Towards a distributed heterogeneous task scheduler for the ATLAS offline software framework [*]

Paolo Calafiura[1,], *Julien* Esseiva[1,], *Xiangyang* Ju[1,], *Charles* Leggett[1,], *Beojan* Stanislaus[1,], and *Vakho* Tsulaia[1,]

[1]Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, USA

Abstract. With the increased data volumes expected to be delivered by the HL-LHC, it becomes critical for the ATLAS experiment to maximize the utilization of available computing resources ranging from conventional GRID clusters to supercomputers and cloud computing platforms. To run its data processing applications on these resources, the ATLAS software framework must be capable of efficiently executing data processing tasks in heterogeneous distributed computing environments. Today, using the Gaudi Avalanche Scheduler, whose implementation is based on Intel TBB, we can efficiently schedule Athena algorithms to multiple threads within a single compute node. We aim to develop a new framework scheduler capable of supporting distributed heterogeneous environments, based on technologies like HPX or Ray. After the initial evaluation phase of these technologies, we began the development of a prototype distributed task scheduler for the Athena framework. This contribution describes this prototype scheduler and the preliminary results of performance studies within ATLAS data processing applications.

1 Introduction

The ATLAS [1] experiment operates one of the detectors at the LHC at CERN. To carry out physics analysis, a lot of data is generated, both by the ATLAS detector as well as Monte Carlo simulations.

Athena [2] is the software framework used by ATLAS to process data. It is built on Gaudi [3], a cross-experiment data processing framework for HEP. ATLAS currently uses either a multi-process (AthenaMP) or multithreaded (AthenaMT) version of Athena depending on the task to handle parallelism within a node.

The majority of ATLAS computing resources are provided by the Worldwide LHC computing grid (WLCG). HEP is a high-throughput computing domain, that is, we care about how many decoupled tasks can be processed over a long period. The WLCG fulfills that need very well by providing access to many independent resources. PanDA [4] is the ATLAS distributed management system, responsible for assigning jobs to different sites on the grid.

Recently, High-performance computing (HPC) resources have been integrated with AT-LAS. With the significant paradigm shift - HPC is optimized for short, parallel, and coupled jobs requiring low communication latency - this has proven challenging. Some of these challenges include

---

- Load balancing the work across nodes as HPCs expect multi-node jobs and Athena is designed to run on a single node.

- Utilization of GPUs and other accelerators, where the majority of the FLOPS of the newest generation of HPCs comes from.

As an intermediate step, we have implemented Raythena [5] a task farm managing multiple AthenaMP processes within a single HPC allocation. This layer of scheduling requires a lot of boilerplate components to stay compatible with the WLCG. In addition, Athena still being a single-node application does not allow for a fine-tuned control of work scheduling across nodes. We, therefore, started exploring different frameworks for implementing a distributed version of Athena. The initial results were presented at ACAT2022 [6].

We present here the prototype implementation of Gaudi with extended distributed capabilities using HPX, in Section 2 and 3, Gaudi and HPX are briefly presented to give the necessary understanding of the distributed Gaudi-HPX implementation presented in 4. We then present scaling obtained with the current implementation, discussing potential bottlenecks and the next step towards a production version of a distributed version of Athena.
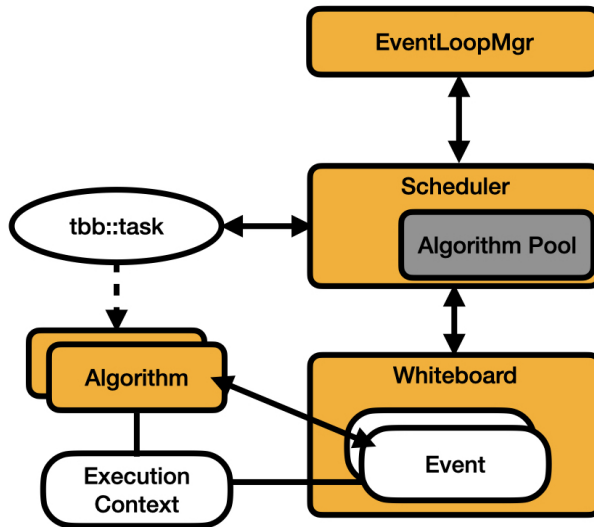
2 Gaudi

Gaudi [3] is a cross-experiment event processing framework on which Athena is built. Figure 1 illustrates the relationship between components. A core component of Gaudi is the `Algorithm`, responsible for transforming data. Processing one event will require the execution of many algorithms. Algorithms read and write data in a shared event store, the whiteboard. They declare their dependencies by having read / write key handles to the event store as class data members. The scheduler maintains a pool of `Algorithm` instances and several events in flight. It is responsible for solving the dependency graph of algorithms and scheduling them onto cores using Intel Thread Building Blocks (TBB) [7]. Usually, a single instance of a given algorithm exists and the event to process is passed as a parameter to the `execute` function of the algorithm. Algorithms can be declared as either *reentrant*, in which case many threads can call the algorithm concurrently, or *cloneable*, allowing different instances to be scheduled on different threads. The event loop manager is responsible for pushing events to the scheduler.

3 HPX

HPX [8] is a C++ library for concurrency and parallelism. It presents an API conforming to the C++ standard, implementing parts of the standard library such as parallel algorithms and execution policies, control objects such as futures, latch, barriers, or future proposals such as Concurrency TS and P2300. More importantly, HPX extends these APIs with asynchronous and distributed implementations, allowing library users to run their applications in both a distributed and a multi-threaded environment in the same way. HPX follows a cooperative multithreading model; a future scheduled on the HPX threading subsystem will execute until it finishes or yields control by synchronizing with other tasks. It also integrates with GPUs, allowing execution and synchronization between CPU and GPU code.

The distributed features of HPX use either TCP, MPI, or libfabric as the networking backend. Serialization of data between nodes, both native and user-defined types, is supported following the Boost serialization API. HPX also provides a global address space supporting distributed objects called **components**. Remote procedures are called **actions** and can be either a free function or a member function on an HPX component. HPX provides different semantics for applying actions such as fire-and-forget, synchronous, or asynchronous calls.

Figure 1. Gaudi architecture and interaction between different components.

HPX also defines **localities** which are remote processes. When applying an action or instantiating a component, a locality identifier is needed to specify where the procedure should be executed.

4 Gaudi-HPX Integration

Our HPX integration within Gaudi tries to keep the changes to a minimum [9]. Figure 2 illustrates the changes in architecture. They can be summarized in two main changes:

- Replace the TBB task arena with an HPX arena using fire-and-forget futures.

- Implement actions as free functions. They are remotely called by locality 0's `EventLoopMgr` to push events to schedulers on remote processes.

Unless we bind threads to specific cores or NUMA domains, TBB and HPX do not play well with each other as they would be competing for resources. To minimize the changes required in Gaudi, we swapped out the TBB task arena with a thread arena implementation using HPX as the backend, exposing the same API as TBB's task arena. The Gaudi scheduler queues a future representing the algorithm to execute within the HPX thread pool using `hpx::apply`, i.e. fire-and-forget semantics. HPX is responsible for scheduling the algorithm for execution on hardware resources. The Gaudi scheduler is still responsible for resolving algorithm dependencies; it will only offload algorithms that are ready to be executed and have their dependencies resolved. From HPX's point of view, algorithms have no dependencies; they can be scheduled for execution as soon as received. Events are still processed within a single locality. Once an event ID is assigned to a given locality by locality 0's distributed `EventLoopMgr`, the local scheduler will only schedule work within the same process. Changing this behavior would require significant architectural changes because of the whiteboard communication model.

The `EventLoopMgr` has been extended to work in distributed environments. The implementation follows a controller/worker architecture. One process will be started per node or

NUMA domain, the locality 0 will take the role of the controller while all the other instances will be workers as illustrated in Figure 2. The only difference between the controller and workers is the work done by the `EventLoopMgr`. The worker's `EventLoopMgr` is disabled, it doesn't push events to the local scheduler. However, it still exists in memory to keep the application alive.

The controller's `EventLoopMgr` retrieves the processes within the cluster using HPX's API. Then, instead of pushing events to the local scheduler, it does remote procedure calls (RPC); calling the HPX action (free functions) to schedule all the events on workers. The only information exchanged is an event index; each process reads event data from files. Once all the events are scheduled, it queues calls to drain each scheduler, i.e., waiting for all events to complete. Finally, it notifies each worker that the work is done, resuming the disabled event loop on each worker that will simply return without doing anything, causing workers to terminate

The action to dispatch the event executes on the worker. It has a reference to the local, disabled, `EventLoopMgr`, which is used to push the event to the scheduler as the `EventLoopMgr` would normally do in the non-distributed scenario. One caveat is that the scheduler is not reentrant; to make sure that actions are not concurrently pushing events to the scheduler, execution on each worker is serialized by a mutex.

Figure 3 illustrates in more detail how events are distributed to workers. Each scheduler has a given number of event slots, i.e., the number of events in flight. The distributed event loop will iterate over the workers and try to schedule an event. The action on the worker will check if the scheduler has an event slot available. If it does, it schedules the event and returns. If no event slots are available, it will asynchronously schedule a task to drain the scheduler to clear up slots of finished events and return. In cases where the event can not be scheduled on a worker, the main event loop will try to schedule it on the next worker until it finds an empty slot. The remote call within the event loop to schedule the event on a worker is synchronous, however, all the work happening within that call (on the worker) is asynchronous so it is fast but still incurs network latency on which the event loop has to wait before moving to the next event.
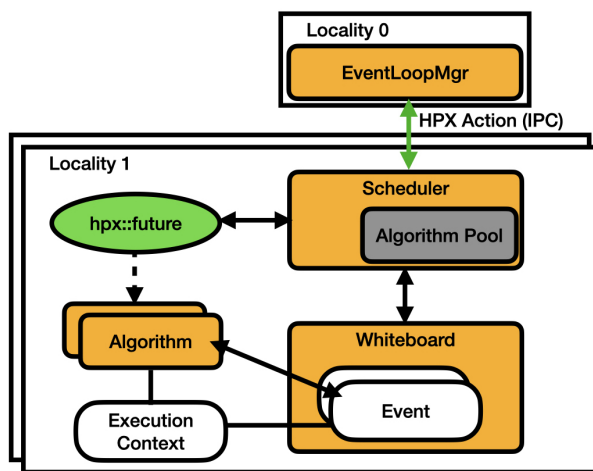
Figure 2. Modifications to the Gaudi architecture for HPX integration.

Inter-process event scheduling
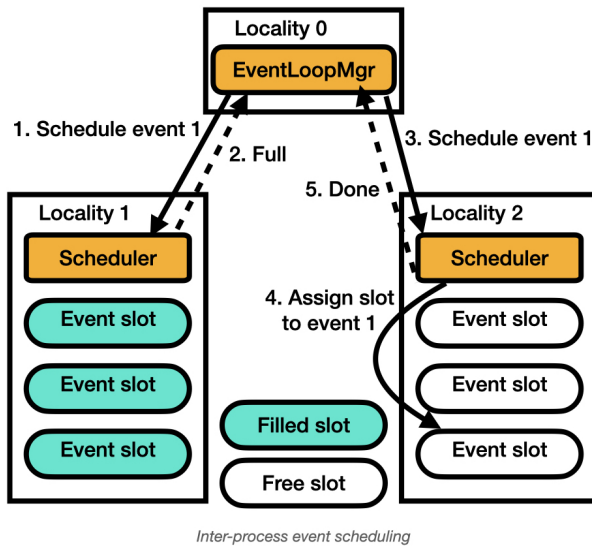
Figure 3. Events scheduling policy between controller and workers.

5 Results

We measured the weak scaling of the HPX-Gaudi implementation. The problem size is configured to be 3200 events per node, with a minimum of 12800 events. Measurements were done on the Cori KNL partition using 128 cores per node, and HPX was configured to use TCP as the networking backend. Events were configured to take around 15 s of processing time and are purely CPU-bound, i.e. they do not require file system I/O.

Figure 4 shows that the throughput does not scale beyond 10 nodes. We demonstrated in [6] that HPX was linearly scaling up to at least 30 nodes for much shorter tasks of hundreds of milliseconds. Knowing the average time of the tasks, the number of processors available, and the scheduling policy presented in Section 4, we can easily estimate that, for the breaking points of 10 nodes, one iteration of the event loop in the controller process can spend approximately 10 ms scheduling an event on a node. If we are not providing work to cores within that time constraint, some will sit idle.

We measured that it takes the event loop 30 ms on average to schedule an event. We identified two main contributors to this latency. The first is that we are using TCP instead of the recommended high-speed MPI network. The second is that HPX uses the same thread pool for processing algorithms offloaded by the local scheduler and executing actions called by the main event loop which pushes events to the scheduler. This means that if a node is full it can block the main event loop trying to schedule an event on that node since the call to schedule events is synchronous leading to the potential starvation of other nodes.

6 Conclusion and Future Work

Our initial standalone evaluation of HPX showed a promising outcome for HPX, we were able to work around the identified challenges. The integration with Gaudi revealed new bottlenecks. It would require a major redesign of the Gaudi scheduler to solve them. The challenge is not only technical but also organizational; Gaudi is used by experiments other
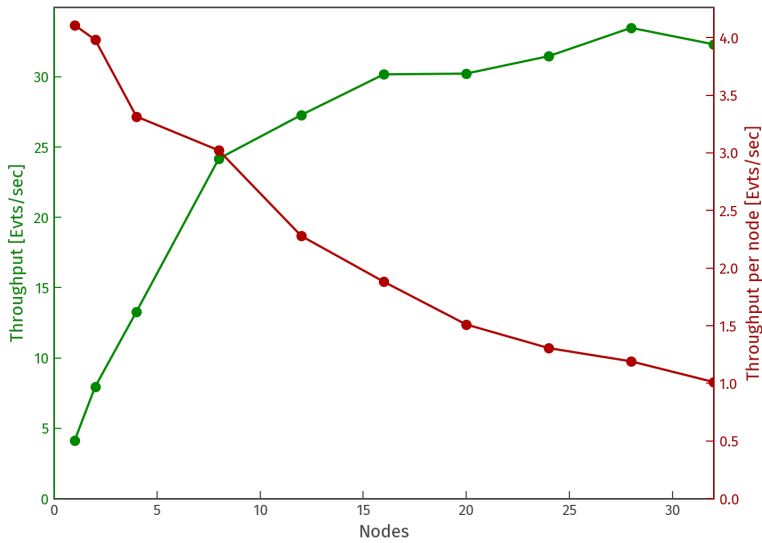
Figure 4. Event throughput measured on Cori KNL nodes.

than ATLAS. While this might be inevitable as a distributed scheduler and a local scheduler are very different, we still want to consider and have started work on, a pure MPI implementation. The MPI implementation follows a pull architecture; workers request events from the controller's event loop instead of having them pushed. The initial performance measurements are promising and show perfect scaling up to 40 nodes so far.

References

[1] ATLAS Collaboration 2008 The ATLAS Experiment at the CERN Large Hadron Collider J. Inst. **3** S08003, 10.1088/1748-0221/3/08/S08003

[2] ATLAS Collaboration. (2019). Athena (22.0.1). Zenodo. 10.5281/zenodo.2641997

[3] G. Barrand et al., GAUDI — A software architecture and framework for building HEP data processing applications, 2001, url: https://gitlab.cern.ch/gaudi/Gaudi

[4] De K et al. 2015 The future of PanDA in ATLAS distributed computing J. Phys. Conf. Ser. **664** 062035, 10.1088/1742-6596/664/6/062035

[5] Miha Muškinja, Paolo Calafiura, Charles Leggett, Illya Shapoval, Vakho Tsulaia Raythena: a vertically integrated scheduler for ATLAS applications on heterogeneous distributed resources EPJ Web Conf. **245** 05042 (2020), 10.1051/epjconf/202024505042

[6] Paolo Calafiura, Julien Esseiva, Xiangyang Ju, Charles Leggett, Beojan Stanislaus, Vakho Tsulaia, Next generation task scheduler for ATLAS software framework, url: https://indico.cern.ch/event/1106990/contributions/4991224

[7] Intel Threading Building Blocks, url: https://github.com/oneapi-src/oneTBB

[8] Kaiser et al., (2020). HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software, **5(53)**, 2352, 10.21105/joss.02352

[9] HPXGaudi implementation, url: https://gitlab.cern.ch/hpxgaudi/Gaudi