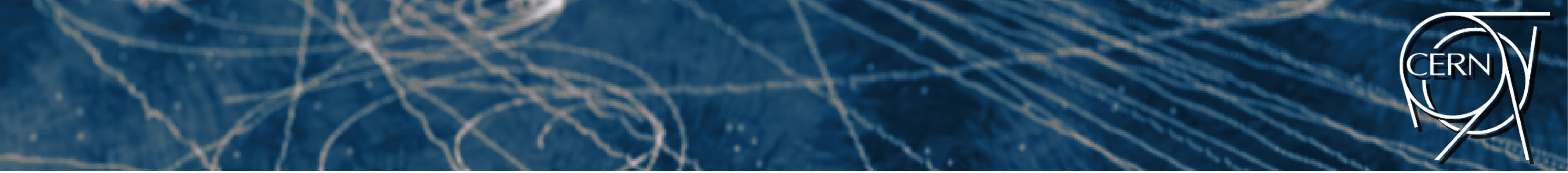


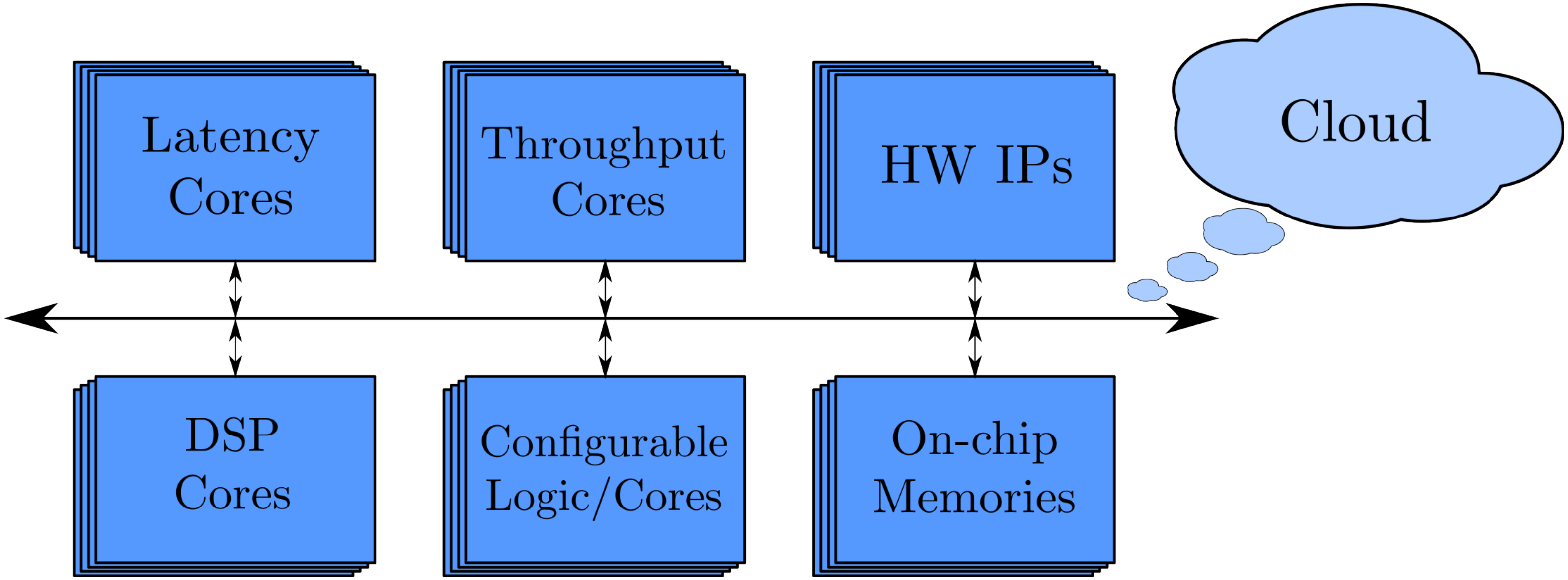
Introduction to Parallel Programming

Felice Pantaleo

CERN – Experimental Physics Department

felice@cern.ch

Heterogeneous computing



Lamborghini Huracán Display

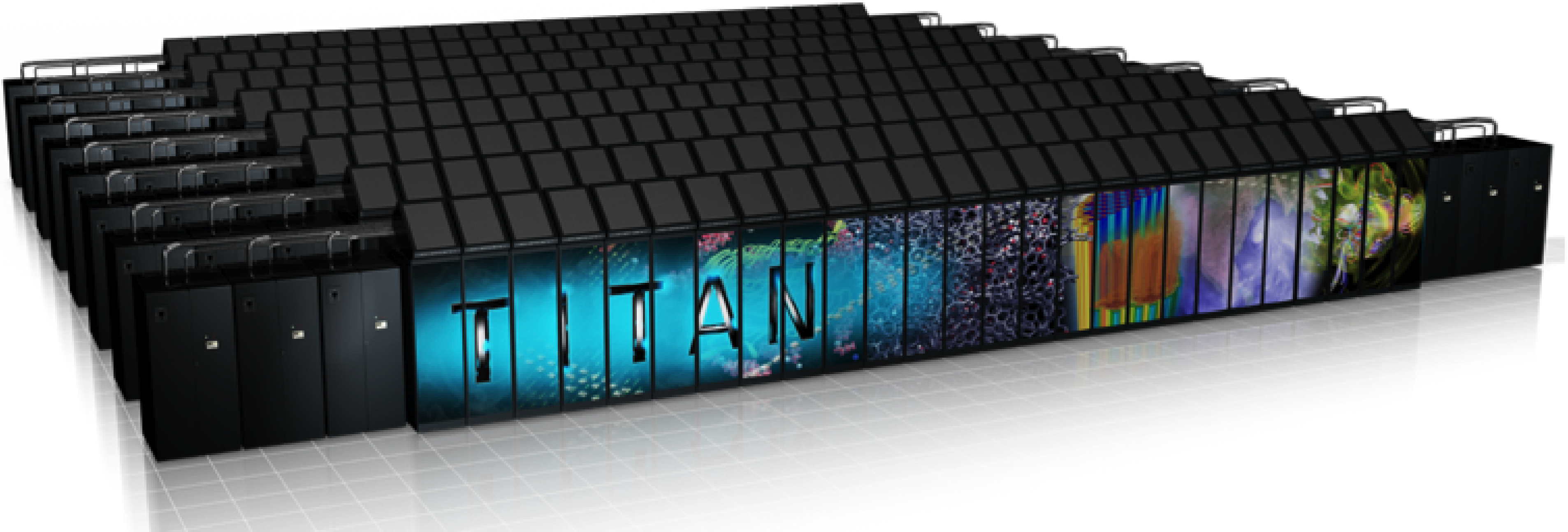
Heterogeneous computing system:

- Best match for the job
- Energy efficiency
- Higher Performance

Supercomputing systems

TITAN: 200 cabinets Cray XK7 – 18688 nodes – 17.5 PFLOPS

(AMD Opteron 16 cores + NVIDIA Tesla K20)



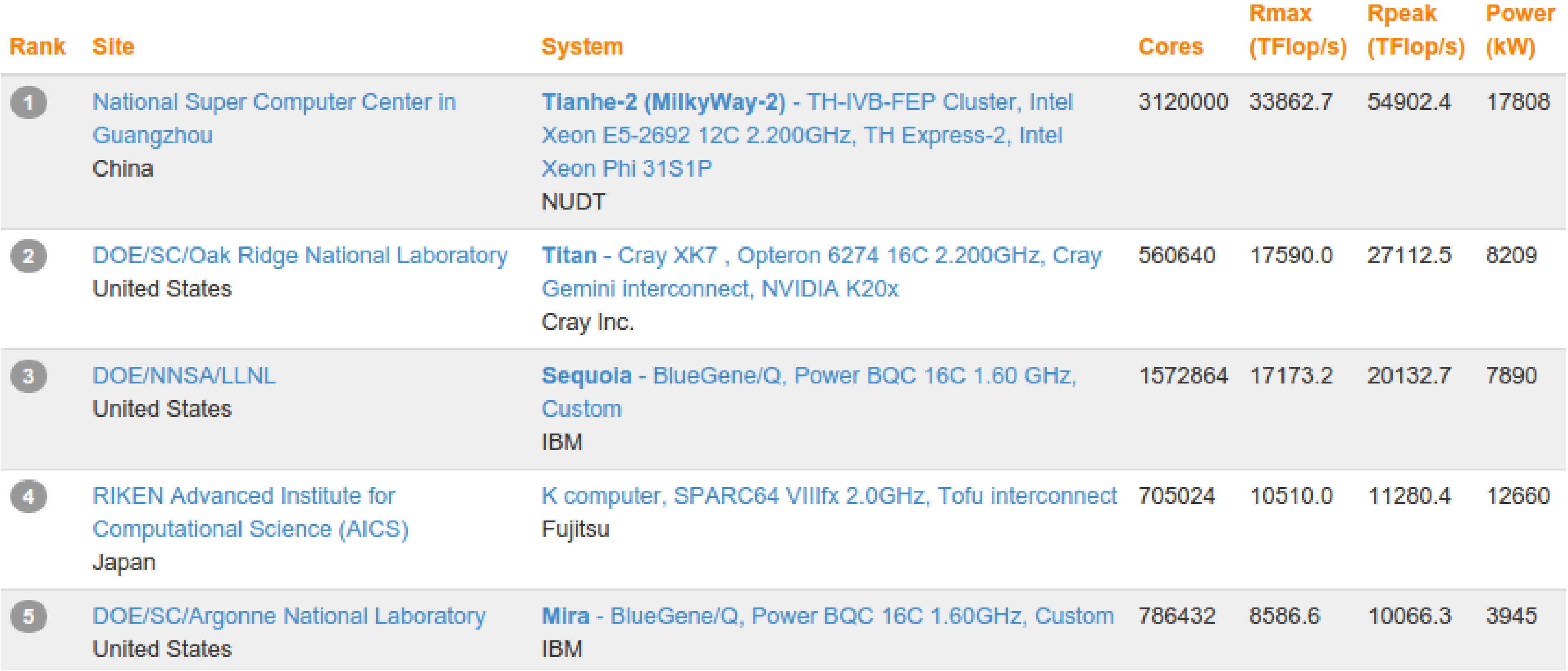
- Achieve high computing power
- Dedicated to execute heavy computation
- Usually belong to big companies or research institutes
- Resources shared by using a batch queue system

LINPACK & TOP500

LINPACK is a benchmark introduced in the '70s to

- Ease the choice of the best computer for a job
- Define the performance of a computer independently from the architecture
- Consists in solving a dense system of linear equations

TOP500: list of the world 500 fastest supercomputers ranked accordingly to LINPACK benchmark

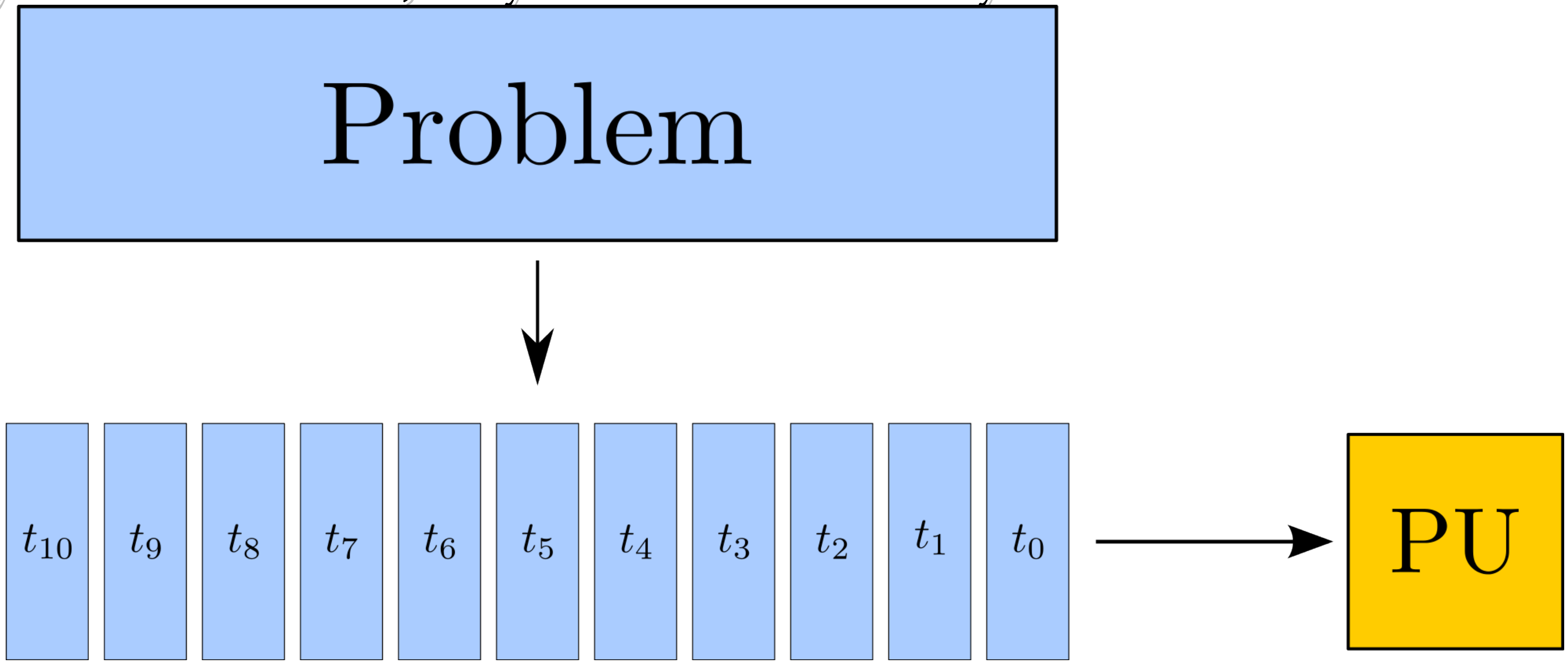


Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945

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



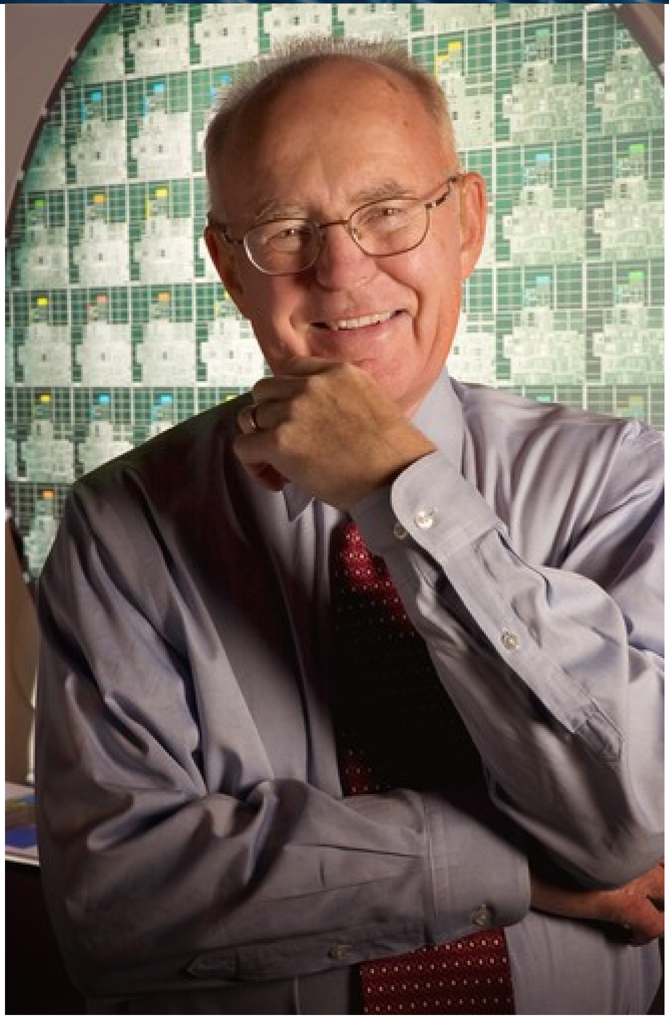
Moore's Law

In 1965, Gordon Moore: "*The performance of microprocessors and the number of their transistors will double every 12 months*"

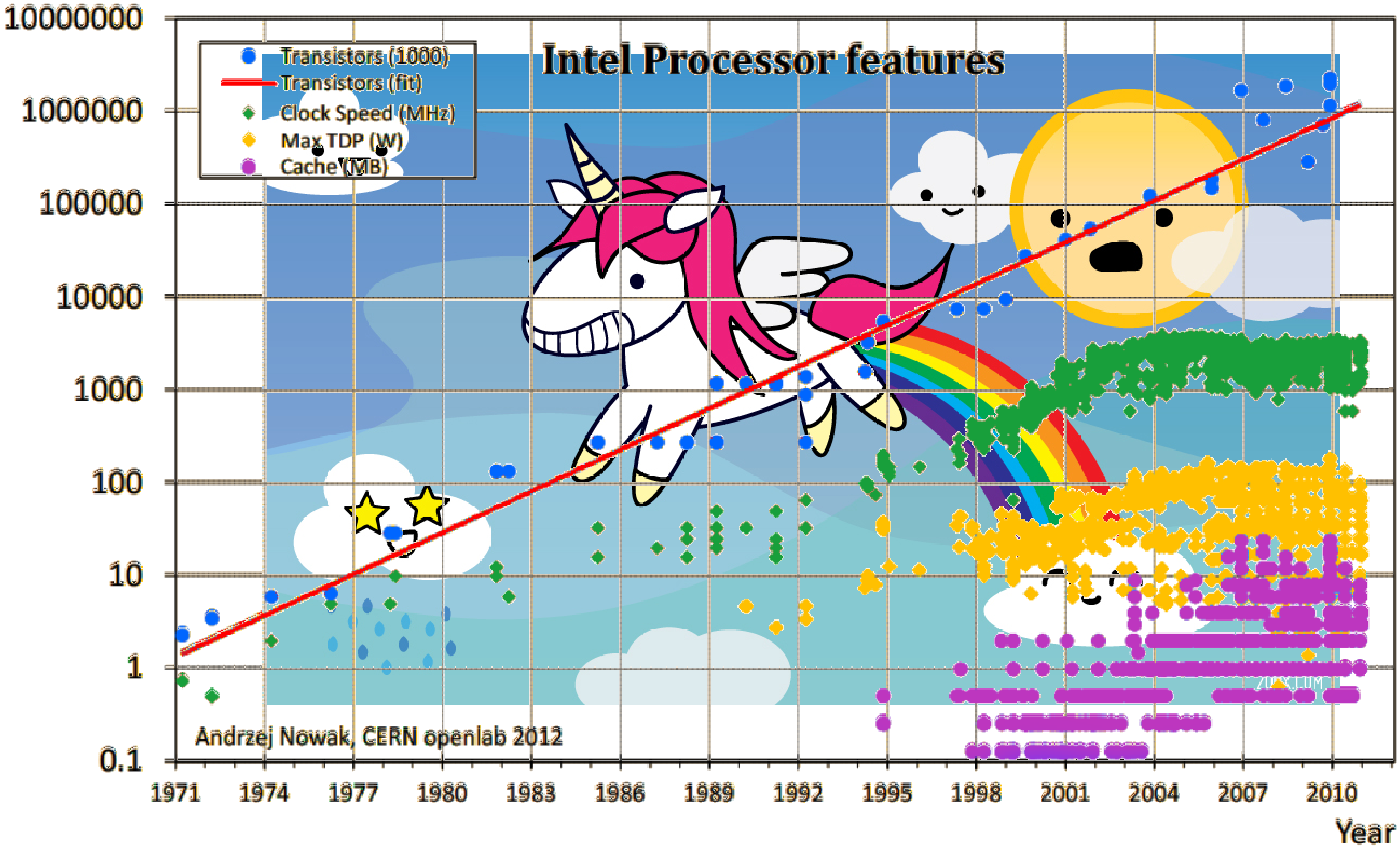
Elaborated in its final form: microprocessor performance doubles every *18 months*

Engineers found out that computation could be accelerated by increasing the clock speed:

The march towards higher clock frequencies started!



Moore's Law (ctd.)



Andrzej Nowak, CERN openlab 2012

Power Wall

$$\text{Power} = f(C, V^2, \text{frequency})$$

Reducing the voltage is not always possible:

- Faster clock rates sometimes demand higher voltage
- Higher voltage means less trouble due to random noise

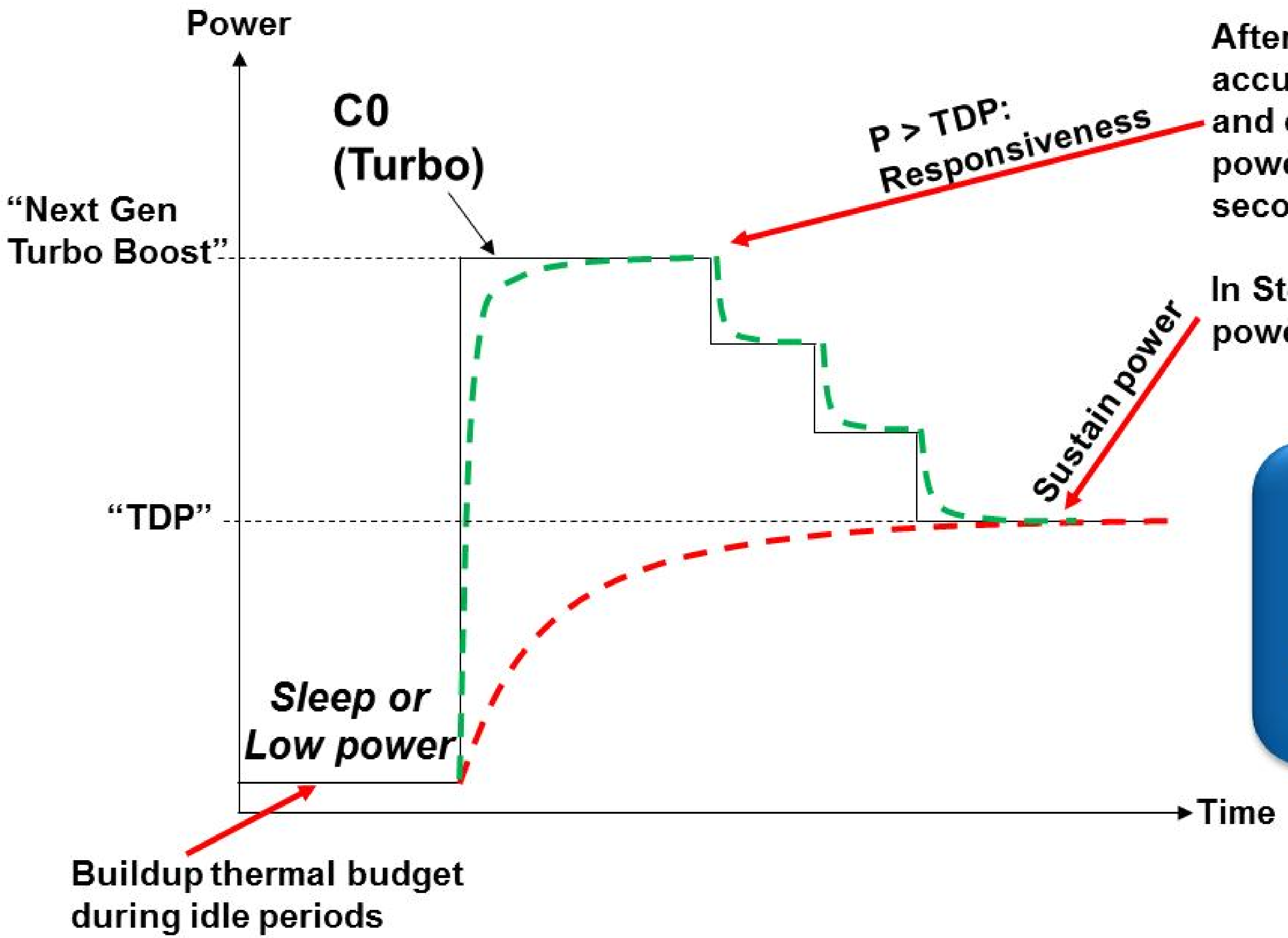
Many commercial chip manufacturers adopted a throughput oriented philosophy:

Increase the throughput of a number of programs running concurrently.

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

Mitigating the Power Wall

Intel Turbo Boost:



After idle periods, the system accumulates "energy budget" and can accommodate high power/performance for a few seconds

In Steady State conditions the power stabilizes on TDP

Buildup thermal budget during idle periods

Use accumulated energy budget to enhance user experience

Memory 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?

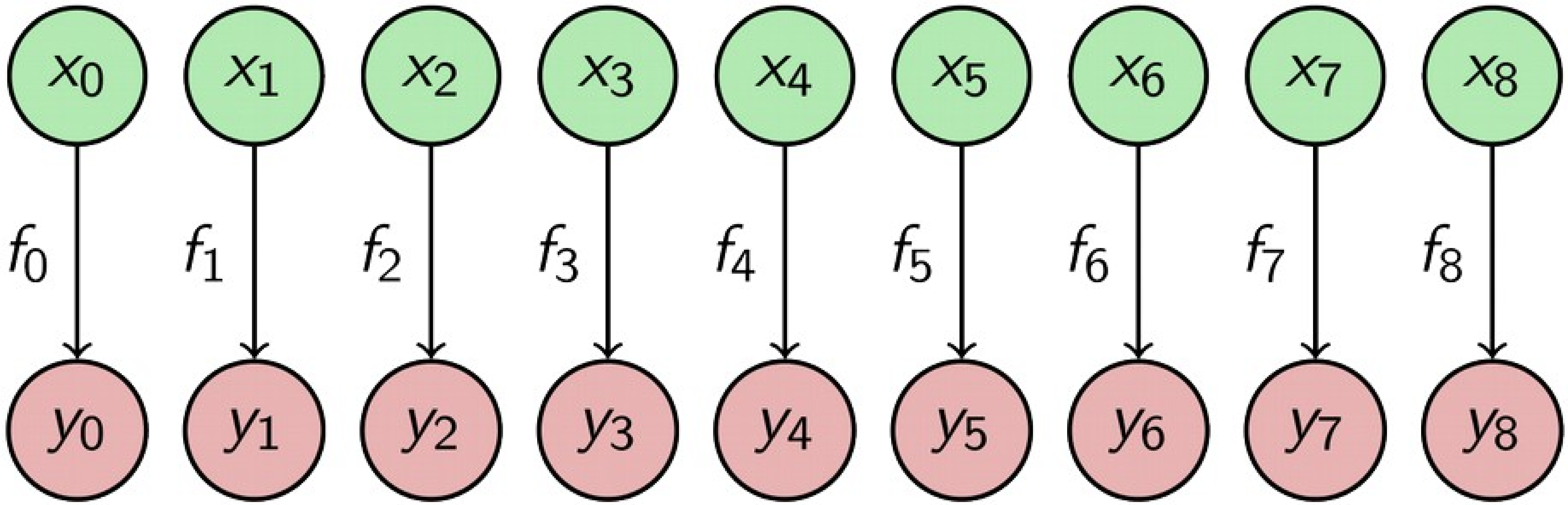
Contention of the memory bus.

Mitigating the Memory Wall

- Reuse data and instructions
- Move the data close to where the execution happens
- Increase the memory transfer speed
- Increase the amount of data to transfer
- Improve the pattern of access to memory

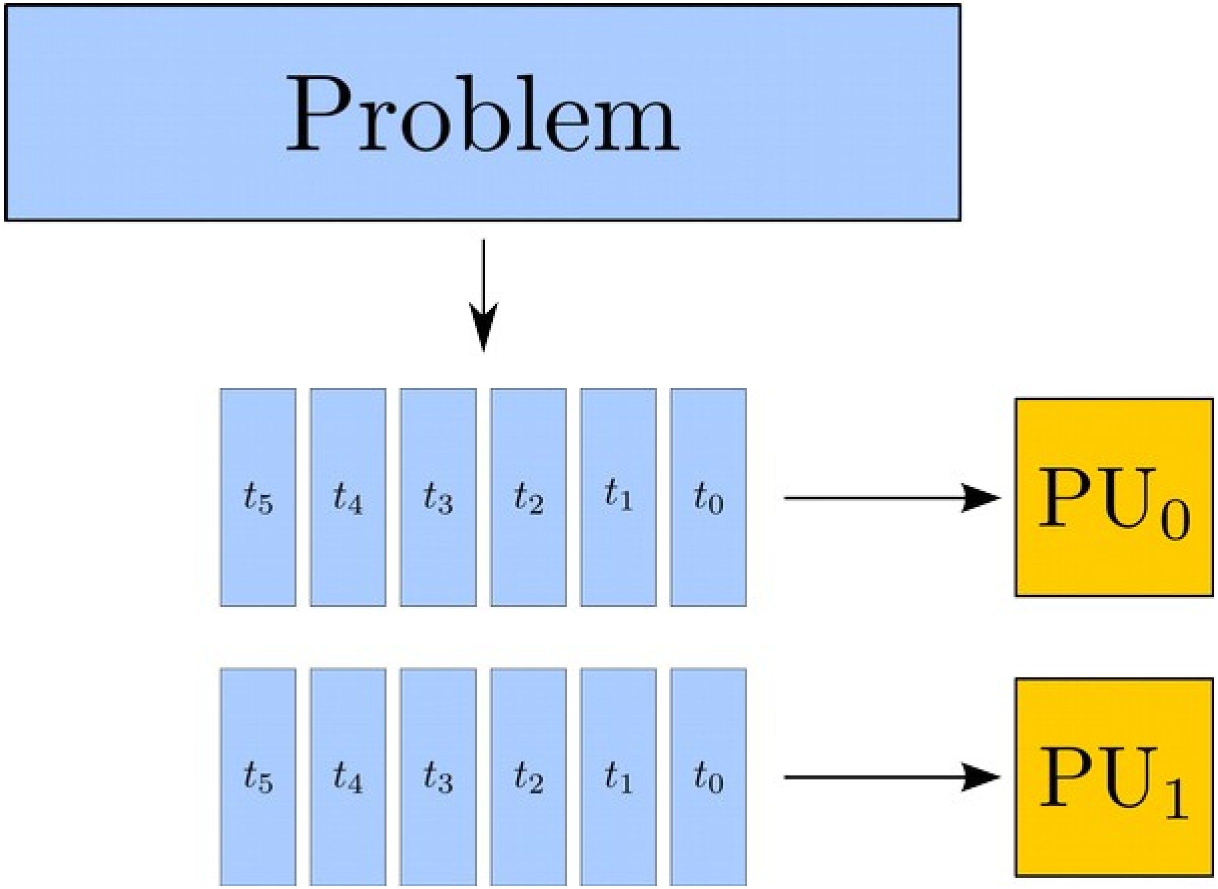
Embarrassingly parallel problems

$$y_i = f_i(x_i)$$



Embarrassingly parallel problems (ctd.)

Workload can be divided into a number of independent sub-problems that can be processed by different PUs



Embarrassingly parallel problems (ctd.)

Examples:

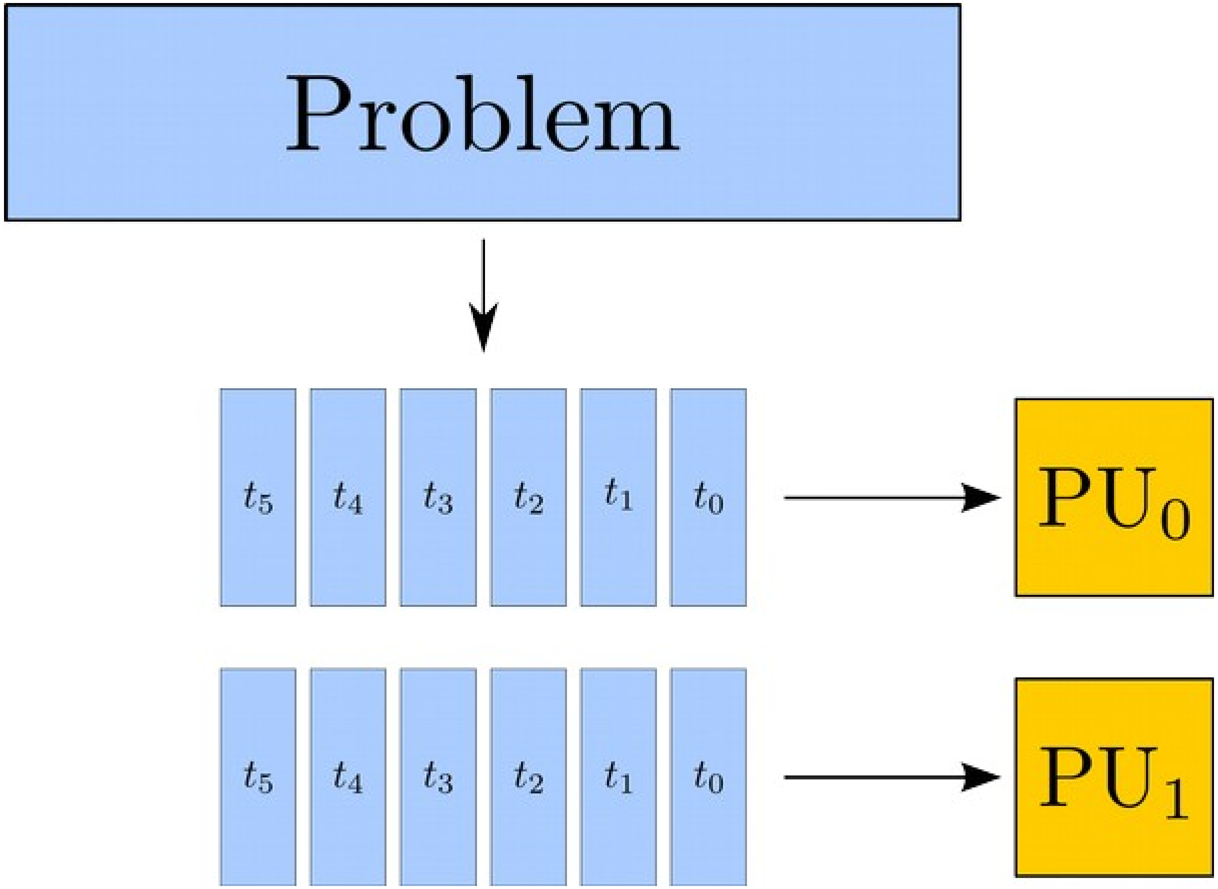
- Linear Algebra
- Image Processing
- Monte Carlo Simulation
- Bruteforce
- Weather forecast
- Random number generation
- Encryption
- Software compilation

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 threads 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 thread per time can enter.

Terminology (ctd.)



Flynn's Taxonomy

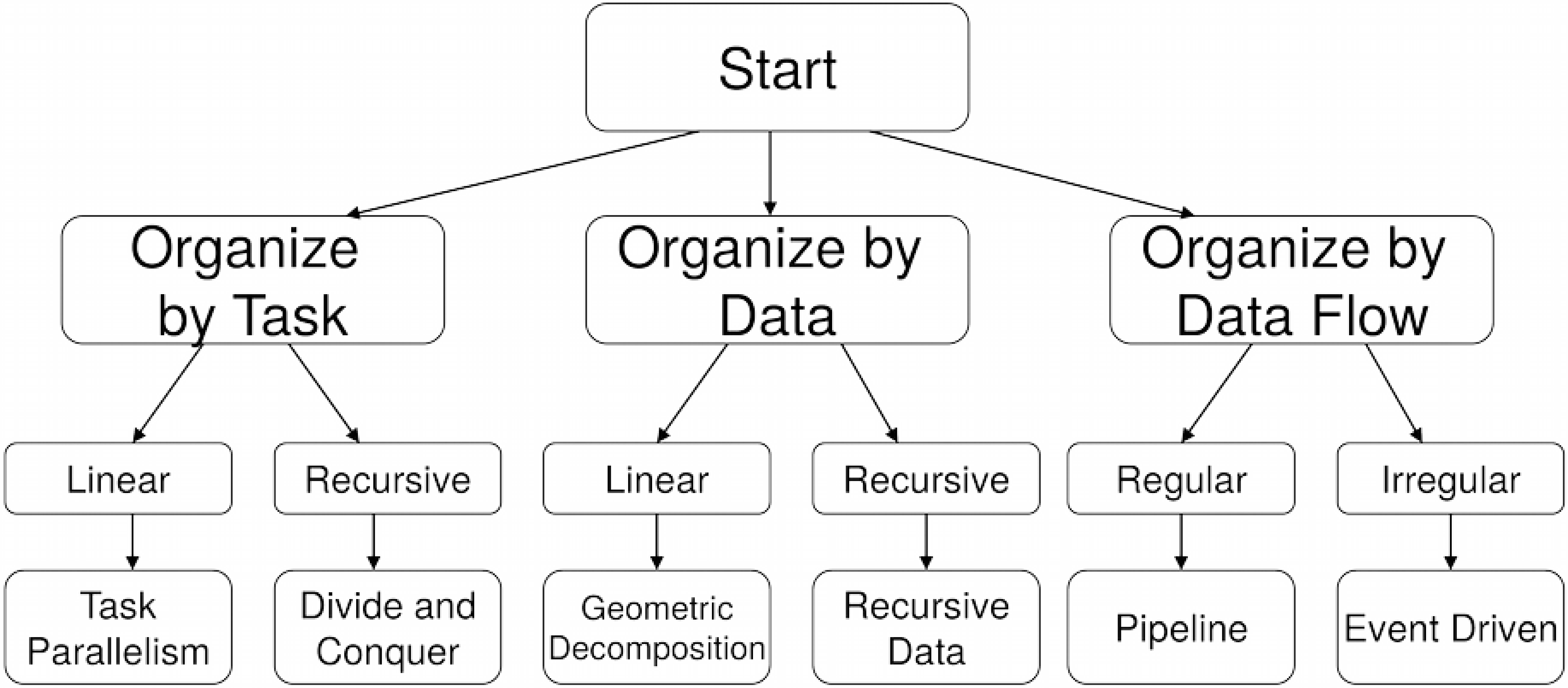
Classification of computers describes four classes in both serial and parallel contexts:

- **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.

Patterns for Parallel Programming



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

Killer kernels

Researchers at UC at Berkeley defined these killer kernels, dwarfs as the patterns of the modern parallel computing

A dwarf is an algorithmic method that captures a pattern of computation and communication

- Dense Linear Algebra
- Sparse Linear Algebra
- Spectral Methods
- N-Body Methods
- Structured Grids
- Unstructured Grids
- MapReduce
- Combinational Logic
- Graph Traversal
- Dynamic Programming
- Backtrack and Branch-and-Bound
- Graphical Models
- Finite State Machines

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

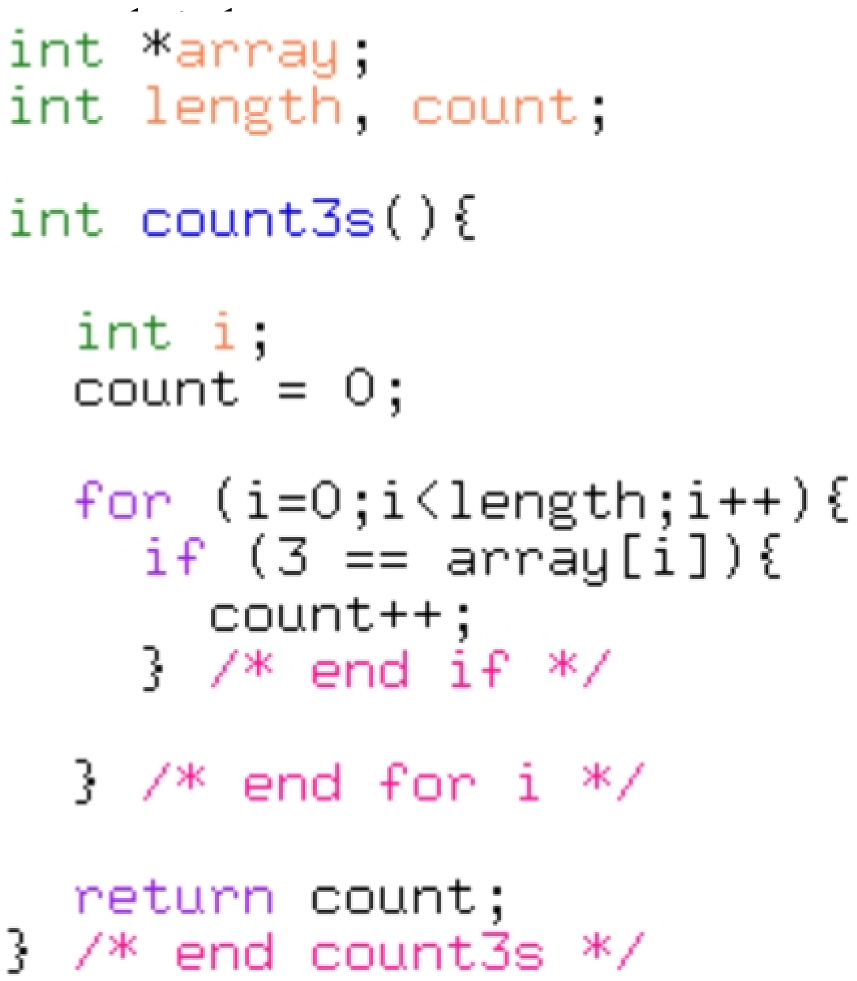
Examples:

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

count3s

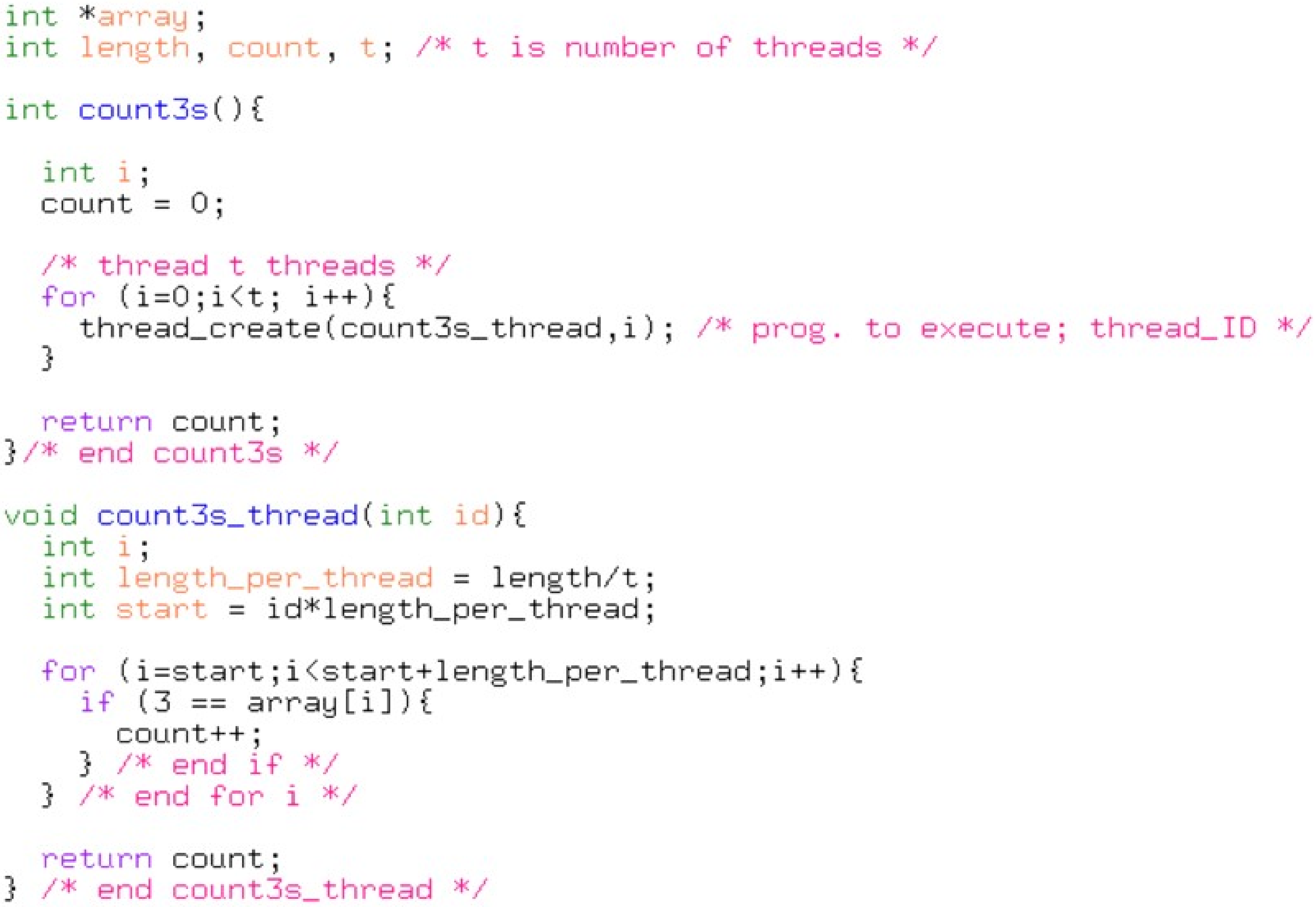
Parallel programming is not easy:

- Trivial problems like counting the number of “3”s in an array



```
int *array;
int length, count;

int count3s(){

   int i;
   count = 0;

   for (i=0;i<length;i++){
      if (3 == array[i]){
         count++;
      } /* end if */

   } /* end for i */

   return count;
} /* end count3s */
```

count3s (ctd.)



```
int *array;
int length, count, t; /* t is number of threads */

int count3s(){

   int i;
   count = 0;

   /* thread t threads */
   for (i=0;i<t; i++){
      thread_create(count3s_thread,i); /* prog. to execute; thread_ID */
   }

   return count;
}/* end count3s */

void count3s_thread(int id){
   int i;
   int length_per_thread = length/t;
   int start = id*length_per_thread;

   for (i=start;i<start+length_per_thread;i++){
      if (3 == array[i]){
         count++;
      } /* end if */
   } /* end for i */

   return count;
} /* end count3s_thread */
```

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

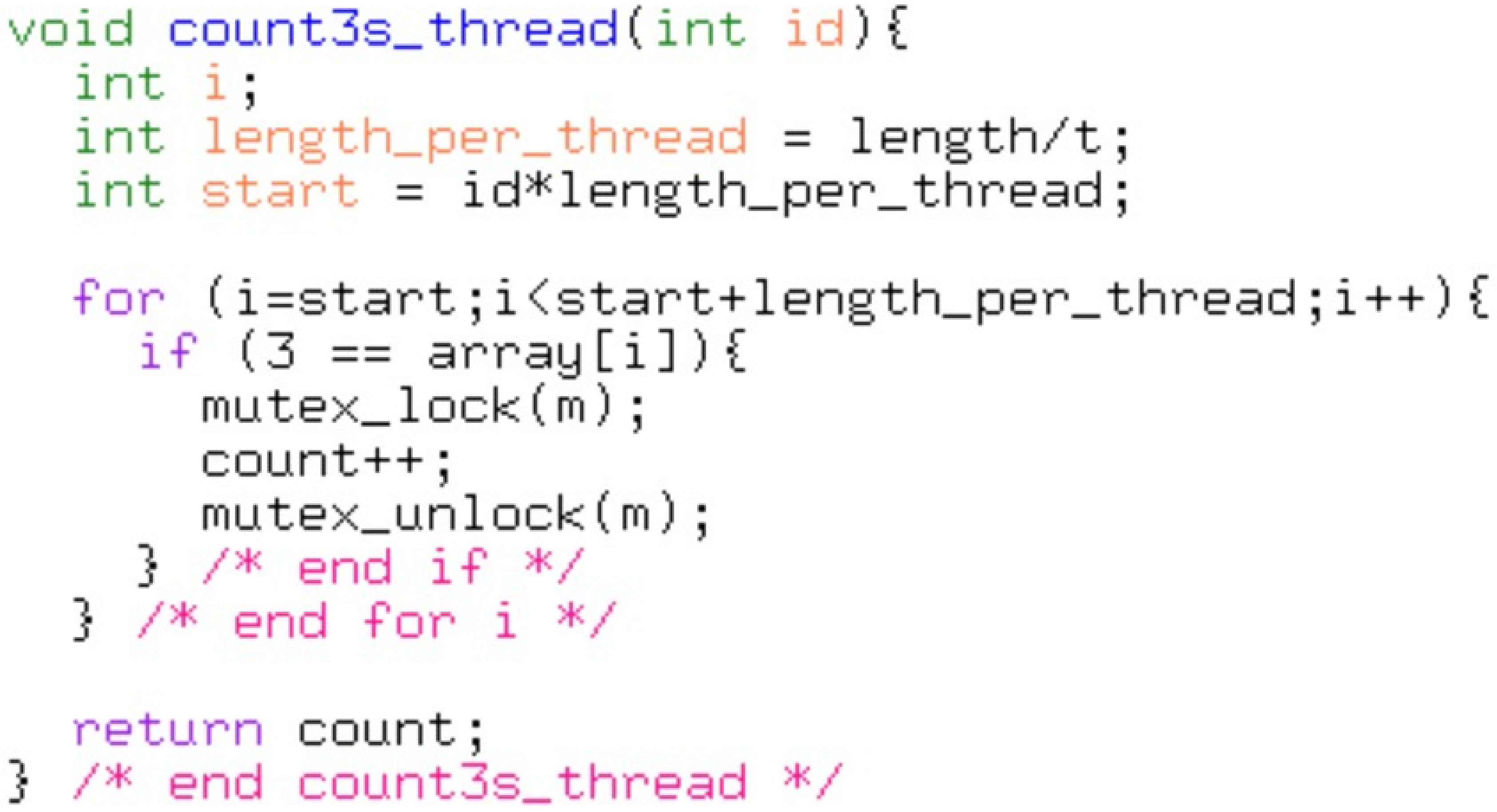
Overlooking data hazards can lead to the corruption of the shared state (race conditions)

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*.

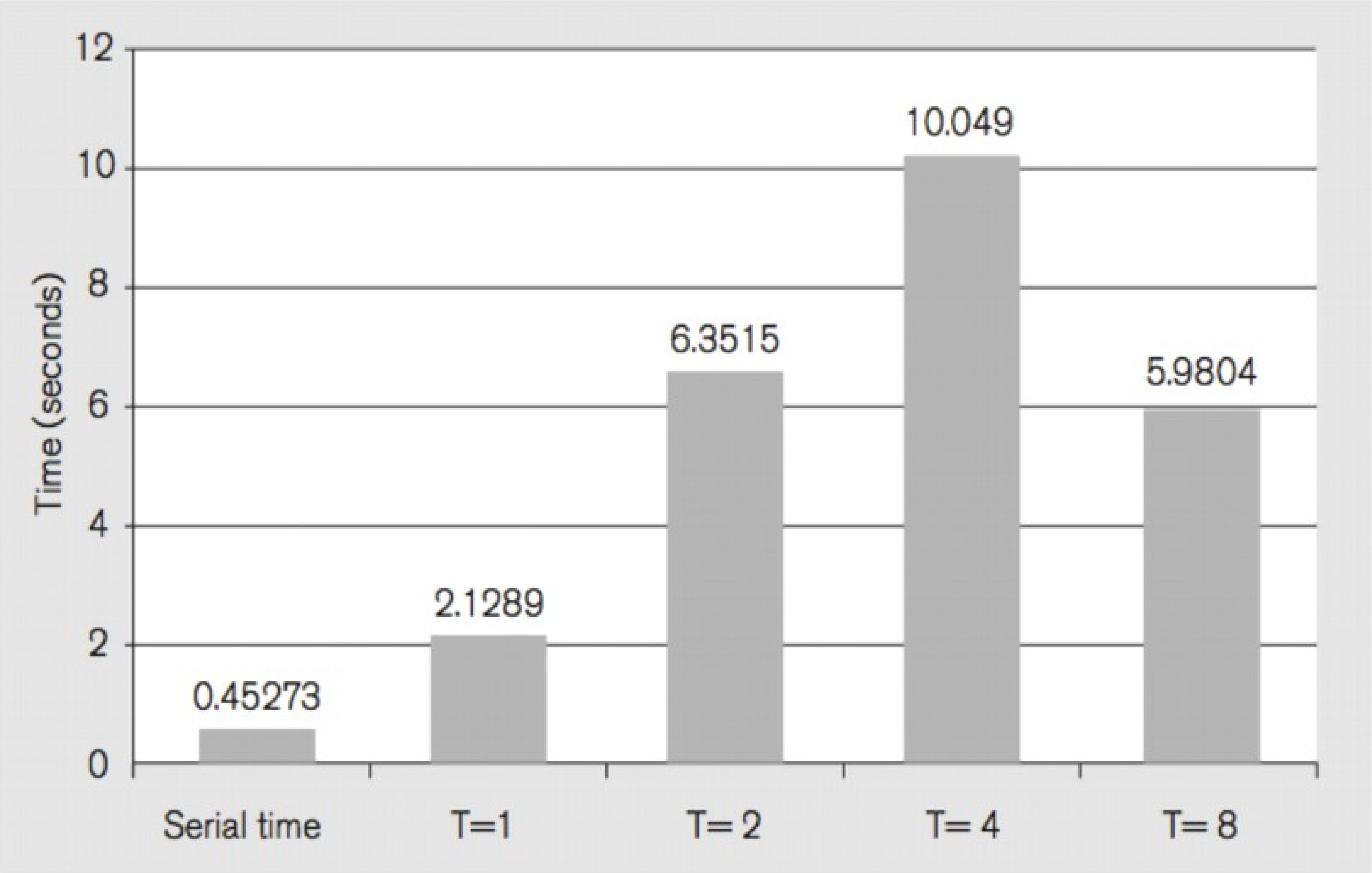
The easiest ways to avoid conflicts in critical sections is to grant access one thread at a time: *mutex* (mutual exclusion)

Mutex



```
void count3s_thread(int id){
  int i;
  int length_per_thread = length/t;
  int start = id*length_per_thread;

  for (i=start;i<start+length_per_thread;i++){
    if (3 == array[i]){
      mutex_lock(m);
      count++;
      mutex_unlock(m);
    } /* end if */
  } /* end for i */

  return count;
} /* end count3s_thread */
```

count3s timing



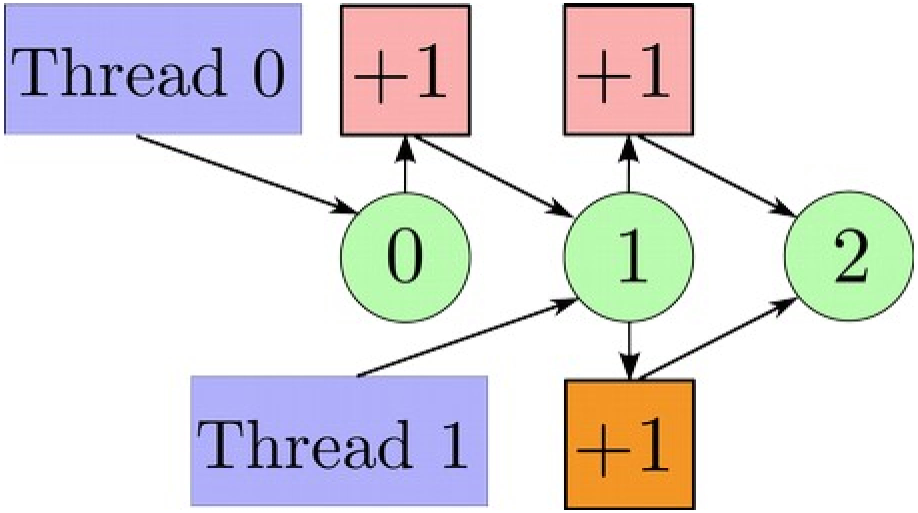
Data Hazards

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

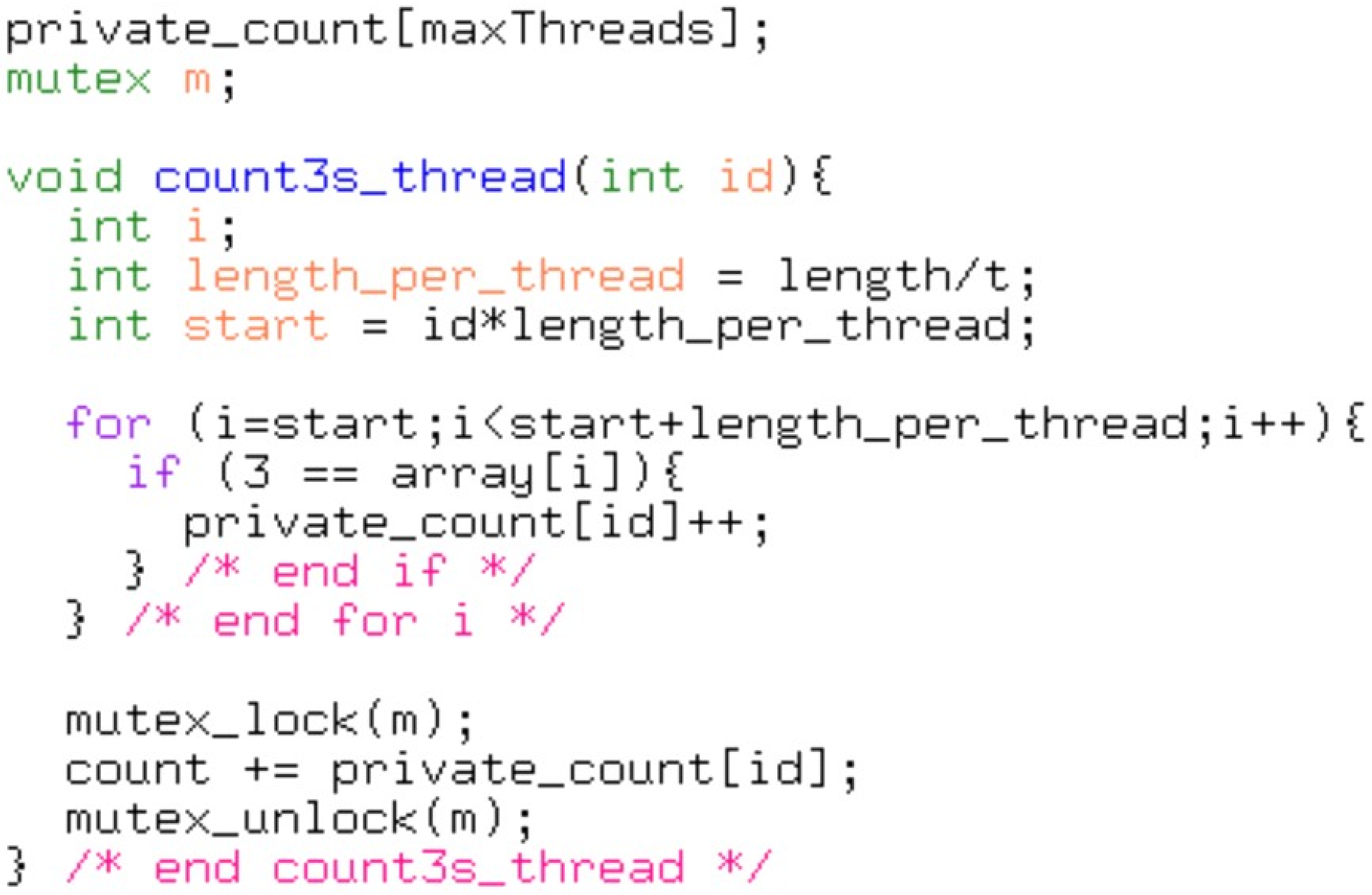
Tricky to debug since the result depends on the timing between concurrent threads: unpredictable!

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)

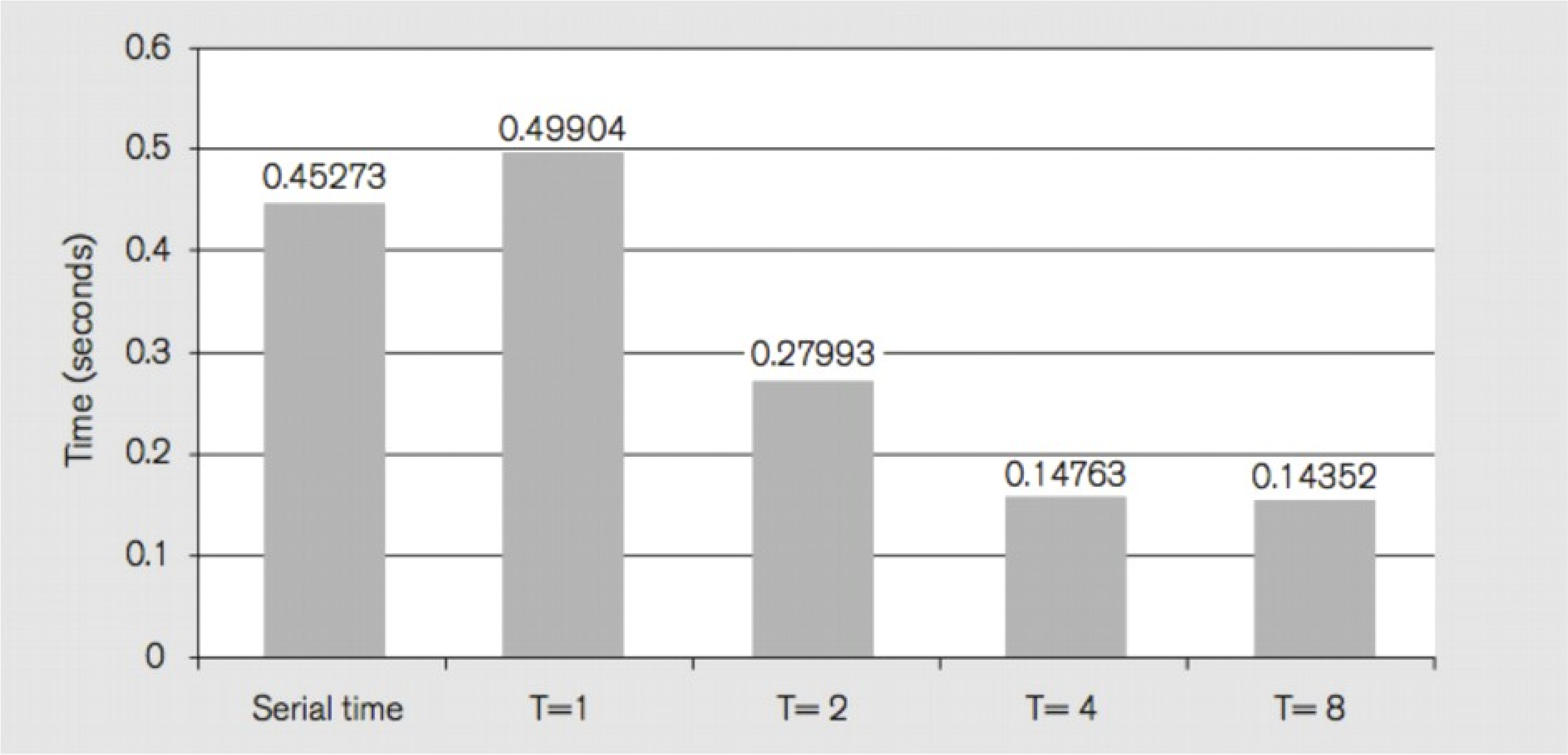


Granularity



```
private_count[maxThreads];
mutex m;

void count3s_thread(int id){
  int i;
  int length_per_thread = length/t;
  int start = id*length_per_thread;

  for (i=start;i<start+length_per_thread;i++){
    if (3 == array[i]){
      private_count[id]++;
    } /* end if */
  } /* end for i */

  mutex_lock(m);
  count += private_count[id];
  mutex_unlock(m);
} /* end count3s_thread */
```

Granularity



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:

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

Amdahl's Law (ctd.)

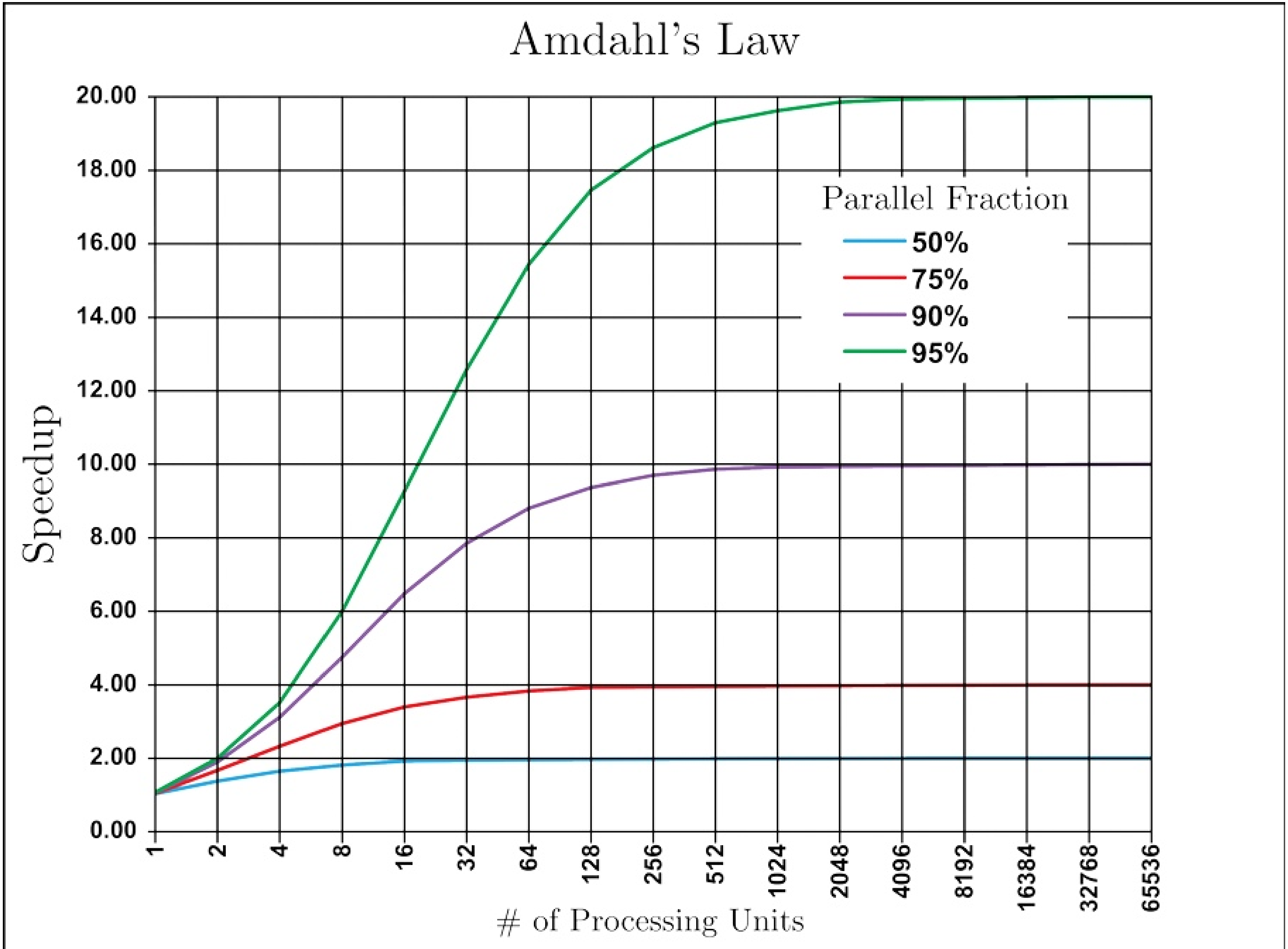
- The speedup becomes:

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

- And the maximum possible speedup for infinite PUs

$$S_{max} \equiv \lim_{p \to \infty} S(p, f) = \frac{T_s}{f \cdot T_s} = \frac{1}{f}$$

Amdahl's Law (ctd.)



Mitigating Amdahl's Law

- Many times, the increase of the size of a problem does not correspond to a growth of the sequential part
- Increase the size of the problem to increase the opportunities for parallelization

Gustafson's Law:

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

- In the hypothesis above:

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

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

Conclusion

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 level
3. Tune core software for the specific architecture

Think, think, think!

- 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
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part ;-])