A New Parallel Window-Based Implementation of the Elliptic Curve Point Multiplication in Multi-Core Architectures

Saikat Basu

Department of Computer Science and Engineering, National Institute of Technology
M. G. Avenue, Durgapur - 713209, India
(Email: deepbasu007@gmail.com)

(Received July 24, 2010; revised and accepted Oct. 4 & Nov. 13, 2010)

Abstract

Point multiplication is an important computation in elliptic curve cryptography. Various methods like binary method and window method have been implemented in the past for performing efficient elliptic curve point multiplications. However, all these implementations rely on serial computations performed on uni-core architectures. A new approach on multi-core implementation has been proposed in this paper. Hence, a new parallel algorithm has been designed and implemented on machines with upto 8 cores. Later, experimental studies have been performed with different window sizes and degrees of parallelism.

Keywords: Elliptic curve, exponentiation, point multiplication, parallel algorithm, window method

1 Introduction

Elliptic curves over a finite field FP are used in conjunction with existing cryptographic schemes such as the Diffie-Hellman scheme, ElGamal scheme and RSA scheme. Since their inception in [17], Elliptic curve cryptosystems have been in wide use among the cryptographic community for their relatively better security and ease of implementation. The size of the elliptic curve determines the difficulty of the problem. It is believed that the same level of security afforded by an RSA-based system with a large modulus can be achieved with a much smaller elliptic curve group. Using a small group reduces storage and transmission requirements. It is known that for a 80-bit security level, the key size is 1024 for RSA while it is only 160 bits for ECC as shown in [20]. Owing to these factors, computations over elliptic curves have been a vibrant area of research in recent years.

The basic operation performed on an elliptic curve is the computation of a multiple d.P of a point P on the elliptic curve modulo n, which corresponds to the computation of $xd \bmod n$. For large n and d, the time complexity of elementary operations as well as the number of elementary operations is very high. Thus reducing the number of such operations as well as the cost of each is an important issue involved in the elliptic curve cryptosystems. Various schemes have been proposed in the past that highlight this issue through methods like the binary method [11], the window method [3] and the addition-subtraction chains [12]. A refined algorithm was proposed in [14] that uses a signed binary window method to speed up the computation of d.P.

Although, all the aforementioned computation schemes propose techniques to improve the computation times, but all of them are implemented in serial. For example, the binary and window methods compute the value of d using its binary representation, considering the bits (or windows) from left to right, i.e., starting from the most significant bit and continuing up to the least significant one. The signed binary window method proposed in [14] consists of four phases (1) representation of d. (2) spitting the representation into segments (windows), (3) computing the segments, and (4) concatenating all the segments. Out of these the phases 3 and 4 are the computationally most intensive ones.

Here, a new *parallel algorithm* is proposed that divides these computation phases into parallel segments each implemented in one of the cores on a multi-core architecture. This consists of the computation of segments and concatenating the computed segments in parallel.

The rest of the paper is organized as follows: Section 2 describes the literature review and the foundation of the proposed algorithm. Section 3 illustrates the new parallel algorithm. The implementation details are discussed in Section 4. Section 5 contains the experimental studies whereas Section 6 concludes the paper.

Table 1: Computing times of an addition (ECADD) and a doubling (ECDBL)

Coordinate System	ECADD		ECDBL	
	$Z \neq 1$	$Z = 1$	$\alpha \neq -3$	$\alpha = -3$
A	2M+aS+1I	-	2M+2S+1I	
P	12M+2S	9M+2S	7M+5S	7M+6S
τ	12M+4S	8M+3S	4M+6S	4M+4S
τ^C	11M+3S	8M+3S	5M+6S	5M+4S
τ^m	13M+6S	9M+5S	4M+4S	

2 Literature Review

An *elliptic curve* E *over a field* K is defined by an equation

$$E \quad : \quad y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (1)$$

where a_1, a_2, a_3, a_4, a_6, K and $\Delta = 0$, where Δ is the *discriminant* of E. Table 1 shows the computing times of an addition (ECADD) and a doubling (ECDBL).

Equation (1) is the Weierstrass equation defined over K. If the characteristic of K is not equal to 2 or 3, then the admissible change of variables,

$$(x, y) \quad = \quad (\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24})$$

transforms E to the curve

$$y^2 \quad = \quad x^3 + ax + b$$

where a, b, K. The discriminant of this curve is $\Delta = -16(4a^3 + 27b^2)$.

The Group Law:
The set of points $E(K)$ forms an abelian group with ∞ serving as its identity. It is this group that is used in the construction of elliptic curve cryptographic systems. Algebraic formulas for the group law are presented next for elliptic curves E of the simplified Weierstrass form (2) in affine coordinates when the characteristic of the underlying field K is not 2 or 3 (e.g., $K = F_p$ where $p > 3$ is a prime).

Group Law for $E/K : y^2 = x^3 + ax + b$**,** $char(K) \neq 2, 3$

1) *Identity.* $P + \infty = P$ for all $P \in E(K)$.

2) *Negatives.* If $P = (x, y) \in E(K)$, then $(x, y) + (x, -y) = \infty$. The point $(x, -y)$ is denoted by $-P$ and is called the negative of P.

3) *Point Addition.* Let $P = (x_1, y_1) \in E(K)$, where $P \neq \pm Q$. Then $P + Q = (x^3, y^3)$ where

$$x_3 \quad = \quad (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2$$

and

$$y_3 \quad = \quad ((y_2 - y_1)/(x_2 - x_1)(x_1 - x_3) - y_1$$

4) *Point doubling.* Let $P = (x_1, y_1) \in E(K)$, where, $P \neq -P$. Then, $2P = (x_3, y_3)$, where,

$$x_3 \quad = \quad (\frac{3x_1^2 + a}{2y_1})^2 - x_1$$

and

$$y_3 \quad = \quad (\frac{3x_1^2 + a}{2y_1})(x_1 - x_3) - y_1.$$

3 The Proposed Parallel Algorithm

In this section, a new parallel algorithm is proposed that computes the elliptic curve point multiplication d.P in parallel. Accordingly, the bit stream d is at first split into segments. Then, each segment is assigned to a processor. Once all the segment values are computed, they are concatenated in parallel. Since both the pre-computation and post-computation stages are performed in parallel, so, there is a significant improvement in speedup values. The various steps of the algorithm can be listed as follows.

3.1 Representation of d

Here, the binary representation of d is converted to a form in which there are three digits, viz, 0, 1 and -1 (denoted by $\bar{1}$). The transformation algorithm discussed next increases the average length of zero runs while minimizing the weights (number of non-zero digits) in the new representation marked as T. This way of representation was first proposed in [14]. The algorithm is incorporated in the present paper for the sake of maintaining integrity and clarity of discussions. In T, the average length of zero runs $Z(T)$ is defined as follows.

$$Z(T) \quad = \quad \frac{1}{L}\sum_{i=0}^{L-1} z(i)$$

where, $z(i) = 1 + z(i - 1)$ if $t_i = 0$. $(0 \leq i \leq L - 1) = 0$, if $t_i \neq 0$ and, $z(-1) = 0$.

This is an improved version of the non-adjacent form (NAF) in which no two non-zero digits are adjacent to each other. Increasing the average number of zero runs helps in reducing the number of non-doubling addition

operations while computing d.P. at the cost of doubling which are computationally less expensive.

The above algorithm converts the bit stream B to the transformed string T as follows (see Algorithm 1). Let $\sharp 1$ (B) indicate the number of 1's in the bit stream B and $\sharp 0$ (B) be the number of 0's. The algorithm counts

$$D(B) \quad = \quad \sharp_1(B) - \sharp_0(B)$$

and applies the transformation rule repeatedly to B with $D(B) \geq 3$. Thus a threshold value of 3 is used here. The output of the algorithm is not sparse as two adjacent digits may be non-zero. The Addition-subtraction chain method proposed in [12] sets this threshold value as 2. So, it is more sparse as compared to the one proposed in this paper. The transform algorithm creates a bit stream with an average length of zero runs being 1.42.

3.2 Splitting

The next phase is spitting that divides the number T into segments (also called windows) of a fixed length. The following algorithm generates the list of segments (see Algorithm 2).

3.3 Computing the Segments

Due to the property of the transform algorithm, the value of a segment cannot attain the value (2^w-1) or $-(2^w-1)$ and it can attain a maximum value of (2^w-3) or $-(2^w-3)$. Now, for computing the segment values we calculate an addition sequence $\{1, 2, 3, 5, 7, 9, 11, 13, \ldots, (2^w-3))\}$ containing all the segment values. This is primarily called the pre-computation phase whereby a matrix is created prior to the actual computations containing the values of the segments. This pre-computation phase involves the computation of the addition sequence for which a new parallel implementation is proposed in this paper. For this purpose, the values in the addition sequence are divided among the cores, each core computing a particular set of the sequence values. This is done by assigning the alternate segment values to the individual cores. For instance, in a dual-core implementation, the computation of addition sequence values, i.e., $1P, 2P, 3P, 4P, 5P, 9P, 13P, \ldots$ are assigned to one core, while the computation of the addition sequence values $1P, 2P, 3P, 4P, 7P, 11P, 15P, \ldots$ are assigned to another core. Although the computation of the first four values up to $4P$ is done by both the cores but still this parallel implementation is quite efficient for large window values where window width $w = 10$. This is because the addition sequences then have to computed from $1P$ up to $(2w-3)P$, i.e., $1021P$ (see Algorithm 3).

The algorithm discussed above computes the segment values in parallel with the load evenly distributed among the cores. The average number of computations performed by the serial execution in only one core takes $\sharp(C)_s$ number of computations where

$$\sharp(C)_s \quad = \quad 2^w - 3 \tag{2}$$

Algorithm 1 The transformation algorithm

```
1: begin:
2: M ← 0, J ← 0, Y ← 0, X ← 0, U ← 0, V ← 0, W ← 0, Z ← 0;
3:   while X < log_2 d
4:   if B[X] = 1
5:     Y ← Y + 1;
6:   else
7:     Y ← Y − 1;
8:   end
9:   X ← X + 1;
10:  if M = 0
11:    if Y − Z ≥ 3
12:      while J < W
13:        T[J] ← B[J];
14:      J ← J + 1;
15:      end
16: T[J] ← −1; J ← J + 1; V ← Y; U ← X; M ← 1;
17:    else if Y < Z
18:      Z ← Y; W ← X;
19:    end
20:  else if V − Y ≥ 3
21:    while J < V
22:      T[J] ← B[J] − 1; J ← J + 1;
23:    end
24:    T[J] ← 1; J ← J + 1; Z ← Y; W ← X; M ← 0;
25:  else if Y > V
26:    V ← Y; U ← X;
27:  end
28: end
29: if (M = 0) ∨ (M = 1 ∧ V ≤ Y)
30:  while J < X
31:    T[J] ← B[J] − M; J ← J + 1;
32:  end
33:  T[J] ← 1 − M; T[J + 1] ← M;
34: else
35:  while J < V
36:    T[J] ← B[J] − 1; J ← J + 1;
37:  end
38: T[J] ← 1; J ← J + 1;
39: while J < X
40:    T[J] ← B[J]; J ← J + 1;
41:  end
42:    T[J] ← 1; T[J + 1] ← 0;
43:  end
44:  return T
45: end
```

whereas, the parallel implementation takes $\sharp(C)_p$ computations where,

$$\sharp(C)_p \quad = \quad 2n + \frac{2^w - 3 - 2n}{n} \tag{3}$$

The $2n$ part on the right hand side of Equation (3) is the serial part and the $\frac{2^w-3-2n}{n}$ part represents the parallel portion of the implementation. It can be seen that putting $n = 1$ in Equation (3) yields Equation (2)

Algorithm 2 Algorithm 2 (procedure split)

input T, output S
begin:
Let S be the empty segment list
w is the window size
while $length(T) \geq w$
  $W \leftarrow$ left w digits of T
  $R \leftarrow T$ excluding W
$\tilde{W} \geq \leftarrow W$ excluding the right 0s /*Generates the new window*/
$R \leftarrow R$ excluding left 0s
$S \leftarrow S + \tilde{W}$ /*Adds the new window obtained to the existing list of segments*/
$T \leftarrow \tilde{R}$
  end
return S
end

Algorithm 3 The parallel computation algorithm

begin:
$w \leftarrow$ width of the windows assigned
$n \leftarrow$ total no. of cores
compute the values $1P, 2P, 3P, \ldots, 2nP$
for each core i
  for $k \leftarrow 0$ to $\frac{2^(w-1)-i-n-2}{n}$
    compute the value $(2n + 2i + 1 + 2kn)P$
  end for
end for
end

which proves the argument.

3.4 Concatenating The Computed Segments

Once the segment values are computed in the pre-computation stage, the next step is to perform the actual point multiplication using the concept of point doublings and non-doubling additions. This requires the pre-computed values obtained in the previous step. In signed-binary window method proposed in [14], the concatenation of the segments is done in serial using only one core. In that work, the window values are computed starting from the most significant segment and continuing up to the least significant one with the output of the first segment carried forward to the next and so on till the last one. In the present work, this concatenation part is implemented in parallel whereby a divide and conquer algorithm is proposed. This algorithm computes the segment values after dividing them among the cores, each core computing a particular block of segments. In the serial implementation discussed in [14], the value dP where d=$d_n d_{n-1} d_{n-2} \ldots d_0$ is computed as follows.

$$\begin{aligned} dP &= ((\ldots(((dnP.2^{zeros(n)+length(n-1)} + d_{n-1}P) \\ &\quad . \ 2^{zeros(n-1)+length(n-2)} + d_{n-2}P). \\ &\quad 2^{zeros(n-2)+length(n-3)} + \ldots) + d_0).2^{zeros(0)} \end{aligned}$$

The above equation shows that the computation begins at the most significant segment denoted by d_n and ends at the least significant one, i.e., d_0.

Parallel concatenation.

In the present work, a new parallel algorithm is proposed that pre-allocates a group of segments to each core and combines the results after the cores have computed the segment values. This parallel implementation can be represented as follows.

$$\begin{aligned} dP &= (..(d_nP.2^{zeros(n)+length(n-1)} + d_{n-1}P) \\ &\quad . \ 2^{zeros(n-1)+length(n-2)} \text{computed by core 0} \\ &\quad + \ (d_{n-2}P).2^{zeros(n-2)+length(n-3)} + \ldots) \ldots \\ &\quad + \ d_0).2^{zeros(0)} \text{computed by core 1} \ \ldots \end{aligned}$$

where, $zeros(n)$ denotes the number of zeros to the right of the segment d_n and $length(n)$ denotes the width of segment d_n. The parallel algorithm attains substantial load balancing in terms of computational speedup and efficiency (see Algorithm 4).

Algorithm 4 The algorithm

begin:
for $i \leftarrow 0$ to n
  $sum_i \leftarrow \infty$
end for
for each core i
  for $t \leftarrow \frac{iP}{n}$ to $\frac{(i+1)P}{n}$
    compute the value of the segment d_tP
      $sum_i \leftarrow sum_i.2^{zeros(t)+length(t+1)} + d_tP$
    end for
end for
for $i \leftarrow 0$ to $n-1$
  $sum \leftarrow sum_i.2^{zeros(i)+length(i+1)} + sum_{i+1}$
end for
end

The aforementioned parallel concatenation algorithm does not change the number of elementary operations, it only reduces the effective computation time per operation by allocating the computation of individual set of segments to each core of the multi-processor. Although earlier implementations of elliptic curve algorithms were all designed to reduce the number of elementary operations, the algorithm proposed in this work strives to achieve both reduction in the number of elementary operations through the use of the transformation algorithm proposed in Section III.A as well as significant improvement in computation speed through the use of parallelism.

4 The Implementation

In this paper, the elliptic curve arithmetic has been implemented in Affine coordinates. The equation being chosen has a and b values both equal to 1. So, the resultant equation is as follows.

$$y^2 = x^3 + x + 1.$$

The parallel algorithm has been implemented in the C language using the OpenMP library. This library is available in gcc 4.3.2 and later. It helps in performing parallel computations in multi-core architectures available in Intel machines. For experimental purposes, an Intel Xeon x8 machine @ 2.33 Ghz has been used.

Amdahl's Law:
In the case of parallelization, Amdahl's law [1] states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and (1 - P) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is

$$\frac{1}{(1-P) + \frac{P}{N}}.$$

In the limit, as N tends to infinity, the maximum speedup tends to $1/(1-P)$. In practice, performance to price ratio falls rapidly as N is increased once there is even a small component of $(1-P)$.

Gustafson's Law:
On the other hand, Gustafson's Law [7] states that any sufficiently large problem can be efficiently parallelized using sufficient computing power. According to the law, the speedup due to N processors on a piece of code in which $(1-P)$ is the proportion that cannot be parallelized (remains serial) is given by

$$S = N - (1-P)(N-1).$$

But, there is a major difference between the implementation of the two Laws, i.e., the usage of the parameter P, which is different for both the laws. Asillustrated in [18], since both the P values are not same so, for the purpose of clarity, the P values for Amdahl's law is denoted by PA and that for Gustafson's law is denoted by PG. For Amdahl's Law, the parameter PA is derived using the time taken by the various portions of the algorithm as follows:

$$PA = \frac{t_s}{t_s + t_p(1)}$$

where, t_s denotes the processing time of the serial part of the program (using only 1 processor).

$t_p(1)$ denotes the processing time of the parallel part of the program (using only 1 processor).

So, using Amdahl's law, the theoretical maximum speedup can be defined as follows:

$$SpeedupS_A = \frac{t_s + t_p(1)}{t_s + \frac{t_p(1)}{N}} = \frac{1}{P_A + \frac{(1-P_A)}{N}}$$

For Gustafson's Law, the parameter PG is derived using the following equation.

$$P_G = \frac{t_s}{t_s + t_p(N)}$$

where, $t_p(N)$ denotes the processing time of the parallel part of the program using N processors.

So, using Gustafson's law, the experimental speedup can be defined as follows:

$$Speedup = S_G = \frac{t_s + Nt_p(N)}{t_s + t_p(N)} = P_G + (1-P_G)N.$$

In the present work, both Amdahl's Law and Gustafson's Law has been used. This is because, when the serial algorithm is implemented in parallel, the structure of the code changes. So, there is a difference between the computational complexities in the parallel portion of the code implemented in 1 processor and that of the complexity using multiple processors. Hence, the speedup obtained through the implementation of Amdahl's law gives the theoretical speedup obtained with the assumption that the structure of the code does not change when implemented in parallel. On the other hand, the speedup obtained through Gustafson's law gives the actual experimental speedup taking into consideration the change in the structure of the algorithm when implemented in parallel.

5 Experimental Studies

The experimental studies have been performed to compute dP where d is of width λ, i.e., d is λ bits long. The binary representation of d is divided into windows of width δ. Different theoretical and experimental speedup values are obtained for different values of N and δ, where, N is the total number of processors used. For the purpose of experimentation, an Intel Xeon x8 @ 2.33 Ghz machine was used. In the following graphical representations (see Figures 1-5), the number of processors used is listed along the X-axis, whereas, the Y-axis denotes the computational speedup attained. The green lines indicate the theoretical maximum speedup obtained through Amdahl's law, while, the red lines denote the experimental speedup obtained through Gustafson's law.

It is evident from the experimental results that the theoretical speedups obtained through Amdahl's law are an upper bound on the experimental speedup values obtained through Gustafson's law. Also, it can be seen that the speedup values are nearly equal to the number of processors used, i.e., N as would happen in the ideal case. It is also evident from the graphs that as the window sizes increase, there is an increase in the speedup values as can be seen that when the window size δ approaches the value 12, the experimental speedup nearly reaches the theoretical maximum. However, there is a disadvantage associated with choosing a high value of δ since a high value indicates higher pre-computation and hence a higher amount

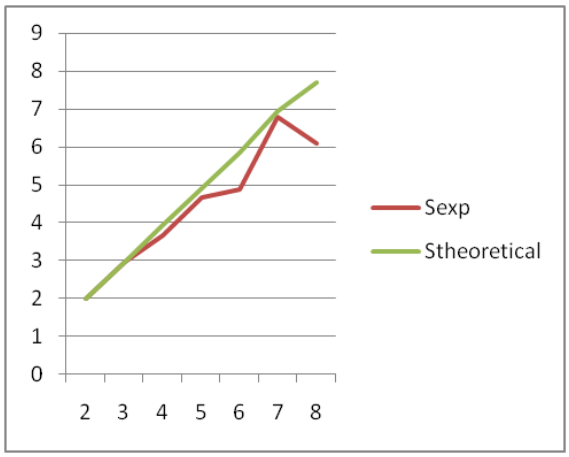

Figure 1: No. of processors, $\lambda = 1024$, $\delta = 4$

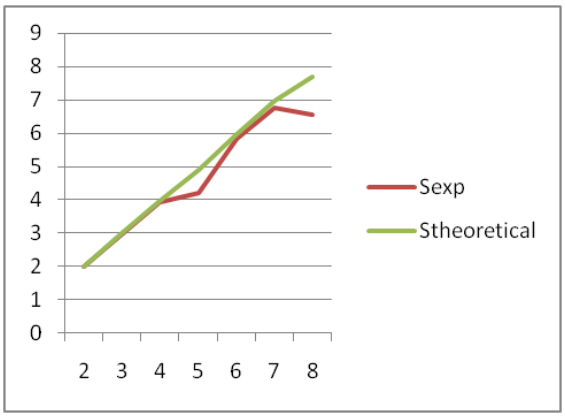

Figure 2: No. of processors, $\lambda = 1024$, $\delta = 6$

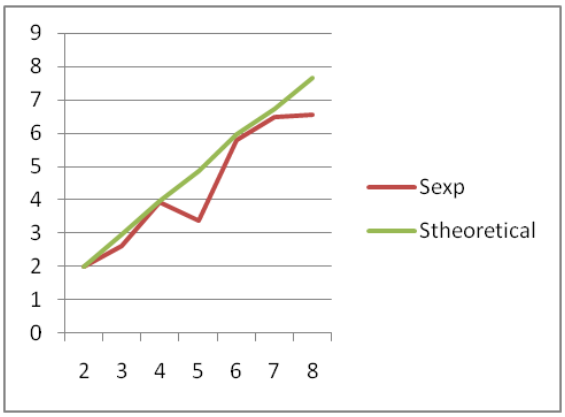

Figure 3: No. of processors, $\lambda = 1024$, $\delta = 8$

of storage required for the pre-computed array. The pre-computation stage with window sizes of 4 bits involves the computation of 2^4, that is, 16 values, whereas, it involves the computation of 2^{15}, that is, 32768 values. So, it leads to frequent segmentation faults on systems with lesser amount of cache memory. In order to maintain the

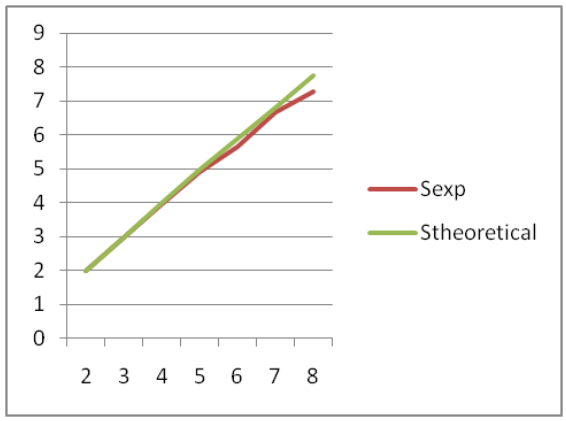

Figure 4: No. of processors, $\lambda = 1024$, $\delta = 10$

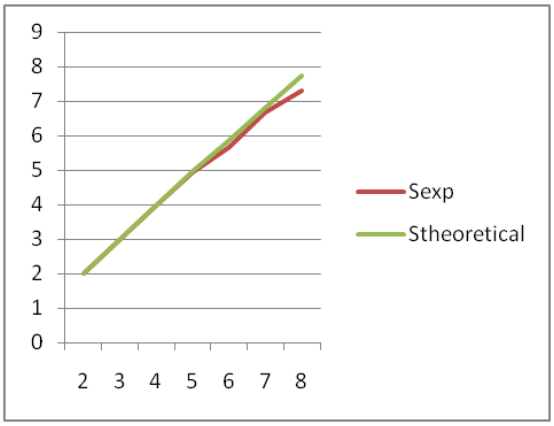

Figure 5: No. of processors, $\lambda = 1024$, $\delta = 12$

tradeoff between the window sizes and the ease of implementation on machines with limited cache, window sizes of the order of 10 are selected.

6 Conclusions

The transformation algorithm proposed in Section 3A reduces the number of elementary operations whereas the parallel computation and concatenation stages proposed in Sections 3.C and 3.D reduce the computation time through efficient parallel implementation. The simulation studies performed by implementing the parallel algorithm on the multi-core architectures indicate speedup that attain values nearly equal to the ideal case, i.e., speedup of the order of N, for N number of processors. This is achieved by implementing both the pre-computation and post-computation stages in parallel. The tradeoff between the amount of pre and post-computations is maintained by selecting optimum window sizes. As a result of the above implementations, efficient speedup is obtained while performing point multiplication in multi-core architectures. No parallel point multiplication algorithm is available as of the date of this work. Although, a fast par-

allel elliptic curve multiplication algorithm is discussed in [10], but, it only highlights the resistance of the algorithm against Side-channel attacks. Also, the parallel algorithm illustrated in that work is based on the point addition and point multiplications being implemented in two registers which is an altogether different approach of study as compared to the present work. So, comparative studies with existing algorithms on the basis of speedup values could not be done in this work. However, simulation studies show that parallel implementations of the elliptic curve point multiplication attains speedup of the order of (N-1), for N number of processors and hence provides a good starting point for parallel elliptic curve algorithms.

References

[1] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," *AFIPS Conference Proceedings*, vol. 30, pp. 483V485, 1967.

[2] D. F. Aranha, J. Lȯpez, and D. Hankerson, "High-speed parallel software implementation of the $\eta\tau$ pairing," *Topics in Cryptology - CT-RSA 2010*, vol. 5985, pp. 89-105, 2010.

[3] J. Bos and M. Coster, "Addition chain heuristics," *Proceddings of CRYPTO'89*, vol. 435, pp. 400-407, 1989.

[4] R. P. Gallant, R. J. Lambert, and S. A. Vanstone, "Faster point multiplication on elliptic curves with efficient endomorphisms," *Advances in Cryptology - CRYPTO 2001*, vol. 2139, pp. 190-200, 2001.

[5] J. M. Garcia and R. M. Garcia, "Parallel algorithm for multiplication on elliptic curves," Report - 2002/179, 2002.

[6] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129-146, Apr. 1998.

[7] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-533, 1988.

[8] A. Gutub and M. Ibrahim, "High performance elliptic curve $GF(2^k)$ cryptoprocessor architecture for multimedia," *Proceedings IEEE International Conference on Multimedia & Expo*, pp. 81- 84, ICME, Baltimore, Maryland, USA, July 6-9, 2003.

[9] D. Hankerson, "Implementing elliptic curve cryptography (a narrow survey)," *Workshop in Implementation of Cryptographic Methods*, pp. 1-110, 2005.

[10] T. Izu and T. Takagi, "A fast parallel elliptic curve multiplication resistant against side channel attacks," *Proceedings of Indocrypt 2002*, LNCS. 2274, pp. 371-374, Springer-Verlag, 2002.

[11] D. E. Knuth, *Art of Computer Programming*, vol. 2, Addison Wesley, 1969.

[12] F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," *Theoretical Informatics and Applications*, vol. 24, no. 6, pp. 531-544, 1990.

[13] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203-209, 1987.

[14] K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method," *Proceedings of Crypto 1992*, vol. 740, pp. 345-357, 1998 .

[15] T. C. Lin, "Algorithms on elliptic curves over fields of characteristic two with non-adjacent forms," *International Journal of Network Security (IJNS)*, vol. 9, no. 2, pp. 117-120, 2009.

[16] M. M. Rasslan, "A stamped hidden-signature scheme utilizing the elliptic curve discrete logarithm problem," *International Journal of Network Security (IJNS)*, vol. 13, no. 1, pp. 49-57, 2011.

[17] F. Rodṙiguez-Henṙiquez, N. A. Saqib, and A. Dȉaz-Pėrez, "A fast parallel implementation of elliptic curve point multiplication over $GF(2^m)$" *Microprocessors and Microsystems*, vol. 28, no. 5-6, pp. 329-339, Aug. 2004.

[18] Y. Shi, *Reevaluating Amdahl's Law and Gustafson's Law*, Oct. 1996. Unpublished monograp.

[19] J. Shi and H. Yun, "Software implementations of elliptic curve cryptography," *International Journal of Network Security (IJNS)*, vol. 7, no. 1, pp. 141-150, 2008.

[20] R. Soram and M. Khomdram, "Juxtaposition of RSA and elliptic curve cryptosystem," *International journal of Computer Science and Network Security*, vol. 9, no. 9, pp. 11-21, 2009.

[21] D. Yong, Y. F. Hong, W. T. Wang, Y. Y. Zhou, and X. Y. Zhao, "Speeding scalar multiplication of elliptic curve over $GF(2^{mn})$," *International Journal of Network Security (IJNS)*, vol. 11, no. 2, pp. 70-77, 2010.

Appendix

This section contains the results of the experimental studies performed in section V. Table 2 lists the speedup values obtained when the algorithm is implemented in parallel in an Intel Xeon x8 machine @ 2.33 Ghz. The parallel algorithm is run iteratively for 10 consecutive times. The time taken to run the serial and parallel parts of the program are used to derive the values of SA and SG that denote the theoretical and experimental speedup values respectively. The size of bit streams (λ) is chosen as 1024 for all the experiments and the window widths (δ) are chosen serially.

Saikat Basu is a final year undergraduate student of the Computer Science and Engineering department at National Institute of Technology, Durgapur in India. He joined the institute in 2007 and graduates in 2011. His primary research interest lies in the fields of Cryptography and Network Security.

Table 2: Listing of the speedup values for various window sizes and number of processors

Size of $d(\lambda)$	**No. of** Processors (N)	**Window** Width (δ)	**Theoretical** Speedup (S_A)	**Experimental** Speedup (S_G)
1024	2	4	1.9917	1.9891
1024	2	6	1.9969	1.9882
1024	2	8	1.9973	1.9879
1024	2	10	1.9972	1.9914
1024	2	12	1.9969	1.9846
1024	3	4	2.9702	2.96
1024	3	6	2.9862	2.966
1024	3	8	2.967	2.619
1024	3	10	2.9938	2.976
1024	3	12	2.9843	2.9475
1024	4	4	3.9407	3.669
1024	4	6	3.9790	3.9331
1024	4	8	3.9781	3.9300
1024	4	10	3.9792	3.9467
1024	4	12	3.9158	3.89
1024	5	4	4.8947	4.6594
1024	5	6	4.8901	4.195
1024	5	8	4.8827	4.3732
1024	5	10	4.9834	4.9192
1024	5	12	4.8505	4.7611
1024	6	4	5.8408	4.8807
1024	6	6	5.9727	5.8348
1024	6	8	5.9748	5.8098
1024	6	10	5.8772	5.6632
1024	6	12	5.7751	5.7322
1024	7	4	6.944	6.7826
1024	7	6	6.9616	6.7761
1024	7	8	6.7219	6.4874
1024	7	10	6.8175	6.6674
1024	7	12	6.9010	6.6974
1024	8	4	7.6894	6.0934
1024	8	6	7.6964	6.5543
1024	8	8	7.6735	6.5578
1024	8	10	7.7517	7.289
1024	8	12	7.8936	7.5890