Production Experience with the ATLAS Event Service

D Benjamin[1], P Calafiura[2], T Childers[6], K De[3], W Guan[4], T Maeno[5], P Nilsson[5], V Tsulaia[2], P Van Gemmeren[6] and T Wenaus[5] on behalf of the ATLAS Collaboration

[1]Duke University, 134 Chapel Drive, Durham, NC 27708, USA
[2]Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, USA
[3]University of Texas at Arlington, 701 South Nedderman Drive, Arlington, TX 76019, USA
[4]University of Wisconsin, 1150 University Avenue, Madison, WI 53706, USA
[5]Brookhaven National Laboratory, PO Box 5000, Upton, NY 11973, USA
[6]Argonne National Laboratory, 9700 S. Cass Ave, Argonne, IL 60439, USA

E-mail: VTsulaia@lbl.gov

Abstract. The ATLAS Event Service (AES) has been designed and implemented for efficient running of ATLAS production workflows on a variety of computing platforms, ranging from conventional Grid sites to opportunistic, often short-lived resources, such as spot market commercial clouds, supercomputers and volunteer computing. The Event Service architecture allows real time delivery of fine grained workloads to running payload applications which process dispatched events or event ranges and immediately stream the outputs to highly scalable Object Stores. Thanks to its agile and flexible architecture the AES is currently being used by grid sites for assigning low priority workloads to otherwise idle computing resources; similarly harvesting HPC resources in an efficient back-fill mode; and massively scaling out to the 50-100k concurrent core level on the Amazon spot market to efficiently utilize those transient resources for peak production needs. Platform ports in development include ATLAS@Home (BOINC) and the Google Compute Engine, and a growing number of HPC platforms.

After briefly reviewing the concept and the architecture of the Event Service, we will report the status and experience gained in AES commissioning and production operations on supercomputers, and our plans for extending ES application beyond Geant4 simulation to other workflows, such as reconstruction and data analysis.

ATL-SOFT-PROC-2017-006
07 January 2017

1. Introduction

The ATLAS Experiment [1] processes its data at about 140 computing centers around the world at a scale of about 4M CPU-hours/day. To date it has accumulated a globally distributed data volume in excess of 220 Petabytes. Even with such a massive processing scale, the experiment is resource limited. The ATLAS physics program can benefit from applying more compute resources to Monte Carlo simulation, and over the next decade the situation will become even more critical because the LHC [2] and ATLAS upgrade programs will bring an order of magnitude increase in computing requirements. In view of the steady demand for new computing resources, it becomes very important for the experiment to not only efficiently use all CPU power available to it, but also to proactively leverage opportunistic computing resources.

Opportunistic computing resources have a large potential for expanding the ATLAS processing pool. Such resources include cost-effective clouds such as the Amazon spot market [3], supercomputers (HPCs), shared Grid resources and volunteer computing (ATLAS@Home) [4]. Porting of regular ATLAS workloads (e.g. simulation, reconstruction) to opportunistic resources does not come for free. In order to use them fully and efficiently ATLAS has implemented a fine-grained event processing system - the ATLAS Event Service (AES) [5] - in which the job granularity changes from input files to individual events or event ranges. The Event Service delivers fine-grained workload to the running event processing application (the payload) in real time. After processing each event range (about 10 min processing time), the Event Service writes the corresponding output into a separate file and saves the output file into a secure location, such that Event Service jobs can be terminated practically at any time with minimal data losses. This architecture allows the Event Service to efficiently adapt to the characteristics of opportunistic resources, in which a job slot lifetime is unpredictable and may be either very short or very long.

In order to efficiently utilize CPU resources of supercomputers, we have developed an HPC-specific implementation of the Event Service called Yoda [6], which leverages MPI for running massively parallel event processing jobs on multiple HPC compute nodes simultaneously. Yoda has been developed and prepared for production usage on the Edison supercomputer at the National Energy Research Scientific Computing Center (NERSC), Berkeley, USA. Since late 2015 Yoda has been running ATLAS simulation production workloads at NERSC and in 2016 it delivered about 20M CPU hours to the experiment.

In section 2 of this paper we describe the concept and the architecture of the Event Service. Yoda is described in Section 3 and the AES commissioning status is presented in Section 4. During the commissioning phase of Yoda we studied various factors which can have a visible effect on the CPU efficiency of compute nodes. Such factors include initialization time of the payload application, sequential running of several payloads on a compute node within the same MPI-submission, and handling of fine-grained outputs. The former two factors are discussed in Section 5, while in Section 6 we present the results of our studies of the performance of Object Stores, which are used by the Event Service as an intermediate storage for fine-grained outputs produced by payload applications.

2. The ATLAS Event Service

The JEDI [7] (Job Execution and Definition Interface) extension to PanDA [8] adds new functionality to the PanDA server to dynamically break down tasks in a way that optimally utilizes available processing resources. With this capability, tasks can be broken down at the level of either individual events or event clusters (ranges). This functionality allowed us to develop the ATLAS Event Service capable to dynamically deliver to a compute node only that portion of the input data which will be actually processed there by the payload application. Input data is streamed to the compute node in real time in small portions. While the payload persists, it can elastically continue to consume new inputs and stream away outputs with no need to tailor workload execution time to resource lifetime. A schematic view of the Event Service workflow is shown in Figure 1.

On the compute node the PanDA Pilot establishes a connection with the PanDA server over HTTP and starts a parallel event processing application (payload) in order to utilize all available CPU cores. The payload application in the Event Service is represented by AthenaMP [9], a process-parallel version of the ATLAS data processing framework Athena. AthenaMP starts as a serial process, which first goes through the application initialization phase, then forks several event processors (workers) and informs the pilot that it is ready for data processing. The pilot downloads event range identifiers (strings) from the PanDA server and delivers them in real time to the running AthenaMP application, which assigns them to its workers on a first-come, first-served basis. The worker uses the event range string to locate the corresponding input file

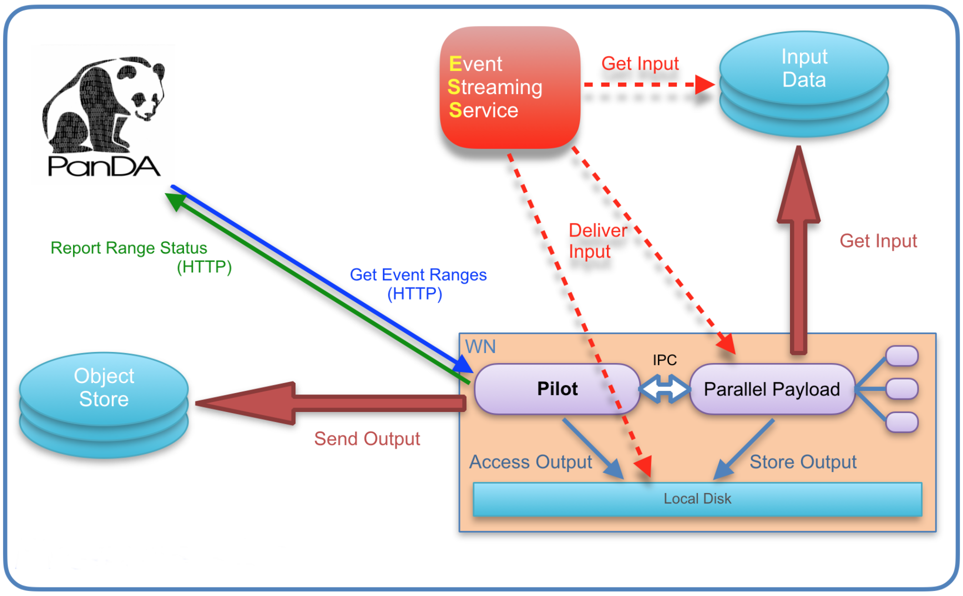

Figure 1. Schematic view of the Event Service

and find event range data within the file. After processing the given event range, the worker writes the output into a separate file on the local disk and declares its readiness to process another event range. AthenaMP reports back to the Pilot the locations of output files produced by its workers and the Pilot takes care of streaming the outputs in real time to a remote storage system (Object Store), and informing the PanDA server of the event range completion status.

In the present architecture each AthenaMP worker individually reads input event data, which couples data reading and its associated latency with the event processing. In the long term we plan to make data retrieval across the WAN fully asynchronous to the processing in order to avoid inefficiencies from WAN latency. Data access will be mediated by the Event Streaming Service (ESS), represented by the red box on Figure 1. ESS is not yet part of the deployed Event Service. It is in development and is expected to provide us with additional efficiency measures such as utilizing local cache preferentially over WAN access, and marshaling data sent over the WAN to limit data transferred to what is actually needed by the payload. An important step towards design and implementation of the ESS is the development and testing of a first prototype of asynchronous data pre-fetching on compute nodes.

3. Yoda - Event Service on HPC

Supercomputers are one of the important deployment platforms for the Event Service. However, compute nodes on most HPC machines are not connected to the outside world over WAN. This limitation makes it impossible to deploy the conventional Event Service on such supercomputer systems because in the AES architecture the PanDA Pilot running on a compute node must communicate with central services (e.g. job brokerage and data aggregation facilities) over the network. In order to overcome this limitation we have developed an HPC-specific implementation of the Event Service, called Yoda, which leverages MPI to run on multiple compute nodes simultaneously. A schematic view of Yoda is presented on Figure 2.

Yoda is an MPI application which gets submitted to the HPC batch system by a specialized component of the PanDA Pilot running on the HPC edge node, i.e. the node which is connected to the WAN. The Pilot also downloads input data to the HPC Shared File System, gets job

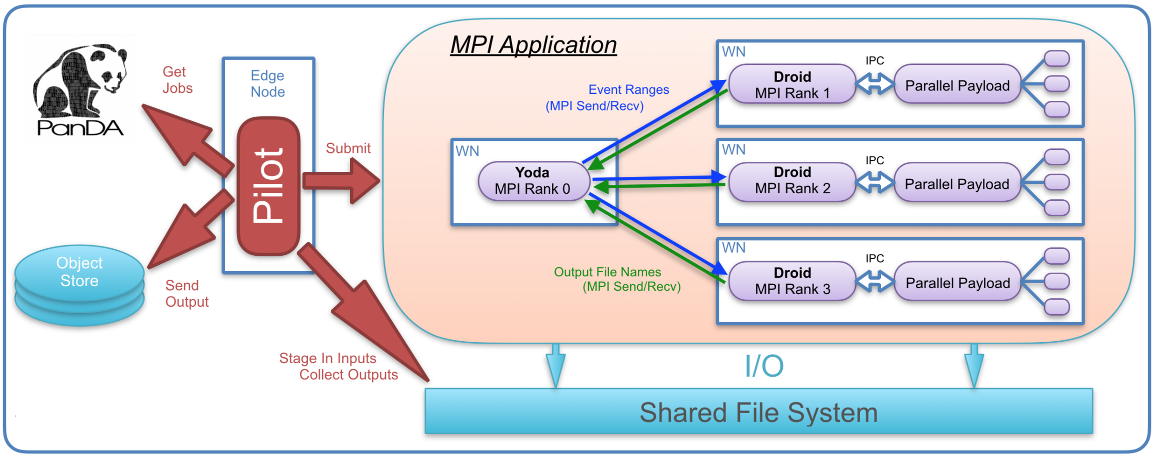

Figure 2. Schematic view of Yoda

definitions from the PanDA server and streams out the outputs produced by Yoda jobs to the Object Store. Yoda applications implement the master-slave architecture in which rank 0 is the master and all other ranks are the slaves. For the development of Yoda ranks we reused the code of the conventional Event Service and implemented lightweight versions of PanDA JEDI (hence Yoda, a diminutive Jedi) and the PanDA Pilot (Droid). Yoda (rank 0) orchestrates the entire MPI-application by continuously distributing fine-grained workloads to Droids (rank N, N!=0) and collecting their outputs. On a compute node Droid starts the payload application and delivers the workload to it exactly the same way the PanDA Pilot does it on compute nodes of the conventional Event Service applications. This allows us to run the same configuration of AthenaMP payload on HPC and on other Event Service platforms such as the Grid and Clouds. The outputs produced by AthenaMP on the compute nodes are temporarily stored to the HPC Shared File System until the Pilot streams them out to Object Stores.

4. Event Service commissioning

We have chosen ATLAS Geant4 Simulation [10, 11] as a first use-case for the Event Service in general and for Yoda in particular. Simulation jobs use a substantial fraction of the ATLAS CPU budget on the Grid which makes it very beneficial for the experiment to offload its simulation to other computing platforms such as opportunistic resources and HPCs. On the other hand, simulation jobs are CPU-intensive with minimal I/O requirements and relatively simple handling of in-file metadata, characteristics which allowed us to make rapid progress in the development of the Event Service and Yoda components and to begin commissioning the Event Service for production usage.

Until recently NERSC supercomputers (Edison and Cori Phase I) had been our primary platforms for the development and commissioning of the AES. We started to run Simulation production workloads with the Event Service (Yoda) on the Edison HPC in late 2015, and in 2016 Yoda delivered about 20M CPU-hours to the ATLAS collaboration. Also in late 2015 we successfully scaled Event Service up to 50,000 concurrent processors on the Amazon Spot Market cloud. In Summer 2016 the Event Service commissioning effort was shifted over to Grid sites and it has been showing steady progress since then. Event Service deployment on volunteer computing (ATLAS@Home) has not progressed significantly due to manpower shortages.

5. Performance studies

During the commissioning of Yoda on the Edison supercomputer we studied various factors which can have a visible effect on the CPU efficiency of Yoda ranks. In this section we discuss payload initialization time and sequential running of several payloads on a compute node within the same MPI submission.

5.1. Payload initialization

During its initialization step AthenaMP reads a large number of files from the disk. These files include python scripts, shared libraries, XML configuration files, static replicas of the geometry and conditions database, etc. If the ATLAS offline software release is installed on the HPC shared file system, then the concurrent reading of software installation files by many compute nodes during the payload initialization phase can lead to a serious performance bottleneck. For example, we have observed rather poor scaling of AthenaMP initialization time on Edison compute nodes when all instances of AthenaMP were accessing a software release installed on Edison's scratch file system (Lustre).

In order to work around this problem we package the entire ATLAS software release into a single tarball. At the beginning of its execution the Droid first unpacks this tarball into the memory-resident disk on the compute node, then it starts AthenaMP and lets it initialize on the local copy of the software release. With this approach we eliminate concurrent reading of the shared release installation by all Yoda payloads which considerably speeds up the initialization phase of the entire Yoda application.

Although with this mechanism we achieved very good scaling up to 1,000 concurrent starts, the preparation of software release tarballs requires considerable manual effort and so is not considered sustainable in the long run. On the Cori Phase I supercomputer we studied AthenaMP initialization performance scaling by installing software releases on different systems including Lustre, Burst Buffer [12] and Shifter [13]. So far the results obtained with the Shifter system look the most promising.

5.2. Sequential running of multiple payloads on the same compute node

Before submitting Yoda jobs to the HPC batch system, the Pilot first needs to get the workload from a PanDA production task. This mechanism is illustrated by Figure 3. PanDA tasks consist of many jobs and each job requires processing of many events. When a new task gets defined in PanDA all its jobs contain the same number of events. Depending on the number of ranks (compute nodes) allocated for a given Yoda job, the Pilot decides how many PanDA jobs should be processed by this MPI-job and passes this information over to Yoda. Yoda then assigns each PanDA job to one or more ranks. The strategy here is to keep each compute node busy for the entire lifetime of the MPI-job. In cases when Yoda does not have enough time to process all events from a PanDA job, all leftover events are returned back to the PanDA server, which generates new PanDA jobs containing only these leftover events. This mechanism leads to the creation of many PanDA jobs with a number of events less than the task's default number per job.

If Yoda has to process PanDA jobs with a small number of events, it assigns several such jobs to a single compute node. The Droid running on this compute node deals with multiple PanDA jobs in sequence, which means several instances of AthenaMP are started and stopped by the Droid during its lifetime. While AthenaMP is going through the initialization phase, all CPU cores on the node are idling and in this way significant CPU time is wasted. Dealing with multiple PanDA jobs within a single PanDA task in this way quite often leads to rather poor overall CPU efficiency of Yoda jobs. In the future we plan to overcome this problem by implementing a new concept of Jumbo Jobs in PanDA. With Jumbo Jobs each production task in PanDA will be represented by a single PanDA job. Thus, Yoda will not have to deal

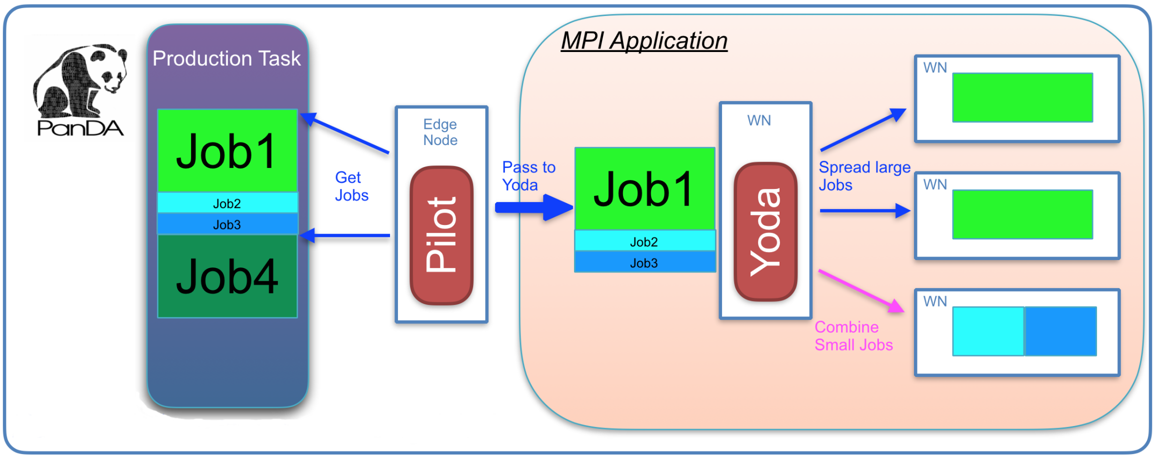

Figure 3. Yoda dealing with multiple PanDA jobs

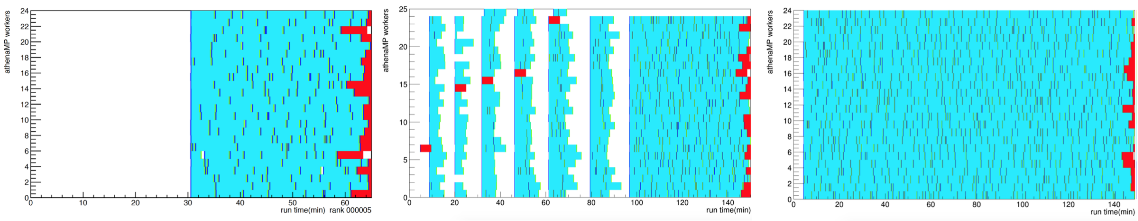

Figure 4. CPU efficiency of Yoda compute nodes

with multiple PanDA jobs and no time will be wasted initializing more than one instance of AthenaMP on a single compute node.

5.3. CPU efficiency of Yoda compute nodes

Figure 4 shows three time-line plots demonstrating CPU efficiency of Yoda compute nodes. These plots were obtained from Yoda test runs on Edison. The X axis of each plot shows the wall time in minutes since the beginning of Droid execution on the compute node, and each bin on the Y axis corresponds to one CPU core (Edison compute nodes have 24 physical CPU cores). The white color on the plot means the core is idle, turquoise means the core is processing an event and red means event processing was started but not finished for some reason (e.g. segmentation fault occurred, or the job was killed because it reached its wall time limit).

- The plot on the left is an example of poor CPU efficiency caused by the very long initialization time of AthenaMP;
- The plot in the middle is an example of poor CPU efficiency caused by running more than one PanDA job on a single compute node;
- The plot on the right is an example of good CPU efficiency: just one PanDA job runs on the compute node, initialization is fast and the number of events is enough to keep the node busy for the entire job lifetime.

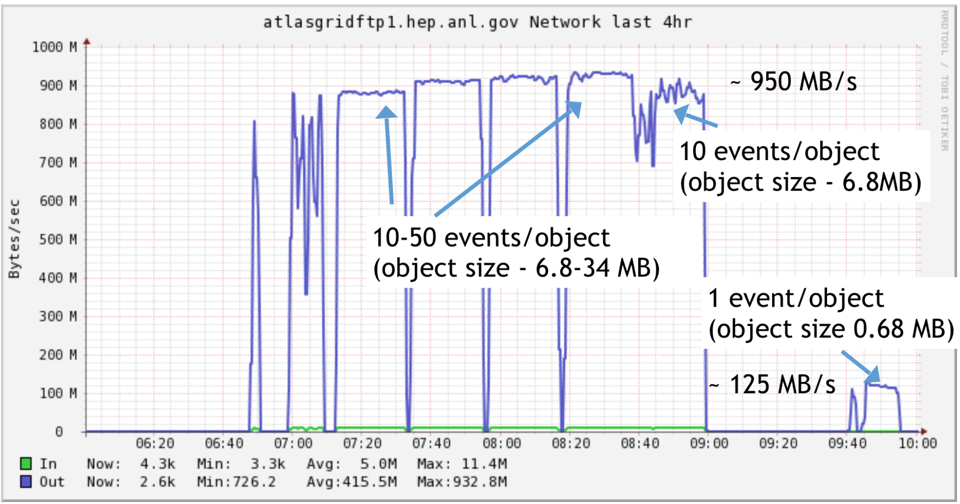

Figure 5. OS bandwidth dependency on the object size

6. Interaction with Object Stores

Intermediate output files produced by the Event Service payload applications are shipped in real time to the Object Stores (OS). PanDA then generates specialized jobs which merge these files into final outputs. Such merge jobs usually run on Grid sites. The initial implementation of Yoda was sending event range outputs directly from Edison compute nodes to the Object Store at BNL. The stage-out process was coupled with the event processing, therefore data transfer issues (e.g. network connection problems, slow file upload) were affecting the CPU efficiency of Yoda compute nodes. To avoid these problems we decoupled data transfer to the OS from event processing on the compute nodes, making output uploading the responsibility of the Pilot, which runs on the HPC edge node.

As part of the Event Service commissioning at NERSC we studied the Object Store performance by running a series of tests which involved uploading of objects of different size to the CEPH OS at BNL. We observed that the clients can overload the OS with various errors occurring including authentication errors, inability to connect to a bucket, inability to write an object, long running writer, etc. This suggests that either the client software should have retry and perhaps queuing capabilities, or we need a server side system that can regulate OS writes.

Another important observation is that we can achieve much higher bandwidth by increasing the object sizes. This is demonstrated on Figure 5, which shows that by grouping 10-50 events into a single transfer (transfer size 6.8-34 MB) we achieved 950MB/s upload speed vs 125MB/s for single event transfers (transfer size 0.68 MB).

7. Summary

The Event Service has been commissioned to run ATLAS Geant4 Simulation production on HPC systems. The commissioning process on Grid sites is well underway and other deployment platforms (e.g. clouds, volunteer computing) are expected to follow.

Several important lessons were learned during the development and testing of Yoda at NERSC:

(i) Primary causes of sub-optimal usage of CPU resources on the compute nodes are slow initialization of the payload and the fact that for the time being Yoda must combine multiple PanDA jobs into a single MPI-submission;

(ii) By staging out large numbers of small files we can saturate Object Stores;

(iii) Data stage out must be decoupled from event processing.

By addressing the issues listed above we were able to successfully scale production Yoda jobs up to 700 compute nodes (almost 17,000 cores) on Edison HPC at NERSC.

In the future we plan to further develop Event Service functionality by implementing the Event Streaming Service. Also we will be applying the Event Service to other ATLAS production workflows beyond Geant4 Simulation (e.g. Reconstruction and Analysis) with the ultimate goal to make the Event Service a unified workflow architecture across all ATLAS computing platforms.

8. Acknowledgments

The results presented in this paper have been obtained by using resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] ATLAS Collaboration, 2008 *JINST* **3** S08003

[2] L. Evans and P. Bryant LHC Machine, 2008 *JINST* **3** S08001

[3] The Amazon Elastic Computing Cloud, http://aws.amazon.com/ec2/

[4] Adam-Bourdarios C et al. on behalf of the ATLAS Collaboration 2015 ATLAS@Home: Harnessing Volunteer Computing for HEP *J. Phys.: Conf. Ser.* **664** 022009

[5] Calafiura P et al. on behalf of the ATLAS Collaboration 2015 The ATLAS Event Service: A new approach to event processing *J. Phys.: Conf. Ser.* **664** 062065

[6] Calafiura P et al. on behalf of the ATLAS Collaboration 2015 Fine grained event processing on HPCs with the ATLAS Yoda system *J. Phys.: Conf. Ser.* **664** 092025

[7] De K, Golubkov D, Klimentov A, Potekhin M and Vaniachine A on behalf of the ATLAS Collaboration 2014 Task Management in the New ATLAS Production System *J. Phys.: Conf. Series* **513** 032078

[8] Maeno T for the ATLAS Collaboration 2008 PanDA: Distributed production and distributed analysis system for ATLAS *J. Phys.: Conf. Series* **119** 062036

[9] Calafiura P et al. on behalf of the ATLAS Collaboration 2015 Running ATLAS workloads within massively parallel distributed applications using Athena Multi-Process framework (AthenaMP) *J. Phys.: Conf. Ser.* **664** 072050

[10] GEANT4 Collaboration, S. Agostinelli et al., 2003 *Nucl. Instrum. Meth. A* **506** 250

[11] ATLAS Collaboration 2010 ATLAS Simulation Infrastructure *Eur. Phys. J* **C70** 823

[12] Bhimji W et al. 2016 Extreme I/O on HPC for HEP using the Burst Buffer at NERSC. Proceedings of the CHEP2016 conference *J. Phys.: Conf. Ser.*

[13] Gerhardt L et al. 2016 Using Shifter to Bring Containerized CVMFS to HPC. Proceedings of the CHEP2016 conference *J. Phys.: Conf. Ser.*