High Performance Data Analysis via Coordinated Caches

High Performance Data Analysis via Coordinated Caches

M Fischer, C Metzlaff, E Kühn, M Giffels, G Quast, C Jung and T Hauth

Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe

E-mail: {max.fischer, eileen.kuehn, manuel.giffels, guenter.quast, christopher.jung, thomas.hauth}@kit.edu, christian.metzlaff@student.kit.edu

Abstract. With the second run period of the LHC, high energy physics collaborations will have to face increasing computing infrastructural needs. Opportunistic resources are expected to absorb many computationally expensive tasks, such as Monte Carlo event simulation. This leaves dedicated HEP infrastructure with an increased load of analysis tasks that in turn will need to process an increased volume of data. In addition to storage capacities, a key factor for future computing infrastructure is therefore input bandwidth available per core. Modern data analysis infrastructure relies on one of two paradigms: data is kept on dedicated storage and accessed via network or distributed over all compute nodes and accessed locally. Dedicated storage allows data volume to grow independently of processing capacities, whereas local access allows processing capacities to scale linearly. However, with the growing data volume and processing requirements, HEP will require both of these features. For enabling adequate user analyses in the future, the KIT CMS group is merging both paradigms: popular data is spread over a local disk layer on compute nodes, while any data is available from an arbitrarily sized background storage. This concept is implemented as a pool of distributed caches, which are loosely coordinated by a central service. A Tier 3 prototype cluster is currently being set up for performant user analyses of both local and remote data.

1. Introduction

Computing of High Energy Physics (HEP) collaborations [1, 2, 3, 4] of the LHC experiment is often seen as synonymous with the Worldwide LHC Computing Grid (WLCG) [5], globally coordinated workflows and international distribution of vast amounts of data. Much of the daily analysis work of HEP physicists is performed with only slight interaction with the grid, however. Local clusters, file servers, and interactive development environments make up the largest portion of resources used by many scientists.

Whereas the WLCG would have been unfeasible without dedicated developments from the HEP communities, local computing resources are built easily using standard technologies. Thus, improvements mostly have reduced direct benefit for scientists. Yet technological advances in local processing by third parties remain largely unadopted by HEP due to conflicting specializations. For example, the Hadoop [6] processing framework closely matches HEP analysis requirements, but is incompatible with binary data formats and dataset splitting.

To merge HEP and modern processing paradigms, the KIT CMS work group has developed a new middleware targeting end user analysis throughput. Our approach accelerates data driven analyses via selective caching, thereby freeing shared resources for less demanding applications.

2. End User Performance Studies

HEP user tasks can be divided into two categories: private Monte Carlo simulation, and data analysis. Since we expect simulation tasks to be handled by opportunistic resources in the future, our focus is on data analysis performance.

Previous studies [7] show that analyses based on the ROOT framework running on modern processors can be limited by input data rates. Following these technical studies, we decided to assess the limits of common input sources with realistic use cases. We have chosen CMS calibration studies as a reference workflow. It is the most I/O dependent of all CMS analyses performed by KIT research groups. The analysis codebase is well maintained since LHC run 1; most importantly, it includes optimizations for use in classic HEP environments, such as automatic use of TTreeCache.

Like all analyses, the calibration workflow uses a compact, skimmed dataset derived once from event data stored in the WLCG. This data is analyzed for a multitude of variates and parameters multiple times per day as the application is developed. Before the input data is superseded by a newer version, it is read hundreds to thousands of times.

The calibration analyses stand out due to most reconstruction of physical properties and objects being performed during skimming. This makes the analysis itself dependent on data input rate as few calculations are performed compared to e.g. Higgs searches. Each analysis processes a data volume of 1 TB to 5 TB, which is expected to increase by up to a magnitude with LHC run 2. We estimate required processing rates of 10 MB/s per core to achieve acceptable turnaround cycles.

Our benchmark deploys a fixed number of concurrent analysis jobs in a controlled environment. Jobs are launched using the regular analysis workflow but targeting the local host instead of a batch system. During execution key performance metrics are monitored. The *dstat* [8, 9] tool monitors metrics of the host, while the *time* system tool monitors individual jobs.

The benchmark was deployed repeatedly on the same worker node (see table 1), varying the number of concurrent processes and data sources (see figure 1). Four data sources were used: a dedicated file server connected via **1 Gb/s**, a separate worker node equipped only with SSDs mimicking a dedicated file server connected via **10 Gb/s**, four local disks connected to a software **RAID0x4**, and a single local **SSD**.

Table 1. test cluster worker node features.

CPU	2x	Intel Xeon E5-2650v2 @ 2.66 GHz (à 8 cores, 16 threads)
Memory	8x	8GB RAM
Cache	1x	Samsung SSD 840 PRO 512GB or
	2x	Samsung SSD 840 EVO 256GB
HDD	4x	WDC WD4000 4TB
Network	1x	Intel X540-T1 (10GigE/RJ45)

The 1 Gbit/s benchmark shows the issue of network as a shared resource: while at first throughput scales linearly with reading processes, this stalls at the maximum bandwidth of the connection at roughly 100 MB/s. No amount of additional hosts or CPUs can increase throughput beyond this barrier if the same connection is used.

Access via the better 10 Gbit/s connection shows notable performance improvements not just in scalability but also in speed. Since we expect a 10 times better bandwidth, we also expect

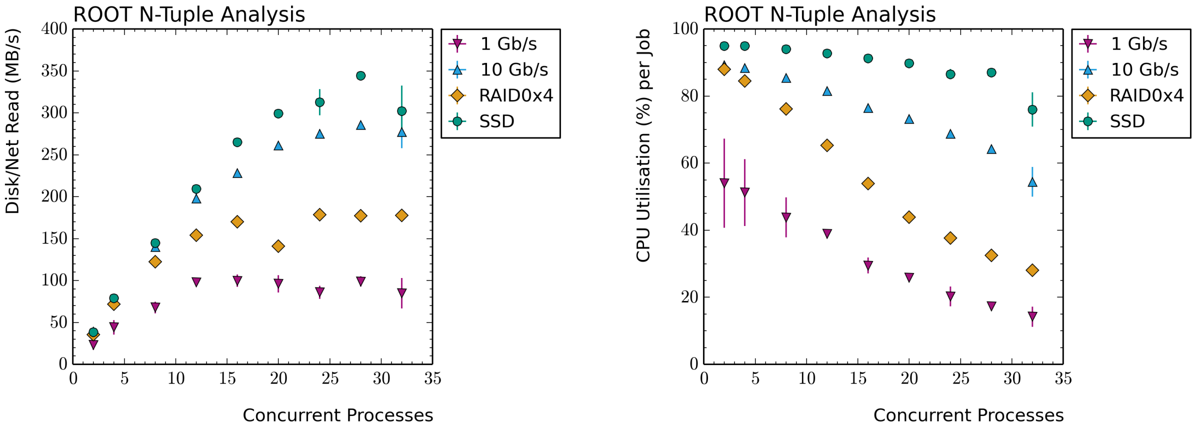

Figure 1. Sum of input rate and average of CPU usage for concurrent benchmark jobs on a single host. Both **1 Gb/s** and **10 Gb/s** benchmarks exclusively accessed data via network, whereas **RAID0x4** and **SSD** benchmarks used only local devices. Additional 48 processes were statically deployed on other hosts for the **10 Gb/s** benchmark; its range of local concurrent processes thus corresponds to 49 to 80 processes in total.

a stalling to be shifted to 10 times more processes. To measure this, we statically deployed 48 benchmark processes on additional worker nodes, concurrently accessing data via the same connection as the benchmark host. The 1 to 32 benchmarking jobs dynamically deployed thus equal 49 to 80 jobs concurrently reading data from a single 10 Gbit/s connection. As expected, we observe a gradual drop in CPU utilization when nearing 32 benchmark jobs (i.e. 80 concurrent readers), indicating a lack of input bandwidth.

It is worth pointing out that a 10 Gbit/s connection is the limit of feasible connections for a work group such as the one at KIT. Aside from funding limitations, a further upgrade would require a complete replacement of the network infrastructure.

The alternative to shared resources are those local to each worker node. Hard disk drives (HDD) combine adequate cost, volume and single read speed of roughly 1 Gbit/s. However, the spinning disk construction of HDDs mean reduced performance during random access. Even with four disks combined, read performance stalls at 2 to 3 concurrent accesses per device.

The alternative to local HDDs are local Solid State Drives (SSD). With no mechanical parts, there is no notable penalty for random access; in addition, a single device can provide roughly 5 Gbit/s read speed. Indeed our benchmark shows both advantages: Single read performance is the best of all tested data source and CPU usage degrades only slightly with many concurrent accesses, meaning optimal usage of system resources.

3. Feasibility of SSD device locality in HEP

While local SSD access is excellent for scalability and performance, the direct application of local SSD storage is not feasible: the cost per storage is up to an order of magnitude higher and storage volume about an order of magnitude lower compared to HDDs. This makes local SSD inadequate for primary storage. They are too valuable for replications or other fault tolerance mechanisms as well as occupation with inactive user data. Therefore, we consider SSDs only feasible as expendable, automated caches for larger, fault-tolerant background storage.

While many caching implementations exist in operating systems, file systems and analysis frameworks, none of these are sensitive to defining HEP workflow features: First, while jobs execute on individual hosts, actual workflows are distributed entities in a cluster; a caching infrastructure must therefore also act as a distributed pool. Second, workflows vary widely in

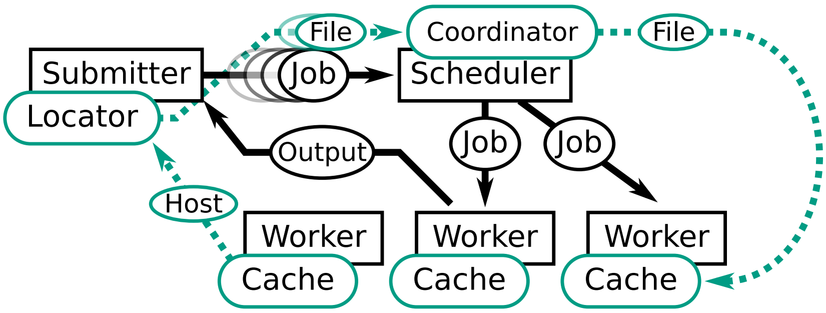

Figure 2. Layout of an HTDA cluster: Individual components can be closely identified by the batch system components they interact with or correspond to. Cache nodes are deployed on worker nodes, where they keep copies of input data on cache devices. Locator nodes are the frontend of the system, providing the location of files on submission. The coordinator nodes are data schedulers, calculating the importance of files, assigning them to cache nodes or dismissing them.

their features, with computationally intensive tasks not benefiting from higher input rates; thus, caching algorithms must be sensitive to workflow features.

Furthermore, HEP workflows impose constraints, be it for technical or usability reasons. Notably, most applications are made for execution in classic batch systems and usability strongly depends on (mostly) POSIX compliant storage. Also, splitting of data to jobs is usually performed on submission and practically never matches mechanisms of distributed file systems.

Based on these considerations, we conclude that it is not feasible to use local SSD storage for HEP efficiently. While many technologies exist that could address individual problems, it is not realistic to merge them without extensive modifications and drastic changes to workflows. Therefore, we propose the development of a distributed caching middleware.

4. The HTDA Middleware

In order to combine batch processing with coordinated caches, the KIT CMS group has developed a new middleware layer, called "**H**igh **T**roughput **D**ata **A**analysis" (HTDA) [10]. Providing a layer between batch system and remote storage, it transparently adds distributed caching for user workflows.

Fundamentally the middleware acts similar to a batch system, with the key difference of handling data instead of jobs. Consequently, we have chosen a design that mimics the layout of batch systems(see figure 2). A pool of nodes provides the persistent service for scheduling data to worker nodes and maintaining it, while job hooks submit new meta-data to the pool.

The HTDA layer consists of three node types or *components* which provide the distributed cache when working together:

Cache nodes maintain copies of files on cache devices, providing local access.

Locator nodes map file names to hosts caching them for quick lookup.

Coordinator nodes select files to cache and assign them to cache nodes.

4.1. Prototype Overview

The HTDA prototype implementation is built purely in Python 2.7 for extensibility and maintainability. As a central design choice, each component is built as a plugin to the same generic node application and based on the same code package.

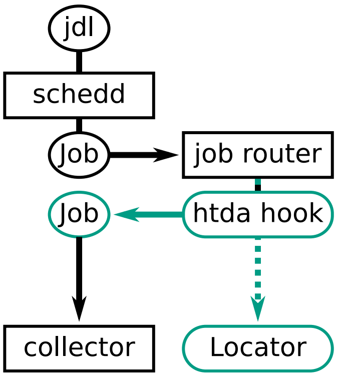

Figure 3. Interaction of HTDA hooks with an HTCondor batch system: Jobs submitted to an HTCondor *schedd* are claimed by the *job router* if they provide an input file list. It then calls the HTDA hooks, which publish job meta-data and add matching cache nodes to the host RANK expression. The *job router* actually creates a clone of the job with this information: the original mirros the clone, either exiting with it or taking its place if the service fails.

The major reason for this is the provisioning of a uniform infrastructure for components to act as part of a pool. Nodes employ heartbeats to dynamically determine the available members of their pool. In addition, facilities for remote method calls between nodes are provided. Both are exposed to components as a high-level interface allowing dynamic inter-component communication.

Interaction between components is stateless, ensuring that components may dynamically enter and exit the pool. We consider this integral to be future-proof in light of clusters expanding to opportunistic resources. Furthermore, stateless interactions make it possible to interface the HTDA infrastructure with external resources without requiring persistent clients.

Currently, communication is implemented as REST APIs using HTTP/HTTPS via CherryPy [11] servers and PycURL [12] clients; the abstraction allows us to consider alternative implementations, using e.g. message queues, for the future.

4.2. Batch System Integration

The general design of the middleware is agnostic to the batch system used. Only a single feature is strictly required: It must be possible to influence job scheduling to prefer specific hosts. All other interactions may be handled externally as part of the job submission and retrieval workflow, e.g. via automated management tools.

For our prototype, we have chosen to provide support for the HTCondor [13] batch system. Aside from an HTCondor test pool being present at EKP, this was largely motivated by the ease of interfacing with the batch system directly. Thus users have most functionality available as part of the existing infrastructure without requiring notable explicit or implicit tool usage. The only change required by users is for their jobs to provide an explicit input file list; this has easily been automated in our users' job submission tool [14], requiring no manual adjustment.

The HTDA software is linked into the HTCondor system via the *job router*. This optional service allows to run hooks on job events (submission, execution, success and removal) and modify the job information. For every job, the HTDA hooks query a locator node for worker nodes caching input files and modify the job's desired scheduling to prefer these nodes. In addition, they send job features to the pool, such as input files, owner or CPU and memory usage.

4.3. Pool Coordination

The information gathered from the batch system is aggregated by a coordinator node into a database. This provides possible caching algorithms with access to a history of detailed access and request information. The collected information is used to regularly rate the importance of each file currently known to the system. Given the response time of the overall system limited by jobs and file transfers, this computation may be complex and take into account correlations between files.

The prototype implementation currently uses an expanded *least recently used* [15] algorithm: the basic access time rating is weighted by the frequency of recent file accesses. This reflects the goal of maximizing cache usage per data transferred. To respect the splitting of datasets by jobs, the actual groups of files requested are rated; individual files are assigned the highest score of their groups.

Once files have been rated, the score is simply propagated to cache nodes: files already registered at a node have their score updated; files not yet assigned are distributed to nodes to match the group splitting used by jobs.

4.4. Data Provisioning

Cache nodes are agnostic to the distribution and rating scheme employed by the coordinators; only the meta-data and rating of files assigned to the cache are known[1]. Each node sequentially allocates the files it knows to the cache backends it manages. Files rated too low to be allocated are released and may be reassigned by coordinators.

Cache nodes require two types of backends, of which an arbitrary number may be in use: *Storage* backends are used to fetch files for caching. *Cache* backends are used to place files for better access. Access is abstracted with a minimalistic API, which must provide modification time and size of files, read access and, for cache backends, write access.

The prototype implementation provides POSIX backends for using local or NFS file systems. In addition, a *logical* cache backend is available: it slaves other backends, creating a single high-volume backend similar to LVM. Future support for object storage or grid storage, e.g. via XRootD [16], is feasible with the API. Additional logical backends are planned to better support distributed, shared file systems.

An integral design choice of HTDA is that the service itself does not handle cache access. Instead, access is redirected with external means. This makes usage resistant to service failure and, most importantly, provides lightweight access with minimal overhead.

For the POSIX backends of the prototype, we use union file systems to merge remote storage and local caches into a single logical file system (see figure 4). Tests with AUFS [17], a union file system available for Scientific Linux 6, show that any performance overhead is negligible against regular system performance fluctuations (see figure 5).

5. HTDA Testbed

For testing and development, the HTDA middleware is deployed as part of the EKP HTCondor cluster. Users have access to interactive login nodes for development, data management and job submission. A shared home directory as well as six file servers with a total capacity of 305 TB are available to users on the login nodes and worker nodes. The infrastructure is connected with a mixture of 1 Gbit/s and 10 Gbit/s connections.

5.1. EKP Analysis Cluster

The HTDA portion of the cluster runs on dedicated worker nodes, providing additional local storage capacities for caching. Two SuperMicro 2U Twin Servers, each having CPU dual socket, four SATA 2 and two SATA 3 ports, host a total of four worker/cache nodes. Per node (see table 1), we use two Intel Xeon E5-2650v2 for 32 logical cores and either two Samsung SSD 840

[1] A demonstrative implementation had caches use a time dependent rating *parameterized* by the coordinator; this reduces the computational load on the coordinator. Reimplementation of this feature via a plugin mechanism is under consideration.

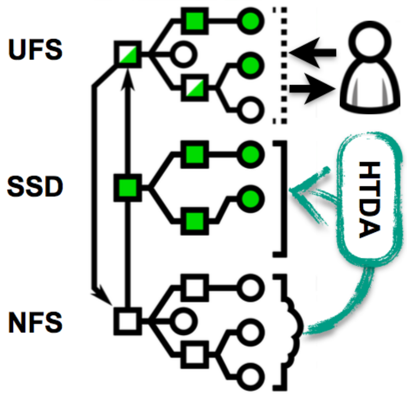

Figure 4. Data provisioning on caches: cache nodes work on hierarchies of cache and storage devices, copying and validating files. User access is provided via *union file systems*, squashing cache and storage mounts to a single directory structure. Reads iteratively try caches, while writes go directly to storage.

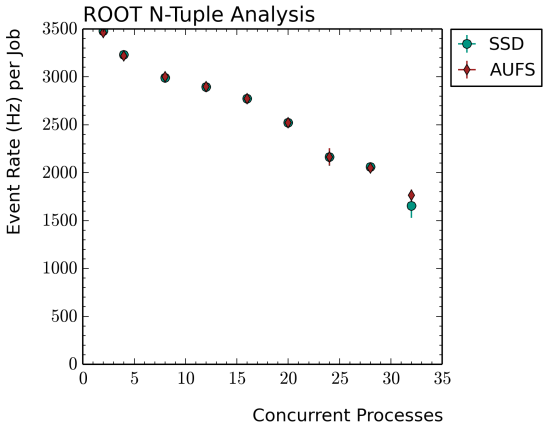

Figure 5. Performance of AUFS: the provisioning of cache access with *union file systems* is lightweight and independent of node performance. Benchmarks with end user analyses show no notable change of performance.

EVO 256GB or one Samsung SSD 840 PRO 512GB as cache devices. The cache nodes support all six file servers.

A single interactive login node is the entry point for users to the system. Here, the HTCondor submission node is configured to use the HTDA hooks, which in turn connect to the locator and coordinator nodes.

5.2. Operational Experience

The HTDA test cluster has been in test operation for about 12 weeks. During this time, the system has handled about 400k jobs successfully. No major problems occured.

Preliminary stress tests have not revealed any apparent scaling issues of the middleware. The batch system hooks can handle the submission of several hundred jobs per second. Memory usage of nodes is in the order of 200 MB under full load. Cache and coordinator nodes require between 5% to 10% of CPU, which primarily depends on their work cycle frequency; Locator nodes generate no notable CPU load.

In order to benchmark the performance of jobs, we deploy a reference workflow to the cluster. The workflow is a CMS calibration analysis, reading about 400 GB of LHC run 1 data.

6. Conclusion and Outlook

This paper discusses and evaluates dedicated cache developments for batch systems. This is relevant to optimize infrastructure utilization and thus costs. In the past, user needs and increasing data size were addressed by implicit, vertical scaling of network capacities. To date, new developments promoting horizontal scaling have not been adopted in HEP.

Therefore, we evaluated distributed caching to improve horizontal scaling. Benchmark results with end user analyses show an imminent limit for network throughput in classical batch clusters. At the same time they reveal concurrency or capacity limits for local storage. We conclude that no existing solution adequately combines shared and local resources efficiently.

Instead, we suggest a middleware layer for batch systems to provide distributed caching via local devices. To show its efficiency, a prototype for the HTDA middleware is developed at

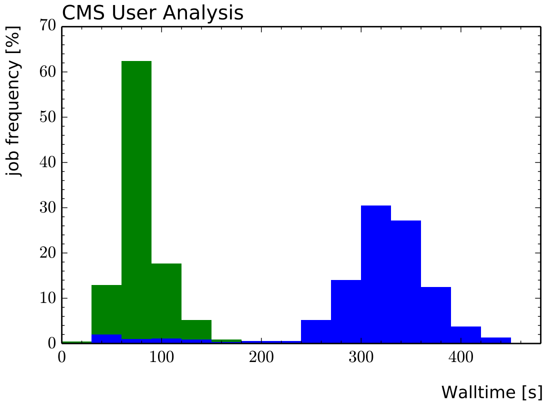

Figure 6. Speedup of cache for analysis performance: Walltime distribution for the same workflow deployed on the HTDA test cluster, with cache enabled or disabled. The system generates a consistent speedup once regular access is limited by shared resources.

KIT by the CMS work group. It hooks into batch systems to cache files in respect to executed workflows. The usage of the HTDA prototype improves existing data driven analyses in both efficiency and throughput. Additionally, horizontal scalability is given by design.

Future improvements for the caching alhorithm focus on leveraging workflow information for predictive caching. Additions to the caching mechanisms will enable to use XRootD to cache access to other grid sites as well as efficiently using distributed, parallel file systems available in HPC and cloud resources.

Acknowledgments

The authors wish to thank all people and institutions involved in the project Large Scale Data Management and Analysis (LSDMA), as well as the German Helmholtz Association, and the Karlsruhe School of Elementary Particle and Astroparticle Physics (KSETA) for supporting and funding the work.

References

[1] Grandi C, Stickland D, Taylor L et al. 2004 The CMS computing model preprint CERN-LHCC-2004-035/G-083
[2] Jones R and Barberis D 2008 The ATLAS computing model J. Phys.: Conf. Series **119** 072020 doi:10.1088/1742-6596/119/7/072020
[3] Brook N 2004 LHCb computing model CERN-LHCb-2004-119
[4] Carminati F et al. 2004 ALICE computing model CERN-LHCC-2004-038/G-086
[5] Shiers J 2007 The Worldwide LHC Computing Grid (worldwide LCG) *Computer Physics Communications* 1-2 **177** 219-23
[6] The Hadoop project homepage **URL** http://hadoop.apache.org/
[7] Fischer M, Giffels M, Jung C, Kühn E and Quast G 2015 Tier 3 batch system data locality via managed caches *J. Phys.: Conf. Series* Proceedings of 16th International workshop on Advanced Computing and Analysis Techniques (to be published)
[8] The dstat project homepage **URL** http://dag.wiee.rs/home-made/dstat/
[9] The dstat project repository **URL** https://github.com/dagwieers/dstat
[10] The HTDA project repository **URL** https://bitbucket.org/kitcmscomputing/hpda
[11] The CherryPy project repository **URL** http://cherrypy.org
[12] The PycURL project repository **URL** http://pycurl.sourceforge.net
[13] Thain D, Tannenbaum T and Livny M 2005 Distributed computing in practice: the Condor experience *Concurrency Computat.: Pract. Exper.* **17** 323–56 (doi: 10.1002/cpe.938)
[14] The grid-control project homepage **URL** https://ekptrac.physik.uni-karlsruhe.de/trac/grid-control
[15] Lee D, Choi J, Kim J, Noh, S.H., Min S L, Cho Y and Kim C S 2001 LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies *IEEE Transactions on Computers 12* **50** 1352-61
[16] The XRootD project homepage **URL** http://xrootd.org
[17] The AUFS project homepage **URL** http://aufs.sourceforge.net