World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:2, 2013

Parallel-Distributed Software Implementation of Buchberger Algorithm

Praloy Kumar Biswas, Prof. Dipanwita Roy Chowdhury

Abstract—Grobner basis calculation forms a key part of computational commutative algebra and many other areas. One important ramification of the theory of Grobner basis provides a means to solve a system of non-linear equations. This is why it has become very important in the areas where the solution of non-linear equations is needed, for instance in algebraic cryptanalysis and coding theory. This paper explores on a parallel-distributed implementation for Grobner basis calculation over GF(2). For doing so Buchberger algorithm is used. OpenMP and MPI-C language constructs have been used to implement the scheme. Some relevant results have been furnished to compare the performances between the standalone and hybrid (parallel-distributed) implementation.

Keywords—Grobner basis, Buchberger Algorithm, Distributed-Parallel Computation, OpenMP, MPI.

I. Introduction

GROBNER basis has been a cornerstone for Computational Commutative Algebra. Along with many problems relating to System Theory [3], many problems from Algebra like Ideal Membership problem or from Geometry like Automatic Theorem Proving can be solved by constructing Grobner basis [6]. There have been some very important and interesting problems from Cryptanalysis [2] and Error Correcting Codes and like these many more can be tackled by applying Grobner basis in GF(2).

There have been several algorithms and implementations - both standalone and distributed - proposed to calculate Grobner basis. First algorithm to compute the Grobner basis is proposed by Bruno Buchberger [3]. However the complexity bounds of this algorithm is quite high and as a matter of fact the complexity of any general methods to compute Grobner basis is very stiff [12]. Later on, some improvements have been done upon this algorithm primarily to reduce the unnecessary S-Polynomial computations [6]. Recently two major contributions to this end have been made by Jean Charles Faugere when he proposed F4 and F5 [7], [8]. Along with these basic improvements, there are quite a few distributed and parallel implementation of Grobner basis calculation have been reported. As the best algorithm to compute Grobener basis, namely F4 and F5, rely heavily on linear algebraic techniques or more precisely on Gaussian Elimination, Faugere et. al. have published a parallel Gaussian Elimination technique for Grobner basis calculation over finite fields[5]. Another work reported to this end is by Soumen Chakraborty et. al. [4], where they proposed a set and priority queue data structures. Their approach also made balance between shared memory and distributed designs. The work by Heinz Kredel [10] reported a distributed and parallel implementation of Grobner basis by Java Computer Algebra Library (JAS). Their sole target is to provide an modern object oriented implementation of the tool. Based upon the observation that a Grobner basis of a set can be calculated from the Grobner basis of its subsets, Hemal V. Shah et. al. [17] proposed a technique to compute Grobner basis in a parallel machine in a tree like fashion. As far as the calculation of Grobner basis on GF(2) is concerned, the paper by Y. Sato [16] described a way to compute Grobner basis in distributed way over Boolean field. However his target was to solve the set constraint problem. He showed that set constraints can be represented by certain equations over certain boolean rings. In his work [11], Anton Leykin, elaborates on a parallel computation of Grobner basis. His idea is to use the traditional Buchberger algorithm and dedicate a process as the master process while making others as slaves.

Along with these works, some tools to calculate Grobner basis are already available, for instance Sage, Macauley, Singular. However these tools don't provide the option to run in parallel or distributed mode, in particular they don't give options to cash in the benefits of multicore or cluster machines. Moreover these tools are general purpose tools capable of calculating in any fields. Therefore, it appeared that a parallel-distributed implementation of Grobner basis calculation algorithm which can run in cluster environment will be good addition. As some recent works on algebraic cryptanalysis and algebraic coding theory make use of Grobner basis in Boolean field, the implementation done here is mainly for Boolean field. In so doing, memory efficient data structures to store and time efficient basic algorithms to manipulate Boolean polynomials have been conceived. An ingenious representation for Boolean polynomial has been reported in the documentation of Sage tool [1], however for this work our representation seems to be more apt. Load balancing and synchronization aspects of parallel distributed techniques have been taken care of by adopting an ingenious scheme as will be described down the line. The work is presented in this paper in the following way.

Section II presents Buchberger Algorithm along with some basic definitions and theoretical background of the work. The description of the basic data structures and polynomial manipulation routines are given in Section III. It also deals with the standalone implementation. Section IV describes the strategies adopted here for the hybrid (parallel-distributed) implementation of the tool. It also elaborates on how these strategies have been materialized in practical settings. Results

Praloy Kumar Biswas and Prof. Dipanwita Roy Chowdhury is with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India, e-mail: (see http://www.facweb.iitkgp.ernet.in/ drc/biography.html.

and observations are provided in Section V. Section VI draws the conclusion of the work.

II. Grobner Basis and Buchberger Algorithm

Some relevant portions of the theory is provided in this section. [18], [6] are consulted for this section. Let $P = F[x] = F[x_1, \ldots, x_n]$ be a polynomial ring in n variable over a field F. A power product of a variable is called a $term$. τ denotes the set of all terms in R. Then $\tau_d \subset \tau$ is the set of all terms of degree d. The $degree$ of a term $t = x_1^{d_1} x_2^{d_2} \ldots x_n^{d_n}$ is defined as $deg(t) = \sum_{i=1}^{n} d_i$. The product of a term and an element $c \in R$ is called a $monomial$.

Let $f = \sum c x_1^{a_1} \ldots x_n^{a_n} \in P$ be a non-zero polynomial. Let's define $T(f) = \{x_1^{a_1} \ldots x_n^{a_n} \in \tau : c \neq 0\}$ and $M(f) = \{c x_1^{a_1} \ldots x_n^{a_n} : c \neq 0\}$, and $T_d(f) = T(f) \cap \tau_d$. The $degree$ of f, denoted by deg(f), is the maximal d such that $T_d(f) \neq \emptyset$

Next for any set of polynomials $S \subset R$ let's define the following.

$$T(S) = \cup_{f \in S} T(f)$$

, $T_d(S) = T(S) \cap \tau_d$. Moreover $< S >$ is used to denote the ideal generated by all $f \in S$.

A *term order* $\preceq$ is a linear order on the set of terms τ if for all $t, t_1, t_2 \in \tau$ it holds that $1 = x_1^0 x_2^0 \ldots x_n^0 \leq t$ and if $t_1 \preceq t_2$, then $t_1 t \preceq t_2 t$. There can be several term orderings possible for the multinomials. Here lexicographical term order has been used, which is by definition $x_1^{d_1} \ldots x_n^{d_n} \preceq_{lex} x_1^{e_1} \ldots x_n^{e_n}$ iff there exists some i with $1 \leq i \leq n$ such that $d_i < e_i$ and $d_j = e_j$ for all $1 \leq j \leq i-1$. Let a term order $\preceq$ be fixed. For any two monomials at_1 and bt_2 with $t_1, t_2 \in \tau$ and non-zero coefficients $a, b \in F$, $at_1 \preceq bt_2$ iff $t_1 \preceq t_2$.

The maximal element of $T(f)$ w.r.t. $\preceq$ is called the *leading term* of f and is denoted by $LT(f)$. And in the same way, $LM(f) = max_{\preceq}(M(f))$, is called the *leading monomial* of f, amd its coefficient, denoted by $LC(f)$, is the *leading coefficient* of f. Clearly, $LM(f) = LC(f).LT(f)$. Also for any $S \subset R$ put $LT(S) = \{LT(f) : f \in S\}$.

A. Grobner Basis

Let $\preceq$ be a term order on τ. Let $G = \{g_1, \ldots, g_m\} \subset P$ be a set of polynomials. A polynomial $f \in P$ is called *reducible* modulo G, if there exists a term $t \in T(f)$ that is divisible by some leading term of G. The following algorithm describes a generalized division of f by G for multivariate case. It produces the reduced polynomial h produced after reduction of f by G. It is generally called the *normal form* of w.r.t. G. It is simply written by the notation as $f \xrightarrow{G} h$ or $\overline{f}^G$. Either of the notations will be used appropriately.

Algorithm 1: *Polynomial Reduction*

Input: A set $G = \{g_1, \ldots, g_n\} \subset P$ and $f \in P$
Output: $h \in P$ such that $f \xrightarrow{G} h$ where $h \in P$

```
begin
    h ← f
    while h is reducible modulo G do
        Select a monomial m ∈ M(h) such that m = a.t
        where a ∈ F and t = t1.LT(gi) ∈ τ for some
        1 ≤ ileqn and t1 ∈ τ
        h ← h − c.t1.g1, where c = a/LC(gi)
end
```

From the above algorithm it's clear that h is not reducible modulo G and there are $f_1, \ldots, f_m \in P$ such that

$$f = \sum_{i=1}^{m} f_i g_i + h$$

and $LT(f_i g_i) \preceq LT(f)$ for all $1 \leq i \leq m$. It is quite possible that the reduction of the polynomial f is not uniquely defined since the there can be several leading terms in G, dividing a particular term of f. However, any $f \in P$ has a unique normal form w.r.t. G if G is a spacial kind of basis, namely Grobner basis.

Definition Let $I \subset P$ be an ideal. A finite set of polynomials $G \subset I$ is called a Groner basis of I (w.r.t $\preceq$) if $< LT(G) >=< LT(I) >$.

The key concept behind the idea of Grobner basis is of S-Polynomial. The following theorem sums up the key to calculate Grobner basis.

Theorem 1: Let $G \subset P$ be a finite set of polynomials. Then G is a Grobner basis iff $\overline{spol(g_i, g_j)}^G = 0$ for any $g_i, g_j \in G$, where the polynomial $spol(g_i, g_j)$ called the S-Polynomial of g_i and g_j is given by

$spol(g_i, g_j) = \frac{lcm(LT(g_i), LT(g_j))}{LM(g_i)} . g_i - \frac{lcm(LT(g_i), LT(g_j))}{LM(g_j)} . g_j$

The work presented in this paper is based upon Buchberger Algorithm which works as follows.

Algorithm 2: *Buchberger Algorithm*

Input: A set $G = \{g_1, \ldots, g_n\} \subset P$
Output: Grobner basis for the ideal $< G >$

```
begin
    CP ← {(gi, gj) : ∀1 ≤ i < j ≤ n}
    while CP ≠ ∅ do
        Select (f, g) ∈ CP
        CP ← CP − {(f, g)}
        if spol(gi, gj)^G ≠ 0 then
            CP ← CP ∪ {(g, h) : ∀g ∈ G} and
            G ← G ∪ {h}, where h = spol(gi, gj)^G
end
```

In the above algorithm the elements of CP are called *critical pairs*.

Uptil now the theory and algorithms for calculating Grobner basis in general settings have been presented. Here the theories relevant for GF(2) will be given and only the statements of the important theorems are provided here. The source of this part is [13]. At first some definitions are furnished.

Definition A *Boolean polynomial* is same as a polynomial

defined above with conditions that $c, a_i \in GF(2)$ where $1 \leq i \leq n$. And the set of boolean polynomials in the variables $x_1, \ldots, x_n$ will be denoted as B.

Definition The set of polynomials $FP = \{a_1^2+a_1, \ldots, a_n^2+a_n\} \subset F_2[a_1, \ldots, a_n]$ is called the set of *Field polynomials*.

Definition The multiplication "." between two terms of Boolean polynomials is called a *Boolean multiplication* and is defined as

$(a_1^{\alpha_i} \ldots a_n^{\alpha_n}).(a_1^{\beta_1} \ldots a_n^{\beta_n}) = a_1^{max(\alpha_1,\beta_1)} \ldots a_n^{max(\alpha_n,\beta_n)}$, where $\alpha_i, \beta_i \in \{0.1\}$.

The typical way of calculating Grobner basis in GF(2) is to include FP in the initial basis and then feed that to the Grobner basis calculating algorithm. However, due to the following theorem the things can be handled differently in GF(2).

Theorem 2: Let $S \subset B$ be a generating system of some ideal, such that $FP \subset S \subset B \cup FP$. Then all the polynomials created in the classical Buchberger Algorithm applied to S are either Boolean polynomials or field polynomials, if a reduced normal form is used.

Theorem 3: FP is a Grobner basis.

This theorem tells that during the computation the field equations FP matter only in the intermediate ideal basis G and the set of critical pairs B. Therefore one doesn't need to represent them directly as data structure in the algorithm and can implicitly assume that they are there. And due to Theorem 2, the implementations of basic polynomials operations can be designed explicitly for Boolean polynomials. In the following section, the details of such polynomial representation and manipulation are given.

III. Basic Data Structures and Polynomial Manipulation Algorithms

A. Data Structures to store Boolean polynomials

The primary concern while developing a tool for Grobner basis is how to design the data structures to store the polynomials. According to the definition of Boolean polynomial one can store them by a number having at least l bit length of its binary representation and treating the bits as follows.

Let T be a term which is represented by a number $N = (b_1 \ldots b_l)_2$. Then T will be stored in the following way

$b_i = 1$, if a_i appears in T

$= 0$, otherwise

In case of the present work, the polynomial-terms having at most 128 variables are treated. So a structure having two long integers has been used. However our representation is fairly typical, it can be extended to greater lengths also. So the structure for a term is *Term { long high, low ;}*.

A polynomial has a structure which entails $number\_of\_terms$ and a pointer pointing to an array of terms. So the structure is *Polynomial { int number_of_terms; Term *t;}*.

B. The basic Boolean polynomial manipulating routines and the standalone implementation of Buchberger Algorithm

From the description of Buchberger Algorithm, it is clear that in order to implement it, the implementation of the routines for basic polynomial operations such as polynomial addition, polynomial multiplication and polynomial reduction by a base, are needed. After presenting the data structures to represent the polynomials in the previous subsection, here we elaborate the implementation of the basic polynomial operations and in so doing only the crux will be explained for the auxiliary routines.

The first thing to note is that for monomial ordering lexicographical ordering has been used and ordering of the variables considered to be $a_1 \succeq a_2 \succeq \ldots \succeq a_n$. Therefore, according to the proposed representation of the polynomial terms, the ordering of the natural numbers $\mathbb{N}$ itself will provide the lexicographic ordering of the terms. Based on this observation the routine to compare two terms *TermComparator* has been implemented. It just compares the constituent numbers in the terms involved and returns the result accordingly.

In order to restore the polynomials generated during the operations in lexicographic order, a sorting routine *SortAPolynomial* to sort the terms in a particular polynomial has been developed. As already the TermComparator routine provides the ordering of the terms, Bubble sort algorithm has been materialized using it. Based upon these two routines the polynomial addition routine has been developed as follows. It just merges the polynomials involved but discarding the common terms.

Algorithm 3: *AdditionOfTwoPolynomials*

```
   Input: P1, P2 of type Polynomial
   Output: R of type Polynomial such that R = P1 + P2
1  begin
2      size1 ← P1.number_of_terms
3      size2 ← P2.number_of_terms
4      R.terms ← AllocateMemoryForSize(size1 + size2)terms
5      while (i < size1)&&(j < size2) do
6          if TermComparator(P1.terms[i], P2.terms[j]) > 0 then
7              R.terms[k++] ← P1.terms[i++]
8          else if TermComparator(P1.terms[i], P2.terms[j]) < 0
           then
9              R.terms[k++] ← p2.terms[j++]
10         else
11             i++ ; j++ ;
12     while i < size1 do
13         R.terms[k++] ← p1.terms[i++]
14     while j < size2 do
15         R.terms[k++] ← p2.terms[j++]
16     R.number_of_terms ← k
17     return R
18 end
```

In case of *MultiplicationOfTwoPolynomials* an accumulator polynomial has been used and it has been added with the intermediate polynomial generated by multiplying the latest term of one operand polynomial with the other operand polynomial. The addition has been done through AdditionOfTwoPolynomial routine. For *MultiplyTwoTerms*, the bitwise OR between the constituent numbers representing the terms involved will do.

Algorithm 4: *MultiplyTwoPolynomials*

Input: $P1$, $P2$ of type Polynomial
Output: R of type Polynomial such that $R = P1 \times P2$

begin
  $size1 \leftarrow P1.number\_of\_terms$
  $size2 \leftarrow P2.number\_of\_terms$
  $R.number\_of\_terms \leftarrow 0$
  if $(size1 == 0)||(size2 == 0)$ **then**
    return R
  $R.terms \leftarrow AllocateMemoryForASingleTerm$
  $middle\_storage.number\_of\_terms \leftarrow size2$
  $middle\_storage.terms \leftarrow AllocateMemoryOf\_size2_Terms$
  for $i \leftarrow 0; i < size1; i++$ **do**
    for $j \leftarrow 0; j < size2; j++$ **do**
      $middle\_storage.terms[j] \leftarrow MultiplyTwoTerms(P1.terms[i], P2.terms[j])$
    $temp \leftarrow R$
    $R \leftarrow AddTwoPolynomials(R, middle\_storage)$
  return R ;
end

Algorithm 5: *ReduceAPolyByABase*

Input: $base$ an array of type Polynomial, $base\_size$ the size of $base$, P of type Polynomial
Output: $Remainder$ of type Polynomial such that $Remainder = P - \sum base[i]$ for some i in $\{1, \ldots base\_size\}$

begin
  $division_occured \leftarrow 0$
  $poly\_size \leftarrow poly.no\_of\_terms$
  $Remainder.no\_of\_terms \leftarrow 0$
  while $poly_size > 0$ **do**
    $division_occured \leftarrow 0$
    $i \leftarrow 0$
    while $i < base\_size$ && !$division\_occured$ **do**
      if $IsDivisible(poly.terms[0], base[i].terms[0])$ **then**
        $term \leftarrow DivideATermByATerm(poly.terms[0], base[i].terms[0])$
        $R2 \leftarrow MultiplicationOfATermAndAPoly(base[i], term)$
        $temp\_poly \leftarrow SubtractTwoPoly(poly, R2)$
        $poly \leftarrow temp\_poly$
        $poly\_size \leftarrow poly.no\_of\_terms$
        $division\_occured \leftarrow 1$
      else
        $i \leftarrow i + 1$
    if $division\_occured == 0$ **then**
      $Remainder \leftarrow AddATermToAPoly(Remainder, poly.terms[0])$
      $poly \leftarrow AddATermToAPoly(poly, poly.terms[0])$
      $poly\_size \leftarrow poly.no\_of\_terms$
  return Remainder
end

The term division, *DivideATermByATerm* is carried out by the observation that divisible term will lose the variables which are common with the dividing term. Therefore, the bitwise XOR operation between the constituent numbers in the terms involved is done. However before applying this routine it needs to be checked if the term to be divided is actually divisible by the dividing term (*IsDivisible*). It was done by checking if the bitwise OR between the numbers in the terms involved gives back the dividing term or not. Having presented these auxiliary routines, in Algorithm 5 the polynomial reduction routine is presented. Two other routines, namely *MultiplicationOfATermAndAPoly* and *AddATermToAPoly* are used here. As these routines are quite straightforward, their descriptions are skipped.

According to the definition of $spol$ given in Theorem 1, *MultiplyTwoTerms*, *DivideATermByATerm*, *MultiplicationOfATermAndAPoly*, *SubtractTwoPolynomials* these routines are sufficient to implement that *SPoly*. After describing the implementation of the needed routines, next the implementation of Buchberger Algorithm is presented.

Algorithm 6: *Buchberger*

Input: $base$, an array of type Polynomial, $base_size$ the size of $base$
Output: Generate the Grobner basis and store it to a file. Returns the modified $base_size$

begin
  InitializeIndicator() ;
  $remainder.no\_of\_terms \leftarrow 0$
  while $changed > 0$ **do**
    $changed \leftarrow 0$
    for $i \leftarrow 0; i < base\_size - 1; i++$ **do**
      for $j \leftarrow i + 1; j < base\_size; j++$ **do**
        if *!_indicator[i][j]* **then**
          $\_indicator[i][j] \leftarrow 1$
          $spoly \leftarrow SPoly(base[i], base[j])$
          if *spoly.no_of_terms* **then**
            $remainder \leftarrow DivideAPolyByAnyBase(base, base\_size, spoly)$
          if $remainder.no\_of\_terms$ **then**
            AddToTheBase(base, base_size, remainder) ;
            base_size++ ;
            StoreAPolyToAFile (fp, remainder) ;
            $changed \leftarrow 1$
  return base_size ;
end

In the following section, the details of parallel-distributed (hybrid) implementation of the tool is presented.

IV. Hybrid Implementation of Buchberger Algorithm

The fundamental property for which the Grobner basis calculation can be made distributive is that the base polynomials are independent of each other and they interact in a well specified way. Therefore here the idea is to store the generated basis in distributed fashion across several processes and device a scheme whereby the basis polynomials interact with each other in the specified way. Apart from this, an effort has been made here to parallelize the S-Polynomial calculations. The most time consuming part of Buchberger Algorithm is the calculation of S-Polynomials. However, the advantage is there is no interdependency between the S-Polynomials of two distinct pairs of polynomials. So here a technique has been adopted whereby the processes involved can calculate the S-Polynomials parallely and they communicate these S-Polynomials among themselves in a well defined way.

In the distribution scheme a process has been dedicated to do the coordination among the processes involved. It's named as "Driver". Rest of the processes involved is termed as "Auxiliary processes". Before elaborating the details we stress on the fact that here the polynomials as well as the processes are identified by their identification number. For polynomials it's given by their positions (index) in the basis and for the processes it's their process identification number (PID).

The number of polynomials each process will have, at least initially is $parts = \lceil \frac{base\_size}{no\_of\_processes\_involved-1} \rceil$. And each process will start reading its share of polynomial base from the polynomial number, named as $start\_index$, calculated as follows.

$start\_index = parts \times (PID - 1)$. Note here by PID, the $rank$ of a particular process is meant. Moreover the process with rank 0 is treated as the *Driver Process*.

To store the information about which polynomial resides in which process, a table named "Process Statistics Table" (PStat) has been maintained. The entries in this table has the following structure $\{$*int process_no, number_of_polynomials*$\}$. A routine *AppropriateProcessNumber* has been implemented as follows:

Algorithm 7: *AppropriateProcessNumber*

Input: $global_index$ of the Polynomial, $PStat$ the Stat table itself
Output: The PID of the process which posses the polynomial and the $local\_index$ of the polynomial within that process

```
1 begin
2     j ← 1
3     sum1 ← 0
4     sum ← PStat[j].number_of_polynomials
5     while global_index > sum do
6         sum1 ← sum
7         sum+ = PStat[j++].number_of_polynomials
8     local_index ← global_index − sum1
9 end
```

On inquiring for a particular polynomial with its index, its job is to give the appropriate process number where the polynomial is and the index of the polynomial within that process.

Before elaborating on "Load Balancing" and "Synchronization" aspects of the work in the following subsection dedicated to explain overall strategy.

A. The Basic Idea of Distribution and Parallelization of the Algorithm

Figure 1, depicts the scheme. The "Driver" is the process which actually finds the new polynomials to be included in the basis. For each possible polynomial pairs it requests to a particular process to give the S-Polynomial for that pair. Upon requested by the "Driver", that particular "Auxiliary process" comply to the request. As all the "Auxiliary processes" calculates the S-Polynomials parallely, it takes shorter time than the case where the one particular process needs to calculate all the s-Polynomial required. Here each process runs two threads, one to calculate -by and large- the S-Polynomials and the other to communicate with other processes.

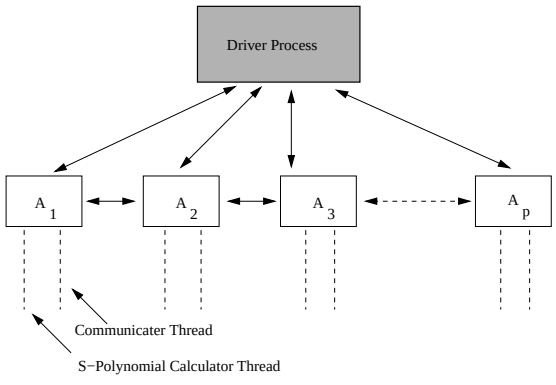

Fig. 1. Communication protocol between the processes

B. Load Balancing and Increasing the S-Polynomial Hit

After the polynomials in the basis get distributed, the idea is that a particular process will start calculating the S-Polynomials for the pairs of polynomials it can find in its memory. As the polynomials are identified by their unique indexes, a severe problem arises when the "Driver" starts requesting the S-Polynomials. A typical nested loop is there in the "Driver" to get all possible distinct order independent pairs of polynomial indexes. Obviously, then the pair $\{1, j\}$, where $2 \leq j \leq N$, N : Number of polynomials in the basis will occur maximum number of times. Then $\{2, j\}$ will occur much of the time. And so on. Therefore the processes which stores the polynomials with indexes of lesser values, need to calculate more and more S-Polynomials than others. That's the work distribution across the process will be skewed. Another problem is, suppose the "Driver" needs the S-Polynomial for a pair $\{i, j\}$ where both the polynomials belongs to different processes, then neither of the processes can calculate the S-Polynomial and comply to "Driver"'s request. In such cases, the "Driver" itself needs to calculate the S-Polynomial, which clearly will kill some time. Let's dub such events as "S-Polynomial Miss" and where "Driver" is complied with the S-Polynomial request, is described as "S-Polynomial Hit".

Therefore, the scheme designed should be such that it balances the load across the processes involved and as well as gives most S-Polynomial Hit. To address this issue, an equivalence class of indexes is formed for each polynomial index. The idea is the process which contains the polynomial having the head index of some equivalence class will calculate the S-Polynomials for the pair $\{i, j\}$ such that j belongs to i's equivalence class. The algorithm presented in the box has been used to generate the equivalence classes of the indexes.

Initialize $i = 1$, and then do the following.
while $(i \leq n - 1)$ {
1. Initialize the equivalence class for i with a viable pair, say $\{i, j\} = \{i, i + 1\}$.
2. Then do as follows steps and keep on adding the pairs generated in the equivalence class for i. $(i, j) \rightarrow (j + 1, i)$ and then generate $(j + 1, i) \rightarrow (i, j + 2)$, then again repeat the process as both i or j remains within the required bound i.e. $\leq n$, where n is the size of the basis.
Once the series got stuck for i, start with $i = i + 1$.
}

The following example illustrates the idea behind the scheme.

Example: Let's $n = 5$. Then the equivalence classes generated will be $\{\underline{1}, 2, 4\}$, $\{\underline{2}, 3, 5\}$, $\{\underline{3}, 4, 1\}$, $\{\underline{4}, 2, 5\}$. Here the underlined numbers in each class are the class heads. In the Figure 2, consider the numbers as the indexes of the polynomials, then the dots represent a pair of distinct polynomials. Obviously then the dots shown in the upper triangular matrix in left hand side of the figure, represents the pairs for which the S-Polynomials are needed to calculate. Clearly it depicts the skewness where polynomials with lower indexes have to be frequently paired up than the polynomials with higher indexes. However, after the applying equivalence algorithm, the dots spread out uniformly across the cells in the matrix and it also doesn't hamper the S-Polynomial calculation as S-Polynomial(i,j) is equivalent to S-Polynomial(j,i) as far as Buchberger algorithm is concerned.

Therefore to store these information regarding equivalence class a 2D array of integers is allocated and is named as "EquivalenceTable". The crux of this table is that for each

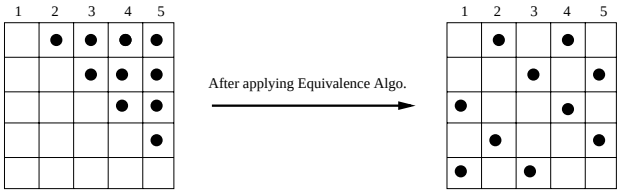

Fig. 2. Redistribution of the polynomial pairs

$\{i, j\}$ pair, either j will occur in i's equivalence class or i will occur in j's equivalence class. Now, when the "Driver" needs to request the S-Polynomial of a pair $\{i, j\}$ it consults the "Equivalence Table" and sees which among i and j appears as the head of the other. It then request the process which contains the polynomial with the head index among i, j for the required S-Polynomial. As according to the scheme a particular process calculates all the S-Polynomials of the pair $\{i, j\}$ where i is the index of a polynomials in it and j is the index appeared in i's equivalence class, "S-Polynomial Hit" will surely occur.

There is another issue which deals with how to augment these informations when the basis itself gets augmented. In the following subsection, an integrated view of these strategies along with the augmentation scheme are presented.

C. Execution Flow of the Overall Implementation

"Driver" stores the new polynomials generated in the processes circularly i.e. giving turn to each process one by one. After doing so it updates the "PStat" and "Equivalence Table" accordingly. It then broadcasts the information of the newly generated polynomial to other processes which in turn update their tables. In the following box, the overall process is summarized from the perspective of the "Driver".

1. Driver process creates the Equivalence Table and PSTAT table based on size of the initial basis. It then broadcasts PSTAT table to all other processes but selectively sends the Equivalence table.
2. The auxiliary processes start reading out their shares of base polynomials from a file. Each auxiliary process when done with such reading, intimates that to the Driver.
3. for all possible order independent pairs $\{i, j\}$ with $0 < i < j \leq Size\_Of\_The\_Basis$
   a. The Driver consults the Equivalence table and figure out which process to request to for S-Polynomials(p_i, p_j).
   b. The Driver then requests and get the S-Polynomial.
   c. Divide the S-Polynomial by rest of the polynomials and note the Remainder generated.
   d. If Remainder is non-zero, Driver adds it to the process which holds the current highest number and broadcast the information that a new polynomial has been introduced with polynomial number of the newly created polynomial. Driver then augments the Equivalence table. The generated Remainder is also added to the file, Grobner_basis.txt.
   e. After receiving the notification for the introduction of a new polynomial, the auxiliary processes updates their own "Equivalence" and "PStat" tables.

From the perspective of a particular "Auxiliary Process", the overall scheme is being represented as follows. However in case of such processes the challenge is how to store the calculated S-Polynomials. It's obvious that on average the number of S-Polynomials needed to be calculated by a process is roughly $S = \lceil \binom{number\_of\_polynomials}{2} / total\_number\_of\_auxiliary\_processes \rceil$. Therefore to store the calculated S-Polynomials a 2D array of $S \times S$ dimension is maintained. Then all the indexes occurred in the equivalence classes of the indexes of the polynomials local to it, are orderly assigned a number, starting from 0. So, whenever a request for S-Polynomial $\{i, j\}$ comes, it checks the order number for i and j and quarries the S-Polynomial 2D array at the index given by these order numbers.

1. Read out its share of initial basis from the initial basis file.
2. After reading out, intimates that to "Driver".
3. Receive the "PStat" and "Equivalence Table" from "Driver" and stores these to its local memory.
4. Consults the local basis of the polynomials and the "Equivalence Table" and starts computing the S-Polynomials.
5. If the other polynomial to compute the S-Polynomial is not in its place, it consults with "PStat" table and requests the appropriate process to send the required polynomial to it. After calculating S-Polynomial, say for the pair $\{i, j\}$ it stores it at the position shown above in its local S-Polynomial repository.
6. If it gets and request from the "Driver" for S-Polynomial for a particular pair, it complies to that request.
7. In case it has the augmentation notification, it updates the tables in the same way as the "Driver" does.

For **augmenting** the "Equivalence Table", it is observed that when the latest entry in the table is "Even" number then, only it will add to the equivalence class of the "odd" entries, and it has the equivalent class with "even" entries. On the other hand, if the latest entry in the table is "odd", it will add itself to the equivalence entry of the "even" numbers and have the "odd" entries as its own equivalent member.

In what follows some important snippets of implementation have been provided.

D. Important Implementation Snippets

In all the above implementation, for communication between the processes MPI_Send, MPI_Recv and MPI_Bcast have been used. For the technical details of the APIs we have consulted [14], [9]. Typical communicating portion of the implementation are given below:

```
// For Polynomial Sending
MPI_Send (buffer, 1, MPI_INT, process_no, 123, MPI_COMM_WORLD);
for (k = 0; k <  spoly.number_of_terms; k++)
MPI_Send (&(spoly.terms[k].a), 1, MPI_LONG, process_no, 123,
        MPI_COMM_WORLD) ;

// For Polynomial Receiving
temppoly2.number_of_terms = buffer[0];
temppoly2.terms = (Term*)malloc(temppoly2.number_of_terms*
                                        sizeof(Term));
for (k = 0; k < temppoly2.number_of_terms; k++)
MPI_Recv(&(temppoly2.terms[k].a), 1, MPI_LONG, process_no,
                              123, MPI_COMM_WORLD, &status);


// For MPI Broadcasting
for (k = 1; k <= size; k++)
MPI_Bcast(&(PStat[k+1].process_no), 1, MPI_INT, 0,
                                      MPI_COMM_WORLD) ;
```

Note here that as the polynomials are being sent and received by the processes one term at a time, the communication cost will increase with the size of the polynomials.

Apart from these aspects, there is another issue to look at i.e. how the different processes access a common file

"Grobner_Basis.txt", which stores the generated Grobner basis in a collaborative way, particularly for writing. To do that MPI C provides some file handling functions. We have used the following function.

int MPI_File_write(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

Where fh : file handle (handle); buf : initial address of buffer (choice); count: number of elements in buffer (nonnegative integer) ; datatype : datatype of each buffer element (handle)

And it outputs a status object.

A typical use of this is as follows:

```
// For writing to the file
MPI_File_write(fh, buf, 1, MPI_CHAR, MPI_STATUS_IGNORE) ;
```

V. Results

For testing the performance of the tool, we used the equations generated for the Bivium stream ciphers [15] for different number of rounds. Below a comparison between the time taken to generate the Grobner basis by the standalone implementation and that by hybrid implementation is given. For the standalone part the following platform has been used:

Processor: Intel(R)xeon(R)CPU E5606@2.13GHz×8, Memory: 7.8 GiB, Operating System: 12.04 (Ubuntu Server) 64-bit

And for the cluster part we have used "Daksh" cluster at SAG, DRDO, New Delhi, India Office. It's a normal state of the art IBM super computer.

Table 1 shows the time comparison to calculate the Grobner basis between 8 core machines with standalone implementation and using 20 cores using a cluster. The left hand part shows the timing for standalone part and that of right is showing the timing for cluster. The timings are given for 30 variable equations i.e. the initial base size is 30, however the size of the polynomial i.e. the number of terms in a polynomial is increased as the round increases. A rough estimation of the polynomial sizes involved in the initial basis while doing the experimentation is shown in Table 1. In Table 1, "M" stands for minutes, "S" for seconds and "Mi" for microseconds.

From the Table 1, it's clear that initially the hybrid code doesn't prove to be very effective but as the round increases it started to give some advantages. The timing advantage is most visible when the polynomial size is moderate. Then we guess communication overhead started to show up and the this timing gap diminishes. However if some more results can be had, perhaps the tendency can be more clear.

TABLE I
Timing comparison for generating Grobner basis of the equations for 30-variable BIVIUM

No. of Terms	Standalone Machine			Hybrid Machine		
	M	S	Mi	M	S	Mi
17	6	2	2091	4	24	2366
23	6	48	65213	4	59	7210
50	7	12	754	7	782	5531
145	7	58	10092	7	30	9478
250	9	34	8011	8	28	1121
559	10	21	4526	9	30	10972
790	12	18	70034	10	49	7452
1067	13	27	845002	10	40	1780
1908	13	58	907223	10	37	45031
2400	14	57	3598	10	29	107948
2600	15	1	80017	11	41	87101
3090	15	32	715327	11	53	83167
3500	17	24	98151	13	34	302917
4000	19	10	860	14	12	87328
4300	23	28	9718	18	39	578
5800	27	15	4090	20	46	19888
6210	30	31	36342	23	12	383426
7900	32	42	42752	27	31	35325
8930	37	56	83421	31	34	47841
9400	45	47	726885	37	15	456348
10050	55	19	52327	45	42	8352
11500	62	56	34266	57	30	63852
12000	80	23	25721	69	28	35437
13500	100	31	735724	85	27	34527
14050	120	53	17311	102	9	311
15000	145	48	24619	136	27	34521
17000	180	29	981543	170	11	16429

VI. Conclusion

The work presented here is about the development of a Grobner basis calculating software in GF(2) by applying distributed and parallel techniques. This tool can be applied in many areas of Coding and Cryptography where large amount of equations needed to be processed. The approach of this present work has been to develop a tool from scratch so that a firm grip can be availed on the kind of data structures and operations applied during the Grobner basis calculation. The design and implementation of the tool has taken care of the basic design criterion of the distributed algorithms such as load balancing and synchronization. The tool can be used in cluster environment as well. Experimentation done so far at authors end shows that it's working as expected. As for future improvement, the tool can be improved by adopting more advanced Grobner basis calculating algorithms.

References

[1] Url: http://www.sagemath.org/doc/reference/sage/rings/polynomial/pbori.html.
[2] Gregory V. Bard. *Algebraic Cryptanalysis*. Springer, New York, USA, 2009.
[3] B. Buchberger. *Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory.* Reidel Publishing Company, Dodrecht - Boston - Lancaster, 1985.
[4] Soumen Chakrabarti and Katherine A. Yelick. Distributed data structures and algorithms for grobner basis computation. *Lisp and Symbolic Computation*, 7(2-3):147–172, 1994.
[5] Jean charles Faugère and Sylvain Lachartre. Parallel gaussian elimination for grbner bases computations in finite fields. In *ACM proceedings of The International Workshop on Parallel and Symbolic Computation (PASCO)*, pages 1–10. ACM, 2010.
[6] Donal O'Shea. David Cox, John Little. *Ideals, Varieties, and Algorithms.* Springer Verlag, 1992.
[7] Jean Charles Faugère. A new efficient algorithm for computing grobner bases (f4). In *Journal of Pure and Applied Algebra*, pages 75–83. ACM Press, 1999.

[8] Jean Charles Faugère. A new efficient algorithm for computing grobner bases without reduction to zero (f5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, ISSAC '02, pages 75–83, New York, NY, USA, 2002. ACM.

[9] Gerhard Wellein. George Hager. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press (Taylor & Francis Group)., New York, USA, 2011.

[10] Heinz Kredel. Distributed parallel groebner bases computation. In *CISIS*, pages 518–524, 2009.

[11] Anton Leykin. On parallel computation of gröbner bases. In *ICPP Workshops*, pages 160–164, 2004.

[12] Ernst W. Mayr. Some complexity results for polynomial ideals. *JOURNAL OF COMPLEXITY 13, ARTICLE NO. CM970447*, pages 303–325, 1997.

[13] Gert-Martin Grenel Markus Wedler Oliver Wienand. Michael Brickenstein, Alexander Dreyer. New developments in the theory of grobner bases and application to formal verification. *Joournal of Pure and Applied Algebra. Vol. 213*, pages 1612–1635, 2008.

[14] Michael J. Quinn. *PARALLEL PROGRAMMING in C with MPI and OpenMP.* TATA McGRAW-HILL Education Pvt. Ltd., New Delhi, India, 2010.

[15] Havvard Raddum. Cryptanalytic results on trivium. *The eStream Project*, 2006.

[16] Yosuke Sato. Parallel computation of boolean grobner bases. *SIGSAM Bull.*, 34(1):27–28, March 2000.

[17] Hemal V. Shah and Jose A. B. Fortes. Tree structured grobner basis computation on parallel machines. In *ECE Technical Reports. Paper 199*, 1994.

[18] Philippe Loustaunau. Williums W. Adams. *An Introduction to Grobner Basis.* American Mathematical Society (AMS), 1994.

Praloy Kumar Biswas Praloy Kumar Biswas was born at Siliguri, a town in the state of West Bengal in India in 1983. He did his schooling upto higher secondary education at Siliguri. Then he got admitted in Jadavpur University, Kolkata in 2001 to study Computer Science & Engineering. After receiving his bachelor's degree from there, in 2005 he began his professional career as a project engineer in Wipro Technologies, Hyderabad. He has also worked in Atrenta India Pvt. Ltd., Noida and Verific Design Automation, Kolkata as software engineer. In 2009, he enrolled in the MS (Master of Science) program of Computer Science & Engineering Department in Indian Institute of Technology, Kharagpur to study Cryptography.

Dipanwita Roy Chowdhury Dipanwita Roy Chowdhury received her B.Tech and M.Tech degrees in Computer Science from University of Kolkata in 1987 and 1989 respectively, and the PhD degree from the Department of Computer Sc and Engg, Indian Institute of Technology, Kharagpur in 1994.

She is a Professor in the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India. Her current research interests are in the field of Cryptography, Error Correcting Code, Cellular Automata, and VLSI Design and Testing.

She has published more than 130 technical papers in International Journals and Conferences. Dr. Roy Chowdhury is the recipient of INSA Young Scientist Award and Associate of Indian Academy of Science and is the fellow of Indian National Academy of Engineers (INAE).