



Master in High Performance Computing

# Optimizing the LIGO Summary Pages Pipeline

*Supervisor(s)*:

Gabriela González,

Irina Davydenkova

*Candidate*:

Iara Naomi Nobre Ota

# Abstract

The LIGO summary pages are web-based graphical summaries of the performance of the LIGO gravitational wave detectors and their many subsystems. These pages are essential for the ongoing monitoring and noise characterization efforts. Therefore, it is crucial to optimize the pipeline that generates these pages. In this work, we present a new workflow for the LIGO Summary Pages pipeline, which takes advantage of the High-Throughput Computing (HTC) paradigm to parallelize the processing and creation of the webpages. The new workflow was implemented by subdividing the process into smaller components that can be executed independently and concurrently on small compute nodes. With this new workflow, each process runtime was reduced to a median runtime of 10 $\pm$5 minutes.

# Acknowledgement

I am grateful to Professor Gabriela González, my supervisor at LSU, for her guidance and support throughout this work. I am also thankful for the invaluable opportunity to contribute to the LIGO collaboration under her mentorship. I would also like to express my gratitude to Dr. Irina Davydenkova, my supervisor at ICTP/SISSA, for her ongoing support, enlightening lectures during my time at ICTP, and for overseeing the MHPC program.

I am indebted to my colleagues at LIGO and LSU for their collaborative assistance. A special acknowledgment goes to Evan, Joe, and Robert from the Detector Characterization computing group, whose indispensable support significantly influenced the development of this work.

I would like to thank Dr. Ivan Girotto for directing the MHPC program and Manuela for her support during my time at ICTP. Moreover, I appreciate all the lecturers whose contributions enriched my learning experience during the program.

Heartfelt appreciation goes out to both old and new friends who made my time in Italy and in the USA more enjoyable, providing invaluable support along the way.

Finally, my deepest thanks to my parents, André and Inês, and my siblings Laura, Nádia and Iuri for their love and support. I am deeply grateful to Bruno for all his love, care and friendship throughout this entire period.

The completion of this thesis was made possible through the support of the ICTP Training and Research in Italian Laboratories (TRIL) program and I acknowledge support from the LIGO Scientific Collaboration, the Louisiana State University, the University of Wisconsin-Milwaukee and National Science Foundation grant PHY-2110594.

# Table of contents

# 1 Introduction

Gravitational Wave astronomy is a new and rapidly developing scientific field. Initially theorized by Einstein's General Theory of Relativity, the gravitational waves passing through Earth were first directly detected in 2015 by the Laser Interferometer Gravitational-Wave Observatory (LIGO). Today, a network of detectors has been established, including the LIGO facilities, the Virgo detector in Italy, and Japan's KAGRA detector. Together, these detectors form the LIGO-Virgo-KAGRA (LVK) collaboration, actively working to detect and understand gravitational waves.

The Theory of General Relativity describes gravity as curvature in spacetime, and gravitational waves are oscillations of spacetime. The gravitational radiation is transversal, meaning that the oscillations occur in the plane orthogonal to the wave propagation direction. The intensity of the deformations induced by a passing gravitational wave is determined by the properties of the source and the separation between the source and the observer.

Laser interferometers are instruments capable of detecting gravitational radiation by measuring the interference pattern of a laser beam as waves pass through the detectors. Figure 1.1 illustrates the principle of these instruments. A laser beam is split into two orthogonal beams, reflected by mirrors, and then recombined. The mirrors are separated by the same distance $L$ from the beam splitter, and the laser beam is tuned to interfere in a known pattern after bouncing back to the photon detector. When a gravitational wave passes through the detector, it causes the arms to stretch and squeeze, resulting in a difference in the interference pattern. This difference can be related to the gravitational waveform.

At a distance far away from the source, a gravitational wave will induce to an object with a length of $L$ a length variation of $\Delta L/L \propto G\ddot{I}/(c^4 r)$ [1], where $G$ represents the Gravitational constant, $c$ denotes the speed of light, $\ddot{I}$ is the second time derivative of the source's quadrupolar moment, and $r$ is the separation between the source and the object. The factor $G/c^4 \sim 10^{-45}\mathrm{s}^2/(\mathrm{m\ kg})$ indicates that the distortions caused by gravitational radiation are extremely small and the detections rely on "violent" events that create substantial disturbances in the quadrupolar moment. An example of such an event is the merger of binary compact systems, such as black holes and neutron stars. To date, these are the only systems detected by the LKV collaboration [2].

The first LIGO observation, GW150914 [3], involved a binary black hole merger with initial black hole masses $m_1 \sim 36M_\odot$ and $m_2 \sim 31M_\odot$, where $M_\odot$ is the Solar mass,

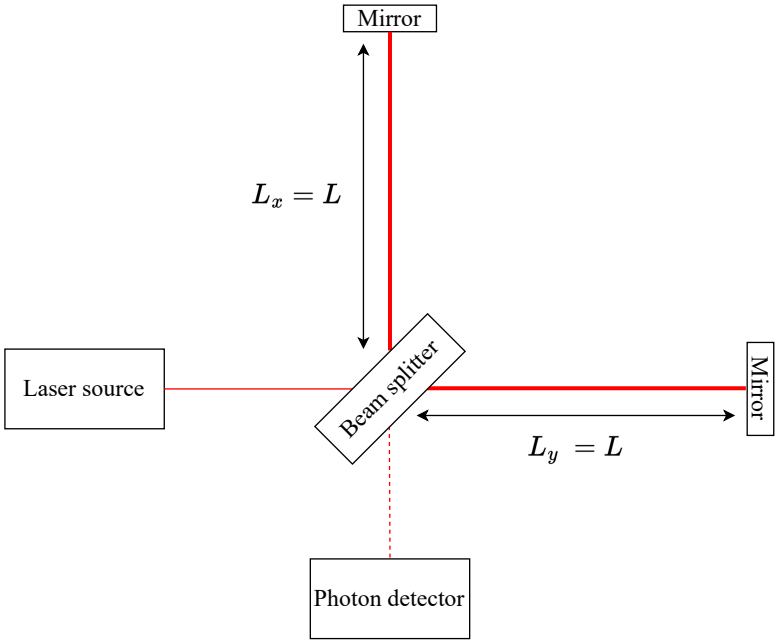


Figure 1.1: Laser interferometer diagram

and a distance of approximately 440 Mpc. The fractional length variation of LIGO's arms, which have a length $L = 4$ km, was $\Delta L/L \sim 10^{-21}$, which is the same order of magnitude as the ratio between the radius of a proton and the radius of the Earth. Therefore, laser interferometers are exceptionally sensitive detectors, and a thorough understanding of noise sources is essential for the operation of these instruments.

## 1.1 LIGO Summary Pages

Assessing the quality of the data obtained from the LIGO interferometers is a crucial step in signal detection. Various sources of noise, including thermal variations, seismic fluctuations, and even human activities, can significantly diminish the sensitivity of the instruments. A meticulous understanding and characterization of these diverse noise origins is highly important for the operation of these detectors.

The LIGO instruments incorporate a wide variety of sensors, such as photodiodes, microphones, seismometers and temperature sensors. These sensors' analog signals are converted into digital signals, which are then combined to create time series known as channels. Subsequently, the channels undergo post-processing to extract information that is analyzed by the scientists. The gravitational wave strain channel detects the distortions in spacetime, and it is used to reconstruct a passing waveform.

Constant work to enhance the sensitivity of the gravitational wave channel is a significant step towards improving detections. This involves efforts to comprehend and mitigate

the source of noises. Maintaining constant monitoring of the channels is essential for this goal, as issues detected by one channel could be strongly correlated to an increase in noise in the gravitational wave channel.

The LIGO Summary Pages provide an overview of key channels, which are analyzed by scientists on a daily basis. The `GWSumm` Python [4] package is responsible for processing the channels and generating webpages. These pages are organized into tabs, each holding plots and sometimes tables for different groups of processed channels.

The web pages are divided into five distinct categories. Specifically, there are two pages dedicated to the daily updates of the two LIGO detectors, Hanford and Livingston. These are the most comprehensive pages, each currently holding 214 tabs. The third daily page is titled the "Network" page and it has 42 tabs. This page offers information not only about the LIGO detectors but also includes details from the Virgo detector in Italy, and the German GEO600, and will soon incorporate information from KAGRA Japanese detector. Figure 1.2 shows a screenshot of the Network Summary page. The "Epoch" page encompasses information about the instruments' performance over an extended period, summarizing in 9 tabs the current observations run since its beginning. Furthermore, there's the Public page [5], which is publicly accessible and provides an overview of the key detectors' details. This page comprises 6 tabs for daily updates and one epoch page for each observing run.

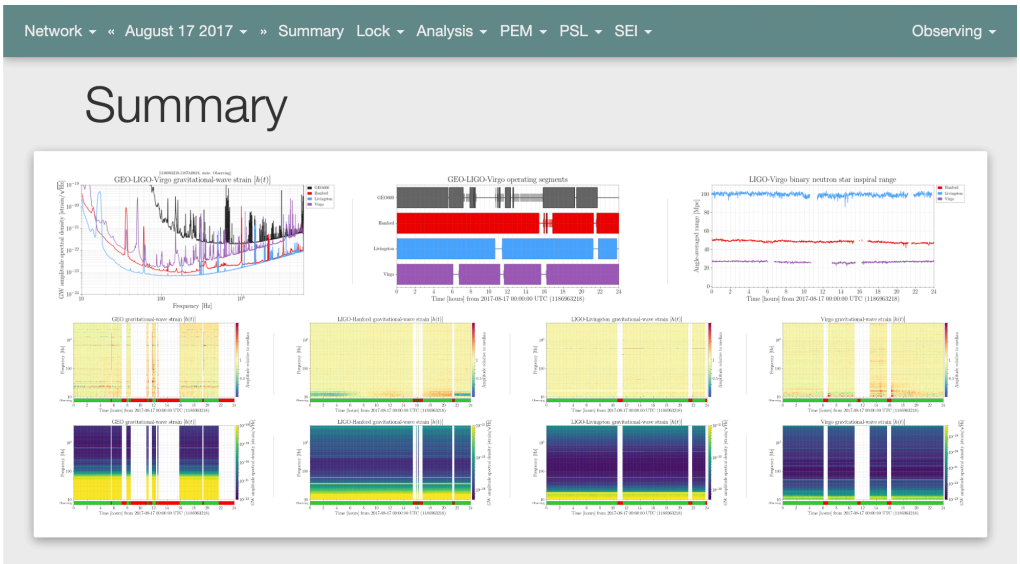


Figure 1.2: Network Summary Page.

Hence, the LIGO Summary Sages require a substantial computing effort. It is not only crucial to achieve nearly real-time updates, but it also needs to process a substantial volume of data. The optimization of the Summary Pages pipeline is essential to reduce the overall time needed for updating critical results obtained from the LIGO detectors.

# 2 High-throughput computing

Numerous scientific challenges require substantial computational resources for their resolution. High-Performance Computing (HPC) leverages supercomputers and computer clusters to address these challenges, emphasizing the acceleration of computations through metrics like Floating-Point Operations Per Second (FLOPS). The concept of High Throughput Computing (HTC) was introduced to acknowledge that many scientific applications prioritize the efficient long-term utilization of computer resources over raw computational speed [6]. We can also grasp this distinction by referring to the Open Science Grid's (OSG) definition of HTC as "the execution of computational work in the form of numerous, self-contained tasks to optimize their overall completion across available computing resources" [7], whereas HPC usually concentrates on resource-intensive tasks demanding tightly coupled parallel processing.

In the field of Gravitational Waves, obtaining the waveform solution for a binary black hole merger can be likened to an HPC problem. This waveform represents a solution to the Einstein Field Equations, which, in essence, involves solving ten coupled partial differential equations in a highly dynamical and non-linear regime. The computational demands are so substantial that solving the problem for a single binary system may take several months. The utilization of supercomputers and computer clusters can significantly reduce the computational time, making it an HPC challenge. On the other hand, this challenge can also be viewed through the lens of HTC. Characterizing gravitational waves relies on our understanding of the underlying theory, necessitating a vast number of simulations. In the long run, our goal is to efficiently utilize the available resources to run as many simulations as possible, highlighting the HTC aspect of the problem.

The LIGO Summary Pages pipeline, introduced in Section 1.1, is responsible for producing plots from various sensors within the LIGO instrument. This process continues indefinitely, generating daily pages. While our goal is to generate plots promptly, the HTC paradigm naturally fits into this pipeline. The plots from different sensors operate independently of each other and can be readily distributed.

The pages are hosted on the International Gravitational-Wave Observatory Network (IGWN) Computing Grid [8], maintained in collaboration with OSG. The IGWIN Computing Grid utilizes HTCondor [9], a specialized system for managing workloads that require significant computation.

## 2.1 HTCondor overview

In this section, we will provide a concise overview of HTCondor, referencing the HTCondor Manual [10] for detailed information. HTCondor enables the definition of discrete work units that need to be completed, distributing them effectively across the available resources. It provides advanced scheduling, prioritization, monitoring, and reporting capabilities. Here we refer to HTCondor simply as Condor.

### 2.1.1 Condor jobs

In Condor, a *job* is described as an "atomic unit of work." Typically, a job can utilize multiple CPU cores, but it executes on a single machine. Condor's design allows it to efficiently handle a large number of jobs. Consequently, the initial step in utilizing Condor involves breaking down the workflow into numerous jobs. These jobs can be interdependent through input and output files, but they operate asynchronously in relation to each other.

The `condor_submit` command is used to submit jobs and requires a submission description file, which contains all the necessary information for Condor to execute the job. This single submission description file can be used to run the same program multiple times with different arguments, inputs, outputs, and other parameters. The `executable` field in the description file specifies the main program that should be executed.

Consider having a bash script named `my_program.sh` that needs to run with various arguments. The list of arguments can be compiled in a file named `args.txt`. The script can be submitted through Condor using the following description file:

```
executable = my_program.sh
arguments = $(args)

log = path/to/logs/job.$(ProcId).log
error = path/to/error/job.$(ProcId).err
output = path/to/output/job.$(ProcId).out

queue args from args.txt
```

In addition to executing the script with the specified arguments, this process also determines the standard output and error paths, as well as the Condor log, all associated with the unique job ID. While the example provided is straightforward, it effectively illustrates the job definition in Condor. Instead of having a single, large task that runs in parallel, the workflow is divided in such a way that a single executable can be used to execute the same job multiple times with varying arguments.

Some useful commands include `batch_name`, which sets a user-defined name, `max_materialize`, which limits the number of jobs running concurrently, and `request_cpus`, `request_disk`, and other `request_<name>` to specify the machine resources required for the job. An extensive list of submission description file commands is available in the Condor manual [10] [1].

To execute jobs periodically, Crontab commands can be used. They share syntax similarities with the Unix cron daemon. In the following example, the command runs `my_program.sh` every minute with the same argument, `some_argument`, and saves the output to a file named `out_<num_job>.out`, where each job has a distinct number identifier. The `on_exit_remove` command ensures that the job will continue to run.

```
executable = my_program.sh
arguments = some_argument
on_exit_remove = false
cron_minute = *
cron_hour = *
cron_day_of_week = *
cron_day_of_month = *
cron_month = *
log  = log_$$([NumJobStarts]).log
error  = err_$$([NumJobStarts]).err
output  = out_$$([NumJobStarts]).out
queue
```

One final crucial command is `universe`[2]. This command specifies the execution environment for the job. The default is the *vanilla* universe, where jobs run in parallel within the "pool." In Condor's context, "pool" usually refers to a localized collection of resources, often within a single organization. Another option is to run jobs in the *grid*, allowing Condor to operate in the computing grid, which typically spans multiple organizations and geographical locations. If the job needs to run on more than one machine, such as MPI jobs, the *parallel* universe should be used. For jobs running on the machine where the job is submitted, the *local* universe is used. Very lightweight jobs can be executed immediately using the *scheduler* universe. Additionally, Condor allows jobs to run in virtual machines and containers, for which the *java*, *vm*, *container*, and *docker* universes can be used.

[1] To enhance the clarity of this presentation, we have altered the channel names in this example. Please note that the modified channel names are as follows: change `L1:GRAVITATIONAL_WAVE_CHANNEL` to `L1:GDS-CALIB_STRAIN_NOLINES` and `L1:RANGE_FRAME` to `L1:DMT-SNSL_EFFECTIVE_RANGE_MPC` for accurate representation.

[2] https://htcondor.readthedocs.io/en/latest/users-manual/choosing-an-htcondor-universe.html

### 2.1.2 Condor DAGs

Multiple jobs can be organized within a workflow using a directed acyclic graph (DAG). In graph theory, a DAG consists of nodes connected by directed edges that do not form closed loops. In the context of computing, the nodes represent jobs, and the directed edges signify the dependencies between the jobs. The acyclic nature of a DAG ensures that a parent job does not depend on a child job.

Consider the following example DAG file named `my_dag.dag`:

```
JOB A A.sub
JOB B B.sub
JOB C C.sub
JOB D D.sub
JOB E E.sub
PARENT A CHILD B C D
PARENT B C D CHILD E
```

The DAG directory for this example should contain the following files before submission:

```
A.sub B.sub C.sub D.sub E.sub my_dag.dag
```

In the structure of the DAG file, the jobs are organized as depicted in Figure 2.1.

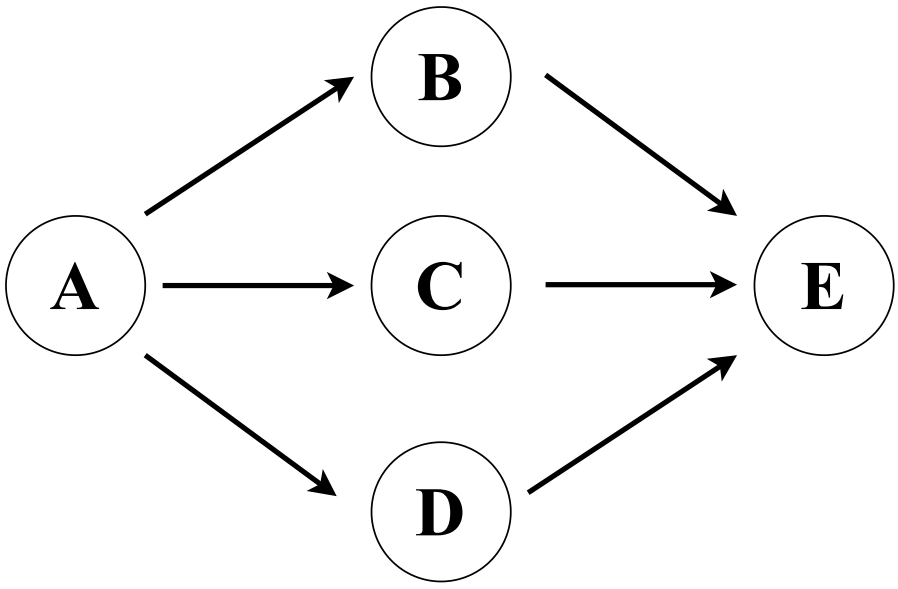


Figure 2.1: DAG example.

In this structure, job C depends on jobs B1, B2, and B3, all of which, in turn, depend on job A. The DAG file guides Condor in determining the correct order for job submission,

considering these dependencies. To execute the DAG, the `condor_submit_dag` command is used with `my_dag.dag` as an argument. Upon successful execution of the DAG, the following status files will be generated in the DAG directory:

```
my_dag.dag.condor.sub my_dag.dag.dagman.log
my_dag.dag.dagman.out my_dag.dag.lib.err
my_dag.dag.lib.out my_dag.dag.nodes.log
my_dag.dag.dagman.metrics
```

`*.dagman.out` provides detailed logging of the DAG run, `*.condor.sub` and `*.dagman.log` detail the submission job process created by DAGMan and its corresponding log, `*.lib.err` and `*.lib.out` contain standard error and standard output for the DAGMan job process, `*.nodes.log` provides a consolidated log for all jobs within the DAG, and `*.dagman.metrics` presents a summary of DAG statistics.

To monitor running DAGs and jobs, the `condor_q` command can be used. `condor_q` will display the DAG in a single row, indicating the total number of jobs the DAG is running. For more detailed information about each job's status, the `condor_q -nobatch` command is used. This command displays individual jobs as they run, offering comprehensive information about their status.

If a node job within the DAG encounters a failure, DAGMan will proceed with the execution of all other jobs that are not dependent on the failed node. It is considered a best practice to assign a single process per submit file. This approach ensures that the failure of one process does not trigger the failure of other processes, and the retry of a failed process does not interfere with the execution of successful ones.

### 2.1.3 Running a pipeline periodically

A pipeline, typically organized using DAGs, can be scheduled to run periodically by configuring a Condor job. You can create a bash script named `dag_executer.sh` to execute the DAG using the condor_dag_submit command, and accompany it with the following submission description file:

```
universe = local
executable = dag_executer.sh

batch_name = "Daily Pipeline Run: $(ClusterId)"

log  = log_$$([NumJobStarts]).log
error  = err_$$([NumJobStarts]).err
output  = out_$$([NumJobStarts]).out
```

```
on_exit_remove = false
cron_minute = 00
cron_hour = 00
cron_day_of_month = *
cron_month = *
cron_day_of_week = *

queue
```

The decision to run in the local universe is based on the fact that this job is essentially orchestrating other jobs. Running it locally ensures consistency with user submissions, but the jobs run in any universe designated for the pipeline. In the above example, the pipeline is scheduled to run daily at 00:00. Establishing a fixed periodic schedule is a good practice, promoting predictability, resource management, and facilitating troubleshooting in case of errors.

# 3 LIGO Summary Pages Pipeline

## 3.1 Current workflow

The `GWSumm` is a Python package [4] jointly developed by a collaboration between LIGO and GEO600. It serves the purpose of fetching the data frames associated with instruments' channels, processing the time series into visual plots and tables, and generating HTML pages. The main output of a `GWSumm` process is a *tab*. Each tab corresponds to an individual HTML page that includes any variety of plots and other data set by the user.

The LIGO Summary Pages, introduced in Section 1.1, are generated using the `GWSumm` package. Within the LIGO collaboration, an internal package named `ligo-summary-pages` is maintained. This package includes configurations in INI format that specify which data should be fetched, how it is processed, and how it is presented on the LIGO Summary Pages website. These configurations are operational within the detectors' production environment and constitute the main component of the LIGO Summary Pages.

The production workflow consists of Condor DAGs that create a single job for *each* configuration file. Within the LIGO Summary Pages pipeline, there are five distinct DAGs in operation.

The "HTML" DAG is responsible for generating the HTML pages. It creates a job that does not involve data processing. The "Fast" DAG includes the configuration of the main tabs that require prompt updates. This includes processed data from the gravitational wave channel and the status of the detectors. The "Slow DAG" encompasses all other tabs. while these tabs are required for the daily analysis, they are not as critical as the "Fast" DAG tabs.

The "Rerun" DAG operates on the following day and addresses any potential gaps that might exist due to latency issues. Lastly, the "Clean" DAG is responsible for removing cached data stored in HDF5 format that is over 40 days old. This step serves to decrease the disk space usage. Figure 3.1 provides a visual summary of this pipeline.

As previously mentioned, a single configuration file groups similar tabs together. This results in certain configuration files fetching hundreds of channel data frames and processing tens or even hundreds of plots. This concentration of tasks within a single process


Figure 3.1: LIGO Summary Pages pipeline

demands a substantial amount of memory, which imposes limitations on the nodes where these jobs can be executed.

Due to the high memory usage, the summary pages pipeline operates within the local universe, specifically making use of the access point allocated to the Detector Characterization working group. This login node is equipped with two 2.4 GHz Xeon E5-2630v3 CPUs and 128 GB of RAM. This setup is enough to accommodate the summary pages processes, but it does imply that most of the jobs must be executed sequentially.

Parallelism among configuration files is possible across different DAGs. This implies that all five DAGs can be executed concurrently. This clarifies the label "Slow" for the DAG that covers the majority of tabs. These tabs are processed sequentially, which considerably delays the completion of the jobs within this DAG.

### 3.1.1 Parallelizing the workflow

The goal of this project is to minimize the time it takes to generate LIGO Summary Pages. The strategy to achieve this goal involves breaking down the memory-intensive configuration files into smaller files. This approach enables the jobs to take advantage of the numerous available compute nodes.

Having one configuration file for each tab is not an optimal approach. In cases where a tab executes quickly, the time delay between job submission and the start of execution could become significant in comparison to the actual execution duration. Therefore, the first step in the project is assessing which tabs should remain grouped and which ones should have their own process.

The execution time of a tab should depend on the number of channels it is using and the type of processing it undertakes on those channels. For instance, generating a spectrogram, which displays the frequency power as a function of time, requires multiple

Fourier transforms of the data and it is significantly more computationally intensive compared to generating a time series plot of the raw data. To identify which are the most time-consuming tabs, we can directly analyze the configuration files.

#### 3.1.1.1 Structure of the configuration files

The LIGO Summary Pages configuration files are formatted in INI style. The acronym INI stands for "initialization", and INI configuration files are composed of *sections* which organize the *properties*.

Each property is presented in a key-value format, separated by an equal sign:

```
key = value
```

Properties may be grouped in sections, although this is not necessary. Sections are arbitrary groups for the properties, and they are defined using square brackets:

```
[section_name]
key_1 = value_1
key_2 = value_2
```

There is no method to explicitly end a section, they either end at the end of the file or at the introduction of the next section. All the properties defined after the section declaration are considered associated with that section. If the same `[section_name]` is specified in two or more configuration files, the latest loaded file will overwrite the keys it defines from the other configuration files. In the LIGO Summary Pages context, we are interested in three main sections: tab, channel and plot.

The *tab* section is defined using the syntax `[tab-<tab_name>]`, where the `tab-` prefix serves as an identifier for `GWSumm` to recognize this as a tab configuration. `<tab_name>` is a customizable string that uniquely designates the particular tab to be generated.

The main configuration keys for dealing with the organization of the tabs' sections are the following: the `name` and `shortname`, which assign labels to the tab, the `parent` key categorizes the tab and also serves to create the dropdown menu in the top of the page. For example, in the Public Page [5] "Home", "Summary", "Environment" and "Instrument performance" represent `parent` values.

At a lower hierarchical level, there is the `group` key, which organizes similar tabs within a `parent` dropdown menu. The `layout` key dictates the arrangement of the plots on the pages. The `states` key determines the instrument status displayed on the page.

The plot data is defined using an integer key, determining the display sequence of the plots. Each integer corresponds to a specific plot. To illustrate, the first plot is `1 = some plot`, the second is `2 = another plot` and so forth. This sequential arrangement allows for a structured presentation of the plots.

These properties have various child properties that define specific plot options. These child properties consist of functions from `matplotlib.pyplot` and plot parameters. Each child property is identified by an integer followed by a property name. For instance, the title of the first plot is specified with the key `<integer>-title`. Other examples include `<integer>-color`, `<integer>-labels`, `<integer>-linestyle`, and `<integer>-linewidth`. It is important to ensure that the values assigned to these child keys are compatible with the requirements of the Matplotlib library.

The values for the `<integer>` key can be either the path of a plot or a list of channels followed by their associated plot type (the plot types will be explained later, as they constitute a distinct section in the configuration files). By analyzing the `<integer>` key within each configuration file, we can determine how many channels are being processed and the type of plots that a specific configuration is generating.

For every channel listed as a value under the `<integer>` key, a corresponding entry should be present in a *channel* section. Mirroring the structure of the tab section, a channel section is specified as `[channels-<channel_group>]`. In this format, the prefix `channels-` is recognized by `GWSumm` as a channel section, while `<channel_group>` is a user-selected string that characterizes a group of channels sharing the same configurations.

The primary key in this section is the `channels` key, whose values constitute a list of channels. Additionally, this section specifies aspects of the channel data, such as `frametype`, `unit` and `frequency-range`. Furthermore, some keys in this section establish specific processing configurations, such as the power spectrum parameters like `stride`, `fftlenght` and `overlap`.

The final important section in our analysis is the *plot* section, defined as `[plot-<plot_category>]`. Once again, the prefix `plot` is required by `GWSumm`, while `<plot_category>` is a string defined by the user. The string `<plot_category>` must align with the value attributed to the `<integer>` key in the tab section. It is essential that the same `<plot_category>` string is used consistently.

The majority of keys within the plot sections are `matplotlib.pyplot` functions and plot parameters. These values are overwritten in case they are defined as a child key for the `<integer>` key in the tab section. The main key in this section is the `type`, which specifies what kind of plot should be created. Among important values considered here include `spectrum`, `spectrogram` and `timeseries`.

The code block below gives an illustrative example of how a tab named "Strain" might be configured [1]. This tab would encompass two plots for the Livingston detector. The first plot displays the power spectrum of the gravitational wave channel, while the second

[1] To enhance the clarity of this presentation, we have altered the channel names in this example. Please note that the modified channel names are as follows: change `L1:GRAVITATIONAL_WAVE_CHANNEL` to `L1:GDS-CALIB_STRAIN_NOLINES` and `L1:RANGE_FRAME` to `L1:DMT-SNSL_EFFECTIVE_RANGE_MPC` for accurate representation.

plot shows the binary neutron star range. This range represents the furthest distance, averaged over sky location, at which the gravitational waves emitted from binary neutron star systems can be detected. This range is computed from the noise present in the gravitational wave channel.

```
[tab-strain]
name = Gravitational-wave strain
shortname = Strain
1 = L1:GRAVITATIONAL_WAVE_CHANNEL plot-spectrum
1-label = L1 Strain
1-color = '#4ba6ff'
1-title = r'LIGO Gravitational-wave strain'
1-ylim = 1e-24,1e-19
1-label = L1 Strain

2 = L1:RANGE_FRAME timeseries
2-title = L1 binary neutron star inspiral range
2-yticks = 0,20,40,60,80,100,120,140,160,180
2-yticklabels = %(2-yticks)s
2-ylabel = Angle-averaged range [Mpc]
2-color = '#4ba6ff'
2-all-data = True
2-label = L1 Range

[plot-spectrum]
type = 'spectrum'
xlabel = 'Frequency [Hz]'
xscale = 'log'
ylabel = r'GW amplitude spectral density [strain/%(rtHz)s]'
yscale = 'log'
legend-loc = 'lower right'
legend-fontsize = 11
legend-handlelength = 1

[channels-hoft]
channels = L1:GRAVITATIONAL_WAVE_CHANNEL
unit = 'strain'
frequency-range = 10,5000
asd-range = 1e-24,2e-18
stride = 60
fftlength = 4
overlap = 2
```

```
[channels-range]
channels = L1:RANGE_FRAME
amplitude-range = 0,180
unit = 'Mpc'
```

The resulting page generated using this configuration file is displayed in Figure 3.2.

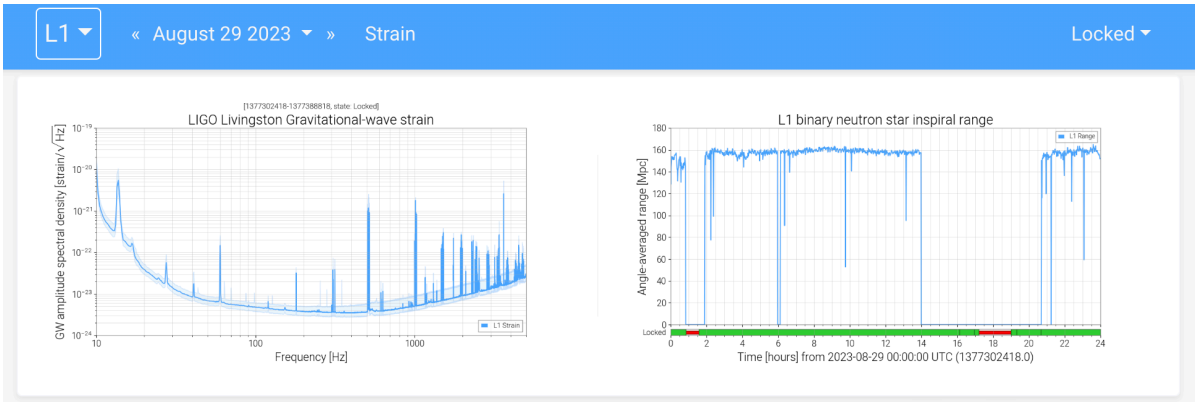


Figure 3.2: Example of a LIGO summary page

#### 3.1.1.2 Breaking down the configuration files

When we divide the configuration files, we consider three critical factors: processing time, the processed data, and their interdependence. These elements are intricately linked. For instance, consider a tab that includes a time series, a power spectrum plot, and a spectrogram for a specific channel. If we contemplate splitting this configuration by isolating the spectrogram (which is the most time-consuming plot) from the other plots, it might appear to enable parallelization and faster computation. However, all these plots access the same data. This results in two processes attempting to retrieve the same data from the disk, and also leads to increased memory consumption, as the same data is fetched twice, and potential inefficiency. Although the processing of a spectrogram is technically independent of the processing of a power spectrum, we treat them as interdependent because they operate on the same dataset.

Hence, our initial approach in refactoring is to create separate files with non-redundant channels. This means that the `channels` key in the `[channels-<channel_group>]` section should have unique values for each new configuration file. In most cases, creating plots for a single channel doesn't put significant demands on computational resources, allowing us to include multiple tabs in a single configuration file. Once we've identified the independent channels, we proceed to divide the configuration file to reduce the computation time to at least under 30 minutes. This often entails splitting the tabs with the more time-consuming plots into separate files.

Additionally, we prioritize the relationships between grouped pages, aiming to ensure that they are logically organized within the same category. This approach significantly enhances the maintainability of the code.

#### 3.1.1.3 Identifying the most time-consuming configuration files

While memory usage is the key constraint for running jobs in the vanilla universe, it's important to note that job runtimes are directly proportional to RAM consumption. The most time-consuming jobs, responsible for reading and processing larger volumes of data, naturally require more memory. Moreover, reducing runtime is a critical aspect of the pipeline to ensure that no pages lag the updates.

The most straightforward method for identifying the most time-consuming configuration files is to examine the `*.dag.nodes.log` of a full-day run DAG, which includes the running time of each job within the DAG. This file follows a consistent structure for each job:

```
000 (<job_id>) YYYY-MM-DD hh:mm:ss Job submitted from host:
<host_information> DAG Node: <job_name>
...
001 (<job_id>) YYYY-MM-DD hh:mm:ss Job executing on host:
<host_information>
...
005 (<job_id>) YYYY-MM-DD hh:mm:ss Job terminated.
    (1) Normal termination (return value 0)
        Usr 0 hh:mm:ss, Sys 0 hh:mm:ss  -  Run Remote Usage
        Usr 0 hh:mm:ss, Sys 0 hh:mm:ss  -  Run Local Usage
        Usr 0 hh:mm:ss, Sys 0 hh:mm:ss  -  Total Remote Usage
        Usr 0 hh:mm:ss, Sys 0 hh:mm:ss  -  Total Local Usage
    0  -  Run Bytes Sent By Job
    0  -  Run Bytes Received By Job
    0  -  Total Bytes Sent By Job
    0  -  Total Bytes Received By Job
...
```

This structure is maintained for all other jobs within the DAG and is appended to the same file. The line that starts with `000` provides the name of the job following `DAG Node:`, allowing us to associate it with a `<job_id>` and identify the configuration file. The line that begins with `005` includes the `<job_id>` and the runtime details. Of particular interest is the "Run Local Usage," which denotes the time it took for the job to complete.

Figure 3.3 shows the runtimes for each configuration file for a full-day run of the LIGO Livingston Summary Pages. The total runtime for this period was 14 hours and 57

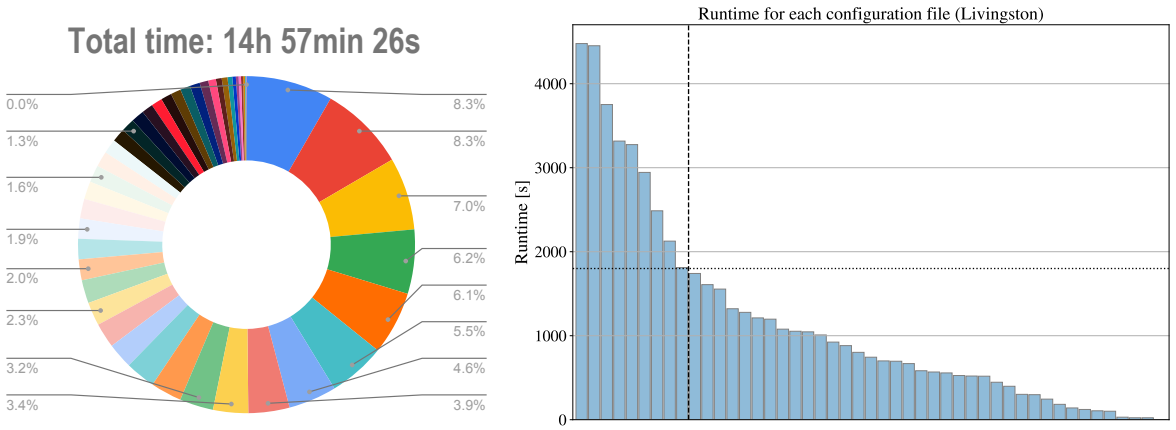


Figure 3.3: LIGO Livingston Summary Pages runtimes: *Left:* runtime percentages of each configuration file. *Right*: runtime in seconds of each configuration file. The dashed vertical line denotes the point where the cumulative runtimes to the left exceed 50% of the total time. The horizontal dotted line indicates 30 minutes.

minutes. The dashed line on the right bar plot highlights when the runtimes of the most time-consuming configuration files collectively exceed half of the total runtime, encompassing 8 out of 46 configurations. Additionally, the vertical dotted line at 30 minutes indicates that these 8 files also surpass the 30-minute mark. This suggests that the majority of the files do not require any refactorization to run in the vanilla universe.

## 3.2 Results

The runtimes displayed in Figure 3.3 provide a valuable overview of the configuration files that require attention. However, it's important to note that these runtimes might be subject to overestimation or underestimation since they are based on a single run. Furthermore, this run was necessitated by a cluster failure, leading to a full-day summary page rerun. The process took place on the login node allocated to the Detector Characterization group, running alongside the other ongoing DAGs that consistently run on this node.

In the benchmarking process, we use a separate node designated for development purposes. This particular node is equipped with four 3.0GHz Xeon Gold 6154 processors, providing a total of 72 cores and 1.5TB of RAM. All benchmarked jobs in this analysis were executed using 8 cores. It is worth noting that the varying factor among these jobs primarily lies in the configuration files, as the executable used is a `GWSumm` process.

### 3.2.1 Benchmark setup

To benchmark the jobs effectively, we need to conduct multiple runs for each process and subsequently calculate the average results. This method ensures statistical reliability and helps mitigate the impact of outlier results. Multiple runs can be easily managed using a simple Condor submission description file.

The executable is common to all jobs, which we named `run_summary.sh`

```
#! /bin/sh
CONFIG_FILE=$1

# This is the path containing the configurations
SUMMARY_CONFIG_PATH=/home/<user>/configurations

# Sets up the defaults configurations, common to every configuration
FILES=${SUMMARY_CONFIG_PATH}/defaults.ini,
FILES+=${SUMMARY_CONFIG_PATH}/common/global.ini
# Sets up the common configuration file
FILES+=${SUMMARY_CONFIG_PATH}/common/${CONFIG_FILE}.ini,
# Sets up the detector specific (Livingston) file
FILES+=${SUMMARY_CONFIG_PATH}/l1/l1${CONFIG_FILE}.ini

# define the output directory of the GWSumm process
OUTPUT=/home/<user>/public_html/summary

# Run GWSumm in Livingston detector (--ifo L1)
# the day 26/08/2023 was arbitrarily chosen
python -m gwsumm day 20230826 --multi-process 12 --ifo L1
--output-dir ${OUTPUT} --config-file ${FILES}
```

Running `./run_summary.sh <configuration_file_name>` will execute GWSummm for the date 26/08/2023, which was arbitrarily chosen. To perform this process multiple times and across various configuration files, we have set up the following Condor submission description file:

```
# running in the local universe to compare from the same machine
universe = local

# The executable is common to all configurations
executable = run_summary.sh
# the arguments change according to configs.txt
# see last line
```

```
arguments = $(config)

# give a name to identify in the job in `condor_q`
batch_name = "$(config) ID: $(ClusterId)"

log     =   log/$(SUBMIT_TIME).$(config).$(ProcId).log
error   =   log/$(SUBMIT_TIME).$(config).$(ProcId).err
output  =   log/$(SUBMIT_TIME).$(config).$(ProcId).out

# Limits the number of concurrently executing jobs
max_materialize = 6

# get the arguments from configs.txt
# runs 15 times the same argument
queue 15 config from configs.txt
```

`configs.txt` simply contains a list of the configuration file names. We chose to run the processes in the local universe to ensure consistent comparisons on the same machine. The description above will execute each configuration file in `configs.txt` 15 times, with a limit of 6 concurrently executing jobs to prevent machine overload. For the original configuration file, we will set up a special submit file with `max_materialize = 1`.

### 3.2.2 RAM usage

The initial configuration file we divided up handled a total of 194 channels, including 194 power spectrum plots and 51 spectrograms. This configuration had 15 distinct tabs defined within it. To facilitate effective organization, we divided it into groups of 2, resulting in the creation of 8 new configuration files. This decision was influenced by the physical location of the sensors associated with each tab. The sensor groups within the split are geographically separated and independent from one another.

Figure 3.4 depicts the RAM usage of the original configuration (black) and the new configuration files (colored) over time. To gather this memory data, we utilized the Memory Profile Python package [11]. The plot is based on information from 30 different runs for each configuration file, with the thick lines representing the average of the corresponding curves.

The plot clearly illustrates that the original process experienced a continuous increase in RAM usage. Each "step" in the RAM usage corresponds to the introduction of a newly created configuration file. This consistent rise in memory consumption is attributed to `GWSumm` accumulating loaded data in memory throughout the process. Our results demonstrate that a well-structured division of the configuration files can significantly reduce the RAM usage of a `GWSumm` process.

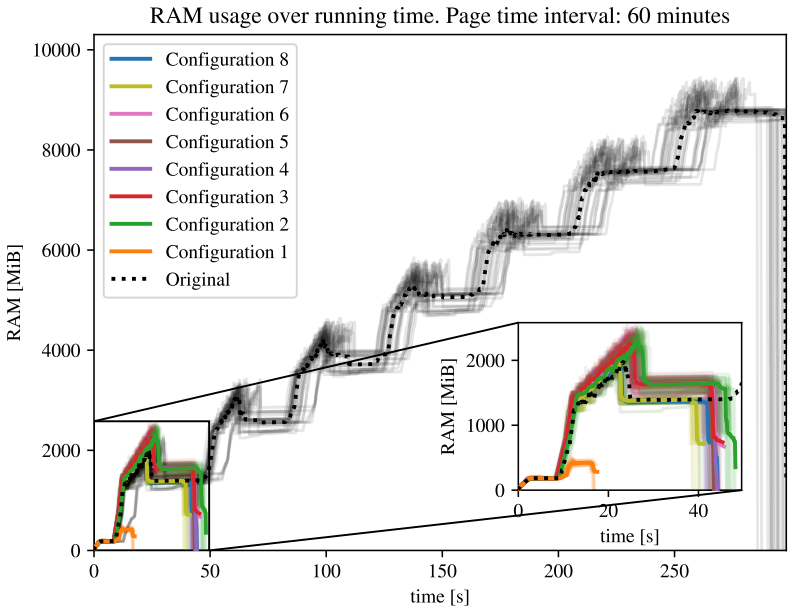


Figure 3.4: RAM usage report for `GWSumm` jobs. The black lines represent the original configuration file, while the colored curves correspond to the eight new configuration files.

Figure 3.4 displays the memory profile tabs generated for a 60-minute interval, containing sensor data accumulated over this duration. It is important to note that our system runs daily summary pages, with a 24-hour interval. Figure 3.5 shows the RAM consumption peak with varying data durations. The rightmost scatter points are the 24-hour interval.

In this plot, the solid straight lines represent a linear fit to the points (note that the plot is in log-log scale, hence the appearance of a different dependency). The data suggests that RAM usage increases linearly with the considered data interval. Notably, the RAM usage of the new configurations never exceeds 30% of the peak RAM usage observed in the original file (depicted as a gray dotted line).

Although the runtime shown in Figure 3.4 may not be the most accurate due to the memory profiler process, this section demonstrates the effectiveness of splitting a configuration file into several smaller ones. This division significantly reduces both memory usage and runtime, addressing the optimization requirements of the Summary Pages pipeline.

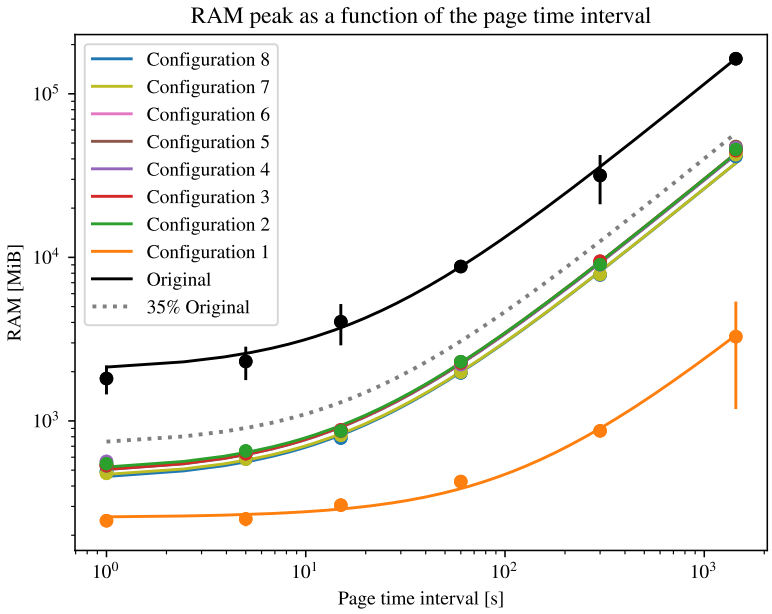


Figure 3.5: RAM peak for `GWSumm` jobs. The black lines represent the original configuration file, while the colored curves correspond to the eight new configuration files. The dashed gray line indicates 30% of the original (black) configuration file

### 3.2.3 Results benchmark

As the maximum RAM usage is the primary constraint for running jobs in the small compute nodes, we focus on determining the processes' peak RAM usage. Thus, there is no need to profile RAM consumption over time. In this section, we benchmark jobs using the GNU time command, `/usr/bin/time`, to extract both the peak RAM usage and the runtime of a job.

Based on the runtimes provided by Figure 3.3, we refactored the most time-consuming configuration files into smaller files. Each file was individually analyzed with the following goals in mind:

1. The runtime should be smaller than half an hour;
2. The peak of RAM should be smaller than 64 GB;
3. The files should be logically separated into related groups of tabs;
4. The files should have similar runtimes and memory consumption.

The logical rearrangement of the files is prioritized for the maintainability of the pipeline. For example, in Figure 3.4, "Configuration 1" in orange is clearly much lighter than the other configurations. As explained, the choice was made due to the physical location of the sensors in each configuration file.

Figure 3.6 displays the runtime of the refactored configurations. The eight most time-consuming configurations, as identified in Figure 3.3, are arranged along the horizontal axis, numbered in the same order as Figure 3.3. The black points represent the original. configuration files, while the colored points below represent the new files obtained by splitting up the originals. The horizontal dashed line represents the 30-minute constraint. The median runtime of the new configuration files is $601 \pm 283$ seconds, or $10 \pm 5$ minutes, as indicated by the dotted line and gray band in the plot.

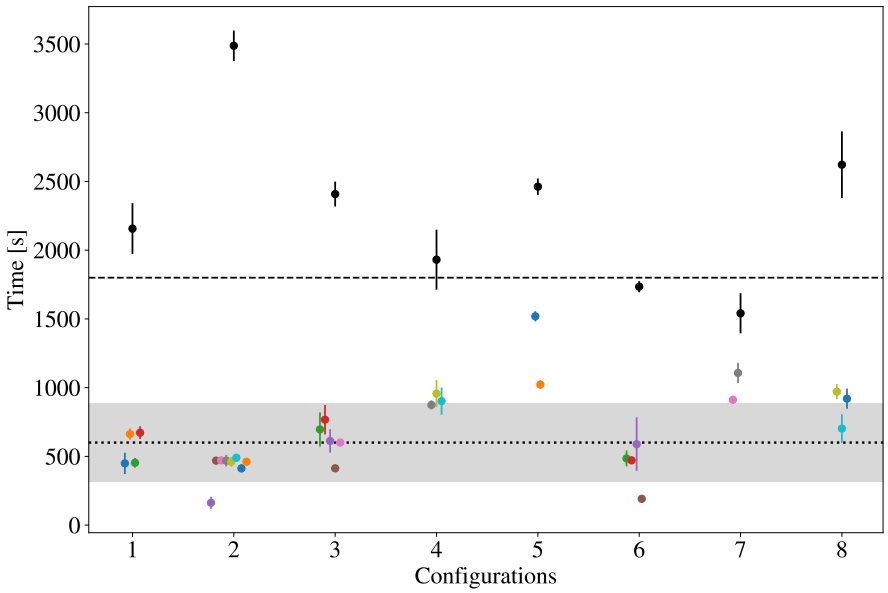


Figure 3.6: Runtime of refactored configurations. The horizontal axis represents different configurations, with black points indicating the original configuration and colored points representing the new refactored configurations. The horizontal dashed line indicates 30 minutes. The dotted line indicates the median runtime of the new files and the gray band is the standard deviation.

We notice that the differences in runtimes given by Figure 3.3 and Figure 3.6 (black points) are expected. First of all, the processes run on different machines. While Figure 3.3 was identified by a run in production in the Detector Characterization node, Figure 3.3 was obtained on a development node. Moreover, the development node was not busy when the benchmark was done, while the Detector Characterization node is always very busy running the Summary Pages pipeline and other tools. Finally, Figure 3.3 shows a single run, while Figure 3.3 has 15 runs for each configuration file, which should help identify changes in runtime within the error bars.

Figure 3.7 shows the peak of RAM consumption for the same configuration files as Figure 3.6. In this case, the dashed horizontal line indicates the 64 GB limit. The dotted line and gray band indicate the median RAM peak of the new configuration files and its standard deviation, $44 \pm 31$ GB. We observe that memory consumption is not trivially related to runtime, as some processes consume very little memory compared

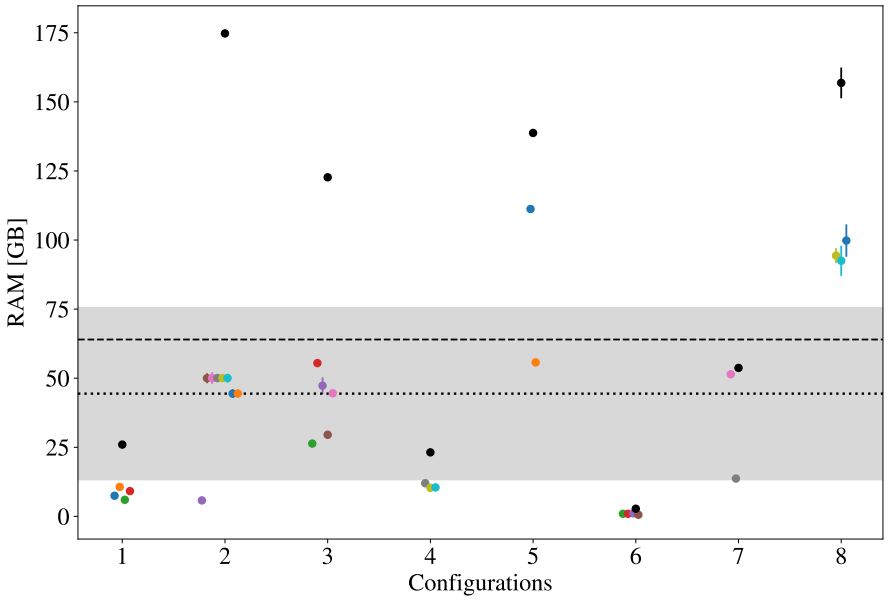


Figure 3.7: Peak of RAM of refactored configurations. The horizontal axis represents different configurations, with black points indicating the original configuration and colored points representing the new refactored configurations. The horizontal dashed line indicates 64 GB. The dotted line indicates the median RAM peak of the new files and the gray band is the standard deviation.

to others with a similar runtime. A key factor is how much data the process needs to load. For example, the configuration labeled as 1 in Figure 3.6 and Figure 3.7 had 128 channels, with 76 of them having a 1-second time step while 52 have a 1-minute time interval. The configuration labeled as 2 is the same configuration presented in detail in Section 3.2.2, and it has 194 1-minute time interval channels. Therefore, it is expected that it needed much more memory, as configuration 2 loads and processes more data.

Figure 3.7 also highlights that some of the most time-consuming tabs did not require much RAM but took a long time to process the data. This implies that the refactorization was not necessary to decrease the RAM usage. However, we chose to refactor these configurations to ensure there is no bottleneck in the pipeline, with one process taking an extended amount of time to complete.

Finally, we observe that the configurations labeled 5 and 8 did not meet the 64 GB constraint. These are special cases where it is not possible to further refactor the configuration file to decrease memory usage. The issue arises because these configurations contain several channels in a single plot, requiring all these channels to be loaded in the same process and consuming a significant amount of memory. To reduce the memory consumption of this configuration, the GWSumm package would need to be refactored to allow the individual data processing of channels in the same plot. Implementing such features in GWSumm is beyond the scope of this thesis.

While one process still consumes a substantial amount of memory, it doesn't prevent the execution of this process in the vanilla universe. However, it limits the number of nodes on which this process can run, as there are fewer nodes with more than 100 GB of available RAM.

# 4 Conclusions

The LIGO Summary Pages play a crucial role in the daily work of scientists seeking to comprehend and mitigate noise sources in gravitational wave observatories. These pages provide essential plots from thousands of sensors within the detectors, and the speed of information updates is crucial. In the current pipeline, LIGO Summary Pages processes run sequentially, taking several hours to update the entire set of pages. In this thesis, we optimized the most time-consuming processes using the High Throughput Computing framework.

The idea behind High Throughput Computing involves optimizing the number of outputs by restructuring processes into small, independent tasks that can run concurrently. The LIGO Summary Pages pipeline utilizes the `GWSumm` package [4], a tool designed for loading and processing gravitational wave data to generate HTML pages. In the pipeline, configuration files define `GWSumm` processes and some of these files include multiple pages with numerous plots, leading to slow and memory-intensive tasks.

We identified the most time-consuming processes by analyzing the log files of the Condor DAG that runs the pipeline. The first noticeable information we obtained is that most processes in the pipeline are already small unit tasks, and eight of them represent 50% of the runtime.

Following the identification of the most time-consuming processes, we refactored each of these processes one by one. Careful attention was given to ensuring that the new configuration files remained independent, thus preventing the loading and processing of the same data in two different jobs. The primary objectives of the refactorization were to achieve runtimes smaller than half an hour for each job and to minimize RAM usage. However, maintaining a logical structure for the files was considered extremely important for maintainability reasons. Each configuration file needed to contain plots of sensors that serve for similar analysis, and not randomly distributed into configuration files to ensure a balanced division of resources between jobs.

The eight most time-consuming configuration files were split into thirty-one new files. The median runtime of the new configuration files is $10 \pm 5$ minutes, a substantial improvement compared to the original jobs. Most of the original jobs took more than 30 minutes to run, and the slowest job took almost one hour. In contrast, the slowest job among the new files now runs in approximately 25 minutes. This optimization allows all new files to run in parallel, leading to a significant reduction in overall processing time.

The memory consumption of the jobs was considerably reduced. Some of the old jobs had peak RAM usage exceeding 150 GB. The median peak RAM usage of the new jobs is $44 \pm 31$ GB, enabling most jobs to benefit from the smaller compute nodes available in the LIGO clusters. The RAM optimization of two configuration files was not fully achieved due to the nature of the pages they generated. These pages contain information from several sensors in a single plot, which forces the process to load a considerable amount of data. However, this does not prevent the parallel execution of these jobs, as there are compute nodes with as much as 256 GB of RAM available.

To advance this work, the optimizations implemented in this thesis need to be deployed to the production environment. Since the LIGO clusters support various tools crucial for the gravitational wave detectors, determining how to execute the pipeline in the vanilla universe should involve coordination with cluster administrators to establish prioritization for the tools. Therefore, the next phase of the project involves restructuring a DAG for the pipeline that orchestrates the LIGO Summary Pages pipeline in alignment with the requirements of the LIGO collaboration.

Further tasks related to this thesis involve the improvement of the `GWSumm` package. We showed that some jobs cannot be further split to consume less memory due to their dependence on several different sensors. Furthermore, we observed that the processes's memory consumption can consistently grow with runtime, this is attributed to the implementation of a global variable retaining loaded and processed data in memory until job completion.

# References


[1] M. Maggiore, *Gravitational Waves. Vol. 1: Theory and Experiments*. Oxford University Press, 2007. doi: 10.1093/acprof:oso/9780198570745.001.0001

[2] R. Abbott *et al.*, "GWTC-3: Compact Binary Coalescences Observed by LIGO and Virgo During the Second Part of the Third Observing Run," Nov. 2021, Available: https://arxiv.org/abs/2111.03606

[3] R. Abbott *et al.*, "Observation of gravitational waves from a binary black hole merger," *Phys. Rev. Lett.*, vol. 116, p. 061102, Feb. 2016, doi: 10.1103/PhysRevLett.116.061102. Available: https://link.aps.org/doi/10.1103/PhysRevLett.116.061102

[4] "Gravitational-wave Summary Information Generator." https://github.com/gwpy/gwsumm.

[5] "Gravitational-Wave Observatory Status." https://gwosc.org/detector_status/.

[6] A. Beck, "High Throughput Computing: An Interview with Miron Livny." https://www.hpcwire.com/1997/06/27/high-throughput-computing-an-interview-with-miron-livny/, 1997.

[7] "Open Science Grid - OSG." https://osg-htc.org/about/introduction/.

[8] "IGWN Computing Grid - ICG." https://computing.docs.ligo.org/guide/dhtc/.

[9] "HTCondor." https://htcondor.org/.

[10] "HTCondor Manual." https://htcondor.readthedocs.io/.

[11] "Memory Profiler." https://github.com/pythonprofilers/memory_profiler.