An Optimal Software Framework for Parallel Computation of CRC

Hamed Sheidaeian[1] and Behrouz Zolfaghari[2]

[1]Department of Engineering, Islamic Azad University, Garmsar Branch, Iran
sheidaeian@ut.ac.ir
[2]Department of Engineering, Islamic Azad University, Garmsar Branch, Iran
zolfaghari@aut.ac.ir

ABSTRACT

CRC is a common error detection method used in different areas such as information storage and data communication. CRC depends on modulo-2 division by a predetermined divisor called the generator. In this method, the transmitter divides the message by the generator and concatenates the calculated residue to the message. CRC is not able to detect every kind of errors. The properties of the generator determine the range of errors which are detectable in the receiver side. The division operation is currently performed sequentially, so developing methods for parallel computation of the residue makes CRC suitable for network protocols and software applications. This paper presents a novel software framework for parallel computation of CRC using ODP polynomials.

KEYWORDS

Parallel CRC computation, ODP polynomial, OZO generator

1. Introduction and Basic Concepts

CRC is a widely adopted method for detecting errors which is used in various systems. Applications of CRC [4-8] as well as developing methods for improving its efficiency [9-11] have been research focuses in recent years. Before discussing the CRC method and introducing the proposed approach, some definitions and basic concepts are needed which they are shortly presented in the following.

Polynomial: is a notation for a bit string in which the positions of *1s* a shown by the exponents of a variable such as x . In other words, a polynomial appears in the form of $\sum a_i x^i$. Each a_i can be *0* or *1* and is equal to the corresponding bit in the string. Each x^i shows that the corresponding bit has been located in the position i of the string. For example the bit string *1101* can be shown by the polynomial $x^3 + x^2 + 1$. The degree of a polynomial is equal to the largest exponent of x in the polynomial. A polynomial of degree m is equivalent to a string than consists of length $m + 1$ bits. Every polynomial of an even degree is referred to an even polynomial in this paper. Similarly polynomials having odd degrees are called odd polynomials.

Modulo-2 bitwise addition and subtraction: are both equal to logic XOR. Modulo-2 addition and subtraction generate no carry or borrow. Thus, the addition/subtraction of two strings is simply accomplished by adding/subtracting corresponding bits of the strings.

Modulo-2 bitwise multiplication: is equal to logical AND. String multiplication is accomplished by bitwise multiplications, shifts and string additions. Especially Multiplying a bit string like s by 2^n (or equally x^n) means concatenating n zero bits to the right end of the string. Therefore, $s.2^n$ is called a *shifted multiple* of s for every value of n .

Modulo-2 division: is often explained in terms of a polynomial division method called long division which imitates consecutive modulo-2 subtractions. In other words, modulo-2 division is accomplished by subtracting the dividend by multiples of the divisor until the degree of the residue is less than that of the divisor.

Generator: is a predetermined unique string (polynomial) used as the divisor by both the transmitter and the receiver. The generator plays a key role in the error detection capability of the CRC method. For example, a generator containing an even number of *1s* can detect every error whose vector contains an odd number of *1s.*

Above concepts and definitions are essential for the reader to understand the rest of this paper, so they were discussed briefly. Readers are referred to [12] for more details regarding these concepts.
In a system which uses CRC for error detection, the transmitter adds a redundancy code to the end of the message which can be used by the receiver in order to determine whether the message has been changed during the transmission. The length of the code varies from one system to another. Especially the Ethernet protocol uses a 32-bit CRC 9.The process of using an n-bit CRC is as follows:

- The transmitter concatenates *n* zero bits to the end of the message M (converting it to $M * 2^n$). Next, it divides the whole string (including the message as well as the added zero bits) by the predetermined generator (G) which is *n+ 1 bit* long and calculates the residue ($(M * 2^n)\,Mod\ G$). In the next step, the transmitter inserts the n-bit residue of the division (R) in place of the zero bits and transmits the result string which will be equal to $MR = M * 2^n \pm (M * 2^n)\,Mod\ G$. Figure 1 demonstrates these steps.
- The receiver on the other hand, divides the received string (MR) by the generator and calculates the residue again. If the transmitted string does not change while passing through the channel, the residue here will be equal to $(M * 2^n \pm (M * 2^n)Mod\ G)Mod\ G = 0$ (The notation $\pm$ has been used to emphasize the fact that addition and subtraction are the same in modulo-2 arithmetic). Thus, if the residue is not equal to zero, the receiver considers an error. In fact, if an error occurs during the transmission, the corresponding error vector (E) will be added to MR . In this case, the receiver will obtain $(MR + E)Mod\, G = E\, Mod\, G$ instead of zero.

One of the shortcomings of CRC is that it cannot detect all types of errors. To make this point more clear, suppose that E is a multiple of G . In this case the residue calculated in the receiver side will be equal to $E\ Mod\ G = 0$. This misleads the receiver to assume that there has been no error. In fact, if the error vector is a multiple of the generator, the receiver cannot detect the error. To mitigate this problem, electrical and mechanical properties of the channel should be analyzed carefully as well as the environmental noises. This way, the dominant types of error can be determined and calculated the corresponding error vectors. Now a generator should be selected which does not have any multiples equal to the dominant error vectors.

In this paper a software framework is proposed for accelerated computation of CRC using OZO-based generators. The rest of this paper is organized as follows. Section 2 presents Preliminary discussions about OZO generators and parallel computation of CRC using them, section 3 explains a traditional software framework for sequential computation of CRC. The proposed accelerated framework is discussed in section 4. Section 5 is dedicated to finding OZO generators in the framework .Section 6 shows experimental results and finally section 7 presents conclusions and suggests further works.

2. Preliminary discussions about parallel CRC computation

2.1. OZO Generators

CRC is traditionally computed by serial circuits called LFSRs (Linear Feedback Shift Registers). An LFSR is a special kind of shift register in which the output of the last flip flop is fed to the input of the first flip flop through a number of XOR gates. This paper utilizes a novel method for parallel computation of CRC using mathematical properties of a special category of generator polynomials called ODPs (OZO Dividing Polynomials). ODPs are polynomials having multiples of form 100…001. The latter form of polynomials is called OZO (One-Zero-One).It is demonstrated that if the generator is selected from this category, the CRC can be calculated by parallel circuits with minor hardware requirements. B.Zolfaghari and H.Sheidaeian [1, 2 & 3] introduced OZOs and ODPs. They developed a systematic method for constructing ODP polynomials. This method has been used in proposed software framework for computing OZO-based polynomials.

A burst Error is defined in their works as an error which flips a large number of consecutive bits. The vector of such an error will have such a form: *00…011…100…0*. Such strings are called *ZOZ (Zero-One-Zero)* strings in this paper. A ZOZ string consists of three substrings; an all-zero substring in the left side, an all-1 substring in the middle and another all-zero substring in the right side. The all-1 substring of a ZOZ string is referred as an ALO (All One) string. Every polynomial that has an ALO multiple also is referred as an ADP (ALO Dividing Polynomial). Especially, every ADP of degree 32 is called an ADP32. Every ALO polynomial of an even degree is called an even ALO polynomial and every ALO polynomial of an odd degree is called an odd ALO polynomial.

ZOZ strings can be equivalently shown by ZOZ polynomials which contain a set of consecutive exponents of x like $\sum_{i=m}^{n} x^i = x^m(\sum_{j=1}^{n-m} x^j + 1)$ where n is the degree of the polynomial and m is the number of 0s in the right side of the string. The sum $\sum_{j=1}^{n-m} x^j + 1$ (or equally $\sum_{j=0}^{n-m} x^j$) shows the polynomial form of the ALO substring.

An OZO (One-Zero-One) string is one which contains a single *1* in the left side, a number of consecutive *0s* in the middle and another single *1* in the right side (like this: 100…001). The equivalent polynomial form of an OZO string is like $x^n + 1$.

2.2. Analytical Discussions

In order to calculate CRC using the proposed parallel algorithm, a software program should be developed which generates all possible 32-bit strings S and concatenate 32 zero bits to each S to make $S.2^n$ .In the next step, $S.2^n$should be divided into G to calculate the remainder R and then each S should be stored in a file along with the corresponding R.

The next required program should take an input string M and repeat the following steps until there remains 32 bits from M as the final remainder.

- Take 32 bits from the left of M.
- Lookup the taken 32-bit string among S strings and find the corresponding R.
- XOR R with next 32 bits and truncate the first 32 bits from M.

The above programs aim at totally parallel calculation of CRC using a table-driven approach. Suppose a dividend M is to be divided against a divisor G. The goal of such a division is to permanently subtract multiples of G from M and turn left side bits of G to 0 reduce the length of M. The division is finished when the length of M is smaller than that of G. at the end of this process, M itself will be the remainder. If G is n+1 bits long, the length of the final remainder will be equal to n. In this approach, a special multiple of G is found in each step which is 2n bits long and has n bits equal to M in the left. Thus in each step, n bits of M turn to zero instead of just one bit. This multiple of G named G’’. In fact G’’ has an n-bit left prefix exactly equal to M and since the modulo-2 operation is the same as XOR operation, each step of the division works like shown in the figure 1.

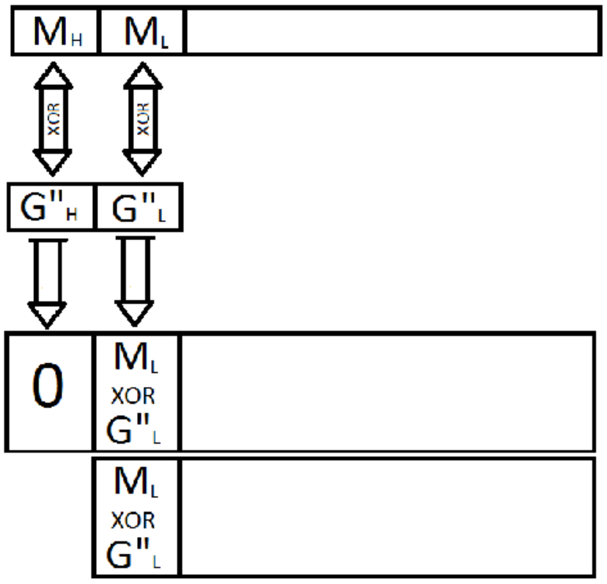


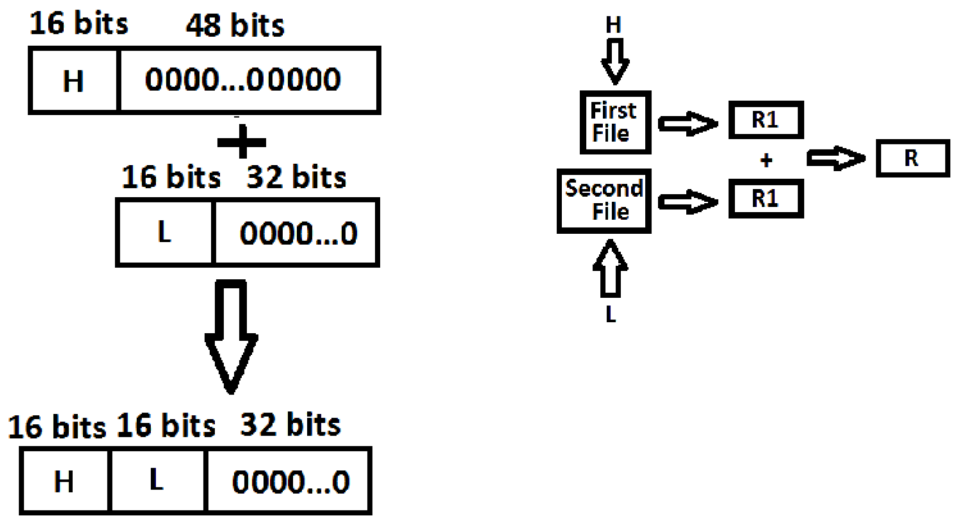
Figure 1. Schematic of a single step of the algorithm

As seen in the above image, this algorithm depends on parallel conversion of n bits of M to zero in each step instead of converting a single bit. In each step n bits from the left of are truncated and the next n bits (M_L) are XORed with n least significant bits of G’’ (G''_L). Note that M_L, M_H , G''_L and G''_H are each n bits long.

In order to calculate G’’ from G, n zero bits should be concatenated to M_H and get $M_H . 2^n$. Then $M_H . 2^n$ should be divided against G and remainder R should be gathered. if R is subtracted from $M_H . 2^n$, a string will be reached which is obviously divisible to G. This multiple of G is in fact $M_H R$ which is named G’’.

Now the division can be performed with G’’ as a multiple of G instead of G itself. To do this, all possible n-bit strings *M* should be created and stored in a table along with the corresponding $M_H . 2^n \, Mod \, G$. In each step, an n-bit prefix M is truncated from M and next n-bit is XORed with $M_H . 2^n \, Mod \, G$. The string $M_H . 2^n \, Mod \, G$ should be looked up in the table.

If this approach is to be applied to 32-bit CRC, 2^{32} strings should be calculated each 32 bits length and each of them should be stored in a table with the corresponding 32-bit remainder. Thus 2^{32} division operations will be needed and the size of the table will be equal to $2^{32} \times (32 + 32) = 2^{68}$, but in order to reduce the number of division operations and the size of the table, 32-bit strings like M can be generated and stored along with the 32-bit string $M_H . 2^{48}\ Mod\ G$ in the table. The figure 2 shows the way this is done.



a. b.

Figure 2. Computing mid-sums

In this approach another table containing 16-bit M strings will be needed along with the corresponding 32-bit remainders $M_H . 2^{32}\ Mod\ G$. The total size of the two tables will be equal to $2 \times 2^{16} \times (2^{16} + 2^{32}) = 2^{17} \times (2^{16} + 2^{32}) = 2^{33} + 2^{48}$ which shows an obvious reduction. Also the number of required division operations will be equal to $2 \times 2^{16} = 2^{17}$ which exhibits an obvious reduction again. But each step of this approach includes to cycles and two XOR operations. The choice between the two approaches depends on the most important parameters. In fact there is a trade-off between the size of the table and the static calculations in one hand and the dynamic run-time calculations in other hand. This trade-off should be resolved by the designer.

3. A Software Framework for Regular CRC Computation

The main function which used for CRC computation is ***GenerateCRCResidue****:*

```
void GenerateCRCResidue(string data, string CRCGen) {
   string zero = InitBinary(32);
   string pattern = data + zero;
   string res[] = DivideModulo2(pattern,CRCGen);
   string R = res[1];
   return R;
}
```

In this function CRC generator is stored in *CRCPoly* variable. The function calculates CRC using returned residue value of a Modulo-2 division function. Some extra subroutines are used in above function as follows.

InitBinary function initializes an n-bit string-based binary number with zero value. ***Increament*** function takes one string based binary parameter and returns that value plus one. ***ConvertToBinary*** function is used for converting an integer number to equivalent N-bit binary string with alignment of N.

DivideModulo2 function calculates modulo-2 binary division.

```
string[] DivideModulo2(string a, string b) {
      string[] res = new string[2];
      if (a.Length < b.Length) {
            res[0] = "0";
            res[1] = a;
      }
      else {
            ArrayList mid_quotients = new ArrayList();
            string tempa = a;
            while (tempa.Length>=b.Length) {
                string tempb = b;
                string tempquot = "1";
                int len_dif = tempa.Length-b.Length;
                for (int i=0;i<len_dif;i++) {
                   tempquot += "0";
                   tempb += "0";
                }
                mid_quotients.Add(tempquot);
                tempa = SumModulo2(tempa,tempb);
                if (tempa.Length>=b.Length)
                   tempa = RemoveZeroFromLeft(tempa);
                if (tempa.Length<b.Length) {
                   string t = tempa;
                   for (int i=0;i<b.Length-t.Length-1;i++)
                         tempa = "0"+tempa;
                }
           }
           string quotient = "0";
           for (int i=0;i<mid_quotients.Count;i++)
                SumModulo2(mid_quotients[i].ToString(),quotient);
           res[0] = quotient;
           res[1] = tempa;
   }
   return res;
}
```

SumModulo2 function computes modulo-2 addition. *Abs* method returns the absolute value of a number and *Convert.ToByte* method converts the specified string representation of a number to an equivalent 8-bit unsigned byte. *RemoveZeroFormLeft* function eliminates extra zeroes from left side of a string-based binary number.

```
string[] SumModulo2(string a, string b) {
      string tempa = a;
      string tempb = b;
      int len_dif  = Abs(a.Length-b.Length);
      if (a.Length>b.Length) {
            for (int i=0;i<len_dif;i++)
                  tempb = "0"+tempb;
      }
      else if (a.Length<b.Length) {
            for (int i=0;i<len_dif;i++)
                  tempa = "0"+tempa;
      }
      string result = "";
      for (int i=tempa.Length-1;i>=0;i--) {
            byte bita = Convert.ToByte(tempa.Substring(i,1));
            byte bitb = Convert.ToByte(tempb.Substring(i,1));
            byte bit_xor = (byte)(bita ^ bitb);
            result = bit_xor.ToString() + result;
      }
return result;
}
```

```
string RemoveZeroFromLeft(string x) {
      string res = "";
      int i = 0;
      for (i=0;i<x.Length;i++) {
            byte bit = Convert.ToByte(x.Substring(i,1));
            if (bit==1) break;
      }
      Res = x.Substring(1);
return res;
}
```

4. Proposed Framework for Accelerated Computation of CRC

Sequential schema of proposed *CalculateCRC* subroutine is shown in next page. In this pseudo-code input data is fragmented to 32-bit segments and *DivideModulo2* is performed on each segment to retrieve residue value as partial CRC. Finally XOR logic operation is used between these partial CRCs for computing final CRC value.

The input data can be divided to 32-bit segments and multithread programming features of .Net can be used for parallel and fast computation of partial CRCs instead of this sequential schema. *XOR* helper function performs modulo-2 addition between its first argument and 32 least significant bits of its second argument, then concatenates this intermediate value to 32 most significant bits of the second arguments and finally returns this result.

```
string CalculateCRC(string data, string CRCPoly) {
      string temp = data;
      string zero = InitBinary(32);
      for (int i=0;i<(int)(data.Length/32)-1;i++) {
            string split = temp.Substring(0,32);
            string data = split+zero;
            string[] res = DivideModulo2(data, CRCPoly);
            string R = res[1];
            temp = temp.Substring(32);
            temp = XOR(R,temp);
      }
return temp;
}


string XOR(string s, string main) {
      string subMain = main.Substring(0,32);
      string res = SumModulo2(subMain,s)+main.Substring(32);
      return res;
}
```

5. Finding OZO-based Generators

There are five steps for generating all 32 bits OZO-based polynomials can used for parallel computation of CRC. Figure 3 shows these steps.

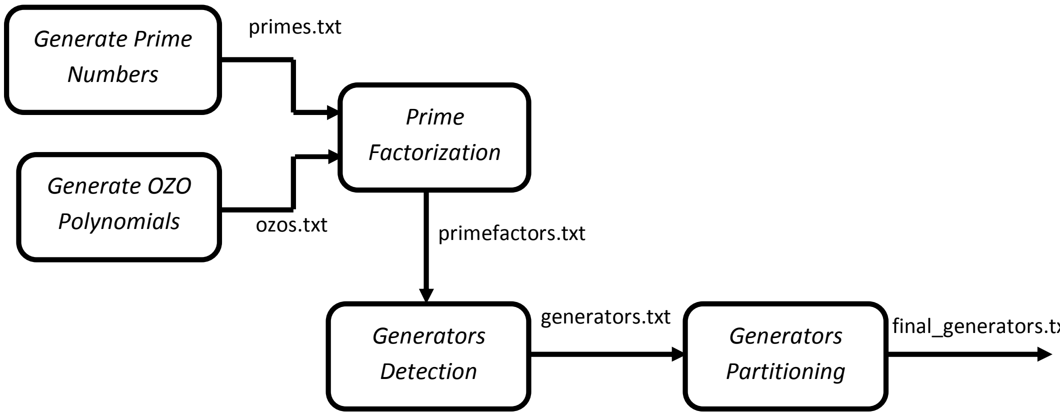


Figure 2. Block diagram of *finding OZO-based generators* subsystem

5.1 Generating Prime Numbers

First of all a global *ArrayList* variable is needed for preserving all 32-bits prime numbers. So in first step 32 bits prime numbers should be computed using sieve of Eratosthenes algorithm. Result prime numbers are stored in a text file using a special *StreamWriter* object that implements a text writer for writing strings to an I/O stream in a particular encoding.

```
void CreatePrimNumbers(string main, string s) {
      //sw : prime numbers
      PrimNumbers.Add("11");
      sw.WriteLine("11");
      for (int i=4;i<=Int32.MaxValue;i++) {
            string binary_val = ConvertToBinary(i);
            int count = CountOfCharInString(binary_val,'1');
            // i must not be even and the count of 1s in binary
                          val must be odd
            if ((i&1)==1 && (count&1)==1) {
                  bool isPrime = true;
                  for (int j=0;j<PrimNumbers.Count;j++) {
                        if((Floor((decimal)(binary_val.Length/2
                         +1)<PrimNumbers[j].ToString().Length)
                              break;
                        string[] div = DivideModulo2(binary_val,
                             PrimNumbers[j].ToString());
                        if (IsZero(div[1]) {
                              isPrime = false;
                              break;
                        }
                  }
                  if (isPrime==true) {
                        PrimNumbers.Add(binary_val);
                        sw.WriteLine(binary_val);
                  }
            }
      }
      sw.Close();
}
```

Floor method returns the largest integer less than or equal to the specified number. Some other helper functions are used in above code. *CountOfCharInString* function counts repetition number of a character in a string and *IsZero* function determines if a string variable is a binary zero or not.

5.2 Generating OZO Polynomials

In this step all OZO polynomials is generated easily and stored in a text file.

```
void CreateOZO() {
      //sw : ozos
      sw.WriteLine("11");
      for (int i=0;i<=128;i++) {
            string tmp = "1";
            for (int j=0;j<=I;j++)  tmp += "0";
            tmp += "1";
            sw.WriteLine(tmp);}
      sw.Close();
}
```

5.3 Prime Factorization

In prime factorization step all previous generated data (32 bits prime numbers and OZO polynomials) are read by two special *StreamReader*s object. This object implements a text reader that reads strings from an I/O stream in a particular encoding. In the next step a loop structure is used for modulo-2 dividing each OZO polynomial by all prime numbers. If an OZO polynomial is divisible by a prime number (all bits of string-based residue should be zero), prime number will be stored in a text file and replace current OZO value by quotient of division for examination using next prime number.

```
void PrimeFactorization() {
      //sw : primefactors
      string tmp = sr_prime.ReadLine();
      ArrayList primes = new ArrayList();
      ArrayList ozos = new ArrayList();

      while (tmp!=null) {
            primes.Add(tmp);
            tmp = sr_primes.ReadLine();
      }

      sr_primes.Close();
      tmp = sr_ozos.ReadLine();

      while (tmp!=null) {
            ozos.Add(tmp);
            tmp = sr_ozos.ReadLine();
      }
      sr_ozos.Close();

      for (int i=0;i<ozos.Count;i++) {
            string result = "";
            string ozo = ozos[i].ToString();
            for (int j=0;j<primes.Count;j++) {
                  if (ozo.Length>=primes[j].ToString().Length) {
                        String pr = primes[j].ToString();
                        string[] res = DivideModulo2(ozo,pr);
                        if (!res[1].Contains("1")) {
                              ozo = res[0];
                              result += pr+" ";
                              j--; countinue;
                        }
                  }
                  else break;
            }
            sw.WriteLine(result);}
      sw.Close();
}
```

5.4 Generators Detection

The prime factors generated in previous step (the set of prime divisor for each OZO string) are used for generating power set for them. Then for each subset exist in this power set, all divisors are multiplied together. If the length of this result is 33 bits, it should be added to generators file. *Split* method in related pseudo code returns a string array that contains the substrings in this string that are delimited by a specified character.

An example in integer domain can explain the operation:
Assume that we want to find all numbers with a multiple equal to '30'. So we should find all prime divisors of 30 first (prime decomposition of 30): 30 = 2 * 3 * 5
$PF(30) = \{2,3,5\}$
After that, the power set of PF(30) is:
$PS(PF(30)) = \{\{\ \}, \{2\}, \{3\}, \{5\}, \{2,3\}, \{2,5\}, \{3,5\}, \{2,3,5\}\}$

Now if we multiply all members in each subset together, we can produce a set of numbers which '30' is a multiple of them.
$D(30) = \{2,3,5,6,10,15,30\}$

5.5 Generators Partitioning

If an k-bit OZO polynomial is presented by OZO(k), it can be easily shown that OZO(2k-1) is a multiple of OZO(k). The intersection of OZO(k) and OZO(2k-1) divisor sets (which are produced in previous stage) definitely is not empty. In the set theory, the intersection of two sets A and B is the set that contains all elements of A that also belong to B, but no other elements. So the union of divisor sets (retrieved from OZO polynomials in previous step) is computed and duplicate members are removed. The union of two sets is the set of all distinct elements in them.

```
void DistinctGenerators() {
     // sw :final_generators
     // sr :generators
     ArrayList gen = new ArrayList():
     String tmp = sr.ReadLine();
     while (tmp!=null) {
          string[] factors = tmp.Split(' ');
          for (int i=0;i<factors.Length;i++) {
               if (!gen.Contains(factor[i])) {
                    gen.Add(factors[i]);
                    sw.WriteLine(factors[i]);
               }
          }
          tmp = sr.ReadLine();
     }
     sw.Close();
     sr.Close();
}
```

```
void DetectGenerators() {
// sw : generators
// sr : primefactors
String tmp = sr.ReadLine();
int k = 0;
while (tmp != null) {
k++;
string[] factors = tmp.Split(' ');
ArrayList generator = new ArrayList();
for (int i=0;i<factors.Length;i++) {
if (!generator.Contains(factors[i]) &&
factors[i].Length=33)
generator.Add(factors[i]);
}
string pattern = InitBinary(factors.Length);
while (pattern.Contains("0")) {
pattern = Increament(pattern);
string multiply = "1";
int len_sum = 0;int count = 0;
for (int j=0;j<factors.Length;j++) {
if (pattern.Substring(j,1)== "1") {
len_sum += factors[j].Length;
count++;
}
}
len_sum -= count-1;
if (len_sum==33) {
for (int j=0;j<factors.Length;j++) {
if (pattern.Substring(j,1)== "1")
multiply = MultiplyModulo2(
multiply,factors[j]);
}
if (!generator.Contains(multiply))
generator.Add(multiply);
}
}
tmp = sr.ReadLine(); string gen = "";
for (int i=0;i<generator.Count;i++)
gen += generator[i].ToString()+" ";
sw.WriteLine(gen);
}
sr.Close(); sw.Close();
}
```

6. EXPERIMENTAL RESULTS

Sequential *GenerateCRCResidue* method and proposed *CalculateCRC* are implemented in C# 3.0 using Visual Studio 2008. Multithreading is used for parallel implementation of *CalculateCRC*. Parallel code is executed on Intel Core i5 CPU. Table 1 shows the average execution time (in millisecond) needed for computing CRC for all possible 32-bit input strings (all possible 2^{32} values) using both sequential and parallel schemas. Elapsed computation time is calculated using StopWatch class in C#. This table and related figure 4 show that the multithreaded parallel computation of CRC code works faster than its corresponding sequential code.

Table 1. Execution time for sequential and parallel CRC computation of all 32-bit input values

Input String Length (bit)	**# of threads**	**Execution time (Sequential)**	**Execution time (MT)**
320	2	47	24
	4		12.8
1600	2	1774	934
	4		492
6400	2	65892	36204
	4		18934

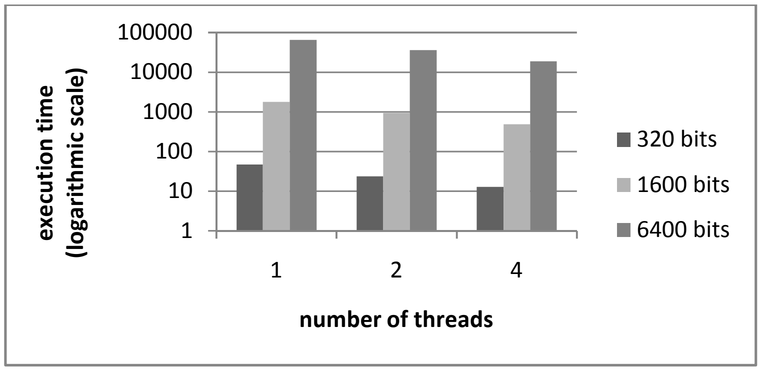


Figure 4. Execution time of sequential and parallel CRC computation

7. CONCLUSION

The CRC computation is traditionally implemented using sequential schemas. These codes divide an n bit dividend by an m bit divisor in n iterations regardless of the size of the divisor. In this paper a systematic method is proposed and explained to implement a software framework for parallel commutation of CRC based on modulo-2 mathematics. This paper also shows that if the divisor polynomial is selected from a special family of 32-bits strings called OZO, the division can be implemented using a parallel schema like multithreading and this code works faster than its corresponding sequential code. This work can be continued with designing parallel codes for other families of divisors.

REFERENCES

[1] Hamed Sheidaeian, Behrouz Zolfaghari, *Parallel Computation of CRC Using Special Generator Polynomials*, International Journal of Computer Networks & Communications (IJCNC), Vol. 4, No. 1, January 2012

[2] Behrouz Zolfaghari, Hamed Sheidaeian, Saadat Pour Mozafari, *Systematic Selection of CRC Generator Polynomials to Detect Double Bit Errors in Ethernet Networks*, In Proceedings of The Third International Conference on Computer Networks & Communications (CoNeCo 2011), Ankara, Turkey, June 26 ~ 28, 2011.

[3] Behrouz Zolfaghari, Saadat Pour Mozaffari, Haleh Karkhane, *A Systematic Approach to the Selection of CRC Generators to Detect Burst Errors in Ethernet Networks*, In proceedings of the IEEE International Conference on Intelligent Network and Computing (ICINC 2010), Kuala Lumpur, Malaysia, November 2010.

[4] Hamed Sheidaeian, Behrouz Zolfaghari, *An Efficient and Secure Approach to Multi-User Image Steganography Using CRC-Based CDMA*, In Proceedings of IEEE 3rd International Conference on Signal Acquisition and Processing, 2011, Singapure, Singapure.

[5] Xiaodong Deng, Mengtian Rong, Tao Liu, Yong Yuan, Dan Yu, *Segmented Cyclic Redundancy Check: A Data Protection Scheme for Fast Reading RFID Tag's Memory*. In Proceedings of IEEE Wireless Communications & Networking Conference (WCNC 2008), pp. 1576-1581, March 31 2008 - April 3 2008, Las Vegas, Nevada, USA.

[6] Ahmad, A. and Hayat, L., *Algorithmic Polynomial Selection Procedure for Cyclic Redundancy Check for the use of High Speed Embedded Networking Devices*, In Proceedings of International Conference on Computer and Communication Engineering 2008 (ICCCE'08), Kuala Lumpur, Malaysia - on 13-15 May, 2008.

[7] Yun Pana, Ning Ge, Zaiwang Dong, *CRC Look-up Table Optimization for Single-Bit Error Correction*, Tsinghua University Journal of Science & Technology, Tsinghua Science & Technology, Vol. 12, Issue 5, pp. 620-623, October 2007.

[8] Liu Zhanli, Liang Xiao, Zhao Chunming, Wang Jing, *CRC-Aided Turbo Equalization For MIMO Frequency Selective Fading Channels*, Journal of Electronics(China), Vol. 24, Issue 1, pp. 69-74, 2007.

[9] Walma Mathys, *Pipelined Cyclic Redundancy Check (CRC) Calculation*, In Proceedings of International Conference on Computer Communications and Networks, 2007 ICCCN 2007, In Proceedings of 16th International Conference on, pp 365-370, 13-16 August 2007.

[10] Raman Assaf, Tyszberowicz Shmuel, *The EasyCRC Tool*, In Proceedings of 2007 International Conference on Software Engineering Advances (ICSEA 2007), pp. 25-31, August 2007.

[11] Yalamarthy, Ragha Sudha; Wilson, G. Stephen, Near-ML *Decoding of CRC Codes*, In Proceedinggs of 41st Annual Conference on Information Sciences and Systems, pp. 92-94, 14-16 March 2007.

[12] Andrew. S. Tanenbaum, *Computer Networks*, 5th Edition, 2010, Prentice Hall

Authors

Hamed Sheidaeian is a Ph.D. student in computer engineering at University of Tehran, Iran. His r esearch areas include computer architecture, embedded system design, data communication and multimedia systems.

Behrouz Zolfaghari is a Ph.D. student in computer engineering at Amirkabir University of Technology (AUT), Tehran, Iran. His research areas include image processing, computer architecture and computer networks.