Automatic Energy-Efficient Job Scheduling in HPC: A Novel Slurm Plugin Approach

Anders Aaen Springborg
aaspringborg@gmail.com
Aalborg University
Aalborg Øst, Denmark

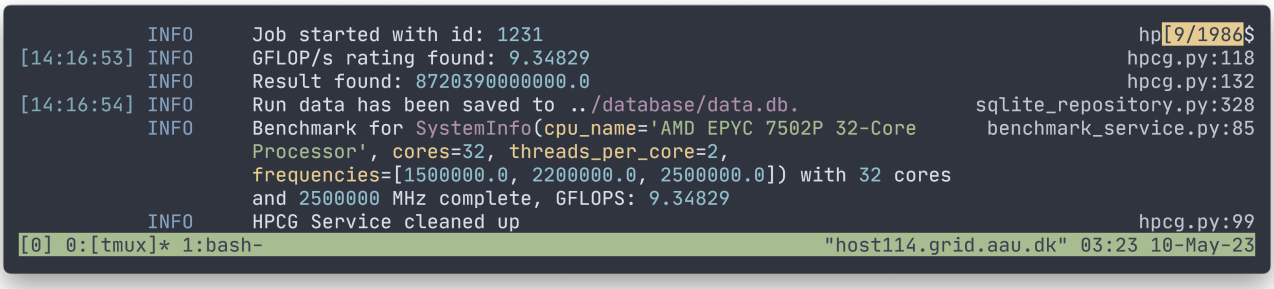


Figure 1: Log from Chronus, making an energy benchmark

ABSTRACT

This paper presents a novel approach to energy-efficient job scheduling in High Performance Computing (HPC) environments. The primary objective is to reduce the technology gap between research and production in energy-efficient scheduling models for HPC. I propose an architecture and a program that decouples scheduling heuristics to a Python application in the HPC scheduler Slurm, traditionally written in C. This approach leverages the principles of Service-Oriented Architecture (SOA) and Clean Architecture to create a system that is adaptable for production setups. It provides a platform for integrating various energy-efficient scheduling models.

My implementation demonstrates a potential energy saving of 11% in the High Performance Conjugate Gradients (HPCG) benchmark, which represents modern applications' data access patterns and computation. This showcases the approach in a single-node HPC cluster.

The paper underscores the importance of this work as a foundation for integrating research into production. It provides a realistic example of how energy-efficient HPC can be achieved in a production setting. Furthermore, it opens up possibilities for more complex applications, such as automatically scheduling jobs when energy is cheap and renewable, a practice already in use in companies utilizing HPC.

This work contributes to the field by demonstrating a practical, energy-efficient solution for job scheduling in HPC, and by highlighting the potential for future enhancements in this area.

1 INTRODUCTION

High-Performance Computing (HPC) is a rapidly expanding field, driven by increasing business demands for computational power. However, this surge has led to a rise in energy consumption, with projections suggesting it could equal the world's total energy production by 2050, looking at the growth rate from 2010 to 2020 [4]. This highlights the need for effective HPC management to optimize performance while minimizing power draw. In the context of machine learning (ML) systems, the cost of computation has been growing significantly, indicating the increasing expense of these training runs and the willingness of actors to invest in them. Therefore, energy-efficient strategies in HPC are crucial for managing these costs and facilitating AI progress.[5]

The need for efficient HPC software scheduling has become especially apparent since the European Energy Crisis of 2022 when Russia cut off its gas supply to Europe due to political tensions. This crisis made the prices for energy increase significantly [11]. Vestas, a Danish wind turbine company, are facing the same problems when running HPC applications. Vestas are watching their CO_2 footprint, and trying to maintain costs, by only scheduling HPC applications when there is cheap or green energy in the market.[2]

HPC software scheduling for improved energy efficiency is essential for reducing environmental impact while ensuring optimal performance from high-performance computers. Recognizing the importance of managing resources efficiently, businesses can make substantial progress towards sustainable computing practices that will benefit both society and their bottom line.

This paper proposes a novel approach for the integration of energy-efficient models within High-Performance Computing (HPC) systems, architected with a keen focus on ensuring interoperability among the components of an HPC system. A review of existing solutions in Section 2 highlights the prevalent technology gap between academic research and practical implementation in the field of HPC systems. Section 3 delineates the construction and operational mechanisms of the proposed system, providing an in-depth understanding of its functionality and application in real-world

scenarios. Section 4 delves into the technical intricacies that make the proposed solution feasible, while Section 5 presents an empirical study demonstrating the system's efficacy in enhancing energy usage efficiency. Finally, Section 6 outlines potential avenues for future research, drawing conclusions about the effectiveness of the proposed solution and serving as a roadmap for future endeavors aimed at enhancing the energy efficiency of HPC systems. In Appendix B, C and D there are section about my Progress. Appendix B describes where to find the code. Appendix C is about my work process. Appendix D is about how I verified that everything worked.

2 RELATED WORKS

This section provides a brief overview of the existing solutions and highlights the technology gap that the proposed solution aims to bridge.

2.1 Slurm and HPC

Simple Linux Utility for Resource Management, Slurm. Slurm is a highly scalable, open-source job scheduler designed for Linux clusters in HPC environments[25]. It manages computational workloads by allocating resources, providing a framework for starting, executing, and monitoring work, and managing a queue of pending work. Developed at Lawrence Livermore National Laboratory, Slurm is now widely used in many of the world's supercomputers, research universities, and commercial companies involved in HPC[3]. It supports a variety of job types, including single or multiple-task jobs, job arrays, and job steps where multiple tasks of a job can be distributed across multiple nodes. Slurm is a widely-used scheduling system employed by 60% of the top 10 supercomputers[24] in the world.

Niagara: Canada's National Supercomputing System. [18] provides a comprehensive overview of the Niagara supercomputer, which uses Slurm as its job scheduler and resource manager. The authors discuss the system's software stack, job scheduler, file systems, and monitoring tools. They highlight the flexibility of Slurm in its configuration, emphasizing that fewer constraints on a scheduler can lead to better performance in delivering high and fair utilization of the cluster. They also mention the use of the Slurm multifactor priority plugin to balance various factors used in priority computation, such as job age and size, the partition it was submitted to, the job's quality of service, and the user's fair share of the system. Niagara is an example of a complex HPC system, where everything is tailored to their cluster. As they describe it, that is the best practice way. That means that every HPC cluster, that is following best practices will have different setups. It is also an example of how complex it can be to set up an HPC system.

Energy efficiency approaches.

2.1.1 Indutry approaches to energy efficiency.

The Language Mojo. [10] is a new programming language under development. It was announced in May 2023. It is focused on AI, where they have features such as `autotune`. Autotune automatically calculated the fastest parameters of your code, caches it, and then uses that for compiling your code. An example of a parameter is the tile size of a matrix, to fit your hardware cache. This is similar to the approach proposed in this paper, where we are trying to find the most energy-efficient parameters. The difference is that they are doing it at compile time, and this paper's approach is configuring runtime parameters.

Lancium. is a company that has developed its unique strategy to achieve energy efficiency in HPC workloads. Lancium's approach leverages the variability of renewable energy generation to optimize the scheduling of HPC jobs. By aligning the execution of computationally intensive tasks with periods of high renewable energy availability. Based on historical data Lancium's solution significantly reduces the carbon footprint and energy costs associated with HPC operations. They are looking at the energy market at a macro scale to optimize for green energy, whereas this paper's approach is optimizing the program to use less energy.

2.1.2 Energy efficiency models for Slurm.

Energy-Optimal Configurations for Single-Node HPC Applications. [21] presents a method for finding energy-optimal configurations for single-node HPC applications. The method uses a genetic algorithm to find the optimal configuration for a given application. The results show that the proposed method can find energy-optimal configurations for a wide range of applications. They outperformed the default Linux DVFS scheme in its best case with an average of 6% energy savings. In DVFS's worst case, the savings were about 790%, on average. The experiment for this paper was done by setting frequencies getting a admin to set the frequencies of cores, and then running the program on those cores. This does not work in a production setting, as you do not know which cores your job is running on before it is out of the queue and scheduled.

Dynamic Power Management for Value-Oriented Schedulers in Power-Constrained HPC System. [12] presents a framework for dynamically managing the power consumption of HPC systems. This framework allows users to customize the power budget and adjust it according to their needs. The results show that the proposed framework can reduce power consumption by up to 30%. This is useful for making the plugin dynamically change the order of jobs, to be more efficient. They are evaluating it by simulating it, and not running it on a HPC system.

Several existing solutions have been proposed to address the issue of energy efficiency in HPC systems. However, these solutions are either theoretical or standalone, and there is a lack of a comprehensive architecture that can be integrated with existing HPC systems to enhance their energy efficiency. This paper proposes an approach that is designed to be used in production, to facilitate great solutions like the ones mentioned above, with the potential to be integrated into a production environment.

3 THE ECO PLUGIN

The primary focus of this contribution is the development of a novel Slurm plugin. It focuses on Slurm, as it is used in 6 out of the 10 best supercomputers in the world, as described in Section 2.1. It is designed to enhance the energy efficiency of applications scheduled in high-performance computing systems. in a production environment. Production environments often have multiple users,

and they have a scheduler that determines which users turn it is, to use the cluster.

The eco plugin is made of two parts. One: `job_submit_eco` which is a C plugin inside of Slurm, which is responsible for changing parameters for a job. Two: Chronus, which is a Python application responsible for determining the best configuration settings for CPU frequencies, number of scheduled cores, and threads per core, intending to minimize energy consumption given the same executable and problem size. By allowing the integration of various algorithms or models, this work provides a flexible solution that can accommodate input from domain experts without requiring them to be proficient in software engineering.

The primary goal of the eco plugin is to enhance energy efficiency in high-performance computing by achieving more GFLOPS (billion floating-point operations per second) per watt. This is accomplished by incorporating energy-efficient configurations into job scheduling within Slurm.

The novelty of this contribution lies in its potential integration of energy-efficient scheduling and getting performance data in a production environment. While other solutions have focused on saving energy for individual applications, this approach brings energy efficiency to the forefront in a widely-adopted scheduling system. By implementing energy-efficient configurations, the eco plugin not only saves energy but also promotes more sustainable high-performance computing with only a minor increase in execution time. This can be compared to driving a car at a moderate speed to achieve better miles per gallon, as opposed to driving at higher speeds with reduced fuel efficiency.

3.1 Functionality of the Eco Plugin

The Eco Plugin works by interacting with Slurm, modifying its scheduling behavior to prioritize energy efficiency instead of time for the application to finish. By integrating this plugin into the Slurm environment, users can adopt greener scheduling practices into their existing environment.

The Eco Plugin is developed as a standalone component that can be integrated into an existing Slurm installation. The plugin consists of two parts. Part 1 is a plugin inside the Slurm code base, called "job_submit_eco". Part 2 is a Python application Chronus, that `job_submit_eco` uses to get the energy-efficient configuration. This makes it possible to add new integrations and energy efficiency models to Chronus, without having to redeploy Slurm. This resembles a Service-oriented architecture, in that regard. Figure 2 shows the architecture of Slurm, with the eco plugin highlighted in green. Slurm is a distributed system, where the `slurmctld` is running on a head node, and the compute nodes are workers running `slurmd`. The box in the left lower corner split into 4 are Slurm commands, that can be run by the user, to interact with `slurmctld`. This paper only touches on `srun` is used to submit an interactive job and directly run it on the allocated resources, and `sbatch` which is used to submit a batch job script, allowing for non-interactive, scheduled execution of tasks. The green arrow between `slurmctld` and Chronus is "job_submit_eco". For Chronus to work it needs a database, where it can store benchmarks and models. This is shown as a cylinder disk, with the text "Chronus database". Chronus is built to work with any database. Right now it has a SQLite and a CSV implementation. Chronus blob storage is built the same way and can be of any type, like the Network File System (NFS), which allows a computer to access files over a network like how local storage is accessed [15], or the Server Message Block (SMB) protocol, often known as Samba, which provides a method for client applications in a computer to read and write to files and to request services from server programs in a computer network [20]. Alternatively, a cloud solution like an S3 bucket on AWS can be used, which provides scalable object storage through web services interfaces [8]. For now, it is just a local disk, as it is the easiest to work with.

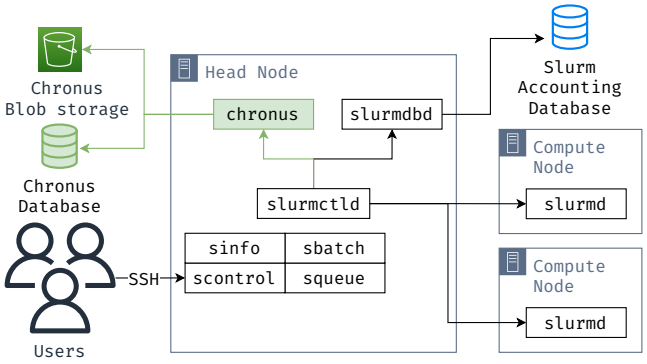


Figure 2: Slurm architecture w/ Chronus

All supercomputers are built differently and have different requirements. They have different hardware, different power sources, and different interconnects. Chronus determines the energy-efficient configuration for a given application by benchmarking various configurations, different core counts, frequencies, and threads per core. It then builds a prediction model based on these benchmarks to identify the most energy-efficient configurations for a specific HPC cluster, on a specific application. The `job_submit_eco` plugin in Slurm's architecture, modifies the job parameters upon submission, setting the optimal core count, frequency, and thread per core configuration, including whether to use hyper-threading if available.

3.1.1 Slurm: Job submit eco. Slurm is designed with a plugin architecture. There are 38 different types of plugins, with multiple implementations of each type as of version 22.05.9 [19]. The eco plugin is of the type `"job submit"`, hence the name `job_submit_eco`. This type of plugin is called when a job is submitted to the scheduler. The plugin can then modify the job before it is added to the queue. Chronus calculates the most energy-efficient configuration. The Slurm plugin is then modifying the job, to use the most energy-efficient configuration.

3.1.2 External program: Chronus. Chronus has 4 functions:

(1) Benchmarking
(2) Model building
(3) Pre-load model
(4) Predict energy efficient configuration

Figure 3 shows an overview of how everything is wired together and is a more detailed version of Figure 2. Each color or arrow is linked to a task and has a corresponding description to the arrow.

Below is a more detailed description of each step, which also references back to Figure 3.

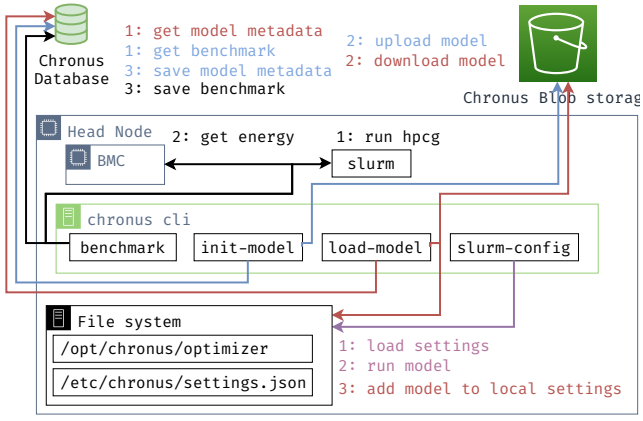


Figure 3: Chronus flow

Benchmarking - black arrows.

Step 1 Chronus starts a job on the HPC cluster, with a given configuration. This is done by submitting a job to Slurm, using the `sbatch` command.

Step 2 Chronus keeps sampling the energy usage from the Baseboard Management Controller (BMC), using Intelligent Platform Management Interface(IPMI)[9]. This is done at a 2-second interval.

Step 3 When the job is done, Chronus saves the energy usage and the results of the job to a benchmark in a database.

Chronus benchmarks the application, with different configurations: It benchmarks the application with different core counts, frequencies, and threads per core. It then saves energy usage, and execution time for each configuration. This is the baseline for making a prediction model. It keeps doing this until it has benchmarked all configurations. If no configurations are given, it will benchmark all configurations based on the system CPU.

Model building - blue arrow.

Step 1 Loads benchmarks from one system and application. Builds a prediction model, based on the benchmarks

Step 2 Uploads the model to blob storage.

Step 3 Saves metadata for the model to the database. Metadata is path in blob storage, time on creation, etc.

Chronus builds a prediction model, based on the benchmarks. It uses the energy usage over time, execution time, and the configuration of the system: the number of cores, threads per core, and frequency to build a model. The model is then saved to blob storage.

Pre-load model - red arrow.

Step 1 Loads metadata for the model from the database.

Step 2 Downloads the model from blob storage.

Step 3 Saves the model to a local disk on the head node.

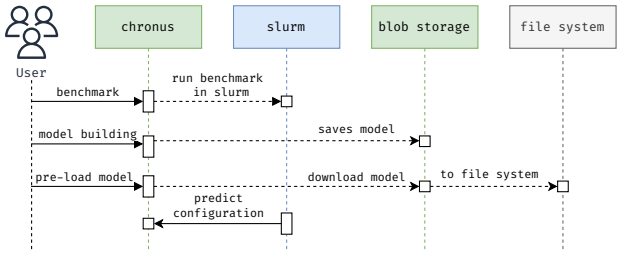


Figure 4: The Eco plugin sequence diagram

Chronus loads the model's metadata from the database. It uses the path from metadata to download and saves it to a local disk on the head node, which has `slurmctld` running. This is done to speed up the prediction process, as Slurm has a very short time to make a decision when a job is submitted. This is a constraint set by Slurm and raises an error if a plugin takes too long to run.

Predict energy efficient configuration - purple arrow.

Step 1 Loads the model from the local disk.

Step 2 Predicts the energy-efficient configuration.

Step 3 Returns the energy-efficient configuration to the eco plugin.

This step is only called by `job_submit_eco` in Slurm. The Chronus command returns a JSON, that contains the energy-efficient configuration. `job_submit_eco` calls it with the system identifier and the hash of the binary. The hash is used to determine the configuration of the binary.

Figure 4 shows a sequence of how you use the system. The solid lines are commands the user and Slurm run. The dotted lines are what happens in software, when the command is run.

3.2 Chronus Integrations

Chronus offers a series of integration interfaces for integrating with various HPC setups. The integration interfaces of Chronus are designed to change to fit different setups of HPC Clusters. In an HPC cluster comprised of a single node, it is feasible to utilize IPMI to ascertain the system's power consumption. Conversely, in a multi-node setup, you need power from multiple nodes where you have an API that measures power draw. Then there is a need for an integration in Chronus, that can read the power draw from that API. That is two different implementations for the same integration interface.

In an HPC cluster comprised of a single node, it is feasible to utilize IPMI to ascertain the system's power consumption. Conversely, in a multi-node configuration, obtaining power data necessitates an application programming interface (API) measuring power consumption across multiple nodes. Despite the differing execution methods, both scenarios aim to achieve the same goal - to provide system power measurement. Chronus needs to facilitate both, through a common integration interface. Figure 5 presents an overview of all the integrations currently in Chronus. In this section, each integration will be described.

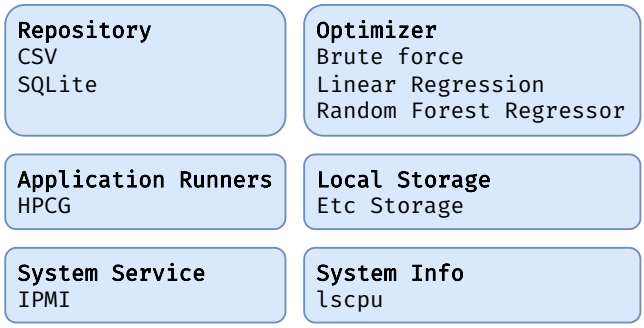


Figure 5: Integration interfaces in Chronus

Repository. Designed as a bridge for remote storage, it assists in saving metadata related to runs, benchmarks, system information, and models. This integration interface is managing data in the Chronus system.

The current implementations of the integration interface Repository are:

- CSV File
- SQLite Database

Optimizer. Designed to fit different efficiency model that calculates the optimal configuration for energy usage. This is where new methods for predicting energy efficiency will be implemented. Current implementations of the integration interface Optimizer are:

- Linear Regression
- Random Forest Regressor
- Brute Force

Application Runner. is designed to run applications for benchmarking the HPC system. The best energy efficiency configuration changes for each application. This is made for Chronus to be able to integrate with all applications. Current implementations of the integration interface Application Runner are:

- High Performance Conjugate Gradients (HPCG)

The High Performance Conjugate Gradient[6] (HPCG) benchmark is a tool designed for ranking computer systems based on their performance, while accurately reflecting the performance of modern applications. Unlike the High-Performance Linpack (HPL)[7] benchmark, which is often used for ranking computer systems, HPCG is designed to better represent how today's applications perform. It is based on a simple additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient solver, and it allows for certain code transformations. This makes HPCG a more realistic measure of system performance in practical use cases.

Local Storage. This interface is tasked with managing local settings storage. It allows for the saving and retrieving of settings and conversion of relative paths into full file paths.

The current implementation of the integration interface Local Storage is:

- ETC Storage

System Service. This service is the monitoring service. It collects all the system telemetry like CPU frequency and power draw. It is used for data sampling while running benchmarks.

The current implementation of the integration interface System Service is:

- IPMI (Intelligent Platform Management Interface)

System Info. This service gathers system information such as the number of cores, threads, frequencies, and RAM. This is what identifies the system.

The current implementation of the integration interface System Info:

- lscpu

File Repository. An integration interface used for storing optimizers in Chronus, providing a consistent API for managing optimizers.

The current implementation of the integration interface File Repository is:

- Local Storage

This implementation saves models to a folder called `./optimizers` in the current directory of the user, using Chronus. This could also be NFS, AWS blob storage, or similar.

Each integration interface provided by Chronus has been designed to cater to different aspects of the system's operation. Their flexibility and adaptability make them suitable for various implementations, thereby enhancing the overall functionality of the eco plugin. By utilizing different integration interfaces and their respective implementations, Chronus offers a robust and flexible platform for optimizing energy usage in HPC systems. Section 4.1 focuses on implementation, and it shows how to add an implementation practically.

3.3 Using the system

After installing Chronus and Slurm, and enabling `job_submit_eco` in slurm, the user can interact with the plugin through Chronus's Command Line Interface (CLI)

When `job_submit_eco` plugin is enabled for an HPC cluster, by default it will not change any settings when submitting a job. To enable it in a job, the user needs to add this line to their `sbatch` script. `#SBATCH –comment "chronus"`. This will enable the plugin for this job, and the plugin will then modify the job, to use the energy-efficient configuration. The system admin can also disable it for all jobs. This is done by using Chronus's CLI.

Chronus offers CLI commands for each of the tasks above. Each of them corresponds to one of the actions described in Section 3.1.2. These are the 5 commands.

- `benchmark`: Runs benchmarks on different configurations.
- `init-model`: Initializes the prediction model.
- `load-model`: Loads a pre-trained model.
- `slurm-config`: Executes the main functionality.
- `set`: Changes the configuration of the plugin.

The following is a description of the usage of the five commands listed above.

```
chronus benchmark [HPCG_PATH]
    --configurations [CONFIG_FILE]
```

To run a benchmark, call the `benchmark` command. In the current version, there is only an application runner for HPCG, and it requires a path to the binary, to begin the benchmark. As default, it runs in all configurations, but you can specify which configurations to benchmark, with the `-configuration` flag, with a path to a JSON with an array of configurations. A JSON configuration looks like this:

```
[
  {
    "cores": 32,
    "threads_per_core": 2,
    "frequency": 2200000
  },
  ...
]
```

The benchmarking process can take a while. One of the implementations of application runners is HPCG, which takes 30 minutes to run once. When this is run, it is an advantage to run it as a background process, or in a multiplexer like `tmux` or `screen`. When running it write logs to the terminal, and to `/var/log/chronus.log`. The benchmarking process is shown in Figure 6.

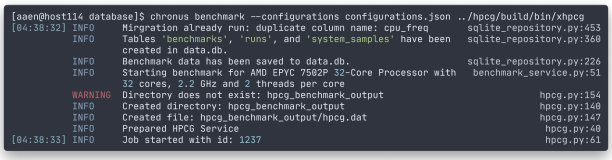


Figure 6: Chronus benchmark

```
chronus init-model
    --model [MODEL_TYPE]
    --system [SYSTEM_ID]
```

To build a model call the `init-model` command. This command takes a model type and a system identifier. The model type is the type of model Chronus should build. Currently, there are three models available: brute force, linear regression, and random tree. The options for choosing one can be displayed by running the command with the `-help` flag. See Figure 7 for a output of `-help`.

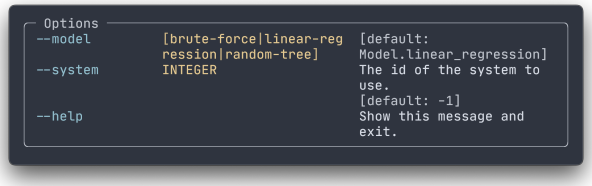


Figure 7: Set help message

The system identifier is a unique identifier for the system. This is used to identify the model when loading it. When you do not specify a system identifier, it will present you with systems that are already in the database. Figure 8 shows the CLI presenting available systems, and then how it looks building a model.

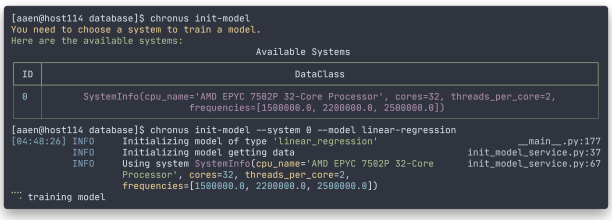


Figure 8: CLI init model

```
chronus load-model --model [MODEL_ID]
```

To pre-load a model you call the `load-model` command. The model id is the id of the model to load. When you do not specify a model id, it will present you with models that are already in the database. Figure 9 shows the CLI presenting available models, and then how it looks loading a model.

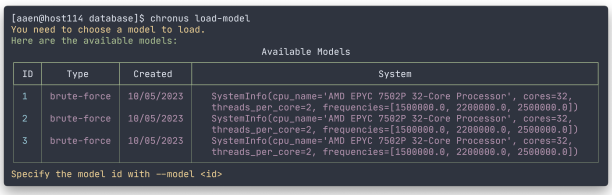


Figure 9: Displaying available models

```
chronus slurm-config [SYSTEM_IDENTIFYER] [BINARY_HASH]
```

Given a cluster identifier and a job, Chronus predicts the most efficient energy configuration. It returns the energy-efficient configuration to the Eco plugin, which modifies the job accordingly. This is not supposed to be called by the user but is called by `job_submit_eco`. If you call it, it returns a configuration in JSON format.

```
chronus set COMMAND [ARGS]
```

To change the configuration of the plugin you call the `set` command. This command takes a configuration type and a value. The configuration type is the type of configuration to change. Currently, there are three configurations available: model, system, and benchmark. The options for choosing one can be displayed by running the command with the `-help` flag.

These are the available things to set

```
Options
--help          Show this message and exit.

Commands
blob-storage  The path to the blob storage.
database      The path to the database.
state         activates, sets it to user or deactivates
              the plugin.
```

Figure 10: Set help message

3.4 Installation

Installing the eco plugin is a two-step process. First, you need to enable the plugin in Slurm, and then you need to install Chronus on the head node. Both of them have a few requirements, for them to work.

3.4.1 Slurm plugin requirements. Slurm in itself is a complex distributed system. It is built with autotools and has a lot of dependencies. The `job_submit_eco` plugin does not have any extra dependencies on Slurm. If you install a version of Slurm with this plugin in the source code, it will be compiled with Slurm.

To enable the plugin, you need to add this line to your Slurm config file.

```
JobSubmitPlugins=eco
```

The default place for this file is `/etc/slurm/slurm.conf`. This will enable the plugin, and it will be called when a job is submitted.

3.4.2 Chronus requirements. Chronus is written in Python and requires Python 3.9 or newer. Chronus is tested and works on Ubuntu 20.04, Centos 7.9, Rocky Linux 8.7, Rocky Linux 9.1, and PopOS 22.04, and should work on any Linux distribution.

Chronus can be installed as a Python package. In a virtual environment, you can install it with `pip install chronus`. If you are using Conda you can install it with `conda install chronus`. This will install Chronus in the user's path. Chronus needs to be available for the Slurm user and the user that is running the plugin. Slurm is often setup to run as a user called `Slurm` and needs to be able to run Chronus on job submit to get a configuration. The users using Chronus as a CLI needs to be able to modify system settings like `/etc/chronus/settings.json`. You could install it system-wide, and use `sudo` when changing system settings.

Chronus reads most of the system information in Linux files. An example is the CPU scaling frequencies, which are read from:

```
/sys/devices/system/cpu/cpu0/cpufreq/
    scaling_available_frequencies
```

When using IPMI to read the power consumption of the nodes, Chronus needs access to IPMI. On most systems, only admins can read the `special files` in `/dev`. Chronus needs to be able to read the onboard communication to the BMC through `/dev/ipmi0`, or provided with credentials to the BMC. Making Chronus able to read onboard communication, can be done by making `/dev/ipmi0` readable by users on the system. This can be done by running

```
chmod o+r /dev/ipmi0
```

This will allow Chronus to read the power consumption of the node. Credentials to the BMC can be provided in the settings file.

4 IMPLEMENTATION

This section discusses the implementation details of the eco plugin, and how it integrates into the proposed architecture. First `job_submit_eco` and then Chronus. We are going to go through how to extend Chronus and then how they interact. Section 3.1 already went through the high-level architecture, and this will not be explained further in this section.

4.1 Extending Chronus

To support the extensibility described in Section 3.1.2, Chronus is made with Clean Architecture and Dependency Inversion.

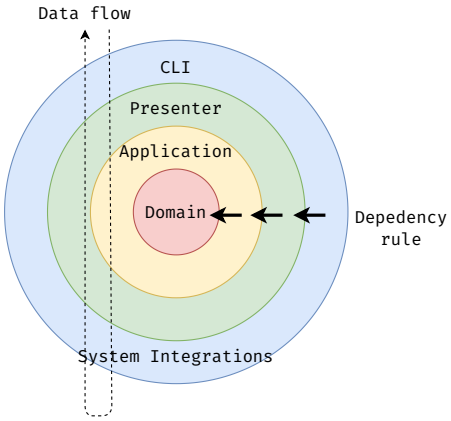


Figure 11: Clean Architecture for Chronus

Clean architecture[14], as proposed by Robert C. Martin, is a software design philosophy that emphasizes the separation of concerns. It encourages the development of systems that are independent of specific details, such as databases and user interfaces. For Chronus, it is independent of application runner, system sampling, etc. from Section 3.1.2. This independence allows for easier maintenance and testing, as changes in one part of the system do not affect others. Furthermore, it enables the system to be extended without significant restructuring, which is particularly important in the context of HPC systems due to their diverse nature.

Chronus design is shown in Figure 11. It is a clean architecture, where each ring in the figure represents a layer in the system. The arrow on the right shows the path of the dependencies, where each layer only has dependencies towards the center. The green ring `Presenter` is the contextual boundary between the core of the system, and system integrations mentioned in 3.1.2. Each system integration implements an integration interface, which the application layer needs, to do the business logic. The Presenter layer acts as a mapping for the data structure that is most convenient for the UI,

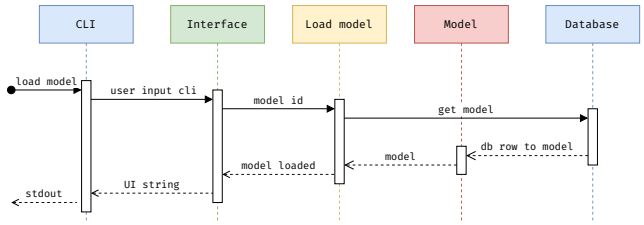


Figure 12: Sequence diagram of data flow in loading model

and the data structure that is most convenient for the application. Data still flows through the system from `CLI` to Database, even with dependencies only pointing towards the center. Figure 12 is a simplified example of loading a model from Chronus's `CLI`.

The way Chronus archives this dependency structure is by using the `Dependency Inversion Principle`[13]. This principle, which is one of the five principles of Solid Design, states that high-level modules should not depend on low-level modules; both should depend on abstractions. By adhering to this principle, we ensure that our system's components are loosely coupled, making them easier to modify and extend. This is crucial for supporting the wide range of HPC systems, each with its unique characteristics and requirements. The application needs to describe how it uses the system integrations. We do this by using interfaces. As an example, see Listing 1 for how an interface is injected into the constructor, and later used.

```python
# application/load_model.py
class LoadModelService:
  def __init__(self, repository: RepositoryInterface):
      self.repository = repository

  def run(self, model_id: int):
    model = self.repository.get_model(self.model_id)
    # load model to local storage...
    return model.name, model.path

# system_integrations/repository_interface.py
class RepositoryInterface:
  def get_model(self, model_id: int) -> Model:
      raise NotImplementedError()
```

Listing 1: Load model command in Application layer

It takes an object that implements the `RepositoryInterface` interface. The interface is defined in the `Application` module and is shown in Listing 1. Python is a dynamic typed language, and do not have interfaces. The interface is an abstract class instead, that defines the methods that the application layer needs. The application layer does not care how the repository is implemented, as long as it implements the interface.

This allows the application layer to be independent of the real implementation. In the outer layer of Figure 11, the real implementation is defined. The real implementation is the system integration and is defined in the `System Integrations` layer.

In the entry point of your application, you specify which implementations there are for each interface. `init model` is an example of where these dependencies are interchangeable. When `init model` is called, you specify which model you want to make. This is an example of how dependencies can be swapped in and out, while being used for the same purpose. A part of the domain is to know what kinda of model it is. Therefore the class `Model` has a string called type. This string is defined by the model interface and is used to load the correct model. In the entry point of Chronus, there is a mapping between the model type and the model implementation. This is shown in Listing 2. This is an example of how the application layer is independent of the real implementation.

```python
# main.py

class ModelFactory:
  def get_optimizer(model_type: str) -> OptimizerInterface:
    if model_type == LinearRegressionOptimizer.name():
        return LinearRegressionOptimizer()
    elif model_type == BruteForceOptimizer.name():
        return BruteForceOptimizer()
    elif model_type == RandomTreeOptimizer.name():
        return RandomTreeOptimizer()
    else:
      raise Exception("Unknown optimizer type")

# presenter/load_model
def load_model(model_id: str):
  # ...
  model_type = model.type
  model = ModelFactory.get_model(model_type)
```

Listing 2: Injection model into application layer

Python was chosen as the implementation language as both of the energy efficiency models in Related Works are implemented in Python and it has the most popular libraries for data analysis. Stackoverflow's 2022 developer survey[17], shows that out of the top 11 most popular frameworks and libraries in all fields 5 of them are data science related. All 5 are Python technologies. This decision is made to make it people who are making energy efficiency models to integrate their existing Python code, into a production system.

4.2 Integration between Chronus and Slurm

As shown in section 3 the eco plugin is a distributed system, and there is some communication between Chronus and `job_submit_eco`. This section will explain how `job_submit_eco` gets a configuration from Chronus and set the parameters in slurm.

Slurm has a plugin structure. It works by loading a shared object file and calling functions in the file. When a job is submitted to Slurm, the `job_submit_eco` plugin is called. It is called with the job id, and the job id is used to get the job information from Slurm. The job information is then used to get the system information. The system information is then used to get the configuration from Chronus.

4.2.1 Load config. Getting a config for a job submit Chronus needs a binary hash and a system hash. The binary hash is a hash of the executable that is run in the job. The system hash is a hash of the system that the job is run on. The binary hash is used to identify the application running, and the system hash is used to identify the system configuration. This paper focused on single node clusters, and the system identifier is the CPU information and ram size hashed. The system information is read from `/proc/cpuinfo` and

ram size from `/proc/meminfo`. There is some error handling on reading files, and then they get concatenated into a string. The string is then hashed with the simple hash function. The hash is then returned to the plugin. The simple hash code is showed in Listing 3.

```c
unsigned long simple_hash(const char *str) {
  unsigned long hash = 5381;
  int c;
  while ((c = *str++))
      hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
  return hash;
}

```

Listing 3: Simple hash function

4.2.2 Set configuration on compute node. When the plugin has gotten a configuration from Chronus, it needs to set the configuration on the compute node. The configuration is set by setting using Slurm's API in the plugin. They provide a struct called `job_description`, which contains all the information about the job. The `job_description` can be modified, to change the behavior of a job. The configuration is set by modifying these variables in the struct:

- `job_description->threads_per_cpu` for enabling hyper-threading
- `job_description->num_tasks` for the number of cores
- `job_description->min_frequency` for the CPU frequency
- `job_description->max_frequency` for the CPU frequency

Listing 4 shows how the configuration is set.

```c
extern int job_submit(job_desc_msg_t *job_desc, uint32_t
↪ submit_uid, char **err_msg)
{
    //... init code
    load_config(&config);
    job_desc->num_tasks = config.num_tasks;
    //... rest of the code
}
```

Listing 4: Snippet of setting job configuration

4.2.3 Running benchmarks. Chronus does the higher-level logic in the application layer. See Figure 11 for reference of the application layer. Each implementation of the application runner interface is responsible for running the implementation in slurm. In the HPCG implementation, it is done by using system calls. In Listing 5 you see a code snippet for how it is calling `sbatch` to schedule a job.

The slurm file that is generated on line 3 in Listing 5 makes a Slurm file, that uses Slurm's CLI input to set its cores, threads, and frequency for the run. The function generating the file is shown in Listing 6. On line 4 the number of cores is set, on line 5 the frequency is set and on line 7 hyper-threading is specified as a parameter to `srun`

```python
def run(self, cores: int = 1, frequency: int = 1_500_000,
↪ thread_per_core=1):
  # generate file, with benchmark configuration
  slurm_file_content =
  ↪ self._generate_slurm_file_content(cores, frequency,
  ↪ thread_per_core)

  # Preparing the file HPCG_BENCHMARK.slurm...
  # SKIP CODE LINE

  job = subprocess.run(
      ["sbatch", "HPCG_BENCHMARK.slurm"],
      cwd=self._output_dir + "hpcg_benchmark_output",
      stdout=subprocess.PIPE,
  )
```

Listing 5: Running HPCG as a benchmark in Chronus

```python
  def _generate_slurm_file_content(self, cores, frequency,
  ↪ thread_per_core) -> str:
  return f"""#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks={cores}
#SBATCH --cpu-freq={frequency}

srun --mpi=pmix_v4 --ntasks-per-core={thread_per_core}
↪ {self._hpcg_path}"""

```

Listing 6: Slurm file for HPCG

5 EXPERIMENTS

This section presents the experimental setup and results obtained from running benchmarks with different configurations of cores, frequency, and hyper-threading. The experiments were conducted on a Lenovo ThinkSystem SR650 equipped with an AMD Epyc 7502P CPU and 256 GB of RAM, running Rocky Linux 8.7 with Linux kernel 4.18. IPMI sensors are used to measure power consumption. The HPCG benchmark is used to measure performance.

5.1 Power Measurement

To get accurate results timing of power consumption, I needed to automate it with code. I did not have access to a smart PSU, which has a digital wattmeter with an API to get the power consumption. An alternative was using IPMI, as the node used for testing BMC had IPMI sensors.[9] IPMI is a standardized interface for out-of-band management of computer systems. It is a firmware that is running on the BMC and is used to monitor the system. The BMC is a microcontroller that is on the motherboard and is used to monitor the system. To test the accuracy of IPMI, I compared it to a digital wattmeter. The wattmeter was connected to the machine's two power supply units (PSUs). During the HPCG benchmark run, the first PSU reported a power draw of 129.7 watts, while the second PSU reported 143.7 watts. IPMI sensors reported a total power draw of 258 watts, resulting in a percentage difference of 5.96% between the wattmeter and IPMI readings, as per the calculations in Equation

1.

$$\text{Percentage Difference} = \frac{|\text{IPMI ussage} - \text{Watt meter usage}|}{\text{IPMI ussage}} \times 100$$
$$\text{Percentage Difference} = \frac{|258 - 273.4|}{258} \times 100$$
$$\text{Percentage Difference} = 5.96\% \tag{1}$$

Therefore, the IPMI report differs from the real total PSU usage by approximately 5.96%, which is relatively small. A cropped picture of the power measurement setup is provided for reference in Figure 13. A full picture is provided in Figure 16 in Appendix A.

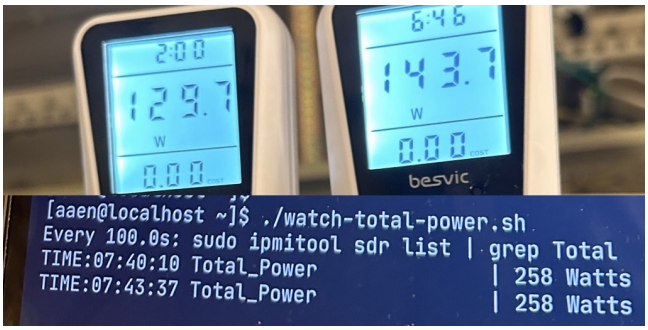


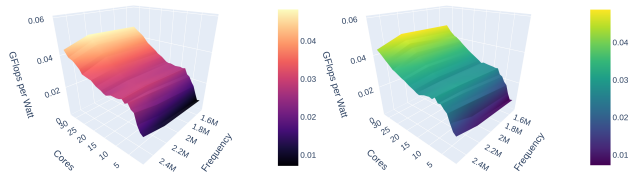
Figure 13: IPMI vs Wattmeter

5.2 Experiment Results

The HPCG benchmark was used for the experiments. Each job ran for 20 minutes, with data sampled every 3 seconds. It is run with the default problem size $x = 104, y = 104, z = 104$[6], which used 32GB ram. That is 12.5% of the system's 256GB of ram. The GFLOPS performance of a job is calculated by HPCG.

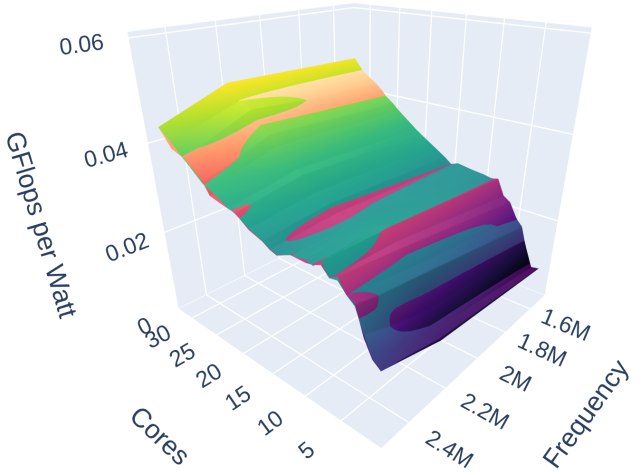
5.2.1 GFLOPS per Watt. In Figure 14 you see three surface plots to visualize GFLOPS per watt for different numbers of cores and frequencies. The first two plots in Figure 14a and 14a show a plot of GFLOPS per watt for different numbers of cores and frequencies with either or without hyper-threading. The third plot is both surface plots in the same figure. In Figure 14a and 14a, we see a surface that has a similar shape, and therefore we focus on the overlapping plot. Figure 14 shows that non-hyper-threaded workloads have better or worse $\frac{GFLOPS}{Watt}$. This is likely because the HPCG benchmark is memory-bound, a characteristic shared by many real-world applications. Table 1 reports the data points for the best 13 configurations. The Performance column is calculated by taking how many $\frac{flops}{second}$ each benchmark ran. Rows with a grey background are with hyper-threading enabled and the blue row is the standard configuration Slurm runs without the plugin. There is a full table in Appendix A Table 4,, 5 and 6 with all the data points. There are multiple things to observe in this data:

(1) The $\frac{GFLOPS}{Watt}$ is best for 32 cores and 2.2 GHz without hyper-threading, with a value of 0.0488 $\frac{GFLOPS}{Watt}$. That is 13% better than the standard configuration, with only a 2% decrease in performance



(a) GFLOPS/watt with hyper-threading

(b) GFLOPS/watt without hyper-threading



(c) Overlapping

Figure 14: $\frac{GFLOPS}{Watt}$ for each configuration

(2) Running without hyper-threading results often beats its hyper-threading counterpart in $\frac{GFLOPS}{Watt}$. Hyper-threading works by splitting the CPU into two virtual cores, and they take turns doing the computation, while the other cores are waiting for something like I/O. An article from the HPC Revolution[23] explains that cache-friendly workloads and workloads with high CPU utilization, like HPCG, might not benefit from hyper-threading as logical cores share the cache, and the core might already have work to do. HPCG might be so well optimized, that the cores memory channels are completely saturated, and therefore adding hyper-threading does not scale linearly with performance per watt.

(3) At lower cores, especially 7, hyper-threading gives a better $\frac{GFLOPS}{Watt}$. This is likely because the data does not use the cache as efficiently, and therefore hyper-threading can by having one logical core computing, while waiting for the other to load data into the cache.

5.2.2 Power Over Time. Figure 15 shows a plot of the power consumption of the system for the best configuration and the standard Slurm configuration. Table 2 presents an overview of the sum and average of the lines. The data shows that the power consumption of the system is more stable in the new configuration, compared to the standard configuration. The power consumption of the best configuration is lower than the standard configuration, as expected.

Cores	GHz	Hyper-thread	GFLOPS / watt	GFLOPS / watt %	Performance %
32	2.2	f	0.0488	1.13	0.98
32	2.2	t	0.0483	1.12	0.98
32	1.5	f	0.0480	1.11	0.90
32	1.5	t	0.0469	1.09	0.90
30	2.2	t	0.0456	1.06	0.93
30	2.2	f	0.0456	1.06	0.93
30	1.5	t	0.0446	1.03	0.86
28	2.2	f	0.0444	1.03	0.88
30	1.5	f	0.0441	1.02	0.86
28	2.2	t	0.0437	1.01	0.88
32	2.5	f	0.0432	1.00	1.00
32	2.5	t	0.0431	1.00	1.00
28	1.5	t	0.0425	0.99	0.81

Table 1: $\frac{GFLOPS}{Watt}$ comparison

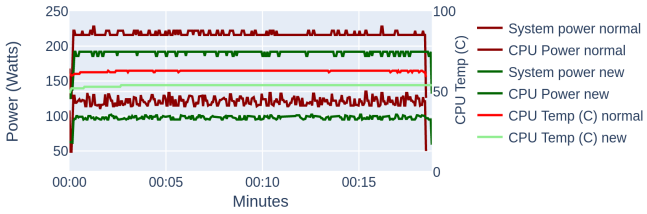


Figure 15: System samples for best and standard configuration

As assumed earlier in this section, the HPCG benchmark is memory-bound, and therefore the CPU is not the bottleneck. The new computation might fit better to the memory channel bottleneck and therefore not have to increase and decrease power during the benchmark. Increasing and decreasing power is inefficient, as the system has to ramp up and down the power consumption. It is like pressing the gas, lifting off over and over again in a car, compared to running at a constant speed. The CPU temperature is also lower in the best configuration, which means that power can be saved on cooling. The average system power draw is 190 watts for the best configuration and 216 watts for the standard configuration. The best configuration runs for 18 seconds longer and uses 26 watts per second. In total, the best configuration uses $214.4K_J$ and the standard configuration uses $240.2K_J$. This is an 11% reduction in energy usage. The total CPU power for the best configuration is $109.8K_J$ watts and $133.5K_J$ watts for the standard configuration. This is an 18% reduction in CPU power.

5.2.3 Comparison with Other Work. This research was analyzed alongside related work within the same domain. Notably, the study "Energy-Optimal Configurations for Single-Node HPC Applications" [21] which is referenced in Section 2, reported a substantial average 106% improvement in system power efficiency for 32 cores. Their result is compared to Linux's with their built-in Dynamic voltage and frequency scaling (DVFS)[22] in "`On Demand`" mode.

Name	Avg Sys (W)	Avg Cpu (W)	Sys KJ	Cpu KJ	Avg Temp (C)	Run time
Standard	216.6	120.4	240.2	133.5	62.8	0:18:29
Best	190.1	97.4	214.4	109.8	53.8	0:18:47

Table 2: Average Watt Usage, KJ used, Average CPU Temp and Runtime

This paper is compared to Slurm's standard configuration, which is DVFS in Performance mode.[19]. This value is derived by dividing the new system power by the standard power. In contrast, our paper expresses efficiency as a proportion of the initial consumption. To ensure uniformity with our findings, we recalculated the improved efficiency from the mentioned paper into a fraction of the original power consumption using Equation 2.

$$\text{Standard system power} = \text{New system power} * \text{Better efficiency}$$

$$\text{Standard system power} = \text{New system power} * \frac{106}{100}$$

$$\frac{\text{New system power}}{\text{Standard system power}} = \frac{100}{106}$$

$$\frac{\text{New system power}}{\text{Standard system power}} = 94.34\%$$

$$\begin{aligned} \text{Reduction} &= 100\% - \frac{\text{New system power}}{\text{Standard system power}} \\ \text{Reduction} &= 100\% - 94.34\% \\ \text{Reduction} &= 5.66\% \end{aligned} \tag{2}$$

The derived energy reduction is 5.66. We also see a 14% reduction in cooling, which can lead to using less power on cooling in a data center. This substantial reduction in system power consumption holds the potential for significant cost savings and positive environmental impacts. See Table 3 for a comparison.

Plugin	CPU (%)	Reduction	System Reduction (%)
Eco	18%		11.00%
Related work[21]	NaN^a		$5.66\%^b$

[a]The value for CPU Reduction is not available for the related work.
[b]With DVFS set to `On Demand`

Table 3: Comparison of System Power Reduction

6 CONCLUSIONS & FUTURE WORK

The eco plugin has made significant strides in the field of energy-efficient job scheduling in high-performance computing (HPC). The main contribution of this project is the architecture of the plugin, which increases development velocity in this area. The system design of the plugin, with its service-oriented architecture (SOA),

decouples some of the systems to Chronus, thereby increasing the flexibility and scalability of the system.

With a ThinkSystem SR650 equipped with an AMD Epyc 7502P CPU and 256 GB of RAM, running Rocky Linux 8.7 with Linux kernel 4.18, the eco plugin was able to reduce the system power consumption by 11% and the CPU power consumption by 18%. This also affected the average CPU temperature and reduced it by 14%. Compared to related works, the eco plugin is able to reduce the system power consumption by 11% compared to their 5.66%. This reduction in power consumption could lead to significant cost savings and environmental benefits.

6.1 Limitations

Despite the advancements made, the current plugin has its limitations. The most significant limitation is that it only works on single-node systems, whereas most production setups are on multi-node systems. This limitation restricts the plugin's applicability in real-world scenarios.

6.1.1 Single node. The plugin's current implementation is designed to work on single-node systems. This design choice limits the plugin's applicability in multi-node setups, which are common in production environments.

6.1.2 Find binary path. The current implementation of the plugin uses a constant string to find the binary path. This was an implementation detail that was never fixed after an initial trial and error.

6.1.3 Simple model. The model interface in the system is simple and does not take into account various parameters that could be optimized, such as time and program input. This simplicity limits the plugin's ability to optimize job scheduling in a more comprehensive manner.

6.2 Future works

Several potential future enhancements could be made to the eco plugin project. These enhancements include multi-node support, scheduling jobs at specific times, incorporating deadlines for energy-efficient job completion, and tuning GPU frequency for better energy efficiency.

6.2.1 Deadline. Future enhancements could include giving a deadline as an input in `sbatch`, and the model finds the best configuration that still finishes before the deadline (statistically). This feature would allow the plugin to work in environments where coordination is critical. An example would be if Vestas needed a simulation to be done by Monday morning.

6.2.2 GPU frequency. Another potential enhancement is to tune the clock rate and memory frequency to get better energy efficiency on GPU. Research has found that this can save 28% energy for 1 % performance loss [1]. Nvidia provides telemetry tools for this purpose, which could be integrated into the plugin.[16]

6.2.3 Multi node. The plugin could be developed to support multi-node HPC setups. This enhancement would require a mechanism for identifying a multi-node system and extending the main part of the system to handle affinity and sockets for multi-node systems.

6.2.4 Time scheduling. The model could be given access to schedule a job at a specific time. This enhancement could be used to get a better price for the energy or be tailored to only use renewable energy, based on the energy market. This approach could lead to significant cost savings and environmental benefits.

In conclusion, the eco plugin project has made significant strides in energy-efficient job scheduling in HPC. However, there are still several areas for potential future enhancements. With these enhancements, the eco plugin could become a powerful tool for energy-efficient job scheduling in HPC.

ACKNOWLEDGMENTS

A big thank you to Josva Kleist from the DEIS group at Aalborg University, for providing hardware, and helping with infrastructure problems on the hardware.
A big thank you to Samuel Xavier-de-Souza from the Laboratory of Parallel Architectures for Signal Processing at Universidade Federal do Rio Grando de Norte, who has helped with providing great domain knowledge and helping me not derailing from the focus of this paper.
A big thank you to Michele Albano from the DEIS group at Aalborg University, for being a great supervisor, and very flexible with meetings a reviewing material.

REFERENCES

[1] Yuki Abe, Hiroshi Sasaki, Martin Peres, Koji Inoue, Kazuaki Murakami, and Shinpei Kato. 2012. Power and performance analysis of GPU-accelerated systems. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems.*
[2] Cloud architect catalyst engineer lead at Microsoft Anders Lybecker. 2023. private interview.
[3] Brett M Bode, David M Halstead, Ricky Kendall, Zhou Lei, and David Jackson. 2000. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters.. In *Annual Linux Showcase & Conference.*
[4] Semiconductor Research Corporation. 2020. Decadal Plan for Semiconductors. (2020). https://www.src.org/about/decadal-plan/
[5] Ben Cottier. 2023. Trends in the Dollar Training Cost of Machine Learning Systems. *Epoch. January* 31 (2023), 2023.
[6] Jack Dongarra, Piotr Luszczek, and M Heroux. 2013. HPCG technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752* (2013).
[7] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.
[8] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. 2008. Cloud computing and grid computing 360-degree compared. In *2008 grid computing environments workshop.* Ieee, 1–10.
[9] U. Guide. [n. d.]. Lenovo ThinkSystem System Manager User Guide. ([n. d.]).
[10] Modular Inc. [n. d.]. Mojo : Programming language for all of ai. https://www.modular.com/mojo
[11] Alfred Kammer, Jihad Azour, Abebe Aemro Selassie, Ilan Goldfajn, and Changyong Rhee. 2022. How war in Ukraine is reverberating across world's regions. *Washington: IMF, March* 15 (2022), 2022.
[12] Nirmal Kumbhare, Ali Akoglu, Aniruddha Marathe, Salim Hariri, and Ghaleb Abdulla. 2020. Dynamic power management for value-oriented schedulers in power-constrained HPC system. *Parallel Comput.* 99 (2020), 102686.
[13] Robert C Martin. 2008. *Clean code.* Prentice Hall, Philadelphia, PA.
[14] Robert C Martin. 2017. *Clean architecture.* Prentice Hall, Philadelphia, PA.
[15] Marshall Kirk McKusick, Keith Bostic, Michael J Karels, and John S Quarterman. 1996. *The design and implementation of the 4.4 BSD operating system.* Vol. 2. Addison-Wesley Reading, MA.
[16] Scott McMillan and Michael Knox. 2019. Job Statistics with NVIDIA Data Center GPU Manager and SLURM. https://developer.nvidia.com/blog/job-statistics-nvidia-data-center-gpu-manager-slurm/
[17] Stack Overflow Network. 2022. https://survey.stackoverflow.co/2022/#worked-with-vs-want-to-work-with-language-worked-want-prof
[18] Marcelo Ponce, Ramses Van Zon, Scott Northrup, Daniel Gruner, Joseph Chen, Fatih Ertinaz, Alexey Fedoseev, Leslie Groer, Fei Mao, Bruno C Mundim, et al. 2019. Deploying a top-100 supercomputer for large parallel workloads: The

niagara supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. 1–8.
[19] SchedMD. 2023. SLURM Workload Manager. https://github.com/SchedMD/slurm/tree/slurm-22-05-9-1/src/plugins Accessed: 2023-05-23.
[20] Richard Sharpe. 2002. Just what is SMB? *Oct* 8 (2002), 9.
[21] Vitor RG Silva, Alex FA Furtunato, Kyriakos Georgiou, Carlos AV Sakuyama, Kerstin Eder, and Samuel Xavier-de Souza. 2019. Energy-optimal configurations for single-node HPC applications. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE.
[22] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. 2011. Green governors: A framework for continuously adaptive dvfs. In *2011 International Green Computing Conference and Workshops*. IEEE, 1–8.
[23] Rizwan Ali Tau Leng, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. 2002. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution* 45 (2002).
[24] Storage The International Conference for High Performance Computing, Networking and Analysis. 2018. SLURM User Group Meeting. https://sc18.supercomputing.org/proceedings/bof/bof_pages/bof106.html
[25] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9*. Springer, 44–60.

A DATA SHEETS

A.1 Power measurement

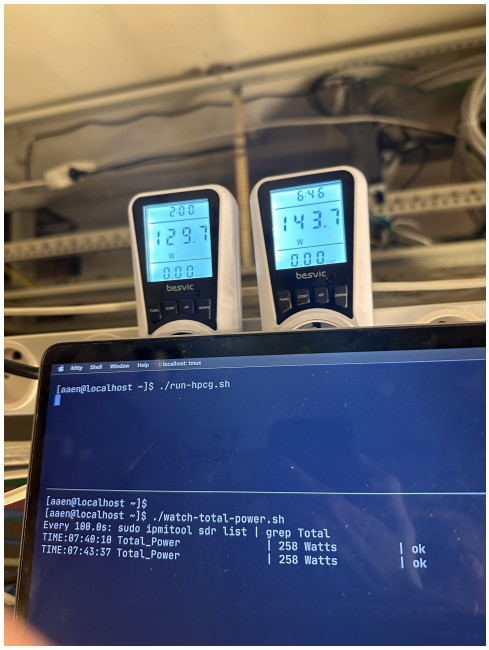


Figure 16: Full image of power measure

A.2 Benchmarks

Cores	GHz	GFLOPS p/ watt	Hyper-thread
32	2.2	0.048767	False
32	2.2	0.048286	True
32	1.5	0.047978	False
32	1.5	0.046933	True
30	2.2	0.045618	True
30	2.2	0.045603	False
30	1.5	0.044614	True
28	2.2	0.044392	False
30	1.5	0.044127	False
28	2.2	0.043690	True
32	2.5	0.043168	False
32	2.5	0.043122	True
28	1.5	0.042526	True
27	2.2	0.042289	True
27	2.2	0.042171	False
28	1.5	0.041438	False
27	1.5	0.041218	True
30	2.5	0.040994	False
27	1.5	0.040803	False
25	2.2	0.040196	False
25	2.2	0.039824	True
30	2.5	0.039537	True
28	2.5	0.038596	True
25	1.5	0.038480	False
28	2.5	0.038408	False
24	2.2	0.038154	False
24	2.2	0.037978	True
25	1.5	0.037609	True
27	2.5	0.037581	True
27	2.5	0.037275	False
24	1.5	0.037072	False
24	1.5	0.036513	True
25	2.5	0.035153	True
25	2.5	0.034758	False
21	2.2	0.034490	False
21	2.2	0.034477	True
24	2.5	0.034234	False
20	2.2	0.033840	False
21	1.5	0.033378	False
20	2.2	0.033332	True
21	1.5	0.033251	True
24	2.5	0.032800	True
20	1.5	0.032278	False
21	2.5	0.031940	False
21	2.5	0.031821	True
20	1.5	0.031744	True
20	2.5	0.031623	True
20	2.5	0.031473	False
18	2.2	0.031221	False
18	2.2	0.031209	True
18	1.5	0.030226	False

Table 4: Gflops per watt part 1

Cores	GHz	GFLOPS p/ watt	Hyper-thread
18	1.5	0.030030	True
8	2.5	0.030025	False
16	2.2	0.029694	False
18	2.5	0.029675	False
16	2.2	0.029481	True
8	2.2	0.029461	True
18	2.5	0.029385	True
9	2.2	0.029378	False
8	2.2	0.029355	False
8	2.5	0.029334	True
10	2.2	0.029024	False
10	2.5	0.028914	False
10	2.2	0.028787	True
9	2.2	0.028717	True
6	2.5	0.028709	True
9	2.5	0.028601	True
12	2.2	0.028460	False
9	2.5	0.028423	False
16	2.5	0.028402	False
12	2.5	0.028379	True
12	2.5	0.028355	False
16	2.5	0.028317	True
10	2.5	0.028312	True
15	2.2	0.028312	True
12	2.2	0.028258	True
14	2.2	0.028235	True
16	1.5	0.028144	False
14	2.2	0.028097	False
6	2.5	0.027928	False
15	2.2	0.027785	False
7	2.5	0.027625	False
7	2.5	0.027594	True
14	1.5	0.027554	False
16	1.5	0.027520	True
15	2.5	0.027500	False
15	2.5	0.027353	True
7	2.2	0.027228	True
14	1.5	0.027054	True
7	2.2	0.027033	False
14	2.5	0.027008	False
12	1.5	0.026994	False
15	1.5	0.026925	True
15	1.5	0.026879	False
14	2.5	0.026860	True
6	2.2	0.026797	True
10	1.5	0.026599	False
8	1.5	0.026577	True
10	1.5	0.026549	True
6	2.2	0.026512	False
8	1.5	0.026397	False
9	1.5	0.026236	False
12	1.5	0.026219	True
9	1.5	0.026151	True
5	2.5	0.026056	True
5	2.5	0.026028	False

Table 5: Gflops per watt part 2

Cores	GHz	GFLOPS p/ watt	Hyper-thread
4	2.5	0.025157	True
4	2.5	0.024648	False
5	2.2	0.023307	False
7	1.5	0.022859	True
5	2.2	0.022752	True
7	1.5	0.022643	False
4	2.2	0.022313	False
6	1.5	0.021718	True
6	1.5	0.021681	False
4	2.2	0.021294	True
3	2.5	0.020024	False
3	2.5	0.019348	True
5	1.5	0.018599	True
5	1.5	0.018445	False
4	1.5	0.016654	False
4	1.5	0.016160	True
2	2.5	0.016094	False
2	2.5	0.015917	True
3	2.2	0.015503	True
1	2.5	0.014558	False
1	2.5	0.014548	True
3	2.2	0.014462	False
2	2.2	0.011852	False
3	1.5	0.011503	True
2	2.2	0.011355	True
3	1.5	0.011177	False
1	2.2	0.010560	True
1	2.2	0.010462	False
1	1.5	0.007571	True
1	1.5	0.007569	False
2	1.5	0.007236	False
2	1.5	0.007150	True

Table 6: Gflops per watt part 3

A.3 Gflops per watt configurations

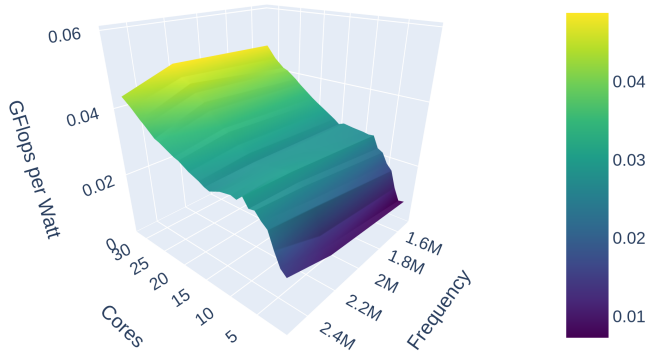


Figure 17: Full image of no hyper-threading testing results

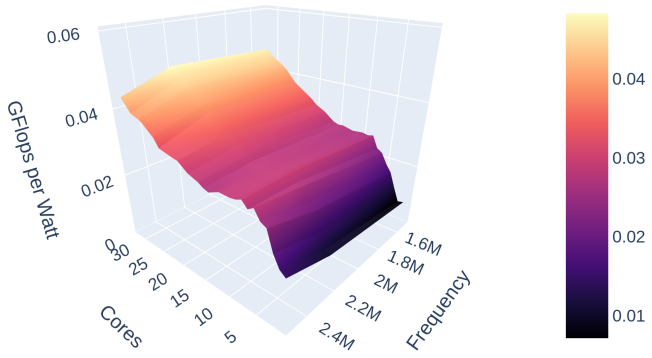


Figure 18: Full image of hyper-threading testing results

B CODE

The code for this project is all open-sourced and on GitHub.

Click here to go to the repository or you can find Chronus at:

`https://github.com/AndersSpringborg/chronus`

I made a fork of Slurm, and implemented `job_submit_eco` in the fork. To make it simple to see what I added, I made a Pull Request that shows the differences from what I added. Click here to get to the repository, or find it in the link below:

`https://github.com/AndersSpringborg/slurm`

The Pull Request can be found here:

`https://github.com/AndersSpringborg/slurm/pull/1/files`

C WORK PROCESS

This appendix provides an overview of the process followed to complete the study, focusing on the significant challenges encountered and the solutions implemented.

Project management

The project was meticulously planned and managed to ensure a steady pace of development and to meet all the deadlines. A detailed schedule was created at the beginning of the project, outlining the tasks to be completed each week. This schedule served as a roadmap, guiding the project's progress and keeping it on track.

The project was divided into several phases, each focusing on a specific aspect of the project. The first phase, spanning weeks 7 to 9, was dedicated to developing the model for predicting energy efficiency configurations. I did this while waiting to get hardware, to start setting up my cluster. This involved parsing data, and creating a tool that could operate in multiple configurations. This later became Chronus.

The second phase, from weeks 10 to 14, focused on the implementation of the project. This involved setting up the hardware, installing dependencies for SLURM on a node in the basement, building SLURM on the machine, and initializing the plugin. Challenges encountered during this phase, such as the unsuccessful build of SLURM on Rocky Linux 9.1, were addressed by reformatting the system to Rocky 8.7 and retrying the build.

Week 15 to 17 were dedicated to developing the eco plugin. This phase involved writing debug messages in slurmctld (master) and slurmd (compute) to fixing some of the dependencies, that were installed incorrectly.

The third phase, from weeks 18 to 19, was dedicated to testing and benchmarking. This phase ensured that all components of the project functioned as expected and that the system was ready for the final phase.

The final phase, from weeks 20 to 23, was dedicated to finalizing the paper. Each section was tackled one at a time, beginning with the eco plugin (Section 3) and implementation details (Section 4), followed by experiments (Section 5) and future work (Section 6). After completing these sections, the related works section was revisited to ensure it remained relevant and up-to-date. The conclusion, abstract, and process sections were written, summarizing the overall project, its findings, and the methodology followed.

This structured approach to project management ensured that all tasks were completed in a timely manner and that the project progressed smoothly. The detailed schedule provided a clear roadmap for the project, allowing for efficient time management and ensuring that all aspects of the project were adequately addressed.

Figure 19 presents a Gantt chart of schedule.

I used a Kanban board throughout, to keep track of what I was doing that week. At the start of the week, I would split the week into smaller task, and always keep track on my current task, and the rest of the tasks for the week.

Slurm Integration and Development

The integration of Slurm posed a series of challenges, as it required installing dependencies and building the software on a specific machine. The initial attempt to build Slurm on Rocky Linux 9.1 was unsuccessful, leading to a decision to reformat the system and try again the following week.

Upon formatting the system to Rocky 8.7, the necessary dependencies were installed, and Slurm was successfully built on the machine. The next steps involved implementing (C code in Slurm). Although optional, an attempt was made to get the Chronus CLI to run Slurm.

Once the Slurm plugin was initialized, several tasks were performed, including writing debug messages in `slurmctld` (master) and `slurmd` (compute), fixing the `openmpi` installation to support PMIx, and setting frequency using a hardcoded approach on the compute node. These steps were crucial in preparing a demo for the project supervisors and progressing the development.

Supercomputer Club and Competition Experience

During this study, I also founded a supercomputer club focused on high-performance computing (HPC). The club provided an opportunity to deepen my understanding of HPC concepts by teaching others and engaging in practical experiences. This involvement proved valuable in complementing the research and development process of the paper.

The supercomputer club went to California for 9 days to attend a competition, where we competed against other teams in various HPC challenges. The competition allowed me and club members to learn more about the HPC space, network with professionals

and academics, and gain hands-on experience in optimizing and deploying HPC systems.

The knowledge and skills acquired during the competition and through the supercomputer club's activities contributed to my knowledge in the field, ultimately benefiting the research and development process for the paper. The club's experiences helped to contextualize the challenges faced during the study and offered additional insights into the practical applications of HPC solutions.

Figure 19 shows how I have used my time

D TESTING

The testing process was comprehensive and thorough, ensuring that all components of the project functioned as expected. This process was divided into several stages, including unit testing, integration testing, and end-to-end testing.

Unit Testing

Unit testing was performed on Chronus to verify that each individual component worked as intended. This process involved testing each command individually, including benchmark, init-model, load-model, and slurm-config. The integrations with HPCG, lscpu, CSV repository, SQLite repository, IPMI service, and optimizers were also tested.

To facilitate these tests, system processes were mocked and a temporary folder was used for file integrations. The C code was tested in a small program with just a main function before integrating it into Slurm. This step was crucial as Slurm took a significant amount of time to build. Once the C code was confirmed to work correctly in the small program, it was then tested in Slurm while monitoring the log files for any potential issues, specifically looking for print statements.

Integration Testing

Integration testing was performed to ensure that all the components of the project worked together seamlessly. This process involved using temporary files to test all integrations against the file system. The temporary files were written to a folder and then deleted after the test was completed.

Before executing commands in Chronus, they were tested manually to ensure their correctness. These commands included Slurm commands and lscpu. `job_submit_eco` was tested with a small bash script that initially returned a JSON, allowing control over the output. Once this was successful, it was pointed to the real implementation of Chronus for further testing.

End-to-End Testing

End-to-end testing was performed extensively on the command-line interface (CLI) by running all the scripts, including benchmark, init-model, load-model, and slurm-config. The tests verified that these scripts worked with Slurm by checking squeue and scontrol to confirm their presence. The output of the slurm-config was also examined.

Power Measurement Tool Testing

The power measurement tool was tested by setting up a digital wattmeter and observing the output while running HPCG. Simultaneously, the IPMI sensors were monitored and the results were noted. This process ensured that the power measurement tool was accurately capturing the energy consumption.

Conclusion

The testing process was successful, with all tests passing. The tests were run at every pull request, ensuring that the code remained functional throughout the development process. A linter and a formatter were also used to maintain code quality and consistency.

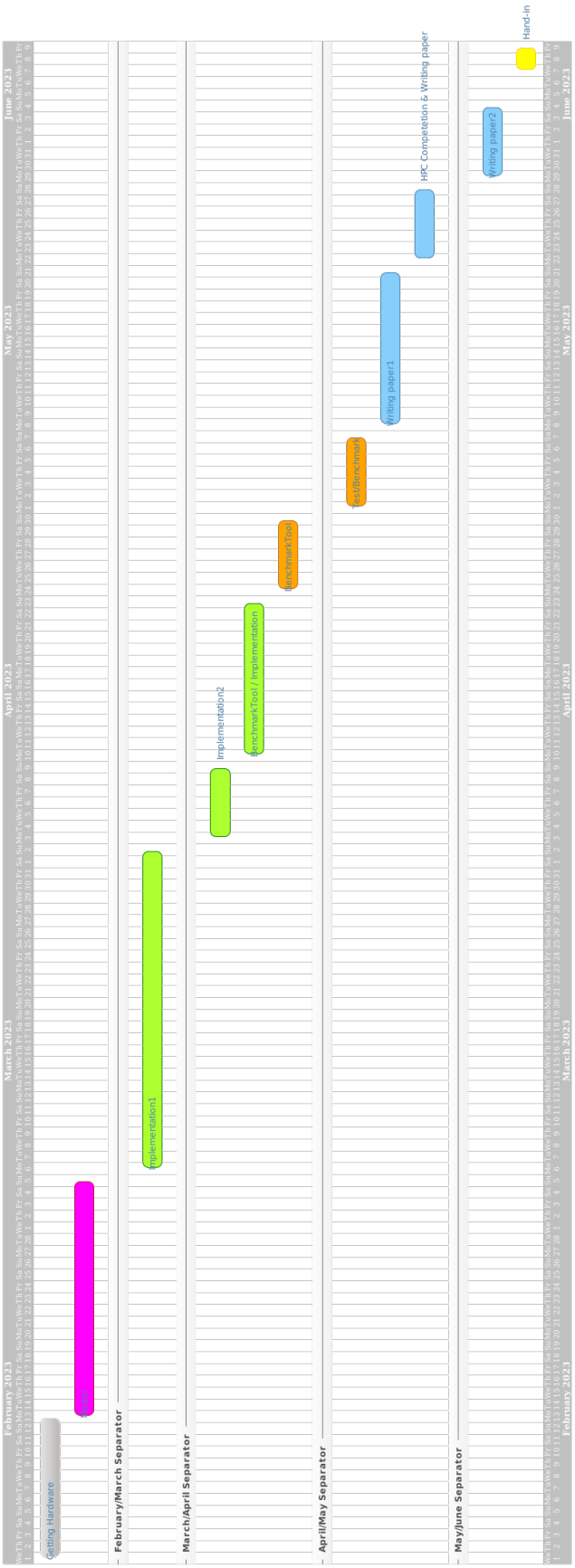


Figure 19: Gantt diagram of my process

Over the course of the project, 19 pull requests were made and 80 unit tests were conducted, demonstrating a rigorous and thorough testing process. This comprehensive testing approach ensured that all components of the project worked individually and together as a system, providing confidence in the reliability and accuracy of the final product.