Introduction to Parallel Computing

Ananth Grama, Anshul Gupta,
George Karypis, and Vipin Kumar

To accompany the text ``Introduction to Parallel Computing'',
Addison Wesley

Course Objectives & Outcomes

To understand different parallel programming models

CO1:

Understand various Parallel Paradigm

1: Introduction to Parallel Computing

Syllabus:

Introduction to Parallel Computing: Motivating Parallelism

Modern Processor: Stored-program computer architecture, General-purpose Cache-based Microprocessor architecture **Parallel Programming Platforms:** Implicit Parallelism, Dichotomy of Parallel Computing Platforms, Physical Organization of Parallel Platforms, Communication Costs in Parallel Machines. Levels of parallelism,

Models: SIMD, MIMD, SIMT, SPMD, Data Flow Models, Demand-driven Computation**,**

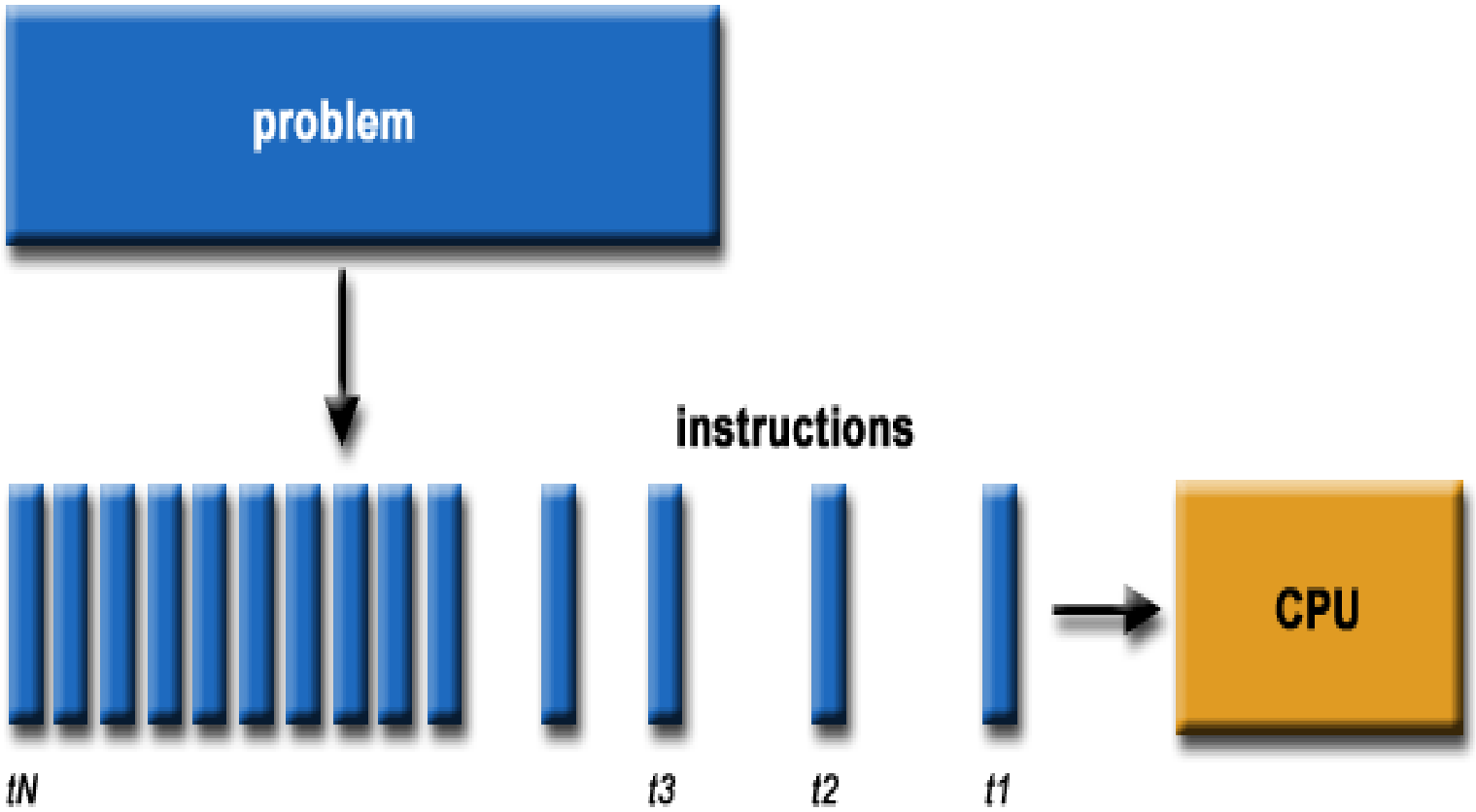
Architectures: N-wide superscalar architectures, multi-core, multi-threaded**.**

Introduction to Parallel Computing

Traditionally, software has been written for ***serial*** computation:

- To be run on a single computer having a single Central Processing Unit (CPU);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

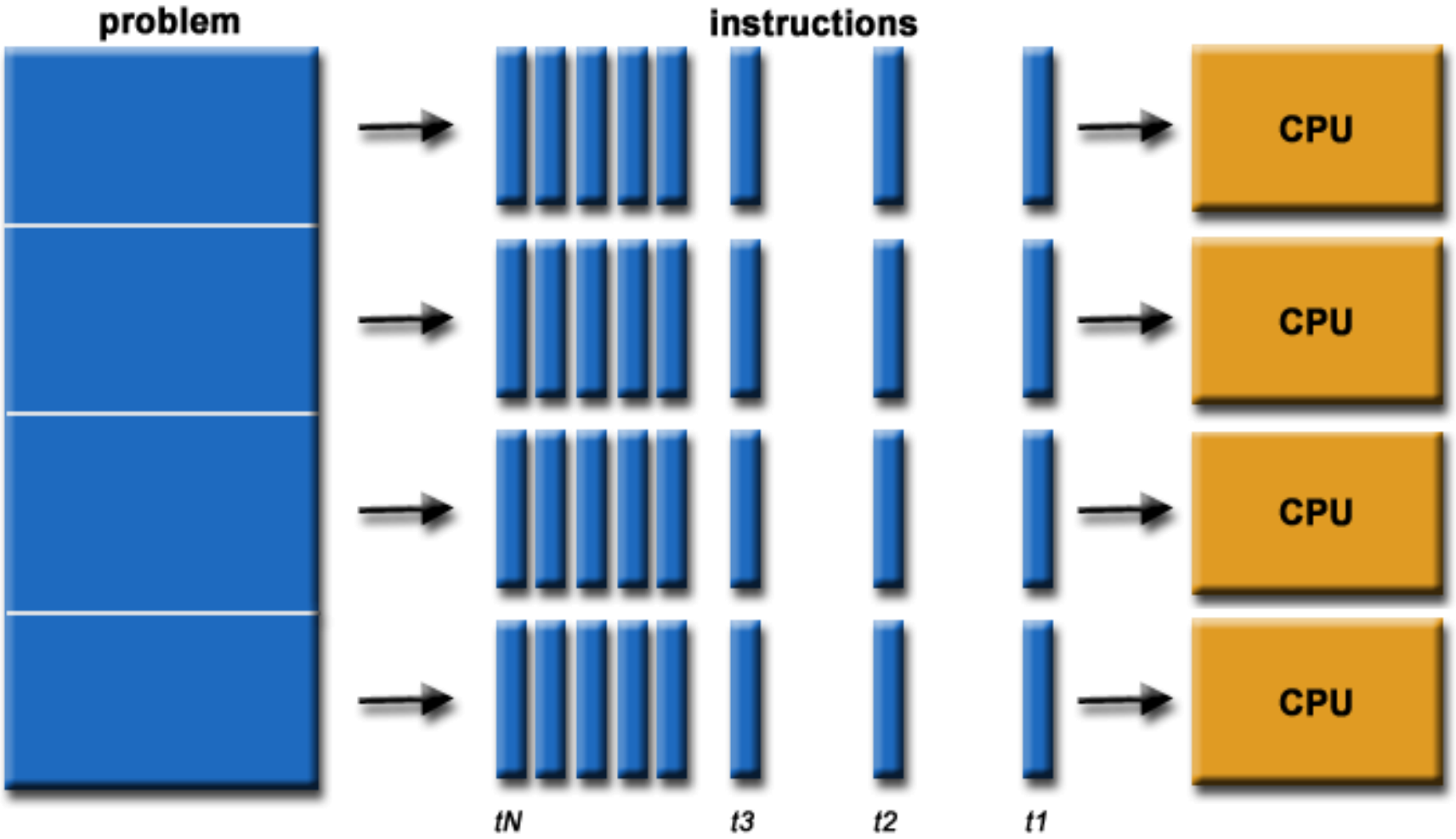
Serial computation:



Parallel Computing

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.
- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

CONT..



Motivating Parallelism

- Development of parallel software has traditionally been thought of as time and effort intensive.
- This can be largely attributed to the inherent complexity of specifying and coordinating concurrent tasks, a lack of portable algorithms, standardized environments, and software development toolkits.

1. **The Computational Power Argument – from Transistors to FLOPS**
2. **The Memory/Disk Speed Argument**
3. **The Data Communication Argument**

The Computational Power Argument – from Transistors to FLOPS …

- **In 1965, Gordon Moore made the following simple observation**:

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.

Certainly over the short term this rate can be expected to continue, if not to increase.

Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.

That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."

The Memory/Disk Speed Argument

The overall speed of computation is determined not just by the speed of the processor, but also by the ability of the memory system to feed data to it. While clock rates of high-end processors have increased at roughly 40% per year over the past decade, DRAM access times have only improved at the rate of roughly 10% per year over this interval.

- The overall performance of the memory system is determined by the fraction of the total memory requests that can be satisfied from the cache

The Data Communication Argument

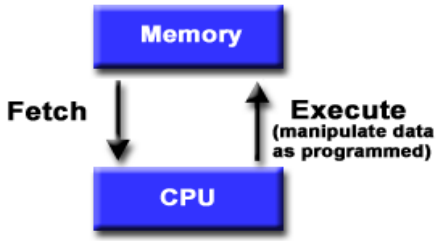
- In many applications there are constraints on the location of data and/or resources across the Internet.
- An example of such an application is mining of large commercial datasets distributed over a relatively low bandwidth network.
- In such applications, even if the computing power is available to accomplish the required task without resorting to parallel computing, it is infeasible to collect the data at a central location.
- In these cases, the motivation for parallelism comes not just from the need for computing resources but also from the infeasibility or undesirability of alternate (centralized) approaches.

Reference Book: Ananth

Modern Processor:

1. Stored-program computer architecture : Its defining property, which set it apart from earlier designs, is that its instructions are numbers that are stored as data in memory. Instructions are read and executed by a control unit; a separate arithmetic/logic unit is responsible for the actual computations and manipulates data stored in memory along with the instructions

A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.



Cont..

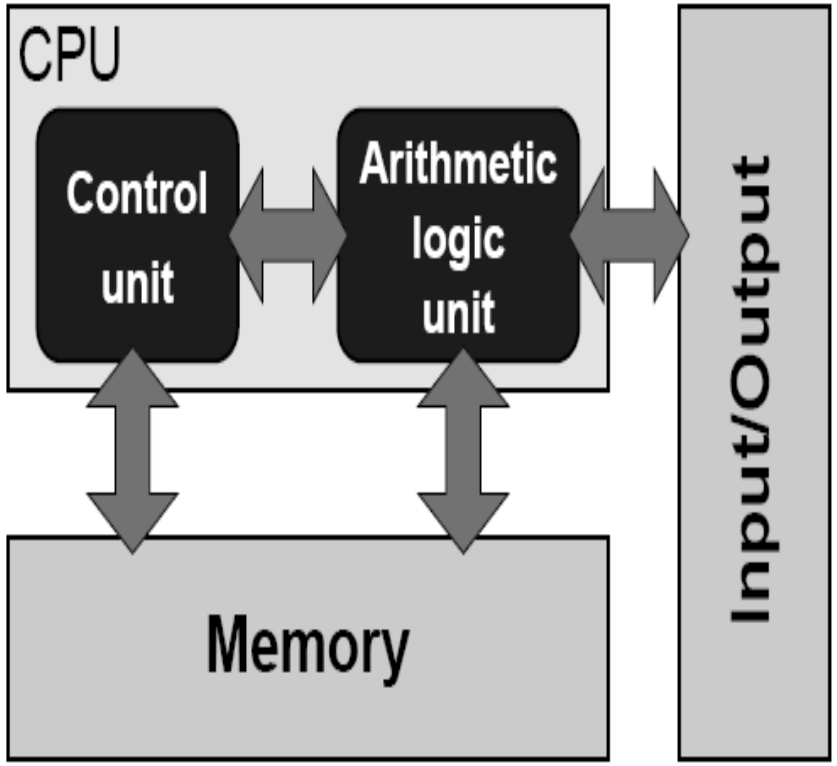


Figure 1.1: Stored-program computer architectural concept. The "program," which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.

Cont..

Instructions and data must be continuously fed to the control and arithmetic units, so that the speed of the memory interface poses a limitation on compute performance.

The architecture is inherently sequential, processing a single instruction with (possibly) a single operand or a group of perands from memory.(SISD)

Cont..

2. General-purpose Cache-based Microprocessor architecture :

- Microprocessors implement stored pgm....
- Modern processors have lot of componets but only a small part does the actual work -AU for fp and int operations.
- Rest are CPU regs, nowdays processors req all operands to reside in regs.
- LD(load) and ST(store) units handle instruction transfer.
- Queues for instructions
- Finally Cache

Cont…

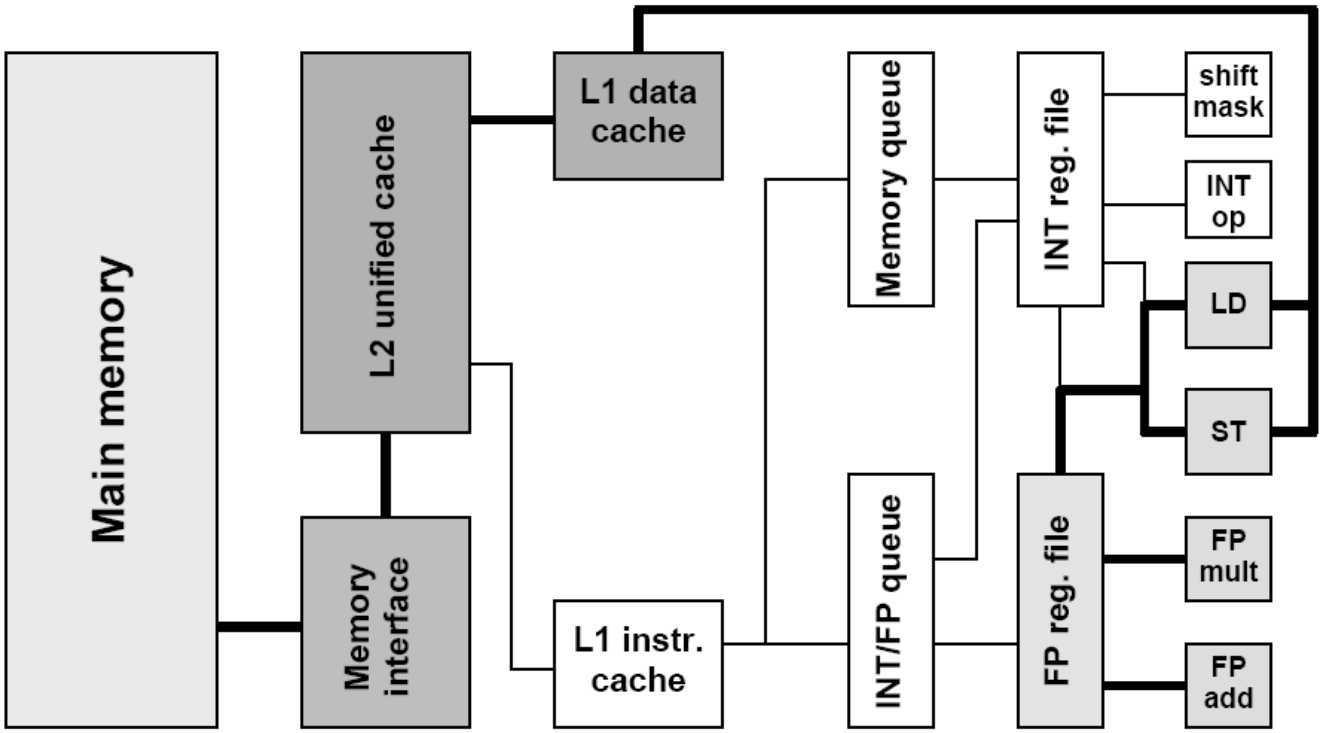


Figure 1.2: Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.

References

Book Title: Introduction to High Performance Computing for Scientist and Engineers

Authors: George and Wellien

- Reference: http://prdrklaina.weebly.com/uploads/5/7/7/3/5773421/introduction_to_high_performance_computing_for_scientists_and_engineers.pdf

Scope of Parallelism

- Conventional architectures coarsely comprise of a processor, memory system, and the datapath.
- Each of these components present significant performance bottlenecks.
- Parallelism addresses each of these components in significant ways.
- Different applications utilize different aspects of parallelism - e.g., data itensive applications utilize high aggregate throughput, server applications utilize high aggregate network bandwidth, and scientific applications typically utilize high processing and memory system performance.
- It is important to understand each of these performance bottlenecks.

Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
- Higher levels of device integration have made available a large number of transistors.
- The question of how best to utilize these resources is an important one.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.

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

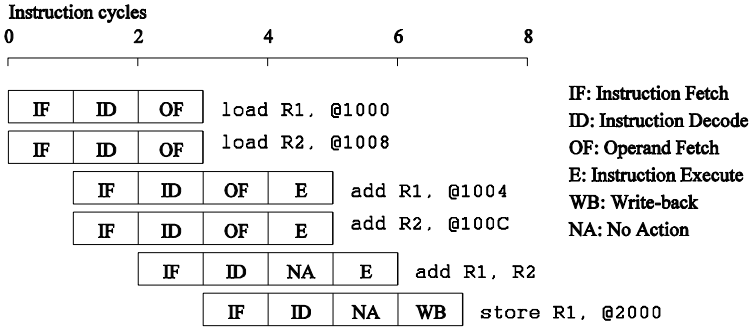
- One simple way of alleviating these bottlenecks is to use multiple pipelines.
- The question then becomes one of selecting these instructions.
- In Below example, there is some wastage of resources due to data dependencies.

Superscalar Execution: An Example

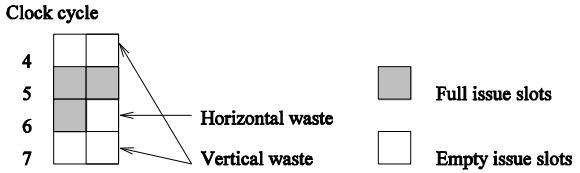
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000 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

(i) (ii) (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.
- Conventional microprocessors typically support four-way superscalar execution.

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.

VLIW & Superscalar

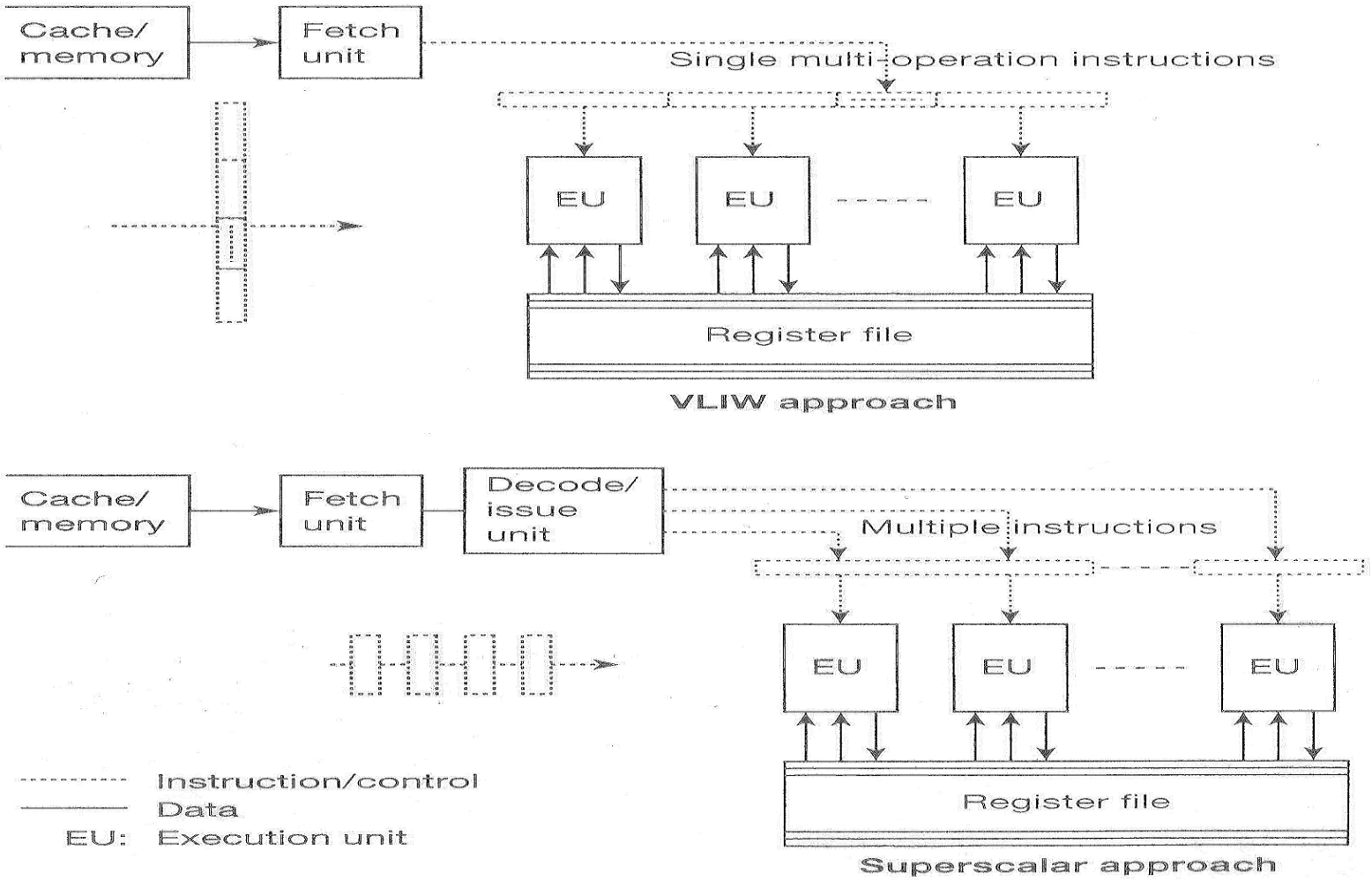


Figure 6.2 Main differences between superscalar processors and VLIW architectures.

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.

Dichotomy of Parallel Computing Platforms

- An explicitly parallel program must specify concurrency and interaction between concurrent subtasks.
- The former is sometimes also referred to as the control structure and the latter as the communication model.

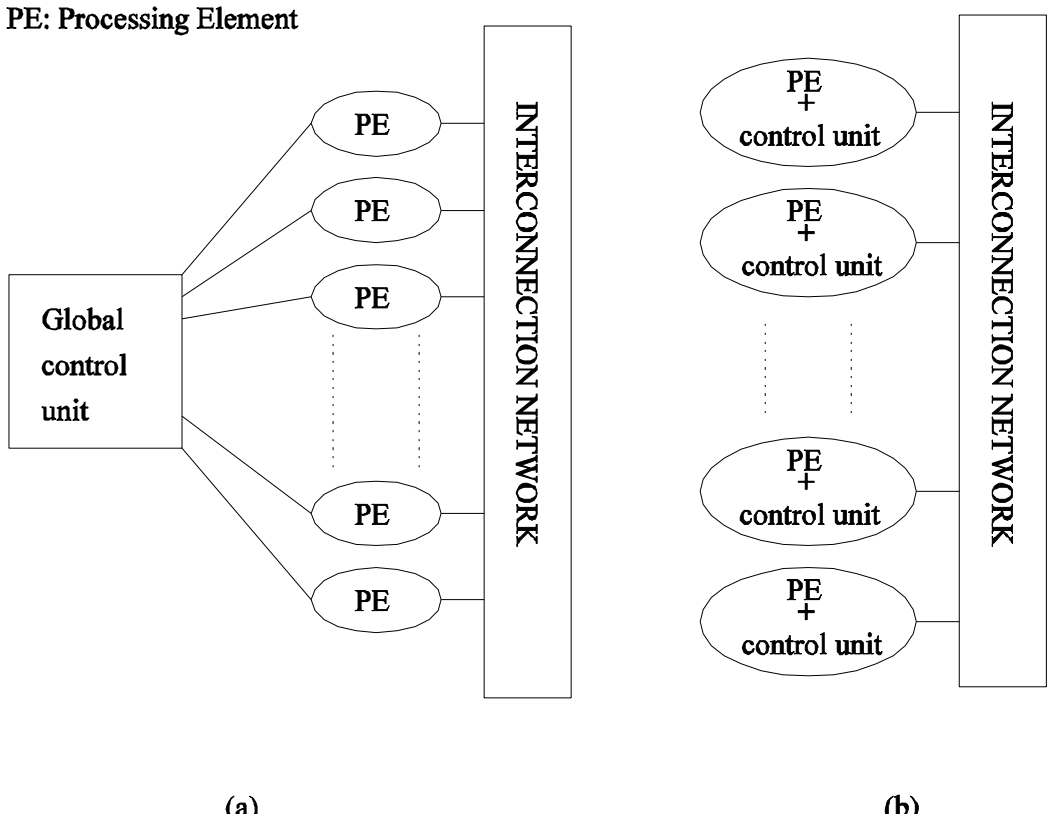
Control Structure of Parallel Programs

- Parallelism can be expressed at various levels of granularity - from instruction level to processes.
- Between these extremes exist a range of models, along with corresponding architectural support.

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

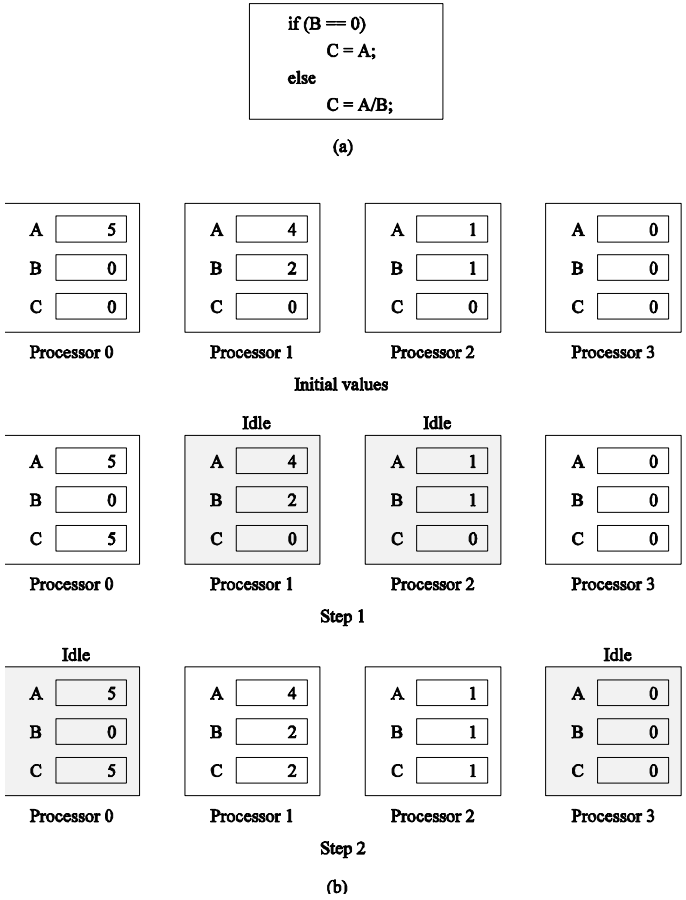


A typical SIMD architecture (a) and a typical MIMD architecture (b).

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



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 ae 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.

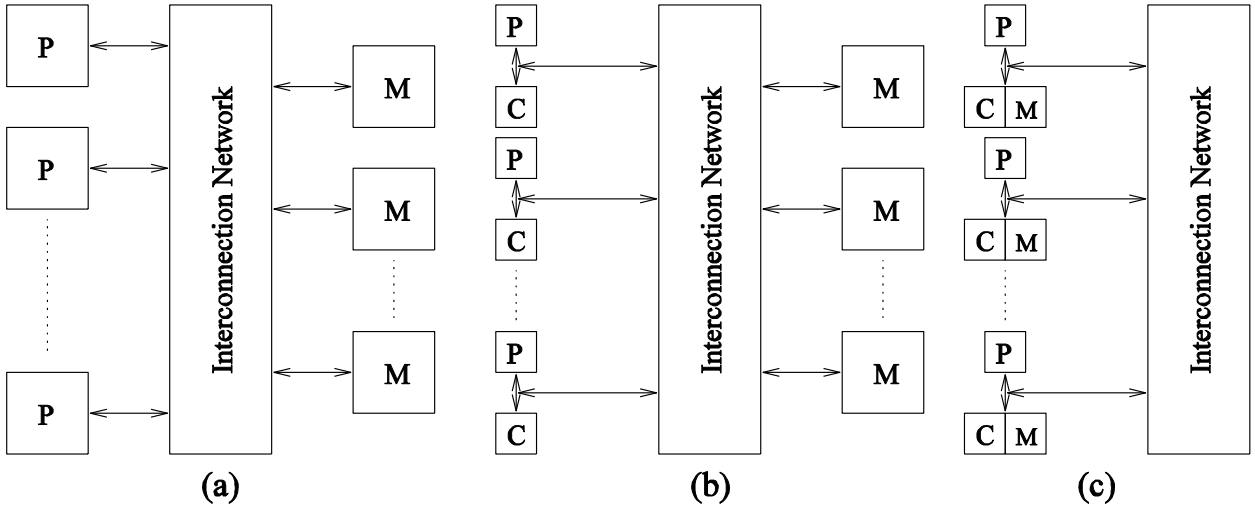
Communication Model of Parallel Platforms

- 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).

Physical Organization of Parallel Platforms

We begin this discussion with an ideal parallel machine called Parallel Random Access Machine, or PRAM.

Architecture of an Ideal Parallel Computer

- A natural extension of the Random Access Machine (RAM) serial architecture is the Parallel Random Access Machine, or PRAM.
- PRAMs consist of *p* processors and a global memory of unbounded size that is uniformly accessible to all processors.
- Processors share a common clock but may execute different instructions in each cycle.

Architecture of an Ideal Parallel Computer

- Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.
  - Exclusive-read, exclusive-write (EREW) PRAM.
  - Concurrent-read, exclusive-write (CREW) PRAM.
  - Exclusive-read, concurrent-write (ERCW) PRAM.
  - Concurrent-read, concurrent-write (CRCW) PRAM.

Cont..

2. Interconnection Networks for Parallel Computers

Classification of interconnection networks:

(a) a static network; and (b) a dynamic network.

3. Network Topologies: Bus-Based Networks, Crossbar Networks, Multistage Networks, Star-Connected Network, Linear Arrays, Meshes, and k-d Meshes etc
4. Evaluating Static Interconnection Networks : Diameter, connection and Bisection Width
5. Evaluating Dynamic Interconnection Networks
6. Cache Coherence in Multiprocessor Systems: Snoopy cache based and Directory based

Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as static or dynamic.
- Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

Evaluating Static Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound *k*-ary *d*-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Evaluating Dynamic Interconnection Networks

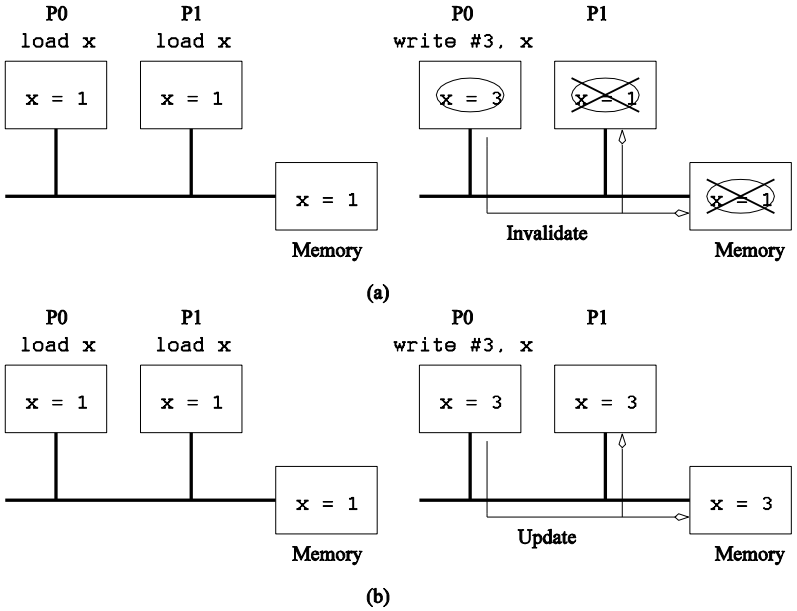
Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2 \log p$	1	2	$p - 1$

Cache Coherence in Multiprocessor Systems

- Interconnects provide basic mechanisms for data transfer.
- In the case of shared address space machines, additional hardware is required to coordinate access to data that might have multiple copies in the network.
- The underlying technique must provide some guarantees on the semantics.
- This guarantee is generally one of serializability, i.e., there exists some serial order of instruction execution that corresponds to the parallel schedule.

Cache Coherence in Multiprocessor Systems

When the value of a variable is changes, all its copies must either be invalidated or updated.



Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

Communication Costs in Parallel Machines

- Along with idling and contention, communication is a major overhead in parallel programs.
- The cost of communication is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols.

Message Passing Costs in Parallel Computers

- The total time to transfer a message over a network comprises of the following:
  - *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  - *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
  - *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

Store-and-Forward Routing

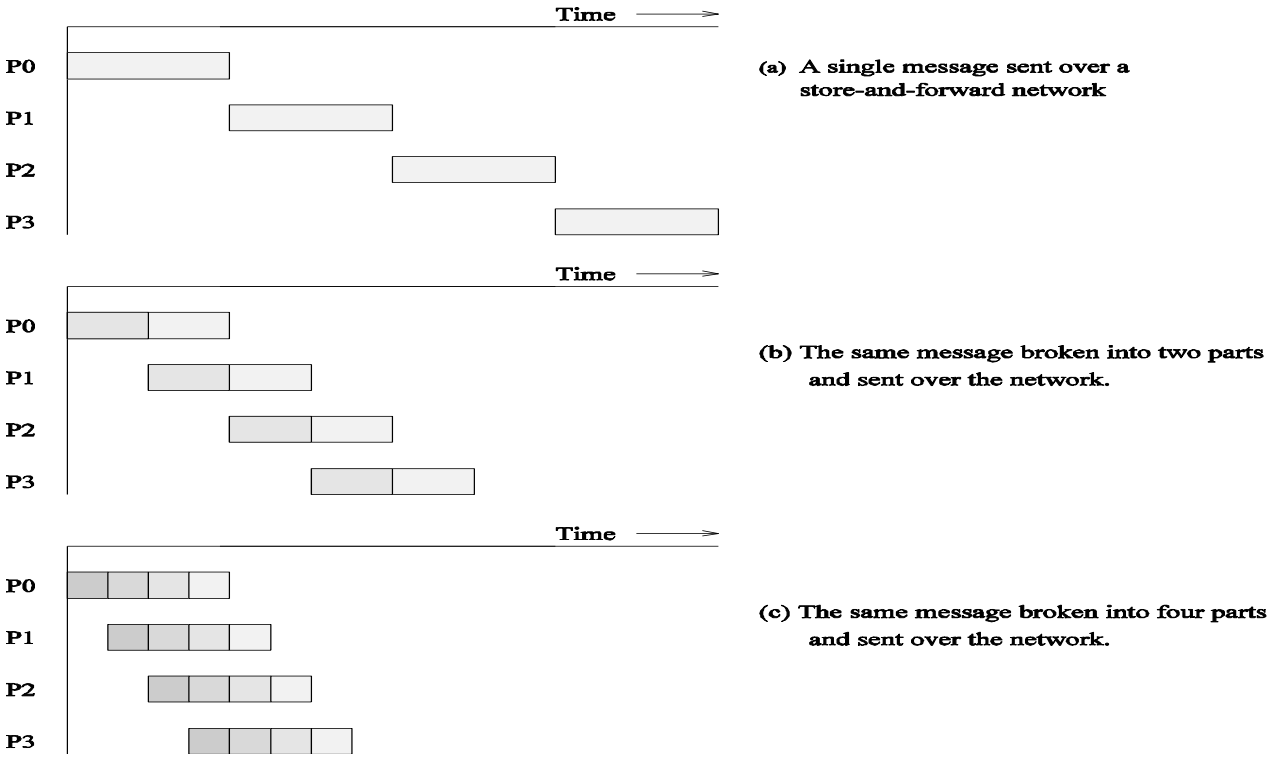
- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size *m* words to traverse *l* communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, t_h is small and the above expression can be approximated by

$$t_{comm} = t_s + mlt_w.$$

Routing Techniques



Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

B]Communication Costs in Shared-Address-Space Machines

- While the basic messaging cost applies to these machines as well, a number of other factors make accurate cost modeling more difficult.
- Memory layout is typically determined by the system.
- Finite cache sizes can result in cache thrashing.
- Overheads associated with invalidate and update operations are difficult to quantify.
- Spatial locality is difficult to model.
- Prefetching can play a role in reducing the overhead associated with data access.
- False sharing and contention are difficult to model.

Levels of parallelism

1. Data Parallelism: Many problems in scientific computing involve processing of large quantities of data stored on a computer. If this manipulation can be performed in parallel, i.e., by multiple processors working on different parts of the data, we speak of data parallelism. As a matter of fact, this is the dominant parallelization concept in scientific computing on MIMD-type computers. It also goes under the name of SPMD (Single Program Multiple Data), as usually the same code is executed on all processors, with independent instruction pointers.

Ex: Medium-grained loop parallelism, Coarse-grained parallelism by domain decomposition

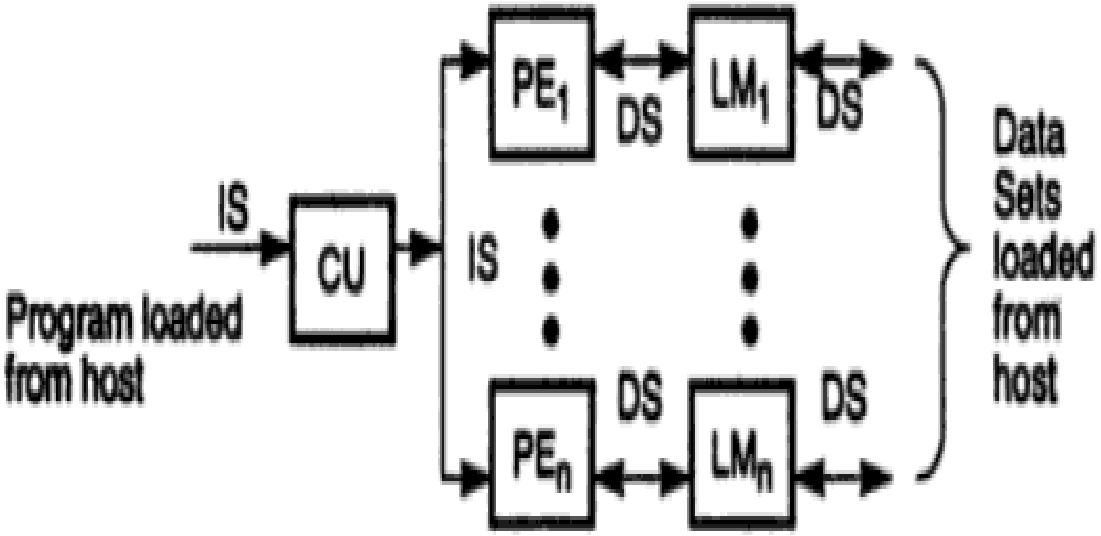
Levels of Parallelism

2. Functional Parallelism: Sometimes the solution of a “big” numerical problem can be split into more or less disparate subtasks, which work together by data exchange and synchronization. In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called MPMD (Multiple Program Multiple Data).

Ex: Master Worker Scheme, Functional decomposition

Ref:http://prdrklaina.weebly.com/uploads/5/7/7/3/5773421/introduction_to_high_performance_computing_for_scientists_and_engineers.pdf

Models : SIMD, MIMD, SIMT, SPMD



(b) SIMD architecture (with distributed memory)

Captions:

CU = Control Unit

PU = Processing Unit

MU = Memory Unit

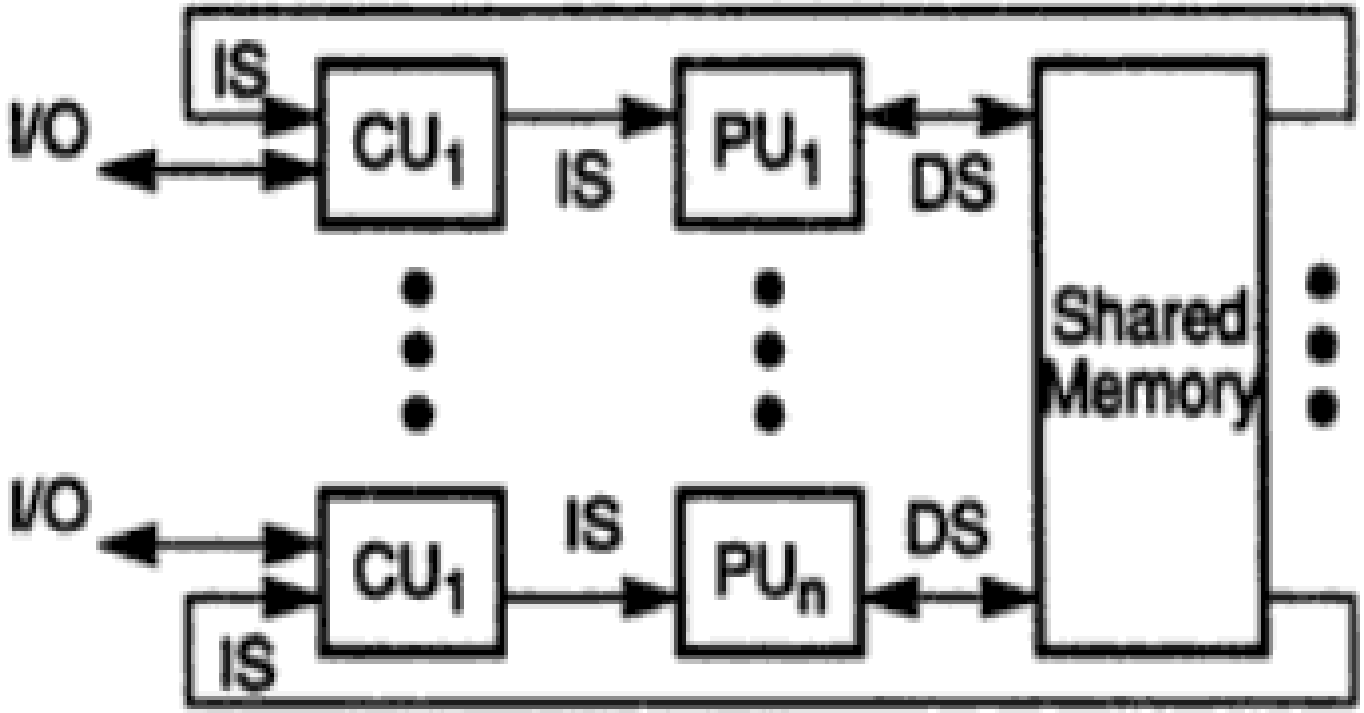
IS = Instruction Stream

DS = Data Stream

PE = Processing Element

LM = Local Memory

MIMD



SIMT

Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading. It is different from SPMD in that all instructions in all "threads" are executed in lock-step. The SIMT execution model has been implemented on several GPUs and is relevant for general-purpose computing on graphics processing units (GPGPU),

e.g. some supercomputers combine CPUs with GPUs.

SPMD

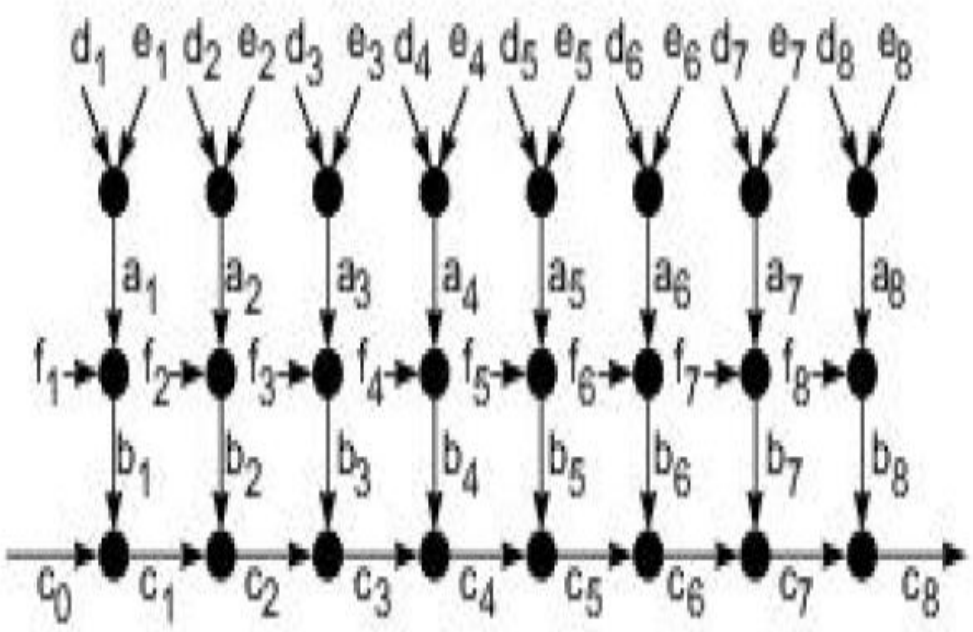
- In contrast to SIMD processors, MIMD processors can execute different programs on different processors.
- A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors.
- It is easy to see that SPMD and MIMD are closely related in terms of programming 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.

Data Flow

- In a daraflow computer, the execution of an instruction is driven by data availability instead of being guided by a program counter, In theory, any instruction should be ready for execution whenever operands become available.
- The instructions in a data-driven program are not ordered in any way. instead of being stored separately in a main memory, data are directly held inside instructions.
- This data-driven scheme requires no program counter, and no eonlrol sequencer. However, it requires special mechanisms to detect data availability, to match data tokens with needy instructions, and to enable the chain reaction of asynchronous instruction executions. No memory sharing between instructions results in no side effects.

Cont..

```
input d, e, f
  c0 = 0
for i from 1 to 8 do
  begin
    ai := di + ei
    bi := ai * fi
    ci := bi + ci-1
  end
output a, b, c
```



(a) A sample program and its dataflow graph

Demand Driven

In a *reduction machine*, the computation is triggered by the demand for an operation's result. Consider the evaluation of a nested arithmetic expression $a = ((b + 1) \times c - (d \div e))$. The data-driven computation seen above chooses a bottom-up approach, starting from the innermost operations $b + 1$ and $d \div e$, then proceeding to the $\times$ operation, and finally to the outermost operation $-$. Such a computation has been called *eager evaluation* because operations are carried out immediately after all their operands become available.

A *demand-driven* computation chooses a top-down approach by first demanding the value of a, which triggers the demand for evaluating the next-level expressions $(b + 1) \times c$ and $d \div e$, which in turn triggers the demand for evaluating $b + 1$ at the innermost level. The results are then returned to the nested demander in the reverse order before a is evaluated.

A demand-driven computation corresponds to *lazy evaluation*, because operations are executed only when their results are required by another instruction. The demand driven approach matches naturally with the functional programming concept. The removal of side effects in functional programming makes programs easier to parallelize. There are two types of reduction machine models, both having a recursive control mechanism as characterized below.

N-Wide Superscalar

- Ideally: in an n-issue superscalar, n instructions are fetched, decoded, executed, and committed per cycle

In practice:

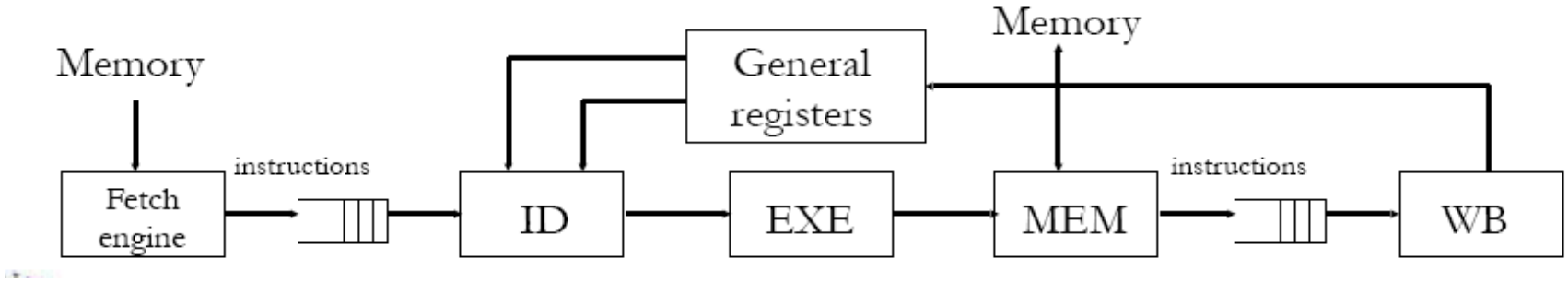
– Data, control, and structural hazards spoil issue flow

– Multi-cycle instructions spoil commit flow

- Buffers at issue (issue queue) and commit (reorder buffer)

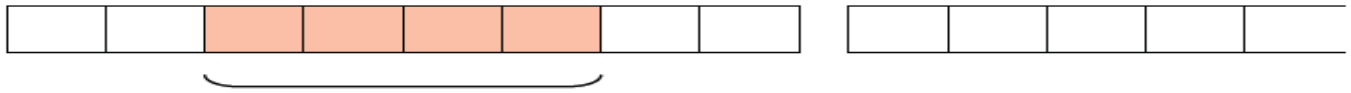
  decouple these stages from the rest of the pipeline and regularize somewhat breaks in the flow

Cont..

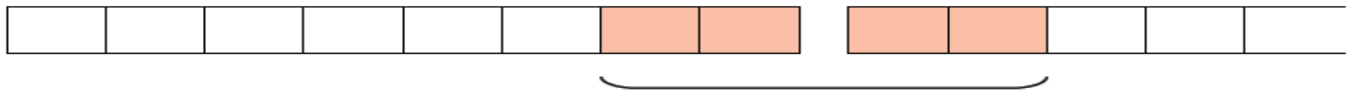


– e.g., 32 bit instructions and 32 byte instruction cache lines → 8 instructions per cache line; 4-wide superscalar processor

Case 1: all instructions located in same cache line and no branch



Case 2: instructions spread in more lines and no branch



– More than one cache lookup is required in the same cycle
– Words from different lines must be ordered and packed into instruction queue

Multicore Processor

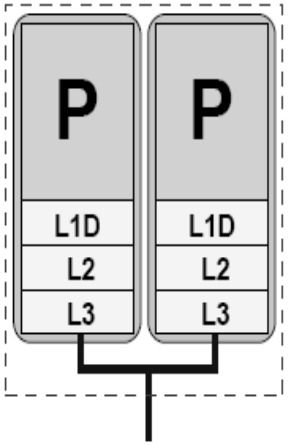


Figure 1.15: Dual-core processor chip with separate L1, L2, and L3 caches (Intel "Montecito"). Each core constitutes its own cache group on all levels.

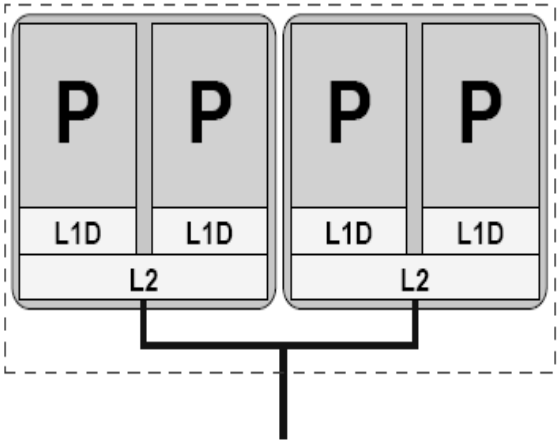


Figure 1.16: Quad-core processor chip, consisting of two dual-cores. Each dual-core has shared L2 and separate L1 caches (Intel "Harpertown"). There are two dual-core L2 groups.

Multithreaded Processor

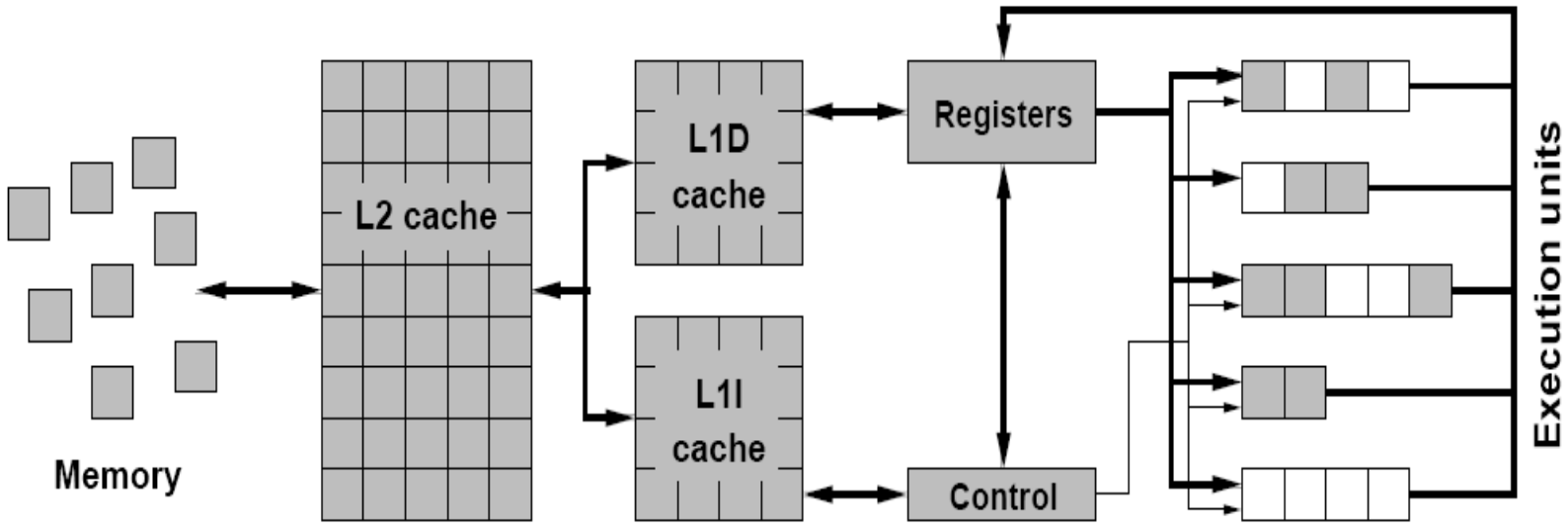


Figure 1.19: Simplified diagram of control/data flow in a (multi-)pipelined microprocessor without SMT. White boxes in the execution units denote pipeline bubbles (stall cycles). Graphics by courtesy of Intel.

Modern Processor and Architecture

Book Title: Introduction to High Performance Computing for Scientist and Engineers

Authors: George and Wellien

- Refernce:http://prdrklaina.weebly.com/uploads/5/7/7/3/5773421/introduction_to_high_performance_computing_for_scientists_and_engineers.pdf

Demand Driven and Data flow

- Book Name:
- ADVANCED COMPUTER ARCHITECTURE

Author name: Kai Hwang

THANK YOU