, x_2^{-1}, \cdots, x_n^{-1}$.

1. For $i = n$ to $(2n-1)$, $A[i] \leftarrow x_{i-n+1}$;
2. For $i = (n-1)$ down to 1, $A[i] \leftarrow A[2i] * A[2i+1]$;
3. Invert $A[1]$, i.e. $A[1] \leftarrow A[1]^{-1}$;
4. For $i = 2$ to $2n-1$ step 2,
   $t \leftarrow A[i]$;
   $A[i] \leftarrow A[\lfloor i/2 \rfloor] * A[i \oplus 1]$;
   $A[i+1] \leftarrow A[\lfloor i/2 \rfloor] * t$;
5. Output $A[i]$, $(n \leq i \leq (2n-1))$;

Proposition 1. *The cost of Algorithm 1 is* $3(n-1)$ *multiplications and one inversion.*

Proof : It is obvious that Step 2 and 4 of the algorithm require $n-1$ and $3(n-1)$ multiplications respectively. There is one inversion in Step 3.

The algorithm requires $2n$ memory locations, each capable of holding one field element each ($(2n-1)$ for $A[]$ and 1 for t). The elements $A[n]$ to $A[2n-1]$ in the array store the input data and $A[1]$ to $A[n-1]$ are used for storing intermediate variables. Montgomery's trick also demands same amount of memory. However, the beauty of Algorithm 1 is that it can be implemented in parallel. In the next section we exhibit how it can be implemented in parallel.

4 Parallel Implementation

Let the elements to be inverted be $x_1, x_2, \cdots, x_n$ where $2^{r-1} \leq n \leq 2^r$. We assume that the algorithm is to be processed by 2^{r-1} multipliers and we have sufficient memory to store $2 \times n$ field elements. We name the multipliers as $P_1, P_2, \cdots, P_{2^{r-1}}$. In fact we do not need more than 2^{r-1} multipliers. The algorithm can also be run with less number of multipliers, but the number of parallel multiplication rounds will be more. We will discuss that scenario in details later. Next, we describe our algorithm.

Algorithm 2
Input: Field elements $x_1, x_2, \cdots, x_n$.
Output: $x_1^{-1}, x_2^{-1}, \cdots, x_n^{-1}$.
1. **Initialisation:** For $i = n$ to $2n-1$, $A[i] \leftarrow x_i$;
2. For $k \leftarrow 1$ to r do in parallel
   (Round k:)
   For $i = 2^{r=k}$ to $\min\{2^{r-(k-1)} - 1, n-1\}$;
   $P_{i+1-2^{r-k}}$ computes $A[i] \leftarrow A[2i] * A[2i+1]$
3. **Round r+1:**
   Invert the element in $A[1]$ and store to $A[1]$;
4. For $k \leftarrow r+2$ to $2r+1$ do
   (Round k:)
   For $2^{k-(r+1)} \leq i \leq 2^{k-r} - 1$,
   $P_{i-2^{k-(r+1)}+1}$ computes in parallel $A[i] \leftarrow A[\lfloor i/2 \rfloor] * A[i \oplus 1]$;

Output $A[i]$, $n \leq i \leq (2n-1)$.

Proposition 2. *Algorithm 2 correctly computes the inverses of 2^r elements in $2r$ parallel multiplication rounds.*

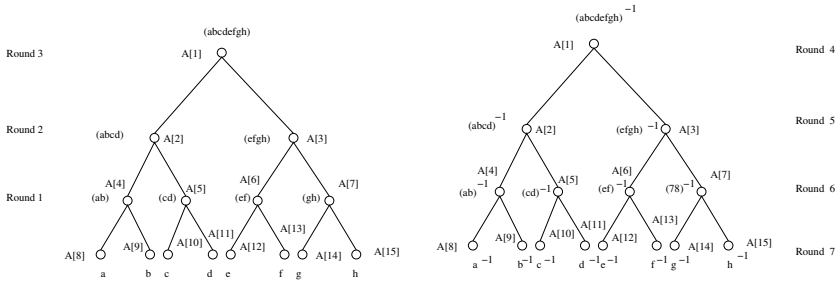

Fig. 1. Algorithm 2 in Action with $n = 8$. The figure demonstrates various parallel rounds of the Algorithm 2 for inverting 8 elements $a, \cdots, h$. Round 4 is the inversion round.

4.1 Computing with Lesser Number of Multipliers

With 2^{r-1} multipliers Algorithm 2 can compute the inverses in $2r$ parallel rounds. Let the number of available multipliers be $t = 2^p$. Then the obvious way of carying out the computations is to allow the available multipliers to parallelly compute one round of Algorithm 2 possibly in more than one parallel rounds.

To carry out the computations of round $k, (1 \leq k \leq r)$ of Algorithm 2, the t processors will require $\lceil 2^{r-k}/2^p \rceil$ parallel rounds of computations. The $(r+1)$st round is an inversion round. Similarly, for round k, $r+2 \leq k \leq 2r+1$, the t processors will require $\lceil 2^{k-r-1}/2^p \rceil$ parallel rounds of computations. Hence we have,

Proposition 3. *With $t = 2^p$ multipliers Algorithm 2 can be computed in*

$$\sum_{k=1}^{r} \lceil \frac{2^{r-k}}{2^p} \rceil + \sum_{i=r+2}^{2r} \lceil \frac{2^{k-r-1}}{2^p} \rceil = 2 \sum_{k=1}^{r-1} \lceil \frac{2^k}{2^p} \rceil + 2^{r-p}$$

parallel rounds of computation besides one inversion round.

Table 1. Number of parallel rounds required for inverting $n = 8, 16, 32$ elements with $k = 8, 4, 2$ multipliers by Algorithm 2.

n/k	8	4	2
8	6	7	11
16	9	13	23
32	15	25	47

5 Performance and Comparison

In the Table 1 we show the number of parallel rounds required for inverting n number of elements by k number of multipliers using Algorithm 2.

As Montgomery's trick is a sequential algorithm it will be unfair to compare performance of Algorithm 2 with the performance of Montgomery's trick. Recently in [5] a parallel algorithm for computing the inverses has been proposed. We compare performance of Algorithm 2 with the algorithm presented in [5]. For that we cite the following table (Table 2 from [5].

Table 2. Number of rounds required by k multipliers for inverting n elements by Algorithm in [5].

$n\backslash k$	8	4	2
8	6	9	16
16	12	21	40

Clearly, the algorithm presented in this work performs better. However, algorithm presented in [5] inverts 2^r elements with $3 \times 2^{r-1}$ multipliers in $r + 1$ parallel multiplication rounds for which Algorithm 2 takes $2r$ parallel multiplication rounds. So if there is no restriction on the number of multipliers then the algorithm presented in [5] is better than Algorithm 2. However, in practical implementations, where the number of multipliers is limited, Algorithm 2 is preferable.

5.1 Memory Requirement

Here by a memory unit we mean a storage unit capable of storing a field element. It is easy to check that Algorithm 1 and Montgomery's trick require n additional memory units to invert n elements. Algorithm 2 requires $(n - 1)$ additional memory units. In memory requirement, algorithm presented in [5] is the best. It requires only $n/2$ additional memory units.

6 Conclusion

In this work we presented a new algorithm and its parallel version to compute inversion of several field elements simultaneously. The parallel version can fit into parallel implementations of any cryptographic primitive which requires simultaneous inversions. In terms of performance, the new algorithm is better than the one presented in [5] in practical scenario where there is a limitation on the number of multipliers used.

References

1. K. Fong and D. Hankerson and J. López and A. Menezes. Field inversion and point halving revisited. Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2003.
2. A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
3. P. Montgomery. Speeding the Pollard and Elliptic Curve Methods for Factorisation. In *Math. Comp.*, vol 48, pp 243-264, 1987.
4. H. Shacham, D. Boneh. Improving SSL Handshake Performance via Batching. In *CT-RSA*, LNCS 2020, Springer-Varlag, 2001.
5. P. Sarkar, P. K. Mishra and R. Barua. A Parallel Algorithm for Computing Simultaneous Inversions with Application to Elliptic Curve Scalar Multiplicaton (Extended Abstract) To appear in *Proceeding of 46th Midwest Symposium on Systems and Circuits, 2003.*