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

G ROBNER 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, Macaulay, 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.
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 \( P = F[x_1, \ldots, x_n] \) be a polynomial ring in \( n \) variable over a field \( F \). A power product of a variable is called a term, \( \tau \) denotes the set of all terms in \( R \). Then \( \tau_d \subseteq \tau \) is the set of all terms of degree \( d \). The degree of a term \( t = x_1^{d_1} \cdots x_n^{d_n} \) 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 \( f = \sum c_i x_1^{d_1} \cdots x_n^{d_n} \in P \) be a non-zero polynomial. Let’s define \( T(f) = \{ x_1^{d_1} \cdots x_n^{d_n} \in \tau : c_i \neq 0 \} \) and \( M(f) = \{ c_1 x_1^{d_1} \cdots x_n^{d_n} : c \neq 0 \} \), 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 \( S \subseteq R \) let’s define the following:
\[
T(S) = \bigcup_{f \in S} T(f), \quad T_d(S) = T(S) \cap \tau_d. \quad \text{Moreover} \quad <S> \text{ is used to denote the ideal generated by all } f \in S.
\]

A term order \( \preceq \) is a linear order on the set of terms \( \tau \) if for all \( t, t_1, t_2 \in \tau \) it holds that \( 1 = x_1^{d_1} \cdots x_n^{d_n} \preceq t \) if and only if \( t_1 \preceq t_2 \). There can be several term orderings possible for the multinomials. Here lexicographical term order has been used, which is by definition \( x_1^{d_1} \cdots x_n^{d_n} \preceq_{\text{lex}} x_1^{e_1} \cdots x_n^{e_n} \) 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 \( a t_1 \) and \( b t_2 \) with \( t_1, t_2 \in \tau \) and non-zero coefficients \( a, b \in F, \) \( a t_1 \preceq b t_2 \) 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 \( f \), and its coefficient, denoted by \( LC(f) \), is the leading coefficient of \( f \). Clearly, \( LM(f) = LC(f) \cdot LT(f) \). Also for any \( S \subseteq R \) put \( LT(S) = \{ LT(f) : f \in S \} \).

A. Grobner Basis

Let \( \preceq \) be a term order on \( \tau \). Let \( G = \{ g_1, \ldots, g_n \} \subseteq P \) 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 \( f \xrightarrow{G} h \). Either of the notations will be used appropriately.

**Algorithm 1: Polynomial Reduction**

**Input:** A set \( G = \{ g_1, \ldots, g_n \} \subseteq P \) and \( f \in P \)

**Output:** \( h \in P \) such that \( f \xrightarrow{G} h \) where \( h \in P \)

1. \( h \leftarrow f \)
2. While \( h \) is reducible modulo \( G \) do
3. Select a monomial \( m \in M(h) \) such that \( m = a t \) where \( a \in F \) and \( t = t_1, LT(g_i) \in \tau \) for some \( 1 \leq i \leq n \) and \( t_1 \in \tau \)
4. \( h \leftarrow h - c t_2 g_i, \text{ where } c = a/LC(g_i) \)
5. 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 \( I \subseteq P \) be an ideal. A finite set of polynomials \( G \subseteq I \) is called a Grobner basis of \( I \) (w.r.t \( \preceq \)) if \( LT(G) \supseteq LT(I) \).

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 \( G \subseteq P \) be a finite set of polynomials. Then \( G \) is a Grobner basis iff \( \text{spol}(g_j, g_i) = 0 \) for any \( g_i, g_j \in G \), where the polynomial \( \text{spol}(g_j, g_i) \) called the S-Polynomial of \( g_i \) and \( g_j \) is given by
\[
\text{spol}(g_j, g_i) = \frac{lcm(LT(g_j), LT(g_i))}{LM(g_j)} \cdot g_i - \frac{lcm(LT(g_i), LT(g_j))}{LM(g_i)} g_j
\]

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

**Algorithm 2: Buchberger Algorithm**

**Input:** A set \( G = \{ g_1, \ldots, g_n \} \subseteq P \)

**Output:** Grobner basis for the ideal < \( G >$

1. \( CP \leftarrow \{(g_j, g_i) : \forall 1 \leq i < j \leq n \} \)
2. While \( CP \neq \emptyset \) do
3. Select \((f, g) \in CP \)
4. \( CP \leftarrow CP - \{(f, g)\} \)
5. If \( \text{spol}(g, f) = 0 \) then
6. \( CP \leftarrow CP \cup \{(g, h) : \forall g \in G \} \) and \( G \leftarrow G \cup \{h\}, \text{ where } h = \text{spol}(g, j) \)
7. 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 theorems 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 $FP = \{a_1 + a_2 + \ldots a_n| a_1, a_2, \ldots, a_n \in P_2\}$ 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 $$(a_1^{\alpha_1} \ldots a_n^{\alpha_n}) . (a_1^{\beta_1} \ldots a_n^{\beta_n}) = a_1^{\max(\alpha_1, \beta_1)} \ldots a_n^{\max(\alpha_n, \beta_n)},$$

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 $FP \cup S \subset B$. 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 polynomial 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 $N = (b_1 \ldots b_l)$. Then $T$ will be stored in the following way

- $b_i = 1$, if $a_i$ appears in $T$
- $b_i = 0$, 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 $\text{Term}$ {

| long high, low ; |

A polynomial has a structure which entails $\text{number_of_terms}$ and a pointer pointing to an array of terms. So the structure is $\text{Polynomial}$ {

| 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 $a_1 \geq a_2 \geq \ldots \geq a_n$. 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 $\text{TermComparator}$ 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 $\text{SortAPolynomial}$ 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**: $P_1, P_2$ of type Polynomial  
**Output**: $R$ of type Polynomial such that $R = P_1 + P_2$

1. **begin**
2. $\text{size1} \leftarrow P_1.\text{number_of_terms}$
3. $\text{size2} \leftarrow P_2.\text{number_of_terms}$
4. $R.\text{terms} \leftarrow \text{AllocateMemoryForSize(size1 + size2) terms}$
5. while ($\text{i} < \text{size1}$) and ($\text{j} < \text{size2}$) do
6. if ($\text{TermComparator}(P_1.\text{terms}[i], P_2.\text{terms}[j]) > 0$ then
7. $\text{R.terms}[k] \leftarrow P_1.\text{terms}[i] + P_2.\text{terms}[j]$ |
8. else if ($\text{TermComparator}(P_1.\text{terms}[i], P_2.\text{terms}[j]) < 0$ then
9. $\text{R.terms}[k] \leftarrow P_1.\text{terms}[i] + P_2.\text{terms}[j]$ |
10. else
11. $\text{i}++ : \text{j}++ ;$
12. while ($\text{i} < \text{size1}$) do
13. $\text{R.terms}[k] \leftarrow P_1.\text{terms}[i] + P_1.\text{terms}[i+1]$ |
14. while ($\text{j} < \text{size2}$) do
15. $\text{R.terms}[k] \leftarrow P_2.\text{terms}[j] + P_2.\text{terms}[j+1]$ |
16. $\text{R.\text{number_of_terms}} \leftarrow \text{k}$
17. return $R$
18. **end**

In case of $\text{MultiplicationOfTwoPolynomials}$ 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 $\text{AdditionOfTwoPolynomials}$ routine. For $\text{MultiplyTwoTerms}$, the bitwise OR between the constituent numbers representing the terms involved will do.
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. 

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 devise 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 \( \text{parts} = \left\lfloor \frac{\text{base size}}{\text{base size per process}} \right\rfloor \). And each process will start reading its share of polynomial basis from the polynomial number, named as \( \text{start index} \), calculated as follows.

\[
\text{start index} = \text{parts} \times (\text{PID} - 1).
\]

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, number of polynomials\}. A routine AppropriateProcessNumber has been implemented as follows:

**Algorithm 7: AppropriateProcessNumber**

```
begin
  j ← 1
  sum1 ← 0
  sum ← PStat[j].number_of_polynomials
  while global_index > sum do
    sum1 ← sum
  sum ← PStat[j + 1].number_of_polynomials
  local_index ← global_index - sum1
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 request by the “Driver”, that particular “Auxiliary process” complies to the request. As all the “Auxiliary processes” calculates the S-Polynomials in parallel, 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.

```
Hexagon
```

```
Driver Process
```

```
A_1
```

```
A_2
```

```
A_3
```

```
A_4
```

```
S-Polynomial Calculator Thread
```

```
Communicator Thread
```

```
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 \{i,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 \(i \leq n - 1\) \{
    Initialize the equivalence class for i with a viable pair, say \(i, j = [i, i + 1]\).
    Then do as follows steps and keep on adding the pairs generated in the equivalence class for \(i, j\rightarrow (j + 1, i)\) 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. \(i \leq n\), 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 \(\{1, 2, 4\}, \{2, 3, 5\}, \{3, 4, 1\}\). 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
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 

\[ S = \left( \frac{\text{number of polynomials}}{\text{total number of auxiliary processes}} \right) \]

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 queries the S-Polynomial 2D array at the index given by these order numbers.

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:

```c
// For Polynomial Sending
MPI_Send(buffer, 1, MPI_INT, process_no, 123, MPI_COMM_WORLD);
for (k = 0; k < s.poly.number_of_terms; k++)
    MPI_Send(&(s.poly.terms[k].a), 1, MPI_LONG, process_no, 123,
             MPI_COMM_WORLD);
// For Polynomial Receiving
temp_poly2 = (Term*)malloc(temp_poly2.number_of_terms * sizeof(Term));
for (k = 0; k < temp_poly2.number_of_terms; k++)
    MPI_Recv(temp_poly2.terms[k].a, 1, MPI_LONG, process_no, 123,
              MPI_COMM_WORLD, &status);
// For MPI Broadcasting
MPI_Bcast(&(PStat[k].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.

```c
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.

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


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 Science and Engineering, 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 (INAIE).