17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09) IOP Publishing
Journal of Physics: Conference Series **219** (2010) 042002 doi:10.1088/1742-6596/219/4/042002

Harnessing multicores: strategies and implementations in ATLAS

S Binet[1], P Calafiura[2], S Snyder[3], W Wiedenmann[4] and F Winklmeier[5]

[1] LAL/IN2P3, France
[2] LBNL, USA
[3] BNL, USA
[4] University of Wisconsin, USA
[5] CERN, Switzerland

E-mail: binet@cern.ch

Abstract. Computers are no longer getting faster: instead, they are growing more and more CPUs, each of which is no faster than the previous generation. This increase in the number of cores evidently calls for more parallelism in HENP software. If end-users' stand-alone analysis applications are relatively easy to modify, LHC experiments frameworks, being mostly written with a single 'thread' of execution in mind and consequent code bases, are on the other hand more challenging to parallelize. Widespread and inconsiderate changes so close to data taking are out of the equation: we need clear strategies and guidelines to reap the benefits out of the multicore/manycore era while minimizing the code changes.

1. Introduction

The *"Free Lunch"* is over: Moore's law [1] does not hold anymore, computer scientists and software writers have now to be familiar with Amdahl's law [2]. Indeed, computers are no longer getting faster: instead, they are growing more and more CPUs, each of which is no faster than the previous generation.

This increase in the number of cores evidently calls for more parallelism in HENP software. Fortunately, typical HENP applications (event reconstruction, event selection,...) are usually *embarrassingly parallel*, at least at the coarse-grained level: one "just" needs to parallelize the event loop. However, if end-users' stand-alone analysis applications are relatively easy to modify, LHC experiments frameworks, being consequent code bases and mostly written with a single thread of execution in mind, are on the other hand more challenging to parallelize. Widespread and inconsiderate changes so close to data taking are out of the equation: we need clear strategies and guidelines to reap the benefits out of the multicore/manycore era while minimizing the code changes.

Exploiting parallelism is traditionally achieved via *a)* multithreading or *b)* multiple processes (or a combination of both) but each option has its own set of trade-offs in terms of code changes (and code complication) and possible speed-ups.

This paper describes the different strategies the Offline and Online communities from the ATLAS collaboration have investigated. These strategies have been implemented and integrated into the Athena framework. A description of this integration and how well the

© 2010 IOP Publishing Ltd

strategies perform in terms of speed-ups, memory usage, I/O and code development will be discussed.

We first present the work integrated in AthenaMT to harness the High Level Trigger farms' computing power via multithreading, the needed improvements and modifications applied to Athena/Gaudi [3] in order to raise its thread awareness and the impact on common design patterns to preserve thread safety across release cycles. Threads sharing the same address space, AthenaMT is the most promising option in terms of speed-ups and memory usage efficiency at the cost of an increased development load for the code writer needing to worry about locks, data races and the like.

Section 2 then describes AthenaMP which leverages the fork() system call and the *Copy On Write* (*COW*) mechanism through the multiprocessing python module [4], to free oneself from the usual concurrency problems that stem from sharing state, while still retaining part of the memory usage efficiency at the cost of a diminished flexibility (compared to threading.) We'll detail the AthenaMP implementation, highlighting the minimized code changes, how they seamlessly blend into the Athena framework and their interplay with I/O and OS resources.

2. Experience with multithreading: AthenaMT

Using threads is a well known technique to increase applications computing throughput. A thread can be described as a set of independent instructions which can be run and scheduled by the operating system. This independent flow of control is accomplished thanks to the ability of threads to keep their own stack pointer, registers, signals and other specific data. threads within a single executable share all these resources and are traditionally called *lightweight processes* because of the small context switch time incurred to "jump" from one thread of execution to the other. Sharing resources fits well efficiency but, being automatic, it introduces subtle bugs when *e.g.* shared data is modified concurrently by multiple threads, leaving the program in an inconsistent state. Thus, programmers have to explicitly identify and mark specific portions of code as critical and carefully control their access via locks and mutexes.

In the following section we describe the modifications applied to the ATLAS software offline framework to leverage the Trigger Data Acquisition (TDAQ) computing farms in the context of the Trigger Level 2 Processing Unit (L2PU.)

2.1. Thread safety in Athena/Gaudi

Since the event selection software executes in multiple worker threads as shown in Figure 1, the framework must provide a thread-safe environment. At the same time, and in order to provide an easy-to-use framework for offline developers, the framework must hide all technical details of thread handling and locks. Thread safety has been implemented by using Gaudi's name-based object and service bookkeeping system. Copies of components that need to be thread-safe are created for each worker thread with different labels. The labels incorporate the thread-id of the worker thread, as obtained from the data flow software: TriggerSteering/TrigStr__0, of the form <ComponentType>/<ComponentInstanceName>__<ID>.

However, not all Athena components need to be thread-specific: entities like the DetectorStore or the GeometrySvc serving *read only* data may safely be shared by all threads. This is easily configurable via the usual Athena facilities during the job submission. The number of threads created by the data flow software is transferred to the scheduler, which transparently creates the number of required copies. In this scheme, the same configuration can be used in the offline and in the L2PU environments. In single threaded environments, the thread-id collapses to null as it is not needed.

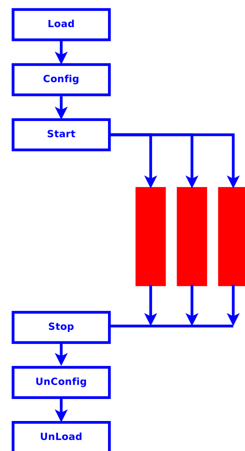

Figure 1. Basic state machine of a L2 processing unit: the workload is parallelized at the event loop level after the start step. Workers are joined at the stop state.

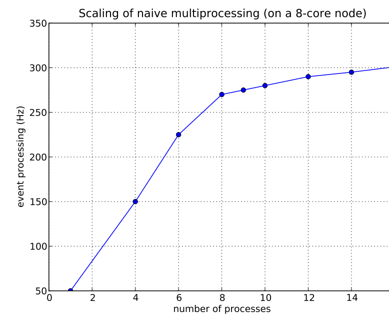

Figure 2. Computing throughput of an unmodified Athena application on an 8-core machine. Scaling is quite good up to the point where the number of spawned processes matches the number of cores.

2.2. Deployment and performances
During the tests, it was observed that the event throughput in a L2PU didn't scale in the expected way with regard to the number of worker-threads. This was due to the use of a common memory pool for container objects the default memory allocation scheme of the *Standard Template Library* (STL) in production at the time (gcc-2.95.) The event processing model of L2PU favors a scheme where every thread allocates its own memory pool. Such an allocation scheme is available in the STL. After carefully optimizing the code with it, the expected scaling behaviour was observed. Such an optimization poses limitations on the offline components used and their external dependencies. For instance, external utility libraries may not be available with this allocation scheme. To avoid frequent memory allocation during event processing, all large containers that hold detector data were designed to allocate memory only during initialization time and to reset their data for each new event.

The use of offline components in multiple worker threads and the requirement to avoid *Locks* in the event selection code limits certain design implement choices, *e.g.* the use of *Singletons*. These restrictions were communicated to the HLT developers at an early stage.

Nevertheless, preserving thread-safety and optimized code over release cycles and in a large heterogeneous community are very time consuming tasks. Developers have to be familiar with thread programming and debugging. This usually means some special and additional training of them is required. Synchronization problems for multi-threaded code are notoriously tedious to debug. Moreover, event selection code typically changes rapidly due to physics needs and thus needs to be constantly re-optimized for a multi-threaded environment. To ease the development burden, an emulator of the L2PU multi-threaded environment has been provided for the offline software developers: AthenaMT.

3. Experience with multiprocessing: AthenaMP
3.1. Process-based parallelism
Parallelization using multiple processes is fairly easy to achieve. In its simplest and naive incarnation, it is implemented by merely spawning n instances of the application at hand with

a priori no required code changes. We do observe a good scaling behaviour with respect to the number of cores, as shown in Figure 2.

The obvious problem with this approach is the suboptimal usage of resources as nothing - except for the dynamic shared objects (DSO) - is shared between all these multiple processes. This leads to increased resource requirements, scaling with the number of concurrent process' instances, which are bound to eventually hit hardware limits and severely hamper scaling. Memory is a scarce resource in typical reconstruction jobs and is usually the first to be depleted but other *Operating System* resources need to also be considered such as *e.g.* file descriptors and network sockets whose maximal allowed number is hard-coded in the kernel.

3.2. *fork and* COW *based multiprocessing*

One way out of this memory and resources' duplications problems is to use the fork(3) system call to share as much memory as possible. Upon invocation, fork clones a process, including its entire address space. This sounds as counter-productive for the problem at hand, but modern operating systems (such as GNU/Linux) use the *Copy On Write* mechanism as an efficient optimization. The memory of both the parent (which is calling fork) and the child (which is being created) processes is actually shared up to the point of when one of these two processes writes to it. At that point, the memory page will be copied and the affected changes will become "unshared". Memory-wise, the optimal strategy is of course to fork as late as possible but *before* any output is written to not invalidate the file descriptors. The memory which has been modified will become unshared but all the static configuration data will remain shared between both processes. As processes have different address spaces, no user code needs to be changed to serialize access to critical section of the code, only core framework code needs to for *e.g.* manage the I/O related and event scheduling parts. Also, relying on fork will ensure that as much memory as possible will be shared and automatically done so, by the GNU/Linux kernel.

However, the main problem with this approach is that once a memory page has been unshared, it can not be re-shared and may lead to a suboptimal memory footprint, as illustrated in Figure 3 where 2 sub-processes may end-up with no memory for the conditions data being shared even if they do use the same conditions data. This may not be a real issue in the context of carefully planned reconstruction where events with alike conditions data will be scheduled in bundles, but "chaotic analyses" mode will trigger such an inefficiency.

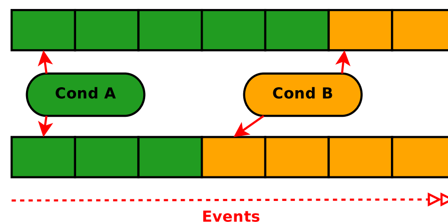

Figure 3. The re-sharing problem. Two sub-processes initially sharing memory for a set A of conditions data may end-up sharing nothing as the initial set of conditions data is replaced by a different set B. Even if the data is the same, the memory pages have been modified and triggered the *COW* mechanism.

3.3. *AthenaMP implementation*

One of the constraints of parallelizing Athena is to avoid clients' code changes: AthenaMP fulfils this condition by hiding the multiprocess semantics inside the Athena/Gaudi

framework instead of publishing them as a layer atop. This has been achieved as only 3 core-framework classes needed to be modified to perform additional operations, mainly related to I/O subsystem (re-)initialization. The bulk of the other changes being limited to writing new code to handle and manage the multiprocess aware event loop.

This has been implemented thanks to the python multiprocessing[1] module. This package allows to fork sub-processes and greatly eases their management by abstracting the fork and *Inter-process communication* (*IPC*) mechanisms away from its users. Besides the re-sharing problem which is specific to our goal of minimizing the memory footprint of our software (and its implementation via *COW*), the traditional biggest issue with multiprocess-based parallelism is state sharing. As the different workers are located in different processes, state is communicated between them via *IPC*: serializing and sending objects over the wire via pipes or sockets. This does incur non negligible overhead.

Relying on the python multiprocessing module completely side-stepped the coding burden of implementing the *IPC* layer and synchronization logic. This will probably ease the implementation of more sophisticated event scheduling mechanisms (such as work-stealing algorithms) in the future, even if a simple round-robin event scheduler is currently deployed.

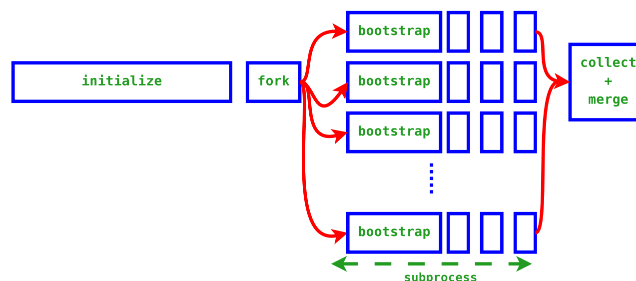

Figure 4. Schematic view of the AthenaMP event scheduling. The main process follows the usual Athena sequence up to the end of the initialize() stage and then forks off n sub-processes whose results are then collected and merged together.

A new Athena event loop manager has been implemented in python, wrapping the original one with "MP" semantics, as shown in Figure 5. The number of workers can either be specified at the command line level or automatically inferred from the number of cores present in the hardware. A pool of forked sub-processes is created up-front just before entering the event loop. These sub-processes are given a bootstrap function to correctly handle I/O reinitialization (*e.g.* reopen file descriptors) and finally asynchronously scheduled to process all the input events in a round-robin fashion: they delegate the actual work of managing the event loop to the usual manager, adequately configured for our needs.

Eventually, when no more events are to be processed and the finalize() stage is reached, the control is given back to the main process which will take care of merging all the output files (if any) created by the workers. This is the most problematic part of AthenaMP as POOL, the persistency layer ATLAS is using, does not provide any API for automatically merging files. Another issue stemming from technical details related to Athena's persistency strategy[2] prevented us from using the hadd utility provided by ROOT [5]. The current implementation is thus to have a complete Athena instance doing the merging by reading in all these files to create a single one.

[1] Athena is using python-2.5 so we had to back-port the module (available for 2.6) to our environment.

[2] Athena makes the distinction between transient and persistent representations of objects, allowing for optimized disk-size and I/O performances.

```
class MpEventLoopMgr (PyAthena.Svc):
    def executeRun (self, maxevt):
        """Process `maxevt` events as Run (beginRun->endRun)"""
        if self._ncpus <= 0:
            return self._evtloop_mgr.executeRun(maxevt)

        import multiprocessing as mp
        info ("nbr of workers: %i", self._ncpus)
        info ("master workdir: %s", self._wkdir)
        wrks = mp.Pool(processes=self._ncpus,
                       initializer=self._worker_bootstrap)
        results = wrks.map_async(func=batch_run,
                                 iterable=(maxevt,)*self._ncpus)
```

Figure 5. Excerpt of the python based `multiprocessing` event loop manager. The main process first creates a pool of workers, passing them a bootstrap method to handle `I/O` initialization, and then asynchronously waits for the result of the run over the events.

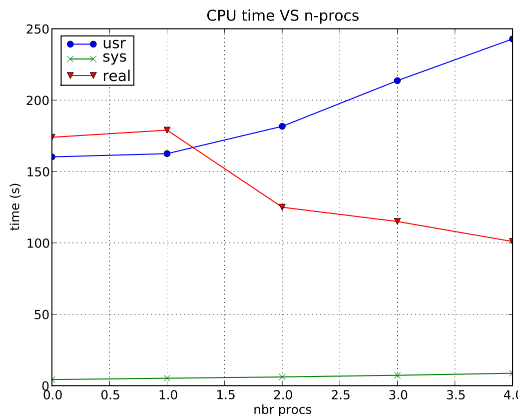

Figure 6. Evolution of user, system and real time for reconstructing a 50 $t\bar{t}$ events file versus the number of forked sub-processes. The increase of user time with the number of processes is due to additional bootstrapping (loading of `I/O` dictionaries) and post-processing (file merging.)

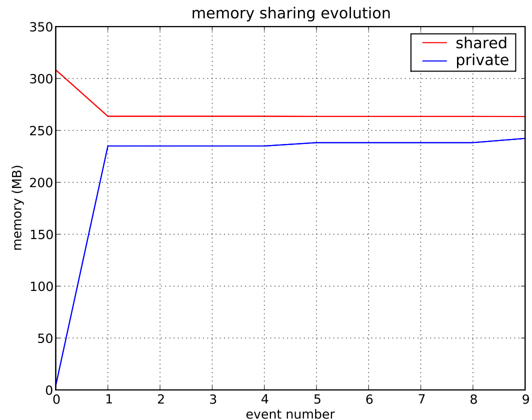

Figure 7. Evolution of the amount of memory shared between sub-processes against time, obtained from `pmap(1)` for a $\sim$ 700 MB `VMem` application.

3.4. Performances

Figure 6 shows the CPU timings for a reconstruction job over a 50 events file on an LXPLUS node with 4 cores, in 5 configurations: the unmodified `Athena` ($nprocs = 0$) and `AthenaMP` with 1, 2, 3 and 4 sub-processes.

While the additional work stemming from bootstrapping the sub-processes and merging the output files into a single one is clearly shown by the increase of needed user time when the

number of sub-processes increase, real-time decreases as expected.

Figure 7 shows the evolution of the amount of memory shared between sub-processes against time, before the event loop (event number 0) and after. The drop in memory being shared and the increase of memory being private to each process at the first event is due to the loading of conditions data callbacks as well as the loading of `Reflex` dictionaries needed to read/write objects from/to files. Work is underway to move this late initialization out of the main event loop.

4. Conclusions

Due to the large code basis written as single threaded applications and the data-taking phase being so close, wide-spread and inconsiderate changes to `Athena` are not wise. From the authors' experience, it is probably best in the near future to exploit the multiprocess approach for multi-core machines.

Indeed, even if multi-threading may offer bigger performance gains, it is more difficult to correctly implement, especially on such a large code basis written over a long time. The multiprocess approach, coupled with `fork(3)` and *COW* seems the less error-prone option - thanks to separated address spaces preventing implicit sharing of data and enforcing explicit sharing via *IPC* - while minimizing the impact on clients' code. Moreover, error handling and error recovery is easier as one failing worker process won't tear all the others down or - worst - silently corrupt the memory of the others.

Subtle issues such as random numbers and seeds' reproducibility are still open and would need a more detailed treatment. Another interesting problem to solve will be the efficient submission of multiprocess jobs on the GRID whose current working model is to allocate one job to one computing node. Indeed, chaotically harvesting the computing power of the GRID nodes would result in counter-productive overbooking of resources. A possible solution would be to leverage the already existing `MPI` [6] job submission infrastructure to prevent such a catastrophic scenario. However, the first issue to be solved for multiprocess-based applications to use the GRID, will be the GRID virtual memory monitoring tools which double-count the memory shared by forked sub-processes.

In the course of implementing `AthenaMT` and `AthenaMP`, the authors have felt the need for more general support libraries for typical recurring code in `HENP` applications on multi-core machines:

- accessing and managing "logical" file descriptors shared or split among different processes and/or threads,
- high level "functions" to manage shared memory,
- support for parallel `I/O`.

Having such a toolkit ready for general usage will help to address the `I/O` bottleneck we are bound to face with the current simple-minded strategy, in the upcoming *many*-core era.

References

[1] Moore E, "Cramming more components onto integrated circuits", Electronics Magazine, 1965
[2] Amdahl G, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967
[3] Mato P 1998 `Gaudi`-architecture design document Tech. Rep. LHCb-98-064 Geneva
[4] The `multiprocessing` module, `http://docs.python.org/library/multiprocessing.html`
[5] The `ROOT` framework, `http://root.cern.ch`
[6] MPI: A Message Passing Interface Standard, `http://www.mpi-forum.org`