TCS 802
Advanced Computer Architecture

Hwang, Chapter 1

Parallel Computer Models

1.1 The State of Computing

Web link: http://pksmmmec.googlepages.com/

The State of Computing

- Early computing was entirely mechanical:
  - abacus (about 500 BC)
  - mechanical adder/subtracter (Pascal, 1642)
  - difference engine design (Babbage, 1827)
  - binary mechanical computer (Zuse, 1941)
  - electromechanical decimal machine (Aiken, 1944)
- Mechanical and electromechanical machines have limited speed and reliability because of the many moving parts. Modern machines use electronics for most information transmission.

Computing Generations

- Computing is normally thought of as being divided into *generations*.
- Each successive generation is marked by sharp changes in hardware and software technologies.
- With some exceptions, most of the advances introduced in one generation are carried through to later generations.
- We are currently in the fifth generation.

First Generation (1945 to 1954)

- Technology and Architecture
  - Vacuum tubes and relay memories
  - CPU driven by a program counter (PC) and accumulator
  - Machines had only fixed-point arithmetic
- Software and Applications
  - Machine and assembly language
  - Single user at a time
  - No subroutine linkage mechanisms
  - Programmed I/O required continuous use of CPU
- Representative systems: ENIAC, Princeton IAS, IBM 701

Second Generation (1955 to 1964)

- Technology and Architecture
  - Discrete transistors and core memories
  - I/O processors, multiplexed memory access
  - Floating-point arithmetic available
  - Register Transfer Language (RTL) developed
- Software and Applications
  - High-level languages (HLL): FORTRAN, COBOL, ALGOL with compilers and subroutine libraries
  - Still mostly single user at a time, but in batch mode
- Representative systems: CDC 1604, UNIVAC LARC, IBM 7090

Third Generation (1965 to 1974)

- Technology and Architecture
  - Integrated circuits (SSI/MSI)
  - Microprogramming
  - Pipelining, cache memories, lookahead processing
- Software and Applications
  - Multiprogramming and time-sharing operating systems
  - Multi-user applications
- Representative systems: IBM 360/370, CDC 6600, TI ASC, DEC PDP-8

Fourth Generation (1975 to 1990)

- Technology and Architecture
  - LSI/VLSI circuits, semiconductor memory
  - Multiprocessors, vector supercomputers, multicomputers
  - Shared or distributed memory
  - Vector processors
- Software and Applications
  - Multprocessor operating systems, languages, compilers, and parallel software tools
- Representative systems: VAX 9000, Cray X-MP, IBM 3090, BBN TC2000

Fifth Generation (1990 to present)

- Technology and Architecture
  - ULSI/VHSIC processors, memory, and switches
  - High-density packaging
  - Scalable architecture
  - Vector processors
- Software and Applications
  - Massively parallel processing
  - Grand challenge applications
  - Heterogenous processing
- Representative systems: Fujitsu VPP500, Cray MPP, TMC CM-5, Intel Paragon

Elements of Modern Computers

- The hardware, software, and programming elements of modern computer systems can be characterized by looking at a variety of factors, including:
  - Computing problems
  - Algorithms and data structures
  - Hardware resources
  - Operating systems
  - System software support
  - Compiler support

Computing Problems

- Numerical computing
  - complex mathematical formulations
  - tedious integer or floating-point computation
- Transaction processing
  - accurate transactions
  - large database management
  - information retrieval
- Logical Reasoning
  - logic inferences
  - symbolic manipulations

Algorithms and Data Structures

- Traditional algorithms and data structures are designed for sequential machines.
- New, specialized algorithms and data structures are needed to exploit the capabilities of parallel architectures.
- These often require interdisciplinary interactions among theoreticians, experimentalists, and programmers.

Hardware Resources

- The architecture of a system is shaped only partly by the hardware resources.
- The operating system and applications also significantly influence the overall architecture.
- Not only must the processor and memory architectures be considered, but also the architecture of the device interfaces (which often include their advanced processors).

Operating System

- Operating systems manage the allocation and deallocation of resources during user program execution.
- UNIX, Mach, and OSF/1 provide support for
  - multiprocessors and multicomputers
  - multithreaded kernel functions
  - virtual memory management
  - file subsystems
  - network communication services
- An OS plays a significant role in mapping hardware resources to algorithmic and data structures.

System Software Support

- Compilers, assemblers, and loaders are traditional tools for developing programs in high-level languages. With the operating system, these tools determine the bind of resources to applications, and the effectiveness of this determines the efficiency of hardware utilization and the system's programmability.
- Most programmers still employ a sequential mind set, abetted by a lack of popular parallel software support.

System Software Support

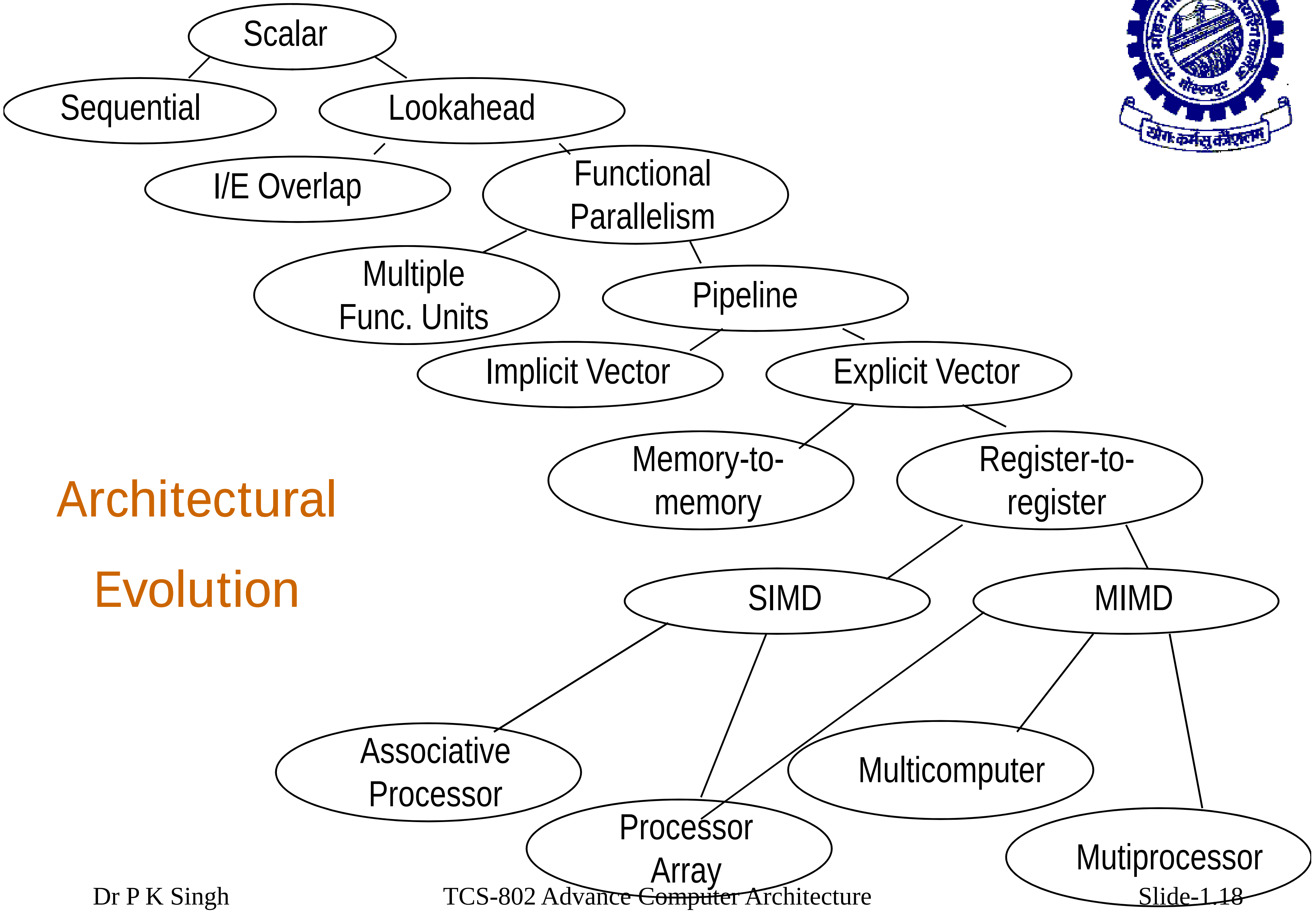
- Parallel software can be developed using entirely new languages designed specifically with parallel support as its goal, or by using extensions to existing sequential languages.
- New languages have obvious advantages (like new constructs specifically for parallelism), but require additional programmer education and system software.
- The most common approach is to extend an existing language.

Compiler Support

- Preprocessors
  - use existing sequential compilers and specialized libraries to implement parallel constructs
- Precompilers
  - perform some program flow analysis, dependence checking, and limited parallel optimzations
- Parallelizing Compilers
  - requires full detection of parallelism in source code, and transformation of sequential code into parallel constructs
- Compiler directives are often inserted into source code to aid compiler parallelizing efforts

Evolution of Computer Architecture

- Architecture has gone through evolutional, rather than revolutional change.
- Sustaining features are those that are proven to improve performance.
- Starting with the von Neumann architecture (strictly sequential), architectures have evolved to include processing lookahead, parallelism, and pipelining.



Architectural Evolution

Flynn’s Classification (1972)

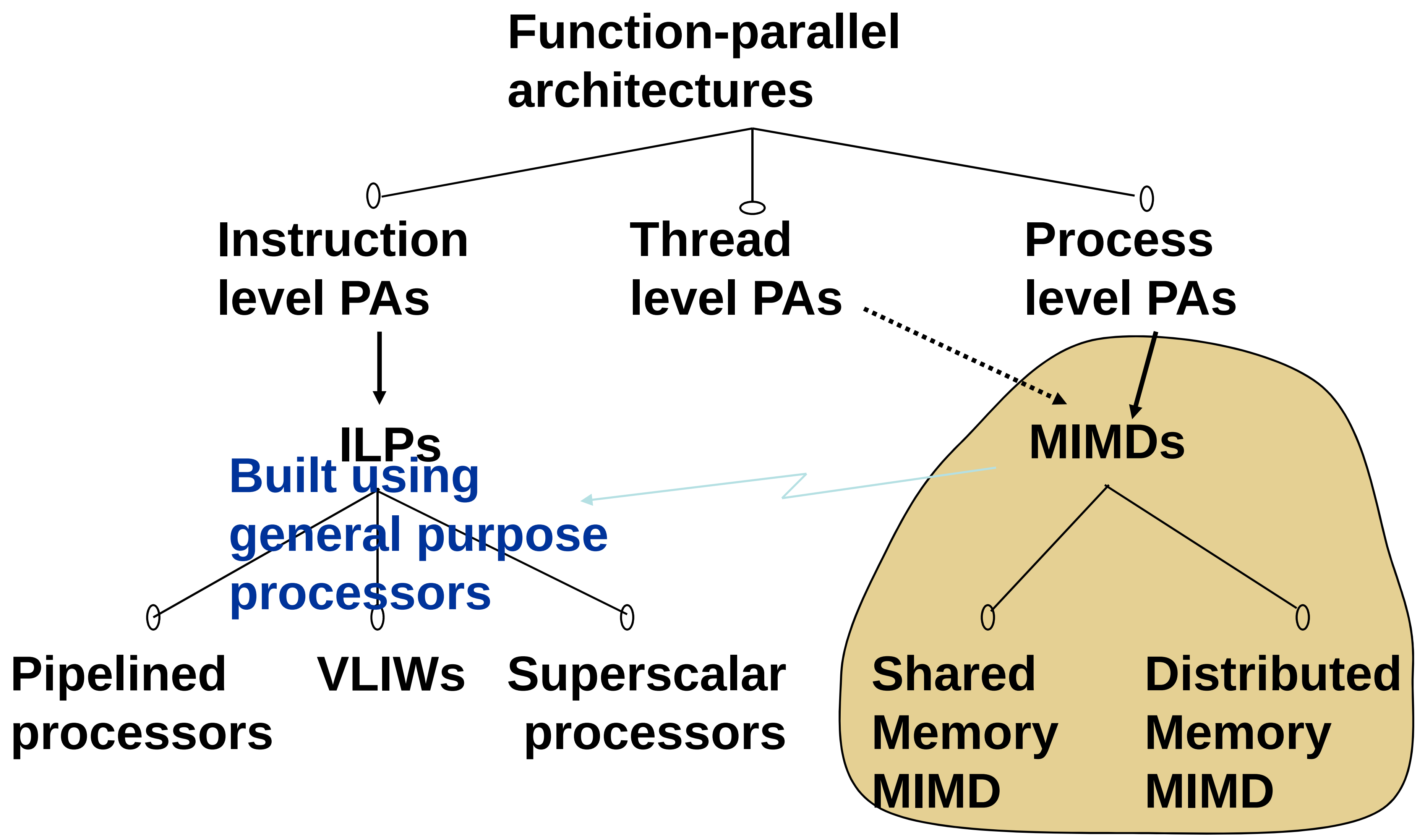
- Single instruction, single data stream (SISD)
  - conventional sequential machines
- Single instruction, multiple data streams (SIMD)
  - vector computers with scalar and vector hardware
- Multiple instructions, multiple data streams (MIMD)
  - parallel computers
- Multiple instructions, single data stream (MISD)
  - systolic arrays
- Among parallel machines, MIMD is most popular, followed by SIMD, and finally MISD.

Parallel Architectures

Sima's Classification

$$\begin{tikzcd}
 & \text{Parallel architectures PAs} \arrow[dl, no head] \arrow[dr, no head] & \\
\text{Data-parallel architectures} & & \text{Function-parallel architectures}
\end{tikzcd}$$

Function Parallel Architectures



Parallel/Vector Computers

- Intrinsic parallel computers execute in MIMD mode.
- Two classes:
  - Shared-memory multiprocessors
  - Message-passing multicomputers
- Processor communication
  - Shared variables in a common memory (multiprocessor)
  - Each node in a multicomputer has a processor and a private local memory, and communicates with other processors through message passing.

Pipelined Vector Processors

- SIMD architecture
- A single instruction is applied to a vector (one-dimensional array) of operands.
- Two families:
  - Memory-to-memory: operands flow from memory to vector pipelines and back to memory
  - Register-to-register: vector registers used to interface between memory and functional pipelines

SIMD Computers

- Provide synchronized vector processing
- Utilize *spatial parallelism* instead of *temporal parallelism*
- Achieved through an array of *processing elements* (PEs)
- Can be implemented using associative memory.

Development Layers (Ni, 1990)

- Hardware configurations differ from machine to machine (even with the same Flynn classification)
- Address spaces of processors vary among different architectures, and depend on memory organization, and should match target application domain.
- The communication model and language environments should ideally be machine-independent, to allow porting to many computers with minimum conversion costs.
- Application developers prefer architectural transparency.

System Attributes to Performance

- Performance depends on
  - hardware technology
  - architectural features
  - efficient resource management
  - algorithm design
  - data structures
  - language efficiency
  - programmer skill
  - compiler technology

Performance Indicators

- Turnaround time depends on:
  - disk and memory accesses
  - input and output
  - compilation time
  - operating system overhead
  - CPU time

- Since I/O and system overhead frequently overlaps processing by other programs, it is fair to consider only the CPU time used by a program, and the user CPU time is the most important factor.

Clock Rate and CPI

- CPU is driven by a clock with a constant *cycle time* τ (usually measured in nanoseconds).

- The inverse of the cycle time is the *clock rate* ($f = 1/\tau$, measured in megahertz).

- The size of a program is determined by its *instruction count*, I_c, the number of machine instructions to be executed by the program.

- Different machine instructions require different numbers of clock cycles to execute. CPI (*cycles per instruction*) is thus an important parameter.

Average CPI

- It is easy to determine the average number of cycles per instruction for a particular processor if we know the frequency of occurrence of each instruction type.
- Of course, any estimate is valid only for a specific set of programs (which defines the instruction mix), and then only if there are sufficiently large number of instructions.
- In general, the term CPI is used with respect to a particular instruction set and a given program mix.

Performance Factors (1)

- The time required to execute a program containing I_c instructions is just T = $I_c \times CPI \times \tau$.

- Each instruction must be fetched from memory, decoded, then operands fetched from memory, the instruction executed, and the results stored.

- The time required to access memory is called the memory cycle time, which is usually *k* times the processor cycle time τ. The value of *k* depends on the memory technology and the processor-memory interconnection scheme.

Performance Factors (2)

- The processor cycles required for each instruction (CPI) can be attributed to
  - cycles needed for instruction decode and execution (p), and
  - cycles needed for memory references ($m \times k$).
- The total time needed to execute a program can then be rewritten as $T = I_c \times (p + m \times k) \times \tau$.

System Attributes

- The five performance factors (I_c, p, m, k, τ) are influenced by four system attributes:
  - instruction-set architecture (affects I_c and p)
  - compiler technology (affects I_c and p and m)
  - CPU implementation and control (affects $p \times \tau$)
  - cache and memory hierarchy (affects memory access latency, $k \times \tau$)
- Total CPU time can be used as a basis in estimating the execution rate of a processor.

MIPS Rate

- If C is the total number of clock cycles needed to execute a given program, then total CPU time can be estimated as $T = C \times \tau = C / f$.

- Other relationships are easily observed:
  - CPI = C / I_c
  - $T = I_c \times CPI \times \tau$
  - $T = I_c \times CPI / f$

- Processor speed is often measured in terms of *millions of instructions per second*, frequently called the MIPS rate of the processor.

MIPS Rate

$$MIPS\ rate = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} = \frac{f \times I_c}{C \times 10}$$

- The MIPS rate is directly proportional to the clock rate and inversely proportion to the CPI.
- All four system attributes (instruction set, compiler, processor, and memory technologies) affect the MIPS rate, which varies also from program to program.

Throughput Rate

- The number of programs a system can execute per unit time, W_s , in programs per second.
- CPU throughput, W_p, is defined as

$$W_p = \frac{f}{I_c \times CPI}$$

- In a multiprogrammed system, the system throughput is often less than the CPU throughput.

Feb 2, 2009

Programming Environments

- Programmability depends on the programming environment provided to the users.
- Conventional computers are used in a sequential programming environment with tools developed for a uniprocessor computer.
- Parallel computers need parallel tools that allow specification or easy detection of parallelism and operating systems that can perform parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

Implicit Parallelism

- Use a conventional language (like C, Fortran, Lisp, or Pascal) to write the program.
- Use a parallelizing compiler to translate the source code into parallel code.
- The compiler must detect parallelism and assign target machine resources.
- Success relies heavily on the quality of the compiler.
- Kuck (U. of Illinois) and Kennedy (Rice U.) used this approach.

Explicit Parallelism

- Programmer write explicit parallel code using parallel dialects of common languages.
- Compiler has reduced need to detect parallelism, but must still preserve existing parallelism and assign target machine resources.
- Seitz (Cal Tech) and Daly (MIT) used this approach.

Needed Software Tools

- Parallel extensions of conventional high-level languages.
- Integrated environments to provide
  - different levels of program abstraction
  - validation, testing and debugging
  - performance prediction and monitoring
  - visualization support to aid program development, performance measurement
  - graphics display and animation of computational results

Categories of Parallel Computers

- Considering their architecture only, there are two main categories of parallel computers:
  - systems with shared common memories, and
  - systems with unshared distributed memories.

Shared-Memory Multiprocessors

- Shared-memory multiprocessor models:
  - Uniform-memory-access (UMA)
  - Nonuniform-memory-access (NUMA)
  - Cache-only memory architecture (COMA)
- These systems differ in how the memory and peripheral resources are shared or distributed.

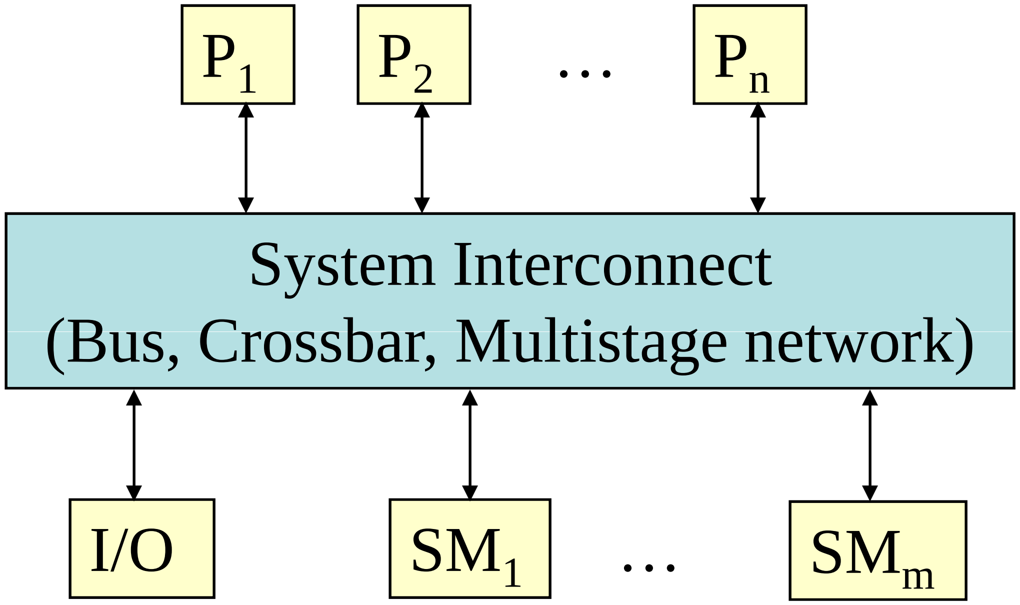
The UMA Model - 1

- Physical memory uniformly shared by all processors, with equal access time to all words.
- Processors may have local cache memories.
- Peripherals also shared in some fashion.
- *Tightly coupled systems* use a common bus, crossbar, or multistage network to connect processors, peripherals, and memories.
- Many manufacturers have multiprocessor (MP) extensions of uniprocessor (UP) product lines.

The UMA Model - 2

- Synchronization and communication among processors achieved through shared variables in common memory.
- Symmetric MP systems – all processors have access to all peripherals, and any processor can run the OS and I/O device drivers.
- Asymmetric MP systems – not all peripherals accessible by all processors; kernel runs only on selected processors (master); others are called *attached processors* (AP).

The UMA Multiprocessor Model



Example: Performance Calculation

- Consider two loops. The first loop adds corresponding elements of two *N*-element vectors to yield a third vector. The second loop sums elements of the third vector. Assume each add/assign operation takes 1 cycle, and ignore time spent on other actions (e.g. loop counter incrementing/testing, instruction fetch, etc.). Assume interprocessor communication requires *k* cycles.

- On a sequential system, each loop will require N cycles, for a total of 2*N* cycles of processor time.

Example: Performance Calculation

- On an M-processor system, we can partition each loop into M parts, each having $L = N / M$ add/assigns requiring L cycles. The total time required is thus $2L$. This leaves us with M partial sums that must be totaled.
- Computing the final sum from the M partial sums requires $l = \log_2(M)$ additions, each requiring k cycles (to access a non-local term) and 1 cycle (for the add/assign), for a total of $l \times (k+1)$ cycles.
- The parallel computation thus requires

$$2N / M + (k + 1) \log_2(M) \text{ cycles.}$$

Example: Performance Calculation

- Assume N = 2^{20}.
- Sequential execution requires 2N = 2^{21} cycles.
- If processor synchronization requires k = 200 cycles, and we have M = 256 processors, parallel execution requires

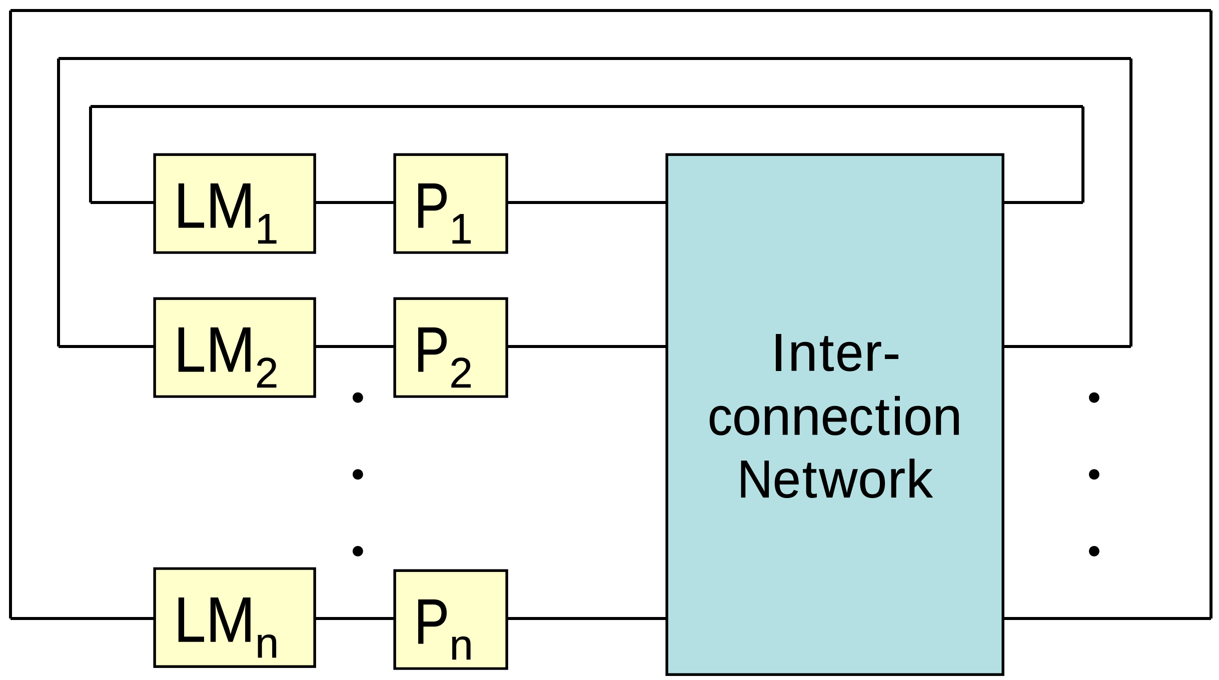
$$
\begin{aligned}
& 2N / M + (k + 1) \log_2(M) \\
= \quad & 2^{21} / 2^8 + 201 \times 8 \\
= \quad & 2^{13} + 1608 \;=\; 9800 \text{ cycles}
\end{aligned}
$$

- Comparing results, the parallel solution is 214 times faster than the sequential, with the best theoretical speedup being 256 (since there are 256 processors). Thus the efficiency of the parallel solution is 214 / 256 = 83.6 %.

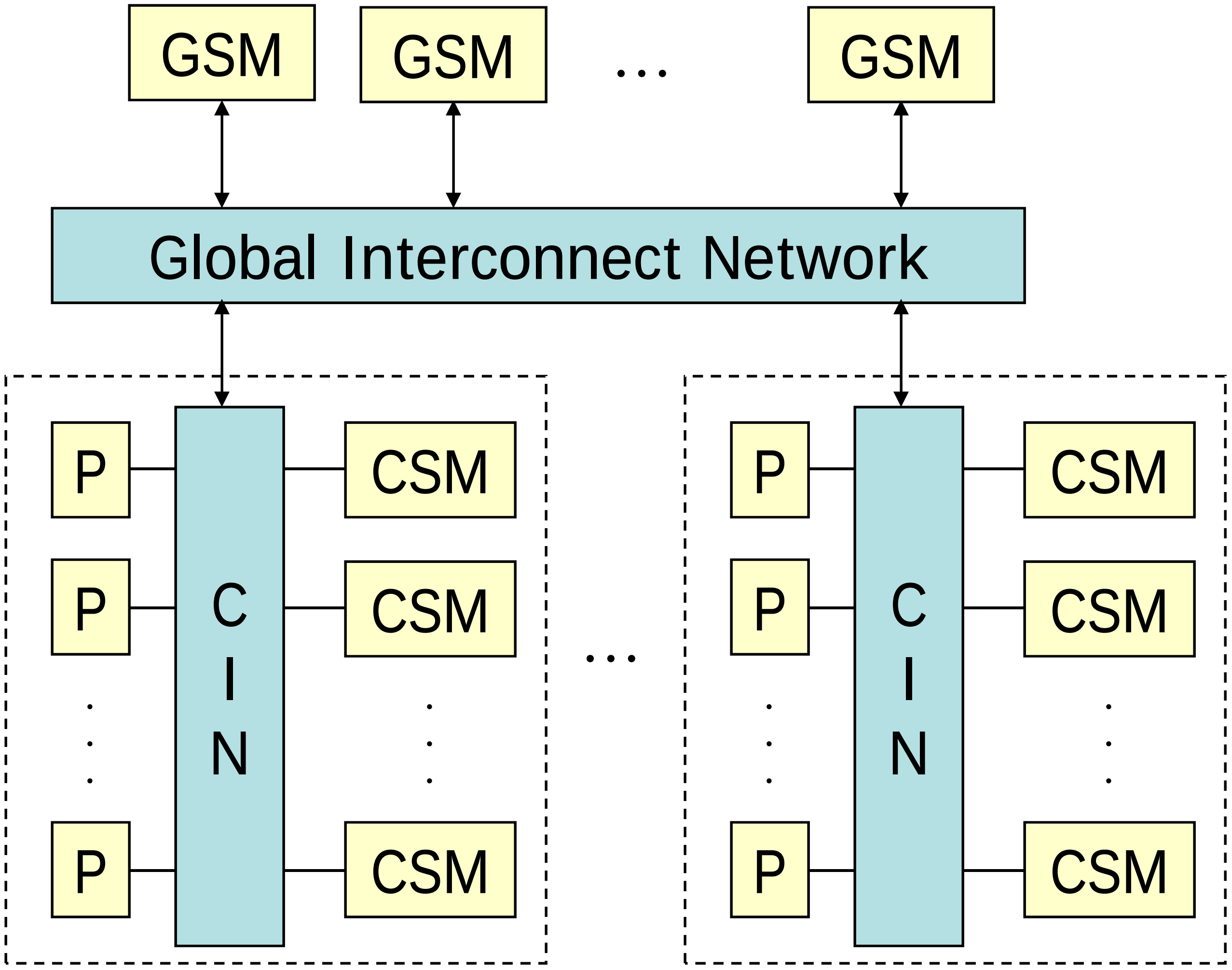
The NUMA Model - 1

- Shared memories, but access time depends on the location of the data item.
- The shared memory is distributed among the processors as local memories, but each of these is still accessible by all processors (with varying access times).
- Memory access is fastest from the locally-connected processor, with the interconnection network adding delays for other processor accesses.
- Additionally, there may be global memory in a multiprocessor system, with two separate interconnection networks, one for clusters of processors and their cluster memories, and another for the global shared memories.

Shared Local Memories



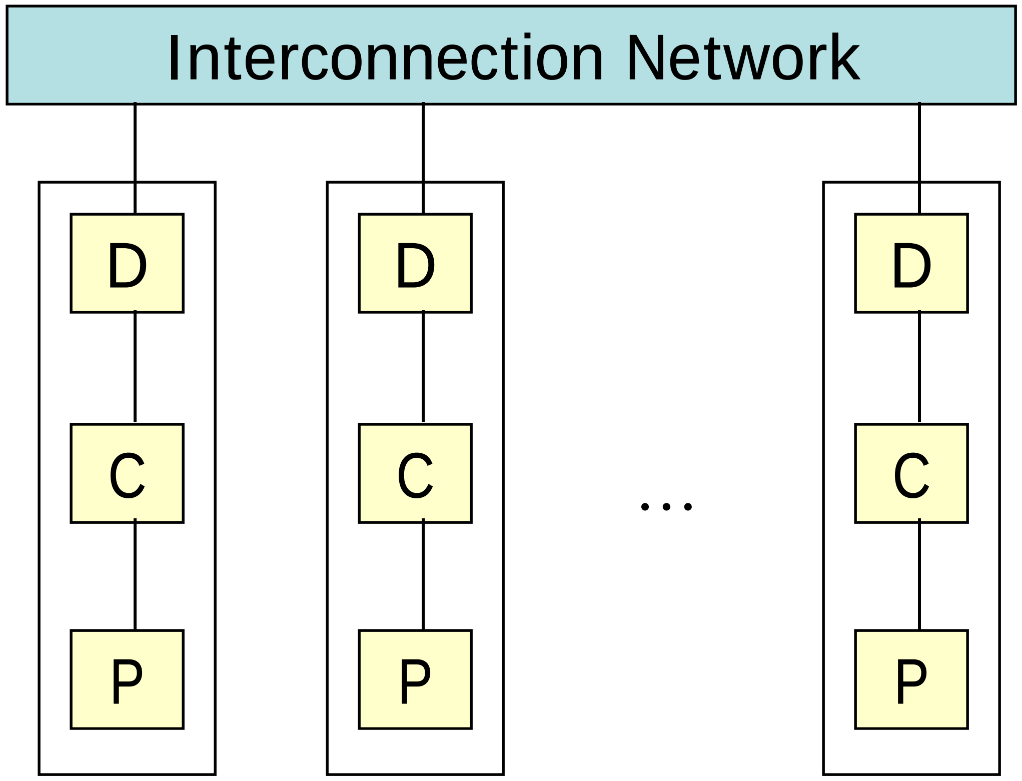
Hierarchical Cluster Model



The COMA Model

- In the COMA model, processors only have cache memories; the caches, taken together, form a global address space.
- Each cache has an associated directory that aids remote machines in their lookups; hierarchical directories may exist in machines based on this model.
- Initial data placement is not critical, as cache blocks will eventually migrate to where they are needed.

Cache-Only Memory Architecture



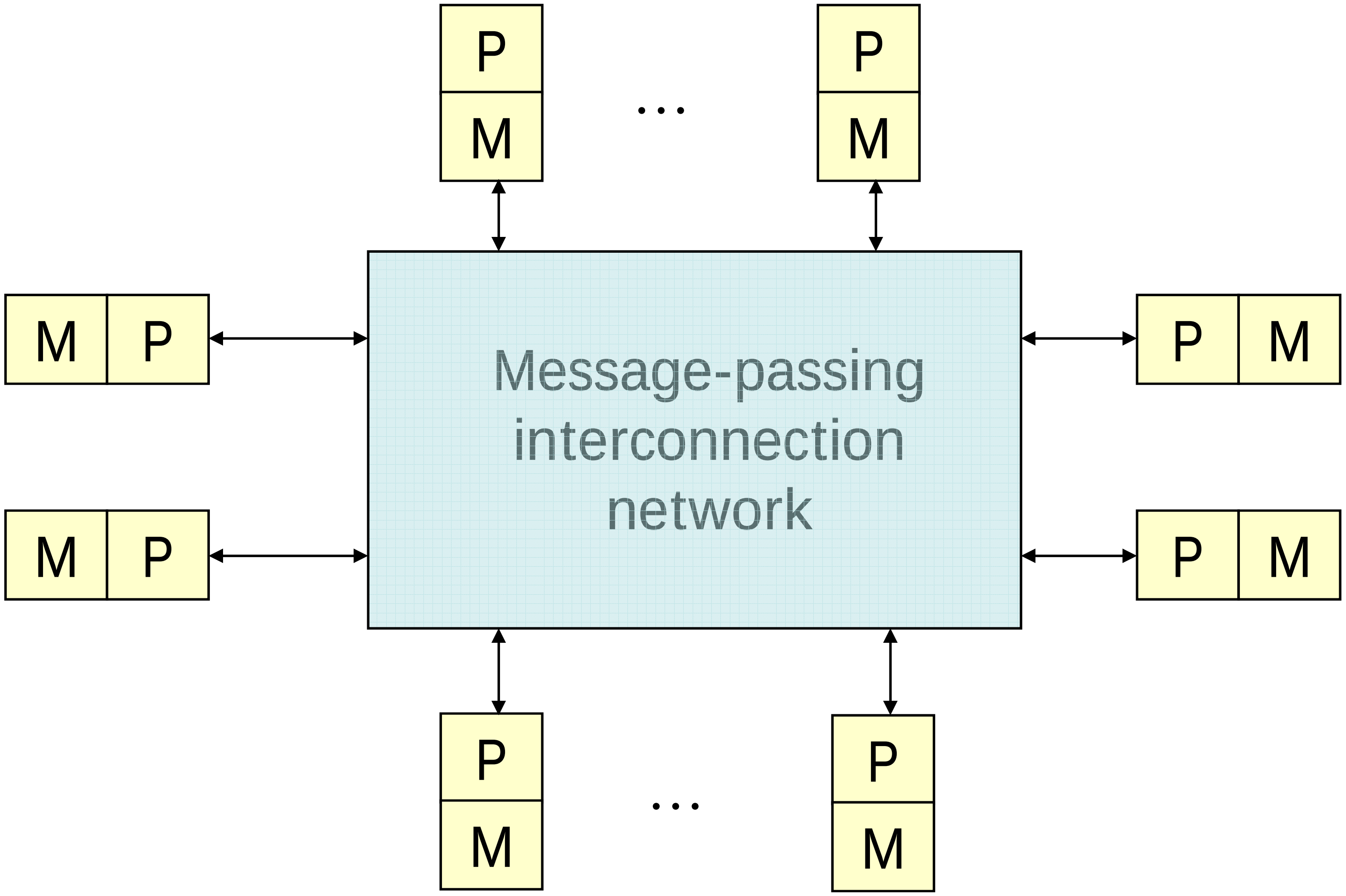
Other Models

- There can be other models used for multiprocessor systems, based on a combination of the models just presented. For example:
  - cache-coherent non-uniform memory access (each processor has a cache directory, and the system has a distributed shared memory)
  - cache-coherent cache-only model (processors have caches, no shared memory, caches must be kept coherent).

Multicomputer Models

- Multicomputers consist of multiple computers, or *nodes*, interconnected by a message-passing network.
- Each node is autonomous, with its own processor and local memory, and sometimes local peripherals.
- The message-passing network provides point-to-point static connections among the nodes.
- Local memories are not shared, so traditional multicomputers are sometimes called *no-remote-memory-access* (or NORMA) machines.
- Inter-node communication is achieved by passing messages through the static connection network.

Generic Message-Passing Multicomputer



Multicomputer Generations

- Each multicomputer uses routers and channels in its interconnection network, and heterogeneous systems may involved mixed node types and uniform data representation and communication protocols.
- First generation: hypercube architecture, software-controlled message switching, processor boards.
- Second generation: mesh-connected architecture, hardware message switching, software for medium-grain distributed computing.
- Third generation: fine-grained distributed computing, with each VLSI chip containing the processor and communication resources.

Multivector and SIMD Computers

- Vector computers often built as a scalar processor with an attached optional vector processor.
- All data and instructions are stored in the central memory, all instructions decoded by scalar control unit, and all scalar instructions handled by scalar processor.
- When a vector instruction is decoded, it is sent to the vector processor’s control unit which supervises the flow of data and execution of the instruction.

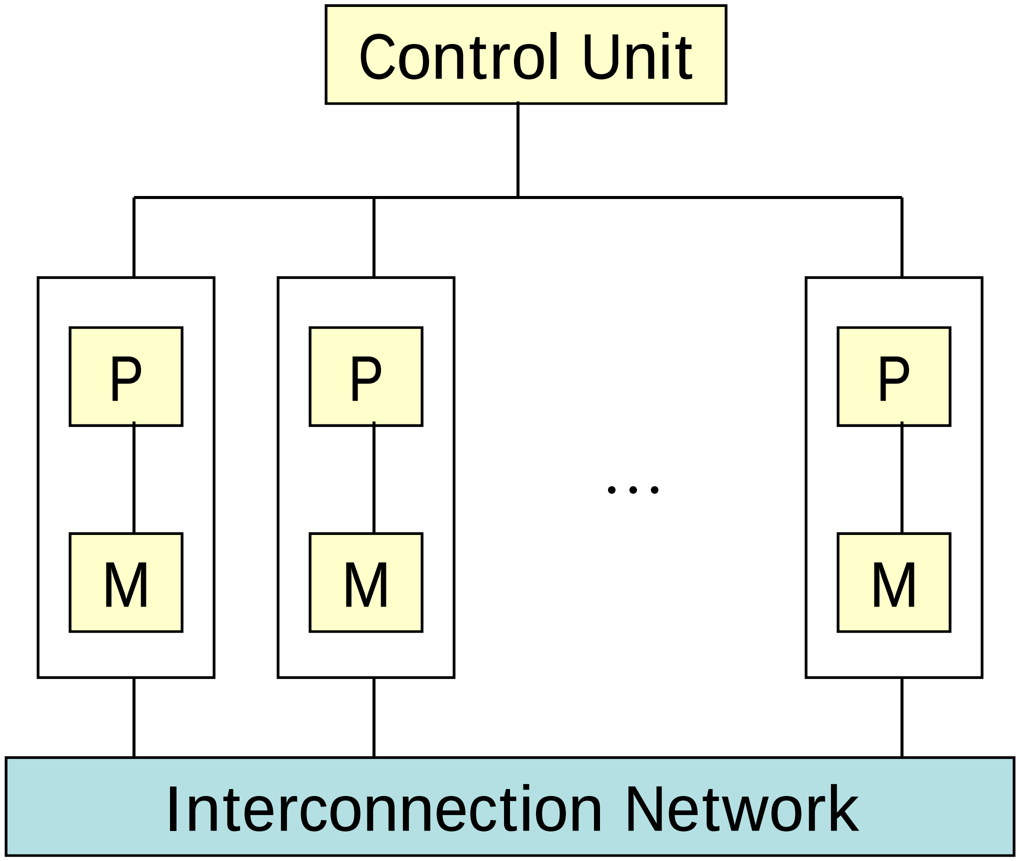
Vector Processor Models

- In register-to-register models, a fixed number of possibly reconfigurable registers are used to hold all vector operands, intermediate, and final vector results. All registers are accessible in user instructions.
- In a memory-to-memory vector processor, primary memory holds operands and results; a vector stream unit accesses memory for fetches and stores in units of large superwords (e.g. 512 bits).

SIMD Supercomputers

- Operational model is a 5-tuple (*N*, *C*, *I*, *M*, *R*).
  - *N* = number of processing elements (PEs).
  - *C* = set of instructions (including scalar and flow control)
  - *I* = set of instructions broadcast to all PEs for parallel execution.
  - *M* = set of masking schemes used to partion PEs into enabled/disabled states.
  - *R* = set of data-routing functions to enable inter-PE communication through the interconnection network.

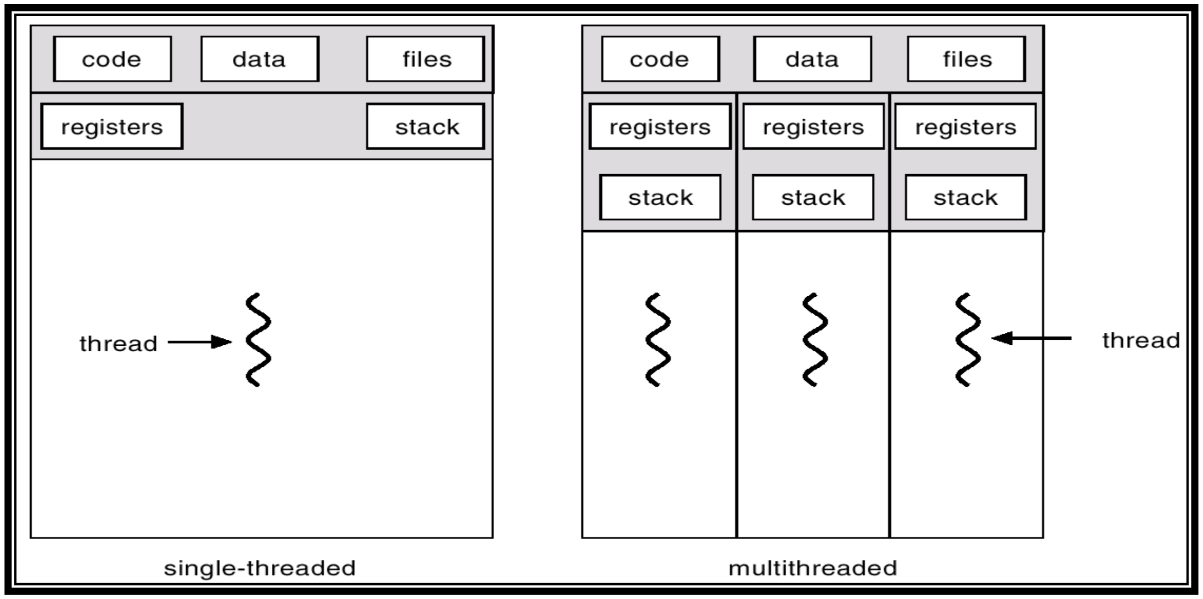
Operational Model of SIMD Computer



Threads: Overview

- A thread sometimes is called a lightweight process (LWP)
- LWP comprises of: thread ID, program counter, register set, and stack
- LWP shares with other threads belonging to the same process, process code section, data section, some OS resources such as open files and signals
- A traditional heavyweight process has a single thread of control
- If the process has multiple threads of control, it can do more that one a single task at a time
- Processes can accomplish several tasks by running more than a single thread

Threads -- Overview



- A simple way to think about a process is as an address space (containing code, data, etc.) in which there is a single thread of execution
- The thread is the active part of a process

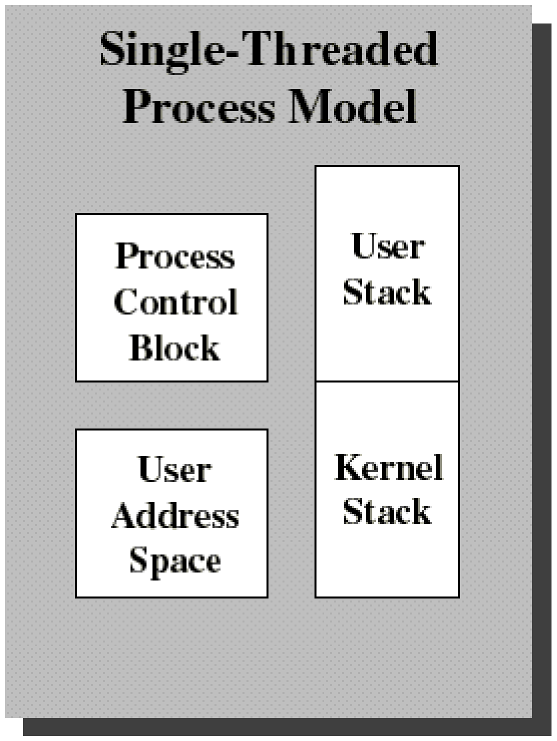
Threads -- Motivation

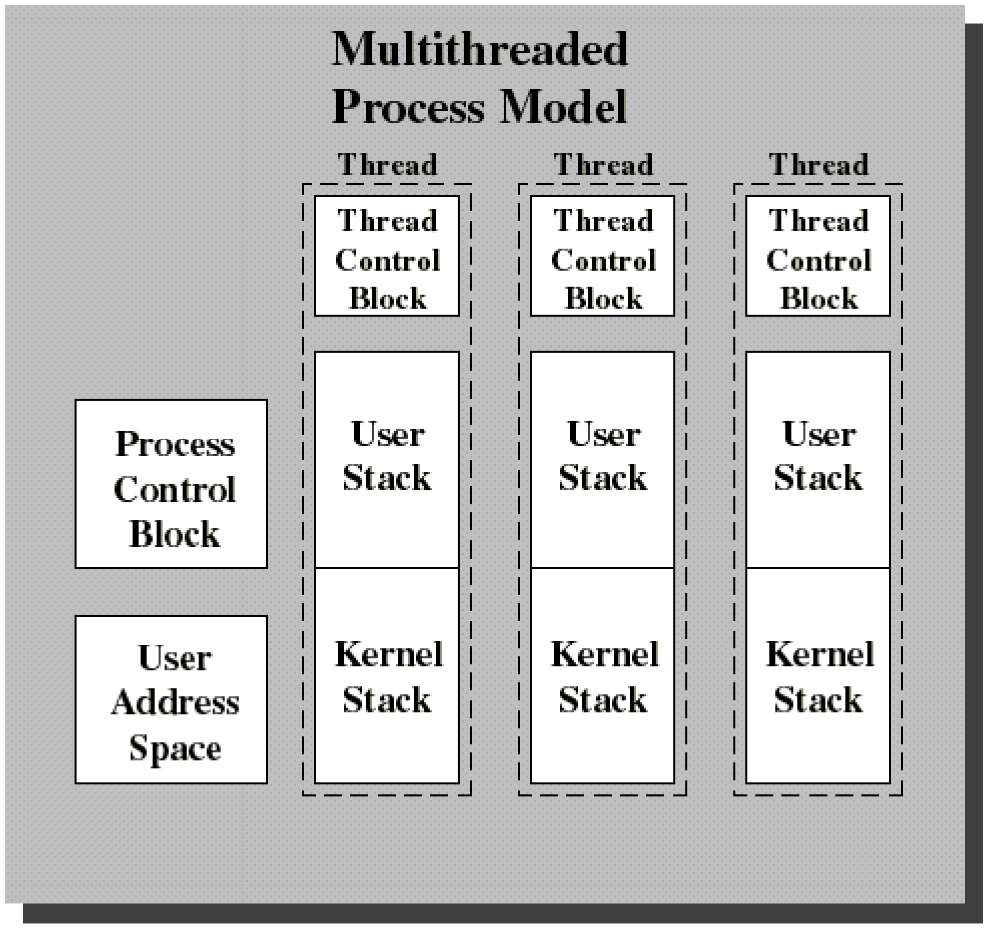
- Processes can do several things concurrently be running more than a single thread
- Each thread is a different stream of control that can execute its instructions independently
- A program (e.g. Browser) may consist of the following thread:
  - GUI thread
  - I/O thread
  - computation

Threads -- Benefits

- Responsiveness: multithreading is useful in an interactive application. For example, a multithreaded web browser can allow user interaction even though an image is still downloading
- Resource sharing: threads share memory and resources allocated to the process to which they belong
  - For example: code sharing, where different threads of activity all within the same address space

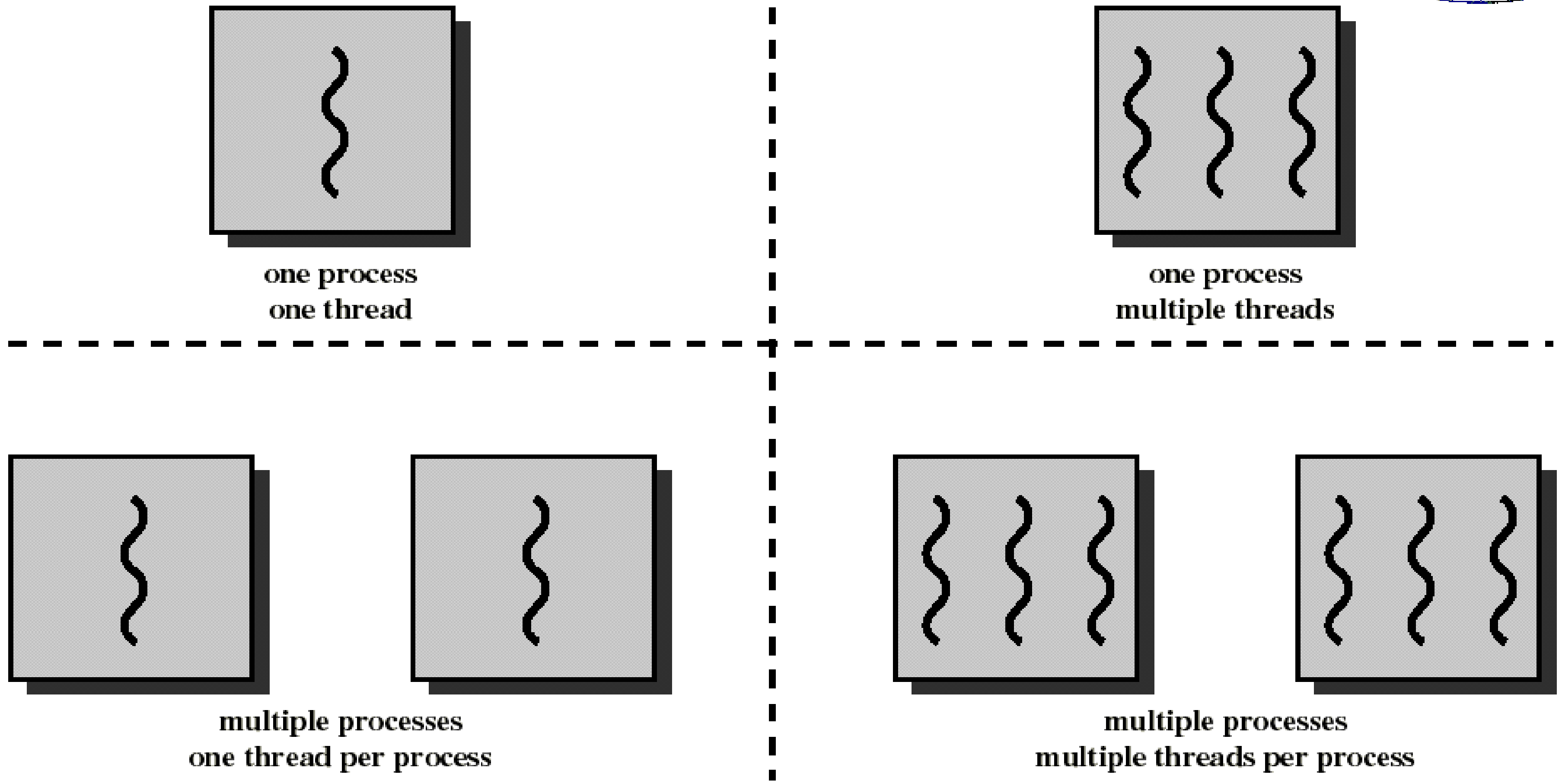
Single Threaded and Multithreaded Process Models





- **Thread Control Block contains a register image, thread priority and thread state information.**

Threads Versus Processes



User and Kernel Threads

- Support of thread can be either supported by user level of kernel level
- User threads:
  - Supported above the kernel
  - Implemented by the thread library at the user level
  - User thread are generally fast to create and manage
- User threads have drawbacks
  - If the kernel is single-threaded, then any user thread performing blocking will cause the entire process to block (even though there are other user threads available to run within the application)

User and Kernel Threads

- Example of user thread libraries: *Pthreads* and *Mach C-threads*
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though this library may be part of another library, such as libc
- Mach operating system was started in 1985
- Mach goals include
  - is to provide interprocess communication functionality at the kernel level
  - Kernel level support for light-weight threads

User and Kernel Threads

- User threads
  - The kernel is not aware of the existence of threads
  - All thread management is done by the application by using a thread library
  - Thread switching does not require kernel mode privileges (no mode switch)
  - Scheduling is application specific
- Threads library contains code for:
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts

User and Kernel Threads

- Kernel threads are supported directly by the operating system
- The kernel performs creation, scheduling, and management
- Kernel threads are generally slower to create and manage that user threads
- However since the kernel manages the threads, if a thread performs a system call, the kernel can schedule another thread
- Modern OSs such as Windows NT, Windows 2000 and Solaris 2 support kernel threads