Università degli Studi di Torino
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Fisica delle Interazioni Fondamentali

Tesi di Laurea Magistrale

A Prototype of a dynamically expandable Virtual Analysis Facility for the ALICE Experiment

Dario Berzano

Relatore
Prof. Massimo Masera

Correlatore
Dott. Stefano Bagnasco

Controrelatore
Prof. Giovanni Pollarolo

Anno Accademico 2007/2008

«Il fascismo privilegiava i somari in divisa. La democrazia privilegia quelli in tuta. In Italia, i regimi politici passano. I somari restano. Trionfanti.»

Indro Montanelli

Contents

Introduction

Although the LHC experiments at CERN – ALICE, ATLAS, CMS and LHCb – are widely known as a physical tool that will hopefully provide a better explanation of interactions of complex and dynamically evolving systems of finite size, thus extending the theory of elementary particles and their fundamental interactions (the Standard Model), what is less evident is the ancillary technology that has been developed in order to make this possible.

Among all other technologies, computing plays a fundamental role in LHC experiments: simulating, storing and analyzing data requires both hardware and software infrastructures that are under constant and heavy development. LHC experiments cannot simply rely on existing "stable" and commercialized computing infrastructures, because they constitute a completely new use case in the computing world: this is the reason why *physicists write most of the software they need on their own*, and why these infrastructures are to be considered "leading-edge technologies", and may even constitute, in the near future, the bases for a change in the way computers are used by "ordinary" people, as it happened for the World Wide Web[7, 6], born to satisfy the need to share documents within the HEP community.

The main large-scale computing infrastructure introduced by the LHC experiments is the Grid[13], a distributed computing infrastructure designed in order to allow equal access to data and computing power by every physicist in every part of the world. The only inconvenients from the technical point of view are that the Grid is not interactive, and the physicist should wait a random time in order for its tasks to be executed, making it not usable for tasks that need, for instance, to be repeated several times with different parameters in order to choose the better ones: this is the reason why a solution like PROOF[5] was developed, and particularly well integrated with the ALICE experiment's software.

Interactive analysis is complementary to the Grid and does not represent an alternative to it: the two models do have to coexist and share resources because the costs for a dedicated interactive facility could seldom be afforded. This is the reason why efforts were made in order to efficiently share resources between

batch and interactive tasks, mainly by running daemons as ordinary batch jobs that enable interactivity on demand on the Grid[16, 17].

This thesis is about the development, implementation and testing of a working prototype of a PROOF facility in Torino's Tier-2 centre that shares computing resources with the Grid dynamically (i.e. resources can be assigned or cut without any service interruption) through the use of virtual machines – a method that moves the problem of sharing resources from the Grid level to the machine level. We call our facility the **Virtual Analysis Facility**, and we'll discuss the advantages and disadvantages of this approach, by showing with the aid of benchmarks that virtualizing a high-performance and high-throughput computing infrastructure is nowadays feasible.

❖ ❖ ❖

In **Chapter 1** the ALICE experiment is described both from the physical and from the computational point of view, in order to show which are the reasons why interactive analysis is essential in ALICE (and in other LHC experiments too), and what is the status of the computing model at present.

Chapter 2 describes the feasibility tests run to show that virtualization of HEP tasks through Xen is feasible, even in a configuration with dynamically-assigned resources that are changed while jobs are running.

Once demonstrated the feasibility, a prototype was developed, whose configuration and implementation are described in detail in **Chapter 3**. Finally, some tests on the working VAF prototype were done in order to test performance and different storage models with real-world ALICE jobs.

Lastly, in **Chapter 4** some use cases of PROOF concerning the ALICE experiment are presented by exploiting the VAF with different analysis tasks, and in particular a CPU-intensive heavy flavour vertexing of the $D^+ \to K^-\pi^+\pi^+$ open charm decay.

Chapter 1

Overview of the ALICE Experiment

1.1 The Physics of the ALICE Experiment

The ALICE experiment is one of the four experiments at the LHC at CERN. LHC is the largest particle accelerator in the world with a circumference of ∼27 km, and it's also the one with the highest center-of-mass energy, with 14 TeV for pp collisions. All the four experiments will acquire data during pp runs, but ALICE is the only one specifically designed for *heavy ion* collisions; ALICE does then provide two specific physics programs, which we are going to describe.

1.1.1 Heavy-ion collisions

The focus of heavy-ion physics is to study and understand how collective phenomena and macroscopic properties, involving many degrees of freedom, emerge from the microscopic laws of elementary particle physics. Specifically, heavy-ion physics addresses these questions in the sector of strong interactions by studying nuclear matter under conditions of extreme density and temperature.

The nucleon-nucleon center of mass energy for collisions of the heaviest ions at the LHC ($\sqrt{s} = 5.5$ TeV) will exceed that available at RHIC by a factor of ~ 30, opening up a new physics domain. Heavy-ion collisions at the LHC access not only a quantitatively different regime of much higher energy density, but also a qualitatively new regime, mainly because of the following reasons.

- A novel range of Bjorken-x values where strong nuclear gluon shadowing

is foreseen, can be accessed. The initial density of these low-x gluon is expected to be close to saturation.

- Hard processes are produced at sufficiently high rates for detailed measurements.
- Weakly interacting hard probes become accessible, thus providing information about nuclear parton distributions at very high Q^2.
- Parton dynamics dominate the fireball expansion.

All these features will allow an accurate study of the phase transition in the hot and dense hadronic matter environment.

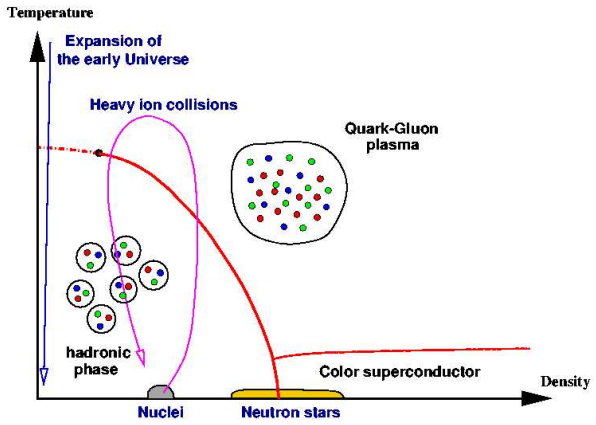


Figure 1.1: The QCD phase diagram.

The most striking case of a collective bulk of phenomenon predicted by the Standard Model is the occurrence of phase transitions in quantum fields at characteristic energy densities. The generic form of the QCD phase diagram is shown in Figure 1.1. Lattice calculations of QCD predicts that at a critical temperature of $T_c \simeq 170$ MeV, corresponding to an energy density of $\epsilon_c \simeq 1\ GeV/fm^3$, nuclear matter undergoes a phase transition to a deconfined state of quarks and gluons. In addition, at high temperature T and vanishing chemical potential μ_B (quantity related to baryon number density), chiral symmetry is approximately restored and quark masses are reduced from their large effective values in hadronic matter to their small bare ones.

The basic mechanism for deconfinement in dense matter is the Debye screening of the color charge. When the screening radius r_D becomes less than the binding radius r_h of the quark system (hadron), the confining force can no longer hold the quarks together and hence deconfinement sets in.

The phase transition can be well described by QCD thermodynamics, and in particular by finite temperature lattice calculations. However, the transition from hadronic matter to quark gluon plasma can be illustrated by simple and intuitive arguments, based on the "MIT bag model". This model describes the confined state of matter as an ideal gas of non-interacting massless pions, with essentially three degrees of freedom. On the contrary, even a two flavor QGP (composed by massless u and d quarks only), has 16 gluonic and 12 quark degrees of freedom. In the passage from a confined to a deconfined state, the energy density, which is proportional to the degrees of freedom, undergoes a sudden enhancement (latent heat of deconfinement). The behavior is shown in Figure 1.2, obtained with lattice calculations.

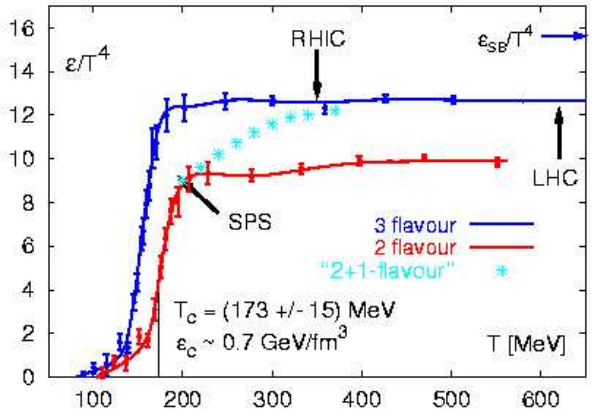


Figure 1.2: Temperature dependence of the energy density ϵ over T^4 in QCD with two and three degenerate quark flavors as well as with two light and a heavier *(strange)* quark. The arrows on the right-side ordinates show the value of the Stefan-Boltzmann limit for an ideal quark-gluon gas.

At present the only way to achieve the energy densities necessary for the QGP formation is through heavy ion collisions. The process that leads from the initial collision to hadronization and freeze-out, is described in Figure 1.3. The main steps follow.

- **Pre-equilibrium** ($\tau < 1$ fm/c). The initial partons scatter among each other giving rise to an abundant production of quarks and gluons.
- **QGP** ($\tau \simeq 10$ fm/c). The quark gluon gas evolves into thermal equilibrium: the QGP is formed and starts expanding.
- **Mixed phase**. The QGP, while expanding, starts converting into a hadron gas.

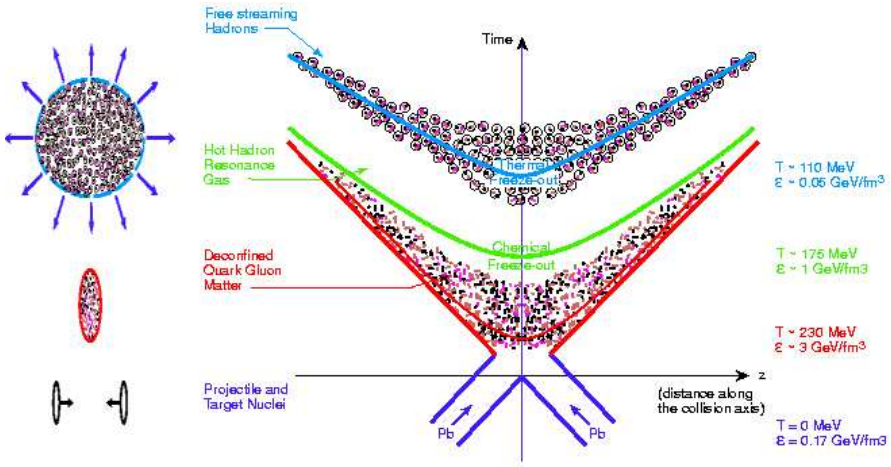


Figure 1.3: Expected evolution of a nuclear collision.

- **Hadronization** ($\tau \simeq 20$ fm/c). As far as the system expands, its temperature decreases until quarks and gluons are again confined in hadrons.

- **Freeze-out**. Hadrons decouple from the gas, thus becoming free.

The very short lasting time of QGP (only few 10^{-23} s), together with the interdiction to detect free quarks, do not allow to measure the transition directly. Nevertheless, information are indirectly provided by series of "probes", specifically thought to test different aspects of the medium. In the following, a short overview of such signals will be presented.

1.1.2 Experimental signatures of QGP

Phase transitions show critical behaviors, and the general way to probe such behaviors consists in finding the transition point and determine how the system and its observables change from one side to the other. In the case of complex phenomena, such as the QGP formation, different observables can be used in order to investigate different aspects of the same system, in many phases of its evolution.

1.1.2.1 Probes of the equation of state

The basic idea behind this class of probes is the identification of modifications in the dependence of energy density (ϵ), pressure (P) and entropy density (s) of superdense hadronic matter on temperature T and baryochemical potential μ_B.

A rapid rise in the ratios ϵ/T^4 or s/T^4 is, indeed, an evidence of a first-order phase transition.

The observables related to such quantities, obtainable through an analysis of the particle spectra, are the average transverse momentum $\langle p_T \rangle$ and the charged particle multiplicity per rapidity unit dN/dy or transverse energy per rapidity unit at mid-rapidities dE_T/dy. In particular, a first-order transition should manifest itself through a saturation of $\langle p_T \rangle$ during the mixed phase.

Particle spectra can provide information about another class of phenomena related to the equation of state: *the flow*, meaning a collective motion of particles superimposed to the thermal one. The flow is directly related to the pressure gradient, and can quantify the effective equation of state of the matter.

1.1.2.2 Signatures of chiral symmetry restoration

One of the most important probes of the chiral symmetry restoration comes from the study of the light vector meson resonances, ρ, ω and ϕ Such particles, created in the hot hadronic phase, can provide direct access to in-medium modifications. The ρ meson, in particular, plays a key role since its e^+e^- decay width (through which resonances are main ly detected) is a factor of $\sim$ 10 larger than the ω, and $\sim 5\times$ of ϕ. In addition, the ρ has a well-defined partner under $SU(2)$ chiral transformations, the $a_1(1260)$.

The approach toward restoration of chiral symmetry at temperature T_c requires the spectral distributions in the corresponding vector and axial channel to become degenerate. How this degeneracy occurs is one of the crucial questions related to the chiral phase transition.

The possibilities range from both the ρ and a_1 masses dropping to (almost) zero, the so-called Brown-Rho scaling conjecture, to a complete melting of the resonance structures, due to the intense rescattering in the hot and dense hadronic environment, or scenarios with rather stable resonance structures.

1.1.2.3 Soft probes of deconfinement: strangeness enhancement

In pp collisions the production of particles containing strange quarks is strongly suppressed, as compared to the production of hadrons with u and d quarks. The suppression, probably due to the higher mass of the $s\bar{s}$ pair, increases with the strangeness content of the particles.

QGP formation in nucleus-nucleus (AA) collisions leads to a different scenario. In this case the strange hadron yield derives from two independent reaction steps following each other in time:

1. in a deconfined QGP, strange quark pairs ($s\bar{s}$) can be copiously produced

through gluon-gluon fusion, while in hadronic gas $s\bar{s}$ pairs have to be produced via pairs of strange hadrons with higher production thresholds;

2. the initial s and $\bar{s}$ survive the process of fireball expansion, thus resulting, at hadronization, in an unusually high yield of strange and multi-strange baryon and anti-baryon abundance.

The process is represented in Figure 1.4.

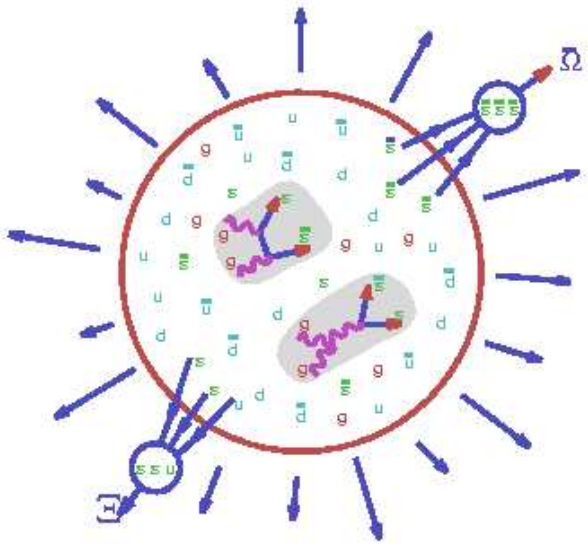


Figure 1.4: Illustration of the two steps mechanism of strange hadron formation from QGP.

In the ensuing hadronization, quark recombination leads to emergence of particles such as $\Xi(ssq)$ and $\bar{\Omega}(\bar{s}\bar{s}\bar{s})$, which otherwise could only very rarely be produced[24], as well as to a global increase of the strange particles production.

The described enhancement as a function of the centrality of collision has been already observed[11] in experiments such as NA57 at SPS, as it is clearly shown in Figure 1.5.

It is trivial to stress the importance of measuring strange production even in pA and pp collisions, as the enhancement can be noticed only in comparison with such data.

1.1.2.4 Hard and electromagnetic probes of deconfinement

In order to be sensitive to the onset of deconfinement, any probe must satisfy some requirements, and in particular they must:

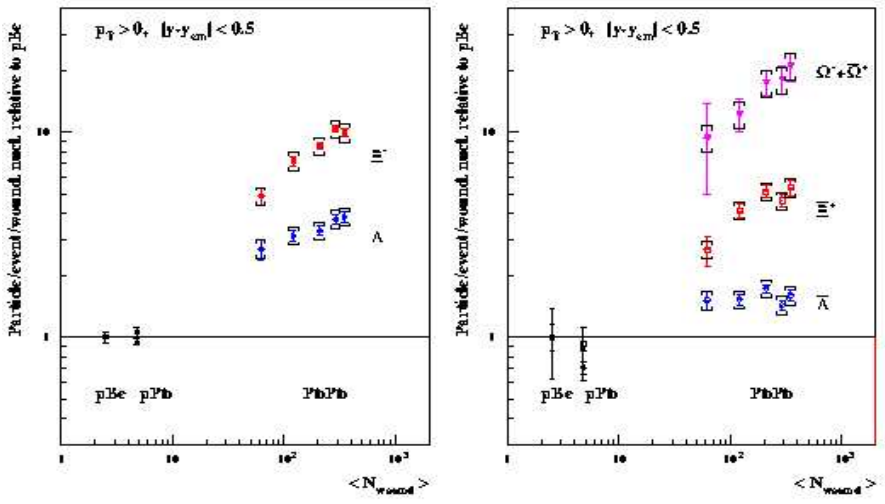


Figure 1.5: Centrality dependence of hyperon enhancements at 158 A GeV/c.

- be hard enough to resolve sub-hadronic scales;
- be able to distinguish confinement and deconfinement;
- be present in the early stage of the evolution;
- retain the information throughout the subsequent evolution.

The last point requires that probes should not be in thermal equilibrium with later evolution stages, since this would lead to a loss of memory of the previous stages.

So far, two types of probes satisfying these conditions fully or in part have been considered.

- **External probes** are produced essentially by primary collisions, before the existence of any medium. Their observed behavior can indicate whether the subsequent medium was deconfined or not. The most important observables are the production of quarkonium states and the energy loss or attenuation of hard jets.
- **Internal probes** are produced by the quark-gluon plasma itself. Since they must leave the medium without being affected by its subsequent states, they should undergo only weak or electromagnetic interactions after their formation. Thus the main candidates are thermal dileptons and photons.

Quarkonium suppression was suggested as a signal of deconfinement[19] long ago. If a heavy quark bound state ($Q\bar{Q}$) is placed into a hot medium of deconfined quarks and gluons, color screening will dissolve the binding, so that the Q and $\bar{Q}$ separate. When the medium cools down to the confinement transition point, they will therefore in general be too far apart to see each other, and the heavy quark will combine with a light one to form heavy flavored mesons.

Due to their small size, quarkonia can, in principle, survive the deconfinement phase transition. However, because of color screening, no bound state can exist at temperatures $T > T_D$, when the screening radius, $1/\mu_D(T)$ becomes smaller than the typical bound-state size.

With increasing temperature, a hot medium will lead to successive quarkonium melting: bigger size resonances, such as χ_c and ψ' are dissolved first, while more tightly bound states, such as J/ψ, are destroyed later. Hence the suppression of specific quarkonium states serves as a thermometer of the medium.

In fact, a slight reduction in quarkonium production can be noticed even in ordinary nuclear matter, due to absorption by nucleons and comoving secondaries. In order to take into account this effects, it is of extreme importance to achieve a good knowledge of the quarkonia absorption cross section behavior from pA and pp data. Only when such a baseline is clearly understood, it is finally possible to search for "anomalous" suppression patterns, which are a clear signature of deconfinement.

Evidences of the phenomenon have been found by the NA50 experiment at CERN SPS[3], as shown in Figure 1.6.

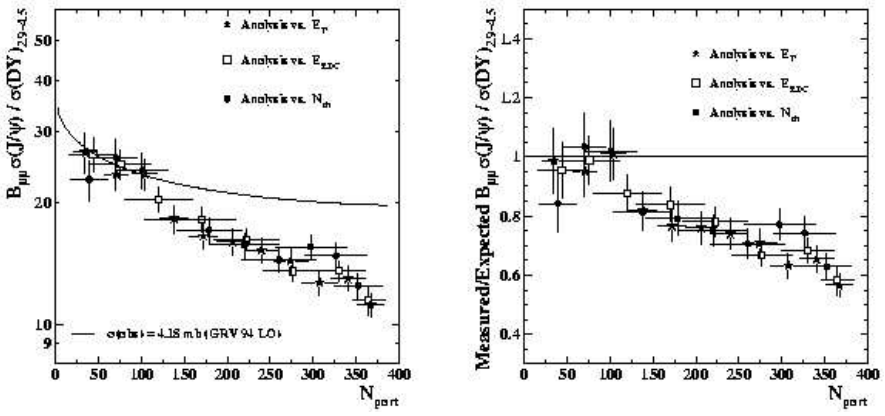


Figure 1.6: The J/ψ/Drell-Yan cross section ratio as a function of N_{part} for three analysis of the $PbPb$ 2000 data sample in Na50, compared to *(left)* and divided by *(right)* the normal nuclear absorption pattern[3].

The LHC will add a significant contribution in the understanding of QGP via heavy quarkonia probes. The achievable energy is unique for suppression studies since it allows, for the first time, the spectroscopy of charmonium and bottomonium states in heavy ion collisions. In particular, because the Υ is expected to dissolve significantly above the critical temperature, the spectroscopy of the Υ family at the LHC energies should reveal unique information on the characteristics of the QGP.

On the other hand, the study of heavy quark resonances in heavy ion collisions at the LHC is subject to significant differences with respect to lower energies. In addition to prompt charmonia produced directly via hard scattering, secondary charmonia can be produced from bottom decay, $D\bar{D}$ annihilation, and by coalescence mechanisms which could result in enhancement rather than suppression.

The role of jets as a deconfinement probe was first proposed in 1982 by Bjorken. He stressed t hat a "high-p_T or gluon might lose tens of GeV of its initial transverse momentum while plowing through quark-gluon plasma produced in its local environment". While Bjorken estimates based on collisional energy loss had to be revised, it was later suggested that the dominant energy-loss mechanism is radiative rather than collisional. In particular, the mechanism is not the direct analogous of the Abelian bremsstrahlung radiation, but a genuine non-Abelian effect: gluon rescattering.

Since the partonic energy loss grows quadratically with the in-medium path length and is proportional to the gluon density, the observation of jet quenching in heavy ion collisions can be accounted as a proof of deconfinement. It is clear that, in order to notice a " quenching", comparisons with jets in ordinary matter have to be performed. An important benchmark for fragmentation function of jets will be provided by analyses of pp collisions.

Hadronic probes are not the only ones able to give information on the formed medium. A lot of advantages can arise from the use of *electromagnetic probes*. Indeed, owing to their small coupling, photons, once produced, don't interact with the surrounding matter and can thus provide information on the state of matter at the time of their formation.

The production of photons in the different stages of heavy-ion collision can be summarized qualitatively as follows.

1. Early in the collisions, so-called "prompt" photons are produced by *parton-parton* scattering in the primary nucleus-nucleus collisions. An important background to such photons is the decay $\pi^0 \rightarrow \gamma\gamma$.

2. In the following stage of the collision, a quark-gluon plasma is expected to be formed with a temperature up to 1 GeV. Photons are radiated off the quarks which undergo collisions with other quarks and gluons in the

thermal bath. The energy spectrum of these photons is exponentially suppressed, but should extend up to several GeV.

3. The plasma expands and cools. At the critical temperature, a hadronic phase is formed, during which photons can be produced in hadron rescattering or in resonance decays. The mechanism continues until the resonances cease to interact, that means until the freeze-out temperature ($\sim$ 100 MeV) is reached. Photons produced in this phase will have energies between few hundred MeV and several GeV.

4. Finally, after freeze-out, further photons can be produced by the decay of π^0's, η's and higher resonances. Their energy lies in the range of up to few hundred MeV.

The prompt photons of phase one constitutes an "irreducible" background to thermal photons of phase two and three. Such background has to be kept under control, for example via comparison to the pp benchmark. The occurrence of an excess in thermal photons (after background subtraction) in the few GeV range, would be a clear indication of a thermalized medium.

Lepton pair production shows analogies with the photon generation. In fact, they are emitted throughout the evolution of the system, and with the same stages described above.

The prompt contribution to the continuum in the dilepton mass range above pair mass $M \simeq 2$ GeV is dominated by semileptonic decays of heavy flavor mesons and by the Drell-Yan process (Figure 1.7). The latter was particularly important in previous experiments, not as a deconfinement probe, but because it gives information on the initial state. Its prediction were usually adopted as a benchmark in heavy ion collisions, as it is affected only by ordinary nuclear matter effects, but it is not modified by the formation of a hot dense system. However, in the LHC, it is overwhelmed by heavy quark decays, which dominate the lepton pair continuum between the J/ψ and the Z^0 peaks.

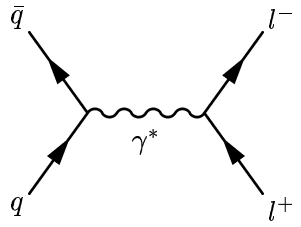


Figure 1.7: Drell-Yan process.

Dileptons have the same functionality as photons in the different stages of the system evolution, but, in addition, they offer distinct advantages. In particular, lepton pairs carry an additional variable, the pair invariant mass, which encodes dynamical information on the vector excitations of matter.

At masses above $\sim$ 1.5 GeV thermal radiation is expected to originate from the early hot phases, with a rather structureless emission rate determined by perturbative $q\bar{q}$ annihilation. The physics objective is then similar to that of the photon case, which is the discrimination of the QGP radiation from the large prompt background.

At low masses (less than 1.5 GeV), thermal dilepton spectra are dominated by radiation from the hot hadronic phase. Here, the electromagnetic current is saturated by the light vector mesons (ρ, ω and ϕ), allowing direct access to their in-medium modifications.

A schematic view of the characteristic dilepton sources in ultrarelativistic heavy ion collisions is shown in Figure 1.8. The plot was obtained for center of mass energies lower than the LHC one, and it is shown here in order to get a rough idea of the dilepton mass distribution. As previously said, at high energies contributions from bottom flavored hadrons become important.

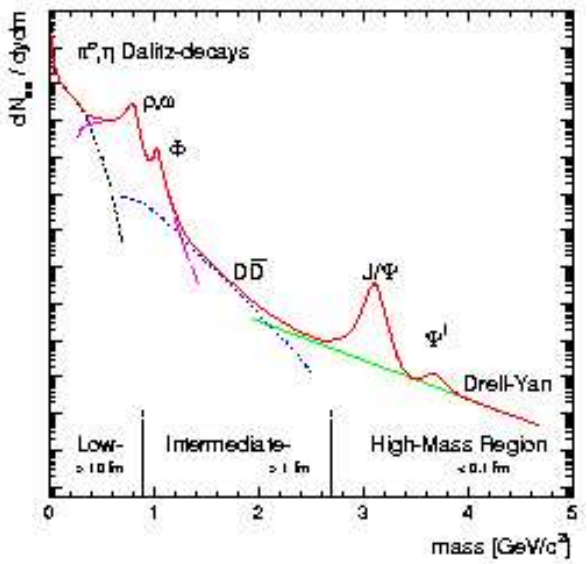


Figure 1.8: Expected sources for dilepton production as a function of invariant mass in ultra-relativistic heavy ion collisions. The plot, obtained for lower energies than LHC, is meant to give a rough idea of mass spectra in the low mass region. At higher energies contributions from bottom flavored hadron decays become important.

As in the case of photons, the determination of details of the medium effect relies on comparisons to smaller systems and to pp collisions.

1.1.2.5 Proton-proton collisions

ALICE has several features that make it an important contributor to proton-proton physics at the LHC. Its design allows particle identification over a broad momentum range, powerful tracking with good resolution from 100 MeV/c to 100 GeV/c, and excellent determination of secondary vertices. These, combined with a low material thickness and a low magnetic field, will provide unique information about low-p_T phenomena in pp collisions.

A benchmark for heavy ion physics The identification of phenomena due to the formation of a new state of matter needs a good knowledge of ordinary nuclear matter effects, that can be achieved through comparison with pp collisions. A long list of observables have to be analyzed to this aim; some of them have already been presented in the previous section, but the overview will be completed in the following.

- **Particle multiplicities:** differences in particle multiplicities between pp and AA are related to the features of parton distributions in the nucleon with respect to those in nuclei *(shadowing)* and to the onset of saturation phenomena occurring at small x.
- **Particle yields and ratios:** particle ratios are indicative of the chemical equilibration achieved in AA collisions and should be compared to those in pp collisions.
- **Ratios of momentum spectra:** the ratios of transverse momentum spectra at sufficiently high momenta allow to discriminate between the different partonic energy losses of quarks and gluons. In particular, due to their different color representation, hard gluons are expected to loose approximatively a factor of two more energy than hard quarks.

The dominant error for all these observables is often due to the systematics. In order to reduce it, it is thus of fundamental importance to measure the physical quantities in the same experimental setup, as it will be done in ALICE.

Specific aspects In addition to the benchmark role for $PbPb$ collisions the study of pp physics in the ALICE experiment has an importance of its own. In particular, the characteristics of the LHC will allow the exploration of a new range of energies and Bjorken-x values. More generally, the ALICE pp programme aims

at studying non-perturbative strong coupling phenomena related to confinement and hadronic structure. The main contribution will be in the low transverse momentum domain for which the ALICE detector was optimized.

During pp collisions, ALICE efforts will be focused in the study of a large amount of observables.

- **Particle multiplicity.** Feynman predicted a simple scaling law for the $\sqrt{s}$ dependence of the observable, which was proved to be only approximate. Thus, a direct measurement is necessary, in order to get a better understanding of the phenomenon. Moreover the high statistics charged multiplicity study could allow to get access to the initial states, where new physics such as high-density effects and saturation phenomena sets in.

- **Particle spectra.** The analysis will allow to study the minijet[1] contribution, by determining the hardening of p_T spectra and various correlations between particles with high p_T.

- **Strangeness production.** The possibility to access considerably increased charged particle densities, together with a better control of the transverse momentum measurements, should allow ALICE to explain not well understood phenomena observed in previous experiments. One of these is the fact that correlation between the mean kaon transverse momentum and the charged particle multiplicity observed at the Tevatron is significantly stronger than that for pions.

- **Baryon number transfer in rapidity.** The rapidity distribution of baryon number in hadronic collisions is not understood. A number of models provide explanation of the experimental data, some involving diquark exchange, some others adopting purely gluonic mechanism. The ALICE detector, with its particle identification capabilities, is ideally suited to clarify this issue with abundant baryon statistics in several channels in the central-rapidity region.

- **Correlations.** Two-particle correlations have been traditionally studied in pp multiparticle production in order to gain insight into the dynamics of high energy collisions via a unified description of correlations and multiplicity.

[1] Jets whose E_T, though larger than the hadronic scale, is much smaller than the hadronic center of mass energy $\sqrt{s}$ (at the LHC it means $E_T \leq 10$GeV). Such jets cannot be understood solely in terms of the fragmentation of partons of comparable E_T, produced in a hard subprocess. The minijets also receive a contribution from the dynamics of underlying events, which in nucleus-nucleus collisions have a substantial transverse activity.

- **Heavy flavor production.** The low p_T cutoff for particle detection will require a smaller extrapolation of the total heavy flavor cross section, thus improving precision and clarifying underestimations of some theoretical models predictions.

- **Jet studies.** Owing to its ability to identify particles and measure their properties in a very high density environment, the detector will be able to study jet fragmentation in a unique way.

- **Photon production.** Although the production of photons at large transverse momentum has been extensively studied, no good agreement between experiment and theory has yet been achieved. The rate of production is essentially proportional to the gluon distribution in the proton, which can be probed directly by looking at the p_T dependence of the photon spectrum. ALICE will be able to measure prompt-photon production in a region where the not well known gluon fragmentation into photon is dominant, and where NLO calculations become insufficient, thus needing only recently explored theories.

- **Diffractive physics.** Even in this case, new physical regions will be reached, because ALICE should be able to observe central-region events with large rapidity gaps as well as very low x phenomena (down to 10^{-6}). Investigation of the structure of the final hadronic state (with particle identification) produced in diffractive processes can provide important information on the mechanism of high energy hadronic interactions.

- **Double parton collisions.** First measurements at Tevatron of double parton collisions show non-trivial correlations of the proton structure in transverse space, which indicate that the structure of the proton is much richer than the independent superposition of single-parton distribution functions accessible by deep-inelastic scattering. Since increasing the center of mass energy leads to an increase of the parton fluxes, it is clear that at LHC multiple-parton collisions will gain more and more importance, thus allowing a deeper study of the phenomenon.

1.2 Overview of the ALICE detector

The ALICE experiment was first proposed as a central detector in the 1993 Letter of Intent (LoI), and later complemented by an additional forward muon spectrometer designed in 1995. It is a general-purpose heavy-ion experiment, sensitive to the majority of known observables (including hadrons, electrons, muons and photons). ALICE was designed in order to measure the flavor content and

phase-space distribution, event by event, for a large number of particles whose momenta and masses are of the order of the typical energy scale involved (temperature $\sim \Lambda_{QCD} \sim 200$ MeV). The experiment will be able to cope with the highest particle multiplicities anticipated for $PbPb$ reactions ($dN_{ch}/dy = 8000$).

The ALICE detector Figure 1.9 has the typical aspect of detectors at colliders, with a cylindrical shape around the beam axis, but with in addition a forward muon spectrometer, detecting muons in a large pseudorapidity domain. Moreover, the central barrel angular acceptance is enhanced by detectors located at large rapidities, thus allowing measurements of low p_T particles and of global event structure. ALICE can be divided in three parts:

1. the **central part**, which covers $\pm 45^\circ$ (corresponding to the pseudorapidity interval $|\eta| < 0.9$) over the full azimuth and is embedded in a large magnet with a weak solenoidal field. It consists (from the inside out) of:
   - an Inner Tracking System (ITS);
   - a cylindrical Time Projection Chamber (TPC);
   - a Transition-Radiation Detector (TRD);
   - a large area Particle Identification (PID) array of Time Of Flight (TOF) counters;
   - an electromagnetic calorimeter: PHOS;
   - an array of counters optimized for High-Momentum inclusive Particle Identification (HMPID);
2. the **forward detectors**, constituted of:
   - a Zero-Degree Calorimeter (ZDC);
   - a Forward Multiplicity Detector (FMD);
   - a Photon Multiplicity Detector (PMD);
3. the **Forward Muon Spectrometer (FMS)**.

1.2.1 Inner Tracking System (ITS)

The main purposes of the ITS are the detection of the primary and secondary vertices (hyperons and charm) and the stand-alone track finding of low p_T charged particles, down to $p_T \simeq 20$ MeV/c for electrons. Moreover it can be used to improve the momentum resolution at high momenta, to reconstruct low energy particles and to identify them via energy loss, and, in the end, to define the angles of the tracks for HBT interferometry analysis.

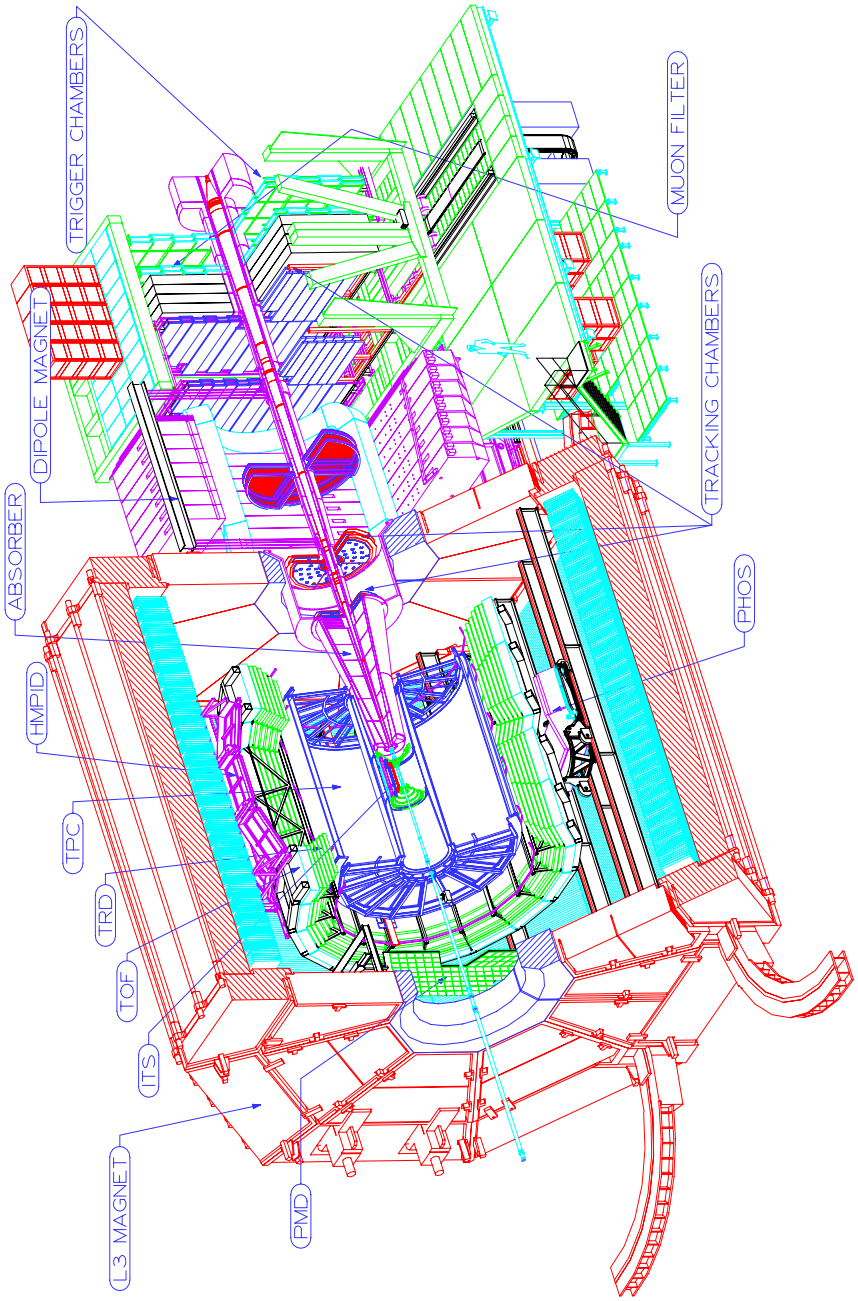


Figure 1.9: The ALICE detector.

The system consists of six cylindrical layers of coordinate-sensitive detectors. The granularity required for the innermost planes, given the expected high multiplicity of charged particle tracks, can only be achieved with silicon micro-pattern detectors with true twodimensional readout, such as SPDs and SDDs. In particular SPDs are used in the first two layers, SDDs in the third and fourth layers, while in the fifth and sixth, where requirements in term of granularity are less stringent, *strip detectors* are adopted.

1.2.2 Time Projection Chamber (TPC)

It is the main tracking detector of ALICE. Beyond track finding, it was specifically designed for momentum measurement and particle identification by dE/dx. The mean momentum of the particles tracked in the TPC is around 500 MeV/c. Despite being a comparatively slow detector, with about 90 μs drift time over the full length of 2.5 m, the time projection chamber can cope with the minimum-bias collision rate in $PbPb$ of about 8 kHz, expected for the design luminosity $\mathscr{L} = 10^{27}$ cm^{-2}s^{-1}.

1.2.3 Transition-Radiation Detector (TRD)

The TRD detector fills the radial space between the TPC and the TOF. It is constituted by a total of 540 detector modules, each consisting of a radiator and a multi-wire proportional readout chamber, together with its front-end electronic.

The detector will provide electron identification for momenta greater than 1 GeV/c, where the pion rejection capability through energy-loss measurement in the TPC is no longer sufficient. Such identification, in conjunction with ITS, will be used in order to measure open charm and open beauty, as well as light and heavy vector mesons produced in the collisions. Moreover, the combined use of TRD and ITS data will allow to separate the directly produced J/ψ mesons from those coming from B decays.

1.2.4 Particle Identification (PID)

Particle Identification (PID) over a large part of the phase space and for many different particles is an important design feature of ALICE. There are two detector systems dedicated exclusively to PID: a TOF and a small system specialized on higher momenta. The time of flight is a MRPC, with a resolution better than 100 ps. It will be used to separate pions from kaons in the momentum range $0.5 < p < 2$ GeV/c, i.e. from the TPC upper limit for K/π separation through dE/dx, to the statistics limit in single event. In addition it will be able to distinguish between electrons and pions in the range $140 < p < 200$ MeV/c.

The HMPID was specifically thought for hadron identification in the momentum region above 1.52 GeV/c. The dedicated detector was chosen to be a RICH, which provides a K/π and K/p separation up to 3.4 GeV/c and 5GeV/c e respectively.

1.2.5 Photon Spectrometer (PHOS)

The PHOS is an electromagnetic calorimeter designed to search for direct photons, but it can also detect γ coming from π^0 and η decays at the highest momenta, where the momentum resolution is one order of magnitude better than for charged particles measured in the tracking detectors. The study of the high momentum particles spectrum is extremely useful because it gives information about the propagation of jets in the dense medium created during the collision *(jet quenching)*.

In addition to photons, the PHOS also responds to charged hadrons and to neutral particles such as K_L^0, n and η. Some measures have to be taken in order to reject these particles, such as the inclusion of a charged-particle veto detector (MWPCs were adopted) in front of the PHOS for charged hadrons, and a cut on the shower width and on the time of flight for neutral particles. The calorimeter is placed at 4.6 m from the beam axis, covers the pseudorapidity region $|\eta| \leq 0.12$ 8 m^2.

1.2.6 Magnet

The last component of the central barrel is the magnet. The optimal choice for the experiment is a large solenoid with a weak field. The choice of a weak and uniform solenoidal field together with continuous tracking in a TPC eases considerably the task of pattern recognition. The field strength of ∼0.5 T allows full tracking and particle identification down to ∼ 100 MeV/c in p_T. Lower momenta are covered by the inner tracking system. The magnet of the L3 experiment fulfills the requirements and, due to its large inner radius, can accommodate a single-arm electromagnetic calorimeter for prompt photon detection, which must be placed at a distance of ∼5 m from the vertex because of the particle density.

1.2.7 Zero-Degree Calorimeter (ZDC)

The main aim of the ZDC is the estimate of the collision geometry through the measurement of the non interacting beam nucleons (the “spectators”). There are four calorimeters, two for neutrons and two for protons, placed at 116 m from the interaction point, where distance between beam pipes (∼8 cm) allows

insertion of a detector. At this distance, spectator protons are spatially separated from neutrons from magnetic elements of the LHC beam line.

The neutron detector is made up of a tungsten alloy, while the proton one is constituted of brass. Both calorimeters have quartz fibers as the active material instead of the conventional scintillating ones.

1.2.8 Forward Multiplicity Detector (FMD)

The purpose of the FMD is to measure $dN/d\eta$ in the rapidity region outside the central acceptance and to provide information for the trigger system in a very short time. The FMD is a silicon detector segmented into seven disks which surround the beam pipe at distances of between ∼42 and 225 cm from the vertex. Together they will cover the pseudorapidity range from -3.4 to -1.7 on the muon arm side and from 1.7 to 5.1 on the opposite hemisphere. It is designed in order to measure charged particle multiplicities from tens (in pp runs) to thousands ($PbPb$ runs) per unit of pseudorapidity.

1.2.9 Photon Multiplicity Detector (PMD)

The PMD is a preshower detector that measures the multiplicity and spatial distribution of photons in order to provide estimates of the transverse electromagnetic energy and the reaction plane. It consists of two identical planes of proportional chambers with a $3X_0$ thick lead converter in between. It will be installed at 350 cm from the interaction point, on the opposite side of the muon spectrometer, covering the region $2.3 \leq \eta \leq 3.5$, in order to minimize the effect of upstream material such as the beam pipe and the structural component of TPC and ITS.

1.2.10 Forward Muon Spectrometer (FMS)

With the forward muon spectrometer it will be possible to study resonances like J/ψ, ψ', Υ, Υ' and Υ'' through their decay into $\mu^+\mu^-$-pairs, and to disentangle them from the continuum given by Drell-Yan processes and semi-leptonic decays of D and B mesons. The study of open heavy flavor production will be interesting too and it will be also accessible through measurements of $e-\mu$-coincidences, where the muon is detected by the muon spectrometer and the electron by the TRD.

A resolution of 70 MeV/c^2 in the 3 GeV/c^2 region is needed to resolve J/ψ and ψ' peaks and of 100 MeV/c^2 in the 10 GeV/c^2 region to separate Υ, Υ' and Υ''.

This detector is located around the beam pipe and covers the pseudorapidity range $-4.0 \leq \eta \leq -2.5$. It consists of a passive front absorber to absorb hadrons

and photons from the interaction vertex. The material must have a small interaction length in order to absorb hadrons and a large radiation length, and thus small Z ($X_0 \propto 1/Z$), in order to reduce multiple scattering of muons.

Muon tracks are reconstructed by tracking chambers consisting of multiwire proportional chambers with cathode pad readout. They are embedded in a magnetic field generated by a dipole magnet located outside the L3 magnet. The dimuon trigger is provided by four layers of RPCs operating in streamer mode located behind the muon filter.

1.2.11 ALICE detector coordinate system

As a conclusion of the detector overview, the officially adopted coordinate system is provided. It is a right-handed orthogonal Cartesian system with the origin at the beam intersection point. The axis are defined as follows:

- x-axis is perpendicular to the mean beam direction, aligned with the local horizontal and pointing to the accelerator center;
- y-axis is perpendicular to the s-axis and to the mean beam direction, pointing upward;
- z-axis is parallel to the mean beam direction.

Hence the positive z-axis is pointing in the direction opposite to the muon spectrometer. The convention is coherent with other LHC experiments and has been changed from the one previously adopted in ALICE.

1.3 The ALICE software framework

The ALICE Offline Project has started developing the software framework in 1998. The decision was taken at the time to build the simulation tool for the Technical Design Reports of the ALICE detector using the OO programming technique and C++ as an implementation language.

This lead to the choice of ROOT as framework and GEANT 3.21 as simulation code. A prototype was quickly built and put in production. The experience with this was positive, and in November 1998 the ALICE Offline project adopted ROOT as the official framework of ALICE Offline.

AliRoot is the name ALICE Offline framework for simulation, reconstruction and analysis. It uses the ROOT system as a foundation on which the framework and all applications are built. Except for large existing libraries, such as GEANT3.21 and Jetset, and some remaining legacy code, this framework is based on the Object Oriented programming paradigm, and it is entirely written in C++.

The ROOT system is now being interfaced with the emerging Grid middleware in general and, in particular, with the ALICE-developed AliEn system.

Along with the AliRoot Grid infrastructure, there's also an *interactive* another method of parallel computing, on which this thesis is also about: the Parallel ROOT Facility (PROOF), which extends the ROOT capability on a *parallel* rather than *distributed* computing platform primarily for large-scale analysis, but for production too.

In this section a description of the main features of the offine framework is carried out.

1.3.1 Overview of the AliRoot Offine framework

The AliRoot design architecture is schematically shown in Figure 1.10. The STEER module provides steering, run management, interface classes, and base classes. The detector code is stored in independent modules that contain the code for simulation and reconstruction while the analysis code is progressively added. Detector response simulation can be performed via different transport codes, the most well-known ones being GEANT3, Fluka (both written in Fortran) and GEANT4 (object-oriented and written in C++).

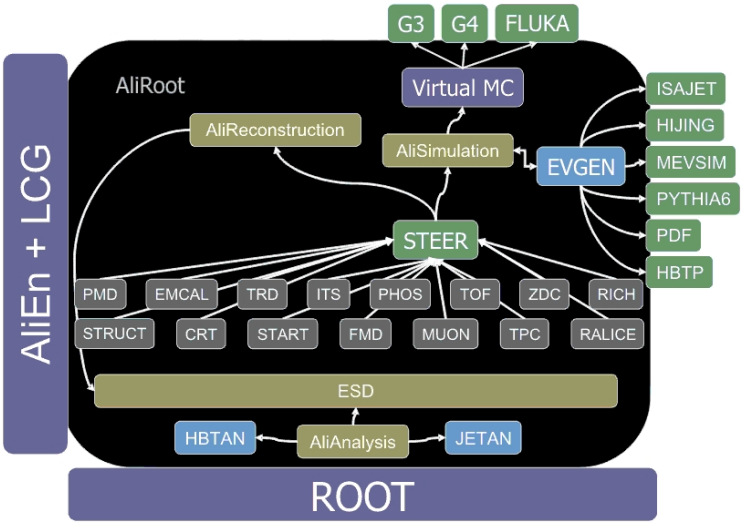


Figure 1.10: The AliRoot design architecture.

1.3.1.1 Simulation

An event generator produces a set of "particles" with their momenta. The set of particles, where one maintains the production history (in form of mother-daughter relationship and production vertex) forms the kinematic tree.

The transport package brings the particles through the set of detectors, and produces hits, which in ALICE terminology means energy deposition at given point. The hits contain also information *(track labels)* about the particles that have generated them. There is one main exception, namely the calorimeter (PHOS and EMCAL) hits, where a hit is the energy deposition in the whole detecting element volume.

This behavior is correct, since inside these detectors the particle is completely stopped. Furthermore in some detectors the energy of the hit is used only for comparison with a given threshold, for example in TOF and ITS pixel layers. These are in fact "digital" detectors in the sense that they are requested only for an On-Off response , depending on the threshold overcoming.

At the next step the detector response is taken into account, and the hits are transformed into digits. As it was explained above, the hits are closely related to the tracks which generated them. The transition from hits/tracks to digits/detectors is shown in Figure 1.11 as the left part of the parabolic path. There are two types of digits: *summable digits*, where one uses low thresholds and the result is additive, and *digits*, where the real thresholds are used, and the result is similar to what one would get in the real data taking. In some sense the summable digits are precursors of the digits. The noise simulation is activated when digits are produced. There are two differences between the digits and the raw data format produced by the detector: firstly, the information about the Monte Carlo particle generating the digit is kept, and secondly, the raw data are stored in binary format as " payload" in a ROOT structure, while the digits are stored in ROOT classes.

Two conversion chains are provided in AliRoot: "hits → summable digits → digits", and "hits → digits". The summable digits are used for the so called *event merging*, where a signal event is embedded in a signal-free underlying event. This technique is widely used in heavy-ion physics and allows to reuse the underlying events with substantial economy of computing resources. Optionally it is possible to perform the conversion "digits → raw data", which is used to estimate the expected data size, to evaluate the high level trigger algorithms, and to carry on the so called computing data challenges.

The whole simulation process, represented in Figure 1.12(a), includes the following steps regardless of the detector response simulation package in use.

- **Event generation of final-state particles.** The collision is simulated by a physics generator code (since they predict different scenarios for the same

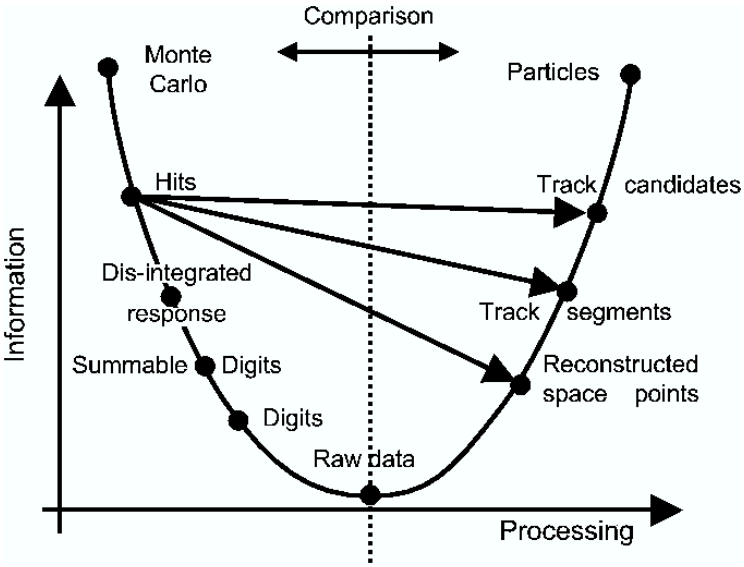


Figure 1.11: The parabolic path of data flow in event generation.

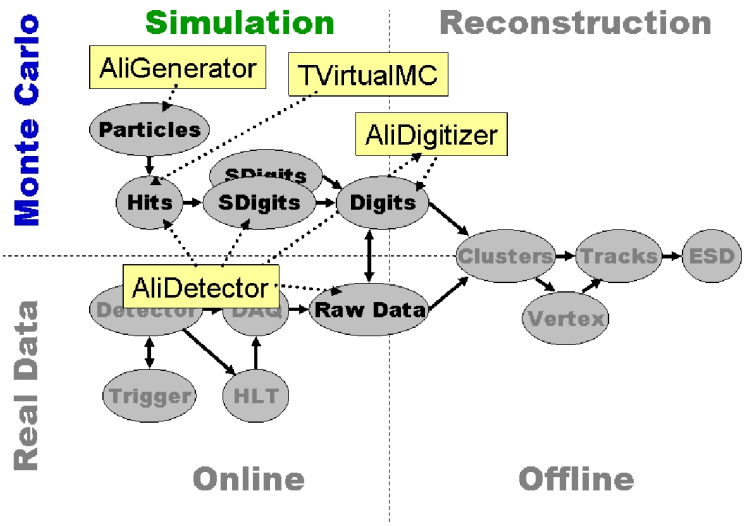
aspect, one has many generators like PHYTIA, HIJING and FLUKA) or a parameterization (with the class AliGenParam) of the kinematical variables and the final-state particles are fed to the transport program.

- **Particle tracking.** The particles emerging from the interaction of the beam particles are transported in the material of the detector, simulating their interaction with it, and the energy deposition that generates the detector response (hits).

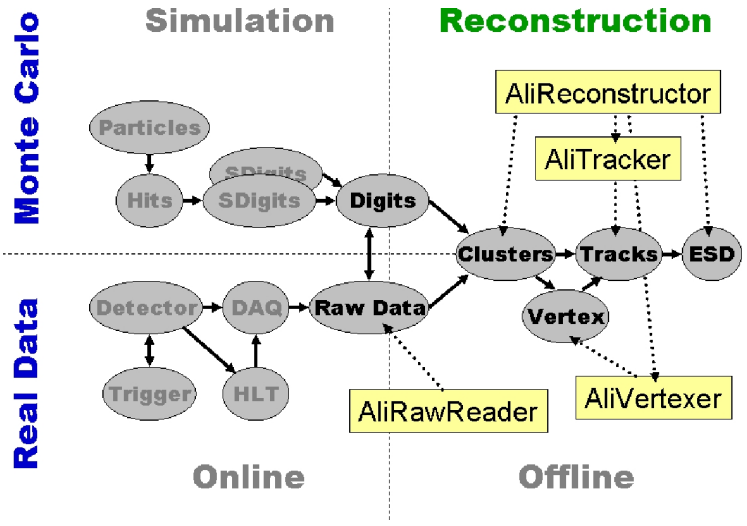
- **Signal generation and detector response.** During this phase the detector response is generated from the energy deposition of the particles traversing it. This is the ideal detector response, before the conversion to digital signal and the formatting of the front-end electronics is applied.

- **Digitization.** The detector response is digitized and formatted according to the output of the front-end electronics and the data acquisition system. The results should resemble closely the real data that will be produced by the detector.

- **Fast simulation.** The detector response is simulated via appropriate parameterizations or other techniques that do not require the full particle transport. The `AliSimulation` class provides a simple user interface to the simulation framework.



(a) Simulation.



(b) Reconstruction.

Figure 1.12: The AliRoot Simulation and Reconstruction frameworks.

1.3.1.2 Reconstruction

Most of the ALICE detectors are tracking detectors. Each charged particle going through them leaves a number of discrete signals that measure the position of the points in space where it has passed. The task of the reconstruction algorithms is to assign these space points to tracks and to reconstruct their kinematics. This operation is called *track finding*.

A good track-finding effciency is required in ALICE for tracks down to $p_T = 100$ MeV/c even at the highest track density, with occupancy of the electronics channels exceeding 40% in the TPC inner rows at the maximum expected track multiplicity. Given this situation, most of the development is done for $PbPb$ central events, since lower multiplicities are considered an easier problem once the high-multiplicity ones can be handled. However, the opposite may be true for some quantities, such as the main vertex position, where a high track multiplicity will help to reduce the statistical error.

The following terms usually describes data at different steps of reconstruction, shown on the right part of Figure 1.11 and highlighted in Figure 1.12(b).

- **RAWS.** This is a digitized signal (ADC count) obtained by a sensitive pad of a detector at a certain time.
- **RECS – reconstructed space point.** This is the estimation of the position where a particle crossed the sensitive element of a detector (often, this is done by calculating the center of gravity of the "cluster").
- **ESD – reconstructed track** This is a set of five parameters (such as the curvature and the angles with respect to the coordinate axes) of the particle's trajectory together with the corresponding covariance matrix estimated at a given point in space. ESDs also contain primary and secondary vertices, pileup information, multiplicities

The input to the reconstruction framework are digits in ROOT `TTree` format or Raw Data format. First a local reconstruction of clusters is performed in each detector. Then vertices and tracks are reconstructed and particle types are identified. The output of the reconstruction is the ESD. The `AliReconstruction` class provides a simple user interface to the reconstruction framework.

1.3.1.3 Fast Simulation

The following description of a fast simulation process refers to the muon spectrometer but the same concepts may as well be applied for central barrel detectors. The high luminosity and the center of mass energy of LHC make it a high statistics "particle factory". The expected number of quarkonium (charmonium

in particular) states detected in the muon spectrometer could be hardly produced and analyzed by full simulations in a reasonable amount of time. Thus a new kind of approach was adopted in heavy quarkonia production: *the fast simulation*.

This technique is based on the parametrization of the response of the muon spectrometer at the single muon level, which allows to considerably reduce the requested computational time.

Given a muon of momentum p, generated at the interaction point at angles θ and ϕ, the fast simulation applies the smearing of the apparatus and gives the reconstructed p', θ' and ϕ', together with the detection probability P_{det} for that muon. This last term is the product of three factors, giving the probability for that muon to satisfy the acceptance (P_{acc}), reconstruction (P_{rec}) and trigger (P_{trig}) requirements.

The first step towards the fast simulation is the so-called *fast reconstruction* of the muon track in the tracking system of the muon spectrometer. This procedure allows to skip the time consuming digitization and clusterization processes. Starting from a sample of muons coming from full simulations, the residual distributions are created and then parametrized by a superposition of two gaussians and a constant.

The residual is defined as $\Delta y = y_{cluster} - y_{hit}$, where $y_{cluster}$ is the impact point coordinate obtained with the cluster reconstruction, while the y_{hit} is the generated hit coordinate.

Parametrizations obtained can be applied to reconstruct the Υ and J/ψ invariant mass spectra with the proper p_T cut. The process still needs the creation of hits, but the skipping of digitization and clusterization leads to a considerable speed gain.

The second step consists in the elimination of the hits creation phase. The objective is actually to direct smear the kinematic variables for each single muon, passing from generation to detector response without any intermediation. In order to obtain this result it is first necessary to parametrize the experimental resolution on the kinematical variables of the muons ($\Delta p = p_{rec} - p_{gen}$, $\Delta\theta = \theta_{rec} - \theta_{gen}$, $\Delta\phi = \phi_{rec} - \phi_{gen}$), together with the acceptance and efficiency in several (p, θ, ϕ) intervals. To this end 3D grids have been prepared *(Look Up Tables)* in which parameters for the acceptance and for the reconstruction and trigger efficiencies are stored.

Comparison with full simulation shows a very good agreement in the region of $p > 8$ GeV/c, but some discrepancies are present at very low momenta. The phase space portion with $p < 8$ GeV/c is quite peculiar, showing steep variations due to the fast rise of acceptance and efficiency. In any case the accepted muons is about the same for full and fast simulation, even in the problematic region.

1.4 Computing Model in the ALICE Experiment

1.4.1 ALICE computing needs

The investment for LHC computing is massive, not only from the computational point of view, but expecially from the storage point of view. For ALICE only we foresee[8]:

- data acquisition rate of 1.25 GiB/s from ion-ion collisions;
- approximately 5 PiB of data stored on tape each year;
- at least 1.5 PB of data permanently online on disks;
- an amount of CPU power estimated at the equivalent of 25 000 personal computers of the year 2003.

The whole resources needed lead to a total cost estimation between 10 and 15 million Euros.

The need for big storages clearly points out that the ALICE computing model must be developed around data, not around computing power: in other words, it has to be *data-centric*. This means that, with a huge amount of data to store and analyze like that, developers must take into account any barriers between the program and the data (e.g., the network infrastructures) and remove it by making the program close to the data and not vice-versa.

Both the cost and the requirements represent a new challenge from the computational point of view, representing a milestone for the whole Computer Science, not only Physics. The preparation of LHC coincided with the rising of a new, distributed computing model called "the Grid", which seems to meet the requirements to manage such a big computing power and storage capacity.

The following sections analyze the so-called Grid[2] distributed computing model and another model of processing data called interactive (or chaotic) analysis, which is indeed the computing model on which this thesis is primarily about.

1.4.2 Distributed computing: the ALICE Grid

As we have seen in § 1.4.1, a large amount of both computing and storage resources is necessary to process and store the data generated by each LHC experiment, including ALICE.

[2]It is a common mistake to consider "Grid" an acronym, although it is not: the name has been choosen in analogy with the "power grid", but the power supplied by the Grid is actually *computing power*.

Since the conceptual design of the LHC experimental programme, it was recognized that, in analogy with the fact that "human resources" (i.e., the physicists) aren't used to work in a single place but they are naturally distributed instead, this time the required data processing and storage resources cannot be consolidated at a single computing centre too.

There are many reasons why to distribute HEP computing facilities through the institutes and universities participating in the experiment.

- First of all, there's a technical reason; concentrating such a huge amount of computing resources in a single place it would be physically impossible.
- As of its nature, CERN and LHC investors are widespread all over the world: it is likely that funding agencies will prefer to provide computing resources locally in their own country.
- Moreover, many physicists work in countries where there's less financial support to the LHC experiments: these physicists should be able to access computational resources though, in order to limit the so-called *digital divide*.

The technical side of the decentralized offline computing scenario has been formalized in the so-called MONARC model[3] schematically shown in Figure 1.13. MONARC describes an assembly of distributed computing resources, concentrated in a hierarchy of centres called Tiers, where Tier-0 is CERN, Tier-1s are the major computing centres which provide a safe data storage (thus providing a natural and distributed backup system through replication), likely in the form of a MSS, Tier-2s are smaller regional computing centres. The MONARC model also foresees Tier-3s which are university departmental computing centres and Tier-4s that are user workstations; however, the important Tiers in our discussion are the Tier-1s and Tier-2s (Torino's site is among them): Tier-0 has a "supervision" role, while the two lowest Tiers are not relevant in current Grid deployments.

The major difference between the first three Tiers is the Quality of Service (QoS) and reliability of the computing resources at every level, where the highest QoS is offered by the Tier 0/Tier 1 centres.

The basic principle underlying the ALICE computing model is that every physicist should have equal access to the data and the computing resources necessary for its processing and analysis. Thus, the resulting system will be very complex with hundreds of components at each site with several tens of sites. A large number of tasks will have to be performed in parallel, some of them following an ordered schedule. Assignment of these tasks based on their nature and resources requirements follow.

[3] `http://www.cern.ch/MONARC/`

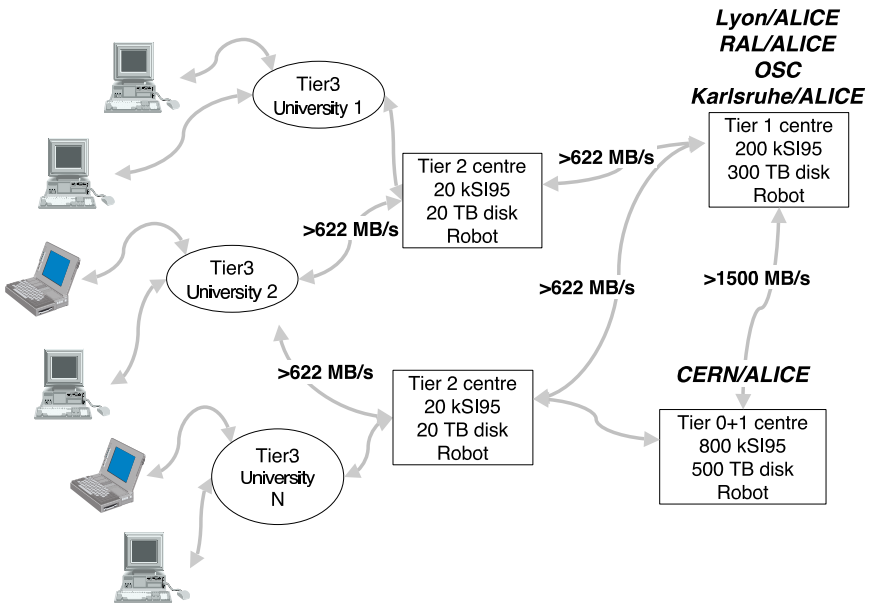


Figure 1.13: The MONARC model.

- The ALICE computing model foresees that one copy of the raw data from the experiment will be stored at CERN (Tier-0) and a second copy will be distributed among the external (i.e. not at CERN) Tier-1 centres, thus providing, as we have already stated, a natural backup. Reconstruction to the ESD level will be shared by the Tier-1 centres, with the CERN Tier-0 responsible for the first reconstruction pass.

- Subsequent data reduction to the AOD level, analysis and Monte-Carlo production will be a collective operation where all Tier-1 and 2 centres will participate. The Tier-1s will perform reconstruction and scheduled analysis; the Tier-2s will perform single-user Monte-Carlo and end-user analysis instead. Tier-2 tasks are largely unpredictable, and they don't meet the QoS and resources requirements of higher Tiers.

This kind of tiered structure of computing and storage resources has to be transparent to the physicists in order to be used efficiently: physicists should have to deal with this whole structure just like it was a single entity.

The commonality of distributed resources management is being realized under the currently ongoing development of the Grid[14]. It was conceived to facilitate the development of new applications based on high-speed coupling of people, computers, databases, instruments, and other computing resources by allowing

"dependable, consistent, pervasive and inexpensive access to high-end resources", as originally stated as main features of the Grid by Foster and Kesselman[13].

We can foresee an evolution of the MONARC model to a *tier-free model*, where that transparent single entity can also be called a single "cloud", and we refer to that model as *cloud computing*: in a well-functioning cloud computing environment the distribution of tasks to the various computing centres would be performed dynamically, based on the availability of resources and the services that they advertise, irrespective of the Tier level division of tasks based on QoS and resources requirements.

1.4.2.1 AliEn: the ALICE Grid user interface

AliEn[4] has been primarily conceived as the ALICE user interface into the Grid world, and it is actually an interface that runs on other real Grid middlewares. The Grid is a relatively new concept from the implementation point of view, so there isn't a standard middleware that can be considered definitive and stable: AliEn's aim is to provide, from the Grid point of view, a uniform abstraction layer that remains standard for ALICE users and developers even when the underlying real middleware eventually changes; from the Pyhsicists point of view AliEn is an interface that makes ALICE resources access transparent, shielding the users from the underlying Grid complexity and heterogeneity.

As new middleware becomes available, we shall interface it with AliEn, evaluating its performance and functionality. Our final objective is to reduce the size of the AliEn code, integrating more and more high-level components from the Grid middleware, while preserving its user environment and possibly enhancing its functionality. If this is found to be satisfactory, then we can progressively remove AliEn code in favour of standard middleware. In particular, it is our intention to make AliEn services compatible with the Open Grid Services Architecture (OGSA) that has been proposed as a common foundation for future Grids. We would be satisfied if, in the end, AliEn would remain as the ALICE interface into the Grid middleware. This would preserve the user investment in AliEn and, at the same time, allow ALICE to benefit from the latest advances in Grid technology. This approach also has the advantage that the middleware will be tested in an existing production environment.

Through interfaces, it can use transparently resources of different Grids developed and deployed by other groups: in other words, not only is AliEn capable of providing a layer of abstraction between the middleware and the user, but it is even capable of enabling different middlewares (that speak different languages) to cooperate through that layer of abstraction: in short words, AliEn can in principle be seen as a federation of different and collaborating Grids (Figure 1.14). In the future, this cross-Grid functionality will be extended to cover other Grid

flavours.

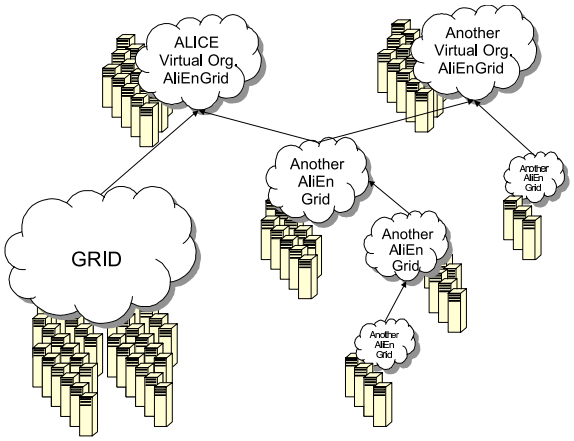


Figure 1.14: The ALICE Grid as a federation of collaborative Grids.

The system is built around Open Source components and uses a Web Services model[4] and standard network protocols. Less than 5% is native AliEn code (mostly code in PERL), while the rest of the code has been imported in the form of Open Source packages and modules.

Web Services play the central role in enabling AliEn as a distributed computing environment. The user interacts with them by exchanging SOAP messages and they constantly exchange messages between themselves behaving like a true Web of collaborating services. AliEn consists of the following components and services:

- authentication, authorization and auditing services;
- workload and data management systems;
- file and metadata catalogues;
- the information service;
- Grid and job monitoring services;
- a SE and a CE for each Grid site, that serve the WNs that actually run jobs.

A schematic view of the AliEn services, their location and interaction with the native services at the computing centres is presented in Figure 1.15.

[4] `http://www.w3.org/2002/ws/`

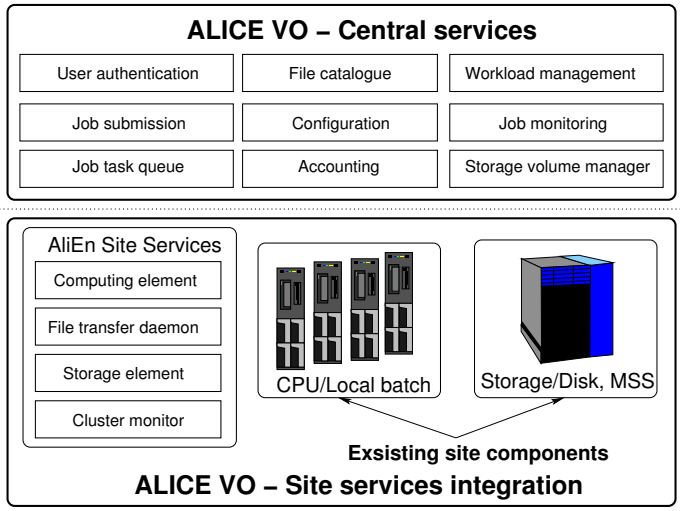


Figure 1.15: Schematic view of the AliEn basic components and deployment principles.

The AliEn workload management system is based on the so-called "pull" approach. A service manages a common TQ, which holds all the jobs of the ALICE VO. On each site providing resources for the ALICE VO, CE services act as "remote queues" giving access to computational resources that can range from a single machine, dedicated to running a specific task, to a cluster of computers in a computing centre, or even an entire foreign Grid. When jobs are submitted, they are sent to the central queue. The workload manager optimizes the queue taking into account job requirements such as the physical location of the SEs that keep needed input files, the CPU time and the architecture requested, the disk space request and the user and group quotas (Figure 1.16); it then makes jobs eligible to run on one or more computing elements. The CEs of the active nodes get jobs from the central queue and deliver them to the remote queues to start their execution. The queue system monitors the job progress and has access to the standard output and standard error.

Input and output associated with any job are registered in the AliEn FC, a virtual file system in which logical names, with a semantics similar to the Unix file system, are assigned to files. Unlike real file systems, the FC does not own the files; it only keeps an association between one or possibly more LFNs and (possibly more than one) PFNs on a real file or MSS. The correspondance is kept via the GUID stored in the FC. The FC supports file replication and caching and it provides the information about file location to the RB when it comes to scheduling jobs for execution. These features are of particular importance, since

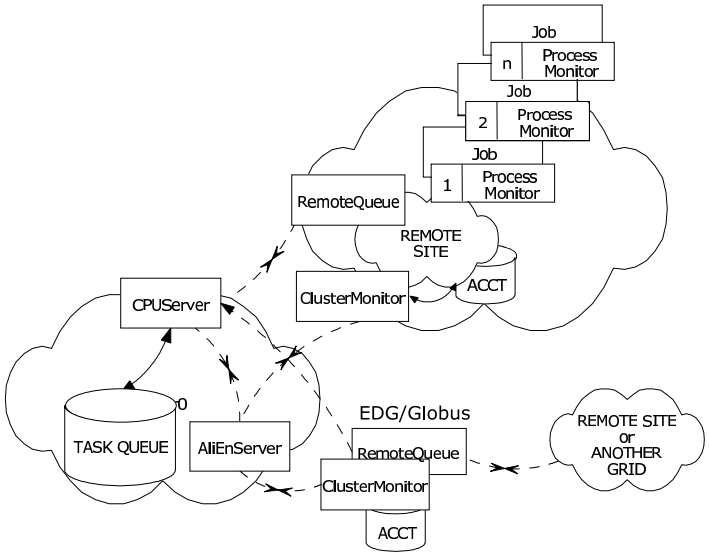


Figure 1.16: The AliEn workload management.

similar types of data will be stored at many different locations and the necessary data replication is assumed to be provided transparently and automatically by the Grid middleware. The AliEn file system associates metadata with LFNs.

The file catalogue is not meant to support only regular files – the file system paradigm was extended and includes information about running processes in the system (in analogy with the `/proc` directory on most Unices). Each job sent to AliEn for execution gets an unique id and a corresponding `/proc/id` directory where it can register temporary files, standard input and output, as well as all job products. In a typical production scenario, only after a separate process has verified the output, will the job products be renamed and registered in their final destination in the file catalogue. The entries (LFNs) in the AliEn file catalogue have an immutable unique file id attribute that is required to support long references (for instance in ROOT) and symbolic links.

The hierarchy of files and directories in the AliEn file catalogue reflects the structure of the underlying database tables. In the simplest and default case, a new table is associated with each directory. In analogy to a file system, the directory table can contain entries that represent the files or again subdirectories. With this internal structure, it is possible to attach to a given directory table an arbitrary number of additional tables, each one having a different structure and possibly different access rights while containing metadata information that further describes the content of files in a given directory. This scheme is highly granular and allows fine access control. Moreover, if similar files are always cata-

logued together in the same directory this can substantially reduce the amount of metadata that needs to be stored in the database. While having to search over a potentially large number of tables may seem ineffective, the overall search scope has been greatly reduced by using the file system hierarchy paradigm and, if data are sensibly clustered and directories are spread over multiple database servers, we could even execute searches in parallel and effectively gain performance while assuring scalability.

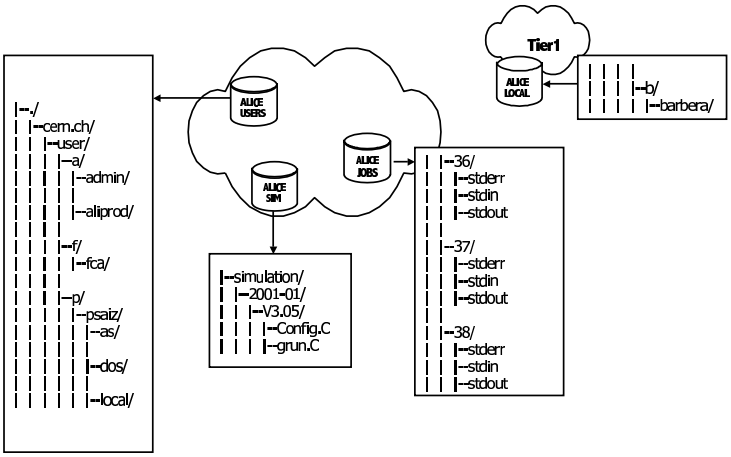


Figure 1.17: The AliEn file catalogue, showing how the virtual AliEn filesystem can also contain real-time information about running jobs, just like the `/proc` filesystem of many Unices.

The Grid user data analysis has been tested in a limited scope using tools developed in the context of the ARDA project[5] (the aliensh interface and the analysis tools based on it). Two approaches were prototyped and demonstrated so far: the *asynchronous* (interactive batch approach) and the *synchronous* (true interactive) analysis. Both of these two approaches are integrated with AliEn: while the first one gives satisfactory results and it is currently used in production, it seems that interactive analysis can not be integrated gracefully with AliEn on small (say, Tier-2) centres, by violating somehow the foreseen AliEn evolution towards a cloud computing model (that is a "tierless" model, as we have already noted).

The asynchronous model has been realized by extending the ROOT functionality to make it Grid-aware. As the first step, the analysis framework has to extract a subset of the datasets from the file catalogue using metadata conditions

[5] `http://lcg.web.cern.ch/LCG/peb/arda/Default.htm`

provided by the user. The next part is the splitting of the tasks according to the location of datasets.

Once the distribution is decided, the analysis framework splits the job into sub-jobs and inserts them in the AliEn queue with precise job descriptions. These are submitted to the local CEs for execution. Upon completion, the results from all sub-jobs are collected, merged and delivered to the user.

A deeper insight on interactive analysis (both AliEn-based and not) will be presented on § 1.4.3.

1.4.3 Interactive analysis: PROOF

The need for interactive analysis arose from tasks that require too much computing power to be run on a local desktop computer or laptop, but that are not meant to be run on the Grid.

Referring to § 1.4.2, let's consider tasks meant to be run on Tier-2s: these are the sites where user analysis runs, along with single-user Monte-Carlo. These tasks generally require many runs in order for physicists to fine-tune analysis or Monte-Carlo parameters. Moreover, an user analysis that needs fine-tuning is usually under heavy development and thus, with many bugs, that can even affect system stability: in other words, this kind of user analysis does not meet the QoS standards in order to be run on the Grid.

Because of fine-tuning and debugging, these tasks show unpredictable computing power and memory usage, along with unexpected behaviors: for these reasons we globally refer to these tasks with the expression "chaotic analysis".

In order for physicists to do chaotic analysis a special facility needs to be developed. This facility will consist on several computers on which the job is split by a transparent user interface, which should give access to the computing resources via the tools physicists are accustomed to and which should hide the implementation (the fact that the job is actually split onto several machines should be done automatically without user intervention).

From these ideas, Parallel ROOT Facility (PROOF) was born. PROOF[5] is an extension of the well-known ROOT system that allows the easy and transparent analysis of large sets of ROOT files in parallel on remote computer clusters. PROOF functionality can be accessed directly from within ROOT: this is why PROOF is transparent for the user.

Besides from transparency, the other main PROOF design goals are scalability and adaptability. *Scalability* means that the basic architecture should not put any implicit limitations on the number of computers that can be used in parallel, and should easily grow with the minimum effort for the system administrator, and with no effort at all for the end user. With *adaptability* we mean that the system should be able to adapt itself to variations in the remote environment (changing

load on the cluster nodes, network interruptions, etc.).

Being an extension of the ROOT system, PROOF is designed to work on objects in ROOT data stores. These objects can be individually *keyed* objects as well as `TTree` based object collections. By logically grouping many ROOT files into a single object, very large data sets can be created. In a local cluster environment these data files can be distributed over the disks of the cluster nodes or made available via a NAS or SAN solution.

PROOF adaptability means that it can be, in principle, used in two different multi-core and multi-machine architectures: distributed computing and parallel computing.

- With **parallel computing** we want a single task to be split on several machines that are aggregated *locally*. Because of the aggregation, the system administrator has the direct control over each PROOF machine and can adapt configurations in order to create a homogeneous system. Since PROOF is meant to be *interactive*, having the workers aggregated simplifies the process of jobs synchronization between the different workers.

- With **distributed computing** we basically mean the Grid model: tasks run on resources that are not under control of a single system administrator, and they are distributed around the world. PROOF is designed to be fault-tolerant, by making transparent for the user the fact that the job is running on local machines or on distributed, distant machines. In such an environment the processing may take longer (in this case the term "interactive" may not be appropriate), but the user will still be presented with a single result, like the processing was done locally. Doing distributed computing with PROOF can be a tricky task, because job synchronization is difficult to be accomplished on distant, inhomogeneous machines, thus requiring an external interface (AliEn, as we will see).

1.4.3.1 PROOF for parallel computing

The typical and most stable use of PROOF is for parallel computing on local facilities, placed close to the physicists that do chaotic analysis. This configuration requires three different kinds of machines.

- **The slaves:** these are the machines that run the job. Each slave is a machine which can hold several *workers*, each worker being a daemon that actually does the computation. On a typical slave configuration we have a worker per each core. In addition, slaves share part of their disk space in a single "pool" to hold data that needs to be analyzed: in other words, the PROOF pool can be seen as a single big Network-Attached Storage (NAS)

which is indeed the sum of the disks of every machine that acts as a PROOF slave.

- **The master:** this server has the role of coordinator. When physicists want to do something on PROOF they simply connects to the master. Since the computing model is data-centric in order to minimize network transfers, the master sends the jobs as close as possible to the machines that hold the data. The best situation occurs when the job runs on a slave that is also the one which has the data to process, since in this case the transfer speed bottleneck is the drive speed, not the network speed.

- **The client:** this is the user's machine running ROOT that connects to the master.

A schematic representation of this structure and network configuration is represented in Figure 1.18 along with possible data transfers that may occur between the machines (mainly slaves).

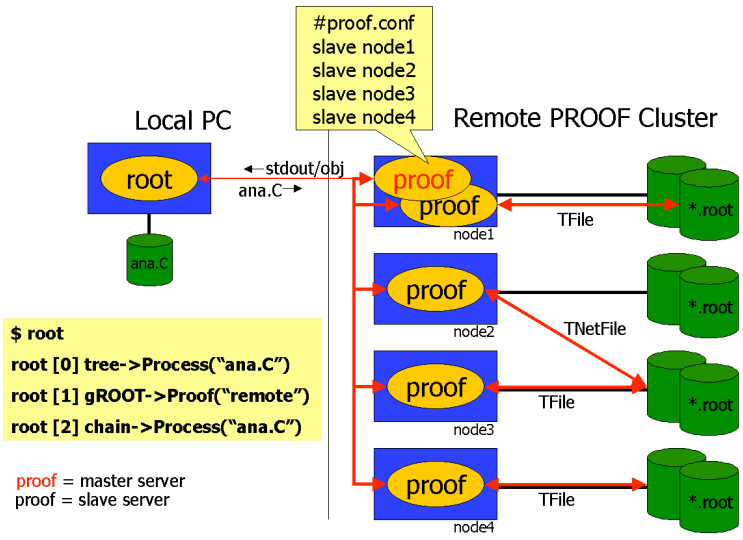


Figure 1.18: Conventional setup of a PROOF farm, also showing machines involved in data transfers.

1.4.3.2 PROOF for distributed computing

A distributed PROOF environment can be seen as an extension of the parallel model, that can in principle be achieved thanks to the functionalities of both

PROOF and AliEn.

The PROOF feature that should qualify it suitable for a distributed, inhomogeneous computing is its *multi-layered architecture*. The multi-layered structure is very similar to the MONARC model: in PROOF terminology, we have on each tier masters that depend on a master of an upper level. Each sub-master is transparently seen by the super-master as a worker: the super-masters distribute jobs through the sub-masters, which distribute jobs to the workers (although there may be several other levels). When a worker finishes its job, the results are gradually merged by the masters until a single file is transparently presented by the super-master to the physicist. The concept of a master being also a worker is illustrated in Figure 1.19.

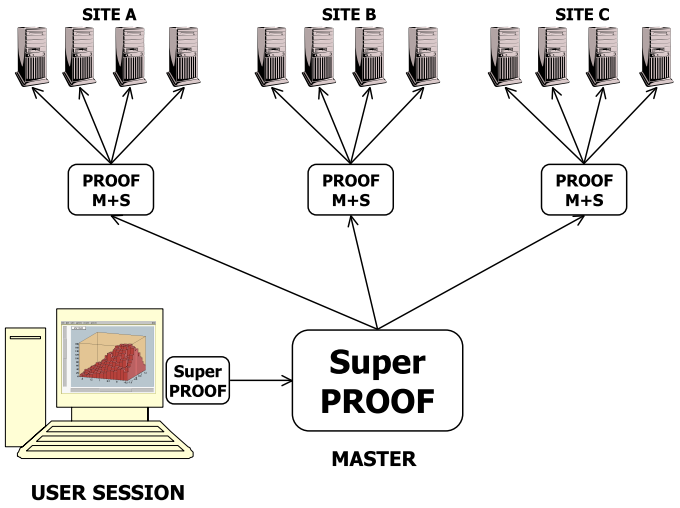


Figure 1.19: Multi-layered setup of a distributed PROOF farm, where each master is seen by a worker by its superior.

However, this solution requires a much tighter integration between ROOT and the Grid services, where the framework should be able to execute in parallel and in real-time all sub-jobs associated to the main user job. In addition, the system should automatically scale the number of running processes to the amount of available resources at the time of execution. All these features are indeed AliEn's features, and by integrating AliEn in PROOF this task can be accomplished. This will allow for an efficient planning of critical analysis tasks, where the predictability of the execution time is very important. As such it is an essential building block of the ALICE computing model.

However a distributed PROOF environment has a number of drawbacks. First

of all, computing resources are limited, so that interactive analysis should share resources with the Grid without meeting the required QoS – interactive analysis is chaotic and does not mix well by conflicting with predictable, non-chaotic Grid jobs. Moreover, as we have already noted, a multi-layered architecture is incompatible with a cloud computing model.

The need for distributing PROOF resources (rather than parallelizing with dedicated local servers) arises because Tier-2s can not in general afford the cost of a dedicated facility and should find a way to share resources with the Grid. The integration between AliEn and PROOF seems the most straightforward solution (because we let AliEn, that already knows how resources are distributed and used, do the load balancing); there's another solution though that permits us to share resources with the Grid on Tier-2s without badly interfering with the jobs and without distributing (but just parallelizing) interactive analysis: the idea of the Virtual Analysis Facility (VAF), that will be explored, discussed, prototyped and benchmarked in the following chapters of this thesis.

Chapter 2

Feasibility of a Virtual Analysis Facility

2.1 Ideas behind the Virtual Analysis Facility

As we have already discussed at the end of the previous chapter, with the beginning of real data acquisition from LHC experiments the need for an interactive data analysis facility which gives great CPU power to physicists without any delay is constantly increasing to cover for CPU needs of rapid turn-around activities.

As the kind of tasks meant to be run on an interactive facility are "bleeding-edge tasks" (debugging code, optimizing calibration cuts, and so on) they are likely to be unstable and to cause instability to the system they are running on: for this reason it would be better to keep the so-called *interactive user analysis* physically close to the physicist, as technical support from the facility system administrator would be faster and more effective.

Other points for keeping this kind of analysis local are *facility scalability* and *customization*: an interactive facility close to physicists also means that system administrators can fine-tune the facility features in order to match the exact physicists requests as much as possible.

These are the reasons why such facilities are becoming more and more popular outside CERN: CERN Analysis Facility (CAF) is just the first example of a PROOF facility for interactive analysis, but its dimensions are unlikely to satisfy the need of every ALICE user in the world. Since it is clear and commonly understood the need for a federal model of interactive analysis data facilities, we can skip to the next point: CPU cores and memories must be dedicated in *delocalized centres*, which can not always afford the cost of buying new multi-core machines

and maintain them.

Even in the case an institute can afford the cost for new machines and the corresponding ancillary resources (network infrastructures, system administrators, racks, storage, and so on) we should note that a local analysis facility is a waste of resources: intrinsically, an *interactive* analysis facility is used only when requested, so that resources are idle most of the time. By considering that these facilities are and will be localized there are particular times when physicists are supposed to sleep or take a rest, so that we expect the CPUs to be mostly idle during the night and the weekends. This discussion cannot obviously be extended to the case the facility is used by many people all around the world that work in many different timezones, such as the CAF (that is an exception to this general rule).

For these reasons alternatives to buying new machines had to be developed: if buying new machines is unfeasible, computing centres should use the existing machines. If the existing machines are Grid-dedicated, resources must be shared somehow between the Grid and the interactive analysis by considering the following guidelines.

- **Promptness.** When the physicist log into the system and asks for computational power, it should be given immediately: this means that resources are yielded to the physicist with an acceptable delay (i.e., in less than 5 minutes).

- **Transparency for the physicist.** physicists should not need to know that the resources they are using are shared with the Grid: interactive jobs should run on the facility with no or little modification, and they should be able in principle to run some analysis the same way they do on a dedicated facility, without the need to explicitly ask for resources. Physicists should also not need to know that they are actually running their tasks on several computers: this layer of astraction is provided by PROOF and AliRoot, which requires *no* code changes with respect to local analysis, and only minimal procedure changes.

- **Transparency for the Grid.** Grid jobs should not need to know that they are sharing computing resources with interactive analysis tasks: Grid jobs should run unmodified and shouldn't notice the presence of concurrent interactive tasks, apart from a temporary and unavoidable slow down.

- **Sandboxing.** Interactive jobs are error-prone and PROOF does not provide enough sandboxing apart from different machine privileges for each user, because machines dedicated to interactive analysis are meant for trial-and-error use cases. However, in a production Grid environment that must

coexist with PROOF, sandboxing becomes a critical point: we definitely do not want PROOF jobs errors to propagate to the whole system and affect the performance (and in the worst case, the results) of Grid jobs. For this reason we need sandboxing, that in our case means performance and failures insulation between Grid and PROOF.

2.1.1 Making Grid and interactive analysis coexist

An overwiev of possible different solutions is discussed in this section by taking into account the above guidelines.

2.1.1.1 Solutions for Tier-1s

If we want to make Grid and interactive jobs coexist on huge centres such as Tier-1s we can use three different approaches (a static one, a dynamic one and a semi-static one).

The *static* approach is to simply take out some machines from the Grid and dedicate them to interactive analysis: this is the CAF approach adopted at CERN.

The *semi-static* one is pretty naïf and consists in configuring each machine with dual-boot capabilities and reboot it with the proper configuration when needed. This can be done on a Tier-1 because of the large number of cores[1] ($\approx$1000) and thus a large rate of finishing jobs: as all jobs on target machines terminate, the machines are rebooted into a PROOF configuration. Even if PROOF failures do not affect the Grid, there is a drawback: the delay between user PROOF connection and effective availability of PROOF makes this approach non-transparent and non-prompt for the user.

The *dynamic* approach consists in putting interactive daemons as high priority Grid jobs in the queue: as before, on a large site jobs finish continously, so that free slots are occupied on demand by PROOF workers. This approach differs from the semi-static one because there is no need to reboot machines, and, as a consequence, there is no need to wait for *all* jobs on each machine to terminate before rebooting. This approach is absolutely feasible and already adopted at GSI[16, 17].

In principle jobs priority can be adjusted by common Unix scheduler commands, and performance insulation can be obtained by physically separating local disks assigned either to Grid or PROOF. It remains, though, the problem of *memory allocation*: with standard Unix kernels there is no possibility to choose which is the maximum amount of memory assigned to a single process so if an interactive task becomes uncontrollable and takes up all the memory it starts swapping

[1] We refer to *cores* and not *CPUs* because each CPU can have different number of cores.

(a) Static LCG facility.

(b) Dynamic VAF with variable resources.

Figure 2.1: Two schemes representing a standard Grid facility compared to a VAF where both WN and the PROOF slave are installed on each physical machine and they share the same resources, heterogeneously distributed by request. SEs are not represented in these schemes.

and forces other Grid jobs to swap too; no perfect sandboxing is possible in this case.

2.1.1.2 A solution for Tier-2s

As already mentioned, Tier-2 centres are smaller (≈100 cores) and more common than Tier-1s: these are the centres where user analysis occurs.

When we look for an approach to allocate resources to interactive analysis without affecting the Grid in Tier-2s we are strongly limited by the small number of cores: any static approach is unfeasible on such centres, and the dynamic approaches described for Tier-1s can not work here, as the rate of finishing jobs is too low, introducing unacceptable delays (plus the time taken to reboot each machine).

The computing centre in Torino is a Tier-2 centre for the ALICE experiment with the need for interactive analysis but with not enough resources to dedicate: this thesis is about the conception and the implementation of a VAF by using a virtual machine for both Grid WN and PROOF slave on each same physical machine in order to provide better sandboxing and better resource assignment (not only CPU priority but *memory* too, as we are about to see) without requiring any guest software modification. The VAF turns a typical AliEn Grid (§ 1.4.2.1) from a static one into a dynamic one, as represented in Figure 2.1.

2.1.2 Virtualization

Every time we talk about virtualization in computing we are referring to a layer of astraction between different computer resources (either hardware or software, or both), or between a computer resource and the user. A simple example comes from the AliEn File Catalogue (FC) in the ALICE experiment, where a Logical File Name (LFN) is a virtual filename that points to one or more Physical File Names (PFNs) that actually refer to the real file. The LFN is *virtual* because it does not actually exist but it provides a layer of abstraction between the file and the user by hiding the exact file location to the latter.

The kind of virtualization needed by the VAF is called *platform virtualization*: the *guest* operating system is installed onto a machine with virtual hardware resources, not necessarily existing in the real world. A machine made up of virtual resources is called a *virtual machine*. For example, two big files on the same physical disk can be seen by the virtual machine as two different virtual disks.

Platform virtualization requires a Virtual Machine Monitor (VMM) to be installed on the physical machine: this is a piece of code (typically either a user-space program or part of the host kernel) that coordinates the access to the resources seen by each virtual machine. This is also called the *hypervisor*.

2.1.2.1 Reasons behind platform virtualization

There are many reasons why one may want to use virtualization software.

- **Consistency.** In a scenario where there must be many servers providing different services it can be a waste of resources to use different machines for each one, but unfeasible to keep all the services to the same host because of software consistency. Let's suppose that we must configure a web server that needs, for compatibility reasons, an old version of certain libraries that conflict with newer versions, unfortunately needed by the bleeding-edge mail server we want to run on the same machine. By running two different *virtualized* operating systems on the same physical machine we can have both services up with only one machine by separately preserving software consistency.

- **Eased inspection.** A virtual machine can be inspected more easily than a physical one, because all virtual hardware is quite the same, and because in case of unrecoverable disasters the virtual machine can be rebooted or recreated without physically touching a single piece of hardware, enabling technicians to do better remote support.

- **Suspension.** Since the hypervisor controls each virtual machine it knows every time the state of each single piece of virtual hardware: a virtual ma-

chine can be "frozen" (suspended) by saving the state of the operating system along with the state of the whole virtual hardware: this becomes very easy, since virtual hardware is indeed a piece of software.

- **Portability and migration.** The guest operating system deals with virtual hardware, and the virtual hardware is supposed to be the same on each physical machine running the same hypervisor. It is then possible to install an operating system on a virtual machine and clone it onto another physical machine, or, using the suspension feature, it is even possible to easily migrate a running virtual machine to a new physical machine.

- **Platform emulation.** Virtual hardware means *emulated* hardware: we can run operating systems designed for CPU architectures different from the physical one. Emulating the CPU usually introduces a big overhead that drastically drops performance. This overhead is almost not noticeable if the current physical CPU is much faster than the CPU we are emulating, as it is the case in the most common application of platform emulation: running old game consoles on modern domestic personal computers.

- **Resource access control.** The hypervisor can decide what (and which amount of) physical hardware resources can be seen as virtual resources by the guest operating system. Thus, the hypervisor can slow down a given virtual machine by silently eroding resources currently assigned to it, or it can deny the access to a specific piece of hardware by a certain guest operating system: this is, for instance, the case of the famous Playstation 3 game console, that can run another operating system in addition to its default one, but only on top of an hypervisor that inhibites, for instance, the usage of video acceleration to decode Blu-Ray discs in order to prevent piracy.

- **High-Availability (HA) and disaster recovery.** Virtual resources never fail, but real hardware does. It is possible to use migration to move a virtual machine running on defective hardware to a working one. It is even possible, by using hypervisor's resource access control, to run every service in a datacenter on a so-called "computing cloud" where the system administrator needn't worry about the exact correspondance between physical and virtual machines: if a physical machine fails, the cloud should be able to squeeze VMs running on healthy hardware to make room for the migration of VMs running on defective hardware. This is how behaves a commercial solution like the well-known VMware Infrastructure.

2.1.2.2 Full virtualization and paravirtualization

Platform virtualization has been presented so far as something that can be achieve without the guest operating system being "conscious" of virtualization, i.e.: the guest operating system should see its assigned virtual resources without knowing they are actually not real, and should behave just like running on real hardware.

However, there are many performance-related issues that arise when putting additional layers of abstraction between the guest operating system and the physical pieces of hardware: this is when the difference between *full virtualization* and *paravirtualization* becomes important.

In 1974, Popek and Goldberg[22] formally classified the instruction sets of an architecture into two categories and defined the three goals for a virtual machine architecture.

Generally speaking, each processor's instruction can be inserted into one of the following two categories.

1. **Privileged instructions**, which can cause a trap when executed in user mode: in a trap, the program we are running in *user mode* raises an instruction captured by the processor that switches the execution in the so-called *system (or kernel) mode*, losing the control of execution and yielding it to the operating system.

2. **Sensitive instructions**, which change the underlying resources *(control sensitive)* or strictly depend on underlying resources variation *(behavior sensitive)*.

According to Popek and Goldberg the virtualization process should match three goals.

1. **Equivalence.** The VM should behave just like ordinary hardware.

2. **Resource control.** The hypervisor (or VMM) should be in complete control of any virtualized resource: it should always be aware of any access to the resources and it should be the one and only which decides how and when assign resources to the VMs.

3. **Efficiency.** The hypervisor should be involved as little as possible while executing user space code (our ordinary programs), i.e.: most instructions should be passed as-is to the physical cores without intermediation.

They also demonstrated that virtualization of an architecture is always possible when its sensitive instructions are a subset of the privileged ones: this basically means that any instruction that causes a change on the underlying resources

should cause a trap, in order to make the VMM aware of the changes and finally in order to satisfy resource control by the VMM.

If every architecture was developed according to Popek and Goldberg's statements we wouldn't have to worry about the efficiency of our code. Unfortunately, the architecture we are going to emulate (x86 architecture, also known as IA-32, by Intel) does not match the required statements: there are many IA-32 instructions that are *sensitive* but *unprivileged* (we call them *critical instructions*), making very difficult for the VMM to be aware of resources changing without significantly slowing down code execution (no more efficiency) and exposing the fact that resources are virtualized to the guest OS (no more equivalence): in short, critical instructions do violate each statement.

Since IA-32 is the most widespread architecture in the world, various technologies were developed and improved in order to deal with critical instructions.

The first approach is to capture every single critical instruction and dynamically rewrite it by using privileged instructions. However this approach requires the code to be constantly monitored, and memory too, because when the guest operating system tries to read a page where the code was dynamically rewritten the VMM should provide the original, non-rewritten memory page. This approach dramatically slows down code execution.

The second approach is to act at hardware level: every sufficiently modern IA-32 CPU has some virtualization extensions that keep the virtual machine state inside the CPU, so that "code rewriting" is done in the latter's registers at hardware level, and software hypervisor is not aware of that. Intel released this technology as Intel VT-x[21], while AMD called it AMD-V[1]. These technologies act in fact as an hardware acceleration for the software hypervisor. However, for instructions that actually affect the system (such as I/O and memory changes), no hardware acceleration is possible and the software hypervisor must be notified of that, so a little performance loss, that becomes relevant when doing essentially I/O, is to be expected.

The so-called *full virtualization* typically uses one of these two approaches, preferably the second one when underlying CPUs support hardware virtualization, and the first one as a fall-back solution: this is what VMware does. Full virtualization can be thought nearly a synonim of *emulation* in our scope: the guest operating system runs unmodified and does not have the perception of being virtualized. The main advantages of this approach is that you don't need special versions of the operating system to do virtualization, and that you can, in principle, emulate any architecture you want over any other architecture: sometimes, when the emulated architecture is not available (because it is a legacy architecture not developed anymore, or because you don't have the possibility to access that architecture easily), full virtualization becomes useful. However, as stated before, we expect a performance loss that it is not relevant if emulating old

consolles, but when virtualizing HPC applications such as HEP ones we are going to emulate, squeezing out every single speed improvement becomes critical, even from the economic point of view.

For these efficiency reasons there's a third approach that is called *paravirtualization*, the approach we are going to use for the implementation of the VAF. With this kind of virtualization code rewriting is delegated to the guest operating system, which then has to be aware of being virtualized and should be modified in order to handle code rewriting. The hypervisor can be taught to completely trust the modified guest kernel and expect it to do all the code rewriting stuff without even checking if every critical instruction is detected. Hardware acceleration technologies are also used with this approach. This method of letting the guest operating system modify the code to be executed by the hardware (combined with hardware acceleration technologies) is the fastest one, since the hypervisor doesn't have to translate raw code anymore, because the modified guest OS knows when exactly the code needs to be translated. We expect the paravirtualization approach to have zero impact on CPU-bound tasks, and we will see later that our expectations are fulfilled.

We finally note that paravirtualization does not fully conform with Popek and Goldberg's rules, because the guest OS knows that it is being virtualized, but the single user-space programs do not need any modification and are not aware of virtualization, so we can say that, by relaxing Popek and Goldberg's statements, paravirtualization still complies with them.

2.1.2.3 The Xen hypervisor

Xen is a virtualization platform that provides all the features we need to implement the VAF. It is an open-source and free VMM for Linux with the ability to paravirtualize properly modified Linux kernels.

Xen is developed by XenSource, acquired by Citrix systems in late 2007. Citrix started developing commercial virtualization solutions based on Xen (namely the XenServer Standard and Enterprise editions). From early February 2009 Citrix released its XenServer commercial product free of charge (but not open-source) with a license that allows deployment of an unlimited number of machines.

We are going to use Xen as a *paravirtualization* hypervisor, but it is also able to run in the so-called Xen HVM mode, just like other tools like VMware, making it capable of fully virtualizing unmodified operating systems by taking advantage of CPU hardware acceleration.

Xen has a layered structure. When booting the physical machine, the Xen hypervisor is first loaded: then, it takes care of managing resources between the other machines, called in Xen terminology *domains*. There's one *privileged* domain called *dom0*, which can be seen as the physical machine itself: the dom0

can access hardware resources directly. Any other domain is called *domU*, or *unprivileged domain*: these domains are the actual VMs.

Xen has its own process scheduler that schedules VMs just as an ordinary scheduler does with processes. With older Xen schedulers there was a strict correspondence between each physical CPU and VCPU, and a certain number of VCPUs could be "pinned" (assigned) to each VM: for instance, in a system with 8 CPUs and two VMs one could assign 2 CPUs to a VM and 6 CPUs to the other one. The new scheduler (called *credit scheduler*) works with priorities (which can be seen as assigning an "amount of CPU") and does not make correspondence between physical CPUs and VCPUs.

Almost every hypervisor is capable of assigning a certain "amount" of CPU to its virtual machines; Xen controls the "amount" of CPU with two per-VM parameters: the *cap* and the *weight*.

The *cap* sets the maximum percentage of CPU (or CPU efficiency, calculated as the fraction of real time not spent in idle cycles) that a VM can use: for instance, cap=800 means that the VM can use the maximum amount of 800% CPU efficiency, that is eight cores. It is even possible to assign a fractionary amount of CPUs to a VM (such as cap=50%, that means "half" CPU), but this is proved to cause problems in timing applications (see § B.2). Differently from older schedulers, in this case in a system with 8 CPUs and two VMs we can assign 8 CPUs to *each* machine (but no more than the number of CPUs on that system), because the cap is the maximum amount of CPU efficiency that can be used, and a VM is now capable of using every core when the other one is idle without having to manually change CPU pinning.

The *weight* is the relative priority of a VM with respect to the other ones: it can be seen as a hypervisor-level version of the Unix `nice` command. Priority is *relative*, meaning that, on equal terms (same cap, same memory), a VM running a CPU-bound job with weight=16 is eight times faster than the same host with weight=2.

These two parameters can be changed *dynamically*, i.e. without rebooting neither the hypervisor nor the single domUs: this was a required feature of the VAF. By intelligently combining cap and weight we can, for instance, configure an environment with two VMs where both have the same cap (set as the number of cores) but different weight. The *lower* weight can be assigned to the LCG WN and the *higher* weight to the PROOF slave: because the PROOF slave is mostly idle, the WN takes up all the CPUs most of the time, but when PROOF is active it has priority and it's the one who takes almost all the CPUs.

We have not yet mentioned the most important feature of the Xen hypervisor: almost any hypervisor is capable of dynamically changing CPU resources, but Xen is also capable of changing *assigned system memory* dynamically, resulting in maximizing the usage of every resource in the system and solving the problem

of underutilized resources. In terms of costs, we don't need twice the RAM as before to run two VMs on the same machine: we simply take the RAM from one machine when it's idle, and give it to the other one.

Since Grid nodes are always busy, if we subtract RAM from such nodes disk swap usage occurs. This fact has a big impact on Grid performances, but the slowdown is expected to last just as long as the interactive job is active – that is, a very small fraction of time of the Grid jobs. In principle we don't care about Grid slowdown, but surely we don't want that issue to propagate to the interactive node: since heavily swapping affects I/O, we will demonstrate that by physically separating disks we can obtain almost perfect performances isolation.

2.2 Virtualization feasibility benchmarks

Given two same-equipped machines we install and configure a paravirtualized host on one of them. The paravirtalized host is configured to use almost all the physical resources, leaving the least resources for the hypervisor (dom0).

By running the same suite of benckmarks on the physical host and on the paravirtualized one we shall be able to estimate the latter's performance loss, and finally if it is feasible or not to virtualize a HPC/HEP application.

The following types of benchmark will be run:

- **SysBench**: benchmarks on specific tasks (CPU, threads, mutexes, memory, disk I/O, MySQL transactions);
- **Geant4 Monte Carlo**: a real use case benchmark.

Specific-task benchmarks makes us able to know where exactly the performance loss is, while the Geant4 Monte Carlo benchmark, that makes a balanced use of each resource, will tell us whether a single task's performance loss is negligible or an effective bottleneck for real use.[2]

Please note that the Monte Carlo used in these tests is CPU bound, while there may be other simulations that are I/O bound.

2.2.1 Machines and operating system configuration

Head node is not relevant in these tests. The configuration for slave machines follows.

[2]At `http://www.bullopensource.org/xen/benchs.html` other interesting Xen benchmarks can be found, but at the time of writing our tests are more complete.

- Two HP ProLiant quad-core 2.33 GHz Intel CPUs *(8 cores total)*
- 8 GiB RAM
- Two 1000BASE-T ("gigabit") Ethernet network interfaces
- Two 140 GiB disks in Redundant Array of Inexpensive Disks (RAID) mirroring (RAID-1)

Linux CentOS 5.1 has been chosen for its rich Xen support. CentOS is a Linux distribution both free and open, and was developed to maintain binary compatibility with the commercial Red Hat Enterprise Linux. The commercial support makes Xen suitable for professional use, while its open source nature enabled us to modify and add code if needed: this aspect will be useful later.

Initial installations were performed on Ubuntu 7.10 Server, but the distribution was abandoned due to a poor Xen support. Moreover, the Ubuntu Team is officially supporting Kernel-based Virtual Machine (KVM) virtualization but not Xen paravirtualization starting from Hardy Heron (8.04)[3]. However, a Debian-based Linux distribution like Ubuntu will be taken into consideration for future installations because of a much easier package manager.

Some relevant operating system data:

- Kernel versions:
  - 2.6.18-53.el5xen on dom0 and domU
  - 2.6.18-53.el5PAE on the physical host
- Xen version: 3.0
- Xen scheduler: SMP Credit Scheduler (credit)[4]

2.2.2 SysBench benchmarks

SysBench performs several kinds of benchmarks. SysBench has been chosen over other free and open benchmark solutions because it is very fast and easy to compile (it does not require "exotic" mathematical libraries, as many other benchmarks do) and because it is *scalable*, i.e. it is capable of spawning different threads (user-selectable) and to divide the task over them.

Although SysBench was primarily developed to test a system that has to run a database server (such as MySQL), it has been chosen over other more generic

[3] `https://wiki.ubuntu.com/HardyHeron/Alpha4#Virtualization`

[4] Obtained with the command `xm dmesg|grep sched` on dom0.

benchmarks because it is multithreading: this feature enables us to see how the time to complete a task changes by changing the number of threads spawned[5].

2.2.2.1 CPU benchmark

This benchmark stresses the CPU with an algorithm that determines if a number between 1 and a maximum given (20000 in this case) is a prime number; this algorithm can scale by dividing its job into parallel threads. The number of threads is configurable: we run the test 64 times, from 1 to 64 threads.

Since the workload is balanced between each thread, we are expecting that there is an optimal number of threads, corresponding to the number of (V)CPUs, which minimizes the execution time. We also expect that there will be no relevant performance loss, since each VCPU is directly seen by the virtual machine without any architecture emulation: in other words, it is *paravirtualized*, not *fully virtualized*.

As expected, by looking at the plots (Figure 2.2, Figure 2.3) we note that the minimum time is reached at the number of 8 threads, that is the number of available cores; moreover, the two graphs are roughly overlapping, apart from some peaks limited under 15% (Figure 2.4). These differences can be explained by simply considering that in Xen there is no exact correspondence between virtual and physical CPUs, and the job is moved from a CPU to another more often than in an ordinary, non-virtualized Linux kernel.

2.2.2.2 Threads benchmark

Threads benchmark is designed to evaluate scheduler's performance: each thread tries to access the same set of resources, each of them allowing only one thread at a time. Threads are thus *concurrent*, and resources are called *mutexes* (from *mutual exclusion*): these resources are locked and released as soon as possible in this kind of test. More detauls about mutex access are found in § A.2.

The test was run with 1000 iterations and 8 mutexes, with an increasing number of concurrent threads (from 1 to 16 with a step of 1, then from 16 to 128 with a step of 8).

What we clearly see (Figure 2.5, Figure 2.6, Figure 2.7) is that the SMP Credits scheduler becomes really inefficient with more than six threads competing for the same mutexes.

[5] At `http://sysbench.sourceforge.net/docs/` the complete SysBench documentation can be found.

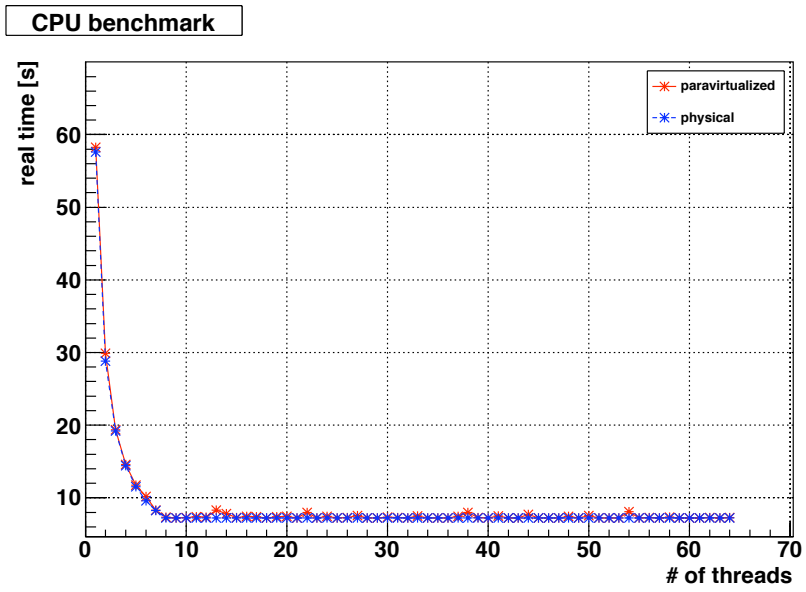


Figure 2.2: CPU benchmark: execution time comparison between physical and virtual host *(lower is better)*.

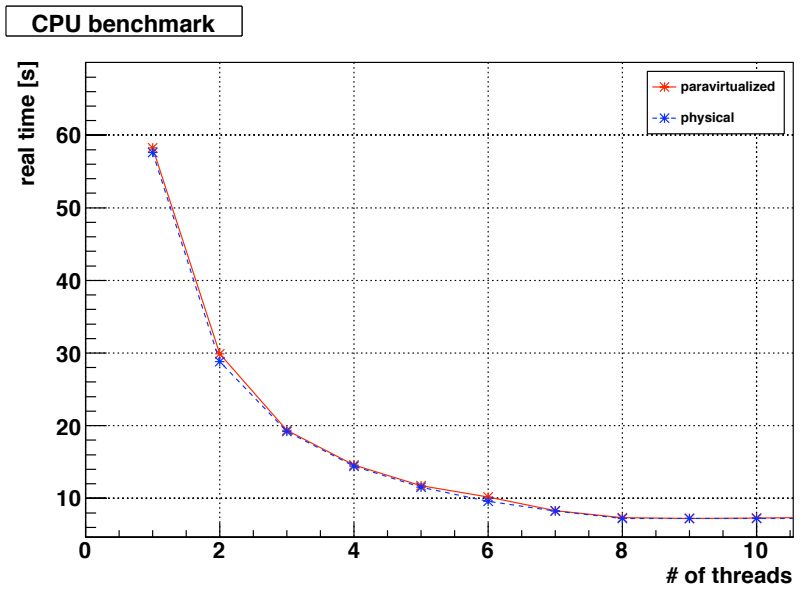


Figure 2.3: CPU benchmark: zoom-in showing the peaks of the paravirtualized host's execution time *(lower is better)*.

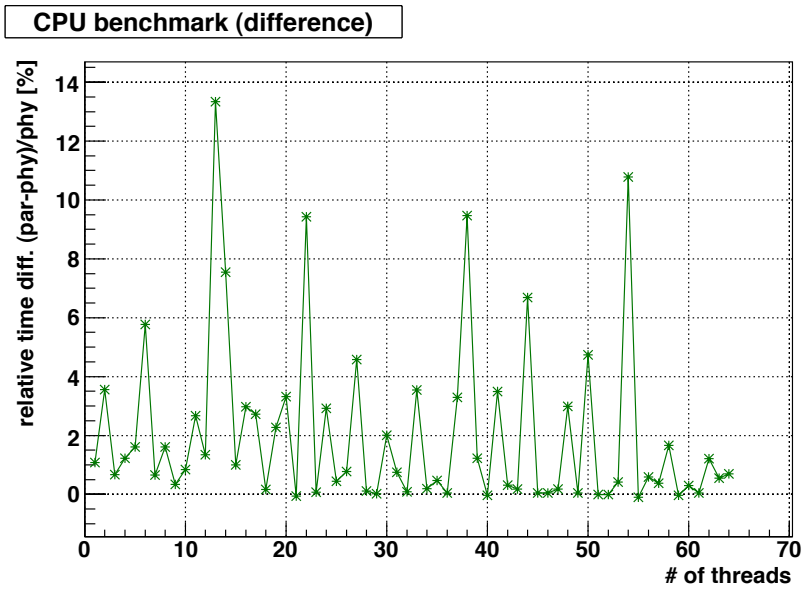


Figure 2.4: CPU benchmark: execution time difference percentage relative to the physical host.

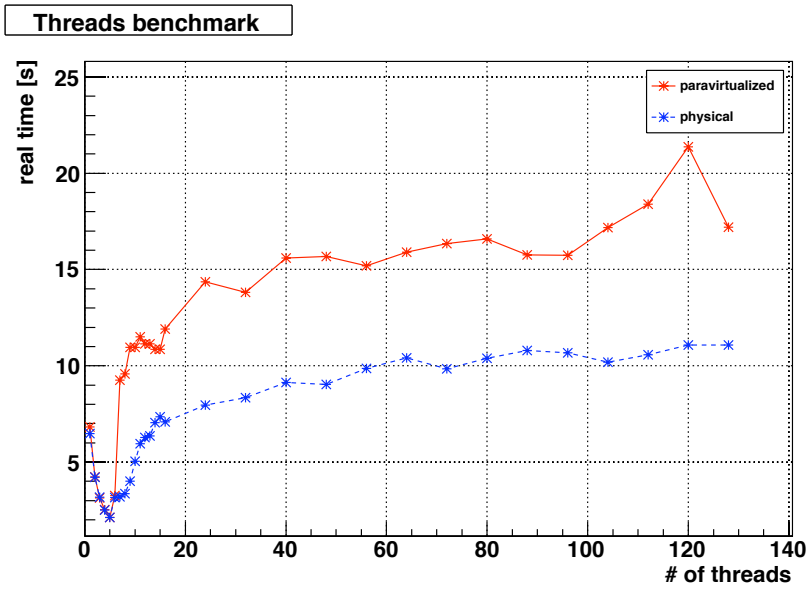


Figure 2.5: Threads benchmark: execution time comparison between physical and virtual host *(lower is better)*.

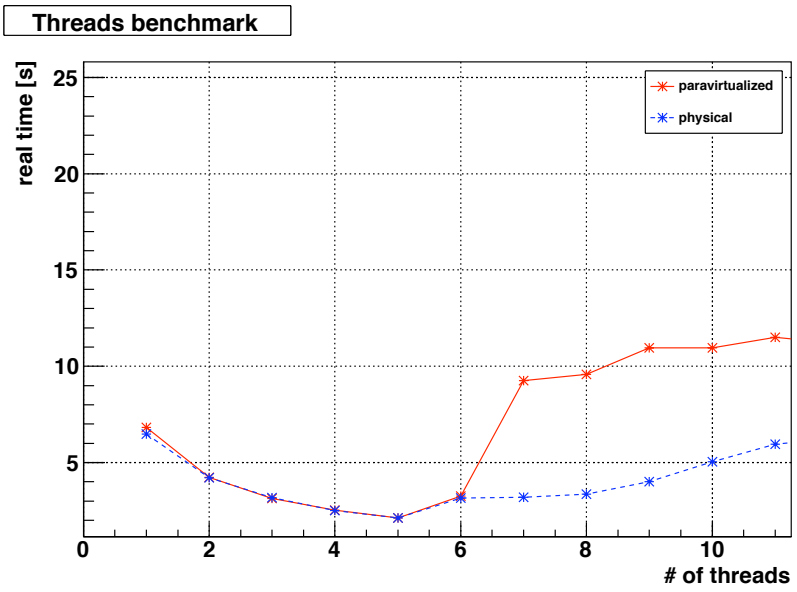


Figure 2.6: Threads benchmark: zoom-in showing the peaks of the paravirtualized host's execution time *(lower is better)*.

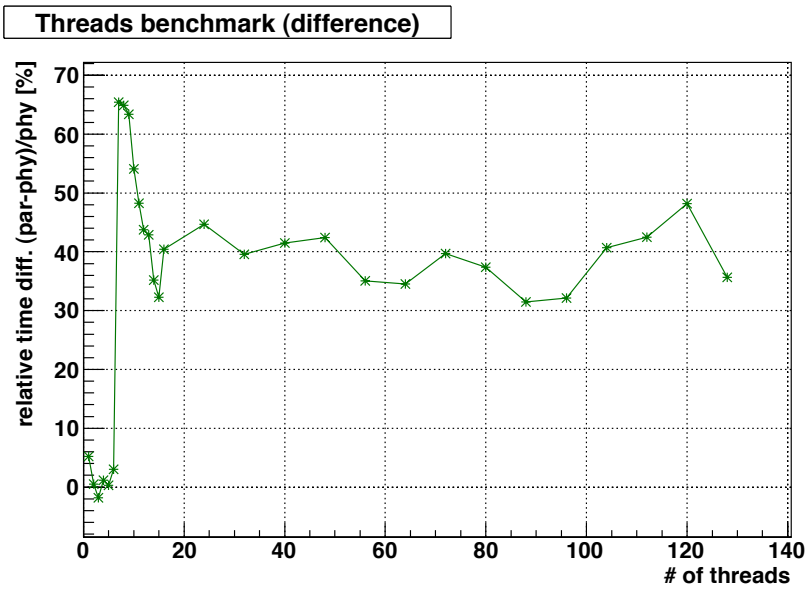


Figure 2.7: Threads benchmark: execution time difference percentage relative to the physical host.

2.2.2.3 Mutex benchmark

Unlike the threads benchmark, where many threads lock and immediately unlock a few resources, in this benchmark the lock is kept for a greater (even if short) amount of time. The test with 4096 mutexes, 50000 locks per request and 10000 null iterations before effectively acquiring the lock.

Since threads have other things to do (i.e. the null iterations) apart from locking resources, they are kept for a short period of time, which introduces less competition than in threads benchmark. Since there is little competition and no cooperation we expect a linear behavior: and that's what we get (Figure 2.8, Figure 2.9), where peaks are to be explained once again by the credit scheduler strategy, which can either affect thread scheduling or time measurements, or both.

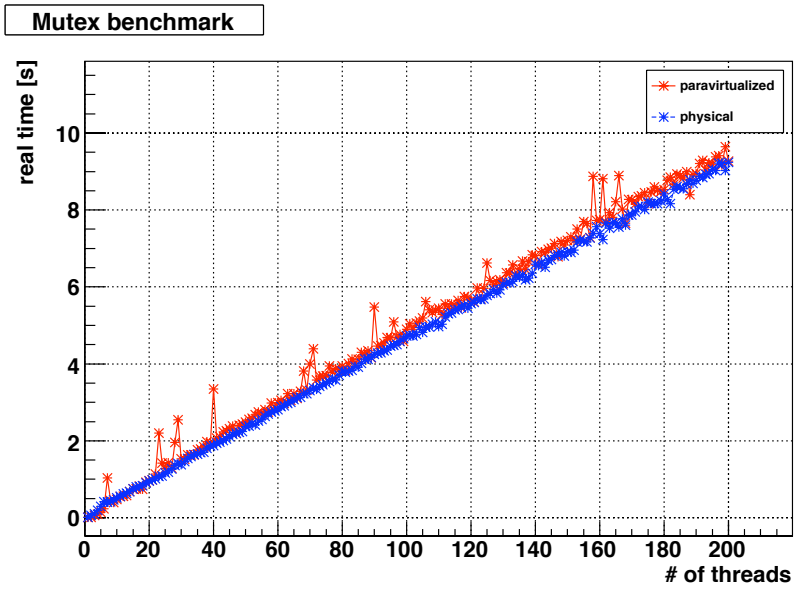


Figure 2.8: Mutex benchmark: execution time comparison between physical and virtual host *(lower is better)*.

2.2.2.4 Memory benchmark

This test benchmarks memory operations. We run the test by writing 5 GiB of memory with increasing number of threads (1 to 64 with a step of 1).

The results (Figure 2.10, Figure 2.11) show big performance loss: this happens because the hypervisor must apply an offset to each memory page request.

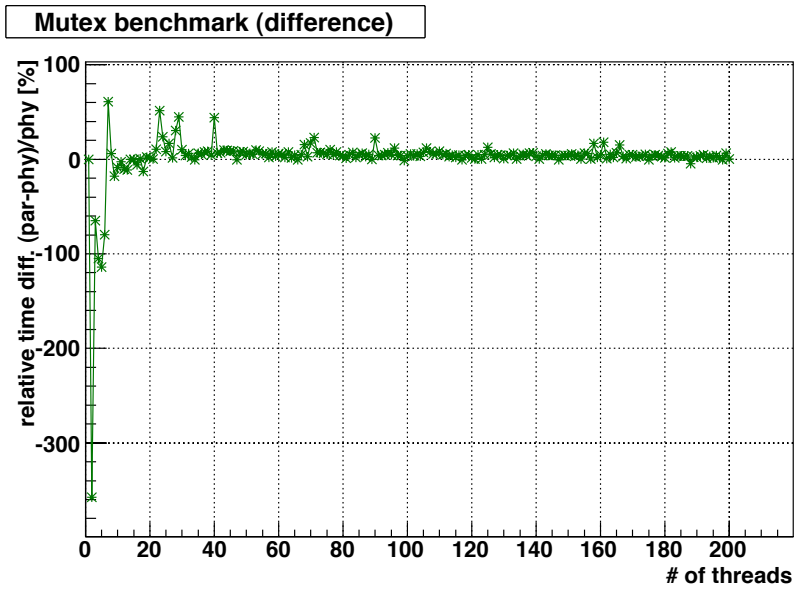


Figure 2.9: Mutex benchmark: execution time difference percentage relative to the physical host.

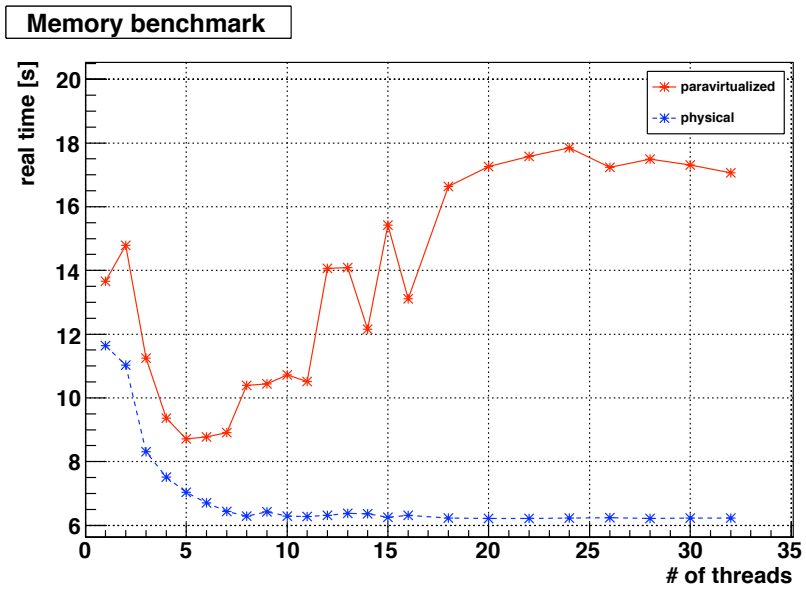


Figure 2.10: Memory writing benchmark: execution time comparison between physical and virtual host *(lower is better)*.

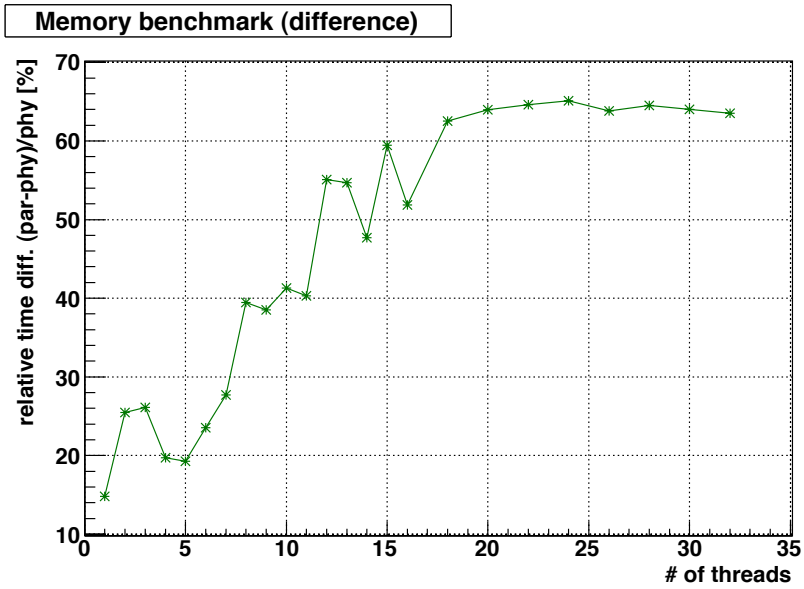


Figure 2.11: Memory writing benchmark: execution time difference percentage relative to the physical host.

2.2.2.5 Database benckmark

This benchmark, also called On-Line Transaction Processing (OLTP) benchmark, acts on a database by issuing several SQL queries within concurrent threads. In our test, 10^5 SELECT queries are executed on 10^6 records. The table must be created, then the test is executed, then the table is dropped.

Databases are the best testbed to benchmark memory and multithreading performances, since they use a lot of disk space and memory and they deal with concurrent access by using many threads and mutexes, and they do not use the CPU at full capacity (this fact can be seen by issuing a `top` command during test execution). The results of this test are useful to see the overall performance loss of Xen, but may not be significant for a HEP application that only stresses memory, disk and CPU and typically does not have any multithreaded features.

Since this test focuses on multithreading and concurrency shows performance loss in the previous tests, we don't expect great performances from the paravirtualized host. As a matter of facts we conclude that, by looking at the plots (Figure 2.12, Figure 2.13), an increasing number of threads shows no relevant performance loss, but on the countrary, an increasing number of threads accessing the same mutexes leads to relevant performance loss. A clear minimum can be found with 8 threads in both paravirtualized and physical host.

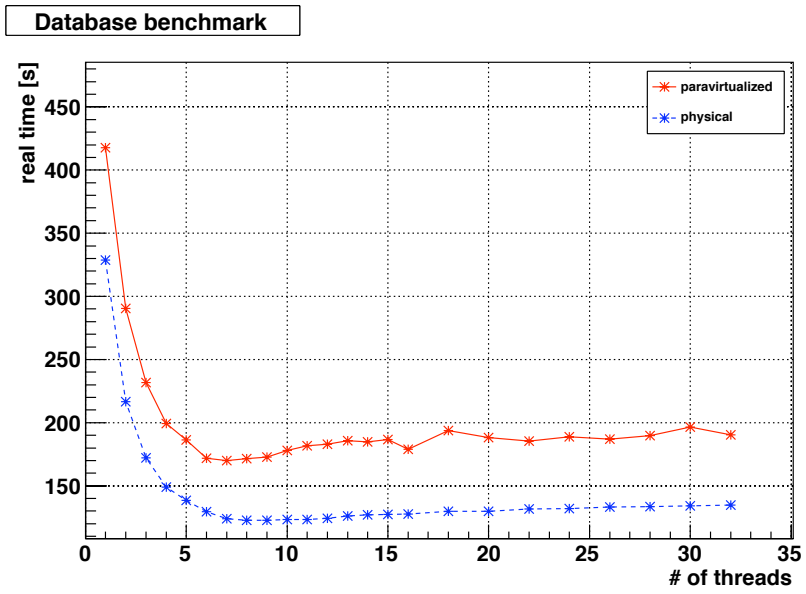


Figure 2.12: Database benchmark: execution time comparison between physical and virtual host *(lower is better)*.

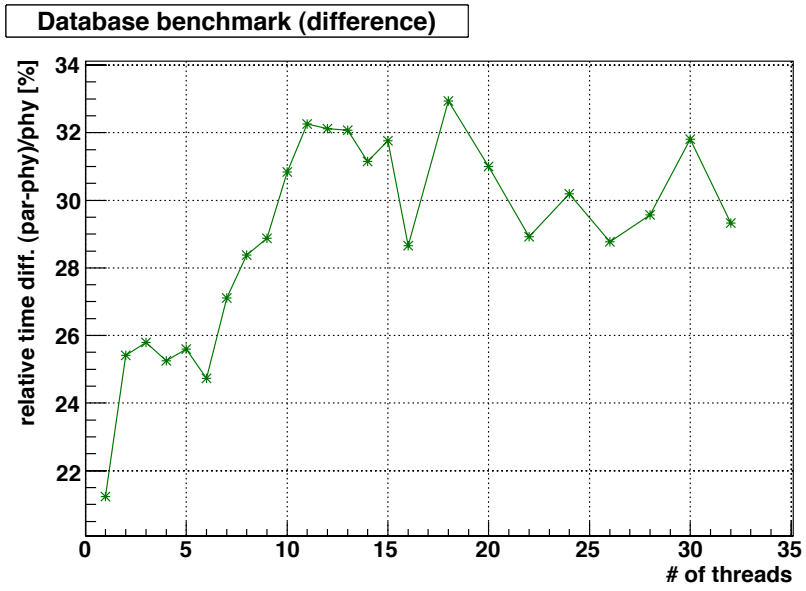


Figure 2.13: Database benchmark: execution time difference percentage relative to the physical host.

2.2.2.6 File I/O benchmark

File I/O has always been a performance issue for virtualized hosts: with these tests, run on both a native host and a Xen domU, we are going to estimate how much a Xen domU does worse than its native counterpart.

The test is subdivided into a *sequential write benchmark* and a *sequential read benchmark*. "Sequential" here means that data is read or written byte after byte, as opposed to a "random" access mode. Both tests read or write a total of 5 GiB[6] distributed in 128 files, 40 MiB each. On each run, 10 I/O workers (*POSIX threads*, from the Unix point of view) are spawn: one for each core (there are 8 cores) plus a little bit "overcommitting" to make the test as Grid environment like as possible.

Some tests are run on dom0s, others on domUs; when the test is done on a domU, it obeys to the following guidelines.

- It's the only domU running.
- domU has 7 GiB of RAM: this is extremely important, because more memory means more buffer cache and lower reading times.
- No other I/O intensive jobs are run neither on the domU nor on the hosting dom0 in order to avoid interference.

The write and read tests are performed as follows.

Write test

1. Sample files are written to the filesystem.
2. The sequential write test is repeated 5 times, collecting the timings in a file: mean and standard deviation are computed from these values.
3. Sample files are removed from the filesystem.

Read test

1. Sample files are written to the filesystem.
2. The sequential read test is executed once to allow data caching: no timing is collected at this point.
3. The sequential read test is then repeated 5 times, collecting the results in a file: mean and standard deviation are computed from these values too.

[6] 1 GiB = 1024^3 bytes

4. Sample files are removed from the filesystem.

The description of the tests done on native and virtual machines follows, along with a description of the underlying filesystem layers and a comment on their results.

Native host

- **dom0 native** *(Table 2.1)*: direct disk I/O on an ext3 filesystem created on a LVM logical volume with RAID 0+1. Although this is supposed to be the fastest configuration, the presence of RAID slightly slows down the write process with respect to a domU with no RAID.
- **dom0 remote NFS** *(Table 2.2)*: a NFS export is mounted from a remote host on the same subnet *(no NAT is performed)*.

Xen unprivileged guest

- **domU NFS** *(Table 2.3, Table 2.4)*: a NFS export is mounted from a NFS server listening on dom0, or from a NFS server listening onto an external network (requiring NAT). The first test acts like a sort of a "loopback" and its purpose is to measure if adding the layer of virtualization for disk access is comparable to adding the layer of a remote storage. The test shows that our hypothesis was true, since the results obtained in this case and in a physical partition used as VBD are nearly identical.

  For what concerns the remote NFS server we clearly see that, since NAT adds one more layer of abstraction between the client and the server, slows down performances by generating bigger packets: access to remote storage should *never* be accessed through NAT.

- **domU blkd** *(Table 2.5, Table 2.6)*: a native partition is used as a VBD; a partition table is built on it and an ext3 partition is created and used. This configuration is tested either in a RAID environment (two mirrored – RAID 0+1 – disks) or with a single disk. In this case we compare two identical situations where the sole difference is the presence or absence of RAID: as we look at the results we see that RAID slightly slows down the file I/O (but of course it can be useful for data integrity).
- **domU img** *(Table 2.7, Table 2.8)*: VBD is stored on a 20 GiB file located on dom0's filesystem, that is formatted either as ext3 or XFS, to test filesystem performance issues with very big files. No disk array (i.e., no RAID) is used here. The first thing we observe is that having the virtual machine disk on files dramatically cuts down performance (up to three times slower

that any other tested configuration in writing), while a positive feature is the improved portability and easier migration; however, we are not interested in these features. XFS deals much better with very big files than ext3 in their default configuration (4 KiB, or 4096 bytes block size). Probably the maximum block size (64 KiB for XFS, 8 KiB for ext3) would have been more appropriate, but this value should be choosen accordingly to the maximum memory page size on the current architecture (that is, 4 KiB on IA-32). Moreover, it was impossible to configure XFS with bigger blocks way because block size is a hard coded value in standard CentOS kernel XFS module (which is not fully supported, ant it's even not present in the default installation).

Results are represented in Figure 2.14. The conclusions of these tests are very clear: we are going to use physical partitions as VBDs for our virtual machines, since it is the fastest and most feasible method, even if it lacks flexibility of migration (but again, we are not interested in such a capability). Even if there's a performance loss due to file access with respect to the physical situation we should consider that many I/O operations are to and from remote storage (in the Grid), and are mainly CPU-bound and not I/O bound.

#	**read [s]**	**write [s]**
0	2.1081	91.3679
1	2.0814	90.3059
2	2.1985	95.6482
3	2.4558	92.1826
4	2.1373	93.4190
Mean	**2.1962±0.1515**	**92.5847±2.0565**

Table 2.1: File I/O test results on a native *(i.e., non virtualized)* dom0.

2.2.3 Geant4 benchmark

This benchmark measures the real time taken to complete a medical physics simulation that uses the Geant4 framework[2]. Since Monte Carlo simulations generate random events, simple parallelization can be achieved by contemporarily launching eight instances of the same simulation.

The simulation is being currently developed by Andrea Attili and Faiza Bourhaleb in the contest of the INFN-TPS project[18].

#	read [s]	write [s]
0	2.2330	128.8870
1	2.0375	124.9416
2	2.1855	124.6543
3	2.1780	126.1185
4	2.1003	128.0597
Mean	**2.1469±0.0775**	**126.5322±1.8779**

Table 2.2: File I/O test results on a dom0 mounting a remote NFS export on the same local network.

#	read [s]	write [s]
0	11.4913	107.5554
1	10.2563	103.7454
2	11.3242	99.6695
3	11.4358	96.2365
4	10.0555	105.7627
Mean	**10.9126±0.6970**	**102.5939±4.6085**

Table 2.3: File I/O test results on a domU mounting a NFS exported from its dom0.

#	read [s]	write [s]
0	9.2514	148.1351
1	9.4769	148.1883
2	10.1800	147.8830
3	9.3267	147.8920
4	9.8397	146.8373
Mean	**9.6149±0.3886**	**147.7871±0.5487**

Table 2.4: File I/O test results on a domU mounting a remote NFS export on an external network that requires NAT.

#	read [s]	write [s]
0	11.4188	97.4485
1	10.7277	97.9478
2	11.1306	97.5696
3	10.5785	98.6422
4	10.7937	98.1198
Mean	**10.9299±0.3399**	**97.9456±0.4754**

Table 2.5: File I/O test results on a domU using an entire physical partition as a VBD. The physical partition runs on a hardware RAID 0+1.

#	read [s]	write [s]
0	11.4027	77.7030
1	11.1304	77.9048
2	11.0184	78.0775
3	10.7040	79.3414
4	11.0898	79.1975
Mean	**11.0691±0.2507**	**78.4448±0.7660**

Table 2.6: File I/O test results on a domU using an entire physical partition as a VBD. No RAID is active for the physical partition.

#	read [s]	write [s]
0	11.1148	374.2615
1	10.7136	394.2198
2	11.1170	448.6667
3	11.0762	438.3493
4	11.0046	465.7587
Mean	**11.0052±0.1693**	**424.2512±38.4662**

Table 2.7: File I/O test results on a domU using an image file on dom0 as VBD. The filesystem on dom0 is ext3.

#	read [s]	write [s]
0	10.8362	241.4401
1	11.2859	322.0828
2	10.9252	291.5342
3	10.5696	286.1381
4	11.4289	295.4730
Mean	**11.0092±0.3475**	**287.3336±29.1507**

Table 2.8: File I/O test results on a domU using an image file on dom0 as VBD. The filesystem on dom0 is XFS.

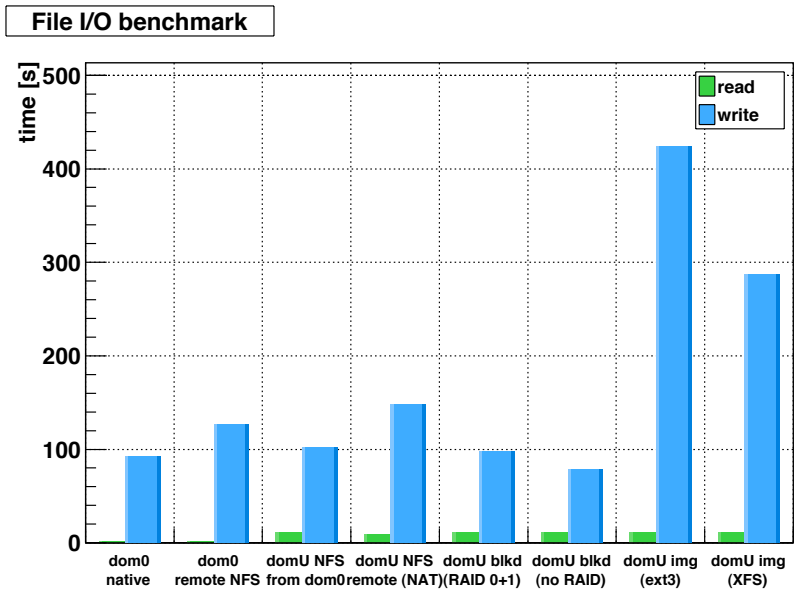


Figure 2.14: File I/O test results *(lower is better)*.

We have slightly modified the code in order to choose the output file and to explicitly set the random seed: this feature is fundamental to achieve reproducibility and therefore comparability, because we are assured that exactly the same code is executed through the whole simulation, but most importantly because time differences up to ∼10% may occur with randomly choosen seeds, as measured by a previous test with the original version of the simulations, that used the return value of the C function `time()` as a seed.

Results are reported in Table 2.9(a), (b) and (c). As we can see, in a simulation that is essentially CPU-bound, time difference is negligible. We can say that the simulation is CPU-bound because in ∼ 30 hours of eight simulations running parallely only 8 GiB of data are produced. We conclude that in a real, CPU-bound use case, there's no difference between running the task on a physical or a virtual machine, stating once again that virtualizing computing resources is feasible from the performance point of view.

The scripts written to launch and control multiple parallel instances of the simulation can be found in § E.1.1 and § E.1.2.

2.3 Real-time monitoring with dynamic resources reallocation

2.3.1 Synopsis

2.3.1.1 Resources erosion vs. suspension

The use case of the VAF consists on tasks that use a big amount of memory and CPUs at their full capacity, both on the Grid and the PROOF side: the prototype will consist of two VMs, one for the Grid WN and one for the PROOF slave. Once the prototype is complete we shall be able to reallocate resources from one machine to the other one with no impact on system stability. Resources can be reallocated either dynamically, with both guests running, or by temporarily suspending the unused guest.

Dynamic allocation consists in eroding both memory and CPU from a running guest, *without rebooting it*, and assigning them to the other one: as we have already seen, Xen is capable of dynamically moving memory, while other solutions are not (i.e. VMware Server).

An alternative to dynamic allocation is to fully *suspend* a VM and then *resuming* the other one. Although this sounds appealing, primarily because no memory erosion and no intensive disk swap usage occur, we should keep in mind that running jobs on a Grid need to access remote data located on servers that are not under our control: suspending a VM while a remote transaction is in

#	[hr:min:s]
1	29:00:49
2	28:26:54
3	28:03:40
4	29:09:33
5	28:50:46
6	28:19:00
7	28:40:18
8	29:20:41
avg	**28:43:58**
std	**00:26:25**

(a) Physical machine.

#	[hr:min:s]
1	28:35:39
2	28:41:02
3	28:03:57
4	28:27:22
5	28:39:33
6	28:06:07
7	28:40:53
8	28:31:05
avg	**28:28:12**
std	**00:15:06**

(b) Virtual machine.

Overall time difference [hr:min:s]	02:06:03
Mean time difference [hr:min:s]	00:15:45 ± 00:30:25
Diff. relative to phys. machine time	0.91%
Total GEANT4 events generated	200000
Per-instance events	25000
Time difference per event [s]	0.04 ± 0.07

(c) Summary and comparison.

Table 2.9: Time taken to complete Geant4 simulation on each core on the physical and the virtual machine.

progress could easily lead to job crash, unexpected results or even data loss. One could code jobs that are aware of *transaction interruption* (i.e., if a remote file or database operation is interrupted, both local client and remote server should be able to recover the interrupted transaction somehow.), but existing simulation and reconstruction programs are mostly unaware of that, thus the better solution under a practical point of view seems to be the VM *slowdown*, not the *suspension*, even if under the computing point of view the latter is certainly more efficient.

2.3.1.2 Purposes of performance monitoring

We decide to test the feasibility of dynamic allocation. The first issue we want to measure is how much time is spent in reallocating memory: it certainly takes some time for a VM we want to slow down to move most of its RAM pages into a swap space, and viceversa, and it should be a small amount of time in order to be hypothetically driven on demand.

Another issue is *stability*: we want to know *how much* a job is slowed down, and last but not least if it is slowed down to "death" or not – in other words, if the running jobs crash or not if subjected to memory and CPU erosion, particularly during the transition phase.

Also, from the Linux point of view, after this test we should be able to better understant the Linux memory management and the AliRoot memory usage.

These are the questions we are going to answer with our test.

- How much does it take for a domU to yield or to gain memory?
- How much the jobs are slowed down by this resource erosion?
- Do the jobs crash?
- How does AliRoot use the memory?
- How does the Linux kernel memory manager behave?

2.3.2 Linux facilities to get system resources information

Before effectively describing our tests, a description of the Linux facilities used to monitor system performance in real time follows.

There are different approaches to system and processes monitoring under Linux. Since process monitoring is generally a resource-consuming task a special procedure has been developed by combining a proper hardware extension, a Linux kernel patch and a user space program: the kernel framework and the user space tools are called perfmon2[12].

perfmon2 is an unobtrousive approach to performance monitoring, because on dedicated hardware (namely, Itanium/IA-64) the monitoring task is executed by a special CPU extension called PMU: in this case, no performance loss occurs at all while monitoring resources usage. However, since PMUs are not available for older hardware (for example, IA-32), perfmon2 is capable of monitoring performance in software mode with only a negligible impact on system performance[15].

However, perfmon2 can not be used in our tests, because there is no way a VMs can access the PMU. Without hardware extensions it can be used, but a patch is available only for kernel versions greater than 2.6.24, but Xen were only available for 2.6.18 at the time of the test.

Another approach is to periodically read the Linux `proc` filesystem to gather system information. The Linux `proc` filesystem is a virtual filesystem that contains files with information (usually in text format) about currently used system resources and running processes. Each process has its own subdirectory with files holding information about its priority, CPU usage, environment variables, and so on. More information about the `proc` filesystem can be found in § B.1.

2.3.3 Tests details

Doing such a test requires:

- an AliRoot simulation that does exactly the same things every time is run;
- a script to periodically yield/take resources from the domU;
- a script to monitor the system resources usage on domU;
- a simple text-based interface to control the background simulations and to collect data from monitor.

2.3.3.1 The AliRoot simulation and event timing

The AliRoot simulation is controlled by a configuration file that essentially tells the framework (see § 1.3.1.1) to simulate a pp collision. Each instance is launched by a shell script (see § E.2.4).

No data is received or sent through the network and output files are relatively small, making this simulation CPU bound and not I/O bound when resources are fully available to the VM (while we expect it to suddenly become I/O bound when swap heavy occurs).

The need for running several instances occurs because AliRoot uses only one core. To make AliRoot execute exactly the same instructions every time we set the random seed to a known number that does never change.

Because AliRoot uses a great amount of memory we should assure that swap does not occur with the maximum memory set for the VM and the number of instances we'd like to run: this is the situation if we decide to launch *four instances with 3.5 GiB RAM and a cap of 400*. No other VMs are running on the same physical machine, so the relative weight is not an important parameter.

This AliRoot simulation can be run with different number of events per job to collect different kinds of information.

- **A job with only one event** can tell us the cumulative time to load AliRoot, perform the event simulation and write data to disk: the average real time to complete is about one minute and a half (with resources fully available to the VM). Such a job is put into an infinite loop that continuously relaunches the simulation when one has finished, writing to a file when the simulation exited and how long did it run. Thus, this test tells us how long a simulation takes to complete with a different amount of resources available.

- **A job with a greater number of events** stays in memory (RAM and swap) for longer, giving us information about AliRoot and Linux stability and AliRoot memory usage.

Each instance is run in a separate screen to allow backgrounding for processes possibly needing a tty. Event timing is appended in a separate file (see § C.1) for each instance: the files are subsequently gathered together and merged.

2.3.3.2 Resource control on dom0

A shell script has been written to do resource control on the dom0. The script periodically sets the available resources for each virtual guest. In our tests, the configurations in Table 2.10 are chosen every two hours (7200 s). For more details and the code, see § E.2.1.

2.3.3.3 Monitoring on domU

A bash script called `Monitor.sh` (see § E.2.3) is run in background along with some AliRoot instances. It collects information about the system and the running simulation jobs every five seconds. Every information is read from the `proc` filesystem (see § B.1) and written on a file (see § C.2).

Note that a poll interval of less than five seconds interferes significantly with system overall performance, because interpreted scripts are used and many Unix utilities are launched during the execution.

No.	Mem [MiB]	Cap [%]
1	3584	400
2	256	100
3	3584	400
4	256	100
5	1792	300
6	512	200
7	1792	300
8	512	200

Table 2.10: Configuration list from which the `MoveResource` script switches every two hours.

2.3.3.4 The control interface: SimuPerfMon

A control interface named SimuPerfMon has been written to control simulation jobs on domU, to avoid the repetitivity of some tasks. With this interface one can launch and stop jobs, monitor running time and instances, clean up, collect and archive data. Details about this interface can be found in § C.3.

2.3.3.5 Data processing

Every information collected by SimuPerfMon is finally plotted as a function of time in three different plots:

- memory (total and used RAM, used swap, buffer cache, swap cache);
- CPU efficiency;
- event duration.

CPU efficiency is not directly measured; instead, it is calculated using the following collected values.

- **User time**: the time spent by the CPU(s) to execute job's instructions.
- **System time** or **kernel time**: the time the kernel worked upon the job's request. This is the case of I/O, for example: if the process wants to do I/O asks the kernel to do that, so the time spent is accounted as *system time*, not *user time*.

- **Real time**: it is the total execution time, i.e. the difference between job's end and start timestamps. It can be interpreted as the sum of user, system and idle time, where *idle time* is simply the time spent by the job in doing nothing.

Mean CPU usage is calculated as follows *(in percent)*:

$$\text{mean\_cpu\_usage} = 100 \times \frac{\text{user\_time} + \text{system\_time}}{\text{real\_time}}$$

"Differential" CPU usage between two adjacent samplings is calculated as follows *(in percent)*:

$$\text{diff\_cpu\_usage}_i = 100 \times \frac{(\text{usr\_time}_i - \text{usr\_time}_{i-1}) + (\text{sys\_time}_i - \text{sys\_time}_{i-1})}{\text{uptime}_i - \text{uptime}_{i-1}}$$

where all times must use the same unit of measurement.

2.3.4 Results, comments and conclusions

The following two tests are executed as described before. The only difference is that the first test repeaditly runs four AliRoot instances with only one event each, while the second test runs the same four instances for a large number of events: since in the second test we are interested in stability, no event time is measured.

2.3.4.1 Event timing test

Results of this test are plotted in Figure 2.15 and Figure 2.16. Observations on the plots follow.

Peaks in the CPU and memory plots. AliRoot instances are reloaded each time the previous job has finished: this is why we have many peaks in the plots. Memory consumption rises when AliRoot is loaded, and it drastically falls down when it is terminated. The same thing happens with CPU usage because reloading AliRoot causes the reload of large amount of static data (i.e. calibration of detectors) that does I/O without using the CPU: when data has finished loading *(low CPU usage)* event simulation starts *(full capacity CPU usage)*.

CPU usage The density of the magenta peaks *(CPU usage by the AliRoot instances)* on the big plot shows that it is difficult for the processes to reach the cap when memory is far below 3.5 GiB. This is due to the swap space usage: the more swap is used *(orange line)*, the more the AliRoot simulations become I/O bound. The peak right after 32000 s is probably due to a glitch in the system interrupt timer (see § B.2).

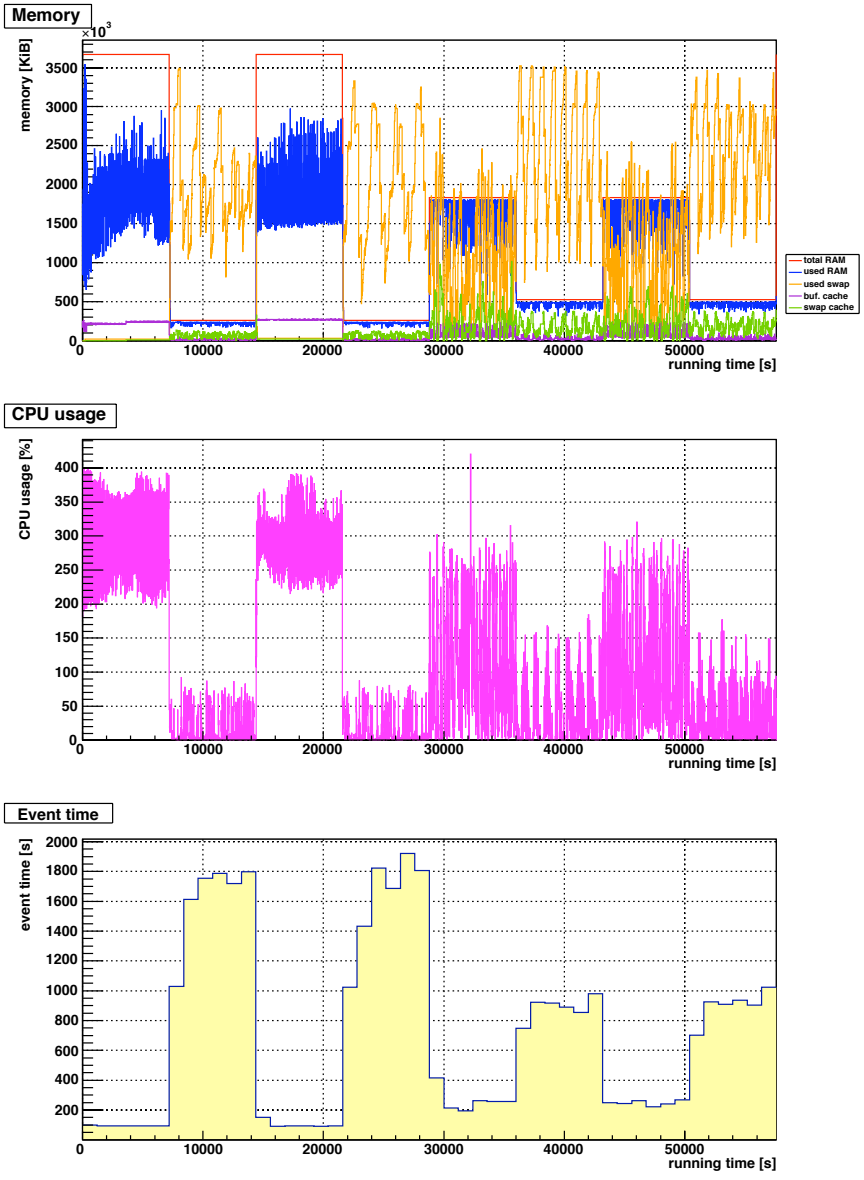


Figure 2.15: Resources monitor plot with event timing.

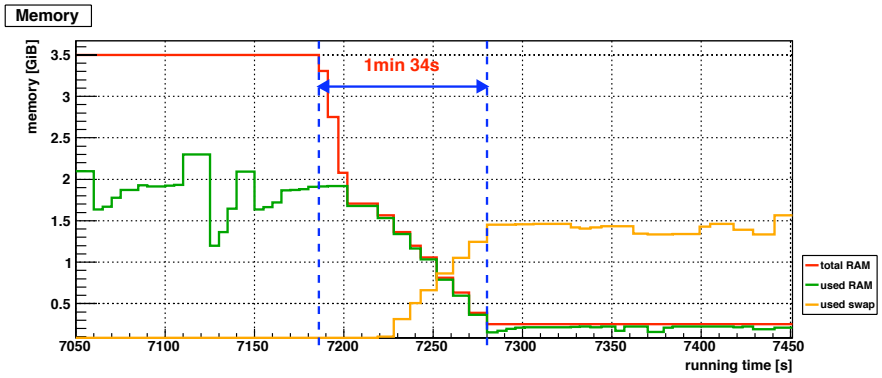


Figure 2.16: Memory monitor plot magnification that shows the time needed to yield memory, i.e. to turn into a configuration with much less RAM.

Swap usage As expected, when eroding memory, the swap space usage increases. In the second test we will see a less trivial result that occurs when giving memory back.

Buffer cache usage Linux uses all the free RAM to cache read and written data from system disks: caches are automatically invalidated (and consequently freed) when an active process requests memory. The second test clearly shows it, but in this case we can only see that the buffer cache drastically falls to zero when reducing memory.

Event time With full memory resources (cap=400%, mem=3.5 GiB) and half memory resources (cap=300%, mem=1.75 GiB) execution time depends more on CPU than on memory: this means that swap pages are written but rarely or never accessed. This is probabily due to AliRoot calibration data. When memory is lower, execution time increases several times compared to the full resources time (from 1600 to 2000 s compared to ∼100-120 s) because of the swap usage. Jobs *tremendously slow down*, but *they do never crash*, even during transitions from high to low resources, showing the stability of both the underlying operating system facilities and the analysis and simulation framework.

Resources transition time The time for a VM to yield resources while jobs are running can be estimated by magnifying the steepest transition (from 3.5 GiB to 256 MiB), as shown in the zoomed plot (Figure 2.16). The ramp lasts for at least 100 s before reaching the lowest limit. This low transition time is a very good result because it is little enough to allow on-demand resources reallocation.

2.3.4.2 Stability and memory management test

Results of this test are plotted in Figure 2.17. Observations on the plots follow.

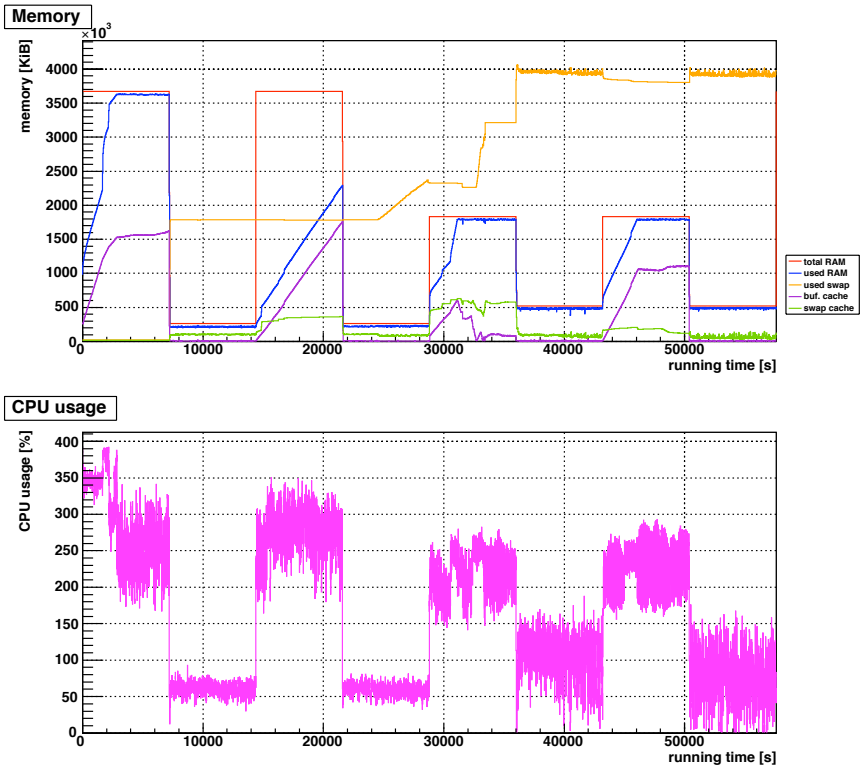


Figure 2.17: Resources monitor plot with no event timing.

CPU usage Since there are no repeated loadings and unloadings of AliRoot the memory or swap are never cleared, so the swap space is used continuously with low memory resources, causing less peaks and a CPU usage far below the cap. Note that also in the maximum resources configuration window CPU usage falls down when the buffer cache *(purple line)* cannot grow anymore, meaning that there is some disk data that needs to be accessed often and cannot be cached. In this case, the simulation becomes I/O bound for the disk data it needs to access, not for the swap usage – that is *zero (orange line)*, as one can easily see.

Swap usage and swap cache Swapped memory remains high, after the first ramp, and never decreases: this is in part due to the *swap cache*, but mostly

because there is a lot of data that AliRoot loads and *never uses*. With long runs, the impact of unused calibration data on system memory and swap is evident.

Buffer cache. As noticed before, since many swap memory pages are never invalidated, when restoring resources RAM memory becomes more and more used, *but this is not due to AliRoot itself*: used memory grows as buffer cache grows. Linux cleverly uses memory for speeding up data read and write, even if these caching effects are not visible in the simulation used, because each piece of data is read and written only once during the lifetime of the run.

Jobs stability The same four instances constantly run during all the test duration (16 hours), and they never crash.

2.3.4.3 Conclusions

These tests are very important to estimate the feasibility of the VAF. The only unclear point is the huge memory occupation by unused resources done by AliRoot, but the overall results are positive: jobs never crash and Linux is a good choice for its memory, cache and swap management. Moreover we don't care about AliRoot memory usage, because unused pages go to swap and they are never accessed, by freeing the memory. We can safely conclude that a VAF is feasible for our purposes.

Chapter 3

Working prototype of the Virtual Analysis Facility

3.1 Prototype implementation

Since we have demonstrated that a VAF is feasible we are going to discuss the implementation of a small (but scalable) VAF production prototype.

The main target of setting up a working prototype is to test if our analysis facility works in a production environment, with both Grid and PROOF nodes active. We will configure our facility by focusing on the ease of mainteinance and quick scalability: in other words, it should be easy to add new machines and to fix problems on existing ones.

3.1.1 Machines, network and storage

Our machines are four HP ProLiant DL360 servers with eight cores each, plus a multi-purpose head node (IBM eServe) that hosts the ALICE software and runs any other ancillary service. A picture of these machines inside a rack can be seen in Figure 3.1.

Three of the ProLiants are equipped with 8 GiB RAM, while one has 16 GiB. Each ProLiant has six SAS disk slots, two of which are filled with ∼140 GiB disks. These disks are not in a RAID array: the first disk hosts the dom0 kernel and hypervisor, plus the Grid VM with its swap partition, while the second disk hosts the PROOF VM, again along with its swap partition. The main reason why the disks are separated is to ensure performance insulation: when taking out resources from one VM, it starts to swap, and if the other machine were on the same disk it

would suffer from slowdowns in its I/O. Moreover, we are not interested in any kind of data redundancy on the machines at this point of the implementation, because important data is stored in dedicated storage facilities belonging to the Grid, and if one of our disks fails we just replace it and reinstall the software because we probably won't have lost critical data.

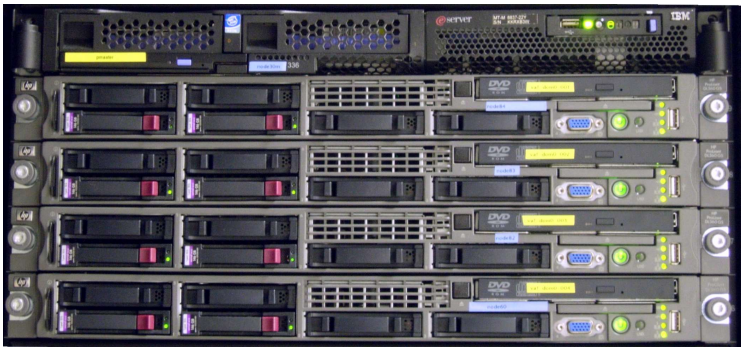


Figure 3.1: Picture of the VAF machines in a rack: starting from the top, there's the head node (IBM eServe) and the four HP ProLiant "slaves".

Our HP machines have two gigabit ethernet interfaces, plus a separated one dedicated to a feature called iLO: by assigning an appropriate IP address and connecting (via ssh) to iLO it is possible to see what happens on the machine from a remote host as if there were a screen and a keyboard physically attached to the server. iLO is also available when the server is turned off but powered, so we can remotely control the boot sequence and configure machine parameters such as RAID via a textual BIOS interface. An iLO web interface is also available.

Power supplies on our HP machines are *fully redundant*, i.e. there are two power supplies each.

In our prototype each machine (both physical and virtual) is connected to a non-dedicated switch; altogether the machines are part of a local network that sees any other machine through NAT. NAT for the entire local network is done by the head node. Having a non-dedicated switch and using NAT has a strong impact on performances, but this is going to change in the near future.

The head node runs the following services for the network: IP gateway and NAT, DNS, NTP, DHCP, web server, install server – including PXE boot server via DHCP, and TFTP – and, of course, the most important one: PROOF master and remote login for the users and administrators via ssh.

3.1.2 Mass Linux installations

In our scenario we have got several identical virtual and physical hosts which will act as computing nodes. Since we are building a prototype the machines are a few, but in a hypothetical "real world" usage we may end up with dozens of machines to maintain, so the prototype is built with *scalability* in mind. There are at least three methods to face this situation.

The "manual" way As the name suggests, we may simply install each dom0 and domU "by hand". In a large server room this means repeating the same steps for partitioning, selecting packages, creating a root user, and so on. A KVM switch needs also to be connected to the machine we need to control, and we need to be physically close to the machine to perform the installation.

PXE and NFS root PXE is a technology developed by Intel and other hardware vendors in order to provide a bootloader and a kernel image to boot via a network, functioning as an extension of DHCP (which is used to assign the IP address and tell that there is the possibility to boot from the network) and TFTP (which is used to send files to the booting client).

NFS is the most common centralized and networked file system used in Unix, because its protocol is simple and it is very easy and painless to configure. "NFS root" means that each client's root directory is mounted using NFS.

Using both PXE and NFS root one could configure a machine that does not need any local disk to run – a so-called *diskless* machine. This kind of approach seems appealing at first, because:

- all files of many machines are stored and managed once;
- no local disk is strictly required to run the system.

Indeed, it has been tried and several drawbacks emerged.

- Each machine requires a local space to store temporary files. This may be achieved by mounting as ramdisks some directories as /tmp, but we don't want to waste RAM for that.
- NFS root and DHCP-at-boot are simple features but require *kernel recompilation*, but compiling the Xen hypervisor and a Linux kernel that supports Xen is a complicated task for a system administrator due to version matchings and patches. We prefer to use pre-compiled kernels because they are easier to maintain and they can be updated by simply using a package manager.

Centralized installation system: Cobbler Since our machines have hard disks we may think about the solution of centrally managing the installation and per-machine customizations through a single server (the "install server"). Since we use a Red Hat based Linux distribution (CentOS), the installation system (called *anaconda*) supports a way to automatize every step of the installation procedure through scripts named "kickstarts". Through the install server we can reinstall an operating system that gets broken for whatever reason without even logging on the remote machine through remote control interfaces such as iLO.

Cobbler is an install server with plenty of interesting features: it does everything it's needed to boot, install and manage network installations. Cobbler itself is only a bunch of command-line utilities and web configuration pages, but it has the power to aggregate every feature we need. Once installed, Cobbler takes care of:

- PXE for network installations;
- DHCP and DNS through various methods *(either dhcpd+bind or dnsmasq)*;
- rsync for repositories mirroring and synchronization;
- a web interface from which we can access most of the functionalities (see Figure 3.2).

There's nothing particular in our Cobbler configuration: the only non-default parameter is the DNS. For the DNS service we decided to use dnsmasq instead of the more complicated and default bind.

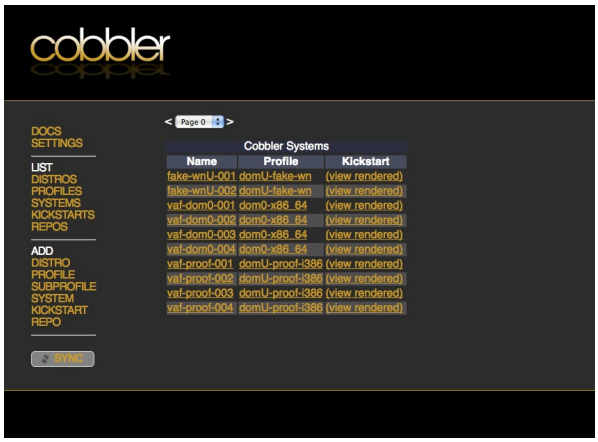


Figure 3.2: Screenshot of the Cobbler web interface, showing the list of the machines managed by Cobbler.

3.1.3 Monitoring and control interface

We are currently developing a monitoring and control web interface for the VAF. There's an online preliminary version that runs on secure HTTP (apache2 is used as webserver).

The control interface is a *web application* built upon many technologies: GWT, the Google Web Toolkit[1], is the framework used to design the interface. Communications between the web application and the server occur through AJAX and PHP technologies, in order to avoid web page reload as soon as data is updated. A screenshot of the current interface can be seen in Figure 3.3.

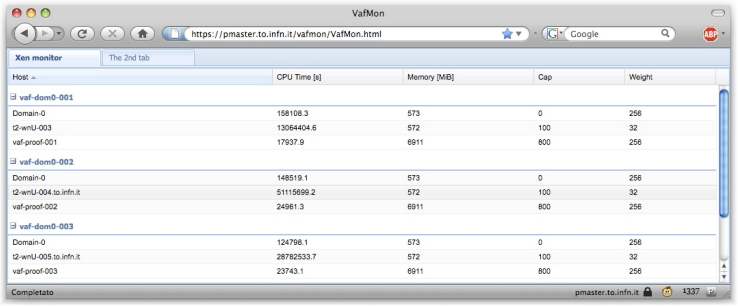


Figure 3.3: Screenshot of the VAF monitor web interface, showing VMs grouped by their dom0s. Data is dynamically updated every few seconds.

3.1.4 Scalla and PROOF configuration

The Scalla suite, also known as *xrootd*[10] is a software cluster architecture designed to bring low-latency remote data access (via a daemon called `xrootd`) and to develop a scalable storage solution (via `cmsd`, that recently replaced `olbd`).

Data from within ROOT is always accessed via the `TFile` class, which acts as a wrapper on a more specific subclass dependant on the file access method. If the file is to be obtained via a xrootd pool the `TXNetFile`[10] is the proper interface, transparent to the user.

Outside ROOT there are other methods to access xrootd: the most common including the `xrdcp` command-line client, and a FUSE extension called XrootdFS[2] is currently under development.

Scalla/xrootd functionalities can be expanded by writing appropriate plugins, including the PROOF daemon. The PROOF approach to parallelization is *data-*

[1] `http://code.google.com/webtoolkit/`

[2] `http://wt2.slac.stanford.edu/xrootdfs/xrootdfs.html`

centric: this means that jobs are sent near the data they are going to process, not vice-versa, optimizing network usage and reducing transfer times. PROOF was (and still is) originally available as a standalone daemon (`proofd`), but using it as a xrootd plugin provides better integration with the data-centric model, basically because xrootd natively knows where its data is stored.

Configuring xrootd and cmsd for PROOF usage can be non-trivial: configuration files descriptions follow. The entire configuration files are reported in § E.3.

3.1.4.1 xrootd, cmsd, PROOF configuration file

The main configuration file (`vaf.cf` in our case, see § E.3.1) is shared between xrootd, cmsd and the PROOF plugin (`libXrdProofd.so`). Moreover, this file is read by every machine in the network: for this reason it is possible to use *if clauses* in order to activate certain configuration directives only on certain machines, or for certain daemons (for instance, the PROOF plugin is loaded only if xrootd reads the configuration file, but it is ignored by cmsd); wildcards can also be used.

The configuration file for the xrootd daemon substantially specifies the machine `proof.to.infn.it` as PROOF master and xrootd redirector, loads the PROOF plugin and specifies the paths that are part of the pool on each machine.

The `cms.*` directives specify various cmsd configuration parameters: we use them to specify which machines can connect to the pool (the PROOF head node and PROOF slaves only).

For the PROOF plugin we specify which version of ROOT will be used (the *trunk* is the only version available for now, but there's the possibility to add more than one version that the user can choose) and the authentication method: GSI authentication is used, as described later on. PROOF also needs a so-called "resource locator": this is an external file (an example is shown in § E.3.2) that specifies which hosts are available to the PROOF farm, and their type (master or worker). Even if this resource locator is considered "static", the file content is actually read when a user connects to PROOF, so that there is no need to restart xrootd in order to apply the new configuration.

3.1.4.2 Security through GSI authentication

GSI authentication is a form of authentication commonly used on the Grid that needs the user who wants to authenticate to present a valid *certificate* (that is sort of a "digital fingerprint" of the user) signed by a trusted *certification authority*, and a *key*. Certificates and keys must be in a format specified by the X.509 ITU-T standard; the key can be optionally locally encrypted using a passphrase.

Once the user has unlocked its private key, authentication is performed; if authentication succeeds, a proxy certificate signed with that key is created. The proxy certificate has the same subject as the user certificate, but with a `/CN=proxy` appended: they are temporary certificates used to authenticate on the Grid without entering the passphrase every time.

Other forms of authentication could have been used (such as *password authentication*), but GSI authentication has been chosen because physicists are already accustomed to it and because it is very secure, as long as the user keeps the key encrypted and in a secure place: the need to have a key eliminates the common security problem of password authentication, say the user chooses a trivial password easy to be found by a malicious user.

For instance, every physicist in ALICE that uses the Grid (or the CAF) knows that the steps in order to authenticate are:

1. place `usercert.pem` and `userkey.pem` into `~/.globus` directory and make them readable only by the user[3] *(only the first time)*;
2. launch `alien-token-init`;
3. type in your passphrase to unlock the key;
4. start using ALICE services such as PROOF or `aliensh`.

To use GSI authentication we must first instruct PROOF to load the security plugin `libXrdSec.so`, then configure the security plugin to use GSI authentication. When the user is authenticated, it must be assigned to a Unix user present on every computer in the facility: in other words, there must be user mapping between Grid users and local PROOF users.

User mapping can be statically done either by a file named `grid-mapfile` (see § E.3.3), which must be present even when empty, or else xrootd fails to load, or via a hardcoded mapping function (called the "gmapfun"), that can be loaded externally by another plugin.

As external function we use the `libXrdSecgsiGMAPLDAP.so` plugin, which simply calls the command-line utility `ldapsearch` that, given a certificate subject, searches for a corresponding user name on a LDAP server (in our case, the AliEn LDAP server). The `grid-mapfile` has precedence over LDAP (and over every other "gmapfun"). The configuration file can be found in § E.3.4.

Note that the token generated by `alien-token-init` is not propagated on every PROOF slave, so it is impossible without workarounds to access files on the AliEn File Catalogue (FC) (see § 1.4.2.1): we are currently working on a solution for this problem.

[3] `chmod -R 0400 ~/.globus`

3.1.5 VAF command-line utilities

Some command-line utilities have been written to manage the VAF by only accessing the head node: they are shell scripts which take care of common operations like controlling Xen resources on the whole farm, defining and activating resources profiles and controlling xrootd and cmsd daemons. These scripts use SSH to automatically and transparently access the single machines, whose names are in environment variables on the head node. A detailed description of these utilities can be found in § D.

3.1.6 Documentation and user support

User and administrator documentation is being kept on a website[4] as a Wiki hosted on a CMS named DokuWiki[5], because it is open-source, it does not need any database configuration and it stores pages as text files readable by any text editor.

3.2 Pre-production tests

At this point of the work we have tested the feasibility of the VAF approach in some specific cases, and we have implemented a small but scalable prototype of the VAF. By actually putting our virtual WNs into a standard Grid queue and stressing the interactive VMs we can test the VAF in a real production environment, and finally state the feasibility of our approach in real use cases.

Four virtual Grid nodes are actually in the LCG queue since late December 2008. Only two Grid nodes were in production until late November 2008: this is when the tests were done, so the Grid test with real jobs has been executed on two machines only out of four total available machines.

3.2.1 Test generalities

We are going to execute different tests and measurements both on the PROOF slaves and on the Grid WNs. These tests were executed in three different resources profiles, that are manually choosen by the `VafProfiles` script, that give the priority either to the PROOF node or to the Grid WN, as we are going to discuss later in the results.

[4] `http://newton.ph.unito.it/~berzano/w/doku.php?id=vaf:start`

[5] `http://www.dokuwiki.org`

3.2.1.1 Grid test

On the Grid nodes we execute tests with both real Grid jobs (mainly ALICE jobs) and "fake loads". Grid jobs are not under our control: the machines are in the Grid queue and they just execute what they are told to run by the CE. We have decided to run our tests in a period of time where ALICE jobs took most of the time, with respect to other Grid VOs, and the WNs are *never* idle, in order to test machine performances in the worst case. The "fake load" tests are actually not Grid tests: a CPU-intensive job (a primality test done with SysBench) is executed repeatedly in order to achieve intensive CPU usage.

Since real Grid jobs are very different in nature and execution time, no event rate is measured for them. An "event rate" is measured for the primality test instead, defined as the number of integers tested per second.

In real production Grid tests we also measure the number of jobs that fail to finish correctly on virtual nodes to see if it is greater than average.

3.2.1.2 PROOF test

A generic PROOF analysis is launched interactively while Grid WNs are using most of the system resources. Two different PROOF analysis are launched: one analyzes 100 evenly distributed files on two machines (16 workers) with the real Grid jobs; the other one analyzes 200 evenly distributed files on four machines (32 workers) with the primality test. Event rate is measured three times then averaged.

Data is evenly distributed across the nodes because of the data-centric nature of PROOF: if we uniformly distribute data, then also CPU usage is uniformly distributed. Since there are four machines, 50 files are stored on each one. These files are the same ROOT file replicated several times, in order to avoid timing inconsistencies because of different data on different machines.

3.2.2 Resource profiles and results

Results are plotted in Figure 3.4 (primality test on WNs) and Figure 3.5 (real Grid jobs on WNs). In each barchart the fourth column represents the reference event rate, that is the maximum event rate reached when no other load is run on the other machine; this event rate is set equal to 1: any other event rate represented there is normalized according to the reference event rate.

Since the primality test uses little memory, when shrinking the WN *swap does not occur*. Resources profiles are activated via the `VafProfile` script (see § E.4.5). In their description we conventionally treat each physical host like the only running guests on it were the PROOF node and the Grid WN, even if a small

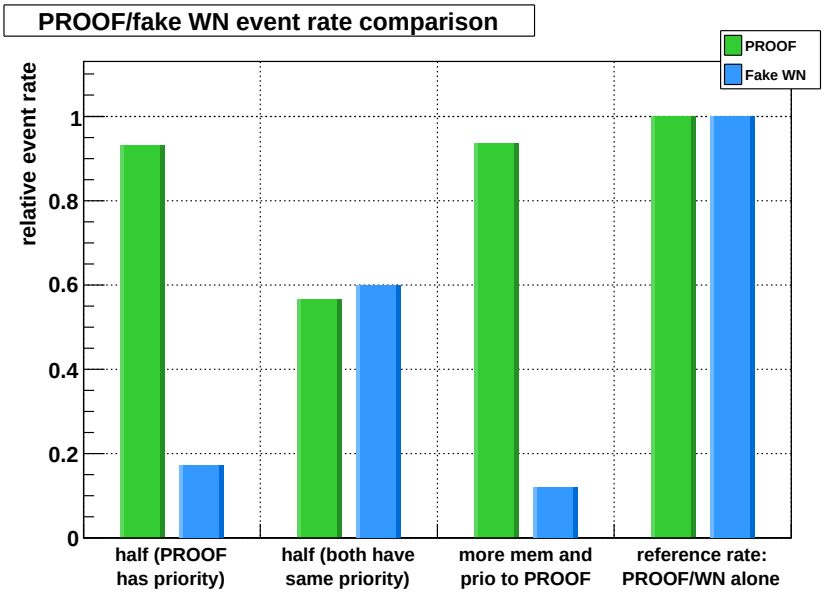


Figure 3.4: PROOF and WN normalized event rates with primality test (no real Grid job) on the WN: this test has been run on four machines.

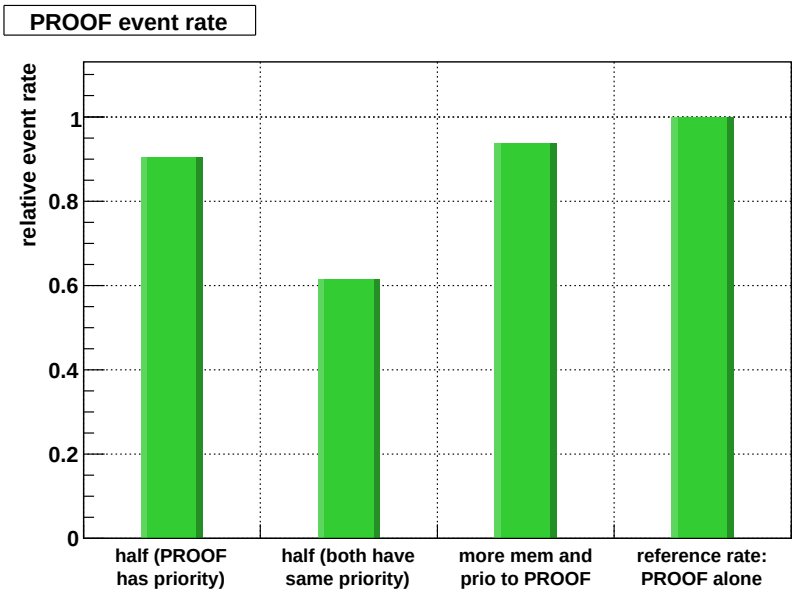


Figure 3.5: PROOF and WN normalized event rates with real Grid jobs (mostly ALICE jobs) on the WN: this test has been run on two machines.

amount of resources (512 MiB of memory) should be assigned to the dom0 guest too, that represents the physical host itself.

1. The **"halfmem-proof-prio"** profile gives half of the available memory to each node. PROOF has eight times the priority (the *weight*) with respect to the WN, and PROOF can use all the CPUs (cap=800%), while the WN only one (cap=100%). PROOF is always idle, but when it requests CPU power it behaves quite like it's the only application running on the whole physical machine, as we can see by looking at the first column of the plot.

2. The **"half"** profile is the same as before, but *both machines have the same priority*: indeed, the bars in the second column are almost the same height, proving that the Xen scheduler actually assigns about half CPU cycles each VM.

3. The **"proof"** profile assigns most of the resources to interactive analysis: PROOF is given ∼90% RAM and cap=800%, while WN is given ∼10% RAM and only one CPU (cap=100%); moreover, PROOF has (as in the "half" profile) eight times the priority with respect to the WN. This configuration behaves quite like the first one: the only difference between the two configurations is the memory assignment. By repeating this test with different PROOF analysis we can use it to tune the memory to assign to our analysis tasks: in our case we see that no more than 50% of total memory is needed, so the memory assigned with this profile is too much and underutilized for this particular analysis task.

For what concerns the test with real Grid jobs (Figure 3.5), the main difference with respect to the primality test is that they use much memory, thus when shrinking the WN *swap does occur*. Since the heterougeneous nature of Grid jobs, no event rate could be measured for the WN in this case. Despite of swap space usage, by comparing the green bars (those related to PROOF) with the ones on Figure 3.4 we see almost no difference: it is worth underlining that this result is obtained by physically using two different disks for the two VMs, and this result states that with this method we really obtain complete performance isolation, as we have been expecting.

3.2.2.1 PROOF scaling

An important and accessory measurement is made through our tests: by running the same test while increasing each time the number of cores per node from one to eight we measure the event rate (three times, then it's averaged). The results are plotted in Figure 3.6, where it's rather evident how the event rate

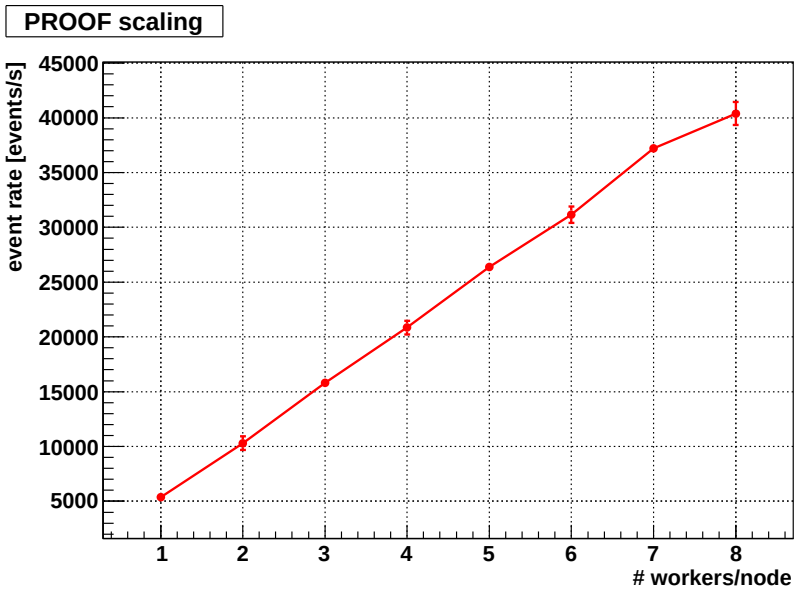


Figure 3.6: PROOF speed of execution (measured as events per second) increases linearly until the number of workers reaches the maximum possible (one worker per core).

increases linearly while increasing the number of workers. No overbooking has been considered in this test, i.e. no more than one worker per free core is used.

3.2.2.2 ALICE jobs CPU efficiency

The CPU efficiency of sole ALICE Grid jobs running on the two virtual WNs is measured and then plotted on Figure 3.7 with a blue dot corresponding to when the job finished its execution. Since each Grid job uses one core only the maximum CPU efficiency is 1. We measure CPU efficiency over a long period of time (∼60 hours) while changing the underlying resources (as labelled on the plot). Only the jobs that started and terminated inside the measurement window are considered. We also should note that jobs not only from the ALICE VO were executed during this period, but the vast majority were ALICE's.

We note that the more swap is used, the more the jobs turn from CPU-bound to I/O bound: this fact, in addition to the reduced cap to be shared by the same number of jobs, increases the average time needed to complete each job (that can be seen by the wider distribution of the blue dots) and decreases CPU efficiency. This is clearly visible in the third resource configuration (512 MiB of RAM and cap=100%).

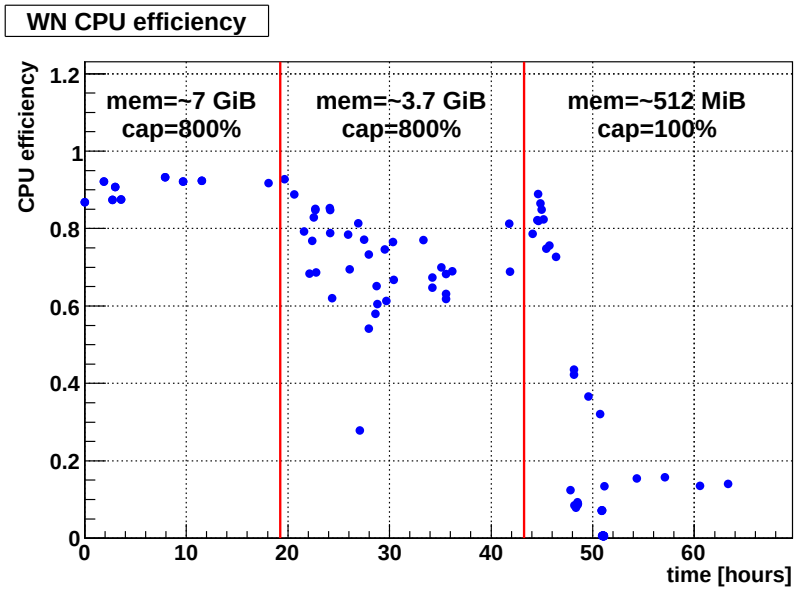


Figure 3.7: CPU efficiency by each ALICE Grid job whose execution started and terminated during interactive analisys stress test. Resources *(memory, cap)* assigned to the WNs vary and are indicated on the plot. Vertical red lines indicate that resources changing occurs there.

3.2.3 Storage methods comparison

A PROOF analysis task specific to ALICE has been run in order to compare two different storage methods.

The commonly used storage method is the PROOF pool, where the PROOF worker nodes are also part of a xrootd storage cluster not only for temporary files, but to store data that is to be analyzed. The method used at CAF is to ask the facility to "stage" (mirror locally) the data from the Grid storage: once the staging step is completed, the analysis can be run[6]. Local storage is supposed to be much faster than retrieving data from another SE, but maintaining a local storage of disks in addition to the Tier-2 SE may be unfeasible (for economic and administrative reasons): for these centres (just like Torino) it may be more feasible for PROOF to access data directly from the local storage element, and to possibly mirror there some data stored in some other place in the world that needs to be locally analyzed.

In this test, these two different methods are compared. The tests are executed three times: startup time and execution time are measured, then averaged. When the local pool is used, data is evenly distributed.

As we can clearly see in the results (Table 3.1 and Figure 3.8), getting data from the local SE is very slow (33.5× slower) compared to the local pool solution. The issue is probably the slow network configuration of the current VAF prototype: SE is accessed behind NAT and VAF hosts are connected to a heavily-used switch. This benchmark needs to be repeated with a better network configuration, maybe with a dedicated access to the SE.

	Startup [s]	**Duration [s]**	**Total time [s]**
PROOF pool	2.5 ± 0.1	24.3 ± 0.6	26.9 ± 0.6
Torino SE	204.0 ± 1.1	694.7 ± 20.8	898.7 ± 20.9
	80.5×	**28.5×**	**33.5×**

Table 3.1: Time taken to complete the analysis with two different storage methods: the PROOF pool and the SE. The last line shows how much accessing files on the SE is slower than on the PROOF pool.

[6] Actually, with PROOF it's possible to run the analysis even on partially-staged data.

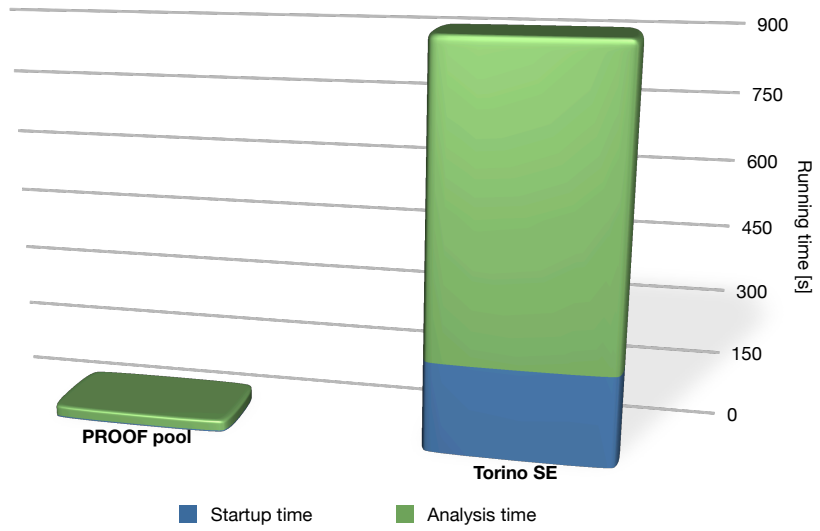


Figure 3.8: Time comparison of the same PROOF analysis task run on data stored with two different methods: on the local PROOF pool and on the Torino SE.

Chapter 4

PROOF use cases

4.1 Introduction

Although it should be clear that, in the ALICE computing model, the Grid is the preferred solution for both event simulation and reconstruction, all user analysis that comes after reconstruction can be done on PROOF.

After LHC runs and ALICE data acquisition, or after a simulation, the collected "raw" data need to be reconstructed. The interface for reconstruction provided by AliRoot consists of two classes: a steering class called `AliReconstruction`, responsible of calling the methods of several specific classes for each detector named `Ali<DET>Reconstructor` that inherit from the base class which is called `AliReconstructor`.

Output of a reconstruction are one or (usually) more ROOT files called Event Summary Data (ESD), which are usually named `AliESDs.root` and subdivided by number of run and number of job in several subdirectories.

User analysis is done on those ESDs: every relevant information to analyze the events can be obtained from there. Since ESDs also contain information that is not needed for every kind of analysis, a lightweight representation of the events can be obtained by filtering only the needed ESD data and producing the so-called Analysis Object Datasets (AODs): AOD files are usually called `AliAODs.root` and they are smaller in size, reducing the memory needed to do repeated analyses.

Although PROOF can be used to do virtually any kind of task, it was primarily designed and thus tested in order to run user analysis on distributed ESDs or AODs as well: these user analyses are called "analysis tasks", and their respective object-oriented implementation is the base class `AliAnalysisTask`.

4.2 PROOFizing a task

In this section we will see how a generic task can be run on PROOF, and which are the requirements in order to achieve task parallelization.

PROOF is not a tool that does real parallelization of a single task: on the countrary it can run several tasks simultaneously instead of sequentially. In other words, PROOF can parallelize all the kinds of tasks which solution can be formulated as a set of independent subtasks (Figure 4.1): this kind of parallel computing is called *embarrassing parallel*.

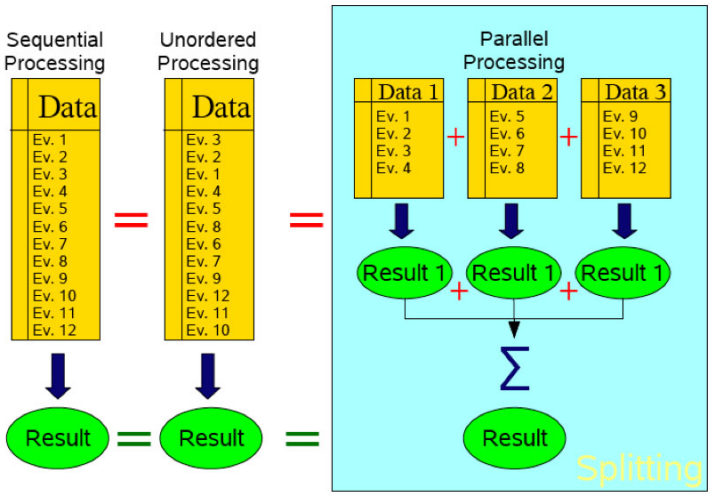


Figure 4.1: Difference between sequential (local) and parallel (PROOF) data processing: the comparison with the "unordered" processing is made to state that a requirement of embarrassingly parallelizable tasks is that their final result should be independent on the data processing order.

The file format used to hold events and analyses data is the standard ROOT file format: this format is capable of holding in an efficient way instances of every object (descendant of class `TObject`), an operation normally referred to as *serialization* or *marshalling*.

Since several independent tasks are run simultaneously, each task produces its own data: each PROOF worker is able to serialize output through the network to the master, which then merges several outputs in a single stream that is finally written on a ROOT file. ROOT is intelligent enough to use the right merging method according to the type of data container (the class) of the output stream – for instance, a `TH1` or a `TList` have merging functions already implemented, while custom classes should implement a `Merge(TCollection*)` method.

ROOT provides a class as an interface for every analysis task that reads entries from a `TTree`: this interface is called a `TSelector`[5]. An analysis written as a `TSelector` descendant can be run both sequentially on a desktop computer, or simultaneously on several PROOF workers, but the analysis code remains the same and does not need any change. The user can choose at runtime whether to run the analysis in local or on PROOF: this fact basically means that the physicist shouldn't make, in principle, any effort in order to port its code on an embarrassing parallel environment, by gaining greater computing power without the need to know a single complicated concept behind parallel computing.

As we have introduced before, the ALICE experiment Offline Framework provides a class for analysis called `AliAnalysisTask`[20]: every analysis task can have several inputs and several outputs connected. Even if this class is more specific to the ALICE experiment, its behaviour is more general, since it provides the possibility to process different inputs in other formats besides the `TTree` class. The idea behind the `AliAnalysisTask` remains the same: analysis code written for local usage should be re-used in PROOF without any modification.

As we are about to see, this is true for ESD processing but some code of AliRoot still needs to be adapted in order to exploit PROOF functionalities: luckily this is only a matter of implementation, not of design, since AliRoot's object-oriented features make it flexible enough for PROOF and any future evolution of it.

4.3 Example use cases

Very specific user analyses are beyond the scope of this thesis; however, in the following sections we will see the requirements of a task in order to be run on PROOF, a basic example of distributions from an ESD and a filtering from ESD to AOD via the PROOF located on the VAF.

4.3.1 Spectra from multiple ESDs

The first use case we analyze is an analysis task that reads several ESDs and produces two distributions (transverse momentum, p_t and energy) that are saved in a single output file.

ESD files are taken from an official Physics Data Challenge (PDC) production and are stored on the local PROOF pool via a Bash script. The files are located on AliEn under the following path:

```
alien:///alice/sim/PDC_08a/LHC08x/180110
```

which holds event data of simulated pp collisions with forced hadron decays and charm signal. The directory structure (`.../#run/#job/AliESDs.root`) is preserved when staging data on the PROOF pool. Data could also have been accessed directly from the local ALICE storage in Torino, but the network configuration currently in place does not yet allow a fast analysis with this data access model (see § 3.2.3). Each ESD contains 100 events: there are 967 files, for a total of 96700 events.

The analysis task has a single input container: a `TChain` created from ESD files listed line by line in a text file with an external macro (see § E.5.1). Event by event, transverse momentum (p_t) and energy (E) are collected from the reconstructed tracks and are stored in two output containers (two `TH1F` histograms) saved in a single ROOT file. At the end of the analysis task, the plots are printed on screen on the client machine, as in Figure 4.2(a) and (b). Header file and implementation of this task (called `AliAnalysisTaskDistros`) are reported in § E.5.2.1 and § E.5.2.2.

The analysis is run by a single caller script (§ E.5.3) that can be told to run the analysis either locally (on one core) or on PROOF (on 32 workers): timing measures are made three times then averaged for each case. Timings are reported in Table 4.1 and plotted in Figure 4.3.

	Run time [s]	**Error [s]**	**Error [%]**	**Rate [evt/s]**
PROOF	26.8	0.4	1.6	3608.2
Local	964.7	28.3	2.9	100.2

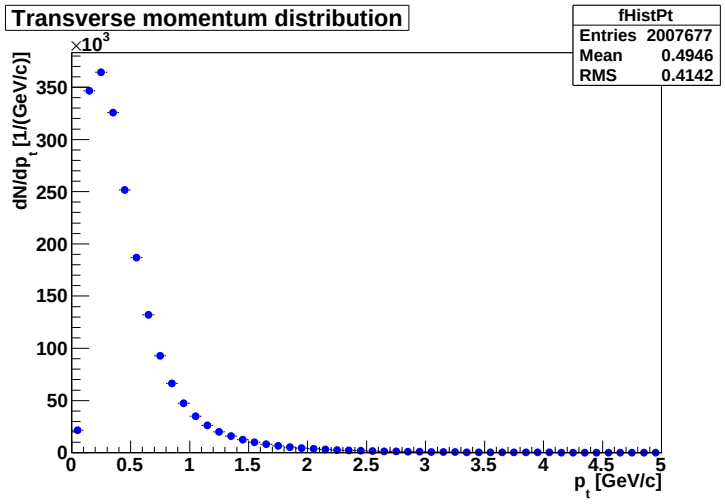
Table 4.1: Local versus PROOF measurements for the spectra analysis task: errors are calculated from measurements repeated three times.

Timing measurements were made on the sole analysis, ignoring the time required for `TChain` creation. Two different (but comparable) methods were used to measure the running time on local and on PROOF.

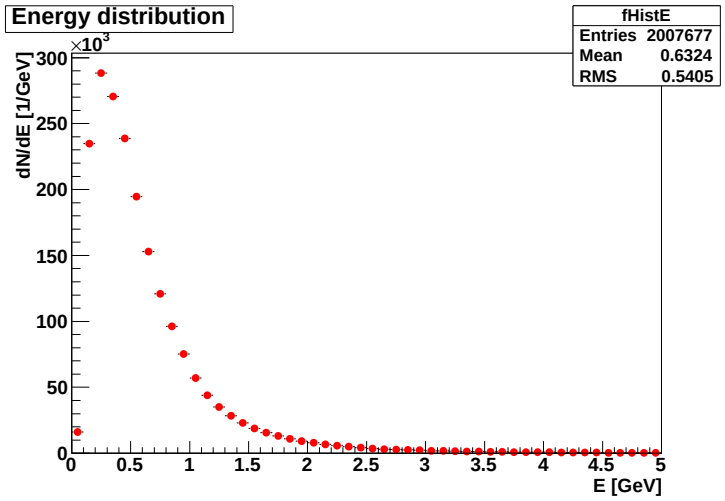
- On PROOF, timing is subdivided in a "startup time" that PROOF uses to know where data is located, and a "running time", that is the effective processing time: PROOF prints out the timing on a graphical console (Figure 4.4.) In this case, total time is calculated as the sum of the two times.

- Locally, total timing is directly calculated by using a `TStopwatch` object[1].

[1] `http://root.cern.ch/root/html/TStopwatch`



(a) p_t distribution.



(b) E distribution.

Figure 4.2: Two sample spectra obtained from a simple ALICE analysis task run on 96700 events.

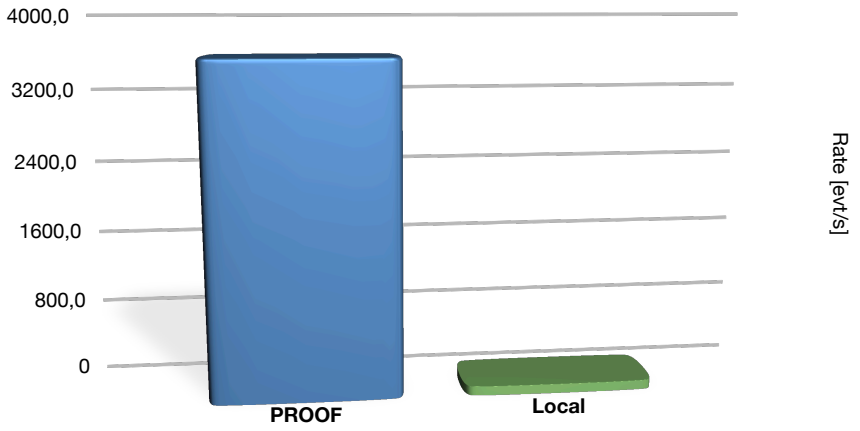


Figure 4.3: Local versus PROOF event rate *(higher is better)* on a simple ALICE analysis task that fills two spectra from 96700 events distributed in 967 ESD files: PROOF is shown to be 36 times faster, by using 32 workers.

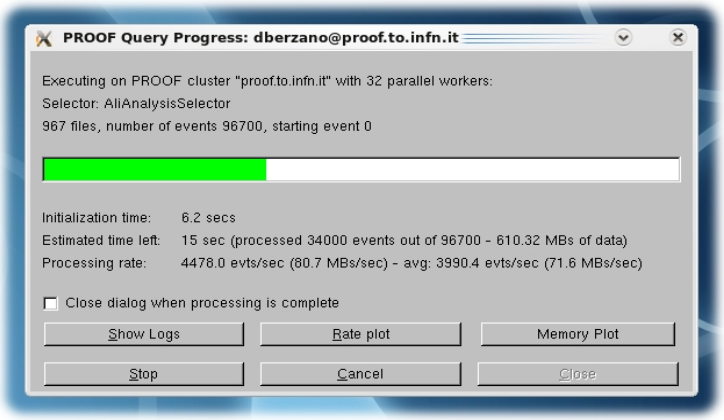


Figure 4.4: The graphical PROOF console while running an analysis task on the VAF.

Results of the timings are clear: as expected, PROOF analysis is much (36×) faster than local analysis. The only effort for the physicist was to substitute this line of code:

```
mgr->StartAnalysis("local", chainESD);
```

with this one:

```
mgr->StartAnalysis("proof", chainESD);
```

plus the commands to enable PROOF and the required libraries on each node.

4.3.2 Multiple ESDs filtering to a single AOD

Analysis Object Datasets (AODs) are compact representations of ESDs: while ESDs hold every event information obtained after the reconstruction, AODs hold only the information needed for a specific type of analysis, thus using less memory.

The process of creating AODs from ESDs is called "event filtering". This kind of task is usually not run on PROOF but rather on the Grid, because it's rather slow if run sequentially on a desktop computer: however, since the classes responsible of filtering ESDs (called `AliAnalysisTaskESDfilter` and other similar names) inherit from `AliAnalysisTask`, it is in principle possible to PROOFize them, enabling physicists to produce their AODs immediately, without waiting for them to be run on the Grid.

A common "analysis train" (a set of different analyses run sequentially on the same dataset) was slightly modified in order to be run on PROOF. The filters applied to the ESDs are:

- the standard ESD filter;
- the muon filter;
- the PWG3 vertexing.

The analysis train has been run on the same 967 ESDs (holding 96700 events globally) as the previous analysis task, producing a single (and rather big, ∼ 90 MiB) AOD. Even if producing a single and big AOD it's rather inefficient because it makes subsequent AOD analysis non-embarrassingly parallelizable, it is trivial to modify the train code in order to run several times the filters on different ESDs by producing several AODs.

In analogy with the spectra analysis task, in this analysis train we measure the time taken for the task to complete with a `TStopwatch` in the local case and

	Run time [s]	Error [s]	Error [%]	Rate [evt/s]
PROOF	61.2	4.4	7.2	1579.2
Local	1308.3	39.6	3.0	73.9

Table 4.2: Local versus PROOF measurements for the analysis train that produces one AOD from several ESDs: errors are calculated from measurements repeated three times.

by reading startup and analysis time from the graphical console (see again Figure 4.4) in the PROOF case. Analyses are repeated three times, then the timings are averaged and event rate is calculated. Results are positive even in this case, as expected: as we can see in Table 4.2, PROOF with 32 workers perform 21.4 times faster than sequential analysis. Event rate is reported in Figure 4.5.

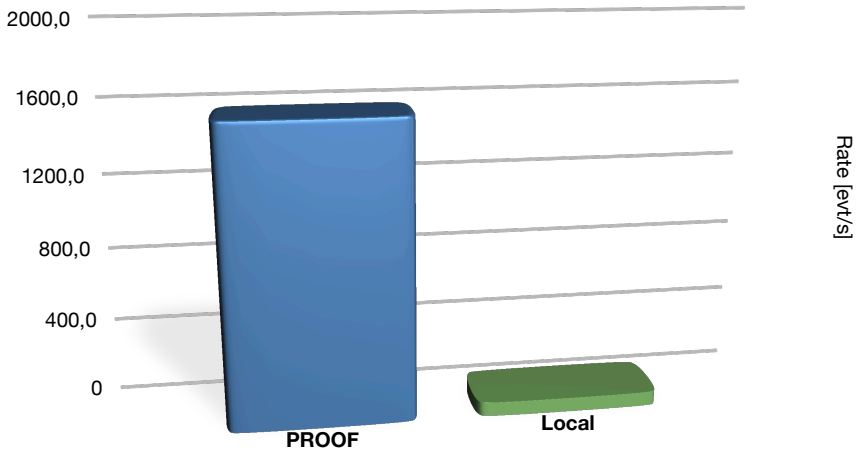


Figure 4.5: Local versus PROOF event rate *(higher is better)* on an analysis train that filters 96700 events distributed in 967 ESD files in order to generate a single big AOD: PROOF is shown to be 21.4 times faster, by using 32 workers.

4.4 Reconstruction of an open charm decay

In this section we will show how much a real CPU-intensive Physics analysis gains speed if executed on PROOF instead of locally. The analysis is a reconstruction of secondary interaction vertexes that selects candidate tracks for a specific open charm decay: $D^+ \to K^-\pi^+\pi^+$.

4.4.1 Open charm physics

Heavy quarks, like charm and bottom, are produced in the early stages of high energy nucleus-nucleus collisions and their lifetime is longer than the expected lifetime of the QGP. This makes open charmed mesons a powerful tool to investigate nuclear effects on charm production, propagation and hadronization in the medium produced in heavy-ion collisions at high energy.

In addition, measurements of mesons with open charm and beauty provide a natural normalization for charmonia and bottomonia production at the LHC. At the SPS the Drell-Yan process ($q\bar{q} \rightarrow \mu^+\mu^-$) was used as reference signal for charmonia normalization because it scales with the number of binary collisions and it is independent on nuclear effects; at LHC energies the Drell-Yan signal is expected to be completely shadowed into dileptons from semi-leptonic decays of open charm and open beauty even when energy loss of heavy quarks is assumed.

Measurement of charm and beauty production in *proton-proton* and *proton-nucleus* collisions is of great interest too, not only because it provides a baseline to compare the results obtained with nucleus-nucleus collisions, but also because it provides important tests for QCD in a new energy domain.

4.4.1.1 Heavy quark production in *pp* collisions

The main contribution to heavy-quark production in pp collisions is thought to come from partonic hard scatterings. Since the heavy-quarks have a large mass ($m_Q > \Lambda_{QCD}$), their production can be described by Perturbative QCD (pQCD).

The single-inclusive differential cross section for the production of a heavy-flavoured hadron can be expressed as follows[9]:

$$\left.\frac{d\sigma}{dp_T}\right|_{AA\rightarrow H_Q X} = \sum_{i,j=q,\bar{q},g} f_i \otimes f_j \otimes \left.\frac{d\hat{\sigma}}{d\hat{p}_T}\right|_{ij\rightarrow Q\bar{Q}} \otimes D_Q^{H_Q}$$

where the meaning of the terms follows.

- $f_i = f_i(x_i, \mu_F^2)$ is the Parton Distribution Function (PDF): it gives the probability of finding a quark or a gluon of type i carrying a momentum fraction x_i of the nucleon; μ_F is the factorization scale. The evolution of PDFs in QCD is governed by the parameter α_s/π: once the initial conditions of the nucleon structure are experimentally determined through the measurement of deep inelastic cross-section at a given Q^2, the PDFs can be predicted by pQCD at higher Q^2.

- $d\hat{\sigma}/d\hat{p}_T{}^{ij\rightarrow Q\bar{Q}}$ is the partonic cross section: this term can be described by pQCD, since it is related to interactions of partons at high Q^2.

- $D_Q^{H_Q}$ is the fragmentation function which represents the probability for the scattered heavy quark Q to materialize as a hadron H_Q with momentum fraction $z = p_{H_Q}/p_Q$. The fragmentation functions can be experimentally determined by measuring jets in e^+e^- collisions.

A precise measurement of the heavy-quark cross section cannot be done directly from the measurement of the total cross section for heavy-flavoured hadrons, because there are several experimental constraints, like the restricted detector acceptance, which do not allow the measurement of total cross sections. Nevertheless, since both the parton distribution functions and the fragmentation functions are well measured, they can be used to determine the heavy-quark cross section starting from the measured heavy-flavoured hadrons.

4.4.1.2 Heavy quark production in *AA* collisions

Heavy-quarks are produced on a time scale of $\sim \hbar/(2m_Q c^2) \simeq 0.08$ fm/c (for $c\bar{c}$ production) from hard partonic scatterings during the early stages of the nucleus-nucleus collision and also possibly in the QGP, if the initial temperature is high enough. No production should occur at later times in the QGP, whose expected τ is 10 fm/c, and none in the hadronic matter. Thus, the total number of charm quarks should get frozen very early in the evolution of the collision, making them good candidates for a probe of the QGP.

The total inclusive cross section corresponding to a given centrality interval ($0 < b < b_c$) for a hard process involving two nuclei A and B is expressed as a function of the impact parameter b_c:

$$\begin{aligned} \sigma_{AB}^{hard}(b_c) &= \int_0^{b_c} db\, 2\pi b \left[1 - (1 - \sigma_{NN}^{hard} T_{AB}(b))^{AB}\right] \\ &\simeq \int_0^{b_c} db\, 2\pi b \cdot AB \sigma_{NN}^{hard} T_{AB}(b) \end{aligned}$$

where:

- σ_{NN}^{hard} is the cross section for a given hard process, considered small in the approximation of the last passage;
- $T_{AB}(b)$ is the thickness function, related to the overlapping volume of the two nuclei at a given collision impact parameter b.

Several effects can possibly break the binary scaling; these effects can be divided into two classes: *initial-state effects*, such as nuclear shadowing, which affect the heavy-quark production by modifying the parton distribution functions

in the nucleus (there are predictions that these effects may even occur in pp collisions at the LHC energies), and *final-state effects*, due to the interactions of the initially produced partons in the nuclear medium, which affect the heavy-flavoured meson production by modifying the fragmentation function (energy loss of the partons and development of anisotropic flow patterns are among such processes).

4.4.2 Reconstruction of $D^+ \to K^-\pi^+\pi^+$

A full reconstruction of the $D^+ \to K^-\pi^+\pi^+$ open charm decay is performed, using particle identification, tracking and vertexing: the analysis focuses particularly on secondary vertices finding.

4.4.2.1 Candidates selection strategy

The selection strategy for the $D^+ \to K^-\pi^+\pi^+$ (Figure 4.6) decay candidates is based on the invariant mass analysis of fully reconstructed topologies originating from displaced vertices[23].

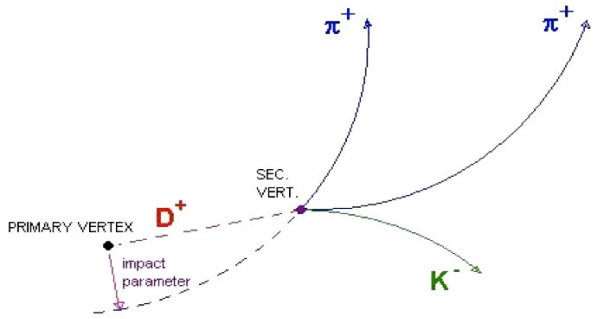
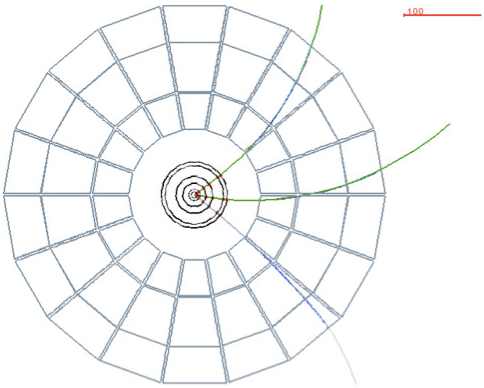


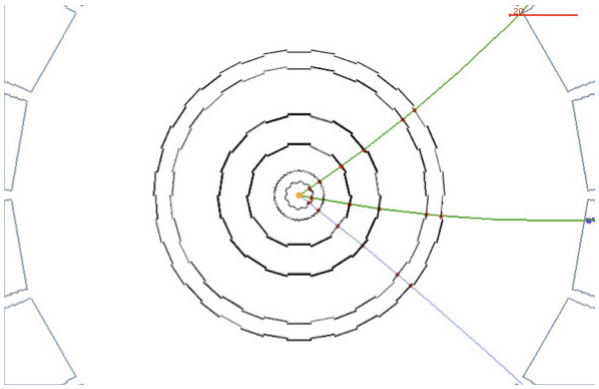
Figure 4.6: The $D^+ \to K^-\pi^+\pi^+$ decay, showing primary and secondary interaction vertices and tracks of the decay products.

The three ALICE detectors involved in an open charm decay reconstruction are the ITS that performs both tracking and vertexing, the TPC dedicated to particle tracking and the TOF that identifies k and π: the decay is reconstructed from the outside in, as for the sample event displayed in Figure 4.7.

The data collected by these three detectors are used at first to build triplets of tracks with correct combination of charge signs and large impact parameter; then, each track is tagged with particle identification.



(a) TPC level.



(b) ITS level.

Figure 4.7: Event display of a $D^+ \rightarrow K^-\pi^+\pi^+$ decay, from the point of view of the tracking and vertexing detectors involved, from the outside in.

From the tagged and tracked triplets, the secondary vertex is calculated using the Distance of Closest Approach (DCA) method; in the end, the good pointing of the reconstructed D momentum to the primary vertex is verified.

The whole reconstruction method is not a complex algorithm, but all the different triplets must be considered per each event, involving a very large number of iterations: this is the reason why the execution time of a reconstruction of several ($\approx 10^5$) events is expected to take hours to complete.

The algorithm must be optimized in order to choose the cuts that candidate tracks must pass in order to be taken into account in the final invariant mass distribution: if this analysis is run locally or on the Grid, as it is largely done at this time in ALICE, optimization is a process that may take days or weeks, while a PROOF analysis may potentially reduce of an order of magnitude or two the required time, as we are about to see.

4.4.2.2 Analysis results

The analysis runs on ESD files: if reconstructed tracks pass an appropriate cut they are written as candidates in a AOD file file specific for Heavy Flavour vertexing (namely `AliAOD.VertexingHF.root`). This file contains information in a `TTree` that can be attached as a "friend tree" to a standard AOD. This analysis can be found in the `PWG3/vertexingHF` subdirectory of the AliRoot framework: the class that accomplishes the analysis is a standard analysis task, and it is called `AliAnalysisTaskSEVertexingHF`.

Currently, the example macro that runs the analysis does not implement a PROOF mode, since it is designed to run either locally or on the Grid – the latter solution being *non-interactive*: the macro was modified in order to run the analysis on PROOF and to build a `TChain` of several ESDs read from a file list in text format with one ESD path per line. As stressed before, no change in the analysis task was necessary, making it easy to run the analysis on PROOF even without specific knowledge about ideal parallel computing.

Vertexing of heavy flavours is a real testbed for PROOF. From the computing resources point of view it is very CPU and memory intensive: on the VAF configuration there are currently four machines with eight PROOF workers each, meaning eight parallel ROOT sessions per machine that fill up the memory, a situation that may rapidly lead to system instability and crashes. The other issue is that many data must be sent through the network in order to be finally assembled by the PROOF master, and network issues may also negatively affect the stability of PROOF by leading to locks and corrupted final data. In other words, this real Physics application is useful to test PROOF stability, to check final data integrity and to benchmark speed improvements with respect to the same analysis run locally.

ESDs used in this analysis are taken from a run of an official pp Grid production with "forced" charm, a simulation directive that guarantees the presence of roughly one $D+$ per generated pp collision; 97600 events distributed in 976 files were analyzed. The analysis has been run twice on these files: locally and on PROOF. Speed benchmarks and proper comparisons between local and PROOF analyses are found in Table 4.3 and Figure 4.8: we can see that in this real use case PROOF is ∼31 times faster. Scaling is almost linear, because we took care of the high memory requirements of vertexing analysis: we allocated 7 GiB on three hosts (24 workers) and 15 GiB on one host (8 workers); moreover, *none of the workers crashed during the test*.

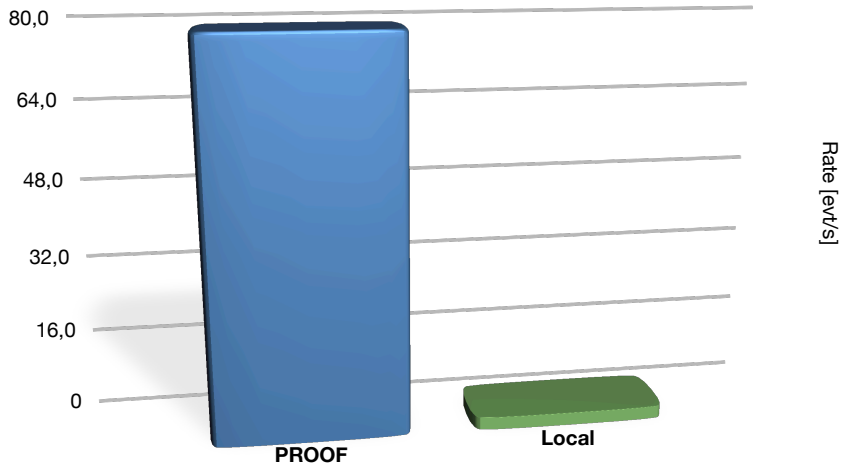


Figure 4.8: Local versus PROOF event rate *(higher is better)* on an analysis task for the Heavy Flavour vertexing.

	Run time [s]	**Rate [evt/s]**
PROOF	1221.0	79.2
Local	37620.0	2.6

Table 4.3: Local versus PROOF measurements for the Heavy Flavour vertexing.

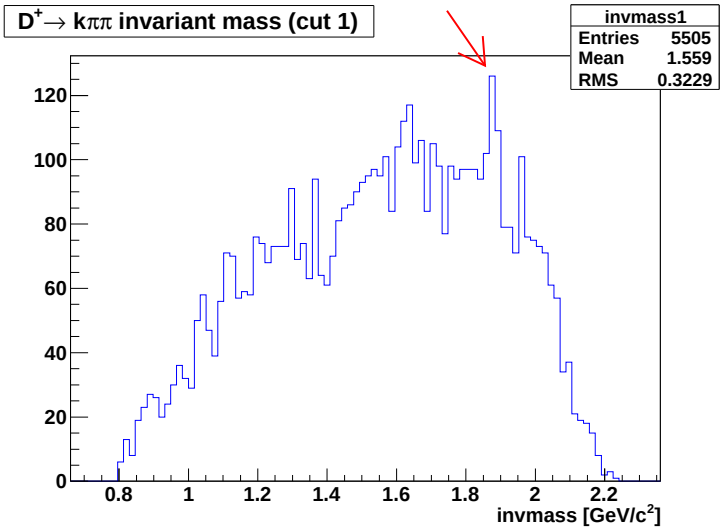
In order to check data integrity and results significance, a macro that plots the invariant mass distribution for the candidates of the $D^+ \to K^-\pi^+\pi^+$ decay is run: data is taken from the `TTree` in the output AOD, and no background subtraction is done neither during the analysis nor during the histogram filling. The histogram was plotted both on the locally and on the PROOF produced AOD.

The invariant mass distribution obtained is represented in Figure 4.9(a) and (b), with two different set of cuts choosen in order to highlight the D^+ peak at 1869.62 MeV/c^2: the same results, as expected, have been obtained with PROOF and local analysis.

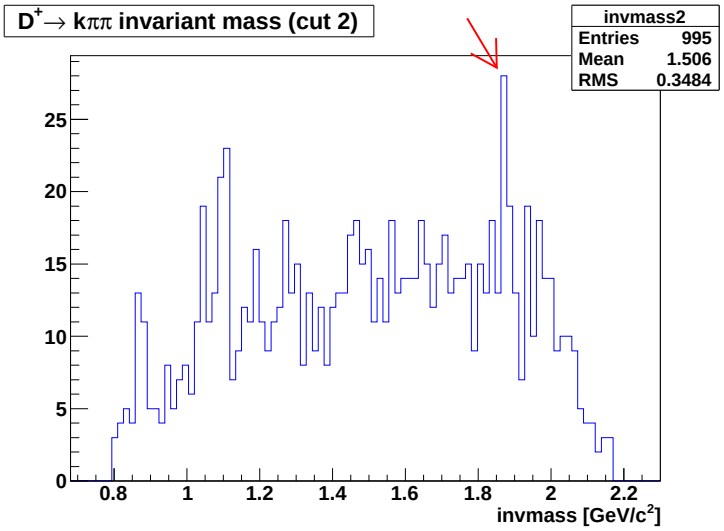
We should note that the statistics of the number of events choosen is not large enough to obtain evident D^+ peaks; moreover, as we have already noted, no background signal subtraction is performed at this time. Nevertheless, the D^+ peak can be spotted, and should be compared with a more complex analysis (with more statistics and background subtraction), which gives as output the very clean distribution shown in Figure 4.10.

For its nature, this analysis should be considered an example of what can be done on PROOF and the most important results must be underlined: there's a very big speed improvement in PROOF analysis, and the final results are the same, showing the stability and scalability of this specific analysis task.

Therefore, PROOF is shown to be suitable and advisable for analysis optimizations because of its robustness and great speed enhancement.



(a) First set of cuts: more statistics, peak less evident.



(b) Second set of cuts: less statistics, peak more evident.

Figure 4.9: Invariant mass distribution generated from the candidates for a $D^+ \rightarrow K^-\pi^+\pi^+$ decay. D^+ has a mass of 1869.62 MeV: the peak corresponding to this mass is indicated with an arrow in both figures. The number of events processed in this analysis is not enough to achieve both large statistics and an evident D^+ peak.

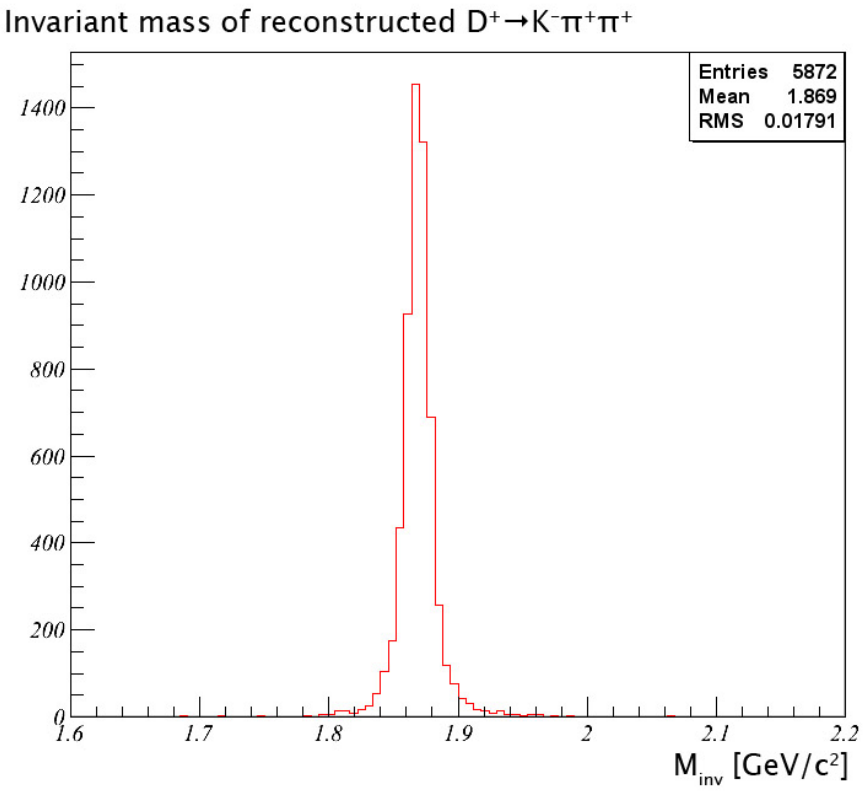


Figure 4.10: $D^+ \rightarrow K^-\pi^+\pi^+$ invariant mass distribution with background subtraction and a high statistics: this distribution is a reference for our analysis.

Chapter 5

Conclusions

Since real data taking from LHC is expected to start in a very short time, the demand for an infrastructure dedicated to interactive and chaotic user analysis is growing: our benchmarks on the prototype of the Virtual Analysis Facility demonstrate that virtualization could constitute a valid method of sharing the same computing resources between a largely accepted computing model (the Grid) and a complementary one without any modification in the existing standards.

5.1 Achieved results

In summary, our feasibility tests **(§ 2.2)** show that paravirtualization (and, in principle, any other form of hardware-accelerated virtualization) does not relevantly affect the performance of typical HEP jobs, namely *simulation*, *reconstruction* and *chaotic analysis*, as far as job performances are not I/O bound; and, even in that case, since most jobs do I/O to and from remote storage, disk performance loss shouldn't be an issue. Moreover, the feature of dynamically reassigning memory and CPU to virtual hosts introduced by Xen was heavily tested and proven to be stable **(§ 2.3)**. For their intrinsic generality, our tests have a scope that also goes beyond HEP applications and show that code quality of virtualization solutions has reached a mature and stable state.

The VAF is at this time a *working prototype*: four virtual Grid WNs are already in production since late December 2008 and they constantly work without any failure above the average. VAF implementation was designed to be scalable **(§ 3.1)** through a mass operating system installation procedure and several ad-hoc utilities: the installation of a new host only requires a few clicks on a graphical web interface (Cobbler) and adding the new hostname to an environment variable on the head node. An incremental backup solution automatically

backs up head node's disk in order to revert it to a previously working state in case of failure. An extended documentation is maintained directly on the Web as a wiki that may be useful in the near future as a reference and a support for both system administrators and VAF users.

By implementing a prototype we were also able to test Xen and hardware behaviour in a real production environment, with both Grid and PROOF running, and potentially concurring for the same resources: these tests show that *real performance isolation* is achieved even under heavy loads **(§ 3.2)** through Xen global accounting policy, so that the WN and the PROOF slave do not interfere with each other.

The VAF implements an authentication method, the GSI certificate authentication **(§ 3.1.4.2)** widely used in the ALICE experiment and well-known by end users: they don't need to learn any particular new authentication method, nevertheless strong security and access limitation has been achieved.

Differently from the PROOF On Demand (POD) solution (formerly known as *gLitePROOF*), the VAF is feasible on every small-sized (namely, Tier-2) ALICE computing centre; moreover we can easily achieve real performance isolation between the Grid WN and the PROOF slave, and also perfect sandboxing, by combining an hypervisor with physical disks separation.

Lastly, the facility was used in order to measure speed improvements gained through PROOF analysis compared to local analysis: these improvements are shown to be very high **(§ 4)**. Particularly, the AliRoot macro used to determine the secondary vertex for heavy flavours was successfully adapted in order to run the vertexing algorithm on PROOF: as an example use case of the algorithm, a secondary vertex was reconstructed for the open charm $D^+ \to K^-\pi^+\pi^+$ decay candidates, showing an invariant mass distribution with a peak in correspondence to the D^+ mass **(§ 4.4)**.

5.2 Outlook and future improvements

A couple of issues need to be solved before opening it to real users, both related to the data access model we intend to use – reading data directly from a SE already in place for ALICE in Torino. The first issue is the network configuration: the NAT between the SE and the VAF needs to be bypassed, and probably the best solution would be a dedicated network bone for our purposes. The PROOF model was primarily built to access data staged on the local pool and any other approach is necessarily slower: if we want to access directly the SE we should remove any bottleneck that may negatively affect network performance. The second issue is related to how the Grid proxy is used to access data on the Grid: when using PROOF there are several workers that independently access the AliEn

storage, each of them needing its own proxy. PROOF does not propagate the proxy certificate on the workers, at the moment, so we should find a solution to allow workers authentication to the AliEn File Catalogue: a workaround would be to manually propagate the certificate directly from the user macro used to launch a custom analysis task, while a complete solution involves changes in the ROOT upstream code.

Although the utilities used to manage the VAF are stable and widely used it would be better to rewrite them in order to support *several virtualization methods*: a valid abstraction layer that interacts with several hypervisors is libvirt[1], which also has Python bindings, a feature that might be useful if we want to rewrite our utilities in a more suitable language than shell scripting. Also, the Web interface needs further developement.

At this time resources are statically assigned to the PROOF nodes via a manual command: a better approach would be to "turn on" dynamically the facility when users need it, without neither system administrator nor user intervention. This kind of approach probably requires the PROOF Scalla plugin to be modified in order to trigger an event when a user connects to PROOF.

Lastly, highest priority must be currently given to the data access model because once completed we are allowed to open the VAF to potential users which will give a realistic feedback on their analysis experience.

[1] http://libvirt.org/

Appendix A

SysBench command-line parameters

Command-line parameters to run and repeat the SysBench tests run in § 2.2.2 follow.

A.1 CPU benchmark

The test is run with `<threads>` varying from 1 to 64, with a step of 1. The maximum prime number tested is 20000.

```
sysbench --test=cpu --cpu-max-prime=20000 --num-threads=<threads> run
```

A.2 Threads benchmark

Understanding the threads SysBench benchmark requires a little bit of knowledge on how threads scheduling work. A shared resource that can be accessed by one thread at a time is called a *mutex* (from *mutual exclusion*). A SysBench iteration in this kind of benchmark consists of a cycle of lock/yield/unlock.

A *lock* occurs when a thread takes ownership of a resource: the resource is then locked, meaning that no other thread can access it. On the countrary, *unlock* is when a mutex is freed. A *yield* occurs when a thread explicitly yields control to the part of the implementation responsible for scheduling threads: this is done to avoid *deadlocks*[1] (that occur when two or more threads are waiting for each

[1] See `http://en.wikipedia.org/wiki/Deadlock` with exhaustive examples.

other to unlock a resource) and may have the effect of allowing other threads to run: in other words, each thread must explicitly have a "breakpoint" that gives control to the thread scheduler, because the latter is unable to interrupt a thread on its own initiative. In any case, SysBench threads benchmark avoids deadlocks.

This test is run with 1000 iterations and 8 mutexes, the default values, with a variable number of threads.

```
sysbench --test=threads num-threads=<threads> run
```

A.3 Mutex benchmark

This test is run with the default parameters, that are listed below.

- number of mutexes: 4096
- number of mutex locks per each request: 50000
- number of null iterations before acquiring the lock: 10000

```
sysbench --test=mutex --num-threads=<threads> run
```

A.4 Memory benchmark

SysBench provides both read and write benchmarks: we are interested in write benchmark, since it is supposed to be the slowest. An amount of 5 GiB of random data is written with `<threads>` varying from 1 to 64 with a step of 1.

```
sysbench --test=memory -num-threads=<threads> \
  --memory-total-size=5G --memory-oper=write run
```

A.5 Database benchmark

This test performs several queries on a MySQL database locally installed. The database is accessed with the proper username and password. The test must be run three times: the first time *(prepare mode)* creates the proper records on the database; the second time *(run mode)* performs several queries to get the records from the database; the last step *(cleanup mode)* deletes the records. Table must be created and dropped manually.

Time is measured from run mode only, that by default performs `SELECT` queries. SysBench is told to perform 10^5 queries on 10^6 records, with a variable number of concurrent threads.

```
sysbench --test=oltp --mysql-user=<user> --mysql-password=<pass> \
  --max-requests=100000 --oltp-table-size=1000000 \
  --num-threads=<threads> [prepare|run|cleanup]
```

A.6 File I/O benchmark

File I/O benchmark is run both in sequential read (`seqrd`) and in sequential write (`seqwr`) mode on a default value of 128 files for a total size of 5 GiB. This test, just like the database benchmark, has a "prepare" mode that creates the files, a "run" mode when files are read or written and a "cleanup" mode where files are finally deleted. Time is measured on the run mode only.

Unlike the other tests this is executed with a fixed number of threads choosen to be 10 in order to simulate one process per core (our machines have eight cores) plus a few eventually overcommitted jobs or ancillary services (such as swap).

```
sysbench --test=fileio --file-total-size=5G \
  --file-test-mode=[seqrd|seqwr] \
  --num-threads=10 [prepare|run|cleanup]
```

Note that all NFS exports are done with this simple `/etc/exports` file, that contains no particular configuration:

```
/var/vafexport  192.168.1.0/24(rw,sync,root_squash)
```

Appendix B

Linux monitoring and memory internals

B.1 The `proc` virtual filesystem

The Linux `proc` filesystem is a virtual filesystem where files and directories contain information about running processes and system resources. Altough the proc filesystem year after year has become a container for several information that are not inherent to running processes, despite of its name (e.g. the /proc/bus/usb subtree used to contain information about the connected USB devices), its main structure is to contain some files with global system information within the root tree, and several directories named after the pids of running processes, each of them containing specific process information.

The virtual `proc` filesystem is an useful and easy to use interface to obtain information within a shell or a shell script, because the information is usually returned as text, not binaries.

Many Linux process information utilities use the `proc` filesystem to print out information: this is the case, for example, of `top`, `ps` and `uptime`. Knowing the `proc` filesystem is useful for performance monitoring mainly because directly accessing the information from the virtual files is a less time-consuming task than retrieving information about every process and then parsing out only the processes we need to monitor.

A complete description of what the `proc` filesystem does is available, as usual, on the Linux manpages[1]. However, since we are going to use this method for per-

[1] `man 5 proc`

Field no.	Name	Description
1	uptime	time since boot [s]
2	idle time	time spent in idle cycles since boot [s]

Table B.1: `/proc/uptime` fields and their description.

formance monitoring, a brief description of the most relevant virtual files follows.

B.1.1 `/proc/cpuinfo`

This file contains information about all available system CPUs. The output is in a human-readable form, with output description. To simply obtain the number of available cores on the system, we can cast:

```
grep -c processor /proc/cpuinfo
```

B.1.2 `/proc/meminfo`

As the name suggests, this file contains information about every system memory, from RAM usage to caches. Output is similar to `cpuinfo`: it is human-readable, with fields description. The most relevant fields are:

- `MemTotal`: total RAM installed on the system [kiB]
- `MemFree`: free RAM [kiB]
- `SwapTotal`: total swap space [kiB]
- `SwapFree`: available swap space [kiB]
- `Cached`: the buffer cache [kiB]
- `SwapCache`: the swap cache (see below) [kiB]

A more detailed explanation on how Linux caches work can be found later on (see § B.3).

B.1.3 `/proc/uptime`

This file contains only two numerical fields, separated by a space. The list of fields is reported in Table B.1.

Field no.	Name	Description
1	pid	process id
3	state	one of RSDZTW[2]
14	utime	user-mode time [jiffies]
15	stime	kernel-mode [jiffies]
22	starttime	when the proc. was started since boot

Table B.2: Most relevant fields from `/proc/[pid]/stat`.

B.1.4 `/proc/[pid]/stat`

This virtual file contains status information about the process to which the pid refers. There are many space-separated fields, that can be easily parsed (e.g. by using `awk`). The most relevant fields from the `stat` file are reported in Table B.2.

B.2 Jiffies

In Computer Science's jargon, a *jiffy* is a time measurement unit. Its value isn't absolute, and depends on the system. System interrupts are handled every jiffy, so 1 jiffy = system interrupt timer tick. Equivalently, process scheduling occurs every jiffy [3].

On every computer used in these tests, 1 jiffy = 0.01 s. This is hardcoded at kernel level and can be changed by recompilation.

In recent Xen versions system clock synchronization between the physical host and domUs is very precise, meaning we don't need to manually synchronize system clocks in VMs with a NTP server.

However, there is only one global credit scheduler running on dom0, whose task is to schedule every process on every domU (and dom0 itself of course), according to a "maximum VCPU usage" given in percentage (where, e.g., 200 means two VCPUs), called the *cap*, and a "relative priority" called *weight* (where a VM with a weight=24 gets twice priority over a VM with a weight=12).

VCPUs in VMs count ticks, but with a global scheduling policy there is no exact correspondence between a physical CPU and a VCPU. As a first result, ticks in VMs last about 0.01 s, with some small glitches that become evident when

[2] **R**: running; **S**: sleeping in an interruptible wait; **D**: waiting in uninterruptible disk sleep; **Z**: zombie; **T**: traced or stopped (on a signal); **W**: paging.

[3] `http://en.wikipedia.org/wiki/Jiffy_(time)#Use_in_computing`

monitoring the CPU usage from within the VM. This is because the Xen scheduler moves the system ticker from one physical CPU to another, transparently for the VM.

For the same reason, there is a second issue: with a cap (VM priority) not multiple of 100 (meaning we are not assigning a whole number of CPUs to the VM) system interrupt ticks are irregularly spaced, different from 1/100 s: in that situation we wouldn't be able to measure precisely, e.g., the CPU percentage used by a process, because user time and system time are returned in *jiffies* instead of *seconds*.

B.3 Buffer cache and swap cache

The *buffer cache*[4] is used to speed up disk I/O; it can be *write-through* or *write-back*. The *swap cache*[5] is part of the swap space used by data that used to be on the swap but it's now on the RAM: when data needs to be swapped, it does not need to be written again.

[4] About the buffer cache and how to invalidate it see:
`http://www.faqs.org/docs/linux_admin/buffer-cache.html`
`http://aplawrence.com/Linux/buffer_cache.html`
`http://www.linux-tutorial.info/modules.php?name=MContent&pageid=310`

[5] `http://kerneltrap.org/node/4097`

Appendix C

Resources monitoring and event timing utilities and file formats

C.1 Event timing on domU

As soon as an event is generated event timing for each instance is appended at the end of a file named `evtime_<num>.out`. An example output follows:

```
345703.40 70.30 26.85 1:48.68
345809.07 71.52 26.36 1:45.47
345913.79 71.18 26.42 1:44.60
346017.91 71.54 26.47 1:44.06
346116.82 71.18 25.71 1:38.88
```

where the columns meaning is:

1. uptime when sim. exited [s]
2. cumulative user time [s]
3. cumulative system time [s]
4. elapsed real time [min:s]

Cumulative user time, system time and real time can be obtained using the time utility with a format specifier (`-a` means “append”):

```
/usr/bin/time -f "%U %S %E" -a -o "<output_file>"
```

Note that if we launch `time` without any path we could not control the output format, because some shells (e.g. bash, tcsh) have a time built-in command that has precedence over the external program. The `time` utility can be called by simply specifying the full path (and this is what we do).

C.2 Resources monitoring on domU

The monitor script (§ E.2.3) writes its output on a file, called `monitor.raw`, subsequently cleaned up by SimuPerfMon (§§ C.3, E.2.2), that produces a `monitor.out`. Each line of these files is rather long: there are several fields separated by spaces. The fields are explained in Table C.1(a) and (b).

No.	Descr.	u.m.
1	uptime	[s]
2	total RAM	[kiB]
3	free RAM	[kiB]
4	buffer cache	[kiB]
5	swap cache	[kiB]
6	total swap	[kiB]
7	free swap	[kiB]

(a) System-related fields.

No.	Descr.	u.m.
$\mathbf{8+3\times n}$	start time	[jiffies]
$\mathbf{9+3\times n}$	user time	[jiffies]
$\mathbf{10+3\times n}$	system time	[jiffies]

(b) Repeated for each n-th instance.

Table C.1: Fields in `monitor.{out,raw}` temporary files when monitoring performance on domU.

C.3 Details on SimuPerfMon

SimuPerfMon is a shell script that allows to control running simulations, monitor the execution time and collect benchmark results into a single archive. It uses the `dialog` utility, a program that uses the well-known ncurses library to bring text-mode menus and dialog windows to shell scripts.

Two screenshots of the interface are presented in Figure C.1(a) and C.1(b). A description of available menu items follows.

- The **Run** menu item, available only if no running simulation has been detected, starts in background a screen for each simulation (each screen is

named `AliRoot_<n>`, where `<n>` is the *zero-based* instance number) and a screen with the resources monitor script. The script that takes care of running each instance is `AliRoot/RunSingle.sh`. This script runs each simulation in a directory named after the instance number (`BenchRunOut/Instance_<n>`), so no instance interferes with the others.

- The **Stop and collect data** item kills the screens running the AliRoot instances, then cleans up the data in `Data/monitor.raw` and outputs `Data/monitor.out`, as described above. The whole directory is compressed in an archive that contains in its name the host name and the current date and time, such as `Data-alivaf-001-20080522-140357.tar.bz2`.

- There is also an item called **Reset swap space**, which simply does first a `swapoff` then a `swapon`. Because of the swap cache, some unused memory pages may persist on swap even if there is enough space on the RAM. Resetting swap space forces these pages to be removed from swap and eventually moved on the RAM. Note that `swapoff` correctly fails if the total memory used by the system is greater than the system RAM, that is when there isn't enough room for every memory page solely on RAM.

- When the jobs are running, SimuPerfMon prints out the **running time** *(in days, hours, minutes and seconds)* and the number of **running instances**.

(a) AliRoot is not running.

(b) AliRoot is running in background: execution time is printed.

Figure C.1: Two screenshots of SimuPerfMon, the control interface for resource monitoring benchmark.

Appendix D

VAF command-line utilities details

Some utilities have been written to manage the VAF from the head node. Each utility is a shell script and its name begins with Vaf, so the administrator can simply type `Vaf` then `[TAB]` two times to list all available VAF management utilities. A description of these utilities follows.

- `VafBroadcast` (§ E.4.2) broadcasts a command via ssh to the specified machines (PROOF workers, dom0s, PROOF master).

- `VafCopyPublicKey` (§ E.4.3) copies the VAF root ssh public key to the target machine's authorized keys list, to enable passwordless (thus, non-interactive) ssh access. This procedure is not done by the kickstart (i.e. at installation level) for security reasons, so that the administrator exactly knows which machines can be accessed by key.

- `VafMoveResources` (§ E.4.4) tells each dom0 to change the resources assigned to a particular VM (matched with wildcards expansion), in particular memory, cap and weight. If changing resources of more than one machine simultaneously and, for instance, you want to give more memory to a machine and reduce memory from another one, the script takes care of taking out memory first in order to have enough free memory to give to the other machine. This script is also responsible of automatically generating the static PROOF resource locator (`proof.conf`, see § E.3.2). Memory setting can be also specified either in MiB or as a percentage of the total memory.

- `VafProfiles` (§ E.4.5) calls `VafMoveResources` with several predefined usage profiles (such as a profile with all resources to Grid, a profile with all resources to PROOF and a profile with half resources each).
- `VafProof` is the name of two different scripts to start, stop and monitor status and log of xrootd and cmsd daemon. The first one is called on each machine and actually does the job (§ E.4.6.2); the second one resides only on the head node and calls the first script on each machine sequentially, outputting the results (§ E.4.6.1). Output is color-coded to ease error finding.

Appendix E

Code listings

E.1 Geant4 simulation control scripts

E.1.1 `run_mult.sh`

This Bash script opens a screen in background which launches a wrapper script for the Geant4 simulation.

```
#!/bin/bash

for (( i=0; i<8; i++ )); do
  screen -dmS water270_$i ./run_single.sh $i
done
```

E.1.2 `run_single.sh`

This Bash script prepares the shell environment needed by Geant4 and ROOT and passes a seed to the simulation.

```
#!/bin/bash

source env.sh

seeds=(4563421 47654168 4646876 723845 78245872348 478456382 23256583483 \
  9823897522)

seq=$1

# Automatically reads number of events from ion.mac
```

```
evt=‘grep /run/beamOn ion.mac | cut -b13-‘

input="$evt\n100\n${seeds[$seq]}\n_$seq\nexit"

date  +%d/%m/%Y\ %H.%M.%S > "time_$seq"
echo -e $input | water270
date  +%d/%m/%Y\ %H.%M.%S >> "time_$seq"
```

E.2 Moving and monitoring resources

E.2.1 MoveResources

This is the script that runs on the dom0 and changes resources configuration every two hours (7200 s).

```
#!/bin/bash

# On a single machine, every two hours

MEM=( 3584 256 3584 256 1792 512 1792 512 )
PSR=( 400 100 400 100 300 200 300 200 )

VM="alivaf-001"

CYCLE_SECS=7200

function VDate() {
  date ’+%d/%m/%Y %H.%M.%S’
}

while [ 1==1 ]
do

  for ((I=0; $I<${#MEM[*]}; I++))
  do
    echo -n "$(VDate) [$(($I+1))/${#MEM[*]}] cap=${PSR[$I]} mem=${MEM[$I]} --> "
    xm sched-credit -d $VM -c ${PSR[$I]}
    xm mem-set vaf01 ${MEM[$I]}
    echo "done"
    sleep $CYCLE_SECS
  done

done
```

E.2.2 SimuPerfMon

This is the control interface that runs on domU launches resource monitoring in background.

```
#!/bin/bash

# SimuPerfMon -- by Dario Berzano <dario.berzano@gmail.com>
    INSTANCES=1
  fi
}

# Set number of events
function SetEvents()
{
  DIALOG_OUT=$(mktemp "$DIALOG_TEMPLATE")

  dialog --no-cancel --inputbox \
    "Number of events:" 0 0 $NUM_EVENTS \
    2> "$DIALOG_OUT"

  ANS=$(cat "$DIALOG_OUT")
  rm "$DIALOG_OUT"

  NUM_EVENTS=$(($ANS+0))

  if [ $NUM_EVENTS == 0 ]; then
    NUM_EVENTS=1
  fi
}

# Run simulations
function Run()
{
  clear

  # Launches $INSTANCES instances of the simulation loop
  echo "Launching $INSTANCES instances of aliroot with $NUM_EVENTS event(s)..."
  for (( COUNT=0; COUNT<$INSTANCES; COUNT++ )); do
    screen -dmS AliRoot_$COUNT "$SCRIPTS_DIR/RunSingle.sh" $COUNT \
      $NUM_EVENTS "$SCRIPTS_DIR" "$BENCH_DIR" "$DATA_DIR"
  done

  # Launches the resources monitor script (see it for details)
  echo "Launching performance monitor..."
  screen -dmS monitor "$SCRIPTS_DIR/Monitor.sh" "$DATA_DIR" "$BENCH_DIR" \
```

```
    "$INSTANCES"

  # Ok
  Pause
}

# Resets swap space
function ResetSwap()
{
  # Swap partition
  SWAP=‘grep swap /etc/fstab | cut -d’ ’ -f1‘

  clear

  # Reset swap space - asks for root password
  if [ "$SWAP" != "" ]; then
    echo "Resetting swap space..."
    su - -c "swapoff $SWAP ; swapon $SWAP"
  else
    echo "No swap partition found."
  fi

  Pause
}

# Clean up data directory
function ClearData()
{
  echo ""
  echo -n "Clearing data directory..."
  OWD=$(pwd)
  cd "$DATA_DIR"
  rm * 2> /dev/null
  cd "$OWD"
  echo "done"
}

# Stops simulations, collects data, clears data directory
function StopCollect()
{
  clear

  KillAll
  ArchiveData
```

```
  RET=$?

  if [ $RET == 0 ]; then
    ClearData
  fi

  Pause

  if [ $RET == 0 ]; then
    dialog --msgbox "A file named $ARCHIVE has been created." 0 0
  else
    dialog --msgbox "An error occured while creating $ARCHIVE." 0 0
  fi
}

# Kill all simulations, monitor and related screens
function KillAll()
{
  # Don't kill aliroot, because RunSingle.sh eventually respawns it!
  # Note that in emergency cases -KILL instead of -HUP could be used.
  killall -HUP RunSingle.sh > /dev/null 2>&1
  MONITOR_PID=`screen -ls | grep monitor | cut -d. -f1`

  if [ "$MONITOR_PID" != "" ]; then
    kill -9 $MONITOR_PID
  fi

  echo "Killing simulation(s)..."
  echo ""
  sleep 3
  screen -wipe > /dev/null 2>&1
  screen -ls
}

# Archives data to a file in cur. directory (for easy transfer)
function ArchiveData()
{
  ARCHIVE="$SIMU_BASE_DIR/Data-$HOSTNAME-$(date +%Y%m%d-%H%M%S).tar.bz2"
  OWD=$(pwd)

  echo -n "Cleaning up the raw file..."

  # Clean-up the raw file
  NFIELDS=$((7+3*$RUNNING_INSTANCES))
  DISCARDED=0
```

```
  while read LINE
  do

    FCNT=0
    for F in $LINE
    do
      FCNT=$((FCNT+1))
    done

    if [ $FCNT == $NFIELDS ]; then
      echo $LINE >> "$DATA_DIR/monitor.out"
    else
      DISCARDED=$((DISCARDED+1))
    fi

  done < "$DATA_DIR/monitor.raw"

  echo "done"

  echo ""
  echo "Archiving data..."
  echo ""

  cd "$DATA_DIR"
  tar cjvf "$ARCHIVE" *.out
  RET=$?
  cd "$OWD"

  return $RET
}

# Pause
function Pause()
{
  echo ""
  echo -n "Press [Return] to continue..."
  read
}

# Converts a time from seconds into a nice format (return stored in $TIME_STR)
function NiceTime()
{
  TIME_S=$1
  TIME_STR=""
```

```
  if [ $TIME_S -gt 86399 ]; then
    DYS=$((TIME_S / 86400))
    TIME_S=$(($TIME_S - $DYS * 86400))
    TIME_STR="${DYS}d "
  fi

  if [ $TIME_S -gt 3599 ]; then
    HRS=$(($TIME_S / 3600))
    TIME_S=$(($TIME_S - $HRS * 3600))
    TIME_STR="${TIME_STR}${HRS}h "
  fi

  if [ $TIME_S -gt 59 ]; then
    MIN=$(($TIME_S / 60))
    TIME_S=$(($TIME_S - $MIN * 60))
    TIME_STR="${TIME_STR}${MIN}m "
  fi

  if [ $TIME_S -gt 0 ]; then
    TIME_STR="${TIME_STR}${TIME_S}s "
  fi

  LEN=$((${#TIME_STR} - 1))

  TIME_STR="${TIME_STR:0:$LEN}"
}


# Gets the running time
function GetRunningTime()
{
  UPTIME_START=$(head -n1 "$DATA_DIR/monitor.raw" | cut -d' ' -f1)
  UPTIME_NOW=$(cut -d' ' -f1 /proc/uptime)
  RUNNING_TIME=$(echo "scale=0;a=$UPTIME_NOW/1;b=$UPTIME_START/1;a-b" | bc)
  NiceTime $RUNNING_TIME
  RUNNING_TIME="$TIME_STR"
}

# The main loop
function MainLoop()
{
  while [ "$ANS" != "Quit" ]
  do
```

```
DIALOG_OUT=$(mktemp "$DIALOG_TEMPLATE")

RUNNING_INSTANCES=$(screen -ls | grep -c "AliRoot_")

if [ $RUNNING_INSTANCES == 0 ]; then
  RUN_OR_STOP="Run"
  SET_RUN_INST="Set instances"
  MENU_INST=$INSTANCES
  RUN_NO_RUN="No simulation is running"
  RUNNING_TIME=""
else
  RUN_OR_STOP="Stop and collect data"
  SET_RUN_INST="Running instances"
  MENU_INST=$RUNNING_INSTANCES
  RUN_NO_RUN="Running time"
  GetRunningTime
fi

dialog --no-cancel --menu \
  "Welcome to SimuPerfMon." 0 0 0 \
  "$RUN_NO_RUN"             "$RUNNING_TIME" \
  ""                        "" \
  "$RUN_OR_STOP"            "" \
  "$SET_RUN_INST"           "$MENU_INST" \
  "Set events"              "$NUM_EVENTS" \
  ""                        "" \
  "Reset swap space"        "" \
  "Show screens"            "" \
  "Top"                     "" \
  ""                        "" \
  "Quit"                    "" 2> "$DIALOG_OUT"

ANS=$(cat "$DIALOG_OUT")
rm "$DIALOG_OUT"

case $ANS in

  "Run")
    Run
  ;;

  "Set instances")
    SetInstances
  ;;
```

```
      "Set events")
        SetEvents
      ;;

      "Stop and collect data")
        StopCollect
      ;;

      "Reset swap space")
        ResetSwap
      ;;

      "Show screens")
        clear
        screen -ls
        Pause
      ;;

      "Top")
        top
      ;;

    esac

  done

  clear
}

# Do
Init
MainLoop
```

E.2.3 Monitor.sh

This is the monitoring script launched in background by SimuPerfMon.

```
#!/bin/bash

if [ $# -lt 3 ]; then
  echo "Usage: $0 <data_dir> <bench_dir> <instances>"
  exit 1
fi

OUT="$1/monitor.raw"
```

```
INST_PREFIX="$2/Instance_"
EVERY_SECS=5
INSTANCES="$3"

if [ -e "$OUT" ]; then
    mv "$OUT" "$OUT.bak"
fi

while [ 1 ]
do
  UPTIME="$(cut -d' ' -f1 /proc/uptime)"

  # Global system resources
  echo -n "$UPTIME " >> "$OUT"
  echo -n $(egrep 'MemTotal|MemFree|Cached|SwapTotal|SwapFree' /proc/meminfo |
    cut -b15-23) >> "$OUT"

  # Process resources
  for ((I=0; $I<$INSTANCES; I++))
  do
    if [ -e "${INST_PREFIX}${I}/proc_stat" ]; then
      echo -n \
        " $(awk '{ print $22,$14,$15 }' "${INST_PREFIX}${I}/proc_stat")" \
        >> "$OUT"
    else
      echo -n " -1 -1 -1" >> "$OUT"
    fi
  done

  echo "" >> "$OUT"

  sleep $EVERY_SECS

done
```

E.2.4 RunSingle.sh

This script, called by SimuPerfMon*, launches the AliRoot pp simulation used in performance monitoring tests. The script takes care of multiple instances by allowing them to run in parallel without writing the same output files but using different folders.*

```
#!/bin/bash

# RunSingle.sh
```

```
# * Original file by Francesco Prino
# * Heavily adapted for benchmarking by Dario Berzano

if [ $# -lt 5 ]; then
  echo "Usage: $0 <suffix> <num_of_events> \\"
  echo "         <scripts_dir> <bench_dir> <data_dir>"
  exit 1
fi

# Command-line parameters
SUFFIX="_$1"
NEV="$2"
SCRIPTS_DIR="$3"
BENCH_DIR="$4"
DATA_DIR="$5"

# Load aliroot environment variables
source "$SCRIPTS_DIR/env.sh"

# Dir. output risp. a dir. corrente
RUN="$BENCH_DIR/Instance$SUFFIX"

# Simulation parameters
export RANDSEED=123456
export NPARTICLES=10

mkdir -p "$RUN"
cd "$RUN"

while [ 1 ]
do

rm -rf *

# Creates output file with timings
touch "$DATA_DIR/evtiming$SUFFIX.out"

# Temporary file with timings
TEMP_TIME=$(mktemp "$DATA_DIR/time.XXXXX")

# AliRoot called with a system stopwatch
/usr/bin/time -f "%U %S %E" -a -o "$TEMP_TIME" \
aliroot -b <<EOI > "gen.log"
TString c = TString::Format("ln -s /proc/%d/stat proc_stat", gSystem->GetPid());
system(c.Data());
```

```
AliSimulation sim("$SCRIPTS_DIR/Config.C");
sim.SetQA(0);
sim.Run($NEV);
.q
EOI

# Simulation finished: collect data in an ordered mode
UPTIME="$(cut -d' ' -f1 /proc/uptime)"

echo $UPTIME $(cat "$TEMP_TIME") >> "$DATA_DIR/evtiming$SUFFIX.out"
rm "$TEMP_TIME"

done
```

E.3 Scalla/PROOF configuration files

E.3.1 `vaf.cf`

This configuration file is read by xrootd, cmsd and the PROOF plugin, both on the master node and the slave nodes. Note the use of if directives. *Full documentation of configuration directives for the latest version can be found on* `http://xrootd.slac.stanford.edu/#cvsheaddoc`.

```
# vaf.cf -- configuration file for xrootd/cmsd/PROOF
# by Dario Berzano <dario.berzano@gmail.com>

# Rootsys
xpd.rootsys /opt/vafsw/root/trunk
#xpd.rootsys /opt/root/default

# Fslib
xrootd.fslib libXrdOfs.so

# Specify the cms manager of the network, and its port.
all.manager proof.to.infn.it:1213

# Manager or server? The manager cannot serve files, but it can only act as a
# redirector!
xrd.port any

if proof.to.infn.it
#xrootd.redirect all
xrd.port 1094
all.role manager
```

```
xpd.role any
else
all.role server
xpd.role worker
fi

# What to export? Default is r/w!
all.export /pool/box
all.export /pool/space

# Which machines are allowed to connect?
cms.allow host proof*
cms.allow host vaf-proof-*

# Adminpath
all.adminpath /var/run/xrootd-adm

### The PROOF part ###

if exec xrootd
xrd.protocol xproofd:1093 libXrdProofd.so
fi

# The working directory for proof
xpd.workdir /pool/box

xpd.poolurl root://proof
xpd.namespace /pool/space

oss.cache public /pool/cache*
oss.path /pool/space r/w

# The resource finder for PROOF, i.e.: where are the workers?
xpd.resource static /opt/vafsw/proof.conf

# GSI Authentication
xpd.seclib libXrdSec.so
sec.protparm gsi -gmapfun:/opt/vafsw/root/trunk/lib/libXrdSecgsiGMAPLDAP.so
sec.protparm gsi -gmapfunparms:/opt/vafsw/XrdSecgsiGMAPFunLDAP.cf
sec.protparm gsi -gridmap:/opt/vafsw/grid-mapfile
xpd.sec.protocol gsi -crl:1 -dlgpxy:1 -cert:/opt/vafsw/globus/pmaster-20081227-c
  ert.pem -key:/opt/vafsw/globus/pmaster-20081227-key.pem -certdir:/opt/vafsw/al
  ien/globus/share/certificates -d:0

### The master needs this to authenticate the workers
```

```
### DO NOT PUT ANY DOUBLE QUOTES!
### I.E.: XrdBlaBla="ciao" != XrdBlaBla=ciao
# Environment variables to put in proofsrv.exe environments.
xpd.putenv XrdSecGSICADIR=/opt/vafsw/alien/globus/share/certificates
xpd.putenv XrdSecGSISRVNAMES=pmaster.to.infn.it
```

E.3.2 `proof.conf`

PROOF static resources configuration file. Since the current version of PROOF is only able to use this static file as resource finder and our facility needs resources to be configured dynamically, this file is dynamically generated by the resources control script of the VAF. When modified, xrootd does not need to be restarted in order to apply changes, since this file is read every time a new PROOF session is started. There's no possibility to specify the number of workers per each machine, so the `worker` *line has to be repeated several times (one per core in this example).*

```
# Generated by vafmon:/root/VafManagement/VafMoveResources
# By Dario Berzano <dario.berzano@gmail.com>

master proof

# PROOF node on vaf-dom0-001
worker vaf-proof-001
worker vaf-proof-001
worker vaf-proof-001
worker vaf-proof-001
worker vaf-proof-001
worker vaf-proof-001
worker vaf-proof-001
worker vaf-proof-001

# PROOF node on vaf-dom0-002
worker vaf-proof-002
worker vaf-proof-002
worker vaf-proof-002
worker vaf-proof-002
worker vaf-proof-002
worker vaf-proof-002
worker vaf-proof-002
worker vaf-proof-002

# PROOF node on vaf-dom0-003
worker vaf-proof-003
worker vaf-proof-003
```

```
worker vaf-proof-003
worker vaf-proof-003
worker vaf-proof-003
worker vaf-proof-003
worker vaf-proof-003
worker vaf-proof-003

# PROOF node on vaf-dom0-004
worker vaf-proof-004
worker vaf-proof-004
worker vaf-proof-004
worker vaf-proof-004
worker vaf-proof-004
worker vaf-proof-004
worker vaf-proof-004
worker vaf-proof-004
```

E.3.3 `grid-mapfile`

An example line of the `grid-mapfile` follows:

```
"/C=IT/O=INFN/OU=Personal Certificate/L=Torino/CN=Dario Carlo \
  Domenico BERZANO" dberzano,guest,testuser1,testuser2
```

where the user with that certificate subject is statically mapped to four comma-separated Unix users on each machine. This file is actually empty on the present VAF configuration (but its presence is mandatory or else xrootd fails to load), and it has a debug purpose: mapping the administrator's subject to a certain user may help solving user-specific problems without actually using the user's key, which needs a passphrase to be unlocked.

E.3.4 `XrdSecgsiGMAPFunLDAP.cf`

Specifies the LDAP server where to search for a matching user name by certificate subject.

```
srv:  ldap://aliendb06a.cern.ch:8389
base: ou=People,o=alice,dc=cern,dc=ch
attr: uid
```

E.4 VAF command-line utilities

For a description of these utilities, see § 3.1.5.

E.4.1 ~/.vafcfg

Every VAF utility needs the following variables to be set. These variables are stored in the `~/.vafcfg` *file,* `source`*'d by the* `~/.bashrc` *file with the following line of code:*

```
source ~/.vafcfg
```

```
# VAF Workers
export VAF_DOM0="vaf-dom0-001 vaf-dom0-002 vaf-dom0-003 vaf-dom0-004"

# VAF Workers
export VAF_WRK="vaf-proof-001 vaf-proof-002 vaf-proof-003 vaf-proof-004"

# VAF Master
export VAF_MST="proof"

# VAF Scripts Path
export VAF_MANAGEMENT_PATH="/root/VafManagement"

# The Path
export PATH="$PATH:$VAF_MANAGEMENT_PATH"
```

E.4.2 VafBroadcast

```
#!/bin/bash

#
# Global variables
#

NODES=""
CMD=""

#
# Entry point
#

while [ $# -ge 1 ]
do

  if [ "${1:0:2}" == "--" ]; then
    PARAM=${1:2}
  elif [ "${1:0:1}" == "-" ]; then
    PARAM=${1:1}
```

```
  else
    PARAM=""
    CMD="$1"
  fi

  case $PARAM in

    master|m)
      NODES="$NODES $VAF_MST"
    ;;

    dom0|d)
      NODES="$NODES $VAF_DOM0"
    ;;

    workers|w)
      NODES="$NODES $VAF_WRK"
    ;;

  esac

  shift

done

#
# Execute commands
#

if [ "$CMD" == "" ] || [ "$NODES" == "" ]; then

  echo ""
  echo "Usage: VafBroadcast <hosts> \"<command>\""
  echo ""
  echo "Where <hosts> can be one or more of:"
  echo ""
  echo "  --workers  [-w]  all PROOF workers"
  echo "  --dom0     [-d]  all dom0s"
  echo "  --master   [-m]  the PROOF master"
  echo ""

else

  for N in $NODES
  do
```

```
    echo "==> $N <=="
    ssh $N "$CMD"
  done

fi
```

E.4.3 VafCopyPublicKey

```
#!/bin/bash

if [ "$1" == "" ]; then
  echo "Usage: $0 <hostname>"
else
  echo "Creating .ssh directory..."
  ssh root@"$1" '( mkdir -p ~/.ssh )'

  if [ "$?" == 0 ]; then
    echo "Copying authorized_keys..."
    scp /root/.ssh/id_rsa.pub root@"$1":~/.ssh/authorized_keys

    if [ "$?" == 0 ]; then
      echo "Done!"
    else
      echo "Copying authorized_keys failed!"
    fi
  else
    echo "Creation of .ssh failed!"
  fi
fi
```

E.4.4 VafMoveResources

```
#!/bin/bash

#
# Start and stop xrootd and cmsd.
#
# by Dario Berzano <dario.berzano@gmail.com>
#
# This script is distribution-agnostic (i.e. should run flawlessly on any Linux
# distribution). Every configuration variable needed to run xrootd and cmsd can
# be set here: no other files are needed.
#
```

```
#
# Global variables
#

# Where is everything installed
PREFIX="/opt/vafsw"

# Binaries of xrootd and cmsd
XROOTD_BIN="$PREFIX/root/trunk/bin/xrootd"
CMSD_BIN="$PREFIX/root/trunk/bin/cmsd"

# xrootd libraries
XROOTD_LIB="$PREFIX/root/trunk/lib"
CMSD_LIB="$PREFIX/root/trunk/lib"

# Log directory (hostname based)
LOG_DIR="/var/xrootd/VafLogs/‘hostname -s‘"

# xrootd
XROOTD_USER="xrootd"
XROOTD_LOG="$LOG_DIR/xrootd.log"
XROOTD_CF="/opt/vafsw/vaf.cf"
XROOTD_EXTRAOPTS=""

# cmsd
CMSD_USER="xrootd"
CMSD_LOG="$LOG_DIR/cmsd.log"
CMSD_CF="$XROOTD_CF"
CMSD_EXTRAOPTS=""

#
# Functions
#

# Is this terminal capable of colors?
function canColor() {

  # TODO for now, skip the rest and always color...
  return 1

  if [ "$TERM" == "xterm-color" ]; then
    return 1
  fi

  return 0
```

```
}

# Echo function with a prefix
function vEcho() {

  canColor

  if [ $? == 1 ]; then
    echo -e "\033[33m\033[1m[VafProof]\033[0m $@"
  else
    echo "[VafProof] $@"
  fi

}

# Echo separators
function sEcho() {

  canColor

  if [ $? == 1 ]; then
    vEcho "\033[31m===\033[0m \033[33m$@\033[0m \033[31m===\033[0m"
  else
    vEcho "=== $@ ==="
  fi

}

# PID guessing
function GuessPids() {
  XROOTD_PID=`ps ax | grep -v grep | grep $XROOTD_BIN | awk '{print $1}'`
  CMSD_PID=`ps ax | grep -v grep | grep $CMSD_BIN | awk '{print $1}'`
}

# Start daemons
function Start() {

  mkdir -p "$LOG_DIR"

  touch $XROOTD_LOG
  chown $XROOTD_USER "$XROOTD_LOG"

  touch $CMSD_LOG
  chown $CMSD_USER "$CMSD_LOG"
```

```
  # Admin directories with sockets - directly read from cf
  XROOTD_ADM=`grep adminpath "$XROOTD_CF" | awk '{print $2}'`
  CMSD_ADM=`grep adminpath "$CMSD_CF" | awk '{print $2}'`

  mkdir -p "$XROOTD_ADM"
  chown -R $XROOTD_USER "$XROOTD_ADM"

  mkdir -p "$CMSD_ADM"
  chown -R $CMSD_USER "$CMSD_ADM"

  # TODO
  # Maybe this method is better, don't know...
  #
  #su $XROOTD_USER -c \
  #  "( export LD_LIBRARY_PATH=\"\$LD_LIBRARY_PATH:$XROOTD_LIB\" ;
  #     export XrdSecGSISRVNAMES=\"pmaster.to.infn.it\" ;
  #     $XROOTD_BIN -b -l $XROOTD_LOG -c $XROOTD_CF $XROOTD_EXTRAOPTS )"

  vEcho "Starting xrootd"
  ( export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$XROOTD_LIB" ; $XROOTD_BIN -b \
    -l $XROOTD_LOG -R $XROOTD_USER -c $XROOTD_CF $XROOTD_EXTRAOPTS )
  vEcho "xrootd started"

  vEcho "Starting cmsd"
  ( export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$XROOTD_LIB" ; $CMSD_BIN -b \
    -l $CMSD_LOG -R $CMSD_USER -c $CMSD_CF $CMSD_EXTRAOPTS )
  vEcho "cmsd started"

}

# Stop daemons
function Stop() {

  GuessPids

  if [ "$XROOTD_PID" != "" ]; then
    vEcho "Stopping xrootd"
    kill -9 $XROOTD_PID
    vEcho "xrootd stopped"
  else
    vEcho "No running xrootd found, nothing stopped"
  fi

  if [ "$CMSD_PID" != "" ]; then
```

```
    vEcho "Stopping cmsd"
    kill -9 $CMSD_PID
    vEcho "cmsd stopped"
  else
    vEcho "No running cmsd found, nothing stopped"
  fi
}

# Status
function Status() {

  GuessPids

  if [ "$XROOTD_PID" != "" ]; then
    vEcho "xrootd is $RUNNING"
  else
    vEcho "xrootd is $NOT_RUNNING"
  fi

  if [ "$1" == "--logs" ]; then
    sEcho "last lines of xrootd log"
    tail -n10 "$XROOTD_LOG"
    sEcho "EOF"
  fi

  if [ "$CMSD_PID" != "" ]; then
    vEcho "cmsd is $RUNNING"
  else
    vEcho "cmsd is $NOT_RUNNING"
  fi

  if [ "$1" == "--logs" ]; then
    sEcho "last lines of cmsd log"
    tail -n10 "$CMSD_LOG"
    sEcho "EOF"
  fi
}

#
# Entry point
#

canColor

if [ $? == 1 ]; then
```

```
  RUNNING="\033[32mrunning\033[0m"
  NOT_RUNNING="\033[31mNOT running!\033[0m"
else
  RUNNING="running"
  NOT_RUNNING="NOT running!"
fi

#
# Check user
#

if [ $USER != "root" ]; then
  vEcho "Only root is allowed to run this script, aborting."
  exit 1
fi

case "$1" in

  start)
    Start
  ;;

  start-check)
    Start
    sleep 1
    Status
  ;;

  stop)
    Stop
  ;;

  restart)
    Stop
    Start
  ;;

  restart-check)
    Stop
    Start
    sleep 1
    Status
  ;;

  status)
```

```
    Status
  ;;

  logs)
    Status --logs
  ;;

  *)
    vEcho "Usage: $0 {start[-check]|stop|restart[-check]|status|logs}"
  ;;

esac
```

E.4.5 VafProfiles

```
#!/bin/bash

VAF_MOVE_RES="$VAF_MANAGEMENT_PATH/VafMoveResources"

# Majority of resources to PROOF
NAME[1]="proof"
PROF[1]="-d Domain-0 -m  7% -d proof -m 86% -c 800 -w 256 \
  -d wnU -m  7% -c 100 -w  32 -p 8"

# Half memory each, PROOF has priority
NAME[2]="half-proof"
PROF[2]="-d Domain-0 -m  7% -d proof -m 46% -c 800 -w 256 \
  -d wnU -m 47% -c 800 -w  32 -p 8"

# Half memory each, PROOF has priority, WN is stretched to one CPU
NAME[2]="halfmem-proof-prio"
PROF[2]="-d Domain-0 -m  7% -d proof -m 46% -c 800 -w 256 \
  -d wnU -m 47% -c 100 -w  32 -p 8"

# Half memory each, no priority
NAME[3]="half-noprio"
PROF[3]="-d Domain-0 -m  7% -d proof -m 46% -c 800 -w 256 \
  -d wnU -m 47% -c 800 -w 256 -p 8"

# Majority of resources to WN
NAME[4]="wn"
PROF[4]="-d Domain-0 -m  7% -d proof -m  7% -c 100 -w  32 \
  -d wnU -m 86% -c 800 -w 256 -p 0"

LEN=${#PROF[@]}
```

```
function List() {
  for ((I=1; $I<=$LEN; I++))
  do
    echo ${NAME[$I]}: ${PROF[$I]}
  done
}

function ShortList() {
  echo -n "Choose one of: "
  for ((I=1; $I<=$LEN; I++))
  do
    echo -n "${NAME[$I]} "
  done
  echo ""
}

function Set() {

  if [ "$1" == "" ]; then
    echo "set: profile name required."
    ShortList
    return 1
  fi

  # Search profile index
  INDEX=-1

  for ((I=1; $I<=$LEN; I++))
  do
    if [ "${NAME[$I]}" == "$1" ]; then
      INDEX=$I
      break
    fi
  done

  if [ $INDEX == -1 ]; then
    echo "set: profile \"$1\" not found."
    ShortList
    return 2
  fi

  # Run command
  shift
  $VAF_MOVE_RES $@ ${PROF[$INDEX]}
```

```
}

case "$1" in

  --list)
    List
  ;;

  --set)
    shift
    Set "$@"
  ;;

  *)
    echo "Usage: $0 {--set <profile_name> [options...]|--list}"
  ;;

esac
```

E.4.6 VafProof

E.4.6.1 Remote control from the head node

```
#!/bin/bash

VAFPROOF="/opt/vafsw/VafProof"

source ~/.vafcfg

for V in $VAF_MST $VAF_WRK
do
  echo "==> Node $V <=="
  ssh $V "$VAFPROOF $1"
done
```

E.4.6.2 Client script on each slave

```
#!/bin/bash

#
# Start and stop xrootd and cmsd.
#
# by Dario Berzano <dario.berzano@gmail.com>
#
```

```
# This script is distribution-agnostic (i.e. should run flawlessly on any Linux
# distribution). Every configuration variable needed to run xrootd and cmsd can
# be set here: no other files are needed.
#

#
# Global variables
#

# Where is everything installed
PREFIX="/opt/vafsw"

# Binaries of xrootd and cmsd
XROOTD_BIN="$PREFIX/root/trunk/bin/xrootd"
CMSD_BIN="$PREFIX/root/trunk/bin/cmsd"

# xrootd libraries
XROOTD_LIB="$PREFIX/root/trunk/lib"
CMSD_LIB="$PREFIX/root/trunk/lib"

# Log directory (hostname based)
LOG_DIR="/var/xrootd/VafLogs/‘hostname -s‘"

# xrootd
XROOTD_USER="xrootd"
XROOTD_LOG="$LOG_DIR/xrootd.log"
XROOTD_CF="/opt/vafsw/vaf.cf"
XROOTD_EXTRAOPTS=""

# cmsd
CMSD_USER="xrootd"
CMSD_LOG="$LOG_DIR/cmsd.log"
CMSD_CF="$XROOTD_CF"
CMSD_EXTRAOPTS=""

#
# Functions
#

# Is this terminal capable of colors?
function canColor() {

  # TODO for now, skip the rest and always color...
  return 1
```

```
  if [ "$TERM" == "xterm-color" ]; then
    return 1
  fi

  return 0

}

# Echo function with a prefix
function vEcho() {

  canColor

  if [ $? == 1 ]; then
    echo -e "\033[33m\033[1m[VafProof]\033[0m $@"
  else
    echo "[VafProof] $@"
  fi

}

# Echo separators
function sEcho() {

  canColor

  if [ $? == 1 ]; then
    vEcho "\033[31m===\033[0m \033[33m$@\033[0m \033[31m===\033[0m"
  else
    vEcho "=== $@ ==="
  fi

}

# PID guessing
function GuessPids() {
  XROOTD_PID=`ps ax | grep -v grep | grep $XROOTD_BIN | awk '{print $1}'`
  CMSD_PID=`ps ax | grep -v grep | grep $CMSD_BIN | awk '{print $1}'`
}

# Start daemons
function Start() {

  mkdir -p "$LOG_DIR"
```

```
  touch $XROOTD_LOG
  chown $XROOTD_USER "$XROOTD_LOG"

  touch $CMSD_LOG
  chown $CMSD_USER "$CMSD_LOG"

  # Admin directories with sockets - directly read from cf
  XROOTD_ADM=`grep adminpath "$XROOTD_CF" | awk '{print $2}'`
  CMSD_ADM=`grep adminpath "$CMSD_CF" | awk '{print $2}'`

  mkdir -p "$XROOTD_ADM"
  chown -R $XROOTD_USER "$XROOTD_ADM"

  mkdir -p "$CMSD_ADM"
  chown -R $CMSD_USER "$CMSD_ADM"

  # TODO
  # Maybe this method is better, don't know...
  #
  #su $XROOTD_USER -c \
  #  "( export LD_LIBRARY_PATH=\"\$LD_LIBRARY_PATH:$XROOTD_LIB\" ;
  #     export XrdSecGSISRVNAMES=\"pmaster.to.infn.it\" ;
  #     $XROOTD_BIN -b -l $XROOTD_LOG -c $XROOTD_CF $XROOTD_EXTRAOPTS )"

  vEcho "Starting xrootd"
  ( export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$XROOTD_LIB" ; $XROOTD_BIN -b \
    -l $XROOTD_LOG -R $XROOTD_USER -c $XROOTD_CF $XROOTD_EXTRAOPTS )
  vEcho "xrootd started"

  vEcho "Starting cmsd"
  ( export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$XROOTD_LIB" ; $CMSD_BIN -b \
    -l $CMSD_LOG -R $CMSD_USER -c $CMSD_CF $CMSD_EXTRAOPTS )
  vEcho "cmsd started"

}

# Stop daemons
function Stop() {

  GuessPids

  if [ "$XROOTD_PID" != "" ]; then
    vEcho "Stopping xrootd"
    kill -9 $XROOTD_PID
    vEcho "xrootd stopped"
```

```
  else
    vEcho "No running xrootd found, nothing stopped"
  fi

  if [ "$CMSD_PID" != "" ]; then
    vEcho "Stopping cmsd"
    kill -9 $CMSD_PID
    vEcho "cmsd stopped"
  else
    vEcho "No running cmsd found, nothing stopped"
  fi
}

# Status
function Status() {

  GuessPids

  if [ "$XROOTD_PID" != "" ]; then
    vEcho "xrootd is $RUNNING"
  else
    vEcho "xrootd is $NOT_RUNNING"
  fi

  if [ "$1" == "--logs" ]; then
    sEcho "last lines of xrootd log"
    tail -n10 "$XROOTD_LOG"
    sEcho "EOF"
  fi

  if [ "$CMSD_PID" != "" ]; then
    vEcho "cmsd is $RUNNING"
  else
    vEcho "cmsd is $NOT_RUNNING"
  fi

  if [ "$1" == "--logs" ]; then
    sEcho "last lines of cmsd log"
    tail -n10 "$CMSD_LOG"
    sEcho "EOF"
  fi
}

#
# Entry point
```

```
#

canColor

if [ $? == 1 ]; then
  RUNNING="\033[32mrunning\033[0m"
  NOT_RUNNING="\033[31mNOT running!\033[0m"
else
  RUNNING="running"
  NOT_RUNNING="NOT running!"
fi

#
# Check user
#

if [ $USER != "root" ]; then
  vEcho "Only root is allowed to run this script, aborting."
  exit 1
fi

case "$1" in

  start)
    Start
  ;;

  start-check)
    Start
    sleep 1
    Status
  ;;

  stop)
    Stop
  ;;

  restart)
    Stop
    Start
  ;;

  restart-check)
    Stop
    Start
```

```
    sleep 1
    Status
  ;;

  status)
    Status
  ;;

  logs)
    Status --logs
  ;;

  *)
    vEcho "Usage: $0 {start[-check]|stop|restart[-check]|status|logs}"
  ;;

esac
```

E.5 Use case: an analysis task that runs on PROOF

E.5.1 MakeESDInputChain.C

This macro produces a TChain *from a list of ESDs given one by line in a text file. If the input file cannot be read, an error message is printed and a null pointer is returned.*

```
TChain *MakeESDInputChainFromFile(TString esdListFn) {

  TChain *chainESD = 0x0;
  TString line;

  ifstream is(esdListFn.Data());

  if (is) {
    chainESD = new TChain("esdTree");

    while (is.good()) {
      is >> line;
      if (line.Length() > 0) {
        chainESD->Add(line);
      }
    }

    is.close();
```

```
  }
  else {
    Printf("Can't open \"%s\" for reading: no chain has been created.",
      esdListFn.Data());
  }

  return chainESD;
}
```

E.5.2 Analysis task that makes E and p_t spectra

The original code of this class is used in the ALICE Offline Tutorial[20], so credits go to its authors: Panos Cristakoglou, Jan Fiete Grosse-Oetringhaus and Christian Klein-Boesing.

E.5.2.1 AliAnalysisTaskDistros.h

```
/*
 * Example of an analysis task creating a p_t and E spectra: class definition.
 *
 * This is a slightly modified version of the example PROOF macro in the ALICE
 * Offline Tutorial.
 */

#ifndef AliAnalysisTaskDistros_cxx
#define AliAnalysisTaskDistros_cxx

class TH1F;
class AliESDEvent;

#include "AliAnalysisTask.h"

class AliAnalysisTaskDistros : public AliAnalysisTask {

 public:
  AliAnalysisTaskDistros(const char *name = "AliAnalysisTaskDistros");
  virtual ~AliAnalysisTaskDistros() {}

  virtual void ConnectInputData(Option_t *);
  virtual void CreateOutputObjects();
  virtual void Exec(Option_t *option);
  virtual void Terminate(Option_t *);

 private:
```

```
  AliESDEvent *fESD;    // ESD object
  TH1F        *fHistPt; // Pt spectrum
  TH1F        *fHistE;  // Energy spectrum

  AliAnalysisTaskDistros(const AliAnalysisTaskDistros&); // not implemented
  AliAnalysisTaskDistros& operator=(const AliAnalysisTaskDistros&); // not implemented

  ClassDef(AliAnalysisTaskDistros, 1); // example of analysis
};

#endif
```

E.5.2.2 AliAnalysisTaskDistros.cxx

```
/*
 * Example of an analysis task creating a p_t and E spectra: implementation of
 * the class.
 *
 * This is a slightly modified version of the example PROOF macro in the ALICE
 * Offline Tutorial.
 */

#include "TChain.h"
#include "TTree.h"
#include "TH1F.h"
#include "TCanvas.h"

#include "AliAnalysisTask.h"
#include "AliAnalysisManager.h"

#include "AliESDEvent.h"
#include "AliESDInputHandler.h"

#include "AliAnalysisTaskDistros.h"


ClassImp(AliAnalysisTaskDistros);

AliAnalysisTaskDistros::AliAnalysisTaskDistros(const char *name) :
  AliAnalysisTask(name, ""), fESD(0x0), fHistPt(0x0), fHistE(0x0) {

  // Define input and output slots here

  // Input slot #0 works with a TChain
  DefineInput(0, TChain::Class());
```

```
  // Output slots #0, #1 write into a TH1 container
  DefineOutput(0, TH1F::Class());  // Pt
  DefineOutput(1, TH1F::Class());  // E
}

void AliAnalysisTaskDistros::ConnectInputData(Option_t *) {

  TTree* tree = dynamic_cast<TTree*>( GetInputData(0) );

  if (!tree) {
    Printf("Error: could not read chain from input slot 0");
  }
  else {
    // Disable all branches and enable only the needed ones
    // The next two lines are different when data produced as AliESDEvent is read
    //    tree->SetBranchStatus("*", kFALSE);
    //    tree->SetBranchStatus("fTracks.*", kTRUE);

    AliESDInputHandler *esdH = dynamic_cast<AliESDInputHandler*>(
      AliAnalysisManager::GetAnalysisManager()->GetInputEventHandler() );

    if (!esdH) {
      Printf("Error: could not get ESDInputHandler");
    }
    else {
      fESD = esdH->GetEvent();
    }
  }
}

void AliAnalysisTaskDistros::CreateOutputObjects() {

  // Transverse momentum [GeV/c] distribution
  fHistPt = new TH1F("fHistPt", "Transverse momentum distribution", 50, 0., 5.);
  fHistPt->GetXaxis()->SetTitle("p_{t} [GeV/c]");
  fHistPt->GetYaxis()->SetTitle("dN/dp_{t} [1/(GeV/c)]");
  fHistPt->SetMarkerStyle(kFullCircle);

  // Energy distribution [GeV] distribution
  fHistE = new TH1F("fHistE", "Energy distribution", 50, 0., 5.);
  fHistE->GetXaxis()->SetTitle("E [GeV]");
  fHistE->GetYaxis()->SetTitle("dN/dE [1/GeV]");
  fHistE->SetMarkerStyle(kFullCircle);
```

```
}

void AliAnalysisTaskDistros::Exec(Option_t *)  {
  // Main loop
  // Called for each event

  if (!fESD) {
    Printf("Error: fESD not available");
    return;
  }

  Printf("There are %d tracks in this event", fESD->GetNumberOfTracks());

  // Track loop to fill spectra
  for (Int_t iTracks = 0; iTracks < fESD->GetNumberOfTracks(); iTracks++) {

    AliESDtrack* track = fESD->GetTrack(iTracks);

    if (!track) {
      Printf("Error: Could not receive track %d", iTracks);
      continue;
    }

    fHistPt->Fill(track->Pt());
    fHistE->Fill(track->E());
  }

  // Post output data (first number should be the output container number);
  PostData(0, fHistPt);
  PostData(1, fHistE);
}

//________________________________________________________________________
void AliAnalysisTaskDistros::Terminate(Option_t *) {
  // Draw result to the screen
  // Called once at the end of the query

  fHistPt = dynamic_cast<TH1F*>( GetOutputData(0) );
  fHistE  = dynamic_cast<TH1F*>( GetOutputData(1) );

  if ((!fHistPt) || (!fHistE)) {
    Printf("Error: some histograms are not available!");
    return;
  }
```

```
  TCanvas *c1 = new TCanvas("cPt", "Pt spectrum");
  c1->cd(1)->SetLogy();
  fHistPt->DrawCopy("E");

  TCanvas *c2 = new TCanvas("cE", "E spectrum");
  c2->cd(1)->SetLogy();
  fHistE->DrawCopy("E");
}
```

E.5.3 run.C

This macro is able to run the analysis on a given list of files either locally or on PROOF: this parameter is not hardcoded, thus it can be changed at runtime.

```
/*
 * Test macro for an AliAnalysisTask.
 *
 * It can run in "local", "proof" or "grid" mode: you can specify it when
 * loading the macro.
 */

void run(TString analysisMode = "local", TString esdListFn = "ESDList.txt",
  TString proofServer = "proof") {

  Bool_t useParFiles = kFALSE;

  gROOT->LoadMacro("MakeESDInputChain.C");

  if (analysisMode == "proof") {
    TProof::Open(proofServer.Data());

    if ( gProof->UploadPackage("VAF") ) {
      Printf("The package cannot be uploaded: analysis aborted.");
      return;
    }

    if ( gProof->EnablePackage("VAF") ) {
      Printf("The package cannot be enabled: analysis aborted.");
      return;
    }

    if ( gProof->Load("AliAnalysisTaskDistros.cxx+") ) {
      Printf("The AnalysisTask cannot be loaded on PROOF: analysis aborted.");
    }
  }
```

```
else {

  TString libs("libVMC,libNet,libTree,libPhysics,libSTEERBase,libANALYSIS,"
    "libESD,libAOD,libANALYSISalice,libRAWDatabase,libRAWDatarec,libProof,"
    "libCDB,libSTEER,libMinuit,libTPCbase,libTPCrec");

  TObjArray *tok = libs.Tokenize(",");
  TIter i(tok);
  TObjString *l = 0x0;

  while (l = (TObjString *)i.Next()) {
    if ( gSystem->Load(l->GetString()) < 0 ) {
      Printf("Unable to load library \"%s\": aborting analysis.",
        (l->GetString()).Data());
    }
  }

  tok->SetOwner(kTRUE);
  delete tok;

  // Uncomment to test if libraries are loaded. This is useful in order to
  // avoid error messages while loading files like:
  //
  //   Warning in <TClass::TClass>: no dictionary for class [...]
  //
  //TFile::Open("root://proof//pool/space/dberzano/Bala/ESDs/180110/001/"
  //  "AliESDs.root");
  //return;

  gROOT->ProcessLine(".include $ALICE_ROOT/include");

  if ( gROOT->LoadMacro("AliAnalysisTaskDistros.cxx+") < 0 ) {
    Printf("The AnalysisTask can not be loaded: aborting analysis.");
    return;
  }
}

// Files are taken from the Grid in the form alien://[...]
if (analysisMode == "grid")
  TGrid::Connect("alien:", 0, 0, "t");

TChain *chainESD = 0x0;

chainESD = MakeESDInputChainFromFile(esdListFn);
if (chainESD == 0x0) {
```

```
  Printf("No chain was created: aborting analysis.");
  return;
}

// Create the analysis manager
AliAnalysisManager *mgr  = new AliAnalysisManager("MyManager", "MyManager");
mgr->SetDebugLevel(0);

// Input
AliESDInputHandler *inputHandler = new AliESDInputHandler();
mgr->SetInputEventHandler(inputHandler);

// Aanalysis task
AliAnalysisTaskDistros *task = new AliAnalysisTaskDistros();
//task->SetDebugLevel(0);
mgr->AddTask(task);

// Create containers for input/output and connect to slots
AliAnalysisDataContainer *cInput = mgr->CreateContainer("cInput",
  TChain::Class(), AliAnalysisManager::kInputContainer);
mgr->ConnectInput( task, 0, mgr->GetCommonInputContainer() );

AliAnalysisDataContainer *cOutput0 = mgr->CreateContainer("cOutput0",
  TH1::Class(), AliAnalysisManager::kOutputContainer, "Distros.root");
mgr->ConnectOutput( task, 0, cOutput0 );

AliAnalysisDataContainer *cOutput1 = mgr->CreateContainer("cOutput1",
  TH1::Class(), AliAnalysisManager::kOutputContainer, "Distros.root");
mgr->ConnectOutput( task, 1, cOutput1 );

// Run the analysis
Printf( "Loaded chain has %d entries.\n", (Int_t)(chainESD->GetEntries()) );

// Timings, start
TStopwatch cron;
cron.Start();

if (mgr->InitAnalysis()) {
  mgr->PrintStatus();
  mgr->StartAnalysis(analysisMode, chainESD);
}

// Timings, stop
Printf("Time for analysis execution (excluding chain creation) follows.");
cron.Stop();
```

```
  cron.Print();
}
```

List of Acronyms

AJAX	Asynchronous JavaScript And XML, an “umbrella term” that describes the technoogies used to dynamically send data to web pages without reloading them, by separating content from layout
ALICE	A Large Ion Collider Experiment
ATLAS	A Toroidal LHC ApparatuS
AOD	Analysis Object Dataset
AliEn	ALICE Environment
ARDA	A Realisation of Distributed Analysis for LHC
CAF	CERN Analysis Facility
CERN	Organisation Européenne pour la Recherche Nucléaire
CE	Computing Element
CMS	Compact Muon Solenoid
CMS	Content Management System
DCA	Distance of Closest Approach
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Server
EMCAL	Electromagnetic Calorimeter
ESD	Event Summary Data
FC	File Catalogue, referred to an AliEn facility
FMS	Forward Muon Spectrometer

FUSE File System in User Space

GSI Grid Security Infrastructure

GUID Global Unique Identifier, an unique identifier used as a file name in AliEn

GWT Google Web Toolkit, a toolkit developed by Google to write web applications in Java that takes care of multibrowser compatibility and JavaScript+HTML translation

HA High-Availability

HBT Hanbury-Brown Twiss

HEP High-Energy Physics

HPC High-Performance Computing

HTTP Hypertext Transfer Protocol

HVM Hardware Virtual Machine

ITS Inner Tracking System

TPC Time Projection Chamber

TRD Transition-Radiation Detector

KVM Kernel-based Virtual Machine

KVM Keyboard Video Mouse

LCG LHC Computing Grid

LDAP Lightweight Directory Access Protocol

LFN Logical File Name

LHC Large Hadron Collider

LHCb Large Hadron Collider Beauty

LVM Logical Volume Manager

MONARC Models of Networked Analysis at Regional Centres

MRPC Multigap Resistive Plate Chamber

MSS Mass-Storage System

MWPC Multi-Wire Proportional Chamber

NAS Network-Attached Storage

NAT Network Address Translation

NFS Network File System

NLO Next-to-Leading Order

NTP Network Time Protocol, a simple protocol used to transport time onto a network for automatic time synchronization

OLTP On-Line Transaction Processing, a class of systems that facilitate and manage transaction-oriented applications, typically for data entry and retrieval transaction processing

OO Object-Oriented

PDC Physics Data Challenge

PDF Parton Distribution Function

PFN Physical File Name

PHP PHP: Hypertext Preprocessor, one of the most popular server-side languages for generating dynamic web pages

PID Particle Identification

PMU Performance Monitoring Unit

POD PROOF On Demand

POSIX Portable Operating System Interface, formerly known as IEEE-IX

pQCD Perturbative QCD

PROOF Parallel ROOT Facility

PXE Preboot Execution Environment

QCD Quantum Chromodynamics

QGP Quark-Gluon Plasma

QoS Quality of Service

RAID Redundant Array of Inexpensive Disks

RHIC Relativistic Heavy-Ion Collider, placed at the Brookhaven National Laboratory in Upton, NY

RICH Ring Imaging Čerenkov

RPC Resistive Plate Chamber

SAN Storage Area Network

SAS Serial Attached SCSI, see `http://en.wikipedia.org/wiki/Serial_Attached_SCSI`

SDD Silicon Drift Detector

SE Storage Element

SOAP Simple Object Access Protocol

SPD Silicon Pixel Detector

SQL Structured Query Language

SSH Secure Shell

TFTP Trivial File Transfer Protocol, a lightweight file transfer protocol that runs on top of UDP

TOF Time Of Flight

PHOS Photon Spectrometer

HMPID High-Momentum Particle Identification

TQ Task Queue

VAF Virtual Analysis Facility

VBD Virtual Block Device

VCPU Virtual CPU, a CPU as seen by the VM with no exact correspondence with a physical CPU

VMM Virtual Machine Monitor

VM Virtual Machine

VO Virtual Organization

WN Worker Node

ZDC Zero-Degree Calorimeter

FMD Forward Multiplicity Detector

PMD Photon Multiplicity Detector

iLO Integrated Lights-Out

References

[1] AMD Virtualization (AMD-V^{TM}) Technology. `http://www.amd.com/us-en/0,,3715_15781_15785,00.html`.

[2] Agostinelli, S., Allison, J., Amako, K., Apostolakis, J., Araujo, H., Arce, P., Asai, M., Axen, D., Banerjee, S., Barrand, G., et al. GEANT4 – A simulation toolkit. *Nuclear Inst. and Methods in Physics Research, A 506*, 3 (2003), 250–303.

[3] Alessandro, B., Alexa, C., Arnaldi, R., Atayan, M., Beolè, S., Boldea, V., Bordalo, P., Borges, G., Castanier, G., Castor, J., et al. A new measurement of J/ψ suppression in Pb-Pb collisions at 158 GeV per nucleon. *The European physical journal. C, Particles and fields 39*, 3 (2005), 335–345.

[4] Bagnasco, S., Betev, L., Buncic, P., Carminati, F., Cirstoiu, C., Grigoras, C., Hayrapetyan, A., Harutyunyan, A., Peters, A. J., and Saiz, P. Alien: Alice environment on the grid. *Journal of Physics: Conference Series 119*, 6 (2008), 062012 (9pp).

[5] Ballintijn, M., Roland, G., Brun, R., and Rademakers, F. The PROOF distributed parallel analysis framework based on ROOT. *Arxiv preprint physics/0306110* (2003).

[6] Berners-Lee, T., and Cailliau, R. The world-wide web. *Communications of the ACM* (1994).

[7] Berners-Lee, T., Cailliau, R., Groff, J., and Pollermann, B. World Wide Web: An Information Infrastructure for High Energy Physics. In *Proceedings of the Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics* (1992).

[8] Carminati, F., Foka, P., Giubellino, P., Morsch, A., Paic, G., Revol, J.-P., Safarik, K., Schutz, Y., and Wiedemann, U. A. ALICE: Physics Performance Report, Volume I. *Journal of Physics G: Nuclear and Particle Physics 30*, 11 (2004), 1517–1763. ALICE Collaboration.

[9] COLLINS, J. C., SOPER, D. E., AND STERMAN, G. Heavy particle production in high-energy hadron collisions. *Nuclear Physics B 263*, 1 (1986), 37 – 60.

[10] DORIGO, A., ELMER, P., FURANO, F., AND HANUSHEVSKY, A. XROOTD/TXNetFile: a highly scalable architecture for data access in the ROOT environment. In *TELE-INFO'05: Proceedings of the 4th WSEAS International Conference on Telecommunications and Informatics* (Stevens Point, Wisconsin, USA, 2005), World Scientific and Engineering Academy and Society (WSEAS), pp. 1–6.

[11] ELIA, D. Strange particle production in 158 and 40 A GeV/c Pb-Pb and p-Be collisions. *Journal of Physics G: Nuclear and Particle Physics 31*, 4 (2005), S135–S140. NA57 Collaboration.

[12] ERANIAN, S. The perfmon2 interface specification. Tech. rep., Technical Report HPL-2004-200 (R. 1), HP Labs, 2005.

[13] FOSTER, I., AND KESSELMAN, C. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.

[14] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the Grid: enabling scalable virtual organizations. *International Journal of High Performance Computing Applications 15*, 3 (2001), 200.

[15] JARP, S., JURGA, R., AND NOWAK, A. Perfmon2: A leap forward in Performance Monitoring. In *Journal of Physics: Conference Series* (2008), vol. 119, Institute of Physics Publishing, p. 042017.

[16] MALZACHER, P., MANAFOV, A., AND SCHWARZ, K. RGLite, an interface between ROOT and gLite – PROOF on the Grid. In *Journal of Physics: Conference Series* (2008), vol. 119, Institute of Physics Publishing, p. 072022.

[17] MALZACHER, P., AND SCHWARZ, K. Grid Activities at GSI. *GSI SR* (2006).

[18] MARCHETTO, F., ATTILI, A., PITTÀ, G., CIRIO, R., DONETTI, M., GIORDANENGO, S., GIVEHCHI, N., ILIESCU, S., AND KRENGLI, M. The INFN TPS project. In *Il Nuovo cimento della Società italiana di fisica. C* (2008).

[19] MATSUI, T., AND SATZ, H. J/ψ suppression by quark-gluon plasma formation. *Physics Letters B 178*, 4 (1986), 416–422.

[20] MEONI, M., AND GROSSE-OETRINGHAUS, J. F. *Alice Offline Tutorial – Part III: PROOF*. CERN, January 2009. `http://aliceinfo.cern.ch/export/sites/AlicePortal/Offline/galleries/Download/OfflineDownload/OfflineTutorial/III_Proof.ppt`.

[21] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: hardware support for efficient processor virtualization. *Intel Technology Journal 10*, 3 (2006), 167–177. http://www.intel.com/technology/itj/2006/v10i3/1-hardware/5-architecture.htm.

[22] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[23] PRINO, F. Open heavy flavour reconstruction in the ALICE central barrel.

[24] RAFELSKI, J., AND LETESSIER, J. Strangeness and quark-gluon plasma. *Journal of Physics G: Nuclear and Particle Physics 30*, 1 (2004), S1–S28.

Acknowledgements – Ringraziamenti

Forse non ho mai ringraziato come si deve le persone che mi hanno seguito e sostenuto durante tutto il mio lavoro, il mio percorso di studi e la mia vita: eccomi qua, a ripetere ancora una volta il mio "grazie", perché sono sicuro che una volta di più non sia mai troppa.

Grazie innanzitutto al mio relatore, Massimo Masera, per avermi proposto questo lavoro di tesi e per avermi dato fiducia e carta bianca, seppur con grande autorevolezza[1]; grazie al mio correlatore, Stefano Bagnasco, per avere avuto l'idea da cui è nata la VAF, ma soprattutto per avermi riportato paternamente sulla strada maestra quando tendevo (come mio solito) a perdermi nei futili dettagli del lavoro[2].

Un grazie ed un abbraccio a tutto il gruppo Pinot che ha accettato di buon grado la mia presenza "clandestina" insieme a loro: Alessandro, Alfredo, Anna, Chiara, Diego (che ora è a Nantes), Francesco, Grazia, Livio, Nora, Pietro, Roberta, Roberto, e i Prof. Chiavassa, Gallio, Scomparin e Vercellin. Dopo la mia assegnazione ai sotterranei di Fisica mi avete offerto asilo salvandomi dalla muffa, ma ho ottenuto anche compagnia, conversazioni piacevoli, salatini e vino rosso...

Grazie a Gerardo Ganis e Marco Meoni del CERN, che si sono dimostrati immensamente disponibili nei miei confronti pur senza avermi mai visto prima.

A big Thank You to Renu Bala, not only for your precious support that helped me testing the VAF and correcting some bugs, but for your good temper and your wonderful chat status messages too :)

[1] E senza mai ridurmi così: `http://www.phdcomics.com/comics.php?n=1139` ;)

[2] Si veda `http://xkcd.com/554/`...

E infine, grazie a tutti i ragazzi e meno ragazzi del centro di calcolo di Torino: Federico, Giuseppe, Riccardo e Stefano, che mi hanno aiutato specialmente nella messa a punto della rete, nell'installazione fisica delle macchine e nel setup della parte Grid della VAF.

❖ ❖ ❖

E ora i ringraziamenti che vanno al di là di questo lavoro.

Grazie ai miei Genitori: a mio Padre, i cui princìpi sono i pilastri della mia esistenza, ed a mia Madre, che mi ha costantemente mostrato con l'esempio che cos'è la dignità.

Grazie a tutti i miei Nonni e ai miei Zii che mi hanno dato una casa in cui studiare tranquillamente in questi ultimi anni, e che mi hanno sempre voluto bene fin da quando sono nato.

Grazie a mio cugino Ruben, sei come un fratello per me.

Grazie a Milli, *mi vida, my honey bee*: i sorrisi hanno parlato per noi prima di mille parole, e farti smettere di sorridere sarebbe il più grave dei crimini ♡

Grazie a Riccardo ed a tutta la sua famiglia: il "grazie" è per aver coltivato insieme la passione per l'informatica fin da quando eravamo piccoli, e per avermi fatto provare per la prima volta un computer; il fatto di avere sempre qualcosa da dirsi dopo 19 anni meriterebbe di più di una semplice citazione nei ringraziamenti di una tesi, quindi non ti ringrazierò qui per questo.

Grazie a Laura, l'amica di sempre con la quale posso fare "i discorsi che non si possono fare con gli amici maschi", grazie soprattutto per quanto mi sei stata vicina in un momento molto difficile.

Grazie a Piergiorgio per la birra e la compagnia durante la mia settimana al CERN, e grazie a Melinda per il suo cuore d'oro.

Un ringraziamento speciale a tutti i miei amici di Fisica che mi hanno sopportato per tutto questo tempo, in particolare Albi, Cisco, Dido, Fede, Francesco (ti ringrazio per la seconda volta, ma te lo meriti perché è difficie trovare persone che poggino le loro idee, qualunque esse siano, su fondamenta solide), Giova (anche

per quella volta che mi hai gridato contro, ma avevi ragione tu), Greg, Jacopo (per la musica e per il nostro modo di santificare le feste, affinché diventi una tradizione), Moreno (o dovrei forse dire *Loris*!?), Roby e Vale (specie per i primissimi esami che preparammo insieme, e qua scatta l'*amarcord* su quanto eravamo giovani). Non ci sono parole che esprimano il mio sentimento per tutti voi, per cui ne userò due inventate: *smurglo* e *cabroglia*[3].

Ah, e grazie alle ragazze (quelle *blu*, non quelle *bianche*) di STBC per aver portato un po' di tocco femminile fra noi Fisici, ed in particolare grazie ad Elisa per avermi portato Milli :)

. . . e come potrei dimenticare i miei *webcomics* preferiti che mi hanno allietato tra una riga di codice e l'altra? Grazie a *xkcd*[4], *The Perry Bible Fellowship*[5], *Tira Ecol*[6] e *Piled Higher and Deeper*[7].

Infine, vorrei ringraziare tutti coloro che studiano e lavorano nell'Università e nella Ricerca in Italia, perché oltre al ben noto problema della mancanza di fondi si è arrivati di recente anche al dileggio ed all'insulto da parte di uno Stato che ci definisce alternativamente "spreconi" e "guerriglieri" in luogo di "risorsa essenziale per lo sviluppo": in una situazione del genere è soltanto la passione per quello che facciamo che ci permette di andare avanti.

. . . *infine,* lectori benevolo*: non posso che ringraziare il Lettore di questa tesi per essere arrivato in fondo, mi auguro senza annoiarsi, o anche per avere solamente letto i ringraziamenti – chiedo venia se non ti avevo ringraziato fino a qui.*

[3] Ok, questa non l'ho inventata io. . .

[4] `http://xkcd.org/`

[5] `http://pbfcomics.com/`

[6] `http://en.tiraecol.net/modules/comic/`

[7] `http://www.phdcomics.com/`