Distributed Cloud Computation For The Identification of The Higgs Boson Via ZZ Decay

Bryn Drury
School of Physics, University of Bristol.
(Dated: December 5, 2024)

The distributed application completed the analysis in approximately the same time as the original code. The original code completed the computation in 18.20 ± 1.31 seconds, while the distributed implementation took 20.03 ± 1.07 seconds when not using compression, a 10% increase over the original. Whilst using compression, it took 29.11 ± 1.36 seconds to complete (a 45.4% increase over no compression and a 59.9% increase over the original). The system does demonstrate how distributed systems can be used for high-throughput calculations.

INTRODUCTION

The identification process of the Higgs boson requires the computational analysis of large quantities of experimental data. The data used in this study is from the ATLAS open data programme and contains a mix of experimental data and simulated Monte Carlo data. The Higgs boson is being identified by identifying the events which follow the decay chain shown in equation 1. This equation shows the Higgs boson H decaying into two Z bosons, further decaying into four leptons, $\ell\ell\ell\ell$.

$$H \rightarrow ZZ^* \rightarrow \ell\ell\ell\ell \tag{1}$$

As the computation required for each event in the dataset is disconnected, this task is embarrassingly parallelisable. This independence, along with the potentially large quantity of data to analyse, makes it a prime candidate for using high-throughput computing techniques. This study will analyse and evaluate the application of cloud technologies to parallelise this task on a distributed cloud system.

This study will take advantage of the software Docker. Docker uses containers, which are simplified virtual machines, to ensure compatibility and scalability across devices. The tests were run on a Laptop with twelve cores and 18GB of memory, although Docker was given access to just eight cores along with 8GB of memory.

EXPERIMENTAL DETAILS

The code for this project is based on the code detailed in a notebook provided by the ATLAS Open Data Programme, from the ATLAS Collaboration[1].

The computational data analysis consists of two stages; the first is the cutoff, whereby events are filtered for relevance. These cutoffs identify the events with the correct number of leptons, and those with the proper charge ensure it is conserved. The second stage then calculates the invariant mass of the remaining events, and the Monte Carlo data calculates the events' weights.

The code in the reference notebook loops over all the data sources, retrieves the data from an external online source and then completes the computation. When taking a deeper look at the most time-consuming aspects of this process, it is clear that a large proportion of time is not just taken up by the analysis; it is also taken in downloading the data. This fact heavily impacted the design of the distributed computing system developed for this study. It was also clear that data retrieval and computation vary hugely across the sources. Some sources take less than a second to download and analyse, whilst others take upwards of 30 to 40 seconds.

The distributed cloud system developed for this study uses two Python scripts, a manager and worker scripts, run in their own Docker container. They communicate over a network using Pika, a RabbitMQ server, and a message broker, which runs in a separate container. The distributed application is run using Docker compose, a tool for running multi-container applications[2], this make the entire system of containers easy to manage.

This distributed implementation divides the tasks via the data sources. The manager creates a job queue and populates it with jobs. These jobs contain the relevant information to locate a given online data source. The manager also creates a results queue, which the workers use to return the computed data to the manager.

The worker takes a job from the job queue and then uses the information to download the data for the task. It then completes the required computations, as detailed previously and then adds the results to the results queue. During the development process, it was found that the results data array would exceed the message size for the RabbitMQ server channel. To resolve the issue, the maximum channel size was increased to account for larger result arrays. The option to compress the results before sending was also added if the results message sizes become even greater. However, this compression increases the computational overhead for the workers and the manager and hence is not enabled by default.

Once the job queue is empty, the manager deletes the job queue, which prompts the workers to shut down. It then goes through the results in the results queue, decompressing the data and formatting ready for the plotting stage. Once all the data is extracted from the results queue, the queue is deleted, and the manager calls the plotting function to produce the final figure.

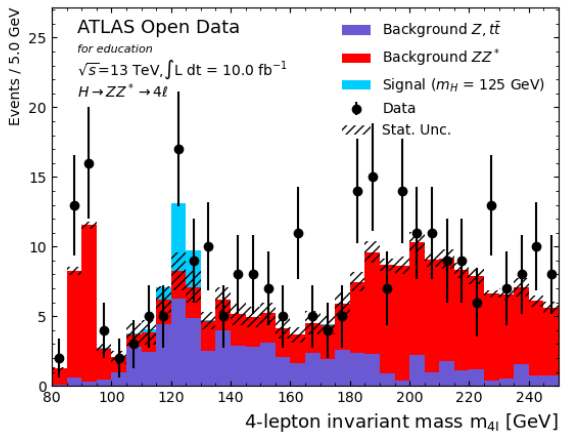

FIG. 1. The graph shows the event counts plotted against the invariant mass. With a peak around 125 GeV indicating the mass of the Higgs boson.

RESULTS AND DISCUSSION

This study aimed to replicate the outcome of the original code using a distributed computing implementation; the graph produced by the original code and the distributed system is shown in figure 1.

The computation in the original code took 18.20 ± 1.31 seconds. The distribution, computation and collection in the distributed implementation without compression took 20.03 ± 1.07 seconds. The implementation with compression took 29.11 ± 1.36 seconds, a 45.4% increase over no compression and a 59.9% increase over the original. In this example, the original code is slightly faster. However, this exercise is designed to demonstrate how the computation can be distributed across many computer systems. The data used also provides a worst-case scenario for the distributed system; many data packages being analysed are small, making the overhead of setting up the system and transferring the data between them proportionally greater.

As mentioned, a significant amount of time is spent downloading and collecting data from external online sources. When run serially, as in the original code, this will combine and extend the processing time significantly. In the case of a distributed system, the downloads can be completed simultaneously, meaning that the time for the processing is almost only reliant on the longest individual task. The small data packets in the current dataset mean that the overhead of the distributed system don't outweigh the benefits of being able to simultaneously execute the analysis, hence why the run-time increased.

If the dataset computed by a given worker becomes large, the results data may also become large; at this point, it might be helpful to enable the compression and decompression of the data when communicating between workers and managers. However, compression increases the computation required but significantly reduces the network traffic. Depending on the configuration of the distributed host system, it might require that the bandwidth or quantity of data be regulated; this would make the compression of the data viable. When the datasets being analysed become more significant, it might become beneficial to keep the data compressed either in memory or on file in storage to minimise the system requirements of the machine(s) running the analysis application.

The distributed implementation can be deployed on a networked system. Once the system has finished its computation, all the workers and the manager will shut down, leaving just the RabbitMQ server running. However, this can quickly and easily be shut down using the Docker GUI or CLI. The system requirements for running the RabbitMQ service are minimal, especially when no clients are connected, meaning it is not an issue to keep running. One could set up the system with another script to notify the user when the analysis process has been completed so they do not have to check the manager's progress manually. The results are saved in a persistent volume belonging to the manager, even once the application and its constituent containers have been closed. This persistent volume is where the plot is saved and can be accessed once the application has been completed.

The current distributed system implementation also heavily depends on the manager to do the last stint of the computation. Once the workers have computed the results, the manager is responsible for reformatting and plotting the results; this takes a couple of seconds and does not have to happen in the original code. This could be mitigated by having a secondary set of workers responsible for reformatting the data; for example, they could take the processed results and pre-bin the data ready for plotting, reducing the computation left to the manager.

CONCLUSIONS

The distributed application computed the computation in approximately the same time as the original code, even given that the given workload was not ideal for a system of its design. The original code completed the computation in 18.20 ± 1.31 seconds, whilst the distributed implementation took 20.03 ± 1.07 seconds when not using compression and 29.11 ± 1.36 seconds when using compression (a 45.4% increase). The graph in figure 1 shows the output of all implementations, as all produced the same results as expected. The distributed implementation demonstrates a valid architecture for the distributed system; however, as stated in the discussion, when the format and size of the data being processed change, alterations to the system will be required to best utilise the systems and their resources.

REFERENCES

[1] ATLAS collaboration. How to rediscover the higgs boson yourself! https://github.com/atlas-outreach-data-tools/notebooks-collection-opendata/blob/master/13-TeV-examples/uproot$_python/HZZAnalysis.ipynb$.

[2] Docker Inc. *Docker compose manual*. https://docs.docker.com/compose/.