Parallel Architectures

- Introduction
- Multithreading
- Multiprocessors (shared memory architecture)
- Multicomputers (message passing architecture)

Explicit Parallelism

- So far, we considered various ways to exploit the implicit parallelism embedded in the instructions that make up a program.
- The programmer conceives a program to solve a given problem as a sequence of instructions that will be executed by the CPU *one after another*
- The programmer ignores (he is entitled to do so) how the compiler will manage these instructions (static ILP) and how the CPU will actually execute them (dynamic ILP).

Explicit Parallelism

- What can we do if the program runs too slowly ? Well, we can surely use a “quicker” CPU, and possibly code in a better way the program. And once all this has been done?
- The only way is using an architecture that embeds more computing units; possible, at least some fraction of the problem can be parallelised.
- If the programmer is aware of the parallel architecture, he can choose an algorithm that can exploit this new capability: parallel execution.

Limits in Explicit Parallelism

- Actually, many problems lend themselves to a solution based on a set of programs running in parallel.
- However, (but for very particular instances) the increase in performance (**speed-up**) one can obtain by using multple CPUs to run many programs in parallel is **less than linear in the number of available CPUs (or cores)**.
- Indeed, programs running in parallel must synchronize, to share the data produced by each of them; alternatively, a preliminary common phase is required, before the actual parallel computations can start: in either case, there is always a fraction of computation that cannot be carried out in parallel.

Limits in Explicit Parallelism

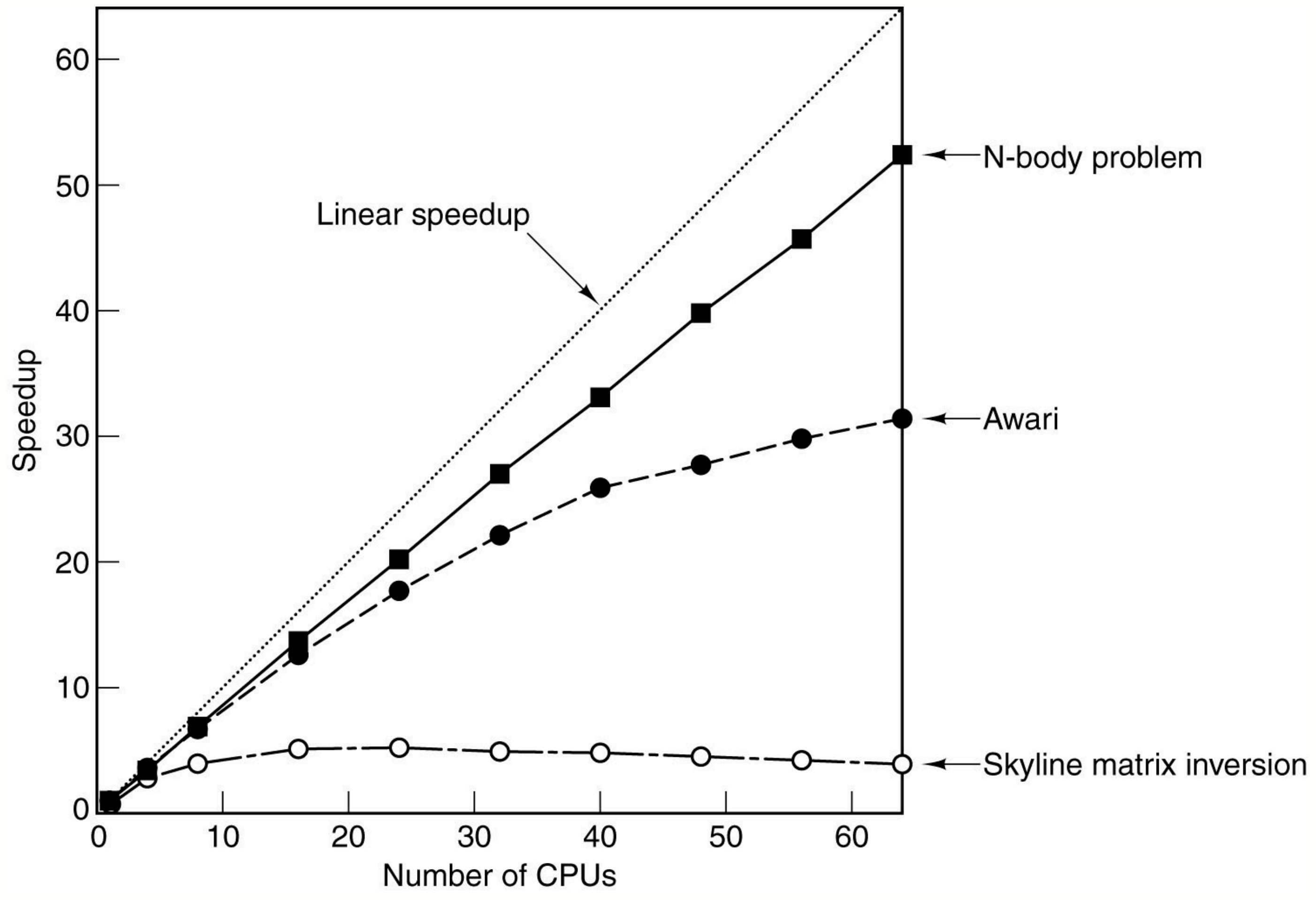
- Let us consider the following compilation command:
  - gcc main.c function1.c function2.c –o output
- Let us assume to compile on a mono-processor architecture, with the following figures:
  - 3 seconds to compile main.c
  - 2 seconds to compile function1.c
  - 1 second to compile function2.c
  - 1 second to link object modules (main.o, function1.o, function2.o)
- giving a total of 7 seconds.

Limits in Explicit Parallelism

- With three CPUs, the three sources can be compiled in parallel to produce the correspoding object modules.
- Once the objects have been generated, they can be linked using one of the three processors.
- However, linking can start only after all three object codes have been generated, that is, only after 3 seconds.
- The total time to produce *output* is 3+1=4 seconds, against 7 seconds in the mono-processor, with a speed-up of 7/4 = *1.75*, with *3* processors !!
- Even if all compilations required 1 second, the speed-up would be 4/2 = *2*.
- What happen if the RAM is not shared among the CPUs?

Limit in Explicit Parallelism

- Computational tasks exhibit a different "outcome" (formally, a different *speed-up*) when one tries to solve them by distributing the work among multiple CPUs (Tanenbaum, Fig. 8.47):



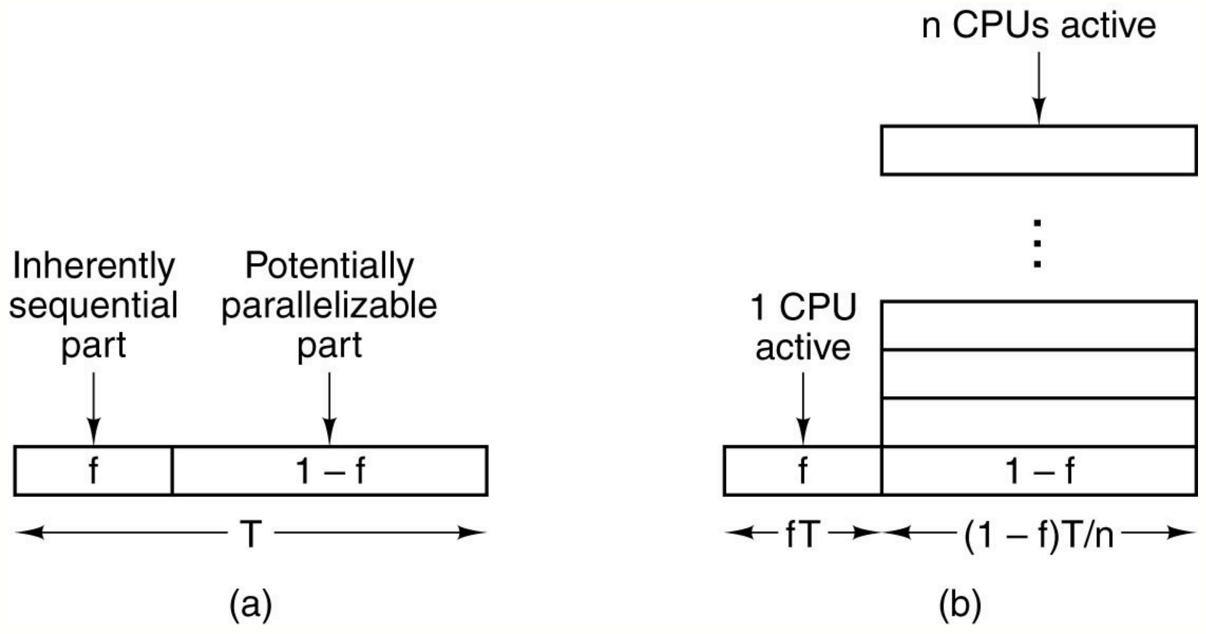
A typical operational research problem

a checkboard game

a 5 speed-up maximum problem, whichever the # of CPUs

Limits in Explicit Parallelism

- Every program consists (also) of a set of sequential operations for which no parallel execution is possible. Formally:
- Let P be a program run in time T on a single processor, and let f be the fraction of T due to inherently sequential code, and (1-f) the fraction due to parallelizable code (Tanenbaum, Fig. 8.48):



Limits in Explicit Parallelism

- The excution time due to the parallelizable fraction changes from (1-f)T to (1-f)T/n if n processors are available.
- The speed-up is obtained as the ratio of execution time in a single CPU over execution time on n CPUs:

\

$$\text{speed-up} = \frac{fT + (1 - f)T}{fT + (1 - f)T/n} = \frac{n[fT + (1 - f)T]}{nfT + (1 - f)T}$$

$$= \frac{nT}{T(1 + nf - f)} = \frac{n}{1 + (n-1)f} \rightarrow$$ **Amdahl's Law**

Limits in Explicit Parallelism

- Amdahl's Law states that a perfect speed-up, equal to the number of available CPUs, is only possible if $f = 0$.
- As an instance, which fraction of the original computation can be sequential, if we want a speed-up of 80 with 100 CPUs?
- $80 = 100 / (1 + 99f)$; $f = 20/(80 \times 99) = 0.0025252525$
- That is, only 0.25% of the original computation time can be due to sequential code.
- Is Amdhal's Law valid in single-core architectures?
- Can we think of a "superlinear" speed-up, in actual cases ?
- Amdhal's Law considers a fixed-size problem – another view point is running larger problems (Gustavson's Law to be discussed later)

Limits in Explicit Parallelism

- Actually, the main problems with Explicit Parallelism are two: 1) software and 2) hardware.

1. **The limit amount of parallelism available in programs**, or at least the amount that can be made explicit and thus used and deployed.

- Parallel algorithm design is still today a very active research area, quite because of the potential gains it offers.

2. **High costs of processor/memory communication**, which can raise considerably the cost of a cache miss, or the synchronization between two processes run on different CPUs.

- These costs depend both on the architecture and on the number of CPUs, and are generally much higher than in a uniprocessor system.

Explicit Parallelism

- Of course, achieving a sub-linear speed-up (e.g. n with 2n processors), is very welcome in many applications, since CPUs cost is decreasing.

- The increase in computational power is not the only driving force for designing multi-CPU Architectures:

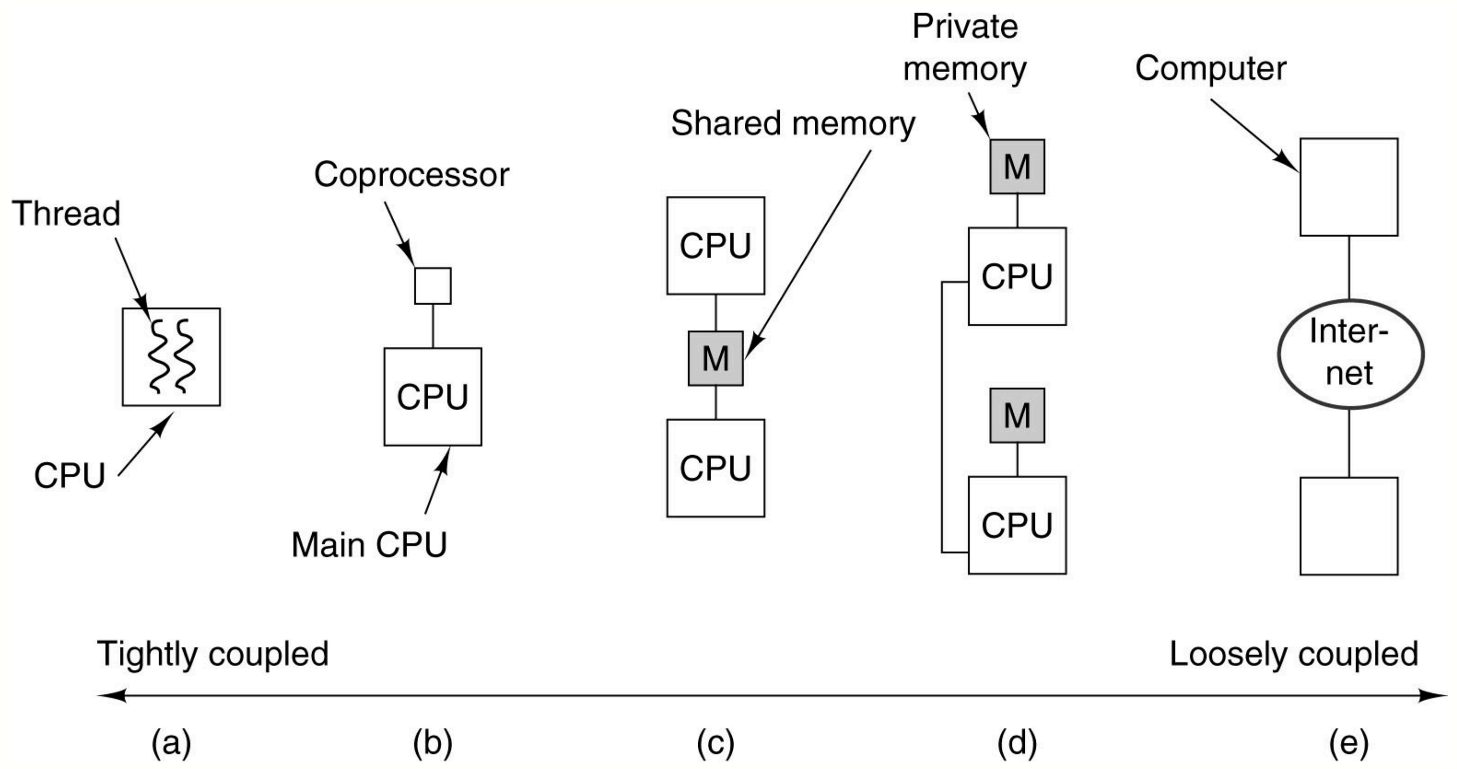
a) Having multiple processors raises systems reliability: if one of them fails, the others can step in and carry out its work.

b) Services that are inherently issued on a geografical scale must be realized with a distributed architecture. If the system were centralized in a single node, concurrent accesses to this node would become a bottleneck and would slow down the service offered, ultimately making it unavailable.

Explicit Parallelism

- At a coarse scale, we can list three types of explicit Parallel Architectures (Tanenbaum, Fig. 8.1):

1. Multi-threading (a)
2. Shared-memory systems (b,c)
3. Distributed-memory systems (d,e)



A taxonomy of computer architectures

Architectures taxonomy

- We have considered different types of architectures, and it is worth considering some way to classify them.
- Indeed, there exists a famous taxonomy of the various architectures: it is well known, though it is rough and not precise (it was conceived back in 1966 !!): **Flynn's taxonomy** (Tanenbaum, Fig. 8.20):

Instruction streams	Data streams	Name	Examples
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

Architectures taxonomy

- Flynn’s taxonomy uses two basic concepts: parallelism in *instruction* stream, and paralleism in *data* streams.
- A *n* CPU system has *n* program counters, so there are *n* “*instruction streams*” that can execute in parallel.
- A *data stream* can be thought of as a sequence of data. In a stream, each data is processed in the sequence it belongs to. There can be multiple independent streams, with the computation carried out on a stream being separate and distinct from that carried out on another stream.
- Data and instruction streams are, to a certain degree, orthogonal, and there exist 4 possible, different combinations:

Architectures taxonomy

- **SISD: Single Instruction (stream) Single Data (stream)** it is the classical uniprocessor architecture, where instructions are executed one at a time on a single data stream: variables in the program being executed.
- Owing to ILP, this view if fairly inaccurate (just consider the operation of a single pipeline), and it is currently correct only for simple processors in embedded applications, such as Intel 8051.
- Intel 8051 familty processors have no pipeline, (most instructions take a single clock cycle), they issue and execute instructions in order, and have no cache.
- They are 8-bit architectures, have a clock frequency of some tens of MHz, cost10, 20 euro cents, and are by large the processor that sell the most: some 8 BILLION pieces per year !

Architectures taxonomy

- They are deployed in alarm clocks, in wash machines, in microwave owens, in cordless phones, in some "electronic" toys, in some medical appliances, and so on.
- Classifying superscalar, multiple-issue processors in the SISD category is somewhat dubious, but is common practice. Actually, modern processors are the off-spring of the classical Von Neumann architecture

Architectures taxonomy

- **MISD: Multiple Instruction (stream) Single Data (stream):** is this a class of architectures that makes any sense? Many authors think it does not.
- However, some argue that considering modern processors an instance of SISD architectures is less precise than ascribing them to the MISD class: the data to be processed flow from one instruction to the next as the instructions flow within the stages of the pipeline.

Architectures taxonomy

- **SIMD: Single Instruction (stream) Multiple Data (stream):** the SIMD model was adopted in one of the first models of parallel architectures ever proposed, the well known Illiac IV.

- Illiac IV was designed in the mid 60's and was actually built in 1972: it cost 31 million $ (1972 $) and had a computational power of some 50 MFLOPS (the initial target was 1 GFLOP).

- Illiac IV is the most famous case of an architectural model currently almost disappeared, known as **array processor**: a huge number of identical processors that execute the same sequence of instructions on different data streams.

Architectures taxonomy

- There is a second type of SIMD architectures, conceptually very similar to array processors: **vector processors**.
- Vector processors work on arrays of data, but a single processor is in charge of carrying out the operation on all elements in the array.
- These processors did have some success in the “supercomputer” arena (some CRAY models adopted this architectures), but are currently declining.

Architectures taxonomy

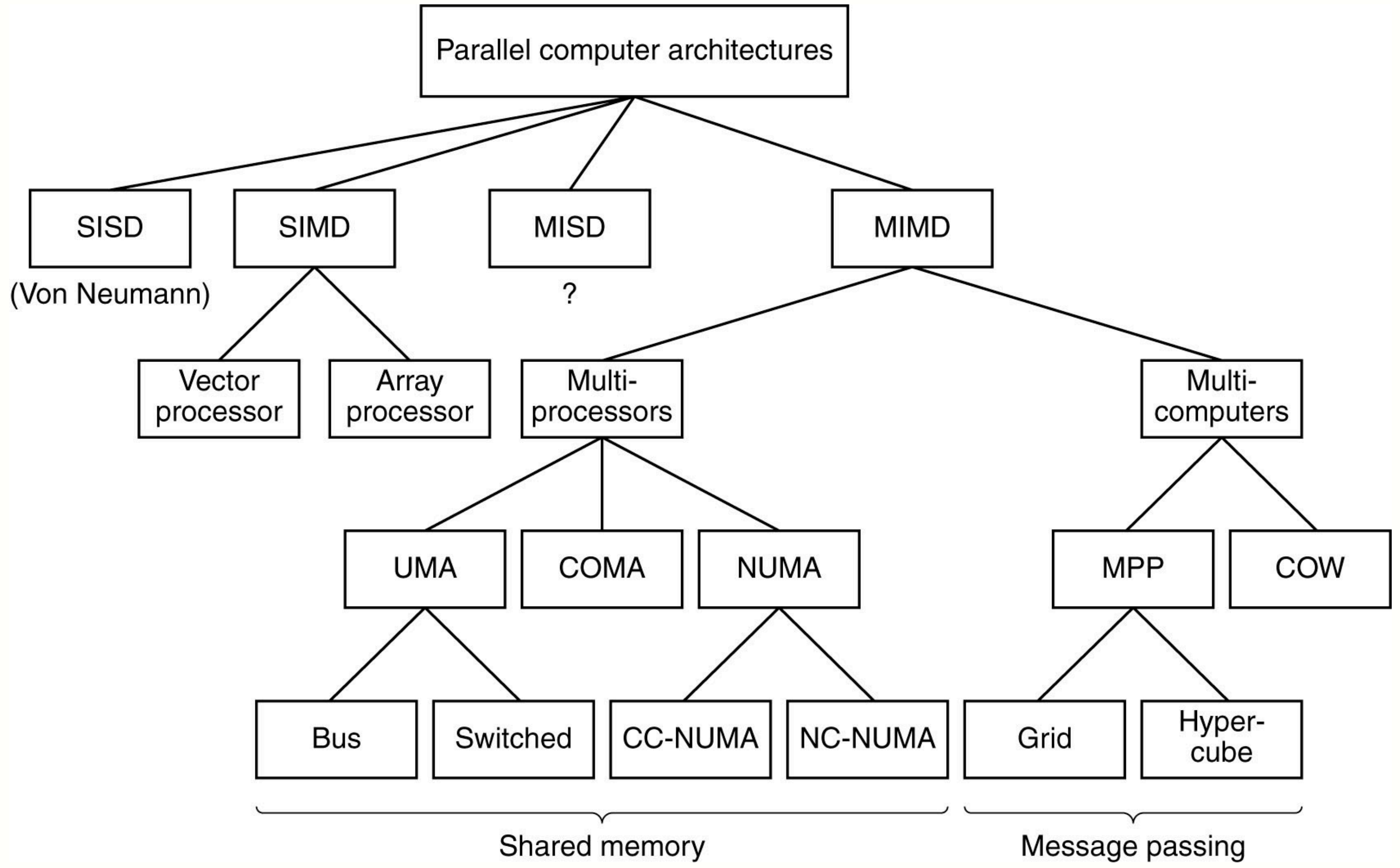
- The SIMD paradigm has been used also within the microarchitecture of general purpose processors by extending their ISA to handle multimedia data. It the so called "**multimedia extensions**".
- But the most notable exploitation of this paradigm is currently in the graphics processing domain, where GPU (**Graphical Processing Unit**) model has produced a specific implementation of SIMD architecture for handling multiple data streams of graphic objects.

Architectures taxonomy

- **MIMD: Multiple Instruction (stream) Multiple Data (stream):** This encompasses all multiprocessor and multicomputer architectures (from dual core processors, to small UMA systems, up to clusters such as Google).
- It is worth underlining that Flynn's taxonomy is really coarse, and there are architectures that do not fit well in this scheme:
  - – multithreaded processors : SISD or MIMD?
  - – general purpose processors with SIMD extensions: where do they belong?

Architectures taxonomy

- Flynn’s taxonomy (Tanenbaum, Fig. 8.21).


Parallel computer architectures
SISD
SIMD
MISD
MIMD
(Von Neumann)
?
Vector processor
Array processor
Multi-processors
Multi-computers
UMA
COMA
NUMA
MPP
COW
Bus
Switched
CC-NUMA
NC-NUMA
Grid
Hyper-cube
Shared memory
Message passing