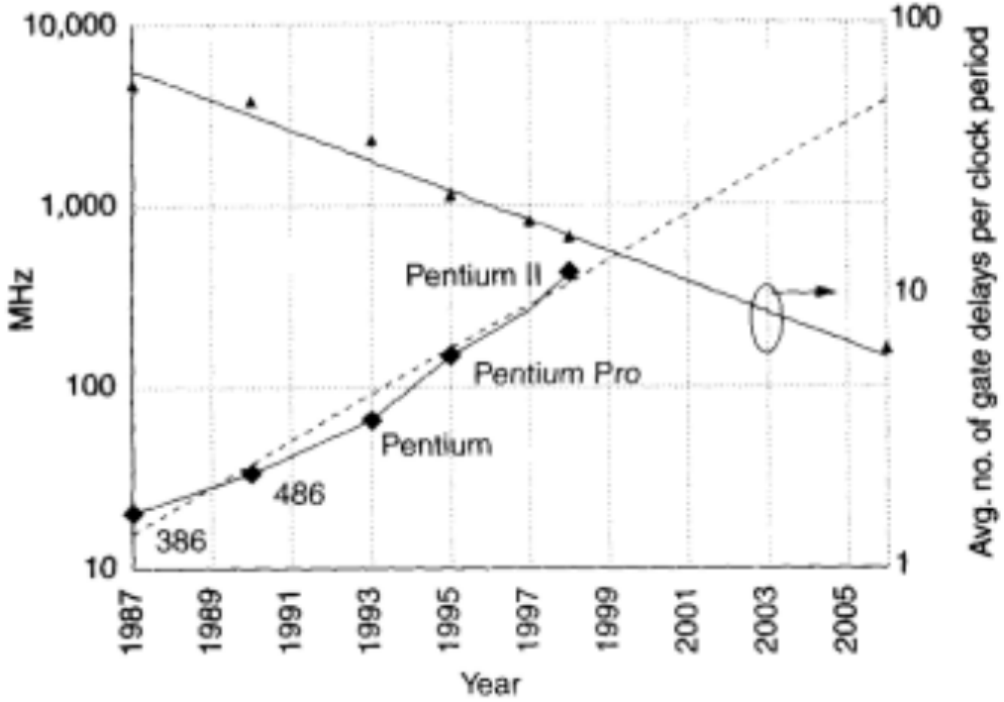
Computer architecture evolution and the performance challenge

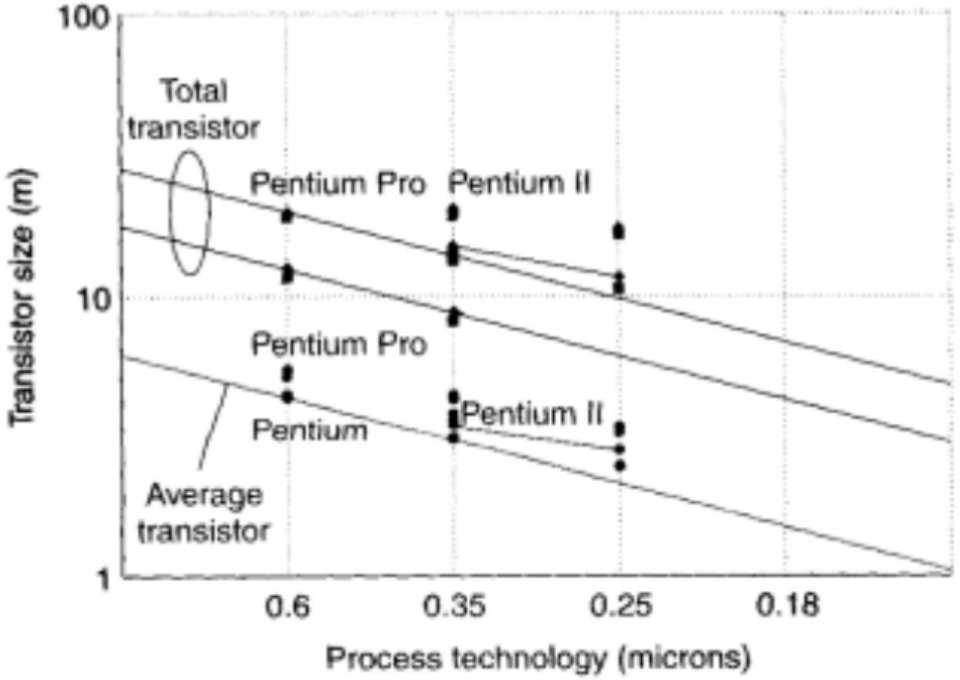
Felice Pantaleo

CERN Experimental Physics Department

felice@cern.ch

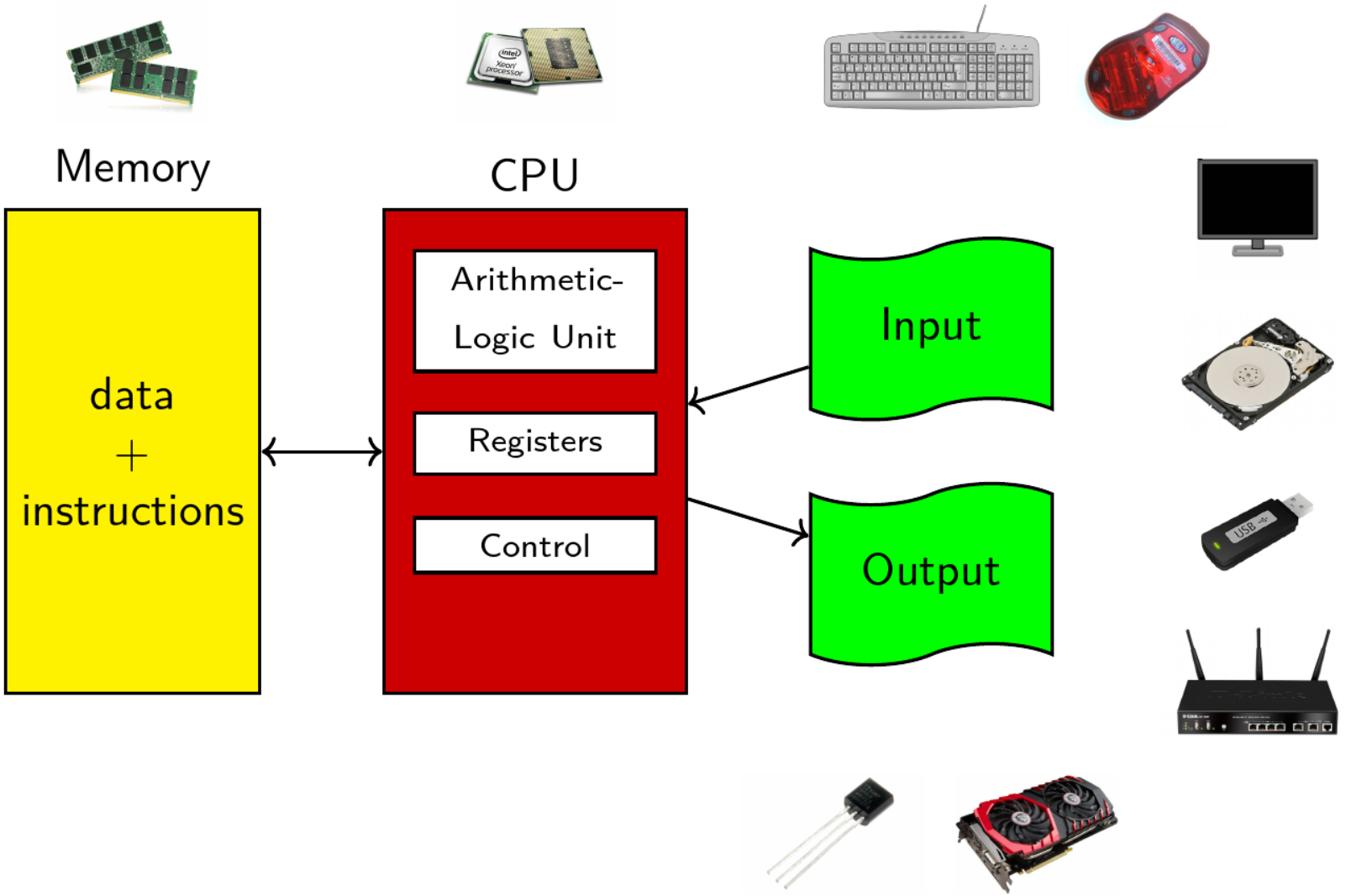
Previously, in Moore's Paradise





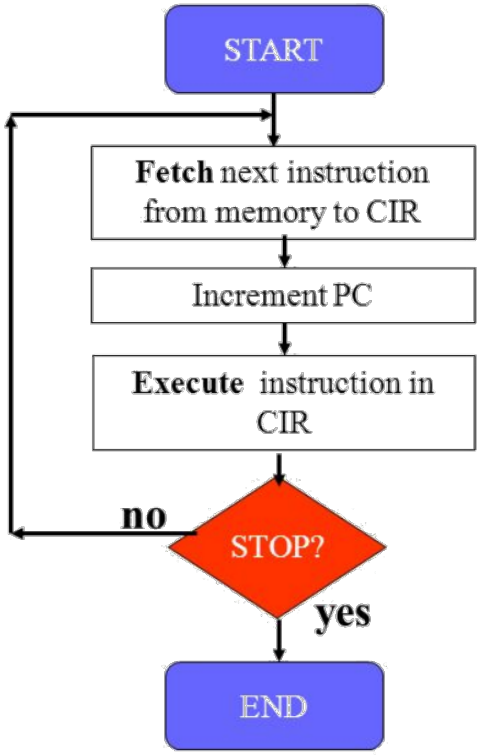
- The main contribution to the gain in microprocessor performance at this stage came by increasing the clock frequency.
- Applications' performance doubled every 18 months without having to redesign the software or changing the source code

Von Neumann Architecture

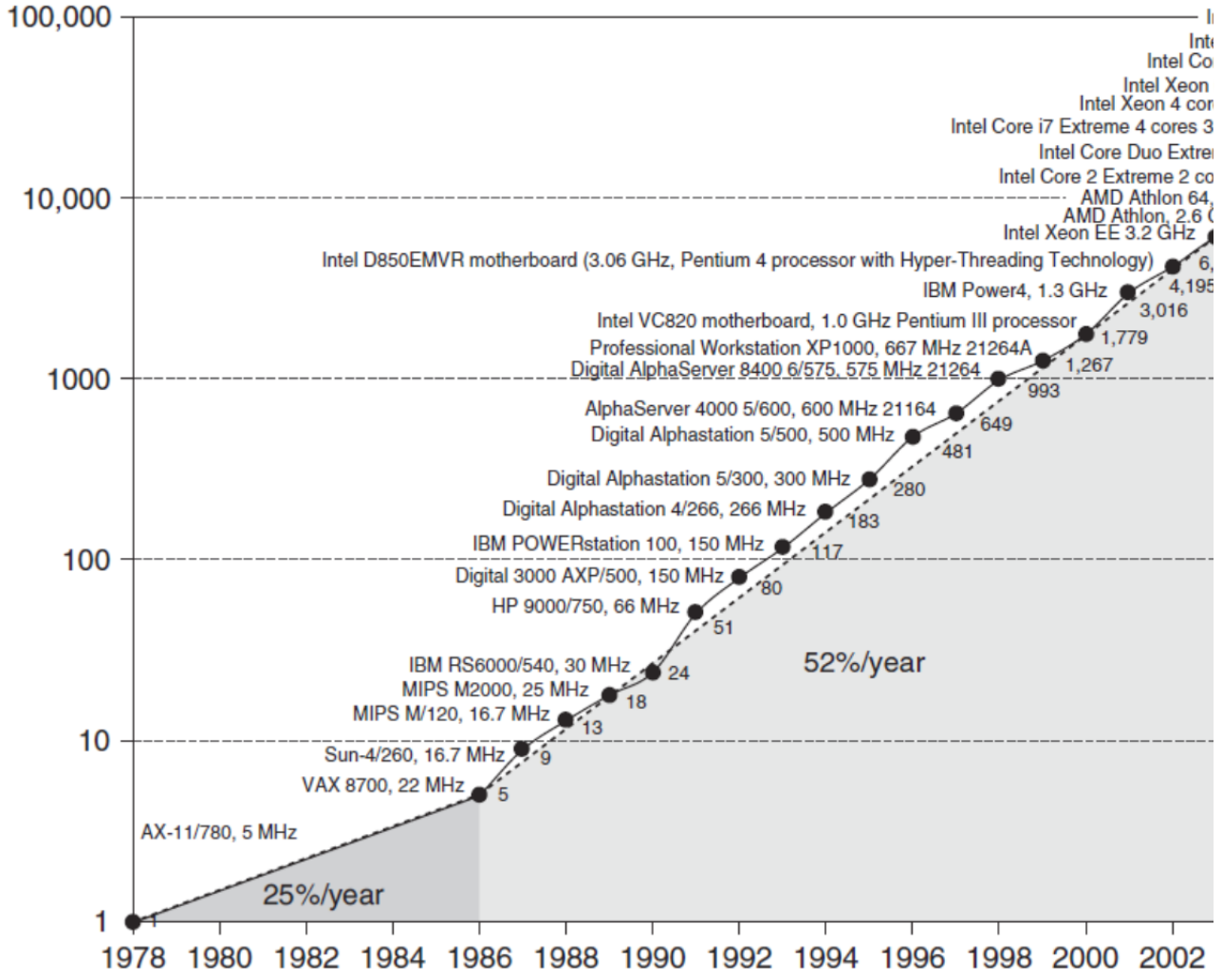


Von Neumann Architecture

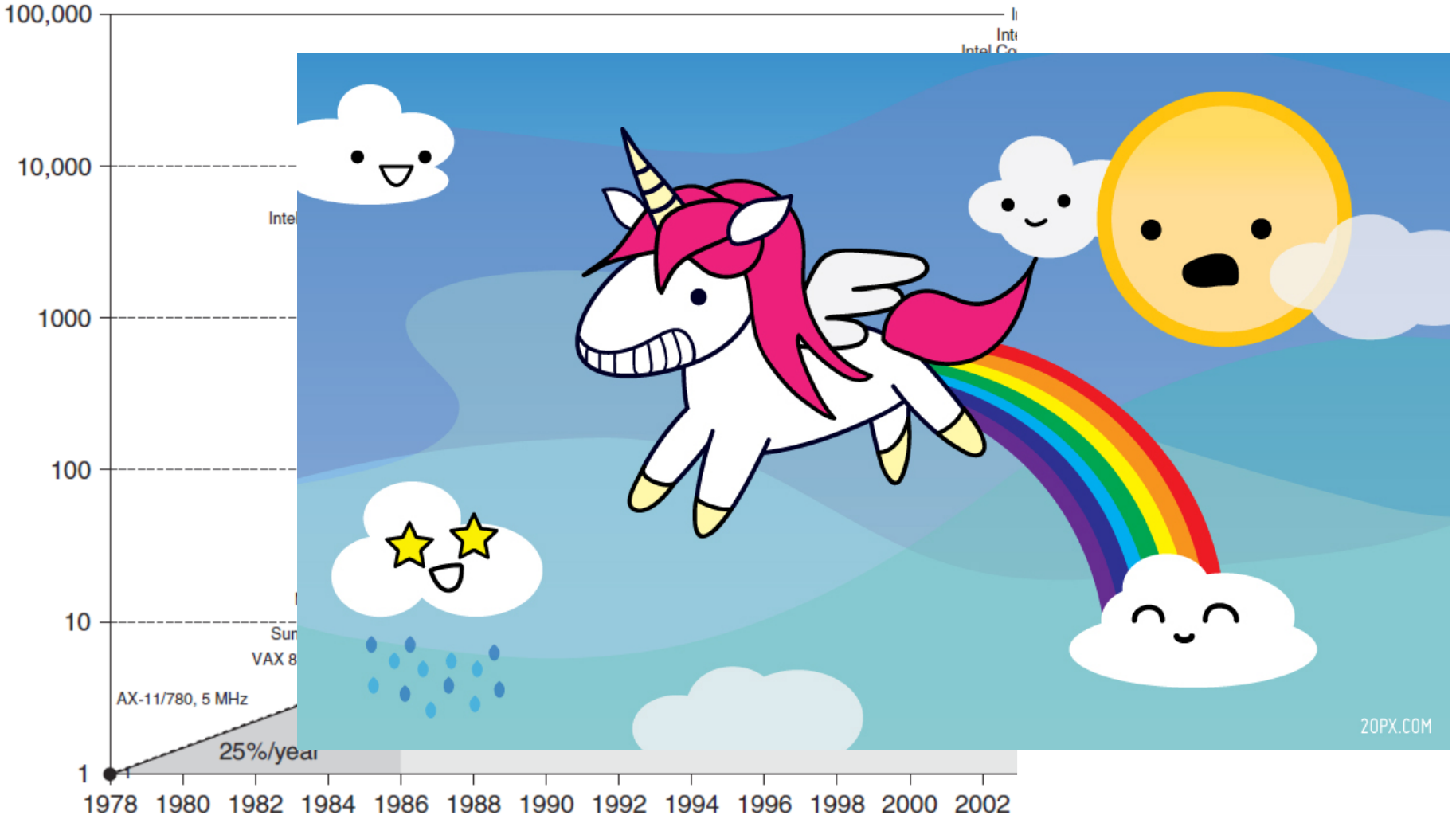
- The basic operation that every Processing Unit (PU) has to process is called instruction and the address in memory containing the instruction is saved
- A *Program Counter* (PC) holds the address of the next instruction
- *fetch*: the content of the memory stored at the address pointed by the PC is loaded in the Current Instruction Register (CIR) and the PC is increased to point to the next instruction's address
- *decode*: the content of the CIR is interpreted to determine the actions that need to be performed
- *execute*: an Arithmetic Logic Unit performs the decoded actions.



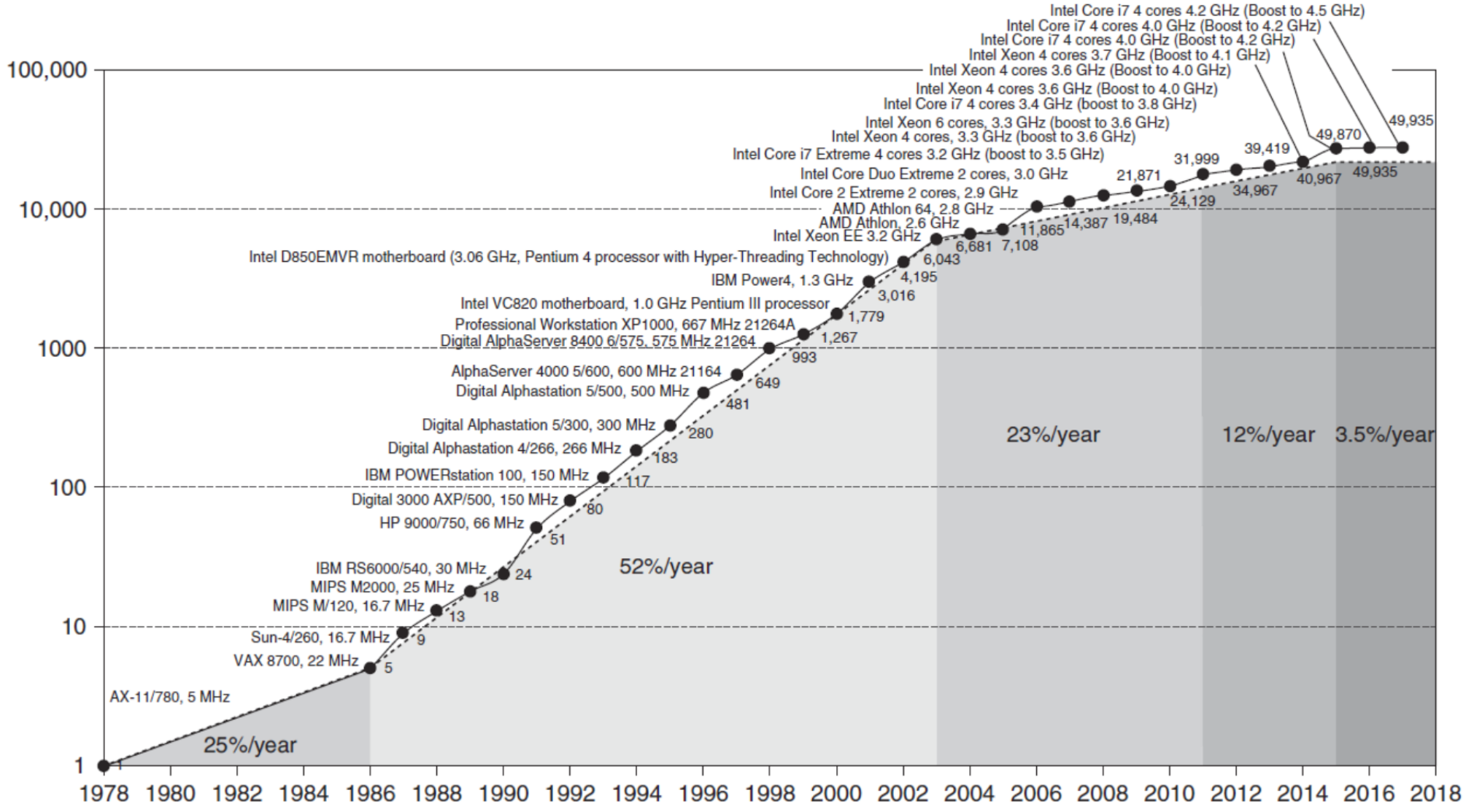
Moore's Law (ctd.)



Moore's Law (ctd.)



Moore's Law (ctd.)



Back on Earth

- The power dissipated by a processor scales as

$$P = QCV^2 f + V I_{\text{leakage}}$$

- Q number of transistors
- C capacity
- V voltage across the gate
- f the clock frequency
- I current
- In the early 2000s, the layer of silicon dioxide insulating the transistor's gate from the channels through which current flows was just five atoms thick and could not be shrunk anymore

Power and Energy

Thermal Design Power (TDP)

- Characterizes sustained power consumption
- Used as target for power supply and cooling system
- Lower than peak power (usually 1.5X higher), higher than average power consumption

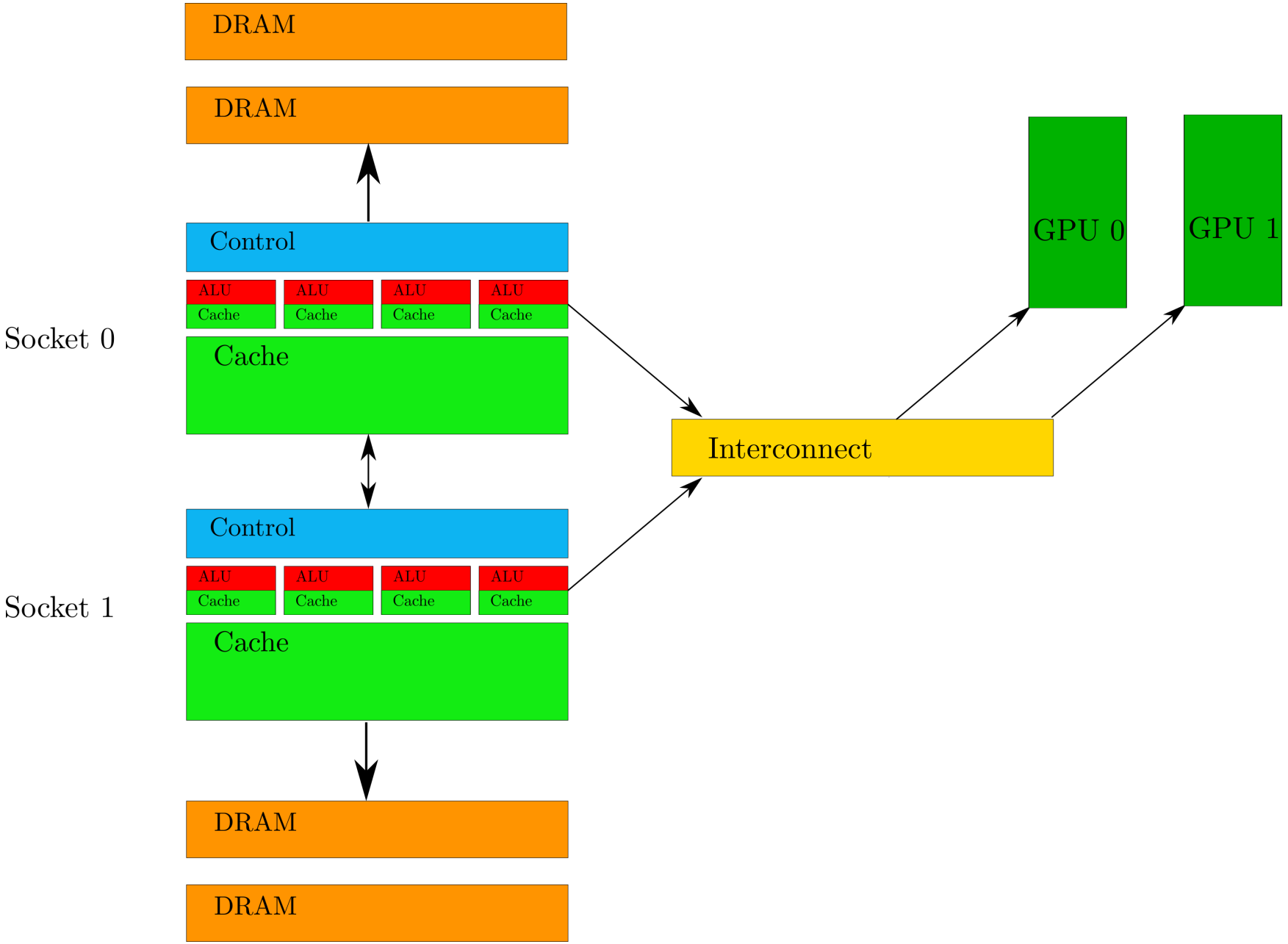
- Clock rate can be reduced dynamically to limit power consumption
- Energy per task is often a better measurement

"The party isn't exactly over, but the police have arrived, and the music has been turned way down" (P. Kogge, IBM)

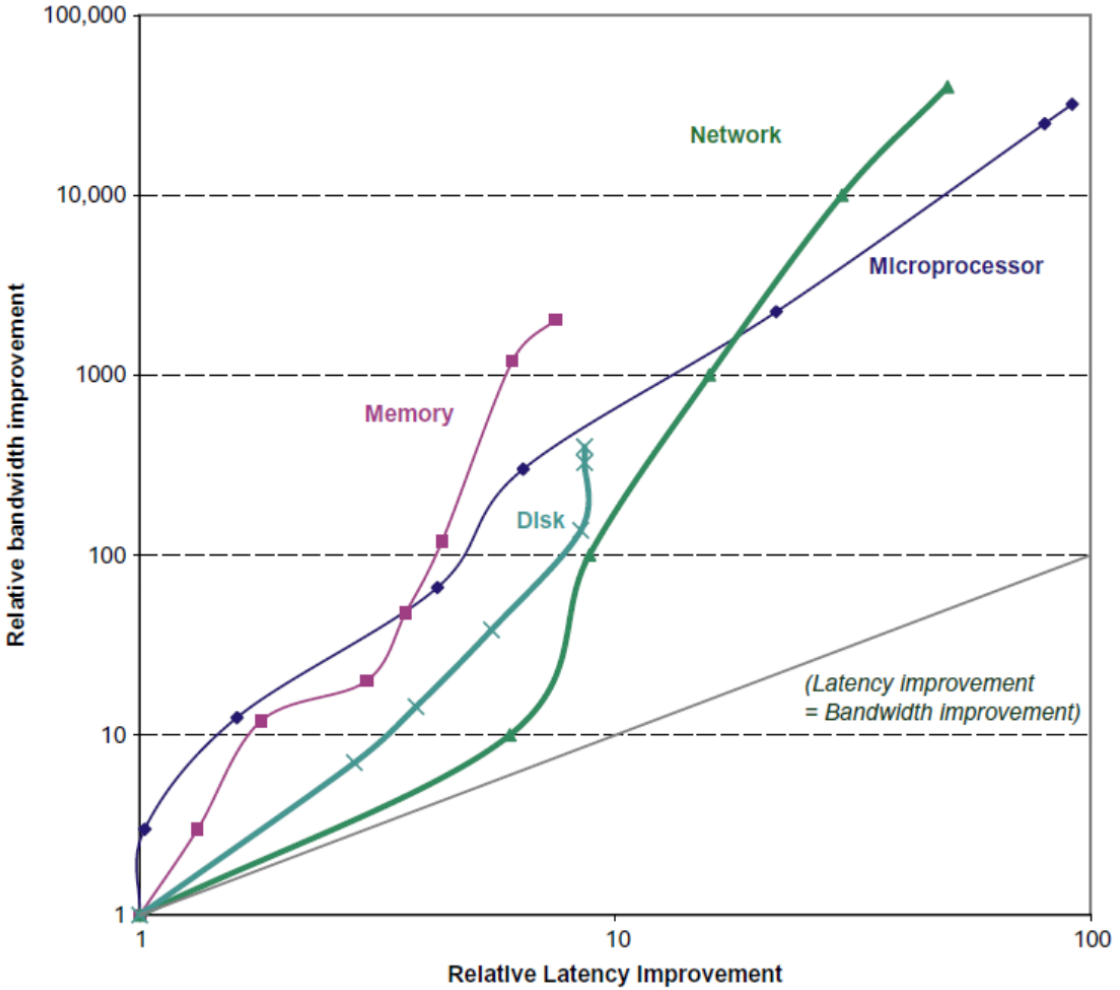
Evolution of system architecture

- Increased number of Processing Units
- More complex control
  - Pipelining
  - hardware threading
  - out-of-order execution
  - instruction-level parallelism
- Deeper memory hierarchy
- Accelerators
- Interconnects

What you will master soon

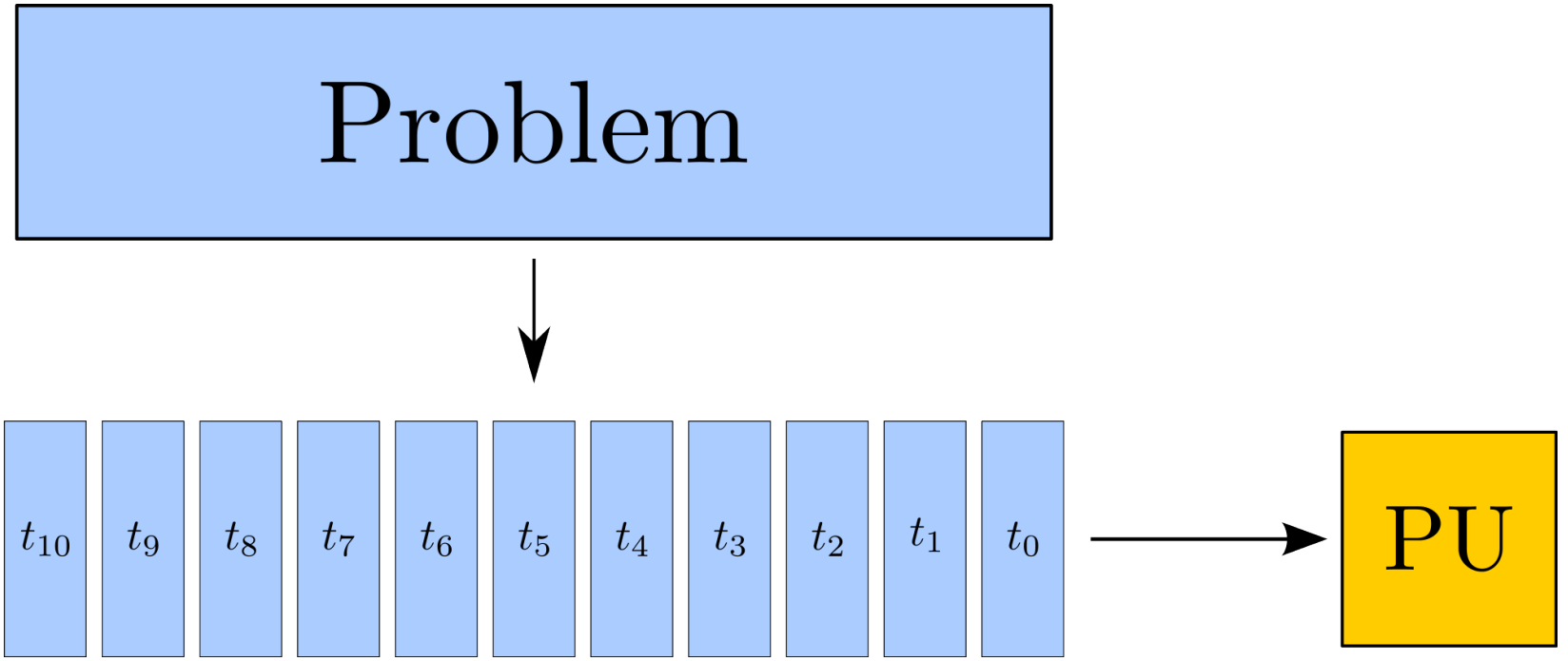


Latency vs Bandwidth



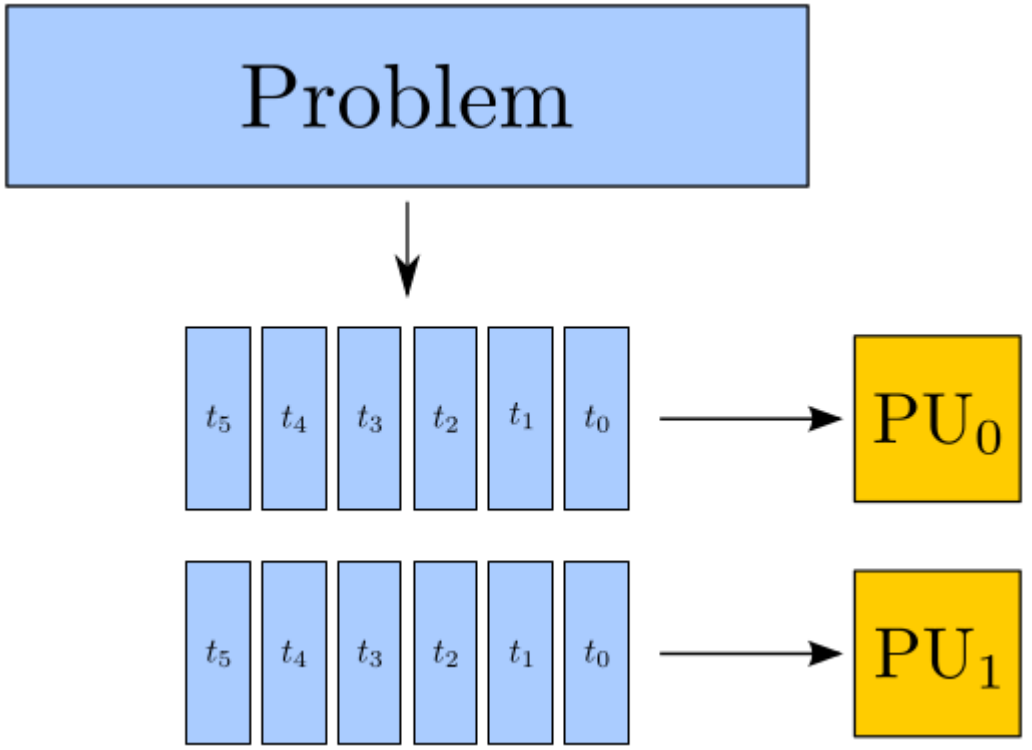
Serial computation

- Software traditionally written for serial computation:
- the sequence of instructions that forms the problem is executed by one Processing Unit (PU)
- every instruction has to wait for the previous one to be completed before its execution can start
- at any moment in time, only one instruction may execute



Parallel computation

- In parallel computation, if two instructions have no data dependency, they can be executed in parallel, at the same time, by two PUs



Pizza Wall

- How many cooks does a pizzeria need to achieve the best production rate possible?
- If all the ingredients are in the same fridge and there is only one oven? Maybe 1, 2, 64, infinity?

Mitigating the Pizza Wall

- Reuse of ingredients and tools which are used often: put them on a small table close to you
- Increase the frequency of travels to the fridge
- Increase the amount of ingredients you transfer from the fridge
- If ingredients are located all in the same box in the fridge, you can carry more of them with a single transfer
- Better organization of order of instructions, keeping cooks busy

Memory Wall

- How many PUs does a program need to achieve the best performance possible?

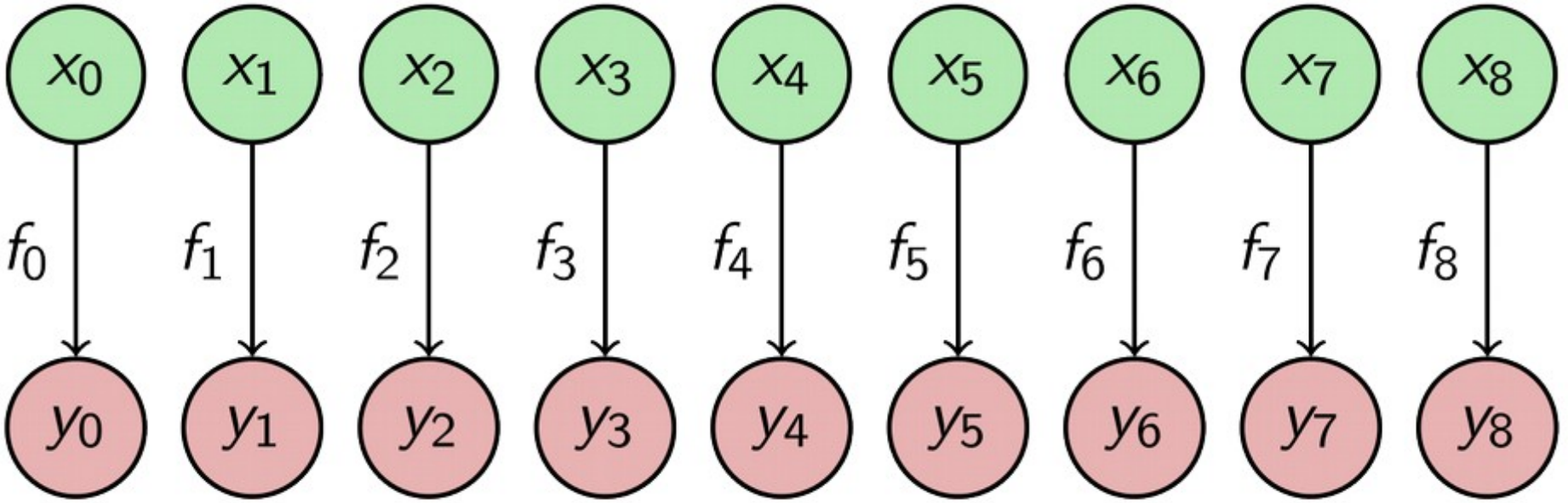
Mitigating the Memory Wall

- Reuse data and instructions: data and instructions which are used often are stored in a on-chip memory called cache.
- Increase the memory transfer speed: this can be done by increasing frequency, which is limited by the power wall.
- Increase the amount of data to transfer: memory transfers have overheads, which can become negligible if more memory is transferred in one instruction.
- Improve the access pattern to memory: if more processing units are reading adjacent memory locations, they can all be fed by a single memory transfer.
- Better organization of order of instructions, keeping PU busy
- Smarter prefetching

Parallel Computing

Embarrassingly parallel problems

$$y_i = f_i(x_i)$$



Embarrassingly parallel problems (ctd.)

Examples:

- Linear Algebra
- Image Processing
- Monte Carlo Simulation
- Cryptomining
- Weather forecast
- Random number generation
- Encryption
- Software compilation

Terminology

- *Granularity*: size of tasks
- *Scheduling*: order of assignment of tasks
- *Mapping*: assignment of tasks to a PU
- *Load balancing:* the art of making the computation of multiple tasks end at the same time
- *Barrier*: a checkpoint at which all the parallel workers should wait for the last one.
- *Speedup*: time of the serial application/time of the parallel application
- *Efficiency*: Speedup/# of PUs
- *Race condition*: When the result of execution depends on sequence

  and/or timing of events. Result could be incorrect if this is not taken in consideration
- *Critical section*: Only one worker per time can enter.

Flynn's Taxonomy

Classification of computers describes four classes in both serial and parallel contexts:

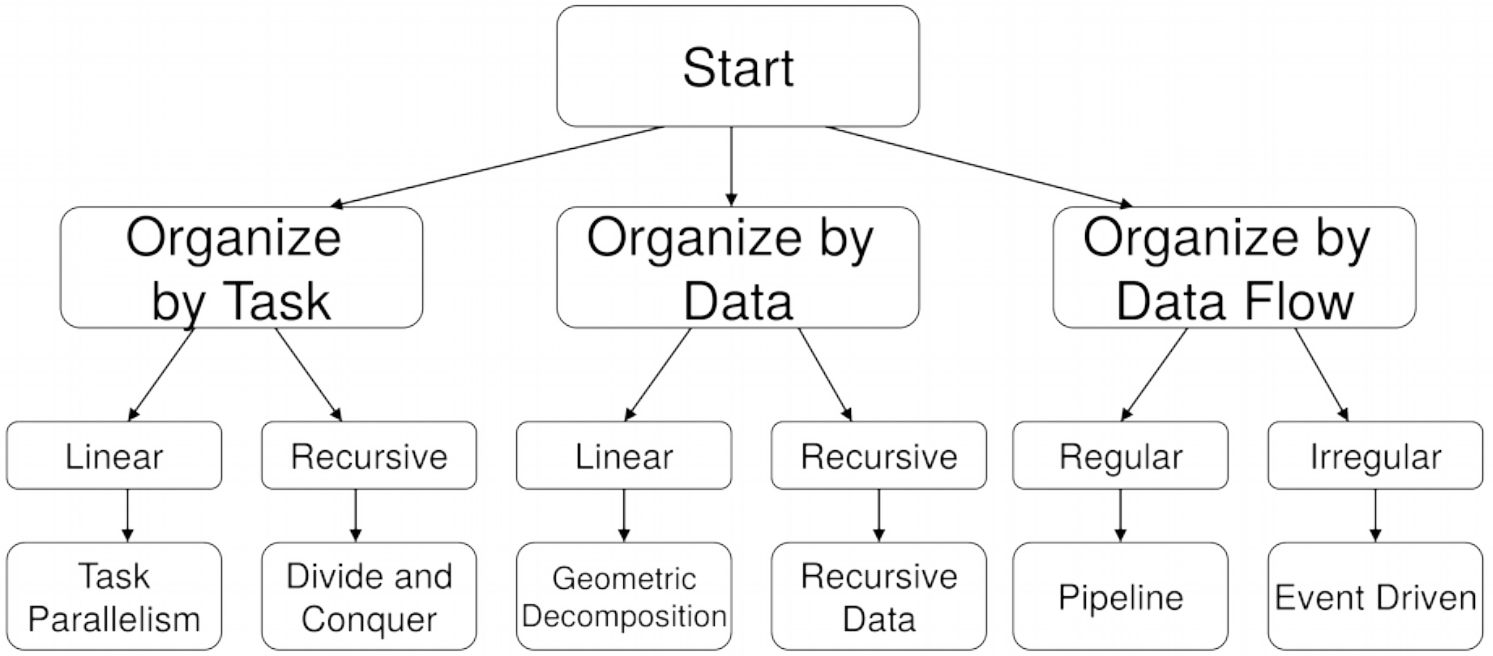
- **SISD** - *Single Instruction stream - Single Data stream*
  - A single processor computer that executes one stream of instructions on one set of data. Single-core processors belong to this class.
- **SIMD** - *Single Instruction Stream - Multiple Data stream*
  - A multiprocessor where each processing unit executes the same instruction stream as the others on its own set of data.
  - A set of processors shares the same control unit, and their execution differs only by the different data elements each processor operates on.

Flynn's Taxonomy (ctd.)

- **MISD** - *Multiple Instruction stream - Single Data stream*
  - Each processing element of the multiprocessor executes its own instructions, but operates on a shared data set.
- **MIMD** - *Multiple Instruction stream - Multiple Data stream*
  - Each processing element executes its own instruction stream on its own set of data.
- **SIMT** - *Single Instruction - Multiple Thread*
  - SIMD is combined with multithreading: we will see this with GPUs

Patterns for Parallel Programming

Parallel programming is not easy:
Apparently simple problems can hide many traps!



Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

Reduce

Reduction is a very common pattern in parallel computing:

- Large input data structure distributed across many PU
- Each PU computes a tally of its input
- These tally values are combined to produce the final result

Examples:

- The sum of the elements of an array
- The maximum/minimum element of an array
- Find the first occurrence of x in an array

count number of 5s

```
array[N]
numberOf5 = 0
for i in [0,N[:
    if array[i] == 5
        numberOf5++
return numberOf5
```

```
numberOf5 = 0
nWorkers = 4
count5(array, workerId):
  beg = workerId*N/nWorkers
  end = beg + N/nWorkers
  for i in [beg,end[:
      if array[i] == 5:
          numberOf5++
```

Data Hazards

Threads within a process share the same address space but not their execution stack

Pro: Threads can communicate using shared memory

Cons: Data Hazards if threads are not synchronized

Data hazards usually occur when threads modify data in different points in the instruction pipeline and the order of reading and writing operation matters (data dependence)

- Read-After-Write (RAW)
- Write-After-Read (WAR)
- Write-After-Write (WAW)

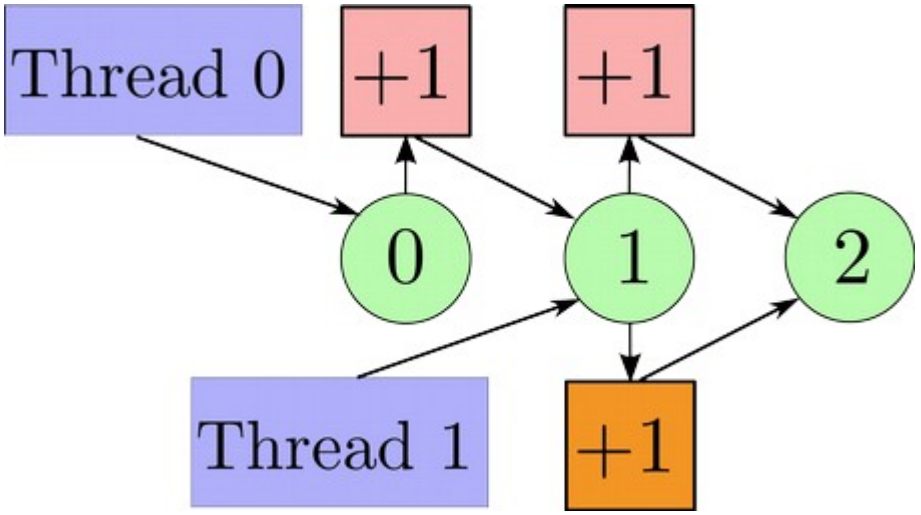
Data Hazards

Overlooking data hazards can lead to the corruption of the shared state (race condition)

Tricky to debug since the result depends on the timing between concurrent threads: unpredictable!

When a piece of code is clean of data hazards, it is said to be thread-safe.

The easiest ways to avoid conflicts in critical sections is to grant access one thread at a time: *mutex* (mutual exclusion)

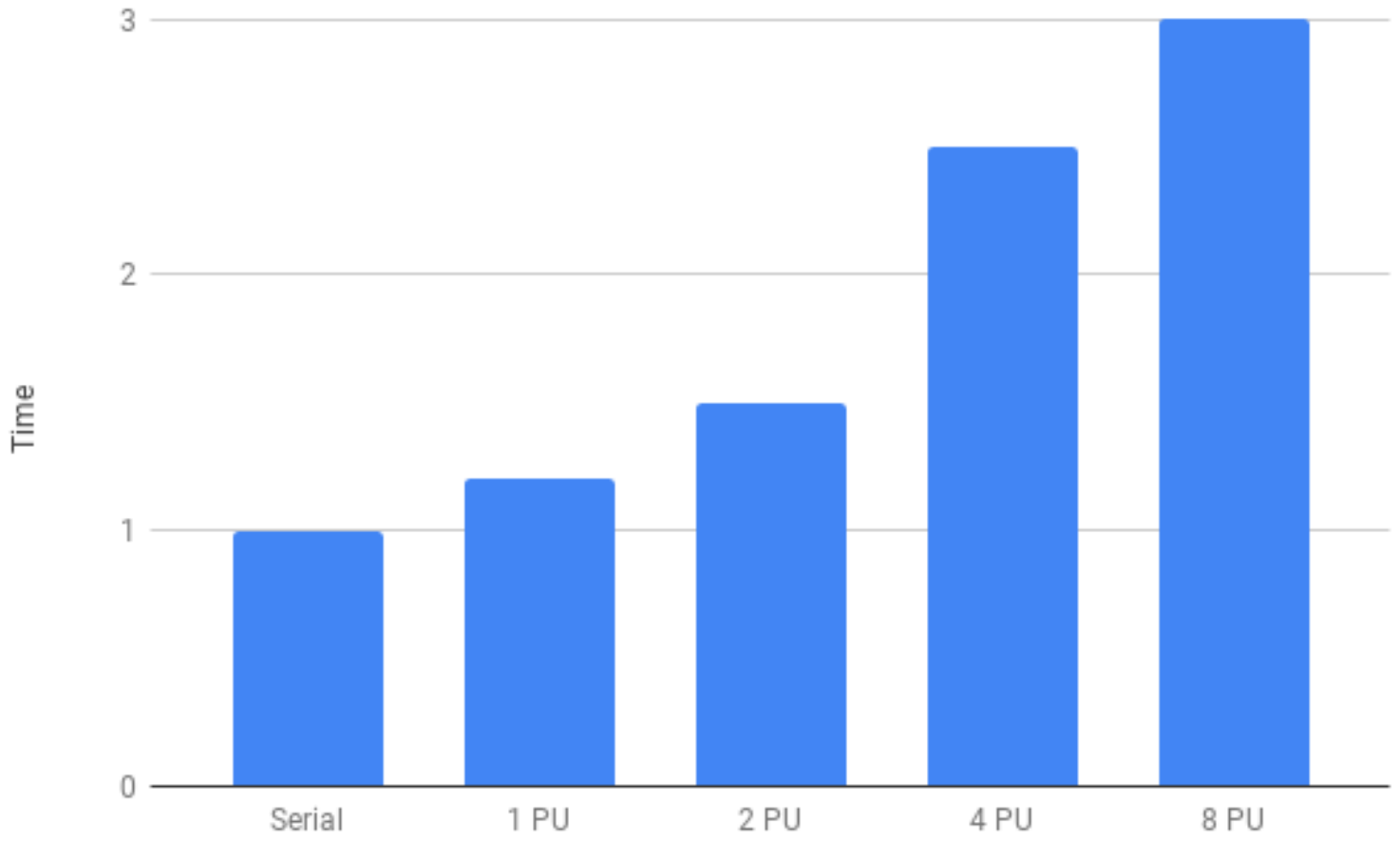


count number of 5s

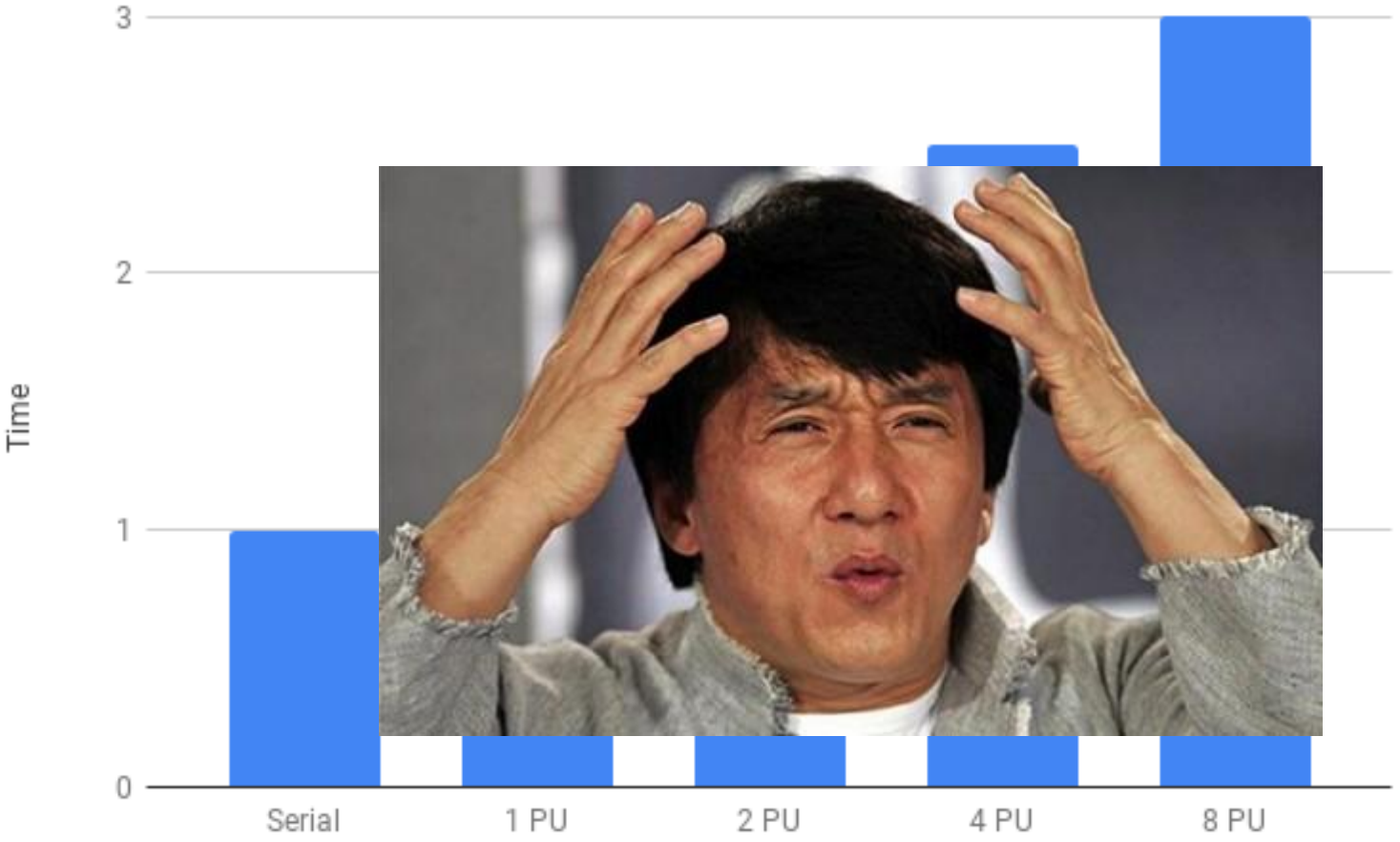
```
array[N]
numberOf5 = 0
for i in [0,N[:
    if array[i] == 5
        numberOf5++
return numberOf5
```

```
numberOf5 = 0
nWorkers = 4
count5(array, workerId):
  beg = workerId*N/nWorkers
  end = beg + N/nWorkers
  for i in [beg,end[:
      if array[i] == 5:
          lock()
          numberOf5++
          unlock()
```

Performance



Performance



Contention

- Conflicting Data Updates Cause Serialization and Delays:
- Massively parallel execution cannot afford serialization

Mitigating contention

Contention can be mitigated with:

- Privatization
- Transformation of the access pattern

- Avoid frequent transactions to/from the global main memory and read/write the data locally as much as possible before updating the global value
- Make use of registers and shared memory for aggregating partial results
- Requires storage resources to keep copies of data structures

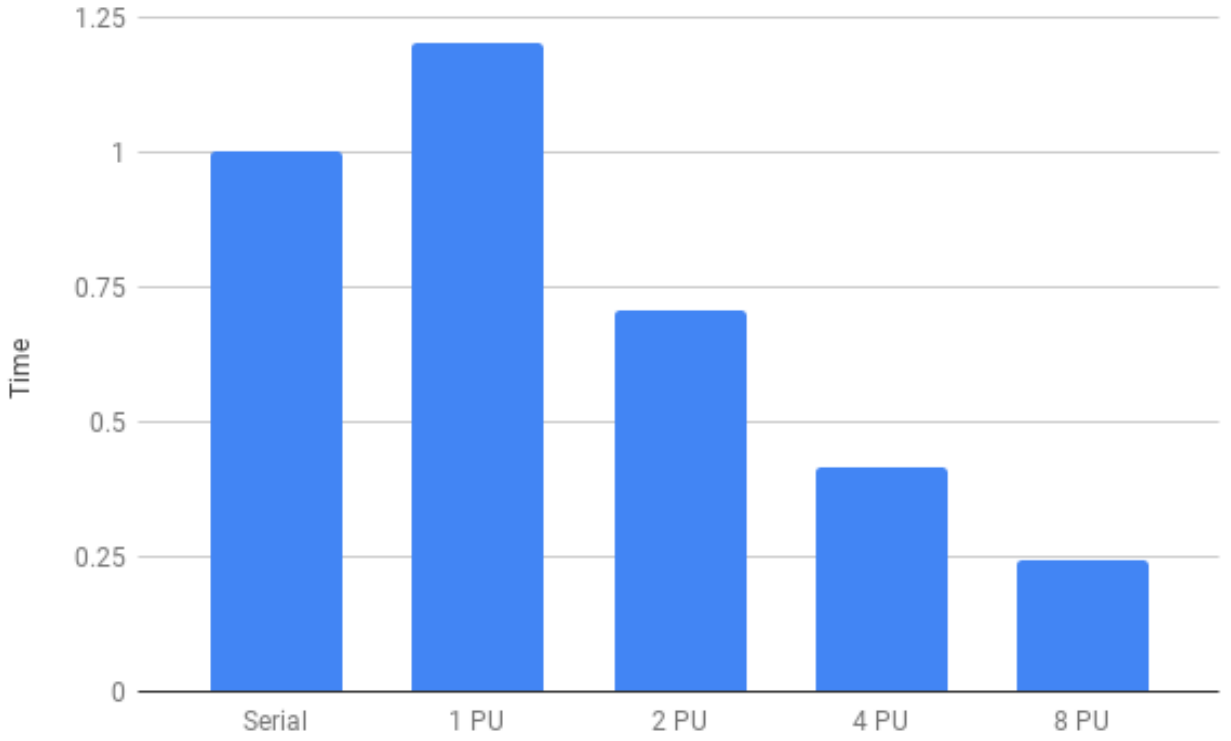
count number of 5s

```
array[N]
numberOf5 = 0
for i in [0,N[:
    if array[i] == 5
        numberOf5++
return numberOf5
```

```
numberOf5 = 0
nWorkers = 4
count5(array, workerId):
  privateResult = 0
  beg = workerId*N/nWorkers
  end = beg + N/nWorkers
  for i in [beg,end[:
      if array[i] == 5:
          privateResult++

  lock()
  numberOf5 += privateResult
  unlock()
```

Privatization



The T=8 version does not take half of the time w.r.t. T=4... Why?

Amdahl's Law

The maximum theoretical throughput is limited by Amdahl's Law:

- Every program contains a serial part
- Only one PU can execute the serial part
- The speedup using p PUs is given by

$$S(p) = \frac{T_s}{T_p}$$

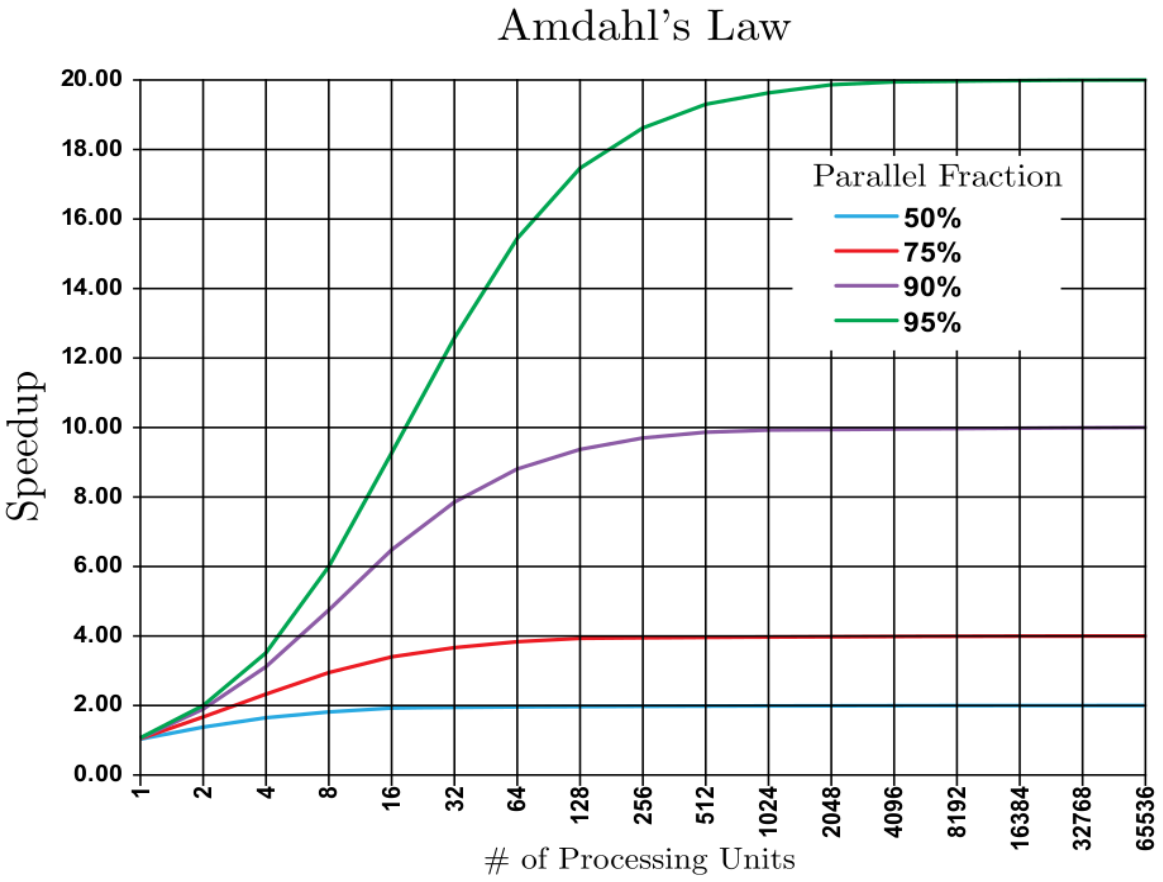
- If f is the fraction of the program that runs serially, the parallel execution time is given by:

$$T_p = fT_s + (1-f)T_p = fT_s + \frac{(1-f)t}{p}$$

Amdahl's Law (ctd.)

The speed-up becomes

$$S(p,f) = \frac{T_s}{fT_s + \frac{(1-f)t}{p}} \rightarrow \frac{1}{f} = S_{max}(f)$$



Mitigating Amdahl's Law: Gustafson's Law

- Amdahl's Law assumes that a problem can be split in a number of independent chunks n that can be processed in parallel and that this number is fixed
- Many times, the increase of the size of a problem does not correspond to a growth of the sequential part
  - increasing the size of the problem does not change the time spent executing the sequential part, and only affects the parallel portion
- Let $f\ (n)$ be the sequential code fraction of the program

$$S(n) = f(n) + p[1 - f(n)]$$

- $f(n)$ decreases to 0 when n approaches infinity.
- The maximum speedup is then given by:

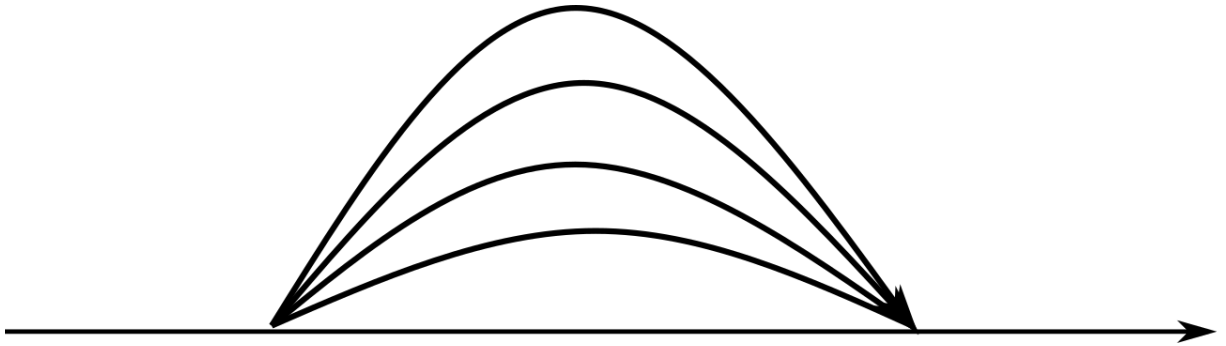
$$S_{max} \equiv \lim_{n \to \infty} S(n) = p$$

It's still worth to learn parallel computing: computations involving arbitrarily large data sets can be efficiently parallelized!

Fork-join

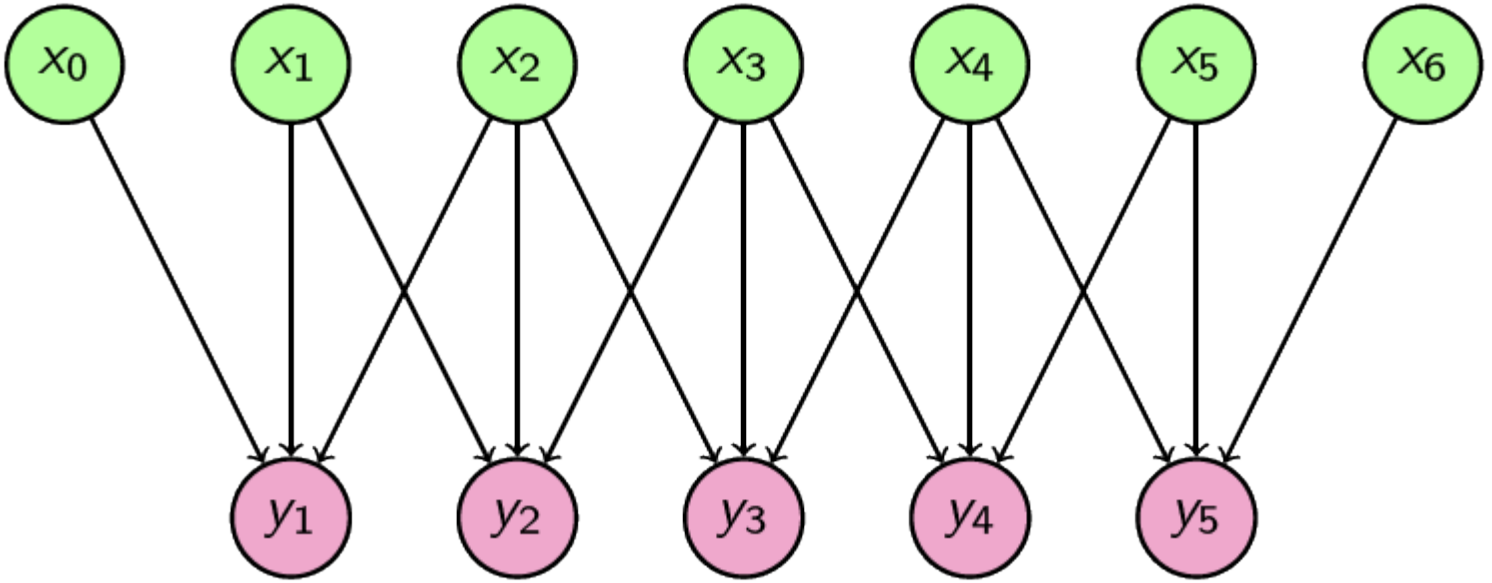
When thinking about possible parallel solutions:

- How to partition the problem
- How to share information



Data Partitioning

$$y_i = f_i(range(x_i, \delta))$$



Partitioning

- Static:
  - all information available before computation starts
  - use off-line algorithms to prepare before execution time
  - Run as pre-processor, can be serial, can be slow and expensive
- Dynamic:
  - information not known until runtime
  - work changes during computation (e.g. adaptive methods)
  - locality of objects can change (e.g. particles move)
  - use on-line algorithms to make decisions mid-execution
  - must run side-by-side with application
  - should be parallel, fast, scalable.
  - Incremental algorithm preferred (small changes in input result in small changes in partitions)

Why? In order to minimize idle time.

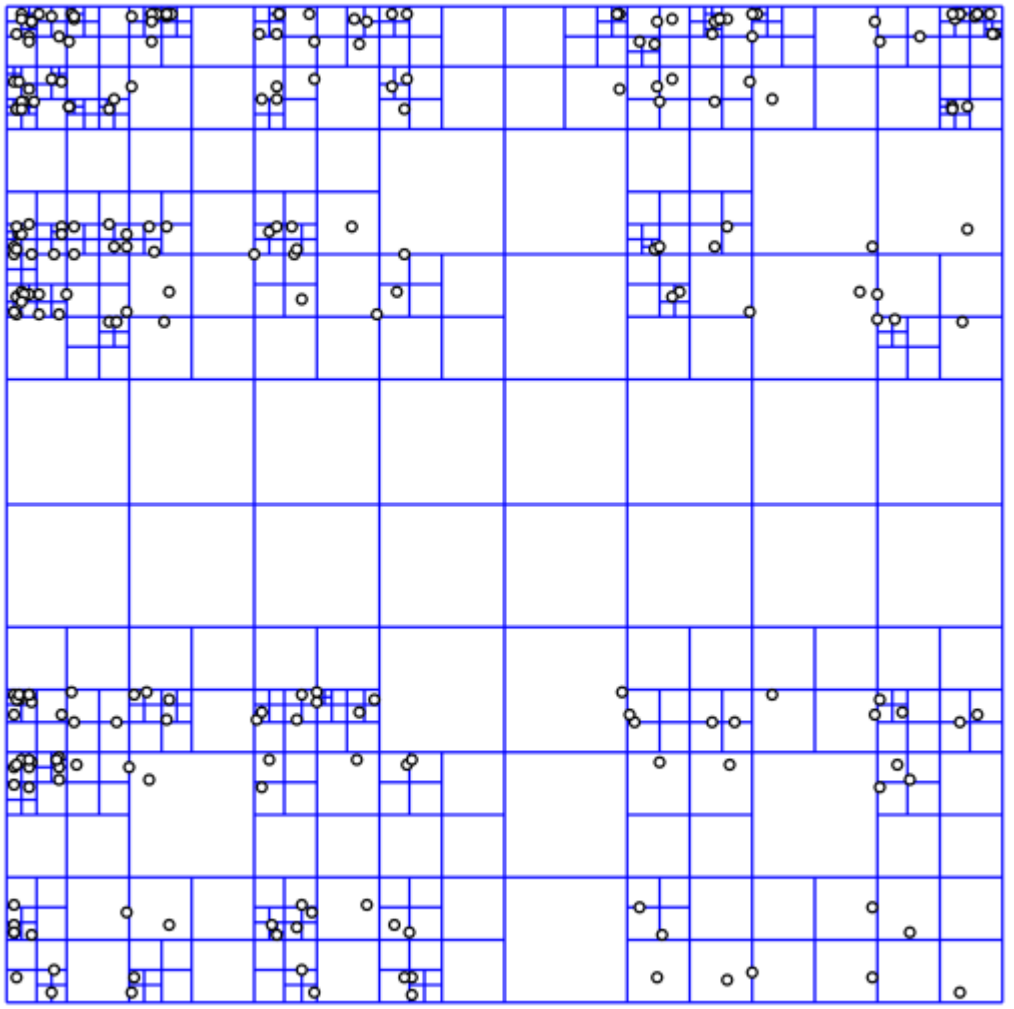
Load balancing

Sometimes dividing the input data in two does not mean that the load has been also divided in two.

Example:

Total load: 100

- If 5 workers take 20 each
  - Speedup 5
- If 1 worker takes 50
  - Speedup 2



Partitioning and Load Balancing

- Assignment of application data to processors for parallel computation
- Applied to grid points, elements, matrix rows, particles

Non-uniform data distributions

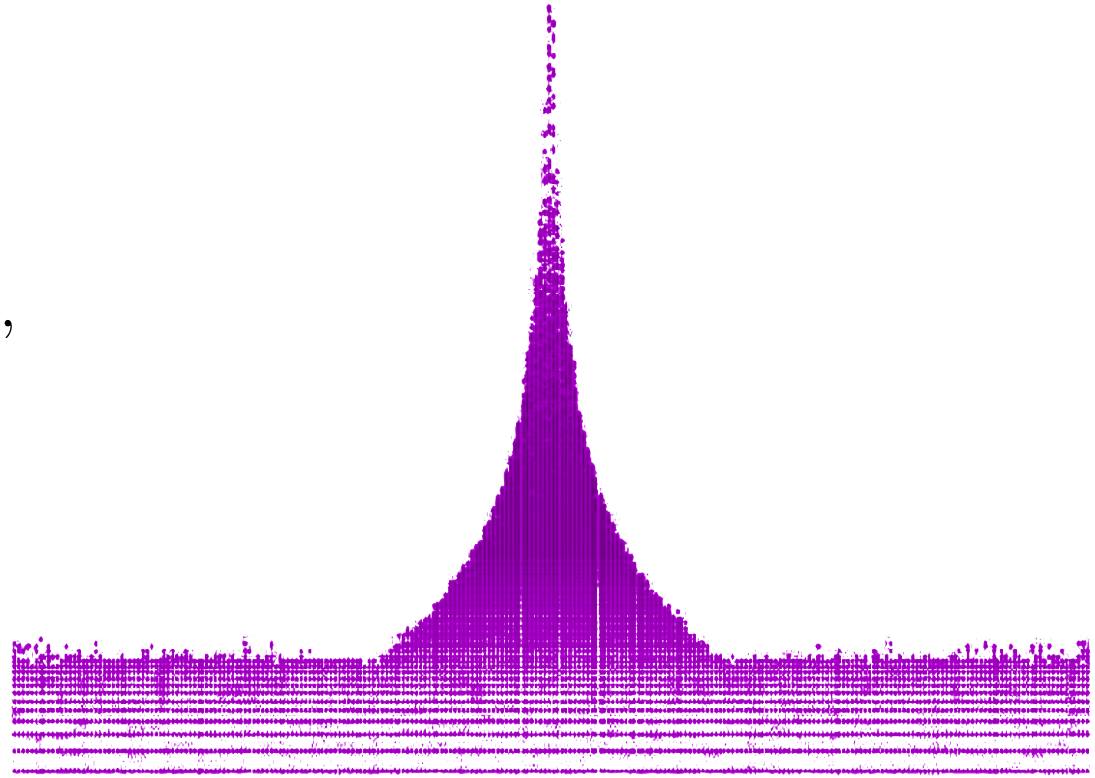
- Highly concentrated spatial

data areas

- Astronomy, medical imaging,

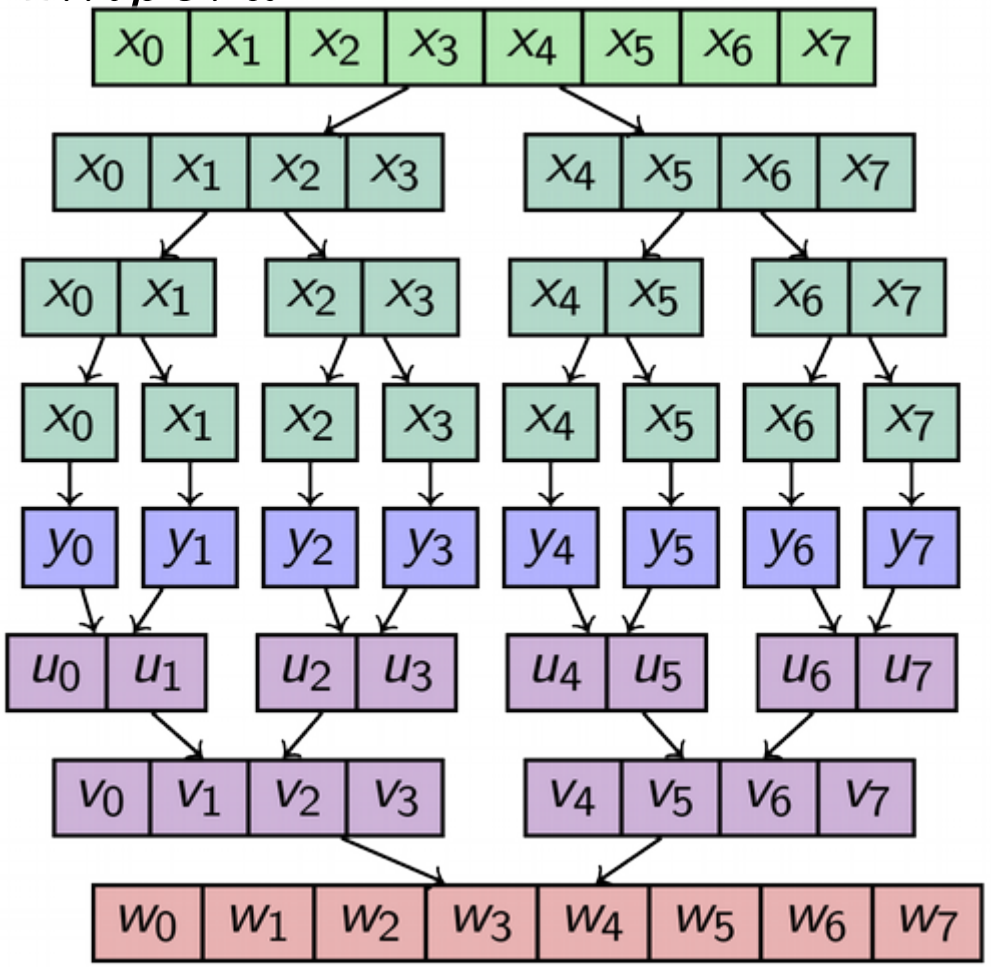
computer vision, rendering

If each thread processes the input data of a given spatial volume unit, some will do a lot more work than others



Divide et Impera

When you don't have any idea on how to approach the parallelization of a problem, try *Divide et Impera*



Load Imbalance

Sometimes load imbalance could also be caused by some underestimated consideration

- Example:

```
int N = 1000;
for(int i=0; i<N; ++i){
...
}
```

Load Imbalance

Sometimes load imbalance could also be caused by some underestimated consideration

- Example:

```
i_start = my_id * (N/num_threads);
i_end = i_start + (N/num_threads);
if (my_id == (num_threads-1))
   i_end = N;
for (i = i_start; i < i_end; i++) {
...
}
```

Load Imbalance

- The last thread executes the remainder

```
i_start = my_id * (N/num_threads);
i_end = i_start + (N/num_threads);
if (my_id == (num_threads-1))
   i_end = N;
for (i = i_start; i < i_end; i++) {
...
}
```

- If the number of threads is 32, each thread will execute 31 instructions
- The last thread will execute 8 more instructions
- Try to extrapolate to a bigger number of iterations and of threads!

Parallel computing

All exponential laws come to an end...

Parallel computing becomes useful when:

- The solution to our problem takes too much time (Amdahl's Law)
- The size of our problem is big (Gustafson's Law)
- The solution of our problems is poor, we would like to have a better one

Three steps to a better parallel software:

1. Restructure the mathematical formulation
2. Innovate at the algorithm and data structure level
3. Tune core software for the specific architecture

Microarchitecture and Metrics

CPU time

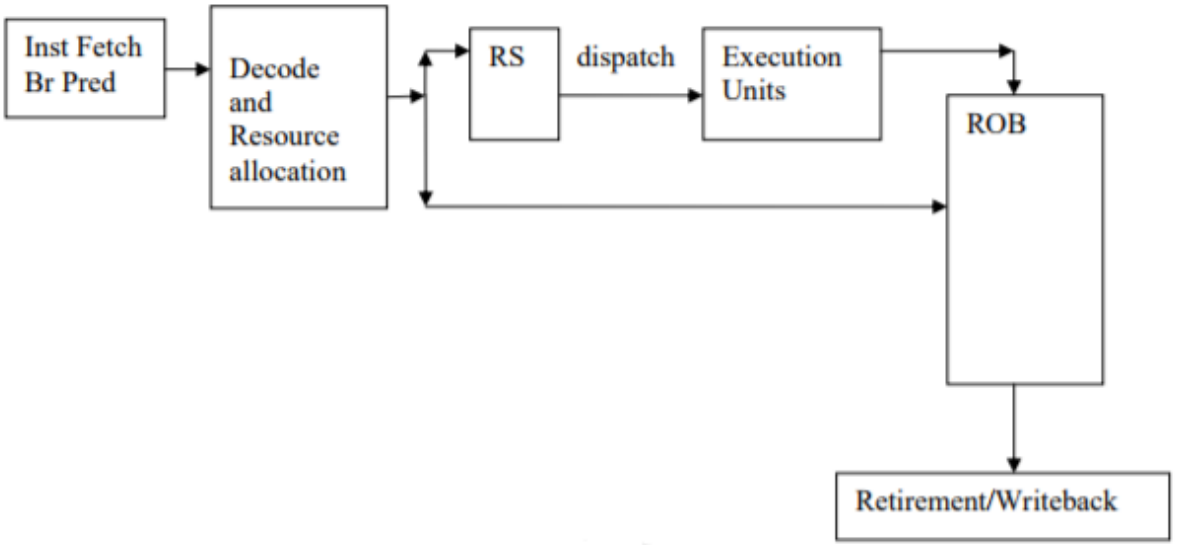
You want to minimize the CPU time and understand what handles you have

$$Time_{CPU} = \text{Instruction Count} \times \text{Cycles per Instruction} \times \text{Clock Cycle Time}$$

$$Time_{CPU} = \sum_i (\text{IC}_i \times \text{CPI}_i) \times \text{Clock Cycle Time}$$

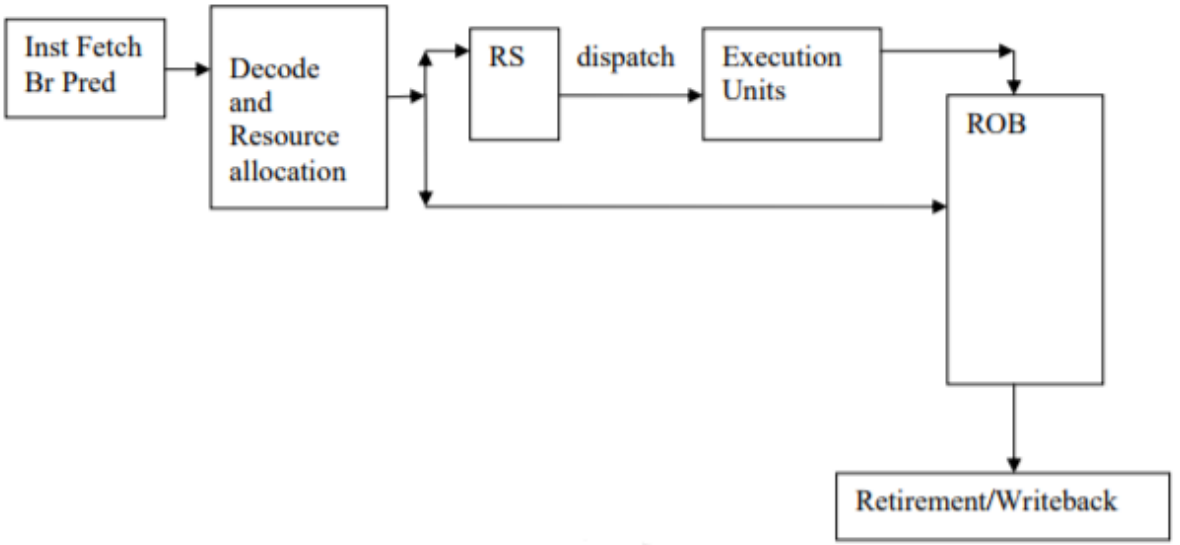
Speculative execution

- Modern processors execute many more instructions than the program flow needs (Core Out Of Order pipeline).
- The Front-end fetches the program code decodes instructions into one or more low-level hardware operations called micro-ops (`uOps`).
- The `uOps` are then fed to the Back-end in a process called allocation.
- Leaving the Retirement Unit means that:
  - the instructions are finally executed
  - their results are correct and visible in the architectural state as if they execute in-order



Retired instructions

- Instructions that were "proven" as indeed needed by the program execution flow are **retired**
- Instructions and `uOps` of incorrectly predicted paths are flushed
- Then the `uOps` associated with the instruction to be retired have completed (together with older instructions)
- Retirement of the correct execution path instructions can proceed



Clockticks per Instructions Retired (CPI)

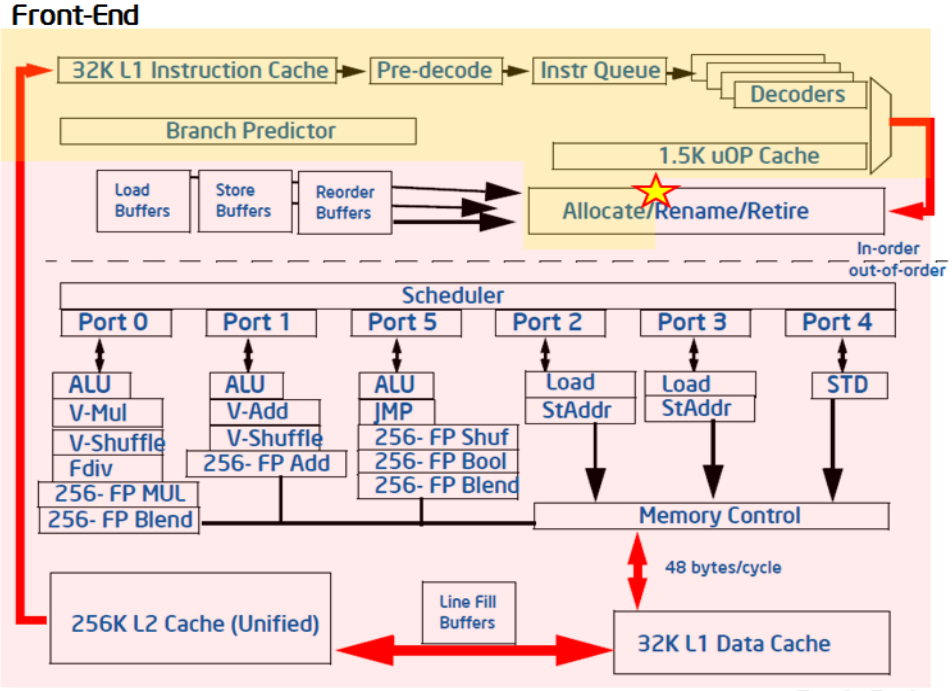
- The CPI value of an application or function is an indication of how much **latency** affected its execution
  - Higher CPI means: on average, it took more clockticks for an instruction to retire.
  - Latency in your system can be caused by cache misses, I/O, or other bottlenecks
- CPI $<$ 1: instruction bound code
- CPI $>$ 1: stall cycle bound or memory bound.

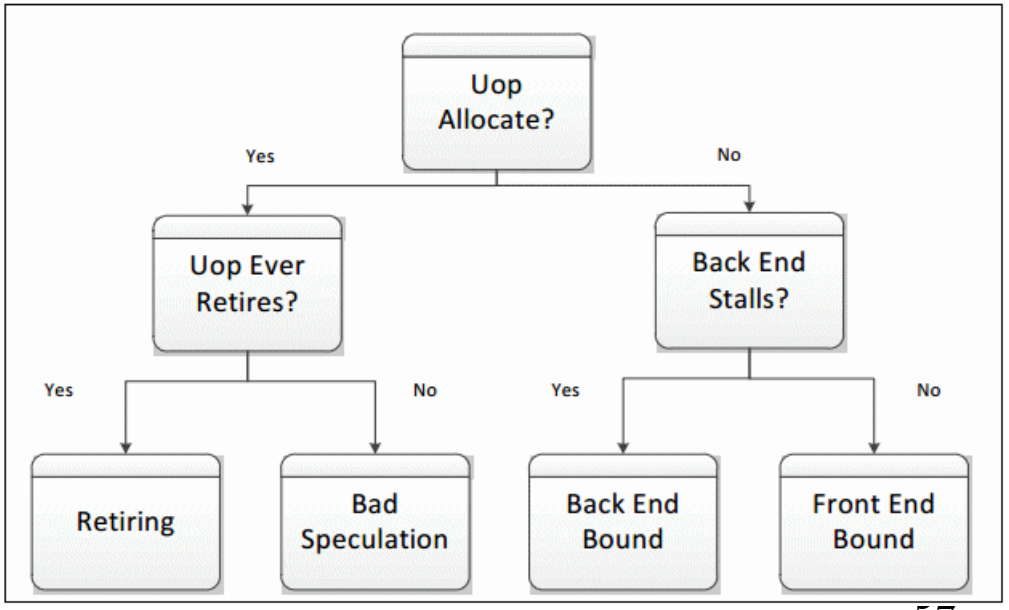
CPI vs Retired instructions

- Optimizations will affect either CPI or the number of instructions to execute, or both.
- Using CPI without considering the number of instructions executed can lead to an incorrect interpretation of your results.

Instructions pipeline

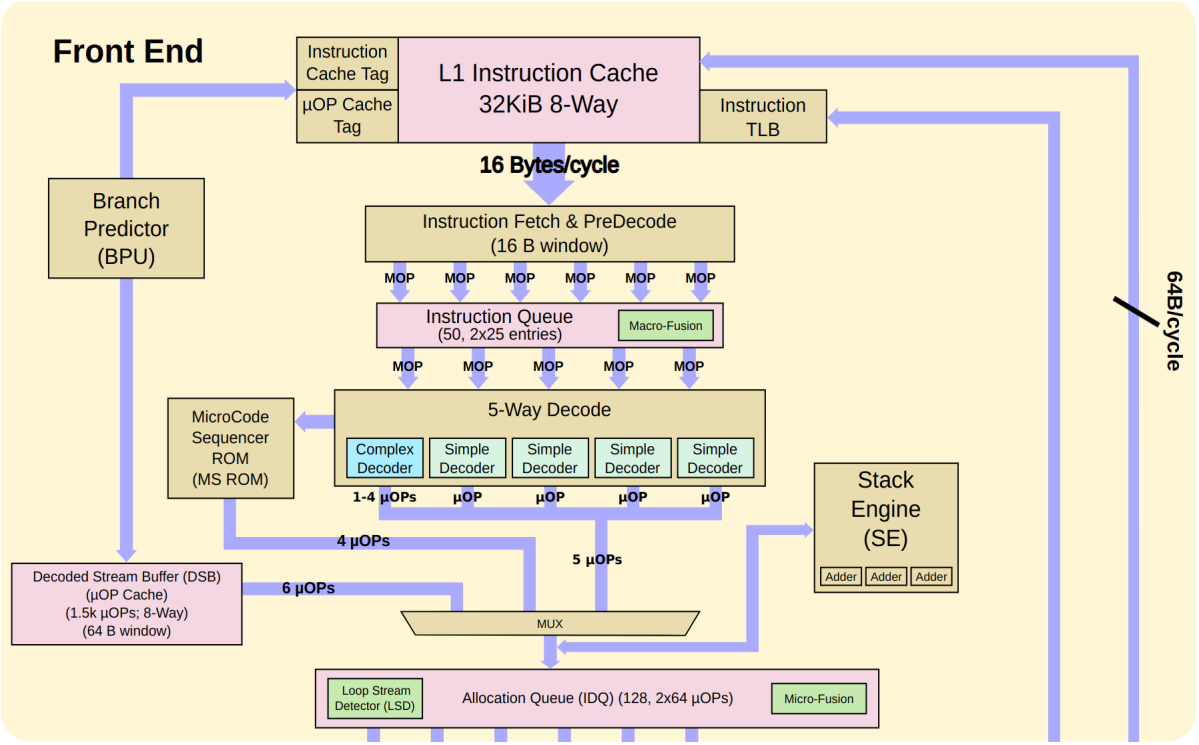
-
- The Front-end of the pipeline can allocate four uOps per cycle
- The Back-end can retire four uOps per cycle
- A pipeline slot represents the hardware resources needed to process one uOp.
- For each CPU core, on each clock cycle, there are four pipeline slots available.
- During any cycle, a pipeline slot can either be empty or filled with a uOp. If a slot is empty during one clock cycle, this is attributed to a stall. The next step needed to classify this pipeline slot is to determine whether the Front-end or the Back-end portion of the pipeline caused the stall





https://en.wikichip.org/w/images/e/ee/skylake_server_block_diagram.svg

Front-end



- Feeds "decoded" instructions to the scheduler
- Affected by instruction non-locality (iCache-miss, iTLB misses) and mispredicted branches

Main metrics:

- `L1-icache-load-misses (icache.ifdata_stall )` Cycles where a code fetch is stalled due to L1 instruction cache miss.
- `branch-misses (br_misp_retired.all_branches)` This event counts all mispredicted branch instructions retired.

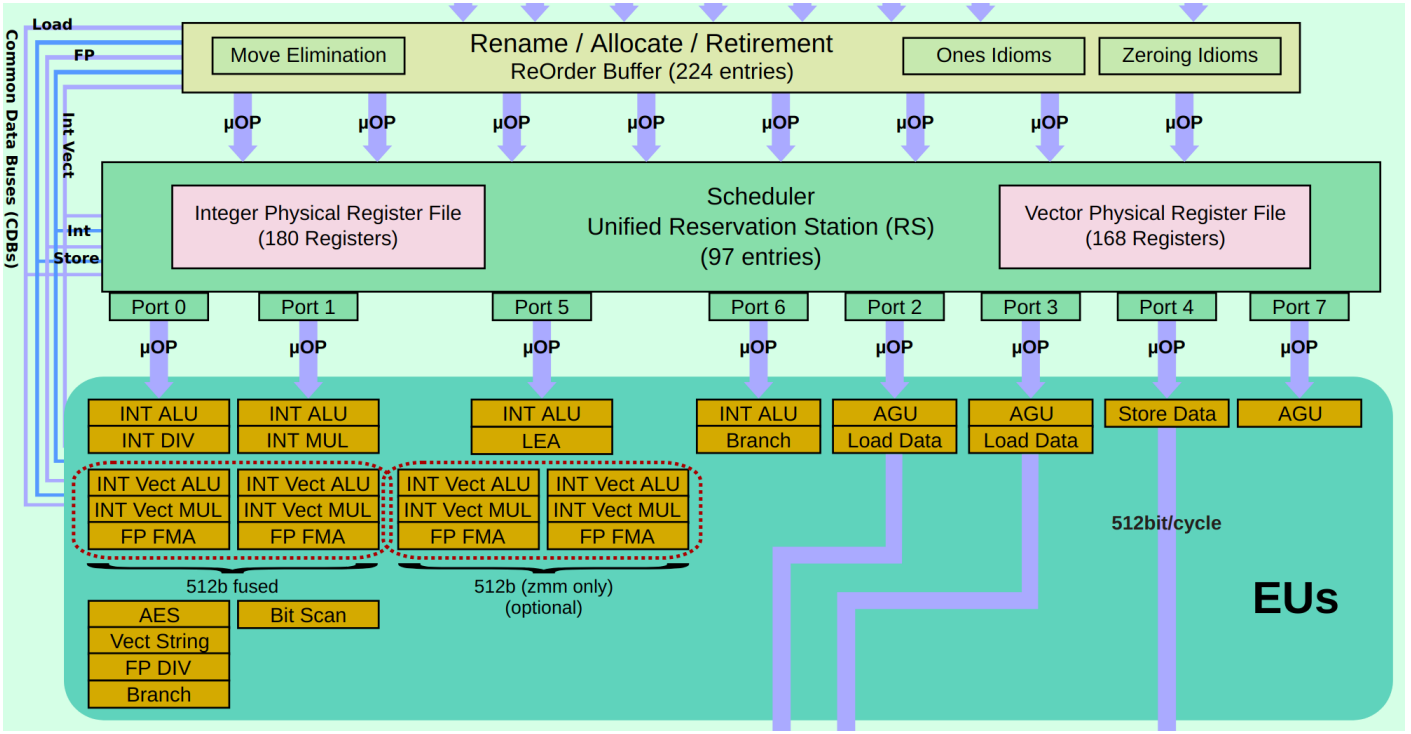
Helping the Front-end

- Avoid complex branching patterns
- Keep code local (inline)
- Keep loop short (so they fit in µOp cache)

Back-end

Computational engine of the CPU:
Affected by

- instruction dependency
  - instruction parallelism
  - pipelining
- Memory access
- Latency of slow instructions
  - `div sqrt`
- Vectorization



Main Metrics:

`uops_executed.stall_cycles`

This event counts cycles during which no uops were dispatched from the Reservation Station (RS)

`uops_executed.thread`

Number of uops to be executed each cycle.

`cycle_activity.stalls_mem_any`

Execution stalls while memory subsystem has an outstanding load.

`arith.divider_active`

Cycles when divide unit is busy executing divide or square root operations. Accounts for integer and floating-point operations.

Real-life latencies

- Most integer/logic instructions have a one-cycle execution latency:
  - For example
    - ADD, AND, SHL (shift left), ROR (rotate right)
  - Amongst the exceptions:
    - IMUL (integer multiply): 3
    - IDIV (integer divide): 13 – 23
- Floating-point latencies are typically multi-cycle
  - FADD (3), FMUL (5)
    - Same for both x87 and SIMD double-precision variants
  - Exception: FABS (absolute value): 1
  - Many-cycle, no pipepine : FDIV (20), FSQRT (27)
  - Other math functions: even more
- As of Haswell:
  - FMA (5 cycles)
- As of Skylake:
  - SIMD ADD, MUL, FMA: 4 cycles

http://www.agner.org/optimize/instruction_tables.pdf

Helping the Back-end

- Keep data at hand
- Vectorize
- Recast loop to help the compiler to vectorize
- Avoid divisions and sqrt!

Helping the compiler to vectorize

- Vectorization is enabled in gcc by the flags:
  - `-ftree-vectorize`
  - `-O3`
- Vectorizable:
  - Countable innermost loops
  - No variations in the control flow
  - Contiguous memory access
  - Independent memory access
- Avoid aliasing problems with `restrict`
- Use countable loops, with no side effects (`break`, `continue`, non-inlined function calls)
- Avoid indirect memory access (`x[y[i]]`)

Not vectorizable

```
while(x[i] != 42)
{
    if(x[i] == 0)
        x[i] = x[i-1]
}
```

Conclusion

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
  - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data, data structures, memory
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part)

References

- V. Innocente, ESC19 Architecture lecture
- John Hennessy, David Patterson, Computer Architecture A Quantitative Approach