CERN School of Computing

Lecture 3

Expressing Parallelism Pragmatically

This Lecture

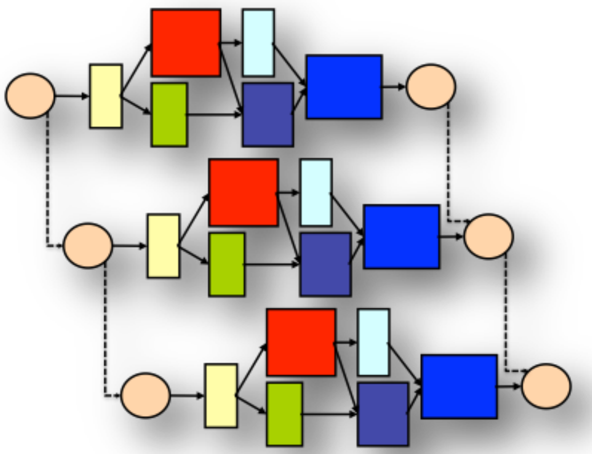
The Goals:

1) Understand the difference between data and task parallelism and the potential of their combination
2) Become more aware about hardware and OS features related to multithreading
3) Appreciate the usefulness of abstraction from details achieved through a 3^{rd} party library

The outline:

- Parallel software design: an introduction
- Threads and parallelism in C++
- Elements of Threading Building Blocks

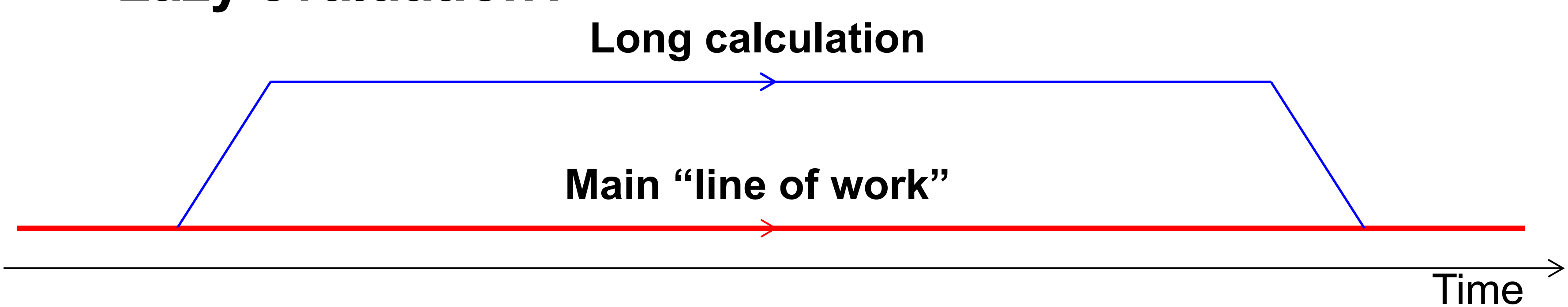
Asynchronous Execution



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

```
#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

Header for async and future

"Launch" the calculation

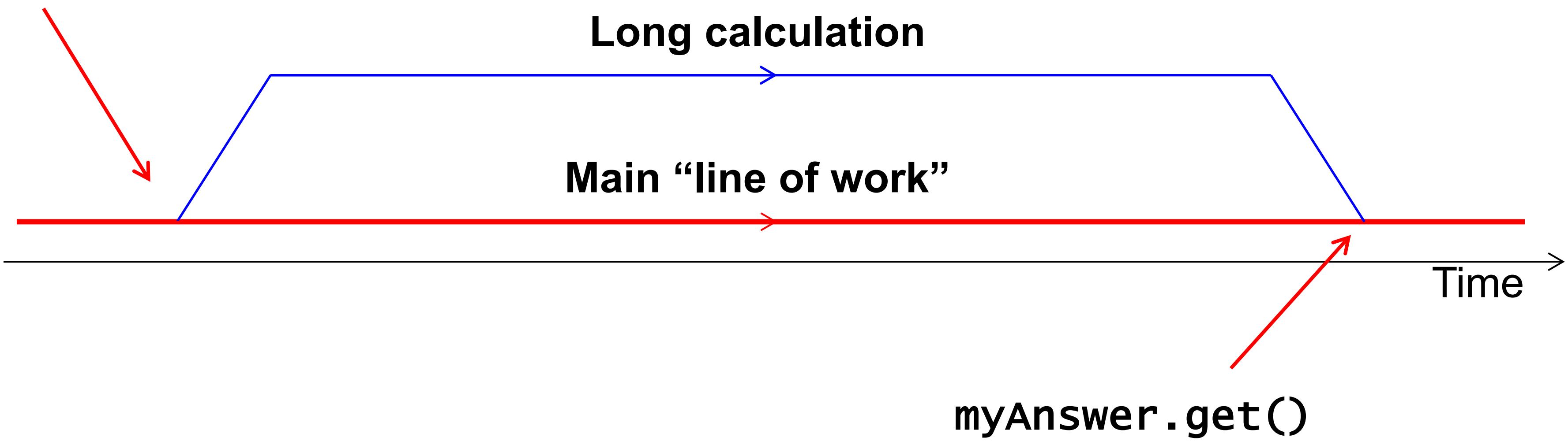
Retreive result

```
#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;
  }
```

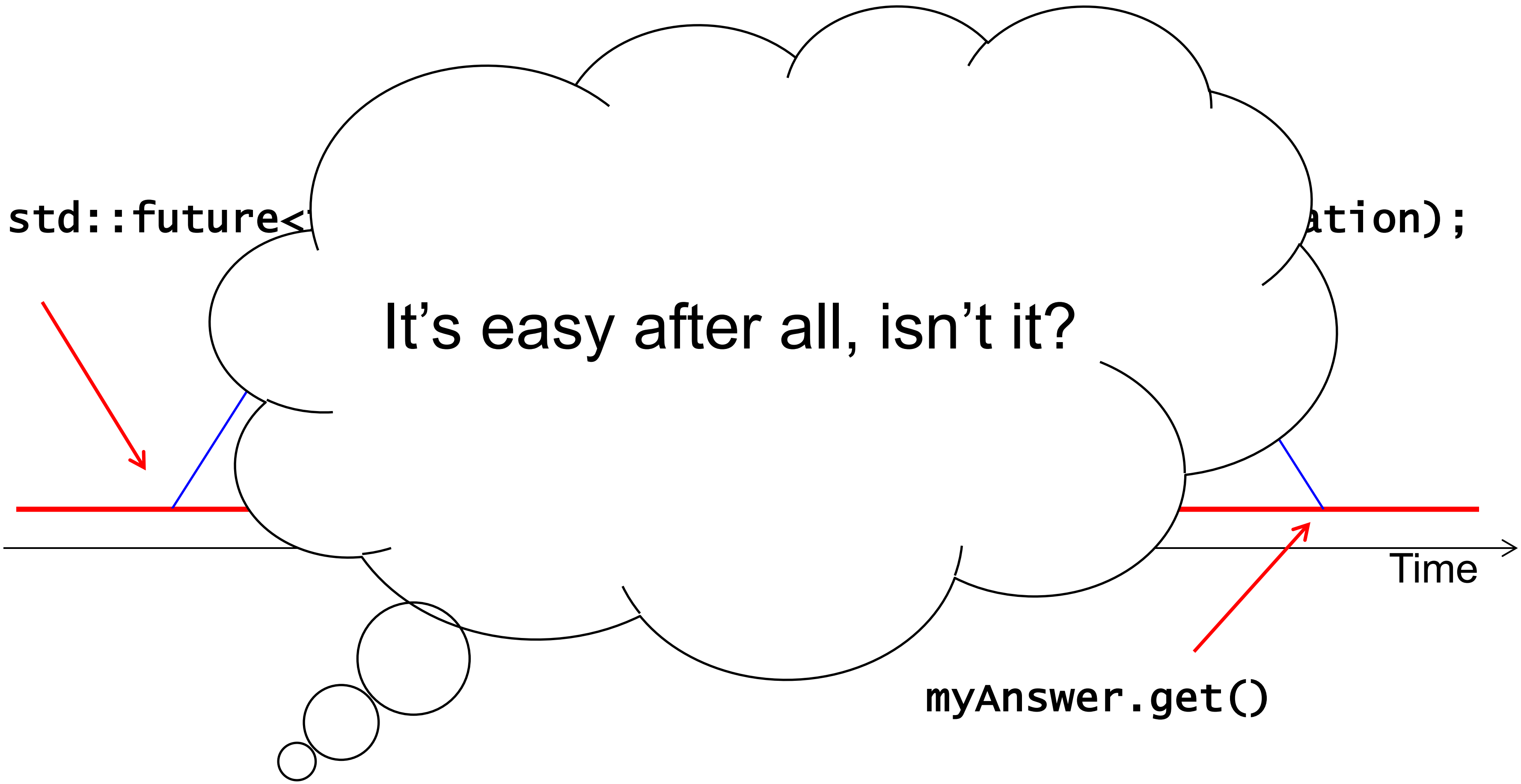
- `std::async` can have a second parameter, the "policy":
  - `std::launch::async`: execute function in a new separate thread
  - `std::launch::deferred`: defer call until `get()` is called (lazy)
  - Default: "`async` or `deferred`", the implementation chooses!

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 non parallel large software systems 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 ☺

CERN School of Computing

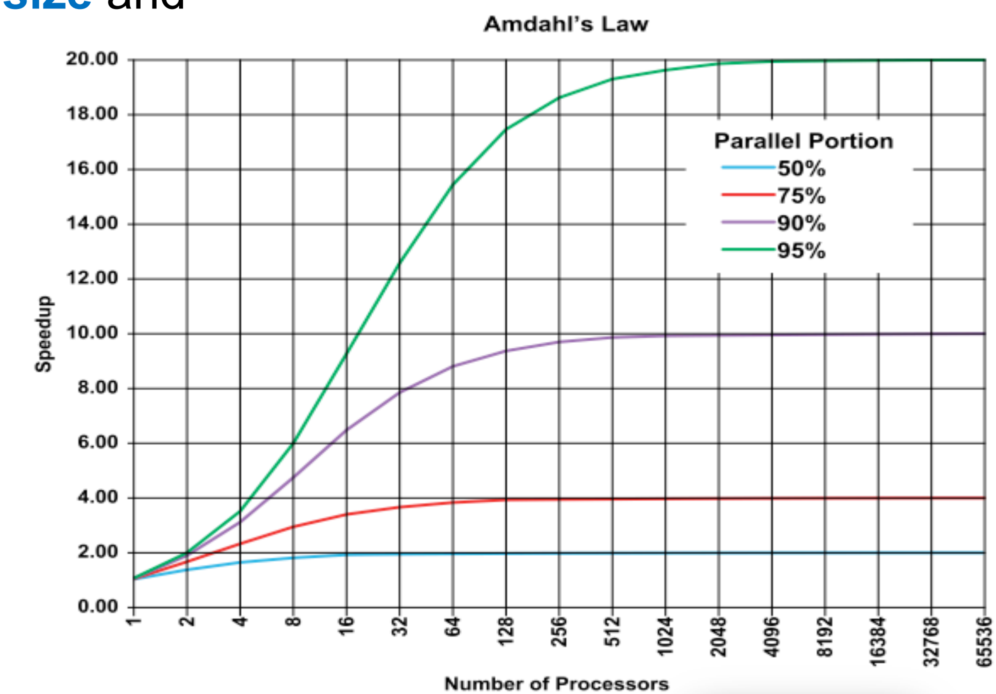
Amdahl's Law

It tells us something about parallel execution:
It states the **maximum speedup achievable** given a **certain problem of FIXED size** and **sequential portion** of the program.

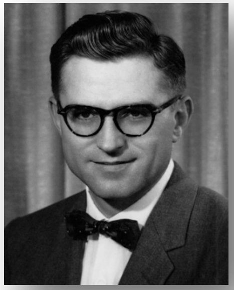
$$\Delta t = \Delta t_0 \cdot \left[(1-P) + \frac{P}{N} \right]$$

$$Speedup = \frac{\Delta t_0}{\Delta t} = \frac{1}{(1-P) + \frac{P}{N}}$$

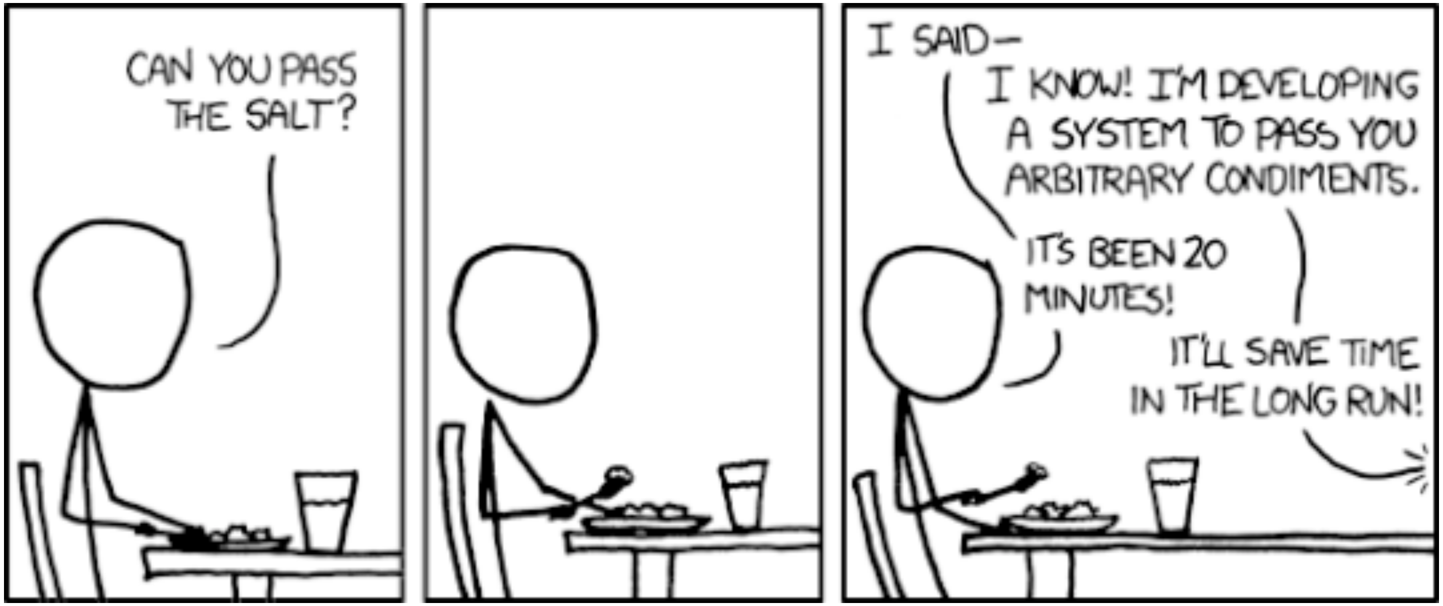
N: number of workers
P: parallel portion
Δt_0: serial exec. time



"… 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



Parallel Software Design: an Introduction



CERN School of Computing

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

CERN School of Computing

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

This step takes place in front of a whiteboard

First Step: Finding Concurrency

What can be executed concurrently?

Two techniques to figure this out:

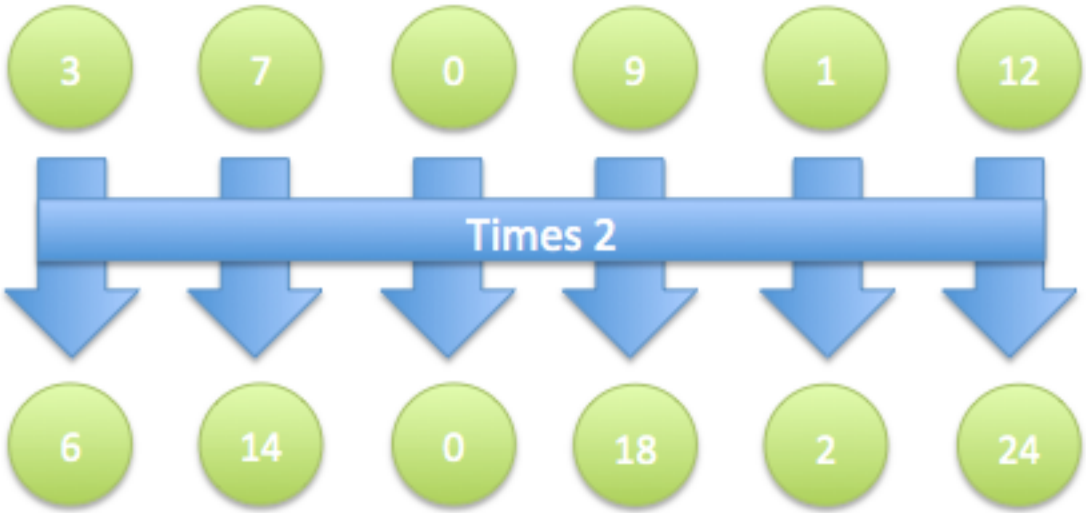
- ***Data decomposition***
  - The partition of the data domain
  - Achieve data parallelism
- ***Task decomposition***
  - Split according to logical tasks
  - Achieve task parallelism

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

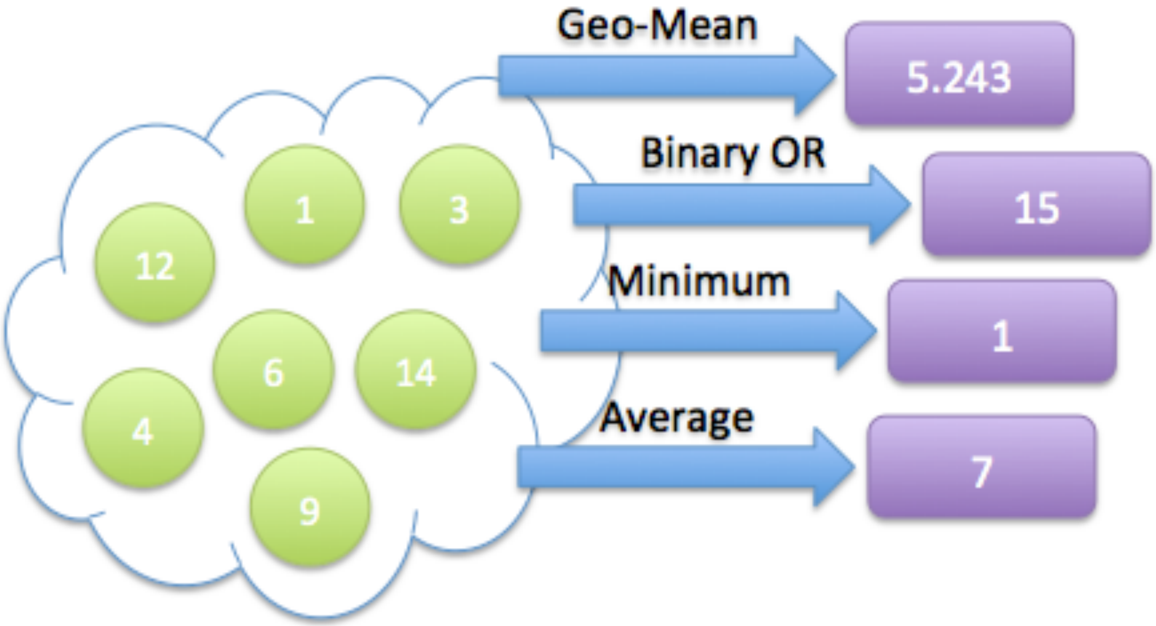
Increase floating point throughput acting on mathematical functions:

- 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**, **AMD Libm**, **VDT, Yeppp** libraries.

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



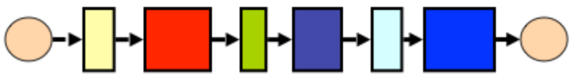
A bit too simple: no dependency between tasks!

Task Parallelism: An example

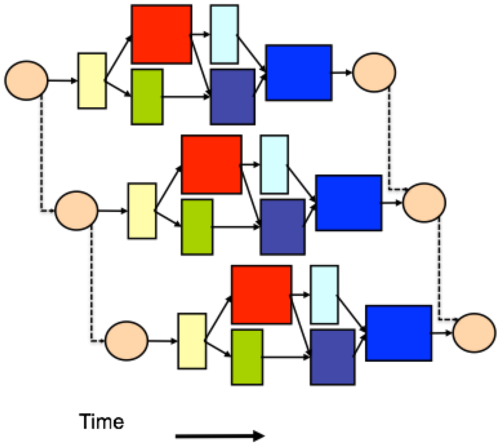
HEP data processing frameworks

- 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

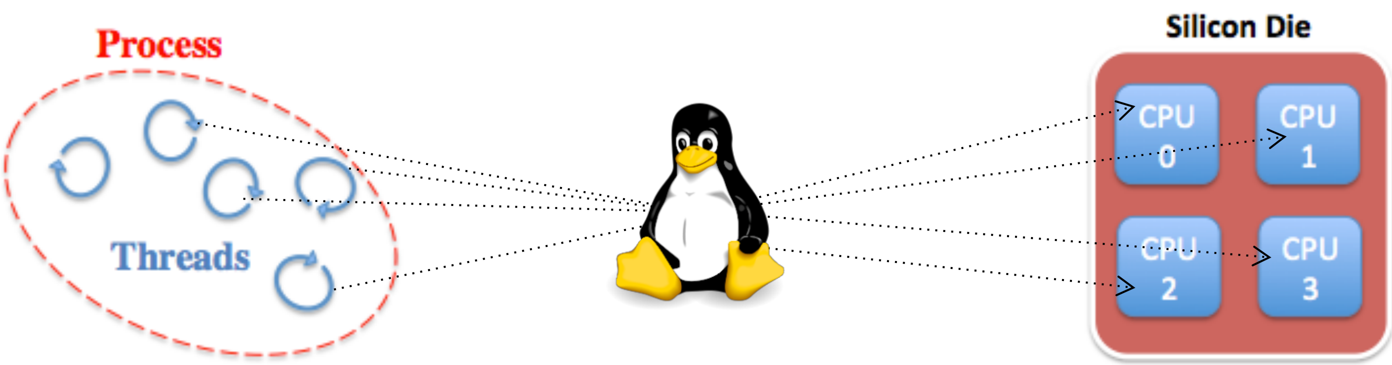
- We do not need to “choose” to approach a problem with a task or data parallelism based solution
- Actually, pure task/data parallelism is rare!
- Combining the two is the key

Threads and C++

CERN School of Computing

Let's change gears: Threads

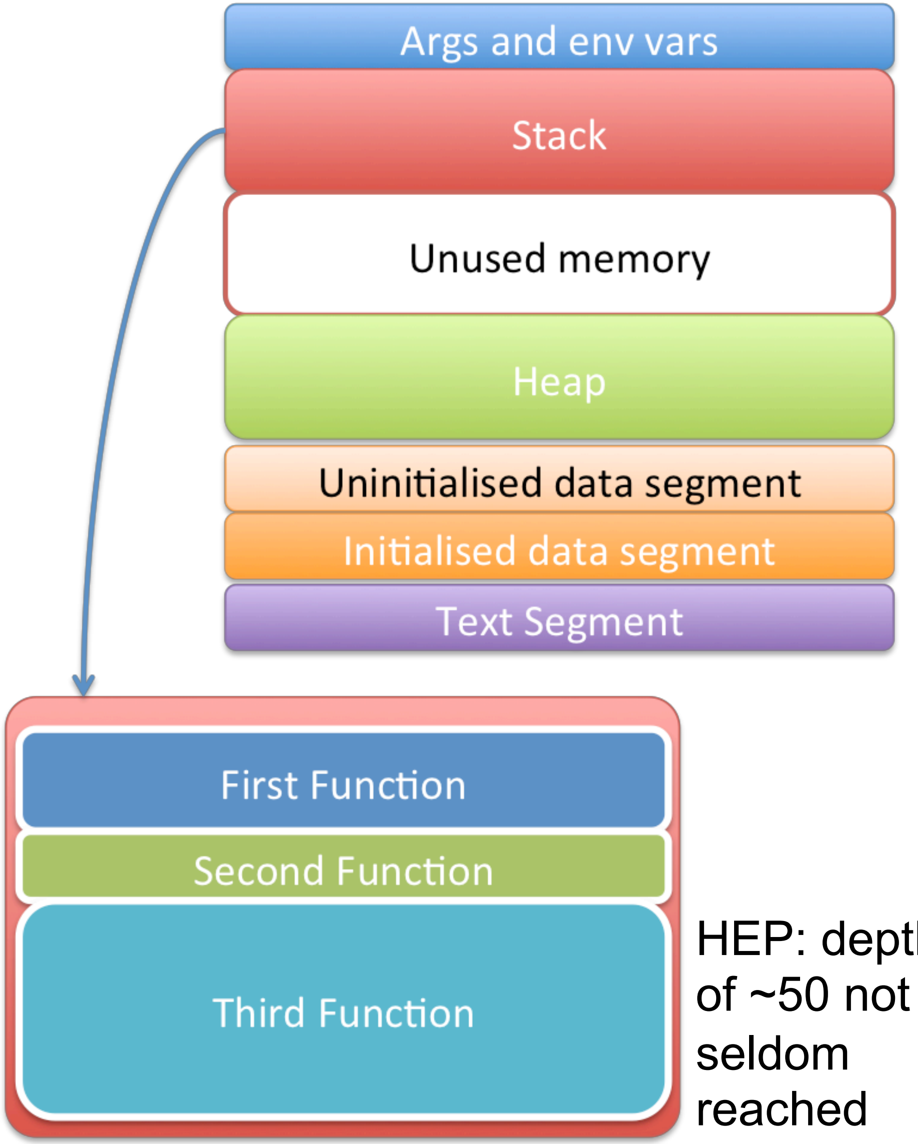
- From the operating system point of view:
  - **Process**: isolated instance of a program, with its own space in (virtual) memory, can have multiple threads
  - **Thread**: light-weight process within process, **sharing the memory** with the other threads living in the same process
- The kernel manages 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`").



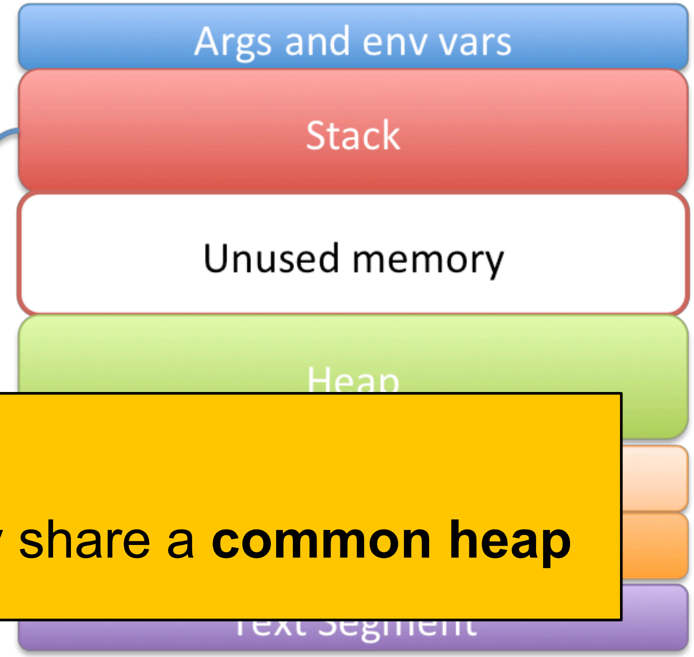
HEP: depth of ~50 not seldom reached

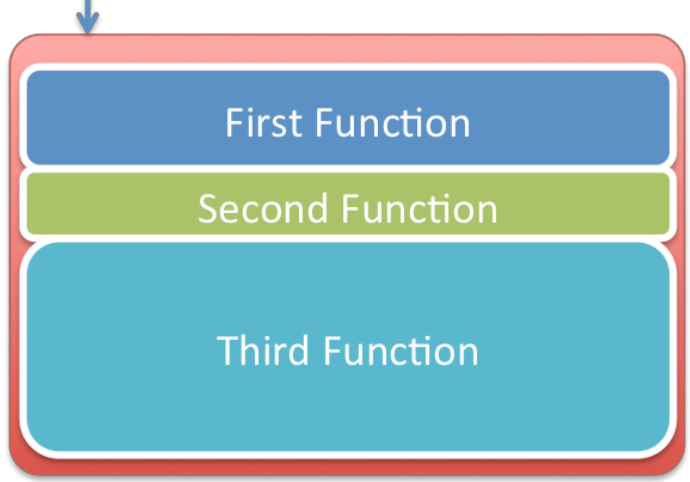
Interlude: A Program in Memory

- **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

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”)

Threads in C++

- C++ offers a construct to represent a thread: `std::thread`
- Interfaced to the underlying backend provided by the OS – 100% portable:
  - Linux: `pthreads`
  - Windows: Windows threads
  - …
- 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

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


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

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 Pitfall with Threads

```
#include <thread>
#include …

void g(){
  std::string s(“Hello\n“);
  std::thread t([&s]{std::cout << s;});
  t.detach(); // lets the thread run w/o the need for joining
  }
```

CERN School of Computing

A Pitfall with Threads

```
#include <thread>
#include …

void g(){
  std::string s(“Hello\n“);
  std::thread t([&s]{std::cout << s;});
  t.detach(); // lets the thread run w/o the need for joining
  }
```

String s lives in the scope of function g

Captured by reference!

Parallel programs: variables' lifetime even more important than in serial world

Typical behaviour of the example above:

- Function g terminates before the lambda: s is a dangling reference!
- Corruption and segfaults are guaranteed

A Pitfall with Threads

```
#include <thread>
#include …

void g(){
  std::string s(“Hello\n“);
  std::thread t([s]{std::cout << s;});
  t.detach();
  }
```

Always carefully consider ownership!

- A possible solution: create a string object and pass it by value
- But it’s a copy of a string! Yes.
- The phase-space of design and implementation choices significantly expands when introducing concurrency!

CERN School of Computing

How To Manage Threads?

- Direct utilisation of threads: works (well) for simple cases
- Difficult to scale:
  - Risk of a proliferation of threads
- Need a more abstract model
  - Task oriented programming

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();
}
```

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

ng!

```
#include
#include
#include

void pr
 printf(                                                    );
 }

int mai
 std::ve
 for (i
     myTh

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

```
-> 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

Getting Back the Return Value

- `std::packaged_task:` wraps a callable (function, lambda, bind), gives handle to the result via a future

```
// includes of: <thread> <functional> <future> <algorithm> <utility>

bool isReachable(long ip){[…];return pingResult;};
[…]
std::vector<std::thread> threads;
std::vector<std::future<bool>> availabilities;
for (int i=0; i<10; i++){
 std::packaged_task<bool(long)> task(isReachable);
 availabilities.emplace_back(task.get_future());
 std::thread t(std::move(task),ips[i]);
 threads.emplace_back(std::move(t));
}
for (auto& thr : threads) thr.join(); int n=0;
for (auto& avail : availabilities) if(avail.get()) n++;
std::cout << "Nodes available: " << n << std::endl;
}
```

CERN School of Computing

Getting Back the Return Value

- `std::packaged_task:` wraps a callable (function, lambda, bind), gives handle to the result via a future

```
// includes of: <thread> <functional> <future> <algorithm> <utility>

bool
[…]
std::
std::
for (
  std:
  avai
  std:
  thre
}
for (auto& thr : threads) thr.join(); int n=0;
for (auto& avail : availabilities) if(avail.get()) n++;
std::cout << "Nodes available: " << n << std::endl;
}
```

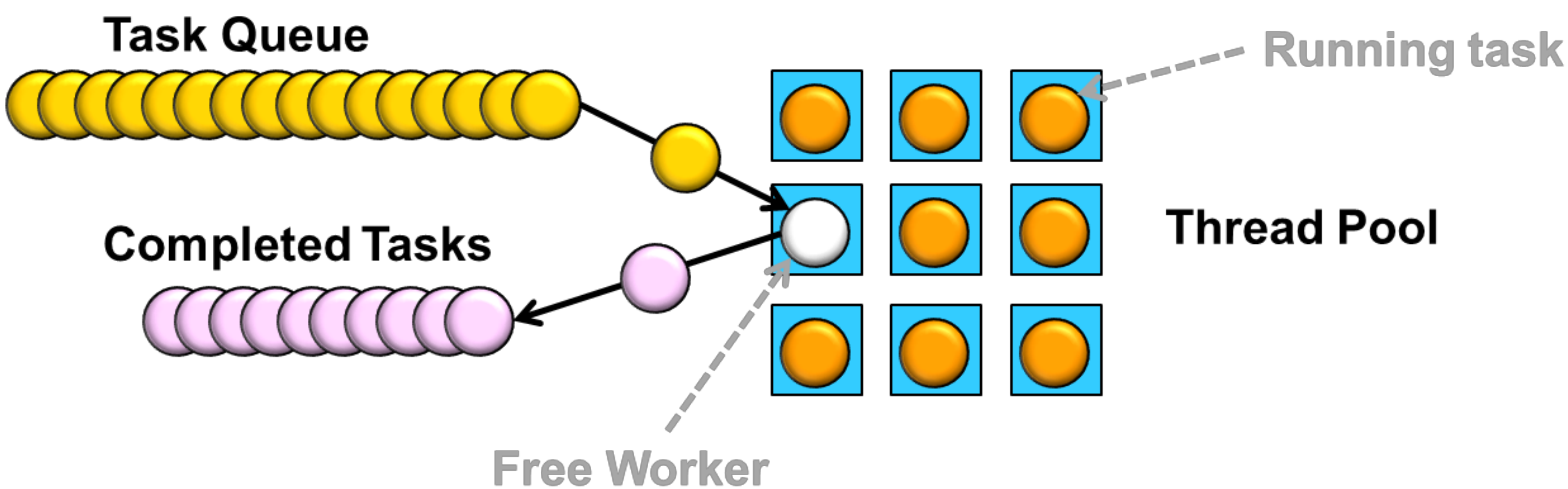
All of this is complex.
Direct thread management does not suit complex applications:

- **Overhead** of creating and destroying threads
- **Risk** to overcommit the machine
- **Better to focus** on “packetisation” of work rather than manual thread management (done at the whiteboard, not at the terminal)

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)

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 introspection!

* root.cern.ch

Elements of TBB

CERN School of Computing

An Example: TBB

www.threadingbuildingblocks.org

- A free and open source C++ library for parallel programming
  - Takes care of managing multitasking
- An Intel product, actively maintained
- A GPL (+ Runtime Exception) - Intel library exists as well
- C++14 ready: lambdas, move semantics
- Can be mixed with other threading mechanisms
  - STL threads
- Good documentation available, comprehensive examples

More in the next lecture

1 2 3

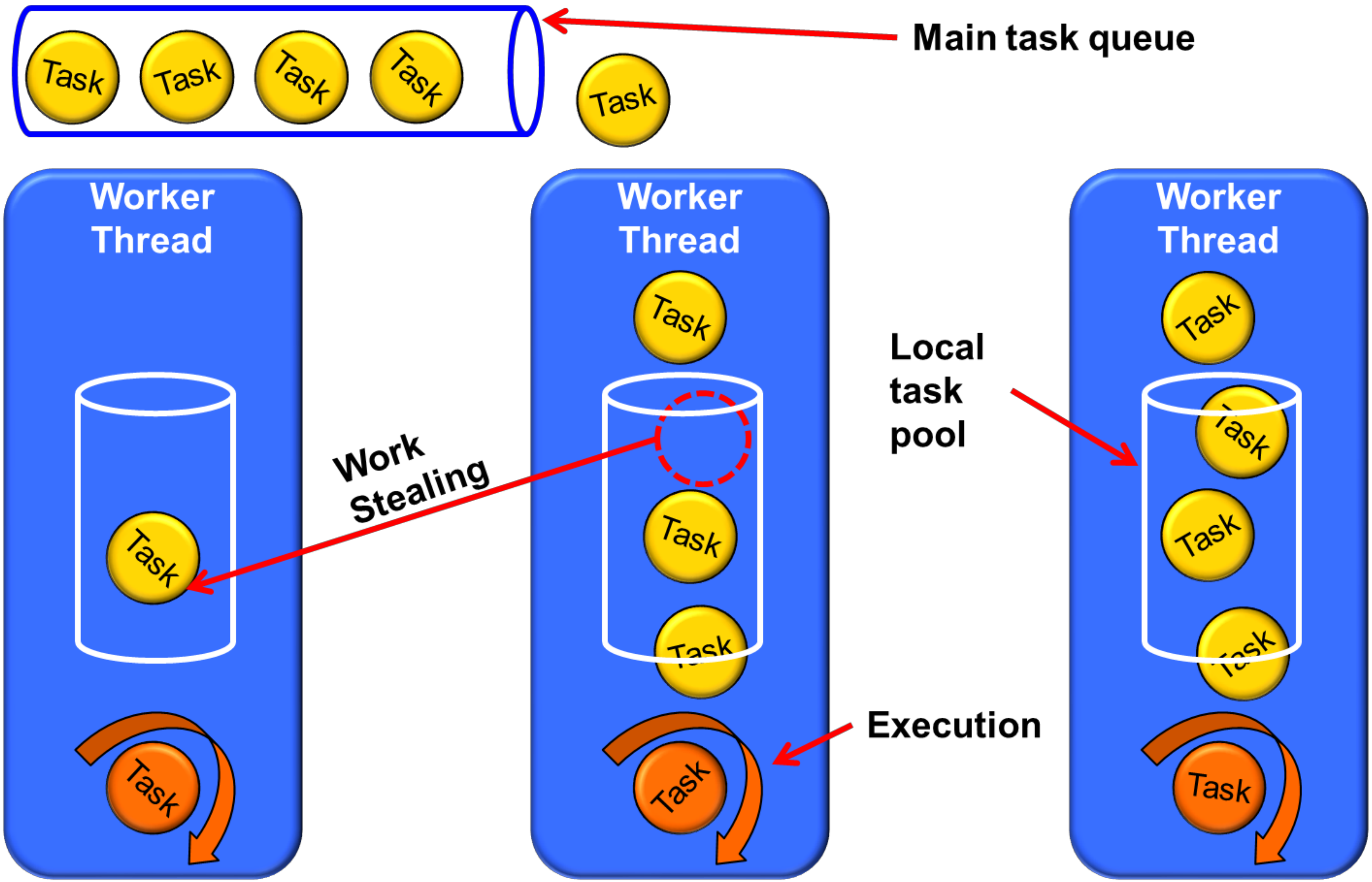
- **Task scheduler**, **generic parallel algorithms**, **concurrent containers**
  - Plus TLS, TBB Flow Graph, synch primitives, memory allocators

Among other features, readily usable implementation of 3 important concepts in parallel programming

TBB Task Scheduler In a Nutshell

- Single main and local thread task queues
- Interfaced with a thread pool (automatically initialised), dispatches work to workers
  - Maps logical work items, tasks, onto physical threads
- Takes care of load balancing. Rule of the thumb: provide many more tasks than threads in the pool
  - Work stealing: before sleeping, a thread overtakes work from other threads
- Tasks organised in a directed graph
  - Tasks can have a continuation tasks (profit from hot caches)
  - Tasks can inject N tasks in the scheduler, they have priority
  - Depth-first approach: deepest -> most recent -> hotter cache
- Lots of features:
  - Sometimes needed: parallel HEP frameworks, File systems

A Graphical Representation



Task in Action

```
#include <tbb/task.h>
Class myTask: public tbb::task {
public:
 myTask([args])
 tbb::task* execute() {
 do_work();
 return 0;
 }
};
myTask* t = new(tbb::task::allocate_root()) myTask([args]);
tbb::task::enqueue(*t);
```

Task in Action

Task interface

```
#include <tbb/task.h>
class myTask: public tbb::task {
public:
 myTask([args])
 tbb::task* execute() {
 do_work();
 return nullptr;
 }
};
auto t = new(tbb::task::allocate_root()) myTask([args]);
tbb::task::enqueue(*t);
```

Inherited method called by TBB runtime

Can return a child task

`Enqueue`: push to main queue
`Spawn`: push into local queue

Special TBB overloaded new. Several available: example: `allocate_child`

Goal: fine tune performance of scheduler.

Other programming models allow task parallelism, for example OpenMP4…

Parallel For

- Task based parallelism behind the scenes
  - Thread pool and scheduler initialised lazily once behind the scenes
- C++ - Template function, lambda: real syntactical advantage
- Partitioning of the work in chunks managed by the runtime
  - Can be tuned

```
void sum(const int* in1, const int* in2, std::size_t size, int* out) {
  tbb::parallel_for(std::size_t(0), size,
    [=](std::size_t i) {
      out[i] = in1[i] + in2[i];
      });
}
```

Lambda

CERN School of Computing

Take Away Messages

- Designs that follows principles like data and task parallelism 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
  - Phase space of possible issues bigger than in the sequential case
  - … And we did not talk about resource protection yet!
- Abstraction needed: e.g. thread pool
  - Do not forget the basics: ownership, OS, hardware.
  - Rely on 3rd party products → more time to focus on your problem