# Parallel programming

Paolo Burgio
paolo.burgio@unimore.it


Multithreaded programming
Theory
Actual

# Definitions

- Parallel computing
  - Partition computation across different compute engines
  - E.g., PThreads w/Shared mem, but also multi-process on the same machine!

- Distributed computing
  - Paritition computation across different machines
  - E.g., multiprocess (MPI, MQTT) w/message passing

Same principle, more general

# 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

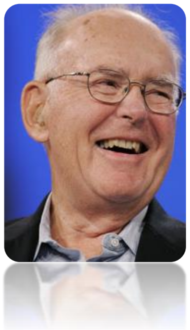
- "The number of transistors that we can pack in a given die area doubles every 18 months"

Dennard's scaling

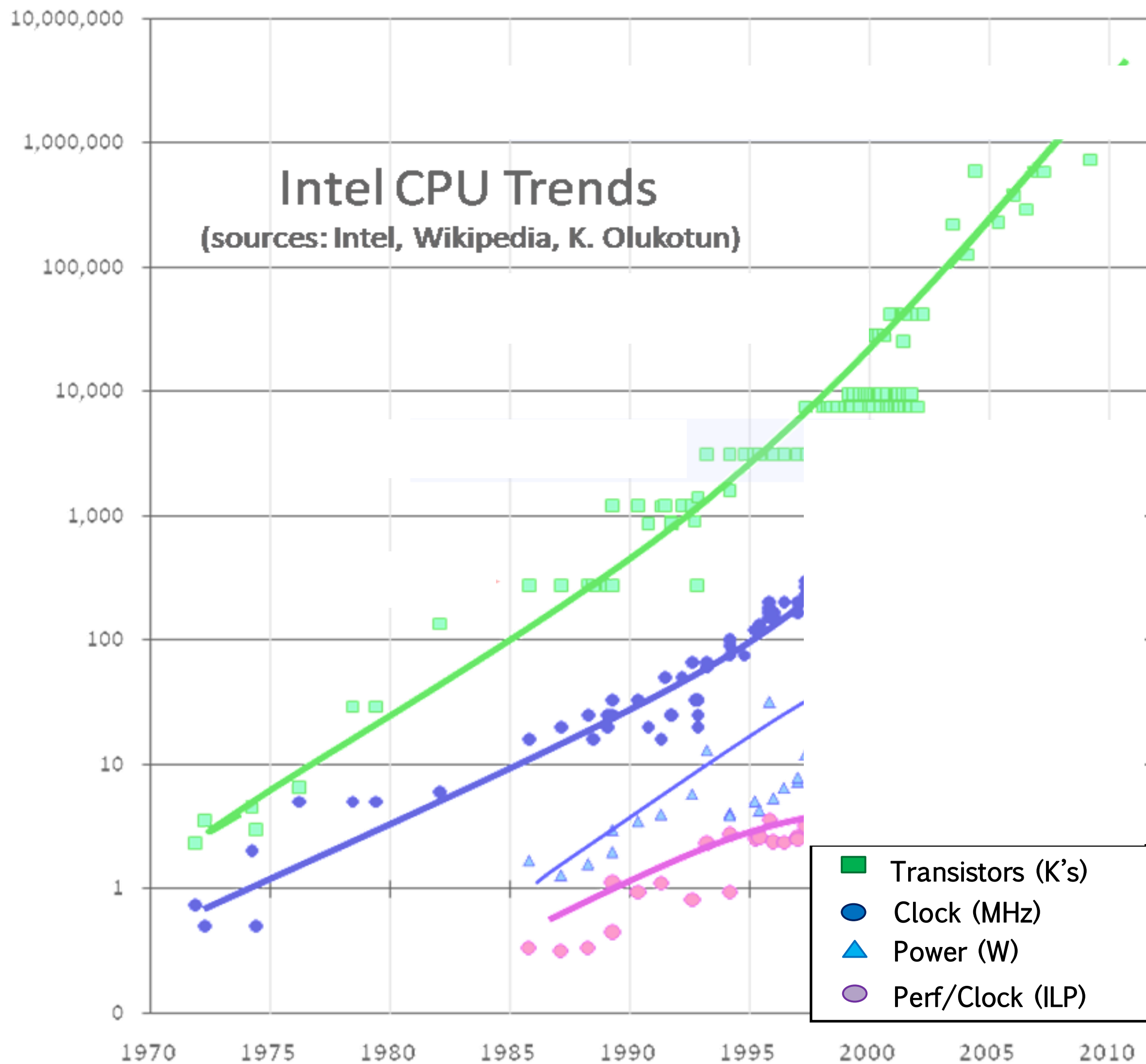
- "performance per watt of computing is growing exponentially at roughly the same rate"

# The answer #2 – the "hot" one

- SoC design paradigm

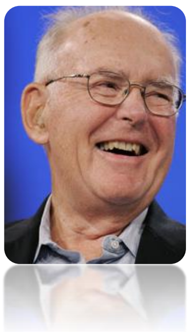


- Gordon Moore
  - His law is still valid, but…

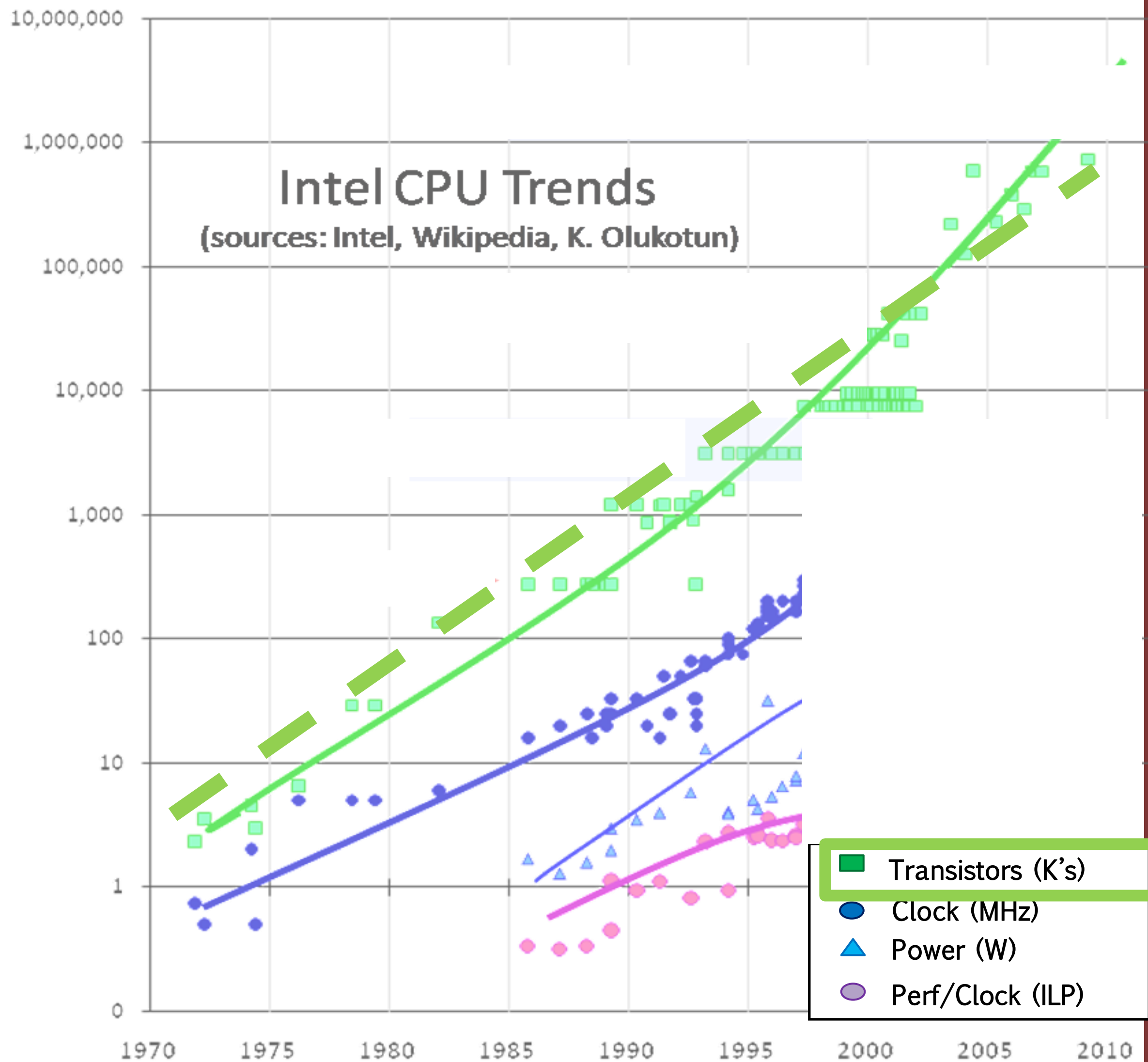


# The answer #2 – the "hot" one

- SoC design paradigm

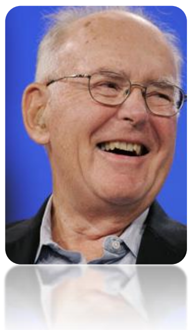


- Gordon Moore
  - His law is still valid, but…

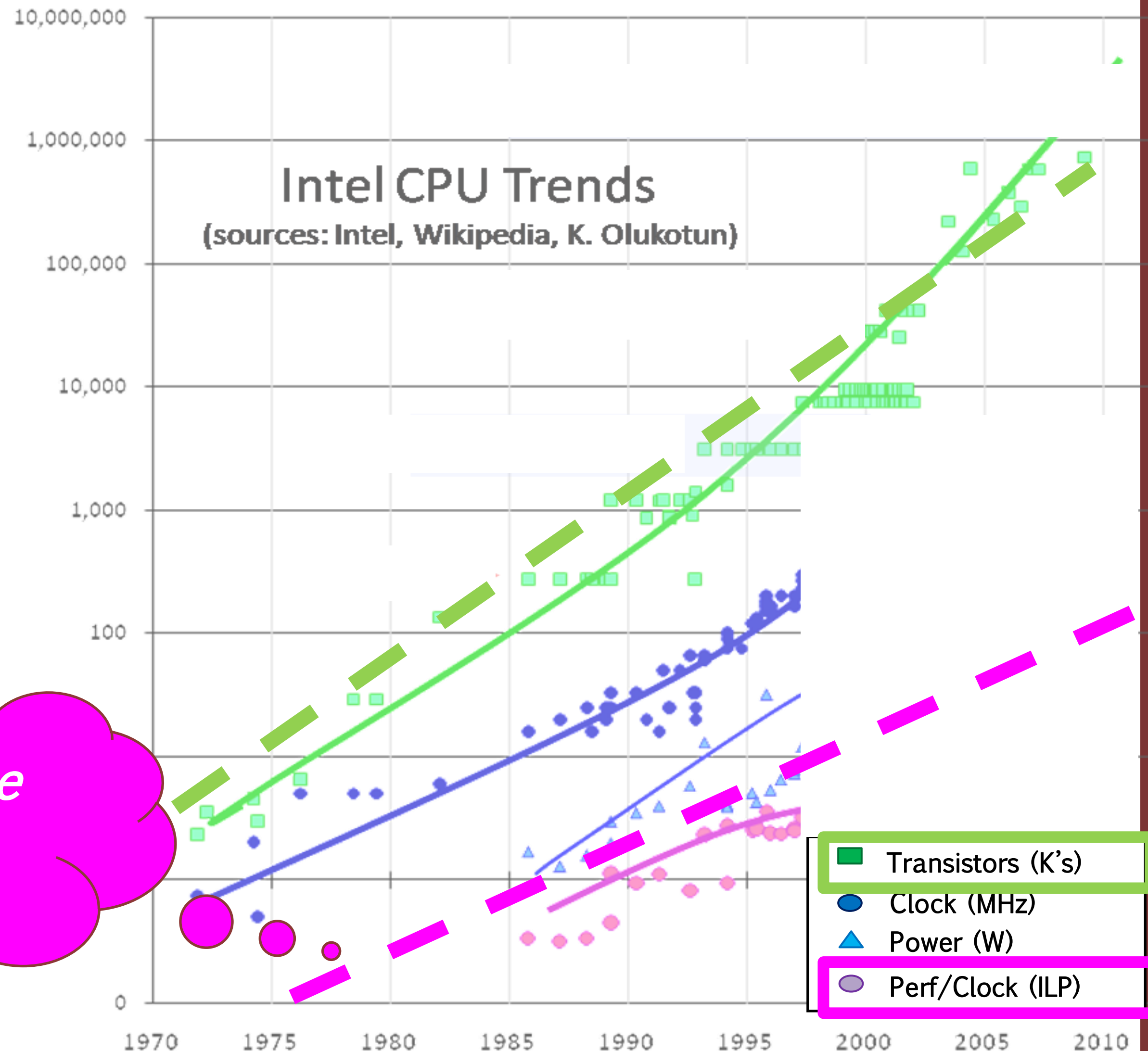


# The answer #2 – the "hot" one

- SoC design paradigm



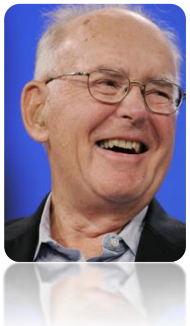
- Gordon Moore
  - His law is still valid, but…



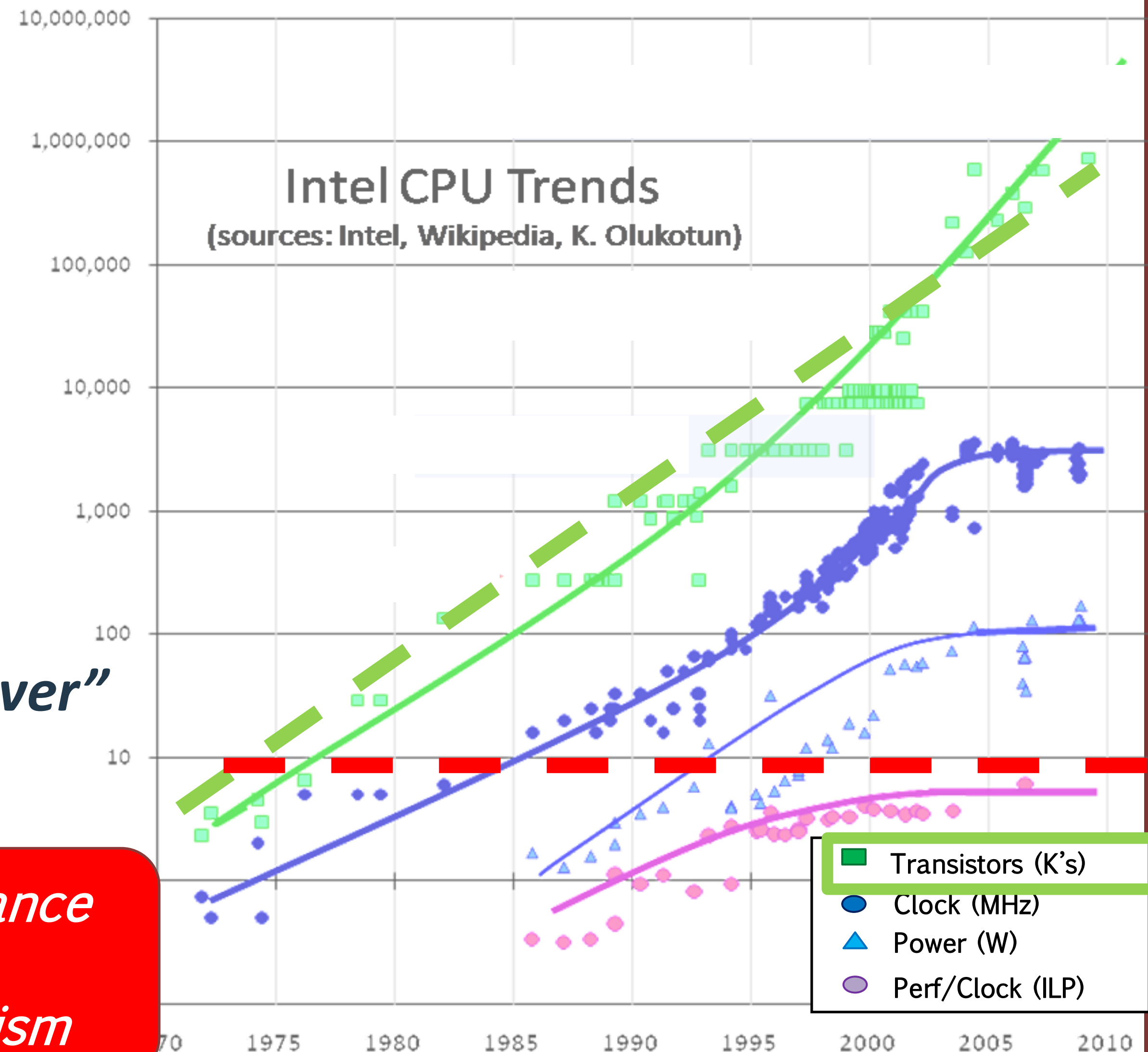

*Performance → frequency*

# The answer #2 – the "hot" one

- SoC design paradigm

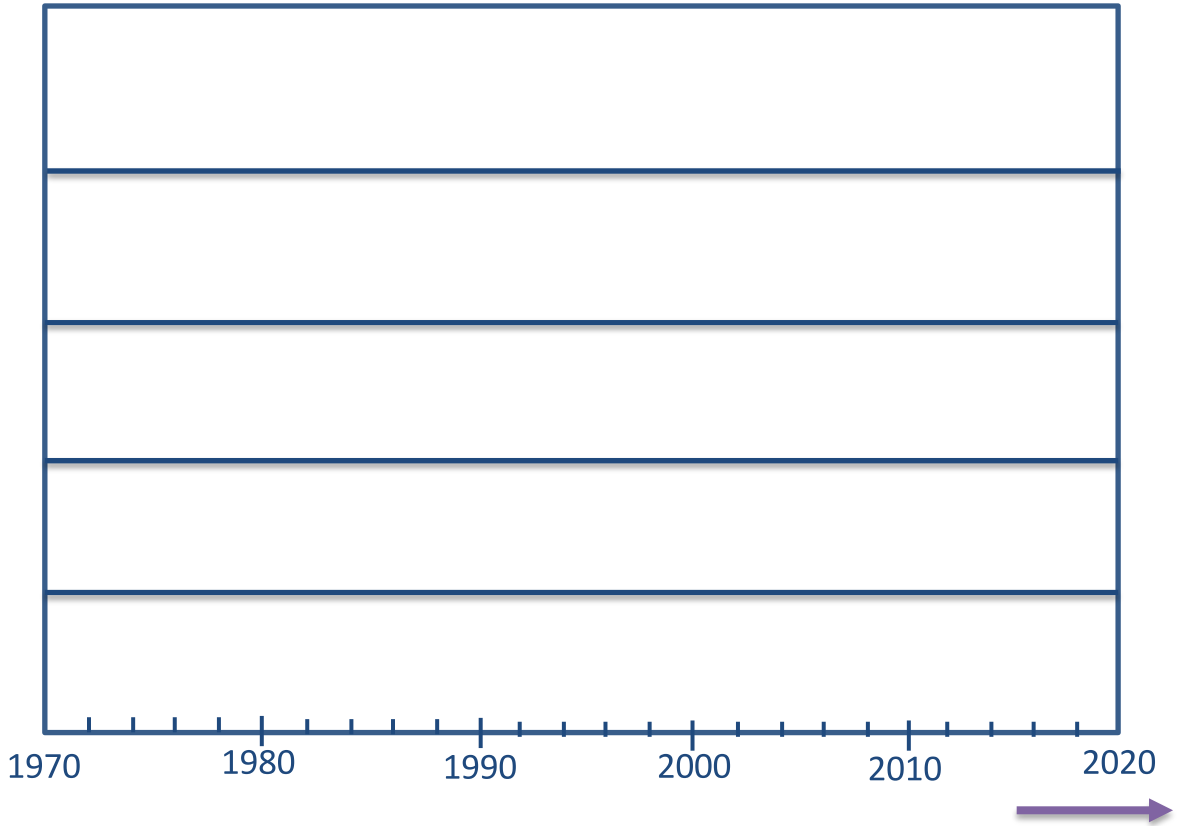


- Gordon Moore
  - His law is still valid, but…
- ***"The free lunch is over"***
  - **Herb Sutter, *2005***



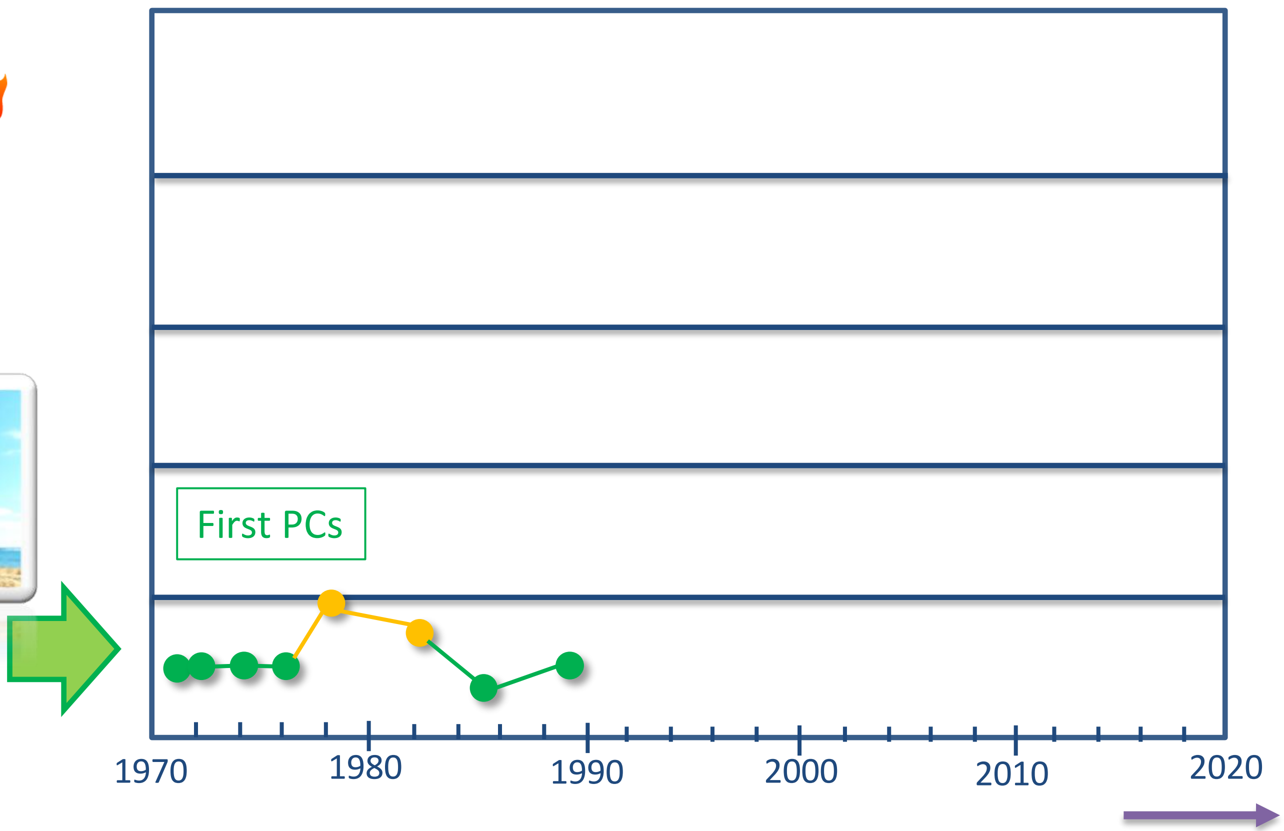

*Performance → parallelism*

# In other words…



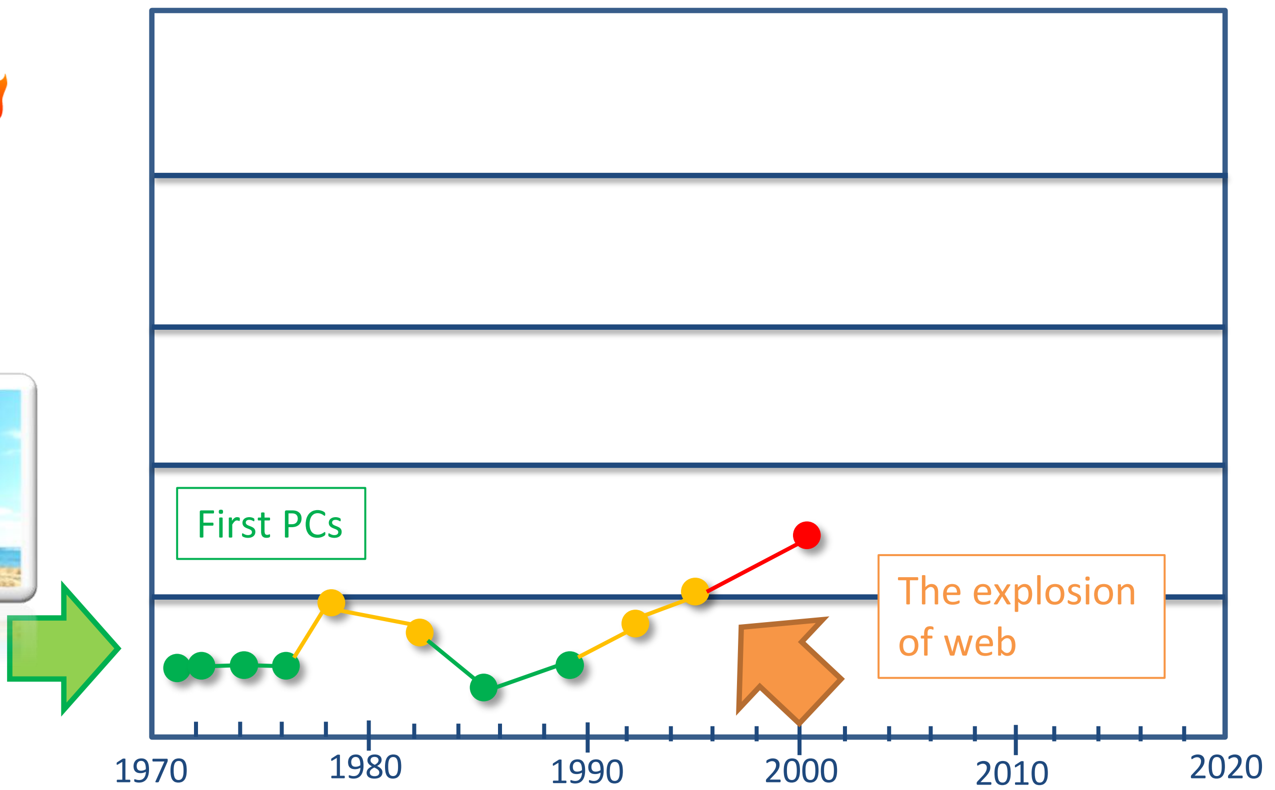
# In other words…

Summer temperature




# In other words…

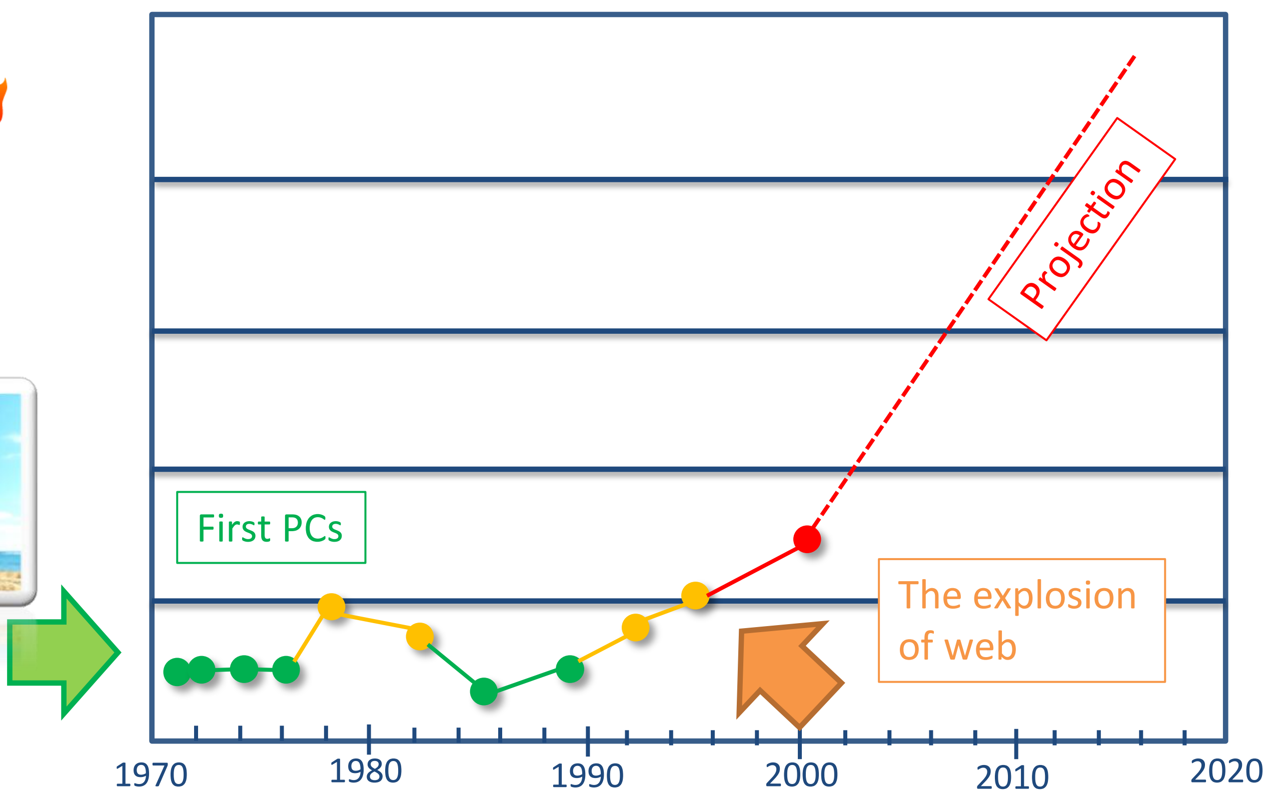
Summer temperature




Hot plate

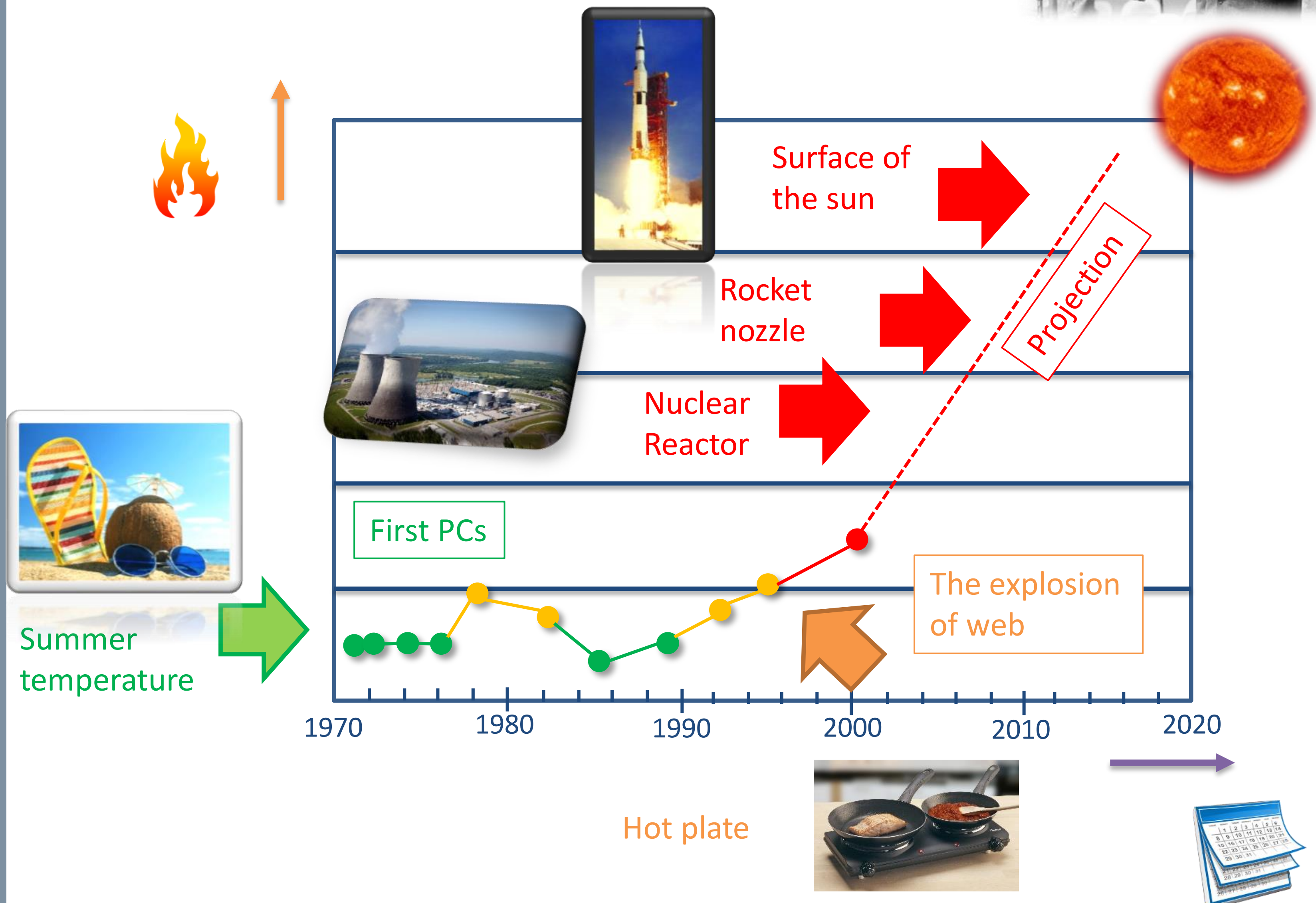
# In other words...

Summer temperature

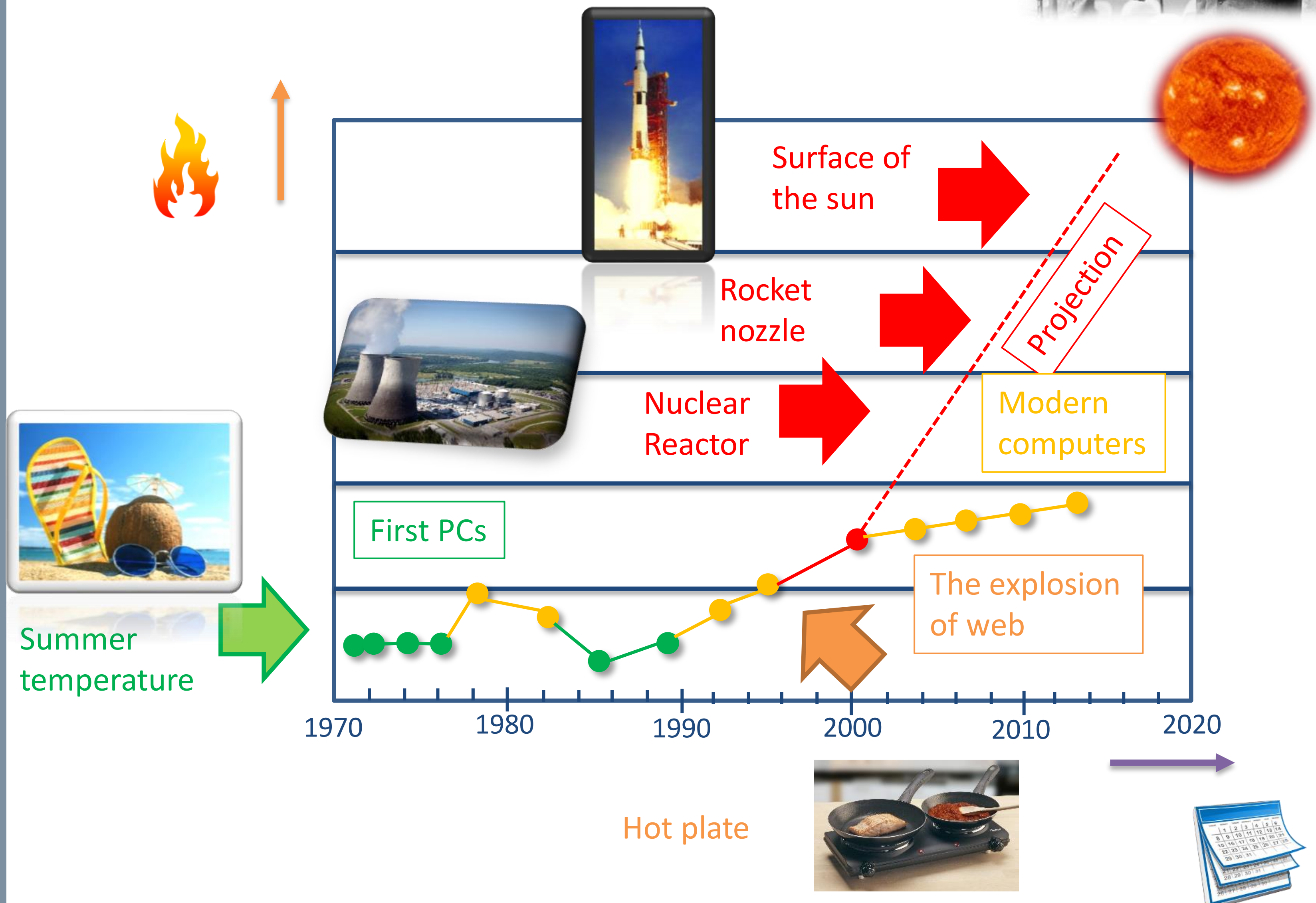



Hot plate

# In other words...



# 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 <u>Big</u> problem

- Effectively programming in parallel is <u>difficult</u>

*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?"

# 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,

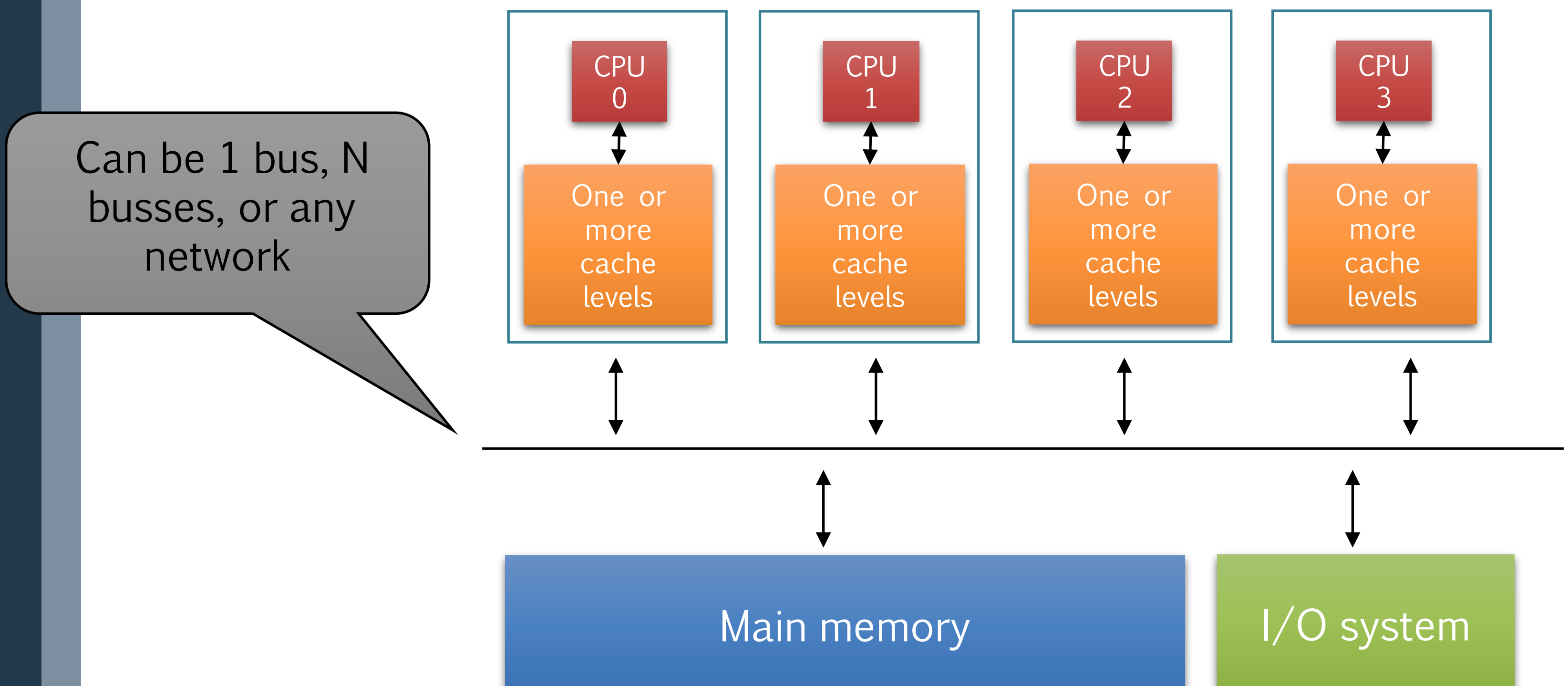
$$T_{par} = 100_{sec} - 95_{sec} = 5_{sec}$$

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

…20x, on one billion cores!!!

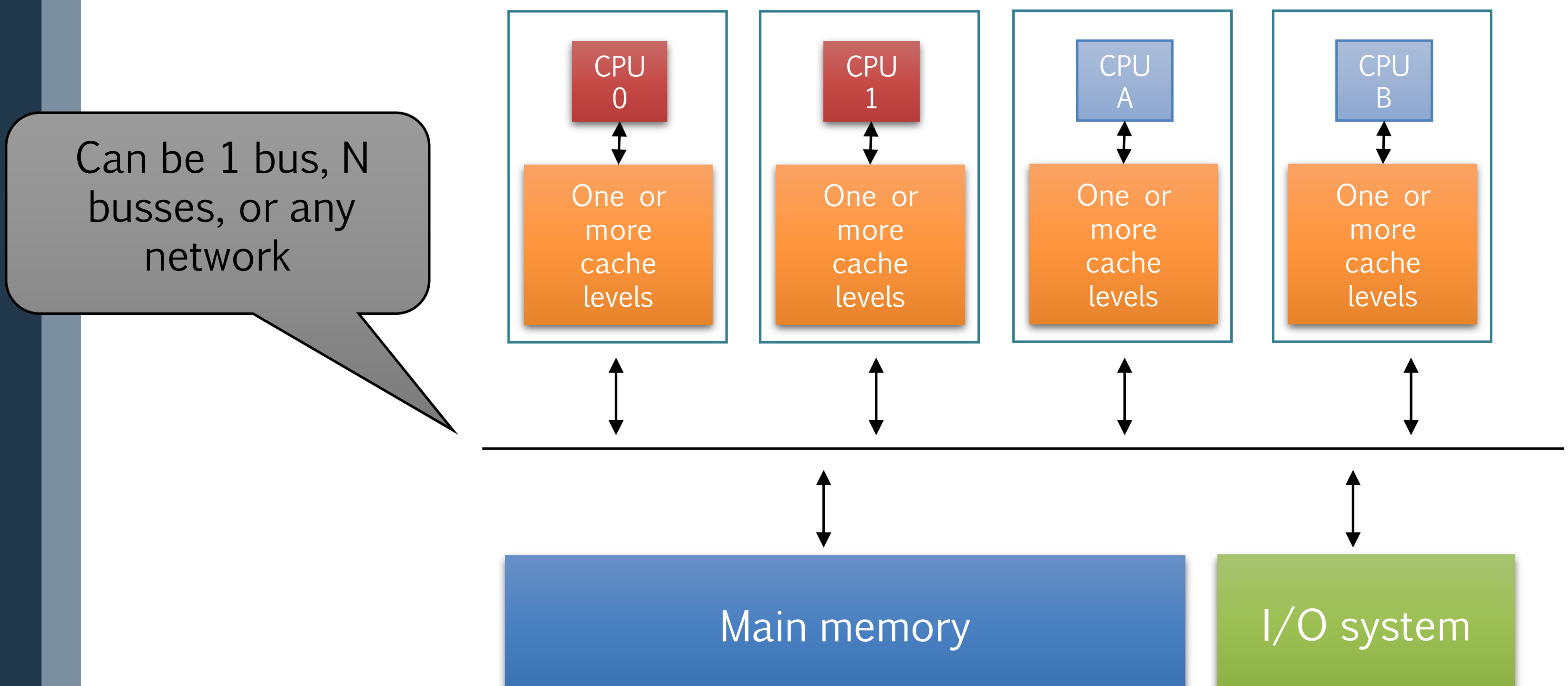
# Symmetric multi-processing

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



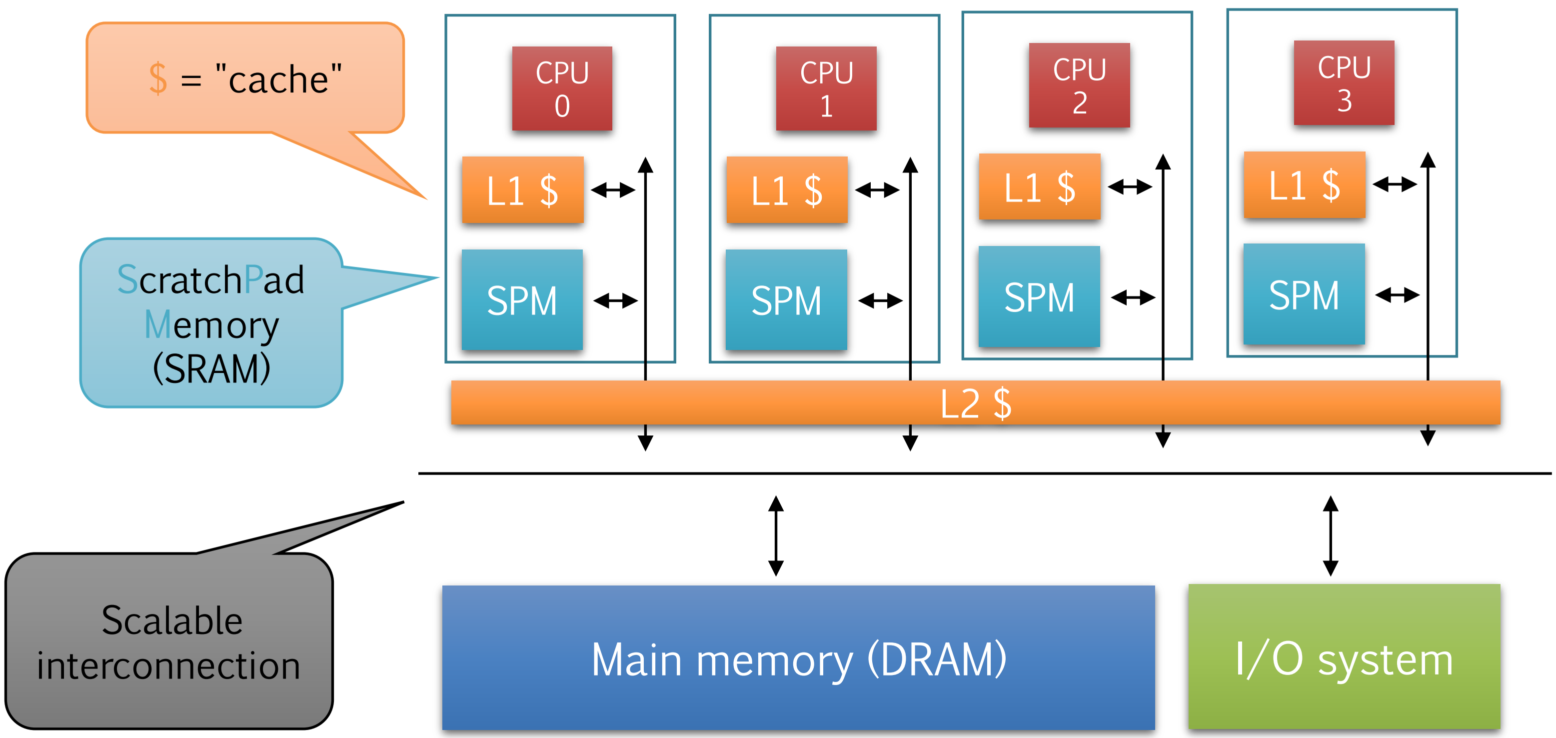
# Asymmetric multi-processing

- 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)



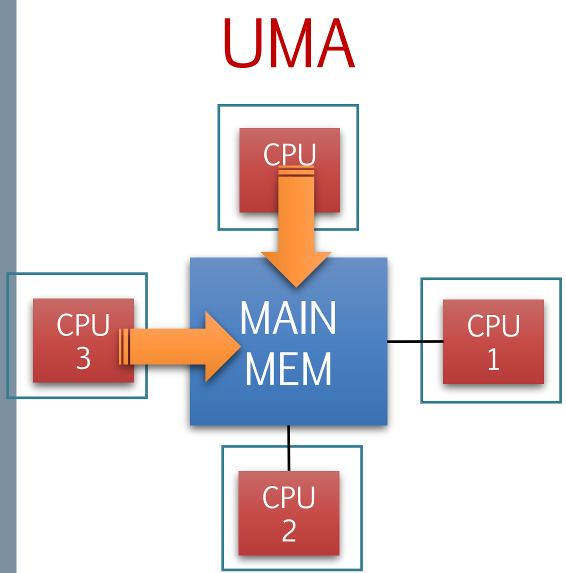
# SMP – distributed shared memory

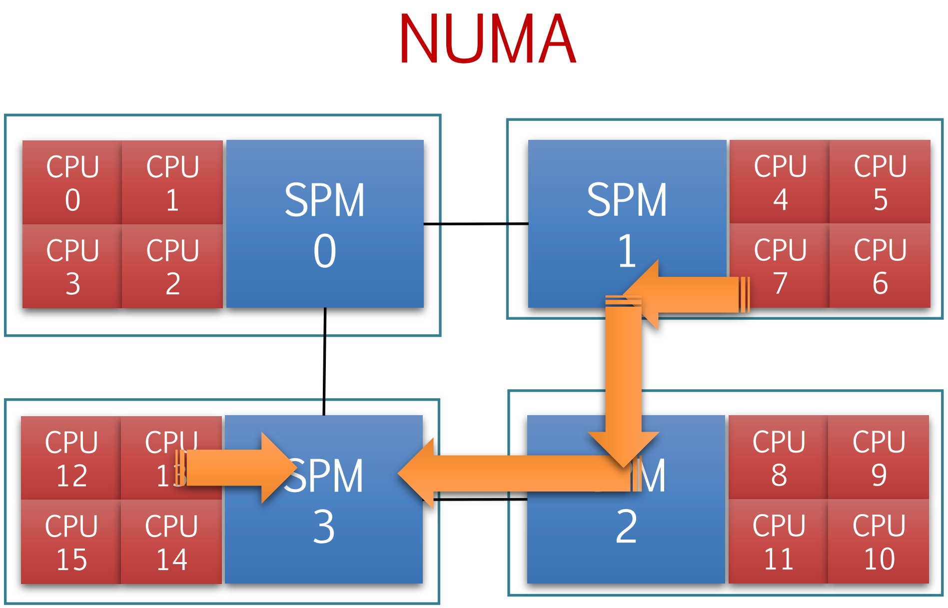
- Non-Uniform Access Time - NUMA
- Scalable interconnect
  - Typically, many cores
  - Examples: embedded accelerators, GPUs



# 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



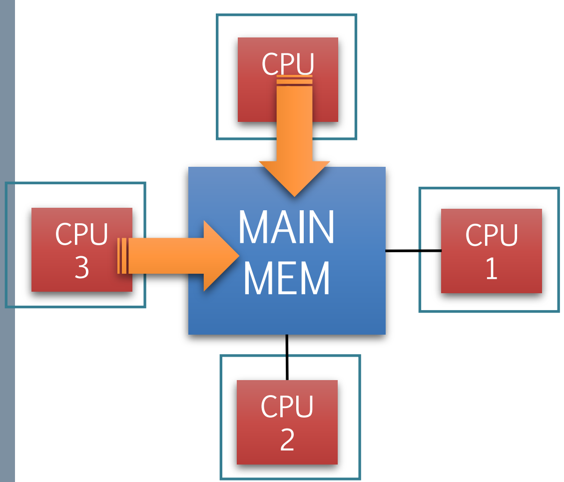



# UMA vs. NUMA

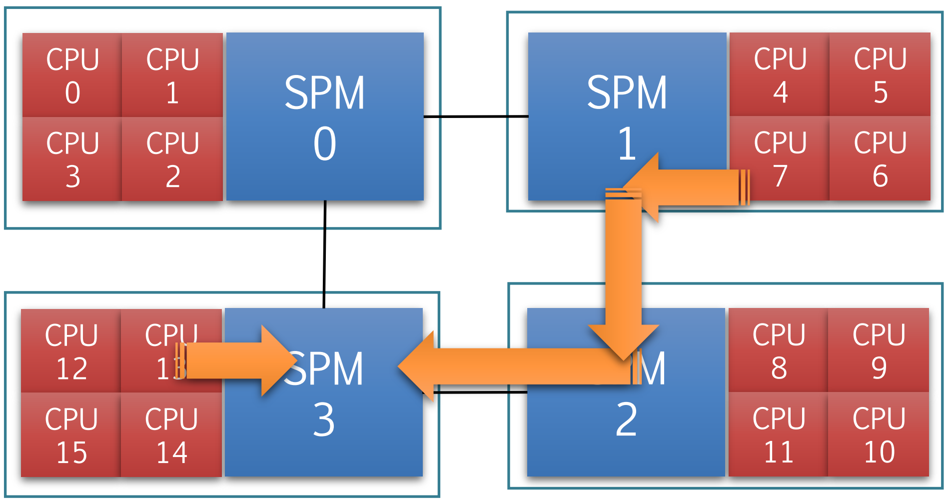
- Shar
  - (
- Unif (NUMA)
  - D

| | 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 | 0 clock |

UMA




NUMA



# Programming abstractions

# (Simplest) threading model

Fork-join execution model

- The main, single thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!


  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```

# (Simplest) threading model

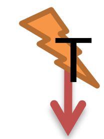
Fork-join execution model

- The main, single thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!


  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```



# (Simplest) threading model

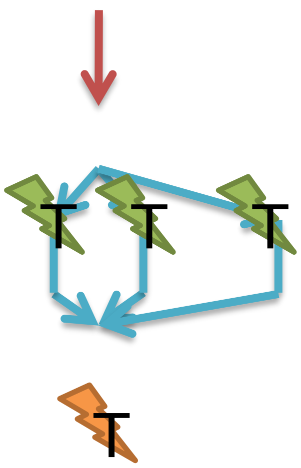
Fork-join execution model

- The main, single thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!


  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```



# (Simplest) threading model

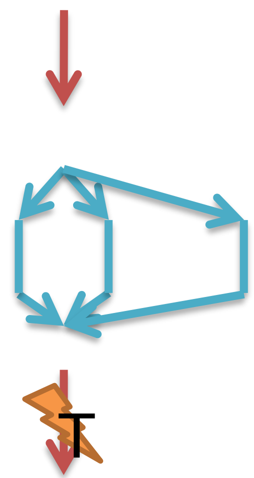
Fork-join execution model

- The main, single thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!


  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```



# (Simplest) threading model

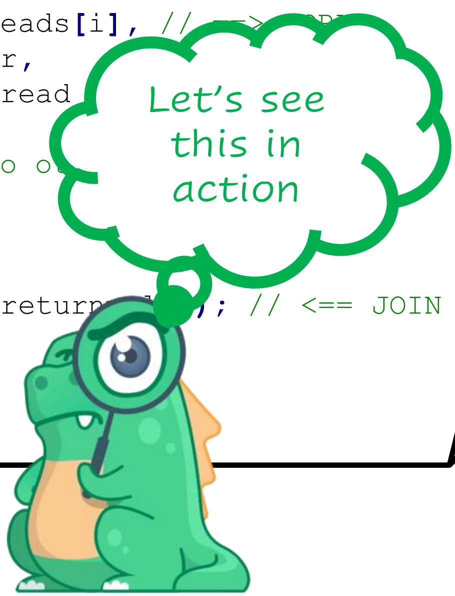
Fork-join execution model

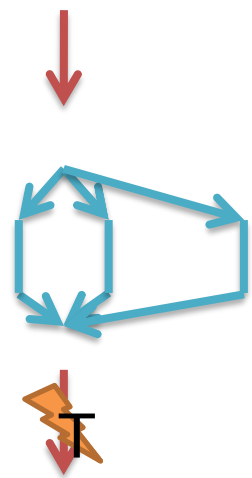
- The main, single thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], //
                          &myattr,
                          my_pthread
                          NULL);
  // Here, the main thread can do o


  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &return ); // <== JOIN

}
```



# Master-slave threading model

Fork-join execution model

- The main, Master thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!
  other_fn();

  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```

# Master-slave threading model

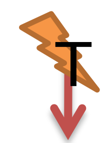
Fork-join execution model

- The main, Master thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!
  other_fn();

  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```



# Master-slave threading model

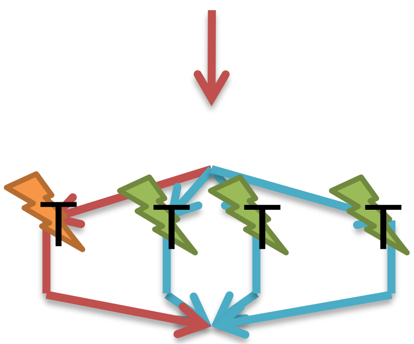
Fork-join execution model

- The main, Master thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!
  other_fn();

  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```



# Master-slave threading model

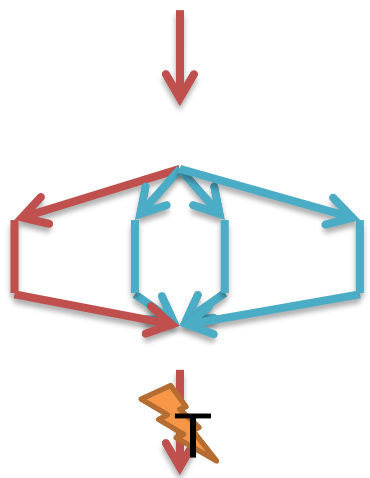
Fork-join execution model

- The main, Master thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel
- At the end, they are joined one by one (aka: barrier)

```
int main()
{
  int err;
  pthread_t mythreads[NTHREADS];
  for (int i=0; i<NTHREADS; i++)
    err = pthread_create (&mythreads[i], // ==> FORK
                          &myattr,
                          my_pthread_fn,
                          NULL);
  // Here, the main thread can do other stuff!
  other_fn();

  for (int i=0; i<NTHREADS; i++)
    pthread_join(mythreads[i], &returnvalue); // <== JOIN

}
```



# Work partitioning

Several models, here to cite a few

- Data parallelism (see also GPGPUS)
  - We're getting there, don't worry…
- Reduction
- Task parallelism (aka: work queue)
- ...

# Data parallelism

(Aka: data decomposition, loop decomposition, SPMD, SIMD*…)

Parallel threads execute the same operation(s) on multiple data

- Data is typically an array, a matrix (image)….
  - Note: you typically map the iteration id of the loop to the data index
- **Partitioning strategy** defines how many iterations (**chunk**) every thread will perform
  - From 1 iteration, to loop size


* Single Program, Multiple Data; Single Instruction, Multiple Data

# Exercise

Let's
code!

Create an array of *N* elements

› Put inside each array element its index, multiplied by '2'

› `arr[0] = 0; arr[1] = 2; arr[2] = 4;` ...and so on..

Now, do it in parallel with a team of *T* PThreads

› Assume *N* is a multiple of *T*

› “Decompose” the `for` construct, so that every thread manages (chunk size is) *N/T* iterations

# Reduction

wikipedia

*The reduction clause can be used to perform some forms of recurrence calculations (involving mathematically associative and commutative operators) in parallel. For parallel [...], a private copy of each list item is created, one for each implicit task, as if the private clause had been used. [...] The private copy is then initialized as specified above. At the end of the region for which the reduction clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified reduction-identifier.*

E.g., average value of a sequence (array/vector)

- Create a thread-local copy of a variable
- Accumulate sums only for the assigned part of the array/vector
- Then, sum the partial sums (and divide by size)

# Exercise

Let's
code!

Create an array of *N* elements

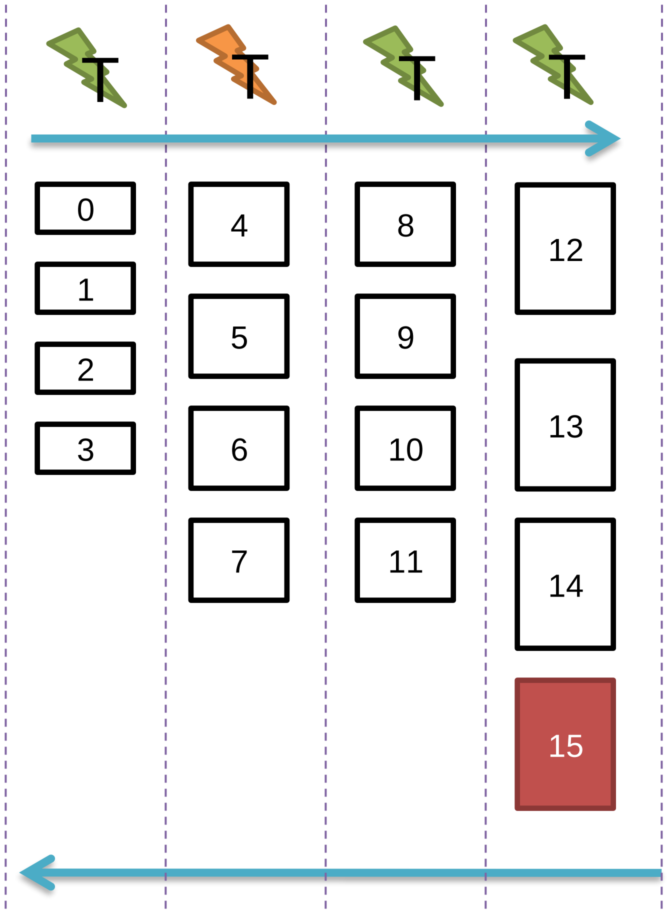
› ..or a vector

› Initiate it randomly

› Now, compute its average value using multi-treading and reduction paradigm

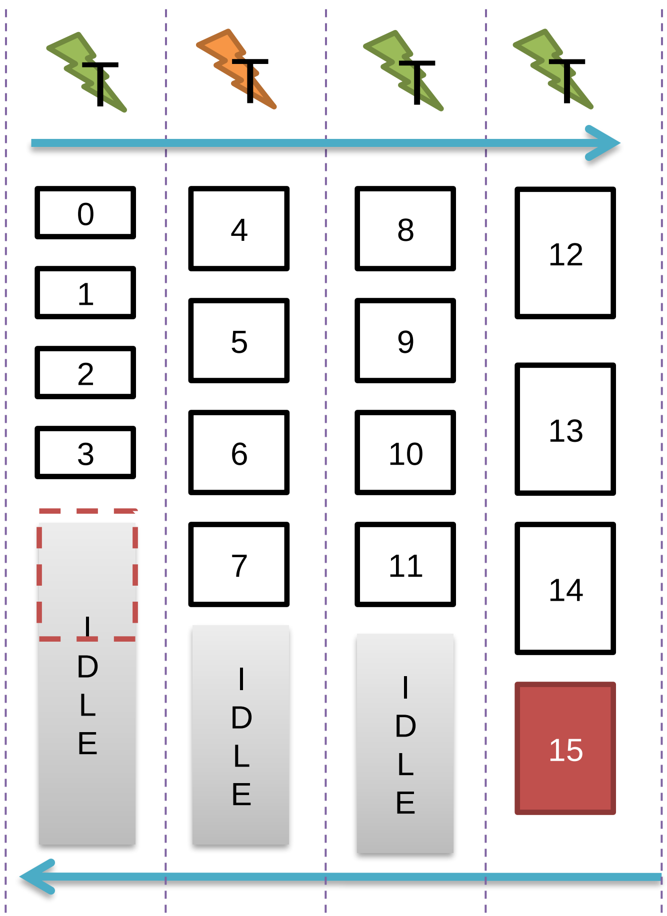
# Unbalanced loop partitioning

- So far, we assigned iterations «statically»
  - Might not be effective nor efficient



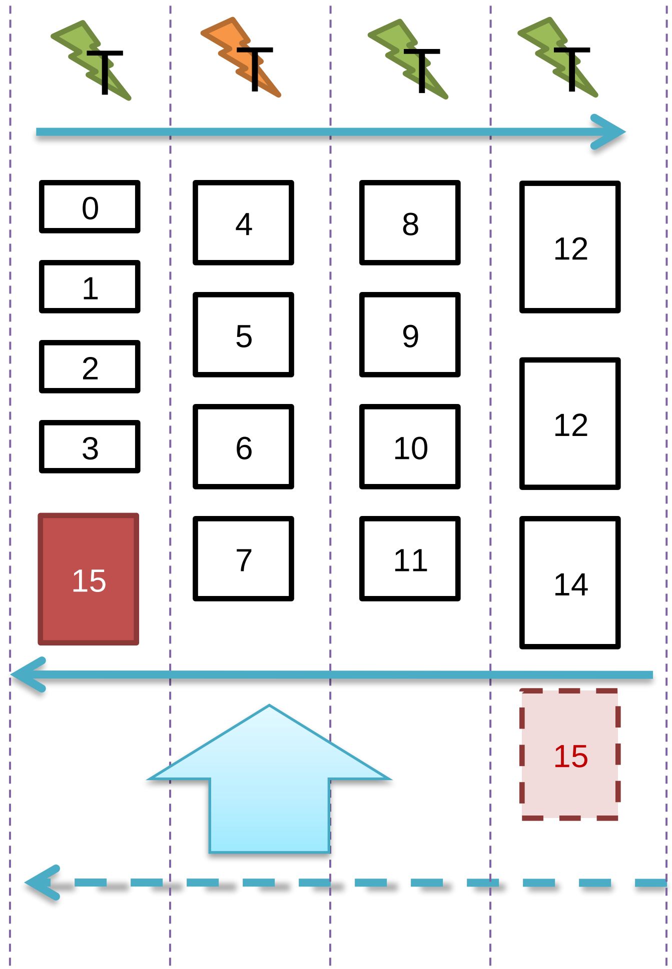
# Unbalanced loop partitioning

- So far, we assigned iterations «statically»
  - Might not be effective nor efficient



# How can we manage dynamics/irregular workloads?

› We would like something like this..

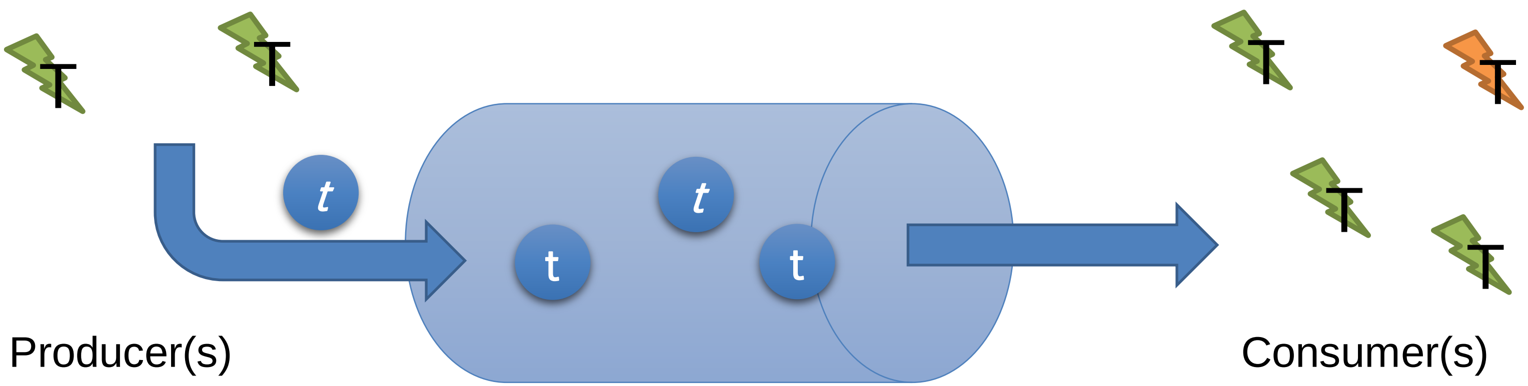


# A different parallel paradigm

Implements a producer-consumer paradigm

Managed by a **task queue**

- Where units of work (**tasks**)
- are pushed by threads (**q_push** primitive)
- and pulled and executed by threads (**q_pop** primitive)



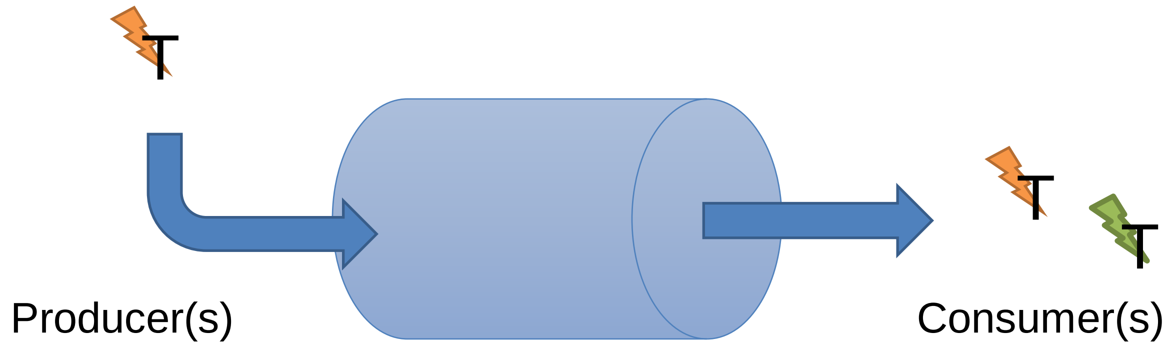
# «What» happens «when»?

```
void t0() {
  // Task 0
}
void t1() {
  // Task 1 pushes t2 in the q
  q_push(t2());
}
void t2() {
  // Task 2
}

void thread_fn() {
  // Push t0 and t1
  q_push(t0());
  q_push(t1());
}
```

```
void other_thread_fn() {
  // Pop a task (which one?)
  t = q_pop();
  // Execute it
  t();
}
```

*pseudo-code



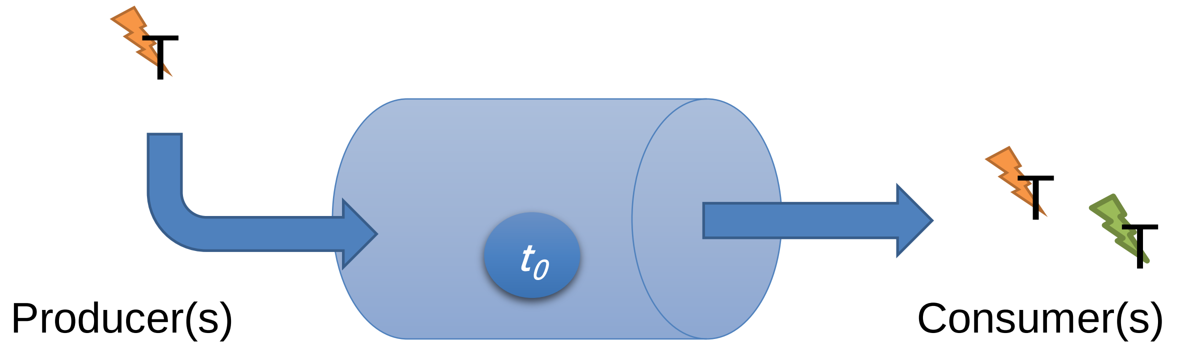
# «What» happens «when»?

```
void t0() {
  // Task 0
}
void t1() {
  // Task 1 pushes t2 in the q
  q_push(t2());
}
void t2() {
  // Task 2
}

void thread_fn() {
  // Push t0 and t1
  q_push(t0());
  q_push(t1());
}
```

```
void other_thread_fn() {
  // Pop a task (which one?)
  t = q_pop();
  // Execute it
  t();
}
```

*pseudo-code



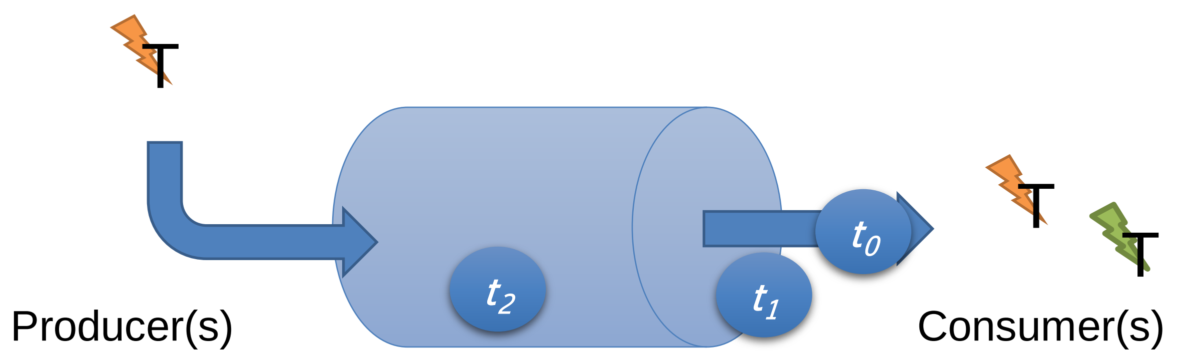
# «What» happens «when»?

```
void t0() {
  // Task 0
}
void t1() {
  // Task 1 pushes t2 in the q
  q_push(t2());
}
void t2() {
  // Task 2
}

void thread_fn() {
  // Push t0 and t1
  q_push(t0());
  q_push(t1());
}
```

```
void other_thread_fn() {
  // Pop a task (which one?)
  t = q_pop();
  // Execute it
  t();
}
```

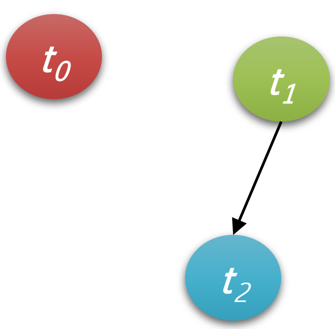
*pseudo-code

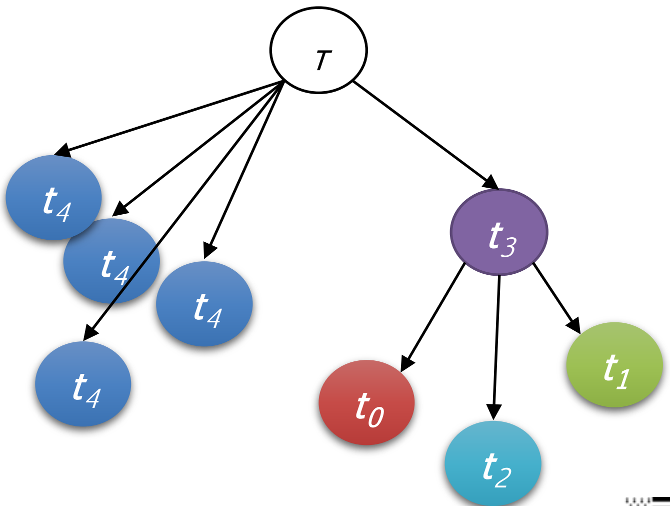


# The queue

It's a shared resource

- Its primitives q_push and q_pop are *thread-safe*, i.e., their concurrent access is protected by semaphores
- Typically implemented as a FIFO queue
- ..but we can also have more complex semantics (e.g., parent-son => DAGs) among tasks



# References

**Course website**

- http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

**My contacts**

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

**Resources**

- “Parallel programming” course by “a guy” @UNIMORE
  - https://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/index.html
  - https://github.com/HiPeRT/cp19/
- A "small blog"
  - http://www.google.com