Thread-Level Parallelism and Multiprocessing

Introduction

- Initial computer performance improvements came from use of:
  - Innovative manufacturing techniques
  - Advancement of VLSI technology
- In later years,
  - Most improvements came from exploitation of ILP.
  - Both software and hardware techniques are being used.
  - Pipelining, dynamic instruction scheduling, out of order execution, VLIW, vector processing, etc.
- ILP now appears fully exploited:
  - Modern multiple-issue processors have become incredibly complex and processor performance improvement through increasing complexity, increasing silicon and increasing power seem to be diminishing.

Parallel Execution of Programs

- Parallel execution of a program can be done in one or more of following ways:
  - Instruction-level (Fine grained): individual instructions of any one thread are executed in parallel.
    - Parallel execution across a sequence of instructions (block) -- could be a loop, a conditional, or some other sequence of statements.
  - Thread-level (Medium grained): different threads of a process are executed in parallel.
  - Process-level (Coarse grained): different processes can be executed in parallel.

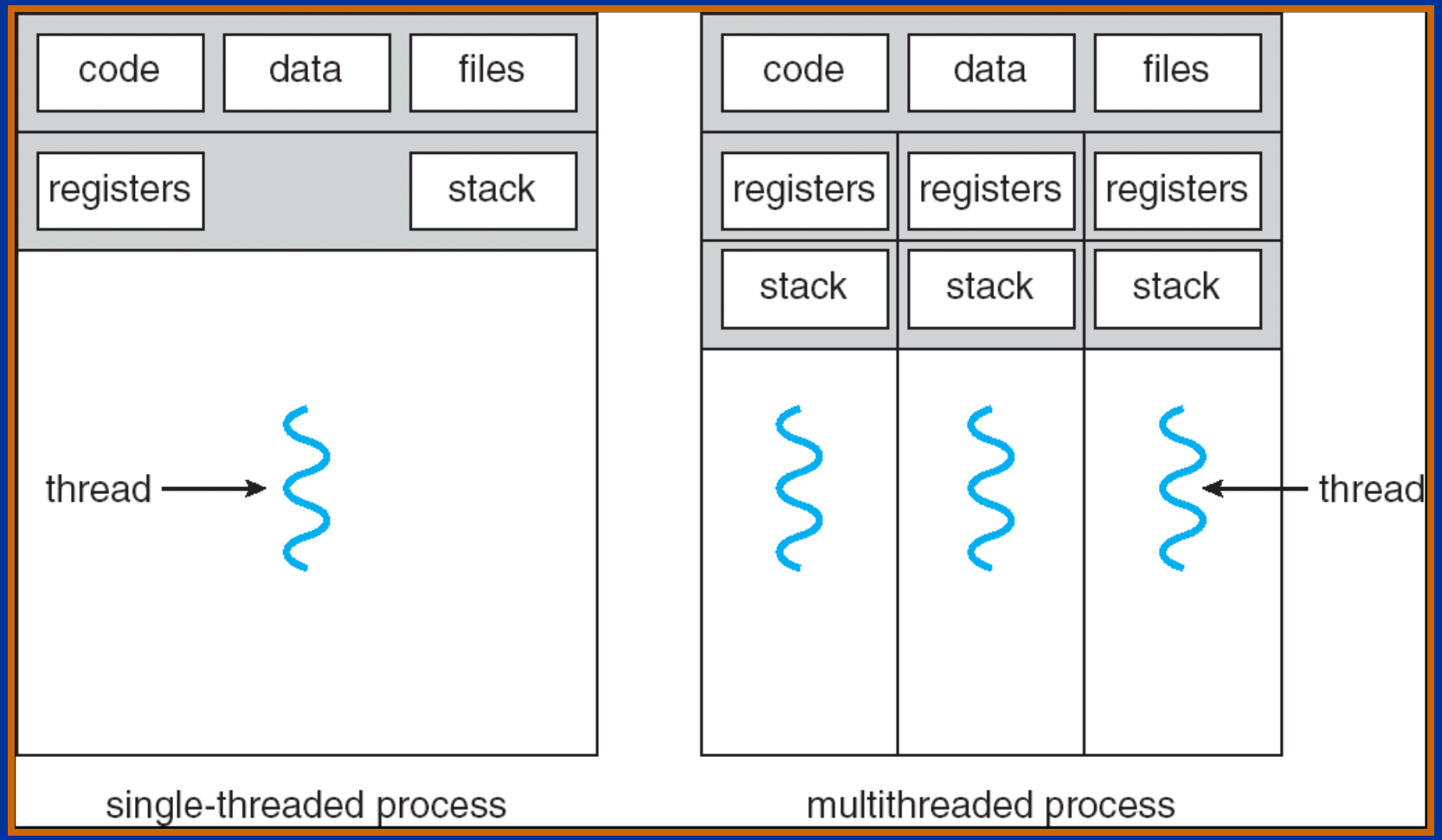
Thread and Process-Level Parallelism

- The way to achieve higher performance:
  - Of late, exploitation of thread and process-level parallelism is being focused.
- Exploit parallelism existing across multiple processes or threads:
  - Cannot be exploited by any ILP processor.
- Consider a banking application:
  - Individual transactions can be executed in parallel.

Processes versus Threads

- **Processes:**
  - **A process is a program in execution.**
  - **An application normally consists of multiple processes.**
- **Threads:**
  - **A process consists of one or more threads.**
  - **Threads belonging to the same process share data, and code space.**

Single and Multithreaded Processes



How can Threads be Created?

- By using any of the popular thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads, etc.

User Threads

- Thread management done in user space.
- User threads are supported and managed without kernel support.
  - Invisible to the kernel.
  - If one thread blocks, entire process blocks.
  - Limited benefits of threading.

Kernel Threads

- Kernel threads supported and managed directly by the OS.
  - Kernel creates Light Weight Processes (LWPs).
- Most modern OS support kernel threads:
  - Windows XP/2000
  - Solaris
  - Linux
  - Mac OS, etc.

Benefits of Threading

- **Responsiveness:**
  - **Threads share code, and data.**
  - **Thread creation and switching therefore much more efficient than that for processes.**
- **As an example in Solaris:**
  - **Creating threads 30x less costly than processes.**
  - **Context switching about 5x faster than processes.**

Benefits of Threading

- **Truly concurrent execution:**
  - **Possible with processors supporting concurrent execution of threads: SMP, multi-core, SMT (hyperthreading), etc.**

A Case for Processor Support for Thread-level Parallelism

- Using pure ILP, execution unit utilization is only about 20%-25%:
  - Utilization limited by control dependency, Cache misses during memory access, etc.
  - It is rare for units to be even reasonably busy on the average.
- In pure ILP:
  - At any time only one thread is under execution.

A Case for Processor Support for Thread-level Parallelism

- Utilization of execution units can be improved:
  - Have several threads under execution:
    - called active threads in PIII.
  - Execute several threads at the same time:
    - SMP, SMT, and Multi-core processors.

Threads in Applications

- Threads are natural to a wide ranging set of applications:
  - Often more or less independent.
  - Though share data among themselves to some extent.
  - Also, synchronize sometimes among themselves.

A Few Thread Examples

- Independent threads occur naturally in several applications:
  - **Web server:** different http requests are the threads.
  - **File server**
  - **Name server**
  - **Banking:** independent transactions
  - **Desktop applications:** file loading, display, computations, etc. can be threads.

Reflection on Threading

- To think of it:
  - Threading is inherent to any server application.
- Threads are also easily identifiable in traditional applications:
  - Banking, Scientific computations, etc.

Using ILP Support to Exploit Thread-level Parallelism

- Possible processor configurations:
  - A superscalar with no multithreading support
  - A superscalar with coarse-grained multithreading
  - A superscalar with fine-grained multithreading
  - A superscalar with simultaneous multithreading

Thread-level Parallelism: Cons

- Threads have to be identified by the programmer:
  - No rules exist as to what can be a meaningful thread.
  - Threads can not possibly be identified by any automatic static or dynamic analysis of code.
  - Burden on programmer: requires careful thinking and programming.
- Threads with severe dependencies:
  - May make multithreading an exercise in futility.
- Also not as "programmer friendly" as ILP

Thread Vs. Process-Level Parallelism

- Threads are light weight (or fine-grained):
  - Threads share address space, data, files, etc.
  - Even when extent of data sharing and synchronization is low: Exploitation of thread-level parallelism is meaningful only when communication latency is low.
  - Consequently, shared memory architectures (UMA) are a popular way to exploit thread-level parallelism.

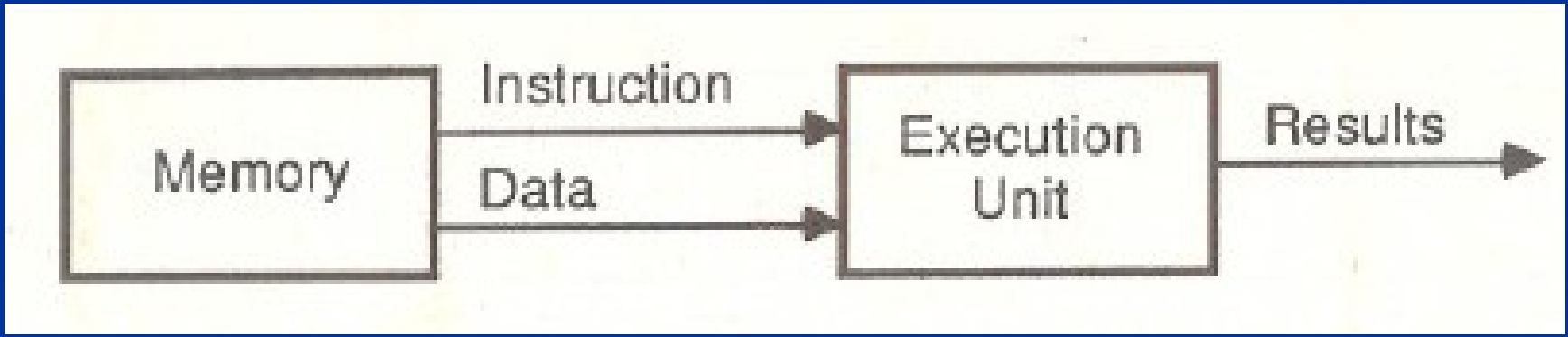
Thread Vs. Process-Level Parallelism

- Processes are coarse-grained:
  - Communication to computation requirement is lower.
  - DSM (Distributed Shared Memory), Clusters, Grids, etc. are meaningful.

A Taxonomy of parallel Architectures

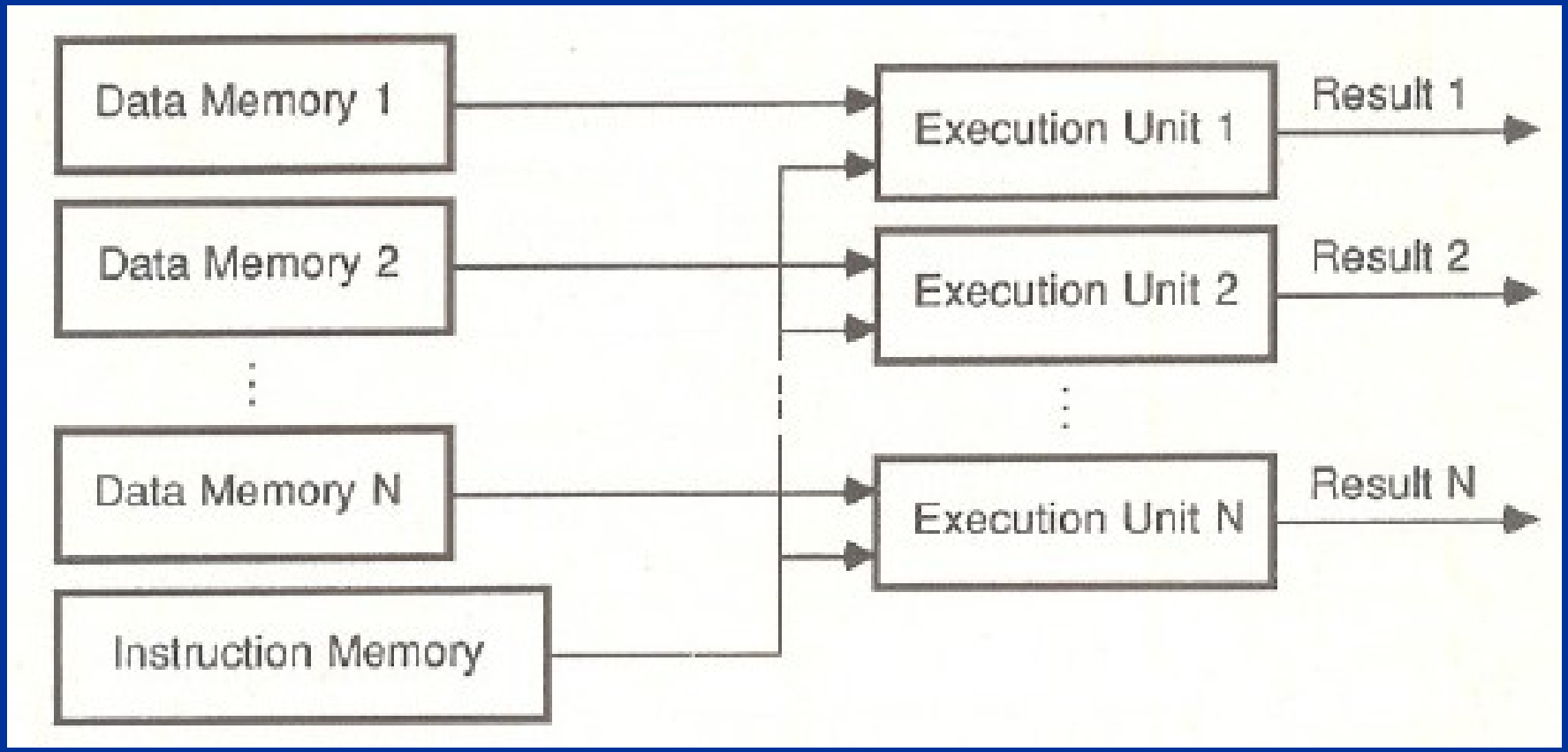
- In a landmark paper in 1966 Flynn outlined four classes for the organization of high-performance computers based on instruction and data stream

- SISD (Single Instruction Single Data) Uniprocessors



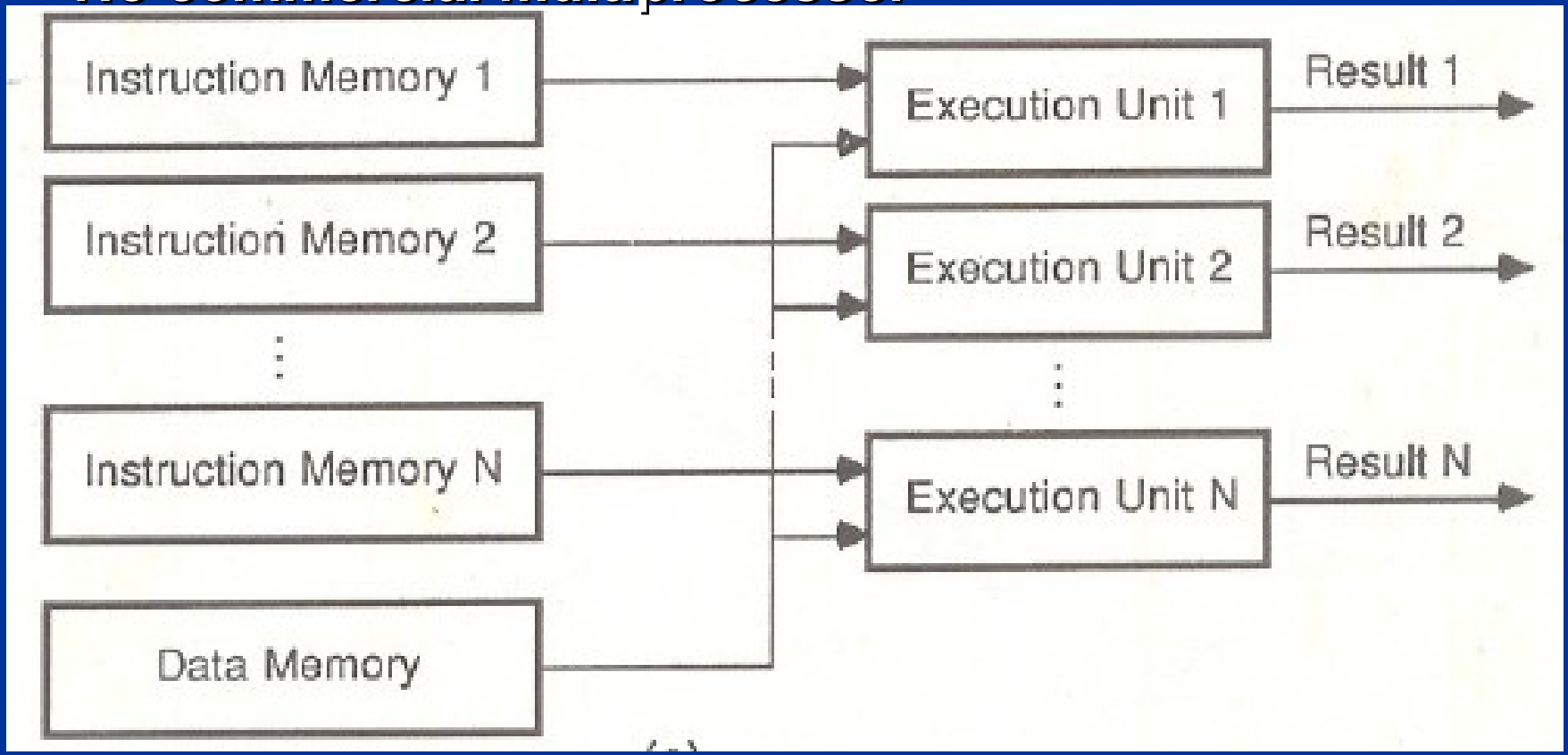
A Taxonomy of parallel Architectures

- SIMD (Single Instruction Multiple Data)
- Classic form of Array Processors, Vector Processor



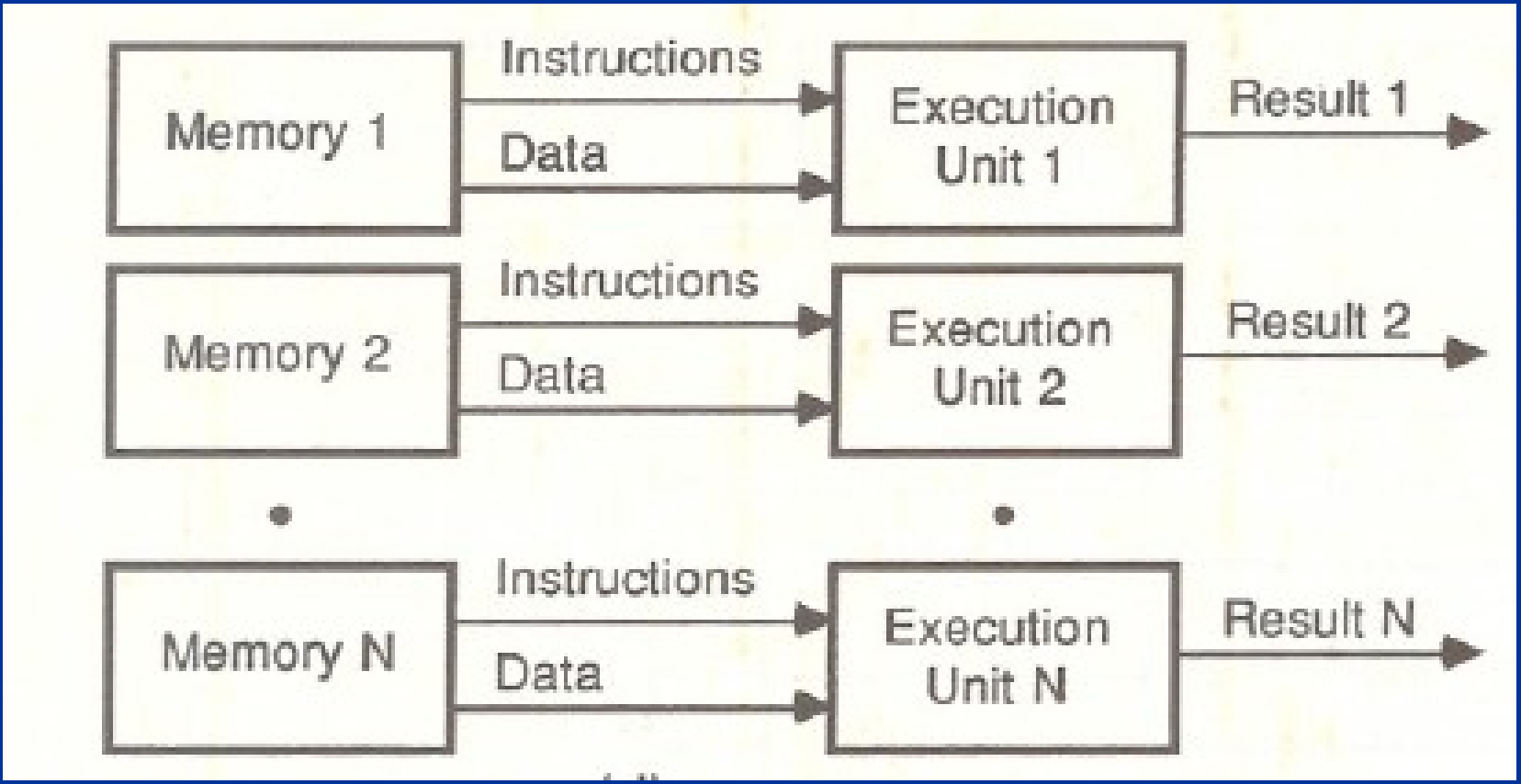
A Taxonomy of parallel Architectures

- **MISD** (Multiple Instruction Single Data)
- Systolic arrays for pipelined execution of specific algorithm
- No commercial multiprocessor



A Taxonomy of parallel Architectures

- **MIMD (Multiple Instruction Multiple Data)**
  - **General purpose, commercially important**

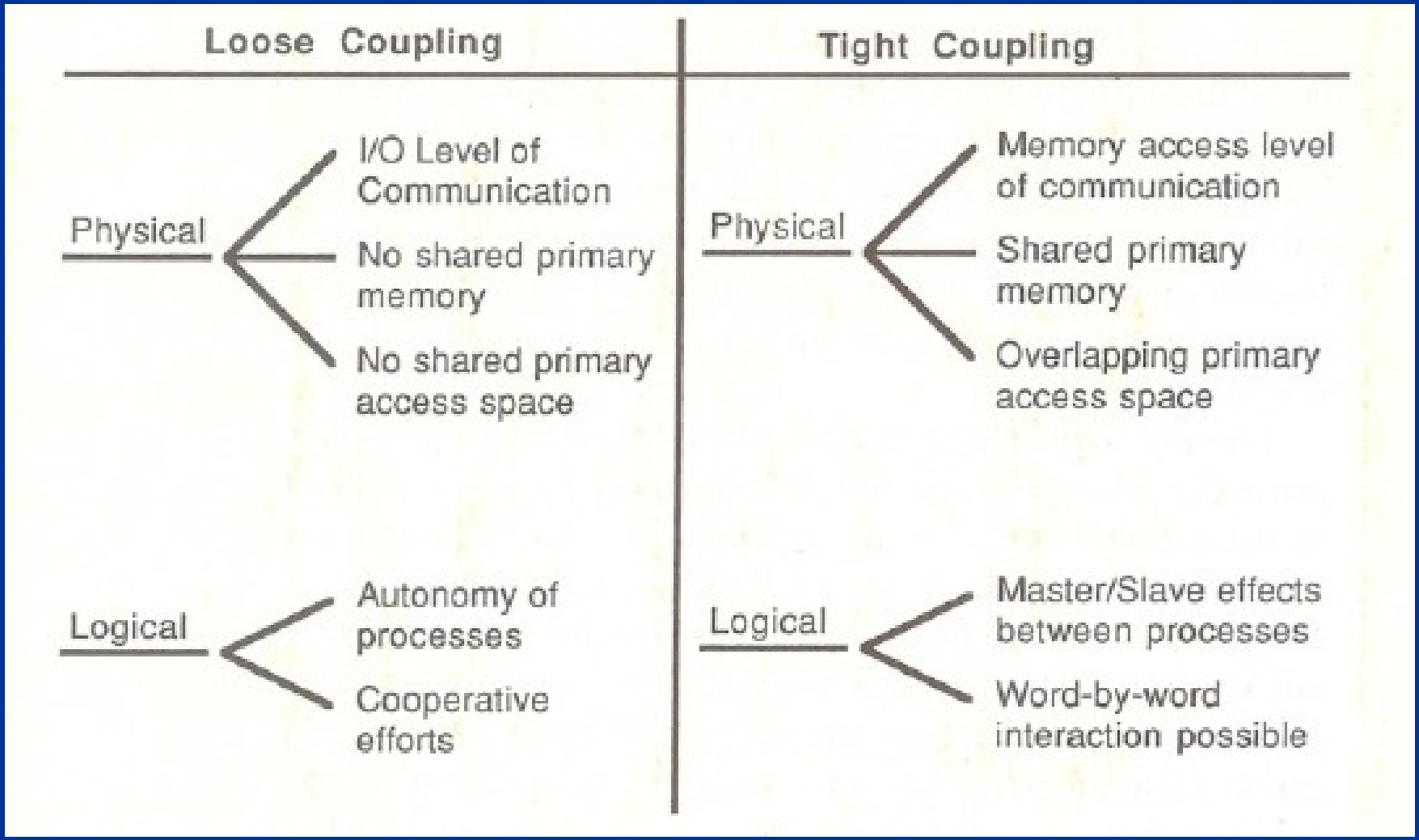


Classification for MIMD Computers

- Shared Memory
  - Processors communicate through a shared memory.
  - Typically processors connected to each other and to the shared memory through a bus.

- Distributed Memory
  - Processors do not share any physical memory.
  - Processors connected to each other through a network.

Classification for MIMD



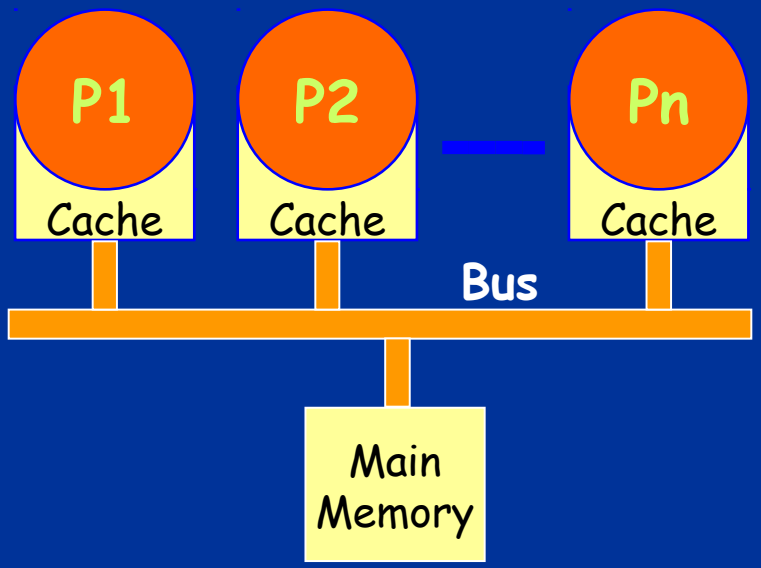
Shared Memory

- Shared memory located at a centralized location:
  - May consist of several interleaved modules – same distance (access time) from any processor.
  - Also called Uniform Memory Access (UMA) model.
  - Most popular

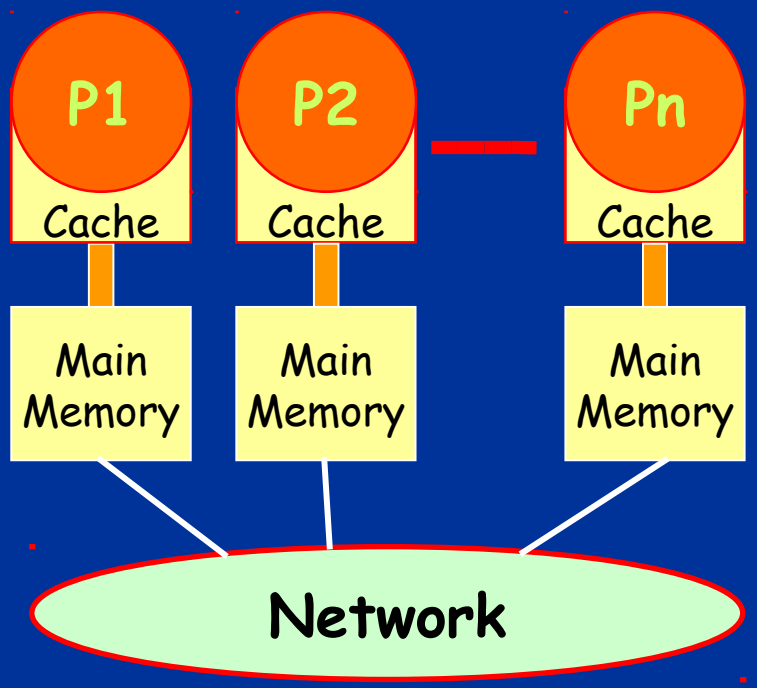
Distributed Memory

- Memory is distributed to each processor:
  - improves scalability.
  - Lower latency to access local memory
- Non-Uniform Memory Access (NUMA)
  - (a) Message passing architectures – No processor can directly access another processor's memory.
  - (b) Distributed Shared Memory (DSM)– Memory is distributed, but the address space is shared.

UMA vs. NUMA Computers



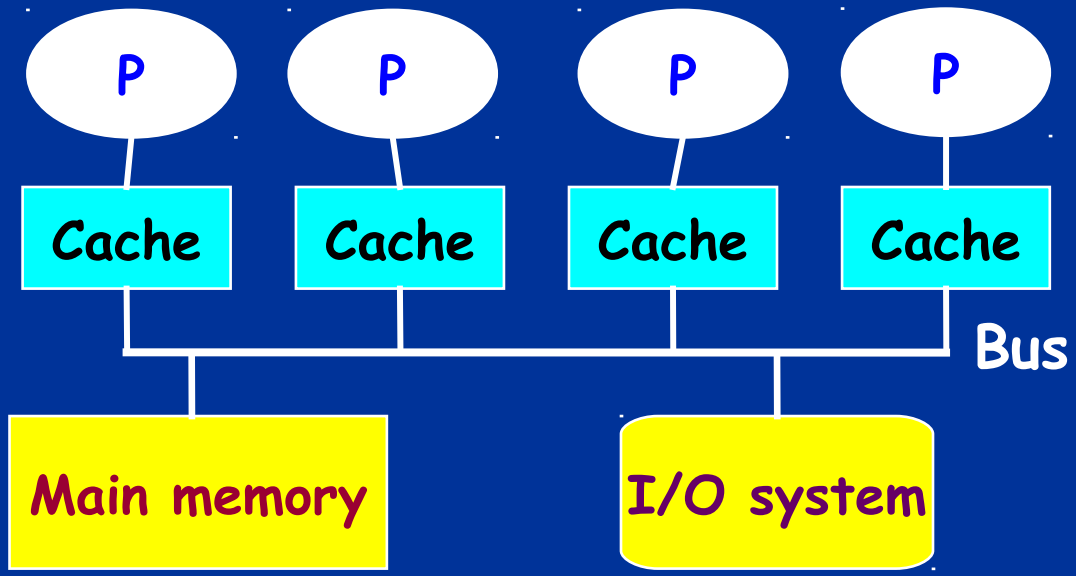
(a) UMA Model



(b) NUMA Model

Symmetric Multiprocessors (SMPs)

- SMPs are a popular shared memory multiprocessor architecture:
  - Processors share Memory and I/O
  - Bus based: access time for all memory locations is equal --- "Symmetric MP"



SMPs: Some Insights

- In any multiprocessor, main memory access is a bottleneck:
  - Multilevel caches reduce the memory demand of a processor.
  - Multilevel caches in fact make it possible for more than one processor to meaningfully share the memory bus.
  - Hence multilevel caches are a must in a multiprocessor system!

Different SMP Organizations

- Processor and cache on separate extension boards (1980s):
  - Plugged on to the backplane.
- Integrated on the main board (1990s):
  - 4 or 6 processors placed per board.
- Integrated on the same chip (multi-core) (2000s):
  - Dual core (IBM, Intel, AMD)
  - Quad core

Pros and Cons of SMPs

- **Pros: Ease of programming:**
  - **Especially when communication patterns are complex or vary dynamically during execution**
- **Cons: As the number of processors increases, contention for the bus increases.**
  - **Scalability of the SMP model restricted.**
  - **One way out may be to use switches (crossbar, multistage networks, etc.) instead of a bus.**
  - **Switches set up parallel point-to-point connections.**
  - **Again switches are not without any disadvantages: make implementation of cache coherence difficult.**

SMPs

- Even programs not using multithreading (conventional programs):
  - Experience a performance increase on SMPs
  - Reason: Kernel routines handling interrupts etc. run on a separate processor.
- Multicore processors are now a common place:
  - Pentium 4 Extreme Edition, Xeon, Athlon64, DEC Alpha, UltraSparc…

A Taxonomy of parallel Architectures