Development of a Next Generation Concurrent Framework for the ATLAS Experiment

P. Calafiura[1], W. Lampl[2], C. Leggett[1], D. Malon[3], G. Stewart[4], B. Wynne[4]

[1]Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley CA 94720, USA
[2] CERN
[3]Argonne National Laboratory, 9700 S. Cass Ave, Argonne, IL 60439, USA
[4] Glasgow

E-mail: `cgleggett@lbl.gov`

Abstract.
The ATLAS experiment has successfully used its Gaudi/Athena software framework for data taking and analysis during the first LHC run, with billions of events successfully processed. However, the design of Gaudi/Athena dates from early 2000 and the software and the physics code has been written using a single threaded, serial design. This programming model has increasing difficulty in exploiting the potential of current CPUs, which offer their best performance only through taking full advantage of multiple cores and wide vector registers. Future CPU evolution will intensify this trend, with core counts increasing and memory per core falling. With current memory consumption for 64 bit ATLAS reconstruction in a high luminosity environment approaching 4GB, it will become impossible to fully occupy all cores in a machine without exhausting available memory. However, since maximizing performance per watt will be a key metric, a mechanism must be found to use all cores as efficiently as possible.

In this paper we report on our progress with a practical demonstration of the use of multi-threading in the ATLAS reconstruction software, using the GaudiHive framework. We have expanded support to Calorimeter, Inner Detector, and Tracking code, discussing what changes were necessary in order to allow the serially designed ATLAS code to run, both to the framework and to the tools and algorithms used. We report on both the performance gains, and what general lessons were learned about the code patterns that had been employed in the software and which patterns were identified as particularly problematic for multi-threading. We also present our findings on implementing a hybrid multi-threaded / multi-process framework, to take advantage of the strengths of each type of concurrency, while avoiding some of their corresponding limitations.

1. Introduction

Due to thermal power density limitations, clock speeds of silicon processors have plateaued since 2005 (see Figure 1). To compensate for this, manufacturers have increased both the core count and number of hardware threads of processors (see Figure 2). Meanwhile, memory prices, which had been falling at an exponential rate for almost 40 years, have leveled out, and are holding steady at $8/GB (see Figure 3).

Current ATLAS reconstruction jobs require more than 3GB of physical memory to run without swapping. This requirement will only grow as the event multiplicity increases in LHC Run 3. Combining all these numbers results in an estimate of the cost to fully populate a current

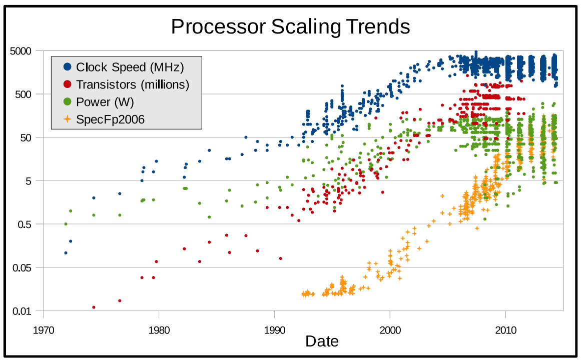

Figure 1. Processor scaling trends since 1970

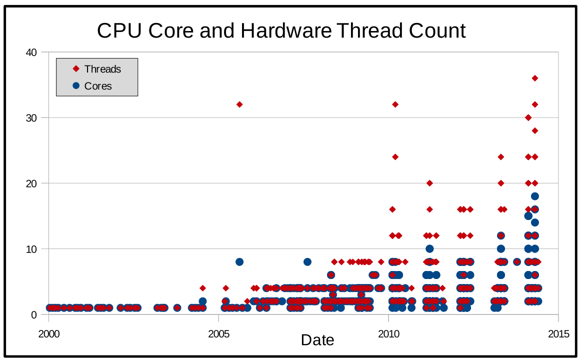

Figure 2. Historical evolution of CPU core counts and hardware threads

high end compute node with 4 CPUs and 18 processor cores per CPU, with enough memory to run one reconstruction job per hardware thread, we arrive at a cost of over \$ 4000 (18 cores/processor x 2 threads/core x 4 processor/node x \$8 / GB). Extending this estimate to the grid, where ATLAS uses over 160000 cores, would require more than \$5 million. Furthermore, this is an estimate for current hardware - core counts to be significantly higher in Run 3.

ATLAS already has a concurrent framework, called AthenaMP [1], which makes use of copy-on-write semantics for page sharing between forked workers. It offers significant memory savings over serial jobs, but as core counts continue to rise, and the ratio of installed memory to cores falls, even this will be insufficient in the long term, and alternate concurrency and memory saving solutions must be explored.

GaudiHive [2] is the extension to Gaudi[3] that enables concurrency via multi-threading, and since Athena, ATLAS's reconstruction and simulation framework, is based on Gaudi, this was an obvious starting point in investigating multi-threaded frameworks. The key features of GaudiHive are:

- automatic scheduling of Algorithms by the framework based on the appearance of their required Data Objects in the event store
- Algorithms execute in their own threads, from a shared thread pool
- pipelining of both Algorithms and events, such that as many Algorithms as there are threads available can be executed simultaneously, and these Algorithms can be executing in the context of different events.
- Algorithms can be cloned, creating multiple instances of the same object, so that the same Algorithm can be executed simultaneously in the context of different events. Cloning is not obligatory, permitting the balancing of memory usage and performance.

2. Simulation of Reconstruction

ATLAS reconstruction was simulated by extracting the Algorithm data dependencies and run times from a regular ttbar reconstruction job. A directed graph of the data dependencies is shown in Figure 4. Though not evident in this depiction, an analysis of this graph shows the possibility of upwards of 8 parallel data flows. A CPU Cruncher Algorithm was implemented, which mimicked CPU usage of a real Algorithm using the extracted values. This configuration was run through GaudiHive, with various tuning parameters:

- number of concurrent events
- size of the thread pool

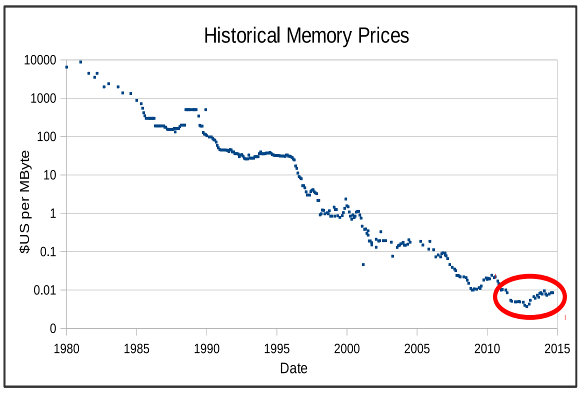

Figure 3. Historical Memory Prices. Exponential decrease for 40 years until 2005.

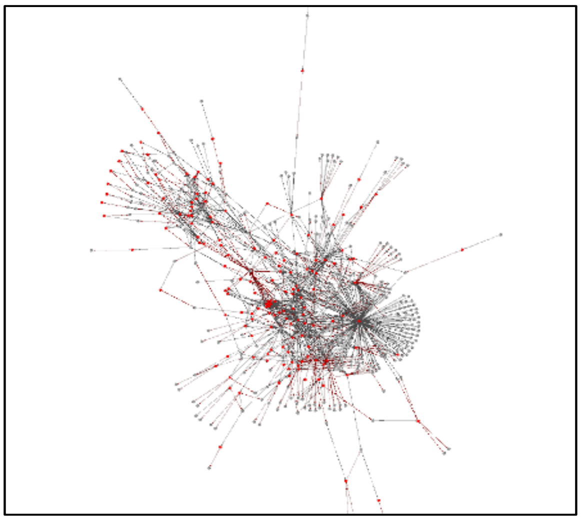

Figure 4. Algorithmic data dependencies for ATLAS reconstruction job.

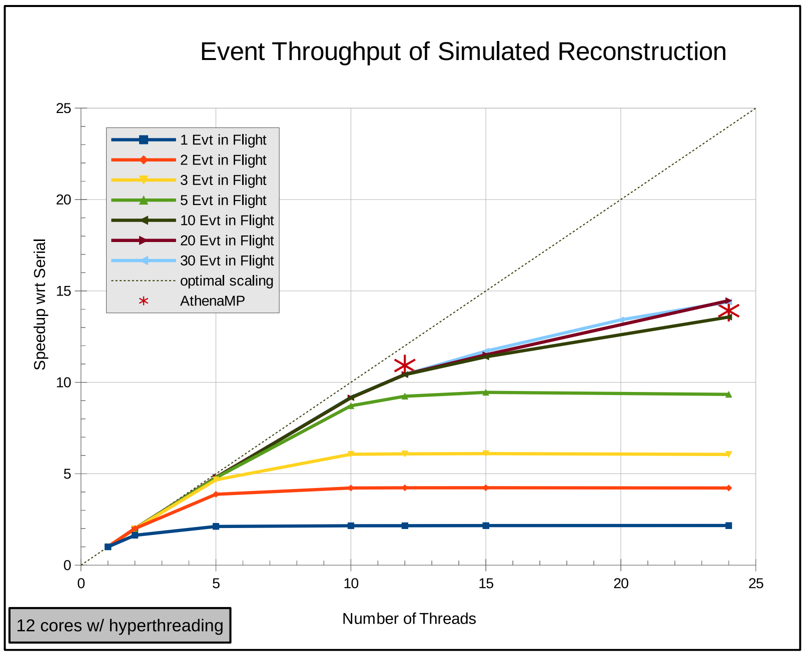

Figure 5. Performance of simulated reconstruction as a function of thread count and concurrent events.

- total number of simultaneously executing Algorithms
- Algorithm cloning

The results show good scaling with the number of threads up to the total number of physical cores on the test node (see Figure 5). Note that turbo boost was disabled on this node. Best performance was achieved with approximately 10 concurrent events. Increasing the concurrent event count beyond this point did not increase performance. Also note that this is a best case scenario, as this is a toy simulation, with no memory access effects, blocking, I/O, or other real world considerations.

The effect of limiting the cloning of various Algorithms was also investigated. We discovered that disabling cloning of all but the seven slowest Algorithms (those with average event run times of longer than 1s), had only a small effect on the total event throughput - total performance diminished by only 1.2% with this configuration. This is very important, as it tells us that we only need to make a very small number of Algorithms capable of having multiple instances run concurrently in different threads - if an Algorithm is not cloned, then it will never run simultaneously in more than one thread, and thus there is no risk of contention of shared data. Furthermore, as memory usage grows with the number of cloned Algorithms, limiting the number can have significant implications on total memory usage.

3. Real Detector Testbeds

Once the performance benefits of multi-threading had been demonstrated at the toy level, the next step was to use real detector reconstruction code and real data in a multi-threaded environment. This was essential in order to understand what needed to be modified in both user and framework code in order to function with GaudiHive, as well as to measure the memory usage and performance for real code. A job configuration with a small subset of reconstruction Algorithms were chosen for two different sub-detector testbeds, one for the Liquid Argon Calorimeter, with 5 reconstruction Algorithms and 16 data objects, and one for the Inner Detector, with 7 Algorithms and 19 data objects.

We discovered several incompatibilities which had to be addressed before we could even run with more than a single thread, such as

- rewriting the event loop manager
- ensuring Algorithms did not modify data after registration with the Event Store
- pre-loading of event data at the beginning of the event instead of lazy loading via proxies
- understanding the sources of cyclical data dependencies

Once these issues had been addressed, and we were running with more than one thread, and more than one concurrent event, we immediately encountered more serious issues, such as

- use of memory pools
- event related asynchronous callbacks (Incidents) being processed by the wrong event
- Algorithms, Tools and Services[1] that were caching data between events
- Services and Tools that were not thread safe
- I/O is not thread safe

These issues were addressed either by fixing user code, modifying the framework, or serializing the affected section of code (by locking mutexes around code blocks, or disabling cloning of an entire Algorithm). Given the small number of Algorithms in the job configuration, which limited concurrency, the best performance for the Calorimeter testbed was with 6 concurrent events. A profile of the memory consumption of the job as a function of time is shown in Figure 6, and relative memory and performance metrics in Table 1. We can understand these results in two ways, either by comparing to a single non-turbo boosted serial job, where we see that running with GaudiHive gives us a 401% increase in performance, with an associated increase in memory consumption of 38%. Alternatively we can compare with 6 serial jobs running simultaneously, in which case the Hive scenario has 67% of the event throughput, but only consumes 23% of the memory. In either case, we see significant benefits in terms of memory consumption when running in a multi-threaded environment.

[1] See [3] for a description of these components.

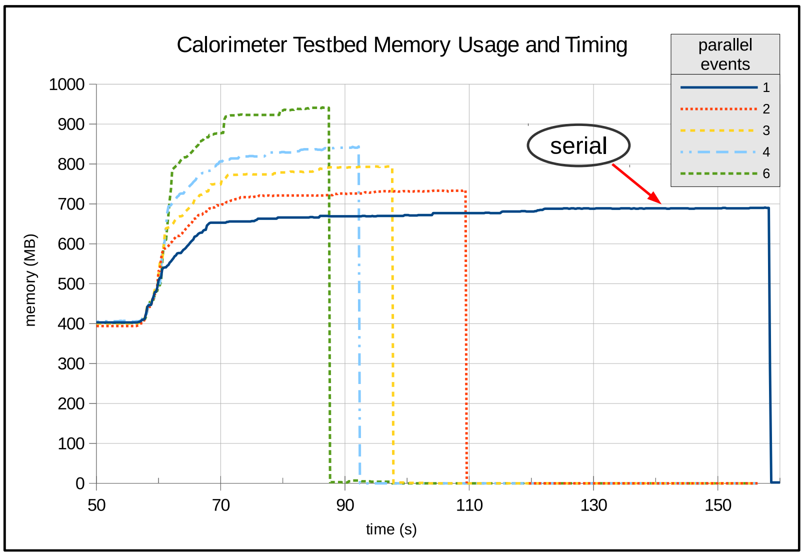

Figure 6. LAr Calorimeter testbed memory consumption as a function of time.

Table 1. Normalized Calorimeter Testbed Performance.

parallel events	speedup *wrt* serial	speedup *wrt* n*serial	memory ratio to serial	memory ratio to n*serial
1	1.00	1.00	1.00	1.00
2	2.07	1.04	1.06	0.53
3	2.80	0.93	1.15	0.38
4	3.33	0.83	1.22	0.31
6	4.01	0.67	1.36	0.23

Table 2. Calorimeter Testbed Performance for Hybrid Framework.

	time /s	memory /MB
4 workers, 1 thread	32.8	1847
2 workers, 2 threads	34.4	1241
1 worker, 4 threads	53.6	935
pure hive, 4 threads	47.8	817

Though a great deal was learned during the process of making the Inner Detector testbed function in the Gaudi Hive environment, it had considerably less potential parallelism than the Calorimeter testbed, due to shared Tools that could not be made thread safe, and so its performance tests are not presented here.

Having to modify both user level code, as well as the framework, led us to a very important question - to what extent could we avoid modifying user code by finding work arounds at the framework level? For instance if a Service or Tool was caching data between events, could we have one instance of these components per concurrent event, instead of the present use of a singleton pattern? After investigating a number of possibilities, we determined that to a significant extent this was indeed possible, but it would result in either reduced performance due to forced serialization of components, or increased memory usage. In effect, it would negate any of the advantages of running in a multi-threaded environment.

4. Hybrid Multi-Process / Multi-Threaded Framework

Since ATLAS already has a multi-process enabled framework in production, an interesting avenue to explore was to run multiple threads within each was AthenaMP worker, as combining multi-process and multi-threading should permit the fine tuning of the total memory usage. Implementing this framework required some significant changes to the event loop manager, and to the mechanism by which AthenaMP distributed events to the worker processes, but proved feasible. Two modes of operation were envisioned:

- Run only one concurrent event in each worker, but allow parallel execution of different Algorithms in different threads. This would limit the risks of thread-unsafe code, but would only benefit configurations with several reconstruction pathways and large number of Algorithms
- Allow full concurrency of both events and Algorithms in each worker.

This hybrid framework was tested with the same LAr Calorimeter testbed configuration as was used for our initial performance tests. The various configurations of running with 4 concurrent events are as follows, with the results shown in Table 2:

- 4 AthenaMP workers, 1 event/thread in each worker
- 1 AthenaMP worker, 4 events/threads in each worker
- 2 AthenaMP workers, 2 events/threads in each worker
- pure GaudiHive with 4 events/threads

As expected, 4 AthenaMP workers each with 1 thread is the fastest and consumes the most memory, 1 worker with 4 concurrent threads/events is the slowest, but consumes the least memory, and a configuration with 2 workers, each with 2 concurrent threads/events falls between the two, in terms of both memory and speed. We expect 4 AthenaMP workers to be faster than one worker with 4 threads, as we cannot achieve perfect multi-threaded concurrency with the LAr Calorimeter testbed, as some Algorithms could not be cloned due to thread safety issues, which causes the execution in some threads to block until a resource has been freed in another. Similarly, we expect a pure GaudiHive environment to use less memory and be slightly faster than an AthenaMP job with one worker and an equivalent number of threads due to the overhead of the mother process in the AthenaMP job, and also due to the inefficiency of the mechanism by which the events are distributed to the workers via the shared queue.

Event throughput *vs* memory consumption, normalized to a serial job, is shown in Figure 6, for increasing numbers of AthenaMP workers, and increasing number of concurrent events within each worker. A 16 physical core Xeon node with hyperthreading enabled was used for these runs. This plot shows that given certain memory requirements, we can choose a job configuration specifying a certain number of AthenaMP workers, and concurrent threads within each worker, that makes optimal use of the available processors. It should be noted that if the thread count within each worker is increased beyond a certain maximum, such that $n_{workers} \times n_{threads} > n_{cores} - \epsilon$, where ϵ is usually 1 or 2, then performance decreases, as overhead from the large number of concurrent threads overwhelms the system. ϵ is non-zero, as there is further overhead from the mother process of an AthenaMP job, and also from the controller threads of a GaudiHive process.

5. Geant4 Simulation

GaudiHive shows its advantages over AthenaMP in environments with high core counts. We expect it to shine in systems such as supercomputers, where the core counts can reach the tens of thousands if not more. An obvious candidate job for such an environment is event simulation, especially since this accounts for more than half of ATLAS's CPU budget, and we have begun implementing an ATLAS Geant4 simulation job in GaudiHive, leveraging on the thread safety of the newly released Geant4 v10.

Geant4 v10 can already do event level concurrency via multi-threading, but in order to make use of standard ATLAS simulation Algorithms, we need to take control of the event loop manager from Geant4, and use the Athena Hive version.

However, there are several significant barriers in utilizing the multi-threaded implementation of Geant4. Namely, v10 has changed the mechanism by which user classes are initialized, separating the thread local and global aspects via the so-called "split classes". This has resulted

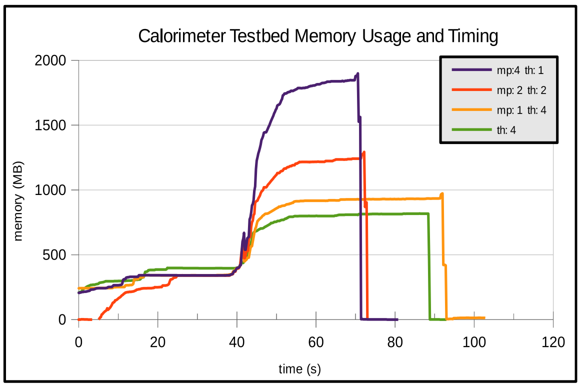

Figure 7. Memory consumption profile of hybrid LAr Calorimeter testbed.

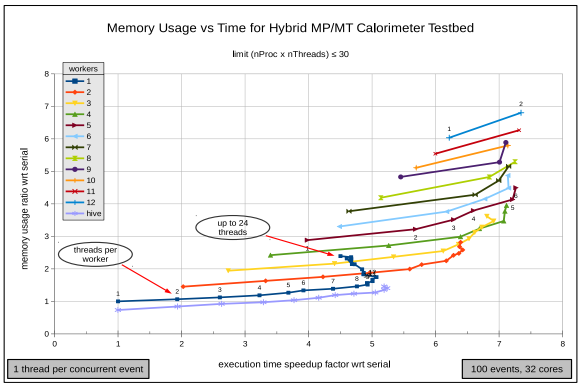

Figure 8. Memory consumption of LAr Calorimeter testbed for various AthenaMP workers and thread counts.

in having to rewrite a substantial section of ATLAS G4 sensitive detector code, which we are in the process of doing.

Currently we have a functional prototype that performs with multiple concurrent threads, and our goal is to test production simulation apps by the end of the year.

6. Conclusions

Our investigations have shown that for ATLAS, a multi-threaded framework can offer significant memory savings over both regular Athena, as well as AthenaMP. Furthermore, by choosing the appropriate configuration of parallel workers and threads, we can optimize utilization of CPU and memory resources on a compute node in a way that would not be possible with plain event level parallelism. However, a multi-threaded framework is not a drop-in solution. There are several issues that must be surmounted, both at the user class level, and at a more fundamental Athena level. If we attempt to side-step thread safety issues with user code by modifying the framework instead, we will reduce the performance benefits of multi-threading, and increase memory consumption.

Much user code can survive unscathed however. Since Algorithms that don't need to be cloned do not need to be thread safe, we can choose where to focus our energies in making only the most expensive Algorithms thread safe. Algorithms will need to be able to communicate their level of thread safety to the framework, so that they can be scheduled accordingly.

Services and public Tools on the other hand will need to be made thread safe, and ensure that they do not cache data between events. This is likely to be non-trivial, and will require substantial intervention and rewriting of code.

The vast majority of these changes are expected to be backward compatible, and will likely make current serial code better and more efficient.

Thus we envision an evolutionary progression towards a multi-threaded compatible framework, rather than a revolutionary one.

References

[1] Binet S et al. 2012 Multicore in production: Advantages and limits of the multiprocess approach in the ATLAS experiment *J. Phys.: Conf. Series* **368** 012018 (ACAT2011 proceedings)
[2] M. Clemencic, B. Hegner, P. Mato and D. Piparo, J. Phys. Conf. Ser. **513**, 022013 (2014).
[3] G. Barrand *et al.*, Comput. Phys. Commun. **140**, 45 (2001).