Program and Network Properties

Here we discuss about fundamental properties of program behavior and introduces major classes of interconnection networks. We begin with a study of computational granularity, conditions for program partitioning, matching software with hardware, program flow mechanisms, and compilation support for parallelism. Interconnection architectures introduced include static and dynamic networks. Network complexity, communication bandwidth, and data-routing capabilities are discussed.

Conditions of Parallelism

The exploitation of parallelism has created a new dimension in computer science. In order to move parallel processing into the mainstream of computing, H.T. Kung (1991) has identified the need to make significant progress in three key areas: *computation models* for parallel computing, *interprocessor communication* in parallel architectures, and *system integration* for incorporating parallel systems into general computing environments.

A theoretical treatment of parallelism is thus needed to build a basis for the above challenges. In practice, parallelism appears in various forms in a computing environment. All forms can be attributed to levels of parallelism, computational granularity, time and space complexities, communication latencies, scheduling policies, and load balancing. Very often, tradeoffs exist among time, space, performance, and cost factors.

Data and Resource Dependences

The ability to execute several program segments in parallel requires each segment to be independent of the other segments. The independence comes in various forms as defined below separately. For simplicity, to illustrate the idea, we consider the dependence relations among instructions in a program. In general, each code segment may contain one or more statements.

We use a *dependence graph* to describe the relations. The nodes of a dependence graph correspond to the program statements (instructions), and the directed edges with different labels show the ordered relations among the statements. The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

Data Dependence: The ordering relationship between statements is indicated by the data dependence. Five types of data dependence are defined below:

(1) Flow dependence: A statement S2 is *flow-dependent* on statement S1 if an execution path exists from S1 to S2 and if at least one output (variables assigned) of S1 feeds in as input (operands to be used) to S2. Flow dependence is denoted as SI → S2.

(2) *Antidependence:* Statement S2 is *antidependent* on statement S1, if S2 follows S1 in program order and if the output of S2 overlaps the input to S1. A direct arrow crossed with a bar as in S1 -I—> S2 indicates antidependence from Si to S2.

(3) *Output dependence:* Two statements are *output-dependent* if they produce (write) the same output variable. S1 → S2 indicates output dependence from S1 to S2.

(4) *I/O dependence:* Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file is referenced by both I/O statements.

(5) *Unknown dependence:* The dependence relation between two statements cannot be determined in the following situations:

- The subscript of a variable is itself subscribed (indirect addressing).
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is nonlinear in the loop index variable.

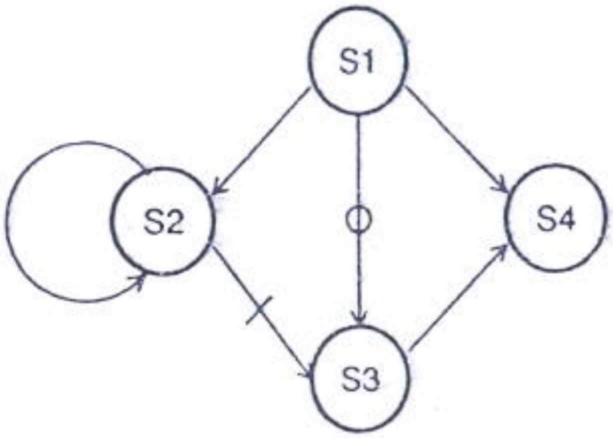
When one or more of these conditions exist, a conservative assumption is to claim unknown dependence among the statements involved.

Example : Data dependence in programs

Consider the following code fragment of four instructions:

```
S1 :    Load R1,A               /R1 ← Memory(A)/
S2 :    Add R2,R1               /R2← (R1) +(R2)/
S3 :    Move R1 ,R3             /R1← (R3)/
S4 :    Store B,R1              /Memory(B) ← (R1)/
```

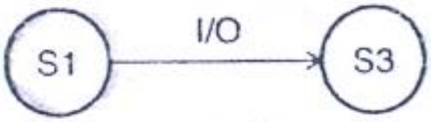
As illustrated in diagram (a), S2 is flow-dependent on S1 because the variable A is passed via the register Rl. S3 is antidependent on S2 because of potential conflicts in register content in R1. S3 is output-dependent on S1 because they both modify the same register Rl. Other data dependence relationships can be similarly revealed on a pairwise basis. Note that dependence is a partial ordering relation; that is, the members of not every pair of statements are related. For example, the statements S2 and S4 in the above program are totally *independent.*



(a) Dependence graph

Next, we consider a code fragment involving I/O operations:

```
S1:             Read (4) A(I)
S2:             Rewind(4)
S3:             Write (4) B(I)
S4:             Rewind(4)
```



(b) I/O dependence caused by accessing the same file by the read and write statements

From the above diagram, the read/write statements, S1 and S3, are I/O-dependent on each other because they both access the same file from tape unit 4. The above data dependence relations should not be arbitrarily violated during program execution. Otherwise, erroneous results may be produced with changed program order. The order in which statements are executed in a program is often well defined. Repetitive runs should produce identical results. On a multiprocessor system, the program order may or may not be preserved, depending on the memory model used. Determinism yielding predictable results can be controlled by a programmer as well as by constrained modification of writable data in a shared memory.

Control Dependence: This refers to the situation where the order of execution of statements cannot be determined before run time. For example, conditional statements (IF in Fortran) will not be resolved until run time. Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions. Dependence may also exist between operations performed in successive iterations of a looping procedure. In the following, we show one loop example with and another without control-dependent iterations. The successive iterations of the following loop are *control-independent:*

```
        DO 20 I = 1, N
        A(I) = C(I)
        IF (A(I) .LT. 0) A(I) = 1
 20 Continue
```

The following loop has *control-dependent* iterations:

```
        Do 10 I = 1, N
        IF (A(I - 1) .EQ. 0)
         A(I) = 0
10      Continue
```

Control dependence often prohibits parallelism from being exploited. Compiler techniques are needed to get around the control dependence in order to exploit more parallelism.

Resource Dependence : This is different from data or control dependence, which demands the independence of the work to be done. *Resource dependence* is concerned with the conflicts in using shared resources, such as integer units, floating-point units, registers, and memory areas, among parallel events. When the conflicting resource is an ALU, we call it *ALU dependence.*
If the conflicts involve workplace storage, we call it *storage dependence.*

The transformation of a sequentially coded program into a parallel executable form can be done manually by the programmer using explicit parallelism, or by a compiler detecting implicit parallelism automatically. In both approaches, the decomposition of programs is the primary objective.
Program partitioning determines whether a given program can be partitioned or split into pieces that can execute in parallel or follow a certain pre specified order of execution.

Bernstein's Conditions: In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel. A *process* is a software entity corresponding to the abstraction of a program fragment defined at various processing levels. We define the *input set I,* of a process *Pi* as the set of all input variables needed to execute the process. .

Similarly, the *output set* 0, consists of all output variables generated after execution of the process *P,*. Input variables are essentially operands which can be fetched from memory or registers, and output variables are the results to be stored in working registers or memory locations.

Now, consider two processes *P1* and *P*2 with their input sets *I* and J; and output sets *O1* and *O2*, respectively. These two processes can execute in parallel and are denoted *Pi* ‖ *Pj* if they are independent and do not create confusing results.

Formally, these conditions are stated as follows:

$$\left.\begin{array}{lllll} I_1 & \cap & O_2 & = & \emptyset \\ I_2 & \cap & O_1 & = & \emptyset \\ O_1 & \cap & O_2 & = & \emptyset \end{array}\right\}$$

These three equations are known as *Bernstein's conditions.* The input set I1, is also called the *read set* or the *domain* of *Pi* by other authors. Similarly, the output set O1 has been called the *write set* or the *range* of a process *P,*. In terms of data dependences, Bernstein's conditions simply imply that two processes can execute in parallel if they are flow-independent, antiindependent, and output-independent.

The parallel execution of two processes produces the same results regardless of whether they are executed sequentially in any order or in parallel. This is possible only if the output of one process will not be used as input to the other process. Furthermore, the two processes will not modify (write) the same set of variables, either in memory or in the registers.

In general, a set of processes, *P*1, P2…. Pk, can execute in parallel if Bernstein's conditions are satisfied on a pairwise basis; that is, *P1* || *P2* || P3 || - - • || Pk if and only if *Pi* || *Pj* for all *i≠ j.*

In general, the parallelism relation || is commutative; i.e., P, || Pj implies *Pj* || P,. But the relation is not transitive; i.e., Pi || *Pj* and *Pj* || Pk do not necessarily guarantee *Pi* || Pk.. For example, we have *P1* || P5 and P5 || P2, but *P1≠* P2, where || means *P1*and P2 cannot execute in parallel. In other words, the order in which *P* and P2 are executed will make a difference in the computational results.

Hardware and Software Parallelism

For implementation of parallelism, we need special hardware and software support. The key idea being conveyed is that parallelism cannot be achieved free. Besides theoretical conditioning, joint efforts between hardware designers and software programmers are needed to exploit parallelism in upgrading computer performance.

Hardware Parallelism : This refers to the type of parallelism defined by the machine architecture and hardware multiplicity. Hardware parallelism is often a function of cost and performance tradeoffs. It displays, the resource utilization patterns of simultaneously executable operations. It can also indicate the peak performance of the processor resources.

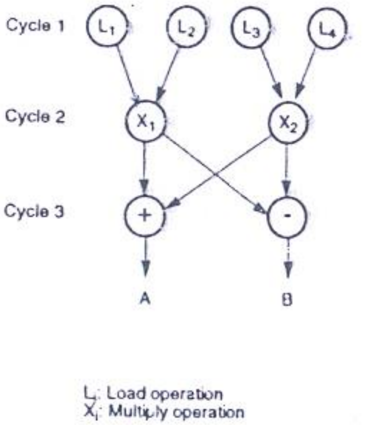
One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle. If a processor issues *k* instructions per machine cycle, then it is called a *k-issue* processor.

A conventional processor takes one or more machine cycles to issue a single instruction. These types of processors are called *one-issue* machines, with a single instruction pipeline in the processor. In a modern processor, two or more instructions can be issued per machine cycle.

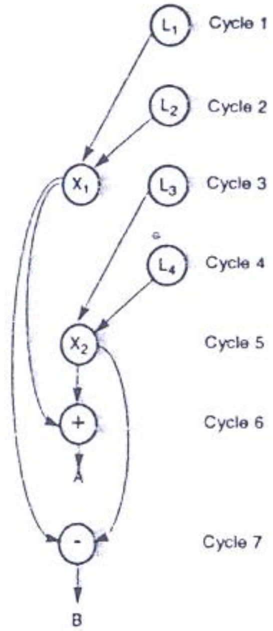
A multiprocessor system built with n fc-issue processors should be able to handle a maximum number of *nk* threads of instructions simultaneously.

Software Parallelism :This type of parallelism is defined by the control and data dependence of programs. The degree of parallelism is revealed in the program profile or in the program flow graph'.- Software parallelism is a function of algorithm, programming style, and compiler optimization. The program flow graph displays the patterns of simultaneously executable operations. Parallelism in a program varies during the execution period. It often limits the sustained performance of the processor.

Example : Mismatch between software parallelism and hardware parallelism (Wen-Mei Hwu, 1991)
Consider the example program graph in following diagram (a) . There are eight instructions (four *loads* and four *arithmetic* operations) to be executed in three consecutive machine cycles. Four *load* operations are performed in the first cycle, followed by two *multiply* operations in the second cycle and two *add/subtract* operations in the third cycle. Therefore, the parallelism varies from 4 to 2 in three cycles. The average software parallelism is equal to 8/3 = 2.67 instructions per cycle in this example program.



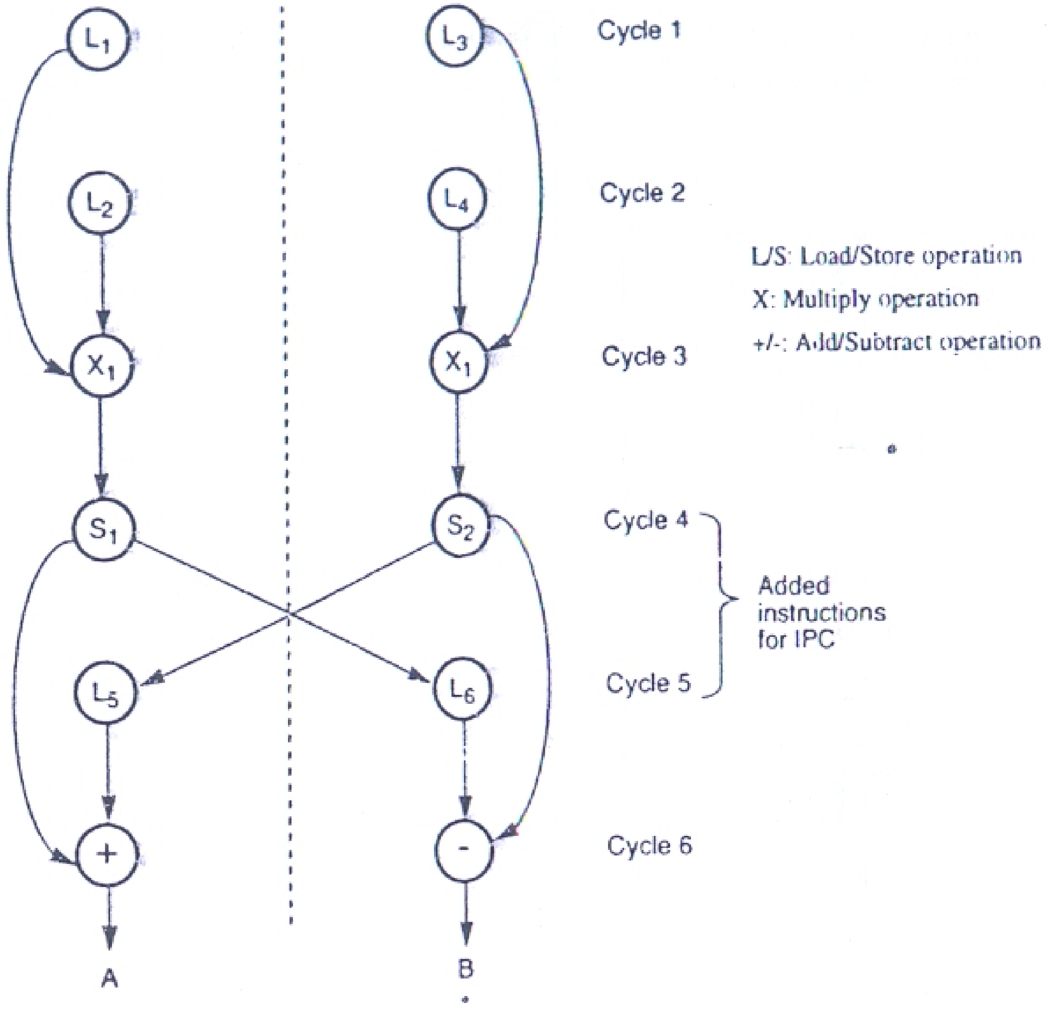
(a) Software parallelism



(b) Hardware parallelism

Now consider execution of the same program by a two-issue processor which can execute one memory access (*load* or *write)* and one arithmetic *(add, subtract, multiply,* etc.) operation simultaneously. With this hardware restriction, the program must execute in seven machine cycles as shown in above digram(b) . Therefore, the *hardware parallelism* displays an average value of 8/7 =1.14 instructions executed per cycle. This demonstrates a mismatch between the software parallelism and the hardware parallelism.
Let us try to match the software parallelism shown in diagram (a) in a hardware platform of a dual-processor system, where single-issue processors are used.

The achievable hardware parallelism is shown in following diagram. where *L/S* stands for *load/store* operations. Note that six processor cycles are needed to execute the 12 instructions by two processors. *S1* and *S*2 are two inserted *store* operations, and l5 and l*6* are two inserted *load* operations. These added instructions are needed for interprocessor communication through the shared memory.



Dual-processor execution of the program

Of the many types of software parallelism, two are most frequently cited as important to parallel programming: The first is *control parallelism,* which allows two or more operations to be performed simultaneously. The second type has been called *data parallelism,* in which almost the same operation is performed over many data elements by many processors simultaneously.

Control parallelism, appearing in the form of pipelining or multiple functional units, is limited by the pipeline length and by the multiplicity of functional units. Both pipelining and functional parallelism are handled by the hardware; programmers need take no special actions to invoke them.

Data parallelism offers the highest potential for concurrency. It is practiced in both SIMD and MIMD modes on MPP systems.

The Role of Compilers

Compiler techniques are used to exploit hardware features to improve performance. The pioneer work on the IBM PL8 and Stanford MIPS compilers has aimed for this goal. Other optimizing compilers for exploiting parallelism include the CDC STACKLIB, Cray CFT, Illinois Parafrase, Rice PFC, Yale Bulldog, and Illinois IMPACT.

Interaction between compiler and architecture design is a necessity in modern computer development. Most existing processors issue one instruction per cycle and provide a few registers. This may cause excessive spilling of temporary results from the available registers. Therefore, more software parallelism may not improve performance in conventional scalar processors.

There exists a vicious cycle of limited hardware support and the use of a naive compiler. To break the cycle, one must design the compiler and the hardware jointly at the same time. Interaction between the two can lead to a better solution to the mismatch problem between software and hardware parallelism.

The general guideline is to increase the flexibility in hardware parallelism and to exploit software parallelism m control-intensive programs. Hardware and software design tradeoffs also exist in terms of cost, complexity, expandability, compatibility, and performance. Compiling for multiprocessors is much more involved than for uniprocessors. Both granularity and communication latency play important roles in the code optimization and scheduling process.

Program Partitioning and Scheduling

Here we introduces the basic definitions of computational granularity or level of parallelism in programs. Communication latency and scheduling issues are illustrated with programming examples.

Grain Sizes and Latency :

Grain size or granularity is a measure oi the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as *fine*, *medium*, or *coarse*, depending on the processing levels involved.

Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the *memory latency* is the time required by a processor to access the memory. The time required for two processes to synchronize with each other is called the *synchronization latency.* Computational granularity and communication latency are closely related.

Parallelism has been exploited at various processing levels. As illustrated in following diagram, five levels of program execution represent different computational grain sizes and changing

communication and control requirements. The lower the level, the finer the granularity of the software processes.
In general, the execution of a program may involve a combination of these levels. The actual combination depends on the application, formulation, algorithm, language, program, compilation support, and hardware limitations.

Instruction Level: At instruction or statement level, a typical grain contains less than 20 instructions, called *fine grain* in the diagram below . Depending on individual programs, fine-grain parallelism at this level may range from two to thousands. Butler efc al. (1991) has shown that single-instruction-stream parallelism is greater than two. Wall (1991) finds that the average parallelism at instruction level is around five, rarely exceeding seven, in an ordinary program. For scientific applications, Kumar (1988) has measured the average parallelism in the range of 500 to 3000 Fortran statements executing concurrently in an idealized environment.

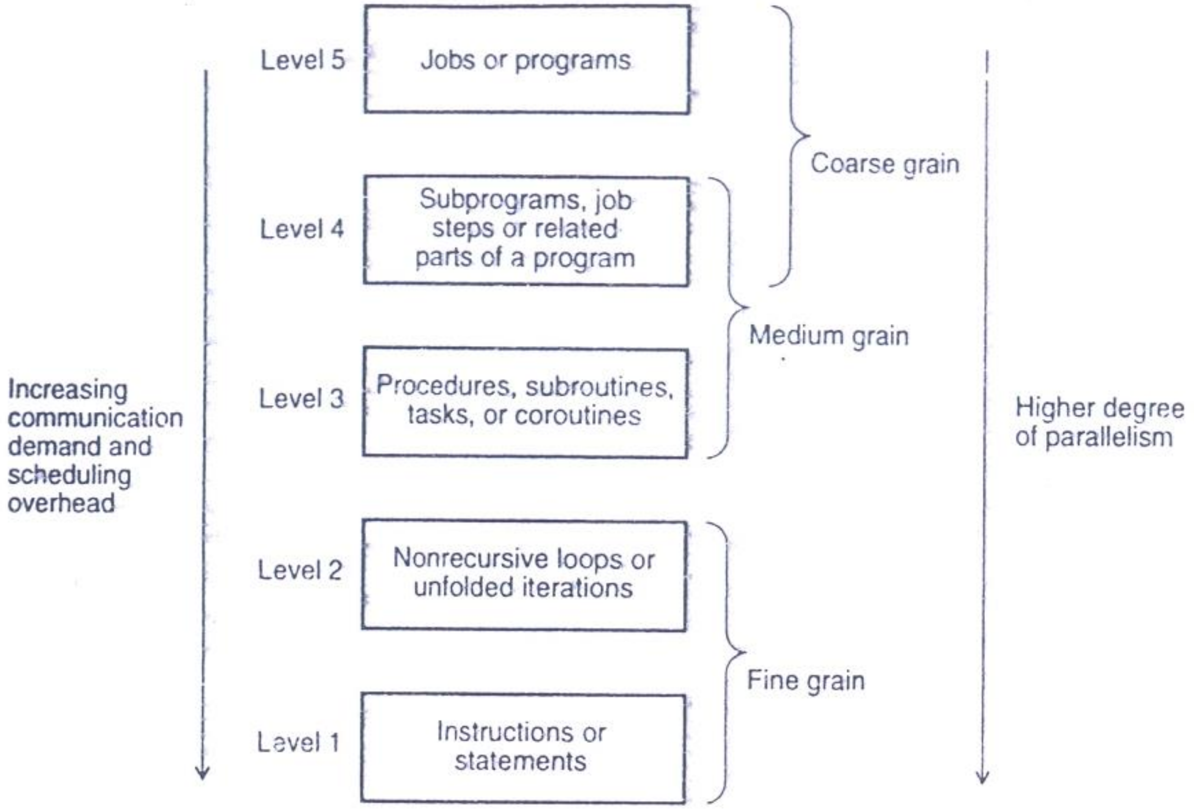
The advantage of fine-grain computation lies in the abundance of parallelism. The exploitation of fine-grain parallelism can be assisted by an optimizing compiler which should be able to automatically detect parallelism and translate the source code to a parallel form which can be recognized by the run-time system.

Loop Level :This corresponds to the iterative loop operations. A typical loop contains less than 500 instructions. Some loop operations, if independent ̇n successive iterations, can be vectorized for pipelined execution or for lock-step execution on SIMD machines. Some loop operations can be self-scheduled for parallel execution on MIMD machines.

Loop-level parallelism is the most optimized program construct to execute on a parallel or vector computer. However, recursive loops are rather difficult to parallelize. Vector processing is mostly exploited at the loop level by a vectorizing compiler. The loop level is still considered a fine grain of computation.

Procedure Level : This level corresponds to medium-grain size at the task, procedural, subroutine, and coroutine levels. A typical grain at this level contains less than 2000 instructions. Detection of parallelism at this level is much more difficult than at the finer-grain levels. Interprocedural dependence analysis is much more involved and history-sensitive.

Subprogram Level : This corresponds to the level of job steps and related subprograms. The grain size may typically contain thousands of instructions. Job steps can overlap across different jobs. Subprograms can be scheduled for different processors in SPMD or MPMD mode.


Level 5
Jobs or programs
Level 4
Subprograms, job steps or related parts of a program
Level 3
Procedures, subroutines, tasks, or coroutines
Level 2
Nonrecursive loops or unfolded iterations
Level 1
Instructions or statements
Coarse grain
Medium grain
Fine grain
Increasing communication demand and scheduling overhead
Higher degree of parallelism

Levels of parallelism in program execution on modern computers.
(Reprinted from Hwang, *Proc. IEEE*, October 1987)

Job (Program) Level: This corresponds to the parallel execution of essentially independent jobs (programs) on a parallel computer. The grain size can be as high as tens of thousands of instructions in a single program. For supercomputers with a small number of very powerful processors, such coarse-grain parallelism is practical. Job-level parallelism is handled by the program loader and by the operating system in general. Time-sharing or space-sharing multiprocessors explore this level of parallelism. In fact, both time and space sharing are extensions of multiprogramming.

To summarize, fine-grain parallelism is oft en exploited at instruction or loop levels, preferably assisted by a parallelizing or vectorizing compiler. Medium-grain parallelism at the task or job step demands significant roles for the programmer as well as compilers. Coarse-grain parallelism at the program level relies heavily on an effective OS and on the efficiency of the algorithm used.

Message-passing multicomputers have been used for medium- and coarse-grain computations. In general, the finer the grain size, the higher the potential for parallelism and the higher the communication and scheduling overhead. Fine grain provides a higher degree of parallelism, but heavier communication overhead, as compared with coarse-grain computations. Massive parallelism is often explored at the fine-grain level, such as data parallelism on SIMD or MIMD computers.

Communication Latency : By balancing granularity and latency, one can achieve better performance of a computer system. Various latencies are attributed to machine architecture, implementing technology, and communication patterns involved. The architecture and technology affect the design choices for latency tolerance between subsystems.

The latency incurred with interprocessor communication is another important parameter for a system designer to minimize. Besides signal delays in the data path, IPC latency is also affected by the communication patterns involved. In general, n tasks communicating with each other may require n(n - l)/2 communication links among them. Thus the complexity grows quadratically. This leads to a communication bound which limits the number of processors allowed in a large computer system.
Communication patterns are determined by the algorithms used as well as by the architectural support provided. Frequently encountered patterns include *permutations* and *broadcast*, *multicast*, and *conference* (many-to-many) communications. The communication demand may limit the granularity or parallelism. Very often tradeoffs do exist between the two. •.

The communication issue thus involves the reduction of latency or complexity, the prevention of deadlock, minimizing blocking in communication patterns, and the tradeoff between parallelism and communication overhead.

Grain Packing and Scheduling

Two fundamental questions to ask in parallel programming are: (i) How can we partition a program into parallel branches, program modules, microtasks, or grains to yield the shortest possible execution time? and (ii) What is the optimal size of concurrent grains in a computation?
This grain-size problem demands determination of both the number and the size of grains in a parallel program. Of course, the solution is both problem- dependent and machine-dependent.

The goal is to produce a short schedule for fast execution of subdivided program modules.
There exists a tradeoff between parallelism and scheduling/synchronization overhead. The time complexity involves both computation and communication overheads The program partitioning involves the algorithm designer, programmer, compiler, operating system support, etc.

Example : Program graph before and after grain packing (Kruatrachue and Lewis, 1988)

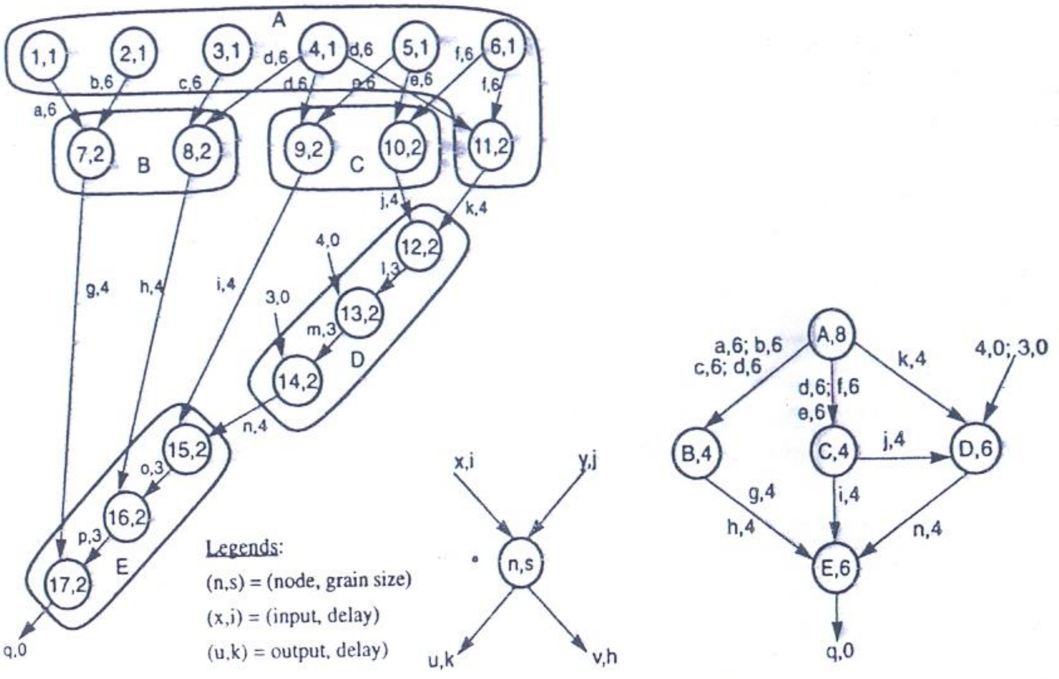
The basic concept of program partitioning is introduced below diagram.We show an example *program graph* in two different grain sizes. A following program graph shows the structure of a program. It is very similar to the dependence graph. Each node in the program graph corresponds to a computational unit in the program. The *grain size* is measured by the number of basic machine cycles (including both processor and memory cycles) needed to execute all the operations within the node.

We denote each node in the diagram by a pail (n,s), where *n* is the *node name* (id) and *s* is the grain size of the node. Thus grain size reflects the number of computations involved in a program segment. Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.

The edge label (v, *d)* between two end nodes specifies the output variable *v* from the source node or the input variable to the destination node, and the communication delay *d* between them. This delay includes all the path delays and memory latency involved.

There are 17 nodes in the fine-grain program graph (refer diagram) and 5 in the coarse-grain program graph (Fig. 2.6b). The coarse-grain node is obtained by combining (grouping) multiple fine-grain nodes. The fine grain corresponds to the following program:

Var *a, b,* c, *d, e,* /, *g, h, i ,j , k ,* /, m, n, o,p, *q*



(a) Fine-grain program graph before packing

(b) Coarse-grain program graph after packing

A program graph before and after grain packing in Example 2.4. (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

```
Begin
        1.    a := 1
        2.    b := 2
        3.    c := 3
        4.    d := 4
        5.    e := 5
        6.    f := 6
        7.    g := a × b
        8.    h := c × d
        9.    i := d × e
        10.   j := e × f
        11.   k := d × f
        12.   l := j × k
        13.   m := 4 × l
        14.   n := 3 × m
        15.   o := n × i
        16.   p := o × h
        17.   q := p × q
End
```

Nodes 1, 2, 3, 4, 5, and 6 are memory reference (data fetch) operations. Each takes one cycle to address and six cycles to fetch from memory. All remaining nodes (7 to 17) are CPU operations, each requiring two cycles to complete. After packing, the coarse-grain nodes have larger grain sizes ranging from 4 to 8 as shown.

The node (A,8) in Fig. 2.6b is obtained by combining the nodes (1,1), (2,1), (3,1), (4,1), (5,1), (6,1), and (11,2) in Fig. 2.6a. The grain size, 8, of node A is the summation of all grain sizes (14-1 + 1 + 1 + 1 + 1+ 2 = 8) being combined.

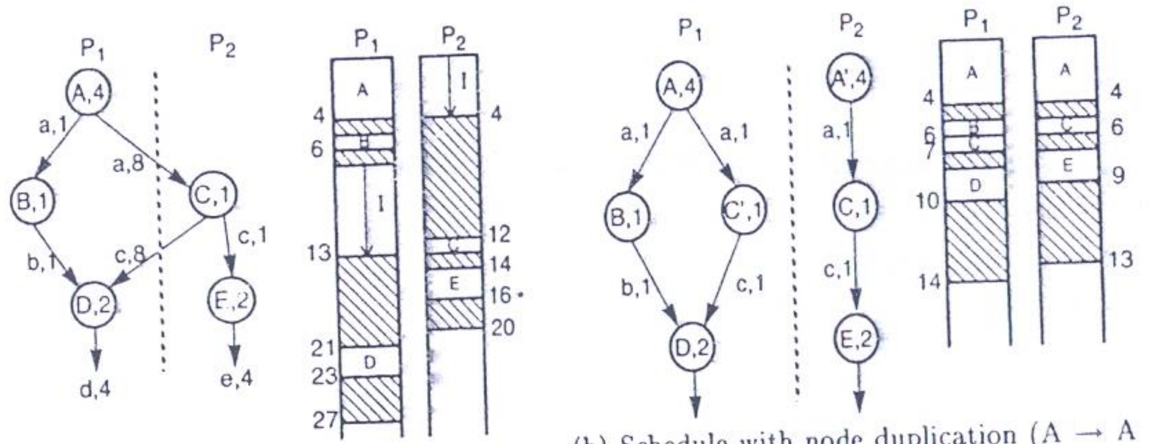
The idea of grain packing is to apply fine grain first in order to achieve a higher degree of parallelism. Then one combines (packs)^ multiple fine-grain nodes into a coarse- grain node if it can eliminate unnecessary communications delays or reduce the overall scheduling overhead.

Internal delays among fine-grain operations within the same coarse-grain node are negligible because the communication delay is contributed mainly by interprocessor delays rather than by delays within the same processor. The choice of the optimal grain size is meant to achieve the shortest schedule for the nodes on a parallel computer system.

Static Multiprocessor Scheduling

Grain packing may not always produce a shorter schedule. In general-, dynamic multiprocessor scheduling is an NP-hard problem. Very often heuristics are used to yield suboptimal solutions. We introduce below the basic concepts behind multiprocessor scheduling using static schemes.

Node Duplication In order to eliminate the idle time and to further reduce the communication delays among processors, one can duplicate some of the nodes in more than one processor.
The following diagram(a) shows a schedule without duplicating any of the five nodes. This schedule contains idle time as well as long interprocessor delays (8 units) between PI and P2. In diagram (b) , node A is duplicated into A' and assigned to P2 besides retaining the original copy A in Pi. Similarly, a duplicated node C' is copied into Pi besides the original node C in P2. The new schedule shown in Fig. 2.8b is almost 50% shorter than that in diagram(a). The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.



(a) Schedule without node duplication

(b) Schedule with node duplication (A → A and A'; C → C and C')

Node-duplication scheduling to eliminate communication delays between processors. (I: idle time; shaded areas: communication delays)

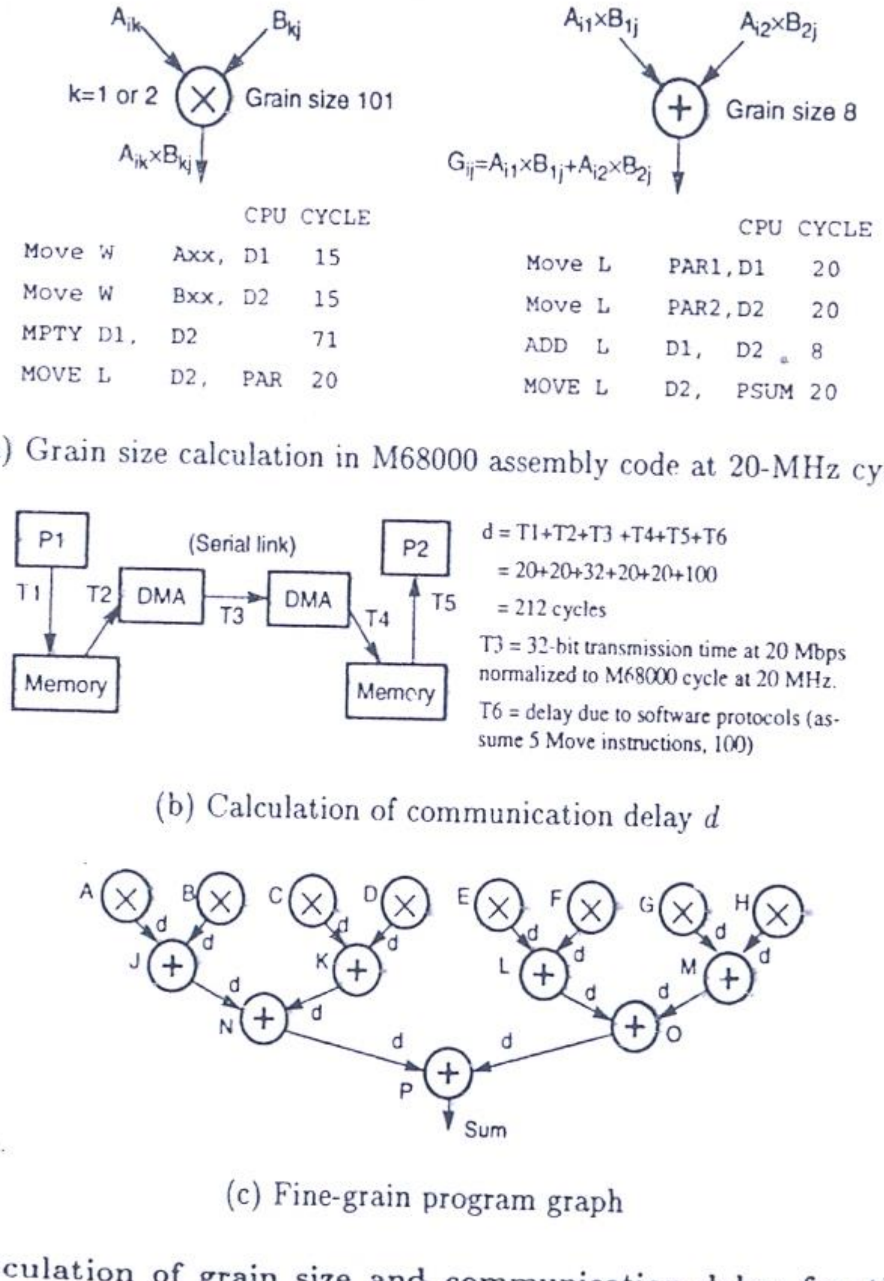
Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule. Four major steps are involved in the grain determination and the process of scheduling optimization;

Step 1. Construct a fine-grain program graph.
Step 2. Schedule the fine-grain computation.
Step 3. Grain packing to produce the coarse grains.
Step 4. Generate a parallel schedule based on the packed graph.

The purpose of multiprocessor scheduling is to obtain a minimal time schedule for the computations involved. The following example clarifies this concept.

Example : Program decomposition for static multiprocessor scheduling (Kruatrachue and Lewis, 1988)
The following diagram shows an example of how to calculate the grain size and communication latency. In this example, two 2x2 matrices *A* and *B* are multiplied to compute the sum of the four elements in the resulting product matrix *C* = *A* x *B*. There are eight multiplications and seven additions to be performed in this program, as written below:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$



```
                     CPU CYCLE
Move W   Axx, D1     15
Move W   Bxx, D2     15
MPTY D1,  D2         71
MOVE L   D2,  PAR    20
```

```
                          CPU CYCLE
Move L    PAR1,D1         20
Move L    PAR2,D2         20
ADD  L    D1,  D2         8
MOVE L    D2,  PSUM       20
```

(a) Grain size calculation in M68000 assembly code at 20-MHz cycle

d = T1+T2+T3 +T4+T5+T6

= 20+20+32+20+20+100

= 212 cycles

T3 = 32-bit transmission time at 20 Mbps normalized to M68000 cycle at 20 MHz.

T6 = delay due to software protocols (assume 5 Move instructions, 100)

(b) Calculation of communication delay d

(c) Fine-grain program graph

Calculation of grain size and communication delay for the program graph in Example 2.5. (Courtesy of Kruatrachue and Lewis; reprinted with permission from *IEEE Software*, 1988)

$$
\begin{aligned}
C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\
C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\
C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\
C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \\
\text{Sum} &= C_{11} + C_{12} + C_{21} + C_{22}
\end{aligned}
$$

As shown in diagram(a), the eight multiplications are performed in eight ® nodes, each of which has a grain size of 101 CPU cycles. The remaining seven additions are performed in a 3-level binary tree consisting of seven © nodes. Each additional node requires 8 CPU cycles.

The interprocessor communication latency along all edges in the program graph is eliminated as *d* = 212 cycles by adding all path delays between two communicating processors (diagram (b)).

Program Flow Mechanisms

Conventional computers arc based on a control flow mechanism by which the order of program execution is explicitly stated in the user programs. Dataflow computers are based on a data-driven mechanism which allows the execution of any instruction to be driven by data (operand) availability. Dataflow computers emphasize a high degree of parallelism at the fine-grain instructional level. Reduction computers are based on a demand-driven mechanism which initiates an operation based on the demand for its results by other computations.

Control Flow Versus Data Flow

Conventional von Neumann computers use a *program counter* (PC) to sequence the execution of instructions in a program. The PC is sequenced by instruction flow in a program. This sequential execution style has been called *control-driven*, as program flow is explicitly controlled by programmers.

Control-flow computers use shared memory to hold program instructions and data objects. Variables in the shared memory are updated by many instructions. The execution of one instruction may produce side effects on other instructions since memory is shared. In many cases, the side effects prevent parallel processing from taking place.

In fact, a uniprocessor computer is inherently sequential, due to use of the control- driven mechanism. However, control flow can be made parallel by using parallel language constructs or parallel compilers. Until the data-driven or demand-driven mechanism is proven to be cost-effective, the control-flow approach may continue to dominate the computer industry.

In a *dataflow computer*, the execution of an instruction is driven by data availability instead of being guided by a program counter. In theory, any instruction should be ready for execution whenever operands become available. The instructions in a data-driven program are not ordered in any way. Instead of being stored in a shared memory, data are directly held inside instructions. Computational results (*data tokens)* are passed directly between instructions. Data tokens, once consumed by an instruction, will no longer be available for reuse by other instructions.

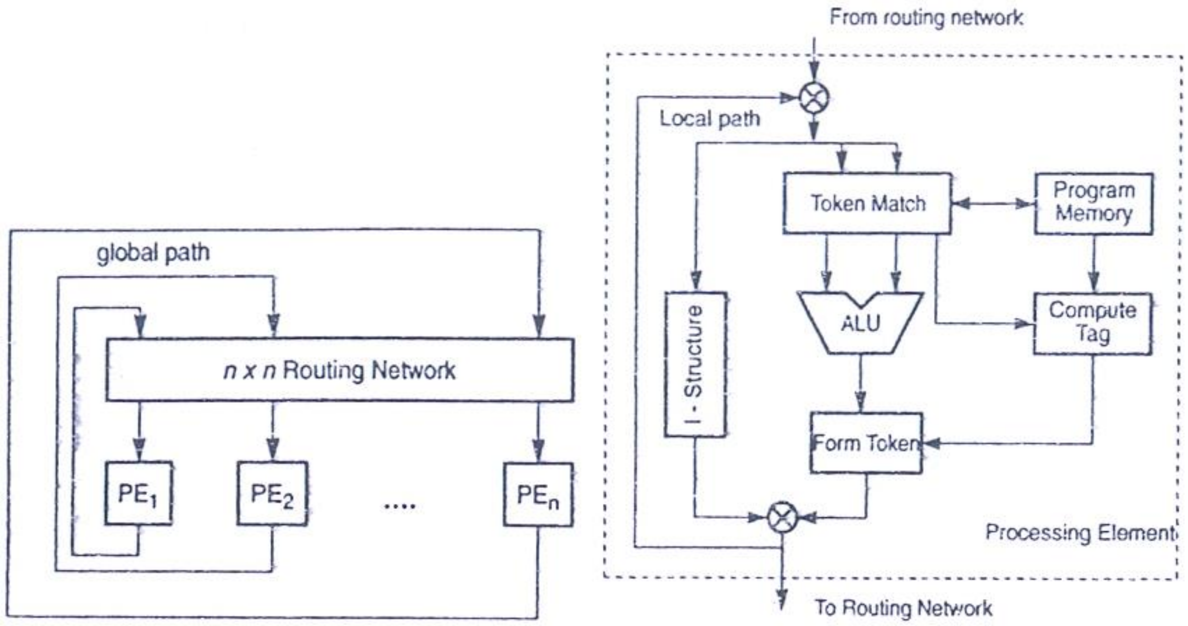
This data-driven scheme requires no shared memory, no program counter, and no control sequencer. However, it requires special mechanisms to detect data availability, to match data tokens with needy instructions, and to enable the chain reaction of asynchronous instruction executions. No memory sharing results in no side effects.

A Dataflow Architecture :There are quite a few experimental dataflow computer projects. Arvind and his associates at MIT have developed a tagged-token architecture for building dataflow computers. As shown in following diagram, the global architecture consists of n processing elements (PEs) interconnected by an n x n routing network. The entire system supports pipelined dataflow operations in all n PEs. Inter-PE communications are done through the pipelined routing network.

Within each PE, the machine provides a low-level *token-matching* mechanism which dispatches only those instructions whose input data (tokens) are already available. Each datum is tagged with the address of the instruction to which it belongs and the context in which the instruction is being executed. Instructions are stored in the program memory. Tagged tokens enter the PE through a local path. The tokens can also be passed to other PEs through the routing network. All internal token circulation operations are pipelined without blocking.

One can think of the instruction address in a dataflow computer as replacing the program counter, and the context identifier replacing the frame base register in a control flow computer. It is the machine's job to match up data with the same tag to needy instructions. In so doing, new data will be produced with a new tag indicating the successor-instruction(s).

Another synchronization mechanism, called the *I-structure,* is provided within each PE. The I-structure is a tagged memory unit for overlapped usage of a data structure by both the producer and consumer processes. Each word of I-structure uses a 2-bit tag indicating whether the word is *empty,* is *full,* or has *pending read* requests. The use of I-structure is a retreat from the pure dataflow approach. The purpose is to reduce excessive copying of large data structures in dataflow operations.



(a) The global architecture

(b) Interior design of a processing element

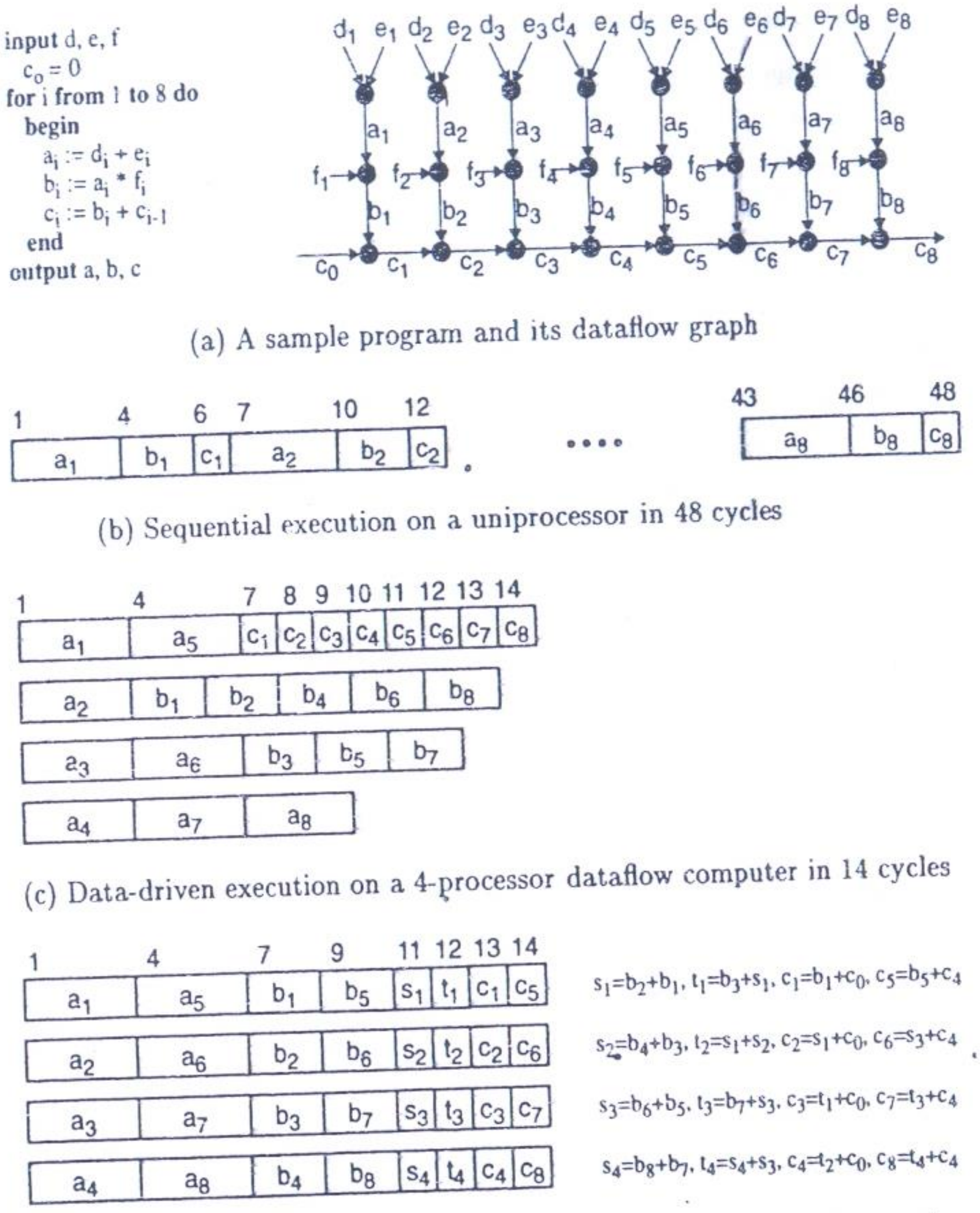
The MIT tagged-token dataflow computer. (Adapted from Arvind and Iannucci, 1986 with permission)

Example : Comparison between data flow and control flow computers

The dataflow graph in diagram(a) shows that 24 instructions are to be executed (8 *divides,* 8 *multiplies,* and 8 *adds). A* dataflow graph is similar to a dependence graph or program graph. The only difference is that data tokens are passed around the edges in a dataflow graph. Assume that each *add, multiply,* and *divide* requires 1, 2, and 3 cycles to complete, respectively. Sequential execution of the 24 instructions on a control flow uniprocessor takes 48 cycles to complete, as shown in diagram (b).

On the other hand, a dataflow multiprocessor completes the execution in 14 cycles in diagram (c). Assume that all the external inputs are available before entering the Do loop. With four processors, instructions a1,a2,a3 and a_4 are all ready for execution in the first three cycles.

The diagram(d) shows the execution of the same set of computations on a conventional multiprocessor using shared memory to hold the intermediate results (si and *ti* for *i* = 1,2,3,4). Note that no shared memory is used in the dataflow implementation. The example does not show any time advantage of dataflow execution over control flow execution.



(a) A sample program and its dataflow graph

(b) Sequential execution on a uniprocessor in 48 cycles

(c) Data-driven execution on a 4-processor dataflow computer in 14 cycles

(d) Parallel execution on a shared-memory 4-processor system in 14 cycles

Comparison between dataflow and control-flow computers. (Adapted from Gajski, Padua, Kuck, and Kuhn, 1982; reprinted with permission from *IEEE Computer*, Feb. 1982)

One advantage of tagging each datum is that data from different contexts can be mixed freely in the instruction execution pipeline. Thus, instruction-level parallelism of dataflow graphs can absorb the communication latency and minimize the losses due to synchronization waits.

Demand-Driven Mechanisms

In a *reduction machine,* the computation is triggered by the demand for an operation's result. Consider the evaluation of a nested arithmetic expression *a* = ((6 + 1) x c- (d + e)). The data-driven computation chooses a bottom-up approach, starting from the innermost operations 6+1 and *d* + e. then proceeding to the x operation, and finally to the outermost operation -. Such a computation has been called *eager evaluation* because operations are carried out immediately after all their operands become available.

A *demand-driven* computation chooses a top-down approach by first demanding the value of *d*, which triggers the demand for evaluating the next-level expressions (6+1) x c and *d-i-e*, which in turn triggers the demand for evaluating 6 +1 at the innermost level. The results are then returned to the nested demander in the reverse order before *a* is evaluated.

A demand-driven computation corresponds to *lazy evaluation,* because operations are executed only when their results' are required by another instruction. The demand- driven approach matches naturally with the functional programming concept. The removal of side effects in functional programming makes programs easier to parallelize. There are two types of reduction machine models, both having a recursive control mechanism as characterized below.

Reduction Machine Models: In a *string reduction* model, each demander gets a separate copy of the expression for its own evaluation. A long string expression is reduced to a single value in a recursive fashion. Each reduction step has an operator followed by an embedded reference to demand the corresponding input operands. The operator is suspended while its input arguments are being evaluated. An expression is said to be fully reduced when all the arguments have been replaced by literal values.

In a graph *reduction* model:, the expression is represented as a directed graph. The graph is reduced by evaluation of branches or subgraphs. Different parts of a graph or subgraphs can be reduced or evaluated in parallel upon demand. Each demander is given a pointer to the result of the reduction. The demander manipulates all references to that graph.

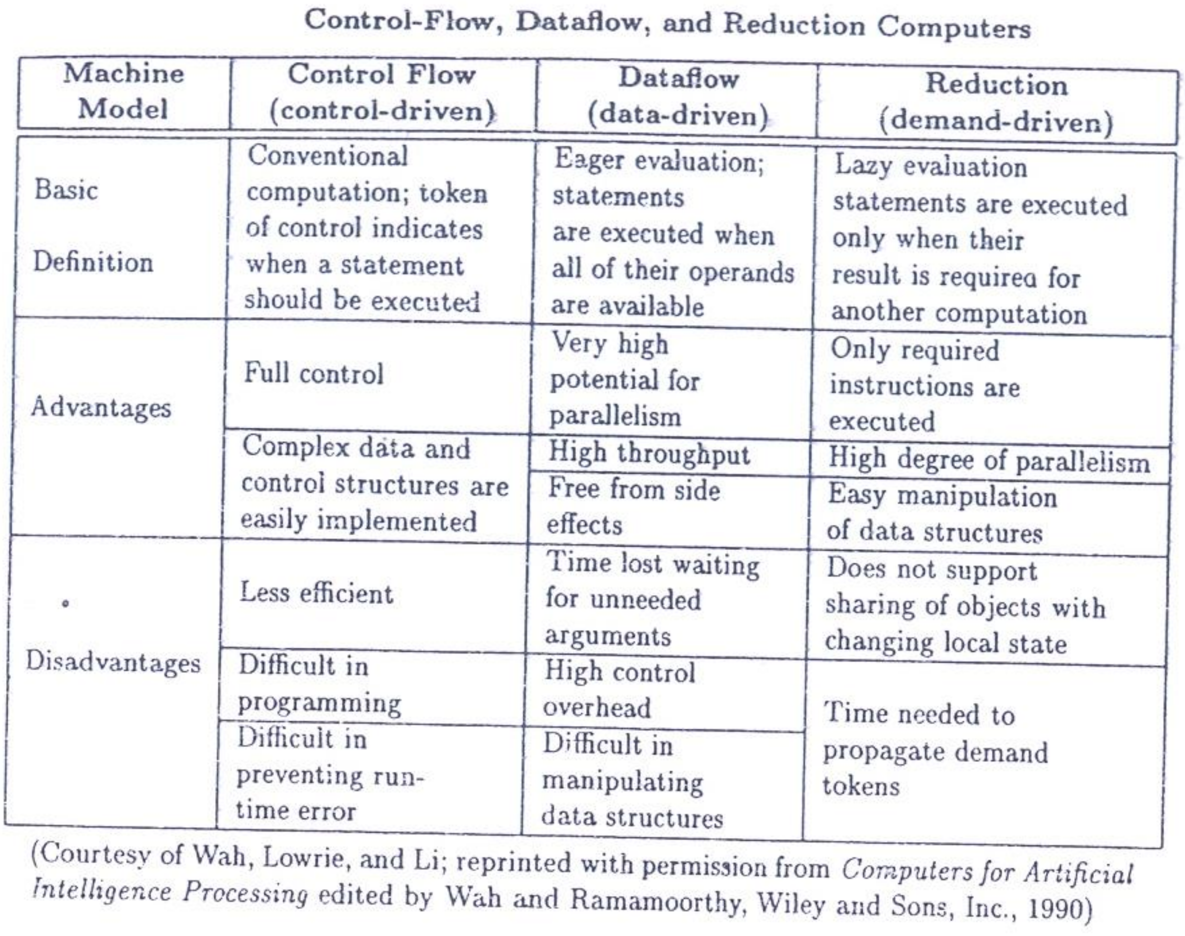
Graph manipulation is based on sharing the arguments using pointers. This traversal of the graph and reversal of the references are continued until constant arguments are encountered.

Comparison of Flow Mechanisms

Control-flow, dataflow, and reduction computer architectures are compared in following Table . The degree of explicit control decreases from control-driven to demand-driven to data-driven. Highlighted in the table are the differences between *eager evaluation* and *lazy evaluation* in data-driven and demand-driven computers, respectively.

Furthermore, control tokens are used in control-flow computers and reduction machines, respectively. The listed advantages and disadvantages of the dataflow and reduction machine models are based on research findings rather than on extensive operational experience.

Even though conventional von Neumann model has many disadvantages, the industry is still building computers following the control-flow model. The choice was based on cost-effectiveness, marketability, and the narrow windows of competition used by the industry. Program flow mechanisms dictate architectural choices. Both dataflow and reduction models, despite a higher potential for parallelism, are still in the research stage. Control-flow machines still dominate the market.



Control-Flow, Dataflow, and Reduction Computers

Machine Model	Control Flow (control-driven)	Dataflow (data-driven)	Reduction (demand-driven)
Basic Definition	Conventional computation; token of control indicates when a statement should be executed	Eager evaluation; statements are executed when all of their operands are available	Lazy evaluation statements are executed only when their result is requirea for another computation
Advantages	Full control	Very high potential for parallelism	Only required instructions are executed
	Complex data and control structures are easily implemented	High throughput	High degree of parallelism
		Free from side effects	Easy manipulation of data structures
Disadvantages	Less efficient	Time lost waiting for unneeded arguments	Does not support sharing of objects with changing local state
	Difficult in programming	High control overhead	Time needed to propagate demand tokens
	Difficult in preventing run-time error	Difficult in manipulating data structures	

(Courtesy of Wah, Lowrie, and Li; reprinted with permission from *Computers for Artificial Intelligence Processing* edited by Wah and Ramamoorthy, Wiley and Sons, Inc., 1990)

System Interconnect Architectures

Static and dynamic networks for interconnecting computer subsystems or for constructing multiprocessors or multicomputers are introduced below. We study first the distinctions between direct networks for static connections and indirect networks for dynamic connections. These networks can be used for internal connections among processors, memory modules, and I/O disk arrays in a centralized system, or for distributed networking of multicomputer nodes.

Various topologies for building networks are specified below. These include latency analysis, bisection bandwidth, and data-routing functions. Finally, we analyze the scalability of parallel architecture in solving scaled problems.

The communication efficiency of the underlying network is critical to the performance of a parallel computer. What we hope to achieve is a low-latency network with a high data transfer rate and thus a wide communication bandwidth. These network properties help make design choices for machine architecture.

Network Properties and Routing

The topology of an interconnection network can be either static or dynamic. *Static networks* are formed of point-to-point direct connections which will not change during program execution. *Dynamic networks* are implemented with switched channels, which are dynamically configured to match the communication demand in user programs.

Static networks are used for fixed connections among subsystems of a centralized system or multiple computing nodes of a distributed system. Dynamic networks include buses, crossbar switches, and multistage networks, which are often used in shared- memory multiprocessors. Both types of networks have also been implemented for inter- PE data routing in S1MD computers.

Before we analyze various network topologies, let us define several parameters often used to estimate the complexity, communication efficiency, and cost of a network. In genera), a network is represented by the graph of a finite number of nodes linked by directed or undirected edges. The number of nodes in the graph is called the *network size.*

Node Degree and Network Diameter : The number of edges (links or channels) incident on a node is called the *node degree d.* In the case of unidirectional channels, the number of channels into a node is the *in degree,* and that out of a node is the *out degree.* Then the node degree is the sum of the two. The node degree reflects the number of I/O ports required per node, and thus the cost of a node. Therefore, the node degree should be kept a constant, as small as possible in order to reduce cost. A constant node degree is very much desired to achieve modularity in building blocks for scalable systems.

The *diameter D* of a network is the maximum shortest path between any two nodes. The path length is measured by the number of links traversed. The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a figure of communication merit for the network. Therefore, the network diameter should be as small as possible from a communication point of view.

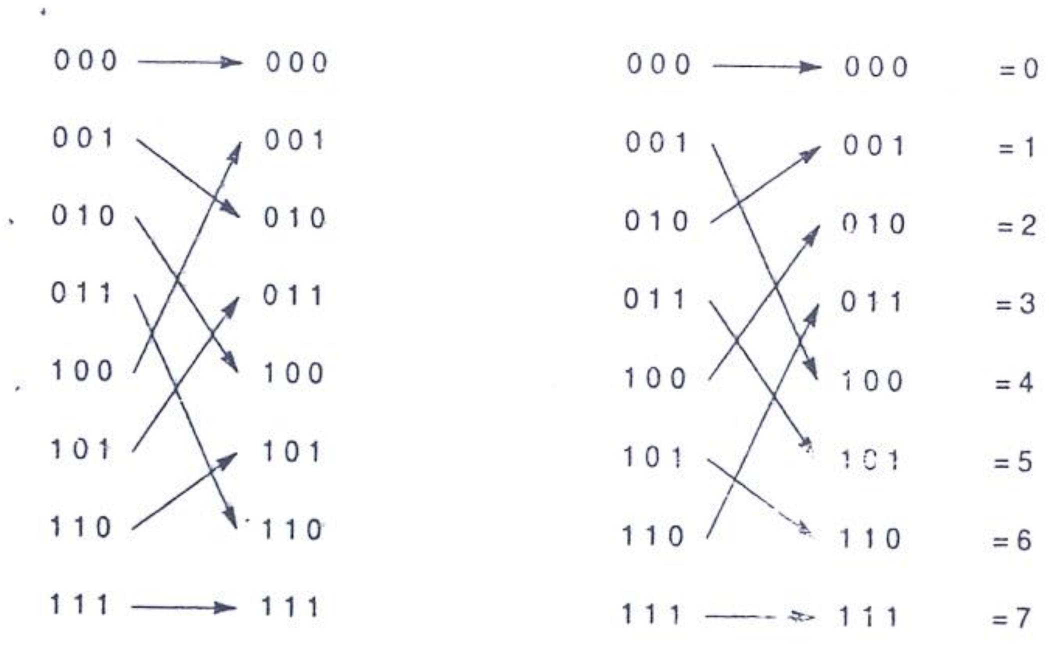
Bisection Width :When a given network is cut into two equal halves, the minimum number of edges (channels) along the cut is called the *channel bisection width b.* In the case of a communication network, each edge corresponds to a *channel* with *w* bit wires. Then the *wire bisection width* is *B* = *bw.* This parameter *B* reflects the wiring density of a network. When *B* is fixed, the *channel width* (in bits) *w* = *B/b.* Thus the bisection width provides a good indicator of the maximum communication bandwidth along the bisection of a network. All other cross sections should be bounded by the bisection width.
Another quantitative parameter is the *wire length* (or channel length) between nodes. This may affect the signal latency, clock skewing, or power requirements. We label a network *symmetric* if the topology is the same looking from any node. Symmetric networks are easier to implement or to program. Whether the nodes are homogeneous, the channels are buffered, or some of the nodes are switches are other useful properties for characterizing the structure of a network.

Data-Routing Functions: A data-routing network is used for inter-PE data exchange. This routing network can be static, such as the hypercube routing network used in the TMC/CM-2, or dynamic such as the multistage network used in the IBM GF11. In the case of a multicomputer network, the data routing is achieved through message passing. Hardware routers are used to route messages among multiple computer nodes.
We specify below some primitive data-routing functions implementable on an inter- PE routing network. The versatility of a routing network will reduce the time needed for data exchange and thus can significantly improve the system performance.

Commonly seen data-routing functions among the PEs include *shifting, rotation, permutation* (one-to-one), *broadcast* (one-to-all), *multicast* (many-to-many), *personalized communication* (one-to-many), *shuffle, exchange,* etc. These routing functions can be implemented on ring, mesh, hypercube, or multistage networks.

Permutations: For n objects, there are n! permutations by which the n objects can be reordered. The set *of* all permutations form a *permutation group* with respect to a composition operation. One can use cycle notation to specify a permutation function.
For example, the permutation Π = (a, *b, c){d, e)* stands for the bijection mapping: *a* → 6,b → c, c → *a,d* → e, and e → *d* in a circular fashion. The cycle (a, b, c) has a period of 3, and the cycle *(d, e)* a period of 2. Combining the two cycles, the permutation *n* has a period of 2 x 3 = 6. If one applies the permutation six times, the identity mapping I = (a), (b), (c), (d), (e) is obtained.

Perfect Shuffle and Exchange: Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in following diagram (a) . Its inverse is shown on the right hand side as in diagram (b) :



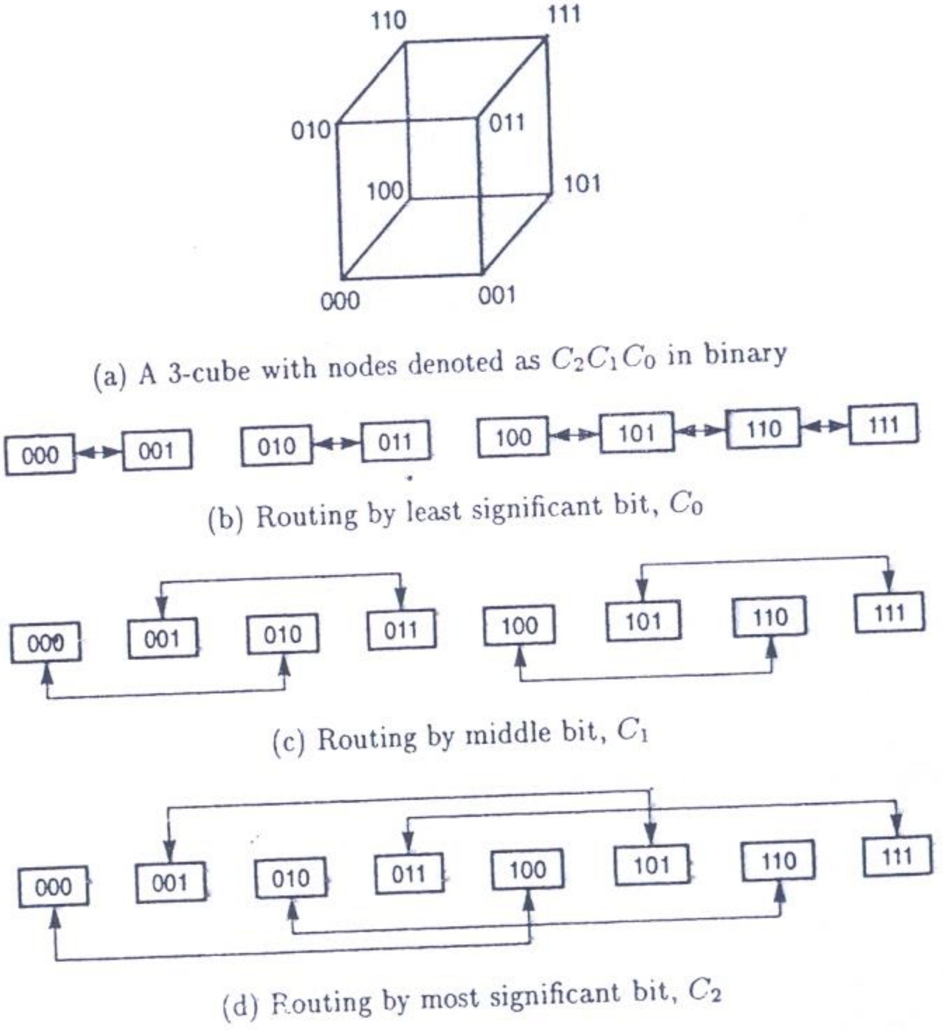
(a) Perfect shuffle

(b) Inverse perfect shuffle

Perfect shuffle and its inverse mapping over eight objects. (Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers*, 1971)

In general, to shuffle $n = 2^k$ objects evenly, one can express each object in the domain by a *k*-bit binary number $x = (x_{k-1},\ldots.x_1,x_0)$

Hvpercube Routing Functions :A three-dimensional binary cube network is shown in the following diagram. Three routing functions are defined by three bits in the node address. For example, one can exchange the data between adjacent nodes which differ in the least significant bit C0, as shown in Fig. (b).
Similarly, two other routing patterns can be obtained by checking the middle bit C1 (diagram (c)) and the most significant bit *C2* (diagram(d)), respectively. In general, an n-dimensional hypercube has n routing functions, defined by each bit of the n-bit address. These data exchange functions can be used in routing messages in a hypercube multicomputer.



(a) A 3-cube with nodes denoted as $C_2C_1C_0$ in binary

(b) Routing by least significant bit, C_0

(c) Routing by middle bit, C_1

(d) Routing by most significant bit, C_2

Three routing functions defined by a binary 3-cube.

Broadcast and Multicast : *Broadcast* is a one-to-all mapping. This can be easily achieved in an SIMD computer using a broadcast bus extending from the array controller to all PEs. A message-passing multicomputer also has mechanisms to broadcast messages. *Multicast* corresponds to a mapping from one subset to another (many to many).

Personalized broadcast sends personalized messages to only selected receivers. Broadcast is often treated as a global operation in a multicomputer. Personalized broadcast may have to be implemented with matching of destination codes in the network.

Network Performance To summarize the above discussions, the performance of an interconnection network is affected by the following factors:

(i) *Functionality* — This refers to how the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence.

(ii) *Network latency* — This refers to the worst-case time delay for a unit message to be transferred through the network.

(iii) *Bandwidth* — This refers to the maximum data transfer rate, in terms of Mbytes/s transmitted through the network.

(iv) *Hardware complexity* — This refers to implementation costs such as those for wires, switches, connectors, arbitration, and interface logic.

(v) *Scalability* — This refers to the ability of a network to be modularly expandable with a scalable performance with increasing machine resources.