

Demonstrator for HEP event-processing framework in Julia: performance perspective

Mateusz Fila, Graeme A Stewart (CERN)

Justin Kowalski (ETH Zürich)

Oleksandr Shchur (Ukrainian Catholic University)

JuliaHEP 2025 Workshop, 30.07.2025



Event-processing application frameworks

Many HEP domain specific applications follow similar concept:

For each event:

- Load event data
- Checkout non-event data, meta-data
- Apply transformations, filters
- Write output

Frameworks (Gaudi, CMSSW, ...) support creating such application by providing:

- Execution engine
- Configuration layer
- Event data, non-event data, meta-data management
- Shared resources, services
- ...


Heterogeneous scheduling

A few decades of evolution:

- Single-process, single-thread event loop
- Parallel event processing with multiple processes
- Parallel event processing with multiple threads
- Gradual introduction of offload to accelerators and multi-node super-frameworks

Our R&D project:

- Explore new libraries and systems for heterogeneous computing
- Prototype schedulers starting from a greenfield
- Use realistic workflows extracted from the current frameworks used by the LHC experiments

For porting existing framework to Julia see  [CMS pixel reconstruction talk](#)

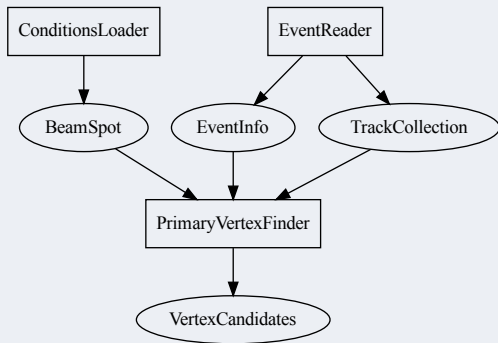
Workflow description

Data-flow graph

Directed acyclic graph (DAG) describing data dependencies between data transformations.

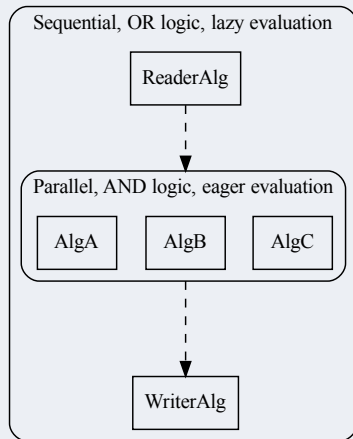
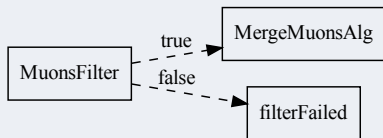
A data-flow graph consists of:

- Algorithms - vertices describing the transformations (rectangular shape)
- Data objects - vertices describing the data (oval shape)
- Directed edges representing dependency relation



Control-flow graph

DAG or subgraphs describing non-data dependencies and conditional scheduling of the algorithms.



Workflows extracted

- ATLAS standard reconstruction test job (q449)
 - ~800 algorithms
 - ~3800 data-objects
- FCC ALLEGRO full simulation
- Artificial workflows featuring specific scenarios (sequential and parallel chains, complex dependencies, ...)

Extracted information:

- data-flow graph
- control-flow graph
- algorithms' timings

Data-object memory footprints measured for selected workflows.


Graphs in Julia

- Workflows used also in other scheduling project and saved in GraphML format.
- Absolutely no problem reading GraphML in C++ (Boost.Graph) or Python (NetworkX).
- No support for reading GraphML with properties in JuliaGraphs
- JuliaGraphs organization seems to have serious maintenance issues:
 - Several months to react to issues or PRs
 - Some packages (GraphViz.jl) broken since Julia 1.11.0

Demonstrator projects

Demonstrators

1. In C++ using Taskflow

 [m-fila/taskflow-fwk](https://github.com/m-fila/taskflow-fwk)

2. In Julia using Dagger.jl*

 [key4hep/key4hep-julia-fwk](https://github.com/key4hep/key4hep-julia-fwk)

- Both limited to data-flow scheduling by executing mockup CPU-crunching algorithms reproducing timings of extracted workflows
- No GPU support implemented yet, for now focused on multi-threading: process multiple events concurrently and exploit intra-graph parallelism

Used Julia 1.11.6 unless stated otherwise.

Algorithms in Julia demonstrator don't allocate any memory.

Taskflow – C++



 [taskflow/taskflow](https://github.com/taskflow/taskflow)

“A General-purpose Task-parallel Programming System — write parallel programs with high performance and simultaneous high productivity”

- Advertised as a modern replacement for oneTBB
- Supports conditional scheduling, CUDA
- Actively developed and maintained, stable releases
- Beautiful, modern C++ API
- Developed at the University of Wisconsin–Madison
- Great documentation



 [JuliaParallel/Dagger.jl](https://github.com/JuliaParallel/Dagger.jl)

*“Dagger.jl is a framework for parallel computing across all kinds of resources, like **CPUs** and **GPUs**, and across **multiple threads** and **multiple servers**.”*

- Under active development, version 0.18.17 used in the demonstrator, unstable public API
- Active and responsive community, weekly user meetings, active channel at Julia's slack
- Developed at MIT
- No obvious alternatives in Julia

Dagger scheduling

Dagger provides 3 different APIs for scheduling:

Task-dependencies

- Assemble graph by passing future-like objects
- Overhead of recreating the graph for each event

Data-dependencies

- Assemble graph by passing wrappers for data objects
- Synchronous, can't process multiple events in parallel

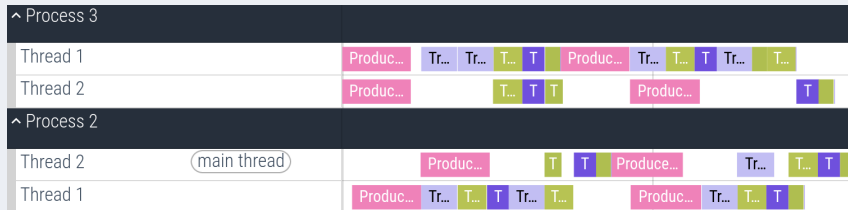
Streaming-tasks

- Create reusable tasks to avoid re-creating graph for each event
- Later addition to Dagger, potentially the best choice for us

Task-dependencies used in the demonstrator

Dagger multi-processing

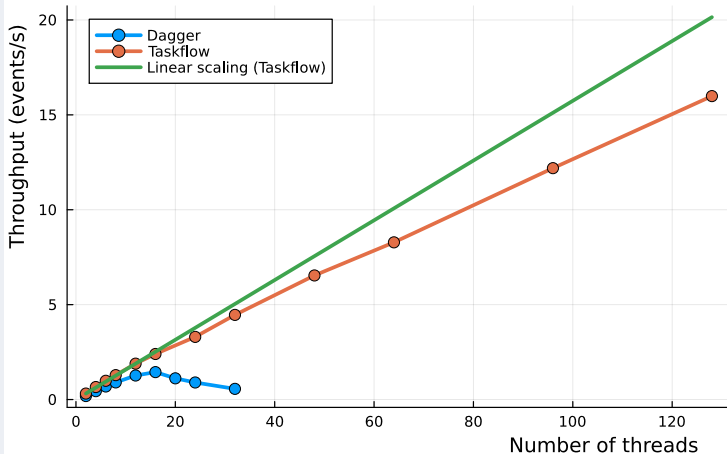
By default a single event is allowed to migrate between processes.
Scoping to a single worker not implemented in the demonstrator.



Occasional errors due to thread-safety problems in standard library Distributed used internally by Dagger. Same errors when changing Dagger's backend to DistributedNext (maintained fork of Distributed with thread-safety fixes). By now Distributed accepted thread-safety fixes from DistributedNext.

Multi-threaded throughput

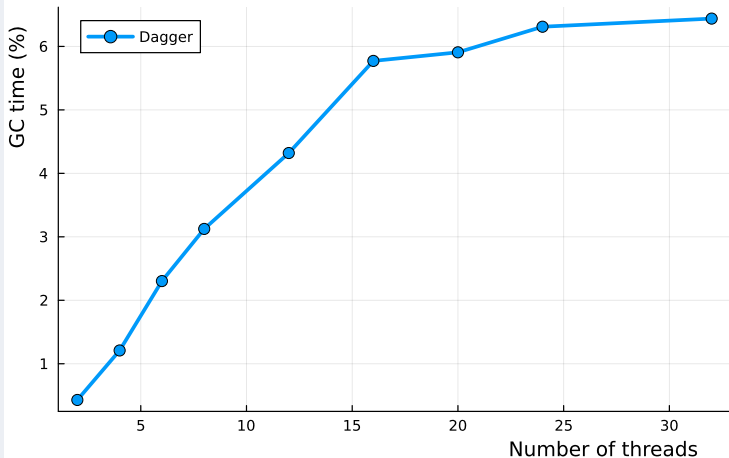
Throughput scaling (Intel Xeon 6780E 144 threads)



- ATLAS q449 data-flow
20 events per thread
20 events warmup per thread,
no allocations inside
algorithms
- No issues with Taskflow
- Substantial performance and
scaling issues with Dagger
- Demonstrator starts throwing
errors above a dozen threads
due to internal data-races in
Dagger

Memory management at fault?

Garbage collection time (Intel Xeon 6780E 144 threads)

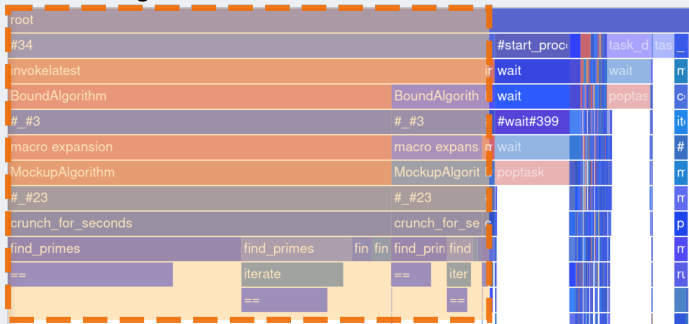


- Julia GC locks all threads but that doesn't fully explain poor throughput scaling
- Dagger developers suggested their internal data-store might require further optimization for multi-threading

Profiling

Flamegraphs

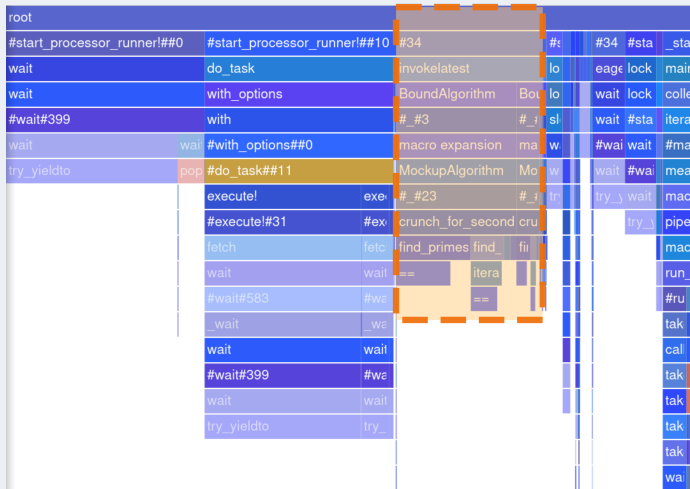
Algorithm execution



- Julia has built-in sampling profiler
- Another recommended profiler PProf.jl fails with "Unexpected 0 in data, please file an issue" error
- Extra configuration might be required to use external profilers

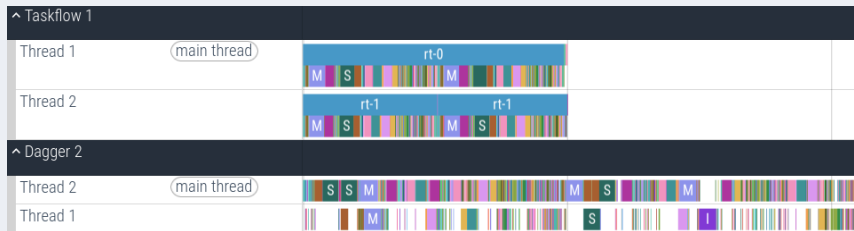
Flamegraphs – walltime

Algorithm execution



- Sampling profilers measure only running tasks
- Upcoming Julia 1.12 adds walltime profiler taking into account waiting tasks. Useful for high task contention or IO
- Walltime profiler shows we have a lot of waiting tasks

Timeline



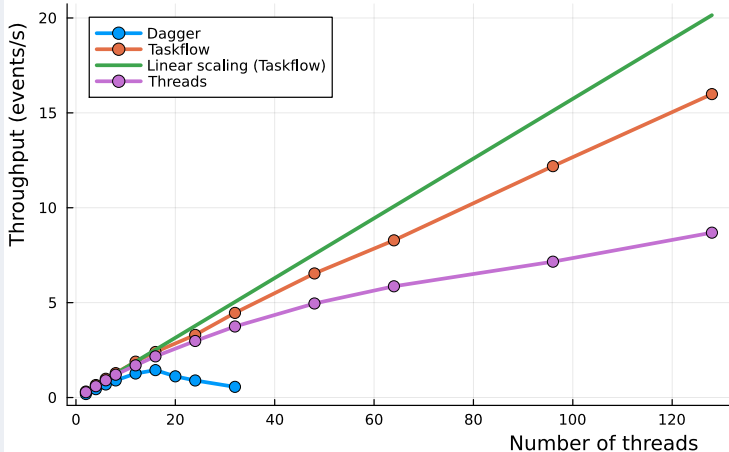
2 threads, 2 concurrent events, 4 events, 4 events warmup, ATLAS q449 data-flow

- Dagger logging noticeably distorts execution with $\sim 40\%$ overhead ($< 1\%$ profiling overhead for Taskflow).
- Switched to NVTX.jl - Julia bindings for NVTX (custom annotations in Nsight profilers). Brings $< 1\%$ overhead, can also annotate GC and JIT activity.

Scheduling revisited

Threads performance

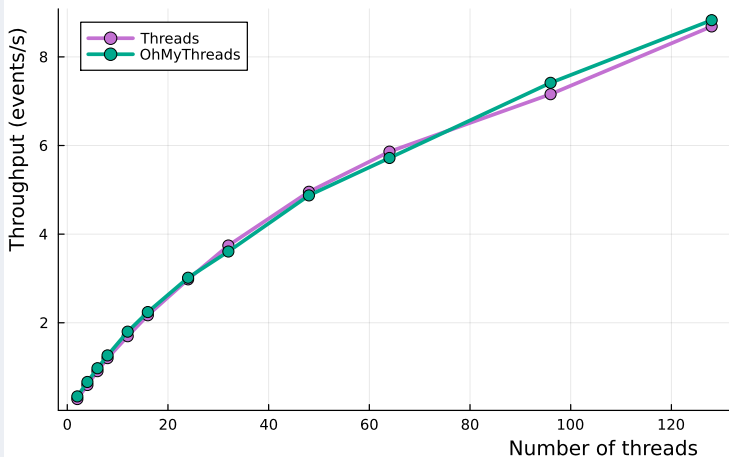
Throughput scaling (Intel Xeon 6780E 144 threads)



- ATLAS q449 data-flow no allocations inside algorithms
- Replaced Dagger with standard library Threads (contrary to the name tasks-based)
- Better performance and scaling, no more crashes
- Lost multi-processing capabilities

OhMyThreads performance – the last minute check

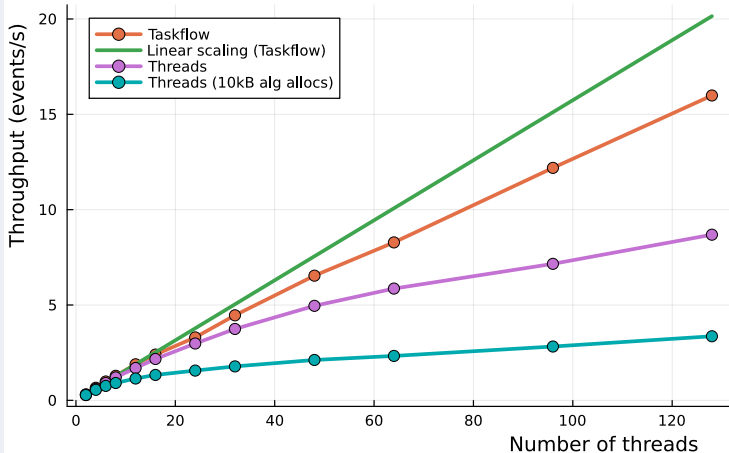
Throughput scaling (Intel Xeon 6780E 144 threads)



- OhMyThreads – the famous data-parallel oriented alternative to Threads
- Re-evaluated after suggestion by Jerry Ling
- Slight improvement for small number of threads, no breakthroughs
- Still a few ideas to try out with Threads

(Back to) Threads with allocations – preliminary view

Throughput scaling (Intel Xeon 6780E 144 threads)



- Not doing any allocations inside the algorithms is unrealistic
- Adding allocations of arbitrary size (here 10 kB) in each algorithms causes significant drop of performance
- GC time growing, default `--gcthreads` used, to be tuned

Custom memory management?

GC is integral part of Julia and custom memory management is non-trivial.

Upcoming Julia 1.13 features:

- Adding support for 3rd part GC (requiring custom Julia build)
- Alternative GC with MMTK (Memory Management Toolkit)

Currently the best advice is to follow guidelines (Julia v1.13-dev docs):

1. Reduce allocation rate: Focus on algorithmic improvements
2. Adjust GC threads: Experiment with different `--gcthreads` settings
3. Use concurrent sweeping: Enable background sweeping with `--gcthreads=N,1`
4. Profile memory patterns: Identify allocation hotspots and optimize them

Demonstrator executable can be AOT compiled with juliac from Julia 1.12.0-rc
Same performance as JITed code

- `--trim=no` – ~ 150 s compilation, ~ 500 M executable
- `--trim=safe` – fails with ~ 50 errors from dependencies.

Some quirks:

- `@time` still reports non-zero compilation time
- With juliac `function @main(args::Vector{String})::Cint` gets extra program name argument which might break your ArgParse. Use separate entrypoints for julia and juliac!
- Set Julia arguments with environmental variables


For compiling libraries see [!\[\]\(339a16584d5da0f0a3ca4e9ec17bf6a1_img.jpg\) Jet Reconstruction talk](#)


Summary

Summary

- Demonstrator for HEP event-processing application framework in Julia, simulating realistic workloads.
- Initially using **Dagger.jl** (integrated multi-threading, multi-processing, GPU scheduling). Benchmarks revealed **very poor throughput scaling** for multi-threading and general **instability**.
- Replaced Dagger with Julia's built-in **Threads**, which provided **better performance and improved reliability**, although limited to multi-threading.
- Although more effort was devoted to the Julia project than to its C++ counterparts, its performance remains **below the C++ alternatives**.
- Julia's memory management and garbage collection may significantly impact multi-threaded applications with frequent allocations, as is common in many HEP workflows.

FrameworkDemo.jl  [key4hep/key4hep-julia-fwk](https://github.com/key4hep/key4hep-julia-fwk)

Taskflow-demonstrator  [m-fila/taskflow-fwk](https://github.com/m-fila/taskflow-fwk)

- Mateusz Fila, *CERN*,  mateusz.jakub.fila@cern.ch
- Graeme A Stewart, *CERN*
- Justin Kowalski, *ETH Zürich*
CERN Summer Student Programme 2025
- Oleksandr Shchur, *Ukrainian Catholic University*
Bachelor thesis student

The work has been supported by the CERN Strategic Programme on Technologies for Future Experiments. <https://ep-rnd.web.cern.ch/>

This work has been partially funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.

This work was supported by computing resources from CERN OpenLab.

