

AN ABSTRACT OF THE THESIS OF

Manoj Raisinghani for the degree of Master of Science in Electrical and Computer Engineering presented on April 07, 1994.

Title : Allocation Of SISAL Program Graphs To Processors Using BLAS

Redacted for Privacy

Abstract approved: _____

Dr. Ben Lee

There are a number of well known techniques for extracting parallelism from a given program. They range from hardware implementations, building restructuring compilers or reorganizing of programs so as to specify all the available parallelism. The success rate of any of the known techniques is rather poor over all types of programs. This has pushed the research community to explore new languages and design different architectures to exploit program parallelism.

The principles of dataflow architectures have addressed the problem of exploiting parallelism in systems by executing dataflow graphs. These graphs or programs represent data dependencies among instructions and execution of the graph proceeds in a data-driven manner. That is, an instruction is executed as soon as all its operands are available, without waiting for any *program counter* to sequence its execution, as is the case in conventional von Neumann architectures.

In this thesis, data flow graphs are generated during the intermediate compilation of a functional language called SISAL (Streams and Iterations in a Single Assignment Language). The Intermediate Form (IF1) is a graphical language consisting of multiple acyclic function graphs that represent a given program. Each graph consists of a sequence of nodes and edges. The nodes specify the operation and the edges indicate the dependencies between the nodes. The graphs are further connected to each other by means of implicit dependencies.

The **Automator** package developed in this project, preprocesses these multiple IF1 graphs and translates them into a single connected graph. It converts all implicit dependencies into actual ones. Additionally, complex language constructs like ForAll, loops and if-then-else are treated in special ways together with their nested levels by the

Automator. There is virtually no limit to the number of nested levels that can be translated by this package.

The Automator's prime contribution is in translating real programs written in SISAL into a specified format required by an allocation algorithm called the Balanced Layered Allocation Scheme (BLAS).

BLAS partitions a connected graph into independent tasks and assigns them to processors in a multicomputer system. The problem of program allocation lies in maximizing parallelism while minimizing interprocessor communication costs. Hence, allocation is based on the best choice of communication to execution ratio for each task. BLAS utilizes heuristic rules to find a balance between computation and communication costs in the target system. Here the target architecture is a simulated nCUBE 3E computer, having a hypercube topology.

Simulations show that, BLAS is effective in reducing the overall execution time of a program by considering the communication costs on the execution times. The results will help in understanding - the effects in packing nodes (grain-packing), routing issues in the network and in general, the allocation problem to any processor in a network. In addition, tasks have also been assigned to adjacent processors only, instead of any processor on the hypercube network. The adjacent allocation to processors helps to determine trade-offs required between achieved speed-ups and the time it takes to completely allocate large graphs on compilation.

Allocation Of SISAL Program Graphs To Processors Using BLAS

by

Manoj H. Raisinghani

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed April 07, 1994

Commencement June 1994

Approved:

Redacted for Privacy

Assistant Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented

April 07, 1994

Thesis written and typed by

Manoj H. Raisinghani

Acknowledgment

This thesis is dedicated to my parents whose blessings and confidence in me has inspired my pursuit of knowledge and excellence. I am grateful to my lovely wife-Jasleen, for her unconditional support and patience. She has stood by me at each step of this work.

My special thanks to Dr. Lee for his guidance and valuable advice during my academic life at Oregon State University. Also, special thanks to my committee members : Dr. Herzog, Dr. Lu and Professor James Coakley. Dr. Herzog inspired me to do my best at all times.

Heart filled thanks to my brother-in-law Gopal Karnani and my sister Veenu for being with me and providing positive energy through my stay in Oregon.

I extend gratitude to my sister Pam and her husband Upen for wishing me well in my thesis work.

I am very thankful to my friends Kamal, Lalit, Duggi, Shashank, Shridhar K., Cherag, Anna, Ziggy, Sameer, Hassan, Vincent Freytag and Shawn Savage.

My special thanks to the group at Livermore National Laboratories (California) : John Feo, Patrick Miller and Walter Cedenro for their quick response and help on queries relating to SISAL and the IF1 stages.

I am thankful to Otto Gyga and Tom from the ECE support for being my friends and giving me an opportunity to work with them.

Finally, my regards and appreciation for Gregory Gritton who believed in my ideas for this project and was always available for brain-storming with me on any issue regarding this thesis. His help and feedback proved most valuable to my successful completion of the project.

Table of Contents

CHAPTER 1 — INTRODUCTION	1
1.1 The Potential	1
1.2 Scheduling and Partitioning	2
1.3 Motivation	3
1.4 Thesis Organization.....	4
CHAPTER 2 — ARCHITECTURES and PERFORMANCE ISSUES.....	6
2.1 Overview of Machine Architectures	6
2.1.1 Models.....	6
2.2 Choice of a Distributed Architecture - nCUBE 3E Series	8
2.3 Cache.....	9
2.4 The Interconnect.....	9
2.4.1 Interconnect Parameters	11
2.4.2 Hypercube	12
2.5 Routing Concepts	13
2.6 E-Cube Routing in a 3 dimension Hypercube.....	14
2.7 Communication Latency	16
2.7.1 Store-and-Forward Routing (SFr).....	16
2.7.2 Cut-Through Routing (CTr).....	17
2.8 nCUBE 3E InterConnect.....	18
2.9 Performance Parameters.....	20
2.9.1 Speed Up and Efficiency.....	20
2.9.2 Ideal Speed Up	20
2.9.3 Cost	21
2.9.4 Performance Ratio.....	21
2.10 Parallelism.....	21
2.10.1 Types	22
2.11 Program Granularity.....	23
2.12 Processor Granularity	23
2.13 Utilization.....	24
CHAPTER 3 — A PARALLEL LANGUAGE - SISAL.....	25
3.1 The Data Flow Concept.....	25

3.2	Functional Languages	26
3.2.1	Properties.....	27
3.3	SISAL.....	28
3.3.1	The SISAL Compiler	29
3.4	IF1 Phase	30
3.4.1	Motivation	31
3.5	The IF1 Language	32
3.5.1	Dissection of the matmult.if1 section.....	34
3.6	Nodes.....	34
3.6.1	Simple Nodes of IF1	34
3.6.2	Compound Nodes of IF1	36
3.6.3	Dependencies in a Compound Structure	36
3.7	Edges	37
3.8	Types	38
3.9	Graph Boundaries.....	38
3.10	Structures in IF1 -- An Overview.....	39
3.11	Characteristics of the IF1 Structure.....	41
CHAPTER 4	— THE AUTOMATOR.....	46
4.1	Where does the Automator fit ?	46
4.2	Why is it required ?	46
4.3	Implementation.....	47
4.4	Multiple Instances	49
4.5	Concentric Circles	52
4.6	A Special Case	57
4.7	Node Packing	57
4.8	Execution Costs of IF1 nodes.....	58
4.9	Automator's way with complex constructs	60
4.9.1	ForAll	60
4.9.1.1	Body Count	61
4.9.2	User-Interaction.....	61
4.9.3	Loops - LoopA and LoopB	62
4.9.4	Select.....	63
4.9.5	TagCase.....	65
4.10	The User-Interface.....	65
4.11	Features	65

4.12	MAPPING	68
4.12.1	AScatter	69
4.12.2	Range Generate	69
CHAPTER 5 — BLAS and SIMULATION RESULTS		72
5.1	Static Scheduling	72
5.2	BLAS	72
5.2.1	Objectives of BLAS	73
5.3	Overview	73
5.3.1	Examples of Allocation	74
5.4	Order of the BLAS algorithm	80
5.5	Summary of Changes that were made to the BLAS source code	81
5.6	Modified BLAS	84
5.6.1	Effectiveness of Modified BLAS	85
5.7	Simulations and Results	85
5.8	Programs for the Simulation	86
5.8.1	Matrix Multiplication	86
5.8.2	LU Decomposition	90
5.8.3	Simple	91
5.9	Simulation Results	91
5.10	C/T	102
5.11	Adjacent versus Any	103
CHAPTER 6 — CONCLUSIONS and FUTURE WORK		105
6.1	A Review	105
6.2	Future Study	106
6.3	Improvements	106
6.3	Similar Work	107
BIBLIOGRAPHY		109

Table of Figures

2.1	SIMD and MIMD Architectures	7
2.2	Shared Memory Architectures	10
2.3	Message Passing Architecture (Distributed Memory)	10
2.4	Hypercubes for different 'k' dimensions.....	13
2.5	E-Cube Routing in a 3-dimensional Hypercube network	15
2.6	Comparing Store-and-Forward with Cut-Through routing for a single message	18
3.1	Structure of SISAL's "osc" compiler.....	30
3.2	matmult.sis	32
3.3	Section of the matmult.if1 file (Innermost ForAll)	33
3.4	Innermost Nested Level of matmult.if1	35
3.5	Function Graph for matmult.if1 (Outer Level)	40
3.6	Internal IF1 Structure for the Outer Boundary of matmult.if1 file	42
3.7	The Complete IF1 representation of matmult.if1 file	43
3.8	Internal IF1 Structure for the entire matmult.if1 file	44
4.1	Automator presents Linked lists to BLAS	46
4.2	Steps in the metamorphosis of matmult2.if1.....	53
4.3	Final STEP V in the metamorphosis of matmult2.if1	54
4.4	Concentric Circles	56
4.5	Special Case	59
4.6	Automator's implementation of the IF1 Select Structure	64
4.7	User Interface of the Automator tool	66
4.8	Trace of compaction for the program simple16.opt.....	67
4.9	Mapping of an IF1 Range Generator.....	70
5.1	Example 1 demonstrating BLAS.....	75
5.2	Trace file for Example 1 obtained when executing BLAS.	76
5.3	Trace file indicating the allocation of nodes of matmultif1.i.....	78
5.4	Allocation of matmult2if1.i to PE layers in a 2-dim. Hypercube	79
5.5	An example speed-up file when simulating a 5x5 LU problem.....	82
5.6	Example to demonstrate the Flaw and Correction made in BLAS.	84
5.7	mat 3x3.i.....	88
5.8	A single Body of mat 3x3.i	89
5.9	Matrix Multiplication -- Speed-up curves.....	93

5.10	Matrix Multiplication -- Corresponding Efficiency curves.....	93
5.11	Matrix Multiplication -- Performance Ratio curves.....	94
5.12	LU with Pivot -- Speed-up curves.....	95
5.13	LU with Pivot -- Corresponding Efficiency curves	95
5.14	LU with Pivot -- Performance Ratio curves.....	96
5.15	LU with NO Pivot -- Speed-up curves.....	97
5.16	LU with NO Pivot -- Corresponding Efficiency curves.....	97
5.17	LU with NO Pivot -- Performance Ratio curves.....	98
5.18	Simple -- Speed-up curves	99
5.19	Simple -- Corresponding Efficiency curves.....	99
5.20	Simple -- Performance Ratio curves	100
5.21	Curves comparing 'Adj' with 'Any' allocations.....	104

Table of Tables

Table 4a	Main-Array Organization.....	49
Table 5a	Parameter values from Simulations.....	101
Table 5b	Speed up for the lu nxn at 64 PEs	102
Table 6a	Calculated values for Verification.....	108

Allocation Of SISAL Program Graphs To Processors Using BLAS

CHAPTER 1. INTRODUCTION

1.1 The Potential

Ever increasing demand for computational power has spawned an idea of the *Grand Challenge*. A *Grand Challenge* is a fundamental problem in science or engineering that has a broad economic and scientific impact, and whose solution could be advanced by applying high performance computing techniques and resources. Here, the concepts of parallelism have offered enormous potential for increase in performance.

The primary constraint with technology today is the actual design complexity that is incorporated within a chip. Newer fabrication processes using sub-micron geometry allow for Very Large Scale Integrated Circuits (VLSI) to support more than 3 million transistors on a single microprocessor chip for example, Intel's latest product offering : The *Pentium* processor. These accomplishments are being further improved upon to accommodate nearly twice the number of transistors on a single CMOS package, equipped with faster clocks and novel cooling systems.

There is only so much that can go onto a single chip using available technologies. With advent of the fifth generation microprocessors, based on ultra-large-scale integrated circuits (ULSI) or Very High Speed Integrated Circuits (VHSIC), system architects are confident that, one is closely approaching the limits in terms of attainable higher densities and faster speeds in circuits.

An ironic aspect of the quest for ever-increasing speed is that, most of the transistors are idle nearly all the time. Any modern day computer with one CPU (Central Processing Unit) is dominated by multiple memory chips. The CPU issues sequential requests over a bus to memory that responds to one request at a time. This is the traditional *von Neumann bottleneck*. Any attempts to break this bottleneck assumes a computer with multiple control units and multiple memory modules. Thus, the only plausible alternative is to go parallel. In other words, programs can execute on multiple processors connected in a regular structure and can access multiple memory banks, thus performing computations and memory transfers concurrently.

Opinions are fairly divided in the research community as to the number of processors needed for a cost-effective, efficient and fast system. One group looks at the

practicality of Amadahl's thesis [14, 15 and 16] where-in, systems will face a severe bottleneck on encountering the serial fraction within the application that is being executed. Hence, large sections of serial code will over-ride the need for higher number of processing elements (PEs). Other groups cite simulations of the excellent speed-ups that can be attained using large number of PEs [20 and 25] over parallel algorithms. Also, the commercial world has little experience with efficient use of large number of PEs [25].

Several factors need to be considered when building a supercomputer. Parallel algorithms, well designed parallel architectures coupled with powerful programming environment which include sophisticated restructuring compilers, in all, play vital roles in very fast performance. Crucial issues, like scheduling, synchronization and communication need to be adequately addressed in order to exploit the inherent flexibility of parallel processor systems.

Investment in serial (conventional) software over the years has been tremendous and cannot be discarded completely. The transformation to new parallel versions will take several years and a firm financial commitment. This brings users to agree for the development of restructuring compilers that will take existing serial software written in FORTRAN, Pascal, C or the like, and make it viable for use on the parallel processor systems without any reprogramming. In program restructuring, a compiler or a preprocessor identifies parts of the program that can take advantage of the architectural characteristics of a machine [26]. Coding a program to take full advantage of the system is a tedious task and a compiler will do a better job than any skillful programmer.

However, new software can be developed in a language specifically designed for parallel computation. A number of parallel languages are available including functional and data parallel languages. Finally, serial languages like C, FORTRAN, or Smalltalk can be extended with parallel constructs [22 and 27].

1.2 Scheduling and Partitioning

The fore-most challenge of parallel systems is to maximize the program speed-up (defined in chapter 2 in section 2.8). So, it is imperative to design a good processor allocation or scheduling scheme that minimizes execution time and interprocessor communication for any program. Due to complexity of the scheduling problems, only a few simple cases have been solved optimally in polynomial time [26].

Two types of scheduling strategies - Static and Dynamic scheduling are in focus these days [25].

Static schemes can be accomplished by the user or a compiler and are deterministic in nature. Scheduling is done before execution of the program, based on all *global* program information. A common approach is to compute or predict the execution time of different program segments, and in general, consider parameters that are unknown at compile time and then decide the best scheduling strategy. This exposes the parameters to various approximations in order to achieve optimal performance. Static strategies are usually very complex and time consuming as will be seen with the BLAS (Balanced Layered Allocation Scheme) in [12 and 18] and also in the discussion in chapters 5 and 6. Also, compilation time is as important as the execution or run-time when evaluating performance issues on a particular system.

Dynamic schemes are enabled with help through the operating system (OS) or the hardware and have found their use in real machines [25]. This scheme is usually based on the *local* information about the running program. As the scheduling decisions are made at run time, a penalty is incurred in terms of overheads that may be introduced. Hardware implementation is sometimes done for special architecture systems or for scheduling special types of tasks. This is costly and lacks generality [25]. On the other hand, any operating system implementation has a high overhead cost if the granularity of the task is small. Frequent invocation of the OS will outweigh the benefits of parallelism. Being a run-time approach makes dynamic schemes indeterministic.

Both schemes have their disadvantages with respect to either optimality or time or cost or some combination there-of [25 and 26]. Trade-offs are necessary as the best is yet to be found.

An important ingredient of scheduling is the **partitioning phase** which, by itself is a complex optimization problem. A given parallel program needs to be partitioned or broken down into independent tasks also called processes or threads that will subsequently be allocated to different PEs.

1.3 Motivation

The **aim** of this research is, to develop a platform in order to study the effectiveness of an existing static-allocation scheme (BLAS) on real algorithms written in a functional language called SISAL (Streams and Iterations in a Single Assignment Language). This

requirement led to the development of a *graph preprocessor* package called **The Automator**. It is an interface between the SISAL programs and BLAS.

The BLAS algorithm was originally written by Dr. Lee [18] and further extended by Freytag [12]. It laid the foundation for this thesis and will be explained in chapter 5.

The Automator operates on multiple function graphs produced during intermediate compilation of SISAL programs. It expands the complex language constructs of the intermediate form (IF1), converts all implicit dependencies between the graphs to actual dependencies, does node packing and helps in exposing the existing parallelism. The target of the Automator is a file containing a linked list of nodes in a specified format readable by BLAS. The BLAS algorithm identifies all parallelizable threads in the Automator's output and allocates them to different PEs.

This thesis has required work in three areas : 1. Significant analysis of the IF1 graphical language together with its internal structure and organization, so as to successfully interface with it. 2. The development of the Automator package to preprocess multiple IF1 graphs and prepare them for BLAS, and 3. To verify that, the graphs produced by the Automator worked well with BLAS. In the process of verification, some bugs were noticed in BLAS and important changes were made to certain routines so as to make it work for all types of program graph inputs. The corrections together with changes made to BLAS have been listed in section 5.5.

1.4 Thesis Organization

Chapter 2 starts with an overview of machine architectures that exist today and the choice of using message passing computers in this work. Certain architectural aspects of the system : namely, the routing and interconnect issues that affects performance of the allocation process, are described. The choice of Hypercube topology is explained and some of its properties are highlighted. The parameters that are used to specify the extent of parallelism achieved on a particular architecture are defined. These parameters will be clearer when reading plots in chapter 5. Finally, parallelism issues are discussed and it lays the foundation for chapter 3, 4 and 5.

Chapter 3 describes the need for parallel languages and the influence of the data flow community on the research in exploiting maximum parallelism from algorithms. A note on SISAL is included with the various stages in its compilation. Motivation for using SISAL and its intermediate form is discussed. The IF1 graphical language is described in some detail, in relevance to the development of the Automator. The IF1

representation consists of acyclic graphs of different functions that make up a SISAL program. Knowledge of the language, and its internal data structures gives a better understanding of the Automator's work in preprocessing graphs.

Chapter 4 discusses the Automator package in its entirety and elaborates on all the techniques involved in preprocessing and presenting graphs to the Balanced Layered Allocation Scheme (BLAS) for assignment to different processors. The treatment of the various compound constructs of IF1 are explained. A section on mapping of IF1 nodes is also included. It provides an initial platform with regards to actual mapping of the IF1 constructs on any multicomputer system

Chapter 5 is a brief on the BLAS algorithm and the simulation results obtained with BLAS. The allocation strategy is explained with two examples. Also, included are the limitations of BLAS and the various changes made to it. The programs used for simulations are described in brief. They are Matrix Multiplication, LU-Decomposition and SIMPLE. Speed-up, Efficiency and Performance Ratio (explained in chapter 2) curves are plotted to show the effectiveness of BLAS on the allocation of tasks to PEs.

Chapter 6 concludes this thesis with a brief overview of the work and future areas that need to be explored. A section on the possible improvements in BLAS and Automator is included. A comparison with hand calculations is also shown at the end, to highlight the effectiveness of this work.

CHAPTER 2. ARCHITECTURES and PERFORMANCE ISSUES

2.1 Overview of Machine Architectures

The buzz word *Pipeline* has become synonymous with almost any practical system today. Almost any current processor supports a pipelined instruction execution cycle. The 5 phases : Instruction fetch (IF), Decode, Execute, Memory and WriteBack are pipelined, with logically subsequent instructions going through the different phases of execution simultaneously. Another feature coupled with this is the support of Vector operations for functional units, where-in long vectors of operands are processed at a time. These features form the prime technological trends that give us supercomputers and Massively Parallel Processing (MPP) systems. Pipelined, Array and Vector processor based systems are categorized according to Flynn's taxonomy [16] as Single Instruction Multiple Data Stream (SIMD) because of their single control unit. SIMD implies synchronous systems with a single control unit and all PEs driven by the same clock.

A different organization [16 and 26] is the multiprocessor or parallel processor system. Composed of a set of independent or autonomous PEs, which are fully or partially interconnected in some way, these systems are classified in Flynn's taxonomy as Multiple Instruction stream and Multiple Data streams (MIMD) architectures. They can be synchronous or asynchronous. The control is distributed, with each PE having its own control unit and driven by its own clock. Global control may be used in some hierarchical architectures built around the MIMD philosophy. The SIMD and MIMD architectures are shown in figure 2.1.

2.1.1 Models

Parallel Processor systems further get categorized based on their PE-Memory access. These are *Shared Memory* systems and *Message Passing* systems. All PEs in a Shared Memory system share the same memory address space and are connected to this shared physical memory by a high bandwidth bus or some multistage interconnection network. Inter PE communication is accomplished through shared areas in memory. Examples of shared memory architectures are Cray X-MP, Cray-2, Alliant FX/8 [14, 15, 16 and 25]. Each PE accesses the shared data by reading from or writing to a shared variable in the globally accessible memory area. However, because more than one PE

may try to access the same variable, discrepancies will rise in the final value of the variable.

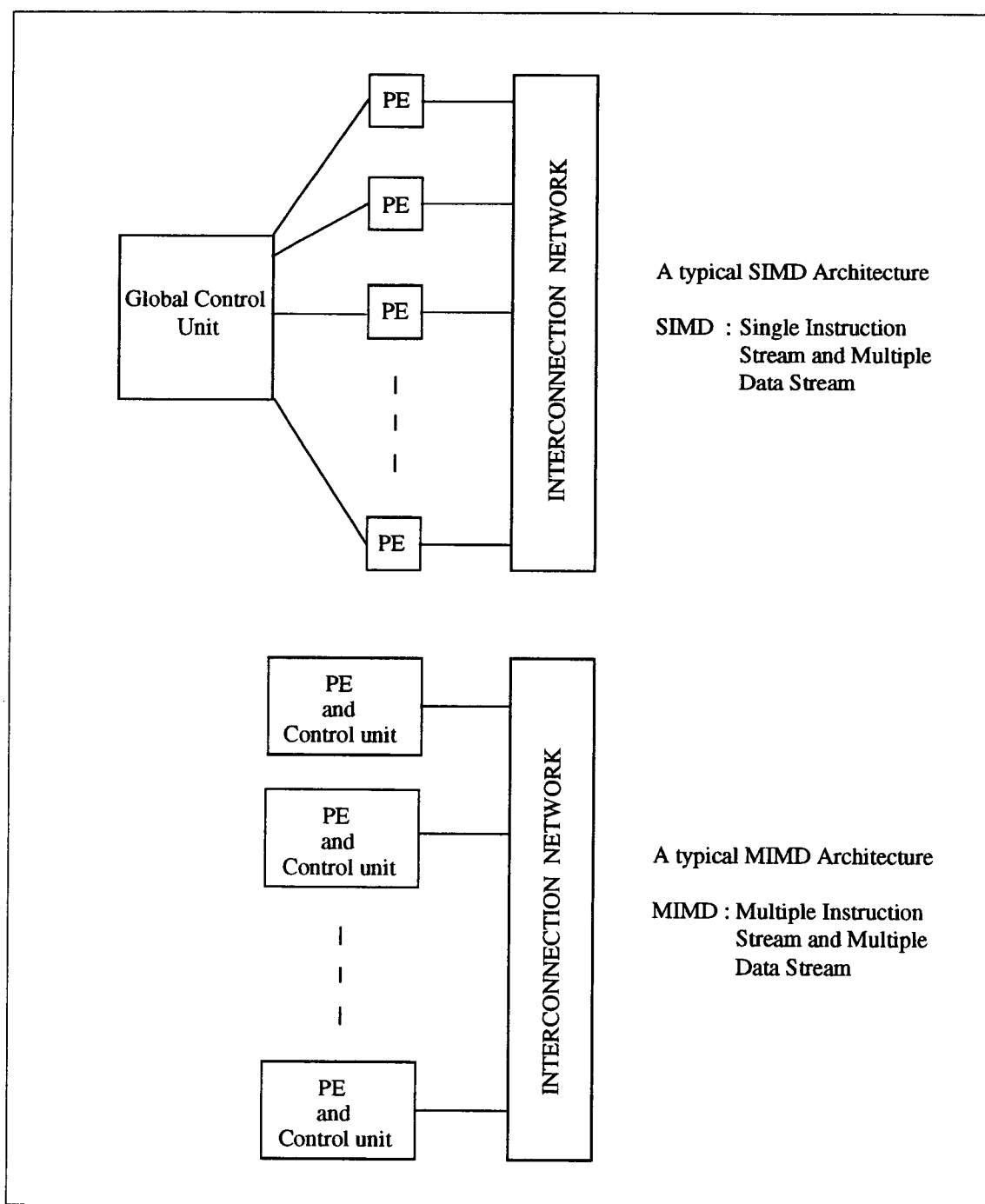


figure 2.1 SIMD and MIMD Architectures

For example, if PE1 is attempting to multiply a shared variable x by 5 and PE2 needs to decrement it, then different results will occur depending on which PE gets hold of that variable first. To circumvent undesirable effects, programming for shared memory systems require low level primitives (instructions) to resolve such mutual-exclusion problems.

In contrast, PEs in a Message Passing organization have their own local memory with no concept of any global shared area. The PEs communicate via messages that are generated and passed using a particular scheme. Usually, these architectures are also described as *multicomputer* systems [14, 15, 16 and 35]. There are variants in this organization where one PE can access the memory of some remote PE. Systems may require a dedicated PE or a host that exercises initialization and global control of the organization. Refer to figure 2.2 and figure 2.3 for elaboration on the classification of systems.

The Message Passing model can work with synchronous or asynchronous communication. In the synchronous paradigm, the sending process and the receiving process must synchronize in time and space for passing a message. If one process is ready to communicate but the other process is not, then blocking or waiting occurs.

Asynchronous communication utilizes buffers for each PE channel to avoid any synchronization conflicts of the processes. The ready process passes the message without waiting and so, makes the asynchronous transfer almost non-blocking. Since channel buffers are finite in size, a sender may eventually be blocked.

Intel's and nCUBE's hypercube based systems are very good examples of message passing architectures using the Hypercube Interconnect. Results for this project were obtained by simulations based on nCUBE's latest system: **nCUBE 3E Series** in which the processing elements communicate synchronously on a hypercube network.

2.2 Choice of a Distributed Architecture - nCUBE 3E Series

Shared memory systems represent a more traditional programming model and extensive research has enabled it to practical realizations. However there are inherent performance issues due to data contention and access time costs to non-local data (applicable to systems with PEs having small local memories and a large global space -refer to figure 2.2(b)).

Experience with these machines has proved that, they perform well for a small number of PEs. But, these architectures do not scale well as processors and memory are

added. When PEs are added, contention for the system's limited bus bandwidth is increased, degrading processor-memory performance. Most shared access systems have a local *cache* at each PE to increase their effective PE-memory bandwidth.

2.3 Cache

Cache works on the principle of *locality of reference* [16], which describes the fact that over an interval of time, the addresses generated by a typical program is clustered in space. In other words : **1.** Recently referenced items are likely to be referenced again in the near future - *Temporal locality* . **2.** A process has a tendency to access items whose addresses are near one another - *Spatial locality* . **3.** In typical programs, execution of instructions follow a sequential order unless a branch or out-of-order instruction is encountered - *Sequential locality*.

Thus, caches enable fast accesses of memory. However, in the shared systems, a problem of data inconsistency or cache coherence is introduced. This is possible when one PE updates a variable in its cache, and another PE reads the variable from shared memory, thus getting the old incorrect value. Methods have been devised to alleviate this problem in totality, using hardware techniques. So, adding PEs, requires more logic to manage the memory system thus, increasing its cost and reducing performance.

Distributed-memory or Message Passing systems however, do not suffer from the above mentioned drawbacks. Since each PE has its own memory, potential bottlenecks associated with PE-memory bus and the need for elaborate cache management subsystems are completely eliminated. Adding PEs adds memory, and PE-memory bandwidth scales with computational power. The nCUBE 3E systems have incorporated a sophisticated interconnect topology to facilitate a high bandwidth inter-PE communication. Their methods and the interconnect topologies will be described in another chapter. Further, based on their bus interconnections, systems in figure 2.2(a) are called tightly coupled and figure 2.2(b) systems are loosely coupled.

2.4 The Interconnect

A variety of interconnection networks can be used to connect the PEs in any distributed or shared memory systems. Broadly, interconnection networks are classified into *Static* or *Dynamic* networks.

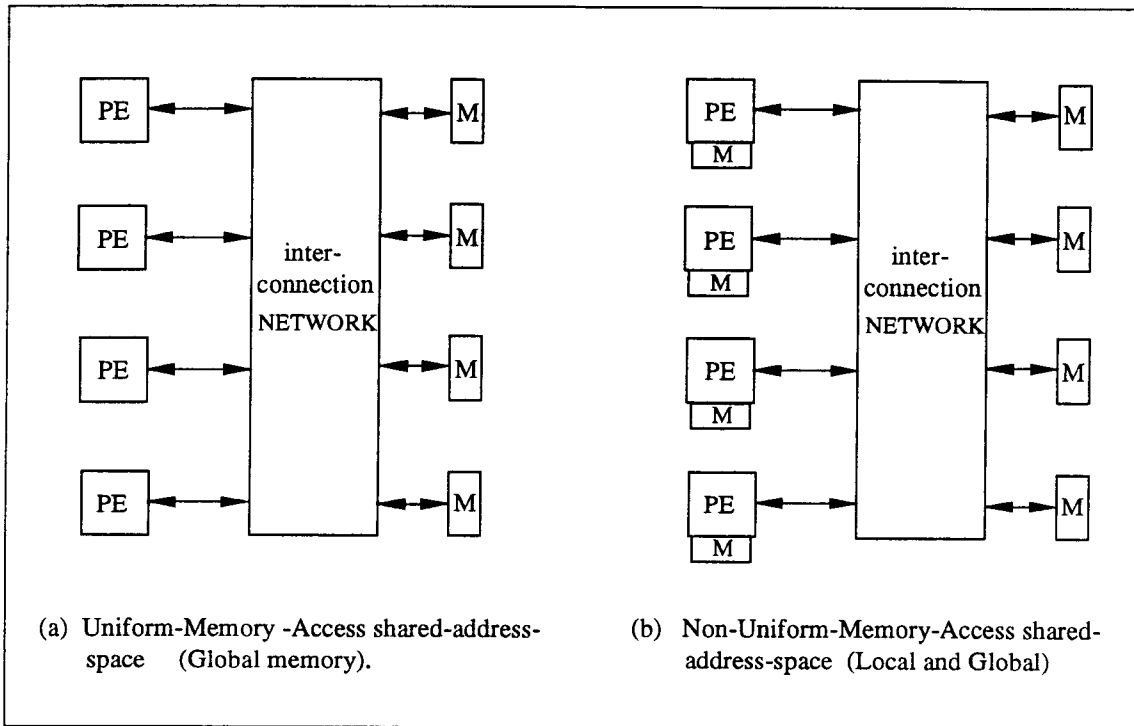


figure 2.2 Shared Memory Architectures

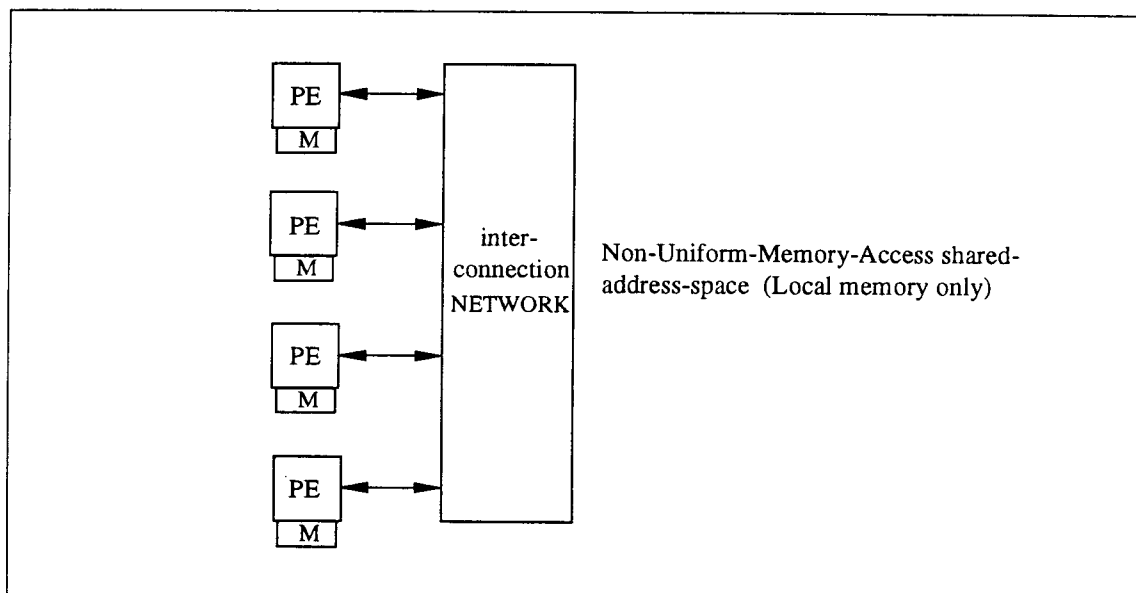


figure 2.3 Message Passing Architecture (Distributive Memory)

As the name implies a Static Interconnect consists of fixed links between the PEs and can be considered point-to-point or a direct network. Linear Arrays, Rings, Chordal Rings, Mesh, Torus, Tree, Star, Barrel Shifters and Hypercubes all fall in this category [16 and 28]. These topologies are used to construct message passing computers. Dynamic networks are implemented with switches and communication links. The switched channels are dynamically configured to match the communication demand in user programs so as to establish paths among PEs and memory banks. These are referred to as indirect networks [35] and are used to build shared address space systems [15 and 35].

The communication efficiency of the underlying network is critical to the performance of a parallel computer. The **aim** is to achieve a low latency network with a high data transfer rate and thus a wide communication bandwidth [16]. Thus, the network properties need to be analyzed in order to make design choices of the machine size, system scalability and flexibility for data routing operations. Further, any topology under consideration should provide a high fault tolerance, regularity of structure, simple deadlock free routing algorithm, high I/O bandwidth, and small diameter. It should also be able to *embed* other topologies like ring, mesh and trees so as to facilitate portability of the applications written on the latter topologies [12].

The representative topology in this work has been the Hypercube. This thesis is not constrained to just this network, it can be extended to other systems too. Massively Parallel systems require an *interconnect topology capable of scaling to large number of PEs without degrading communication performance or increasing latency*. The hypercube meets these requirements. Additionally, the hypercube can efficiently emulate other topologies, assuring that users will always have the correct interconnect for the task at hand.

2.4.1 Interconnect Parameters

The *diameter* of a network is the maximum of the shortest distance between any two PEs in the network. Small diameter implies shorter distance and lower communication latencies. For a hypercube the diameter $d = \log_2 p$ where p is the maximum number of PEs. The dimension of a hypercube is its diameter.

Connectivity is the measure of the multiplicity of paths between any two PEs. High connectivity is desirable as it provides for alternate routing schemes and more reliability or fault-tolerance. For a **k-dimensional** hypercube the connectivity is **k**.

Bisection Width is the minimum number of communication links that have to be removed to partition the network into two equal halves. For a hypercube it is $p/2$.

Channel Width is the number of physical wires in each communication link between the PEs. Each wire represents a bit.

Channel Rate is the peak rate at which a single physical wire can deliver bits.

Channel Bandwidth is the peak rate at which data can be communicated between the ends of the communication links. It is the product of channel rate and channel width.

Bisection Bandwidth is the minimum volume of communication allowed between any two halves of the network with an equal number of PEs. It is a product of channel bandwidth and bisection width.

A good way of determining the cost of the network is by considering the number of communication links it has or the number of wires in all. The hypercube has $(\log_2 p) * (p/2)$ links. Thus, a 3 ($k=3$) dimension Cube with $p = 8$ has a cost of $3 * 4 = 12$ links.

2.4.2 Hypercube

Definition [12] : A k -dimensional hypercube is an undirected graph of 2^k nodes (PEs) having addresses between 0 and 2^k-1 such that there is an edge between any two nodes, if and only if, the binary representations of their addresses differ in only one bit.

Above, in our discussion of parameters, some of the properties of hypercubes have been outlined and a few more follow. Hypercubes for dimension k equal to 1, 2 and 3 are shown in figure 3.1.

Property 1: A k -dimensional hypercube [12] can be constructed recursively using two $(k-1)$ -dimensional hypercubes.

Property 2: In a k -dimensional cube, each PE is directly connected to ' k ' other PEs.

Property 3: The minimum distance between two nodes PE_i and PE_j is the number of bits that differ between the addresses of node PE_i and PE_j . This is called the **Hamming Distance** $H(PE_i, PE_j)$ [12]. Also, the Hamming Distance is the number of 'ones' in the binary representation of ' $PE_i.label$ EXOR $PE_j.label$ ' where EXOR is the bitwise exclusive-or operation and ' $PE_i.label$ ' indicates the bit address of PE_i .

Property 4: For a k -dimensional hypercube the binary representation of ' $PE_i.label$ EXOR $PE_j.label$ ' can at most contain ' k ones'. Thus the shortest path between any two PEs cannot have more than ' k links'.

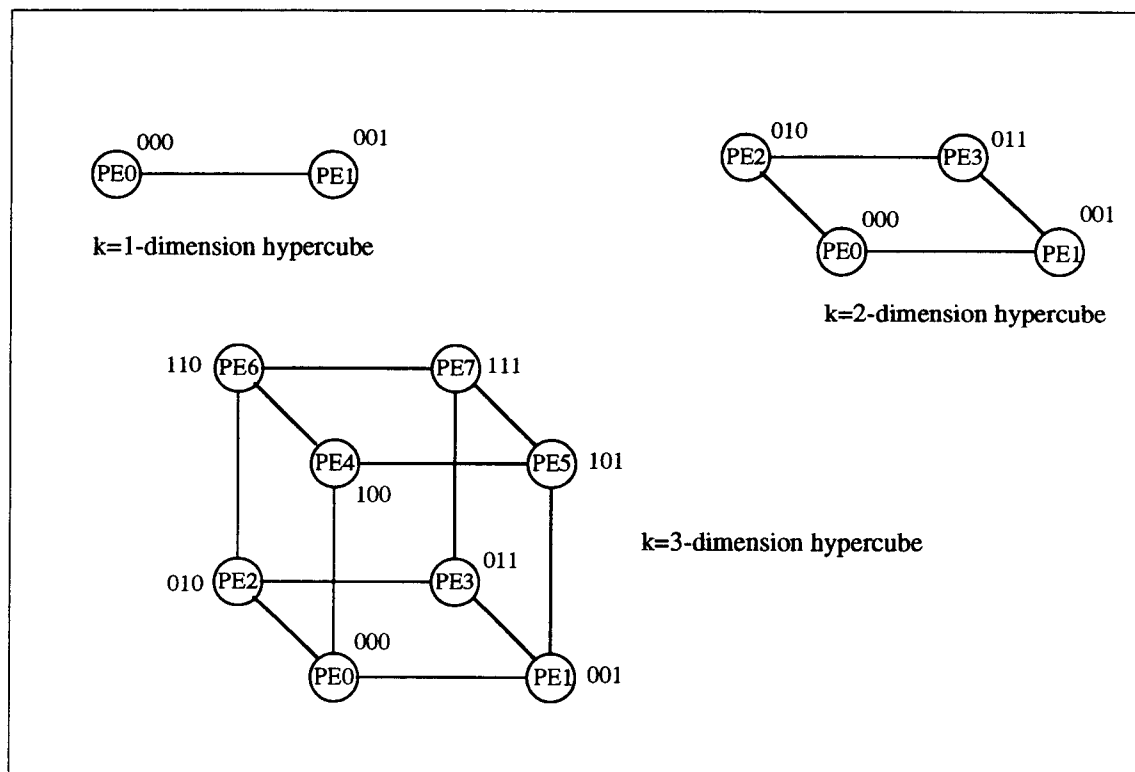


figure 2.4 Hypercubes for different 'k' dimensions

An exhaustive study of the properties of hypercubes can be found in [12] and figure 2.4 shows 3 different hypercube topologies.

2.5 Routing Concepts

Performance is a function of the algorithm used to route a message in parallel computers. The path it takes a message through the network to go from source to destination is called the routing mechanism. As this thesis is based on message passing multicomputers, the routing schemes, latency analysis or communication costs for it, with respect to the hypercube topology will be discussed.

A *message* is a logical unit of information for inter-PE communication. It can have variable length, and consists of arbitrary number of fixed length packets. A *packet* is a basic unit containing the destination address for routing and sequence number for

purposes of reassembly at the destination, as the packets may arrive in a random order and will need to be enveloped into the original message. Each packet also contains data.

A packet may alternatively be further divided into distinct **flow-control digits** or *flits*. The routing information (destination address and sequence number) occupy the header flits and the remaining flits are the data elements of the packet. Two switching schemes that are used for routing messages in hypercubes are *Store-and-Forward (SFr)* and *Cut-Through routing (CTr)*. These will be described in sections 2.7.1 and 2.7.2 respectively. However, the smallest unit of information in SFr is a packet and in CTr, it is the flit. Typical packet lengths vary from 64 bits to 512 bits.

Routing mechanisms can be classified by two main criteria : one classification is based on the selection of the path, namely *minimal* and *nonminimal* and the other classification is based on how the network-state information is used, namely *deterministic* and *adaptive*.

In short, minimal routing scheme requires each link to bring the message closer to the destination by selecting the shortest path between the source and the destination. This may cause congestion at some nodes on the network. In contrast, the nonminimal will not care for the shortest path, but try to avoid any network congestion and inadvertently always route the message by a longer path.

Deterministic mechanism determines [15] a unique path for a message, based on the source and destination. This may result in uneven use of the network resources. Adaptive routing, in contrast requires the current state of the network so as to route the message around any congestion's as they may arise.

The adaptive-nonminimal routing scheme requires added intelligence at each node and is more expensive in terms of time and money. A commonly used routing mechanism for hypercubes called *E-Cube* routing which falls under *deterministic-minimal* scheme is described. Further, the E-Cube routing algorithm may use either, the SFr or CTr switching schemes.

2.6 E-Cube Routing in a 3 dimension Hypercube

$k=3$ (dimension)

Let PE0.label = 000 source address (src) and
 PE7.label = 111 destination address (dst)
 src **EXOR** dst = 111

The 3 bit positions - Least Significant Bit (LSB on extreme right) bit0, bit1 and Most Significant Bit (MSB) bit2, denote the 3 dimensions of the hypercube. PE0 first forwards the message along the dimension corresponding to LSB. This means the message will go to a connected PE whose address will differ only in bit0. It is PE1. label = 001. This is now the new source 'src'.

$$\text{src EXOR dst} = 001 \text{ EXOR } 111 = 110$$

PE1 will forward it to PE3 whose label differs in the bit1 position. PE3 now becomes the new source and its label = 011

$$\text{src EXOR dst} = 011 \text{ EXOR } 111 = 100$$

PE3 will finally forward the message to the destination PE7 along its MSB (bit3) dimension. Refer to figure 2.5 for the above example on E-Cube routing.

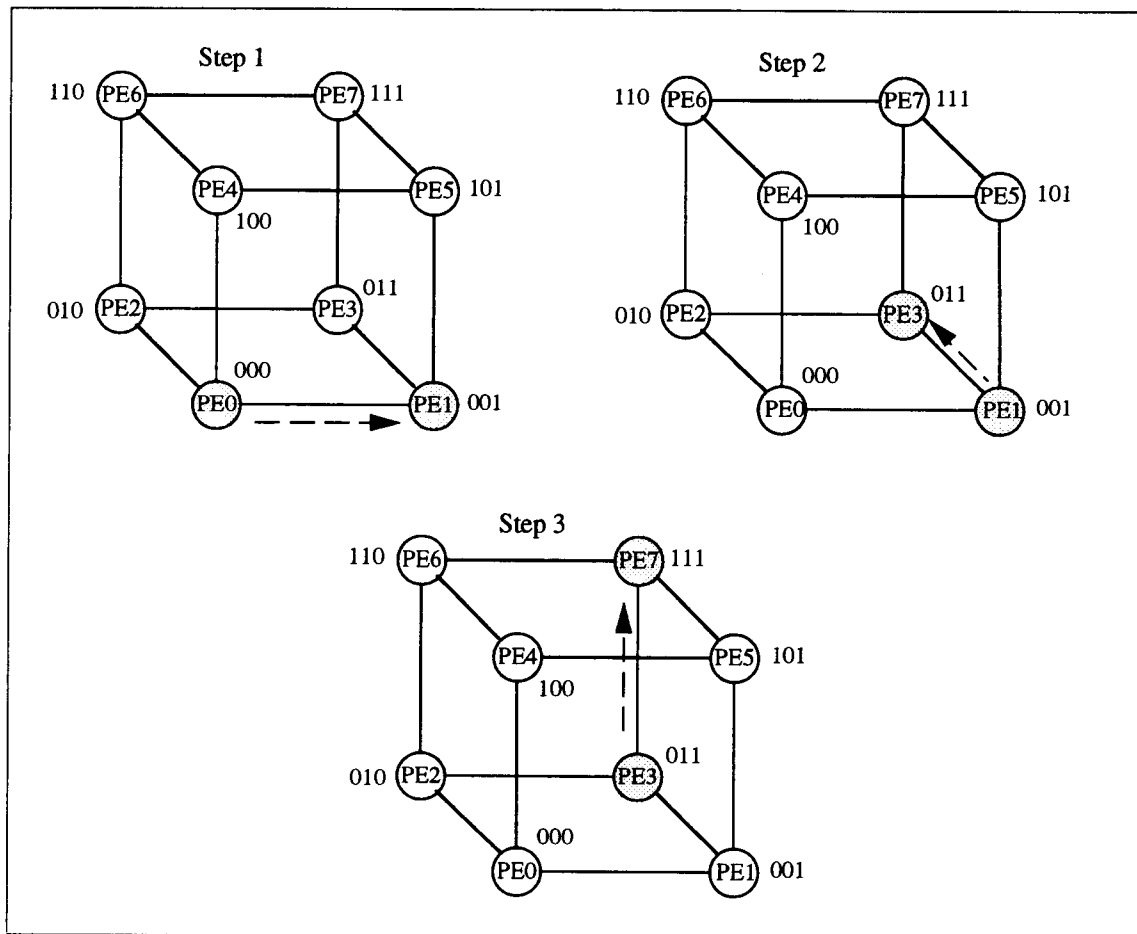


figure 2.5 E-Cube Routing in a 3-dimensional Hypercube network

2.7 Communication Latency

The total time it takes to prepare the message for transmission and the time taken to traverse the network to reach the destination is equal to communication latency t_{comm} [35].

Startup Time t_s : It is the time to prepare the message at the source PE. It includes the time to add the header, trailer, error correction information and also includes the time taken to execute the routing algorithm and establish the path [35]. This is a one-time cost.

Hop-Time t_h : The time taken by the header to travel between directly connected processors on the network. Called also as *Node latency*, it is related to the routing switch delay for determining which output buffer or channel the message should be forwarded to [35].

Word Transfer Time t_w : If channel bandwidth is x words/second then each word takes the time of $t_w = 1/x$ to traverse the link [35].

2.7.1 Store-and-Forward Routing (SFr)

A packet is transmitted from a source to destination via intermediate PEs in the path. Each node has a packet buffer. When an intermediate PE receives a packet it stores the complete packet in its buffer. Then it is forwarded to the next node or PE only if the desired output channel and the packet buffer on the receiving end are both available [15].

Let m : message size - m words long.

l : number of links to be traversed

Cost for traversing each link is t_h for the header and $m.t_w$ for the remaining message. Thus,

$$t_{comm} = t_s + (t_h + m.t_w).l$$

as t_s is incurred only once [35]. However, in today's systems t_h is quite small and if neglected from above equation, the communication latency becomes

$$t_{comm} = t_s + (m.t_w).l$$

2.7.2 Cut-Through Routing (CTr)

As mentioned earlier, the smallest unit of information is the *flit*. In particular, CTr is also called *wormhole routing*. All flits in the same message are transmitted in order as inseparable companions [15] in a pipelined fashion. Only the header flit knows where the packet is destined for, and the other flits just tow along the header flit.

The message is routed from the incoming link to the outgoing link as it arrives. An intermediate PE does not wait to receive the entire message as in SFr, but the hardware routers in each node read in the header flit information and advance it to the right channel. Different packets can be interleaved during transmission, but the flits within cannot be mixed up. Also, flits will always go on the same path taken by the header. Since, each PE is not required to store the entire message, CTr requires less memory and memory bandwidth at each intermediate PE and is real fast. In fact it can be proved [15] that wormhole routing has a latency almost independent of the source and destination.

There can be instances of *deadlock*. If a flit is available at each of the PEs to be dispatched, but the outgoing links of those PEs are already occupied by other flits, or the receiving PE flit buffers are yet to be emptied, then a state of deadlock is reached as nothing moves till one flit is discarded from the network. This loss of information is unacceptable and so, a routing algorithm like E-Cube routing is preferred as it is deadlock free [26 and 35].

For the same identity of 'm' and 'l' from above, the message header takes a time of $l.t_h$ to reach the destination. The entire message will arrive in $m.t_w$ after the header. Thus, the total communication latency for CTr is

$$t_{\text{comm}} = t_s + l.t_h + m.t_w$$

It is clear that Cut Through routing takes less time to route a message and it is the scheme used in today's multicomputers employing Hypercube topology.

For all parameters being the same for a particular k-dimension hypercube, the STr is a function of (m.l) and CTr is a function of (m+l). So, for small 'm' or 'l' both behave the same but for larger 'm' or large 'l' (higher dimensions) the CTr is better. For a single message a comparison of SFr and CTr is shown with respect to time in figure 2.6.

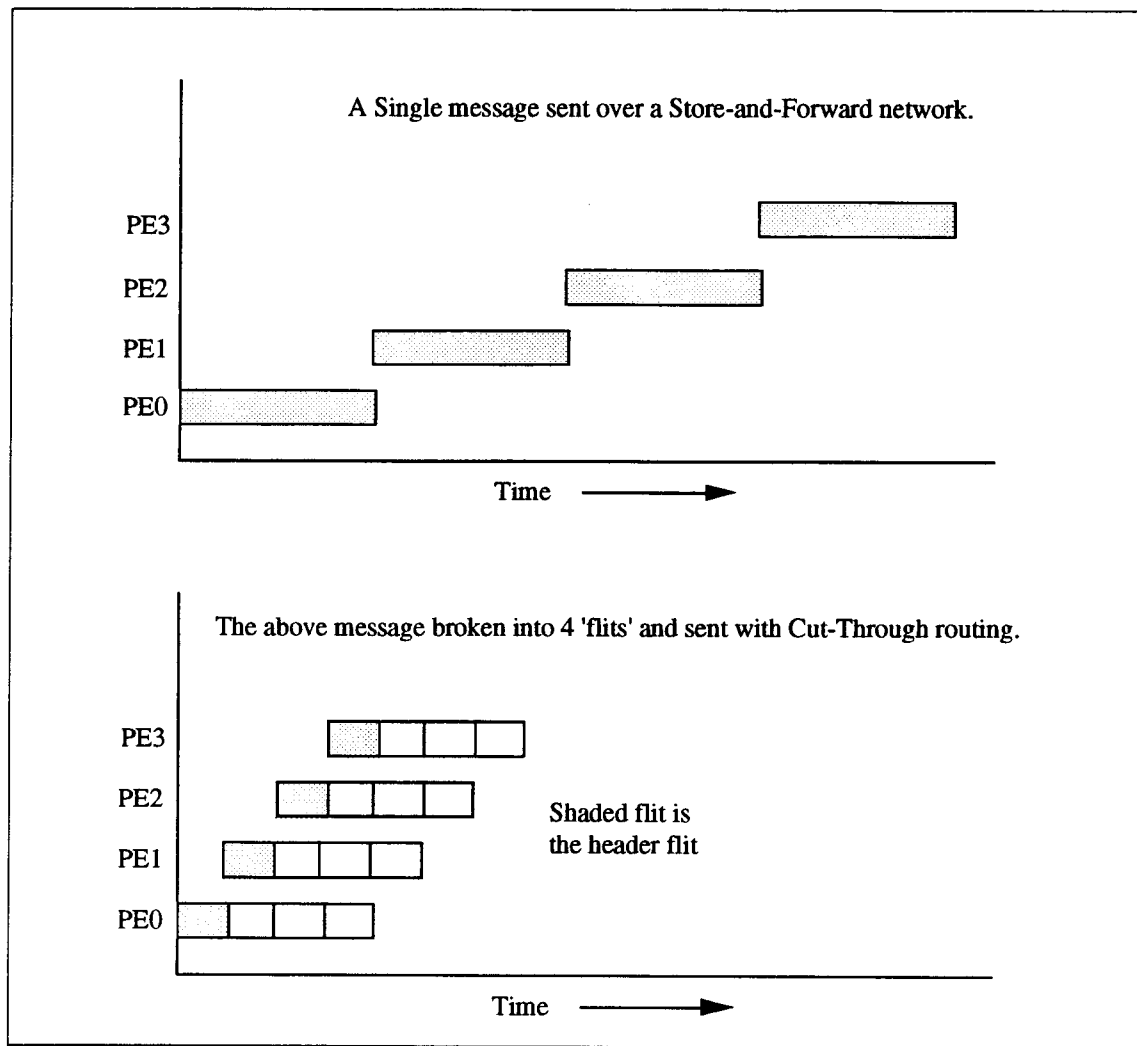


figure 2.6 Comparing Store-and-Forward with Cut-Through routing for a single message.

2.8 nCUBE 3E InterConnect

The nCUBE 3E can scale from 8 to 65,536 processors. This implies a dimension range of 3 to 16, so a maximum of 16 hops (as $2^{16} = 65,536$) in a completely configured system. The architecture can be upgraded in increments of 8 processors.

The Start Up Time is as low as $5 \mu\text{s}$ ($1 \mu\text{s} = 1 \text{ microsecond} = 10^{-6} \text{ seconds}$) as compared to other available machines having $100 \mu\text{s}$. This low value of $t_s = 5 \mu\text{s}$ is achieved by using *Direct Memory Access Channels* (DMA) at each node that initiate the transfer.

It employs cut-through routers which forward the messages with no software intervention and buffering, at an intermediate hop latency of 200 ns (1 ns = 1 nanosecond = 10^{-9} seconds).

It has internal data buffers between each hypercube DMA channel and the memory system, sustaining high transfer rates on multiple channels with minimal interference. As cache and DMA buffers are on the same device, cache resident data can be directly transferred to a message without first being written to memory.

The Channel Bandwidth = 50 MB per sec (MB = Mega Bytes). nCUBE 3E channels are full duplex and parity checked for reliability. Full duplex indicates a two way transmission. Parity-check implies checking for the odd or even number of '1's in a sequence of bits at sending and receiving ends so as to ensure integrity of data. The nCUBE 3E has 16 channels reserved for interprocessor communication. This supports a dimension of 16 providing for 65,536 computational nodes.

Importance of the above parameters can be appreciated and their effect noticed when the communication latency is examined over a particular topology for different message sizes and hops.

Let the dimension $k = 4$

1 Word = 8 bytes = 64 bits (data path of nCUBE 3E processor)

Start up time: $t_s = 5 \mu s$

Hop Time: $t_h = 200 \text{ ns}$

Channel BandWidth = 50 MB per sec

Word Transfer Time: $t_w = 1 \div (50 \times 10^6)$ seconds per byte

$t_w = 20 \text{ ns/byte}$

Now for a message size of 1 word the communication latency from PE0 (address = 000) to PE7 (address = 111) involves 3 hops (1). From above section of CTr

Communication Latency :

$$\begin{aligned}
 t_{\text{comm}} &= t_s + l.t_h + m.t_w \\
 &= 5 \mu s + 3 \text{ hops} \cdot 200 \text{ ns/hop} + \\
 &\quad 1 \text{ word} \cdot 8 \text{ bytes/word} \cdot 20 \text{ nano-seconds per byte} \\
 &= 5.10^{-6} + 600.10^{-9} + 160.10^{-9} \\
 t_{\text{comm}} &= 5.76 \mu s
 \end{aligned}$$

One word takes 5.76 μs and 10 words take 7.2 μs . Further for 100 words the latency is 21.6 μs . This indicates that the dominating factor here is the start up time t_s . Even when the order of the message size is increased by a factor of 10, the latency increases at a much lower rate.

If the number of hops is doubled, say 6, for a 6 dimension Cube, then the latency increases to $6.36 \mu\text{s}$ for 1 word and to $7.56 \mu\text{s}$ for 12 hops in a 12 dimension cube.

Present research trends have brought down the communication factor considerably, but still more work is required to achieve close to ideal results.

The speed of any PE for nCUBE 3E is given to be 50 MHz (1 MHz = 10^6 cycles per second). This gives the processor cycle time to be

$$t = 20 \text{ ns/cycle.}$$

Hence, $t_s = 250 \text{ cycles}$

and $t_h = 10 \text{ cycles}$

Execution times of all operations in this thesis are expressed in cycles.

2.9 Performance Parameters

A parallel system is a combination of an algorithm and the parallel architecture it is implemented on. A parallel algorithm can best be evaluated when taking the architecture into account. Following are a few parameters that are important to this thesis

2.9.1 Speed Up and Efficiency

For a given program T_{seq} denotes the sequential or serial time to execute the program on a single processor. T_p denotes the parallel execution time on p PEs. Thus, for p processors the speed-up S_p is :

$$S_p = T_{\text{seq}} / T_p \quad (2.1)$$

It is the relative benefit of solving the problem in parallel. Consequently Efficiency for the above system of p PEs will be given as :

$$E_p = S_p / p \quad (2.2)$$

As speed-up can never exceed the number of PEs for a given run, we have the condition : $0 \leq E_p \leq 1$.

2.9.2 Ideal Speed Up

It is the *ratio* of the Sequential-Time to the Critical Path Time. Critical path time is the lower bound time or the minimum time above which the performance cannot

improve. This is dependent on the application algorithm and node execution times. The modus operandi of identifying the critical path is discussed in the section on BLAS in chapter 5. Thus, ideal speed up is given as:

$$\text{Ideal Speed up} = T_{\text{seq}} / \text{CP-Time} \quad (2.3)$$

2.9.3 Cost

Cost of solving an algorithm on a parallel machine is :

$$\text{Cost} = p \cdot T_p \quad (2.4)$$

It reflects the sum of the time spent by each PE in solving the problem. Also, it is a denominator for the expression of Efficiency if S_p is substituted. Higher the Cost lower is the Efficiency. Also, Efficiency can be viewed as the ratio of Sequential Cost to Parallel Cost. Cost is also expressed as *Processor-Time product* ($p \cdot T_p$) [25 and 35].

A Cost-Optimal parallel system is one in which the cost of solving the problem is proportional to the execution time of the sequential algorithm on a single processor. A Cost-Optimal parallel system has an efficiency of 1.

2.9.4 Performance Ratio

The variable that gives a better understanding of the scalability effects in . This is indicated by **PR** and is -

$$\text{PR} = S_p / \text{Ideal Speed-up} * 100 \% \quad (2.5)$$

This percentage gives the merit for how well the algorithm on a parallel system reaches its ideal speed-up when the PEs and the problem size is increased. Since,

$$1 \leq S_p \leq p \text{ and}$$

ideal speed-up is greater than S_p , then $SR/100$ lies between 0 and 1. Further elaboration will be done when discussing results in chapter 5.

2.10 Parallelism

Primary goal for designing parallel systems is to minimize program execution time. A large number of scientific and engineering problems have pushed for the need to exploit as much parallelism so as to adequately solve them in real time.

Problems may have different parallel formulations, which result in varying benefits, and all problems are not equally amenable to parallel processing.

Two major components of parallel algorithm design consists of 1) specifying the concurrency and 2) specifying the data locality [35]. *Concurrency* is the identification and the specification of the problem as a set of tasks that can be concurrently performed. It is necessary to keep the processors busy. *Data Locality* is mapping of these tasks onto various PEs so as to minimize the overhead of communication.

2.10.1 Types

Parallelism can be broadly seen as *Structured* or *Unstructured*. The former implies a set [25] of independent identical tasks or processes that operate on different data sets. The latter implies different instructions with different data streams. Also termed as *Data Parallelism*, structured parallelism, by its very nature has been easier to deal with. An example of Data Parallelism is the matrix multiplication problem which has been discussed in detail in this thesis in chapters 3, 4, 5 and 6. Further, unstructured parallelism is also referred to as *Control Parallelism* [16 and 35]. A good example of Control Parallelism is where processors are used in a pipeline. The computation is parallelized by executing different parts of a program at each PE and sending intermediate results to the next PE. The result is a pipeline of data flowing between processors.

Many problems show a certain amount of both data parallelism and control parallelism [35]. However, the amount of control parallelism available in a program is independent of the size of the problem and is thus limited, but, the amount of data parallelism in a problem is directly proportional to the size of the program. Therefore, to use a large number of PEs efficiently on large problems, it is necessary to exploit any inherent data parallelism within the application.

Specifying parallelism is an important step towards extracting best results from parallel systems. A user chooses the best available parallel algorithm. The data structures are then organized to suit the parallel architecture, e.g., vectors are better than linked list for they allow parallel access [16, 25 and 26]. The algorithm is then coded using an appropriate language. Either this language is a known serial language like FORTRAN or C with parallel constructs or it is a new parallel language like SISAL or Id. SISAL has been the choice for this thesis and its brief discussion and motivation will be presented in chapter 3.

2.11 Program Granularity

Coding the program into independent modules of computation called subroutines helps in its parallel execution. Parts of the computation are organized as subroutines that can be made to execute in parallel. Parallelism at the subroutine level can be described as *Coarse Grain*.

Loop level parallelism is *Medium Grained*. Parallelism can be specified by the user or the compiler can figure it out by doing a dependence analysis at the loop level. Different types of parallelism exist at this level based on the dependence graph of the loop.

Operation or Instruction level parallelism is called *Fine Grained*. Though actual program runs have shown that fine grain parallelism does not give better speed-ups as compared to medium grain, it is still important with respect to the parallelism that can be exploited within the PEs like using pipelining, multiple functional units and other intra PE parallelism.

2.12 Processor Granularity

Parallel Computers are constructed with varying number of PEs. Some machines like Cray Y-MP contain a small number of very powerful PEs and are called *Coarse Grain* computers. Cray Y-MP offers 8-16 PEs, each capable of several Gflops (1 Gflop = 10^9 floating point operations per second). *Fine Grain* computers are those which consist of a large number of less powerful PEs [26 and 35]. These are CM-2 which has at a maximum of 65,536 one-bit PEs and MasPar-MP2 containing up to 16,384 PEs. In between, *Coarse* and *Fine* there exists a class of computers in which each PE gives workstation type performance. These are the *Medium Grain* computers containing a few thousand processors like nCUBE2, Thinking Machine's CM-5 and Intel's Paragon XP/S. This thesis work has simulated a **nCUBE 3E Series** architecture (primarily the network) configuring it as a medium to coarse grain system (64 to 1024 PEs).

Applications that have higher degree of concurrency make effective use of a large number of less powerful PEs in fine to medium grain computers. In contrast, applications with lower degree of concurrency make cost effective use of coarse grain systems. Further, PEs that are used in coarse grain machines are fast, specially fabricated and not available in large scale, so they cost more [15 and 25] as compared to the medium grain and fine grain computers that have a record of using off-the-shelf processors. This

requires a tradeoff between cost and utility of the computer [35] when choosing processor granularity.

The granularity (defined formally in [35]) of a parallel computer is the ratio of the time required for a basic communication operation to the time required for a basic computation. This is also applicable to program granularity where-in, the C/T ratios are calculated for different programs (section 5.8) to compare their grain sizes and achieved speed ups.

2.13 Utilization

Experience has shown that only a small fraction of the maximum available parallelism can be realized [25 and 26]. This will also be seen in the results section in chapter 5 when considering the maximum attainable parallelism and achieved speed ups.

Restricting forces during parallel execution include overheads of synchronization, inter PE communication, scheduling and any other random delays due to memory and network conflicts. Many considerations [25, 27, 29 and 30] have prompted researchers to write more sophisticated and automated tools to specify parallelism, restructure and map problems, rather than leave any of these steps to skills of the best programmer. This thesis is a step towards that automation.

SISAL programs on their intermediate compilation gives a file containing multiple acyclic graphs. The implicit dependencies between these graphs are translated to actual dependencies by a graph preprocessor called the Automator. The translated graphs are then allocated to different PEs in a hypercube networked system by the allocation algorithm - BLAS. Chapter 3 describes SISAL and the IF1 stage.

CHAPTER 3. A PARALLEL LANGUAGE - SISAL

After extensive research and use, *auto-parallelizing* compilers have not met expectations in automatically parallelizing imperative languages. Imperative languages like Fortran-77, C, Pascal, Cobol and others have been developed for sequential machines. The model of computation for these assumes a single processor and a single instruction stream. Since, a large installed base of imperative code already exists, the auto-parallelizing compilers will stay for some time to come but, they do not present a long term solution. Also, [22] the existing languages reflect the storage structure of the von Neumann concept of computer organization in that each language has some method of effecting a change in state of the memory that cannot be modeled as a local effect. This makes it very difficult to analyze parts of the program that may be executed concurrently.

Many of the present parallel languages for shared memory or message passing systems are essentially sequential (imperative) languages 'augmented' by a set of special *system calls*. These calls provide low-level instructions (also called primitives) for message passing, process synchronization, process creation, mutual exclusion, and other necessary functions [35]. *Synchronization* is a general term for timing constraints imposed by the communications and interactions between concurrent processes. In order for these 'extended' languages to be used on parallel systems, information stored in different PEs must be explicitly shared using the above primitives. These features will make programs efficient, but, also difficult to understand, debug and maintain. Further, lack of standards in parallel languages has made program portability across architectures very difficult. In general, [25] if the communication patterns required by the parallel algorithm are not supported by the parallel language, then the parallel program will be less efficient. Also, the communication patterns will vary over different architectures.

3.1 The Data Flow Concept

A well known representative to the approach of making parallel processing more accessible are the data flow systems. These start with the *data driven* model of computation, employ a different style of high level programming and design special purpose architectures targeted specifically to the execution of data flow programs. The study of data flow systems is out of scope for this work but can be found in [2, 3, 6, 7, 13, 19 and 23]. However, the concepts laid out by the data flow research community has

opened many avenues to integrate the data flow model with the conventional von Neumann architectures. This has resulted in hybrid systems and, in keeping alive new parallel languages suited for parallel computing.

The von Neumann model [1, 2, 3, 6, 7 and 13] is characterized by two main aspects:

1. It has a global addressable memory to hold the program and data objects, with program instructions frequently updating the contents of memory during execution.
2. It has an instruction counter which holds the address of the next instruction to be executed. This counter is either implicitly (incremented after each instruction) or explicitly (depending on the jump address) updated to provide the machine with a sequence of instructions to execute. This characteristic implies a single locus of control - a fundamental limitation for parallel processing.

In contrast, the data flow model [1, 2, 3, 6, 7 and 13] is characterized by 2 features:

1. It deals only with values and not with names of value containers i.e., the addresses. This property is fundamental to functional or applicative languages which have no built-in concept of global updatable memory. An operator in these languages produces a value that is used by other operators.
2. There is no single locus of control i.e. no program-counter.

An instruction in this model is enabled (executed or issued), if and only if, the required input values or operands have been computed earlier. The executing instruction produces a set of output values which will be passed on to the other instruction or instructions that may need it. Thus, an instruction [2 and 17] in data flow has no other side-effects, and a language based on such a model does not introduce sequencing constraints other than ones imposed by the data dependencies in the algorithm. In principle, it is possible to expose all of the parallelism in a data flow program. A program in any high level data flow language like Id, VAL or SISAL directly translates into a graph where the *nodes represent functions* and *arcs represent dependencies*.

3.2 Functional Languages

The term *functional* is often used [2, 3 and 6] to describe a language that operates by application of functions to values.

Few properties of functional languages and their brief explanations will allow a better understanding.

3.2.1 Properties

Property 1. Locality of effect [1, 2 and 13]

Locality of effect means instructions do not have unnecessary far reaching data dependencies. Consider a program fragment

$$M = X + Y$$

$$N = M / Y$$

$$O = X * M$$

Here M, N and O are all temporaries. Another fragment appearing elsewhere in the program may use the same temporaries to do an unrelated computation. Both fragments will not be allowed to execute in parallel because, of apparent data dependencies that is created from duplication of temporaries. Functional languages take care of this by limiting the *scope* of a variable to a particular region or procedure in which it is active. Thus, they avoid global variables.

Property 2. Freedom from Side Effects

Side Effects can be seen when procedures modify variables in the calling program or even when they modify their own arguments. In general, if arrays or records exist as global objects in memory and are manipulated by statements and passed as pointers or procedure parameters, it is very difficult to know what effects a modification on an array element may have elsewhere in the program.

A solution sought by functional languages is to use *call by value* instead of the traditional von Neumann approach of *call by reference*. In 'call by value' schemes a procedure copies its arguments, even if they are arrays. Thus, the actual arguments are never modified in the calling program .

Freedom from side effects ensures that data dependencies are the same as sequencing constraints. It is more difficult [1 and 13] to achieve than Locality as it requires fundamental changes in the way a language's *virtual machine* processes data.

Property 3. Single Assignment Principle [1 and 13]

A variable may appear on the left side of an assignment only once within the area of the program within which it is active. It allows more clarity and ease of program verification. This feature is easy to achieve.

The above 3 properties are amongst the important ones and a more detailed explanation may be sought in [1 and 13]. Hence, functional languages promote the construction of correct parallel programs by isolating the programmer from the overwhelming complexities of parallel computing.

3.3 SISAL

SISAL belongs to a class of applicative languages which share the following characteristics :

1. Expression based or Function Based.
2. Determinate results when a function is invoked with same arguments any number of times.
3. No side effects.
4. Lack of global variables.

SISAL stands for *Streams and Iterations in a Single Assignment Language*. The single assignment feature, as described above, allows a name to receive a value only once in each *name-scope* after which it becomes a read-only value to be shared amongst any number of consumers. This property eliminates one of the major causes of indeterminacy in parallel execution [8 and 22]

Being entirely free of side effects each module or well formed [22] portion of SISAL program corresponds to a mathematical function and the entire effect of putting two parts together is to compose the corresponding functions. The input and output issues are addressed through the introduction of streams of values that communicate between program modules.

The language includes the standard scalar types namely Boolean, integer, real and double precision. In addition there are aggregate types like arrays, records, unions and streams. Multidimensional arrays are implemented as arrays of arrays [31]. The streams in SISAL are *non-strict* : the values are available as soon as produced. The loops in SISAL may be sequential or parallel. It supports the construct ***ForAll*** to specify loops with all iterations to be executable in parallel. In addition, decision constructs like ***Select***

executing the *if-then-else* format and **TagCase** is available. These complex constructs typical to **SISAL-IF1** will be discussed at length in the chapters 9 and 10.

The SISAL project [8] is a collaborative effort between *Lawrence Livermore National Laboratory (LLNL)*, Colorado State University, University of Manchester and Digital Equipment Corporation.

3.3.1 The SISAL Compiler

The *osc* is an Optimizing SISAL Compiler developed at LLNL. The figure 3.1 shows the different stages of the compiler. SISAL allows for independent compilation units. Being typical to applicative computing, the *osc* compiler [31] eliminates the extra copying and memory overheads.

The *osc* starts out at the SISAL front-end by translating the source files with suffix **.sis** into an intermediate form called **IF1**. The IF1 is the basic core around which this work revolves. IF1 files define the dependencies between data flow graphs. The **IF1LD** links the IF1 files into a monolith and presents them to the successive stages for optimization.

The **IF1OPT** is a machine independent optimizer that carries out some standard optimizations described below.

Loop fission; Loop splitting; Constant folding; Invariant removal; Parallel operation hoisting (from sequential loops); Loop inversion (pulling tests out of loops); Loop fusion; Common subexpression elimination; Dope vector optimizations; Loop unrolling and Cascaded test removal.

Details on the optimizations can be obtained from [8]. The next phases in the compilation are the **IF2** stages, which are essentially machine dependent supersets of IF1. The IF2 stages support abstract memory referencing. The **IF2MEM** is a build-in-place analyzer which allocates abstract memory locations to IF1 nodes. **IF2UP** is the update-in-place analyzer identifying the opportunities for carrying out some operations in place without additional memory allocation [8 and 31]. The monolith, now optimized and translated into IF2, is given to the **IF2PART** for parallelization. Finally, **IF2GEN** translates the IF2 graphs into **C code** or **FORTRAN code**. The C or FORTRAN compiler links the runtime libraries and produces binary executable code to be run on the target machine.

The translation for BLAS begins *after* the IF1OPT stage because the feature of machine independence is maintained only till that phase in compilation.

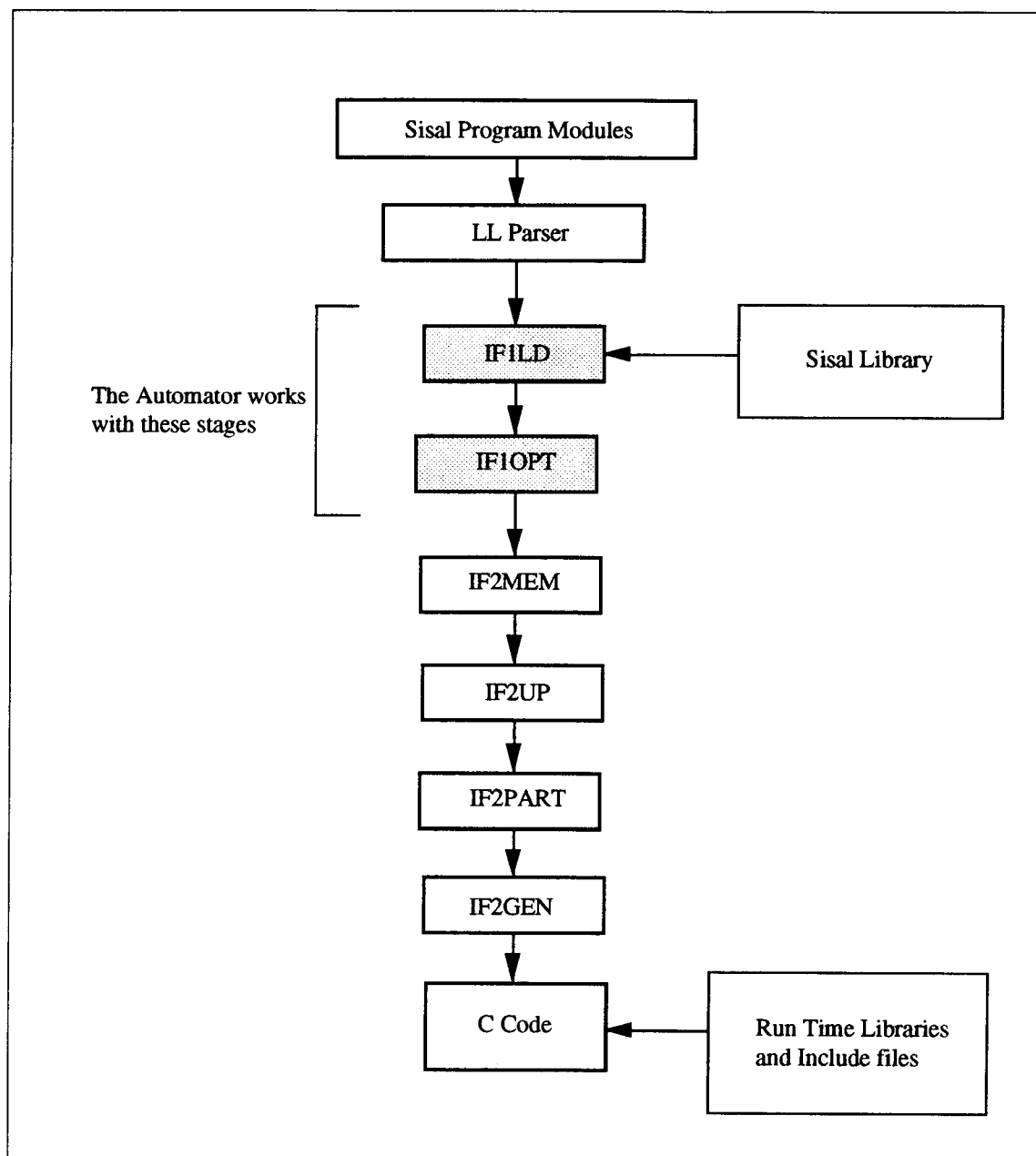


figure 3.1 Structure of SISAL's "osc" compiler

3.4 IF1 Phase

Any restructurer has to perform sophisticated dependence analysis and build a *data dependence graph* (**DDG**) of a program [25]. The DDG is an internal representation of a

program that is used by most subsequent passes to carry out variety of transformations and optimizations keeping the semantics of the source program. The DDG is a directed graph, where nodes correspond either to program statements or tasks and arcs represent dependencies. Dependencies imply a partial order of execution on the program statements or tasks as it may be. The IF1 stage is such a restructurer.

Intermediate Form-1 is a graphical language based only on acyclic data flow graphs. There are four components to an IF1 graph [31]: *Nodes*, *Edges*, *Types* and *Graph Boundaries*. Nodes denote operations of the language, Edges represent values and operands that are passed from node to node, Types are attached to each edge or function and lastly, Graph boundaries surround a group of nodes and edges. These components will be addressed in detail in the section 3.5 and will find place in the description of the Automater.

3.4.1 Motivation

Motivation to use IF1 in this project primarily came from following considerations :

- 1) IF1 is a graphical representation of the program. The acyclic nature of the graphs gives an opportunity to realize the Automater tool with ease.
- 2) There are no control operators within the graph as is in Id - a well known data flow language used widely with true data flow architectures. Language constructs like iterations and decision blocks like if-then-else are clearly defined to contain separate subgraphs each having its own scope and acyclic nature.
- 3) Preparing these graphs for any scheduling scheme can be achieved as they are completely architecture independent. Thus, even though the SISAL project was initially focused on multiprocessor systems, the intermediate compilation and first stage of standard optimizations were made independent of the architecture.
- 4) Lastly, the source code for SISAL and IF structures is easily available from LLNL at no extra cost; in fact, support for the language is free and computer time on a CRAY supercomputer at LLNL is offered free for any research group working on SISAL related projects.

In contrast, the other candidate for consideration - Id Nouveau fell short of the above features in our evaluation in addition to its having a more dynamic scheme of graphical representation during run time.

3.5 The IF1 Language

A good aid in understanding the graphical language is to trace a real example, which will be followed through the text. The matrix-multiplication program and its IF1 representation follows. However, the complete description of the algorithm is left to a later section on the Implementation of the Automater.

A SISAL Program for matrix multiplication called **matmult.sis** is given below :

```
define Main
type OneD = array[double_real];
type TwoD = array[OneD];

% Multiple A[1..x, 1..y] and B[1..y, 1..z].
%
function Main(x,y,z: integer; A, B: TwoD returns TwoD)

  for i in 1, x cross j in 1, z
    Cij := for k in 1, y
              returns value of sum A[i, k] * B[k, j]
            end for
    returns array of Cij
  end for

end function % multiply
```

figure 3.2 matmult.sis

A detailed explanation of the program constructs can be obtained from the Language Reference Manual [22]. Brief analysis follows : The **define** statement is an export declaration stating which function will export values to the outside world. This is followed by a data **type** definition where-in, a 2-dimensional array-TwoD is an array of another array-OneD. The data typing is strict in SISAL. Source comments follow the % sign till end of that line only. Rest of the program is reserved for **function** definition which comprises of the function signature and one or more expressions defining the results. The function declaration for **Main** is read as : Composed of : variables - x, y, z (all integers) and Arrays - A and B (both 2-dimensional); it returns a 2-dimensional value. The **for** construct will be dwelled into, in chapter 4. The **cross** operation is known as the *cartesian* or *outer product* of sequences defining an implicit *nesting* of the **for** expressions. The above cross product means :

for i in 1, x

for j in 1, z

.....

The *osc* compiler interprets the 'cross product' as a parallelizable section.

When the source `matmult.sis` is compiled with the command

osc -IF1 matmult.sis

the source outputs an intermediate compilation stage called **matmult.if1**. A complete explanation for deciphering the IF1 file is available in the manual [33]. Here a brief idea is presented for clarity.

```
{ Compound 1 0
G 0 %sl=12
N 1 142 %sl=12
L 1 1 4 "1" %mk=V
E 0 1 1 2 4 %na=y %mk=V %sl=11
E 1 1 0 6 17 %na=k %mk=V
G 0 %sl=12
N 1 105
E 0 2 1 1 10 %na=a %mk=V
E 0 3 1 2 4 %na=i %mk=V
N 2 105 %sl=12
E 1 1 2 1 9 %mk=V
E 0 6 2 2 4 %na=k %mk=V
N 3 105
E 0 4 3 1 10 %na=b %mk=V
E 0 6 3 2 4 %na=k %mk=V
N 4 105 %sl=13
E 3 1 4 1 9 %mk=V
E 0 5 4 2 4 %na=j %mk=V
N 5 152 %sl=12
E 2 1 5 1 3 %mk=V
E 4 1 5 2 3 %mk=V
E 5 1 0 7 3 %mk=V
G 0 %sl=12
N 4 149 %sl=13
L 4 1 20 "SUM" %mk=V
L 4 2 3 "0.0D0" %mk=V
E 0 7 4 3 21 %mk=V
E 4 1 0 1 3 %mk=V
} 1 0 3 0 1 2 %sl=12
```

figure 3.3 Section of the `matmult.if1` file (Innermost *ForAll*)

Evidently, it is cumbersome to read such a representation. It serves better to create a graph (a drawing) from such a language for analysis. Also, the *.if1* files will be presented graphically for the remainder of this thesis.

The above section represents the innermost *for* statement in '`matmult.sis`', evaluating the expression for different values of 'k'. To see the graphical representation of the above section refer to figure 3.4.

3.5.1 Dissection of the `matmult.if1` section

The first non-blank character on any line in an `.if1` file distinguishes the structures. If the line begins with the following (upper case) ASCII letters then they mean :-

C	Comment
E	Edge
L	Literal
N	Simple Node
{	Start of Complex Node
}	End of the corresponding Complex Node
T	Type label
G	Graph (Function Graph or a Subgraph)
X	Graph (Exported Function)
I	Graph (Imported Function)

As mentioned earlier an IF1 graph consists of nodes, edges, types and graph boundaries. Here, a brief explanation is presented, summarizing important aspects of reading `.if1` files before one ventures to understand the actual data structures around which the code for IF1 files was built. The Automator tool has been developed using the structures of IF1 files to translate and present a SISAL program to the Balanced Layered Allocation Scheme.

3.6 Nodes

Nodes represent operations on values they receive. Each node has 2 set of ports : input ports to import the values from another node, or the graph boundary, or a constant (**Literal**) and output ports to export values to another node or graph boundary. Nodes are of two types : *Simple* and *Compound*.

3.6.1 Simple Nodes of IF1

Simple Nodes begin with a N in the above file and the are of the form:

N Label Operation

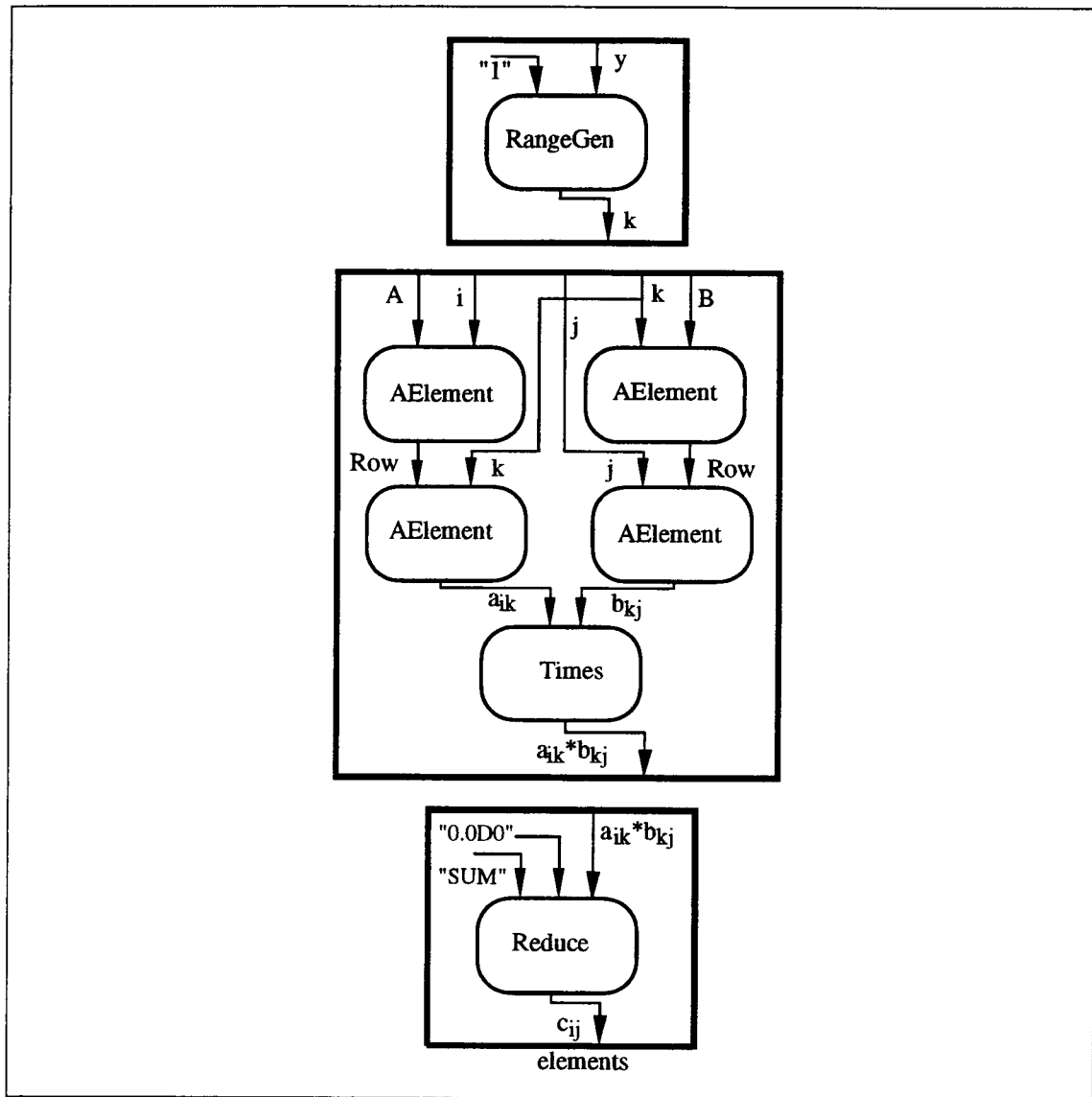


figure 3.4 Innermost Nested Level of matmult.if1

The **Operation** field is a code-number that may be obtained from the manual. The number '142' is a *RangeGenerate* operation. It generates a sequence of integers and places them on the output port. The inclusive range is given on its input ports. Most simple nodes have a fixed number of input and output ports although there are variable [33] number of inputs to nodes such as those that build structured data types. Simple nodes do : the arithmetic and Boolean operations; arrays, records, unions and streams manipulations; *Gather* and *Scatter* of multiple values and function calls.

3.6.2 Compound Nodes of IF1

Compound nodes contain subgraphs, while simple nodes do not. The semantics of the Compound Nodes determine the manner in which its subgraphs interact. Their form is :

```
{
G
...
G
...
G
...
} Label   Operation Code   Association List (Integers separated by a single space)
```

Hence, the section from **matmult.if1** file in figure 3.3, is a *ForAll* Compound node. The line beginning with a **G** represents a subgraph. The first **G** is the *Generator*, followed by the *Body* subgraph and finally the *Returns* subgraph. The **Operation Code** for *ForAll* (from [33]) is 0 and the **Association List** on the last line beginning with '}', is : **3 0 1 2 3**. The integers indicate 3 subgraphs numbered 0, 1 and 2 where-in, subgraph 0 feeds subgraph 1 and the subgraph 1 outputs are packaged by the Returns subgraph 2. The first integer in an association list gives the subgraph count. Defined within the scope of each **G** are the nodes, edges and literals.

3.6.3 Dependencies in a Compound Structure

Three kinds of implicit dependencies can exist within a compound node [33]:

- 1) Data Dependence between the compound node and its subgraphs.
- 2) Data Dependence between subgraphs of a compound node.
- 3) Control Dependence that may occur as a result of a Boolean condition being true or false.

Dependence 1) will occur when a compound node passes its input values to its subgraphs, or when any subgraph returns values that become the results of the compound node. Dependence 2) will be seen usually in loops, when the loop values, calculated in the initialization subgraph, and values of the loop body subgraph are passed to the results part and back into the loop body. Finally, the control dependence is evident in the if-then-else structure called Select and iterative loops called LoopA or LoopB. A predicate

determines which expression to evaluate in a Select or when an iterative loop is to terminate.

3.7 Edges

Edges represent *explicit* data dependencies within the graph. Semantics of a compound node describe the implicit dependencies between the subgraphs, thus, not all data dependencies are explicitly given in the edge list.

The edge list begins with **E** in the *.ifl* file and each edge is 6-tuples. That is, they have 6 fields to describe their interaction. They are Source-Node, Source-Port, Destination-Node, Destination-Port, Reference to Type Label and Comments. The following line in the above *.ifl* file is read as :

```
E    0 1   1 2   4    %na=y %mk=V %sl=11
```

An edge having its source at Node 0 at its port-1, goes to Node 1 at its port-2. The reference to the **Type label** is 4 (Types not shown here). The Comments field are usually filled with *Pragmas*. Each pragma variable starts with a **%** sign and consist of a 2 character code. Pragmas are used to record the source line (**%sl**) of the generated edge, give a name (**%na**) to the edge and mark this edge 'by reference' (**%mk=R**) or 'by value' (**%mk=V**). There are more pragmas that the *osc* compiler may output in the *.ifl* line, than those described above, and their details can be found in the manual [8].

Literals are a form of edge with a difference, that, they do not have a source. They are 5-tuples having 5 fields, the extra fifth field being the *value* of the Literal. The *value* is indicated in double inverted commas as shown in the following example taken from the above *.ifl* file :

```
L          1 1   4 "1"  %mk=V
```

The above line is read as : a Literal going to Node 1 at its port-1 and having *value* of "1". Literals can depict the entry of Constants, Imported Functions and Boolean conditions into a Node.

Edges can also be viewed as being exported when leaving a node and imported when entering a node. Literals may only be imported.

3.8 Types

Type Description will always appear in the beginning of the *.ifl* file and is in the format :

T	Label	TypeCode	BasicCode	Comments
---	-------	----------	-----------	----------

First 6 **Labels** are reserved for the '*Basic*' **Type Code** which describe the edge as follows:

T 1	1 0	%na=Boolean
T 2	1 1	%na=Character
T 3	1 2	%na=Double
T 4	1 3	%na=Integer
T 5	1 4	%na=NULL
T 6	1 5	%na=Real

Once again, the comments are pragmas introduced by the compiler indicating the name of the edge. Here, the Type Code 1 represents a type called *Basic*. The **Basic Codes** describing the actual nature of the edge are in the fourth field, ranging from 0 to 5. There are variations in the above format depending on the type-descriptor for Arrays, Functions, Records and other such language constructs. Details can be referenced in [33]. In the above example of an edge **E**, in section 3.7, the number 4 in the field 'Reference to Type Label' indicates that, the edge is carrying integer values.

3.9 Graph Boundaries

Graph Boundaries begin with the line containing **G** and a **Type reference** followed by a **name** in inverted commas. A **Type reference** of '0' indicates that it is a subgraph and the **name** field is left blank, as in the file section of figure 3.3. Lines beginning with **X** (*exported function*) or **I** (*imported function*) have the same format and indicate graph boundaries.

The boundary denoted by a black line surrounding the nodes in figure 3.4, serves as the collector of imported and exported values for all edges inside it. The label of the boundary is always an implicit '0'. When a value is imported from a boundary, the Node receiving it has its source node as '0' and if the graph is returning values of computed

results, then these values are sent to node zero. Hence, a node whose destination node is 0 in the above *.if1* file section (figure 3.3), implies that, the node is exporting values to the boundary. Label zero is reserved for the graph boundary in the IF1 language specification.

Labels are unique only within a graph and if the graph contains subgraphs then the label definitions do not extend into the subgraphs.

3.10 Structures in IF1 -- An Overview

After getting a good look at the graphical form and its numerical representation, it is simpler to comprehend how the IF1 structures are actually organized. The IF1 structures are the central theme around which the Automater Tool is built. Refer to the IF1 graph in figure 3.5. This represents the outer block of the *matmult.if1* file. It is the Functional graph for matrix multiplication with the nested levels of the *ForAll* node not shown here.

The internals of IF1 are essentially organized into two main data structures : **1. Node-Structure (NS)** and **2. Edge-Structure (ES)**. There are other structures but, this discussion will not pursue these for sake of clarity. Details can be found in the source of SISAL from LLNL.

Outer wall of the above graph is a graph boundary for the Function and is represented as a Node-Structure. Boundaries to all the graphs and subgraphs are Node-Structures. Nodes that lie within a boundary wall, are connected to each other and the wall itself, via edges, which are completely characterized by Edge-Structures.

As can be seen, the Function has two nodes labeled N1 (*ForAll*) and N2 (*ASetL*). N1 receives five of its inputs from the boundary and N2 receives one edge from N1 and an input Literal. N2 also outputs an edge carrying results of the graph, to the boundary. This graph translates into figure 3.6. Here, the edges are shown.

Big rectangles are used to denote graph-boundary structures, small shaded rectangles represent Edge-Structures and the circular figures are the actual nodes within the graphs. Sequential labeling of nodes allows for each node to be a successor to the previously labeled node. Hence, the main Function Graph has a **node-successor** (*nsucc*) in *ForAll* which in turn has a *nsucc* in *ASetL*. The *ForAll* node is a complex structure and has subgraphs within its structure. This is reached by the **graph-successor** (*gsucc*) link. Each subgraph is connected with its previous one with a *gsucc* link. Subgraphs are Node-Structures, as described above, but they are differentiated from actual IF1 nodes by being interlinked with *gsucc*. Without Edge-Structures, that is, only with *gsucc* and *nsucc* links

present, this structure can be said to form a skeleton of the program. The skeletal graph is shown in figure 3.7. It represents the entire `matmult.ifl` program.

Body vessels that carry data are Edge-Structures, traversing in between the Node-Structures. All the edges that emanate from a node, are said to be exported from its structure. Thus, the **source** (*src*) of those edges is that node. Edges are **imported** (*imp*) by a node when they carry some value to that Node-Structure.

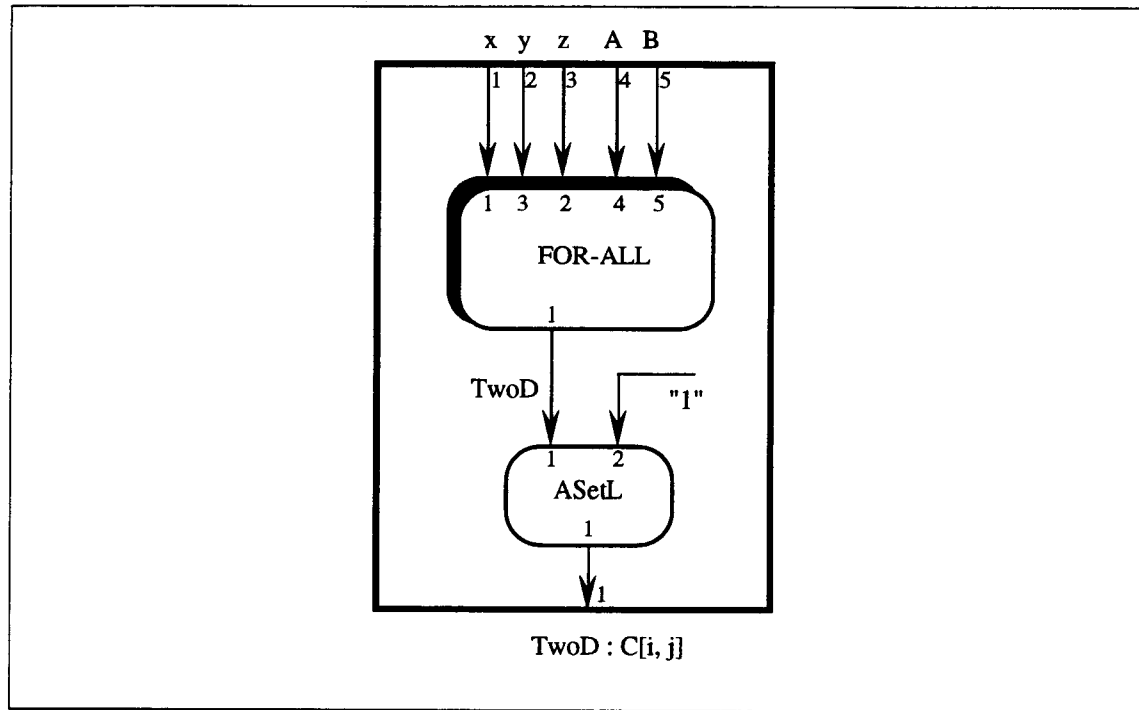


figure 3.5 Function Graph for `matmult.ifl` (Outer Level)

Accordingly, each of those Edge-Structure's has its **destination** (*dst*) as that NS, to which they carry values. An Edge-Structure is denoted by $[e(N_i-N_j); \text{value}]$. N_i-N_j implies the edge is exported by N_i and imported by N_j . The **value** field indicates the 'data value' that is being carried by the edge.

In section 3.6 describing Nodes for the `.ifl` file, it was mentioned that each node had a set of input and output ports on which it received and passed out values respectively. These ports are numbered. To keep track of the port numbering, the **export-successor** (*esucc*) and **import-successor** (*isucc*) fields are added to the Edge-Structure. In the figure 3.6, there are four out of five edges exported from the Function NS (Node 0) to the

ForAll node N1. Each edge is an *esucc* of the edge emanating from the previous port. The first edge is exported on port-1 and the others on subsequent ports become the *esucc* of its previous one like in a daisy chain. Hence, Edge-Structure ([e(0-1); y]) receiving values from port-3 of Function NS is the *esucc* to the ES that receives values on port-2 and so on. The same also applies to the import of edges. A good feature that is exhibited by the import-export fields is the handling of cross-overs. This can be seen in figure 3.5 and figure 3.6. An edge is exported at port-2 of the Function NS and is imported by N1 at port-3 then, such a cross-over in ports can be successfully depicted in the IF1 internal structure.

Literals are edges that have no source but have a destination node to which they provide some **constant** or **Called function**. This can be seen in the case of the *ASetL* receiving a Literal of "1" depicted by the ES [e(L);"1"] .

Figure 3.7 shows the complete IF1 file as it is interpreted from the *matmult.if1* file. Figure 3.8 is the skeleton of this representation.

3.11 Characteristics of the IF1 Structure

1. All Graph Boundaries including the subgraph boundaries are labeled Node 0.
2. Subgraphs of a Complex structure are linked by *gsucc* in a predefined fashion. That is, a Body Subgraph of a *ForAll* is always a *gsucc* to the Generator subgraph. The compound nodes and their predefined subgraphs will be elaborated in chapter 7.
3. When multiple functions are involved in the same *.if1* file, then Function-2 is a *gsucc* of Function-1 and so on.
4. Edges are exported (*isucc*) from one Node and imported (*imp*) by another Node. No edge has the same NS as its source and destination. Thus, no cycles are permitted.
5. All Edge-Structures have a destination (*dst*). Excluding Literals, all ES's have a source (*src*) in some NS.
6. Export-successor's and import-successor's depict the port numbering of the Node-Structures to which they apply.
7. All *successor* fields in the ES and NS have corresponding opposite going *predecessor* fields. These predecessor lines are not shown in the figure 3.6 to avoid confusion. That is, each *nsucc* line has an opposite going *npred* line and the same is true for *isucc-ipred*, *esucc-epred* and *gsucc-gpred* fields.

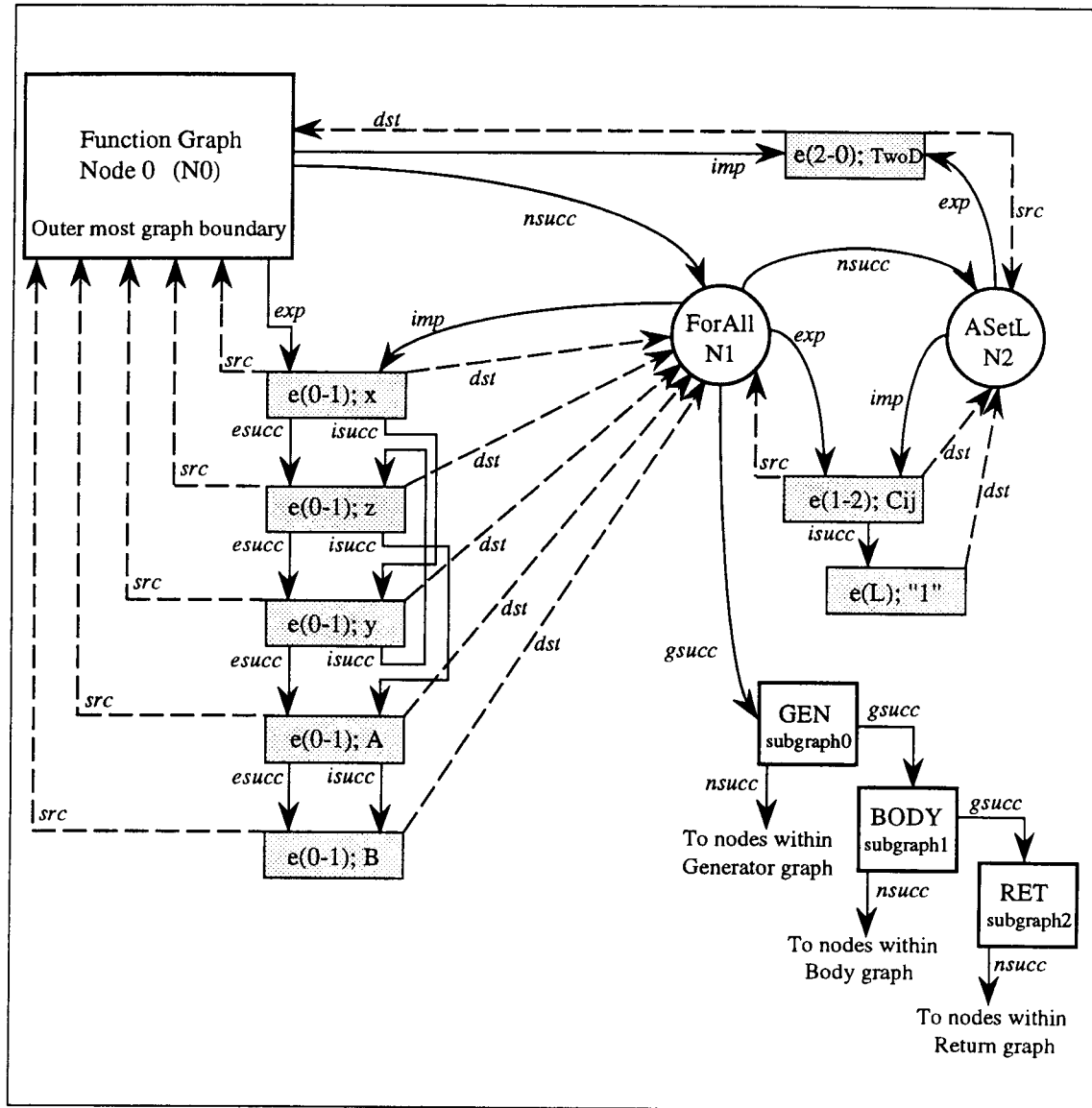


figure 3.6 Internal IF1 Structure for the Outer Boundary of `matmult.if1` file

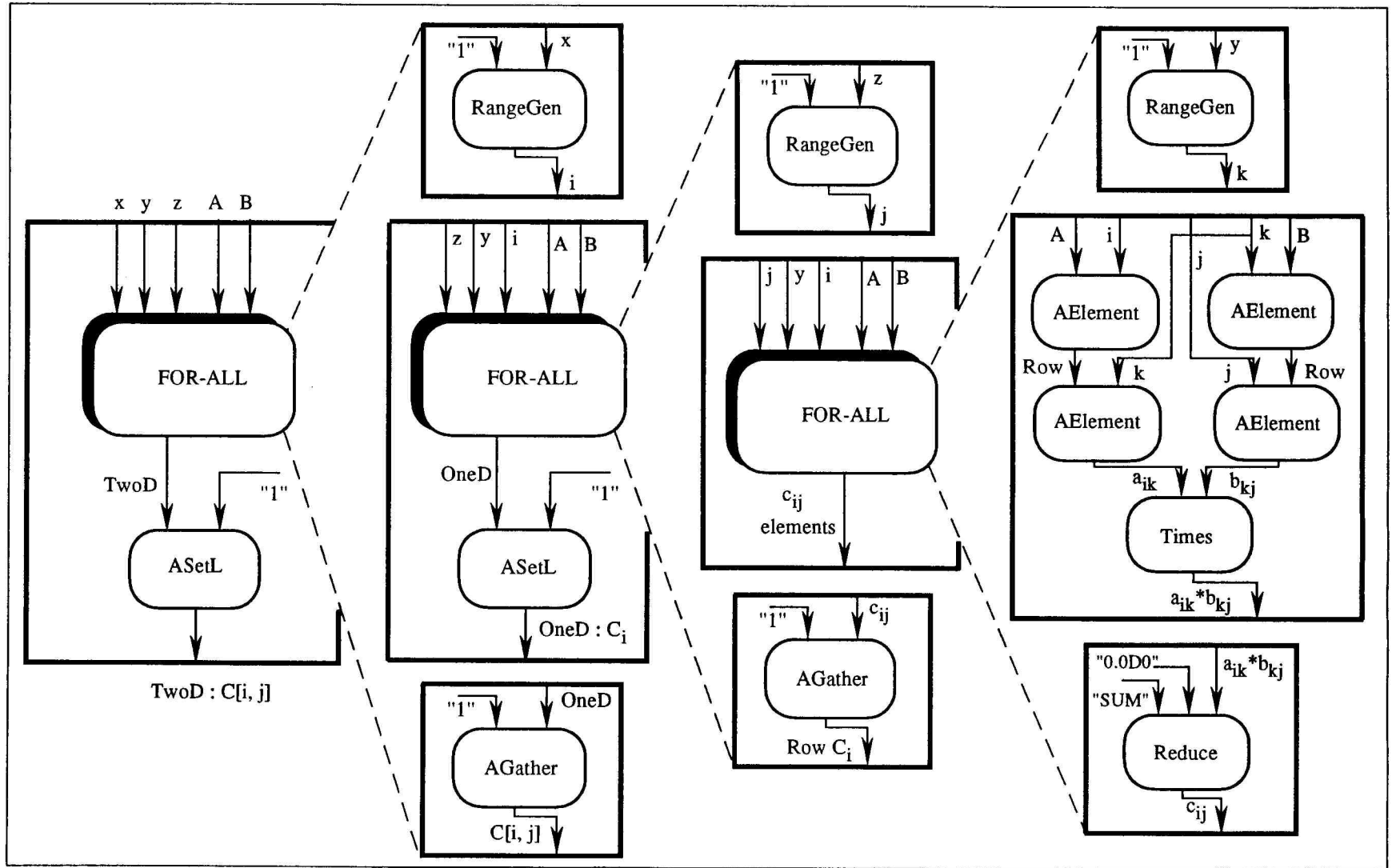


figure 3.7 The Complete IF1 representation of matmult.if1 file

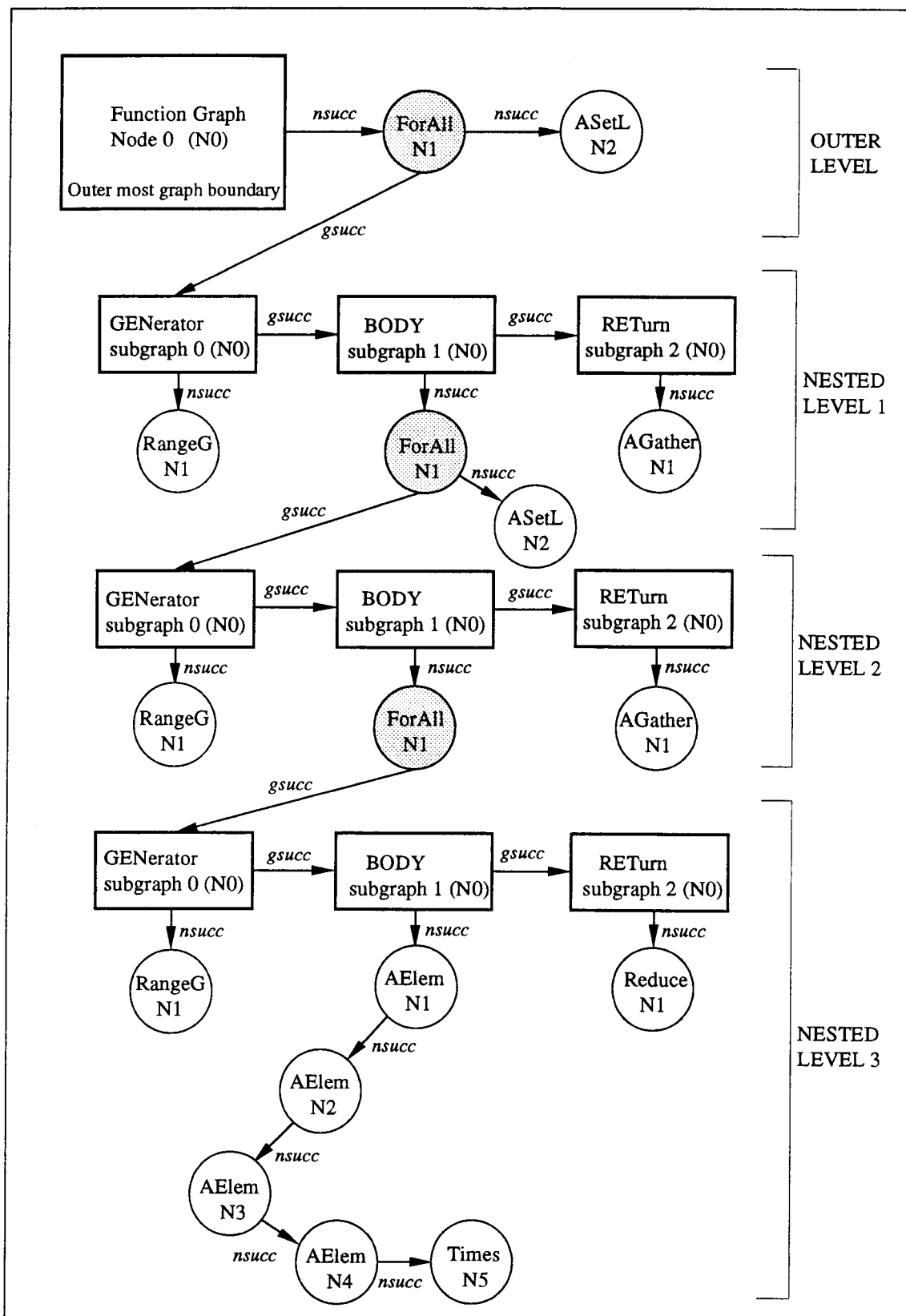


figure 3.8 IF1 structure for the entire matmult.if1 file

8. For a Node-Structure, fields relevant to this discussion are *exp*, *imp*, *gsucc*, *nsucc*, in addition to *type* and *label*. Each NS is characterized either as a Simple node, a Complex node or just a Graph Boundary for a Function or subgraphs. The *types* are represented as numbers in the IF1 specification. For simple nodes they range from 100 to 160 (inclusive), for complex nodes the *types* range from 0 to 5 (inclusive) and for graph boundaries the range is from 1000 to 1003 with 9999 being any *type* that is not defined. The *label* field is assigned in the *.if1* file, to sequence the nodes. Here, the *label* and *type*, correspond to same information as seen earlier, in the *.if1* section in figure 3.3.
9. The relevant fields for an Edge-Structure are *src*, *dst*, *isucc*, *esucc*, in addition to *iport* (import port identifier) and *eport* (export port identifier). The *iport* and *eport* carry the actual port numbers associated with the edges. These identifiers will not be used in the Automator implementation. Another field called *CoNsT* is used to carry the information for a Literal edge. Automator will use the *CoNsT* field to calculate the body count of the complex nodes.

CHAPTER 4. THE AUTOMATOR

4.1 Where does the Automator fit ?

The IF1 stage produces a graph that is acyclic in nature. This graph has nodes and boundaries that have implicit dependencies. The Automator package preprocesses the IF1 graph and presents a translated version to the Balanced Layered Allocation Scheme (BLAS). The Automator converts all the implicit dependencies to actual ones in its translation. The use of this tool will become more evident in the further discussion on how it goes about preprocessing. Hence, the Automator sits in between SISAL's IF1 stage and the input to BLAS as shown in the figure 4.1

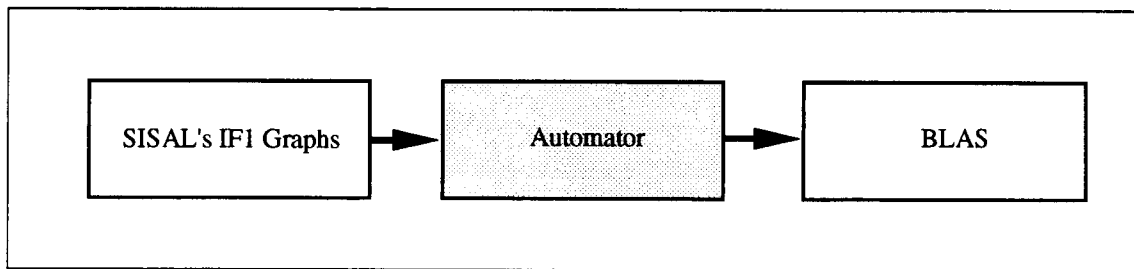


figure 4.1 Automator presents Linked lists to BLAS

4.2 Why is it required ?

BLAS is a static allocation scheme that partitions graphs to tasks at compile time. It then assigns these tasks to PEs in a given architecture based on the communication costs between co-operating PEs. The algorithm and its implementation will be discussed in chapter 5, on BLAS. In the current context, visualize BLAS as a black-box requiring a graph in its linked list format at the input. Before the Automator was constructed, only 'simple toy programs' were put to test using BLAS. These 'toy' program graphs consisted only of simple operations, lacking any of the complex language constructs like *if-then-else*, *do-all*, *case* and iterative *loops*. Further, these graphs were hand drawn and were limited in size to around 200 nodes.

With the Automator in place, 'real life' programs written and compiled in SISAL have been used to test the effectiveness of the allocation scheme. This has required

modifications in BLAS's implementation, which will be described later in the discussion of BLAS Chapter 5. There is virtually no limit to the size of the graph that can be generated, other than the space and time constraints of the computer running the Automator. A million node graph takes about 30 hours on a SPARC-10 processor. However, BLAS algorithm has complexity in the Order of n^3 where n is the number of nodes. This poses a severe limitation on real time allocation of large graphs, limiting the maximum size to below 10,000 nodes.

The name given to the *executable* file of the Automator is **blast**, as it "blasts" open the intermediate form. The line of commands to run the package and carry out a BLAS simulation are :

```
prompt% osc -IF1 matmult.sis
```

The above creates the matmult.if1 file.

```
prompt% blast matmult.if1 matif1.i
```

The .if1 file is the input and the output of the Automator is stored into the file matif1.i.

```
prompt% blas
```

Above runs the BLAS algorithm. When invoked, it will ask for a file to allocate, and the user will type in *matif1.i* at that prompt.

The user interface for the Automator will be described in a separate section. BLAS's user interface was re-done and will be described in the discussion on the Allocation Scheme.

4.3 Implementation

The Automator preprocesses a *.if1* file for the Allocation scheme. It has to present a graph in a linked list format readable by the BLAS package. Input format for BLAS is as follows :

Total_Number_of_Nodes

Node_Label Execution_Time Indegree -> Dest1 Dest2 Destn 0

Node_Label Execution_Time Indegree -> 0

The first line gives the maximum number of nodes that are in a graph. Second line gives the general format of how the nodes of a graph will be linked. The '->' operator is a delimiter, to the right of which is a list of destinations '**Dest#**' of that node, each separated by a space. **Node_label** is a node number assigned by the Automator and is not

to be confused for any IF1 labeling. End of the linked list is marked by a sentinel '0'. In the third line it indicates that the node is an exit point for that graph as it has no destinations. **Indegree** specifies the number of arcs that are incident to a node. Also, it gives the number of times a node appears in the destination fields of other nodes.

The Automator reads in the IF1 structure, and reorganizes it by removing all implicit dependencies that exist between graph boundaries. To examine the implementation process, the same example of *matmult.if1* is taken. Refer first to the outer block or the Program Function Graph in figure 3.5. It has a *ForAll* followed by *ASetL*.

Any graph has an entry point called the *Root* and a single or more end points called *Exits*. For BLAS, a graph has to have one *Root* and one *Exit*. Such a node may not exist, as a graph may have multiple exit points. But, in this implementation of the Automator, all logical end points are forced to converge at a single Exit. The introduced Exit node is assigned a zero execution time. Hence, the name *dummy node*. By default the Automator labels node 1 as the dummy Root and node 2 as the dummy Exit. Thus, labels for the Root and Exit of a graph are always known in advance.

The graph is organized in an array called **Main-Array** (also called *Main*) where each element of *Main* is a structure representing a node. The node has fields to indicate its label, execution, indegree and a pointer to a linked list of destinations. Details of the package can be referred in the source : 4100 lines of C code with elaborate commentary. Only the important aspects to aid in the understanding will be mentioned in this discussion.

The Automator traverses the *nsucc* line and processes node after node from the IF1 structure. Each node's :

- 1) *imp*, *isucc* and *src* lines give the **indegree** information,
- 2) *exp*, *esucc* and *dst* lines give the **destinations** of that node and
- 3) *type* information gives the IF1 type, namely: Simple or Compound.

The Function Node-Structure is broken down into a Root and an Exit for the *matmult.if1* graph. Once the NS is processed as element 1 of *Main*, the fields in *Main* are manipulated and Root at label 1 and Exit at label 2 are created. Then the *Main-Array* is filled as shown in Table 4a. In general, Literals are only considered when a range is being specified. Range specification occurs when two Literals appear for a RangeGenerator node. Hence, *ASetL* has only one indegree here.

Consecutive elements in *Main-Array* do not imply any flow-dependency. That is, node 2 occurs before node 3 in the array and is the Exit node in the graph. It will be seen in following figures that, lower node numbers will often be the destinations of higher

node numbers. Thus, all the figures, where arrows are not explicitly shown imply a top to down flow of the graph. Also, in the above Table 5a, Root has 5 arcs going to node 3, but only one is shown here for clarity as it would mean the same.

lastNode				
Index	1	2	3	4
Label	1 : ROOT	2 : EXIT	3 : ForAll	4 : ASetL
Indegree	0	1	5	1
Exec. Time	0	0	Not Eval.	4
flag	0	0	1: Set	0
Destinations	3	0	4	2
Destinations	0		0	0
Destinations				

Table 4a Main-Array Organization.

If a node is Compound, like N1 is a *ForAll*, then a *flag* field of element 3 in *Main-Array* is set to 1. After all the nodes are traversed in the outer block, the Program Graph is said to be processed externally. It is time to check and process compound nodes from within, in the externally processed Graph.

The graph is retraversed and this time only the *flag* field is checked for compound nodes. On encountering the first compound node, it is entered using the *gsucc* lines to get to all the sub graphs that lie within. Refer to figure 3.7. Each sub graph is then processed as a fresh graph having nodes within. All sub graphs are not treated the same and this will get clearer with the discussion in the following sections. A variable called *lastNode* keeps track of the last filled element of *Main-Array*. The first encountered sub graph is placed after the *lastNode* in *Main* and as each node or NS is processed, the *lastNode* is incremented. This cycle keeps repeating until all sub graphs and finally all complex structures have been decomposed from the *Main-Array*.

4.4 Multiple Instances

A *ForAll* node has 3 sub graphs : **GEN** for Generator, **BODY** and **RET** for Returns. The GEN graph produces multiple values, which implies the replication of the BODY

sub graph. The RET sub graph gathers data from multiple instances of the BODY graph, to give the final output of a *ForAll* Node. Its correspondence in SISAL can be seen in the program segment of figure 3.2:

```

Cij := for k in 1, y
      returns value of sum A[i, k] * B[k, j]
end for
```

The GEN sub graph generates the range of k from 1 to y , the BODY contains the expression 'A[i, k] * B[k, j]' and there are 'y' such bodies. Finally, the RET is indicated by the statement **returns value of sum** collecting the results from the 'y' BODY sub graphs and 'summing' them.

Assume in the matmult.sis program in figure 3.2, that the body-count is 2 indicating a matrix multiplication of 2x2 matrices. That is $x = y = z = 2$ in the SISAL program. In the *Main-Array*, the lastNode is at label 4. The *ForAll* is at label 3 so after reading its flag, the node-to-expand variable is 3. A procedure to expand this *ForAll* is called. This procedure will start filling *Main* from the next empty element, which is 5. This position is called the RETurn NS. The destination links of label 3 which are destinations of the original *ForAll*, now become the destinations of the RETurn node. The indegree of RET is equal to the body count of the *ForAll* (which is 2) and its execution is the cumulative execution of the nodes that are in the RETurn sub graph. This is how the RET sub graph is treated.

As the *ForAll* node is being disintegrated into three sections, the implicit dependencies that existed within each sub graph will now become actual dependency arcs between nodes. Hence, the destinations of the original *ForAll* are actually the destinations of its last sub graph. In a way, the outer shell is being discarded and the whole complex structure will undergo a *metamorphosis*.

Indegree of the RET node is equal to the number of body nodes and that is 2. There is only one simple node **AGather** in the RETurn sub graph and its execution time becomes the execution of RET node.

The next two places in *Main-Array* are reserved for BODY nodes of the *ForAll*. They are 6 and 7. Once the RET node is processed, the GEN node is created at the original position of *ForAll*, that is, at label 3. The indegree for GEN is that of the original *ForAll* node. Its destinations are the two bodies at 6 and 7 and its execution time as before, is the cumulative execution of the nodes in the GEN sub graph. In this case only

one node : *RangeGenerator*, exists in the GEN sub graph. Any *ForAll* structure will have at most one node in its GEN and RET sub graphs. This is from observation.

Finally, the BODY nodes need to be processed. Bodies of the same compound structure in question, will have identical internal nodes, each receiving the same number of inputs (usually one) from the GEN node. However, each BODY is a separate instance. Also, better described as, the same expression for a different variable, at each instance. In the figure 3.2, successive values of *k* will cause successive elements of the matrices to be evaluated.

The number of inputs to a BODY, from a GEN is one, as all the inputs of a GEN are routed to its corresponding BODY sub graph and the Automator groups the edges and considers it as a single edge, without loss of any data in the graph. This is an assumption based on the premise, that values are readily available at all times at a node and no memory fetches are explicitly carried out. This assumption is seen to cause one of the primary limitations when mapping IF1 nodes to actual multicomputer systems. That is left to a discussion on the Mapping of Nodes in section 4.16.

Indegree of a BODY is 1 and its destination is always the RET node label, as it evaluates the expression and sends it to the RETurn sub graph for packaging. A BODY node may contain none, few or many nodes. These nodes may be simple, compound or a mixture of both types. Thus, 3 cases arise :

Case1. Only Simple Nodes -

In such a case the execution time of the BODY node is the cumulative sum of all execution times of its internal nodes.

Case2. One Compound Node and None or Multiple Simple Nodes -

The execution time will be ascertained by recursively expanding that compound node, as has been explained, by traversing its sub graphs via its *gsucc* and *nsucc* links, and placing the processed nodes after label 7. The simple node execution times will be cumulatively summed and added to any node from the compound structure that will occupy the present BODY. The present BODY undergoes a metamorphosis to become a GEN node of a nested *ForAll* (at Level 2). As seen in figure 4.2, the execution time for this node is that of the *ASetL* and the new GEN2.

Case3. Multiple Compound Nodes and None or Multiple Simple Nodes -

This is a special case. It is treated like a new Function Graph where each of the simple nodes if any, and compound nodes occupy a distinct place in the *Main-Array*. This is analogous to re-processing a new Function Graph as is being done with a *matmult.if1* in the above example. This case will be elaborated further with reference to an example in the section 4.6.

The BODY in the above example fits the second case and the program is actually a three nested-level graph. The complete metamorphosis are shown in the figure 4.2 and figure 4.3 in Steps I, II, III, IV and V respectively. It is clear from step III and IV that, after external processing is completed, each compound body is traversed right through all its nested levels, before expanding the other compound body in the same level. Hence, on expanding node 6 in level 2, the Automator expands node 9 and node 10 in that order in level 3, before coming back one level to node 7.

In the above transformation, multiple BODY nodes will replicate the information processed for the first BODY that is reached from the GEN node. This is possible as the bodies of a compound structure will be equivalent to each other. In the Automator, a feature called **Automate** to do this replication is included. It will be described in the section of **Features**.

Thus, the example cited here takes us through a widely used matrix multiplication algorithm. It has three nested levels that are implemented using the *ForAll* Structure. The actual computation is done in the third nested level. The Automator is capable of **blasting** such acyclic, multiple unconnected, IF1 representations into a connected static graph that can be presented to a system for allocation. Hence, a *matmult2.if1* file was blasted to give a *matmult2if1.i* ('i' indicates an input file and 'matmult2' is for 2x2 matrices).

4.5 Concentric Circles

The issue of removing all implicit dependencies is as important as dealing with each of the complex nodes, their nested levels and the different combinations of nodes, that exist within each nested level. The Automator treats any program as a composition of Concentric Circles. Refer to figure 4.4. The outer level may have any simple, compound

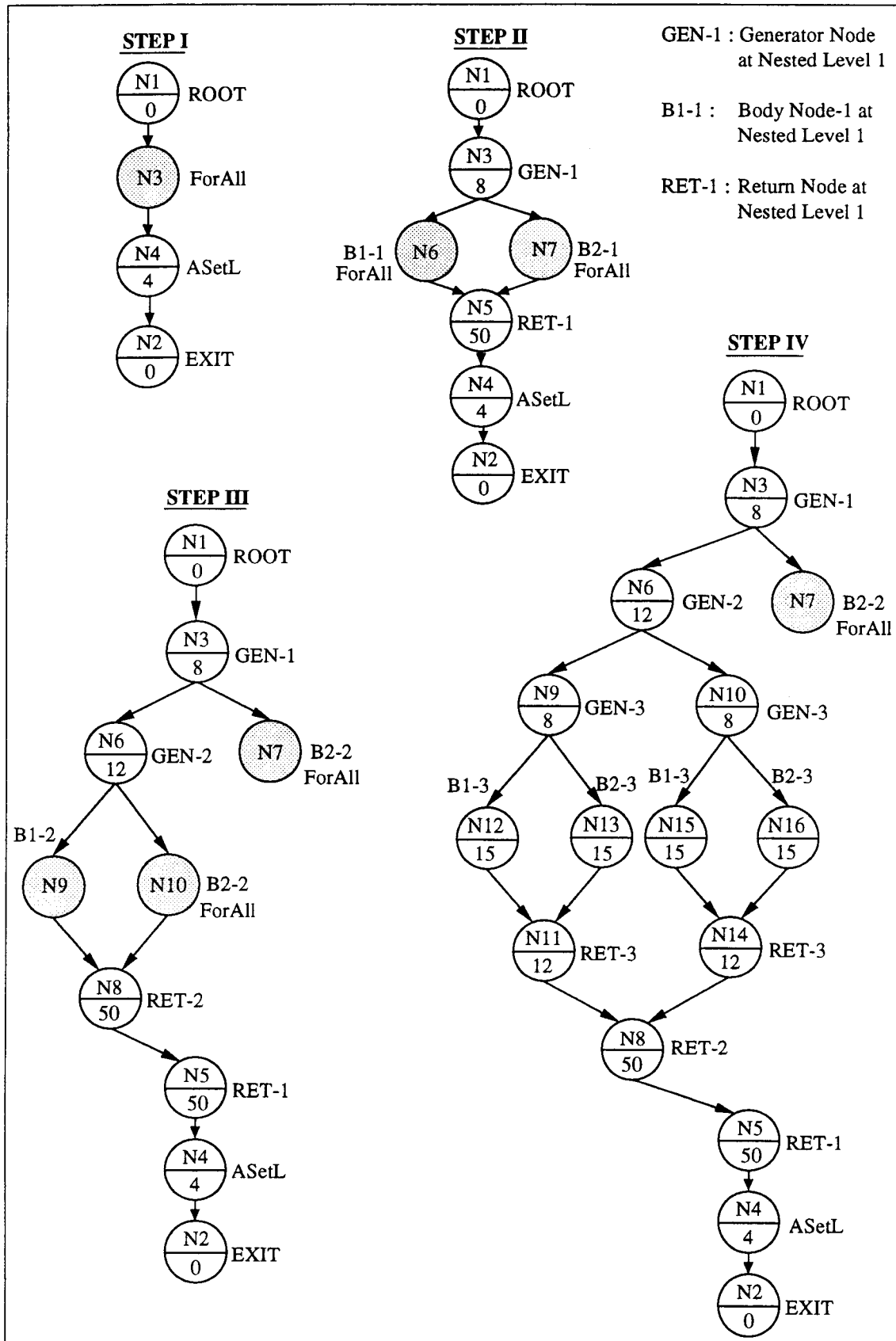


figure 4.2 STEPs in the metamorphosis of matmult2.if1

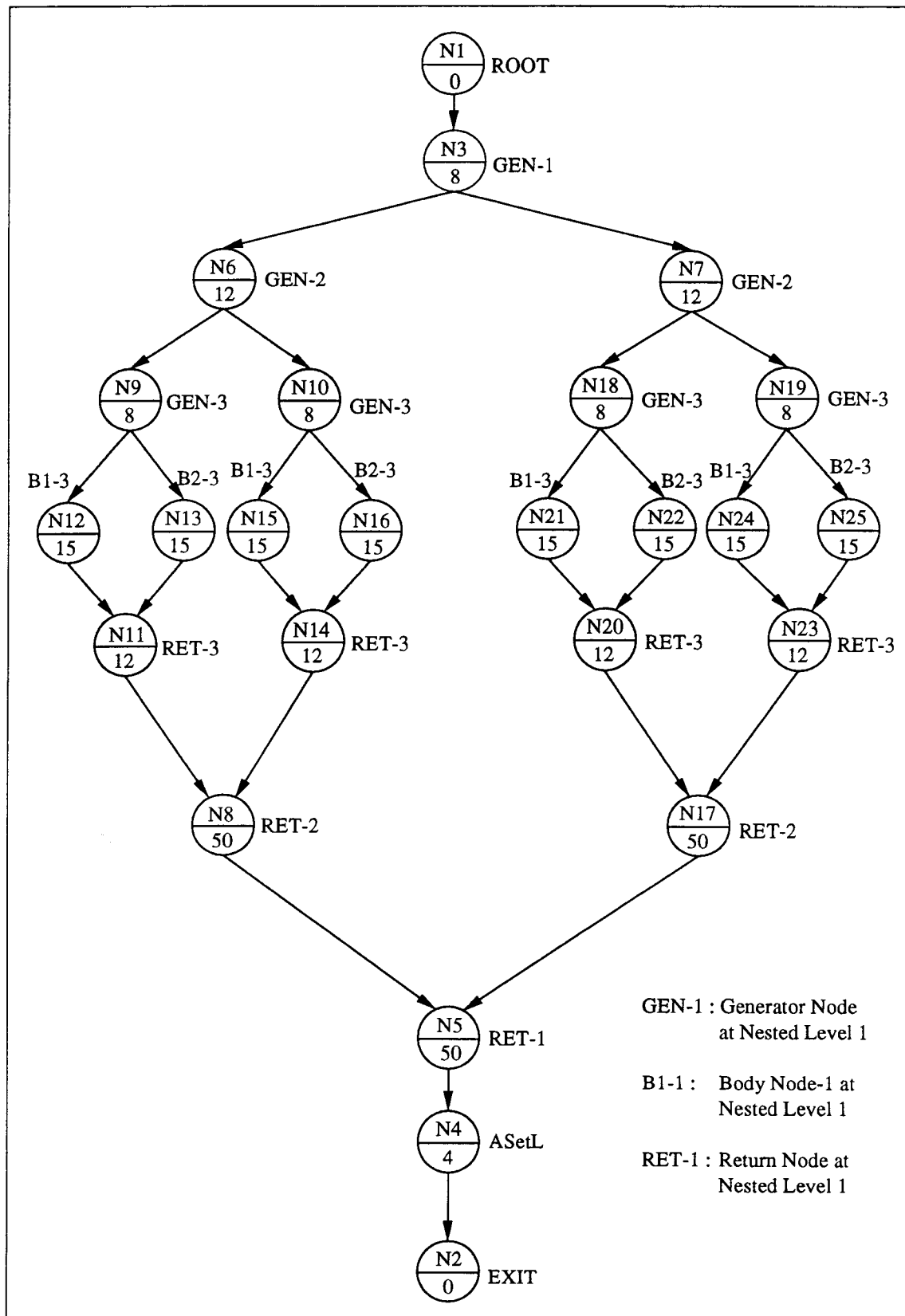


figure 4.3 Final **STEP V** in the metamorphosis of `matmult2.if1`

or a mixture of nodes, and so will any level within it, as shown. This gives rise to a great number of combinations that can be handled.

A real program graph may look like :

Outer Level	<i>ForAll</i>
Nested 1	<i>ForAll</i>
Nested 2	<i>LoopA</i>
Nested 3	<i>Select</i> (has a True sub graph and a False sub graph)
Nested 4	True Graph of <i>Select</i> has 2 nodes : <i>LoopA</i> and a <i>ForAll</i>
Nested 5	<i>LoopA</i> has a <i>ForAll</i> which has no more Nests
Nested 6	<i>ForAll</i> from Nest 4 has a <i>Select</i>
Nested 7	True from <i>Select</i> has simple nodes only
Nested 8	False from <i>Select</i> has simple nodes only
Nested 9	False Graph of <i>Select</i> at Nest 3 has another <i>Select</i>
Nested 10	True from <i>Select</i> has a <i>ForAll</i> which has simple nodes only
Nested 11	False from <i>Select</i> has a <i>ForAll</i>
Nested 12	<i>ForAll</i> has another <i>ForAll</i>
Nested 13	<i>ForAll</i> has a <i>LoopA</i> with no more Nests
Outer Level	<i>LoopB</i>
Nested 1	<i>LoopA</i>
Nested 2	<i>ForAll</i>
Nested 3	<i>ForAll</i>

Example of Concentric Circles as shown in figure 4.4

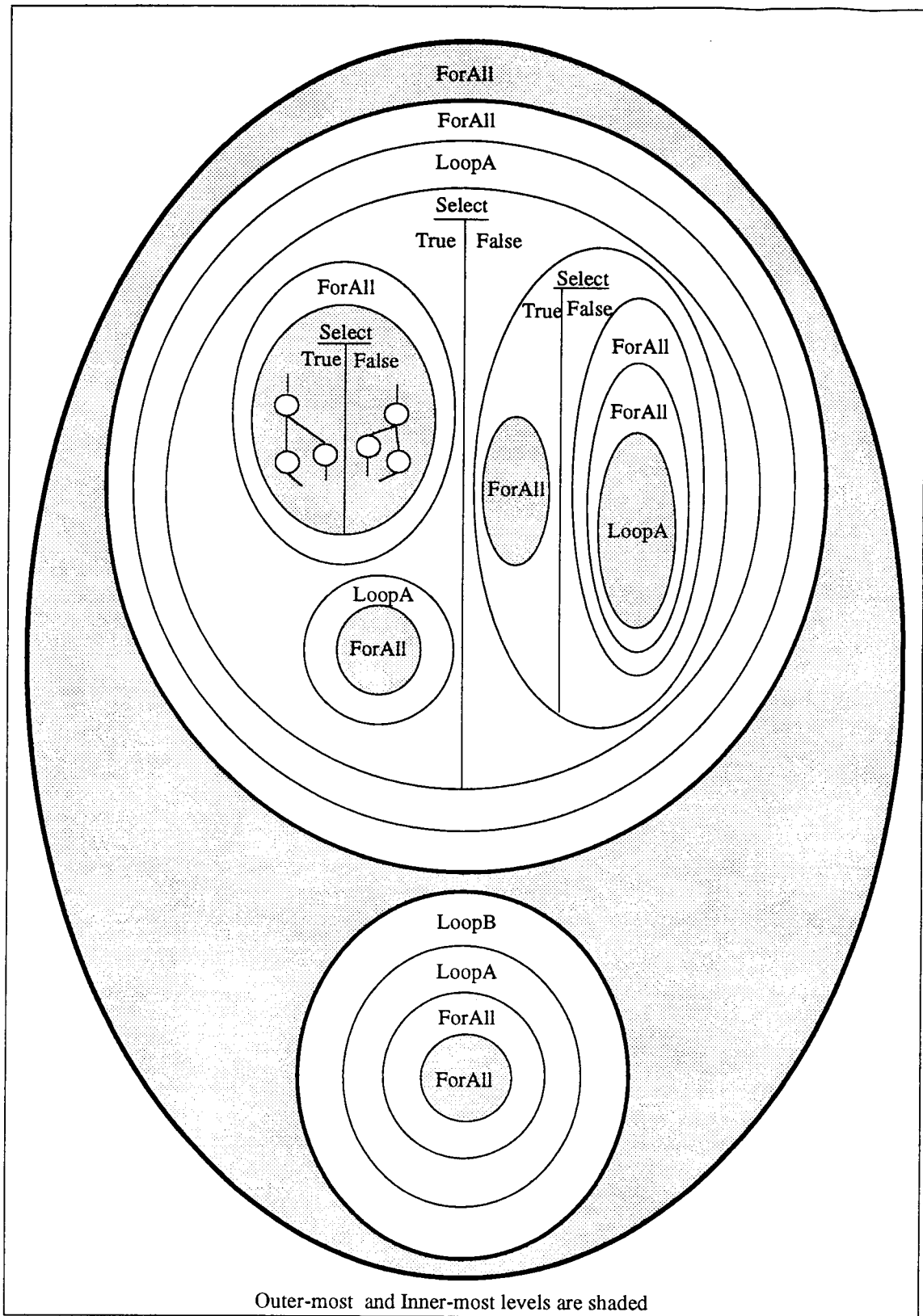


figure 4.4 Concentric Circles.

The Outer Level has two nodes : a *ForAll* and a *LoopB*. The *ForAll* has 13 nested levels and *LoopB* has 3. The graph can get real complex. IF1 phase creates individual graphs for every compound node, each with its own boundary wall. The Automator, using a well known process of recursion, makes a connected static graph from all the individual graphs. Code for the Automator is well documented, highlighting the aspects of recursion that are employed.

Any nested level is spawned off only from the body sub graph of a compound node. Other sub graphs either do a test, initialization, a range generation or return of results and other such simple operations. In case of *Select* compound nodes there are True and False sub graphs. Each may have nested levels. These compound nodes and their behavior as exploited by the Automator, will be described from section 4.9 onwards..

4.6 A Special Case

A case in which a body sub graph contains multiple compound nodes and none or multiple simple nodes (refer to Case 3 above), is treated as if it were a new Function Graph. Consider the example (left) given in figure 4.5. The *LoopB* body has 2 compound nodes : *LoopB* and a *ForAll*, and 3 simple nodes, in the sequence : *LoopB*; *ForAll*, *Plus*, *Minus*, *Times*. Also, unlike above, each of *LoopA* and *ForAll* have no more nested levels within.

The Automator appends two dummy nodes : *StartBody* and *EndBody*, analogous to the Root and Exit of a new Function Graph. The compound nodes are treated and further nesting is checked as they are approached. An important difference here, is that the simple nodes enjoy a separate place in the *Main-Array*. Thus, their execution times is not just summed up as before.

4.7 Node Packing

Any sub graph is a Node Structure and is translated as a node by the Automator. The sub graph contains nodes within. The Automator packs up a node by cumulatively summing all the internal simple node execution times. This feature of the Automator is also called Grain Packing. The dependency arcs within a sub graph, are dissolved when packing it. This is done for all cases except when multiple compounds exist in a particular body (Case 3).

The idea of grain packing is to combine multiple fine grain nodes into a coarse grain node if it can eliminate unnecessary communication delays or reduce the overall scheduling overhead in an allocation scheme. Fine grain nodes are the individual simple nodes that imply operations. Thus, all fine grain operations within a single coarse grain node are assigned to the same processor for execution. A fine grain [15] partition of a program will demand more inter processor communication than that required in a coarse grain partition. Hence, such packing introduces a tradeoff between parallelism and scheduling overhead. Coarse grain nodes can also be viewed as *Light Weight Threads* of computation. The weight is the summed execution times of the node. A *thread* is described as an individual fragment of computation, where a fragment consist of multiple operations, all linked together.

In the discussion of BLAS, it will be clear, that grain packing also helps in a faster execution of the BLAS algorithm, as the nodes are fewer.

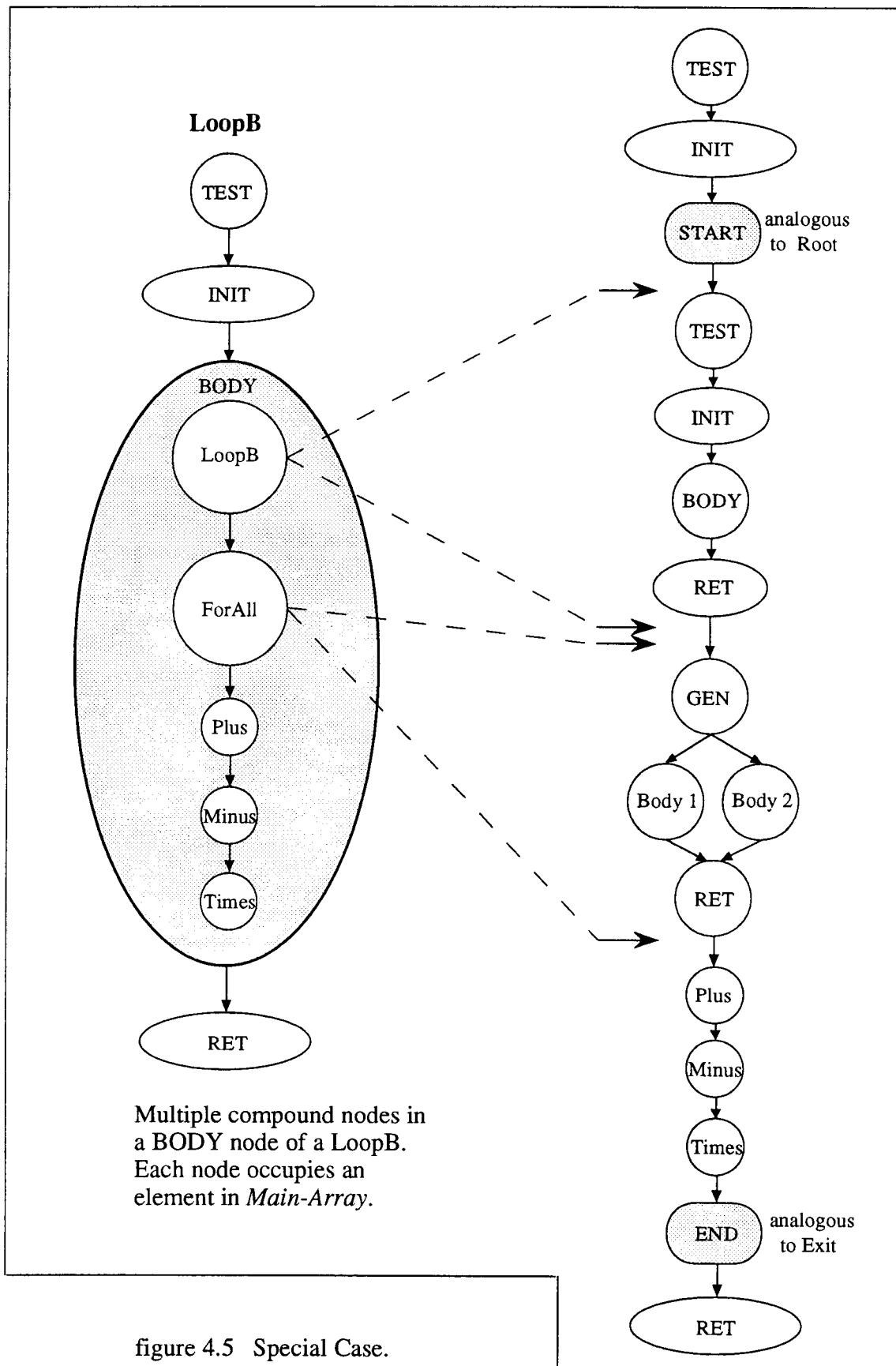
Thus, sub graphs are treated differently depending on what they contain.

4.8 Execution Costs of IF1 nodes

Since each node implies an operation, it has an associated execution time. The SISAL group assigned an execution time to each node. These times are an approximation. A large number of systems, primarily shared memory, were studied for correspondence between any node and assembly operations involved in executing that node. These results were averaged out to give the execution times of the nodes. These times, though representing shared memory processor costs, can be approximated very closely to message passing processor costs except when referencing memory.

The execution table is organized into **two** sections : The *Cost* of doing an operation and the operation *Time* of nodes. Thus if an operation is **Plus**, then depending if it is an integer or float operation, a corresponding cost is added to the time of operation. That is, 1 cycle for integer and 4 cycles for floating point operation of the same node. Based on this, each operation has an associated cost. Similarly, an *Array Copy Cost* is associated with some node that builds arrays and if the array consists of *real* elements, then any operation on the elements will have to bear the *real* cost. The costs and times may be referenced in the source for the Automator.

The costs assume a processor architecture. When examined closely, these costs and times represent RISC (*Reduced Instruction Set Computers* [14 and 16]) class processors. Mapping of SISAL nodes to a particular architecture will elaborate on the exact



processing element to be used. However, one major difference is the memory access schemes that are prevalent in shared memory vis-a-vis message passing computers. As the costs are an approximation of the former, performing static scheduling on message passing systems with these costs will not give a true picture for all cases. Hence, Mapping becomes a very important next step to this project. Some ideas are presented in this work for mapping, but its complete treatment is out of scope in this thesis.

4.9 Automator's way with complex constructs

This section will elaborate on the important aspects of compound nodes and their treatment by the Automator. There are in all 5 complex structures which are relevant to the Automator's implementation. These are *ForAll*, *LoopA*, *LoopB*, *Select* and *TagCase*. The *LoopA* and *LoopB* are explained under one heading of Loops. This discussion will highlight some of the important features built into the Automator and how the User-Interface looks when preprocessing graphs. The **GOAL** of the Automator is '*to achieve vertical connectivity with horizontal expansion using recursion on the philosophy of Concentric circles.*'

4.9.1 ForAll

The *ForAll* node also called *product form loop* [22] is used to denote independent as opposed to iterative instances of an expression. It is the most parallelizable construct in the language. It has three sub graphs : GEN - generator, BODY - body, RET - returns. The BODY will contain the expression to be evaluated. The generator produces values for each instance of the body as seen earlier. The nodes, used to generate values are either the *AScatter* or *RangeGenerate* nodes. The *AScatter* node [33] takes arrays or streams as its inputs and places its elements at one of its port and the index value to that element on the other port. The *RangeGenerator* node [33] generates a sequence of integers in the inclusive range of its inputs and places them on its output port in a sequence. Each such integer or element is send to a distinct instance of the body and are not broadcast to all instances of the body. Thus, the values and the instances are ordered and the results of the subgraph, RET, will gather it in the same order. This order is not an order in time, but in semantics specifying the ports.

4.9.1.1 Body Count

If the *RangeGenerate* has two constants (numbers) as its inputs then the Automator figures out the body count of the *ForAll* node it is processing. If any input to the *RangeGenerator* is a character implying an integer constant, then the user needs to give the body count. This requires a user to know the program profile before subjecting it to the Automator. The feature of automatic calculation of the body count is called *RunOff* and is a User-Interface option. It avoids any user interaction with the translation process. This feature can be used when translating various *matmult.sis* files where-in replacing variables x, y, z with actual numbers (all same). The results section for matrix multiplication was generated this way.

4.9.2 User-Interaction

Interaction with the Automator during preprocessing is enabled in 3 ways :

1. At the processing of *each* compound *ForAll*, the user can make a decision of whether to expand that *ForAll* or not. For either case the user specifies the body count. This is a tedious approach and is best suited for experimentation or to study the behavior of program characteristics. The no-expansion option is also for experimenting on the choice of grain size during scheduling. The compound structure may not be expanded but, other nested compound nodes in subsequent levels may be expanded. This requires that the body count be reflected right through the nested structures. It is accomplished by multiplying the execution time of every node that lies within the boundary of a non-expanded compound node, with the body count. Hence, assuring the right weight of each node.
2. Specifying *at start*, in the User-Interface, the behavior of compound structure with respects to expansion or no-expansion. Then, throughout the translation that option will hold. But, the body count will still be queried each time from the user. The default on expansions of any compound structure may be enabled in the program code itself. This will disable user interaction completely for that option. While generating results for this thesis, the *ForAll* node was defaulted to 'expand always'.
3. If the *RunOff cannot be enabled*, but the program profile is known to have a compound node with a particular body count throughout the program, then that body count may be entered during the **Query-Time** phase of the User-Interface. This will allow the translation to proceed without any interaction. A feature of same body

count through the program, is usually applicable only with matrix based programs. All the example simulations presented in the results section are matrix based.

The concepts of user Interaction and Body count, as explained for *ForAll* nodes are also applicable to *LoopA*, *LoopB* and *Select*, with some changes. These changes will be highlighted in their respective discussions.

4.9.3 Loops - LoopA and LoopB

The above two are considered together as they come under the class of Iterative constructs. These loops introduce carried values in every iteration and are primarily sequential in nature. Some exceptions exist but, thorough profiles are required to establish their parallelization.

The Loops have four sub graphs : INIT- Initialization, TEST, BODY, RET - returns. The body and returns sub graph are identical to the *ForAll* in construct. *LoopA* is the iteration compound that tests for termination of the loop after the body has executed at least once. *LoopB*, in contrast, performs the termination test immediately after giving initial values to the loop body. When the result of the TEST is false, the Loops make its results available at the output ports.

These loops cannot be parallelized most of the time because *loop-carried dependencies* are introduced with every iteration. That is, old values from the previous iteration will be required in the present iteration and this forbids parallel execution of the loop.

Thus, the Loops need not be expanded by default. But, options are available for expansion as explained in the section of Interaction in *ForAll*. However, the body count (**bc**) of a non-expanded Loop will apply differently depending on the compound Loop. The execution times are calculated as

$$\text{LoopA.exec} = \text{INIT} + \text{loopA_bc} \cdot (\text{TEST} + \text{BODY} + \text{RET})$$

and

$$\text{LoopB.exec} = \text{INIT} + \text{loopB_bc} \cdot (\text{TEST}) + (\text{loopB_bc}-1) \cdot (\text{BODY} + \text{RET})$$

The Automator treats both constructs in separate procedures so that different variations for each type of loop can be sought, and no generalization for one construct will apply to the other.

4.9.4 Select

A *Select* node is analogous to the *if-then-else* (conditional branch) construct of any imperative language with one big difference. In this implementation of IF1, it represents only a two way selection, that is, no *case* or *elseif* constructs are supported.

The *Select* consists of 3 sub graphs : TEST - selector, TRUE - true alternative, FALSE - false alternative. The selector returns one Boolean value : either a 0 or a 1. The '0' maps to the false sub graph and '1' maps to true. All input ports to the *Select* node are connected to corresponding ports of all its sub graphs and the result ports of each true and false graphs' become the output ports of the *Select* node. These implicit dependencies give rise to 3 cases when implementing a *Select* node. Each of the three cases are only possible either if the program profile is already known or if a generic assumption is made. They are : Referring to figure 4.6.

Based on known program profile :-

1. *Select* Test outputs a 1. This corresponds to the Selector followed by the True node in Case I.
2. *Select* Test outputs a 0. This corresponds to Case II in figure.

Based on generic assumption :-

3. Case III - Processing the True and False graphs with probabilities of occurrence [18]. These probabilities are obtained from examining the program profiles with regards to whether a True or False section was executed.

The probability of a True case occurring is in the range of 0.5 to 0.8. However, in this project it is assumed that the True graph is executed 80% of the time. In this scheme, the probabilities are applied right through every nested level of the True and False nodes. If a True node had three nested levels and its probability of occurrence is 80% then, all the nodes in the nested levels will have their execution times multiplied by 0.8 and those in the False node will be multiplied by 0.2.

There is a potential limitation with BLAS in this regard. This probability should also apply to the edges between the nodes when traversing through the *Select* structure. Edges can be assigned a probability just as nodes have been assigned execution times in the Automator. However, BLAS needs to be revamped to accommodate this feature. The simulations have been carried out without considering edge probabilities, but future work must look into this important aspect.

Selects may be treated as non-expanded nodes just like the previous constructs. All the nested structures however, may be expanded. This is possible for only cases 1 and 2 as no probabilities are assigned in these cases.

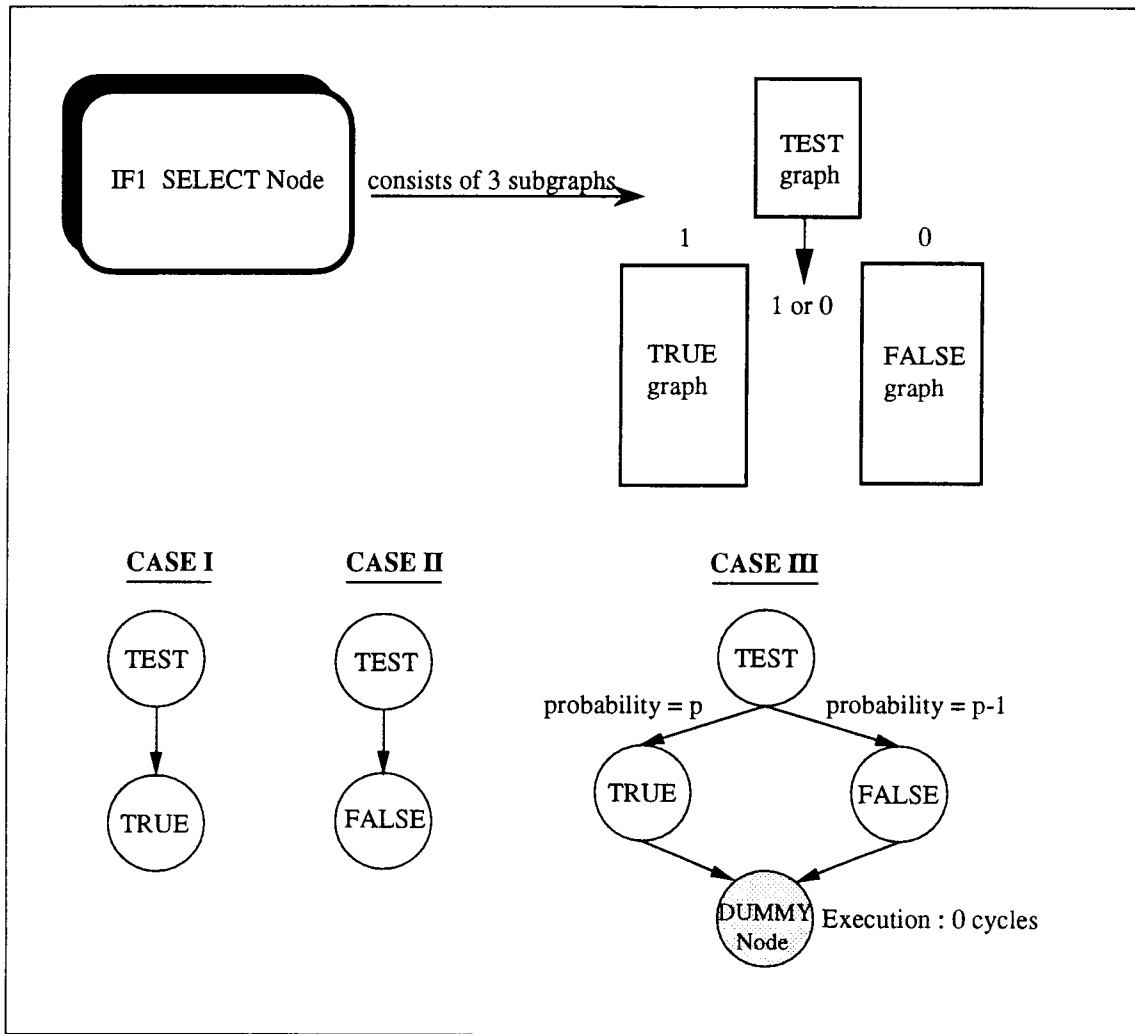


figure 4.6 Automator's implementation of the IF1 Select Structure

4.9.5 TagCase

The Automator does not expand this structure by default. No options are available in the User-Interface, as this structure is a rare occurrence in IF1 files. It is used to access fields of a union object. Details on its implementation are given in [33] and the source for the Automator.

4.10 The User-Interface

The User-Interface looks as shown below in figure 4.7. Additional features of the Automator will be briefly described. Details on all implementation issues can be read in the comments section of the source code.

The Automator uses the IF1OPT stage for translation because :

1. The optimizations are machine independent.
2. Multiple functions of the program are expanded out into one 'main' function. That is, CALL nodes are replaced by the function module it calls. The Automator does not have to deal with multiple functions and it translates one monolith function file.
3. Since optimized, the number of nodes are reduced and this is advantageous when applying BLAS.

4.11 Features

1. As the Automator progresses, the trace of the translation flashes by on the screen and a hard copy *file.tr* may be obtained for verification with the interface option. This is very helpful in debugging and verifying the correct translation. It states every node that is encountered and processed. The output of the Automator will give the connectivity of nodes and the trace file will show the node's type, label and which nested level it is in. The above information saves one the trouble to decipher the *.ifl* files in any case.
2. Addition to *Grain packing*, a feature called *Compaction* can be enabled during translation. Nodes A and B may be compacted only if Node A has one destination in Node B and Node B has only one indegree that coming from Node A (in short, Node B is the only successor to Node A). Thus, a series of nodes obeying the above conditions may be compacted into one node. The compaction process is done

```

/*****
AutoMator
Revision 131 1993/Dec/01  MANOJ RAISINGHANI
*****/
LOOK FOR .opt FILES IN THE opt SUB-DIRECTORY

DID you want a Trace File of this Run ? (y/n) : y
If YES then give the Trace File Name : lu5
The Trace will be in file lu5.tr

ENABLE COMPACTING of GRAPH (y/n) : y

If this is a mat#.opt or mat#.if1 file then ....
USE 'runoff' feature to avoid user-interaction with the RUN
Type 1 for a runoff OR 0 for Normal Interaction : 0

THE runoff feature is OFF

Do you want to CHANGE QUERY-TIME parameters ? (y/n) : y

QUERY TIME Elements

Element0 - '0' for ASK QUESTIONS
           '1' for NO y/n Questions
Element1 - '0' equates to 'n' NO EXPansion
           '1' equates to 'y' YES EXPansion
Element2 - '0' NO BODY COUNT so ASK
           '#' Take this BodyCount
Input Values in 1 and 0 only

SELECT ... Should Expand Always

SEL[2]={1,1} .... CHANGE (y/n) : y
Input SEL[0] ...0-ASK Questions, 1-NO ASK : 1
Input SEL[1] ...0-NO EXP, 1-YES EXP : 1
SELprob = {0.80,0.20} .... Is {TRUE,FALSE} Probability ok ? (y/n) : y

LoopA ... Avoid Expanding

LpA[3]={1,0,0} .... CHANGE (y/n) : y
Input LpA[0] ...0-ASK Questions, 1-NO ASK : 1
Input LpA[1] ...0-NO EXP, 1-YES EXP : 0
Input LpA[2] ...0-ASK BdyCnt, #-BdyCnt : 5

LoopB ... Avoid Expanding

LpB[3]={1,0,0} .... CHANGE (y/n) : y
Input LpB[0] ...0-ASK Questions, 1-NO ASK : 1
Input LpB[1] ...0-NO EXP, 1-YES EXP : 0
Input LpB[2] ...0-ASK BdyCnt, #-BdyCnt : 5

FOR-ALL ... Should be expanded always

FA[3]={1,1,0} .... CHANGE (y/n) : y
Input FA[2] ...0-ASK BdyCnt, #-BdyCnt : 5

Press another RETURN :

```

figure 4.7 User Interface of the Automator tool

externally, that is, after the first pass of the translation and before the target for BLAS is output. Figure 4.8 shows how it is effected.

3. The target for input to BLAS is checked for any indegree mismatch or any cycles in the graph. These checks are performed before and after Compaction to validate the compacting of nodes. An example print out of the compaction when translating a program : simple16.opt is shown in figure 4.8.
4. **Query-Time** : The Query-Time feature is described in the code for the Automator. It is shown in figure 4.7 and forms a major part of the User-Interface. It is organized into small arrays of 2 elements for *Select* and 3 elements for Loops and *ForAll*. This option allows a user to specify almost any combination to avoid interaction with the translation. The 'ASK' statements imply *ask when translating*. The default values are given in curly brackets. Recommendations are included with each name of compound structure, namely 'Avoid Expanding LoopA' and the like. Further, different probabilities can be assigned to the True and False nodes of a *Select* Structure for various runs based on actual program profiles or experimental needs.

```

As many as 8689 nodes maybe compacted...Thats a Good 12.18% !

SEQUENTIAL EXECUTION TIME BEFORE compacting : 55950479
TOTAL NODES is 71318

    CHECKING FOR NODE CYCLES ..... No Cycles :)
    CHECKING FOR INDEGREE MISMATCH ..... FILE is OK :)

COMPACTED Nodes = 8689 (12.18% of total nodes.)

Once again CHECK after COMPACTING

SEQUENTIAL EXECUTION TIME AFTER compacting : 55950479
TOTAL NODES is 62629

    CHECKING FOR NODE CYCLES ..... No Cycles :)
    CHECKING FOR INDEGREE MISMATCH ..... FILE is OK :)

COMPACTED GRAPH is PRINTING

```

figure 4.8 Trace of compaction for the program simple16.opt

5. **AutoMate** : A complex structure has additional fields associated with it in Main-Array. These fields keep record of the choices for 'expansion' and 'body counts' taken for the nested levels in the first body of that compound node. This enables each body to replicate the behavior of the first body with respect to all its nested levels. Here is a real significance of the name 'Automator'.

4.12 MAPPING

Mapping IF nodes to a message passing architecture is the next important area of research. A correct mapping will validate the results obtained from simulations of BLAS on the Automator outputs. The mapping problem involves the actual implementation and operations of each of the IF1 nodes in that particular architecture.

The architecture assumed here is that of a nCUBE 3E series. It supports broadcast of values to different PEs from a particular PE. This saves on the startup costs when doing any array manipulation. That is, the array does not have to be sent to each forked process on a different PE, one after the other. All PEs can receive it at the same time.

The hardware and the compiler play a vital role in the mapping issues. For instance in a matrix multiplication program, the matrices have to be dispersed over different PEs for the various operations that need be performed. This may be achieved by the compiler doing a *striped partitioning* of the matrix or a *checkerboard partitioning*. In striped partitioning the matrix is divided by the compiler into groups of complete rows or columns and each processor is assigned one such group. Thus, a $n \times n$ matrix can be striped over n processors. Checkerboard partitioning involves the compiler to divide the matrix into smaller square or rectangular blocks [35] or submatrices that are distributed amongst PEs. No processor is assigned a complete row or a column. A $n \times n$ matrix, can be at most, distributed over n^2 processors. This will allow for more concurrency to be exploited, but the topology best suited to it will be a mesh interconnection. The nCUBE hypercube can embed a mesh topology.

Based on the above general concepts, two nodes will be mapped onto this architecture using two techniques : one without broadcast and the other with broadcast.

4.12.1 AScatter

The function of *AScatter* requires it to generate the various elements of an array together with their indices. This function has to be carried out by a processor, in which the complete array may be read from memory and each index with its value is given out in a sequence to multiple instances, and not broadcasted. With every element-index pair, the processor forks off a process containing that pair. If communication delays are permissible, the forked process is executed on another processor, which after completion of computation will send back the results.

The above mapping can be made more efficient if broadcast is made use of. In that, the processor that holds the *AScatter* node will broadcast the array to all other PEs and the compiler will cause each PE to generate its own index depending on the section of the array it will operate on. The index is generated based on a precalculation done by the compiler and it need not be passed by the processor that has broadcasted the array. Thus, a processor forks different instances and broadcasts the array to all PEs with each PE extracting the element based on the index it holds.

4.12.2 Range Generate

This node is a fictitious node. It implies that a processor calculates the range or the number of instances and actually no real data is passed on the edges to the multiple instances. The processor spawns or forks as many multiple instances in a sequence as the range indicates. Each instance, based on the sequence in which it is spawned off, evaluates the expression. This allows the processor to fork many instances and let them execute in parallel. This will cause the first instance to complete first and send its value to the processor that is responsible for packing. As results come in from each completing process, they are packed in order by the processor holding the RET node. BLAS will always have one processor hold the range generator and the collector nodes, by very nature of the graph and algorithm.

The above method introduces some sequentiality. This can be done away with completely. The compiler will cause each of the PEs to hold their respective index. This index calculation is done based on some operation on the **processor id** (Identification) number. The Array is broadcast by the PE that originally held the *RangeGenerate*. Thus, the *RangeGenerate* will be dissolved and as shown in figure 4.9.

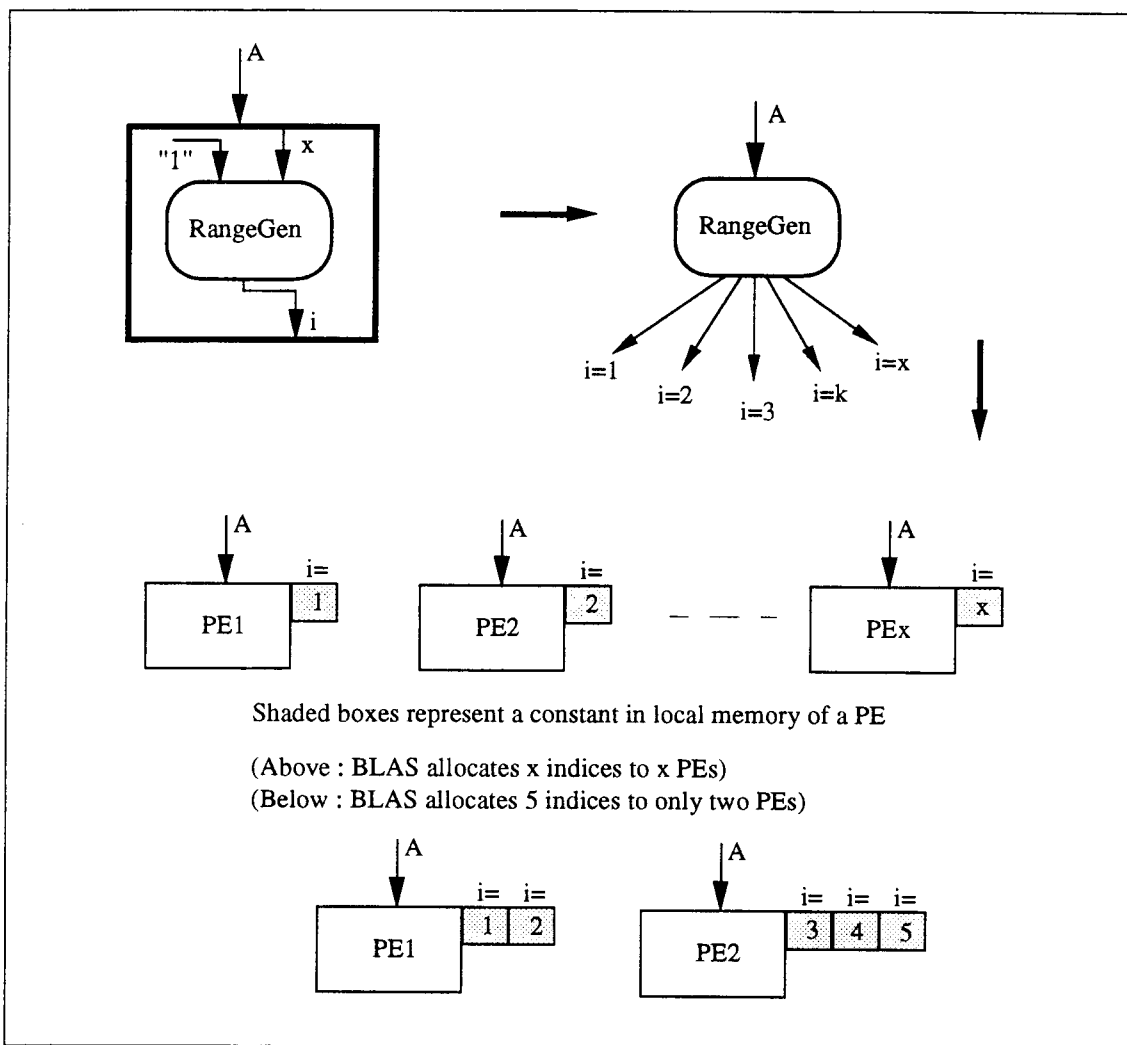


figure 4.9 Mapping of an IF1 Range-Generator Node to a multicomputer.

If the BLAS allocates two of the five indices (for example) to a single PE1 and other three indices to PE2, then the compiler will generate 2 constants for PE1 and 3 for PE2. This will require an array broadcast to only two PEs.

This outline needs to be further elaborated and an actual implementation of the mapping needs to be carried out.

CHAPTER 5. BLAS and SIMULATION RESULTS

5.1 Static Scheduling

Static scheduling means allocating tasks to processors at compile time. This thesis involves static scheduling of independent tasks to various PEs in a hypercube network. Extensive work has already been done in this area and many instances of these problems are proved to be NP-Complete (cannot be solved in polynomial time [25 and 26]). However, with increased research, time consuming optimal solutions are being traded off for more practical heuristic approaches to an optimal solution. The Balanced Layered Allocation Scheme (BLAS) is such a heuristic approach towards an optimal solution. By considering the nature of tasks that occur in real programs one is able to design heuristic approximations that can be used efficiently in practice.

The only scheduling problem that has been solved optimally in polynomial time is the case where the number of processors is 2 and all tasks have a unit execution time [25 and 26]. Such a case is impractical. A family of heuristic algorithms that have been developed for general cases are the 'list scheduling' algorithms. The most popular is the 'critical path' heuristic. The basic concept behind 'list scheduling' is to arrange the tasks of a given graph in a priority list and assign tasks with the highest priority, each time a processor becomes idle [25]. The 'critical path' heuristic finds a critical path of the graph, and gives priority to those tasks that compose the critical path. BLAS is a modified and improved version of the 'critical path' method.

5.2 BLAS

BLAS finds a balance between computation and communication costs when assigning tasks to processors. Originally developed by Lee et al [18], it was called the Vertically Layered Allocation scheme. The VL scheme was subsequently extended by Freytag [12] and called BLAS. The limitations of the VL scheme and details of the proposed BLAS algorithm are outlined in Freytag's thesis. The previous work laid the base for the present research. A brief overview is presented here with an actual calculation for the matrix multiplication example that has been followed in this text.

5.2.1 Objectives of BLAS

The Allocation problem is one of maximizing the inherent concurrence of a program while minimizing the contention for processing resources. The Balance Layered algorithm is based on three general objectives [12] and is a compile time method :

- (1) assign concurrently executable nodes to separate PEs,
- (2) assign data dependent nodes to the same processing element, and
- (3) assign nodes to PEs that lead to an earlier completion time of the program.

Objective (1) maximizes concurrence and (2) minimizes communication costs. Both these objectives seem to conflict, but are compromised by (3), which requires to take into account the effects of inter PE communication costs and execution times, before assigning the nodes to a PE. This trade off provides a heuristic to the algorithm.

5.3 Overview

BLAS can be viewed as composed of two main blocks : 1) The **Partitioning** block that initially takes a data flow graph and separates it out into vertical layers. 2) **Assignment** Block that allocates these layers to processors in a system. The vertical layers are determined by first identifying a *critical path* (CP) for the graph. The remaining layers are found by recursively determining the *Longest Directed Path* (LDP) from the nodes that have already been allocated, using the same approach as that was used in determining the CP. Thus, the program is broken down into modules based on two parameters: execution times and communication delays. The modules are then allocated to PEs only after iteratively considering these parameters on all the processors.

The CP dictates the earliest execution time [12] of a program and thus is given the highest priority in assignment. It is determined in two passes : the *forward pass* determines the earliest time a node can finish and the *backward pass* gives the latest time that node finishes execution. CP time gives a lower bound to the execution of a graph. The recursive determining of LDPs is also done in two passes as above.

A **Horizontal** block has been introduced before the partitioning phase, in this project, to divide the graph into horizontal layers and keep an account of which nodes occupy a given horizontal layer. This introduction has been made in order to correct some major bugs in the implementation code of BLAS. These bugs had never surfaced earlier, as BLAS was not tried with real life program cases till the Automator got into

shape. An overview of the changes will be discussed in a later section and their details are well commented in the program code (BLAS version 5.0).

5.3.1 Examples of Allocation

Two cases for allocation of graphs will be presented. The first one is a small example graph that is to be allocated in a 2 processor system with no communication delays. The second will be the allocation of `matmult2if1.i` (from the discussion of the Automator in Chapter 4). The matrix multiplication program is allocated on a 4 processor or 2-dimension Hypercube system.

Example 1 : Refer to figure 5.1 and the following trace file in figure 5.2.

The CP is determined first and dictates a lower bound time on the execution of a graph. For one processor case, the execution time represents the serial execution of the graph. This entails an addition of all the node execution times. BLAS divides the graph into Horizontal layers. The **H** in the adjacency list of a graph **G** having **N** nodes and **A** arcs, is the horizontal layer within which a node lies. See above mentioned figures.

For the 2 processor case, the CP is allocated to layer 1 which corresponds to PE1 and is indicated in the trace as L 1. After this assignment, LDP1 having Root N9 and Exit N12 is allocated to PE0. The question is that 'which PE will N8 be placed ?'. If placed in PE1, the overall execution time of the graph becomes 60 cycles. Node 6 cannot execute before N8 has completed. As there are no communication costs to be considered, assume N8 is placed in PE0.

Then, the time spent in execution may be calculated for purposes of analysis at the end of each Horizontal layer for both PEs. At the start of H4, it is 20 cycles for PE1 and only 15 cycles for PE0, indicating that N11 has started execution 5 cycles earlier on PE0, than N5 has on PE1. Hence, it will complete at time = 25 cycles and N8 being in the same PE, can start execution and will be halfway through, when N5 completes. N6 will have to wait only an extra 5 cycles. Thus, total completion time of the graph is 55 cycles, saving 5 cycles if N8 was with PE0.

The execution cycles of each node is taken arbitrarily for explaining how the allocation scheme works. Also, this is a typical case when the communication delays are not taken into account.

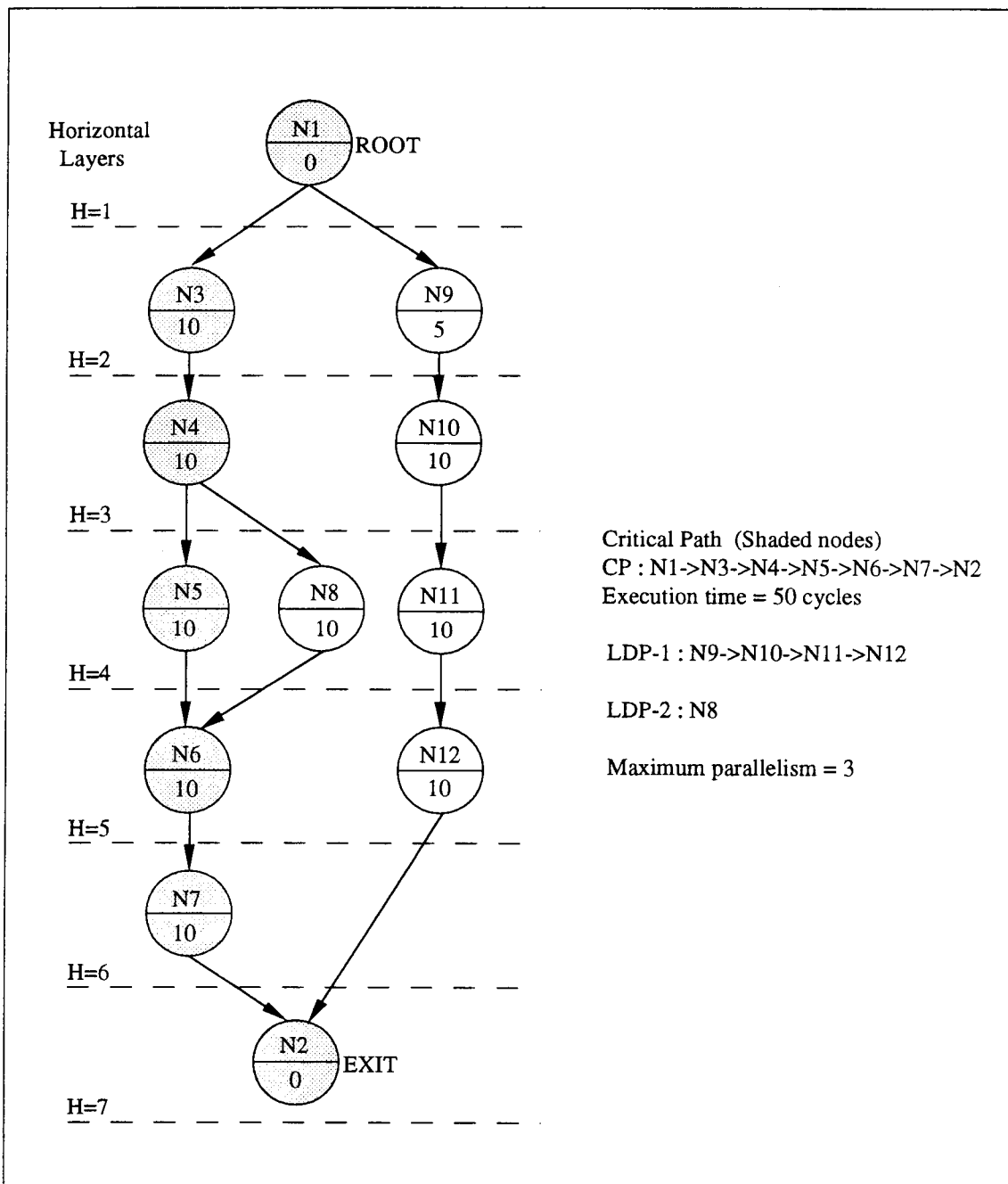


figure 5.1 Example 1 demonstrating BLAS.

```

LDP TRACE FILE : Example1.Tany
ALLOCATION : Any Processor
Maximum parallelism is 3

#####
PARTITIONING PHASE
#####

adjacency list of G(N,A)-----
N1 t=0 H=1 indegree=0 -> 3 9
N2 t=0 H=7 indegree=2 ->
N3 t=10 H=2 indegree=1 -> 4
N4 t=10 H=3 indegree=1 -> 5 8
N5 t=10 H=4 indegree=1 -> 6
N6 t=10 H=5 indegree=2 -> 7
N7 t=10 H=6 indegree=1 -> 2
N8 t=10 H=4 indegree=1 -> 6
N9 t=5 H=2 indegree=1 -> 10
N10 t=10 H=3 indegree=1 -> 11
N11 t=10 H=4 indegree=1 -> 12
N12 t=10 H=5 indegree=1 -> 2

Determining Critical Path
Root = 1, Exit = 2
LDP->1 ->3 ->4 ->5 ->6 ->7 ->2

Starting LDPs (Longest Directed Paths)
Root = 9, Exit = 12
LDP->9 ->10 ->11 ->12

Root = 8, Exit = 8
LDP->8

All LDPs are Done .... LDP COUNT is CP + 2

Lower bound time = 50.000 (Critical Path Time)

#####
NUMBER OF PROCESSORS IS 1
#####
Total execution time with communication is 95.0.
L 0-----
  1 3 4 5 6 7 2
    9 10 11 12
      8
-----

#####
NUMBER OF PROCESSORS IS 2
#####
Total execution time with communication is 55.0.
L 0-----
  9 10 11 12
    8
-----
L 1-----
  1 3 4 5 6 7 2
-----

```

figure 5.2 Trace file for Example 1 obtained when executing BLAS.

The key features to be noted here are that :

- 1) Once a node is assigned, it is marked and never gets processed by any PE during further allocation.
- 2) Each node other than those belonging to CP, are iteratively assigned for examining a best possible completion time of the graph.
- 3) Horizontal layering exposes the ordering of a graph and any other dependencies.

Example 2 : Refer to figure 5.3 for the trace of LDPs and figure 5.4 for an allocation on 2-dimension hypercube. The allocation in figure 5.3 is from a simulation. The execution times in this example are real and from the SISAL code. An example calculation verifying above follows:

The CP is assigned arbitrarily to PE2. The CP nodes are marked and each of the LDPs is assigned iteratively to each PE layer. In this manner, effects of execution times and communications costs can be weighted against different layer assignments for each LDP. An LDP is assigned to a layer, after weighing its effect in each PE layer and according to the objectives of BLAS.

Consider LDP1 : The completion times (in cycles) of each PE layer, when LDP1 is assigned to it can be given in a *layer-set* format as $\{t_0, t_1, t_2, t_3\}$. These *layer-sets* are given below when LDP1 is assigned to -

layer 0 = {159+26+26, 0, **159**, 0}

layer 1 = {0, 159+27+27, **159**, 0}

layer 2 = {0, 0, **159**+95, 0}

layer 3 = {0, 0, **159**, 159+26+26}

Notes :

1. t_2 equals **159** cycles, indicating the Critical Path which was initially assigned.
2. The communication latencies that are produced when dependencies cross the layers are 26 or 27 cycles. That is, in layer 0 : 25+1, for N3 to send data to N7 and 25+1, for N17 to return data to N5, since, layer 0 and 2 are adjacent. For non-adjacent layers, communication is the **Start up time + hop distance . interhop latency**.
3. The maximum of each *layer-set* gives the maximum completion time of a graph when the LDP is assigned to that layer.


```

LDP TRACE FILE : matmult2if1.Tany
ALLOCATION : Any Processor
Maximum parallelism is 8

Determining Critical Path
Root = 1, Exit = 2
LDP->1 ->3 ->6 ->9 ->12 ->11 ->8 ->5 ->4 ->2

Critical Path Time = 159 cycles

Longest Directed Paths
1) Root = 7, Exit = 17
   LDP->7 ->18 ->21 ->20 ->17
2) Root = 10, Exit = 14
   LDP->10 ->15 ->14
3) Root = 13, Exit = 13
   LDP->13
4) Root = 19, Exit = 23
   LDP->19 ->24 ->23
5) Root = 22, Exit = 22
   LDP->22
6) Root = 16, Exit = 16
   LDP->16
7) Root = 25, Exit = 25
   LDP->25

Serial Execution Time = 386.0 cycles

Communication Latencies :
Start Up Time = 25 cycles
Inter PE communication delay = 1 cycle

#####
NUMBER OF PROCESSORS IS 4
#####
Total execution time with communication is 271.0.
L 0-----
    16
L 1-----
    25
L 2-----
  1  3  6  9 12 11  8  5  4  2
    10 15 14
      13
L 3-----
    7 18 21 20 17
    19 24 23
      22
-----

```

figure 5.3 Trace file indicating the allocation of nodes of matmultif1.i

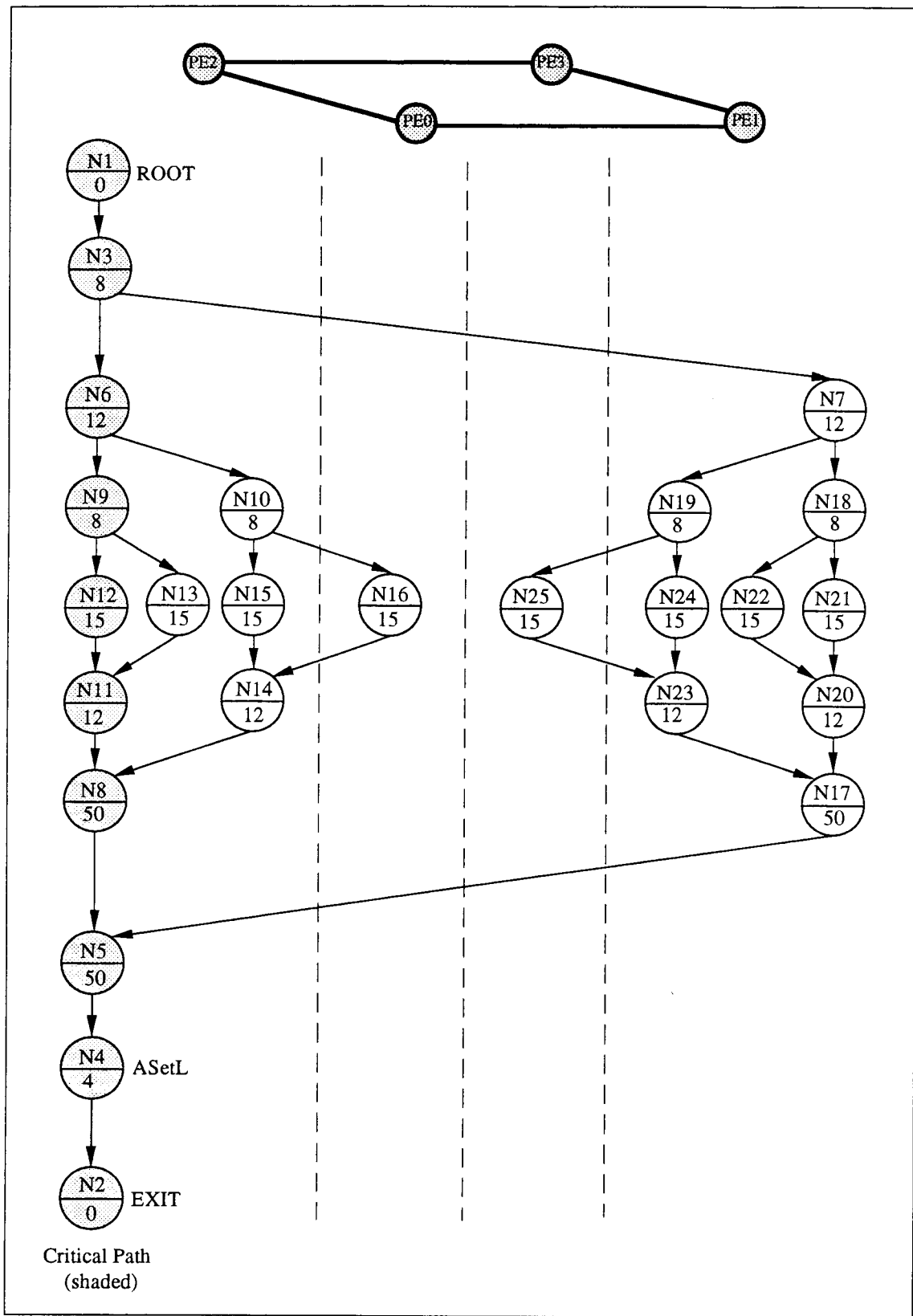


figure 5.4 Allocation of matmult2.if1 to PE layers in a 2-dim Hypercube

4. All the *layer-set* maximums are collected in a *graph-set*. So, for the given *layer-sets*, the *graph-set* is : $\{211, 213, 256, 211\}$
5. Finally, the LDP is assigned to a layer which yields lowest completion time in the *graph-set*. That is, assign LDP to a layer which yields $\min[\textit{graph-set}]$

Similarly,

For LDP2 : $\textit{graph-set} = \{211, 213, 194, 360\}$; LDP2 should be assigned to PE layer2 or PE2 and,

For LDP3 : $\textit{graph-set} = \{211, 213, 209, 298\}$; LDP3 is assigned to PE2

For LDP4 : $\textit{graph-set} = \{213, 211, 422, 184\}$; LDP4 is assigned to PE3

For LDP5 : $\textit{graph-set} = \{213, 211, 343, 209\}$; LDP5 is assigned to PE3

For LDP6 : $\textit{graph-set} = \{211, 213, 224, 410\}$; LDP6 is assigned to PE0

For LDP7 : $\textit{graph-set} = \{280, 211, 420, 214\}$; LDP7 is assigned to PE1

Further explanation for the algorithm is to be referenced in [12]. However, if in the above example calculations, the start up time and inter PE delay was increased by a factor of 10 as is the case with the nCUBE 3E systems then, the allocation of all LDPs would be on a single processor PE2 where the CP was arbitrarily assigned.

5.4 Order of the BLAS algorithm

The critical path can be determined with a complexity of $O(N^2)$ [12]. The complexity of determining the LDPs decreases with each iteration, but the worst case will occur for one node at a time. This gives an order of $O(N^3)$ for determining the CP and LDPs. As this is done p times for each LDP, where p is the number of PEs or PE layers, the complexity order worsens to $O(pN^3)$ for $p \leq N$. This poses a time and cpu-memory limitation on graph sizes. That is, it takes more than 30 days to run a 70,000 nodes graph with 64 processors on a SUN Sparc10 system. Graph sizes have been limited to a maximum of 10,000 nodes. In addition, the use of optimized files from SISAL's compilation, instead of IF1 files helped towards this end. The IF1OPT phase is discussed in chapter 3.

5.5 Summary of Changes that were made to the BLAS source code

1. Code for BLAS was originally written in ThinkC for the MACs. This has been ported to UNIX on the SUNs.
2. A choice for running simulations over variable number of PEs in a Hypercube topology is given. A user may choose the dimension of the Hypercube topology and the simulation will run for the complete range of PEs for that dimension. Also, a feature for a *single run* of PEs has been added. This saves time for incremental simulations. Previous Macintosh versions required to go through an entire range of PEs everytime a simulation was started.

3. The user interface has been completely overhauled. There are 3 output files that BLAS produces with each run. The Trace file (*file.T*) is the actual trace of the simulation and is useful in debugging. The Speed Up file (*file.D*) gives the information for speed-ups, efficiency and other parameters of the run. The Graph file (*file.E*) gives an ASCII output of speed up, efficiency and speed-up ratios for each PE, to enable automatic graphing in Microsoft's Excel package. All these files may have suffixes called **any** (*file.Tany*) indicating it is for **Any Processor Allocation**.

There are two schemes available in BLAS: one forces the allocation of each LDP to **adjacent** processors only, and the other is the regular 'any' PE allocation. An example of *.D* file is given in figure 5.5 for a LU decomposition with pivot (*lu5piv.Dany*).

The 16 processor case was simulated in a 'single run' mode and is included here to show the saved time.

4. The Partitioning phase was altered as it had some hidden bugs. These bugs did not allow for correct partitioning of all graphs. An example of the flaw and its correction is presented below. Consider figure 5.6. Assume that the Critical Path has been determined. Nodes N1, N3, N4 and N2 are marked.

Flaw : All nodes emanating from N1 are checked for building the subsequent LDPs. Let N5 be the first choice. BLAS maintains a stack of nodes to be processed. Originally, the Root node (N5 in this case) is pushed on the stack. Then, the node's output arcs are processed, and a copy of the indegree count (variable:*keep_count*) for each successor node is decremented. If the count reaches 0, that node is pushed on the stack, to be processed later. A node is not processed until its indegree count reaches 0 to insure that all of its predecessor nodes are processed first and that no node is processed more than once.

```

SPEED-UP AND DATA FILE : lu5piv.Dany
Sun Nov 28 22:26:51 1993
Manoj Raisinghani

ALLOCATION : Any Processor

EXECUTION :
  Sequential Execution = 11748
  Total Nodes = 228
  Average Execution Base (Seq.Exec/Tot.Nodes) : 51.526

COMMUNICATION Costs for a 50 Mhz processor for nCUBE 3:
  Max. StartUpTime = 250 cycles with HyperCube topology
  Max. InterProcessor Communication is 10 cycles

DEPTH in horizontal layers = 43
MAXimum PARallelism = 55

  Computation STARTs at Sun Nov 28 22:26:55 1993

COMMUNICATION to EXECUTION TIME RATIO (C/T) is 10/51.526
  Comm Latency = StartUpTime + (No. of Hops * 10)

Critical Path Time (Lower Bound Time) = 2846.000
Average Parallelism (Ideal SpeedUp) is :
  SequentialTime/CriticalPathTime = 4.128

  1 PROCESSOR(s)
(a) Total execution time with communication is 11748.0.
(b) SpeedUp = 1.000
(c) Efficiency = 1.000 or 100.0 percent
(d) Performance Ratio = 0.242 or 24.2 percent of Ideal SpdUp

  2 PROCESSOR(s)
(a) Total execution time with communication is 7022.0.
(b) SpeedUp = 1.673
(c) Efficiency = 0.837 or 83.7 percent
(d) Performance Ratio = 0.405 or 40.5 percent of Ideal SpdUp

  4 PROCESSOR(s)
(a) Total execution time with communication is 5096.0.
(b) SpeedUp = 2.305
(c) Efficiency = 0.576 or 57.6 percent
(d) Performance Ratio = 0.558 or 55.8 percent of Ideal SpdUp

  8 PROCESSOR(s)
(a) Total execution time with communication is 3934.0.
(b) SpeedUp = 2.986
(c) Efficiency = 0.373 or 37.3 percent
(d) Performance Ratio = 0.723 or 72.3 percent of Ideal SpdUp

  Computation ENDs at Sun Nov 28 22:27:23 1993

  Computation Starts at Sun Nov 28 22:45:23 1993

  16 PROCESSOR(s)
(a) Total execution time with communication is 3914.0.
(b) SpeedUp = 3.002
(c) Efficiency = 0.188 or 18.8 percent
(d) Performance Ratio = 0.727 or 72.7 percent of Ideal SpdUp

  Computation ENDs at Sun Nov 28 22:45:55 1993

```

figure 5.5 An example speed-up file when simulating a 5x5 LU problem.

Once the root node (N5) is finished, the next node on the stack is processed in the same manner. This continues until the stack is empty.

The problem with this scheme is that a node is only processed if all of its predecessors are processed. If any of its predecessors cannot be reached from the root node, or are marked, the node won't be processed.

For example, node N6 has three incoming arcs, one from each of nodes N5, N3, and N7, giving it an indegree of 3. Node N5 is processed first, and its output arc to N6 is evaluated, reducing node N6's indegree count to 2. However, N3 and N7 are never processed in determining this LDP; node N3 is already marked and N7 can't be reached from N5. Therefore, node 6's other incoming arcs aren't processed, its indegree remains 2. It is not pushed on the stack, and therefore, isn't considered for further processing in this LDP. The final result is that N5 becomes its own LDP, while N5 and N6 should have made up one LDP.

Thus, this 'bug' was missed in the MAC version and had to be cleared to run with the present programs.

Correction : Use of Horizontal layering of the graph has been made. The graph is divided into Horizontal Layers (as was done in the Macintosh versions) and now, arrays consisting of nodes present in each layer is maintained. That is, an array is dynamically created for each Horizontal layer and it contains the node numbers within that layer. The graph is then traversed layer by layer. The CP is determined as before and its nodes are marked. Consider N5 as the new Root for forming an LDP. In this correction process, the indegree-count for each node is not used and the node is never stacked either (when indegree-count equal to zero) for latter processing. Infact, a new array (of type : Boolean) called **on_path[]** is introduced which indicates whether a node is on the path that is being considered currently, or not.

N5 is in layer H2 and all its successors will lie in layers H3 and above (in ascending order). The Root-N5 is marked as on-path. Layer 3 is investigated for nodes that obey 2 conditions : **1)** The nodes in the layer should be successors to a node on the path and **2)** The successor nodes should not have been marked already. N6 and N8 obey above conditions and N4 is a successor but, is a marked node. Hence, N6 and N8 are now marked as on-path and the output arcs of these nodes are considered. As can be seen from the graph, the output arcs of N6 and N8 point to successor nodes that are already marked.

This implies that N6 or N8 may be the Exit node for the LDP with Root N5. The condition for ascertaining the Exit node is **3)** It should have no successors at all, or its successors are only the marked nodes and it should have the highest accumulated

time during this pass. N6 obeys the above condition and is considered the Exit node for that LDP as it has the highest accumulated time as compared to N8. This makes an LDP: N5->N6. Similarly, the LDP: N7->N8 is determined as above. All nodes in a layer are considered first before processing any subsequent layers.

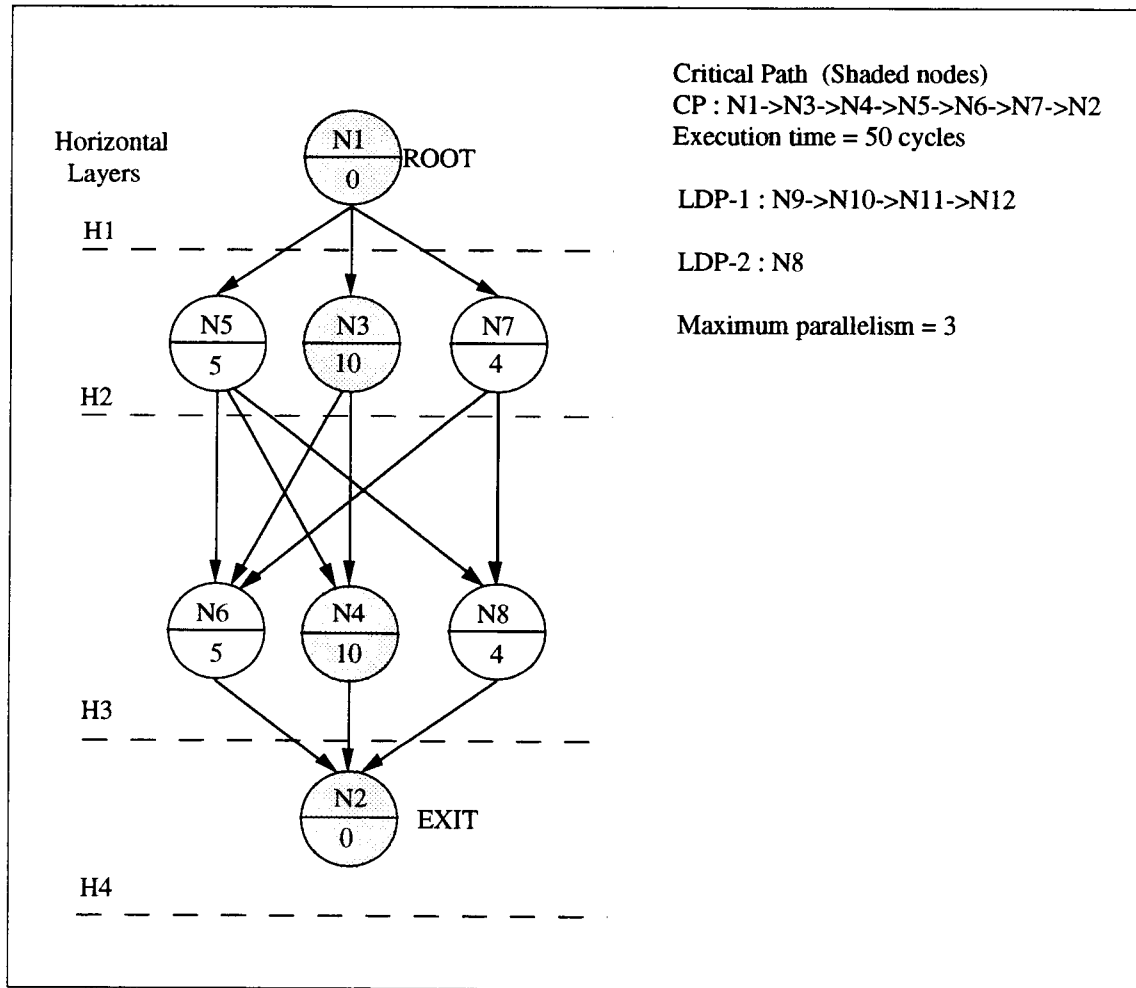


figure 5.6 Example to demonstrate the Flaw and Correction made in BLAS.

5.6 Modified BLAS

The time complexity of BLAS assignment drastically increases for large graphs over large number of processors. Freytag [12] proposed a variation in the allocation algorithm. The time to allocate a program to processors can be reduced efficiently by

considering the allocation of each set of nodes or LDPs to a layer that is at most one processor away from a parent thread. This change was based on the complexity analysis and the algorithm was called Modified BLAS. The time complexity for Modified BLAS is reduced and is of the Order

$$\log_2(2p)O(N^2) \quad \text{for } p \leq N$$

As expected, the total execution time for an assignment using Modified BLAS decreases as the number of PEs increases. This is a relative merit vis-a-vis BLAS (Any PE) assignment.

5.6.1 Effectiveness of Modified BLAS

Freytag [12] in his work, cited simulation results on hand drawn sample graphs which indicated that :

- 1) Modified BLAS does not perform as well as the BLAS for small Communication to Execution ratios (C/T) but,
- 2) Modified BLAS performs as well as BLAS for larger C/T ratios.

In this thesis, it is apparent from the results taken for matrix multiplication and LU decomposition, that the original BLAS allocation outweighs Modified BLAS for increasing sizes of the graphs or increase in the number of LDPs. Refer to graphs in the results section. An option is provided in the user interface for BLAS to either run the simulation for 'Adjacent Processor' or 'Any Processor' allocation. The former will give output files with .T, .D or .E extensions and the latter will attach a suffix 'any' as described earlier.

However, Modified BLAS has to be used for very large programs having more than 4000 nodes. As, it is the only option that will complete the simulation in practical time.

BLAS code needs to be optimized so as to run faster and also remove any redundancy and unrequired computation in the assignment phase of the code.

5.7 Simulations and Results

The previous chapters have described the process that allows one to actually simulate a program graph's allocation on multiple PEs. When parallelizing a graph, performance parameters and concepts of routing with associated delays will be best evident with simulations of that graph on a target architecture. The target system is the

nCUBE 3E series having a hypercube topology with a PE-Start up time of 250 cycles at 50 MHz operation. The interhop latencies between two adjacent PEs is 10 cycles. Based on these parameters, the simulation results for three programs, each with different sizes, are presented. The number of PEs for each simulation has been varied depending on the time of run and available parallelism.

5.8 Programs for the Simulation

The three programs chosen for presentation in this thesis are matrix multiplication, LU decomposition and the Simple benchmark. Material, for matrix multiplication and LU decomposition is available in [34]. Information on Simple can be obtained from LLNL and [9]. There were a large number of programs that were obtained from the SISAL group at LLNL and simulated. However, the above three will be described shortly before going into the discussion of the results.

5.8.1 Matrix Multiplication

The algorithm is widely used in various areas of research. It consists of multiplying two square matrices $A[i, k]$ and $B[k, j]$ to give a third matrix $C[i, j]$. It is represented as

$$C[i, j] = A[i, k] * B[k, j]$$

If A and B are 3×3 square matrices having elements $a_{11}, a_{12}, \dots, a_{33}, b_{11}, b_{12}, \dots, b_{33}$, then, element $c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$ and so on. This says that, each row of A is iteratively multiplied by all columns of B to produce the corresponding row of C . It implies that a large amount of parallelism can be exploited from the algorithm. This is because each operation is independent of any other computation and all computations can be undertaken simultaneously if the matrices can be accessed at the same time.

The sequential order of the algorithm is n^3 (as there are three nested loops) where $n \times n$ is the size of the two matrices. The *matmult.sis* program presented in the discussion of IF1 (in chapter 3), indicate its sequential order of n^3 by the 3 nested loops. For larger size matrices, the corresponding *matmult.if1* files, when passed through the Automator, give large number of nodes. This can make the allocation time very high. To avoid large number of nodes, the optimized version is chosen. A 5×5 matrix multiplication is transformed by the Automator to 140 nodes for the *.if1* file and only 39 nodes for *.opt*

file; a remarkable savings in time! The trick is the reduced number of nested levels in the *.opt* file; in fact only one level of *ForAll* is present.

The SISAL compilation in its optimization phase (IF1OPT), expands the graph. It does so by opening the innermost *k ForAll* and the middle level *j ForAll* structures, to extract the elements of the $B[k, j]$ matrix before presenting it to the outer most *ForAll* with *i* iterations. The optimized file **mat 3x3.opt** is transformed by the Automator to give **mat 3x3.i** file which is shown in figure 5.7.

At this point it is interesting to see how the algorithm behaves with respect to the graph. The graph is essentially divided into two : top half called the Extraction Phase (in figure 5.7) with more nodes than the bottom half - Computation Phase. An analysis follows - The top half consists of nodes from the *Root* N1 to before the *ForAll* at N15. These in-between nodes are *AElement* structures. An *AElement* node [31] extracts the element of an input array at the index position given on its other input port. The first set of nodes from the *Root* (N3, N5 and N7) extract a row each for $k = 1$ to 3, from the *B* matrix. Each row is then presented to another set of *AElements* to extract each of the three elements from that row for $j = 1$ to 3. All rows are treated similarly. The extracted elements from the *B* matrix, are presented to a *ForAll* structure in the lower half.

Each body of the *ForAll* receives a single column of *B* and comprises of simple nodes to carry out the computation such that the first row for the *C* matrix is produced at its output. All 3 elements of a particular row depending on the value of *i*, are computed and assembled. These rows are then gathered by the *Return* node containing the *AGather* structure, that assembles the entire matrix and exports it to the *Exit*.

The execution time of the *ForAll* generator node is $4 \times 3 = 12$ cycles, since, it will spawn off 3 bodies and the individual execution time of the *RangeGenerate* node in this case is 4 cycles. The same is true for the *AGather* whose execution is 75 cycles, as it collects from 3 nodes. Each *Body* execution time is a cumulative addition of all simple nodes that are contained in it. An example of computing the first element of *C* (*c11*) is shown in figure 5.8. The execution times of each of the simple nodes are small enough.

If they were not contained in one node in the graph, they would still be grouped under one PE by BLAS because of their interdependencies. This is a good example of how Node Packing is effected. To get different sizes for the matrix multiplication case, variables *x*, *y* and *z* in the program described in IF1 Language need to be replaced by the value of *n*.

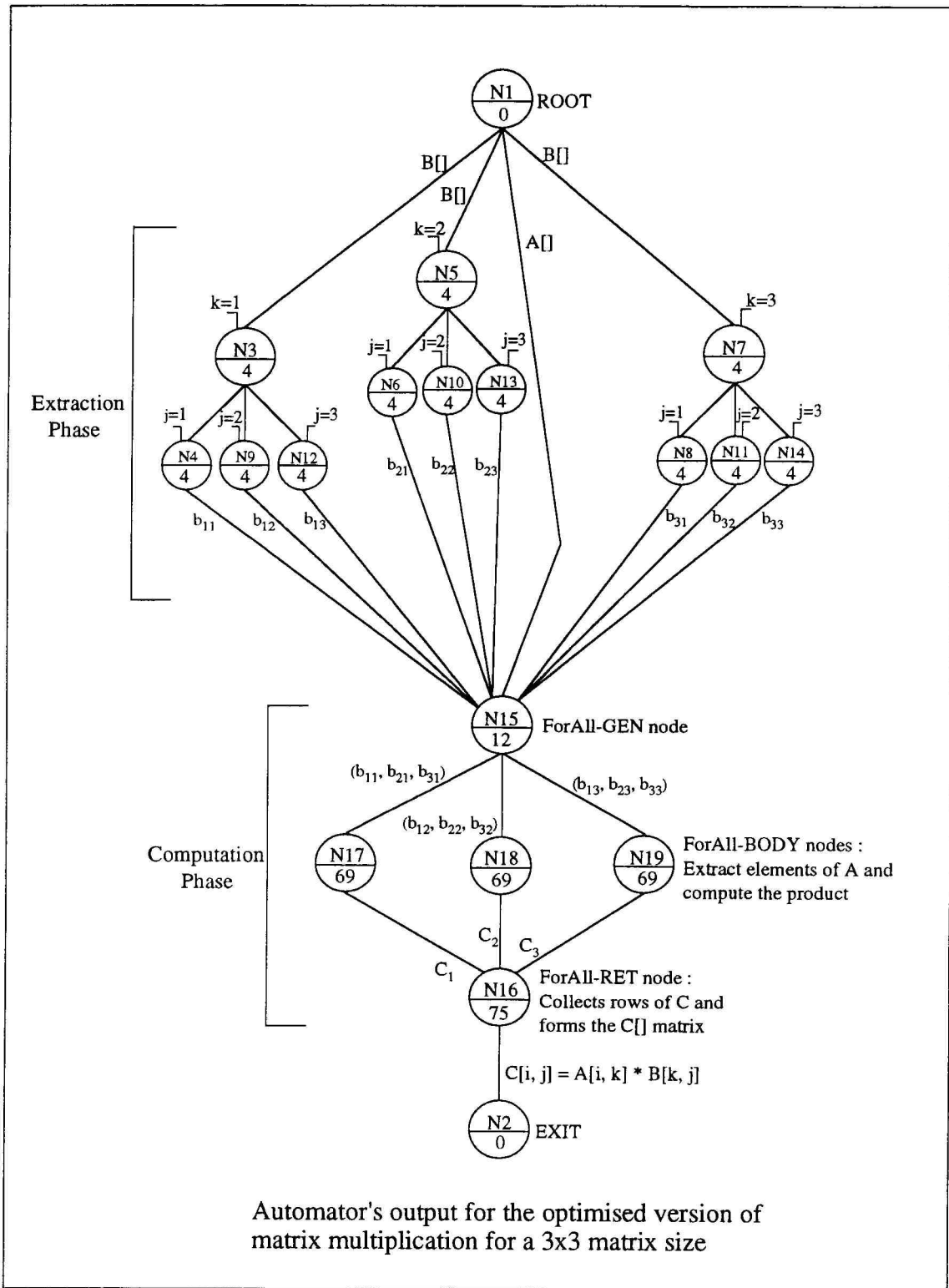
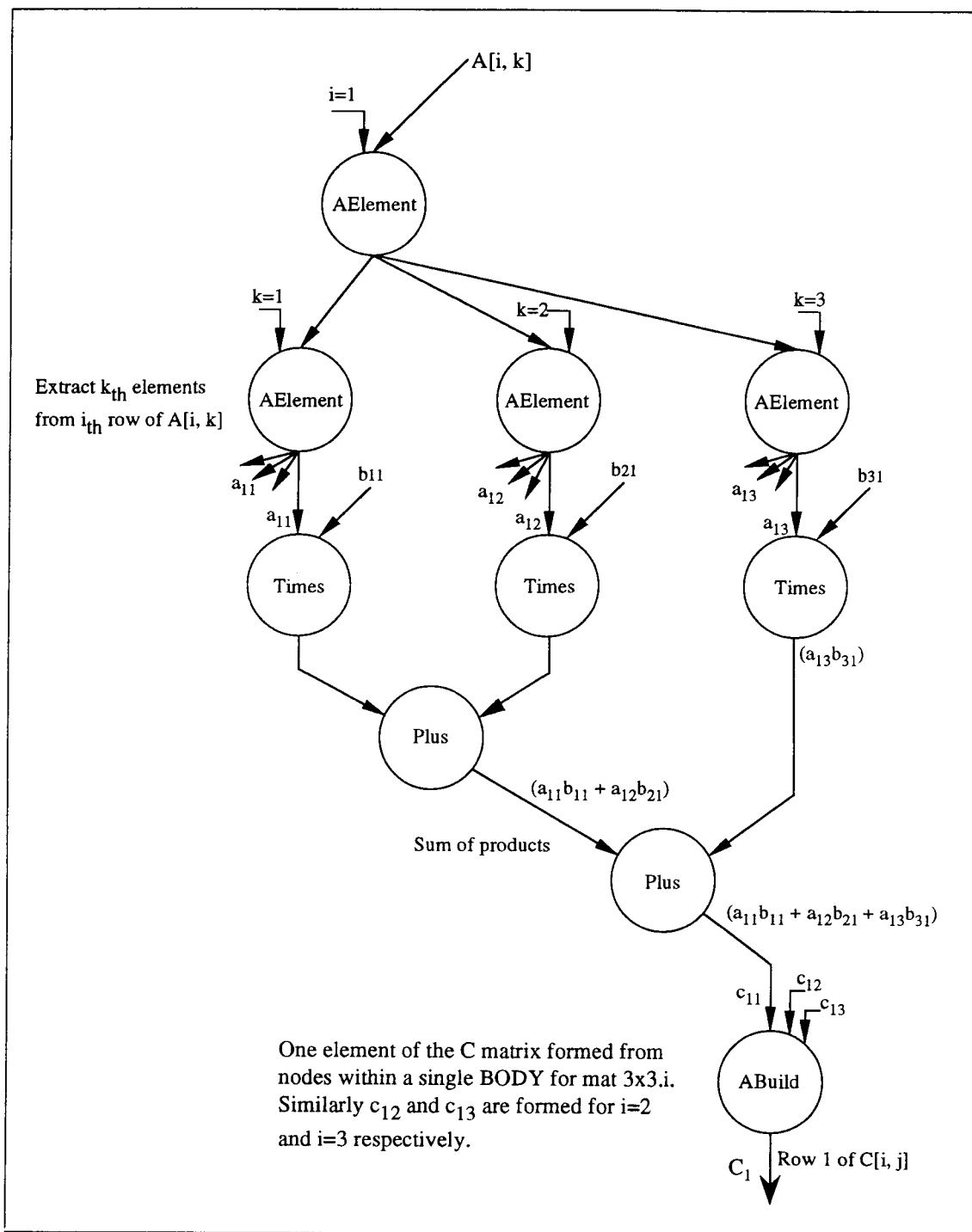


figure 5.7 mat 3x3.i :

figure 5.8 A single Body of **mat 3x3.i**

5.8.2 LU Decomposition

This method is based on the fact that a square matrix **A** can be factorized into the form **LU** where **L** is the unit lower triangular matrix and **U** is the upper triangular matrix, if all the principal minors of **A** are non-singular. A linear system of equations consisting of three variables is given as :

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \quad \text{and can be written as } \mathbf{AX}=\mathbf{B} \quad (1)$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

A is non-singular implies a_{11} not 0 and *minors* of **A** not 0 [34]. It is a standard result of algebra that, when such a result exists, it is unique. Also, **B** is column vector.

$$\text{Let } \mathbf{A} = \mathbf{LU} \quad (2) \quad \text{where:}$$

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

$$\text{Hence (1) becomes } \mathbf{LUX} = \mathbf{B} \quad (3)$$

$$\text{Setting } \mathbf{Y} = \mathbf{UX} \quad (4)$$

$$\mathbf{LY} = \mathbf{B} \quad (5)$$

Which is now equivalent to finding Y's from :

$$y_1 = b_1$$

$$l_{21}y_1 + y_2 = b_2$$

$$l_{31}y_1 + l_{32}y_2 + y_3 = b_3$$

This can be solved using forward substitution. When **Y** is known, the system (4) becomes

$$y_1 = u_{11}x_1 + u_{12}x_2 + u_{13}x_3$$

$$y_2 = u_{22}x_2 + u_{23}x_3$$

$$y_3 = u_{33}x_3$$

The above can be solved for x_1 , x_2 and x_3 by backward substitution which involves pivotal condensation.

For the simulations, **A** is a **nxn** array and **B** is a **n** element vector. To get different sizes, the value for **n** is set to $n = 5$, $n = 10$ and $n = 20$ and the corresponding output from the Automator is simulated. If values are known (may be random) for **A** and **B** such that

the **A** matrix is non-singular, then these values may be fitted into the SISAL program - **lu.piv.sis**, subsequently compiled and passed through the Automator preparing it for BLAS.

5.8.3 Simple

The Simple benchmark was developed at LLNL to represent large scale scientific code which is executed on supercomputers today. It is a hydrodynamics [9 and 20] and heat conduction application which simulates the behavior of a fluid in a sphere, using Lagrangian formulation and equations.

It primarily consists of three main procedures : velocity-position, hydrodynamics and conduction. All other procedures [9 and 20] are run only once or are called by one of the above. The description is out of scope for this work. Details of the algorithm are available with the SISAL group. Simple benchmark too, is matrix based and its '.opt' file from SISAL compilation results in very large number of nodes even for small sized programs.

This benchmark can be rescaled in the **simple.sis** stage itself. At the very beginning of the function **main** in SISAL code, the following five names are defined,

t_{max} - maximum time. Increasing or decreasing this value will increase or decrease the number of iterations

k_{mn}, k_{mx} - the lower and upper bounds of the number of rows in the matrix.

l_{mn}, l_{mx} - the lower and upper bounds of the number of columns in the matrix.

If an array size of 10x10 is required then set: *k_{mn}* = 2; *k_{mx}* = 9 and *l_{mn}* = 2; *l_{mx}* = 9.

Also, the default for *t_{max}* is 4.999 which indicates 60 iterations. The loop body counts for the Automator are simply the value of the matrix size.

5.9 Simulation Results

For the matrix multiplication program, the Any Processor allocation was simulated over a maximum of 1024 processors. The matrix sizes were varied as explained above. The graph **mat nxn** indicates a curve for the size **n**x**n**.

The LU decomposition program was tried over 3 sizes and the curves are called **lu nxn**. The above algorithm can also be evaluated using **no** pivotal condensation during

back substitution. This case has been graphed and curves are called *luN nxn*. Both **lu** and **luN** are simulated for the 'Any Processor' allocation scheme in BLAS. In general, the LU decomposition graphs have fared real well in the simulations.

Finally, the Simple program was tried over the 'Adjacent Processor' allocation scheme. It gave very high number of nodes for small size graphs. Hence the 'adjacent' option saved immense time in simulations. Even then, a 16x16 size problem transforms to 62,629 nodes which takes a minimum of 3 weeks on a Sparc10 Sun system.

For all above cases, Speed-up, Efficiency and Performance Ratio graphs are presented. Performance Ratio is a figure of merit for the algorithms behavior with respect to BLAS. As can be seen the LU algorithm fares the best. This parameter indicates how fast the program graph reaches its ideal speed-up value. However such a merit has significance only if the number of PEs over which the run is being taken is greater than the Maximum parallelism of the graph. Refer to Table 6a for the various parameters obtained during the simulation run.

As the number of nodes increases, so does the LDP count. As a result the maximum parallelism also increases. For regular graphs like matrix multiplication and LU with No pivot, the LDP count is almost similar to the maximum parallelism that is present. This is not the case with graphs like LU with *pivot* and Simple, as these have all the complex structures and various combinations of serial and concurrent sections within each.

In the Table 5a, Horizontal Layers for any given program is almost same for varying sizes. This depicts the goal of the Automator which is to achieve vertical connectivity with horizontal expansion.

The graphs for all algorithms show a steady increase in the Performance ratio which implies better speed-ups for larger sized problems. The cost of a system is the processor-time product as described earlier, and is inverse of efficiency. As the number of PEs is increased for the same problem size, the cost also rises with a decline in efficiency. But, for the same algorithm, efficiency increases for the same number of PEs, over larger problem sizes.

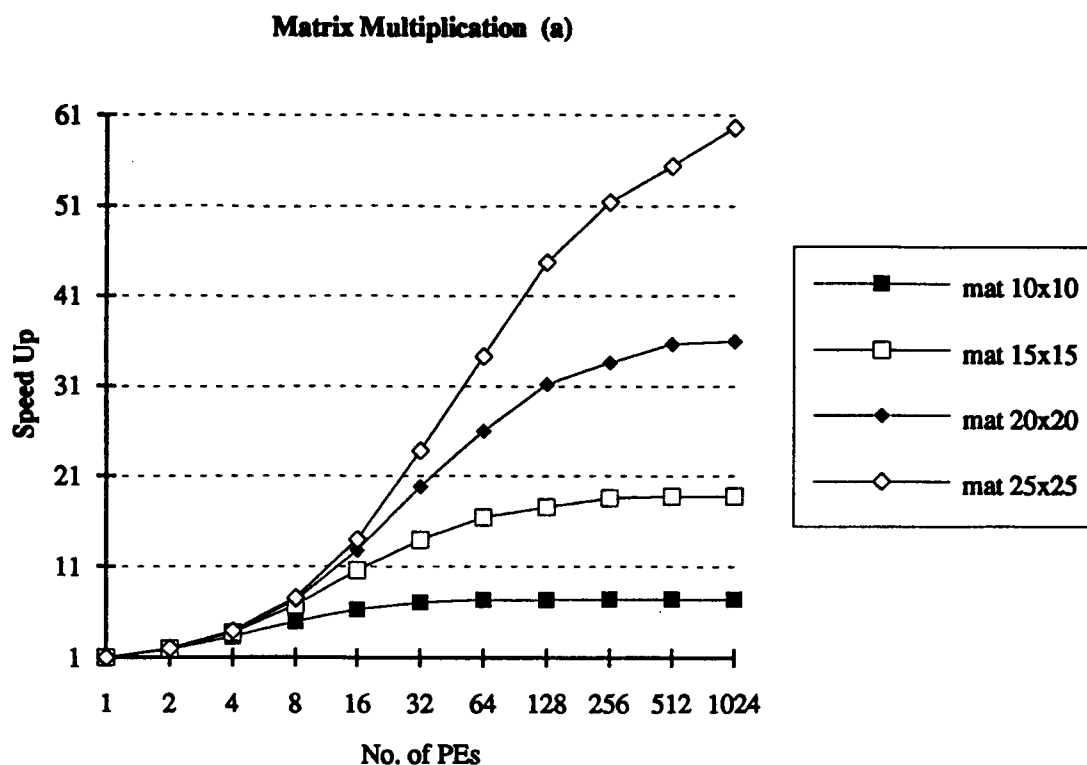


figure 5.9 Matrix Multiplication -- Speed-up curves

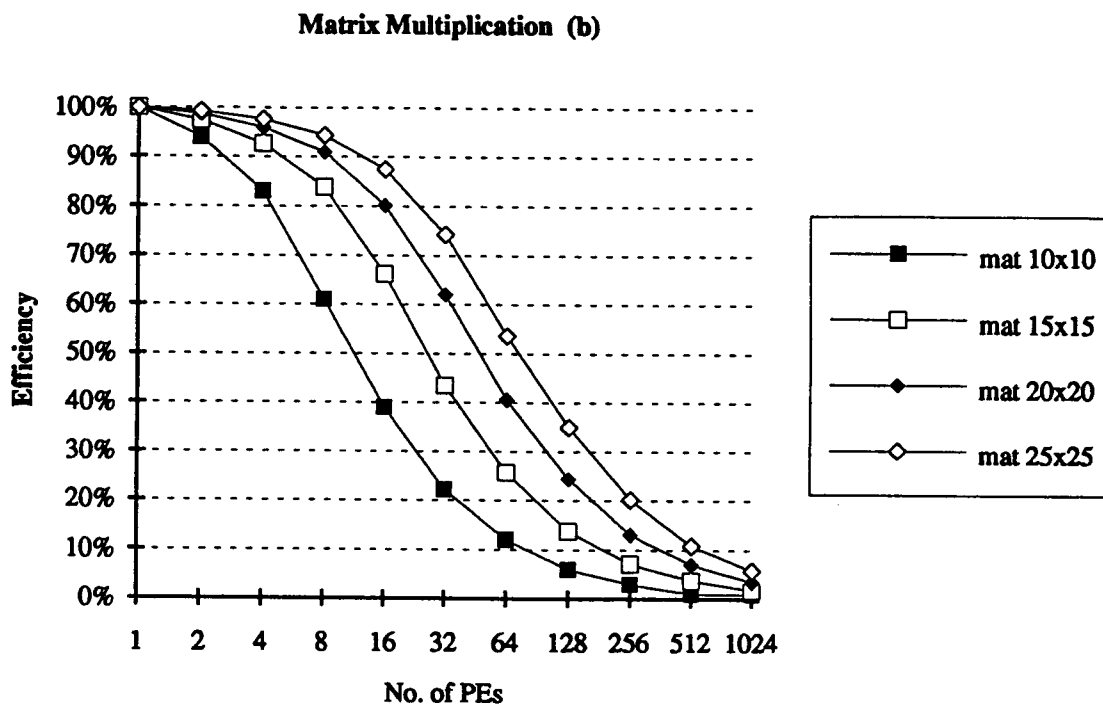


figure 5.10 Matrix Multiplication -- Corresponding Efficiency curves

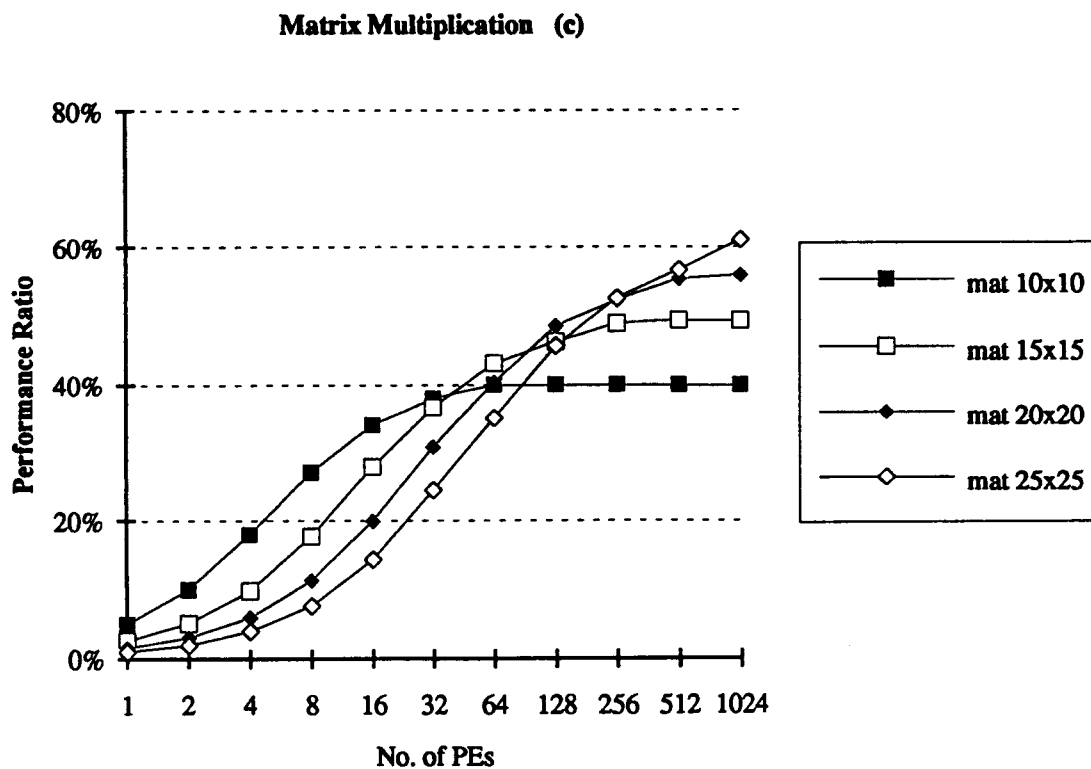


figure 5.11 Matrix Multiplication -- Performance Ratio curves

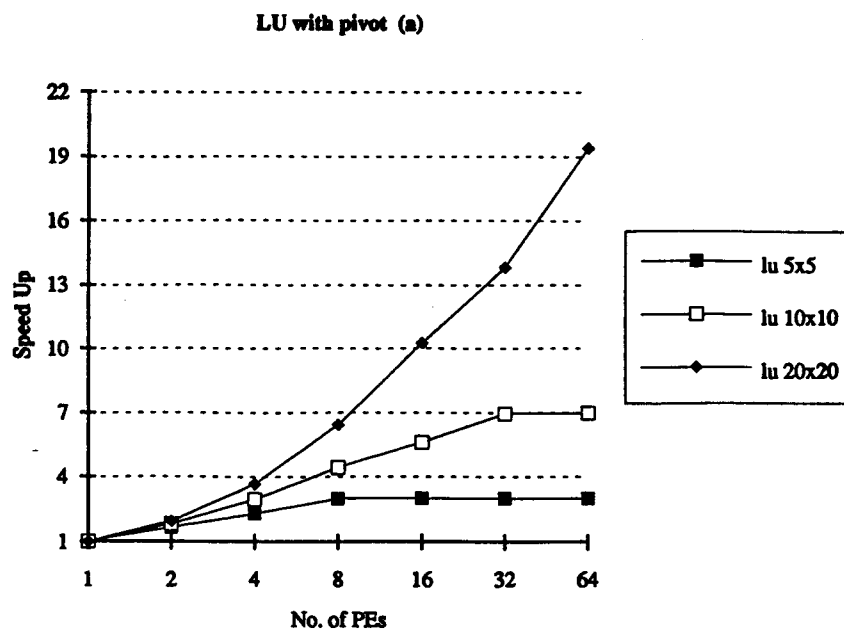


figure 5.12 LU with Pivot -- Speed-up curves

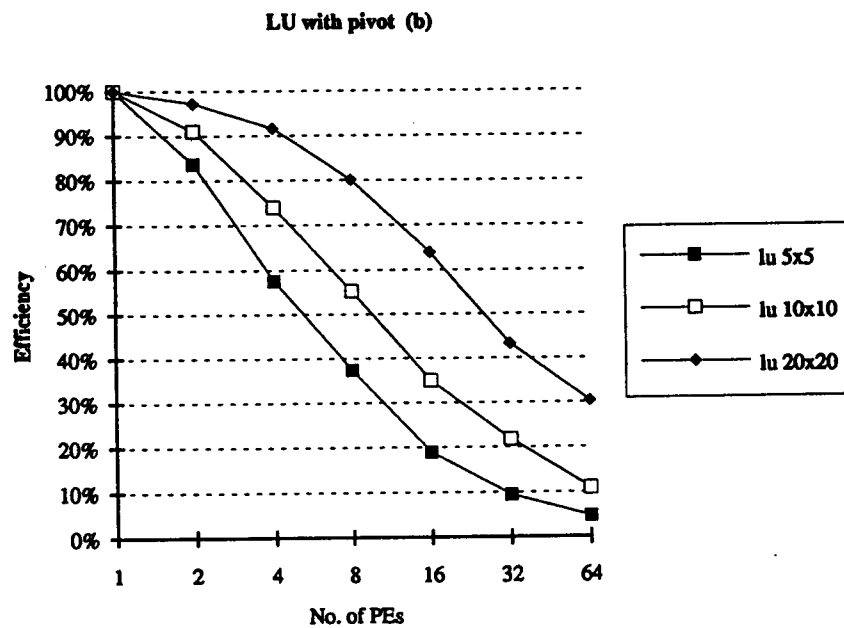


figure 5.13 LU with Pivot -- Corresponding Efficiency curves

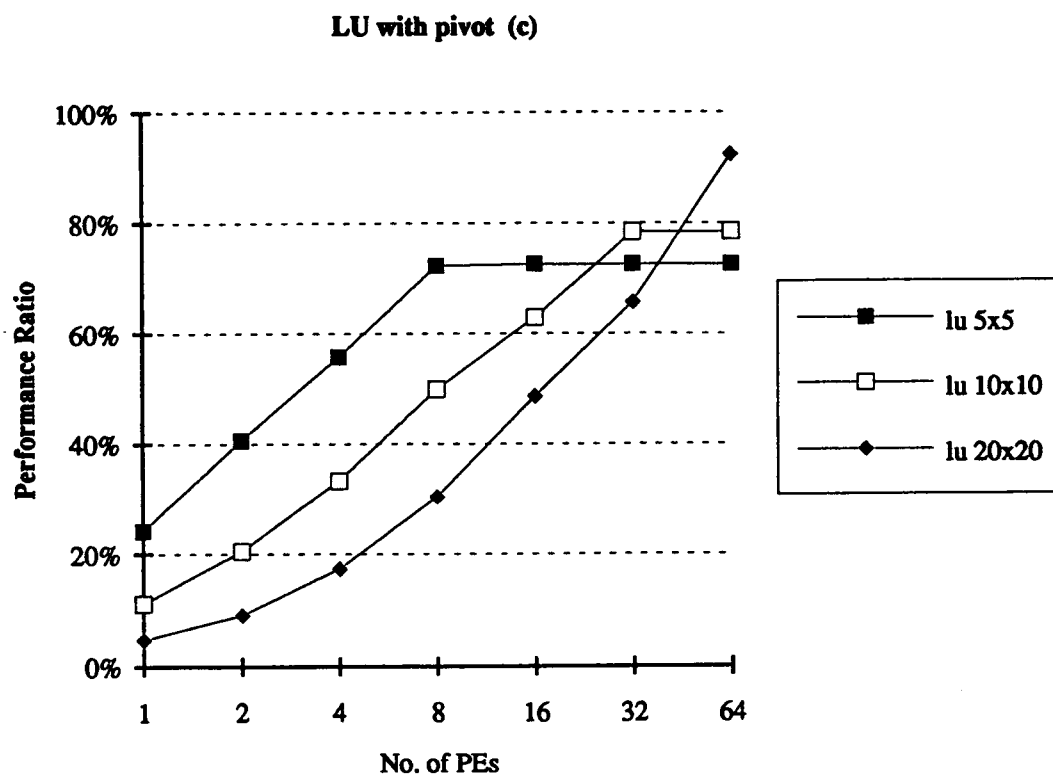


figure 5.14 LU with Pivot -- Performance Ratio curves

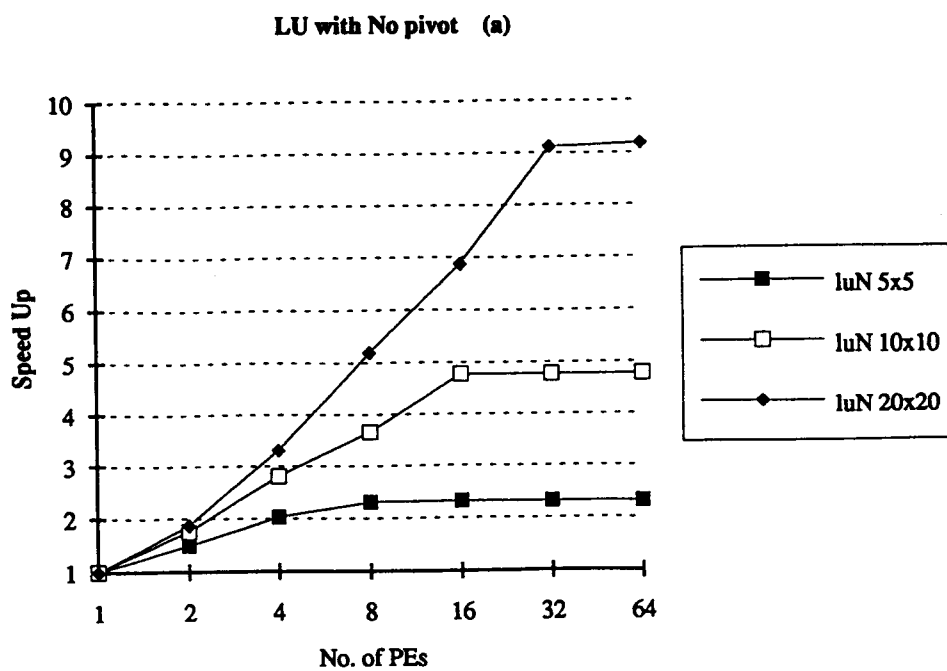


figure 5.15 LU with NO Pivot -- Speed-up curves

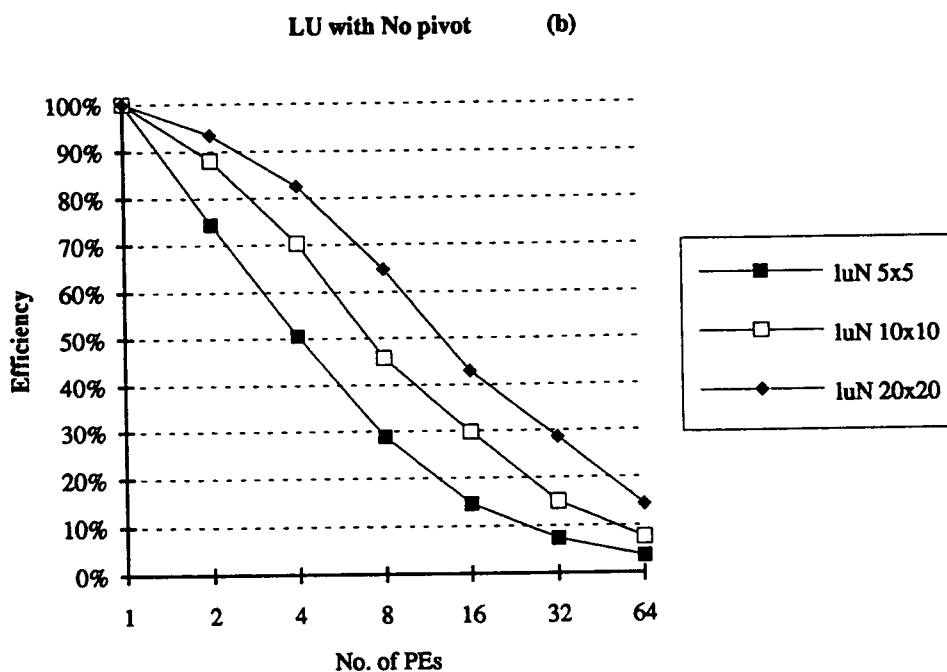


figure 5.16 LU with NO Pivot -- Corresponding Efficiency curves

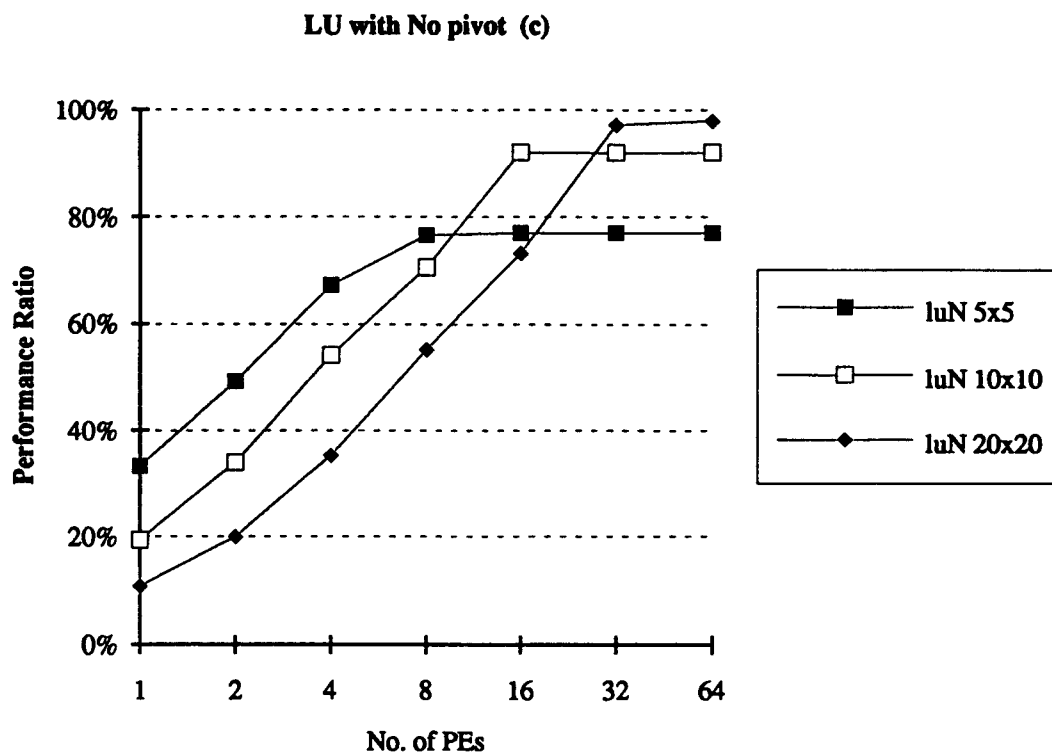


figure 5.17 LU with NO Pivot -- Performance Ratio curves

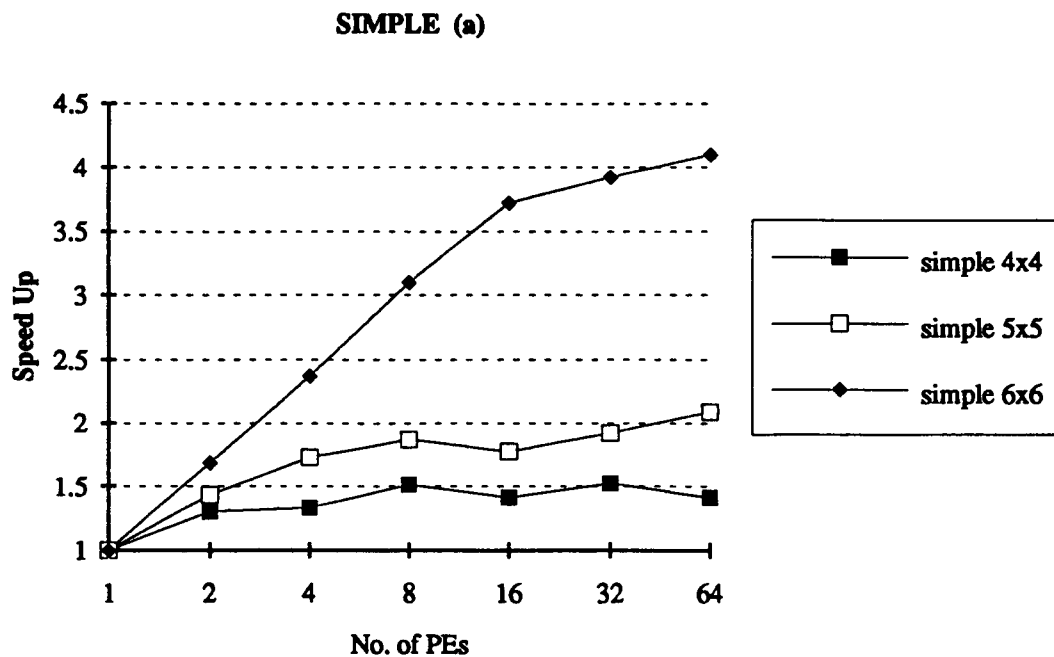


figure 5.18 Simple -- Speed-up curves

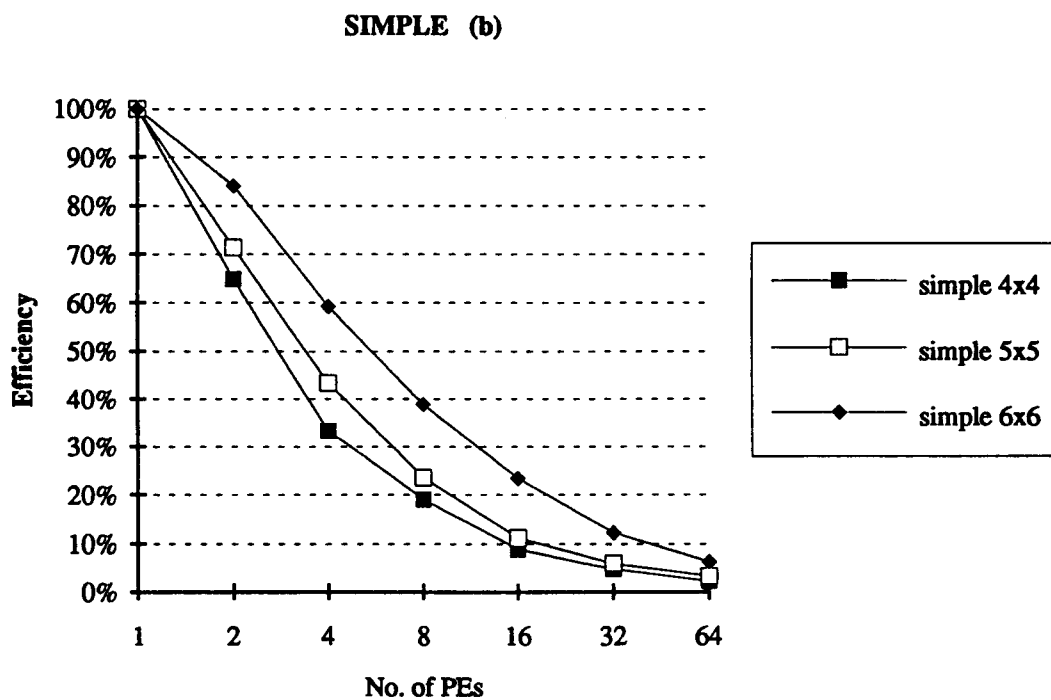


figure 5.19 Simple -- Corresponding Efficiency curves

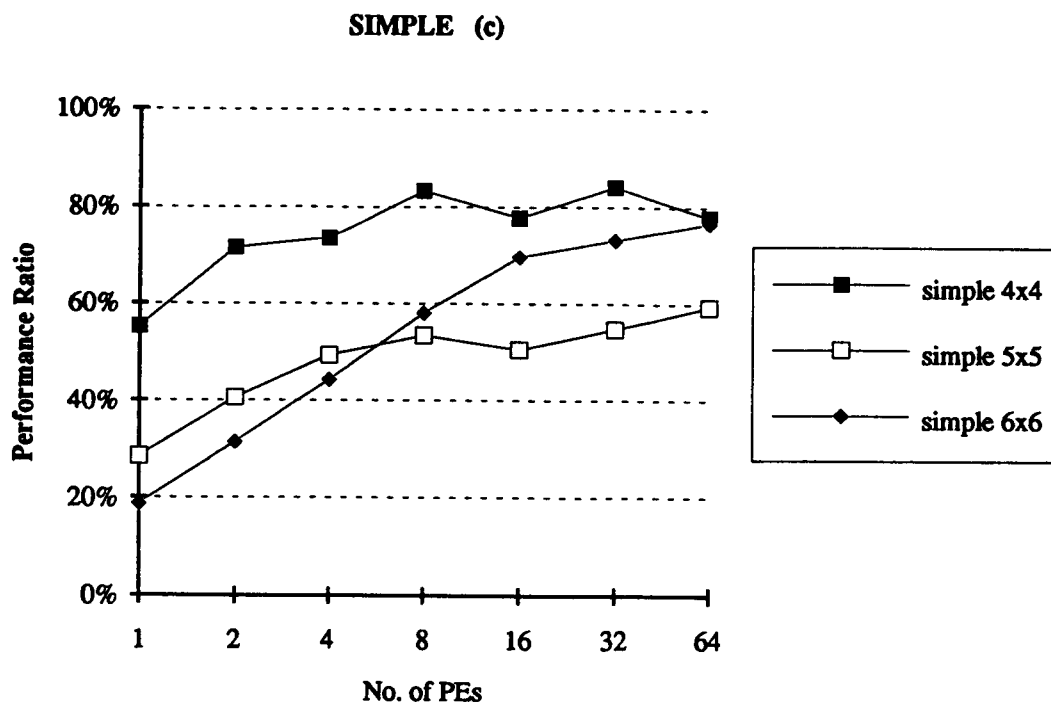


figure 5.20 Simple -- Performance Ratio curves

Matrix Multiplication					
Size	10x10	15x15	20x20	25x25	30x30
C/T(av)	2.61	1.71	1.3	1.04	0.87
Number of Nodes	134	274	464	704	994
Sequential Exec.	13330	40995	92660	175825	297990
Ideal Speed-up	18.488	37.923	64.3	97.626	137.89
Max. Parallelism	100	225	400	625	900
LDP Count	CP+108	CP+238	CP+418	CP+648	CP+928
Horizontal Layers	8	8	8	8	8

LU with Pivot			
Size	5x5	10x10	20x20
C/T(av)	5.05	1.42	0.41
Number of Nodes	228	613	1983
Sequential Exec.	11748	112018	1268358
Ideal Speed-up	4.128	8.874	21.049
Max. Parallelism	55	210	820
LDP Count	CP+131	CP+456	CP+1706
Horizontal Layers	43	43	43

LuN with No Pivot			
Size	5x5	10x10	20x20
C/T(av)	3.55	1.12	0.35
Number of Nodes	85	265	925
Sequential Exec.	6221	61336	689606
Ideal Speed-up	3.013	5.186	9.378
Max. Parallelism	50	200	800
LDP Count	CP+49	CP+199	CP+799
Horizontal Layers	10	10	10

Simple BenchMark			
Size	4x4	5x5	6x6
C/T(av)	8.33	7.73	3.75
Number of Nodes	1375	3135	5547
Sequential Exec.	42903	105435	384953
Ideal Speed-up	1.812	3.546	5.355
Max. Parallelism	59	152	256
LDP Count	CP+486	CP+1211	CP+2056
Horizontal Layers	323	325	335

Table 5a Parameter values from Simulations

5.10 C/T

The Communication to Execution Ratio is given as C/T_{av} in the Table 5a. This ratio indicates the granularity of the program that is being run. The Communication parameter : C , is averaged out as a constant, and is equal to 'Startup time + One Hop Time' (i.e., 260 cycles). The execution time T_{av} (also referred as T) is the average execution time of a particular node in that problem and is equal to the ratio of sequential time of a graph to the number of nodes in that graph.

For the same problem, the C/T_{av} decreases for larger sizes, but for a given PE, the speed-up and efficiency increases. Thus, smaller the grain and larger the size of the program, better it is suited for allocation over parallel processors. This is only an approximation and each case may be validated for the right number of hops when deducing the T_{av} parameter.

Consider the 3 cases of 'LU with pivot' algorithm being allocated over 64 PEs. The performance tables for 'Any' and 'Adjacent' assignment is as given.

	lu 5x5	lu 10x10	lu 20x20
C/T_{av}	5.05	1.42	0.41
Adj. PE Speed-up	2.52	5.6	9.47
Any PE Speed-up	3.00	6.97	19.4

Table 5b Speed up for the **lu nxn** at 64 PEs

Modified BLAS or Adj. PE allocation does not perform as well as the BLAS (Any allocation) for small Communication to Execution ratios (C/T) but, performs as well as BLAS for larger C/T ratios. This is in congruence with the findings in [12]. A comparative effect of running the simulations with Adjacent and Any allocation is presented below.

5.11 Adjacent versus Any

The 'Any Processor' allocation scheme has far outweighed the Adjacent scheme in the simulation results. The performance curves for an 'adjacent' allocated **mat 30x30** program in comparison to 'any' allocated 15x15, 20x20 and 25x25 is shown in figure 5.21. These curves indicate that BLAS performs well, as it attempts to find the best allocation for each LDP over all the different PEs, by considering the effects of communication costs on execution times. Modified BLAS, has one important difference, each LDP is assigned in an iterative fashion to a PE layer that is at most one processor away from its parent thread.

Comparing LU (with Pivot) with matrix multiplication brings to surface an important aspect of parallelism. The LU for a size of 20x20 allocated over 64 PEs, reaches 92.2% of its *ideal* speed-up. For the same case in matrix multiplication, the speed-up ratio is as low as 40% of the *ideal*. A plausible explanation is that matmult problems have a very high *ideal* speed-up and lower number of LDPs to distribute over PEs. In contrast, the LU problem has low *ideal* speed-ups and many more LDPs in comparison to allocate. *Ideal* speed-up for LU is nearly a third of the number of PEs (i.e. 21) in contrast to **mat 20x20** which has almost the same *ideal* speed-up as the number of processors over which it is being allocated (i.e., 64.3 over 64 PEs).

Hence, if an algorithm has a higher ideal speed-up it must have a large number of LDPs so as to saturate faster over increasing number of PEs. However, this is not the case with the Simple benchmark. The only explanation to this anomaly is that the results are taken for 'Adjacent' allocation and the above is true for 'Any Processor' allocation.

Also, from the discussion on C/T, Adjacent schemes give better speed-ups only for high C/T ratios. But, in these current simulations, C/T is high for smaller sized problems, which any way compile in less time using 'Any' PE allocation. Hence, using 'Adjacent' schemes has the advantage of saving time during the assignment phase of large sized problems.

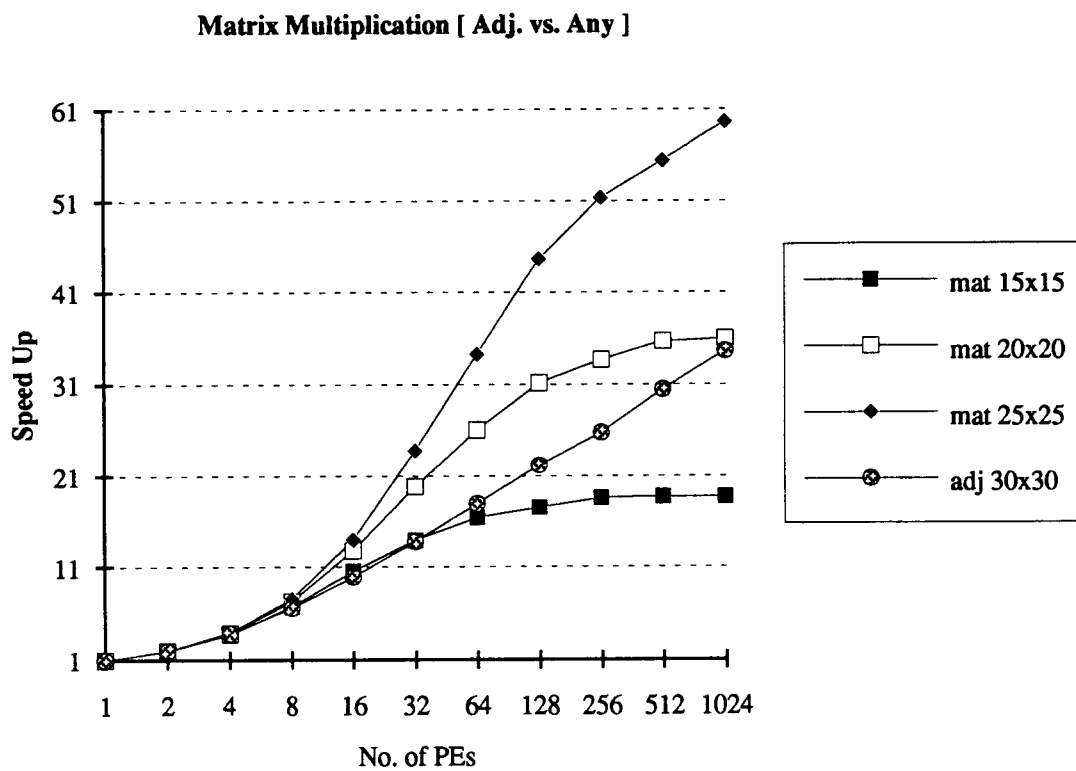


figure 5.21 Curves comparing 'Adj' with 'Any' allocations

CHAPTER 6. CONCLUSIONS and FUTURE WORK

6.1 A Review

In this work, concepts of parallelism have been described in terms of the available parallel architectures and their interconnect and routing issues. The characteristics of parallelism have prompted researchers to augment available high level languages to describe parallelism within a code. Alternatively, many have tried bestowing the power of automatic restructuring to a compiler, which transforms a given sequential algorithm into its parallel form using compilation and optimization techniques. This desire for expressing parallelism finally led to the making of parallel functional languages. Though these languages for example SISAL are still in their infancy, the future is promising. This thesis was developed around the compilation of SISAL programs.

Once the expression for parallelism is achieved, the next step is to partition the program into tasks and then assign these tasks to different PEs interconnected in a particular topology. The aspect of presenting a parallel program to a system of PEs in a hypercube topology is where this thesis takes root.

The IF1 graphs are processed by the Automator package developed for this project. It takes into account all implied dependencies between IF1 function graphs and produces a single graph with actual dependencies between the all the nodes. The IF1OPT files serve better for the metamorphosis of multiple acyclic unconnected graphs to a static connected graph. Many features have been incorporated into the graph preprocessor tool, to help experiment with different algorithms and various graph cases.

The Automator tool has opened new avenues to continue research of the Balanced Layered Allocation Scheme that was proposed by [12 and 18]. However, any research is open ended, and its openness lies in the limitations of the current work. BLAS was ported to UNIX and changes were made in the code and the user-interface, resulting in a newer **version 5.0**. Important routines of partitioning-phase were changed, making it more stable over all cases of graphs.

Finally, simulations were carried out with various algorithms each with different sizes. These simulations were timed to see the effective preprocessing time that will be associated when doing a practical static scheduling for any graph. The 'Any' PE allocation fares better for performance and efficiency considerations, but Adjacent assignment becomes a default requirement for very large graphs.

6.2 Future Study

The static allocation scheme needs to be compared with existing and new dynamic allocation approaches. The pragmatics of both have to be further investigated. This will involve focusing on their implementations. In dynamic scheduling, the critical aspect is the run time overhead, which dictates program and machine performance as well as the allowable level of task granularity.

An important issue in parallelization is that of task definition and size. Should they be defined at compile time as was done in BLAS or should they be formed during run time or else can both be used, with defining the tasks at compile time and doing effective resizing during run time to balance the load on the system ?. Also, the larger the task granularity, more significant is the load balancing issue and if the overhead of dispatching a task to a PE is significant then larger grain size is practical to shadow this overhead with gains in execution speed. This has been seen in the case of **mat 3x3.i** in figure 5.7 and figure 5.8, where the bodies were subjected to node packing by the Automator before presenting for allocation. Minimizing the overhead incurred from communication, synchronization and scheduling, is a non-trivial optimization problem and is proved to be NP-Complete [25, 26 and 30]. Moreover, attempting to minimize the communication costs by merging nodes of a graph to avoid the communication overhead, often reduces the degree of available vertical parallelism. Hence, many tradeoffs have to be made for optimality.

6.3 Improvements

The *ForAll* node is well defined structure of a parallel loop. The parallel loops account for the greatest percentage of program parallelism [25].

The loops : LoopA and LoopB have loop-carried dependencies and so, cannot be parallelized. However, the dependencies are not unknown. They are manifest in the code. A dependency exists wherever the key word "old" is used. "old x" refers to the value of x on the previous iteration, while x refers to the value on the current iteration. SISAL being a functional language, there are **NO** unknown dependencies [8 and 33]. By unrolling LoopA and LoopB constructs, one can realize a type of pipeline parallelism. That is, instructions on the next iteration may execute as soon as the "old" values they reference are available, and do not have to wait for all of the previous iterations to finish. Such an implementation requires hardware control to fire off instructions. On

conventional machines such hardware does not exist and is very expensive to implement in software. Multithreaded machines can support pipeline parallelism.

The Automator does encourage some approximations, or alternatively, heavy user interaction with respects to Loop structures. When a body count is not known in advance some technique should be incorporated in the Automator to compute the body count automatically for a given case. The feasibility of this change needs more attention.

When assigning probability to a Select structure, all the nested levels within its True and False graphs have their execution-times multiplied by the probability factor assigned to that True or False node respectively. This probability factor is also applicable to the edges, but BLAS has to be enhanced to indicate that. Within the Automator, an additional field in the Node structure can be assigned to indicate the probability of the incoming edge to that node. A node cannot have both edges, i.e., with True and False as its imports. It will have only one of them. With this implementation in place, the node execution-times need not be multiplied with the probability factor but instead have their actual costs considered during allocation. BLAS will utilize the edge information to allocate the nodes to different PE layers using C/T ratios. This will affect the priority in the assignment phase.

Finally, mapping of IF1 nodes to a message passing system will give a better picture about how effective is the transformation and whether BLAS is at all a good scheme for multicomputers. As of now, the results look good in the simulations. Here, it is to be noted that SISAL and IF1 were originally written with shared memory processor systems in mind, even though the intermediate compilation produced a machine independent graphical form.

6.3 Similar Work

Parallel work has excited researchers to explore different angles with which to get optimal results for a given problem. The contributions of the data flow community has been immense. Lubomir et al [20] describe in their paper, a scheme to subdivide a data flow graph generated using Id, into Subcompact Processes (SP) and then assign these SPs to a target architecture like Intel's iPSC/2. The dataflow program is executed as a collection of SPs and this system is called PODS or Process Oriented Dataflow System.

Similarly Santosh and his group [31] have worked on a scheduler that allocates IF2 graphs [35] generated from SISAL, to an Intel Touchstone i860 system. The above two [20, 31] and [10, 11 and 21] are steps towards proving how data flow concepts can be

utilized on available von Neumann systems using shared memory architecture. Where as this thesis addresses the distributed memory architecture systems.

The simulation results obtained in this project, were compared (by Dr. Lee : Electrical and Computer Engineering Dept., OSU) with hand analysis calculations. The calculations were obtained from Striped partitioning concepts explained in chapter 4 (section 4.12) of the matrix multiplication program implemented on a hypercube network [35]. The computations were done for an order of n^2 , and they match quite closely to the Simulation results as shown. The IF1OPT phase during compilation, reduces the order of the program from n^3 to n^2 for large sizes (i.e., for 15x15 and upwards upto 30x30). After matrix size of 30x30, the IF1OPT leaves the three nested levels as is. This is because, the SISAL group incorporated some checks within the compilation steps to limit the size of the output files when any nested levels are opened and expanded. Table 7a shows a good matching of the simulation results with hand calculations using Striped Partitioning concepts.:

	mat 15x15		mat 20x20		mat 30x30	
PEs	Calc.	Simul.	Calc.	Simul.	Calc.	Simul.
1	1.0	1.0	1.0	1.0	1.0	1.0
2	2.0	1.95	2.0	1.97	2.0	1.98
4	3.7	3.7	3.9	3.84	3.0	3.9
8	6.6	6.7	7.3	7.28	7.6	7.55
16	10.4	10.59	12.9	12.81	14.2	14.0
32	13.8	13.9	20.3	19.83	24.5	23.79
64	15.5	16.4	27.1	26.0	37.3	34.3
128	15.6	17.55	31.1	31.17	48.5	44.66
256	14.8	18.56	31.9	33.62	54.6	51.37

Table 6a Calculated values for Verification

Bibliography

- [1] Ackerman W. B., "Data Flow Languages", *IEEE Computer*, February 1982, pg. 15-25.
- [2] Agerwala T. and Arvind, "Data Flow Systems", *IEEE Computer*, February 1982, pg. 10-13.
- [3] Arvind and Culler, D. E., "Dataflow Architectures", *Annual Review of Computer Science*, 1986, Vol. 1, pg. 225-253.
- [4] Arvind and Ekanadham K., "Future Scientific Programming on Parallel Machines", *Journal of Parallel and Distributed Computing*, November 1988, pg. 460-493.
- [5] Arvind and Iannucci R. A., "Two Fundamental Issues in Multiprocessing", In *Proceedings of DFLVR - Conference 1987 on Parallel Processing in Science and Engineering*, Bonn, Germany, June 1987.
- [6] Arvind and Nikhil, R. S., "Executing a Program on the MIT Tagged-Token Dataflow Architecture", *IEEE Transactions on Computers*, March 1990, Vol. 39, No. 3, pg. 300-318.
- [7] Buehrer R. and Ekanadham K., "Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution", *IEEE Transactions on Computers*, December 1987, Vol. C-36, No. 12, pg. 1515-1522.
- [8] Cann D. C., "The Optimizing SISAL Compiler : version 12.0", *Manual L-306*, Lawrence Livermore National Laboratory, 1985.
- [9] Crowley W. P, Hendrickson C. P. and Rudy T., "The SIMPLE code", *Technical Report UCID 17715*, Lawrence Livermore National Laboratory, February 1978.
- [10] Culler D. E., Schauser K. E. and Eicken V. T., " Two Fundamental Limits on Dataflow Multiprocessing", *Technical Report UCB/CSD 92/716*, Computer Science Division, University of California at Berkeley.
- [11] Dinning A., "A Survey of Synchronization Methods for Parallel Computers", *IEEE Computer*, July 1989, pg. 66-77.
- [12] Freytag V. R., "Program Allocation for Hypercube Based Dataflow Systems", *Master's Thesis*, Electrical and Computer Engineering, Oregon State University, March 18, 1993.
- [13] Gaudiot J. and Lubomir B., *Advanced Topics in Data-Flow Computing*, Prentice Hall Inc., 1991.
- [14] Hennesy J. L. and Patterson D. A., *Computer Architecture: A Quantitative Approach.*, San Mateo, California: Morgan Kaufmann Publishers, Inc. 1990.
- [15] Hwang K., *Advanced Computer Architecture*, McGraw Hill Inc., 1993.

- [16] Hwang K. and Briggs F. A., *Computer Architecture and Parallel Processing* McGraw-Hill, Inc. 1984.
- [17] Iannucci R. A. "Toward a Dataflow/von Neumann Hybrid Architecture", In *Proceedings of the 15th International Symposium on Computer Architecture*, 1988, pg. 131-140.
- [18] Lee B., Hurson A. R., and Feng T. Y., "A Vertically Layered Allocation Scheme for Data Flow Systems," *Journal of Parallel and Distributed Computing*, 1991, Vol. 11, pg. 175-187.
- [19] Lubeck O. M., "A User's View of Dataflow Architectures", *Proceedings of Comcon90*, March 1990, pg. 84-87.
- [20] Lubomir B., Roy J. M. and Nagel M., "Exploiting Iteration-Level Parallelism in Dataflow Programs", *Technical Report 91-57*, Computer Science, University of California, Irvine, CA, 1991.
- [21] Nikhil R. S., and Arvind, "Can Dataflow Subsume von Neumann Computing?", In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989.
- [22] Oldehoeft R. R., Cann D. C., Feo J. T. and Bohm A. P. W., "The SISAL 2.0 reference manual", *Technical Report UCRL-53745*, Lawrence Livermore National Laboratory, CA, December 1991.
- [23] Papadopoulos G. M. and Culler D. E., "Monsoon: An Explicit Token Store Architecture", *1990 IEEE 17th Annual International Symposium on Computer Architecture*, May 1990, pg. 82-91.
- [24] Papadopoulos G. M. and Traub K. R., "Multithreading: A Revisionist View of Dataflow Architectures", *18th Annual International Symposium on Computer Architecture*, 1991, pg. 342-351.
- [25] Polychronopoulos C. D., *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.
- [26] Polychronopoulos C. D., Bannerjee U., "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds", *IEEE Transactions on Computers*, Volume C-36, Issue 4, April 1987.
- [27] Quinn M. J., *Designing Efficient Algorithms for Parallel Computers*, 2nd ed., New York: McGraw-Hill Book Company, 1991.
- [28] Saad Y. and Schultz M. H., "Topological Properties of Hypercubes", *IEEE Transactions on Computers*, July 1988, Vol. 37, No. 7, pg. 867-872.
- [29] Saavedra-Barrerra R., Culler D. E. and Eicken V. T., "Analysis of Multithreaded Architectures for Parallel Computing", In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architecture*, July 1990.
- [30] Sarkar V. and Hennessy J., "Compile-time Partitioning and Scheduling of Parallel Programs", *Proceedings of the Symposium on Compiler Construction*, 1986, pg. 17-26.

- [31] Santoshkumar S. P., Agarwal D. P. and Maunney Jon, "A New Threshold Scheduling Strategy for Sisal programs on Private Memory Machines", In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, 1993, Vol. II, pg. 536-545.
- [32] Schauser K. E., Culler D. E. and Eicken V. T., "Compiler Controlled Multithreading for Lenient Parallel Languages", In *Proceedings of 1991 Conference on Functional Programming Language and Computer Architecture*, Cambridge, MA, August 1991.
- [33] Skedzielewski S. K. and Glauret J., "IF1 - An intermediate form for applicative languages", *Manual M-170*, Lawrence Livermore National Laboratory, 1985.
- [34] V. Rajaraman, *Computer Oriented Numerical Methods*, Prentice Hall India, 1988.
- [35] Vipin. K, Grama A, Gupta A and Karypis G, *Introduction To Parallel Computing*, The Benjamin/Cummings Publishing Company Inc., 1994.
- [36] Welcome M. L., Skedzielewski S. K., Yates R. K. and Ranelletti J. E., "IF2: an applicative language intermediate form with explicit memory management", *Manual M-195*, Lawrence Livermore National Laboratory, 1986.