

# Multi-processing in ROOT

## Status and Plans

G. Ganis, CERN, PH-SFT

3<sup>rd</sup> Annual Concurrency Forum Meeting  
CERN, 2-3 April 2014

# Multi-processing

- Another way to achieve parallelism
  - Only one possible across machines
- Advantages
  - Code encapsulation
  - Heterogeneous usage
    - From multi-core to multi-node
- Disadvantages
  - Possible setup overhead
    - Process launch, environment customization, ...
  - Need to merge the results

# Multi-processing in ROOT: PROOF

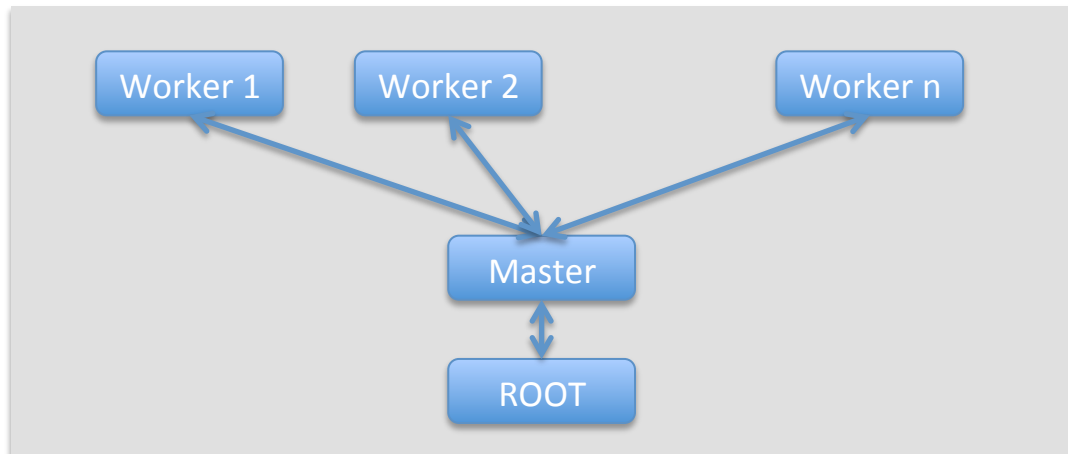
- Main goal: increase effective I/O bandwidth in processing a dataset
  - Exploiting embarrassing parallelism in the data
- Flexible target
  - Potentially extremely large facilities
    - Dedicated, Batch- or Cloud-managed
    - PROOF-On-Demand, Virtual Analysis Facility
  - Or large number of in-node cores
    - PROOF-Lite

# Outline

- Brief reminder
- Recent developments
  - Dynamic workers
  - Debugging, Benchmarking
  - Merging
- Current plans

# PROOF in a nutshell

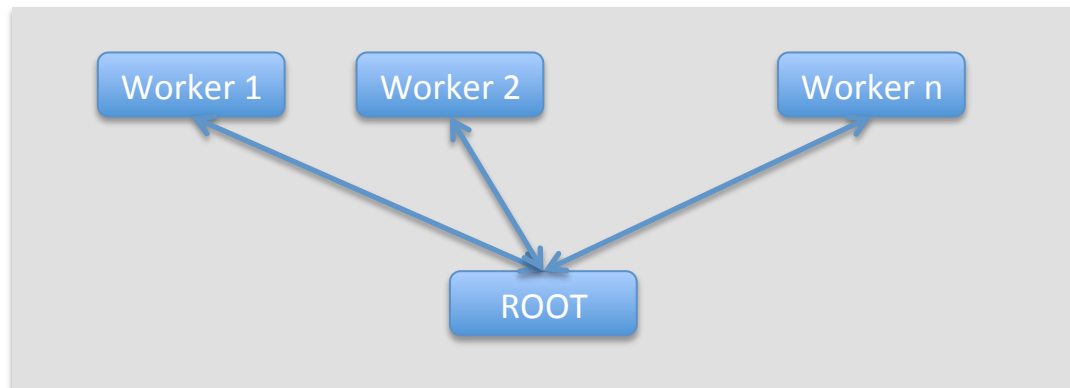
- ROOT processes working concurrently
  - Master-worker architecture



- Pull-architecture for work distribution
- Processes started via a dedicated daemon
  - Communication via TCP sockets

# PROOF-Lite in a nutshell

- ROOT client session acts as master



- Processes started from the shell (system)
  - Communication via UNIX sockets

# PROOF(-Lite) user interface

- Create the session

```
TProof::Open("url")      "pod://", "lite://", "master"
```

- Set up the environment

```
TProof::Load(...)  
TProof::EnablePackage(...), ...
```

- Run

```
TProof::Process(dataset, selector, ...)  
TProof::DrawSelect(dataset, varexp, selection, ...)  
TProof::Process(selector, cycles, ...)
```

- Interface with **TChain**

```
TChain::SetProof()
```

TSelector



Non data driven



# (More or less) recent developments

- Setup evolution
  - Solving stability issues
- Exploiting master-worker interactivity
  - Dynamic workers
- Usability, performance, debugging
  - Access to dataset meta-information
    - Interface with experiment catalogues
  - Output handling
    - Merging, file saving
  - TProofBench
  - Interface with igprof, valgrind



# Setup evolution

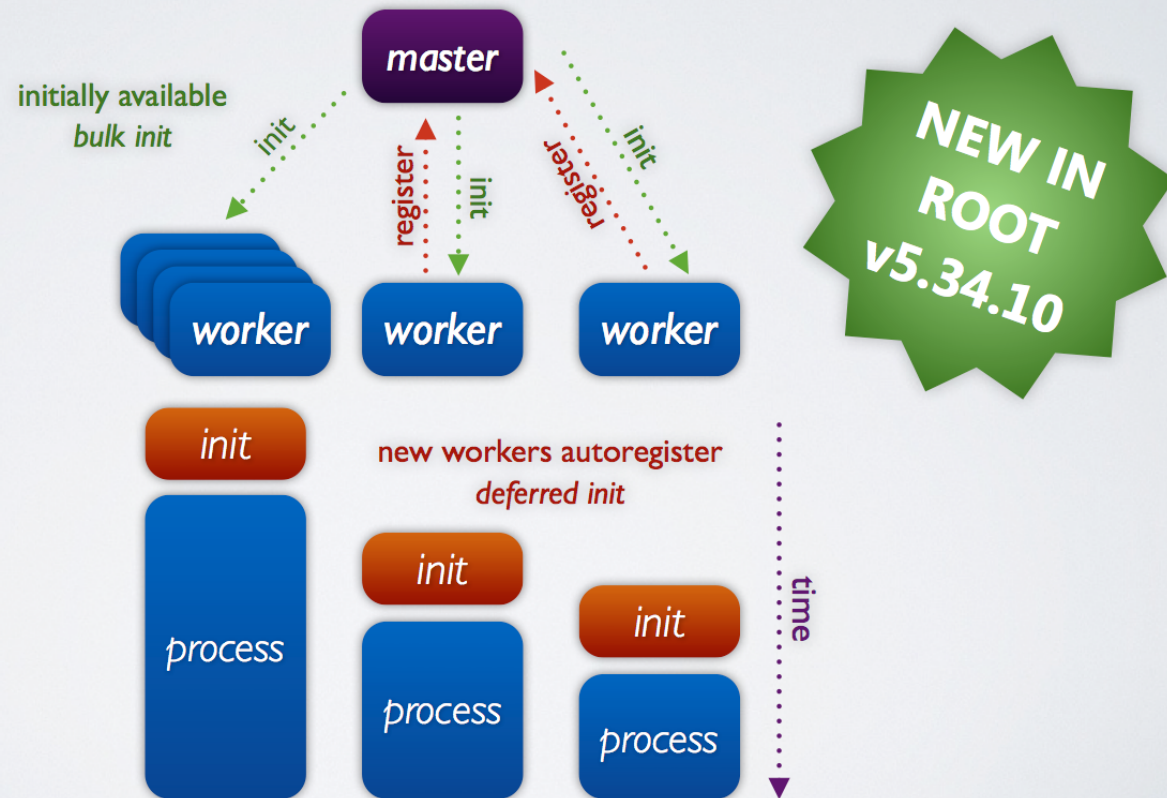
- Setup based on Proof-on-Demand (PoD)
  - Plug-in interface to Resource Management Systems
    - HTCondor, LSF, PBS, ...
    - Dedicated/static resources via 'ssh'
  - Delegates authentication, sandboxing, priorities
- Virtual Analysis Facility
  - Proof-As-A-Service on cloud-managed resources
  - Dynamic scale-up/down
  - See D. Berzano talk at CHEP 2013



# PROOF is cloud-aware



*Dynamic addition of workers*  
*new workers can join and offload a running process*

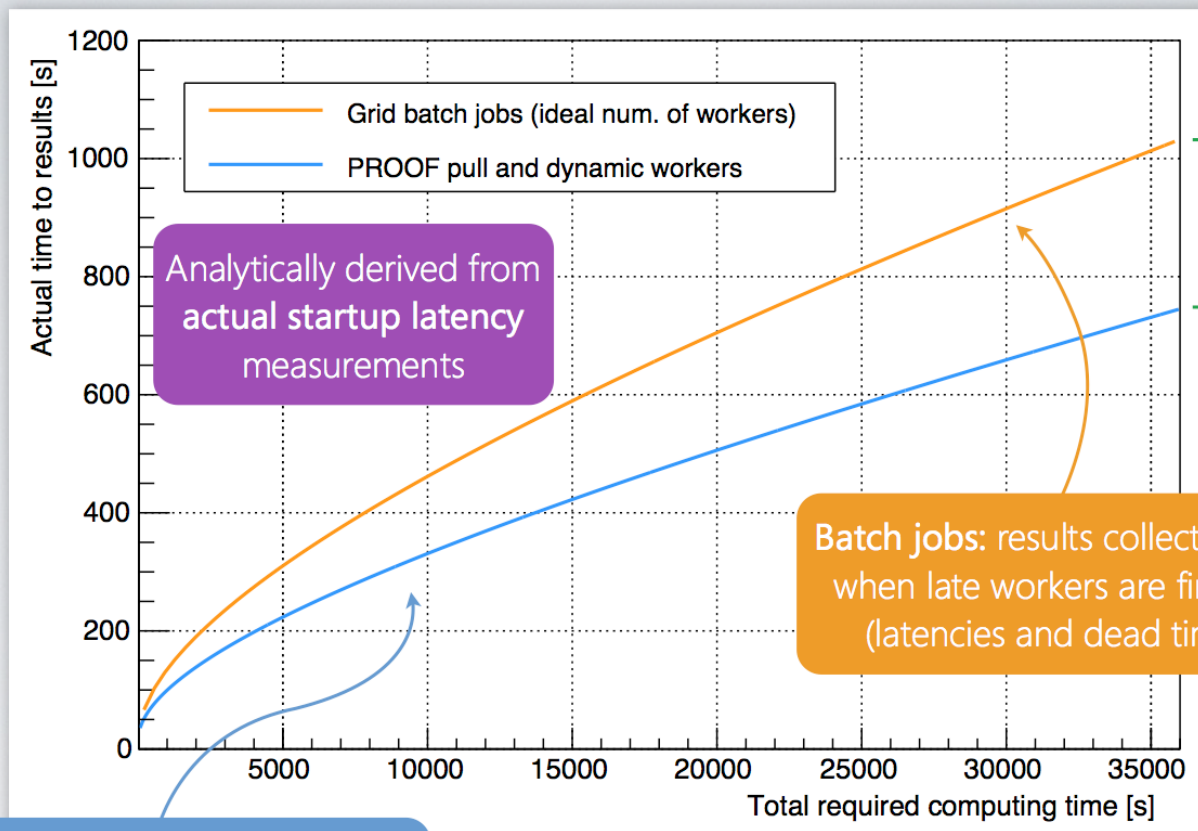


Dario.Berzano@cern.ch - PROOF as a Service on the Cloud - <http://chep2013.org/contrib/308>

7



# PROOF dynamic workers



Analytically derived from actual startup latency measurements

Batch jobs: results collected only when late workers are finished (latencies and dead times)

PROOF with Dynamic Workers: all job time spent in computing (never idle, no latencies)

PROOF is up 30% more efficient on the same computing resources by design (analytical upper limit)

# Merging issues

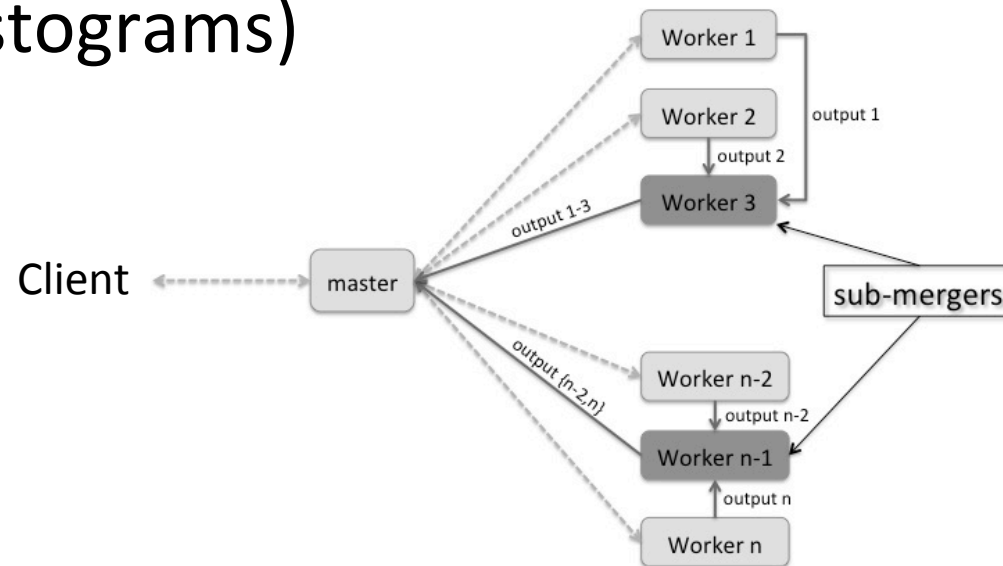
- Performance
  - Serial phase at the end
    - Limits scalability
- Resource requirements
  - Scales with output size
  - Large outputs are the norm
    - 10000's N-d histograms, big trees
  - RAM, network issues, ...

# Merging in PROOF

- Mostly solutions to optimize resource utilization
- Merge objs 1-by-1 (not N object in one go)
  - Limit required RAM to twice the biggest object
    - Recent optimizations on this (5.34/12+)
- Merge via file
  - Workers save to file, master runs TFileMerger
- For TTree outputs
  - Create metadata for transparent access (e.g. TChain)
    - May optimize subsequent access

# Parallel merging with submerges

- Submerger: faster worker promoted merger
- Helps improving performance with objects of fixed size (e.g. histograms)



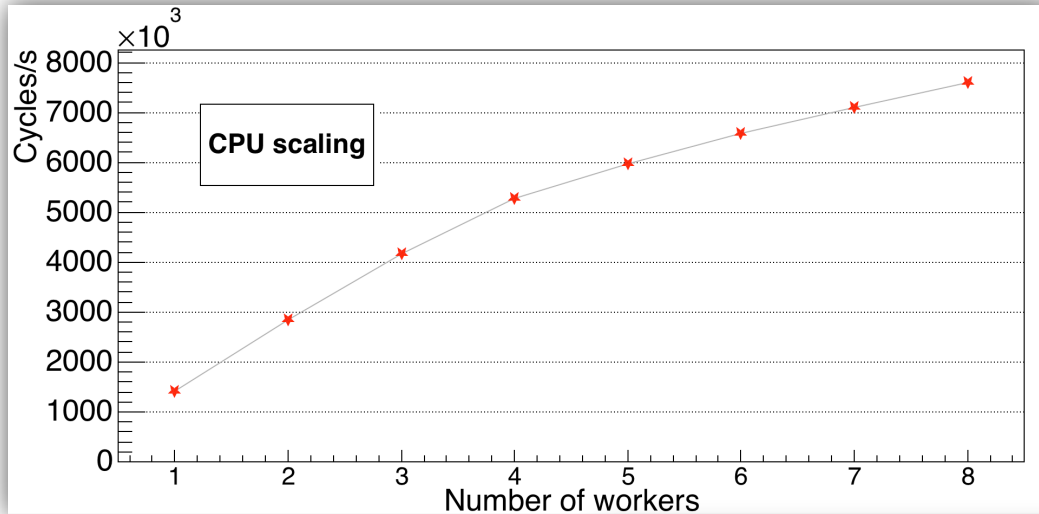
- Optimal number:  $\sim \text{Sqrt}(\# \text{ or workers})$

# TProofBench

S.Ryu, GG  
ACAT 2011

- CPU intensive
  - **Cycle**: generation of random numbers, fill histos
  - **Cycle/s** versus **# of workers**
- I/O intensive
  - **Cycle**: read entry from a TTree + some filtering
  - **Mbytes/s** versus **# of workers/node** or **# of workers**
  - **Cold read**: reset RAM cache before each run
- {Average, RMS} of 4 measurements / point
- Max and average rate
  - Average includes PROOF overhead
- Can use custom TSelector and dataset

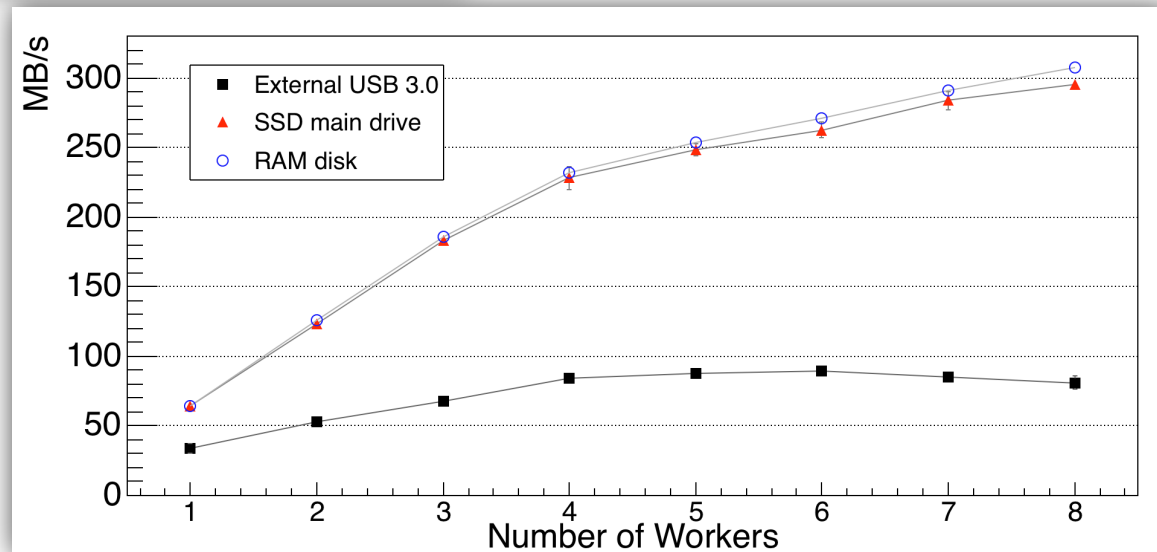
# Performances examples: PROOF-Lite



MacBookPro i7 2.3 GHz

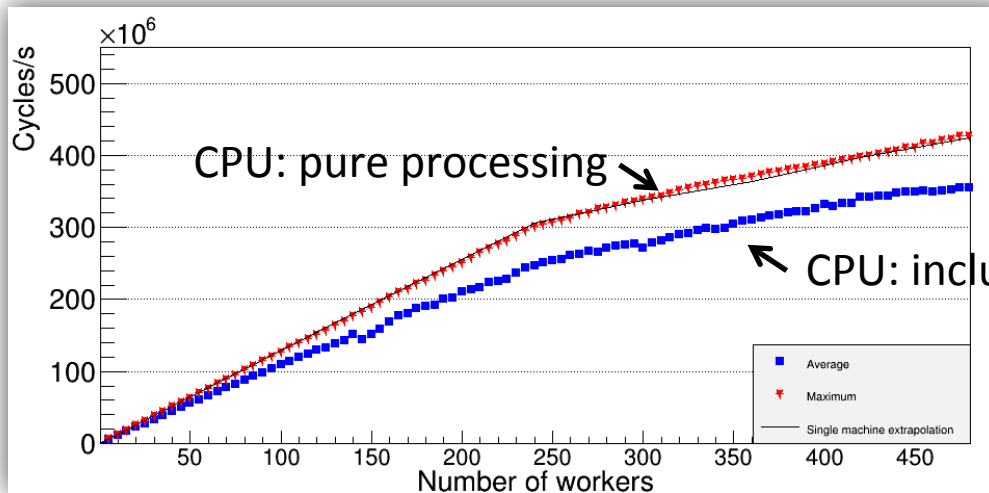
- Hypertreading kink visible

- RAM ~ SSD:  
300 MB/s  
CPU limited
- Ext USB  
~100 MB/s



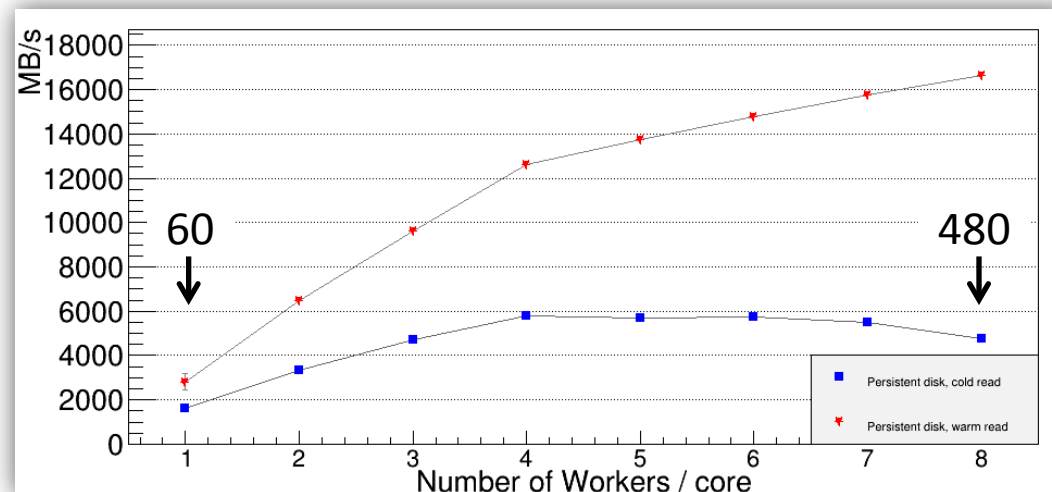


# Performances examples: cloud



Google Compute Engine  
480 cores, 60 nodes  
30 GB RAM / node

- **16 GB/s** from RAM (1.8 TB RAM total)
- **6 GB/s** from storage (cold reads)



GG, S. Panitkin  
CHEP 2013

April 1st, 2014

Multi-Processing in ROOT, G. Ganis

16



# How to use PROOF with IgProf



*TProof::Open("user@host", "igprof-pp");*

The screenshot shows the PROOF interface with a list of servers on the left and a flat profile on the right. The flat profile is a table with columns for total time, self time, and function name.

% total	Self	Function
25.09	2.92	*Random::Rand(int) [23]
9.07	1.13	__lee754_log [28]
9.21	1.07	cos [29]
7.97	0.93	sin [31]
6.43	0.75	*Track::Track(float) [21]
5.56	0.65	*Random::Rand(double, double) [24]
3.45	0.40	*Random::Ranxor(float&, float&) [22]
1.95	0.23	__lee754_sqrt [44]
1.80	0.21	*Storage::IsOnHeap(void*) [45]
1.49	0.17	*Math::Sqrt(double) [33]
1.44	0.17	_int_free [49]
1.23	0.14	malloc [43]
0.98	0.11	Event::AddTrack(float, float) [20]
0.93	0.11	_int_malloc [55]
0.87	0.10	*ProcessID::AssignID(TObject*) [34]
0.87	0.10	*Bits::SetBitNumber(unsigned int, bool) [53]
0.82	0.10	*TObject::TObject() [39]
0.77	0.09	*Track::GetPt() const [48]
0.72	0.08	*Math::Sin(double) [30]
0.72	0.08	sqrt [37]
0.72	0.08	system [64]
0.62	0.07	*TLockGuard::~TLockGuard() [78]
0.57	0.07	*TObjArray::At(int) const [95]
0.51	0.06	*Math::Log(double) [25]
0.46	0.05	log [26]
0.46	0.05	_init [59]
0.46	0.05	*TObject::ResetBit(unsigned int) [105]
0.41	0.05	*ProcessID::PutObjectWithID(TObject*, unsigned int) [51]
0.41	0.05	*ProcessID::GetObjectWithID(unsigned int) [57]
0.41	0.05	*TLockGuard::TLockGuard(TVirtualMutex*) [116]
0.41	0.05	__memset_sse2 [118]
0.36	0.04	*Bits::Bits(unsigned int) [38]
0.36	0.04	*TObjArray::AddAtAndExpand(TObject*, int) [58]
0.36	0.04	*Math::Max(int, int) [122]
0.36	0.04	*TObject::SetUniqueID(unsigned int) [123]
0.36	0.04	*TCollection::GetSize() const [125]
0.31	0.04	Event::Build(int, int, float) [19]
0.31	0.04	*TClonesArray::Clear(char const*) [36]
0.31	0.04	*TObjArray::AddAtAndExpand(TObject*, int) [68]
0.31	0.04	*TClonesArray::operator[](int) [94]
0.31	0.04	*TObjArray::UncheckedAt(int) const [140]
0.31	0.04	*TObject::GetUniqueID() const [141]
0.31	0.04	free [142]
0.31	0.04	memset [144]
0.26	0.03	*Math::operator=(TObject*) [32]
0.26	0.03	*Track::Clear(char const*) [41]
0.26	0.03	*TSeqCollection::Changed() [152]
0.26	0.03	*TObject::ResetBit(unsigned int) const [153]
0.26	0.03	__slow [155]
0.21	0.02	Event::Clear(char const*) [35]
0.21	0.02	*Bits::Clear(char const*) [50]
0.21	0.02	vsprintf [124]
0.21	0.02	*Math::Abs(double) [165]

- When processing finishes, special IgProf logfiles appear
- Same technique used with Valgrind
- **IgProf is not needed on the client!**

# Current plans

- Improving merging
- Improving usability

# Merging during run

- Exploit Master-Worker interactivity
- Master collects results from workers during run and creates directly the final objects
- For trees or alike objects (size  $\sim N$ )
  - Integrate Philippe's multi-producer/consumer technology
  - PROOF is an ideal application case
- For histograms or alike objects (size  $\sim$  fixed)
  - Stream buffers of entries to master
    - Buffering already exists for automatic bin range mode
  - May need new interface for general application

# Improving usability

- Package management
  - Versioning, distribution, default makefiles
- Transparency
  - Re-usage of TTree code, e.g. for Drawing
  - Automatic switch to PROOF-Lite on desktops
- Simplified interface for user code
  - Ideally usable in a multi-threaded environment
- Optimizations for math calculations
  - E.g. reuse same setup for multiple calculations

# Use of fork in PROOF-Lite

- Improve usability (environment setting) and resource utilization
- Idea is to try forking the client ROOT session
  - Need to evaluate issues related to components loaded but not need on workers (e.g. graphics ... )

# Summary

- Multi-processing in ROOT means PROOF
- Consolidated technology to efficiently operate
  - Large facilities (clouds)
    - PROOF-As-A-Service
  - Multi-cores
- Continue effort to improve usability and performances