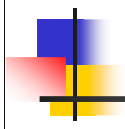
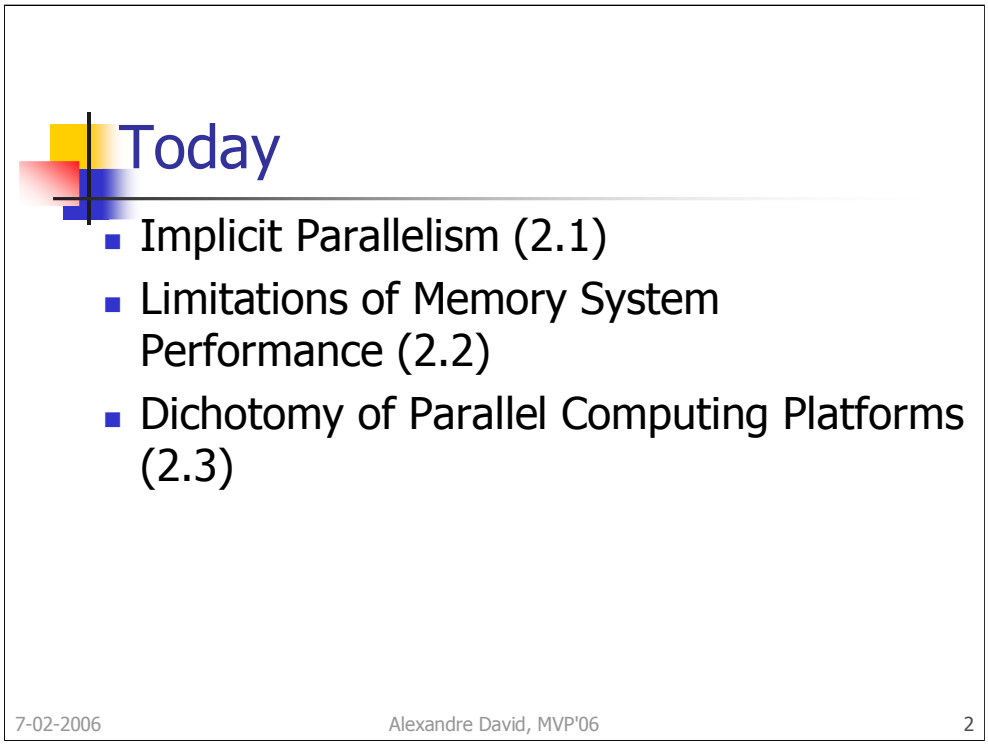


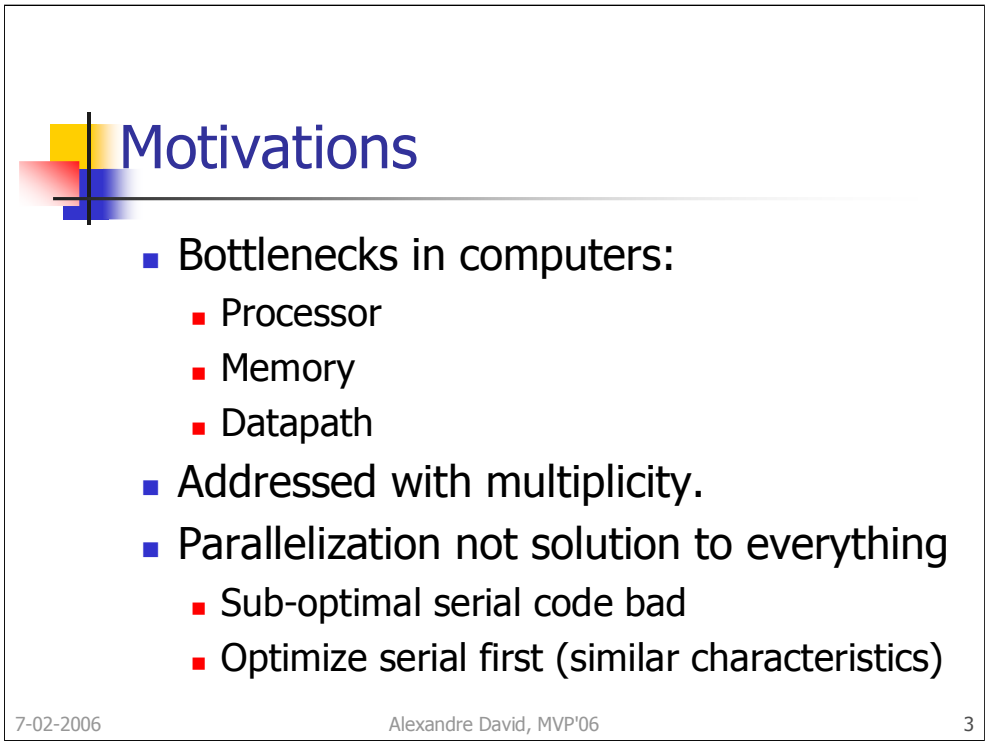
Parallel Programming Platforms

Alexandre David
B2-206

http://www.cs.aau.dk/~adavid/teaching/MTP-06/



I will follow the book, skipping sections from time to time but the order will be respected. References to sections are given on the web and reminded during the lectures (plan may change). Lectures are intended to teach you and help you read the book.



Bottlenecks of different kinds.

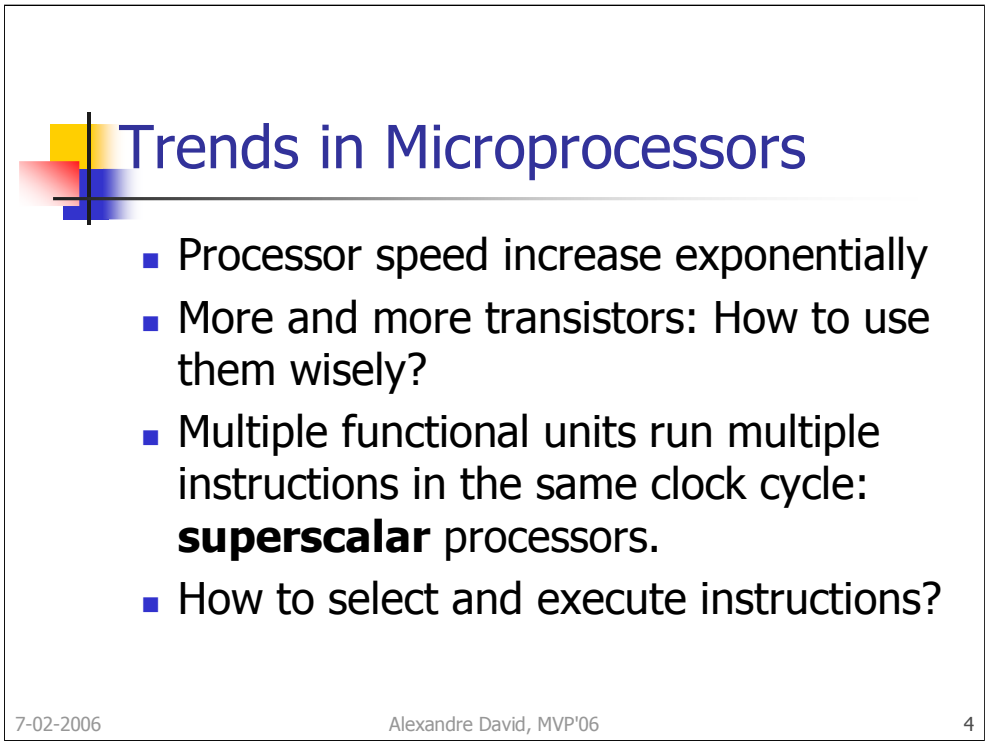
•Processor: less and less, though can be depending on some bad behaviors (branch miss prediction in large pipelines).

•Memory: more and more considering the speed gap between processor and memory, the problem being how to feed the processor with data so that it does not stay idle.

•Datapath: depends on programs and architecture, linked to previous ones.

Motivations for optimizing serial programs and why we talk about implicit parallelism: Sub-optimal serial code exhibits **unreliable** and **misleading** behaviors. Undesirable effects coming from poor cache utilization, bad branch prediction, etc … that may become even worse in a parallel context (distribute data, synchronize, etc …).

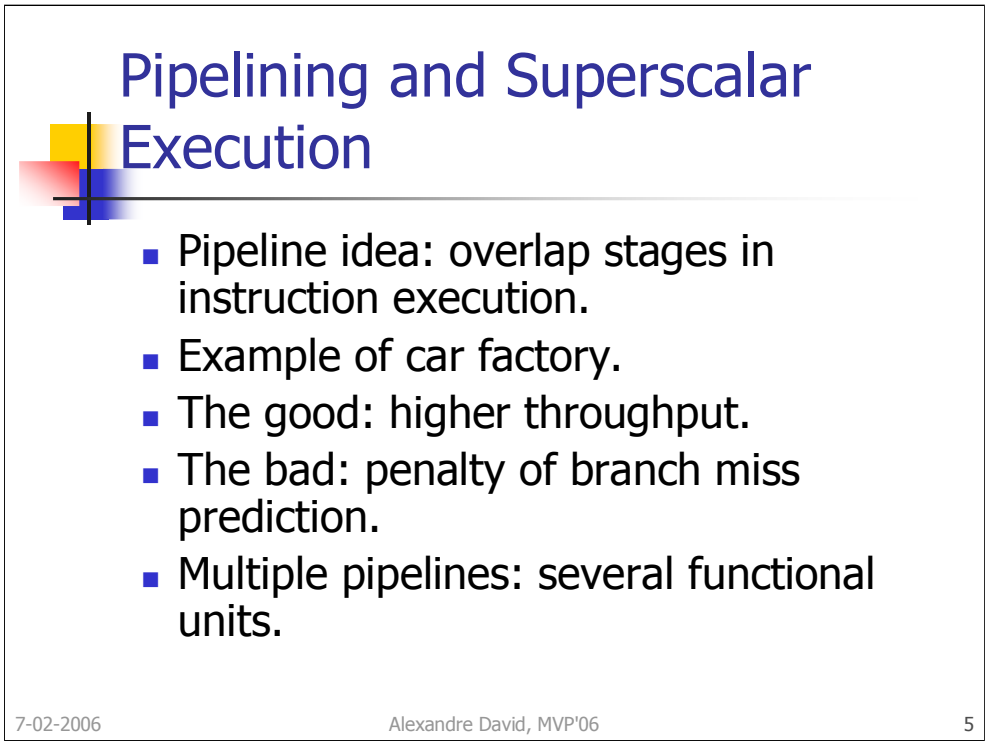
Similar characteristics in serial programs with intrinsic parallelism of modern processors (pipelines). Understanding architecture is the first step to good programming.



Already mentioned: Moore's law power x2 every 18 month, but for processor only. Global performance plagued by memory abysmal performance. Higher level of integration poses the question of how to best utilize transistors.

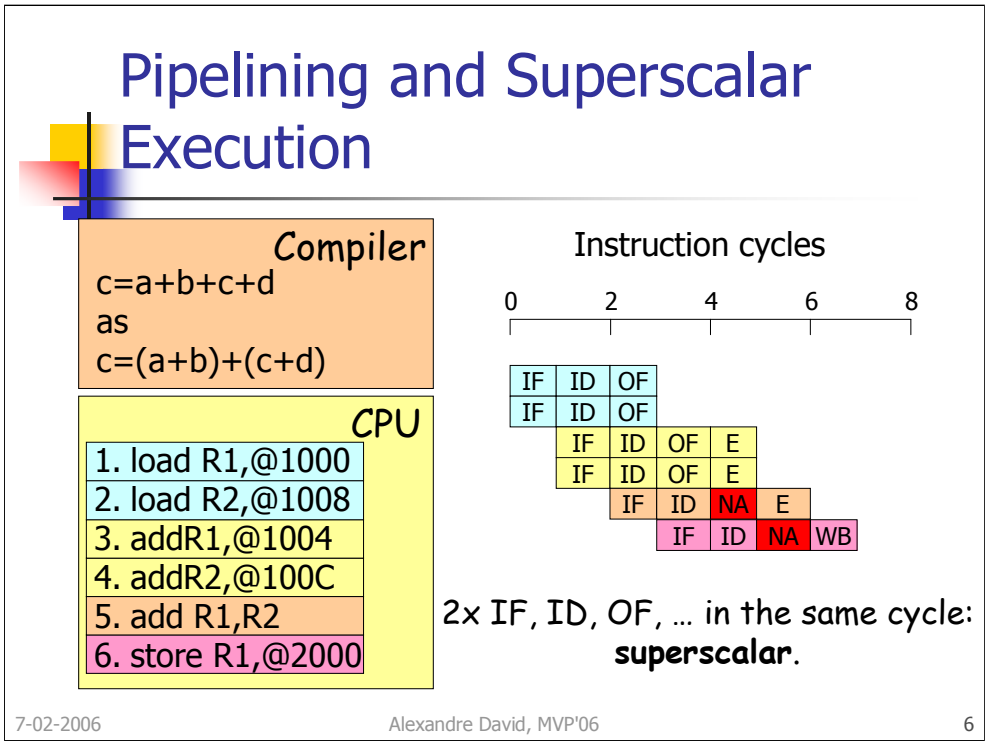
Functional units: MMU, FPU, etc usually part of the marketing buzz of microprocessor companies.

How to? That's the different architectures, not going into details but basically all processors are +/- RISC processors with a translation from assembler to microcode. Then most have branch prediction.



Overlapping stages: cut instructions in small pieces, one per cycle, and try to occupy all the stages. But why after all? Not better to have a super powerful one stage-do-all? Car factory: Imagine a fast factory where it takes 12h to complete one car. If there is one unit doing all, then it will be busy all the time and throughput would be 1 car per 12h. How to improve? Buy 11 other full scale units? Super expensive! Cut the bit unit in 12 smaller parts, 1h per part, every car needs 12h but throughput is 1 car per hour. 12x faster, cost efficient.

Branch prediction: try to keep the pipeline busy by filling it ahead, but if did it wrong, then need to flush it (and loose all the computations). P4: 20 stages, miss prediction means loose 20 cycles.



Dual issue or two-way superscalar execution.

IF: Instruction Fetch.

ID: Instruction Decode.

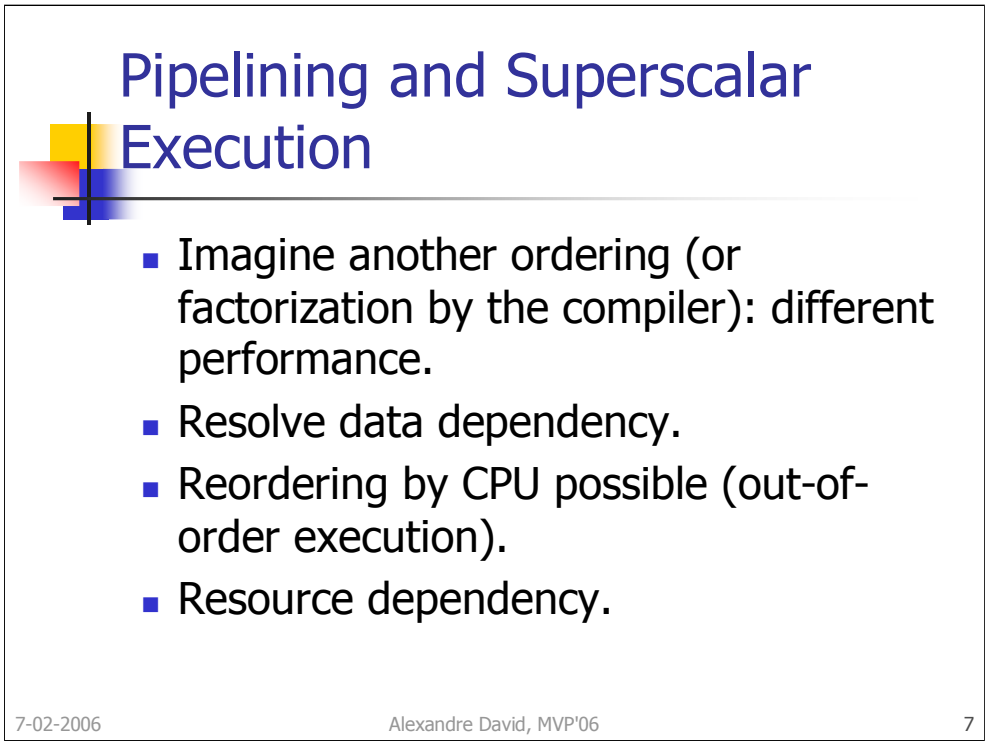
OF: Operand Fetch.

E: Instruction Execute.

WB: Write back.

NA: No Action.

Note: begin to execute 6th instruction at 4th clock cycle.



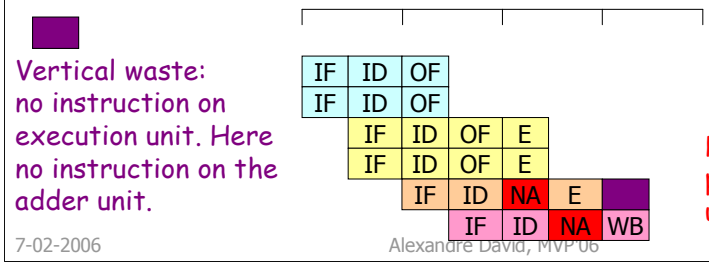
Data dependency: needs previous results in order to continue computations. A=(B+C)*D, we need B+C before computing *D.

Resource dependency: needs functional units. A=B*C+C*D+D*E+E*F+F*G, obviously not all * can be done in parallel because of lack of functional units.

Most processors are capable of out-of-order execution, not Xbox 360.

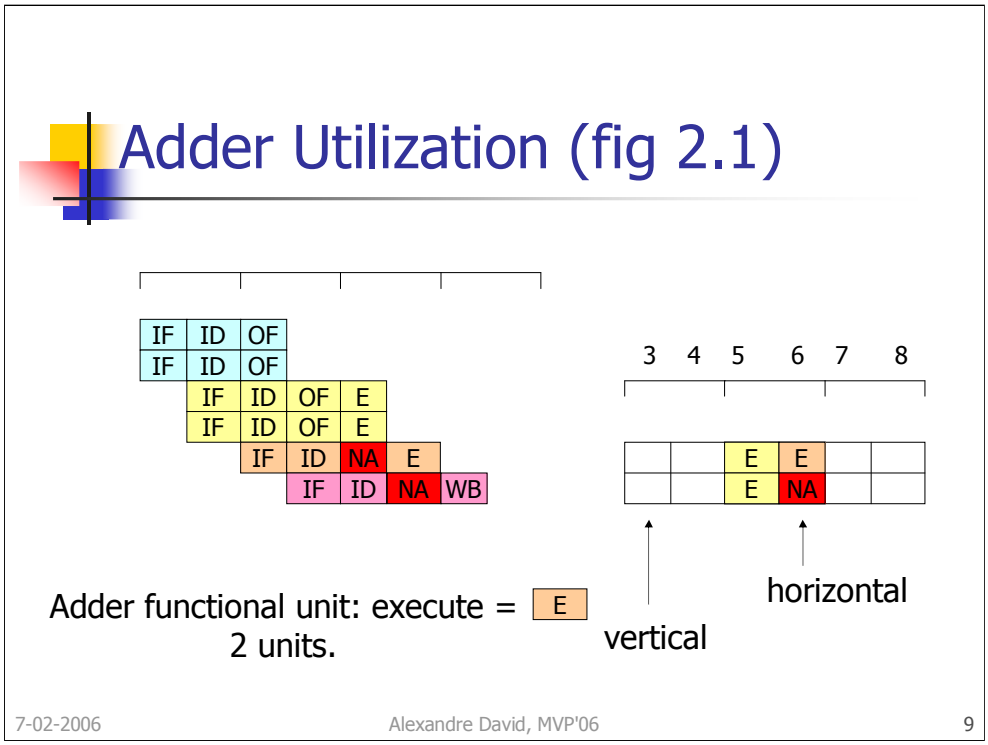
Limitations

- Bottleneck: slowest stage -> small stages to go fast -> long pipelines
  - BUT miss prediction gives big penalties
- How to keep busy the functional units?

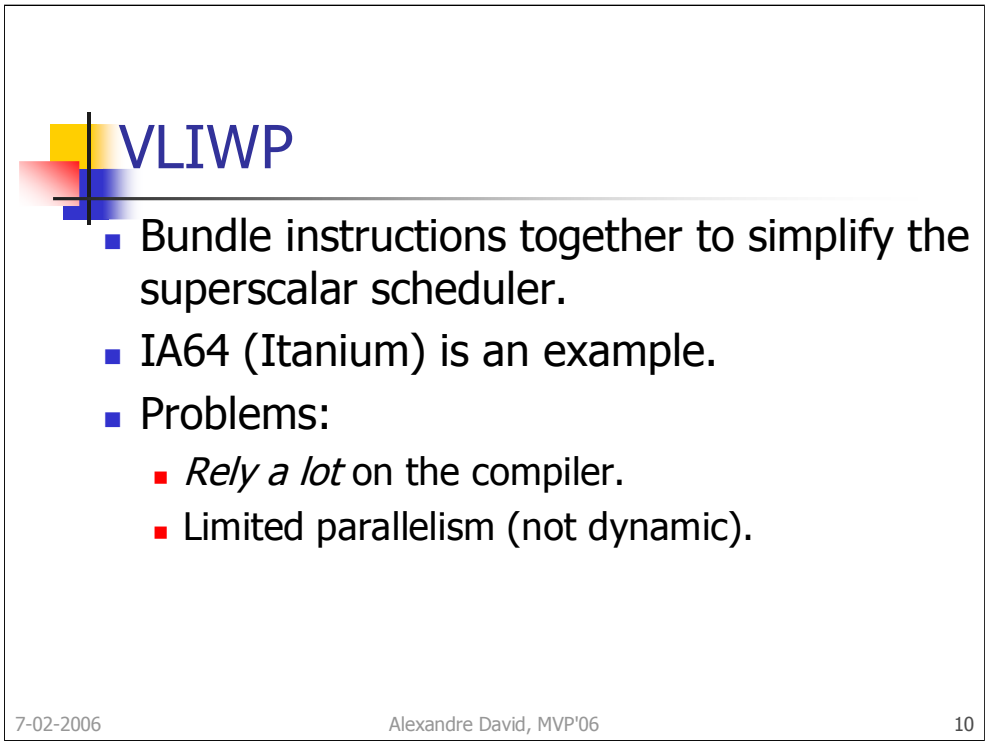


Intrinsic parallelism: Pipeline (multiple stages) & multiple functional units (superscalar) implement parallelism.

Modern processors: 4 way superscalar, 10-20 stage pipeline.

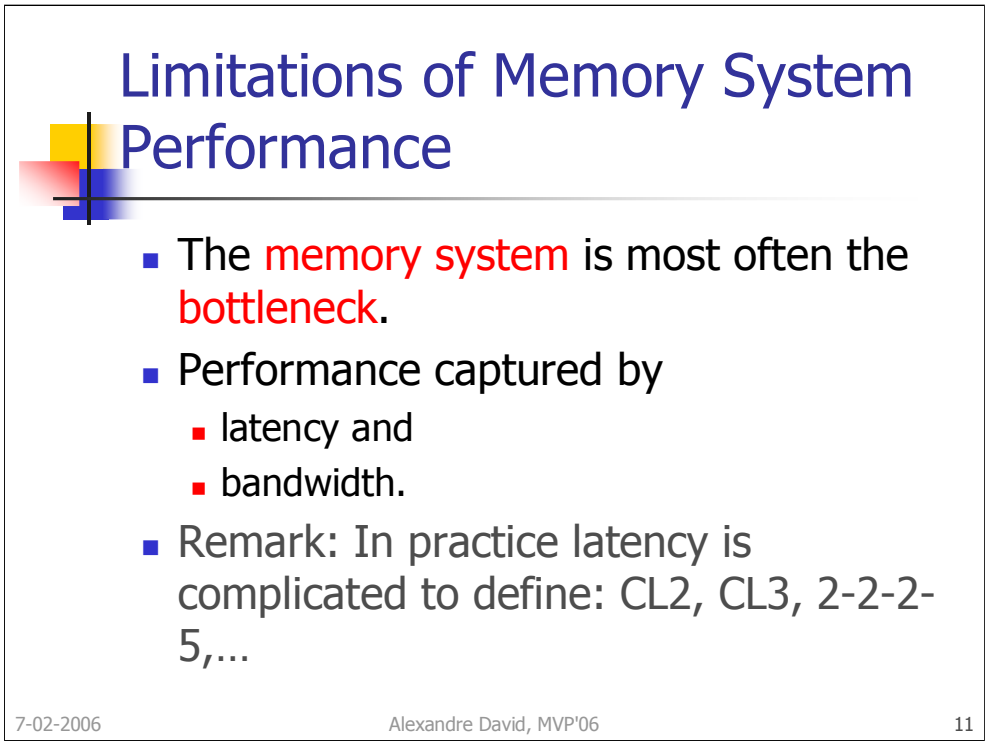


More explanation for fig 2.1.



Very Long Instruction Word Processors!

Superscalar schedulers are complex and expensive (transistors). VLIW design idea is to rely on the compiler to bundle instructions together, so that the scheduler becomes very simple.

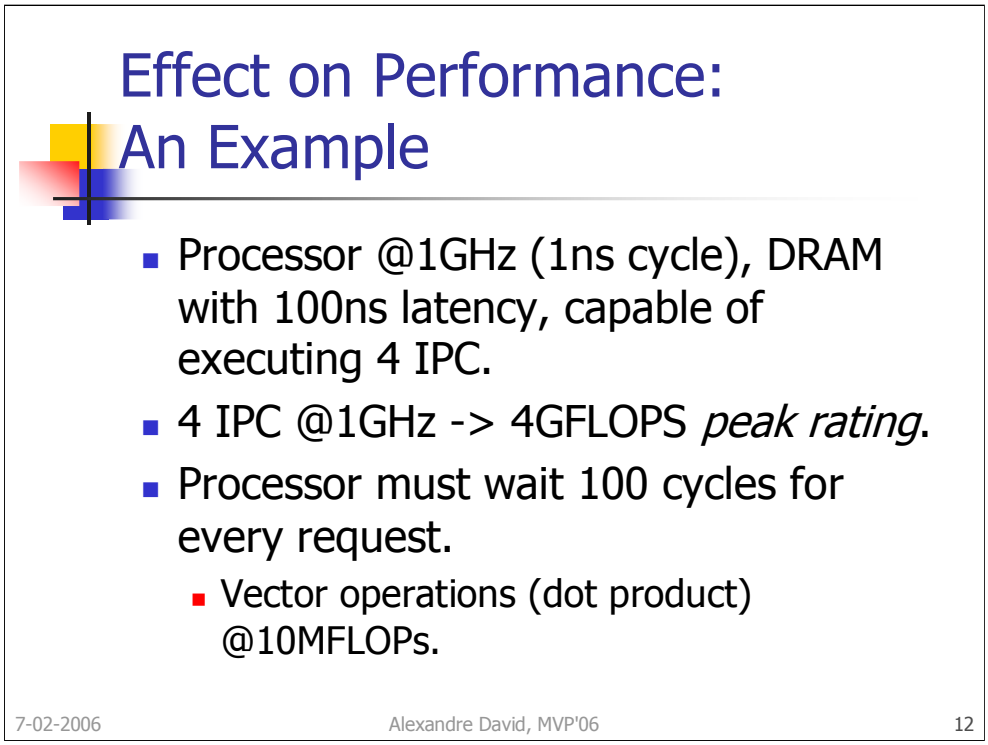


The problem is most often how to feed the processor with continuous data so that it does not **stall**.

Latency is the time from the issue of a memory request to the time the data becomes available to the processor.

Bandwidth is the rate at which data can be pumped to the processor.

Example: water hose. Latency: time before first drop of water comes out. Bandwidth: rate flow of water.

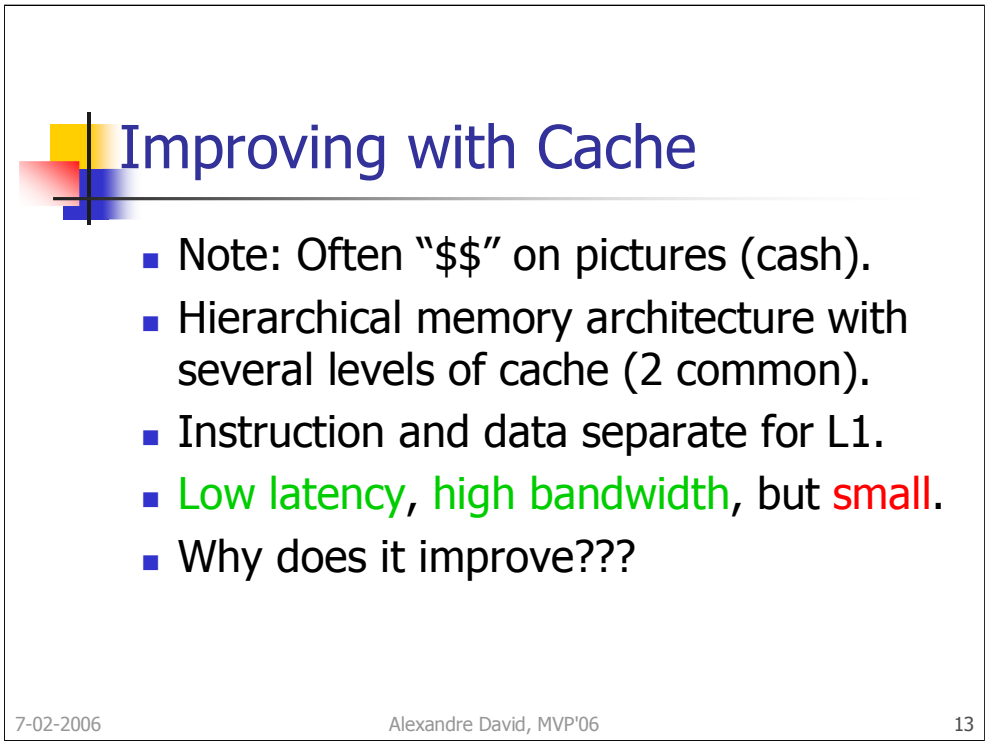


No cache in this example to simplify. It is still general enough since we can consider first access to some memory and take cache miss into account.

ALU: arithmetic and logical unit.

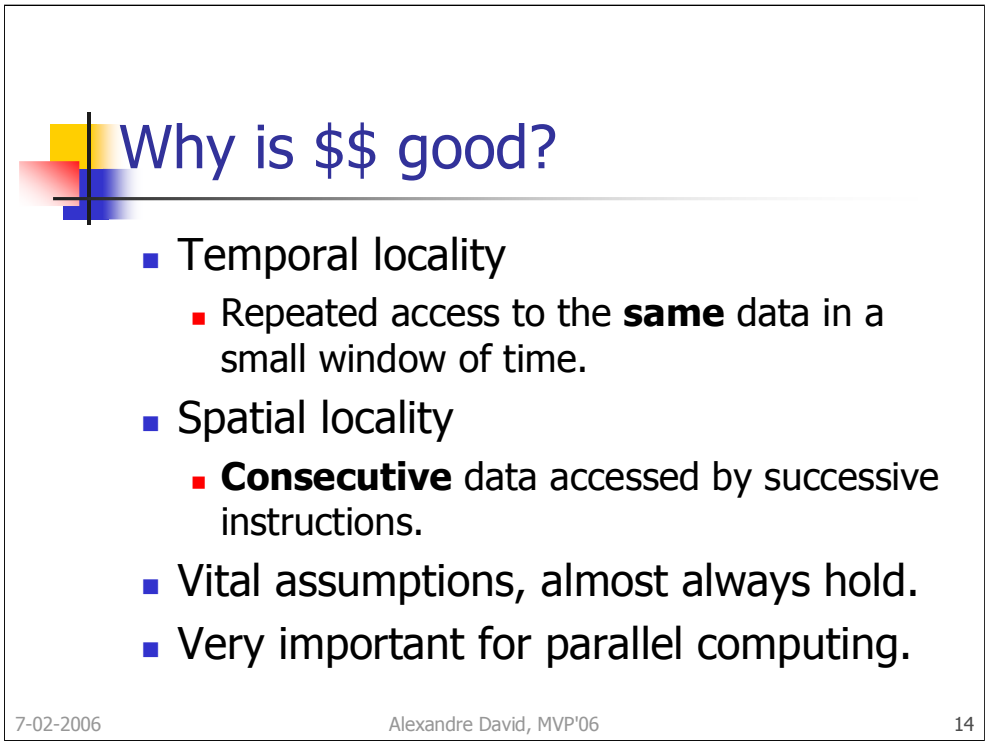
FPU: floating point unit.

Here absolute worst case scenario but still we loose a factor 100 in performance.



Common: Athlon 64 64K+64K L1, 1M L2. Pentium 4 more complicated NetBurst with execution trace cache (12K) and 16K L1, with 1M L2.

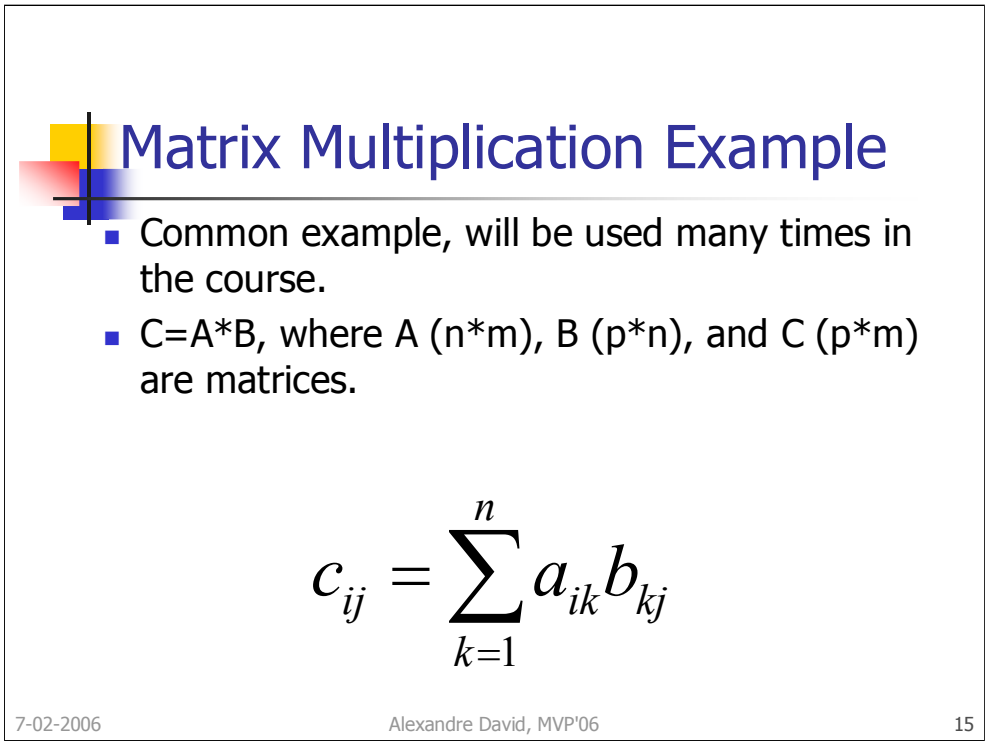
Now you have to think some time about why it helps. You know about cache hit ratio, cache miss, at least you've heard about it.



REMEMBER these two! They are common to almost all programs and are vital to cache performance.

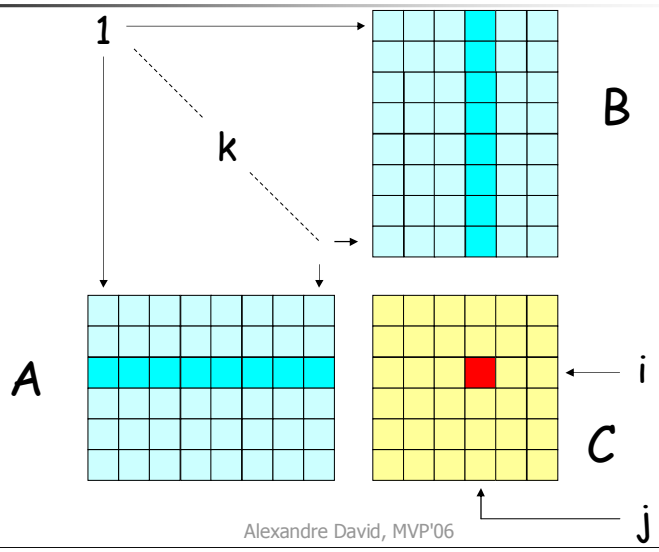
For parallel computing, even more important: apart from the aggregate higher amount of cache that must be used wisely, we have more penalty for moving data around processors (or processor nodes).

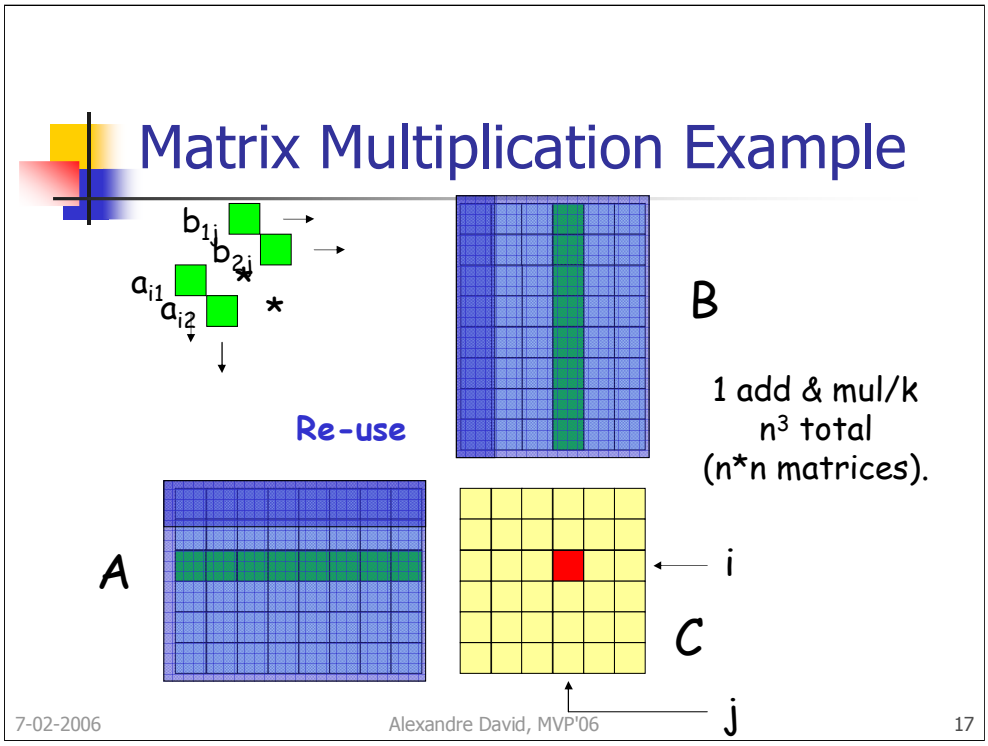
That also explains the model numbering for AMD processors. ☺



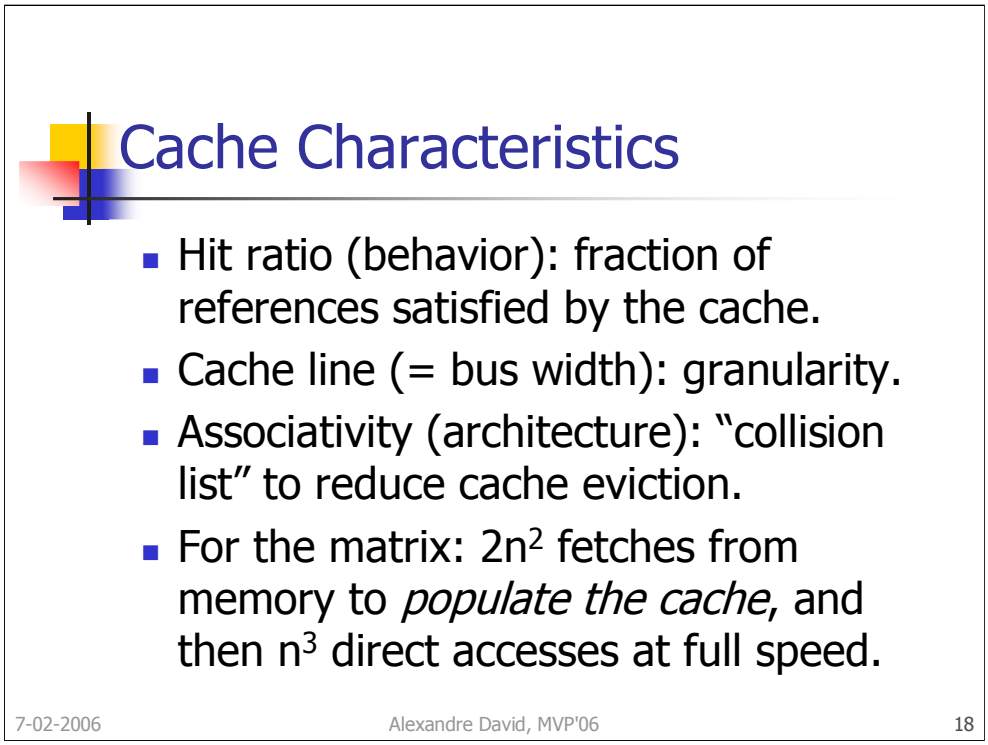
Compatible dimensions required. In practice n*n.

Matrix Multiplication Example





Re-use: spatial and temporal localities. Intuitively: n^3 accesses on 2x n^2 matrices (if n*n).



Data re-use is the keyword. Cache line: word granularity is too expensive and bad for spatial locality. 4 words usually for L2 (access to system bus), and internally 256-bit data bus for L1<->L2 (8 words).

The term *cache eviction* is not mentioned in the book and is missing. It is an important notion to know.

Impact on Memory Bandwidth (and Latency)

- Access to successive words much better than random access.
  - Higher bandwidth (whole cache line at once)
  - Better latency (successive words already in cache)

Example: Strided Access

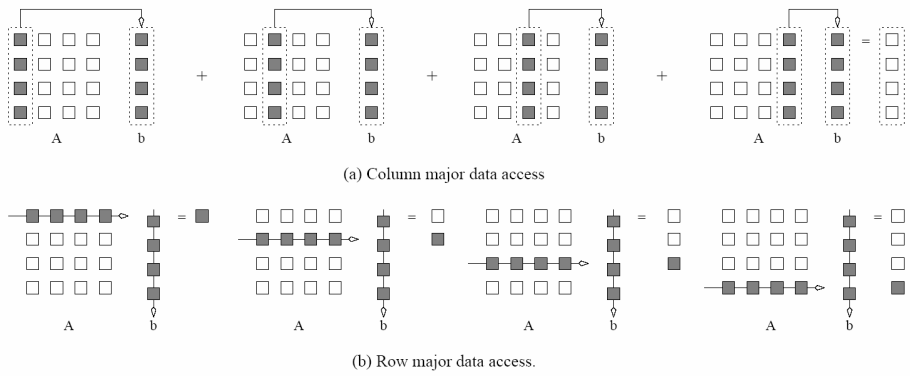
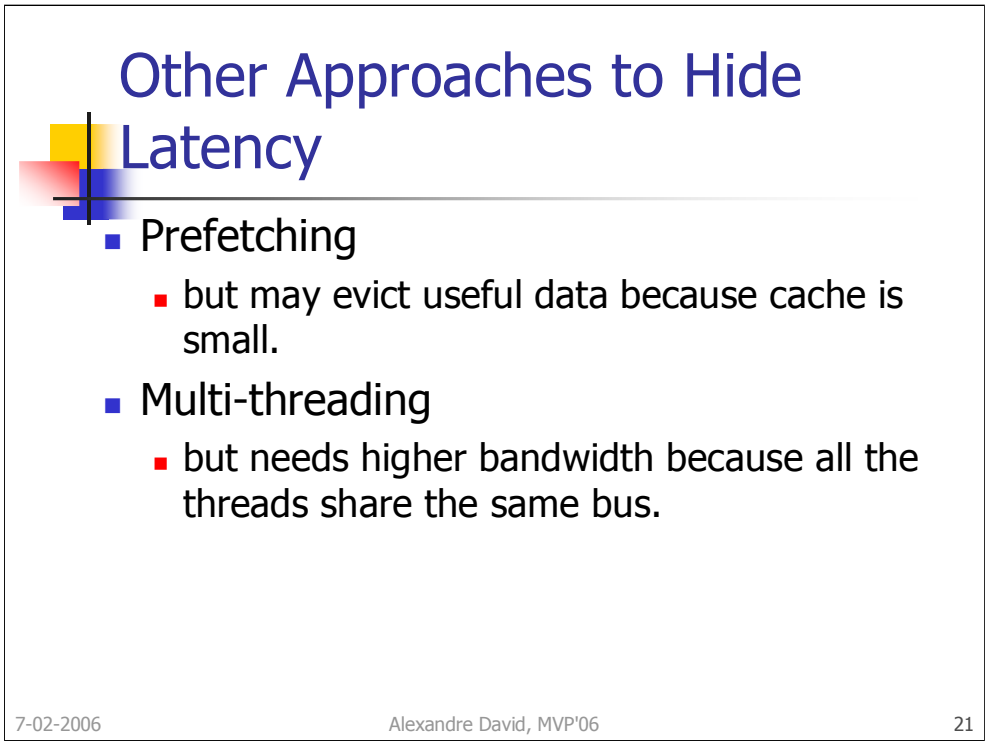


Figure 2.2 Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

a) Add vectors (temporary results) to get final result.

b) Compute final result incrementally.

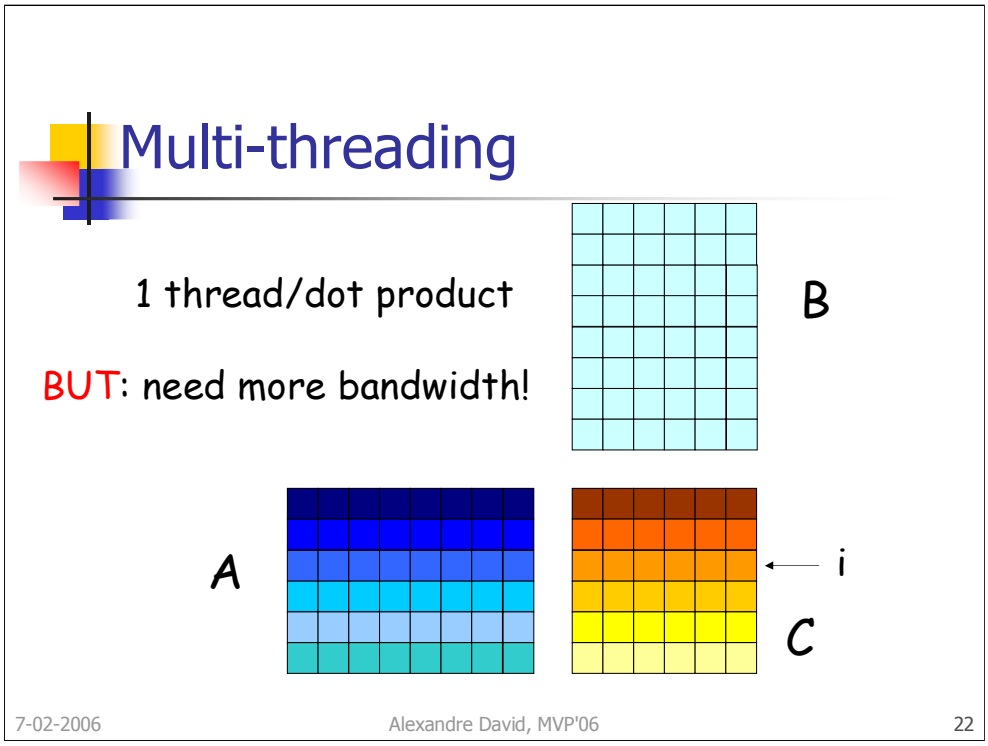
Strided access (a) yields poor performance.



Prefetching is like fetching the next cache line when possible (hardware decides), or same effect by reordering instructions (hardware or compiler) to issue loads long before usage. Works if consecutive words are accessed by consecutive instructions (spatial locality).

Multi-thread: switch to another thread when a thread stalls for data and keep the processor busy.

In fact, both solutions address the latency problem and exacerbate the bandwidth problem. That was probably the design idea behind RAMBUS, though no multi-threading at the time to use it! + the fact that latency was way higher than other systems.

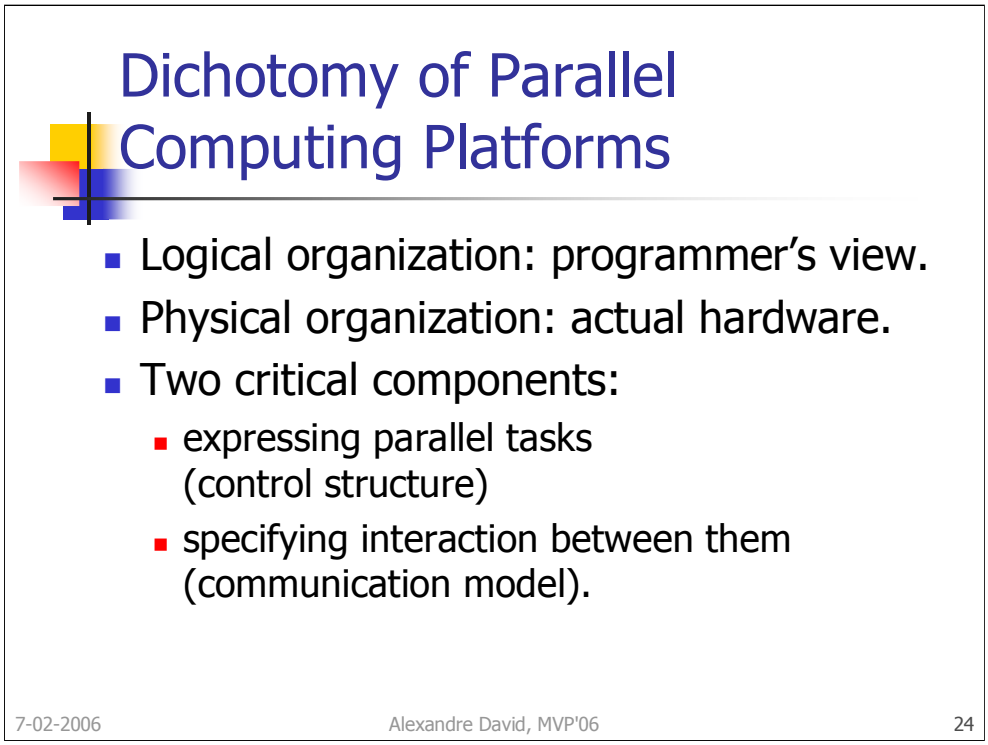


Software: need to create the threads explicitly.

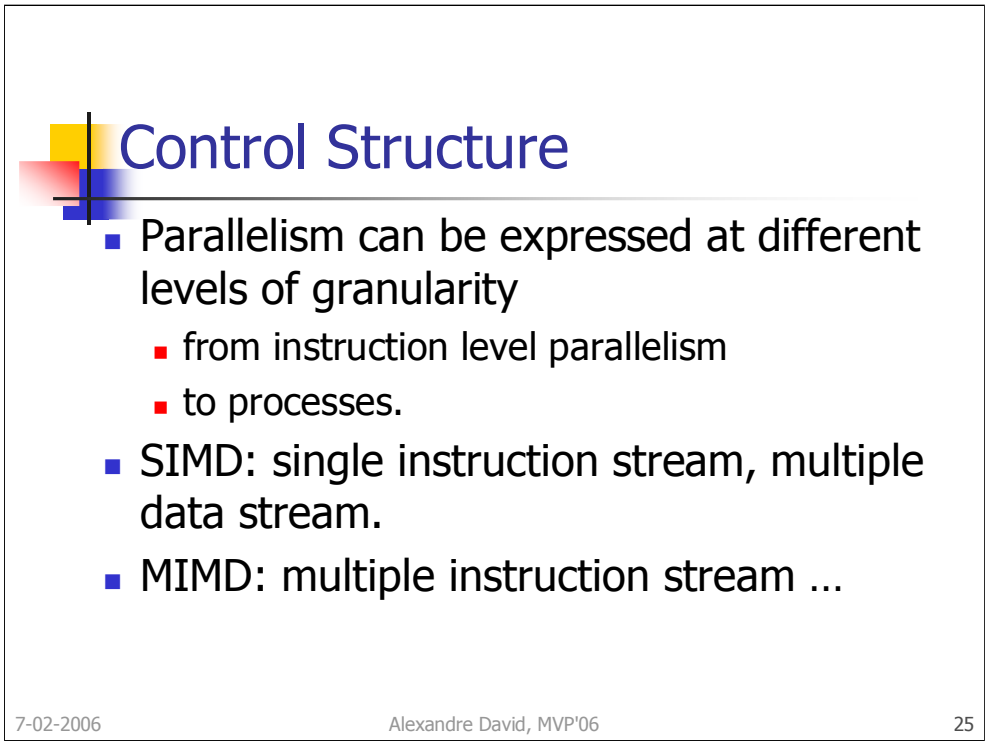
Hardware: need support for multi-thread.

Summary on Memory

- Exploit spatial and temporal locality in programs. For sequential and parallel programs!
- Operations/memory accesses is a good indicator of tolerance to memory bandwidth.
- Processing is cheap, memory is expensive.



The 2 critical components both through logical and physical organizations.



Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.

•SIMD: A single control unit dispatches the same instruction to various processors or functional units.

•MIMD: Each processor has its own control unit and can execute different instructions on different data items.

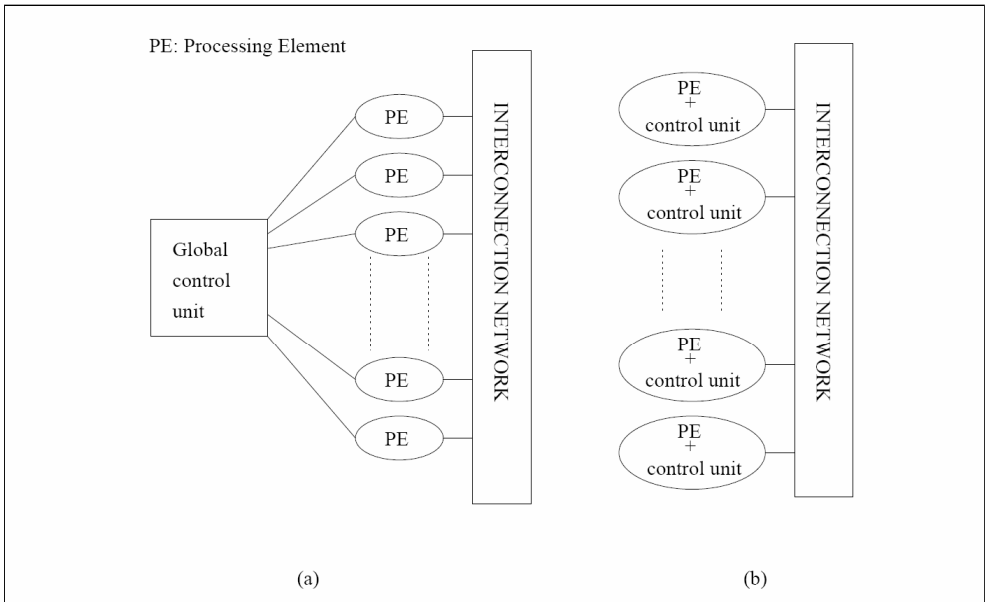


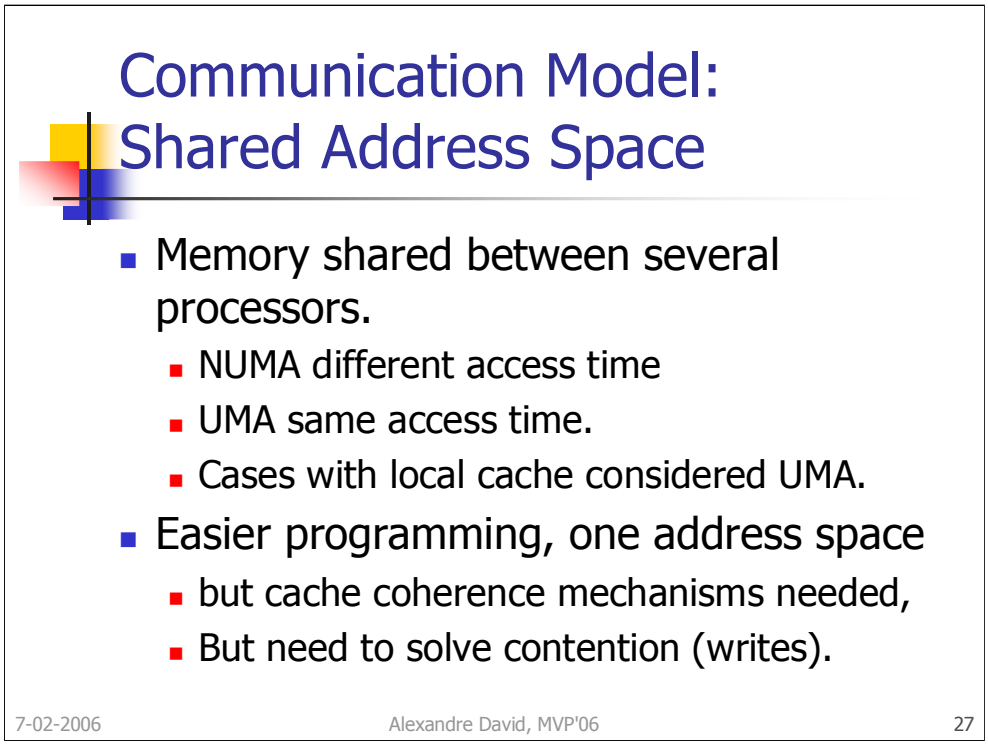
Figure 2.3 A typical SIMD architecture (a) and a typical MIMD architecture (b).

PE: processing element.

Typically, SIMD implemented as special instruction sets on processors (MMX, SSE, SSE2, 3DNow!), and MIMD implemented as multiprocessor machines or clusters.

SIMD relies on the regular structure of computations (such as those in image processing).

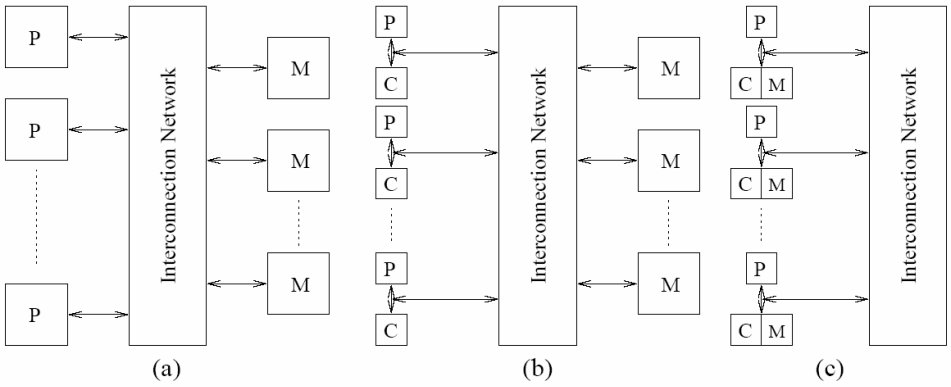
A variant of MIMD, called single program multiple data streams (SPMD) executes the same program on different processors, which is often the case in practice.



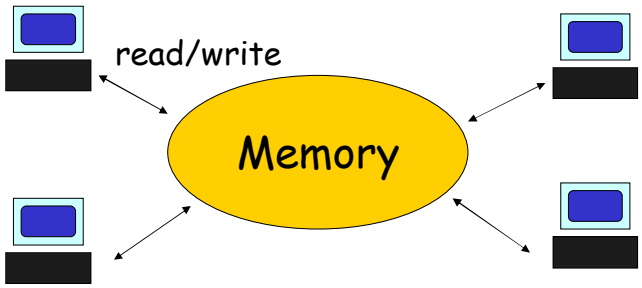
NUMA: non-uniform memory access. Cheaper and easier to implement. Need locality to perform well.

UMA: uniform memory access. Performant uniform access expensive.

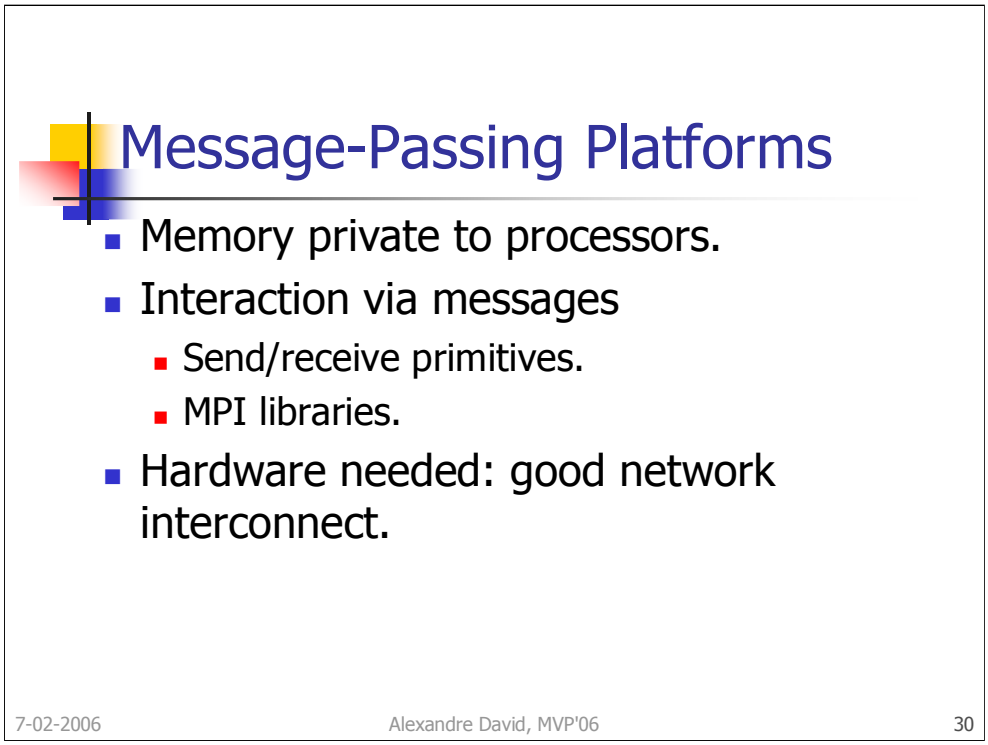
UMA vs. NUMA


P
P
P
Interconnection Network
M
M
M
(a)
P
C
P
C
P
C
Interconnection Network
M
M
M
(b)
P
C M
P
C M
P
C M
Interconnection Network
(c)

Communication Model: Shared Address Space



Implemented as shared memory computers or distributed memory computers.



Cheap and popular solution: cluster of MP machines connected via high bandwidth network.