# Software Design in the Many-Cores era

A. Gheata, E. Tejedor

CERN, EP-SFT

CERN School of Computing 2019

# Parallelism in a Modern HEP Data Processing Framework

## Outline of This Lecture

**The Goals:**

1) *Understand why we need parallelisation*
2) *Understand the problem domain of physics processing*
3) *Break down big problems into work items that can be tackled in parallel*
4) *Be aware of the limitations for parallelisation*

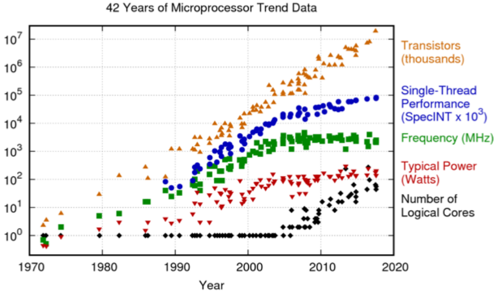
- From sequential to parallel
- Experiment Frameworks: basic principles, design
- Laws of parallelism
- Concurrency Models: task-based parallelism

## Hitting the Wall(s)

- Once upon a time, the life of software developers was much easier
  - Sequential programming
  - Want your program to run faster? Buy yourself a new machine!
- The fairy tale ended in the early 2000s
  - Processor manufacturers had to rethink CPU architectures
  - No more free lunch for software

# The Power Wall

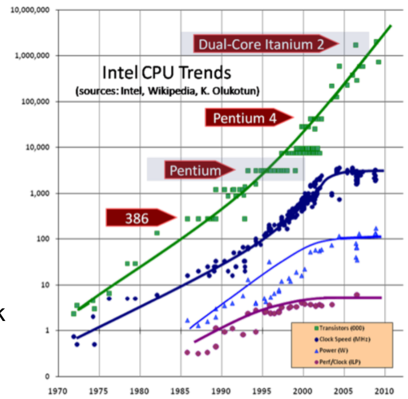
- Manufacturers could not keep improving processor performance by increasing frequency
  - Not at the same rate at least
- **Power consumption and dissipation** became limiting factors
  - Higher clock rate could lead to overheating



https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

# The ILP Wall

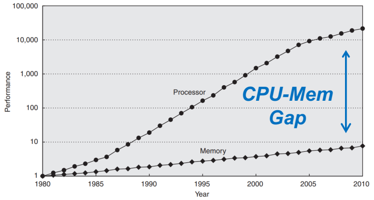
- Processors apply multiple techniques to optimise the execution flow
  - Pipelining
  - Branch prediction
  - Out-of-order execution
  - …
- **Instruction-Level Parallelism growth also flattened**
  - Hard to squeeze more work out of a clock cycle

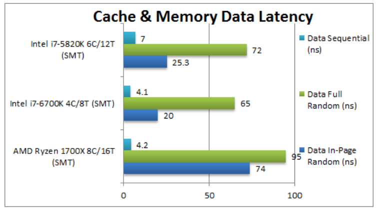


http://www.gotw.ca/publications/concurrency-ddj.htm

# The Memory Wall

- Processor clock rates have been increasing faster than memory clock rates
- **Latency in memory access** is often the major performance issue in modern software applications
- **Larger and faster cache** memories help alleviate the problem but do not solve it
- Often the CPU is just waiting for data…

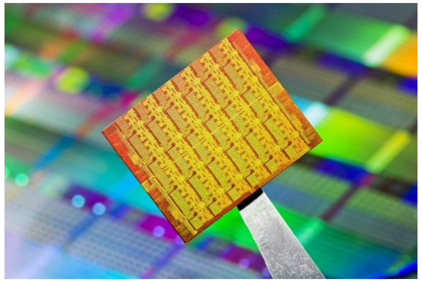




# Multi/Many Core to the Rescue

- Let’s change strategy
  - Grow by **combining simpler processing units**
  - Moore’s law reinterpreted: number of cores per chip will double every two years

***How to make the most of all these resources?***



# From Single to Multi/Many core

| | Irwin-dale | Wood-crest | Gaines-town | Sandy Bridge | Haswell | Broad-well | Skylake |
|---|---|---|---|---|---|---|---|
| Year | 2005 | 2006 | 2009 | 2012 | 2015 | 2016 | 2017 |
| Cores | 1 | 2 | 4 | 8 | 18 | 24 | 28 |
| Freq (GHz) | 3.8 | 3.0 | 3.33 | 2.3 | 2.1 | 2.2 | 2.5 |
| LL Cache | L2 (2MB) | L2 (4MB) | L3 (8MB) | L3 (20MB) | L3 (45MB) | L3 (60MB) | L3 (38MB) |

**Evolution of Intel Xeon processors (https://ark.intel.com)**

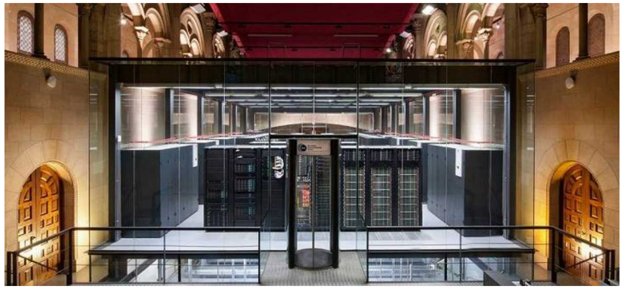
# Need for Parallelism

- Change of programming paradigm
  - Need to deal with systems with many parallel threads
  - Improvement in performance comes with **exploitation of concurrency**
- Will all programmers have to be parallel programmers?
  - Different levels of exposure: **explicit vs. implicit** parallelism
  - First step is to **change the way of thinking**!

***Parallelism is here to stay***

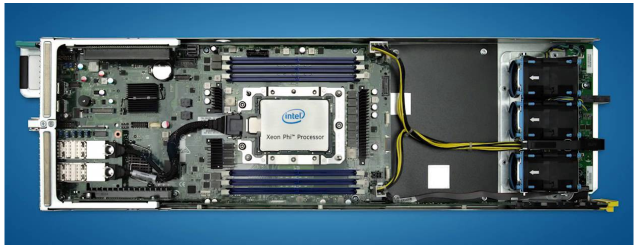
# A Supercomputer

- Perhaps the most striking example of parallelism
  - top500.org: beyond 10M cores
  - Parallelism intra-node and inter-node
  - Multi/many core, hybrid setups: CPU - GPU



# Parallel Hardware

***Accelerators for massive parallelism***



# How is Parallelism Achieved?

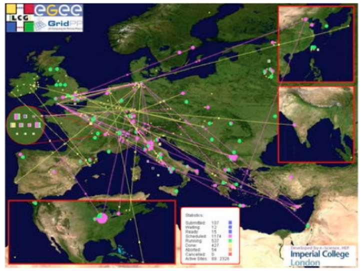
- Supercomputer design tailored for **High-Performance Computing**:
  - Homogeneous nodes (+ accelerators)
  - High-bandwidth low-latency networks (**InfiniBand**, Aries)
  - Parallel distributed file system (**Lustre**, GPFS)

- **Explicit** low-level parallelism dominates
  - **MPI** for distributing processes, message passing
  - **OpenMP** inside a node (+ CUDA, OpenCL)

# Parallelisation in HEP

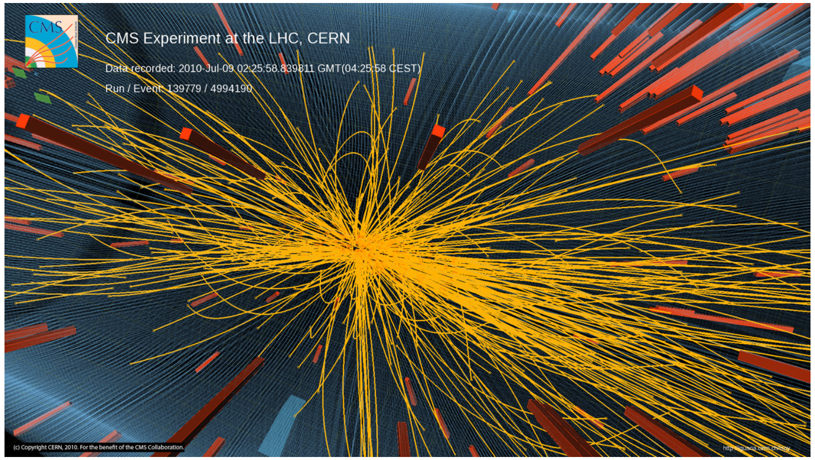
***LHC Computing Grid (WLCG)***

- HEP is parallel since more than a decade
- Computations are distributed among hierarchically organised data centres spread around the globe
- Tens of billions of LHC events are processed per year, running 24/7 365 days a year



**Huge parallel infrastructure!**

# Physics Challenges


CMS
CMS Experiment at the LHC, CERN
Data recorded: 2010-Jul-09 02:25:58.839811 GMT(04:25:58 CEST)
Run / Event: 139779 / 4994190
(c) Copyright CERN, 2010. For the benefit of the CMS Collaboration.

# Physics Challenges II

***So why not treating every many-core computer in the WLCG as a computing centre of its own with many independent jobs on it?***

- Due to the beam intensity ("luminosity") at the LHC multiple proton-proton collisions take place at once (**pile-up**)
- Pile-up expected to increase further in Run 3 and especially in **HL-LHC**
- As a result, memory consumed by experiment' s reconstruction jobs will go up, making it hard to run many simultaneous jobs on a single computer
  - Independent jobs do not share memory!

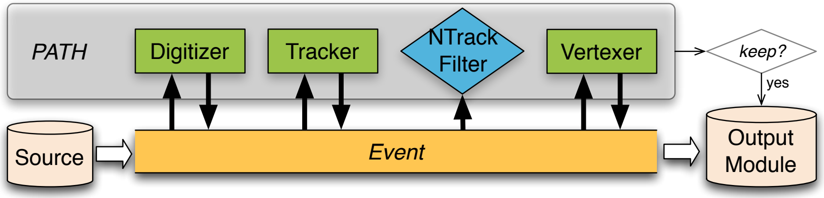
Furthermore:

- **Merging** of results of independent jobs takes significant amount of time

**Another parallelisation strategy is needed!**

# Framework Primer

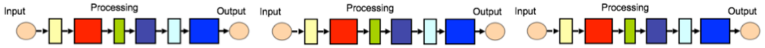
## Experiment Software Follows the Idea of a Software Bus



Each experiment has software with about 5 million lines of code based on this model

# Framework Primer II

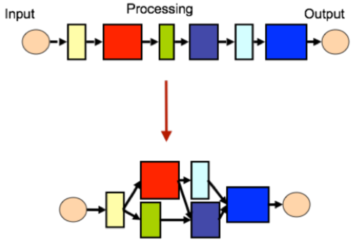
- Multiple events are being processed **sequentially**



- The result is being put into a single output file
- This keeps **only one core busy** at a time

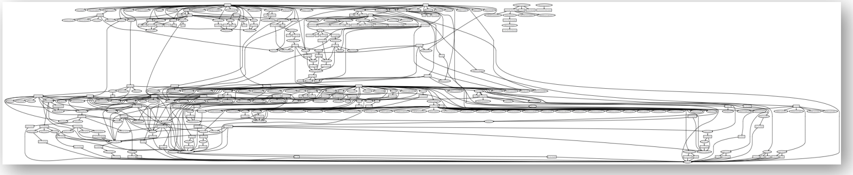
# How to Introduce Concurrency

- The algorithms and their data dependencies form a **DAG** (directed acyclic graph)
  - Schedule the algorithms according to the DAG

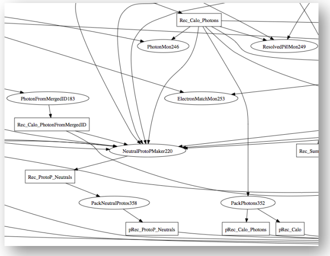


- Sounds more trivial than it is
  - Existing HEP software has many “backdoor” communication channels making the DAG non-obvious.

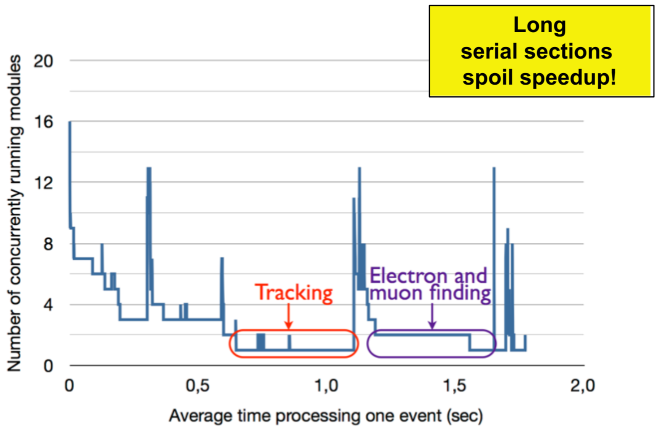
# Real World Example



- Particular example taken from LHCb reconstruction program “Brunel”
- Gives an idea for the potential concurrency
- ATLAS and CMS just don’t fit on a slide…



# The DAG Can Get Narrower



Long serial sections spoil speedup!

# Is Parallelisation Worth It?

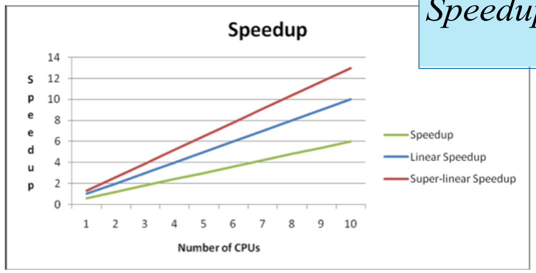
- We hit the wall very early – game over and that’s it?
- Whenever thinking about parallelisation, one should spend some thoughts on whether the effort is worth it
  - The total cost of ownership of one additional box might be smaller than the design-implementation-maintenance costs
- What is the performance gain we can expect?

***Amdahl’s and Gustafson’s laws can help you there!***

# 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**

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

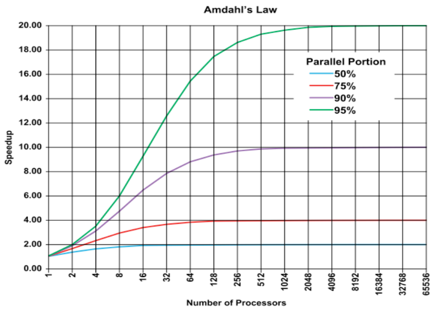


# Amdahl’s Law

- 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***

# Gustafson's Law

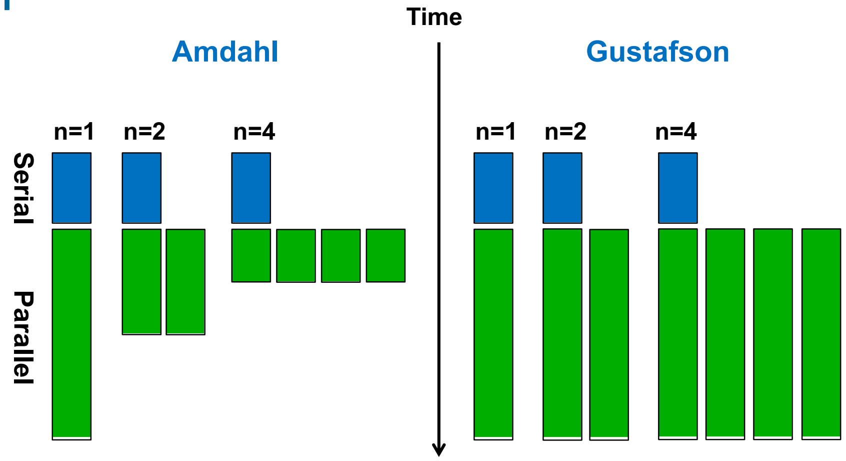
- Often **problem size increases**, while serial parts remain constant
- If problem size increases, so does the opportunity for parallelisation
- Solve bigger problems in the same amount of time by using more resources

$$Speedup = 1 - p + np$$

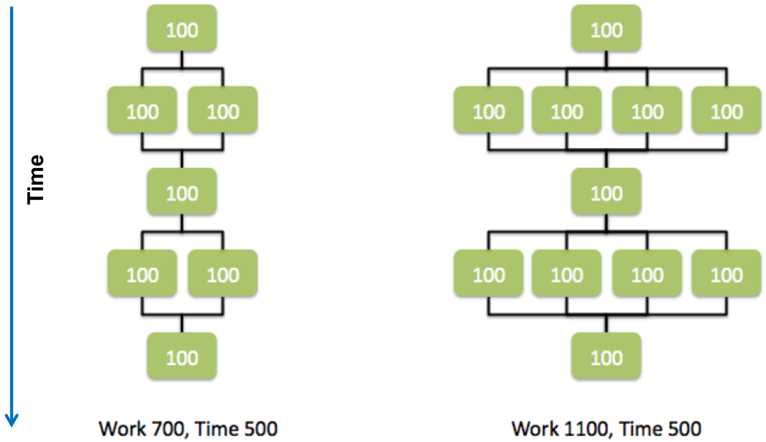
**n**: number of cores
**p**: parallel portion

*"... speedup should be measured by scaling the problem on the number of processors, not by fixing the problem size." - 1988*

# Amdahl vs Gustafson



# Increase the Problem Size!



Work 700, Time 500
Speedup 1.4x

Work 1100, Time 500
Speedup 2.2x

# Strong and Weak Scaling

**Case A**

- ○ A human is waiting in front of the terminal: **strong scaling**
- ○ A problem of a fixed size is processed by an increasing number of processors
- ○ Best modelled with **Amdahl's law**

**Case B**

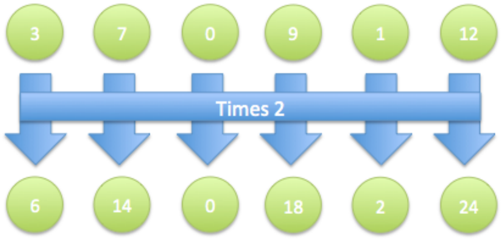
- ○ Want to get the most done in a certain amount of time: **weak scaling**
- ○ Every processor has a specified amount of work to do, and then when adding processors, we also add work
- ○ Best modelled with **Gustafson's law**

***Two sides of the same coin!***

# Data Parallelism

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

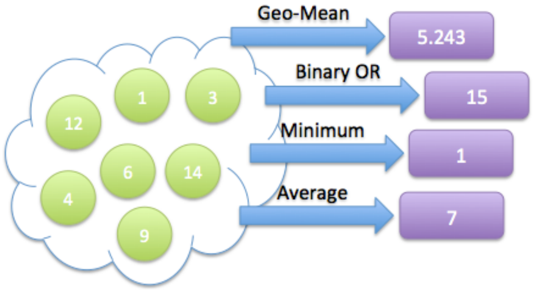
***Example of pure data parallelism***: multiplication of an array of values (ordinary administration for vector units and GPUs!)



# Task Parallelism

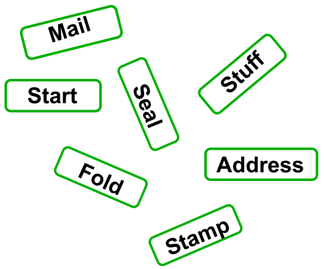
**Definition:** parallelism achieved through the partition of load in small work baskets consumed by a pool of resources.

***Example of pure task parallelism***: calculate mean, binary OR, minimum and average of a set of numbers



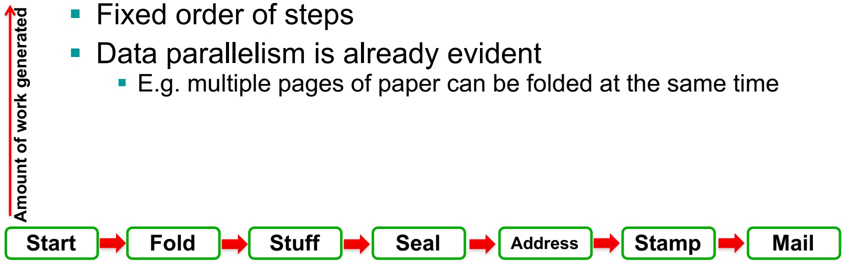
# Mixed Solutions

**Mandate:** Build an efficient letter sending system mixing data and task parallelism



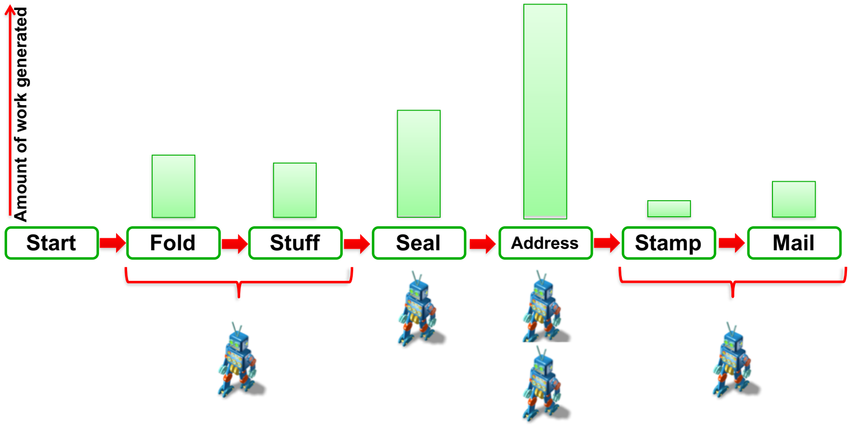
# Mixed Solutions

- For this case:
  - Fixed order of steps
  - Data parallelism is already evident
    - E.g. multiple pages of paper can be folded at the same time



# Mixed Solutions

- These operations require different amount of work though



# Finding Concurrency

What can be executed concurrently?

Some techniques to figure this out:

- ***Data decomposition***
  - The partition of the data domain
- ***Recursive decomposition***
  - Divide and conquer
- ***Functional decomposition***
  - Split according to program functions
- ***Task decomposition***
  - Split according to logical tasks

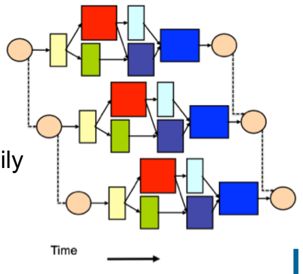
**DIVIDE**
**ET**
**IMPERA**

# Mixed Data and Task Parallelism

- Pure task/data parallelism is difficult to achieve in reality
  - Sometimes close enough to real use cases!
- **Mixing data and task parallelism** is the key
  - Many different algorithms applied to a **stream of data**
  - Items processed in **stages** where data parallelism is expressed
  - Many items can pass through the **pipeline** simultaneously
  - Think of items as “collision events” and algorithms as “HEP data processing units”!

# Rethinking the Parallel Framework

- **Need to change the problem size**
  - Process **multiple events concurrently**
  - Helps on tails of sequential processing
- **Contradicts a lot of the basic assumptions in existing code**
  - Code prepared to process only one event at a time in memory
  - But existing code can’t be thrown away easily
  - Need to localise distributed states
- **Major effort ongoing in all LHC experiments**
  - Exciting times for curious programmers!



# A Glimpse on Complications

1. **The DAG is not known to its entirety**
   - Hidden dependencies
2. **Shared states are rarely safe**
   - "Caches" that do not behave like… well… caches
3. **Algorithms are not thread-safe**
   - E.g. track reconstruction cannot be run on two events concurrently
   - Making all algorithms thread-safe is an impossible task
4. **External libraries are not thread safe**
   - But independent parts of the framework access them
   - Not all of the libraries will be thread safe ever!

# Solutions?

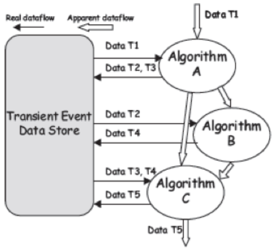
We need a **smart scheduling** environment

1. **The DAG has to be "fixed" by changing the existing code**
2. **Shared states are replaced by task-local data, avoid locks!**
   - More in the next lectures
3. **If an algorithm requires a non-thread safe resource, it has to 'reserve' it beforehand**
   - No two algorithms using the resource are scheduled at the same time

# Scheduling Directions

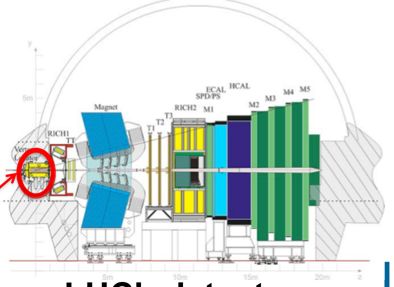
Three ways of coding up a scheduler for the DAG:

- ***On demand***
  Start with the last algorithms in the DAG and invoke whatever algorithm is needed on-the-fly.
  (*backward scheduling*)
- ***Data driven***
  Start with the first algorithms in the DAG and start new algorithms whenever the necessary inputs are there.
  (*forward scheduling*)
- ***Global view***
  Analyse the entire DAG and schedule algorithms according to the dependency order (*graph scheduling*)



# A Simplified Example

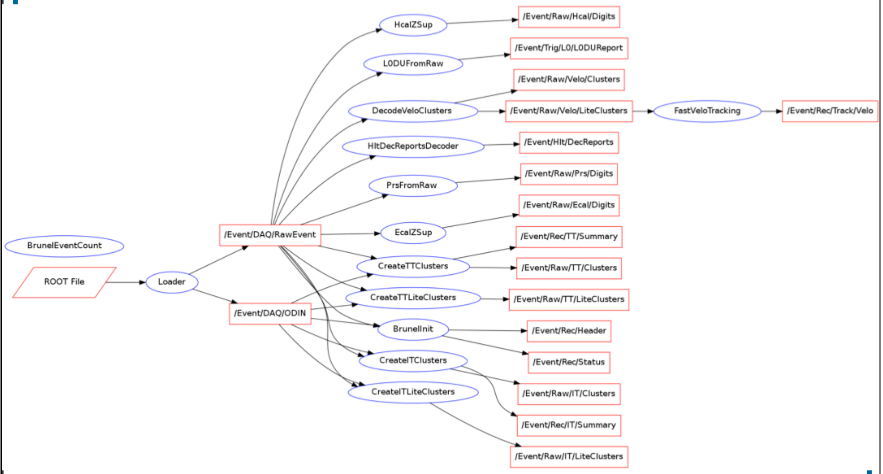
- **Such a parallel framework is not only theory**
- **They already exist for**
  - CMS offline software (CMSSW)
  - ATLAS/LHCb framework (Gaudi)
- **Let's have a look at an example workflow**
  - A slice of the LHCb reconstruction
  - Only the low level objects of the vertex locator (VELO)



**This part of the detector**

**LHCb detector**

## The Velo Low-Level Reco DAG



## Take-Away Messages

- **Dealing with parallelism is inevitable**
  - Software must exploit parallel hardware
  - But there are different levels of exposure to parallelism
- **High energy physics has a history of parallelisation**
  - However at a rather naïve level
  - The next steps require a harder approach
- **Parallelisation can be exploited in multiple ways**
  - Data parallelism and task parallelism
- **Amdahl's and Gustafson's laws** give a handle for scaling behaviour
- There is a clear strategy for parallelising HEP software
  - Use of a **task-based** approach

# Patterns for Parallel Software Development

## Outline of This Lecture

**The Goals:**

1) *Understand a few basic patterns of sequential algorithms*
2) *Know how to map these onto parallel concepts*
3) *Understand how these scale*

# What is a Pattern?

**Software design pattern**

General, **reusable** solution to a **commonly occurring problem** in a given context in software design

**Parallel pattern**

Recurring combination of **task distribution** and **data access** that solves a specific problem in parallel algorithm design

# Serial Control Flow Patterns

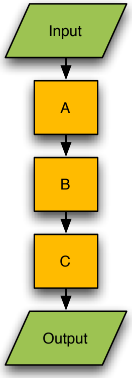
- Before starting with parallelism let’s look at what we know about the serial case
- We will have a look at the following ones:
  - **Sequence**
  - **Selection**
  - **Iteration**
- These are all simple concepts, but the vocabulary is important!

# Sequence

- A **sequence** is an ordered list of tasks/commands to be carried out in a given **order**
  - The exact dependencies of the commands do not matter
  - Side-effects do not matter
  - There is only **one task** executed at a time
  - The tasks are executed as defined

**Note that**

The compiler and the CPU may re-order instructions if they think it optimises runtime

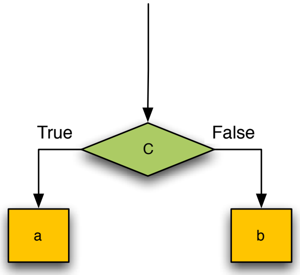


# Selection

- In a **selection**
  - The commands a and b **depend on decision** of c
  - **Always only one** of the two sides is being executed

**The «if» statement**

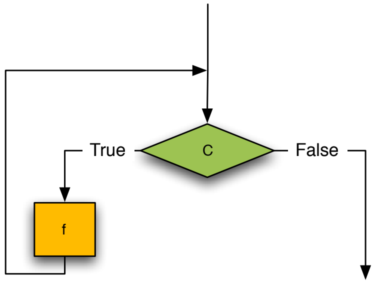
The CPU may apply speculative execution, but it always takes care of sanity



# Iteration

- In an **iteration** a certain function f is executed as long as a certain condition c is true.
  - This is the famous *while* loop

```
while ( c ) {
  f;
}
```



# Iteration II

- **How do condition and function depend on each other?**
  - There must be some dependency, otherwise it is an infinite loop
- Sometimes the dependency is trivial and can be re-formulated as a *for* loop (a.k.a. counted loop)

```
i = 0;
while ( i < n ) {
  f;
  ++i;
}
```

→

```
for (i = 0; i < n; ++i ) {
  f;
}
```

- In serial code this is mainly just syntactic sugar
  - However, it gives some nice hints to the compiler

# Iteration III

- The serial iteration pattern might seem trivially parallelisable but…
  - Beware of **dependencies**!
- **Do multiple iterations depend on each other?**
  - Loop-carried dependency
- Different kinds of dependencies translate to **different parallelisation possibilities**

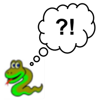
# Iteration IV

```
void doIt( int n, double x[], int a[], int b[], int c[] ) {

   for (int i = 0; i < n; ++i) {
      x[ a[i] ] = x[ a[i] ] * x[ b[i] ] * x[ c[i] ]
   }

}
```

- Any chance of parallelising this?
- What are the obstacles?
  - i.e. what are the dependencies?

# Modern Syntax: An Interlude

- C++ is ever improving with new standards (C++11, C++14, C++17, …)
- Two (not so) recent additions are:
  - `auto var = retrieveSomeObject();`
  - `for (auto& element : myCollection)`



- **`auto`**: do not specify the type, the compiler finds it out at compile time. Useful to avoid tedious typing also detrimental for readability of the code!
- **Range-based loops**: build a loop with a concise syntax!

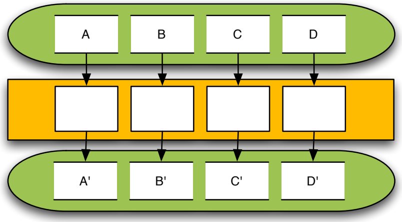
**Take advantage of this! ☺**

# Parallel Patterns

- After reminding ourselves about serial control patterns, let's have a look at a few parallel patterns
  - Can help you structure your parallel program
- The serial **iteration** pattern has many parallel offsprings
  - **Map**
  - **Partition**
  - **Reduce**
  - **Scan**
- Other useful patterns
  - **Pipeline**
  - **Superscalar Sequences**

# Map

- The **map** is the most trivial parallel extension of the serial iteration
  - Apply the same function `f` on multiple elements of a collection in parallel
  - We **hide the loop**!

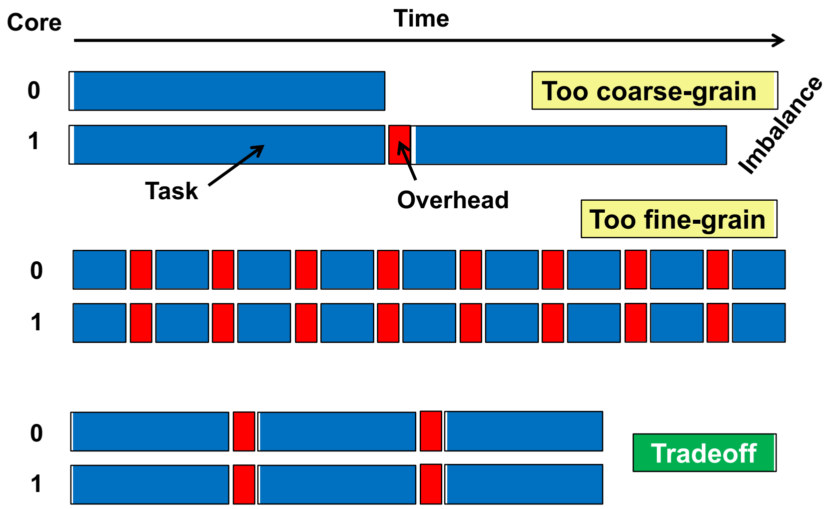


- **Requirements**:
  - No loop-carried dependency
  - Function `f` is pure, i.e. without side-effects
- **Scaling:** n (linear w.r.t. the number of elements in the collection)

# Partition

- The map pattern helps when parallelising on collections
- However, sometimes it is useful to treat multiple items together
  - E.g. for the combination of multithreading and vectorisation
  - Multi-level parallelism!
- **Partitioning** allows for a custom split of the collection into subcollections or *chunks*
- A variant of partitioning is called **geometric decomposition**
  - Update of a partition needs data from other partitions
  - Might require synchronisation

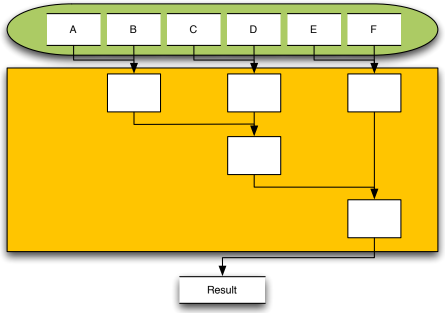
# Granularity



# Reduce

- A **reduction** combines the elements of a collection into a single result using a **combiner function**
- **Requirements**:
  - No loop-carried dependency apart from the combined result
  - Combiner function is **associative**
    - Be careful with floating point operations!
  - Having a commutative function is beneficial

# Reduce II



- **Speedup:** n / log n
- Counters are a typical example for reduction input
- Before coming to a real example, let's have a look at modern C++ again…

# Interlude – Lambdas

- Lambda expressions are anonymous functions and can be assigned to the `std::function` type
- They can be passed as parameters as if they were regular variables
- When defined, they can capture a specific set of variables ( or all )
- Once they have been defined, they can be passed to functions like `std::for_each` or TBB's `parallel_for`

```
std::function< double (double, double) > f =
   [ ] (double a, double b) { return a + b; };

std::cout << f ( 23.0, 24.0 );
```

# Interlude – Lambdas II

- Using the C++ `auto` keyword simplifies the syntax, but does not change the behavior

```
auto f = [ ] ( double a, double b ) { return a + b; };
```

- Capture the variable `globalOffset` as a reference and use it in the computation

```
auto f = [ &globalOffset ] ( double a, double b )
         { return a + b + globalOffset; };
```

- Capture all variables defined in the current scope by value

```
auto f = [ = ] ( double a, double b )
         { return a + b + globalOffset; };
```

- Can you think of the difference in behavior when using capture-by-value instead of capture-by-reference?

# Reduce III

- Libraries like Intel's Threading Building Blocks (TBB) provide already all ingredients for standard patterns like reduce:

```
int sum = tbb::parallel_reduce(
  // The input array, which will be partitioned automatically
  tbb::blocked_range<int*>(array, array + size),
  // Identity value for the sum reduction
  0,
  // Lambda that returns the sum of all elements in a partition
  [](const tbb::blocked_range<int*>& r, int v) {
    for (auto i = r.begin(); i != r.end(); ++i) v += *i;
      return v;
  },
  // Reduction operation that combines the per-partition sums
  [](int x, int y) { return x+y; }
);
```

# Map and Reduce Combined

- Usually **map** and **reduce** go hand in hand:
  - A **function** being applied to single elements
  - The results are then passed to a **combiner function**
- A concrete example:
  - Count the number of times a certain word appears in a text
- Solution:
  - **Partition**: Split the text in equally-sized chunks
  - **Map:** Do the word count
  - **Reduce:** Add the counts
- Various map/reduce frameworks at your disposal!

# The Power of Map-Reduce

- The combination of the Map and Reduce patterns has been extremely successful in **massive distributed data processing**
- A little bit of history…
  - 2004: Google publishes the MapReduce paper
  - 2006: Hadoop is released, inspired by MR
- Nowadays, MR is **behind every click** on popular web sites or services
  - Facebook, Twitter, Yahoo, …
  - Analytics to predict user interests, target ads, show recommendations, … and many more
  - Robust, fault tolerant
  - Scale to crunch large datasets

# Map-Reduce and Functional Chains

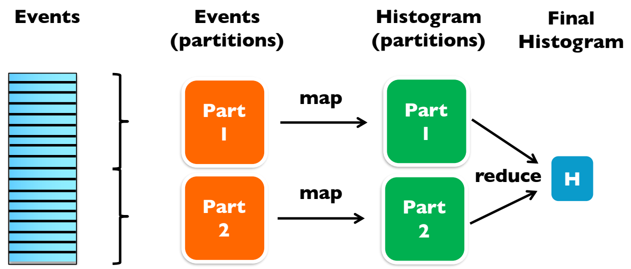
- Map and reduce were born in **functional programming**
  - Declare **what** you want to do, not how
  - No side-effects
- **High-level view**, based on two main concepts:
  - Data is organised in **collections** of elements
  - We apply functions to those elements, possibly in a chain

```
histo = events.map(fillHist).reduce(mergeHist)
```

- Implemented by frameworks like Spark
  - No need to manage parallelisation, just **think about opportunities for parallelism**!

# Map-Reduce and Functional Chains II

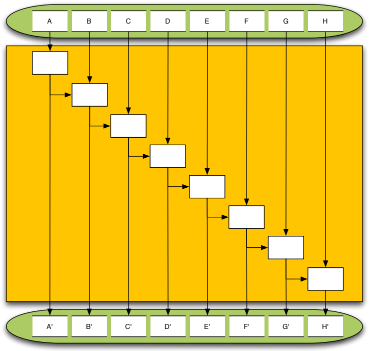
- Implementation responsible for producing a **parallel execution plan**
  - Where are the data?
  - What resources are available?
  - What optimisations can be applied?



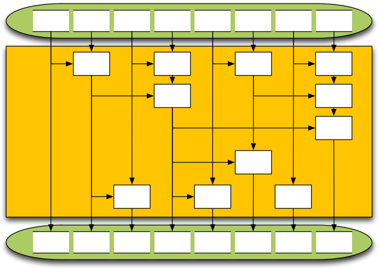
# Scan

- **Scan** is another offspring of the iteration pattern with more relaxed boundary conditions
- **Requirements**:
  - Result of element `n` depends on `n-1`
  - Successor function is **associative**

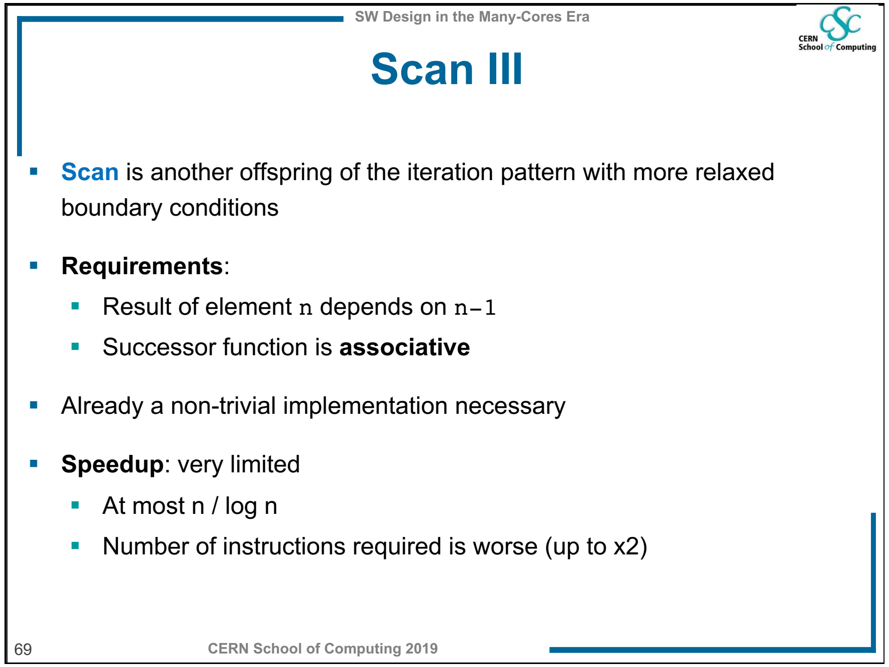
# Scan II



Serial version

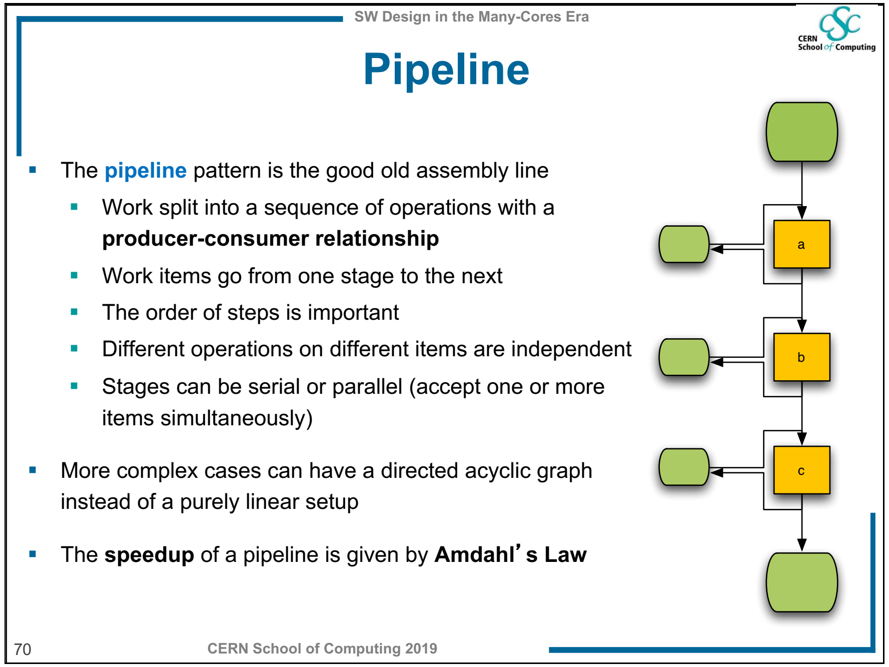


Parallel version



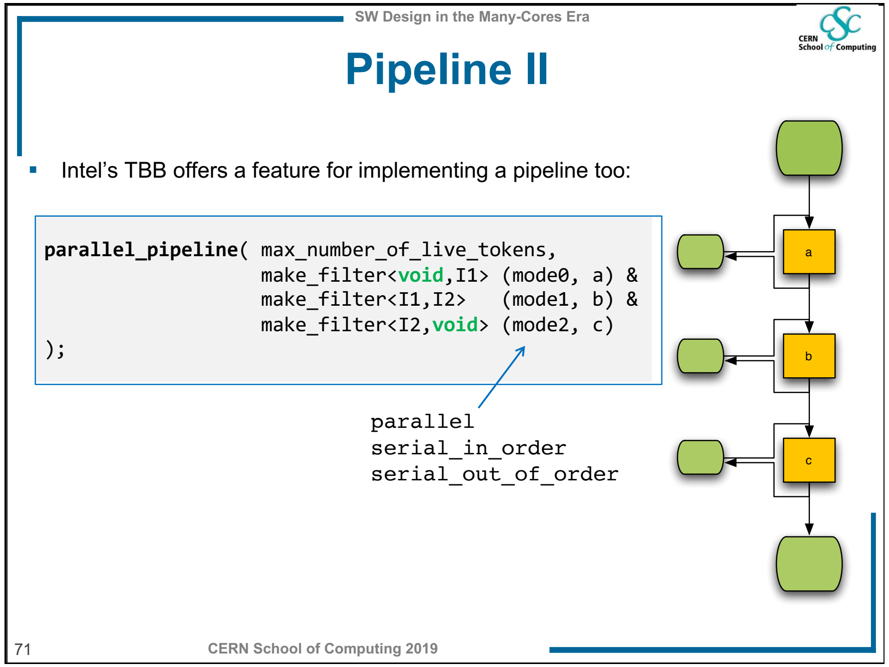
# Scan III

- **Scan** is another offspring of the iteration pattern with more relaxed boundary conditions
- **Requirements**:
  - Result of element `n` depends on `n-1`
  - Successor function is **associative**
- Already a non-trivial implementation necessary
- **Speedup**: very limited
  - At most n / log n
  - Number of instructions required is worse (up to x2)



# Pipeline

- The **pipeline** pattern is the good old assembly line
  - Work split into a sequence of operations with a **producer-consumer relationship**
  - Work items go from one stage to the next
  - The order of steps is important
  - Different operations on different items are independent
  - Stages can be serial or parallel (accept one or more items simultaneously)
- More complex cases can have a directed acyclic graph instead of a purely linear setup
- The **speedup** of a pipeline is given by **Amdahl's Law**

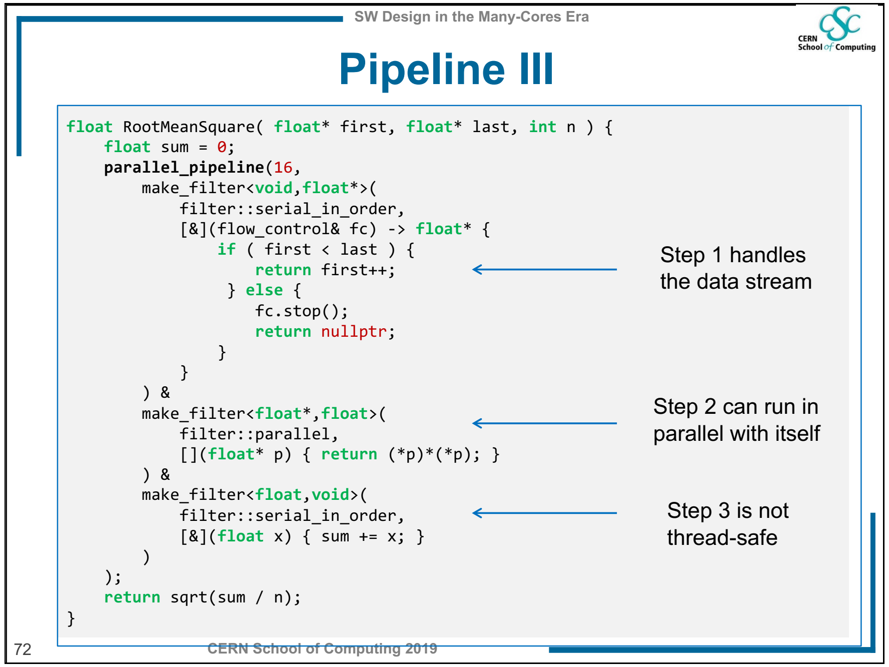


# Pipeline II

- Intel's TBB offers a feature for implementing a pipeline too:

```
parallel_pipeline( max_number_of_live_tokens,
                   make_filter<void,I1> (mode0, a) &
                   make_filter<I1,I2>   (mode1, b) &
                   make_filter<I2,void> (mode2, c)
);
```

```
parallel
serial_in_order
serial_out_of_order
```



# Pipeline III

```
float RootMeanSquare( float* first, float* last, int n ) {
    float sum = 0;
    parallel_pipeline(16,
        make_filter<void,float*>(
            filter::serial_in_order,
            [&](flow_control& fc) -> float* {
                if ( first < last ) {
                    return first++;
                 } else {
                    fc.stop();
                    return nullptr;
                }
            }
        ) &
        make_filter<float*,float>(
            filter::parallel,
            [](float* p) { return (*p)*(*p); }
        ) &
        make_filter<float,void>(
            filter::serial_in_order,
            [&](float x) { sum += x; }
        )
    );
    return sqrt(sum / n);
}
```

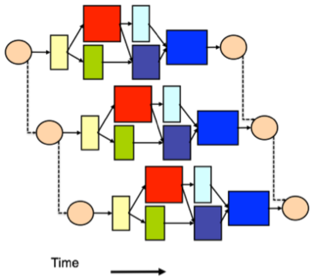
Step 1 handles the data stream

Step 2 can run in parallel with itself

Step 3 is not thread-safe

# Superscalar Sequences

- Split work into a number of tasks and define their data dependencies
- Let a task scheduler do the rest
- **Pattern followed by concurrent HEP data processing frameworks**



- Assumption of this model is that there are **no hidden data dependencies** and **no side-effects** unknown to the scheduler
  - Let's have a look at these assumptions…

# Hidden Data Dependencies

```
std::atomic_bool doit(false);

void task1() {
  …
  if (doit)) {
    eventstore.put(fancystuff);
  }
}

void task2() {
  doit = true;
}
```

Thread-safe boolean variable

- Content of the event store depends on the execution order
- Thread-safe objects don't help at all
- It is a pure logic flaw

# Side Effects

- Triggered when a computation modifies some **shared state** outside of its local environment
  - e.g. a global variable
- They are a major obstacle for parallelism
  - Watch out for them when applying your parallel patterns!
- In general every non thread-safe resource is an issue
- Remember from previous lectures:
  - Side-effect free resources are the ideal solution
  - If not possible, tell the scheduler about what you need and **"reserve"** what is unsafe

# Take-Away Messages

- There exist design patterns to help you parallelising your programs
  - Check if you can **reuse** them!
- They all have their origin in serial patterns, but add constraints to the operations allowed
- **Map-Reduce** is a very successful pattern, used every day for distributed processing of large amounts of data
- High-level features like C++ lambdas, the TBB library or the Spark framework make it easier for you to get started with these patterns

# Base Concepts of Parallel Programming: A Pragmatic Approach

**Thanks to Danilo Piparo for preparing these slides and lecturing them in the previous years!**

## Outline of This Lecture

**The Goals:**

*1) Become familiar with the basic concepts of parallel programming through the discussion of concrete examples in C++*

*2) Know what is behind the scenes of a task based approach*

*3) Be able to start developing parallel applications.*

- Concurrency: asynchronous execution and threads
- Synchronisation: design principles, replication, atomics, transactions and locks

## C++: A Reminder

- The approach of this lecture is **pragmatic.**
  - “Forward declarations” to concepts treated later will be used!
  - Concepts are illustrated through **concrete examples** involving C++ constructs.
- **C++ is the programming language of HEP** for frameworks, event generators, simulation toolkits, analysis and reconstruction applications (number crunching code!)
  - Python is also widespread for configuration, analysis and scripting
- C++: “The power, elegance and simplicity of a hand grenade”

## Object Orientation

C++ allows OO programming.

Now, are objects good?

**Almost copied from Tony Albrecht: “Pitfalls of Object Oriented Programming”**

# Object Orientation

C++ allows OO programming.

Now, are objects good?

All of the HEP code moved from FORTRAN to C++ in the early 90s

- Well, yes

Almost copied from Tony Albrecht: "Pitfalls of Object Oriented Programming"

# Object Orientation

C++ allows OO programming.

Now, are objects good?

All of the HEP code moved from FORTRAN to C++ in the early 90s

- Well, yes
- And no

Almost copied from Tony Albrecht: "Pitfalls of Object Oriented Programming"

# Object Orientation

C++ allows OO programming.

Now, are objects good?

All of the HEP code moved from FORTRAN to C++ in the early 90s

- Well, yes
- And no

Almost copied from Tony Albrecht: "Pitfalls of Object Oriented Programming"

# Object Orientation

C++ allows OO programming.

Now, are objects good?

All of the HEP code moved from FORTRAN to C++ in the early 90s

- Well, yes
- And no

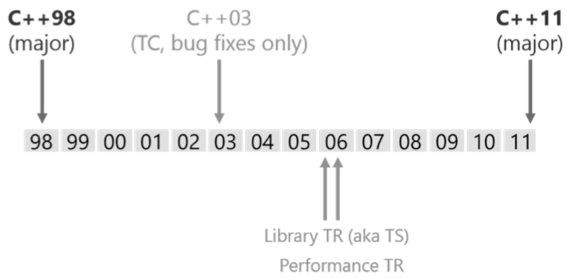
Keyword: Data Oriented Design (re-design?)

Almost copied from Tony Albrecht: "Pitfalls of Object Oriented Programming"

# C++ Evolves!

- A committee reviews the C++ standard
  - CERN is part of it!

**Extensive support for concurrency!**

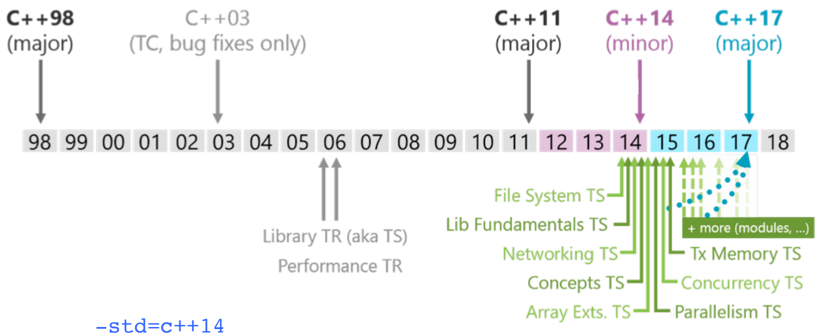


**Widely supported by compilers since some time, e.g.:**
- **GCC >= 4.8**
- **Clang >= 3.4**

**Commercial compilers somewhat lagging behind novelty**

# C++ Evolves!

- A committee reviews the C++ standard
  - CERN is part of it!



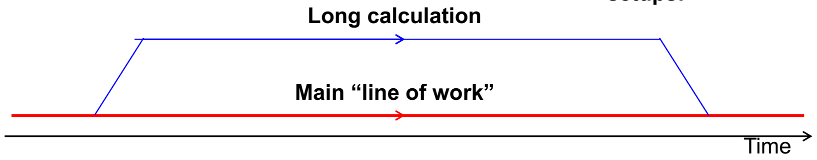
`-std=c++14`
switch to activate it

# Concurrency

# 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?**

**Among the simplest asynchronous setups!**



# 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

# 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;
  }
```

# std::async in Action

Header for async and future

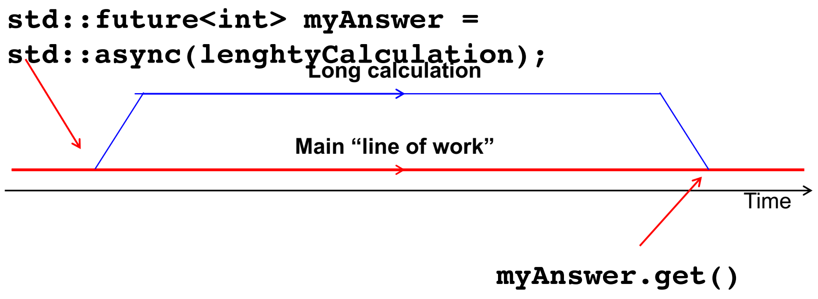
“Launch” the calculation

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;
  }
```

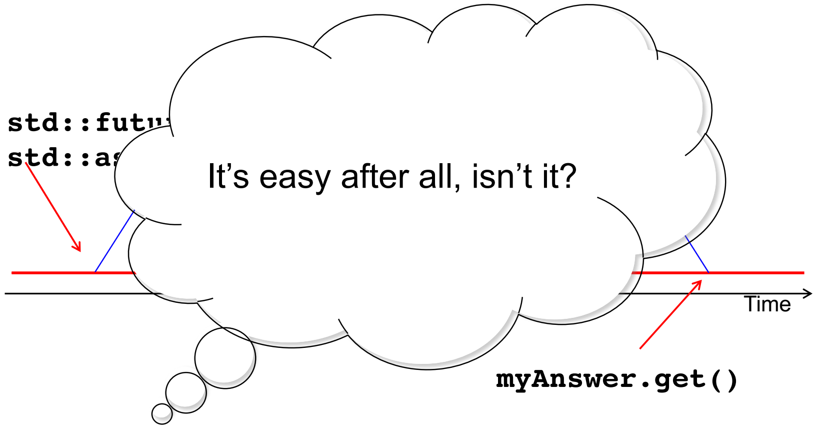
- `std::async` can have a second parameter, the “policy”:
  - `std::launch::async`: execute function in a new separate thread
  - `std::launch::deferred`: defer call until `get()` is called (lazy)
  - Default: “`async` or `deferred`”, the implementation chooses!

# std::async in Action



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

# std::async in Action



```
std::futu
std::as
```

It's easy after all, isn't it?

```
myAnswer.get()
```

# Well, to be Honest

- 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

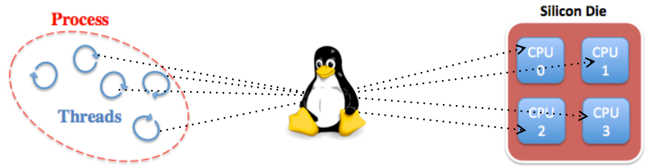
Need to think parallel
- Evolve the existing systems
- Be disruptive and think to the future

Unity of opposites ☺

# Threads

# Let' s switch gears: Threads

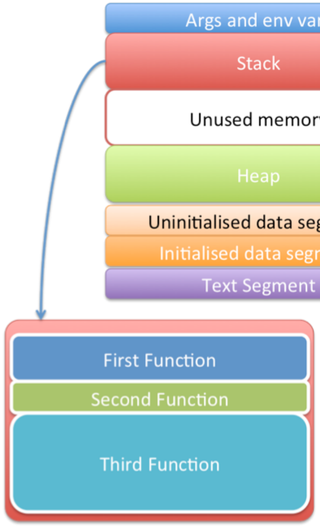
- From the operating system point of view:
  - **Process**: isolated instance of a program, with its own space in (virtual) memory, can have multiple threads
  - ★ **Thread**: light-weight process within a process, **sharing the memory** with the other threads living in the same process
- The kernel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

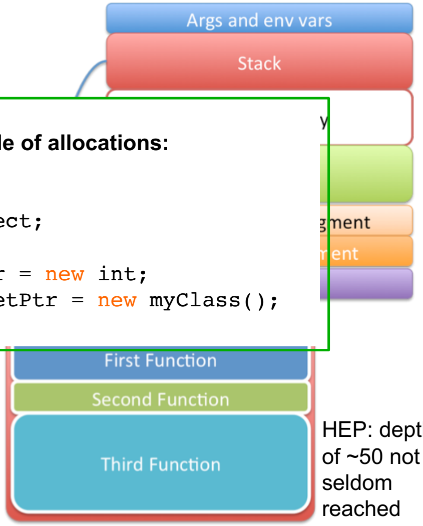
- **Text Segment**: code to be executed.
- **Initialized Data Segment**: global variables initialized by the programmer.
- **Uninitialized Data Segment**: This segment contains uninitialized global variables.
- **The stack**: The stack is a collection of stack frames. It grows whenever a new function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



**Example of allocations:**

- **On the stack:**

```
int a=12;
myClass myObject;
```

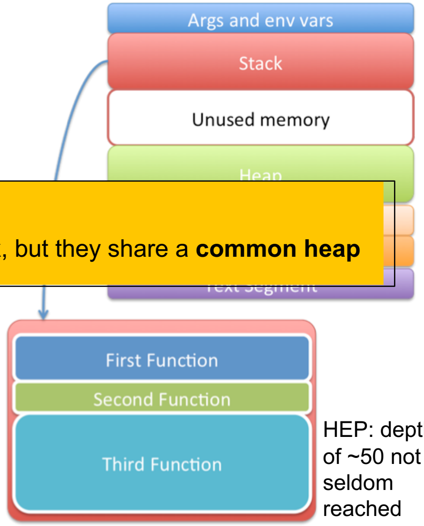
- **On the heap:**

```
int* a_pointer = new int;
myClass myObjetPtr = new myClass();
```

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
- **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`").



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

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

# 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!

# Threads example

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

void f(){std::cout << “Hello Concurrent World!\n”; }

int main(){
  std::thread t(f);
  t.join();
  }
```

# Threads example

Header for `std::thread`

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

void f(){std::cout << “Hello Concurrent World!\n”; }

int main(){
  std::thread t(f);
  t.join();
  }
```

Create and start a thread

**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…

# A Pitfall with Threads

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

void f(const std::string& s){std::cout << s; }

void g(){
  std::string s(“Hello\n“);
  std::thread t(f,s);
  t.detach();
  }
```

# A Pitfall with Threads

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

void f(const std::string& s){std::cout << s; }

void g(){
  std::string s("Hello\n");
  std::thread t(f,s);
  t.detach();
  }
```

Passed by reference

String s lives in the scope of function g

Parallel programs: variables' lifetime even more important than in sequential 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 <iostream>

void f(const std::string s){std::cout << s; }

void g(){
  std::string s("Hello\n");
  std::thread t(f,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!

# A First Abstraction

**A possible prototype backend behind task oriented programming!**

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

std::mutex myMutex;
void printThreadID(int i){
 std::lock_guard<std::mutex> myLock(myMutex);
 std::cout << "thread num " << i << " - id "
           << std::this_thread::get_id() << std::endl;
 };

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();
}
```

# A First Abstraction

**A possible prototype backend behind task oriented programming!**

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

std::mutex myMutex;
void printThreadID(int i){
 std::lock_guard<std::mutex> myLock(myMutex);
 std::cout << "thread num " << i << " - id "
           << std::this_thread::get_id() << std::endl;
 };

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();
}
```

Be patient for the moment! ☺

Identify the thread

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

Limitation: cannot retrieve the return value.

# A First Abstraction

**A possible prototype backend**

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

std::mut
void pri
 std::lc
 std::cc

 };

int mair
 std::ve
 for (ir
    myThreads.emplace_back(printThreadID,i);

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

ng!

```
-> g++ –std=c++14 –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
```

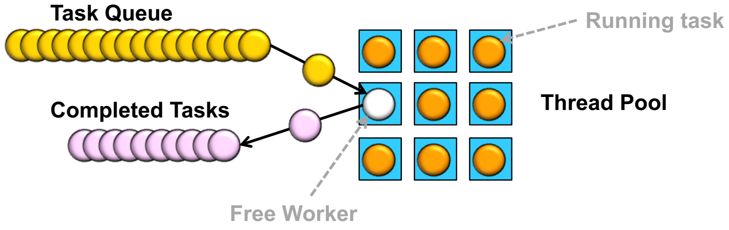
When dealing with concurrency, asynchronous events are daily business!

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

Limitation: cannot retrieve the return value.

# The Thread Pool Model

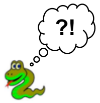
- Thread pool: ensemble of worker threads which are …
- Initialised once, consuming work from …
- .. A work queue …
- .. to which elements of work (tasks) can be added



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

# Modern Syntax: An Interlude

- A nice byproduct of the previous examples - three C++ constructs:
  - `std::vector<T>::emplace_back(T&&)`
  - `auto`
  - `for (auto& element : myCollection)`



- **`emplace_back`**: do not construct and then copy/move back in the vector (push_back) but construct *in place*. One copy less!
- **`auto`**: do not specify the type, the compiler finds it out at compile time. Useful to avoid tedious typing also detrimental for readability of the code!
- **Range based loops**: build a loop with a concise syntax!

**A modern approach to scientific computation cannot avoid the usage of the most modern tools!**

# Synchronisation: Good Design, Replication, Atomics, Transactions and Locks

# The Problem

- Fastest way to share data: access the same shared memory
  - 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)

# 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.

# 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

# Why so many strategies?

- Design, replication, atomics, transactions and locks !
- There is no silver bullet to solve the issue of “resources protection”
  - Complex problematic
- Case by case investigation needed
  - Better to be aware of several strategies
- Best solution: often a tradeoff
  - The lightest in the serial case?
  - The lightest in presence of high contention?

# What is not Thread Safe?

**Everything, unless explicitly stated!**

In four words: **Shared State 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

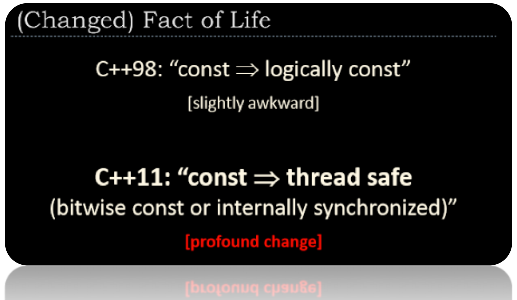
It sounds depressing. But there are several ways to protect thread unsafe resources!

# Const Means Thread Safe

More a “new convention” rather than a technique.

• True for the STL and all at least C++11 compliant code.

**“I do point out that const means immutable and absence of race conditions[…]”** B. Stroustrup



(Changed) Fact of Life

C++98: “const ⇒ logically const”

[slightly awkward]

**C++11: “const ⇒ thread safe**

(bitwise const or internally synchronized)”

[profound change]

From H. Sutter, “You don't know const and mutable”

# 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**

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

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

- Stl algorithms: map an operation to a list of values.
- Decompose operations in functions, percolate the information through their arguments

Even without becoming purists, functional programming principles can avoid lots of headaches typical of parallel programming

# 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`

**`Example:`**
**`boost::thread_specific_ptr`**

- Analogies with multi-process approach but
  - Does not rely on kernel features (copy-on-write)
  - Can have high granularity
- 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

# 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

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

# 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(IncrAn
 auto t2 = std::thread(IncrAn
 IncrAndPrint("main",0);
 t1.join(); t2.join();
}
```

One private copy per thread will exist

**Possible output:**

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

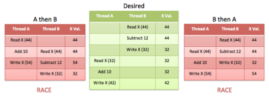
**Possible output w/o tls (not correct!):**

```
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)

# Atomic Operations

- Building block of thread safety: **an atomic operation is an operation seen as non-splittable by other threads**
  - Other real life examples: 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*)
- Well behaved with:
  - boolean, integer types. E.g. `std::atomic<unsigned long>`
  - Pointer to any type. E.g. `std::atomic<MyClass*>`

# Atomic Counter

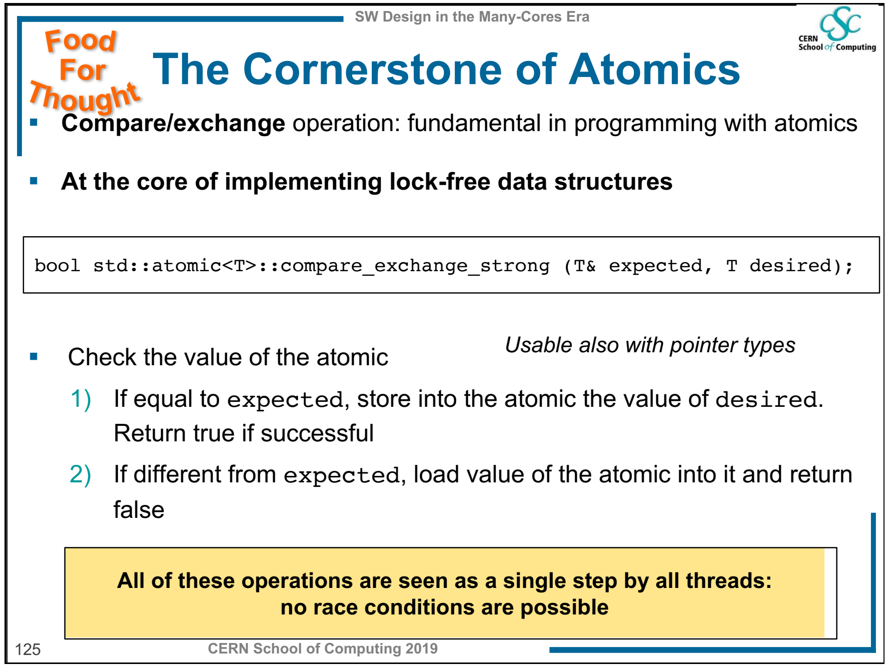
```
#include <atomic> …

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!**



Food For Thought

## The Cornerstone of Atomics

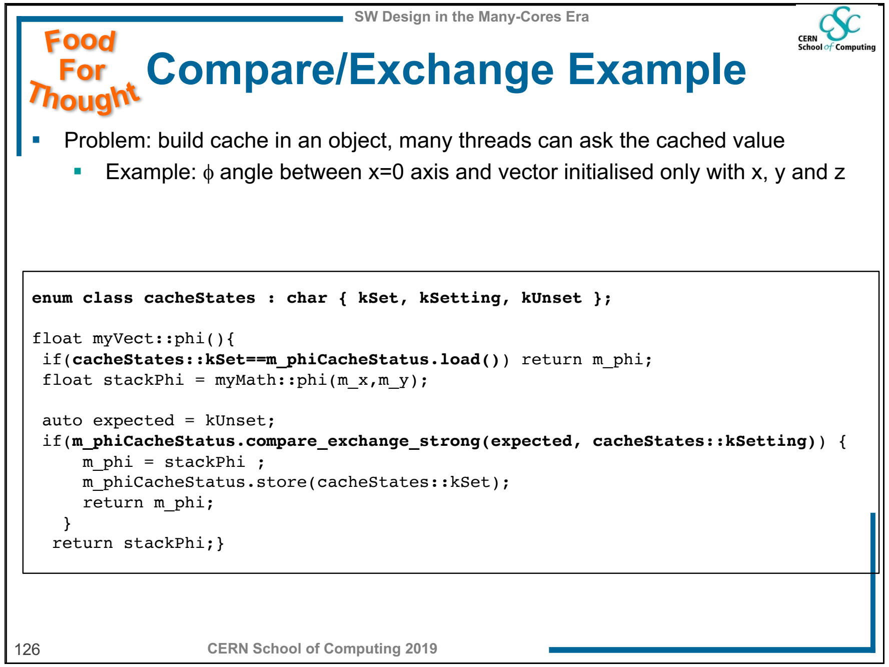
- **Compare/exchange** operation: fundamental in programming with atomics
- **At the core of implementing lock-free data structures**

```
bool std::atomic<T>::compare_exchange_strong (T& expected, T desired);
```

*Usable also with pointer types*

- Check the value of the atomic
  1) If equal to `expected`, store into the atomic the value of `desired`. Return true if successful
  2) If different from `expected`, load value of the atomic into it and return false

**All of these operations are seen as a single step by all threads: no race conditions are possible**

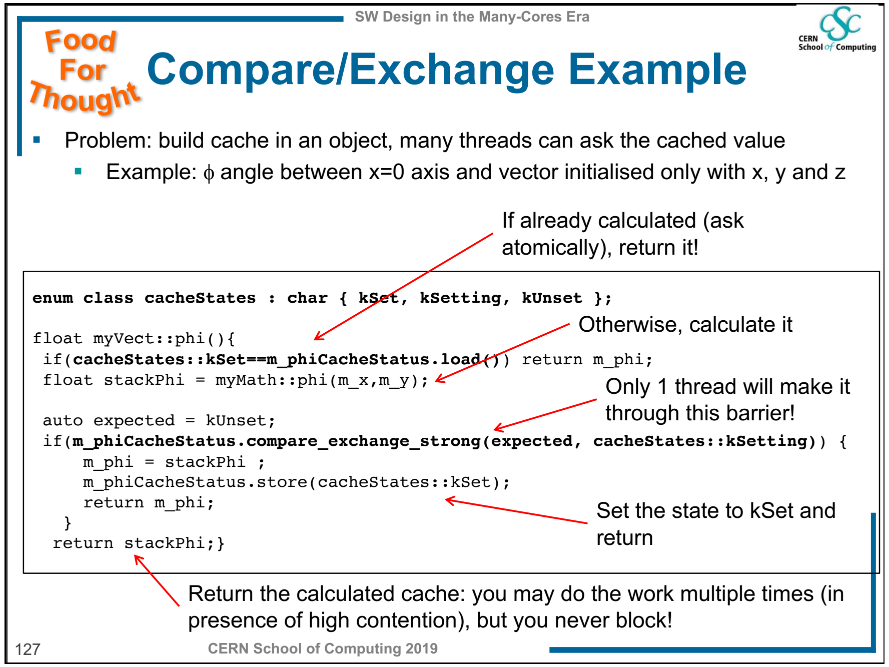


Food For Thought

## Compare/Exchange Example

- Problem: build cache in an object, many threads can ask the cached value
  - Example: ϕ angle between x=0 axis and vector initialised only with x, y and z

```
enum class cacheStates : char { kSet, kSetting, kUnset };

float myVect::phi(){
 if(cacheStates::kSet==m_phiCacheStatus.load()) return m_phi;
 float stackPhi = myMath::phi(m_x,m_y);

 auto expected = kUnset;
 if(m_phiCacheStatus.compare_exchange_strong(expected, cacheStates::kSetting)) {
     m_phi = stackPhi ;
     m_phiCacheStatus.store(cacheStates::kSet);
     return m_phi;
   }
  return stackPhi;}
```



Food For Thought

## Compare/Exchange Example

- Problem: build cache in an object, many threads can ask the cached value
  - Example: ϕ angle between x=0 axis and vector initialised only with x, y and z

If already calculated (ask atomically), return it!

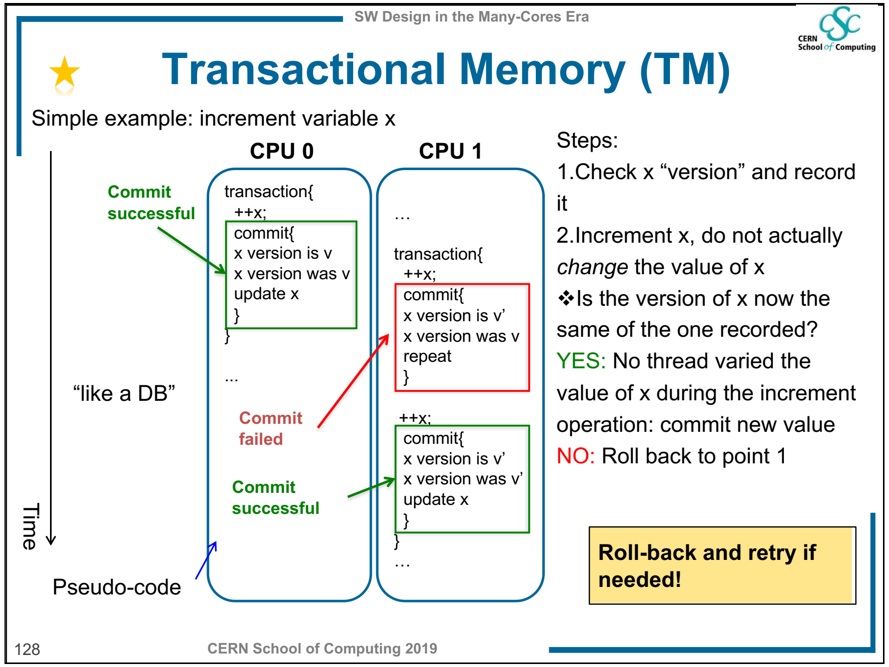
```
enum class cacheStates : char { kSet, kSetting, kUnset };

float myVect::phi(){
 if(cacheStates::kSet==m_phiCacheStatus.load()) return m_phi;
 float stackPhi = myMath::phi(m_x,m_y);

 auto expected = kUnset;
 if(m_phiCacheStatus.compare_exchange_strong(expected, cacheStates::kSetting)) {
     m_phi = stackPhi ;
     m_phiCacheStatus.store(cacheStates::kSet);
     return m_phi;
   }
  return stackPhi;}
```

Otherwise, calculate it

Only 1 thread will make it through this barrier!

Set the state to kSet and return

Return the calculated cache: you may do the work multiple times (in presence of high contention), but you never block!



## ★ Transactional Memory (TM)

Simple example: increment variable x

CPU 0:

```
transaction{
 ++x;
 commit{
 x version is v
 x version was v
 update x
 }
}

...
```

CPU 1:

```
...

transaction{
 ++x;
 commit{
 x version is v'
 x version was v
 repeat
 }

 ++x;
 commit{
 x version is v'
 x version was v'
 update x
 }
}
...
```

Steps:

1. Check x "version" and record it
2. Increment x, do not actually *change* the value of x

❖Is the version of x now the same of the one recorded?

YES: No thread varied the value of x during the increment operation: commit new value

NO: Roll back to point 1

**Roll-back and retry if needed!**

# Transactional Memory (TM)

- Software Transactional Memory (STM) already widely available
- STM can be slow – Useful tool to learn how to protect resources!
- TM supported by modern CPUs (e.g. Intel Haswell – TSX extensions)

- Concept not part of C++ (yet?), but supported at least by GCC compilers. Two types of transactions:
- ***__transaction_atomic***: isolated, may contain only safe code (i.e. possible to roll back, no I/O, no volatile variables…)
- ***__transaction_relaxed***: isolated only from other transactions. Used where atomic transaction is not suited (e.g. IO, volatile, atomic memory access).

Atomic transactions can be nested into atomic and relaxed transactions, relaxed transactions only in relaxed transactions.

**We will focus on atomic transactions only!**

# An STM Concurrent Stack

```
[ … rest of the implementation … ]
bool stmQueue::push(const float f){
  bool statusCode = false;
    __transaction_atomic {
    if (m_size<m_MaxStackSize){
      m_v[m_size]=f;
      m_size++;
      statusCode=true;
    }
  }
  return statusCode;
}
bool stmQueue::pop(float& f){
  bool statusCode = false;
  __transaction_atomic {
    if (m_size>0){
      m_size--;
      f=m_v[m_size];
      statusCode=true;
    }
  }
  return statusCode;
}
[ … rest of the implementation … ]
```

One single, non splittable block of operations.

Transactions are a powerful way to implement synchronisation

- Code simple to understand and maintain
- Lock pathologies automatically avoided

# 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:

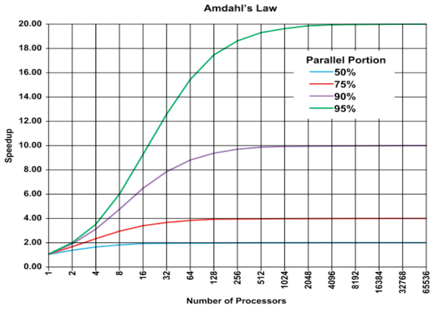
- Before the section, the thread is said to *acquire a lock on a mutex*
- After that, no other thread can acquire the lock
- After the section, the thread is said to *release the lock*

# 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***

# 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

# 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!)

# Scoped Locks: the Proper Way

Food For Thought

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 should be used in modern and performant C++

# 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.

# A deadlock

# 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

# Take Away Messages

**Concurrency:**

- Know the internals behind a task based approach
  - Threads and shared memory
- Asynchronous execution and non-determinism permeate concurrent applications:
  - Paradigm shift needed to understand and design parallel software solutions

**Synchronisation:**

- Try not to be obliged to synchronise: choose the right design
- Choose atomic types and memory transactions whenever possible
  - Atomic types supported by C++
- Locks are the last resort:
  - Reduce the critical sections to the bare minimum
  - Hold locks for the smallest time possible

# Understanding, Debugging and Profiling a Complex Multithreaded Application

# Outline of This Lecture

**The Goals:**

- Understand the relation between performance and correctness
- Master the strategies to be able to **analyse**, **debug** and **profile** a complex parallel application

**Three logical steps**

*Before running the application:*

1) Elements of static code analysis: Clang

*If something goes wrong:*

2) Understanding and debugging a multithreaded application with GDB

*Now that it works, how fast is it?*

3) Elements of performance measurement: igprof

# Performance and Correctness

- **Correctness comes first**: if your program is buggy, unreliable, unpredictable, no performance consideration makes sense (at all)
- **Performance is then crucial**: algorithms translate to real machine code, running on real hardware with its own features (CPUs, memory hierarchy, accelerators)

**A high quality test suite must be part of every software tool**

# Performance and Correctness

- **Correctness and performance: tightly correlated**
- Correctness checked quickly and extensively → runtime/memory improvements validated more easily
  - **Be in condition to label "changes"** in the final results as "acceptable", "expected" or "in the wrong direction"
  - Pandora's box**: what is the "right" result?** The one we had before? The new one? The "reference" one? **Not trivial at all!**
  - Use a grain of salt, **be in control of what happens!**

# Features of A Good Testsuite

- It's easy to run
  - One single command runs all tests
  - Tests can be selected, e.g. with regular expressions
- It's automatically ran
  - N times per day
  - Continuously check new code committed by developers
- Results are easy to interpret
  - E.g. Published on the web
  - Easy to track down problem, e.g. "test # 1206 failed with this output"

# Testing and parallel execution

- Test: minimal program aiming to stress a particular feature of the code
- Parallel code: no predictable order of operations possible
- The “same” test, execution pattern can be “different”
- Solution: properly designed tests
  - E.g. Maximising contention to “challenge” stability of the software

# Reproducibility

- E.g. two subsequent runs of the program produce the same histograms, identical bin by bin
- Simple for small setups
- Can be tricky with 5M lines, ~100 shared libraries
- Performance optimisations can lead to variations in final result (e.g. migration of entries to neighbouring bins)
  - Fundamental to remove all sorts of “noise”
- Non reproducibility in the sequential case: absence of control on the system
  - E.g. uninitialised variables, sloppy seeding of random generations, bogus memory access

# Attitude Towards Testing

- Aim to test-driven development: write tests before code
  - Test features individually one by one
- For each bug reported/found: create a reproducing test, add it to the suite, fix it.
  - If it’s not reproduced the bug does not exist!
- Don’t live with broken windows: follow up each failure
  - Assume it always points to a serious problem
- Time invested in writing tests is strategic
  - It always rewards

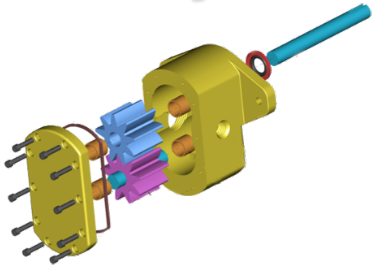
**If a software tool or one of its functionalities is not tested always assume it does not work**

# The Broken Windows Theory

“[…] Consider a building with a few broken windows. If the windows are not repaired, the tendency is for vandals to break a few more windows. Eventually, they may even break into the building, and if it's unoccupied, perhaps become squatters or light fires inside.”

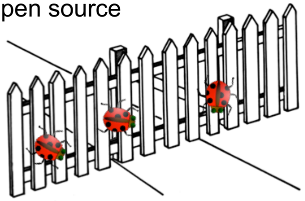
Wilson, James Q. "Broken windows: The police and neighborhood safety James Q. Wilson and George L. Kelling." Criminological perspectives: essential readings 400 (2003).

# Elements of Static Code Analysis



## Static Code Analysis

- Idea: embed static analysis in testing suites
- The procedure of **analysing the source code *before* compiling and running to automatically find bugs**
  - Rise (yet other) fences to protect against mistakes and bugs
  - Easily pluggable in big projects' build infrastructures
  - E.g. code blocks never executed because of faulty logic in if statements, *thread unsafe constructs*, etc.
- Several tools available, commercial and open source
  - Reference on the market: Coverity
  - Open source: Clang Static Analyzer



## LLVM and Clang

**LLVM**

- Free and open source
- A compiler infrastructure
- Frontend [C++,C,...] → Optimizer → Backend [x86, CUDA, ...]
- http://llvm.org

**Clang**

- LLVM frontend for C,C++ and Objective-C
- A possible alternative to GCC in some respects
- A lot of users - e.g. Apple, Intel (OpenCL)
- http://clang.llvm.org

Very powerful technology: e.g. C++ interpreter built on LLVM & Clang, **Cling**

http://root.cern.ch/drupal/content/cling

## Clang Static Analyser

**The static analyser is part of the clang frontend**

It offers the possibility to examine the program code on two levels:

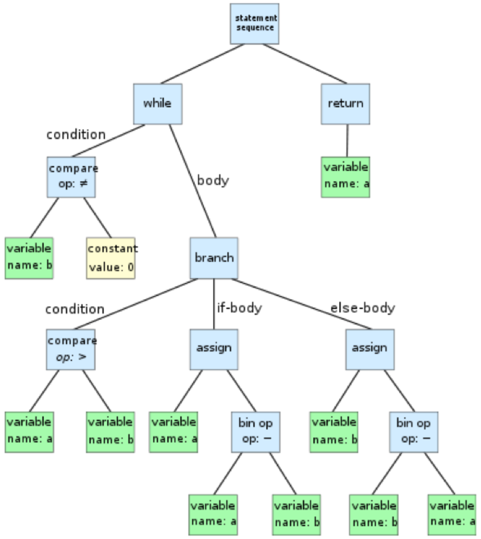
- Analysis of the Abstract Syntax Tree ( AST )

**http://clang-analyzer.llvm.org/**

- Symbolic Execution:
  - Every possible path through the program is explored and validated
- **A battery of checks already included**: uninitilialised access, dead stores, dereferencing null, invalid malloc calls, ...
  - User-defined can be added
- HTML report created automatically, detailed annotations of the source
- `scanbuild` tool: automatically replace the calls to the compiler in a makefile

To fire static analysis: `scan-build make`

# An AST



statement sequence
while
return
condition
body
compare op: ≠
variable name: a
variable name: b
constant value: 0
branch
condition
if-body
else-body
compare op: >
assign
assign
variable name: a
variable name: b
variable name: a
bin op op: −
variable name: b
bin op op: −
variable name: a
variable name: b
variable name: b
variable name: a

# Analysis for Thread Unsafety

Clang Static Analyzer: Custom checks can be added in form of a plugin written in C++

Checks for thread unsafety were developed by the LHC experiments

- Used in production for Q/A of experiments software
- **Useful in general!**

Some examples:

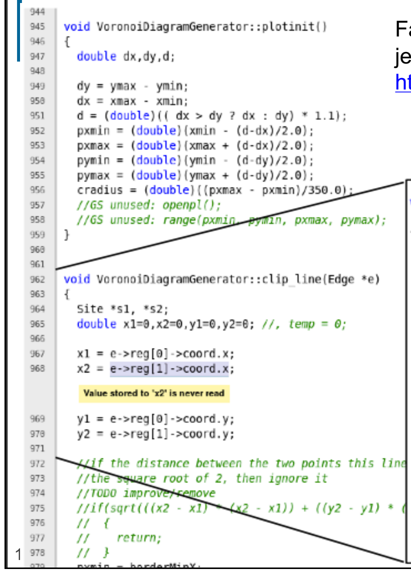
- Non const global/local statics
- Use of mutable keyword
- Use of const_cast to remove constness
- Other removals of constness (e.g. explicit cast)
- …

**Note the importance of const correctness**

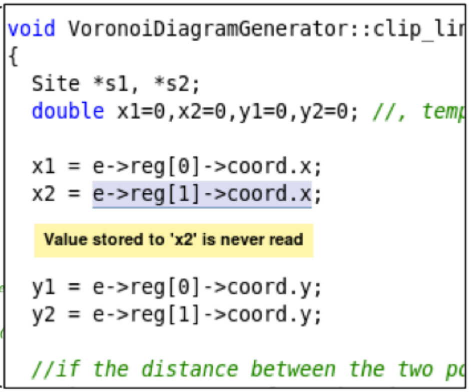
# Web Reports: an Example

FastJet tool taken as guinea pig: used to cluster jets by several experiments' software - [http://fastjet.fr/](http://fastjet.fr/)

Example coming from an old version of fastjet!

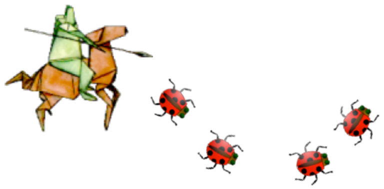


```
void VoronoiDiagramGenerator::plotinit()
{
  double dx,dy,d;

  dy = ymax - ymin;
  dx = xmax - xmin;
  d = (double)(( dx > dy ? dx : dy) * 1.1);
  pxmin = (double)(xmin - (d-dx)/2.0);
  pxmax = (double)(xmax + (d-dx)/2.0);
  pymin = (double)(ymin - (d-dy)/2.0);
  pymax = (double)(ymax + (d-dy)/2.0);
  cradius = (double)((pxmax - pxmin)/350.0);
  //GS unused: openpl();
  //GS unused: range(pxmin, pymin, pxmax, pymax);
}


void VoronoiDiagramGenerator::clip_line(Edge *e)
{
  Site *s1, *s2;
  double x1=0,x2=0,y1=0,y2=0; //, temp = 0;

  x1 = e->reg[0]->coord.x;
  x2 = e->reg[1]->coord.x;

   Value stored to 'x2' is never read

  y1 = e->reg[0]->coord.y;
  y2 = e->reg[1]->coord.y;

  //if the distance between the two points this line
  //the square root of 2, then ignore it
  //TODO improve/remove
  //if(sqrt(((x2 - x1) * (x2 - x1)) + ((y2 - y1) * (
  //  {
  //    return;
  //  }
  pxmin = borderMinX;
```



```
void VoronoiDiagramGenerator::clip_lir
{
  Site *s1, *s2;
  double x1=0,x2=0,y1=0,y2=0; //, temp

  x1 = e->reg[0]->coord.x;
  x2 = e->reg[1]->coord.x;

  Value stored to 'x2' is never read

  y1 = e->reg[0]->coord.y;
  y2 = e->reg[1]->coord.y;

  //if the distance between the two po
```

# Understanding and Debugging - GDB



# Debugging

Suppose something is wrong with your application:

- It nicely terminates but yields wrong results (worst case scenario!)
- It crashes
- It runs forever occupying several CPUs
- It hangs forever with no CPU usage (e.g. a deadlock)

**Effective debugging strategies and tools are the solution**

The same techniques are also handy not only in case of problems

- Suppose that the overall behaviour of a very complex application (~MLOC) is to be understood
  - E.g. CMS/Atlas/LHCb/Alice reconstruction

# Debugging Strategies

**Write and use programs without bugs?**

- There is *no such thing*, except in totally trivial cases
- All programs have and will have bugs

**If possible, try not to introduce bugs in the first place!**

Debug printouts as *'poor man's solution'*:

⊕ Immediate to everybody: sometimes it's enough!

⊖ Hard (impossible) to add printouts in 3rd party libraries

⊖ Distract the user from focussing on the debugging itself

⊖ Hard to use in a parallel program, encourage Heisen-Bugs influencing timing behaviour

**Or better: Use a debugger like GDB**

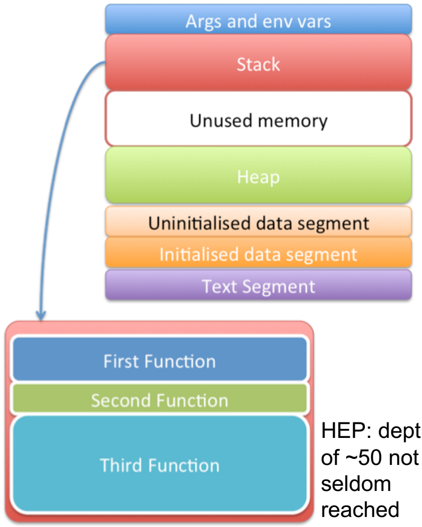
# ★ GDB: The GNU Project Debugger

- **Free and open source**, available on every Linux box
- GDB is an interactive command line tool which can "see":
  - Within a program during its execution
  - A posteriori, what a program was doing when it crashed
- Works with applications written in C and C++ (among other languages)
- No recompilation needed (although debugging symbols can be handy)
- Stop the execution at some specified point
  - Execute line by line, stepping into functions if needed
- Examine what is happening: e.g. print variable content
- Thread aware: e.g. Stop threads, switch among them …

**http://www.gnu.org/software/gdb/**

# Reminder: 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.
- **The heap**: Dynamic memory (e.g. requested with "`new`").



HEP: depth of ~50 not seldom reached

# An Example

```
#include <iostream>

void display(int x, int *xp) {
   std::cout << "In display():\n"
         << " o value of x is " << x
         << ", address of x is " << &x <<std::endl
         << " o xp points to " << xp
         << " which holds " <<  *xp <<std::endl; }

int main() {
   int a = 42;
   int *ap = &a;
   std::cout << "In main():\n"
        << " o value of a is "<< a
        << ", address of a is " << &a << std::endl
        << " o ap points to " << ap
        << " which holds " << *ap << std::endl;
   display(a, ap);
   return 0; }
```

```
g++ –o myExample myExample.cpp –g
```

To fire gdb: `gdb myexecutable`

# An Example

```
$ gdb myExample
[ … Some output … ]
(gdb) run
Starting program: /Users/<whoever>/gdb/myExample
Reading symbols for shared libraries
+++............................ done
In main():
 o value of a is 5, address of a is 0x7fff5fbff744
 o ap points to 0x7fff5fbff744 which holds 5
In display():
 o value of x is 5, address of x is 0x7fff5fbff71c
 o xp points to 0x7fff5fbff744 which holds 5

Program exited normally.
```

# Break Points

- So far so good – you could have done this already without GDB!
- **But,** GDB allows you to stop the execution of the application at a certain line or function with break points:

```
(gdb) break 12
Breakpoint 1 at 0x100000c60: file myExample.cpp, line 12.
(gdb) run
Starting program: /Users/<whoever>/gdb/myExample

Breakpoint 1, main () at myExample: 12
12 int *ap = &a;
(gdb)
```

The break could have been introduced when a certain function is invoked:

→***break <function name>***

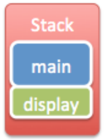
**(“*break display*” in our case)**

Impossible to do with printouts ☺

# And Now?

You can dump the stack with ***where***:

```
(gdb) where
#0  display (x=42, xp=0x7fff5fbff7c4) at myExample:6
#1  0x0000000100000d2c in main () at myExample.cpp:14
```



See some of the surrounding code with ***list***:

```
(gdb) list
1       #include <iostream>
2
3       void display(int x, int *xp) {
4          std::cout << "In display():\n"
5            << " o value of x is " << x << ", address of x is " <<
&x << std::endl
6             << " o xp points to " << xptr << " which holds " <<  *xp
<< std::endl;
7        }
8       int main() {
9          int a = 42;
10         int *ap = &a;
```

# ★ Interlude: Debugging Symbols

**The compiler does not automatically bring the names of the symbols in the executables and libraries,** the machine does not need them!

Humans do: include *debugging symbols* in the compiled binaries.

- Names of variables, functions, classes, namespaces, …

**Debugging symbols, 3 facts to remember:**

- Do not slow down the program!
- Do not increase its memory footprint!
- Do make binaries bigger (more disk space needed)!

With GCC:
*g++ […] –g*

# Navigating Program Execution

To “navigate” the program execution you can use:

- **step:** continue running until control reaches new line. “Step into” functions
- **next:** like step but functions are executed without stopping
- **finish:** continue until end of current stack frame
- **return <*expression*>:** prematurely exit the stack returning expression.
- **break:** show break points list
  - **disable** <*n*>: disable break point n
  - **enable** <*n*>: enable break point n
  - **delete** <n>: delete break point n

•**info threads:** show threads

•**thread <n>:** step into thread n

# The Print Statement

```
#include "time.h"
#include <iostream>
int main(){
   int t = clock();
   std::cout << t << std::endl;
   return 0;
}
```

**print** allows you to inspect the value of a variable.

```
(gdb) break 5
Breakpoint 1 at 0x100000d50: file ex12_2.cpp, line 5.
(gdb) run
Starting program: /Users/danilopiparo/gdb/ex12_2
Reading symbols for shared libraries
++............................ done

Breakpoint 1, main () at ex12_2.cpp:5
5           std::cout << t << std::endl;
(gdb) print t
$1 = 6637
(gdb) next
6637
6           return 0;
```

# Interlude 3: Machine Code with GDB

Food
For
Thought

```
double myFloor(double x){
  const int xi = int(x);
  return x<0?xi-1:xi;
 }

int main(){
  myFloor(-3.14);
 }
```

- Looking at the assembly is the only way to understand what the compiler actually did
- GDB allows to do that easily with `disass`

# Interlude 3: Machine Code with GDB

```
(gdb) disass /m myFloor
Dump of assembler code for function myFloor(double):
1       double myFloor(double x){
2         const int xi = int(x);
  0x00000000004004e0 <+0>:     cvttsd2si %xmm0,%eax
3         return x<0?xi-1:xi;
  0x00000000004004e4 <+4>:     cmpltsd 0x113(%rip),%xmm0        # 0x400600
  0x00000000004004ed <+13>:    lea     -0x1(%rax),%edx
  0x00000000004004f0 <+16>:    cvtsi2sd %eax,%xmm2
  0x00000000004004f4 <+20>:    cvtsi2sd %edx,%xmm1
  0x00000000004004f8 <+24>:    andpd   %xmm0,%xmm1
  0x00000000004004fc <+28>:    andnpd %xmm2,%xmm0
  0x0000000000400500 <+32>:    orpd    %xmm1,%xmm0

4       }
  0x0000000000400504 <+36>:    retq
End of assembler dump.
```

Food For Thought

# GDB And Threads

- GDB allows to inspect the behaviour of the threads of a process
  - `info threads`: display running threads
  - `thread <n>`: step into a thread

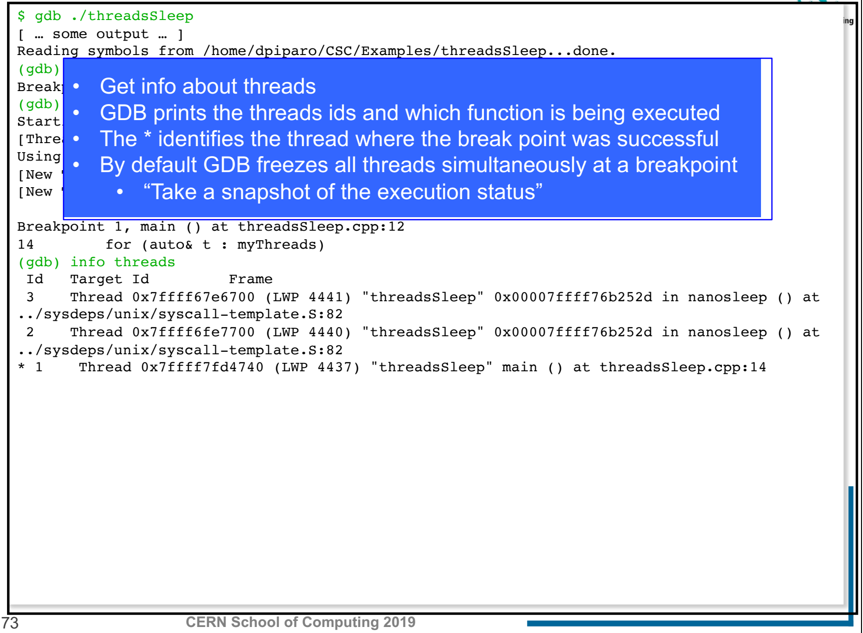
```
#include <thread>
#include <vector>
#include <chrono>

void sleep(){
 std::this_thread::sleep_for(std::chrono::seconds(100)); };

int main(){
 std::vector<std::thread> myThreads;
 for (int i=0; i<2; i++) myThreads.emplace_back(std::thread(sleep));
 // Line 11
 for (auto& t : myThreads)  t.join();
}
```

```
$ gdb ./threadsSleep
[ … some output … ]
Reading symbols from /home/dpiparo/CSC/Examples/threadsSleep...done.
(gdb) break 11
```

Set a break point at line 11

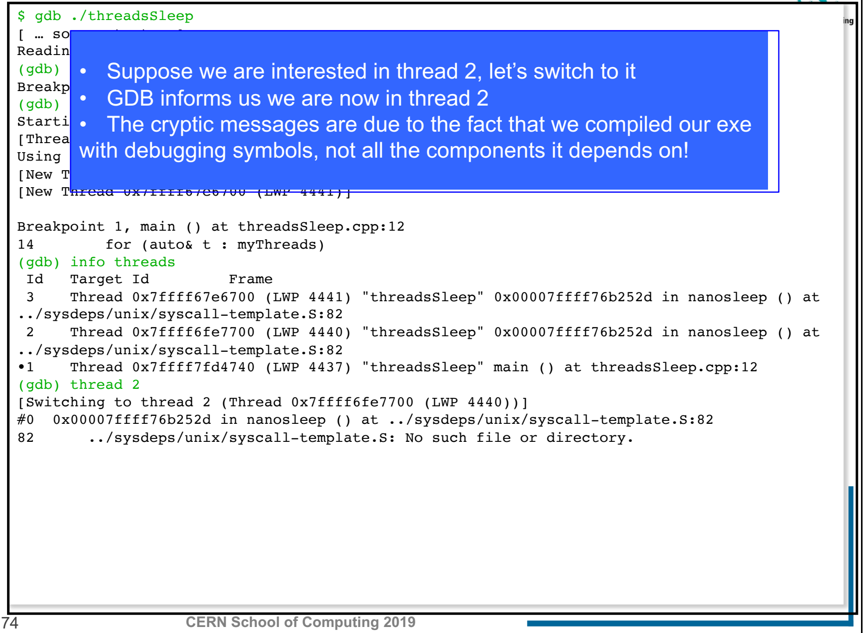
```
$ gdb ./threadsSleep
[ … some output … ]
Reading symbols from /home/dpiparo/CSC/Examples/threadsSleep...done.
(gdb) break 11
Breakpoint 1 at 0x400a8f: file threadsSleep.cpp, line 11.
(gdb) run
Starting program: /home/dpiparo/CSC/Examples/threadsSleep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff6fe7700 (LWP 4440)]
[New Thread 0x7ffff67e6700 (LWP 4441)]

Breakpoint 1, main () at threadsSleep.cpp:12
12        for (auto& t : myThreads)
```

- GDB informs us it found the line at which it will break
- Run the application
- GDB informs us that 2 threads were spawned
- The breakpoint is reached



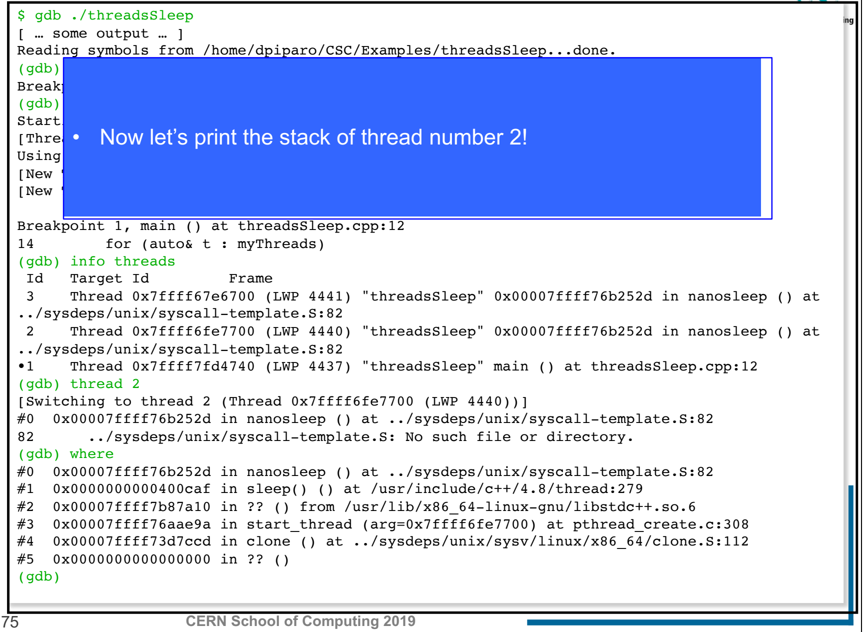
- Get info about threads
- GDB prints the threads ids and which function is being executed
- The * identifies the thread where the break point was successful
- By default GDB freezes all threads simultaneously at a breakpoint
  - “Take a snapshot of the execution status”

```
$ gdb ./threadsSleep
[ … some output … ]
Reading symbols from /home/dpiparo/CSC/Examples/threadsSleep...done.
(gdb)
Break
(gdb)
Start
[Thre
Using
[New
[New

Breakpoint 1, main () at threadsSleep.cpp:12
14         for (auto& t : myThreads)
(gdb) info threads
 Id   Target Id         Frame
 3    Thread 0x7ffff67e6700 (LWP 4441) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
 2    Thread 0x7ffff6fe7700 (LWP 4440) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
* 1    Thread 0x7ffff7fd4740 (LWP 4437) "threadsSleep" main () at threadsSleep.cpp:14
```



- Suppose we are interested in thread 2, let’s switch to it
- GDB informs us we are now in thread 2
- The cryptic messages are due to the fact that we compiled our exe with debugging symbols, not all the components it depends on!

```
$ gdb ./threadsSleep
[ … so
Readin
(gdb)
Breakp
(gdb)
Starti
[Threa
Using
[New T
[New Thread 0x7ffff67e6700 (LWP 4441)]

Breakpoint 1, main () at threadsSleep.cpp:12
14         for (auto& t : myThreads)
(gdb) info threads
 Id   Target Id         Frame
 3    Thread 0x7ffff67e6700 (LWP 4441) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
 2    Thread 0x7ffff6fe7700 (LWP 4440) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
•1    Thread 0x7ffff7fd4740 (LWP 4437) "threadsSleep" main () at threadsSleep.cpp:12
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff6fe7700 (LWP 4440))]
#0  0x00007ffff76b252d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
82      ../sysdeps/unix/syscall-template.S: No such file or directory.
```



- Now let’s print the stack of thread number 2!

```
$ gdb ./threadsSleep
[ … some output … ]
Reading symbols from /home/dpiparo/CSC/Examples/threadsSleep...done.
(gdb)
Break
(gdb)
Start
[Thre
Using
[New
[New

Breakpoint 1, main () at threadsSleep.cpp:12
14         for (auto& t : myThreads)
(gdb) info threads
 Id   Target Id         Frame
 3    Thread 0x7ffff67e6700 (LWP 4441) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
 2    Thread 0x7ffff6fe7700 (LWP 4440) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
•1    Thread 0x7ffff7fd4740 (LWP 4437) "threadsSleep" main () at threadsSleep.cpp:12
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff6fe7700 (LWP 4440))]
#0  0x00007ffff76b252d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
82      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) where
#0  0x00007ffff76b252d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
#1  0x0000000000400caf in sleep() () at /usr/include/c++/4.8/thread:279
#2  0x00007ffff7b87a10 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#3  0x00007ffff76aae9a in start_thread (arg=0x7ffff6fe7700) at pthread_create.c:308
#4  0x00007ffff73d7ccd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:112
#5  0x0000000000000000 in ?? ()
(gdb)
```

# GDB And Threads

- **By default GDB stops all threads simultaneously if a breakpoint is reached** (so called “stop mode”)
- It allows also to stop the thread where the breakpoint was reached and let the others proceed (“non-stop mode”)
  - De facto the user can bend the runtime behaviour of the application to her needs!

```
# Enable the async interface.
set target-async 1

# Pagination breaks non-stop.
set pagination off

# Finally, turn it on [off]!
set non-stop on [off]
```

Commands to switch between stop and non-stop modes within the gdb prompt

# More GDB (Black) Magic

Suppose your program behaves in a weird way now.

- You can “attach” gdb to a running process (e.g. 300% CPU since minutes…)
- ***gdb <PID>***

Impossible to do with printouts ☺

Get your pid:
ps aux | grep <Program name>

Suppose your program crashed after hours of running, leaving you with no plots, but a core dump.

- You can resume it as it was at the moment of the crash
- **gdb *program core-file***

# ★ Helgrind and DRD

- Another pair of tools useful for debugging parallel programs
- Part of the Valgrind suite
- Allow to catch thread errors at runtime
  - Valgrind –tool=helgrind ./myProgram

- Detection of potential thread unsafe operations, lock ordering problems, …
  - Difference between DRD and Helgrind: detection algorithms
- Downside: false positives ☹
- Complementary tools: address and thread sanitiser offered by CLANG and GCC compiler suites.

# High Level Profiling

```
// Templated function to compute
// sum of all parameters
template <class T, class ...P>
T sum(T t, P ...p)
{
  if (sizeof...(p))
  {
    t += sum(p...);
  }
  return(t);
}
[…]
auto theSum = sum(1.,3,5.f);
```

# A Simple Question

**Q:** Why should we strive for software performance, correctness, efficiency, ultimately throughput?

# For Money!

From "The Wolf of Wall Street"

# Code Optimisation

- When dealing with large software projects, **performance measurement is daily business**
  - Especially for multithreaded applications: *parallel Vs serial case, performance of different configurations of the parallel applications …*
- **The identification of the hotspots** (and their removal) is worth an enormous amount of resources ★
  - But don't optimise before you know "what"!
- A plethora of tools available, covering all quantities related to performance
- Focus on a simple and easy to use one: igprof
  - Identify features common to all profilers

http://igprof.org/

- Igprof answers questions like:
  - What are the symbols that have the longest runtime?
  - What are the symbols that allocate the most memory?

# The Golden Rule of Optimisation

Don't develop theories,
measure your program!

***It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.* Sherlock Holmes**

# ★ Igprof

- Very general tool, free and open source
- **Measure runtime (speed) and memory of an application**
  - Works in the multithreaded case
  - Different metrics available
- **Little or negligible overhead**
  - Runtime: 50 MB RSS and ~no runtime overhead
  - Memory: it depends. For a large HEP application (600 libraries, ~2GB of memory, allocations at ~1MHz): 1 GB RSS and 250% runtime
- Non intrusive
  - No instrumentation needed
  - No Kernel modules needed
  - Hooking mechanism implemented

Results can be looked at as web or ASCII reports

Today we focus on the runtime measurements

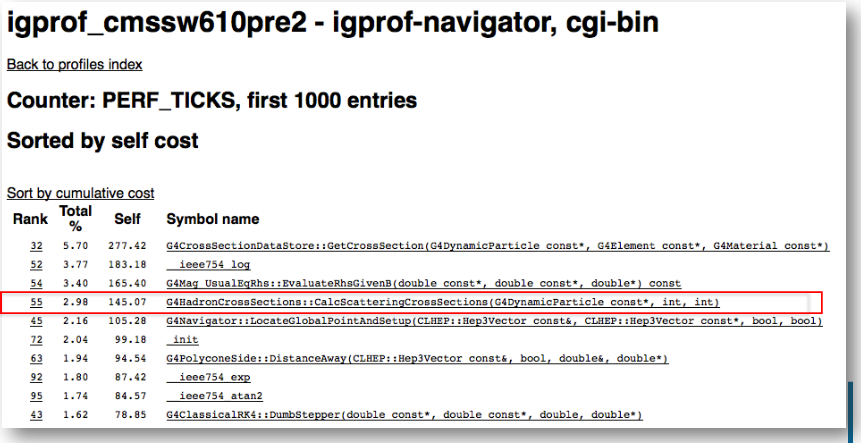
# Igprof Runtime Profiler

- **Igprof measures the time spent in all the functions in seconds**
  - Easy overall view, understand rapidly where the time is spent
- **Two kinds of costs per symbol (typical of ~every profiler):**
  - Cumulative cost: time spent in the symbol and callees (i.e. int main() has the biggest cumulative cost)
  - Self cost: time spent in the symbol itself

Both are necessary to find hot spots

**Produce profiling information:**
**igprof -d -pp -z -o myProfInfo.pp.gz myApp [arg1 arg2 ...]**
**Produce report out of it:**
**igprof-analyse --sqlite -d -v -g myProfInfo.pp.gz | sqlite3 igreport_perf.sql3**

# Igprof: An Example of CMS Simulation



## igprof_cmssw610pre2 - igprof-navigator, cgi-bin

Back to profiles index

### Counter: PERF_TICKS, first 1000 entries

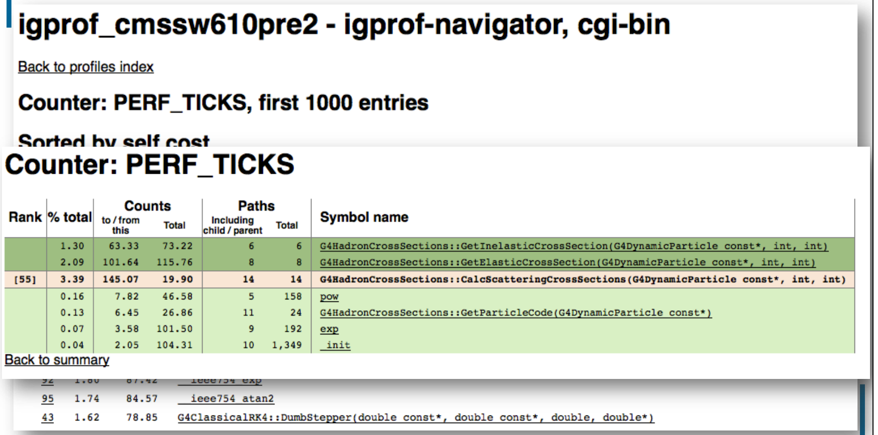
### Sorted by self cost

Sort by cumulative cost

| Rank | Total % | Self | Symbol name |
|---|---|---|---|
| 32 | 5.70 | 277.42 | G4CrossSectionDataStore::GetCrossSection(G4DynamicParticle const*, G4Element const*, G4Material const*) |
| 52 | 3.77 | 183.18 | __ieee754_log |
| 54 | 3.40 | 165.40 | G4Mag_UsualEqRhs::EvaluateRhsGivenB(double const*, double const*, double*) const |
| 55 | 2.98 | 145.07 | G4HadronCrossSections::CalcScatteringCrossSections(G4DynamicParticle const*, int, int) |
| 45 | 2.16 | 105.28 | G4Navigator::LocateGlobalPointAndSetup(CLHEP::Hep3Vector const&, CLHEP::Hep3Vector const*, bool, bool) |
| 72 | 2.04 | 99.18 | _init |
| 63 | 1.94 | 94.54 | G4PolyconeSide::DistanceAway(CLHEP::Hep3Vector const&, bool, double&, double*) |
| 92 | 1.80 | 87.42 | __ieee754_exp |
| 95 | 1.74 | 84.57 | __ieee754_atan2 |
| 43 | 1.62 | 78.85 | G4ClassicalRK4::DumbStepper(double const*, double const*, double, double*) |

Real measurement, not the latest greatest: it does not reflect the current state of the CMS software.

# Igprof: An Example of CMS Simulation



## igprof_cmssw610pre2 - igprof-navigator, cgi-bin

Back to profiles index

### Counter: PERF_TICKS, first 1000 entries

### Sorted by self cost

### Counter: PERF_TICKS

| Rank | % total | Counts | | Paths | | Symbol name |
|---|---|---|---|---|---|---|
| | | to / from this | Total | Including child / parent | Total | |
| | 1.30 | 63.33 | 73.22 | 6 | 6 | G4HadronCrossSections::GetInelasticCrossSection(G4DynamicParticle const*, int, int) |
| | 2.09 | 101.64 | 115.76 | 8 | 8 | G4HadronCrossSections::GetElasticCrossSection(G4DynamicParticle const*, int, int) |
| [55] | 3.39 | 145.07 | 19.90 | 14 | 14 | G4HadronCrossSections::CalcScatteringCrossSections(G4DynamicParticle const*, int, int) |
| | 0.16 | 7.82 | 46.58 | 5 | 158 | pow |
| | 0.13 | 6.45 | 26.86 | 11 | 24 | G4HadronCrossSections::GetParticleCode(G4DynamicParticle const*) |
| | 0.07 | 3.58 | 101.50 | 9 | 192 | exp |
| | 0.04 | 2.05 | 104.31 | 10 | 1,349 | _init |

Back to summary

| | | | |
|---|---|---|---|
| 92 | 1.80 | 87.42 | __ieee754_exp |
| 95 | 1.74 | 84.57 | __ieee754_atan2 |
| 43 | 1.62 | 78.85 | G4ClassicalRK4::DumbStepper(double const*, double const*, double, double*) |

Real measurement, not the latest greatest: it does not reflect the current state of the CMS software.

# Take Away Messages

Dealing with a parallel application is complex:

Use procedures to **rise fences to protect against mistakes**, like static analysis to find bugs in an automatic way

- **Embed such tools in the build infrastructure of your SW**

Use tools to **inspect, manipulate their behaviour at runtime, like GDB**

- Become familiar with them, multithreaded programs are tough to debug

**Use tools to measure performance**, do not speculate

- Start from simple yet powerful tools like igprof
- Choose more complex ones to dive into the details