Graph-based Task Scheduling on Heterogeneous Resources

Josh Ott
Department of Physics, North Carolina State University
Supervisors: Mateusz Fila and Benedikt Hegner

i. Introduction

In high performance computing applications, one is typically concerned with breaking up a large problem into many small parts that can be run in parallel and synchronized. In high-energy physics (HEP) applications, we instead work with many small, independent physics events that are easy to parallelize. Meeting the data processing and computational demands of high-energy physics thus requires unique frameworks designed for handling these events. This report covers the development of a demonstrator framework for the purpose of exploring possible directions for the software stacks of colliders like the Future Circular Collider.

Data processing workflows of the Large Hadron Collider (LHC) experiments can be represented with directed acyclic graphs (DAGs) consisting of data and algorithm nodes. Algorithm nodes transform data and perform computations, outputting results into one or more data nodes. We also make the assumption that no two algorithms can write to the same data node, though algorithm nodes can pull from multiple data nodes. On top of these data flow graphs, workflow descriptions also include control flow graphs which allow for conditional execution of subgraphs. For now, we only consider static graphs.

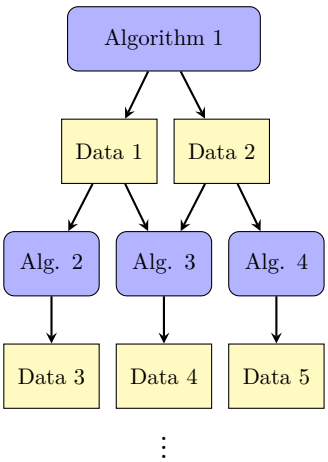


Figure 1: Example data flow graph.

ii. Implementation

Software in HEP is typically written in one of two languages: C++ or Python. Their uses are quite disjoint, as the former offers excellent performance with not-so-intuitive code while the latter offers a high-level, intuitive experience with poor performance. We have chosen to use Julia, a language which aims to solve this dilemma by combining the readability of Python with the performance of C++. Julia has been receiving increasing attention from the HEP community [1] for its possible role in future software. As the language is still relatively quite young, development is very much ongoing to fill in the gaps required by HEP.

Frameworks for handling task-scheduling have historically not been designed from the start with heterogeneous computing in mind. Making full use of all available resources is crucial, so we have made that a priority for our demonstrator framework. As such, we handle task-scheduling with the `Dagger.jl` package [2], which attempts to unify various tools in Julia for parallelization. The graphs we examine are either arbitrary or from usage patterns of existing frameworks used in LHC experiments. The work detailed in this report is a continuation of previous work done establishing the graph parsing and scheduling foundations for the framework [3].

iii. Data Dependencies

A key aspect of executing data-flow DAGs is the creation and transfer of data. Especially on distributed, heterogeneous systems, transferring data between workers can be a costly operation, making it a necessary consideration when assessing the performance of a scheduler. In this section we investigate ways for making the treatment of data in our framework truer to real-life scenarios.

The first problem comes in a dissonance between Dagger's conception of dependence and our chosen graph description. In Dagger, scheduled tasks return a `DTask` – an object that represents the result of a future, in progress, or completed task. It is through passing these `DTasks` on to another task that Dagger recognizes dependence. As such, to ensure that an algorithm runs after a parent data node is populated, our framework would previ-

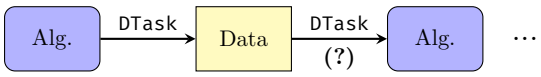
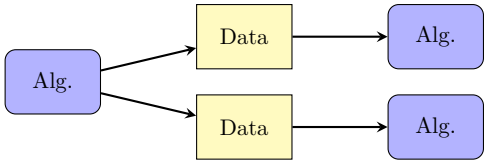
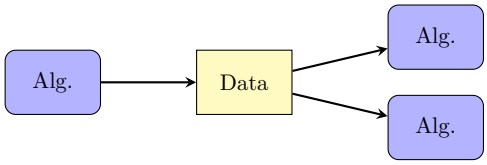


Figure 2: Propagation of `DTask` objects across all nodes.



(a) Desired graph topology



(b) How the graph is treated

Figure 3: Illustration of mismatch in graph description vs. graph execution.

ously schedule the data node and propagate the resulting `DTask` to its children. As data nodes do not perform any operations, however, they should not require time to be scheduled on a worker.

Another problem arises in the restriction of spawned function outputs. We often want to separate function results into disjoint data nodes to minimize unnecessary data transfer. Calls to `Dagger.@spawn` return only one `DTask`, regardless of how many values are returned by the spawned function. Thus, if an algorithm node has two child data nodes, the propagated `DTask` will contain the data for both data nodes, effectively resulting in there being only one data node as shown in Fig. 3. To separate the data according to their destination, we need another way to retrieve them from scheduled tasks.

An alternative to describing dependencies with `DTasks` is found in the function `spawn_datadeps()`, which allows for function arguments to be declared as mutable and later used as dependencies. The documentation [2] provides

```
Dagger.spawn_datadeps() do
    Dagger.@spawn add!(InOut(B), In(A))
    Dagger.@spawn copyto!(Out(C), In(B))
end
```

as an example. Since `B` has been marked as mutable in the `add!` call and `copyto!` needs to read from `B`, `copyto!` will wait until `add!` has completed. Following this example, parent data nodes can therefore be handled with mutable objects wrapped in `In(...)`, with child data nodes getting wrapped in `Out(...)` to show dependence.

Related to this are the specific contents of the data nodes generated by algorithms. The expense of data transfer is of course a function of the size of said data, so to have an accurate representation of HEP workflows, we must emulate the size of data nodes in the graph descriptions of workflows in LHC experiments. Now combining the need for mutable data objects and access to data expectations, we introduce a `DataObject` struct

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

Graph metadata is inaccessible to worker processes, so this struct contains a `size` field to influence the data generated on the worker.

The `data` field of the `DataObject` is what will get modified in algorithms, meaning the `DataObject` needs to exist prior to algorithm execution. To ensure this, we populate data nodes prior to scheduling, setting `data` to `nothing` and `size` according to the graph metadata. This covers the extent of the work we need to do with data nodes, meaning scheduling them is no longer necessary. Next, we loop through all algorithm nodes within a `spawn_datadeps` block, collecting their parent and child `DataObjects` as arguments wrapped with `In(...)` and `Out(...)` respectively.

With these changes, the major problems with data management have been addressed. Unfortunately, an unforeseen quirk with `spawn_datadeps` turned out to be incompatible with the use case of the framework. When scheduling tasks with `spawn_datadeps`, Dagger waits until all scheduled tasks in the block have finished before moving on. As these blocks are at the scope of a single event, this means we can only schedule one event at a time. Of course, a defining characteristic of HEP workflows is having many events that can be processed in parallel. Until asynchronous behavior is possible, then, we will leave these changes out of the main branch.

iv. CPU Crunching

Next, we focus on the behavior of algorithm nodes. Just as data nodes need to emulate the data transfer conditions, algorithm nodes need to emulate the computational load of transforming and processing data. Inspired by GAUDI [4], the framework used by LHCb and ATLAS, we equip our mockup algorithm with a prime number-finding function to keep workers busy.

The goal of our algorithm is to keep a worker busy – which we refer to as crunching – for a precise and accurate amount of time. It is therefore better to be especially naïve with the prime number-finder implementation so it remains most predictable. Our algorithm, described in Alg. 1, is manifestly $\mathcal{O}(n^2)$, and as such we assume that the time complexity takes the form $T(n) = an^2 + bn$. To go from a desired duration to the corresponding n,

we need to know the approximate values of the coefficients a and b on the target worker. In the context of a heterogeneous computing, these coefficients could be very different depending on the worker, necessitating a calibration to be performed on each one.

Algorithm 1 Prime finding algorithm

Require: $n \geq 2$
1: $X \leftarrow [2]$
2: **for** $i \leftarrow 3, n$ **do**
3: $p \leftarrow$ true ▷ stays true if prime
4: **for** $j \leftarrow 2, i/2$ **do**
5: **if** j divides i **then**
6: $p \leftarrow$ false
7: **end if**
8: **end for**
9: **if** $p = 1$ **then**
10: append(X, i)
11: **end if**
12: **end for**
13: **return** X[end]

To calibrate the crunching parameters on each worker, we utilize the `Shard` object in Dagger. Shards represent a group of mutable objects, each scoped to a different worker. For instance, invoking `Dagger.@shard f()` will run `f()` on all workers, saving the results to each corresponding "shard piece." When a task using the shard is scheduled, the responsible worker will use the object stored in its local memory during execution. With this, we just need a calibration function that saves the relevant parameters to each worker for algorithm execution.

To calibrate a and b in $T(n)$ on a worker, we run two benchmarks: one at low n ($n_1 = 1\,000$) and another at high n ($n_2 = 200\,000$), recording the elapsed time. With times T_1 and T_2, we solve the equation

$$\begin{bmatrix} n_1^2 & n_1 \\ n_2^2 & n_2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix} \tag{1}$$

for a and b. Despite this approach being rather crude, it performs reasonably well in approximating the time complexity on a given worker, as shown in Fig. 4.

Average runtime information can be extracted from frameworks currently used by LHC experiments and saved as a property of algorithm nodes in our graphs. As this information is not available to workers, we represent algorithms with a struct containing a runtime field, similar to our treatment of data objects. This time though, we make the struct callable so it can be directly scheduled in Dagger.

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

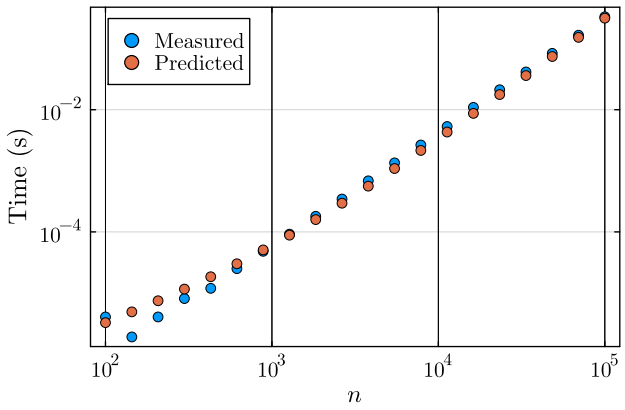


Figure 4: Comparison of calibrated prediction of algorithm runtime to actual, measured runtime.

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

    return alg.name
end
```

The procedure then goes like this: read in the data for an algorithm node in the graph, initialize an appropriate `MockupAlgorithm`, then schedule the algorithm in Dagger, using the callable struct as the executable argument. In practice, the runtime of the mockup algorithm varies depending on number of threads, processes, graphs, etc., with simpler scenarios seeing runtimes closer to the provided average. A smarter calibration scheme could be used to alleviate this, though it performs reasonably well as is.

v. Structure and Usage

Besides just adding new features, work was also done to add structure to the framework repository [5]. This first meant refactoring the project to be in the form of a typical Julia module such that the primary functions of the framework can be made available with a simple `using FrameworkDemo` call. The two primary actions of the framework, parsing and scheduling graphs, were put into easy-to-use functions.

Structuring the framework as a module then made it easier to create a flexible runner script that handles the whole process for the user. To emulate the use case of wanting to process many physics events in the same way, the user provides the description of a single graph which gets scheduled n times, with some limit on concurrently scheduled graphs. This is just the bare minimum for now, with options for ignoring algorithm runtime and rich logging capabilities planned for the future.

vi. Summary and Outlook

Though the implementations of data dependencies and CPU crunching described here both hit all of our goals, the former will have to wait for a future Dagger version which allows for asynchronous and flexible data dependency setups. This has however led us to having discussions with the `Dagger.jl` developers about the needs of HEP and our specific use cases. We hope this contributes to the creation Julia ecosystem fit for HEP software, both in Dagger and elsewhere.

Looking ahead, future directions to explore include dynamic scheduling based on control flow graphs, fleshing out the running script to grant more fine-tuned control over execution, and testing the framework on real heterogeneous systems. We would also like to work towards extending this demonstrator framework to schedule tasks on GPU hardware.

Acknowledgements

Thank you to Mateusz Fila and Benedikt Hegner for their mentorship and support this summer. Thank you to CERN and the Summer Student Programme for organizing such an enriching summer experience. Thank you to Myron Campbell, Junjie Zhu, and Steven Goldfarb of the University of Michigan CERN REU for hosting and supporting me. Funding for this experience was provided by the National Science Foundation, under award number PHY-2243608.

References

[1] Jonas Eschle et al. "Potential of the Julia Programming Language for High Energy Physics Computing". In: *Comput. Softw. Big Sci.* 7.1 (2023), p. 10. DOI: 10.1007/s41781-023-00104-x. arXiv: 2306.03675 [hep-ph].

[2] `Dagger.jl` *documentation*. Accessed: 2024-07-31. 2024. URL: https://juliaparallel.org/Dagger.jl/v0.18.12/.

[3] *Research of the Scheduling Mechanisms in Julia Dagger Library*. 2024. URL: https://indico.cern.ch/event/1402909/contributions/5987257/attachments/2878731/5042526/Julia%20research%20of%20the%20scheduling%20mechanisms%20With%20Dagger%20Library.pdf.

[4] G. Barrand et al. "GAUDI - A software architecture and framework for building HEP data processing applications". In: *Comput. Phys. Commun.* 140 (2001), pp. 45–55. DOI: 10.1016/S0010-4655(01)00254-5.

[5] *Framework repository*. 2024. URL: https://github.com/key4hep/key4hep-julia-fwk.