The Evolution of Software in High Energy Physics

René Brun, rene.brun@cern.ch
CERN, Geneva, Switzerland

Abstract. The paper reviews the evolution of the software in High Energy Physics from the time of expensive mainframes to grids and clouds systems using thousands of multi-core processors. It focuses on the key parameters or events that have shaped the current software infrastructure.

1. Introduction

A review of the evolution of the software or hardware in the past 40 or 50 years has been made several times in the past few years. A very interesting article by D.O. Williams [1] was published in 2005 with a general overview of the software evolution and a detailed description of the hardware and networks areas. A book [2] was recently published with a detailed description of the events, systems and people involved in this evolution. The intention of this paper is to focus on a few elements that have been very important in the development of the general tools and libraries commonly used today in HEP. As we are living in a world with an increasing frequency of changes and new features, we must prepare the ground for massive upgrades of our software systems if we want to make an efficient use of the rapidly coming parallel hardware.

The general tendency has been to build a coherent family of systems as illustrated in Figure 1. In the seventies the building blocks were the dynamic structures management systems HYDRA [3] and ZBOOK [4]. The functionality of the two systems was merged into ZEBRA [5] in 1983. The histogram library HBOOK [6] and the minimisation package MINUIT [7] were the two widely used public libraries. The detector simulation system GEANT3 [8] including the first general purpose geometry detector system came in 1981, using the ZBOOK system, then upgraded to ZEBRA in 1983. The Physics Analysis Workstation (PAW) system [9] came in 1984. It was based on ZEBRA, HBOOK and MINUIT. The first version of ROOT [10, 11] came in 1995, strongly influenced by PAW and its components. GEANT4 [12] appeared in 1995 too, implementing in C++ most of the electromagnetic physics from GEANT3 and a very similar geometry system, then making significant developments in most physics processes. Work is currently in progress to design the next step in the GEANT series and GEANT5 (see proceedings of this conference) is expected somewhere in 2015. This version designed for parallel systems will use the ROOT facilities.

2. Hardware: From Mainframes to Clouds

The simulation, reconstruction and analysis phases of HEP experiments had always taken advantage of the cheapest available computing technology. The solutions were based on Mainframes for many years with machines like the CDC6600, CDC7600, CRAY XMP, IBM3090 very common in the scientific domains. In the early 80s supermini systems such as the VAX780...
from DEC became very popular, becoming affordable to many more laboratories or universities. The VAX780 had a user friendly operating system and an excellent development environment. With the advent of workstations from Apollo, Sun Microsystems or SGI in the middle eighties, it became possible to develop new software with better graphics and user interfaces. These workstations were connected with the emergent Ethernet networks. In turn clusters of many workstations became an interesting alternative to mainframes and superminis in the early nineties. When the Intel Pentium processors appeared around 1993, workstations were gradually replaced by PC boxes (both as desktop machines and central production clusters). Up to this point, data files were moved from site to site via magnetic tapes of growing capacity. With the growing network speeds it became possible to move directly large disk files between sites. Tapes were used primarily for backups or primary storage. Coinciding with the LHC data taking, it became possible to use the emerging Grids for all aspects of computing in HEP. The CPU, RAM, disk, network capabilities have grown during the past 40 years following Moore’s law. As shown in Figures 2, 3 the memory available for a program has increased by more than a factor 1000 during this period. It is interesting to note that on this range of machines we saw systems with word sizes of 16 bits (PDP/11, HP), 32 bits (many), 36 bits (Univac), 56 bits (BESM6), 60 bits (CDC) and 64 bits, complicating the data storage and forcing the development of machine independent I/O libraries.

3. Experiments
Like for the hardware, experiments have grown in size from about 10 to 50 physicists in 1975 to about 500 physicists for the LEP experiments to more than 3000 physicists today at the LHC. In the 60s the software was an afterthought written when the experiment data taking was completed. This changed rapidly around 1975 with the SPS experiments, where more layers appeared, introducing in particular common experiment independent libraries. At the same time the need for common detector simulation and analysis software forced the creation of central software support groups in the large laboratories. Central support groups were already setup at the time of Bubble Chambers, but the creation of the central groups for the so-called electronic experiments proved to be a sociological event. Across the years, experiments learnt how to
setup a common software repository, build systems and a shared development environment. For many years standard versions of a software system were maintained via the PATCHY [16] or UPDATE code managers and users were testing their changes via patches relative to the official version. A substantial progress was achieved by the use of distributed code development tools like CVS and SVN or their variants allowing programmers in different places to access and edit a central repository.

Most experiments developed their own frameworks built on top of standard data structure management systems like ZEBRA [5], BOS [14] or Jazelle [15] at the time of Fortran or ROOT [10, 11] in the most recent years.

The current experiment frameworks are large systems that try to hide the system complexity from the average user. These frameworks are typically developed by experts who try to foresee all possible use cases. They force users to think the same way, sometimes generating conflicts during the data analysis phases where physicists call for simpler and more flexible solutions.

4. Languages and Data Structures

For many years Fortran has been the standard language in HEP and other scientific fields. Most physicists learnt Fortran by looking at examples from colleagues. With the growing software base it became rapidly mandatory to use dynamic memory management systems like ZEBRA or BOS, in particular to write/read efficient and portable data files. At the Erice workshop [13] in 1991, the state of the art of all these systems was highly debated at a time when new languages as well as object orientation and data base systems were appearing on the commercial market.

The programming style with ZEBRA or BOS was somehow archaic. For example to access the 3 first words of a bank containing the kinematics information of a track in an array \( \mathbf{Q} \) at offset \( \text{IKINE} \), one had to code 

![Figure 2. Software layers and Systems in 1980](image-url)
REAL px, py, pz
PARAMETER (kPX=1, kPY=2, kPZ=3)
px = Q(1KINE+kPX)
py = Q(1KINE+kPY)
pz = Q(1KINE+kPZ)

where instead with C or C++ one would use directly kine.px, kine.py, kine.pz with kine being a struct or a class object. However, despite this archaic programming style, ZEBRA had many interesting features that experienced Fortran programmers were badly missing when the Physics community moved to C++:

- ZEBRA had the distinction between structural and reference pointers. When a ZEBRA bank was deleted, all its sub-banks pointed to by a structural pointer were deleted too. Tasks such as counting the total memory occupied by a data structure was simple compared to the same task in C++, where complex algorithms had to be developed to avoid circular dependencies. In contrast, one of the most frequent mistakes in C++ is either to forget the destruction of an object in a class destructor, or vice-versa to delete it multiple times. Of course, everything being possible in C++, experts will tell you that you can use a template type and customised it to your needs, or use a special library. However, because ownership is so important in the design of a data structure, this concept should be implemented in a simple and standard way.

- Thanks to these structural pointers, it was easy to visualise a complex data structure such as the detector geometry structure in GEANT3/ZEBRA shown in picture 4. The automatic display of data structures within ZEBRA was a fundamental feature, facilitating the documentation of a data model. This feature was, by far, much more important than the UML diagrams in the Object-Oriented world.

- ZEBRA had the concept of Divisions. A program was typically structured into several divisions, one division for the detector constants (geometry, field, calibrations, etc), one for
the event specific data, one for histograms, one for local dynamic working space. In this way, adding a new data structure inside a division had no impact on the performance (navigation penalty) for structures in other divisions. At the end of an event, it was sufficient to wipe in one call all banks created inside this division, therefore making it hard to generate memory leaks. Divisions were also helping in keeping related data structures in a restricted region in memory, avoiding the catastrophic level 2 cache misses that we experience even with the best new memory allocators in C++. Using memory pools and advanced memory allocators is becoming vital for performance.

We have a major problem with C++ addressing techniques (but also present in all current high level languages). When we call operators like `new` or `malloc` the system returns an absolute pointer in the address space of the running process. This is fine when running with one process only. However if one has to make deep copies of data structures, then the operations involved are quite complex, and even made more complex given the above remark that C++ does not distinguish between structural and reference pointers. A deep copy implies a serialisation of the data structure where the pointers are transformed as simple integer offsets with respect to the buffer where the operation takes place. This operation may become a big bottleneck on parallel systems where, for example, one has to copy from a global shared memory area in the main processor to the local memory of a sub-processor (eg GPU). Even on a simple shared memory system, one cannot currently use C++ pointers across different processes due to the way the virtual table mechanism operates. Let’s consider a frequent and simple example where multiple cores process events in parallel. Each core has an event data structure containing vertices having tracks, having hits, etc. Inside each core the access to the track number $i$ is obtained with

$$\text{Track} \ast \text{track} = \text{event}.\text{track}[i];$$

Figure 4. Track Kinematics bank and Detector Geometry in GEANT3/ZEBRA
Now let’s assume a global thread in charge of I/O operations collecting all events from all cores. With the current C++, we are forced to use the serialisation technique mentioned above. It would be much simpler and faster to imagine code in this thread like:

```cpp
for (core=0; core<ncore; core++) {
    Event *ev = event{core}; // or event.at(core)
    // then loop on all tracks for the event at this core with
    Track *track = ev->track[i];
}
```

where the `ev` and `track` pointers are now valid C++ pointers in the I/O thread (i.e., the system must be able to compute automatically the addressing offsets between all processes, something acting like a fast URL. Without these techniques we run the risk to see ad-hoc data structures serialisation software across processes to populate a central store before doing global operations.

5. I/O and Data Bases

HEP has been traditionally dealing with two kinds of data storage and access problems:

- Event data are stored into highly structured files, yesterday ZEBRA files, today essentially ROOT files. In May 2012, LHC experiments had stored more than 100 Petabytes of data in this format. Event data are typically written once and read many times. They are read sequentially, possibly skipping sections not related to the user analysis tasks. The data files must be highly compressed to save disk space and optimise network transfers. They must be self-describing and support classes schema evolution in an automatic way (addition/removal of data members, renaming, type changes, etc). This is a quite complex process that took many years to master in the ROOT system in an efficient, automatic and simple way. It would be too complex to detail here the different steps and requirements pushing the ROOT system to read complex and evolving data structures in the most efficient way on local disks, local or wide area networks, or simply across shared memory systems on multi-core machines.

- Non-event specific data (calibrations, magnetic fields, run/luminosity information). This type of data is in general stored in relational data bases. Many ad-hoc experiment specific solutions have been developed in this area. This is still an evolving field where many simplifications are required (may be NoSQL).

The days when Object-Oriented Data Bases were proposed to store the two types of information look now very far away. The main solution in 1995 based on the Objectivity commercial system [17] had so many drawbacks that it looks very odd today that this system was considered as a potential candidate for the LHC data between 1995 and 2000! Objectivity had no way to separate transient and persistent data. In-memory data were directly written to a file ignoring portability of data types across machines. Objects were stored object-wise when we know now that storing data member-wise is a must for compression and partial access performance. The concept of a central data base was also in total contradiction with the emerging grids and parallel architectures.

Today most data processing (simulation, reconstruction and data analysis) is performed on thousands of computers in parallel on the grids. We are still learning how to cache efficiently data on local systems to minimise the transfers across wide area networks. With the multi-many core systems, the I/O systems will have to be upgraded to cope with multi-threaded programs. For instance, ROOT Tree branch buffers could be processed in parallel, both in writing and reading. It is also becoming urgent to eliminate the file merging step and replace it by a parallel buffer merging system to drastically reduce the number of I/O operations and make jobs more scalable by eliminating gradually all the sequential phases.
6. Impact of Parallelism
Various types of parallel systems have always been available. In the early eighties, the MegaFlops machines like the CRAY XMP, the CYBER205 or ETA10 were available in several HEP laboratories. We spent a lot of time figuring out how to make efficient use of vectorisation in our programs, in particular GEANT3. Despite the huge manpower investment the exercise was never successful. We used to say that we needed a system processing rapidly MegaIFs rather than MegaFlops code! Looking a posteriori, we were trying to run a program designed for conventional sequential processing with not enough deep changes to create large vectors. We were also penalised by the small amount of RAM (typically 1 MByte) on this expensive type of hardware. Vector machines were quickly abandoned in favour of clusters of RISC workstation processors where our conventional Fortran programs were running with a good efficiency.

In 1990, the MPP (Massively Parallel Processors) systems like the CM2 or CM5 from Thinking Machines with thousands of processors were flourishing. Our naive and tempting approach was to make use of our embarrassing event level parallelism without realising again that massive changes were required in our programs in order to fight the Amdahl’s law problems. In 1993 these machines were quickly abandoned when the Intel Pentium (in particular the Pentium Pro) invaded the PC market at a very low cost. Because we though that Moore’s law will continue to be valid forever for a single processor speed, we continued to improve our good old sequential programs.

The LHC data processing is based on the successful Grid systems with their hierarchy of Tier0, 1, 2 and 3 sites. We expect that the current HEP grid hierarchical infrastructure will evolve to a real grid with more interconnections at all levels, as shown in Figure 5. It should be possible to run jobs anywhere and access data from anywhere. Job-level parallelism is a well understood and successful technique today. Several thousand jobs per experiment are continuously running on more than hundred thousand processors distributed in many sites in the world. This huge success has somehow delayed the effort to parallelise our programs. Since a few years now, it is becoming clear that the clock speed of the new processors will not increase anymore. All the new announced architectures provide at least 2 different levels of parallelism: in short threads or processes at the CPU/core level and vectorisation for specialised embedded processors like GPUs or data flow engines. The only way to gain speed is to put a large number of processor units on the same problem and also save a lot of memory because a lot of code and read only data can be shared. It looks now unavoidable to redesign our millions of lines of code in order to take advantage of the new hardware. In this process we might well make the same mistakes again. The main mistake would be to spend a considerable amount of time in simply adapting our programs to be thread-safe and think only one level (core level) of parallelism.

This new situation is a fantastic opportunity to make a substantial redesign of our most time consuming systems, in particular our detector simulation libraries.

7. Summary
Following the tremendous changes in hardware and networking, the software in High Energy Physics has grown from the infancy times when no detector simulation existed and when the reconstruction software was written by one single person to the current experiments with hundreds of developers and thousands of users. We have moved from small libraries of independent subroutines to hundreds of shared libraries containing complex class hierarchies. Even if it took longer than originally expected, the physics community has moved from the Fortran era to the C++ era. Physicists do not seem scared by using even more languages (eg Python) in fields like data analysis. Meanwhile the computing scene has evolved from mainframes in the main laboratories to a world-wide distributed system thanks to the fantastic developments in the network industry. We are now entering a new phase where many competing types of parallel systems are flourishing. Exploiting these new technologies is a unique opportunity to
Figure 5. Expected evolution of the Grid Systems

rethink the organisation of our large programs and algorithms. The main risks are conservative attitudes, porting existing code with minimal changes to the new system. Instead, HEP software should take the opportunity of the coming long shutdown of the LHC to build a coherent strategy for the years 2015 and beyond. We have seen in the past that software committees are not a solution for an efficient design. Small groups of talented and hard working people are required for this job. One of the obvious areas where expected substantial gains could be achieved is the detector simulation. Let’s hope that the existing GEANT5 prototype presented at this CHEP conference will be a successful demonstrator of these ideas.

8. References
[1] D Williams, "Fifty years of research at CERN from past to future", www.scribd.com/doc/61425123/Cern-History
[2] R Brun, F Carminati, G Carminati, "From the Web to the Grid and beyond", Computing Paradigms driven by High-Energy-Physics, Springer, The Frontiers Collection.
[3] R Bock, E Pagliola, J Zoll, et al. "HYDRA Topical Manual, book MQ, basic HYDRA". CERN, 1981.
[4] R Brun, F Carena, H Grote, M Hansroul, JC Lassalle, and W Wojcik. "ZBOOK – User Guide and Reference Manual", Program Library Q210. CERN, 1984.
[5] R Brun and J Zoll: "ZEBRA – Data Structure Management System". CERN Program Library, Q100. - 1989.
[6] R Brun and D Lienart: "HBOOK User Guide – Version 4". CERN Program Library, Y250. - 1988.
[7] F James: "MINUIT – Interpretation of the Errors on Parameters". CERN Program Library D506 – Supplement. – 1988.
[8] R Brun et al.: “GEANT3 – Detector Description and Simulation Tool”. CERN Program Library Long Writeup W5013. http://wwwasdoc.web.cern.ch/wwwasdoc/geant_html3/geantall.html
[9] R Brun et al.: “PAW – Physics Analysis Workstation”. The Complete CERN Program Library. - 1987. - Version 1.07.
[10] R Brun and F Rademakers, "ROOT - An Object Oriented Data Analysis Framework", Proceedings AIPENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. and Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.
[11] "ROOT A C++ framework for petabyte data storage, statistical analysis and visualization", Computer Physics Communications; Anniversary Issue; Volume 180, Issue 12, December 2009, Pages 2499-2512.
[12] http://geant4.web.cern.ch/geant4/index.shtml  
[13] R Brun, P Kunz, P Palazzi, "Data Structures for Particle Physics Experiments: Evolution or Revolution?" Proceedings of the 14th Workshop of the INFN Eloisatron Project, World Scientific, 1991.  
[14] V Blobel, "The BOS System, Dynamic Memory Management", Report DESY R1-88-01.  
[15] T H Burnett, Computer Physics Communications 45 (1987) 195-199.  
[16] H J Klein and J Zoll. "PATCHY Reference Manual", Program Library L400. CERN, 1988.  
[17] http://www.objectivity.com/