A New Parallel Architecture for Sparse Matrix Computation Based on Finite Projective Geometries

Narendra Karmarkar

AT&T Bell Laboratories
Murray Hill, NJ 07974

Indian Institute of Technology
Bombay, India

Abstract

Many problems in scientific computation involve sparse matrices. While dense matrix computations can be parallelized relatively easily, sparse matrices with arbitrary or irregular structure pose a real challenge to the design of highly parallel machines.

In this paper we propose a new parallel architecture for sparse matrix computation based on finite projective geometries. Mathematical structure of these geometries play an important role in defining the pattern of interconnection between memories and processors as well as solving several difficult problems arising in parallel systems (such as load balancing, data-routing, memory-access conflicts etc.) in an efficient manner.

1. Introduction

1.1 Application domain

The architecture described in this paper was motivated by certain types of problems arising in scientific computation such as linear programming, solution of partial differential equations, signal processing, simulation of non-linear electronic circuits, non-linear programming etc. A large body of research in parallel computation is directed towards finding the maximum amount of parallelism available in a given task. Fortunately, in the types of applications we are interested in, there is plenty of intrinsic parallelism available at a fine-grain level. The real difficulties arise not in *finding* parallelism, but in *exploiting* it *efficiently.*

1.2 Difficulties in exploiting parallelism

These difficulties arise at roughly three different levels: architectural level, hardware operational-level and software level. Some of these difficulties are briefly described below.

1.2.1 Problem-independent interconnection topology

Each computational problem usually has an associated topology that is most naturally suited to it. In case of some important problem one may be able to justify building a machine whose interconnection pattern is dedicated to solving that particular problem. On the other hand, one can base the architecture on a general interconnection network that can simulate any topology efficiently. There have been a number of interconnection networks proposed and explored, differing in the trade-off they make between generality and efficiency.

1.2.2 Sparse matrices with irregular non-zero structure

While dense matrix computations can be parallelized relatively easily on traditional pipe-lined vector machines (e.g. Cray), as well as on many other parallel architectures, many scientific applications give rise to sparse matrices with arbitrary or irregular pattern of non-zero locations. Such problems pose a real challenge to the design of highly parallel machines. On the other hand, very efficient data structures and programming techniques have been developed for sparse matrix computation on sequential machines. When evaluating the effectiveness of a parallel architecture on a sparse matrix problem, it is important to compare the performance with the best sequential method for solving the same problems rather than comparing some algorithm on the two architectures.

1.2.3 Load balancing

It is necessary to distribute the computational load among processors as evenly as possible to obtain high efficiency.

1.2.4 Routing algorithm

The movement of data through the network interconnecting various hardware elements of the system such as processors and memories needs to be governed by a routing algorithm. here the issues are congestion, delays, ability to find conflict-free paths, amount of buffering needed at intermediate nodes, and speed of the routing algorithm itself.

1.2.5 Memory accesses

A memory access conflict arises when two processors try to access the same memory location at the same time. One needs to either make sure that memory access conflicts do not arise or provide a mechanism for resolving them. Another serious difficulty regarding memory accesses is caused by mismatch of bandwidth, i.e., memory is unable to supply data at the rate the processor needs. This has been a problem even for single processor machines. Typically processors are faster than memories, they can be pipelined. In the conventional fetch-decode-execute mode, it is difficult to pipeline memory accesses.

1.2.6 Difficulties in programming

Some parallel architectures require the user to decompose the task and assign the subproblems to processors and also to program the communication between processors. In case of a problem involving a regular square grid of nodes, it is easy to decompose the problem into isomorphic subproblems. Unfortunately, many real problems involve irregular grids or boundaries, sharp transitions in the functions being computed etc. The task of decomposing such problems can be quite tedious. It is desirable to have the compiler do efficient mapping of user's problem on the underlying hardware. Another issue that comes up is the level of granularity at which the parallelism is to be exploited. If one restricts the parallelism available at a coarse-grain level, it is easier to exploit but as one goes to finer levels, there is more parallelism available. The real challenge is to provide the ability to exploit even fine-grain parallelism.

1.3 An architecture based on geometric subspaces

There have been a number of parallel architectures proposed in the last decade. Typically one first decides a method of interconnecting the hardware elements such as processors and memories. Several algorithms are then found to control the operation of the system such as moving data through the network, assigning and balancing the load on processors, etc. A question naturally arises: is it possible to define a parallel system in which the most elementary instruction you can give to the system as a whole automatically results in coherent, conflict-free operation and uniform, balanced use of all the resources such as processors, memories and wires connecting them, and can one express computation specified by the user in terms of a sequence of such instructions? Thus an instruction for the system should be more than just a collection of instructions for the individual elements of the system. It should have three further properties: first, when individual elements of the system follow such an instruction, there should be no conflicts or inefficient use of the resources. Secondly the collection of such system-instructions should be powerful enough so that computation specified by the user can be expressed in terms of these instructions. Furthermore the instruction set should have a structure that permits this process of mapping user programs onto the underlying architecture to be carried out efficiently. It seems that the mathematical structure of objects known as finite geometries is eminently suited for defining such a parallel system for solving the types of problem in scientific computation described earlier. The work reported here began at first as a mathematical curiosity to explore how the structure and symmetry of finite geometries can be exploited to define the interconnection pattern between memories and processors, assigning load to processors and accessing memories in a conflict-free manner. Since then it has grown in several directions: how to design application algorithms for various problems in scientific computation? How best to map a given computation onto the architecture, how to design the hardware both at system level and VLSI chip level etc. These issues will be addressed in a series of papers that document the study performed by several researchers at AT&T Bell Laboratories and Indian Institute of Technology. This paper, which is the first in the series describes the mathematical concepts underlying the new architecture and two applications important in scientific computation, namely matrix-vector multiplication and gaussian elimination for sparse matrices. The other papers describe the compiler [DHI 89], simulation results [DKR 91] and aspects of hardware design. In addition to showing how the mathematical structure of finite geometries helps in solving many difficult problems in parallel system design, our work is guided by the objective that even the first implementation based on these concepts should result in a practical machine that many scientists and engineers would want to use.

Consequently, the first version of the compiler and hardware are limited to applications that involve execution of a data-flow graph whose symbolic structure remains fixed but which is executed with different numerical values several times during the course of the algorithm. As an example, let us consider the problem of electronic circuit-simulation, a problem which is notoriously difficult for parallelization. A single execution of the simulator typically involves several hundred time steps having the same data-flow graph, each time step which solves non-linear algebraic equations involves several linear-system solutions having the same symbolic structure and each linear-system, if solved by an iterative method like conjugate-gradient, involves several multiplications with a sparse matrix with fixed non-zero structure. Since the non-zero pattern of the matrix depends only on the structure of electronic circuit being simulated, a single execution of the simulator may involve several thousand matrix-vector multiplications having the same data-flow graph.

Typical number of iterations of the same data-flow graph for several other applications are compiled in Table #1.

LP Problems	Problem Number	Repetition Count of Data Flow Graph
Partial Differential Eqns.	1	1,428
	2	12,556
Fractional Hypergraph Covering	1	16,059
	2	6,299
	3	7,592
Control Systems	1	708
	2	1,863
	3	7,254

Table #1. Iterations on Same Data-Flow Graph

2. Description of the Architecture

2.1 Host processors and the attached processor

The machine proposed here is meant to be used as an attached processor to a general purpose machine referred to as host processor. The two processors share a common global memory. The main program runs on the host processor. Computationally intensive subroutines or macros that have fixed symbolic structure but are executed several times with different numerical values are to be carried out on the attached processor. Since the two processors share the same memory, it is not necessary to communicate large amounts of data between the processors. Only certain structural information such as base addresses of arrays need to be communicated from host processor to attached processor before invoking a subroutine to be executed on the attached processor.

2.2 Interconnection scheme based on subspaces of a projective geometry

A finite geometry of dimension d consists of a finite set of points S, and a collection of subsets of S associated with each integer $i \leq d$ which constitute subspaces of dimension i. Thus subsets associated with dimension 1 are called lines, those associated with dimension 2 are called planes, etc. These subsets have intersection properties similar to the properties of the familiar 3-dimensional (infinite) geometry. e.g. Two points determine a unique line, three non-collinear points determine a plane, two lines in a plane intersect in a point etc. In section 3 we will define the class of geometries we are going to use more precisely. The geometric structure is used for defining the interconnection between memories and processors as follows. Given a finite geometry of dimension d, choose a pair of dimensions d_m, $d_p \leq d$. Put the processors in the system in one-to-one correspondence with all subspaces of dimension d_p, put the memory modules in one-to-one correspondence with all subspaces of dimension d_m; and connect a processors and memory module if the corresponding subspaces have a non-trivial intersection.

2.3 Memory system

The memory system of the attached processor is partitioned into n modules denoted by $M_1, M_2, \ldots, M_n$. The type of memory access possible in one machine cycle in this architecture is between the two extremes of random access and sequential access and could be called "structured" access. Only certain combinations of words can be accessed in one cycle. These combinations are designed using certain symmetries present in the projective geometry so that no conflicts can arise in either accessing the memory or sending the accessed data through the interconnection network. The total set of allowed combinations is powerful enough so that a sequence of such structured accesses is as effective as random accesses. On the other hand, such a structured-access memory has a much higher bandwidth than a random-access memory implemented using comparable technology. In section 3, a mathematical characterization of allowed access patterns is given. Furthermore, these individual access patterns can be combined to form certain special sequences again using the symmetries present in the geometry. Each such sequence, called a perfect sequence, defines the operation of the hardware for several consecutive machine cycles at once, and has the effect of utilizing the communication bandwidth of the machine fully. As a result, it is possible to connect the memories and processors in the system so that the number of wires needed grows *linearly* with respect to the number of processors and memories in the system.

2.4 Rule for load-assignment

The assignment of computational load to processors is done at a fine-grain level. Consider a binary operation that takes two operands a and b as inputs and modifies one of them, say a, as output

$$a \leftarrow a \circ b \ .$$

Suppose the operand 'a' belong to the memory module M_i and operand 'b' belongs to the memory module M_j. Then we associate an index-pair (i, j) with this operation. Similarly with a ternary operation, we associate an index-triplet. If we number the processor as $P_1, P_2, \ldots, P_n$ then the processor P_l that is responsible for doing a binary operation with associated index pair (i, j) is given by a certain function that depends on the geometry.

$$l = f(i, j) \ .$$

Thus two operations having the same associated index-pair (or triplet) always get assigned to the same processor. Furthermore, the function used for load-assignment is compatible with the structure of the geometry, i.e. the processor numbered $f(i, j)$ is connected to memory modules i and j.

Since the assignment of operations to processors is determined entirely by the location of the data, the compiler has an indirect control over load-balancing by exploiting the freedom it has of assigning memory locations to intermediate operands in the data-flow graph and moving the operands from one memory module to another by inserting move (i, j) instructions if necessary. (Even random assignment of memory locations tends to distribute the load evenly as the size of the data-flow graph increases.)

2.5 Instructions for the system and its elements

A system-instruction decides what action each of the individual hardware elements are to perform in a single machine cycle (or over a sequence of machine cycles in case of instructions corresponding to perfect sequences of conflict-free patterns). Each hardware element such as an arithmetic processor, a switch or a memory module has its own instruction set. The compiler takes a data-flow graph as input, uses system-level instructions implicitly to produce as output a collection of programs, one for each element of the system, consisting of list of instructions drawn from the instruction set of that element. Each hardware element of the system has local memory for storing the instruction sequence to be used by that element. The initial loading of these instruction sequences for each subroutine to be executed on the attached processor is done by the host. Once loaded, the instruction sequence may typically get executed many times before being overwritten by the instruction sequence for some other subroutine. Depending on the size of local instruction memories, instruction sequences for several subroutines can reside simultaneously in these memories. Since the instruction sequences are accessed sequentially, an hardware implementation of the instruction storage memory can take advantage of sequential access.

2.6 Pipelining of memory accesses

The instruction sequence for each memory module specifies a list of addresses of operands along with a read/write bit. The instruction sequence is stored in the same module. Since the address sequence is known in advance, full pipelining is possible in decoding addresses, accessing bits in memory and in dynamic error detection and correction. Operands being written can also be placed in a shift register whose length depends on the number of stages in the pipeline, so that consistent values of operands can be supplied. Alternatively, the compiler can ensure that any memory location that is modified is not immediately accessed for reading (within as many machine cycles as the length of the pipeline for writing).

In a large-scale implementation each memory module can be connected to a separate ''local'' disk, to provide for secondary storage in addition to the secondary storage of the host processor.

2.7 Arithmetic processors

Arithmetic processors are capable of performing the basic arithmetic and logical operations. The instruction sequence to be followed by a processor (created by a compiler) can also contain move (i, j) operations simply for moving operands from memory module M_i to M_j, as well as "no-ops". In the case of pipe-lined operations taking more than one machine cycle "execution" of the corresponding instruction means initiating the operation. It is up to the compiler to ensure that delay in the availability of the results is taken into account when the data-flow graph is processed. The instruction sequence to be followed by a processor is stored into its own local instruction memory. The instruction sequence does not contain addresses of operands, but only the type of operation to be performed (including no operation). There is no concept of "fetching" an operand. The processors simply operates on whatever data flows on its dedicated links. Each processor has its own local memory. In an application such as multiplication of a sparse matrix by a vector, the matrix elements can be stored in the local memories of processors and the input and output vectors can be stored in the shared, partitioned global memory.

2.8 Compiler

Given a sequence of instructions, many rearrangements of the sequence are possible that produce the same end result. An optimizing compiler for a sequential machine seeks to perform such rearrangements with the goal of reducing the sequential complexity of the program.

In case of a parallel machine, the number of possible rearrangements is even larger because of a greater degree of freedom available in assigning operations to processors, in moving the data through the interconnection network etc. In order to make this freedom more visible and available to the compiler, input to the compiler is expressed in terms of a data-flow graph that represents a partial order to be satisfied by operations based on dependencies. The first version of the compiler [DHI 89] is restricted to applications that involve repeated execution of the same data-flow graph with different numerical values. The compiler does extensive processing of the data-flow graph to re-express the computation in balanced, conflict-free chunks as much as possible. Any remaining inefficiency shows up in the form of "holes" in the perfect patterns which result in "no-ops" in the instruction sequences for processors.

3. Application of Finite Projective Geometries

3.1 Projective spaces over finite fields

In this section we briefly review the concept of a projective space over a finite field and introduce some notation used later.

Consider a finite field $F = GF(s)$ having s elements, where s is a power of a prime number p, $s = p^k$, and k is a positive integer.

A projective space of dimension d over the finite field F, denoted by $\mathbb{P}^d(F)$, consists of one dimensional subspaces of a $(d+1)$ dimensional vector space F^{d+1} over the finite field, F. Elements of this vector space can be represented as $(d+1)$-tuples $(x_1, x_2, x_3, \ldots, x_{d+1})$ where each $x_i \in F$. Clearly, the total number of such elements is $s^{d+1} = p^{k(d+1)}$. Two non-zero elements $\underline{x}$, $\underline{y} \neq 0$ of this vector space are said to be equivalent if there exists a $\lambda \in GF(s)$ such that $\underline{x} = \lambda \underline{y}$. Each equivalence class gives a point in the projective space. Hence the number of points in $\mathbb{P}^d(F)$ is given by

$$P_d = \frac{s^{d+1} - 1}{s - 1} .$$

An m-dimensional projective subspace of $\mathbb{P}^d(F)$ consists of all one dimensional subspaces of an $(m+1)$-dimensional subspace of the vector space. Let $\underline{b_0}, \underline{b_1}, \ldots \underline{b_m}$ be a basis of the latter vector subspace. The elements of the vector subspace are given by

$$\underline{x} = \sum_{i=0}^{m} \alpha_i \underline{b_i} \quad \text{where } \alpha_i \in F .$$

Hence the number of such elements is s^{m+1}, and the number of points in the corresponding projective subspace is

$$p_m = \frac{s^{m+1} - 1}{s - 1} .$$

Let $r = d - m$. Then an $(m+1)$-dimensional vector subspace of F^{d+1} can also be described as the set of all solutions of a system of r independent linear equations $\underline{a_i}^T \underline{x} = 0$ where $\underline{a_i} \in (F^*)^{d+1}$, the dual of F^{d+1}, $i = , \ldots r$; this vector subspace and the corresponding projective subspace are said to have co-dimension r.

Let Ω_l denote the collection of all projective subspaces of dimension l. Thus Ω_0 is the set of all points in the projective space, Ω_1 is the set of all lines, Ω_{d-1} is the set of all hyperplanes, etc. For $n \geq m$, define

$$\phi(n, m, s) = \frac{(s^{n+1} - 1)(s^n - 1)\ldots(s^{n-m+1} - 1)}{(s^{m+1} - 1)(s^m - 1)\ldots(s - 1)} .$$

Then the number of l-dimensional projective subspaces of $\mathbb{P}^d(GF(s))$ is given by

$$\phi(d, l, s) .$$

The number of points in an l-dimension projective subspace is given by

$$\phi(l, 0, s) .$$

The number of distinct l-dimensional subspaces through a given point is

$$\phi(d - 1, l - 1, s)$$

and the number of distinct l-dimensional subspaces, $l \geq 2$, containing a given line is

$$\phi(d - 2, l - 2, s) .$$

More generally, for $0 \leq l < m \leq d$, the number of m-dimensional subspaces of $\mathbb{P}^d(GF(s))$ containing a given l-dimensional subspace is

$$\phi(d - l - 1 , m - l - 1 , s)$$

and the number of l-dimensional subspaces contained in a given m-dimensional subspace is

$$\phi(m, l, s) .$$

3.2 Interconnection scheme based on two-dimensional geometry

In this section, we introduce the simplest scheme that is based on a two dimensional projective space $\mathbb{P}^2(F)$, where $F = GF(s)$, also called the projective plane of order s.

The number of points in a projective plane of order s is

$$\phi(2, 0, s) = \frac{s^3 - 1}{s - 1} = s^2 + s + 1 .$$

The number of lines is given by

$$\phi(2, 1, s) = \frac{(s^3 - 1)(s^2 - 1)}{(s^2 - 1)(s - 1)} = s^2 + s + 1 ,$$

which is the same as the number of points.

Each line contains $(s + 1)$ points and through any point there are $(s + 1)$ lines. Every distinct pair of points determine a line and every distinct pair of lines intersect in a point. Note that there are no exceptions to the latter rule in a projective geometry since there are no parallel lines.

Let $n = s^2 + s + 1$. Given n processors and a memory system partitioned into n memory modules $M_1, M_2,\ldots M_n$, a method of interconnecting them based on the incidence structure of a finite projective plane, can be devised as follows.

Put the memory modules in one-to-one correspondence with points in the projective space, and processors in one-to-one correspondence with lines. A memory module and a processor are connected if the corresponding point belongs to the corresponding line. Thus each processor is connected to $(s + 1)$ memory modules and vice versa. Consider a binary operation with associated index pair (i, j). Suppose $i \neq j$. Then the memory modules M_i and M_j correspond to a distinct pair of points in the projective space. This pair of points determines a line in the projective space that correspond to some processor p_l. Then the binary operation is assigned to this processor.

If $i = j$ or if the operation is unary, we have some freedom in assigning the operation. A ternary operation can not be directly performed in this scheme.

3.3 Example of an application of the two-dimensional scheme

Even this simple scheme can be used to construct practically useful devices such as a fast matrix-vector multiplier for sparse matrices with arbitrary or irregular sparsity pattern. Suppose we have an $p \times q$ matrix A; and we wish to compute the matrix-vector product

$$\underline{y} = A\underline{x} .$$

First, the indices 1 to q of the vector $\underline{x}$ and indices 1 to p of vector $\underline{y}$ are assigned to logical memory modules $M_1\ldots M_n$ by means of two functions, say f and g, which can be hashing functions

$$\text{i.e. let} \quad \mu = f(i)$$
$$\text{and } \nu = g(j) ,$$
$$\text{where} \quad \mu, \nu \in \Omega_0 .$$

Then $x(i)$ is stored in the memory module M_μ and $y(j)$ is stored in the memory module M_ν. Now consider the following multiply-and-accumulate operation, corresponding to a non-zero element $A(j, i)$ of the matrix A

$$y(j) \leftarrow y(j) + A(j, i) * x(i) .$$

We associate the index-pair (μ, ν) with this operation. Assuming $\mu \neq \nu$, let p_l be the processor corresponding to the line passing through the pair of points corresponding to memory modules M_μ and M_ν. The matrix entry $A(j, i)$ is stored in the local memory of the processor p_l. Note that the processor p_l is connected to memory modules M_μ and M_ν because the line corresponding to processor p_l contains points corresponding to memory modules M_μ and M_ν. The input operands $x(i)$ and $y(j)$ are sent from the partitioned shared memory to the processor p_l through the interconnection network. The processor p_l also receives $A(j, i)$ from its local memory and performs the arithmetic operation. The output $y(j)$ is shipped back through the interconnection network to the memory module M_ν in the partitioned shared global memory. This method is very effective if the same matrix A is used for several matrix-vector multiplications; a typical situation in many iterative linear-algebraic methods occurring in applications such as linear programming or solution of partial differential equations.

3.4 Perfect access patterns

We will first define a perfect access pattern for a two-dimensional geometry.

Let the number of points (and hence the number of lines) in the geometry be n.

A perfect access pattern is a collection of n ordered pairs of points

$$P = \{(a_1, b_1), (a_2, b_2), \ldots (a_n, b_n) | a_i \neq b_i, \\ a_i, b_i \in \Omega_0, i = 1, \ldots n\}$$

having the following properties:

1. First members $\{a_1, a_2, \ldots a_n\}$ of all the pairs form a permutation of all points of the geometry.
2. Second members $\{b_1, b_2, \ldots b_n\}$ of all pairs form a permutation of all points of the geometry.
3. Let l_i denote the line determined by the i^{th} pair of points. i.e.

   $$l_i = \langle a_i, b_i \rangle$$

   Then the lines $\{l_1, l_2, \ldots l_n\}$ determined by these n pairs form a permutation of all lines of the geometry.

	Point Pairs	Corresponding Lines
1	(p_1, q_1)	$l_1 = \langle p_1, q_1 \rangle$
2	(p_2, q_2)	$l_2 = \langle p_2, q_2 \rangle$
$\vdots$	$\vdots$	
n	(p_n, q_n)	$l_n = \langle p_n, q_n \rangle$

Table #2. Perfect access patterns for 2-d geometry

Clearly, if one schedules a collection of binary operations corresponding to such a set of index-pairs for simultaneous parallel execution, then we have the following situation:

1. There are no read or write conflicts in memory accesses.
2. There is no conflict or waiting in processor usage.
3. All processors are fully utilized.
4. Memory bandwidth is fully utilized.

Hence the name "perfect pattern."

Recall that a memory module is connected to a processor if the point α corresponding to the memory module belongs to the line β corresponding to the processor. Thus we can denote this connection by the ordered pair (α, β).

Let C denote the collection of all processor-memory connections.

i.e. $C = \{(\alpha, \beta) | \alpha \in \Omega_0, \beta \in \Omega_1, \alpha \subseteq \beta\}$.

If (a_i, b_i) is one of the pairs in a perfect pattern P and l_i is the corresponding line, we say that the perfect pattern P *exercises* the connections (a_i, l_i) and (b_i, l_i).

A sequence of perfect access patterns is called a perfect sequence if each connection in C is exercised the same number of times collectively by the patterns contained in the sequence. If such a perfect sequence is packaged as a single instruction that defines the operation of the machine over several machine cycles, it leads to uniform utilization of the communication bandwidth of the wires connecting processors and memories.

Perfect access patterns and perfect sequences can be easily generated using the group-theoretic structure of projective geometries, as described in the next section.

3.5 Group-theoretic structure of projective spaces

Recall that points in the projective space of dimension d over $GF(s)$ were defined as rays through the origin in the vector space of dimension $(d + 1)$ over $GF(s)$, which contains s^{d+1} elements. Since $s = p^k$, a power of a prime number, so is $s^{d+1} = p^{k(d+1)}$. Hence there is a unique finite field with s^{d+1} elements. One might suspect that there may be some relation between the finite field $GF(s^{d+1})$ and the projective space $\mathbb{P}^d(GF(s))$, which would give the projective space additional structure based on the multiplication operation in $GF(s^{d+1})$, besides its geometric structure. Indeed, there is such a relation, and we want to elaborate it further.

If p is a prime number, then $GF(p^m)$ contains $GF(p^n)$ as a subfield if and only if $n \mid m$. Therefore $GF(s^{d+1})$ (where $s = p^k$) contains $GF(s)$ as a subfield and the degree of $GF(s^{d+1})$ over $GF(s)$ is $(d + 1)$. Hence $GF(s^{d+1})$ is a vector space of dimension $(d + 1)$ over $GF(s)$. Each non-zero element of this vector space determines a ray through the origin and hence a point in the projective space $\mathbb{P}^d(s)$.

Let G^* = the multiplicative group of non-zero elements in $GF(s^{d+1})$

and

H^* = the multiplicative group of non-zero elements in $GF(s)$

Clearly H^* is a subgroup of G^* and both groups are cyclic. Let $x \in G^*$. The ray through origin determined by x consists of the points

$$\{\lambda x \mid \lambda \in GF(s)\} .$$

This is precisely the coset of H^* in G^* determined by x, along with the origin.

This establishes a one-to-one correspondence between points of the projective space $\mathbb{P}^d(s)$ and elements of the quotient group G^*/H^*. This correspondence allows us to define a multiplication operation in the projective space.

Let g be a fixed point in the projective space $\mathbb{P}^d(s)$ and consider the mapping of $\mathbb{P}^d(s)$ onto itself defined by

$$L_g : x \rightarrow g \circ x$$

where $g \circ x$ is the multiplication operation introduced above. It is easy to check that this operation maps lines in the projective space onto lines, planes onto planes and, in general any projective subspace of dimension k onto another projective subspace of the same dimension. Such a mapping is called an *automorphism* of the projective geometry.

Since G^* and H^* are cyclic, so is G^*/H^*. Let g be a generator of G^*/H^*. (By abuse of notation, we will not henceforth distinguish between elements of G^*/H^* and points of $\mathbb{P}^d(s)$). Thus we can denote points in $\mathbb{P}^d(s)$ as g^i, $i = 0,\dots n - 1$. The mapping L_g becomes

$$L_g : g^i \rightarrow g^{i+1} .$$

This will be called a *shift* operation.

Any power L_g^k of the shift operation is also an automorphism of the geometry and the collection of all powers of the shift operation forms an automorphism group denoted henceforth by L, which is a subgroup of the group of all automorphisms of the geometry. A subgroup G of the full automorphism group is said to act transitively on subspaces of dimension k if for any pair of subspaces H_1, H_2 of dimension k, there is an element of G which maps H_1 to H_2.

For any projective space, the shift-operation subgroup L acts transitively on the points and on subspaces of co-dimension one (hyperplanes). This property is used in the next section for generating perfect patterns and sequences for the two dimensional geometries. Perfect patterns for 4-dimensional case will be described in section 3.8.

3.6 Generation of perfect patterns for 2-d geometry

In case of a two-dimensional geometry, the hyperplanes (co-dimension = 1) are the same as lines (dimension = 1). Hence the shift operation L_g and its powers L_g^k act transitively on lines.

To generate a perfect pattern using the shift operation, take any pair of points a, b, $a \neq b$, in the geometry. Let l denote the line generated by a and b

$$\text{i.e.} \quad l = \langle a, b \rangle .$$

Set $a_0 = a$, $b_0 = b$ and $l_0 = l$ and define a_k, b_k and l_k, $k = 1,\dots n - 1$, by successive application of the shift operation L_g as follows:

$$a_k = L_g \circ a_{k-1}$$

$$b_k = L_g \circ b_{k-1}$$

$$\text{and} \quad l_k = L_g \circ l_{k-1}$$

Since L_g is an automorphism of the geometry, we have

$$l_{k-1} = \langle a_{k-1}, b_{k-1} \rangle \Leftrightarrow l_k = \langle a_k, b_k \rangle \, .$$

Then $P = \{(a_k, b_k), k = 0, \ldots n-1\}$ is a perfect pattern of the geometry.

Now in order to generate a perfect sequence of such patterns, take any line $l = \{a_1, a_2, \ldots a_k\}$ of the geometry. Form all ordered pairs (a_i, a_j), $i \neq j$, from the points on l. Generate a perfect pattern from each of the pairs. The collection of perfect patterns obtained this way (sequenced in any order) forms a perfect sequence.

3.7 Example of an application of 4-dimensional geometry

In this section we illustrate the use of higher order subspaces of a projective space by means of an architecture suitable for performing sparse Gaussian elimination, an operation required in many scientific and engineering computations.

A typical operation in symmetric gaussian elimination applied to matrix A is

$$A(i, k) \leftarrow A(i, k) - \frac{A(i, j) * A(j, k)}{A(j, j)}$$

where $A(j, j)$ is the pivot element. Such an operation needs to be carried out only if $A(i, j) \neq 0$ and $A(j, k) \neq 0$. When non-zero elements in the matrix are not in consecutive locations, it is very difficult to obtain uniformly high efficiency on vector machines. However, the operation shown above has an interesting property, regardless of the pattern of non-zero in the matrix: Two elements $A(i, j)$ and $A(i', j')$ of the matrix need to be brought together for a multiplication, division or subtraction only if the index pairs (i, j) and (i', j') have at least one of the constituent indices in common.

This property can be exploited as follows:

1. Map the row and column indices $1, 2, \ldots, l$ to points of the projective space by means of an assignment function f, which can be a hash function.

$$\alpha = f(i) \quad \alpha \in \Omega_0 \, .$$

2. Put the memory modules in the logical partition in one-to-one correspondence with lines, i.e. with elements of Ω_1.

3. A non-zero element $A(i, j)$ is assigned to a memory module as follows

   let $\alpha = f(i)$

   and $\beta = f(j)$ where $\alpha, \beta \in \Omega_0$

   Then the pair of points α, β in the projective space determines a line $l \in \Omega_1$ (if $\alpha = \beta$ we have some freedom in determining the line). The element $A(i, j)$ is stored in the memory module corresponding to line l.

4. Processors are put in one-to-one correspondence with the 2-dimensional subspaces i.e., elements of Ω_2.

If a line corresponding to a memory module is contained in a plane corresponding to a processor then a connection is made between the memory module and the processor. Again consider a typical operation in gaussian elimination

$$A(i, k) \leftarrow A(i, k) - \frac{A(i, j) * A(j, k)}{A(j, j)}$$

and let $\alpha = f(i)$, $\beta = f(j)$ and $\gamma = f(k)$. Then all the index-pairs involved in the above operation are subsets of the triplet (α, β, γ).

Assuming that the triplet of points (α, β, γ) are in general position, they determine a plane, say δ, of the projective space. i.e. $\delta \in \Omega_2$. The above operation is assigned to the processor corresponding to δ. In order to carry out this operation, the processor needs to be able to communicate with memory modules corresponding to the pairs (α, β) (β, γ) and (α, γ). Note that the lines determined by these pairs are contained in the plane determined by the triplet (α, β, γ), hence the necessary connections exist.

1. In a projective space, the number of subspaces of dimension m equals the number of subspaces of co-dimension $m + 1$. Hence if we are interested in a symmetric scheme with equal number of processors and memory modules, then we should make the co-dimension of subspaces corresponding to processors one more than the dimension of subspaces corresponding to memory modules. Thus in the present case we should choose a four dimensional geometry, i.e., $d = 4$.

2. In the projective space $\mathbb{P}^4(F)$ where $F = GF(s)$ the number of lines is

$$n = \phi(4, 1, s) = \frac{(s^5 - 1)(s^4 - 1)}{(s^2 - 1)(s - 1)} \, .$$

Therefore

$$n = (s^2 + 1)(s^4 + s^3 + s^2 + s + 1) .$$

The number of planes is

$$\phi(4, 2, s) = \phi(4, 1, s) = n .$$

Hence the number of processors is also n. The number of planes containing a given line is

$$\phi(d - 2, 0, s) = \phi(2, 0, s) = \frac{s^3 - 1}{s - 1}$$

$$= s^2 + s + 1 .$$

Let $m = s^2 + s + 1$. Thus $m = 0(\sqrt[3]{n})$. Hence each memory module is connected to $m = 0(\sqrt[3]{n})$ processors.

Similarly the number of lines contained in a given plane is $\phi(2, 1, s) = s^2 + s + 1 = m$. Hence each processor is connected to $m = 0(\sqrt[3]{n})$ memory modules.

The number of interconnections required between memories and processors can be reduced significantly if the communication between processors and memories is carried out in a disciplined and co-ordinated manner, based on the *perfect sequences* of *conflict-free patterns.* In a machine designed to support only these perfect patterns, it is possible to connect memories and processors so that the number of wires needed grows *linearly* with respect to the number of processors in the system.

Note that the fundamental property of Gaussian elimination exploited here is the fact that index pairs (i, j) and (i', j') need to interact only when they have an index in common. There are many examples of computations having the same property, e.g. finding the transitive closure of a directed graph or the "join" operation for binary relations in a relational data-base. Hence the scheme described is also applicable to such problems.

3.8 Perfect access patterns for 4-d geometry

In $\mathbb{P}^4(s)$ let n denote the number of lines, which is also equal to the number of 2-dimensional planes.

A perfect access pattern is a collection n non-collinear triplets.

$$P = \{(a_i, b_i, c_i) \mid a_i, b_i, c_i \in \Omega_0 ,$$
$$dim \langle a_i, b_i, c_i \rangle = 2 , i = 1,\dots n\}$$

having the following properties.

1. Let u_i, $i = 1,\dots n$, denote the lines generated by first two points from each triplet.

   i.e. $u_i = \langle a_i, b_i \rangle$.

   Then the collection of lines $\{u_1, u_2, \dots u_n\}$ forms a permutation of all the lines of the geometry.

2. Let v_i, $i = 1,\dots n$, denote the lines

   $$v_i = \langle b_i, c_i \rangle .$$

   Then the collection of lines $\{v_1, v_2, \dots v_n\}$ forms a permutation of all the lines of the geometry.

3. Let w_i, $i = 1,\dots n$, denote the lines $\langle c_i, a_i \rangle$

   $$w_i = \langle c_i, a_i \rangle .$$

   Then the collection of lines $\{w_1, \dots w_n\}$ forms a permutation of all the lines of the geometry.

4. Let h_i, $i = 1,\dots n$, denote the planes generated by the triplets (a_i, b_i, c_i)

   $$h_i = \langle a_i, b_i, c_i \rangle .$$

   Then the collection of planes $\{h_1, h_2, \dots h_n\}$ forms a permutation of all the planes of the geometry.

When an operation having (a_i, b_i, c_i) as the associated triplet is performed, the three memory modules accessed correspond to the three lines $\langle a_i, b_i \rangle$, $\langle b_i, c_i \rangle$ and $\langle c_i, a_i \rangle$, and the processor performing the operation corresponds the plane $\langle a_i, b_i, c_i \rangle$. Hence it is clear that if we schedule n operations where associated triplets form a perfect pattern to execute in parallel in the same machine cycle then

1. there is no read or write conflicts in memory accesses
2. there is no conflict in the use of processors
3. all the processors are fully utilized
4. the memory bandwidth is fully utilized.

Hence such a collection of triplets is called perfect pattern.

Automorphisms of the geometry based on cyclic shifts are not enough for generating perfect access patterns for the 4-d geometry. First we consider other types of automorphism.

In a finite field of characteristic p, the operation of raising to the pth power

i.e. $x \rightarrow x^p$

forms an automorphism of the field.

$$\text{i.e.} \quad (x+y)^p = x^p + y^p$$

$$\text{and} \quad (xy)^p = x^p y^p \ .$$

Since the points in the projective space $\mathbb{P}^d(s)$ correspond to elements of the multiplicative group G^*/H^*, the operation raising to the p th power can also be defined on the points of $\mathbb{P}^d(s)$. It is easy to show that such operation is an automorphism of the projective space.

The operations cyclic shift and raising to the p th power together are adequate for generating the perfect access patterns for the smallest 4-dimensional geometry $\mathbb{P}^4(2)$ which has 155 lines and planes. For bigger 4-d geometries we need other automorphisms. The most general automorphism of $\mathbb{P}^d(s)$ is obtained by means of a non-singular $(d+1) \times (d+1)$ matrix over $GF(s)$. More detailed discussion of generation of perfect patterns for 4-dimensional geometries along with specific examples of practical interest will be given in a subsequent paper.

4. Areas for Further Research

The ideas presented in this paper lead to a number of interesting areas for further investigation. Some of these are briefly described below.

4.1 Efficient mapping of algorithms

In this paper, we showed how matrix multiplication and inversion can be mapped onto the proposed architecture, using two and four dimensional geometries. How best can one map a variety of other commonly used methods in scientific computation? How can one exploit geometries of even higher dimension?

4.2 Compilation

The first version of the compiler for the proposed architecture has been designed and implemented. Description of this work can be found in [DHI89], [DKR91]. Results of application of the compiler to a number of large-scale real-life problems in scientific computation arising in domains such as linear programming, electronic circuit simulation, partial differential equations, signal processing, queueing theory, etc. are reported in [DKR90]. This work on the compilation process suggests many further research topics.

4.3 System partitioning and embedding

Since the geometry of a typical medium (e.g. printed circuit boards) for building hardware is basically two-dimensional and euclidean, how does one embed a two or four dimensional finite projective geometry into such a medium? How should the system be partitioned to reduce the number of wire-crossings between different parts?

4.4 Design of custom integrated circuits

How should one define the basic ''building blocks'' of the system so that each block can be implemented as a custom integrated circuit? What should be the internal architecture of such chips in order to fully exploit the structure of underlying geometry to achieve high performance?

Results of ongoing exploration of many of these issues will be reported in a forthcoming series of papers.

References

[ADL89] Adler, I., Karmarkar, N. K. Resende, G. C. R., Veiga, G., *Data Structure and Programming Techniques for the Implementation of Karmarkar's Algorithm,* OSRA Journal on Computing, Vol. 1, No. 2, Spring 1989.

[BN71] Bell, G. C., Newell, A., **Computer Structures: Readings and Examples,** McGraw-Hill, 1971.

[DHI89] Dhillon, I. S., *A Parallel Architecture for Sparse Matrix Computations,* B. Tech. Project Report, Indian Institute of Technology, Bombay, 1989.

[HAL86] Hall, Marshall, Jr., **Combinatorial Theory,** Wiley-Intescience Series in Discrete Mathematics, 1986.

[DKR90] Dhillon, I., Karmarkar, N. and Ramakrishnan, K. G., *Performance Analysis of a Proposed Parallel Architecture on Matrix Vector Multiply Like Routines,* Technical Report #11216-901004-13TM, AT&T Bell Laboratories, Murray Hill, N.J., October 1990.

[DKR91] Dhillon, I. Karmarkar, N. and Ramakrishnan, K. G., *An Overview of the Compilation Process for a New Parallel*

Architecture, Proceedings of the Fifth Canadian Supercomputing Conference, Fredericton, N.B., Canada, June 1991.

[KAR90A] Karmarkar, N. *A New Parallel Architecture for Sparse Matrix Computations,* Proceedings of the Workshop on Parallel Processing, BARC, Bombay, February 1990, pp. 1-18.

[KAR90B] Karmarkar, N., *A New Parallel Architecture for Sparse Matrix Computations Based on Finite Projective Geometries,* invited talk at the SIAM Conference on Discrete Mathematics, Atlanta, June 1990.

[VDW70] Van Der Waerden, **Algebra,** Vol. 1, Unger 1970.