DATE
17/06/2024

EP-SFT Group Meeting

RESEARCH OF THE SCHEDULING MECHANISMS IN JULIA DAGGER LIBRARY

Summary

PRESENTED BY

Oleksandr Shchur
(Ukrainian Catholic University,
Ukrainian Remote Student Program)

SUPERVISOR

Benedikt Hegner

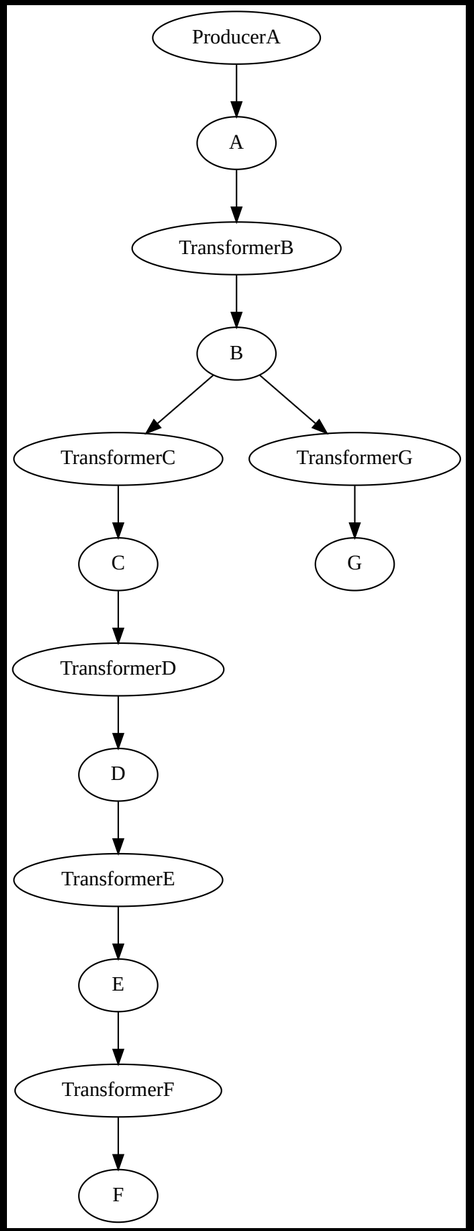
CO-SUPERVISOR

Mateusz Jakub Fila

Introduction

What is this project about?

The workloads of modern data processing frameworks used by LHC experiments can be very often described as directed acyclic graphs of tasks.



Objectives

- Parse a directed acyclic graph (DAG) representing the tasks workflow
- Schedule the tasks parallelly with **Dagger**
- Get scheduler events and metrics logged
- Display the logging data using meaningful representations

Parsing DAG

```xml
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-in
  <key id="key0" for="node" attr.name="class" attr.type="string" />
  <key id="key1" for="node" attr.name="node_id" attr.type="string" />
  <key id="key2" for="node" attr.name="type" attr.type="string" />
  <graph id="G" edgedefault="directed" parse.nodeids="free" parse.edgeids="canonical" parse.order="no
    <node id="n0">
      <data key="key0">MicroProducer</data>
      <data key="key1">ProducerA</data>
      <data key="key2">GaudiAlgorithm</data>
    </node>
    <node id="n1">
      <data key="key0"></data>
      <data key="key1">A</data>
      <data key="key2">DataObject</data>
    </node>
    <node id="n2">
      <data key="key0">MicroTransformer</data>
      <data key="key1">TransformerB</data>
      <data key="key2">GaudiAlgorithm</data>
    </node>
    <node id="n3">
      <data key="key0"></data>
      <data key="key1">B</data>
      <data key="key2">DataObject</data>
    </node>
```

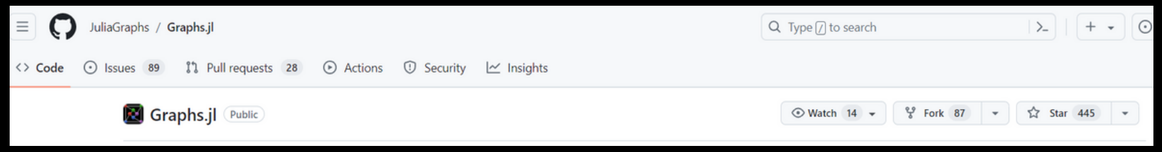
```julia
mutable struct MetaDiGraph{T<:Integer,U<:Real} <: AbstractMetaGraph{T}
    graph::SimpleDiGraph{T}
    vprops::Dict{T,PropDict}
    eprops::Dict{SimpleEdge{T},PropDict}
    gprops::PropDict
    weightfield::Symbol
    defaultweight::U
    metaindex::MetaDict
    indices::Set{Symbol}
end
```

.graphml file examples

MetaDiGraph structure

Parsing DAG: pitfalls

Popular Julia package for graphs processing:



Related packages

It is an explicit design decision that any data not required for graph manipulation (attributes and other information, for example) is expected to be stored outside of the graph structure itself.

Additional functionality like advanced IO and file formats, weighted graphs, property graphs, and optimization-related functions can be found in the packages of the JuliaGraphs organization.

JuliaGraphs organization packages:

Packages

Algorithms and graph types

The central package of the ecosystem is Graphs.jl. It contains a standard graph interface and some basic types for unweighted graphs, as well as a set of combinatorial algorithms like shortest paths.

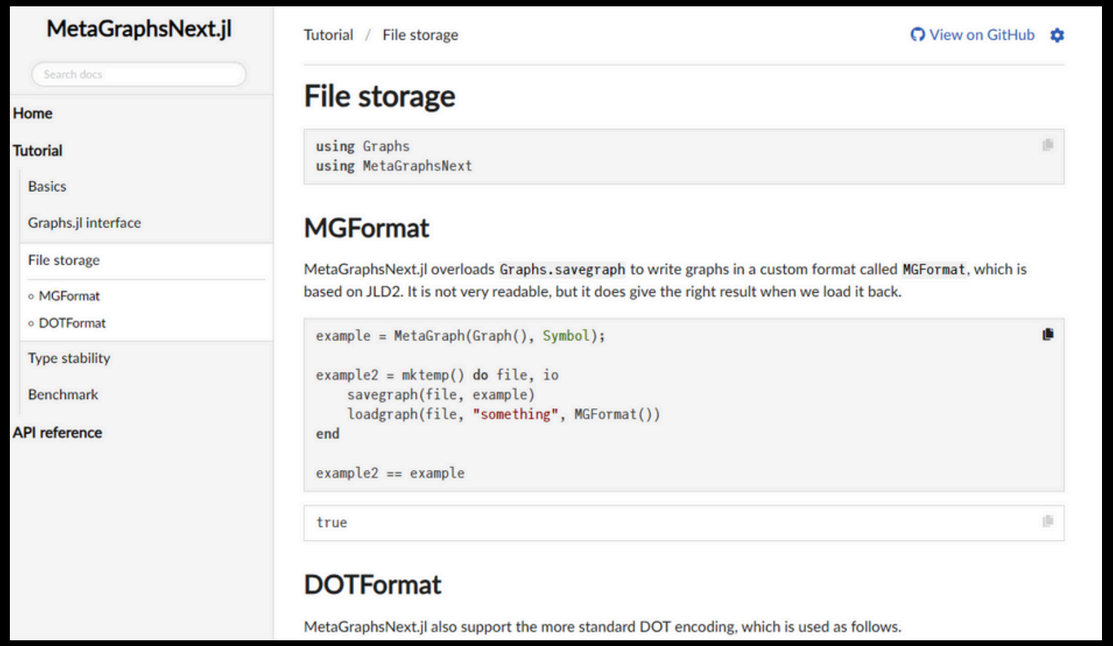
Many of the other packages we list rely only on this interface, so as to be compatible with arbitrary graph types.

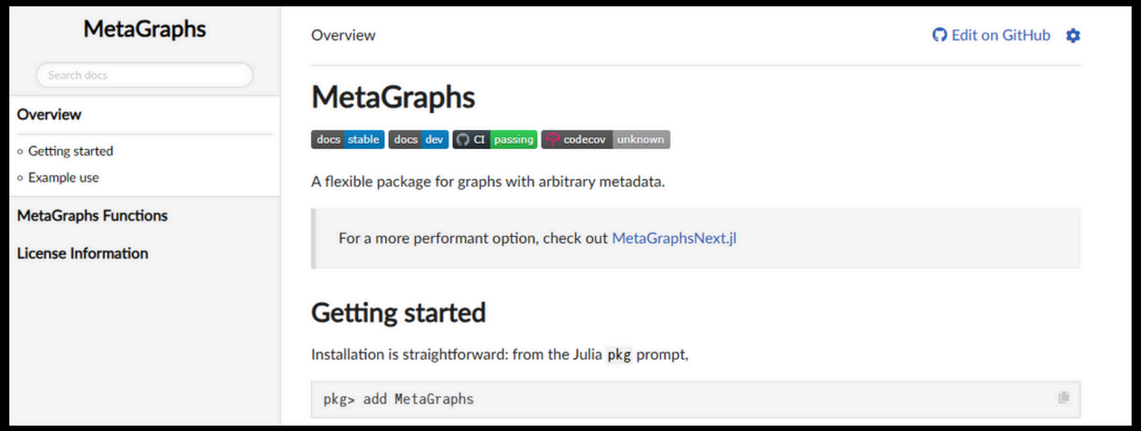
More graph types

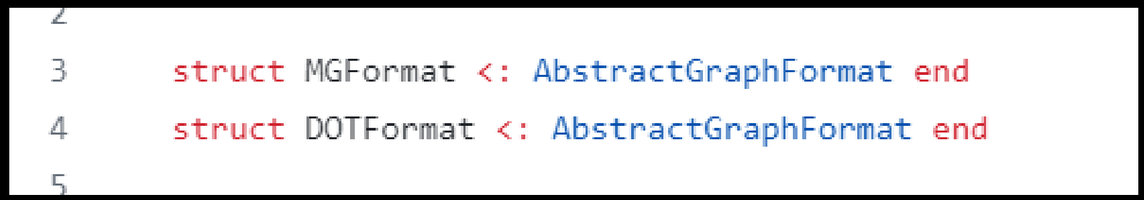
- SimpleWeightedGraphs.jl: Simple graphs with weighted edges.
- MetaGraphs.jl: Graphs with metadata on the vertices and edges.
- MetaGraphsNext.jl: A more efficient but less flexible alternative to MetaGraphs.jl.
- StaticGraphs.jl: Memory-efficient immutable graphs.
- MultilayerGraphs.jl: Graphs with multiple layers.

Parsing DAG: metagraphs

Both **MetaGraphs.jl** and **MetaGraphsNext.jl** do not support loading graphs from **.graphml**







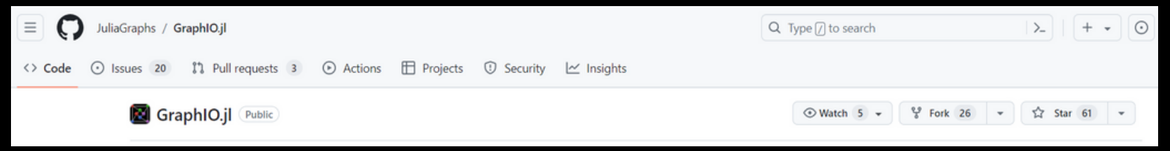
Parsing DAG: GraphIO

JuliaGraphs also mentions:

Interfaces and visualization

Input / Output

- GraphIO.jl: Read graphs from files and write them to files in various formats.
- SNAPDatasets.jl: Extract graphs from the SNAP Datasets collection.



README License

GraphIO

CI passing codecov 97% code style blue tested with Aqua.jl

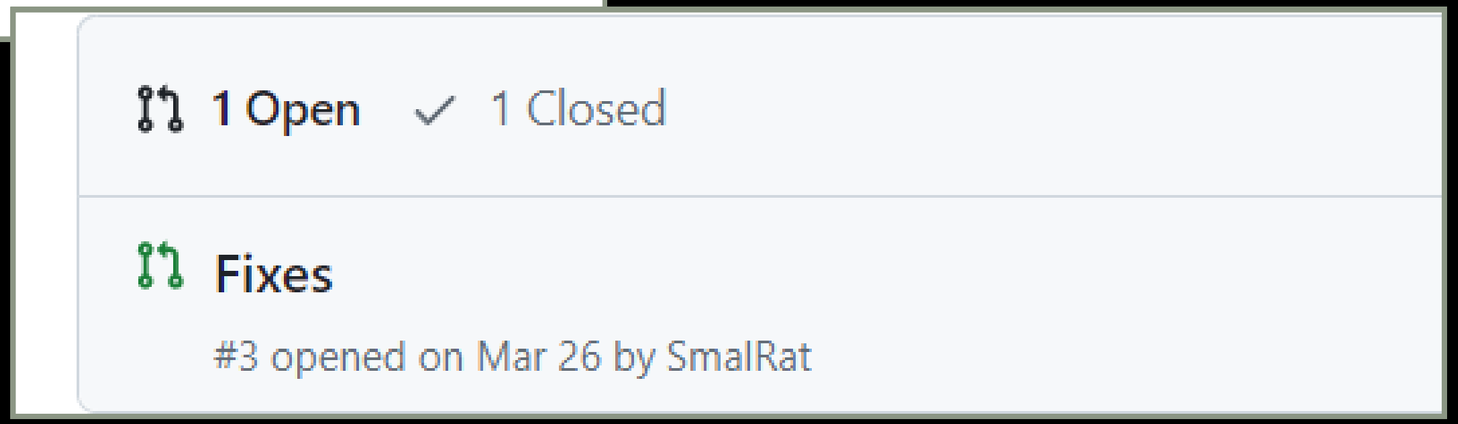
GraphIO provides support to Graphs.jl for reading/writing graphs in various formats.

Currently, the following functionality is provided:

Format	Read	Write	Multiple Graphs	Format Name	Comment
EdgeList	✓	✓		EdgeListFormat	a simple list of sources and dests separated by whitespace and/or comma, one pair per line.
GML	✓	✓	✓	GMLFormat	
Graph6	✓	✓	✓	Graph6Format	
GraphML	✓	✓	✓	GraphMLFormat	
Pajek NET	✓	✓		NETFormat	
GEXF		✓		GEXFFormat	
DOT	✓		✓	DOTFormat	

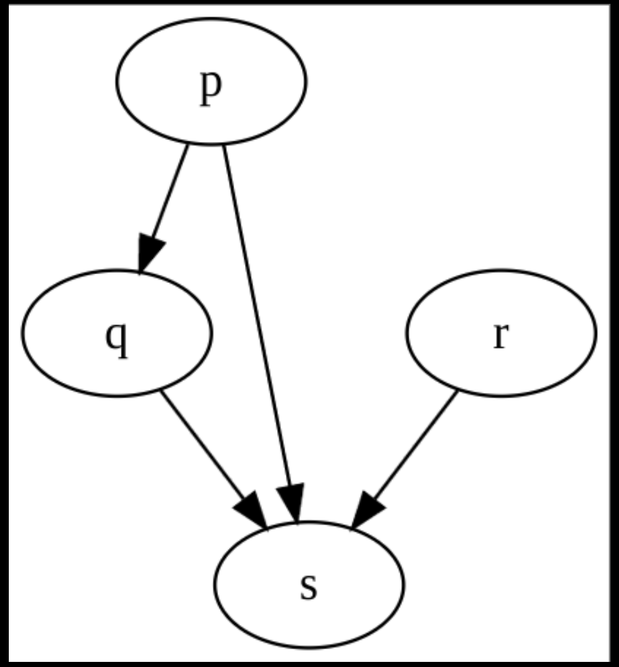
But it does not parse metadata...

Parsing DAG: GraphMLReader



Dagger: purpose

Essentially, **Dagger** is a scheduler, which "can run computations represented as directed-acyclic-graphs (DAGs) efficiently on many Julia worker processes and threads" (from Github readme)



Example from the Dagger documentation

Dagger

Objectives:

- Schedule tasks DAG (data dependencies only).
- Tasks are simple mockups with the minimal data exchange
- Set the limit of concurrently running DAGs
- Re-run tasks DAG without the creation overhead

Dagger: API

Old Dagger API

```
function oldAPI_graph_setup(time_to_sleep)
    a = Dagger.delayed(taskA)(time_to_sleep, 1)
    b = Dagger.delayed(taskB)(time_to_sleep, 2, a)
    c = Dagger.delayed(taskC)(time_to_sleep, 3, a, b)
    d = Dagger.delayed(taskD)(time_to_sleep, 4, a, b, c)
    e = Dagger.delayed(taskE)(time_to_sleep, 5, a)
    f = Dagger.delayed(taskF)(time_to_sleep, 6, a, b, c, d, e)

    return f
end

ctx = Dagger.Sch.eager_context()
scheduled_graph = oldAPI_graph_setup(1)
result_reference = compute(ctx, scheduled_graph)
res = collect(ctx, result_reference)
```

Static predefined DAG execution

Modern Dagger API

```
function modernAPI_graph_setup(time_to_sleep)
    a = Dagger.@spawn (taskA)(time_to_sleep, 1)
    b = Dagger.@spawn (taskB)(time_to_sleep, 2, a)
    c = Dagger.@spawn (taskC)(time_to_sleep, 3, a, b)
    d = Dagger.@spawn (taskD)(time_to_sleep, 4, a, b, c)
    e = Dagger.@spawn (taskE)(time_to_sleep, 5, a)
    f = Dagger.@spawn (taskF)(time_to_sleep, 6, a, b, c, d, e)

    return f
end

scheduled_graph = modernAPI_graph_setup(1)
res = fetch(scheduled_graph)
```

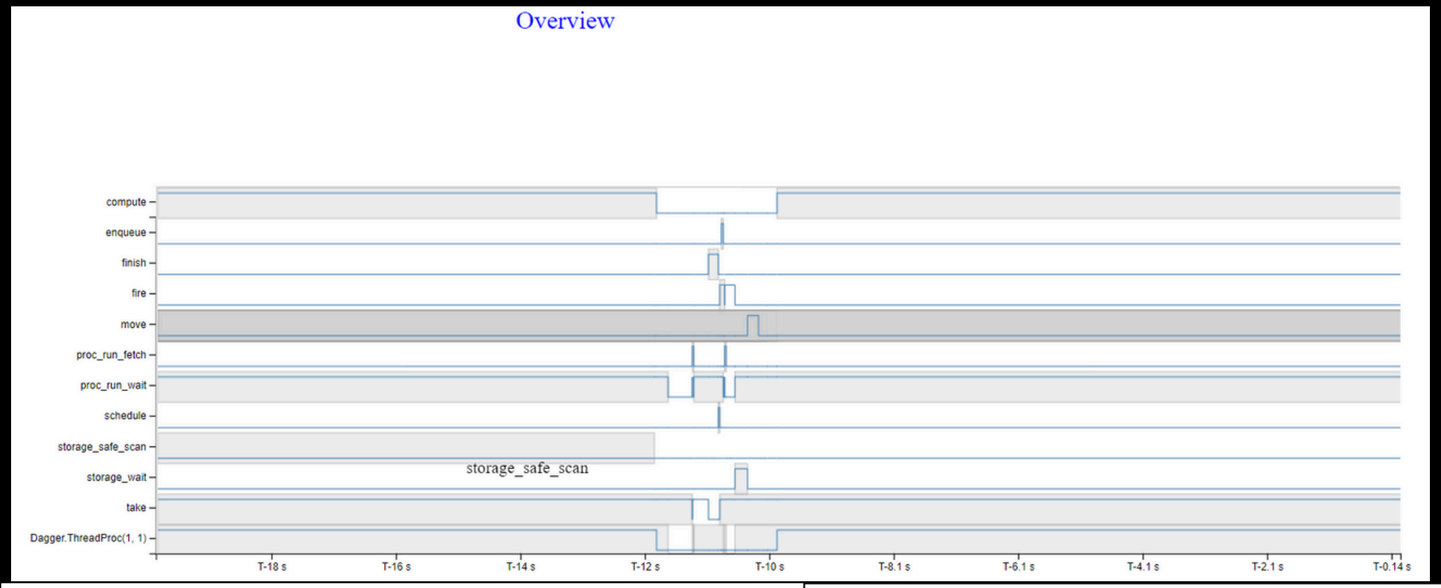
Allows for the dynamic changes

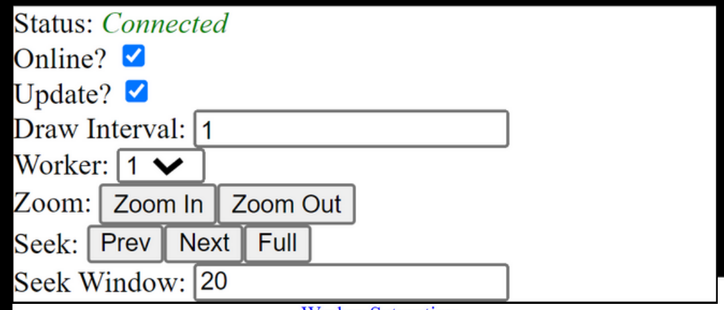
Dagger: logging (v 0.18.8)

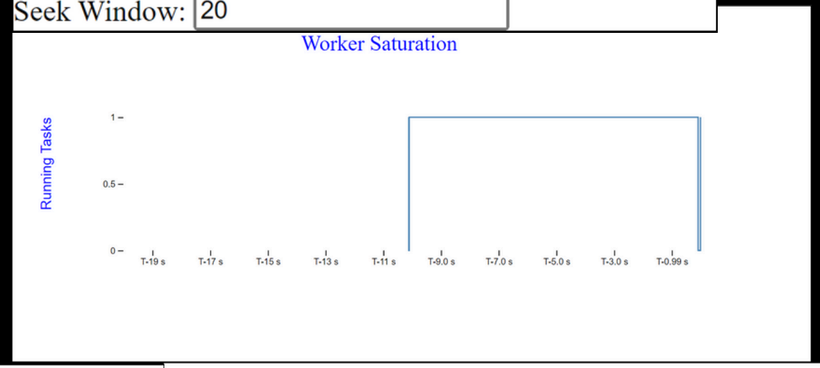
DaggerWebDash

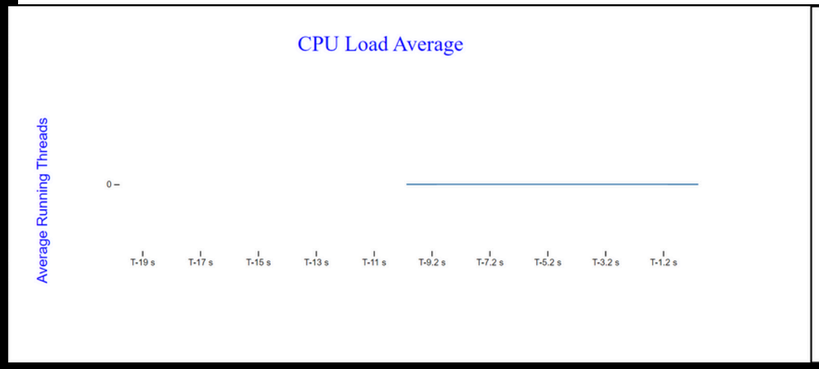
Raw logs processing

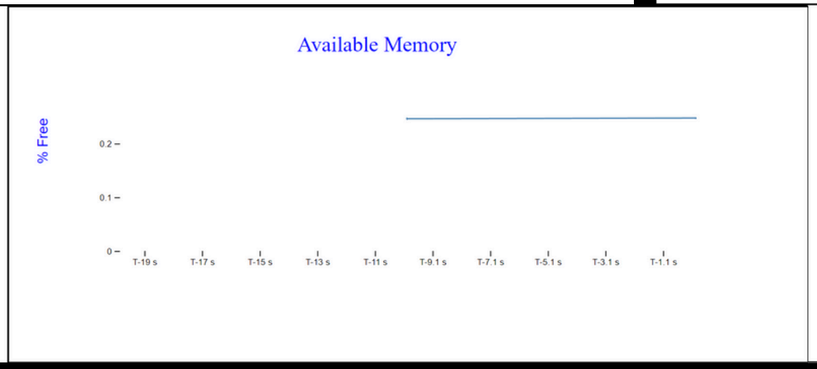
DaggerWebDash: metrics

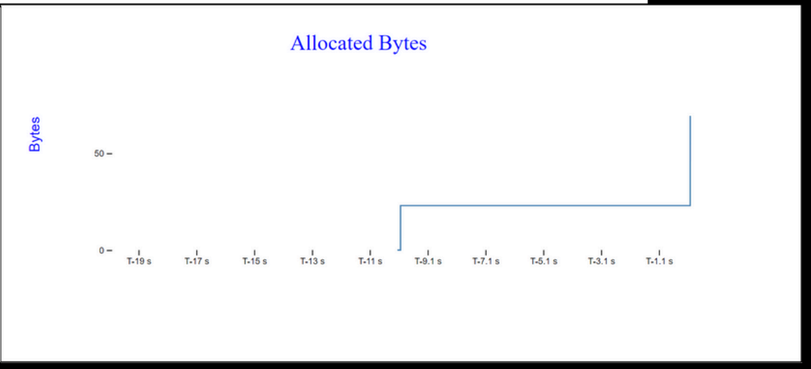




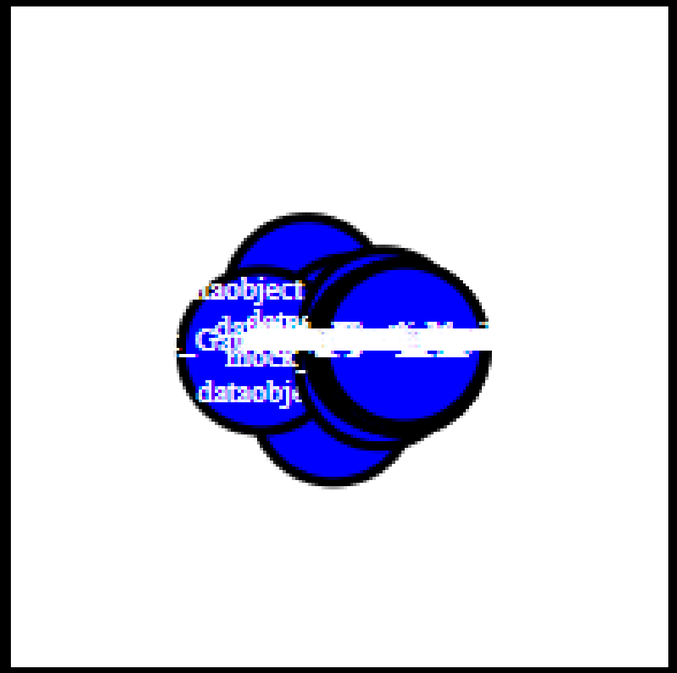


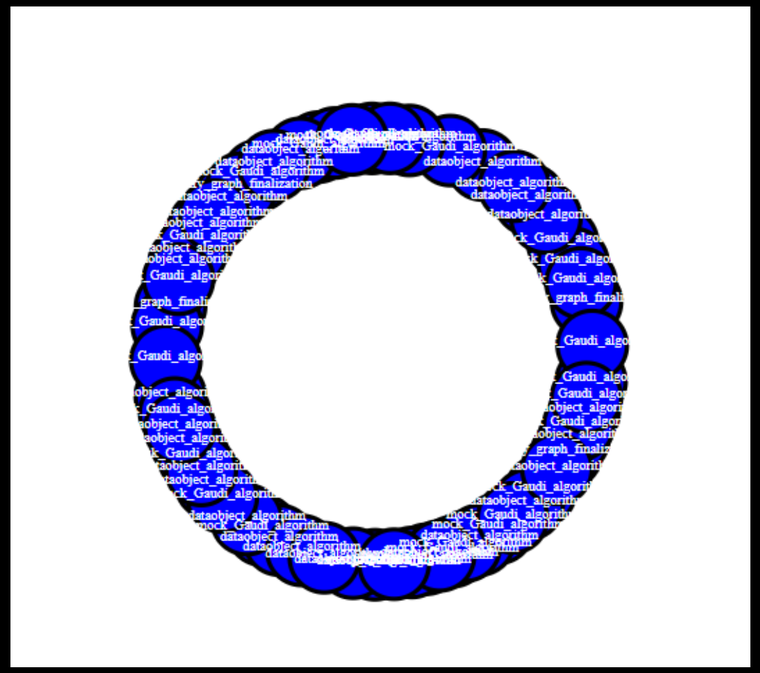


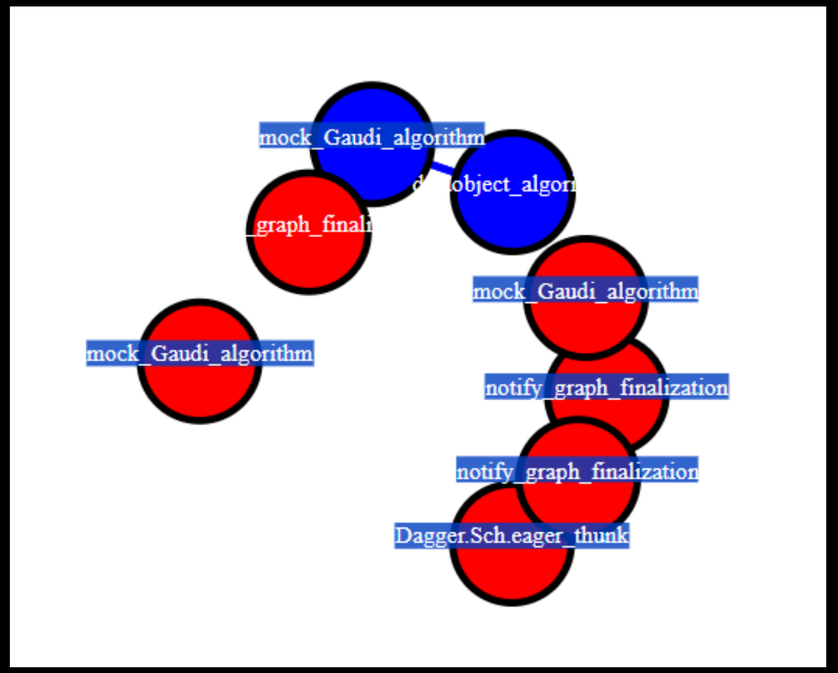




DaggerWebDash: DAGs







DaggerWebDash: errors

When using worker processes

```
      From worker 5:    |        @ TimespanLogging C:\Users\A
dmin\.julia\packages\TimespanLogging\DsD3O\src\core.jl:294
      From worker 5:    |     [19] (::Dagger.Sch.var"#138#146
"{Dagger.Sch.ProcessorInternalState, UInt64, RemoteChannel{Ch
annel{Any}}, Dagger.ThreadProc})()
      From worker 5:    |        @ Dagger.Sch C:\Users\Admin\
.julia\packages\Dagger\Tx54v\src\sch\Sch.jl:1197
      From worker 5:    └ @ TimespanLogging C:\Users\Admin\.j
ulia\packages\TimespanLogging\DsD3O\src\core.jl:229
      From worker 5:    ┌ Error: Error during log aggregation
:
      From worker 5:    |   exception =
      From worker 5:    |    IOError: listen: address already
 in use (EADDRINUSE)
      From worker 5:    |    Stacktrace:
      From worker 5:    |      [1] uv_error
      From worker 5:    |        @ .\libuv.jl:100 [inlined]
      From worker 5:    |      [2] #listen#13
      From worker 5:    |        @ C:\Users\Admin\.julia\juli
aup\julia-1.10.2+0.x64.w64.mingw32\share\julia\stdlib\v1.10\S
ockets\src\Sockets.jl:628 [inlined]
      From worker 5:    |      [3] listen
      From worker 5:    |        @ C:\Users\Admin\.julia\juli
aup\julia-1.10.2+0.x64.w64.mingw32\share\julia\stdlib\v1.10\S
ockets\src\Sockets.jl:627 [inlined]
      From worker 5:    |      [4] #listen#10
      From worker 5:    |        @ C:\Users\Admin\.julia\juli
aup\julia-1.10.2+0.x64.w64.mingw32\share\julia\stdlib\v1.10\S
```

Upon receiving the request

```
IOError: write: operation canceled (ECANCELED)
Stacktrace:
  [1] uv_write(s::Sockets.TCPSocket, p::Ptr{UInt8}, n::UInt64
)
    @ Base .\stream.jl:1066
  [2] unsafe_write(s::Sockets.TCPSocket, p::Ptr{UInt8}, n::UI
nt64)
    @ Base .\stream.jl:1120
  [3] unsafe_write
    @ .\io.jl:698 [inlined]
  [4] unsafe_write(s::Sockets.TCPSocket, p::Base.RefValue{UIn
t16}, n::Int64)
    @ Base .\io.jl:696
  [5] write
    @ .\io.jl:699 [inlined]
  [6] write
    @ .\io.jl:702 [inlined]
  [7] writeframe(io::HTTP.Connections.Connection{Sockets.TCPS
ocket}, x::HTTP.WebSockets.Frame)
    @ HTTP.WebSockets C:\Users\Admin\.julia\packages\HTTP\sJD
5V\src\WebSockets.jl:187
  [8] send(ws::HTTP.WebSockets.WebSocket, x::String)
    @ HTTP.WebSockets C:\Users\Admin\.julia\packages\HTTP\sJD
5V\src\WebSockets.jl:542
  [9] client_handler(sock::HTTP.WebSockets.WebSocket, id::Int
64, port::Int64, port_range::UnitRange{Int64}, config_updated
::Base.RefValue{Bool}, config::Vector{Any}, seek_store::Nothi
```

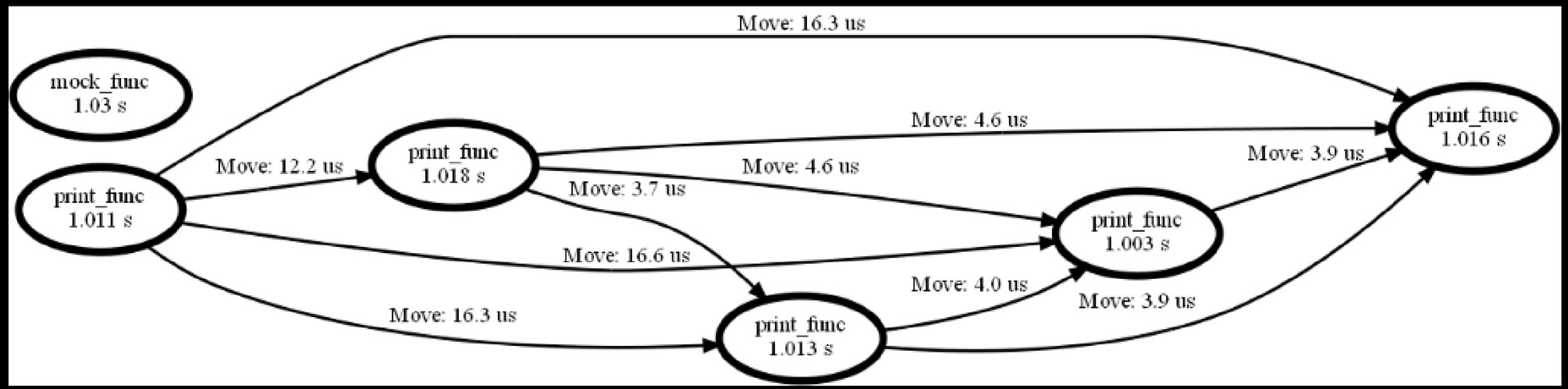
DaggerWebDash

Summary

- Plots are quite incomprehensible
- It is hard to summarize information for all the workers at once
- Random update times (from seconds to complete stagnation)
- Constantly throwing errors

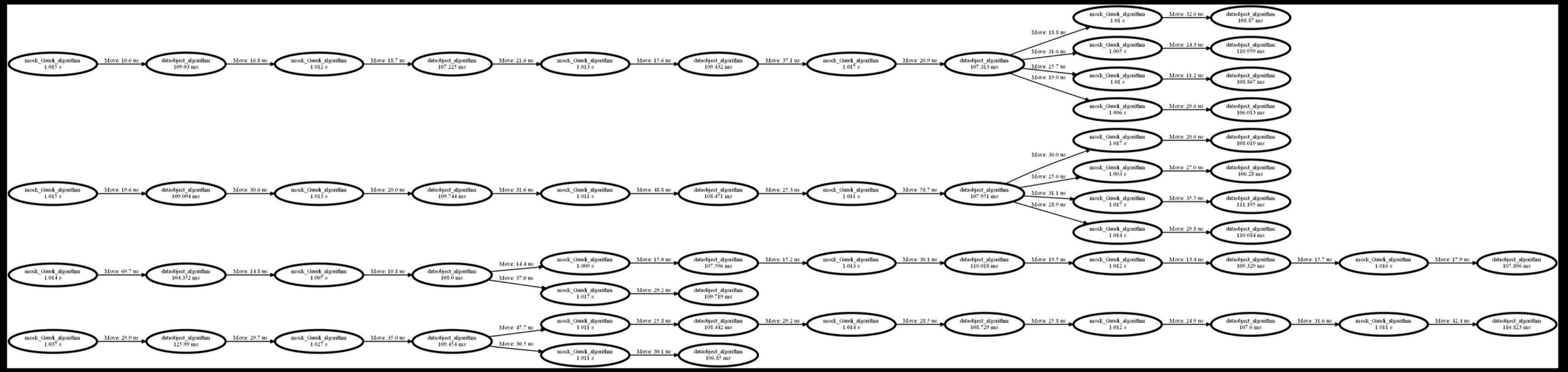
Using raw logs

Dagger provides means to convert logs to a DAG in the **.dot** format



DAG from logs: results

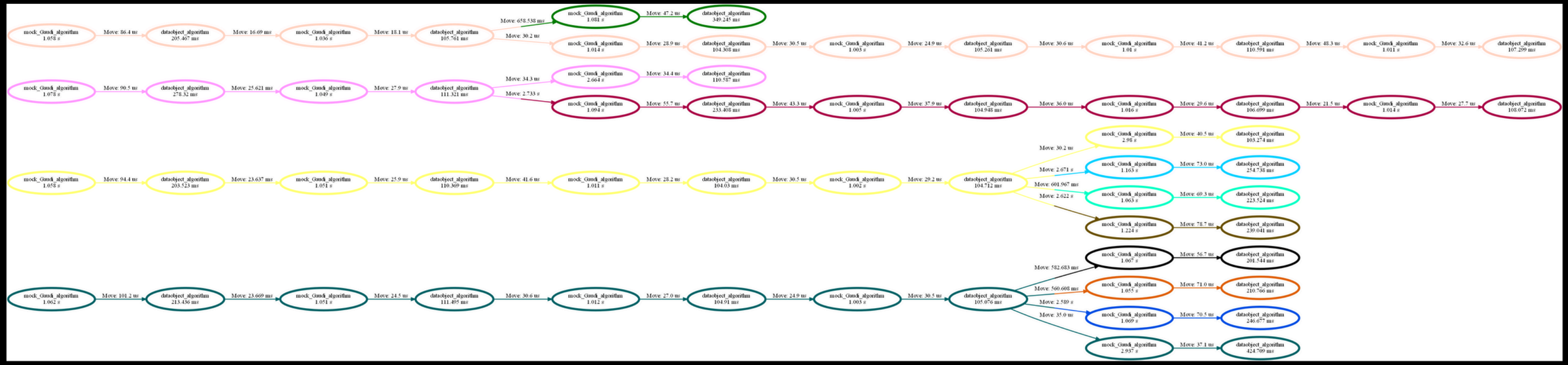
"Move" operations time: ~10-100 us



Sequential DAG execution (1 process, 1 thread)

DAG from logs: results

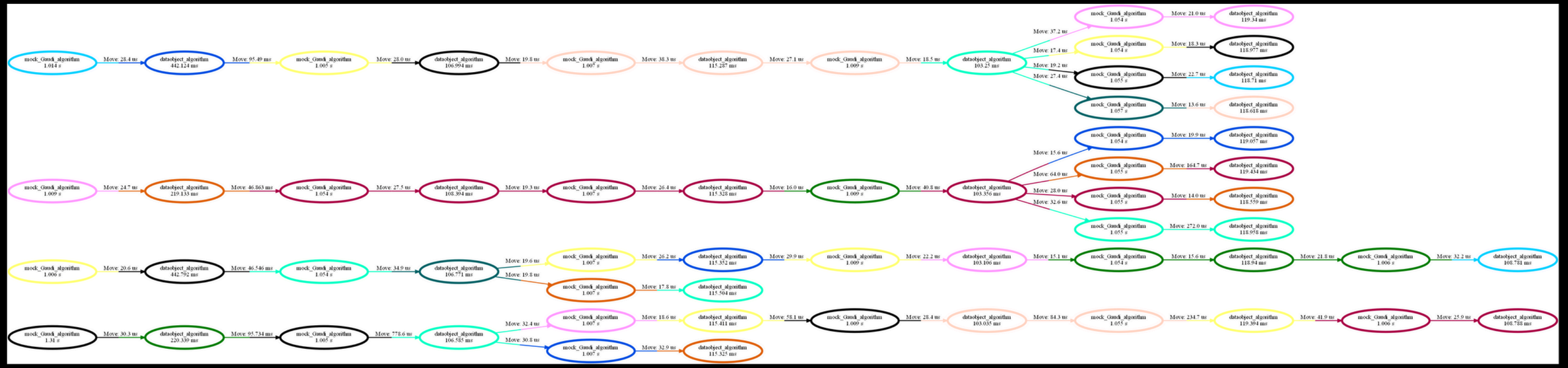
"Move" operations time: ~500-3000 ms (between processes)



Multiple processes DAG execution (13 processes, 1 thread each)

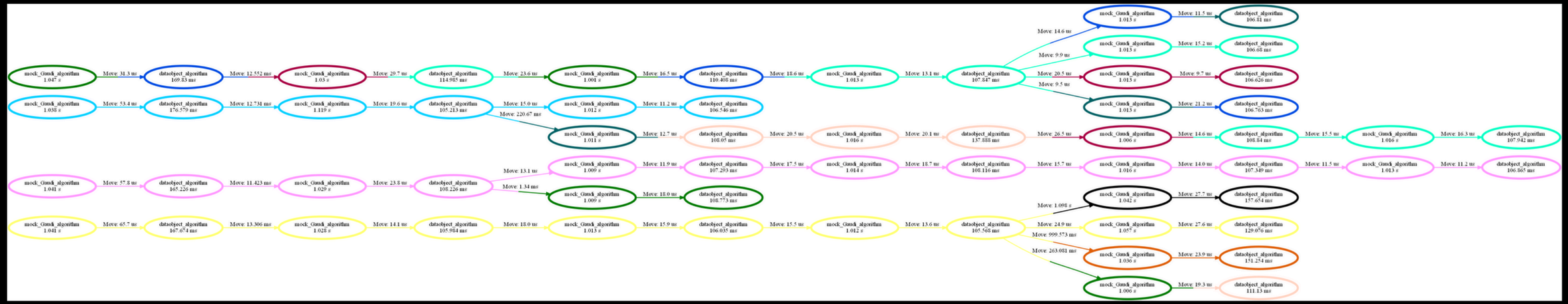
DAG from logs: results

"Move" operations time: ~10-100 us (however, there are some outliers)



Multithreaded DAG execution (1 process, 13 threads)

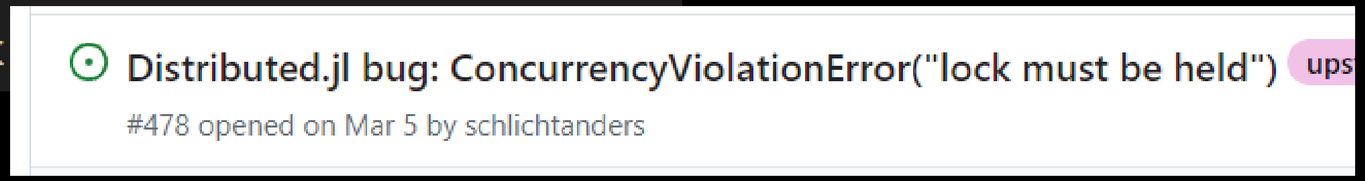
DAG from logs: results



DAG execution (Main process - 7 threads; 6 workers - 1 thread)

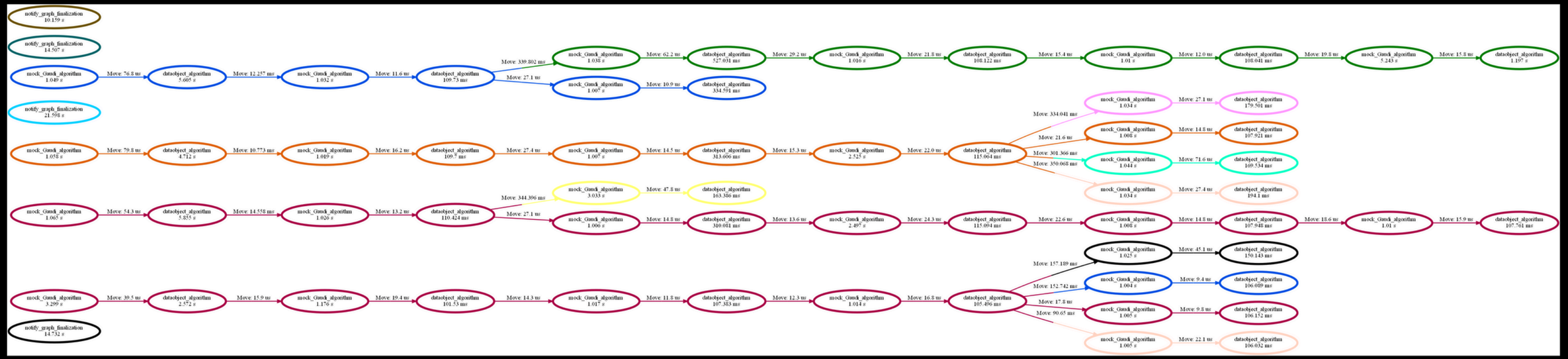
DAG from logs: results

```
ThunkFailedException:
  Root Exception Type: CapturedException
  Root Exception:
ConcurrencyViolationError("lock
```



DAG execution (Main process - 1 thread; 6 workers - 2 threads)

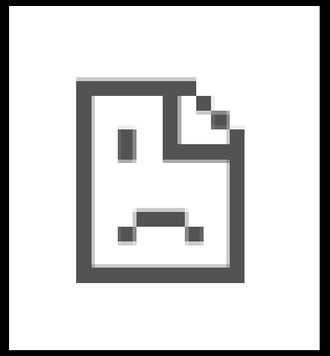
Limit the number of concurrently running DAGs



Each DAG notifies the dispatcher (process 1) by **RemoteChannel** upon its completion. After that, the new tasks are scheduled.

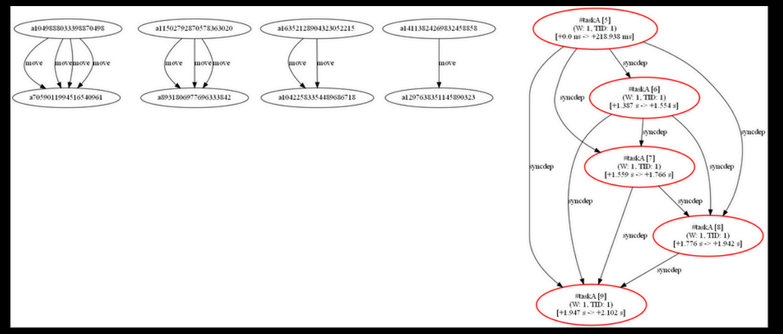
Dagger: logging (v 0.18.11)

DaggerWebDash

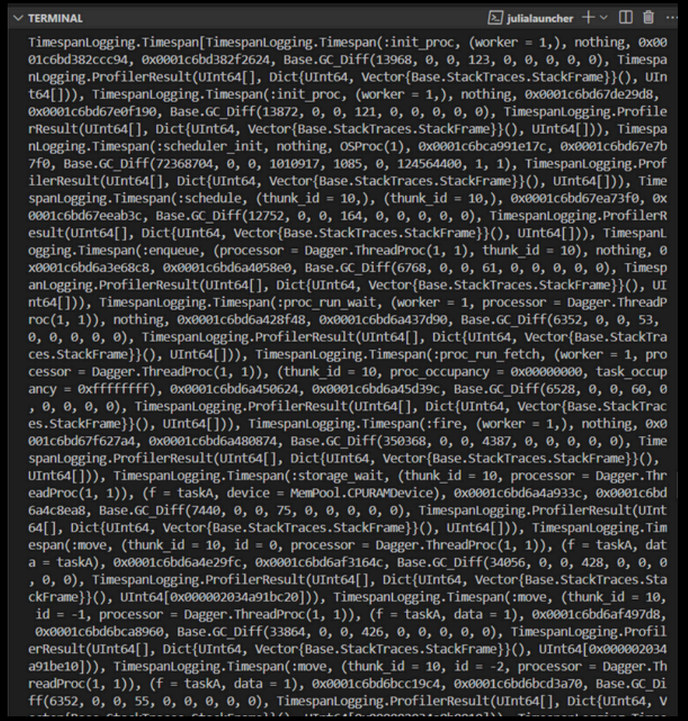


Doesn't work anymore

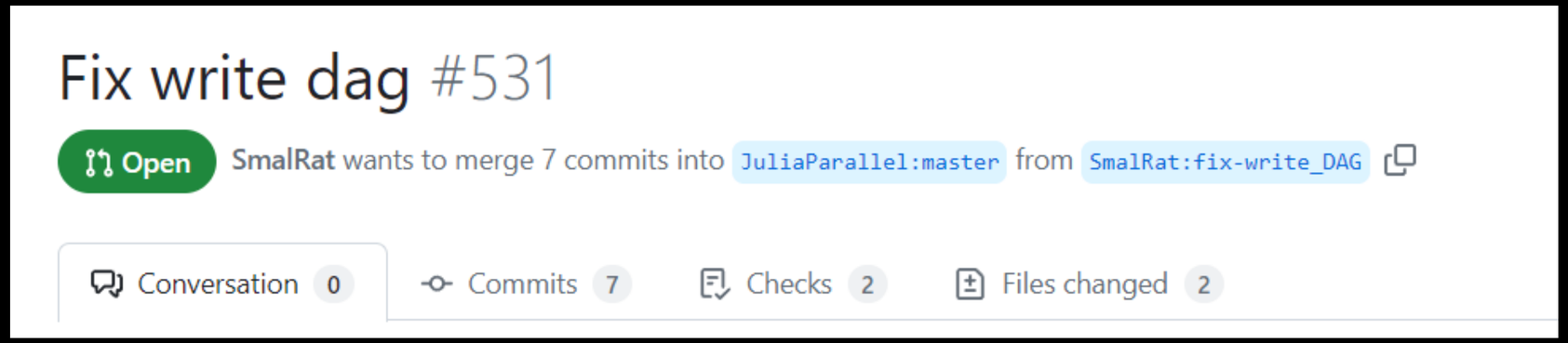
Render logs



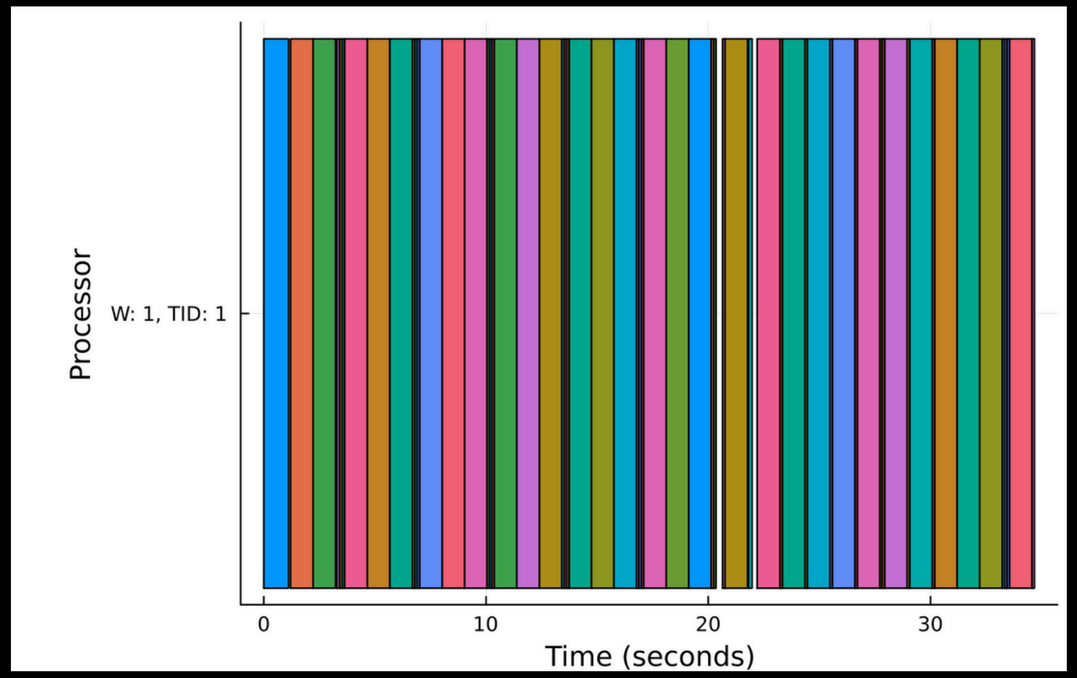
Raw logs processing



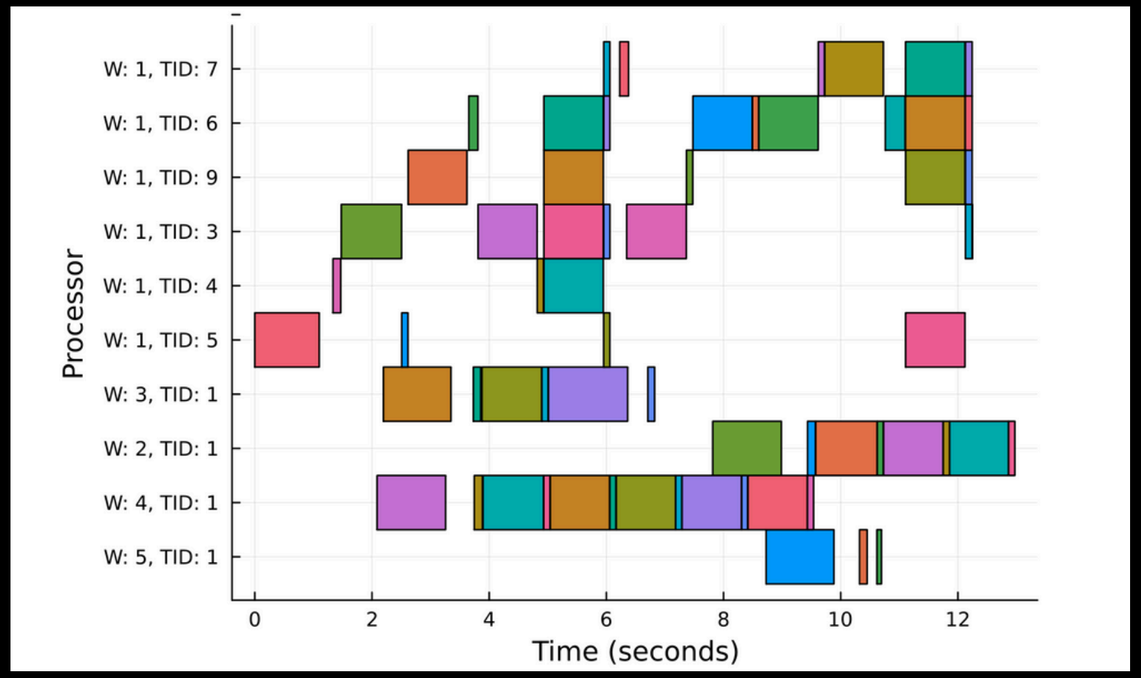
Dagger: fixing visualization



Dagger 0.18.11: Gantt chart



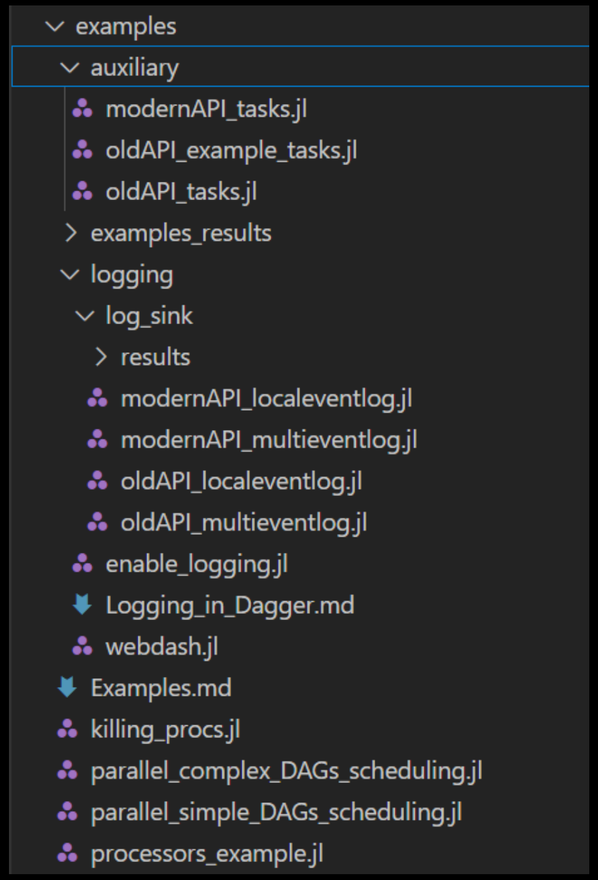
1 process, 1 thread



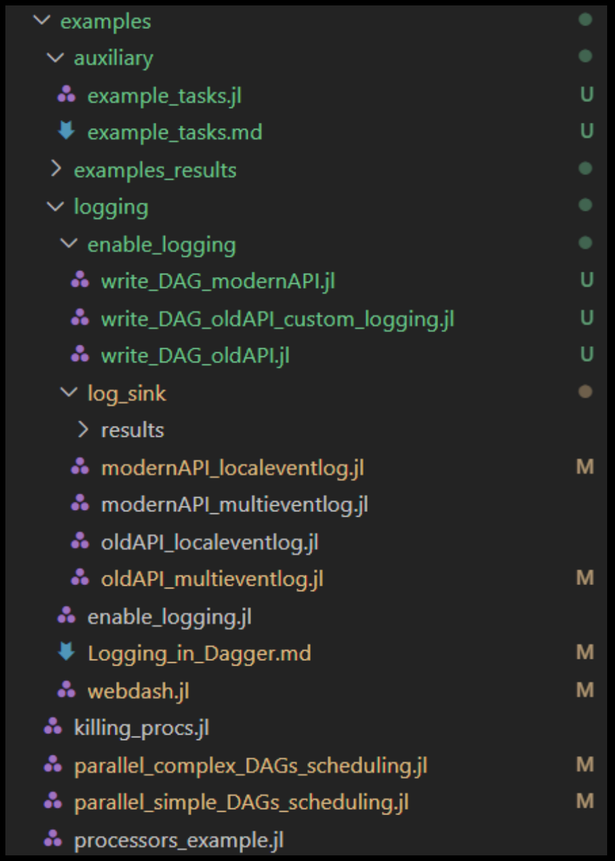
Main process - 9 threads,
4 workers - 1 thread

Examples & Documentation

0.18.8



0.18.11



Dagger: summary

Issues

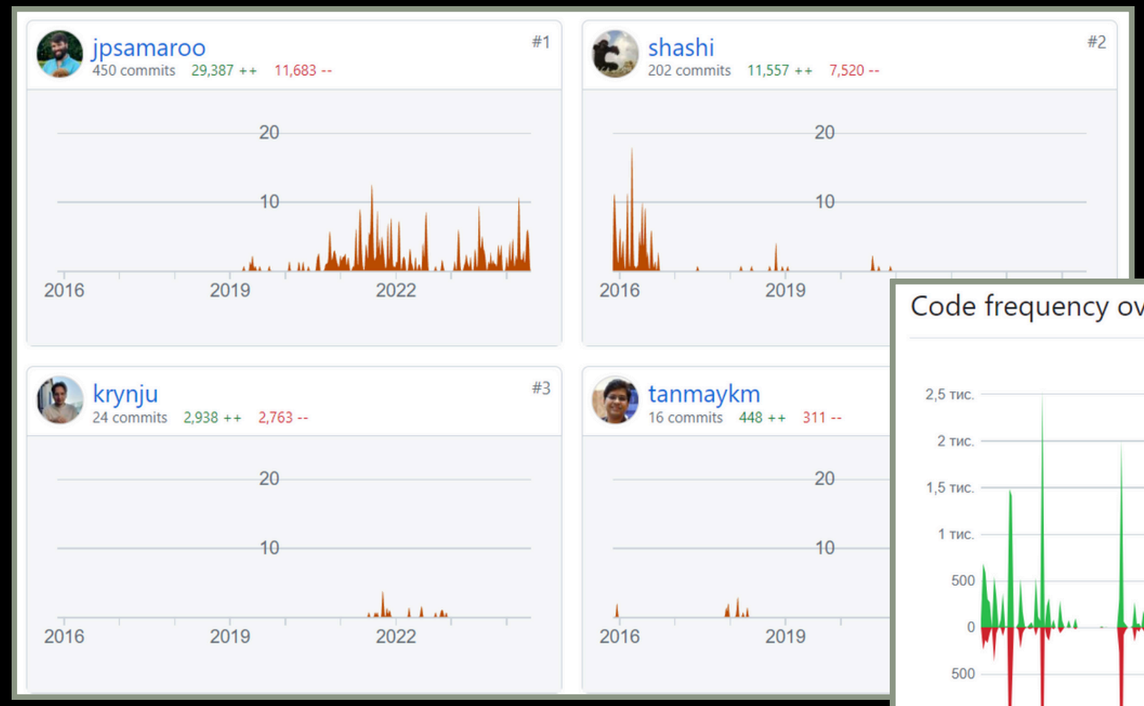
- Many features are undocumented
- Some documented features do not work
- Frequent updates are not backward compatible
- Relatively small community

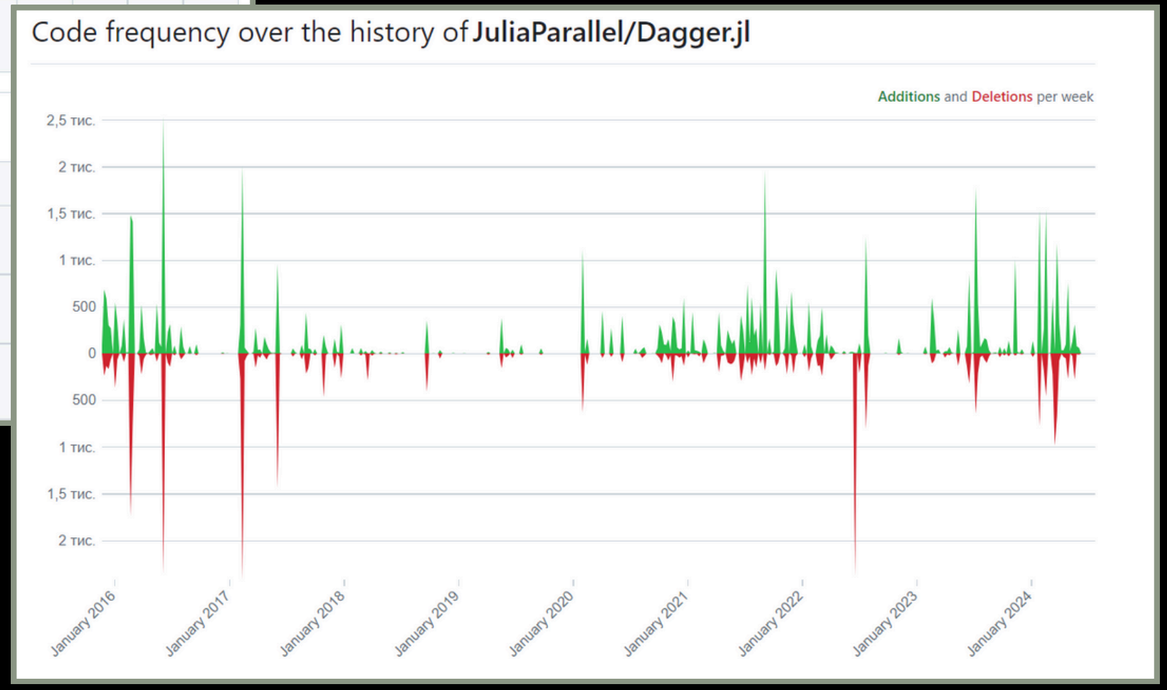
Advantages

- Actively maintained
- Easy to use, when documented

Dagger is promising , but it may be too early to use it

Dagger: updates





Thank you for your attention!