JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self-Belief | Self Discipline | Self Respect”

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8491 – COMPUTER ARCHITECTURE

(Regulation 2017)

Year/Semester: II/IV CSE

2020 – 2021

Prepared by

Ms. R. Revathi

Assistant Professor/CSE

UNIT IV

PARALLELISM

Parallel processing challenges – Flynn‘s classification – SISD, MIMD, SIMD, SPMD, and Vector Architectures - Hardware multithreading – Multi-core processors and other Shared Memory Multiprocessors - Introduction to Graphics Processing Units, Clusters, Warehouse Scale Computers and other Message-Passing Multiprocessors.

PARALLEL PROCESSING CHALLENGES:

- It is difficult to write software that uses multiple processors to complete one task is faster.
- Parallel processing will increase the performance of processor and it will reduce the utilization time to execute a task.
- By obtaining the parallel processing is not an easy task.
- The difficulty is not in hardware side it is in software side. We can understand that it is difficult to write parallel processing programs that are fast, especially as the number of processor increases.

Advantage:

1. To get better performance
2. It produce better energy efficiency

Performance:

If we are running a program on two different desktop computers, we will say that the faster one is the desktop computer that gets the job done first.

In some situation we cannot get parallel processing as faster than sequential programs. The reasons are,

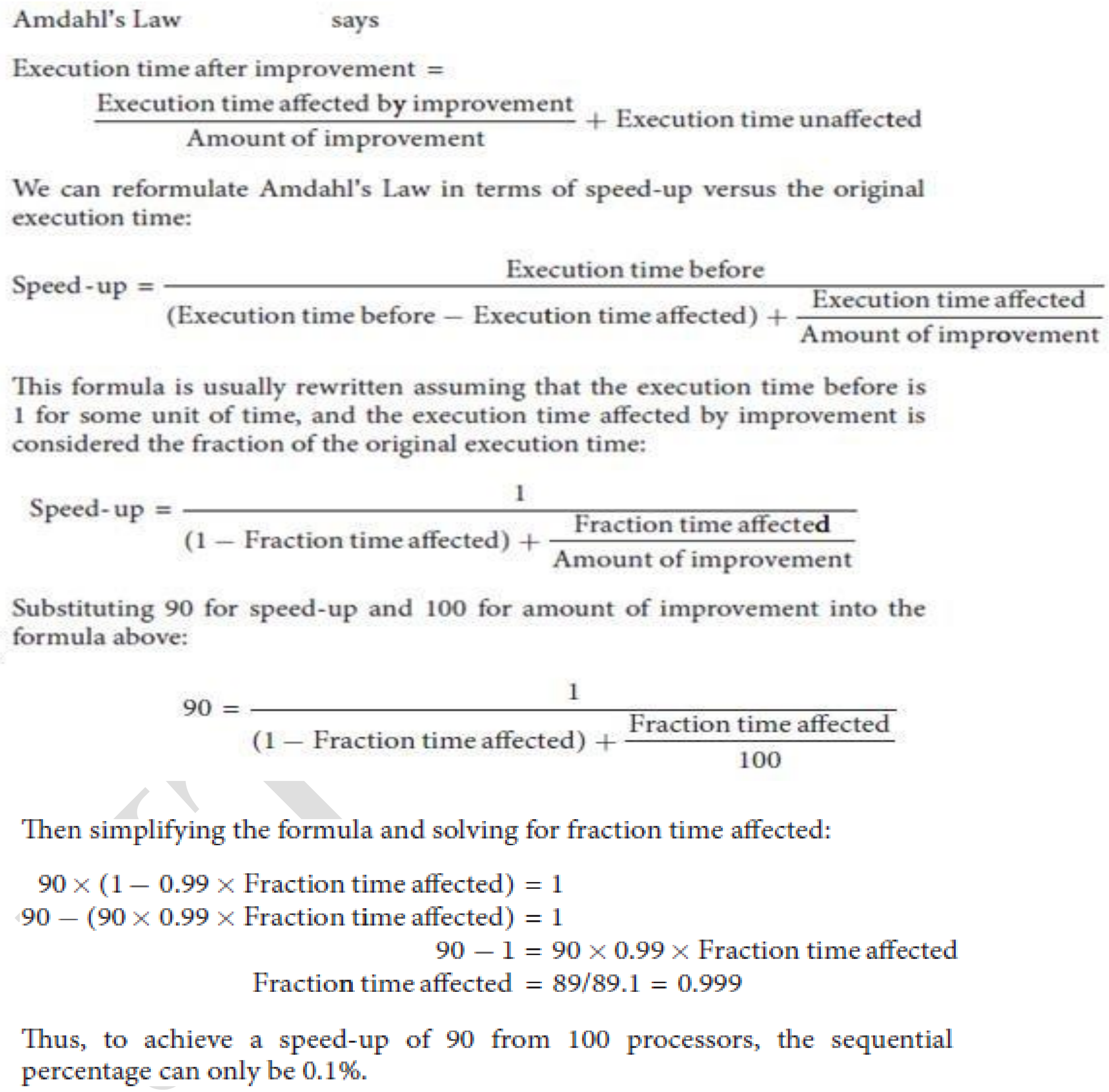
1. **Scheduling:** It is done to load balance and share system resources effectively and achieve a target. It can be done by the following ways,
    1. Long term
    2. Medium Term
    3. Short Term
    4. Dispatcher
2. **Portioning the work into parallel pieces:** Divide the task equally to all the processor.
3. **Load Balancing:** Work load distribute evenly and amount of execution time is also equal.
4. **Time to Synchronize:** Here the throughput must be high.
5. **Overhead for Communication**

Strong scaling: Speedup achieved on a multiprocessor without increasing the size of the problem.

Weak scaling: Speedup achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Speed-up Challenge

Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentage of the original computation can be sequential? **[Dec'17]**



Amdahl's Law says

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

We can reformulate Amdahl's Law in terms of speed-up versus the original execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \dfrac{\text{Execution time affected}}{\text{Amount of improvement}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of time, and the execution time affected by improvement is considered the fraction of the original execution time:

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \dfrac{\text{Fraction time affected}}{\text{Amount of improvement}}}$$

Substituting 90 for speed-up and 100 for amount of improvement into the formula above:

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \dfrac{\text{Fraction time affected}}{100}}$$

Then simplifying the formula and solving for fraction time affected:

$$90 \times (1 - 0.99 \times \text{Fraction time affected}) = 1$$

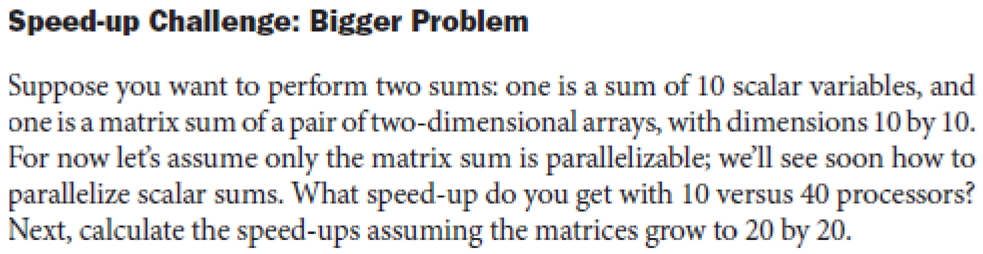
$$90 - (90 \times 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - 1 = 90 \times 0.99 \times \text{Fraction time affected}$$

$$\text{Fraction time affected} = 89/89.1 = 0.999$$

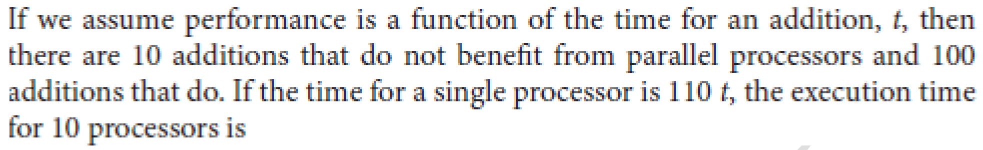
Thus, to achieve a speed-up of 90 from 100 processors, the sequential percentage can only be 0.1%.

Example: 2

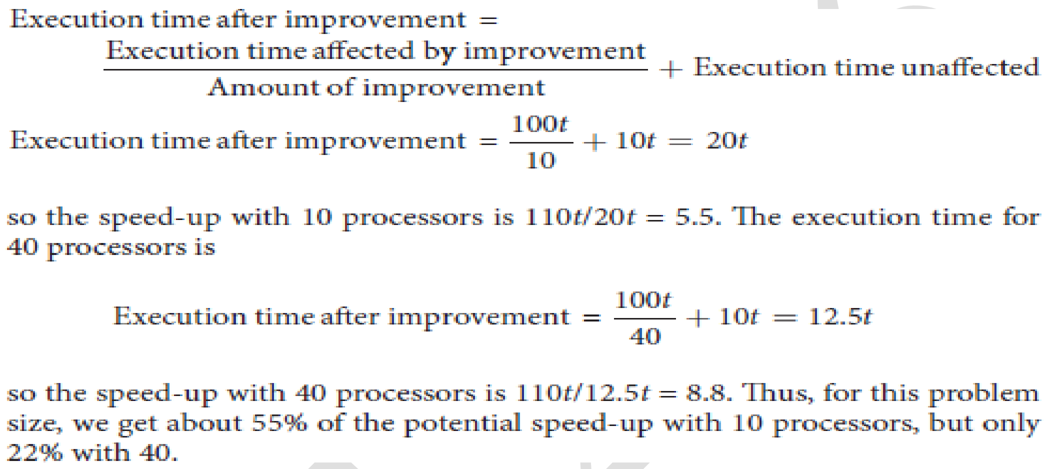


Speed-up Challenge: Bigger Problem

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is parallelizable; we'll see soon how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming the matrices grow to 20 by 20.



If we assume performance is a function of the time for an addition, *t*, then there are 10 additions that do not benefit from parallel processors and 100 additions that do. If the time for a single processor is 110 *t*, the execution time for 10 processors is



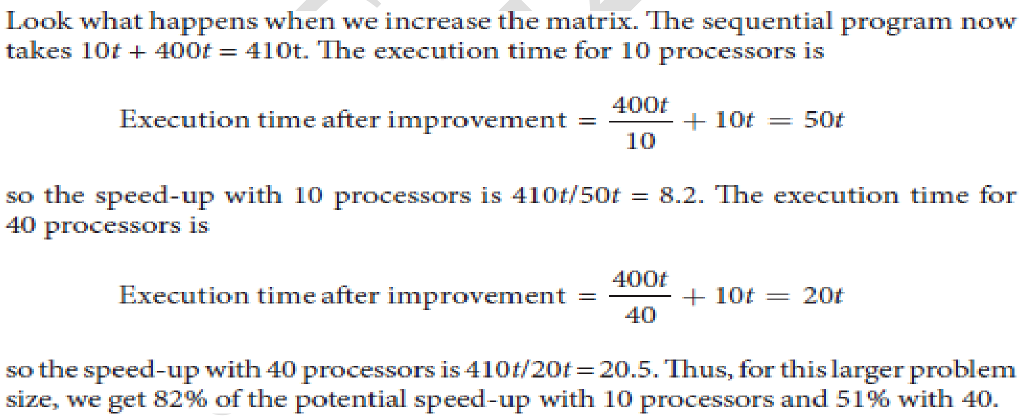
$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t$$

so the speed-up with 10 processors is $110t/20t = 5.5$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t$$

so the speed-up with 40 processors is $110t/12.5t = 8.8$. Thus, for this problem size, we get about 55% of the potential speed-up with 10 processors, but only 22% with 40.



Look what happens when we increase the matrix. The sequential program now takes $10t + 400t = 410$t. The execution time for 10 processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t$$

so the speed-up with 10 processors is $410t/50t = 8.2$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t$$

so the speed-up with 40 processors is $410t/20t = 20.5$. Thus, for this larger problem size, we get 82% of the potential speed-up with 10 processors and 51% with 40.

Speed-up Challenge: Balancing Load

Example:3

To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced. That is, each of the 40 processors had 2.5% of the work to do. Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at twice the load (5%) and five times the load (12.5%) for that hardest working processor. How well utilized are the rest of the processors?

Solution:

If one processor has 5% of the parallel load, then it must do 5% × 400 or 20 additions, and the other 39 will share the remaining 380. Since they are operating simultaneously, we can just calculate the execution time as a maximum

$$\text{Execution time after improvement} = \text{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

The speed-up drops from 20.5 to $410t/30t = 14$. The remaining 39 processors are utilized less than half the time: while waiting $20t$ for hardest working processor to finish, they only compute for $380t/39 = 9.7t$.

If one processor has 12.5% of the load, it must perform 50 additions. The formula is:

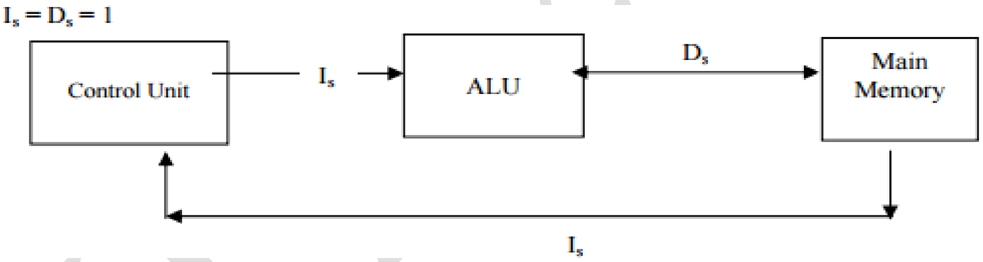
$$\text{Execution time after improvement} = \text{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

The speed-up drops even further to $410t/60t = 7$. The rest of the processors are utilized less than 20% of the time ($9t/50t$). This example demonstrates the importance of balancing load, for just a single processor with twice the load of the others cuts speed-up by a third, and five times the load on just one processor reduces speed-up by almost a factor of three.

PROCESSOR ORGANAIZATION [FLYNN'S CLASSIFICATION]

SISD

- Single Instruction stream, Single Data stream.
- Example of SISD is uniprocessor.
- It has a single control unit and producing a single stream of instruction.
- It has one processing unit and the processing has more than one functional unit these are under the supervision of one control unit.
- It has one memory unit.

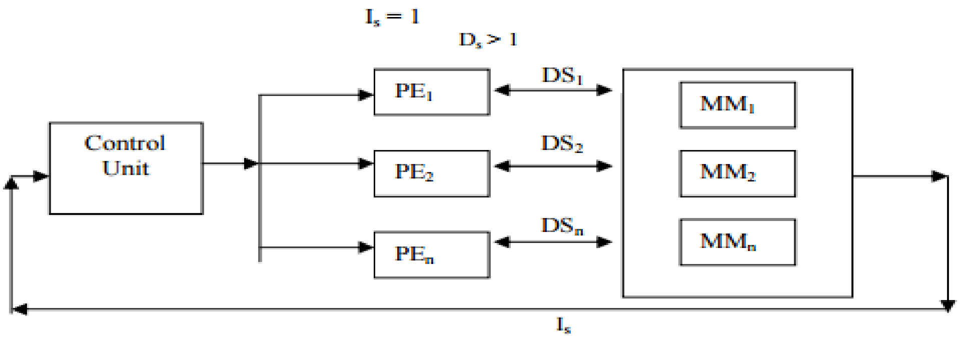


SIMD

- It has one instruction and multiple data stream.
- It has a single control unit and producing a single stream of instruction and multi stream of data.
- It has more than one processing unit and each processing unit has its own associative data memory unit.
- In this organization, multiple processing elements work under the control of a single control unit.
- A single machine instruction controls the simultaneous execution of a number of processing element.
- Each instruction to be executed on different sets of data by different processor.
- The same instruction is applied to many data streams, as in a vector processor.
- 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.
- 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.
- Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous.
- Example of SIMD is Vector Processor and Array Processor.

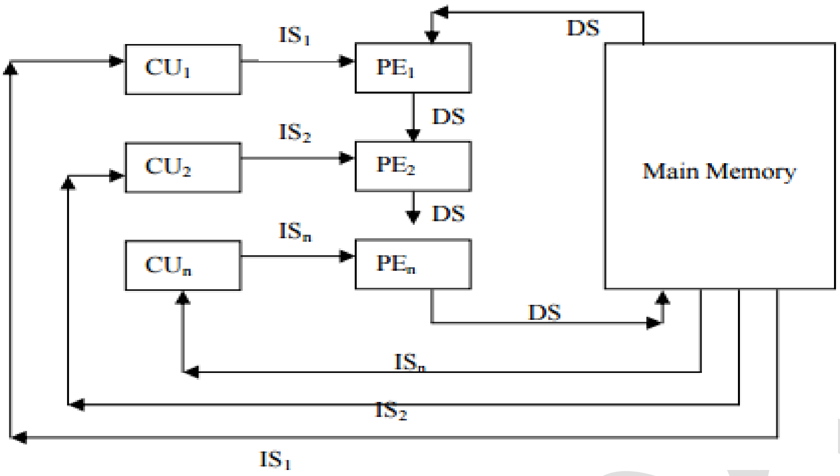


Advantage of SIMD:

- The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units.
- Another advantage is the reduced instruction bandwidth and space.
- SIMD needs only one copy of the code that is being simultaneously executed while message-passing MIMDs may need a copy in every processor, and shared memory MIMD will need multiple instruction caches.
- SIMD works best when dealing with arrays in for loops because parallelism achieved by performing the same operation on independent data.
- SIMD is at its weakest in case or switch statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data must be disabled so that units with proper data may continue.

MISD

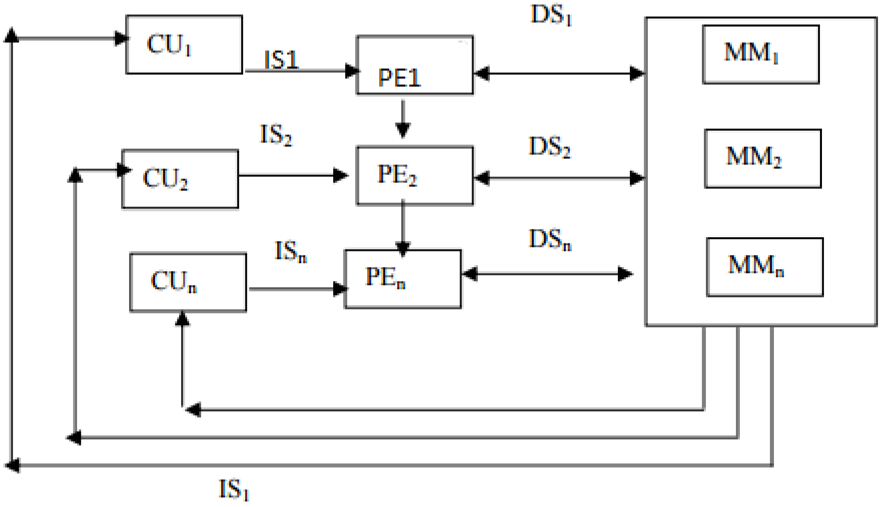
- Multiple Instruction and Single Data stream (MISD)
- In this organization, multiple processing elements are organized under the control of multiple control units.
- Each control unit is handling one instruction stream and processed through its corresponding processing element.
- But each processing 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 organized in this classification.
- All processing elements are interacting with the common shared memory for the organization of single data stream as shown in figure.
- The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.



MIMD

- Multiple Instruction streams and Multiple Data streams (MIMD). In this organization, multiple processing elements and multiple control units are organized.
- Compared to MISD 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.
- 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 organization is said to be a Parallel computer.

$I_s > 1$
$D_s > 1$



SIMD-VECTOR ARCHITECTURE [SPMD]

- SIMD is called vector architecture.

- It is also a great match to problems with lots of data-level parallelism.i.e. Parallelism achieved by performing the same operation on independent data.
- Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors.
- The vector architectures pipelined the ALU to get good performance at lower cost.
- The basic idea of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using pipelined execution units, and then write the results back to memory.
- A key feature of vector architectures is then a set of vector registers. Thus, vector architecture might have 32 vector registers, each with 64-bit elements.

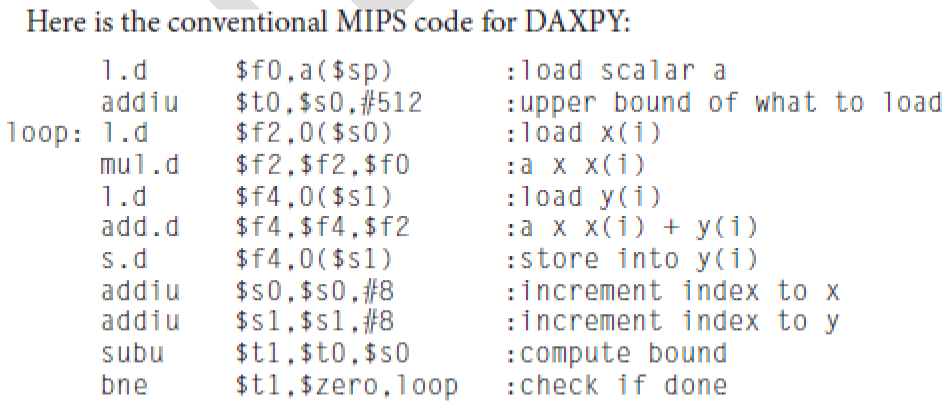
Comparing Vector to Conventional Code

- Suppose we extend the MIPS instruction set architecture with vector instructions and vector registers. Vector operations use the same names as MIPS operations, but with the letter V appended.
- For example, addv.d adds two double-precision vectors. The vector instructions take as their input either a pair of vector registers (addv.d) or a vector register and a scalar register (addvs.d).
- The value in the scalar register is used as the input for all operations.
- The operation addvs.d will add the contents of a scalar register to each element in a vector register.
- The names lv and sv denote vector load and vector store, and they load or store an entire vector of double-precision data.
- One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory.
- The conventional MIPS code versus the vector MIPS code for

$$Y = a\times X + Y$$

- Where X and Y are vectors of 64 double precision floating-point numbers, initially resident in memory, and a is a scalar double precision variable.
- This example is the so-called DAXPY loop that forms the inner loop of the DAXPY stands for double precision a × X plus Y.).
- Assume that the starting addresses of X and Y are in $s0 and $s1, respectively.

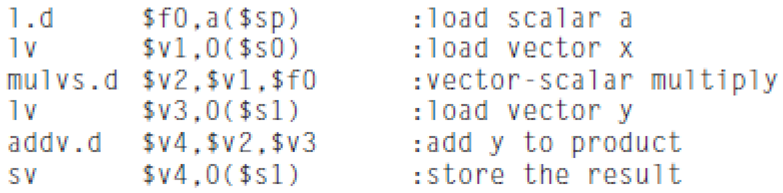
Here is the conventional MIPS code for DAXPY:



Here is the conventional MIPS code for DAXPY:

```
      l.d     $f0,a($sp)       :load scalar a
      addiu   $t0,$s0,#512     :upper bound of what to load
loop: l.d     $f2,0($s0)       :load x(i)
      mul.d   $f2,$f2,$f0      :a x x(i)
      l.d     $f4,0($s1)       :load y(i)
      add.d   $f4,$f4,$f2      :a x x(i) + y(i)
      s.d     $f4,0($s1)       :store into y(i)
      addiu   $s0,$s0,#8       :increment index to x
      addiu   $s1,$s1,#8       :increment index to y
      subu    $t1,$t0,$s0      :compute bound
      bne     $t1,$zero,loop   :check if done
```

Here is the vector MIPS code for DAXPY:



```
l.d      $f0,a($sp)      :load scalar a
lv       $v1,0($s0)      :load vector x
mulvs.d  $v2,$v1,$f0     :vector-scalar multiply
lv       $v3,0($s1)      :load vector y
addv.d   $v4,$v2,$v3     :add y to product
sv       $v4,0($s1)      :store the result
```

- The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus almost 600 for the traditional MIPS architecture.
- This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the vector code.

Vector versus Scalar

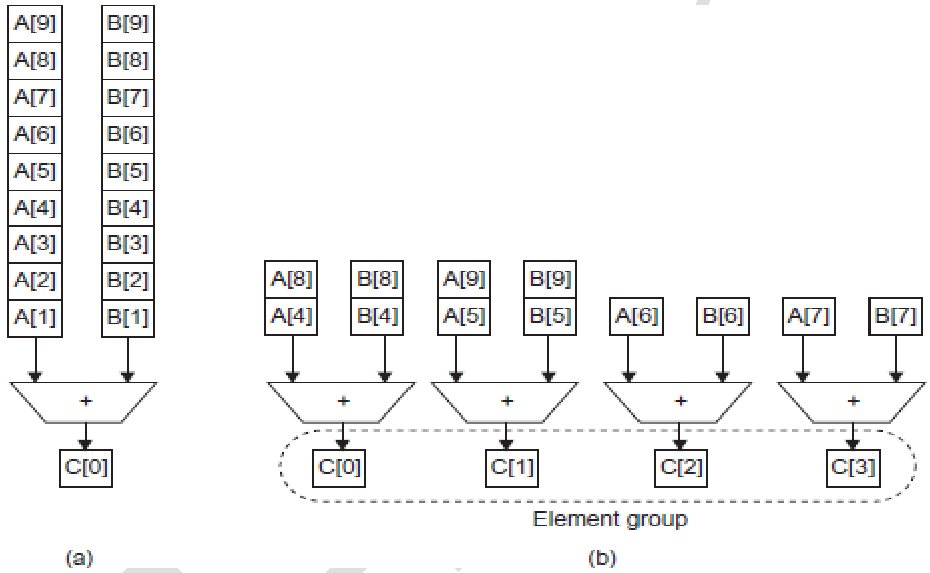
Vector	Scalar
The instruction fetch and decode bandwidth needed is dramatically reduced.	The instruction fetch and decode bandwidth needed is not reduced.
The compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector,	The compiler or programmer indicates that the computation of each result is not independent of the computation of other results.
Hardware does not have to check for data hazards within a vector instruction.	Hardware to check for data hazards within a vector instruction.
Vector architectures and compilers have a reputation of making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.	Not easy to perform this operation
Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors.	Hardware need only check for data hazards for every element within the array.
Save energy because of reduced checking	It does not **s**ave energy because it has more number of checking.
The cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.	The cost of the latency to main memory is seen for each word of the scalar.
Efficient use of memory bandwidth and instruction bandwidth.	No efficient use of memory bandwidth and instruction bandwidth.
Entire loop behavior is predetermined	Entire loop behavior is not a predetermined

Vector Versus Multimedia Extensions

- Like multimedia extensions found in the x86 AVX instructions, a vector instruction specifies multiple operations.
- However, multimedia extensions typically specify a few operations while vector specifies dozens of operations.
- The number of elements in a vector operation is not in the opcode but in a separate register.
- This distinction means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility.
- In contrast, a new large set of opcodes is added each time the vector length changes in the multimedia extension architecture of the x86: MMX, SSE, SSE2, AVX, AVX2 …
- Also unlike multimedia extensions, the data transfers need not be contiguous.

- Hardware finds the addresses of the items to be loaded in a vector register.

- Indexed accesses are also called gatherscatter, in that indexed loads gather elements from main memory into contiguous vector elements and indexed stores scatter vector elements across main memory.
- The following figure illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.
- The figure using multiple functional units to improve the performance of a single vector add instruction, C = A + B.
- The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle.
- The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle.



Vector Lane

- One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.
- Figure shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks per vector instruction by roughly a factor of four.
- The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit.
- For multiple lanes to be advantageous, both the applications and the architecture must support long vectors.
- The elements within a single vector add instructions are interleaved across the four lanes.
- The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register.
- Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction.

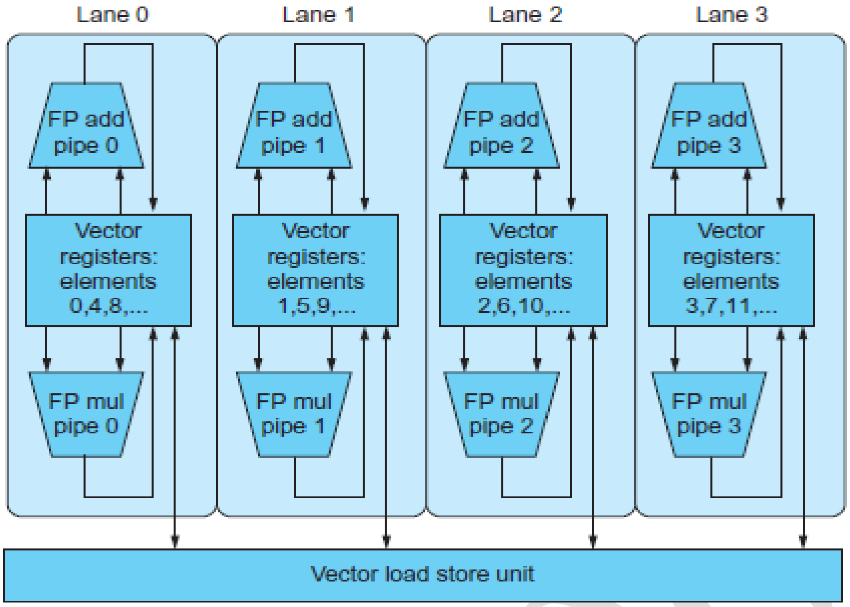


Figure: Structure of a vector unit containing four lanes.

HARDWARE MULTITHREADING

Multithreading

- A mechanism by which the instruction streams is divided into several smaller streams (threads) and can be executed in parallel is called multithreading.

Hardware Multithreading

- Increasing utilization of a processor by switching to another thread when one thread is stalled is known as hardware multithreading.

Thread

- A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

Thread Switch

- The act of switching processor control from one thread to another within the same process. It is much less costly than a processor switch.

Process

- A process includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

What are the approaches to hardware multithreading?

There are two main approaches to hardware multithreading.

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

Fine-grained Multithreading

- A version of hardware multithreading that implies switching between threads after every instruction resulting in interleaved execution of multiple threads. It switches from one thread to another at each clock cycle.
- This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle.

- To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle.

Advantage

- Vertical waste is eliminated.
- Pipeline hazards cannot arise.
- Zero switching overhead
- Ability to hide latency within a thread i.e., it can hide the throughput losses that arise from both short and long stalls.
- Instructions from other threads can be executed when one thread stalls.
- High execution efficiency
- Potentially less complex than alternative high performance processors.

Disadvantage

- Clock cycles are wasted if a thread has little operation to execute.
- Needs a lot of threads to execute.
- It is expensive than coarse-grained multithreading.
- It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained Multithreading

- Coarse-grained multithreading was invented as an alternative to fine-grained multithreading.
- A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.
- This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall.

Advantage

- To have very fast thread switching.
- Doesn’t slow down thread.

Disadvantage

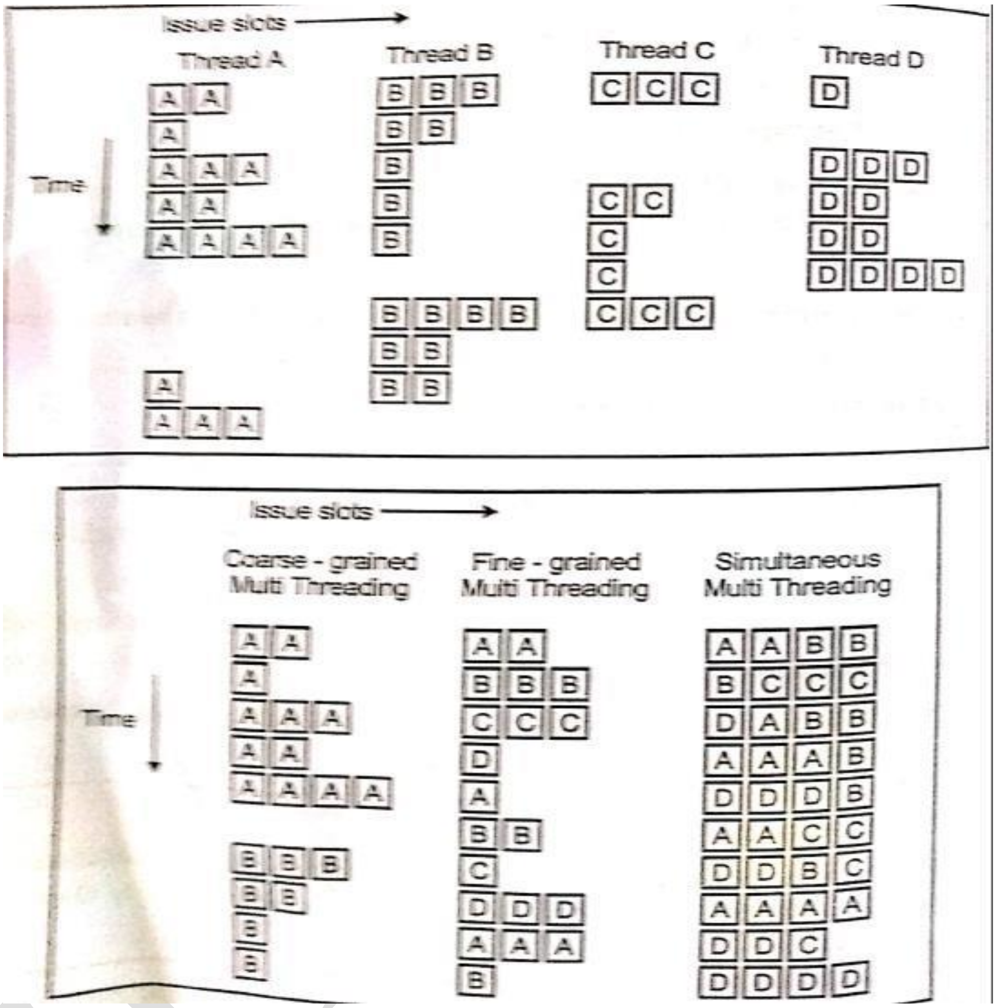
- It is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs.
- Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied.
- New thread must fill pipeline before instructions can complete.
- Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous multithreading (SMT)

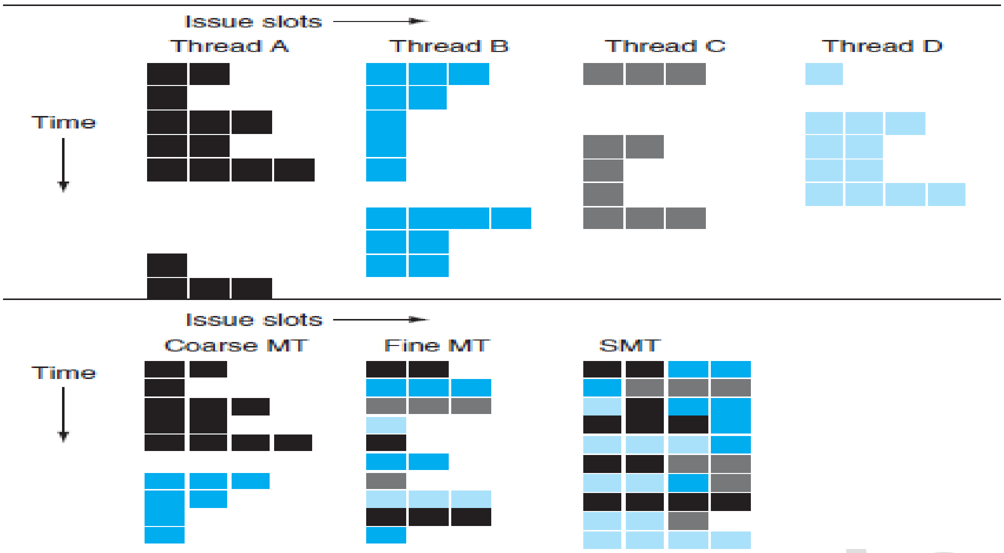
- It is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled pipelined processor to exploit thread-level parallelism at the same time it exploits instruction level parallelism.
- The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use.

- Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle.
- Instead, SMT is always executing instructions from multiple threads, to associate instruction slots and renamed registers with their proper threads.

Figure: How four threads use the issue slots of a superscalar processor in different approaches?


Issue slots
Thread A
Thread B
Thread C
Thread D
Time
Issue slots
Coarse - grained Multi Threading
Fine - grained Multi Threading
Simultaneous Multi Threading
Time

- The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support.
- The three examples at the bottom show how they would execute running together in



three multithreading options.

- 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 corresponding issue slot is unused in that clock cycle.
- The shades of gray and color correspond to four different threads in the multithreading processors.
- The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading.

Advantage

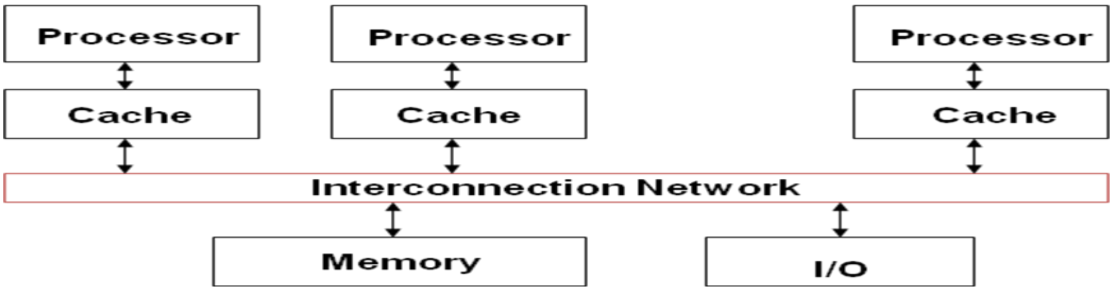
- It is ability to boost utilization by dynamically scheduling functional units among multiple threads.
- It increases hardware design facility.
- It produces better performance and add resources to a fine grained manner.

Disadvantage

It cannot improve performance if any of the shared resources are the limiting bottlenecks for the performance.

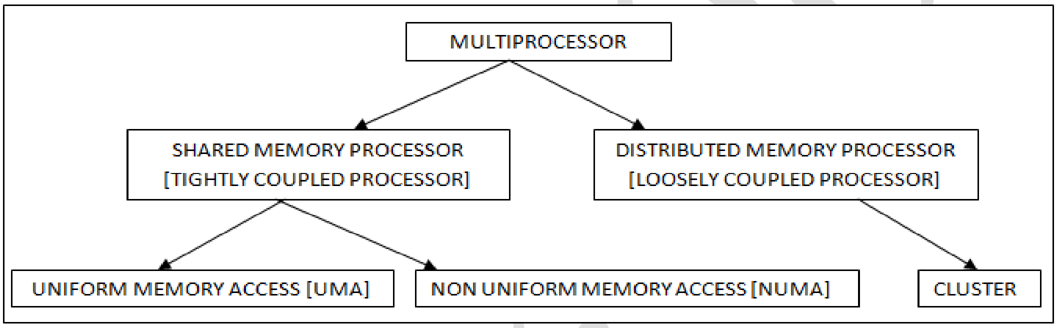
MULTICORE AND OTHER SHARED MEMORY MULTIPROCESSORS

- **Multiprocessor:** A computer system with at least two processors
- **Multicore:** More than one processor available within a single chip.



- The conventional multiprocessor system used is commonly referred as shared memory multiprocessor system.

- **Shared Memory Multiprocessor (SMP)** is one that offers the programmer a single physical address space across all processors which is nearly always the case for multicore chips.
- Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores.
- Systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.
- Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time


MULTIPROCESSOR
SHARED MEMORY PROCESSOR [TIGHTLY COUPLED PROCESSOR]
DISTRIBUTED MEMORY PROCESSOR [LOOSELY COUPLED PROCESSOR]
UNIFORM MEMORY ACCESS [UMA]
NON UNIFORM MEMORY ACCESS [NUMA]
CLUSTER

Shared Memory Multiprocessor System.[Tightly coupled processor]

- The conventional multiprocessor system used is commonly referred as shared memory multiprocessor system.
- Single address space shared by all processors. Because every processor communicates through a shared global memory.
- For high speed real time processing, these systems are preferable as their throughput is high as compared to loosely coupled systems
- In tightly coupled system organization, multiple processors share a global main memory, which may have many modules.
- Tightly coupled systems use a common bus, crossbar, or multistage network to connect processors, peripherals, and memories.

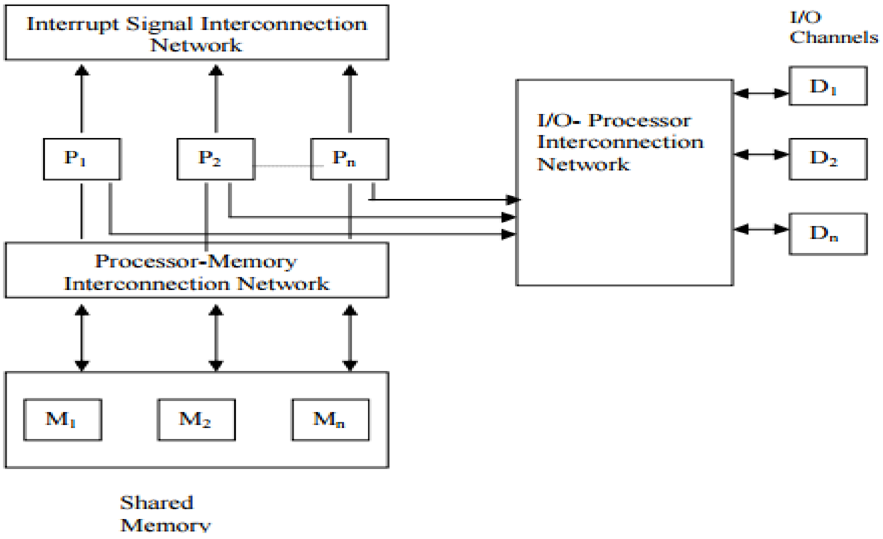
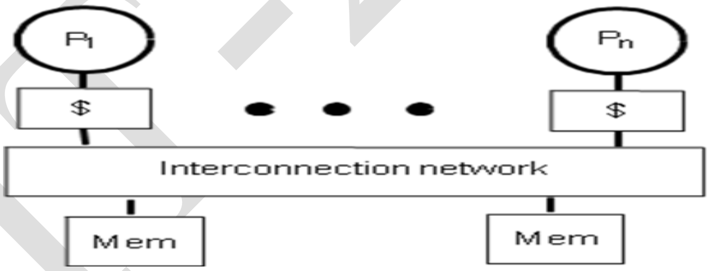

Interrupt Signal Interconnection Network
I/O Channels
P_1
P_2
P_n
I/O- Processor Interconnection Network
D_1
D_2
D_n
Processor-Memory Interconnection Network
M_1
M_2
M_n
Shared Memory

Figure : Tightly coupled system organization

Two common styles of implementing Shared Memory Multiprocessors (SMP) are,

Uniform memory access (UMA) multiprocessors

- In this model, main memory is uniformly shared by all processors in multiprocessor systems and each processor has equal access time to shared memory.
- This model is used for time-sharing applications in a multi user environment
- Tightly-coupled systems (high degree of resource sharing) suitable for general purpose and time-sharing applications by multiple users



- Physical memory uniformly shared by all processors, with equal access time to all words.
- Processors may have local cache memories. Peripherals also shared in some fashion.
- UMA architecture models are of two 20types,

Symmetric:

- All processors have equal access to all peripheral devices. All processors are identical.

Asymmetric:

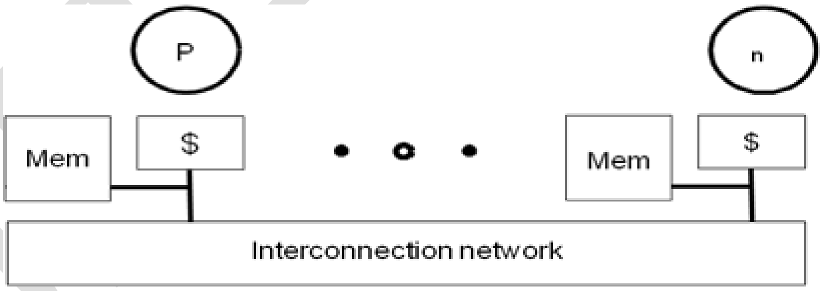
- One processor (master) executes the operating system other processors may be of different types and may be dedicated to special tasks.

Non Uniform Memory Access (NUMA) multiprocessors

- In shared memory multiprocessor systems, local memories can be connected with every processor. The collections of all local memories form the global memory being shared.
- In this way, global memory is distributed to all the processors. In this case, the access

to a local memory is uniform for its corresponding processor as it is attached to the local memory.

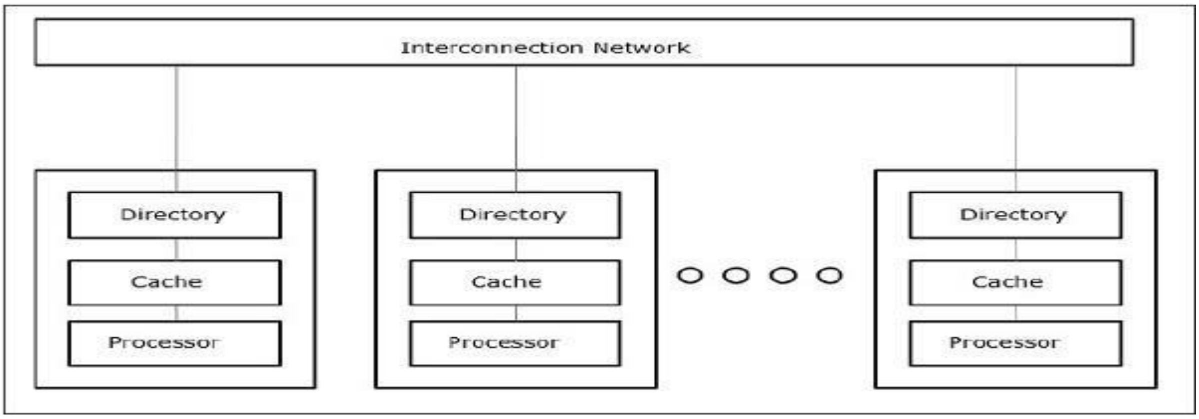
- But if one reference is to the local memory of some other remote processor, then the access is not uniform.
- It depends on the location of the memory. Thus, all memory words are not accessed uniformly. All local memories form a global address space accessible by all processors
- Programming NUMAs are harder but NUMAs can scale to larger sizes and have lower latency to local memory
- Memory is common to all the processors. Processors easily communicate by means of shared variables.
- These systems differ in how the memory and peripheral resources are shared or distributed
- The access time varies with the location of the memory word.
- The shared memory is distributed among the processors as local memories, but each of these is still accessible by all processors (with varying access times).
- Memory access is fastest from the locally –connected processor, with the interconnection network adding delays for other processor accesses.
- Additionally, there may be global memory in a multiprocessor system, with two separate interconnection networks, one for clusters of processors and their cluster memories, and another for the global shared memories.
- Local memories are private with its own program and data. No memory contention so that the number of processors is very large
- The processors are connected by communication lines, and the precise way in which the lines are connected is called the topology of the multicomputer.



Distributed Memory (NUMA)

<u>COMA:</u>

- Cache Only Memory Architecture. The COMA model is a special case of the NUMA model. Here all the distributed memories are converted to cache memories.
- The local memories for the processor at each node are used as cache instead of actual



memory.

<u>Distributed Memory [Loosely Coupled Systems]</u>

- These systems do not share the global memory because shared memory concept gives rise to the problem of memory conflicts, which in turn slows down the execution of instructions.
- Therefore, to alleviate this problem, each processor in loosely coupled systems is having a large local memory (LM), which is not shared by any other processor.
- Thus, such systems have multiple processors with their own local memory and a set of I/O devices.
- This set of processor, memory and I/O devices makes a computer system.

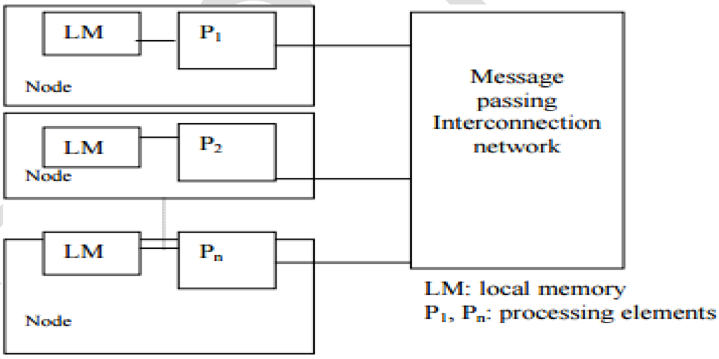
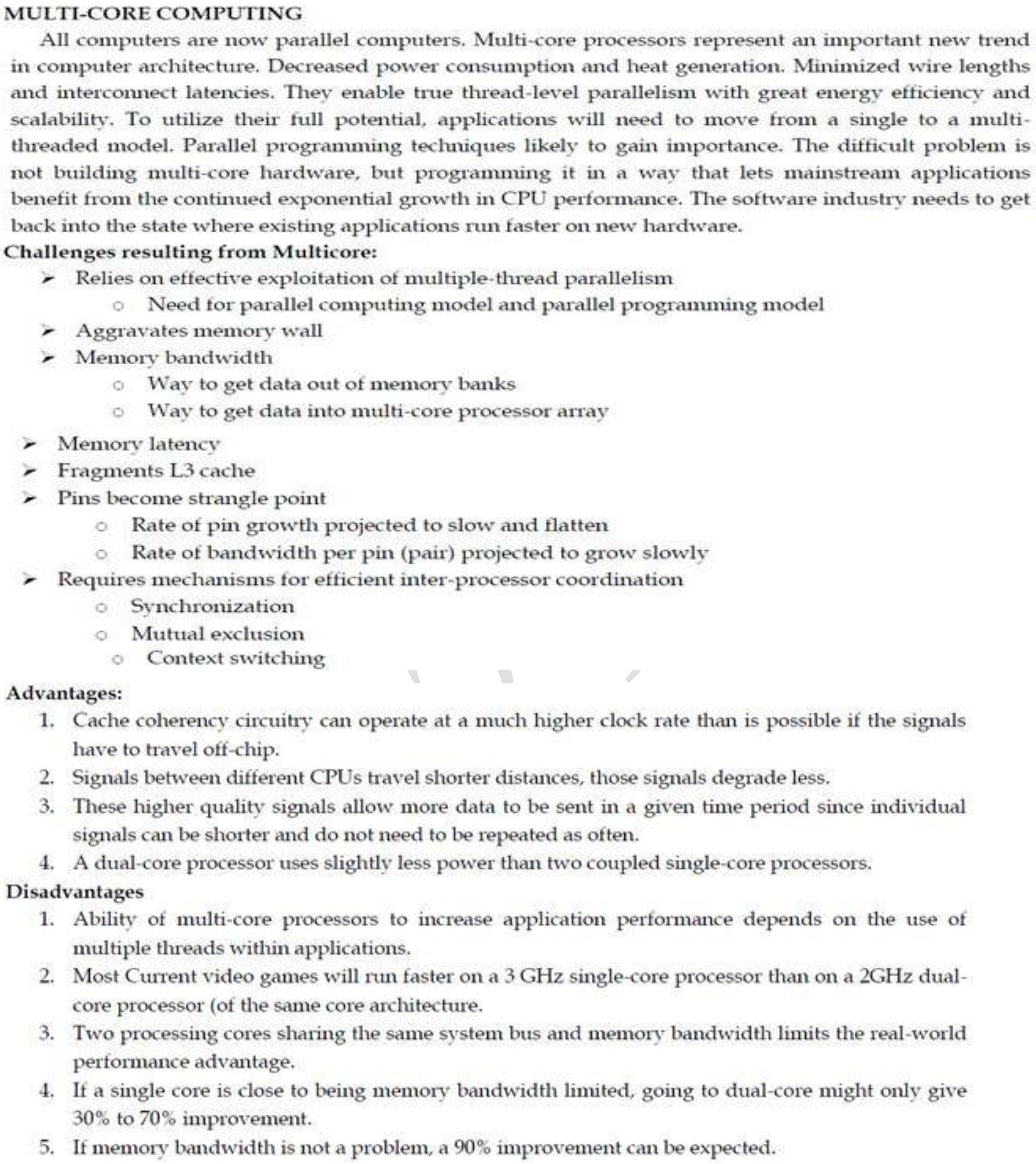

LM
P_1
Node
LM
P_2
Node
LM
P_n
Node
Message passing Interconnection network
LM: local memory
P_1, P_n: processing elements

Figure 14: Loosely coupled system organisation

- Therefore, these systems are also called multi-computer systems. These computer systems are connected together via message passing interconnection network through which processes communicate by passing messages to one another.
- Since every computer system or node in multicomputer systems has a separate memory, they are called distributed multicomputer systems. These are also called loosely coupled systems.
- **<u>Message passing:</u>** Communicating between multiple processors by explicitly sending and receiving information.
- **<u>Clusters:</u>** Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.
- **<u>Send message routine:</u>** A routine used by a processor in machines with private memories to pass a message to another processor.
- **<u>Receive message routine:</u>** A routine used by a processor in machines with private memories to accept a message from another processor.



MULTI-CORE COMPUTING

All computers are now parallel computers. Multi-core processors represent an important new trend in computer architecture. Decreased power consumption and heat generation. Minimized wire lengths and interconnect latencies. They enable true thread-level parallelism with great energy efficiency and scalability. To utilize their full potential, applications will need to move from a single to a multi-threaded model. Parallel programming techniques likely to gain importance. The difficult problem is not building multi-core hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance. The software industry needs to get back into the state where existing applications run faster on new hardware.

Challenges resulting from Multicore:

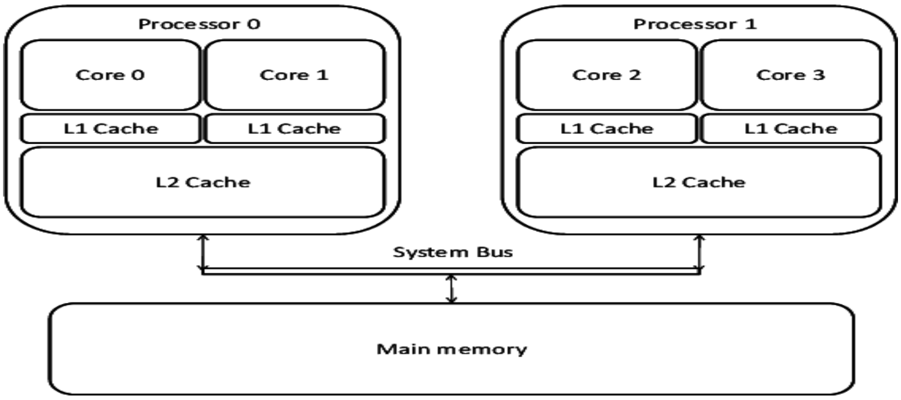
- Relies on effective exploitation of multiple-thread parallelism
  - Need for parallel computing model and parallel programming model
- Aggravates memory wall
- Memory bandwidth
  - Way to get data out of memory banks
  - Way to get data into multi-core processor array
- Memory latency
- Fragments L3 cache
- Pins become strangle point
  - Rate of pin growth projected to slow and flatten
  - Rate of bandwidth per pin (pair) projected to grow slowly
- Requires mechanisms for efficient inter-processor coordination
  - Synchronization
  - Mutual exclusion
  - Context switching

Advantages:

1. Cache coherency circuitry can operate at a much higher clock rate than is possible if the signals have to travel off-chip.
2. Signals between different CPUs travel shorter distances, those signals degrade less.
3. These higher quality signals allow more data to be sent in a given time period since individual signals can be shorter and do not need to be repeated as often.
4. A dual-core processor uses slightly less power than two coupled single-core processors.

Disadvantages

1. Ability of multi-core processors to increase application performance depends on the use of multiple threads within applications.
2. Most Current video games will run faster on a 3 GHz single-core processor than on a 2GHz dual-core processor (of the same core architecture.
3. Two processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage.
4. If a single core is close to being memory bandwidth limited, going to dual-core might only give 30% to 70% improvement.
5. If memory bandwidth is not a problem, a 90% improvement can be expected.


Processor 0
Core 0
Core 1
L1 Cache
L1 Cache
L2 Cache
Processor 1
Core 2
Core 3
L1 Cache
L1 Cache
L2 Cache
System Bus
Main memory