RAMAIAH
Institute of Technology

Multi core Architecture

Code:CS72 Credits:3:1:0:0

Incharge

Mallegowda M

Syllabus

- **UNIT I**

  Introduction to High–Performance Computers, Memory Hierarchy, CPU Design: Reduced Instruction Set Computers, Multiple–Core Processors, Vector Processors, **Self study** **Parallel Semantics, Distributed Memory Programming**

- **UNIT II**

  Programming Shared Address Space Platforms: Thread Basics, Why Threads? The POSIX Thread API, Thread Creation and Termination, Synchronization Primitives in Pthreads, Controlling Thread and Synchronization Attributes, Thread Cancellation, Composite Synchronization Constructs, **Self study** **Tips for Designing Asynchronous Programs, OpenMP: a Standard for Directive Based Parallel Programming**

- **UNIT III**

  Programming using the Message-Passing Paradigm: Principles of Message-Passing Programming, The Building Blocks: Send and Receive Operations, MPI: the Message Passing Interface, Topologies and Embedding, Overlapping Communication with Computation, Collective Communication and Computation Operations**, Self study Groups and Communicators**.

- **UNIT IV**

  Introduction: GPUs as Parallel Computers, Architecture of a Model GPU, Why More Speed or Parallelism? Parallel Programming Languages and Models, Overarching Goals. History of GPU Computing: Evolution of Graphics Pipelines, GPU Computing. Introduction to CUDA: Data Parallelism, CUDA Program Structure, A Matrix-Matrix Multiplication Example, Device Memories and Data Transfer, **Self study Kernel Functions and Threading**.

- **Unit V**

  CUDA Threads: CUDA Thread Organization, Using blockIdx and threadIdx, Synchronization and Transparent Scalability, Thread Assignment, Thread Scheduling and Latency Tolerance. CUDA Memories: Importance of Memory Access Efficiency, CUDA Device Memory Types, A Strategy for Reducing Global Memory Traffic, **Self study Memory as a limiting Factor to Parallelism**. Performance Considerations: More on Thread Execution, Global Memory Bandwidth, Dynamic Partitioning of SM Resources, Data Perfecting, Instruction Mix, Thread Granularity, Measured Performance and Summary.

TEXT BOOKS

1. Rubin H Landau, Oregon State University, http://science.oregonstate.edu/ rubin/
2. Ananth Grama,Anshul Gupta,Vipin kumar, George Karypis Introduction to parallel computing, second edition, Pearson education publishers.
3. David B Kirk, Wen-mei W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach”, First Edition, Elsevier and nvidia Publishers, 2010.

REFERENCE BOOKS

1. Thomas Rauber and Gudula Runger Parallel Programming for Multicore and cluster systems,Springer International Edition, 2009 .
2. Hennessey and Patterson Computer Architecture: A quantitative Approach, Morgan Kaufman Publishers
3. Michael J.Quin “Parallel Programming in C with MPI and Open MP”, McGraw Hill.

Course outcomes

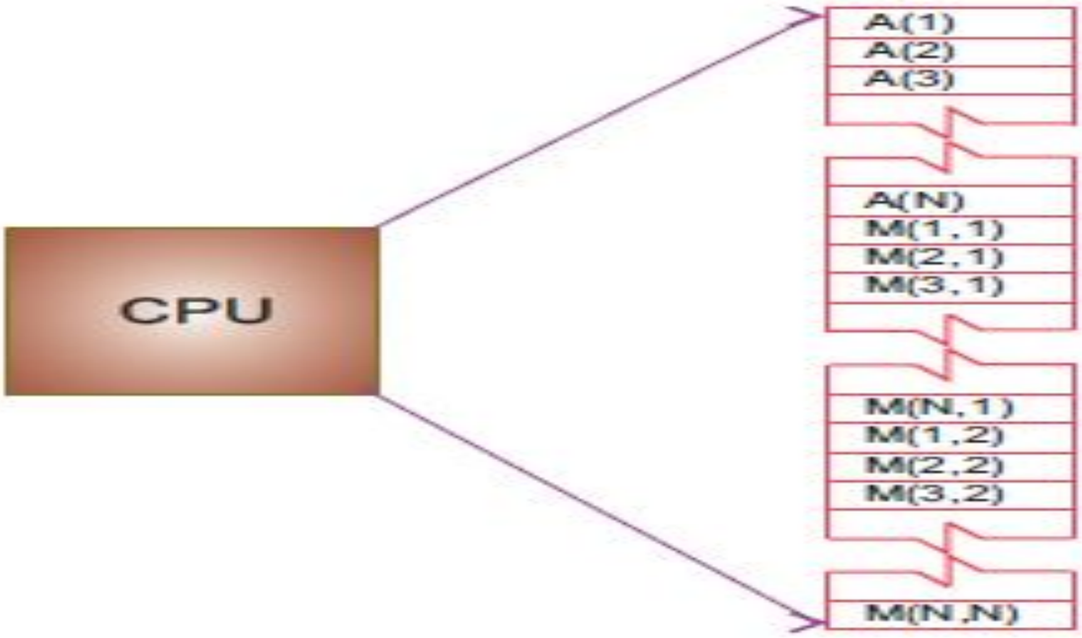
1. Explain the technologies and architectures used for parallel computing.
2. Design and develop parallel programs using OpenMP programming interface.
3. Elaborate the principles and architecture of message-passing programming paradigm for solving real world problems.
4. Provide an understanding of Graphical Processing Units and their architecture.
5. Analyze the features of GPUs, their functionalities and also Design parallel applications using CUDA-C.

Definition

- supercomputers are the fastest and most powerful computers available, and at present the term refers to machines with hundreds of thousands of processors.
- They are the super-stars of the high performance class of computers. Personal computers (PCs) small enough in size and cost to be used by an individual, yet powerful enough for advanced scientific and engineering applications, can also be high performance computers.

- High performance computers as machines with a good balance among the following major elements:
- Multistaged (pipelined) functional units.
- Multiple central processing units (CPUs) (parallel machines).
- Multiple cores.
- Fast central registers.
- Very large, fast memories.
- Very fast communication among functional units.
- Vector, video, or array processors.
- Software that integrates the above effectively.

Figure 1: The logical arrangement of the CPU and memory showing a Fortran array A(N) and matrix M(N, N) loaded into memory.



Memory Hierarchy

- An idealized model of computer architecture is a CPU sequentially executing a stream of instructions and reading from a continuous block of memory. To illustrate, in Figure 1 we see a vector A[ ] and an array M[ ][ ] loaded in memory and about to be processed. The real world is more complicated than this.
- First, matrices are not stored in blocks but rather in linear order. For instance, in Fortran it is in column major order:

M(1; 1)M(2; 1)M(3; 1)M(1; 2)M(2; 2)M(3; 2)M(1; 3)M(2; 3) M(3; 3);

while in Python, Java and C it is in row major order:

M(0; 0)M(0; 1)M(0; 2)M(1; 0)M(1; 1)M(1; 2)M(2; 0)M(2; 1) M(2; 2):

Second, as illustrated in Figures 2 and 3, the values for the matrix elements may not even be in the same physical place. Some may be in RAM, some on the disk, some in cache, and some in the CPU.

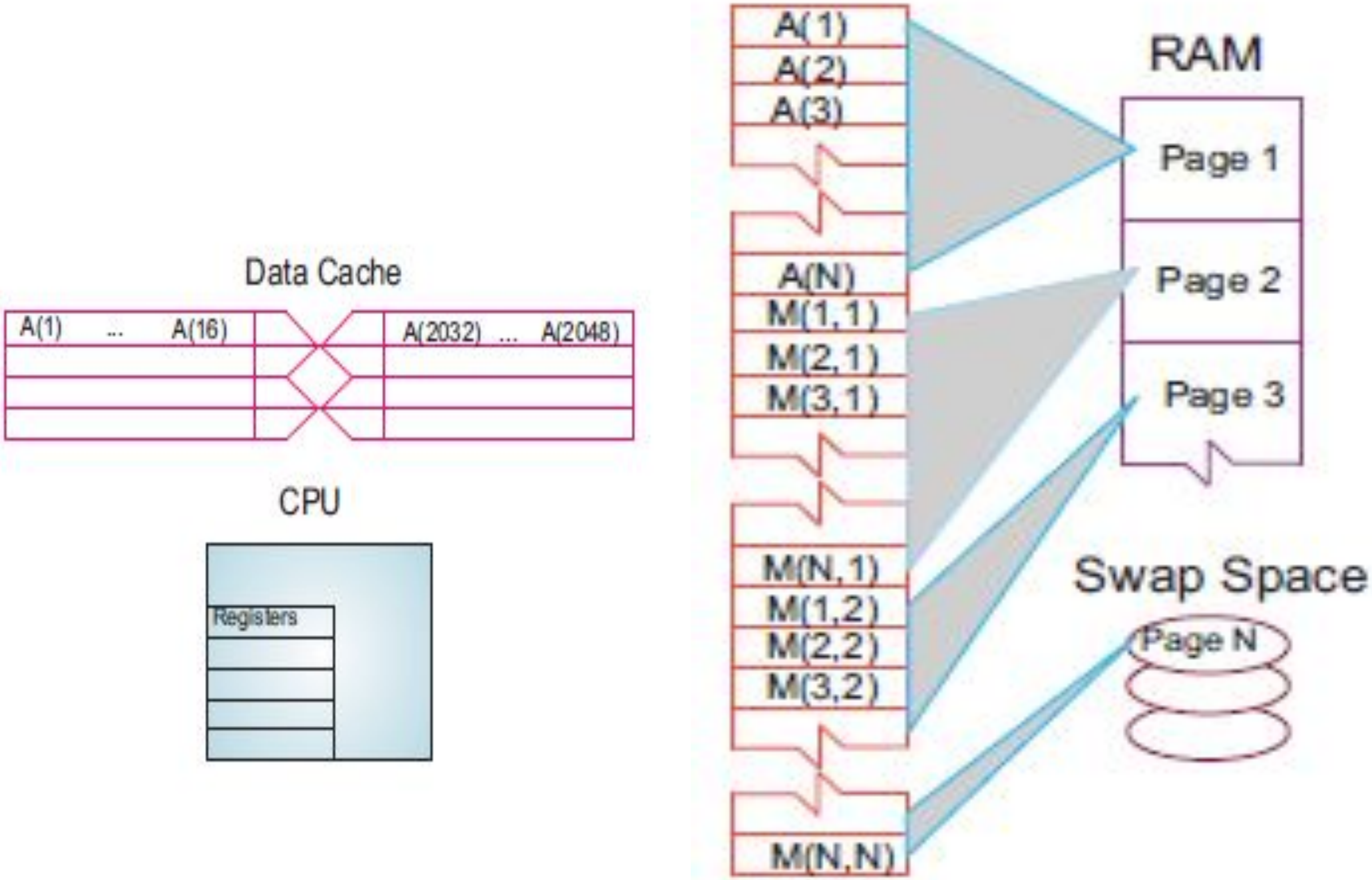
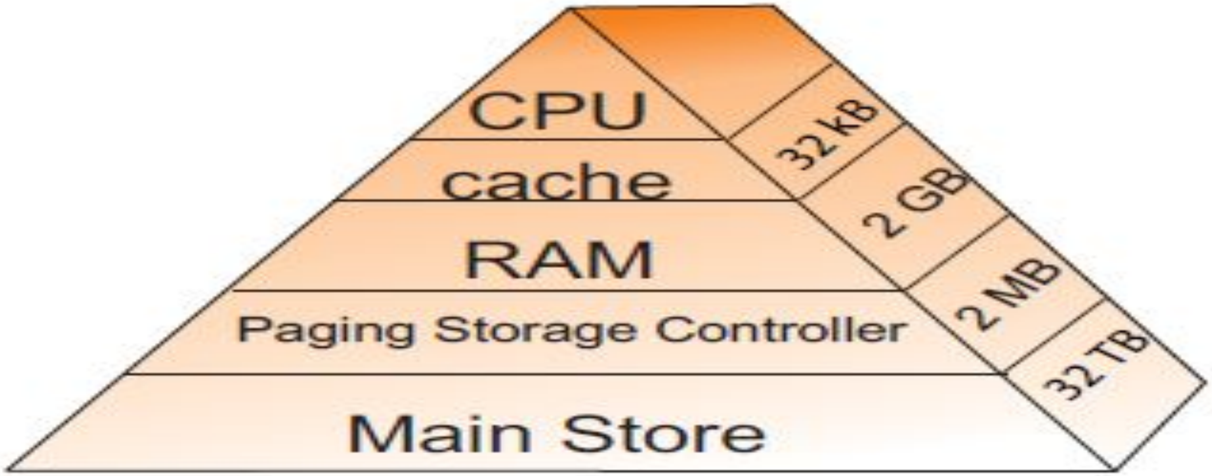


Figure 3: Typical memory hierarchy for a single processor, high performance computer (B = bytes, K, M, G, T = kilo, mega, giga, tera).



CPU

- Central processing unit, the fastest part of the computer. The CPU consists of a number of very high speed memory units called registers containing the instructions sent to the hard-ware to do things like fetch, store, and operate on data. There are usually separate registers for instructions, addresses, and operands (current data). In many cases the CPU also contains some specialized parts for accelerating the processing of floating point numbers

Cache

A small, very fast bit of memory that holds instructions, addresses, and data in their passage between the very fast CPU registers and the slower RAM. (Also called a high speed bues.) This is seen in the next level down the pyramid in Figure 3. The main memory is also called dynamic RAM (DRAM), while the cache is called static RAM (SRAM). If the cache is used properly, it can greatly reduce the time that the CPU waits for data to be fetched from memory.

RAM

Random access or central memory is in the middle of the memory hierarchy in Figure 3. RAM is fast because its addresses can be accessed directly in random order, and because no mechanical devices are needed to read it.

Pages

Central memory is organized into pages, which are blocks of memory of fixed length. The operating system labels and organizes its memory pages much like we do the pages of a book; they are numbered and kept track of with a table of contents. Typical page sizes are from 4 to 16 kB, but on supercomputers may be in the MB range.

Hard disk

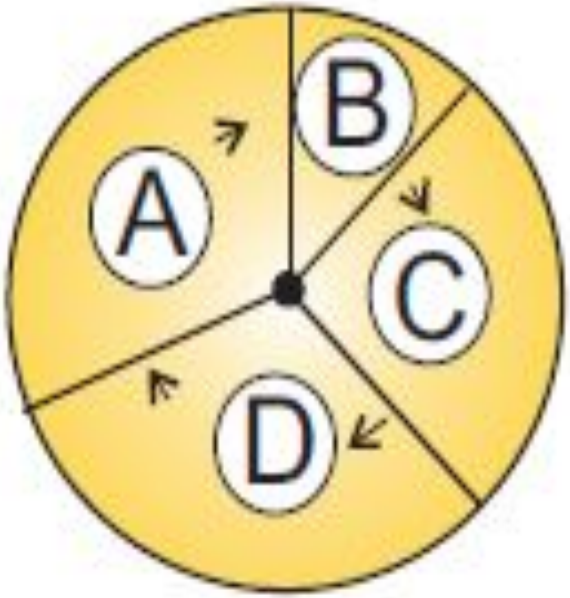
Finally, at the bottom of the memory pyramid is permanent storage on magnetic disks or optical devices. Although disks are very slow compared to RAM, they can store vast amounts of data and sometimes compensate for their slower speeds by using a cache of their own, the paging storage controller.

Virtual memory

- True to its name, this is a part of memory you will not find in our figures because it is virtual. It acts like RAM but resides on the disk

- Virtual memory also allows multitasking, the simultaneous loading into memory of more programs than can physically fit into RAM (Figure 4). Although the ensuing switching among applications uses computing cycles, by avoiding long waits while an application is loaded into memory, multitasking increases the total throughout and permits an improved computing environment for users. For example, it is multitasking that permits a windowing system, such as Linux, OS X or Windows, to provide us with multiple windows. Even though each window application uses a fair amount of memory, only the single application currently receiving input must actually reside in memory the rest are paged out to disk.

Figure 4: Multitasking of four programs in memory at one time in which the programs are executed in round robin order.



- How does the CPU get to be so fast?

Pre-fetching and pipelining that is, it has the ability to prepare for the next instruction before the current one has finished. It is like an assembly line or a bucket brigade in which the person filling the buckets at one end of the line does not wait for each bucket to arrive at the other end before filling another bucket. In the same way a processor fetches, reads, and decodes an instruction while another instruction is executing.

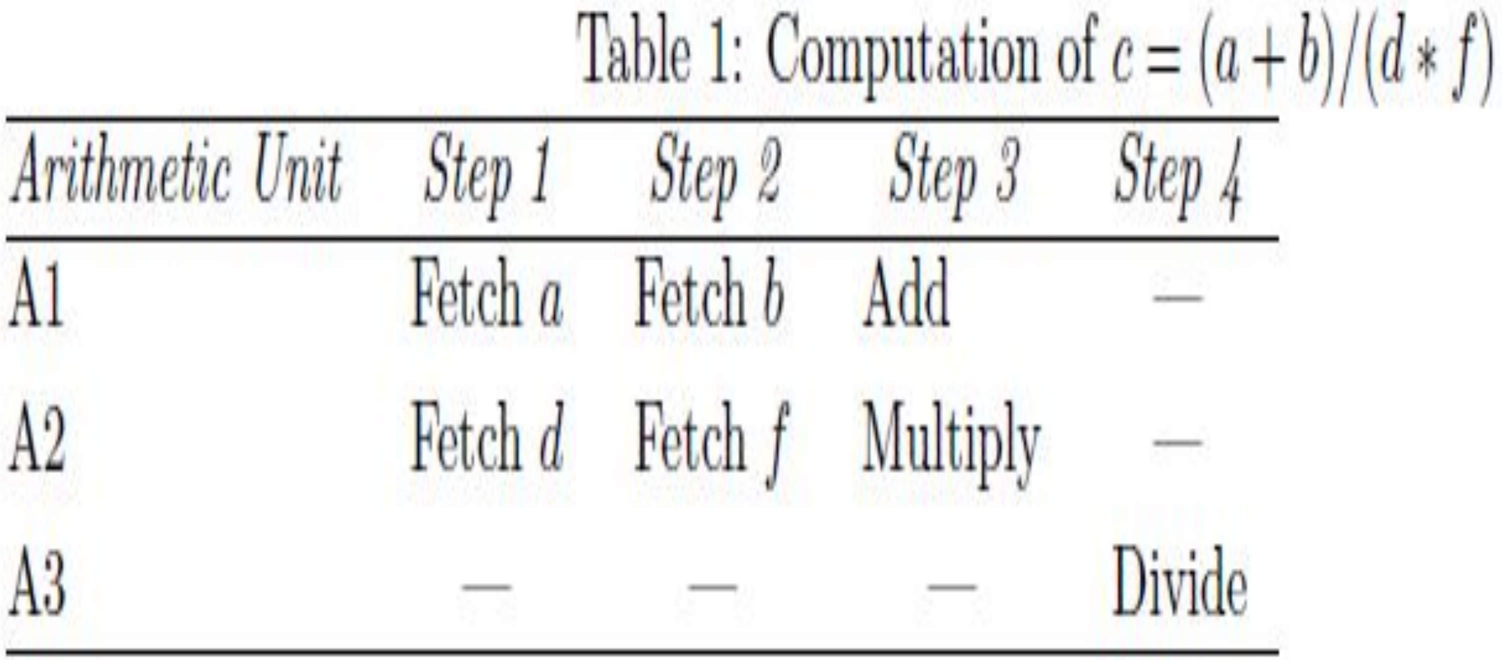


Table 1: Computation of $c = (a+b)/(d*f)$

Arithmetic Unit	*Step 1*	*Step 2*	*Step 3*	*Step 4*
A1	Fetch a	Fetch b	Add	—
A2	Fetch d	Fetch f	Multiply	—
A3	—	—	—	Divide

CPU Design

- Reduced instruction set computer (RISC) architecture
- Very Long Instruction Word (VLIW) Processors
- Multiple Core Processors
- Vector Processors

Reduced instruction set computer (RISC) architecture

- Reduced instruction set computer (RISC) architecture (also called superscalar) is a design philosophy for CPUs developed for high-performance computers and now used broadly. It increases the arithmetic speed of the CPU by decreasing the number of instructions the CPU must follow. To understand RISC we contrast it with complex instruction set computer (CISC), architecture. In the late 1970s, processor designers began to take advantage of very-large-scale integration (VLSI), which allowed the placement of hundreds of thousands of devices on a single CPU chip. Much of the space on these early chips was dedicated to microcode programs written by chip designers and containing machine language instructions that set the operating characteristics of the computer.

- CPU time = number of instructions *cycles/instruction *cycle time:

In summary, the elements of RISC are the following:

Single cycle execution, for most machine level instructions.

Small instruction set, of less than 100 instructions.

Register based instructions, operating on values in registers, with memory access conned to loading from and storing to registers.

Many registers, usually more than 32.

Pipelining, concurrent preparation of several instructions that are then executed successively.

High level compilers, to improve performance.

Pipelining and Superscalar Execution

- Pipelining overlaps various stages of instruction execution to achieve performance.
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- This is akin to an assembly line for manufacture of cars.

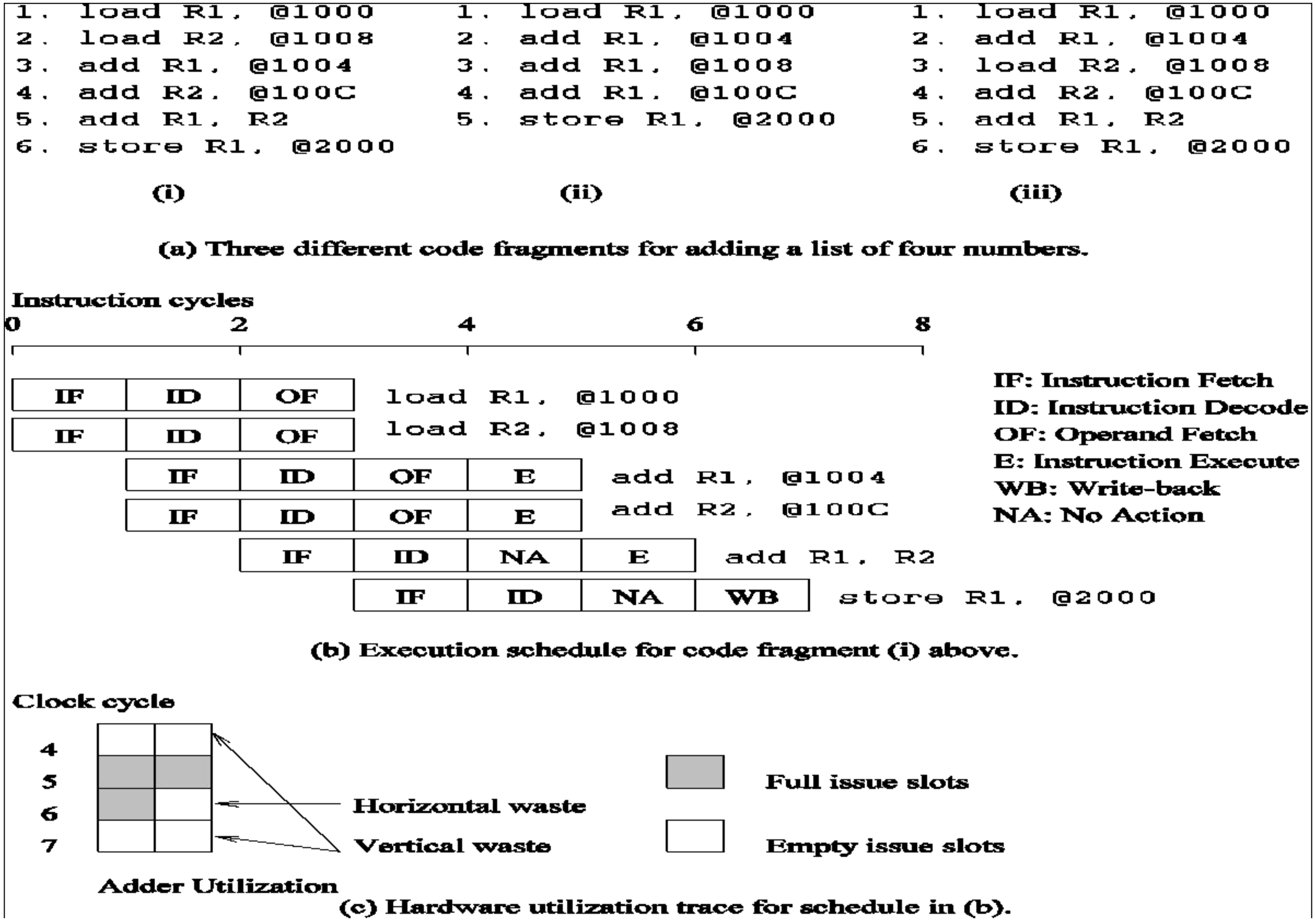
Pipelining and Superscalar Execution

- Pipelining, however, has several limitations.
- The speed of a pipeline is eventually limited by the slowest stage.
- For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in state-of-the-art Pentium processors).
- However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.
- The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

Pipelining and Superscalar Execution

- One simple way of alleviating these bottlenecks is to use multiple pipelines.
- The question then becomes one of selecting these instructions.

Superscalar Execution: An Example



```
1.  load R1, @1000
2.  load R2, @1008
3.  add R1, @1004
4.  add R2, @100C
5.  add R1, R2
6.  store R1, @2000
```

(i)

```
1.  load R1, @1000
2.  add R1, @1004
3.  add R1, @1008
4.  add R1, @100C
5.  store R1, @2000
```

(ii)

```
1.  load R1, @1000
2.  add R1, @1004
3.  load R2, @1008
4.  add R2, @100C
5.  add R1, R2
6.  store R1, @2000
```

(iii)

(a) Three different code fragments for adding a list of four numbers.

(b) Execution schedule for code fragment (i) above.

(c) Hardware utilization trace for schedule in (b).

Example of a two-way superscalar execution of instructions.

Superscalar Execution: An Example

- In the above example, there is some wastage of resources due to data dependencies.
- The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.

Superscalar Execution

- Scheduling of instructions is determined by a number of factors:
  - True Data Dependency: The result of one operation is an input to the next.
  - Resource Dependency: Two operations require the same resource.
  - Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.
  - The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
  - The complexity of this hardware is an important constraint on superscalar processors.

Superscalar Execution: Issue Mechanisms

- In the simpler model, instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called *in-order* issue.
- In a more aggressive model, instructions can be issued out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.
- Performance of in-order issue is generally limited.

Superscalar Execution: Efficiency Considerations

- Not all functional units can be kept busy at all times.
- If during a cycle, no functional units are utilized, this is referred to as vertical waste.
- If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.
- Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.

Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
- These instructions are packed and dispatched together, and thus the name very long instruction word.
- This concept was used with some commercial success in the Multiflow Trace machine (circa 1984).
- Variants of this concept are employed in the Intel IA64 processors.

Very Long Instruction Word (VLIW) Processors: Considerations

- Issue hardware is simpler.
- Compiler has a bigger context from which to select co-scheduled instructions.
- Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.
- Branch and memory prediction is more difficult.
- VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
- Typical VLIW processors are limited to 4-way to 8-way parallelism.

Limitations of Memory System Performance

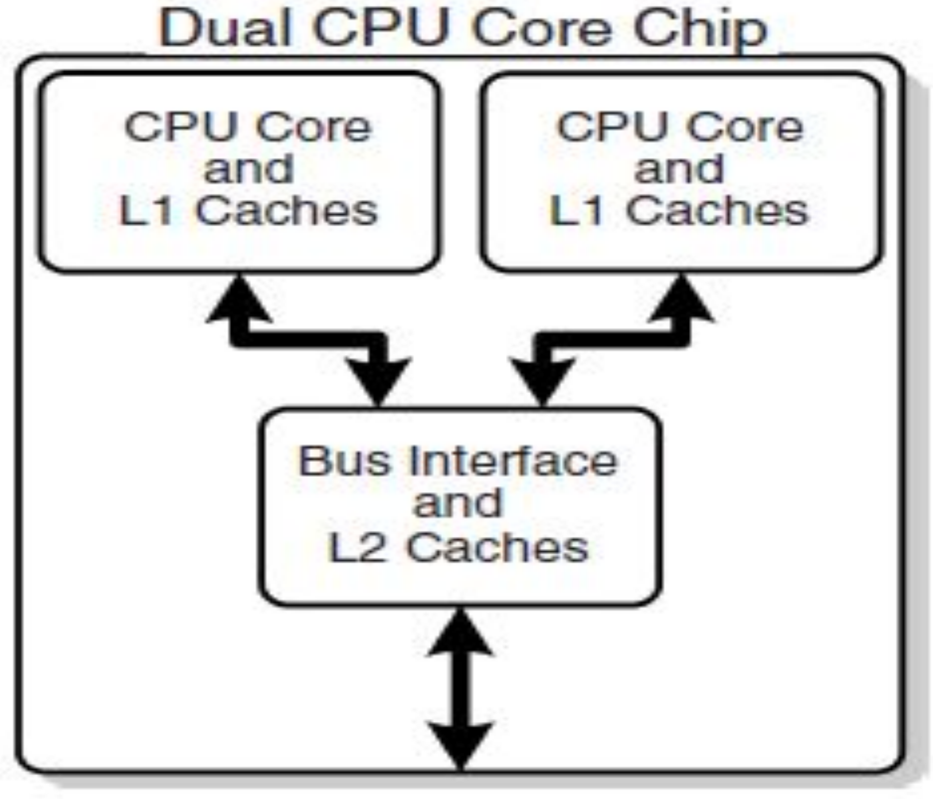
- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Multiple-Core Processors

The present time is seeing a rapid increase in the inclusion of dual core, quad core, or even sixteen core chips as the computational engine of computers. As seen in Figure 5, a dual core chip has two CPUs in one integrated circuit with a shared interconnect and a shared level2 cache. This type of configuration with two or more identical processors connected to a single shared main memory is called symmetric multiprocessing, or SMP.

- multicore chips were designed for game playing and single precision, they are finding use in scientific computing as new tools, algorithms, and programming methods are employed.
- These chips attain more integrated speed with less heat and more energy efficiency than single corechips, whose heat generation limits them to clock speeds of less than 4 GHz.
- In contrast to multiple single core chips, multicore chips use fewer transistors per CPU and are thus simpler to make and cooler to run.

Figure 5: A generic view of the Intel core-2 dual core processor, with CPU-local level-1 caches and a shared, on-die level-2 cache



Vector Processors

- On a classic (von Neumann) scalar computer, the addition of two vectors of physical length 99 to form a third ultimately requires 99 sequential additions (Table 2). There is actually much behind the scenes
- work here. For each element i there is the fetch of a(i) from its location in memory, the fetch of b(i) from its location in memory, the addition of the numerical values of these two elements in a CPU register, and the storage in memory of the sum in c(i). This fetching uses up time and is wasteful in the sense that the computer is being told again and again to do the same thing.

Table 2: Computation of Matrix $[C]=[A]+[B]$

Step 1	*Step 2*	$\cdots$	*Step 99*
$c(1)=a(1)+b(1)$	$c(2)=a(2)+b(2)$	$\cdots$	$c(99)=a(99)+b(99)$

Table 3: Vector Processing of Matrix $[A]+[B]=[C]$

Step 1	Step 2	$\cdots$	Step Z
$c(1)=a(1)+b(1)$			
	$c(2)=a(2)+b(2)$		
		$\cdots$	
			$c(Z)=a(Z)+b(Z)$

Introduction to Parallel Computing

- In our view, message passing has too many details for application scientists to worry about and (unfortunately) requires coding at an elementary level reminiscent of the early days of computing. However, the increasing occurrence of clusters in which the nodes are symmetric multiprocessors has led to the development of sophisticated compilers that follow simpler programming models; for example, par-titioned global address space compilers such as Co Array Fortran, Unified Parallel C, and Titanium.
- In these approaches the programmer views a global array of data and then manipulates these data as if they were contiguous.

Parallel Semantics (Theory)

- Single instruction, single data (SISD): These are the classic (von Neumann) serial computers executing a single instruction on a single data stream before the next instruction and next data stream are encountered.
- Single instruction, multiple data (SIMD): Here instructions are processed from a single stream, but the instructions act concurrently on multiple data elements. Generally the nodes are simple and relatively slow but are large in number.

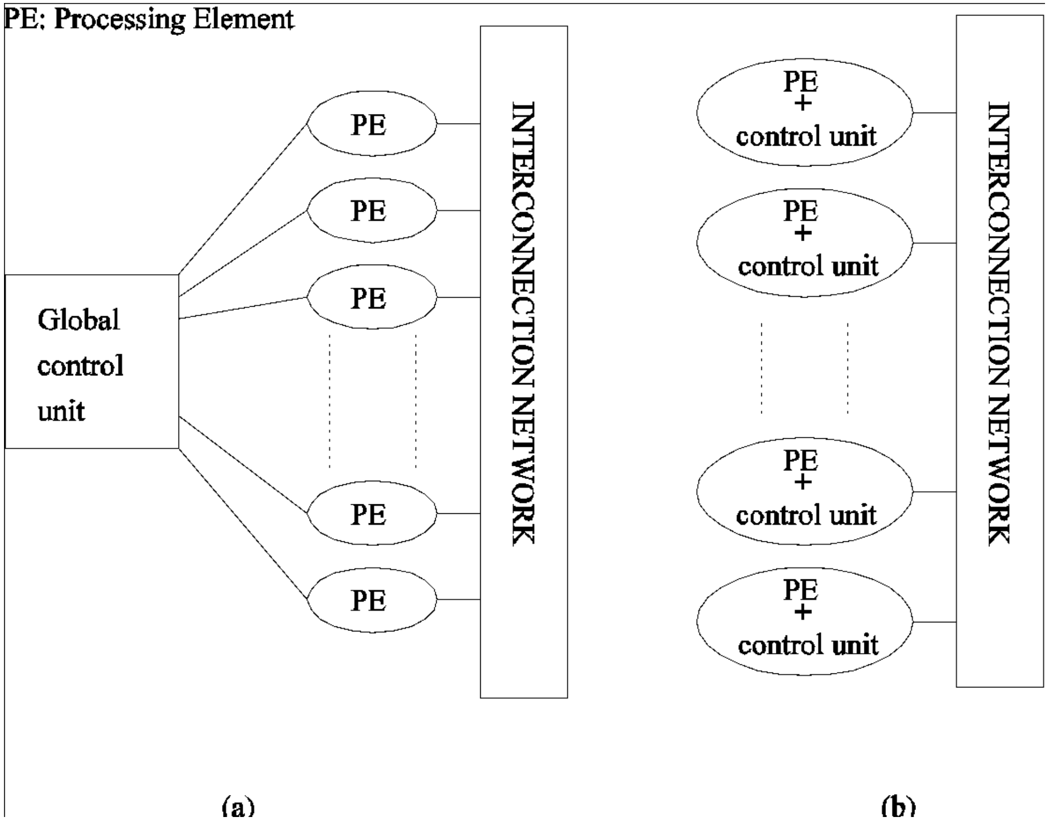
MIMD

- Multiple instructions, multiple data (MIMD): In this category each processor runs independently of the others with independent instructions and data. These are the types of machines that employ message{passing packages, such as MPI, to communicate among processors. They may be a collection of workstations linked via a network, or more integrated machines with thousands of processors on internal boards, such as the Blue Gene computer described in x16. These computers, which do not have a shared memory space, are also called multicomputers. Although these types of computers are some of the most difficult to program, their low cost and effectiveness for certain classes of problems have led to their being the dominant type of parallel computer at present.

Control Structure of Parallel Programs

- Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.
- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).
- If each processor has its own control control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).

SIMD and MIMD Processors

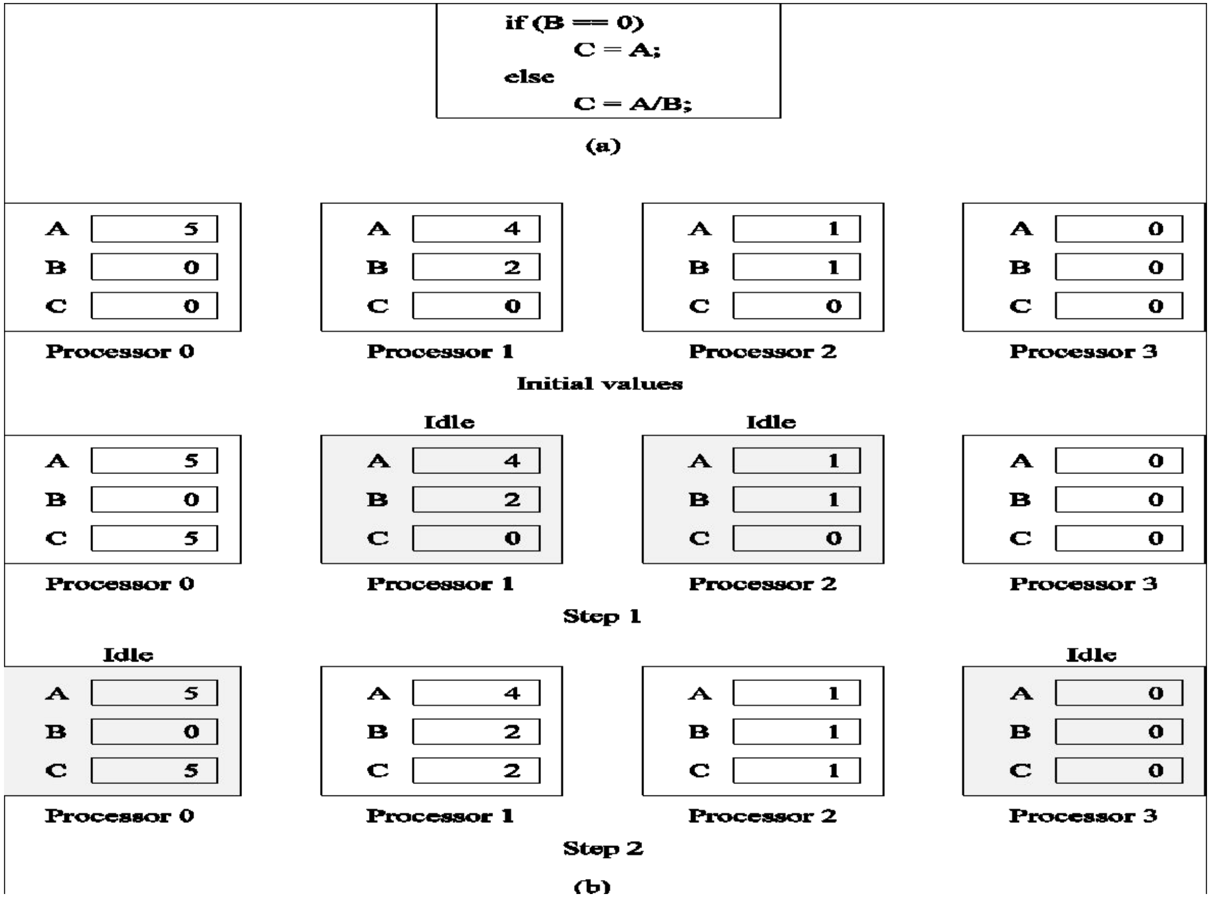


A typical SIMD architecture (a) and a typical MIMD architecture (b).

SIMD Processors

- Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines.
- Variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc.
- SIMD relies on the regular structure of computations (such as those in image processing).
- It is often necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an ``activity mask'', which determines if a processor should participate in a computation or not.

Conditional Execution in SIMD Processors



```
if (B == 0)
        C = A;
else
        C = A/B;
```

(a)

	Processor 0	Processor 1	Processor 2	Processor 3
A	5	4	1	0
B	0	2	1	0
C	0	0	0	0

Initial values

	Processor 0	Processor 1 (Idle)	Processor 2 (Idle)	Processor 3
A	5	4	1	0
B	0	2	1	0
C	5	0	0	0

Step 1

	Processor 0 (Idle)	Processor 1	Processor 2	Processor 3 (Idle)
A	5	4	1	0
B	0	2	1	0
C	5	2	1	0

Step 2

(b)

Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

MIMD Processors

- In contrast to SIMD processors, MIMD processors can execute different programs on different processors.
- A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors.
- It is easy to see that SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support.
- Examples of such platforms include current generation Sun Ultra Servers, SGI Origin Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles.
- Not all applications are naturally suited to SIMD processors.
- In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time.

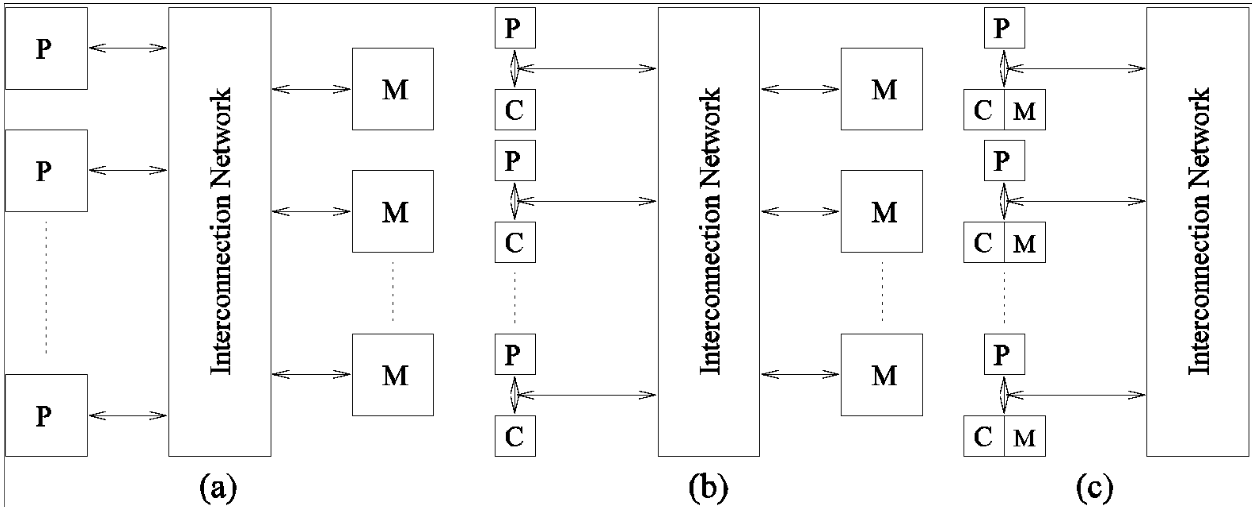
Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks - accessing a shared data space and exchanging messages.
- Platforms that provide a shared data space are called shared-address-space machines or multiprocessors.
- Platforms that support messaging are also called message passing platforms or multicomputers.

Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system global or local is identical, the platform is classified as a uniform memory access (UMA), else, a non-uniform memory access (NUMA) machine.

NUMA and UMA Shared-Address-Space Platforms



Typical shared-address-space architectures: (a) Uniform-memory access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

NUMA and UMA Shared-Address-Space Platforms

- The distinction between NUMA and UMA platforms is important from the point of view of algorithm design. NUMA machines require locality from underlying algorithms for performance.
- Programming these platforms is easier since reads and writes are implicitly visible to other processors.
- However, read-write data to shared data must be coordinated (this will be discussed in greater detail when we talk about threads programming).
- Caches in such machines require coordinated access to multiple copies. This leads to the cache coherence problem.
- A weaker model of these machines provides an address map, but not coordinated access. These models are called non cache coherent shared address space machines.

Shared-Address-Space vs. Shared Memory Machines

- It is important to note the difference between the terms shared address space and shared memory.
- We refer to the former as a programming abstraction and to the latter as a physical machine attribute.
- It is possible to provide a shared address space using a physically distributed memory.

Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
- Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.
- These platforms are programmed using (variants of) send and receive primitives.
- Libraries such as MPI and PVM provide such primitives.

Message Passing vs. Shared Address Space Platforms

- Message passing requires little hardware support, other than a network.
- Shared address space platforms can easily emulate message passing. The reverse is more difficult to do (in an efficient manner).

Granularity

- A grain is defined as a measure of the computational work to be done, more specifically, the ratio of computation work to communication work.
- Coarse grain parallel
- Medium grain parallel
- Fine grain parallel

Coarse grain parallel

- Separate programs running on separate computer systems with the sys- tems coupled via a conventional communication network. An illustration is six Linux PCs sharing the same files across a network but with a different central memory system for each PC. Each computer can be operating on a different, independent part of one problem at the same time.

Medium grain parallel

- Several processors executing (possibly different) programs simultaneously while accessing a common memory. The processors are usually placed on a common bus (communication channel) and communicate with each other through the memory system.
- Medium grain programs have different, independent, parallel subroutines running on different processors. Because the compilers are seldom smart enough to figure out which parts of the program to run where, the user must include the multitasking routines in the program.

Fine grain parallel:

- As the granularity decreases and the number of nodes increases, there is an increased requirement for fast communication among the nodes. For this reason fine grain systems tend to be custom designed machines. The communication may be via a central bus or via shared memory for a small number of nodes, or through some form of high speed network for massively parallel machines. In the latter case, the user typically divides the work via certain coding constructs, and the compiler just compiles the program. The program then runs concurrently on a user specified number of nodes.
- For example, different for loops of a program may be run on different nodes.

Classes of Parallelism and Parallel Architectures

Parallelism at multiple levels is now the driving force of computer design across all four classes of computers, with energy and cost being the primary constraints. There are basically two kinds of parallelism in applications:

1. *Data-Level Parallelism (DLP)* arises because there are many data items that can be operated on at the same time.
2. *Task-Level Parallelism (TLP)* arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

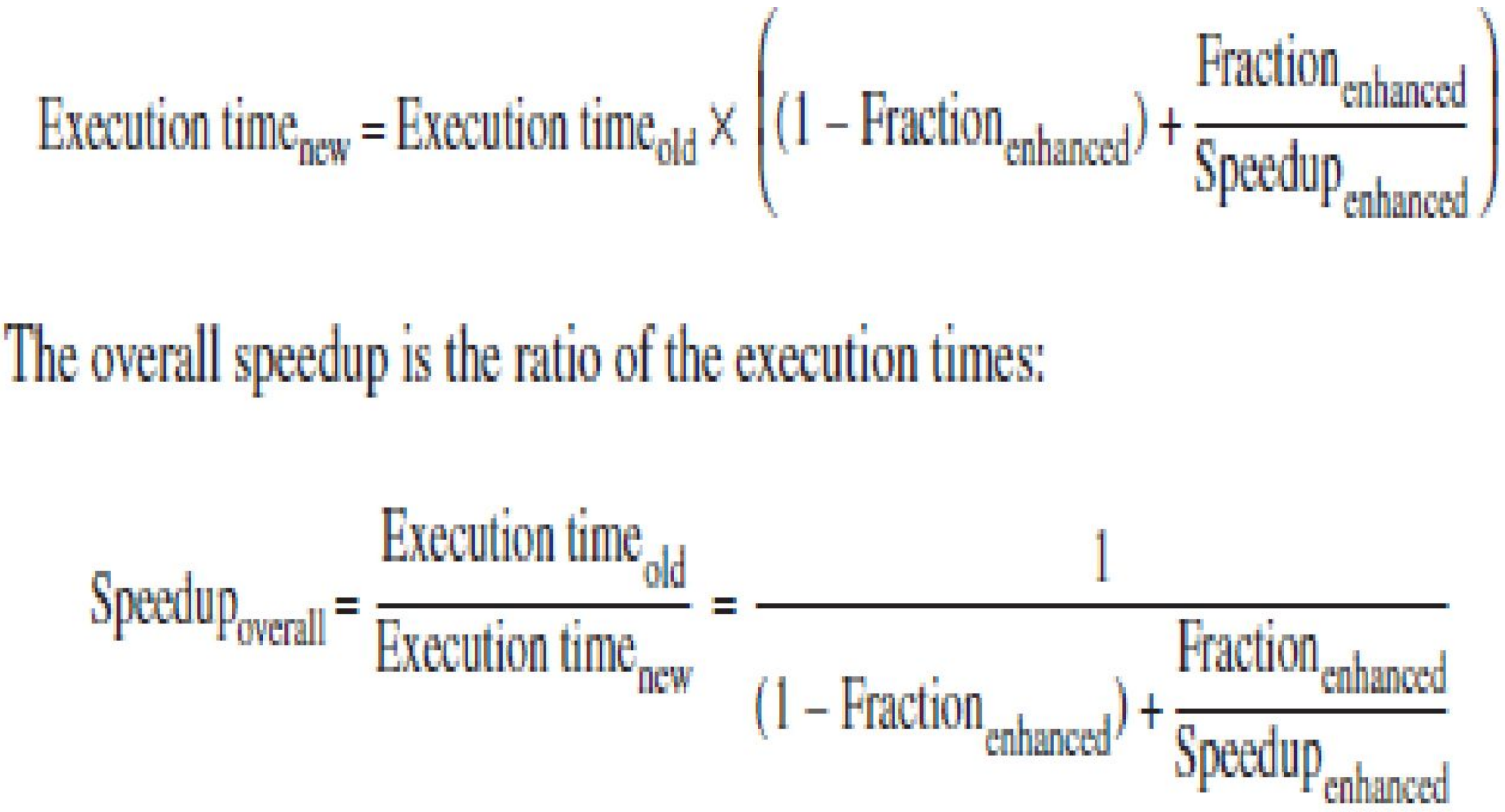
1. *Instruction-Level Parallelism* exploits data-level parallelism at modest levels with compiler help using ideas like pipelining and at medium levels using ideas like speculative execution.
2. *Vector Architectures* and *Graphic Processor Units (GPUs)* exploit data-level parallelism by applying a single instruction to a collection of data in parallel.
3. *Thread-Level Parallelism* exploits either data-level parallelism or task-level parallelism in a tightly coupled hardware model that allows for interaction among parallel threads.
4. *Request-Level Parallelism* exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

Amdahl’s Law

- Amdahl’s law is used to find the performance gain that can be obtained by improving some portion or a functional unit of a computer.
- It states that “ The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used”.

- This law defines the ***speedup.***
- Speedup tells us how much faster a task will run using the computer.
- It depends on 2 factors:
- ***Fraction***$_{\textbf{\textit{enhanced}}}$***:*** The fraction of the execution time in the original machine that can be converted to take advantage of the enhancement. It is always less than or equal to 1. Example: If 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60.
- ***Speedup***$_{\textbf{\textit{enhanced:}}}$ The improvement gained by the enhanced execution mode. It is always greater than or equal to 1. Example: If the enhanced mode takes, say 2 seconds for a portion of the program, while it is 5 seconds in the original mode, then the improvement is 5/2.

Imagine a situation where three friends, Sam, Jack, and Harry, are going to a party. There are two conditions to be able to get into the party hall; first, they need to go separately, and second, all of them need to be present at the door to be able to get in. Now, assume that Sam is coming in his car, Jack is using a motorcycle, and Harry is coming by foot. In this case, Harry is considered a bottleneck in the performance of the whole system. In other words, no matter how fast Sam and Jack can reach the party hall, all need to wait for Harry to arrive to be able to attend the party. This means that in order to accelerate the overall process, you need to concentrate on improving the performance of Harry much more than the other two.



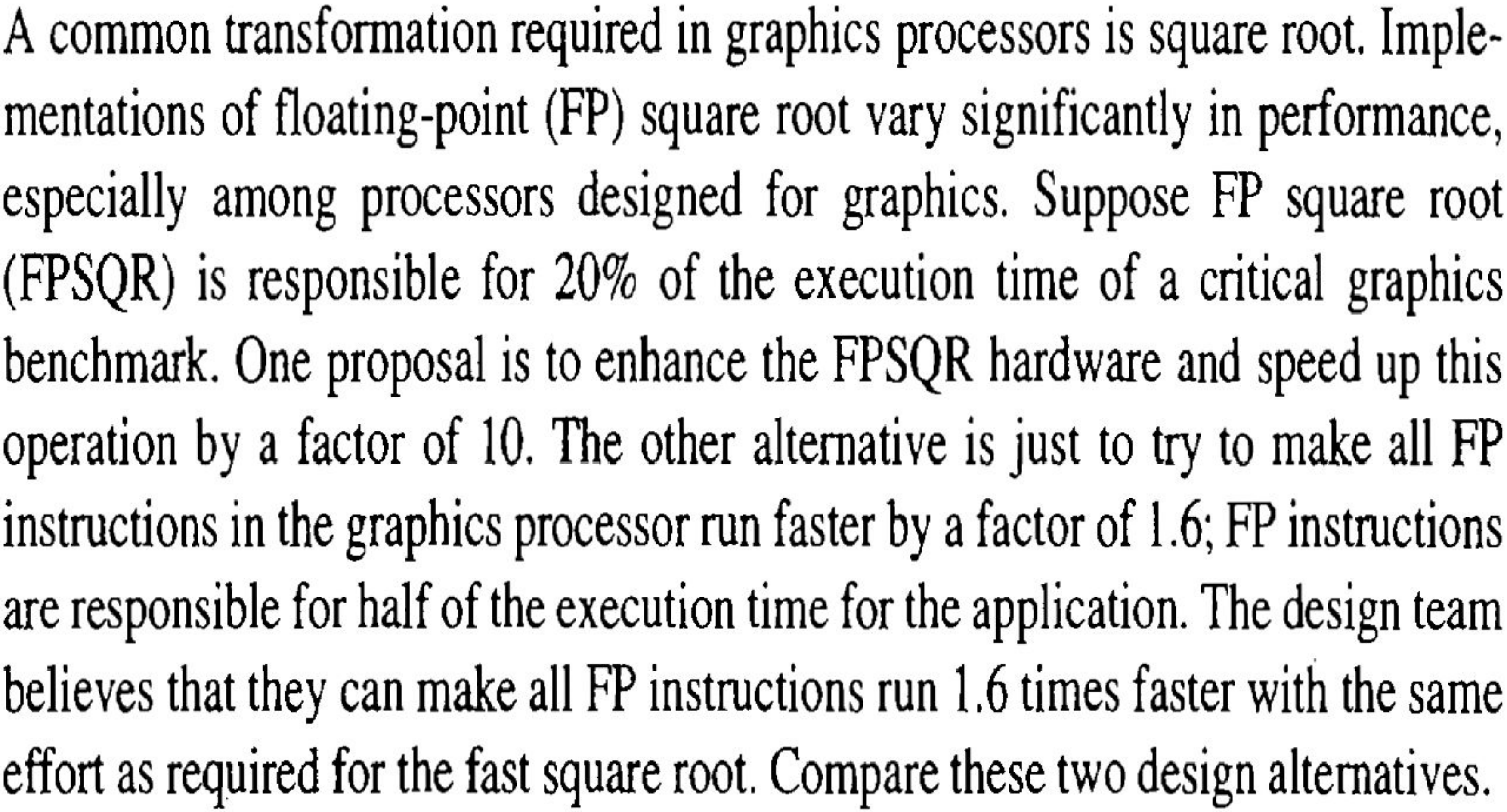
$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Example Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer $\text{Fraction}_{\text{enhanced}} = 0.4$; $\text{Speedup}_{\text{enhanced}} = 10$; $\text{Speedup}_{\text{overall}} = \dfrac{1}{0.6 + \dfrac{0.4}{10}} = \dfrac{1}{0.64} \approx 1.56$



A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Answer We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

problems

- What is the overall speedup if you make 10% of a program 90 times faster?

- Amdahl's law

$$OverallSpeedup = \frac{1}{(1-f)+\frac{f}{s}}$$

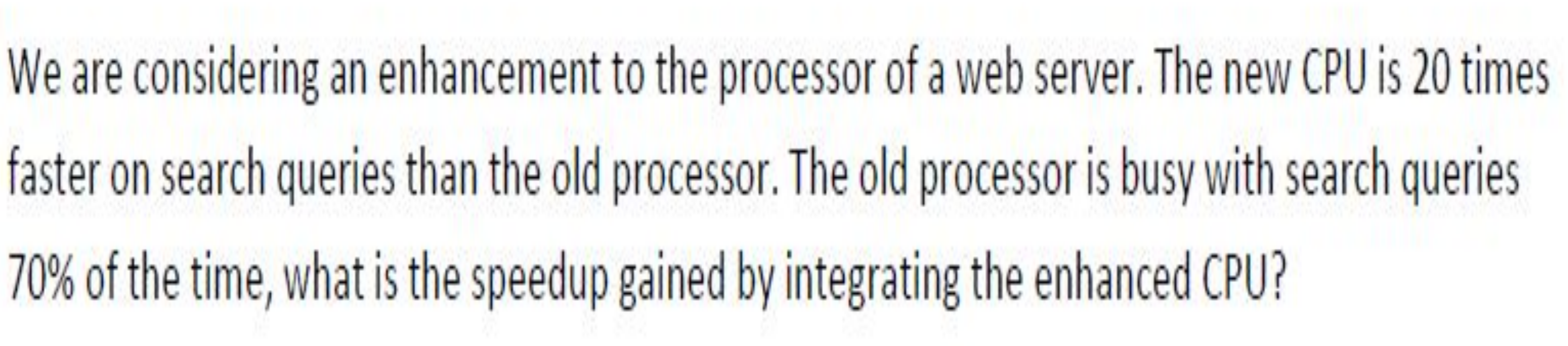
- What is the overall speedup if you make 10% of a program 90 times faster?

$$\frac{1}{(1-0.1)+\frac{0.1}{90}} \approx \frac{1}{0.9011} \approx 1.11$$

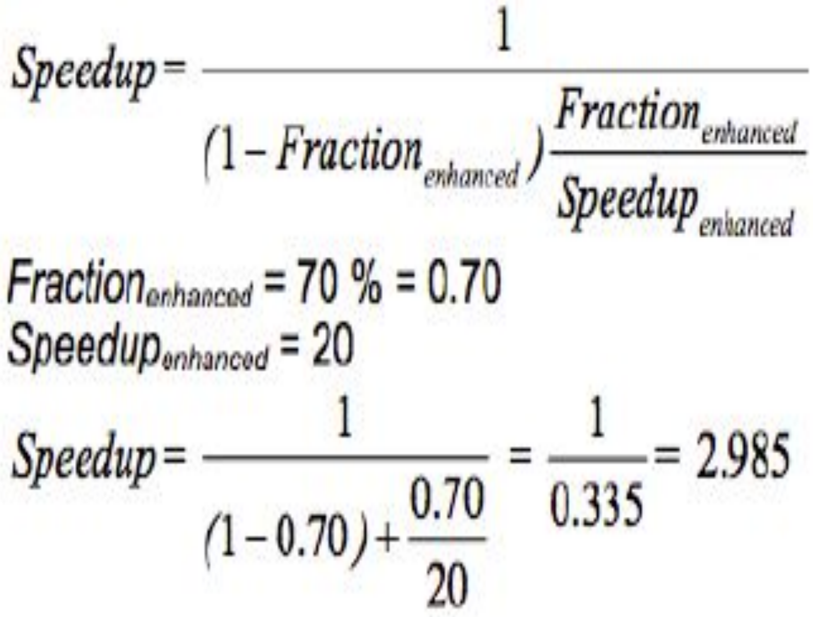
- What is the overall speedup if you make 90% of a program 10 times faster?

- What is the overall speedup if you make 90% of a program 10 times faster

$$\frac{1}{(1-0.9)+\frac{0.9}{10}} = \frac{1}{0.19} \approx 5.26$$



We are considering an enhancement to the processor of a web server. The new CPU is 20 times faster on search queries than the old processor. The old processor is busy with search queries 70% of the time, what is the speedup gained by integrating the enhanced CPU?



$$Speedup = \frac{1}{\left(1 - Fraction_{enhanced}\right) \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$Fraction_{enhanced} = 70\ \% = 0.70$

$Speedup_{enhanced} = 20$

$$Speedup = \frac{1}{\left(1 - 0.70\right) + \frac{0.70}{20}} = \frac{1}{0.335} = 2.985$$

- We are considering an enhancement to the processor of a server. The new CPU 10X faster. I/O bound server, so 60% time waiting for I/O.

- New CPU 10X faster
- I/O bound server, so 60% time waiting for I/O

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$= \frac{1}{\left(1 - 0.4\right) + \dfrac{0.4}{10}} = \frac{1}{0.64} = 1.56$$