Instruction Level parallelism – Parallel processing challenges – flynns Classification – Hardware Multithreading – Multicore Processors

UNIT 4

II-year IT (CA) ① [illegible]

Instruction-Level Parallelism and Dynamic Exploitation

3.1 Instruction-Level Parallelism: Concepts and Challenges:

Instruction-level parallelism (ILP) is the potential overlap the execution of instructions using pipeline concept to improve performance of the system. The various techniques that are used to increase amount of parallelism are reduces the impact of data and control hazards and increases processor ability to exploit parallelism

There are two approaches to exploiting ILP.

1. Static Technique – Software Dependent
2. Dynamic Technique – Hardware Dependent

The static technique is compiler-intensive approach, which have broader adoption in the embedded market than the desktop or server markets, the new IA-64 architecture and Intel's Itanium, use this more static approach.

The dynamic technique is hardware intensive approach, which dominate the desktop and server markets and are used in a wide range of processors, including: the Pentium III and 4, the Althon, the MIPS R10000/12000, the Sun ultraSPARC III, the Power-PC 603, G3, and G4, and the Alpha 21264.

The simplest and most common way to increase the amount of parallelism is loop-level parallelism. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:

for (i=1;i<=1000; i=i+1) x[i] = x[i] + y[i];

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are a number of techniques for converting such loop-level parallelism into instruction-level parallelism are basically work by unrolling the loop either statically by the compiler or dynamically by the hardware. An important alternative method for exploiting loop-level parallelism is the use of vector instructions.

Data Dependence and Hazards:

To exploit instruction-level parallelism, determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls. If two instructions are dependent they are not parallel and must be executed in order.

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

Data Dependences:

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j, or
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
Loop: LW F0,0($S1) ; F0=array element
ADD.D F4,F0,F2 ; add scalar in F2
S.D F4,0($S1) ;store result
DADDUI $S1,$S1,#-8 ;decrement pointer 8 bytes
 BNE $S1,$S2,LOOP ; branch $S1!=zero
```

The data dependences in this code sequence involve both floating point data:

```
Loop: L.D F0,0(R1) ;F0=array element
ADD.D F4,F0,F2 ;add scalar in F2 S.D F4,0(R1) ;store result
```

and integer data:

```
        DADDIU R1,R1,-8 ;decrement pointer ;8 bytes (per DW)
        BNE R1,R2,Loop ; branch R1!=zero
```

Both of the above dependent sequences, as shown by the arrows, with each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. Dependences are a property of programs. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the pipeline organization. This difference is critical to understanding how instruction-level parallelism can be exploited.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. The importance of the data dependences is that a dependence (1) indicates the possibility of a hazard, (2) determines the order in which results must be calculated, and (3) sets an upper bound on how much parallelism can possibly be exploited. Such limits are explored in section 3.8.

Name Dependences

The name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

There are two types of name dependences between an instruction i that precedes instruction j in program order:

An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.

add $s4, $s1, $s2 → read
Sub $s2, $s5, $S1 → write

- An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

add $s1, $s2, $s3 } both write to same Register
Sub $s1, $s5, $s6.

Both anti-dependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Register Renaming :-

add \$s1 , \$s2 , \$s3
sub \$s1 , \$s4 , \$s5

(\$s1 is renamed to \$s6) ⟹ add \$s6 , \$s2 , \$s3
add \$s1 , \$s4 , \$s5

Control Dependences:

A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment:

```
if p1 { S1;
};
if p2 { S2;
}
```

S1 is control dependent on p1, and S2is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is **no longer controlled** by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the if-statement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is **controlled** by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

Control dependence is preserved by two properties in a simple pipeline, **First- Exception Behaviour** Preserving the *exception behavior* means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.. **Second, - Data flow** The data flow is the actual flow of data values among instructions that produce results and those that consume the.

limitations of ILP.

3

3.8. Studies of the Limitations of ILP

The Hardware Model

An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. *Register renaming*—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.
2. *Branch prediction*—Branch prediction is perfect. All conditional branches are predicted exactly.
3. *Jump prediction*—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.

4. *Memory-address alias analysis*—All memory addresses are known exactly and a load can be moved before a store provided that the addresses are not identical.

Assumptions 2 and 3 eliminate *all* control dependences. Likewise, assumptions 1 and 4 eliminate *all but the true* data dependences. Together, these four assumptions mean that *any* instruction in the ~~of the~~ program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends.

Limitations on the Window Size and Maximum Issue Count

A dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor. consider what the perfect processor must do:

1. Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly.
2. Rename all register uses to avoid WAR and WAW hazards.
3. Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly.
4. Determine if any memory dependences exist among the issuing instructions and handle them appropriately.
5. Provide enough replicated functional units to allow all the ready instructions to issue.

Obviously, this analysis is quite complicated. For example, to determine whether *n* issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires

$$2n-2+2n-4+\ldots\ldots..+2 = 2\Sigma_{i=1}^{n-1} i = [2\,(n-1)n]/2 = n^2 -n$$

comparisons. Thus, to detect dependences among the next 2000 instructions—the default size we assume in several figures—requires almost *four million* comparisons! Even issuing only 50 instructions requires 2450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once.

In existing and near-term processors, the costs are not quite so high, since we need only detect dependence pairs and the limited number of registers allows different solutions. Furthermore, in a real processor, issue occurs in-order and dependent instructions are handled by a renaming process that accommodates dependent renaming in one clock. Once instructions are issued, the detection of dependences is handled in a distributed fashion by the reservation stations or scoreboard.

The set of instructions that are examined for simultaneous execution is called the *window*. Each instruction in the window must be kept in the processor and the number of comparisons required every clock is equal to the maximum completion rate times the window size times the number of operands per instruction (today typically 6 x 80 x 2= 960), since every pending instruction must look at every completing instruction for either of its operands. Thus, the total window size is limited by the required storage, the comparisons, and a limited issue rate, which makes larger window less helpful

The window size directly limits the number of instructions that begin execution in a given cycle.

The Effects of Realistic Branch and Jump Prediction :

Our ideal processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed. Our data is for several different branch-prediction schemes varying from perfect to no predictor. We assume a separate predictor is used for jumps. Jump predictors are important primarily with the most accurate branch predictors, since the branch frequency is higher and the accuracy of the branch predictors dominates

The five levels of branch prediction shown in these figures are

1. *Perfect*—All branches and jumps are perfectly predicted at the start of execution.
2. *Tournament-based branch predictor*—The prediction scheme uses a correlating two-bit predictor and a noncorrelating two-bit predictor together with a selector, which chooses the best predictor for each branch.
3. *Standard two-bit predictor with 512 two-bit entries*—In addition, we assume a 16-entry buffer to predict returns.
4. *Static*—A static predictor uses the profile history of the program and predicts that the branch is always taken or always not taken based on the profile.
5. *None*—No branch prediction is used, though jumps are still predicted. Parallelism is largely limited to within a basic block.

The Effects of Imperfect Alias Analysis

Our optimal model assumes that it can perfectly analyze all memory dependences, as well as eliminate all register name dependences. Of course, perfect alias analysis is not possible in practice: The analysis cannot be perfect at compile time, and it requires a potentially unbounded number of comparisons at runtime

the impact of three other models of memory alias analysis, in addition to perfect analysis.
The three models are:

1 *Global/stack perfect*—This model does perfect predictions for global and stack references and assumes all heap references conflict. This model represents an idealized version of the best compiler-based analysis schemes currently in production. Recent and ongoing research on alias analysis for pointers should improve the handling of pointers to the heap in the future.

2 *Inspection*—This model examines the accesses to see if they can be determined not to interfere at compile time. For example, if an access uses R10 as a base register with an offset of 20, then another access that uses R10 as a base register with an offset of 100 cannot interfere. In addition, addresses based on registers that point to different allocation areas (such as the global area and the stack area) are assumed never to alias. This analysis is similar to that performed by many existing commercial compilers, though newer compilers can do better, at least for loop-oriented programs.

3. *None*—All memory references are assumed to conflict.

The dynamically scheduled processors rely on dynamic memory disambiguation and are limited by three factors:

1 To implement perfect dynamic disambiguation for a given load, we must know the memory addresses of all earlier stores that not yet committed, since a load may have a dependence through memory on a store. One technique for reducing this limitation on in-order address calculation is memory address speculation. With memory address speculation, the processor either assumes that no such memory dependences exist or uses a hardware prediction mechanism to predict if a dependence exists, stalling the load if a dependence is predicted. Of course, the processor can be wrong about the absence of the dependence, so we need a mechanism to discover if a dependence truly exists and to recover if so. To discover if a dependence exists, the processor examines the destination address of each completing store that is earlier in program order than the given load. If a dependence that should have been enforced occurs, the processor uses the speculative restart mechanism to redo the load and the following instructions. (We will see how this type of address speculation can be supported with instruction set extensions in the next chapter.)

2 Only a small number of memory references can be disambiguated per clock cycle.

3 The number of the load/store buffers determines how much earlier or later in the instruction stream a load or store may be moved.

Both the number of simultaneous disambiguations and the number of the load/ store buffers will affect the clock cycle time.

FLYNN'S CLASSIFICATION

This classification was first studied and proposed by Michael Flynn in 1972. Flynn did not consider the machine architecture for classification of parallel computers; he introduced the concept of instruction and data streams for categorizing of computers. All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn's classification. Since, this classification is based on instruction and data streams, first we need to understand how the instruction cycle works.

2.3.1 Instruction Cycle

The instruction cycle consists of a sequence of steps needed for the execution of an instruction in a program. A typical instruction in a program is composed of two parts: Opcode and Operand. The Operand part specifies the data on which the specified operation is to be done. (See Figure 1). The Operand part is divided into two parts: addressing mode and the Operand.

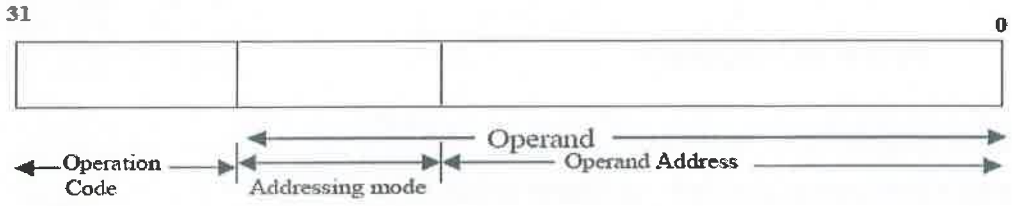


Figure 1: Opcode and Operand

Instruction Stream and Data Stream

The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called **instruction stream.** Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called **data stream**. These two types of streams are shown in Figure 3.

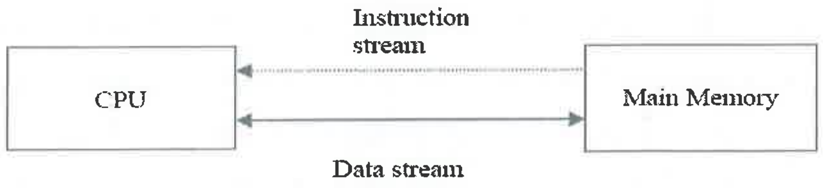


Figure 3: Instruction and data stream

Flynn's Classification

Flynn's classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let I_s and D_s are minimum number of streams flowing at any point in the execution, then the computer organisation can be categorized as follows:

1) **Single Instruction and Single Data stream (SISD)**

In this organisation, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit as shown in Figure 4. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organisation is depicted in the diagram

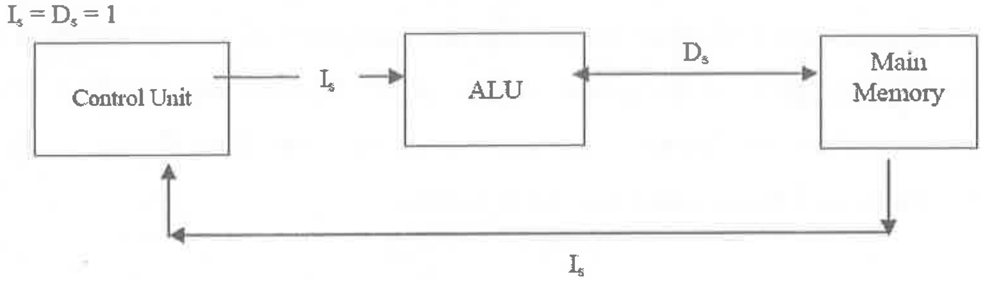


Figure 4: SISD Organisation

Examples of SISD machines include:

- CDC 6600 which is unpipelined but has multiple functional units.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing

2) **Single Instruction and Multiple Data stream (SIMD)**

In this organisation, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory as shown in Figure 5. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous. Examples of SIMD organisation are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).

This type of computer organisation is denoted as

$I_s = 1$

$D_s > 1$

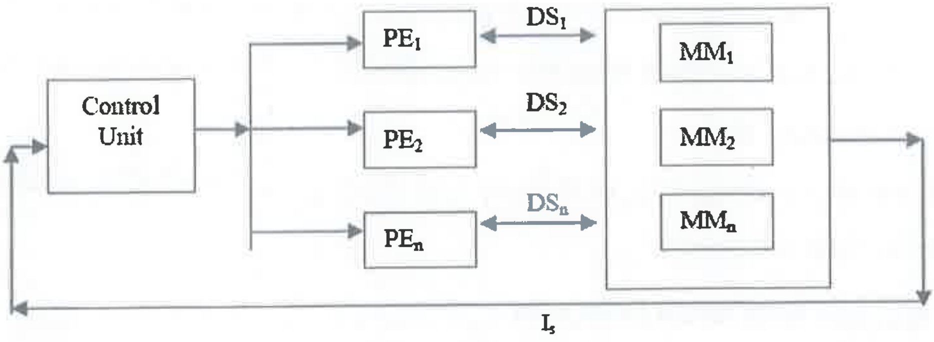


Figure 5: SIMD Organisation

3) Multiple Instruction and Single Data stream (MISD)

In this organization, multiple processing elements are organised under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification. All processing elements are interacting with the common shared memory for the organisation of single data stream as shown in Figure 6. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.

This type of computer organisation is denoted as:

$I_s > 1$
$D_s = 1$

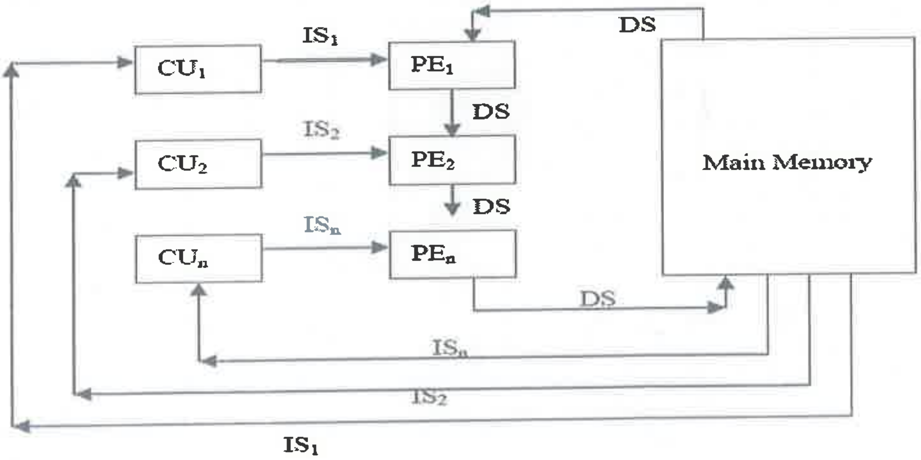


Figure 6: MISD Organisation

This classification is not popular in commercial machines as the concept of single data strea' .s executing on multiple processors is rarely applied. But for the specialized applications, MISD organisation can be very helpful. For example, Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data. This is also known as N-version programming. All these redundant data

are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

4) Multiple Instruction and Multiple Data stream (MIMD)

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams . Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory as shown in Figure 7. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organisation is said to be a Parallel computer. All multiprocessor systems fall under this classification. Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m*, BBN Butterfly, Meiko Compu:ing Surface (CS-1), FPS T/40000, iPSC.

This type of computer organisation is denoted as:

$I_s > 1$

$D_s > 1$

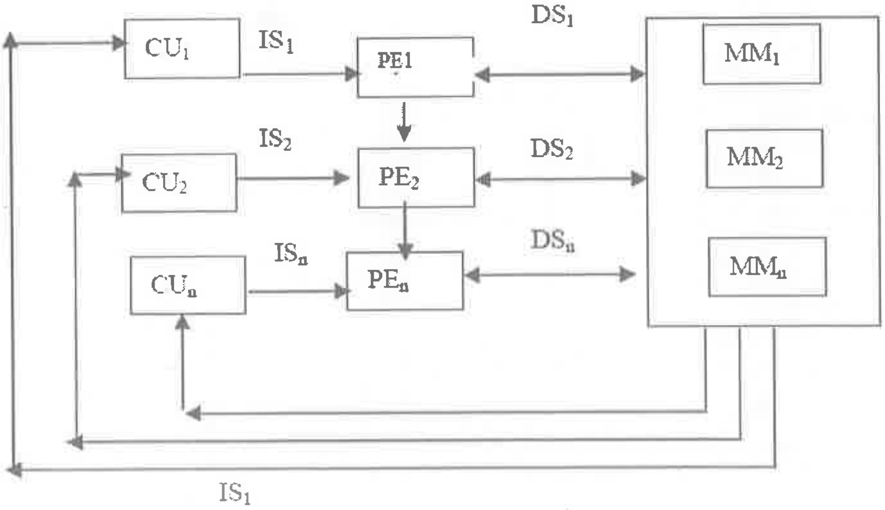


Figure 7: MIMD Organisation

Parallel processing challenges:- (Refer Book also) 7

Parallel processing:-

It is form of processing in which many calculations are carried out simultaneously operating on a principle which divides large problems into smaller ones and then solve them concurrently (in parallel).

→ There are several different forms of parallele processing.

1. bit Level
2. Instruction level
3. data and task parallelism.

Parallel computers has become the dominant paradigm in computer architecture mainly in the form of multi-core processors.

→ Parallel computers can be roughly classified according to the level at which the hardware support parallelism.

1. Multicore & Multiprocessor - having multiple processing elements within a single machine or CPU.
2. Clusters, MPPs (Massively parallel processing), grid - multiple CPU's running in parallel on the same task.

Parallel computer programs are more difficult to write than sequential ones because of the problem of concurrency.

So communication and synchronization between the different subtasks are the greatest obstacles & in getting parallel program performance.

Parallel processing challenges :- (Speed up Challenges).

The two main challenges of parallel processing are

① Limited Parallelism or Insufficient parallelism
② Long Latency to Remote Memory

1) Limited parallelism :-

The limitation in available parallelism make difficult to achieve good speedups in any parallel processor.

Amdhals law is used to measure the performance Multiprocessor.

Amdahl's law :-

It is also known as Amdahl's argument, is named after computer architect Gene Amdahl and used to find the maximum expected improvement to an overall system when only or one part of the system is improved.

It is often used in parallel computing to predict the maximum speedup using multiple processors.

$$\boxed{\text{Speedup} = \frac{1}{\frac{\text{fraction enhanced}}{\text{Speedup enhanced}} + (1 - \text{franction enhanced})}}$$

Example :-

what fraction of the orginal computation can be sequential to achieve speedup of 80 with 100 processor.

Solution :-

Amdahl's law :-

$$\text{Speedup} = \frac{1}{\frac{\text{fraction enhanced}}{\text{Speedup enhanced}} + (1 - \text{fraction enhanced})}$$

$$80 = \frac{1}{\frac{\text{fraction enhanced}}{100} + (1 - \text{fraction enhanced})}$$

$$\frac{0.8\cancel{0}}{\cancel{1}0\cancel{0}}(\text{fraction enhanced}) + 80 - 80\,\text{fraction enhanced} = 1$$

$$= 0.8\,\text{fraction enhanced} + 80 - 80\,\text{fraction enhanced} = 1$$

$$80 - \overset{79.2\times}{\cancel{72}}\,\text{fraction enhanced} = 1$$

$$-79.2 \times \text{fraction enhanced} = 1 - 80$$

$$\cancel{-}79.2 \times \text{fraction enhanced} = \cancel{-}79$$

$$\text{fraction enhanced} = \frac{79}{79.2} = \underline{\underline{0.9975}}$$

$$\Rightarrow 1 - 0.9975 = 0.0025 \times 100 \Rightarrow 0.25\,\%\,//$$

→ To achieve linear speedup, the entire programs must usually be parallel with no serial portions.

→ programs do not operate in fully parallel or sequential mode, but often use less than the full complement of the processor when running in parallel mode.

2) Long latency to remote Memory :-

The communication of data between processor in the existing Shared Mememory Multiprocessors may range from 50 clock cycles to 1000 clock cycles.

This communication mainly depend on the Communication Mechanism, type of interconnection network, and scale of the multiprocessor.

We can calculate the cycles per instructions (CPI) for the multiprocessing systems by

$$\boxed{\text{CPI} = \text{Base CPI} + \text{Remote Request Rate} \times \text{Remote Request Cost}}$$

Base CPI - CPI when all reference hit in the Local cache.

Remote Request rate = Rate at which the Processor ~~[illegible]~~ request the remote ~~Processor~~. Memory

$$\boxed{\text{Remote Request cost} = \frac{\text{Remote Access cost}}{\text{Cycle time}}}$$

Remote Access Cost - Time taken to handle the Reference to Remote Memory.

~~Overcoming the Ethe~~

How to Overcome the two challenges :-

① Insufficent parallelism can be attacked primarily in software with new Algorithms that can have better parallel performance.

② Reducing the long Remote latency can be attacked both by the architecture and by the programmer.

③ We can reduce the frequency of the remote access with either Hardware Mechanisms, like Caching shared data, or some software

Mechanisms like reconstructing the data to make more access local.

→ We can also use Multithreading or by using pre-fetching.

Two terms that describe the ways to scale up are:-

① Strong scaling
② Weak scaling.

Strong scaling- Measuring speedup while keeping the problem size fixed.

Weak Scaling:- Means that the program size grows proportionally to the increase in the number of processor.

Consider size of the problem, M is the working set in main memory, & P processor then

Memory per processor for strong scaling is approximately M/P.

& for weak scaling it is approximately M.

Hardware Multithreading

(Refer your text Book also)

Thread :-

A thread is a seperate process with its own instruction and data. It represents a process that is a part of a parallel program consisting of multiple processes.

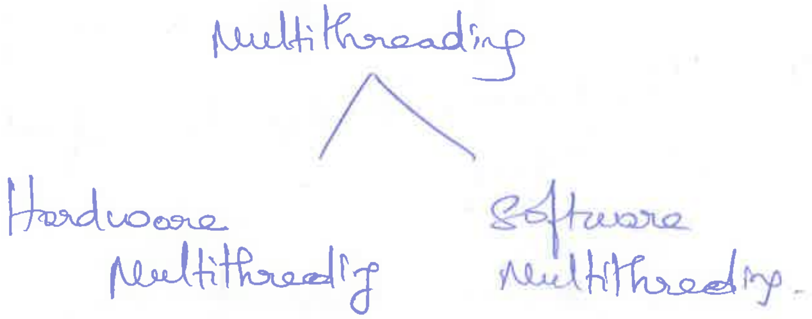
Each thread has all the state (instruction, data, PC, register, state etc) which are necessary to execute.

Multithreading :-

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion.

To allow ~~the~~ sharing the processor must duplicate the independent state of each thread.

Multithreading is classified into two types


Multithreading

Hardware Multithreading

Software Multithreading.

Multithreading is a higher-level-parallelism called as Thread-Level-parallelism where it is logically structured as seperate thread of Execution.

Thread - Level - parallelism :-

It is an important alternative to Instruction- ~~thread~~ level - parallelism , it is most cost - effective to exploit the instruction Level parallelism . " Here multiple thread are executed in overlapping fashion".

Hardware Multithreading :-

Hardware Multithreading is a multithreading that allow multiple threads to share the functional units of a single processor in an overlapping fashion.

The processor must duplicate the Indipendent state of each thread ,

1. Register file
2. Pc (Program Counter)
3. Memory [shared through Virtual - Memory Mechanism)
4. Hardware supports fast - thread switching .

Here the Hardware must support the ability to change to a different thread relatively quickly, Called thread switch . A thread switch should be much more efficient than a process switch which requires hundreds to thousands of processor cycle.

Approaches of Hardware Multithreading.

There are two main approaches,

1. Coarse grained Multithreading
2. Fine-grained Multithreading.

Fine-grained Multithreading :-

Fine-grained Multithreading switches between threads on each instruction, causing the execution of multiple threads to be interleaved.

The Interleaving is done in round-robin fashion, skipping any thread that are stalled at that time.

The CPU must be able to switch threads on every clock cycles.

Advantage :-

One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls.

Instruction from other thread can be executed when one thread stalls.

Disadvantage :-

* Slow down the execution of individual threads
* A thread is ready to executed without stall

will be delayed will be delayed by instruction from other thread.

2) Coarse - Grained Multithreading:-

* Coarse - grained multithreading is an alternative to fine-grained multithreading.

* It switches thread only on costly stall, such as level 2 cache ~~issues~~ misses.

* This change relieves the need to have thread switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued when thread encounters a costly stall.

Advantage:-

→ It doesnt need to have very fast thread-switching

→ Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall.

Disadvantge:-

* Hard to overcome throughput losses from shorter stalls due to pipeline start-up costs.

→ Since cpu issues instruction from ~~other threads issued~~ one thread, when a stall occurs the pipeline must be emptied or frozen.

Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction-level parallelism (SMT):-

* A Simultaneous Multithreading Architecture is superior in performance to a multiple issue multiprocessor (Multi-issue CMP) → Chip Level Multiprocessing.

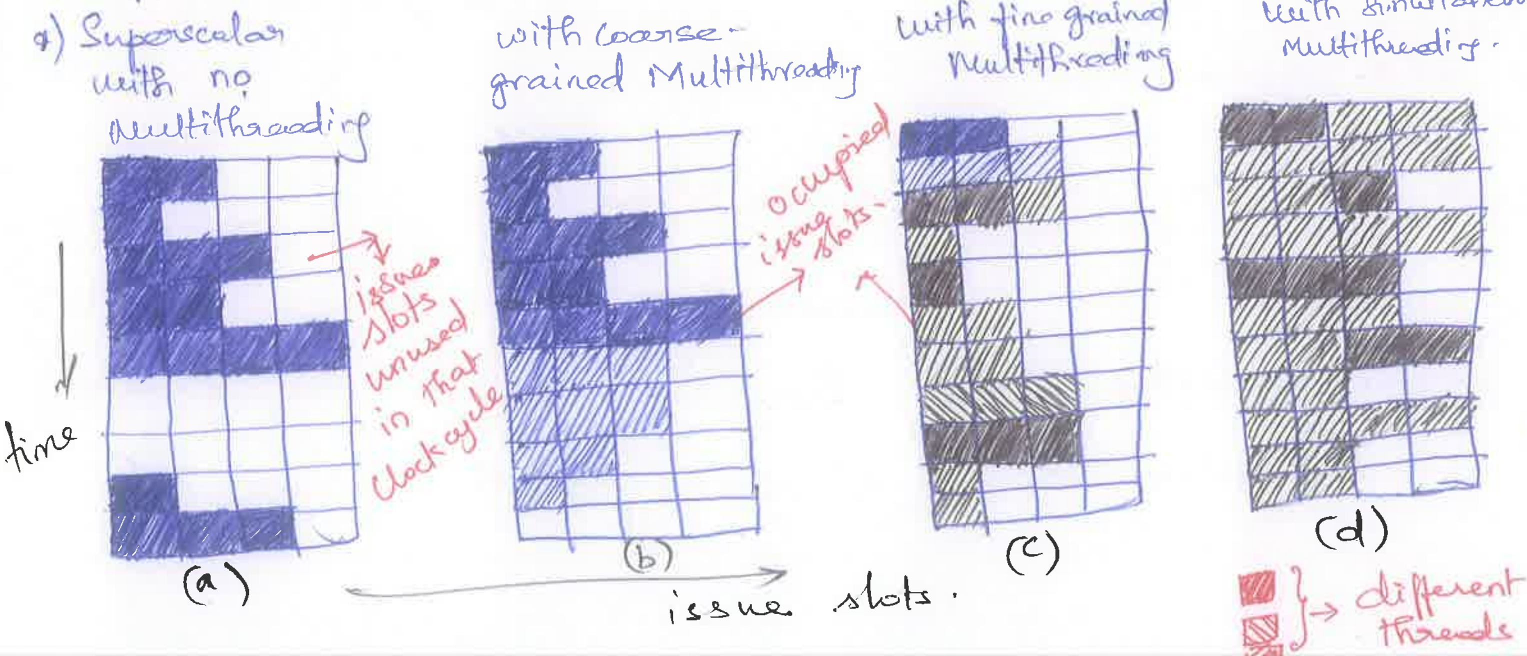
* SMT boosts the utilization by dynamically scheduling functional unit among multiple threads.

* SMT also increases hardware design flexibility.

* It increases the complexity of instruction scheduling.

* Simultaneous Multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit TLP at the same time it exploits ILP.

Consider the figure that illustrates the difference in a processor's ability to exploit the resources of a superscalar for the following processor configuration.


a) Superscalar with no Multithreading
with coarse-grained Multithreading
with fine grained Multithreading
with Simultaneous Multithreading.
time
issues slots unused in that clock cycle
occupied issue slots.
(a)
(b)
(c)
(d)
issue slots.
} → different threads

* Horizontal dimentions represents the instruction issue capability in each clock cycle.

* Vertical Dimention represents a sequence of clock cycles.

* Empty slots indicate that the corresponding issue slots are unused in that clock cycle.

(a) In the superscalar without Multithreading – ~~the~~ ~~block~~ supports the use of issue slots is limited by lack of ILP. a major stall such as an instruction cache miss, can leave the entire processor idle.

(b) In the coarse-grained multithreading superscalar:-.

The long stalls are partially hidden by switching to another thread that uses the resources of the processor.

This reduces the no of completely clock cycles.

Since thread switching occurs only during the long stall, there is likely to some fully idle cycles remaining.

(c) In the finegrained Multithreading:-

The interleaving of threads eliminates fully empty slots. Because only one thread issues instruction in a given clock cycle.

(d) In the SMT Case : TLP and ILP are exploited simultaneously, with multiple thread using the issue slots in a single clock cycle

The issue slot usage is limited by the following factors :-

* Resource availability over Multiple thread.

* Finitie limitations of buffer.

* Imbalances in Resource need

* Number of Active threads considered.

* Ability to fetch enough instructions from Multiple threads.

Hardware Support for SMT :-

* large set of Virtual register used to hold the register set of independent threads.

* Register Renaming provides unique register identifiers

* Out of Order Completion allows thread to execute out of order and get better utilization of the Hardware.

Design challanges in SMT processors :-

* ~~Dealing with a~~ (larger registers files are needed to hold multiple contexts)

* Maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instruction need to be considered and in instruction completion, where choosing what instructions to commit may be challenging. ~~and~~

* Ensuring that the cache conflicts generated by the simultaneously execution of multiple threads do not cause significant performance degradation.

* In viewing these problems, two observations are important. In many cases, the (1) potential performance overhead due to multithreading is small, and simple choices work well enough. second, the (2) efficiency of current superscalar is low enough that there is room for significant improvement, even at the cost of some overhead.

Works well if

* No of compute intensive thread doesnt exceed the no of thread supported in ~~[illegible]~~. SMT.

* Threads have highly different characteristics.

doesnt work well if

* Threads try to utilize the same functional unit.

* Assignment problem (eg) dual processor system, each processor supporting two thread simultaneously.

6.5 Multicore and Other Shared Memory Multiprocessors

While hardware multithreading improved the efficiency of processors at modest cost, the big challenge of the last decade has been to deliver on the performance potential of Moore's Law by efficiently programming the increasing number of processors per chip.

Given the difficulty of rewriting old programs to run well on parallel hardware, a natural question is: what can computer designers do to simplify the task? One answer was to provide a single physical address space that all processors can share, so that programs need not concern themselves with where their data is, merely that programs may be executed in parallel. In this approach, all variables of a program can be made available at any time to any processor. The alternative is to have a separate address space per processor that requires that sharing must be explicit; we'll describe this option in the Section 6.7. When the physical address space is common then the hardware typically provides cache coherence to give a consistent view of the shared memory (see Section 5.8).

As mentioned above, a *shared memory multiprocessor* (SMP) is one that offers th programmer a *single physical address space* across all processors—which is nearly always the case for multicore chips—although a more accurate term would have been shared-*address* multiprocessor. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. Figure 6.7 shows the classic organization of an SMP. Note that such systems ca still run independent jobs in their own virtual address spaces, even if they all share a physical address space.

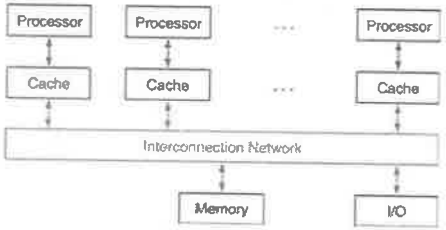


FIGURE 6.7 Classic organization of a shared memory multiprocessor.

Single address space multiprocessors come in two styles. In the first style, the latency to a word in memory does not depend on which processor asks for it. Such machines are called **uniform memory access (UMA)** multiprocessors. In the second style, some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different microprocessors or to different memory controllers on the same chip. Such machines are called **nonuniform memory access (NUMA)** multiprocessors. As you might expect, the programming challenges are harder for a NUMA multiprocessor than for a UM multiprocessor, but NUMA machines can scale to larger sizes and NUMAs can ha lower latency to nearby memory.

uniform memory access (UMA)
A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

nonuniform memory access (NUMA)
A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called **synchronization**, which we saw in Chapter 2. When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a **lock** for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable. Section 2.11 of Chapter 2 describes the instructions for locking in the MIPS instructio set.

synchronization
The process of coordinating the behavior of two or more processes, which may be running on different processors.

15

lock

A synchronization device that allows access to data to only one processor at a time.

A Simple Parallel Processing Program for a Shared Address Space

Example

Suppose we want to sum 64,000 numbers on a shared memory multiprocessor computer with uniform memory access time. Let's assume we have 64 processors.

Answer

The first step is to ensure a balanced load per processor, so we split the set of numbers into subsets of the same size. We do not allocate the subsets to a different memory space, since there is a single memory space for this machine; we just give different starting addresses to each processor. Pn is the number that identifies the processor, between 0 and 63. All processors start the program by running a loop that sums their subset of numbers:

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
sum[Pn] += A[i]; /*sum the assigned areas*/
```

(Note the C code `i += 1` is just a shorter way to say `i = i + 1`.)

The next step is to add these 64 partial sums. This step is called a **reduction**, where we divide to conquer. Half of the processors add pairs of partial sums, and then a quarter add pairs of the new partial sums, and so on until we have the single, final sum. Figure 6.8 illustrates the hierarchical nature of this reduction.

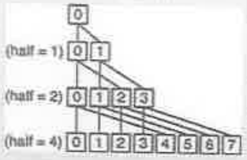


FIGURE 6.8 The last four levels of a reduction that sums results from each processor, from bottom to top.

In the SMT case, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads.

In Figure 4.6, the horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the orresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

- The issue slot usage is limited by the following factors:
  - Resource availability over multiple threads.
  - Finite limitations of buffer.
  - Imbalances in the resource needs.
  - Number of active threads considered.
  - Ability to fetch enough instruction from multiple threads.
- Simultaneous Multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads.
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads.
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW.
- Just adding a per-thread renaming table and keeping separate PCs.
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread.

4.5.4 Design Challenges in SMT

- Impact of fine-grained implementation on single thread performance.
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts

- Not affecting clock cycle time, especially in
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by simultaneous execution of multiple threads do not cause significant performance degradation.

The following two observations that are made with respect to these problems are:

(i) Potential performance overhead due to multithreading is small.

(ii) Efficiency of superscalar is low with the room for significant improvement.

- Works well if:
  - Number of compute intensive threads does not exceed the number of threads supported in SMT.
  - Threads have highly different characteristics.
- Does not work well if:
  - Threads try to utilize the same function units.
  - Assignment problems (e.g.,) dual processor system, each processor supporting two threads simultaneously.

4.6 MULTICORE PROCESSOR

- ➢ A multicore design takes several processor cores and packages them as a single processor. The goal is to enable system to run more tasks simultaneously and thereby achieve greater overall system performance.
- ➢ Starting in the 1990s, the increasing capacity of a single chip allowed designers to place multiple processors on a single die. This approach, initially called onchip multiprocessing or single-chip multiprocessing , has come to be called multicore , a name arising from the use of multiple processor cores on a single die.
- ➢ Multicore processor is one in which there will be more than one processor, examples are dual core, Quadracore processors etc.Here both processors will work independently and so speed will be improved.
- ➢ In such a design, the multiple cores typically share some resources, such as a second- or third-level cache or memory and I/O buses.
- ➢ Recent processors, including the IBM Power5, the Sun T1, and the Intel Pentium D and Xeon-MP, are multicore and multithreaded. Just as using multiple copies of a microprocessor in a multiprocessor leverages a design investment through replication, a

multicore achieves the same advantage relying more on replication than the alternative of building a wider superscalar.

- With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. A process is a segment of code that may be run independently; the state of the process contains all the information necessary to execute that program on a processor.
- In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of other processes.
- It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called threads
- Today, the term thread is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space.
- For example, a multithreaded architecture actually allows the simultaneous execution of multiple processes, with potentially separate address spaces, as well as multiple threads that share the same address space.
- To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute.
- The independent threads within a single process are typically identified by the programmer or created by the compiler.
- The threads may come from large-scale, independent processes scheduled and manipulated by the operating system.
- A thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop. Although the amount of computation assigned to a thread, called the grain size, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel.
- Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor may be able to efficiently parallelize operations on short vectors, the resulting grain size when the parallelism is split among many threads may be so small that the overhead makes the exploitation of the parallelism prohibitively expensive.

- Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors is likely to change over time.
- The first group, which we call centralized shared-memory architectures, has at most a few dozen processor chips (and less than 100 cores) in 2006.
- For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory. With large caches, a single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors.
- By using multiple point-to-point connections, or a switch, and adding additional memory banks, a centralized shared-memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable, sharing a centralized memory becomes less attractive as the number of processors sharing it increases.

Major MIMD styles

The existing MIMD multiprocessors fall into two classes depending on the number of processors involved.

I. Centralized Shared Memory ("Uniform Memory Access" time or "Shared Memory Processor").

2. Physically Distributed-Memory Multiprocessor (Decentralized Memory or Memory Module with CPU).

4.6.1 Centralized Shared-Memory Architecture

Centralized Shared-Memory Architecture share, a single centralized memory, interconnect processors and memory by a bus. It is known as "uniform memory access" (UMA) or "symmetric (shared-memory) multiprocessor" (SMP) because there is a single main memory that has a:

- A symmetric relationship to all processors.
- A uniform memory access time for any processor.

Figure 4.7 shows what these multiprocessors based on a multicore chip. Multicore processor is composed of two or more independent cores. Manufacture typically integrate the cores into a single integrated circuit die (known as a chip multi-processor or CMP), or onto multiple dies in a single chip package.

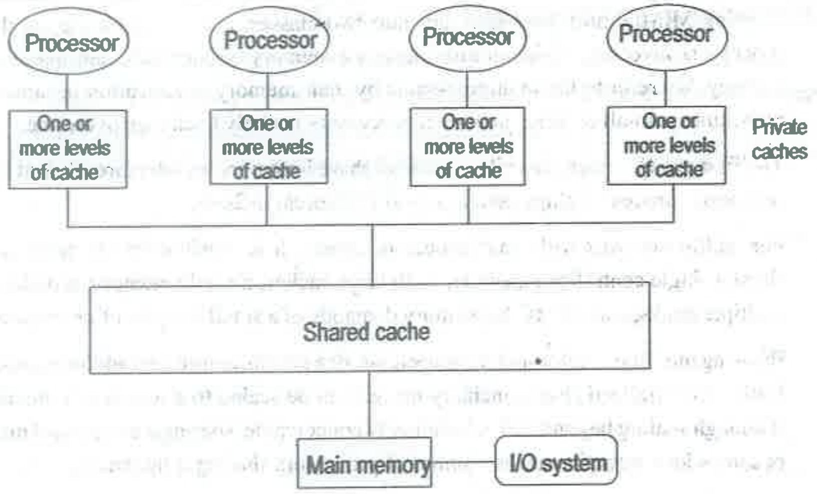

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 4.7: Basic structure of a centralized shared-memory multiprocessor based on a multicore chip.

A many-core processor is one in which the number of cores is large enough that traditional multiprocessor techniques are no longer efficient-largely due to issues with congestion supplying sufficient instructions and data to the many processors.

Multiple processor–cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.

Advantages:

Large caches can satisfy the memory demands of a small number of processors.

Disadvantages:

Scalability Problem: Less attractive for large scale processors.

- ➢ This type of symmetric shared-memory architecture is currently by far the most popular organization.
- ➢ The second group consists of multiprocessors with physically distributed memory. Figure 4.8 shows what these multiprocessors look like.

4.6.2 Distributed-Memory Multicore Processor

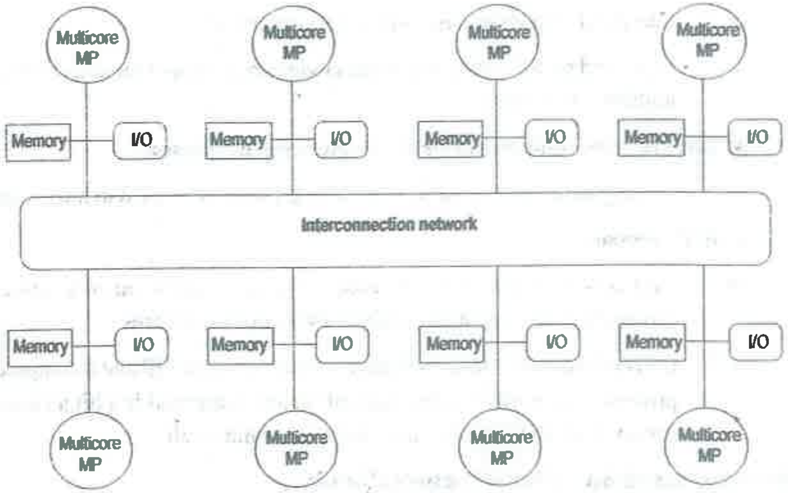


Figure 4.8 The basic architecture of a distributed-memory multiprocessor in 2011 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes.

Each processor core shares the entire memory, although the access time to the lock memory attached to the core's chip will be much faster than the access time to remote memories.

To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The larger number of processors also raises the need for a high-bandwidth interconnect.

Both direction networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used.

Distributing the memory among the nodes has two major benefits.

1. It is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node.
2. It reduces the latency for accesses to the local memory.

These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency.

- **Advantages of multicore processor**
  - Increased responsiveness and woker productivity.
  - Improved performance in parallel environments when running computations on multiple processors.
- **Multicore processor superior to single core processor**
  - On singlecore processor multithreading can only be used to hide latency.
- **Cell Processor**
  - Cell is heterogeneous multi core processor comprised of control intensive processor and compute intensive SIMD processor-cores.
  - Cell consists of 1 control intensive processor core(PPE) and 8 compute intensive processor core(SPE)- EIB (element interconnect bus) is a high speed bus used for interconnecting the processor cores within a cell.

Classification based on Communication Models.

1. Distrubuted Shared Memory (DSM)

Distributed Shared-Memory (DSM) architecture is a multiprocessor with a shared address space in which communication occurs through a shared address space.

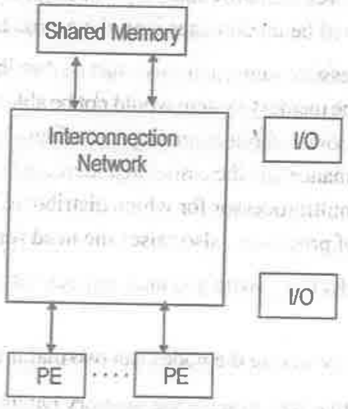


Fig. 4.9: Shared Memory Multiprocessor

Shared Address Space – the address space that is shared i.e., the same physical address on two processor refers to the same location in memory.

- UMA (Uniform Memory Access Time)
- NUMA (Non-Uniform Memory Access Time Multiprocessors).

2. Message Passing Multiprocessors

Message Passing Multiprocessor is a multiprocessor with multiple address space in which communication of data occurs by explicitly passing message among the processors.

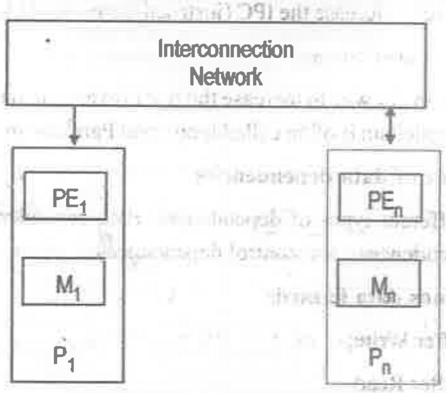


Fig. 4.10: Message Passing Multicomputers

Each processor memory module is essentially a separate computer. For this multicomputer arrangement, communication is done by message passing either by synchronous or asynchronous.

TWO MARKS QUESTIONS AND ANSWERS

1. **Expand ILP.**

   ILP = Instruction Level Parallelism.

2. **What is ILP?**

   ILP is the technique which is used to overlap the execution of instruction.

3. **What are the needs of ILP.**

   - Sufficient resources
   - Parallel scheduling
     - Hardware solution
     - Software solution
   - Application should contain ILP

4. **How to calculate the value of CPI.**

   The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls: Pipeline CPI = Ideal pipeline CPI+Structural Stalls + Data Hazard Stalls + Control Stalls.