Proceedings of the 2007 IEEE International Conference on Telecommunications and
Malaysia International Conference on Communications, 14-17 May 2007, Penang, Malaysia

Multi-threading Elliptic Curve Cryptosystems

Uma S.Kanniah and Azman Samsudin
School of Computer Sciences, Universiti Sains Malaysia, 11800 Penang, Malaysia.
uma@cs.usm.my, azman@cs.usm.my

***Abstract*—Elliptic Curve Cryptosystems (ECC) is being widely used in cryptographic systems due to its unique security features. In many cases, ECC had also been crafted onto specialized processor for performance reason. However, building a specialized ECC processor is not only time consuming but also costly. Therefore, from the cost-effectiveness point of view, the software approach of ECC in a multiprocessing environment is indeed more attractive, especially with the recent development in multi-core processors. The main objective of this research is therefore to implement identified parallel mathematical algorithms in multi-threading environment to support the development of ECC in software. Two parallel mathematical algorithms, Karatsuba and Montgomery, are incorporated and tested for point multiplication in Elliptic Curve (EC) algorithm to achieve high performance in execution for on-the-shelf multi-core architecture processors.**

Index—**Cryptography, Elliptic Curve Cryptosystems, Multi-threading, Karatsuba Algorithm, and Montgomery Algorithm.**

I. INTRODUCTION

CRYPTOGRAPHY is an ancient science of secret writing that has found a new usage in the modern world. Among the mathematic primitives that are being used in cryptography is the Elliptic Curve Cryptography (ECC). The ECC is applicable to be used in a constrained environment such as smart cards and mobile devices. Subsequently, ECC is preferred because it can provide a much higher security with smaller key size.

There are many mathematical algorithms for basic mathematical operation that can be used in the ECC development. Some of the earlier algorithms that have been discussed are the Karatsuba and Montgomery algorithms. The algorithms mentioned above are widely used in implementing the ECC for hardware purposes. Among the mathematical functions found in ECC is point multiplication which allows the multiplication of large integers.

In this paper, the software solution of ECC computations is incorporated in the dual-core machine architecture to allow more operations to be done in a same unit of time. This will open a new gateway in computing the existing ECC functions. The integration between ECC and dual-core will definitely constitute to the development of ECC as well as the devices that exploits ECC.

II. ELLIPTIC CURVE CRYPTOGRAPHY OVERVIEW

ECC was first proposed by Koblitz and Miller in 1986 [2]. Since then, ECC has evolved and proven to offer more security than any other public key cryptosystem per key-bits. The advantages of ECC is its inverse operation will get harder against the increasing key length if compared with the inverse operation that takes place in Diffie-Hellman and RSA. This actually means that as the security requirements become more stringent and the processing power becomes cheaper and easily available; ECC will definitely become a much more practical system to use.

Commonly, Elliptic Curve (EC) is defined over real numbers, complex numbers and other fields. Alternatively, the EC can also be clearly defined over finite fields. As been mentioned earlier, ECC offers the smallest key size and the highest strength per bit of any other known public key cryptosystem. In particular, binary polynomial field are considered as it allows fast computation on both hardware and software based implementations. An EC is defined in a standard, two dimensional x, y Cartesian coordinate system by the following equation : $y^2 = x^3 + ax + b$ [1]. The graph will turn out to be gently looping line. The graph can be seen in various forms. An example of EC graph is shown below in Figure 1.

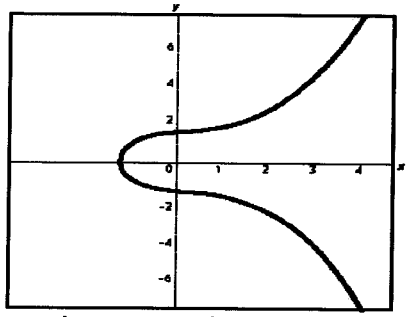

Fig 1.: An example of Elliptic Curve [1]

The capability of EC in providing computational efficiency on both public and private key operations with relatively small key size, makes ECC not only applicable to hosts secure protocols over wired networks but also to small wireless devices such as cell phones, PDAs and smart cards.

The most important operation for ECC is known as the scalar multiplication operation. It is defined as follows: Let n

be a positive integer and **P** a point on an EC. Then, the scalar multiple $\mathbf{Q} = k\mathbf{P}$ is the point resulting from adding k-1 copies of **P** to itself [1]. The security of ECC is indeed based upon the difficulty in solving the elliptic curve discrete logarithm problem. They are defined as follows: Given two point **Q** and **P** that belong to the curve, the idea is to find a positive scalar such that $\mathbf{Q} = n\mathbf{P}$ is true. This problem is referred to as the elliptic curve discrete logarithm problem in this scalar multiplication [6].

In the sense of implementing an ECC, several steps need to be taken into consideration. Firstly, we must decide the type of the underlying finite field. Secondly, we will have to take a look at the algorithms for implementing the basic algebraic operations. Next is to determine the type of the EC that are to be used as well as its generations, and finally is to decide on the EC protocols. The field that is usually taken for consideration is prime fields (denoted by $\mathbf{F}_p$ where p is a prime field). By selecting a prime field implies proper choice of p, since all basic operations will be modulo to that prime and the security of the system will depend on its size. The bigger the prime, the more secure but slower the cryptosystem will be. The basic algebraic operations of the EC group that must be implemented are points' addition and scalar multiplication on an EC [6].

III. RELATED WORK

A. Karatsuba Multiplication Algorithm

Karatsuba multiplication is also known as Karatsuba-Ofman multiplication algorithm which was developed by Anatolii Alekseevich Karatsuba and Ofman in 1962. The algorithm is a divide and conquer algorithm that multiply two numbers together with fewer operations than the normal grade school algorithm. The basis for Karatsuba multiplication is explained in the article presented in Everything$_2$.com [3]. The basic equation is as follows.

$$(a + bx2^m) \times (c+d \times 2^m) = axc + ((a+b) \times (c+d) - (a \times c) - (b \times d)) \times 2^m + (b \times d) \times 2^{2m}$$

Equation 1: Karatsuba multiplication [3]

From the equation, the $(\mathbf{a} + \mathbf{b} \times 2^n)$ will be referred to as the first number that need to be multiplied. This number will later be divided into two parts whereas the $(\mathbf{c} + \mathbf{d} \times 2^n)$ represents the other number which will also be divided in half. The most crucial aspect that needs attention is the normal multiplication algorithms. It happens to have $O(n^2)$ complexity; multiplying two n umbers together.

At the second half of the equation, the value can be computed only by using three multiplications of $(n/2)$ digit numbers. Since, we are only computing $\mathbf{a} \times \mathbf{c}$ and $\mathbf{b} \times \mathbf{d}$ once and this will reduce the number of multiplications that are required to $3 \times (n/2)^2$ as compared to n^2. The other subcomponent, $\mathbf{a} \times \mathbf{c}$, $(\mathbf{a} + \mathbf{b}) \times (\mathbf{c} + \mathbf{d})$, and $\mathbf{b} \times \mathbf{d}$ will also require some multiplications. Karatsuba indeed can be applied here recursively to perform the computation as well.

Karatsuba multiplication involves multiplications of large numbers. These algorithms are used in performing public key cryptography such as RSA, DSA and Diffie Hellman. RSA are generally referred as integer factorization cryptography (IFC) whereas DSA and Diffie-Hellman are generally referred as finite field cryptography (FFC). In addition, the primary operations existed in these algorithms are exponentiation, which can be implemented using a square and multiply algorithm.

B. Montgomery Multiplication Algorithm

This algorithm is used in order to compute point addition, point doubling and particularly EC point multiplication. The entire discussions on Montgomery algorithm is based upon the notation that are presented in [5].

Let P(x) be a degree-m polynomial, irreducible over $GF(2)$. Then, $\mathbf{P}(x)$ generates the finite field $\mathbf{F}_q = \mathbf{GF}(2^m)$ of characteristic two. A non-supersingular elliptic curve $\mathbf{E}(\mathbf{F}_q)$ is defined to be the set of points $(x, y) \in \mathbf{GF}(2^m) \times \mathbf{GF}(2^m)$ that satisfy the affine equation.Where a and $b \in \mathbf{F}_q$, $b \neq 0$, together with the point at infinity denoted by 0. Let $\mathbf{P} = (x_1, y_1)$ and $\mathbf{Q} = (x_2, y_2)$ be two points that belong to the curve 1. $\mathbf{P} + \mathbf{Q} = (x_3, y_3)$ and $\mathbf{P} - \mathbf{Q} = (x_4, y_4)$ also belong to the curve [4]. We only need the x coordinates of P, Q and $P - Q$ to exactly determine the value of the x-coordinate of the point $P + Q$. Let the x coordinate of **P** be represented by X=Z. Then the points $2P = (X_{2P}, Y_{2P}, Z_{2P})$ and $P + Q = (X_3, Y_3, Z_3)$ are the representation of their coordinates and can be computed as follows [7],

$$\begin{aligned} X_{2P} &= X^4 + b \cdot Z^4; \\ Z_{2P} &= X^2 \cdot Z^2; \\ Z_3 &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2; \qquad (3) \\ X_3 &= x \cdot Z_3 + (X_1 \cdot X_2) \cdot (X_2 \cdot Z_1); \end{aligned}$$

Input: $k = (k_{n-1}, k_{n-2}, \ldots, k_1, k_0)$ 2 with $k_{n-1} = 1$,
$P(x, y) \in E(F_{2^m})$
Output: $Q = kP$

1. Set $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
2. **For** i **from** $n-2$ **downto** 0 **do**
3. **if** ($k_i = 1$) **then**
4. Madd(X_1, Z_1, X_1, Z_2), Mdouble(X_2, Z_2)
5. **else**
6. Madd(X_1, Z_1, X_1, Z_2), Mdouble (X_1, Z_1)
7. **Return** ($Q = M_{xy}$ (X_1, Z_1, X_1, Z_2))

Fig. 2: Montgomery point multiplication [5]

The algorithm that is shown in Figure 2 will compute the elliptic point multiplication over $\mathbf{GF}(2^m)$ based on the previous discussion. The algorithm continues with the inclusion of point addition and point doubling.

C. *Multi-threading*

Multi-threading happens to be a specialized form of multi-tasking. The term multi-tasking can be separated into two types which are *process-based* multi-tasking and *thread-based* multi-tasking. For development purposes which include the ECC computations, we consider the thread-based multi-tasking. A *thread* is often referred to as dispatchable unit of executable code which is originally derived from the concept of "thread of execution." The *thread-based* multi-tasking ensures each process have at least one thread, however a process can even have more than one thread function. Eventually, it points out that a single program will be able to perform two or more tasks concurrently. This saves time and cause less overhead. The thread-based multi-tasking apparently handles the concurrent execution of pieces of the same program. In order to ensure the *thread-based* multi-tasking is taking place, multiple-CPU system is needed. This supports the initial plan for developing ECC computations on a multi-core processor. Multi-threading automatically change the architecture of a program. This is because the multi-threaded program executes portion of its program concurrently. Indisputably, this will support the concept of parallelism. The multi-threaded program will be responsible in managing the interactions among the threads as it adapts well to the concept of parallelism.

D. *Dual-Core Architecture*

As mentioned earlier, the implementation of the ECC computations will be tested on dual-core processor. Once the functions in the existing algorithm have been applied with threads, they are ready to be tested in dual-core processor. A dual-core processor revolves around an Integrated Circuit (IC) to which two processors have been attached. This is done to enhance the performance of the computing power which in return will reduce the power consumption as the circuit will not increase the heat.

A dual-core processor plays an important role by simultaneously processing multiple tasks in a more efficient manner. Dual-core processing is experiencing an extensive growth since the single-core processor is reaching the physical limits in term of speed and possible complexity. The dual-core processor contains only two independent processors. Multi-national companies have ventured into the development and research of dual-core-core processor and have successfully developed them and made it available in the open market.

IV. FRAMEWORK FOR THE PROPOSED SYSTEM

The proposed framework (see Figure 3) is the guideline in developing the software implementation of the multi-threaded ECC. The framework provides a smooth flow of what needs to be done. The ECC consists of many steps. Among the most important ones is the point multiplication (implemented by multi-threaded Karatsuba and multi-threaded Montgomery). At the point multiplication step, we add either the Karatsuba or Montgomery multi-threaded functions. This step will allow multi-tasking which will enable more processing to take place. Besides that, faster ECC operation means reduced security and longer key size for the same level of security. The results obtained from running the functions will then be brought back to the ECC as shown by Figure 3.

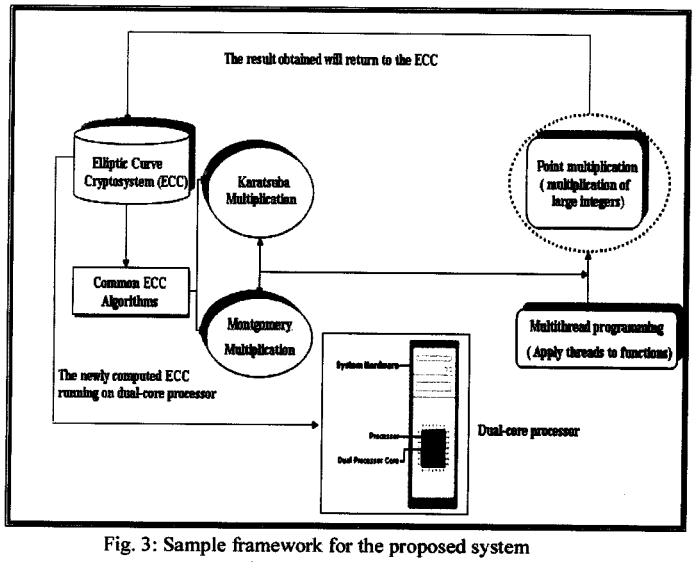

Fig. 3: Sample framework for the proposed system

V. IMPLEMENTATION

This section discusses on the software implementation of the ECC multiplications using two different multi-threaded mathematical algorithms. The implementation is done using Microsoft Visual C++ 6.0 in Windows environment with multi-threading programming support.

A. *Implementation of EC Multiplication*

The implementation of EC multiplication is based upon the scalar multiplication, $\mathbf{Q} = k\mathbf{P}$ where the value of point **Q** is obtained by the multiplication of scalar k and point **P**. The multiplication process is represented as the repeated addition process. The implementation starts of by defining the EC. For this development purposes we have decided to use EC defined over prime fields $\mathbf{GF}(p)$ as it is easier to be developed. The implementation steps are shown in Figure 4. As mentioned earlier, the two main algorithms used are Karatsuba and Montgomery where each algorithm will run independently with the existing EC functions.

B. *Implementation of EC Multiplication Using Karatsuba Algorithm*

The implementation process is similar to EC multiplication; however the difference here is that a new mathematical approach is used in designing the EC multiplication. The code development is done by referring to the original Karatsuba code and original EC multiplication. The differences in the implementation of the code are as follows:

- The base point $\mathbf{P}(x,y)$ will now be divided into two parts which will be represented as $\mathbf{P}(x)$ and $\mathbf{P}(y)$. $\mathbf{P}(x)$ will

represent the first multiplicand while **P(*y*)** will represent the second multiplicand.

- Next, we introduced point **r [6×A_SIZE]** which will be providing the multiplication of the points at the later stage.
- Besides that, we assigned the maximum length of each base point. ***k*** will be set as the maximum length thus resulting into ***k*_P(*x*)** represents the maximum length of base **P(*x*)** and ***k*_P(*y*)** represents the maximum length of base **P(*y*)**.

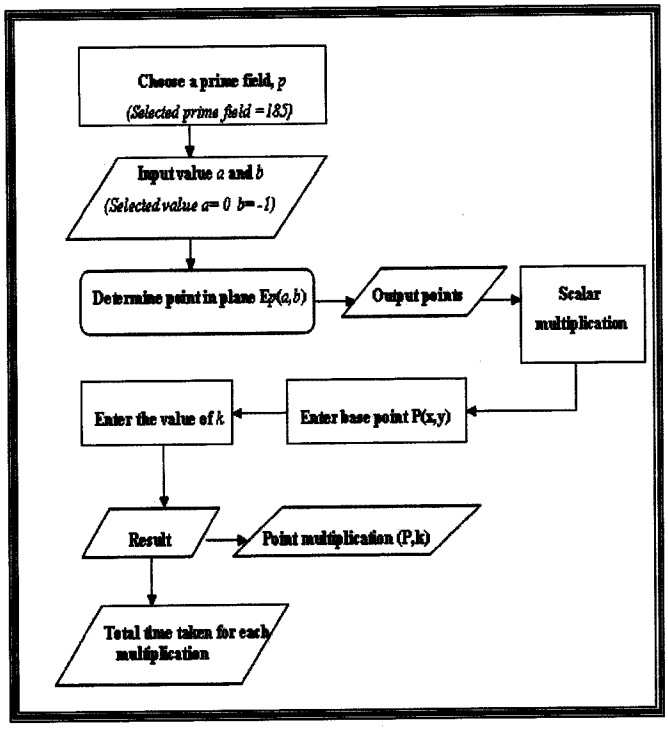

Fig. 4: Flow chart representing the EC multiplication process.

The implementation of EC using the Karatsuba (see Figure 5) is rather difficult as the main operation of dividing the base point **P(*x*, *y*)** had to be carefully coded together with the maximum length of each base point represented as ***k***. This is done as we do not want to add any complexity to the existing code that constitutes the EC multiplication. Furthermore, we had to alter the functions mentioned earlier so that the scalar multiplication process can take place smoothly. We will obtain two results where one will represent the point multiplied and the other is the time taken for the operation to take place.

C. Implementation of EC Multiplication Using Montgomery Algorithm

We developed the code by referring to the original Montgomery code and original EC multiplication. The modifications made in the implementation code are as follows:

- Introducing the *double_point* function. By doing so, we can see that the complexity of the *add_point* functions reduced thus enabling the EC to perform the scalar multiplication operations in a much shorter time.
- Introduction of element ***Z*** that represent the *mod* value as the basic multiplication of Montgomery.
- The *mod* value is attached to the scalar ***k*** in the original EC multiplication. The mod operation eventually speeds up the entire multiplication process.
- The *double_point* function has reduced the number of points required for repeated addition process. Similarly, the main function will also be changed as follows and int ***k*** will be changed to be mod ***k***.

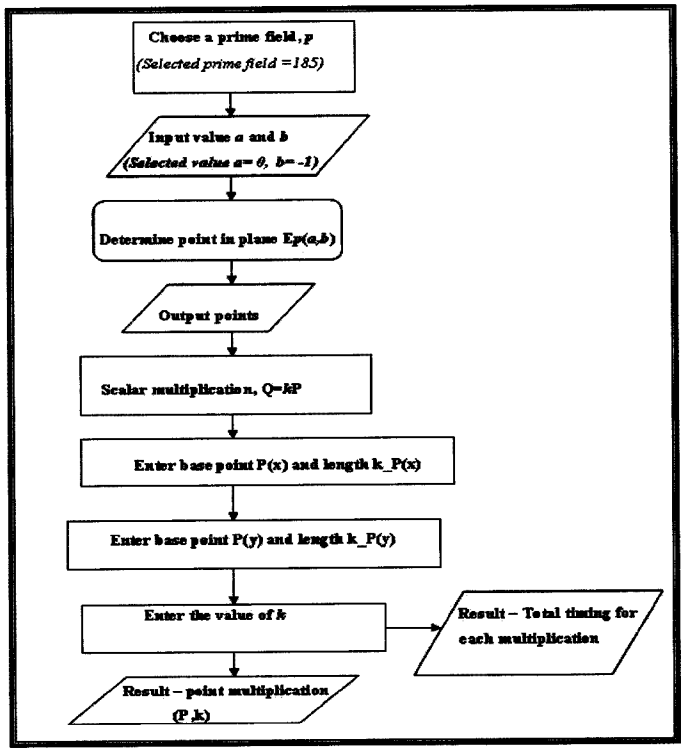

Fig. 5: Flow chart representing the EC multiplication using Karatsuba approach.

D. Multi-Threaded Implementation of EC Multiplication Using Both Karatsuba and Montgomery Algorithm

The multi-threading codes written for both Karatsuba and Montgomery are similar. The steps involved that have been taken in writing the multi-threading codes are as follows:

- First, we create a new application by selecting > *Win32 Console App* and this file is publicly available.
- Next, we add in the C++ source code where we want to add the thread function. The focus was to multi-thread the time for both EC multiplication using Karatsuba and Montgomery approach.
- After adding the codes, a common header file known as *windows.h* which is a publicly available file is added since it is essential in order to perform multi-threading in Windows.
- At the *Project* column, we must select *Settings*, thus will show us a new panel known as *Project Settings*. On the hand, at the left side of the panel, a tab known as *C/C++*

can be seen. Once we click on it, at the *Category* tab, we must choose for *Code Generation*. Later, to ensure that multi-threading process are taking place, we change the *Use run time library* to *Debug Multi-threaded*.

- Once this is done, the code will be multi-threaded but this function represents only the basic of multi-threading in Windows.
- The thread functions have been divided into two functions. The first thread function represented as Thread 1 will perform until the addition of two points. The Thread 2 function will initiate once we enter the scalar multiplication process.
- We declare a new header declarations into the existing ones known as *long* WINAPI *Thread Two*(*long IParam*).
- At the main functions, we create a function called HANDLE *hThread* which will be responsible in handling the thread functions.
- The multi-threading code for the EC multiplication using the two mathematical approaches are placed under the function *long* WINAPI *ThreadTwo*(*long* Param).

VI. TESTING AND RESULT

Here, we present the results obtained from the execution of different EC multiplication codes. All the codes were tested on the same machine. The specification of the machine used for the testing purposes is AMD Athlon 64x2, 2.4GHz Dual Core Processor using Microsoft Windows XP Professional Edition and for testing on single-core we switched off the dual-core processor and turned on the single-core processor. The times are calculated in millisecond (ms) and are presented in Table 1. Based upon the result presented at Table 1, The Karatsuba algorithm is indeed faster than the original EC multiplication but as for Montgomery it is much faster than both original EC multiplication and Karatsuba algorithm. The maximum bit tested here are 192-bits.

TABLE I
COMPUTATION TIME TESTED ON BOTH SINGLE-CORE AND DUAL-CORE PROCESSOR

Size (bits)	Timing on original EC multiplication (*ms*)	Total timing on single-core processor (*ms*)		Total timing on dual-core processor (*ms*)	
		EC + Karatsuba	EC + Montgomery	EC + Karatsuba	EC + Montgomery
32	4.969	4.391	*4.000*	3.203	3.187
64	11.781	10.390	*9.160*	7.187	6.219
96	20.375	17.860	*15.984*	13.203	12.219
128	39.813	34.250	*29.969*	18.203	16.219
160	51.406	45.125	*41.969*	28.188	26.234
192	91.984	81.407	72.969	47.203	46.234

ms = milliseconds

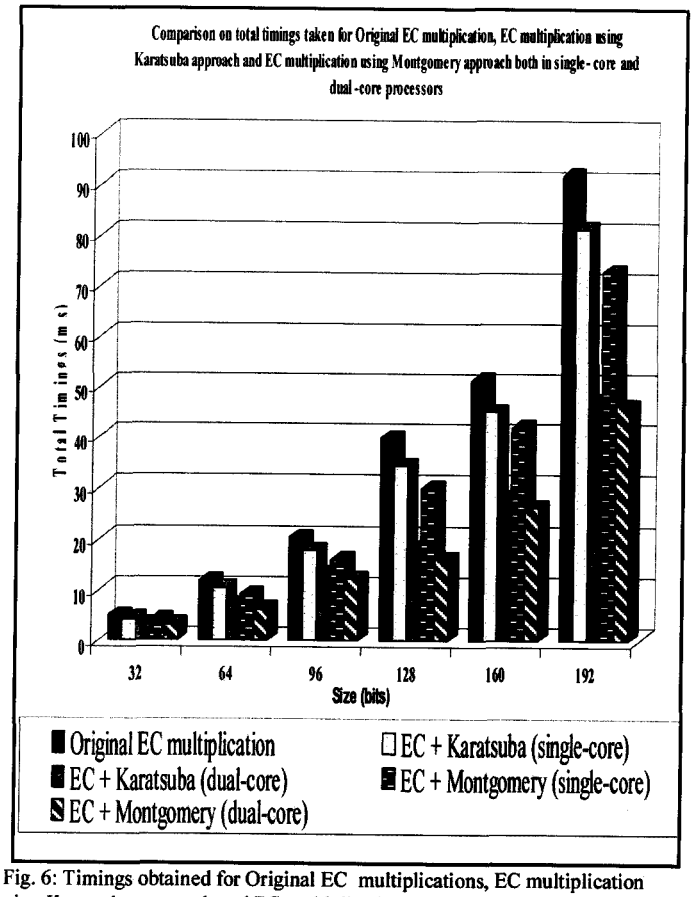

Fig. 6: Timings obtained for Original EC multiplications, EC multiplication using Karatsuba approach and EC multiplication using Montgomery approach on both single-core and dual-core processor.

VII. CONCLUSION

The implementation of ECC in multiprocessing environment revolves around numerous tasks and steps. Most importantly, we must understand the fundamentals underlying beneath the EC as it ensures the smooth development to take place. The inclusion of these algorithms has reduced the total computation time where these mathematical algorithms tackle the multiplication process in parallel, in its own way and speeds up the overall process. As an example, Karatsuba multiplication is typically slower than long multiplication but it happens so that generally Karatsuba becomes faster than the classical algorithm when reaching upon numbers over 2^320. Karatsuba is indeed a special case of Toom-Cook multiplication as it also focuses on multiplying two large integers but due to some circumstances this multiplication algorithm is slower than the long multiplication with small numbers instead Toom-Cook is best to be used for intermediate size multiplication. In contrast, the Schönage-Strassen algorithm happens to be an asymptotically fast method known in multiplying large integers. This algorithm was written to outperform the older multiplication algorithm such as Karatsuba but for this implementation our focus was entirely on Karatsuba and Montgomery multiplication algorithm. The inclusion of multi-threading was entirely to parallelize the two algorithms and

allowing these processes to be tested using different machines. One of the main reasons was to look at the overall performance between the single-core and the dual-core processors. At the end, the parallelism of this EC multiplication was enhanced. Subsequently, the parallel multiplication has also reduced the total computation time up to fifty percent. Based upon the timings that have been obtained, Montgomery algorithm are better to be used with the original EC multiplication.

REFERENCES

[1] Certicom. 2004. An Intro to Elliptic Curve Cryptography. www.deviceforge.com/articles/.html.

[2] Koblitz, N.1987. Elliptic Curve Cryptosystems. *In Mathematics of Computation* 48(177) : 203-209. In addition to Miller ,V. 1985. Use of elliptic curves in cryptography. *CRYPTO 85.*

[3] Montecarlo. 2005. Karatsuba Multiplication (idea). Everything$_2$.com. http://everything2.com/index.

[4] Multithreading in C++. 2005. Contributed by McGraw-Hill and Osborne. http://www.devarticles.com/c/a/Cplusplus/ Multithreading.

[5] N.A. Saqib, F. Rodriguez-Henriquez and A. Díaz-Pérez. A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over GF(2^m). *IEEE Transactions.* 2004 *Parallel and Distributed Processing Symposium.* Proceedings 18th International.

[6] P``uhringer, C. 2005. High Speed Elliptic Curve Cryptography Processor for GF(p). http://jcewww.iaik.at/sic/layout/set/print/content/download/713/5324/file /2005_ECC_Puehringer.pdf

[7] Wolkerstorfer, J. *Hardware Aspects of Elliptic Curve Cryptography.* PhD dissertation, Graz University of Technology, Austria, Institute for Applied Information Processing and Communications, July 2004.