Software challenges for the next decade

DESY: October 29 2007

René Brun
CERN
My mini crystal ball

• Concentrate on HEP software only
• I am biased by the development of large software frameworks or libraries and their use in large experiments.
• Because of inertia or time to develop, there are things easy to predict.
• Technology can come with good or/and bad surprises. See weather Forecast
### Software Challenges for the Next Decade

**OS Machines**
- NORSK
- UNIVAC
- APOLLO
- VAX
- SUN
- HP
- SG
- LINUX
- MAC

**Compiled Languages**
- FORTRAN IV
- F77
- ADA
- OBJC
- F90
- JAVA
- C++

**Code Management**
- UPDATE
- PATCHY
- CMZ
- CVS

**Data Structures**
- ZBOOK
- HYDRA
- ZEBRA
- FATMEN
- ROOT

**I/O**
- BOS
- X11
- NOTIF
- JAVA
- MYSQL/Oracle

**GUI**
- PIONS
- CORE
- GKS
- HIGZ
- GL
- QT

**Graphics**
- GD3
- HBOOK
- AIDA
- ROOT

**Histograms**
- SUMX
- HBOOK
- AIDA
- ROOT

**Statistics**
- SIGMA
- TkTcl
- COMIS
- Perl
- Python
- CINT

**Scripting Interpreters**
- ZCEDEX
- KUIP
- JAS
- ROOT

**Interactive Analysis**
- GEANT1.2
- MCNP
- GHEISHA
- GEANT3

**Detector Simulation**
- EGS
- GEANT1.2
- GEANT3
- GEANT4

**Years**
- 1970
- 1975
- 1980
- 1985
- 1990
- 1995
- 2000
- 2005
- 2010
Time to develop: Main message

- Big latency in the process to collect requirements, make a design, implementation, release and effective use.
- It takes between 5 and 10 years to develop a large system like ROOT or Geant4.
- It takes a few years for users to get familiar with the systems.
- It takes a few years for systems to become de facto standards.
- For example, LHC exps are discovering now the advantages of split-mode I/O designed in 1995 and promoted in 1997.
- This trend will likely continue will the large collaborations.
- This has of course positive and negative aspects. Users like stability, BUT competition is mandatory.
The crystal ball in 1987

- **Fortran** 90X seems the obvious way to go
- OSI protocols to replace TCP/IP
- Processors: Vector or MPP machines
- PAW, Geant3, Bos, Zebra: Adapt them to F90X
- Methodology trend: **Entity Relationship Model**
- Parallelism: **vectorization** or MPP (SIMD and MIMD)

- **BUT** hard to anticipate that
  - The WEB will come less than 3 years later
  - The 1993/1994 revolution for languages and projects
  - The rapid growth in CPU power starting in 1994 (Pentium)
Situation in 1997

- LHC projects moving to C++
- Several projects proposing to use Java
- Huge effort with OODBMS (ie Objectivity)
- Investigate Commercial tools for data analysis
- ROOT development not encouraged
- Vast majority of users very sceptic.

- RAM <256 MB
- Program Size < 32 MB
- <500 KLOCs
- libs < 10
- static linking
- HSM: tape->Disk pool <1 TByte
- Network 2MB/s
The crystal ball in 1997

- **C++** now, **Java** in 2000
- Future is OODBMS (ie Objectivity)
- Central Event store accessed through the net
- Commercial tools for data analysis

- But fortunately a few people did not believe in this direction :):
- First signs of problems with Babar
- FNAL RUN2 votes for ROOT in 1998
- GRID: an unknown word in 1997 :D
Situation in 2007

• It took far more time than expected to move people to C++ and the new frameworks.

• ROOT de facto standard for I/O and interactive analysis.

• The GRID:
  • Experiment frameworks are monsters
Software Hierarchy

- End user Analysis software: 0.1 MLOC
- Experiment Software: 2 MLOC
- Frameworks like ROOT, Geant4: 2 MLOC
- OS & compilers: 20 MLOC
- Hardware
- Networking

Software challenges for the next decade
Challenge 0
Usability: Making things SIMPLER

- Guru view vs user view
- A normal user has to learn too many things before being able to do something useful.
- LHC frameworks becoming monsters
- Fighting to work on 64 bits with <2 GBytes
- Take for ever to start because too much code linked (shared libs with too many dependencies)
- Fat classes vs too many classes
- It takes time to restructure large systems to take advantage of plug-in managers.
Challenge 1
Problem decomposition

Will have to deal with many shared libs
Only a small fraction of code used
Some Facts

100 shared libs
1800 classes

10 shared libs
200 classes

ROOT
In 1995

PAW model
Plug-in manager

ROOT
In 2007
code used in a batch use case

- **Libs used**: 4/86
- **Classes loaded**: 586/1459
- **Classes used (CU)**: 51
- **Methods all libs**: 46438
- **Methods loaded**: 19550
- **Methods in CU**: 2666
- **Methods used in CU**: 325
- **Code in all libs**: 76 Mb
- **Code loaded**: 7.1 Mb
- **Code Really Used (CRU)**: 0.7 Mb
- **Time to compile CRU**: 17 s

Fraction of ROOT code really used in a batch job
Software challenges for the next decade

Fraction of ROOT code really used in a job with graphics

Libs used: 14/86
Classes loaded: 865/1459
Classes used (CU): 142
Methods all libs: 46438
Methods loaded: 26996
Methods in CU: 6676
Methods used in CU: 740
Code in all libs: 76 Mb
Code loaded: 12 Mb
Code Really Used (CRU): 1.2 Mb
Time to compile CRU: 29 s
Fraction of code really used in one program

Per cent of code used

%functions used

%classes used

Lines of code

10^2  10^3  10^4  10^5  10^6  10^7
Large Heap Size Reduction

ROOT size at start-up

Also speed-up start-up time
Challenge 2

Hardware will force parallelism

- Multi-Core (2-8)
- Many-Core (32-256)
- Mixture CPU + GPU-like
- Virtualization
- May be a new technology?
- Parallelism: a must
Challenge 3
Design for Parallelism

• The GRID is a parallel engine. However you will not use the GRID software on your 32-core laptop.
• Minimize globals and make tasks as independent as possible.
• Be thread-safe and better thread-capable
• Think Top->Down and Bottom->Up

Coarse grain: job, event, track
Fine grain vectorization
Parallelism: Where?

- Multi-Core CPU laptop/desktop
- Network of desktops
- Local Cluster
  - with multi-core CPUs
- GRID(s)
Challenge 4
Design for Client-Server

- The majority of today’s applications are client-server (xrootd, Dcache, sql, etc).
- This trend will increase.
- Be able to stream objects or objects collections.
- Server logic robust against client changes.
- Server able to execute dynamic plug-ins.
- Must be robust against client or network crash
Challenge 5
Sophisticated Plug-in Managers

• When using a large software base distributed in hundred of shared libs, it is essential to discover automatically where to find a class.

• The interpreters must be able to autoload the corresponding libraries
Challenge 6
The Language Reflexion System

• Develop a robust dictionary system that can be migrated smoothly to the reflexion system to be introduced in C++ in a few years.
• Meanwhile reduce the size of dictionaries by doing more things at run time.
• Replace generated code by objects stored in ROOT files.
• Direct calls to compiled code from the interpreter instead of function stubs. This is compiler dependent (mangling/de-mangling symbols).
Today `cint/reflex` dictionaries are machine dependent. They represent a very substantial fraction of the total code. We are now working to reduce this size by at least a factor 3!
Challenge 7
Opportunistic Use of Interpreters

• Use interpreted code only for:
  – External and thin layer (task organizer)
  – Slots execution in GUI signal/slots
  – Dynamic GUI builder in programs like event displays.

• Instead optimize the compiler/linker interface (eg TACLIC) to have
  – Very fast compilation/linking when performance is not an issue
  – Slower compilation but faster execution for the key algorithms

• ie use ONE single language for 99% of your code and the interpreter of your choice for the layer between shell programming and program orchestration.
Challenge 8
LAN and WAN I/O caches

• Must be able to work very efficiently across fat pipes but with high latencies.
• Must be able to cache portions or full files on a local cache.
• This requires changes in data servers (Castor, Dcache, xrootd). These tools will have to interoperate.
• The ROOT file info must be given to these systems for optimum performance. See TTreeCache improvements.
Disk cache improvements with high latency networks

- The file is on a CERN machine connected to the CERN LAN at at 100MB/s.
- The client A is on the same machine as the file (local read)
- The client F is connected via ADSL with a bandwidth of 8Mbits/s and a latency of 70 milliseconds (Mac Intel Coreduo 2Ghz).
- The client G is connected via a 10Gbits/s to a CERN machine via Caltech latency 240 ms.
- The times reported in the table are realtime seconds

<table>
<thead>
<tr>
<th>client latency (ms)</th>
<th>cache size = 0</th>
<th>cache size = 64KB</th>
<th>cache size = 10MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.0</td>
<td>3.4</td>
<td>3.4</td>
</tr>
<tr>
<td>F</td>
<td>72.0</td>
<td>743.7</td>
<td>48.3</td>
</tr>
<tr>
<td>G</td>
<td>240.0</td>
<td>&gt;1800s</td>
<td>125.4s</td>
</tr>
</tbody>
</table>

One query to a 280 MB Tree

I/O = 16.6 MB

We expect to reach 4.5 s
I/O: More

- Efficient access via a LAN AND WAN
- Caching
- Better schema evolution
- More support for complex event models
- zip/unzip improvements (separate threads)
- More work with SQL data bases
Challenge 9

Code Performance

- HEP code does not exploit hardware (see S.Jarp talk at CHEP)
- Large data structures spread over >100 Megabytes
- templated code pitfall
  - STL code duplication
  - good perf improvement when testing with a toy.
  - disaster when running real programs.
- std::string passed by value
- abuse of new/delete for small objects or stack objects
- linear searches vs hash tables or binary search
- abuse of inheritance hierarchy
- code with no vectors -> do not use the pipeline
Compilation Time

Lines of code per library

- Lines of code compiled + DLL: 1439990
- Time to compile: 1759.7 seconds
- \( <\text{lines/second}> : 818.316 \)
- \( <\text{lines/second for libs with } <T> : 261.112 \)
- \( <\text{lines/second for C libs} : 1926.19 \)
- \( <\text{lines/second for other libs} : 866.254 \)
- Time to compile if all with \( <T> : 5514.84 \text{ s} \)
- Time to compile if all without \( <T> : 1662.32 \text{ s} \)
- Time to compile if all in C \( : 747.586 \text{ s} \)

C-like code

templated code
## LHC Software

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Atlas</th>
<th>CMS</th>
<th>ROOT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of Lines in Header Files</strong></td>
<td>102282</td>
<td>698208</td>
<td>104923</td>
<td>153775</td>
</tr>
<tr>
<td><strong>Classes Total</strong></td>
<td>1815</td>
<td>8910</td>
<td>???</td>
<td>1500</td>
</tr>
<tr>
<td><strong>Classes in Dict</strong></td>
<td>1669</td>
<td>&gt;4120</td>
<td>835</td>
<td>1422</td>
</tr>
<tr>
<td></td>
<td>2140</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Lines in Dict</strong></td>
<td>479849</td>
<td>455705</td>
<td>103057</td>
<td>698000</td>
</tr>
<tr>
<td><strong>Classes C++ Lines</strong></td>
<td>577882</td>
<td>1524866</td>
<td>277923</td>
<td>857390</td>
</tr>
<tr>
<td><strong>Total Lines Classes+dict</strong></td>
<td>1057731</td>
<td>???</td>
<td>380980</td>
<td>1553390</td>
</tr>
<tr>
<td><strong>Total f77 Lines</strong></td>
<td>736751</td>
<td>928574</td>
<td>???</td>
<td>3000</td>
</tr>
<tr>
<td><strong>Directories</strong></td>
<td>540</td>
<td>19522</td>
<td>&lt;500</td>
<td>958</td>
</tr>
<tr>
<td><strong>Comp Time</strong></td>
<td>25’</td>
<td>750’</td>
<td>90’</td>
<td>30’</td>
</tr>
<tr>
<td><strong>Lines Compiled/s</strong></td>
<td>1196</td>
<td>50 (70)</td>
<td>71</td>
<td>863</td>
</tr>
</tbody>
</table>

Software challenges for the next decade
Challenge 10
Towards Task-oriented programming

- Browsing
- OS files
- Data hierarchy
- Dynamic tasks
Challenge 11
Customizable and Dynamic GUIs

• From a standard browser (eg ROOT TBrowser) on must be able to include user-defined GUls.
• The GUls should not require any pre-processor.
• They can be executed/loaded/modified in the same session.
Browser Improvements

- The browser (TBrowser and derivatives) is an essential component (from beginners to advanced applications).
- It is currently restricted to the browsing of ROOT files or Trees.
- We are extending TBrowser such that it could be the central interface and the manager for any GUI application (editors, web browsers, event displays, etc).
TGhtml web browser plug-in

- You can browse a root file
- You can execute a script
Macro Manager/Editor plug-in

Click on button to execute script with CINT or ACLIC

Software challenges for the next decade
GL Viewer plug-in

Alice event display prototype using the new browser

Software challenges for the next decade
Challenge 12
Executing Anywhere from Anywhere

- One should be able to start an application from any web browser.
- The local UI and GUI can execute transparently on a remote process.
- The resulting objects are streamed to the local session for fast visualization.
- Prototype in latest ROOT using ssh technology.

```
root > .R lxplus.cern.ch
lxplus > .x doSomething.C
lxplus > .R
root > //edit the local canvas
```
Challenge 13
Evolution of the Execution Model

• From stand alone modules
• To shared libs
• To plug-in managers
• To distributed computing
• To distributed and parallel computing
Executable module in 1967

- x.f -> x.o -> x.exe

Input.dat → x.exe → Output.log
Executable module in 1977

- x.f $\rightarrow$ x.o
- x.o + libs.a $\rightarrow$ x.exe

- x.exe
- Input.dat
- non portable binary file
- Output.log
Executable module in 1987

- many_x.f -> many_x.o
- many_x.o + many_libs.a -> x.exe

Input.dat (free format)

portable Zebra file

Output.log

Software challenges for the next decade
Executable module in 1997

- many_x.f -> many_x.o
- many_x.o + some_libs.a
- + many_libs.so -> x.exe

- Input.dat (free format)
- Zebra file
- RFIO
- x.exe
- Objectivity? ROOT?
- Output.log

Software challenges for the next decade
Executable module in 2007

**u.so**

**a.so**

**b.so**

**x.exe**

- **Config.C (interpreter)**
- **Output.log**
- **ROOT files**
- **Dcache**
- **castor**
- **xrootd**

- **Shared libs dynamically loaded/unloaded by the plug-in manager**

**LAN**

- **Oracle**
- **Mysql**

**ROOT files**

Software challenges for the next decade
Challenge 14
Data Analysis on the GRID

5,000 physicists
in 1000 locations

100,000 computers
in 1000 locations

LAN

WAN
GRID: Users profile

Few big users submitting many long jobs (Monte Carlo, reconstruction)
They want to run many jobs in one month

Many users submitting many short jobs (physics analysis)
They want to run many jobs in one hour or less
Big but few Users

- Monte Carlo jobs (one hour → one day)
  - Each job generates one file (1 GigaByte)
- Reconstruction job (10 minutes -→ one hour)
  - Input from the MC job or copied from a storage centre
  - Output (< input) is staged back to a storage centre
- Success rate (90%). If the job fails you resubmit it.
- For several years, GRID projects focused effort on big users only.
Small but many Users

- **Scenario 1**: submit one batch job to the GRID. It runs somewhere with varying response times.
- **Scenario 2**: Use a splitter to submit many batch jobs to process many data sets (e.g., CRAB, Ganga, Alien). Output data sets are merged automatically. Success rate < 90%. You see the final results only when the last job has been received and all results merged.
- **Scenario 3**: Use PROOF (automatic splitter and merger). Success rate close to 100%. You can see intermediate feedback objects like histograms. You run from an interactive ROOT session.
GRID & Parallelism: 1

- The user application splits the problem into N subtasks. Each task is submitted to a GRID node (minimal input, minimal output).
- The GRID task can run synchronously or asynchronously. If the task fails or time-out, it can be resubmitted to another node.
- One of the first and simplest uses of the GRID, but not many applications in HEP.
- Examples are SETI, BOINC
GRID & Parallelism: 2

- The typical case of Monte Carlo or reconstruction in HEP.
- It requires massive data transfers between the main computing centres.
- This activity has concentrated so far a very large fraction of the GRID projects and budgets.
- It has been an essential step to foster coordination between hundreds of sites, improve international network bandwidths and robustness.
GRID & Parallelism: 3

• Distributed data analysis will be a major challenge for the coming experiments.
• This is the area with thousands of people running many different styles of queries, most of the time in a chaotic way.
• The main challenges:
  – Access to millions of data sets (eg 500 TeraBytes)
  – Best match between execution and data location
  – Distributing/compiling/linking users code (a few thousand lines) with experiment large libraries (a few million lines of code).
  – Simplicity of use
  – Real time response
  – Robustness.
GRID & Parallelism: 3a

- Currently 2 different & competing directions for distributed data analysis.
  - **Batch** solution using the existing GRID infrastructure for Monte Carlo and reconstruction programs. A front-end program partitions the problem to analyze ND data sets on NP processors.
  - **Interactive** solution PROOF. Each query is parallelized with an optimum match of execution and data location.
Scenario 1 & 2: PROS

• Job level parallelism. Conventional model. Nothing to change in user program.
  – Initialisation phase
  – Loop on events
  – Termination

• Same job can run on laptop or on a GRID job.
Scenario 1 & 2: CONS(1)

- Long tail in the jobs wall clock time distribution.
Scenario 1 & 2: CONS(2)

- Can only merge output after a time cut.
- More data movement (input & output)
- Cannot have interactive feedback
- Two consecutive queries will produce different results (problem with rare events)
- Will use only one core on a multi core laptop or GRID node.
- Hard to control priorities and quotas.
Scenario 3: PROS

- Predictive response. Event level parallelism. Workers terminate at the same time.
- Process moved to data as much as possible, otherwise use network.
- Interactive feedback
- Can view running queries in different ROOT sessions.
- Can take advantage of multi core cpus
Scenario 3: CONS

- Event level parallelism. User code must follow a template: the TSelector API.
- Good for a local area cluster. More difficult to put in place in a GRID collection of local area clusters.
- Interactive schedulers, priority managers must be put in place.
- Debugging a problem slightly more difficult.
Challenge 15
Languages

• C++ clear winner in our field and also other fields

• see, eg a recent compilation at [http://www.lextrait.com/vincent/implementations.html](http://www.lextrait.com/vincent/implementations.html)

• From simple C++ to complex templated code

• Unlike Java, no reflexion system. This is essential for I/O and interpreters.

• C++2009: better thread support, Aspect-oriented

• C++2014: first reflexion system?
Challenge 16
Software Development Tools

- better integration with Xcode, VisualStudio or like
- fast memory checkers
- faster valgrind
- faster profilers
- Better tools to debug parallel applications
- Code checkers and smell detection
- Better html page generators
Challenge 17
Distributed Code Management

- patchy, cmz -> cvs
- cvs -> svn
- cmt? scram? (managing dependencies)
- automatic project creation from cvs/svn to VisualStudio or Xcode and vice-versa
Challenge 18
Simplification of Software Distribution

- tar files
- source + make
- install from http://source
- install from http://binary proxy
- install on demand via plugin manager, autoloader
- automatic updates
- time to install
- fraction of code used

See BOOT Project
First release
In June 08
Challenge 19
Software Correctness

• -big concern with multi million lines of code
• -validation suite
• -unit test
• -combinatorial test
• -nightly builds (code + validation suite)
Challenge 20
Scalable Software Documentation

• Legacy Doxygen
• Need for something more dynamic, understanding Math, Latex, 2-D and 3-D graphics, interactive tutorials.
• See results of new THtml at:
Challenge 21

Education

• Training must be a continuous effort
• Core Software guys often desperate with newcomers.
• Software Engineering and discipline required to participate to large international projects is absent in University programs.
Summary

- A large fraction of the software for the next decade already in place or shaping up.
- Long time between design and effective use.
- Core Software requires Open Source and international cooperation to guarantee stability and smooth evolution.
- Parallelism will become a key parameter
- More effort must be invested in software quality, training and education.
Summary-2

• But the MAIN challenge will be to deliver **scalable systems**:

• **Simple to use** for beginners with some very basic rules and tools.

• Browsing (understanding) an ever growing dynamic code and data will be a must.
Summary-3

• Building large software systems is like building a large city. One needs to standardize on the communication pipes for input and output and setup a basic set of rules to extend the system or navigate inside.