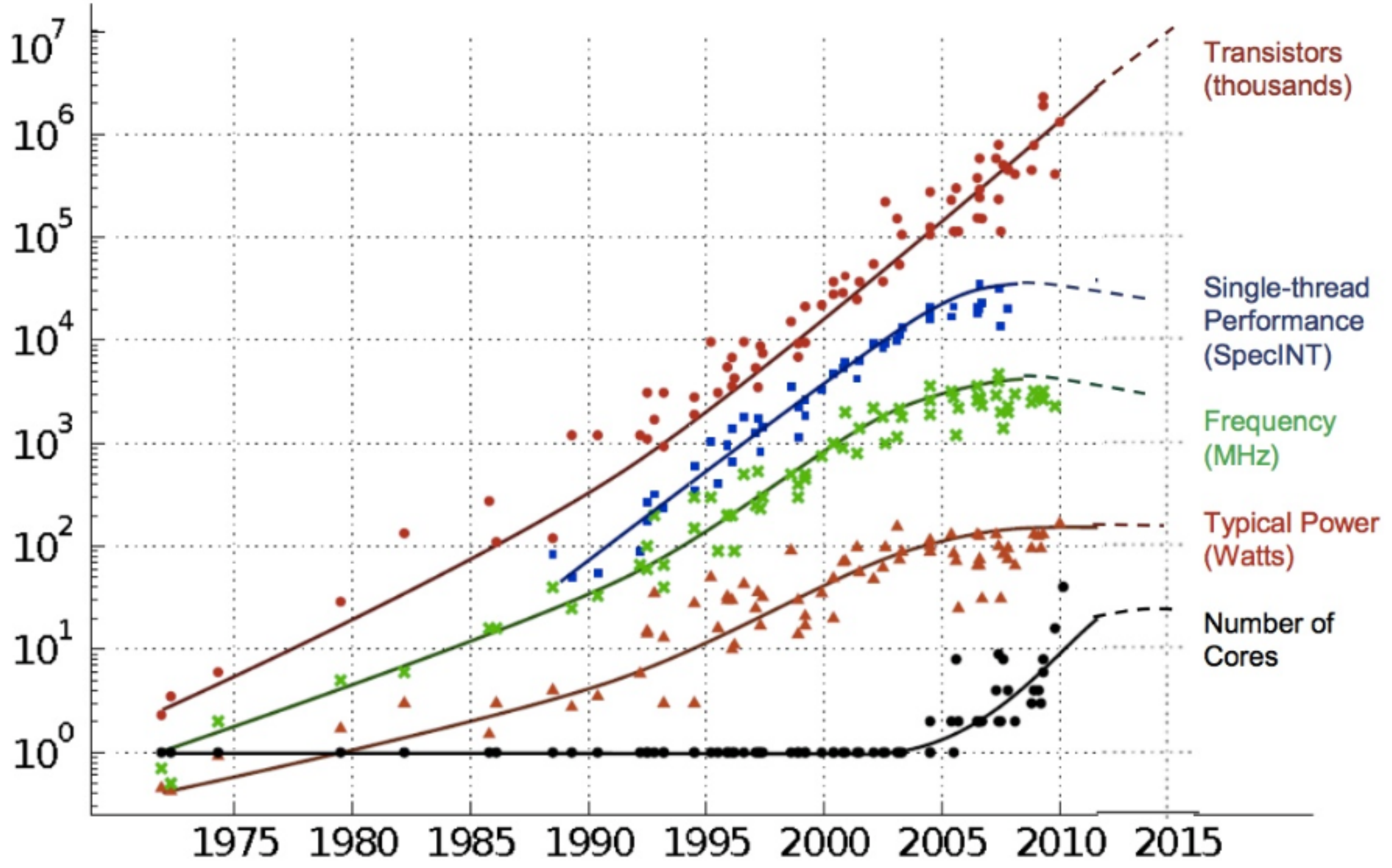
Parallel Computing

Prof. Marco Bertini

Modern CPUs

UNIVERSITÀ DEGLI STUDI FIRENZE

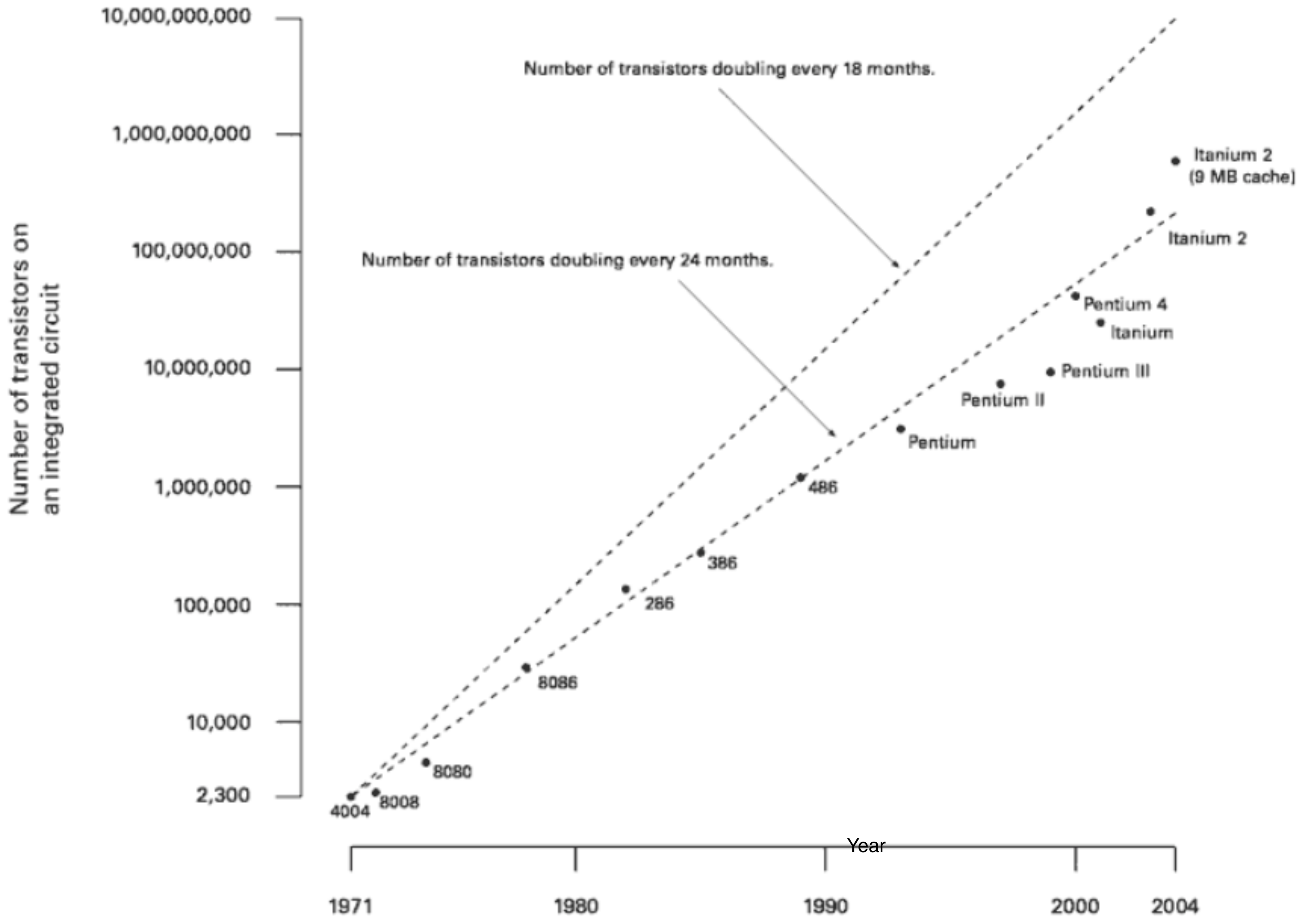
Historical trends in CPU performance



From ‘Data processing in exascale class computer systems’, C. Moore

http://www.lanl.gov/orgs/hpc/salishan/salishan2011/3moore.pdf

Moore's law



- Still works for # of transistors, not for clock speed.

Memory / Caches

- There's need of hundreds of CPU cycles to access RAM. To avoid to wait too much for data the solution is to use:
  - Several layers of cache memory;
  - Prefetch data: even if we write a dumb loop that reads and operates on a large block of 64-bit (8-byte) values, the CPU is smart enough to prefetch the correct data before it's needed. E.g. it may be possible to process at about 22 GB/s on a 3Ghz processor.
    - A calculation that can consume 8 bytes every cycle at 3Ghz only works out to 24GB/s (we're losing just ~8% performance by having to go to main memory)

Out of Order Execution

- Since several years x86 chips have been able to speculatively execute and re-order execution (to avoid blocking on a single stalled resource).
- But a x86 CPU is required to update externally visible states, like registers and memory, as if everything were executed in order.
- The implementation of this involves making sure that, for any pair of instructions with a dependency, those instructions execute in the correct order with respect to each other.
- This implies also that loads and stores to the same location can’t be reordered with respect to each other.

UNIVERSITÀ DEGLI STUDI FIRENZE

Memory / Concurrency

- x86 loads and stores have some other restrictions:
  - In particular, for a single CPU, stores are never reordered with other stores, and…
  - …stores are never reordered with earlier loads,
- regardless of whether or not they're to the same location.

In this example, and all the following, will be used the AT&T assembler style: `OP src, dst`
Intel uses the `OP dst, src` format.

UNIVERSITÀ DEGLI STUDI FIRENZE

Memory / Concurrency

- x86 loads anc restrictions:
  - In particula are never reordered v
  - …stores ar rlier loads,
- regardless of he same location.

```
mov 1, [%esp]
mov [%ebx], %eax
```

can be executed as

```
mov [%ebx], %eax
mov 1, [%esp]
```

but not vice-versa.

In this example, and all the following, will be used the AT&T assembler style: `OP src, dst`
Intel uses the `OP dst, src` format.

Memory / Concurrency

- The fact that loads may be reordered with older stores has an effect in multi-core systems:
  - Let us have x and y are in shared memory, both initialized to zero, while r1 and r2 are processor registers.

Thread 0	Thread 1
`mov 1, [_x]`	`mov 1, [_y]`
`mov [_y], r1`	`mov [_x], r2`

  - When the two threads execute on different cores, the non-intuitive result `r1 == 0` and `r2 == 0` is allowed.
    - Notice that such result is consistent with the processor executing the loads before the stores (which access different locations).

UNIVERSITÀ DEGLI STUDI FIRENZE

Memory / Concurrency

- The fact that loads may be reordered with older stores has an effect in multi-core systems:
  - Let us have x and y are in shared memory, both initialized to zero, while r1 and r2 are processor registers.

```
     Thread 0              Thread 1
  mov 1, [_x]           mov 1, [_y]
  mov [_y], r1          mov [_x], r2
```

This may even break algorithms like the Peterson lock that is used to perform mutual exclusion for two threads.

The solution is to use appropriate additional x86 instructions that force the ordering of the instructions, i.e. memory carries (aka fences)

- When the … on-intuitive r…
  - Notice … or execut… different locations).

UNIVERSITÀ DEGLI STUDI FIRENZE

Memory / Concurrency

- There's also multi-core ordering. The previous restrictions all apply; if core0 is observing core1, it will see that all of the single core rules apply to core1's loads and stores.
- However, if core0 and core1 interact, there's no guarantee that their interaction is ordered:
- For example, say that core0 and core1 start with eax and edx set to `0`, `[_foo]` and `[_bar]` in shared memory set to `0`, and core0 executes

```
mov 1, [_foo] ; move 1 to the bytes in memory at address _foo

mov [_foo], %eax ; move the content of he memory addressed by _foo in EAX

mov [_bar], %edx ; move the content of he memory addressed by _bar in EDX
```

- while core1 executes

```
mov 1, [_bar] ; move 1 to the bytes in memory at address _bar

mov [_bar], %eax ; move the content of the memory addressed by _bar in EAX

mov [_foo], %edx ; move the content of the memory addressed by _foo in EDX
```

- For both cores, eax has to be 1 because of the within-core dependency between the first instruction and the second instruction. However, it's possible for edx to be `0` in both cores because line 3 of core0 can execute before core0 sees anything from core1, and vice-versa.

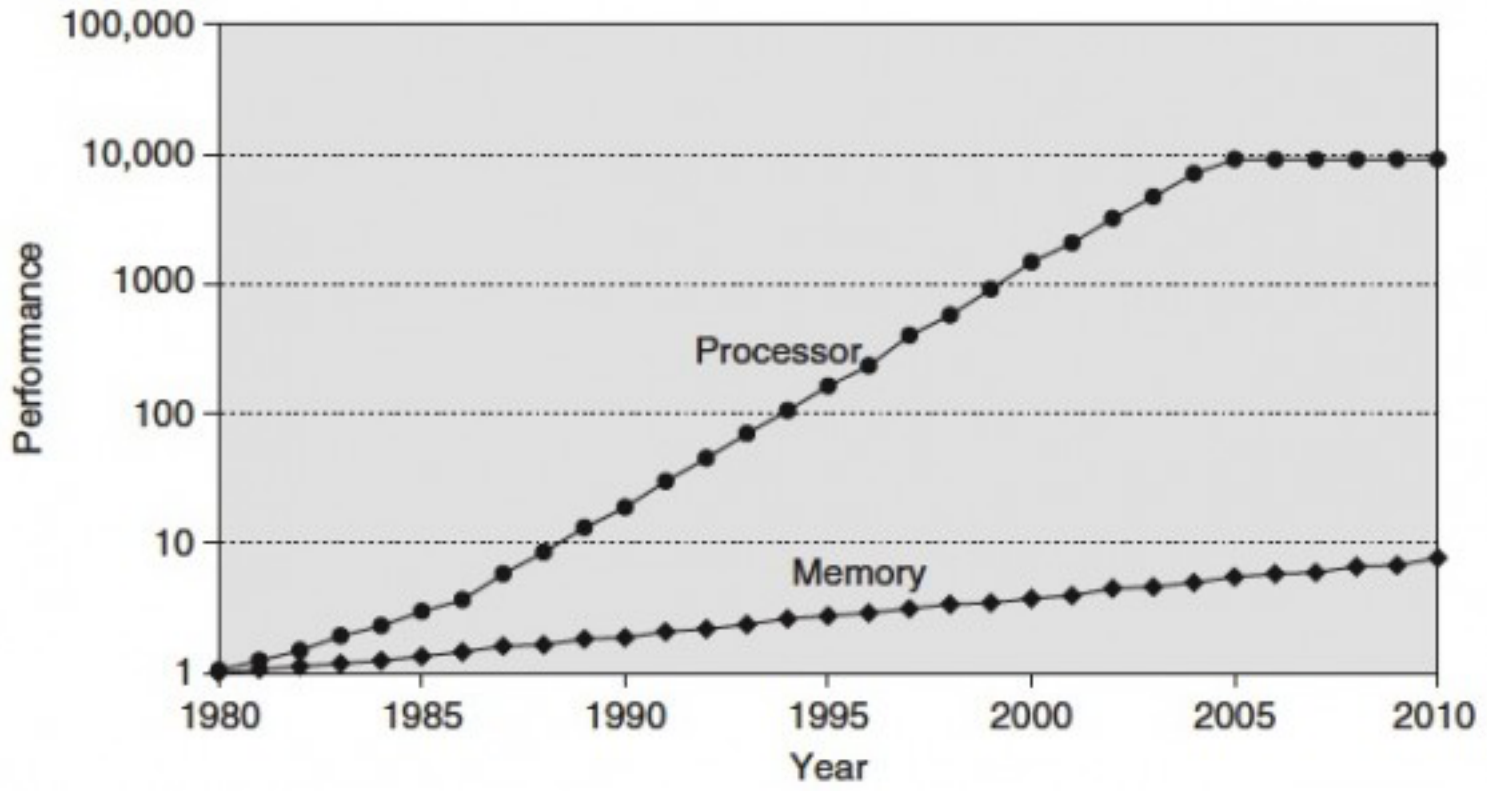
Locking

- In modern x86 CPUs locking is cheaper than memory barriers.
- It is possible to lock some instructions using the `lock` prefix
- In addition to making a memory transaction atomic, locks are globally ordered with respect to each other, and loads and stores aren't re-ordered with respect to locks.

UNIVERSITÀ DEGLI STUDI FIRENZE

Memory architecture

- The importance of the memory architecture has increased with the advances in performance and architecture in CPU.
  - The CPU performance plateau is due to the introduction of multi-core architectures.

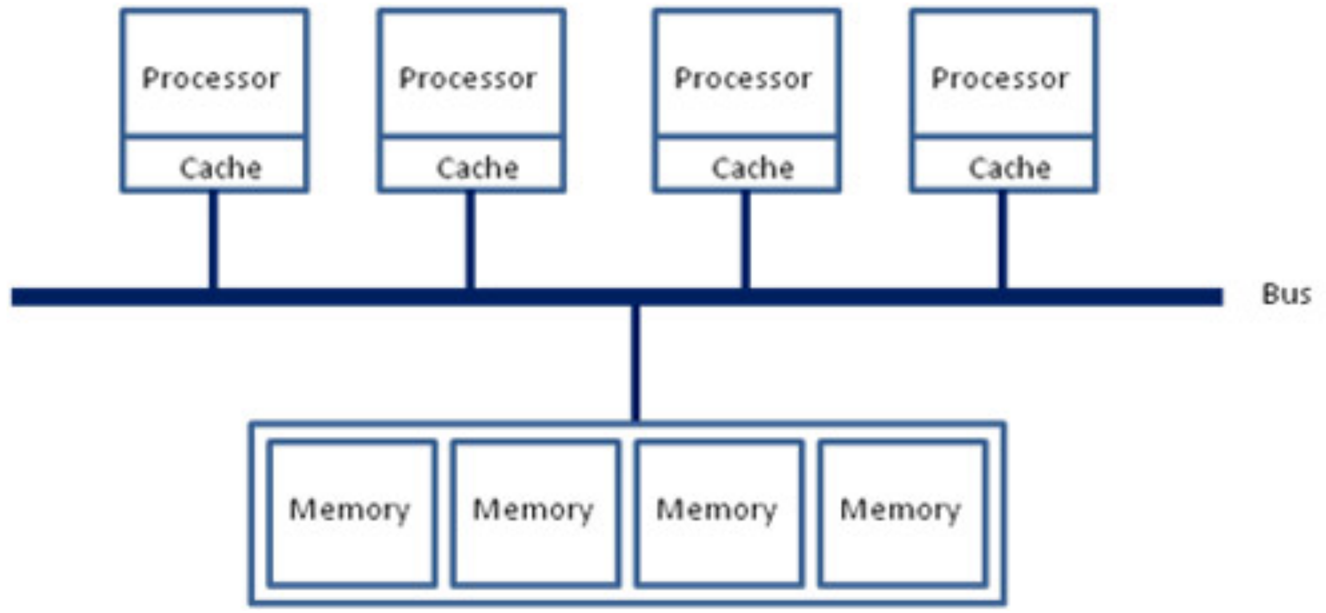


Source: Computer Architecture, A quantitative Approach by Hennessy and Patterson

Memory/ UMA

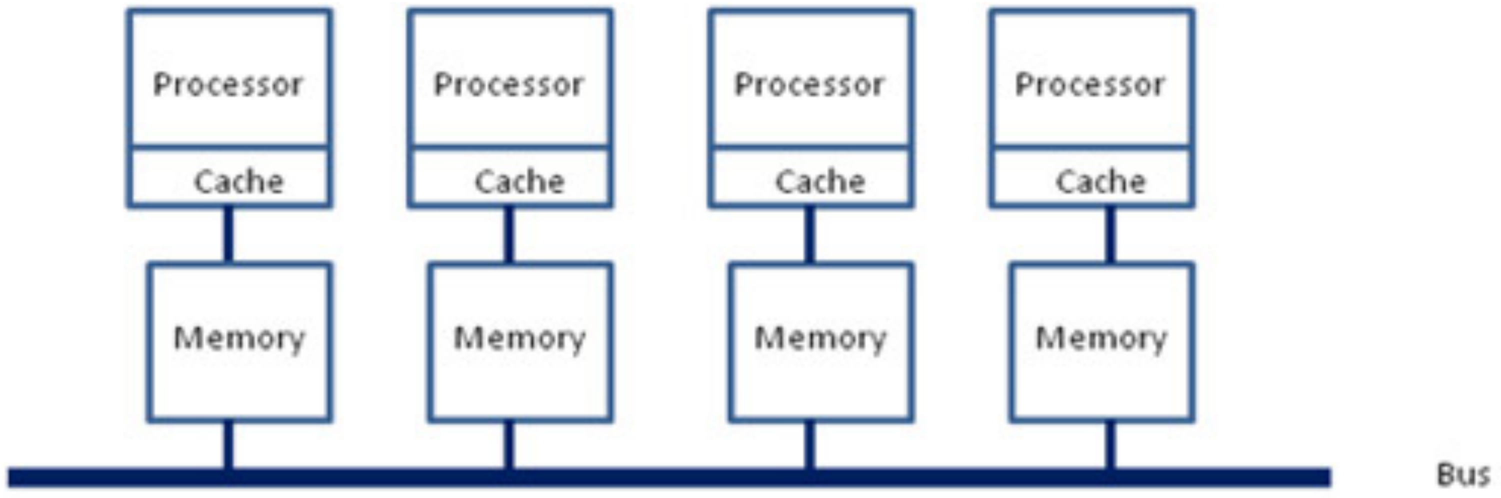
- Shared Memory Architecture is split up in two types: Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA).

- UMA: memory is shared across all CPUs. To access memory, the CPUs have to access a Memory Controller Hub (MCH). This type of architecture is limited on scalability, bandwidth and latency.



Memory / NUMA

- With NUMA memory is directly attached to the CPU and this is considered to be local.
- Memory connected to another CPU socket is considered to be remote. To access it there is need to traverse the interconnect and connect to the remote memory controller.
- As a result of the different locations memory can exists, this system experiences “non-uniform” memory access time.



Memory / NUMA

- Non-uniform memory access, where memory latencies and bandwidth are different for different processors, is so common that we can we assume it as default.
- Threads that share memory should be on the same socket, and a memory-mapped I/O heavy thread should make sure it’s on the socket that’s closest to the I/O device it’s talking to.

Memory / NUMA

- It's hard to sync more than 4 CPUs and their caches w.r.t. memory: each one has to warn the others about the memory that it's going to touch… the bus would saturate.

- Multi-core CPUs have a directory to reduce the N-way peer-to-peer broadcast, but the problem with CPUs is still valid.

  - The simplest solution is to have each socket to control some region of memory. You pay e penalty performance when trying to access memory assigned to other chips.

  - Typically <128 CPUs system have a ring-like bus: you 1) pay the latency/bandwidth penalty for walking through a bunch of extra hops to get to memory, 2) use a finite resource (the ring-like bus) and 3) slow down other cross-socket accesses.

- All this is handled by the O.S.

Context Switches / Syscalls

- A side effect of all the caching that modern cores have is that context switches are expensive. In HPC it is better to:
  - use a thread-per-core rather than thread-per-logical-task
  - use userspace I/O stacks for very high performance I/O.

SIMD

- all modern x86 CPUs support SSE, 128-bit wide vector registers and instructions.
- Since it’s common to want to do the same operation multiple times, Intel added instructions that will let you operate on
    - a 128-bit chunk of data as 2 64-bit chunks,
    - 4 32-bit chunks,
    - 8 16-bit chunks,
    - etc.
- ARM supports the same thing with a different name (NEON).

Example: SIMD

Let us suppose to have v1 and v2 that are 4-dim float vectors, and we want to sum them with SSE instructions:

```
movaps [v1], xmm0 ;xmm0 = v1.w | v1.z | v1.y | v1.x

addps  [v2], xmm0 ;xmm0 = v1.w+v2.w | v1.z+v2.z
                  ;     | v1.y+v2.y | v1.x+v2.x

movaps xmm0, [vec_res]
```

it's faster than performing 4 `fadd` on the single components of the vectors

SIMD

- Compilers are good enough at recognizing common patterns that can be vectorized in simple code.
- E.g. the following code, will automatically use vector instructions with modern compilers:

```
for (int i = 0; i < n; ++i) {
  sum += a[i];
}
```

SIMD

- It is possible to use directly SIMD instructions like SSE in C/C++, using “intrinsic” / “builtin” functions, that are directly mapped to CPU instructions.

- Include the required header and ask the compiler to activate the required SSE level
  - <mmintrin.h> MMX
  - <xmmintrin.h> SSE
  - <emmintrin.h> SSE2
  - <pmmintrin.h> SSE3
  - <tmmintrin.h> SSSE3
  - <smmintrin.h> SSE4.1
  - <nmmintrin.h> SSE4.2
  - <ammintrin.h> SSE4A
  - <wmmintrin.h> AES
  - <immintrin.h> AVX

Simultaneous Multi Threading / Hyper-Threading

- Hyper-Threading Technology is a form of simultaneous multithreading.

- Architecturally, a processor with HT consists of two logical processors per core, each of which has its own processor architectural state.

  - Each logical processor can be individually halted, interrupted or directed to execute a specified thread, independently from the other logical processor sharing the same physical core.

Simultaneous Multi Threading / Hyper-Threading

- The logical processors in a hyper-threaded core share the execution resources.
- These resources include:
  - the execution engine
  - caches
  - system bus interface;
- the sharing of resources allows two logical processors to work with each other more efficiently.
- This works by duplicating certain sections of the processor (5% more die area) - those that store the architectural state - but not duplicating the main execution resources.

Simultaneous Multi Threading / Hyper-Threading

- Performance improvements are very application-dependent
  - high clock CPUs have long pipelines. The CPU scheduler may be optimistic to fill the pipeline with instructions, but they may not be executable (e.g. because of missing data)
  - a specific part of the CPU (replay system in Pentium4) catches those instructions that can not be executed and reissues them until they execute successfully, occupying the execution units
  - if the execution units sit idly (typically 66% time) it’s not a problem, but in HTT this interferes with the execution of the instructions of the other thread.
    - CPUs with a replay queue, that check if the data can be retrieved from the various cache levels or waits for it from memory, suffer less from this.

Intel Turbo-Boost

- It’s dynamic over-clocking of some Intel CPUs
- It is activated when the operating system requests the highest performance state of the processor. These performance states are defined by the open standard Advanced Configuration and Power Interface (ACPI) specification, supported by all major operating systems.
- The increased clock rate is limited by the processor's power, current and thermal limits, as well as the number of cores currently in use and the maximum frequency of the active cores

Credits

- These slides report material from:
  - Dan Luu (Microsoft)