# An Overview of the Compilation Process for a New Parallel Architecture

*Inderjit S. Dhillon* *†
*Narendra K. Karmarkar* *†
*K.G. Ramakrishnan* *

* *Mathematical Sciences Research Center*
*AT&T Bell Laboratories, Murray Hill, NJ 07974, USA*
† *Indian Institute of Technology*
*Bombay 400076, India*

April, 1991

## Abstract

This paper discusses the design of a compiler for a new parallel machine. In a first version of the parallel machine, we concentrate on scientific applications that involve several numerical iterations on the same symbolic structure of the problem. The traditional compilation process is a 2-phase process,i.e., compilation followed by execution. In order to speed up later numerical iterations, we propose a 3-phase process. In the first phase, the precompilation phase, a higher level language representation of the computation, along with the symbolic structure of the input data. is converted to an intermediate representation. a data flow graph(DFG), that captures the data flow of the symbolic instance of the computation. In the second phase, the compiler takes a DFG as input and produces machine code for execution on the parallel machine. The compiler maps the data onto the memory modules and schedules the operations onto the processors in a conflict free manner, in order to achieve maximum efficiency. The final phase, the numerical execution phase. execution of the machine code given numerical input data results in several numerical iterations. For some important sparse matrix operations, we have designed a bypass compiler which directly produces machine code for the particular computation given symbolic input data.

We have designed a compiler for a novel architecture where the interconnection between the processors and memory modules is based on finite projective geometries [KAR90, KAR91]. Properties of the geometry enable the compiler to efficiently detect conflict-free operations, partition the data among the memory modules, and balance the load equally among the processors. Our simulation experiments show that high efficiency($> 90\%$) can be achieved for matrix-vector multiply routines on a parallel machine based on two dimensional projective geometries.

**Key words:** Data flow graphs, compiler, parallel architecture. finite projective geometry

Supercomputing Symposium '91
Fredericton, NB Canada
June 3-5, 1991

# 1. Introduction

In this paper, we discuss some ideas behind the design of a compiler for a parallel machine. A description of this parallel machine may be found in [KAR90, KAR91]. Research on the software and hardware aspects of this machine has been carried out at AT&T Bell Laboratories, NJ and at the Indian Institute of Technology, Bombay[DHI89-1, DHI89-2].

The paper also touches on some ideas behind the implementation of a compiler for a parallel architecture based on finite geometries. Simulation results given in Sec. 5.2 indicate that high efficiency can be obtained for matrix-vector multiply routines on such architectures.

First, we will briefly describe the organization of the parallel machine.

## 1.1. Computational Environment

The parallel machine is designed to be used as an attached processor to a general purpose machine referred to as the host processor. The two processors share a common global memory (see Fig. 1.1). The main program runs on the host machine, while computationally intensive subroutines run on the coprocessor. The parallel machine consists of partitioned memory modules, globally shared by a bank of arithmetic processors through an interconnection network.

Traditionally, the instruction set of a machine is made up of instructions, each of which typically contains sub-instructions for various elements of the system. The CPU fetches these instructions, decodes them and then sends the sub-instructions to the relevant elements of the system. For example, on decoding an *add* instruction, the adder is instructed to perform an *add* operation. The adder normally resides on the same chip as the instruction decoding unit. However, the *add* operation requires input operands which may reside on a memory module connected to the processor by a bus. The CPU would then send the address of the input operands to the memory modules, where the address would be decoded following which the input operands would be sent back to the CPU. Thus, typical instructions involve handshaking protocols between elements of the system which enable them to synchronize their actions.

We have taken a novel approach where we have designed instruction sets for each element of the attached processor. Each element has instructions which just specify that particular element's actions. There are no explicit instructions for the synchronization of various elements in the system — the synchronization is implicit in the order in which these instructions are executed by the various elements. Notice that all elements of the attached processor proceed synchronously using a global clock. We now describe the instruction sets for three important elements of the parallel machine — processors, memory modules and switches.

In our scheme, a processor performs the basic computational operations — arithmetic, logical or others. The instruction set for a processor consists of instructions encoding just the types of operations to be performed, such as add,multiply etc. The processor has *no information* about the *memory addresses of operands*. Instructions for a memory module consist of an address along with a read/write bit. The read/write bit indicates whether the execution of the instruction by the memory module results in the reading or overwriting of the contents of the particular address. The switches realize the connections between the processors and memory modules. All instructions for a switch are simply configurations which specify the connections between the input and output ports of the switch. A program for an element is a sequence of instructions drawn from the instruction set of that element. The compiler analyses the computation, schedules operations on the attached processor and as output produces a collection of programs, one for each element of the system. Thus, as opposed to the traditional instruction sequence for a machine, the instruction sequence here is distributed over the various elements of the system, and there is no duplication of information. Each element of the system follows its own instruction sequence, and computation on the whole proceeds synchronously.

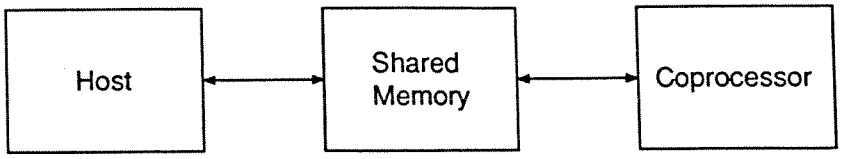


Figure 1.1: Host-Coprocessor Interface

## 1.2. Application Domain

We are mainly interested in problems arising in diverse scientific applications such as optimization, solution of partial differential equations, circuit simulation, finite element methods, signal processing, etc. In a first version of the parallel machine, we are concentrating on applications involving computationally intensive subroutines which have fixed symbolic structure but are to be executed several times with different numerical values.

Many scientific applications lead to sparse matrix computations, such as, matrix-vector multiplication and the solution of a linear system of equations. Typical iterative matrix computations arising in scientific applications involve several iterations on matrices with the same sparsity structure, each iteration involving different numerical values. These sparse matrices may involve an arbitrary or irregular pattern of non-zeros, and computations involving such sparse matrices are very difficult to parallelize on traditional vector and pipelined machines. The concepts discussed in this paper enable us to exploit parallelism in such sparse matrix computations.

# 2. Conceptual View of the Compilation Process

For purposes of this discussion, we define a computation to be an algorithm which may be applied to various input data sets. When the algorithm is applied to a specific input data set, we refer to it as an execution of the particular instance of the computation. A symbolic execution of the computation is the application of the algorithm to symbolic input data, e.g., a symbolic execution of Gaussian Elimination is the application of the corresponding algorithm to the input matrix structure. By compilation process of a computation, we refer to the process which involves the translation of a high level description of the computation to an executable module, and its subsequent execution given numerical input data.

## 2.1. A 2-Phase versus a 3-Phase process

The traditional compilation process of a computation is a 2-phase process (see Fig. 2.1) :

- a higher level language(HLL) program representing a computation is compiled into a machine language representation.
- the machine level representation is executed with a particular numerical input data set.

An HLL representation of a computation is usually machine independent, and captures the computation at a level which can be easily understood and used,e.g., a FORTRAN or C program. The machine level representation is tied to a particular machine.

As mentioned earlier, many iterative algorithms involve several numerical iterations on some fixed symbolic input data. In order to speed up the later numerical iterations, we would like to analyze the data flow in a symbolic instantiation of the computation. This would not be possible in the 2-phase process described above since nothing is known about the input data prior to the execution phase. We propose a 3-phase compilation process as shown in Fig. 2.2 :

- an HLL program and a symbolic input data set is translated into an intermediate level which captures the data flow in a symbolic instantiation of the computation. We call this intermediate level as a data flow graph(DFG) (see Sec. 2.2), and this phase as the precompilation phase.

- the DFG is analyzed and converted into machine language. We call this symbolic execution phase as the compilation phase. Thus, by definition, our **compiler** takes a DFG as input and produces machine code as output. In the ensuing discussions the term compiler will always take on the above interpretation.

- the machine level representation is executed with a particular numerical input data set. We shall call this phase as the numerical execution phase.

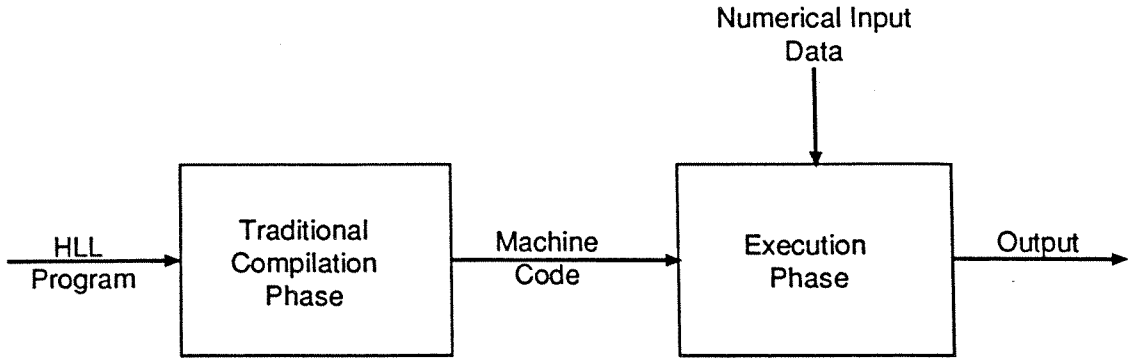


Figure 2.1: A 2-Phase Compilation Process

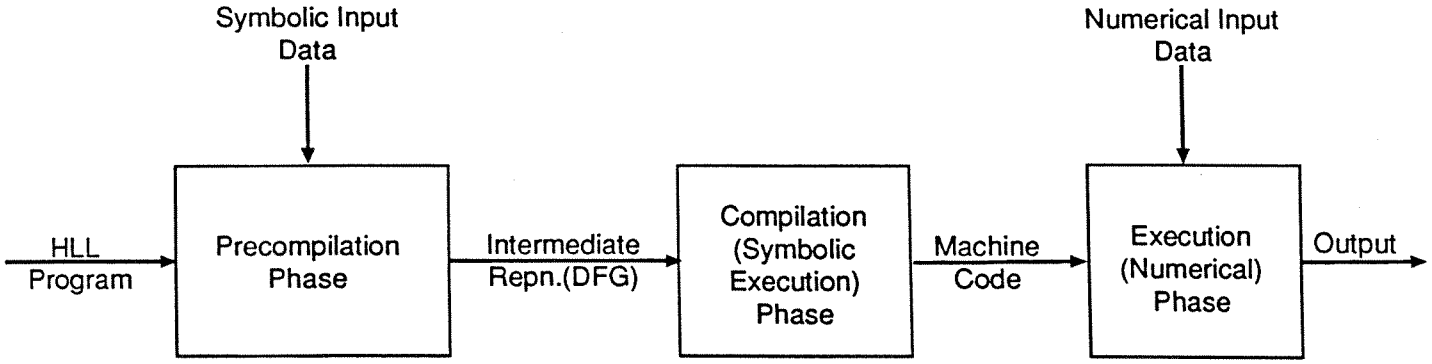


Figure 2.2: A 3-Phase Compilation Process

Note that for typical iterative computations, the first two phases need to be performed just once, while the numerical execution phase involves several iterations, with different numerical values at each iteration. Thus, the numerical execution phase is the computationally intensive phase, and our aim is to exploit the parallelism in the DFG representation to speed up this computationally intensive phase.

We now briefly discuss the intermediate level representation of a symbolic instantiation of a computation, a DFG. Such a DFG would be the input to the compiler.

## 2.2. Data Flow Graphs

A data flow graph(DFG) is a directed graph consisting of labelled nodes and directed edges. Nodes model operations while edges model operands to the operations. Edges terminating at a node represent input operands to that operation, whereas edges originating from the node represent its output operands. A node is ready to be *fired*, i.e., an operation is ready to be performed, as soon as its input operands are available. Precedence of operations is enforced by making the output of one operation as the input operand for another operation (In this case a directed edge will originate at the first node and terminate at the second node). For a particular node, its predecessors are the

nodes which produce its input operands, while its successors are the nodes which consume its output operands.

A DFG specifies a partial order among the operations whereas normal HLL representations force the user to overspecify this partial order,i.e., they force the user to give a total order to the operations in the computation even though a partial order is enough to represent the computation correctly. Thus, a DFG captures the inherent parallelism in a particular instance of a computation. There is plenty of parallelism in most computations, especially at a fine-grain level [ARV88].

For example, Fig. 2.3 shows a DFG which models the sequence of arithmetic expressions :

$$\begin{aligned} y &= x + z \\ p &= x * y \end{aligned} \tag{2.1}$$

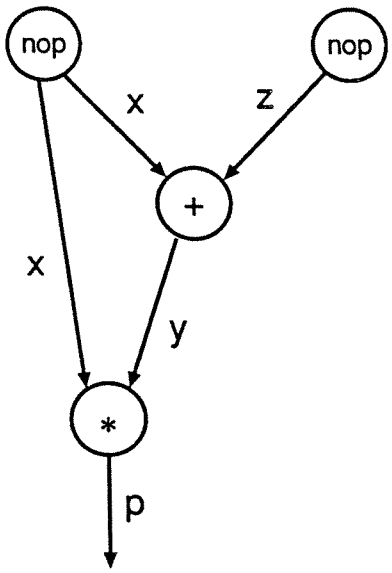


Figure 2.3: An Example DFG

In the example, all the input variables are modelled as *dummy* nodes with no input operands and *nop* as the operation. The output of a *dummy* node is the value of the variable. The example shown in Fig. 2.3 models each arithmetic operation ( +, * ) as a DFG node.

## 2.3. The Precompilation Phase

As discussed before, the preprocessing phase converts an HLL representation of the computation to a DFG. This phase is independent of the underlying machine architecture. There are many options in which this may be accomplished :

- Analysis of the HLL program.
- Automatic DFG generation by executing the HLL program with different interpretation for numerical operations.
- DFG Generator as a replacement for an HLL program text.

In the first option, the precompilation phase is subdivided into two steps as shown in Fig. 2.4. In the preprocessing phase, the HLL program is parsed and translated to a compact internal representation of the computation suitable for DFG generation. Note that this first step is independent of the symbolic input data. In the DFG generation phase, the internal representation is executed using the symbolic input data, and this produces a DFG as output.

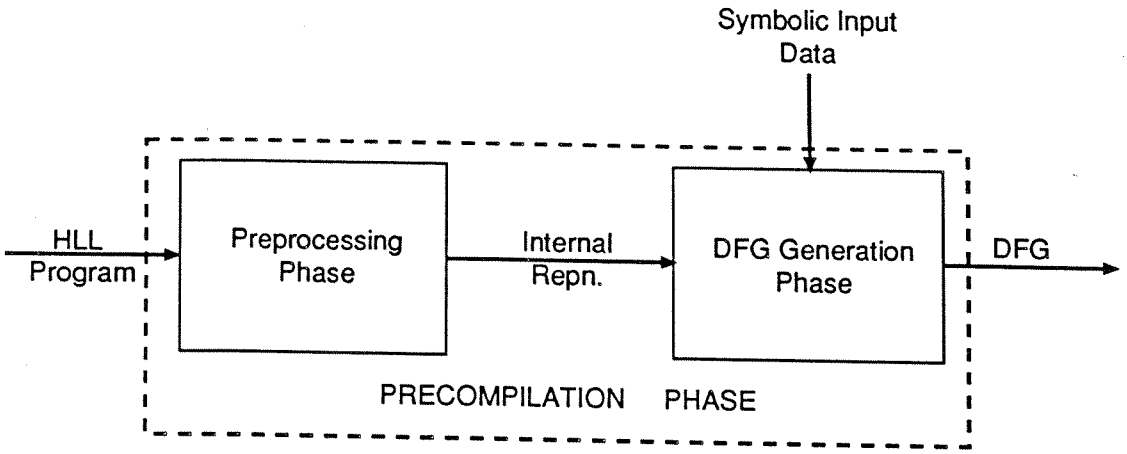


Figure 2.4: Precompilation Phase by Analysing the HLL program

In the second option, these two steps are merged into one and the need for an internal representation is obviated by directly parsing and executing the HLL program with a different interpretation for the numerical operations. For example, parsing the arithmetic operations shown in Eqn. 2.1 would result in the formation of the DFG shown in Fig. 2.3. Note that the arithmetic operations would not be executed.

In the third option, the user supplies a DFG generator, which when executed given some symbolic input data would produce the DFG(see Fig. 2.5). Notice that the DFG generator may be translated into an HLL program, which may be used for validation purposes(see Fig. 2.6).

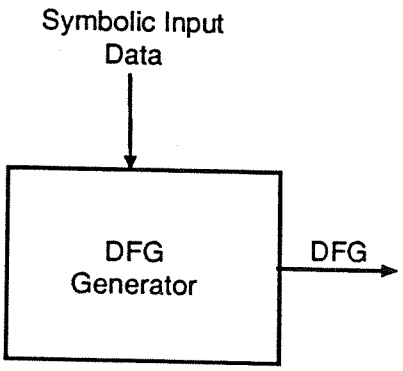


Figure 2.5: Execution of a DFG Generator

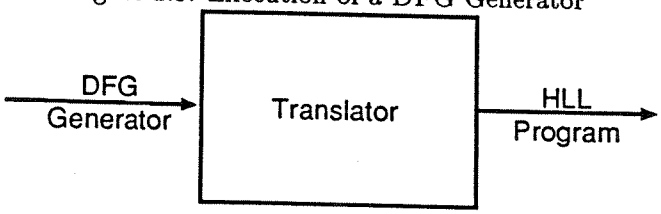


Figure 2.6: Translation of a DFG Generator into an HLL Program

## 2.4. Bypass Compiler for Specific Computations

There are some overheads associated with formation of a DFG, both in terms of its memory requirements and the time to process it. For some important computations, it is desirable if an instance

of the computation can be directly translated into machine code. We call this as a bypass compiler for the instance of the computation. Thus, given symbolic input data, the bypass compiler directly generates machine code from its own knowledge of the computation(see Fig. 2.7). The DFG for the symbolic instantiation need not be created explicitly if the properties of the DFG can be captured by alternate data structures. For example, symbolic Gaussian Elimination may be viewed as a graph problem where a pivot operation results in the formation of a clique between nodes adjacent to the pivot node [GEO81]. Thus, a data structure capturing information about the adjacencies of the nodes would suffice in such a case, enabling us to capture the properties of the DFG.

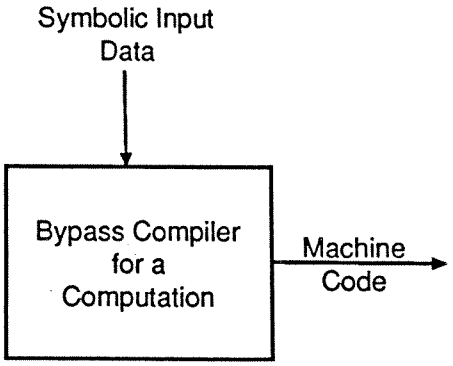


Figure 2.7: A Bypass Compiler for a Particular Computation

## 3. The Compiler

We now discuss in more detail the compiler, which takes as input a DFG, maps it onto the underlying architecture and as output, produces a collection of programs for the various elements of the system, including code which specifies how the host and the attached processor interact.

### 3.1. Conceptual View

The compiler essentially performs a symbolic execution of the DFG, rearranging it and matching the communication pattern of the particular instance of the computational program with the underlying interconnection network. The compiler uses various lists, which reflect the state of execution at a particular stage. At each stage of the execution, the compiler forms a *ready* list of operations. This list consists of all the operations which could be scheduled for numerical execution in the present cycle, if there were unlimited resources. The length of the *ready* list at each stage is indicative of the potential parallelism available at that stage. A processor would typically perform operations in an arithmetic pipeline. Different operations would require different number of cycles for execution. A multiplication operation, for example, would need far more cycles than an addition operation. The compiler simulates these arithmetic pipelines by *pipeline* lists. A *pipeline* list may be viewed as an expansion of a node in the DFG, reflecting its execution in an arithmetic pipeline. The length of each list is equal to the number of cycles needed for executing the corresponding arithmetic operation. After execution of the operation, the output operands are moved by the compiler to a *write* list. The *write* list consists of all the operations which could be scheduled for writing in the present cycle if there were unlimited resources.

The *ready* list is initialized by operations whose input operands are available at the start of the symbolic execution,i.e., by operations whose input operands are either constants, or input data to the problem itself, such as elements belonging to input matrices or vectors. From this list, only a subset of operations which don't result in memory access and processor usage conflicts, subject to switch constraints, are chosen to be scheduled in the present cycle. These operations are *fired* and removed from the *ready* list to be put into the appropriate *pipeline* list. After a particular operation has traversed through all the stages in its *pipeline* list, it is put in the *write* list. The

output operands scheduled for writing either onto the register file or onto the global partitioned memory. The compiler assigns addresses to the output operands produced by operations from the *write* list so that there are no conflicts, and then removes them from the *write* list. New operations, which can now be *fired* as a result of their input operands becoming available, are added to the *ready* list. The programs for each element are appended, and the compiler then moves onto the next stage in the computation. Additional data structures are needed by the compiler in order to store the code that it generates for various elements of the system.

An operation, after being initiated, passes through all the stages in the *pipeline* list, and only after its output operand is guaranteed to be written is the output operand made available for *firing* the successors of the operation. In this manner, the compiler ensures data consistency by making output operands available only they are written onto memory.

## 3.2. Conflict-Free Scheduling

The compiler schedules operations on the parallel machine avoiding processor, memory and switch conflicts. The manner in which the compiler assigns memory modules to operands and operations to processors, depends greatly on the underlying architecture and the switch constraints. The input data would typically be stored in some memory modules before the computation starts. The compiler may or may not generate instructions to move this input data. For intermediate data, memory modules are assigned when the particular data item is scheduled to be written. Hence, for any operation the memory modules where the input operands reside are known before the operation is scheduled. In Sec. 4, we propose an interconnection network where processor load assignment is determined by the way in which the data is partitioned among the various memory modules. The compiler would try to assign memory modules to the intermediate data in such a way that the twin objectives of processor load balancing and high efficiency could be realized. In the case of other interconnection networks, there may be some freedom in choosing the processor to perform the operations. The determination of whether the input operands can be fetched to initiate an operation depends on the computation scheme adopted. The input operands may always be fetched in consecutive cycles, or they may be fetched in non-consecutive cycles — we assume that a processor has a single port. The latter case would increase the complexity of the compiler, for the compiler would need to have some lookahead, and also have to generate additional code to identify correctly the pair of operands required for an arithmetic operation. Also, it would require a greater complexity in the instruction sequence of a processor.

A given interconnection network can be modelled by an incidence matrix representing the processor and memory connections. Given a computation scheme and the incidence matrix, the compiler needs to choose at each stage some subset of operations from the *ready* list so that maximum efficiency can be achieved. To detect processor usage, memory access and switch conflicts the compiler must maintain various *busy* lists, which contain information about the state of various elements of the system at that particular instant. Thus, in a general scheme, the compiler must examine each node in the DFG and its effects on the state of various elements of the system in order to avoid conflicts.

## 3.3. Data Structures for the DFG

In the previous sections, we observed that the compiler needs to maintain various lists to map the operations onto the parallel machine. The various lists model the state of the parallel machine and the DFG. In order that the compiler execute the DFG as described above, the data structures for the DFG must also be carefully chosen. The following information should be stored with each node:

- the *type* of operation
- an *operand count* indicating the number of operands currently needed to *fire* the operation
- back pointers to its predecessors
- forward pointers to its successors

- memory module where its output operand is to be stored
- processor which is to perform this operation

The *type* of operation encodes information about the number of machine cycles needed to perform the operation on a single processor. An *operand count* indicating the number of operands currently needed to *fire* the operation prevents unnecessary traversal of the DFG. The *operand count* is initialized to the number of input operands of the operation. Whenever a node is fired, the *operand counts* of its successors are decremented. A node is ready to be *fired*,i.e.,it is added to the *ready* list, when its *operand count* becomes zero. Note that the *operand counts* of input operands available at the start of computation are initialized to zero. The back pointers to predecessors, and forward pointers to successors capture the precedence relations among the operations. Information about the processor performing this operation, and the memory module where its output operand resides, is needed in order to be able to avoid conflicts as the node moves from the *ready* list to the *write* list, through the *pipeline* lists. Fig. 3.1 gives a high level description of the compiler.

## 3.4. Levels of Granularity in a DFG

In a DFG we define an *atomic* node to be one which models a single arithmetic or logical operation. Define a *macronode* to consist of several *atomic* nodes linked together with their precedence constraints. Note, that a *macronode* is a DFG in its own right. We impose the constraint that an *atomic* or *macronode* may be executed on a single processor. DFGs consisting entirely of *atomic* nodes capture the **fine grain parallelism** of a computation. However, the drawback in modelling fine grain parallelism is that they require a substantial amount of memory. Also, the compiler may consume a lot of time processing such a DFG. Alternatively, a **coarse grain parallelism** approach can be taken. In this approach, several arithmetic operations may be combined into a *macronode*. As an example of coarse grain parallelism, consider the matrix-vector multiply operation $\mathbf{y} = A\mathbf{x}$. One can think of computing one element of $\mathbf{y}$, i.e., $y_i = \mathbf{a}_i^T\mathbf{x}$, as a *macronode*, where $\mathbf{a}_i$ is the $i$th column vector of $A$. The DFG consisting of these *macronodes* would be more compact.

## 3.5. Difficulties in Exploiting Parallelism

Thus far, we have discussed the basic concepts behind the design of the compiler for the parallel architecture without much emphasis on the efficiency of the compiler itself. It is hoped that the compiler takes time proportional to only one or two iterations in a numerical execution of the computation represented by the DFG. A DFG with *atomic* nodes requires a substantial amount of memory, especially for very large problems ( which are just the problems one would like to solve on a supercomputer). An alternative is the use of *macronodes*, but in this way, we can only exploit a *coarser* grain of parallelism. Our aim in designing this parallel machine is to be able to achieve high efficiency on a wide range of problems — on problems with a regular structure as well as on those with arbitrary structure. In general, a *coarser* grain of parallelism would lead to a loss in efficiency on problems with arbitrary structure. Thus, there is a trade-off between the efficiency of the compiler and the efficiency achieved for the given instance of the computation on the parallel machine. One way to reduce the time to process a DFG is the identification of *supernodes* in the DFG. A *supernode* is just like a *macronode* which occurs repeatedly in the DFG. However, unlike a *macronode*, the *atomic* nodes in the *supernode* may be scheduled for execution on different processors. The compiler needs to analyze only the first occurrence of a *supernode* and generate the machine code for the execution of the *supernode* on the parallel machine. Subsequent occurrences of the *supernode* need not be analyzed — that *supernode* can simply be replaced by the equivalent machine code on the parallel machine. Note that if a particular node is both a *macronode* and a *supernode*, each occurrence of the *supernode* will be scheduled for execution on a single processor.

For important computations, we may reduce the processing time by not forming the DFG explicitly — using the bypass scheme described in Sec. 2.4.

```
procedure compiler(dfg, inc)
inputs:     dfg: data flow graph
            inc: incidence matrix modelling the interconnection network
outputs:    processor, memory and switch programs
begin
    initialize ready list with operations having input operands
            as the input data itself;
    while (ready list is not empty)
    begin
            move operations down the arithmetic pipeline lists;
            for (all operations in the write list) do
            begin
                    if (there is a free memory module)
                    begin
                            assign the memory module to this output operand;
                            mark the memory module as busy;
                            mark the appropriate switch interconnection busy;
                            remove the operation from the write list;
                            decrement operand counts of all the operations
                                    consuming this output operand;
                            if (any of the above counts becomes zero)
                                    add the operation to the ready list;
                    end
            end
            for (all operations in the ready list) do
            begin
                    if (the processor to do this operation is free
                            and the memory operands can be fetched)
                    begin
                            mark processor busy;
                            mark memory modules busy;
                            mark the appropriate switch interconnection busy;
                            remove the operation from the ready list;
                            place this operation in the appropriate pipeline list;
                    end
            end
            append processor, memory and switch programs;
    end
end
```

Figure 3.1: A high level description of the Compiler

# 4. A Compiler for an Architecture based on Finite Projective Geometries

The compiler has to schedule operations in a conflict-free manner. This may become very time consuming especially if the attempt is to exploit fine-grain parallelism. In this section, we introduce mathematical objects known as finite projective geometries on which the interconnection network may be based. Properties of this geometry enable the compiler to efficiently detect conflict-free operations, partition the data among the memory modules, and balance load equally among the processors.

We first describe some relevant properties of finite projective geometries, and then describe how one can map the physical elements in the system onto the subspaces of the geometry. The next section assumes some knowledge of finite fields which may be found in [HAL86].

## 4.1. Projective Geometries

Consider a finite field, $\mathcal{F}_s$ having $s$ elements, where $s$ is a prime power,i.e. $s = p^k$, where $p$ is a prime and $k$ is a positive integer. Let $\mathcal{F}_s^*$ consist of the non-zero elements of $\mathcal{F}_s$.

A projective geometry of dimension $d$, over a finite field $\mathcal{F}_s$, denoted by $\mathcal{P}^d(\mathcal{F}_s)$, consists of one-dimensional subspaces of the $(d+1)$-dimensional vector space, $\mathcal{F}_s^{d+1}$ over the field $\mathcal{F}_s$. A one-dimensional subspace of $\mathcal{F}_s^{d+1}$ generated by $\mathbf{x}$, $\mathbf{x} \in \mathcal{F}_s^{d+1}$, contains all elements belonging to $\mathcal{F}_s^{d+1}$ which are of the form $\lambda\mathbf{x}$, where $\lambda \in \mathcal{F}_s^*$. These one-dimensional subspaces of $\mathcal{F}_s^{d+1}$ are the points of the projective geometry, $\mathcal{P}^d(\mathcal{F}_s)$. There are $s^{d+1}-1$ non-zero elements in $\mathcal{F}_s^{d+1}$ and $(s-1)$ non-zero elements in $\mathcal{F}_s^*$. Hence, the number of points in $\mathcal{P}^d(\mathcal{F}_s)$ is given by $n_d = (s^{d+1}-1)/(s-1)$.

An $m$-dimensional subspace of $\mathcal{P}^d(\mathcal{F}_s)$ consists of all one-dimensional subspaces of an $(m+1)$-dimensional subspace of the vector space $\mathcal{F}_s^{d+1}$. If $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m$ form a basis of the vector subspace, then the elements of the vector subspace are given by $\lambda_0\mathbf{x}_0 + \lambda_1\mathbf{x}_1 + \cdots + \lambda_m\mathbf{x}_m$ where $\lambda_i \in \mathcal{F}_s^*$. The number of non-zero elements in this vector subspace is $s^{m+1}-1$. Hence, the number of points in an $m$-dimensional projective subspace is given by $n_m = (s^{m+1}-1)/(s-1)$. From now on, we shall denote by $\Omega_m$ the set of all $m$-dimensional subspaces of $\mathcal{P}^d(\mathcal{F}_s)$.

For $n \geq m$, define

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

Let $0 \leq l \leq m \leq d$. Then, the number of $l$-dimensional subspaces of $\mathcal{P}^d(\mathcal{F}_s)$ contained in a given $m$-dimensional subspace is $\phi(m,l,s)$. The number of $m$-dimensional subspaces of $\mathcal{P}^d(\mathcal{F}_s)$ containing a given $l$-dimensional subspace is $\phi(d-l-1, m-l-1, s)$.

## 4.2. Parallel Architectures based on Projective Geometries

First, we present an example of how to map the the physical elements of a parallel processing system onto the subspaces of a projective geometry.

### 4.2.1. An Example

Consider a two-dimensional projective geometry, $\mathcal{P}^2(\mathcal{F}_s)$. The number of points, as well as the number of lines in $\mathcal{P}^2(\mathcal{F}_s)$ equals $s^2+s+1$. Each line contains $(s+1)$ points and through any point there are $(s+1)$ lines. Every distinct pair of points determines a unique line, and every distinct pair of lines intersects in a unique point.

Based on this geometry, we construct a parallel architecture as follows. The memory modules are put in one-to-one correspondence with points of $\mathcal{P}^2(\mathcal{F}_s)$ and the processors in one-to-one correspondence with lines. A processor is connected to a memory module if the line corresponding to the processor contains the point corresponding to the memory module. Let $n = s^2+s+1$. Let $P_0, P_1, \ldots, P_{n-1}$ and $M_0, M_1, \ldots, M_{n-1}$ denote the processors and memory modules respectively.

Consider a binary operation, denoted by $\circ$, :

$$c \leftarrow a \circ b$$

Let the input operands $a$ and $b$ reside in memory modules $M_i$ and $M_j$ respectively. These modules correspond to two distinct points in $\mathcal{P}^2(\mathcal{F}_s)$. Since two distinct points uniquely determine a line, this binary operation is uniquely assigned to a processor. The output operand may be placed in any of the memory modules connected to this processor. If $i = j$ or the operation is unary, we have some freedom in choosing the processor to which this operation is assigned. In such a scheme, a ternary operation would have to be broken up into binary operations.

Any DFG could be broken up into a sequence of binary and unary operations. It could be compiled for execution on such a parallel machine by assigning memory modules to the input operands of operations in the *ready* list. The processor load assignment is automatic once the memory modules of the input operands are fixed. It is desirable that the input and intermediate data be mapped onto the memory modules in a way that the load be balanced equally among the processors. In cases where the computation is naturally associated with an index pair and each data item is associated with a single index, the data mapping may be based on a hash function which hashes the single index onto a memory module. For example, a basic operation in the matrix-vector multiplication,$\mathbf{y} = A\mathbf{x}$, is the multiply-accumulate operation:

$$y_i \leftarrow y_i + a_{ij}x_j$$

The vector elements, $y_i$ and $x_j$ may be mapped onto memory modules $M_\mu$ and $M_\nu$ by hashing functions $f$ and $g$, such that $\mu = f(i)$ and $\nu = g(j)$. This operation would then be performed by the processor connected to $M_\mu$ and $M_\nu$.

Such an interconnection scheme allows efficient computation in cases where a large number of operations are associated with an index pair. For example, it can be used for very fast matrix-vector multiply involving matrices with arbitrary or irregular structure. For details and simulation results, refer to [DHI90].

### 4.2.2. A General Scheme

In a general scheme, an architecture may be based on projective geometries by mapping processors onto $k$-dimensional subspaces and memory modules onto $j$-dimensional subspaces. In such an architecture, the number of processors equals $\phi(d, k, s)$ and the number of memory modules equals $\phi(d, j, s)$. The number of processors may be made equal to the number of memory modules by choosing the dimension of the projective geometry such that $\phi(d, k, s)$ equals $\phi(d, j, s)$. This happens when $d = j + k + 1$. Suppose that $0 \leq j < k < d$. The number of processors connected to a memory module equals $\phi(d - j - 1, k - j - 1, s)$, while the number of memory modules connected to a processor equals $\phi(k, j, s)$. The number of processors connected to a memory module equals the number of memory modules connected to a processor in the case where $d = j + k + 1$. Thus, an architecture based on $\mathcal{P}^d(\mathcal{F}_s)$ where $d = j + k + 1$ may be called a symmetric architecture.

A DFG may be compiled for execution on such a parallel machine as follows. The data operands are to be mapped onto the memory modules or onto processors. It is preferable that a data operand, which is to be used only by a single processor, be mapped onto a processor so that it can be directly stored in the processor's local memory or registers, and does not have to go through the interconnection network. This data mapping may be achieved by assigning to each data operand either a $j+1$-tuple or a $k+1$-tuple. The data operand associated with a $j+1$-tuple may be stored in the memory module corresponding to the $j+1$-dimensional projective subspace determined by the $j+1$-tuple or in the local memory of the processor determined by the $k+1$-tuple. In this architecture, two memory modules may not be connected to a common processor unless $2j + 1 \leq k$. Hence, if $2j+1 > k$ it is desirable that any operation be associated with a $k+1$-tuple so that it can be directly mapped onto a processor. Suppose that each operation in the DFG is associated with a $k+1$-tuple. Such a structure may be present in the original DFG, or may be achieved by a partitioning of the DFG into *macronodes*, where each *macronode* can be associated with a $k+1$-tuple.This operation s scheduled for execution on the processor determined by the $k+1$-tuple. Operations which may be performed without processor or memory conflicts subject to switch constraints, are chosen from the *ready* list to be scheduled on the parallel machine.

An example where $2j + 1 > k$ is a symmetric architecture based on a $\mathcal{P}^4(\mathcal{F}_s)$ where $j = 1$ and $k = 2$. In this architecture, two arbitrary memory modules need not be connected to a common processor. Each memory module is determined by a pair of indices and only those memory modules which share an index are connected to a common processor. A property of Gaussian Elimination is that two matrix elements come together for an operation only if they share a common index. This provides us with a natural way of mapping data to memory modules, and operations to processors on an interconnection network based on $\mathcal{P}^4(\mathcal{F}_s)$, where memory modules correspond to lines and processors to planes[KAR90, KAR91].

Observe that the processor load assignment is automatic, and is determined by the mapping of the data operands onto the memory modules. However, to determine the operations that don't lead to any conflicts, the compiler has to scan each individual operation, and maintain several lists which capture the current state of the parallel machine. In the next section, we show how some properties of finite projective geometries can be used to partition the *ready* list into buckets, each of which is conflict-free by construction. This greatly simplifies the identification of conflict-free operations by the compiler.

## 4.3. Partitioning the *Ready* List into Conflict Free Buckets

A connection pattern specifies the connections of processors to memory modules at a particular instant of time. We represent a connection pattern at time instant $t$ by a set of ordered pairs, $C_t = \{(P_i, M_j) : 1 \leq i \leq p, 1 \leq j \leq m\}$, where $p$ is the number of processors, and $m$ is the number of memory modules. $(P_i, M_j) \in C_t$ if and only if processor $P_i$ is connected to module $M_j$ at time instant $t$. We assume that all processors have a single port, and so can be connected to only one memory module at a particular instant of time. Therefore, $|C_t| \leq p$. A conflict free connection pattern is one where each processor is connected to a memory module, i.e., $|C_t| = p$.

We give an example of some conflict free connection patterns in an architecture based on $\mathcal{P}^2(\mathcal{F}_2)$ having 7 processors and 7 memory modules :

$$C_1 = \{(P_0, M_0), (P_1, M_1), (P_2, M_2), (P_3, M_3), (P_4, M_4), (P_5, M_5), (P_6, M_6)\}$$

$$C_2 = \{(P_0, M_1), (P_1, M_2), (P_2, M_3), (P_3, M_4), (P_4, M_5), (P_5, M_6), (P_6, M_0)\}$$

$$C_3 = \{(P_0, M_3), (P_1, M_4), (P_2, M_5), (P_3, M_6), (P_4, M_0), (P_5, M_1), (P_6, M_2)\}$$

Note that in a conflict free connection pattern, the first indices of all pairs form a permutation of the processors while the second indices form a permutation of all the memory modules. Thus, there are no conflicts in memory access and processor usage. At the same time, all the processors and memory modules are being utilized. Clearly, the number of possible connection patterns is much greater than the number of conflict-free connection patterns. The hardware is simplified if these conflict free connection patterns are the only allowable connection patterns. Simulations on a parallel machine based on $\mathcal{P}^2(\mathcal{F}_s)$ show that under certain conditions, such a machine does not lose much in efficiency[DHI90]. We now describe how the structure of the geometry can be exploited for easy detection of conflict free operations.

Consider a symmetric architecture where processors correspond to $k$-dimensional subspaces, and memory modules to $j$-dimensional subspaces. Automorphisms of the projective geometry can be used to generate all subspaces of a particular dimension[KAR90]. In particular, all $k$-dimensional subspaces may be represented as ordered $n_k$-tuples of points, where each ordered tuple can be obtained from another by means of an automorphism of the geometry. Choose $j + 1$ linearly independent points from positions $i_0, i_1, \ldots, i_j$ in each of these tuples. Then all the $j$-dimensional subspaces determined by these $j + 1$ points are distinct. Let $v \in \Omega_k$ and $w \in \Omega_j$, where $w$ is formed from the tuple representing $v$ as described above. Form a set of ordered pairs $(v, w)$, where $v$ ranges over all $k$-dimensional subspaces,(and hence, $w$ ranges over all $j$-dimensional subspaces). Clearly, this forms a conflict free connection pattern in this architecture.

Suppose now that each operation in the *ready* list requires $l$ input operands and produces $m - l$ output operands, where $0 < l < m$. Assume that every operation can be associated with a $k+1$-tuple and every operand can be associated with a $j + 1$-tuple. Extending the method described above,

form a set of ordered $m+1$-tuples $(v, w_1, w_2, \ldots, w_m)$ where $v$ ranges over all the $k$-dimensional subspaces, and $w_i$ ranges over all the $j$-dimensional subspaces. This set realizes $m$-conflict free connection patterns. The $w_i$'s correspond to the memory modules where the input and output operands reside. We can form a collection of such sets such that an operation can be put in a unique set. Thus, the *ready* list can be partitioned into such sets, which we call conflict free buckets. Operations in these buckets are conflict free by construction. In this way, properties of a finite projective geometry enable the compiler to efficiently detect conflict free operations.

# 5. Conclusions

## 5.1. Library of Routines

For the first version of the machine, we plan to develop a library of routines for specific parallel architectures based on finite projective geometries :

- a compiler which takes a DFG as input and produces machine code.
- a DFG Generator which generates a DFG for some important sparse matrix computations, like matrix-vector multiply and Gaussian Elimination(see Sec. 2.3).
- a Bypass Compiler for important sparse matrix computations, like matrix-vector multiply and Gaussian Elimination (see Sec. 2.4).

We expect that the user would augment the existing routines with his own library of routines, so that he can use the parallel machine for efficient execution of problems arising in particular applications.

## 5.2. Simulation Results

We have conducted extensive simulations to analyze the performance of a parallel architecture based on two-dimensional finite projective geometries on matrix-vector multiply like routines. Some simulation results for an architecture based on $\mathcal{P}^2(\mathcal{F}_2)$ are presented in Table 5.1. In this table, we have listed simulation results only on large problems drawn from various applications. The problem sizes, efficiencies of execution on the parallel machine and the memory overheads for the parallel implementations are tabulated in the results. The total memory for the parallel implementation is the sum of data memory and instruction memory required for various elements of the system, whereas the total memory required for a serial implementation is the storage required for a sparse representation for the matrix. In a serial implementation, we have assumed a sparse matrix data structure as given in [ADL89] for a column-wise representation of the matrix. The machine is restricted to have only conflict free connection patterns(see Sec. 4.3). It is observed that computation may be allowed to proceed in an SIMD fashion without much degradation in performance. This would lead to a further saving in memory requirements. More details about the problem classes may be found in [KAR89], and about the simulations in [DHI90].

High efficiency is obtained on problems drawn from a wide variety of applications. The fact that the matrices have a regular or arbitrary structure does not lead to much variation in the results obtained. This is as opposed to traditional existing supercomputers which perform extremely well on problems with a regular structure but not so well on problems with arbitrary structure. This is observed in the Perfect Club Benchmarks [BER89] where most of the supercomputers are seen to perform well on problems with regular structure but give poor performance on problems, such as SPICE for circuit simulation, which have arbitrary structure and are difficult to vectorize at a coarse-grain level.

The compiler is required to run only once in the beginning when it schedules operations for later numerical executions. Speedup results from the fact that there are many subsequent numerical executions of the same DFG which justify the initial cost of compilation. Certain routines, such as an $n$-point fast fourier transform, could be compiled just once and stored in secondary memory. Thus, when these routines are to be used, their compiled code need just be loaded into main memory and executed.

| Problem | Problem Size | | | Efficiency (%) | Memory Overhead (%) |
|---|---|---|---|---|---|
| | Rows | Columns | Nonzeros | | |
| Wave Mechanics | 147,456 | 147,456 | 737,280 | 99.99 | 25.72 |
| Fast Fourier Transforms | 65,536 | 65,536 | 131,072 | 99.98 | 5.92 |
| Circuit Simulation | 3,388 | 3,388 | 44,616 | 84.57 | 27.60 |
| Hypergraph Covering | 7,500 | 25,098 | 75,294 | 97.15 | 10.27 |
| Minimum Cost Network Flow | 40,001 | 80,000 | 159,800 | 98.88 | 4.66 |
| Partial Differential Equations | 40,000 | 80,000 | 240,000 | 99.98 | 8.80 |
| Linear Ordering | 4,950 | 333,300 | 980,100 | 94.19 | 11.91 |
| Completely Dense | 1,000 | 3,000 | 2,001,000 | 99.80 | 23.83 |
| Control Systems | 4,105 | 15,809 | 1,159,628 | 97.87 | 24.21 |
| Queuing | 6,252 | 31,252 | 299,127 | 92.07 | 22.58 |

Table 5.1 : Simulated Performance of the Code Generated
by the Compiler for a Parallel Architecture for Matrix-Vector Multiply Routines[1]

[1] No. of Processors = No. of Memory Modules = 7
No. of Memory Modules Connected to a Processor = No. of Processors Connected to a Memory Module = 3.

## References


[ADL89] Ilan Adler, Narendra K. Karmarkar, Mauricio G.C.R. Resende, and Geraldo Veiga. Data Structures and Programming Techniques for the Implementation of Karmarkar's Algorithm. *ORSA Journal on Computing*, 1(2), Spring 1989.

[ARV88] Arvind, David E. Culler, and Gino K. Maa. Assessing the Benefits of Fine-grain Parallelism in Dataflow Programs. In *Proceedings of Supercomputing '88*, pages 60–69. IEEE Computer Society Press, November 1988.

[BER89] M. Berry et al. The PERFECT Club Benchmarks : Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 1989.

[DHI89-1] Inderjit S. Dhillon. A Compiler for Sparse Matrix Computations on New Parallel Architectures. Conference on Interior Point Methods, Bombay, January 1989.

[DHI89-2] Inderjit S. Dhillon. Parallel Architectures for Sparse Matrix Computations. B.Tech Thesis, Indian Institute of Technology, Bombay, April 1989.

[DHI90] Inderjit S. Dhillon, Narendra K. Karmarkar, and K.G. Ramakrishnan. Performance Analysis of a Proposed Parallel Architecture on Matrix-Vector-Multiply-Like Routines. Technical Report 11216-901004-13TM, AT&T Bell Laboratories, Murray Hill, NJ, October 1990.

[GEO81] A. George and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1981.

[HAL86] Marshall Hall, Jr. *Combinatorial Theory.* Wiley-Interscience Series in Discrete Mathematics, 2nd edition, 1986.

[KAR89] Narendra K. Karmarkar and K.G. Ramakrishnan. Implementational and Computational Results of the Karmarkar Algorithm for Linear Programming, Using an Iterative Method for Computing Projections. Technical Report 11211-891011-10TM, AT&T Bell Laboratories, Murray Hill, NJ, October 1989.

[KAR90] Narendra K. Karmarkar. A New Parallel Architecture for Sparse Matrix Computations. Invited Talk at the SIAM Conference on Discrete Mathematics, Atlanta, June 1990.

[KAR91] Narendra K. Karmarkar. A New Parallel Architecture for Sparse Matrix Computations based on Finite Projective Geometries. In *Proceedings of Supercomputing Symposium '91*, June 1991.