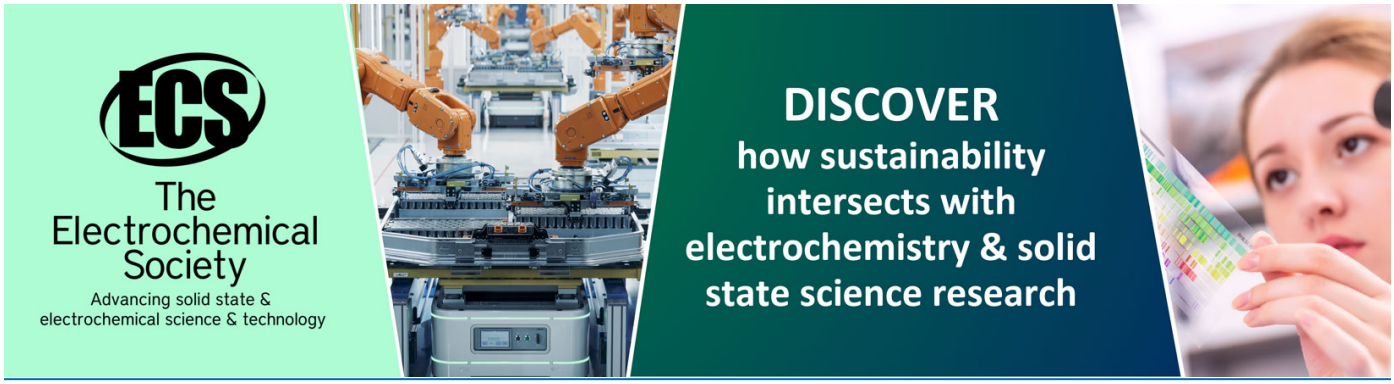
I/O Strategies for Multicore Processing in ATLAS



Journal of Physics: Conference Series **396** (2012) 022054 doi:10.1088/1742-6596/396/2/022054

I/O Strategies for Multicore Processing in ATLAS

P van Gemmeren[1], S Binet[2], P Calafiura[3], W Lavrijsen[3], D Malon[1] and V Tsulaia[3] on behalf of the ATLAS collaboration

[1]Argonne National Laboratory, Argonne, Illinois 60439, USA

[2]Laboratoire de l'Accélérateur Linéaire/IN2P3, 91898 Orsay Cédex, France

[3]Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

E-mail: gemmeren@anl.gov

Abstract. A critical component of any multicore/manycore application architecture is the handling of input and output. Even in the simplest of models, design decisions interact both in obvious and in subtle ways with persistence strategies. When multiple workers handle I/O independently using distinct instances of a serial I/O framework, for example, it may happen that because of the way data from consecutive events are compressed together, there may be serious inefficiencies, with workers redundantly reading the same buffers, or multiple instances thereof. With shared reader strategies, caching and buffer management by the persistence infrastructure and by the control framework may have decisive performance implications for a variety of design choices. Providing the next event may seem straightforward when all event data are contiguously stored in a block, but there may be performance penalties to such strategies when only a subset of a given event's data are needed; conversely, when event data are partitioned by type in persistent storage, providing the next event becomes more complicated, requiring marshalling of data from many I/O buffers. Output strategies pose similarly subtle problems, with complications that may lead to significant serialization and the possibility of serial bottlenecks, either during writing or in post-processing, e.g., during data stream merging. In this paper we describe the I/O components of AthenaMP, the multicore implementation of the ATLAS control framework, and the considerations that have led to the current design, with attention to how these I/O components interact with ATLAS persistent data organization and infrastructure.

1. Introduction

ATLAS has developed AthenaMP, a multicore implementation of its event processing framework Athena, which exploits process based parallelism and the Linux kernel 'Copy On Write' (COW) mechanism to run on multi-core computing architectures [1]. Initially, the primary goal of AthenaMP was to reduce the total memory consumption, as memory is a scarce resource on most multicore systems. Memory demands for a single ATLAS reconstruction job can be larger than the hardware memory limit per core, preventing ATLAS to utilize all cores of a single node.

Within AthenaMP, there is a single initialization stage run in the master process, which allocates and initializes shareable objects (such as detector geometry, magnetic field and conditions data) in

memory. Then worker processes are forked to bootstrap and process the events, each worker is scheduled to a disjoint set of events which are processed without relying on multi-threading or sub-event parallelism. The COW mechanism allows these workers to share enough memory to enable ATLAS to utilize all cores on a node without getting close to the physical limit. AthenaMP has successfully run reconstruction and simulation with little to no change to the application code.

At this point another system resource, which is equally scarce and harder to share, has moved into the development focus of AthenaMP: Input and Output. As computing speed of multi- and many-core computing systems grows, the imbalance to their I/O bandwidth becomes more significant. The strategies for I/O in the AthenaMP framework are discussed in this paper.

2. Current AthenaMP I/O infrastructure

In the current AthenaMP framework, multiple worker processes handle their I/O independently using distinct instances of the serial I/O framework [2] (see figure 1). No collective I/O operations are performed and each worker process produces its own output file. Changes to the I/O framework were limited to allowing worker re-initialization and seeking to event data by entry number. The output files of all processes need to be merged in a serial processing step and a special merge routine was developed to avoid severe performance degradation (see section 4.2).

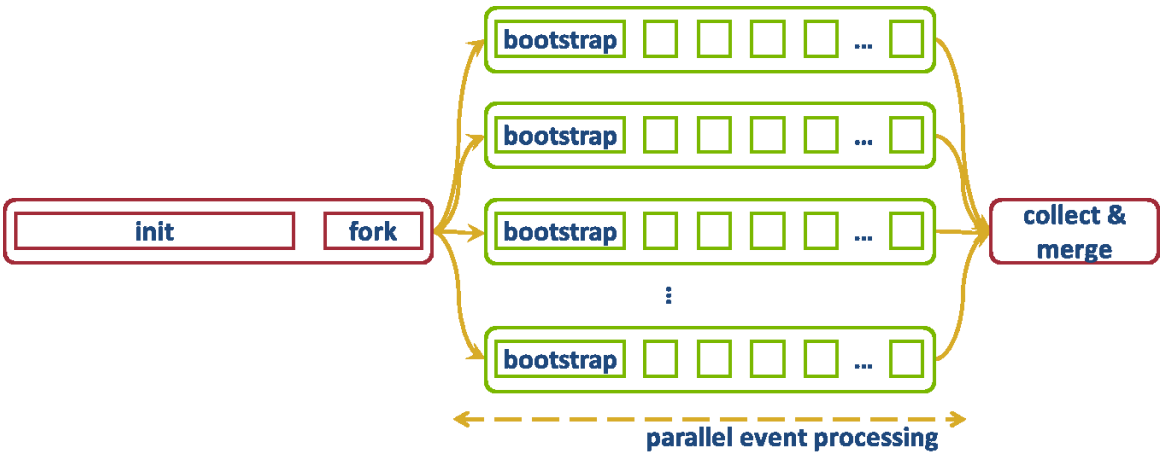


Figure 1. The current AthenaMP control framework for parallel event processing is focused on reducing the total memory footprint and uses serial I/O infrastructure components.

3. Data storage

ATLAS uses two very different formats to store data processed by either Athena or AthenaMP framework: Raw data is stored in a so-called ByteStream format and generated, simulated reconstructed and derived data products are stored via POOL / ROOT [3][4].

3.1. Raw data (ByteStream).

ATLAS Raw data are events as delivered by the Event Filter for reconstruction. They are essentially a serialization of detector readouts, trigger decisions and Event Filter calculations stored in ByteStream format that consists of simple C-style structures. The uncompressed event size is about 1.2 MB, but since 2011, ATLAS compresses Raw data events, which saves about 50% disk storage. Event-wise compression, rather than compression of the whole file, preserves efficient single event reading, as only data of the event of interest needs to be decompressed (see figure 2).

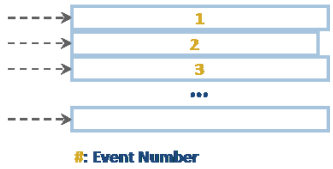


Figure 2. ByteStream format: Raw data events are event-wise compressed and can be individually read in a single transaction.

3.2. Simulated, reconstructed, derived data: (POOL / ROOT).
Data Objects for simulation and reconstruction are much more complex than Raw data. The ATLAS transient data model is written in C++, and uses multiple and virtual inheritance, polymorphism, template classes and methods, Standard Template Library and Boost classes, and a variety of external packages. Each transient data model class has a persistent counterpart, which is simpler and allows for schema evolution using Transient/Persistent conversion. Collections of persistent objects are streamed member-wise into individual ROOT TBranches, so they can be read separately on demand. ROOT compresses adjacent entries into shared baskets and ATLAS chooses that, depending on data product, 5 / 10 events share the same basket (see figure 3).

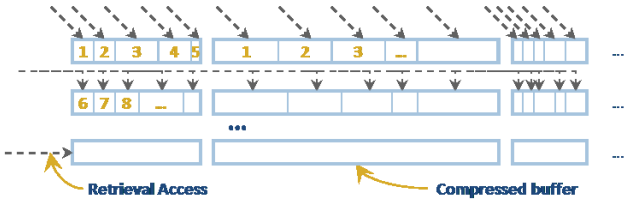


Figure 3. POOL / ROOT format: Processed event data is written as collections of data objects, which are compressed into baskets containing data for multiple events.

4. Handicaps of current I/O infrastructure on multicore platforms for ROOT data

The absence of a well-designed multicore I/O infrastructure has consequences on the performance of the AthenaMP framework in regards to CPU and wall clock times, memory consumption and output file size. Even the read performance on the created data product may be degraded.

4.1. Read data.
A process, either master or each of the worker, may read part of an input file to retrieve one event or a collection of objects for one event. In the current AthenaMP framework, all worker processes use the same input file. This causes multiple disk accesses that may mean poor read performance, especially if events are not consecutive. For POOL / ROOT, where collections are read on demand, it is quite likely that a worker will retrieve a collection, after the same collection for a later event has already been processed by a different worker. This may cause a 'back read' which hurts I/O performance.

4.1.1. Decompress / Stream ROOT baskets. Each worker process will retrieve its own event data, which means reading many ROOT baskets, decompressing them and streaming them into persistent objects (see figure 4). ROOT baskets contain object member of several events, so multiple workers use the same baskets and each of them will decompress them independently. Such behavior wastes CPU time for multiple decompress operations on the same data and is memory inefficient, as multiple copies of the same decompressed baskets are created in memory by different workers.

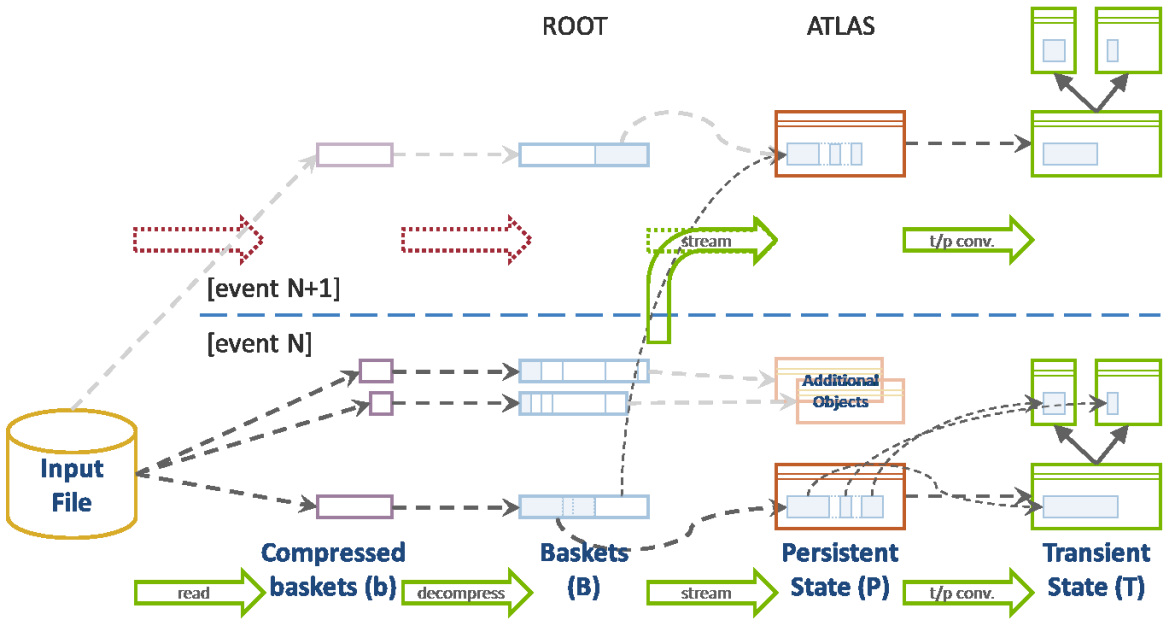


Figure 4. Reading event data from (POOL) / ROOT via the Athena I/O framework. The lower part of the diagram shows the reading, decompression, streaming and transient/persistent conversion of an event and the upper half shows the same for a collection from the next event, which in AthenaMP is done by a different worker process. Therefore reading and decompression (shown shaded) are duplicated. If both events are processed by the same worker (such as in serial Athena), the data for the subsequent events is streamed from the existing decompressed Baskets.

4.2. Write data.
Each AthenaMP worker process writes its own output file which needs to be merged serially. Inside the Athena framework, a job to merge the individual output files, would read (decompress, stream, convert persistent state to transient object) all data from all files, do nothing, and rewrite (convert transient object to persistent state, stream, compress) the data into a merged file. Compared to event processing this takes about 5% of CPU time, but because it has to be done in serial, this is the instant performance bottleneck.

With this merge procedure, the event throughput of AthenaMP suffers greatly and is, depending on system parameters, about 30% lower than without merge. Therefore, a utility which uses fast event data append and a reference redirection layer was developed (see figure 5). Together with in-file metadata propagation done by the Athena framework, the total merge time is reduced by almost an order of magnitude. In-file metadata merging requires Athena, as it is summarized (not just appended) and needs semantics that cannot be provided by an I/O layer. Because the merge has to be executed in serial, this remains a potential bottleneck and the re-writing of data increases the likelihood of file corruption.

4.2.1. Compress / Stream to ROOT baskets. Writers compress their data separately, which will result in suboptimal compression factor especially for small baskets (cause by either small number of entries or high split level). This will cost storage as output files can be larger and may waste CPU and user time when reading the resulting data files. This strategy is also not memory efficient, as each worker needs its own set of output buffers.

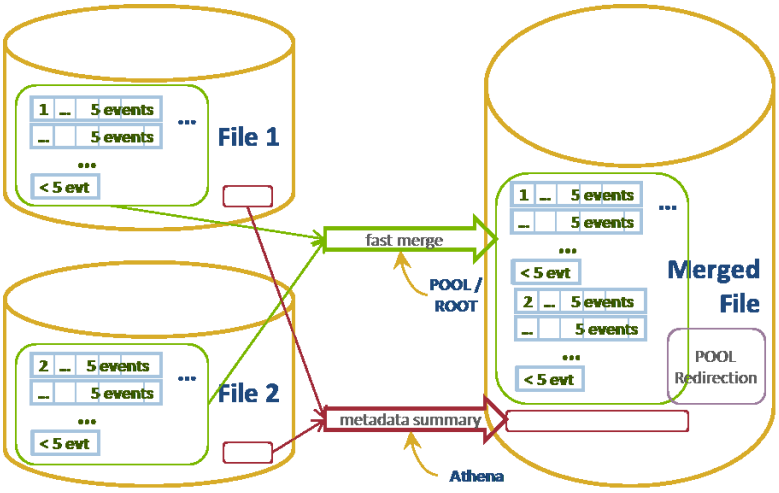


Figure 5. The ATLAS fast merge utility appends event data without decompressing the baskets and uses the Athena framework to summarize metadata.

5. Scatter / Gather architecture for multicore I/O

ATLAS considers implementing a scatter / gather like architecture. A shared reader process would handle the input for all workers and output would be handled by a common writer process avoiding the need to merge temporary output files (see figure 6).

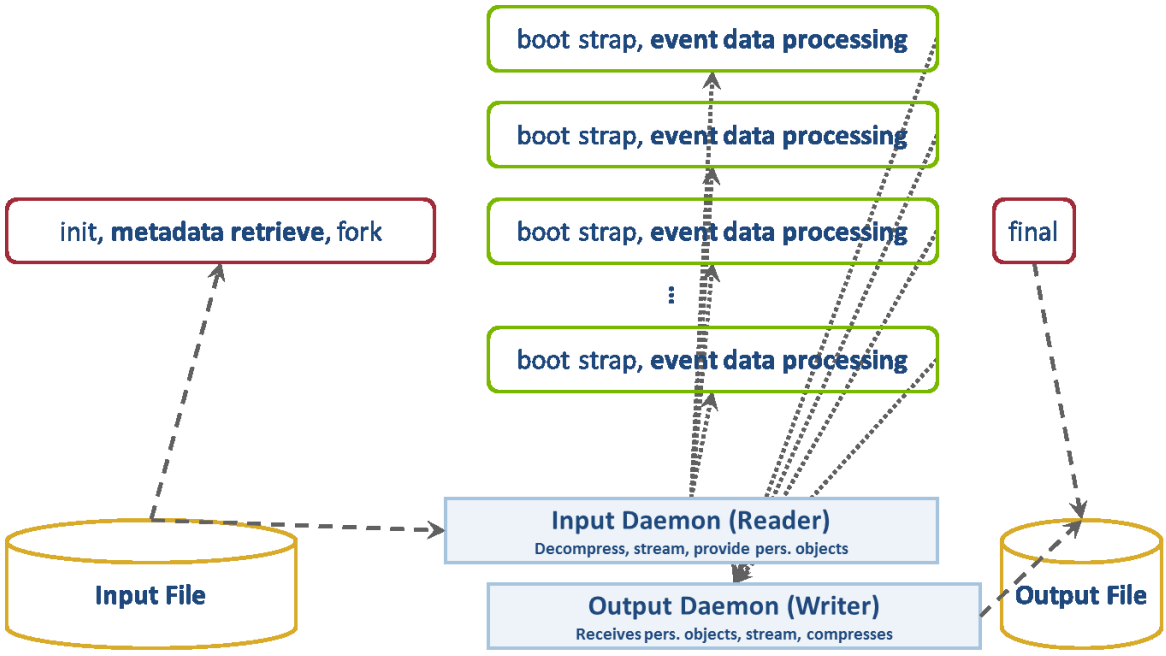


Figure 6. Scatter / Gather architecture for AthenaMP. The event worker will obtain input event data (either the complete event or single collections) from the reader. The writer will collect the output data from all workers and write them to a common output file.

5.1. Shared reader strategies: ByteStream data

For Raw data, providing the next event is rather straightforward as all event data is contiguously stored in a single block which is read entirely. At the same time, the benefit of a single reader for ByteStream is small as Raw data events can be decompressed individually and the handicaps described in section 4.1 therefore do not apply.

A simple shared ByteStream reader for AthenaMP has been developed (see figure 7). In this first implementation, a separate reader process is forked by the master process. Even though this is a clone of the master, re-initialization will cause the EventSelector to iterate over events and serve the data in a shared memory segment. The worker processes are forked and the re-initialization of their EventSelector will make them clients, reading event data from shared memory rather than the input files.

Communication between the reader and the workers is done via a special control structure in shared memory. This structure holds the event and file sequence numbers, the memory location, size of the event data block and the processing status of the event. The processing status indicates, whether an event has been completely read by the reader, requested by a worker, or copied to a worker.

The master process schedules the worker to process events by using the event sequence number (in order). In client mode, the worker process will request that event from the reader, rather than seeking to it in the file, and wait until the data is available. The reader will pre-fetch event data, so that the worker wait may be shorter than the time needed for reading an event itself. As the worker will retrieve the event data at the very beginning of event processing, event requests to the reader will tend to be in order and with processing times that are much longer than reading times; sequential read is efficient even with just one pre-fetched event.

The shared reader is being tested, at this time no significant performance differences have been found as long as the single reader isn't serving too many (more than about 8) workers. This limitation can be resolved by allowing the reader to pre-fetch events or configuring more than one shared reader process.

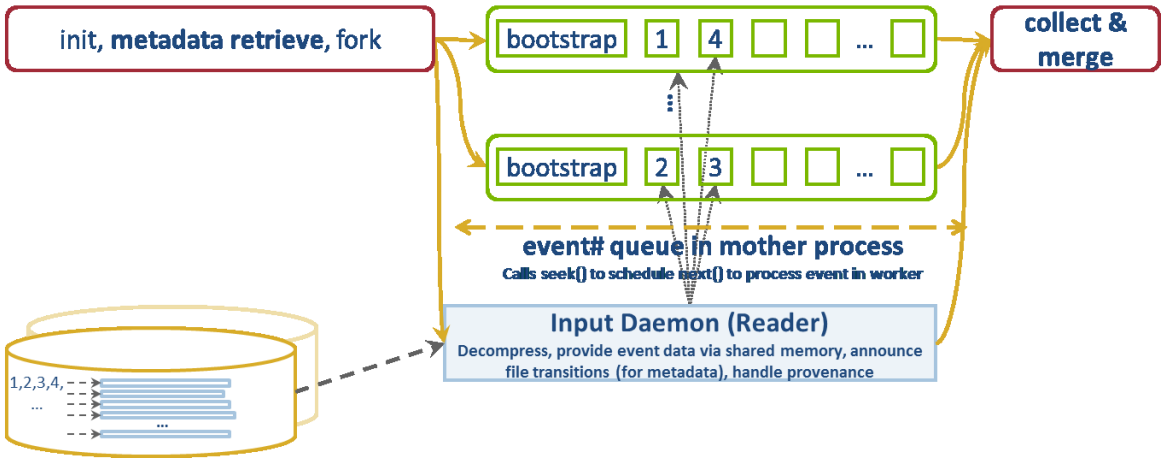


Figure 7. Shared ByteStream data reader. The shared reader process is launched by the master. It will read and decompress events from the input file and provide them to the worker processes. The reader will also inform the workers of file transitions so that in-file metadata from the input files can be propagated to the worker.

5.2. Challenges for shared ByteStream reader
Some required functionality made implementing the shared reader approach for ByteStream data somewhat challenging and may be more difficult for POOL / ROOT data.

5.2.1. Metadata propagation. As for multicore processing in general, metadata poses one of the key challenges for the shared ByteStream reader. In this architecture the reader is the only component that detects file transitions and will need to inform worker to fire file incidents which control in-file metadata propagation. Each worker will then fire the End File Incident after processing its last event from the previous file and the Begin File Incident before processing the next event. Input metadata needs to be provided to all worker processes. For Raw data, ATLAS uses in-file metadata which is very limited in content and structure and can be communicated via shared memory.

5.2.2. Provenance. When reading Raw data, the ByteStream reader records the input file identifier and the event offset of each event into a provenance record (i.e., DataHeader) which gets propagated to downstream data products. This record needs to be shared between reader and workers in addition to the event data.

5.2.3. Event status information. Some event status information, such as DAQ status, is not part of the event data, but discovered by the reader. This data is encoded and transmitted to the worker process via shared memory as part of the control structure.

5.3. Shared reader strategies: POOL / ROOT data
For simulated, reconstructed or derived data, there may be performance penalties to event scatter strategies such as shown in section 5.1 for ByteStream, as POOL / ROOT data is accessed for each collection of data objects on demand and not all are needed for each job. As ATLAS uses a flat data store and transient/persistent separation, there currently is no concept of a persistent event. Instead, individual collections of data objects are read.

However, as ATLAS stores several hundreds of collections per event, scattering these becomes more complicated, requiring marshalling of data from many I/O buffers and increased communication between the reader process and the workers. ATLAS could use the on demand retrieval mechanism of the existing Athena I/O framework to send a request to the reader, which then would provide the collection. To improve performance, advanced strategies, such as learning what collections are needed, so the reader could pre-fetch them, would need to be developed.

As noted in section 3.2, transient objects in simulation and reconstruction are much more complex than Raw data events and may not be shareable via shared memory. As another benefit of the transient/persistent separation layer, ATLAS could pass persistent objects through shared memory. This also has the advantage that the transient/persistent conversion would be done by the worker processes in parallel.

5.4. Alternative reader strategies for POOL / ROOT data
Since 2011, ATLAS uses the ROOT auto-flush feature for their main event data TTree to write clusters of 5 / 10 events depending on data product. At the same time, to avoid baskets getting too small for efficient I/O, splitting was switched off for all but the largest collections. So all the data of a collection is streamed member-wise into a single TBranch. Therefore, compressed baskets contain data from only 5 / 10 subsequent events, which is small enough to have the entire cluster of events is processed by the same worker, without significantly degrading load balance.

With such a change, most of the performance disadvantages from section 4.1 would be solved. Only some of the inefficiencies will remain due to reading of auxiliary TTrees that are not synchronized with event numbers and these TTrees store less than 5% of the event data volume. In first, very preliminary test, ATLAS found that it can be expected to speed up reading by 20 – 50% in

CPU time. In addition memory is better utilized as there are no duplicated buffers on different workers.

Implementing the strategy is not without complications: Fast merged files (as described in section 4.2 and shown in figure 5) will destroy event cluster organization. The last basket in each (pre-merged) file may be incomplete and fast merge only appends baskets without modifying their content. Some additional metadata describing the event clustering and infrastructure to query it is needed to allow worker processes to retrieve a cluster of events.

5.5. Writing

Output strategies pose similarly subtle problems, with complications that may lead to significant serialization and the possibility of serial bottlenecks, either during writing or in post-processing, e.g., during data stream merging. The fast merge utility has allowed ATLAS to avoid the most severe performance degradation. However there are other drawbacks of the multi-writer strategy of AthenaMP.

Fast merge for event data obtains a speed up because it appends existing I/O buffers, rather than decompressing them and re-compressing them into new baskets. These time savings come at a cost, if the previously written baskets were small and poorly optimized. This may increase the file size due to bad compression and may cost CPU and user time for jobs reading the data. Poor output optimization can happen if the worker processes only small number of events compared to the maximum entries for a basket. In the old, fully split data layout, a small 2KB basket would hold up to 500 entries and would often not be filled by a single worker process. Since switching to the new persistent layout, most collections are streamed member-wise into a single basket, which is flushed every 5 / 10 events, so that most baskets should be well optimized (i.e., are the same in Athena and AthenaMP).

5.5.1. TMemFile. ROOT also prepares for multicore I/O and is introducing new features that could help ATLAS software. One of these newer components is TMemFile, the ROOT way of merging several TTrees to the same output TTree, without having to write them to disk. With the migration away from POOL and potentially closer to ROOT, ATLAS may decide to leverage these features directly. However, there are still open questions, such as how to support external tokens to event objects, on which the ATLAS navigational infrastructure relies. Currently theses tokens use the file identification and the TTree entry number, neither one will be valid if writing to a memory resident file. Also, ATLAS uses multiple event data TTrees, which would need to remain synchronized and of course metadata objects require summarization not just appending.

6. Conclusion

A multi-process control framework, such as the current implementation of AthenaMP, to enable HEP event processing on multicore and eventually manycore computing architectures is an important step, but it is only one of many steps that need to be accomplished. Optimizing event data storage, so that it can be efficiently retrieved in the granularity needed by the multiple worker processes is key to avoiding performance penalties during reading. The 2011 change to member-wise streaming with a small number of entries per basket will help ATLAS to tackle inefficiencies in multi-process reading of ROOT data. The data layout also must be designed and implemented to ensure both that data produced by multiple processes can be efficiently combined, as merging is typically done serially, and that the resulting output is as efficient to read and store as if it had been produced by a single process. By focusing the data access granularity on event level (rather than simply file level), the recent changes to the storage layout will be beneficial for multicore processing.

Even with fully optimized data storage; there is a scale of parallelism that cannot be adequately supported using essentially serial I/O components. ATLAS is in the process of developing an I/O architecture that can efficiently support even higher numbers of parallel worker processes than are used in current implementations. These developments include complex components to scatter / gather event data. First prototypes are being tested, but much work remains.

References

[1] Tsulaia A et al 2011 Multicore in Production: Advantages and Limits of the Multi- Process Approach in the ATLAS Experiment *14th International Workshop on Advanced Computing and Analysis Techniques in Physics Research* to be published

[2] van Gemmeren P and Malon D 2009 The event data store and I/O framework for the ATLAS experiment at the Large Hadron Collider *IEEE International Conference on Cluster Computing* doi: 10.1109/CLUSTR.2009.5289147

[3] van Gemmeren P and Malon D 2011 Persistent Data Layout and Infrastructure for Efficient Selective Retrieval of Event Data in ATLAS *ePrint* arXiv:1109.3119v1

[4] Brun R and Rademakers F 1996 ROOT: An Object Oriented Data Analysis Framework *AIHENP'96 Workshop* Nucl. Inst. & Meth. In Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch

Acknowledgments

Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, under contract DE-AC02-06CH11357.