Theory of Parallelism: Parallel Computer Models, The State of Computing, Multiprocessors and Multicomputer, Multivector and SIMD Computers, PRAM and VLSI Models, Program and Network Properties, Conditions of Parallelism, Program Partitioning and Scheduling, Program Flow Mechanisms, System Interconnect Architectures, Principles of Scalable Performance, Performance Metrics and Measures, Parallel Processing Applications, Speedup Performance Laws. For all Algorithm or mechanism any one example is sufficient

1.1 The State of Computing

1.1.1 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 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.

1.1.2 Elements of Modern Computers

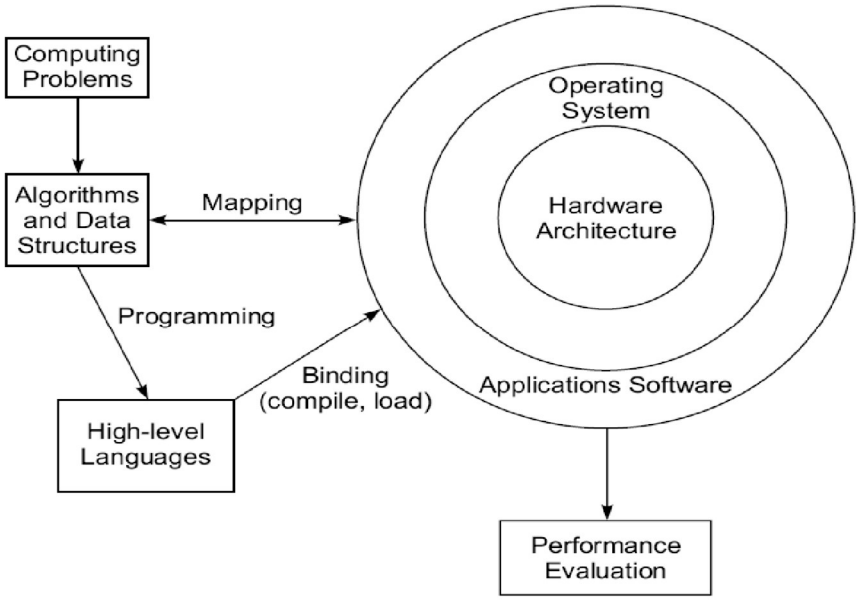


Fig. 1.1 Elements of a modern computer system

- **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
- Processors, memory, and peripheral devices form the hardware core of a computer system. resources, an operating system, and **application** software.

- 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.

1.1.3 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.

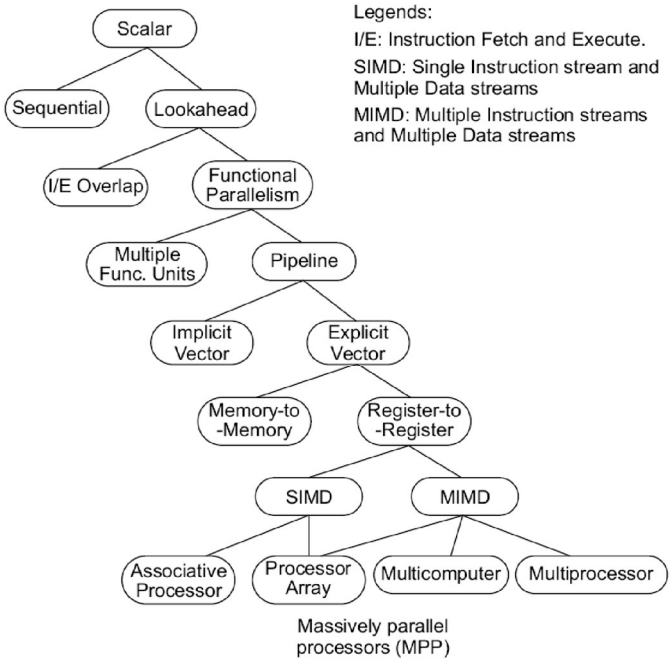


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

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:

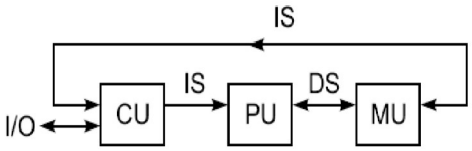
1. using multiple functional units simultaneously,
2. to practice pipelining at various processing levels.

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

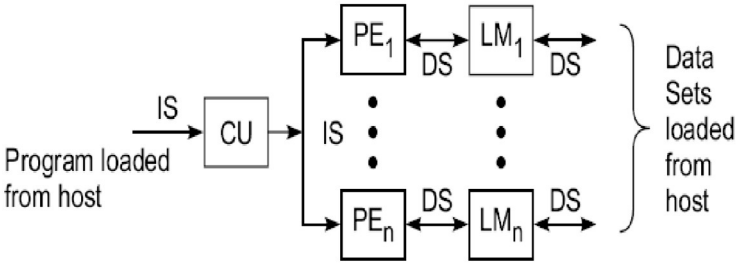


(a) SISD uniprocessor architecture

- 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

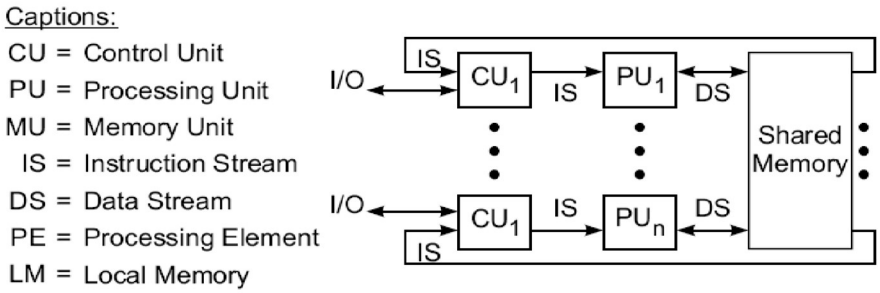


(b) SIMD architecture (with distributed memory)

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

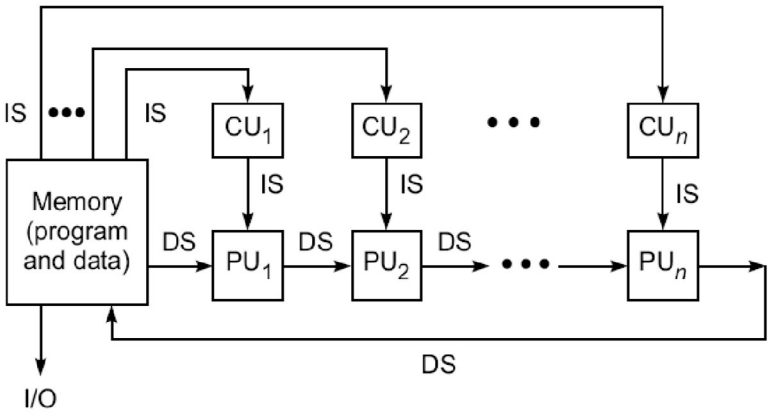
3. MIMD (Multiple Instruction streams over Multiple Data streams) machines.



(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.

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.

1.1.4 System Attributes affecting Performance

System Performance

1. The ideal performance of a computer system demands a perfect match between machine capability and program behavior.
2. Machine capability can be enhanced with better hardware technology, innovative architectural features, and efficient resources management.
3. Factors affecting program behavior, includes algorithm design, data structures, language efficiency, programmer skill, and compiler technology.

Clock Rate and CPI

- The CPU (or simply the processor) of today‘s digital computer is driven by a clock with a constant Clock Time t. The inverse of the cycle time is the Clock rate f = 1/t

- The size of a program is determined by its Instruction Count Ic.

- Different machine instructions may require different numbers of clock cycles to execute which we can call it as Clock cycles Per Instructions (CPI

Performance Factors

- The CPU time (T in seconds / program) needed to execute the program is estimated by finding the product of three contributing factors: T= Ic X CPI X t

- This can be extended as T= Ic X (p +m * k) X t

- Where ‘p’ is the number of processor cycles needed for instruction decode & execution. ‘m’ is the number of memory reference cycles & ‘k’ is the ratio of memory cycle & processor cycle

MIPS Rate

- MIPS Rate: The processor speed is often measured in terms of miflion instructions per second (MIPS).
- MIPS rate varies with respect to a number of factors, including the clock rate (f), the instruction count (Ic), and the CPI of a given machine, as defined below

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

Throughput Rate

- how many programs a system can execute per unit time, called as throughput rate
- In a multi programmed system, the system throughput is often lower than the CPU throughput Wp defined by:

$$W_p = \frac{f}{I_c \times \text{CPI}}$$

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.

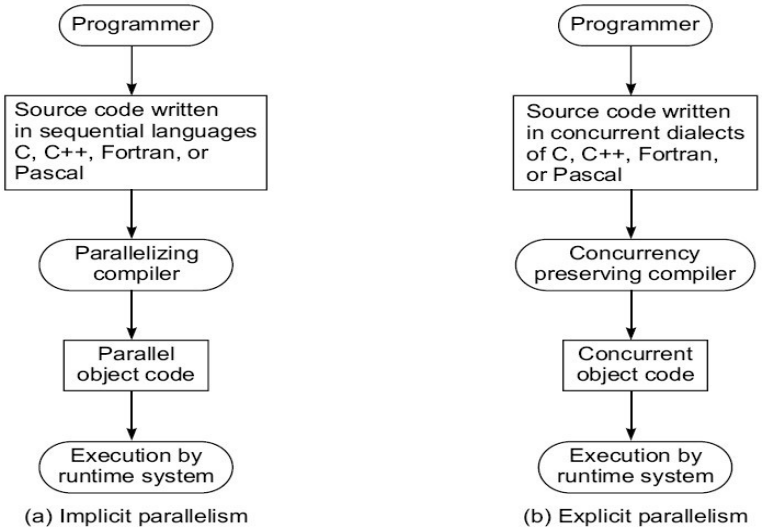


Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from

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.

1.2 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

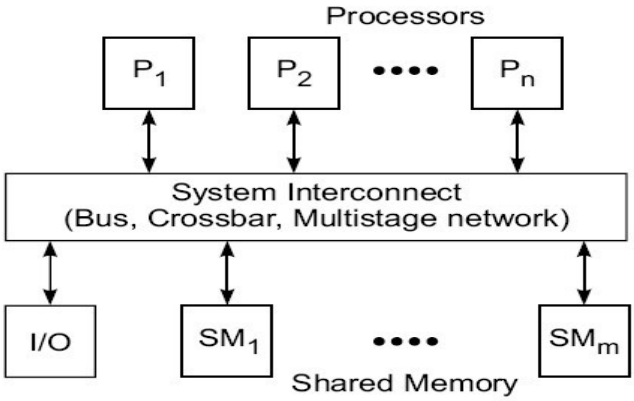


Fig. 1.6 The UMA multiprocessor model

- 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* due 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 .

- 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

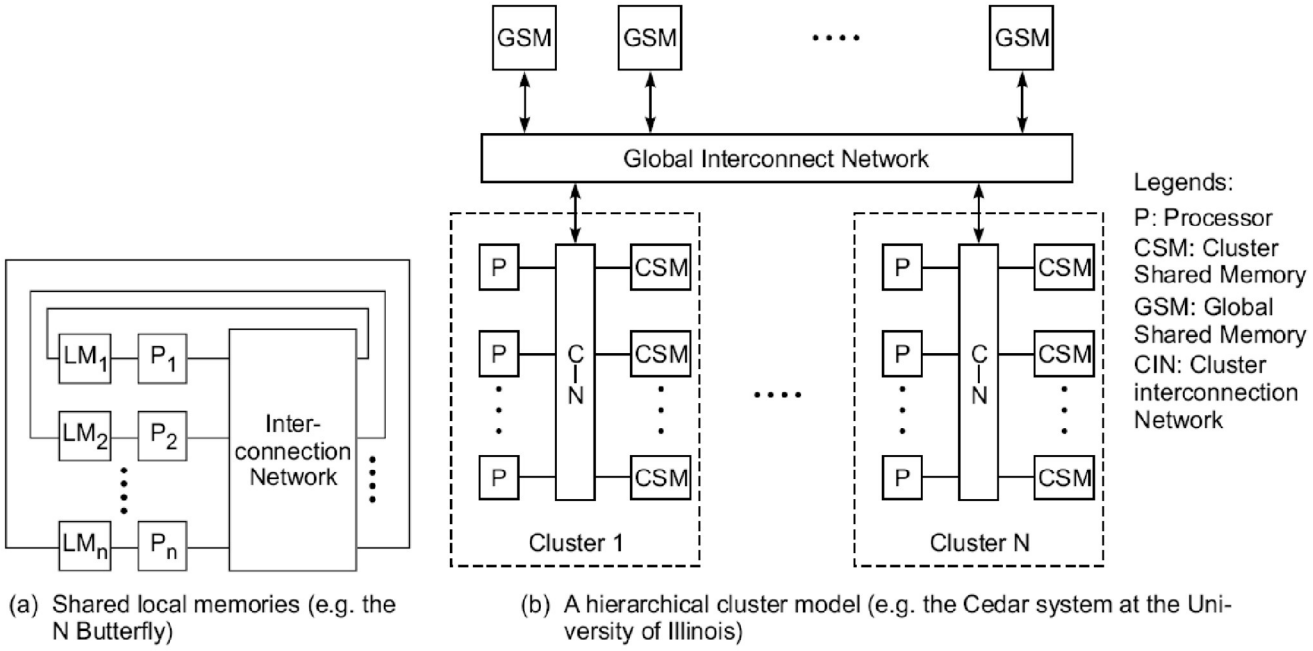


Fig. 1.7 Two NUMA models for multiprocessor systems

- 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

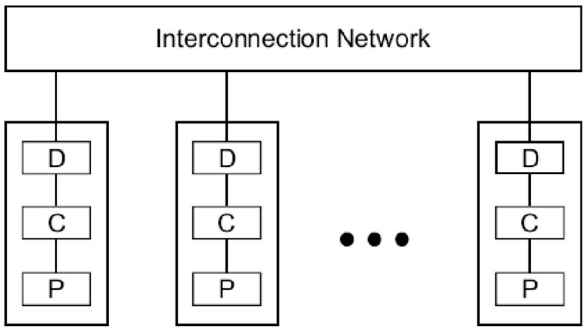


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

- 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

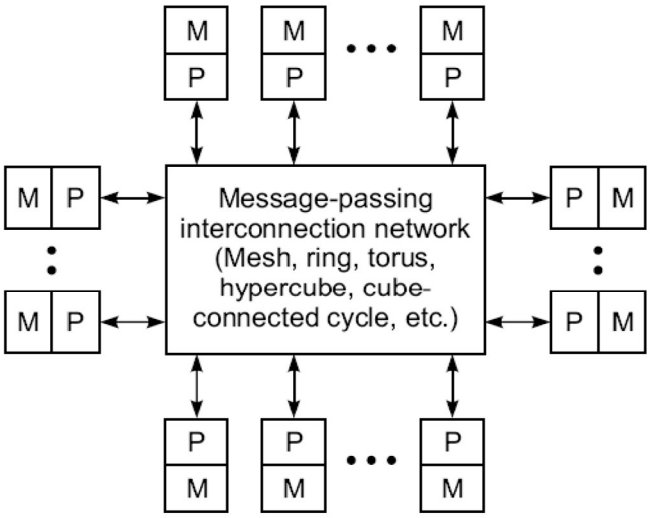


Fig. 1.9 Generic model of a message-passing multicomputer

- 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.

1.3 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

1.3.1 Vector Supercomputers

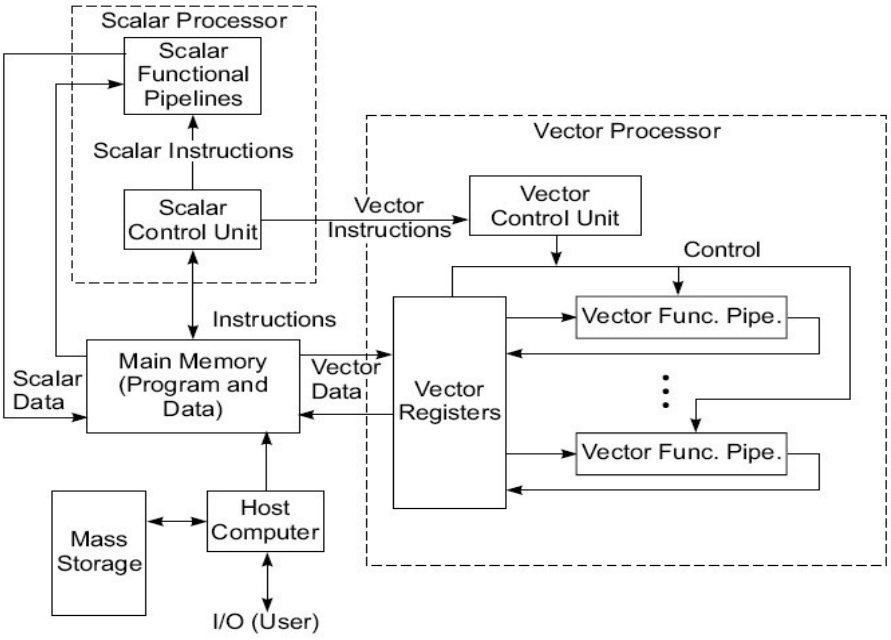


Fig. 1.11 The architecture of a vector supercomputer

- 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.

1.3.2 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.

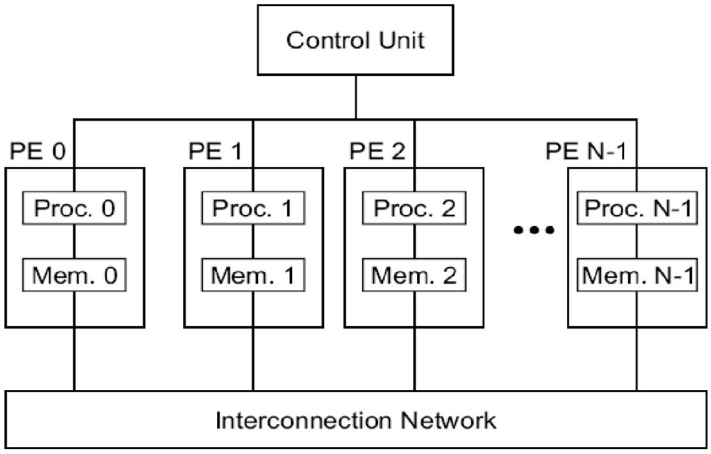


Fig. 1.12 Operational model of SIMD computers

1.4 PRAM AND VLSI MODELS

1.4.1 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.
-

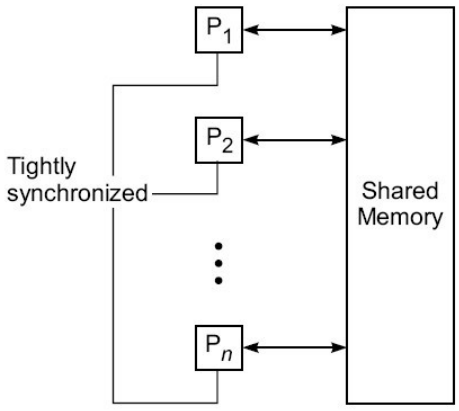


Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all *n* processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

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:

2.1 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 $S_1 \mapsto S_2$

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

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 nondeterministic results.

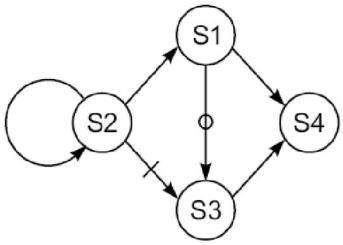
Example: Consider the following fragment of a program:

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

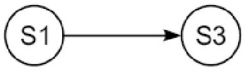
- Here the Flow dependency from **S1 to S2**, **S3 to S4** , **S2 to S2**
- Anti-dependency from **S2 to S3**
- Output dependency **S1 toS3**

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 the set of all output variables for a process *Pi* . Oi is also called write set. □ 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 □ If P*1* and P*2* can execute in parallel (which is written as P*1* || P*2*), 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 (P*i* || P*j* implies P*j* || P*i*), but not transitive (P*i* || P*j* and P*j* || P*k* does not imply P*i* || P*k*) .
- 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.

- 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.

2.1.2 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.

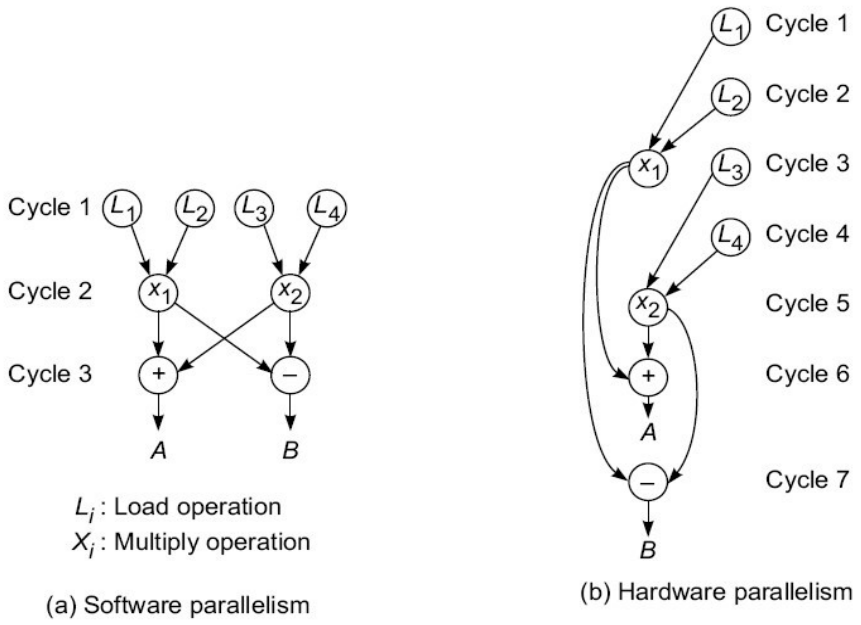


Fig. 2.3 Executing an example program by a two-issue superscalar processor

- Let us try to match the software parallelism shown in Fig. 2.3a in a hardware platform of a dualprocessor 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.

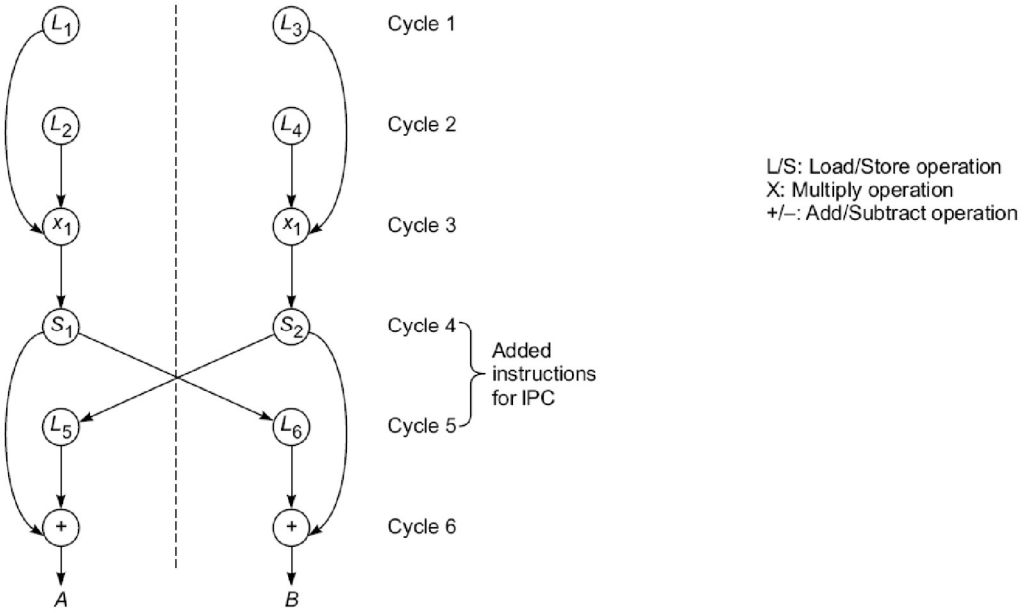


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

2.1.3 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.

2.2 Program Partitioning & Scheduling

2.2.1 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 the time required for two processes to synchronize their execution. Computational granularity and latency, for example, is the time required by a processor to access memory. Synchronization latency is 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

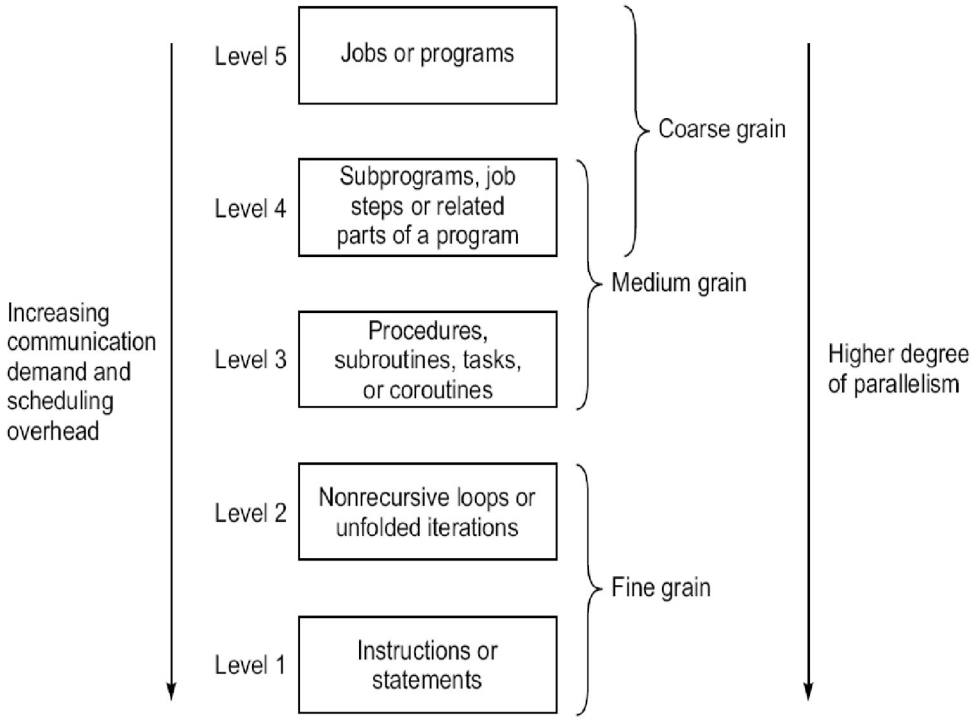


Fig. 2.5 Levels of parallelism in program execution on modern computers (Reprinted from Hwang, *Proc. IEEE*, October 1987)

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.

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

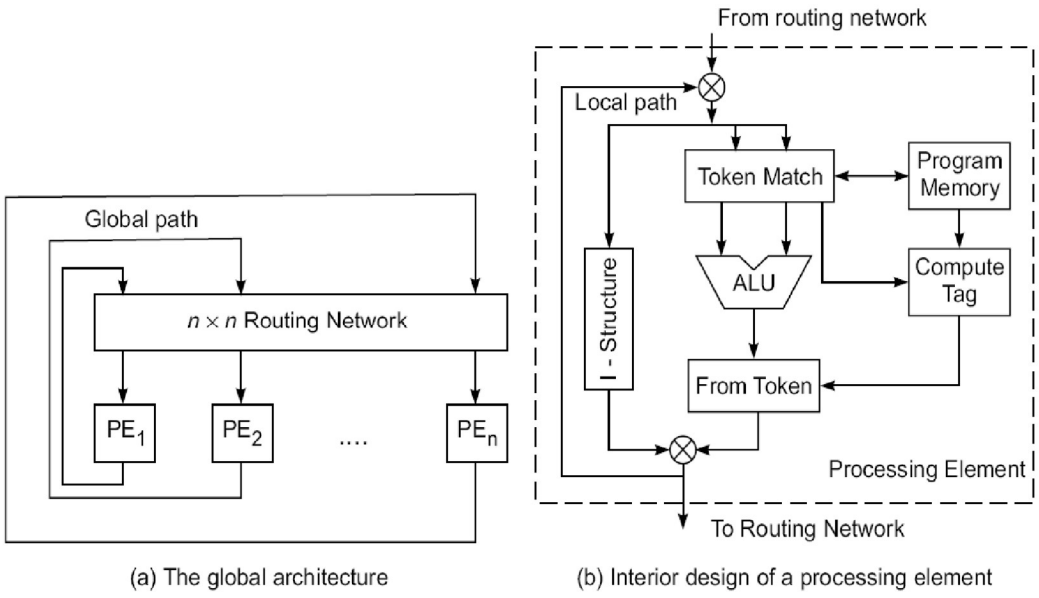


Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

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

- Each datum is tagged with available.

  o address of instruction to which it belongs o 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.