Applications of Projective Geometry in Computing and Communications

Dual Degree Project Stage-III Report

Submitted in the Partial Fulfillment of the

Requirements for the award of Dual Degree

in

Electrical Engineering

by

Abhishek Patil

Roll No: 04D07022

Under the Guidance of

Prof. Sachin B. Patkar, Prof. D. K. Sharma and Dr. B. S. Adiga

Department of Electrical Engineering

Indian Institute of Technology

Mumbai - 400076

Approval Certificate

Department of Electrical Engineering ,

Indian Institute of Technology,

Bombay Powai, Mumbai-76

The DDP Third stage report titled "Applications of Projective Geometry in Computing and Communications" submitted by Abhishek Patil (Roll no. 04D07022) was done under my guidance and may be accepted for evaluation.

Date: 23rd June,2009

Prof. Sachin Patkar

Acknowlegdement

I would like to thank Prof. S. Patkar for his guidance and support during the course of this project. I would also like to thank Prof. H. Narayanan, Prof. D. K. Sharma and Dr. B. S. Adiga for their useful insights.

I would like to acknowledge that the work on expander codes has been done in partnership with Swadesh Choudhary and Yash and I am thankful to both of them.

I also want to thank Mr. Nachiket Gajare and Mr.Hrishikesh Sharma for reviewing my work and providing constructive feedback, and Mr. Balamurali, Tata Consultancy Services, Bangalore, for supporting this work under the TCS Project Code 1009298.

Date:23rd June,2009 Abhishek Patil

04D07022

Contents

Table of Contents **iii**

1 Introduction . . . 1
2 Karmarkar's Architecture . . . 3
  2.1 Projective Spaces Over Finite Fields . . . 3
  2.2 LU Decomposition . . . 4
  2.3 4-D Projective Geometry over GF(2) . . . 6
    2.3.1 Frobenius Automorphism . . . 8
    2.3.2 Shift Automorphism . . . 8
  2.4 Perfect Matching Sequences . . . 9
3 Problem-mapping strategies . . . 10
  3.1 Algorithm Mapping Scheme I . . . 10
    3.1.1 Data Distribution . . . 10
    3.1.2 Operations Involved . . . 11
    3.1.3 Distribution Of Computations . . . 12
    3.1.4 Details of 0-th iteration . . . 14
    3.1.5 Design Analysis . . . 15
  3.2 Algorithm Mapping Scheme II . . . 15
    3.2.1 Distribution of Computational Load . . . 15
    3.2.2 Details of 0-th iteration . . . 16
    3.2.3 Design Analysis . . . 18
  3.3 Simulation Results . . . 18
  3.4 Conclusions . . . 21
4 Perfect Difference Networks . . . 22
  4.1 Perfect Difference Sets . . . 22
  4.2 Networks based on PDS . . . 22
  4.3 Routing Algorithms . . . 23
  4.4 Edge Expansion of Perfect Difference Networks . . . 25
  4.5 Conclusions . . . 27
5 Expander Graphs and Expander Codes . . . 28
  5.1 Characteristics of a Linear Code . . . 28
  5.2 Expander Graph . . . 28
  5.3 Construction of Expander Codes . . . 28
  5.4 Sipser and Spielman's Decoding Algorithm . . . 29
  5.5 Zemor's Construction and Decoding Algorithm . . . 30

5.6 MATLAB Implementation: Code Construction 31
5.7 MATLAB Implementation: Results 32
5.8 Conclusions 33
Bibliography 34
A Appendix 35
A.1 Generation of GF(2^5) from GF(2) 35
A.2 Projective Geometry Structure 36
A.3 Graph for expander codes 42

List of Figures

1 Scheme I: Diagonal block communication 11
2 Scheme I: Execution of 0-th iteration on $(0, 1, 2, 5, 11, 18, 19)$ (steps 1-9) 12
3 Scheme I: Execution continued (steps 10-16) 13
4 Scheme II: Diagonal block communication 16
5 Scheme II: Execution of 0-th iteration on bus $(0, 1, 2, 5, 11, 18, 19)$ 17
6 Perfect Difference Network with 7 nodes [3] 23
7 (a)Complete Graph, G_1 (b)Perfect Difference Network, G_2 25
8 (a)Cut shown in Complete Graph, G_1 (b)Lines in PG with points on either side of the cut(marked with *) (c)Cut in Perfect Difference Network, G_2 26

List of Tables

1 Transfer of diagonal data, 0th iteration 14
2 Block Sizes and Relative Time Periods 19
3 Total communication and computation cycle count for the three schemes 19
4 Total time required for the three schemes 20
5 Average Active Processor Cycles 20
6 Normalized Perfect Difference Sets [3] 23
7 Change in parameters of $\mathbb{C}$ with variation in minimum distance of subcode 32
8 Elements of GF(2^5) 35
9 Points of $\mathbb{P}$(4,GF(2)) 36
10 Planes of $\mathbb{P}$(4,GF(2)) and the mappings $S_1, \ldots, S_7$ 41
11 Points of $\mathbb{P}$(5,GF(2)) 43

Abstract

Projective geometry(PG) based graphs have found many applications in the areas of computing and error-control coding. Firstly, we look at the use of PG in the design of processor-memory interconnection architectures. We have used projective subspaces to create novel schemes to solve the problem of LU decomposition. We have simulated these architectures and compared them with conventional schemes. In this report, we present our findings. Then, we study a new family of networks based on PG. We understand how the regularity and symmetry in their structure is used to develop communication primitives for them. Finally, we look at expander codes, which are based on expander graphs, and present our construction of such codes based on graphs obtained using PG. Through simulations, we have found that these codes perform better than the theoretical bounds suggested in literature. We present the results of these simulations in this report.

1 Introduction

Projective geometry and graphs based on projective spaces are being extensively studied for many different applications. We look at the use of projective geometry in computations and development of good error-correcting codes.

For speeding up computations, it has become necessary to look towards parallel architectures. The two major problems encountered in parallelising computations are

- **Load balancing:** Distributing the computations in such a way that each processor gets equal computational load.
- **Memory-access conflicts:** In parallel architectures, many situations arise when two or more processors need to access the same data and this leads to memory-access conflicts. These conflicts have to be resolved on a case-to-case basis by the programmer.

In [2], Karmarkar suggested a novel processor-memory interconnection architecture based on finite projective geometry. The processors and memories are associated with elements of these geometries and the interconnections are based on the incidence relations between these elements. The computations assigned to a processor also depend on the geometry and incidence relations and because of the symmetric nature of the geometry, the computational load on each processor is balanced. The automorphisms governing these geometries are used to develop 'perfect-access patterns' and 'perfect-access sequences', which ensure that all the processors and memories are simultaneously involved in communication of data without any conflicts. Algorithms to solve various problems on this architecture can be developed using these properties.

In the first part of this report, we take a closer look at Karmarkar's architecture and then focus on the problem of LU decomposition. We have devised schemes for distribution of data in the memory modules and distribution of computations on different processors based on a 4-dimensional projective geometry. In this report, we have evaluated two different approaches based on the amount of communication and computation required for solving a fixed problem. A comparison is also drawn with parallel algorithms based on conventional mesh architecture to judge the performance of these approaches.

We then take a look at another projective geometry based interconnection pattern called Perfect Difference Network, which were suggested by Parhami and Rakov in [3]. We look at some of the important features of these networks such as a constant diameter of 2, smaller node degree and higher bisection width. We also study the communication primitives suggested in [4].

In the final part, we investigate the use of projective geometry in error-correcting codes, especially in expander codes. We study the construction of regular bipartite graphs based on projective geometry. The findings from simulations carried

out on such expander codes have also been reported.

The organisation of this report is as follows. The next section deals with the study of Karmarkar's architecture and development of the geometry and its automorphisms needed for LU decomposition. In the third section, we look at the approaches we have developed to solve LU decomposition on Karmarkar's architecture. We also discuss our results from the simulations and present a comparative study. The following section deals with perfect difference networks, its properties and study of some communication algorithms in these networks. In the last section, the work by Sipser and Spielman, and Zemor on expander codes is studied and a software implementation of these codes has been discussed. The appendix, at the end of the report, details the construction of finite fields and projective spaces used in our implementations.

2 Karmarkar's Architecture

As stated before, Karmarkar's architecture defines the interconnection pattern between processors and memories based on finite projective geometry. A finite geometry of dimension d consists of a set of points $\mathbb{S}$, which form the zero-dimensional subspaces. These points can be grouped together to form subspaces of higher dimensions $(1, \ldots, d)$. The subspaces of dimension 1 are called lines, 2-dimensional subspaces are called planes and the $d-1$-th dimensional subspaces are called hyperplanes. Once the appropriate geometry for a problem has been identified, a pair of dimensions d_m and d_p are chosen. The processors are associated in one-to-one correspondence with the subspaces of dimension d_p while the memories are associated with subspaces of dimension d_m and a connection between a processor and memory is established if the corresponding subspaces have a non-trivial intersection.

The access of memory is done in a structured fashion. By exploiting the symmetry of the geometry, it is possible to identify processor-memory pairs, involving all the processors and memories, which can communicate in a conflict-free manner. Each such set of processor-memory pairs forms a perfect-access pattern. A collection of all such patterns together forms a perfect-access sequence, which ensures that every processor gets to communicate with every memory it is directly connected to.

For the distribution of computational work between processors, first the problem is broken down into atomic computations and operations that can be carried out parallely are considered together. Then the memories, which contain the operands needed for a particular operation, are identified and the operation is assigned to the processor connected to these relevant memories, which is unique for most operations and depends on the problem and the underlying geometry. The symmetry of the geometry ensures that a balance is maintained in the distribution of load among the processors. Thus, the data required for computation is brought in parallely using parallel access sequences and the computations are then carried out parallely on each processor, ensuring efficient use of resources while avoiding conflicts and maintaining load balance.

2.1 Projective Spaces Over Finite Fields

In this section, we look at how the projective spaces are generated from finite fields.

Consider a finite field $F = \mathrm{GF}(s)$ with s elements, where s is a power of a prime number p i.e. $s = p^k$, k being a positive integer. A projective space of dimension d is denoted by $\mathbb{P}(d, F)$ and consists of one-dimensional subspaces of the $(d+1)$-dimensional vector space over F (an extension field over F), denoted by F^{d+1}. Elements of this vector space are of the form $(x_1, \ldots, x_{d+1})$, where each $x_i \in F$. The total number of such elements are $s^{(d+1)} = p^{k(d+1)}$. An equivalence

relation between these elements is defined as follows: two non-zero elements $\mathbf{x}$, $\mathbf{y}$ are equivalent if there exists an element $\lambda \in \mathrm{GF}(s)$ such that $\mathbf{x} = \lambda\mathbf{y}$. Clearly, each equivalence class consists of s elements of the field ($s-1$ non-zero elements and $\mathbf{0}$) and forms a one-dimensional subspace which corresponds to a point in the projective space. Points are the zero-dimensional subspaces of the projective space. Therefore, the total number of points in $\mathbb{P}(d, F)$ are

$$P(d) = \frac{\text{number of non-zero elements in the field}}{\text{number of non-zero elements in one equivalence class}} \tag{1}$$

$$= \frac{s^{d+1}-1}{s-1} \tag{2}$$

An m-dimensional subspace of $\mathbb{P}(d, F)$ consists of all the one-dimensional subspaces of an $(m+1)$-dimensional subspace of the vector space. The basis of this vector subspace will have $(m+1)$ linearly independent elements, say $b_0, \ldots, b_m$ and every element of the subspace can be represented as a linear combination of these basis vectors.

$$\mathbf{x} = \sum_{i=0}^{m} \alpha_i b_i, \text{ where } \alpha_i \in F(s) \tag{3}$$

Clearly, the number of elements in the vector subspace are $s^{(m+1)}$ and the number of points in the m-dimensional projective subspace are given by $P(m)$. This $(m+1)$-dimensional vector subspace and the corresponding projective subspace are said to have a codimension of $r = d - m$ (the rank of the null space of this vector subspace).

Let us denote the collection of all the l-dimensional projective subspaces by Ω_l. Now, Ω_0 represents the set of all the points of the projective space, Ω_1 is the set of all lines, Ω_2 is the set of all planes and so on. To count the number of elements in each of these sets, we define the function

$$\phi(n, l, s) = \frac{(s^{n+1}-1)(s^n-1)\ldots(s^{n-l+1}-1)}{(s-1)(s^2-1)\ldots(s^{l+1}-1)} \tag{4}$$

Now, the number of m-dimensional subspaces of $\mathbb{P}(d, F)$ is $\phi(d, m, s)$. For example, the number of points in $\mathbb{P}(d, F)$ is $\phi(d, 0, s)$. Also, the number of l-dimensional subspaces contained in an m-dimensional subspace (where $0 \leq l < m \leq d$) is $\phi(m, l, s)$, while the number of m-dimensional subspaces containing a particular l-dimensional subspace is $\phi(d-l-1, m-l-1, s)$.

2.2 LU Decomposition

We now look at the kind of computations that are involved in LU decomposition and justify our choice of 4-dimensional projective geometry as the underlying architecture.

Consider an $N \times N$ matrix A. The LU decomposition of A is its factorization into an upper triangular matrix, U and a lower triangular matrix L such that

$$A = LU \tag{5}$$

Rather than working with individual elements, we perform the LU decomposition by breaking A into blocks, which are then assigned to processors. The algorithm that we consider for mapping to Karmarkar's architecture is the block-level generalization of trailing matrix update algorithm, which is described below:

```
1:  for i=0:B-1 do
2:      A_{i,i} ← blockLU(A_{i,i})
3:      for j=i+1:B-1 do
4:          L_{j,i} ← A_{j,i} U_{i,i}^{-1}
5:      end for
6:      for k=i+1:B-1 do
7:          U_{i,k} ← L_{i,i}^{-1} A_{i,k}
8:      end for
9:      for j=i+1:B-1 do
10:         for k=i+1:B-1 do
11:             A_{j,k} ← A_{j,k} - L_{j,i} U_{i,k}
12:         end for
13:     end for
14: end for
```

The LU decomposition of a matrix is not unique. However, if we impose the constraint that all diagonal elements of L should be 1, then the decomposition becomes unique.

This algorithm has two advantages. Firstly, it offers a lot of parallelism, which can be exploited during load distribution. The updates at line (11) of the algorithm are independent of each other and can be carried out parallely on different processors. Secondly, it works inplace and the L and U matrices are generated and stored in the lower and upper halves of A respectively. Hence, no additional memory is required to store L and U and the memory requirements are minimal, which becomes significant when large matrices are being considered.

Clearly, the dominant operation being performed repeatedly in this algorithm is

$$A_{j,k} \leftarrow A_{j,k} - L_{j,i}U_{i,k} \tag{6}$$

From this equation, we see that for the update of $A_{j,k}$, we need the matrices $L_{j,i}$ and $U_{i,k}$, which, when created, occupy the same memory position as $A_{j,i}$ and $A_{i,k}$ respectively. So, for the computation, we only need blocks that have one index in common with the updated block. This property is exploited in adapting this algorithm for the Karmarkar architecture.

The basic concept proposed by Dr. Karmarkar in his paper [2] is as follows. The indices of the blocks of the matrix are mapped to points of the projective geometry. The memory modules are associated with the lines, while the processors are associated with the planes. The interconnection pattern depends on the incidence relations of the lines and planes of the geometry. The block $A_{i,j}$ is stored in the memory module corresponding to the line joining the points associated with i and j.

Each operation of the form shown in equation 6 can be characterised by the triplet (i, j, k) of indices of the elements involved, where i is the iteration number and j, k are the indices of the updated block. Now, in the geometry, any general triplet of points determines a plane. Hence, the computation involving (i, j, k) is assigned to the processor associated with the plane determined by the points corresponding to i, j and k. In performing this computation, this processor needs to interact with three memory modules, containing one of $A_{j,i}$, $A_{j,k}$ and $A_{i,k}$, and all these modules are connected to it directly according to our interconnection pattern. In the different approaches we have studied, these concepts have been slightly modified. However, the choice of the geometry is still governed by these considerations.

In a projective space, the number of m-dimensional subspaces is equal to the number of subspaces with codimension $m+1$. Therefore, for a symmetric scheme involving the same number of processors and memory modules, we need the geometry to have same number of lines and planes, i.e. $d = (2+1)+1 = 4$. Thus, the most appropriate geometry for this problem is the 4-dimensional projective geometry.

2.3 4-D Projective Geometry over GF(2)

We now look at the smallest 4-D projective geometry, generated over GF(2) containing 32 elements. First, the extension field GF(2^5) is created using the following polynomial of degree 5 over GF(2):

$$x^5 + x^2 + 1 = 0$$

By definition, the root of this equation, x, is one of the elements of the extension field and is also a generator of the extension field. We take successive powers of x and find their remainders when divided by $x^5 + x^2 + 1$ to generate the successive elements of the field. The generated field and its polynomials are shown in table 8 in the appendix.

Now, consider the one-dimensional subspaces of this 5-dimensional vector space. Each of them represents a point in $\mathbb{P}$(4,GF(2)). As discussed earlier, each one-dimensional subspace will have two elements, **0** and a non-zero element. Hence, each non-zero element of the field will give rise to one point in the geometry, thus giving us a total of 31 points in the field. In the following discussion, each

point (of the form $(0, x^q)$, $q \in 0, 1, \ldots, 30$) is represented by the index q of the non-zero element.

The number of points on each line is $\phi(1, 0, 2) = (2^2 - 1) = 3$. Also, there exists a line between any two distinct points. To obtain a line, we start with any two points, say 0 and 1, and combine the two one-dimensional subspaces associated with these points and find all the linear combinations of the elements involved. This generates a 2-dimensional subspace, which represents a line in the projective space. The third point on the line can be obtained by studying the elements generated as linear combinations of the starting elements. For example, on the line joining $0(0, 1)$ and $1(0, x)$, the third point is $18(0, x+1)$. The other line-point incidence relations can be generated using the automorphisms of the geometry. This process is described in the next section. In the following discussion, each line will be represented by a triplet indicating the three points that lie on it.

Next, we look at the generation of planes. The number of points on each plane is $\phi(2, 0, 2) = 7$, which is also the number of lines on a plane($\phi(2, 1, 2) = 7$). Further, there exists a plane between any two distinct lines that intersect. To form a plane, we start with two intersecting lines, say $(0, 1, 18)$ and $(0, 2, 5)$, and combine the corresponding 2-dimensional subspaces to form the 3-dimensional subspace representing the plane containing the two lines. The other points on the plane are obtained by studying the 3-dimensional subspace. For example, the plane containing $(0, 1, 18)$ and $(0, 2, 5)$ also contains the points 11 and 19. As in the case of lines, we use automorphisms to generate the other planes. Also, for further reference, we will use a 7-tuple (indicating the 7 points) for a plane.

A few important details about $\mathbb{P}$(4,GF(2)) should be noted before we proceed further.

- There are 31 points, 155 lines ($\phi(4, 1, 2)$) and 155 planes ($\phi(4, 2, 2)$) in the geometry. Also, there is a unique line passing through two distinct points and a unique plane passing through two lines intersecting in a point.
- Each line has 3 points on it while a plane has 7 points.
- Each plane has 7 lines (forms a Fano plane) and exactly 3 lines belonging to one plane pass through any point.
- A line is incident on 7 different planes.
- A point is present on 15 different lines ($\phi(3, 0, 2)$) and 35 different planes ($\phi(3, 1, 2)$).

2.3.1 Frobenius Automorphism

A field automorphism is a bijective ring homomorphism from the field to itself, i.e., it is a one-to-one, onto function $g : F \to F$ such that

$$g(a+b) = g(a) + g(b) \;\forall\; a, b \in F \tag{7}$$
$$g(ab) = g(a)g(b) \;\forall\; a, b \in F \tag{8}$$

From the definition, we have $g(0) = 0$ and $g(1) = 1$. Such a mapping preserves the structure of the field.

One such automorphism is the Frobenius map given by

$$\Phi(x) = x^p \tag{9}$$

where $x \in F$ and p is the characteristic of the field, which is 2 in our case. As there is a one-to-one correspondence between the elements of the field and the points, this map is also an automorphism of the projective space. In $\mathbb{P}$(4,GF(2)), the application of the automorphism corresponds to doubling of the index representing each point (i.e. finding x^2) and taking its remainder modulus 31, since $x^{2^5} = x \Rightarrow x^{31} = 1 \;\forall\; x \in$GF$(2^5)$ (for any field GF(q), $x^{q-1} = 1 \;\forall\; x \in$GF$(q)$). Using Frobenius map repeatedly, in addition to $x^2(\Phi(x))$, we get other automorphisms like $x^{2^2}(\Phi^2(x))$, $x^{2^3}(\Phi^3(x))$ and $x^{2^4}(\Phi^4(x))$ after which they begin to repeat. So in all, we get 5 automorphisms using Frobenius map.

2.3.2 Shift Automorphism

In the geometry, the point i corresponds to the one-dimensional subspace $(0, x^i)$, where x is a root of the polynomial x^5+x^2+1 over GF(2). Consider the following mapping:

$$L_x : (0, x^i) \to (0, x^{i+1}), \;\forall\; i \in 0, 1, \ldots, 30 \tag{10}$$

Clearly, the mapping L_x maps points from $\mathbb{P}$(4,GF(2)) to itself, i.e.,

$$L_x(i) = i + 1 \;\forall\; i \in 0, 1, \ldots, 29 \;\textit{and}\; L_x(30) = 0 \;\textit{as}\; x^{2^5-1} = x^{31} = 1 \tag{11}$$

Also, under this transformation, it is easy to verify that lines get mapped onto lines, planes onto planes and in general, any k-dimensional subspace onto another k-dimensional subspace. This map forms an automorphism of the projective space called shift automorphism. We can create higher order shift automorphisms by repeatedly applying L_x. In all, there are 31 shift automorphisms possible, L_x^0 to L_x^{30}, after which they start to repeat.

Using the Frobenius and shift automorphisms and a starting line or plane, it is possible to create the entire list of incidence relations. Consider the line $(0, 1, 18)$. Using Frobenius automorphisms, we get the lines $(0, 1, 18)$, $(0, 2, 5)$, $(0, 4, 10)$, $(0, 8, 20)$ and $(0, 16, 9)$. Applying the shift automorphisms on these, each line gives rise to 31 lines, hence, giving us all the 155 (31×5) lines. Similarly, we can obtain all the planes. The complete list of lines and planes is shown in table 10 in the appendix.

2.4 Perfect Matching Sequences

Using the automorphisms, we develop perfect matching sequences, which are bijective maps between lines and planes. This is possible because we have the same number of planes and lines, both of which have been generated using the same automorphisms. Of these, the mappings that are relevant to our work are those that map a plane to one of the 7 lines that lie on it.

Consider the first sequence, $S_1 : \Omega_2 \rightarrow \Omega_1$.

$$S_1(p) = L_x^a(\Phi^b(0,1,18)),\ \textit{if}\ p = L_x^a(\Phi^b(0,1,2,5,11,18,19)) \tag{12}$$

(The automorphism of a line or a plane is the set formed by the automorphisms of individual points on that line or plane)

We start by mapping the first plane $(0,1,2,5,11,18,19)$ to the first line $(0,1,18)$. Every other plane, obtained from plane $(0,1,2,5,11,18,19)$ by b $(0 \leq b \leq 4)$ Frobenius automorphisms followed by a $(0 \leq a \leq 30)$ shift automorphisms, is mapped to the line obtained by applying the same set of automorphisms, in the same order, to $(0,1,18)$. Varying a and b in their respective ranges, we obtain the complete mapping, which is shown in table 10. Similarly, mapping the first plane to one of the other lines lying on it, we can create six other such perfect matching sequences, S_2 to S_7, which are shown in the appendix.

After this brief introduction to the basic concepts about Karmarkar's architecture and the $\mathbb{P}$(4,GF(2)) geometry, in the next section, we look at the various strategies devised to solve the problem of LU decomposition on this architecture.

3 Problem-mapping strategies

There are 2 different schemes that we have discussed here. The two schemes are based on the complete geometry using all the 155 lines and planes. There are 155 processors and 155 memory modules in each case. However, they differ in their architecture and in the distribution of computational load over the processors.

For all the strategies, the number of blocks in each row and column in matrix A is a multiple of 31, as the block indices are associated with the points and there are 31 points in all. The indices are taken to be *zero-based*. The block indices are mapped to points by taking their remainder modulo 31. Therefore, we have the mapping function f : block indices $\rightarrow$ points as

$$f(b) = b \pmod{31} \tag{13}$$

3.1 Algorithm Mapping Scheme I

In this design, we have 155 processors and 155 memory modules and we use the entire $\mathbb{P}(4,\mathrm{GF}(2))$ geometry in defining the interconnection network. Each processor is connected to its own exclusive memory module, and this processor-memory pair is associated with a line. In addition, the processor is also associated with the plane mapped to the line through perfect matching S_1. The processor is directly connected to processor-memory pairs representing other lines on its plane. Hence, each processor is connected to 12 other processors - 6 processors that lie on its own plane and 6 other processors in whose plane it lies. For example, the processor-memory pair $(0, 1, 18)$ is associated with the plane $(0, 1, 2, 5, 11, 18, 19)$ and is connected to the processors $(0, 2, 5)$, $(1, 2, 19)$, $(11, 19, 0)$, $(1, 5, 11)$, $(2, 18, 11)$ and $(18, 19, 5)$, which lie on its plane and to processors $(13, 14, 0)$, $(30, 0, 17)$, $(27, 29, 1)$, $(18, 22, 28)$, $(23, 0, 12)$ and $(0, 16, 9)$, as it lies on planes associated with these processors [1].

3.1.1 Data Distribution

The distribution of data in the memory modules depends on the indices of the matrix block and the triplet of points representing each module. Consider the following function $M(i, j)$ from point doublets (elements of $\Omega_0 \times \Omega_0$) to Ω_1, which specifies the memory module for $A_{p,q}$ if $f(p) = i$ and $f(q) = j$.

$$M(\alpha, \beta) = \text{line joining points } \alpha \text{ and } \beta \ \forall\ \alpha, \beta \in 0, 1, \ldots, 30 \text{ and } \alpha \neq \beta \tag{14}$$

$$M(i, i) = L_x^i(\Phi^a(0, 1, 18)), a \in 0, 1, 2, 3, 4 \tag{15}$$

As can be seen from these equations, every block $A_{i,j}$, with distinct i, j, gets stored in the memory module with the corresponding points in its 3-tuple representation. For example, the blocks $A_{0,1}$ and $A_{32,31}$ go into the module $(0, 1, 18)$. This specifies the storage for all the non-diagonal blocks.

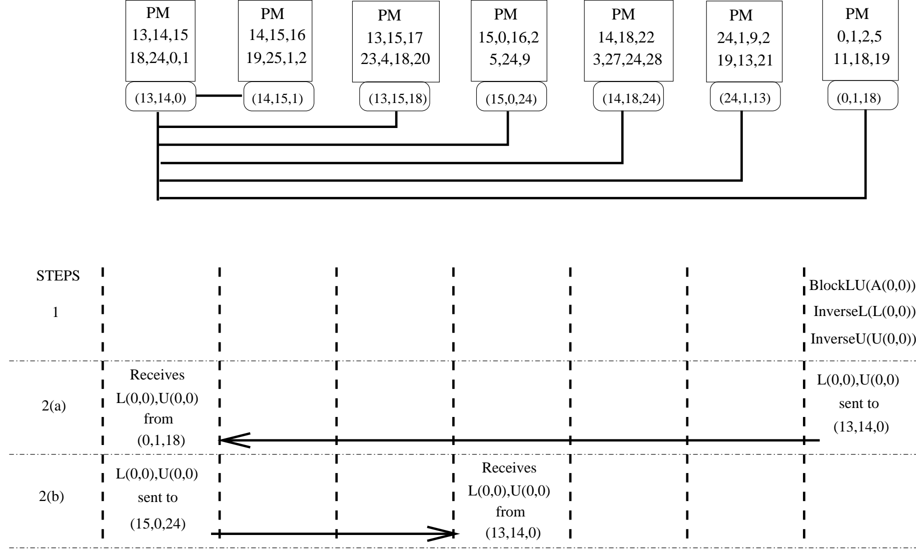


Figure 1: Scheme I: Diagonal block communication

Each diagonal block is stored in 5 memory modules. These 5 copies help in faster distribution of $L_{i,i}$ and $U_{i,i}$, as will be seen later. The 5 modules, in which p-th diagonal block gets stored, are those that correspond to lines obtained after applying $f(p)$ shift automorphisms on the line $(0, 1, 18)$ and its Frobenius automorphisms.

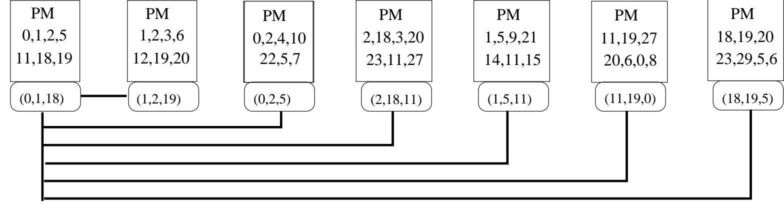
3.1.2 Operations Involved

In each iteration of the main loop, the operations to be carried out in LU decomposition are as follows:

- **Row/Column Update**: In the i-th iteration, the blocks present in the i-th column and i-th row have to be multiplied with $U_{i,i}^{-1}$ and $L_{i,i}^{-1}$, as shown in lines (4) and (7) of the algorithm respectively. This operation needs the blocks $A_{i,j}$ or $A_{j,i}$, and $A_{i,i}$ and can be characterized by a triplet (i, i, j) or (i, j, i) such that $j \in \{i+1, i+2, \ldots, B-1\}$.

- **Trailing Matrix Update**: These operations involve the update of all elements $A_{j,k}$ such that $j, k \in \{i+1, i+2, \ldots, B-1\}$ in the i-th iteration. These updates can be characterized by the triplet (i, j, k) of the indices involved in the operation.

We now look at the mapping of these operations onto the processors.



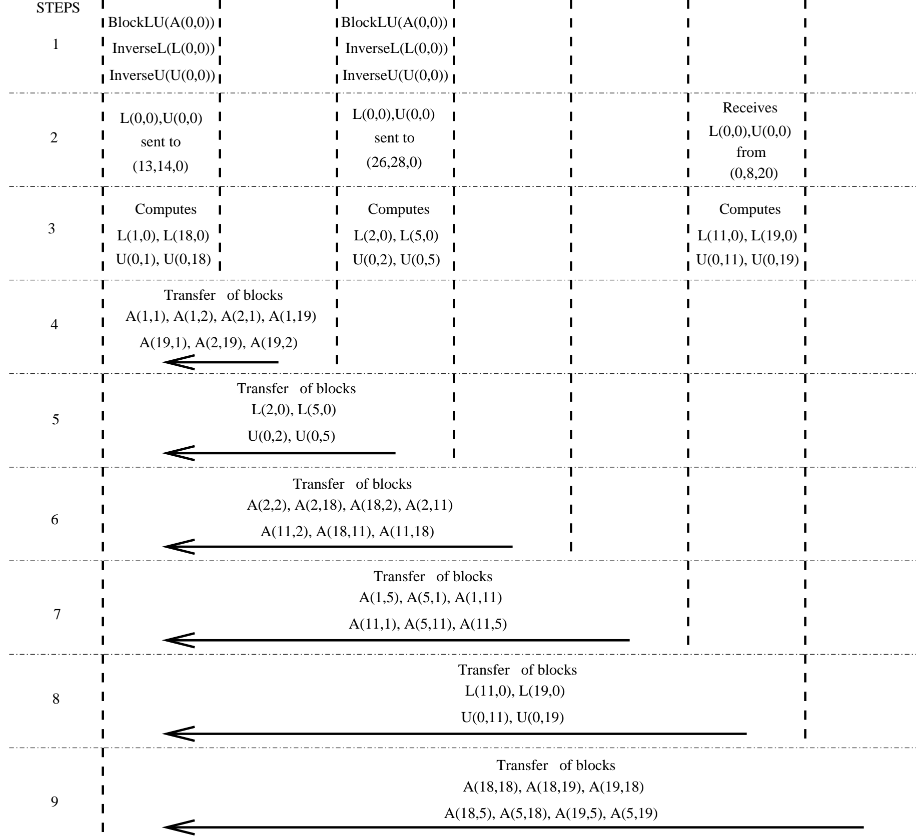
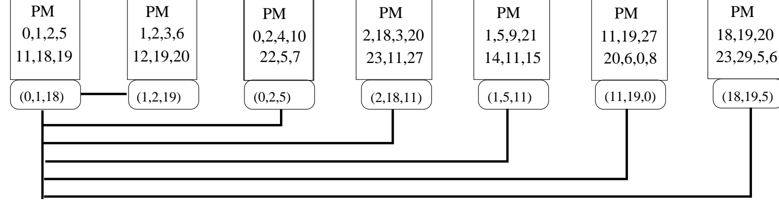


Figure 2: Scheme I: Execution of 0-th iteration on $(0, 1, 2, 5, 11, 18, 19)$ (steps 1-9)

3.1.3 Distribution Of Computations

The computations, which are represented by triplets of block indices are first converted to a triplet of points using the f map. These point triplets are dis-



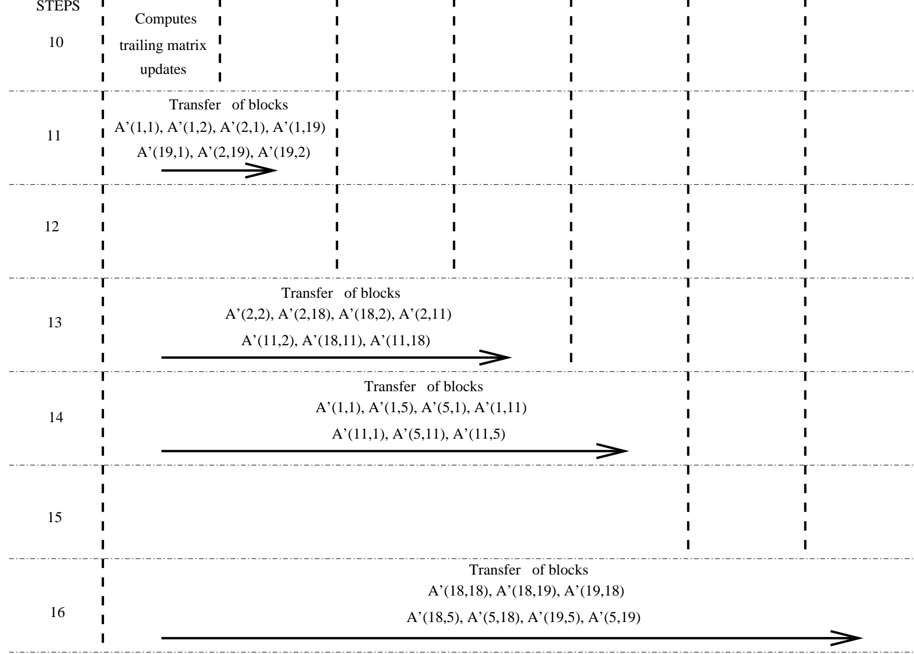


Figure 3: Scheme I: Execution continued (steps 10-16)

tributed according to the following map $P(\alpha,\beta,\gamma) : \Omega_0 \times \Omega_0 \times \Omega_0 \rightarrow \Omega_2$.

$$
\begin{aligned}
P(\alpha,\beta,\gamma) &= \text{plane through non-collinear points } \alpha,\ \beta \text{ and } \gamma && (16)\\
P(\alpha,\alpha,\beta) &= S_1^{-1}(\text{line joining } \alpha,\ \beta) && (17)\\
P(\alpha,\beta,\alpha) &= S_1^{-1}(\text{line joining } \alpha,\ \beta) && (18)\\
P(\alpha,\beta,\beta) &= \text{planes passing through } \alpha \text{ and the lines M}(\beta,\beta) && (19)
\end{aligned}
$$

As can be seen in the above equation, the computations corresponding to non-collinear triplets are allocated to the processor associated with the plane passing through that triplet. The column updates for the i-th iteration are carried out on the processors obtained by using the perfect matching sequence S_1^{-1} on each of the 15 memory modules associated with lines passing through point i. The update of a diagonal block is done alongwith the update of other blocks stored

Table 1: Transfer of diagonal data, 0th iteration

$(0,1,18) \xrightarrow{1^{st}Cycle} (13,14,0) \xrightarrow{2^{nd}Cycle} (15,0,24)$
$(0,2,5) \xrightarrow{1^{st}Cycle} (26,28,0) \xrightarrow{2^{nd}Cycle} (30,0,17)$
$(0,4,10) \xrightarrow{1^{st}Cycle} (21,25,0) \xrightarrow{2^{nd}Cycle} (29,0,3)$
$(0,8,20) \xrightarrow{1^{st}Cycle} (11,19,0) \xrightarrow{2^{nd}Cycle} (27,0,6)$
$(0,16,9) \xrightarrow{1^{st}Cycle} (22,7,0) \xrightarrow{2^{nd}Cycle} (23,0,12)$

in the same module.

3.1.4 Details of 0-th iteration

The distribution of computations is done according to the map P described above. Their execution is based on the 7 perfect matching sequences introduced earlier. Let us consider the 0-th iteration for describing the execution in detail. The computation of all updates is done on the processors associated with the planes containing the point 0. Some of the 35 processor-memory pairs involved in the computations are $(0,1,18)$, $(12,13,30)$, $(13,14,0)$, $(20,21,7)$ and $(26,27,13)$, all of which are matched to 0-containing planes through S_1. First, the 5 processor-memory pairs containing the block $A_{0,0}$ perform block-level LU decomposition, generating $L_{0,0}$ and $U_{0,0}$. This is followed by the computation of their inverses. This pair of inverses is then transferred to all the other 15 processor-memory pairs associated with 0-containing lines via a two-step broadcast through some specific processors. This data transfer schedule is shown in table 1. This communication is conflict-free as it is carried out on links associated with different planes. Fig. 1 shows the process being carried out on one particular plane $(13,14,15,18,24,0,1)$. The $2(B-1)$ row and column block updates are carried out by the 15 processor-memory pairs, to which they are mapped according to P. These processors compute $L_{j,0}$ and $U_{0,k}$, $j,k \in \{1,\ldots,B-1\}$. After this step, we perform the trailing matrix update.

As can be seen from P mapping, all the trailing matrix updates in the 0-th iteration are carried out on 35 processors associated with planes containing the point 0. Before starting the update of trailing matrix blocks, we need to move the $L_{j,0}$, $U_{0,k}$ and $A_{j,k}$ blocks to the processors involved in this round of computation. This communication is done in 6 steps. In the qth step, each of the computing processors receives data *directly* from the processor mapped to its plane through each *inverse* perfect matching pattern S_{q+1}^{-1}, $q \in 1,2,3,4,5,6$. Once these six steps are completed, each of the 35 processors will have complete information required to update the blocks it holds. This update computation is then carried out parallely on all the 35 processors. Once this is completed, we again have six communication steps to store back the updated blocks in the appropriate processor-memory pairs, in the same manner as they

were accessed earlier. An illustration of all these steps taking place on the plane $(0, 1, 2, 5, 11, 18, 19)$ in the case of 0-th iteration can be seen in figs. 2 and 3.

The scheme given above works for the 0-th iteration. For the ith iteration, the data transfer scheme is obtained by applying $f(i)$ shift automorphisms to all the involved entities.

3.1.5 Design Analysis

This design makes use of the entire $\mathbb{P}$(4,GF(2)) geometry and the perfect matching sequences, based on automorphisms, are applied to develop communication strategies. This makes the design scalable - the use of a bigger geometry and remapping of the problem is easy in this case.

The primary issue with this design is its under-utilization of parallelism. Either 15 or 35 out of 155 processors are working in each cycle, though they are load-balanced among themselves. As the iteration index i increases, the number of computations associated with line (11) of the algorithm decreases, which leads to corresponding reduction in number of operational processors. Also, in later iterations, because of duplicacy of diagonal elements, certain blocks are communicated to extra number of processors. With these considerations in mind, we provide our next scheme, which improves upon the resource usage as well as the wiring density.

3.2 Algorithm Mapping Scheme II

As in the earlier design, we have 155 processors and 155 memory modules in this scheme, with each memory module connected exclusively to one processor. Each such processor-memory pair is associated with a line of the geometry. These lines are connected by buses corresponding to the planes. Each bus is connected to the 7 processors associated with the lines that are incident on its representative plane. Thus, we have 155 buses in all. Also, as a line is incident on 7 planes, each processor lies on 7 busses. Therefore, the processor node degree, in this design, is less than in the earlier one. The distribution of data in the memory modules is the same as in the earlier scheme and is governed by $M(i, j)$.

3.2.1 Distribution of Computational Load

The distribution of computational load is done differently. This can be understood by the following modified function P_2 for computation allotment:

$$\begin{aligned} P_2(\alpha, \beta, \gamma) &= \text{line through distinct points } \beta \text{ and } \gamma && (20) \\ P_2(\alpha, \alpha, \beta) &= \text{line joining } \alpha \text{ and } \beta && (21) \\ P_2(\alpha, \beta, \alpha) &= \text{line joining } \alpha \text{ and } \beta && (22) \\ P_2(\alpha, \beta, \beta) &= \text{line allocated } A_{\beta,\beta} && (23) \end{aligned}$$

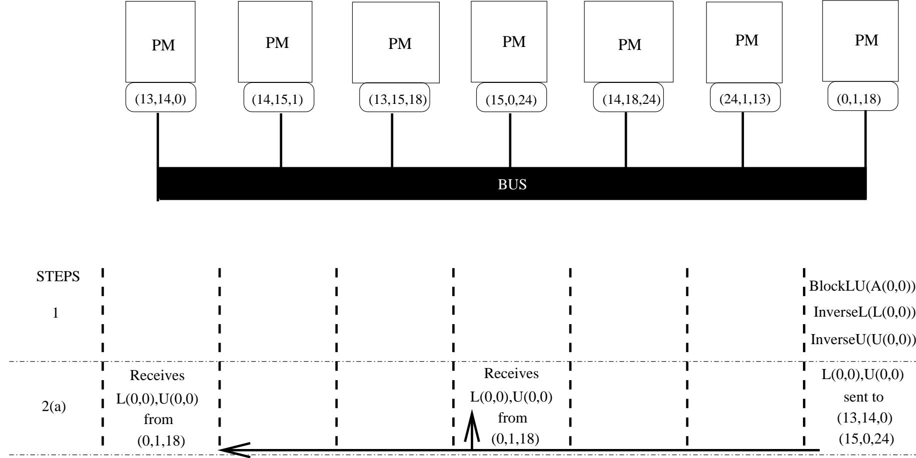


Figure 4: Scheme II: Diagonal block communication

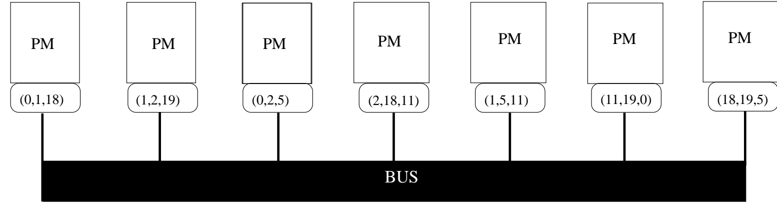
In this scheme, each processor computes the update for all the elements in its own memory, with the 5 copies of each diagonal element being updated in 5 different modules.

3.2.2 Details of 0-th iteration

We will again study the 0-th iteration in detail to understand the execution of the computations on this architecture. Just as in the earlier case, the computation begins with the five processors, which store $A_{0,0}$ block, performing block-level LU decomposition, generating $L_{0,0}$ and $U_{0,0}$. This is followed by computation of their inverses. These inverses are then transferred to other processors associated with 0-containing lines through 5 busses, which are

- $(0, 1, 18)$ on bus $(13, 14, 15, 18, 24, 0, 1)$
- $(0, 2, 5)$ on bus $(26, 28, 30, 5, 17, 0, 2)$
- $(0, 4, 10)$ on bus $(21, 25, 29, 10, 3, 0, 4)$
- $(0, 8, 20)$ on bus $(11, 19, 27, 20, 6, 0, 8)$
- $(0, 16, 9)$ on bus $(22, 7, 23, 9, 12, 0, 16)$

The transfer of diagonal L and U blocks can be better understood in fig. 4. This figure shows the steps of the transfer on bus $(13, 14, 15, 18, 24, 0, 1)$. The $2(B-1)$ row/column updates are scheduled now on each of the 15 processors with blocks $A_{0,i}$ and $A_{i,0}$, $i \in \{1, \ldots, (B-1)\}$. The data required for this update is already present on these processors. For the trailing matrix updates, each processor will require the calculated $L_{i,0}$ and $U_{0,i}$, which is present on 15 processors associated with 0 containing lines. These blocks are communicated to other processors for



STEPS							
1	BlockLU(A(0,0)) InverseL(L(0,0)) InverseU(U(0,0))		BlockLU(A(0,0)) InverseL(L(0,0)) InverseU(U(0,0))				
2	L(0,0),U(0,0) sent to (13,14,0) (15,0,24)		L(0,0),U(0,0) sent to (26,28,0) (30,0,17)			Receives L(0,0),U(0,0) from (0,8,20)	
3	Computes L(1,0), L(18,0) U(0,1), U(0,18)		Computes L(2,0), L(5,0) U(0,2), U(0,5)			Computes L(11,0), L(19,0) U(0,11), U(0,19)	
4		L(1,0), U(0,1)	Broadcasts L(2,0), L(5,0) U(0,2), U(0,5) on (0,2,4,10,22,5,7)	L(18,0), U(0,18)	L(1,0), U(0,1)	Broadcasts L(11,0), L(19,0) U(0,11), U(0,19) on (11,19,27,20,6,0,8)	L(18,0), U(0,18)
5	Broadcasts L(1,0), L(18,0) U(0,1), U(0,18) on (30,0,1,4,10,17,18)		Broadcasts L(2,0), L(5,0) U(0,2), U(0,5) on (29,0,2,8,20,3,5)	NO TRANSFER OF DATA ON THIS BUS		Broadcasts L(11,0), L(19,0) U(0,11), U(0,19) on (3,11,19,12,29,23,0)	
6	Broadcasts L(1,0), L(18,0) U(0,1), U(0,18) on (0,16,1,18,21,9,25)	L(2,0), U(0,2)		L(2,0), U(0,2)	L(5,0), U(0,5)	Broadcasts L(11,0), L(19,0) U(0,11), U(0,19) on (11,15,19,0,24,21,25)	L(5,0), U(0,5)
7	Broadcasts L(1,0), L(18,0) U(0,1), U(0,18) on (27,29,0,6,18,1,3)		Broadcasts L(2,0), L(5,0) U(0,2), U(0,5) on (23,27,0,12,5,2,6)	NO TRANSFER OF DATA ON THIS BUS		Broadcasts L(11,0), L(19,0) U(0,11), U(0,19) on (10,26,11,28,0,19,4)	
8	Broadcasts L(1,0), L(18,0) U(0,1), U(0,18) on (23,0,8,1,18,12,20)		Broadcasts L(2,0), L(5,0) U(0,2), U(0,5) on (15,0,16,2,5,24,9)	NO TRANSFER OF DATA ON THIS BUS		Broadcasts L(11,0), L(19,0) U(0,11), U(0,19) on (9,11,13,19,0,14,16)	
9	Broadcasts L(1,0), L(18,0) U(0,1), U(0,18) on (18,22,26,7,0,28,1)	L(19,0), U(0,19)	Broadcasts L(2,0), L(5,0) U(0,2), U(0,5) on (26,28,30,5,17,0,2)	L(11,0), U(0,11)	L(11,0), U(0,11)		L(19,0), U(0,19)
10	Broadcasts L(1,0), L(18,0) U(0,1), U(0,18) on (13,14,15,18,24,0,1)			NO TRANSFER OF DATA ON THIS BUS		Broadcasts L(11,0), L(19,0) U(0,11), U(0,19) on (22,30,7,0,17,11,19)	
11	Computes A'(1,18), A'(18,1)	Computes A'(1,1) A'(1,2), A'(2,1) A'(1,19),A'(19,1) A'(2,19),A'(19,2)	Computes A'(2,5) A'(5,2)	Computes A'(2,2) A'(2,18), A'(18,2) A'(2,11),A'(11,2) A'(18,11),A'(11,18)	Computes A'(1,1) A'(1,5), A'(5,1) A'(1,11),A'(11,1) A'(5,11),A'(11,5)	Computes A'(11,19) A'(19,11)	Computes A'(18,18) A'(18,19), A'(19,18) A'(18,5),A'(5,18) A'(19,5),A'(5,19)

Figure 5: Scheme II: Execution of 0-th iteration on bus $(0, 1, 2, 5, 11, 18, 19)$

calculation of trailing matrix updates. This communication is carried out in 7 steps. In q^{th} step, the processor with this information broadcasts it on the bus mapped to it through perfect matching pattern S_q^{-1}, $q \in 1, \ldots, 7$. By the end of these steps, the data has been broadcast once on each of the 7 busses connected to a particular processor. A processor represented by line (x, y, z) will need $L_{i,0}$ and $U_{0,i}$, if one of its indices is same as i. Now, the line corresponding to this processor and the line through the points 0 and i share a common point (i), and hence, form a plane and the data transfer happens on its associated bus. So at some point, this processor will get the required data. At the end of these 7 steps, each processor will have the entire information needed by it to calculate the trailing matrix updates. All the 155 processors now parallely compute the updates for the trailing matrix blocks that they possess, thus completing the 0th iteration. All these steps have been illustrated in fig. 5, which shows the execution of the 0-th iteration on the bus $(0, 1, 2, 5, 11, 18, 19)$.

As in the earlier design, for higher iteration indices, the scheme for data transfer and distribution of computation is obtained by using shift automorphisms on all the involved entities.

3.2.3 Design Analysis

The use of complete geometry and perfect matching sequences based on automorphisms to create communication patterns indicates that this design will be scalable. Because of this, a larger 4-dimensional projective geometry can be applied in this design. Also, the distribution of computation is almost balanced through most iterations and the source of imbalance is mainly the reduction in the size of trailing matrix for higher iterations. The resource usage is high compared to the earlier scheme and also the communication is limited.

The main issue in this design is the large number of buses involved and their latencies in broadcasting data. Alternate interconnection patterns can also be considered to alleviate this problem.

3.3 Simulation Results

To verify the functional correctness of our schemes, C++ programs were developed for both the schemes. For performance comparison, a mesh-based scheme was also developed and simulated. Our schemes require 155 processors, and hence, the mesh size was chosen to be 12×12. In our implementation, having B(number of blocks per row/column) as multiple of 31 helps in one-to-one mapping to lines. Hence for comparability, a matrix size of 372(31×12) or multiples of it is best suited. For simulations, we have taken the matrix size to be 744×744.

By simulation on a uniprocessing system, the three schemes were indeed found to be working correctly. The performance of these schemes can be understood as follows.

Block size	Normalized time	
	$O(b^3)$	$O(b^2)$
62×62	17.2400	0.2776
31×31	2.1550	0.0694
24×24	1.0000	0.0416
12×12	0.1250	0.0104
8×8	0.0370	0.0046
6×6	0.0156	0.0026
4×4	0.0046	0.0012

Table 2: Block Sizes and Relative Time Periods

Scheme	Block size	Total Cycles		
		Comp ($O(b^3)$)	Comm ($O(b^2)$)	Sub ($O(b^2)$)
Mesh	62×62	69	221	11
	31×31	196	583	56
PG, 1st	24×24	856	2241	651
	12×12	5023	12635	4427
	8×8	15291	37981	14118
	6×6	34450	84975	32514
PG, 2nd	24×24	393	846	188
	12×12	1693	2994	1097
	8×8	4458	6444	3285
	6×6	9246	11196	7310
	4×4	27123	24606	23103

Table 3: Total communication and computation cycle count for the three schemes

The active period of each processor can be classified into three categories. A processor either spends time doing $O(b^3)$ computations, or $O(b^2)$ computations, or $O(b^2)$ communication. The $O(b^3)$ computations comprise of block LU decompositions, (block) matrix inversions and (block) matrix multiplications. The $O(b^2)$ computations comprise of matrix subtraction done during trailing update. Each block has b^2 elements, and therefore, any transfer of a single block during the algorithm's run requires $O(b^2)$ cycles. Varying the block size results in different relative values of b^3 and b^2. The block sizes that we consider, and these related values have been tabulated in table 2. We take 24^3 as the normalization factor.

The total number of cycles taken, and the overall time taken for mesh scheme as well as the two PG schemes are presented in tables 3 and 4. The overall time for a computation or communication of particular complexity ($O(b^3)$ or $O(b^2)$)

Scheme	Block size	Time required		
		Comp ($O(b^3)$)	Comm ($O(b^2)$)	Sub ($O(b^2)$)
Mesh	62×62	1189.56	61.35	3.05
	31×31	422.38	40.46	3.89
PG, 1st	24×24	856.00	93.23	27.08
	12×12	627.88	131.40	46.04
	8×8	565.77	175.47	65.23
	6×6	538.45	220.94	84.54
PG, 2nd	24×24	393.00	35.19	7.82
	12×12	211.63	31.14	11.41
	8×8	164.95	29.77	15.18
	6×6	144.47	29.11	19.01
	4×4	124.77	28.42	26.68

Table 4: Total time required for the three schemes

Scheme	Block size	Average Cycles		
		Comp ($O(b^3)$)	Comm ($O(b^2)$)	Sub ($O(b^2)$)
Mesh	62×62	4	7	3
	31×31	34	41	30
PG, 1st	24×24	82	171	73
	12×12	669	1234	633
	8×8	2319	4050	2238
	6×6	5590	9485	5446
PG, 2nd	24×24	82	44	73
	12×12	669	214	633
	8×8	2319	510	2238
	6×6	5590	932	5446
	4×4	19227	2154	18903

Table 5: Average Active Processor Cycles

is calculated as total number of cycles multiplied by the normalized time for that complexity and block size, as shown in table 2. The schemes presented in this report are referred in this table by PG, 1st and PG, 2nd, respectively. One can infer from this table that for comparable block sizes(such as 24×24 and 31×31), the number of cycles taken by the second scheme is somewhat more than that taken by mesh scheme. This can be attributed to presence of more blocks due to smaller block size and the extra computation done on diagonal blocks in our schemes. If one compares the normalized time required for these schemes and corresponding sizes, however, the PG scheme performs better than the mesh scheme. Across all the schemes, one can further see that as block size decreases, the performance improves due to more fine-grained distribution of computational load.

We also calculated the *average* number of cycles in which each processor is active for the three different categories. The results are shown in table 5. Average processor utilization in each category can be defined as the average number of cycles taken by processor in that category, divided by total number of cycles (from table 3) in which all processors finish that category's job in parallel. Here we see from the table that our schemes have higher processor utilization than the mesh scheme for all different block sizes. Between the two schemes, for each given block size, the 2nd scheme needs much less amount of average communication cycles while having same average computation cycles, and hence improves upon the 1st scheme.

3.4 Conclusions

We have introduced two new schemes for processor interconnection based on projective geometry graphs, which work efficiently for the computation of LU decomposition. To do that, we make use of 4-dimensional projective spaces and its automorphisms. Results show that in terms of total time required, the 2nd scheme does better than the conventional mesh-based scheme. The schemes are currently designed for well-conditioned matrices not requiring pivoting. It needs to be seen whether pivoting can be implemented using perfect access sequences. Also, issues related to synchronization of the data communication steps need to be addressed. Another direction of research is to make these schemes handle large-sized matrices and also, sparse matrices with varying sparsity structures. Scalability of this architecture should also be evaluated by using 4-dimensional geometries over higher fields. The current simulation-based performance evaluation was done on a uniprocessor, but in future design of a parallel prototype and its evaluation should be tried.

4 Perfect Difference Networks

Similar to the projective geometry based graphs, there is a family of networks called perfect difference networks, which were introduced by Parhami and Rakov [3],[4]. These networks have a diameter of 2 with node degree being $O(n^{0.5})$ and bisection width being $\Omega(n^{1.5})$ (where n is the total number of nodes in the graph). In this section, we take a look at the construction and properties of these networks.

4.1 Perfect Difference Sets

Perfect difference set (PDS) is a set of $p+1$ distinct integers $\{s_0, s_1, \ldots, s_p\}$ which satisfy the condition that the $p(p+1)$ differences $s_i - s_j, 0 \leq i, j \leq p$, $i \neq j$ are congruent to the numbers $0, 1, \ldots, p^2+p \bmod p^2+p+1$ in some order [3]. Such set is guaranteed to exist for any power of a prime number. Given a PDS $\{s_0, \ldots, s_p\}$, the set $\{s_0 - b, \ldots, s_p - b\}$ also forms a PDS.

A PDS $\{s_0, s_1, \ldots, s_p\}$ is said to be in its normalised form if it contains 0 and 1 and if its elements satisfy the property that $s_i < s_{i+1} \leq \delta^2 + \delta$, $0 \leq i < \delta$. Any PDS can be converted into its normalised form by finding the pair s_v, s_u such that $s_v - s_u = 1 \pmod{p^2+p+1}$ and subtracting s_u from each of the integers. This normalized set is of the form $\{0, 1, s_2, \ldots, s_p\}$. Two PDSs are said to be equivalent if both have the same normal form. There are multiple normalised PDSs possible for a particular order p. For example, for order 2, we have the sets $0, 1, 3$ and $0, 1, 5$. Similarly, both the sets $0, 1, 3, 9$ and $0, 1, 4, 6$ form PDS of order 3. This multiplicity gives us alternate network designs for the same number of nodes.

The creation of these sets is based on projective planes. Consider the projective plane over GF($p = a^b$), where a is a prime and b is a positive integer. To obtain this plane, first, a Galois field is generated using a primitive polynomial of degree 3 over GF(p). Each element of the field can be expressed as a power of the generator element. The points on the projective plane correspond to 1-dimensional subspaces of GF(p). These subspaces and the corresponding points are indexed by the smallest power of the generator element belonging to it. Lines on the projective plane correspond to 2-dimensional subspaces of the field. The indices of the points lying on a particular line form a perfect difference set. Some normalized PDSs are shown in the table 6 below.

4.2 Networks based on PDS

We now define perfect difference networks based on these sets. The network has $n = p^2+p+1$ nodes, numbered 0 to $n-1$. Node i is connected to nodes $i \pm 1 \pmod{n}$ and $i \pm s_j \pmod{n}$ where $2 \leq j \leq p$. The links connecting i to $i + s_j$ are called forward links, while the links connecting it to $i - s_j$ are called backward links. The smallest such network with 7 nodes is shown in fig. 6.

p	p^2+p+1	Perfect Difference Set of order p
2	7	{0,1,3}
3	13	{0,1,3,9}
4	21	{0,1,4,14,16}
5	31	{0,1,3,8,12,18}
7	57	{0,1,3,13,32,36,43,52}
8	73	{0,1,3,7,15,31,36,54,63}

Table 6: Normalized Perfect Difference Sets [3]

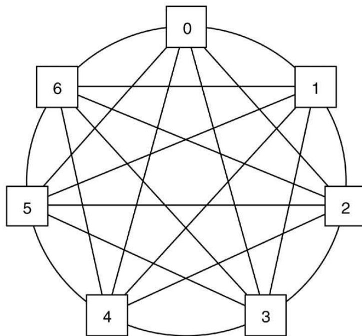


Figure 6: Perfect Difference Network with 7 nodes [3]

This network has a degree $d = 2p$. Hence, its node degree is $O(n^{0.5})$. The diameter of the network is 2 as to reach node y from node x, the intermediate node is obtained by first expressing the difference $y-x$ as s_u-s_v (s_u, s_v belonging to the PDS) and then taking $x+s_u$ mod n or $x-s_v$ mod n. In addition to the small diameter, PDN has the following topological properties [3]:

1. Average internode distance of a PDN of order p is $\frac{2p^2}{p^2+p+1}$.

2. A lower bound on the bisection width of PDN is $\lceil \frac{(p+1)(p^2+p+2)}{4} \rceil$.

4.3 Routing Algorithms

In this section, routing algorithms suited for perfect difference networks have been discussed [4]. We first consider the oblivious shortest-path routing algorithm from source node x to destination y. If y is of the form $x \pm 1$ or $x \pm s_j$, then a direct link exists between the two nodes, which forms the shortest path between them. If this is not the case, then a two-hop path exists between the nodes. To find the intermediate node, we find two numbers, s_u and s_v belonging to the PDS such that $s_u - s_v = x - y$. Using this, we get two paths - one through $x + s_v$ mod n and another through $x - s_u$ mod n. The first path uses a forward link followed by a backward link while the second path uses a backward

link first. If we consider that the first path (with the forward link being used first) is taken in all such two hop cases, then we find that, for random traffic between nodes, each link is used in the forward as well as backward directions for δ nodes. Therefore, under a random communication pattern, this algorithm will show a balanced distribution of communication load. If the messages are evenly distributed among the $n(n-1)$ possible messages, the number of hops required per message is $\frac{(n(n-1-2p)*2)+(2pn*1)}{n(n-1)} = \frac{2p^2}{(n-1)}$ and the number of messages passing over each link is $\frac{\frac{2p^2}{(n-1)}*n(n-1)}{np} = 2p$. Hence, a PDN can emulate a complete graph with a slowdown of $2p$. Pipelining the data transfer can reduce the slowdown to p.

Another class of important communication algorithms are broadcasting algorithms. In this network, a one-to-all broadcast can be carried out in 2 steps as the diameter is 2. In the first step, the broadcasting node x sends data to its forward neighbours using the forward links. In the second step, each of the $p+1$ nodes with the data send it to the rest of the nodes on the backward links. The number of steps required in this case is $2p$, p for the data transmission over forward links and p for the second step.

For all-to-all broadcast, the number of steps required is $p^2 + p$. First, each node transmits the packets to each of its p forward neighbours in p steps. This transmission is conflict-free if done using perfect access patterns[2]. Then, each node transmits p packets to each of its p backward neighbours, requiring p^2 steps. This step is also carried out using perfect access patterns.

To implement complete exchange (where a personalized message is sent by each node to every other node) in a PDN, in the first $2p$ transmissions, each node sends personalized messages to its neighbours. For the messages that need two hops, each node first sends $p-1$ messages to each of its forward neighbours, which in turn send them on their backward links to their intended recipients. Both steps use perfect access patterns and in all, there will be $2p+2p(p-1) = 2p^2$ message transmissions, which is optimal as each node sends p^2+p messages and each message requires $\frac{2p^2}{p^2+p}$ hops on an average.

Implementation of shift permutation, where each node x communicates with node $x + c$ (c being a constant), in a PDN can also be done through perfect access patterns. If $c = s_i$, then all nodes can parallely send their messages in a single step. For any other c, two elements, s_u and s_v, of the PDS need to be identified such that $c = s_u - s_v$, and the path for data transmission then becomes $x \rightarrow x + s_u \rightarrow x + c$.

These communication primitives can be used to develop more involved algorithms and also for mapping algorithms created for other networks such as complete graph, hypercube, etc. onto these networks.

4.4 Edge Expansion of Perfect Difference Networks

A family $\mathcal{G} = \{G_1, G_2, \ldots\}$ of d-regular graphs is an edge expander family if there is a constant $c > 0$ such that the edge expansion $h(G) \geq c$ for all $G \in \mathcal{G}$. The edge expansion is calculated as

$$h(G) = \min_{1 \leq |S| \leq \frac{n}{2}} \frac{|\partial(S)|}{|S|} \tag{24}$$

where the minimum is over all non-empty sets S of atmost $\frac{n}{2}$ vertices from G and $\partial(S)$ is the set of edges with exactly one endpoint in S.

By choosing different values for p, we get a family of PDNs such that the degree $d = O(|G|^{0.5})$. We now prove that this family of PDNs satisfy the edge expansion condition and hence, form an expander family. For this, we consider two graphs, each with p^2+p+1 nodes. In each graph, the nodes are numbered from 0 to p^2+p. The first graph G_1 is a complete graph such that any two vertices are connected by one and only one edge. The second graph G_2 is the perfect difference network over GF(p). For the case of GF(2), the two graphs are as shown in fig. 7. We also look at the projective plane over GF(p) with p^2+p+1 points and as many lines, the points are numbered as in the other graphs.

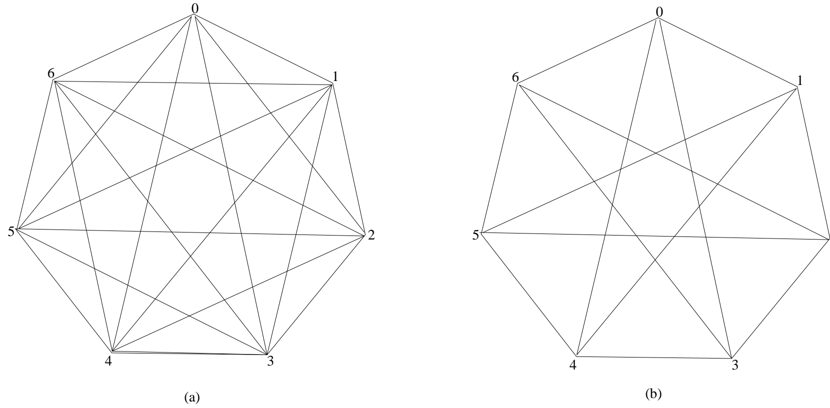


Figure 7: (a)Complete Graph, G_1 (b)Perfect Difference Network, G_2

Let us now consider a cut in the two graphs, separating the nodes into two sets, x and $S-x$ (without loss of generality, let $|x| < |S-x|$). In G_1, the number of edges passing across this cut is $|x| * |S-x|$. Now, any line in the projective plane consists of $p+1$ points and corresponds to $\frac{(p+1)p}{2}$ edges in G_1. Of these, atmost $\lfloor\frac{p+1}{2}\rfloor\lceil\frac{p+1}{2}\rceil$ edges can pass across the cut (when $\lfloor\frac{p+1}{2}\rfloor$ nodes are on one side of the cut and $\lceil\frac{p+1}{2}\rceil$ are on the other side). Therefore, the minimum

number of lines, whose corresponding edges in G_1 pass across the cut, are

$$\frac{|x| \times |S - x|}{\lfloor \frac{p+1}{2} \rfloor \lceil \frac{p+1}{2} \rceil} \tag{25}$$

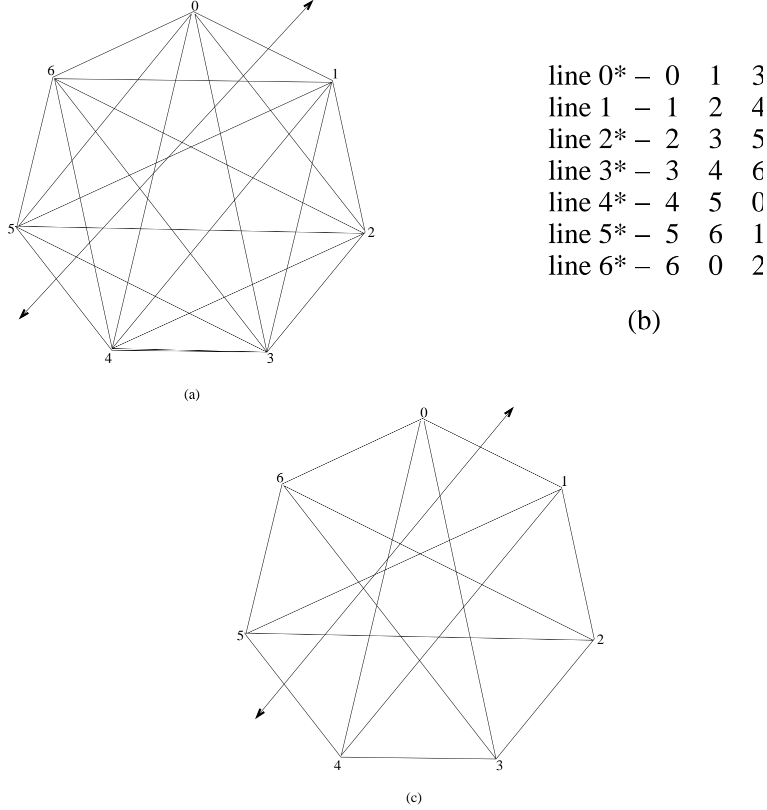


Figure 8: (a)Cut shown in Complete Graph, G_1 (b)Lines in PG with points on either side of the cut(marked with *) (c)Cut in Perfect Difference Network, G_2

Each of these lines of the projective geometry has points corresponding to nodes lying on both sides of the cut. Therefore, in G_2, each of these lines will contribute to atleast one edge passing across the cut. Therefore, the edges in G_2 passing across the cut are atleast equal to the minimum number of lines given in 25. The edge expansion of the graph, therefore, is

$$h(G_2) = \min_{1 \leq |x| \leq \frac{p^2+p+1}{2}} \frac{|x| \times |S - x|}{\lfloor \frac{p+1}{2} \rfloor \lceil \frac{p+1}{2} \rceil \times |x|} \tag{26}$$

and the minimum, achieved when the difference between $|S - x|$ and $|x|$ is

minimum (i.e. $|x| = \frac{p^2+p}{2}$ and $|S-x| = \frac{p^2+p}{2}+1$), is

$$h(G_2) = \frac{p^2+p+2}{2\lfloor\frac{p+1}{2}\rfloor\lceil\frac{p+1}{2}\rceil} \tag{27}$$

$$\geq \frac{p^2+p+2}{2(\frac{p+1}{2})(\frac{p+1}{2})} \tag{28}$$

$$\geq 1.75 \textit{ for } p \in \mathcal{N} \tag{29}$$

4.5 Conclusions

Perfect Difference Networks exhibit some good properties such as diameter 2, node degree $O(n^{0.5})$ and bisection width $\Omega(n^{1.5})$. Parhami and Rakov have suggested some good communication primitives, using perfect access patterns, using which a complete graph of the same size can be simulated on a PDN with a slowdown of p. The use of these communication primitives and good topological properties in the development of fast algorithms for this network should be researched. The expansion properties of PDNs should also be studied further for their application as expander graphs.

5 Expander Graphs and Expander Codes

Expander codes, introduced by Sipser and Spielman in [5], are an asymptotically good family of linear error-correcting codes based on expander graphs. The formulation of the code is similar to Gallager's Low-Density Parity Check (LDPC) codes. Sipser and Spielman suggested an iterative decoding algorithm which corrected a constant fraction of errors. In a later paper [6], Zemor reported a new decoding algorithm, built on a special type of expander graphs, which improved the fraction by a factor of 12. We first study the relevant proofs and results from [5] and [6] and then understand a MATLAB implementation of these ideas.

5.1 Characteristics of a Linear Code

A q-ary linear error-correcting code of length n and rank k is a linear subspace $\mathbb{S}$ of F^n of dimension k, where $F = \mathrm{GF}(q)$. Each codeword belonging to this code will have k message symbols and the other $n-k$ symbols are determined by the message. The total number of codewords in $\mathbb{S}$ is q^k. Another important characteristic of the code is the minimum hamming distance d between any two codewords, which implies that the code can correct upto $\lfloor \frac{d-1}{2} \rfloor$ errors. The rate of the code is given by $r = \frac{k}{n}$. A linear code is generally described by the triplet (n, k, d).

5.2 Expander Graph

Let $G = (V, E)$ be a graph on n vertices. The expansion property of the graph is characterised by ϵ and δ such that for every $S \subset V$

$$|S| \leq \epsilon n \Rightarrow |y \in V : \exists x \in S \text{ such that } (x, y) \in E| > \delta |S| \tag{30}$$

Consider a (g, h)-regular unbalanced bipartite graph. Its vertices can be divided into two sets - one set having degree g and the other with degree h - such that no two vertices in the same set share an edge. Considering the expansion of one of these vertex sets, in a (g, h, ϵ, δ) expander, every subset of ϵ fraction of g-regular vertices expands by a factor of atleast δ. One method for obtaining such graphs is to take the edge-vertex incidence graph of an h-regular graph. The edge-vertex incidence graph of a graph $G = (V, E)$ is the bipartite graph with vertex set $E \cup V$ and edge set $\{(e, v) \in E \times V : v \text{ is an endpoint of } e\}$. Hence, the edge-vertex incidence graph of an h-regular graph on n vertices gives a $(2, h)$-regular unbalanced bipartite graph with n vertices on one side and $\frac{nh}{2}$ on the other.

5.3 Construction of Expander Codes

Assume that G is a (g, h)-regular unbalanced bipartite graph with n g-regular vertices and $\frac{gn}{h}$ h-regular vertices and $h > g$. Each of the n nodes on the larger side of the graph is associated with one of the symbols, called variable, of a code

of length n. The nodes on the other side represent constraints that the variables have to satisfy. Each constraint restricts the h variables connected to it in such a way that they form a codeword belonging to a linear code of length h. As all the constraints are linear, the resulting expander code is linear. A more formal definition is as follows [5]:

Let G be a (g, h)-regular unbalanced bipartite graph between n nodes $\{v_1, \ldots, v_n\}$ called variables and $\frac{gn}{h}$ nodes $\{C_1, \ldots, C_{\frac{gn}{h}}\}$ called constraints. Define a function $b(i, j)$ such that for every constraint C_i, the set $\{v_{b(i,1)}, \ldots, v_{b(i,h)}\}$ gives its neighbouring variables. Consider an error correcting code $\mathbb{S}$ of block length h. The expander code $\mathbb{C}(G, \mathbb{S})$ is a code of block length n given by $(x_1, \ldots, x_n)$ such that for each $i \in \{1, \ldots, \frac{gn}{h}\}$, the symbols $(x_{b(i,1)}, \ldots, x_{b(i,d)})$ form a codeword in $\mathbb{S}$.

If the underlying graph has good expansion properties and the sub-code is sufficiently good, the generated expander codes exhibit good properties. An important result in this regard is as follows [5]:

Let G be a $(g, h, \alpha, \frac{g}{h\epsilon})$ expander and $\mathbb{S}$ be a linear code of block length h, rate $r > \frac{g-1}{g}$ and minimum distance ϵh. Then, $\mathbb{C}(G, \mathbb{S})$ of block length n has rate atleast $gr - (g - 1)$ and minimum distance atleast αn.

Considering an edge-vertex incidence graph obtained from an h-regular graph, the following result can be obtained [5].

If $\mathbb{S}$ is a linear code of rate r, block length h and minimum distance ϵh and if G is the edge-vertex incidence graph obtained from an h-regular graph with second largest eigenvalue λ, then the expander code $\mathbb{C}(G, \mathbb{S})$ has rate atleast $2r - 1$ and minimum distance atleast $h(\frac{\epsilon - \frac{\lambda}{h}}{1 - \frac{\lambda}{h}})^2$.

5.4 Sipser and Spielman's Decoding Algorithm

For a binary expander code constructed as described above, Sipser and Spielman suggested the following parallel decoding algorithm: Let $\mathbb{S}$ be a $(h, rh, \epsilon h)$ code with its symbols belonging to GF(2). In each decoding round, the ensuing steps are carried out.

- For every constraint C_i, if the variables $(v_{b(i,1)}, \ldots, v_{b(i,h)})$ differ from a valid codeword in $\mathbb{S}$ in atmost $\frac{h\epsilon}{4}$ symbols, then a flip message is sent to all the differing variables.
- Parallely, any variable, which receives atleast one flip message, is flipped.

The following result elaborates on the error-correcting capability of each decoding round of the above algorithm [5].

Let $\mathbb{S}$ be a linear code $(h, rh, \epsilon h)$ and let G be the edge-vertex incidence graph of an h-regular graph with second largest eigenvalue λ. If a word of distance αh from a valid codeword is given as input, one round of the parallel decoding algorithm will output a word of relative distance atmost

$$\alpha(\frac{2}{3} + \frac{16\alpha}{\epsilon^2} + \frac{4\lambda}{\epsilon h}) \tag{31}$$

By limiting the sum of the second two terms in the parentheses in equation 31 to be less than $\frac{1}{3}$ (so that the distance from the codeword decreases after one round), we get the upper bound on the fraction of errors that the algorithm can correct as $\alpha < \frac{\epsilon^2}{48}$. By introducing the Gilbert-Varshamov bound, it can be proved that for all ϵ such that $1 - 2H(\epsilon) > 0$ ($H(.)$ being the binary entropy function), there exists a family of expander codes of rate $1-2H(\epsilon)$ and minimum relative distance arbitrarily close to ϵ^2 in which any $\alpha < \frac{\epsilon^2}{48}$ fraction of error can be corrected using the above decoding algorithm [5].

5.5 Zemor's Construction and Decoding Algorithm

Zemor's construction is based on a h-regular balanced bipartite graph, $G' = (V, E)$. The set V is divided into two sets A and B, with $|A| = |B| = n$ such that every edge has one endpoint in A and another in B. For any vertex t, the set of edges incident on t is denoted by E_t. As the graph is bipartite, the sets E_t $\forall t \in A$ induce a partition on E. A similar partition can be created using the edge sets of the vertices belonging to B. The expander code, $\mathbb{C}(G, \mathbb{S})$ is created on the edge-vertex incidence graph G of the graph G' with a binary subcode $\mathbb{S}$. The block length of code $\mathbb{C}$ is $N = nh$. As before, the second largest eigenvalue of G' is denoted by λ.

The steps in one decoding round of the algorithm suggested by Zemor are as follows:

- Each constraint t in set A completely decodes the subvector associated with the set of h variables, E_t, and replaces it with the closest codeword in $\mathbb{S}$. This step can be carried out in parallel by all constraints in A as no symbol is shared between two constraints.
- The constraints in set B replace the subvector associated with its edge sets, $E_t, t \in B$, with the closest codeword in $\mathbb{S}$. This again can be carried out in parallel by all constraints.

The following result expresses the reduction in unsatisfied constraints in each step of this algorithm.

Suppose $\mathbb{S}$ is a linear code (h, rh, d) and $d \geq 3\lambda$. Let P be a subset of A such that

$$|P| \leq \beta n(\frac{d - 2\lambda}{2h}) \tag{32}$$

where $\beta < 1$. Let Q be a subset of B and suppose that there exists a set $Y \subset E$ such that

1. Every edge of Y has one endpoint in P.
2. Every vertex of Q has atleast $\frac{d}{2}$ edges of Y incident on it.

Then

$$|Q| \leq \frac{1}{2-\beta}|P| \tag{33}$$

Using this result, Zemor [6] proves that suppose $d \geq 3\lambda$, the total fraction of errors that can be corrected using the above algorithm is $\alpha \leq \beta\frac{d}{2h}(\frac{d-2\lambda}{2h})$ for $\beta < 1$.

5.6 MATLAB Implementation: Code Construction

The construction of an expander code $\mathbb{C}(G, \mathbb{S})$ consists of two parts: selecting an expander graph G and selecting a suitable subcode $\mathbb{S}$. We derive the expander graph using projective geometry. A projective space $\mathbb{P}(d,\mathrm{GF}(q))$ has the following two properties:

1. The number of subspaces of dimension m is equal to the number of subspaces of dimension $d - m - 1$.
2. The number of m-dimensional subspaces incident on each $d - m - 1$-dimemsional subspace is equal to the number of $d - m - 1$-dimensional subspaces incident on each m-dimensional subspace.

We use these two properties of projective subspaces to create regular bipartite graphs. We associate one vertex of the graph with each m-dimensional subspace and one with each $d-m-1$-dimensional subspace. Two vertices are connected by an edge if the corresponding subspaces are incident on each other. As edges lie only between subspaces of different dimensions, the graph is bipartite with vertices associated with m-dimensional subspaces forming one set and vertices associated with $d-m-1$-dimensional subspaces forming another. Also, the two properties, listed above, ensure that both the vertex sets have the same number of elements and that each vertex has the same degree.

We consider the graph, $G' = (V, E)$ obtained by taking the points and hyperplanes of $\mathbb{P}(5,\mathrm{GF}(2))$. This projective space is generated from $\mathrm{GF}(2^6)$. The motivation behind selecting this projective space was to create a code similar to the RS coding scheme used in CDROM, which currently implements a (32,28) Cross-Interleaved Reed Solomon code. Also, as burst errors are more important in the case of CD decoding, our focus is mainly on improving burst error performance. In this projective space, the number of points (= number of hyperplanes) is $\phi(5,0,2) = 63$. Each point is incident on $\phi(4,3,2) = 31$ hyperplanes and each hyperplane has $\phi(4,0,2) = 31$ points. Therefore, we have $|V| = 126$ and $|E| = 1953$. This implies that the block length of code

$\mathbb{C}$ is 1953 and the number of constraints in the code is 126. The calculation of elements of GF(2^6) and the generation of incidence relations between points and hyperplanes in $\mathbb{P}$(5,GF(2)) has been elucidated in section 3 of the appendix.

As the graph G' is 31-regular, the block length of the subcode should be 31. We have chosen the 31 symbol Reed Solomon code as the subcode, each symbol consisting of five bits. We use the built-in RS decoding and encoding functions from MATLAB. For testing the error correcting capability of the code, we introduce errors into the zero vector and check its convergence to the zero vector. As the code is linear, the performance obtained in these tests will be valid for the entire code.

5.7 MATLAB Implementation: Results

We have varied the minimum distance and rate of the subcode and the variation in the parameters of $\mathbb{C}$ have been tabulated in table 7.

Our MATLAB implementation differs from Zemor's construction in two aspects:

1. We have used a non-binary RS code instead of a binary code.
2. Zemor suggests that the minimum distance of the subcode should be atleast 3λ. The second largest eigenvalue (λ) of the generated graph G' is 4, which translates to a high minimum distance of 12 for the subcode and at such high distances, the rate of the code will suffer. Therefore, we tried using subcodes with lesser distances and found them to work in this scheme.

The bounds stated in table 7 are worst-case bounds for the given d_0 and cause decoding failure under specific error distributions, which are unlikely to occur. Simulations have shown that many more errors can be corrected if they are randomly distributed. Also, for $d_0 = 11$, we see that the number of errors corrected

Minimum distance of subcode (d_0)	Subcode rate	Lower bound on rate of $\mathbb{C}$	Error-correcting capability	Zemor's bound
3	0.94	0.87	3	–
5	0.87	0.74	8	–
7	0.81	0.61	15	–
9	0.74	0.48	24	–
11	0.68	0.35	35	–
13	0.61	0.23	48	32
15	0.55	0.1	63	52

Table 7: Change in parameters of $\mathbb{C}$ with variation in minimum distance of subcode

by the code is more than Zemor's bound for $d_0 = 13$, which indicates a better error correcting capability at a higher code rate.

These bounds can also be derived geometrically. We will calculate the bounds for $d_0 = 5$ and $d_0 = 11$ by logically looking at the worst-case scenario possible. It follows similarly for the other cases. Let us first consider the case of $d_0 = 5$, i.e., a vertex can correct atmost 2 errors. Consider 3 vertices corresponding to 3 points lying on a line. If each of these constraints gets 3 errors, they will fail to decode the errors and in the worst-case, corrupt the rest of the variables associated with these constraints. The PG line corresponding to these points lies on $\phi(5-1-1, 4-1-1, 2) = 15$ hyperplanes. Hence, the errors introduced by these 3 vertices may translate to 3 errors each on atleast 3 vertices on the other side. These errors will continue to propagate from one side to the other, thus causing the decoding to fail. Hence, we have a case in which introduction of 9 errors causes the decoding to fail. Also, if we introduce 8 or less errors, atmost 2 constraints can introduce more errors, which will result in a maximum of 2 errors on the other side. These errors will get corrected as $d_0 = 5$ and the codeword will get decoded in one iteration itself.

In the case of $d_0 = 11$, we consider 6 vertices on one side that get 6 errors each. In such a case, if all the 6 points(corressponding to these vertices) lie on a plane, there will be $\phi(5-2-1, 4-2-1, 2) = 7$ hyperplanes which will contain each of these points. Each of the corresponding 7 vertices in the graph might receive 6 errors each causing them to introduce more errors into the overall code. With 35 or less error symbols, atmost 5 vertices will have 6 errors which will result in atmost 5 errors for any constraint on the other side after the first step. These will get corrected in the first iteration.

The burst error performance of the code depends on the sequence in which the symbols are transmitted. For example, if the code transmission is done subcode by subcode, in the case of $d_0 = 3$, the code can correct upto 3 burst errors. However, this performance can be improved by interleaving the subcode symbols on transmission. In the case of $d_0 = 3$, on interleaving, the code can correct upto 64 errors.

5.8 Conclusions

From the simulations, we have seen that projective geometry based expander codes perform better than the bounds suggested by Zemor by correcting more errors with a faster code rate. This could be due to the high regularity of PG-based graphs and it can be exploited to create good PG-based expander codes. Further research should look at evaluating these codes by comparing them with the $(32, 28)$ Cross Interleaved Reed Solomon code to see if they can be applied to CD decoding. Another important problem is the design of an encoder for these codes, which is complicated by the greater dependency between constraints due to the good expansion properties of the graph.

Bibliography

[1] B. S. Adiga. Matlab programs. 2008.

[2] Narendra Karmarkar. A new parallel architecture for sparse matrix computation based on finite projective geometries. In Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 358–369. ACM, 1991.

[3] B. Parhami and M. Rakov. Perfect difference networks and related interconnection structures for parallel and distributed systems. IEEE Transactions on Parallel and Distributed Systems, 16(8):714–724, 2005.

[4] B. Parhami and M. Rakov. Performance, algorithmic, and robustness attributes of perfect difference networks. IEEE Transactions on Parallel and Distributed Systems, 16(8):725–736, 2005.

[5] Michael Sipser and Daniel A. Spielman. Expander codes. IEEE Transactions on Information Theory, 42:1710–1722, 1996.

[6] G. Zemor. On expander codes. Information Theory, IEEE Transactions on, 47(2):835–837, Feb 2001.

A Appendix

A.1 Generation of GF(2^5) from GF(2)

We start with x, a root of the equation $x^5 + x^2 + 1 = 0$. By taking successive powers of x and substituting x^5 with $x^2 + 1$ (equivalent to taking the remainder of the power of x with respect to the polynomial as in GF(2), $-1 = 1$), we generate the rest of the elements in the field. These elements are shown in table 8

Power of x	Element of GF(2^5)
x^0	1
x^1	x^1
x^2	x^2
x^3	x^3
x^4	x^4
x^5	$x^2 + 1$
x^6	$x^3 + x$
x^7	$x^4 + x^2$
x^8	$x^3 + x^2 + 1$
x^9	$x^4 + x^3 + x$
x^{10}	$x^4 + 1$
x^{11}	$x^2 + x + 1$
x^{12}	$x^3 + x^2 + x$
x^{13}	$x^4 + x^3 + x^2$
x^{14}	$x^4 + x^3 + x^2 + 1$
x^{15}	$x^4 + x^3 + x^2 + x + 1$

Power of x	Element of GF(2^5)
x^{16}	$x^4 + x^3 + x + 1$
x^{17}	$x^4 + x + 1$
x^{18}	$x + 1$
x^{19}	$x^2 + x$
x^{20}	$x^3 + x^2$
x^{21}	$x^4 + x^3$
x^{22}	$x^4 + x^2 + 1$
x^{23}	$x^3 + x^2 + x + 1$
x^{24}	$x^4 + x^3 + x^2 + x$
x^{25}	$x^4 + x^3 + 1$
x^{26}	$x^4 + x^2 + x + 1$
x^{27}	$x^3 + x + 1$
x^{28}	$x^4 + x^2 + x$
x^{29}	$x^3 + 1$
x^{30}	$x^4 + x$
x^{31}	1

Table 8: Elements of GF(2^5)

A.2 Projective Geometry Structure

The points in the geometry are enlisted in table 9.

Index	1-D subspace
0	$\{0, 1\}$
1	$\{0, x^1\}$
2	$\{0, x^2\}$
3	$\{0, x^3\}$
4	$\{0, x^4\}$
5	$\{0, x^2+1\}$
6	$\{0, x^3+x\}$
7	$\{0, x^4+x^2\}$
8	$\{0, x^3+x^2+1\}$
9	$\{0, x^4+x^3+x\}$
10	$\{0, x^4+1\}$
11	$\{0, x^2+x+1\}$
12	$\{0, x^3+x^2+x\}$
13	$\{0, x^4+x^3+x^2\}$
14	$\{0, x^4+x^3+x^2+1\}$
15	$\{0, x^4+x^3+x^2+x+1\}$

Index	1-D subspace
16	$\{0, x^4+x^3+x+1\}$
17	$\{0, x^4+x+1\}$
18	$\{0, x+1\}$
19	$\{0, x^2+x\}$
20	$\{0, x^3+x^2\}$
21	$\{0, x^4+x^3\}$
22	$\{0, x^4+x^2+1\}$
23	$\{0, x^3+x^2+x+1\}$
24	$\{0, x^4+x^3+x^2+x\}$
25	$\{0, x^4+x^3+1\}$
26	$\{0, x^4+x^2+x+1\}$
27	$\{0, x^3+x+1\}$
28	$\{0, x^4+x^2+x\}$
29	$\{0, x^3+1\}$
30	$\{0, x^4+x\}$

Table 9: Points of $\mathbb{P}$(4,GF(2))

The lines are 2-dimensional subspaces of GF(2^5). The first line is generated by combining the 1-dimensional subspaces corresponding to points 0 and 1. From these two points, we get the line $(0, 1, 18)$. By applying shift and Frobenius automorphisms on this line, we can get the other lines in the geometry. The planes are 3-dimensional subspaces of GF(2^5). The first plane is generated by finding the linear combinations of 3 non-collinear points (0, 1 and 2). The other planes are obtained using automorphisms. The following table 10 shows the planes and also the perfect matching patterns $S_1, \ldots, S_7$. The entire list of lines can be obtained from any map S_i.

Index	Plane	Mapping under						
		S_1	S_2	S_3	S_4	S_5	S_6	S_7
1	(0,1,2,5,11,18,19)	(0,1,18)	(1,2,19)	(0,2,5)	(2,18,11)	(1,5,11)	(11,19,0)	(18,19,5)
2	(1,2,3,6,12,19,20)	(1,2,19)	(2,3,20)	(1,3,6)	(3,19,12)	(2,6,12)	(12,20,1)	(19,20,6)
3	(2,3,4,7,13,20,21)	(2,3,20)	(3,4,21)	(2,4,7)	(4,20,13)	(3,7,13)	(13,21,2)	(20,21,7)
4	(3,4,5,8,14,21,22)	(3,4,21)	(4,5,22)	(3,5,8)	(5,21,14)	(4,8,14)	(14,22,3)	(21,22,8)
5	(4,5,6,9,15,22,23)	(4,5,22)	(5,6,23)	(4,6,9)	(6,22,15)	(5,9,15)	(15,23,4)	(22,23,9)
6	(5,6,7,10,16,23,24)	(5,6,23)	(6,7,24)	(5,7,10)	(7,23,16)	(6,10,16)	(16,24,5)	(23,24,10)
7	(6,7,8,11,17,24,25)	(6,7,24)	(7,8,25)	(6,8,11)	(8,24,17)	(7,11,17)	(17,25,6)	(24,25,11)
8	(7,8,9,12,18,25,26)	(7,8,25)	(8,9,26)	(7,9,12)	(9,25,18)	(8,12,18)	(18,26,7)	(25,26,12)
9	(8,9,10,13,19,26,27)	(8,9,26)	(9,10,27)	(8,10,13)	(10,26,19)	(9,13,19)	(19,27,8)	(26,27,13)
10	(9,10,11,14,20,27,28)	(9,10,27)	(10,11,28)	(9,11,14)	(11,27,20)	(10,14,20)	(20,28,9)	(27,28,14)
11	(10,11,12,15,21,28,29)	(10,11,28)	(11,12,29)	(10,12,15)	(12,28,21)	(11,15,21)	(21,29,10)	(28,29,15)
12	(11,12,13,16,22,29,30)	(11,12,29)	(12,13,30)	(11,13,16)	(13,29,22)	(12,16,22)	(22,30,11)	(29,30,16)
13	(12,13,14,17,23,30,0)	(12,13,30)	(13,14,0)	(12,14,17)	(14,30,23)	(13,17,23)	(23,0,12)	(30,0,17)
14	(13,14,15,18,24,0,1)	(13,14,0)	(14,15,1)	(13,15,18)	(15,0,24)	(14,18,24)	(24,1,13)	(0,1,18)
15	(14,15,16,19,25,1,2)	(14,15,1)	(15,16,2)	(14,16,19)	(16,1,25)	(15,19,25)	(25,2,14)	(1,2,19)
16	(15,16,17,20,26,2,3)	(15,16,2)	(16,17,3)	(15,17,20)	(17,2,26)	(16,20,26)	(26,3,15)	(2,3,20)
17	(16,17,18,21,27,3,4)	(16,17,3)	(17,18,4)	(16,18,21)	(18,3,27)	(17,21,27)	(27,4,16)	(3,4,21)
18	(17,18,19,22,28,4,5)	(17,18,4)	(18,19,5)	(17,19,22)	(19,4,28)	(18,22,28)	(28,5,17)	(4,5,22)
19	(18,19,20,23,29,5,6)	(18,19,5)	(19,20,6)	(18,20,23)	(20,5,29)	(19,23,29)	(29,6,18)	(5,6,23)
20	(19,20,21,24,30,6,7)	(19,20,6)	(20,21,7)	(19,21,24)	(21,6,30)	(20,24,30)	(30,7,19)	(6,7,24)
21	(20,21,22,25,0,7,8)	(20,21,7)	(21,22,8)	(20,22,25)	(22,7,0)	(21,25,0)	(0,8,20)	(7,8,25)
22	(21,22,23,26,1,8,9)	(21,22,8)	(22,23,9)	(21,23,26)	(23,8,1)	(22,26,1)	(1,9,21)	(8,9,26)
23	(22,23,24,27,2,9,10)	(22,23,9)	(23,24,10)	(22,24,27)	(24,9,2)	(23,27,2)	(2,10,22)	(9,10,27)
24	(23,24,25,28,3,10,11)	(23,24,10)	(24,25,11)	(23,25,28)	(25,10,3)	(24,28,3)	(3,11,23)	(10,11,28)
25	(24,25,26,29,4,11,12)	(24,25,11)	(25,26,12)	(24,26,29)	(26,11,4)	(25,29,4)	(4,12,24)	(11,12,29)
26	(25,26,27,30,5,12,13)	(25,26,12)	(26,27,13)	(25,27,30)	(27,12,5)	(26,30,5)	(5,13,25)	(12,13,30)
27	(26,27,28,0,6,13,14)	(26,27,13)	(27,28,14)	(26,28,0)	(28,13,6)	(27,0,6)	(6,14,26)	(13,14,0)
28	(27,28,29,1,7,14,15)	(27,28,14)	(28,29,15)	(27,29,1)	(29,14,7)	(28,1,7)	(7,15,27)	(14,15,1)
29	(28,29,30,2,8,15,16)	(28,29,15)	(29,30,16)	(28,30,2)	(30,15,8)	(29,2,8)	(8,16,28)	(15,16,2)
30	(29,30,0,3,9,16,17)	(29,30,16)	(30,0,17)	(29,0,3)	(0,16,9)	(30,3,9)	(9,17,29)	(16,17,3)
31	(30,0,1,4,10,17,18)	(30,0,17)	(0,1,18)	(30,1,4)	(1,17,10)	(0,4,10)	(10,18,30)	(17,18,4)

Index	Plane	Mapping under						
		S_1	S_2	S_3	S_4	S_5	S_6	S_7
32	(0,2,4,10,22,5,7)	(0,2,5)	(2,4,7)	(0,4,10)	(4,5,22)	(2,10,22)	(22,7,0)	(5,7,10)
33	(1,3,5,11,23,6,8)	(1,3,6)	(3,5,8)	(1,5,11)	(5,6,23)	(3,11,23)	(23,8,1)	(6,8,11)
34	(2,4,6,12,24,7,9)	(2,4,7)	(4,6,9)	(2,6,12)	(6,7,24)	(4,12,24)	(24,9,2)	(7,9,12)
35	(3,5,7,13,25,8,10)	(3,5,8)	(5,7,10)	(3,7,13)	(7,8,25)	(5,13,25)	(25,10,3)	(8,10,13)
36	(4,6,8,14,26,9,11)	(4,6,9)	(6,8,11)	(4,8,14)	(8,9,26)	(6,14,26)	(26,11,4)	(9,11,14)
37	(5,7,9,15,27,10,12)	(5,7,10)	(7,9,12)	(5,9,15)	(9,10,27)	(7,15,27)	(27,12,5)	(10,12,15)
38	(6,8,10,16,28,11,13)	(6,8,11)	(8,10,13)	(6,10,16)	(10,11,28)	(8,16,28)	(28,13,6)	(11,13,16)
39	(7,9,11,17,29,12,14)	(7,9,12)	(9,11,14)	(7,11,17)	(11,12,29)	(9,17,29)	(29,14,7)	(12,14,17)
40	(8,10,12,18,30,13,15)	(8,10,13)	(10,12,15)	(8,12,18)	(12,13,30)	(10,18,30)	(30,15,8)	(13,15,18)
41	(9,11,13,19,0,14,16)	(9,11,14)	(11,13,16)	(9,13,19)	(13,14,0)	(11,19,0)	(0,16,9)	(14,16,19)
42	(10,12,14,20,1,15,17)	(10,12,15)	(12,14,17)	(10,14,20)	(14,15,1)	(12,20,1)	(1,17,10)	(15,17,20)
43	(11,13,15,21,2,16,18)	(11,13,16)	(13,15,18)	(11,15,21)	(15,16,2)	(13,21,2)	(2,18,11)	(16,18,21)
44	(12,14,16,22,3,17,19)	(12,14,17)	(14,16,19)	(12,16,22)	(16,17,3)	(14,22,3)	(3,19,12)	(17,19,22)
45	(13,15,17,23,4,18,20)	(13,15,18)	(15,17,20)	(13,17,23)	(17,18,4)	(15,23,4)	(4,20,13)	(18,20,23)
46	(14,16,18,24,5,19,21)	(14,16,19)	(16,18,21)	(14,18,24)	(18,19,5)	(16,24,5)	(5,21,14)	(19,21,24)
47	(15,17,19,25,6,20,22)	(15,17,20)	(17,19,22)	(15,19,25)	(19,20,6)	(17,25,6)	(6,22,15)	(20,22,25)
48	(16,18,20,26,7,21,23)	(16,18,21)	(18,20,23)	(16,20,26)	(20,21,7)	(18,26,7)	(7,23,16)	(21,23,26)
49	(17,19,21,27,8,22,24)	(17,19,22)	(19,21,24)	(17,21,27)	(21,22,8)	(19,27,8)	(8,24,17)	(22,24,27)
50	(18,20,22,28,9,23,25)	(18,20,23)	(20,22,25)	(18,22,28)	(22,23,9)	(20,28,9)	(9,25,18)	(23,25,28)
51	(19,21,23,29,10,24,26)	(19,21,24)	(21,23,26)	(19,23,29)	(23,24,10)	(21,29,10)	(10,26,19)	(24,26,29)
52	(20,22,24,30,11,25,27)	(20,22,25)	(22,24,27)	(20,24,30)	(24,25,11)	(22,30,11)	(11,27,20)	(25,27,30)
53	(21,23,25,0,12,26,28)	(21,23,26)	(23,25,28)	(21,25,0)	(25,26,12)	(23,0,12)	(12,28,21)	(26,28,0)
54	(22,24,26,1,13,27,29)	(22,24,27)	(24,26,29)	(22,26,1)	(26,27,13)	(24,1,13)	(13,29,22)	(27,29,1)
55	(23,25,27,2,14,28,30)	(23,25,28)	(25,27,30)	(23,27,2)	(27,28,14)	(25,2,14)	(14,30,23)	(28,30,2)
56	(24,26,28,3,15,29,0)	(24,26,29)	(26,28,0)	(24,28,3)	(28,29,15)	(26,3,15)	(15,0,24)	(29,0,3)
57	(25,27,29,4,16,30,1)	(25,27,30)	(27,29,1)	(25,29,4)	(29,30,16)	(27,4,16)	(16,1,25)	(30,1,4)
58	(26,28,30,5,17,0,2)	(26,28,0)	(28,30,2)	(26,30,5)	(30,0,17)	(28,5,17)	(17,2,26)	(0,2,5)
59	(27,29,0,6,18,1,3)	(27,29,1)	(29,0,3)	(27,0,6)	(0,1,18)	(29,6,18)	(18,3,27)	(1,3,6)
60	(28,30,1,7,19,2,4)	(28,30,2)	(30,1,4)	(28,1,7)	(1,2,19)	(30,7,19)	(19,4,28)	(2,4,7)
61	(29,0,2,8,20,3,5)	(29,0,3)	(0,2,5)	(29,2,8)	(2,3,20)	(0,8,20)	(20,5,29)	(3,5,8)
62	(30,1,3,9,21,4,6)	(30,1,4)	(1,3,6)	(30,3,9)	(3,4,21)	(1,9,21)	(21,6,30)	(4,6,9)

Index	Plane	Mapping under						
		S_1	S_2	S_3	S_4	S_5	S_6	S_7
63	(0,4,8,20,13,10,14)	(0,4,10)	(4,8,14)	(0,8,20)	(8,10,13)	(4,20,13)	(13,14,0)	(10,14,20)
64	(1,5,9,21,14,11,15)	(1,5,11)	(5,9,15)	(1,9,21)	(9,11,14)	(5,21,14)	(14,15,1)	(11,15,21)
65	(2,6,10,22,15,12,16)	(2,6,12)	(6,10,16)	(2,10,22)	(10,12,15)	(6,22,15)	(15,16,2)	(12,16,22)
66	(3,7,11,23,16,13,17)	(3,7,13)	(7,11,17)	(3,11,23)	(11,13,16)	(7,23,16)	(16,17,3)	(13,17,23)
67	(4,8,12,24,17,14,18)	(4,8,14)	(8,12,18)	(4,12,24)	(12,14,17)	(8,24,17)	(17,18,4)	(14,18,24)
68	(5,9,13,25,18,15,19)	(5,9,15)	(9,13,19)	(5,13,25)	(13,15,18)	(9,25,18)	(18,19,5)	(15,19,25)
69	(6,10,14,26,19,16,20)	(6,10,16)	(10,14,20)	(6,14,26)	(14,16,19)	(10,26,19)	(19,20,6)	(16,20,26)
70	(7,11,15,27,20,17,21)	(7,11,17)	(11,15,21)	(7,15,27)	(15,17,20)	(11,27,20)	(20,21,7)	(17,21,27)
71	(8,12,16,28,21,18,22)	(8,12,18)	(12,16,22)	(8,16,28)	(16,18,21)	(12,28,21)	(21,22,8)	(18,22,28)
72	(9,13,17,29,22,19,23)	(9,13,19)	(13,17,23)	(9,17,29)	(17,19,22)	(13,29,22)	(22,23,9)	(19,23,29)
73	(10,14,18,30,23,20,24)	(10,14,20)	(14,18,24)	(10,18,30)	(18,20,23)	(14,30,23)	(23,24,10)	(20,24,30)
74	(11,15,19,0,24,21,25)	(11,15,21)	(15,19,25)	(11,19,0)	(19,21,24)	(15,0,24)	(24,25,11)	(21,25,0)
75	(12,16,20,1,25,22,26)	(12,16,22)	(16,20,26)	(12,20,1)	(20,22,25)	(16,1,25)	(25,26,12)	(22,26,1)
76	(13,17,21,2,26,23,27)	(13,17,23)	(17,21,27)	(13,21,2)	(21,23,26)	(17,2,26)	(26,27,13)	(23,27,2)
77	(14,18,22,3,27,24,28)	(14,18,24)	(18,22,28)	(14,22,3)	(22,24,27)	(18,3,27)	(27,28,14)	(24,28,3)
78	(15,19,23,4,28,25,29)	(15,19,25)	(19,23,29)	(15,23,4)	(23,25,28)	(19,4,28)	(28,29,15)	(25,29,4)
79	(16,20,24,5,29,26,30)	(16,20,26)	(20,24,30)	(16,24,5)	(24,26,29)	(20,5,29)	(29,30,16)	(26,30,5)
80	(17,21,25,6,30,27,0)	(17,21,27)	(21,25,0)	(17,25,6)	(25,27,30)	(21,6,30)	(30,0,17)	(27,0,6)
81	(18,22,26,7,0,28,1)	(18,22,28)	(22,26,1)	(18,26,7)	(26,28,0)	(22,7,0)	(0,1,18)	(28,1,7)
82	(19,23,27,8,1,29,2)	(19,23,29)	(23,27,2)	(19,27,8)	(27,29,1)	(23,8,1)	(1,2,19)	(29,2,8)
83	(20,24,28,9,2,30,3)	(20,24,30)	(24,28,3)	(20,28,9)	(28,30,2)	(24,9,2)	(2,3,20)	(30,3,9)
84	(21,25,29,10,3,0,4)	(21,25,0)	(25,29,4)	(21,29,10)	(29,0,3)	(25,10,3)	(3,4,21)	(0,4,10)
85	(22,26,30,11,4,1,5)	(22,26,1)	(26,30,5)	(22,30,11)	(30,1,4)	(26,11,4)	(4,5,22)	(1,5,11)
86	(23,27,0,12,5,2,6)	(23,27,2)	(27,0,6)	(23,0,12)	(0,2,5)	(27,12,5)	(5,6,23)	(2,6,12)
87	(24,28,1,13,6,3,7)	(24,28,3)	(28,1,7)	(24,1,13)	(1,3,6)	(28,13,6)	(6,7,24)	(3,7,13)
88	(25,29,2,14,7,4,8)	(25,29,4)	(29,2,8)	(25,2,14)	(2,4,7)	(29,14,7)	(7,8,25)	(4,8,14)
89	(26,30,3,15,8,5,9)	(26,30,5)	(30,3,9)	(26,3,15)	(3,5,8)	(30,15,8)	(8,9,26)	(5,9,15)
90	(27,0,4,16,9,6,10)	(27,0,6)	(0,4,10)	(27,4,16)	(4,6,9)	(0,16,9)	(9,10,27)	(6,10,16)
91	(28,1,5,17,10,7,11)	(28,1,7)	(1,5,11)	(28,5,17)	(5,7,10)	(1,17,10)	(10,11,28)	(7,11,17)
92	(29,2,6,18,11,8,12)	(29,2,8)	(2,6,12)	(29,6,18)	(6,8,11)	(2,18,11)	(11,12,29)	(8,12,18)
93	(30,3,7,19,12,9,13)	(30,3,9)	(3,7,13)	(30,7,19)	(7,9,12)	(3,19,12)	(12,13,30)	(9,13,19)

Index	Plane	Mapping under						
		S_1	S_2	S_3	S_4	S_5	S_6	S_7
94	(0,8,16,9,26,20,28)	(0,8,20)	(8,16,28)	(0,16,9)	(16,20,26)	(8,9,26)	(26,28,0)	(20,28,9)
95	(1,9,17,10,27,21,29)	(1,9,21)	(9,17,29)	(1,17,10)	(17,21,27)	(9,10,27)	(27,29,1)	(21,29,10)
96	(2,10,18,11,28,22,30)	(2,10,22)	(10,18,30)	(2,18,11)	(18,22,28)	(10,11,28)	(28,30,2)	(22,30,11)
97	(3,11,19,12,29,23,0)	(3,11,23)	(11,19,0)	(3,19,12)	(19,23,29)	(11,12,29)	(29,0,3)	(23,0,12)
98	(4,12,20,13,30,24,1)	(4,12,24)	(12,20,1)	(4,20,13)	(20,24,30)	(12,13,30)	(30,1,4)	(24,1,13)
99	(5,13,21,14,0,25,2)	(5,13,25)	(13,21,2)	(5,21,14)	(21,25,0)	(13,14,0)	(0,2,5)	(25,2,14)
100	(6,14,22,15,1,26,3)	(6,14,26)	(14,22,3)	(6,22,15)	(22,26,1)	(14,15,1)	(1,3,6)	(26,3,15)
101	(7,15,23,16,2,27,4)	(7,15,27)	(15,23,4)	(7,23,16)	(23,27,2)	(15,16,2)	(2,4,7)	(27,4,16)
102	(8,16,24,17,3,28,5)	(8,16,28)	(16,24,5)	(8,24,17)	(24,28,3)	(16,17,3)	(3,5,8)	(28,5,17)
103	(9,17,25,18,4,29,6)	(9,17,29)	(17,25,6)	(9,25,18)	(25,29,4)	(17,18,4)	(4,6,9)	(29,6,18)
104	(10,18,26,19,5,30,7)	(10,18,30)	(18,26,7)	(10,26,19)	(26,30,5)	(18,19,5)	(5,7,10)	(30,7,19)
105	(11,19,27,20,6,0,8)	(11,19,0)	(19,27,8)	(11,27,20)	(27,0,6)	(19,20,6)	(6,8,11)	(0,8,20)
106	(12,20,28,21,7,1,9)	(12,20,1)	(20,28,9)	(12,28,21)	(28,1,7)	(20,21,7)	(7,9,12)	(1,9,21)
107	(13,21,29,22,8,2,10)	(13,21,2)	(21,29,10)	(13,29,22)	(29,2,8)	(21,22,8)	(8,10,13)	(2,10,22)
108	(14,22,30,23,9,3,11)	(14,22,3)	(22,30,11)	(14,30,23)	(30,3,9)	(22,23,9)	(9,11,14)	(3,11,23)
109	(15,23,0,24,10,4,12)	(15,23,4)	(23,0,12)	(15,0,24)	(0,4,10)	(23,24,10)	(10,12,15)	(4,12,24)
110	(16,24,1,25,11,5,13)	(16,24,5)	(24,1,13)	(16,1,25)	(1,5,11)	(24,25,11)	(11,13,16)	(5,13,25)
111	(17,25,2,26,12,6,14)	(17,25,6)	(25,2,14)	(17,2,26)	(2,6,12)	(25,26,12)	(12,14,17)	(6,14,26)
112	(18,26,3,27,13,7,15)	(18,26,7)	(26,3,15)	(18,3,27)	(3,7,13)	(26,27,13)	(13,15,18)	(7,15,27)
113	(19,27,4,28,14,8,16)	(19,27,8)	(27,4,16)	(19,4,28)	(4,8,14)	(27,28,14)	(14,16,19)	(8,16,28)
114	(20,28,5,29,15,9,17)	(20,28,9)	(28,5,17)	(20,5,29)	(5,9,15)	(28,29,15)	(15,17,20)	(9,17,29)
115	(21,29,6,30,16,10,18)	(21,29,10)	(29,6,18)	(21,6,30)	(6,10,16)	(29,30,16)	(16,18,21)	(10,18,30)
116	(22,30,7,0,17,11,19)	(22,30,11)	(30,7,19)	(22,7,0)	(7,11,17)	(30,0,17)	(17,19,22)	(11,19,0)
117	(23,0,8,1,18,12,20)	(23,0,12)	(0,8,20)	(23,8,1)	(8,12,18)	(0,1,18)	(18,20,23)	(12,20,1)
118	(24,1,9,2,19,13,21)	(24,1,13)	(1,9,21)	(24,9,2)	(9,13,19)	(1,2,19)	(19,21,24)	(13,21,2)
119	(25,2,10,3,20,14,22)	(25,2,14)	(2,10,22)	(25,10,3)	(10,14,20)	(2,3,20)	(20,22,25)	(14,22,3)
120	(26,3,11,4,21,15,23)	(26,3,15)	(3,11,23)	(26,11,4)	(11,15,21)	(3,4,21)	(21,23,26)	(15,23,4)
121	(27,4,12,5,22,16,24)	(27,4,16)	(4,12,24)	(27,12,5)	(12,16,22)	(4,5,22)	(22,24,27)	(16,24,5)
122	(28,5,13,6,23,17,25)	(28,5,17)	(5,13,25)	(28,13,6)	(13,17,23)	(5,6,23)	(23,25,28)	(17,25,6)
123	(29,6,14,7,24,18,26)	(29,6,18)	(6,14,26)	(29,14,7)	(14,18,24)	(6,7,24)	(24,26,29)	(18,26,7)
124	(30,7,15,8,25,19,27)	(30,7,19)	(7,15,27)	(30,15,8)	(15,19,25)	(7,8,25)	(25,27,30)	(19,27,8)

Index	Plane	Mapping under						
		S_1	S_2	S_3	S_4	S_5	S_6	S_7
125	(0,16,1,18,21,9,25)	(0,16,9)	(16,1,25)	(0,1,18)	(1,9,21)	(16,18,21)	(21,25,0)	(9,25,18)
126	(1,17,2,19,22,10,26)	(1,17,10)	(17,2,26)	(1,2,19)	(2,10,22)	(17,19,22)	(22,26,1)	(10,26,19)
127	(2,18,3,20,23,11,27)	(2,18,11)	(18,3,27)	(2,3,20)	(3,11,23)	(18,20,23)	(23,27,2)	(11,27,20)
128	(3,19,4,21,24,12,28)	(3,19,12)	(19,4,28)	(3,4,21)	(4,12,24)	(19,21,24)	(24,28,3)	(12,28,21)
129	(4,20,5,22,25,13,29)	(4,20,13)	(20,5,29)	(4,5,22)	(5,13,25)	(20,22,25)	(25,29,4)	(13,29,22)
130	(5,21,6,23,26,14,30)	(5,21,14)	(21,6,30)	(5,6,23)	(6,14,26)	(21,23,26)	(26,30,5)	(14,30,23)
131	(6,22,7,24,27,15,0)	(6,22,15)	(22,7,0)	(6,7,24)	(7,15,27)	(22,24,27)	(27,0,6)	(15,0,24)
132	(7,23,8,25,28,16,1)	(7,23,16)	(23,8,1)	(7,8,25)	(8,16,28)	(23,25,28)	(28,1,7)	(16,1,25)
133	(8,24,9,26,29,17,2)	(8,24,17)	(24,9,2)	(8,9,26)	(9,17,29)	(24,26,29)	(29,2,8)	(17,2,26)
134	(9,25,10,27,30,18,3)	(9,25,18)	(25,10,3)	(9,10,27)	(10,18,30)	(25,27,30)	(30,3,9)	(18,3,27)
135	(10,26,11,28,0,19,4)	(10,26,19)	(26,11,4)	(10,11,28)	(11,19,0)	(26,28,0)	(0,4,10)	(19,4,28)
136	(11,27,12,29,1,20,5)	(11,27,20)	(27,12,5)	(11,12,29)	(12,20,1)	(27,29,1)	(1,5,11)	(20,5,29)
137	(12,28,13,30,2,21,6)	(12,28,21)	(28,13,6)	(12,13,30)	(13,21,2)	(28,30,2)	(2,6,12)	(21,6,30)
138	(13,29,14,0,3,22,7)	(13,29,22)	(29,14,7)	(13,14,0)	(14,22,3)	(29,0,3)	(3,7,13)	(22,7,0)
139	(14,30,15,1,4,23,8)	(14,30,23)	(30,15,8)	(14,15,1)	(15,23,4)	(30,1,4)	(4,8,14)	(23,8,1)
140	(15,0,16,2,5,24,9)	(15,0,24)	(0,16,9)	(15,16,2)	(16,24,5)	(0,2,5)	(5,9,15)	(24,9,2)
141	(16,1,17,3,6,25,10)	(16,1,25)	(1,17,10)	(16,17,3)	(17,25,6)	(1,3,6)	(6,10,16)	(25,10,3)
142	(17,2,18,4,7,26,11)	(17,2,26)	(2,18,11)	(17,18,4)	(18,26,7)	(2,4,7)	(7,11,17)	(26,11,4)
143	(18,3,19,5,8,27,12)	(18,3,27)	(3,19,12)	(18,19,5)	(19,27,8)	(3,5,8)	(8,12,18)	(27,12,5)
144	(19,4,20,6,9,28,13)	(19,4,28)	(4,20,13)	(19,20,6)	(20,28,9)	(4,6,9)	(9,13,19)	(28,13,6)
145	(20,5,21,7,10,29,14)	(20,5,29)	(5,21,14)	(20,21,7)	(21,29,10)	(5,7,10)	(10,14,20)	(29,14,7)
146	(21,6,22,8,11,30,15)	(21,6,30)	(6,22,15)	(21,22,8)	(22,30,11)	(6,8,11)	(11,15,21)	(30,15,8)
147	(22,7,23,9,12,0,16)	(22,7,0)	(7,23,16)	(22,23,9)	(23,0,12)	(7,9,12)	(12,16,22)	(0,16,9)
148	(23,8,24,10,13,1,17)	(23,8,1)	(8,24,17)	(23,24,10)	(24,1,13)	(8,10,13)	(13,17,23)	(1,17,10)
149	(24,9,25,11,14,2,18)	(24,9,2)	(9,25,18)	(24,25,11)	(25,2,14)	(9,11,14)	(14,18,24)	(2,18,11)
150	(25,10,26,12,15,3,19)	(25,10,3)	(10,26,19)	(25,26,12)	(26,3,15)	(10,12,15)	(15,19,25)	(3,19,12)
151	(26,11,27,13,16,4,20)	(26,11,4)	(11,27,20)	(26,27,13)	(27,4,16)	(11,13,16)	(16,20,26)	(4,20,13)
152	(27,12,28,14,17,5,21)	(27,12,5)	(12,28,21)	(27,28,14)	(28,5,17)	(12,14,17)	(17,21,27)	(5,21,14)
153	(28,13,29,15,18,6,22)	(28,13,6)	(13,29,22)	(28,29,15)	(29,6,18)	(13,15,18)	(18,22,28)	(6,22,15)
154	(29,14,30,16,19,7,23)	(29,14,7)	(14,30,23)	(29,30,16)	(30,7,19)	(14,16,19)	(19,23,29)	(7,23,16)
155	(30,15,0,17,20,8,24)	(30,15,8)	(15,0,24)	(30,0,17)	(0,8,20)	(15,17,20)	(20,24,30)	(8,24,17)

Table 10: Planes of $\mathbb{P}(4,\mathrm{GF}(2))$ and the mappings $S_1, \ldots, S_7$

A.3 Graph for expander codes

We have chosen the 31-regular bipartite graph between the point and hyperplane subspaces of the projective space $\mathbb{P}(5,\text{GF}(2))$. The primitive polynomial used to generate $\text{GF}(2^6)$ is $x^6 + x + 1$. The points of this projective space are given in table 11. The points lying on a hyperplane can be obtained by taking a 5-dimensional vector subspace of the 6-dimensional vector space $\text{GF}(2^6)$ and then taking the points from the table that correspond to vectors belonging to that subspace. One such hyperplane is $(0, 1, 2, 3, 4, 6, 7, 8, 9, 12, 13, 14, 16, 18, 19, 24, 26, 27, 28, 32, 33, 35, 36, 38, 41, 45, 48, 49, 52, 54, 56)$. Other hyperplanes can be obtained by applying shift automorphism to this hyperplane.

Index	1-D subspace
0	$\{0,1\}$
1	$\{0,\ x^1\}$
2	$\{0,\ x^2\}$
3	$\{0,\ x^3\}$
4	$\{0,\ x^4\}$
5	$\{0,\ x^5\}$
6	$\{0,\ x^1+1\}$
7	$\{0,\ x^2+x\}$
8	$\{0,\ x^3+x^2\}$
9	$\{0,\ x^4+x^3\}$
10	$\{0,\ x^5+x^4\}$
11	$\{0,\ x^5+x^1+1\}$
12	$\{0,\ x^2+1\}$
13	$\{0,\ x^3+x\}$
14	$\{0,\ x^4+x^2\}$
15	$\{0,\ x^5+x^3\}$
16	$\{0,\ x^4+x^1+1\}$
17	$\{0,\ x^5+x^3+x^2\}$
18	$\{0,\ x^3+x^2+x+1\}$
19	$\{0,\ x^4+x^3+x^2+x\}$
20	$\{0,\ x^5+x^4+x^3+x^2\}$
21	$\{0,\ x^5+x^4+x^3+x+1\}$
22	$\{0,\ x^5+x^4+x^2+1\}$
23	$\{0,\ x^5+x^3+1\}$
24	$\{0,\ x^4+1\}$
25	$\{0,\ x^5+x^1\}$
26	$\{0,\ x^2+x^1+1\}$
27	$\{0,\ x^3+x^2+x^1\}$
28	$\{0,\ x^4+x^3+x^2\}$
29	$\{0,\ x^5+x^4+x^3\}$
30	$\{0,\ x^5+x^4+x+1\}$
31	$\{0,\ x^5+x^2+1\}$

Index	1-D subspace
32	$\{0,\ x^3+1\}$
33	$\{0,\ x^4+x\}$
34	$\{0,\ x^5+x^2\}$
35	$\{0,\ x^3+x+1\}$
36	$\{0,\ x^4+x^2+x\}$
37	$\{0,\ x^5+x^3+x^2\}$
38	$\{0,\ x^4+x^3+x+1\}$
39	$\{0,\ x^5+x^4+x^2+x\}$
40	$\{0,\ x^5+x^3+x^2+x+1\}$
41	$\{0,\ x^4+x^3+x^2+1\}$
42	$\{0,\ x^5+x^4+x^3+x\}$
43	$\{0,\ x^5+x^4+x^2+x+1\}$
44	$\{0,\ x^5+x^3+x^2+1\}$
45	$\{0,\ x^4+x^3+1\}$
46	$\{0,\ x^5+x^4+x\}$
47	$\{0,\ x^5+x^2+x+1\}$
48	$\{0,\ x^3+x^2+1\}$
49	$\{0,\ x^4+x^3+x\}$
50	$\{0,\ x^5+x^4+x^2\}$
51	$\{0,\ x^5+x^3+x+1\}$
52	$\{0,\ x^4+x^2+x\}$
53	$\{0,\ x^5+x^3+x\}$
54	$\{0,\ x^4+x^2+x+1\}$
55	$\{0,\ x^5+x^3+x^2+x\}$
56	$\{0,\ x^4+x^3+x^2+x+1\}$
57	$\{0,\ x^5+x^4+x^3+x^2+x\}$
58	$\{0,\ x^5+x^4+x^3+x^2+x+1\}$
59	$\{0,\ x^5+x^4+x^3+x^2+1\}$
60	$\{0,\ x^5+x^4+x^3+1\}$
61	$\{0,\ x^5+x^4+1\}$
62	$\{0,\ x^5+1\}$

Table 11: Points of $\mathbb{P}$(5,GF(2))