Architecture of Digital Systems II

5 Multiprocessors

Multiprocessing Systems

A multiprocessing system is a computer system that contains more than one instruction set processor.

A general distinction is made between symmetric and asymmetric multiprocessing.

Symmetric multiprocessing (SMP):

"All processors are equal"; communication is usually carried out via shared memory.

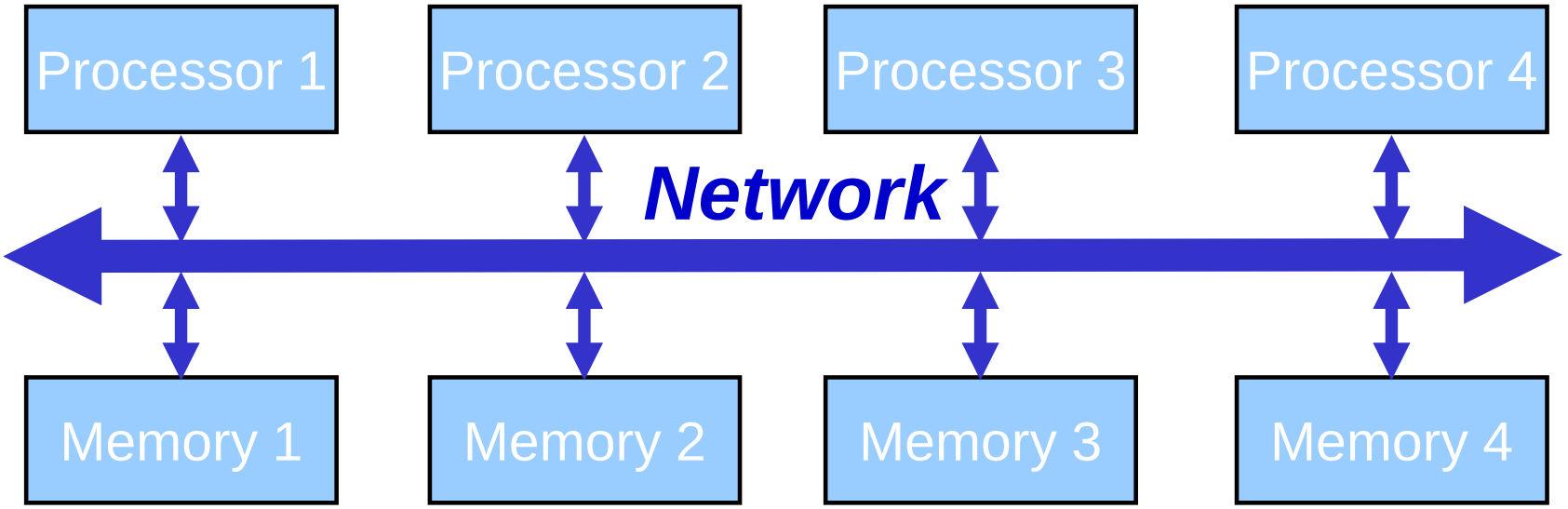
Asymmetric multiprocessing (AMP):

Separate specialized processors are used for specific tasks; memory is local to each processor; communication is usually carried out through message-passing.

Symmetric Multiprocessing

In symmetric multiprocessing (SMP), a pool of processors and a pool of memory are connected by a communication network. In general, both structures are highly regular, allowing for a regular programming model. This approach delivers very high computation performance for *homogeneous* problems that can be easily split into many identical tasks.

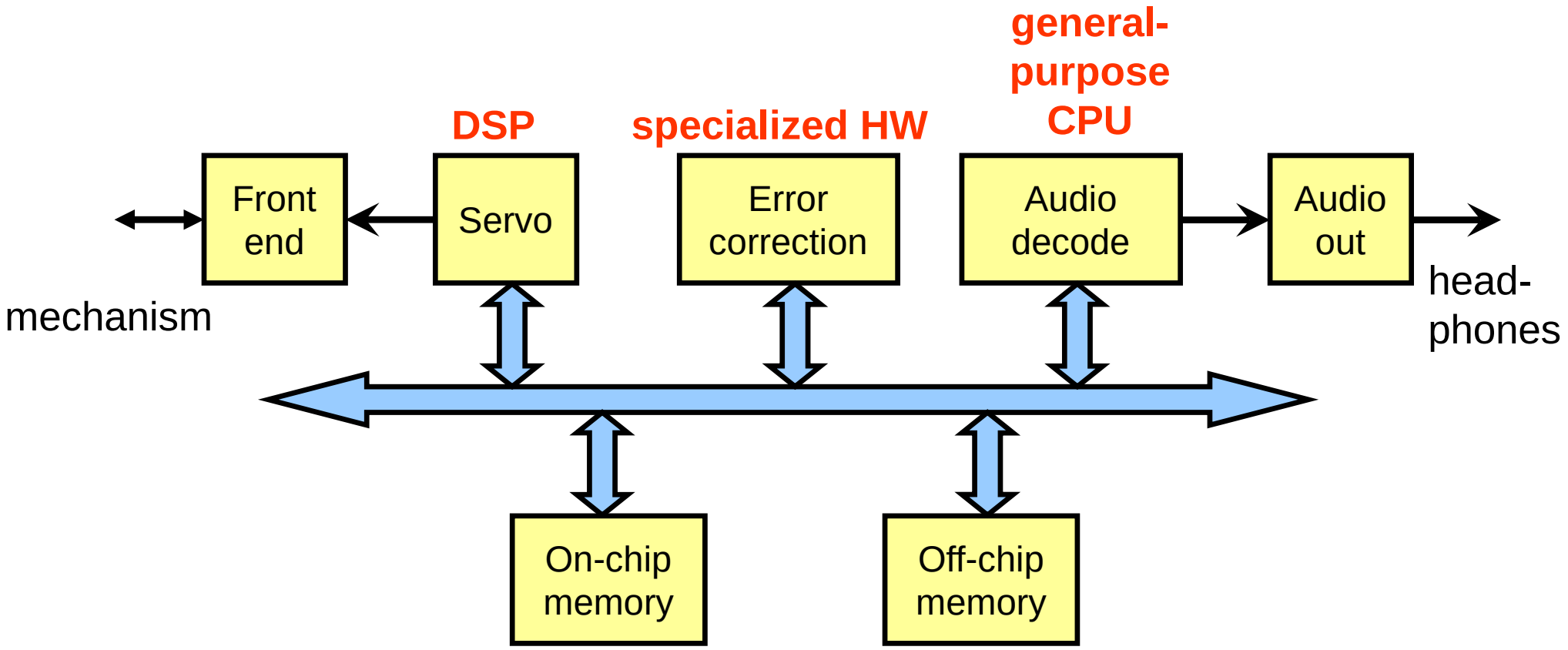
Examples: complex numerical simulations: climate, weather; biological (genetics), chemical and pharmaceutical applications.



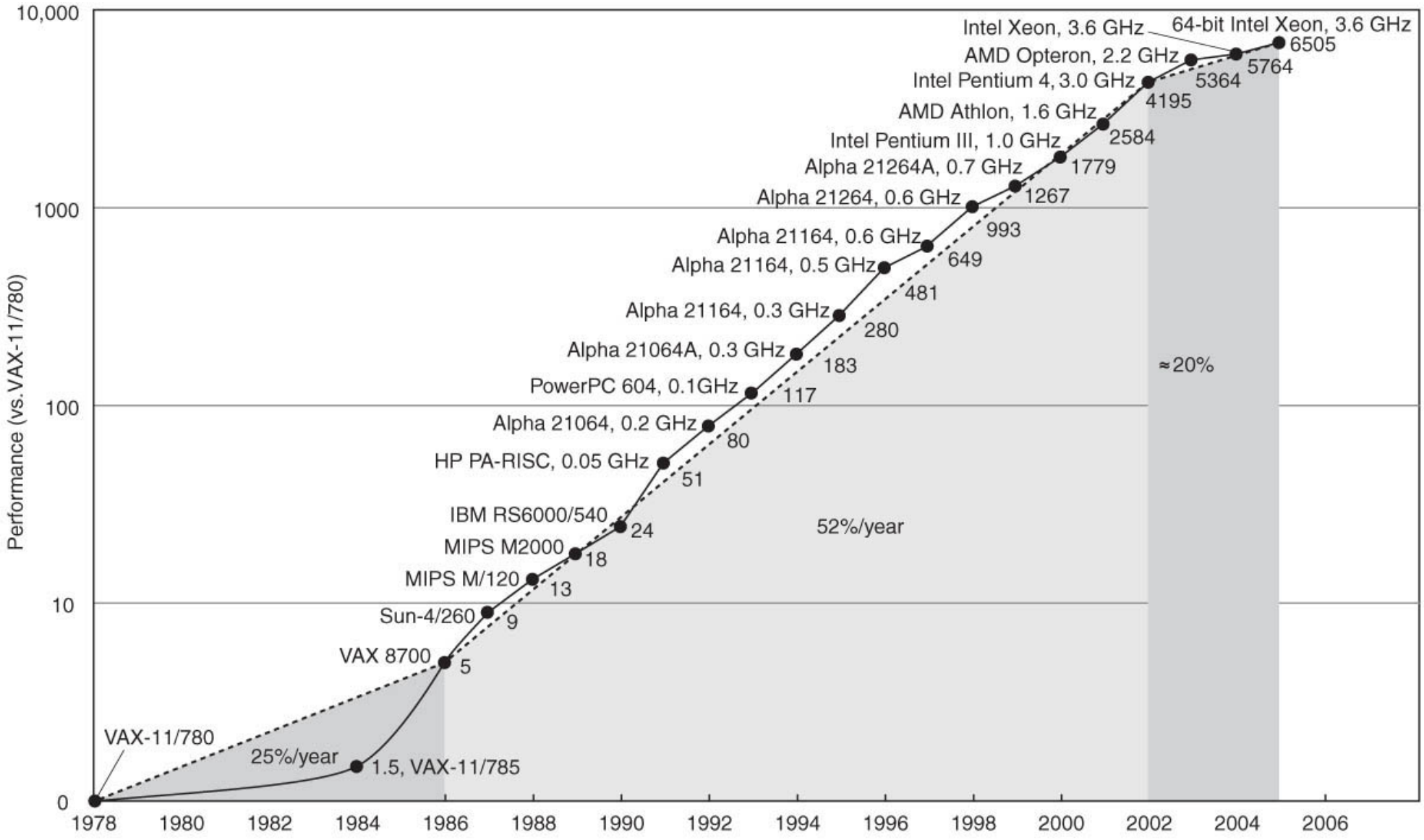
Asymmetric Multiprocessing

Asymmetric multiprocessing (AMP) targets problems that are split into *heterogeneous* sub-tasks. The individual tasks are solved by specialized hardware components (CPUs or custom modules). Many embedded systems are AMP systems.

Example: System-on-Chip for CD/MP3 player



Growth in Processor Performance



Source: [2] Patterson/Hennessy. Copyright © 2009 Elsevier, Inc.

Multi-core Processors

(Caption from figure on previous slide)

FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

There is a limit on the clock frequency of a processor due to heat dissipation, i.e., the power that can be taken away from the chip by cooling. How can we increase the computational power of a computer without raising the clock frequency? Use more than one processor!

High-end processors today have several computing “cores”. We speak of multi-core processors versus traditional uni-processors.

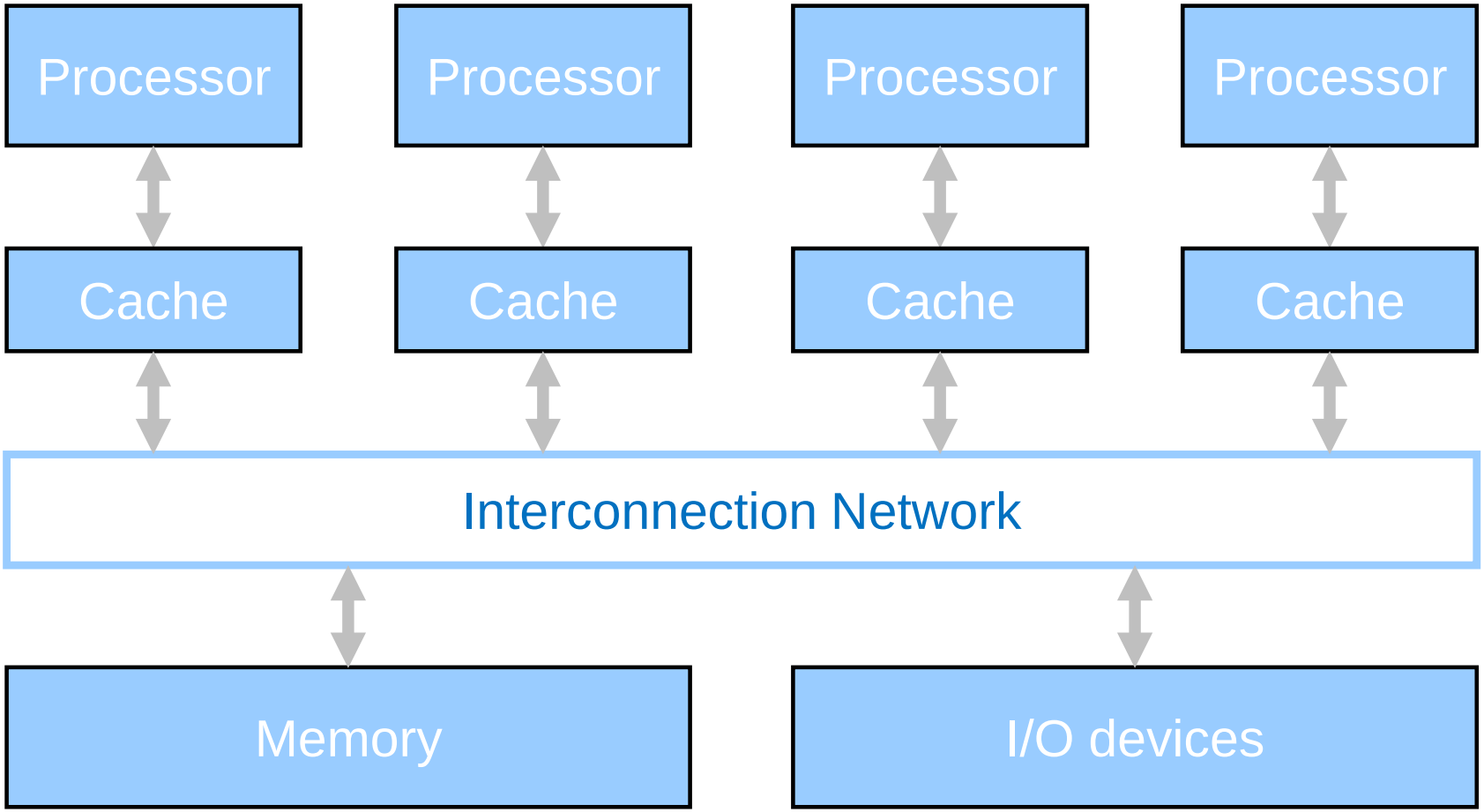
Parallel Programming

Having more than one processor for solving a task requires non-trivial programming efforts. Traditionally, a sequential software program would simply run faster on the next generation of a processor. With multiple cores, the software must be designed carefully to work as a set of coordinated, concurrent program components. Challenges and requirements are:

- Load balancing: All available cores should be loaded with roughly the same amount of work.
- Scheduling: The sub-tasks must be chosen such that the processors can do their work mostly *concurrently*.
- Synchronization: To implement the schedule processors must synchronize.
- Communication: To dispatch sub-tasks and to share results, software components must communicate.

Parallel implementation requires overhead that needs to be kept low.

Shared Memory Multiprocessors



A shared-memory multiprocessor is a multiprocessor where all processors share a single common physical address space. Communication happens through shared variables in memory.

(Shared-Memory Multiprocessors)

In shared-memory multiprocessors, the processors are tightly coupled and coordinated by a single operating system.

Shared-memory multiprocessors exploit Thread-Level Parallelism (TLP) which comes in two flavors:

- Parallel processing – The individual threads are coupled tightly to collaborate on a common task
- Request-level parallelism – The threads are, at the most, loosely coupled or they run independently of each other on the multiprocessor. Examples: the queries/transactions in a database system or independent processes spawned by a user on a desktop computer.

Example: Shared-Memory Parallel Program (1)

Task:

Sum 100,000 numbers on an shared-memory UMA multiprocessor.
We have N=100 processors.
The numbers are in a shared array `A[0..99999]`.

Load balancing:

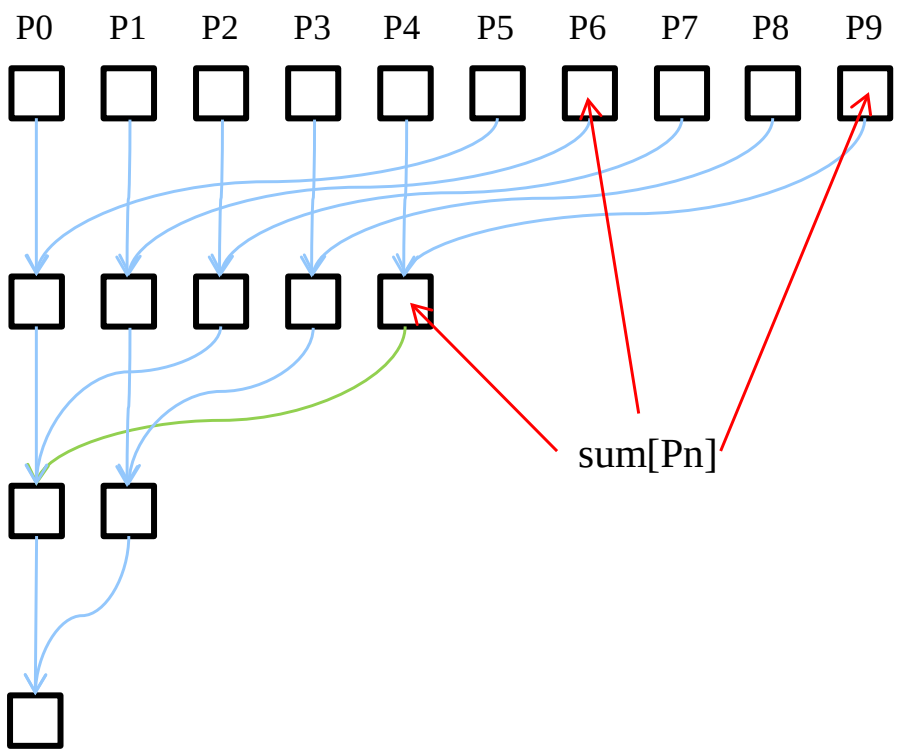
Split the set of numbers equally.
Let `Pn` be the processor index, `Pn = 0..99`.
Each processor first computes:

```
int i;      /* local */
sum[Pn]=0;  /* shared */
for (i = 1000*Pn; i < 1000*(Pn+1); i++) {
        sum[Pn] += A[i];
}
```

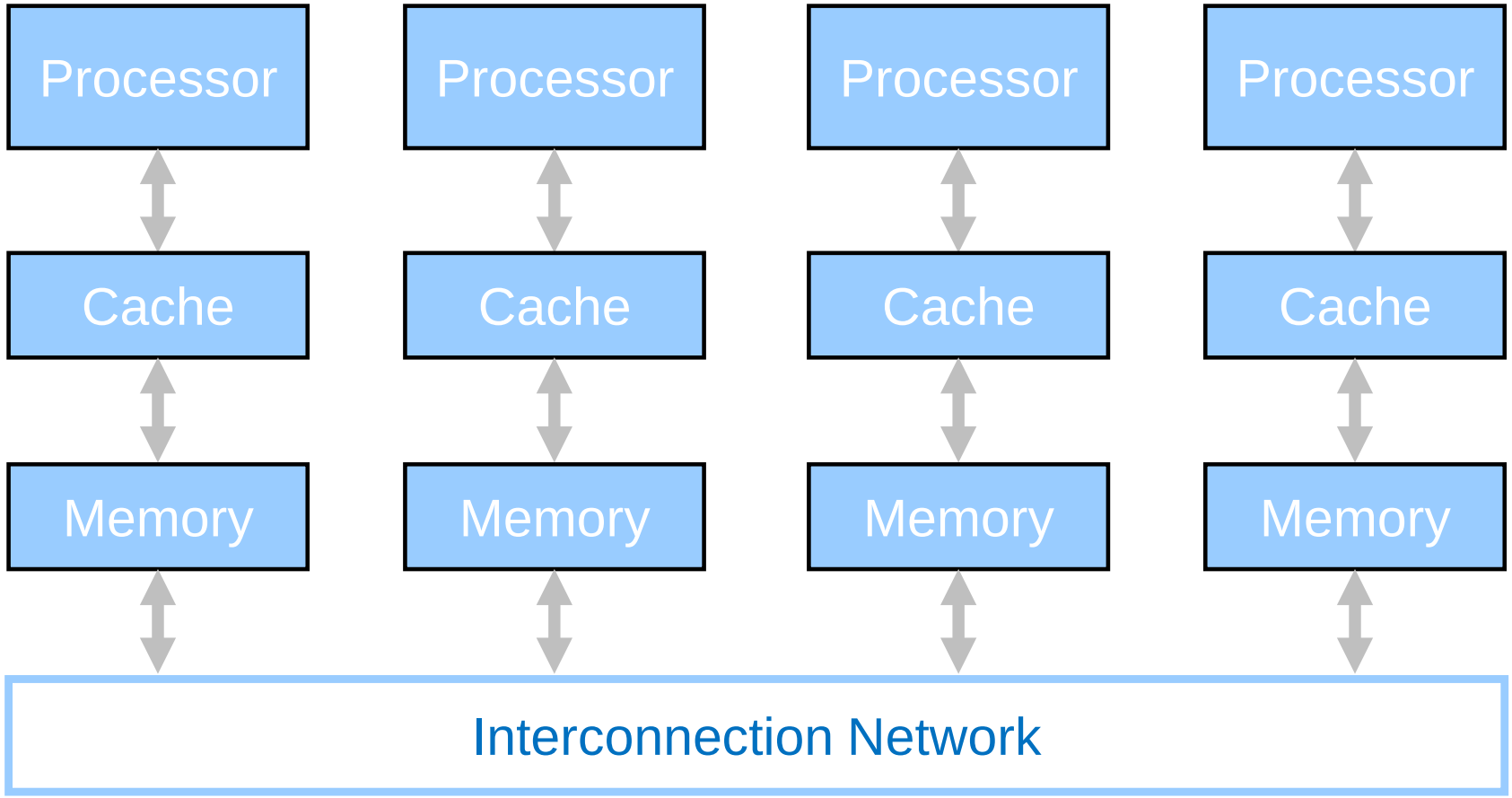
Example: Shared-Memory Parallel Program (2)

Parallel execution: On each processor, execute:

```
int half=100; /* local */
while (half > 0) {
   bool odd; / *local */
   synch();
   odd = (half % 2 != 0);
   half = half/2;
   if (Pn < half) {
      sum[Pn] += sum[Pn+half];
      if (Pn == 0 && odd) {
         sum[0] += sum[half+half];
         /* leftover partial sum */
      }
   } /* else processor is not needed
*/
}
```



Message-Passing Multiprocessors



In a message-passing multiprocessor every processor has its own private memory and communicates with other processors by message passing.

Clusters

A cluster is a prominent example of a message-passing parallel computer. Clusters are usually built from standard computing components (e.g., PCs, workstations) and standard communication components (LANs, standard network switches). Every computer in the cluster runs its own copy of an operating system.
Typical applications of clusters are internet services (e.g., web search, email servers, file servers).

Drawbacks of clusters when compared to shared-memory MPs:

- Higher administration costs (*n* machines in the cluster)
- Lower communication bandwidth (machines communicate over I/O rather than over the memory interconnect which is usually faster)
- Memory is strongly divided between the machines
- Larger overhead in total memory consumption because a copy of the OS resides locally in each node's memory

Example: Message-Passing Parallel Program (1)

Task: Sum 100,000 numbers on a message passing multiprocessor.
We have N=100 processors. Initially, processor P0 has all the data.

Load balancing:
Split the set of numbers equally.
Let `Pn` be the processor index, `Pn = 0..99`.
First step: distribute the 100 subsets of 1000 numbers to each of the processor using SEND/RECV.

Note:
Only P0 uses the full range of A.
All other processors use only A[0..999].

```
int A[100000];
int i, p;
int sum = 0;

if (Pn == 0) {
   for (p = 1; p < 100; p++) {
      for (i = 0; i < 1000; i++) {
         send(p, A[p*1000+i];
      }
   }
} else {
   for (i = 0; i < 1000; i++) {
      recv(0, &A[i];
   }
}
```

Example: Message-Passing Parallel Program (2)

Parallel execution:

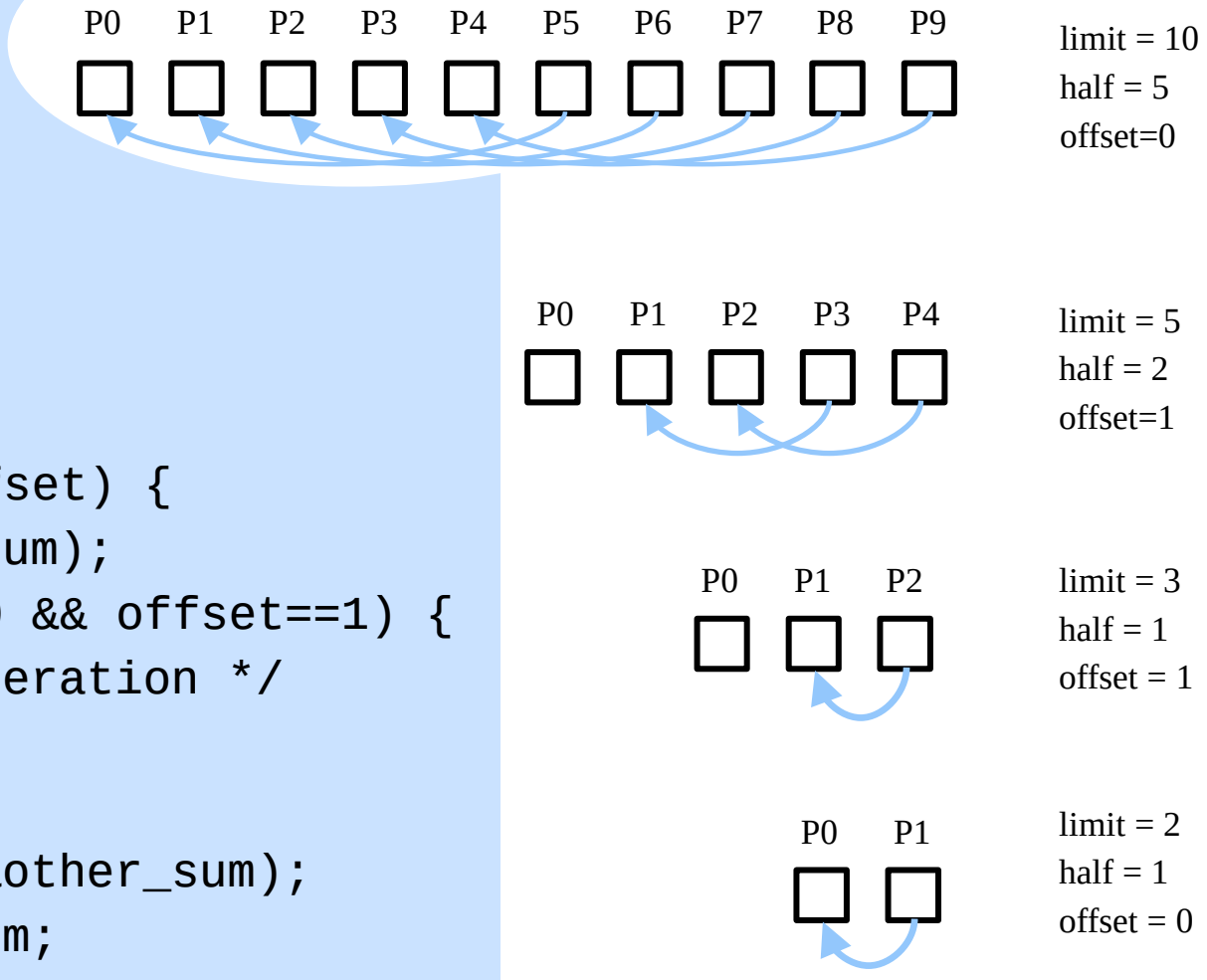
On each processor, first sum up the local numbers to compute a partial sum.

```
for (i = 0; i < 1000; i++) {
   sum += A[i];
}
```

Then, the partial sums need to be added up. (See next page.) Similarly as in the shared-memory program we apply an iterative "Divide and conquer" approach. In each iteration, half the processors send their results to the other half. If the number of processors in an iteration is odd then processor 0 is left out in that iteration.

Example: Message-Passing Parallel Program (3)

```
int limit=100;
while (limit > 1) {
   int offset = 0;
   int half = limit / 2;
   if (limit%2 != 0) {
      offset = 1;
   }
   if (Pn < limit) {
      if (Pn >= half+offset) {
         send(Pn-half, sum);
      } else if (Pn == 0 && offset==1) {
         /* skip this iteration */
      } else {
         int other_sum;
         recv(Pn+half, &other_sum);
         sum += other_sum;
      }
   } /* Pn not needed anymore */
   limit = half + offset;
}
```



Shared-Memory MP: UMA vs. NUMA

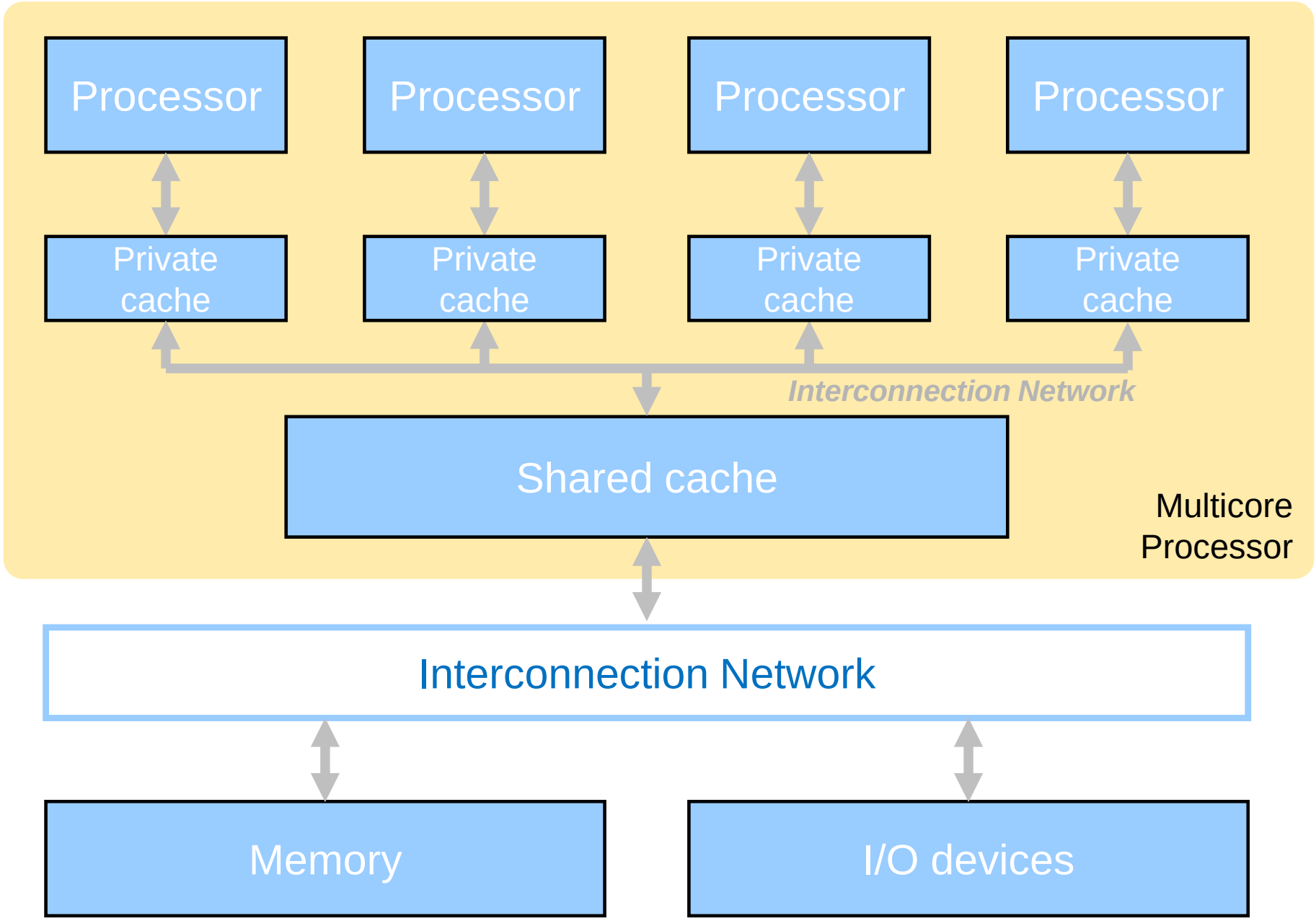
Shared-memory MP architectures differ in the memory access times that the processors see.

In uniform memory access (UMA) multiprocessors the access times for main memory are roughly the same for all processors.

In non-uniform memory access (NUMA) multiprocessors the access times may differ vastly, depending on which processor is asking for which address. A processor may have a much faster access time to some specific part of the memory, e.g., memory physically close to the processor, than to the rest.

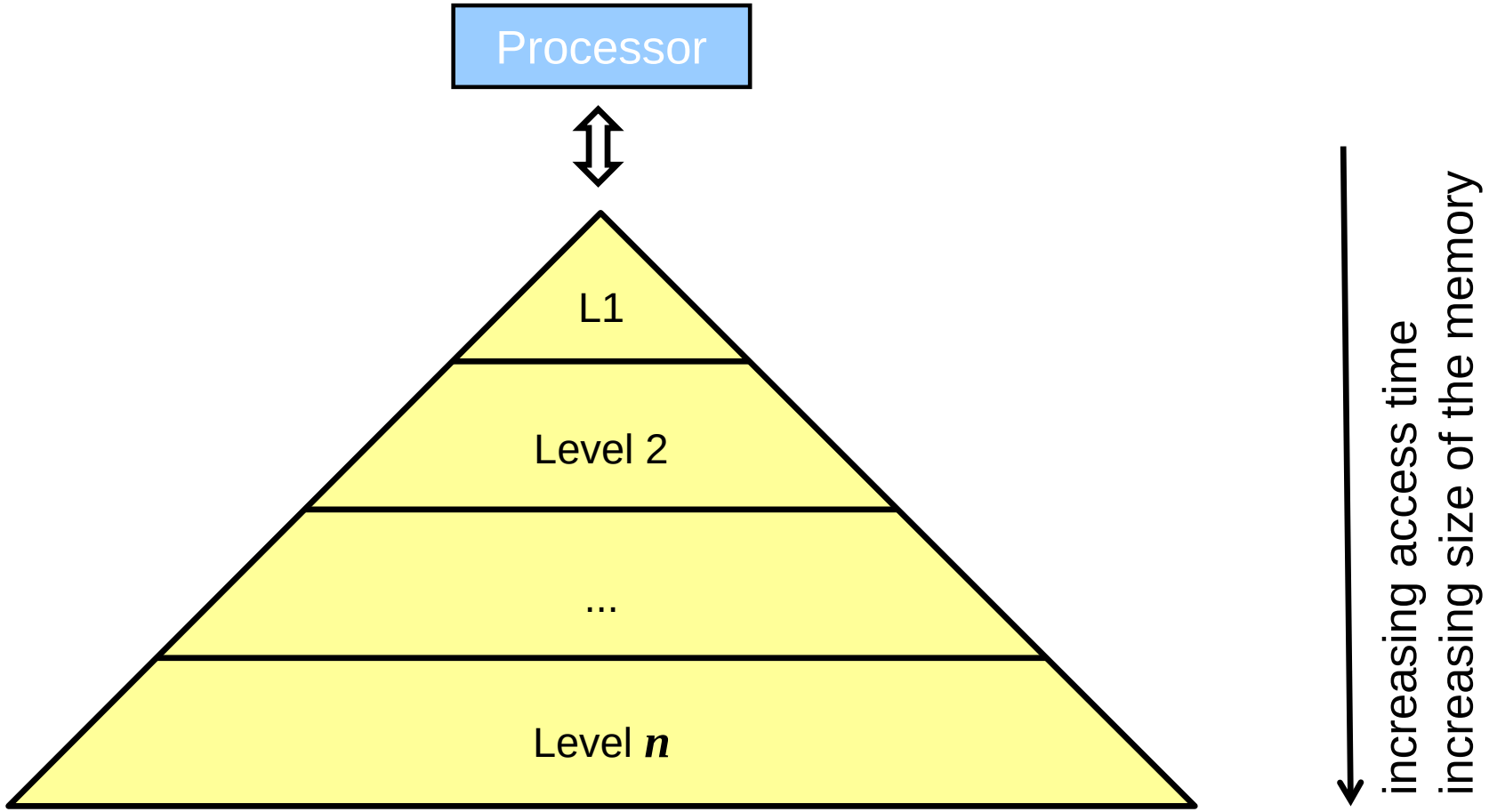
NUMA multiprocessors can usually scale to larger sizes. However, they are harder to program, because efficient software needs to exploit the discrimination between fast and slow memory access.

Example UMA SMP: Multicore Processor



Memory Hierarchy

Caches are part of the memory hierarchy of a computer system.



Memory technologies

Memory technologies differ in access times and in cost per bit.
Typical values from 2012:

Memory technology	Typical access time	Cost per GB in 2012
SRAM (semiconductor)	0.5 .. 2.5 ns	$500 .. $1000
DRAM (semiconductor)	50 .. 70 ns	$10 .. $20
Flash (semiconductor)	5,000 .. 50,000 ns	$0.75 .. $1.00
Magnetic disk	5,000,000 .. 20,000,000 ns	$0.05 .. $0.10

Caches are typically built with SRAM.
Main memory is typically DRAM.

Shared-Memory Multiprocessing and Caches

Shared-memory multiprocessors can be effectively improved in performance by using local caches, through migration and replication of shared data items:

Migration

Accesses to the shared data items are moved to the local cache. This reduces access latency as well as bandwidth demand on the original shared memory location.

Replication

Simultaneous read accesses to shared memory by several processors are carried out on the local cache copies. This reduces contention on the original shared memory location.

However, caching shared memory poses the cache coherence problem.

Cache Coherence

Every processor in a shared-memory multiprocessor system has its own view of the shared memory in its cache. Making sure that these views are equal for all processors is called the cache coherence problem.

Example of flawed system:

Time step	Event	CPU A Cache	CPU B Cache	Memory Loc. X
0				17
1	CPU **A** reads X (*cache miss*): 17	**17**		17
2	CPU **B** reads X (*cache miss*): 17	17	**17**	17
3	CPU **A** writes 53 into X	**53**	17	**53**
4	CPU **B** reads X (*cache hit*): **17**	53	17	53

(Cache Coherence)

When is a cache memory system coherent?

1. If a processor P writes a value V to a location X and the same processor P later reads X, with no other writes to X by other processors in between, then P should read the value V.

2. If a processor P writes a value V to a location X and another processor Q later reads X, with no other writes to X by other processors in between, then Q should read the value V.

3. Writes to the same location must be serialized: Writes to the same location by one or more processor are seen in the same order by all processors.

Cache Coherence Protocols

Coherence of caches in shared-memory MP systems is maintained using so-called cache coherence protocols. These protocols keep track of the state of shared memory locations.

The most important type are snooping protocols. In these protocols, the sharing state is distributed over the memory system because it is stored locally in the caches. The cache controllers monitor any accesses to memory blocks they have cached. This requires that any changes to memory blocks be broadcast to every cache that is affected by the change.

For example, in bus-based implementations, all bus transactions are visible to every node connected to the bus. A node listens to all bus transactions, i.e., also to those in which it is neither master nor slave, in order to find out about the status of memory blocks it holds in its cache. It “snoops” on the bus.

Note: Another class of cache coherence protocols: directory-based. These store the sharing status in one location, the directory.

Write-Invalidate Snooping

One way of enforcing coherence is to make write access to shared memory locations exclusive to a processor/cache node. Such protocols are called write-invalidate protocols. Before a processor writes to a memory location it invalidates all copies of the memory location in other caches. A read to the location by another processor causes a cache miss and a load of the updated value.

If it happens that two processors try to write to the same location one will be the first to send the “invalidate” transaction. The other will miss in the cache, load an updated value and then send the “invalidate” for its own write. This enforces write serialization.

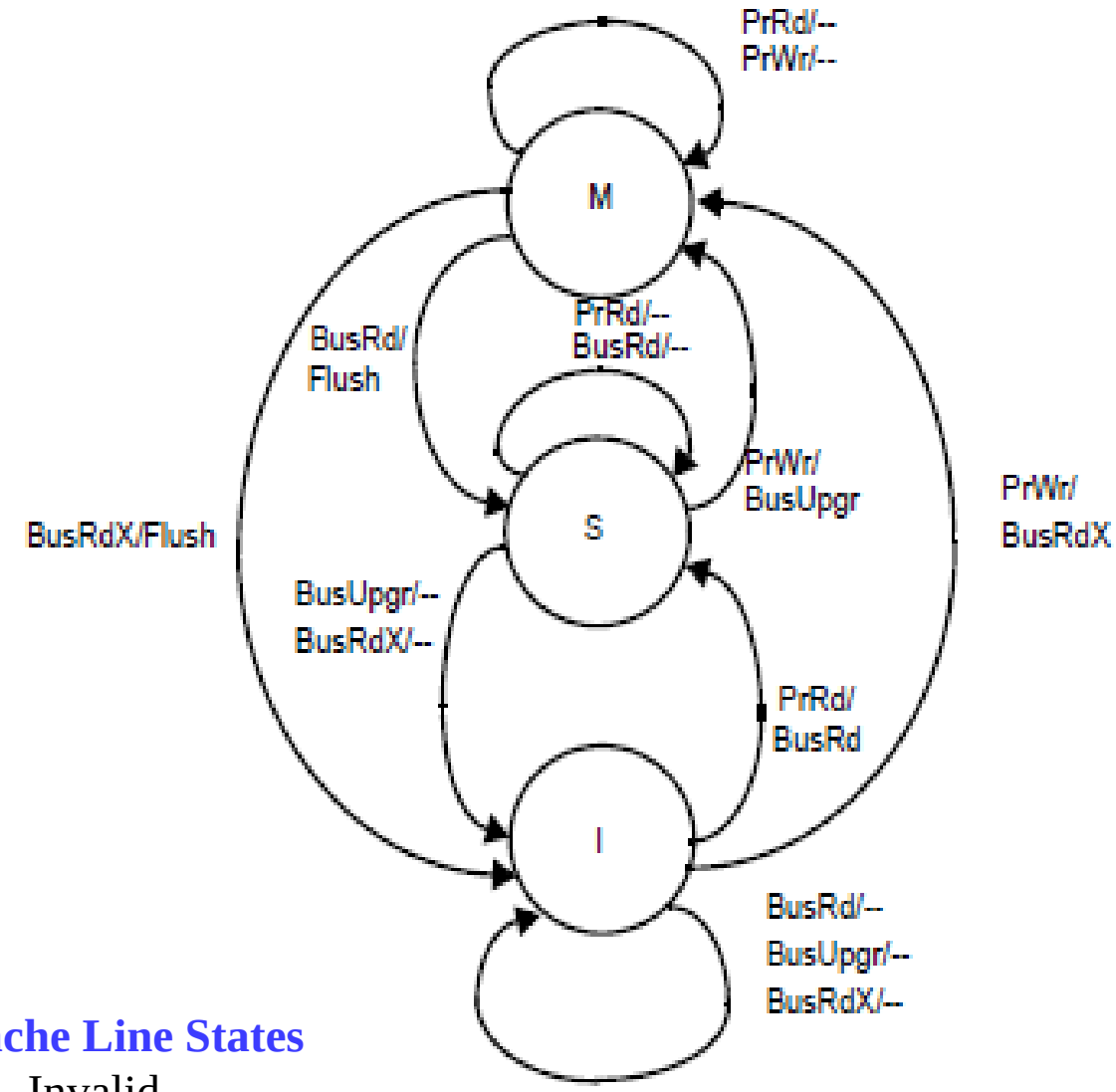
Write-Invalidate Snooping (Example)

Assume a write-back cache (memory is not updated until a cache line is replaced, i.e., old cache line written back, new line read in). In this case, write back must also occur on read requests by other nodes.

Time step	Event	Bus activity	CPU A Cache	CPU B Cache	Mem. Loc. X
0					17
1	CPU **A** reads X (17)	Cache miss for X	**17**		17
2	CPU **B** reads X (17)	Cache miss for X	17	**17**	17
3	CPU **A** writes 53 to X	Invalidation for X	**53**		17
4	CPU **B** reads X (53)	Cache miss for X	53	**53**	**53**

When CPU A writes to X (at time step 3) all other caches that have a copy of X are invalidated. When CPU B reads X (at time step 4) it misses in the cache. The read transaction from B is answered by A, canceling the response from memory. The memory as well as B's cache are updated.

MSI Protocol State Machine



Processor -side Requests	
PrRd	Processor read
PrWr	Processor write

Bus-side Requests	
BusRd	Read request for a cache line; no intent to modify
BusUpgr	Invalidate other copies
BusRdX	Read cache line and invalidate other copies
Flush	supply a cache line to a requesting remote cache; update memory at the same time

Cache Line States

I: Invalid

S: Shared: one or more copies of the cache line exist, memory is clean

M: Modified (“dirty”): one copy in local cache, memory is stale

Hardware Multithreading

Multithreading refers to the capability of a single processor core to concurrently execute several threads. It represents a way of exploiting thread-level parallelism without duplicating all of the hardware of a core as in a multi-core design. Only the private execution state variables of a thread such as registers, program counter, page tables are replicated for each thread. The threads share the functional units of a core.

The goal is to increase efficiency by compensating pipeline and memory latencies (cache misses).

Thread switching inside the hardware must happen very quickly, (much faster than context switching in an operating system).

There are three main forms of multithreading:

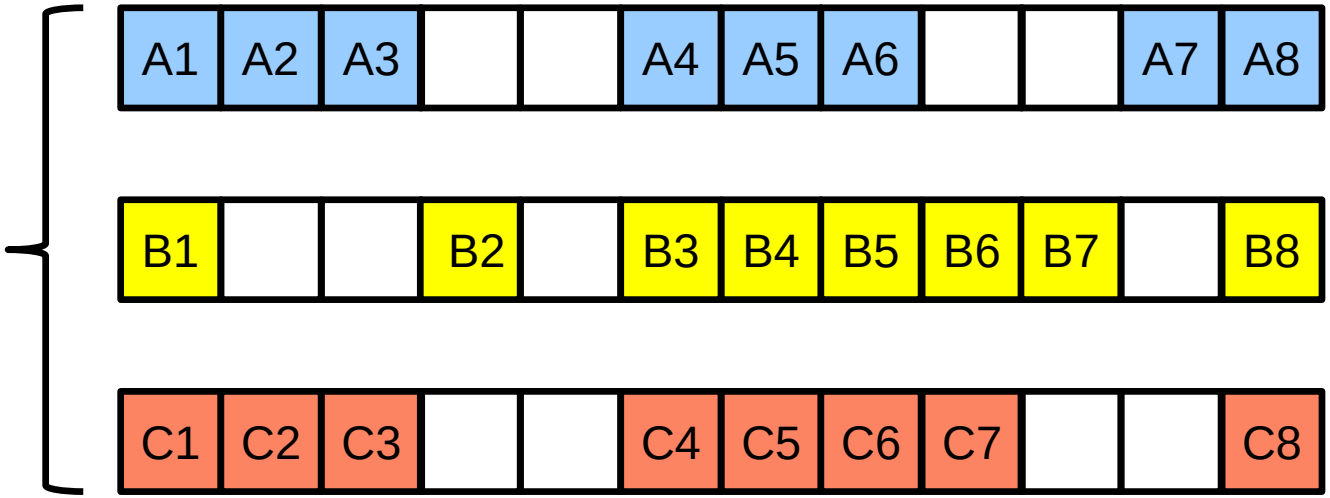
- Coarse-grained multithreading
- Fine-grained multithreading
- Simultaneous multithreading (SMT)

Coarse-grained Multithreading

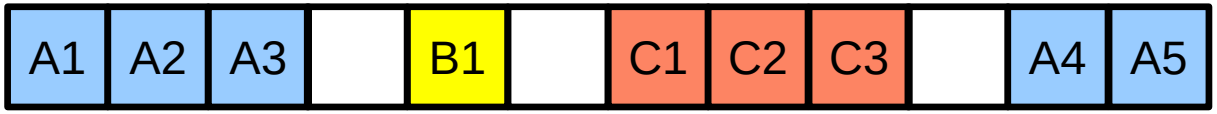
In coarse-grained multithreading the processor switches threads only on expensive stalls (such as level-2 or level-3 cache misses).

Basic idea:

Three threads, as they are executed on a single-issue pipeline without multithreading. Stalls due to cache misses.



Coarse-grained multithreading

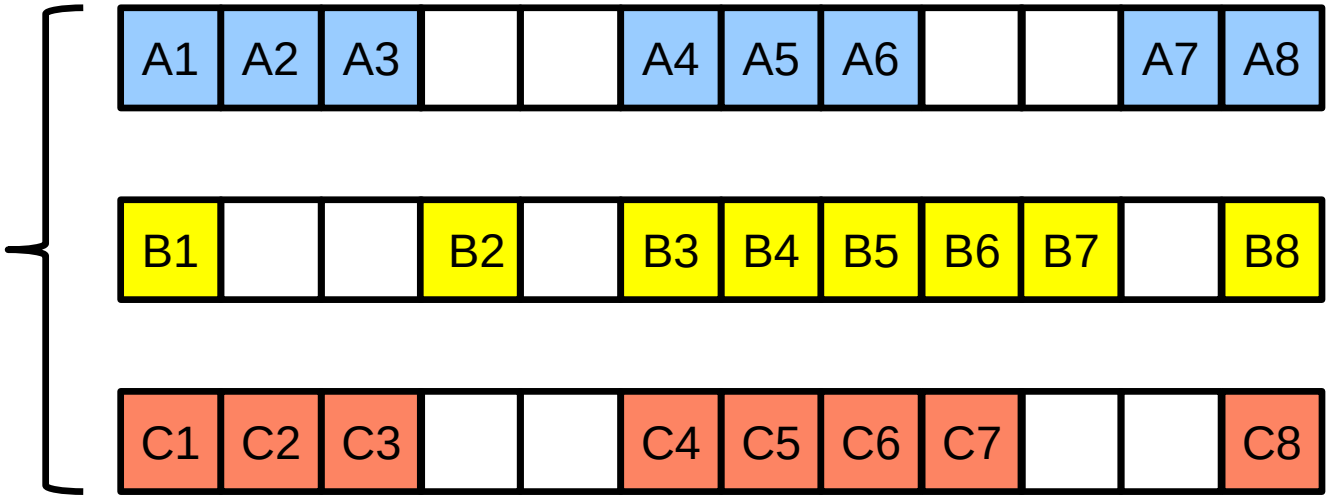


Fine-grained Multithreading

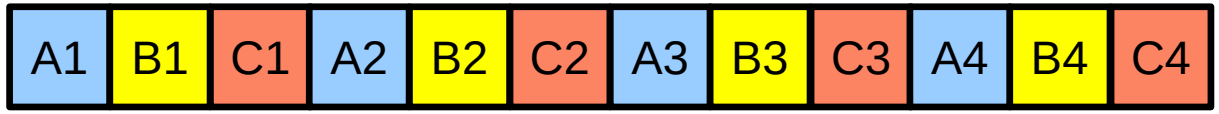
In fine-grained multithreading the processor switches between threads in every clock cycle. It selects the threads round-robin, skipping over threads that are currently stalled.

Basic idea:

Three threads, as they are executed on a single-issue pipeline without multithread-ing. Stalls due to cache misses.



Fine-grained multithreading



Simultaneous Multithreading

Simultaneous multithreading (SMT) is a variation of fine-grained multithreading for a multiple-issue processor with dynamic scheduling. The idea is that the dynamic scheduling mechanisms that are already in place can handle parallel execution of instructions from independent threads.

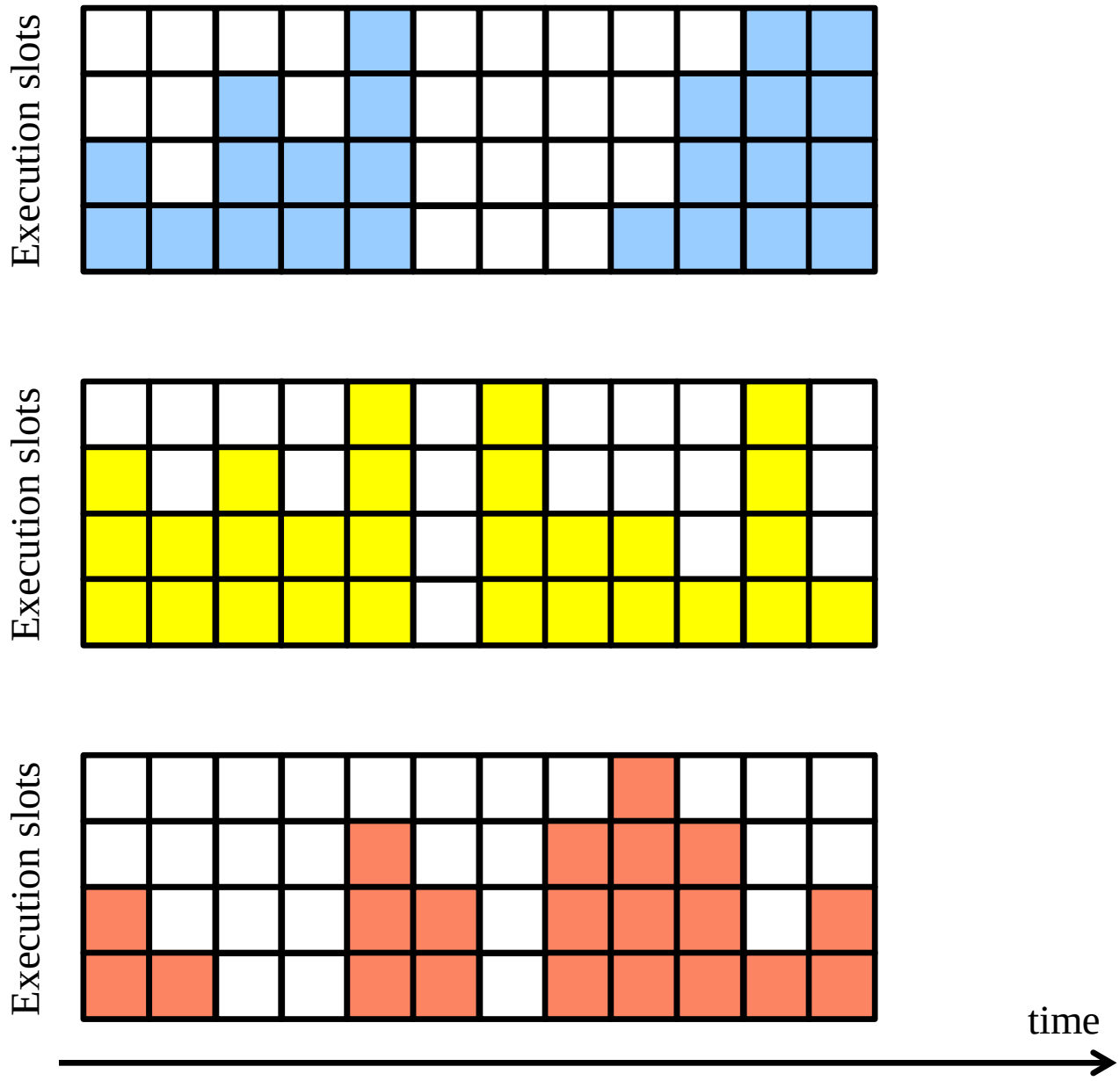
As in fine-grained multithreading, in every clock cycle only instructions from one thread are issued. However, instructions from several threads may initiate execution in the same cycle, when they become ready as determined by the dynamic scheduling hardware.

The next slides show an example of the three multithreading approaches for a superscalar pipeline.

Multithreading on a Superscalar Pipeline (1)

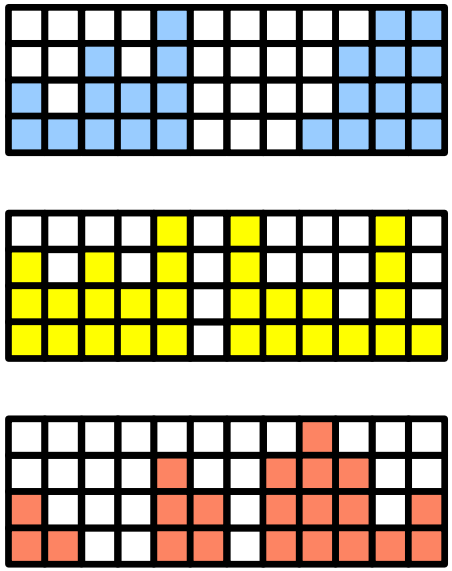
Example:

Three threads, as they are executed on a multiple-issue pipeline without multithreading. Threads exploit ILP but completely idle cycles are possible due to memory latencies.

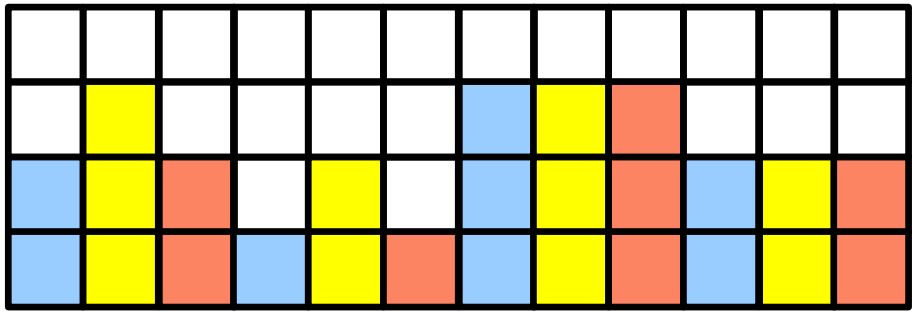


Multithreading on a Superscalar Pipeline (2)

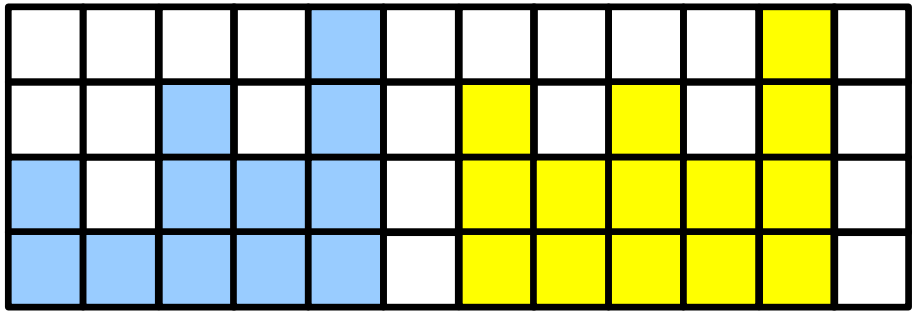
(Original threads, without MT)



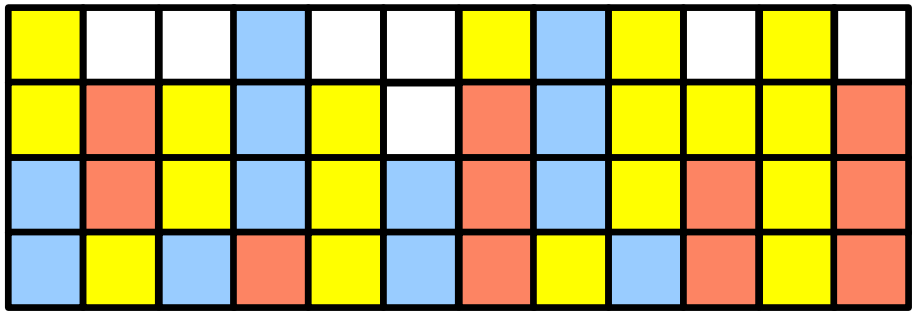
Fine-grained multithreading



Coarse-grained multithreading



Simultaneous multithreading (SMT)



SMT is an implementation of fine-grained multithreading on a multiple-issue processor with dynamic scheduling. Instructions from more than one thread can execute in a given clock cycle. The dynamic scheduling hardware determines what instructions are ready to initiate execution.

time

Flynn's Taxonomy

Michael J. Flynn proposed the following well-known classification of computer architectures in 1966.

		Data streams	
		Single	Multiple
Instruction streams	Single	**SISD**: Uniprocessors	**SIMD**: Vector processors, GPUs, multimedia extensions (MMX, SSE). Exploits **data-level parallelism**.
	Multiple	**MISD**: (No examples)	**MIMD**: Multiprocessors