CSC Latin America 2026

Tools & Techniques + Software Design

High-Performance Computing for Experimental Physics and Data Analysis

Arturo Sánchez Pineda — INAIT AI

January 2026

https://indico.cern.ch/e/CSC2026-Chile

Welcome & Overview

What you'll learn:

- **Tools & Techniques:** Correctness, collaboration, and performance fundamentals
- **Software Design:** Multi-core computing, data layout, and optimisation strategies
- **Practical Skills:** Hands-on exercises with real HEP workflows

Why this matters:

Modern HEP experiments generate petabytes of data. Writing correct, maintainable, and performant code isn't optional — it's essential.

Philosophy: Correctness first, then performance — always with collaboration in mind.

About Me

Arturo Sánchez Pineda — from Venezuela

- Senior DevOps Engineer and Infrastructure Responsible at INAIT AI, Switzerland.
- PhD in Physics, University of Naples Federico II.
- Former co-creator and coordinator of the ATLAS Open Data project at CERN.
- Former researcher/educator at ULA, ICTP, INFN, University of Udine.
- Co-founder of LA-CoNGA physics (Latin American alliance for Capacity-building in Advanced Physics), CEVALE2VE, and the Creative Commons Venezuela and Switzerland chapters.

Mission: Making scientific education accessible worldwide — especially in Latin America.

Schedule at a Glance

Day	Session	Topic
Mon, Jan 12	12:30	T&T L1: Correctness first
	14:30	T&T L2: From my code to our code
	17:00–18:00	T&T E1–E2: Exercises
Tue, Jan 13	11:30	SD L1: Many-core foundations
Wed, Jan 14	09:45	SD L2: Patterns & data layout
	12:30	T&T L3: Performance fundamentals
	14:30–15:30	SD E1–E2: Exercises
Thu, Jan 15	11:30	SD L3: Profiling to optimisation

Courses Philosophy

Three Pillars of Quality Software:

1. **Correctness** — Does it produce the right answer?
2. **Maintainability** — Can others (including future you) work with it?
3. **Performance** — Does it run fast enough for production?

In that order. A fast, wrong answer is useless. An unmaintainable codebase dies.

My approach: Learn tools and techniques that enforce correctness, enable collaboration, and unlock performance — systematically.

Fun fact: This course uses real **ATLAS Open Data at 13 TeV**. (The Higgs boson was discovered with 7–8 TeV data recorded in 2011–2012.)

Learning Objectives

By the end of this course, you will:

- ✅ Debug efficiently with sanitisers (ASan/UBSan/TSan)
- ✅ Write robust tests with Catch2/pytest
- ✅ Collaborate at scale with Git workflows and CI/CD
- ✅ Measure performance scientifically
- ✅ Parallelise code correctly with OpenMP/TBB
- ✅ Optimise systematically
- ✅ Work with real HEP data (ROOT + ATLAS Open Data)

Assessment: Hands-on exercises. You'll do these things, not just hear about them.

Tools We'll Use

Languages & Frameworks:

- C++17/20, Python 3.x
- ROOT (CERN's data analysis framework)
- Catch2 (C++ testing), pytest (Python testing)

Build & CI:

- CMake, Git, GitHub Actions, GitLab CI

Performance & Debugging:

- GDB, ASan/UBSan/TSan, perf, Google Benchmark
- OpenMP, oneTBB (Threading Building Blocks)

Infrastructure Context:

- CVMFS, EOS, Apptainer (Singularity)

TOOLS & TECHNIQUES

Lecture 1: Correctness First

"Make it work, make it right, make it fast — in that order"

60 minutes

Learn to catch bugs before they become disasters

T&T L1: Correctness First

Why Correctness Matters

Real ATLAS examples:

Case 1: A memory leak in track reconstruction running on the Worldwide LHC Computing Grid (WLCG) — 500,000 cores processing data 24/7. One leak wastes **years of CPU time overnight**.

Case 2: An off-by-one error in event selection code. Months of analysis were invalidated. Paper submission delayed. Career impact for PhD students.

Case 3: Race condition in parallel Higgs analysis. Intermittent wrong results that only appear in 1% of runs. Nearly impossible to debug without proper tools.

In HEP: We're not just writing code — we're writing **scientific instruments**. Bugs don't just crash programs; they corrupt physics results and waste years of work.

The Testing Mindset

Interactive question: Raise your hand if you've ever...

- Spent hours debugging, then found the bug was a typo?
- Had code that "worked on your machine" but failed elsewhere?
- Refactored code and accidentally broke something that was working?
- Trusted a result, then found it was wrong months later?

The pattern: All these are preventable with testing.

The Testing Mindset

Question: When should you write tests?

Wrong Answers:

- "After the code works"
- "When I have time"
- "For important code only"
- "When someone asks"

Right Answer:

- **Always**
- **From the start**
- **For all code**
- **Because science**

Test-Driven Development (TDD): Write the test first, watch it fail, make it pass, refactor.

Testing Frameworks: Catch2 (C++)

```cpp
#include <catch2/catch_test_macros.hpp>

TEST_CASE("Energy is correct", "[physics]") {
    Particle p{1.0, 2.0, 3.0, 0.139};
    REQUIRE(p.energy() == Catch::Approx(3.744).epsilon(0.001));
}

TEST_CASE("Momentum conserved", "[physics]") {
    Particle p1{1.0, 0.0, 0.0, 0.139};
    Particle p2{-1.0, 0.0, 0.0, 0.139};
    REQUIRE((p1 + p2).px() == Catch::Approx(0.0).margin(1e-5));
}
```

Run: `cmake --build build && ctest`

Testing Frameworks: pytest (Python)

```python
from analysis import ParticleSelector

def test_pt_cut():
    sel = ParticleSelector(pt_min=20.0)
    events = [{"pt": 15.0}, {"pt": 25.0}, {"pt": 30.0}]

    result = sel.apply(events)
    assert len(result) == 2
    assert all(e["pt"] >= 20.0 for e in result)

def test_empty_input():
    sel = ParticleSelector(pt_min=20.0)
    assert sel.apply([]) == []
```

Run: `pytest -v`

The Holy Trinity of Sanitisers

Three Tools That Change Everything

ASan (AddressSanitizer)

- Detects memory errors
- Buffer overflows
- Use-after-free
- Memory leaks

UBSan (UndefinedBehaviorSanitizer)

- Catches undefined behaviour
- Integer overflow
- Null pointer dereference
- Division by zero

TSan (ThreadSanitizer)

- Finds data races
- Detects race conditions
- Thread safety violations
- Lock order inversions

Sanitisers: A Demonstration

Without ASan (Traditional Debugging)

```
$ gdb ./reconstruction
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0×00007ffff7a5d123 in ?? ()
(gdb) bt
#0  0×00007ffff7a5d123 in ?? ()
#1  0×0000000000000000 in ?? ()
```

Time to find bug: 15–45 minutes of detective work

Sanitisers: A Demonstration

With ASan (Modern Debugging)

```
$ ./reconstruction
==12345==ERROR: AddressSanitizer: heap-buffer-overflow on address
    0x603000000044 at pc 0x0000004a2b76 bp 0x7fffffffd950 sp 0x7fffffffd948
READ of size 4 at 0x603000000044 thread T0
    #0 0x4a2b75 in TrackReconstructor::processHit(Hit const&)
       reconstruction.cpp:142
    #1 0x4a1f32 in TrackReconstructor::run()
       reconstruction.cpp:87
    #2 0x4a0d21 in main main.cpp:23

0x603000000044 is located 4 bytes to the right of 64-byte region
allocated by thread T0 here:
    #0 0x4a3e88 in operator new(unsigned long)
```

Time to find bug: Instant, with exact line number

Demo: Sanitisers in Action

The Bug

```
void processEvents(const std::vector<Event>& events) {
    float energies[100];

    for (size_t i = 0; i ⩽ events.size(); ++i) {  // BUG: ⩽ should be <
        energies[i] = events[i].totalEnergy();
    }
    // More processing...
}
```

What happens:

- Without sanitisers: Random crashes, corrupted data, or "works on my machine"
- With ASan: Immediate error message pointing to line with `⩽`

Common Debugging Horror Stories

The Heisenbug:

"Crashes only on Fridays after 3 pm when I'm not watching."

→ Race condition. TSan catches it instantly.

"It Worked Yesterday":

"I didn't change anything!"

→ Uninitialised memory. UBSan catches it.

The Performance Cliff:

"Gets slower and slower, then crashes after 6 hours."

→ Memory leak. ASan catches it.

Lesson: Sanitisers turn mysterious bugs into obvious errors.

Static Analysis: Catching Bugs Before Runtime

clang-tidy

What it does: Analyses code without running it, finds potential bugs and style violations

```
// Code:
void updateMomentum(Particle* p) {
    if (p->mass() = 0.0) {  // BUG: = instead of ==
        p->setMomentum(0, 0, 0);
    }
}
```

clang-tidy output:

```
warning: using the result of an assignment as a condition
         without parentheses [bugprone-assignment-in-if-condition]
    if (p->mass() = 0.0) {
        ~~~~~~~~~~^~~~
```

C++ Core Guidelines

Modern C++ Best Practices

Key principles we'll follow:

- **P.1:** Express ideas directly in code
- **P.5:** Prefer compile-time checking to run-time checking
- **F.7:** For general use, prefer `unique_ptr` to raw pointers
- **ES.20:** Always initialise an object
- **C.45:** Don't define a default constructor that only initialises data members

Tool: `clang-tidy` with `-checks=cppcoreguidelines-*`

Resources: https://isocpp.github.io/CppCoreGuidelines/

Reproducible Environments

The "Works On My Machine" Killer

Common scenario:

- Developer: "The code works perfectly on my laptop"
- CI: ❌ Build failed
- Colleague: "I get segfaults when I run it"
- Production: Catastrophic failure!

Root cause: Environment differences (compiler versions, libraries, OS quirks)

Solution: Dev Containers + Docker

Dev Containers

Containerised Development Environment

`.devcontainer/devcontainer.json`:

```
{
  "name": "HEP Analysis Environment",
  "image": "gitlab-registry.cern.ch/linuxsupport/alma9-base:latest",
  "features": {
    "ghcr.io/devcontainers/features/git:1": {},
    "ghcr.io/devcontainers/features/common-utils:2": {}
  },
  "postCreateCommand": "bash -lc 'source /cvmfs/sft.cern.ch/lcg/views/LCG_104/x86_64-el9-gcc13-opt/setup.sh'",
  "customizations": {
    "vscode": {
      "extensions": ["ms-vscode.cpptools", "ms-python.python"]
    }
  }
}
```

HEP Context: CVMFS vs Containers

CVMFS (CernVM-FS)

- **Purpose:** Software distribution
- Read-only file system
- Cached globally via Squid
- `/cvmfs/sft.cern.ch/`
- Perfect for: Sharing LCG releases

Containers (Apptainer/Docker)

- **Purpose:** Runtime isolation
- User namespaces
- Portable execution
- Bundled dependencies
- Perfect for: Reproducible jobs

Best practice: CVMFS for distribution layer + Containers for runtime layer

Summary: T&T L1

Key Takeaways:

1. **Test everything** — Use Catch2/pytest from day one
2. **Sanitisers save time** — ASan/UBSan/TSan find bugs instantly
3. **Static analysis catches bugs early** — Run clang-tidy in CI
4. **Reproducibility matters** — Dev Containers eliminate environment issues
5. **HEP infrastructure** — CVMFS for distribution, containers for execution

Next: How do we move from "my code" to "our code"?

TOOLS & TECHNIQUES

Lecture 2: From My Code to Our Code

"Individual brilliance doesn't scale — collaboration does"

60 minutes

Master Git, CI/CD, and workflows for 3,000-person teams

T&T L2: From My Code to Our Code

Motivation

Individual brilliance doesn't scale.

In HEP experiments:

- 3,000+ collaborators across 40+ countries
- Millions of lines of code (Athena, CMSSW)
- Code from 2008 is still running!
- 6 continents, 12 time zones

Challenge: Write code that survives collaboration

Example: ATLAS Athena: 6M lines of C++/Python, 500+ developers, 20k commits/year.

Git for Large Collaborations

Branching Models

Feature Branch Workflow:

```
main
  ├── feature/track-reco
  ├── feature/electron-id
  └── hotfix/memory-leak
```

Used by: CMSSW, Athena

Git Flow:

```
main (production)
  └── develop
        ├── feature/xyz
        └── release/v1.2
```

Used by: Many ROOT projects

Key principle: `main` is always deployable. Development happens in branches.

Commit Hygiene

Atomic Commits

Bad commit:

```
commit abc123
"Fixed stuff"
- Fixed track reconstruction bug
- Updated documentation
- Refactored particle class
- Added new analysis
- Cleaned up whitespace
```

Good commits:

```
commit def456
"Fix: Correct momentum calculation in TrackReconstructor"

commit ghi789
"Docs: Add usage examples for ParticleAnalysis"

commit jkl012
"Refactor: Extract ParticleFilter into a separate class"
```

Why? Each commit should represent a single logical change.

Why Atomic Commits Matter

Enable `git bisect`

Scenario: A bug was introduced sometime in the last 100 commits

```
$ git bisect start
$ git bisect bad HEAD               # Current version is broken
$ git bisect good v1.2.3            # v1.2.3 was working
Bisecting: 50 revisions left to test

# Git automatically checks out a commit
$ ./run_tests.sh
$ git bisect good  # or bad

# Repeat ~7 times ...
# Git identifies the exact commit that broke it
```

This only works if commits are atomic and buildable!

Commit Messages: Good vs Bad

❌ Bad:

```
"fix"
"update"
"stuff"
"final version 2"
"I hate git"
```

✅ Good:

```
Fix: Correct pT threshold

Updated from 24 to 25 GeV
to match the Run 3 trigger.

JIRA: ATL-PHYS-2026-123
```

Why? Six months later, which one helps you understand what changed?

Commit Message Rules

1. Separate subject from body with a blank line
2. Limit subject line to 50 characters
3. Capitalise the subject line
4. Do not end subject with a period
5. Use imperative mood in subject ("Fix bug" not "Fixed bug")
6. Wrap body at 72 characters
7. Use body to explain what and why, not how

Template:

```
Short (50 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 72 characters. Explain the problem this commit solves.

Fixes: #42
```

Continuous Integration as a Factory

Not Just Testing—Production

Modern CI does:

1. ✅ **Build** the code (multiple compilers, platforms)
2. ✅ **Test** functionality (unit, integration, system)
3. ✅ **Analyse** code quality (static analysis, coverage)
4. ✅ **Benchmark** performance (track regressions)
5. ✅ **Publish** artifacts (plots, documentation, binaries)
6. ✅ **Deploy** (if tests pass)

Philosophy: If CI passes, it's production-ready.

Dual-Stack CI: GitHub + GitLab

Why Both?

GitHub Actions:

- Great for open-source projects
- Easy integration with GitHub features
- Large ecosystem of actions and industry standard

GitLab CI (CERN Standard):

- Integrated with CERN infrastructure
- Access to CVMFS, EOS, LXPLUS runners
- Better for internal & self-hosted projects

Best practice: Master GitLab CI (what CERN uses), understand GitHub Actions (almost everywhere else)

GitHub Actions: Basic CI

```yaml
name: CI
on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    container: rootproject/root:latest
    steps:
      - uses: actions/checkout@v4
      - run: cmake -B build -DCMAKE_BUILD_TYPE=Release
      - run: cmake --build build -j
      - run: ctest --test-dir build --output-on-failure
```

Add sanitisers as a separate job for cleaner output.

GitLab CI (.gitlab-ci.yml)

```yaml
stages: [build, test]

build:
  stage: build
  image: gitlab-registry.cern.ch/linuxsupport/alma9-base
  script:
    - source /cvmfs/sft.cern.ch/ ... /setup.sh
    - cmake -B build && cmake --build build
  artifacts:
    paths: [build/]

test:
  stage: test
  script:
    - ctest --test-dir build --output-on-failure
```

Key: Artifacts pass built files between stages.

CI in Action

The Workflow

1. **Student opens PR:** "Add electron selection criteria"
2. **CI automatically:**
   - ✅ Builds code
   - ✅ Runs tests
   - ✅ Compiles ROOT analysis macro
   - ✅ Generates histogram
   - ✅ Posts comment with plot link
3. **Reviewer sees:**
   - All checks passed
   - Visual comparison of electron distributions
   - Code coverage report
4. **Decision:** Merge with confidence

Publishing Artifacts from CI

```
benchmark:
  stage: test
  script:
    - ./run_benchmarks.sh > benchmark_results.txt
    - python scripts/generate_plots.py
  artifacts:
    paths:
      - benchmark_results.txt
      - plots/
    reports:
      metrics: metrics.txt
    expire_in: 1 week
  only:
    - main
    - merge_requests
```

Result: Benchmark results and plots are available directly in the merge request for review.

Code Review Best Practices

For Authors

- Keep PRs small (<400 lines if possible)
- Write descriptive PR descriptions
- Link to relevant issues
- Respond to feedback promptly
- Don't take criticism personally

For Reviewers

- Review within 24 hours
- Focus on correctness first, style second
- Ask questions, don't demand changes
- Approve when good enough, not perfect
- Use inline comments for specific issues

Semantic Versioning

MAJOR.MINOR.PATCH

Example: v2.3.1

- **MAJOR** (2): Incompatible API changes
- **MINOR** (3): Backwards-compatible new features
- **PATCH** (1): Backwards-compatible bug fixes

Rules:

- Start at v0.1.0 for initial development
- v1.0.0 means "production-ready with stable API"
- Increment MAJOR when breaking changes
- Increment MINOR when adding features
- Increment PATCH for bug fixes

Learn more: https://semver.org/

CHANGELOG.md

Keep Users Informed

```
# Changelog

## [2.3.1] - 2026-01-15
### Fixed
- Corrected momentum calculation in TrackReconstructor (#142)
- Fixed memory leak in event loop (#145)

## [2.3.0] - 2026-01-08
### Added
- Support for Run 3 data format
- New electron identification algorithm
### Changed
- Improved performance of track fitting by 30%
### Deprecated
- Old track fitting API (will be removed in v3.0.0)
```

Release Workflow

Tagging and Distribution

```
# 1. Update version in code
sed -i 's/VERSION "2.3.0"/VERSION "2.3.1"/' CMakeLists.txt
# 2. Update CHANGELOG.md
vim CHANGELOG.md

# 3. Commit changes
git add CMakeLists.txt CHANGELOG.md
git commit -m "Release v2.3.1"
# 4. Create an annotated tag
git tag -a v2.3.1 -m "Version 2.3.1 - Bug fixes"

# 5. Push tag
git push origin v2.3.1

# CI automatically builds and publishes release artifacts
```

Summary: T&T L2

Key Takeaways:

1. **Branching models** — Feature branches keep main stable
2. **Atomic commits** — Enable git bisect and clean history
3. **CI is production** — Automate build, test, analyse, publish
4. **Dual-stack CI** — GitHub Actions + GitLab CI
5. **Semantic versioning** — Communicate changes clearly
6. **CHANGELOG.md** — Document what changed and why

Next: Performance fundamentals — how to make code fast

TOOLS & TECHNIQUES

Lecture 3: Performance Fundamentals

"Premature optimisation is the root of all evil, but procrastination is worse"

60 minutes

Learn why code is slow and how to make it fast — scientifically

T&T L3: Performance Fundamentals

Why Performance Matters in HEP

Scale of the problem:

- LHC collisions: 40 MHz proton bunch crossings
- ATLAS/CMS: ~1 petabyte of data per second (before trigger)
- After trigger: ~1 GB/s to disk
- Full Run 3: ~100 petabytes per year

Reality: We can't store or process everything. Performance determines what science is possible.

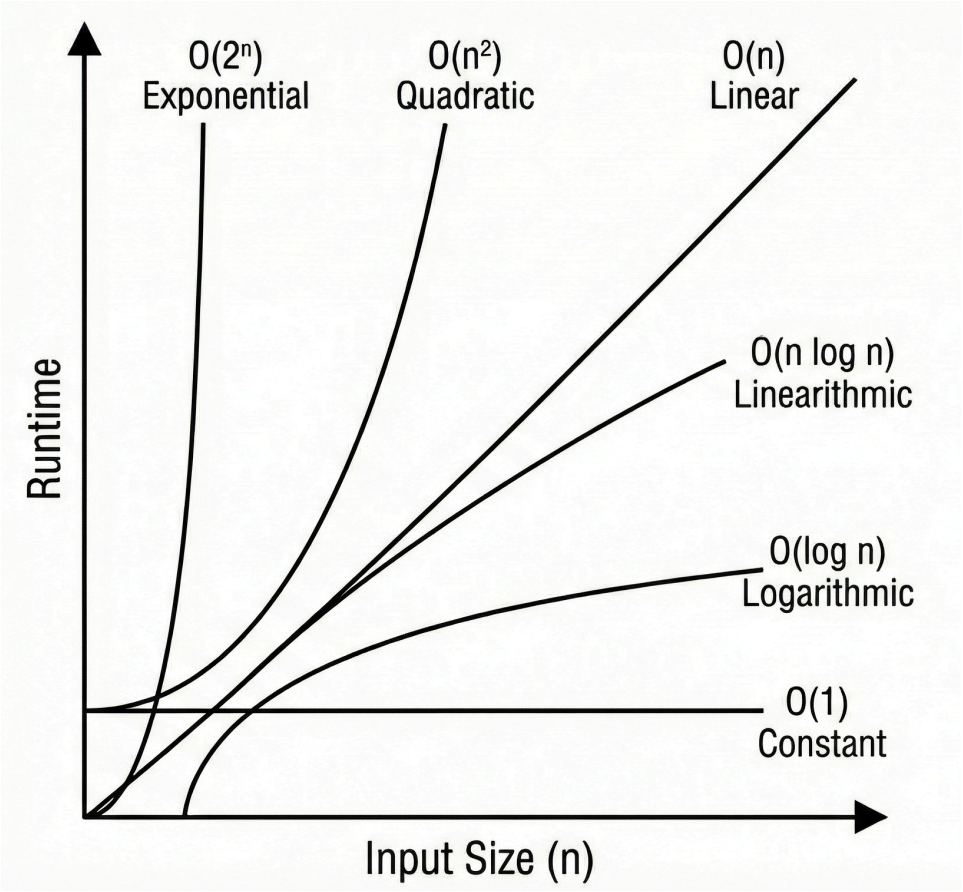
Algorithmic Complexity

Big-O Notation

Definition: How runtime scales with input size n

Complexity	Name	Example
O(1)	Constant	Array access
O(log n)	Logarithmic	Binary search
O(n)	Linear	Single loop
O(n log n)	Linearithmic	Merge sort
$O(n^2)$	Quadratic	Nested loops
$O(2^n)$	Exponential	Naïve recursion

Rule: Always know the complexity of your algorithms



HEP Example: Particle ID Lookup (1/5)

ATLAS Use Case: Finding Z→ee Candidates

Problem: You have 10,000 electron candidates.
You need to find pairs with invariant mass near 91 GeV (Z boson).

Bad: Linear Scan — O(n²)

HEP Example: Particle ID Lookup (2/5)

```
// Naïve approach: Check every pair
std::vector<ZCandidate> findZCandidates(const std::vector<Electron>& electrons) {
    std::vector<ZCandidate> candidates;

    for (size_t i = 0; i < electrons.size(); ++i) {
        for (size_t j = i+1; j < electrons.size(); ++j) {
            float mass = invariantMass(electrons[i], electrons[j]);
            if (std::abs(mass - 91.2) < 10.0) {
                candidates.push_back({electrons[i], electrons[j]});
            }
        }
    }
    return candidates;
}
// For 10,000 electrons: 50 million comparisons! Takes ~5 seconds
```

Reality check: ATLAS has millions of events. This is unusable.

HEP Example: Particle ID Lookup (3/5)

Good: Sorted + Binary Search — O(n log n)

```
// Smart approach: Sort by pT, then search
std::vector<ZCandidate> findZCandidates(std::vector<Electron> electrons) {
    // Sort once: O(n log n)
    std::sort(electrons.begin(), electrons.end(),
              [](const Electron& a, const Electron& b) {
                  return a.pt() > b.pt();
              });

    std::vector<ZCandidate> candidates;
    // (continued on next slide)
```

HEP Example: Particle ID Lookup (4/5)

Good: Sorted + Binary Search — O(n log n)

```
    // (continued from previous slide)
    // For each electron, binary search for compatible partners
    for (size_t i = 0; i < electrons.size(); ++i) {
        auto range = findCompatibleRange(electrons, i);
        for (auto j = range.first; j != range.second; ++j) {
            // ... check invariant mass
        }
    }
    return candidates;
}
// For 10,000 electrons: ~100,000 comparisons. Takes ~0.1 seconds
// 50× faster!
```

HEP Example: Particle ID Lookup (5/5)

Best: Hash Map — O(1)

Note: When the problem allows keyed lookup (e.g., finding a particle by PDG ID), hash maps are ideal:

```
// Build index once
std::unordered_map<int, size_t> pdgIndex;
for (size_t i = 0; i < particles.size(); ++i) {
    pdgIndex[particles[i].pdgID()] = i;
}

// Lookup is constant time
int findParticleID(int pdgID) {
    auto it = pdgIndex.find(pdgID);
    return (it ≠ pdgIndex.end()) ? it→second : -1;
}
// For 10,000 particles: ~1 comparison (amortised)
```

The Memory Wall

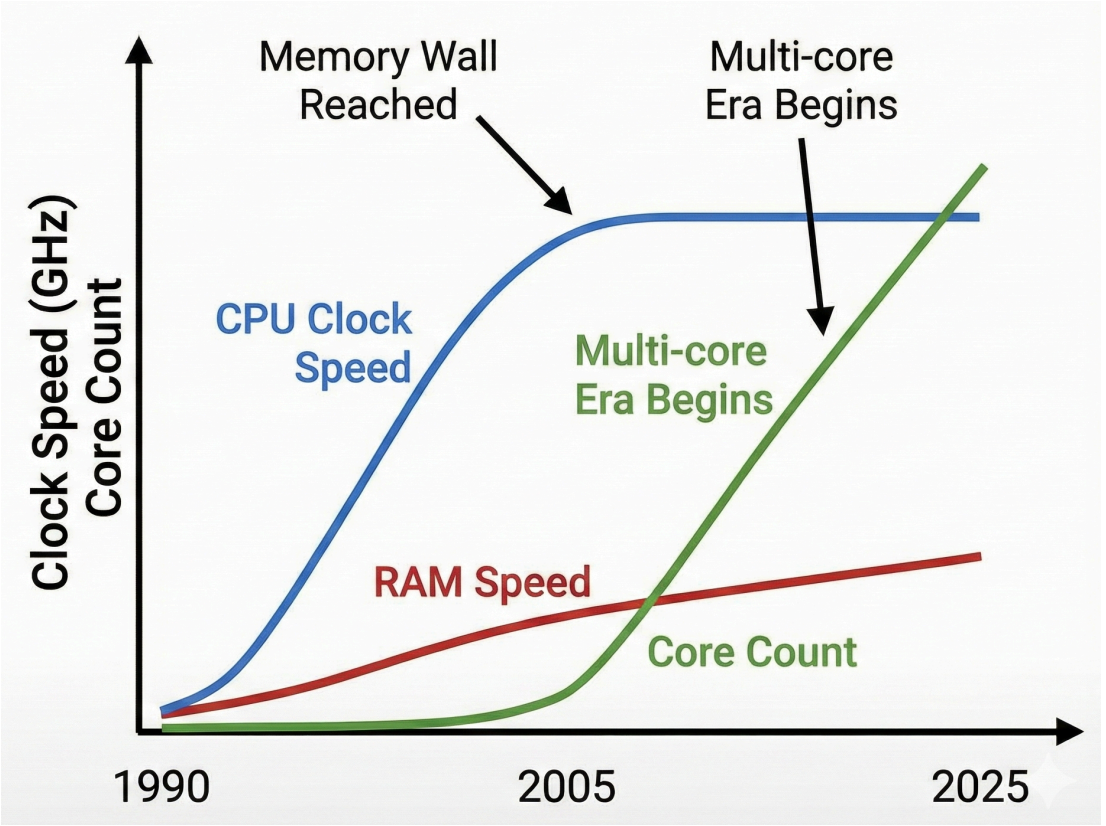
Hardware Timeline

Golden Age (1980–2005):

- 1990: 486 @ 50 MHz
- 2005: Pentium 4 @ 3.8 GHz
- CPU: 76× faster | RAM: 4× faster

Modern Era (2005–now):

- 2025: Server CPU @ 3.5 GHz (up to 128 cores)
- Clock speed: barely changed
- Solution: More cores, not faster cores



CPUs Are Fast, Memory Is Slow

Modern CPU (2026):

- L1 cache: ~4 cycles, 32–64 KB
- L2 cache: ~10–12 cycles, 256 KB–1 MB
- L3 cache: ~40–50 cycles, 8–64 MB
- Main RAM: ~200+ cycles

Gap: large!

Your algorithm might spend most of its time waiting for data.

ATLAS track reconstruction is often memory-bound in practice.

Cache Lines and Locality (1/2)

How Caches Work

Cache line: typically 64 bytes on x86-64. CPU fetches memory in chunks.

When you access `array[0]`, CPU loads `array[0..15]` (if int = 4 bytes)

Sequential access (fast):

```
for (int i = 0; i < n; ++i) {
    sum += array[i];  // ✅ Cache hits
}
```

Cache Lines and Locality (2/2)

How Caches Work

Random access (slow):

```
for (int i = 0; i < n; ++i) {
    sum += array[randomIndex[i]];  // ❌ Cache misses
}
```

Sequential access can be an order of magnitude faster than random — often 10–20× on modern CPUs!

Structure of Arrays vs Array of Structures

The Critical Data Layout Choice

AoS (Array of Structures):

```
struct Particle {
    float px, py, pz;
    float energy;
    int charge;
};

std::vector<Particle> particles;
```

Memory layout:

```
[px py pz E c][px py pz E c]...
```

SoA (Structure of Arrays):

```
struct Particles {
    std::vector<float> px;
    std::vector<float> py;
    std::vector<float> pz;
    std::vector<float> energy;
    std::vector<int> charge;
};
```

Memory layout:

```
[px px px...][py py py...]...
```

Why SoA Matters

Example benchmark (memory-bound loop)

Task: Calculate $pT = \sqrt{(px^2 + py^2)}$ for 1M particles

```
// AoS: loads 20 bytes, uses 8
for (auto& p : particles) {
    pT = sqrt(p.px*p.px + p.py*p.py);
}
// Time: 45 ms

// SoA: loads only what's needed + vectorises
for (size_t i = 0; i < n; ++i) {
    pT = sqrt(px[i]*px[i] + py[i]*py[i]);
}
// Time: 6 ms
```

Result: ~7.5× speedup just from data layout in this scenario.

Measuring Performance Safely

Pitfalls to Avoid

Common mistakes:

1. **Frequency scaling:** CPU clock changes dynamically (Turbo Boost)
2. **Cold caches:** First run is always slower
3. **Background processes:** Other programs interfere
4. **Optimiser tricks:** Compiler might optimise away your benchmark
5. **Small inputs:** Doesn't represent real workload

Solution: Use proper benchmarking tools

Google Benchmark

```cpp
#include <benchmark/benchmark.h>

static void BM_PtCalc_AoS(benchmark::State& state) {
    auto particles = generateData(10000);
    for (auto _ : state) {
        float sum = 0;
        for (const auto& p : particles)
            sum += std::sqrt(p.px*p.px + p.py*p.py);
        benchmark::DoNotOptimize(sum);
    }
}
BENCHMARK(BM_PtCalc_AoS);
BENCHMARK_MAIN();
```

Compile: `g++ -O3 -lbenchmark bench.cpp`

Using `perf` for System-Level Profiling

Hardware Performance Counters

```
# Measure cache misses
$ perf stat -e cache-references,cache-misses,cycles,instructions \
    ./reconstruction input.root

Performance counter stats:
    234,567,890  cache-references
     45,678,901  cache-misses      # 19.5% of all cache refs
  1,234,567,890  cycles
  2,345,678,901  instructions      # 1.90  insns per cycle

    2.345678 seconds time elapsed
```

- High cache miss rate (>10%) → memory-bound
- Low instructions per cycle (<1.5) → CPU stalls

Modern HEP: ROOT RDataFrame

Writing Thread-Safe Analysis Code

Traditional ROOT (single-threaded):

```
TFile* file = TFile::Open("data.root");
TTree* tree = (TTree*)file->Get("Events");

float pt;
tree->SetBranchAddress("pt", &pt);
TH1F* h = new TH1F("h", "pT distribution", 100, 0, 200);

for (Long64_t i = 0; i < tree->GetEntries(); ++i) {
    tree->GetEntry(i);
    if (pt > 20.0) h->Fill(pt);
}
```

Problem: Sequential processing — can't use multiple cores

ROOT RDataFrame + IMT

```cpp
#include <ROOT/RDataFrame.hxx>

// Enable multithreading BEFORE creating RDataFrame
ROOT::EnableImplicitMT();

ROOT::RDataFrame df("Events", "data.root");
auto h = df.Filter("pt > 20.0")
           .Histo1D({"h", "pT", 100, 0, 200}, "pt");
h->Draw();
```

Result: Often 2–8× speedup on modern CPUs (I/O can limit).

Demo: RDataFrame Speedup

```
// Single-threaded
ROOT::DisableImplicitMT();
benchmark();  // Time: 8.2s

// Multi-threaded (8 cores)
ROOT::EnableImplicitMT(8);
benchmark();  // Time: 1.4s → 5.9× speedup!
```

Typical results:

Threads	Time	Speedup
1	8.2s	1.0×
4	2.3s	3.6×
8	1.4s	5.9×

One line of code — no manual thread management needed!

Summary: T&T L3

Key Takeaways:

1. **Algorithm choice matters most** — O(n) vs O(log n) vs O(1)
2. **Memory is the bottleneck** — Cache-friendly code is fast code
3. **SoA > AoS for HEP** — Better cache usage and vectorisation
4. **Measure correctly** — Use Google Benchmark and perf
5. **ROOT RDataFrame** — Easy thread safety with implicit MT

Next: Hands-on exercises to apply these concepts

TOOLS & TECHNIQUES

Exercises: E1 + E2

"Learning by doing — the only way it sticks"

2 hours total
Pairs, hands-on, real HEP data

Debug broken code with sanitisers
Optimise real ATLAS Open Data I/O

Exercise E1: Make It Right

Debugging with Sanitisers (60 minutes)

Scenario: You've inherited a "broken" track reconstruction library. It compiles but produces incorrect results and occasionally crashes.

Your mission:

1. Use **ASan** to find memory bugs
2. Write **Catch2 tests** to prevent regression
3. Fix the bugs
4. Ensure all tests pass in CI

Files provided: `reconstruction.cpp`, `reconstruction.h`, `test_reconstruction.cpp`, `CMakeLists.txt`

E1: Getting Started

Setup

```
# Clone the exercise repository
git clone https://github.com/artfisica/csc2026.git
cd exercise-e1

# Create development container (optional but recommended)
devcontainer open .

# Build with ASan
cmake -B build -S . \
    -DCMAKE_BUILD_TYPE=Debug \
    -DCMAKE_CXX_FLAGS="-fsanitize=address -g"
cmake --build build

# Run tests
cd build && ctest --output-on-failure
```

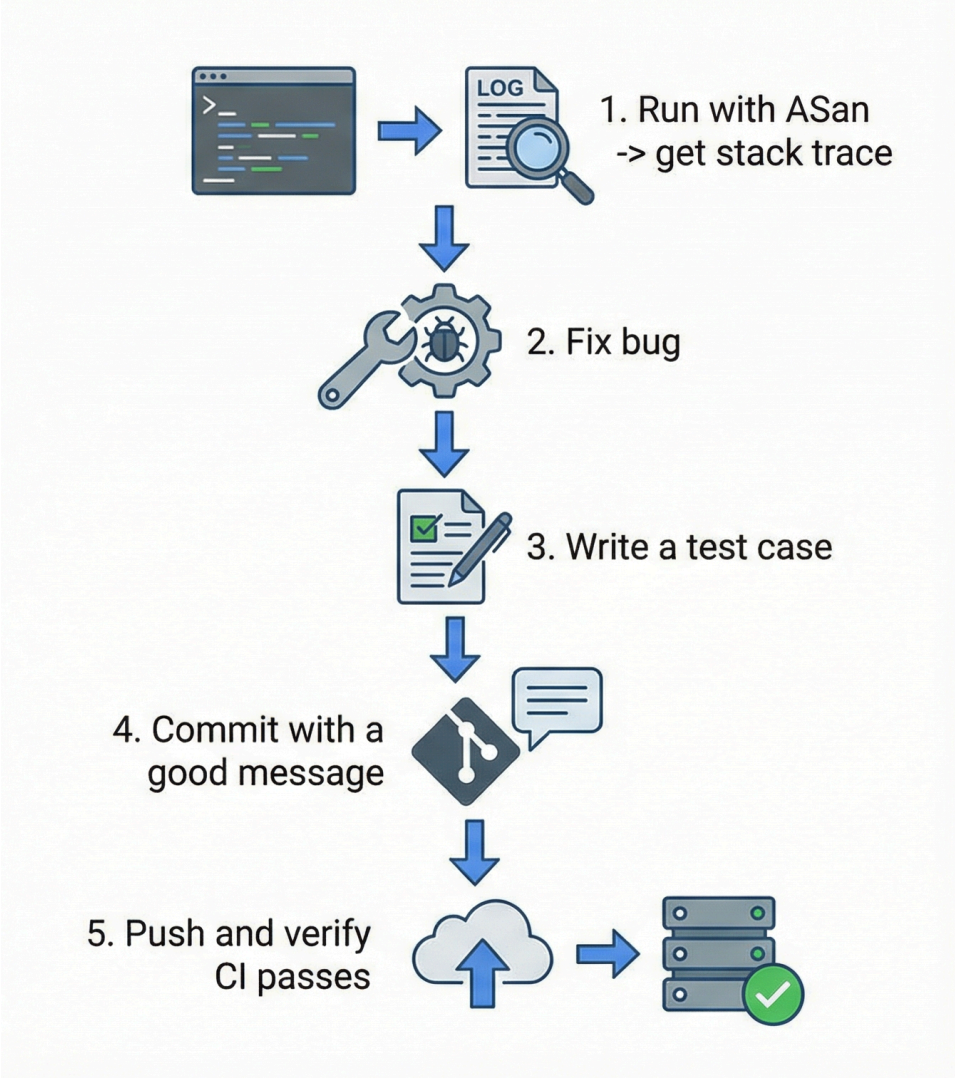
E1: Expected Bugs

You'll encounter:

1. **Buffer overflow:** Array access out of bounds
2. **Use-after-free:** Accessing deleted memory
3. **Memory leak:** Forgetting to deallocate

Workflow:

1. Run with ASan → get stack trace
2. Fix bug
3. Write a test case that would catch it
4. Commit with a good message
5. Push and verify CI passes



Exercise E2: Make It Fast Enough (1/2)

Performance Optimisation (60 minutes)

Two tracks available

Track A (Algorithmic): Optimise a particle sorting/search routine

- Profile with `perf`
- Identify hotspots
- Apply algorithmic improvements
- Demonstrate speedup

Exercise E2: Make It Fast Enough (2/2)

Track B (Real I/O with ATLAS Open Data): Optimise ROOT file reading

- Profile I/O throughput
- Optimise branch reading
- Test RDataFrame
- Generate CI artifact showing MB/s improvement

E2 Track A: Algorithmic Optimisation

The Challenge

File: `particle_search.cpp`

Current implementation: Naïve $O(n^2)$ algorithm for finding particle pairs

Your tasks:

1. Profile with Google Benchmark
2. Identify the complexity bottleneck
3. Implement O(n log n) solution
4. Verify correctness with tests
5. Document speedup in CI artifact

Hint: Think about sorting + binary search, or spatial indexing

E2 Track B: ROOT I/O Optimisation

The Challenge

Files: Real ATLAS **Open Data (13 TeV)** — suitable for teaching derived from Run formats **(the Higgs discovery in 2012 used 7–8 TeV data)**

Dataset: NanoAOD-style ROOT files with electron, muon, jet collections (~100 MB each)

Current code: Reads all branches, slow iteration

Your mission:

1. **Baseline:** Measure current I/O throughput (MB/s)
2. **Optimise:** Use `SetBranchStatus` to disable unneeded branches
3. **Optimise:** Adjust basket size/cache settings
4. **Modern approach:** Try RDataFrame
5. **Deliverable:** CI artifact with before/after throughput comparison

E2 Track B: Getting Started

```
// Baseline measurement
void baseline() {
    TFile* file = TFile::Open("data.root");
    TTree* tree = (TTree*)file->Get("Events");
    // Read all branches (slow)
    tree->SetBranchStatus("*", 1);

    Long64_t nEntries = tree->GetEntries();
    auto start = std::chrono::high_resolution_clock::now();

    for (Long64_t i = 0; i < nEntries; ++i) {
        tree->GetEntry(i);
        // Process ...
    }

    auto end = std::chrono::high_resolution_clock::now();
    // Calculate MB/s ...
}
```

E2 Track B: Optimisation Strategies (1/2)

1. Selective Branch Reading

```
// Only read branches you need
tree->SetBranchStatus("*", 0);           // Disable all
tree->SetBranchStatus("Electron_pt", 1); // Enable specific
tree->SetBranchStatus("Electron_eta", 1);
tree->SetBranchStatus("Electron_phi", 1);
```

2. Cache Tuning

```
// Increase cache size (default is often too small)
tree->SetCacheSize(50 * 1024 * 1024); // 50 MB
tree->AddBranchToCache("Electron_*", kTRUE);
```

E2 Track B: Optimisation Strategies (2/2)

3. RDataFrame (Best Practice)

```
ROOT::RDataFrame df("Events", "data.root");
auto filtered = df.Filter("Electron_pt[0] > 25");
```

Both Tracks: CI Integration

Your CI Should:

1. **Build** with optimisations enabled (`-O3`)
2. **Run benchmarks** automatically
3. **Generate comparison** (before/after)
4. **Post artifact** (plot or table)

Example `.gitlab-ci.yml` addition:

```yaml
benchmark:
  stage: test
  script:
    - ./build/benchmark_baseline > baseline.txt
    - ./build/benchmark_optimised > optimised.txt
    - python scripts/compare.py baseline.txt optimised.txt
  artifacts:
    paths:
      - comparison.png
      - improvement_report.txt
```

Exercise Tips & Getting Help

Working Together

- Work in pairs if you can
- Talk through your approach
- Commit often with good messages
- Test as you go
- Ask for help, seriously
- Document what you tried

When stuck (totally normal!):

1. Talk to partner first
2. Read error messages
3. Raise your hand
4. Ask the group
5. Use the tools (ASan/TSan/perf)

There are no stupid questions! Keep notes of what worked.

Time Management

E1: "Make it Right" (~60 min)

- 20 min: Find bugs with ASan
- 20 min: Write tests
- 15 min: Fix bugs
- 5 min: CI verification

E2: "Make it Fast" (~60 min)

- 15 min: Baseline
- 20 min: Find bottleneck
- 20 min: Optimise
- 5 min: Verify

Tips

- Get it working, then improve.
- Prioritise correctness first

SOFTWARE DESIGN

Lecture 1: Many-Core Foundations

"The free lunch is over — welcome to the parallel era"

60 minutes

Understand why and how to write parallel code

Power wall, Amdahl's Law, OpenMP, TBB

SD L1: Many-Core Foundations

The Hardware Reality

2005: The Industry's Biggest Pivot

Before: "Wait 18 months, get 2× faster CPU"
After: "Here are more cores. Figure it out."

What happened:

- Intel/AMD cancel 10 GHz plans
- Power ≈ C × V² × f (raising f often requires higher V → superlinear power growth)
- Result: Clock speeds stuck at ~3–5 GHz

Everything about "faster computers = faster code" stopped being true.

Today's Landscape

Consumer: 8–32 cores typical
HEP (WLCG): 32–128 cores per node
Total WLCG: ~1 million cores worldwide

- https://home.cern/science/computing/grid

The challenge: Your code must use all cores or waste 90%+ compute!

The Power Wall

Why Clock Speeds Stopped Growing

Power consumption: $P \approx C \times V^2 \times f$

Where:

- C = capacitance
- V = voltage
- f = frequency (clock speed)

Problem: Power grows with frequency. Higher clocks → more heat → can't cool

Solution: More cores at lower frequency instead of fewer cores at higher frequency

Implication: To use modern hardware, **we must parallelise**

Amdahl's Law

The Serial Bottleneck

Formula: Speedup = 1 / [(1 - P) + P/N] | Where P = parallel fraction, N = cores

Cores	Speedup	Time	Efficiency
1	1.0×	100s	100%
2	1.75×	57s	88%
4	2.96×	34s	74%
8	4.21×	24s	53%
∞	6.67×	15s	0%

ATLAS Example: 100s/event total:

- I/O: 15s (serial);
- Track/vertex: 85s (parallel);
- P = 0.85

Lesson: 15% serial code limits max speedup to 6.67×!

Gustafson's Law

Scale the Problem, Not the Solution

Alternative view: As hardware improves, we tackle larger problems

Formula: Speedup = N - (1 - P) × (N - 1)

Where P is a parallel fraction, N is the number of cores

Key insight: If you can scale problem size, parallel efficiency improves

HEP context: We don't process the same 1M events faster — we process 1B events

Concurrency vs Parallelism

Important Distinction

Concurrency

- About **structure**
- Multiple tasks in progress
- Doesn't require multiple cores
- Example: Event-driven I/O

Parallelism

- About **execution**
- Multiple tasks running simultaneously
- Requires multiple cores
- Example: Multi-threaded processing

In HEP: We need both

- Concurrency: Handle multiple data streams
- Parallelism: Process events on multiple cores

Parallel Programming: Rules of Thumb

✅ Good candidates:

- Loop with >10k independent items
- Each iteration >1 microsecond
- Zero dependencies between iterations
- Contiguous arrays (not lists/trees)

❌ Bad candidates:

- <1k items (overhead > benefit)
- Heavy synchronisation needed
- Rarely executed code

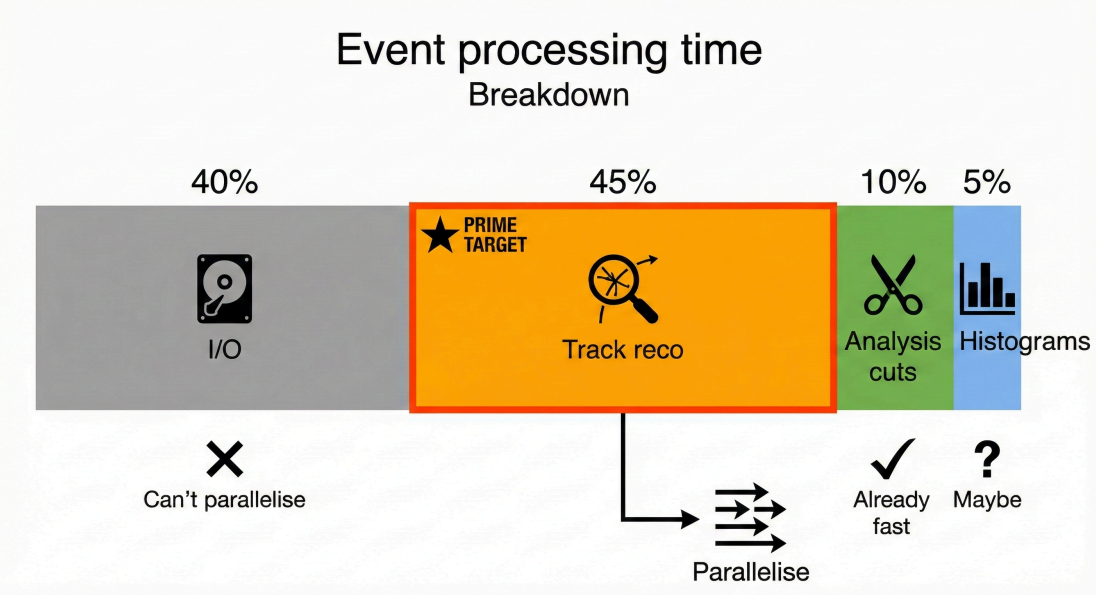
80/20 rule: Profile first! 80% of time in 20% of code.

Real LHC experiment Breakdown

Event processing time:

- 40% I/O → Can't parallelise
- 45% Track reco → **PRIME TARGET**
- 10% Analysis cuts → Already fast
- 5% Histograms → Maybe

Focus: Parallelise the 45%



C++ Standard Library Threading

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

void processEvent(int eventID) {
    // Heavy computation
}
int main() {
    std::vector<std::thread> workers;
    for (int i = 0; i < 8; ++i) {
        workers.emplace_back(processEvent, i);
    }
    for (auto& t : workers) {
        t.join(); // Wait for completion
    }
}
```

Problem: Manual thread management is error-prone

Atomics: Lock-Free Synchronisation (1/2)

Safe Counter Without Locks

```
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> eventCount{0};

void processEvents(int n) {
    for (int i = 0; i < n; ++i) {
        // Do work...
        eventCount.fetch_add(1, std::memory_order_relaxed);
    }
}
```

Atomics: Lock-Free Synchronisation (2/2)

Safe Counter Without Locks

```
int main() {
    std::vector<std::thread> workers;
    for (int i = 0; i < 8; ++i) {
        workers.emplace_back(processEvents, 1000);
    }

    for (auto& t : workers) t.join();
    std::cout << "Total events: " << eventCount << std::endl;
}
```

Mutexes: When You Must Synchronise (1/2)

Protecting Shared State

```
#include <mutex>
#include <map>
#include <thread>

std::map<int, Result> resultsCache;
std::mutex cacheMutex;

void storeResult(int eventID, const Result& result) {
    std::lock_guard<std::mutex> lock(cacheMutex);
    resultsCache[eventID] = result; // Protected
}
// (continued on next slide)
```

Mutexes: When You Must Synchronise (2/2)

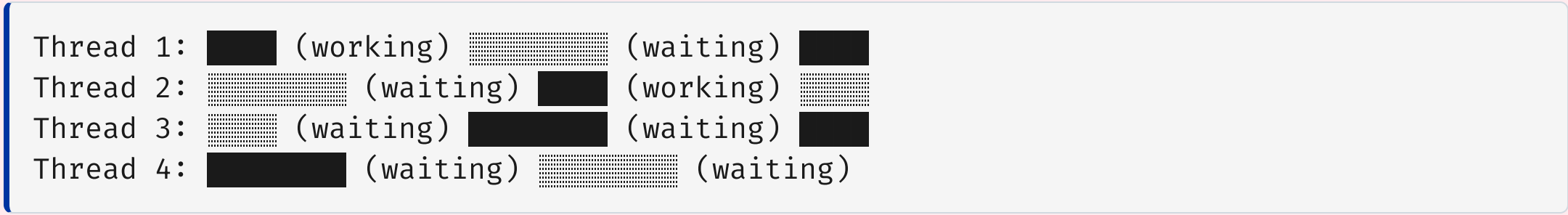
Protecting Shared State

```
// (continued from previous slide)
Result loadResult(int eventID) {
    std::lock_guard<std::mutex> lock(cacheMutex);
    return resultsCache.at(eventID);
}
```

Warning: Mutexes are slow. Avoid when possible. Use lock-free algorithms or partition data instead.

Why Avoid Mutexes? | Contention Kills Performance

Problem: Multiple threads waiting for the same lock



Result: Threads spend more time waiting than working

Better approaches:

1. **Partition data** — Give each thread independent data
2. **Lock-free structures** — Use atomics or lock-free queues
3. **Thread-local storage** — Each thread has own copy

OpenMP: Easy Parallelism

```
#include <omp.h>

void parallelProcess(std::vector<Event>& events) {
    #pragma omp parallel for
    for (size_t i = 0; i < events.size(); ++i) {
        events[i].reconstruct();
    }
}
```

Compile: `g++ -fopenmp program.cpp`

Control threads: `OMP_NUM_THREADS=8 ./program`

OpenMP: Reductions

Parallel Aggregation

```
double sumEnergy(const std::vector<Particle>& particles) {
    double total = 0.0;

    #pragma omp parallel for reduction(+:total)
    for (size_t i = 0; i < particles.size(); ++i) {
        total += particles[i].energy();
    }

    return total;
}
```

How it works: Each thread maintains a local sum, then combines at the end

Supported operations: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, `min`, `max`

oneTBB: Task-Based Parallelism

```
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

void process(std::vector<Event>& events) {
    tbb::parallel_for(
        tbb::blocked_range<size_t>(0, events.size()),
        [&](const auto& r) {
            for (size_t i = r.begin(); i ≠ r.end(); ++i)
                events[i].reconstruct();
        }
    );
}
```

Advantages: Work stealing, composable, modern C++ API

TBB: Parallel Reduction

```
#include <tbb/parallel_reduce.h>

double sumEnergy(const std::vector<Particle>& p) {
    return tbb::parallel_reduce(
        tbb::blocked_range<size_t>(0, p.size()),
        0.0,
        [&](const auto& r, double s) {
            for (size_t i = r.begin(); i ≠ r.end(); ++i)
                s += p[i].energy();
            return s;
        },
        std::plus<double>()
    );
}
```

HEP Context: Where Threading Is Used (1/2)

CMSSW (CMS Software)

- **Framework:** TBB-based task scheduling
- **Event-level parallelism:** Multiple events processed concurrently
- **Module-level parallelism:** Independent modules run in parallel

Geant4 (Detector Simulation)

- **Multi-threading mode:** Multiple events simulated simultaneously
- **Thread-local geometry:** Each thread has navigation state
- **Shared read-only data:** Geometry, physics processes

HEP Context: Where Threading Is Used (2/2)

ROOT

- **RDataFrame:** Implicit multithreading
- **TTree::GetEntry:** Can be parallelised with care
- **Histogram filling:** Thread-safe with proper setup

Common Threading Pitfalls (1/2)

1. Race Conditions

```
int counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; ++i) {
    counter++; // WRONG, Multiple threads writing
}
// Solution: Use reduction or atomic
```

2. False Sharing

```
struct Data { int count; }; // Only 4 bytes
Data perThread[8]; // Each thread's data
// Problem: All in the same cache line → cache ping-pong!
// Solution: struct Data { int count; char padding[60]; };
```

Common Threading Pitfalls (2/2)

3. Deadlocks

```
// Thread 1: lock(A), then lock(B)
// Thread 2: lock(B), then lock(A)  → DEADLOCK!
// Fix: Always acquire locks in the same order
```

Parallel Debugging Checklist

When parallel code fails:

1. ☐ Does it work with 1 thread? (If no → not a parallel bug)
2. ☐ Run with TSan
3. ☐ Check false sharing (variables <64 bytes apart?)
4. ☐ Verify reduction correctness
5. ☐ Check load balance
6. ☐ Inspect scaling (2× cores ≈ 2× speedup?)

90% of bugs caught by TSan + this list!

Summary: SD L1

Key Takeaways:

1. **Power wall** — Clock speeds stopped; we must go wide
2. **Amdahl's Law** — Serial sections limit speedup
3. **Primitives** — Threads, atomics, mutexes (avoid)
4. **OpenMP** — Easy pragma-based parallelism
5. **TBB** — Task-based, composable, better load balancing
6. **HEP frameworks** — CMSSW, Geant4, ROOT all use threading

Next: Data layout and parallel patterns

SOFTWARE DESIGN

Lecture 2: Patterns & Data Layout

"How you organise your data matters more than your algorithm"

60 minutes

Master parallel patterns and cache-friendly programming

Map/Reduce, SoA vs AoS, vectorisation, false sharing

SD L2: Patterns & Data Layout

Why Patterns Matter

Common parallel problems have common solutions

Instead of reinventing wheels:

1. Learn proven patterns
2. Recognise which pattern fits your problem
3. Apply the pattern correctly
4. Avoid common pitfalls

Today's patterns:

- Map/Reduce
- Pipeline
- Tiling

Map/Reduce Pattern (1/2)

The Most Common Parallel Pattern

Map: Apply function to each element independently

```
// Sequential
for (auto& x : data) {
    x = transform(x);
}

// Parallel (map)
#pragma omp parallel for
for (size_t i = 0; i < data.size(); ++i) {
    data[i] = transform(data[i]);
}
```

Map/Reduce Pattern (2/2)

The Most Common Parallel Pattern

Reduce: Combine results into a single value

```
// Parallel (reduce)
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (size_t i = 0; i < data.size(); ++i) {
    sum += data[i];
}
```

Map/Reduce in HEP (1/2)

Event Processing

```
// Map: Process each event independently
std::vector<Result> results(events.size());

tbb::parallel_for(
    tbb::blocked_range<size_t>(0, events.size()),
    [&](const tbb::blocked_range<size_t>& r) {
        for (size_t i = r.begin(); i ≠ r.end(); ++i) {
            results[i] = processEvent(events[i]);
        }
    }
);
```

Map/Reduce in HEP (2/2)

Event Processing

```
// Reduce: Count events passing selection
int passing = tbb::parallel_reduce(
    tbb::blocked_range<size_t>(0, results.size()),
    0,
    [&](const tbb::blocked_range<size_t>& r, int count) {
        for (size_t i = r.begin(); i ≠ r.end(); ++i) {
            if (results[i].passes()) count++;
        }
        return count;
    },
    std::plus<int>()
);
```

Pipeline Pattern

Stages of Processing

Idea: Divide work into sequential stages, parallelise within stages

```
Input → [Stage 1] → [Stage 2] → [Stage 3] → Output
         ↓ ↓ ↓       ↓ ↓ ↓       ↓ ↓ ↓
        parallel    parallel    parallel
```

HEP Example:

1. **Stage 1:** Read events from file
2. **Stage 2:** Reconstruct tracks/clusters (parallel)
3. **Stage 3:** Apply physics selection (parallel)
4. **Stage 4:** Fill histograms (need synchronisation)

Pipeline with TBB (1/2)

```
#include <tbb/pipeline.h>

void processPipeline(const std::string& inputFile) {
    tbb::parallel_pipeline(
        /*max_tokens=*/16,
        tbb::make_filter<void, Event*>(
            tbb::filter_mode::serial_in_order,
            [&](tbb::flow_control& fc) -> Event* {
                Event* evt = readNextEvent(inputFile);
                if (!evt) fc.stop();
                return evt;
            }
        ) &
    // (continued on next slide)
```

Pipeline with TBB (2/2)

```
    // (continued from previous slide)
        tbb::make_filter<Event*, Event*>(
            tbb::filter_mode::parallel,
            [](Event* evt) {
                evt->reconstruct();
                return evt;
            }
        ) &
        tbb::make_filter<Event*, void>(
            tbb::filter_mode::parallel,
            [](Event* evt) {
                evt->select();
                delete evt;
            }
        )
    );
}
```

Tiling Pattern

Block-Based Processing

Idea: Divide the problem into tiles/blocks that fit in the cache

T1	T2	T3
T4	T5	T6
T7	T8	T9

```
Process one tile at a time
→ Better cache reuse
→ Improved locality
```

Applications:

- Matrix operations & Image processing
- Grid-based simulations

Tiling Example: Matrix Multiplication (1/2)

```
// Naïve (cache-unfriendly)
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Tiling Example: Matrix Multiplication (2/2)

```
// Tiled (cache-friendly)
const int TILE = 64;
for (int ii = 0; ii < N; ii += TILE) {
    for (int jj = 0; jj < N; jj += TILE) {
        for (int kk = 0; kk < N; kk += TILE) {
            for (int i = ii; i < std::min(ii+TILE, N); ++i) {
                for (int j = jj; j < std::min(jj+TILE, N); ++j) {
                    for (int k = kk; k < std::min(kk+TILE, N); ++k) {
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
```

Data Layout: AoS vs SoA Revisited

Why This Matters for Parallelism

Array of Structures (AoS):

```
struct Particle {
    float px, py, pz, E;
    int pdgID;
};
std::vector<Particle> particles;
```

Problems:

1. Cache line waste when accessing a single field
2. Cannot vectorise easily
3. False sharing between threads

Data Layout: SoA Benefits

Structure of Arrays (SoA):

```
struct Particles {
    std::vector<float> px, py, pz, E;
    std::vector<int> pdgID;
    size_t size() const { return px.size(); }
};
```

Benefits:

1. ✅ Contiguous memory for each field
2. ✅ Auto-vectorisation by compiler
3. ✅ No false sharing (different cache lines)
4. ✅ Better cache utilisation

Trade-off: More complex indexing, but worth it

False Sharing Example

The Hidden Performance Killer

```
// Bad: False sharing
struct ThreadData {
    int counter; // Only 4 bytes
};
ThreadData data[8]; // Array of per-thread data

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    for (int i = 0; i < 1000000; ++i) {
        data[tid].counter++; // Looks independent, but ...
    }
}
```

Problem: `counter`s share cache lines. Writing from one thread invalidates the cache for others.

False Sharing: Solution

Padding to Cache Line Boundaries

```
// Good: Pad to cache line size (64 bytes)
struct ThreadData {
    int counter;
    char padding[60]; // Total: 64 bytes
};
ThreadData data[8]; // Now each on different cache line

// Or use C++17:
struct alignas(64) ThreadData {
    int counter;
};
```

Result: Each thread's data on separate cache line → no false sharing

Rule: Per-thread data should be at least 64 bytes apart.

Vectorisation (SIMD)

Single Instruction, Multiple Data

Modern CPUs: Execute the same operation on 4–16 values simultaneously

AVX-512 Example:

```
// Scalar (one at a time)
for (int i = 0; i < n; ++i) {    c[i] = a[i] + b[i];   }
// Vectorised (16 floats at once)
for (int i = 0; i < n; i += 16) {
    __m512 va = _mm512_loadu_ps(&a[i]);
    __m512 vb = _mm512_loadu_ps(&b[i]);
    __m512 vc = _mm512_add_ps(va, vb);
    _mm512_storeu_ps(&c[i], vc);
}
```

Speedup: Up to 16× for simple operations

Auto-Vectorisation (1/2)

Let the Compiler Do It

Most compilers auto-vectorise simple loops:

```
// This loop can be auto-vectorised
void add(float* a, float* b, float* c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Auto-Vectorisation (2/2)

Let the Compiler Do It

Compile with:

```
g++ -O3 -march=native -ftree-vectorize -fopt-info-vec
```

Requirements for auto-vectorisation:

- Contiguous arrays (not lists or trees)
- No dependencies between iterations
- Aligned data (64-byte alignment helps)

SoA Enables Vectorisation

```
// AoS: Hard to vectorise
struct Particle { float px, py, pz, E; };
std::vector<Particle> particles;

for (size_t i = 0; i < particles.size(); ++i) {
    particles[i].px *= scale; // Compiler struggles here
}

// SoA: Easy to vectorise
struct Particles {
    std::vector<float> px, py, pz, E;
};

for (size_t i = 0; i < particles.px.size(); ++i) {
    particles.px[i] *= scale; // Compiler auto-vectorises
}
```

HEP Framework Scheduling

Task-Based Parallelism in CMSSW

Traditional (Run 1–2):

- Sequential module execution
- One event at a time

Modern (Run 3+):

- Task graph of module dependencies
- Multiple events in flight
- Modules run when inputs are ready

```
Event 1: [Read] → [Unpack] → [Reco] → [Select] → [Write]
Event 2:          [Read] → [Unpack] → [Reco] → [Select]
Event 3:                   [Read] → [Unpack] → [Reco]
```

Key: Modules with no dependencies run in parallel

Summary: SD L2

Key Takeaways:

1. **Patterns** — Map/Reduce, Pipeline, Tiling
2. **Data layout** — SoA beats AoS for parallelism
3. **False sharing** — Pad per-thread data to 64 bytes
4. **Vectorisation** — SIMD gives 4–16× speedup
5. **HEP frameworks** — Task-based scheduling in CMSSW

Next: Profiling and optimisation workflow

SOFTWARE DESIGN

Lecture 3: Profiling to Optimisation

"In God we trust; all others must bring data"

60 minutes

Profile scientifically, optimise systematically

perf, flame graphs, roofline model, real case study

SD L3: Profiling to Optimisation

The Optimisation Workflow

Four-Step Process:

1. **Measure** — Profile to find hotspots
2. **Analyse** — Understand why it's slow
3. **Optimise** — Apply targeted improvements
4. **Verify** — Confirm speedup and correctness

Golden Rule: Never optimise without profiling first

Corollary: Profile after every change to verify improvement

Why Profile First?

Avoid Wasted Effort

Without profiling:

- Optimise the wrong code
- Spend days on 1% hotspot
- Make code complex for no gain
- Miss the real bottleneck

With profiling:

- Focus on 80% of runtime
- Quick wins are visible immediately
- Quantify every improvement
- Understand trade-offs

Example: Developer spends a week optimising a loop that takes 0.5% of runtime. Meanwhile, I/O takes 60%.

Profiling Tools

The Essential Toolkit

Tool	Purpose	Platform
perf	System-level CPU profiling	Linux
gprof	Function-level profiling	Linux, macOS
Valgrind/Callgrind	Detailed call graphs	Linux
Intel VTune	Advanced CPU/memory analysis	Intel CPUs
Google Benchmark	Micro-benchmarking	Cross-platform
ROOT TStopwatch	HEP-specific timing	Cross-platform

Today's focus: perf (most common in HEP)

Using perf: Basic Commands

CPU Profiling

```
# Record profile (samples every ~100ms)
$ perf record -g ./reconstruction input.root

# View results interactively
$ perf report

# Or generate text report
$ perf report --stdio > profile.txt

# Annotate specific function
$ perf annotate processEvent

# Compare two runs
$ perf diff before.data after.data
```

Key options:

- `-g`: Capture call graphs
- `-F 1000`: Sample at 1000 Hz
- `-e cycles`: Profile CPU cycles (default)

Reading perf Output

Example Report

```
# Overhead  Command   Shared Object      Symbol
# ........  ........  .................  ..............................
    45.23%  reco      libreco.so         [.] TrackFitter::fit()
    18.67%  reco      libreco.so         [.] ClusterBuilder::build()
    12.34%  reco      libRoot.so         [.] TTree::GetEntry()
     8.91%  reco      libreco.so         [.] ParticleID::classify()
     5.43%  reco      libc.so.6          [.] memcpy
     3.21%  reco      libreco.so         [.] EnergyCorrection::apply()
      ...
```

- 45% of time in `TrackFitter::fit()` → **Primary target**
- 18% in `ClusterBuilder::build()` → **Secondary target**
- 12% in ROOT I/O → Consider optimising file reading

Flame Graphs

Visualising Call Stacks

Tool: `FlameGraph` by Brendan Gregg

```
# Generate flame graph
$ perf record -F 99 -g ./reconstruction input.root
$ perf script | stackcollapse-perf.pl | flamegraph.pl > flame.svg
$ firefox flame.svg
```

How to read:

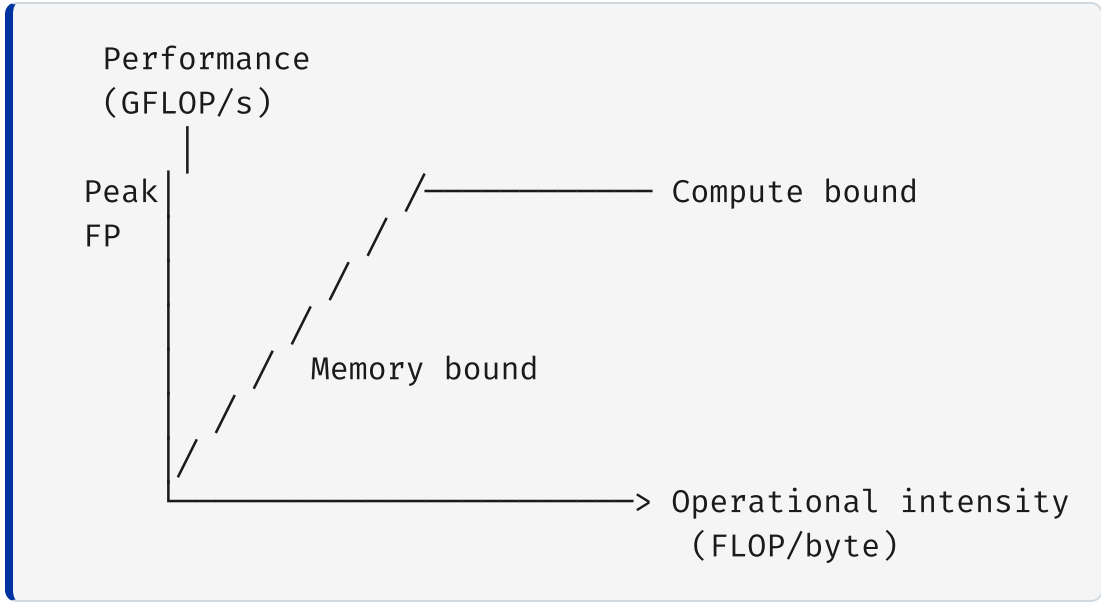
- Width = % of samples
- Stack depth = call hierarchy
- Click to zoom
- Search for function names

Example: Wide plateau at bottom = hot function consuming most time

Roofline Model

Are We Compute-Bound or Memory-Bound?

Roofline plot:



Interpretation:

- **Left of "roof":** Memory-bound → improve data layout
- **Right of "roof":** Compute-bound → algorithmic optimisation

Measuring Operational Intensity

FLOP/byte Ratio

```
# Use perf to count operations
$ perf stat -e fp_arith_inst_retired.scalar_single,\\
              fp_arith_inst_retired.128b_packed_double,\\
              cache-references,cache-misses \\
    ./program

# Calculate:
# FLOP/byte = (Total FP operations) / (Bytes accessed from memory)
```

HEP typical values:

- Track fitting: ~10–50 FLOP/byte (compute-bound)
- Histogram filling: ~0.1–1 FLOP/byte (memory-bound)
- ROOT I/O: ~0.01 FLOP/byte (I/O-bound)

Case Study: Optimising a Physics Kernel

Starting Point

Code: Track parameter calculation

```
// Original implementation
void calculateParameters(const std::vector<Hit>& hits, TrackParams& params) {
    for (const auto& hit : hits) {
        Matrix3×3 cov = hit.covariance();
        Vector3 pos = hit.position();

        // Matrix operations...
        Matrix3×3 invCov = cov.inverse();
        params.accumulate(pos, invCov);
    }
}
```

Profile shows: 60% of time in this function

Case Study: Step 1 — Measure

```
$ perf record -g ./tracking benchmark.root
$ perf report

Overhead  Symbol
  60.2%   calculateParameters
  15.3%   Matrix3×3::inverse
  12.1%   Hit::covariance
   8.7%   Vector3::operator*
```

Analysis:

- Main hotspot confirmed: `calculateParameters`
- Secondary hotspot: `Matrix3×3::inverse` (called inside loop)
- `Hit::covariance` is also expensive

Case Study: Step 2 — Analyse

Digging deeper with `perf annotate`:

```
$ perf annotate calculateParameters

  calculateParameters():
    45.23%: mov    (%rax), %xmm0       # Loading from memory (slow!)
    12.34%: call   Matrix3×3::inverse # Function call overhead
     8.91%: movsd  %xmm0, (%rcx)
      ...
```

Findings:

1. Many cache misses (hits not contiguous)
2. Matrix inverse called every iteration (expensive)
3. Function call overhead

Case Study: Step 3 — Optimise

Improvement 1: Precompute Invariants

```
void calculateParameters(const std::vector<Hit>& hits, TrackParams& params) {
    // Precompute inverse covariances once
    std::vector<Matrix3×3> invCovs;
    invCovs.reserve(hits.size());

    for (const auto& hit : hits) {
        invCovs.push_back(hit.covariance().inverse());
    }
    // Now accumulate (no more inversions)
    for (size_t i = 0; i < hits.size(); ++i) {
        params.accumulate(hits[i].position(), invCovs[i]);
    }
}
```

Speedup: 1.8× (reduced inversions)

Case Study: Step 3 — Optimise

Improvement 2: SoA Data Layout

```
// Before: AoS
struct Hit {
    Vector3 position;
    Matrix3×3 covariance;
};
std::vector<Hit> hits;
// After: SoA
struct HitCollection {
    std::vector<float> x, y, z;
    std::vector<Matrix3×3> covariance;
};
```

Result: Better cache utilisation, vectorisation possible

Speedup: Additional 1.4× → **Total: 2.5×** vs original

Case Study: Step 4 — Verify

```
# Before optimisation
$ perf stat ./tracking benchmark.root
 Performance counter stats:
   10,234,567,890  cycles
    5,123,456,789  instructions      # 0.50 insns per cycle
      234,567,890  cache-misses
       5.234 seconds time elapsed

# After optimisation
$ perf stat ./tracking benchmark.root
 Performance counter stats:
   4,123,456,789  cycles              # 2.5× faster
   6,234,567,890  instructions        # 1.51 insns per cycle
     98,765,432  cache-misses         # 2.4× fewer misses
       2.089 seconds time elapsed
```

Optimisation Priorities

Focus Your Efforts

High Impact:

1. Algorithm choice
2. Data layout (AoS → SoA)
3. Removing allocations
4. Cache optimisation
5. Parallelisation

Low Impact:

1. Variable naming
2. Inline hints (compiler knows)
3. Micro-optimisations
4. Premature vectorisation
5. Obscure tricks

Rule: 80/20 principle applies. First 20% effort gives 80% speedup.

Common Optimisation Techniques (1/2)

Quick Wins

1. Reserve vector capacity

```
std::vector<Track> tracks;
tracks.reserve(estimatedSize); // Avoid reallocations
```

2. Pass by const reference

```
void process(const LargeObject& obj); // Not by value
```

Common Optimisation Techniques (2/2)

Quick Wins

3. Use `emplace_back` not `push_back`

```
tracks.emplace_back(args ... ); // Construct in-place
```

4. Avoid unnecessary copies

```
auto result = compute(); // Move semantics
```

Advanced Optimisation: Loop Unrolling

Help the Compiler Vectorise

```
// Before (compiler may not unroll)
for (size_t i = 0; i < n; ++i) {  result[i] = a[i] * b[i] + c[i];  }
// After (manual unroll by 4)
size_t i = 0;
for (; i + 4 ⩽ n; i += 4) {
    result[i+0] = a[i+0] * b[i+0] + c[i+0];
    result[i+1] = a[i+1] * b[i+1] + c[i+1];
    result[i+2] = a[i+2] * b[i+2] + c[i+2];
    result[i+3] = a[i+3] * b[i+3] + c[i+3];
}
// Handle remainder
for (; i < n; ++i) {  result[i] = a[i] * b[i] + c[i];  }
```

Note: Often compiler does this automatically with `-O3`

When to Stop Optimising

Stop when:

1. **"Good enough" performance achieved** — Meets science requirements
2. **Diminishing returns** — Hours of work for 1% gain
3. **Code becoming unmaintainable** — Complexity not worth it
4. **Hardware bottleneck hit** — Memory bandwidth, disk I/O

Remember: Correct, maintainable code beats fast, broken code **every time**

The Science Behind the Speed

Higgs Discovery Impact

ATLAS 2012:

- Data to analyse: 10 fb^{-1}
- Without optimisation: would have taken far longer
- With optimisation: months instead of years
- **Result: Announced July 4, 2012**

Optimised software enabled faster science.

The Challenge Grows

Run 3 (2022–2026):

- Data rate: higher than Run 1
- CPU budget: limited growth
- **Need: major efficiency improvements**
- Achieved with techniques you're learning now

Your optimisations enable new discoveries.

10% speedup = months saved = faster science

Optimisation Checklist

Before You Start

- [ ] Profile to confirm hotspot (not guessing)
- [ ] Establish baseline measurements
- [ ] Write regression tests
- [ ] Document current performance
- [ ] Set a realistic target

During Optimisation

- [ ] Change one thing at a time
- [ ] Profile after each change
- [ ] Verify correctness with tests
- [ ] Document what you tried (even failures)

Summary: SD L3

Key Takeaways:

1. **Workflow** — Measure → Analyse → Optimise → Verify
2. **Tools** — perf for profiling, flame graphs for visualisation
3. **Roofline model** — Identify if compute or memory bound
4. **Case study** — Real example of 2.5× speedup
5. **Priorities** — Algorithm and data layout first

Next: Hands-on exercises to apply profiling skills

SOFTWARE DESIGN

Exercises: E1 + E2

"Parallel programming: easier to talk about than to do"

2 hours total

Choose your own adventure

Parallelise event processing with OpenMP

GPU programming with CUDA

CPU vectorisation & cache optimisation

Exercise SD E1: Parallelise a CPU Codepath

Threading with OpenMP/TBB (60 minutes)

Starting point: Serial event processing loop

Deliverable: Scaling plot and bottleneck analysis

Your tasks:

1. Parallelise with OpenMP
2. Implement a reduction for counting
3. Measure scaling (1, 2, 4, 8 threads)
4. Diagnose the first bottleneck:
    - False sharing?
    - Load imbalance?
    - Grain size?

SD E1: Getting Started

```
// File: event_processor.cpp (provided)

// Current serial implementation
void processEvents(const std::vector<Event>& events) {
    int totalTracks = 0;

    for (const auto& event : events) {
        auto tracks = event.reconstructTracks();
        totalTracks += tracks.size();

        // More processing...
    }

    std::cout << "Total tracks: " << totalTracks << std::endl;
}
```

Your job: Parallelise this and measure speedup

SD E1: Hints (1/2)

Parallel Implementation

```
void processEvents(const std::vector<Event>& events) {
    int totalTracks = 0;

    #pragma omp parallel for reduction(+:totalTracks)
    for (size_t i = 0; i < events.size(); ++i) {
        auto tracks = events[i].reconstructTracks();
        totalTracks += tracks.size();
    }

    std::cout << "Total tracks: " << totalTracks << std::endl;
}
```

SD E1: Hints (2/2)

Measure scaling

```
for threads in 1 2 4 8; do
    OMP_NUM_THREADS=$threads ./event_processor
done
```

SD E1: What to Look For

Ideal scaling:

```
1 thread:  10.0 seconds (baseline)
2 threads:  5.0 seconds (2.0× speedup)
4 threads:  2.5 seconds (4.0× speedup)
8 threads:  1.25 seconds (8.0× speedup)
```

Reality (you'll see something like):

```
1 thread:  10.0 seconds
2 threads:  5.8 seconds (1.7× speedup) ← Why not 2×?
4 threads:  3.2 seconds (3.1× speedup) ← Why not 4×?
8 threads:  2.1 seconds (4.8× speedup) ← Why not 8×?
```

Your analysis should identify: False sharing? Memory bandwidth? Load imbalance?

Exercise SD E2: Advanced Path

Choose One (60 minutes)

Option A: GPU Porting (if GPUs are available)

- Port hot loop to CUDA/HIP
- Manage host-device transfers
- Perform occupancy analysis
- Compare CPU vs GPU performance

Option B: CPU Vectorisation & Tiling (CPU-only)

- Vectorise critical loop
- Apply tiling for cache optimisation
- Use the roofline model to understand limits
- Show before/after perf counters

SD E2 Option A: GPU Basics — CUDA Example

```
// CPU version
void add(float* a, float* b, float* c, int n) {
    for (int i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}

// GPU version
__global__ void add_kernel(float* a, float* b, float* c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

// Host code
add_kernel<<<(n+255)/256, 256>>>(d_a, d_b, d_c, n);
```

SD E2 Option B: Vectorisation | Manual SIMD

```
#include <immintrin.h> // AVX2

// Vectorised addition (8 floats at once)
void add_avx(float* a, float* b, float* c, size_t n) {
    size_t i = 0;

    // Process 8 floats at a time
    for (; i + 8 ⩽ n; i += 8) {
        __m256 va = _mm256_loadu_ps(&a[i]);
        __m256 vb = _mm256_loadu_ps(&b[i]);
        __m256 vc = _mm256_add_ps(va, vb);
        _mm256_storeu_ps(&c[i], vc);
    }
    // Handle remainder
    for (; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Both Options: Deliverables

What to Submit

Documentation:

- [ ] Performance measurements (before/after)
- [ ] Scaling analysis
- [ ] Bottleneck identification
- [ ] Lessons learned

Code:

- [ ] Clean, commented implementation
- [ ] Tests verifying correctness
- [ ] Build system integration
- [ ] CI running benchmarks

Bonus:

- [ ] Roofline analysis (Option B)
- [ ] Occupancy analysis (Option A)
- [ ] Comparison with theoretical peak

CROSS-CUTTING: INFRASTRUCTURE

The WLCG Stack

15 minutes of context

The WLCG Stack

Worldwide LHC Computing Grid

Problem: LHC data is too big for one site

Solution: Distributed computing across ~170 sites worldwide

Key components:

1. **EOS** — High-throughput storage
2. **CVMFS** — Software distribution
3. **Apptainer** — Container runtime
4. **HTCondor/Slurm** — Job scheduling

EOS: CERN Disk Storage

Exabyte-Scale File System

Characteristics:

- High latency (~10–100ms per operation)
- High throughput (GB/s aggregate)
- Designed for large sequential I/O

Use case: Store physics data files (ROOT, HDF5)

Why we buffer I/O:

```
// Bad: Many small reads (slow on EOS)
for (int i = 0; i < n; ++i) {
    float value;
    file.read(&value, sizeof(float));
}

// Good: One large read (fast on EOS)
std::vector<float> buffer(n);
file.read(buffer.data(), n * sizeof(float));
```

CVMFS: Software Distribution

CernVM File System

Purpose: Distribute software to all grid sites without copying

How it works:

- Read-only file system
- HTTP-based content delivery
- Aggressive caching (via Squid)
- Cryptographic integrity

Mounted at:

- `/cvmfs/sft.cern.ch/`
- `/cvmfs/atlas.cern.ch/`
- etc.

Example: Access LCG software releases

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_104/x86_64-el9-gcc13-opt/setup.sh

root -l # Now have ROOT, Geant4, etc. available
```

Apptainer (formerly Singularity)

Container Runtime for HPC

Why not Docker?

- Docker requires root privileges
- Not allowed on shared grid sites

Apptainer: Rootless container runtime

```
# Pull container
apptainer pull docker://rootproject/root:latest

# Run analysis inside container
apptainer exec root_latest.sif root -l analysis.C

# Mount /cvmfs inside container
apptainer exec --bind /cvmfs root_latest.sif \
    source /cvmfs/sft.cern.ch/lcg/views/LCG_104/setup.sh
```

Why We Need Relocatable Binaries

The Distribution Challenge

Problem: Software built on one site might not run on another

Causes:

- Different library paths (`/usr/lib` vs `/opt/lib`)
- Different compiler versions
- Different OS versions

Solutions:

1. **CVMFS:** All sites mount same software
2. **Containers:** Bundle dependencies
3. **Relocatable builds:** Software works regardless of install path

Putting It Together

Typical HEP Workflow on Grid

```
# 1. Submit job to HTCondor
condor_submit job.sub

# 2. Job runs on a random grid site
# 3. Worker node:
#    - Mounts CVMFS for software
#    - Pulls container if needed
#    - Reads data from EOS via XRootD
#    - Processes events
#    - Writes output back to EOS

# 4. Output available globally
xrdcp root://eosatlas.cern.ch//path/to/output.root ./
```

Key point: Same job runs anywhere on the grid thanks to CVMFS + containers

CONCLUSION

Bringing It All Together

What We've Covered

Tools & Techniques

- ✅ Correctness with testing and sanitisers
- ✅ Collaboration with Git and CI
- ✅ Performance fundamentals

Software Design

- ✅ Many-core programming
- ✅ Parallel patterns and data layout
- ✅ Profiling and optimisation

Infrastructure

- ✅ WLCG stack overview
- ✅ CVMFS, EOS, Apptainer

Key Principles to Remember (1/2)

1. Correctness First

- Test everything
- Use sanitisers
- Don't sacrifice correctness for speed

2. Measure Before Optimising

- Profile to find real hotspots
- Don't guess
- Verify every change

3. Collaborate Effectively

- Atomic commits
- CI as production
- Code review is crucial
- Document decisions
- Ask for help & share knowledge

Key Principles to Remember (2/2)

4. Think About Data Layout

- SoA usually beats AoS
- Cache locality matters
- Pad to avoid false sharing

5. Use the Right Tool

- OpenMP for simple parallelism
- TBB for complex task graphs
- Profile with perf
- Leverage HEP frameworks (ROOT, CMSSW)

Beyond this course & this School

Continue Learning

Resources:

- **Books:**
  - "Computer Systems: A Programmer's Perspective" (Bryant & O'Hallaron)
  - "The Art of Multiprocessor Programming" (Herlihy & Shavit)
- **Online:**
  - CERN/HSF Software Training modules
  - C++ Core Guidelines
  - Intel's optimisation manuals
- **Practice:**
  - Contribute to CMSSW, ROOT, Geant4
  - Optimise your own analysis code
  - Profile real workloads

Final Thoughts

Writing high-performance code for HEP is both:

- **Engineering challenge** — Technical skills matter
- **Scientific discipline** — Correctness is paramount

Hopefully I did a decent job and you now have the tools to:

- Write correct, tested code
- Collaborate effectively in large teams
- Profile and optimise systematically
- Leverage modern parallel hardware

Most importantly: You understand **why** these practices matter for science.

Thank You!

You're Part of History

47th CSC, 1st in Latin America!

You now can:

- Write correct, tested code
- Collaborate in global teams
- Profile and optimise
- Use parallel hardware
- Work with real HEP data

You understand: These skills enable better science.

The Bigger Picture

Remember:

- CERN Open Data makes knowledge accessible
- Same tools discovered the Higgs boson
- Your contributions impact researchers worldwide
- Computing skills enable scientific discovery

"International cooperation is possible and necessary for equity and progress in education."

Your Impact Starts Now

Making a Difference

"International cooperation is possible and necessary for equity and progress in education around the world."

What you can do next:

- Contribute to ATLAS Open Data
- Join HSF working groups
- Mentor students in your country
- Apply these skills in your research
- Share what you learned

The HEP community:

- Is **global** (40+ countries)
- Is **collaborative** (sharing > competing)
- Is **always learning** (~2 weeks at schools like this!)
- **Needs you** to make it better

Let's stay connected. The best discoveries happen when we work together.

What's Next?

Week 1: Set up dev container, add tests, enable sanitisers, profile something

Month 1: Implement CI, parallelise a loop, and contribute a bug fix

This Year: Attend HSF training, present your work, mentor someone

Questions & Contact

During school:

- Ask anytime
- Help each other
- Share what you learn

After:

- https://www.linkedin.com/in/arturo-sanchez-pineda/
- https://github.com/artfisica/csc2026/

Join the Community

Resources:

- HSF Training: https://hepsoftwarefoundation.org/training/
- ATLAS Open Data: https://opendata.atlas.cern
- CERN Docs: https://docs.cern.ch/
- GitHub: https://github.com/HSF
- CERN Mattermost: Join CERN community channels if you can :)

Remember: every expert was once a beginner.

Quick Reference

Testing & Debugging:

```
cmake -DCMAKE_CXX_FLAGS="-fsanitize=address" ..
ctest --output-on-failure
```

Git:

```
git status
git add -p
git commit -m "Fix: ..."
```

Profiling:

```
perf record -g ./program
perf report
```

(Keep this slide handy)

BACKUP SLIDES

Additional Reference Material

Backup: CMake Basics

```
cmake_minimum_required(VERSION 3.16)
project(HEPAnalysis CXX)
set(CMAKE_CXX_STANDARD 17)

find_package(ROOT REQUIRED COMPONENTS Core RIO Tree)
find_package(Catch2 REQUIRED)

add_library(analysis SHARED src/Particle.cpp)
target_link_libraries(analysis PUBLIC ROOT::Core ROOT::Tree)

add_executable(analyze src/main.cpp)
target_link_libraries(analyze analysis)

add_executable(tests tests/test_particle.cpp)
target_link_libraries(tests analysis Catch2::Catch2WithMain)
enable_testing()
add_test(NAME unit_tests COMMAND tests)
```

Backup: GDB Cheat Sheet

Command	Action
`run`	Start execution
`break main`	Set breakpoint
`continue`	Resume after breakpoint
`next` / `step`	Step over / into
`print var`	Print variable
`backtrace`	Show call stack
`frame N`	Switch to frame N
`watch var`	Break when var changes

Start: `gdb ./program` or `gdb --args ./program arg1`

Backup: Compiler Flags

Flag	Purpose
`-O0`	No optimisation (debug)
`-O2`	Good balance
`-O3`	Aggressive
`-march=native`	Optimise for this CPU
`-g`	Debug symbols
`-Wall -Wextra`	Enable warnings
`-fsanitize=address`	ASan

Recommended: `-O2 -g -Wall -Wextra -march=native`

Backup: ROOT Macro Tips

```
void analysis() {
    // Modern way: RDataFrame
    ROOT::RDataFrame df("Events", "data.root");
    auto h = df.Filter("nJets ⩾ 4")
               .Histo1D({"h", "pT", 100, 0, 200}, "pt");
    h->Draw();

    // Save output
    auto out = TFile::Open("output.root", "RECREATE");
    h->Write();
    out->Close();
}
```

Run: `root -l analysis.C`

Backup: Python Performance

```python
import numpy as np

# ❌ Slow: Python loops
result = [x**2 for x in data]

# ✅ Fast: NumPy vectorisation
result = np.array(data) ** 2   # 100× faster!

# ✅ Generators for large data
def process(filename):
    with open(filename) as f:
        for line in f:
            yield transform(line)
```

Profile: `python -m cProfile script.py`

Backup: Performance At-a-Glance

Optimisation Priority:

Impact	Speedup
Algorithm	100–1000×
Data Structure	10–100×
Memory Layout	5–50×
Parallelisation	4–8×
Compiler -O3	2–5×
Micro-opts	1.001×

Start from the top!

Memory Hierarchy:

Level	Cycles	Size
Registers	1	1 KB
L1 Cache	4	64 KB
L2 Cache	12	1 MB
L3 Cache	40	32 MB
RAM	200	128 GB
SSD	50000	2 TB

RAM is 200× slower than L1!

Backup: Further Reading

Books:

- "Effective Modern C++" — Scott Meyers
- "C++ Concurrency in Action" — Anthony Williams
- "Systems Performance" — Brendan Gregg

Communities: CERN Mattermost, HSF Forum

Further Reading — Online

- ROOT documentation — https://root.cern/
- HSF Training — https://hepsoftwarefoundation.org/training/
- C++ Core Guidelines — https://isocpp.github.io/CppCoreGuidelines/

REFERENCES — Core HEP & WLCG

- Worldwide LHC Computing Grid (WLCG): https://home.cern/science/computing/grid
- CERN Open Data Portal: https://opendata.cern.ch/
- ATLAS Open Data: https://opendata.atlas.cern/
- ROOT Framework (home): https://root.cern/

REFERENCES — Testing & Debugging

- Catch2 (C++ testing): https://github.com/catchorg/Catch2
- pytest (Python testing): https://docs.pytest.org/en/stable/
- AddressSanitizer (ASan): https://clang.llvm.org/docs/AddressSanitizer.html
- UndefinedBehaviorSanitizer (UBSan): https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- ThreadSanitizer (TSan): https://clang.llvm.org/docs/ThreadSanitizer.html
- clang-tidy: https://clang.llvm.org/extra/clang-tidy/
- C++ Core Guidelines: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

REFERENCES — Performance & Profiling

- Google Benchmark: https://github.com/google/benchmark
- Linux perf (kernel docs): https://www.kernel.org/doc/html/latest/admin-guide/perf.html
- FlameGraph: https://github.com/brendangregg/FlameGraph
- Roofline Model (LBL): https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

REFERENCES — Parallelism & Frameworks

- OpenMP (official): https://www.openmp.org/
- oneTBB (docs): https://oneapi-src.github.io/oneTBB/
- ROOT RDataFrame (docs): https://root.cern/doc/master/group__dataframe.html
- EnableImplicitMT (ROOT): https://root.cern/doc/master/namespaceROOT.html
- CMSSW (CMS Software): https://cms-sw.github.io/
- Geant4 — Multi-threading: https://geant4-userdoc.web.cern.ch/UsersGuides/ForApplicationDeveloper/html/GettingStarted/multithreading.html

REFERENCES — Infrastructure

- CernVM-FS (CVMFS): https://cernvm.cern.ch/fs/
- EOS (CERN storage): https://eos-docs.web.cern.ch/
- Apptainer (Singularity): https://apptainer.org/docs/
- XRootD: https://xrootd.slac.stanford.edu/

REFERENCES — Training & Community

- HSF Training Center: https://hepsoftwarefoundation.org/training/
- ROOT User Guides: https://root.cern/doc/master/index.html
- CERN Docs: https://docs.cern.ch/
- Communities: CERN Mattermost, HSF Forum

Thank You!

CSC Latin America 2026 — Santiago, Chile

Arturo Sánchez Pineda — INAIT AI | Creative Commons Venezuela