Writing Parallel Software
This Lecture

The outline:

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

---

**Diagram:**

- Long calculation
- Main “line of work”
- Time
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 other words, code to be executed is passed around
std::async in Action

Header for async and future

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

int lengthyCalculation(){ [...] };
void doOtherStuff(){ [...] };

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

"Launch" the calculation
Retreive result
std::async in Action

```cpp
std::future<int> myAnswer = std::async(lenghtyCalculation);
myAnswer.get()
```
std::async in Action

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

It's easy after all, isn't it?
Well, to be Honest… No.

- Unfortunately scientifically relevant / potentially lucrative real life use cases are complex
  - Cannot be solved simply throwing threads at them

- In addition, many existing high-quality sequential large software packages are in production
  - Starting fresh may not be always possible

- Example: software stack of an LHC experiment
  - Tens of (large) packages integrated
  - $O(10^2)$ shared libraries
  - Experiment specific code
  - $\rightarrow$ Millions of nicely working lines of code

Need to think parallel
- Evolve the existing systems
- Be disruptive and think to the future
Parallel Software Design: an Introduction
First Step: Finding Concurrency

What can be executed concurrently?

Two techniques to figure this out:

- **Data decomposition**
  - The partition of the data domain
  - Achieve data parallelism

- **Task decomposition**
  - Split according to logical tasks
  - Achieve task parallelism

This step takes place in front of a whiteboard
**Data Parallelism**

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

*An illustration:* multiplication of an array of values

Data parallelism implies wise design of the data structures to be used!
Increase floating point throughput acting on mathematical functions:

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

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

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

A bit too simple: no dependency between tasks!
Task Parallelism: An example

HEP data processing frameworks

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

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

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

- What is the performance gain we can expect?
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

\[
\text{Speedup} = \frac{\text{Time}_{\text{serial}}}{\text{Time}_{\text{parallel}}}
\]
Amdahl’s Law

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

\[
\text{Speedup} = \frac{1}{(1 - p) + \frac{p}{n}}
\]

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

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

From E. Tejedor
Threads and C++
Let’s change gears: Threads

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

- The kernel manages the existing threads, scheduling them to the available resources (CPUs)*
  - There can be more threads in a single process than cores in the machine!

* Actually mapping user threads to kernel threads, but this simplification ok in first order!
Interlude: A Program in Memory

- **Text Segment**: code to be executed.

- **Initialized Data Segment**: global variables initialized by the programmer.

- **Uninitialized Data Segment**: This segment contains uninitialized global variables.

- **The stack**: The stack is a collection of stack frames. It grows whenever a new function is called. “Thread private”.

- **The heap**: Dynamic memory (e.g. requested with “new”).
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”).

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 or Processes?

Some additional elements to consider for the decision:

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

Writing Parallel Software
Threads in C++

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

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…

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

int main()
{
    std::thread t([]{
        std::cout << "Hello Concurrent World!\n";
    });
    t.join();
}
```
A possible prototype backend behind task oriented programming!

A First Abstraction

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

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

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

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

Identify the thread

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

Limitation: cannot retrieve the return value.
A First Abstraction

A possible prototype backend

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

void printThreadID(int i) {
    printf("thread num %d – id 0x%lu
", i, std::this_thread::get_id);
}

int main() {
    std::vector<std::thread> myThreads;
    myThreads.reserve(10);
    for (int i = 0; i < 10; i++)
        myThreads.emplace_back(printThreadID, i);
    for (auto& t : myThreads)
        t.join();
    return 0;
}
```

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

Limitation: cannot retrieve the return value.

When dealing with concurrency, asynchronous events are daily business!
The Thread Pool Model

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

Hard to program in an optimised and general way! (usually provided by 3rd part libraries)
Processes in Python/C++

Python

- Handy *multiprocessing* module

C++

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

```python
from multiprocessing import Process, Pool

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

def g(x):
    return x*x

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

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

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

* root.cern.ch
Threads and Data Races: Synchronisation Issues
The Problem

- Fastest way to share data: access the same memory region
  - One of the advantages of threads

- Parallel memory access: delicate issue - race conditions
  - I.e. behaviour of the system depends on the sequence of events which are intrinsically asynchronous

- Consequences, in order of increasing severity
  - Catastrophic terminations: segfaults, crashes
  - Non-reproducible, intermittent bugs
  - Apparently sane execution but data corruption: e.g. wrong value of a variable or of a result

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

Desired

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>X Val.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X (44)</td>
<td>Read X (44)</td>
<td>44</td>
</tr>
<tr>
<td>Add 10</td>
<td>Subtract 12</td>
<td>44</td>
</tr>
<tr>
<td>Write X (54)</td>
<td>Write X (32)</td>
<td>32</td>
</tr>
<tr>
<td>Write X (32)</td>
<td>Add 10</td>
<td>32</td>
</tr>
<tr>
<td>Write X (42)</td>
<td>Write X (42)</td>
<td>42</td>
</tr>
</tbody>
</table>

RACE

2 Threads only
No crash
Bogus results!
What is not Thread Safe?

Everything, unless explicitly stated!

In four words: *Shared States Among Threads*

Examples:

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

It sounds depressing. But there are several ways to protect thread unsafe resources!
Useful Design Principles
Minimise Contention

- Designing and implementing software for the serial case to make it parallel afterwards
  - Not exactly a winning strategy

- Rather think parallel right from the start
  - Advice not straightforward to put in place
  - Needs careful planning and thinking

- Depends on the problem being studied
  - Understand what you are doing!
Ex. Functional Programming Style

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

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

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

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

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

Without becoming purists, functional programming principles can avoid lots of headaches typical of parallel programming
Replication, Atomics, Transactions and Locks
Why so many strategies?

- There is no silver bullet to solve the issue of “resource protection”
  - Complex problem

- Case by case investigation needed
  - Better to be aware of many strategies

- Best solution: often a trade-off
  - The lightest in the serial case?
  - The lightest in presence of high contention?
One copy of the data per Thread

- Sometimes it can be useful to have thread local variables
  - A “private heap” common to all functions executed in one thread

- **Thread Local Storage (TLS)**

- Replicate per thread some information
  - C++ keyword `thread_local`

- E.g.: build “smart-thread-local pointers”
  - Deference: provide the right content for the current thread

- **Not to “one size fits them all” solution**
  - Memory usage
  - Overhead of the implementation, also memory allocation strategy
  - Cannot clutter the code with `thread_local` storage specifiers
# TLS in Action

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

thread_local unsigned int tlIndex(0);

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

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

One private copy per thread will exist

Be patient for a moment ;-)
# TLS in Action

```cpp
#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();
}
```

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-splitable by other threads
  - Other real life examples: finance, database transactions
  - Either entirely successful (subtract from A, add to B) or rolled back

- C++ offers support for atomic types
  - `#include <atomic>`
  - Usage: `std::atomic<T>`

- Operations supported natively vary according to T
  - Subtleties present: e.g. cannot instantiate `atomic<MyClass>` under all circumstances (must be trivially copyable)

- For example:
  - boolean, integer types. E.g. `std::atomic<unsigned long>`
  - Pointer to any type. E.g. `std::atomic<MyClass*>`
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

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

- Make a section of the code executable by one thread at the time

- Locks should be avoided, but yet known
  - They are a blocking synchronisation mechanisms
  - They can suffer pathologies
  - … they could be present in existing code: use your common sense and a grain of salt!

Terminology:

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

A lock can be …

- **a spin lock**: if it makes a task spin while waiting ("busy wait")
  - Short tasks: spin is better (putting a thread to sleep costs cycles)
  - Big implications also in terms of power consumption

- **Scalable**: cannot perform worse than serial execution

- **Fair**: it lets threads through in the order they arrive

- **Recursive**: it can be acquired multiple times by the same thread

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

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

Only one thread at the time can access this section.

Acquire/release lock on the mutex.
A first Lock Example

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

- 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

Instance of a class, locks the scope!

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

- Construct an object which lives in the scope to be locked
- C++ provides a class to ease this: `std::lock_guard<T>(T&)`
- When the scope is left, the object destroyed and the lock released
- Application of the RAII idiom (Resource Acquisition Is Initialisation)
  - RAII: “bread and butter” in modern and performant C++
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.
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
Amdahl’s Law

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

\[ \text{Speedup} = \frac{1}{(1 - p) + \frac{p}{n}} \]

- \( n \): number of cores
- \( p \): parallel portion

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

From E. Tejedor
Take Away Messages

- Choose designs that follow principles such as data and task parallelism
  - They lead to scalable and performant applications
  - Focus on algorithms and data structures!

- Asynchronous execution and non-determinism permeate concurrent applications:
  - Paradigm shift needed to understand and design parallel software solution

- Abstraction needed: e.g. thread pool
  - Do not forget the basics: ownership, OS, hardware

- Choose from the start a design which helps avoiding data races:
  - Understand your problem: no silver bullet
  - Prefer approaches w/o global states (e.g. functional)

- Choose non blocking mechanisms whenever possible
  - E.g. atomics and transactions
  - Locks can be present in existing software
  - Use a grain of salt