Running ATLAS workloads within massively parallel distributed applications using Athena Multi-Process framework (AthenaMP)

This content has been downloaded from IOPscience. Please scroll down to see the full text.
(http://iopscience.iop.org/1742-6596/664/7/072050)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 137.138.124.206
This content was downloaded on 24/02/2016 at 13:24

Please note that terms and conditions apply.
Running ATLAS workloads within massively parallel distributed applications using Athena Multi-Process framework (AthenaMP)

Paolo Calafiura¹, Charles Leggett¹, Rolf Seuster², Vakhtang Tsulaia¹ and Peter Van Gemmeren³ on behalf of the ATLAS Collaboration

¹Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, USA
²TRIUMF, 4004 Wesbrook Mall, Vancouver, BC V6T 2A3, Canada
³Argonne National Laboratory, 9700 S. Cass Ave, Argonne, IL 60439, USA
E-mail: VTsulaia@lbl.gov

Abstract. AthenaMP is a multi-process version of the ATLAS reconstruction, simulation and data analysis framework Athena. By leveraging Linux kernel copy-on-write (COW) mechanisms, it allows for sharing of memory pages between event processors running on the same compute node with little to no change in the application code. Originally targeted to optimize the memory footprint of reconstruction jobs, AthenaMP has demonstrated that it can reduce the memory usage of certain configurations of ATLAS production jobs by a factor of 2. AthenaMP has also evolved to become the parallel event-processing core of the recently developed ATLAS infrastructure for fine-grained event processing (Event Service) which allows the running of AthenaMP inside massively parallel distributed applications on hundreds of compute nodes simultaneously. We present the architecture of AthenaMP, various strategies implemented by AthenaMP for scheduling workload to worker processes (for example: Shared Event Queue and Shared Distributor of Event Tokens) and the usage of AthenaMP in the diversity of ATLAS event processing workloads on various computing resources: Grid, opportunistic resources and HPC.

1. Introduction
Memory has always been a scarce resource for ATLAS [1] reconstruction jobs. In order to optimally exploit all available CPU-cores on a given compute node, we needed to have a mechanism which would allow sharing of memory pages between processes or threads. Several years ago we developed AthenaMP - a multi-process version of the ATLAS reconstruction, simulation and data analysis framework Athena [2] - which relies on the Linux kernel copy-on-write (COW) mechanism. The basic strategy of AthenaMP is to go through the initialization phase of a job in a single master process, initialize and allocate as much memory as possible, then fork several worker processes, and let them process all events assigned to the job. The workers share a substantial fraction of process memory and this memory sharing is provided solely by COW mechanisms.

Originally implemented as a Python layer [3], AthenaMP was later completely rewritten in C++ to follow the Gaudi Component Model [4]. It supports several strategies for scheduling
workloads to worker processes (e.g. Shared Event Queue, Shared Distributor of Event Tokens), each of which is implemented by a specialized Gaudi AlgTool.

AthenaMP has demonstrated that it can reduce the memory usage of certain configurations of ATLAS production jobs by a factor of 2. For validating AthenaMP we compared physics results obtained by running serial and multi-process Athena jobs and confirmed that they were identical. As a result, ATLAS currently uses AthenaMP for running a large fraction of its production jobs on multi-core resources of the Grid.

AthenaMP has also evolved to become the parallel event-processing core of the recently developed ATLAS infrastructure for fine-grained event processing (Event Service), which is our approach to running ATLAS specific workloads on diverse computing resources, such as supercomputers, spot market clouds and volunteer computing.

In section 2 of this paper we describe the architecture of AthenaMP. Section 3 contains memory profiling results, which demonstrate the memory optimization achieved by the use of AthenaMP when compared to running several sequential Athena jobs in parallel. Section 4 focuses on the usage of AthenaMP for running ATLAS production jobs on multi-core Grid resources. In section 5 we briefly introduce the ATLAS Event Service with emphasis on its event processing component, which is currently implemented by a special configuration of AthenaMP. Finally, in section 6 we describe future developments for AthenaMP: the implementation of shared reader and writer processes.

2. AthenaMP

The simplest scenario for achieving process-based parallelism on a multi-core host is to run $N$ instances of the same application simultaneously. By doing this, however, nothing is shared between these application instances apart from dynamically loaded objects in memory and files opened in read-only mode. Thus, such parallel approach cannot be considered optimal in terms of the usage of system resources. AthenaMP addresses the resource sharing issue by forking several worker processes from a single master process, and then relying on the Linux copy-on-write\(^1\) mechanism for sharing memory pages between forked processes. The workflow of AthenaMP jobs is represented in Figure 1.

AthenaMP always starts as single master process, which goes through the initialization phase of the job and after that forks several sub-processes (the workers), which then process all events assigned to the given job. By delaying the fork as much as possible, we increase the number of read-only memory pages, which get allocated and initialized by the master process. Later on these memory pages become shared by the forked sub-processes and, as a result, an optimal overall memory footprint of the AthenaMP job is achieved. Relying on COW ensures that all read-only memory pages will be shared and this will be done automatically for us by the Linux kernel.

Another advantage of the fork-based multi-process approach implemented by AthenaMP is that all communications between processes are handled by the framework itself. As a result, from the perspective of user code, there is absolutely no difference between running either within a serial Athena job or within AthenaMP. Thus, no algorithmic code changes were required for porting ATLAS software to AthenaMP.

The master process also creates one additional auxiliary process, whose responsibility is to schedule workloads for the worker processes. The complexity of the ATLAS Event Data Model does not allow us to directly exchange event data objects between processes. As a consequence, for the time being, this auxiliary process distributes just event identifiers - either integers or strings - between the workers. The only exception to this approach is the reconstruction of raw data coming directly from the detector, for which the events are represented as void* memory

\(^1\) http://www.tldp.org/LDP/khg/HyperNews/get/memory/linuxmm.html
buffers, and are delivered to the worker processes directly via a Shared Memory segment. In order to achieve load balancing on the worker processes, the event identifiers are assigned to them on a first-come, the first-served basis. AthenaMP supports different implementations of this strategy, which are used in different contexts (e.g. Grid, Event Service). The default implementation relies on the usage of a shared queue object: event numbers are inserted into the queue by the auxiliary process, and pulled out from the queue by the worker processes.

With the event identifiers in hand, each worker process reads event data from the input file, processes the events and writes its own output file to the disk. Once all events assigned to the given job have been processed by the workers, the master process collects the locations of the workers’ output files and decides whether or not these outputs need to be merged. Such decision is usually based on job configuration parameters, output file sizes and types.

![Schematic view of ATLAS AthenaMP](image)

**Figure 1.** Schematic view of ATLAS AthenaMP

### 3. Memory optimization with AthenaMP

It has been demonstrated several times by various tests that good memory optimization is achieved (on the order of 50%) in ATLAS reconstruction jobs, when AthenaMP is used instead of several sequential Athena processes running in parallel on the same compute node. A typical example of such memory optimization is shown on the Figure 2. This is a memory profiling plot, which compares total amount of RSS of 8 serial Athena jobs to one AthenaMP job with 8 worker processes. The plot was obtained by running tests on an otherwise empty machine and by profiling the memory usage with the `free` utility.

The job used in this example consists of two steps: digitization and reconstruction. The shared part of the memory consists of detector geometry description, magnetic field map, run-dependent detector conditions data, various read-only caches, etc. The exact amount of the shared memory portion depends on the concrete job type and configuration. As is seen in the plot, much better memory sharing is achieved for AthenaMP workers at the reconstruction step, where by switching to AthenaMP we reduce the overall memory footprint by 45%.

![Memory profiling plot](image)
4. AthenaMP on the Grid

In order to run AthenaMP jobs in the ATLAS distributed production system, it was necessary to define special multicore batch queues, where the unit resource is a whole compute node. After successful testing and validation, today ATLAS runs a substantial fraction of its production workloads on multicore Grid resources using AthenaMP. Current workloads include reconstruction and Geant4 [5] simulation jobs. Figure 3 shows the number of CPU-cores used by ATLAS production jobs - single core and multi core - on the Grid in March 2015, where roughly 50% of the total number cores are used by AthenaMP jobs.

5. Processing fine-grained workloads with AthenaMP

5.1. Event Service

The new implementation of the ATLAS production system [6] includes a JEDI extension to PanDA [7], which adds a new functionality to the PanDA server to dynamically break down the tasks based on optimal usage of available processing resources. With this new capability, the tasks can now be broken down at the level of either individual events, or event clusters (ranges), as opposed to the traditional file-based task granularity. This allows the recently developed ATLAS Event Service [8] to dynamically deliver to a compute node only that portion
of the input data which will be actually processed there by the payload application (simulation, reconstruction, data analysis) thus avoiding costly pre-staging operations for entire data files. The Event Service leverages modern networks for efficient remote data access and highly-scalable object store technologies for data storage. It is agile and efficient in exploring diverse, distributed and potentially short-lived (opportunistic) resources: “conventional resources” (Grid), supercomputers, commercial clouds and volunteer computing.

The Event Service is a complex distributed system, in which different components communicate to each other over the network using HTTP. In this system AthenaMP plays a key role of the event processing (payload) component. A PanDA pilot starts an AthenaMP application on the compute node and waits until it goes through the initialization step and forks worker processes. After that the pilot requests an event-based workload from the PanDA/JEDI, which is dynamically delivered to the pilot in the form of event ranges. The event range is a string, which - together with other information - contains positional numbers of events within the file and the file GUID. The pilot streams event ranges to the running AthenaMP application, which takes care of event processing and making output files (a new output file for each range). The pilot monitors the directory in which the output files are produced, and as they appear, sends them to an external aggregation facility for final merging.

In order to be able to retrieve event data corresponding to the given range, AthenaMP first needs to convert the range into a list of event tokens. For this purpose we have developed a specialized utility - the Token Extractor - which retrieves event tokens from the external source (Event Index), and a specialized AthenaMP sub-process - the Token Scatterer - which receives tokens from the Token Extractor and distributes them between the workers. Event tokens allow AthenaMP workers to retrieve event data directly from the input files, which can be either local or remote. In latter case AthenaMP relies on the remote ROOT I/O mechanisms for data retrieval. Finally, AthenaMP uses the Output File Sequencer for making output files, a new mechanism recently developed by ATLAS, which allows the production of a sequence of output files by a single Athena job.

5.2. Yoda - Event Service for supercomputers
We consider supercomputers (HPC) to be one of the deployment targets for the Event Service. However, the conventional Event Service architecture assumes that its sub-components can communicate with each other over the network via HTTP (for example: Pilot and PanDA server). On the other hand, most of the supercomputer systems do not provide internet connectivity from compute nodes to the outside world. Thus, in order to be able to run on such machines a different implementation of the Event Service was necessary, for which we leveraged MPI instead of HTTP for inter-component communication. The ultimate goal was to develop a MPI-program, which implements Event Service architecture and which can run within HPC systems on multiple compute nodes simultaneously.

By reusing the code of conventional the Event Service components, we implemented a lightweight version of PanDA JEDI called Yoda [9] (Yoda - diminutive JEDI), which was capable of communicating over MPI with a lightweight implementation of PanDA Pilot (Droid). In this way we turned Event Service into a MPI-job, in which one rank runs Yoda (rank 0, the master) and all remaining ranks run Droids (the slaves). Like the conventional Event Service, Yoda also uses AthenaMP as its event processing component, which communicates with Droids exactly the same way as is done inside the regular Event Service. Thus, no changes in AthenaMP were required for running within Yoda systems.

5.3. Running at large scale
We have chosen the ATLAS Geant4 simulation [10] as a first use-case for the Event Service. Simulation jobs used more than half of ATLAS CPU-budget on the Grid in 2014, thus by
offloading simulation to other computing platforms (e.g. HPC, clouds) we can free a substantial amount of ATLAS Grid resources. Also, simulation is a CPU-intensive application with minimal I/O requirements and relatively simple handling of meta-data. These characteristics of Geant4 simulations greatly simplified the development process and allowed us to deliver a first working implementation of the Event Service on a short time scale.

As part of the Event Service testing and validation runs on cloud resources (Amazon spot market) and on HPC machines (Edison supercomputer at NERSC, Berkeley, USA) with ATLAS production simulation workloads, we succeeded in running many instances of AthenaMP within massively parallel distributed applications. For example, Yoda has been scaled up to simultaneous running of 4.2K instances of AthenaMP (i.e. over 100K worker processes) on the Edison machine.

6. Future developments
The current mechanism of scheduling workloads to AthenaMP worker processes, when the auxiliary process distributes event identifiers between the workers and lets them read event data independently, is rather inefficient. As a result, for each ROOT basket in the input file there are duplicate reads from several workers, and also the same decompressed baskets are kept in the memory of more than one worker. In order to optimize this process we plan to implement a Shared Event Reader process, aka Event Source, which will be a single place within an AthenaMP job for reading input data objects and decompressing ROOT baskets. The input data objects will be kept in the memory of the reader process, and delivered to the workers via Linux IPC mechanisms (Shared Memory).

In addition to the Shared Reader process, we also plan to implement a Shared Writer process, aka Event Sink, which will collect the output data objects from the workers on-the-fly and write them out to the disk. By introducing the Shared Writer process we plan to decrease load on the file system (which is very important, for example, for running on large supercomputers with a shared file system) and also to eliminate the need for merging workers’ outputs.

Finally, the ATLAS meta-data infrastructure needs to be thoroughly changed in order to cope with fine-grained workloads, file sequencing and merging.

7. Summary
By running ATLAS reconstruction in one AthenaMP job with several worker processes we achieve a significantly reduced overall memory footprint when compared to running the same number of independent serial jobs. AthenaMP has successfully passed physics validation procedures and is now being widely used by ATLAS for running production workloads (e.g. simulation, reconstruction) on Grid multi-core resources. Originally developed for optimizing the memory footprint, AthenaMP has become an efficient mechanism for processing fine-grained event-based workloads within ATLAS Event Service. Recent scaling tests of the Event Service have allowed us to run AthenaMP within massively parallel distributed applications on up to 100K CPU-cores simultaneously.

In order to improve AthenaMP performance even further, we plan to develop specialized I/O sub-processes of AthenaMP - Event Source and Event Sink. The implementation of such processes is a non-trivial task, but is expected to bring visible performance benefits when running AthenaMP on the Grid, and is absolutely critical for further functional enhancements to the ATLAS Event Service.

8. Acknowledgments
For the development, testing and validation of Yoda we used 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