PROOF Best Practices

Or understanding the bottlenecks in your system

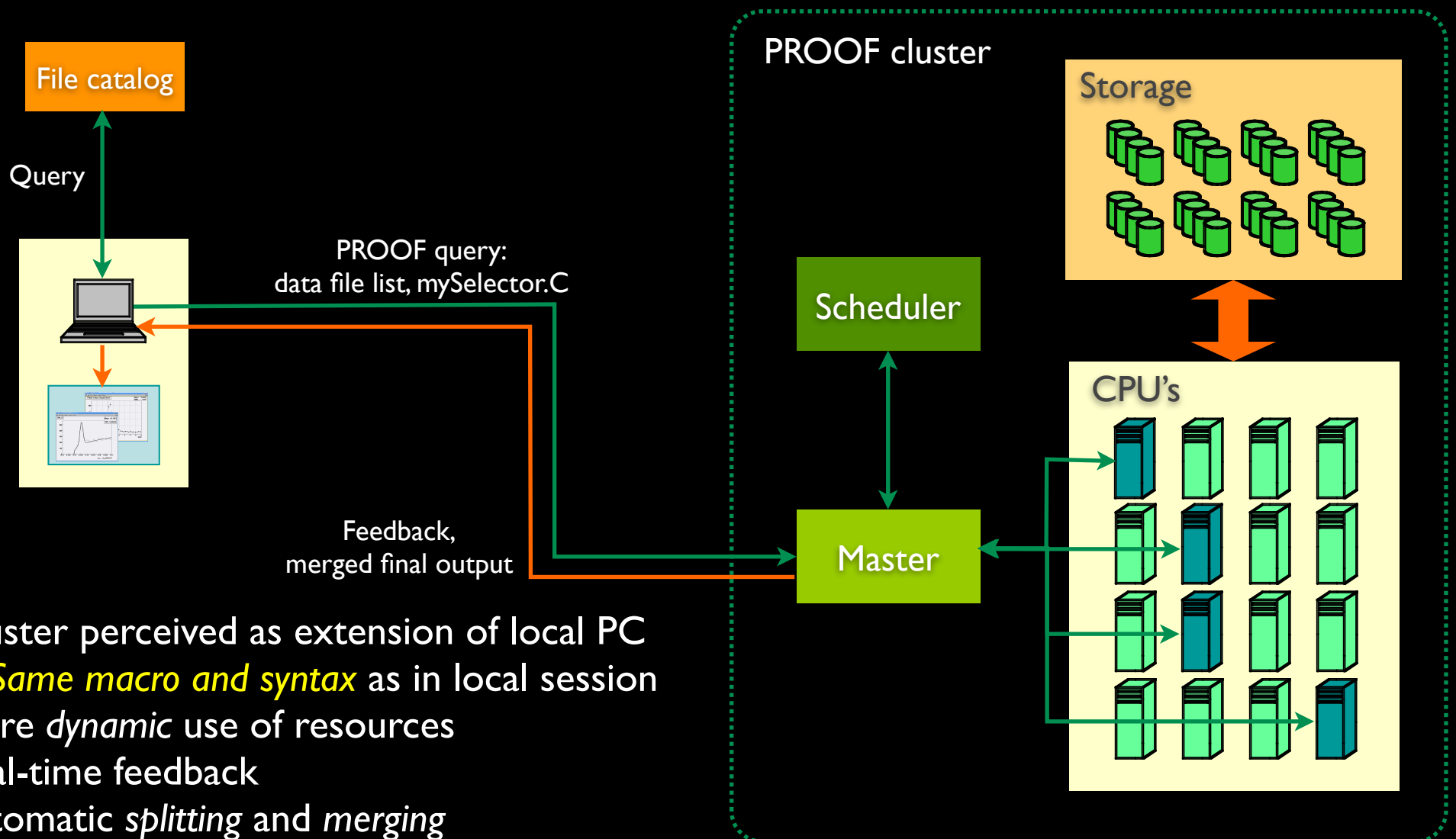
Fons Rademakers, Gerardo Ganis
CERN

ACAT 2010, 26-Feb, 2010, Jaipur, India.

Outline

- What is PROOF
- Typical bottlenecks
- The challenge to efficiently use many-core machines

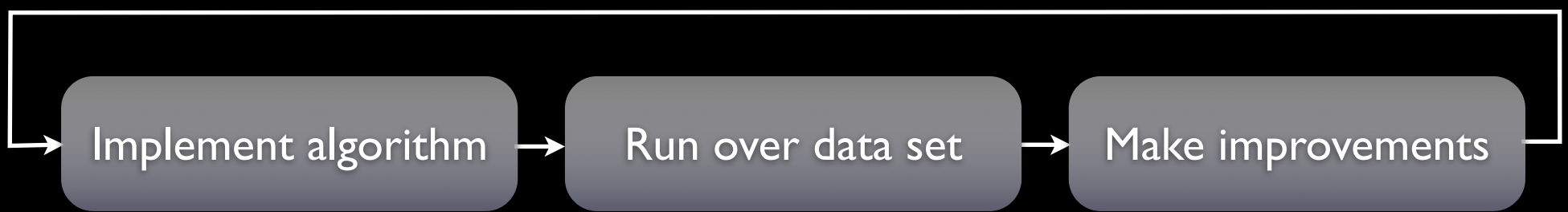
The PROOF Approach



- Cluster perceived as extension of local PC
  - *Same macro and syntax* as in local session
- More *dynamic* use of resources
- Real-time feedback
- Automatic *splitting* and *merging*

HEP Data Analysis

- Typical HEP analysis needs a continuous algorithm refinement cycle



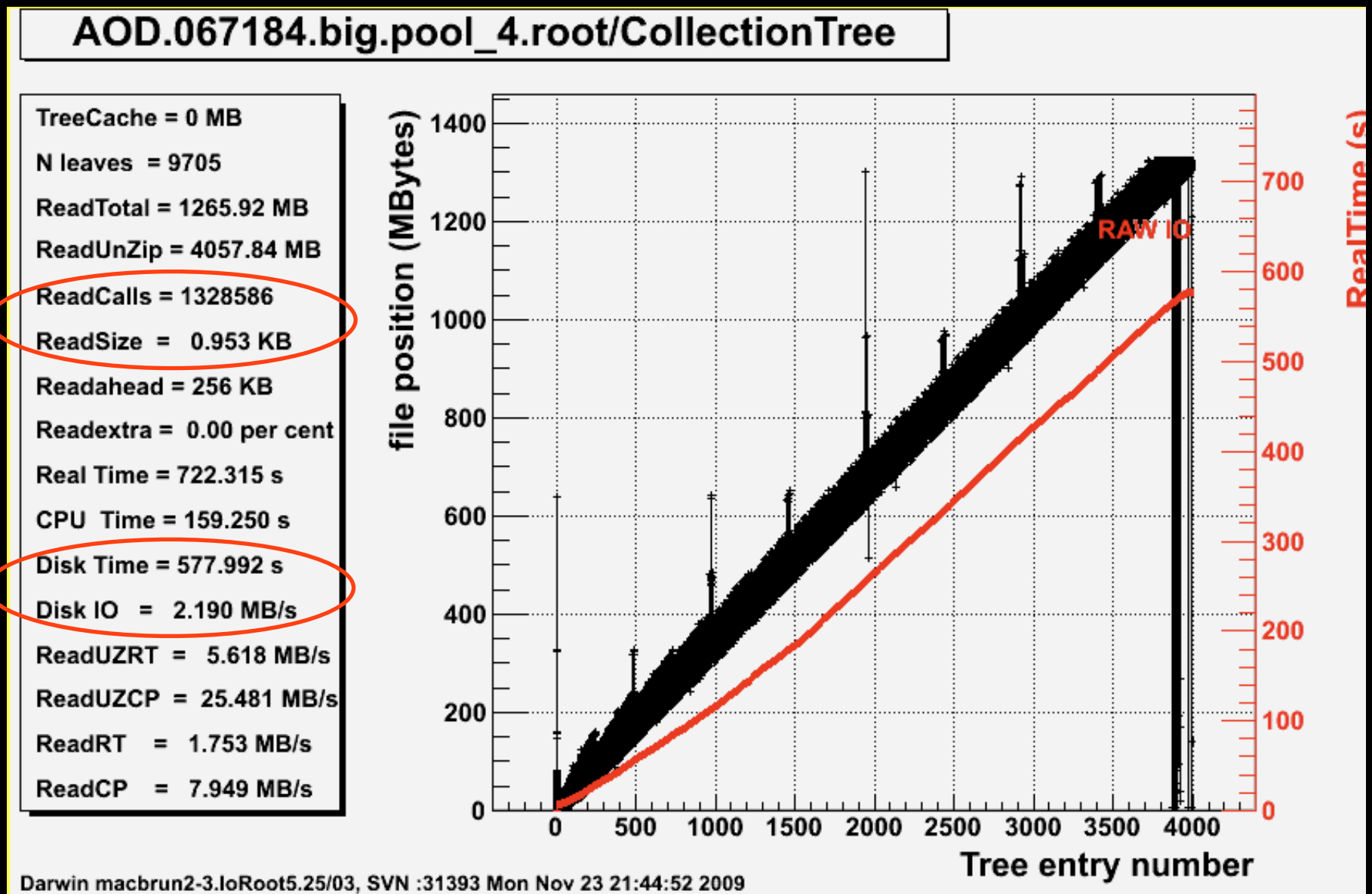
- Ranging from I/O bound to CPU bound
- The more disks/network the higher the I/O rate
- The more RAM the more can be cached
- The more CPUs the faster the processing

Typical Bottlenecks

- Non optimal file layout
- Merging large outputs
- Too few disks
- Too slow network
- Too little RAM
- Too few CPU's

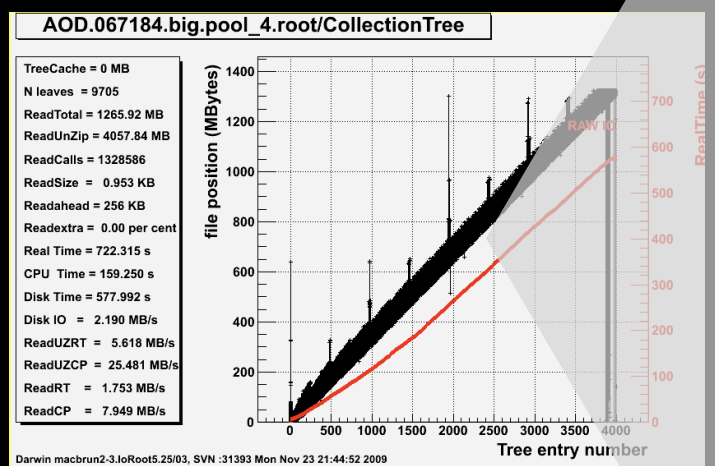
You want to minimize real-time vs CPU-time.
Real time is governed by the CPU speed.
CPU has to become the final bottleneck.

Non Optimal File Layout

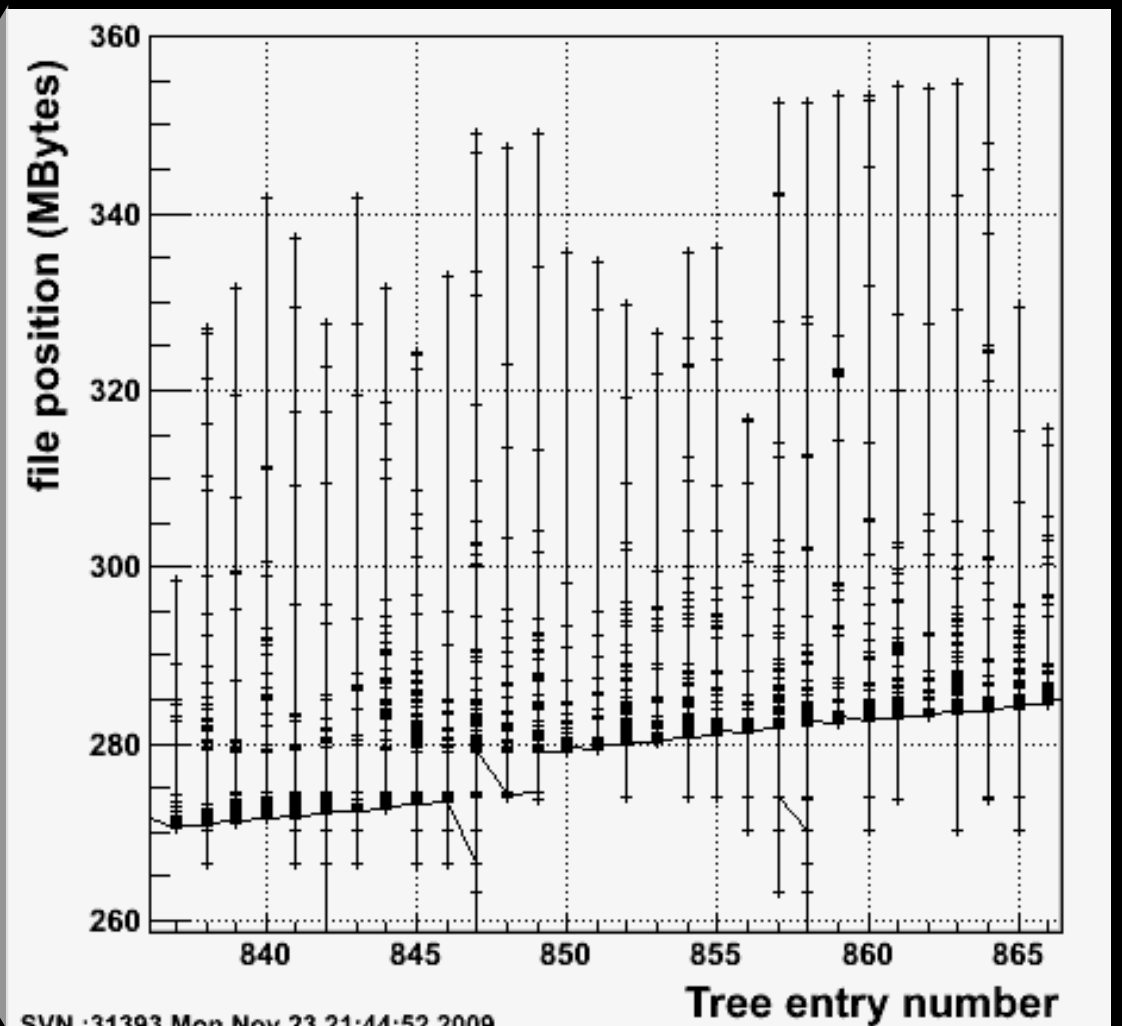


Performance measurements made using the new TTreePerfStats class

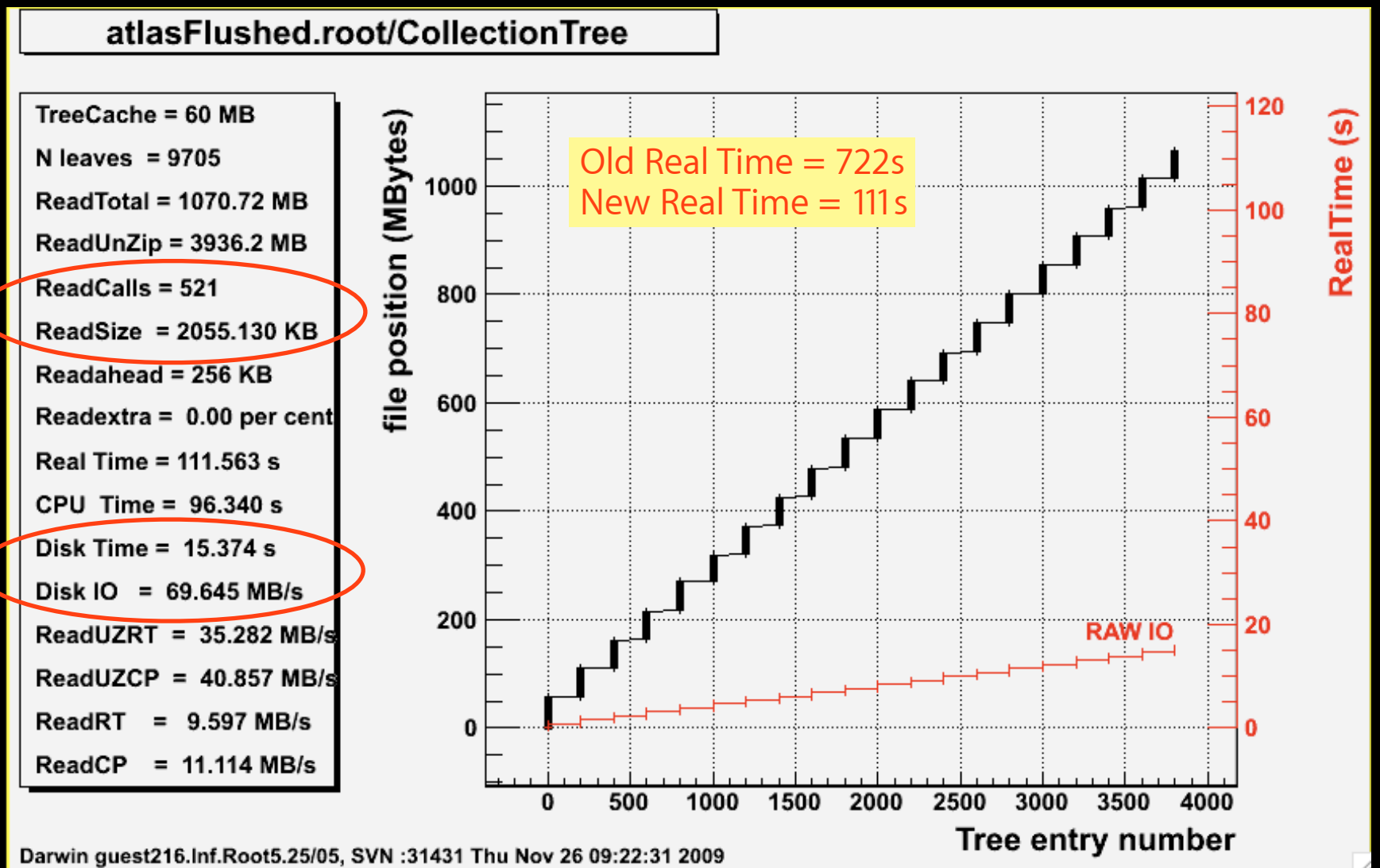
Overlapping Reads



100 MB



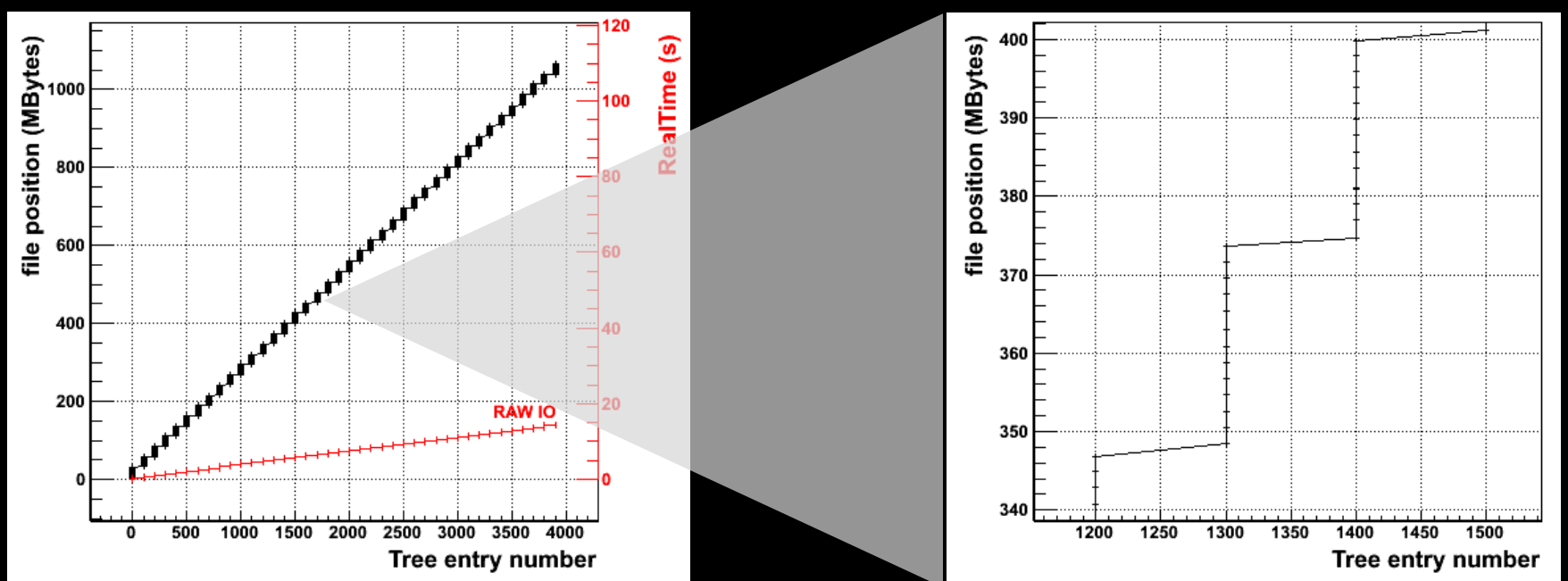
Optimized File Layout



OptimizeBaskets and FlushBaskets

- Solutions, enabled by default:
  - Tweak basket size
  - Flush baskets at regular intervals

Available in v5.26.00



OptimizeBaskets

- The TTree::OptimizeBaskets() method is a new function that will optimize the buffer sizes taking into account the population in each branch
- Tunes the branch buffer size
- Without this tuning branches containing the same event are scattered in the file
- You can call this method on an existing read-only Tree to see the diagnostics

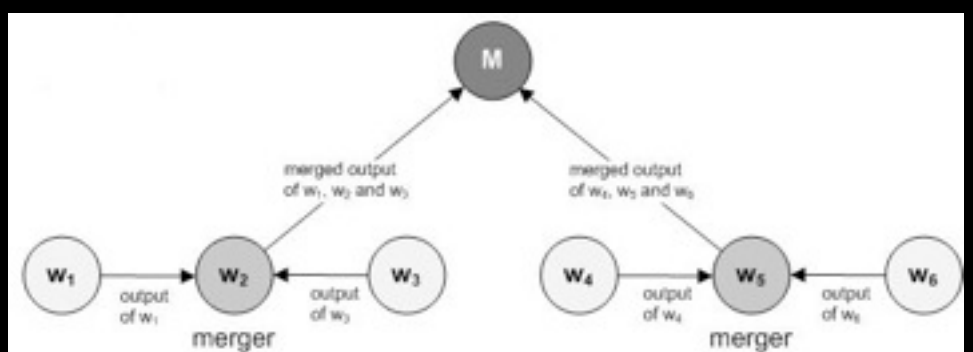
FlushBaskets

- The TTree::FlushBaskets() method was introduced in 5.22 but called only once at the end of the filling process
- In version 5.26 this method is called automatically when a reasonable amount of data (default 30MB) has been written to the file
- The first time that TTree::FlushBaskets() is called, we also call TTree::OptimizeBaskets()
- The frequency to call TTree::FlushBaskets() can be changed by TTree::SetAutoFlush()
- Thanks to TTree::FlushBaskets() there are no backward seeks anymore (for files written with 5.26).

Merging Large Outputs

- Large or non-optimized outputs may have dramatic effects
  - Memory explosion
  - Destroy the parallelization gains during merging
    - Many 3D histograms or 10000's of 1D histograms
- PROOF solution
  - Save outputs on files on worker nodes and
    - Automatically run a merging application (TFileMerger or your own)
    - Automatically create a dataset for further processing
  - Parallel merge: fastest workers act as sub-mergers

Theoretical speedup: $\sqrt{N_{wrk}}$
Available from ROOT 5.26



Too Few Disks

- In many analyses (e.g. skimming) the queries are I/O bound
- Need to optimize for I/O throughput
- ROOT compressed file I/O typically ~20MB/s
- Employ the fastest possible disk technology, depending on the number of cores that need to be fed:
  - Many spindles
  - SAS 10k RPMS disks (90 MB/s)
  - SSD (120MB/s)
  - ioDrive (700 MB/s)
  - RAID 0 (most cards max out at 300MB/s, need multiple cards)

Too Slow Network

- Network becomes critical in case of remote file systems
- The latest 24-32 core servers need at least 10GB Eth (700 MB/s)
- For the file servers containing the remote file system, the same observations are valid as in the previous slide

Too Little RAM

- Fastest I/O resource
- RAM is important to cache file data
  - File system buffer cache
  - Remote file system cache
- Extremely important for repeat queries that could be run completely from RAM

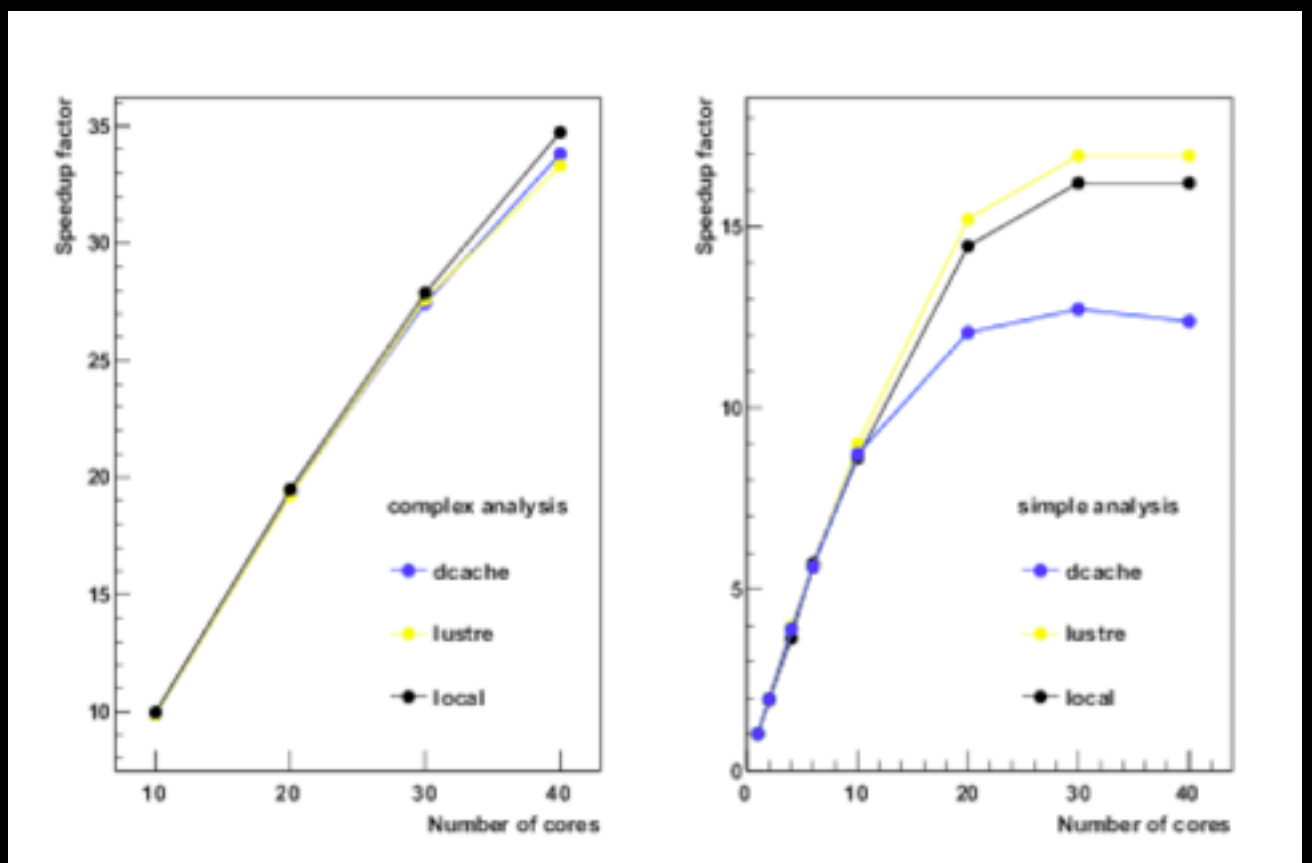
Too Few CPU's

- As PROOF takes advantage of the trivial parallelism in HEP data analysis it scales very well up to a large number of CPU's
  - Architecture minimizes the sequential part, resulting in excellent scalability up to 100's of CPU's
- To reduce real-time get more CPU's

Comparing Different Storage Solutions

- A PROOF cluster of 10 Opteron nodes with 4 cores and 8GB RAM
  - Local disks: data is stored on each local node
  - dCache: access via client/server connections RAID6, 10GB switch
  - Lustre: all nodes can access the data without a dedicated server

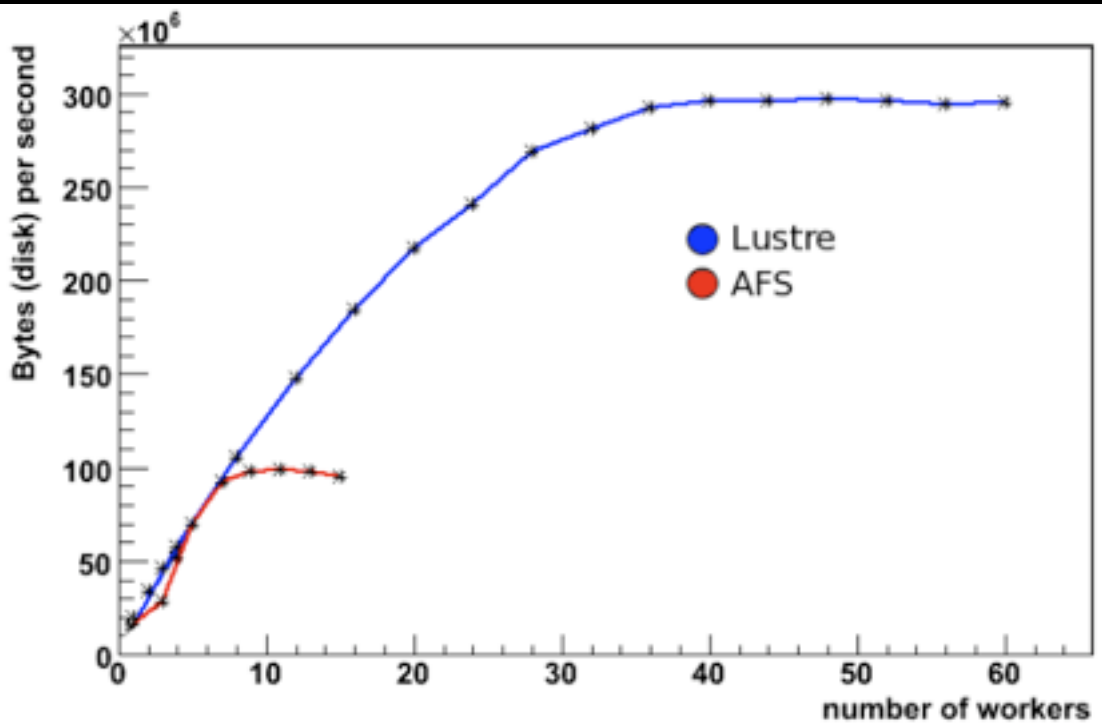
- Input data files:
  1.6M 4kB events



Results courtesy: P. Calfayan, S. Panitkin

Comparing Different Storage Solutions

- Comparing Lustre vs AFS



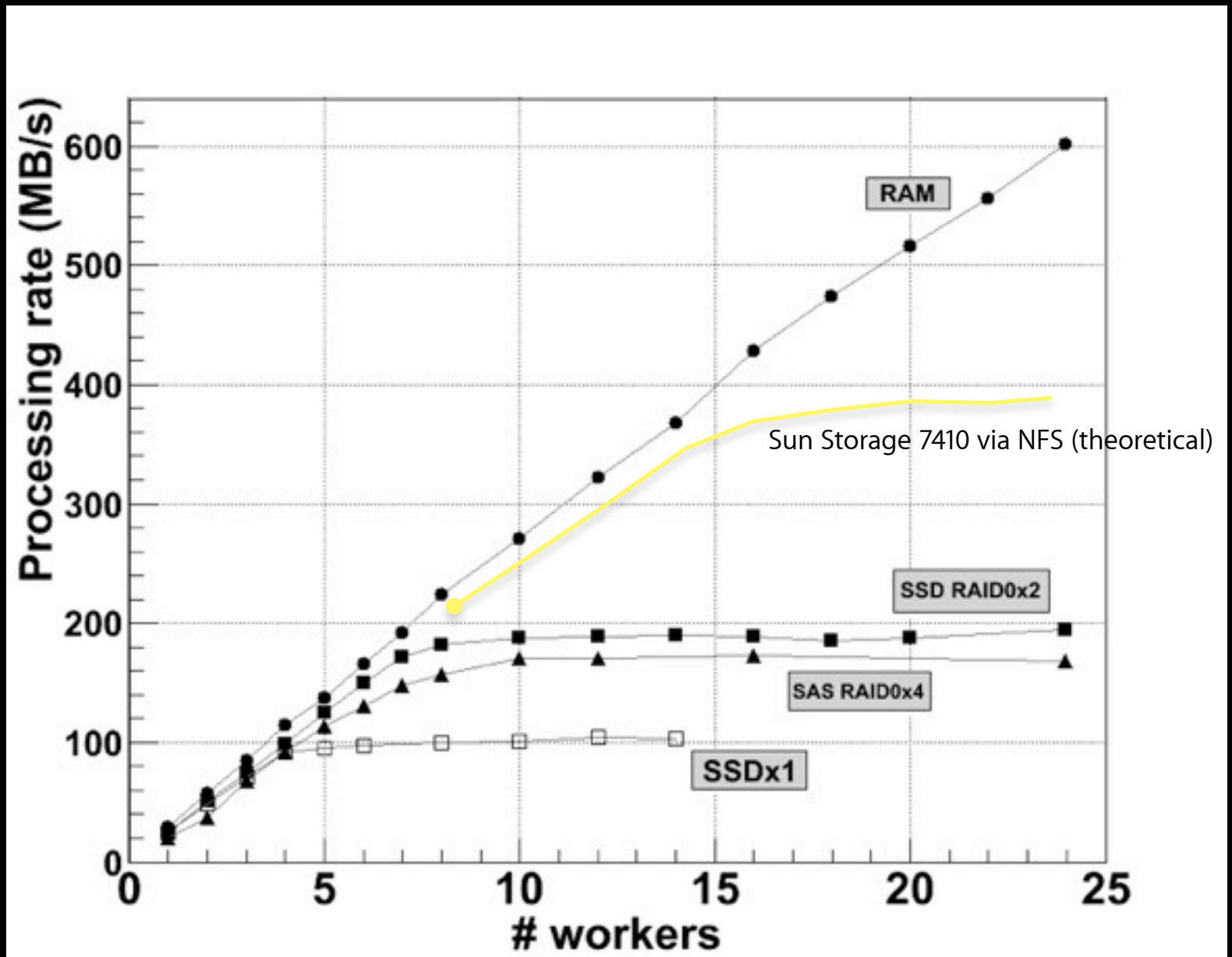
As expected, Lustre via InfiniBand clearly outperforms AFS. Only one file server was used in this test. Thus, the input rate should increase further with additional servers.

Results courtesy: NAF

Running on Many-Core Machines

- For the following tests we run PROOF-Lite on a 24 core HP DL580 with 48 GB RAM
  - PROOF-Lite is PROOF optimized for single many-core machines
  - Zero configuration setup (no config files and no daemons)
  - Workers are processes and not threads for added robustness
  - PROOF-Lite is fully compatible with full blown PROOF
- The following file systems are tested:
  - 4 disk 74GB 10K RPMS SAS RAID 0
  - 2 disk 320GB Intel X25-M SSD RAID 0
  - 1 disk 320GB Intel X25-M SSD
  - Sun Storage 7410 10x1TB 7200 SATA + ZFS log on SSD + 64GB RAM + 10GE

PROOF-Lite on Many Cores



Workload generated using the tutorials/proof/ProofTests suite

Conclusions

- ROOT 5.26 comes with a drastically optimized Tree buffer sizing and writing algorithm
- New parallel output merging in PROOF for large outputs
- Understanding all possible bottlenecks are required to build a PROOF instance that properly scales and uses all resources as efficiently as possible
- Many core machines pose a particular challenge on the I/O infrastructure, be it local or remote disks