An introduction to parallel programming

Paolo Burgio
paolo.burgio@unimore.it

Definitions

- Parallel computing
  - Partition computation across different compute engines

- Distributed computing
  - Paritition computation across different machines

Same principle, more general

Outline

- Introduction to "traditional" programming
  - Writing code
  - Operating systems
  - …
- Why do we need parallel programming?
  - Focus on programming shared memory
- Different ways of parallel programming
  - PThreads
  - OpenMP
  - MPI?
  - GPU/accelerators programming

As a side…

- A bit of computer architecture
  - We will understand why…
  - Focus on shared memory systems

- A bit of algorithms
  - We will understand why…

- A bit of performance analysis
  - Which is our ultimate goal!
  - Being able to identify bottlenecks

Programming basics

Take-aways

- Programming basics
  - Variables
  - Functions
  - Loops
- Programming stacks
  - BSP
  - Operating systems
  - Runtimes
- Computer architectures
  - Computing domains
  - Single processor/multiple processors
  - From single- to multi- to many- core

Why do we need parallel computing?

Increase performance of our machines

› Scale-up

– Solve a "bigger" problem in the same time

› Scale-out

– Solve the same problem in less time

Yes but..

- Why (highly) parallel machines…

- …and not faster single-core machines?

The answer #1 - Money

The answer #2 – the "hot" one

Moore's law

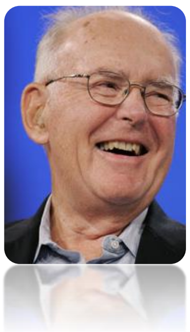
› "The number of transistors that we can pack in a given die area doubles every 18 months"

Dennard's scaling

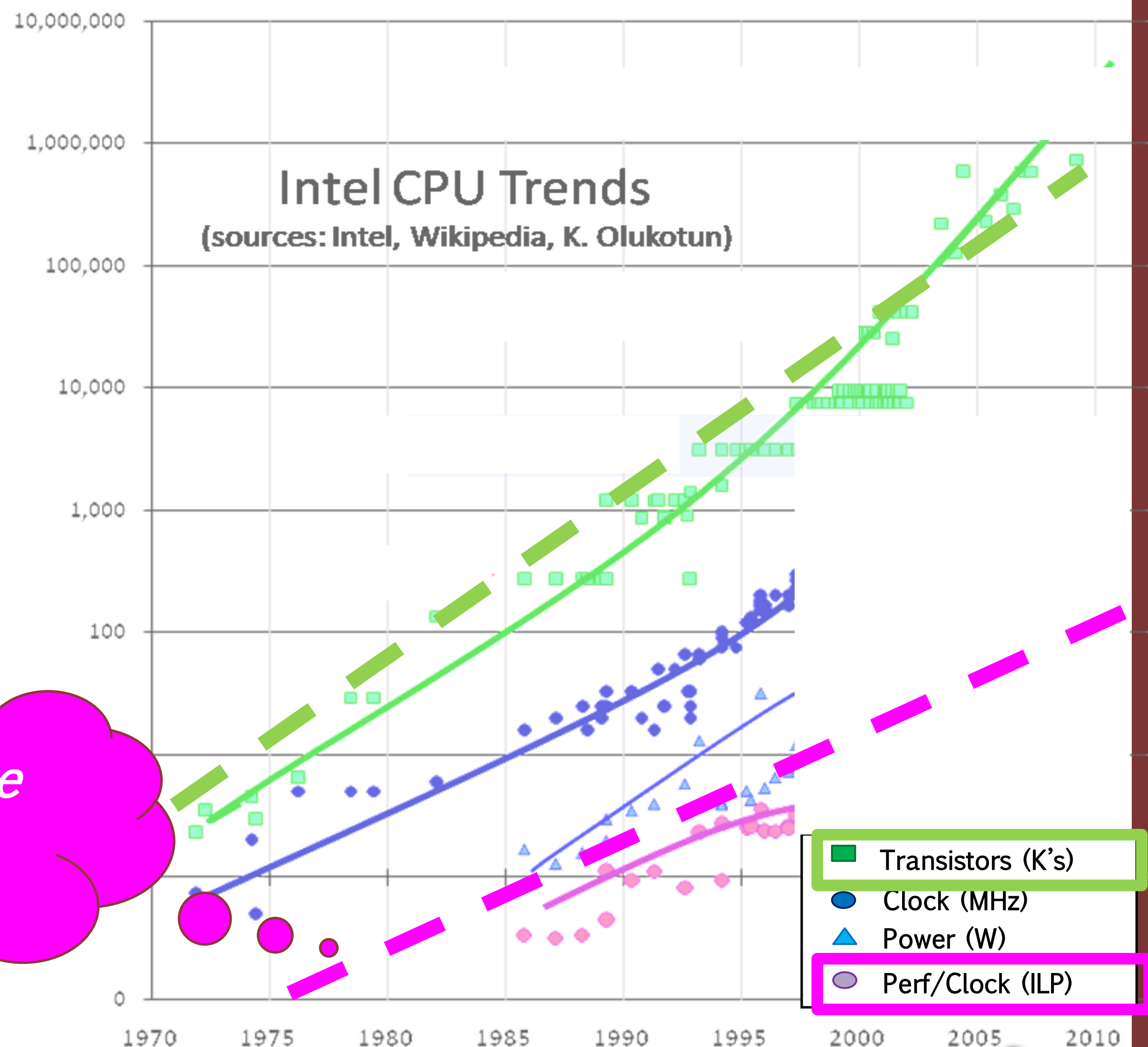
› "performance per watt of computing is growing exponentially at roughly the same rate"

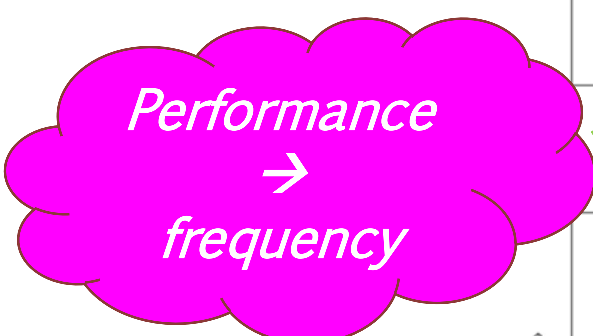
The answer #2 – the "hot" one

- SoC design paradigm



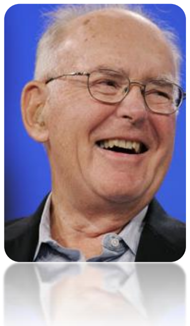
- Gordon Moore
  - His law is still valid, but…





The answer #2 – the "hot" one

› SoC design paradigm



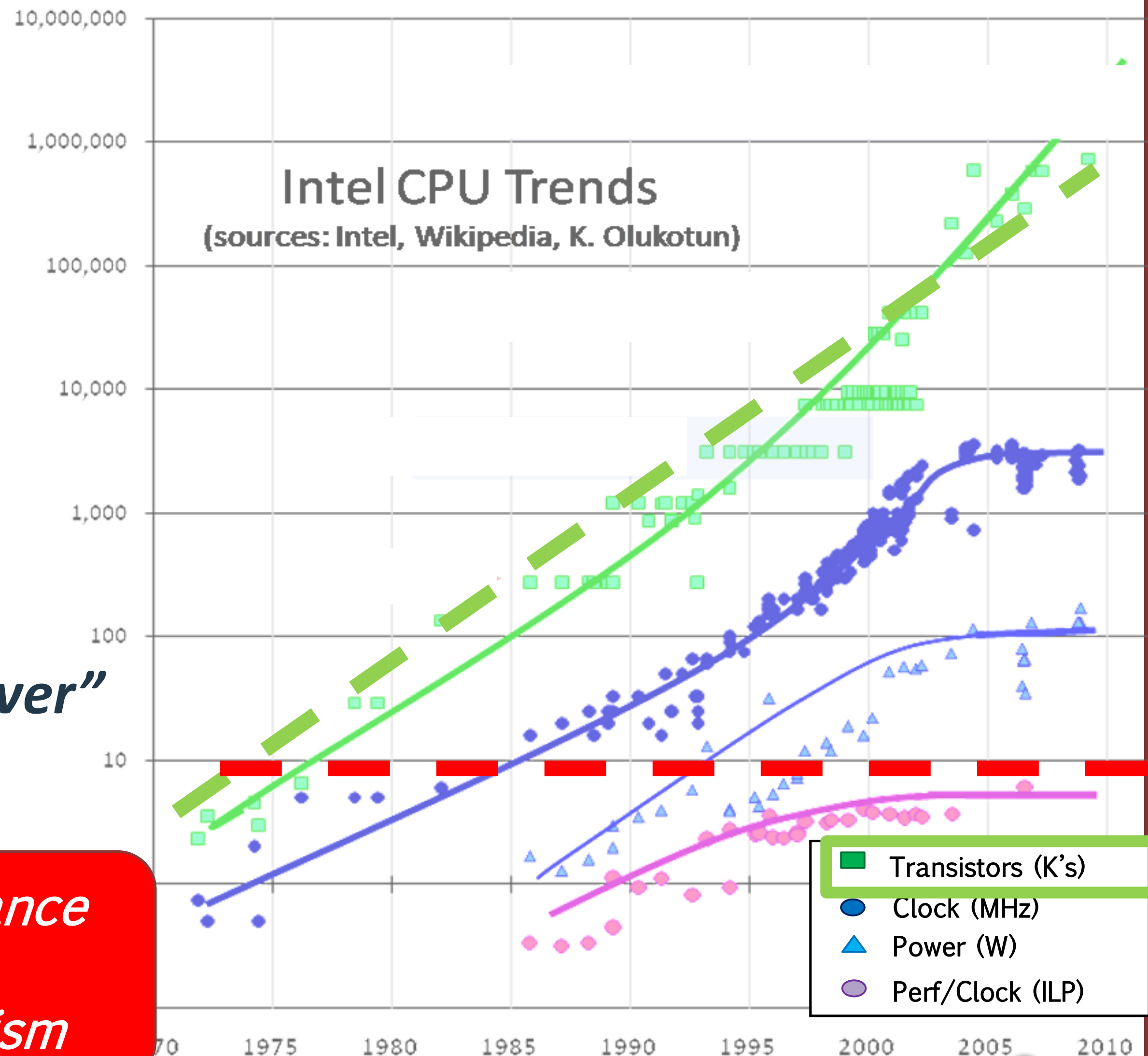
› Gordon Moore

– His law is still valid, but…

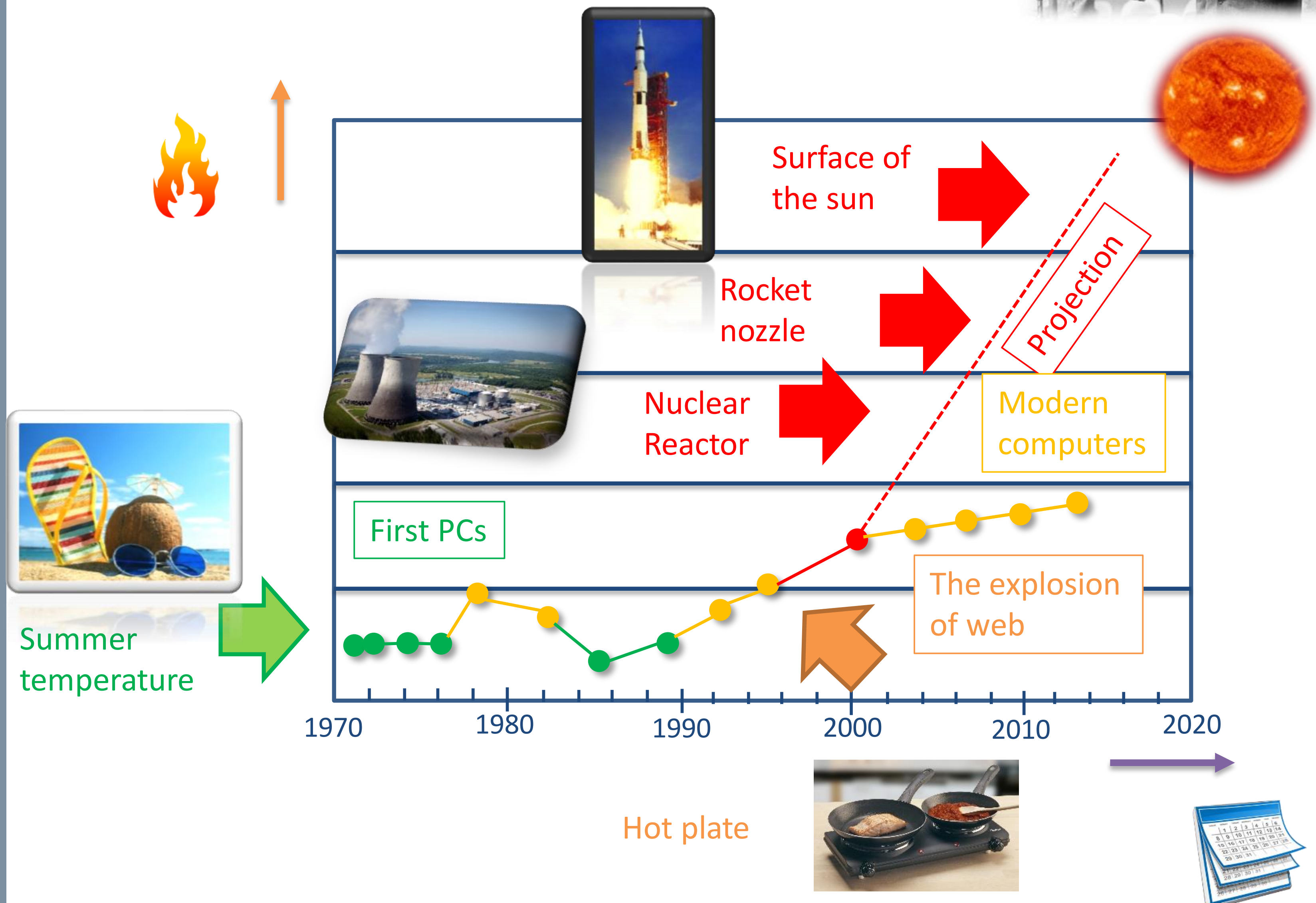
› ***"The free lunch is over"***

– **Herb Sutter, *2005***

Performance → parallelism



In other words…



Instead of going faster..

- ..(go faster but through) parallelism!

Problem #1

- New computer architectures
- At least, three architectural templates

Problem #2

- Need to efficiently program them
- HPC already has this problem!

The problem

- Programmers must know a bit of the architecture!
- To make parallelization effective
- "Let's run this on a GPU. It certainly goes faster" (cit.)

The Big problem

› Effectively programming in parallel is difficult

Brian Kernighan (1942-)
- Researcher, theory of informatics
- Co-authored UNIX and AWK
- Wrote "The C Programming Language" book

"Everyone knows that debugging is twice as hard as writing a program in the first place.

So if you're as clever as you can be when you write it, how will you ever debug it?"

I am *really* sorry guys..

- I will give you code…
- ..but first I need to give you some maths…
- …and then, some architectural principles

Amdahl's Law

Amdahl's law

- A sequential program that takes 100 sec to exec
- Only 95% can run in parallel (it's a lot)
- And.. you are an extremely good programmer, and you have a machine with 1billion cores, so that part takes 0 sec
- So,

$$T_{par} = 100_{sec} - 95_{sec} = 5_{sec}$$

$$Speedup = \frac{100_{sec}}{5_{sec}} = 20x$$

…20x, on one billion cores!!!

Computer architecture

Step-by-step

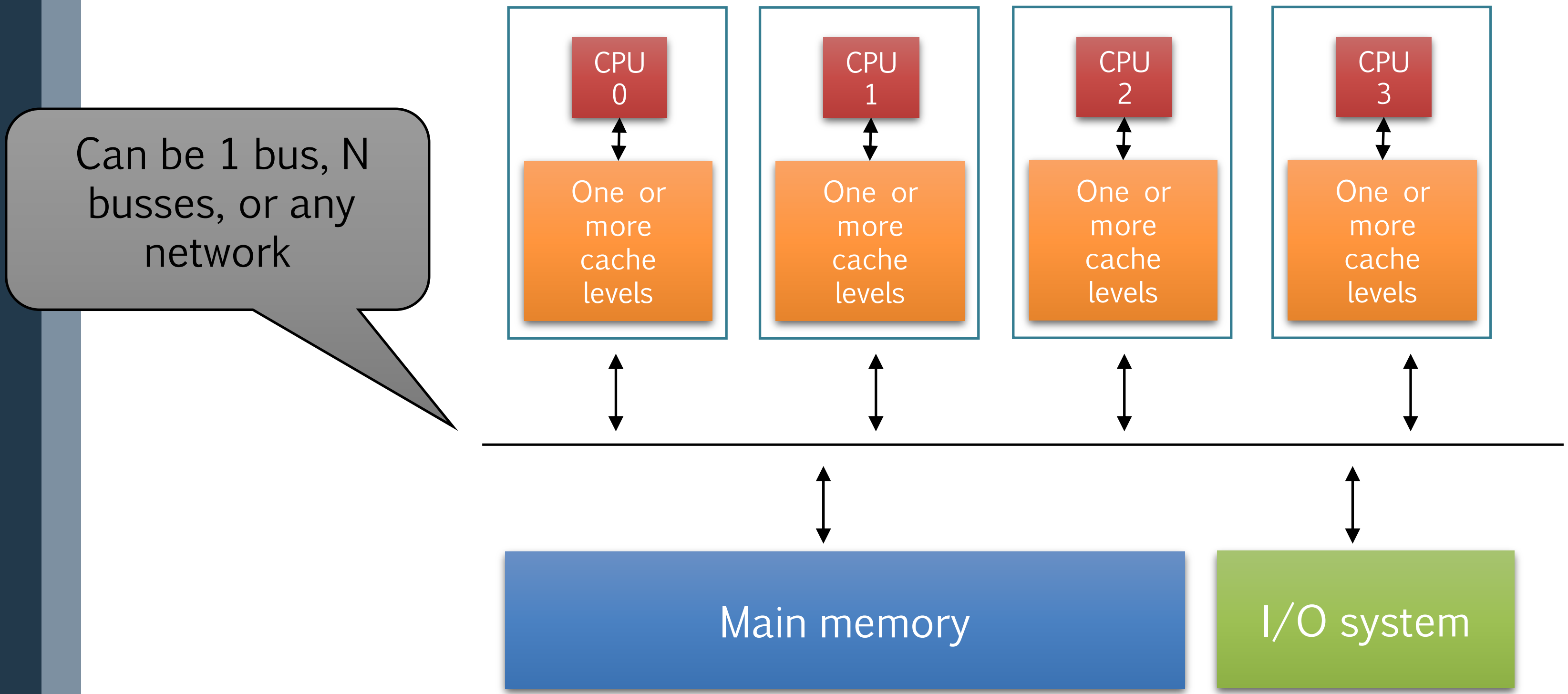
1. "Traditional" multi-cores
   - Typically, shared-memory
   - Max 8-16 cores
   - This laptop

2. Many-cores
   - GPUs but not only
   - Heterogeneous architectures

3. More advanced stuff
   - Field-programmable Gate Arrays
   - Neural Networks

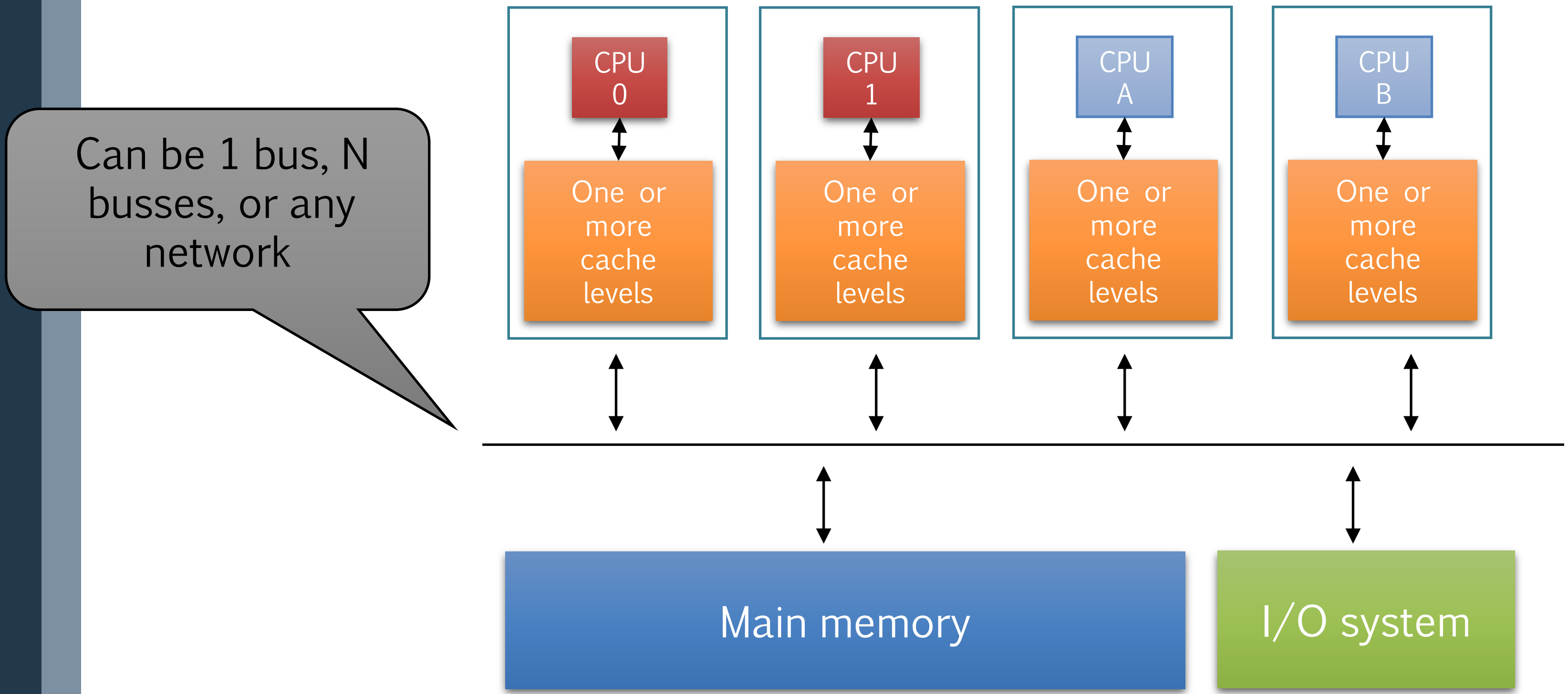
Symmetric multi-processing

- Memory: centralized with bus interconnect, I/O
- Typically, multi-core (sub)systems
  - Examples: Sun Enterprise 6000, SGI Challenge, Intel (this laptop)



Asymmetric multi-processing

- Memory: centralized with uniform access time (UMA) and bus interconnect, I/O
- Typically, multi-core (sub)systems
  - Examples: ARM Big.LITTLE, NVIDIA Tegra X2 (Drive PX)



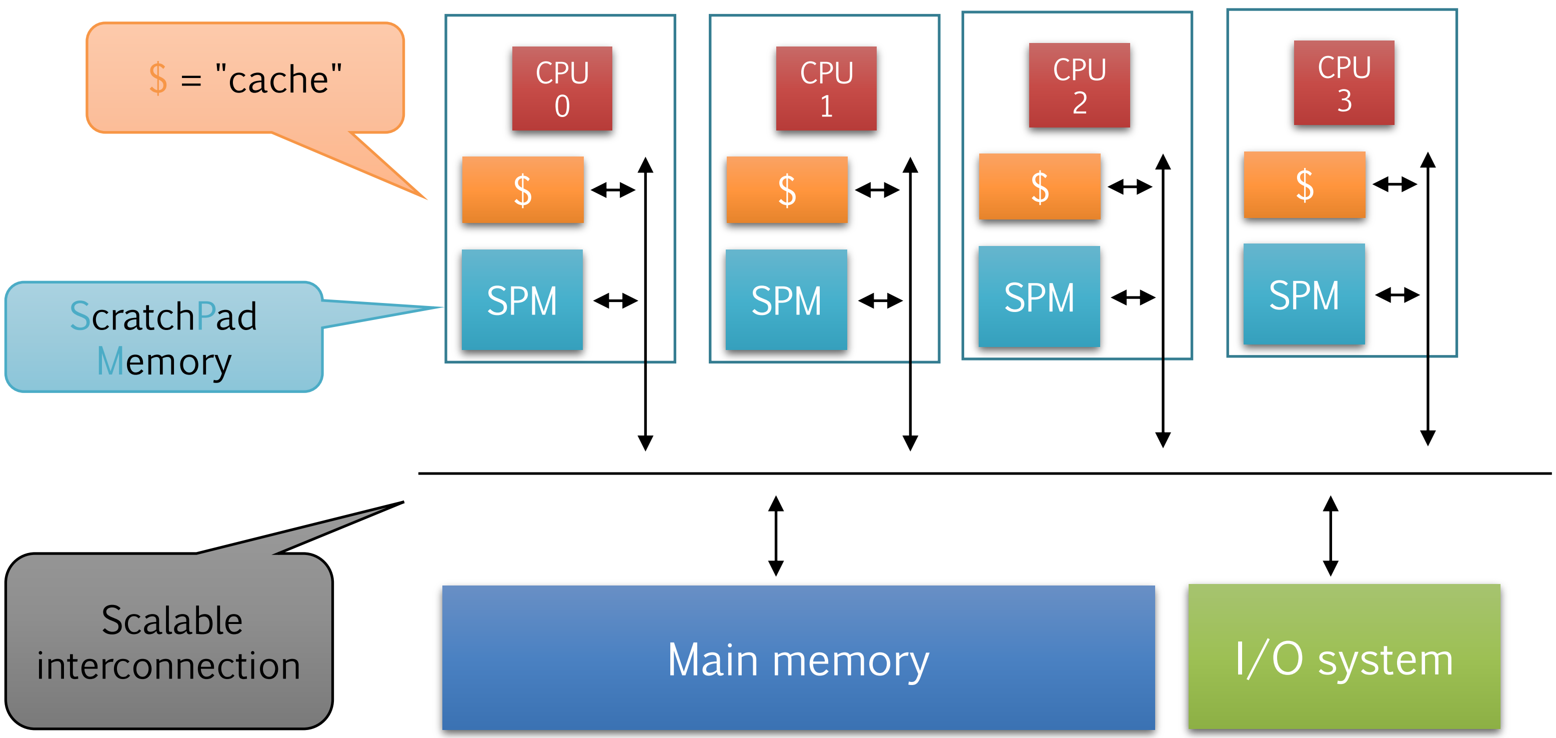
SMP – distributed shared memory

› Non-Uniform Access Time - NUMA

› Scalable interconnect

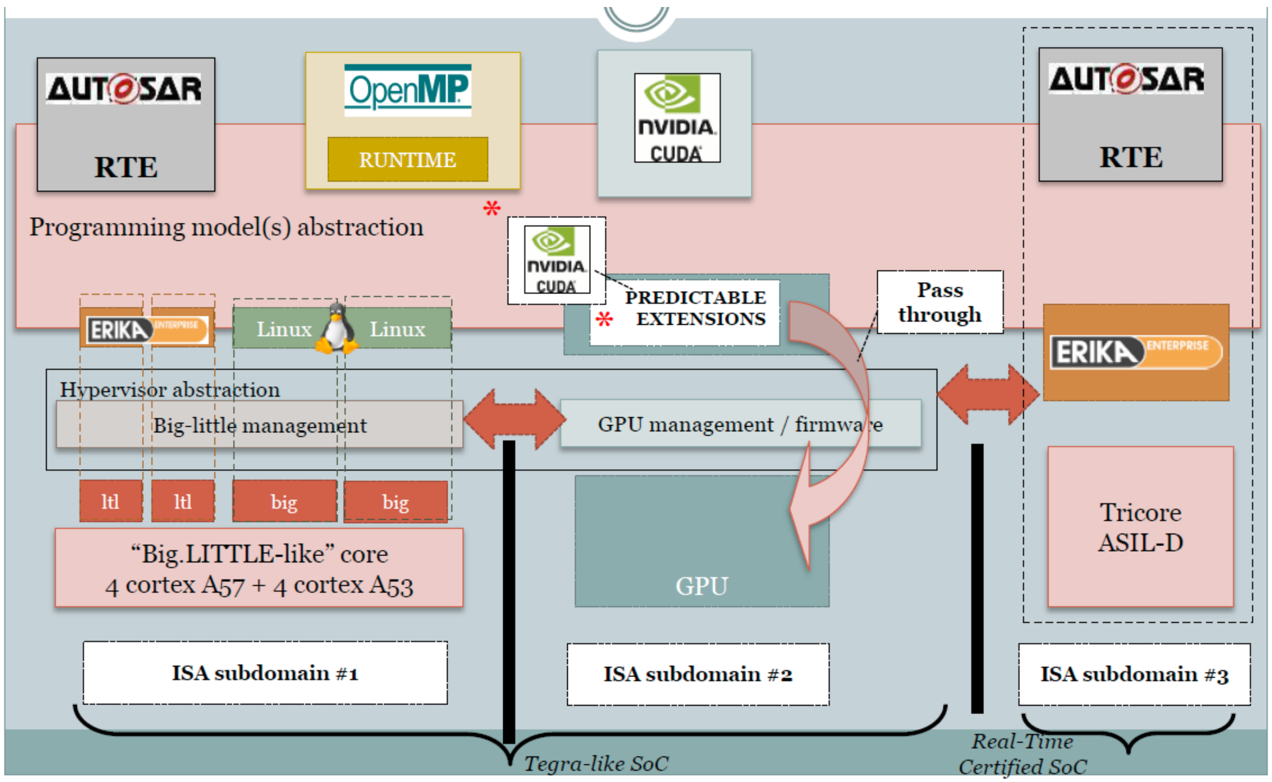
– Typically, many cores

– Examples: embedded accelerators, GPUs



Go complex: NVIDIA's Tegra

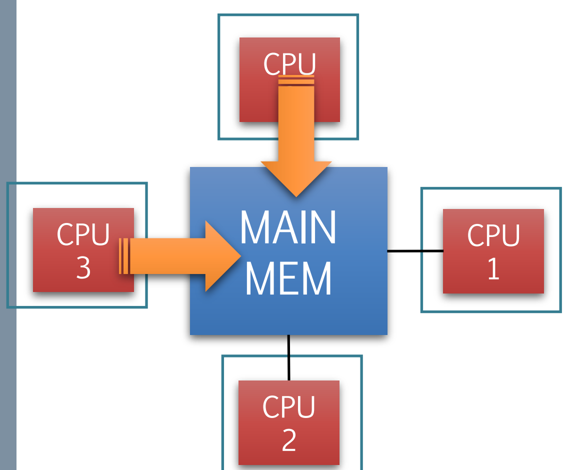
- Complex heterogeneous system
  - 3 ISAs
  - 2 subdomains
  - Shmem between Big.SUPER host and GP-GPU



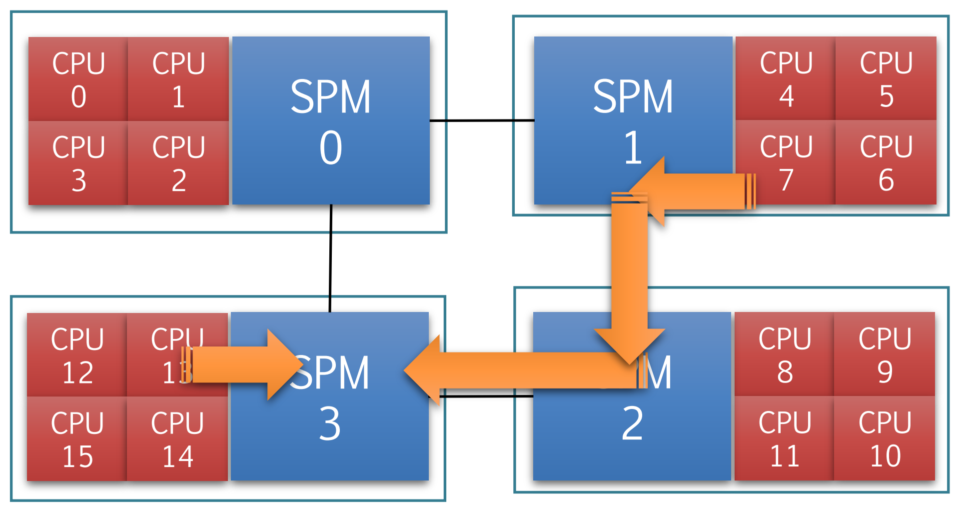
UMA vs. NUMA

- Shared mem: every thread can access every memory item
  - (Not considering security issues…)
- Uniform Memory Access (UMA) vs Non-Uniform Memory Access (NUMA)
  - Different access time for accessing different memory spaces

UMA



NUMA

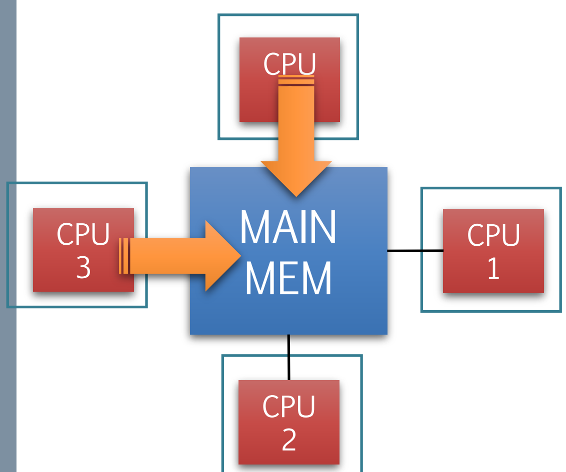


UMA vs. NUMA

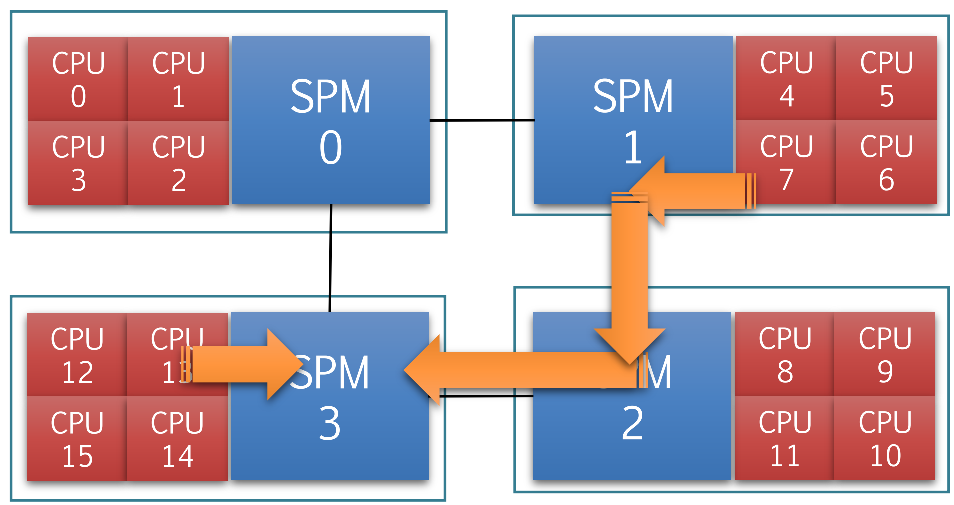
- Shar
  - (
- Unif (NUMA)
  - D

	MEM0	MEM1	MEM2	MEM3
CPU0…3	0 clock	10 clock	20 clock	10 clock
CPU4…7	10 clock	0 clock	10 clock	20 clock
CPU8…11	20 clock	10 clock	0 clock	10 clock
CPU12..15	10 clock	20 clock	10 clock	00 clock

UMA



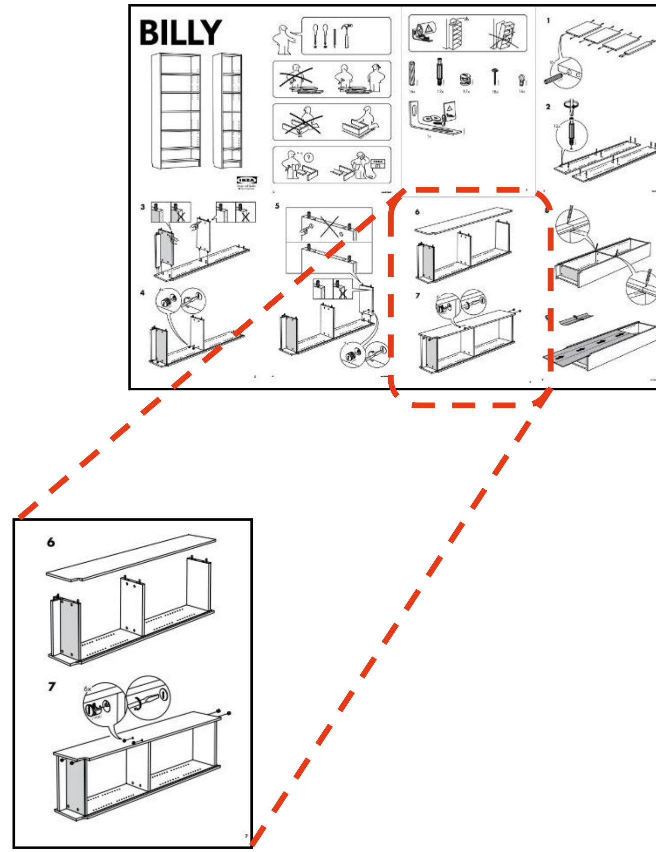
NUMA



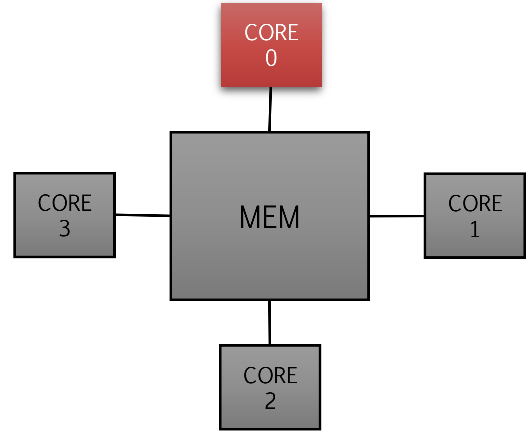
Some definitions

What is...

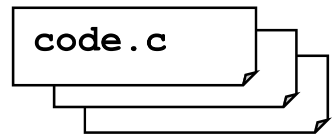
- ..a core?
  - An electronic circuit to execute instruction (=> programs)
- ...a program?
  - The implementation of an algorithm
- ...a process?
  - A program that is executing
- ...a thread?
  - A unit of execution (of a process)
- ..a task?
  - A unit of work (of a program)



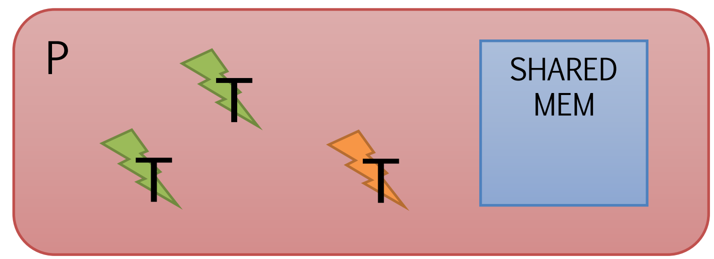
What is...



- ..a core?
  - An electronic circuit to execute instruction (=> programs)
- ...a program?
  - The implementation of an algorithm

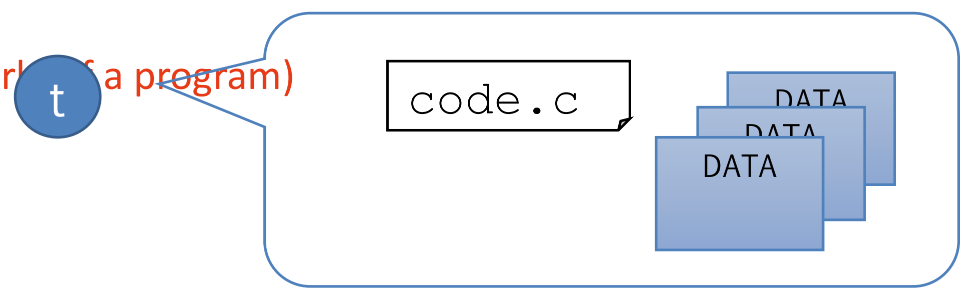


- ...a process?
  - A program that is executing



- ...a thread?

  - A unit of execution (of a process)
- ..a task?
  - A unit of work (of a program)



What is a task?

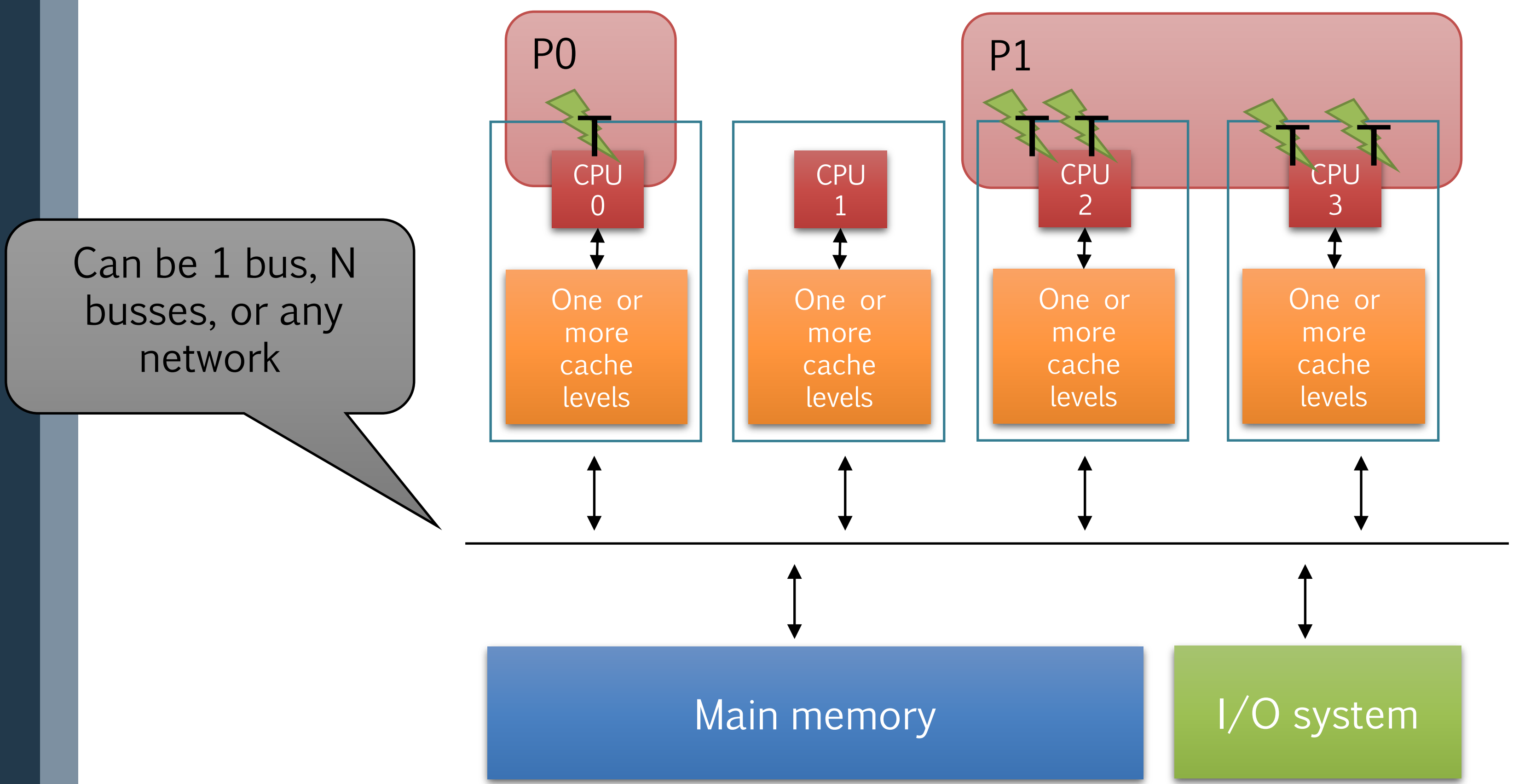
OpenMP
task

Operating
System
task

Real-time
task

Symmetric multi-processing

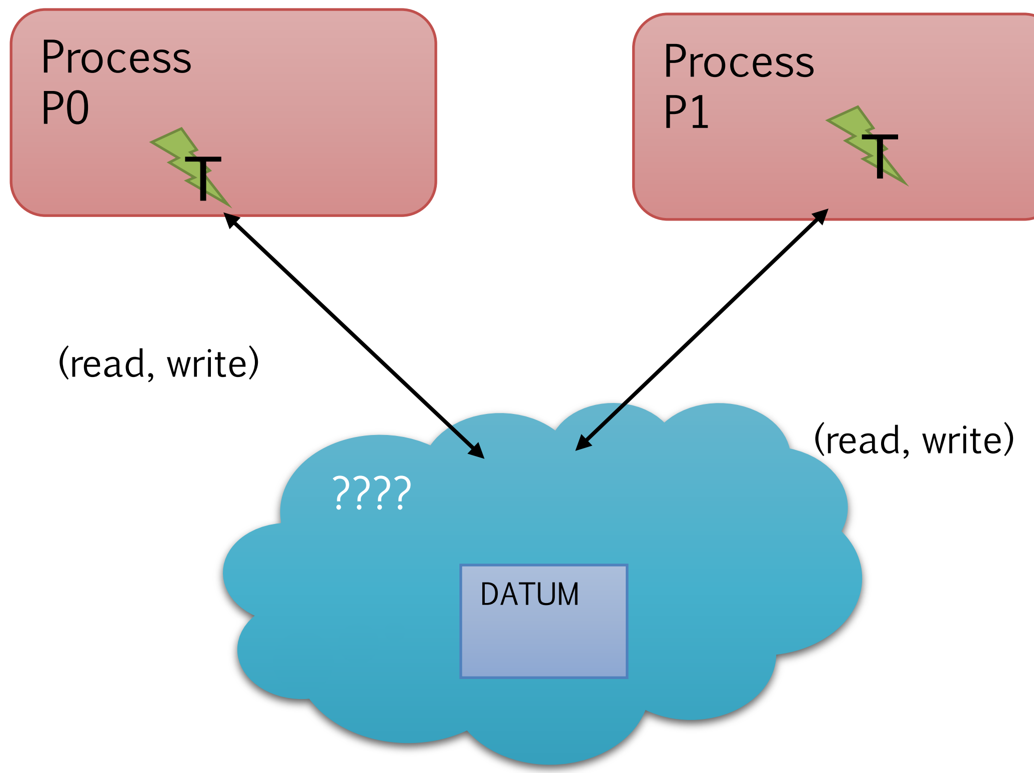
- Memory: centralized with bus interconnect, I/O
- Typically, multi-core (sub)systems
  - Examples: Sun Enterprise 6000, SGI Challenge, Intel (this laptop)



..start simple...

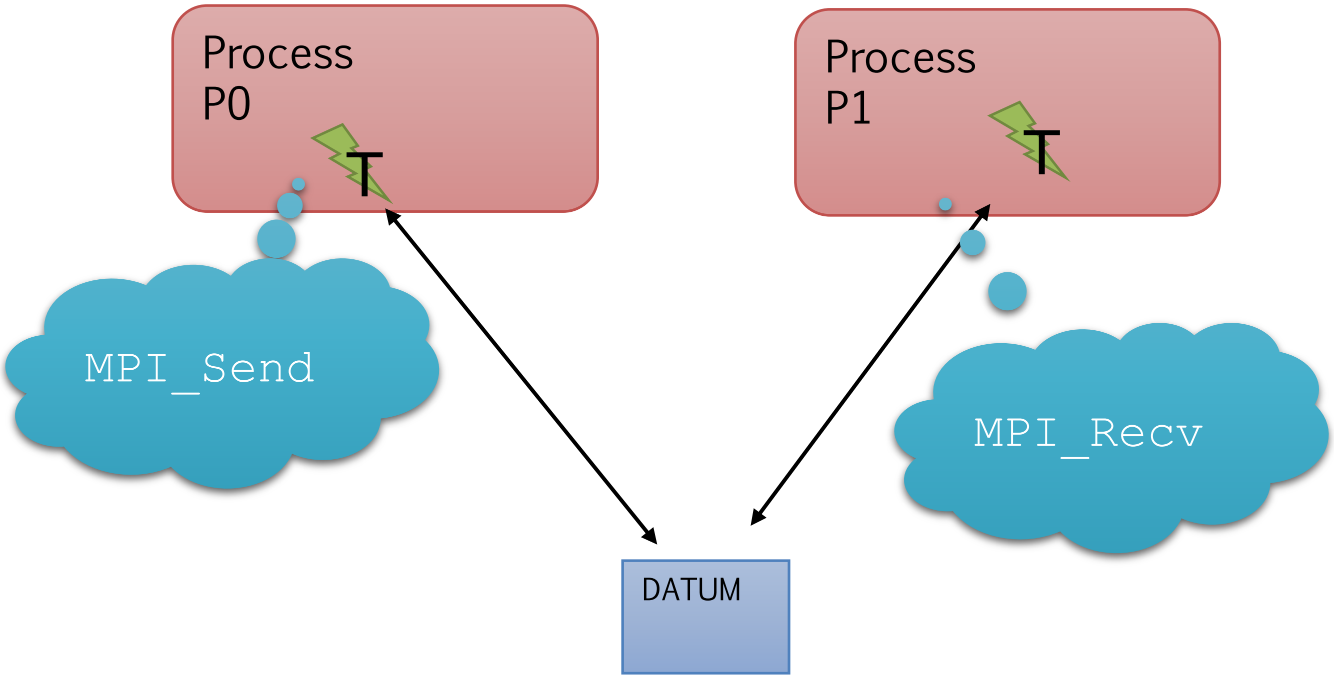
Something you're used to..

- Multiple processes
- That communicate via shared data



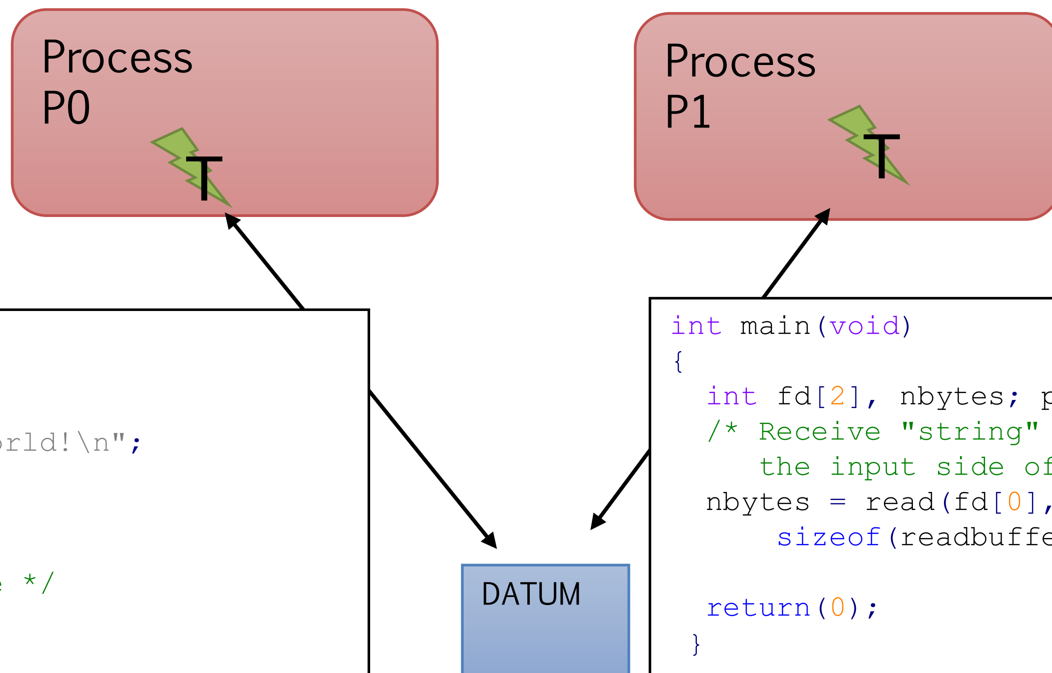
Howto #1 - MPI

- Multiple processes
- That communicate via shared data



Howto #2 – UNIX pipes

- Multiple processes
- That communicate via shared data

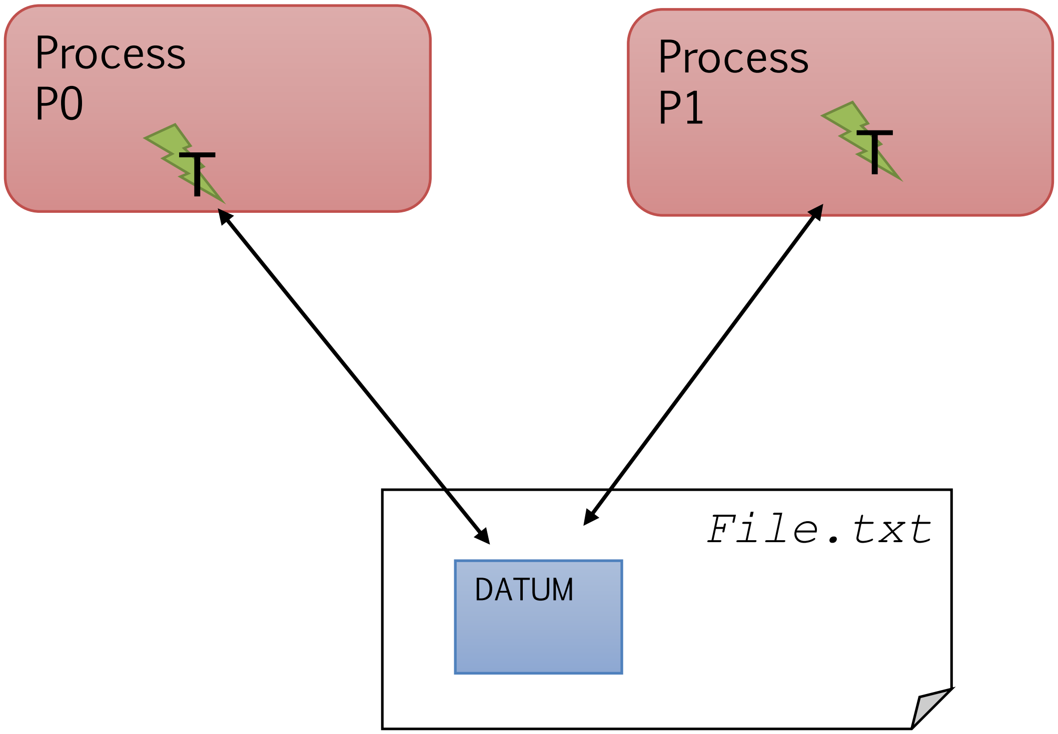



```c
int main(void)
{
  int fd[2], nbytes;
  char string[] = "Hello, world!\n";
  pipe(fd);

  /* Send "string" through
     the output side of pipe */
  write(fd[1], string,
      (strlen(string)+1));
  return(0);
 }
```

```c
int main(void)
{
  int fd[2], nbytes; pipe(fd);
  /* Receive "string" from
     the input side of pipe */
  nbytes = read(fd[0], readbuffer,
       sizeof(readbuffer));

  return(0);
 }
```

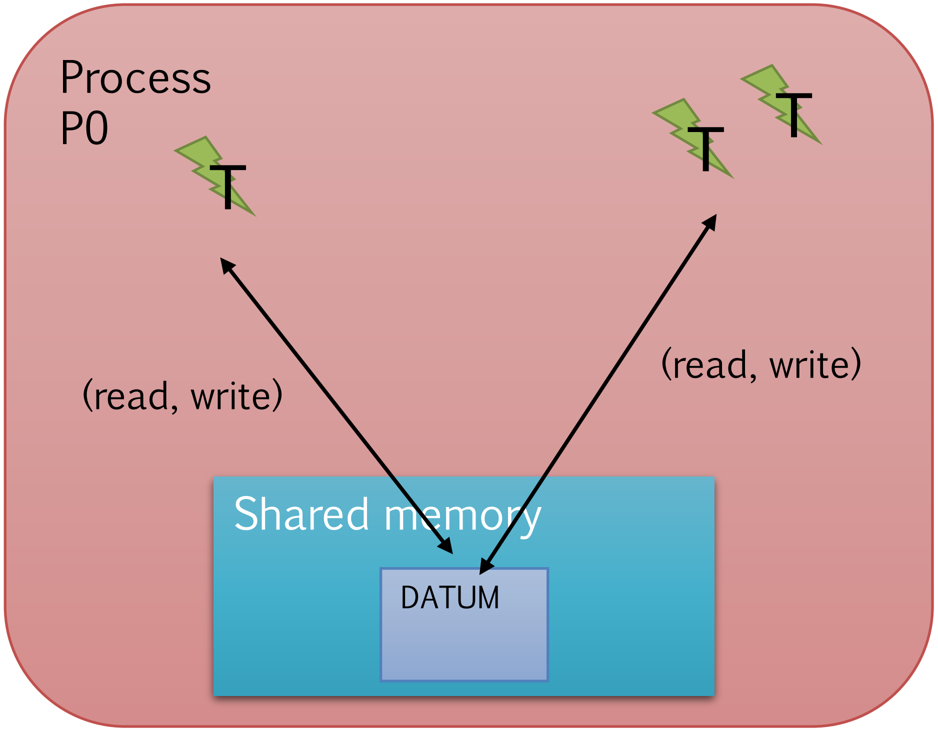
Howto #3 – Files

- Multiple processes
- That communicate via shared data



Shared memory

- Coherence problem
  - Memory consistency issue
  - Data races
- Can share data ptrs
  - Ease-to-use



Useful links

- Course webpage
  - https://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/
- Course GitHub
  - https://github.com/HiPeRT/cp19/
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- A "small blog"
  - http://www.google.com