Analysis of the ATLAS Experiment using Cloud Technology

M.T.D.R.Dolan
School of Chemistry, University of Bristol.
(Dated: April 24, 2024)

The ATLAS experiment at CERN produces 1 petabyte of data a second, and so requires some of the most advanced data analysis systems on Earth. The objective of this project is to show how Docker containerisation can be used to scale analysis to a large cloud-based cluster. It is successful in its containerisation, and allows for parallel processing within a single node, but this is unsuccessfully deployed to a network of virtual nodes, although a method for doing so is described. Code is open-access on a GitHub repository [1].

1 Introduction

1.1 The Atlas Experiment

The ATLAS (A Toroidal LHC Apparatus) experiment is the largest conducted on the LHC at CERN. It collides two proton beams with high enough energy to produce particles of higher mass than any hitherto known, storing 10TB of data a day.

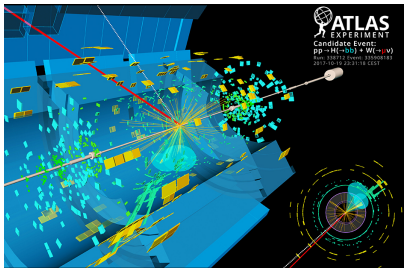

FIG. 1: Event display of a collision [2]

Due to the large processing power required to analyse this much data, the workload is split between multiple data centres around this world. The objective of this project is to explore how cloud computing methods can be used to achieve this. The method will focus on the H $\rightarrow$ ZZ $\rightarrow$ 4l decay, which represents the 'golden channel' used in the discovery of the Higgs boson [3]. It uses an already heavily processed dataset that has been filtered to only include at least 4 leptons per event [4].

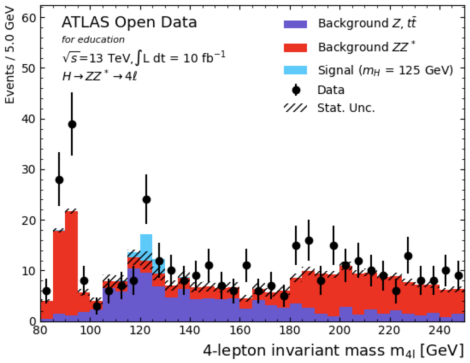

FIG. 2: A histogram of the fully processed data

2.1 The Initial Code

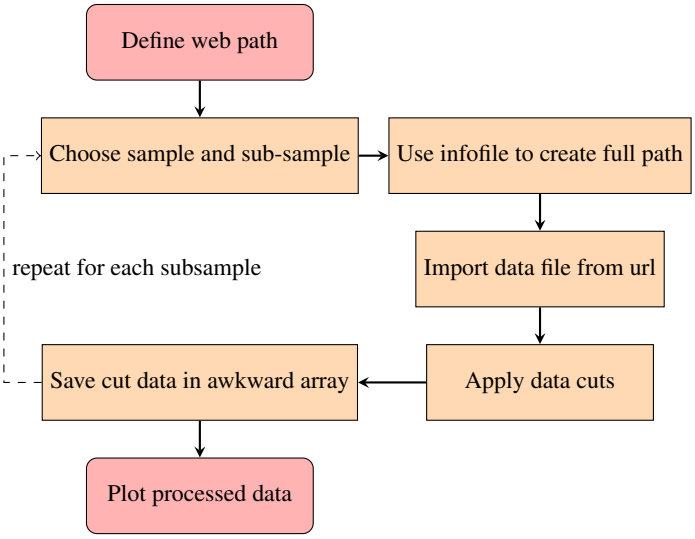

FIG. 3: Pipeline of Example Jupyter Notebook

A Jupyter Notebook [5] that performs the processing is provided by ATLAS opendata. The final processing performed here aims to increase the ratio of signal (H $\rightarrow$ ZZ $\rightarrow$ $llll$) to background (Z, $t\bar{t}$, ZZ $\rightarrow$ $llll$).

It does this by first defining a list of samples and subsamples manually to iterate over, which it does in the get_data_from_files function. This also uses keywords and information from the infofile to build the url to import the data from. This url is fed to a seperate read_file function which is able to import and open the data.

This data is then iterated over, with a series of cuts and weightings being applied in order to increase this ratio of signal to background and events that pass these cuts are saved in an awkward array. Finally the data is binned, and turned into histogram.

The total time it takes to process and plot the data is 140.0s. The final plot, shown in figure 2, is used as a benchmark that latter versions of the analysis process must recreate. While it functions correctly the code in this form is neither efficient nor scalable to a larger system due to the significant background involved in a Jupyter script.

2 Methods

The Jupyter notebook was first translated one complete python script, instead of multiple cells. In order to be able to split the analysis to work on multiple nodes, the task was split into the following three stages:

1. Split the process to allow for parallel processing
2. Create system to allow for automated containerisation of each of these processes
3. Test deployment across a system of multiple virtual nodes

2.1 Splitting Apart

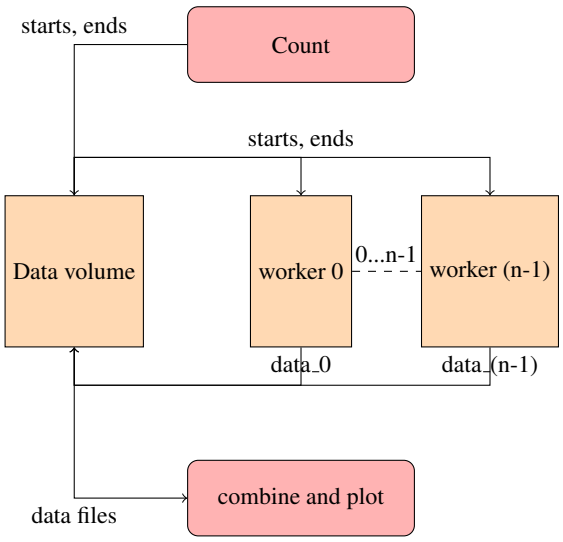

FIG. 4: Layout of split analysis

The most obvious way of splitting the analysis would be to process each sub-sample on a differnet node, but this fails to account for variation in the size of sub-samples. For example, data_A in the data samples has 39 events, whilst sub-sample *llll* in the background ZZ^* samples has 554,279. In addition to this, different sub-samples take different lengths of time per count, as shown in figure 5, so its also not feasible to just give the first 1000 of the total counts across sub-samples to the first node etc.

Therefore, in order to ensure each node has an equal workload, it is easiest to split each sample into equally sized loads that can be done on every worker. This could be done proportionally (i.e. worker 0 does 0-10%, worker 1 10-20% etc.) but it runs into the risk of either missing, or double-counting, events that occur on division borders.

Therefore an initial count script first counts how many events are in each sub-sample, taking around $\approx$ 10s so it is not too computationally expensive. This then creates dictionaries (according to how many workers there are) with a set of start points, and end points for each worker, for each sub-sample, making them as equally sized as possible while still ensuring every point is covered once and only once. These are exported as two pickle objects to a common data volume: a starts dictionary and an ends dictionary. This is also to ensure that no data is ever downloaded, and it is only ever imported within scripts so that it can be applied to as large data files as possible.

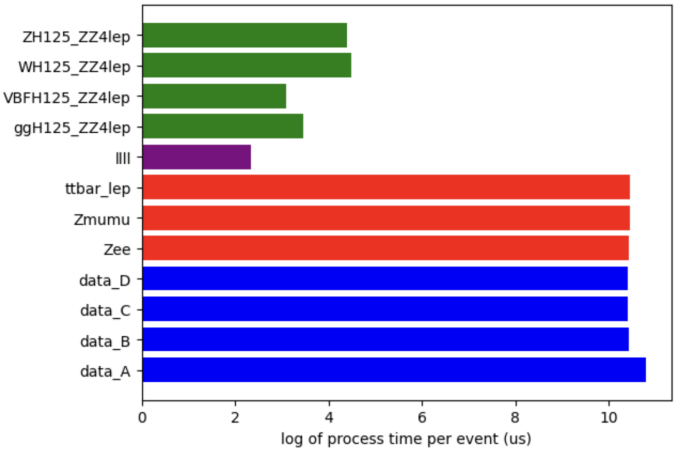

FIG. 5: The process time per event for the different samples

Each worker is then given a rank and imports the starts and ends dictionaries from the data volume. It picks its 'pair' of starts and ends according to its rank, and then processes the data by applying the relevant cuts. This data is then saved as another pickle object to the shared volume. Once all workers have finished, a final collector script can combine all the data files together, and plot.

At this stage, the analysis only acts as a template for transitioning to a multi-node system. It relies on too many dependencies within the system itself, as well as lacking a method of communicating between components beyond having a commonly accessible folder.

2.2 Containerising

Docker [6] enables programs and their dependencies to be wrapped up in one container. This means that components can be deployed to different nodes easily, and are self-contained in their own environments. It also contains extensive back-end support to help with multi-container programs.

In order to implement this several Dockerfiles were created, and files had to be arranged into a considered tree structure, as shown in figure 6. Each component, using its Dockerfile, is turned into an image that contains the script with all its dependencies. This image can then be used to create containers (multiple for the worker image) that can be run independently,

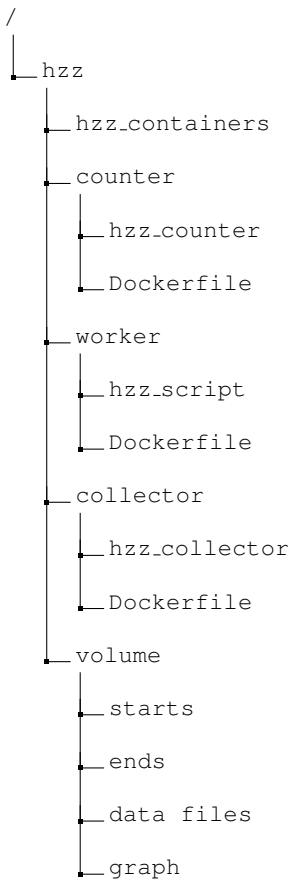

FIG. 6: Directory tree structure for hzz analysis

and provided, they are all mounted with the data folder as a volume, they can all share data.

Since only data files could provide a means of communication, and there was no easily implemented method of ensuring one container ran only once another had finished, measures had to be put into place to account for this. This consisted of conditional wait loops put at the start of worker and collector scripts so that they only ran once all the necessary data files were available (signifying that the previous component in the chain had finished).

Initially, it was attempted to use docker compose to build the images and run the images automatically. This worked for just one division, but the replicas function within it only produced exactly identical worker containers where they needed to be run with different ranks. Therefore, a bash script, hzz_containers.sh, was instead used to successfully allow for automatic set-up, and subsequent parallel processing of worker containers.

2.3 Deployment with Swarm

Testing deployment using docker swarm requires a few minor changes to the bash script. For a start, the number of divisions will need to be automatically detected based on the number of worker nodes, instead of a user input. In addition to this, components must be deployed as services, not containers, to specific manager and worker nodes. Both of these require the use of text search/manipulation commands within the bash script in order to draw out the correct node IDs.

The updated script, hzz_swarm.sh, was tested by deployment on a swarm created using Linux virtual machines. While there were no errors in the script, a security certification error in installing the Awkward package for just the collector service (not counter or worker services) prevented success. At the time of writing, this issue has not been rectified.

3 Results

Divisions	Div 1 (s)	Div 2 (s)	Div 3 (s)	Total time (s)
1	139.4	-	-	158
2	80.6	86.3	-	184
3	56.0	58.3	3362.4	3492
4	47.8	50.0	53.1	-

TABLE I: Times of non-containerised code divisions

Divisions	Div 1 (s)	Div 2 (s)	Div 3 (s)	Total time (s)
1	106.9	-	-	124
2	69.1	63.4	-	88
3	66.5	66.8	2449.7	2473
4	57.3	59.0	58.4	-

TABLE II: Times of containerised code divisions

In both the containerised and non-containerised tests, there is a clear trend towards a lower processing time for each division in an analysis with more divisions. Whilst the data taken fits to a parabolic curve better, as shown in figure 7, it is likely that with more data and for more divisions the trend would be exponential with the rate of decrease becoming smaller and smaller. Initially the containerised version runs quicker per division than the non-containerised version but the opposite is true after 2 divisions. Likely, it is because the containers initially act as a more streamlined environment for the worker component to run in but then any overheads (potentially in the container to volume connection) become a larger effect at shorter times. Overall, the containerised version still is much faster due to divisions being able to be run in parallel.

Curiously, for 3 divisions or more, the final division takes an exceedingly long time to run. For 3 divisions, the first and

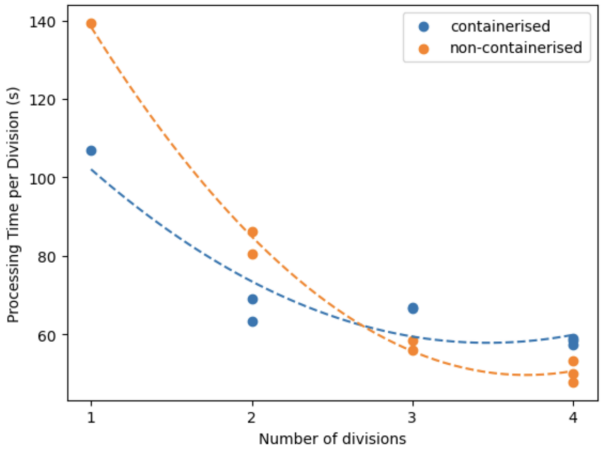

FIG. 7: Division times

second division takes around 60 seconds to run, but the final takes 41 minutes to run for the containerised, and 58 minutes for the non-containerised. When run for 4 divisions, the third division runs normally, but again the 4th and final division takes exceedingly long with overall tests in both cases cancelled due to excessive length. In all cases this only happens in the ggH125_ZZ41ep sub-sample of the m_H = 125 GeV sample, with every other sub-sample taking normal lengths to analyse. This could be down to some flaw within the data file that makes the final few events unsuitable for reading. Investigations to solve this will potentially include applying a similar system to other analysis and seeing if the flaw still exists. At time of writing, this was not able to be resolved

4 Conclusion

Whilst this project outlines a method of dividing the analysis amongst multiples nodes relatively successfully, several issues need to be addressed before full deployment can be achieved. Chiefly, the problem of the excessive processing time in the final division must be solved. The method for deployment over swarm should work in principle, but also has yet to be successfully shown and so there may be further issues there to be seen.

There are also several improvements in the method that can be made. For smaller numbers of nodes, the current divisioning method is sufficient but for a large server farm level setup a dynamic load balancing would be superior. For instance, if there are more available nodes than events in a sub-sample, then some nodes would have unequal workloads. In addition to this, currently the counting component imports the datafiles for counting separately to the worker component which imports the same datafiles. This is a potentially unnecessary repeat of a computationally costly process, structuring the process to instead distribute divided chunks of data may be quicker but considerations will have to be taken as to how large the data chunks can be.

4 References

[1] Code repository https://github.com/maximillian-dolan/atlas_analysis/tree/main
[2] US Department of Energy https://science.osti.gov/hep/Highlights/2018/HEP-2018-08-b (accessed April 2024)
[3] Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC, ATLAS collaboration, *Phys.Let.B* 2012, **716**, 1-29
[4] ATLAS Open Data, https://opendata.atlas.cern/docs/datasets/files/
[5] ATLAS collaboration, https://github.com/atlas-outreach-data-tools/notebooks-collection-opendata/blob/master/13-TeV-examples/uproot_python/HZZAnalysis.ipynb
[6] Docker, https://www.docker.com/