Today

Last lecture

- Quick tour through HPC
- First steps with OpenMP

Today

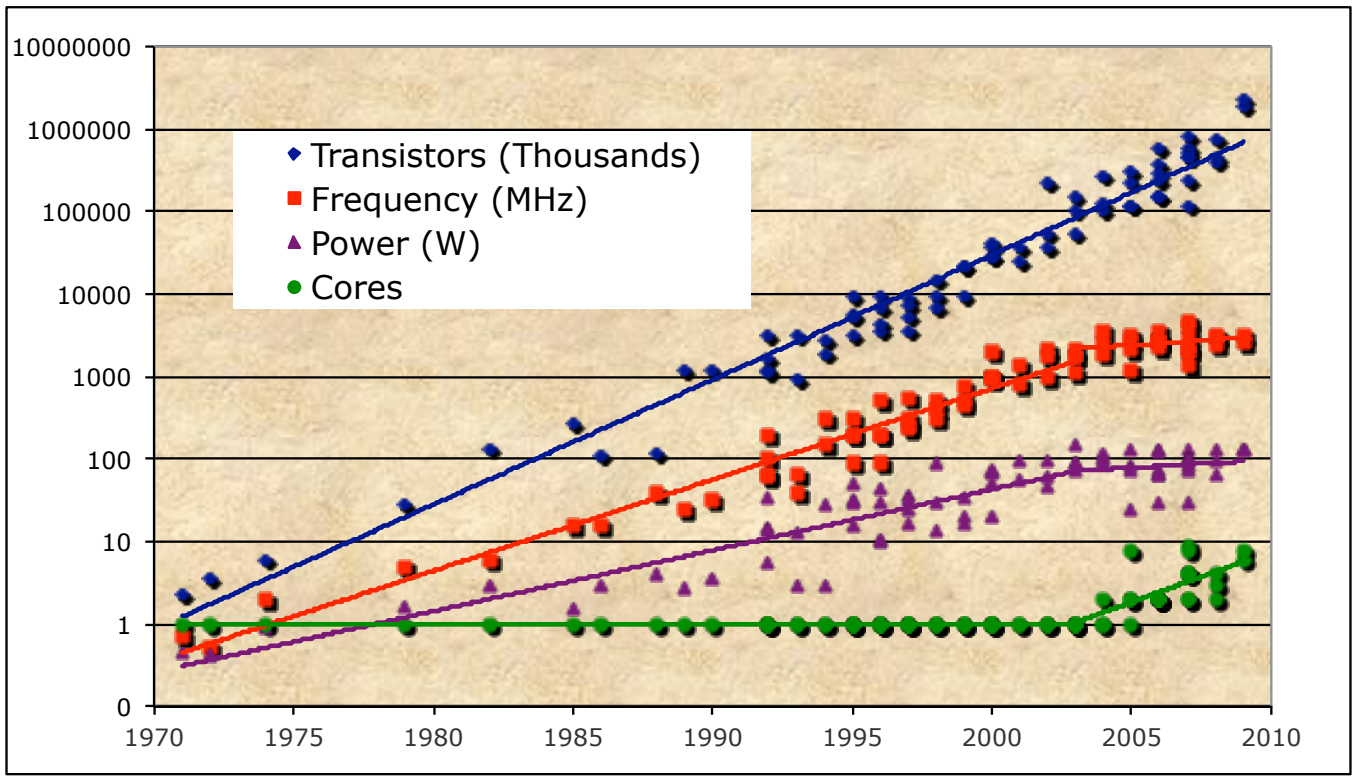
- Memory hierarchies
- Basic performance models
- Tools: valgrind, cachegrind

Announcements and upcoming

- Send me your nyu.edu email address if you audit this course
- Piazza and GitHub link in NYU classes
- First homework is posted (due Mon, Feb 24, 2020)

Moore's law today

- Frequency/clock speed stopped growing in ~ 2004
- Number of cores per CPU
- Moore's law still holds
- Energy use $\sim$bounded



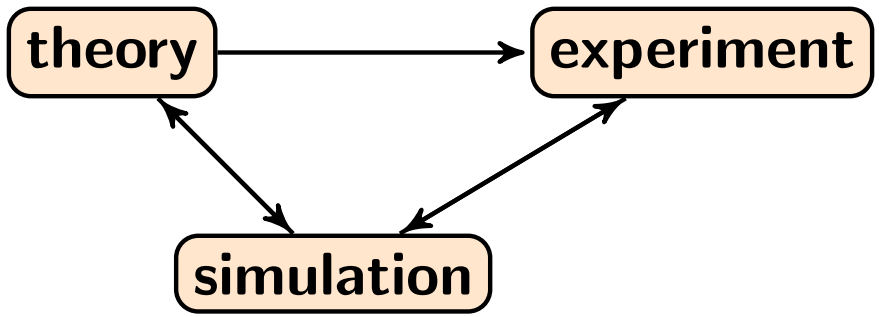
Source: CS Department, UC Berkeley.

Parallel computing $\subset$ high-performance computing

- All major vendors produce multicore chips—need to think differently about applications.
- How well can applications and algorithms exploit parallelism?
- Memory density (RAM) grows at slower rate. Loading/writing to memory is slow ($\sim$100 clock cycles)
- Top500 list: leading machines have $> 10^7$ processor cores, and often two different kinds of compute chips (CPUs and some kind of accelerators (e.g., GPUs)).

Do we really need larger and faster?

Simulation has become the third pillar of Science:

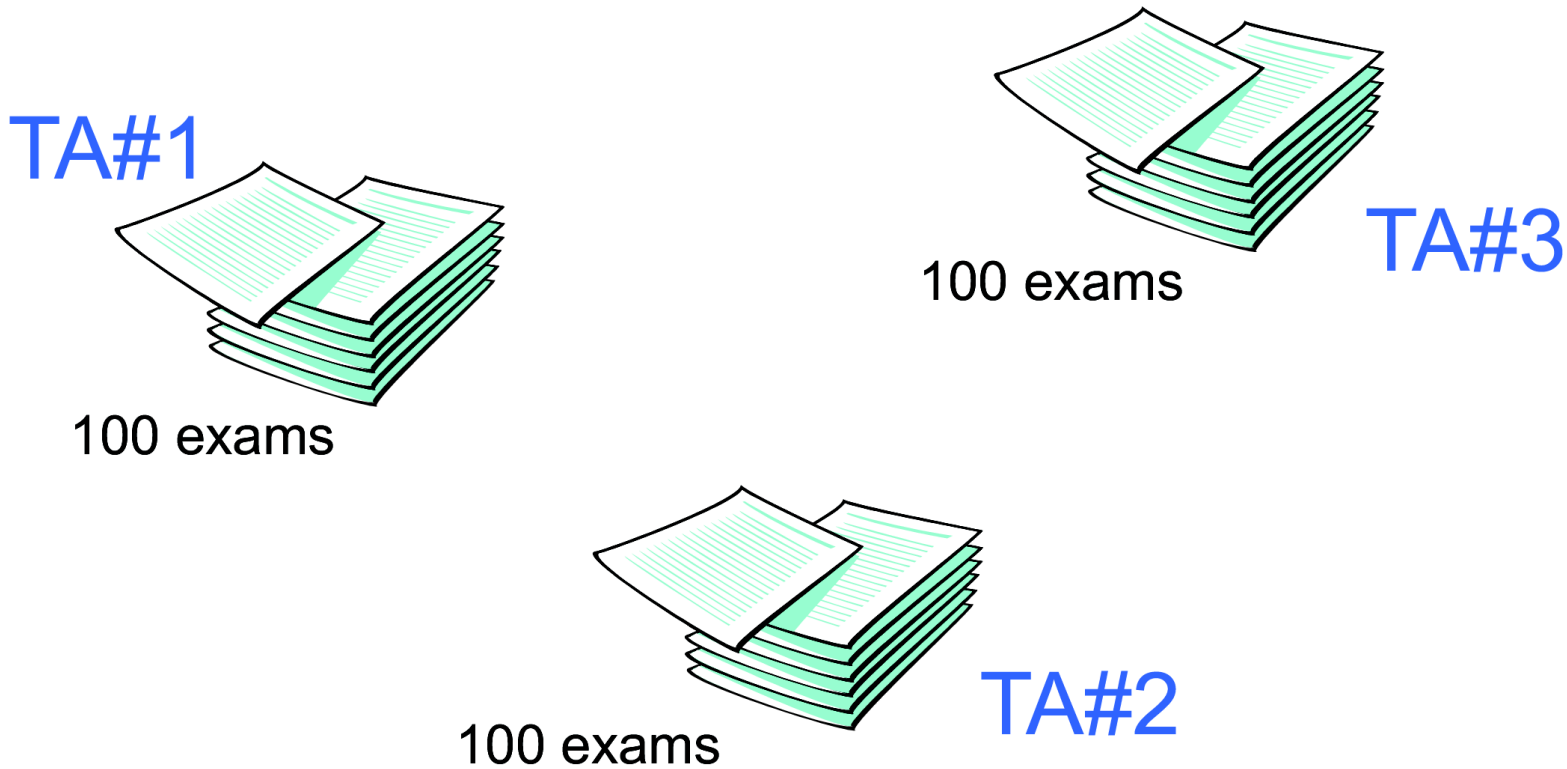


HPC computing used in: weather prediction, climate modeling, drug design, astrophysics, earthquake modeling, semiconductor design, crash test simulations, financial modeling, ...

Basic CS terms recalled

- compiler: translates human code into machine language
- CPU/processor: central processing unit carries out instructions of a computer program, i.e., arithmetic/logical operations, input/output
- core: individual processing unit in a CPU, "multicore" CPU; will sometimes use "processors" in a sloppy way, and actually mean "cores"
- clock rate/frequency: indicator of speed in which instructions are performed
- floating point operation: multiplication add of two floating point numbers, usually double precision (64 bit, about 16 digits)
- peak performance: fastest theoretical flop/s
- sustained performance: flop/s in actual computation

Example: Data parallelism



Example: Task parallelism

TA#1

Questions 1 - 5

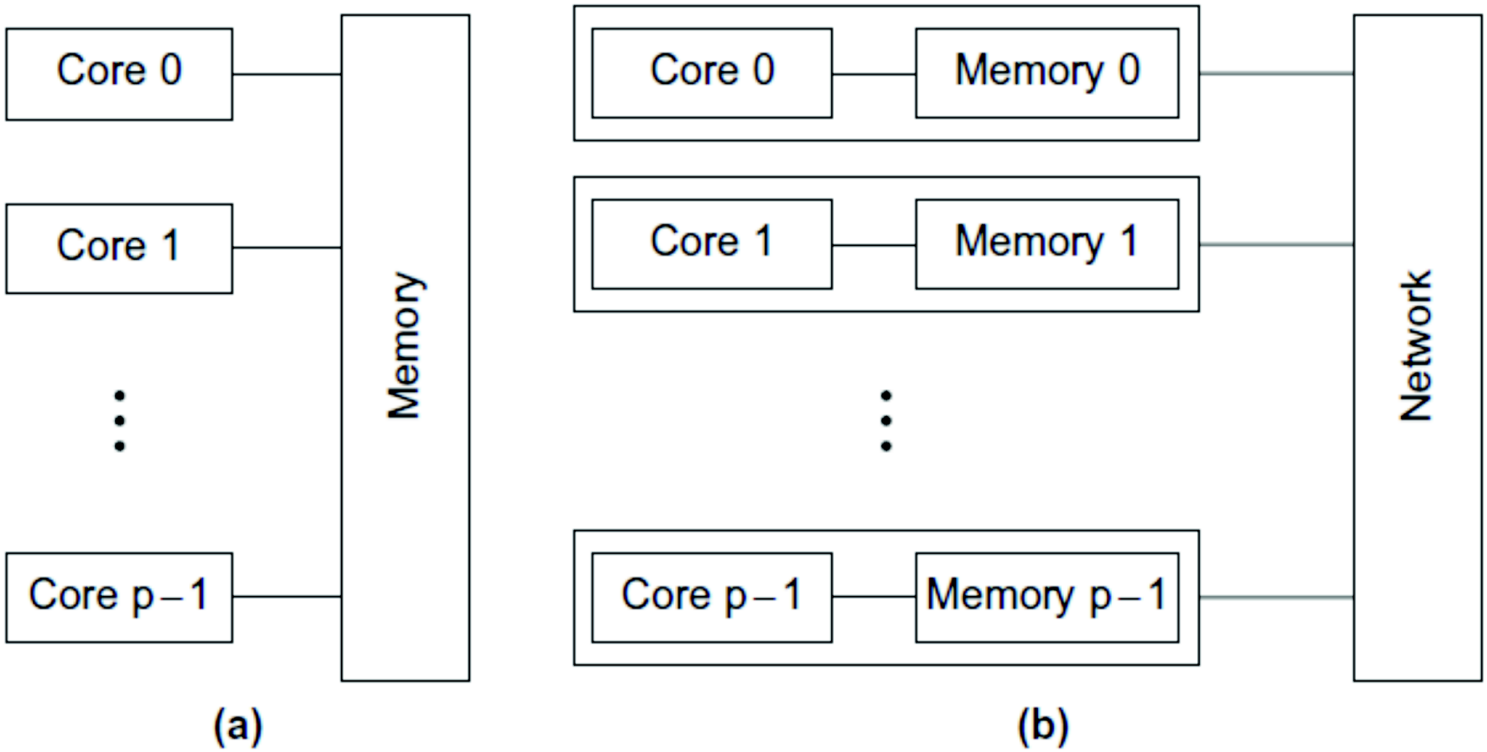
TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

Type of parallel systems (cont'd)

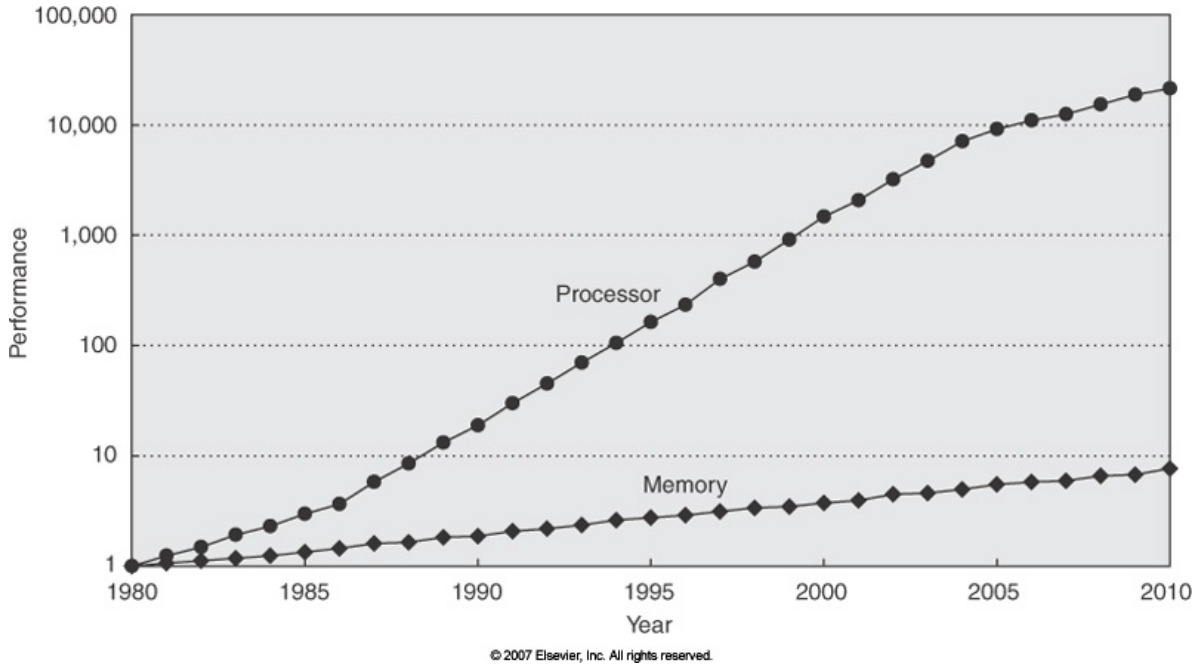


Shared-memory Distributed-memory

[Figure: Elsevier]

Flop/s versus Mop/s

For many practical applications, memory access is the bottleneck, not floating point operations.

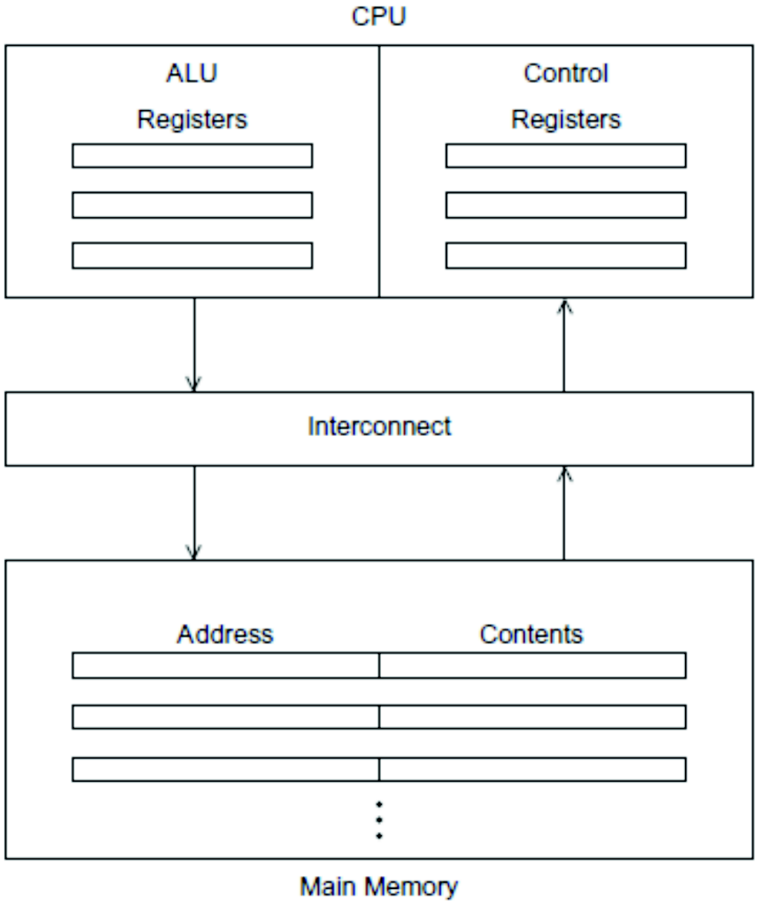


Development of memory versus processor performance.

- Most applications run at $< 10\%$ of the theoretical peak performance.
- Mostly a single core issue; on parallel computers, things become even more difficult.

More about hardware architectures

The von Neumann architecture



- Main memory consists of locations that store instructions and data
- Each location consists of address and contents
- Control unit decides which intstructions the program should execute
- Arithmetic and logic unit (ALU) responsible for executing the actual instructions
- "Von Neumann bottleneck": shared bus between data and program memory; CPU has to wait until data is received from memory

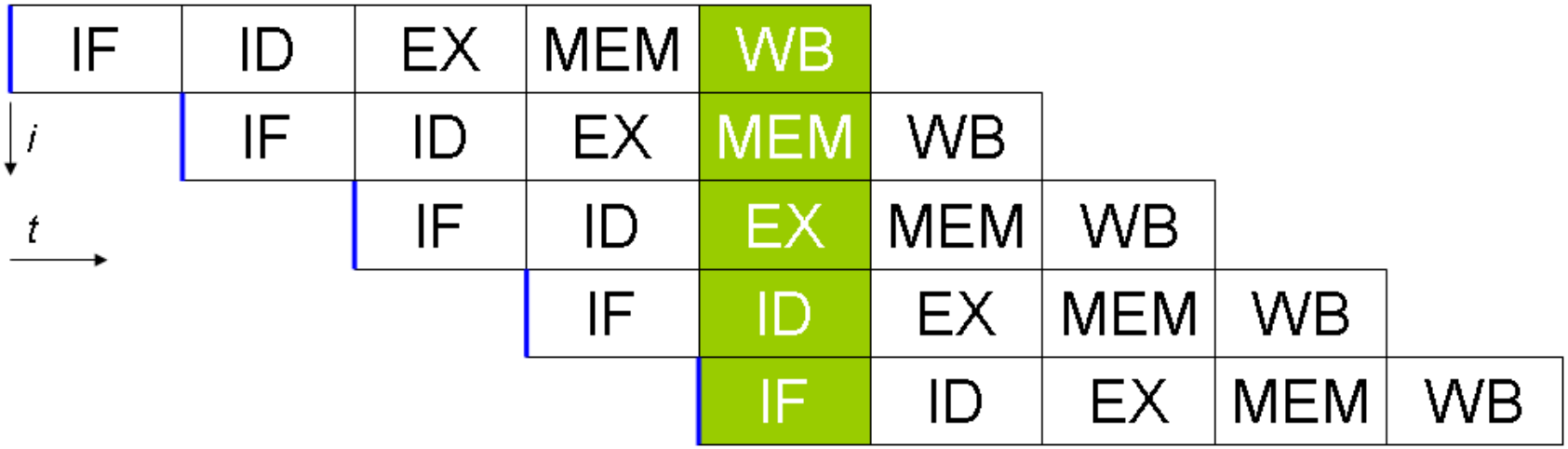
No pipelining

Activities

- Instruction fetch (IF)
- Instruction Decode (ID)
- Execution (EX)
- Memory Read/Write (MEM)
- Result Writeback (WB)

Four out of five units are idle at all time

Pipelining



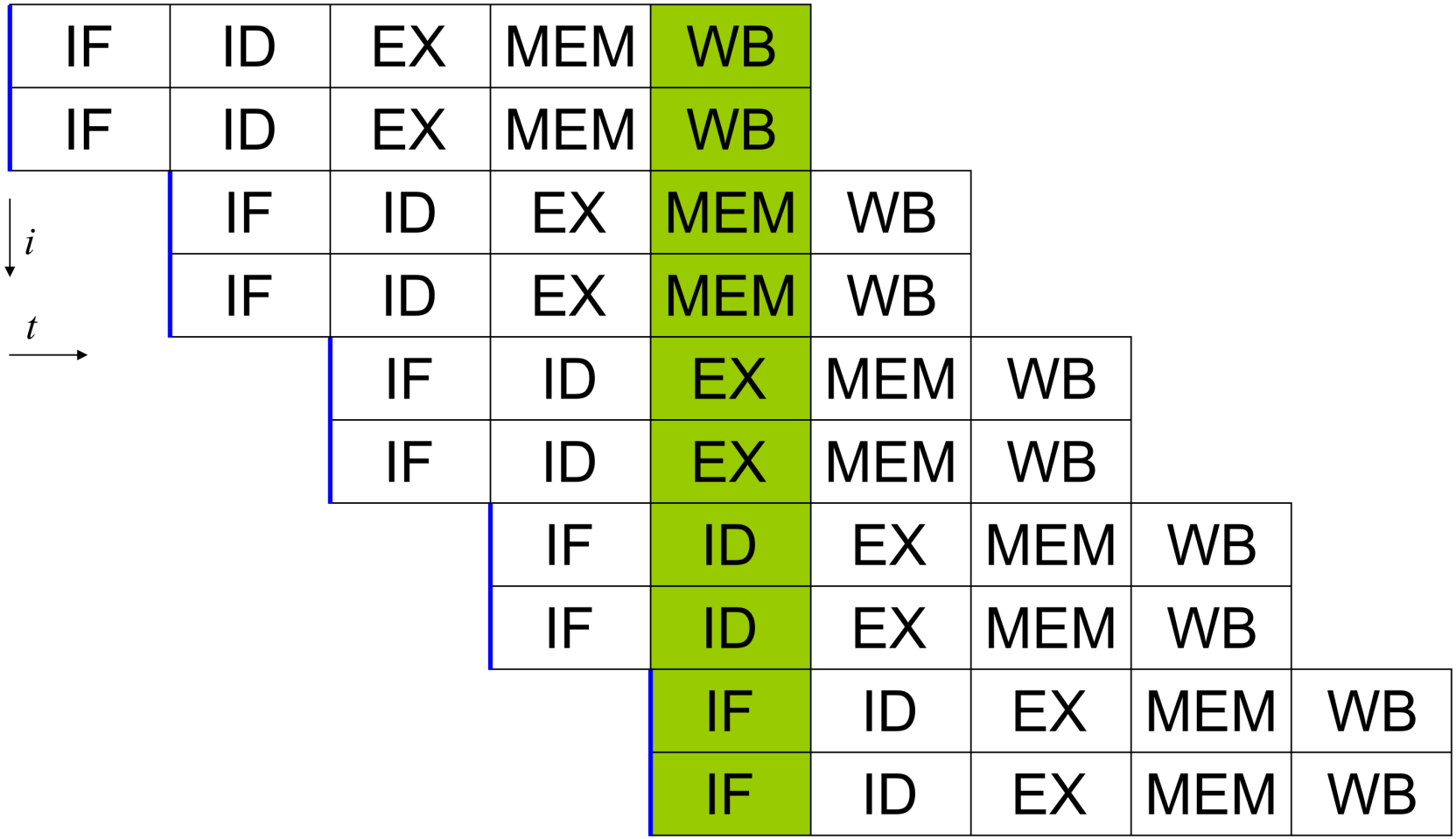
Pipelining

- Improves processor performance by having multiple processor components (functional units) that simultaneously execute instructions
- Number of stages increases with each processor generation

Maximum cycles per instruction (CPI)

- No pipelining in the above gives CPI = 5
  - Each stage requires one clock cycle
  - An instruction passes through the stages sequentially
  - Without pipelining, a new instruction is fetched at stage 1 only after the previous instruction finished at stage 5 $\Rightarrow$ #clock cycles = 5
- With pipelining the above gives CPI = 1
  - New instruction is fetched at every clock cycle
  - It is "five times faster than without pipelining" $\Rightarrow$ CPI = 1 in this case

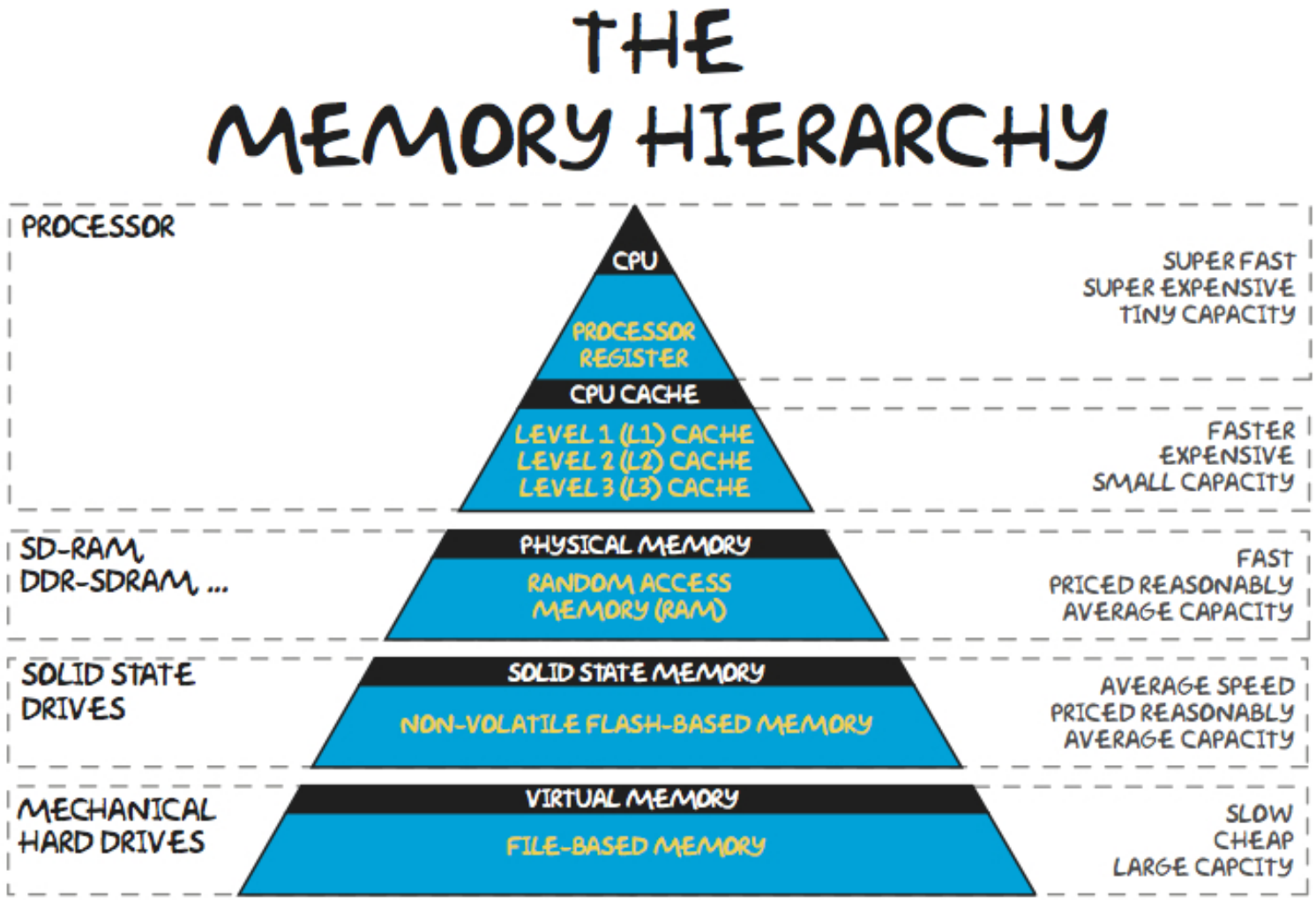
Instruction level parallelism



- ILP (Instruction Level Parallelism)
- Execute several instructions at the same time (subscalar)
- E.g., 2 execution units gives CPI (cycles per instruction) $= 1/2$

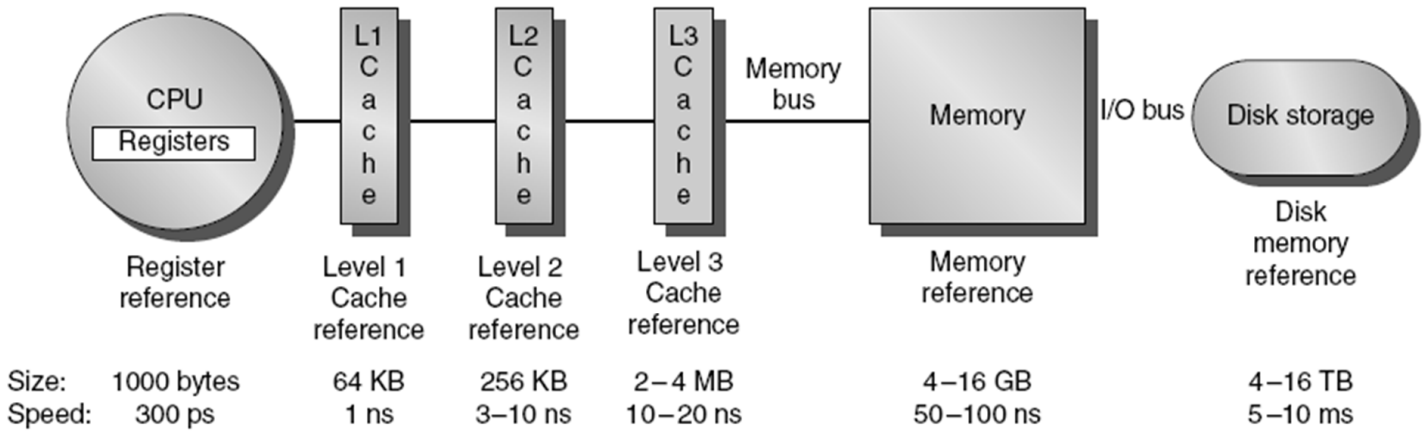
Memory hierarchies

On my Mac Book Pro: 32KB L1 Cache, 256KB L2 Cache, 3MB Cache, 8GB RAM

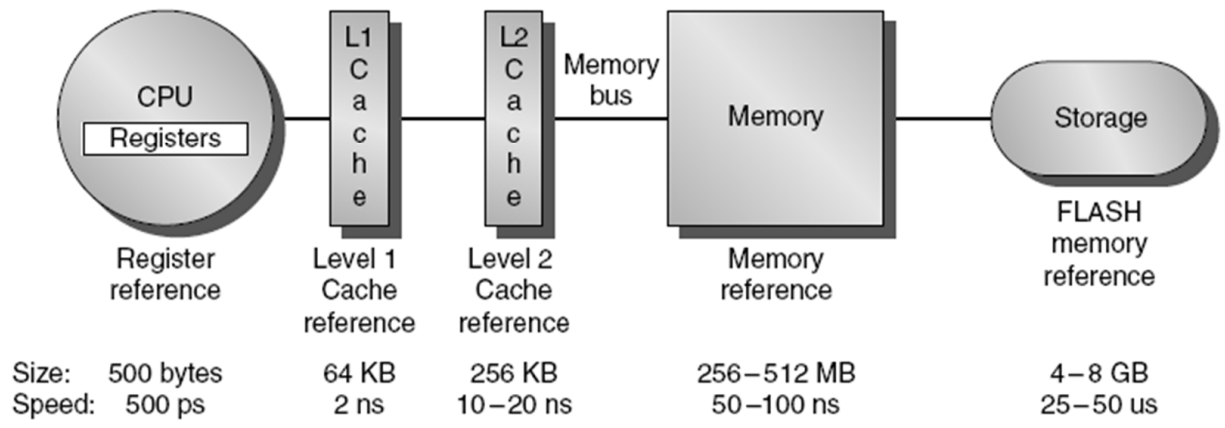


CPU: $\mathcal{O}(1\text{ns})$, L2/L3: $\mathcal{O}(10\text{ns})$, RAM: $\mathcal{O}(100\text{ns})$, disc: $\mathcal{O}(10\text{ms})$

Levels of Memory Hierarchy



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Memory hierarchies

Computer architecture is complicated. We need a basic performance model.

- Processor needs to be "fed" with data to work on.
- Memory access is slow; memory hierarchies help.
- This is a single processor issue, but it's even more important on parallel computers.

More CS terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);

Bandwidth grows faster than latency.

- Eliminate memory operations by saving data in fast memory and reusing them, i.e., temporal locality: Access an item that was previously accessed
- Explore bandwidth by moving a chunk of data into the fast memory: spatial locality: Access data nearby previous accesses
- Overlap computation and memory access (pre-fetching; mostly figured out by compiler, but the compiler often needs help)

More CS terms:

- cache-hit: required data is available in cache $\Rightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Rightarrow$ slow access

Memory hierarchy

Simple model

1. Only consider one level in hierarchy, slow (RAM) memory
2. All data is initially in slow memory
3. Simplifications:
   - Ignore that memory access and arithmetic operations can happen at the same time
4. Computational intensity: flops per slow memory access

$$q = \frac{f}{m}, \text{ where } f \ldots \#\text{flops}, m \ldots \#\text{slow memop}.$$

Actual compute time:

$$ft_f + mt_m = ft_f(1 + \frac{t_m}{t_f}\frac{1}{q}),$$

where t_f is time per flop, and t_m the time per slow memory access.
Computational intensity should be as large as possible.

Matrix-vector multiplication

Matrix-vector multiplication: $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n}$

$$\boldsymbol{y} = \boldsymbol{y} + A\boldsymbol{x}$$

```
for i=1:n
    // load y(i)
    for j=1:n
        // load A(i, j), x(j)
        y(i) = y(i) + A(i, j)*x(j);
    end
    // store y(i)
end
```

Matrix-vector multiplication

Matrix-vector multiplication: $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n}$

$$\boldsymbol{y} = \boldsymbol{y} + A\boldsymbol{x}$$

```
for i=1:n
    // load y(i)
    for j=1:n
        // load A(i, j), x(j)
        y(i) = y(i) + A(i, j)*x(j);
    end
    // store y(i)
end
```

- Number of flops: $2n^2$
- Number of memory read/writes: $2n + 2n^2$

Computational intensity: ~ 1 (memory-bound!)

Matrix-matrix multiplication

```
// To compute C = C + A*B
for i=1:n
    for j=1:n
        // load C(i, j)
        for k=1:n
            // load A(i, j), B(k, j)
            C(i, j) = C(i, j) + A(i, k)*B(k, j);
        end
        // store C(i, j)
    end
end
```

Matrix-matrix multiplication

```
// To compute C = C + A*B
for i=1:n
    for j=1:n
        // load C(i, j)
        for k=1:n
            // load A(i, j), B(k, j)
            C(i, j) = C(i, j) + A(i, k)*B(k, j);
        end
        // store C(i, j)
    end
end
```

- Number of flops: $2n^3$
- Number of memory read/writes: $2n^2 + 2n^3$

Computational intensity: $\sim$1 (memory-bound!)

Can this be improved using fast memory? Yes! (Sketch—will be part of the next homework).

1. Consider two levels in hierarchy, fast and slow (RAM) memory
2. All data is initially in slow memory
3. Simplifications:
   - Ignore that memory access and arithmetic operations can happen at the same time
   - Assume time for access to fast memory is 0

Matrix-matrix multiplication with blocking

```
for i = 1:N
    for j = 1:N
        \\ Make index ranges associated with
        \\ blocks
        I = ((i-1)*N+1):i*N;
        J = ((j-1)*N+1):j*N;
        \\ load block C(I,J) into fast memory
        for k = 1:N
            \\ load blocks A(I,K) and B(K,J)
            \\ into fast memory
            K = ((k-1)*N+1):k*N;
            C(I,J) = C(I,J) + A(I,K)*B(K,J);
        end
        \\ store block C(I,J)
    end
end
```

- Matrix-matrix multiplication with blocking: $A, B, C \in \mathbb{R}^{n \times n}$

$$C = C + AB$$

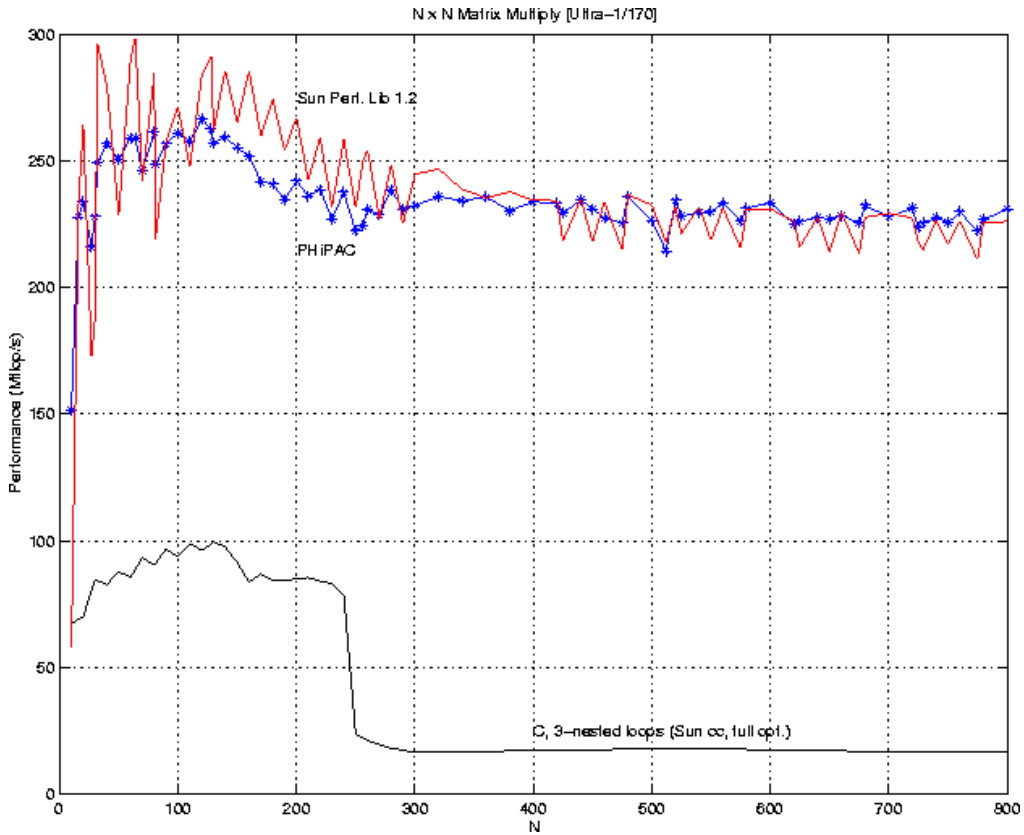
We'll consider $N \times N$ blocks of size $b \times b$ of the matrices, i.e., $b = n/N$.

- $2N^3$ block reads of A, B; $2N^2$ reads/writes of C;
- memory access: $(2N^3 + 2N^2)b^2 \sim 2Nn^2 + 2n^2$
- Computational intensity $q \sim b = n/N$!
- Gives a much higher computational intensity, much faster!

Memory hierarchy

Example: Matrix-matrix multiply

Comparison between naive and blocked optimized matrix-matrix multiplication for different matrix sizes.



Comparison between optimized and naive matrix-matrix multiplication on old hardware with peak of 330MFlops.
Source: J. Demmel, Berkeley

BLAS: Optimized Basic Linear Algebra Subprograms

Memory hierarchy

To summarize:

- Temporal and spatial locality is key for fast performance.
- Simple performance model: fast and slow memory; only counts loads into fast memory; computational intensity should be high.
- Since arithmetic is cheap compared to memory access, one can consider making extra flops if it reduces the memory access.
- In distributed-memory parallel computations, the memory hierarchy is extended to data stored on other processors, which is only available through communication over the network.

Cache coherence

Certain features of parallel computers have a large influence on the behaviour of parallel programs.

This behaviour is often not directly visible to the programmer, especially, since all communication between different processors are performed using shared memory and the hardware providing the shared memory.

Such problems appear either directly due to hardware, especially memory caches, e.g.

- False Sharing and
- Atomic Operations

or indirectly by software, e.g.

- Thread Scheduling or
- Memory Allocation.

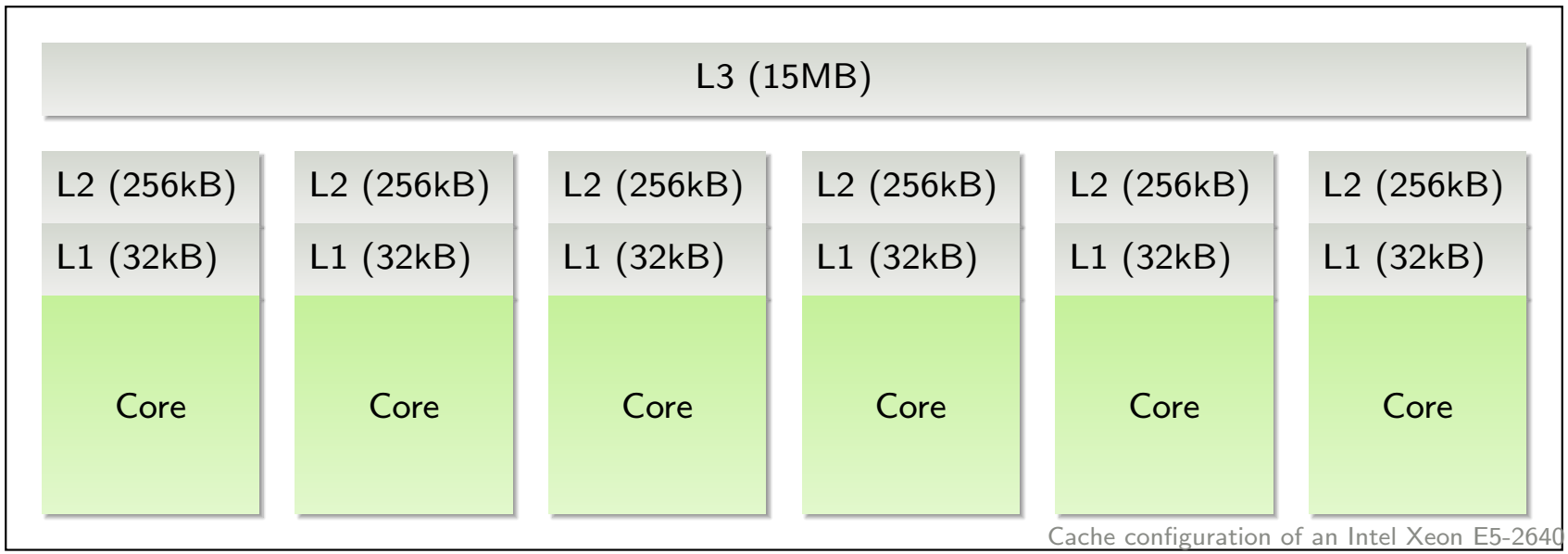
The most notable software in this context is of course the *Operating System*, providing access to the hardware.

Cache coherence

A typical parallel computer today consists of one or more processors with several cores, each having

- first level cache (instruction and data) in the order of 16kB to 128kB,
- second level cache of size 256kB up to some MB,
- optional third level cache up to several MB capacity

The 2nd and 3rd level caches may be shared between different cores.



Cache configuration of an Intel Xeon E5-2640

Cache coherence

Each cache may have its own copy of data from the main memory. If the data is changed in one place, all copies of that data have to be updated.

Cache coherence exists, if all caches contain the most recent version of some shared data from the main memory.

Various algorithm are used to maintain cache coherence in modern processors.

A typical cache coherence protocol will

- listen for memory accesses of all other cores,
- if a write to a locally cached memory position is detected, *invalidates* the local cache entry,
- which enforces a reload of the data from memory.

False sharing

Processors will *not* map individual bytes into local caches, but handle memory in segments of size 64–128 bytes, so called *cache lines*.

If a single byte has changed in memory, not only the entry of this single data item is invalidated in the cache but the whole cache line, leading to a memory read of size 64–128 bytes.

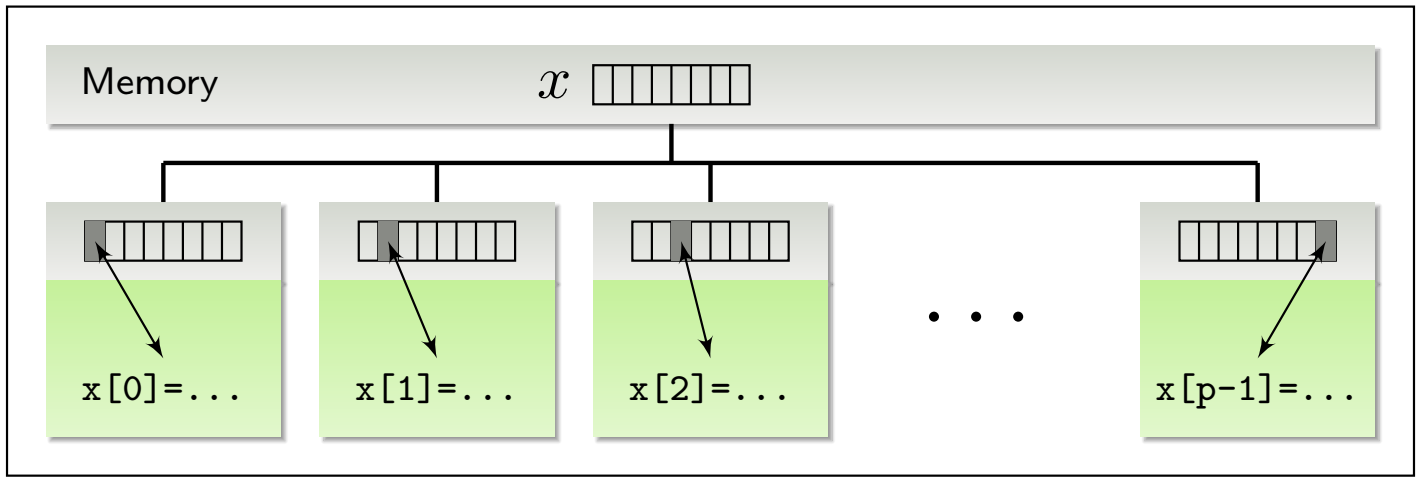
Consider the following example where eight threads will compute a separate coefficient of an array.

```
int main () {
  double                          x[8];
  std::vector< std::thread >      threads( 8 );

  for ( int i = 0; i < 8; ++i )
    threads[i] = std::thread( f, i, x );

  ...
}
```

```
void f ( int tid, double * x ) {
  do {
    x[tid] = compute_update();
  } while ( ... );
}
```

Since each thread will update a private entry of the array `x`, no shared data and no critical region exist.

Unfortunately, `x` is 64 bytes long ($8 \cdot$`sizeof(double)` $= 8 \cdot 8$), and hence, may occupy an entire cache line. Therefore, all processors will hold the complete content of x in their local caches:



If thread i updates the content of the coefficient x_i, this invalidates the cache line of x_i and therefore x in all other processors.

Hence, for each update of a coefficient of x, all processors need to load the memory corresponding to the whole array x into the local caches.

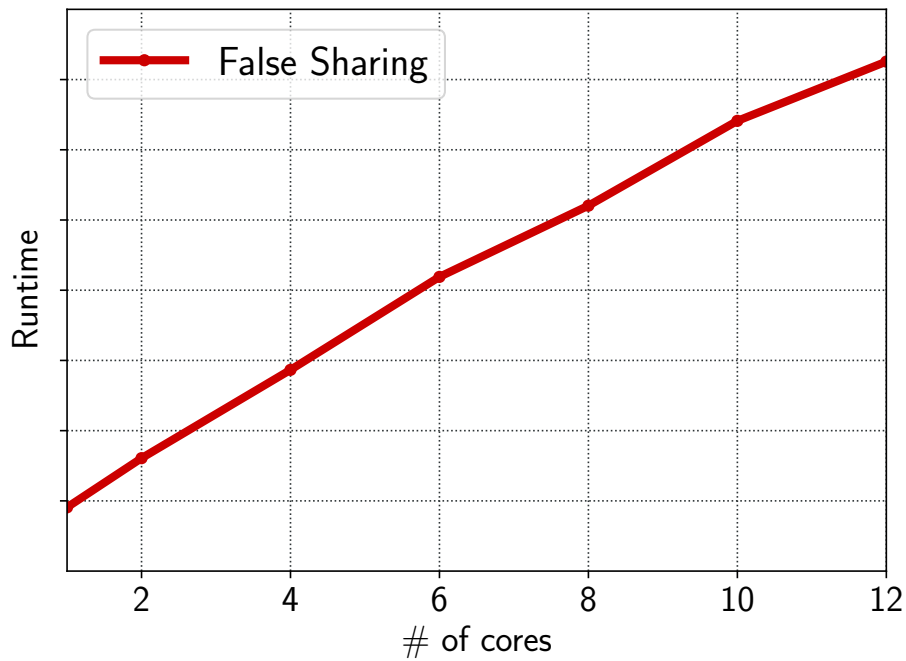
The behavious is known as *false sharing* and is an example of a hidden task interaction in multi-threaded programs.

If the updates to x are fast enough, the memory loads will dominate the computation and severly limit the parallel efficiency. In extreme cases, the parallel runtime may be higher than the sequential runtime!

In the following example, each thread will updated its local coefficient of x via

```
void f ( int tid, double * x ) {
  for ( size_t i = 0; i < 10000000; ++i )
    x[tid] += std::sin( double(i) );
}
```

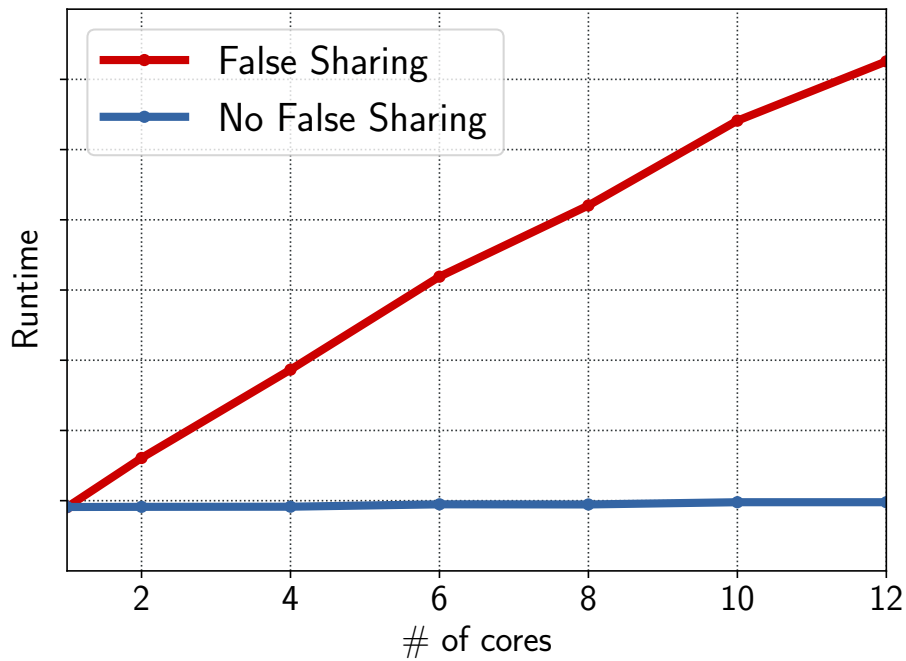
The resulting runtime grows linear with # threads p!



If instead the update is applied to a thread-private variable:

```
void f ( int tid, double * x ) {
  double  t = 0.0;

  for ( size_t i = 0; i < 10000000; ++i )
    t += std::sin( double(i) );

  x[tid] += t;
}
```

the runtime stays constant.

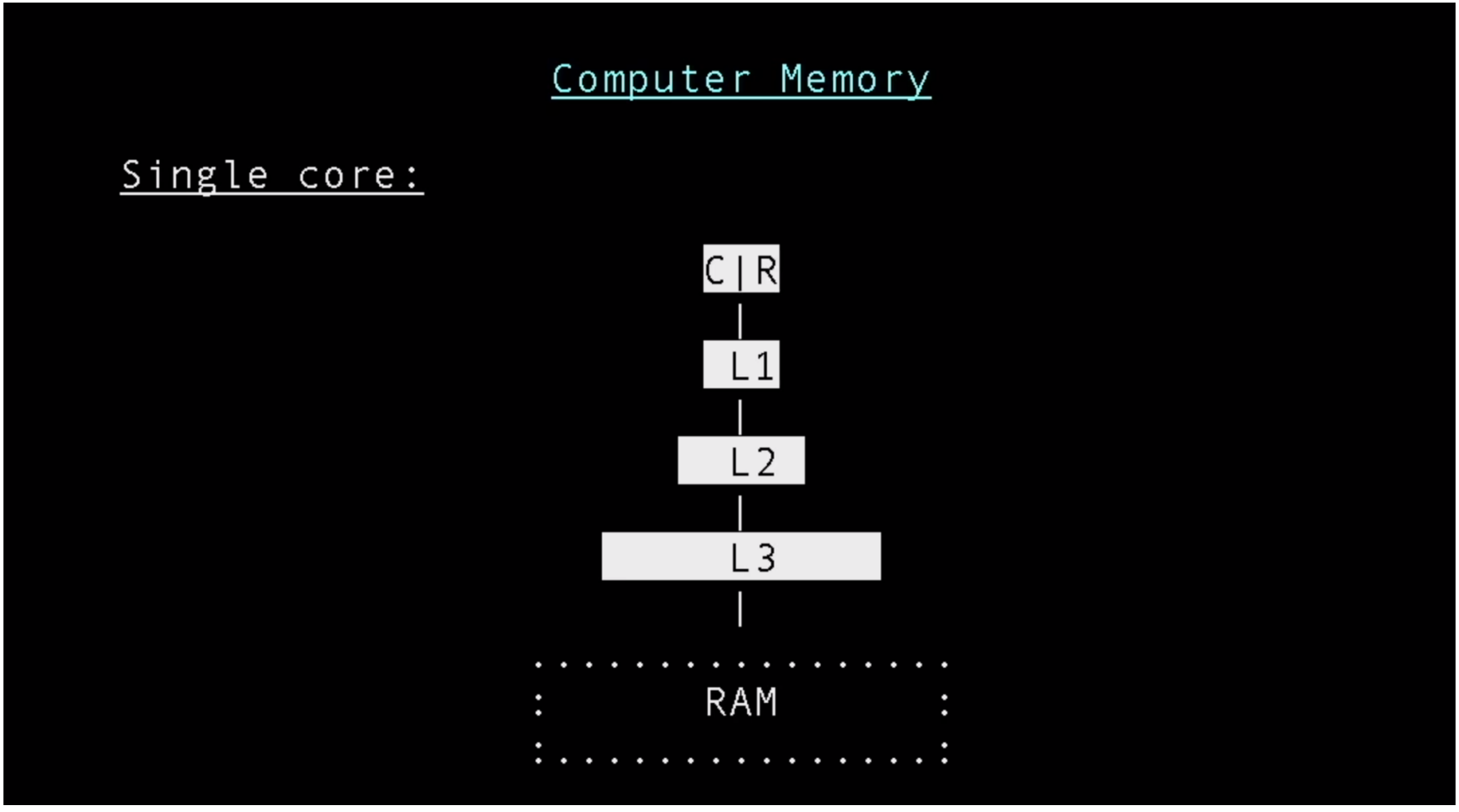


- Note that runtime is constant because each thread iterates until 10^7 (no parfor)

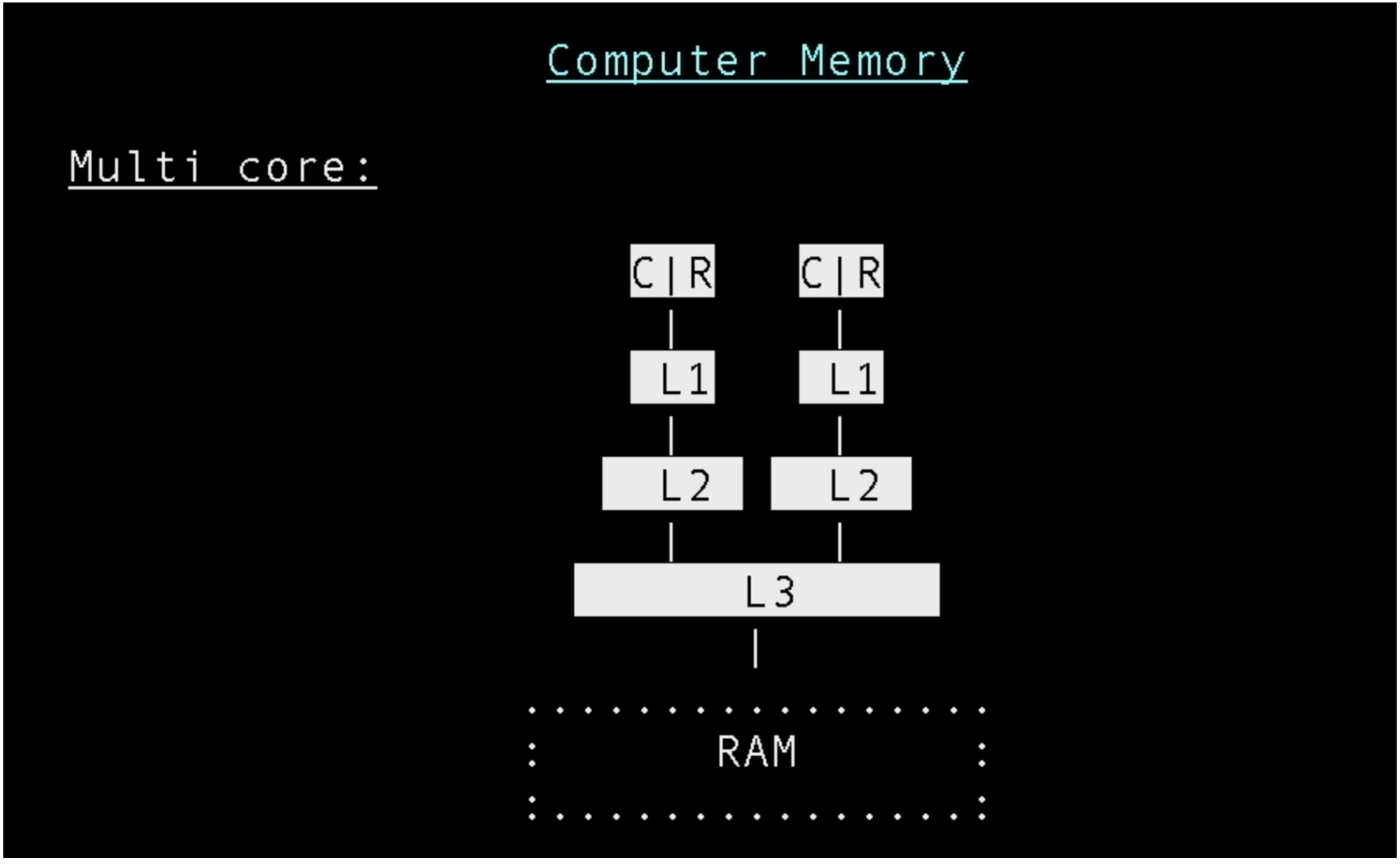
Even for a larger data set, false sharing may be an issue at the per-task data boundaries:

If the computation is *concentrated* at these boundaries, memory loads will be induced in the corresponding processor handling the neighboured task.

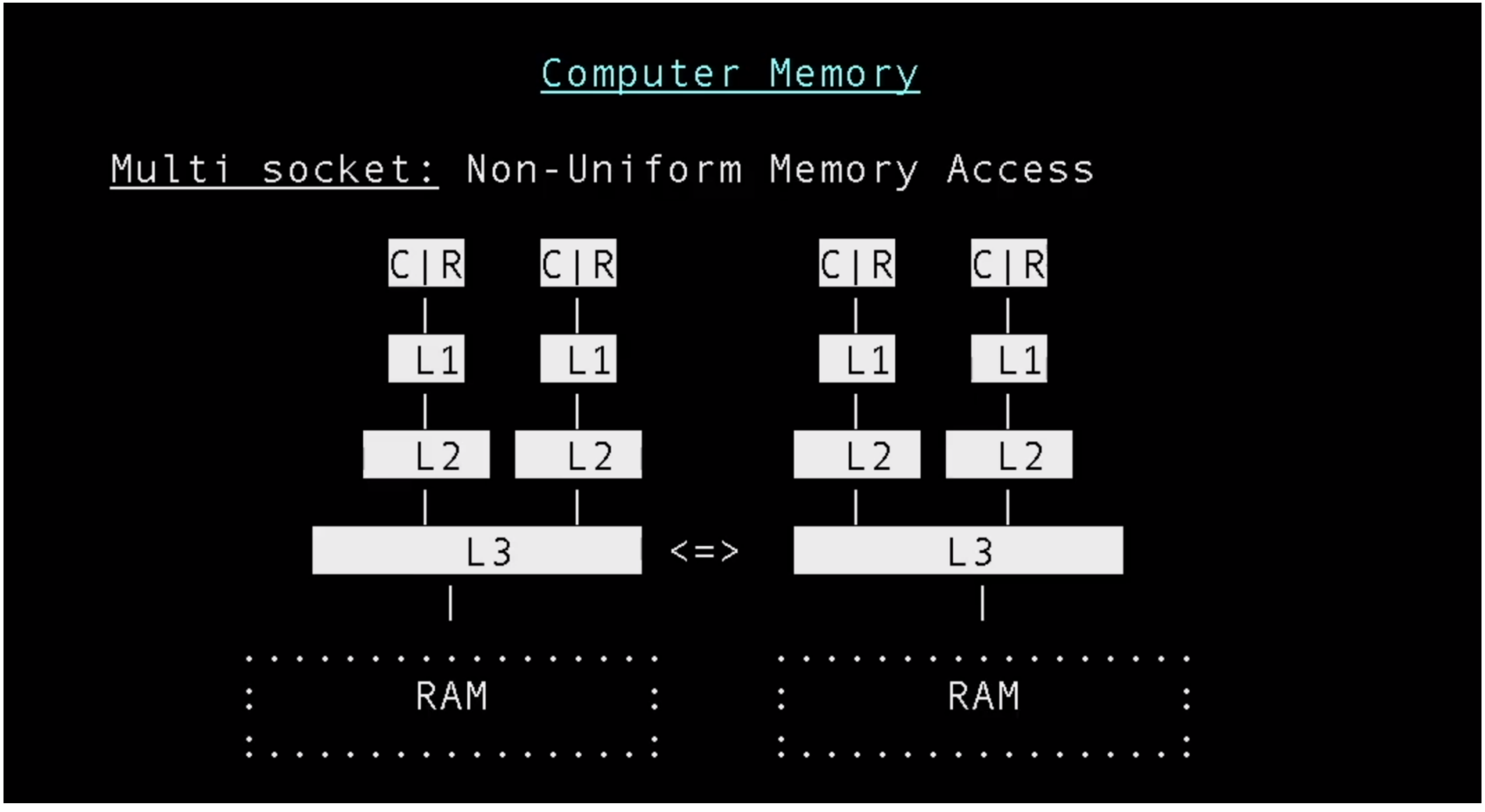
Reminder: Cache hierarchy



Reminder: Cache hierarchy (cont'd)



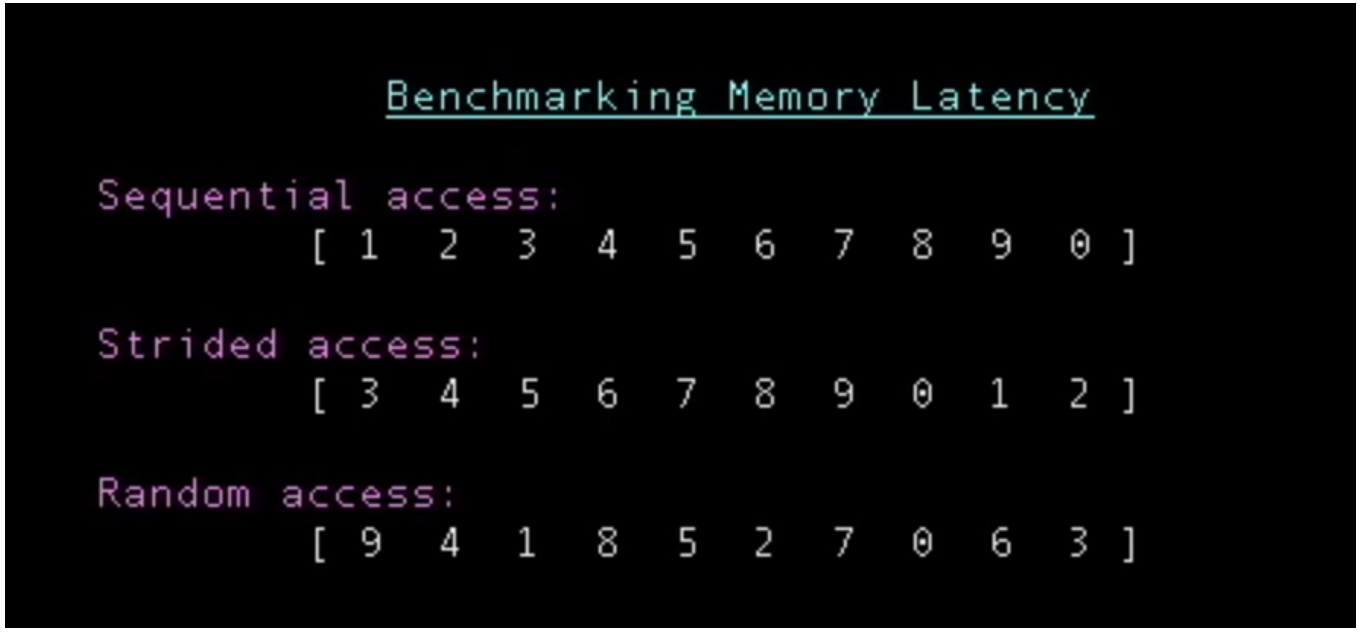
Reminder: Cache hierarchy (cont'd)



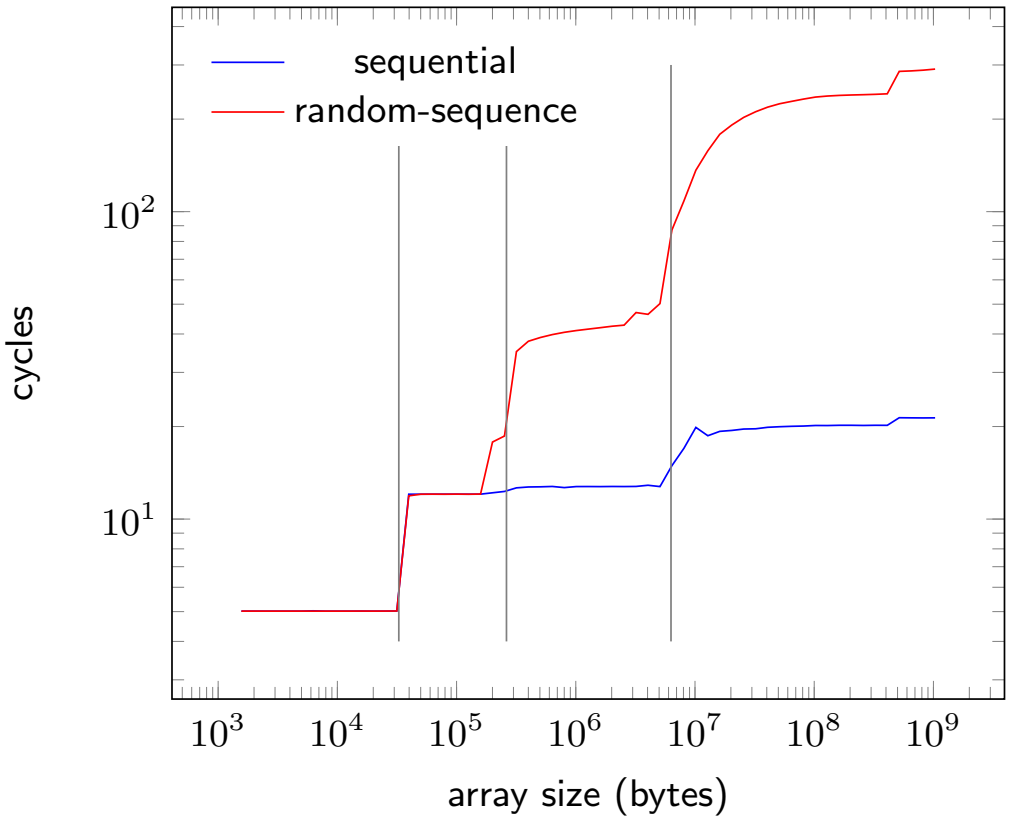
Get cache size

```
pehersto@plab01:~$ getconf -a | grep CACHE_SIZE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_DCACHE_SIZE                 32768
LEVEL2_CACHE_SIZE                  1048576
LEVEL3_CACHE_SIZE                  11534336
LEVEL4_CACHE_SIZE                  0
```

Code: latency.cpp



Latency



- Pre-fetching can significantly improve latency
- For pre-fetching one needs to be able to predict what is read next, which is impossible with random access

Valgrind and cachegrind

Valgrind

- memory management tool and suite for debugging, also in parallel
- profiles heap (not stack) memory access
- simulates a CPU in software
- running code with valgrind makes it slower by factor of 10-100
- Documentation: http://valgrind.org/docs/manual/

memcheck

finds leaks
inval. mem. access
uninitialize mem.
incorrect mem. frees

cachegrind

cache profiler
sources of cache misses

callgrind

extension to cachegrind
function call graph

Valgrind and cachegrind

Usage (see examples):

Run with valgrind (no recompile necessary!)

```
valgrind --tool=memcheck [options] ./a.out [args]
```

Valgrind and cachegrind

Run cachegrind profiler:

```
valgrind --tool=cachegrind [options] ./a.out [args]
```

Visualize results of cachegrind:

```
cg_annotate --auto=yes cachegrind.out***
```

```
valgrind --tool=cachegrind
        ./inner-mem vec_size
```