Writing Parallel Software

Danilo Piparo – CERN, EP-SFT and CMS

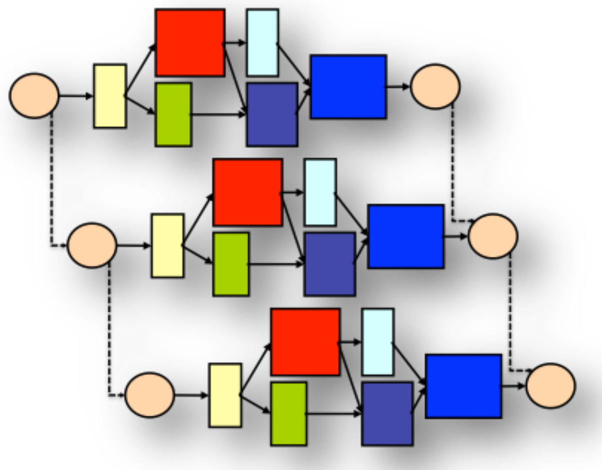
This Lecture

The outline:

- Parallel software design: an introduction
- Threads and parallelism in C++
- Threads and data races: synchronisation issues
- Useful design principles
- Replication, atomics, transactions and locks

CERN School of Computing

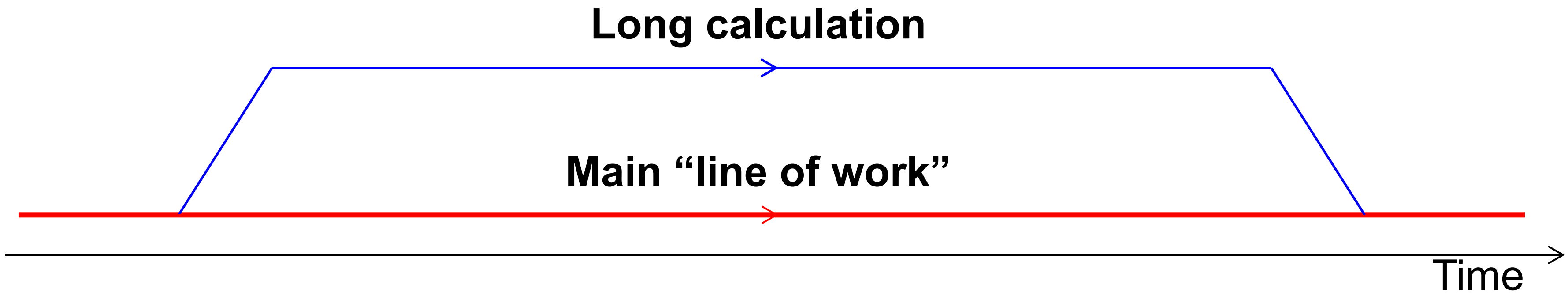
Asynchronous Execution



CERN School of Computing

Asynchronous Task Execution

- Problem: **a long calculation**, the result of which **is not immediately needed**
- Possible solution: **asynchronous execution** of the calculation, retrieval of the result at a later stage
- Nuances: result may or may not be **needed later** depending on the control flow steering the application
  - **Lazy evaluation?**



CERN School of Computing

std::async

- A solution is provided by the standard library natively: `std::async`
  - `#include <future>`
- **Execute a function concurrently in a separate thread** or on demand when the result is needed (lazily)
- **Result is a `std::future`: a "bridge"** between the two locations:
  - **`std::future` "Transports" results and exceptions from thread to thread**
- In orther words, code to be executed is passed around

CERN School of Computing

std::async in Action

Header for async and future

"Launch" the calculation

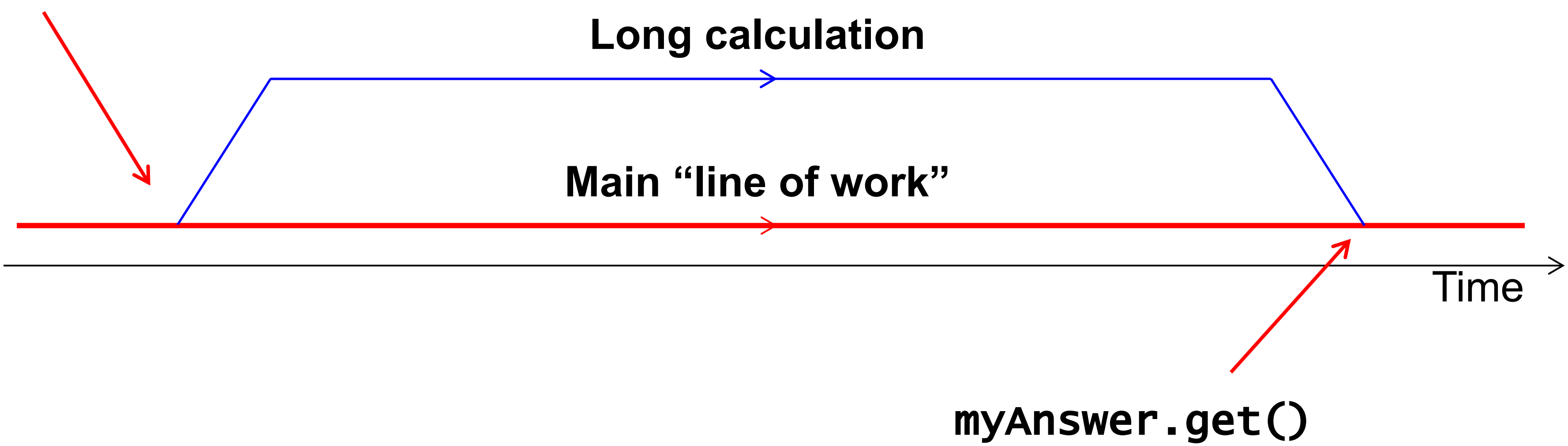
Retreive result

```
#include <future>
#include <iostream>

int lenghtyCalculation(){ […] };
void doOtherStuff(){ […] };

int main(){
  std::future<int> myAnswer = std::async(lenghtyCalculation);
  doOtherStuff();
  std::cout << "The result is: " << myAnswer.get() << std::endl;
  }
```

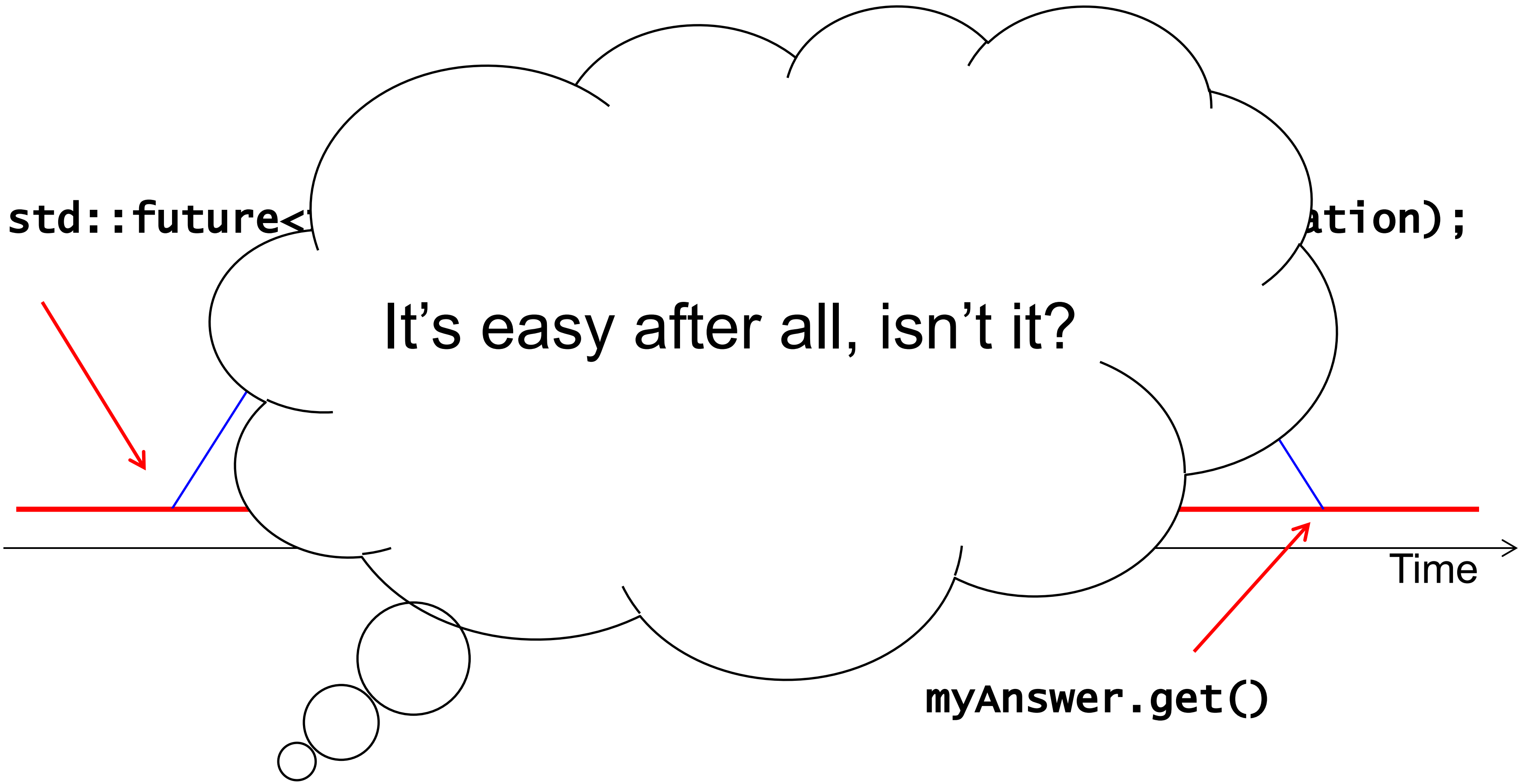
CERN School of Computing

std::async in Action

```
std::future<int> myAnswer = std::async(lenghtyCalculation);
```



std::async in Action



CERN School of Computing

Well, to be Honest… No.

- Unfortunately scientifically relevant / potentially lucrative real life use cases are complex
  - Cannot be solved simply throwing threads at them
- In addition, many existing high-quality sequential large software packages are in production
  - Starting fresh may not be always possible
- Example: software stack of an LHC experiment
  - Tens of (large) packages integrated
  - $O(10^2)$ shared libraries
  - Experiment specific code
  - → Millions of nicely working lines of code

Need to think parallel
- Evolve the existing systems
- Be disruptive and think to the future

Unity of opposites ☺

Parallel Software Design: an Introduction

First Step: Finding Concurrency

What can be executed concurrently?

Two techniques to figure this out:

- ***Data decomposition***
  - The partition of the data domain
  - Achieve data parallelism
- ***Task decomposition***
  - Split according to logical tasks
  - Achieve task parallelism

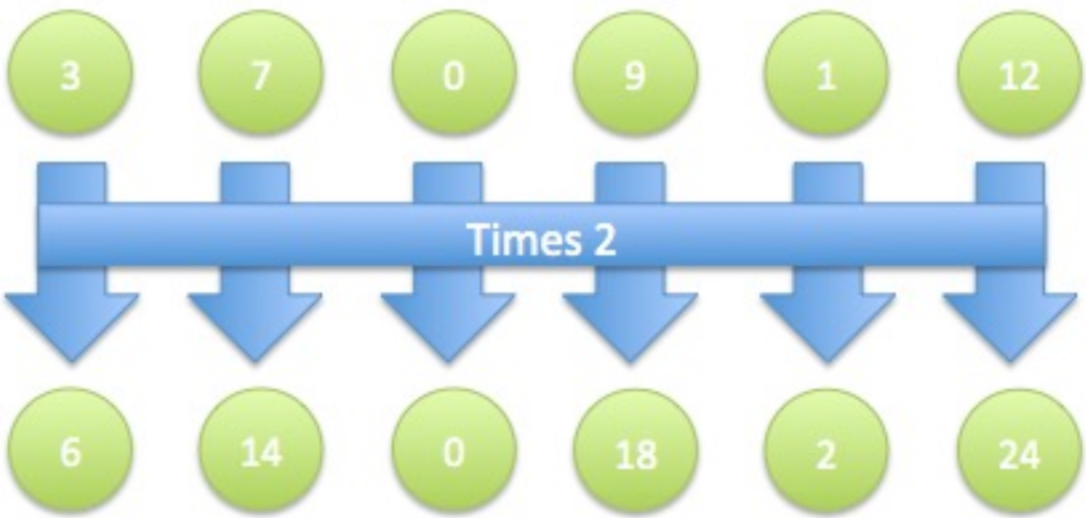
DIVIDE ET IMPERA

This step takes place in front of a whiteboard

Data Parallelism

Definition: parallelism achieved through the application of the same transformation to multiple pieces of data

An illustration: multiplication of an array of values



Data parallelism implies wise design of the data structures to be used!

CERN School of Computing

Data Parallelism: Examples

Increase floating point throughput acting on mathematical functions:

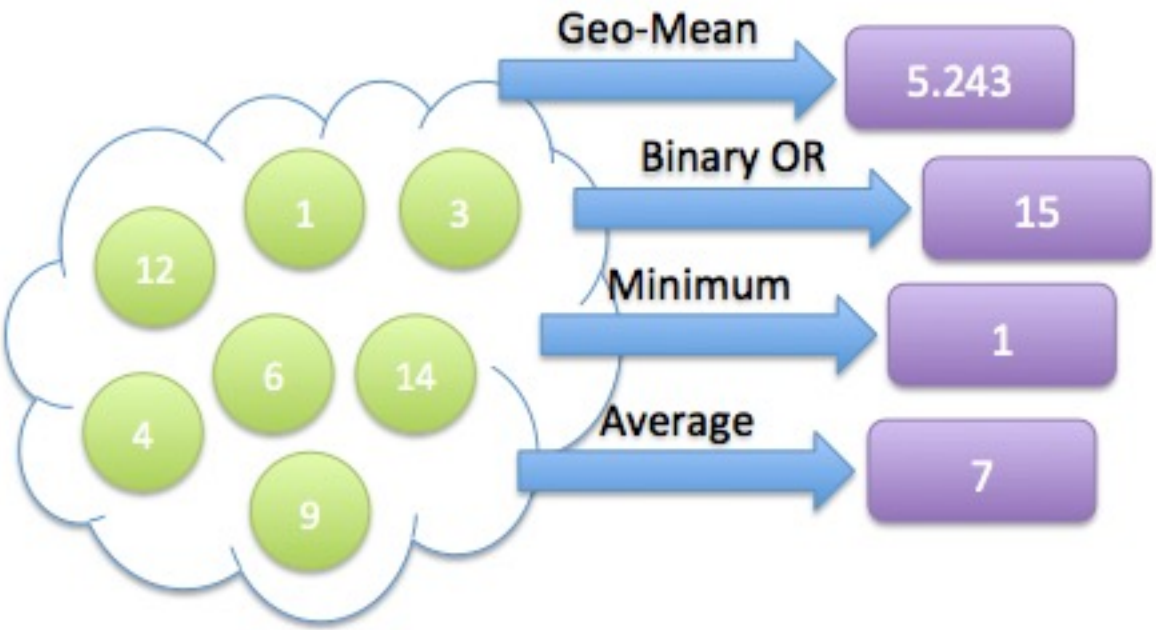
- Math functions account for a significant portion of many scientific applications
- Decompose the functions in simple vectorisable FP operations, at the heart of which there can be some sort of polynomial evaluation
- Calculate math functions on independent inputs in parallel
    - For example using vectorisation techniques
- “Seen in real life”: Intel **MKL**, **VDT,** libraries.

CERN School of Computing

Task Parallelism

Definition: parallelism achieved through the partition of load into “baskets of work” consumed by a pool of resources.

An illustration: calculate mean, binary OR, minimum and average of a set of numbers



A bit too simple: no dependency between tasks!

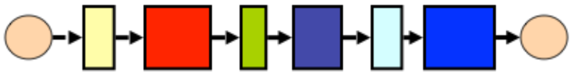
CERN School of Computing

Task Parallelism: An example

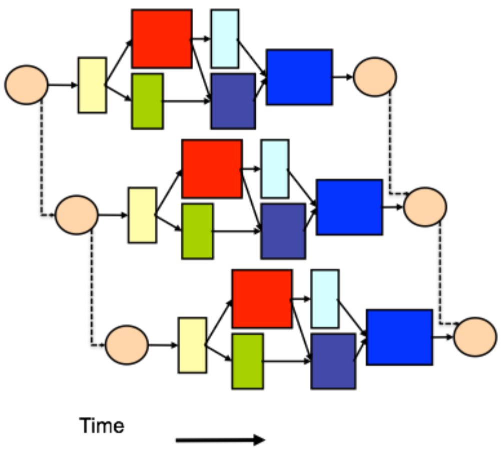
HEP data processing frameworks

- Run in a certain order **algorithms on collision events**
  - In a nutshell: transform data from detector readout electronics into particle kinematics in steps
- For decades, one algorithm executed at the time, one event processed at the time
- Evolving to accommodate parallelism, also outside the single algorithms
- One of the key ideas: **schedule algorithms in parallel according to their data dependencies**, also keeping N events in memory

Sequential Execution



A possible parallel execution graph



Pure Task/Data parallelism

- We do not need to “choose” to approach a problem with a task or data parallelism based solution
- Actually, pure task/data parallelism is rare!
- Combining the two is the key

CERN School of Computing

Is Parallelisation Worth It?

- Whenever thinking about parallelisation, one should spend some thoughts on whether the effort is worth it
  - The total cost of ownership of one additional box might be smaller than the design-implementation-maintenance costs
- **What is the performance gain we can expect?**

CERN School of Computing

Need for Speed(up)

- We parallelise because we want to run our application faster
- **Speedup**: how much faster does my code run after parallelising it?
  - Indicator of **scalability**

$$Speedup = \frac{Time_{serial}}{Time_{parallel}}$$

From E. Tejedor

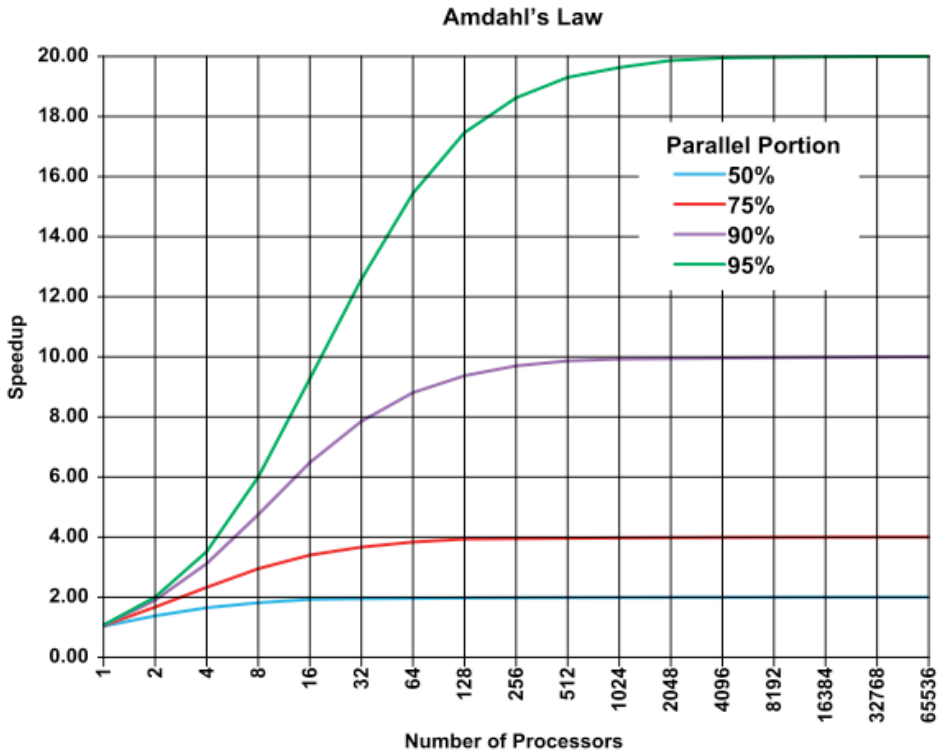
CERN School of Computing

Amdahl's Law

- It predicts the maximum speedup achievable given a problem of **fixed size**

$$Speedup = \frac{1}{(1-p)+\frac{p}{n}}$$

n: number of cores
p: parallel portion



"… the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." - 1967

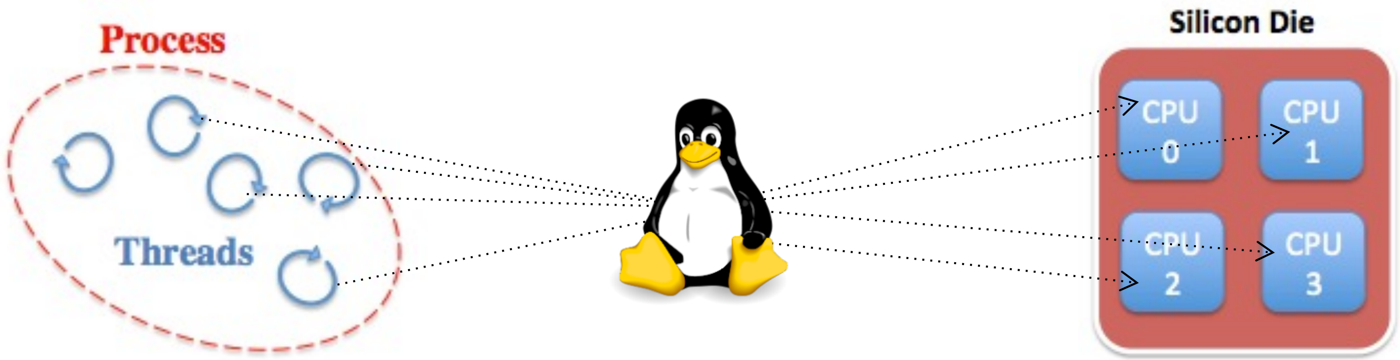
From E. Tejedor

Threads and C++

CERN School of Computing

Let’s change gears: Threads

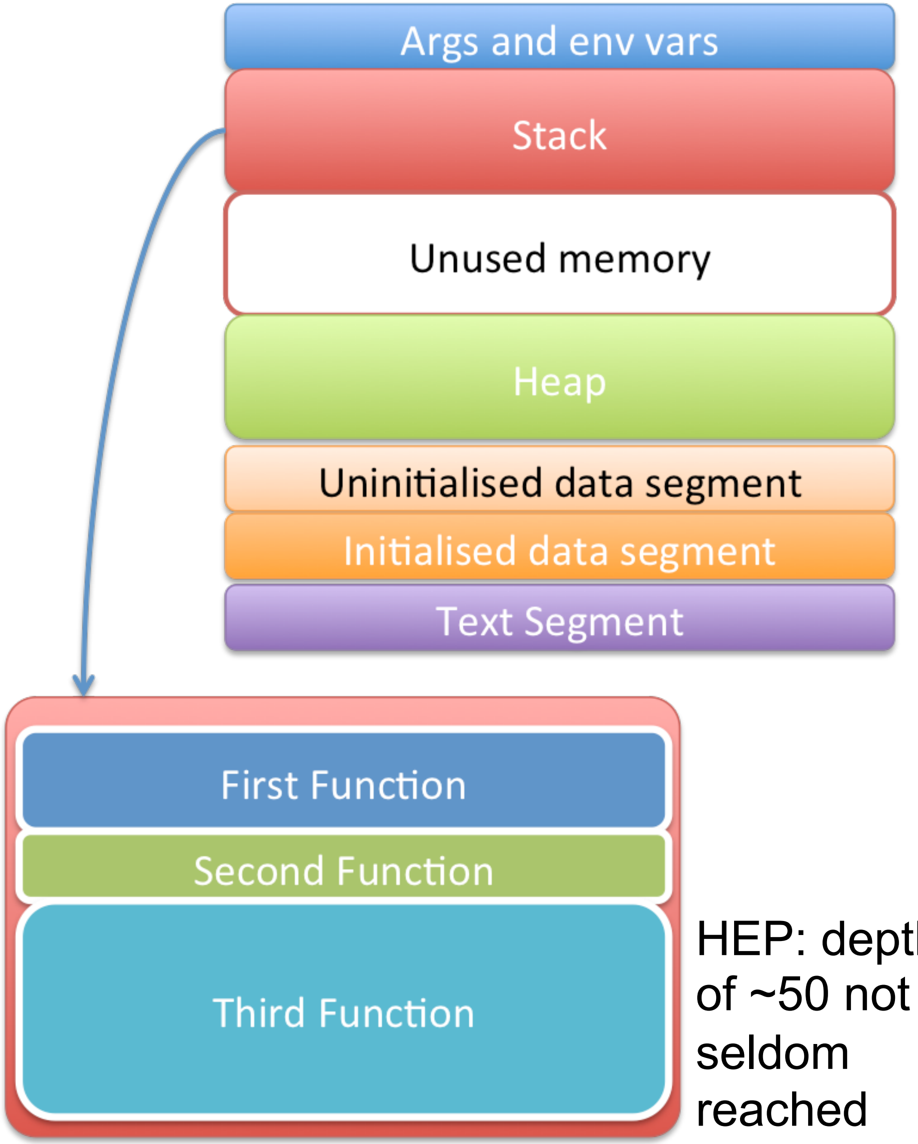
- From the operating system point of view:
  - **Process**: isolated instance of a program, with its own space in (virtual) memory, can have multiple threads
  - **Thread**: light-weight process within process, **sharing the memory** with the other threads living in the same process
- The kernel manages the existing threads, scheduling them to the available resources (CPUs)*
  - There can be more threads in a single process than cores in the machine!



* Actually mapping user threads to kernel threads, but this simplification ok in first order!

Interlude: A Program in Memory

- **Text Segment**: code to be executed.
- **Initialized Data Segment**: global variables initialized by the programmer.
- **Uninitialized Data Segment**: This segment contains uninitialized global variables.
- **The stack**: The stack is a collection of stack frames. It grows whenever a new function is called. "Thread private".
- **The heap**: Dynamic memory (e.g. requested with "`new`").



HEP: depth of ~50 not seldom reached

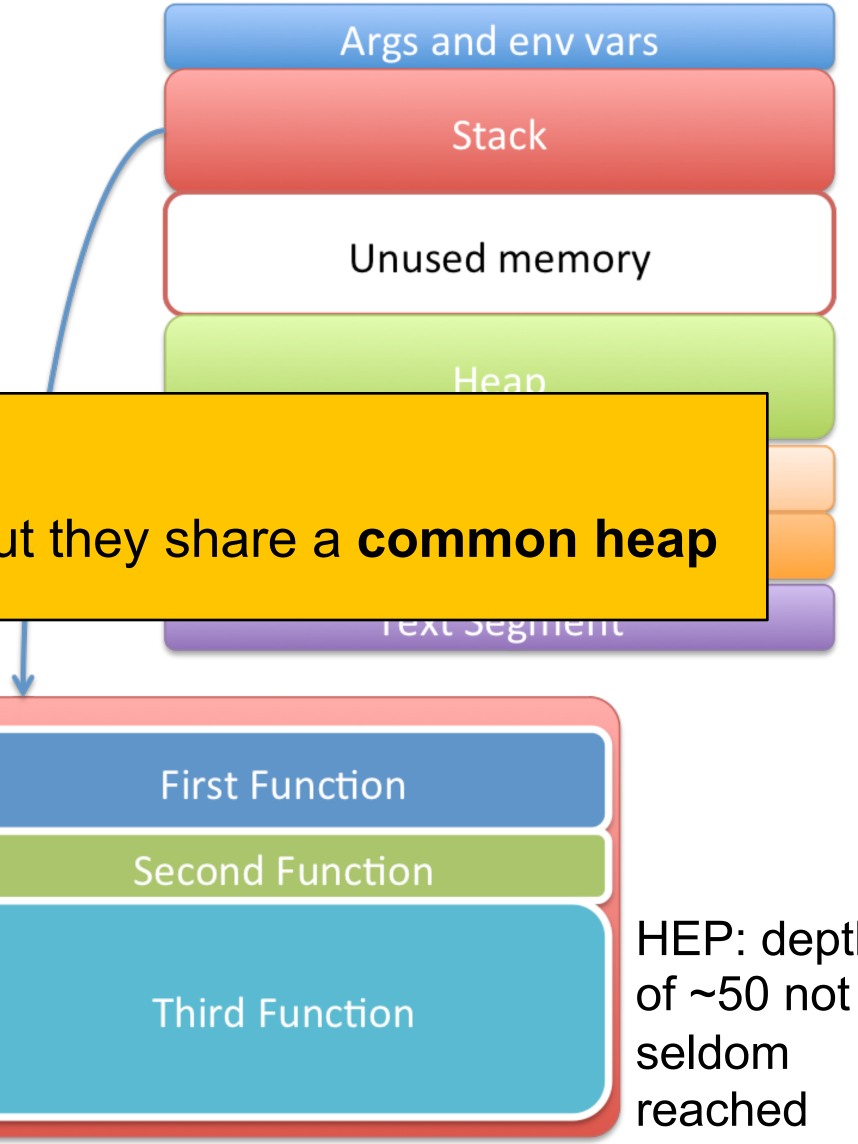
CERN School of Computing

Interlude: A Program in Memory

- **Text Segment**: code to be executed.
- **Initialized Data Segment**: global variables initialized by the programmer.
- **Un** Thi uni

Terminology:
Threads have their **own stack**, but they share a **common heap**

- **The stack**: The stack is a collection of stack frames. It grows whenever a new function is called. “Thread private”.
- **The heap**: Dynamic memory (e.g. requested with “`new`”).



HEP: depth of ~50 not seldom reached

CERN School of Computing

Processes and Threads: Pricetags

Process:

- (+) Isolated (different address spaces)
- (+) Easy to manage
- (−) Communication between them possible but pricey
- (−) Price to switch among them

Threads:

- (+/−) Sharing memory (communication is a memory access)
- (+) Lower overhead for creation, lower coding effort
- (++) Fit well many-cores architectures
- (++) Ideal for a task-based programming model

CERN School of Computing

Threads or Processes?

Some additional elements to consider for the decision:

- Amount of legacy code and resources available to make it thread-safe
- Duration of tasks wrt the overhead of the forking process
- Presence of shared states and their behaviour in presence of contention
    - E.g. Disk I/O, DB I/O, common data structures (e.g. “HEP event”)

CERN School of Computing

Threads in C++

- C++ offers a construct to represent a thread: `std::thread`
- Interfaced to the underlying backend provided by the OS – 100% portable:
- A function (a *callable* in general) can be executed within a thread asynchronously
- Many more possibilities than the simple `std::async` execution
  - Full control on the thread!

CERN School of Computing

Threads example

Header for `std::thread`

```
#include <thread>
#include <iostream>

int main(){
  std::thread t([]{std::cout << "Hello Concurrent World!\n"; });
  t.join();
  }
```

Create and start a thread

Lambda function

Wait for the thread to finish its job

- In general, it is possible that the thread does not need to be joined
  - A "daemon thread": the method to use is `std::thread::detach()`
  - Once detached, the thread cannot be joined anymore!
- Possible usecases: I/O, monitor filesystems, clean caches…

CERN School of Computing

A First Abstraction

A possible prototype backend behind task oriented programming!

```
#include <thread>
#include <vector>
#include <iostream>

void printThreadID(int i){
 printf("thread num %d - id %2x\v", i,std::this_thread::get_id);
 }

int main(){
 std::vector<std::thread> myThreads; myThreads.reserve(10);
 for (int i=0; i<10; i++)
    myThreads.emplace_back(printThreadID, i);

 for (auto& t : myThreads)
    t.join();
}
```

Identify the thread

The first step towards automating the management of threads in the application!

Limitation: cannot retrieve the return value.

CERN School of Computing

A First Abstraction

A possible prototype backend

```
#include
#include
#include

void pr
 printf(                                                        );
 }

int mai
 std::ve
 for (i
     myTh

 for (auto& t : myThreads)
     t.join();
}
```

ng!

```
-> g++ -std=c++17 -lpthread -o myTest myTest.cpp
-> ./myTest
thread num 0 - id 139708894000896
thread num 5 - id 139708852037376
thread num 3 - id 139708868822784
thread num 2 - id 139708877215488
thread num 4 - id 139708860430080
thread num 8 - id 139708826859264
thread num 1 - id 139708885608192
thread num 7 - id 139708835251968
thread num 6 - id 139708843644672
thread num 9 - id 139708818466560
```

When dealing with concurrency, asynchronous events are daily business!

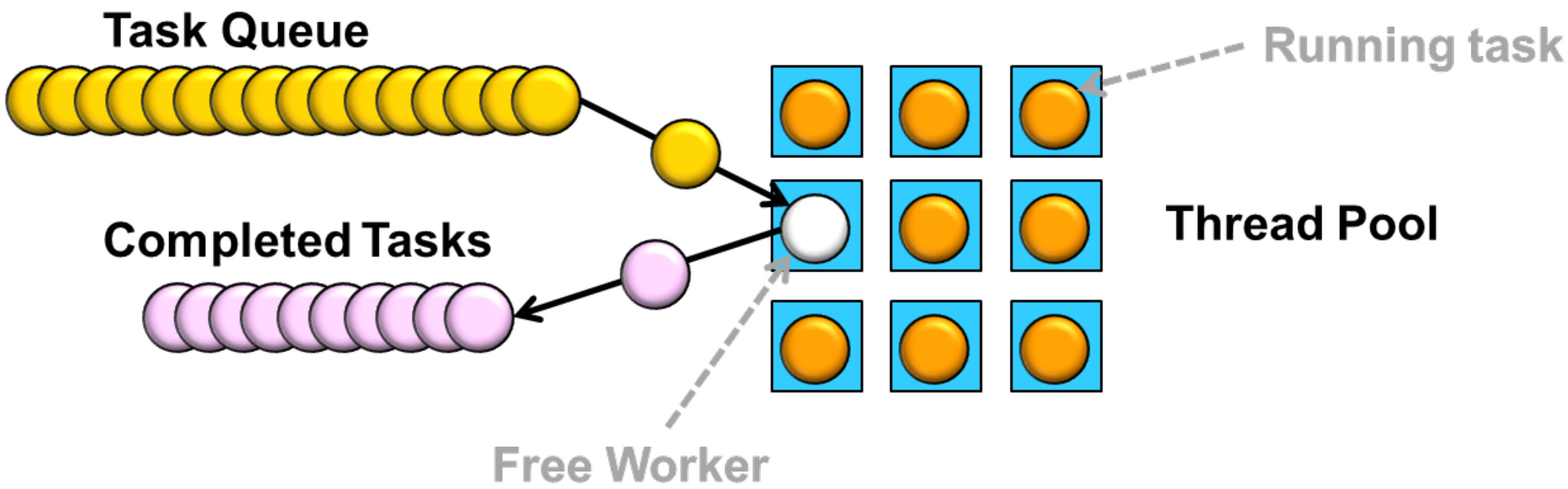
Limitation: cannot retrieve the return value.

The first step towards automating the management of threads in the application!

CERN School of Computing

The Thread Pool Model

- Thread pool: ensemble of worker threads which are …
- Initialised once, consuming work from …
- .. A work queue …
- .. to which elements of work (lambdas, tasks, …) can be added



Hard to program in an optimised and general way!
(usually provided by 3rd part libraries)

CERN School of Computing

Processes in Python/C++

Python

- Handy *multiprocessing* module

C++

- Nothing in the STL
- Some alternative libraries, e.g. *ROOT** TProcessExecutor

```
from multiprocessing import Process, Pool

def f(name):
    print('hello', name)

def g(x):
    return x*x

p = Process(target=f, args=('bob',))
p.start()
p.join()

p = Pool(5)
p.map(g, [1, 2, 3])
```

- No memory shared: need to *serialise* objects to communicte
- Natural in Python, advanced in C++: needs serialisation!

* root.cern.ch

Threads and Data Races: Synchronisation Issues

CERN School of Computing

The Problem

- Fastest way to share data: access the same memory region
  - One of the advantages of threads
- **Parallel memory access: delicate issue - *race conditions***
  - I.e. behaviour of the system depends on the sequence of events which are intrinsically asynchronous
- **Consequences, in order of increasing severity**
  - Catastrophic terminations: segfaults, crashes
  - Non-reproducible, intermittent bugs
  - Apparently sane execution but data corruption: e.g. wrong value of a variable or of a result

Operative definition: An entity which cannot run w/o issues linked to parallel execution is said to be thread-unsafe (the contrary is thread-safe)

CERN School of Computing

To Be Precise: Data Race

Standard language rules, § 1.10/4 and /21:

• Two expression evaluations **conflict** if one of them **modifies** a memory location (1.7) and the other one accesses or **modifies** the same memory location.

• The execution of a program contains a **data race** if it contains two conflicting actions in different threads, at least one of which is **not atomic**, and **neither happens before the other**. Any such data race results in undefined behaviour.

CERN School of Computing

Simple Example

Concurrency can compromise correctness

- Two threads: A and B, a variable X (44)
- A adds 10 to a variable X
- B subtracts 12 to a variable X

2 Threads only
No crash
Bogus results!

A then B

Thread A	Thread B	X Val.
Read X (44)		44
Add 10	Read X (44)	44
Write X (54)	Subtract 12	54
	Write X (32)	32

RACE

Desired

Thread A	Thread B	X Val.
	Read X (44)	44
	Subtract 12	44
	Write X (32)	32
Read X (32)		32
Add 10		32
Write X (42)		42

B then A

Thread A	Thread B	X Val.
	Read X (44)	44
Read X (44)	Subtract 12	44
Add 10	Write X (32)	32
Write X (54)		54

RACE

CERN School of Computing

What is not Thread Safe?

Everything, unless explicitly stated!

In four words: **Shared States Among Threads**

Examples:

- Static non const variables
- **STL containers**
  - Some operations are thread safe, but useful to assume none is!
  - Very well documented (e.g. **http://www.cplusplus.com/reference**)
- Many random number generators (the stateful ones)
- Calls like: `strtok, strerror, asctime, gmtime, ctime` …
- Some math libraries (statics used as cache for speed in serial execution…)
- Const casts, singletons with state: indication of unsafe policies

It sounds depressing. But there are several ways to protect thread unsafe resources!

Useful Design Principles

CERN School of Computing

Minimise Contention

- Designing and implementing software for the serial case to make it parallel afterwards
  - Not exactly a winning strategy
- Rather think parallel right from the start
  - Advice not straightforward to put in place
  - Needs careful planning and thinking
- Depends on the problem being studied
  - Understand what you are doing!

CERN School of Computing

Ex. Functional Programming Style

Operative definition: computation as evaluation of functions the result of which depends only on the input values and not the program state.

- Functions: **no side effects, no input modification, return new values**

Example of 3 functional languages: Haskell, Erlang, Lisp.

C++: building blocks to implement functional programming. E.g.

- Move semantics: can return entities w/o overhead
- Lambdas & algorithms: map a list of values to another list of values.
- Decompose operations in functions, percolate the information through their arguments

> Without becoming purists, functional programming principles can avoid lots of headaches typical of parallel programming

CERN School of Computing

Replication, Atomics, Transactions and Locks

CERN School of Computing

Why so many strategies?

- There is no silver bullet to solve the issue of "resource protection"
  - Complex problem
- Case by case investigation needed
  - Better to be aware of many strategies
- Best solution: often a trade-off
  - The lightest in the serial case?
  - The lightest in presence of high contention?

CERN School of Computing

One copy of the data per Thread

- Sometimes it can be useful to have thread local variables
  - A “private heap” common to all functions executed in one thread
- Thread Local Storage (TLS)
- Replicate per thread some information
  - C++ keyword `thread_local`
- E.g.: build “smart-thread-local pointers”
  - Deference: provide the right content for the current thread
- Not to “one size fits them all” solution
  - Memory usage
  - Overhead of the implementation, also memory allocation strategy
  - Cannot clutter the code with `thread_local` storage specifiers

CERN School of Computing

TLS in Action

```
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>

thread_local unsigned int tlIndex(0);

std::mutex myMutex;
void IncrAndPrint(const char* tName,unsigned int i){
 tlIndex+=i;
 std::lock_guard<std::mutex> myLock(myMutex);
 std::cout << tName << " - Thread loc. Index " << tlIndex
           << std::endl;
};

int main(){
 auto t1 = std::thread(IncrAndPrint,"t1",1);
 auto t2 = std::thread(IncrAndPrint,"t2",2);
 IncrAndPrint("main",0);
 t1.join(); t2.join();
}
```

One private copy per thread will exist

Be patient for a moment ;-)

Thread 1, 2 and main thread (de facto just “threads” for the OS)

CERN School of Computing

TLS in Action

```
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>

thread_local unsigned int tlIndex=0;

std
voi                                         ned int i){
 tl
 st                                         ex);
 std::cout << tName << " - Thread loc. Index " << tlIndex
           << std::endl;
};

int main(){
 auto t1 = std::thread(IncrA
 auto t2 = std::thread(IncrA
 IncrAndPrint("main",0);
 t1.join(); t2.join();
}
```

One private copy per thread will exist

Be patient for a moment ;-)

Possible output:

```
main - Thread loc. Index 0
t2 - Thread loc. Index 2
t1 - Thread loc. Index 1
```

Possible output w/o tls (not correct!):

```
main - Thread loc. Index 0
t2 - Thread loc. Index 3
t1 - Thread loc. Index 3
```

Thread 1, 2 and main thread (de facto just “threads” for the OS)

CERN School of Computing

Atomic Operations

- Building block of thread safety: **an atomic operation is an operation seen as non-splitable by other threads**
  - Other real life examples: finance, database transactions
  - Either entirely successful (subtract from A, add to B) or rolled back
- C++ offers support for atomic types
  - `#include <atomic>`
  - Usage: `std::atomic<T>`

A then B

Thread A	Thread B	X Val.
Read X (44)		44
Add 10	Read X (44)	44
Write X (54)	Subtract 12	54
	Write X (32)	32

RACE

Desired

Thread A	Thread B	X Val.
	Read X (44)	44
	Subtract 12	44
	Write X (32)	32
Read X (32)		32
Add 10		32
Write X (42)		42

B then A

Thread A	Thread B	X Val.
	Read X (44)	44
Read X (44)	Subtract 12	44
Add 10	Write X (32)	32
Write X (54)		54

RACE

- Operations supported natively vary according to T
  - Subtleties present: e.g. cannot instantiate `atomic<MyClass>` under all circumstances (must be *trivially copyable*)
- For example:
  - boolean, integer types. E.g. `std::atomic<unsigned long>`
  - Pointer to any type. E.g. `std::atomic<MyClass*>`

CERN School of Computing

Atomic Counter

```
#include <atomic> …

std::atomic<int> gACounter;
int gCounter;

void f(){ //increment both
  gCounter++;gACounter++;}

int main(){
std::vector<std::thread> v;
v.reserve(10);

for (int i=0;i<10;++i)
  v.emplace_back(std::thread(f));
for (auto& t:v) t.join();

std::cout << "Atomic Counter: "
          << gACounter << std::endl
          << "Counter: "
          << gCounter << std::endl;
}
```

```
$ g++ -o atomic atomic.cpp -std=c++14 -lpthread
$ ./atomic
Atomic Counter: 10
Counter: 9
$./atomic
Atomic Counter: 10
Counter: 10
```

2 trials...

3 observations:

- Atomics allow **highly granular resources protection**.
- Real life example: incorrect reference counting leads to double frees!
- **Bugs in multithreaded code** can have *extremely* subtle effects **and are in general not-reproducible!**

Locks and Mutexes

- Make a section of the code executable by one thread at the time
- Locks should be avoided, but yet known
  - They are a blocking synchronisation mechanisms
  - They can suffer pathologies
  - … they could be present in existing code: use your common sense and a grain of salt!

Terminology:

- Before the section, the thread is said to *acquire a lock on a mutex*
- After that, no other thread can acquire the lock
- After the section, the thread is said to *release the lock*

Lock Classification

A lock can be …

- ***a spin lock:*** if it makes a task spin while waiting (“busy wait”)
  - Short tasks: spin is better (putting a thread to sleep costs cycles)
  - Big implications also in terms of power consumption
- ***Scalable:*** cannot perform worse than serial execution
- ***Fair:*** it lets threads through in the order the they arrive
- ***Recursive:*** it can be acquired multiple times by the same thread

Each attribute comes with a pricetag: an unfair, non-scalable, non-reentrant lock might be ideal in some situations if faster than others!

CERN School of Computing

A first Lock Example

Acquire/release lock on the mutex

```
[…]
std::mutex gMutex;
void g(){
  std::lock(gMutex);
  doWork();
  std::unlock(gMutex);
  }
[…]
```

Only one thread at the time can access this section

CERN School of Computing

A first Lock Example

Acquire/release lock on the mutex

```
[…]
std::mutex gMutex;
void g(){
  std::lock(gMutex);
  doWork();
  std::unlock(gMutex);
  }
[…]
```

Only one thread at the time can access this section

- Potential issue: `doWork()` throws an exception
- The lock is never released: the program will stall forever
- A possible solution: *a scoped lock* (seen in the previous slides!)

CERN School of Computing

Scoped Locks: the Proper Way

Instance of a class, locks the scope!

```
[…]
std::mutex gMutex;
void g(){
  std::lock_guard<std::mutex> lg(gMutex);
  doWork();
}
[…]
```

- Construct an object which lives in the scope to be locked
- C++ provides a class to ease this: `std::lock_guard<T>(T&)`
- When the scope is left, the object destroyed and the lock released
- **Application of the RAII idiom (Resource Acquisition Is Initialisation)**
  - RAII: “bread and butter” in modern and performant C++

CERN School of Computing

Pathologic Behaviours of Locks

Deadlock: Two tasks are waiting for each other to finish in order to proceed.

- One task tries to acquire a lock it already acquired and the mutex is not recursive

Convoying: A thread holding a lock is interrupted, delayed (by the OS, to do some I/O). Other threads wait that it resumes and releases the lock.

Priority inversion: A low priority thread holds a lock and makes a high priority one wait.

Lock based entities do not compose: the combination of correct components may be ill behaved.

CERN School of Computing

Good Practices with Locks

- **Don't use them if possible**
- … Really, don't!
- **Hold locks for the smallest amount of time possible**
- Avoid nested locks
- Avoid calling user/library code you don't control which holds locks
- Acquire locks in a fixed order

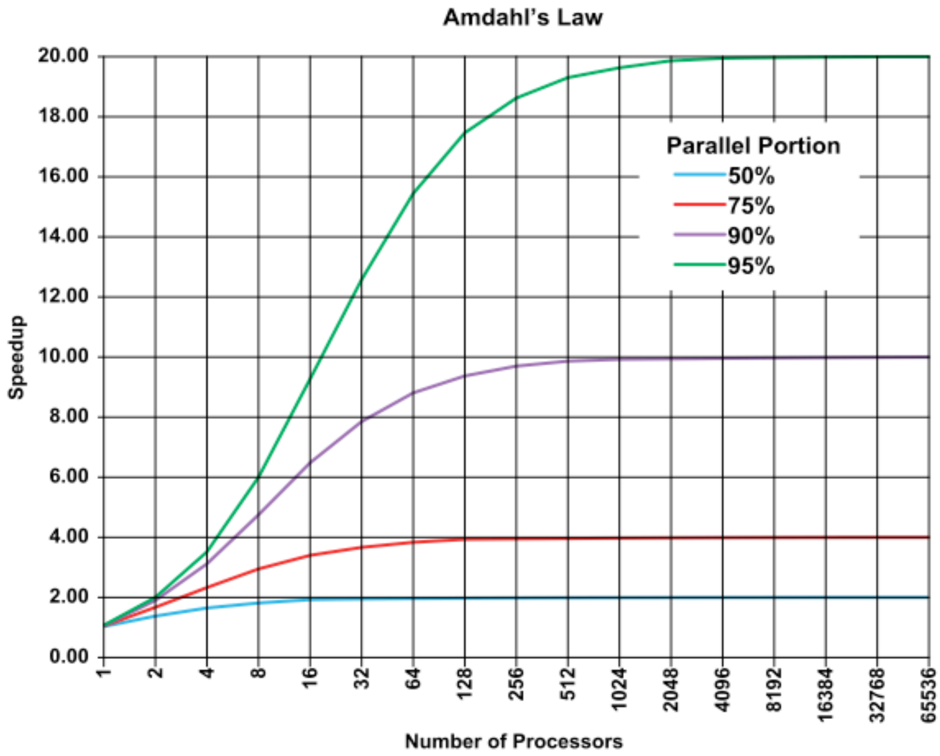
CERN School of Computing

Amdahl's Law

- It predicts the maximum speedup achievable given a problem of **fixed size**

$$Speedup = \frac{1}{(1-p)+\frac{p}{n}}$$

n: number of cores
p: parallel portion



"… the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." - 1967

From E. Tejedor

CERN School of Computing

Take Away Messages

- **Choose designs that follow principles such as data and task parallelism**
  - They lead to scalable and performant applications
  - Focus on algorithms and data structures!
- **Asynchronous execution and non-determinism permeate concurrent applications**:
  - Paradigm shift needed to understand and design parallel software solution
- Abstraction needed: e.g. thread pool
  - Do not forget the basics: ownership, OS, hardware
- Choose from the start a **design which helps avoiding data races:**
  - Understand your problem: no silver bullet
  - Prefer approaches w/o global states (e.g. functional)
- **Choose non blocking mechanisms** whenever possible
  - E.g. atomics and transactions
  - Locks can be present in existing software
  - Use a grain of salt