AT&T

Document Cover Sheet for Technical Memorandum

itle: Performance Analysis of a Proposed Parallel Architecture On Matrix Vector Multiply Like Routines

Authors	Location	Ext.	Company (If other than AT&T-BL)
I. S. Dhillon	MH 6D-422	2957	
N. K. Karmarkar	MH 2C-358	6737	
K. G. Ramakrishnan	MH 2C-126	6722	

Document Nos.	Filing Case No.	Project Nos.
11216-901004-13TM 11211-901004-17TM 11212-901004-34TM	20878	311404-2299 311404-2399 311404-2899

Keywords:

compiler; finite geometry; data flow graph; sparse matrix computations; scientific computing

MERCURY Announcement Bulletin Sections

MAS - Mathematics and Statistics | CMP - Computing | CMM - Communications

Abstract

This paper presents some ideas on the implementation of a compiler which uses the knowledge of the finite geometry that underlies the novel parallel hardware architecture for sparse matrix computations [KAR90]. The compiler takes a data flow graph as input, rearranges it to avoid memory, switch, and processor conflicts, and schedules operations to maximize the efficiency of the parallel hardware. The action of the compiler can be viewed as a discrete-time-driven simulation of the execution of the data flow graph on the parallel machine, the simulation capturing the state of the hardware at a particular time instant.

The paper presents extensive simulation results for matrix-vector multiply routines on the parallel hardware. The matrices have been chosen from diverse LP applications as well as other scientific computations. The results indicate that uniformly high efficiency (above 90%) is achievable on problems with regular as well as arbitrary structure.

Total Pages (including document cover sheet): 67

Mailing Label

Complete Copy

DPHs 1121, 5284
MTSs 11211, 11212, 11216, 11357
Dept. 59136
A. G. Fraser
M. R. Garey
R. L. Graham
G. Arnold
A. K. Curtis
D. J. Houck, Jr.

C. J. McCallum, Jr.
K. Singhal
P. Agrawal
J. Trotter
R. Chadha
J. Fishburn
P. Lloyd
H. Nham
T. G. Szymanski
K. C. Hsu
A. Singhal
N. Woo

Cover Sheet Only

Center 1121
DPHs 1127, 1125, 1135, 5285, 5286

H. T. Brendzel
A. A. Penzias
A. V. Aho
J. M. Holtzman
S. Horing
B. W. Kernighan
R. W. Lucky
K. E. Martersteck
A. N. Netravali

Future AT&T Distribution by ITDS

RELEASE to any AT&T employee (excluding contract employees).

Author Signatures

I. S. Dhillon | N. K. Karmarkar | K. G. Ramakrishnan

Organizational Approval (Optional)

D. S. Johnson | A. M. Odlyzko | D. Mitra | M. R. Garey

For Use by Recipient of Cover Sheet:

Computing network users may order copies via the *library -1* command; for information, type *man library* after the UNIX® system prompt.

Otherwise:
Enter PAN if AT&T-BL (or SS# if non-AT&T-BL). _____
Return this sheet to any ITDS location.

Internal Technical Document Service

( ) AK 2H-28
( ) ALC 1B-102
( ) CB 1L-220
( ) HO 4F-112
( ) IH 7M-103
( ) MV 3L-19
(X) WH 3E-204
( ) DR 2F-19
( ) INH 1C-114
( ) IW 2Z-156
( ) MT 2C-131
( ) NW-ITDS
( ) PR 5-2120

AT&T Bell Laboratories

subject: **Performance Analysis of a Proposed Parallel Architecture On Matrix Vector Multiply Like Routines**
Work Project No. 311404-2299 311404-2399 311404-2899
File Case 20878

date: **October 4, 1990**

from: **I. S. Dhillon**
Org. 11216
MH 6D-422 (908) 582-2957

N. K. Karmarkar
Org. 11211
MH 2C-358 (908) 582-6737

K. G. Ramakrishnan
Org. 11212
MH 2C-126 (908) 582-6722

11216-901004-13 TM
11211-901004-17 TM
11212-901004-34 TM

TECHNICAL MEMORANDUM

1. Introduction

A large fraction of scientific and engineering computations involve sparse matrices, the most typical operations being sparse matrix multiplication and solution of linear system of equations. Typically, these problems lead to computations involving a fixed data flow graph which is executed several times with different instances of input data. Such problems arise in diverse applications like optimization, solution of partial differential equations, finite element methods, signal processing, circuit simulation, design of integrated circuits, etc.

In this paper we discuss some of the ideas behind the implementation of a compiler, which exploits the interconnection structure of a novel parallel architecture based on finite geometries [KAR90]. The compiler takes a data flow graph as input, rearranges it to avoid memory access and processor usage conflicts subject to switch constraints, and ultimately, schedules the operations on the processors distributing the load evenly among them. This symbolic

computation of the data flow graph is to be done once in the beginning. After the compiler schedules the operations, the numerical execution of the data flow graph proceeds on the parallel machine. The action of the compiler may be thought of as a discrete time-driven simulation of the execution of a data flow graph on the parallel machine, the simulation capturing the state of execution at particular time instants enabling synchronous numerical execution of the data flow graph on the parallel machine after the compilation. The speedup results from the fact that this numerical execution is done many times, whereas the compiler is required to run only once in the beginning.

1.1 Computational Environment

The parallel machine is meant to be used as a co-processor to a general purpose host processor. The two processors share a common global memory. The main program runs on the host machine. Computationally intensive subroutines, which have fixed symbolic structure but are to be executed several times with different numerical values, run on the co-processor [refer Fig. 1.1].

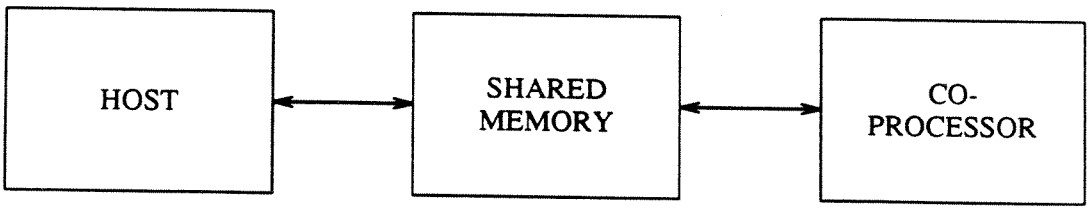


Fig. 1.1

The co-processor consists of partitioned memory modules, globally shared by a bank of arithmetic processors through an interconnection network. All the processors of this co-processor are identical. Each element of the co-processor has local memory for storing instruction sequences to be used by that element. The instruction sequence to be executed by an arithmetic processor consists only of the type of operation to be performed (including noops). The

instruction sequence does not include addresses of operands – the processor is to operate on whatever data comes on its input ports. In addition, *move* instructions for moving data from one memory module to another may be present. The instruction sequence for a memory module consists of lists of addresses along with a read/write bit, while that for a switch simply consists of a list of configurations, each configuration specifying the connections between the input and output ports of the switch. Thus, as opposed to the traditional instruction sequence for a processor, 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 the computation on the whole proceeds synchronously. For a more detailed discussion of the coprocessor architecture, refer to KAR90.

2. The Compiler

The instruction sequences for various elements of the co-processor are generated by the compiler. The compiler analyzes the data flow graph, maps it onto the underlying architecture and as output, produces a collection of programs, one for each element of the system, each program consisting of a list of instructions drawn from the instruction set of that element.

A data flow graph consists of nodes representing operations, and edges representing operands. A node can be *fired*, i.e. an operation can be initiated, only when its input operands are available. We shall be considering only unary and binary operations in data flow graphs. The following information is needed along with each node:

- the type of operation
- the nodes which provide its input operands
- the nodes which consume its output operands.

We now discuss the compiler in somewhat greater detail (refer Fig. 2.1). The compiler,

```
procedure compiler(dfg, inc)
begin
{       dfg: data flow graph
        inc: incidence matrix reflecting the interconnection network  }
        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;
                add the completed operations to the write list;
                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 (a 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 2.1: A high level description of the compiler

essentially performs a symbolic computation of the data flow graph, rearranging it and matching the communication pattern of the computation with the underlying interconnection network. The compiler uses various lists, which reflect the state of computation at a particular stage. At each stage of computation, the compiler forms a *ready* list of operations. This list consists of all the operations which could be performed in the present cycle, if there were unlimited resources. 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. 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 written 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 computation, i.e. 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 those 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 is completed, i.e.after it has traversed through all the stages in its *pipeline* list, it is put in the *write* list. The output operands may be written 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 in a conflict-free manner, 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.

In order that the compiler traverse the data flow graph as described above, the data structures for the data flow graph must 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 operations producing its input operands
- forward pointers to operations consuming its output operand
- memory module where its output operand is stored
- processor which performs this operation.

A *pipeline* list may be viewed as an expansion of the nodes corresponding to that operation in the data flow graph. An operation, after being initiated, passes through all the stages in the *pipeline* list, and it is only after being written that the operand counts of the operations consuming this output operand are decremented. Thus, the compiler ensures data consistency since the output operands are made available only after being written onto memory.

The compiler resolves memory access and processor usage conflicts. The manner in which the compiler resolves conflicts, and 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 move this input data. For intermediate data, memory modules are assigned at the time the particular data item is to be written. Hence, for any operation the memory modules where the input operands reside are known before the operation is performed. Depending on the interconnection network, there may be some freedom in choosing the processor to perform these operations. On the other hand, the processor may be determined by the memory

modules assigned to the input operands. The determination of whether the input operands can be fetched to initiate an operation depends on the computation scheme adopted. The input operands (a maximum of two in our case) may always be fetched in consecutive cycles, or they may be fetched in non-consecutive cycles – we assume that a processor has a single input port. The latter case would increase the complexity of the compiler. The compiler would need to have some look-ahead, 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.

3. Interconnection Scheme

Until now, we have discussed the compiler independently of the underlying architecture. As mentioned before, the compiler is greatly influenced by the interconnection network. We now discuss projective spaces over finite fields, which form the basis of our interconnection network. The next few sections assume some knowledge of finite fields, which may be found in HAL86.

3.1 Projective Spaces Over Finite Fields

Let us consider the Galois Field, $GF(s^{d+1})$, s being a prime power. This is a vector space over its subfield $GF(s)$. Let x be a primitive root of $GF(s^{d+1})$. The non-zero elements of $GF(s^{d+1})$ may be represented as $1, x, x^2, \ldots, x^{s^{d+1}-2}$. These non-zero elements form a multiplicative cyclic group, say G. Let $n_d = \frac{s^{d+1}-1}{s-1}$. Let H be the multiplicative cyclic subgroup generated by x^{n_d}. The elements of H are of the form $1, x^{n_d}, x^{2n_d}, \ldots, x^{(s-2)n_d}$ and together with the zero element, they form the subfield, $GF(s)$. Consider the quotient group, $G' = G/H$, $|G'| = n_d$. Two elements of $GF(s^{d+1})$ belong to the same coset of H if and only if $i \equiv j \pmod{n_d}$. The points of a projective space over the field $GF(s)$, denoted by $\mathbb{P}^d(GF(s))$, are in one-to-one correspondence with the elements of the quotient group G'. Henceforth, we shall always refer to an element of G' by its representative x^i, $i < n_d$. In the projective space, a

point is a subspace of dimension 0, a line is one of dimension 1, and so on.

For $n \geq m$, define

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

Let $0 \leq l \leq m \leq d$, and $s=p^k$ where p is a prime. Then the number of m-dimensional subspaces of $\mathbb{P}^d(GF(s))$ containing a given l-dimensional subspace is:

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

The number of l-dimensional subspaces of $\mathbb{P}^d(GF(s))$ contained in a given m-dimensional subspace is:

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

From now on, we shall refer to the number of points of $\mathbb{P}^d(GF(s))$ contained in a j-dimensional subspace as n_j and the number of j-dimensional spaces in $\mathbb{P}^d(GF(s))$ as h_j

$$n_j = \phi(j,0,s) \quad \text{and} \quad h_j = \phi(d,j,s) .$$

3.2 Structure of $\mathbb{P}^d(GF(s))$

We now present some theorems and their proofs which are needed for an efficient implementation of the compiler.

Let W_j be the set of all j-dimensional subspaces of $\mathbb{P}^d(GF(s))$. We represent $w \in W_j$ as an n_j-tuple $(x^{i_1}, x^{i_2}, \ldots, x^{i_{n_j}})$, $x^{i_k} \in G'$ where x is the primitive root of $GF(s^{d+1})$.

Define an action of G' on W_j, $\psi : W_j \rightarrow W_j$ by $\psi_g(w) = g \cdot w = g(x^{i_1}, \ldots, x^{i_n}) = (gx^{i_1}, \ldots, gx^{i_n})$, $g \in G'$, $w \in W_j$. Define the orbit of w to be $\theta(w) = \{\psi_g(w) \mid g \in G'\}$, and the isotropy subgroup of w, $I(w) = \{g \in G' \mid \psi_g(w) = w\}$. We call $x^i \cdot w = x^i(x^{i_1}, \ldots, x^{i_n}) = (x^i \cdot x^{i_1}, \ldots, x^i, x^{i_n})$ a shift of the subspace w by step i

Intuitively, the orbit of w, $0(w)$ consists of all the j-dimensional subspaces which can be generated by shifting the subspace by steps i, $i \geq 1$. The isotropy subgroup of w, $I(w)$ consists of all points $x^k \in G'$, such that shifting the subspace w by step k, results in the same subspace, i.e., permutes all the points on the subspace.

The action of G' on W_j decomposes W_j into a collection of mutually disjoint subsets, $W_j = \bigcup_{w \in W_j} \theta(w)$. Let $l_j(w) = |\theta(w)|$ be called the length of the orbit of the j-dimensional subspace. Then the following relation holds:

$$l_j(w) \cdot o(I(w)) = o(G') = n_d \quad \text{i.e.} \quad l_j(w) \mid n_d \; \forall \; w \in W_j \; .$$

Theorem. *If* $(n_j, n_d)^* = 1$, *then the length of the orbit of any j-dimensional subspace equals the number of points in* $\mathbb{P}^d(GF(s))$ *i.e.*

$$l_j(w) = n_d \; \forall \; w \in W_j \; .$$

Proof. We already have $l_j(w) \mid n_d \; \forall \; w \in W \Rightarrow \exists$ an integer r st $n_d = r \cdot l_j(w) \Rightarrow r | n_d$.

$l_j(w)$ is the smallest integer such that $\psi_g(w) = w$ where $g = x^{l_j(w)}$, i.e. ψ_g permutes the points in the j-dimensional subspace w. Represent this permutation by π. π^r must be the identity permutation since π^r corresponds to a shift of step $r \cdot l_j(w) = n_d \therefore , r | n_j$.

Since $r | n_d$ and $r | n_j$, $r | (n_d, n_j)$, and if $(n_j, n_d) = 1$, $r = 1 \Rightarrow l_j(w) = n_d$.

Corollary 1: [Singer's Theorem]. *The hyperplanes of* $\mathbb{P}^d(GF(s))$ *as blocks, points as objects, form a symmetric cyclic block design.* [HAL86]

Proof. In $\mathbb{P}^d(GF(s))$, $n_d = h_{d-1}$, i.e., the number of points equals the number of hyperplanes. Each hyperplane contains n_{d-1} points, whereas each point is contained in $\phi(d-1, d-2, s)$

---

* (i, j) denotes the *gcd* of integers i and j.

hyperplanes. Now, $\phi(d-1, d-2, s) = \phi(d-1, 0, s) = n_{d-1}$. Thus, hyperplanes as blocks and points as objects form a symmetric block design. Also, $n_d - s n_{d-1} = 1 \Rightarrow (n_d, n_{d-1}) = 1$. This implies that the design is cyclic.

In the particular case of a two-dimensional geometry, i.e., $\mathbb{P}^2(GF(s))$, the lines as blocks and points as objects form a symmetric cyclic block design.

3.3 Interconnection Structure Using a Two-Dimensional Geometry

The number of points in $\mathbb{P}^2(GF(s))$, $n_2 = \phi(2, 0, s) = s^2 + s + 1$. The number of lines in $\mathbb{P}^2(GF(s))$, $h_1 = \phi(2, 1, s) = s^2 + s + 1$. Each line contains $(s+1)$ points, and through any point there are $(s+1)$ lines. Every distinct pair of points determines a line, and every distinct pair of lines intersects in a unique point.

Based on this geometry, we construct the incidence structure as follows. The memory modules are put in one-to-one correspondence with points of $\mathbb{P}^2(GF(s))$, 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.

Consider a binary operation $\circ$, acting on operands a_i and b_j

$$a_i \leftarrow a_i \circ b_j \ .$$

We say that this operation is associated with an index pair (i, j). Suppose $i \neq j$. Let the input operands for this operation reside in M_i and M_j. These modules correspond to two distinct points in $\mathbb{P}^2(GF(s))$. Since two points determine a unique line, this binary operation is assigned to a unique processor. If $i = j$ or if the operation is unary, we have some freedom in choosing the processor to which this operation is assigned.

Such an interconnection scheme allows efficient computation in cases where a large number of operations are associated with just one index pair. For example, it can be used for very fast

matrix-vector multiplication involving sparse matrices with arbitrary on irregular structure. We shall discuss this in greater detail in Section 4.

3.4 Conflict Free Connection Patterns in the Two-Dimensional Geometry

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, $C_t = \{(P_i, M_j) \mid 1 \le i \le p, 1 \le j \le m\}$, where p is the number of processors, and m 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 a processor can be connected to only one memory module at a particular instant of time. Therefore, $|C_t| \le p$. A conflict free connection pattern is one where each processor is connected to a memory module, i.e. $|C_t| = p$.

Represent a line by an ordered $s+1$-tuple $(x^{i_1}, x^{i_2}, \ldots, x^{i_{s+1}})$. By Singer's theorem, shifts of steps i, $0 \le i \le p-1$ of any line lead to all the distinct lines, which correspond to processors in the two-dimensional geometry. The j^{th} elements x^{i_j}, of all the p ordered $s+1$-tuples are distinct, and these correspond to all the memory modules. Thus, $s+1$ conflict free connection patterns can be generated by taking all possible shifts of any line.

We now present an example to illustrate the generation of these conflict free connection patterns. Consider $\mathbb{P}^2(GF(2))$. Let y be a primitive root of $GF(2^3)$. The elements $1, y, y^2, \ldots, y^6$ correspond to the points of $\mathbb{P}^2(GF(2))$. There are 3 points on any line. The 3-tuple $(1, y, y^3)$ forms a line. We list all the lines generated by shifting this line by all possible step sizes:

$$
\begin{array}{l}
(1, y, y^3) \\
(y, y^2, y^4) \\
(y^2, y^3, y^5) \\
(y^3, y^4, y^6) \\
(y^4, y^5, 1) \\
(y^5, y^6, y) \\
(y^6, 1, y^2)
\end{array}
$$

Giving a labeling to the processors and memory modules, we have the following conflict free connection patterns:

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

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

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

Clearly, the number of possible connection patterns is much greater than the conflict free connection patterns. If these conflict free patterns were all the allowable connection patterns of the parallel machine, the hardware and the compiler would be simpler. Simulations show that under certain conditions, such a machine does not lose much in efficiency.

There are many other such conflict free connection patterns, but the simulation results presented later have been compiled only with the conflict free patterns generated by shifts.

We now discuss the compilation of matrix-vector multiplication on the 2-dimensional geometry. Simulations have been done on $\mathbb{P}^2(GF(2))$ and $\mathbb{P}^2(GF(3))$.

4. Applications

4.1 Compilation of Matrix-Vector Multiplication for the Two-Dimensional Geometry

Let A be an $m \times n$ sparse matrix, and $\mathbf{x}$ and $\mathbf{y}$ be column vectors of dimension n and m respectively. The matrix-vector product is computed as follows:

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

Corresponding to a non-zero element a_{ij} there is a multiply-and-accumulate operation:

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

Since each operation is associated with just one index pair, the 2-dimensional geometry is the

most appropriate. The elements y_i and x_j are assigned to memory modules M_μ and M_ν respectively, by hashing functions f and g, such that $\mu = f(i)$ and $\nu = g(j)$ (the hashing functions f and g need not be distinct). If $M_\mu \neq M_\nu$, the processor to perform this operation is determined uniquely – since there is a unique line which passes through the pair of points corresponding to M_μ and M_ν. The matrix entry a_{ij} is stored in the local memory of this processor. The input operands x_j and y_i are sent to the processor through the interconnection network. The processor performs this operation, and writes the result back onto module M_μ. If $M_\mu = M_\nu$, there is some freedom in choosing the processor which performs this operation.

The number of multiply-accumulate operations equals the number of non-zeros in the sparse matrix A. The compiler forms a *ready* list consisting of all such operations. All the multiply operations can be performed in arbitrary order. So also can the accumulations; we just have to ensure data consistency. A processor may be accumulating the product $a_{ij}x_j$ into y_i, but before y_i is written back, another processor may fetch the old value of y_i. Such a data inconsistency would lead to wrong results. To prevent this we must provide a mechanism to ensure data consistency. We observe that the data flow graph for this computation is quite simple. Hence, we do not need to form an explicit data flow graph for matrix vector multiplication.

We now use the conflict free connection patterns to partition the *ready* list. Let y_i reside in memory module M_μ and x_j in M_ν. The multiply-accumulate operation involves fetching, in some order, y_i and x_j, performing the operation and updating the value of y_i. We assume that the order of fetching y_i and x_j is not important. Thus, such an operation can be associated with the ordered triple (μ, ν, μ) (or (ν, μ, μ)). Let a conflict free bucket contain *operation* lists, where all operations from an *operation* list can be performed without any processor or memory conflict. Using the conflict free connection patterns of the machine, we can form $2 \cdot \binom{s+1}{2} = (s+1)s$ conflict free buckets for operations where $\mu \neq \nu$. These buckets contain operations associated

with all the possible $2\binom{s^2+s+1}{2} = (s^2+s+1)(s+1)s$ triples. Thus, an operation associated with the triple (μ, ν, μ), $\mu \neq \nu$, can be put into a unique bucket. The compiler hashes the vector elements onto memory modules, looks at the triple associated with each operation, and puts its into the appropriate bucket. Thus, we can partition the *ready* list into a collection of mutually disjoint conflict free buckets.

Operations involving the diagonal elements of the matrix are associated with only a single index. Hence, there is some freedom of assigning the processor to perform this operation. These operations are put into another list, called the *free* list. Locks are associated with memory addresses to ensure data consistency. Whenever data which is to be modified is fetched by a processor, the corresponding memory address is locked until the data is written back. In case of matrix-vector multiplication, locks need to be associated only with the elements of the vector **y**. Because of the conflict free buckets, the compiler need not examine each operation in order to resolve conflicts. The compiler simply picks up a chunk of operations, one per processor, from a conflict free bucket of the *ready* list, provided the memory addresses corresponding to their input operands are not locked. Thus, the topological properties of this architecture help in speeding up the compiler substantially. The need for the locks could also be avoided by forming the *ready* list in such an order that scheduling operations in that order would never result in data inconsistency.

The hashing function must distribute operations uniformly among each bucket in the *ready* list. In case it does so, computation proceeds predominantly in an SIMD fashion. Thus, if we can choose a good hashing function, there is not much degradation in performance if computation is forced to proceed in an SIMD fashion. Thus, at a particular instant all processors fetch their input operands, or write the output operand or perform the same operation (refer Fig. 4.1).

```
procedure matvec_compiler()
begin
        hash indices onto memory modules;
        form the partitioned ready list, and free list;
loop: while (ready list is not empty)
        begin
                move operations down the arithmetic pipeline lists;
                if (operations have moved onto the write list)
                        turn write bit on;
                if (write bit is on)
                begin
                        turn write bit off;
                        unlock memory addresses of the output operands;
                        append processor, memory and switch programs;
                        go to loop;
                end
                if (a bucket of the ready list is not empty)
                        if (memory addresses of the input operands are not locked)
                        begin
                                lock memory addresses which are to be modified;
                                append processor, memory and switch programs;
                                remove the operations from the ready list;
                                place these operations in the appropriate pipeline list;
                                go to loop;
                        end
                if (the free list is not empty)
                        if (memory addresses of the input operands are not locked)
                        begin
                                lock memory addresses which are to be modified;
                                append processor, memory and switch programs;
                                remove the operations from the free list;
                                place these operations in the appropriate pipeline list;
                        end
        end
end
```

Figure 4.1: A high level description of the compilation of matrix-vector multiplication

4.2 Data Mapping

The vector elements of **x** and **y** are to be mapped onto the global memory modules. A ''good'' hashing function must distribute operations uniformly among each bucket of the *ready list*. The number of times a particular element of vector **y**, say y_i, is used equals the number of non-zeros in row i of the matrix A. Similarly, the number of non-zeros in column i of the matrix A determines the usage of x_i. A row or column with a large number of nonzeros, would result in a large number of elements being mapped onto a particular memory module, and this would lead to a correspondingly increased load among the processors connected to that memory module. In order to prevent the above occurrence, we propose the execution of an enlarged *similar* data-flow graph instead of the original one. We call two data-flow graphs *similar* if their initial input requirements and the final output produced by them are identical. Thus the intermediate execution on *similar* data-flow graphs may differ, though they produce the same output given identical inputs. The enlarged data flow graph has a greater number of intermediate nodes, which allows greater flexibility in mapping the data onto the global memory modules. We now explain the concept of splitting the rows and columns of the matrix A, which allows us to form an enlarged *similar* data flow graph.

Let A be an $m \times n$ matrix, $\mathbf{e}^{(p)}$ be a p-dimensional unit column vector and $D_{i,p}^{(n)}$ be an $n \times n$ diagonal matrix with 1's on the diagonals of row $\left\lceil \frac{in}{p} \right\rceil$ through row $\left\lceil (i+1)\frac{n}{p} \right\rceil - 1$, and zeros elsewhere.

Splitting row $\mathbf{r}_i^T$ into p parts involves writing $y_i = \mathbf{r}_i^T \mathbf{x}$ as $y_i = \mathbf{e}^{(p)T} R\mathbf{x}$ where R is a $p \times n$ matrix whose j^{th} row, $\mathbf{s}_j^T = \left(D_{j,p}^{(n)} \mathbf{r}_i \right)^T$. Hence, splitting of a row into p parts involves calculating partial dot products in p locations, and then accumulating them in one location.

Splitting column $\mathbf{c}_i$ into p parts involves writing $\mathbf{y} = \mathbf{c}_i x_i$ as $\mathbf{y} = C\mathbf{e}^{(p)} x_i$ where C is an

$m \times p$ matrix, whose j^{th} column, $\mathbf{s}_j = D^{(m)}_{j,p}\ \mathbf{c}_i$. Hence, splitting of column c_i involves copying x_i in p locations and then calculating the dot products.

The idea behind splitting rows and columns into p parts, where p is the number of global memory modules, is to map the vector elements of **x** and **y** such that each memory module is accessed nearly the same number of times. Splitting rows and columns enables us to rewrite the original computation $y = Ax$ as $y = RA'C_{\mathbf{x}}$, where A' is the matrix obtained from A by splitting its rows and columns. Thus, we actually execute an enlarged *similar* data-flow graph (the data-flow graph is never formed explicitly). In the simulations, the rows and columns of a matrix, which have a large number of non-zeros, are split only if a simple hashing, as described in the previous section, is not "good," a measure of "goodness" being explained in Section 5.2.

5. Test Problems and Simulation Results

5.1 Purpose of Empirical Testing

In this section, we describe the main guiding principles we have followed in choosing the test problems. The purpose of empirical testing is more than just finding out how a particular idea performs on a set of test examples. One hopes that the testing should enable one to infer inductively what the performance of an algorithm or hardware might be on other problems or models of potential interest which are not in the test set. In order that such extrapolation from test instances actually solved to other unsolved instances can be carried out with a degree of confidence, it becomes necessary that the test instances be "representative" of large classes of applications of current or potential interest. Many scientific and engineering applications involve sparse matrix computations, the most typical being sparse matrix multiplications and solution of linear systems of equations. In this paper, we have discussed the compilation of routines which involve matrix-vector multiplication. Besides the problems which are typically tackled by matrix-vector multiplication, a lot of real-world applications can be formulated in such a way as

to involve matrix-vector multiplications, e.g. fast fourier transforms, wave mechanics, etc. The test problems we have chosen represent a large variety of applications in sparse matrix computations. Some problems, such as solution of partial differential equations on a grid lead to a highly structured matrix whereas others, like the hypergraph covering problem, involve matrices with an arbitrary structure.

For many of the problems tackled, it is possible to develop special purpose computers and algorithms having better performance than the corresponding general implementations, by exploiting the special structure and properties of the problem. In this computational study, our objective is to study the projected performance of the new architecture on a large class of problems.

5.2 Performance Parameters

One multiply-accumulate operation involves fetching of two input operands from the global memory, one from the local memory, and writing the output operand back into the global memory. Let the whole matrix multiplication involving n operations take x machine cycles on the parallel machine. If the machine cycle time of each processor in the system is c msec, the parallel CPU time is $x \cdot c$ msec. Suppose a serial implementation of the matrix-vector multiplication takes 3 cycles per operation. Then the projected serial CPU time is $3nc$ msec, and

$$\text{Efficiency} = \frac{\text{Serial CPU Time}}{\text{Parallel CPU TIME} \times p}$$

where p is the number of processors. The vector elements y_i and x_j are hashed onto the global memory modules. Each multiply-accumulate operation is then put in the appropriate bucket of the *ready* list. The load imbalance factor, I, is a measure of the unevenness in the distribution of the operations among each conflict free bucket of the *ready* list. Let x_i^b be the number of operations corresponding to processor i in bucket b. Let $x_{\max}^b$ and $\overline{x}^b$ be the maximum and mean respectively of all such x_i's in bucket b. Then,

$$I = \frac{\sum_b (x^b_{max} - \bar{x}^b)}{\sum_b \bar{x}^b} = \frac{\sum_b x^b_{max}}{\sum_b \bar{x}_b} - 1 .$$

The load imbalance factor, I, can be thought of as the degradation in efficiency introduced by the unevenness in distribution due to the hashing function in the case where the only allowable connection patterns in the parallel machine are the conflict free ones.

In the simulation results, tabulated in Section 5.4, we use the hashing function $f(i) = \mu = i \bmod m$ where m is the number of global memory modules. This hashing function is applied to the given ordering of rows and columns of the matrix. However, if this leads to a poor load imbalance factor, the same hashing function is applied to the matrix which results after splitting rows and columns [refer Section 4.2]. In our simulation studies, rows and columns having more than 50 non-zeros are split.

Each element of the parallel machine has an instruction sequence which it follows. The instruction sequence to be followed by a processor is just the type of operation it is to perform. Thus, the width of a processor instruction is $\lceil \log_2 n \rceil$ bits, where n is the total number of operations the processor is capable of performing. The instruction sequence for a memory module consists of a list of addresses along with a read/write bit. The width of such an instruction sequence is $\lceil \log_2 m \rceil + 2$ bits where m is the size, in double words, of each memory module. The switch instruction consists of a list of configurations, each configuration specifying the connections between the input and output ports of the switch. In the 2-dimensional geometry, if all connection patterns possible in the machine are allowed, the instruction width for each switch is $\lceil \log_2 (s+1) \rceil$ bits, since $(s+1)$ memory modules are connected to each processor. However, if only conflict free connection patterns are allowed, all the switches need just one global instruction of $\lceil \log_2 (s+1) \rceil$ bits. Hence, there is a saving of $(2s^2+2s+1) \lceil \log_2 (s+1) \rceil$ bits per machine cycle in the latter case.

For matrix vector multiplication on the 2-dimensional geometry, non-zero elements of the matrix are stored in the local memory of the processor, while the vector elements, y_i's and x_j's are stored in the global memory modules which are accessed through the switching network. The size of the instruction programs for the global memory modules, switches and processors, added to the processor local memory and the global memory requirements gives us the total memory for the parallel implementation. The total memory for a serial implementation is a sum of the memory requirements for a sparse representation of the matrix. In a serial implementation, we have assumed the data structure as given in ADL89 for a column-wise representation of the matrix.

5.3 Simulation Assumptions

In the simulations, the following assumptions have been made:

- Each processor has two input bidirectional ports. One of the ports is connected to the processor's local memory, and the other to a switch through which it accesses the global memory modules connected to it (as determined by the geometry).
- A processor can fetch or write data into memory in one machine cycle.
- The execution of any basic arithmetic operation takes a single machine cycle.
- The switch is capable of changing configuration once per machine cycle.
- Selection of which connection to make is made by switches located at both the processor and memory ends. Hence, the total number of switches in the system equals the sum of the processors and memory modules.
- Each processor is assumed to have a clock cycle of 20 ns.
- Each global memory is assumed to have a size of 16K-double words. Hence, size of each memory instruction = 16 bits.

- Each processor is capable of executing 4 basic operations: add, multiply, subtract and nop. Hence, size of each processor instruction = 2 bits.

5.4 Description of Problem Classes

Besides problems which are given as matrix-vector multiplication, there are many problems which can be formulated in that form. In the following sections we discuss such problems which arise in real-world applications, and their formulations as matrix-vector multiplications.

5.4.1 Wave Mechanics

The objective is to calculate the progress of a 2-dimensional surface (acoustic) wave through a set of deflectors [GUS88]. The Wave Equation is

$$c^2 \nabla^2 \phi = \phi_{tt} \tag{5.1}$$

where ϕ and c are functions of special variables. In general, ϕ represents the deviation of the medium from some equilibrium (pressure, height, etc.) and c is the speed of the wave propagation in the medium (assumed to be isotropic). For non-linear waves, c is also a function of the wave state.

A discrete form of (5.1) for a 2-dimensional problem on $[0,1] \times [0,1]$ is:

$$\begin{aligned} & c^2[F(i,j+1) + F(i,j-1) + F(i+1,j) + F(i-1,j) - 4F(i,j)]/h^2 \\ & \quad = [F_{new}(i,j) - 2F(i,j) + F_{old}(i,j)]/(\Delta t)^2 \end{aligned} \tag{5.2}$$

where $F(i,j) = \phi(ih, jh), h = 1/N$.

The above equation can be rearranged as:

$$\begin{aligned} F_{new}(i,j) = {} & k[F(i,j+1) + F(i,j-1) + F(i+1,j) + F(i-1,j)] \\ & + (2-4k)F(i,j) - F_{old}(i,j) \end{aligned} \tag{5.3}$$

where

Grid Size	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
96	9,216	9,216	46,080	46,080	0.0006	0.395	2.765	99.93
192	36,864	36,864	184,320	184,320	0.0002	1.580	11.059	99.98
384	147,456	147,456	737,280	737,280	0.00004	6.320	44.237	99.99

Grid Size	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
96	9,216	9,216	46,080	2.111	0.528	0.264	0	1.125
192	36,864	36,864	184,320	8.439	2.110	1.055	0	4.50
384	147,456	147,456	737,280	33.751	8.438	4.219	0	18.00

Grid Size	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
96	9,216	9,216	46,080	4.027	2.813	42.44
192	36,864	36,864	184,320	16.104	11.25	43.13
384	147,456	147,456	737,280	64.408	45.00	43.13

TABLE 1(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Wave Mechanics Problems.*

---

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Grid Size	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
96	9,216	9,216	46,080	46,080	0.0006	0.395	2.765	99.92
192	36,864	36,864	184,320	184,320	0.0002	1.580	11.059	99.97
384	147,456	147,456	737,280	737,280	0.00004	6.320	44.237	99.99

Grid Size	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
96	9,216	9,216	46,080	2.111	0.038	0.264	0	1.125
192	36,864	36,864	184,320	8.440	0.151	1.055	0	4.50
384	147,456	147,456	737,280	33.751	0.603	4.219	0	18.00

Grid Size	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
96	9,216	9,216	46,080	3.538	2.813	25.77
192	36,864	36,864	184,320	14.145	11.25	25.74
384	147,456	147,456	737,280	56.573	45.00	25.72

TABLE 1(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Wave Mechanics Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

$$k = \frac{(\Delta t)^2 c^2}{h^2} .$$

The boundary conditions are periodic, i.e.

$$F(i,j) = F(i \bmod N,\ j \bmod N) . \tag{5.4}$$

We now formulate the above problem as a matrix-vector multiplication. Let us represent the 2-dimensional array F by a vector $\mathbf{f}$. We do this by making the element $F(i,j)$ correspond to the element $f(i \cdot N + j)$. Then, we can rewrite (5.3) as:

$$\mathbf{f}_{new} = A\mathbf{f} - \mathbf{f}_{old} \tag{5.5}$$

where A is an $N \times N$ square matrix. The diagonal element in row i of A has the value $2-4k$, while the off diagonal elements in row i of A have the value k corresponding to the elements $F(i-1,j), F(i+1,j), F(i,j-1), F(i,j+1)$.

The simulated performances on the operation (5.5) are shown in tables 1(a) and 1(b). Table 1(a) shows the simulation results on a parallel machine with all connection patterns whereas the results tabulated in table 1(b) assume a parallel machine with only conflict free connection patterns [see Section 3.4]. The matrix A has a nice and regular structure, and the efficiency approaches 100% on larger problems. There is not much degradation in performance by limiting the connection patterns whereas the memory overhead is reduced substantially.

5.4.2 The Fast Fourier Transform (FFT)

The Discrete Fourier Transform (DFT) plays an important role in the analysis, design and implementation of DSP algorithms and systems. The DFT computation can be written as:

$$X(k) = \sum_{n=0}^{N-1} a(n) W_N^{kn} , \quad k = 0, 1, \ldots, N-1 . \tag{5.6}$$

where $W_N = e^{-j(2\pi/N)}$

Fundamental Period	Each Matrix Size			Total Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
1,024	1,024	1,024	2,048	20,480	0.008	0.354	2.458	99.06
16,384	16,384	16,384	32,768	458,752	0.0004	7.868	39.322	99.95
65,536	65,536	65,536	131,072	2,097,152	0.0001	35.956	251.658	99.98

Fundamental Period	Each Matrix Size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
1,024	1,024	1,024	2,048	1.893	0.473	0.237	2.50	0.250
16,384	16,384	16,384	32,768	42.020	10.505	5.252	56.00	4.0
65,536	65,536	65,536	131,072	192.028	48.007	24.003	256.00	16.0

Fundamental Period	Each Matrix Size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
1,024	1,024	1,024	2,048	5.353	4.626	15.72
16,384	16,384	16,384	32,768	117.777	102.001	15.47
65,536	65,536	65,536	131,072	536.038	464.001	15.52

TABLE 2(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Fast Fourier Transform.*

---

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Fundamental Period	Each Matrix Size			Total Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
1,024	1,024	1,024	2,048	20,480	0.008	0.354	2.458	99.06
16,384	16,384	16,384	32,768	458,752	0.0004	7.868	39.322	99.95
65,536	65,536	65,536	131,072	2,097,152	0.0001	35.956	251.658	99.98

Fundamental Period	Each Matrix Size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
1,024	1,024	1,024	2,048	1.893	0.034	0.237	2.50	0.250
16,384	16,384	16,384	32,768	42.020	0.750	5.252	56.00	4.0
65,536	65,536	65,536	131,072	192.028	3.429	24.003	256.00	16.0

Fundamental Period	Each Matrix Size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
1,024	1,024	1,024	2,048	4.914	4.626	6.23
16,384	16,384	16,384	32,768	108.022	102.001	5.90
65,536	65,536	65,536	131,072	491.460	464.001	5.92

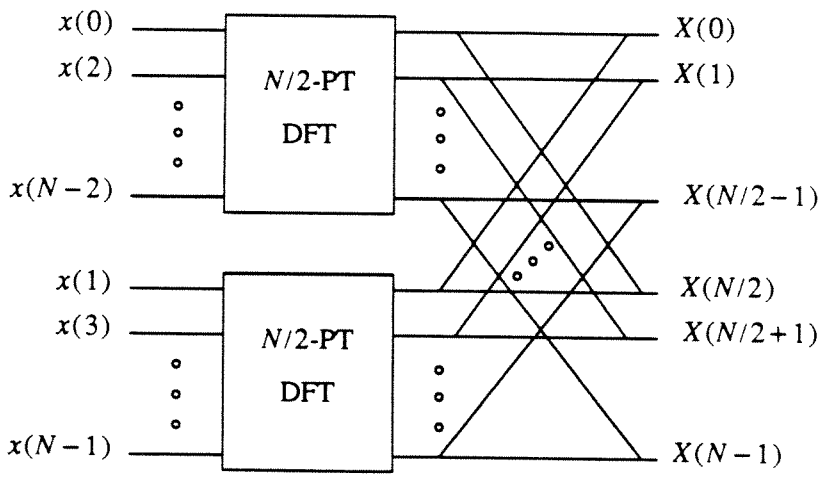
TABLE 2(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Fast Fourier Transforms.*

---

* Simulation Environment.

- Limited patterns allowed.
- 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.
- Clock cycle of each processor assumed to be 20 ns.

The FFT algorithm decomposes the DFT computation into successively smaller DFT computations. When $N = 2^v$, the FFT algorithm enables one to evaluate the DFT in $\log_2 N$ stages, each stage involving $0(N)$ operations.



One can formulate the DFT as a sequence of $\log_2 N$ matrix-vector multiplications, each sequence involving a matrix with 2 non-zeros per row and column.

Tables 2(a) and 2(b) show the simulation results obtained on Fast Fourier transforms. The matrix at each of the $\log_2 N$ stages is very regular, and the efficiency approaches 100% on large problems.

5.4.3 Circuit Simulation

Circuit simulation is essential for fast and reliable design of electronic circuits containing thousands of interconnected components. Circuit simulation of large circuits is a compute intensive problem. In circuit simulation a system of sparse linearized algebraic equations represents the relationship between voltages and currents in the circuit. Circuit simulation performs three important tasks, namely DC, AC and transient analysis. Transient analysis is done repetitively many times during simulation. A standard technique for transient analysis is to use an implicit integration scheme to discretize the differential equations. This results in a system of

non-linear algebraic equations, which are solved using an iterative scheme such as the Newton Raphson method.

Consider a system of n non-linear equations f in n variables x_i. Denoting the vector of variables by $\mathbf{x}$ and the vector of functions by $\mathbf{f}$ we can present the system of equations in compact form:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} . \tag{5.7}$$

Using Taylor series expansion about $\mathbf{x}^*$, assuming that $\mathbf{x}$ is close to $\mathbf{x}^*$ and neglecting higher order terms, we get

$$\mathbf{f}(\mathbf{x}^*) \approx \mathbf{f}(\mathbf{x}) + M(\mathbf{x}^* - \mathbf{x}) \tag{5.8}$$

where

$$M = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & & & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \dfrac{\partial f_n}{\partial x_n} & \cdots & \dfrac{\partial f_n}{\partial x_n} \end{bmatrix}$$

is the Jacobian matrix of the function $\mathbf{f}$.

Setting (5.8) equal to zero and solving, we get some new value for $\mathbf{x}$. Using superscripts to indicate iteration sequence we have

$$\mathbf{f}(\mathbf{x}^k) + M(\mathbf{x}^{k+1} - \mathbf{x}^k) = 0 . \tag{5.9}$$

Defining $\Delta\mathbf{x}^k = \mathbf{x}^{k+1} - \mathbf{x}^k$, we can rewrite (5.9) as

$$M\Delta\mathbf{x}^k = -\mathbf{f}(\mathbf{x}^k) . \tag{5.10}$$

This system of linear equations may be solved by iterative methods, such as conjugate gradient which involve matrix-vector multiplications.

No. of Nodes in Circuit	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
2,750	2,806	2,806	23,470	25,468	0.08	0.229	1.408	87.76
3,750	3,776	3,776	27,590	29,277	0.04	0.259	1.655	91.24
3,350	3,388	3,388	40,545	44,616	0.08	0.397	2.433	87.54

No. of Nodes in Circuit	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
2,750	2,806	2,806	23,470	1.224	0.306	0.153	1.432	0.343
3,750	3,776	3,776	27,590	1.384	0.346	0.173	1.684	0.461
3,350	3,388	3,388	40,545	2.120	0.530	0.265	2.475	0.414

No. of Nodes in Circuit	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
2,750	2,806	2,806	23,470	3.458	2.577	34.21
3,750	3,776	3,776	27,590	4.048	3.102	30.49
3,350	3,388	3,388	40,545	5.804	4.229	37.23

TABLE 3(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Circuit Simulation Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

No. of Nodes in Circuit	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
2,750	2,806	2,806	23,470	25,468	0.08	0.237	1.408	84.90
3,750	3,776	3,776	27,590	29,277	0.04	0.261	1.655	90.46
3,350	3,388	3,388	40,545	44,616	0.08	0.411	2.433	84.57

No. of Nodes in Circuit	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
2,750	2,806	2,806	23,470	1.265	0.023	0.158	1.432	0.343
3,750	3,776	3,776	27,590	1.396	0.025	0.175	1.684	0.461
3,350	3,388	3,388	40,545	2.195	0.039	0.274	2.475	0.414

No. of Nodes in Circuit	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
2,750	2,806	2,806	23,470	3.221	2.577	25.00
3,750	3,776	3,776	27,590	3.740	3.102	20.58
3,350	3,388	3,388	40,545	5.396	4.229	27.60

TABLE 3(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Circuit Simulation Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Tables 3(a) and 3(b) show the results obtained on matrix-vector multiplication arising in Circuit Simulation Problems. For better data mapping, the rows and columns of the matrix are split. The difference in the Operation Count and the number of nonzeros in the matrix is due to the increased number of intermediate operations introduced. Efficiency approaches 90%, and this includes some loss in efficiency due to the intermediate operations. There is a further 2-3% decrease in efficiency on a machine with limited connection patterns.

5.4.4 Problems Arising in Linear Programming

We present simulation results for the matrix-vector multiplication routines which arise in the following linear programming problems:

i) Fractional Hypergraph Covering (FHC)

ii) Minimum Cost Network Flow (MCNF)

iii) Partial Differential Equations with Inequality Constraints (PDE)

iv) Linear Ordering

v) Completely Dense Problems

vi) Control Systems Problems

5.4.5 Fractional Hypergraph Covering (FHC)

A hypergraph $H = (V, A)$ where V is a set of nodes and A is a set of hyperedges. Each hyperedge is a subset of V. Linear programming problems in which the constraint matrix is an incidence matrix of a hypergraph can be used to model many real-world problems. As an example, consider airline scheduling between a collection of cities of V and a collection of round trip flights. Observe that the subset of cities visited by a round-trip flight can be thought as a hyperedge. Associated with this hyperedge is the cost of the round-trip flight. The airline would like to find a set of flights such that each established route between cities is covered by at least

one such flight, and achieving minimum cost at the same time. This is an instance of the hypergraph covering problem. Since this problem can be combinatorially hard, one often solves a fractional relaxation of it.

For our computational experiments, we developed a problem generator, that generates a particular class of hypergraphs, described below.

Consider the tripartite graph $G = (V, E)$ shown in Fig. 5.1. For simplicity, we will assume that each partition has the same number of nodes "n". Let the node sets of the partitions be denoted by V_1, V_2, V_3; $\cup V_i = V$, and let the edges be partitioned into E_1, E_2, and E_3; $\cup E_i = E$. Edges in E_1 have end points in V_1 and V_2; edges in E_2 have end points in V_2 and V_3; edges in E_3 have end points in V_3 and V_1. Define a hyperedge [BER73] as a triplet (i, j, k), $i \in V_1$, $j \in V_2$, $k \in V_3$, such that $(i, j) \in E_1$; $(j, k) \in E_2$; and $(k, i) \in E_3$. Thus (i, j, k) is a triangle consisting of 3 nodes in V_1, V_2 and V_3 and three edges in E_1, E_2 and E_3. Define a hypergraph H [BER73], whose nodes are the nodes of G, and whose edges e_i are the triangles denoted by the triplets (i, j, k) defined before. A sample graph G with $n = 3$ and its corresponding hypergraph H are shown in Fig. 5.2. We define a hypergraph cover as a subset C of hyperedges of H such that if there is an edge (v_1, v_2) in G then there is a hyperedge (v_1, v_2, x) in C; i.e., every edge in G is covered by at least one hyperedge. For example, in Fig. 5.2 the hyperedges (3, 5, 7), (3, 5, 8), (1, 4, 8), (1, 5, 7), and (2, 6, 9) form a cover.

If we associate a cost with each hyperedge, then the hypergraph covering problem is to find the minimum cost hypergraph cover. The mathematical programming formulation of this results in an integer linear program, the relaxation of which gives rise to the FHC problem. Let X_{ijk} be a $0-1$ variable. X_{ijk} is 1 if the hyperedge (i, j, k) is included in the cover; X_{ijk} is 0 otherwise. Associate the cost C_{ijk} with each X_{ijk}. Then the FHC problem becomes:

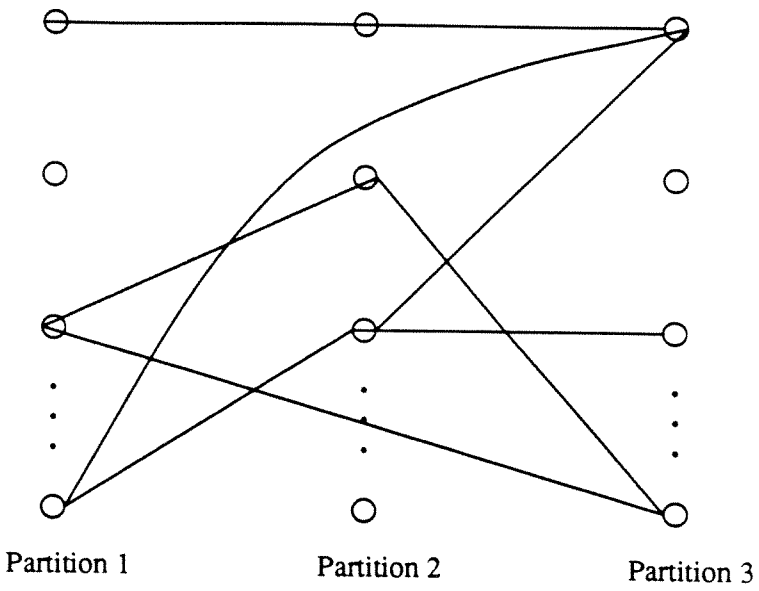


Fig. 5.1. A Tripartite Graph with $3n$ nodes.

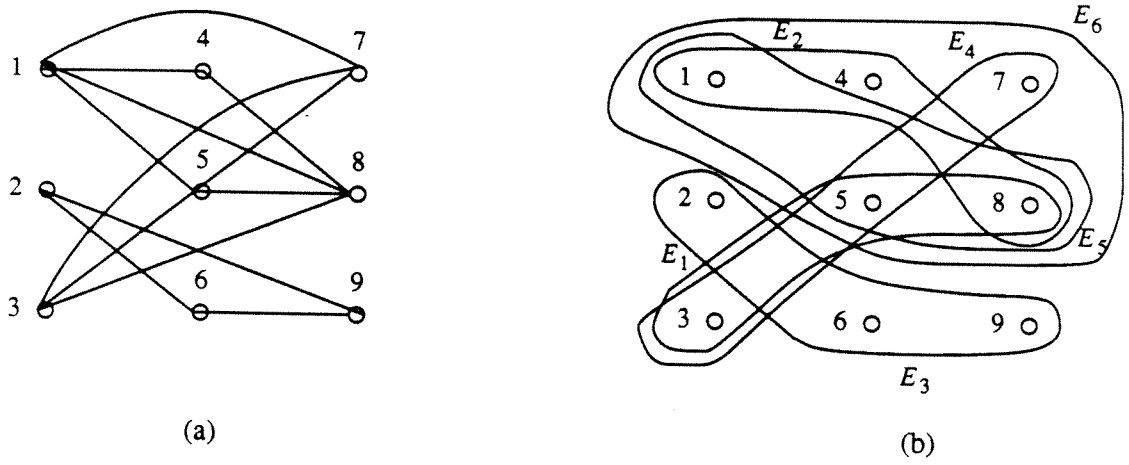


Fig. 5.2. (a) A graph G with 3 nodes in each partition; (b) The corresponding hypergraph H with hyperedges (3, 5, 8), (1, 4, 8), (2, 6, 9), (3, 5, 7), (1, 5, 8), and (1, 5, 7).

No. of Nodes in each partition of Hypergraph	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
30	2,700	8,928	26,784	26,784	0.046	0.237	1.607	97.03
40	4,800	15,917	47,751	47,751	0.037	0.420	2.865	97.35
50	7,500	25,098	75,294	75,294	0.029	0.656	4.518	98.31

No. of Nodes in each partition of Hypergraph	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
30	2,700	8,928	26,784	1.264	0.316	0.158	1.635	0.710
40	4,800	15,917	47,751	2.245	0.561	0.281	2.914	1.264
50	7,500	25,098	75,294	3.506	0.876	0.438	4.596	1.990

No. of Nodes in each partition of Hypergraph	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
30	2,700	8,928	26,784	4.082	3.434	18.85
40	4,800	15,917	47,751	7.266	6.122	18.69
50	7,500	25,098	75,294	11.406	9.649	18.21

TABLE 4(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Fractional Hypergraph Covering Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

No. of Nodes in each partition of Hypergraph	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
30	2,700	8,928	26,784	26,784	0.046	0.240	1.607	95.57
40	4,800	15,917	47,751	47,751	0.037	0.424	2.865	96.45
50	7,500	25,098	75,294	75,294	0.029	0.664	4.518	97.15

No. of Nodes in each partition of Hypergraph	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
30	2,700	8,928	26,784	1.283	0.023	0.160	1.635	0.710
40	4,800	15,917	47,751	2.266	0.040	0.283	2.914	1.264
50	7,500	25,098	75,294	3.548	0.063	0.443	4.596	1.990

No. of Nodes in each partition of Hypergraph	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
30	2,700	8,928	26,784	3.811	3.434	10.96
40	4,800	15,917	47,751	6.769	6.122	10.57
50	7,500	25,098	75,294	10.640	9.649	10.27

TABLE 4(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Fractional Hypergraph Covering Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

$$\min \sum_{(i,j,k)\in H} C_{ijk} X_{ijk} \tag{5.11}$$

subject to

$$\sum_{i} X_{ijk} = 1 , \quad \text{for all } (j,k)\in E_2 \tag{5.12}$$

$$\sum_{j} X_{ijk} = 1 , \quad \text{for all } (k,i)\in E_3 \tag{5.13}$$

$$\sum_{k} X_{ijk} = 1 , \quad \text{for all } (i,j)\in E_1 \tag{5.14}$$

$$0 \le X_{ijk} \le 1 , \quad \text{for all } (i,j,k)\in H . \tag{5.15}$$

The fractional hypergraph covering problems were randomly generated. The problem generator takes three parameters; the number of nodes in each partition of the hypergraph, the probability of generating a hyperedge, and the range for the cost coefficient of the hyperedge. The hypergraph is then randomly generated by sampling the uniform distribution of appropriate range. The FHC problems have a sparse constraint matrix A (only 3 nonzeros per column).

Tables 4(a) and 4(b) show the simulation results for $y = Ax$. The matrix structure is arbitrary. Inspite of that, efficiency is nearly 98% and reduces by 2-3% on a machine with limited connection patterns.

5.4.6 Minimum Cost Network Flow (MCNF)

This class of LP models arises in many applications. A prominent example is the distribution industry, which we use for illustration. We have a directed graph $G = (V, E)$ (representing the distribution network) with a pair of weights (C_{ij}, W_{ij}) for each edge in the graph, representing the unit cost of shipment along that edge, and the carrying capacity of that edge. Nodes (representing warehouses or retail outlets) have supplies and demands for a single commodity. The linear programming problem is to identify the flows along the edges such that all demands are satisfied, no supply and capacity limits are violated, and the cost of shipment is minimized.

If we let X_{ij} denote the amount of the commodity shipped along the edge (i, j), then the linear programming problem is given by

$$\min \sum_{(i,j)\in E} C_{ij} X_{ij} \tag{5.16}$$

subject to

$$\sum_{(i,j)\in E} X_{ij} - \sum_{(j,k)\in E} X_{jk} \leq D_j - S_j ; \quad j \in V \tag{5.17}$$

$$0 \leq X_{ij} \leq W_{ij} ; \quad (i,j) \in E . \tag{5.18}$$

In (5.17) D_j and S_j denote the demand and supply for the commodity at node j. V and E represent the vertex and edge sets of the directed graph.

Our network flow problem generator formulates the LP problem on square grid graphs of a given size. A sample grid graph of size 4 is shown in Fig. 5.3. In the figure, the node denoted by S is the source node. The node denoted by D is the destination node. The nodes in the grid are transhipment nodes (no supplies or demands). The directed edges from each node are oriented to the right, and down.

The capacities of the edges and the costs associated with them are randomly generated integers (sampled from a uniform distribution, the range of which is user-specifiable). The supply of node S is determined by first solving a maximum flow problem (another LP), and then assigning that maximum flow as the supply and demand of nodes S and D, respectively. Thus, the network flow generator generates a ''maximum-flow, minimum-cost'' network flow problem. The constraint matrix, A of an $n \times n$ grid has $n^2 + 1$ rows (vertices) $2n^2$ columns (edges) and $(4n^2 - n)$ non-zeros, with each edge except those with an end point at node D (the demand row being deleted since it is linearly dependent), being associated with 2 non-zeros.

Grid Graph size	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time	Serial CPU time	Efficiency (%)
	rows	cols	nonzeros					
100	10,001	20,000	39,900	39,900	0.023	0.347	2.394	98.54
200	40,001	80,000	159,800	159,800	0.011	1.382	9.588	99.13

Grid Graph size	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
100	10,001	20,000	39,900	1.853	0.463	0.232	2.435	1.831
200	40,000	80,000	159,800	7.379	1.845	0.922	9.753	7.324

Grid Graph size	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
100	10,001	20,000	39,900	6.815	6.094	11.82
200	40,001	80,000	159,800	27.224	24.396	11.59

TABLE 5(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Minimum Cost Network Flow Problems.*

---

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Grid Graph size	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time	Serial CPU time	Efficiency (%)
	rows	cols	nonzeros					
100	10,001	20,000	39,900	39,900	0.023	0.350	2.394	97.78
200	40,001	80,000	159,800	159,800	0.011	1.385	9.588	98.88

Grid Graph size	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
100	10,001	20,000	39,900	1.868	0.033	0.233	2.435	1.831
200	40,000	80,000	159,800	7.398	0.132	0.925	9.753	7.324

Grid Graph size	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
100	10,001	20,000	39,900	6.401	6.094	5.03
200	40,001	80,000	159,800	25.532	24.396	4.66

TABLE 5(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Minimum Cost Network Flow Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

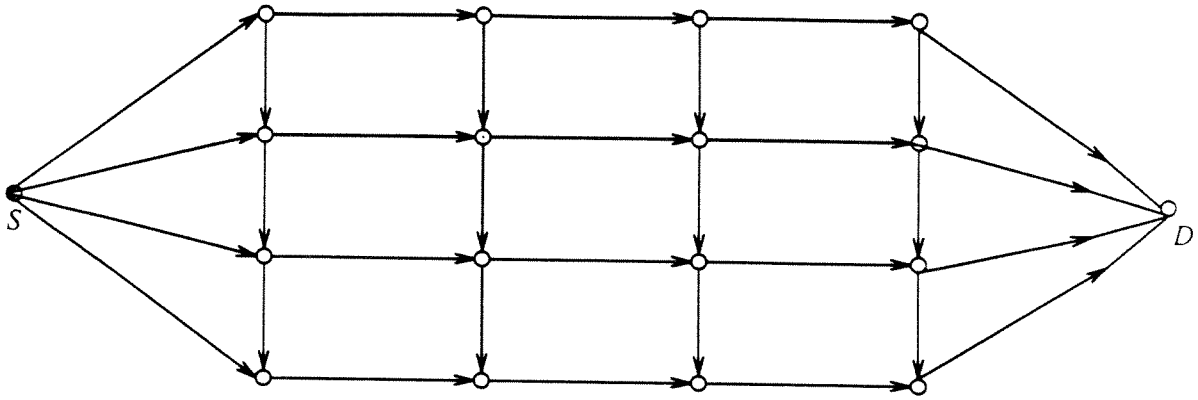


Fig. 5.3. Square Grid Graph of Size 4.

Tables 5(a) and 5(b) show the simulation results on the network flow problems. The matrix has a very regular structure, and efficiency is nearly 99%.

5.4.7 Partial Differential Equations with Inequality Constraints

Many engineering design problems like aircraft wing design, thermal dissipation in VLSI circuits, etc. involve solving partial differential equations with inequality constraints. We study a particular class of this problem, in which we are given a function $\tilde{f}(x, y)$, and we want to find $f(x, y)$ which satisfies the Laplace equation and is as close to $\tilde{f}(x, y)$ as possible in the L_∞ norm. This LP formulation has two sets of constraints: the discretized version of the PDEs, and a set of inequality constraints describing the *closeness* of $f(x, y)$ to $\tilde{f}(x, y)$. These second set of constraints can also be viewed as *smoothing* constraints commonly encountered in adaptive least squares [FRO86]; i.e., $\tilde{f}(x, y)$ has embedded in it some *noise* in the data of the PDEs, and $f(x, y)$ smooths out the noise.

To model these classes of LP problems, we studied the Laplace system of equations

$$\nabla^2 f(x, y) = 0 \tag{5.19}$$

on a square region $\mathcal{S}$. We assume that $f(x, y)$ is periodic, with the period [illegible] equal to the

size of the square.

To numerically solve the Laplace equations, we discretize $\mathcal{R}^2$ into a grid. The south-west corner of $\mathcal{S}$ is translated to the origin $(0, 0)$ of $\mathcal{R}^2$. Let any grid point in $\mathcal{R}^2$ be labelled (x_i, y_j) $-\infty \le i,j \le +\infty$. The grid points of the square region $\mathcal{S}$ are labelled (x_i, y_j) $0 \le i,j \le n-1$, with the understanding that $x_i = ih$, $y_j = jh$, where h is the uniform spacing. The periodicity of $f(x, y)$ in $\mathcal{R}^2$ implies that

$$f(x_i, y_j) = f(x_i \bmod n,\ y_j \bmod n)\ ;\quad -\infty \le i \le +\infty\ ;\quad -\infty \le j \le +\infty\ . \tag{5.20}$$

For convenience, let us define

$$f_{ij} \triangleq f(x_i, y_j)\ ;\quad -\infty \le i \le +\infty\ ;\quad -\infty \le j \le +\infty\ . \tag{5.21}$$

The discretized version of the Laplacian uses the second order finite difference formulas [HIL74]. Introducing the variables b_{ij}, the LP problem for finding a function f_{ij} that is close to a given $\tilde{f}_{ij}$ is given by

$$\min \sum_{0 \le i,j \le n-1} b_{ij} \tag{5.22}$$

subject to

$$-f_{i-1,j} - f_{i,j-1} + 4f_{ij} - f_{i,j+1} - f_{i+1,j} = 0\ ;\quad 0 \le i \le n-1\ ;\quad 0 \le j \le n-1 \tag{5.23}$$

$$|\tilde{f}_{ij} - f_{ij}| \le b_{ij}\ ;\quad 0 \le i \le n-1\ ;\quad 0 \le j \le n-1\ . \tag{5.24}$$

In (5.22)-(5.24) the variables of the LP problem f_{ij} and b_{ij} are unconstrained.

After a series of transformations the primal problem generated by our matrix generator has the following form:

$$\min \quad \sum 2\tilde{f}_{ij}\, x'_{ij} - \sum \tilde{f}_{ij} \tag{5.25}$$

subject to

Grid Size	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
100	10,000	20,000	60,000	60,000	0.001	0.515	3.60	99.92
200	40,000	80,000	240,000	240,000	0.0001	2.057	14.40	99.98

Grid Size	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
100	10,000	20,000	60,000	2.749	0.687	0.344	3.662	1.831
200	40,000	80,000	240,000	10.988	2.747	1.374	14.648	7.324

Grid Size	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
100	10,000	20,000	60,000	9.273	7.935	16.86
200	40,000	80,000	240,000	37.082	31.738	16.84

TABLE 6(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Partial Differential Equations.*

---

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Grid Size	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
100	10,000	20,000	60,000	60,000	0.001	0.515	3.60	99.92
200	40,000	80,000	240,000	240,000	0.0001	2.057	14.40	99.98

Grid Size	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
100	10,000	20,000	60,000	2.749	0.049	0.344	3.662	1.831
200	40,000	80,000	240,000	10.988	0.196	1.374	14.648	7.324

Grid Size	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
100	10,000	20,000	60,000	8.635	7.935	8.82
200	40,000	80,000	240,000	34.531	31.738	8.80

TABLE 6(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Partial Differential Equations.*

* Simulation Environment.
  - 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.
  - Clock cycle of each processor assumed to be 20 ns.

$$- x_{i-1,j} - x_{i,j-1} + 4x_{ij} - x_{i,j+1} - x_{i+1,j} + 2x'_{ij} = 1; \quad 0 \le i \le n-1; \quad 0 \le j \le n-1 \quad (5.26)$$

$$x_{ij} \ge 0; \quad 0 \le x'_{ij} \le 1; \quad 0 \le i \le n-1; \quad 0 \le j \le n-1. \quad (5.27)$$

Thus, the constraint matrix of an $n \times n$ grid has n^2 rows, $2n^2$ columns and $6n^2$ non-zeros with 6 non-zeros per row.

Tables 6(a) and 6(b) show simulation results on the partial differential equations. The matrices arising in such problems have a regular structure, and efficiency approaches 100%.

5.4.8 Linear Ordering (LNORD)

The linear ordering problem is related to the optimal triangulation problem [GRO84], which can be stated as follows: we have a square matrix A of size n whose coefficients are in $\mathcal{R}$. The problem is to find a simultaneous permutation of rows and columns of A such that the sum of strictly upper triangular coefficients of the permuted matrix is as large as possible. The optimal triangulation problem is NP-hard [GAR79]. This problem also has many applications in econometric modelling.

The linear ordering problem can be modelled as an integer linear program: Let $A = [a_{ij}]$ be an $n \times n$ matrix. We introduce integer variables x_{ij}, $i < j$, having the following interpretation: If in the linear ordering being sought, node i comes before node j, then $x_{ij} = 1$; it is 0 otherwise. The LP relaxation of the linear program is given by

$$\text{maximize} \quad \sum_{1 \le i < j \le n} (a_{ij} - a_{ji}) x_{ij} + \sum_{1 \le i < j \le n} a_{ji} \quad (5.28)$$

subject to

Size of the Triangulation Matrix	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
50	1,225	41,650	120,050	128,624	0.001	1.103	7.203	93.32
75	2,775	140,600	410,700	430,124	0.040	3.830	24.642	91.92
100	4,950	333,300	980,100	1,014,749	0.02	8.911	58.806	94.28

Size of the Triangulation Matrix	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
50	1,225	41,650	120,050	5.889	1.472	0.736	7.327	2.617
75	2,775	140,600	410,700	20.453	5.113	2.557	25.067	8.751
100	4,950	333,300	980,100	47.588	11.897	5.948	59.821	20.645

Size of the Triangulation Matrix	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
50	1,225	41,650	120,050	18.041	14.879	21.25
75	2,775	140,600	410,700	61.941	50.640	22.31
100	4,950	333,300	980,100	145.890	120.547	21.03

TABLE 7(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Linear Ordering Problems.*

---

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Size of the Triangulation Matrix	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
50	1,225	41,650	120,050	128,624	0.0001	1.103	7.203	93.32
75	2,775	140,600	410,700	430,124	0.04	3.830	24.642	91.91
100	4,950	333,300	980,100	1,014,749	0.02	8.919	58.806	94.19

Size of the Triangulation Matrix	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
50	1,225	41,650	120,050	5.889	0.105	0.736	7.327	2.617
75	2,775	140,600	410,700	20.454	0.365	2.557	25.067	8.751
100	4,950	333,300	980,100	47.631	0.850	5.954	59.821	20.645

Size of the Triangulation Matrix	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
50	1,225	41,650	120,050	16.674	14.879	12.07
75	2,775	140,600	410,700	57.194	50.640	12.94
100	4,950	333,300	980,100	145.959	120.547	11.91

TABLE 7(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Linear Ordering Problems.*

* Simulation Environment.
  - 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.
  - Clock cycle of each processor assumed to be 20 ns.

$$x_{ij} + x_{jk} - x_{ik} \leq 1; \qquad 1 \leq i < j < k \leq n \tag{5.29}$$

$$(1-x_{ij}) + x_{ik} + (1-x_{jk}) \leq 2; \qquad 1 \leq i < j < k \leq n \tag{5.30}$$

$$-x_{ij} \leq 0; \qquad 1 \leq i < j \leq n \tag{5.31}$$

and

$$x_{ij} \leq 1; \qquad 1 \leq i < j \leq n \tag{5.32}$$

The dual of this problem is in the standard primal form. The constraint matrix of the primal problem has $\frac{n(n-1)(n-2)}{3} + n(n-1)$ columns and $\frac{n(n-1)}{2}$ rows with $2(n-1)$ non-zeros in each row.

Tables 7(a) and 7(b) show simulation results on linear ordering problems. All the rows, each of which have $2(n-1)$ nonzeros are split. The load imbalance factor is quite low, and the loss in efficiency is due to the intermediate operations introduced.

5.4.9 Completely Dense Problems

The completely dense problems are Kuhn-Quandt type random problems having a $m \times 2m$ constraint matrix A [KUH63]. These problems are of the form

$$\min \quad \mathbf{c}^T\mathbf{x} \tag{5.33}$$

subject to

$$A\mathbf{x} = \mathbf{b} \tag{5.34}$$

$$\mathbf{x} \geq 0\,. \tag{5.35}$$

The coefficients of the vector $\mathbf{c}$ and $\mathbf{b}$ are -1 and 10,000, respectively. In addition, the coefficients of A (excluding slack columns) are uniformly distributed integers between 1 and 1000.

Thus, after including the slack variables the constraint matrix is $m \times 3m$ and 66.6% dense; the structure of A is completely amorphous.

Problem Number	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
1	300	900	180,300	180,300	0.007	1.556	10.818	99.34
2	600	1800	720,600	720,600	0.007	6.208	43.236	99.50
3	1000	3000	2,001,000	2,001,000	0.002	17.186	120.060	99.80

Problem Number	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
1	300	900	180,300	8.309	2.077	1.039	11.005	0.073
2	600	1800	720,600	33.152	8.288	4.144	43.982	0.146
3	1000	3000	2,001,000	91.782	22.945	11.473	122.131	0.244

Problem Number	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
1	300	900	180,300	22.502	16.608	35.49
2	600	1800	720,600	89.712	66.174	35.57
3	1000	3000	2,001,000	248.575	183.533	35.44

TABLE 8(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Completely Dense Problems.*

---

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns

Problem Number	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
	rows	cols	nonzeros					
1	300	900	180,300	180,300	0.007	1.556	10.818	99.34
2	600	1800	720,600	720,600	0.007	6.218	43.236	99.34
3	1000	3000	2,001,000	2,001,000	0.002	17.186	120.060	99.80

Problem Number	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
	rows	cols	nonzeros					
1	300	900	180,300	8.309	0.148	1.039	11.005	0.073
2	600	1800	720,600	33.207	0.593	4.151	43.982	0.146
3	1000	3000	2,001,000	91.782	1.639	11.473	122.131	0.244

Problem Number	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
	rows	cols	nonzeros			
1	300	900	180,300	20.573	16.608	23.87
2	600	1800	720,600	82.079	66.174	24.03
3	1000	3000	2,001,000	227.269	183.533	23.83

TABLE 8(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Completely Dense Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Tables 8(a) and 8(b) show simulation results on $y = Ax$ where A is completely dense. Efficiency is very high (99%) and no splitting of rows and columns is required even though each row and column has a large number of non-zeros.

5.4.10 Control Systems

A linear control system is given by

$$\mathbf{x}^{k+1} = A\mathbf{x}^k + B\mathbf{u}^k \qquad 1 \le k \le t \tag{5.36}$$

$\mathbf{x}$ is a vector of state variables of size m and $\mathbf{u}$ is a vector of control variables of size n. The superscript k on $\mathbf{x}$ and $\mathbf{u}$ denotes the discretized time. A is an $m \times m$ state transition matrix and B is an $m \times n$ control matrix. Thus the state of the system at time $(k+1)$ is determined as a linear combination of the current state at time k and the control variables at time k. For the purposes of defining the LP model assume that we are given a desired "*trajectory*" of $\mathbf{x}$ values, say $\mathbf{x}^{*k}$, $1 \le k \le t$. The objective is to determine $\mathbf{u}^k$, $1 \le k \le t-1$ such that small perturbations in $\mathbf{u}^k$ do not alter the resulting $\mathbf{x}^k$, $2 \le k \le t$ too much from the desired trajectory, i.e., the actual control variable values $\bar{\mathbf{u}}^k$ may have small deviations from $\mathbf{u}^k$ when they are applied, because of errors in the instrumentation, resolution of the mechanical or electrical control system, etc. These errors should not cause the state variables $\mathbf{x}^k$ to deviate too much from the desired trajectory.

Formulating robust control systems as LP models, has been studied in [KAR89]. The principal idea behind the robust formulation is to define a simplex around each control point u^k. (For brevity we omit the details of the formulation.) Let the **extreme points** of the simplex around the point $\mathbf{u}^k$ (in control space) be denoted by $\mathbf{v}^{k,1}$, $\mathbf{v}^{k,2}$, ..., $\mathbf{v}^{k,n+1}$. Then the robust control systems formulation is given by:

---

* The control variables $\mathbf{u}^t$ at time t are irrelevant. Similarly, the state variables $\mathbf{x}^1$ (the first time instance) are also irrelevant.

$$\min \varepsilon \tag{5.37}$$

subject to

$$\mathbf{x}^{k+1,j} = A\mathbf{x}^{k,j} + B\mathbf{v}^{k,j}\ ; \qquad 1 \le k \le t-1;\ \ 1 \le j \le n+1 \tag{5.38}$$

$$\mathbf{v}^{k,j} = \mathbf{u}^{k} + \mathbf{c}^{j}\ ; \qquad 1 \le k \le t-1;\ \ 1 \le j \le n+1 \tag{5.39}$$

$$|\mathbf{x}^{k,j} - \mathbf{x}^{*k}| \le \varepsilon\ ; \qquad 2 \le k \le t;\ \ 1 \le j \le n+1 \tag{5.40}$$

$$lb^{k} \le u_i^{k} \le ub^{k}; \qquad 1 \le k \le t-1;\ \ 1 \le i \le m\ . \tag{5.41}$$

Here, $\mathbf{c}^j$ are **constant vectors** (time independent) that specify the size of the simplex. This formulation strives to determine state space variables $\mathbf{x}^{k,j}$, where each $\mathbf{x}^{k,j}$ is a linear recurrence on the simplex vertex j; $1 \le j \le n+1$.

After a series of transformations, this problem is posed naturally in the following dual form:

$$\min \varepsilon \tag{5.42}$$

subject to

$$\left(A^{(k-2)}B\mathbf{u}^{1} + A^{(k-3)}B\mathbf{u}^{2} + \cdots + AB\mathbf{u}^{k-2} + B\mathbf{u}^{k-1}\right)_i - \varepsilon_i^{k}$$

$$\le (-A^{(k-1)}\mathbf{x}^{*1})_i + \min_{1 \le j \le n+1} (x_i^{*k} - y_i^{k,j})\ ;\ \ 1 \le i \le m\ ;\ \ 2 \le k \le t \tag{5.43}$$

$$\left(-A^{(k-2)}B\mathbf{u}^{1} - A^{(k-3)}B\mathbf{u}^{2} - \cdots - AB\mathbf{u}^{k-2} - B\mathbf{u}^{k-1}\right)_i - \varepsilon_i^{k}$$

$$\le (A^{(k-1)}\mathbf{x}^{*1})_i + \min_{1 \le j \le n+1} (y_i^{k,j} - x_i^{*k})\ ;\ \ 1 \le i \le m\ ;\ \ 2 \le k \le t \tag{5.44}$$

$$\varepsilon_i^{k} \le \varepsilon^{k}\ ; \qquad 1 \le i \le m\ ;\ \ 2 \le k \le t \tag{5.45}$$

$$\varepsilon^{k} \le \varepsilon\ ; \qquad 2 \le k \le t \tag{5.46}$$

$$lb^{k} \le u_i^{k} \le ub^{k}\ ; \quad 1 \le k \le t-1\ ;\ \ 1 \le i \le m\ . \tag{5.47}$$

In the above, the superscript in parentheses, $A^{(P)}$, denotes the P^{th} power of the matrix A, and

Control System Size			Problem size			Operation Count in	Hash Imbalance	Parallel CPU	Serial CPU	Efficiency
State Space Size	Control Space Size	Time Steps	rows	cols	nonzeros	Parallel Implementation	Factor	time(msec.)	time(msec.)	(%)
100	15	10	1,045	3,889	139,798	139,798	0.049	1.257	8.388	95.30
100	15	20	2,205	8,209	580,128	580,128	0.020	5.074	34.808	97.99
200	15	20	4,105	15,809	1,159,628	1,159,628	0.022	10.156	69.578	97.87

Control System Size			Problem size			Memory instr.	Switch instr.	Proc instr.	Proc loc mem	Glob mem
State Space Size	Control Space Size	Time Steps	rows	cols	nonzeros	size(Mbit)	size(Mbit)	size(Mbit)	size(Mbit)	size(Mbit)
100	15	10	1,045	3,889	139,798	6.715	0.120	0.839	8.533	0.301
100	15	20	2,205	8,209	580,128	27.100	0.484	3.388	35.408	0.636
200	15	20	4,105	15,809	1,159,628	54.237	0.968	6.780	70.778	1.215

Control System Size			Problem size			Total Memory for parallel	Total Memory for Serial	Memory Overhead for parallel
State Space Size	Control Space Size	Time Steps	rows	cols	nonzeros	implementation(Mbit)	implementation(Mbit)	implementation(%)
100	15	10	1,045	3,889	139,798	16.508	13.219	24.88
100	15	20	2,205	8,209	580,128	67.016	53.998	24.11
200	15	20	4,105	15,809	1,159,628	133.978	107.865	24.21

TABLE 9(a): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Control System Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

Control System Size			Problem size			Operation Count in	Hash Imbalance	Parallel CPU	Serial CPU	Efficiency
State Space Size	Control Space Size	Time Steps	rows	cols	nonzeros	Parallel Implementation	Factor	time(msec.)	time(msec.)	(%)
100	15	10	1,045	3,889	139,798	139,798	0.049	1.257	8.388	95.30
100	15	20	2,205	8,209	580,128	580,128	0.020	5.074	34.808	97.99
200	15	20	4,105	15,809	1,159,628	1,159,628	0.022	10.156	69.578	97.87

Control System Size			Problem size			Memory instr.	Switch instr.	Proc instr.	Proc loc mem	Glob mem
State Space Size	Control Space Size	Time Steps	rows	cols	nonzeros	size(Mbit)	size(Mbit)	size(Mbit)	size(Mbit)	size(Mbit)
100	15	10	1,045	3,889	139,798	6.715	1.679	0.839	8.533	0.301
100	15	20	2,205	8,209	580,128	27.100	6.775	3.388	35.408	0.636
200	15	20	4,105	15,809	1,159,628	54.237	13.559	6.780	70.778	1.215

Control System Size			Problem size			Total Memory for parallel	Total Memory for Serial	Memory Overhead for parallel
State Space Size	Control Space Size	Time Steps	rows	cols	nonzeros	implementation(Mbit)	implementation(Mbit)	implementation(%)
100	15	10	1,045	3,889	139,798	18.067	13.219	36.67
100	15	20	2,205	8,209	580,128	73.307	53.998	35.76
200	15	20	4,105	15,809	1,159,628	146.569	107.865	35.88

TABLE 9(b): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Control System Problems.*

* Simulation Environment.

- 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.
- Clock cycle of each processor assumed to be 20 ns.

$$\mathbf{y}^{k+1,j} \triangleq A\mathbf{y}^{k,j} + B\mathbf{c}^{j} \; ; \quad 1 \le k \le t-1 \; ; \; 1 \le j \le n+1 \tag{5.48}$$

$$\mathbf{y}^{1,j} = \mathbf{0} \; ; \quad 1 \le j \le n+1 \; . \tag{5.49}$$

The constraint matrix for the primal has $(m+n)(t-1)+t$ rows and $(2n+4m)(t-1)+2t-1$ columns.

Tables 9(a) and 9(b) show simulation results on control system problems. Efficiency approaches 98%, and there is almost no reduction in efficiency on a machine with limited connection patterns.

5.4.11 Optimal Routing in Queueing Networks

Linear programming has always played a vital role in design and analysis of computer and communication systems that involve congestion and queueing. Examples of such applications abound the literature. The LP is usually used in determining some parameters of the systems such as the optimal service rate of the server, buffer sizing, etc. We consider a problem of routing and flow control in a network of parallel processors where the LP is employed in determining optimal routing parameters.

This problem was originally proposed by Bovopoulos and Lazar [BOV85]. The optimization problem they considered was to maximize the expected throughput of a multiprocessor system by determining the state-dependent routing probabilities. We have extended the model of Bovopoulos and Lazar, by introducing "reneging", as explained below. The queueing network model of the system we wish to study is shown in Fig. 5.4.

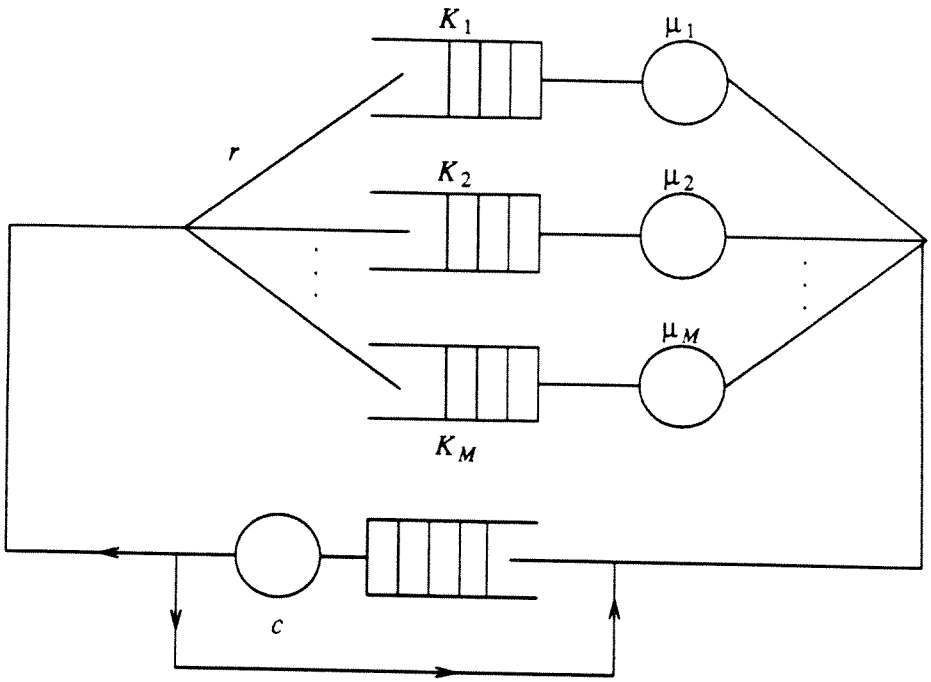


Fig. 5.4. Queueing network model of a multiprocessor.

The M queues model the M processors each with a service rate of μ_i and a finite queue capacity K_i. There is a source with a service rate c that models the generation of packets. c represents the maximum rate at which packets can be transmitted. An arriving packet is routed to one of the processors' queues provided there is space in the finite queue of that processor. This routing decision is made based on the current state of the queues of the processor. If all the queues are full, then the arriving customer recycles back to the generator's queue. After completion of service, the job sends an acknowledgement to the generator which can then decide to transmit one more job to the processors. The problem we are trying to solve is to determine the routing probabilities that maximizes the expected throughput of the processors. In the following discussion we label the generator node with 0. We first define the notation:

E $\triangleq$ state space of the multiprocessor queues; $\{\mathbf{n} = (n_1, n_2, \ldots, n_M) \mid n_i \le K_i\}$

$p(\mathbf{n})$ $\triangleq$ steady state probability that the system is in state $\mathbf{n}$

$r_{\mathbf{n}}^{0i}$ $\triangleq$ routing probability that a job leaving the generator joins processor i's queue, while the state of the system is $\mathbf{n}$;

$$\sum_{0 \le i \le M} r_{\mathbf{n}}^{0i} = 1 . \tag{5.50}$$

The optimization problem then becomes

$$\max \sum_{\mathbf{n}} p(\mathbf{n}) \sum_{1 \le j \le M} \mu_j \, 1(n_j > 0) \tag{5.51}$$

subject to

a. Normalization of state probabilities

$$\sum_{\mathbf{n} \in E} p(\mathbf{n}) = 1 . \tag{5.52}$$

b. Normalization of routing probabilities

$$\sum_{0 \le j \le M} r_{\mathbf{n}}^{0j} = 1 \qquad \mathbf{n} \in E . \tag{5.53}$$

c. Global balance, equation for the Markov chain

$$p(\mathbf{n}) \left\{ \sum_{1 \le j \le M} cr_{\mathbf{n}}^{0,j} 1(n_j < k_j) + \sum_{1 \le j \le M} \mu_j 1(n_j > 0) \right\} = \sum_j p(\mathbf{n} - e_j) cr_{\mathbf{n}-\mathbf{e}_j}^{(0,j)} 1(0 < n_j \le K_j)$$

$$+ \sum_{1 \le j \le M} p(\mathbf{n} + \mathbf{e}_j) \mu_j \, 1(n_j + 1 \le K_j) . \tag{5.54}$$

In (5.51)-(5.54) the notation $1(\textit{relational expression})$ evaluates to a 1 if the relational expression is true; it evaluates to a zero otherwise. Bovopoulos and Lazar introduced an additional constraint to (5.50)-(5.54) which demanded that the response time of a packet be less than a specified amount.

This time delay constraint can be written as (using Little's Law):

$$\sum_{\mathbf{n}\in E}\left\{p(\mathbf{n})\sum_{1\le j\le M} n_j\right\} - T\sum_{\mathbf{n}\in E}\left\{p(\mathbf{n})\sum_{1\le j\le M}\mu_j\, 1(n_j>0)\right\}\le 0 \qquad (5.55)$$

where T is the specified time delay requirement.

The decision variables of the above problem are $p(\mathbf{n})$ and $r_{\mathbf{n}}^{0,j}$, $0\le j\le M$; $\mathbf{n}\in E$.

Reneging. The model described above does not capture one aspect of real systems. In distributed processing, each processor has the expertise to serve specific jobs. Thus the optimal routing policy may conflict with the requirement that job can only be served by a specific processor. Thus, when a job is routed to processor i based on the optimal routing policy, the job may have to "renege" to processor j where it can be served. Let b_{ij} be the state-independent reneging rate of jobs from processor i's queue to processor j's queue. Then the global balance equation (5.54) becomes

$$p(\mathbf{n})\left\{\sum_{1\le j\le M} cr_{\mathbf{n}}^{0,j}1(n_j<K_j) + \sum_{1\le j\le M}\mu_j 1(n_j>0) + \sum_{\substack{1\le i\le M\\ 1\le j\le M\\ i\ne j}} b_{ij}\right\}$$

$$= \sum_{1\le j\le M} p(\mathbf{n}-\mathbf{e}_j)\, cr_{\mathbf{n}-\mathbf{e}_j}^{0,j}\, 1(0<n_j\le K_j)$$

$$+ \sum_{1\le j\le M} p(\mathbf{n}+\mathbf{e}_j)\mu_j\, 1(n_j+1\le K_j)$$

$$+ \sum_{\substack{1\le i\le M\\ 1\le j\le M\\ i\ne j}} p(\mathbf{n}+\mathbf{e}_i-\mathbf{e}_j)\, b_{ij}\, 1(n_i+1\le K_i \text{ and } n_j-1\ge 0)\ . \qquad (5.56)$$

Observe that Eqn. (5.56) is a nonlinear equation since the decision variables $p(\mathbf{n})$ and $r_{\mathbf{n}}^{0,j}$ occur in a product form. To convert (5.56) to a linear equation we introduce, following the methodology of Bovopoulos and Lazar, the transformation

$$x(\mathbf{n}, \mathbf{n}) \triangleq p(\mathbf{n}) r_{\mathbf{n}}^{00} \tag{5.57}$$

$$x(\mathbf{n}, \mathbf{n}+\mathbf{e}_j) \triangleq p(\mathbf{n}) r_{\mathbf{n}}^{0j}\ 1(n_j < K_j) \tag{5.58}$$

$$x(\mathbf{n}, \mathbf{n}-\mathbf{e}_j) \triangleq p(\mathbf{n}) 1(n_j > 0)\ . \tag{5.59}$$

Now the problem (5.51), (5.52), (5.55) and (5.56) can be written in terms of $x(\mathbf{n}, \mathbf{n})$ variables, thus converting the problem into a linear programming problem. We give the final LP problem in the primal form below, omitting the details of the transformations.

$$\min \sum_{\mathbf{n} \in E} \left[\left\{ x(\mathbf{n}, \mathbf{n}) + \sum_{1 \le j \le M} x(\mathbf{n}, \mathbf{n}+\mathbf{e}_j) \right\} \sum_{1 \le j \le M} \mu_j\ 1(n_j > 0) \right] \tag{5.60}$$

subject to

$$\sum_{\mathbf{n} \in E} \left\{ x(\mathbf{n}, \mathbf{n}) + x(\mathbf{n}, \mathbf{n}+\mathbf{e}_j) \right\} = 1 \tag{5.61}$$

$$\begin{aligned}
&\left[\sum_{1 \le j \le M} cx(\mathbf{n}, \mathbf{n}+\mathbf{e}_j) 1(n_j < K_j) \right] + \left[\left\{ x(\mathbf{n},\mathbf{n}) + \sum_{1 \le j \le M} x(\mathbf{n},\mathbf{n}+\mathbf{e}_j) \right\} \sum_{1 \le j \le M} \mu_j\ 1(n_j > 0) \right] \\
&\quad + \left[\left\{ x(\mathbf{n},\mathbf{n}) + \sum_{1 \le l \le M} x(\mathbf{n}, \mathbf{n}+\mathbf{e}_l) \right\} \sum_{\substack{1 \le i \le M \\ 1 \le j \le M \\ i \ne j}} b_{ij} \right] \\
&\quad - \left[\sum_{1 \le j \le M} cx(\mathbf{n}-\mathbf{e}_j, \mathbf{n}) 1(0 < n_j \le K_j) \right] - \left[\sum_{1 \le j \le M} \left[\left\{ x(\mathbf{n}+\mathbf{e}_j, \mathbf{n}+\mathbf{e}_j) \right. \right. \right. \\
&\qquad \left. \left. \left. + \sum_{1 \le l \le M} x(\mathbf{n}+\mathbf{e}_j, \mathbf{n}+\mathbf{e}_j+\mathbf{e}_l) \right\} \mu_j\ 1(\mathbf{n}_{j+1} \le K_j) \right] \right] \\
&\quad - \left[\sum_{\substack{1 \le i \le M \\ 1 \le j \le M \\ i \ne j}} \left[\left\{ x(\mathbf{n}+\mathbf{e}_i-\mathbf{e}_j, \mathbf{n}+\mathbf{e}_i-\mathbf{e}_j) \right. \right. \right. \\
&\qquad \left. \left. \left. + \sum_{1 \le l \le M} x(\mathbf{n}+\mathbf{e}_i-\mathbf{e}_j+\mathbf{e}_l) \right\} b_{ij}\ 1(n_i+1 \le K_i \text{ and } n_j-1 \ge 0) \right] \right] = 0
\end{aligned} \tag{5.62}$$

No. of Nodes	Buffer Capacity	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
		rows	cols	nonzeros					
3	10	2,664	9,924	56,547	57,891	0.04	0.509	3.393	95.22
4	6	4,804	21,268	162,367	175,411	0.07	1.578	9.742	88.19
5	4	6,252	31,252	299,127	321,782	0.01	2.771	17.948	92.54

No. of Nodes	Buffer Capacity	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
		rows	cols	nonzeros					
3	10	2,664	9,924	56,547	2.718	0.680	0.340	3.451	0.768
4	6	4,804	13,102	162,367	8.428	2.107	1.053	9.910	1.591
5	4	6,252	31,252	299,127	14.798	3.699	1.850	18.257	2.289

No. of Nodes	Buffer Capacity	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
		rows	cols	nonzeros			
3	10	2,664	9,924	56,547	7.958	6.248	27.36
4	6	4,804	13,102	162,367	23.090	17.105	34.99
5	4	6,262	31,252	299,127	40.893	30.629	33.51

TABLE 10(a): Simulated Performance of the Parallel Hardware (with all connection patterns) for Matrix-vector Multiply (y = Ax) on Queueing Problems.*

* Simulation Environment:

- 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.
- Clock cycle of each processor assumed to be 20 ns.

No. of Nodes	Buffer Capacity	Problem size			Operation Count in Parallel Implementation	Load Imbalance Factor	Parallel CPU time(msec.)	Serial CPU time(msec.)	Efficiency (%)
		rows	cols	nonzeros					
3	10	2,664	9,924	56,547	57,891	0.04	0.515	3.393	94.19
4	6	4,804	21,268	162,367	175,411	0.07	1.606	9.742	86.66
5	4	6,252	31,252	299,127	321,782	0.01	2.785	17.948	92.07

No. of Nodes	Buffer Capacity Capacity	Problem size			Memory instr. size(Mbit)	Switch instr. size(Mbit)	Proc instr. size(Mbit)	Proc loc mem size(Mbit)	Glob mem size(Mbit)
		rows	cols	nonzeros					
3	10	2,664	9,924	56,547	2.748	0.049	0.344	3.451	0.768
4	6	4,804	21,268	162,367	8.576	0.153	1.072	9.910	1.591
5	4	6,252	31,252	299,127	14.873	0.266	1.859	18.257	2.289

No. of Nodes	Buffer Capacity	Problem size			Total Memory for parallel implementation(Mbit)	Total Memory for Serial implementation(Mbit)	Memory Overhead for parallel implementation(%)
		rows	cols	nonzeros			
3	10	2,664	9,924	56,547	7.361	6.248	17.80
4	6	4,804	21,268	162,367	21.303	17.105	24.539
5	4	6,252	31,252	299,127	37.543	30.629	22.58

TABLE 10(b): Simulated Performance of the Parallel Hardware (with only conflict free connection patterns) for Matrix-vector Multiply (y = Ax) on Queueing Problems.*

* Simulation Environment.
  - 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.
  - Clock cycle of each processor assumed to be 20 ns.

$$\sum_{\mathbf{n}\in E}\left[\left\{x(\mathbf{n},\mathbf{n})+\sum_{1\le j\le M}x(\mathbf{n},\mathbf{n}+\mathbf{e}_j)\right\}\sum_{1\le l\le M}n_l\right]$$

$$-T\left[\sum_{\mathbf{n}\in E}\left[\left\{x(\mathbf{n},\mathbf{n})+\sum_{1\le j\le M}x(\mathbf{n},\mathbf{n}+\mathbf{e}_j)\right\}\sum_{1\le l\le M}\mu_l 1(n_l>0)\right]\right]\le 0 \quad (5.63)$$

$$x(\mathbf{n},\mathbf{m})\ge 0 \qquad \mathbf{n}\in E,\ \mathbf{m}\in E\,. \quad (5.64)$$

Tables 10(a) and 10(b) show simulation results on queueing problems. Rows have to be split for better data mapping, and the efficiency varies between 86% to 95%.

6. Conclusions

From the extensive simulations done, we observe that the efficiency obtained is uniformly high irrespective of the problem class. 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 fail miserably on problems with arbitrary structure. This is observed in the Perfect Club Benchmarks [PER 90] where most of the ''supercomputers'' perform well on problems with regular structure but give extremely poor performance on problems, such as SPICE for Circuit Simulation, which have arbitrary structure and are difficult to vectorize or parallelize at a course-grain level.

On this hardware, we observe that the efficiency is greater than 90% for most of the problems. There is not much loss in efficiency by restricting the machine to have only conflict free connection patterns. Such a restricted machine simplifies the hardware, speeds up the compiler and the savings in the memory overhead are seen to be about 10-20%. Computation may be allowed to proceed in an SIMD fashion for matrix-vector multiplication without any degradation in performance and would lead to further savings in memory requirements. Simulations have also been carried out on $\mathbb{P}^2(GF(3))$, and similar encouraging results have been obtained on it.

LP Problems	Problem Number	Repetition Count of Data-Flow Graph
Fractional Hypergraph Covering	1	16,098
	2	6,267
	3	7,643
Minimum Cost Network Flow	1	2,299
	2	1,974
Partial Differential Equations	1	1,427
	2	12,572
Linear Ordering	1	4,053
	2	9,547
	3	2,543
Completely Dense Problems	1	659
	2	878
	3	1,179
Control Systems	1	731
	2	1,201
	3	7,084

Table 11: Iterations on Same Data-Flow Graph

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 data-flow graph which justify the initial cost of compilation. Table 11 shows some repetition counts of some of the data flow graphs encountered in typical LP problems. Certain routines, such as an N-point fast fourier transform which are used frequently with the same input data could be compiled 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.

I. S. Dhillon

N. K. Karmarkar

11216
MH-11211 – ISD/NKK/KGR – sam
11212

K. G. Ramakrishnan

Att.
References

REFERENCES

[ADL 89] Adler, I., Karmarkar, N. K., Resende, G. C. R., Veiga, G., *Data Structure and Programming Techniques for the Implementation of Karmarkar's Algorithm*, ORSA Journal on Computing, Vol. 1, No. 2, Spring 1989.

[BER 73] Berge, C., **Graphs and Hypergraphs**, North Holland Publishing Company, New York, 1973.

[BOV 85] Bovopoulos, A. D., and Lazar, A. A., *Optimal Routing and Flow Control of a Network of Parallel Processes with Individual Buffers*, Proc. 23rd Annual Allerton Conference on Comm., Control and Computing, Monticello, Illinois, 1985 (pp. 564-573).

[DHI 89-1] Dhillon, I. S., *A Compiler for Sparse Matrix Computations on New Parallel Architectures*, Conference on Interior Point Methods, Bombay, Jan. 1989.

[DHI 89-2] Dhillon, I. S., *Parallel Architectures for Sparse Matrix Computations*, B. Tech. Project Report, Indian Institute of Technology, Bombay, 1989.

[FRO 86] Frost, O. L., *Adaptive Least Squares Optimization Subject to Linear Equality Constraints*, Ph.D. Thesis, Stanford University, 1970.

[GAR 79] Garey, M. R. and Johnson, D. S., **Computer and Intractability: A guide to the theory of NP-completeness,** Freeman, San Francisco, 1979.

[GRO 84] Grötschel, M., Jünger, M., and Reinelt, G., *Optimal Triangulation of Large Real World Input-Output Matrices*, Statistiche Hefte, 25, 1984, pp. 261-295.

[GUS 88] Gustafson, J. L., Montey G. R. and Benner, R. E., *Development of Parallel Methods for a 1024-Processor Hypercube*, SIAM Journal on Scientific and Statistical Computing, Vol. 9, No. 4, July 1988.

[HAL 86] Hall, Marshall, Jr., **Combinatorial Theory,** Wiley-Intescience Series in Discrete Mathematics, 1986.

[HIL 74] Hildebrand, F. B., **Introduction to Numerical Analysis,** second edition, McGraw-Hill, New York, 1974.

[KAR 88] Karmarkar, N. K., *Parallel Architectures for Sparse Matrix Computations*, B. Tech. Project Proposal Seminar, Bombay, 1988.

[KAR 89] Karmarkar, N. K. and Ramakrishnan, K. G., *Implementation and Computational Results of the Karmarkar Algorithm for Linear Programming, Using an Iterative Method for Computing Projections*, to appear in Mathematical Programming – B, 1991.

[KAR 90] Karmarkar, N. K., *A New Parallel Architecture for Sparse Matrix Computation*, Siam Conference on Discrete Mathematics, Atlanta, June, 1990.

[KUH 63] Kuhn, H. W., and Quandt, R. E., *An Experimental Study of the Simplex Method in Experimental Arithmetic, High Speed Computing, and Mathematics*, Proceedings of Symposia in Applied Mathematics xv_i (N. C. Metropolis, et al., eds.), American Mathematical Society, Providence, Rhode Island, 1963, pp. 107-124.

[PER 90] Berry, M. et al, *The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers*, International Journal Supercomputer Applications, 1989.