9th Inverted CERN School of Computing

29 February – 2 March 2016
CERN, IT Amphitheatre (31/3-004)

Live webcast, slides & recording at
https://indico.cern.ch/e/iCSC2016/

Lecturers

Kim ALBERTSSON, Luleå University of Technology
Anastasios ANDRONIDIS, Imperial College London
Valentina CAIRO, University of Calabria
Pedro CORREIA, University of Aveiro
Thomas KECK, KIT
Kamil KROL, CERN
Aram SANTOGIDIS, CERN
Daniel SAUNDERS, University of Bristol
Joshua SMITH, University of Göttingen
Jiří VYSKOČIL, Czech Technical University
Welcome...

... to the iCSC2016, 9th edition of the Inverted School, where Students turn into Teachers.

The Inverted Schools of Computing (iCSC) are part of an annual series of schools organized by the CERN School of Computing (CSC). The iCSC consists of lectures presented over few days by former CSC students, providing advanced education in specialist topics.

The iCSC lectures are specially chosen to create a unique educational program. They are written and delivered by selected students from the previous year’s Main School CSC, who demonstrated a very high level of expertise in a given area during their participation at the annual Main School. So why not find a way to promote and share this knowledge, and turn the students into teachers?

This year’s programme is selected from a range of CSC2015 student proposals, and focuses on challenging and innovative topics.

We are indebted to the lecturers at the main CSC who acted as mentors. My thanks go to all those who developed ideas and proposals, and of course to those who are lecturing. This is their school and I am looking forward to listening to their ideas.

Enjoy the School.

Alberto Pace  
Director  
CERN School of Computing
Welcome 3

Contents 4

Core team 6

Lecturer biographies 7

Lecture abstracts 12

Timetable 20

Appendix : lecture slides 22

Lecture 1
Continuous Integration : how can it help?................................. A 1

Lecture 2
Continuous Delivery and Quality Monitoring ........................... A 12

Lecture 3
Template Metaprogramming for Massively Parallel Scientific Computing - Expression Templates – Part 1 .................................................. A 25

Lecture 4
Multivariate Classification and Machine Learning in HEP – Part 1 .......... A 37

Lecture 5
Formal verification - Robust and efficient code: Introduction to Formal Verification – Part 1 ................................................................. A 60

Lecture 6
Data reconstruction in Modern Particle Physics – Part 1 ..................... A 72

Lecture 7
Detector Simulation for the LHC and beyond: how to match computing resources and physics requirements – Part 1 .................................. A 79

Lecture 8
Template Metaprogramming for Massively Parallel Scientific Computing - Vectorization with Expression Templates – Part 2 .................. A 98

Lecture 9
Multivariate Classification and Machine Learning in HEP – Part 2 .......... A 50

Lecture 10
Formal Verification - Robust and Efficient code: Why Formal Verification – Part 2 .......... A 113
CONTENTS (continued)

Lecture 11
Accelerating C++ applications in Medical Physics ................................. A 125

Lecture 12
Event reconstruction in Modern Particle Physics – Part 2 ......................... A 136

Lecture 13
Detector simulation for the LHC and beyond: how to match computing resources and physics requirements – Part 2 .......................................................... A 145

Lecture 14
Template Metaprogramming for massively parallel scientific computing -
Templates for Iteration; Thread-level Parallelism – Part 3 ......................... A 161

Lecture 15
Shared memory and message passing revisited in the many-core era ........ A 174

Lecture 16
Volatile Environments with Virtualisation Technologies ........................... A 183
# CORE TEAM

**Director**
Alberto PACE

**Administration**
Cath NOBLE

**Mentors**
Benedikt HEGNER
Nikos KASIOUMIS
Giuseppe LO PRESTI
Sebastian LOPIENSKI
Pere MATO
Danilo PIPARO
Sebastien PONCE
Ivica PULJAK
Are STRANDLE
Andreas Joachim PETERS

The inverted School of Computing (iCSC) is part of the annual series of schools organized by the CERN School of Computing [http://cern.ch/csc](http://cern.ch/csc)
Every lecturer presenting here at the iCSC2016 has been specially selected to advance their ideas, help further develop CSC-related themes, and to share and promote their knowledge.

All were students at the 2015 CERN School of Computing (CSC2015) in Kavala, Greece.

Kim ALBERTSSON

*University of Technology - Lulea – Sweden*

Kim Albertsson is currently at Luleå University of Technology where he is involved in a project researching sparse coding and perception. His part is focusing on implementation in VHDL, enabling real-time learning and processing.

Previously he worked at the data acquisition group at the CMS experiment providing tools for the configuration of the DAQ cluster computers. He graduated in Applied Physics and Electrical Engineering from Luleå University of Technology.

*Lecture*

*Formal Verification – robust and efficient code*

*Mentors*

*Giuseppe Lo Presti, Sebastien Ponce*

Anastasios ANDRONIDIS

*Imperial College London – UK*

Anastasis Andronidis is a computer scientist graduated with excellence from Aristotle University of Greece and now he is a PhD candidate in Imperial College London working on software reliability, multi-version execution and dynamic updates.

His work experience starts with 4 years in the GRID, multiple internships at CERN and Google Summer of Code (Apache Hama and Kubernetes) and 2 years at CERN on OpenStack solutions and distributed systems.

*Lecture*

*Volatile Environments with Virtualisation Technologies*

*Mentors*

*Giuseppe Lo Presti, Pere Mato*
Valentina CAIRO

University of Calabria – Arcavacata – Italy & CERN

I got my Master Degree in Physics with honors in 2013 at the University Of Calabria, in Italy. I am now a PhD student at the same University and a Doctoral Student at CERN, in the ATLAS Collaboration.

The current work is essentially split in two main parts: I am contributing to the integration and development of the offline simulation programs of the New Small Wheel and on the main validation and physics simulation activities. I am also actively participating in the data analysis for LHC Run-II at 13 TeV, working on optimisation procedures of analysis programs and strategies, tracking performance, material distribution studies and Standard Model analyses.

Lecture
Detector Simulation for the LHC and beyond: how to match computing resources and physics requirements
Mentors
Danilo Piparo, Benedikt Hegner

Pedro CORREIA

University of Aveiro – Portugal

I am a PhD Student at the University of Aveiro, funded by a Portuguese scholarship, and the topic of my PhD is the study and development of new Positron Emission Tomography (PET) scanners for pre-clinical research. The development of new and robust algorithms for PET image reconstruction and data analysis are one of my main tasks.

I finished my master degree at the University of Aveiro in 2013, and during my thesis I’ve developed algorithms both for the simulation of the operation of Micro-Pattern Gas Detector as well as for the understanding and improvement of such detectors, that are being investigated within RD51 collaboration for the upgrade of experiments such as ATLAS and CMS. The algorithms have been developed in two main software tools, Root and Garfield++, with a strong integration of C++ and Python capabilities. I am also an enthusiastic in new technologies and some of my hobbies are related with 3D printers and development of do-it-yourself electronic solutions.

Lecture
Accelerating C++ applications in Medical Physics
Mentors
Pere Mato, Benedikt Hegner
Thomas KECK

*KIT Karlsruhe – Germany*

I am a Ph. D. student in Physics at the Karlsruhe Institute for Technology in Germany. As a member of the Belle II collaboration I am responsible for the development of data-driven multivariate analysis methods.

In particular my work is focused on hadronic and semileptonic tagging algorithms, and their application to rare B meson decays. I am an experienced C++ and python developer, and interested in any new technologies in the field of computer science. At the moment I have a strong interest in data mining algorithms, pattern recognition using deep neural networks, as well as visualisation and sonification of scientific data. Besides work, I enjoy playing guitar and practicing taekwondo.

*Lecture*

*Multivariate Classification and Machine Learning in HEP*

*Mentors*

*Ivica Puljak, Benedikt Hegner*

Kamil KROL

*CERN – Switzerland*

Kamil Król is a professional Software Engineer. He obtained Master’s Degree in Computer Science and Bachelor’s Degree in Management at AGH UST in Krakow (Poland). Since 2013 he has been working at CERN designing and developing applications for machine protection purposes. Before joining CERN, Kamil had opportunity to gain his professional experience in one of the biggest IT companies in Poland developing software for the telecommunication sector.

Among his main topics of professional interest are cloud and distributed computing, back-end development, NoSQL data storages, systems integration and different aspects of the agile team management. Kamil likes to spend his free time reading and doing sports (hiking, mountaineering, climbing, skiing, cycling and running).

*Lecture*

*Continuous Delivery and Quality Monitoring*

*Mentors*

*Sebastian Lopienski, Nikos Kasioumis*
Aram SANTOGIDIS

CERN – Switzerland

Aram Santogidis received his Diploma in Computer & Communications Engineering from the University of Thessaly in Greece. During his final year of studies he worked as an R&D software engineer for AvMap Italy. After his graduation he did an internship at Alcatel-Lucent Bell Labs in Ireland and soon after that he was hired as software developer at IBM Ireland.

In October 2013 he joined CERN for the “Intel-CERN European Doctorate Industrial Program” project, where he is conducting research on the data transfer aspects on manycore processors for high-throughput computing applications. His research is conducted as part of his PhD studies in the Maynooth University in Ireland. He is interested in systems software technologies that efficiently utilize modern computing architectures and expose intuitive programming interfaces.

Lecture
*Shared memory and message passing revisited in the many-core era*

Mentors
*Sebastian Lopienski, Andreas Joachim Peters*

Daniel SAUNDERS

University of Bristol - United Kingdom

I am currently studying towards my Particle Physics PhD (final year), working on the SoLid and LHCb experiments. Increasingly, both of these experiments involve very large data outputs, introducing many new computational challenges that I am particularly interested in. On SoLid (a novel reactor Neutrino experiment) I focus on raw data reconstruction and related physics analysis, including neutrino cross section measurements.

I also develop software for analysing test beam data for the LHCb Vertex Locator (VELO) upgrade pixel detectors, also focusing on event reconstruction and efficiency measurements.

Lecture
*Event reconstruction in Modern Particle Physics*

Mentors
*Danilo Piparo, Are Strandle*
Joshua SMITH

Georg-August Universität Göttingen - Germany

Joshua obtained his Master’s degree in Physics from the University of Cape Town. His work focused on porting the ATLAS software framework (Athena) to the ARM architecture. The complexity of this task due to the size of the codebase makes a good workflow, such as Continuous Integration, absolutely essential. Recently, he has taken up a PhD position at Georg-August Universität, Göttingen in Germany. Currently he is working on his qualification task for ATLAS, which involves porting outdated consoles used in pixel/IBL to more modern Qt4 code.

He is also continuing the ARM port and validation. When his physics analysis starts it will be in top-quark related physics for the ATLAS experiment.

Lecture
Continuous Integration: how can it help?

Mentors
Sebastian Lopienski, Nikos Kasioumis

Jiří VYSKOCIL

Czech Technical University, Prague - Czech Republic

I am a PhD student at the Faculty of Nuclear Sciences and Physical Engineering of the Czech Technical University in Prague, where I obtained the Masters degree in applied physics for a thesis titled "3D PIC code for simulations of short pulse laser target interactions - three-dimensional effects" under the supervision of assoc. prof. Ondřej Klimo and prof. Jiří Limpouch in 2012.

My research is in the area of particle acceleration in interactions of short ultra-intense laser pulses with plasmas, at present time with focus on electron acceleration using multi-PW lasers driving plasma waves in interactions with gas targets in the framework of the ELI Beamlines project. I am working with various parallel Particle-in-Cell codes, and developing my own massively parallel hybrid MPI/OpenMP code in C++ with heavy use of template meta-programming. My other HPC related interests also include visualizations of large datasets, and general Linux and Unix-like systems development.

Lecture
Template Metaprogramming for Massively Parallel Scientific Computing

Mentors
Nikos Kasioumis, Sebastien Ponce
LECTURE ABSTRACTS

The lectures are broadcast via webcast and will be recorded. All slides, recordings etc can be found at https://indico.cern.ch/e/iCSC2016

Lecture 1

Continuous Integration : how can it help?  

Joshua SMITH

Software becomes more complex as the project size and number of developers grow. As these two factors increase, so too does the potential for more errors in the code. Previous time and money can be wasted trying to track down bugs that could have been easily avoided should there have been a good workflow in place.

Continuous Integration (CI) is one such strategy that can dramatically improve software quality. An in-depth look at what CI is, as well as the fundamental concepts will be explored. Various scenarios of how CI can be incorporated into different types of projects will be covered. There are many CI software packages on the market. It’s not always easy choosing what CI package is best suited for your project. Some main points to keep in mind when beginning to implement CI into your project will also be discussed.

Lecture 2

Continuous Delivery and Quality Monitoring  

Kamil KROL

We’re all involved in some software/physics projects. As a rule of thumb projects start really simple - a couple of scripts, classes and a few external dependencies. At this phase delivering a release to our clients is simple. We can compile the project locally and deliver compiled sources, for example by e-mail. Unfortunately, in most cases the growth of projects is inevitable. Our simple approaches to build, test and deliver applications are not sufficient. We start to spend more and more time on these ‘administrative’ procedures than on the real developments. As the project grows, our productivity declines and we are less responsive to requests from our clients.

In this lecture I will try to present common delivery patterns and tools which facilitate these processes. After introducing Continuous Delivery, I will switch the topic and try to answer the question how much should we invest in quality and how to do it efficiently. My observations reveal that software quality is often considered as the slowing down force. Following this false belief I would like to convince people that software quality can accelerate development within our projects.
Lecture 3

Template Metaprogramming for Massively Parallel Scientific Computing (Part 1 of 3)

Jiří VYSKOČIL

Large scale scientific computing raises questions on different levels ranging from the formulation of the problems to the choice of the best algorithms and their implementation for a specific platform. There are similarities in these different topics that can be exploited by modern-style C++ template metaprogramming techniques to produce readable, maintainable and generic code. Traditional low-level code tend to be fast but platform-dependent, and it obfuscates the meaning of the algorithm. On the other hand, object-oriented approach is nice to read, but may come with an inherent performance penalty.

These lectures aim to present the basics of the Expression Template (ET) idiom which allows us to keep the object-oriented approach without sacrificing performance. We will in particular show how to enhance ET to include SIMD vectorization. We will then introduce techniques for abstracting iteration, and introduce thread-level parallelism for use in heavy data-centric loads. We will show how to apply these methods in a way which keeps the “front end” code very readable.

Part 1: Expression Templates

In this lecture, we will have a quick look at the basics of Template Metaprogramming - how does the computing handle “types”, and how these types map more or less naturally to physical quantities. We will then introduce the idea of using Expression Templates (ET) as a means to bridge the gap between the low-level high-performance approach, and the object-oriented, readable but often severely under-performing approach. We will study the structure of the ETs, and show the basic steps needed to build them.

Lecture 4

Multivariate Classification and Machine Learning in HEP (Part 1 of 2)

Thomas KECK

Traditional multivariate methods for classification (Stochastic Gradient Boosted Decision Trees and Multi-Layer Perceptrons) are explained in theory and practice using examples from HEP. General aspects of multivariate classification are discussed, in particular different regularisation techniques. Afterwards, data-driven techniques are introduced and compared to MC-based methods.
Lecture 5

Formal Verification - Robust and Efficient code (Part 1 of 2)

Kim ALBERTSSON

This talk aims to introduce the concepts of Formal Verification and how they can be used to the benefit of the programmer to produce robust and efficient code. We will be looking into the subject at two levels, both and overview of what FV can concretely bring programmers and going into the nitty-gritty details of theorem proving one of the methods use for FV. In general, FV means "proving that certain properties hold for a given system using formal mathematics". This definition can certainly feel daunting, however, as we will learn, we can reap benefits from the paradigm without digging too deep into the subject. Examples where FV can help include proving that your code cannot raise division by zero exceptions; produce optimised byte code where the optimisations are proven to be safe and help reason about concurrent systems.

Part 1: Introduction to Formal Verification

We will establish two general approaches to FV and where they are applicable: model checking and theorem proving. We will explore the latter in more details and have a brief look at the underlying theory, predicate logic. We will see how this family of logic systems can be used to prove abstract properties of our program and why this is useful. Practical examples will be presented and explained.

Lecture 6

Data reconstruction in Modern Particle Physics (Part 1 of 2)

Daniel SAUNDERS

Particle physics experiments have always been at the forefront of big data experiments: the upgrade of the LHCb experiment will lead to data rates greater than 10Tb's per second! This is key to the success of high-energy physics, where large data samples, sophisticated triggers and robust simulations have lead to observing and understanding extremely rare events, including the Higgs Boson. Continuously, physicists are revisiting computing and electronics decisions to balance the differences between the quality and quantity of physics results, computing effort and available budgets. By drawing on examples of modern particle physics experiments, these lectures will consider the various approaches to tackle such large particle physics data problems:

- Data reduction at the hardware level, including triggers.
- Principles and optimizations of reconstruction algorithms.
- Parallelized reconstruction, including sub processing vs. multithreading.

These topics will be introduced in order, beginning with raw data output from these large experiments, passing through the different stages of data reconstruction and reduction, leading to examples of physics results. The lecture series will end with an outlook to future technologies and the associated physics.
**Lecture 7**

Detector Simulation for the LHC and beyond: how to match computing resources and physics requirements (Part 1 of 2)

Valentina CAIRO

Detector simulation at the LHC is one of the most computing intensive activities. In these lectures we will show how physics requirements were met for the LHC experiments and extrapolate to future experiments (FCC-hh case). At the LHC, detectors are complex, very precise and ambitious: this implies modern modelisation tools for geometry and response. Events are busy and characterised by an unprecedented energy scale with hundreds of particles to be traced and high energy showers to be accurately simulated. Furthermore, high luminosities imply many events in a bunch crossing and many bunch crossings to be considered at the same time. In addition, backgrounds not directly correlated to bunch crossings have also to be taken into account.

Solutions chosen for ATLAS (a mixture of detailed simulation and fast simulation/parameterisation) will be described and CPU and memory figures will be given. An extrapolation to the FCC-hh case will be tried by taking as example the calorimeter simulation.

**Lecture 8**

Template Metaprogramming for Massively Parallel Scientific Computing (Part 2 of 3)

Jiří VYSKOČIL

Large scale scientific computing raises questions on different levels ranging from the formulation of the problems to the choice of the best algorithms and their implementation for a specific platform. There are similarities in these different topics that can be exploited by modern-style C++ template metaprogramming techniques to produce readable, maintainable and generic code. Traditional low-level code tend to be fast but platform-dependent, and it obfuscates the meaning of the algorithm. On the other hand, object-oriented approach is nice to read, but may come with an inherent performance penalty.

These lectures aim to present the basics of the Expression Template (ET) idiom which allows us to keep the object-oriented approach without sacrificing performance. We will in particular show to to enhance ET to include SIMD vectorization. We will then introduce techniques for abstracting iteration, and introduce thread-level parallelism for use in heavy data-centric loads. We will show to to apply these methods in a way which keeps the “front end” code very readable.

**Part 2: Vectorization with Expression Templates**
In this lecture, we will have a closer look at the opportunities for implementing SIMD vectorisation through the Expression Template idiom. We will see how it can create a layer of separation between the algorithm, and the low-level implementation. We will
use the C++ template mechanisms to accommodate our program so that the algorithm itself doesn’t need to explicitly specify SIMD-related types alignment, or operations. We will also explore how our memory data structure layout affects SIMD performance in different workloads, and introduce methods which improve performance in specific cases.

Lecture 9

Multivariate Classification and Machine Learning in HEP (Part 2 of 2)  
Thomas KECK

Traditional multivariate methods for classification (Stochastic Gradient Boosted Decision Trees and Multi-Layer Perceptrons) are explained in theory and practise using examples from HEP. General aspects of multivariate classification are discussed, in particular different regularisation techniques. Afterwards, data-driven techniques are introduced and compared to MC-based methods.

Lecture 10

Formal Verification - Robust and Efficient code (Part 2 of 2)  
Kim ALBERTSSON

This talk aims to introduce the concepts of Formal Verification and how they can be used to the benefit of the programmer to produce robust and efficient code. We will be looking into the subject at two levels, both and overview of what FV can concretely bring programmers and going into the nitty-gritty details of theorem proving one of the methods use for FV. In general, FV means “proving that certain properties hold for a given system using formal mathematics”. This definition can certainly feel daunting, however, as we will learn, we can reap benefits from the paradigm without digging too deep into the subject. Examples where FV can help include proving that your code cannot raise division by zero exceptions; produce optimised byte code where the optimisations are proven to be safe and help reason about concurrent systems.

LECTURE 2: Why Formal Verification

In this lecture we will expand on the concepts of the previous lecture and establish formal methods in a broader context, ignoring implementation detail, and investigate how and where these methods are used today, and where they might be used tomorrow. As concrete examples we will be studying how FV can benefit static analysis and comp-cert, and verified C compiler.
Lecture 11

**Accelerating C++ applications in Medical Physics**

*Pedro CORREIA*

The recent developments in multithreading tools in C++, like OpenMP and TBB, taking advantage of the multicore architecture of the nowadays processors, allowed the creation and improvement of powerful softwares for scientific research. This talk will be focused on the development of such software for simulations, data acquisition and image reconstruction in Positron Emission Tomography, one of the most powerful tools for cancer detection.

Lecture 12

**Event reconstruction in Modern Particle Physics (Part 2 of 2)**

*Daniel SAUNDERS*

Particle physics experiments have always been at the forefront of big data experiments: the upgrade of the LHCb experiment will lead to data rates greater than 10Tb’s per second! This is key to the success of high-energy physics, where large data samples, sophisticated triggers and robust simulations have lead to observing and understanding extremely rare events, including the Higgs Boson. Continuously, physicists are revisiting computing and electronics decisions to balance the differences between the quality and quantity of physics results, computing effort and available budgets. By drawing on examples of modern particle physics experiments, these lectures will consider the various approaches to tackle such large particle physics data problems:

- Data reduction at the hardware level, including triggers.
- Principles and optimizations of reconstruction algorithms.
- Parallelized reconstruction, including sub processing vs. multithreading.

These topics will be introduced in order, beginning with raw data output from these large experiments, passing through the different stages of data reconstruction and reduction, leading to examples of physics results. The lecture series will end with an outlook to future technologies and the associated physics.

Lecture 13

**Detector Simulation for the LHC and beyond: how to match computing resources and physics requirements (Part 2 of 2)**

*Valentina CAIRO*

Detector simulation at the LHC is one of the most computing intensive activities. In these lectures we will show how physics requirements were met for the LHC experiments and extrapolate to future experiments (FCC-hh case). At the LHC, detectors are complex, very precise and ambitious: this implies modern modelisation tools for
geometry and response. Events are busy and characterised by an unprecedented energy scale with hundreds of particles to be traced and high energy showers to be accurately simulated. Furthermore, high luminosities imply many events in a bunch crossing and many bunch crossings to be considered at the same time. In addition, backgrounds not directly correlated to bunch crossings have also to be taken into account.

Solutions chosen for ATLAS (a mixture of detailed simulation and fast simulation/parameterisation) will be described and CPU and memory figures will be given. An extrapolation to the FCC-hh case will be tried by taking as example the calorimeter simulation.

Lecture 14

Template Metaprogramming for Massively Parallel Scientific Computing (Part 3 of 3)

Jiří VYSKOČIL

Large scale scientific computing raises questions on different levels ranging from the formulation of the problems to the choice of the best algorithms and their implementation for a specific platform. There are similarities in these different topics that can be exploited by modern-style C++ template metaprogramming techniques to produce readable, maintainable and generic code. Traditional low-level code tend to be fast but platform-dependent, and it obfuscates the meaning of the algorithm. On the other hand, object-oriented approach is nice to read, but may come with an inherent performance penalty.

These lectures aim to present the basics of the Expression Template (ET) idiom which allows us to keep the object-oriented approach without sacrificing performance. We will in particular show to to enhance ET to include SIMD vectorization. We will then introduce techniques for abstracting iteration, and introduce thread-level parallelism for use in heavy data-centric loads. We will show to to apply these methods in a way which keeps the “front end” code very readable.

Part 3: Templates for Iteration; Thread-level Parallelism

In this lecture, we will look into a specific technique to parallelize a large data-centric workload iterating over a multi-dimensional array. We will show how to separate iteration and computation and how the “front-end” algorithm can then be made independent on the dimensionality, coordinate system, or order of numerical approximation. We will show how this separation further helps to implement thread-level parallelism into the "back-end" and explore some common cases of data dependency. We will finally take a look at an example code combining the ideas of all three lectures.
Lecture 15

Shared memory and message passing revisited in the many-core era

Aram SANTOGIDIS

In the 70s, Edsgar Dijkstra, Per Brinch Hansen and C.A.R Hoare introduced the fundamental concepts for concurrent computing. It was clear that concrete communication mechanisms were required in order to achieve effective concurrency. Whether you're developing a multithreaded program running on a single node, or a distributed system spanning over hundreds of thousands cores, the choice of the communication mechanism for your system must be done intelligently because of the implicit programmability, performance and scalability trade-offs.

With the emergence of many-core computing architectures many assumptions may not be true anymore. In this talk we will try to provide insight on the characteristics of these communication models by providing basic theoretical background and then focus on concrete practical examples based on indicative use case scenarios. The case studies of this presentation cover popular programming models, operating systems and concurrency frameworks in the context of many-core processors.

Lecture 16

Volatile Environments with Virtualisation Technologies

Anastasios ANDRONIDIS

Sometimes our job or even our interest to learn something new, requires from us to install a lot of different software to allow a specific program to run on our operating system. This (in the best case) might just prohibit your program to run due to conflicts between different library or language versions; in the worst case your operating system will start becoming full of junk and later on will be slow and insecure.

In this course we will explore two relatively new but very well established virtualisation technologies: Vagrant and Docker and how those tools can help us to keep a tidy, exportable and transferable developing environment on our home or work computer.
# TIMETABLE

## Monday 29 February

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:00 - 09:10</td>
<td>Welcome</td>
<td></td>
</tr>
<tr>
<td>09:10 - 09:30</td>
<td>Introduction to the inverted CSC</td>
<td></td>
</tr>
<tr>
<td>09:30 - 10:30</td>
<td><strong>L1</strong> Continuous Integration : how can it help?</td>
<td>Joshua SMITH</td>
</tr>
<tr>
<td>10:30 - 11:00</td>
<td>Coffee</td>
<td></td>
</tr>
<tr>
<td>11:00 - 12:00</td>
<td><strong>L2</strong> Continuous Delivery and Quality Monitoring</td>
<td>Kamil KROL</td>
</tr>
<tr>
<td>12:00 - 13:30</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>13:30 - 14:30</td>
<td><strong>L3</strong> Template Metaprogramming for Massively Parallel Scientific</td>
<td>Jiří VYSKOČIL</td>
</tr>
<tr>
<td></td>
<td>Computing - Expression Templates – Part 1</td>
<td></td>
</tr>
<tr>
<td>14:30 - 15:30</td>
<td><strong>L4</strong> Multivariate Classification and Machine Learning in HEP – Part 1</td>
<td>Thomas KECK</td>
</tr>
<tr>
<td>15:30 - 16:00</td>
<td>Coffee</td>
<td></td>
</tr>
<tr>
<td>16:00 - 17:00</td>
<td><strong>L5</strong> Formal verification - Robust and efficient code:</td>
<td>Kim ALBERTSSON</td>
</tr>
<tr>
<td></td>
<td>Introduction to Formal Verification – Part 1</td>
<td></td>
</tr>
</tbody>
</table>

## Tuesday 1 March

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:00 - 10:00</td>
<td><strong>L6</strong> Data reconstruction in Modern Particle Physics – Part 1</td>
<td>Daniel SAUNDERS</td>
</tr>
<tr>
<td>10:00 - 11:00</td>
<td><strong>L7</strong> Detector Simulation for the LHC and beyond: how to match</td>
<td>Valentina CAIRO</td>
</tr>
<tr>
<td></td>
<td>computing resources and physics requirements – Part 1</td>
<td></td>
</tr>
<tr>
<td>11:00 - 11:30</td>
<td>Coffee</td>
<td></td>
</tr>
<tr>
<td>11:30 - 12:30</td>
<td><strong>L8</strong> Template Metaprogramming for Massively Parallel Scientific</td>
<td>Jiří VYSKOČIL</td>
</tr>
<tr>
<td></td>
<td>Computing - Vectorization with Expression Templates – Part 2</td>
<td></td>
</tr>
<tr>
<td>12:30 - 14:00</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>14:00 - 15:00</td>
<td><strong>L9</strong> Multivariate Classification and Machine Learning in HEP – Part 2</td>
<td>Thomas KECK</td>
</tr>
<tr>
<td>15:00 - 16:00</td>
<td><strong>L10</strong> Formal verification - Robust and Efficient code:</td>
<td>Kim ALBERTSSON</td>
</tr>
<tr>
<td></td>
<td>Why Formal Verification – Part 2</td>
<td></td>
</tr>
<tr>
<td>16:00 - 16:30</td>
<td>Coffee</td>
<td></td>
</tr>
<tr>
<td>16:30 - 17:30</td>
<td><strong>L11</strong> Accelerating C++ applications in Medical Physics</td>
<td>Pedro CORREIA</td>
</tr>
<tr>
<td>Time</td>
<td>Room</td>
<td>Session Title</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>09:00 - 10:00</td>
<td>L12</td>
<td>Event reconstruction in Modern Particle Physics – Part 2</td>
</tr>
<tr>
<td>10:00 - 11:00</td>
<td>L13</td>
<td>Detector simulation for the LHC and beyond: how to match computing resources and physics requirements – Part 2</td>
</tr>
<tr>
<td>11:00 - 11:30</td>
<td></td>
<td>Coffee</td>
</tr>
<tr>
<td>11:30 - 12:30</td>
<td>L14</td>
<td>Template Metaprogramming for massively parallel scientific computing: Templates for Iteration; Thread-level Parallelism – Part 3</td>
</tr>
<tr>
<td>12:30 - 14:00</td>
<td></td>
<td>Lunch</td>
</tr>
<tr>
<td>14:00 - 15:00</td>
<td>L15</td>
<td>Shared memory and message passing revisited in the many-core era</td>
</tr>
<tr>
<td>15:00 - 16:00</td>
<td>L16</td>
<td>Volatile Environments with Virtualisation Technologies</td>
</tr>
</tbody>
</table>
Continuous Integration: How can it help?

Joshua Wyatt Smith
Georg-August-Universität Göttingen, Germany

Outline

- The challenge
- Continuous Integration
  - What?
  - Why?
  - How?
  - Who?
- CI General Concepts
  - slave/master model
  - schedulers, Build steps, Testing, Feedback
- CI Examples
  - Jenkins and Buildbot
- Conclusions

The Challenge

- As the project grows, complexity grows:
  - Physical code size
  - Dependencies
  - Number of developers
  - Package versions
- 3 OpenSource examples that show code complexity:
  - ROOT
  - Geant4
  - Linux Kernel

The challenge - ROOT

- HEP analysis framework
  - Data + event reconstruction, data analysis, detector simulation and tracking etc
- Just under 2 million lines of code
  - Mostly C++ and C
Continuous Integration: How can it help?

The challenge - ROOT

- 524 person-years
- Estimated cost: ~ €29 million*
- 48,444 commits from 131 developers

* All cost estimates calculated using basic COCOMO model

The challenge - Geant4

- A framework for the simulation of the passage of particles through matter.
  - Used in HEP, medical and space physics
  - Just under 2 million lines of code
  - Mostly C++

The challenge - Architectures

- Different environments: computer architectures, compilers
  - x86_64-mac-10XY-clang70
  - x86_64-slc6-gcc4X
  - x86_64-slc6-clang35
  - x86_64-centos7-gcc4X
  - x86_64-fedora2X-gcc4Y
  - architecture-OS-compiler
The challenge - Dependencies and versions

- i.e. LCG packages
- Version #'s CRUCIAL!
- Huge dependencies

Continuous Integration (CI) - What?

“. . . a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.”

- Martin Fowler

In a nutshell...

- Use various existing tools to:
  - Combine changes often
  - Build often
  - Test often
  - Deploy often

- CI Software oversees this
- Utilize resources. The days of punch-cards are over (phew!)
Continuous Integration: How can it help?

Components

- Code Repository (SVN, Git, CVS, …)
- Poll
- Feedback

Developers

Slaves

Commits

CI Server

Early/rapid feedback!
- Do all components/projects compile?
- Coding standards?
- Are tests successful?
- Performance requirements?
- Problems archiving or deploying?

Better project visibility
- Possible to notice trends
- What features are needed/being added

Insures clean environments
- Manual tasks automated
  • Speedup of working software turnover
- No large integration steps
  • Much less likely to break something
- A full working/deployable version at ANY POINT IN TIME
- Complete documentation of who did what

In order for CI to work, individual developers should:
- Commit frequently
  • Many small commits
- Run local build first (if possible)
  • Huge code repos may make this difficult
  • Only commit working code
- Fix broken builds immediately
- Write automated tests
CI - Who?

Who should CI?
- Your code that builds your thesis?
  - Probably not
- A project with more than 2 developers?
  - Yes
- Analysis/hardware groups within CERN?
  - ABSOLUTELY!
  - Why don’t they????

CI - General Concepts

- Hopefully you’re convinced that CI is useful
- What do we need to CI?
  - Hardware
    - CI Server + CI Slaves
  - Software
    - What do we build?
    - When do we build?
    - How do we build?
    - How do we test?
    - How do we get feedback?

CI General Concepts: Master - slave

- Everything on 1 server
  - Possible, not ideal unless for special projects
- Distributed builds
  - faster, safer, scalable, multiple platforms, more expensive

CI Server:
- Poll version control repo
- Perform actions on scheduled basis
- Send emails
- Host web accessible dashboard
- Store and display history of builds

CI Slaves:
- Pure workhorse
- Various architectures
- Reduces build time
- Communication through built in ssh-client (or similar)
- Receives action from server
- Carries out action

CI General Concepts: Schedulers

When should we build?
- Polling scheduler
  - If changes in repo, build
- Periodic scheduler
  - Build will start after x seconds
- Nightly scheduler
  - Build at particular time of day/month/year
- Try scheduler
  - Build non-committed code within the CI framework
- Instant gratification scheduler
  - “I want this built now!”
Continuous Integration: How can it help?

### Schedulers (cntd)

- **Dependent or Triggerable scheduler**
  - I.e. a packaging step. Only package if all previous builds and tests were successful
  - Atlas software stack: Each level is dependent on the other. Or, each level triggers the next

### CI General Concepts: Build steps

**How does the project get built?**

- Ideally, one command should build whole project
  - Not always possible.
- **What tools/software are needed to build project?**
  - Cmake, make, java compiler etc.
- **How long does each build take?**
  - Can it be broken up into smaller projects?
  - Even smaller projects?

### CI General Concepts: Testing

- **Absolutely crucial for CI!**
- **Several forms of testing:**
  - **UnitTests**
    - First step of testing
    - Can be run on all commits
    - During build
    - Or after build
  - **Integration tests**
    - Does my new code/changes break the rest of the build?
    - Probably after build

### Testing (cntd)

- **Custom tests**
  - Often much more lengthy, performed less often
  - Software validation
    - Errors? Segfaults?
  - Physics validation (LHC)
    - Correct output?
  - Possibly a dedicated slave
  - **All should be automated!**
  - Performance profiling is also important
Continuous Integration: How can it help?

CI General Concepts: Feedback

Feedback can be many things
- Website widget
- Email notifications
- IRC bot
- Extreme feedback
  - Lamps
  - Ambient orbs
  - Retaliation - Foam missiles (seriously)

CI Tools

- Code repositories
  - BitKeeper, Bzr, CVS, Darcs, Gerrit, Git, Mercurial, Monotone, P4, SVN ...
- Test frameworks
  - CppUnit, Valgrind, JUnit, Unittest, TestNg ...
- Continuous Integration
  - Bamboo, Buildbot, CruiseControl, Jenkins, Gitlab CI ...

Wikipedia has a great comparison table:

Jenkins and Buildbot

- Why?
  - Two CI tools at “opposite end of the spectrum”. I.e. one is well known the other is not
  - Both OpenSource!
  - Similar concepts, executed differently
  - Both easy to use! (spoiler: Jenkins is easier)
  - They are fun to use! (especially Buildbot, but maybe that’s just me)

Jenkins

- Created as Hudson in 2004
  - Due to a dispute with Oracle, Hudson was branched to create Jenkins in 2011. Both are still ongoing
  - Both claim the other are branches
- In the last 12 months:
  - Hudson (still Oracle owned) has: 101 commits with 7 contributors
  - Jenkins has: 4668 commits with 292 contributors
  - Good job Oracle! ✨
- Written in Javascript
- A plugin for just about anything (JIRA, Ults, Authentication, CI Game etc.)
- Can be configured entirely from web GUI
- Large support community
Jenkins

- Has won multiple awards and accomplishments
  - Bossie Awards 2014: The best open source application development tools
  - Geek Choice Award 2014
  - DevOps & SCM, SDTimes 100, 2014
  - etc...
- Used by many high profile organizations/companies

Jenkins - Main page

- Quick overview of builds
- “Weather” notification system
- Everything can be managed through this page

Jenkins - Example

- Testing scripts can be added here

Jenkins - Example

(Should be in a script, in the project)
**Buildbot**

- Written in **Python**, on top of Twisted libraries
  - Twisted = an event driven network program framework
  - Heavily configurable!
- Really more of a CI framework as opposed to a CI program
- Heavily focused around buildslaves
  - Designed to require minimal setup
  - **GUI is an html “Waterfall”**
    - Can control builds from this page, but there are limitations

---

**Buildbot**

- Also has a lot of high profile Organizations/companies

---

**Buildbot - master.cfg**

- Everything specified in this **Python file**
  - Builds Slaves
  - Schedulers
  - Builders
  - Status targets
- A dictionary that Buildbot knows how to handle
  - How you create that dictionary is up to you!
- Initially launch Buildbot daemon from command line

```python
# Example Buildbot master.cfg

c = BuildmasterConfig = {}
c[‘slaves’] = [...] c[‘schedulers’] = [...] c[‘builders’] = [...] c[‘status’] = [...] ...
```

---

**BuildBot - Example**

```python
LGStack_m10 = SingleBranchScheduler(name="LGStack_m10", change_filter=ChangeFilter(branch="LGCMTE_P8"), treeStableTiers=30, builderNames="[LGStack_m10_slave1!"])

gaudi_rel_6_tsch = Triggerable(name="GAUDI_m10 rel_6!", builderNames="[GAUDI_m10 rel_6!"])

# Schedulers

LGFactory.addStep(SSHCommand(name="make all!",
   Commande="make", “all”", “-j10", “-k"),
   workdir="build/logmake!",
   timeout=3600),
   description="Build and make all (-j10 -k)")

LGFactory.addStep(Thread(name="[GAUDI_m10 rel_6!",
   workdir="build/logmake!",
   timeout=3600),
   description="Build and make all (-j10 -k)")
```
Continuous Integration: How can it help?

Buildbot - Customisation example

Subclassing ShellCommand

class setEnvironmentStep(ShellCommand):
def __init__(self, command='', **kwargs):
    if Project == "tdaq-common":
        kwargs["command"] = ['bash', '-c', 'source %s ; source %s ; %s' % (environmentString, tdaqSetup, command)]
        ShellCommand.__init__(self, **kwargs)
    elif Project == "dqm-common":
        kwargs["command"] = ['bash', '-c', 'source %s ; source %s ; %s' % (environmentString, dqmSetup, command)]
        ShellCommand.__init__(self, **kwargs)

Buildbot - Waterfall

- A couple of different views
  - Waterfall
  - Grid
  - T-Grid
  - Console
  - BuildSlaves

CI Tools

Above examples show that different CI tools can have different approaches to same problem:
  - Jenkins is good, very user friendly etc.
  - Makes use of a plugins model
  - Buildbot is an attempt at “something new”
  - Can design own interface to suit your needs
  - You might need to change less when converting a project to CI
  - Very configurable

Keep these things in mind when selecting CI tool!
Conclusions

- Remember The Challenge?
  - As project sizes grow, so too does complexity
    - The risk for errors in code increases
    - The difficulty of integrating changes increases
  - Continuous Integration solves this!
    - Early feedback, automation, reduce release time etc.
- How can we CI?
  - Commit, build and test often
- There are many CI tools out there. What fits into your project best?

Sources

- BlackDuck | OpenHub
  - https://www.openhub.net/
- Continuous Integration - Martin Fowler
  - http://www.martinfowler.com/articles/continuousIntegration.html
- Jenkins Continuous Build system - Jesse Bowes
- Jenkins
  - https://jenkins-ci.org/
- Buildbot
  - http://buildbot.net/
- GitLab CI
  - https://about.gitlab.com/gitlab-ci/

Thank you for your attention!
Continuous Delivery and Quality Monitoring

Kamil Henryk Król
CERN

Inverted CERN School of Computing, 29 February – 2 March 2016

Outline

Continuous Delivery

Quality Monitoring

Software challenges

- Team work
  - Multiple people
  - Different backgrounds
  - High turnover
- Growing complexity of project
- Quality degradations

The cost of fixing the defect grows exponentially in time!
Software challenges

Brian Foote
Joseph Yoder
“Big Ball of Mud”

Outline

Continuous Delivery

Quality Monitoring

Continuous Delivery

Continuous Integration

The challenge
Continuous Delivery

Release

- Release == Risk
- Release == Fear
- Release == Working over time - frequent calls
- Release == Working during weekends
- Release == Long preparations
- Release == Boring repeatable tasks

Can we do something with it?

Continuous Delivery

The Agile Manifesto

... 

We follow these principles:

Our highest priority is to satisfy the customer through early and CONTINUOUS DELIVERY of valuable software.

... 

@see: Agile Manifesto

Continuous Delivery

The definition

“Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.”

@Martin Fowler
Continuous Delivery

Principles

- Continuous Delivery needs to be built on the solid Continuous Integration
  - Frequent integration with others - (ideally) one main trunk
  - Confident tests protecting our development
  - Easy rollback should be possible

- Introduce as much automation as possible
  - Test close to the production
  - Test environments and testing approaches should be similar to real life use cases

- Avoid:
  - unofficial releases
  - hot fixes without releasing
  - patches sent via e-mail

  Full history of releases
Continuous Delivery
Why?

• Small release v. big release?
  ▪ Lower the deployment risk
• As a developer, when do you consider the functionality done (completed)?
  ▪ Sense of progress
• How do you validate/test application functionalities?
• Do we work on the right thing?
  ▪ Quick user feedback

“50% or more of functionality delivered is rarely or never used.”
@Jim Highsmith, Adaptive Leadership

RELEASE A - 5 changes
RELEASE B - 1 change

Something goes wrong during the release...

In case of which release is easier to locate the problem?

“I wish we did it the old way”
@noone
Continuous Delivery

How?

- Continuous Delivery is the natural continuation of the Continuous Integration processes

Continuous Integration

Continuous Delivery

- Continuous Delivery can be implemented using most build servers (both open source and commercial tools available)

TeamCity

Jenkins

Bamboo

Travis CI

How?

- Continuous Delivery is not a complete set of tools which needs to be used
- It more a way of thinking about the development!

Do we do it right?

Answer these 3 questions

- Is it easy to deploy?
- Is it easy to rollback?
- Clients not affected?

If you answer yes to all questions, you are doing the Continuous Delivery right!
Continuous Delivery

Summary

- Continuous Delivery can give us high level of confidence
- It imposes the incremental development
  - We build the right thing
  - It's easy to locate problems
  - It helps developers see the progress
  - Clients receive functionalities quickly
- It's fashionable
- It's a standard

Outline

Continuous Delivery

Quality Monitoring

Quality?

“the standard of something as measured against other things of a similar kind; the degree of excellence of something”

Monitor? Monitoring?

“observe and check the progress or quality of (something) over a period of time; keep under systematic review”
Quality Monitoring

How do we check the quality?

- **Manual**
  - Manual testing
  - Conventions within the team
  - Definition of done
  - Pair programming
  - Code reviews

- **Automatized**
  - Good coverage of good tests
  - Static code analysis

Manual testing

- Tests which are carried manually by a person or the testing team

  - Easy to define

  - Expensive to run on daily basis
  - Prone to human errors
  - Boring ...

Code conventions

- Common way of producing the code
- Shared within the team / within the organisation
- Should be as close as possible to industry standards
Quality Monitoring

Code conventions

- Naming conventions
  - Variables
  - Constants
  - Classes
  - Enums etc...
- Method parameters conventions
- Layout conventions
- Project structure conventions
- Documentation conventions

Examples:
- Java Code Conventions
- Python Style Guide
- C++ Programming Style Guidelines

```java
private static final int MULTIPLIER = 5;
private String userName;

private void createWelcomeMessage(String name) {
    /* Query the database for the users. */
    database.query("SELECT user FROM users"
        + " WHERE name = "$1", name);
}
```

Definition of done

Definition of Done:

Task: => final-review column
- Unit tested
- Green builds
- Sonar metrics improved for all modified files
- A review was created

Story: => done column
- Survives a non-coder test
- Feature covered by automatic acceptance tests
- Features has been PO approved
- All task reviews have been closed
Quality Monitoring

Pair programming

- 2 people - 2 different opinions
- Driver + Navigator
- Often change roles

- Immediate feedback
- On-spot bug fixes
- Synergy effect

Quality Monitoring

Code reviews

- Systematic review with
  - Co-worker
  - Team
  - External people

- Face 2 Face v. ‘Online’

- Fresh mind reviewers
- Improvement ideas
- Great way of sharing knowledge

Files under review

Review participants

Review objectives

Code snippets

Discussions
Quality Monitoring

Good coverage of good tests

- We can (should) test at different levels
  - Unit testing
  - Integration testing
  - Acceptance testing
  - System testing
- Automatic tests
  - Quick
  - Repeatable
  - Easy to maintain
  - Need to be executed continuously

Legend:
- Instruction covered
- Instruction partially covered
- Instruction uncovered

Quality Monitoring

Static code analysis

Tools which help you to find bugs

Are there any?
- Static ≠ without running software
- Can (or even should!) be automatized
- Find errors which are hard to spot by a human
- Easy shortcut to the better quality
Quality Monitoring

Static code analysis

- Code quality issues
- Test coverage
- Complexity metrics
- Project statistics

Quality Monitoring

- Static code analysis
- Life demo time!
  - You can play on your own at: http://sonar.cern.ch

Quality Monitoring - Summary

- Code is read much more often than it is written!
- Keep an eye on the code quality - continuous monitoring can help you a lot.
- Variety of tools available - choose your own set and take advantage of them.
Thanks a lot!

Any questions?
Massively parallel scientific computing

- processing data on large numbers of processors
- grid computing
  - distributed nodes, often heterogeneous
- clusters
  - nodes usually in the same data center
  - fast interconnections (10G Ethernet, Infiniband,...)
  - mostly homogeneous
  - \(~ 10 \text{--} 10,000\) of many-core nodes (K-Computer: 80,000 nodes)
  - \(~ 1,000 \text{--} 1,000,000\) of CPU cores (Tianhe-2: 3,120,000 cores)
- need for efficient scalable algorithms

Lecture series overview

- techniques for writing algorithms for physics computing in modern C++
- we will strive to produce code which is
  - easy to read
  - efficient
  - modular
- we will introduce some specific methods to
  1) enable natural syntax for mathematical operations
  2) tap into the CPU's vector processing capabilities
  3) process large data sets
- and explore their inner workings

In this lecture

**natural syntax for efficient mathematical operations**

- traditionally, efficient code was “low-level”, hard to read
- object-oriented paradigm leads to readable, but often inefficient code
- we can bridge the gap keeping the best of both worlds by utilizing the C++ templating subsystem
- the technique is called Expression Templates (ETs)
- simply put: ETs are pieces of code closely resembling “natural” mathematical expressions which are translated into low-level code according to user-specified rules during compile time before the machine code is emitted
In following lectures

**Tomorrow: Vectorization with Expression Templates**
- the importance and basic elements of SIMD vectorization
- applying the ET idiom to achieve vectorization while keeping the code nice and clean

**Wednesday: Templates for Iteration; Thread-level Parallelism**
- separating the concepts of “iteration” and “computation”
- utilizing the separation to easily introduce parallelization
- practical example of integrating the Maxwell's equations

---

**C++ Templating**

- a method of generic programming
- allow one function (or class) to operate on different data types without having to be declared for all of them

```cpp
void f(int a) {
    cout << sizeof(a);
};
void f(double a) {
    cout << sizeof(a);
};
void f(std::array<float, 3> a) {
    cout << sizeof(a);
};
void f(MyClass a) {
    cout << sizeof(a);
};
```

---

**C++ Template syntax**

- emitted machine code is the same in both cases
- templated version clearly wins in maintainability

```cpp
// without templates
int i;
double d;
std::array<float, 3> a;
MyClass m;

f(i);
f(d);
f(A);
f(M);
```

```cpp
// with templates

```
**Expression Templates**

### C++ Template syntax

**Declaration**
```
template <typename T>
T mul2 (T a) { return 2*a; }
```

**Specialization**
```
template <>
int mul2(int a) { return (a << 1); }
```

**Instantiation**
```
mul2( 4 );
mul2( 6.33 );
mul2<double>( 4 );
```

### Function template

```
template <typename T, int N>
class Pizza {
    T topping;
    N diameter;
};
```

```
Pizza<Anchovies, 35> thepizza;
```

```
typedef Pizza<Anchovies, 42> L_Romana;
```

```
template<int N>
using Romana = Pizza<Anchovies, N>;
```

**Data**

**In physics**
- scalars
- vectors
- matrices
- scalar fields
- vector fields
- tensor fields
- ...

**In a computer**
- one place in memory
- a place in memory and a length
- several places in memory
- float
- double[]
- std::list<double>
- std::array<std::vector<double>>
- ...

### Algorithms

- An algorithm is a sequence of operations
- In our case (computer simulations in physics)
  - manipulating numbers
  - in computer's memory
- In this lecture (C++ 11)
  - operations: functions, operators, methods, ...
  - numbers: “arithmetic types” - int, float, double, ...

1) C++11 Standard, Chapter 3.9.1

### Traditional “low-level” implementation

**Good performance**
- if written correctly

**Handwritten**
- often hard to maintain

**Implementation**
- obfuscates the meaning

### Q: What is this code doing?

```
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j){
        a[i] += A[i][j] * x[j];
    }
}
c[0] = a[1]*b[2] - a[2]*b[1];
c[1] = a[2]*b[0] - a[0]*b[2];
c[2] = a[0]*b[1] - a[1]*b[0];
q = c[0]*c[0] + c[1]*c[1] + c[2]*c[2];
```

Q: What is this code doing? A:

```
q = ||( A \times b ||
```

A. 27

iCSC 2016, Jiří Vyskočil, CTU Prague – FNSPE

CERN, 29 Feb - 2 Mar 2016
Motivation

what we read in a paper

$\vec{c} = A \cdot \vec{x}$

what we traditionally had to code

```cpp
double x[3];
double A[3][3];
double c[3];

for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        c[i] += A[i][j] * x[j];
    }
}
```

what would we want to code

```cpp
Vector x;
Matrix A;
Vector c = A * x;
```

and we want it to run fast!

Object-Oriented approach to the rescue

• vector addition
  ```cpp
template <int N> class Vector {
    std::array<double, N> v;
    // constructors, copy constructor, assignment operator...
  };
```

• vectors (position, velocity)
• define as objects
• use a simple internal data structure
• overloaded operator+
  ```cpp
template <int N>
class Vector {
    std::array<double, N> v;

    Vector(initializer_list<double>& in) {
        copy(begin(in), end(in), begin(v));
    }

    Vector(Vector& b) {
        copy(begin(b.v), end(b.v), begin(v));
    }

    Vector operator+ (Vector& b) {
        Vector<N> q;
        for (int i = 0; i<N; ++i)
            q.v[i] = v[i] + b.v[i];
        return q;
    }
};
```

OO performance penalty

• let’s run a simple program using our OO vector
• and compare its performance to C-like low-level code

<table>
<thead>
<tr>
<th></th>
<th>Vector&lt;3&gt; result</th>
</tr>
</thead>
</table>
|                | for(int n=0; n<N; ++n){
|                |    Vector<3> d(n, n+1, n+2);
|                |    result = result + d;
|                | } |

<table>
<thead>
<tr>
<th></th>
<th>double result[3];</th>
</tr>
</thead>
</table>
|                | for(int n=0; n<N; ++n){
|                |    double d[3] = {n, n+1, n+2};
|                |    for (int i=0; i<3; ++i) {
|                |        result[i] = result[i] + d[i];
|                |    }
|                | } |

• OO vector: 3.9 s
• low-level: 1.5 s

What happened? Temporary objects!

• performance hit due to temporary object construction
  ```cpp
  Vector a = b + c;
  // is in fact:
  temporary = b + c;
  a = temporary; // a(temporary)
  ```

• we have to construct a new object to hold the temporary result of every operation
  ```cpp
  Vector r = a + b + c + d;
  // it gets worse:
  Vector r = a + b + c + d;
  // is:
  temp1 = a + b;
  temp2 = temp1 + c;
  temp3 = temp2 + d;
  r = temp3; // r(temp3)
  ```

★ only 3 with mandatory copy-elision optimization

* A. 28

CERN, 29 Feb - 2 Mar 2016
**Template Metaprogramming (TMP)**

- **C++ templates conceived to support generic programming**
  - a single function declaration "works" on many types

- **TMP is a technique for using the C++ template system to algorithmically generate source code at compile-time**

- **TMP was “discovered accidentally”**
  - first "metaprogram" computed primes as error messages in 1994

1) [http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html#Static-metaprogramming](http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html#Static-metaprogramming)

**A classic TMP example**

- **example metaprogram: calculating the factorial**
- **evaluated at compile time**

```cpp
template <int i> struct Factorial
{
  static const int value = i * Factorial<i - 1>::value;
};
template <> struct Factorial<0>
{
  static const int value = 1;
};

int main()
{
  return Factorial<5>::value;
}
```

- **recursive formula**
- **termination clause**
- **no runtime computation, result is hard-coded in the executable**


**Back to our example**

- **object’s operators are part of its interface**
- **we want to**
  - keep the OO interface
  - get rid of the OO overhead
- **Expression Templates (ET)**
  - high-level syntax for the algorithm
  - use TMP to generate its low-level implementation

1) Todd L. Veldhuizen: C++ Templates are Turing Complete (2003)
C++11 and C++14 recap

- **auto**
  - variable type automatically deduced from its initializer
  - function return type automatically deduced from the type of the variable in the return statement
- **decltype( )**
  - returns the name of the type of a variable or expression you pass to it
  - slightly different deduction rules than auto
- **decltype(auto)**
  - automatic type deduction using decltype rules
  - C++14, works on type-only expressions
  - can be used for types returned from templates

There's much more behind the scenes, but let's keep it at this today.

Building an Expression Template

- we want to keep the OO interface, and use TMP to generate efficient low-level code
- strategy:
  1) form an abstract tree representation of the expression
  2) possibly simplify / modify
  3) evaluate (lazy)
    i.e. "not until it's needed"

```
// expressions
A*x - b;
q * (E + cross(v, B));
```

```
// statements – evaluation needed
c = A*x - b;
Vector r = q * (E + cross(v, B));
```

Abstract Syntax Trees

- expression (arithmetic in our case) represented as a tree
- operations are nodes
- data (numbers, vectors,...) are leaves

```
a + b * 2
```

```
2 * a + b * (p * q)
```
Expression Templates

Abstract Syntax Trees and ETs

- arithmetic expression $a + b \times 2$
- is built of template type nodes:
  - vector_expression<LeftExp, BinaryOp, RightExp>
  - forms the following syntax tree:
    - $a + b \times 2$
    - which is actually a single expression of the type:
      - vector_expression<vector, AddOp, vector_expression<vector, MulOp, double> >

Tree node in C++

- node’s type is decided on instantiation, and is composed of the LeftExp, BinaryOp, and RightExp.
- variables l and r link to subexpressions – other nodes, or leaves depending on their type.
- we pass the subexpressions to the node’s constructor:
  - class vector_expression {
    - LeftExp l,
    - BinaryOp op,
    - RightExp r;
  }
  - public:
  - vector_expression(LeftExp l, RightExp r) {
    - l = l;
    - r = r;
  }

Creating the Expression

- define the addition operator for the vector_expression class:
  - template <typename RE>
    - auto operator+ (RE&& other) const ->
      - decltype(auto)
        - return
          - vector_expression<
            - vector_expression<LeftExp, BinaryOp, RightExp> const&,
            - BinaryOp,
            - decltype(std::forward<RE>(other))
          > (*this,
            - std::forward<RE>(other))
    - }
  - define the addition operator for the vector_expression class:
    - template <typename RE>
      - auto operator+ (RE&& other) const ->
        - decltype(auto)
          - return
            - vector_expression<
              - vector_expression<LeftExp, BinaryOp, RightExp> const&,
              - BinaryOp,
              - decltype(std::forward<RE>(other))
            > (*this,
              - std::forward<RE>(other))
      - }

- we are adding the other expression to the current expression:
- C++14 syntax saves a lot of typing here:
- the result of the operation is a new expression.
Creating the Expression

- The result of the operation is a new expression.
- This new expression has two branches, each of them linking to some subexpression.

```cpp
template<typename RE>
auto operator+ (RE&& other) const ->
decltype(auto)
{
    return
    vector_expression<
        vector_expression<LeftExp,BinaryOp,RightExp> const&,
        BinaryOp,
        decltype(std::forward<RE>(other))
    >( *this,
        std::forward<RE>(other) )
    ;
}
```

- The two subexpressions will be connected via specified binary operation (addition in this case).
- The operation will become the part of the resulting expressions type signature.

```cpp
template<typename RE>
auto operator+ (RE&& other) const ->
decltype(auto)
{
    return
    vector_expression<
        vector_expression<LeftExp,BinaryOp,RightExp> const&,
        BinaryOp,
        decltype(std::forward<RE>(other))
    >( *this,
        std::forward<RE>(other) )
    ;
}
```

- Left-hand side of the new expression will be the current expression.

```cpp
template<typename RE>
auto operator+ (RE&& other) const ->
decltype(auto)
{
    return
    vector_expression<
        vector_expression<LeftExp,BinaryOp,RightExp> const&,
        BinaryOp,
        decltype(std::forward<RE>(other))
    >( *this,
        std::forward<RE>(other) )
    ;
}
```

- RE is whatever we got on the right side of the + operator.
  - It might be a vector, or another expression.
  - We need to preserve its fully qualified type.

```cpp
template<typename RE>
auto operator+ (RE&& other) const ->
decltype(auto)
{
    return
    vector_expression<
        vector_expression<LeftExp,BinaryOp,RightExp> const&,
        BinaryOp,
        decltype(std::forward<RE>(other))
    >( *this,
        std::forward<RE>(other) )
    ;
}
```
Creating the Expression

- chaining operators creates the AST at compile time
- the `vector` class, which holds the data (we'll implement it later) has a similar operator+ with `vector` as LeftExp type, and `AddOp<double>` as BinaryOp

```cpp
template <typename RE>
auto operator+ (RE&& other) const ->
    decltype(auto)
{
    return
    vector_expression<
        vector_expression<LeftExp, BinaryOp, RightExp> const&,
        BinaryOp,
        decltype(std::forward<RE>(other))
    >(*this,
        std::forward<RE>(other))
;
}
```

Lazy evaluation

- defer evaluation until the result is needed
- expression can be further manipulated (simplifications, ...)
- need an evaluation trigger
- only when we really have to evaluate
  - assignment using operator =
  - variable constructor
  - argument to an operator/function not included in our ET class
- the types of the whole statement are:

```
x  =  a  +  b  +  c;
vector  =  vector_expression< vector, AddOp, vector_expression<vector, AddOp, vector> >
```
Evaluating the Expression I

The assignment operator must trigger the evaluation because it will store the result.

We tell the compiler to replace the operator= by the loop, which might also get unrolled.

We're basically casting vector_expression to vector while calling operator[] on the vector_expression object.

Still compile-time.

template <int N>
class vector {
    template <typename RightExpr>
    vector& operator=(RightExpr&& re) {
        for (int i = 0; i < N; ++i)
            v[i] = re[i];
        return *this;
    }
};

We know how to formally convert vector_expression into a vector, but how do we calculate the actual numbers which form the result?

```
vector_expression< vector, AddOp, vector_expression<vector, AddOp, vector> >
vector = ...;
```

Evaluating the Expression II

- operator[] applies the specified BinaryOp on the subexpressions’ i-th element.
- operator[] recursively propagates through the tree.
- When the function reaches a leaf (a vector), operator[] returns the i-th component.
- Evaluation itself can be done using a static function which implements the binary operation.

```
template<RightExpr,BinaryOp,LeftExpr>
class vector_expression {
    auto operator[](int i) const -> decltype(auto) {
        return BinaryOp::apply(l[i], r[i]);
    }
};
```

```
template <typename T>
struct AddOp {
    static T apply(T const& a, T const& b) {
        return a + b;
    }
};
```

Evaluating the Expression III

- Assignment operator triggers the cast from vector_expression to vector.
- Which is performed by recursively applying operator[] in a loop to subexpressions connected by operator+.
- Thus the compiler translates the statement \( x = a + b + c \), where the expression \( a + b + c \) is represented by the syntax tree.

```
+  
/ 
|  
+  
a  +  b  +  c
```

- Into the following code:

```
for(int i=0; i<3; ++i) {
    x[i] = a[i] + b[i] + c[i];
}
```

Generated assembly

- The ET code reads as naturally as the OO code.
- A good compiler will emit the same instructions for well-written ET code, as in a hand-coded version.

<table>
<thead>
<tr>
<th>Simple OO</th>
<th>15 instructions in the loop</th>
<th>3.9 s time spent in the loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr templ.</td>
<td>8 instructions in the loop</td>
<td>1.5 s time spent in the loop</td>
</tr>
<tr>
<td>Hand-coded</td>
<td>8 instructions in the loop</td>
<td>1.5 s time spent in the loop</td>
</tr>
</tbody>
</table>

Vector<3> result;
for(int n=0; n<N; ++n){
Vector<3> d(n, n+1, n+2);
result = result + d;
}

double result[3];
for(int n=0; n<N; ++n){
    double d[3] = {n, n+1, n+2};
    for (int i=0; i<3; ++i) {
        result[n][i] = result[n][i] + d[n][i];
    }
}
Expression transformation / simplification

- expressions can be simplified or otherwise transformed before evaluation
- templates are Turing-complete
  - any transformation algorithm might be employed
- be aware of potential dangers of floating-point arithmetic

1) David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic

Transformation example I

- our evaluation trigger might first perform a transformation
- the result of the transformation will be a vector_expression
- the transformation is performed at compile-time
- remember the trigger is vector::operator=

```cpp
// same as before
template <int N>
class vector {
  template <typename RightExpr>
  vector& operator=(RightExpr&& re) {
    // the Transform functions returns
    // a vector_expression, no cast is
    // done here, no runtime computation
    auto transformed = Transform(re);
    for (int i = 0; i < N; ++i)
      v[i] = transformed[i];
    return *this;
  }
};
```

Transformation example II

- this example recursively changes all operations in the AST to subtraction (not very useful)
- the single-parameter specialization will stop the recursion in case the parameter is not a vector expression
- more precisely – we stop the recursion in the general case, and only recurse if we do have a vector_expression

```cpp
// this is here to stop the recursion
template<
  typename Expr>
auto Transform(const Expr& expr) -> Expr {
  return expr;
}
```

```cpp
// only instantiated for vector_expression type
template<
  LE,
  BinaryOp,
  RE>
auto
Transform(
  vector_expression<
    LE,
    BinaryOp,
    RE>
  const& expr)
-> decltype(auto) {
  return Transform(
    vector_expression<
      LE,
      SubtractOp,
      RE>
    (expr.l, expr.r));
}
```

Input

Output

some details omitted

Algorithm specialization

- some operations might benefit from a specialized algorithm
- create a partial specialization of the vector_expression template
- you can even offload the computation to a library routine

```cpp
template < typename RightExpr >
class vector_expression<
  Matrix, MulOp, RightExpr>
>
{
  ...
  template < typename RE >
  auto operator*(RE& other) const ->
  decltype(auto) {
    result = ....; // run our algorithm
    return result;
  }
};
```
RVO, copy elision, C++11 moves

- Modern compilers can be very smart about optimizations
- Return-Value-Optimization and copy-elision together with additional optimizations turned on by -O3 might remove calls to most superfluous constructors
- Used together with C++11 move semantics, the performance of simple OO expressions might be on par with ETs
  - This is by no means guaranteed
  - Different compilers have vastly different results
  - It’s difficult to argue about more complex cases

<table>
<thead>
<tr>
<th>version</th>
<th>gcc 5.3.1</th>
<th>icc 16.0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple OO</td>
<td>2.1 s</td>
<td>5.4 s</td>
</tr>
<tr>
<td>expr. templ.</td>
<td>2.0 s</td>
<td>1.3 s</td>
</tr>
<tr>
<td>hand-coded</td>
<td>2.0 s</td>
<td>1.3 s</td>
</tr>
</tbody>
</table>

Potential pitfalls

- Simple ETs without appropriate specialized algorithms might be under-performing
- Performance depends on the compiler
  - Be careful when using complex libraries with lots of TMP
  - Test, measure, profile
- Proper inlining is of particular concern
  - More about inlining in lecture 3

Take Away Messages

- Expression Templates allow you to use object-like syntax without the inherent performance penalty
- ETs construct an Abstract Syntax Tree representing your “mathematical expression”
- The AST can be arbitrarily manipulated – the templating system is Turing-complete
- “Lazy evaluation” is triggered by assignment
- Evaluated expressions translate into “low-level” code according to the rules implemented by the ET engine
  - A good ET engine produces efficient low-level code

Thank you for your attention!
Multivariate Classification and Machine Learning in HEP

Lecture 1

Traditional multivariate classification

Thomas Keck
IEKP KIT

Inverted CERN School of Computing, 29 February – 2 March 2016

Outline

- Neyman-Pearson Lemma / Supervised learning
- Discriminant Analysis / Analytical methods
- Decision Trees / Model complexity
- Boosted Decision Trees / Ensemble methods
- Support Vector Machines / Kernel trick
- Sideband subtraction & sPlot / Data-driven techniques
- Available tools

Supervised multivariate classification
Formal Problem

Given \( \mathbf{x}_i \in \mathbb{R}^D \quad i = 1 \ldots N \)

Searched \( f : \mathbb{R}^D \rightarrow \{0, 1\} \)

Background PDF

Signal PDF

PDF (Probability Density Function) see CSC 2015 Data analysis lecture 3

---

Neyman Pearson Lemma

DOI: 10.1098/rsta.1933.0009

IX. On the Problem of the most Efficient Tests of Statistical Hypotheses.

By J. Neyman, Nencki Institute, Soc. Sci. Lit. Varsoviensis, and Lecturer at the Central College of Agriculture, Warsaw, and E. S. Pearson, Department of Applied Statistics, University College, London.

\[
f(\mathbf{x}) = \frac{PDF(\mathbf{x}|S)}{PDF(\mathbf{x}|B)}
\]

Most powerful test to distinguish signal and background

---

Contour Plot of NP-Classifier

Classification Quality

Signal
Background
Type I Error
Type II Error

Background (Classifier)
Signal (Classifier)
Classification Quality

Receiver Operating Characteristic

Multivariate Classification in HEP

A $z$-Vertex Trigger for Belle II

Allowed region for an arbitrary classifier

Neyman Pearson Lemma AUC = 0.934
Multivariate Classification in HEP

- Signal and Background PDF is unknown
  - Multiple sources for signal and background
  - Non Gaussian
  - Nonlinear dependencies among features
- Medium number of dimensions/features $D \approx O(100)$
- Curse of dimensionality $\rightarrow$ PDFs cannot be sampled
- High number of samples/events $N \approx O(1 \, 000 \, 000)$
  - Training and Application time matters
  - Often simulated data ($\rightarrow$ systematics)

Classifier Models

Neyman Pearson Lemma

$$ f(\mathbf{x}) = \frac{PDF(\mathbf{x}|S)}{PDF(\mathbf{x}|B)} $$

Generative Models

- Restricted Boltzmann machine
- Kernel density estimator
- Gaussian mixture model

Discriminative Models

- Linear discriminant analysis
- Quadratic discriminant analysis
- (Boosted) Decision Trees
- Support Vector machine
- Artificial neural networks

Discriminant Analysis

Analytical / closed-form solutions of Neyman-Pearson Lemma

Linear discriminant analysis (LDA)

- Assumes conditional PDFs are normally distributed
- Assumes identical covariances of signal and background
- Equivalent to commonly used Fisher's discriminant
- Requires only means and covariances of sample
- Separating hyperplane is linear

Quadratic discriminant analysis (QDA)

- Assumes conditional PDFs are normally distributed
- Requires only means and covariances of sample
- Separating hyperplane is quadratic
Linear Discriminant Analysis

Quadratic Discriminant Analysis

Discriminant Analysis

Decision Trees / Model complexity
Simple Decision Tree

Complete Decision Tree

Complex model performs poorly
**Overfitting**

- Model is too complex
- Statistical fluctuations in the training data dominate predictions
- Model does not generalize → poor performance on new data

**Underfitting**

- Model is too simple
- Relevant aspects of the data are ignored

**Model Complexity**

Hyperparameters control the model complexity

- Maximal depth of a decision tree
- Number of trees in a boosted decision tree
- Number of hidden neurons in an artificial neural network
- Soft margin parameter in a support vector machine

Optimize hyperparameter to get best performance on new data

- Grid-Search using additional test-sample or cross-validation
- Bayesian methods can automatically determine optimal value

Always test on independent validation sample in the end!
Model Complexity

Regularisation methods reduce effective model complexity
- Pruning of a decision tree or neural network
- Weight decay in an artificial neural network
- Ensemble methods

Boosted Decision Trees / Ensemble methods

Ensemble methods

\[ F'(\vec{x}) = \sum_m \gamma_m h_m(\vec{x}) \]

Idea: Average many simple models to obtain a robust complex model

Boosting

\[ F_m(\vec{x}) = F_{m-1}(\vec{x}) + \arg\min_h \sum_{i=1}^{N} L(y_i, F_{m-1}(\vec{x}_i) - h(\vec{x}_i)) \]

- Reweight events according to current prediction
- Individual classifiers are simple to avoid overfitting (weak-learners)
- Concentrate on events near the optimal separation hyper-plane
- Loss Function L is crucial
  - Least square \rightarrow Regression
  - Binomial deviance \rightarrow Gradient Boost Classification
  - Exponential loss \rightarrow AdaBoost Classification
Bagging

- Bagging – Use only fraction of events / features per classifier
- Robustness against statistical fluctuations in the data
- Sampling Method is crucial:
  - Draw random events with replacement → Bagging
  - Draw random events without replacement → Pasting
  - Draw random features → Random Subspaces

Stochastic Boosted Decision Trees

- BDTs perform really well out of the box!
- Categorisation – Divide feature-space into sub-spaces
  - Different mechanisms and processes in the chosen subspaces
    e.g. train separate classifiers for Barrel and Endcap regions
- Combination – Combine different classifier
  - Different regularisation methods learn different aspects of the data
    e.g. Combine neural network, boosted decision tree and SVM
Support Vector Machines / Kernel trick

SVM

Linear SVM

Kernel Trick

SVM Algorithm depends only on scalar product!

\[ \max f(c_1, \ldots, c_n) = \sum_i c_i - \frac{1}{2} \sum_i \sum_j y_i c_i (\vec{x}_i \cdot \vec{x}_j) y_j c_j \]

Replace scalar product with an arbitrary kernel function

- Poly: \[ k(\vec{x}_i, \vec{x}_j) = (\vec{x}_i \cdot \vec{x}_j)^d \]
- RBF: \[ k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma ||\vec{x}_i - \vec{x}_j||^2) \]
Multivariate Classification and Machine Learning in HEP

RBF SVM

Poly SVM

Performance strongly depends on kernel

Sideband subtraction & sPlot / Data-driven techniques
Data driven techniques

- Discriminating variable
- Control variable
- Control variable
- Control variable

Side-band Subtraction

- Requires # of Signal Events in Signal Region
- Compensates Background Events in the Signal Region with negative Signal Events from the Sideband

sPlot

- Requires yields and covariance of fitted signal + background models
- Uses every event twice, as signal and as background with the correct weight

Classification Quality

Use data-driven techniques if you don't trust you simulated data
Reweighting

- Requires MC and data events
- Trains classifier to distinguish data events and MC events
- Reweighted MC Events using output of classifier
- Trains classifier to distinguish signal and background using reweighted MC

Available tools

Python, scipy and sklearn

C++, ROOT and TMVA

Redesign: http://oproject.org/TMVA
HEP vs. MML community

Outline

- Artificial neural networks
- Highlights
- Driving factors of Deep Learning
- Convolutional neural network
- Recurrent neural network
- Available tools
Artificial neural networks

**Feed Forward**

\[ f(\vec{x}) = \sigma \left( \sum_j w_{j}^{\text{hid}} \sigma \left( \sum_i w_{i}^{\text{inp}} x_i \right) \right) \]

\[ \sigma = \frac{1}{1 + e^{-x}} \]

**Backpropagation**

\[ \Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}} \]
Artificial neural networks

- Perform as well as BDTs
- Typically more difficult to tune

Neyman Pearson Lemma AUC = 0.934
Multi Layer Perceptron AUC = 0.921

Data acquisition
- Experiment
- Simulation
- 3rd party

Data preparation
- Calibration
- Remove outliers
- Suppress noise

Data preprocessing
- Feature extraction
- Data reduction

Workflow
- Data acquisition
- Experiment
- Simulation
- 3rd party

Data preparation
- Calibration
- Remove outliers
- Suppress noise

Data preprocessing
- Feature extraction
- Data reduction

Deep Learning

Idea: Feed in low level features
- Feature extraction is done automatically

Deep Learning

Feasible NNs
Deep learning
Neural Turing Machines
Memory Chips

Luke de Oliveira

CERN, 29 Feb - 2 Mar 2016
Multivariate Classification and Machine Learning in HEP

**Highlights**

- Deep neural network outperforms BDTs
  - DN 0.88
  - BDT 0.81

- Extract information using convolution neural network
- Generate description using recurrent neural network

**Table 1 | Performance for Higgs benchmark.**

<table>
<thead>
<tr>
<th>Technique</th>
<th>Low-level</th>
<th>High-level</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BDT</td>
<td>0.73 (0.09)</td>
<td>0.78 (0.03)</td>
<td>0.81 (0.01)</td>
</tr>
<tr>
<td>NN</td>
<td>0.793 (0.002)</td>
<td>0.777 (0.002)</td>
<td>0.886 (0.0043)</td>
</tr>
<tr>
<td>DN</td>
<td>0.880 (0.001)</td>
<td>0.800 (&lt;0.001)</td>
<td>0.885 (0.002)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Discovery significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN</td>
</tr>
<tr>
<td>2.5σ</td>
</tr>
<tr>
<td>3.1σ</td>
</tr>
<tr>
<td>3.7σ</td>
</tr>
<tr>
<td>DN</td>
</tr>
<tr>
<td>4.9σ</td>
</tr>
<tr>
<td>3.6σ</td>
</tr>
<tr>
<td>5.0σ</td>
</tr>
</tbody>
</table>

- Where is the ring? A: Mount-Doom
- Where is Bilbo now? A: Grey-havens
- Where is Frodo now? A: Shire

Receiving only the pixels and the game score as inputs, achieved a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters.
2. GPU Acceleration

Deep Learning needs a lot of computation time!

3. New Deep Learning Techniques

- Long Short-Term Memory
- Restricted Boltzmann Machines
- Convolutional Neural Network

Tasks

When a user takes a photo, the app should check whether they're in a national park: sure, they've looked up gimmie a few hours... and check whether the photo is of a bird. I'll need a research term and five years.

In CS, it can be hard to explain the difference between the easy and the virtually impossible.
Problem

Lots of different birds in different poses, scales, positions!

Invariance under Transformations

- We can think of different strategies to build a classifier which is invariant under certain transformations of its input
- Use copies of same image with arbitrary translation, rotation and scaling during the training phase
- Add term to Loss-Function that penalizes changes in the model output when input is transformed → Tangent propagation
- Extract hand-crafted features that are invariant under the transformation
- Build invariance properties into structure of neural network → local receptive fields, shared weights and subsampling

Convolutional Neural Network

- Convolution layer: Organized in planes called feature maps
  - Learnable filters (e.g. edge detector)
  - All units take inputs only from small subregion of the image
  - All units are constrained to share the same weights
  → All units detect same pattern but in different locations

- Pooling Layer:
  - Take inputs from small receptive field in the feature maps
  - Perform subsampling by weighted average + activation function
  →Insensitive against small shifts
Convolutional Neural Network
http://parkorbird.flickr.com/

- Architecture: Multiple pairs of convolution + pooling layers
  - Each stage has a larger degree of invariance to input transformations
  - Gradual reduction in resolution compensated by increasing number of features
  - Final layer typically fully connected with softmax output for multiclass classification

Limitations

- Fixed input space dimension
  - How to process sequential data like speech or text?
  - Computationally expensive because the amount of computation scales linearly with the number of image pixels

Recurrent Neural Network
Long Short-Term Memory

- Can remember a value for a long time period
- Input gate decides when to update the stored value
- Output gate decides when to output the stored value
- Forget gate decides when to forget the stored value
→ Can process sequential data (e.g., text and speech)

Character level language model

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Applied on C-Code

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

- Linux Kernel Code on Github (474 MB)
- 3-layer LSTMs (~10 million parameters)
- Few days training time on graphic card
- Learned syntax, but not semantic

Cell that activates inside if statements

Recurrent Models of Visual Attention


- Scans image using recurrent neural network
- Trained using reinforcement learning methods
- Attention models successfully employed for handwriting recognition
Available Tools

- Theano
- Decaffe / Caffe
- Torch
- Kaldi
- Tensorflow
Introduction to FV

Kim Albertsson
Luleå University of Technology

Inverted CERN School of Computing, 29 February – 2 March 2016

Introduction

- Kim Albertsson
  - M.Sc. Engineering Physics and Electrical Engineering
  - Currently studying for M.Sc. Computer Science
- Research interests
  - Automation
  - Machine learning
- Formal methods
  - Great way to automate tedious tasks

Formal Methods

- Managing complexity
- “…mathematically based languages, techniques, and tools for specifying and verifying … systems.”
  - Edmund M. Clarke et al.
- Reveals inconsistencies and ambiguities

Formal Methods

- Specification + Model + Implementation enables verification
- Proving properties for a system
  - Will my algorithm sort the input?
  - Will my soda pop be delivered in time?
  - Can the LHC interlock system end up in an inconsistent state?
Overview Series

- **Approaches**
  - Model Checking and Theorem Proving
  - Logic and automation
  - How to build an ecosystem

- **Application**
  - Robust code
  - Robust and Efficient code

Introduction Lecture 1

- **Introduction**
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts

- **Methods of Formal Verification**
  - Model Checking
  - Theorem Proving

- **In-depth Theorem Proving**
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms

Introduction Lecture 1

- **Errors discovered late**
  - Expensive

- **FV requires**
  - More effort for specification

- **FV gives**
  - Feedback on inconsistencies
  - Reduces late errors
  - Reduces total time

Overview

- **Introduction**
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts

- **Methods of Formal Verification**
  - Model Checking
  - Theorem Proving

- **In-depth Theorem Proving**
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms
Practical Examples

- Space flight, Ariane 5 - 501
- Avionics, Lockheed

Ariane 5 - 501

- Ariane 5
  - Reused inertial reference system of Ariane 4
  - Greater horizontal acceleration
  - 64-bit float cast to 16-bit integer overflowed
- Expensive bug discovery

Lockheed C130J

- C130J Hercules
  - Secure low-level flight control code
- Review concluded
  - Improved quality of product
  - Decreased cost of development

Specification

- Concise, high level description of behaviour
- Varying degree of rigour
  - Informal; written English
  - Formal; using standardised syntax
- Required for verification
A verified program has demonstrated that specification is consistent with implementation.

**Specification**

- Example in whyML
  
  ```ml
  let bubblesort (a: array int) =
  requires { ... }
  ensures { sorted a }
  ...
  ```

**Testing vs. Proving**

- **Testing**
  - Limited to particular point in input space
- **Proving**
  - Reason about complete input space

**Contract**

- **A contract consists of**
  - pre-condition, that must hold before execution
  - post-condition, that must hold after execution
- **Weakest precondition**
  - The set of least restrictive of pre-conditions that still ensure the post-condition
System Composition

Overview

- Introduction
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts
- Methods of Formal Verification
  - Model Checking
  - Theorem Proving
- In-depth Theorem Proving
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms

Methods of FV

- Model Checking
  - General tool
  - Tricky problem
- Theorem Proving
  - General tool
  - Tricky problem
Model Checking

- Verifies that property holds for all states
  - Explores all model states
  - Suitable for finite state models
- Program verification
  - Model program as a graph

Pros
- Easy start-up with when applicable

Cons
- Requires clever algorithms
- Must start anew each time

Theorem Proving

- Define a formal logic system
  - With inference rules
- Proofs are derived by applying rules
  - By hand or by machine
- Turns out full automation is a tricky problem
  - Interactive provers, or proof assistants

Overview

- Introduction
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts
- Methods of Formal Verification
  - Model Checking
  - Theorem Proving
- In-depth Theorem Proving
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms
First Order Logic

- **Object variables**
  - Refers to a unique object
  - Names like x, jim, 1

- **Predicates**
  - Relations like =, <, AtHome

- **Connectives**
  - \( \land, \lor, \to, = \)

- **Functions**
  - Complex names
  - mother(jim)
  - head(x)
  - 1+1

- **Sentence**
  - Combination of the above
  - mother(jim) = kimberly
  - head(x) < 1

**Implication elimination**

\[
\frac{a \to b, a}{\therefore b}
\]

**Negation Introduction**

\[
\frac{a \lor b, a \lor \neg b}{\therefore \neg a}
\]

**Conjunction Introduction**

\[
\frac{a, b}{\therefore a \land b}
\]

**Disjunctive Syllogism**

\[
\frac{a \lor b, \neg a}{\therefore b}
\]

Hoare Logic

- **P**, Precondition
- **Q**, Postcondition
- **C**, Command

When C is executed under P, Q is guaranteed

\[
\{x + 1 < N\} x := x + 1 \{x \leq N\}
\]

Rules required for each action of a language
Hoare Logic

- **Composition**
  - Allows commands to be executed in sequence
  \[
  \frac{\{P\} S \{Q\}, \{Q\} T \{R\}}{\{P\} S, T \{R\}}
  \]

- **While rule**
  - Models while-statement
  - P is called loop invariant
  - Can be extended to prove termination

Theories

- **Theory**
  - Set of sentences that are assumed to be true, T

- **Axiom**
  - Each element in T

- **Theorem**
  - Any sentence that can be concluded from the theory

- **Example**
  - Peano arithmetic, theory of lists...

Satisfiability Modulo Theories

- **Hoare logic**
  - To reason about programs
  \[
  \{P\} C \{Q\}
  \]

- **Reasoning expressed**
  - First order logic
  \[
  \{x + 1 < N\} x := x + 1 \{x \leq N\}
  \]

- **Verification conditions (VC)**

- **Satisfiability Modulo Theories**
  \[
  (x \land \neg y) \land (y \land z)
  \]

  - **Similar to Binary Satisfiability problem, SAT**
    - One of the first problems to be shown to be NP-complete
    - Find assignment of x, y, z ... so that the expression is satisfied

  - **SAT variables are boolean**
    - SMTs are extended to handle FOL constructs

  - **Verifying a proof is easy**
    - Check each step for validity assuming our logic system
Satisfiability Modulo Theories

- Formulas are considered w.r.t background theory
  - For formula F, assume ¬F
  - F is valid when ¬F is not satisfiable under T

- Modern solvers use heuristics
  - Improves performance for specific theories
  - At the cost of general performance

Example in alt-ego

Try for yourself https://alt-ergo.ocamlpro.com/try.php

logic x, y, z: prop
axiom a_1: (x and not y) -> (y or z)
axiom a_2: y -> not x
axiom a_3: x
goal g_1: z

# [answer] Valid (0.0060 seconds) (3 steps)

Verification Platforms

- Unified interface for provers
- Generates verification conditions (VCs)
  - Discharged by any compliant prover, interactive or automatic
- Examples
  - why3, boogie

Example

let insertion_sort (a: array int) =
    for i = 1 to length a - 1 do
        let v = a[i] in
        let j = ref i in
        while !j > 0 && a[!j - 1] > v do
            a[!j] <- a[!j - 1];
            j := !j - 1
        done;
        a[!j] <- v
    done
end
Example

```ocaml
let insertion_sort (a: array int) =
  ensures { sorted a }
  for i = 1 to length a - 1 do
    invariant { sorted_sub a 0 i }
    let v = a[i] in
    let j = ref i in
    while !j > 0 && a[!j - 1] > v do
      invariant { 0 <= !j <= i }
      invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                  -> k1 <> !j
                  -> k2 <> !j -> a[k1] <= a[k2] }
      invariant { forall k: int. !j+1 <= k <= i
                  -> v < a[k] }
      a[!j] <- a[!j - 1];
      j := !j - 1
    done;
    a[!j] <- v
  done
end
```

Example

```ocaml
let insertion_sort (a: array int) =
  ensures { sorted a }
  for i = 1 to length a - 1 do
    invariant { sorted_sub a 0 i }
    let v = a[i] in
    let j = ref i in
    while !j > 0 && a[!j - 1] > v do
      invariant { 0 <= !j <= i }
      invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                  -> k1 <> !j
                  -> k2 <> !j -> a[k1] <= a[k2] }
      invariant { forall k: int. !j+1 <= k <= i
                  -> v < a[k] }
      a[!j] <- a[!j - 1];
      j := !j - 1
    done;
    a[!j] <- v
  done
end
```

Example

```ocaml
let insertion_sort (a: array int) =
  ensures { sorted a }
  for i = 1 to length a - 1 do
    invariant { sorted_sub a 0 i }
    let v = a[i] in
    let j = ref i in
    while !j > 0 && a[!j - 1] > v do
      invariant { 0 <= !j <= i }
      invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                  -> k1 <> !j
                  -> k2 <> !j -> a[k1] <= a[k2] }
      invariant { forall k: int. !j+1 <= k <= i
                  -> v < a[k] }
      a[!j] <- a[!j - 1];
      j := !j - 1
    done;
    a[!j] <- v
  done
end
```

Example

```ocaml
let insertion_sort (a: array int) =
  ensures { sorted a }
  for i = 1 to length a - 1 do
    invariant { sorted_sub a 0 i }
    let v = a[i] in
    let j = ref i in
    while !j > 0 && a[!j - 1] > v do
      invariant { 0 <= !j <= i }
      invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                  -> k1 <> !j
                  -> k2 <> !j -> a[k1] <= a[k2] }
      invariant { forall k: int. !j+1 <= k <= i
                  -> v < a[k] }
      a[!j] <- a[!j - 1];
      j := !j - 1
    done;
    a[!j] <- v
  done
end
```

- $0 \leq j \leq i$
- $0 \leq k_1 \leq k_2 \leq i$
- $k_1 \neq j$
- $k_2 \neq j$
- $a[k_1] \leq a[k_2]$
- $\forall k. j + 1 \leq k \leq i \rightarrow v < a[k]$
Example

```ml
let insertion_sort (a: array int) =
  ensures { sorted a }
  for i = 1 to length a - 1 do
    invariant { sorted_sub a 0 i }
    let v = a[i] in
    let j = ref i in
    while !j > 0 && a[!j - 1] > v do
      invariant { 0 <= !j <= i }
      invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i -> k1 <> !j -> k2 <> !j -> a[k1] <= a[k2] }
      invariant { forall k: int. !j+1 <= k <= i -> v < a[k] }
      a[!j] <- a[!j - 1];
      j := !j - 1
    done;
    a[!j] <- v
  done
end
```

Recap of Lecture 1

- **FV manages complexity**
- **Finds errors early**
  - Save money, time and possibly life
- **Proof is a demonstration**
  - Consistency of specification and implementation

Recap of Lecture 1

- **Proving properties about your system**
- **Model Checking**
  - Exhaustive search of state space
- **Theorem Proving**
  - Deductive approach
Thank you
Data Reconstruction in Modern Particle Physics
Daniel Saunders, University of Bristol

About me…

• Particle Physics student, final year.
  • CSC 2014, iCSC 2015, iCSC 2016
• Main research interests.
  • Detector upgrades for LHC - pixel detectors for LHCb.
  • Neutrino experiments at nuclear reactors - data reconstruction and analysis.
• But day to day…
  • Professional ROOT and git complainer.
  • Developing C++ and python projects to perform above.

But it’s not all about me…

• I assume you know a bit about:
  • The LHC.
  • LHC-like particle physics detectors, and that they give information like:
    • Particle energies.
    • Particle paths.

The Tardis prompts questions for the audience.

Aim of these lectures

What just happened?

Example collision event from CMS.
Aim of these lectures
What just happened?

- LHC detectors produce $O(10)$ petabytes of data per year\(^{(1)}\).
- Data is processed to the stage of physics papers $\rightarrow$ measurements and discoveries.

Many steps involved.
- Each step has computing costs, varying inefficiencies, often in large backgrounds.
- We’ll consider some steps in detail, looking at tradeoffs between these three factors.

Example collision event from CMS.

Higgs discovery at CMS.

Lecture Outline - Scope
Aim of these lectures: ‘How to take detector output and make physics measurements.’

Lecture 1
- Introduction and context:
  - Elements of LHC detectors (CMS & LHCb).
  - Data rates and formats.
- Data taking strategies:
  - Triggers.

Lecture 2
- Event reconstruction principles:
  - Tracking.
  - Particle identification.
  - Vertexing.
- Optimisations:
  - Parallelism.
  - The Grid.

Lecture Outline - Beyond Scope

- Particle Physics detectors exist:
  - Able to measure the position of a crossing particle and energy.
  - Measurements are to a limited resolution.
  - Detectors aren’t perfect: can be inefficient, impure and have bad resolution.
- The exact workings of detectors is not considered.
  - Lecture concepts are general and apply to all particle physics experiments.
  - For detector concepts, see:
- Many physics measurements are made by comparing reconstructed data to simulation.
  - A large topic! Many sophisticated data analysis techniques used.
  - We will stop at this point.

Many of these topics are discussed in other lectures of this school.
Introduction and Motivation

- LHC computing is an enormous task:
  - Data rate of typical LHC experiments in run two (i.e. current) ~1GB/s\[2\].
  - In extreme cases (CMS\[3\] & ATLAS), this is post 1 in ~10,000 filter by electronics.
  - Still large background of uninteresting events.

- Collaborations of thousands of scientists:
  - Many kinds of analysis requiring different data selection.
  - Competitive spirit (e.g. ATLAS vs CMS).
  - Many kinds of people (Computer Scientists, Engineers, Physicists etc.) with varying computing skills.

LHC Reminder

- LHC is a proton - proton (or Pb - Pb) collider.
- Each collision is called an event.

LHC Collisions

- Particles are grouped into bunches of ~ 100 billion protons.
- LHC collides two proton beams, circling in opposite directions.
- A collision between two bunches happens once every 25ns (so 40MHz).
- In each collision, ~25 pairs of proton collisions. About 1 per million is used for physics.
  - The rest is well understood background of un-interesting events.

LHC Detectors

- Large dedicated detectors built surrounding four collision points:
  - ATLAS & CMS: general purpose detectors, discovery machines.
  - ALICE: high energy plasma studies, conditions approx 10^-6 seconds post big bang.
  - LHCb: precision antimatter studies.
  - Many other smaller experiments.
- Detectors generally either perform:
  - Direct searches (e.g. CMS) - looking for new phenomena, such as the Higgs Boson.
  - Indirect searches (e.g. LHCb) - compare precision measurements to theoretical predictions. Differences can be a sign of new physics that has been unaccounted.
- Choose CMS and LHCb to look at in more detail.
Calorimeters - measures energy. Some particles stop.

Magnet - bends tracks, allowing for momentum measurement.

Silicon tracker - tracks particles near the collision region.

Muon chambers - detect particles that leave the detector (most likely muons).

CMS slice, with different particle examples.

LHC Detectors - CMS

CMS slice, with different particle examples.

LHC Detectors - LHCb

RICH (Ring Imaging Cherenkov Detector) - particle ID from radiation induced by crossing particles.

The Task

What just happened?

- All LHC computing is connected to data processing.
- Need to take detector output (~1GB/s of electrical signals) and perform:
  - Detector related studies:
    - Online data quality monitoring.
    - Calibrations.
  - Physics analysis:
    - Discovery of new particles.
    - Comparisons with simulation.

Example collision event from CMS.

The Task

What just happened?

- All LHC computing is connected to data processing.
- Need to take detector output (~1GB/s of electrical signals) and perform:
  - Detector related studies:
    - Online data quality monitoring.
    - Calibrations.
  - Physics analysis:
    - Discovery of new particles.
    - Comparisons with simulation.

Example collision event from CMS.
Data Flow

- Data reconstruction generally involves several steps of processing and reduction:

Triggers

- The trigger decides what data to save in real time.
  - Typically just 1 in a million events considered interesting by trigger.
  - Can be formed of several levels, data reduction and increasing complexity at each step.
  - Executed on FPGAs and/or local CPU/GPU processing farms.

  Effcient (aim >80%) to maximise storage and reconstruction resources.
  - All other events permanently discarded.
  - Many O(50) configurations used (a few examples shown later):
    - Tuned to physics analysis of the experiment.
    - Try to be least biased whilst recording interesting events.
The trigger decides what data to save in real time.
- Typically just 1 in a million events considered interesting by trigger.
- Can be formed of several levels, data reduction and increasing complexity at each step.
- Executed on FPGAs and/or local CPU/GPU processing farms.
- Algorithms that perform quickly.
- Efficient (aim >80%) to maximise storage and reconstruction resources.
- All other events permanently discarded.

Many O(50) configurations used (a few examples shown later):
- Tuned to physics analysis of the experiment.
- Try to be least biased whilst recording interesting events.

Triggers Worked Example - LHCb

- Data rate of entire detector too high for all to be used in trigger:
  - Use a subset of information.
  - Introduce multiple levels of triggers – use more information in higher levels.
  - May deliberately reduce resolution of detectors to reduce data size.
  - E.g. combine cells in tracker, or use a less precise data type.
- LHCb model is very efficient, allowing for physics analysis immediately after trigger - not always possible.

Particles cross each other every 25ns.
- Software trigger selects 1 in 100 events. Since called less frequently, can use full detector information for full event reconstruction.
- First trigger selects 1 in 40 events. Performed in hardware, using subset of detector data.
- Combined trigger selects 1 in 40k events for physics analysis.

CMS selects 1 in O(10^7), whereas LHCb selects 1 in O(10^4) for the same output data rate - what causes the difference?
- LHCb opening angle is smaller! Less coverage.
- LHCb events are very common - can reduce luminosity (i.e. number of collisions per crossing).
Trigger Example 2 - Vertexing at LHCb

- LHCb is designed for precision studies of collisions involving b quarks:
  - Complementary to direct searches (e.g. discovering new particles).
  - Perform indirect searches by comparing experimental rates to theoretical predictions.
- Trigger can be designed for specific physics: typical signature involves a displaced vertex.
  - Not easy to reconstruct - performed in software farm at higher level trigger.
  - (Reconstruction of these objects discussed in lecture 2).

Filtering for events we do know about.

Displaced vertex in LHCb - signature of interesting event.

Lecture 1 Summary

- LHC detectors output a large amount of data.
  - Four major experiments, each outputting O(1) GB/s physics data (post trigger).
  - Many results required live (detector monitoring, calibrations etc.).
  - Reconstruction performed on the GRID (Lecture 2).
- Triggers are used to filter for interesting events in real time.
  - Can be multilayered, using electronics, CPUs and GPUs.
  - Complexity ranges from simple algorithms through to full event reconstruction.
  - Designs heavily influenced by the physics aims of the experiment.
Detector Simulation for the LHC and beyond:
how to match computing resources and physics requirements
(Lecture 1)

Valentina Cairo
CERN & Università della Calabria

Audience, benefits, prerequisites…

- **Targeted audience:** Anybody interested in understanding the principles of simulation programmes in the context of High Energy Physics experiments and anyone curious of knowing where ~50% of the CPU available on GRID is spent.

- **Expected benefits:** The attendees are expected to have a general understanding of the problems to be tackled when simulating a complex detector and of the reasons why good and efficient coding is necessary.

- **Prerequisite knowledge:** The argument is treated in order that no special prior knowledge of the topics is required to benefit from most of these lectures.

- **Dependencies:** Though the two lectures are in principle independent, in this case, there might be some benefit for listeners to have attended the first lecture of the series because the general idea and motivation behind the complex detector simulation will be given, while the second one will focus on more concrete examples related to future experiments.

Introduction

- **Detector simulation** at the LHC is the most computing intensive activity (~50% of the GRID resources)

- **Detectors** are complex, very precise and ambitious: this implies modern modelisation tools for geometry and response. **Events** are busy and characterised by an unprecedented energy scale with hundreds of particles to be traced and high energy showers to be accurately simulated. Furthermore, high luminosities imply many events in a bunch crossing and many bunch crossings to be considered at the same time. In addition, backgrounds not directly correlated to bunch crossings have also to be taken into account.

- Features of detector simulation at the LHC (with emphasis on full simulation) will be described and CPU and memory figures will be given in Lecture 1.

- An extrapolation to the FCC-hh case will be tried and possible solutions (mostly ATLAS-inspired) to be adopted will be discussed in Lecture 2.

Disclaimers

1. Most of the examples shown are ATLAS-based
   - It’s the easiest choice for me:
     - I am a member of the ATLAS collaboration

2. Results can be quite dated
   - Current publication policy does not allow to show unpublished results
   - The message does not change!

3. Quite hard to summarise ~20 years of activities of many people in 2 lectures
   - Some items will be only mentioned…
   - …some other items will not be mentioned at all!
   - Mistakes may have crept in, apologies
Outline – Lecture 1

1. Detector Simulation: what does it mean?
   - Detector Design \(\rightarrow\) Resolution Studies
   - Detector Optimization \(\rightarrow\) Granularity
   - Detector Response \(\rightarrow\) “Real” Events

2. Simulation Strategies
   - Monte Carlo smearing
   - Fast (or parameterised) Simulation
   - Full Simulation

3. Geant4
   - Geometry Model and Field
   - Physics “menus”

4. Simulation Steps
   - Tracking Geometries
   - Physics Processes
   - Magnetic Field Corrections

5. Simulation at the LHC
   - Complexity
   - Pile-up

6. Some Performance Figures
   - CPU
   - Memory

The LHC Scenario

1. Events are characterized by unprecedented energy
   - LHC-Run 2 \(\sqrt{s} = 13\) TeV

2. Hundreds of particles have to be traced

3. High rate
   - At any moment, one has to deal with events from 3 different bunch crossings in the detector

4. Pile-up
   - Multiple interactions in any bunch crossing (in-time)
   - Tens of bunch crossings contributing to final signal (out-of-time)

Stay tuned!
The LHC Scenario

- How does a complex detector work?

- Each particle type has its own "signature" in the detector
- Charged particles are detected both in the tracking chamber and in the electromagnetic calorimeter

Simulation Strategies

1. How complex is your detector?
   - Complicated geometry?
   - Dense material?
2. What level of details do you need?

You have to answer these questions to decide your simulation strategy!

Simulation Strategies – MC smearing

- "Expected" detector resolution to be applied on top of the generated event
- Useful to evaluate discovery potential (or physics performance) of the detector
  - For instance, in the initial phase of development (Technical Design Report TDR)
- It's not related to detector
  - Geometry, read-out, etc, have no influence on the smearing procedure which is mostly used for physics studies
- Very fast but very idealistic!
- Examples:
  - ATLFast (ATLAS), Delphes
Simulation Strategies – Fast

- **Detector response** in simulation is **parameterised according to models** and/or tests with particle beams
- Detector geometry hardly taken into account
- Good on average
- It does not reproduce fine effects or peculiar behavior
- Often used as part of Full Simulation programs as a means to reduce execution time
- It cannot be used to design the detector
  - Experimental effects can be introduced “a posteriori”
  - Requires full simulation and/or real data for realistic tuning

Devil’s in the details...

Simulation Strategies – Full

- Based on well consolidated packages
  - Geant4
- **Accurate** description of the detector
- Particles propagated through the detector step-by-step
- Physics processes are accurately reproduced
- Good for any stage, from detector conception to data taking
- **CPU and Memory Intense**!

Simulation Strategies – Comparison

- The two extreme cases:
  - **MC smearing** (precision, cpu)
  - **Full simulation** (precision, cpu)
- MC smearing CAN NOT be used to compare real data with predictions
  - Useful to simulate physics channels and see what an ideal detector response does with them
- Full simulation CAN ONLY be used to simulate a fraction of the full data sample
  - Useful for physics analyses and to see the actual detector response
- A possible compromise:
  - **Full simulation with some level of parameterisation** (precision, cpu)

Just to set the ballpark not to be taken too seriously!

Simulation Strategies – key points
The Full Simulation Approach

- **Particles are generated** to simulate e.g. a physics event
  - Normally, use particle generators out of the box (Pythia, Herwig etc) – not described here!
- **Particles are propagated** one by one through the simulated detector (tracking) where they:
  - deposit energy
  - undergo physics processes
  - decay
  - etc
- **Energy deposited** in the simulated detector is further processed to reproduce a realist detector response
- Different events can be overlaid to introduce effects like pile-up, backgrounds, etc
- Ideally, fully simulated events can be reconstructed in the same manner as real data

GEometry ANd Tracking – Geant4

- **Geant** was developed in the 80s for the LEP experiments
- **Geant4** is the most recent incarnation aimed at the LHC and beyond
  - Toolkit for the simulation of the passage of particles through matter and detector modeling
  - Its areas of application is mainly high energy physics, but also nuclear and accelerator physics, as well as studies in medical and space science

Geant4 – Detector Geometry

- **Detector modeling** is the first step in the full simulation chain
- Normal approach based on simple primitives (boxes, tubes etc.)
- Primitives combined together to form volume hierarchies
  - ATLAS is made of millions of volumes
- Interface to Computer-Aided Design (CAD) systems available but not used (performance issues!)
- Alternative ways to describe geometries existing (GDML) or developed in the experiments (mostly XML based)
- Going from simple, conceptual designs to super complicated detector layouts

Geant4 – Detector Geometry

- Need a fairly detailed detector description to simulate reality
  - Ideally, every nuts and bolts count, but complexity comes at a price (CPU, memory) → details in a short while
- Need for a compromise (human factor + computing resources)
Geant4 – Detector Fields

- Magnetic field adds an additional level of complication
- It has to be calculated accurately by specialized programs
  - Results are stored in huge field maps which depend on detector layout (for instance CMS solenoid, ATLAS solenoid+toroids), currents in the magnets, distortions etc.
- **Code** to read the maps and interpolate has to be written and made as efficient as possible

![Field Maps](Image)

Geant4 – Physics Processes

- Geant4 provides classes for all particle types and physics processes but it does NOT come with any association between particles and processes
  - It must be implemented in the application
- Pre-compiled collections of models (physics lists) are used according to the experiment requirements and the energy range
  - High energy collision, background calculation, neutron physics…
- Different lists ➔ Different physics scenarios ➔ Different results
  - The list that best matches experimental data is chosen as the reference model
- Release by release, the stability of the results must be verified not to introduce undesired systematics (validation is very important)
  - Different lists are used at all times for comparison and cross-checks

![Energy Response](Image)

Geant4 – How does tracking work?

- Geometry, fields, physics… all influence particle tracking in the simulation
- The more complicated the detector is, the more difficult it becomes to track particles into it

![Tracking Diagram](Image)
Geant4 – How does tracking work?

- Particles propagate in steps
- Particles propagate until
  - A physics process occurs
- All physics processes the primary particle can undergo are sampled
- The most probable process is chosen and it can limit the step
- Possible secondary particles are created and added to the stack of particles to be traced afterwards
- Effects due to continuous processes (ionization, multiple scattering, etc) are taken into account and may modify the primary’s energy and momentum
- CPU goes into calculating probability of physics processes

At a certain point, we were giving up ~15% of CPU when switching on the magnetic field!
Geant4 – Where Do Those Steps Go?

- The number of steps dictates the speed of the simulation:
  - The best code changes get ~3-10% speed ups
  - The best physics changes get ~20-50% speed ups!

Example: 50 ttbar events

<table>
<thead>
<tr>
<th>iCSC 2016, Valentina CAIRO, CERN &amp; Università della Calabria</th>
</tr>
</thead>
</table>

Geant4 – How many particles?

- At (almost) every step new secondary particles are created and added to the stack of particles to be traced

Example: 3 ttbar events

- The stack grows as secondaries are generated
- The stack shrinks as the particles are slowly ground through by the simulation

Geant4 – Tracking Summary

- Complicated geometries slow down the tracking

<table>
<thead>
<tr>
<th>Geant4 – Tracking Summary</th>
</tr>
</thead>
</table>

Geant4 – Tracking Summary

- Tracking in dense materials produces lots of secondaries and slows down simulation
  - Calorimeters (where showers develop) represent the real bottleneck in any simulation program
  - Production thresholds for secondaries must be chosen accurately as a trade off between precision and speed
Geant4 – Tracking Summary

- Tracking in field slows down simulation and requires careful optimization
  - Highly inhomogeneous fields have a big impact on performance

Paso de los Libertadores (Chile-Argentina)

Geant4 – By Detector Type

- Calorimeters (where showers develop) represents the real bottleneck in any simulation program

Geant4 – By Particle Type

- Electrons and gammas do shower in the calorimeters, producing 1000s of secondaries
  - It slows down simulation
  - Raising production thresholds improves CPU performance, but has very bad impact on precision
- Charged hadrons (pions, protons, etc) do shower in the calorimeters, but they are somewhat better than em-particles performance-wise
- Neutrinos do not interact in the simulated detector and should not be traced
- Neutrons seldom interact in the detector and propagate forever
  - Very bad impact on performance
  - Treatment is application-dependent

Geant4 – Detector Digitization

- Geant4 has no idea how detectors react to particles
  - It only deposits energy with a position and time
- First of all, deposited energy must be collected, then energy must be converted into something that can be read out
  - How that’s done depends completely on the detector
  - Have to take into account detector timing, pulse shapes, charge collection effects (sometimes conditions dependent)
- Very complex procedure
  - In ATLAS, all custom ATLAS code (with lots of parameters)
The Full Simulation Chain

- Monte Carlo Simulation follows the evolution of physics processes from collision to digital signals

Where does the LHC come into this?

- Very precise detectors
  - Implies detailed and accurate simulation programs
- Big and complex detectors
  - Requires detailed detector description and accuracy over several orders of magnitude
  - From microns (Pixel detector) to tens of meters (Muon detector)
- High energy means high multiplicity events
- High energy means high energy particles in the final state (details in the next lecture)
- High luminosity means many simultaneous events per bunch crossing (in-time pile-up)
- 25 ns crossing rate means the events of 3 bunch crossings at all time in the detector (out-of-time pile-up)
Where does the LHC come into this?

- Very precise, big and complex detectors
  - One can actually spot differences between data and simulation at the level of < 1mm!

Error or oversimplifications in the detector description lead to visible disagreements between data and MC.

ATLAS Preliminary

$\sqrt{s} = 13$ TeV

ICSC2016, Valentina CAIRO, CERN & Università della Calabria

Where does the LHC come into this?

- High energy means high multiplicity events

ATLAS Preliminary

$\sqrt{s} = 13$ TeV

ATLAS Data, 2015

ICSC2016, Valentina CAIRO, CERN & Università della Calabria
Where does the LHC come into this?

- High energy means high multiplicity events... and it can be even worse!

Simulating very high energy events can take hours of CPU!

The simulation has to be very precise from low-momentum to very-high momentum objects!

- High energy means high energy particle in the final state

- The mass of the di-photon system is 1.5 TeV
- One photon candidate, with a transverse momentum of 530 GeV, is reconstructed in the endcap region, while a second, with a transverse momentum of 400 GeV, is reconstructed in the barrel region

Simulating very high energy events can take hours of CPU!

The simulation has to be very precise from low-momentum to very-high momentum objects!
Where does the LHC come into this?

- High luminosity means many simultaneous events per bunch crossing
  - Display of a proton-proton collision event recorded by ATLAS on 3 June 2015, with the first LHC stable beams at a collision energy of 13 TeV
  - The four inner layers are part of the silicon pixel detector and the four outer layers are part of the silicon strip detector
  - The layer closest to the beam, called IBL, is new for Run 2
  - This event has multiple p-p collisions: 17 reconstructed collision vertices

The Pile-up conundrum

- High luminosity means many simultaneous events per bunch crossing
  - → high pile up
  - Pile up -µ- changes from event to event
    - In time
    - Out of time
    - Background
  - During 2011 and 2012 the average number of collisions per bunch crossing, pile-up, increased up to almost 40
  - In Run 2, typical pile up ~40, with peaks up to 80
  - In some upgrade scenarios, pile-up can reach peaks of ~200!
  - Every physics object is affected (e.g. jets collect more energy)

- High luminosity means many simultaneous events per bunch crossing
  - → high pile up
  - Pile up -µ- changes from event to event
    - In time
    - Out of time
    - Background
  - In ATLAS, a stockpile of ~40 million MinBias events is kept to reproduce pile-up
  - The hits (simulation output) get overlaid before digitization
    - This ensures things like saturation, shadowing, etc are accounted for
  - Overlaid at a rate to match that recorded on the detector
    - Including variations by data taking period
    - Including bunch structure in the machine (long and short gaps)
  - The mechanism becomes almost as time-consuming as simulation at µ ~100
  - Association with generated events is the hard part! → cleared out of most of the MinBias pile-up samples so that it won’t slow down all the digitization...
The Pile-up conundrum

- High luminosity means many simultaneous events per bunch crossing
  - high pile up
- Pile up $\mu$- changes from event to event
  - in time
  - Out of time
- Background
  - The topic would take hours of discussion, but... just to give you an idea:
    - Beam halo (beam scraping on upstream collimators)
    - Beam-gas collisions (interactions with the not-quite-vacuum in the beampipe)
    - Cavern background (neutron and photon gas in the cavern)

The Pile-up conundrum

- Pile-up Digitization: VMEM breakdown
- $\mu \geq 8.0$, Fixed 25ns bunch-spacing.

What about Performance?

- CPU Time per event single particle - Pions VS Electrons -
- CPU time per event single pions
- CPU time per event single electrons

<table>
<thead>
<tr>
<th>Release</th>
<th>CPU Time per event (NCU s)</th>
<th>CPU Time per event (NCU s)</th>
<th>CPU Time per event (NCU s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.0.0</td>
<td>54.936</td>
<td>20.267</td>
<td>1.690</td>
</tr>
<tr>
<td>pion</td>
<td>92.234</td>
<td>33.309</td>
<td>2.332</td>
</tr>
<tr>
<td>electron</td>
<td>59.04</td>
<td>20.76</td>
<td>1.690</td>
</tr>
<tr>
<td>neutron</td>
<td>92.234</td>
<td>33.309</td>
<td>2.332</td>
</tr>
</tbody>
</table>
What about Performance?

- CPU Time per event Full Physics

<table>
<thead>
<tr>
<th>Release</th>
<th>CPU Time per event [NCU s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUSY</td>
<td>1059.265</td>
</tr>
<tr>
<td>MinBias</td>
<td>211.623</td>
</tr>
</tbody>
</table>

What about Performance?

- Memory per domain for various components of simulation

Memory per domain for various components of the simulation

- Total Virtual Memory of Single Particle Jobs

No evident difference among single particle jobs

What about Performance?

- Total Virtual Memory of Full Physics Jobs

<table>
<thead>
<tr>
<th>Release</th>
<th>Total Virtual Memory [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUSY</td>
<td>796.997</td>
</tr>
<tr>
<td>MinBias</td>
<td>773.412</td>
</tr>
</tbody>
</table>
What about Performance?

- **Total Resident Set Size of Full Physics Jobs**

<table>
<thead>
<tr>
<th>Release</th>
<th>Total Resident Set Size [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUSY</td>
<td>695.637</td>
</tr>
<tr>
<td>MinBias</td>
<td>673.748</td>
</tr>
</tbody>
</table>

- **Event Size of Single Particle events**

<table>
<thead>
<tr>
<th>Particle</th>
<th>Event Size [MB] per event</th>
</tr>
</thead>
<tbody>
<tr>
<td>pion</td>
<td>30.737, 19.138, 6.901</td>
</tr>
<tr>
<td>muon</td>
<td>4.476, 5.769, 3.845</td>
</tr>
</tbody>
</table>

- **Size of containers in output file**

<table>
<thead>
<tr>
<th>Container</th>
<th>Size of container [KB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCT</td>
<td>266.826</td>
</tr>
<tr>
<td>MDT</td>
<td>1.869</td>
</tr>
</tbody>
</table>
Recap of Lecture 1

- Now you know:
  - How a High Energy Physics detector works
  - How complex it is
  - Which flavors of simulation are used
  - What are the differences among MC Smearing/Full/Fast simulation
  - What does Geant4 provide and what must be implemented
  - Which are the limitations
  - How the LHC deals with them
  - What are the performances (in term of CPU, memory, etc) and how they change depending on physics (particle type, phenomena, etc)

- Take away messages:
  1. At the development and the early analyses of the LHC experiments, full simulation was often the solution of choice
  2. While this strategy was very successful from the physics point of view, the complexity of the simulation programs led to hit the CPU/memory limits
  3. Pile-up is an issue now and it will become worse in future

Next step

- Tomorrow you will learn
  - What are the new challenges when you move toward a future (futuristic?) detector
    - From the LHC (27 km, 13-14 TeV) to the FCC (100 km, 100 TeV)

Thanks for your attention!

Imagination is more important than knowledge

Albert Einstein
Pile-up

Terminology
- \( \mu \) = interactions per crossing averaged over a specific lumi block for a specific BCID (bunch crossing ID)
- \( \mu_{BCID} \) = interactions per crossing averaged over a specific lumi block and over all colliding BCIDs

Background type determines how events are added to the signal event:
- Minimum Bias: Add a random number of events picked from a Poisson distribution with mean \( \mu \) to each colliding BCID
- Cavern Background: Add a constant number of events to each BCID. Rate depends on \( \mu \) x fraction of colliding BCIDs
- Offset event times according to the BCID they are used

Pile-up in ATLAS

- Multiple pp collisions in each filled LHC bunch-crossing and multiple filled bunch-crossings within the [-800,800] ns sensitive time window of ATLAS
- Simulation in the Athena framework proceeds as follows:
  - Run the event generation and (Geant4) simulation steps for single pp interactions
  - Combine multiple simulated pp interactions during the digitization step
    - "Pile-up Digitization"
  - Attempts to reproduce this situation by digitizing the HITS from many simulated pp interactions all together
  - This includes both in-time and out-of-time pp interactions
  - "Cavern background" events are also added

Pile-up in ATLAS

- Generating huge samples of background events = Expensive!
- Disk constraints limit the maximum sample size
- Create a cache of background events in memory, so they can be re-used
- Save memory by only reading in/caching the parts of each event which are needed
- After a cached event is used, it may be replaced by a fresh event
- Cache size dominated by the size of Truth information

<table>
<thead>
<tr>
<th>Background Type</th>
<th>In-time/Out-of-time</th>
<th>Replacement Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>High pT Minimum Bias</td>
<td>Both</td>
<td>100%</td>
</tr>
<tr>
<td>All</td>
<td>In-time</td>
<td>100%</td>
</tr>
<tr>
<td>Low pT Minimum Bias</td>
<td>Out-of-time</td>
<td>~1% (tunable)</td>
</tr>
<tr>
<td>Cavern Background</td>
<td>Out-of-time</td>
<td>~1% (tunable)</td>
</tr>
</tbody>
</table>
Example of a pile-up model with fixed 50ns spacing between colliding BCIDs:

- In reality the structure of colliding and non-colliding BCIDs can be more complicated.
- The pile-up/detector response is affected by the position of the triggering BCID in the bunch train.
- Bunch structure modelling is included in the pile-up simulation.
  - Patterns can be up to 3564 elements in length and wrap-around if required.
  - Each triggering BCID is picked from the colliding BCIDs in the pattern, with a probability proportional to the relative luminosities of each bunch crossing.

<μ> varies over time:

- μ can also vary greatly from BCID to BCID in data, as the plots above show.
- Both in-time and out-of-time pile-up effects are important.
- Problem: Simulating samples at a fixed <μ> value makes it difficult to re-weight MC to data.

Possible solution: Use a range of <μ> values within each simulated sample.
- The μ and <μ> value used are recorded for each event.
- This can then be used to re-weight the MC sample to match a given set of data periods.
Vectorization with Expression Templates

Jiří Vyskočil
Czech Technical University
AS CR – ELI Beamlines

Inverted CERN School of Computing, 29 February – 2 March 2016

Lecture series overview

- techniques for writing algorithms for physics computing in modern C++
- we will strive to produce code which is
  - easy to read
  - efficient
  - modular
- we will introduce some specific methods to
  1) enable natural syntax for mathematical operations
  2) tap into the CPU’s vector processing capabilities
  3) process large data sets
- and explore their inner workings

Yesterday’s lecture

Expression Templates
“natural syntax for efficient mathematical operations”
- we have created an “Expression Template” system
- by means on compile-time expansion of a tree built using C++ template system
- generate efficient low-level code from a high level “object-oriented” syntax

In this lecture

tap into the CPU’s vector processing capabilities
- modern CPUs contain instructions which can manipulate multiple data elements (numbers) at once
- to use these instructions efficiently, we have to think about our data layout – we will introduce the concepts of Array-of-Structures vs. Structure-of-Arrays
- we will explore a template metaprogram which allows “natural” mapping of OO-like structures into an efficient compact storage
- all the examples presented here will build upon the idea of Expression Templates form yesterday’s lecture
Templates for Iteration; Thread-level Parallelism

- separating the concepts of “iteration” and “computation”
- utilizing the separation to easily introduce parallelization
  - in addition to the benefits from yesterday’s and today’s lectures
- practical example of integrating the Maxwell’s equations

Vectorization

- performing one operation on multiple operands simultaneously
- SIMD - “Single Instruction Multiple Data”
- SIMD instructions – CPU instructions which perform the operation
- SIMD registers – stores in the CPU which hold the operands

Intel SIMD architectures

- Intel introduced SIMD in their x86 chips in 1997
  - “MMX” instruction set with 80 bit registers, integer-only
- SSE introduces 128 bit floating point registers (1999)
  - can hold 4 single precision floats
- SSE2 adds support for double precision FP (2001)
  - still 128 bit – 4 single precision float, or 2 double precision
  - up to SSE4.x (2006) new instructions are added
- AVX/AVX2 increases the length to 256 bit (2008)
  - can hold 4 double precision FP numbers
- AVX-512 – upcoming processors, 512 bit registers
  - that’s 8 double precision, or 16 single precision FP numbers
  - contemporary Xeon Phi (KNC) have also 512 bit SIMD, but a different instruction set

Vertical vectorization

- case from our last lecture – adding N 3D vectors
- “natural” mapping of a “mathematical vector” to a “hardware vector”
  - let’s load our vector into the CPU’s vector register and compute all components at once using the vector operations
    - calculate \( \vec{z} = \vec{a} + \vec{b} \) using one instruction
    - struct vector {
        __m128 v;
    };
    - type holding 4 floats
    - vector a, b, c;
    - \( \vec{c} = \_mm\_add\_ps (\_a.v, b.v); \)
  - simple implementation
  - not scalable
  - possible waste of longer registers
    - \_mm128 can hold 4 floats
    - only 3 utilized if mapped to a 3D vector
    - even more wasting on ‘longer’ architectures
    - function that calls the add instruction for 4-float vectors
Vectorization with Expression Templates

SIMD instructions

- “intrinsic functions” (also called “intrinsics”) translated directly to vector instructions by the compiler
  - __m128_mm_add_ps (__m128 a, __m128 b) SSE2 – 4 floats
  - __m256_mm256_add_pd (__m256 a, __m256 b) AVX – 4 doubles
- modern compilers understand constructs like a+b on vector types, and will emit the correct vector instruction
- modern compilers auto-vectorize inner loops
  - intrinsics
  - assembly (too explicit)

Horizontal vectorization

- operate on multiple 3D vectors at the same time
- instead of
  \[
  \begin{align*}
  c_x & = c_x^1 + c_x^2 + c_x^3 \\
  d_y & = d_y^1 + d_y^2 + d_y^3 \\
  e_z & = e_z^1 + e_z^2 + e_z^3
  \end{align*}
  \]
  for each vector
- for each component (and all vectors jumping by i += register size)
- no waste of registers (if implemented properly)
- code scales to “longer” architectures

Expression Templates for SIMD

- hide the implementation details in an ET library
- keep the OO-like interface
- use proper SIMD data types and operations
- we will have to explicitly deal with SIMD
  - but we will only do it inside the reusable library code

Packed data types

- SIMD instructions operate on “packed” data types
- we have to map arithmetic types into these “packs”

Using the intrinsic types

```c
// load from unaligned array using the intrinsic types
float x[4] = {1.0, 2.0, 3.0, 4.0};
__m128 b = _mm_loadu_ps(x);
// with recent compilers
__m128 a = __m128 loadu_ps(x);
```

Using the gcc vector extensions

```c
typedef int v4si __attribute__((vector_size(16)));
typed int v4si __attribute__((vector_size(16)));
```

```c
// load from unaligned array using the gcc vector extensions
int a, b, c;
__m128 a = __m128 loadu_ps(x);
```

```c
__m128 b = _mm_loadu_ps(x);
```

```c
// with recent compilers
__m128 a = __m128 loadu_ps(x);
```

```c
// a = [1.0, 2.0, 3.0, 4.0];
b = a + 1;
c = a + b;
```
**Simple “vertical” ET**

- straightforward implementation
- just change the storage type to the appropriate SIMD type
- you can explicitly state the SIMD instructions inside the operator structures
- no changes to original code needed
- probably wasting registers
- certainly not scalable

```cpp
class Vector {
  __m128 v;
public:
  template <typename T>
  struct AddOp {
    static T apply(T const& a, T const& b) {
      return __mm_add_ps(a, b);
    }
  };
  Vector(initializer_list<float>& in) {
    v = in;  // possible with recent gcc
  }
};
```

```cpp
Vector a = {1,2,3};
Vector b = {1,2,3};
Vector c = a + b;
```

**Road to “horizontal” ET**

- we want to make the operator+ perform simultaneous operation on the same component of several vectors at once
- the number of these vectors depends on the size of the register, and the size of the arithmetic data type (SP/DP)
- there are more ways to achieve this
- but first let’s look at how the data travels from the main memory to the CPU’s registers

**Cache I**

- hierarchy of memory from fastest to slowest
  - register -> L1 cache -> L2/L3 cache -> main memory
- data from main memory is loaded to a cache in a “line”
  - one “cache line” is 64 bytes on all x86 architectures
- subsequent access to the same cache line is fast
- if the requested data is not found in the cache, the whole line gets evicted, and a whole new line has to be loaded
- we want to “stream” the data to the processor as fast as possible – ideally without “cache misses”

Vectorization with Expression Templates

AoS vs. SoA

- "Array of Structures" data model
  - all members of one vector are next to each other
  - good when you need to access all members at once (e.g. sorting...)
  - example:
    \[
    \begin{array}{cccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
    x & y & z & x & y & z & x & y & z & x & y & z \\
    \end{array}
    \]

- "Structure of Arrays" model
  - same members of different vectors are next to each other
  - better suited for horizontal vectorization
  - we can continuously stream needed data
  - example:
    \[
    \begin{array}{cccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
    x & y & z & x & y & z & x & y & z & x & y & z \\
    \end{array}
    \]

AoS vs. SoA runtime transformation

- if you really must use the AoS format
  - existing code that would be too difficult to change
  - the algorithm performs faster in AoS
  - example:
    \[
    \begin{array}{cccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
    x & y & z & x & y & z & x & y & z & x & y & z \\
    \end{array}
    \]

- shuffle the data into appropriate registers
- perform calculation with full register width
- bad cache utilization
- but might be better than wasting registers on "long enough" architectures
- avoid unless necessary; test, measure, profile

AoS to SoA

- "Array of Structures" data model
  - all members of one vector are next to each other
  - good when you need to access all members at once (e.g. sorting...)
  - example:
    \[
    \begin{array}{cccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
    x & y & z & x & y & z & x & y & z & x & y & z \\
    \end{array}
    \]

- "Structure of Arrays" model
  - same members of different vectors are next to each other
  - better suited for horizontal vectorization
  - we can continuously stream needed data
  - example:
    \[
    \begin{array}{cccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
    x & y & z & x & y & z & x & y & z & x & y & z \\
    \end{array}
    \]

there are algorithms which work better in a AoS layout (or AoSoA, in a couple slides) – test, measure, profile!

AoSoA

- an array containing structures containing smaller arrays where each component is the same length as the register
  - example:
    \[
    \begin{array}{cccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
    x & y & z & x & y & z & x & y & z & x & y & z \\
    \end{array}
    \]

- well suited for vectorization especially if you often access multiple members of the same vector
- arguably most difficult to implement
Expression Templates (lecture 1)

- a vector class and a vector_expression class
- assignment triggers the cast from vector_expression to vector, which is performed by recursively applying operator[] in a loop to subexpressions connected by operator+
- compiler translates the statement \( x = a+b+c \), with \( a+b+c \) represented

\[
\begin{align*}
\text{vector_expression} & : \text{AddOp,} \ > \\
\text{vector(a)} & \\
\text{vector_expression} & : \text{AddOp,} \ > \\
\text{vector(b)} & \\
\text{vector(c)} & \\
\text{vector_expression} & : \text{AddOp,} \ > \\
\text{vector(a)} & \\
\text{vector_expression} & : \text{AddOp,} \ > \\
\text{vector(b)} & \\
\text{vector_expression} & : \text{AddOp,} \ > \\
\text{vector(c)} & \\
\end{align*}
\]
- into the following code

\[
\begin{align*}
\text{for(int i=0; i<3; ++i) } & \{ \\
\quad x[i] = a[i] + b[i] + c[i]; \\
\}
\end{align*}
\]

Trivial ET AoSoA implementation

- we can replace the internal storage with a type that holds 4 float vectors
- REG is the register size
- REAL is the type we will map
- SIMD is the underlying intrinsic type
- MM_ADD is the add instruction
- these are set at compile-time using #ifdefs and #defines together with appropriate includes
- old-school, but proven
- see Agner's "vectorlib" for a more comprehensive implementation

```cpp
#if defined ( __AVX__ )
#include <immintrin.h>
#define REAL float
#define REG 8
#define SIMD __m256
#define MM_ADD _mm256_add_pd
```

```cpp
template <int N>
class vector_pack {
  v std::array<SIMD, N>;
  ...
};
```

```cpp
std::array<vector_pack, NUM / REG> a;
```

- but this means we have to change our loops
- the total number of elements when storing NUM "mathematical" vectors is NUM / REG
- we cannot easily address individual elements – only the whole block
- you can see how it leads to code bloat in the initialization section

- apart from the ugly loop issue the expression templates work as expected, and vectorized code will be emitted

```cpp
for(int i = 0; i < NUM / REG; ++i) {
  for (int comp = 0; comp < N; ++comp {
    for (int reg = 0; reg < REG; ++reg) {
      a[i][comp][reg] = i+reg;
    }
  }
  for (int i = 0; i < NUM / REG; ++i) {
    c[i] = a[i] + b[i]; // operates on packs
  }
}
```

```cpp
std::array<vector, NUM> a, b, c;
```

- we could use this if we don't mind the cumbersome initialization and single-element access
- could be solved by introducing a custom container
- the implementation is faster than the "vertical" approach

```cpp
std::array<vector_pack, NUM / REG> a;
```

```cpp
// ... initialize ...
for (int i = 0; i < NUM / REG; ++i) {
  c[i] = a[i] + b[i];
}
```

```cpp
std::array<vector, NUM> a, b, c;
```

```cpp
// ... initialize ...
for (int i = 0; i < NUM; ++i) {
  c[i] = a[i] + b[i];
}
```
Auto-vectorization

- before moving to implementing a simple SoA approach, let's see what can the compiler do for us
- many loops can be automatically vectorized by the compiler
- there are caveats that prevent vectorization:
  - loop dependencies
  - aliasing
  - non-inlined functions inside the loop
- or result in sub-optimal performance:
  - mis-alignment
  - strided access

// often this might be sufficient for getting vectorized code
for(i=0;i<N; ++i){
a[i] = b[i] * c[i];
}

// might work "out-of-the-box"

$icc -O3 file.cpp
$gcc -O2 -ftree-vectorize file.cpp
$gcc -O3 file.cpp

Loop unrolling

- first step in auto-vectorizing a loop is "unrolling" - increasing the loop's step size and repeating its contents
- some older sources recommend manual loop unrolling to help the compiler see the opportunity for vectorization
- don't do that – let the compiler do the unrolling:
  - manual unrolling might interfere with further compiler optimizations
  - less readable code

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

// unrolled loop
for(i=0; i<N; i+=4) {
  a[i] = b[i] * c[i];
  a[i+1] = b[i+1] * c[i+1];
  a[i+2] = b[i+2] * c[i+2];
  a[i+3] = b[i+3] * c[i+3];
}


Inlining

- after the loop is unrolled, vectorization can be achieved by packing subsequent variables into a SIMD register, and replacing each repeated operation with a SIMD operation
  - this means there such SIMD operation must exist
  - there can be no function calls except for those which can be translated to a sequence of SIMD operations
- if there are function calls in the loop, the compiler must be able to inline them:
  - inlining – replacing the function call with the function's body
  - otherwise the loop won't be vectorized

for(i=0; i<N; ++i) {
  // fine – translates directly to a SIMD operation
  a[i] = b[i] * c[i];
  // fine – most compilers have a vector version
  a[i] += sin(d[i]);
  // only if f could be inlined, and it's body doesn't contain non-vectorizable statements
  f(a[i]);
}

Loop dependencies

- dependencies between reads and writes might prevent vectorization:

<table>
<thead>
<tr>
<th>Dependency Type</th>
<th>Read-After-Write</th>
<th>Write-After-Write</th>
<th>Write-After-Read</th>
</tr>
</thead>
</table>
| read-after-write | for(i=0; i<N; ++i) {
|                 | a[i] = a[i-1] + b[i].
|                 | a[i] = b[i] + c[i];
|                 | ...              |
|                 | a[i+2] = 2 * i;  |
|                 | }                |

writing to a variable, then reading its value
writing to the same variable in more than one iteration
no data race with vectors – we know a[i] won't be written to before being read

not vectorizable
not vectorizable
vectorizable

https://software.intel.com/en-us/node/524710
Aliasing

- when passing arrays by pointers, the target memory of the variables might overlap - “alias”
- if the compiler cannot be sure they won’t, the loop will not be vectorized
- the compiler can tell you if it didn’t auto-vectorize
  - icc pre-15: -vec-report=2,
  - icc 15: -qopt-report=m-qopt-report-phase=vec
  - gcc: -free-vectorizer-verbose=2

Alignment

- position of the beginning of the data structure
- there are different instructions for loading aligned vs. unaligned data into the registers
  - the unaligned version is slower
- unaligned access might cross a cache line boundary
  - the processor then has to do two reads to fill the register

```c
void f (int* offset,
        double* a,
        double* b,
        double* c)
{
    for (int i=0; i<N; ++i)
        c[i] = a[i] + b[i] + (offset[i]);
}
```

Alignment

- in auto-vectorized loops over compact data structures, modern compilers will emit code which does aligned access
  - if it’s possible (e.g. iterating over an array with step = 1)
  - misaligned accesses at the beginning of the loop will be treated separately
    - “peeled loop” - “loop” - “remainder loop”
- in an AoS setting, alignment might also require padding
  - e.g. a struct of three 32 bit floats with 128 bit alignment needs to be padded by additional 32 “useless” bits
  - this will increase memory consumption
  - and for long arrays also traversal time

Manual alignment

- alignas(BYTES) float a[1000];
  - since C++11
- std::align
  - C++11
  - for dynamically allocated variables
- aligned_storage
  - C++11

more at https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization
SoA implementation with auto-vectorization

- let’s take the ET vector from the previous lecture
- change it to represent many N-dimensional vectors
- organized in SoA layout
- we will change the evaluation trigger using function eval(n,i) instead of operator[]

```cpp
alignas(ALIGNMENT) template <int N, int L>
class vector_field {
  float[N][L] v;  // contiguous in "L"

  template <typename RightExpr>
  vector& operator=(RightExpr&& re) {
    for (int n=0; n<N; ++n)
      for (int i=0; i<L; ++i)
        v[n][i] = re.eval(n,i);
    return *this;
  }
};
```

SoA implementation with auto-vectorization

- since the apply recursion is stopped by encountering a leaf node – the vector_array, we have to add eval function to that class as well
- this is because we need to pass two arguments to the evaluator
  - the element index
  - and the coordinate index

```cpp
template<RightExpr,BinaryOp,LeftExpr>
class vector_expression {
  auto eval(int n, int i) const -> decltype(auto) {
    return BinaryOp::apply(l.eval(n,i), r.eval(n,i));
  }
};
```

SoA implementation with auto-vectorization

- we can expose the internal storage by providing an operator[]
- but we will have to change any code that used it
- indexing order by element and its coordinate has changed
- the new syntax feels weird

```cpp
template <int N, int L>
class vector_field {
  float[N][L] v;

  auto& operator[](int i){
    return v[i];
  }
};
```

SoA implementation with auto-vectorization

- again, the interface changed – now we just write the expressions to perform an operation on all elements instead traversing them in a for loop
- we’ve lost the option to operate only on a subset of the array
- one question is – do we really need the for loop?
  - we will answer this tomorrow

```cpp
// instead of
std::array<vector<N>,L> a, b, c;
for (int i; i < L; ++i) {
  c[i] = a[i] + b[i];
}

// we now have
vector_field<N,L> a, b, c;

// instead of
vector_field<N,L> a, b, c;
for (int i; i < L; ++i) {
  c = a + b;
}
```
SoA implementation with auto-vectorization

- use a simple struct for 3D vector
- implement a container which holds 3D vectors in SoA layout
- provide a mechanism to access each of the vectors separately, using "natural" AoS-like syntax

```cpp
template<typename T>
struct Vector3 {
    T x, y, z;
    Vector3 (initializer_list in) : x(in[0]), y(in[1]), z(in[2]) {};
};
```

```cpp
class V3Container;  // see next slide
V3Container<float> cont(10);
Vector3<float> a{1,2,3};
cont[1] = a;
cont[1].x = 4;
```

the example uses ideas from Vincenzo Innocente's UltimateSOA implementation simplified for the purposes of this lecture to only operate on our 3-dimensional vectors of floats; for full implementation, see: https://twiki.cern.ch/twiki/bin/view/Main/VIUltimateSOA

SoA implementation with auto-vectorization

- our simple container will use std::tuple to hold three std::vectors of type T as storage
- the size of the container (number of our vectors) is decided at runtime
- aligned_allocator is not part of C++, you have to roll your own
- many good implementations can be found on the internet

```cpp
template<typename T>
class V3Container {
public:
    using Data = std::tuple<
        std::vector<T, aligned_allocator(16)>,
        std::vector<T, aligned_allocator(16)>,
        std::vector<T, aligned_allocator(16)>
    >;
    Data data;
    V3Container(){});
    V3Container(int n) {
        // resize each data for each component
        resize(n);
    }
};
```

Extension to arbitrary dimension

- we reuse our vector class from the last lecture
- now for N dimensions, our container needs a tuple which holds N elements of the same type
- the size of the tuple is determined by the number of template parameters
- we need to generate this at compile-time

```cpp
template<typename T, int N>
struct Vector {
    float v[N];
    T operator[] (int i) { return Vector3<float>([{ ( std::get<0>(data)[i] >;
    return Vector3<float>({ std::get<0>(data)[i],
        std::get<1>(data)[i],
        std::get<2>(data)[i] } );
});
```
Generating an N-tuple I

- we will create the N-tuple by recursively concatenating an N-1 tuple, and a single-element tuple
- remember that template metaprogramming algorithms operate on types, so the result of our operation will be a new type
- a template specialization with N=0 will end the recursion

```cpp
template<typename V, int N>
struct gen_tuple {
    using type = decltype (std::tuple_cat (std::declval<std::tuple<V>>(),
                                            std::declval<typename gen_tuple<N-1, V>::type>()));
};

template<typename V> struct gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

Generating an N-tuple II

- we want to create a new type
- the decltype keyword resolves the full name of that type

```cpp
template<typename V, int N>
struct gen_tuple {
    using type = decltype (std::tuple_cat (std::declval<std::tuple<V>>(),
                                            std::declval<typename gen_tuple<N-1, V>::type>()));
};

template<typename V> struct gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

Generating an N-tuple III

- we want to create a new type
- the decltype keyword decides the full name of that type
- the result will be a concatenation of two tuples
- `std::tuple_cat` concatenates two tuples

```cpp
template<typename V, int N>
struct gen_tuple {
    using type = decltype (std::tuple_cat (std::declval<std::tuple<V>>(),
                                            std::declval<typename gen_tuple<N-1, V>::type>()));
};

template<typename V> struct gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

Generating an N-tuple IV

- we want to create a new type
- the decltype keyword decides the full name of that type
- the result will be a concatenation of two tuples
- `std::declval` makes it possible to use member functions in decltype expressions without the need to go through constructors

```cpp
template<typename V, int N>
struct gen_tuple {
    using type = decltype (std::tuple_cat (std::declval<std::tuple<V>>(),
                                            std::declval<typename gen_tuple<N-1, V>::type>()));
};

template<typename V> struct gen_tuple<0, V> {
    using type = std::tuple<>;
};
```
Generating an N-tuple V

- we concatenate two tuples:
- a tuple containing a single type V
- and the resulting type generated by gen_tuple instantiated with N-1 as the first template argument
  - that is an N-1 tuple of types V

```cpp
template<typename V, int N>
struct gen_tuple {
    using type = decltype(
        std::tuple_cat(
            std::declval<std::tuple<V>>(),
            std::declval<typename gen_tuple<N-1, V>::type>()
        )
    );
};
```

```cpp
template<typename V> struct gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

Extension to arbitrary dimension contd.

- now we have a Container template that instantiates containers for N-dimensional vectors
- while internally holding their data in N std::vectors

```cpp
template<typename T, int N>
struct vector {
    float v[N];
    T operator[](int i) {
        return v[i];
    }
};
```

```cpp
template<typename V, int N>
struct gen_tuple {
    using type = decltype(
        std::tuple_cat(
            std::declval<std::tuple<V>>(),
            std::declval<typename gen_tuple<N-1, V>::type>()
        )
    );
};
```

```cpp
template<typename V> struct gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

Walking the tuple

- our operator[] worked with fixed-size 3-vector though
- we need a way to pass an arbitrary number of arguments to the initializer list so that we can return references to N-vectors

```cpp
template<typename T, int N>
class Container {
    using Data = gen_tuple<std::vector<T, align_alloc(16)>, N>;
public:
    using Data = gen_tuple<std::vector<T, align_alloc(16)>, N>;
    Container() {}
};
```

```cpp
template<typename T, int N>
class Container {
    using Data = gen_tuple<std::vector<T, align_alloc(16)>, N>;
public:
    using Data = gen_tuple<std::vector<T, align_alloc(16)>, N>;
    Container() {}
};
```

```cpp
Vector<float&, N> operator[](int i) {
    return Vector<float&, N>(
        { std::get<0>(data)[i],
          std::get<1>(data)[i],
          std::get<2>(data)[i] }
    );
}
```
Integer sequences

- we proceed by creating a proxy templated function `ret_impl`
- given an index `i`, and a compile-time sequence of numbers `K`
- it will return an initializer_list composed of `i`-th elements of each tuple member whose index appeared in `K`

```
template<typename T, int N>
class Container {
    template<typename V, int... K>
    V ret_impl(int i, std::index_sequence<K...>) {
        return V( std::get<K>(data)[i] ... );
    }

    Vector<T&> operator[](unsigned int i) {
        return ret_impl(i, std::make_integer_sequence<int, N>{}, { });
    }
};
```

Integer sequences II

- `std::make_integer_sequence` generates a compile-time sequence of integers
  - first parameter is the type of the integer
  - second is the number of integers to be generated
  - it returns integers form 0 to `N`

```
template<typename T, int N>
class Container {
    template<typename V, int... K>
    V ret_impl(int i, std::index_sequence<K...>) {
        return V( std::get<K>(data)[i] ... );
    }

    Vector<T&> operator[](unsigned int i) {
        return ret_impl(i, std::make_integer_sequence<int, N>{}, { });
    }
};
```

Integer sequences III

- we pass the sequence to the `ret_impl` function
- that function takes the element index `i`, and an arbitrary number of integers as its arguments
- the `ellipses` will resolve to repeated calls to `std::get<K>`
  - we can think of them as being concatenated by the comma operator

```
template<typename T, int N>
class Container {
    template<typename V, int... K>
    V ret_impl(int i, std::index_sequence<K...>) {
        return V( std::get<K>(data)[i] ... );
    }

    Vector<T&> operator[](unsigned int i) {
        return ret_impl(i, std::make_integer_sequence<int, N>{}, { });
    }
};
```

Variadic Templates I

- we pass the sequence to the `ret_impl` function
- that function takes the element index `i`, and an arbitrary number of integers as its arguments
- the `ellipses` will resolve to repeated calls to `std::get<K>`
  - we can think of them as being concatenated by the comma operator
Variadic Templates II

- we pass the sequence to the ret_impl function
- that function takes the element index i, and an arbitrary number of integers as its arguments
- the ellipsis will resolve to repeated calls to std::get<K>
  - we can think of them as being concatenated by the comma operator
- the proxy function then returns vector of type V initialized with N values

```cpp
template <typename T, int N>
class Container {
    template<typename V, int... K>
    V ret_impl(int i, std::index_sequence<K...>) {
        return V({ std::get<K>(data)[i] ... });
    }

    Vector<T&> operator[](unsigned int i) {
        return ret_impl(i, std::make_integer_sequence<int, N>{});
    }
};
```

Extension to arbitrary dimension finalized

- we can now use the container for vectors of arbitrary length and numeric type
- our vector class still includes the ET engine
  - operations on individual elements work as in lecture 1
  - we can augment these with explicit vectorization form the beginning of this lecture

```cpp
const int N = 5;
using Vec = Vector<float, N>;
int NUM_ELEM = 1000000;
Container<Vec, N> cont(NUM_ELEM);
Container<Vec, N> data1(NUM_ELEM);
Container<Vec, N> data2(NUM_ELEM);
Vec s{ 1,2,3,4,5 };
for (int i=0; i<NUM_ELEM; ++i){
    data1[i] = s;
    data2[i] = s + Vec{ i, i, i, i, i };
}
```

Take Away Messages

- the length of SIMD registers is increasing, your code has to be scalable – investigate benefits of SoA, and AoSoA layouts
  - but keep in mind these are not silver bullets – test, measure, profile
- template metaprogramming can help us hide the implementation detail in a library
  - good way to test the aforementioned benefits
- we can leverage the Expression Templates – not only on single vectors, but on whole scalar/vector fields
- TMP algorithms are executed by the compiler
  - they operate on types, the result of their operations can be types or fragments of C++ code (in a sense – it’s not a preprocessor, but a part of the C++ language)
Thank you for your attention!

You can find full examples and links to additional sources at: vysko.cz/icsc2016
Introduction

- **Kim Albertsson**
  - M.Sc. Engineering Physics and Electrical Engineering
  - Currently studying for M.Sc. Computer Science

- **Research interests**
  - Automation
  - Machine learning

- **Formal methods**
  - Great way to automate tedious tasks

---

**Formal Verification – Robust and Efficient Code**

**Lecture 2**

**Why FV?**

Kim Albertsson
Luleå University of Technology

Inverted CERN School of Computing, 29 February – 2 March 2016

---

**Formal Methods**

- **Managing complexity**
- “...mathematically based languages, techniques, and tools for specifying and verifying ... systems.” Edmund M. Clarke et al.
- **Reveals inconsistencies and ambiguities**

---

**Formal Methods**

- **Specification + Model + Implementation enables verification**

- **Proving properties for a system**
  - Will my algorithm sort the input?
  - Will my soda pop be delivered in time?
  - Can the LHC interlock system end up in an inconsistent state?
Overview Series

- **Approaches**
  - Model Checking and Theorem Proving
  - Logic and automation
  - How to build an ecosystem

- **Application**
  - Robust code
  - Robust and Efficient code

Recap of Lecture 1

- FV manages complexity
- Finds errors early
  - Save money, time and possibly life
- **Proof is a demonstration**
  - Consistency of specification and implementation

Recap of Lecture 1

- Proving properties about your system
- **Model Checking**
  - Exhaustive search of state space
- **Theorem Proving**
  - Deductive approach

Introduction Lecture 2

- **Robust**
  - Code verification
  - Code synthesis
- **Robust and Efficient**
  - Verified tool-chain
  - Proof carrying code
- Bonus!
  - Concolic Testing
Recap of Lecture 1

- Theorem Proving
  - Based on First Order Logic
  - Satisfiability Modulo Theories (SMT) solvers to find application of rules
  - IVT platforms front-end with many SMT backends

Introduction Lecture 2

- Errors discovered late
  - Expensive
- FV requires
  - More effort for specification
- FV gives
  - Feedback on inconsistencies
  - Reduces late errors
  - Reduces total time

Introduction Lecture 2

- FV and static analysis related
  - Discovering bugs at compile-time rather than run-time
- Mainstream tools slowly developing
  - Free candy!
- Can we do better while minimising effort?

Introduction Lecture 2

- Trust, correctness and code
- First order logic and SMT solvers
  - Tedious to use for non-trivial problems
- What tools are available today?
Overview

- **Robust**
  - Code verification
  - Code synthesis
- **Robust and Efficient**
  - Verified tool-chain
  - Proof carrying code
- **Bonus!**
  - Concolic Testing

Robust

- **High-level verifiers**
  - VCC, Frama-C, ... many more
  - Code compilation is separate
- **Integrate with IVT Platforms**
  - Uses specialised solvers
- **Annotations directly in code**

Code Verification

- **VCC, Verifier for Concurrent C**
- **Verification of Microsoft Hyper-V**
  - As of 2009 partially complete
  - Moderately large ~100 kloc
  - Must guarantee no leakage between host and guest

Code Verification

```c
#include "vcc.h"

int max(int a, int b)
    _(ensures \result == (a > b ? a : b))
{
    if (a > b) return a;
    return b;
}
```
Code Verification

- **Frama-C**
  - Framework for Modular Analysis of C programs
    - http://frama-c.com/
  - Modular framework for static analysis
    - Focus on correctness
  - Provides verification of C programs
    - Through the Jessie plug-in
  - Uses specification to analyze code
    - More than just verification
    - e.g. value analysis, impact analysis.

/*@
ensures \result >= x && \result >= y;
ensures \result == x || \result == y;
*/
int max (int x, int y)
{
    if (a > b) return a;
    return b;
}

Code Synthesis

- Is code redundant?
- Synthesise implementation from specification
  - Requires completeness
  - Implementation is constructive proof
- Security focus
- Requires more power than first order logic

Spec. ➔ Code synth. ➔ Byte code

Coq, the proof assistant
https://coq.inria.fr
- From french word for rooster
- Successor to Calculus of Constructions (CoC)

Proof management system
- Based on higher order logic
- Functional programming language
- with sophisticated type system
Overview

- Robust
  Code verification
  Code synthesis
- Robust and Efficient
  Verified tool-chain
  Proof carrying code
- Bonus!
  Concolic Testing

Robust and Efficient

- Efficient code is always desirable
- Specification allows specific optimisations
- Explicit assumptions leveraged for better byte-code
  - Array out-of-bounds checks unnecessary
  - Automatic parallelisation of loops
  - Dereferencing null-pointers is ok!
- Similar to whole module optimisations but locally!
**Verified tool-chain**

- **Tool-chain of today**
  - No absolute guarantee of correctness
  - Trust each tool in tool chain (verified with testing)
  - Each tool by different programmers
  - Each tool must be trusted!

- **A verified tool chain**
  - Still requires trust
  - Reduces number of places where you have to put it
  - Provides consistency guarantees

**CompCert, a verified C Compiler**
- [http://compcert.inria.fr/](http://compcert.inria.fr/)
- Guarantees optimised AST retains equivalent semantics
  - Can be extended to provide byte code guarantees
- Requires a model of memory management (language semantics)

**Verified tool-chain**

- Covers a language very close to C called CLight
  - Does not support variable length arrays, unstructured switch-statements and longjmp/setjmp
  - No immediate plans for c++
- Verified by automatic + interactive methods + Correct-by-construction
- Performs consistently worse than gcc -O1, but surprisingly close to
  - 10% worse than -O1, 15% worse than -O2, 20% worse than -O3
  - Cannot handle matrix multiplications very well as of 2016-01-29
Verified tool-chain

- No example, it works as-is on your code!

Proof Carrying Code

- Extends verification
  - Cover external code
- Proof delivered alongside code
  - Secure and performant
  - Tampering leads to invalidated proofs

Proof Carrying Code

- Verification before execution
  - Clear safety policy
  - Model of language semantics
- Language models are collaborative effort
- Proof language
  - First order logic
  - Higher order logic

Proof Carrying Code

- G. Necula and P. Lee 1996
- Safe packet filters
  - Time critical application
  - High security
  - Customisability
- User-space code run in kernel space
Proof Carrying Code

- Safe sharing of code without encryption
  - OS kernel
  - Across the Internet
  - XSS, grid-like applications
- Minimal to no performance hit

Overview

- Robust
  Code verification
  Code synthesis
- Robust and Efficient
  Verified tool-chain
  Proof carrying code
- Bonus!
  Concolic Testing

Concolic Testing

- Concolic, concrete + symbolic
  - Proofs reason about area, tests about points
  - Specification used for white-box testing
    - i.e. generating test cases
- Not perfect, but efficient
  - Achieves high code coverage quickly

Concolic Testing

- Execution splits at branches
  - Constraints updated for new paths
- Symbolical path is resolved
  - When "exception" happens
  - Yields concrete example
- To figure out concrete path
  - Use constraint solver (SMT)
Concolic Testing

```c
int get_sign(int x) {
    if (x == 0)
        return 0;
    if (x < 0)
        return -1;
    else
        return 1;
}
```

Concolic Testing

- **KLEE**, from the painter Paul Klee
  - https://klee.github.io/
- **Applied to unix’ coreutils in 2008**
  - Found several serious bugs
  - Coreutils is thoroughly tested through exposure
  - Successfully handled programs of up to 10 kloc in ~1h per program
  - Beat hand constructed test-suites built over 15 years
- **Integrates well with standard tools (llvm)**

```bash
$ llvm-gcc --emit-llvm -c -g get_sign.c
$ klee get_sign.o
```

```c
int main() {
    int a;
    klee_make_symbolic(&a, sizeof(a), "a");
    return get_sign(a);
}
```

```bash
$ ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['get_sign.o']
num objects: 1
object   0: name: 'a'
object   0: size: 4
object   0: data: 1
...
$ ktest-tool --write-ints klee-last/test000002.ktest
object    0: data: -2147483648
$ ktest-tool --write-ints klee-last/test000003.ktest
object    0: data: 0
```

```bash
$ ktest-tool --write-ints klee-last/test000001.ktest
```

```bash
ktest file : 'klee-last/test000001.ktest'
args       : ['get_sign.o']
num objects: 1
object   0: name: 'a'
object   0: size: 4
object   0: data: 1
...
```

```bash
$ test-tool --write-ints klee-last/test000002.ktest
object    0: data: -2147483648
```

```bash
$ test-tool --write-ints klee-last/test000003.ktest
object    0: data: 0
```
Recap Lecture 2

- **Robust**
  - Code verification
  - Code synthesis

- **Robust and Efficient**
  - Verified tool-chain
  - Proof carrying code

- **Bonus!**
  - Concolic Testing

---

Recap Lecture 2

- **Robust**
  - Code verification
  - Code synthesis

- **Robust and Efficient**
  - Verified tool-chain
  - Proof carrying code

- **Bonus!**
  - Concolic Testing

---

Recap Lecture 2

- **Formal methods, applied where correctness is important**
  - Large collaborative projects
  - Verified tool-chains
  - ...

- **Benefits for all**
  - Robustness and quality

- **Taught in school?**
Thank you
Positron Emission Tomography (PET)

- Radioactive nucleus decays in a $\beta^+$ reaction.
- $\beta^+$ annihilates -> two antiparallel 511 keV photons emitted.

In Basics of PET Imaging Physics, Chemistry, and Regulations, Gopal B. Saha
Positron Emission Tomography (PET)

- Detection of the two photons in the same time window is called a coincidence event, and the line is called LOR (line of response).
- A patient is injected with a radiotracer (usually FDG, a radioactive replacement of the deoxyglucose) that accumulates in a region of interest.
- The detection of coincident events allows the image reconstruction.

PET Scanner

Applications

Functional clinical studies
- Oncology
- Cardiology
- Neurology

Pre-clinical research
- New drugs
- Diseases research

Data Acquisition

Gama photon 511 keV

Optical photon (~eV)

Photomultiplier

Scintillator (LYSO)

Discrimination and amplification

Data acquisition
PET Simulations

- Simulations play an important role in Medical Research
  - Develop and optimize new scanners and techniques
    - PET-CT, PET-MRI, PET-CT-MRI, SPECT-CT, Optical Imaging, etc
  - Discover of new drugs
    - Biomarkers
  - Study of diseases and new treatments.
    - Cancer, Alzheimer’s
    - Radiotherapy and Hadrontherapy

GATE

- GATE (Geant4 Application for Tomographic Emission)\textsuperscript{[1,2]}
  - Monte Carlo application
  - Allows the use of simplified macros as primary input mechanism (no C++ knowledge is needed for most of the applications);

GATE Architecture

- Geant4 engine is used for generate and tracking particles
- In a typical simulation:
  - Source with 300 MBq \textsuperscript{18}F-FDG generates $\sim 10^{11}$ decays during a 30 min scan (plus secondary particles)
  - Each event (decay) is independent – Monte Carlo simulation
  - Very time consuming to track all this particles
- Parallelization first approach:
  - Split job into smaller jobs (time slices) in a grid

\textsuperscript{[1]} Jan, S. et al., “Gate: a simulation toolkit for PET and SPECT”
\textsuperscript{[2]} Santina, G. et al., “Evolution of the GATE project: new results and developments”
Image Reconstruction algorithms

- **Analytic methods** (fast, simpler and easier to implement). E.g. Retroprojections.
- **Iterative methods** (slower, more complex but usually with better performance).

From Emission Tomography: The fundamentals of PET and SPECT.

Image Reconstruction Methods

- **The “inverse” problem of the acquisition. Why?**
- **Analytical**:
  - Involves the reconstruction of an image from its X-Ray transform
  - Deterministic problem, usually “ignoring” real data.
  - Efficient and non-iterative algorithms
- **Iterative**:
  - Start with a guessing image
  - Finite iterations over images
  - Requires heavier calculations
  - Suitable for more complex problems

Analytical reconstruction

- **Uses X-Ray transforms**:
  1. Returns all the possible line integrals of an image \( f(x,y) \).
     2D example:
     - The X-Ray transform is the operation \( f(x,y) \rightarrow p(x,\phi) \)
     - \( p(x,\phi) \) is the 1D projection of \( f(x,y) \) for a given angle \( \phi \)

\[
p(x,\phi) = \int f(x,y)\,dy,
\]

\[
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
= 
\begin{bmatrix}
  \cos\phi & -\sin\phi \\
  \sin\phi & \cos\phi
\end{bmatrix}
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix}
\]

From Emission Tomography: The fundamentals of PET and SPECT.
Analytical reconstruction

- Uses X-Ray transforms:
  3. 2-D FBP algorithm
    - Compute 1-D FT of the projections along the detector axes
    - Apply:
      - ‘Ramp-filter’ in the frequency space (1-D Convolution)
      - 1-D iFT to obtain filtered projections
      - Back-projection operator to obtain the image

\[
f(x, y) = \int \int |u_x| |P(v_{xy}, \phi)| e^{2\pi i u_x x + 2\pi i u_y y} d|v_x| d\phi
\]

- Parallelization can be obtained, for example:
  - Projections for different $\phi$ may be calculated in different processors

Iterative Reconstruction

- Different iterative methods are available. Ex: ML-EM, OSEM, ...

- Maximum Likelihood - Expectation Maximization (ML-EM)
  - First introduced for image reconstruction in 1982 by Shepp and Vardi, remains the basis algorithm for iterative statistical image reconstruction
  - It leads to the iterative equation and chart[1]

\[
f^{(n+1)}(x,y) = \sum_{i=1}^{N} \sum_{k=1}^{K} H_{ik} \sum_{j=1}^{J} f^{(n)}(x',y') P_{ij,k}
\]

- Image at iteration $n+1$

Acceleration tools

- Many tools and approaches have been developed in the last years to extract as much performance as the recent computers can give.

- Some examples:
  - Multi-thread (sharing memory): e.g. OpenMP, TBB, CUDA
  - Multi-CPU (splitting memory): e.g. MPI
  - Intrinsic parallelism: split the work in smaller parts

- Monte Carlo simulations are well suitable for intrinsic parallelism[1].

Data parallelism vs Task Parallelism

- We can think about parallelism in two ways:
  - **Task parallelism**: Simultaneous execution of different tasks on the same or different data
  - **Data parallelism**: Simultaneous execution of the same task/function (single instruction, multiple data – SIMD) for various elements on an ensemble

- The use of one or the other approach depends on the user application, usually the use both is the best option

OpenMP

- First released in 1997
  - Designed to ensure an ordered access of different threads to shared data and to be a standard notation among different SMP (Symmetric multiprocessing) architectures.
  - Supports Fortran, C and C++.
  - It’s implemented in many commercial and Open Source compilers.
  - It’s a set of compiler directives, library routines and environment variables as an extension of C, C++ and Fortran standard compilers.
  - For simple applications, only few code lines may be needed.

OpenMP

- Uses the Fork-join[1] model of parallel execution

    Master thread
      Fork
        Parallel thread 1
        Parallel thread 2
      Join
        Master thread

- For-loop example C++:

```cpp
//Typical C++
for (int i = 0; i < 8; ++i) {
  do_some_task(i);
}
```

```cpp
//C++ with OpenMP
#include <omp.h>
omp_set_num_threads(8);
#pragma omp parallel for
for (int i = 0; i < 8; ++i) {
  do_some_task(i);
}
```


Intel® Threading Building Blocks

- “Is a popular software C++ template library that simplifies the development of software applications running in parallel” from https://www.threadingbuildingblocks.org/faq

- Unlike OpenMP, TBB makes use of the typical programming style of C++

- It is focused for tasks instead of threads

```
Task queue

Thread 1

(...) 

Thread N
```
Intel® Threading Building Blocks

- For-loop example C++:

```cpp
#include "tbb/tbb.h"
using namespace tbb;

void Application(size_t size) {
    parallel_for(size_t(0), size, size_t(1), [=](size_t i) {
        do_some_task(i);
    });
}

int main(){
    const size_t size = 8;
    Application(size);
    return 0;
}
```

CUDA®

- Compute Unified Device Architecture, introduced in November 2006 by NVIDIA
- It allows the use of GPUs to solve complex parallelization problems that general CPUs have more difficulty/need more time to handle.
- To make use of it, the installation of a software environment is needed. Its possible to develop applications using different programming languages (C, C++, Fortran, Java, etc).
- Its being widely used in scientific applications nowadays.

Case Study 1

Parallelism in GATE

Parallelism in GATE

- Two possibilities for simulations acceleration:
  - **Time split**:
    - Events are dependent between them.
    - Jobs are separated in the time domain into smaller jobs and distributed among a queuing system.
    - Better for imaging applications, due to time dependent effects
  - **Events split**:
    - Events are independent
    - Each job will only simulate a fraction of the total number of events
    - More suitable for Dose Applications (Radiotherapy, Hadrontherapy) because no dependence between particles is demanding

- Using multithreading capabilities with CUDA:
  - Only **highly-demanding** parts of the simulation go to CUDA kernels - hybrid simulation CPU+GPU
  - Phantom part uses CUDA
    - One particle per thread
    - Specific kernels for physics effects
  - Detector part uses the CPU
    - Time dependence is vital to simulate electronic chain, data acquisition and reconstruct images with accuracy
  - Example: Hybrid-GATE project, funded for 36 months by the French National Research Agency, to accelerate GATE simulations using CPU/GPU capabilities

- Geant4 code has been moved to GPU (random number generator, photon physics effects, etc)[1]

---

**GATE CPU+GPU**

- **Voxelized phantom**
  - GATE (CPU)
    - $10^9$ particles
    - ~90 s
  - GATE (CPU+GPU)
    - $10^9$ particles
    - <2 s

---
Case Study 2

Image Reconstruction: OpenMP vs TBB

- LM OSEM, an iterative algorithm for 3D image reconstruction:
  - PET events (LORs) are split into equally spaced subsets
  - For each subset \( l \in 0, ..., s - 1 \), is calculated \( f_{l+1} \):

\[
f_{l+1} = f_l; \quad c_l = \frac{1}{A_N} \sum_{i \in S_l} (A_i)^2 \frac{1}{A_i};
\]

- Where \( f \) is a 3D image in vector form with dimensions \( n = (X \times Y \times Z) \).
- \( A \in \mathbb{R}^{m \times n} \) and the element \( a_{ik} \) of the row \( A_i \) is the length of the LOR correspondent to the event \( i \) and the voxel \( k \), calculated with Siddon's algorithm\[^1,2\].


LM OSEM algorithm has 3 nested loops:
- 1 outer loop over all subsets
- 2 inner loops, one for the summation, another for the iterations

Using OpenMP

Race condition might happen here

Using TBB

- For TBB, code modifications are needed

```cpp
Using TBB

class ImageUpdate {
    double const c_1;
    double const c_2;
public:
    ImageUpdate(double c_1_, double c_2_, double t_1_) :
        c_1(c_1_), c_2(c_2_), t_1(t_1_) {}
    void operator()(const blocked_range<int>& r) const {
        for (int k = r.begin(); k != r.end(); k++) {
            if (image(k) > 0.0 && c_1(k) > 0.0) {
                f(k) = c_1 * f(k) / c_2;
            }
        }
    }
};
```
Image Reconstruction

OpenMP vs TBB

- LM OSEM algorithm has 3 nested loops:
  - 1 outer loop over all subsets
  - 2 inner loops, one for the summation, another for the iterations
- For TBB, code modifications are needed

```c
for (int i = 0; i < subsets; i++) {
    // compute c, l
    parallel_for(
        block_range<2>(0, subset_size, SUBSET_SIZE),
        SubsetComputation(f, c, l, event_buffer, projection);
    )
    // compute f, L
    parallel_for(
        block_range<2>(0, image_size, IMAGE_SIZE),
        ImageUpdate(f, c, l);
    )
}
```

For TBB, code modifications are needed.

Implementation results:

1. Preventing race conditions (using mutexes or critical sections), OpenMP has shown better performance over TBB.
2. Using OpenMP requires very little program redesign, contrary to TBB
3. TBB is more suitable for the design of new applications from the scratch, while OpenMP is preferable to redesign already developed code.

Running machine: dual quad-core (AMD Opteron 2352, 2.1GHz with

MLEM Acceleration

- The most of the time spent in projection and backprojection operations between the detected LORs and the image voxels.
- Acceleration of the method has been achieved in recent years, using single GPUs.
- Image reconstructions for real-time applications in hospitals demands even higher speed-up of calculations.
- Multi-GPUs systems are being used but the communication between GPUS is a bottleneck.
- GeForce GTX 480 and GTX 285 were tested

[1] Distributed MLEM: An Iterative Tomographic Image Reconstruction Algorithm for Distributed Memory Architectures
Craig S. Levin et al, IEEE TRANSACTIONS ON MEDICAL IMAGING, VOL. 32, NO. 5, MAY 2013
MLEM Acceleration

- In this work, the common MLEM (DG) algorithm is described as a special case of a general optimization problem
- A new MLEM (DMLEM) is derived by maximizing the same likelihood function as the common MLEM, but adapted to a multi-GPU system
- The new algorithms performs several iterations in sub-problems of the original problem, but minimizing the same objective function

Conclusions

- In Medical Physics, several applications has taken advantage of the rapidly increase of the computation resources that are available, specially for:
  - Simulation of the operation of existent and new scanners.
  - Image reconstructions both for scientific or preclinical research and in real time clinical practice.
- New software tools has been used to accelerate these tasks:
  - OpenMP and TBB, using the multi-CPU capabilities.
  - Nvidia CUDA, taking advantage of the power of graphic cards commonly available.
- The use of one tool instead of the other will depend manly on:
  - The desired application (new or renewed).
  - The available resources (time, funds, hardware).
  - The will of the person in charge.
Lecture 2
Event Reconstruction

Event Reconstruction

- Triggered detector collision data → particle interactions.
- Seek the following information as input for physics analysis:
  - What particles were created?
  - Where were they produced?
  - What were the parent particles?
- To find this, perform (at least):
  - Tracking: Reconstruct particle trajectories into tracks.
  - Vertexing: Group particles into vertices.
  - Particle ID: Find the particle identification of each track (e.g. a muon, electron etc.).

- Requirements: Usually anti correlated - a fast algorithm often leads to inefficiency and impurities (see later).
  - Fast.
  - Good quality for physics analysis.

Tracking Algorithms
Aim: to play a game join the dots at 1kHz with many fake dots.

- Tracking particles through detectors involves two step.
  - Pattern recognition: identifying which detector hits for a track.
  - Track fit: approximate the path of the particle with an equation.
- No one size fits all solution.
  - Many detectors use different combinations of algorithms (e.g. LHCb uses 4 different algorithms for difference combinations of sub detectors, but basic ideas are the same).
  - Usually a trade off between:
    - Efficiency: fraction of real tracks found
    - Purity: fraction of tracks that are real
    - Computational speed.

Typically these two are anti correlated: a good efficiency typically has a bad purity, and vice versa. Both good efficiency and purity is usually computationally expensive - see later.

One of the hardest cases - Pb collisions in ALICE, a real event.
Consider tracking in the Vertex Locator (VELO) of LHCb.
- The VELO is a silicon strip detector, consisting of 42 layers.
- Gives an x, y and z position.

Looking side on:
- Particle tracks clearly visible to eye.
- Extra hits present, typically electrical noise or secondary short tracks.

Recall data points in the format:
(x, y, z, time)

Time resolution only accurate to which collision the particles come from (25ns, sometimes worse...).

Have to find an algorithm to track using this information and in these conditions. Many choices - consider the following (LHC) examples:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinatorial</td>
<td>Form every track from each possible combination of hits.</td>
<td>$O(n^{\frac{3}{2}})$</td>
</tr>
</tbody>
</table>
### Tracking - Pattern Recognition Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinatorial</td>
<td>Form every track from each possible combination of hits.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>Access each track by quality (e.g. $\chi^2$) and tag.</td>
<td></td>
</tr>
</tbody>
</table>

#### Hough Transform

- Transform points into a system where clusters form.
- If straight tracks, take the difference between consecutive hits.
- Group (e.g. in a histogram) and tag peaks.

### LHCb VELO data event (2d projection, top half)

![LHCb VELO data event](image-url)
### Tracking - Pattern Recognition Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinatorial</td>
<td>Form every track from each possible combination.</td>
<td>nTracks!</td>
</tr>
<tr>
<td></td>
<td>Access each track by quality (e.g., $\chi^2$) and tag.</td>
<td></td>
</tr>
<tr>
<td>Hough Transform</td>
<td>Transform points into a system where clusters form.</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>E.g., for straight tracks, take the difference between consecutive hits.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group (e.g., in a histogram) and tag peaks.</td>
<td></td>
</tr>
<tr>
<td>Seeding</td>
<td>Form seeds from pairs of hits on a sub set of the detector.</td>
<td>nlog(n)</td>
</tr>
<tr>
<td></td>
<td>Extrapolate the seed and count hits intercepted.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tag if sufficient number of hits.</td>
<td></td>
</tr>
</tbody>
</table>

**LHCb VELO data event (2d projection, top half)**

---

### Tracking - Pattern Recognition Algorithms

- Recall three main factors in choosing such algorithms:
  - Efficiency: fraction of real tracks found
  - Purity: fraction of tracks that are real
  - Computational speed.

- Toy simulation for LHCb VELO:

**LHCb VELO toy event (2d projection)**

---

**Question:** which algorithm to pick?

Even in the case where efficiency and purity are constant?

Still not enough information…

---

### Tracking - Pattern Recognition Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinatorial</td>
<td>Form every track from each possible combination.</td>
<td>nTracks!</td>
</tr>
<tr>
<td></td>
<td>Access each track by quality (e.g., $\chi^2$) and tag.</td>
<td></td>
</tr>
<tr>
<td>Hough Transform</td>
<td>Transform points into a system where clusters form.</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>E.g., for straight tracks, take the difference between consecutive hits.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group (e.g., in a histogram) and tag peaks.</td>
<td></td>
</tr>
<tr>
<td>Seeding</td>
<td>Form seeds from pairs of hits on a sub set of the detector.</td>
<td>nlog(n)</td>
</tr>
<tr>
<td></td>
<td>Extrapolate the seed and count hits intercepted.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tag if sufficient number of hits.</td>
<td></td>
</tr>
</tbody>
</table>

**Toy simulation for LHCb VELO:**

**Any use case for green? Curves!**
Tracking - Pattern Recognition Algorithms

- Typically use a combination of these algorithms. Each example taken from LHC activity at testbeams:
  - Low occupancy, so fast.
  - Efficient and pure.
  - Hough transforms used for more complicated shapes (e.g. rings in LHCb RICH*).
  - All LHC experiments use seeding extensively (highest occupancy).

*Often not needed to actually reconstruct rings.

Track Fitting

- Tracking particles through detectors involves two steps:
  - Pattern recognition: identifying which detector hits for a track.
  - Track fit: approximate the path of the particle with an equation.
  - Typically use a Kalman filter. Basic steps:
    - Track is approximated as a ‘zig-zag’ (fewer free parameters than co-ordinates!).
    - Start with seed or estimate of track parameters (e.g. straight line fit).
    - Propagate to the next plane (approximating B field, account for scattering in material).
    - Predict position of next particle, weighting by closest hits (needs too be tuned).

Track Refinement

- Common to tune pattern recognition to be efficient and impure → refine selection later using full particle information.
  - Can use $\chi^2$ to find well fitting tracks.
  - Can also use/combine with other parameters:
    - Number of hits (complimentary information to $\chi^2$).
    - Fits from different sub detectors
  - Typically build an MVA out of different quality parameters - LHCb uses a neutral net.

  - Caution: If fake/ghost tracks are formed from parts of real tracks, they may be lost.

Tracking Notes

- Kalman filter and pattern recognition can be merged into a single step → computationally more efficient (see later).
  - Detector hits can sometimes form part of multiple tracks.
    - Detector spatial resolution too low to separate near tracks.
    - Secondary tracks produced by interactions with the detector material.
**Vertexing (Briefly)**

- Vertexing involves clustering tracks that originate from the same point.
- Easy in cases where vertex location is known - extrapolate all tracks and apply selection criteria.
- Else, Physics input can narrow search region significantly.
- Can use analytic methods (e.g. distance of closest approach) to seed search.
- Common to seed by projecting into 2D plane and searching for point of high "track density" (essentially a peak finding/clustering problem).

**Particle ID (Briefly)**

- Classify each track as a type of particle event by event:
  - Needed to refine selections for offline analysis (remove background).
  - Many kinds of particle:
    - Not just fundamental particles, also composite hadrons (e.g. Pion, Kaon).
  - Some easy cases:
    - RICH detector at LHCb uses Cherenkov radiation:
      - Light emitted when a particle slows passing through a material.
      - Emission is isotropic, and forms rings on detectors.
      - Often not required to reconstruct the ring itself - instead, test different hypothesis.
      - Sometimes multiple solutions (e.g. high momentum) - can still assign probabilities.
Event Reconstruction Implementation

- Each reconstruction stage typically (sometimes by necessity) follows sequentially, e.g:
  - Such a chain can be performed for a single event, or large set of events.
  - Reminder: each event is (usually) statistically independent of each other.
  - Strategy for single core is obvious, but for multi core, not so much.
  - Nowadays, reconstruction involves tens of thousands of CPUs worldwide - need efficient strategy.
  - Currently limited by memory:
    - E.g.: CMS end of 2011 could only 6 out of 8 cores on average.

Event Reconstruction Optimisations

- Some advantages:
  - Data is essentially a list of collision data.
  - Each collision is (mostly) separate from the next.

- Some disadvantages:
  - Many packages needed, not all thread safe.
  - Large overhead of memory to run even a small job, can quickly reach $O(1)$GB:
    - (e.g. calibration and alignment values).
  - External libraries.

Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Run multiple jobs on the same multi core machine.
    - Very common strategy currently.
    - Split input file (easy), and merge output file (easy).

Potential pitfalls?
Memory!
Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Many segments can be performed in parallel.
    - Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Potential pitfalls? Amdahl’s Law! Typically limited at factor of 5 speedup.

• Many segments can be performed in parallel.
• Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Process multiple events concurrently in one big job.
    - Large overhead of memory (e.g. calibration constants) can be shared between threads.

Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Many segments can be performed in parallel.
    - Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Potential pitfalls? Amdahl’s Law! Typically limited at factor of 5 speedup.

• Many segments can be performed in parallel.
• Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Process multiple events concurrently in one big job.
    - Large overhead of memory (e.g. calibration constants) can be shared between threads.

Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Many segments can be performed in parallel.
    - Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Potential pitfalls? Amdahl’s Law! Typically limited at factor of 5 speedup.

• Many segments can be performed in parallel.
• Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Event Reconstruction Optimisations - Parallelism

- Many possible optimisation strategies:
  - Parallelism - many possible options:
    - Many segments can be performed in parallel.
    - Many segments (e.g. tracking) depend on those previous, which can give long serial sections.

Potential pitfalls? Amdahl’s Law! Typically limited at factor of 5 speedup.

• Many segments can be performed in parallel.
• Many segments (e.g. tracking) depend on those previous, which can give long serial sections.
The Grid

- Majority of reconstruction jobs performed on the Grid (excluding HLTs):
  - ~2 million jobs processed each day.
  - 170 computing centres in 42 different countries.

- Managed by each institute:
  - Large variation of technology and software used.

Centres are organised in a tier system.

- Tier 0 (CERN):
  - Receives data from experiments.
  - First copy of data to tape.
  - Distribute reconstruction in Tier1.
  - Re-reconstruction when LHC is not on.

- Tier 1 (distributed, 10GB/s connection):
  - Offline event reconstruction performed here.
  - Store and distribute jobs on Tier 2.

- Tier 2 (distributed):
  - University sites.
  - Ideal for analysis jobs performed on reconsulted data (e.g. Higgs Searches).

Event reconstruction typically involves a set of algorithms executed in a particular order (often by necessity):

- Each algorithm offers varying efficiency, purity and speed.
- Tracking in particular often a large sequential portion of event reconstruction.

Parallelism required to utilise available computing resources:

- No obvious methods to parallelise.
- Until recently, ok to have multiple processes on multi core machines:
  - Now memory issues - multi threading required to reduce large overhead.
  - In process of multi threading reconstruction packages.

Many computing resources available, not just relying on CERN:

- The Grid is used for processing many reconstruction and analysis jobs.
- Spread worldwide, using a variety of different technologies.

With flat computing budgets and increased luminosities, reconstruction is challenging:

- Clever ideas welcome!

References

[3] CMS upgrade trigger TDR.
[4] LHCb trigger TDR.
[9] Study of a Fine Grained Threaded Framework Design, Christopher Jones,
[10] Using the CMS Threaded Application in a Production Environment, Christopher Jones, CHEP 2015,
  http://indico.cern.ch/event/304944/session/2/contribution/120/attachments/57895796856/Threading_Production_CHEP2015.pdf
[11] many more...
Introduction

- **Detector simulation** at the LHC is the most computing intensive activity (~50% of the GRID resources)
- **Detectors** are complex, very precise and ambitious: this implies modern modelisation tools for geometry and response. **Events** are busy and characterised by an unprecedented energy scale with hundreds of particles to be traced and high energy showers to be accurately simulated. Furthermore, high luminosities imply many events in a bunch crossing and many bunch crossings to be considered at the same time. In addition, backgrounds not directly correlated to bunch crossings have also to be taken into account.
- Features of detector simulation at the LHC (with emphasis on full simulation) will be described and CPU and memory figures will be given in Lecture 1.
- An extrapolation to the FCC-hh case will be tried and possible solutions (mostly ATLAS-inspired) to be adopted will be discussed in Lecture 2.

Disclaimers

1. Most of the examples shown are ATLAS-based
   - It’s the easiest choice for me:
     - I am a member of the ATLAS collaboration
2. Results can be quite dated
   - Current publication policy does not allow to show unpublished results
   - The message does not change!
3. Quite hard to summarise ~20 years of activities of many people in 2 lectures
   - Some items will be only mentioned…
   - …some other items will not be mentioned at all!
   - Mistakes may have crept in, apologies
Recap of Lecture 1

- What did you learn?
  - How a High Energy Physics detector works
  - How complex it is
  - Which flavors of simulation are used
  - What are the differences among MC Smearing/Full/Fast simulation
  - What does Geant4 provide and what must be implemented
  - Which are the limitations
  - How the LHC deals with them
  - What are the performances (in term of CPU, memory, etc) and how they change depending on physics (particle type, phenomena, etc)

- Take away messages:
  1. At the development and the early analyses of the LHC experiments, full simulation was often the solution of choice
  2. While this strategy was very successful from the physics point of view, the complexity of the simulation programs led to hit the CPU/memory limits
  3. Pile-up is an issue now and it will become worse in future

Outline – Lecture 2

1. Recapitulating simulation at the LHC-RUN 1
2. LHC upgrade
3. Future Circular Collider
4. Physics requirements and Computing Resources
   - Fast simulation
     - Calorimeter
     - Inner Detector
     - Integrated Simulation Framework
   - Full Simulation
     - GEANT4 improvements
     - GeantV challenge

Simulation at the LHC-Run1

- ATLAS: order of 10 billion events/year (~1/2 grid allocation)
  - Aiming for data/MC 1/1 ratio (1/3 full + 2/3 fast)
  - Up to 6 minutes/event MB, largely dominated by calorimetry

- CMS: order of 10 billion events/year
  - ~20-100 sec/event full, ~1 sec/event fast

- LHCb: few billion events produced
  - 100/1 (rare signals) 1/100 (rest)
  - Simulation time: 1 min-1 hour/event range
  - Digitization: less than 1% of transport

- ALICE: ~1 billion simulated events (full)
  - Taking more than 50% of GRID allocation
  - p-p at ~60s/event, Pb-Pb MB at ~ 10 min/event
    - Transport and generation: 70% (mostly ZDC)
    - Digitization: 30% (mostly TPC – ExB, diffusion)

How about the future?

- LHC upgrade and High Luminosity LHC – HL-LHC
  - Detectors mostly unchanged, much more challenging experimental conditions

- ILC-Clic (not discussed here)

- Future Circular Collider - FCC
  - FCC-ee / FCC-hh / FCC-eh (focus in this lecture will be on FCC-hh)
  - Unprecedented energy scales
  - Extremely challenging experimental conditions
  - Completely new detectors → new technologies → R&D
The LHC upgrade challenge

- Number of collisions increasing
  - ~ 3-5x for Run 2
  - ~ 10x for Run 3
  - ~ 100x for HL-LHC
- Increase in energy and pileup
  - Non-negligible impact in full simulation time
- Increase of a factor of ~100 in integrated luminosity makes simulating a large fraction of the data sample a prohibitive challenge!

FCC-hh

- The goal of the FCC-hh is to provide proton-proton and ion collisions at much higher energy than the LHC
  - Target centre-of-mass energy is 100 TeV
  - Assuming a nominal dipole field of 16 T such a machine would have a circumference of the order of 80-100 km
  - The machine should accommodate 2 main experiments that are operated simultaneously and have a peak luminosity of 1 - 5 x 10^{34} cm^{-2}s^{-1}

From the LHC to the FCC

- Future Circular Collider (FCC) is an integral conceptual design study for post-LHC particle accelerator options in a global context
  - Two colliders in one: the project comprises a high energy proton-proton collider with a centre-of-mass energy of 100 TeV (FCC-hh), as well as a 80-400 GeV high luminosity e^+e^- machine (FCC-ee a.k.a. "TLEP", not discussed here)
  - It would also allow the use of heavy ions and e-p collisions
  - Powerful search for new phenomena → FCC-hh
    - High energies and rare processes
  - Precise measurements of the Higgs boson and other Standard Model particles → FCC-ee
FCC Software – Scenario
- Flexibility is required to geometry and simulation in order to support FCC-hh, FCC-ee and FCC-eh at the same time and update descriptions easily as soon as the studies progress
- To speed up performance, parallelisation is necessary
  - Pure hardware improvements saturated
  - Software has to be consistent with the parallel hardware architecture
- CPU is faster than memory
  - Software has to be adapted with the physical memory hierarchy
- Architecture is changing heavily
- Software itself progresses
  - New programming languages
  - C++ rapid evolution

FCC Software – Getting Going
- The easiest decision (?!): use existing solution from the LHC. This implies:
  - Framework → Gaudi (used by LHCb, ATLAS, few others)
  - I/O → Root
  - Simulation → Geant4
  - User Analysis → C++, Python
- Re-use software developments from ILC/CLIC
  - Detector Description → DD4Hep (details in the extra slides)
- Improve fast simulation VS full simulation
  - Geant4 fastsim, GFlash, ATLAS Fast simulations (details in the next slides)
- Improve data model
  - Neither LHC experiments’ nor ILC/CLIC are the best solution (not discussed here)

FCC Software – Simulation
- Different detector studies have to be supported by the simulation:
  - Different level of details (again Smearing VS FastSim VS FullSim)
    - Delphes and HepSim
    - Fast simulation in Python
    - Integrated fast/full simulation with Geant4
- R&D and Detector studies still require full simulation
  - Invest on improving performance of existing tools and exploit new trends
    - Multithreading
    - Parallelisation
- For full detector simulation integrate fast simulation models since the beginning
  - Re-use and improve on existing simulation models as developed in the LHC experiments
  - Hopefully integrate them into the main simulation engine

FCC-hh Simulation Challenges
- Higher energy in the centre of mass (~factor of 7) → correspondingly more energy in the detector
  - Simulation time will go up!
- Accurate simulation of the forward regions is a requirement → even more energy!
  - Normally parameterised at the LHC experiments
  - Simulation time will go up!
- Higher particle multiplicity (~factor of 2) means even more simulation time!
  - Full simulation of a FCC event might end up taking a factor 10-100 more than a corresponding event at the LHC
  - Need to define careful parameterisation strategies
    - For low energy particles
    - For high energy showers
    - For more efficient tracking
FCC-hh Simulation Challenges

- Pile-up simulation is the biggest issue!
  - Between 500 and 1000 simultaneous interactions per bunch crossing!
  - 5 ns bunch spacing means a lot of crossings to deal with during the detector integration time
- Strategy used until now in ATLAS and CMS (pile-up simulation at digitization level - Lecture 1) will not work at the FCC!
  - Too many events to be kept in memory at all times
  - ...or too much I/O if the events have to be read in on demand
- On-the-fly simulation of bunch crossings will simply be too demanding
  - Hours of CPU to simulate one bunch crossing
  - How about out-of-time pile-up?
- What to do? …

The FCC-hh scenario is daunting!

From physics, energy reach, instantaneous luminosity, pile-up, background, etc, the problem complexity will increase by orders of magnitude

Resources allocated to simulation are barely sufficient for todays LHC Experiments

New paradigms?
The FCC-hh scenario is daunting!
From physics, energy reach, instantaneous luminosity, pile-up, background, etc, the problem complexity will increase by orders of magnitude

Resources allocated to simulation are barely sufficient for today's LHC Experiments

New paradigms?
FAST... FASTER... FASTEST...

A route to fast(er) simulations

- Attempts to have fast simulation programs in ATLAS and CMS since the beginning
- As an example: ATLFAST
  - A very early fast simulation implementation (1990s) in ATLAS
  - Based on Monte Carlo smearing of generated particles
  - Limit: Different objects (wrt reconstruction) written out
  - Performance was good enough to make it the tool of choice for early ATLAS TDRs
  - Its functionality was overcome in ATLAS by other fast simulation models
- More modern tools now available
  - Pretty Good Simulator PGS (Fortran)
  - Delphes (C++, now the de facto standard in HEP)
  - etc

A route to fast(er) simulations - GFlash

- PROBLEM:
  - Calorimeters are the biggest bottleneck in any Geant-like simulation
  - Due to the big number of particles to be traced (showers)
- GOAL:
  - Speed up simulation in electromagnetic calorimeter (ECAL) without sacrificing too much precision
- A SOLUTION → GFlash
  - Package from H1 (HERA), written in Fortran and working within the framework of Geant3 (Fortran)
  - Provides a set of equations & parameters to describe electromagnetic shower profiles
  - Adapted for CDF calorimeter simulation at Tevatron Run-II
    - CPU gain up to 100 (CDF)
  - Ideal for
    - simple geometry
    - repetitive sampling structure
    - single effective medium
  - Adapted to Geant4 by CMS & ATLAS

GFlash works very well for CMS
- Rather homogeneous calorimeter
A route to fast(er) simulations - GFlash

- In ATLAS, worked pretty well for high-pT e± and γ (doesn't work for π±)
  - Regular showers in a sampling calorimeter
  - Tuned energy and shower shapes
  - Very nice speed improvement

- CPU time gain is irrelevant for full events!
  - Dominated by low-energy objects (mainly hadrons) → no parameterisation

- ATLAS had to find a different approach...

A route to fast(er) simulations

- Critically important to understand where the time goes (in terms of subdetectors and particle type), and what the simulation is doing
  - Do neutrons matter?
  - Which subdetectors take the most time?
  - What particles in there are taking lots of time?

- Despite of what one can think, hadrons are irrelevant!
- Most particles are low energy, the high energy ones become low energy quickly...

A route to fast(er) simulations - shower libraries

- Need to treat low energy particles
  - In Geant4, about half of the time is spent on particles below 10 MeV

- Now using pre-simulated libraries of showers for e±/γ
  - Run simulation, catch low-energy particles, record their positions and momenta
  - Simulate each low-energy particle (<1 GeV), save the energy deposits in the sensitive volumes
  - Trim and adjust the showers to reduce the number of low-energy hits
  - Inject pre-simulated showers with the appropriate properties (detector region, energy, etc), scaling, rotating, etc to match the input particle

- Now a part of the ATLAS default simulation in the forward calorimeter
  - It saves a factor of ~2 in sim time

- Considered, but not yet adopted for the central region
- Also considered for n/π showers, but they don’t make up much simulation time

A route to fast(er) simulations – FastCaloSim/Atlfast2

- Parameterization of calorimeter showers based on single particle simulation

- Starts parameterization from the end of Inner Detector
  - Key to solving the problem that the other parameterizations had

- Can’t be done as a first-principles parameterization – uses histograms to describe showers
  - Lots of work to ensure that the tails of distributions are fully described, including punch-through effects in the calorimeter and tails of MET

- Deposits energy directly into calorimeter cell structure – No need for any finer granularity, of course

- Speed gains are about 10-20x over full simulation
A route to fast(er) simulations – FastCaloSim/Atlfast2

- Advantage of a fast simulation like this: we can tune to data
  - Not totally obvious whether this is something that you want to do
- Quite good agreement in most variables; deemed “safe” to use for all analyses
  - except those that use, for instance, jet substructure

At this point, Inner Detector tracking becomes the most CPU expensive part of the simulation...

A route to fast(er) simulations – FATRAS

- Introduction of the fast track simulation for ATLAS: FATRAS
  - Uses extrapolators and “tracking geometry” developed for reconstruction to quickly move particles through ATLAS
- Needs physics modules, tuned to Geant4, to get the description right – We even use hadronic physics modules straight from Geant4
  - Primary speed gain from simplified geometry

Combined with FastCaloSim, gives several 100x CPU improvement over full simulation

Cores of resolutions are in excellent agreement

It takes a lot of time to characterise tails and understand whether they have reached the appropriate level

Better agreement at high $p_T$
A route to fast(er) simulations – FATRAS

- Combined with FastCaloSim, gives several 100x CPU improvement over full simulation
- Cores of resolutions are in excellent agreement
- It takes a lot of time to characterise tails and understand whether they have reached the appropriate level

Better agreement for muons

A route to fast(er) simulations – ISF

- Recently the concept of the Integrated Simulation Framework (ISF) was introduced in ATLAS
- Controls the stack; allows mixed simulation flavors in one event
- Hard questions about calibration and scale factors to be addressed
- This allowed to wrap up all the simulation flavors into a single application, with a relatively transparent switch for the user to go from one setup to another
- This has been key to keeping bugs out of production configurations and ensuring uniform, correct configurations

A route to fast(er) simulations – Recap

- CPU in Run1 was dominated by MC productions
  - Geant4 physics robustness was a major ingredient for the success
  - Many physics analyses in Run1 limited by the available MC statistics
  - Fast simulation was an important booster for simulated samples
  - It will become indispensable with the LHC increase in luminosity and pile up
- LHC experiments exploring a wide range of FastSim approaches
  - Trying constantly to push up the performance limits (better and faster)
- The LHC upgrade and FCC challenges call for major changes in the FastSim frameworks:
  - Understand very high pile-up impact and find solutions
  - Going from “it serves its purpose” to “integration” approach
  - Combining simulation, tracking, digitization to get ready to analyze data samples
    - Save intermediate steps, I/O
- Improving fast simulation performance does not make life easier...
  - Digitization and tracking become bottlenecks and demand their “fast” versions
- Fast and full simulation are NOT mutually exclusive
  - Performance comes from combining their features

Back to Full Simulation – how about it?

- Fast simulations seem to be the way to go to produce big number of events in a complex environment
- But... Still a Geant-like full simulation maintains a relevant role in the simulation strategies of the LHC experiments and studies towards the FCC
- Exploit current trends in CPU developments and dedicated libraries
  - Multi-core CPUs
  - GPUs
- Ongoing projects to boost full simulation performance
  - Geant4 Multi-Threading
  - Geant4Vectorized
- This is not a simple change in GEANT version!
  - It requires totally new approach
Future improvements – Geant4MT

- Since Geant4 Version 10.0 (December 2013) introduced event level parallelism
- Driving forces:
  - maintain and improve physics quality and minimize user code changes
  - keep approach simple
- Master-worker concurrency model using pthreads
  - Each thread processes its own independent event loop
  - Results merged at the end of a run
  - Lock-free (uses thread-local storage)
  - Shares read-only data to reduce memory
    - Geometry
    - Physics lists
    - etc

Future improvements – Geant4MT

- Since Geant4 Version 10.0 (December 2013) introduced event level parallelism
- Driving forces:
  - maintain and improve physics quality and minimize user code changes
  - keep approach simple
  - Physicians ≠ Computing Scientists
- Design/prototyping phases started in 2010-2012
- Important milestone for Geant4, more features released in the next years
  - Version 10.0 (Dec 2013)
    - Implement correct MT behavior
    - Memory reduction from geometry and physics
  - Version 10.1 (Dec 2014)
    - Improve migration synchronization (GPD, RDM, Vis)
    - Obtain further 2x memory reduction
  - Version 10.2 (Dec 2015)
    - Finalize Vis module
    - Brings integration of G4 with MPI and TBB

Future improvements – Geant4MT

Since Geant4 Version 10.0 (December 2013) introduced event level parallelism
- Driving forces:
  - maintain and improve physics quality and minimize user code changes
  - keep approach simple
- Physicians ≠ Computing Scientists
- Design/prototyping phases started in 2010-2012
- Important milestone for Geant4, more features released in the next years
  - Version 10.0 (Dec 2013)
    - Implement correct MT behavior
    - Memory reduction from geometry and physics
  - Version 10.1 (Dec 2014)
    - Improve migration synchronization (GPD, RDM, Vis)
    - Obtain further 2x memory reduction
  - Version 10.2 (Dec 2015)
    - Finalize Vis module
    - Brings integration of G4 with MPI and TBB

Future improvements – Geant4MT

Since Geant4 Version 10.0 (December 2013) introduced event level parallelism
- Driving forces:
  - maintain and improve physics quality and minimize user code changes
  - keep approach simple
Future improvements – Geant4MT

- To give you an idea, memory consumption on Intel Xeon Phi

<table>
<thead>
<tr>
<th>Versions</th>
<th>9.6</th>
<th>10.0</th>
<th>10.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory size (in MBytes)</td>
<td>170 * Nt</td>
<td>28 * Nt + 151</td>
<td>10 * Nt + 162</td>
</tr>
<tr>
<td>Memory size required for 240 threads (in MBytes)</td>
<td>40,816</td>
<td>6,871</td>
<td>2,592</td>
</tr>
</tbody>
</table>

Future improvements – Geant4MT

- CMS has moved to Geant4MT (ATLAS lagging behind)
  - Build in multi-threaded mode, run in either sequential or multi-threaded mode
  - Integrated into TBB-based CMS multi-thread framework
- No CPU efficiency overhead observed with all available 12 cores
- Significant saving of memory space (4x)
  - RSS (Resident Set Size) required for 12 threads: 4 GB
  - RSS required for 12 processes: 16 GB

Future improvements – GeantV

- Started in 2012
  - Prototype a new approach to speedup particle transport simulation
- Most simulation time spent in few percent of volumes
  - Enforce locality and vectorization
  - Add parallelism on top
  - Add new entity to control the workflow
- Exploring many dimensions of performance
  - Locality (cache coherence, data structures)
  - Parallelism (multi/many core, SIMD)
  - Vector dispatching down to algorithms
  - Transparent usage of resources (CPU/GPU)
  - Algorithm template specializations & generality of code
  - Physics & geometry algorithm improvements
Future improvements – GeantV

- Initial idea: to change the basic unit of transport simulation from a single particle to a vector of particles fulfilling geometry locality criteria
  - Most of the simulation time is spent in only 5-10% of all volumes
  - Geometry locality → Material locality → (partially) Physics locality

- From initial prototype to three parallel lines of development:
  - Geometry modeling
    - Re-design geometry models like Geant4 and Root using techniques supporting vectorization and supporting both CPUs and GPUs
  - Physics modeling
    - Re-factor the Geant4 physics engine and algorithms into vectorized kernels usable on both CPUs and GPUs
  - Kernel and Scheduler
    - Optimize the flow of the simulation application
    - Manage concurrency
    - Deploying vectors of tracks matching locality criteria for physics and geometry

Future improvements – GeantV

- VecGeom – optimizing simulation geometry
  - Interfaces to process many particles at once
  - Use of external/internal SIMD vectorization
  - Use of HPC features of C++ ("templates") to further improve performance
  - Library support for GPUs

- Tracks organised in baskets according to geometry or physics locality
- Filters allow users to add fast physics models
- Scheduler looking after track baskets propagation
- One thread dedicated exclusively to I/O operations

Future improvements – GeantV

- VecGeom – optimizing simulation geometry
- Timings for collision detection for various primitives
- Smaller area = better library performance
Future improvements – GeantV

- Re-use of Geant4 physics would be desirable, but performance optimizations will take a long time
  - Vectorization, kernels + algorithms review from these perspectives
- Goal: have a compact and simple form of realistic physics to study the prototype concepts and behavior
- Requirements: mimic the most important effects of the “real physics” to the tracks and to the characteristics of the transport
  - energy deposit, track length, # steps, # secondary tracks, etc
- Implementation:
  - tabulated values of x-sections(+dE/dx) from any Geant4 physics list for all particles and all elements over a flexible energy grid
  - all major processes are involved
  - flexible number of final states for all particles, all active reactions, all elements are also extracted from Geant4 and stored in tables
- Status:
  - a complete particle transport has been implemented based on these tables both behind the prototype and behind Geant4
  - possible to test new concepts, performance relative to Geant4 tracking
  - individual physics processes can be replaced by their optimized version when ready

Fast and Full Simulation – Recap

- A qualitative jump in the way the simulation software uses the hardware is strongly needed
  - Small window of time to R&D new approaches
  - Looking at performance from all angles, including fast simulation
- High performance is within reach
  - Vectorization and locality can give the expected results
  - Extending to GPU
  - Optimizing geometry and physics is a long scale effort
- Including direct support for fast simulation in the transport framework
  - No limits, at the extreme approach GeantV should be usable as a fast simulation framework
  - A lot to learn from the existing “integration” approaches
- “Full” and “fast” have to crossbreed!

Takeaway Messages

1. For the future, solutions will have to be found to speed up the programs while keeping the same precision
2. No silver bullet → a customizable mixture of full and fast simulation is the only approach which still stands a chance to meet requirements set by HL-LHC and FCC
3. A lot of interesting work in front of us!
Thanks for your attention!

Imagination is more important than knowledge

Albert Einstein

FCC-ee

- The FCC-ee is designed to provide $e^+e^-$ collisions in the beam energy range from $\sim$40 to $\sim$200 GeV
- The main centre-of-mass operating points with large physics interest are 91 GeV (Z-pole), 160 GeV (W pair production threshold), 240 GeV (Higgs resonance) and 350 GeV (ttbar threshold)
- The machine would have a circumference of the order of 80 to 100 km in order to limit the synchrotron radiation power
- The machine should accommodate 4 experiments that are operated simultaneously and deliver peak luminosities above $1 \times 10^{34}$ cm$^{-2}$s$^{-1}$ to each experiments at the ttbar threshold and much higher luminosities at lower energies

FCC-ee vs LEP2

<table>
<thead>
<tr>
<th>Parameter</th>
<th>FCC-ee</th>
<th>LEP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy/beam [GeV]</td>
<td>45</td>
<td>120</td>
</tr>
<tr>
<td>Bunches/beam</td>
<td>13000-60000</td>
<td>500-1400</td>
</tr>
<tr>
<td>Beam current [mA]</td>
<td>1450</td>
<td>30</td>
</tr>
<tr>
<td>Luminosity/IP$_{\text{main}}$ [cm$^{-2}$s$^{-1}$]</td>
<td>$21 - 280 \times 10^{34}$</td>
<td>$5 - 11 \times 10^{34}$</td>
</tr>
<tr>
<td>Energy loss/turn [GeV]</td>
<td>0.03</td>
<td>1.67</td>
</tr>
<tr>
<td>Synchrotron Power [MW]</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td>RF Voltage [GV]</td>
<td>0.3-2.5</td>
<td>3.6-5.5</td>
</tr>
</tbody>
</table>

Extra Slides – FCC-ee
Detector Description

- Learning from past experiences
- Using DD4HeP (Detector Description for High Energy Physics):
  - Complete detector description
    - Includes geometry, materials, visualization, readout, alignment, calibration, etc.
  - Support full experiment life cycle
    - Detector concept development, detector optimization, construction, operation
    - Easy transition from one phase to the next
  - Consistent description, single source of information
    - Use in simulation, reconstruction, analysis, etc.
- Fairly scalable and flexible drivers (generic driver palette available)
- Visualization, radii, layer/module composition in compact xml, volume building in C++ driver
- User decides balance between detail and flexibility
- Usually could do a lot just by modifying the xml. For example:
  - Scale detector
  - Create double layers
  - Create "spiral" endcap geometry
  - ...

DD4HeP & Geant4

- DD4hep facilitates in-memory translation of geometry from TGeo (Root) to Geant4
- Plugin Mechanism:
  - Sensitive detectors, segmentations and configurable actions, …
- Configuration mechanism (via python, XML, CINT)
  - Physics lists, regions, limits, fields, …
- All shared with Reconstruction
- Detailed validation underway
- Already simulating realistic physics events to develop/test tracking and particle flow-based reconstruction
Lecture series overview

- techniques for writing algorithms for physics computing in modern C++
- we will strive to produce code which is
  - easy to read
  - efficient
  - modular
- we will introduce some specific methods to
  1) enable natural syntax for mathematical operations
  2) tap into the CPU’s vector processing capabilities
  3) process large data sets
- and explore their inner workings

In this lecture

“process large data sets”

- the last piece we need to write efficient single-node code
- focus on large multi-dimensional arrays – these can represent scalar/vector fields, etc. in physics
- we will note that an algorithm can often be separated into “iteration” and “computation”
- we will use this separation to easily implement thread-level parallelism over our large data set
- finally we will see how we can combine elements from all the lectures to implement a working Maxwell’s equations integrator with a self-explanatory code

Previous Lectures

Monday: Expression Templates
- by means on compile-time expansion of an expression represented by tree built using C++ template system
- generate efficient low-level code from a high level "object-oriented" syntax

Yesterday: Vectorization with Expression Templates
- SIMD instructions, data layouts (AoS, SoA, AoSoA)
- applying ET idiom to achieve vectorization while keeping the code nice and clean
Large multi-dimensional arrays

- very common in physics
- scalar fields, vector fields, matrices (2 counts as “multi-“)

- no standard way to handle large N-D arrays in C++
  - maybe post-C++17 (Ł. Mendakiewicz & H. Sutter)
- many libraries with different approaches
- in this lecture, try to focus more on the concepts and reasons behind the implementation, than the presented solutions
- “why” is more important than “how”

Separating iteration and computation

- lots of algorithms are in the form
  - iterate through an array
  - perform a computation on each cell, usually also using neighbouring cells, or cells from different a array

- iteration – an operation on the “array” (or arrays)
- computation – an operation on the “cell” (or cells)
- we can treat these separately
  - we can keep the iteration code at one place, and changes will get applied to all algorithms which use it

Functional approach

- replace the for loop with a function which does the iteration
- one parameter is a reference to the “computation kernel”
  - the function which performs the computation
- other parameters are references to the data being manipulated
- the iterating function will call the computation kernel on the elements of the array (or more generally a “collection”)
- called “map” or “apply” in functional programming
  - we say we “map the function on the elements of the array”
  - or that we “apply the function to the elements of the array”

* the distinction between “map” and “apply” is not important for this lecture

Functional approach

- there are many ways to implement this
- C++ offers “std::transform” for collections which support STL iterators
  - std::transform(begin, end, result, function);
  - applies function on elements between iterator begin and end, and writes the result to elements starting at iterator result

- this methods only supports linear access pattern
- multi-dimensional arrays often require different treatment
  - we want to address the dimensions separately

```c++
for(int i=0; i<XMAX; ++i) {
  for(int j=0; j<YMAX; ++j) {
    c[i][j] += (a[i+1][j] + a[i][j+1]) / (h*h);
  }
}
```
### Functional approach example

- **1st “layer of indirection”** - an iterating function “apply_each”
- actually two overloaded “apply_each” functions
- the syntax of these functions will be:
  
  ```
  apply_each(func, data)
  ```
  
  applies function `func` taking no additional arguments to every element in `data`

  ```
  apply_each(func, data, lmin, lmax)
  ```
  
  applies `func` with no arguments to elements of `data` which lie between limits `lmin` and `lmax`

- any additional data to be operated on will be passed by lambda capture

### Lambda expressions I

- lambda expressions serve to create a “closure”: an unnamed function object capable of capturing variables in scope
- the closure can be stored in a variable, or passed as a function parameter
- the syntax is:
  
  ```
  [ capture-list ] ( params ) -> ret { body }
  ```

### Lambda expressions II

- `capture-list` is a comma-separated list of zero or more variables to be captured form the enclosing scope
- these variables will be then accessible within the lambda expression’s body
- you can capture the variable by reference or by value
- you can optionally specify a “default capture” to capture all variables referenced in the lambda's body
- the variables are captured at the point of definition of the lambda expression

```c
int a = 5;
// default capture by value
// by reference
// default capture by reference
auto f1 = [=](){ cout << a; };
auto f2 = [&a]() { ++a; cout << a; };
// default by value, but a by ref.
auto f3 = [a](int x) { a++; return x + a; }
```

```c
auto t1 = [](){ cout << x; };
```

```c
auto t2 = []{ cout << x; };
```

```c
auto t3 = []{ cout << x; };
```

```c
auto t4 = []{ cout << x; };
```

```c
int b = f3(5);
cout << b; // prints 13
cout << a; // prints 8
```

### Functional approach implementation

- let’s start with an “apply” function which will operate on 2D arrays
- for this presentation, we will assume we have a 2D array class which provides:
  - efficient chained operator[] to support the data[i][j] syntax
  - methods size_x() and size_y() which return the array size along respective dimensions

```c
class array_2d {
...;
};

array_2d data(x_max, y_max);
for(int i=0; i<data.size_x(); ++i)
for(int j=0; j<data.size_y(); ++j)
data[i][j] = i + y_max + j;
```
### Functional approach example I

- our basic `apply` function takes an argument of type `std::function<R>` (C++11)
  - `std::function<R>` is a wrapper which can store, copy, and invoke any “Callable” target
  - that includes functions, function references and pointers, lambda expression, basically anything which has an operator()
  - `R` is the function’s signature; no need to worry about it - it will be auto-deduced
- the `apply` function will “translate” the function call into a for loop, relaying indexes

```cpp
template<typename R, typename D>
void apply_each(std::function<R> f, D&& d)
{
    for (int i=0; i<d.size.x(); ++i)
        for (int j=0; j<d.size.y(); ++j)
            f(d, i, j);
}
```

```cpp
array_2d data;  // ... load initial values
apply_each(sqr, data);
```

### Functional approach example II

- the computation kernel needs to accept the indexes as its arguments
- we have to pass the data to the `apply` function as we need to know their size
- any additional arguments can be captured by the lambda expression

```cpp
template<typename R, typename D>
void apply_each(std::function<R> f, D&& d)
{
    for (int i=0; i<d.size.x(); ++i)
        for (int j=0; j<d.size.y(); ++j)
            f(d, i, j);
}
```

```cpp
array_2d data, a;  // ... load initial values
apply_each(
    [&a] (array_2d &d, int i, int j) {
        d[i][j] += a[i][j];
    }, data
);
```

### Functional approach example III

- long computation kernels embedded inside the lambda’s body might lead to unreadable code
- you can define your kernel as a function, and use the lambda to relay needed data
- you need to avoid the function call overhead – your kernel has to be inlined

```cpp
template<typename R, typename D>
void apply_each(std::function<R> f, D&& d)
{
    for (int i=0; i<d.size.x(); ++i)
        for (int j=0; j<d.size.y(); ++j)
            f(d, i, j);
}
```

```cpp
template<typename T>
inline double mul(T& d, T& a, int i, int j) {
    d[i][j] *= a[i][j];
}
array_2d data;  // ... load initial values
apply_each([&a] (array_2d d, int i, int j) {
    mul(d, a, i, j);
}, data);
```

### Inlining I

- inlining – replacing the function call with the function’s body
- this eliminates the function call overhead, as the generated code doesn’t actually contain any function call
- you can hint the compiler that you want some function to be inlined by using the `inline` keyword
- but – it’s entirely up to the compiler to decide whether the function will be inlined or not!

```cpp
inline void f (Data& d, int i) {
    d[i] = 2*i;
}
```

```cpp
// the original loop
for (int i=0; i<N; ++i) {
    f(d[i]);
}
```

```cpp
// actual emitted code will be equivalent to
for (int i=0; i<N; ++i) {
    d[i] = 2*i;
}
```

```cpp
// if the inlining did happen
```
Inlining II

- the compiler decides whether to inline a function or not
- there is a trade-off between the cost of the function call overhead, and the increased executable size caused by the inlining – this can lead to misses in the instruction cache
  - for reasonably-sized kernels, the function call overhead is always worse – think large arrays with millions of elements which would lead millions of function calls
- what will and will not be inlined – a rule of thumb
  - functions defined in a different compilation unit – usually a different cpp file cannot be inlined
  - member functions, and functions defined in included header files can be inlined
- check whether the compiler did the inlining
  - use the -Winline switch on gcc and icc for a report

Bounded iteration I

- with a simple modification, we can introduce a version which performs ranged iteration
- we include bounds as arguments to the apply function
  - the kernel function stays the same
  - now the kernel can access neighbouring cells
    - provided we pass correct bounds
  - we will not keep this version – there is something better on the next slide

```cpp
struct Index {
    int x, y; // and a constructor taking {x, y}
};
template< typename R, typename D >
void apply_each(
    std::function<R> f, D&& d,
    Index bmin, Index bmax
) {
    for (int i=bmin.x; i<bmax.x; ++i)
        for (int j=bmin.y; j<bmax.y; ++j)
            f( d, i, j );
}

array_2d data;  // ... load initial values
apply_each( [&](array_2d &d, int i, int j){
    d[i][j] += d[i][j-1] + d[i-1][j];
}, data,
    5, 100, 6, 200 );
```

Bounded iteration II

- build a structure which holds the indexes
- this adds another “layer of indirection”, now on the indexing
  - syntactically, indexing now “goes through” the “index” structure
  - everything is resolved at compile time, so the resulting machine code is the same as the previous example
- our kernels still stay the same
  - but we could rewrite them to take the index objects instead of i_j

```cpp
struct Index {
    int x, y; // and a constructor taking {x, y}
};
template< typename R, typename D >
void apply_each(
    std::function<R> f, D&& d,
    Index bmin, Index bmax
) {
    for (int i=bmin.x; i<bmax.x; ++i)
        for (int j=bmin.y; j<bmax.y; ++j)
            f( d, i, j );
}

Index bmin{5, 6}, bmax{100, 200};
apply_each( [](array_2d &d, int i, int j){
    d[i][j] += d[i][j-1] + d[i-1][j];
}, data, bmin, bmax );
```

Bounded iteration III

- some might not like that the lambda has to take the data as a parameter, and would prefer using only captures
- you can remove the ”D d” parameter form the ranged version, and always pass the indexes even if you want to operate on the whole array
- here we implemented it together with changing the lambda syntax to only take one parameter – the index

```cpp
struct Index {
    int x, y; // and a constructor taking {x, y}
};
template< typename R >
void apply_each(std::function<R> f,
    Index bmin, Index bmax
) {
    for (int i=bmin.x; i<bmax.x; ++i)
        for (int j=bmin.y; j<bmax.y; ++j)
            f( {i, j} );
}

array_2d data;
Index bmin{0, 0};
Index bmax{data.size_x(), data.size_y()};
apply_each( [](array_2d &d, Index idx){
    data[idx.x][idx.y] *= data[idx.x][idx.y];
}, bmin, bmax );
```
Another indexing option

- the kernel functions in our example take the coordinates as additional parameters
- we can instead provide a proxy class "Cell" with special operator[] which would take relative coordinates
- instead of \( c[i][j] = a[i-1][j] + b[i][j-1]; \)
- we would write \( c = a[-1][0] + b[0][-1]; \)
- this could be achieved by using template expressions with a special operator[] definition

Use case: switching coordinate systems

- let's say we have some code written in cartesian coordinates
- we would like to implement spherical coordinates
- that's just a simple change in one function -- the computation kernel
- everything else stays the same
- except of course the initial values, and the tools we use for data post-processing
- no copy-pasting required except for the function definition header

Dimension-agnostic algorithms I

- we can provide "apply" functions for different number of dimensions
- add a DIM template parameter to the declaration
- specialize for DIM=1, 2, 3...
- this code doesn't compile!
- you cannot partially specialize template functions

Dimension-agnostic algorithms II

- you cannot partially specialize functions
- either we will have to go with an overload

Template< typename R, typename D >
void apply_each(
    std::function<R> f,
    array_2d& d)
{ .... }
Dimension-agnostic algorithms III

- or we could create partially specialized classes
  - you would then inherit from this class in classes which want to use the iteration
  - this works well if you don’t mix 2D and 3D arrays in your code

```cpp
template< int DIM > struct grid_iterator {};  
template<> struct grid_iterator<2> {   
static void apply_each(std::function<R> f, D&& d)   
{   
  // body of function
  ....  
};

class computator: public grid_iterator<2> {   
array_2d a, b, c;   
void add_arrays(){   
  apply_each(  
      [&](array_2d &d, Index idx)   
      {   
        d[idx.x][idx.y] = a[idx.x][idx.y] + b[idx.x][idx.y];   
      },
      c   
);  
};
```

Dimension-agnostic algorithms IV

- there are still issues with the presented examples
- but let’s stop here for the sake of simplicity
- you get the general idea on how to separate iteration from computation using functional programming
  - iteration provided by an “apply” function
  - computation done by “computation kernels”

Multidimensional array I

- our examples used a multidimensional array with chained operator[] subscripts
- we could implement this using template recursion, and std::vector as a storage
- instantiating the array <N> will cause “base” array <N-1> to be instantiated as well
- we provide operator[] which returns a reference to the i-th element of the N-dimensional array, which is an N-1-dimensional array

```cpp

```

Multidimensional array II

- we initialize the sub-arrays using a variadic constructor
- we strip the actual dimension by splitting the parameters into one number head, pack the rest N-1 parameters in tail, and forward this to the lower-dimension constructor

```cpp```
Multidimensional array III

- we stop the recursion by specializing for N=1
- 1-dimensional array is just the plain vector
- subsequent applications of operator[] will return references to vectors of N-1 dimensional containers
- until we reach the vector, where the operator[] returns reference to i-th value

```cpp
template<class T, int N>
struct multi_array {
    ....
};

template<typename T>
struct multi_array<T, 1> : public std::vector<T> {
    typedef std::vector<T> base;
    typedef T value_type;
    typedef base v;

    multi_array(int n): v(n) {};
    auto& operator[](int i) {
        return v[i];
    }
};
```

some implementation details omitted

Data-parallel programming

- **task parallelism**
  - different tasks performed simultaneously by independent threads on the same or different datasets
- **data parallelism**
  - the same task is performed simultaneously by many threads each operating on a subset of the dataset

Domain decomposition

- divide the dataset into subsets which are computed independently
  - if dependencies exist, think of the algorithm as several independent steps separated by communication steps which resolve the dependencies
- can be performed at node-level
  - communication by message-passing – MPI
- or thread-level
  - communication by reading and writing to a shared memory
  - static decomposition – each thread is pre-assigned its data subset
  - dynamic decomposition – subsets boundaries might change between iterations or even during one iteration (often difficult or practically impossible at node-level)

Large multi-dimensional arrays II

- number one candidates for domain decomposition
- domain decomposition often maps naturally to the problem
- "embarrassingly parallel" algorithms
  - those with little (or no) need for communication
  - e.g. normalizing an array of vectors – no communication needed
  - or finite-difference integration – communication on border cells only

only "halo cell" contents has to be passed to the neighbor
Thread-level parallelism

- use the layers-of-indirection approach
- do domain decomposition in the hidden implementation
  - static
  - dynamic
- the parallelism is implemented in one place only
  - transparent to the interface user
  - much easier to maintain

Write vectorization-friendly code

- code which is well vectorized is usually easily parallelized
- cache optimization matters
- beware of aliasing (see previous lecture)
  - when passing arrays by pointers, the target memory of the variables might overlap - "alias"
  - unlike the auto-vectorization examples where a potentially problematic loop wouldn't be vectorized, when writing thread-parallel programs you have to watch for these problems yourself

```c
void f (int* offset, double* a, double* b, double* c)
{
    for (int i=0; i<N; ++i)
    {
        c[i] = a[i] + b[i + (*offset)];
    }
}
```

Loop dependencies

- dependencies between reads and writes might lead to data races and silent failures (wrong results)
- more difficult than with vectorization, because we don't know which thread will execute first

<table>
<thead>
<tr>
<th>read-after-write</th>
<th>write-after-write</th>
<th>write-after-read</th>
</tr>
</thead>
</table>
| for(i=1; i<N; ++i) {
  a[i] = a[i-1] + b[i];
}
| for(i=0; i<N; ++i) {
  a[i] = b[i] + c[i];
  a[i+2] = 2 * i;
}
| for(i=1; i<N; ++i) {
  a[i-1] = a[i] + c[i];
}

writing to a variable, then reading its value
writing to the same variable in more than one iteration
possible data race – we don’t know which thread will execute first

- cannot be done in parallel
- proceed with extreme caution
- proceed with caution

OpenMP basics

- OpenMP is an API for multi-threaded programming
- it uses a declarative approach to threading
  - you specify regions to be parallelized by using #pragma directives
- often used in data-parallel scenarios
- number of spawned threads can be controlled either at runtime, or by the environment variable OMP_NUM_THREADS
  - it usually defaults to the number of cores in your machine, so most of the time you don't have to set anything
OpenMP basics

// declaring a parallel section of code
#pragma omp parallel
{
  // this code will be executed by all threads
#pragma omp for
for(int i=0; i<N; ++i) {
  // work will be divided between the threads
}

// a shorthand for loop parallelization
#pragma omp parallel for
for(int i=0; i<N; ++i) {...}

// also works on random-access iterators
std::vector<float> v(N)
#pragma omp parallel for
for(auto it = begin(v); it != end(v); it++) {...}

● this is already enough to parallelize simple loops without data dependencies
● it will be sufficient for us today
● very important concepts not explained in this lecture:
  - data scope (sharing)
  - thread scheduling

Parallelizing the “apply” function

● we just include the OpenMP parallel for directive to the outermost loop
● we do this in each version of the apply_each function
● if we avoid calling apply_each in cases which might lead to a data race, we’re fine

template<
  typename Func,
  typename Data>
void apply_each<2>(
  Func& f, Data& d)
{
  #pragma omp parallel for
  for(int i=0; i<d.size.x(); ++i)
  for(int j=0; j<d.size.y(); ++j)
  f ( d, Index{i, j} );
}

Putting everything together

● we’ll create a solver for Maxwell’s equations
● for different precisions
● with a choice of 2D or 3D
● vectorized
● OpenMP threading
Finite-Difference Time-Domain method

- a method for integrating Maxwell’s equations
- in each time step, we calculate for all staggered grid points:

\[
\vec{B}_{t+\Delta t/2} = \vec{B}_{t-\Delta t/2} + c \Delta t \vec{rot} \vec{E}_t \]

\[
\vec{E}_{t+\Delta t} = 4\pi c \Delta t \vec{J}_{t+\Delta t/2} + c \Delta t \vec{rot} \vec{B}_{t+\Delta t/2}
\]

\(\vec{rot}\) denotes the discrete curl operator

\(\vec{rot}\) looks familiar?
- it looks almost exactly like microscopic Maxwell equations “multiplied by \(\Delta t\)”, and with the curl operator replaced by the discrete curl

\[
\vec{B}_{t} + \frac{\Delta t}{2} = \vec{B}_{t} - \frac{\Delta t}{2} + c \Delta t \vec{rot} \vec{E}_t
\]

\[
\vec{E}_{t} = 4\pi c \Delta t \vec{J}_t + \Delta t \vec{rot} \vec{B}_t
\]
Maxwell solver

- this is the frontend of the Maxwell solver
- it implements the FDTD method on a staggered "Yee's grid" – the equations

\[
\begin{align*}
B_{t+\Delta t/2} &= B_{t-\Delta t/2} * c \Delta t * rot \vec{E}_t \\
E_{t+\Delta t} &= 4 \pi c \Delta t * J_{t+\Delta t/2} * c \Delta t * rot \vec{B}_{t+\Delta t/2}
\end{align*}
\]

Take Away Messages

- "iteration" and "computation" can be treated separately
- this separation can be expressed in functional programming paradigm as "mapping" or "applying" a computation kernel function to the data
- lambda expressions can be thought of as anonymous function objects which help simplify writing functional-style code – we can define them at the place where we use them
  - they can also capture variables from the enclosing scope without the need for cumbersome specialized function declarations
- when we separate the iteration, we can easily parallelize all loops which use these iterating function in one place using threads (OpenMP is a good choice for this use case)
  - but we must be carefully avoid data races

Conclusion of the Lecture Series

- C++ templates provide a powerful system which can manipulate the syntax creating compile-time abstractions
  - meta-program – a program that writes programs
- combining the ET idiom with other TMP techniques can create interfaces that allow us to write natural front-end code which resolves to high-performing low-level code
- the abstraction can hide access to lower-level hardware features such as vectorization and threading
- resulting programs are portable – they use standard C++
- they are also easier to maintain – one group can focus solely on the backend which operates close to the hardware, the other on higher-level numerical algorithms and physics
Conclusion of the Lecture Series

- the back-end code is sophisticated
  ```cpp
template <typename T, int N> class alignas(32) container {
    template<typename V, int... K> V ret_impl(int i, std::index_sequence<K...> { }
    return V( { std::get<K>(data)[i] ... });
  }
  Vector<T&> operator[](unsigned int i) {
    return ret_impl(i, std::make_integer_sequence<int, N>());
  }
}
```

- the front-end code is beautiful
  ```cpp
  apply_each<DIM>({ [i] (Index idx) {
    B[idx] += c*dt * curl<BCK>(E, idx);
  }});
  container<vector<float, 3>>
  F = q * (E + cross(v, B));
  ```

Thank you for your attention!

You can find full examples and links to additional sources at: vysko.cz/icsc2016
Shared memory and Message passing revisited in the many-core era

Aram Santogidis
CERN

Inverted CERN School of Computing, 29 February – 2 March 2016

The pioneers of concurrent programming

Edsger Dijkstra
• Mutual exclusion
• Cooperating Sequential Processes
• Semaphores

Per Brinch Hansen
• Concurrent Pascal
• Shared Classes
• The Solo OS
• Distributed Processes

C.A.R Hoare
• Communicating Sequential Processes (CSP)
• Monitors

Communication is important

Time

Process 1
Process 2
Process 3

Communication/ Synchronization

Shared memory

VS

Message passing

Agenda of the talk

• Concurrency and communication
• Two basic examples of the two models
• Conventional wisdom for the two models
• Cache coherence and manycore processors
• Emerging paradigm shift in OS architectures
• The future perspective
The Shared Memory model

- Threads communicate implicitly with each other via shared data structures
- Synchronization primitives (locks, semaphores, etc.)

The message passing model

- Threads communicate explicitly with each other by exchanging messages
- Is the more fundamental class from the two
- Synchronous or asynchronous communication

Let's see an example for each model

1. Image processing (shared memory)
2. Simple GUI (message passing)
A shared memory-based example: Convert from colour to grayscale

We parallelize the computation by assigning tiles (pieces) of the image to threads which execute the conversion in parallel.

A message passing-based example

- A GUI with 3 widgets
  - Text Area
  - Up scroll button
  - Down scroll button
- Must be interactive (Immediate feedback)

GUI example implementation: Message passing solution

Conventional wisdom about the characteristics of the two models

1. Performance
2. Programmability
Performance comparison

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware support</strong></td>
<td><strong>Extensive</strong> (All popular architectures)</td>
<td><strong>Limited</strong> (Only special purpose architectures)</td>
</tr>
<tr>
<td><strong>Data transfer</strong></td>
<td><strong>Low</strong> (Cache block management in HW)</td>
<td><strong>High</strong> (Data replication)</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td><strong>Sometimes high</strong> (Critical section contention, NUMA effects)</td>
<td><strong>Low</strong> (Local private memory access)</td>
</tr>
<tr>
<td><strong>Access/Sync</strong></td>
<td><strong>Sometimes high</strong> (Critical section contention, NUMA effects)</td>
<td><strong>Low</strong> (Local private memory access)</td>
</tr>
</tbody>
</table>

Programmability comparison

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication</strong></td>
<td><strong>Implicit</strong></td>
<td><strong>Explicit</strong></td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td><strong>Explicit</strong> (locks etc.)</td>
<td><strong>Implicit</strong> (side-effect)</td>
</tr>
<tr>
<td><strong>Interface (API)</strong></td>
<td><strong>Read/write shared data structures, mutex primitives</strong></td>
<td><strong>Send/Receive messages, Multicast</strong></td>
</tr>
<tr>
<td><strong>Hazards</strong></td>
<td><strong>Race conditions, Deadlocks, Starvation</strong></td>
<td><strong>Deadlocks, Starvation</strong></td>
</tr>
</tbody>
</table>

Towards the manycore architectures

The manycore era

- Power limits the frequency increase of the processor.
- Moore’s law: The transistors keep doubling every two years
- Replication: Increasing number of cores


On the duality of operating systems structures

- Operating Systems are generally classified as:
  - Message passing oriented
  - Procedure-oriented (shared memory)

- Each system from one category has the other category.

- Neither model is inherently better than the other (depends on the machine architecture).


Non Uniform Memory Access (NUMA)

Cache coherence

Cache coherence
**Cache coherence**

When updating shared state, which approach is more expensive (in terms of latency), Shared memory or Message passing?

**A key question**

An experiment of shared memory vs message passing performance

Updating shared state of size [1,8] cachelines, relying on cache coherent shared memory on 4x4 AMD system
Messages scale better than shared memory

Message passing scales better than shared memory when increasing the core count and the size of the shared state.


...some other hints that may lead to further fragmentation of coherency domains


Increasing Heterogeneity of computing platforms

- Message passing: Fundamental for communication in heterogeneous environment
- Shared memory: Hard to implement in a heterogeneous environment

Message passing OS vs Shared memory OS

Barrelfish OS (Message passing)

Linux OS (Shared memory)

Architecture dependant code

Emerging concurrency paradigms

New high level concurrency paradigms are being developed, based on shared memory and/or message passing constructs.

- Asynchronous tasks (Futures/Promises)
- Partitioned Global Address Space (PGAS) languages/libraries
- Actor Model
- Functional Concurrency

The future perspective

- Communication is the key
  - For energy efficiency
  - For runtime performance
  - To manage software complexity
  - To manage hardware heterogeneity
- Innovation in the hardware sector pressures to systems software engineers to develop appropriate support
  - At the operating system level
  - Concurrent programming frameworks level
  - Communication-oriented tools and techniques to design, implement, analyse concurrent programs
References


- A Primer on Memory Consistency and Cache Coherence Daniel J. Sorin, Mark D. Hill, and David A. Wood


Thank you for your attention

Many thanks to my supporters and mentors for this presentation:

Sebastian Lopienski, Sebastian.Lopienski@cern.ch, CERN
Andreas Joachim Peters, Andreas.Joachim.Peters@cern.ch, CERN
Andreas Hirstius, andreas.hirstius@intel.com, Intel GmbH
Spyros Lalis, lalis@inf.uth.gr, University of Thessaly

This work is support by the Marie Curie Early European Industrial Doctorates Fellowship of the European Community’s Seventh Framework Programme under contract number (PITN-GA-2012-316596-ICE-DIP).
Volatile Environments with Virtualization Technologies

Anastasios Andronidis
<a.andronidis@imperial.ac.uk>
Imperial College London

Outline

- Problems during developing, testing and setup
  - Vagrant + Demos!
  - Docker + Demos too!

Motivation (aka problems)

- Three kinds of problems (IMHO):
  - Project switching
  - Project testing
  - Project transferring

This is going to be a technical session with a lot of examples!
Project switching

- Dependencies
- Requirements
- Interlinking components

Project testing

- Various configurations
- Different components
- Altered topologies

Local VM lifecycle

- Download a VM image
- Boot the VM
- Configure resources: RAM, CPUs, network connections, and shared folders
- Install additional software
- User’s configuration
### History of Vagrant

- Initial release: March 8, 2010, 5 years ago
- Mitchell Hashimoto and John Bender
- HashiCorp
- [https://www.vagrantup.com](https://www.vagrantup.com)

### What Is Vagrant?

- Simplifies the workflow necessary to run VMs locally
  - Offers a very simple CLI
  - Supports: VirtualBox, VMWare, and Hyper-V…
  - Supports: Ansible, Chef, Puppet, and Salt
  - Distribute and share virtual environments

### Demo 1

- Basic commands
- `vagrant init` & `vagrant up`
- Shared folders
- `nginx server`
- `vagrant share`

### Demo 2

- Various web projects
- Port forwarding
Demo 3

- Multiple VMs
- Private networking
- Redis Server-Client Benchmark!

Further Reading

- Provisioners!
- https://www.vagrantup.com/docs/
- https://atlas.hashicorp.com
- https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins

History of Docker

- Initial release: 13 March 2013, 2 years ago
- Solomon Hykes
- dotCloud, now: Docker, Inc.
- www.docker.com
Extra features from VMs

- Speed and small memory footprint
- Soft Memory

Demo 1

- LXC example
- chroot on steroids

Demo 2

- Redis
- docker tunnels

Further Reading

- https://www.docker.com
- https://docs.docker.com
- Vagrant Docker Provisioner
Recap

- Use the virtualisation!
- Helps you to keep track of development
- Helps you to share your work
- Keeps complexity and overhead low
- Enforces good practices
- Question?
For more information on the CERN School of Computing:

http://cern.ch/csc