Parallel computing with BEAN - ROOT-based BES-III Analysis Framework

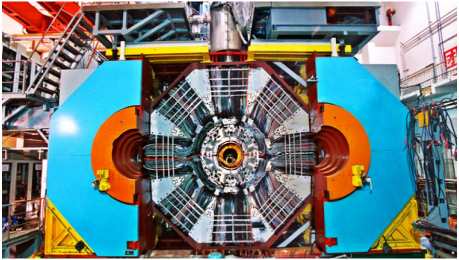
Evgeny Boger

JINR Dubna
LNP NEOVP

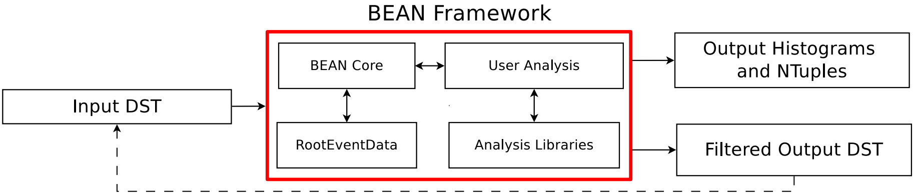
NEC'2015

Outline

- BEAN in a nutshell
- PROOF - The Parallel ROOT Facility
- Using Hadoop for parallel computing in HEP



BEAN — ROOT-based analysis framework



Lightweight tools for interactive analysis of DST

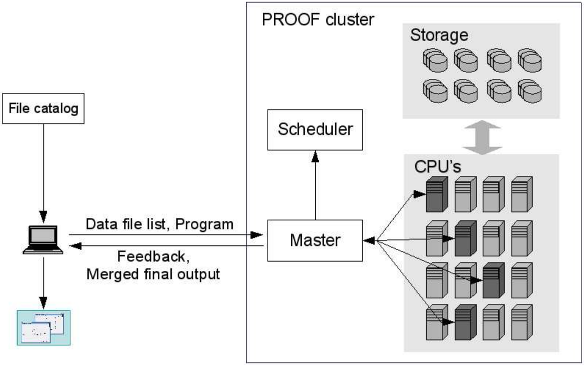
- For interactive analysis code development
- For iterative event filtration
- Fast and optimized for standalone use
- Low number of external dependecies (2 so far)

The main data flow

- Input format = Output format = BESIII DST
- User histograms and ntuples are saved in output root-file.

BEAN parallelization with PROOF

- The PROOF is a part of the ROOT enabling an analysis of large sets of ROOT files in parallel on clusters of computers or many-core machines
- The main idea of Bean is to run on the PROOF system with minimal changes in the user interface



How to run Bean in PROOF mode?

Bean in PROOF mode is a transparent extension of single user session

Example (run Bean at local PC)

```
> ./bean.exe  -u MyAnalysis   root://besdata.jinr.ru//data/bes3/run.dst \
    -h histo.root  -o selected_events.root
```

Example (run Bean in PROOF-Lite mode)

```
> ./bean.exe  -u MyAnalysis   root://besdata.jinr.ru//data/bes3/run.dst \
    -h histo.root  -o selected_events.root  -l
```

Example (run Bean in PROOF-cluster mode)

```
> ./bean.exe  -u MyAnalysis   root://besdata.jinr.ru//data/bes3/run.dst \
    -h histo.root  -o selected_events.root   -p “xrootd@lgdui01”
```

PROOF Summary

Advantages

- Tightly integrated with ROOT
- Very fast (thanks to real-time load balancing and low startup time)
- Data locality is taken into account
- Stable and convinient PROOF-Lite mode (single PC)

Disadvantages

- Again, tightly integrated with ROOT
- Somewhat fragile
- Sensitive to memory leaks
- No partial result support
- Not suited well for large (1k+ nodes) clusters
  - Real-time load balancing doesn't scale well
- No decent scheduling

What is Hadoop?

Apache Hadoop

An open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware.

- Created in 2008. Widely adopted througout the industry.
- Provides reliable distributed file system (HDFS)
- Provides framework for job scheduling and cluster resource management (Hadoop YARN)
- Hadoop MapReduce - system for parallel processing of large data sets built on top of YARN. Follows the “Moving Compute to Data” paradigm.
- Not limited to MapReduce: HBase, Hive, Pig, Crunch, Spark, etc.

Why Hadoop?

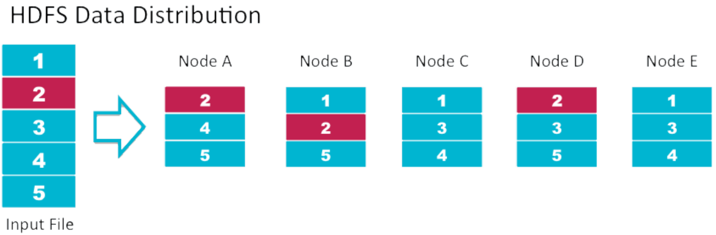
- Industry standard (Yahoo!, Facebook, Baidu, Yandex, etc.)
- Actively maintained and developed by the industry
- Scalable up to 10 000 nodes and 100 000 tasks.
- Reliable
- Commercial support is available from a number of companies
- Typical HEP physics analyses fit well to MapReduce paradigm

Hadoop meets HEP

- Hadoop MapReduce was designed to process (structured) text
- Hadoop MapReduce tasks operate on row-oriented data formats
- Hadoop is written in Java. Python and C++ support are very limited.
- Hadoop development is focused on scalability rather than performance

HDFS Blocks

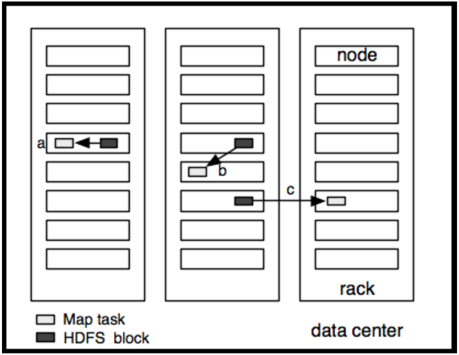
- Hadoop HDFS is a write-once, read-many filesystem
- Files are divided to even-sized blocks as they created (128MB)
- Blocks are being independently replicated and managed
- Blocks from the single file will end up stored on different machines
- HDFS file is a mere list of block ids



http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/hdfs-and-mapreduce.html

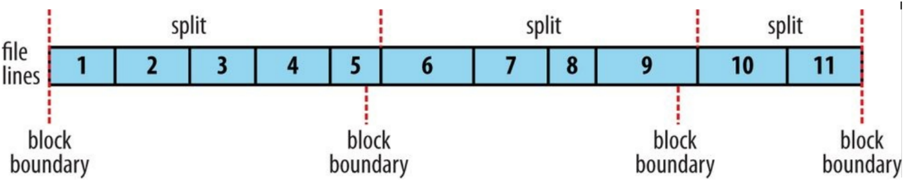
Data Locality and Hadoop Map Taks

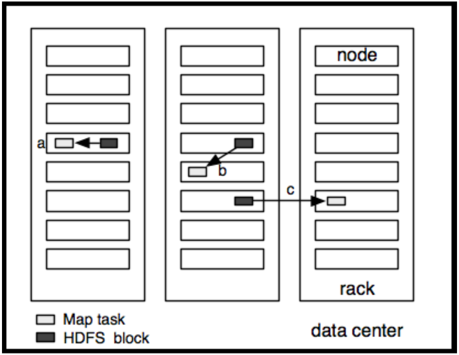
- Each Map Task has its pre-assigned set of data to process (called InputSplit)
- Data locality is taken into account when creating InputSplits
- Typicaly InputSplit is equal in size to one HDFS block (i.e. 128MB)



a) Data-local b)rack-local c)off-rack map tasks

Data Locality and Hadoop Map Taks





a) Data-local b)rack-local c)off-rack map tasks

Using Hadoop with ROOT-based analysis: Task Input

This simple idea was originally proposed in [1]

- Store ROOT files as it is in HDFS
- Make ROOT files consist of no more than one HDFS block
  - Increase HDFS block size to reasonable extent (2GB)
  - typical ROOT files are smaller than 2GB
- Single ROOT file ⇔ single InputSplit ⇔ single Map task

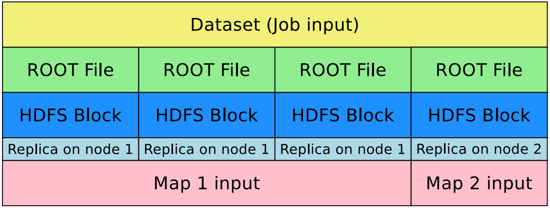
Dataset (Job input)		
ROOT File	ROOT File	ROOT File
HDFS Block	HDFS Block	HDFS Block
Map 1 input	Map 2 input	Map N input

---

[1] Running a typical ROOT HEP analysis on Hadoop MapReduce, S A Russo et al, CHEP2013

Task Input Optimization

- Single InputSplit contains one or more ROOT files
- Data locality is still taken into account
- The deisred size (in MB) of each split is configurable. Boundary cases are
  - Single ROOT file per Map
  - Single Map per Node
- Less overhead from launching JVM and ROOT instances
- More sensitive to failures (such as memory leaks)



Dataset (Job input)			
ROOT File	ROOT File	ROOT File	ROOT File
HDFS Block	HDFS Block	HDFS Block	HDFS Block
Replica on node 1	Replica on node 1	Replica on node 1	Replica on node 2
Map 1 input			Map 2 input

Custom implementation of CombineInputFileFormat is used

Analysis task I/O

Now we want to run ROOT analysis on the set of ROOT files

- The idea is to prevent Hadoop from reading the data and splitting it to the records
- Instead, the HDFS file URL is extracted on task startup. The HDFS file is then downloaded and processed by external application.

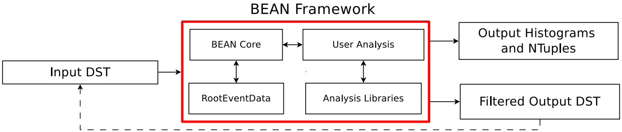
Accessing HDFS files

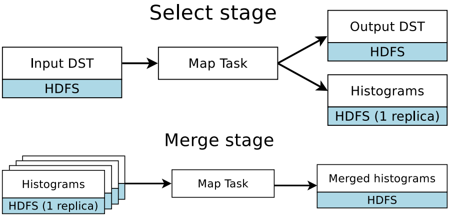
- Copy to temporary local location from JAVA code or external code
- Use FUSE
- Use ROOT I/O plugin (THDFSFile [1])

Optimizing reads

- Remember to enable read short-circuit for HDFS datanodes and clients
- Use libhdfs3 - experimental native HDFS library

Analysis job





Hadoop job sequence

- First map-only job processes input
- Second map-only job merges output histograms to single HDFS file
- Merging may be skipped or done by client
- Output DST files are left on HDFS unmerged to be used as dataset

Output

- “Task side-effect files” are used to produce output
- ROOT writes to HDFS via temporary local files

Generic tool to run ROOT-based analyses on Hadoop

- So far everything wasn't BEAN-specific
- It wasn't even ROOT-specific
- Generic RunOnHadoop.jar JAVA class is provided
- User need to provide the class with selector executable, merger executable (optional) and input and output locations
- List of input files, output locations and other parameters are passed to the executable via environment variables
- Stdout and stderr are saved to job history

Example

```
$ hadoop jar RunOnHadoop.jar RunOnHadoop -files wrapper.py
-archives Bean.zip#Bean hdfs_input hdfs_output
```

BEAN on Hadoop

- It's not hard to run PROOF-aware code on Hadoop
- PROOF support codebase is heavily reused
- Some boilerplate code is used in selector wrapper script to create TChain from HDFS input files and run TSelector on it
- PROOF PAR packages are reused to send user analysis and libraries to worker nodes via Hadoop Distributed Cache.

TODO

- Properly document and share the source code [1]
- Find a way to directly output ROOT files to HDFS
- Implement TProofPlayer interface to mimic PROOF
- Try a (very) experimental native Hadoop job API to get rid of JVM completely

---

[1] Will be available soon at http://bes3.jinr.ru

Credits

- We thank JINR cloud team, namely Nikolay Kutovskiy and Aleksandr Baranov, for providing us with cloud-based cluster for testing purposes.

BACKUP

TSelector

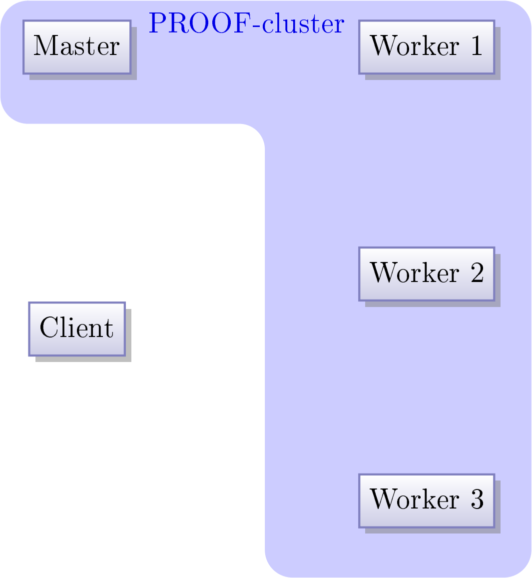
Analysis code to be used with PROOF should be organized as TSelector ancestor class. This class (called selector) is ought to process the single ROOT TTree.

- Begin - executed on client prior to processing
- SlaveBegin - executed on worker nodes prior to processing
- Notify - called by PROOF when the new file is about to be processed
- Process - process the single event
- SlaveTerminate - executed on worker nodes at the end
- Terminate - executed on client at the end

Example (Example PROOF usage)

```
root [0] tree->Process(”ana.C“)
root [1] TProof::Open(”remote”)
root [2] chain->SetProof();
root [3] chain->Process(“ana.C”)
```

PROOF terminology



PROOF-cluster

Master

Worker 1

Worker 2

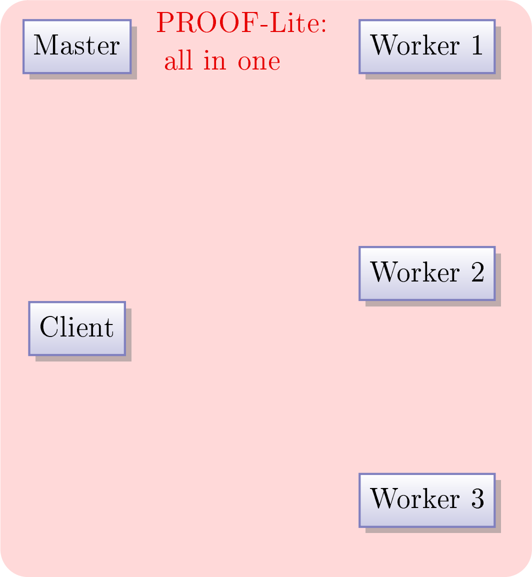
Worker 3

Client

Terminology

- Client:
  Your machine running a ROOT session that is connected to a Master
- Master:
  PROOF machine coordinating work between Workers
- Worker:
  PROOF machine that processes data
- PROOF-Lite:
  Client, Master and Workers are one multicore / multiprocessor PC.

PROOF terminology



PROOF-Lite:
all in one

Terminology

- Client:
  Your machine running a ROOT session that is connected to a Master
- Master:
  PROOF machine coordinating work between Workers
- Worker:
  PROOF machine that processes data
- PROOF-Lite:
  Client, Master and Workers are one multicore / multiprocessor PC.

PROOF Datasets

- ROOT Dataset is a named list of files which can additionally store some meta-information.
- Accessed via TFileCollection
- PROOF usage:
  - dataset can be registered on Master
  - dataset can be checked. Meta-information is being filled on this step
  - User can retrieve list of datasets from Master
  - User can retrieve the single dataset by its name
  - TProof::Process can be used on dataset instead of TChain

Example

root [2] gProof->Process(“Dataset1”, “tutorials/tree/h1analysis.C+”)

PAR Packages

- PAR (Proof Archive)
- .tar.gz archive containing files and meta-data.
  - shell script BUILD.sh - called every time package is updated. Usually contains some building sequence, like call to Make.
  - ROOT script SETUP.C, called every time package is used. Usually handles some kind of dependency control and the loading of shared libraries.
- PAR package can be distributed on worker nodes by request
- Clients are claiming which PAR packages they are using
- PROOF implements version control on packages