Parallel programming

Paolo Burgio
paolo.burgio@unimore.it


I still have no idea
what I'm doing

Definitions

Parallel computing

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

Distributed computing

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

Why do we need parallel computing?

Increase performance of our machines

› Scale-up
– Solve a "bigger" problem in the same time
– Metric is **throughput**

› Scale-out (ex: Speedup)
– Solve the same problem «in less time» (or, less power..)
– Metric is **end-to-end latency**

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

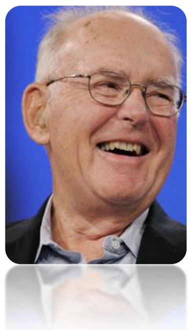
› "The number of transistors that we can pack in a given die area doubles every 18 months"

Dennard's scaling

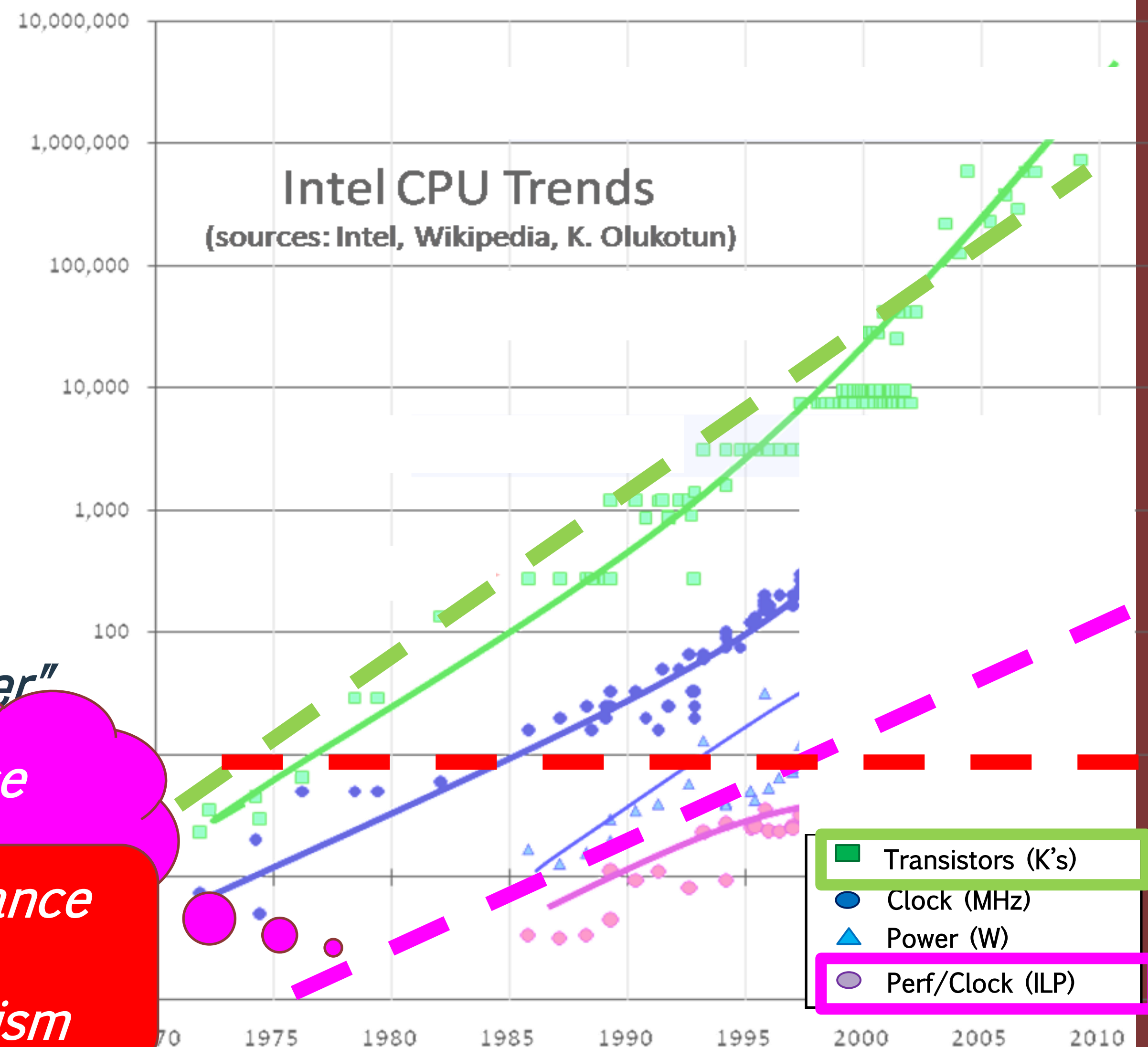
› “Performance per watt of computing is growing exponentially at roughly the same rate"

The answer #2 – the "hot" one

- SoC design paradigm



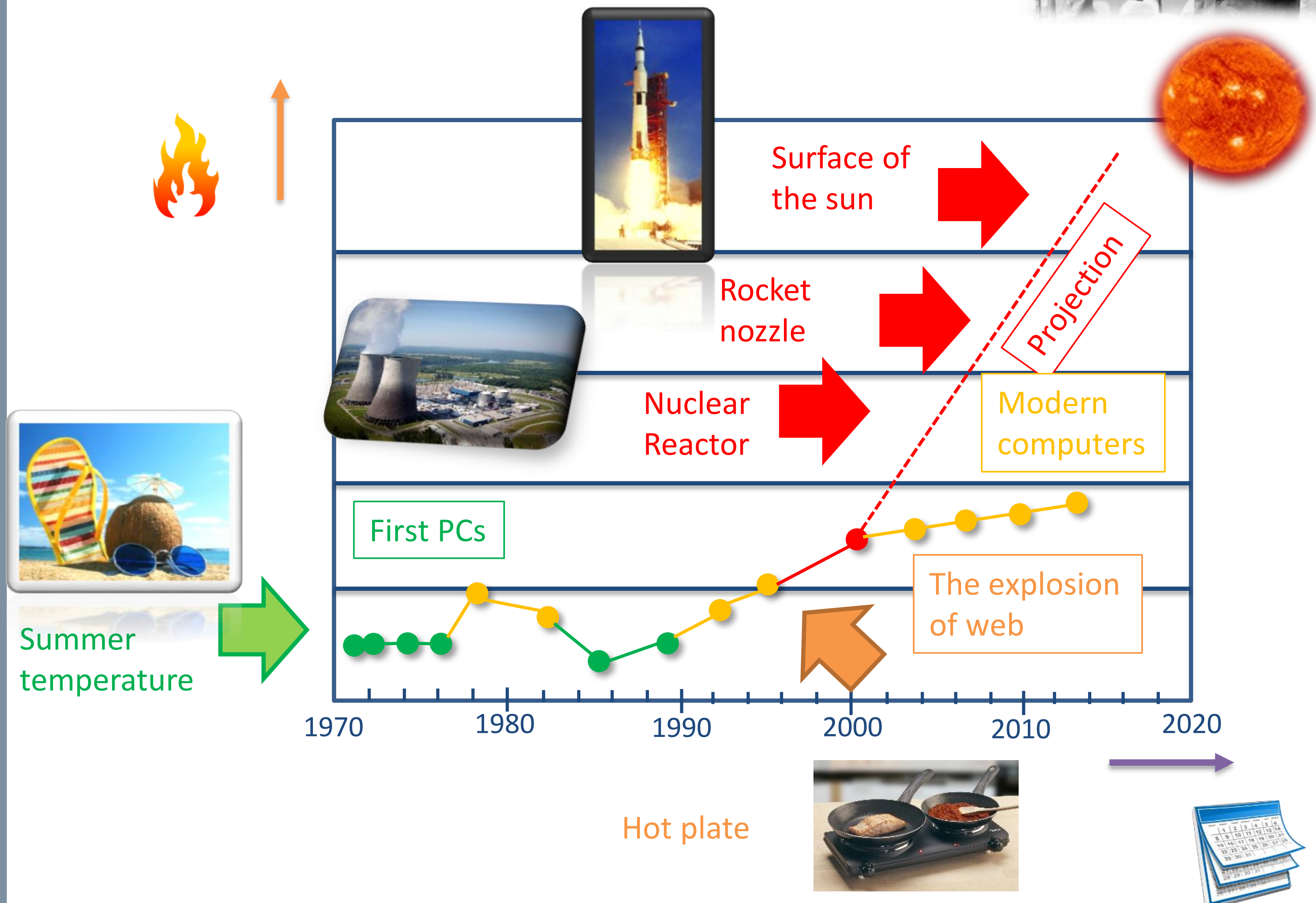
- Gordon Moore
  - His law is still valid, but...
- *“The free lunch is over”*
  - Herb



Performance →

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?"

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,

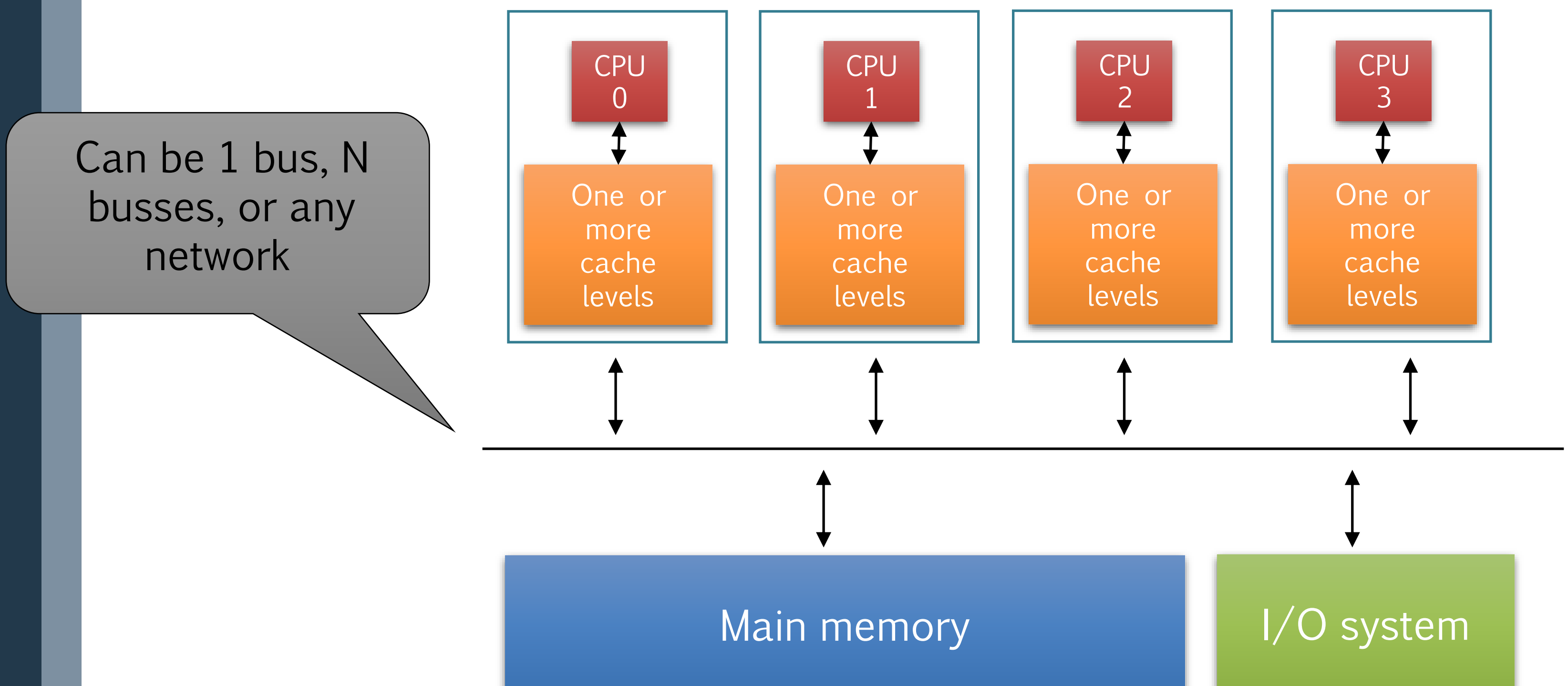
$$T_{par} = 100_{sec} - 95_{sec} = 5_{sec}$$

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

...20x, on one billion cores!!!

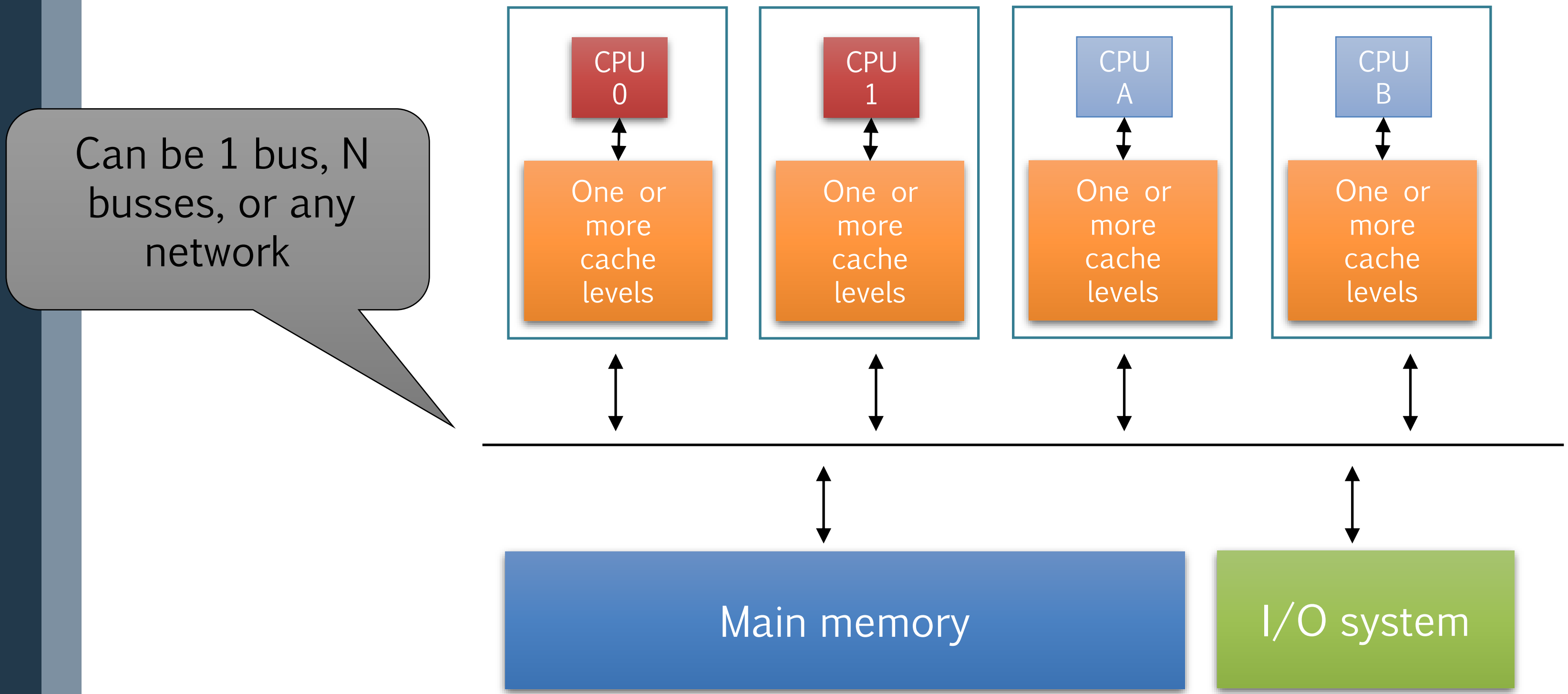
Symmetric multi-processing

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



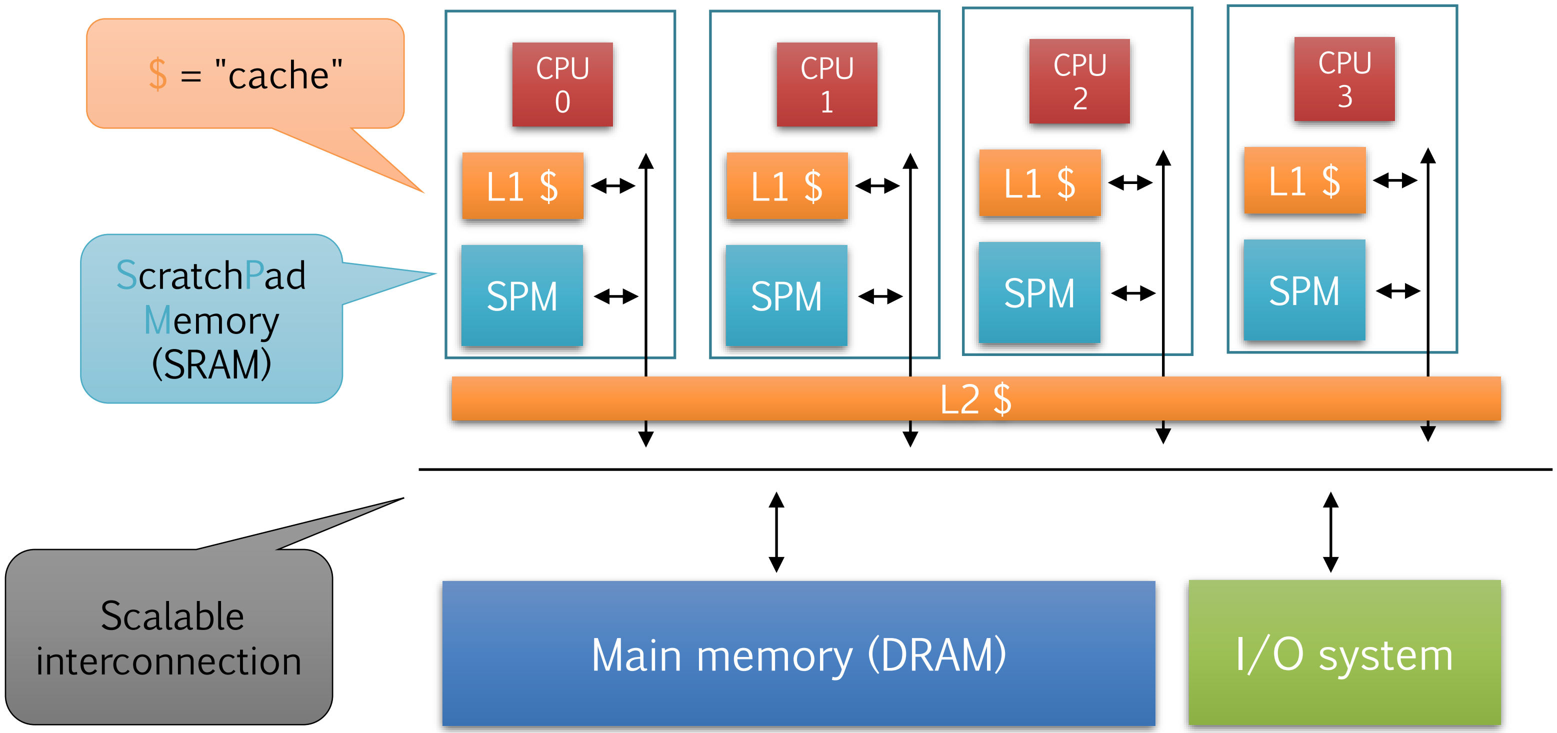
Asymmetric multi-processing

- 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

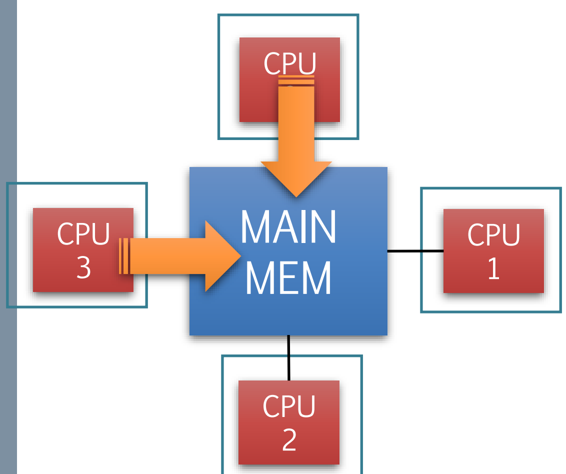
- Non-Uniform Access Time - NUMA
- Scalable interconnect
  - Typically, many cores
  - Examples: embedded accelerators, GPUs



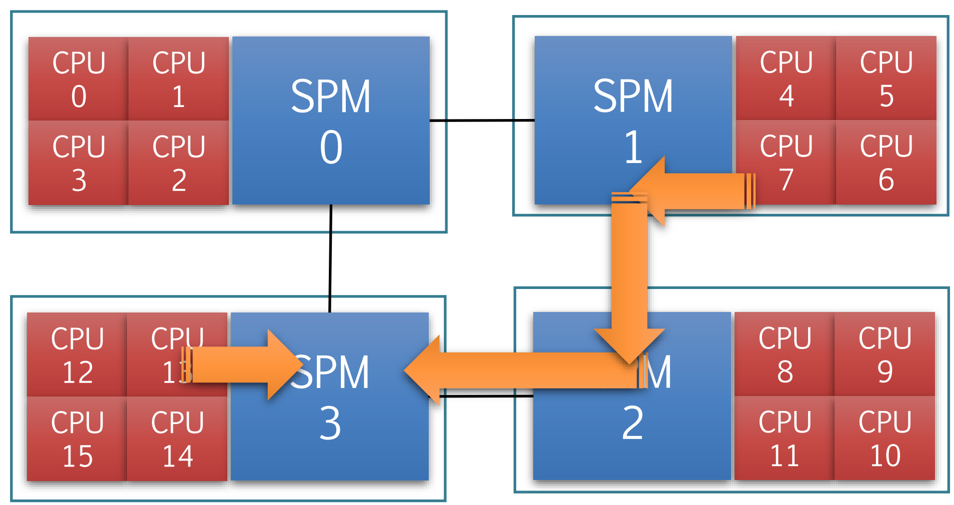
UMA vs. NUMA

	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



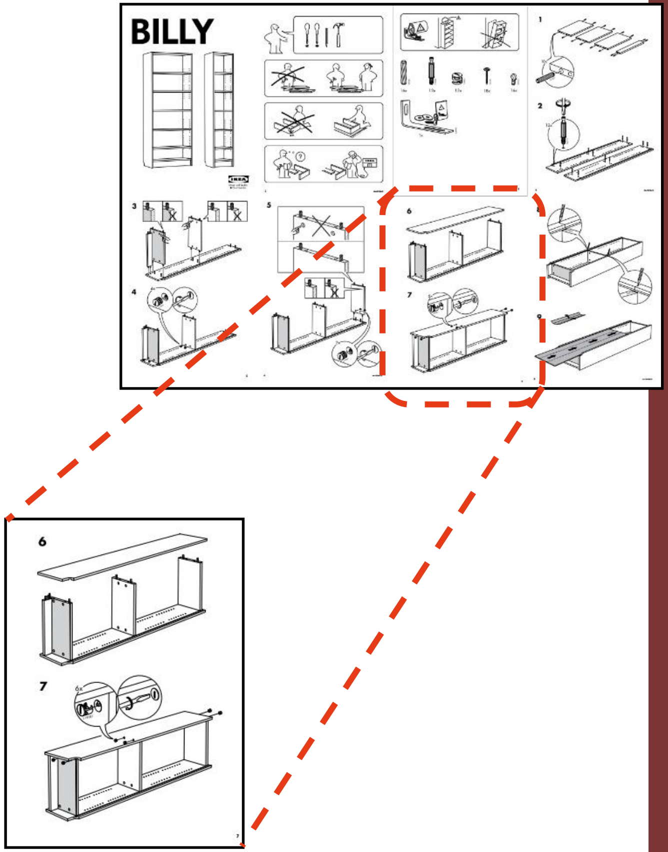
Programming abstractions

What is...

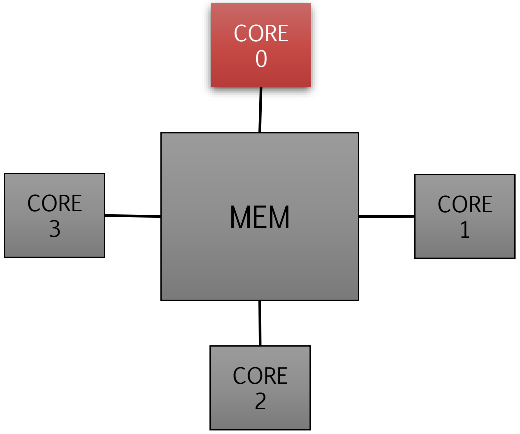
- ..a core?
- ...a program?
- ...a process?
- ...a thread?
- ..a task?

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)

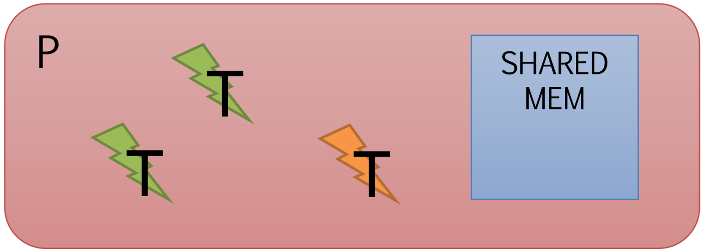


What is...



- ..a core?
  - An electronic circuit to execute instruction (=> programs)

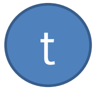
- ...a program?
  - The implementation of an algorithm

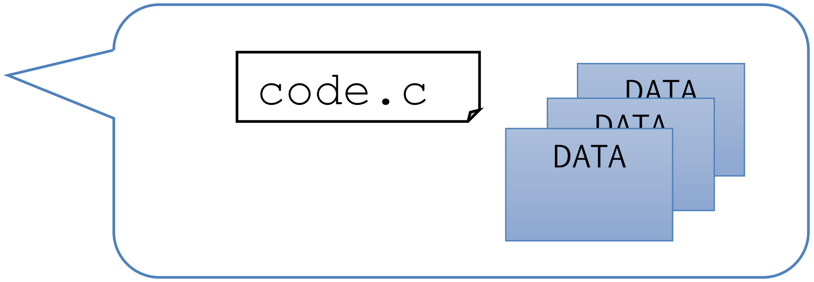


- ...a process?
  - A program that is executing

- ...a thread? 

  - A unit of execution (of a process)

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



(Simplest) threading model

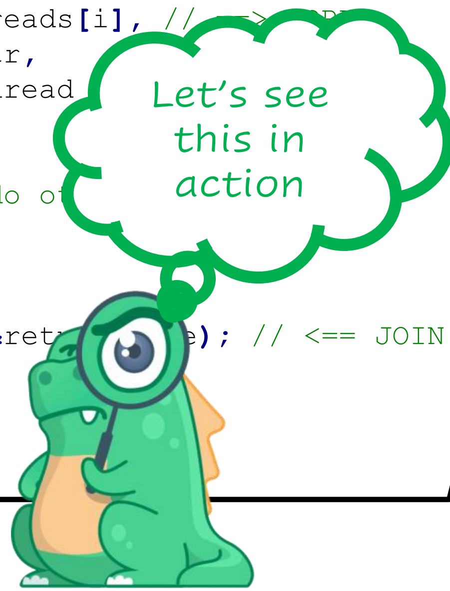
Fork-join execution model

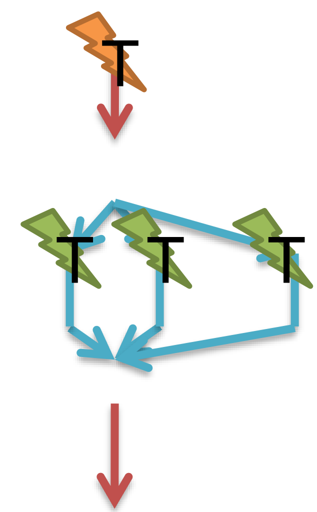
- The main, single thread thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel (as a team of theads)
- At the end, they are joined one by one (aka: barrier)

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


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

}
```





Master-slave threading model

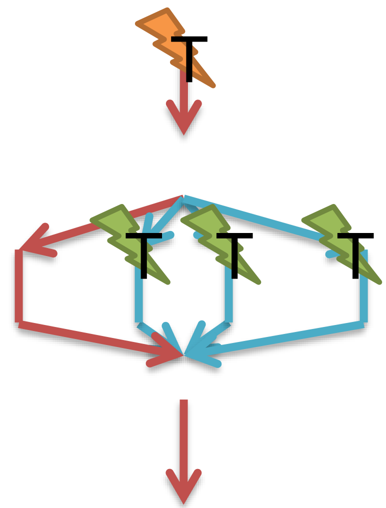
Fork-join execution model

- The main, Master thread spawns a team of Slave threads (here, NTHREADS = 3)
- They all perform computation in parallel (as a team of theads)
- 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);
  // PARALLEL REGION.
  // Here, the main thread can do other stuff!
  other_fn(); // or can run my_pthread_fn

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

}
```

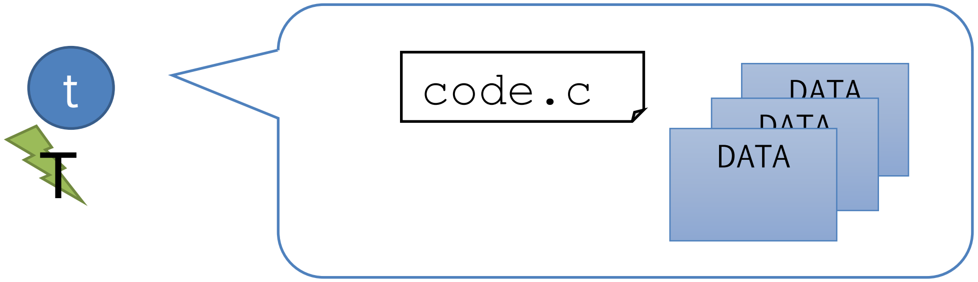


Work partitioning

Several models (aka: patterns), here to cite a few

- Data parallelism (see also GPGPUS)
  - We're getting there, don't worry...
  - Reduction
- Task parallelism
  - Work queue
  - Pipelining
- Offloading

...and a mix of them...



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..

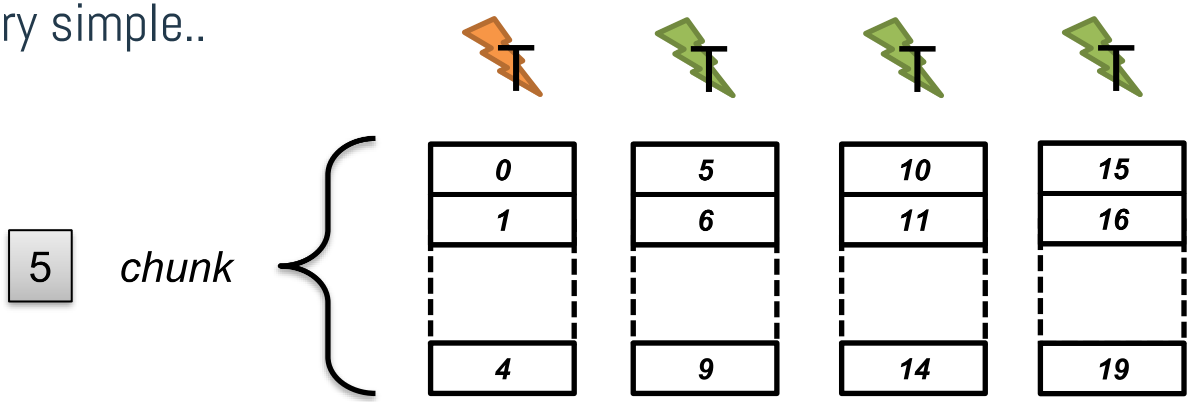
Now, do it in parallel with a team of *T* PThreads

- Assume *N* is always bigger than *T*
- "Decompose" the `for` construct, so that every thread manages (chunk size is) *N/T* iterations

Loop partitioning among threads

Let's
code!

- Case #1: N multiple of T
  - Say, N = 20, T = 4
- chunk = #iterations for each thread
- Very simple..



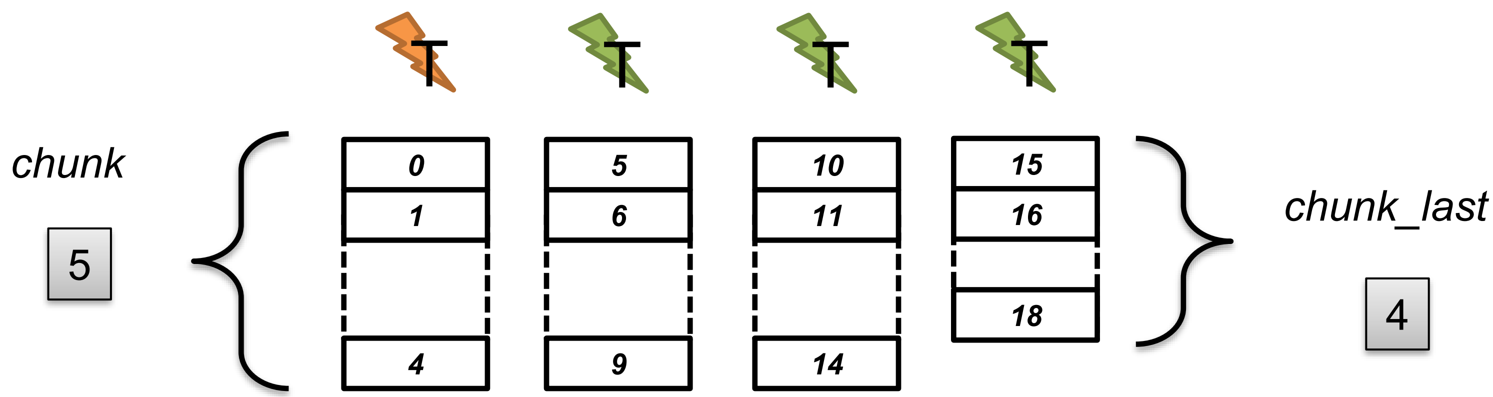
$$chunk = \frac{N}{T};$$

$$i_{start} = thread_{ID} * chunk; \qquad i_{end} = i_{start} + chunk$$

Loop partitioning among threads

Let's code!

- Case #2: N not multiple of T
  - Say, N = 19, T = 4
- chunk = #iterations for each thread (but last)
  - Last thread has less! ($chunk_{last}$)



$$chunk = \frac{N}{T} + 1\,; \qquad chunk_{last} = N\ \%\ chunk$$

$$i_{start} = thread_{ID} * chunk; \qquad i_{end} = \begin{cases} i_{start} + chunk & if\ not\ last\ thread \\ i_{start} + chunk_{last} & if\ last\ thread \end{cases}$$

"Last thread"

Let's code!

- Unfortunately, we don't know which thread will be "last" in time
- But...we don't actually care the order in which iterations are executed
  - If there are not depenencies..
  - And..we do know that

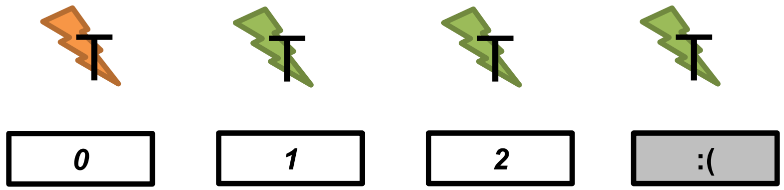
0 <= `myid` < `NUM_THREADS`

- We choose that last thread as highest number

Loop partitioning among threads

Let's code!

- Case #3: N smaller than T
  - Say, N = 3, T = 4
- Similar to Case #1
  - With some adjustments...



Let's put them together!

Let's
code!

- Case #1 (*N* multiple of *T*)

$$chunk = \frac{N}{T} \qquad i_{start} = thread_{ID} * \ chunk; \qquad i_{end} = i_{start} + chunk$$

- Case #2 (*N* not multiple of *T*)

$$chunk = \frac{N}{T} + 1\,; \qquad chunk_{last} = N\ \%\ chunk$$

$$i_{start} = thread_{ID} * \ chunk; \qquad i_{end} = \begin{cases} i_{start} + chunk & if\ not\ last\ thread \\ i_{start} + chunk_{last} & if\ last\ thread \end{cases}$$

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

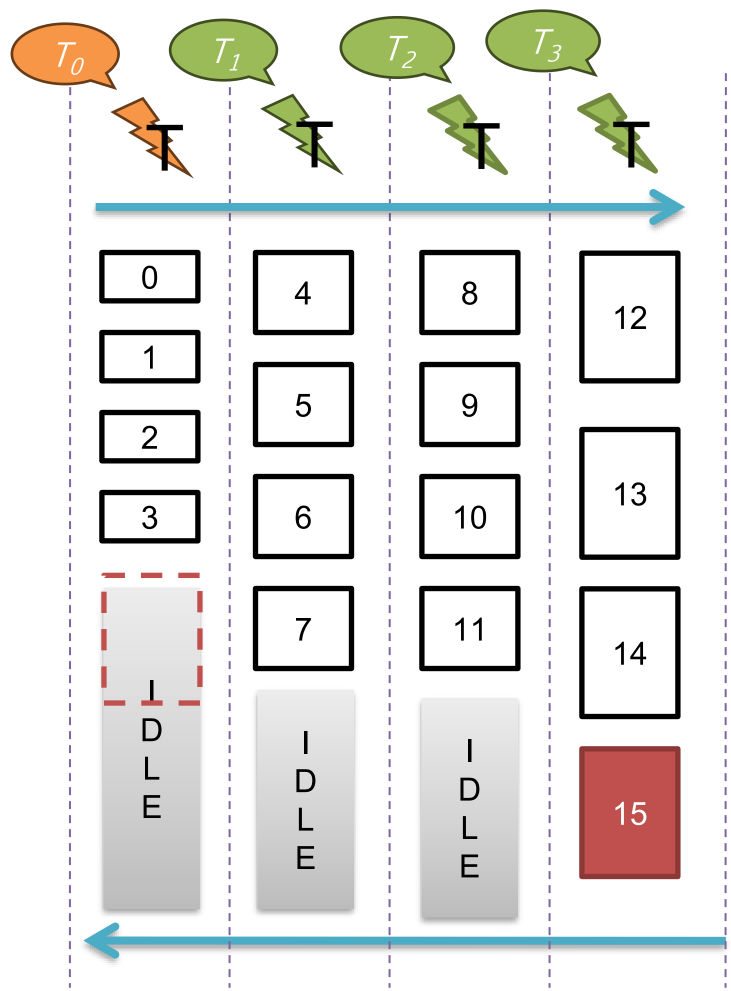
Let's code!

Create an array of *N* elements

- ..or a vector
- Initiate it randomly
- Now, compute its average value using multi-treading and reduction paradigm

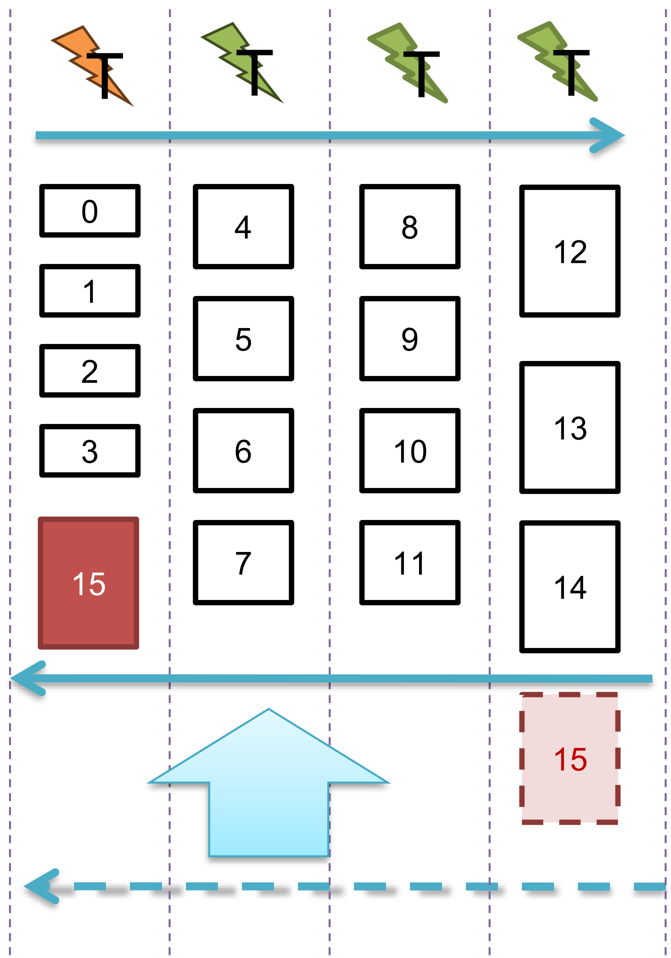
Unbalanced loop partitioning

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



How can we manage dynamics/irregular workloads?

› We would like something like this..

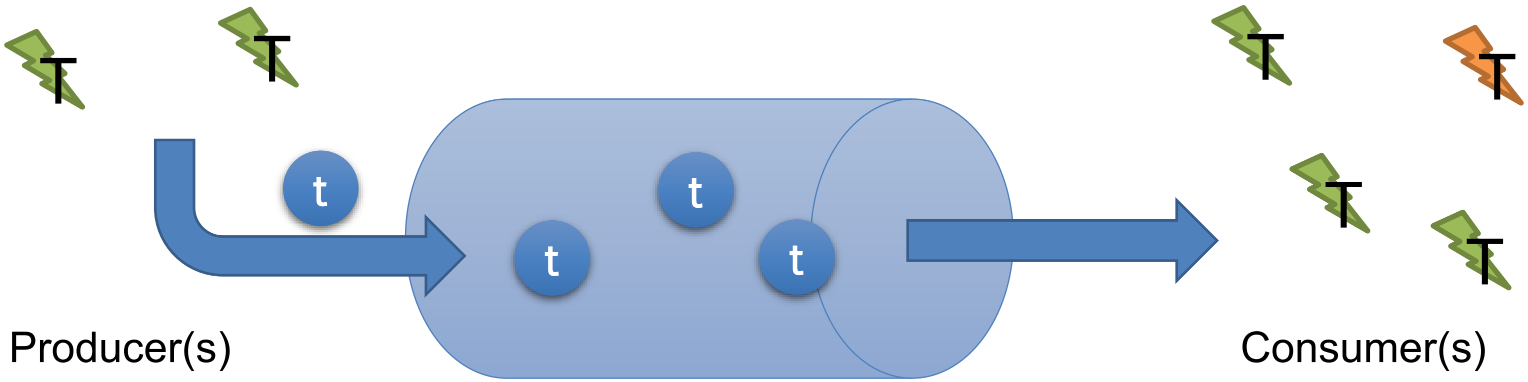


A different parallel paradigm: Tasking

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)



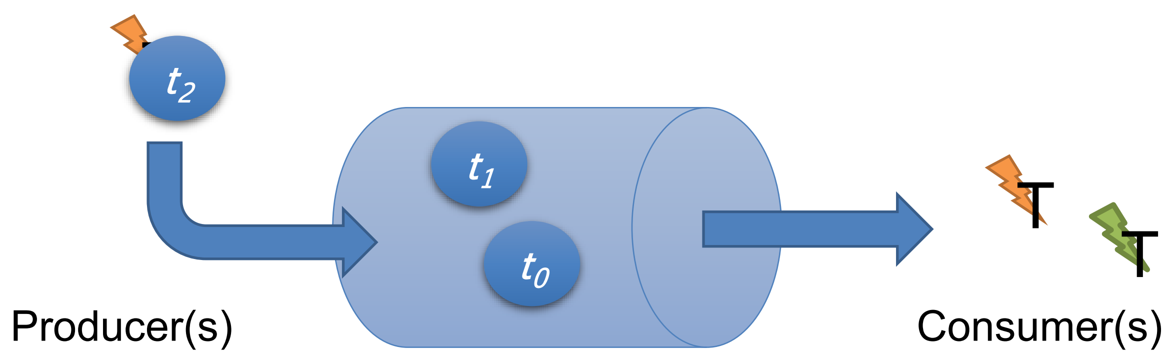
«What» happens «when»?

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

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

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

*pseudo-code



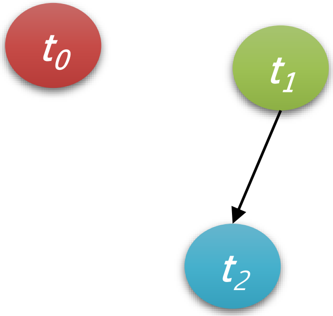
The queue

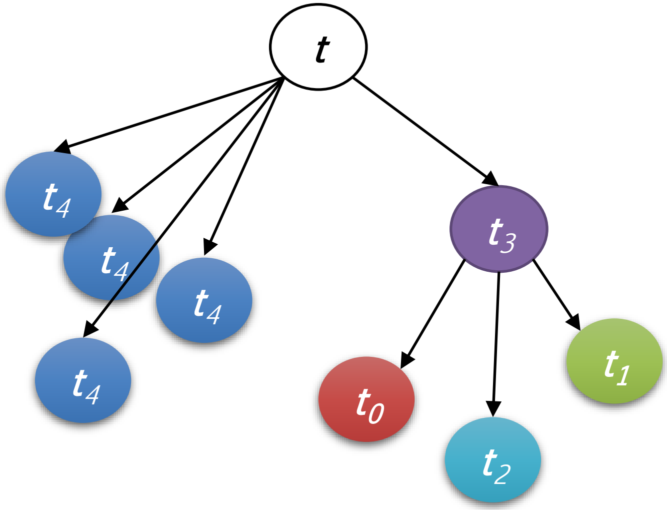
It's a shared resource

- Its primitives **q_push** and **q_pop** are *thread-safe*, i.e., internally, the concurrent access to shared data structure is protected by semaphores/mutexes

Typically implemented as a FIFO queue

- ..but we can also have more complex semantics (e.g., parent-son => DAGs) among tasks
- We can have queues that are tailored/more efficient for a specific problem/domain/algorithm!





Parallel programming patterns

Creational/threading
- *Fork-join*
- *Master-slave*
- *Team of threads*

Work partitioning
- *Data-parallelism*
- *Reduction*
- *Tasking/queue*
- Offloading

Synchronization
- Barrier
- Critical section/Atomic
- Locks, semaphores, mutexes...

Behavioral
- Producer-consumer
- Collaborative (multi-thread)
- Competitive (multi-process)

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/

› Gamma, et.al «Design Patterns – Elements of reusable Object Oriented Software», Addison Wesley

› Douglass – «Design Patterns for Embedded Systems in C», Newnes

› A "small blog"
  – http://www.google.com