B.Tech.

EIGHTH SEMESTER EXAMINATION, 2008-09

ADVANCE COMPUTER ARCHITECTURE

(TCS-802)

Time : 3 Hours] *[Total Marks : 100*

Note :

*(1) Attempt **all** questions.*

(2) All questions carry equal marks.

Q. 1. (a) Define Parallel Computing ? What are the fundamental issues in parallel processing ? Why parallel computing is required, discuss various applications of parallel computing ?

Ans. Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism-with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to getting good parallel program performance.

Q. 1. (b) Explain how DOP and number of processors affect the performance of a parallel computing system ? Discuss various speeds up performance laws ?

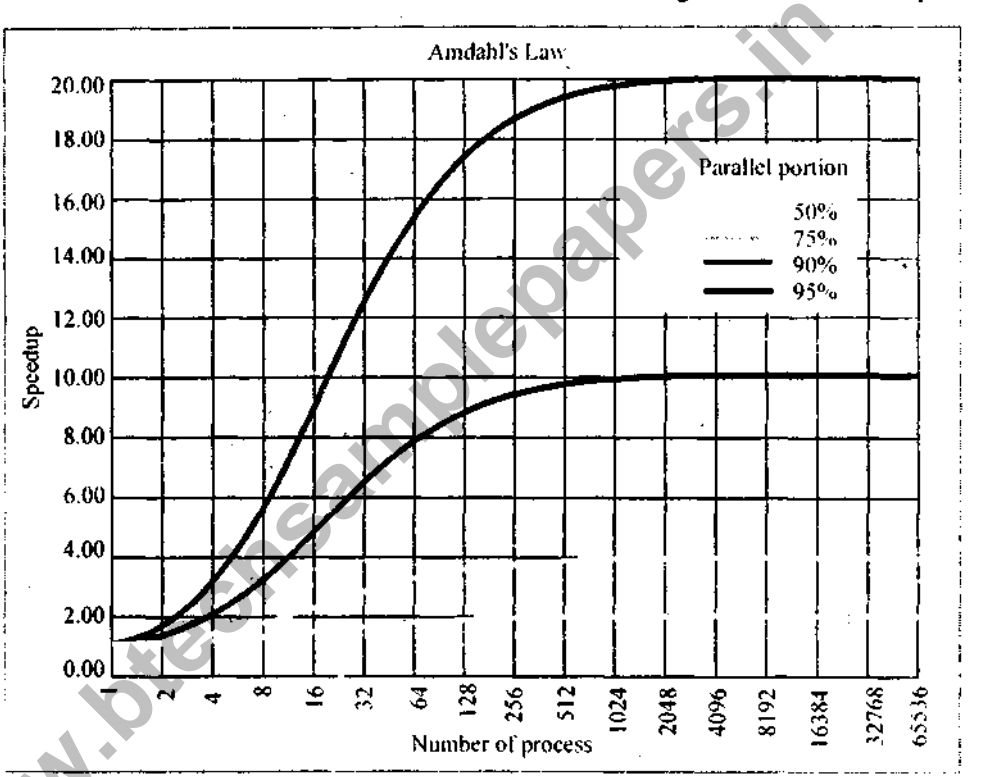
Ans. The degree of parallelism (DOP) is a metric which indicates how many operations can be or are being simultaneously executed by a computer. It is especially useful for describing the performance of parallel programs and multi-processor systems.

A program running on a parallel computer may utilize different numbers of processors at different times. For each time period, the number of processors used to execute a program is defined as the degree of parallelism. The plot of the DOP as a function of time for a given program is called the parallelism profile.

Maximum degree of parallelism = number of bits in a word X number of words processed in parallel

The speed-up of a program as a result of parallelization is governed by Amdahl's law.

Amdahl's law and Gustafson's law



A graphical representation of Amdahl's law. The speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used.

Optimally, the speed-up from parallelization would be linear-doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s.[11] It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. Any large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential)

parts. This relationship is given by the equation:

$$S = \frac{1}{1-P}$$

where S is the speed-up of the program (as a factor of its original sequential runtime), and P is the fraction that is parallelizable. If the sequential portion of a program is 10% of the runtime, we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."

Gustafson's law is another law in computer engineering, closely related to Amdahl's law. It can be formulated as:

Two independent part A B

Original process [redacted]

Make B 5× faster [redacted]

Make A 2× faster [redacted]

Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. With effort, a programmer may be able to make this part five times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A twice as fast. This will make the computation much faster than by optimizing part B, even though B got a greater speed-up (5× versus 2×).

$S(P) = P - \alpha(P - 1)$

Where P is the number of processors, S is the speed-up, and ? the non-parallelizable part of the process. Amdahl's law assumes a fixed-problem size and that the size of the sequential section is independent of the number of processors, whereas Gustafson's law does not make these assumptions.

Q. 1. (c) Explain Flynn's classification of computer architecture and how it is Feng's classification ?

Ans. Flynn's classification of architecture

1. SISD (single instruction stream, single data stream)
   - Corresponds to usual Von Neumann architecture.
   - single CPU
   - executes one instruction at a time (single instruction stream)
   - fetches/stores one data value at a time (single data stream)
2. SIMD (single instruction stream, multiple data stream)
   - executes one instructions at a time (single instruction stream)
   - same operation is performed on many data values at the same time (multiple data stream)
   - these are the so-called ' vector machines ', such as CDC 6600 / 7600 / Cyber machines
   - a vector operation with n elements can be performed in one instruction cycle on SIMD architecture
3. MISD (multiple instruction, single data stream)
   - multiple programs, operating on same data (performing different computations)
   - no MISD machines exist at this point
4. MIMD (multiple instruction stream, multiple data stream)
   - these are multiprocessor systems

- each processor can execute a different program on its own data
- hence, multiple instruction streams (programs) and multiple data streams

Both SIMD and MIMD are parallel processing architectures since the processors execute operations in parallel. They are, by default, multiprocessor architectures, which can be subdivided into two categories.

1. Global memory architectures
   - common, global memory is shared by all processors
2. Local memory architectures
   - one local memory per processor
   - may also have a global memory

Global memory MIMD architectures are also known as tightly-coupled multiprocessor systems; local memory MIMD systems are known as loosely-coupled multiprocessor systems.

Diagram comparing Classifications

Visually, these four architectures are shown below where each "PU" is a processing unit:

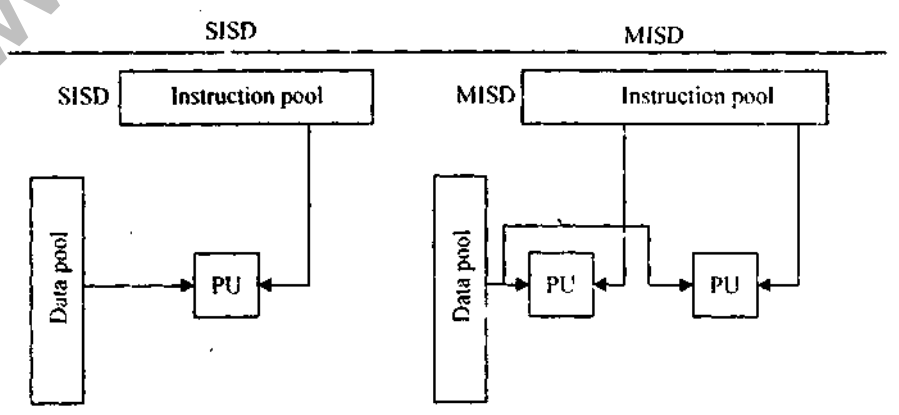
The four classifications defined by Flynn are based upon the number of concurrent instruction (or control) and data streams available in the architecture:

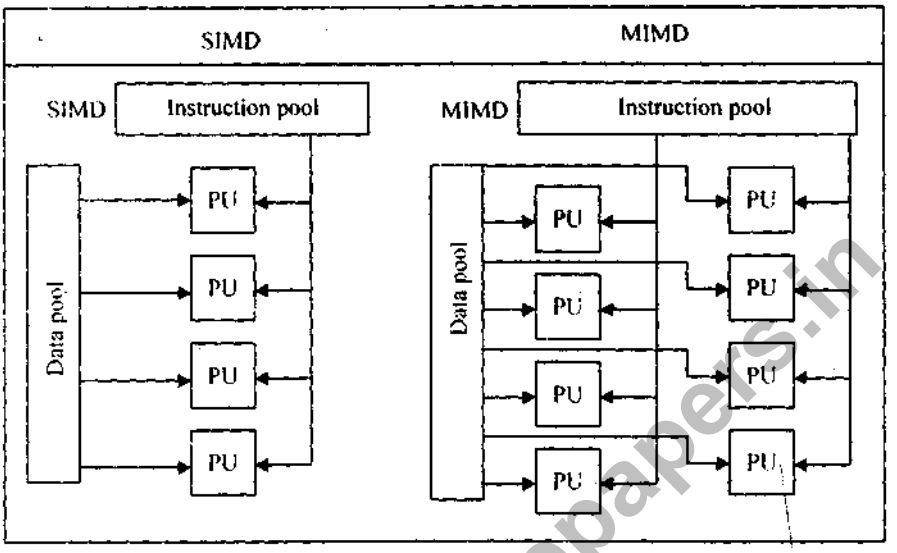
Single Instruction, Single Data stream (SISD) : A sequential computer which exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are the traditional uniprocessor machines like a PC or old mainframes.

Single Instruction, Multiple Data streams (SIMD) : A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or GPU.

Multiple Instructions, Single Data stream (MISD) : Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.

Multiple Instruction, Multiple Data streams (MIMD) : Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD





architectures; either exploiting a single shared memory space or a distributed memory space.

Q. 2. Attempt any two parts of the following :

Q. 2. (a) What do you understand by Pipelining ? Explain it. What are hazards that occur in pipelining in your opinion. Explain it.

Ans. Pipelining is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided in stages. Each stage completes a part of an instruction in parallel. The stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end. Pipelining does not decrease the time for individual instruction execution. Instead, it increases instruction throughput. The throughput of the instruction pipeline is determined by how often an instruction exits the pipeline.

Pipeline is broken into five stages with a set of flip flops between each stage.

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

Advantages of Pipelining :

1. The cycle time of the processor is reduced, thus increasing instruction issue-rate in most cases.

2. Some combinational circuits such as adders or multipliers can be made faster by adding more circuitry. If pipelining is used instead, it can save circuitry vs. a more complex combinational circuit.

In computer architecture, a hazard is a potential problem that can happen in a pipelined processor. It refers to the possibility of erroneous computation when a CPU tries to simultaneously execute multiple instructions which exhibit data dependence. There are typically three types of hazards: data hazards, structural hazards, and branching hazards (control hazards).Instructions in a pipelined processor are performed in several stages, so that at any given time several instructions are

being executed, and instructions may not be completed in the desired order.

A hazard occurs when two or more of these simultaneous (possibly out of order) instructions conflict.

Data hazards : Data hazards occur when data is modified. Ignoring potential data hazards can result in race conditions (sometimes known as race hazards). There are three situations a data hazard can occur in:

1. Read after Write (RAW) or True dependency : An operand is modified and read soon after. Because the first instruction may not have finished writing to the operand, the second instruction may use incorrect data.

2. Write after Read (WAR) or Anti dependency : Read an operand and write soon after to that same operand. Because the write may have finished before the read, the read instruction may incorrectly get the new written value.

3. Write after Write (WAW) or Output dependency : Two instructions that write to the same operand are performed. The first one issued may finish second, and therefore leave the operand with an incorrect data value.

RAW-Read After Write-A RAW Data Hazard refers to a situation where we refer to a result that has not yet been calculated, for example:

i1. R2 <- R1 + R3

i2. R4 <- R2 + R3

The 1st instruction is calculating a value to be saved in register 2, and the second is going to use this value to compute a result for register 4. However, in a pipeline, when we fetch the operands for the 2nd operation, the results from the 1st will not yet have been saved, and hence we have a data dependency.

We say that there is a data dependency with instruction 2, as it is dependent on the completion of instruction 1

WAR-Write After Read-A WAR Data Hazard represents a problem with concurrent execution, for example:

i1. R4 <- R1 + R3

i2. R3 <- R1 + R2

If we are in a situation that there is a chance that i2 may be completed before i1 (i.e. with concurrent execution) we must ensure that we do not store the result of register 3 before i1 has had a chance to fetch the operands.

WAW-Write After Write-A WAW Data Hazard is another situation which may occur in a Concurrent execution environment, for example:

i1. R2 <- R1 + R2

i2. R2 <- R4 x R7

We must delay the WB (Write Back) of i2 until the execution of i1

Structural hazards-A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A structural hazard might occur, for instance, if a program were to execute a branch instruction followed by a computation instruction. Because they are executed in parallel, and because branching is typically slow (requiring a comparison, program counter-related computation, and writing to registers), it is quite possible (depending on architecture) that the computation instruction and the branch instruction will both require the ALU (arithmetic logic unit) at the same time.

Branch (control) hazards-Branching hazards (also known as control hazards) occur when the processor is told to branch - i.e., if a certain condition is true, then jump from one part of the instruction stream to another - not

necessarily to the next instruction sequentially. In such a case, the processor cannot tell in advance whether it should process the next instruction (when it may instead have to move to a distant instruction).

Q. 2. (b) What do you understand by linear & non linear pipeline processors ? explain them.

Ans. A linear pipeline processor is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other. In modern computers, linear pipelines are applied for instruction execution, arithmetic computation and memory access operations.

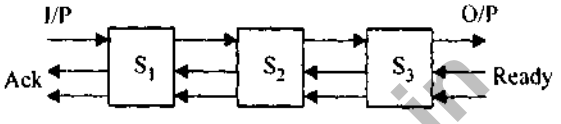
Asynchronous and Synchronous Models

A linear pipeline is constructed with k processing stages (segments). External inputs (operands) are fed into the pipeline at the first stage S1. The processed results are passed from stage Si to stage Si+1 for all i = 1, 2,...., k-1. The final result emerges from the pipeline at the last stage. Depending on the control of data flow along the pipeline, linear pipeline is modelled into two categories:

Asynchronous and synchronous.

Asynchronous Model : As shown in Fig, data flow between adjacent stages in asynchronous pipeline is controlled by a handshaking protocol. When stage Si is ready to transmit; it sends a ready signal to stage Si+1. After stage Si+1 receives the incoming data, it returns an acknowledge signal to Si.

Synchronous Model : The operands pass through all segments in a fixed sequence. Each segment consists of a combinational circuit Si that performs a sub operation over the data stream flowing through the pipe Isolating registers R (latches) are used to interface between stages and hold the intermediate results between the stages. Upon the arrival of a clock pulse, all registers transfer data to the next stage simultaneously.



Non linear pipeline

- Pipes can have branches and loops
- Allow reuse of expensive stages
- Allow different operations to go through the same pipe, sharing stages
- e.g., Floating point arithmetic
- Requires special scheduling of inputs to avoid conflicts

Q. 2. (c) Discuss memory hierarchy technology. Explain inclusion, coherence and locality properties.

Ans. The term memory hierarchy is used in the theory of computation when discussing performance issues in computer architectural design, algorithm predictions, and the lower level programming constructs such as involving locality of reference. A memory hierarchy' in computer storage distinguishes each level in the 'hierarchy' by response time. Since response time, complexity, and capacity are related, the levels may also be distinguished by the controlling technology.

The many trade-offs in designing for high performance will include the structure of the memory hierarchy, i.e. the size and technology of each component. So the various components can be viewed as forming a hierarchy of memories (m1,m2,...,mn) in which each member mi is in a sense subordinate to the next highest member mi-1 of the

hierarchy. To limit waiting by higher levels, a lower level will respond by filling a buffer and then signaling to activate the transfer.

There are four major storage levels.

1. Internal - Processor registers and cache.

2. Main - the system RAM and controller cards.

3. On-line mass storage - Secondary storage.

4. Off-line bulk storage - Tertiary and Off-line storage.

This is a most general memory hierarchy structuring. Many other structures are useful. For example, a paging algorithm may be considered as a level for virtual memory when designing computer architecture.

The memory hierarchy in most computers is:

- Processor registers - fastest possible access (usually 1 CPU cycle), only hundreds of bytes in size
- Level 1 (L1) cache - often accessed in just a few cycles, usually tens of kilobytes
- Level 2 (L2) cache - higher latency than L1 by 2× to 10×, often 512 KiB or more
- Level 3 (L3) cache - higher latency than L2, often 2048 KiB or more
- Main memory - may take hundreds of cycles, but can be multiple gigabytes. Access times may not be uniform, in the case of a NUMA machine.
- Disk storage - millions of cycles latency if not cached, but very large
- Tertiary storage - several seconds latency, can be huge

Note that the hobbyist who reads "L1 cache" in the computer specifications sheet is reading about the 'internal' memory hierarchy.

Most modern CPUs are so fast that for most program workloads, the bottleneck is the locality of reference of memory accesses and the efficiency of the caching and memory transfer between different levels of the hierarchy. As a result, the CPU spends much of its time idling, waiting for memory I/O to complete. This is sometimes called the space cost, as a larger memory object is more likely to overflow a small/fast level and require use of a larger/slower level.

Locality of reference: The locality of reference, also known as the locality principle, is the phenomenon, that the collection of the data locations referenced in a short period of time in a running computer, often consists of relatively well predictable clusters.

Important special cases of locality are temporal, spatial, and equidistant and branch locality.

- Temporal locality: if at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case it is common to make efforts to store a copy of the referenced data in special memory storage, which can be accessed faster. Temporal locality is a very special case of the spatial locality, namely when the prospective location is identical to the present location.

- Spatial locality: if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. There is a spatial proximity between the memory locations, referenced at almost the same time. In this case it is common to make efforts to guess, how big neighborhood around the current reference is worthwhile to prepare for faster access.
- Equidistant locality: it is halfway between the spatial locality and the branch locality. Consider a loop accessing locations in an equidistant pattern, i.e. the path in the spatial-temporal coordinate space is a dotted line. In this case, a simple linear function can predict which location will be accessed in the near future.
- Branch locality: if there are only few amount of possible alternatives for the prospective part of the path in the spatial-temporal coordinate space. This is the case when an instruction loop has a simple structure, or the possible outcome of a small system of conditional branching instructions is restricted to a small set of possibilities. Branch locality is typically not a spatial locality since the few possibilities can be located far away from each other.

In order to make benefit from the very frequently occurring temporal and spatial kind of locality, most of the information storage systems are hierarchical; see below. The equidistant locality is usually supported by the diverse nontrivial increment instructions of the processors. For the case of branch locality, the contemporary processors have sophisticated branch predictors, and on the base of this prediction the memory manager of the processor tries to collect and preprocess the data of the plausible alternatives.

Memory coherence is an issue that affects the design of computer systems in which two or more processors share a common area of memory.

A computer system does useful work by reading data from permanent storage into memory, performing some operation on that data (such as adding up two numbers) and then storing the result back to permanent storage. In a uniprocessor system (such as a simple personal computer) there is only one processor doing all the work, and therefore only one processor that can read or write the data values. Moreover a simple uniprocessor can only do one thing at a time, so when a value in storage is changed, all subsequent read operations will see the updated value.

In multiprocessor systems however there are two or more processors working at the same time, so there is the possibility that the processors will all want to process the same value at the same time. Provided none of the processors updates the value then they can share it indefinitely, but as soon as one updates the value, the others will be working on an out-of-date copy. Some scheme is required to notify all processors of changes to shared values; such a scheme is known as a "memory coherence protocol". The coherency is not enough to be able to write concurrent programs and thus comes along a cache consistency model.

Various protocols have been devised for maintaining memory coherency, such as the

MSI protocol and its derivatives MESI, MOSI and MOESI. Most of the cache protocols in multiprocessors support a sequential consistency model.

Q. 3. Attemp any two parts of the following :

Q. 3. (a) Explain about array computer and Pipeline computers.

Ans. In a pipeline computer, a store sub-instruction address is stored into a most recent address register as well as into an address buffer in response to a store request and a load sub-instruction address is supplied to a main memory in response to a load sub-instruction. When a store sub-instruction data is stored into a buffer following the storage of the store sub-instruction address into the most recent address register, the contents of the address buffer and the data buffer are transferred to a location of the main memory specified by the sub-instruction address. Main memory data is retrieved from a location specified by an associated load sub-instruction address. A match or a mismatch is detected between the address in the most recent address register and an address generated in response to a subsequent load sub-instruction. If the latter occurs just prior to the storage of store sub-instruction data into the data buffer and if it accesses the same data as the store sub-instruction data, a match will be detected and data from an arithmetic logic unit is stored into a replay data register and returned to the request source. If a mismatch is detected and lata read out of the main memory is passed through the reply data register to the request source.

Array computers : A vector processor, or array processor, is a CPU design wherein the instruction set includes operations that can perform mathematical operations on multiple data elements simultaneously. This is in contrast to a scalar processor, which handles one element at a time using multiple instructions. The vast majority of CPUs are scalar (or close to it). Vector processors were common in the scientific computing area, where they formed the basis of most supercomputers through the 1980s and into the 1990s, but general increases in performance and processor design saw the near-disappearance of the vector processor as a general-purpose CPU

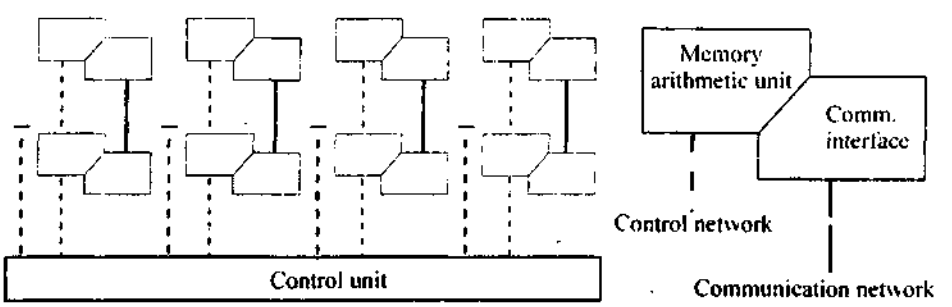
SIMD stands for Single Instruction, Multiple Data. The idea is that you can move an entire array around (and pipeline is processing) with just one CPU instruction -- otherwise, you'd have to queue up individual instructions for each element in the array.

Q. 3. (b)Explain the following :

(i) Neural architecture

(ii) Associative procesors

(iii) Systolic architecture.

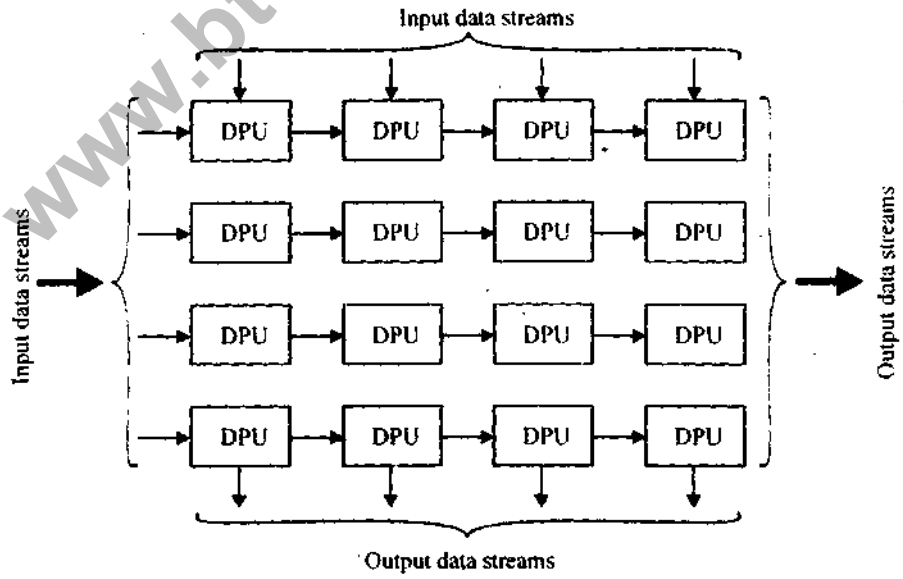


Ans. In computer architecture, a systolic array is a pipe network arrangement of processing units called cells. It is a specialized form of parallel computing, where cells (i.e. processors), compute data and store it independently of each other.

A systolic array is composed of matrix-like rows of data processing units called cells. Data processing units DPUs are similar to central processing units (CPUs),(except for the usual lack of a program counter[1], since operation is transport-triggered, i.e., by the arrival of a data object). Each cell shares the information with its neighbors immediately after processing. The systolic array is often rectangular where data flows across the array between neighbor DPUs, often with different data flowing in different directions. The data streams entering and leaving the ports of the array are generated by auto-sequencing memory units, ASMs. Each ASM includes a data counter. In embedded systems a data stream may also be input from and/or output to an external source.

An example of a systolic algorithm might be designed for matrix multiplication. One matrix is fed in a row at a time from the top of the array and is passed down the array; the other matrix is fed in a column at a time from the left hand side of the array and passes from left to right. Dummy values are then passed in until each processor has seen one whole row and one whole column. At this point, the result of the multiplication is stored in the array and can now be output a row or a column at a time, flowing down or across the array.

Systolic arrays are arrays of DPUs which are connected to a small number of nearest neighbor DPUs in a mesh-like topology. DPUs perform a sequence of operations on data that flows between them. Because the traditional systolic array synthesis methods have been practiced by algebraic algorithms, only uniform arrays with only linear pipes can be obtained, so that the architectures are the same in all DPUs. The consequence is, that only applications with regular data dependencies can be implemented on classical systolic arrays. Like SIMD machines, clocked systolic arrays compute in "lock-step" with each processor undertaking alternate compute |



communicate phases. But systolic arrays with asynchronous handshake between DPUs are called wave front arrays. One well-known systolic array is Carnegie Mellon University's iWarp processor, which has been manufactured by Intel. An iWarp system has a linear array processor connected by data buses going in both directions.

Associative processors : A processor having parallel processing capabilities by virtue of the parallel memory manipulation properties of associative memory. The ability to interrogate and write to selected bits in each word of associative memory in parallel makes possible word-parallel bit-serial operations, which are efficient for the manipulation of large data sets.

A memory that is capable of determining whether a given datum - the search word - is contained in one of its addresses or locations. This may be accomplished by a number of mechanisms. In some cases parallel combinational logic is applied at each word in the memory and a test is made simultaneously for coincidence with the search word. In other cases the search word and all of the words in the memory are shifted serially in synchronism; a single bit of the search word is then compared to the same bit of all of the memory words using as many single-bit coincidence circuits as there are words in the memory. Amplifications of the associative memory technique allow for masking the search word or requiring only a "close" match as opposed to an exact match. Small parallel associative memories are used in cache memory and virtual memory mapping applications.

Since parallel operations on many words are expensive (in hardware), a variety of stratagems are used to approximate associative memory operation without actually carrying out the full test described here. One of these uses hashing to generate a "best guess" for a conventional address followed by a test of the contents of that address. Some associative memories have been built to be accessed conventionally (by words in parallel) and as serial- comparison associative memories: these have been called orthogonal memories.

Systolic architecture : Systolic architectures are designed by using linear mapping techniques on regular dependence graphs. Systolic architectures have a space-time representation where each node is mapped to a certain processing element (PE) and is scheduled at a particular time instance. Systolic design methodology maps an N-dimensional DG to a lower dimensional systolic architecture.

A systolic architecture has the following characteristics

A massive and non-centralised parallelism

Local communications

Synchronous evaluation

Example of systolic network : Linear network a systolic array is a pipe network arrangement of processing units called cells. It is a specialized form of parallel computing, where cells (i.e. processors), compute data and store it independently of each other.

The super systolic array is a generalization of the systolic array. Because the classical synthesis methods (algebraic, i. e. projection-based synthesis), yielding only uniform DPU arrays permitting only linear pipes, systolic arrays could be used only to implement applications with regular data dependencies. By using simulated annealing instead, Rainer Kress has introduced the generalized systolic array: the super systolic

array. Its application is not restricted to applications with regular data dependencies.

Q. 3. (c) Explain the structural and operational differences between register-to=register and memory-to-memory architecture in building multi-pipelined super computers for vector processing. Comment on the advantages and compared with the use of pipelined super computers for vector processing.

Ans. An application that may take advantage of SIMD is one where the same value is being added (or subtracted) to a large number of data points, a common operation in many multimedia applications. One example would be changing the brightness of an image. Each pixel of an image consists of three values for the brightness of the red, green and blue portions of the color. To change the brightness, the R G and B values are read from memory, a value is added (or subtracted) from them, and the resulting values are written back out to memory.

With a SIMD processor there are two improvements to this process. For one the data is understood to be in blocks, and a number of values can be loaded all at once. Instead of a series of instructions saying "get this pixel, now get the next pixel", a SIMD processor will have a single instruction that effectively says "get lots of pixels" ("lots" is a number that varies from design to design). For a variety of reasons, this can take much less time than "getting" each pixel individually, likes with traditional CPU design.

Another advantage is that SIMD systems typically include only those instructions that can be applied to all of the data in one operation. In other words, if the SIMD system works by loading up eight data points at once, the add operation being applied to the data will happen to all eight values at the same time. Although the same is true for any superscalar processor design, the level of parallelism in a SIMD system is typically much higher.

Disadvantages

- Not all algorithms can be vectorized. For example, a flow-control-heavy task like code parsing wouldn't benefit from SIMD.
- Currently, implementing an algorithm with SIMD instructions usually requires human labor; most compilers don't generate SIMD instructions from a typical C program, for instance. Vectorization in compilers is an active area of computer science research. (Compare vector processing.)
- Programming with particular SIMD instruction sets can involve numerous low-level challenges.
  - o SSE has restrictions on data alignment; programmers familiar with the x86 architecture may not expect this.
  - o Gathering data into SIMD registers and scattering it to the correct destination locations is tricky and can be inefficient.
  - o Specific instructions like rotations or three-operand addition aren't in some SIMD instruction sets.
  - o Instruction sets are architecture-specific: old processors and non-x86 processors lack SSE entirely, for instance, so programmers must provide non-vectorized implementations (or different

vectorized implementations) for them. Similarly, the next-generation instruction sets from Intel and AMD will be incompatible with each other (see SSE5 and AVX).

o The early MMX instruction set shared a register file with the floating-point stack, which caused inefficiencies when mixing floating-point and MMX code. However, SSE2 corrects this.

Q. 4. Attempt any two parts of the following :

Q. 4. (a) What do you understand by PRAM Algorithms ? Discus and explain with suitable example about the PRAM algorithm for merging two sorted lists.

Ans. Parallel Random Access Machine (PRAM) the PRAM model is an extension of the familiar RAM model of sequential computation that is used in algorithm analysis. Synchronous PRAM which is defined as follows.

1. There are p processors connected to a single shared memory.

2. Each processor has a unique index 1 _ i _ p called the processor id.

3. A single program is executed in single-instruction stream, multiple-data stream (SIMD) fashion. Each instruction in the instruction stream is carried out by all processors simultaneously and requires unit time, regardless of the number of processors.

4. Each processor has a flag that controls whether it is active in the execution of an instruction. Inactive processors do not participate in the execution of instructions, except for instructions that reset the flag.

The operation of a synchronous PRAM can result in simultaneous access by multiple processors to the same location in shared memory. There are several variants of our PRAM model, depending on whether such simultaneous access is permitted (concurrent access) or prohibited (exclusive access). As accesses can be reads or writes, we have the following four possibilities:

1. Exclusive Read Exclusive Write (EREW) : This PRAM variant does not allow any kind of simultaneous access to a single memory location. All correct programs for such a PRAM must insure that no two processors access a common memory location in the same time unit.

2. Concurrent Read Exclusive Write (CREW) : This PRAM variant allows concurrent reads but not concurrent writes to shared memory locations. All processors concurrently reading a common memory location obtain the same value.

3. Exclusive Read Concurrent Write (ERCW) : This PRAM variant allows concurrent writes but not concurrent reads to shared memory locations. This variant is generally not considered independently, but is subsumed within the next variant.

4. Concurrent Read Concurrent Write (CRCW) : This PRAM variant allows both concurrent reads and concurrent writes to shared memory locations. There are several sub-variants within this variant, depending on how concurrent writes are resolved.

(a) Common CRCW: This model allows concurrent writes if and only if all the processors are attempting to write the same value (which becomes the value stored).

(b) Arbitrary CRCW: In this model, a value arbitrarily chosen from the values written to the common memory location is stored.

(c) Priority CRCW: In this model, the value written by the processor with the

minimum processor id writing to the common memory location is stored.

(d) Combining CRCW: In this model, the value stored is a combination (usually by an associative and commutative

We study PRAM algorithms for several reasons.

1. There is a well-developed body of literature on the design of PRAM algorithms and the complexity of such algorithms.

2. The PRAM model focuses exclusively on concurrency issues and explicitly ignores issues of synchronization and communication. It thus serves as a baseline model of concurrency. In other words, if you can't get a good parallel algorithm on the PRAM model, you're not going to get a good parallel algorithm in the real world.

3. The model is explicit: we have to specify the operations performed at each step, and the scheduling of operations on processors.

4. It is a robust design paradigm. Many algorithms for other models (such as the network model) can be derived directly from PRAM algorithms.

Q. 4. (b) Prove that the best parallel algorithm written for an n-processor EREW PRAM model can be no algorithm for a (CRCW) model of PRAM having the same number of processors.

Ans. The lists having $n/2$ elements each can be merged.

MERGE LISTS (CREW PRAM) :

Given : Two stored lists of $n/2$ elements each, stored in

$A(1)$ .....$A(n/2)$ & $A(n/2+1)$ .....$A(n)$

The two lists & their union have disjoint value.

Final condition : Merged list in location $A(1)$ .....$A(n)$

Global : $A(1-n)$

local X, low high, index

begin

Spawn $(P_1, P_2, ...P_n)$

for all P_i where $1 \le i \le n$ do

{Each processor its bounds for binary search}

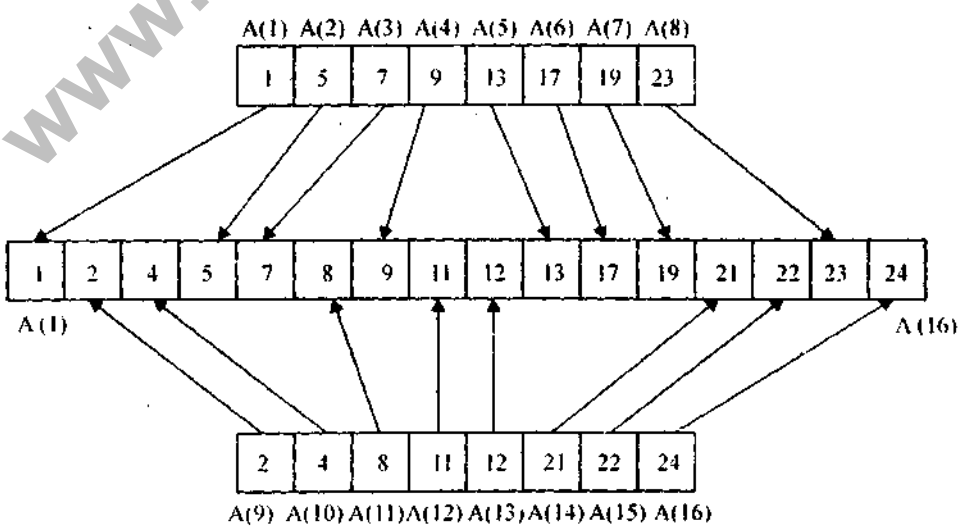
if $i \le n/2$ then

low $\leftarrow (n/2) + 1$

high $\leftarrow$

else

low $\leftarrow 1$



```
        high ← n/2
    end if
        {Each processor performs
                        binary search}
        x ← A(i)
        repeat
        index ← [(low + high)/2]
        if x < A (index) then
                high ← index – 1
        else
                low ← index + 1
        ending
    until low > high
        {Put value in correct position
              on merged list}
    A [High + i – n/2] ← x
    end for
    end.
```

PRAM algorithm to merge two sorted lists.

One optimal PRAM algorithm creates the merged list one element at a time. It requires at most $n - 1$ comparisons to merge two sorted list of $n/2$ elements its time complexity is $\ominus(n)$.

The (CREW) PRAM algorithm perform the same task in Q(log time).

Sum (EREW PRAM)

Initial condition : List of $n \geq 1$ elements stored in $A[0,.....(n-1)]$

final condition : Sum of elements stored in $A(0)$.

Global variable : $n, A[0.....(n-1)], j$

```
begin
    Spawn (P0, P1, P2,.....P⌈n/2⌉–1)
  for all Pi where 0 ≤ i ≤ ⌈n/2⌉– 1 do
      for j ← 0 to ⌈log n –⌉– 1 do
          if i modulo 2^j = 0 & 2i + 2^j < n
                                then
          A [2i] ← A [2i] + A [2i + 2^j]
        end if
      end for
    end for   EREW PRAM algorithm to
              sum n elements using ⌈n/2⌉
              processor.
end
```

Q. 4. (c) Explain the following terms related to shared-variable programming on multiprocessors :

Multiprocessing in MIMD mode

Ans. MIMD multiprocessing architecture is suitable for a wide variety of tasks in which completely independent and parallel execution of instructions touching different sets of data can be put to productive use. For this reason, and because it is easy to implement, MIMD predominates in multiprocessing.

Processing is divided into multiple threads, each with its own hardware processor state, within a single software-defined process or within multiple processes. Insofar as a system has multiple threads awaiting dispatch (either system or user threads), this architecture makes good use of hardware resources.

MIMD does raise issues of deadlock and resource contention, however, since threads may collide in their access to resources in an unpredictable way that is difficult to manage efficiently. MIMD requires special coding in the operating system of a computer but does not require application changes unless the programs themselves use multiple threads (MIMD is transparent to single-threaded programs under most operating systems, if the programs do not voluntarily relinquish control to the OS). Both system and user software may need to use software constructs such as semaphores (also called locks or gates) to prevent one thread from interfering with another if they should happen to cross paths in

referencing the same data. This gating or locking process increases code complexity, lowers performance, and greatly increases the amount of testing required, although not usually enough to negate the advantages of multiprocessing.

Similar conflicts can arise at the hardware level between processors (cache contention and corruption, for example), and must usually be resolved in hardware, or with a combination of software and hardware (e.g., cache-clear instructions).

Q. 5. Write short notes on any two :

(a) Conditional compilation

(b) Run-time library routines

(c) Master and synchronization constructions.

Ans. (a) Conditional compilation of intermediate language code based on current environment includes loading intermediate language code on a device. Portions of the intermediate language code are conditionally just-in-time compiled based on a current environment of the device. In accordance with certain aspects, intermediate language code is loaded on a device and a current environment of the device is identified. The intermediate language code is modified based on the current environment, and portions of the modified intermediate language code are just-in-time compiled as needed when running the intermediate language code.

The OpenMP API defines the preprocessor symbol _OPENMP to be used for conditional compilation. In addition, OpenMP FORTRAN API accepts a conditional compilation sentinel. Some rules for conditional compilation

1. If you are continuing a conditional compilation line, the conditional compilation sentinel must appear on at least one of the continuation lines or on the initial line

2. A conditional compilation sentinel must not contain embedded white space.

3. A valid XL FORTRAN source line must follow the conditional compilation sentinel.

4.A conditional compilation line can contain the EJECT, INCLUDE or no comment directives.

(b) Run time library routines

Followings are the some examples of run time library routines. These are used in the OpenMP programs.

OMP_SET_NUM_THREADS

Sets the number of threads that will be used in the next parallel region. Must be a positive integer.

#include <omp.h>

Void omp_set_num_threads (int num_threads)

This routine can only be called from the serial portions of the code

This call has precedence over the OMP_NUM_THREADS environment variable

OMP_GET_NUM_THREADS -

Returns the number of threads that are currently in the team executing the parallel region from which it is called.

#include <omp.h>

int omp_set_num_threads(void)

The default number of threads is implementation dependent.

OMP_GET_MAX_THREADS

Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.

#include <omp.h>

int omp_get_max_threads(void)

May be called from both serial and parallel regions of code.

OMP_GET_THREAD_NUM

Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0

```
#include <omp.h>
int omp_get_thread_num(void)
```

If called from a nested parallel region, or a serial region, this function will return 0.

OMP_GET_THREAD_LIMIT

New with OpenMP 3.0. Returns the maximum number of OpenMP threads available to a program.

```
#include <omp.h>
int omp_get_thread_limit (void)
```

OMP_GET_NUM_PROCS

Returns the number of processors that are available to the program.

```
#include <omp.h>
int omp_get_num_procs(void)
```

OMP_IN_PARALLEL

May be called to determine if the section of code which is executing is parallel or not.

```
#include <omp.h>
int omp_in_parallel(void)
```

(c) Synchronization Constructs

Consider a simple example where two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0):

```
THREAD 1:
Increment(x)
{
   x = x + 1;
}
THREAD 1:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)

THREAD 2:
Increment(x)
{
   x =x+1;
}
THREAD 2:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)
```

One possible execution sequence:

1. Thread 1 loads the value of x into register A.
2. Thread 2 loads the value of x into register A.
3. Thread 1 adds 1 to register A
4. Thread 2 adds 1 to register A
5. Thread 1 stores register A at location x
6. Thread 2 stores register A at location x

The resultant value of x will be 1, not 2 as it should be.

To avoid a situation like this, the incrementation of x must be synchronized between the two threads to insure that the correct result is produced.

OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

Synchronization Constructs

MASTER Directive

- The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code
- There is no implied barrier associated with this directive

```
#pragma omp master newline
  structured_block
```

It is illegal to branch into or out of MASTER block.

CRITICAL Directive

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

#pragma omp critical [name] newline
structured_block

If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

The optional name enables multiple different CRITICAL regions to exist:

- Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
- All CRITICAL sections which are unnamed, are treated as the same section.

Example : CRITICAL Construct

- All threads in the team will attempt to execute in parallel, however, because of the CRITICAL construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time

```
#include <omp.h>
main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
 {
 #pragma omp critical
 x = x + 1;
 } /* end of parallel section */
}
```

BARRIER Directive

- The BARRIER directive synchronizes all threads in the team.
- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

#pragma omp barrier newline

Restrictions :

- All threads in a team (or none) must execute the BARRIER region.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

ATOMIC Directive

- The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section.

#pragma omp atomic newline
statement_expression

Restrictions :

- The directive applies only to a single, immediately following statement
- An atomic statement must follow a specific syntax. See the most recent OpenMP specs for this.

FLUSH Directive

- The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

- There is a fair amount of discussion on this directive within OpenMP circles that you may wish to consult for more information.

#pragma omp flush (list) newline

- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, note that the pointer itself is flushed, not the object it points to.
- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; ie. compilers must restore values from registers to memory, hardware might need to flush write buffers, etc
- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

ORDERED Directive

- The ORDERED directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.
- Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
- Used within a DO / for loop with an ORDERED clause
- The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.
- A loop which contains an ORDERED directive must be a loop with an ORDERED clause.