Third I.N.F.N. International School on

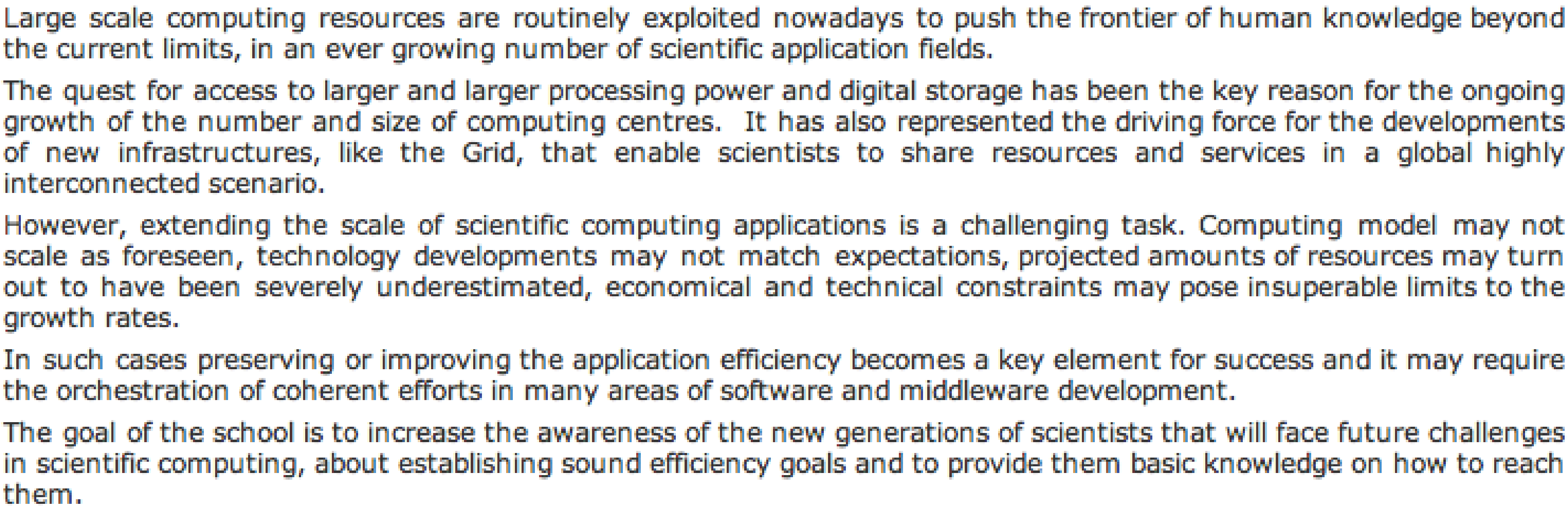
"Architectures, tools and methodologies for developing efficient large scale scientific computing applications"

Ce.U.B. Bertinoro (FC) 23 - 29 October 2011

Introduction

Concepts of Performance and Efficiency

School Goals



Large scale computing resources are routinely exploited nowadays to push the frontier of human knowledge beyond the current limits, in an ever growing number of scientific application fields.

The quest for access to larger and larger processing power and digital storage has been the key reason for the ongoing growth of the number and size of computing centres. It has also represented the driving force for the developments of new infrastructures, like the Grid, that enable scientists to share resources and services in a global highly interconnected scenario.

However, extending the scale of scientific computing applications is a challenging task. Computing model may not scale as foreseen, technology developments may not match expectations, projected amounts of resources may turn out to have been severely underestimated, economical and technical constraints may pose insuperable limits to the growth rates.

In such cases preserving or improving the application efficiency becomes a key element for success and it may require the orchestration of coherent efforts in many areas of software and middleware development.

The goal of the school is to increase the awareness of the new generations of scientists that will face future challenges in scientific computing, about establishing sound efficiency goals and to provide them basic knowledge on how to reach them.

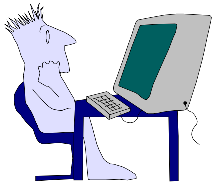
In this presentation I am going to give a basic introduction to the topic of performance/efficiency for scientific applications and give you an overview of the topics you will see over the course of the week.

Performance

- What do we mean by the software performance and efficiency of "**large scale scientific applications**"? Different points of view:
- An individual scientific user may be interested in:
  - Time to completion (from "start" to results)
- Computing center admins, experiments/projects, grid providers, etc. may be more interested in:
  - Total throughput (for all users of the system)
  - Efficiency in the use of the resources (Is it all used? Or sitting idle?)
  - Total resource utilization by a user, experiment, etc.
  - The scalability of the throughput as new resources are added
- A funding agency (the "money man") may be interested in:
  - The total cost of the system (or cost/year)
  - The predictability of the cost evolution of the system
  - Efficiency in the use of the resources (Is it all used? Or sitting idle?)

First lets explore the basic model of the last 10-15 years....

Single user



A single user working on a desktop workstation had until a few years ago a couple of simple options to improve their "time to completion":

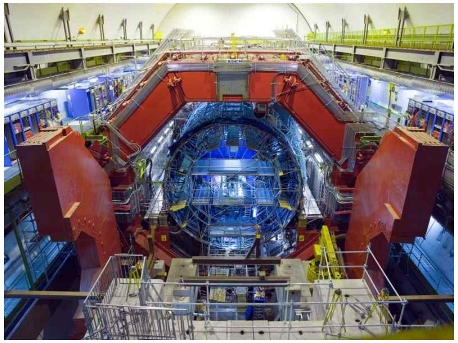
Buy a new computer (CPU, memory, disk, etc.), in general each new generation of machines usually brought a performance gain, even when simply rerunning existing binary programs.

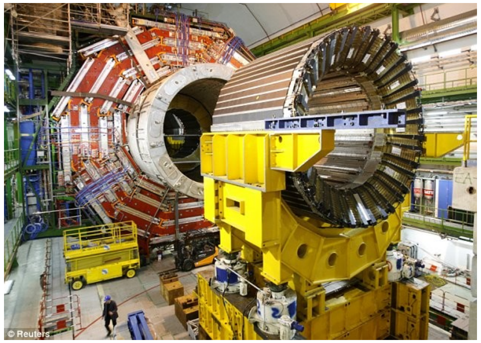
Make modifications to the program to make it run faster.

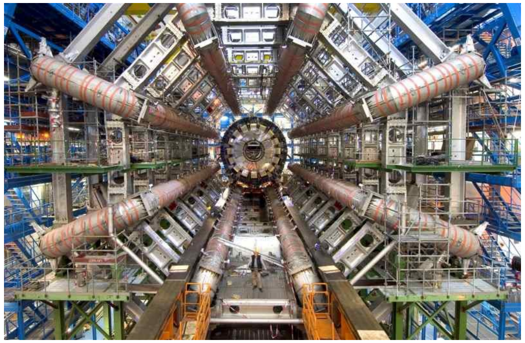
The machine perhaps sat idle when the user was away (on vacation, etc.), so the throughput wasn't being maximized, but at this scale it isn't critical.

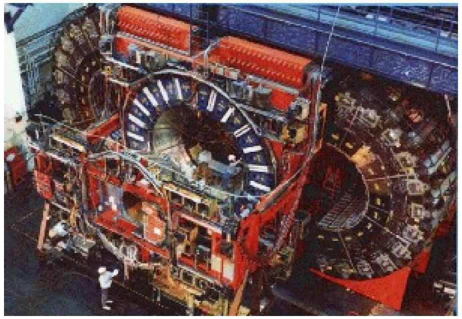
High Energy Physics (HEP)

HEP computing is ***embarrassingly (data) parallel***: N independent instances of an application can be started as simple unix processes, each one processing an independent sets of events. No real communication is needed between the separate processes.

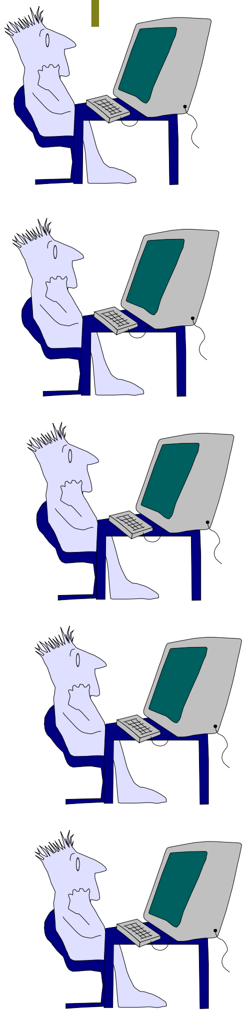








Clusters (and Grids)

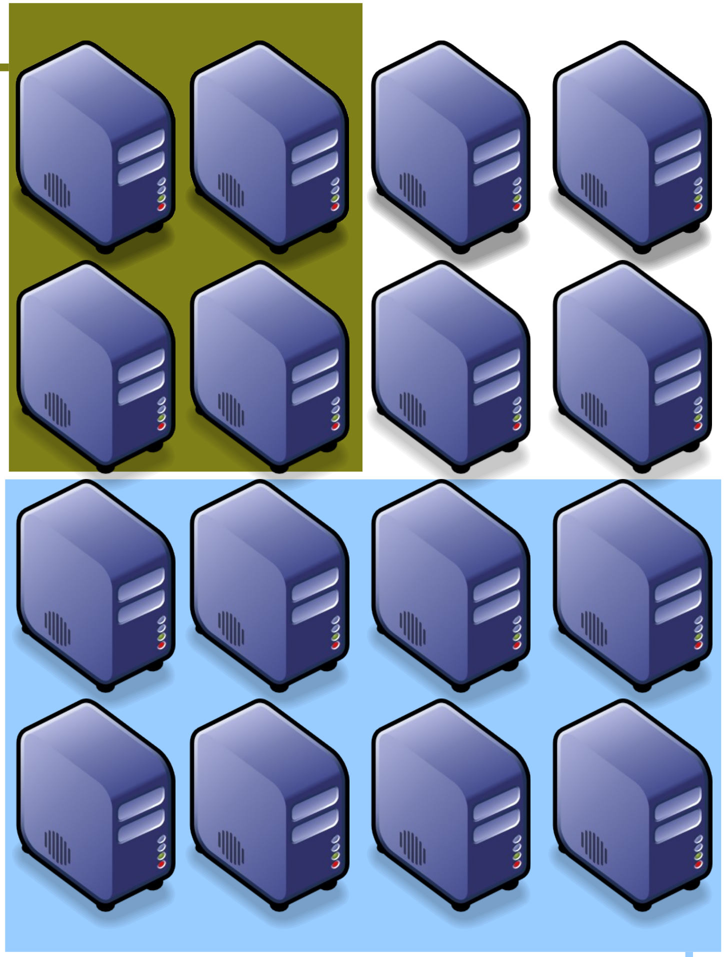


Any individual user can reduce time to completion by using a larger set of machines, if the application is *parallelizable*

The cluster administrator can improve throughput by adding new or additional machines or improve time to completion for individual users by giving them access to more of the common resources

The throughput of the system is however *not necessarily* improved because users parallelize their applications

Both the throughput and the time to completion can be improved if changes can be made to make the application run faster

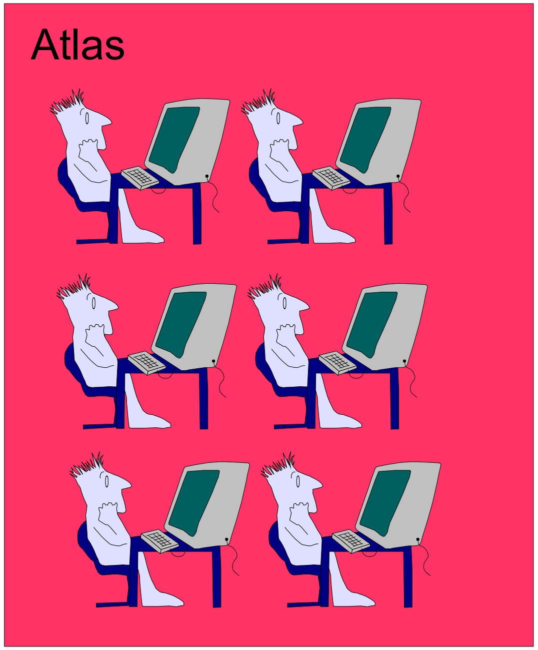


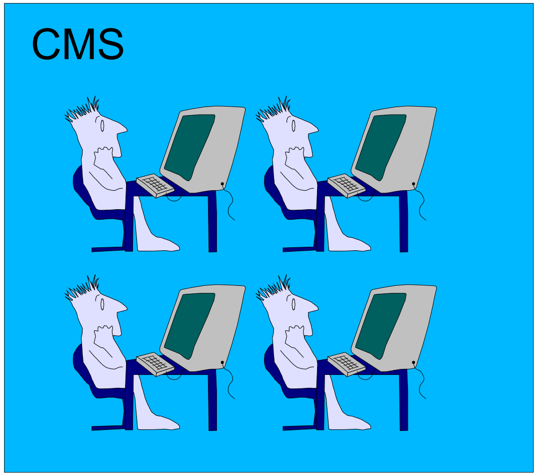
Trivial Example

- Suppose that a particular Geant4 simulation takes 1 minute per event, plus a (one-time) job startup time of 5 minutes
- A single user wants to simulate 10000 total events
- In a single job, the “time to completion” is 10005 minutes and the total resource utilization (counting towards total throughput) is 10005 CPU-minutes
- If the job is run as 10000 separate jobs, each doing 1 event, the user could (in principle) have a “time to completion” of 6 minutes, but the total resource utilization is now 60000 CPU-minutes

- Similar considerations apply to the “serial” and “parallelizable” portions of a particular application or workflow

Grids (and Clouds)

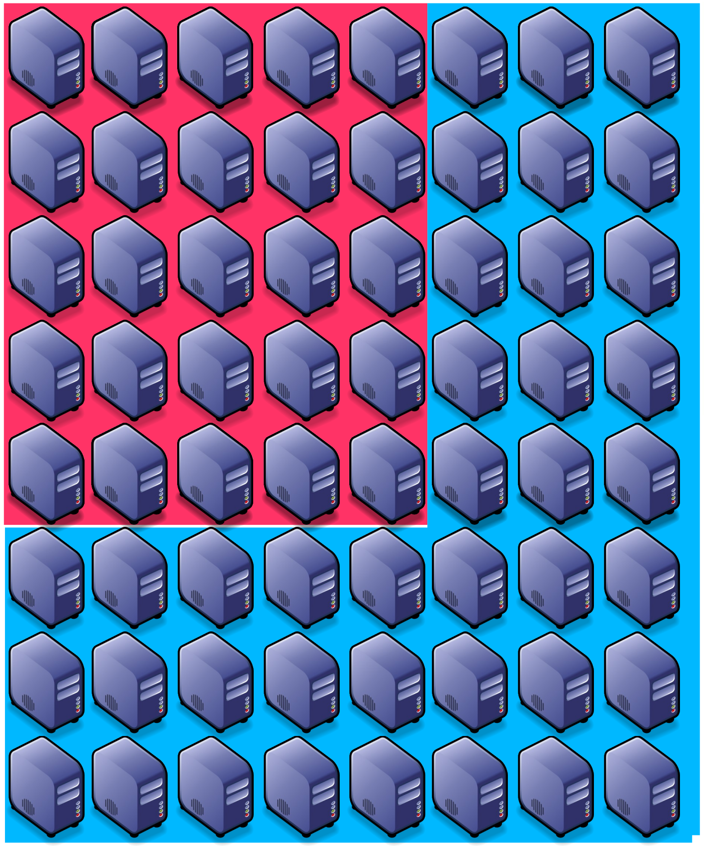




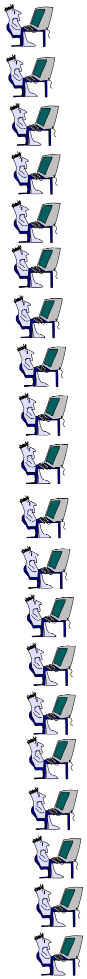
Pooling of ever larger sets of resources provides individuals with even better opportunities to reduce their total “time to completion”.

However the global accounting for what one has used is still there, to satisfy the needs of admins, experiments/projects and the money man.

Thus there is still a need to focus on improvements to the actual application.



Scaling



When using large sets of resources one can also run into scaling limits where adding more or newer resources doesn't result in more throughput.

An example of an “external” constraint is I/O: access to disk or other storage, databases, etc. If this is insufficient, the use of CPU resources (for example) may be very inefficient. Note that such problems can be due to both inadequate hardware as well as poorly behaving Applications.

Scaling issues can also come out from difficulties in making a given application sufficiently parallelizable to exploit the resources available.

Lessons – circa 2005

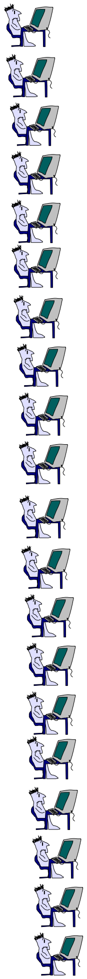
- Much of the performance (both in “time to completion” and “throughput”) boiled down to the art of improvements in the single application performance.
- “Time to completion” could be improved by trivial parallelism and the use of ever larger pools of shared resources (i.e. large clusters and the grid).
- As the HEP problem, at least, is “embarrassingly parallel” and no particular effort needs to be spent on achieving parallelism. In fact “parallelism” wasn't even a term one often needed to use.
- Careful attention to I/O with storage systems is needed to insure scaling and single point scaling bottlenecks (e.g. databases, catalogs) should be avoided or carefully managed.
- These things are still true today, but from ~2005 an additional fly in the ointment appeared....

The fly in the ointment – after 2005

- Around 2005 there was a significant change in the evolution of commodity proceessors, as will be described in detail in the talks later this morning.
- Prior to that we could expect that each subsequent generation of processor would be faster than the previous generation, primarily due to clock frequency scaling.
- However starting around 2005, technical limits (in particular power) led to a plateau in the increase in clock frequencies.
- Since Moore's Law continues unabated, the CPU producers have turned instead to exploiting the increasing number of transistors by providing multiple “cores” within a single CPU
- Instead of getting a processor that is twice is fast at the same price, for example, one effectively gets two processor units.

Hardware evolution

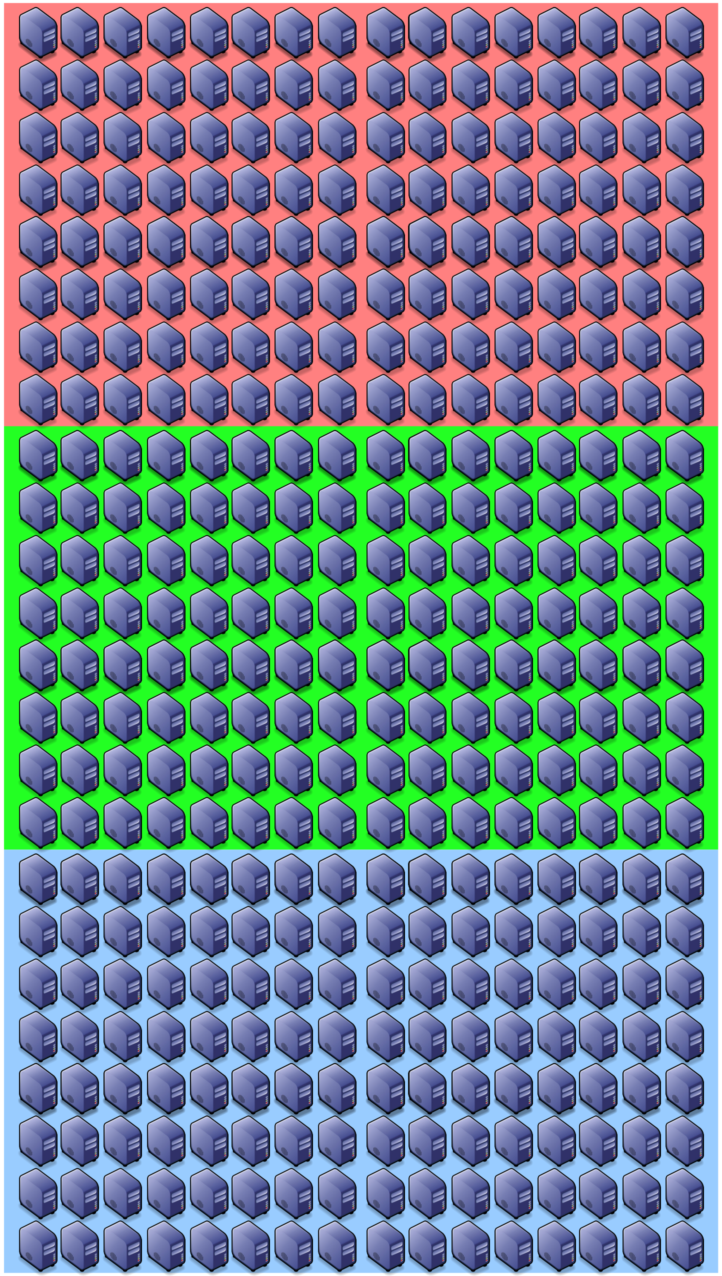
(Through ~2005)



Machines purchased 3 years ago
Each box has 1GB
and perf = P

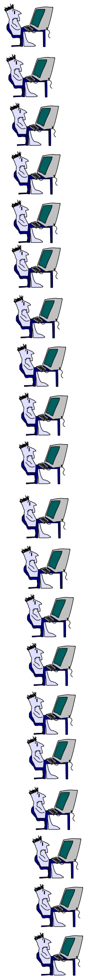
Machines purchased 1.5 years ago
Each box has 1GB
and perf = 2.0 P

Machines purchased this year
Each box has 1GB
and perf = 4.0 P



Hardware evolution

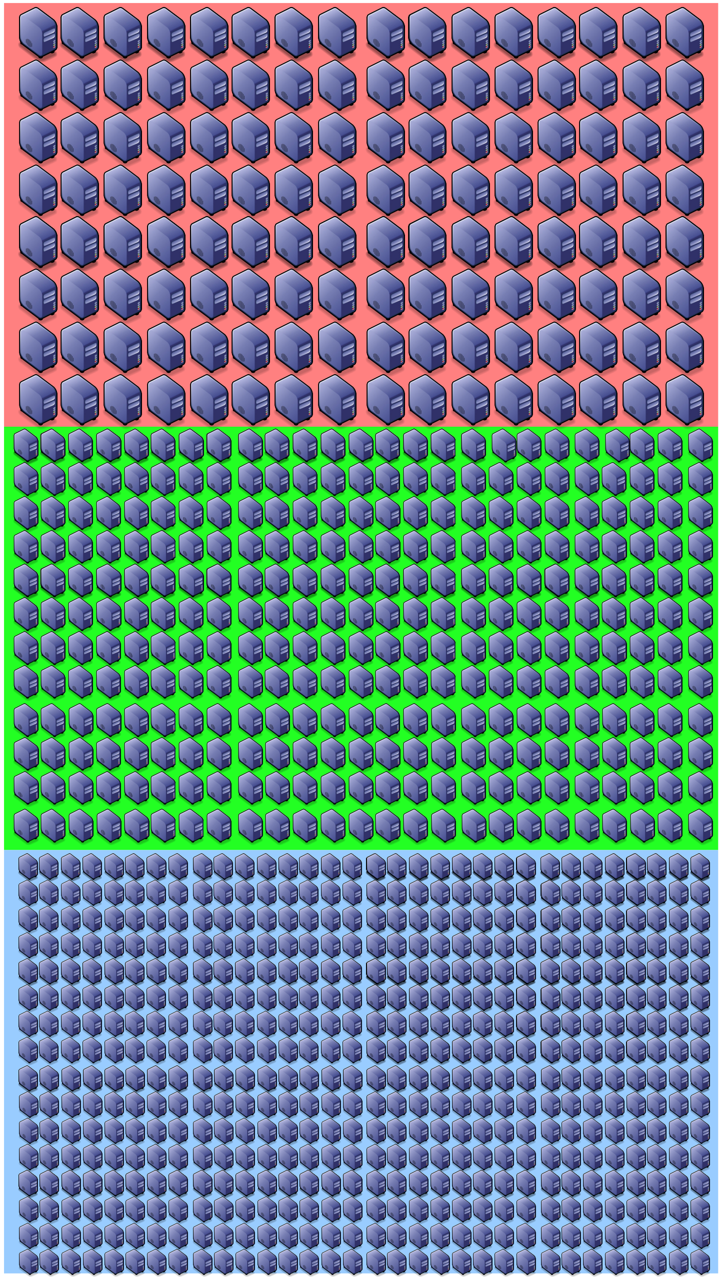
(Treating cores as independent processors)



Machines purchased 3 years ago
Each box has 1GB
and each core perf = P

Machines purchased 1.5 years ago (dual cores)
Each box has 2GB
and each core perf ~ P

Machines purchased this year (quad cores)
Each box has 4GB
and each core perf ~ P



Expectations (with multi/manycore)

- While treating multicore CPU's as if they are simply N independent processors has worked for small numbers of cores, it is expected that this will not scale forever.
- Memory needs are not amortized with each generation of purchases, but instead increase as ~ Moore's Law
- A number of scaling issues arise from an exponentially increasing number of active (and independent) proceses in the systems: I/O, access to services (databases), job and file management
- Performance within a single multicore CPU may not scale perfectly due to memory hierarchy
- Conclusion: the trivial “event” data parallelism is not enough, we need to find other types of parallelism in our applications in order to exploit multi/manycore CPU's.

Lessons – after 2005

- Much of the performance (both in “time to completion” and “throughput”) boiled down to the art of improvements in the single application performance.
- “Time to completion” could be improved by trivial parallelism and the use of ever larger pools of shared resources (i.e. large clusters and the grid).
- Even though HEP problem, at least, is “embarrassingly parallel” on events, that is probably insufficient to fully exploit multi/manycore CPU's. *Additional parallelism must be found and exploited to avoid scaling issues and reduced efficiency.*
- Careful attention to I/O with storage systems is needed to insure scaling and single point scaling bottlenecks (e.g. databases, catalogs) should be avoided or carefully managed.

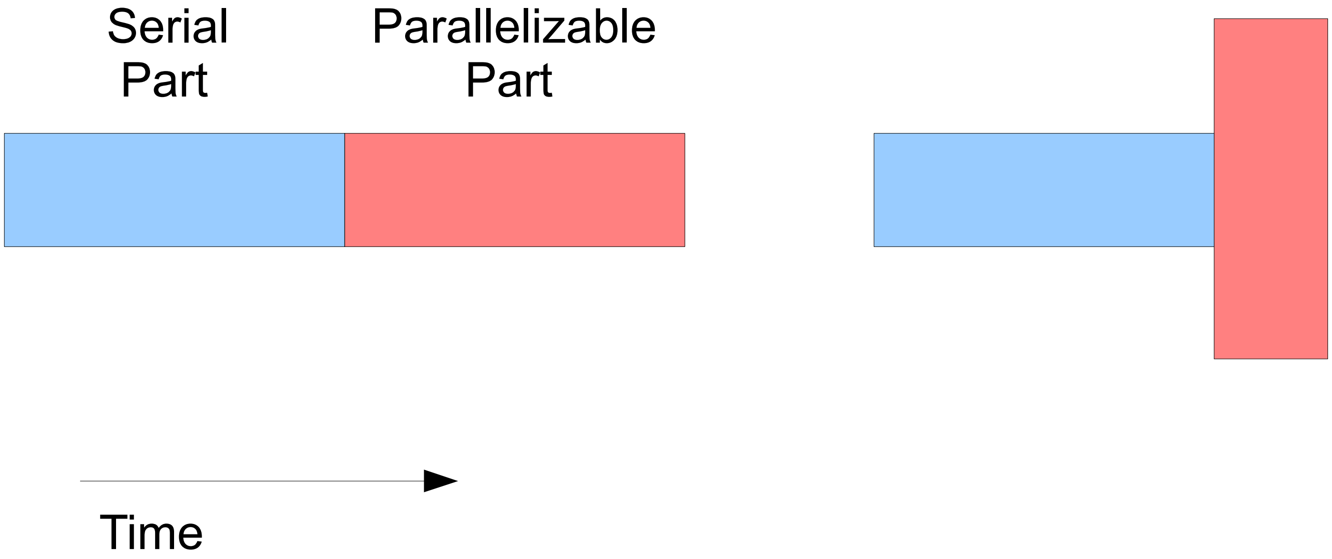
The Art of Application Performance

- What kinds of things are relevant to improve the performance of a single application?

- A number of ingredients affect the realizable performance:
- Hardware – CPU, Memory subystem, I/O
- Software – Application code, compiler and operating system
- Algorithms – Knuth/CS, Scientific, Parallelisation

Amdahl's Law

- The improvement in the total time due to improvements to one part is limited by the amount that part is used
- A similar restatement is: when parallelizing one part of an application, you can never do better than the remaining serial part.



High Level Algorithm choices

- Often the things which most directly determine the performance are simple choices made as to what the program is actually doing, i.e. the high level algorithms.
- For example, if you are running a simulation: are you simulating only the relevant things? Is the level of detail greater than what is needed or needed for all parts of the simulation?
- Such high level considerations can often result in large factors in the time to completion (or resources needed) for any given task.
- It is important to ask such questions near the beginning, and confirm via profiling that the main performance drivers have been identified, before rolling up one's sleeves and diving into the more technical performance tuning.

Profiling tools

- You probably want to make sure that the time you dedicate to working on software performance and efficiency will help
- To do this you should be making decisions based on performance profiles for your application(s)
- In this school you will use several example profiling tools:
  - Igprof – simply statistical profiler and memory profiler
  - Valgrind – general memory debugger/profile
  - A variety of Linux system tools
  - Perfmon – CPU performance counters
- In your experiment, institute or project you may use others
- The important thing is to use profilers as a guide to where the problems/opportunities are, don't guess!

Hardware – CPU architecture/Memory

- We of course compute on actual physical “computers” and thus their evolving capabilities are the most basic component of the achievable performance of some application
- Moore's Law – number of transitors available per unit cost doubles every 1.5 years
- A number of factors conspired to make it possible for many years (1990's through ~2005) to take applications (often without recompiling!) and run them on the next generation of hardware and see a performance gain out-of-the-box.
- This easy ride is over, however. Without changes many applications will not run faster on newer hardware (and many at times actually run slower).
- In addition to “multicore”, exploiting fully CPU's is a challenge.
- Understanding the basics of how to best exploit the hardware going forward will be the topic of several lectures this week.

Operating Systems

- For the most part Linux is the primary operating system considered in these presentations
- The capabilities of the operating system and its runtime environment have can have an important impact on performance, for example:
  - Virtual Memory subsystem – using or abusing this can affect performance
  - Shared libraries and/or other details of "code packaging" can have an impact on performance
  - Math libraries – by default you may be taking the math library (libm) from the system, unless you've made a conscious decision to do otherwise

Compilers

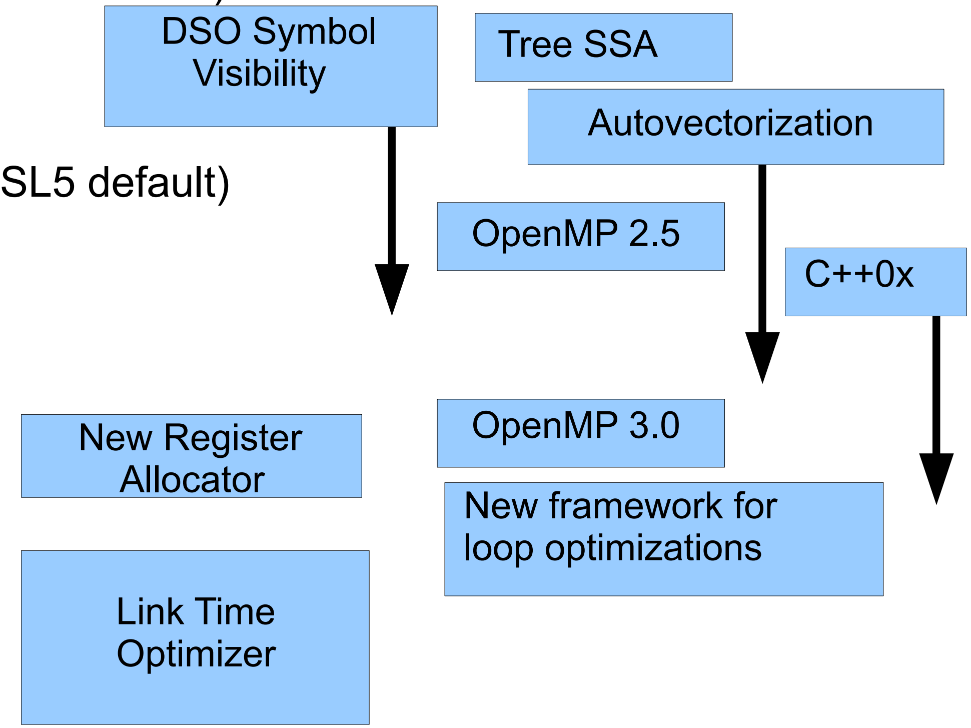
- The compiler is clearly one of the most important tools for achieving optimum code performance
- Unless we want to hand-code everything in assembly, we rely on it to take our code, written in a high-level language like C++, and produce the fastest code possible.
- Usually we also want it to accomplish that in the shortest time possible, to use as little memory as possible doing it, to produce the smallest code possible, etc.
- Note however that compilers cannot always find and optimize things that a human might immediately recognize. In particular compilers are (usually) conservative and will choose code that is guaranteed to be correct over code that might be wrong in some cases.

GNU compiler collection (gcc)

- The workhorse open source compiler, used by most of us, most of the time, these days...
- Front ends for C, C++, Fortran (Ada, Objective-C(++), Java and others)
- Back ends for x86, x86_64 (Alpha, ARM, ia-64, PowerPC, Sparc and many others)
- Most software today is easily configured to build with gcc
- Although most of work on linux/x86(_64) today, or at most MacOSX/x86_64, at least in non-DAQ environments, the wide availability of gcc for different OS/CPU combinations once eased porting C/C++ from one to another.

GCC version timeline/features

- GCC 3.4.0 - 18 Apr, 2004
  - GCC 3.4.6 - 06 Mar, 2006 (~RHEL4/SL4 default)
- GCC 4.0.0 - 20 Apr, 2005
- GCC 4.1.0 - 28 Feb, 2006
  - GCC 4.1.2 - 13 Feb, 2007 (~RHEL5/SL5 default)
- GCC 4.3.0 - 05 Mar, 2008
  - GCC 4.3.2 - 27 Aug, 2008
  - GCC 4.3.4 - 04 Aug, 2009
- GCC 4.4.0 - 21 Apr, 2009
  - GCC 4.4.1 - 22 Jul, 2009
- GCC 4.5.0 - 14 Apr, 2010
  - GCC 4.5.1 31 Jul, 2010
- GCC 4.6.0 – 25 Mar, 2011
  - GCC 4.6.1 – 27 Jun, 2011



DSO Symbol Visibility

Tree SSA

Autovectorization

OpenMP 2.5

C++0x

OpenMP 3.0

New Register Allocator

New framework for loop optimizations

Link Time Optimizer

Various banner improvements in recent gcc4.x compiler versions.
(See Release notes for full list, though!)

LLVM/Clang Compiler

- Recent open source compiler project, aiming to build a set of modular compiler components
- The initial versions replace the optimizer and code generation of gcc, but still reuse the gcc front-end/parser (compatible compiler options!)
- A separate project (Clang) aims to replace gcc front-end for C/C++/Objective-C. As of version 2.8, this claims to be “feature complete” relative to ISO C++ 1998 and 2003.
- Targets both static compilation as well as just-in-time (JIT) compilation
- Sponsorship (in particular) by Apple

Intel Compiler (icc)

- Intel's showcase Fortran/C/C++ compiler(s)
- Arguably focused on demonstrating the best possible performance to be obtained from their processors
- Independent compiler (language syntax, code quality)
- Generates code for all of the Intel processors, plus in principle other x86/x86_64 compatible, i.e. AMD, processors
- Available for Linux/MacOSX/Windows, proprietary license
- The default behaviour for floating point may or may not be what is desired (see presentations about floating point this week)

C++ programming

- In the lectures at this school you will see primarily C++ as it is the most common programming language used for the performance-intensive scientific applications, especially in HEP. (Perhaps a bit of C will make an appearance, too.)
- C++ can be an extraordinarily powerful language, but it is also a very complex language.
- Single lines of seemingly innocuous code can hide major performance problems when compiled into machine code and executed.
- Understanding the “gotchas” of C++ programming is an important ingredient to writing performant applications (if you are using C++, of course)
- A fun google game: try searching for “I hate XXX” for various values of XXX...

Benchmarks

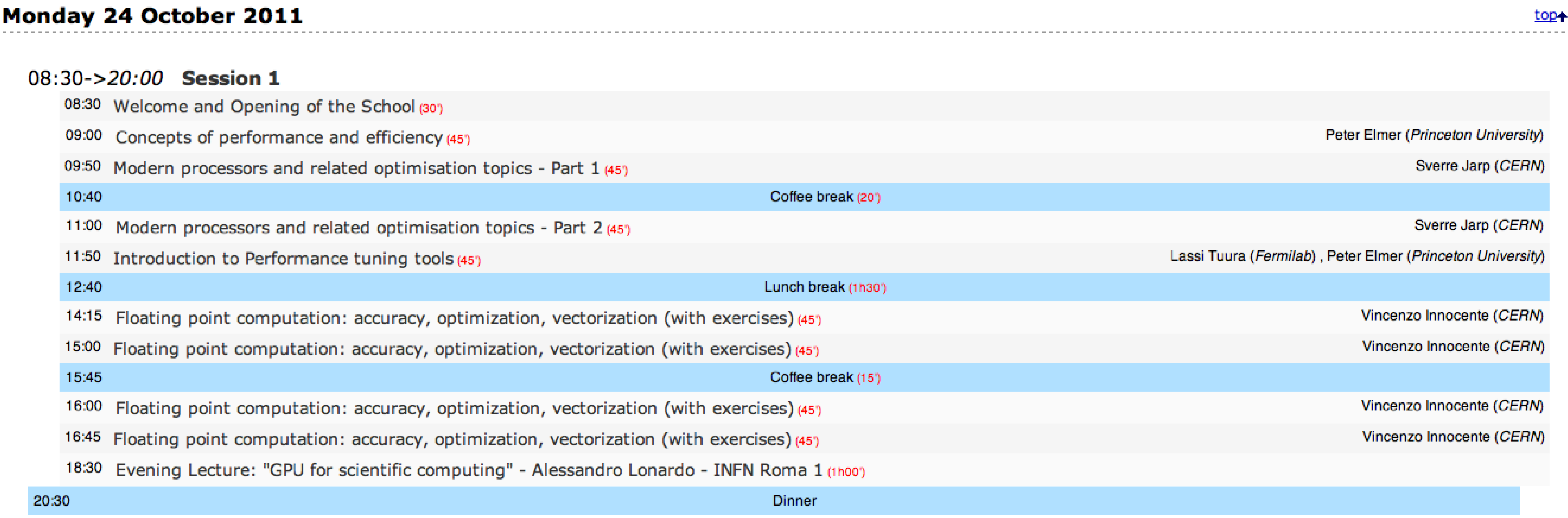
- It should be clear that in a complex environment (CPU, compiler, OS) the best benchmark you can make is simply to run the actual application, with real inputs and configured to make real outputs.
- At times “kernels” can be useful, i.e. small portions of code extracted from a real application after being identified by profiling as performance critical.
- Artificial benchmarks (e.g. specXXX) are less interesting for performance work, except perhaps as means of exploring and understanding the capabilities of processors.

Lecturers

- Sverre Jarp (CERN Openlab)
- Sebastien Binet (LAL) - LHC/Atlas
- Vincenzo Vagnoni (Bologna) – LHC/LHCb
- Peter Elmer (Princeton) - LHC/CMS
- Lassi Tuura (FNAL) - LHC/CMS
- Vincenzo Innocente (CERN) - LHC/CMS
- Niko Neufeld (CERN)
- Tim Mattson (Intel)
- Andrea Arcangeli (Redhat)
- Alessandro Lonardo (Rome)

Monday

After these two introductory talks today, the first focus is on the hardware and in particular modern processors. We will you a short overview of tools you will use this week and then dive into the first basic topic for scientific computing, floating point computation. There will be an evening lecture on GPU's for scientific computing.



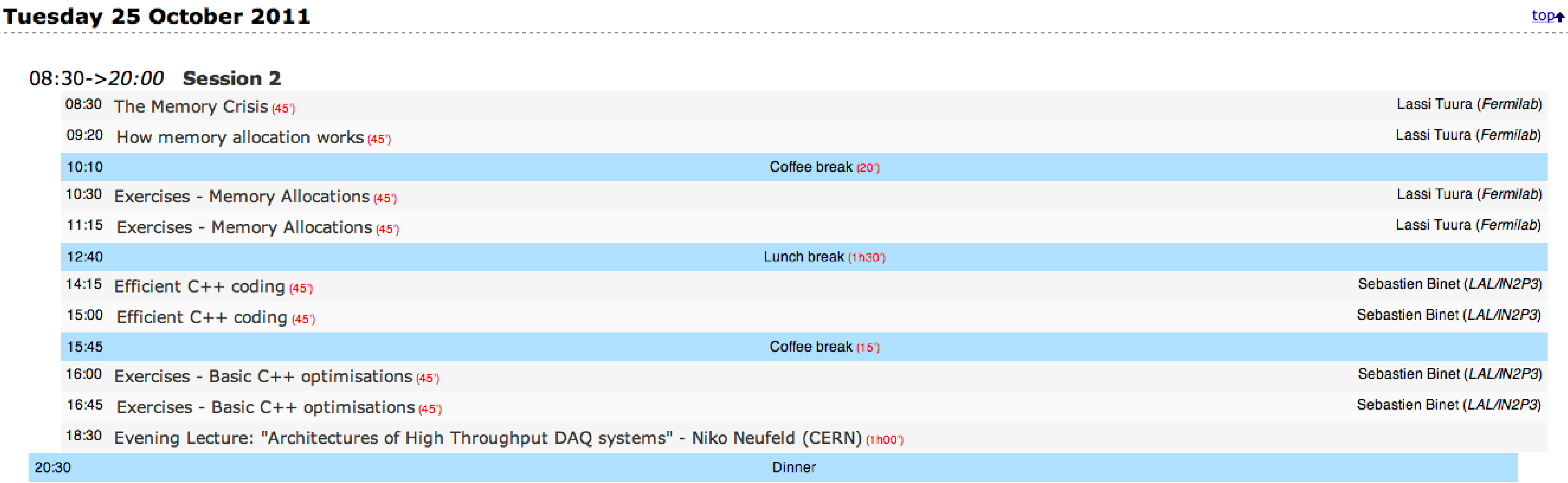
Monday 24 October 2011 top

08:30->*20:00* **Session 1**

08:30	Welcome and Opening of the School (30')	
09:00	Concepts of performance and efficiency (45')	Peter Elmer (*Princeton University*)
09:50	Modern processors and related optimisation topics - Part 1 (45')	Sverre Jarp (*CERN*)
10:40	Coffee break (20')	
11:00	Modern processors and related optimisation topics - Part 2 (45')	Sverre Jarp (*CERN*)
11:50	Introduction to Performance tuning tools (45')	Lassi Tuura (*Fermilab*) , Peter Elmer (*Princeton University*)
12:40	Lunch break (1h30')	
14:15	Floating point computation: accuracy, optimization, vectorization (with exercises) (45')	Vincenzo Innocente (*CERN*)
15:00	Floating point computation: accuracy, optimization, vectorization (with exercises) (45')	Vincenzo Innocente (*CERN*)
15:45	Coffee break (15')	
16:00	Floating point computation: accuracy, optimization, vectorization (with exercises) (45')	Vincenzo Innocente (*CERN*)
16:45	Floating point computation: accuracy, optimization, vectorization (with exercises) (45')	Vincenzo Innocente (*CERN*)
18:30	Evening Lecture: "GPU for scientific computing" - Alessandro Lonardo - INFN Roma 1 (1h00')	
20:30	Dinner	

Tuesday

On Tuesday we will cover the other two basic issues, memory management and use and efficient C++ programming. There will be an evening lecture on high throughput DAQ systems:



Tuesday 25 October 2011

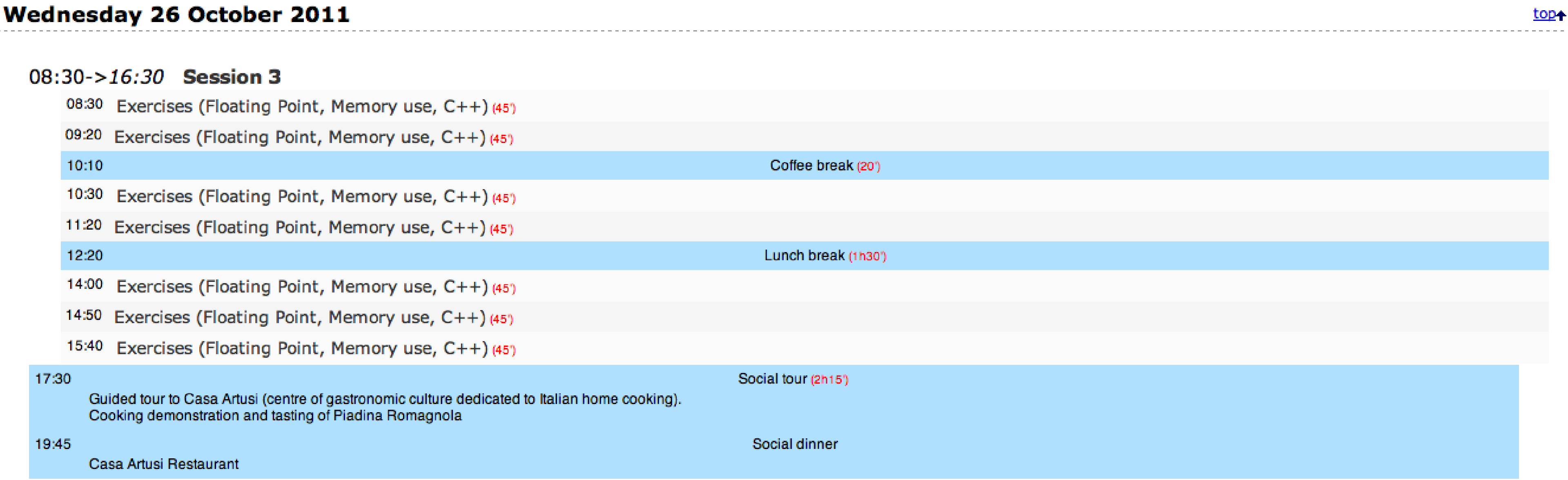
top

08:30->*20:00* **Session 2**

08:30	The Memory Crisis (45')	Lassi Tuura (*Fermilab*)
09:20	How memory allocation works (45')	Lassi Tuura (*Fermilab*)
10:10	Coffee break (20')	
10:30	Exercises - Memory Allocations (45')	Lassi Tuura (*Fermilab*)
11:15	Exercises - Memory Allocations (45')	Lassi Tuura (*Fermilab*)
12:40	Lunch break (1h30')	
14:15	Efficient C++ coding (45')	Sebastien Binet (*LAL/IN2P3*)
15:00	Efficient C++ coding (45')	Sebastien Binet (*LAL/IN2P3*)
15:45	Coffee break (15')	
16:00	Exercises - Basic C++ optimisations (45')	Sebastien Binet (*LAL/IN2P3*)
16:45	Exercises - Basic C++ optimisations (45')	Sebastien Binet (*LAL/IN2P3*)
18:30	Evening Lecture: "Architectures of High Throughput DAQ systems" - Niko Neufeld (CERN) (1h00')	
20:30	Dinner	

Wednesday

Wednesday is devoted to exercises related to the basic topics covered in the first two days. This will be the opportunity to explore the topics more in depth relative to the smaller exercises seen during the lectures themselves.



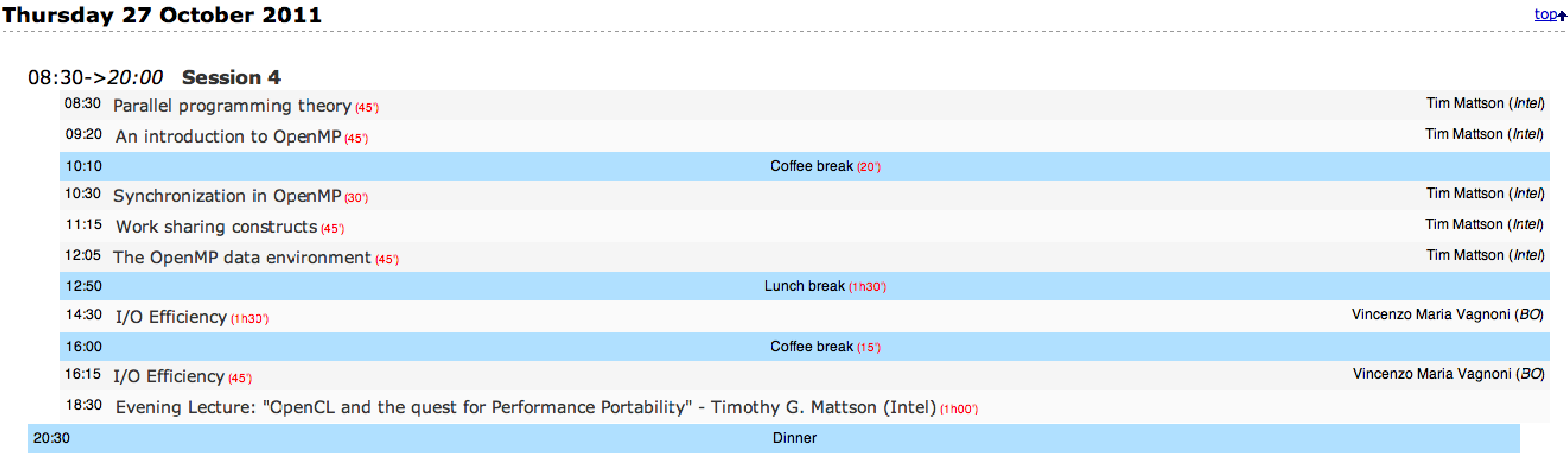
Wednesday 26 October 2011 top

08:30->*16:30* **Session 3**

08:30	Exercises (Floating Point, Memory use, C++) (45')
09:20	Exercises (Floating Point, Memory use, C++) (45')
10:10	Coffee break (20')
10:30	Exercises (Floating Point, Memory use, C++) (45')
11:20	Exercises (Floating Point, Memory use, C++) (45')
12:20	Lunch break (1h30')
14:00	Exercises (Floating Point, Memory use, C++) (45')
14:50	Exercises (Floating Point, Memory use, C++) (45')
15:40	Exercises (Floating Point, Memory use, C++) (45')
17:30	Social tour (2h15') Guided tour to Casa Artusi (centre of gastronomic culture dedicated to Italian home cooking). Cooking demonstration and tasting of Piadina Romagnola
19:45	Social dinner Casa Artusi Restaurant

Thursday

On Thursday we switch to more advanced and special topics: parallel programming and I/O:



Thursday 27 October 2011

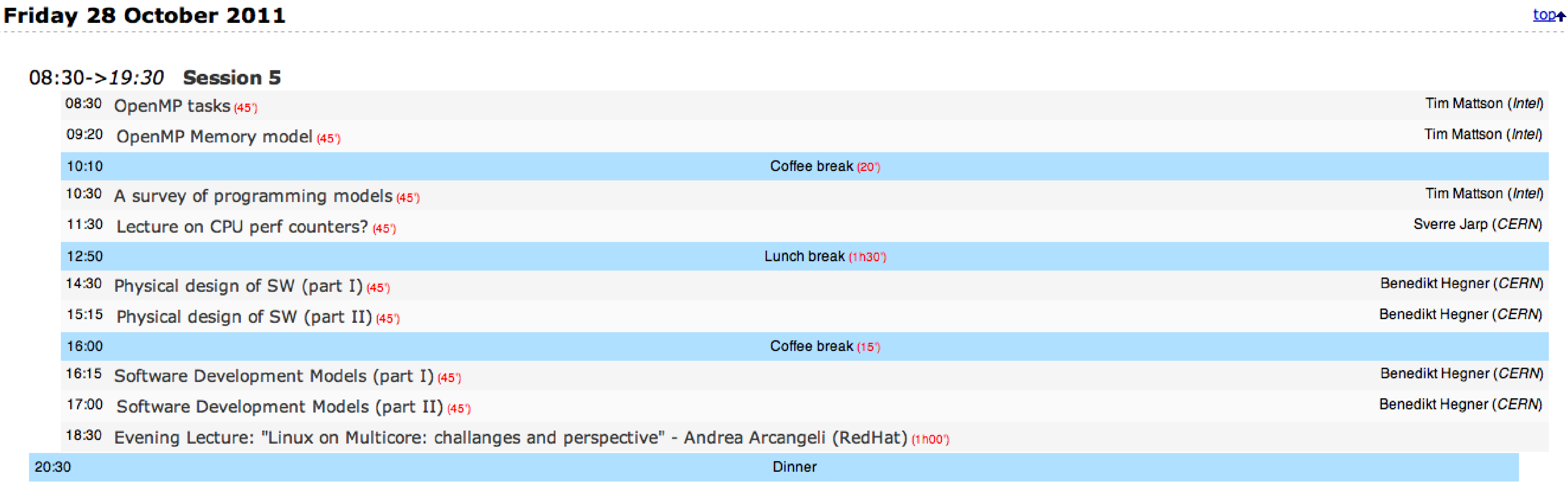
top

08:30->*20:00* **Session 4**

08:30	Parallel programming theory (45')	Tim Mattson (*Intel*)
09:20	An introduction to OpenMP (45')	Tim Mattson (*Intel*)
10:10	Coffee break (20')	
10:30	Synchronization in OpenMP (30')	Tim Mattson (*Intel*)
11:15	Work sharing constructs (45')	Tim Mattson (*Intel*)
12:05	The OpenMP data environment (45')	Tim Mattson (*Intel*)
12:50	Lunch break (1h30')	
14:30	I/O Efficiency (1h30')	Vincenzo Maria Vagnoni (*BO*)
16:00	Coffee break (15')	
16:15	I/O Efficiency (45')	Vincenzo Maria Vagnoni (*BO*)
18:30	Evening Lecture: "OpenCL and the quest for Performance Portability" - Timothy G. Mattson (Intel) (1h00')	
20:30	Dinner	

Friday

On Friday we will continue with parallel programming in the morning (and an evening lecture on linux on multicore), plus the special topic of software design and development models.



Friday 28 October 2011

08:30->*19:30* Session 5

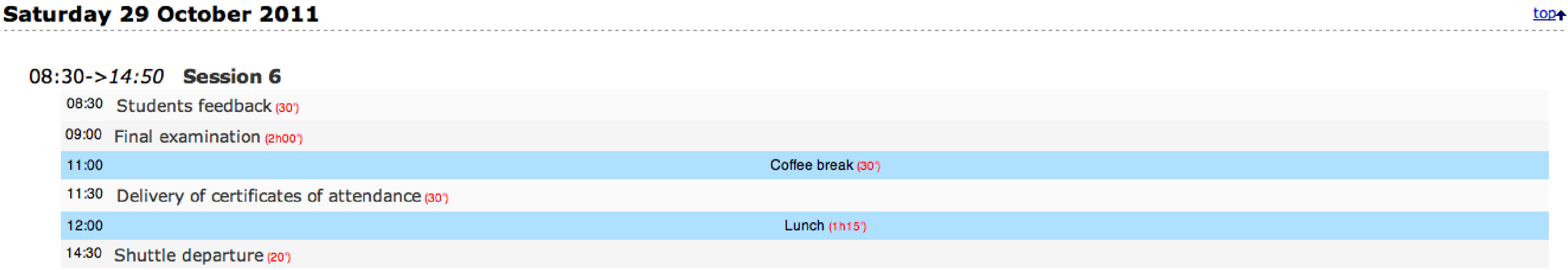
08:30	OpenMP tasks (45')	Tim Mattson (*Intel*)
09:20	OpenMP Memory model (45')	Tim Mattson (*Intel*)
10:10	Coffee break (20')	
10:30	A survey of programming models (45')	Tim Mattson (*Intel*)
11:30	Lecture on CPU perf counters? (45')	Sverre Jarp (*CERN*)
12:50	Lunch break (1h30')	
14:30	Physical design of SW (part I) (45')	Benedikt Hegner (*CERN*)
15:15	Physical design of SW (part II) (45')	Benedikt Hegner (*CERN*)
16:00	Coffee break (15')	
16:15	Software Development Models (part I) (45')	Benedikt Hegner (*CERN*)
17:00	Software Development Models (part II) (45')	Benedikt Hegner (*CERN*)
18:30	Evening Lecture: "Linux on Multicore: challanges and perspective" - Andrea Arcangeli (RedHat) (1h00')	
20:30	Dinner	

By the end of the week...

- … you should have a good working knowledge of performance issues related to:
  - The evolution of CPU architectures
  - The memory subsystem
  - C++ programming
  - Vectorization and floating point
  - Efficient I/O
  - Parallelization
- And you will have seen various related tools and done exercises for all of these topics.
- It is a very large number of topics for a few days, but you should be well positioned after this week to understand and improve the performance of your own applications.

Saturday

And of course at the end we would like feedback on whether we have succeeded plus there is a final examination to allow you to test your knowledge...



Saturday 29 October 2011 top

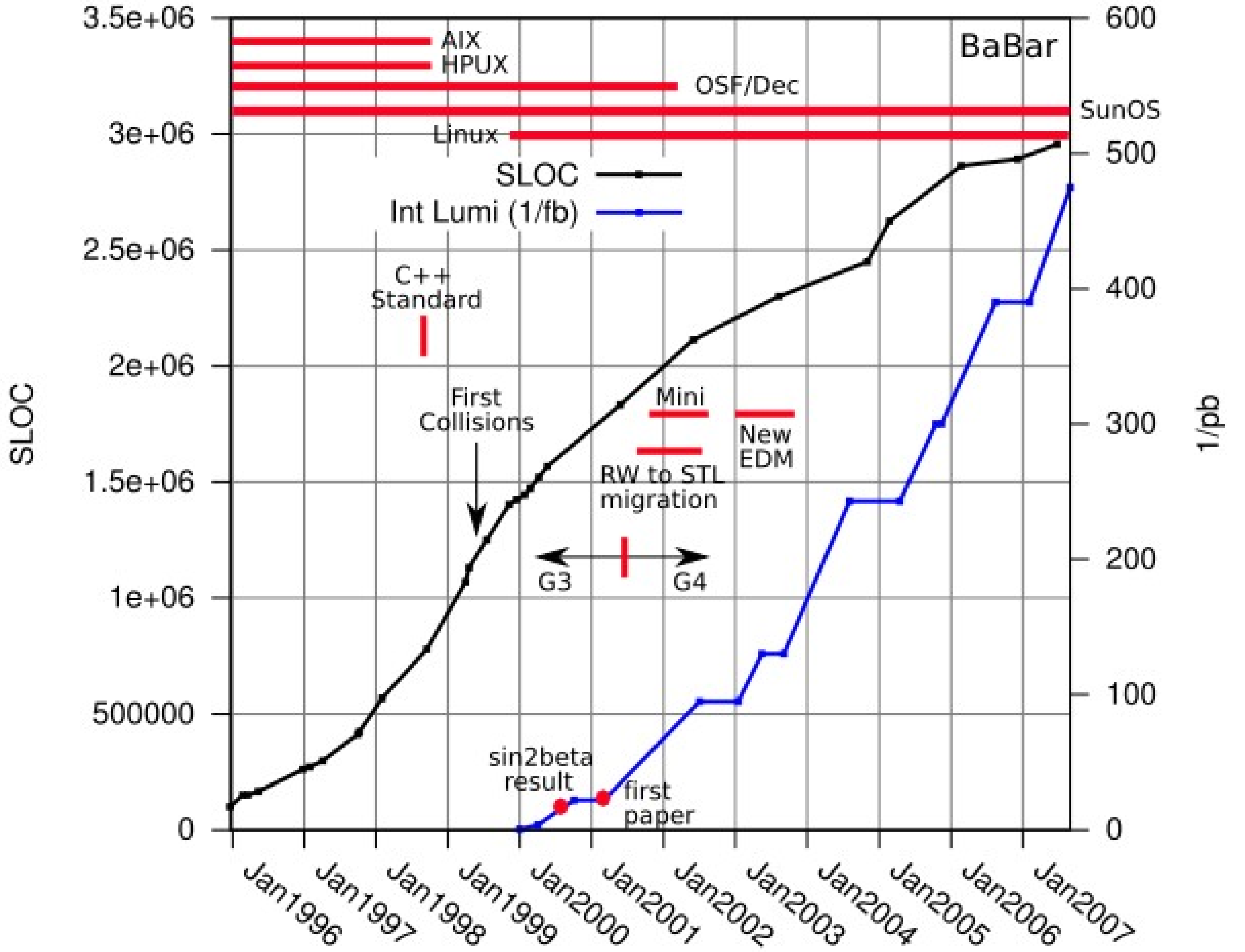
08:30->*14:50* **Session 6**

08:30	Students feedback (30')
09:00	Final examination (2h00')
11:00	Coffee break (30')
11:30	Delivery of certificates of attendance (30')
12:00	Lunch (1h15')
14:30	Shuttle departure (20')

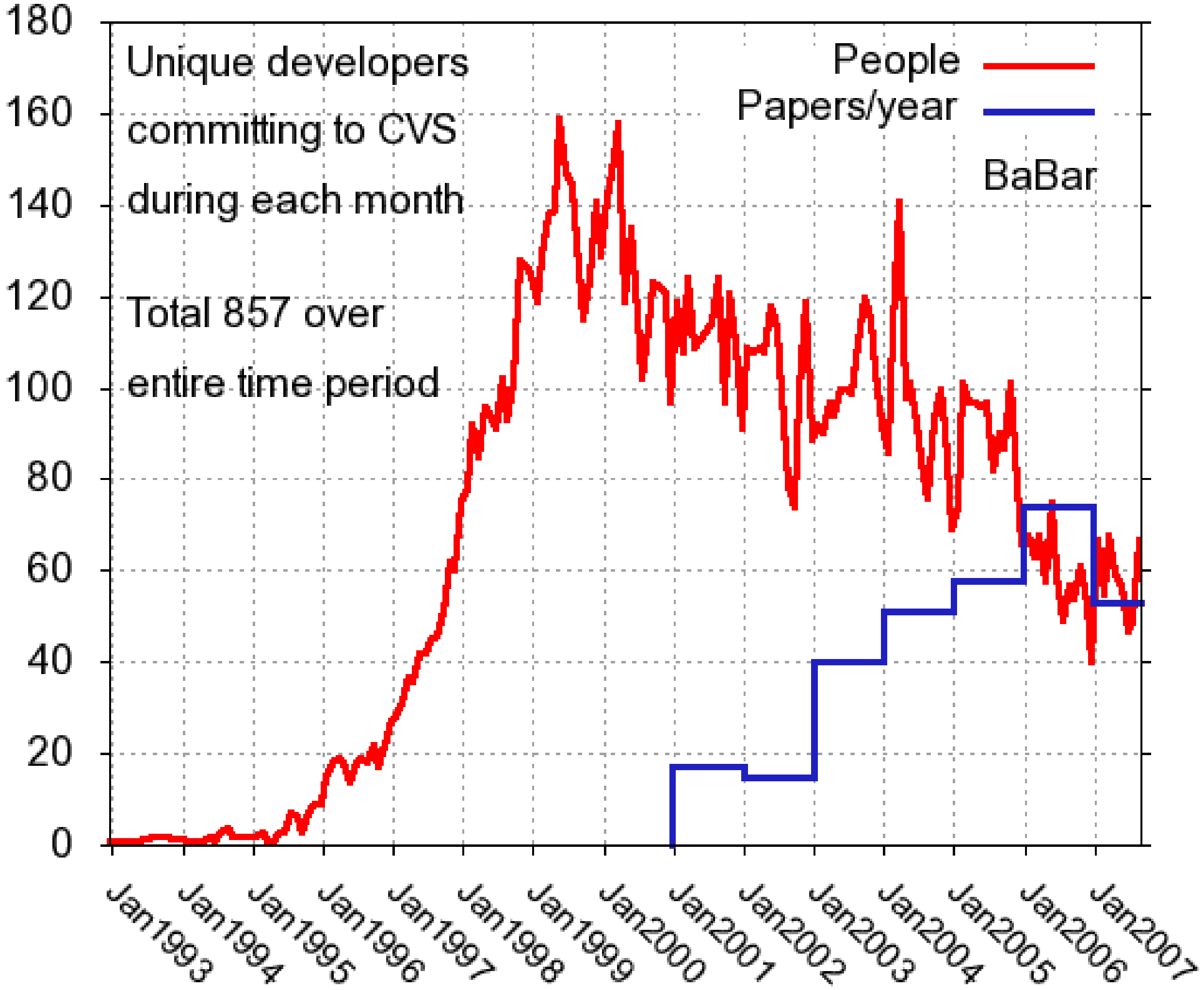
Code lifetimes

- Large scientific projects by definition will extend over many years and sometimes decades
- Technologies change over time and in any regime where underlying laws are exponential (i.e. Moore's Law), the one thing you can guarantee is that new challenges will arise...

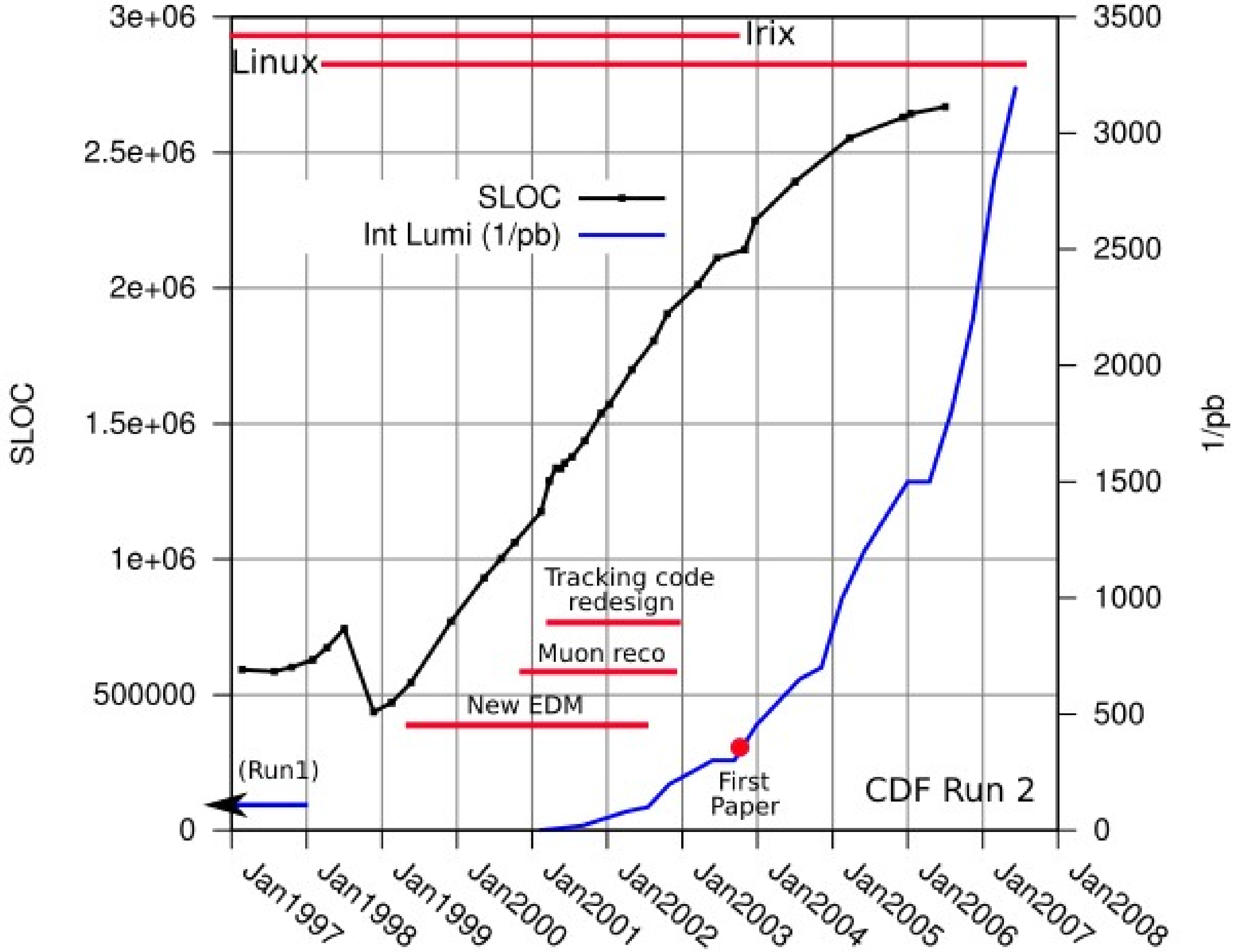
Code evolution - BaBar



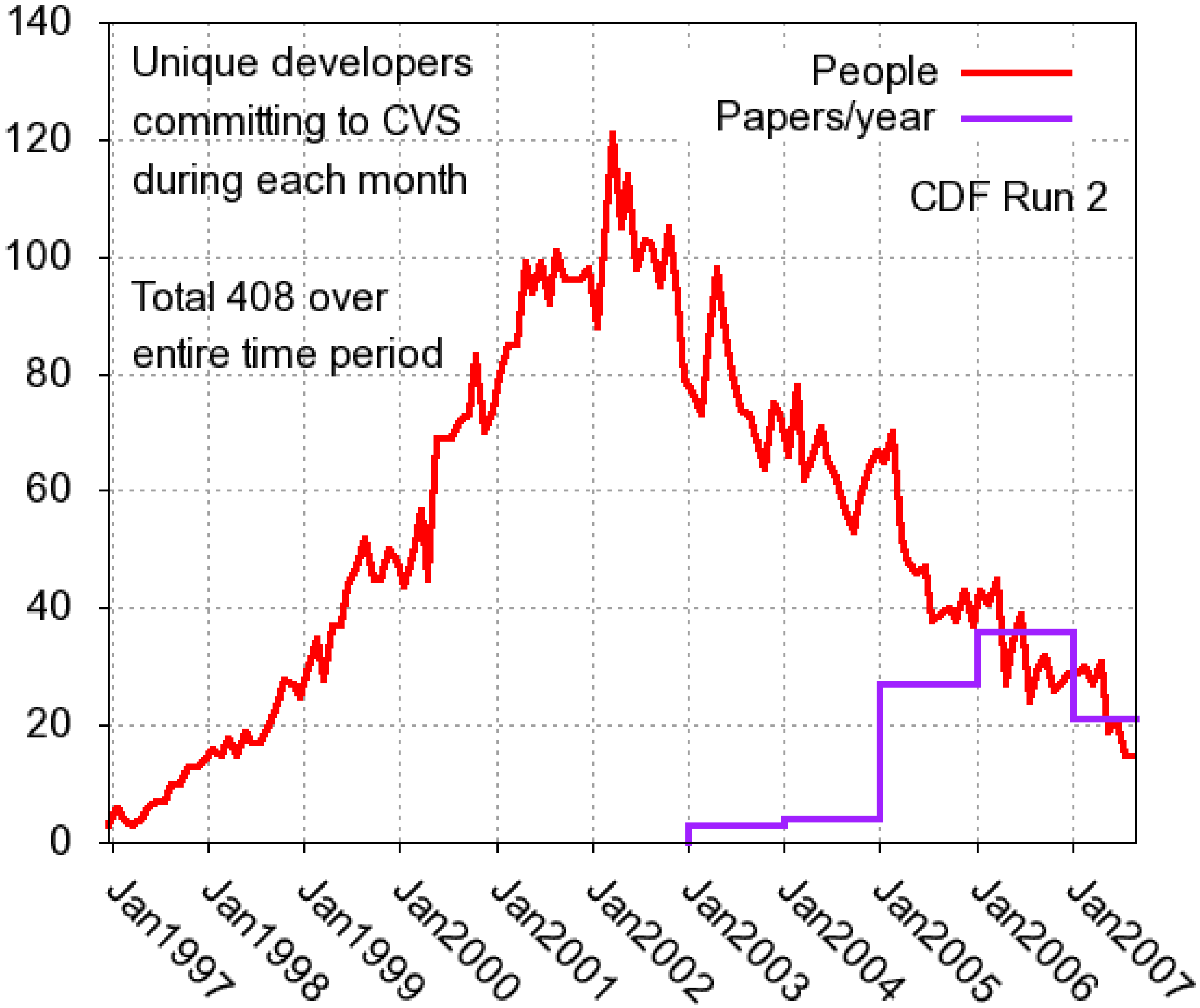
Code evolution - BaBar



Code Evolution – CDF Run II



Code Evolution – CDF Run II



Conclusions

Have a productive week!