advanced computer architecture mod1

advanced computer architecture (CMR Technical Campus)

An Autonomous Institution Affiliated to VTU,
Belagavi Approved by AICTE,
New Delhi
Recognized by UGC under 2(f)&12(B)
Accredited by NBA & NAAC

ADVANCED COMPUTER ARCHITECTURE (17CS72)

MODULE-1

Chapter-1 Parallel Computer Models

The State of Computing

Computer Development Milestones

- Computers have gone through two major stages of development: mechanical and electronic. Prior to 1945, computers were made with mechanical or electromechanical parts.
- The earliest mechanical computer can be traced back to 500 BC in the form of the abacus used in China.
- The abacus is manually operated to perform decimal arithmetic with carry propagation digit by digit.
- Blaise Pascal built a mechanical adder/subtractor in Prance in 1642. Charles Babbage designed a difference engine in England for polynomial evaluation in 1827.
- Konrad Zuse built the first binary mechanical computer in Germany in 1941. Howard Aiken proposed the very first electromechanical decimal computer, which was built as the Harvard Mark I by IBM in 1944.
- Both Zuse's and Aiken's machines were designed for general-purpose computations.

Computer Generations

- Over the past five decades, electronic computers have gone through five generations of development. Each of the first three generations lasted about 10 years.
- The fourth generation covered a time span of 15 years.
- We have just entered the fifth generation with the use of processors and memory devices with more than 1 million transistors on a single silicon chip.

Table 1-1 Five **Generations** of Electronic Computers

Generation	Technology and Architecture	Software and Applications	Representative Systems
First (1945-54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator, fixed-point arithmetic.	Machine/assembly languages, single user, no subroutine linkage, programmed I/O using CPU.	ENIAC, Princeton IAS, IBM 701.
Second (1955–64)	Discrete transistors and core memories, floating-point arithmetic, I/O processors, multiplexed memory access.	HLL used with compilers, subroutine libraries, batch processing monitor.	IBM 7090, CDC 1604, Univac LARC.
Third (1965-74)	Integrated circuits (SSI/-MSI), microprogramming, pipelining, cache, and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, PDP-8.
Fourth (1975-90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multicomputers.	Multiprocessor OS, languages, compilers, and environments for parallel processing.	VAX 9000, Cray X-MP, IBM 3090, BBN TC2000.
Fifth (1991-present)	ULSI/VHSIC processors, memory, and switches, high-density packaging, scalable architectures.	Massively parallel processing, grand challenge applications, heterogeneous processing.	Fujitsu VPP500, Cray/MPP, TMC/CM-5, Intel Paragon.

Elements of Modern Computers

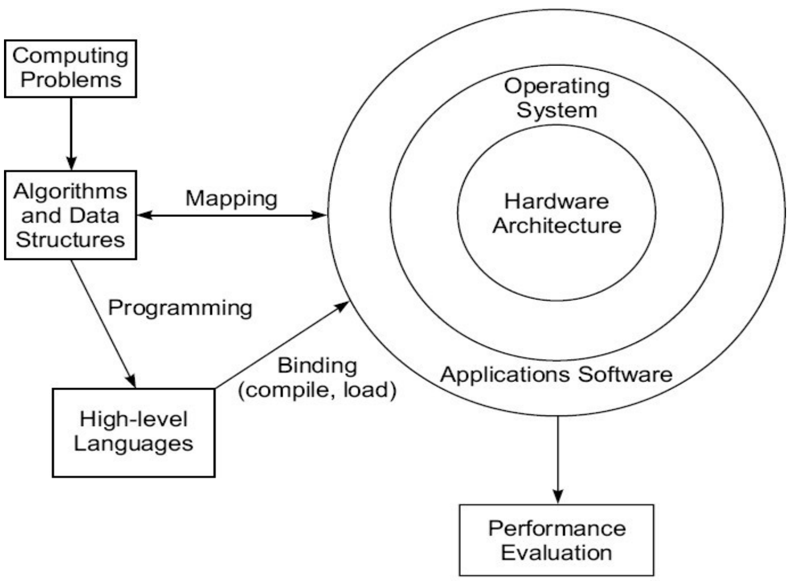


Fig. 1.1 Elements of a modern computer system

- Computing Problems

- The use of a computer is driven by real-life problems demanding fast and accurate solutions. Depending on the nature of the problems, the solutions may require different computing resources.
- For numerical problems in science and technology, the solutions demand complex mathematical formulations and tedious integer or floating-point computations.
- For alpha numerical problems in business and government, the solutions demand accurate transactions, large database management, and information retrieval operations.
- For artificial intelligence (AI) problems, the solutions demand logic inferences and symbolic manipulations.
- These computing problems have been labeled *numerical computing, transaction processing,* and *logical reasoning.*
- Some complex problems may demand a combination of these processing modes.

- Hardware Resources

- A modern computer system demonstrates its power through coordinated efforts by hardware resources, an operating system, and **application** software.
- Processors, memory, and peripheral devices form the hardware core of a computer system.
- Special hardware interfaces are often built into I/O devices, such as terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, file servers, voice data entry, printers, and plotters.
- These peripherals are connected to mainframe computers directly or through local or wide-area networks.

- Operating System

- An effective operating system manages the allocation and deallocation of resources during the execution of user programs.
- Beyond the OS, application software must be developed to benefit the users.
- Standard benchmark programs are needed for performance evaluation.
- Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa.
- Efficient mapping will benefit the programmer and produce better source codes.

- The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc.
- These activities are usually architecture-dependent.

- System Software Support

- Software support is needed for the development of efficient programs in high-level languages. The source code written in a HLL must be first translated into object code by an optimizing compiler.
- The *compiler* assigns variables to registers or to memory words and reserves functional units for operators.
- An *assembler* is used to translate the compiled object code into machine code which can be recognized by the machine hardware. A *loader* is used to initiate the program execution through the OS kernel.

- Compiler Support

There are three compiler upgrade approaches:

- **Preprocessor:** A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs.
- **Precompiler:** The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection.
- **Parallelizing Compiler:** This approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs.

Evolution of Computer Architecture

- The study of computer architecture involves both hardware organization and programming/software requirements.
- As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.
- From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc.

- Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

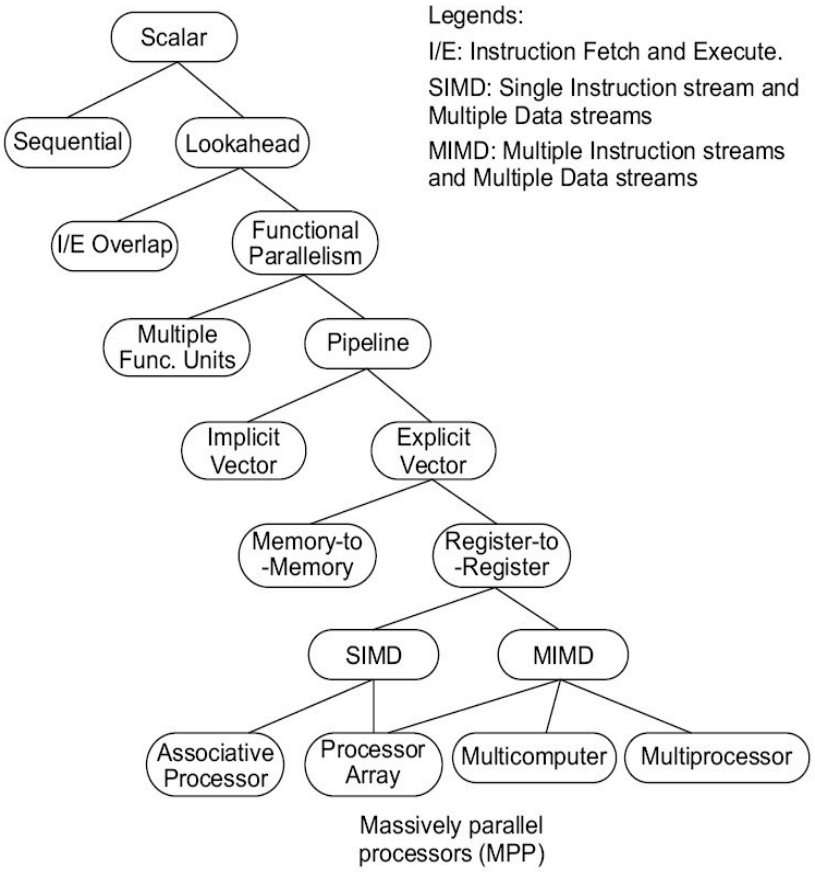


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

Lookahead, Parallelism, and Pipelining

Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches:

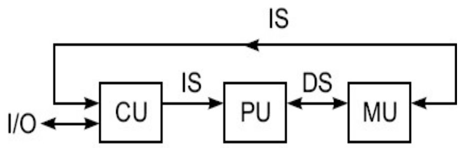
1. using multiple functional units simultaneously,
2. to practice pipelining at various processing levels.

Flynn's Classification

Michael Flynn (1972) introduced a classification of various computer architectures based on notions of instruction and data streams.

1. **SISD** (Single Instruction stream over a Single Data stream) computers
2. **SIMD** (Single Instruction stream over Multiple Data streams) machines
3. **MIMD** (Multiple Instruction streams over Multiple Data streams) machines.
4. **MISD** (Multiple Instruction streams and a Single Data stream) machines

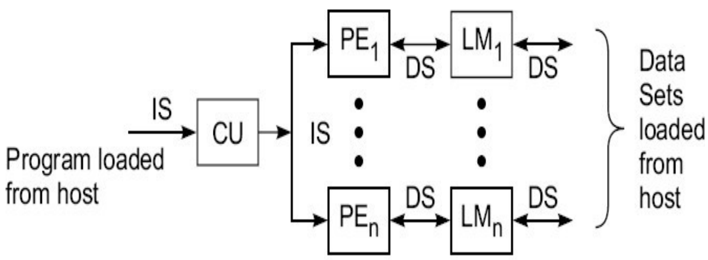
1. SISD (Single Instruction stream over a Single Data stream) computers



(a) SISD uniprocessor architecture

- Conventional sequential machiunes are called SISD computers.
- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes

2. SIMD (Single Instruction stream over Multiple Data streams) machines

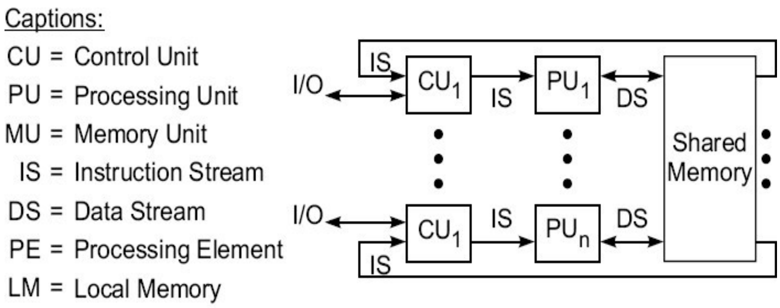


(b) SIMD architecture (with distributed memory)

A type of parallel computer

- **Single instruction**: All processing units execute the same instruction issued by the control unit at any given clock cycle.
- **Multiple data:** Each processing unit can operate on a different data element. The processors are connected to shared memory or interconnection network providing multiple data to processing unit.
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Thus single instruction is executed by different processing unit on different set of data.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.
- Two varieties: Processor Arrays e.g., Connection Machine CM-2, Maspar MP-1, MP-2 and Vector Pipelines processor e.g., IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

3. MIMD (Multiple Instruction streams over Multiple Data streams) machines.

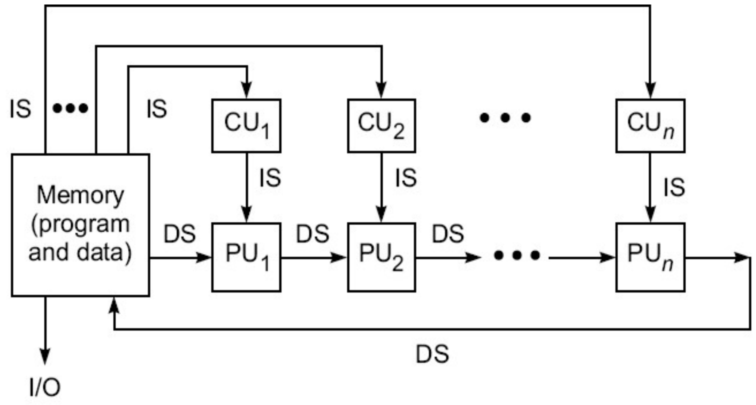


(c) MIMD architecture (with shared memory)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- A single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.
- Thus in these computers same data flow through a linear array of processors executing different instruction streams.
- This architecture is also known as **Systolic Arrays** for pipelined execution of specific instructions. Some conceivable uses might be:

  1. multiple frequency filters operating on a single signal stream
  2. multiple cryptography algorithms attempting to crack a single coded message.

4. MISD (Multiple Instruction streams and a Single Data stream) machines



(d) MISD architecture (the systolic array)

Fig. 1.3 Flynn's classification of computer architectures (Derived from Michael Flynn,

- **Multiple Instructions:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream, multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- There are multiple processors each processing different tasks.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

Development Layers

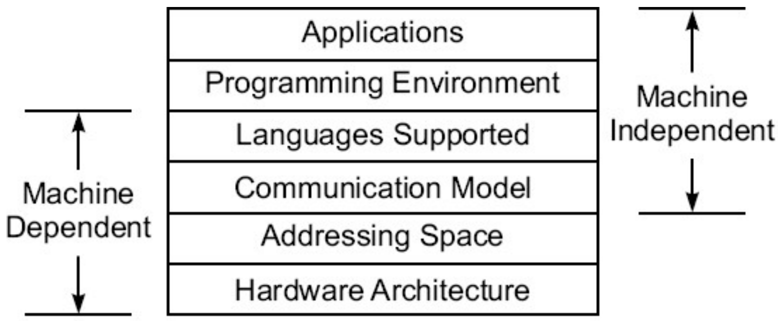


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

Development Layers A layered development of parallel computers is illustrated in Fig. 1.4.

- **Hardware configurations** differ from machine to machine, even those of the same model.
- **Address Space** of a processor in a computer system varies among different architectures. It depends on the memory organization, which is machine-dependent. These features are up to the designer and should match the target application domains.
- We want to develop **Application Programs** and **Programming Environments** which are machine-independent. Independent of machine architecture, the user programs can be ported to many computers with minimum conversion costs.
- **High-level languages** and **Communication Models** depend on the architectural choices made in a computer system. From a programmer's viewpoint, these two layers should be architecture-transparent.
- At present, Fortran, C, Pascal, Ada, and Lisp are supported by most computers.
- However, the **Communication Models**, shared variables versus message passing, are mostly machine-dependent. The Linda approach using *tuple spaces* offers architecture transparent Communication model for parallel computers.
- Application programmers prefer more architectural transparency. However, kernel programmers have to explore the opportunities supported by hardware.
- As a good computer architect, one has to approach the problem from both ends.
- The compilers and OS support should be designed to remove as many architectural constraints as possible from the programmer.

System Attributes affecting Performance

Clock Rate and CPI

- The CPU (or simply the *processor)* of today's digital computer is driven by a clock with a constant *cycle time* (τ in nanoseconds).
- The inverse of the cycle time is the *clock rate* (/ = 1/ τ in megahertz). The size of a program is determined by its *instruction count* (Ic), in terms of the number of machine instructions to be executed in the program.
- Different machine instructions may require different numbers of clock cycles to execute. Therefore, the *cycles* per *instruction* (CPI) becomes an important parameter for measuring the time needed to execute each instruction.
- For a given instruction set, we can calculate an *average* CPI over all instruction types, provided we know their frequencies of appearance in the program.

- An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time.
- Unless specifically focusing on a single instruction type, we simply use the term CPI to mean the average value with respect to a given instruction set and a given program mix.

Performance Factors

- Let Ic be the number of instructions in a given program, or the instruction count.
- The CPU time *(T* in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$T = Ic \times CPI \times \tau \qquad (1.1)$$

- The execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand(s) fetch, execution, and store results.
- In this cycle, only the instruction decode and execution phases are carried out in the CPU.
- The remaining three operations may be required to access the memory. We define a *memory cycle* as the time needed to complete one memory reference.
- Usually, a memory cycle is *k* times the processor cycle τ.
- The value of *k* depends on the speed of the memory technology and processor-memory interconnection scheme used.
- The CPI of an instruction type can be divided into two component terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction.
- Depending on the instruction type, the complete instruction cycle may involve one to four memory references (one for instruction fetch, two for operand fetch, and one for store results). Therefore we can rewrite Eq. 1.1 as follows;

$$T = I_c \times (p + m \times k) \times \tau \qquad (1.2)$$

where p is the number of processor cycles needed for the instruction decode and execution,

m is the number of memory references needed,

k is the ratio between memory cycle and processor cycle,

I_c is the instruction count,

r is the processor cycle time.

Equation 1.2 can be further refined once the CPi components *(p,m,k)* are weighted over the entire instruction set.

System Attributes

- The above five performance factors (I_c, p, *m*, *k*, τ) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, and cache and memory hierarchy, as specified in Table 1.2.
- The instruction-set architecture affects the program length (Ic) and processor cycle needed (p). The compiler technology affects the values of I_c, p and the memory reference count (m).
- The CPU implementation and control determine the total processor time *(p. τ)* needed.
- Finally, the memory technology and hierarchy design affect the memory access latency *(k.* τ). The above CPU time can be used as a basis in estimating the execution rate of a processor.

Table 1.2 Performance Factors Versus System Attributes

System Attributes	Performance Factors				
	Instr. Count,	Average Cycles per Instruction, CPI			Processor Cycle Time, τ
		Processor Cycles per Instruction, p	Memory References per Instruction, *m*	Memory-Access Latency, *k*	
Instruction-set Architecture	X	X			
Compiler Technology	X	X	X		
Processor Implementation and Control		X			X
Cache and Memory Hierarchy				X	X

MIPS Rate

- Let *C* be the total number of clock cycles needed to execute a given program.
- Then the CPU time in Eq. 1.2 can be estimated as $T = C \times \tau = C/f$.
- Furthermore, CPI = C/I_c and $T = I_c \times \text{CPI} \times \tau = I_c \times \text{CPI}/f$. The processor speed is often measured in terms of *million instructions per second* (MIPS).
- We simply call it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors, including the clock rate (f), the instruction count (I_c), and the CPI of a given machine, as defined below:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6} = \frac{f \times I_c}{C \times 10^6} \qquad (1.3)$$

- Based on Eq. 1.3, the CPU time in Eq. 1.2 can also be written as $\mathbf{T = I_c \times 10^{-6} / MIPS}$.
- Based on the system attributes identified in Table 1.2 and the above derived expressions, we conclude by indicating the fact that the MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI.
- All four system attributes, instruction set, compiler, processor, and memory technologies, affect the MIPS rate, which varies also from program to program.

Throughput Rate

- Number of programs a system can execute per unit time, called the *system throughput Ws* (in programs/second).
- In a multiprogrammed system, the system throughput is often lower than the *CPU throughput Wp* defined by:

$$W_p = \frac{f}{I_c \times \text{CPI}} \qquad (1.4)$$

- Note that W_p = (MIPS) X $10^6/I_c$ from Eq. 1.3- The unit for W_p is programs/second.
- The CPU throughput is a measure of how many programs can be executed per second, based on the MIPS rate and average program length (I_c).
- The reason why $Ws < Wp$ is due to the additional system overheads caused by the I/O, compiler, and OS when multiple programs are interleaved for CPU execution by multiprogramming or timesharing operations.
- If the CPU is kept busy in a perfect program-interleaving fashion, then $Ws = Wp$. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Programming Environments

- Programmability depends on the programming environment provided to the users.
- Conventional computers are used in a sequential programming environment with tools developed for a uniprocessor computer.
- Parallel computers need parallel tools that allow specification or easy detection of parallelism and operating systems that can perform parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

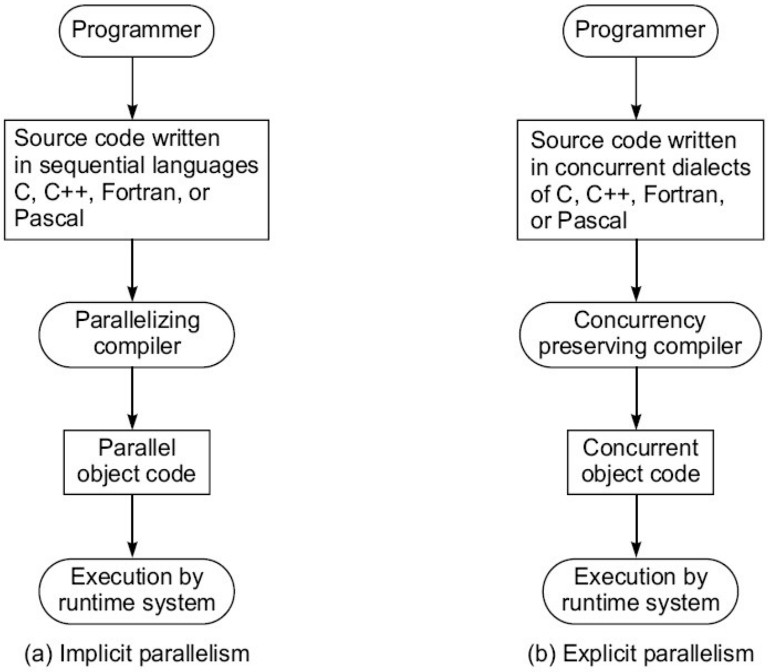


Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from

Implicit Parallelism

- An implicit approach uses a conventional language, such as C, Fortran, Lisp, or Pascal, to write the source program.
- The sequentially coded source program is translated into parallel object code by a parallelizing compiler.
- The compiler must be able to detect parallelism and assign target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.
- With parallelism being implicit, success relies heavily on the "intelligence" of a parallelizing compiler.
- This approach requires less effort on the part of the programmer.

Explicit Parallelism

- The second approach (Fig. 1.5b) requires more effort by the programmer to develop a source program **using parallel** dialects of C, Fortran, Lisp, or Pascal.
- Parallelism is explicitly specified in the user programs.
- This will significantly reduce the burden on the compiler to detect parallelism.
- Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources.

Multiprocessors and Multicomputers

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories.

1. Shared-Memory Multiprocessors

There are 3 shared-memory multiprocessor models:

i. Uniform Memory-access (UMA) model,
ii. Non uniform-Memory-access (NUMA) model
iii. Cache-Only Memory Architecture (COMA) model.

These models differ in how the memory and peripheral resources are **shared** or distributed.

i. Uniform Memory-Access (UMA) model

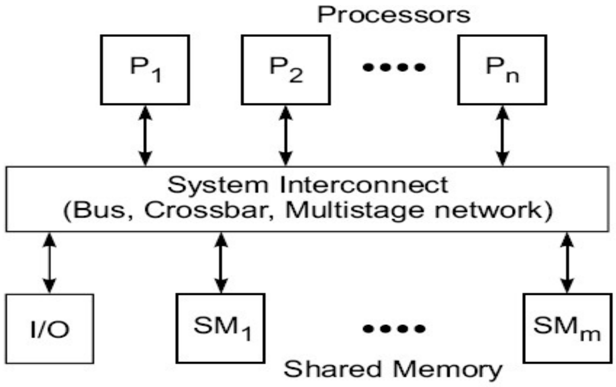


Fig. 1.6 The UMA multiprocessor model

- In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors.
- All processors have equal access time to all memory words, which is why it is called uniform memory access.
- Each processor may use a private cache. Peripherals are also shared in some fashion.
- Multiprocessors are called *tightly coupled systems* dun to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch, or a multistage network.
- Most computer manufacturers have *multiprocessor* (MP) extensions of their *uniprocessor*
- (UP) product line.
- The UMA model is suitable for general-purpose and timesharing applications by **multiple** users. It can be used to speed up the execution of a single large program in time-critical applications. To

coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory.

- When all processors have equal access to all peripheral devices, the system is called a **symmetric multiprocessor.** In this case, all the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.

ii. Non uniform-Memory-Access (NUMA) model

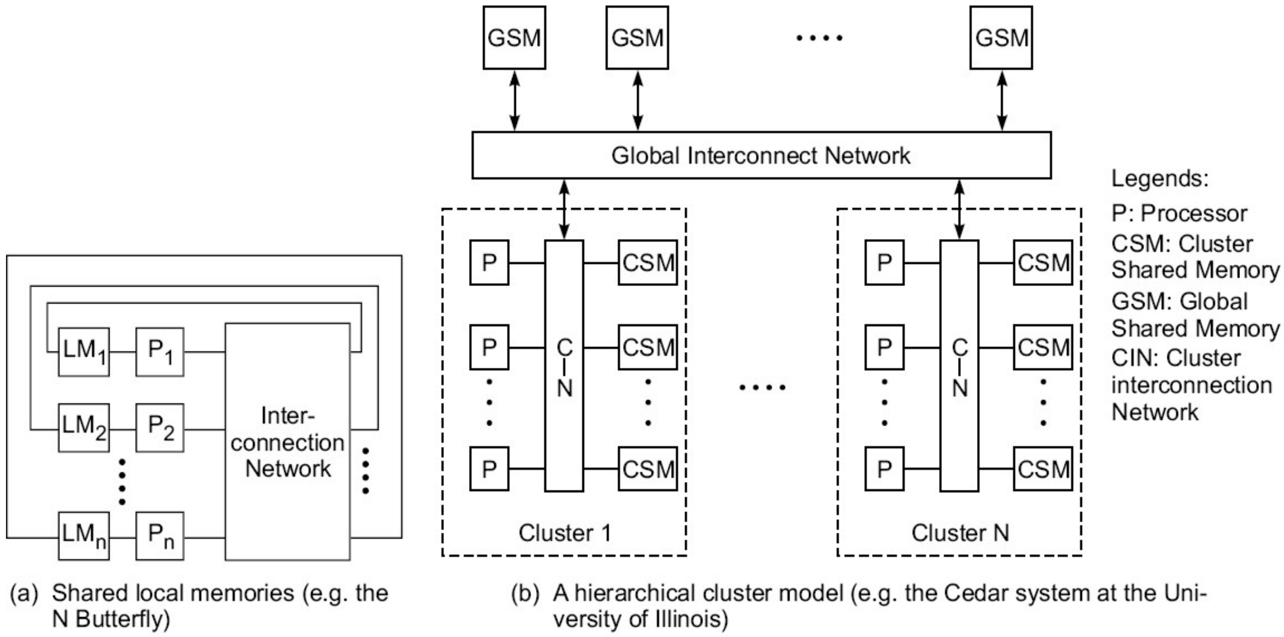


Fig. 1.7 Two NUMA models for multiprocessor systems

- A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word.
- Two NUMA machine models are depicted in Fig. 1.7.
- The shared memory is physically distributed to all processors, called *local memories.*
- The collection of all local memories forms a global address space accessible by all processors.
- It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network.
- The BBN TC-2000 Butterfly multiprocessor assumes the configuration shown in Fig. 1.7a.

iii. Cache-Only Memory Architecture (COMA) model

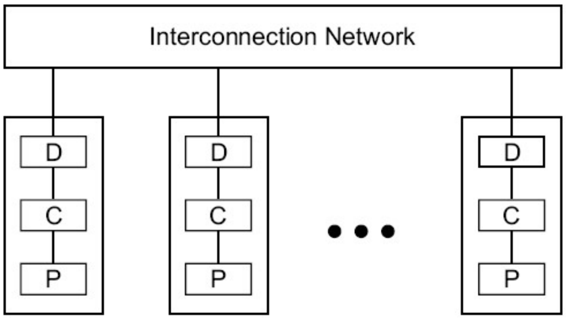


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

- A multiprocessor using cache-only memory assumes the COMA model.
- Examples of COMA machines include the Swedish Institute of Computer Science's Data Diffusion Machine and Kendall Square Research's KSR-1 machine.
- The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches.
- There is no memory hierarchy at each processor node. All the caches form a global address space.
- Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8).
- Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks.
- Initial data placement is not critical because data will eventually migrate to where it will be used.

2. Distributed-Memory Multicomputers

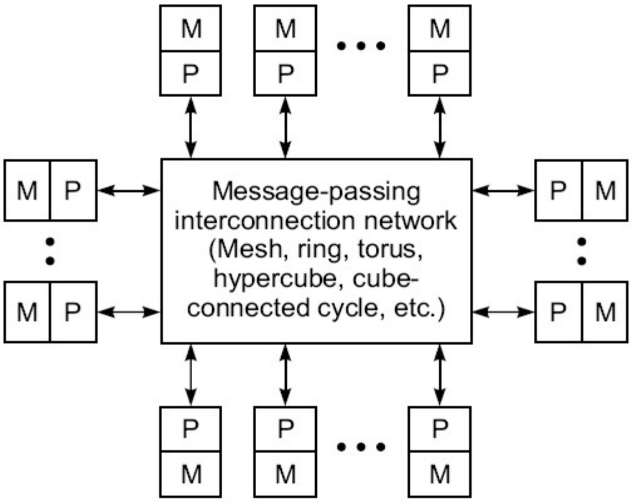


Fig. 1.9 Generic model of a message-passing multicomputer

- A distributed-memory multicomputer system is modeled in the above figure consists of multiple computers, often called *nodes,* interconnected by a message-passing network.
- Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.
- The message-passing network provides point-to-point static connections among the nodes.
- All local memories are private and are accessible only by local processors.
- For this reason, traditional multicomputers have been called *no-remote-memory-access* (NORMA) machines.
- However, this restriction will gradually be removed in future multi computers with distributed shared memories. Internode communication is carried out by passing messages through the static connection network.

Multivector and SIMD Computers

We can classify super computers as:

i. **Pipelined vector machines** using a few powerful processors equipped with vector hardware

ii. **SIMD computers** emphasizing massive data parallelism

Vector Supercomputers

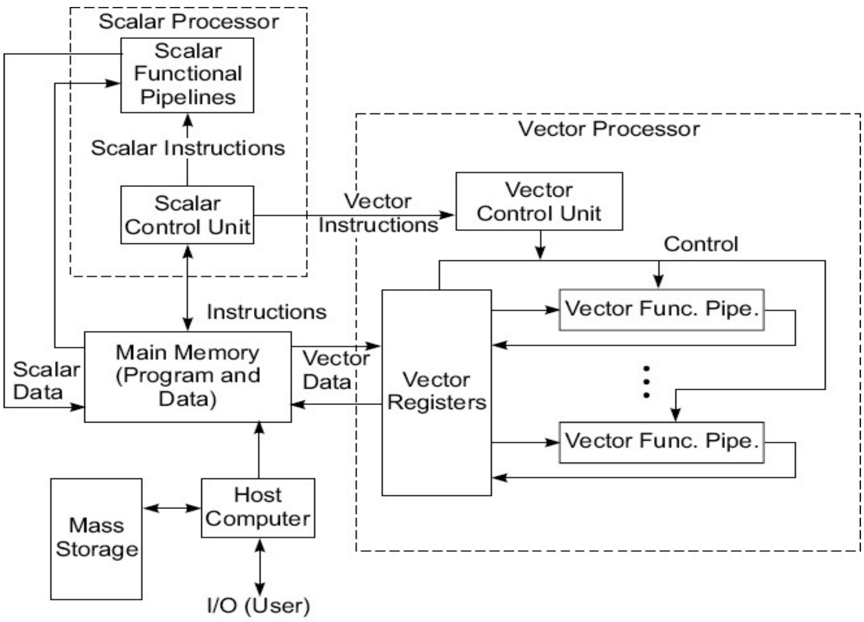


Fig. 1.11 The architecture of a vector supercomputer

- A vector computer is often built on top of a scalar processor.
- As shown in Fig. 1.11, the vector processor is attached to the scalar processor as an optional feature.
- Program and data are first loaded into the main memory through a host computer.
- All instructions are first decoded by the scalar control unit.
- If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.
- If the instruction is decoded as a vector operation, it will be sent to the vector control unit.
- This control unit will supervise the flow of vector data between the main memory and vector functional pipelines.
- The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor.

Vector Processor Models

- Figure l.ll shows a **register-to-register** architecture.
- Vector registers are used to hold the vector operands, intermediate and final vector results.
- The vector functional pipelines retrieve operands from and put results into the vector registers.
- All vector registers are programmable in user instructions.
- Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.
- The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register in a Cray Series supercomputer.
- Other machines, like the Fujitsu VP2000 Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.

SIMD Supercomputers

SIMD computers have a single instruction stream over multiple data streams.

An operational model of an SIMD computer is specified by a 5-tuple:

M = (N,C, I,M, R)

where

1. *N* is the number of *processing elements* (PEs) in the machine. For example, the Illiac IV had 64 PEs and the Connection Machine CM-2 had 65,536 PEs.

2. ***C*** is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.
3. **I** s the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.
4. **M** is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

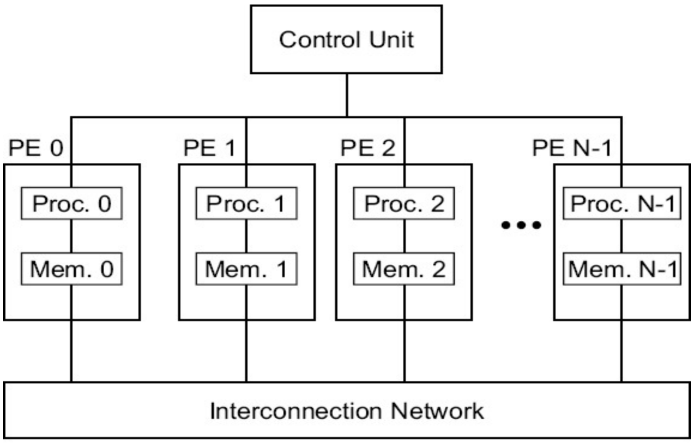


Fig. 1.12 Operational model of SIMD computers

PRAM AND VLSI MODELS

PRAM model (Parallel Random Access Machine)

- PRAM is a theoretical model of parallel computation in which an arbitrary but finite number of processors can access any value in an arbitrarily large shared memory in a single time step.
- Processors may execute different instruction streams, but work synchronously. This model assumes a shared memory, multiprocessor machine as shown:
- The machine size n can be arbitrarily large
- The machine is synchronous at the instruction level. That is, each processor is executing it's own series of instructions, and the entire machine operates at a basic time step (cycle). Within each cycle, each processor executes exactly one operation or does nothing, i.e. it is idle.

- An instruction can be any random access machine instruction, such as: fetch some operands from memory, perform an ALU operation on the data, and store the result back in memory.
- All processors implicitly synchronize on each cycle and the synchronization overhead is assumed to be zero.
- Communication is done through reading and writing of shared variables.
- Memory access can be specified to be UMA, NUMA, EREW, CREW, or CRCW with a defined conflict policy.
- The PRAM model can apply to SIMD class machines if all processors execute identical instructions on the same cycle or to MIMD class machines if the processors are executing different instructions.
- Load imbalance is the only form of overhead in the PRAM model.

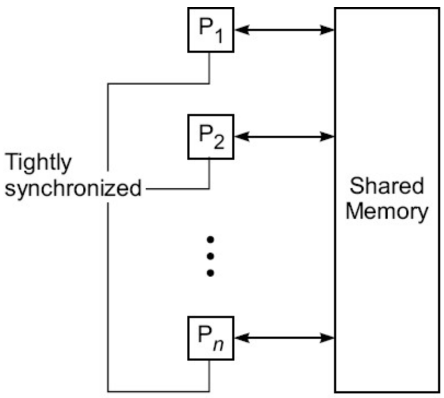


Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all *n* processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

An n-processor PRAM (Fig. 1.14) has a globally addressable memory.

The shared memory can be distributed among the processors or centralized in one place. The n processors operate on a synchronized read-memory, compute, and write-memory cycle. With shared memory, the model must specify how concurrent read and concurrent write of memory are handled.

Four memory-update options are possible:

- **Exclusive Read (ER)** — This allows at mast one processor to read from any memory location in each cycle, a rather restrictive policy.
- **Exclusive Write (EW)** — This allows at most one processor to write into a memory location at a time.
- **Concurrent Read (CR)** — This allows multiple processors to read the same information from the same memory cell in the same cycle.

- **Concurrent Write (CW)** — This allows simultaneous writes to the same memory location. In order to avoid confusion, some policy must be set up to resolve the write conflicts. Various combinations of the above options lead to several variants of the PRAM model as specified below.

PRAM Variants

There are 4 variants of the PRAM model, depending on how the memory reads and writes are handled.

- **EREW – PRAM Model (Exclusive Read, Exclusive Write):** This model forbids more than one processor from reading or writing the same memory cell simultaneously. This is the most restrictive PRAM model proposed.
- **CREW – PRAM Model (Concurrent Read, Exclusive Write);** The write conflicts are avoided by mutual exclusion. Concurrent reads to the same memory location arc allowed.
- **ERCW – PRAM Model** – This allows exclusive read or concurrent writes to the same memory location.
- **CRCW – PRAM Model (Concurrent Read, Concurrent Write);** This model allows either concurrent reads or concurrent writes to the same memory location.

VLSI Model

Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays memory arrays and large scale switching networks. The rapid advent of very large scale intergrated (VSLI) technology now computer architects are trying to implement parallel algorithms directly in hardware. An AT2 model is an example for two dimension VLSI chips

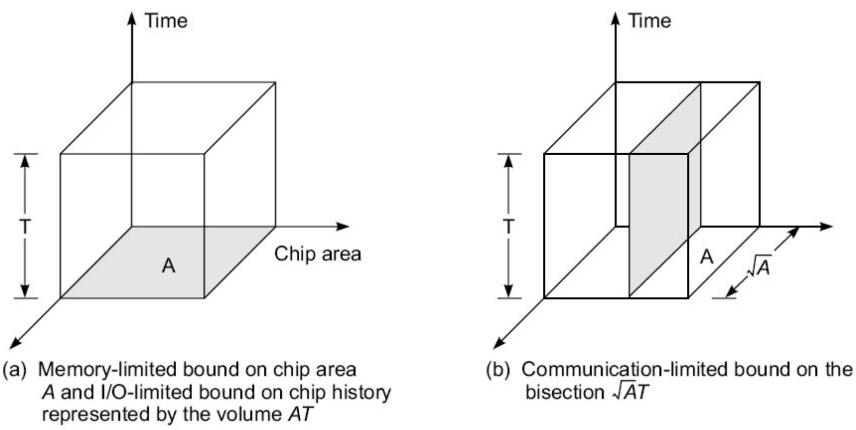


Fig. 1.15 The AT^2 complexity model of two-dimensional VLSI chips

Chapter 2: Program & Network properties

Condition of parallelism

2.2.1 Data and Resource Dependence

- The ability to execute several program segments in parallel requires each segment to be independent of the other segments. We use a dependence graph to describe the relations.
- The nodes of a dependence graph correspond to the program statement (instructions), and directed edges with different labels are used to represent 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 type of data dependence are defined below:

1. **Flow dependence:** A statement S2 is flow dependent on S1 if an execution path exists from s1 to S2 and if at least one output (variables assigned) of S1feeds in as input (operands to be used) to S2 also called RAW hazard and denoted as $S_1 \rightarrow S_2$

2. **Antidependence:** Statement S2 is antidependent on the statement S1 if S2 follows S1 in the program order and if the output of S2 overlaps the input to S1 also called RAW hazard and denoted as $S_1 \nrightarrow S_2$

3. **Output dependence:** Two statements are output dependent if they produce (write) the same output variable. Also called WAW hazard and denoted as $S_1 \circ\!\!\rightarrow S_2$

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 referenced by both I/O statement.

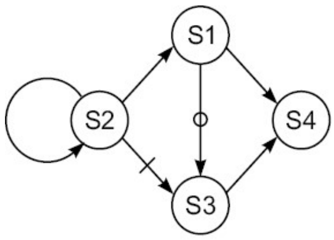
5. **Unknown dependence:** The dependence relation between two statements cannot be determined in the following situations:

- 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 non linear in the loop index variable.

Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

Example: Consider the following fragment of a program:

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)/**



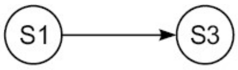
(a) Dependence graph

• Here the Flow dependency from **S1 to S2**, **S3 to S4**, **S2 to S2**

• Anti-dependency from **S2 to S3**

• Output dependency **S1 toS3**

S1: Read (4), A(I)	/Read array A from file 4/
S2: Rewind (4)	/Process data/
S3: Write (4), B(I)	/Write array B into file 4/
S4: Rewind (4)	/Close file 4/



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

The read/write statements S1 and S2 are I/O dependent on each other because they both access the same file.

Control Dependence:

- This refers to the situation where the order of the execution of statements cannot be determined before run time.
- For example all condition statement, where the flow of statement depends on the output.
- Different paths taken after a conditional branch may depend on the data hence we need to eliminate this data dependence among the instructions.
- This dependence also exists between operations performed in successive iterations of looping procedure. Control dependence often prohibits parallelism from being exploited.

Control-independent example:

```
for (i=0; i<n; i++)
{
        a[i] = c[i];
        if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example:

```
for (i=1; i<n; i++)
{
        if (a[i-1] < 0) a[i] = 1;
}
```

Control dependence also avoids parallelism to being exploited. Compilers are used to eliminate this control dependence and exploit the parallelism.

Resource dependence:

- Data and control dependencies are based on the independence of the work to be done.
- Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc. ALU conflicts are called ALU dependence.
- Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions

Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

Notation

- I_i is the set of all input variables for a process P_i. I_i is also called the read set or domain of Pi. Oi is the set of all output variables for a process Pi . Oi is also called write set.
- If P1 and P2 can execute in parallel (which is written as P1 || P2), then:

$$I_1 \cap O_2 = \varnothing$$
$$I_2 \cap O_1 = \varnothing$$
$$O_1 \cap O_2 = \varnothing$$

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.
- The parallelism relation || is commutative (Pi || Pj implies Pj || Pi), but not transitive (Pi || Pj and Pj || Pk does not imply Pi || Pk) .
- Therefore, || is not an equivalence relation. Intersection of the input sets is allowed.

Example: Detection of parallelism in a program using Bernstein's conditions

Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following 5 statements labeled P1, P2, P3, P4, P5 in program order.

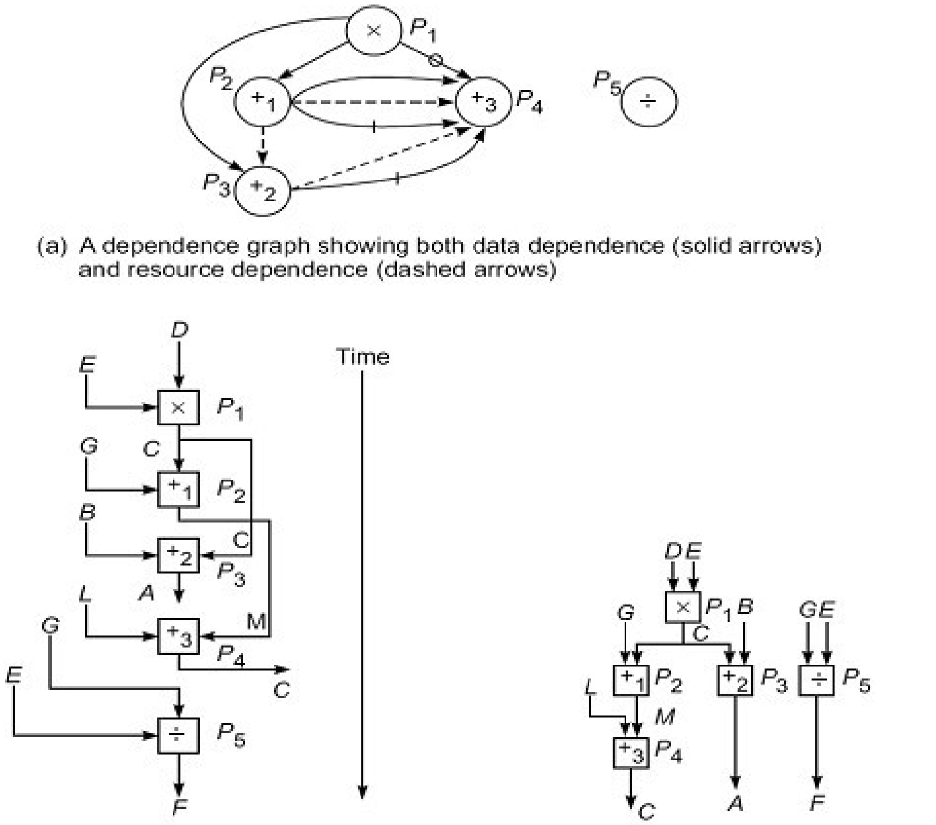
P_1: $C = D \times E$

P_2: $M = G + C$

P_3: $A = B + C$

P_4: $C = L + M$

P_5: $F = G / E$



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)

(b) Sequential execution in five steps, assuming one step per statement (no pipelining)

(c) Parallel execution in three steps, assuming two adders are available per step

Fig. 2.2 Detection of parallelism in the program of Example 2.2

- Assume that each statement requires one step to execute. No pipelining is considered here. The dependence graph shown in 2.2a demonstrates flow dependence as well as resource dependence. In sequential execution, five steps are needed (Fig. 2.2b).
- If two adders are available simultaneously, the parallel execution requires only 3 steps as shown in Fig 2.2c.
- Pairwise, there are 10 pairs of statements to check against Bernstein's conditions. Only 5 pairs, P1||P5, P2||P3, P2||P5, P5||P3 and P4||P5 can execute in parallel as revealed in Fig 2.2a if there are no resource conflicts.
- Collectively, only P2||P3||P5 is possible(Fig. 2.2c) because P2||P3, P3||P5 and P5||P2 are all possible.

Hardware and software parallelism

Hardware parallelism

- Hardware parallelism is defined by machine architecture and hardware multiplicity i.e., functional parallelism times the processor parallelism.
- It can be characterized by the number of instructions that can be issued per machine cycle.
- If a processor issues *k* instructions per machine cycle, it is called a *k-issue* processor.
- Conventional processors are *one-issue* machines.
- This provide the user the information about **peak attainable performance**.

Examples: Intel i960CA is a three-issue processor (arithmetic, memory access, branch).
IBM RS -6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).
A machine with *n k*-issue processors should be able to handle a maximum of *nk* threads simultaneously.

Software Parallelism

Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph i.e., it is defined by dependencies with in the code and is a function of algorithm, programming style, and compiler optimization.

Example: Mismatch between Software parallelism and Hardware parallelism

- Consider the example program graph in Fig. 2.3a. 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.
- 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 Fig. 2.3b. 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.

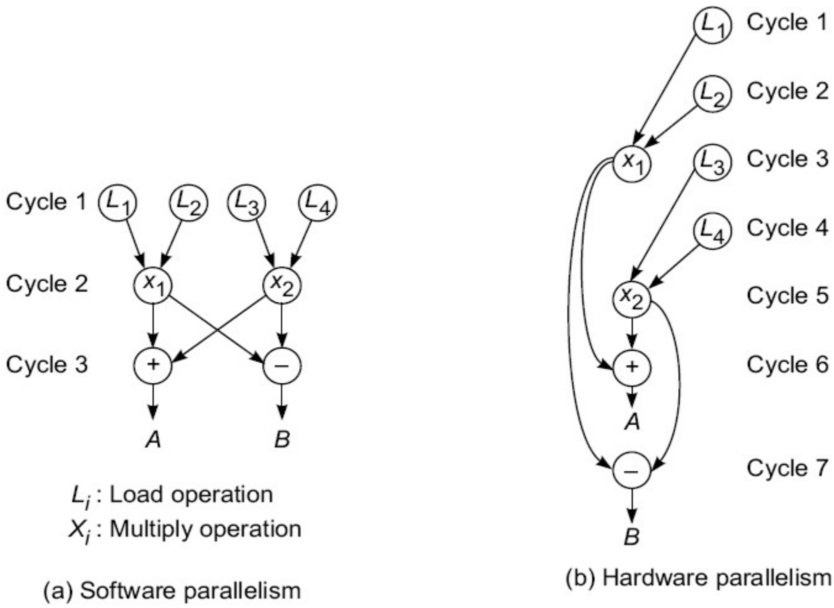


Fig. 2.3 Executing an example program by a two-issue superscalar processor

- Let us try to match the software parallelism shown in Fig. 2.3a in a hardware platform of a dual-processor system, where single-issue processors are used.
- The achievable hardware parallelism is shown in Fig 2.4. Six processor cycles are needed to execute 12 instructions by two processors.
- S1 and S2 are two inserted store operations, l5 and l6 are two inserted load operations for interprocessor communication through the shared memory.

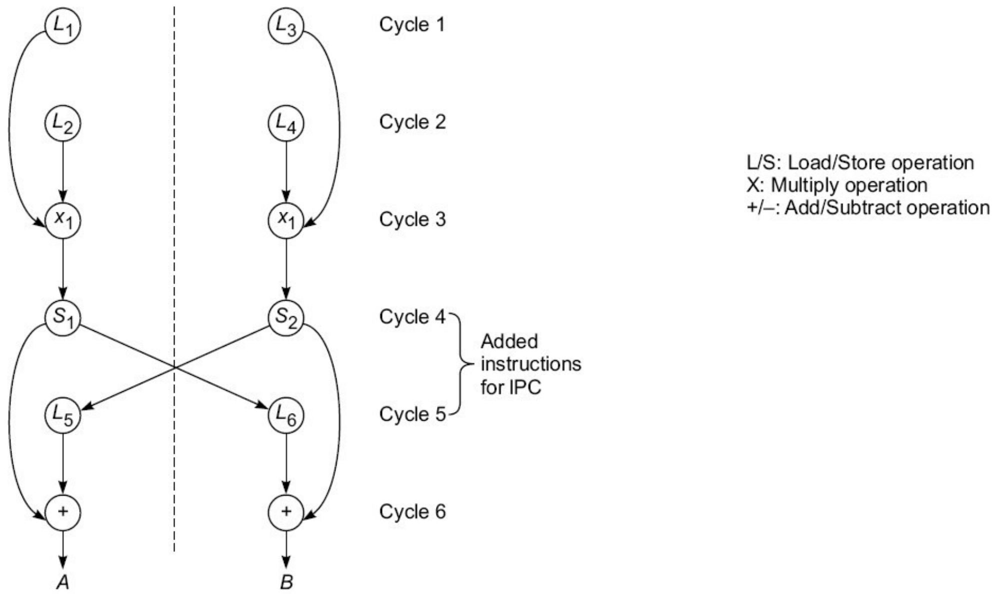


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

The Role of Compilers

- Compilers used to exploit hardware features to improve performance. Interaction between compiler and architecture design is a necessity in modern computer development.
- It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors.
- The hardware and compiler should be designed at the same time.

Program Partitioning & Scheduling

Grain size and latency

- The size of the parts or pieces of a program that can be considered for parallel execution can vary.
- The sizes are roughly classified using the term ―granule size,‖ or simply ―granularity.‖
- The simplest measure, for example, is the number of instructions in a program part.
- Grain sizes are usually described as fine, medium or coarse, depending on the level of parallelism involved.

Latency

Latency is the time required for communication between different subsystems in a computer. Memory latency, for example, is the time required by a processor to access memory. Synchronization latency is the time required for two processes to synchronize their execution. Computational granularity and communication latency are closely related.

Latency and grain size are interrelated and some general observation are

- As grain size decreases, potential parallelism increases, and overhead also increases.
- Overhead is the cost of parallelizing a task. The principle overhead is communication latency.
- As grain size is reduced, there are fewer operations between communication, and hence the impact of latency increases.
- Surface to volume: inter to intra-node comm.

Levels of Parallelism

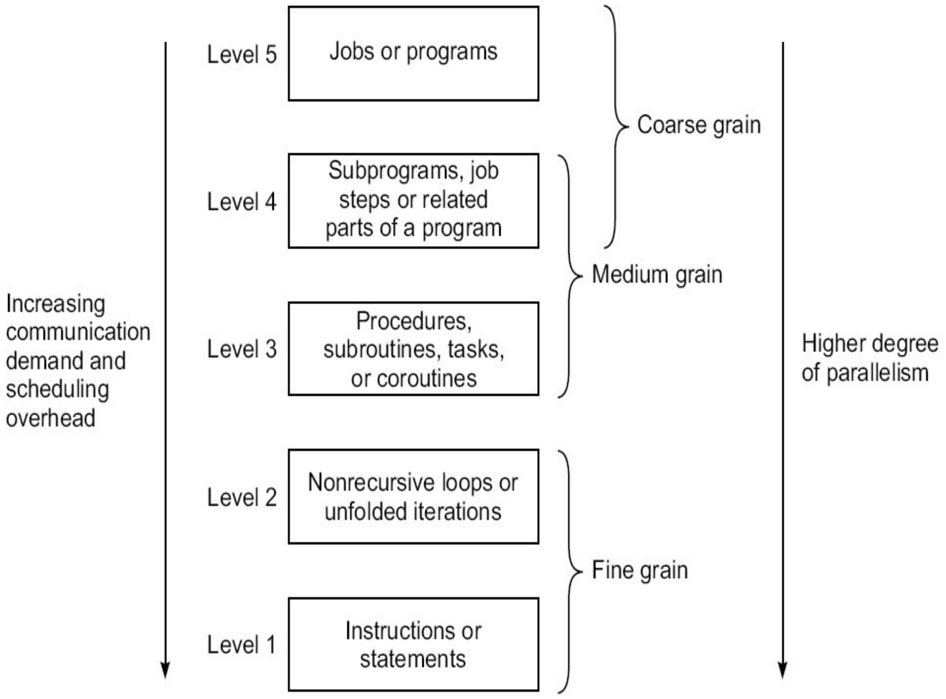


Fig. 2.5 Levels of parallelism in program execution on modern computers (Reprinted from Hwang, *Proc. IEEE*, October 1987)

Instruction Level Parallelism

- This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain.
- The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.

Advantages:

There are usually many candidates for parallel execution. Compilers can usually do a reasonable job of finding this parallelism

Loop-level Parallelism

- Typical loop has less than 500 instructions. If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine.
- Most optimized program construct to execute on a parallel or vector machine.
- Some loops (e.g. recursive) are difficult to handle. Loop-level parallelism is still considered fine grain computation.

Procedure-level Parallelism

- Medium-sized grain; usually less than 2000 instructions.
- Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive.
- Communication requirement less than instruction level SPMD (single procedure multiple data) is a special case Multitasking belongs to this level.

Subprogram-level Parallelism

- Job step level; grain typically has thousands of instructions; medium- or coarse-grain level.
- Job steps can overlap across different jobs. Multiprograming conducted at this level No compilers available to exploit medium- or coarse-grain parallelism at present.

Job or Program-Level Parallelism

- Corresponds to execution of essentially independent jobs or programs on a parallel computer.
- This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

Balancing granularity and latency can yield better performance. Various latencies attributed to machine architecture, technology, and communication patterns used.

Latency imposes a limiting factor on machine scalability.

Ex: Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

Interprocessor Communication Latency

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns **Ex:** n communicating tasks may require n (n - 1)/2 communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include permutations broadcast multicast conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

Grain Packing and Scheduling

Two questions:

- How can I partition a program into parallel ―pieces‖ to yield the shortest execution time?
- What is the optimal size of parallel grains?

There is an obvious tradeoff between the time spent scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.

One approach to the problem is called ―grain packing.‖

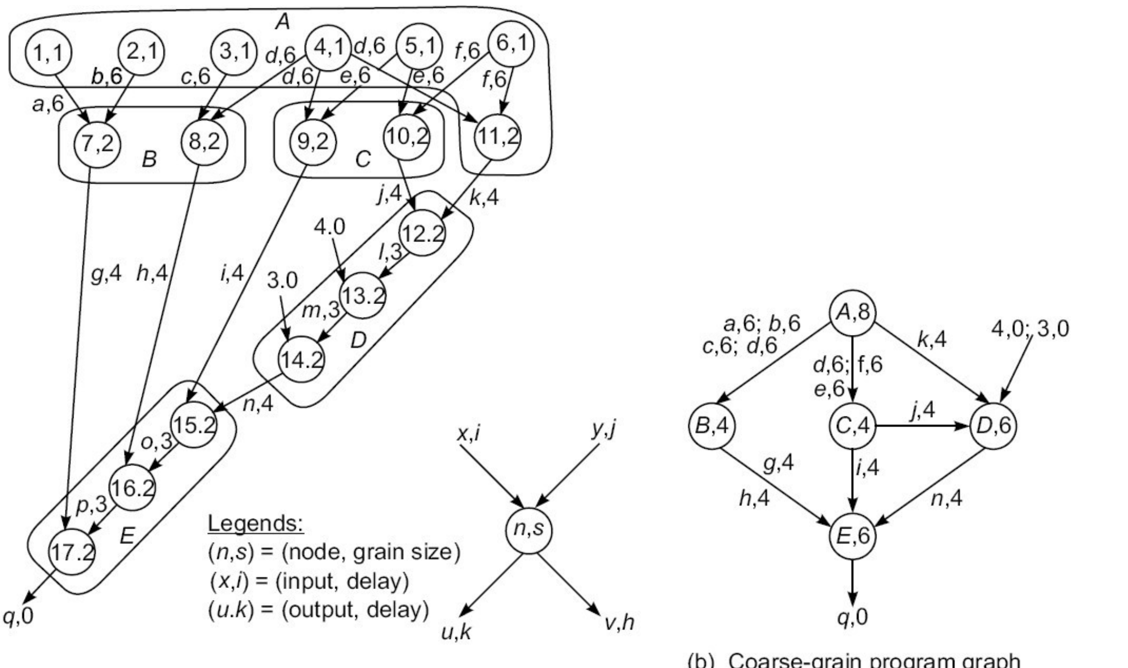
Program Graphs and Packing (Basic concept of Program Partitioning)

- A program graph shows the structure of the program, similar to dependence graph. Each node in the program graph corresponds to a computational unit in the program.
- Grain size is measured by the number of basic machine cycles needed to execute all the operations within the node.
- Each node is denoted by, **Nodes = { (n,s) }**, where n = node name (id), **s = grain size** (larger s = larger grain size), Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.
- **Edges = { (v,d) }**, where v = variable being ―communicated,‖ and d = communication delay.
- Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.
- Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

  Example: Basic concept of Program Partitioning

  Fig. 2.6, shows an example program graph in two different grain sizes.

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



(a) Fine-grain program graph before packing

(b) Coarse-grain program graph after packing

Fig. 2.6 A program graph before and after grain packing in Example 2.4 (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

Scheduling

A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time.

Some general scheduling goals are:

- Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
- Select grain sizes for packing to achieve better schedules for a particular parallel machine.

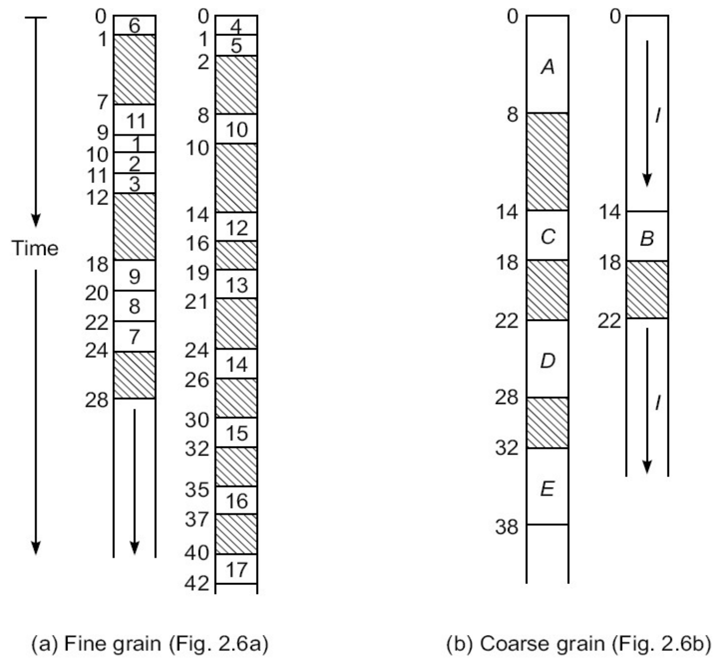
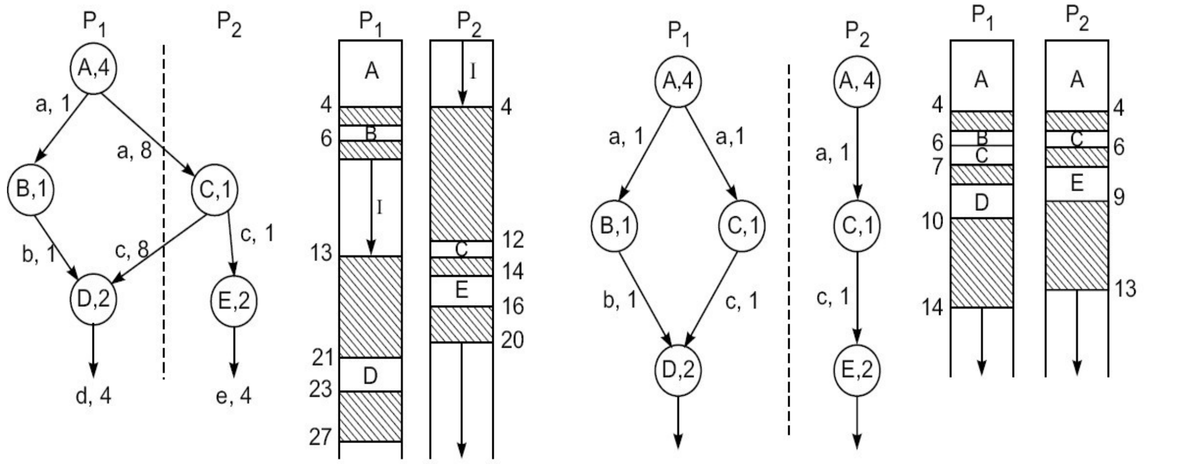


Fig. 2.7 Scheduling of the fine-grain and coarse-grain programs (arrows: idle time; shaded areas: communication delays)

- With respect to the fine-grain versus coarse-grain program graphs in Fig. 2.6, two multiprocessor schedules are shown in Fig. 2.7. The fine-grain schedule is longer (42 time units) because more communication delays were included as shown by the shaded area.
- The coarse-grain schedule is shorter (38 time units) because communication delays among nodes 12, 13 and 14 within the same node D (and also the delays among 15, 16 and 17 within the node E) are eliminated after grain packing.

Node Duplication

- Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule.
- By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.



(a) Schedule without node duplication (b) Schedule with node duplication (A → A and A′; C → C and C′)

Fig. 2.8 Node-duplication scheduling to eliminate communication delays between processors (I: idle time; shaded areas: communication delays)

- Figure 2.8a shows a schedule without duplicating any of the 5 nodes. This schedule contains idle time as well as long interprocessor delays (8 units) between P1 and P2.
- In Fig 2.8b, node A is duplicated into A' and assigned to P2 besides retaining the original copy A in P1.
- Similarly, a duplictaed node C' is copied into P1 besides the original node C in P2.
- The new schedule is shown in Fig. 2.8b is almost 50% shorter than that in Fig. 2.8a. The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.

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: Perform grain packing to produce the coarse grains.

Step 4: Generate a parallel schedule based on the packed graph.

Program Flow Mechanisms

Control Flow vs. Data Flow

- In Control flow computers the next instruction is executed when the last instruction as stored in the program has been executed where as in Data flow computers an instruction executed when the data (operands) required for executing that instruction is available.

- Control flow machines used shared memory for instructions and data.
- Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing.
- Single processor systems are inherently sequential.
- Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. *Data tokens* are passed from an instruction to its dependents to trigger execution.

Program Flow Mechanisms

- **Control flow mechanism**: Conventional machines used control flow mechanism in which order of program execution explicitly stated in user programs.
- **Dataflow machines** which instructions can be executed by determining operand availability.
- **Reduction machines** trigger an instruction's execution based on the demand for its results.

Control flow machines used shared memory for instructions and data. Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing. Single processor systems are inherently sequential.

Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. *Data tokens* are passed from an instruction to its dependents to trigger execution.

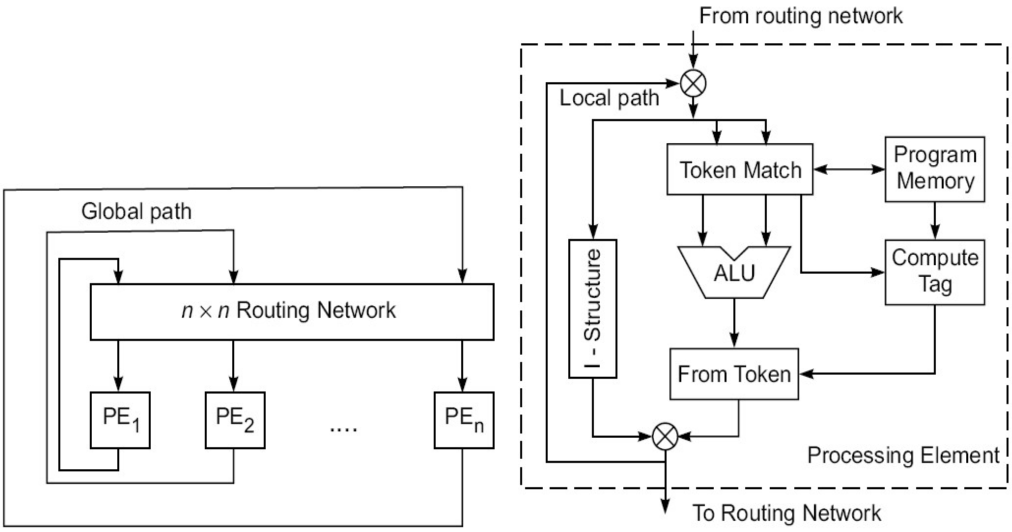
Data Flow Features

No need for

- shared memory
- program counter
- control sequencer

Special mechanisms are required to

- detect data availability
- match data tokens with instructions needing them
- enable chain reaction of asynchronous instruction execution



(a) The global architecture (b) Interior design of a processing element

Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

A Dataflow Architecture

- The Arvind machine (MIT) has N PEs and an N-by-N interconnection network.
- Each PE has a token-matching mechanism that dispatches only instructions with data tokens available.
- Each datum is tagged with
  - address of instruction to which it belongs
  - context in which the instruction is being executed
- Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network.
- Instruction address(es) effectively replace the program counter in a control flow machine.
- Context identifier effectively replaces the frame base register in a control flow machine.
- Since the dataflow machine matches the data tags from one instruction with successors, synchronized instruction execution is implicit.
- An *I-structure* in each PE is provided to eliminate excessive copying of data structures.
- Each word of the I-structure has a **two-bit tag** indicating whether the value **is empty, full** or **has pending read requests**.
- This is a retreat from the pure dataflow approach.
- Special compiler technology needed for dataflow machines.

Demand-Driven Mechanisms

- Demand-driven machines take a top-down approach, attempting to execute the instruction (a *demander*) that yields the final result.
- This triggers the execution of instructions that yield its operands, and so forth.
- The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).

Reduction Machine Models

- **String-reduction model:**
  - each demander gets a separate copy of the expression string to evaluate
  - each reduction step has an operator and embedded reference to demand the corresponding operands
  - each operator is suspended while arguments are evaluated

- **Graph-reduction model:**
  - expression graph reduced by evaluation of branches or subgraphs, possibly in parallel, with demanders given pointers to results of reductions.
  - based on sharing of pointers to arguments; traversal and reversal of pointers continues until constant arguments are encountered.

System interconnect architecture

Various types of interconnection networks have been suggested for SIMD computers. These are basically classified have been classified on network topologies into two categories namely

1. **Static Networks**
2. **Dynamic Networks**

- Direct networks for static connections
- Indirect networks for dynamic connections
- Networks are used for
  - internal connections in a centralized system among
    - processors
    - memory modules
    - I/O disk arrays
  - distributed networking of multicomputer nodes
- The goals of an interconnection network are to provide
  - low-latency

  - high data transfer rate
  - wide communication bandwidth
- Analysis includes
  - latency
  - bisection bandwidth
  - data-routing functions
  - scalability of parallel architecture

- 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.
- Packet switching and routing is playing an important role in modern multiprocessor architecture.

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 IO ports required per node, and thus the cost of a node.
- Therefore, the node degree should be kept a (small) constant, in order to reduce cost.
- 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 bisection width b. In the case of a communication network, each edge may correspond to a channel with w bit wires.

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

Functionality: refers to how the network supports data routing, interrupt handling, synchronization, request-"message combining, and coherence.

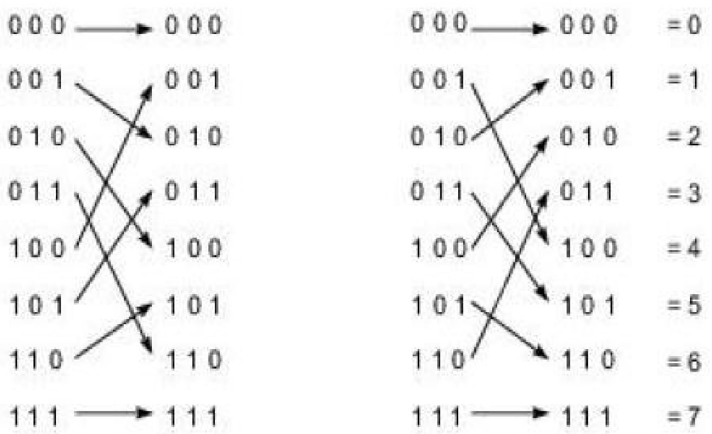
Network Latency:- This refers to the worst-ease time delay for a unit message to be transferred through the network.

Bandwidth: This refers to the maximum data transfer rate, in terms of Mbps or Gbps transmitted through the network.

Hardware Complexity'—This refers to implementation costs such as those for wires, switches, connectors, arbitration, and interface logic.

Scalability—This refers to the ability ofa network to be modularly expandable with a scalable performance with increasing machine resources.

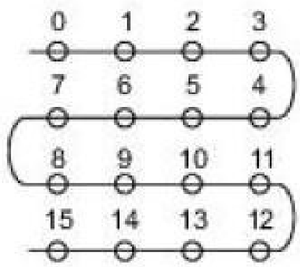
Perfect Shuffle and Exchange Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in Fig. 2.14a. Its inverse is shown on the right-hand side (Fig. 2.14b).



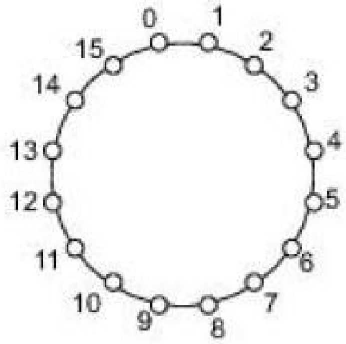
(a) Perfect shuffle (b) Inverse perfect shuffle

2.4.2 Static Connection Networks:

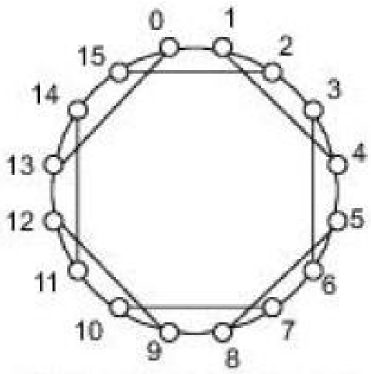
Static networks use direct links which are fixed once built. This type of network is more suitable for building computers where the communication patterns are predictable or implementable with static connections. We describe their topologies below in terms of network parameters and comment on their relative merits in relation to communication and scalability.



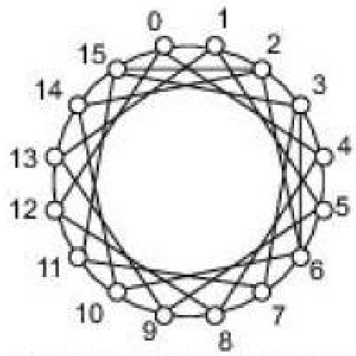
(a) Linear array



(b) Ring



(c) Chordal ring of degree 3



(d) Chordal ring of degree 4 (same as Illiac mesh)

Dynamic Connection Networks

- Dynamic connection networks can implement all communication patterns based on program demands.
- In increasing order of cost and performance, these include
  - bus systems
  - multistage interconnection networks
  - crossbar switch networks
- Price can be attributed to the cost of wires, switches, arbiters, and connectors.
- Performance is indicated by network bandwidth, data transfer rate, network latency, and communication patterns supported.

Digital Buses

- A <u>bus</u> system (<u>contention bus</u>, <u>time-sharing bus</u>) has
  - a collection of wires and connectors
  - multiple modules (processors, memories, peripherals, etc.) which connect to the wires
  - data transactions between pairs of modules

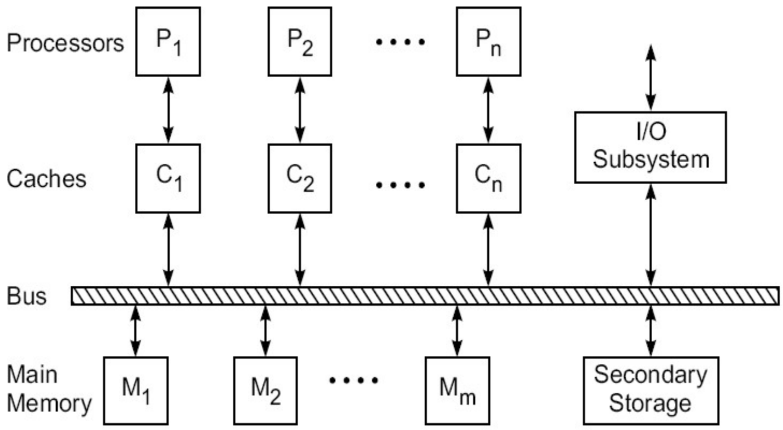


Fig. 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1

- Bus supports only one transaction at a time.
- Bus arbitration logic must deal with conflicting requests.
- Lowest cost and bandwidth of all dynamic schemes.
- Many bus standards are available.

Multistage Networks

Many stages of interconnected switches form a multistage SIMD network. It is basically consist of three characteristic features: The switch box, The network topology and The control structure.

Many stages of interconnected switches form a multistage SIMD networks. Each box is essentially an interchange device with two inputs and two outputs. The four possible states of a switch box are which are shown in figure3.6

1. Straight
2.Exchange
3. Upper broadcast
4. Lower broadcast.

A two function switch can assume only two possible state namely state or exchange states. However a four function switch box can be any of four possible states. A multistage network is capable of connecting any input terminal to any output terminal. Multi-stage networks are basically constructed by so called shuffle-exchange switching element, which is basically a 2 x 2 crossbar. Multiple layers of these elements are connected and form the network.

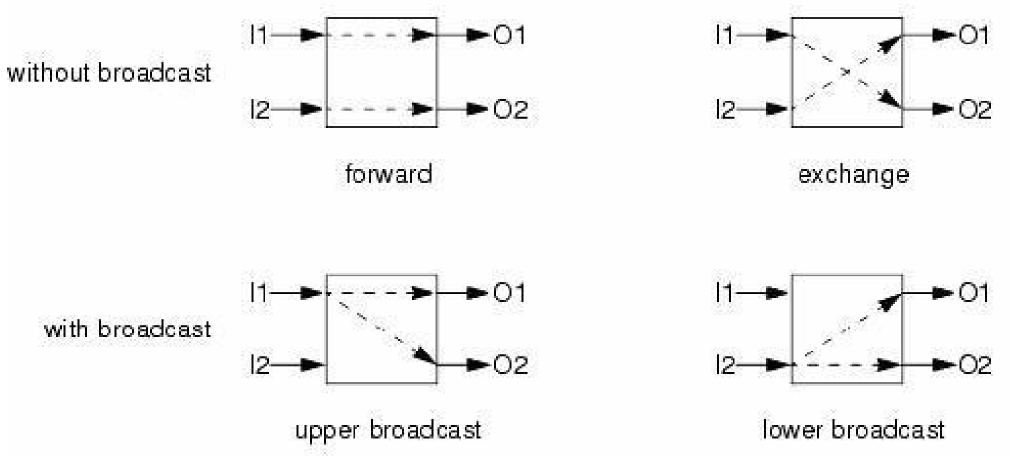


Figure 2.5 A two-by-two switching box and its four interconnection states

A multistage network is capable of connecting an arbitrary input terminal to an arbitrary output terminal. Generally it consists of n stages where $N = 2^n$ is the number of input and output lines. And each stage use N/2 switch boxes. The interconnection patterns from one stage to another stage is determined by network topology. Each stage is connected to the next stage by at least N paths. The total wait time is proportional to the number stages i.e., n and the total cost depends on the total number of switches used and that is $N\log_2 N$. The control structure can be individual stage control i.e., the same control signal is used to set all switch boxes in the same stages thus we need n control signal. The second control structure is individual box control where a separate control signal is used to set the state of each switch box. This provide flexibility at the same time require n2/2 control signal which increases the complexity of the control circuit. In between path is use of partial stage control.

Examples of Multistage Networks

Banyan

Baseline

Cube

Delta

Flip

Indirect cube

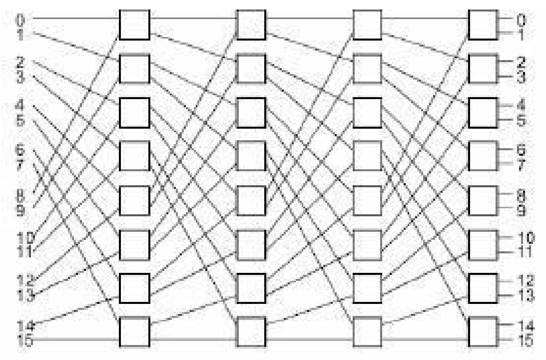
Omega

Multistage network can be of two types

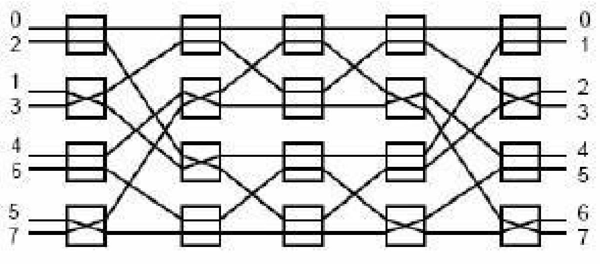
1. One side networks : also called full switch having input output port on the same side
2. Two sided multistage network : which have an input side and an output side.

It can be further divided into three class

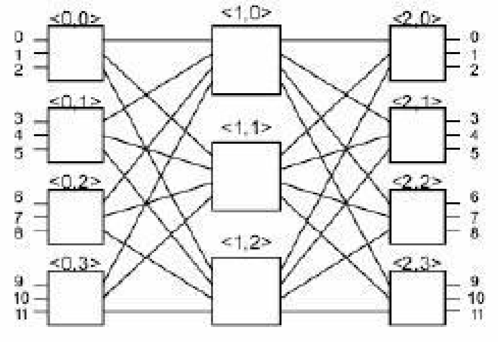
- Blocking: In Blocking networks, simultaneous connections of more than one terminal pair may result conflicts in the use of network communication links. Examples of blocking network are the Data Manipulator, Flip, N cube, omega, baseline. All multistage networks that are based on shuffle-exchange elements, are based on the concept of blocking network because not all possible here to make the input-output connections at the same time as one path might block another. The figure **2.0** (a) show an omega network.
- Rearrangeable : In rearrangeable network, a network can perform all possible connections between inputs and outputs by rearranging its existing connections so that a connection path for a new input-output pair can always be established. An example of this network topology is Benes Network (see figure 2.6 (b) showing a 8** Benes network)which support synchronous data permutation and a synchronous inter-processor communication.
- Non blocking : A non –blocking network is the network which can handle all possible connections without blocking. There two possible cases first one is the Clos network (see figure 2.6(c)) where a one to one connection is made between input and output. Another case of one to many connections can be obtained by using crossbars instead of the shuffle- exchange elements. The cross bar switch network can connect every input port to a free output port without blocking.



(a) Omega Network



(b) Benes Network



(c) Clos Network

Figure 2.6 Several Multistage Interconnection networks Mesh-Connected Illiac Networks

A single stage recirculating network has been implemented in the ILLiac –IV array with N= 64 PEs. Here in mesh network nodes are arranged as a q-dimensional lattice. The neighboring nodes are only allowed to communicate the data in one step i.e., each PEi is allowed to send the data to any one of PE(i+1) , PE (i-1), Pe(i+r) and PE(i-r) where r= square root N(in case of Iliac r=8). In a *periodic mesh*, nodes on the edge of the mesh have wrap-around connections to nodes on the other side this is also called a *toroidal mesh*.

Mesh Metrics

For a q-dimensional non-periodic lattice with kq nodes:

• Network connectivity =q

• Network diameter =q(k-1)

• Network narrowness =k/2

• Bisection width =kq-1

• Expansion Increment =kq-1

• Edges per node =2q

Thus we observe the output of IS k is connected to inputs of OSj where j = k-1,K+1,k-r,k+r as shown in figure below.

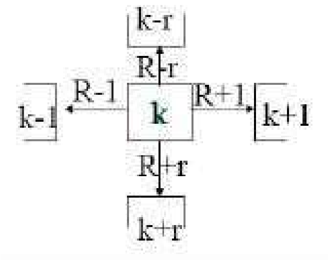


Figure2.7 routing function of mesh Topology

Similarly the OSj gets input from ISk for K= j-1, j+1,j-r,j+r. The topology is formerly described by the four routing functions:

• R+1(i)= (i+1) mod N =>(0,1,2…,14,15)

• R-1(i)= (i-1) mod N =>(15,14,…,2,1,0)

• R+r(i)= (i+r) mod N => (0,4,8,12)(1,5,9,13)(2,6,10,14)(3,7,11,15)

• R-r(i)= (i-r) mod N =>(15,11,7,3)(14,10,6,2)(13,9,5,1)(12,8,4,0)

The figure given below show how each PEi is connected to its four nearest neighbors in the mesh network. It is same as that used for IILiac –IV except that w had reduced it for N=16 and r=4. The index are calculated as module N.

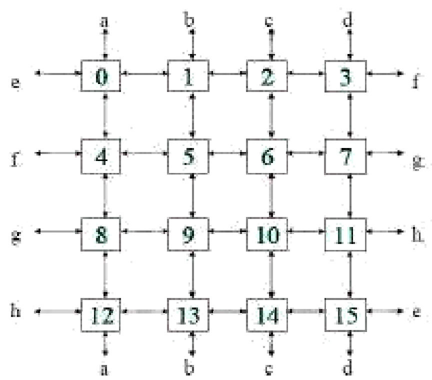


Figure 2.8 Mesh Connections

Thus the permutation cycle according to routing function will be as follows: Horizontally, all PEs of all rows form a linear circular list as governed by the following two permutations, each with a single cycle of order N. The permutation cycles (a b c) (d **b)** stands for permutation a->b, b->c, c->a and d->e, e->d in a circular fashion with each pair of parentheses.

R+1 = (0 1 2 ….N-1)

R–1 = (N-1 ….. 2 1 0).

Similarly we have vertical permutation also and now by combining the two permutation each with four cycles of order four each the shift distance for example for a network of N = 16 and r = square root(16) = 4, is given as follows:

R +4 = (0 4 8 12)(1 5 9 13)(2 6 10 14)(3 7 1115)

R –4 = (12 8 4 0)(13 9 5 1)(14 10 6 2)(15 11 7 3)

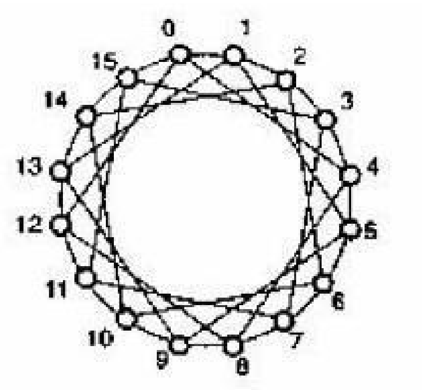


Figure 4.9 Mesh Redrawn

Each PEi is directly connected to its four neighbors in the mesh network. The graph shows that in one step a PE can reach to four PEs, seven PEs in two step and eleven PEs in three steps. In general it takes I steps (recirculations) to route data from PE_i to another PEj for a network of size N where I is upper –bound given by I<= square root(N) -1

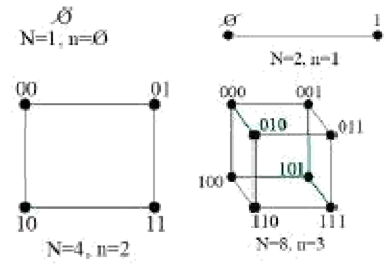
Thus in above example for N=16 it will require at most 3 steps to route data from one PE to another PE and for Illiac –IV network with 64 PE need maximum of 7 steps for routing data from one PE to Another.

Cube Interconnection Networks

The cube network can be implemented as either a recirculating network or as a multistage network for SIMD machine. It can be 1-D i.e., a single line with two pE each at end of a line, a square with four PEs at the corner in case of 2-D, a cube for 3-D and hypercube in

4-D. in case of n-dimension hypercube each processor connects to 2n neighbors. This can be also visualized as the unit (hyper) cube embedded in d-dimensional Euclidean space, with one corner at 0 and lying in the positive orthant. The processors can be thought of as lying at the corners of the cube, with their (x1,x2,...,xd) coordinates identical to their processor numbers, and connected to their nearest neighbors on the cube. The popular examples where cube topology is used are : iPSC, nCUBE, SGI O2K.

Vertical lines connect vertices (PEs) whose address differ in the most significant bit position. Vertices at both ends of the diagonal lines differ in the middle bit position. Horizontal lines differ in the least significant bit position. The unit – cube concept can be extended to an n- dimensional unit space called an n cube with n bits per vertex. A cube network for an SIMD machine with N PEs corresponds to an n cube where $n = \log_2 N$. We use binary sequence to represent the vertex (PE) address of the cube. Two processors are neighbors if and only if their binary address differs only in one digit place.



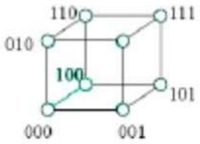
For an n-dimensional cube network of N PEs is specified by the following n routing functions

$C_i (A_{n-1} \ldots A_1 A_0) = A_{n-1} \ldots A_{i+1} A'_i A_{i-1} \ldots A_0$ for i =0,1,2,…,n-1

A n- dimension cube each PE located at the corner is directly connected to n neighbors. The addresses of neighboring PE differ in exactly one bit position. Pease's binary n cube the flip flop network used in staran and programmable switching network proposed for Phoenix are examples of cube networks.

In a recirculating cube network each ISa for 0<=A+< N-1 is connected to n OSs whose addresses are An-1…Ai+1 A'i Ai-1……A0 . When the PE addresses are considered as the corners of an m-dimensional cube this network connects each PE to its m neighbors. The interconnections of the PEs corresponding to the three routing function C0, C1 and C2 are shown separately in below figure.

• Examples



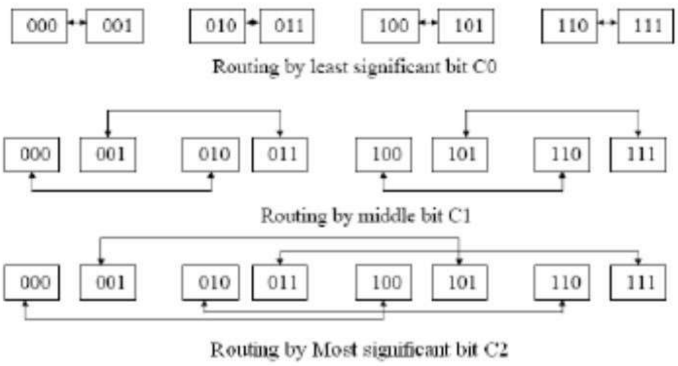


Figure 2.10 The recirculating Network

It takes n<= log2 N steps to rotate data from any PE to another.

Example: N=8 => n=3

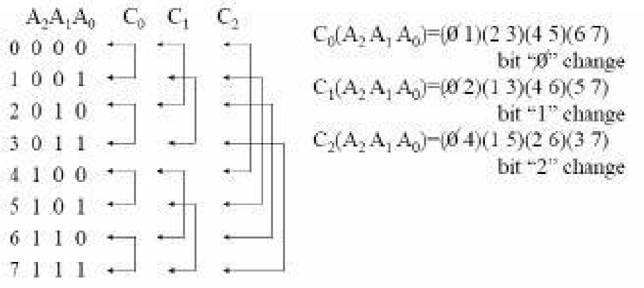


Figure 2.11 Possible routing in multistage Cube network for N = 8

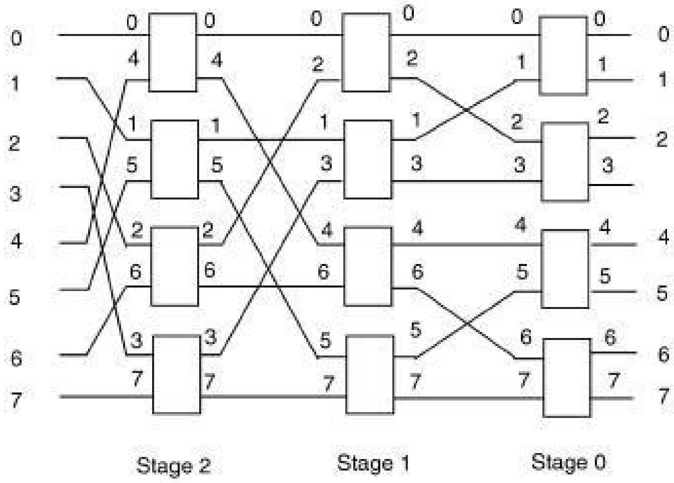


Figure 2.12 **A multistage Cube network for N = 8**

The same set of cube routing functions i.e., C0,C1, C2 can also be implemented by three stage network. Two functions switch box is used which can provide either straight and exchange routing is used for constructing multistage cube networks. The stages are numbered as 0 at input end and increased to n-1 at the output stage i.e., the stage I implements the Ci routing function or we can say at ith stage connect the input line to the output line that differ from it only at the ith bit position.

This connection was used in the early series of Intel Hypercubes, and in the CM-2. Suppose there are 8 process ring elements so 3 bits are required for there address. and that processor 000 is the root. The children of the root are gotten by toggling the first address bit, and so are 000 and 100 (so 000 doubles as root and left child). The children of the children are gotten by toggling the next address bit, and so are 000, 010, 100 and

110. Note that each node also plays the role of the left child. Finally, the leaves are gotten by toggling the third bit. Having one child identified with the parent causes no problems as long as algorithms use just one row of the tree at a time. Here is a picture.

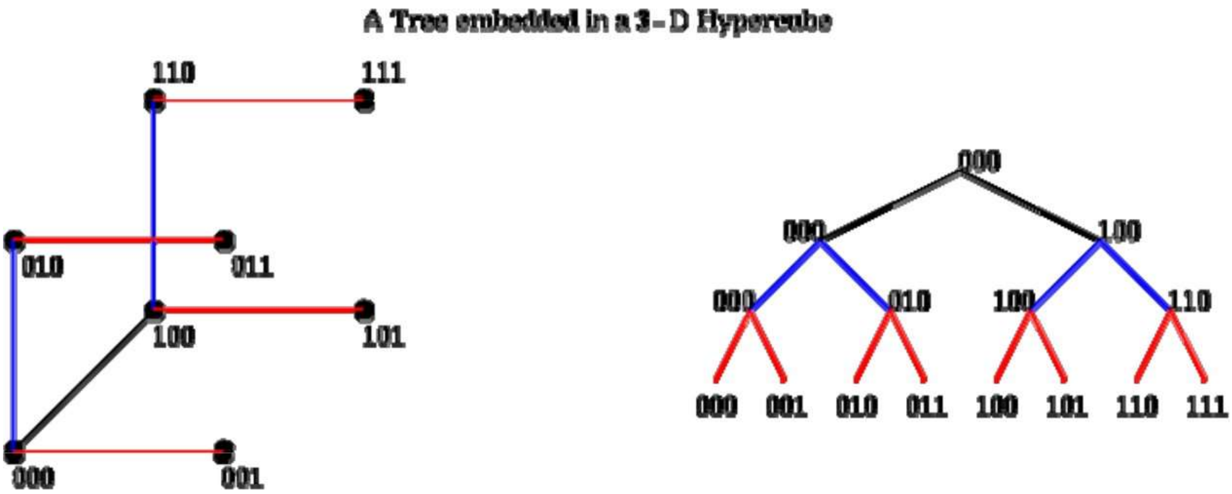


Figure 2.13 A tree embedded in 3-D hypercube Shuffle-Exchange Omega Networks

A shuffle-exchange network consists of $n=2^k$ nodes and it is based on two routing functions shuffle (S) and exchange (E). Let A= An-1…A1A0be the address of a PE than a shuffle function is given by:

S(A)=S(An-1…A1A0)=A.n-2…A1A0An-1, 0<A<1

The cyclic shifting of the bits in A to the left for one bitosition is performed by the S function. Which is effectively like shuffling the bottom half of a card deck into the top half as shown in figure below.

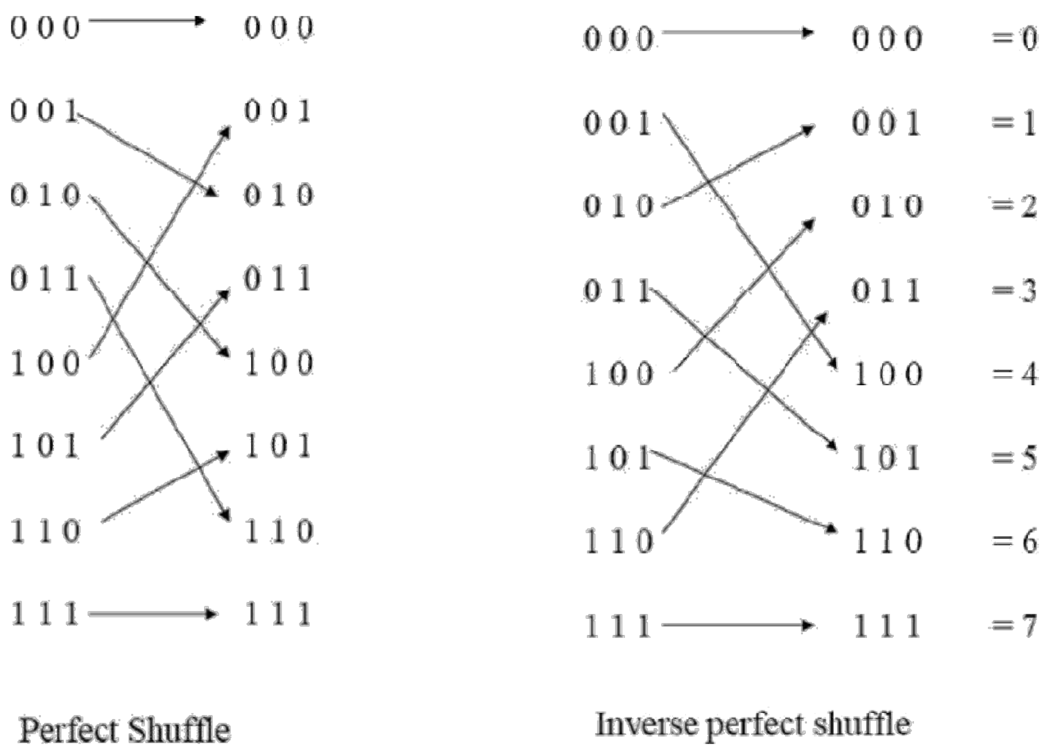


Figure 2.14 Perfect shuffle and inverse perfect shuffle

There are two type of shuffle the perfect shuffle cuts the deck into two halves from the centre and intermix them evenly. *Perfect shuffle provide the routing* connections of node i with node 2i mod(n-1), except for node n-1 which is connected to itself. The inverse perfect shuffle does the opposite to restore the original order it is denoted as exchange routing function E and is defined as :

E(An-1…A1A0)= (An-1…A1A0')

This obtained by complementing the least significant digit means data exchange between two PEs with adjacent addresses. The E(A) is same as the cube routing function as described earlier. *Exchange routing function* connects nodes whose numbers differ in their lowest bit.

The shuffle exchange function can be implemented as either a recirculating network or multistage network. The implementation of shuffle and exchange network through recirculating network is shown below. Use of shuffle and exchange topology for parallel processing was proposed by Stone. It is used for solving many parallel algorithms efficiently. The example where it is used include FFT (fast Fourier transform), sorting, matrix transposition , polynomial evaluations etc.

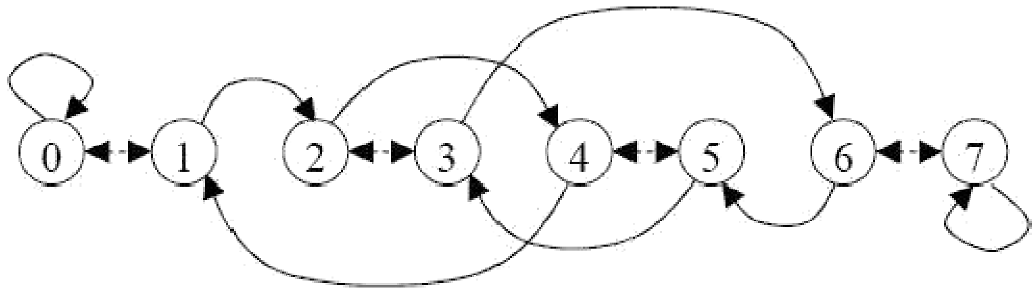
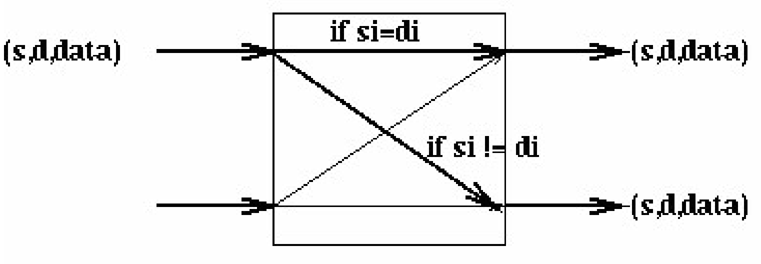


Figure2.15 shuffle and exchange recirculating network for N=8

The shuffle –exchange function have been implemented as multistage Omega network by LAwrie. An N by N omega network, consists of n identical stages. Between two adjacent column there is a perfect shuffle interconnection. Thus after each stage there is a N/2 four-function interchange boxes under independent box control. The four functions are namely straight exchange upper broadcast and lower broadcast. The shuffle connects output P n-l...Pl P0 of stage i to input P n-2...PlP0Pn-l of stage i-1. Each interchange box in an omega network is controlled by the n-bit destination tags associated with the data on its input lines.



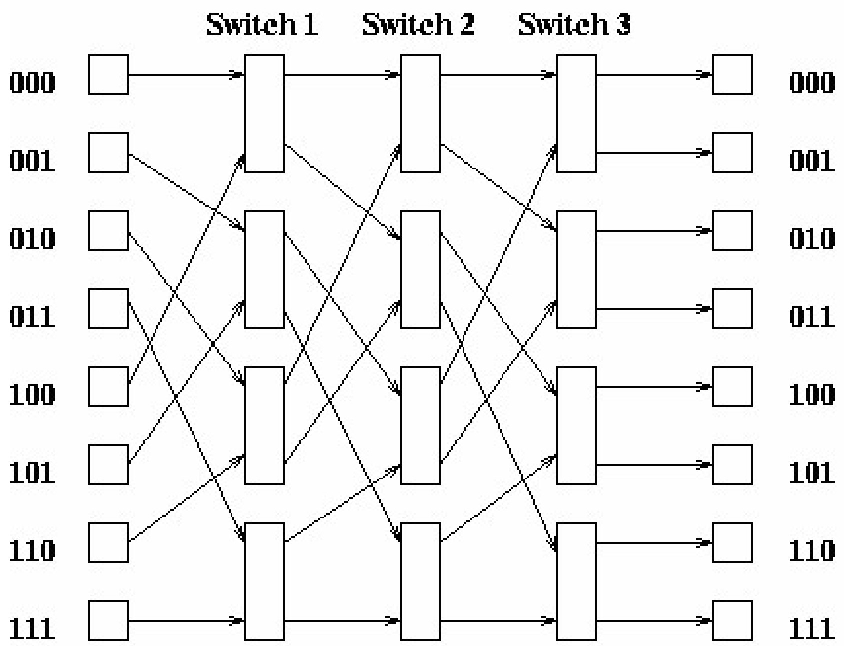


Figure 2.16

The diameter is m=log_2 p, since all message must traverse m stages. The bisection width is p. This network was used in the IBM RP3, BBN Butterfly, and NYU Ultracomputer. If we compare the omega network with cube network we find Omega network can perform one to many connections while n-cube cannot. However as far as bijections connections n-cube and Omega network they perform more or less same.

Summary

Fine-grain exploited at instruction or loop levels, assisted by the compiler. medium-grain (task or job step) requires programmer and compiler support. Coarse-grain relies heavily on effective OS support.
Shared-variable communication used at fine- and medium grain levels.
Message passing can be used for medium- and coarse grain communication, but fine - grain really need better technique because of heavier communication requirements. Control flow machines give complete control, but are less efficient than other approaches. Data flow (eager evaluation) machines have high potential for parallelism and throughput and freedom from side effects, but have high control overhead, lose time waiting for unneeded arguments, and difficulty in manipulating data structures. Reduction (lazy

evaluation) machines have high parallelism potential, easy manipulation of data structures, and only execute required instructions. But they do not share objects with changing local state, and do require time to propagate tokens Summary of properties of various static network

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Summary of properties of various dynamic networks

Network Characteristics	Bus System	Multistage Network	Crossbar Switch
Minimum Latencyfor unit data transfer	Constant	$O(\log_k n)$	Constant
Bandwidth per processor	O(w/n) to O(w)	O(w) to O(nw)	O(w) to O(nw)
Wiring Complexity	O(w)	$O(nw \log_k n)$	$O(n^2w)$
Switching complexity	O(n)	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one to one at a time	Some permutations and broadcast ,if network unblocked	Allpermutations one at a time.

Metrics of dynamic connected nework

Network	Diameter	Bisection Width	Connectivity	Cost (# of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p\log p$
Dynamic Tree	$2\log p$	1	2	$p-1$

Keywords

Dependence graph : A directed graph whose nodes represent calculations and whose edges represent dependencies among those calculations. If the calculation represented by node k depends on the calculations represented by nodes i and j, then the dependence graph contains the edges i-k and j-k.

data dependency : a situation existing between two statements if one statement can store into a location that is later accessed by the other statement

granularity The size of operations done by a process between communications events. A fine grained process may perform only a few arithmetic operations between processing one message and the next, whereas a coarse grained process may perform millions **control-flow computers** refers to an *architecture* with one or more program counters that determine the order in which instructions are executed.

dataflow A model of parallel computing in which programs are represented as *dependence graphs* and each operation is automatically *blocked* until the values on which it depends are available. The parallel functional and parallel logic programming models are very similar to the dataflowmodel.

network A physical communication medium. A network may consist of one or more *buses*, a *switch*, or the *links* joining processors in a *multicomputer*.

Static networks: point-to-point direct connections that will not change duringprogram execution

Dynamic networks: switched channels dynamically configured to match user program communication demands include buses, crossbar switches, and multistage networks **routing** The act of moving a message from its source to its destination. A routing technique is a way of handling the message as it passes through individual nodes. Diameter D of a network is the maximum shortest path between any two nodes, measured by the number of links traversed; this should be as small as possible (from a communication point of view).

Channel bisection width b = minimum number of edges cut to split a network into two parts each having the same number of nodes. Since each channel has w bit wires, the wire bisection width B = bw. Bisection width provides good indication of maximum communication bandwidth along the bisection of a network, and all other cross sections should be bounded by the bisection width.

Wire (or channel) length = length (e.g. weight) of edges between node

PRINCIPLES OF SCALABLE PERFORMANCE

1. Performance Metrics and Measures

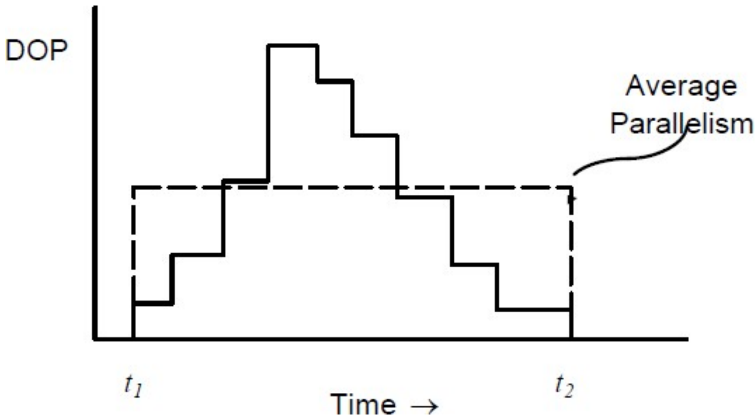
1.1. Parallelism Profile in Programs

1.1.1. Degree of Parallelism

The number of processors used at any instant to execute a program is called the degree of parallelism (DOP); this can vary overtime.

DOP assumes an infinite number of processors are available; this is not achievable in real machines, so some parallel program segments must be executed sequentially as smaller parallel segments. Other resources may impose limiting conditions.

A plot of DOP vs. time is called a parallelism profile.



1.1.2. Average Parallelism –1

Assume the following:

- *n* homogeneous processors
- maximum parallelism in a profile is*m*
- Ideally, *n* >>*m*
- D, the computing capacity of a processor, is something like MIPS or Mflops w/o regard for memory latency,etc.
- *i*is the number of processors busy in an observation period (e.g. DOP = *i)*
- *W* is the total work (instructions or computations) performed by a program
- A is the average parallelism in the program

1.1.3. Average Parallelism –2

1.1.5. Available Parallelism

$$W = \Delta \int_{t_1}^{t_2} DOP(t)\,dt$$

$$W = \Delta \sum_{i=1}^{m} i \cdot t_i$$ where t_i = total time that DOP = i, and $\sum_{i=1}^{m} t_i = t_2 - t_1$

1.1.4. **Average Parallelism –3**

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t)\,dt$$

$$A = \left(\sum_{i=1}^{m} i \cdot t_i \right) / \left(\sum_{i=1}^{m} t_i \right)$$

Various studies have shown that the potential parallelism in scientific and engineering calculations can be very high (e.g. hundreds or thousands of instructions per clock cycle).

But in real machines, the actual parallelism is much smaller (e.g. 10 or 20).

1.1.6. BasicBlocks

A basic block is a sequence or block of instructions with one entry and one exit.

Basic blocks are frequently used as the focus of optimizers in compilers (since its easier to manage the use of registers utilized in the block).

Limiting optimization to basic blocks limits the instruction level parallelism that can be obtained (to about 2 to 5 in typical code).

1.1.7. Asymptotic Speedup –1

$$W_i = i \Delta t_i$$ (work done when DOP = i)

$$W = \sum_{i=1}^{m} W_i$$ (relates sum of W_i terms to W)

$$t_i(k) = W_i / k\Delta$$ (execution time with k processors)

$$t_i(\infty) = W_i / i\Delta$$ (for $1 \le i \le m$)

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta} \quad \text{(resp. time w/ 1 proc.)}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \quad \text{(resp. time w/ } \infty \text{ proc.)}$$

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i / i} = A \quad \text{(in the ideal case)}$$

1.2. Mean Performance

We seek to obtain a measure that characterizes the mean, or average, performance of a set of benchmark programs with potentially many different execution modes (e.g. scalar, vector, sequential, parallel).

We may also wish to associate weights with these programs to emphasize these different modes and yield a more meaningful performance measure.

1.2.1. Arithmetic Mean

The arithmetic mean is familiar (sum of the terms divided by the number of terms).Our measures will use execution rates expressed in MIPS or Mflops. The arithmetic mean of a set of execution rates is proportional to the sum of the inverses of the execution times; it is **not** inversely proportional to the sum of the execution times.

Thus arithmetic mean fails to represent real times consumed by the benchmarks when executed.

1.2.2. Harmonic Mean

Instead of using arithmetic or geometric mean, we use the harmonic mean execution rate, which is just the inverse of the arithmetic mean of the execution time (thus guaranteeing the inverse relation not exhibited by the other means).

$$R_h = \frac{m}{\sum_{i=1}^{m} (1/R_i)}$$

1.2.3. Weighted Harmonic Mean

If we associate weights f*i* with the benchmarks, then we can compute the weighted harmonic mean:

$$R_h = \frac{m}{\sum_{i=1}^{m} \left(f_i / R_i \right)}$$

1.2.4. Weighted Harmonic Mean Speedup

T1 = 1/R1 = 1 is the sequential execution time on a single processor with rate R1 = 1.

Ti = 1/Ri = 1/i = is the execution time using i processors with a combined execution rate of Ri =i.

Now suppose a program has *n*execution modes with associated weights *f1* … *f n*. The weighted harmonic mean speedup is defined as:

$$S = T_1 / T^* = \frac{1}{\left(\sum_{i=1}^{n} f_i / R_i \right)}$$

$T^* = 1 / R_h^*$
(weighted arithmetic mean execution time)

1.2.5. Amdahl's Law

Assume R*i* = *i*, and *w* (the weights) are (a, 0, …, 0, 1-a).Basically this means the system is used sequentially (with probability a) or all *n* processors are used (with probability 1- a).

This yields the speedup equation known as Amdahl's law:

$$S_n = \frac{n}{1 + (n-1)\alpha}$$

The implication is that the best speedup possible is 1/ a, regardless of *n*, the number of processors.

1.3. Efficiency, Utilizations, andQuality

1.3.1. System Efficiency – 1

Assume the following definitions:

O (*n*) = total number of "unit operations" performed by an *n* processor system in completing a program *P*.

T (*n*) = execution time required to execute the program *P* on an *n* processor system.

O (*n*) can be considered similar to the total number of instructions executed by the n processors, perhaps scaled by a constant factor.

If we define $O(1) = T(1)$, then it is logical to expect that $T(n) < O(n)$ when $n > 1$ if the program *P* is able to make any use at all of the extra processor(s).

1.3.2. System Efficiency – 2

Clearly, the *speedup factor* (how much faster the program runs with *n* processors) can now be expressed as

$S(n) = T(1) / T(n)$

Recall that we expect $T(n) < T(1)$, so *S* (*n*) ³ 1.

System efficiency is defined as

E (*n*) = *S* (*n*) / *n* = *T* (1) / (*n*′ *T* (*n*))

It indicates the actual degree of speedup achieved in a system as compared with the maximum possible speedup. Thus 1 / *n* £ *E* (*n*) £ 1. The value is $1/n$ when only one processor is used (regardless of *n*), and the value is 1 when all processors are fully utilized.

1.3.3. Redundancy

The redundancy in a parallel computation is defined as

$R(n) = O(n) / O(1)$

What values can *R* (*n*) obtain?

$R(n) = 1$ when $O(n) = O(1)$, or when the number of operations performed is independent of the number of processors, *n*. *This is the ideal case.*

$R(n) = n$ when all processors performs the same number of operations as when only a single processor is used; *this implies that n completely redundant computations are performed*!

The $R(n)$ figure indicates to what extent the software parallelism is carried over to the hardware implementation without having extra operations performed.

1.3.4. System Utilization

System utilization is defined as

$$U(n) = R(n) \times E(n) = O(n) / (n \times T(n))$$

It indicates the degree to which the system resources were kept busy during execution of the program. Since $1 \le R(n) \le n$, and $1/n \le E(n) \le 1$, the best possible value for $U(n)$ is 1, and the worst is $1/n$.

$1/n \le E(n) \le U(n) \le 1$

$1 \le R(n) \le 1/E(n) \le n$

Quality of Parallelism

The quality of a parallel computation is defined as
$Q(n) = S(n) \times E(n) / R(n)$
$= T^3(1) / (n \times T^2(n) \times O(n))$

This measure is directly related to speedup (S) and efficiency (E), and inversely related to redundancy (R).

The quality measure is bounded by the speedup
(that is, $Q(n) \le S(n)$).

Standard Industry Performance Measures

MIPS and Mflops, while easily understood, are poor measures of system performance, since their interpretation depends on machine clock cycles and instruction sets. For example, which of these machines is faster?

a 10 MIPS CISC computer
a 20 MIPS RISC computer

It is impossible to tell without knowing more details about the instruction sets on the machines. Even the question, "which machine is faster," is suspect, since we really need to say "faster at doing what?"

Other Measures

Transactions per second (TPS) is a measure that is appropriate for online systems like those used to support ATMs, reservation systems, and point of sale terminals. The measure may include communication overhead, database search and update, and logging operations. The benchmark is also useful for rating relational database performance. *KLIPS*is the measure of the number of logical inferences per second that can be performed by a system, presumably to relate how well that system will perform at certain AI applications. Since one inference requires about 100 instructions (in the benchmark), a rating of 400 KLIPS is roughly equivalent to 40MIPS.

SPEEDUP PERFORMANCE LAWS

The main objective is to produce the results as early as possible. In other words minimal turnaround time is the primary goal.

Three performance laws defined below:

1. Amdahl's Law(1967) is based on fixed workload or fixed problem size
2. Gustafson's Law(1987) is applied to scalable problems, where the problem size increases with the increase in machine size.
3. The speed up model by Sun and Ni(1993) is for scaled problems bounded by memory capacity.

Amdahl's Law for fixed workload

In many practical applications the computational workload is often fixed with a fixed problem size. As the number of processors increases, the fixed workload is distributed.

Speedup obtained for time-critical applications is called fixed-load speedup.

Fixed-Load Speedup

The ideal speed up formula given below:

$$S\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} wi}{\sum_{i=1}^{m} \frac{wi}{i}}$$

is based on a fixed workload, regardless of machine size.

We consider below two cases of DOP< n and of DOP ≥ n.

Consider the case where DOP = i>n. Assume all n processors are used to execute W_i exclusively. The execution time of W_i is:

$$t_i(n) = \frac{wi}{i\,\Delta} \left\lceil \frac{i}{n} \right\rceil$$

thus the response time is

$$T(n) = \sum_{i=1}^{m} \frac{wi}{i\,\Delta} \left\lceil \frac{i}{n} \right\rceil$$

Now we define the fixed-load speedup factor as the ratio of T(1) to T(n):

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=0}^{n} wi}{\sum_{i=0}^{n} \frac{wi}{i}}$$