Improving Throughput of a Julia-based Event-Processing Demonstrator

Justin Kowalski[1], Mateusz Fila[2], Andre Sailer[2]

[1]ETH Zürich, [2]CERN

Summer Student Report, 29.08.2025

Abstract

This report presents the development and optimization of a Julia-based demonstrator for High Energy Physics (HEP) event-processing application frameworks. Julia's potential for parallel computing in HEP is evaluated. Previously, the demonstrator relied on a package causing significant performance and stability issues. This has been revised by using Julia's built-in threading capabilities [1]. This improved reliability and performance, however, benchmarks show that the Julia implementation still lags behind C++ alternatives [2], particularly when memory allocations are involved. We suggest that Julia's garbage collection and memory management present challenges for multi-threaded HEP applications.

Introduction

High Energy Physics (HEP) experiments require event-processing frameworks to handle the vast amounts of data produced by detectors due to particle collisions we refer to as events. These frameworks follow a common pattern: loading event data, applying transformations and filters, and writing the output for each event. The computational demands of these workflows necessitate efficient parallel processing across multiple threads and potentially distributed systems.

This project investigated Julia as an alternative to traditional C++ implementations for HEP event-processing frameworks such as Gaudi [3]. Julia promises high performance with more accessible syntax and built-in parallel computing capabilities, potentially reducing development complexity while maintaining computational efficiency. We developed a demonstrator that simulates realistic HEP workflows using extracted data-flow graphs from production systems, particularly focusing on the ATLAS q449 reconstruction test job containing approximately 800 algorithms and 3800 data objects.

Implementation

Workflow Extraction and Representation

The demonstrator processes data flow graphs represented as directed acyclic graphs (DAGs), which means that it has a clear direction and no cycles (see Fig. 1). Vertices represent algorithms (transformations) or data objects, and the edges indicate dependencies. We extracted workflows from production systems including ATLAS standard reconstruction jobs, capturing the graph structure, control-flow logic, and algorithm execution timings. Note that the algorithm execution times are used to simulate a mockup algorithm that searches for prime numbers serving as a CPU crunching.

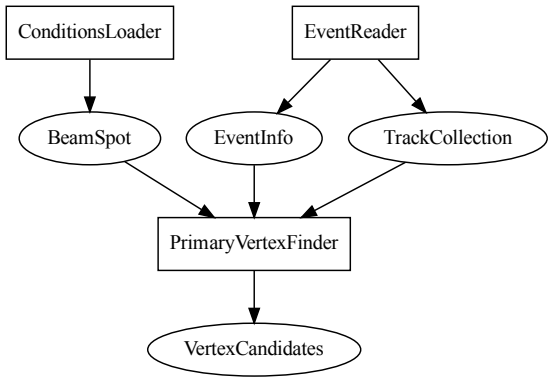

Figure 1: *Example data-flow graph (DAG) of the workflow. The rectangular shapes are algorithms and oval shapes are data objects.*

Initial Approach with Dagger.jl

We initially selected Dagger.jl (version 0.18.17) as the scheduling backend. Dagger promises unified parallel computing on CPUs, GPUs, and distributed systems [4], which makes it attractive for HEP applications. The framework is under active development at MIT. However, the benchmarks revealed severe limitations, as can be seen in Figure 2. The Julia demonstrator is compared to another demonstrator written in C++ and using a Taskflow-based scheduler [5], which is a modern replacement for oneTBB

[6], which is the parallel processing library used in current event processing frameworks such as Gaudi.

- Throughput scaling collapsed beyond 20 threads
- The system exhibited data races causing crashes above 35 threads
- Performance degraded to approximately 1.5 events/s at 30 threads, compared to Taskflow's 4.5 events/s
- Profiling overhead reached 40% with Dagger's logging enabled

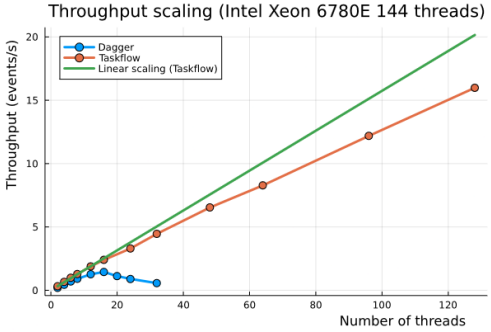

Figure 2: *Throughput scaling comparison between Dagger.jl and Taskflow implementations showing severe performance issues in Dagger beyond 20 threads.*

Transition to Julia Threads

Given Dagger's limitations, we decided to use Julia's standard library threading options. While sacrificing multi-processing support, this approach provides:

- Improved stability with no crashes at high thread counts
- Better scaling up to 144 threads
- Reduced overhead with Nvidia Nsight systems profiling [7] using the NVTX.jl binding [8] (negligible vs 40% with Dagger)
- More predictable performance characteristics

Performance Analysis

Threading Performance

Using the ATLAS q449 workflow with 20 events per thread and no allocations within algorithms, the Threads implementation achieved approximately 10 events/s at 144 threads as shown in Figure 3. While this represents a significant improvement over Dagger, it remains below the C++ Taskflow implementation's 20 events/s linear scaling.

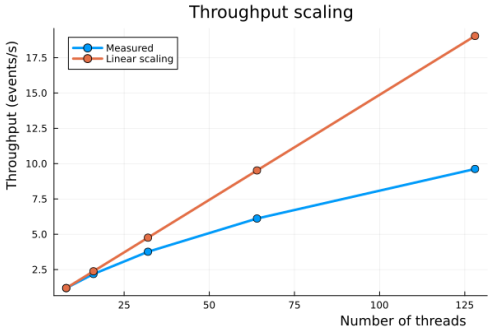

Figure 3: *Throughput scaling with Julia Threads showing improved but sub-linear performance compared to Taskflow's linear scaling.*

Impact of Memory Allocation

Real HEP algorithms frequently allocate memory for temporary calculations and data structures. Adding realistic allocations (10kB per algorithm) as shown in Figure 4 dramatically impacted performance:

- Throughput dropped from 10 events/s to 3.5 events/s at 144 threads
- Scaling efficiency decreased significantly beyond 20 threads
- Garbage collection (GC) pauses became a dominant factor

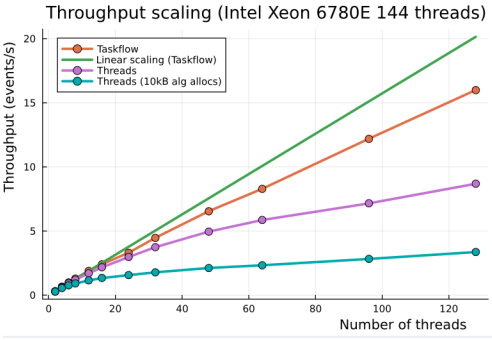

Figure 4: *Throughput scaling with Julia Threads for 10 kB of allocations in the algorithms.*

Experimentation with GC configuration (`-gcthreads` parameter) provided modest improvements as depicted in Figure 5. Setting `-gcthreads=1,1` helped, but fundamental GC limitations persisted. As described, if we start to allocate in algorithms, we run into problems.

Scheduling Overhead

Profiling with NVIDIA Nsight Systems revealed that scheduling overhead exceeded algorithm execution time for the ATLAS q449 workflow. This is particularly problematic given the workflow's characteristics:

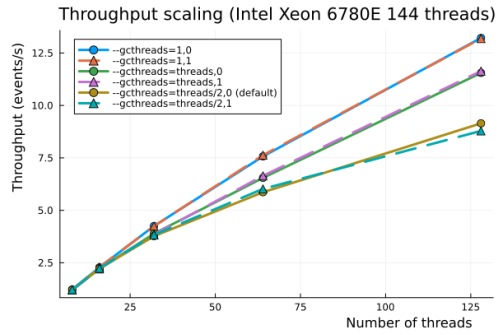

Figure 5: *Throughput scaling with Julia Threads for different* `-gcthreads` *settings.*

- Algorithm execution times span six orders of magnitude (10^{-6} to 1 second)
- Many algorithms execute in microseconds
- The scheduler struggles with fine-grained parallelism

Discussion

Our findings highlight several challenges for Julia in HEP applications:

Garbage Collection: Julia's GC significantly impacts multi-threaded performance, especially with frequent allocations typical in HEP workflows. Unlike C++ with manual memory management or custom allocators, Julia offers limited control over memory allocation patterns.

Scheduling Efficiency: Both Dagger and native threading exhibit higher overhead than specialized C++ solutions like Taskflow. For workflows with numerous short-duration algorithms, this overhead becomes prohibitive.

Performance Gap: Despite Julia's promise of C-like performance, our benchmarks consistently show a 2-5x performance gap compared to C++ implementations, although staying under realistic conditions with memory allocations.

Conclusions

While Julia offers attractive features for scientific computing, our evaluation reveals significant challenges for high-throughput, multi-threaded HEP event processing. The demonstrator successfully processes realistic workflows but with performance penalties that are unacceptable for production systems. The modular approach using dedicated packages for specific parallelization strategies appears to be more viable than integrated solutions like Dagger for current Julia versions. However, C++ remains the more suitable choice for performance-critical HEP frameworks.

Acknowledgments

This work was supported by the CERN Strategic Programme on Technologies for Future Experiments and partially funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project (grant SIF-2023-004). Computing resources were provided by CERN OpenLab.

Thank you to Mateusz Fila and Andre Sailer for their supervision, help and interesting discussions this summer. Also, thank you to the Summer Student Program for such an enriching experience.

References

[1] Mateusz Jakub Fila et al. *FrameworkDemo.jl*. URL: https://github.com/key4hep/key4hep-julia-fwk.

[2] Mateusz Jakub Fila. *taskflow-fwk: Demonstrator for event-processing application framework using Taskflow*. https://github.com/m-fila/taskflow-fwk. Accessed: 2025-08-28. 2025.

[3] G. Barrand et al. "GAUDI — A software architecture and framework for building HEP data processing applications". In: *Computer Physics Communications* 140.1 (2001). CHEP2000, pp. 45–55. ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(01)00254-5. URL: https://www.sciencedirect.com/science/article/pii/S0010465501002545.

[4] Julian Samaroo et al. *Dagger.jl*. Version 0.18.12. June 2024. URL: https://github.com/JuliaParallel/Dagger.jl.

[5] Tsung-Wei Huang et al. "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System". In: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (2022), pp. 1303–1320. DOI: 10.1109/TPDS.2021.3104255.

[6] Intel Corporation. *oneAPI Threading Building Blocks (oneTBB)*. https://github.com/uxlfoundation/oneTBB. Accessed: 2025-08-29. 2021.

[7] NVIDIA Corporation. *NVIDIA Nsight Systems – system-wide performance analysis tool*. https://developer.nvidia.com/nsight-systems. Accessed: 2025-08-28.

[8] Simon Byrne, Tim Besard, et al. *NVTX.jl: Julia bindings for NVTX (NVIDIA Tools Extension)*. https://github.com/JuliaGPU/NVTX.jl. Accessed: 2025-08-28. 2025.