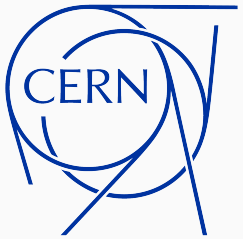


Graph-based Task Scheduling on Heterogeneous Resources

Josh Ott
with Mateusz Fila and Benedikt Hegner (EP-SFT)
North Carolina State University

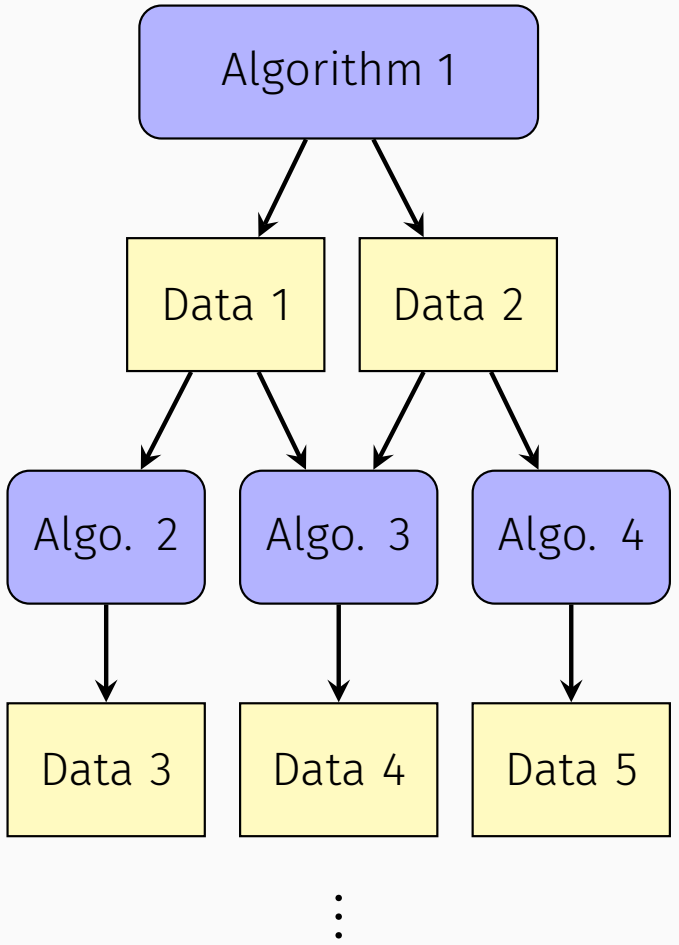
August 8, 2024

Introduction

Motivation

Directed acyclic graphs (DAG)!

- Data processing jobs in LHC experiments can be described with DAGs
- Algorithms transform data, going from any number of input nodes to any number of output nodes
- These graphs are to be scheduled in parallel on heterogeneous resources



Project specifics

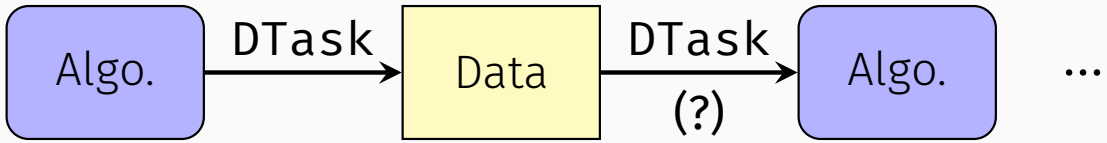
Our goals

- Develop a demonstrator framework to meet LHC requirements
- Design it with heterogeneous computing in mind from the start
- Write it in Julia using the `Dagger.jl` package

Data Dependencies

Treatment of data

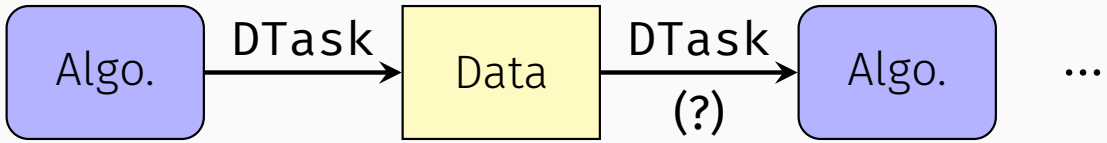
Algorithm and data nodes treated equally – not ideal!



Dagger requires this to recognize dependence.

Treatment of data

Algorithm and data nodes treated equally – not ideal!

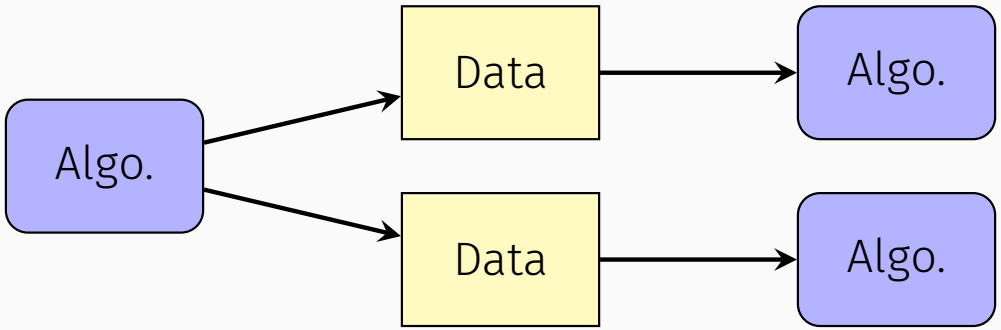


Dagger requires this to recognize dependence.

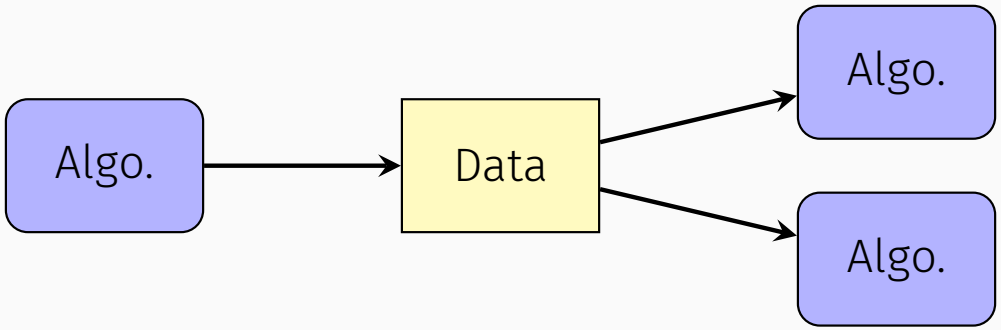
The data generated was also meaningless :(

Graph structures

A graph with topology



would actually get treated like



`DataObjects` with metadata

To utilize the graph metadata, we package everything in structs

```
mutable struct DataObject
    data
    size::UInt
end

function algorithm(inputs... , outputs... )
    for output in outputs
        output.data = zeros(Int8, output.size)
    end
end
```

`DataObjects` with metadata

To utilize the graph metadata, we package everything in structs

```
mutable struct DataObject
    data
    size::UInt
end

function algorithm(inputs... , outputs... )
    for output in outputs
        output.data = zeros(Int8, output.size)
    end
end
```

We populate data nodes with these objects prior to scheduling algorithms.

All data as arguments

Rather than using the function output, treat all outputs as mutable arguments.

```
Dagger.spawn_datadeps() do
    for v in vertices
        Dagger.@spawn algorithm(
            In.(inputs)...,
            Out.(outputs)... )
    end
end
```

All data as arguments

Rather than using the function output, treat all outputs as mutable arguments.

```
Dagger.spawn_datadeps() do
    for v in vertices
        Dagger.@spawn algorithm(
            In.(inputs)... ,
            Out.(outputs)... )
    end
end
```

Now our data is meaningful and properly handled!

Datadeps revisited

This was great until we noticed some strange behavior. The source code confirms:

```
function spawn_datadeps(f::Base.Callable)
```

> *"At the end of executing* `f`, `spawn_datadeps` *will* ***wait for all launched tasks to complete****, rethrowing the first error, if any. The result of* `f` *will be returned from* `spawn_datadeps`*."*

Datadeps revisited

This was great until we noticed some strange behavior. The source code confirms:

```
function spawn_datadeps(f::Base.Callable)
```

> *"At the end of executing* `f`*,* `spawn_datadeps` *will* ***wait for all launched tasks to complete****, rethrowing the first error, if any. The result of* `f` *will be returned from* `spawn_datadeps`*."*

Asynchronicity is crucial, so this will not do. Come back to it later!

CPU Crunching

Initial conditions

Previously, algorithm nodes slept for a fixed amount of time to emulate computation.

```
function mock_algorithm(id, data...)
    println("Algorithm for vertex $(id)")
    sleep(1)

    return id
end
```

Initial conditions

Previously, algorithm nodes slept for a fixed amount of time to emulate computation.

```
function mock_algorithm(id, data...)
    println("Algorithm for vertex $(id)")
    sleep(1)

    return id
end
```

We'd like

- runtime to be realistic, and for
- workers to be busy – not sleeping.

Callable structs

Simple: create algorithms as callable structs so workers don't need to read from the graph.

```
struct MockupAlgorithm
    name::String
    runtime::Float64
end

function (alg::MockupAlgorithm)(args...)
    println("Executing $(alg.name)")
    sleep(alg.runtime)

    return alg.name
end
```

Crunching

Now we just need workers to be actually busy. How can we waste a deliberate amount of time?

Crunching

Now we just need workers to be actually busy. How can we waste a deliberate amount of time?

Find prime numbers!

This is the approach that GAUDI (the current framework) uses.

Calibration

Dagger offers **shards** for distributing objects scoped to specific workers.

```
coefficients = Dagger.@shard calibrate()
```

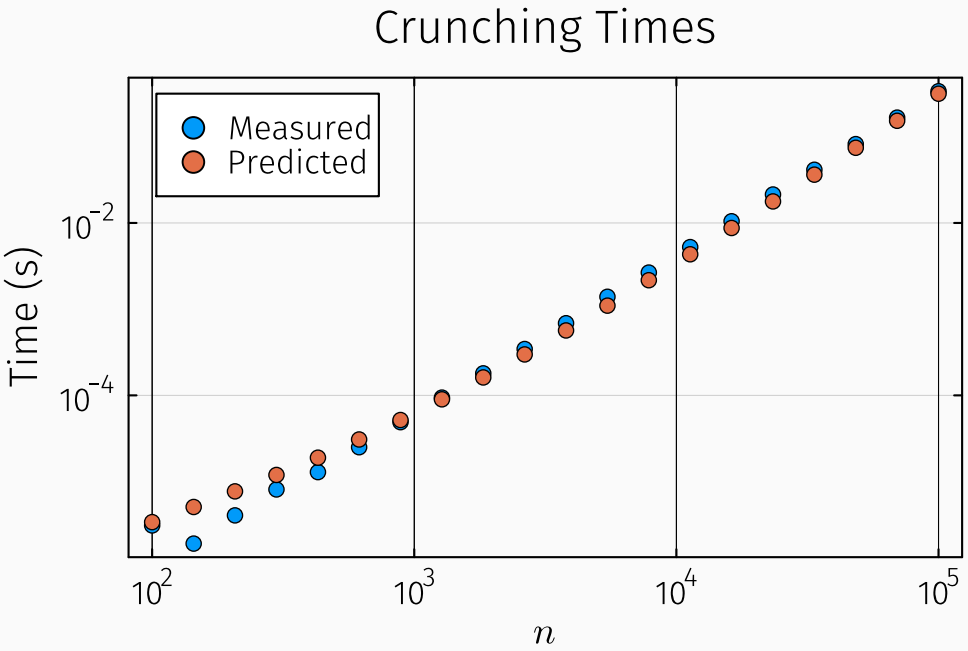
When this gets passed in a `Dagger.@spawn` call, the shard is resolved in the worker process.

Calibration

Dagger offers **shards** for distributing objects scoped to specific workers.

```
coefficients = Dagger.@shard calibrate()
```

When this gets passed in a `Dagger.@spawn` call, the shard is resolved in the worker process.



Algorithm with crunching

Now our algorithm looks like this: we pass through the calibration coefficients and find prime numbers to keep the worker busy.

```
function (alg::MockupAlgorithm)(args... ; coefs)
    println("Executing $(alg.name)")
    crunch_for_seconds(alg.runtime, coefs)

    return alg.name
end
```

Algorithm with crunching

Now our algorithm looks like this: we pass through the calibration coefficients and find prime numbers to keep the worker busy.

```
function (alg::MockupAlgorithm)(args... ; coefs)
    println("Executing $(alg.name)")
    crunch_for_seconds(alg.runtime, coefs)

    return alg.name
end
```

- Accuracy of cruncher varies with environment
- Could be smarter about calibration

Project Structure

Organization

Before:

```
data/
examples/
graphs_scheduling/
  src/
  test/
parsing_graphs/
  src/
  test/
utilities/
  functions.jl
  auxiliary_funcs.jl
```

Much of the previous changes involved rewriting the existing code, but the larger structure also needed reworking.

- Could only be run through an "example"
- Sources split across directories
- "Utilities" served essential functions

Organization

Before:

```
data/
examples/
graphs_scheduling/
  src/
  test/
parsing_graphs/
  src/
  test/
utilities/
  functions.jl
  auxiliary_funcs.jl
```

After:

```
bin/ (an executable!)
data/
deps/
docs/
examples/
scripts/
src/ (now with a module!)
  FrameworkDemo.jl
  ...
test/ (unit tests!)
  runtests.jl
```

Summary

Summary

Accomplished:

- Once Dagger updates `spawn_datadeps`, the new data dependency system should be good to go
- Algorithms now keep workers properly busy for approximately the amount of time we want, though it could be improved
- Framework now much easier to work with!

Looking ahead:

- Rich logging features
- GPU support
- Control flow graphs

Summary

Accomplished:

- Once Dagger updates `spawn_datadeps`, the new data dependency system should be good to go
- Algorithms now keep workers properly busy for approximately the amount of time we want, though it could be improved
- Framework now much easier to work with!

Looking ahead:

- Rich logging features
- GPU support
- Control flow graphs

Thank you everyone! :)