Multi-threaded checksum computation for ATLAS high-performance storage software

Fabrice Le Goff and Giuseppe Avolio
CERN, Esplanade des Particules 1, P.O. Box, 1211 Geneva 23, Switzerland
E-mail: fabrice.le.goff@cern.ch giuseppe.avolio@cern.ch

Abstract. ATLAS is one of the general purpose experiments observing hadron collisions at the LHC at CERN. Its trigger and data acquisition system (TDAQ) is responsible for selecting and transporting interesting physics events from the detector to permanent storage where the data are used for further processing. The transient storage of ATLAS TDAQ is the last component of the online system in the data flow. It records selected events at several GB/s to non-volatile storage before transfer to offline permanent storage. The transient storage is a distributed system consisting of high-performance direct-attached storage servers accounting for 480 hard drives. A distributed multi-threaded C++ application operates the hardware. The transient storage is also responsible for computing a checksum for the data, which is used to ensure data integrity of the transferred data. Reliability and efficiency of this system are critical for the operations of TDAQ as well. This paper presents the existing multi-threading strategy of the software and how the available hardware resources are used. We then introduce how multi-threaded checksum computation was introduced to increase significantly the maximum throughput of the system. We discuss the key concepts of the implementation with a focus on the importance of overhead minimization. Finally the paper reports on the tests done on the production system to demonstrate the validity of the implementation and measurements of the performance improvement in the view of future LHC and ATLAS upgrades.

1. Introduction

ATLAS [1] is one of the general purpose experiments observing hadron collisions at the LHC [2] at CERN. Its trigger and data acquisition system (TDAQ) [3] is responsible for selecting and transporting interesting physics events from the detector to permanent storage where the data are used for further processing. TDAQ is a complex system consisting of thousands of computers interconnected by several networks. Its reliability and efficiency have a direct impact on those of ATLAS.

The last component in the data flow of TDAQ is a storage system that records selected physics events data to non-volatile storage. It buffers the data before they are transferred out of the experiment's facility to a permanent storage complex provided by CERN. This online storage system can store up to 432 TB of data, which represents about 60 hours of data-taking under normal operation conditions. It enables the ATLAS experiment to be independent of the possible failures of CERN's permanent storage system and the connection to it.

As it is part of the online TDAQ system, the performance of the transient storage system has a direct impact on the data-taking capabilities of ATLAS. During Run 2, the operation period from 2015 to 2018, this system was able to record data at a sustained rate of 2 GB/s, with peak up to 5 GB/s.

2. The Software Application

The input of TDAQ's transient storage consists of physics events selected by the High-Level Trigger (HLT) of the experiment. On top of selection, the HLT tags each event with one or more classes, called streams, which constitute consistent sets of event for physics analysis, calibration tasks, data quality monitoring, etc. The HLT sends the selected events to the transient storage via a 10GbE network.

In terms of hardware, the transient storage system consists of eight servers operating by pairs four direct-attached storage devices accounting for a total of 480 hard drives.

A distributed in-house application written in C++ operates this hardware: it receives the data from the network and writes them to disks. Each event stream is written to a dedicated sequence of files. The software uses multi-threading methods to distribute its workload on the available CPU cores. On top of writing data to disk, the application is also responsible for computing a checksum for the data. This checksum is later used to check the correctness of data transfers. Zlib's [4] adler32 is used as the checksum algorithm.

The workload of the application involves therefore both I/O- and CPU- intensive tasks:

- an incoming event is dispatched to the thread associated with its stream,
- checksum is computed for each stream of events,
- and finally each stream is written to a dedicated sequence of files.

The application is completely data-driven: its workload depends on the rate, size, and stream distribution of the incoming events. Figure 1 shows the stream throughput distribution for a typical ATLAS run in 2018. With this workload and one stream clearly dominating the throughput, the application is CPU-bound, limited by the checksum computation capacity for a single CPU core. At peak input rate, the application uses 3.2 of the 12 available CPU cores. The limitation really comes from single core performance, the necessary serialization of events in each stream, and the real-time nature of the workflow.

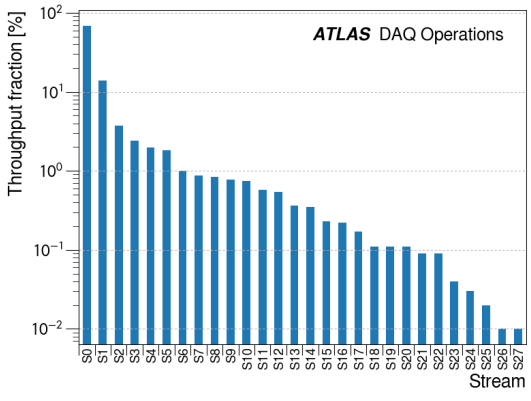

Figure 1. Stream throughput distribution for a typical ATLAS run in 2018 [5]. Events are classified in streams by the High-Level Trigger. This distribution determines the workload of the storage application since each stream must be processed sequentially, i.e. by a single thread. The distribution is unbalanced: one stream dominates the total throughput (more than 66 %). This constitutes a challenge for the software as it cannot simply divide the workload to take advantage of modern multi-core CPUs.

3. Multi-Threaded Checksum Computation

Checksum computation is essentially a sequential process as it requires computing the same elementary operation on the data seen as a stream of bytes. Nevertheless adler32 (as well as CRC32) provides the possibility of combining the checksums of two consecutive chunks of data resulting in the same checksum as if computed for the concatenation of the two original chunks at once [6]:

```
adler32("AB") = adler32_combine(adler32("A"), adler32("B"))
```

This feature can be used to split the events in smaller chunks, compute their checksum in parallel, and combine their result into the original checksum. With a bit of overhead, one can

take advantage of unused CPU resources, increasing the maximum throughput of the storage application. Computing overhead is the set of ancillary time and CPU resources required by an implementation; here, overhead consists of everything on top of the checksum computation itself.

We then introduce a multi-threaded version of the checksum computation. Figure 2 outlines the implementation: the additional steps, as compared to a single thread implementation, are shown in dark gray. This implementation has been designed to:

- be a drop-in replacement for every `adler32(...)` call; it is functionally equivalent and does not require any change in the calling code;
- be configurable: the feature can be disabled, and the number of threads available for checksum computation can be tuned;
- minimize the computing overhead if the feature is not used, either disabled by user configuration or due to operating conditions; this is a safe fallback mechanism: the new implementation shall not worsen the performance and the reliability of the original application;
- minimize the computing overhead when used: as this paper shows in the next section multi-threading overhead has a significant impact on this implementation's performance.

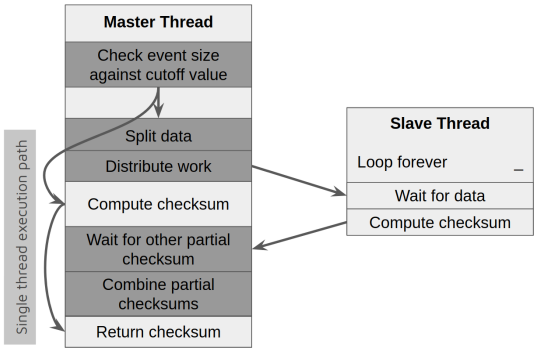

Figure 2. Outline of the multi-threaded checksum implementation. Elements in dark gray represent overhead compared to a single-threaded implementation. When only one thread is used due to configuration or event size (curved arrows), only the first element is an overhead. The synchronization mechanism (straight arrows) constitutes the bulk of the time overhead, as the other elements are very fast.

4. Multi-Thread Overhead

The most important contribution to the overhead of the multi-threaded implementation comes from the necessary thread synchronizations: the input data needs to be distributed to the slave threads, and their results fetched after computation. The `adler32_combine` call has constant execution time with respect to input data size ($O(1)$), and was measured to take around 15 ns on our CPUs.

Synthetic measurements were done to compare the different possible implementations of distributing input data chunks to computing threads and retrieving partial results; three scenarios were studied:

(i) Create a thread for each chunk of data and wait for its termination to get its result.

(ii) Create a configurable number of slave threads at application startup, executing a computing loop that sleeps and uses a condition variable to wake up when data are to be processed, and another condition variable to inform the master thread that the result is ready.

(iii) Create a configurable number of slave threads at application startup, executing a computing loop that sleeps and uses a condition variable to wake up when data are to be processed; the master thread request the result as soon as it can and busy-waits for it.

The last version is justified by the fact that the master thread is also a computing thread: it dispatches N-1 chunks of data and compute the checksum of the n-th chunk itself. When it has finished computing, the chances that the slave threads already have their result ready are very high; therefore the busy-wait loop is almost never called or will be very short. This favors responsiveness at the cost of CPU resources usage and avoids the second condition variable overhead of the second scenario.

Synthetic tests gave the following results for overhead in microseconds per chunk checksum computation:

(i) 12 µs (ii) 5 µs (iii) 3 µs

These results are expected: thread creation is a more complex process than the notification of a condition variable, and the third measurement validates the assumption that the busy-wait loop is more responsive in this specific case. The third scenario was therefore implemented.

5. Results

Figure 3 shows the checksum computation throughput as a function of the event size for the original single-threaded implementation and the new multi-threaded implementation configured with one, two, three, and four threads. As can be seen, when configured with one thread only, the multi-threaded version performs as well as the single-thread version. For multiple-thread configurations, the incompressible overhead leads to a minimum event size under which the multi-thread implementation is slower than a single-thread one. A cutoff value is then used to determine if multiple threads are to be used to compute the checksum for every event. As shown in this plot, this value is independent from the number of threads and only depends on the implementation. Minimizing the overhead is then essential to maximize the number of events that can take advantage of this feature.

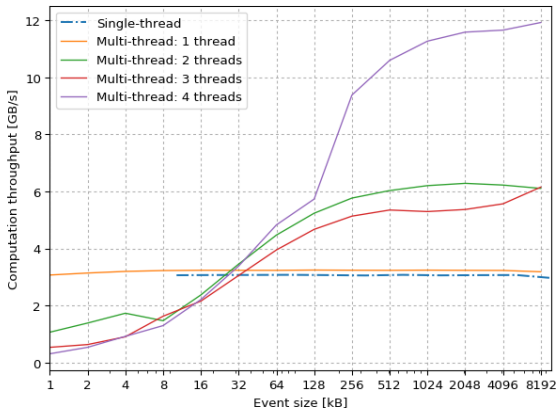

Figure 3. Checksum computation throughput as a function of event size for different implementations and configurations.

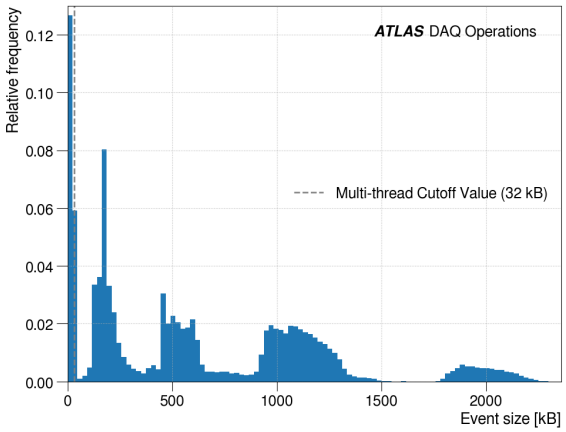

Figure 4. Relative frequency of selected-event sizes for a typical ATLAS Run during 2018 operations [5].

Figure 4 shows the relative frequency of event sizes for a typical ATLAS run in 2018. The different event size populations visible in the plot are due to complete and partially built streams, as well as from different detector readout configurations. With our final implementation and a chosen cutoff value of 32 kB, 82 % of the events have their checksum computed by multiple threads: they represent 99.6 % of the total data size.

Finally, Figure 5 shows the writing throughput of the storage application normalized to the single-threaded implementation throughput, as a function of the number of threads allocated

to checksum computation. The maximum gain is obtained when using all the physical CPU cores available on the servers: +40 % performance. Measurements between 12 and 24 cores show that the checksum computation does not benefit from SMT technologies (also known as hyper-threading).

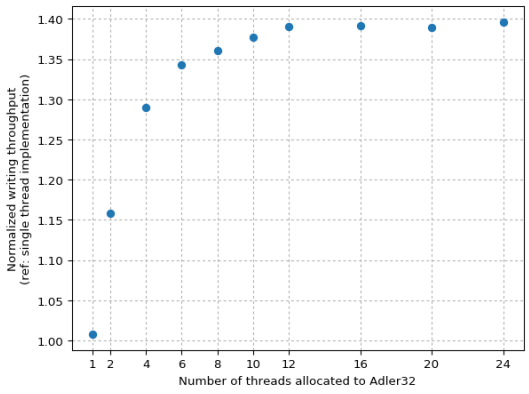

Figure 5. Writing throughput of the storage application, normalized to the original implementation performance, as a function of the numbers of thread allocated to checksum computation. Tests were performed on servers with 2 CPUs x 6 cores x 2 SMT cores. The maximum gain is obtained when using all physical CPU cores. Checksum computation does not take advantage of the SMT technology (virtual CPU cores).

6. Conclusion

The transient storage system of the ATLAS TDAQ system decouples the online operations from the offline systems, to make it more resilient to failures of offline systems and their connectivity. Its workload is essentially unbalanced. The performance of the storage system is determined by the necessary serialization of events, the real-time nature of the workflow and single-CPU-core performance for checksum computation. A novel multi-threaded checksum computation strategy was implemented. Significant efforts were made to reduce the overhead introduced by this strategy to maximize the performance gains. The final implementation increased the performance of the transient storage system by up to 40 % on synthetic tests and is ready to be deployed in the production system for the Run 3 of the LHC to start in 2021.

Acknowledgments

Thanks go to Alejandro Santos for originally pointing me to `adler32_combine`.

References

[1] The ATLAS Collaboration 2008 *Journal of Instrumentation* **3** S08003–S08003
[2] Evans L and Bryant P 2008 *Journal of Instrumentation* **3** S08001
`http://stacks.iop.org/1748-0221/3/i=08/a=S08001`
[3] The ATLAS TDAQ Collaboration 2016 *Journal of Instrumentation* **11** P06008
`http://stacks.iop.org/1748-0221/11/i=06/a=P06008`
[4] Deutsch P and Gailly J L 1996 `https://www.rfc-editor.org/rfc/rfc1950.txt`
[5] The ATLAS TDAQ Collaboration
`https://twiki.cern.ch/twiki/bin/view/AtlasPublic/ApprovedPlotsDAQ`
[6] Gailly J L and Adler M 2017 `https://www.zlib.net/manual.html#Checksum`