PROOF on multi-core machines

G. GANIS
CERN / PH-SFT
for the ROOT team

Workshop on Parallelization and MultiCore technologies for LHC, CERN, April 2008

Outline

- Introduction
- Optimizing for local machines: PROOF-Lite
- Some performance results
- Future

ROOT and threads

- Multi-threads is the natural way to exploit cores
- Support for threads is available since long time in ROOT but many components cannot be used efficiently with multiple threads
  - Current CINT
  - Containers, Files
- Thread-safeness insured via global mutexes which introduce serialization at many places

- Chain processing, looping run generic user code for which you cannot assume thread-safeness
- The situation should improve in the future with the new CINT

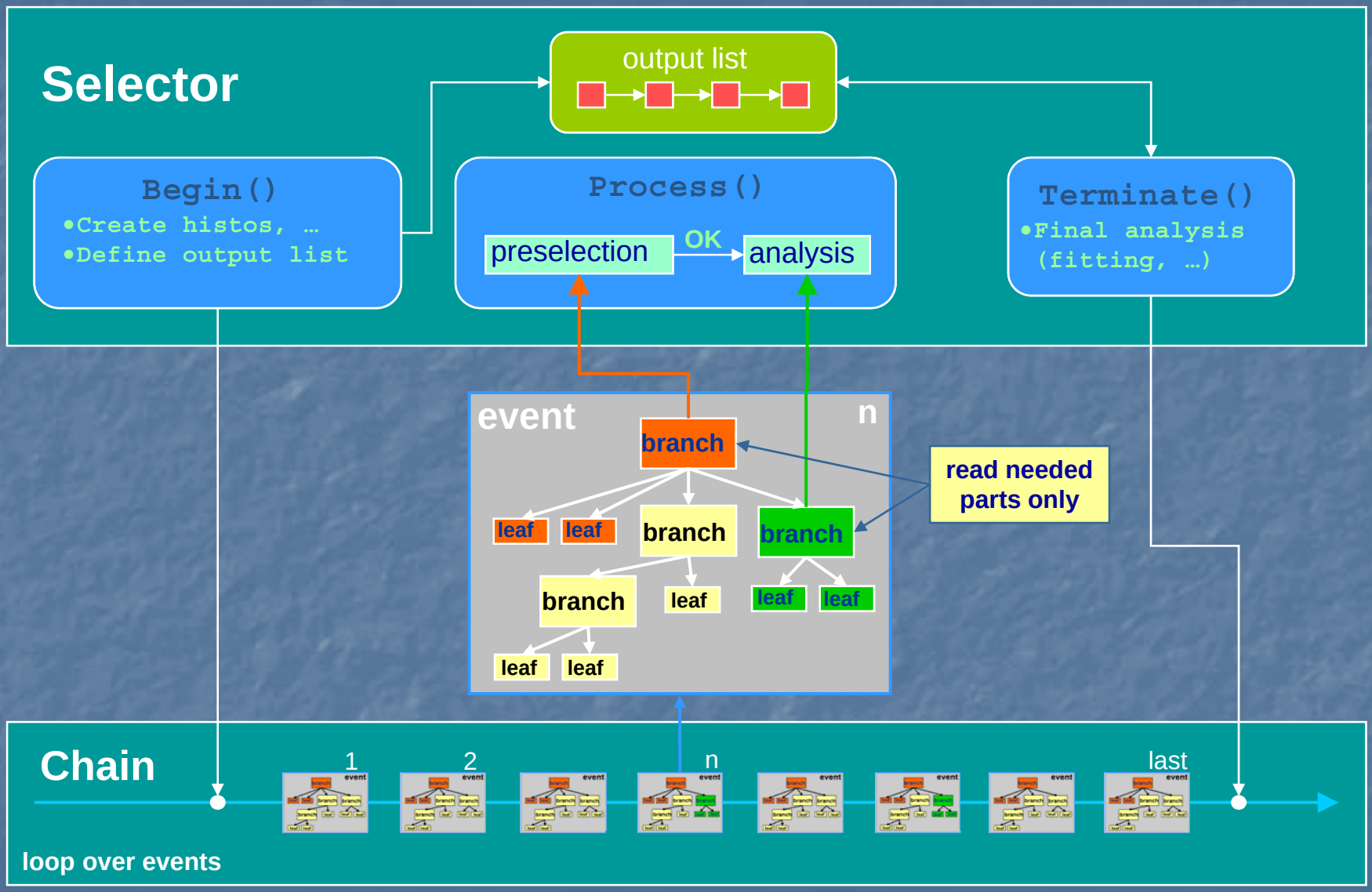
Using cores to improve IO

- When reading data a large fraction of time is spent in decompressing
- This a case where additional core(s) may help and it is a dedicated task under control of ROOT which could already be done now in a separated thread
- Planned for (hopefully) not far future

ROOT way to exploit multiple resources

- PROOF is the ROOT approach to exploit multiple resources to reduce the time needed to solve problems which can be formulated as at a set of independent tasks, i.e. embarrassing or ideally parallel
  - e.g. HEP events in TTree's
- Job splitting to address ideal parallelism is an old concept, but
  - PROOF inter-connects many ROOT sessions in such a way that they are seen as a extension of the normal ROOT shell, with minimal syntax differencies.
  - Splitting is dynamic allowing to optimize loads

The ROOT data model: Trees & Selectors



Typical PROOF session

```
// Get a TFileCollection describing your dataset
root[0] TFileCollection *fc = <MyCatalog>->Get("mydata");
// Create a TChain
root[1] TChain *c = new TChain;
root[2] c->AddFileInfoList(fc->GetList());
// Run over "mydata" you analysis selector mysel.C
root[3] c->Process("mysel.C+")
```

Local processing

```
// Open the PROOF session
root[4] TProof *p = TProof::Open("master")
// Process on PROOF
root[5] c->SetProof()
root[6] c->Process("mysel.C+")
```

PROOF processing

PROOF Processing by name

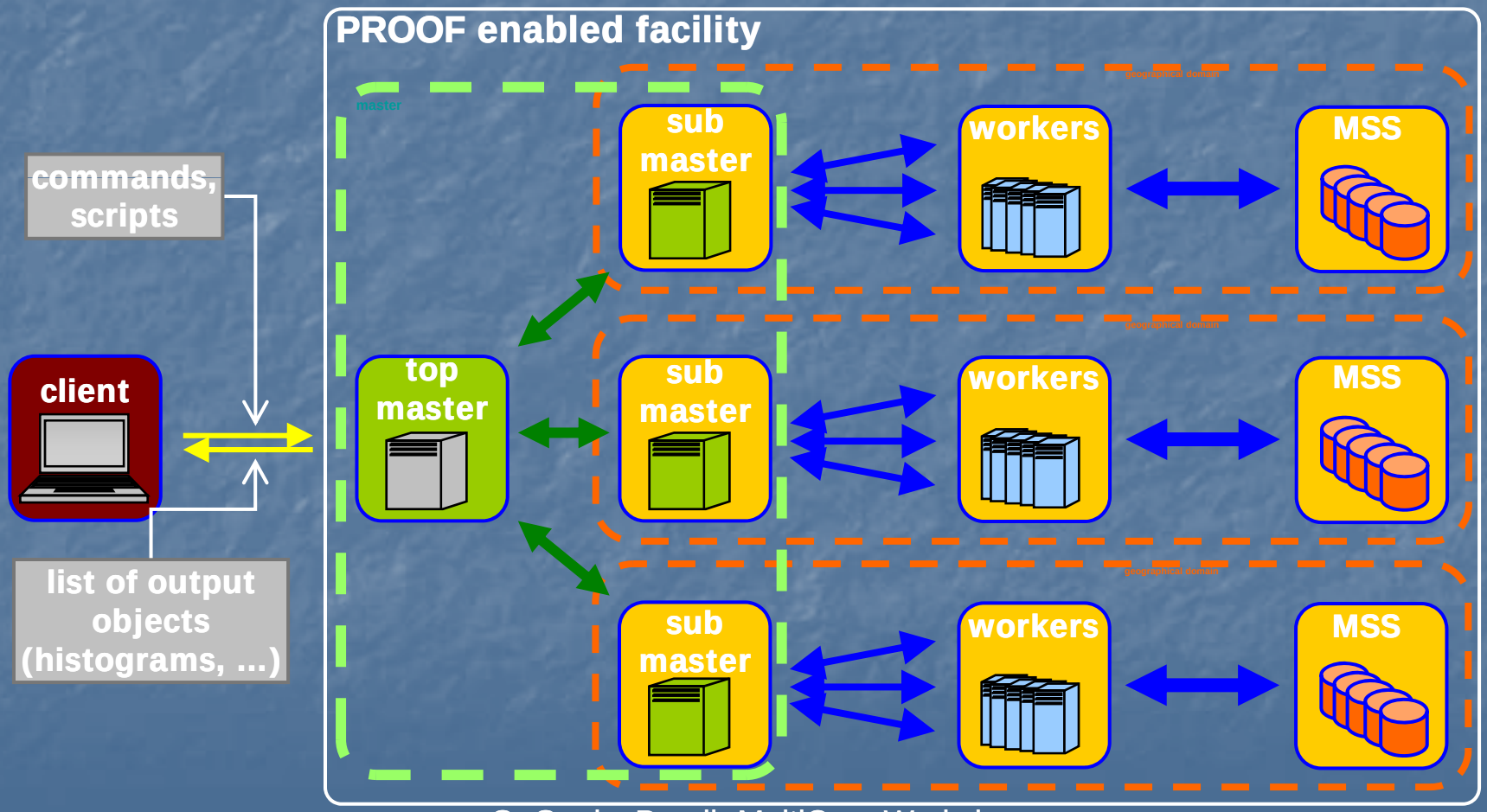
```
// Open the PROOF session
root[0] TProof *p = TProof::Open("master")
// Get a TFileCollection describing your dataset
root[1] TFileCollection *fc = <MyCatalog>->Get("mydata");
// Register your dataset (only once)
root[2] p->RegisterDataSet("mydata", fc);
// Run over "mydata" you analysis selector mysel.C
root[3] p->Process("mydata", "mysel.C+")
```

PROOF

- PROOF has been developed having in mind the case of T2/T3 analysis facilities, clusters O(100) nodes
- Its flexible multi-tier architecture allows to adapt to very different situations, and to move in size in both directions
  - Expand to federate clusters, eventually to the GRID
    - See A. Manarof at PROOF07
  - Shrink to few machines
- Multi-Core is at the extreme: one machine, lot of CPU power …
- How does vanilla PROOF on multi-cores ?

PROOF in a slide

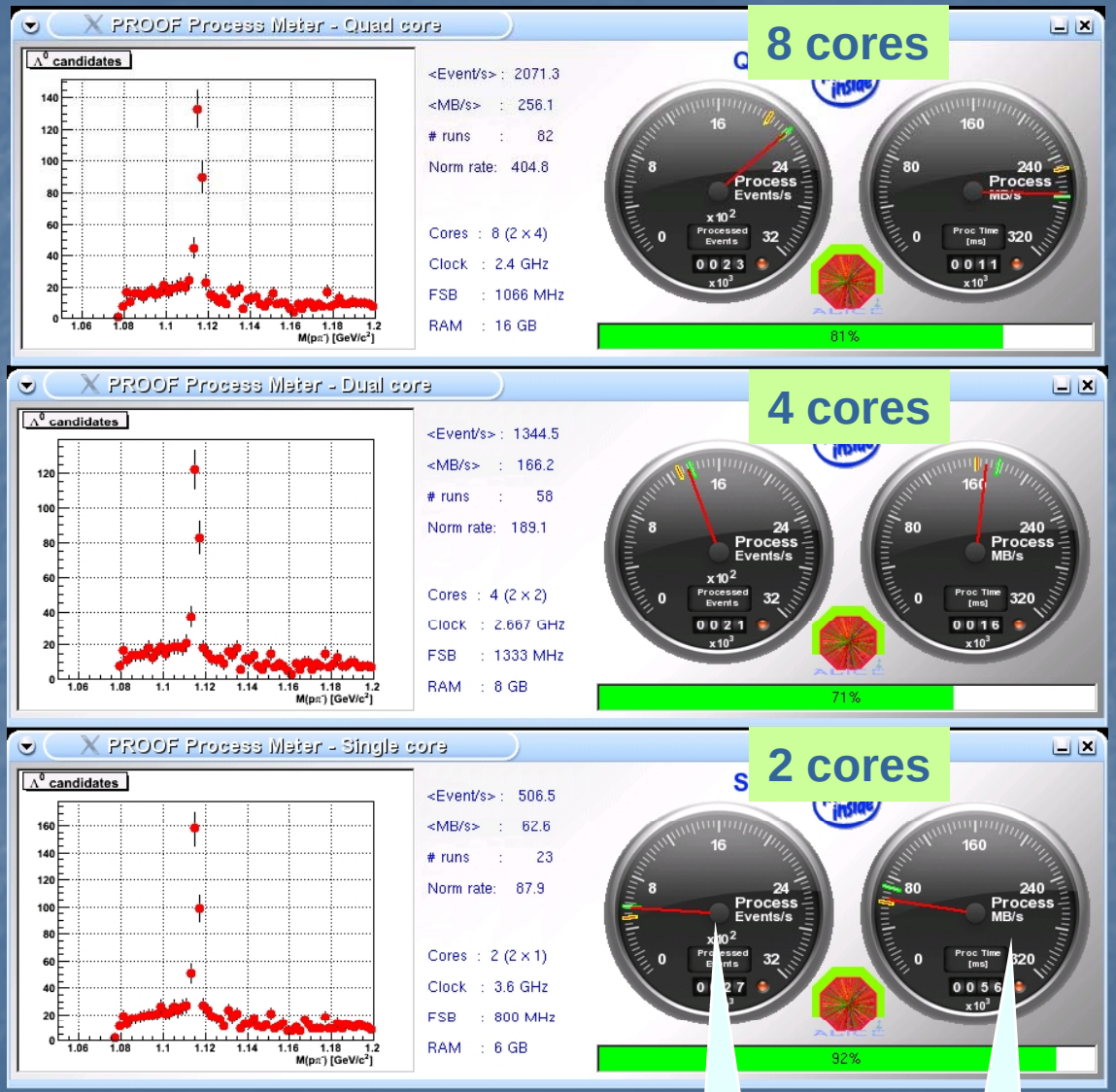
PROOF: Dynamic approach to end-user HEP analysis on distributed systems exploiting the intrinsic parallelism of HEP data



PROOF exploiting multi-cores

- Demo at Intel Quad launch (Nov 2006)
- Analysis: a search for Λ^0's in ALICE
- Data: 4 GB simulated (fit in memory)

Additional computing power fully exploited: quite promising!



However ...

- ... the analysis was effectively CPU-bound with quite small outputs (a few 1D histos).
- What if we are in the opposite extreme (IO bound, large outputs)?
- PROOF forum report (Feb 2007):

  "I have a dual-core 64 bit Intel machine, running SLC 4.3. ... I setup local PROOF system and made a simple tree that I have filled and analyzed. This is faster on one processor without PROOF than on two with PROOF ..."
- What was the problem:
  - one disk, no special hardware
  - Very light events: extremely I/O bound analysis
  - Quite large output: large overhead from merging and object transfer

Lesson

Rather a trivial one

- Depending on what you do, increasing the available CPU is not the end of the story: the bottle neck can be elsewhere
- An improved IO system may be needed
  - e.g. multiple disks, possibly in RAIDs

PROOF optimizations

- While standard, 3-tier, PROOF seems basically OK for certain tasks, there is space of improvements for the cases of large outputs
- Target: minimize number of creations of output objects
- Output object history trace in standard PROOF:
  - Each worker creates an output object and streams it out to the master socket
  - The master re-creates it by streaming-in from the socket
  - The master merges it to the final version object
  - The master streams the final object out to the client socket
  - The client re-creates it by streaming-in from the socket
- Is all this needed locally?

PROOF optimizations (2)

- Not really:
  - The master is not needed locally:
    - the client can have master functionality
  - Communication between processes can be improved
    - Using UNIX sockets
    - Producing the objects in a shared area from to avoid streaming-in/-out from sockets
      - e.g. a file or a shared memory

PROOF Lite

- PROOF Lite is a realization of PROOF in 2 tiers
  - The client starts and controls directly the workers
  - Communication goes via UNIX sockets
- No need of daemons:
  - workers are started via a call to 'system' and call back the client to establish the connection
- Starts N_{CPU} workers by default
- Currently available from SVN 'branches/dev/prooflite'
  - Soon in the trunk

PROOF Lite (2)

Additional reasons for PROOF-Lite

- Can ported on Windows
  - There is no plan to port current daemons to Windows
  - Needs a substitute for UNIX sockets
    - Use TCP initially
- Can be easily used to test PROOF code locally before submitting to a standard cluster
  - Some problems with users' code are difficult to debug directly on the cluster

Merging from files

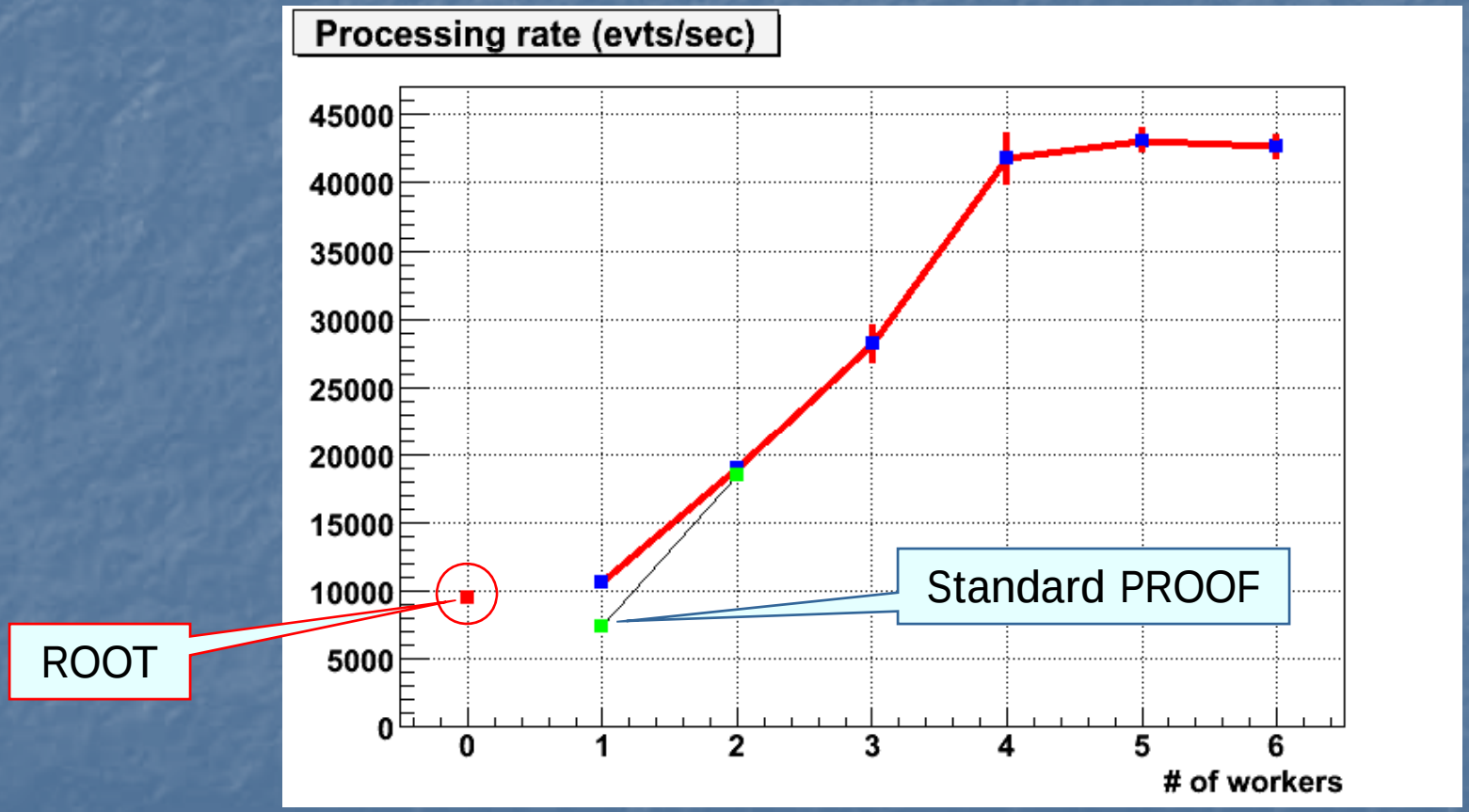
- Recent addition to PROOF
  - Each worker writes the output object to a file
  - The client-master gets just the location of the file and merges them using optimized merging
- Quite significant improvements for the case of large outputs

Some results: test setup

- Test machine
  - Intel Xeon (2x2) x 2.66 Ghz
  - 8 GB RAM
- Analysis:
  - Event generation: simple events ($ROOTSYS/test/Event.h)
    - small output (TH1 histograms)
    - large output (TTree ~350 MB compressed; merging via files)
  - Process TTree from files (~80 MB/file)
    - Full dataset: 22GB
    - Sub-datasets: {2, 4, 6, 7, 8, 9, 10, 12} GB
      - Read from the same disk or from separated disks
- Results from average of {4,10} runs in the same conditions
- Non-PROOF results obtained using the same machinery

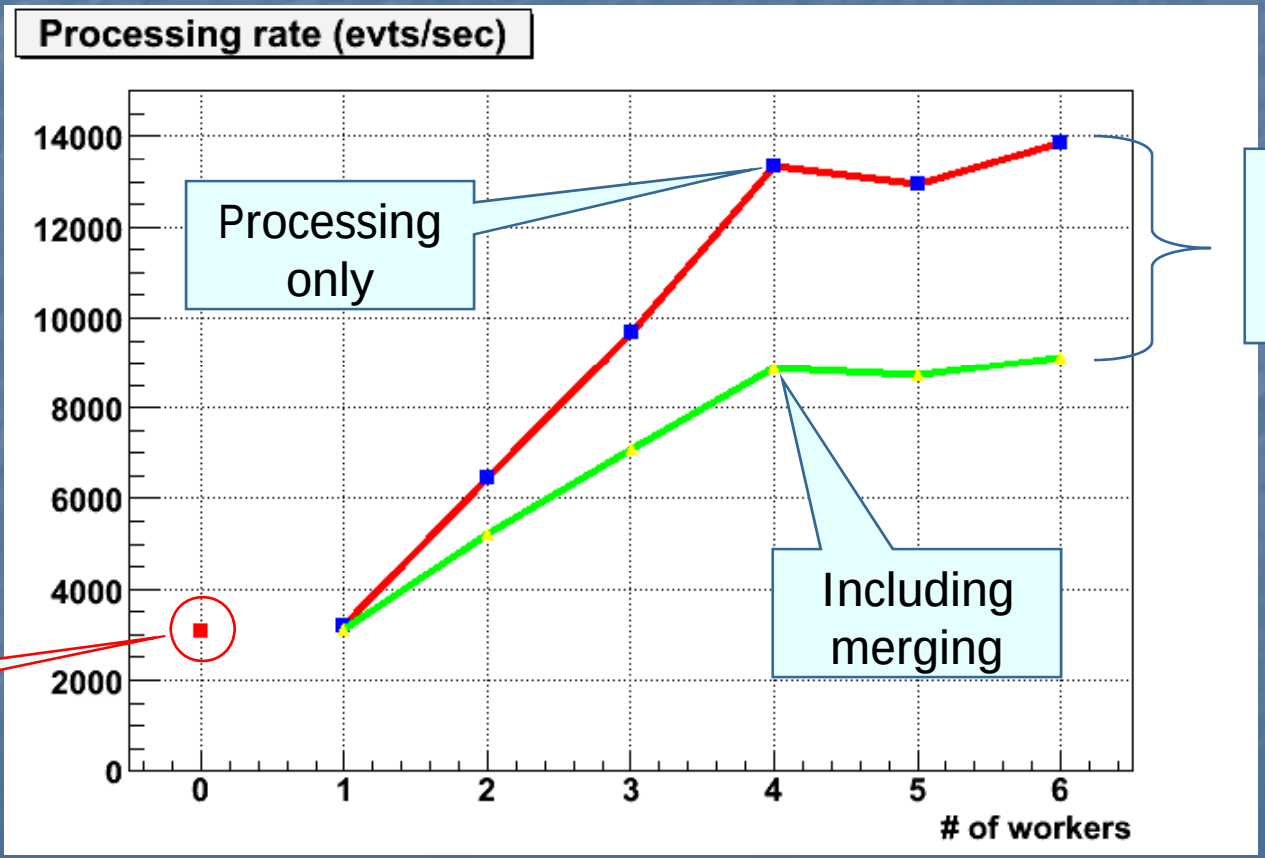
Simple scaling

- Simple event generation and 1D histogram filling



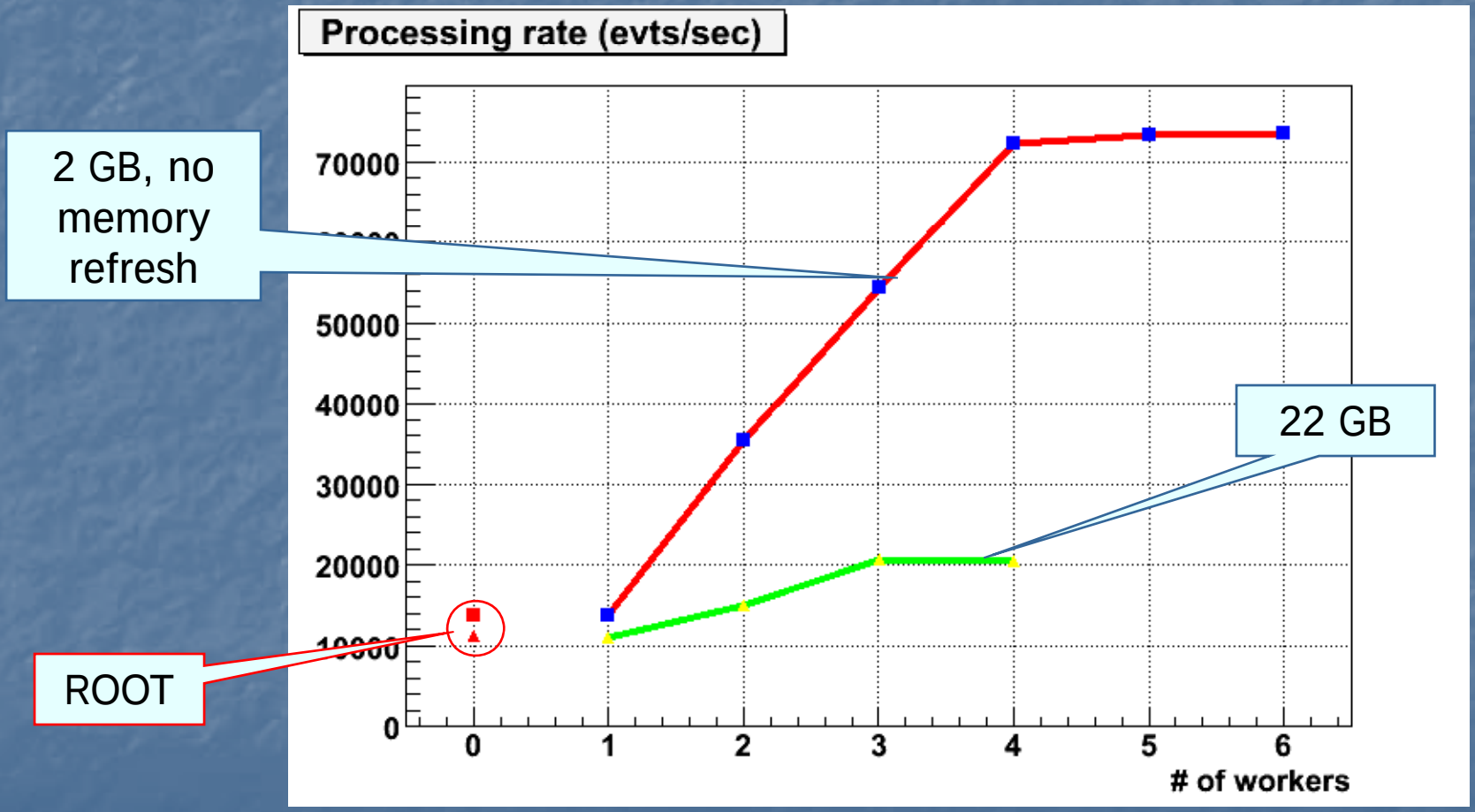
Simple scaling ~large output

- Simple event generation, TTree filling, merging via file (output ~ 350MB after compression)



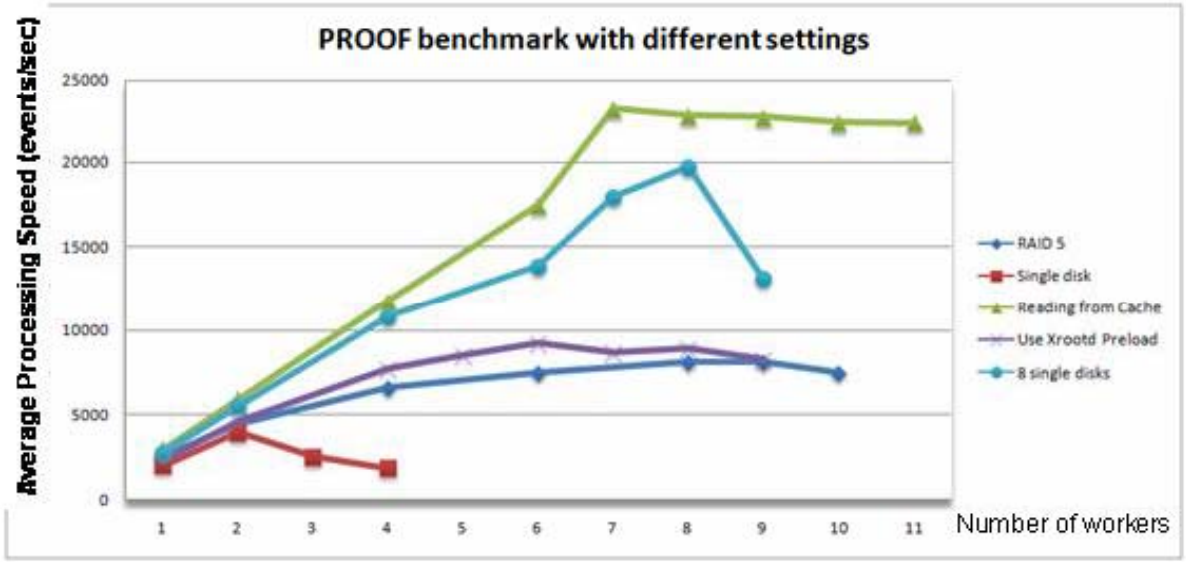
Scaling processing a tree

- Data sets 2 GB (fits in memory), 22 GB



Hardware impact on scaling

An overview of the performance rate



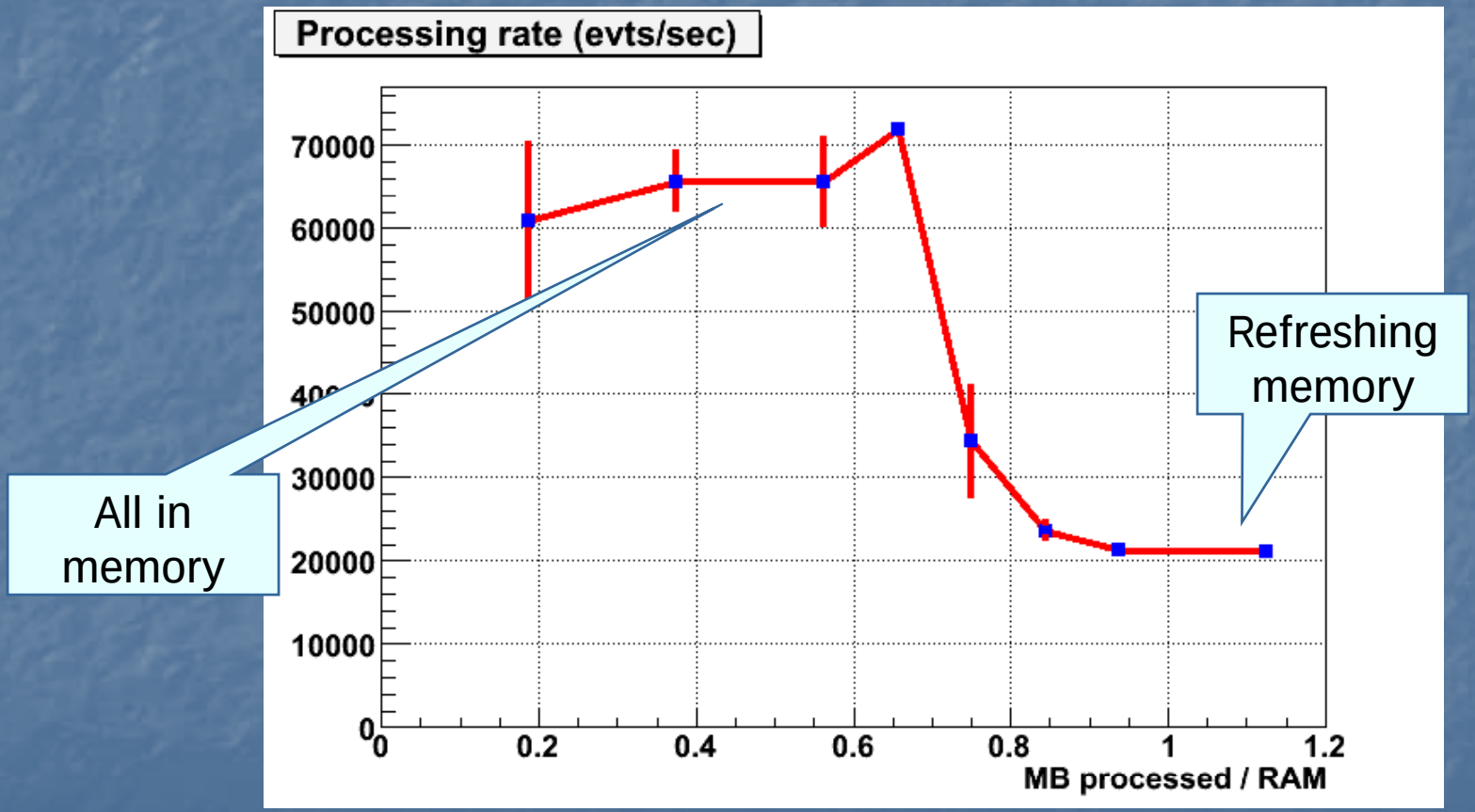
- For I/O bound job, disk throughput will be the bottleneck for most of the hardware configurations; reading data from memory will provide the best scalability and performance.
- Trade-off between the data protection and the working performance has to be made.
- Using Xrootd Preload function will help to increase the analysis speed.
- 2 cores vs. 1 disk seems to be a reasonable hardware ratio without using Raid technology.

16 Neng Xu, University of Wisconsin-Madison 11/29/2007

Courtesy of Neng Xu, Wisconsin PROOF07 Nov 2007

Performance vs fraction of RAM

- Reading datasets of increasing size

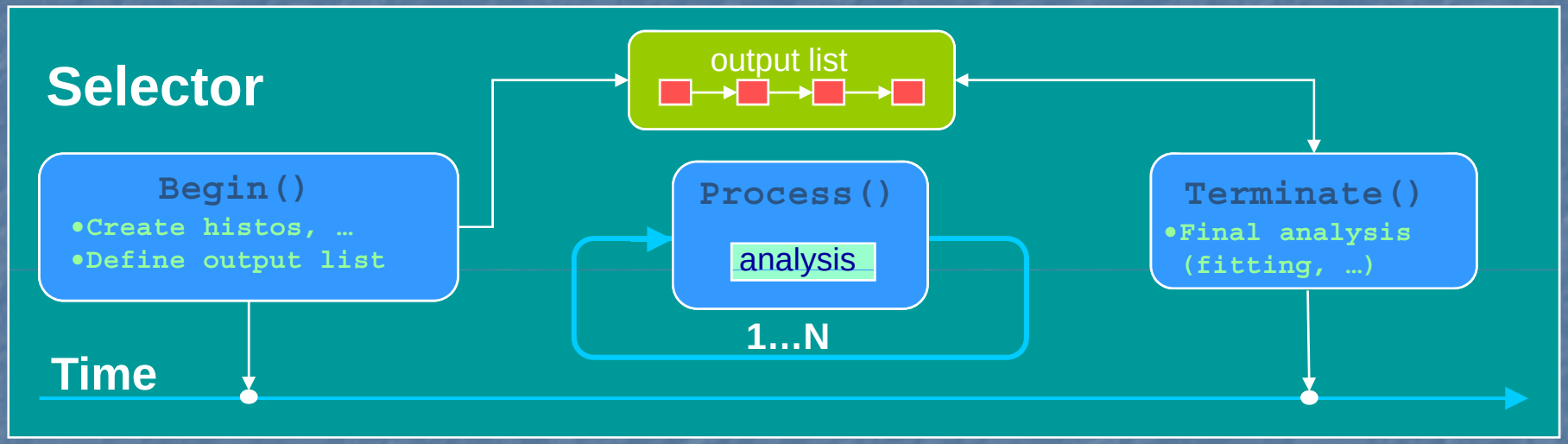


What next?

- PROOF-Lite
  - Further optimizations for merging objects
  - Porting on Windows
- Related developments
  - Improve interface for non-TTree based analysis, currently based directly on TSelector
    - TSelector template where to plug macros
    - Dedicated macros to instrument the code to transparently run loops on PROOF
- Continue testing different scenario to find optimal configurations

Generic, non-data-driven analysis

Implement algorithm in a TSelector



New TProof::Process(const char *selector, Long64_t times)

```
// Open the PROOF session
root[0] TProof *p = TProof::Open("master")
// Run 1000 times the analysis defined in the
// MonteCarlo.C TSelector
root[1] p->Process("MonteCarlo.C+", 1000)
```

Summary

- PROOF is currently the ROOT way to exploit multi-cores
- Performance:
  - CPU-bound: already quite good
  - IO-bound: critically depends on I/O performance as for all systems
- Handling of large outputs significantly improved by file-based merging
- Version optimized for multi-core machine is available for test