Feature : Automatic Job Resubmission

1. Introduction

This report details the design, implementation, and demonstration of "Slurm Longrun," a Python-based utility developed to address the challenges of running long-duration computational jobs on Slurm workload managers. The project focuses on enabling automatic job resubmission in cases of premature termination due to factors such as walltime limits, node failures, or preemption, thereby enhancing the reliability and efficiency of long-running workflows. The utility is published as a package on PyPI and is available for installation via `pip install slurm-longrun`.

2. Problem Statement

In high-performance computing (HPC) environments managed by Slurm, jobs are typically subject to walltime limits. While these limits are essential for resource allocation and fairness, they can present a challenge for computationally intensive tasks, such as large-scale machine learning model training , which may require execution times exceeding a single job's allocated walltime. Manually re-submitting jobs after a timeout or other failures can be time-consuming and inefficient. Furthermore, unexpected events like node failures or preemption can also prematurely terminate jobs, potentially leading to lost progress if not handled effectively.

This project addresses the need for an automated and resilient mechanism to manage and continue long-running Slurm jobs without constant manual oversight, specifically focusing on automatic resubmission under common failure conditions.

3. Solution Overview

Slurm Longrun is a Python package designed to wrap the standard `sbatch` command, providing an intelligent layer for monitoring and automatically resubmitting jobs. Its primary goal is to allow users to run workloads that exceed typical single-job walltimes by transparently handling job interruptions and restarts.

Key features of Slurm Longrun include:

- Automatic Resubmission: Automatically re-submits jobs that terminate due to `TIMEOUT`, `DEADLINE`, `PREEMPTED`, `NODE_FAIL`, or `REVOKED` statuses.
- Configurable Retry Limits: Users can specify the maximum number of automatic resubmissions.
- Terminal Detachment: An optional feature to detach the monitoring process from the terminal, allowing jobs to continue being monitored even after the user logs out.
- Integrated Logging: Utilizes `loguru` for clear and informative logging of job submission, monitoring, and resubmission events.

- Checkpointing Support: Provides an environment variable (`SLURM_LONGRUN_INITIAL_JOB_ID`) to facilitate checkpointing and state recovery within job scripts, ensuring seamless continuation from the last saved state. It can be accessed once the first **re**submission occurs.

4. Implementation Details

4.1. Architecture Overview

The Slurm Longrun solution is built upon a modular architecture, composed of the following Python modules:

4.2. `cli.py` (Command-Line Interface)

This module defines the `sbatch_longrun` command-line utility using the `click` library. It parses arguments specific to the Longrun wrapper (e.g., `--use-verbosity`, `--detached`, `--max-restarts`) and forwards all other arguments directly to the underlying `sbatch` command.

- **`main()` function**: The entry point for the `sbatch_longrun` command. It sets up logging based on verbosity, initializes the `SlurmRunner`, and either runs the monitoring process directly or detaches it into the background if `--detached` is specified.

4.3. `runner.py` (Core Logic)

The `SlurmRunner` class in this module encapsulates the main workflow of submitting, monitoring, and resubmitting Slurm jobs.

- **`__init__(self, sbatch_args, max_restarts, detached)`**: Initializes the runner with the base sbatch arguments, maximum restart count, and detachment preference.
- **`submit()`**: Executes the initial sbatch command and parses the job ID. It raises a `RuntimeError` if submission fails.
- **`fetch_info(job_id)`**: Gathers detailed information about a job by merging outputs from `sacct` and `scontrol`.
- **`parse_status(info)`**: Extracts and normalizes the job state into a `JobStatus` enum.
- **`monitor(job_id)`**: This is the core monitoring loop. It continuously polls Slurm for the job's status. If the job reaches a terminal state that requires resubmission (e.g., `TIMEOUT`), and the maximum restart limit has not been reached, it resubmits the job with `--open-mode=append` to preserve logs and continues monitoring the new job ID. The polling interval is dynamically adjusted based on the remaining walltime.

- `run()`: Orchestrates the entire process: submits the initial job, sets the `SLURM_LONGRUN_INITIAL_JOB_ID` environment variable, and then either detaches the monitoring process or runs it in the foreground.

4.4. `common.py` (Job Status Definitions)

This module defines the `JobStatus` Enum, which enumerates various possible states of a Slurm job. It includes properties to determine if a status is `is_final` (terminal), `is_success`, or `should_resubmit`. This clear categorization is crucial for the monitor logic to decide when to stop polling or when to trigger a resubmission.

4.5. `logger.py` (Logging Configuration)

This module configures the Loguru logger, providing a consistent and readable output format for all Slurm Longrun operations. It supports different verbosity levels (`DEFAULT`, `VERBOSE`, `SILENT`).

4.6. `utils.py` (Helper Functions)

A collection of utility functions supporting the main logic:

- `run_command(cmd)`: Executes a shell command and returns its stdout, handling errors.
- `run_sbatch(args)`: A specialized wrapper for `run_command` to submit sbatch jobs and extract the job ID.
- `get_scontrol_show_job_details(job_id)`: Parses `scontrol show job` output into a dictionary.
- `get_sacct_job_details(job_id)`: Parses `sacct` output into a list of dictionaries.
- `time_to_seconds(timestr)`: Converts Slurm time strings (e.g., "D-HH:MM:SS") into total seconds.
- `run_detached(func, *args, **kwargs)`: Forks a child process and detaches it from the terminal, used for the `--detached` option.
- `detach_terminal()`: Redirects stdin, stdout, and stderr to `/dev/null` for detached processes.

5. Demonstration and Validation

To demonstrate and validate the effectiveness of Slurm Longrun, an example scenario [1] involving the training of a Large Language Model (LLM) was set up. This demonstration highlights the integration of checkpointing within the LLM

[1] I refer to the example provided in `example/assignment2_example/assignment_2`.

training script (`train.py`) to allow for seamless recovery and continuation of training across multiple job submissions.

5.1 Submission Command

The LLM training job was submitted using `sbatch_longrun` with a walltime of 3 minutes and a `SIGTERM` signal sent 20 seconds before the walltime limit to trigger state saving.

```
sbatch_longrun --signal=SIGTERM@20 example/assignment2_example/run_job.sbatch
```

5.2 `run_job.sbatch` Script

This Slurm batch script defines the resources for the training job and executes the Python training script. Crucially, it exports all environment variables, including `SLURM_LONGRUN_INITIAL_JOB_ID`, which is used by the `train.py` script for naming checkpoints.

5.3 Checkpointing in `train.py`

The train.py script implements `store_state` and `recover_state` functions to save and load the model's state and the last completed epoch. The `STATE_PATH` for checkpoints is dynamically generated using `SLURM_LONGRUN_INITIAL_JOB_ID`, ensuring that all resubmitted jobs use the same checkpoint file. A `SIGTERM` handler is registered to gracefully save the training state when the job receives a termination signal, typically sent by Slurm before a walltime limit is reached.

5.4 Execution Logs: Evidence of Functionality

The demonstration produced two sets of logs, which serve as direct evidence of the system's behavior and validate its core functionality:

5.4.1 Slurm Longrun Monitor Logs These logs show the `sbatch_longrun` wrapper's activity. The job (initial ID 454600) repeatedly TIMEOUTs and is successfully Resubmitted by Slurm Longrun. This continuous resubmission across several job instances, until the final `COMPLETED` status, directly demonstrates the automatic job resubmission mechanism.

```
2025-05-21 11:31:10 | SUCCESS  : Job submitted with ID: 454600
2025-05-21 11:31:10 | INFO     : Monitoring job 454600 (submission 1/999)
2025-05-21 11:35:51 | INFO     : Job 454600 reached final state: TIMEOUT
2025-05-21 11:35:52 | SUCCESS  : Resubmitted based on status=TIMEOUT job with ID: 454609
...
2025-05-21 11:55:04 | INFO     : Job 454644 reached final state: COMPLETED
2025-05-21 11:55:04 | SUCCESS  : Job completed successfully.
```

5.4.2 Job Script Output Logs These logs, generated by the `train.py` script, confirm the effective integration of the checkpointing mechanism. The initial Starting from scratch message, followed by Saving state upon `SIGTERM`, and then Recovered model state in subsequent resubmissions, provides clear evidence that training successfully resumed from where it left off, rather than starting anew. The final "Training completed" entry confirms the successful overall execution of the LLM training.

```
2025-05-21 11:31:55 - INFO - Setting up DataLoaders...
2025-05-21 11:31:57 - INFO - Setting up Model...
2025-05-21 11:32:31 - INFO - Starting from scratch, no state found
2025-05-21 11:32:33 - INFO - Registered SIGTERM handler
2025-05-21 11:32:33 - INFO - Starting training!
2025-05-21 11:32:34 - INFO - Step: 1 | Loss: 12.03 | TFLOPs: 148.68
...
2025-05-21 11:33:40 - INFO - Step: 120 | Loss: 7.89 | TFLOPs: 388.77
slurmstepd: error: *** STEP 454600.0 CANCELLED AT 2025-05-21T11:33:42 ***
2025-05-21 11:33:43 - INFO - [Received SIGTERM] : Saving state...
2025-05-21 11:33:55 - INFO - Finished saving state.
...
slurmstepd: error: *** JOB 454600 CANCELLED DUE TO TIME LIMIT ***

2025-05-21 11:36:33 - INFO - Setting up DataLoaders...
2025-05-21 11:36:35 - INFO - Setting up Model...
2025-05-21 11:37:16 - INFO - Recovered model state
2025-05-21 11:37:16 - INFO - Registered SIGTERM handler
2025-05-21 11:37:16 - INFO - Starting training!
2025-05-21 11:37:18 - INFO - Step: 225 | Loss: 7.56 | TFLOPs: 115.39
...
2025-05-21 11:39:18 - INFO - [Received SIGTERM] : Saving state...
...
2025-05-21 11:53:53 - INFO - Step: 995 | Loss: 4.38 | TFLOPs: 394.14
2025-05-21 11:53:56 - INFO - Step: 1000 | Loss: 4.74 | TFLOPs: 390.25
2025-05-21 11:53:56 - INFO - Training completed
```

5.4.3 Comparative Analysis of Loss Evolution To further validate the functional performance of the automatic resubmission mechanism, a comparative analysis was conducted between a single, long-running job (configured with a sufficiently high walltime to complete all 1000 epochs in one go) and the `sbatch_longrun` job, which utilized multiple resubmissions to achieve the same total training duration.

Both scenarios exhibited a similar overall trend in loss reduction. However, the `sbatch_longrun` job showed slightly higher variability in its loss' trajectory. This is likely due to the demonstration only saving the model's state (weights) and not the optimizer's state. Without the optimizer state, it effectively restarts

with each resubmission, introducing discontinuities. Despite this, the overall loss trend remained very similar, with the resubmission-enabled job even achieving a slightly lower final loss.

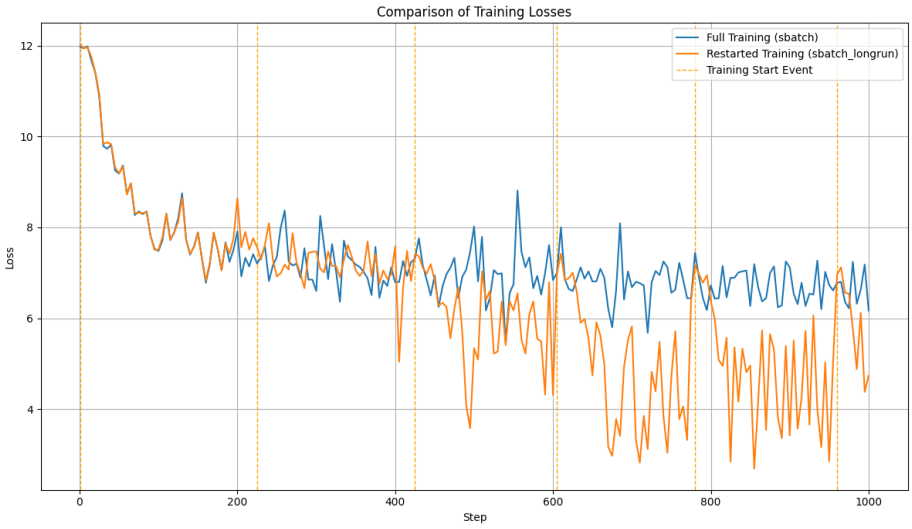

Figure 1: Comparing the loss of `sbatch` and `sbatch_longrun`

6. Conclusion

Slurm Longrun successfully provides a robust and automated solution for managing long-running jobs on Slurm. By abstracting the complexities of job resubmission and integrating seamlessly with checkpointing mechanisms, it significantly improves the user experience for researchers and engineers running demanding computational tasks. The demonstration, supported by the provided execution logs and comparative loss analysis, clearly illustrates its ability to handle job timeouts and resume progress, making it a valuable tool for ensuring the completion of extended workflows without manual intervention.

This project effectively addresses the challenge of walltime limits and unforeseen job interruptions, contributing to more efficient and reliable utilization of HPC resources.