WRATH: Workload Resilience Across Task Hierarchies in Task-based Parallel Programming Frameworks

Sicheng Zhou*, Zhuozhao Li*, Valérie Hayot-Sasson†, Haochen Pan†, Maxime Gonthier†,
J. Gregory Pauloski†, Ryan Chard‡, Kyle Chard†‡, Ian Foster‡†
*Department of Computer Science, Southern University of Science and Technology, Guangdong, China
†Department of Computer Science, University of Chicago, Chicago, IL, USA
‡Data Science and Learning Division, Argonne National Laboratory, Lemont, IL, USA

***Abstract*—Failures in Task-based Parallel Programming (TBPP) can severely degrade performance and result in incomplete or incorrect outcomes. Existing failure-handling approaches, including reactive, proactive, and resilient methods such as retry and checkpointing mechanisms, often apply uniform retry mechanisms regardless of the root cause of failures, failing to account for the unique characteristics of TBPP frameworks such as heterogeneous resource availability and task-level failures. To address these limitations, we propose WRATH, a novel systematic approach that categorizes failures based on the unique layered structure of TBPP frameworks and defines specific responses to address failures at different layers. WRATH combines a distributed monitoring system and a resilient module to collaboratively address different types of failures in real time. The monitoring system captures execution and resource information, reports failures, and profiles tasks across different layers of TBPP frameworks. The resilient module then categorizes failures and responds with appropriate actions, such as hierarchically retrying failed tasks on suitable resources. Evaluations demonstrate that WRATH significantly improves TBPP robustness, tripling the task success rate and maintaining an application success rate of over 90% for resolvable failures. Additionally, WRATH can reduce the time to failure by 20%-50%, allowing tasks that are destined to fail to be identified and fail more quickly.**

***Index Terms*—Resilience, task-based parallel programming, hierarchical retry, failure categorization**

I. INTRODUCTION

Task-based parallel programming (TBPP) is a programming paradigm in which a computational workload is divided into discrete units of work called tasks. These tasks can execute concurrently on the same or different computing nodes, subject to constraints resulting from shared data and communication. Particularly as parallel systems and applications increase in complexity and scale, it becomes crucial to be able to detect and recover from various forms of task failure, such as can result from hardware malfunctions, software bugs, and incompatible environments (e.g., due to different libraries, system modules, and even Python versions).

Commonly used TBPP frameworks, such as Dask [1], Parsl [2], and Ray [3], incorporate various basic resilience mechanisms to mitigate task failures, such as *task retry*, in which the system automatically attempts to rerun a failed task, often with a configurable number of retries, and *checkpointing*, which involves saving the state of a computation periodically and resuming from the last checkpoint. However, these mechanisms are often insufficient for handling the complexities inherent in large-scale distributed systems, where failures may stem from heterogeneous resource availability or task-specific issues. Hence, failures in TBPP applications are diverse, distributed, and may occur at multiple levels—from application-specific bugs, to resource starvation or failures in the distributed hardware on which tasks are executed. Basic resilience methods typically apply uniform retry mechanisms without distinguishing between failure types. This one-size-fits-all approach can lead to inefficient responses, such as needlessly retrying tasks with application-specific bugs that will fail again or overlooking systemic issues that require a more coordinated recovery effort.

In this paper, we explore methodologies and techniques for handling failures in TBPP frameworks. We review failures at different levels of TBPP frameworks, define categories of failures with similar characteristics and necessary responses, examine existing approaches to failure management, and propose improvements and best practices to enhance the robustness of TBPP frameworks. We present a new approach called WRATH (Workload Resilience Across Task Hierarchies) for addressing failures in TBPP applications. WRATH categorizes failures at four layers of TBPP frameworks and defines specific responses to address failures at different levels. WRATH includes a hierarchical *monitoring system* and an intelligent *resilience module*. Specifically, the monitoring system leverages distributed monitoring agents to gather valuable information across the hierarchies of TBPP frameworks. Meanwhile, the resilience module employs new failure categorization methods to identify failures from monitoring information and map them to appropriate handling mechanisms, such as immediate failure responses or retries. Additionally, the resilience module implements a *hierarchical retry* mechanism that dynamically retries failed tasks across different resource pools, thereby increasing the likelihood of successful task execution.

The novelty of WRATH lies in its introduction of a framework that categorizes failures based on four distinct layers of

TBPP frameworks, contrasting with existing approaches that typically apply a uniform retry mechanism for all types of failures. Additionally, WRATH considers the hierarchical nature of TBPP frameworks by proposing a distributed monitoring system and a hierarchical retry mechanism, enabling more effective and tailored responses to diverse failure scenarios.

The key contributions of this paper include:

- A comprehensive survey of failure types in TBPP frameworks and categorization of these types at different layers of the stack;
- Implementation of a distributed monitoring system for real-time data collection across the TBPP stack, facilitating a more informed and adaptive response to failures;
- Development of a resilience module based on the proposed failure categorization methods, which maps identified failures to appropriate handling mechanisms, as well as a hierarchical retry mechanism that dynamically reallocates failed tasks to different resource pools;
- A thorough evaluation of WRATH using a benchmark system with real TBPP applications to demonstrate the effectiveness of WRATH in terms of success rate, overhead, and "fail fast" for non-resolvable failures.

The rest of this paper is as follows: §II introduces background and motivation; §III categorizes failures across the layers of TBPP frameworks; §IV presents methods to detect failures; §V describes ways to respond to failures; §VI provides the detailed implementation of WRATH; §VII presents experiments to evaluate our solution; §VIII discusses related work; and §IX concludes the paper with future remarks.

II. Background and Motivation

In this section, we introduce the fundamentals of task-based parallel programming (TBPP) frameworks, define failures, and highlight the motivations for our approach. We assume here, as is common [1], [2], [4]–[6], atomic tasks: i.e., tasks that either complete and generate output or fail with no output. This property means that resilience can be achieved by re-executing failed tasks.

A. TBPP Frameworks

TBPP is a programming paradigm that divides large computational problems into smaller units known as *tasks*. TBPP frameworks facilitate the definition of tasks with explicit dependencies and enable their scheduling across various computing resources. The interleaved execution of these tasks may be subject to constraints arising from control- and data-flow dependencies [7].

Frameworks such as Dask, Parsl, and Ray abstract low-level parallelization details and offer high-level constructs that enable developers to express parallelism without needing to manage the underlying hardware directly. These frameworks typically include a task scheduler that dynamically assigns tasks to available computing resources based on task priority, dependencies, and resource availability.

While TBPP frameworks offer numerous advantages, such as simple programming models, high performance, and scalability, detecting and responding to failures is difficult as tasks are executed on heterogeneous distributed computing resources. Failures can occur at four layers (i.e., application, framework, runtime, and environment layers, as shown in Figure 1) of a TBPP system [7], each presenting challenges for fault detection, management, and recovery.

The **Application Layer** is where tasks are defined. This layer involves the coding of tasks, including the algorithms, data structures, and logic that will be executed. Tasks at this layer may have explicit dependencies on other tasks. While different TBPP frameworks enable task definition and construction using different interfaces (e.g., YAML, Java, Python), they share common features such as defining task dependencies as directed acyclic graphs (DAGs).

The **Framework Layer** orchestrates the execution of tasks defined in the application layer. Most TBPP frameworks rely on a *central manager* to manage the task dependencies, scheduling, monitoring, and failure handling, although (even fully) decentralized approaches can be used [8]. This layer ensures that tasks are executed in the correct sequence and that computational resources are utilized efficiently. The framework layer may respond to failures from the runtime layer by retrying task execution.

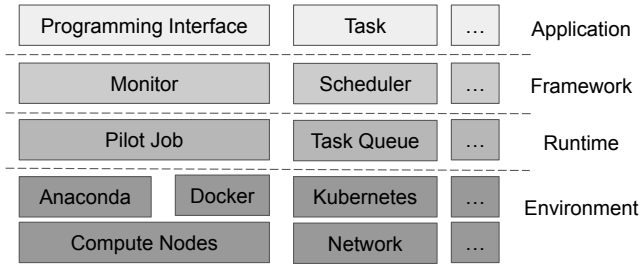


Fig. 1: Typical architecture of TBPP frameworks. In the *Application Layer*, users define applications into tasks using the provided programming interfaces. The *Framework Layer* orchestrates the execution of tasks. The *Runtime Layer* allocates resources to tasks. The *Environment Layer* manages the underlying infrastructure and package dependencies.

The **Runtime Layer** is responsible for managing task execution on underlying computational resources. TBPP frameworks often rely on the *pilot job model* [9], in which placeholder jobs are submitted to computing resources to initialize the execution environment and hold the resources. Usually, the pilot job will start a *node manager* process on each node in the job, which is responsible for receiving tasks and assigning them to *worker* processes responsible for executing the tasks.

The **Environment Layer** includes the underlying infrastructure and the runtime environment in which the application, framework, and runtime system operate. Modern TBPP frameworks often leverage containers (e.g., Docker or Singularity) or environment management software (e.g., conda or virtual environment) to allow developers to create consistent and portable

environments. This layer also enhances the reproducibility of computational tasks, a key requirement in scientific computing and other fields where results must be verified and validated across different platforms.

B. Motivation and the WRATH Approach

As mentioned in §I, existing TBPP frameworks typically rely on *retry* and *checkpointing* approaches to ensure resilience. However, these approaches often adopt a *flat* structure, meaning they treat all failures uniformly without considering their context or root causes within the TBPP layers. This limitation prevents the identification and resolution of failures in a manner that is tailored to the specific characteristics and complexities of the system. Therefore, we are motivated to design a failure-handling system that considers the unique characteristics of TBPP frameworks, including the heterogeneity and layered structure.

In this paper, we introduce WRATH, a failure-handling approach that aims to enable resilient computing for TBPP frameworks. By resilient computing, we refer to the ability of a computing system to continue functioning properly in the presence of failures. WRATH proposes to categorize various failures based on the layers of TBPP frameworks. To monitor the failures in different layers, WRATH integrates a hierarchical monitoring system to gather execution and resource data and report failures as they occur. Based on the categorization methods and the monitoring system, WRATH designs a dynamic resilience module that can intelligently retry failed tasks on the most appropriate resource pools.

In the next three sections, we will detail how WRATH characterizes failures in TBPP frameworks, the design of the hierarchical monitoring system, and the implementation of the dynamic resilience module.

III. CHARACTERIZATION OF FAILURES IN TBPP

Despite the robustness of modern TBPP frameworks, failures can occur at multiple levels, disrupting execution and potentially leading to incorrect results, wasted resources, and performance degradation. Understanding the nature of failures is crucial for designing resilient systems that can either recover from them or minimize their impact effectively.

A. Failure Root Causes

Different types of failures may occur at different layers in TBPP frameworks. We summarize them in Table I.

Failures at the **Application Layer** are **User Failures** due to mistakes or incorrect assumptions made by users when writing their application code and tasks. These failures may be incorrect results, crashes, or inefficient execution. Typical causes include *Syntax Errors*, where mistakes in the code violate programming language syntax rules; *Logic Errors*, such as incorrect use of loops or mathematical calculations, accessing out-of-bounds array indices, or incorrect data types; and *Random Seed Errors*, where the failure is sporadic. An example of Random Seed Error can be seen in a molecule design application [10], during the first period of simulation in which molecule assumptions are generated randomly for further calculation. These assumptions can cause errors in simulation and subsequent processing. But after regeneration, the errors may resolve.

Failures at the **Framework Layer** are TBPP **System Failures** in the components responsible for orchestrating task execution, as well as issues related to dependencies between tasks in the Framework Layer. These failures can significantly impact task scheduling, monitoring, and execution. Examples of system failures include *Monitor Loss*, where the component responsible for overseeing the execution of tasks and maintaining the state information becomes unavailable or unresponsive; *Manager Loss*, where the component responsible for managing task scheduling, resource allocation, and overall workload coordination fails; and *Dependency Failures*, where the frameworks fail to manage dependencies between tasks.

Failures at the **Runtime Layer** are **Resource Failures** that occur while managing and using computational resources required to execute tasks. These errors can affect the availability, allocation, and effective utilization of resources, leading to disruptions in task execution. Examples include *Resource Starvation* where tasks do not receive sufficient resources (CPU, memory, storage, etc.) for execution, and *Pilot Job Initialization Failure* where the pilot job responsible for provisioning and managing computational resources fails to start or initialize correctly.

Failures in the **Environment Layer** are **Hardware & Environment Failures** related to the physical infrastructure and the overall runtime environment in which tasks are executed. Hardware failures are particularly common. For example, 42.1% of all failures observed in the Blue Waters supercomputer over 261 days in 2013 were hardware failures [11], as were 64% of failures seen in 22 HPC systems over nine years (1996-2005) [12]. Such errors can cause significant disruptions, including application or complete system failures. Examples include *Hardware Shutdown*, where components such as servers, storage devices, or network equipment unexpectedly power down or fail, and *Runtime Environment Mismatch*, where the software environment required for executing tasks does not match that available on execution nodes.

B. Failure Manifestation

Failure manifestation refers to the observable signs or indicators that a failure has occurred. These manifestations help in identifying, diagnosing, and addressing errors. They can appear in various forms, such as exception messages, service heartbeats loss, resource usage logs, and other log messages.

Exception Messages are error messages generated by the system when it encounters an unexpected condition or error. These messages typically provide information about the nature of the error and its location in the code. Examples are SyntaxError, FileNotFoundException, and OutOfMemoryError.

A **Service Heartbeat** is a periodic signal sent by a service to indicate that it is operational. The absence of a heartbeat can indicate that the service has failed or become unresponsive.

TABLE I: Failure types and detection strategies. FTL stands for Failure Taxonomy Library; RP for Resource Profiling; and RC for depending on the Root Cause.

Layer & Failure Type	Example Root Cause	Examples & Description	Detection Strategy	Is Failure Retriable
Application Layer (User Failures)	Syntax Errors	Mistakes that violate programming language syntax.	FTL	No
	Logic Errors	Array index out-of-bounds or incorrect data types.	FTL	No
	Random Seed Errors	Molecule Design initialization issue.	/	Yes
Framework Layer (System Failures)	Monitor Loss	Task overseeing component becomes unavailable.	FTL	Yes
	Manager Loss	The component responsible for managing tasks fails.	FTL	Yes
	Dependency Failures	Single task failure causes multiple dependent tasks to fail.	RC	RC
Runtime Layer (Resource Failures)	Resource Starvation	Insufficient memory or CPU to execute a task.	RP	Yes
	Pilot Job Init. Failures	The pilot job fails to initialize correctly.	RP	Yes
Environment Layer (Hw. & Env. Failures)	Hardware Shutdown	Components such as servers or network devices fail.	FTL + RP	Yes
	Runtime Env. Mismatches	Missing required software versions or libraries in env.	FTL	No

Resource Usage Logs track the utilization of system resources such as CPU, memory, disk, and network. Abnormal patterns in resource usage can indicate failures or performance issues. For example, a steady increase in memory usage without release may indicate a memory leak, and a spike in CPU usage may indicate resource contention and a potential bottleneck.

Other Log Messages generated by the system can also provide insights into failures. TBPP frameworks have different levels of log messages, including debug information, warnings, errors, and informational messages. Feedback from submitted jobs (i.e., standard output and error files) also contains useful information for failure detection.

IV. Monitoring across Task Hierarchies

In this section, we describe the monitoring system of WRATH and describe how it can detect failures.

As we have described, failures may occur at different layers of TBPP frameworks. We therefore implement WRATH with a hierarchical monitoring system to observe the execution of tasks in TBPP frameworks. The hierarchical monitoring system consists of *task monitoring agents*, *system monitoring agents*, *a centralized monitoring database*, and *a communication radio*.

Task monitoring agents are processes distributed across the system to observe the behavior of individual tasks at different levels of the task hierarchy. For example, an agent at each compute node is responsible for monitoring real-time metrics such as CPU, memory usage, and execution time, and an agent at the central TBPP manager collects information such as task metadata, task dependencies, and task states (e.g., submitted, ready, completed).

System monitoring agents focus on monitoring node failures and resource availability. Specifically, these agents running on hardware nodes periodically send *heartbeats* to agents at a higher level and the centralized database. If a heartbeat is missed or delayed several times, the system identifies the node as potentially failed or overloaded.

The **centralized monitoring database** consolidates data from task-level monitoring agents, enabling efficient retrieval, analysis, and decision-making. This database simplifies data access for TBPP frameworks, facilitating quick retrieval of monitoring information and supporting effective failure detection and subsequent recovery actions.

The **communication radio** enables communication between the monitoring agents and the centralized database and is responsible for relaying failure alerts, status updates, and metrics between different layers of TBPP frameworks.

V. Resilience Module

The resilience module in WRATH relies on a *failure categorization engine* and a *policy engine* to handle failures when failures are detected.

A. Failure Categorization Engine

The failure categorization engine is responsible for analyzing detected failures and categorizing them based on predefined rules and historical data. Different types of failures, such as hardware faults, software bugs, or resource issues, may require different responses.

The failure categorization engine maintains a **failure taxonomy library** of failure types and associated recovery mechanisms, which can be referenced to determine the best response. Specifically, for failures that occur at the application layer, we summarize the exceptions and errors that may occur in Python. Other failure types are recorded based on the categorization methods introduced in §III, which is done by the failure root cause analyzer.

The **failure root cause analyzer** in this engine performs a comprehensive root cause analysis on the collected monitoring data from multiple layers (application, framework, runtime, and environment) and multiple sources (e.g., logs, exceptions, and resource logs) whenever a failure is detected. This analyzer not only identifies different types of failures but also performs a resource analysis to determine if the failures are due to resource mismatches. Based on the analyzed results from the failure root cause analyzer, WRATH selects an appropriate recovery mechanism using the *resilience policy engine*, detailed in the next subsection.

B. Resilience Policy Engine

For each failure, the resilience policy engine provides an appropriate failure-handling strategy based on the categorizations provided by the failure categorization engine. It contains predefined policies: sets of rules and actions designed to

address different failure scenarios at various layers. Example action list in the policy engine includes *resource denylist*, *immediate termination*, *hierarchical retry*, and *restarting system components*, as shown in Figure 2.

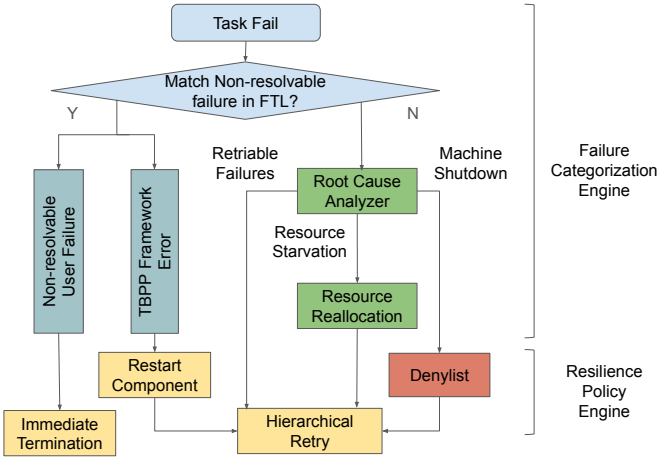


Fig. 2: Flow of the failure categorization engine and resilience policy engine. FTL: Failure Taxonomy Library.

The policy engine maintains a **resource denylist** which records the malfunctioning components in the TBPP framework. The system monitoring agents use the heartbeat mechanism to monitor the status of each component; those that fail to communicate are considered lost and are added to the resource denylist. As in HTCondor [13], resources could be removed from this list if they later resume communication.

When the failure root cause analyzer indicates that a failed task is non-recoverable, the policy engine maps it to an **immediate termination** action, which results in the termination of both the task and the application. This decision is made to prevent further resource consumption since continuing to execute a non-recoverable task can lead to wasted computational resources.

The policy engine employs a **hierarchical retry** mechanism to reschedule failed tasks across the hierarchy of resource pools in TBPP frameworks. It attempts these steps in turn:

1) attempt to retry according to the resource requirements provided by the failure categorization engine to solve certain resource insufficiency problems;
2) retry the task on a different node of the same resource pool in case the task requires specific execution environments;
3) retry where the task has historically succeeded most frequently, ensuring an informed and adaptive retry process;
4) retry on different resource pools (e.g., different Parsl executors [2] or clusters/endpoints in Pegasus [14]).

System failures in TBPP frameworks are often relevant to the failures of the framework components (e.g., the central manager or node managers). In such cases, the policy engine will first attempt to identify the failed components within the system. Once located, it will initiate a **restart of the failed components** to restore functionality. Following this recovery action, the policy engine can then perform a hierarchical retry of any affected tasks, reassigning them to available resources as needed.

C. Overall Failure Handling Flow

The overall flow of the resilience module is illustrated in Figure 2. The process is as follows:

1) The module first examines whether the failures are non-recoverable from hierarchical retries. If failures are deemed non-recoverable:
   - For non-recoverable user failures, WRATH immediately terminates the execution.
   - For system failures within TBPP frameworks, WRATH attempts to restart the failed components and subsequently performs hierarchical retries.
2) If the failures are recoverable, the resilience module utilizes the failure root cause analyzer to identify the specific failures. If these failures are not related to resource issues, WRATH proceeds to execute hierarchical retries directly.
3) In cases where resource failures are identified, the resilience module analyzes the resource profile data to ascertain whether the failure is due to resource starvation or machine shutdown. Based on this analysis, WRATH provides tailored hierarchical retry suggestions to the most appropriate resource pools.

VI. IMPLEMENTATION

We implement a prototype of WRATH and integrate it into Parsl [2], a widely used Python-based TBPP framework. The implementation includes about 3k+ lines of code and is open-source[1]. All the components of WRATH are modularized and can be easily extended to support any appropriate alternatives.

A. Parsl Introduction

Parsl [2] is a Python library for developing parallel and distributed programs. It provides a flexible and scalable runtime for executing scientific workloads and data-intensive applications across various computing resources. Here we introduce the Parsl architecture before describing how we implement WRATH in Parsl.

DataFlowKernel (DFK) is the central manager responsible for managing the flow of tasks and data in the workload. Its functions include dependency resolution, i.e., analyzing the dependencies between tasks and controlling their execution order; task scheduling, i.e., submitting the task to an appropriate executor for execution; and task status tracking.

Executors define the type of computational resources and are responsible for distributing tasks to node managers. They maintain lists of active managers and schedule tasks to them based on their capacities.

Node Managers are responsible for provisioning and allocating resources on an individual node. They ensure that the workers are properly launched and track their status via heartbeat messages. They also maintain task queues and result

[1]https://github.com/ClaudiaCumberbatch/resilient_compute

queues, from which workers pull tasks and to which workers push task results, respectively.

Workers are processes that execute the actual tasks in Parsl. They pull tasks from the task queue to run. Multiple workers can run in parallel on the same node, allowing for efficient utilization of available resources.

B. WRATH *over Parsl*

The overall architecture of WRATH over Parsl is shown in Figure 3. We implement WRATH's task monitoring agents across the hierarchy of Parsl. Specifically, for each node manager in Parsl, the monitoring system launches a *node-level process* that employs Python's `psutil` library [15] to collect the resource information (e.g., CPU and memory utilization) of all the tasks running on that node. This resource profile data, along with detailed task information and the status of each Parsl component, is transmitted via the *radio*, an interface based on the TCP protocol. The radio operates across various locations in the system (e.g., workers, nodes, and the central manager) and sends monitoring data to a modular database. Currently, WRATH supports sending monitoring information to a local database, cloud-hosted database, or a cloud-hosted event fabric for scientific computing (Octopus [16]) to trigger later events.

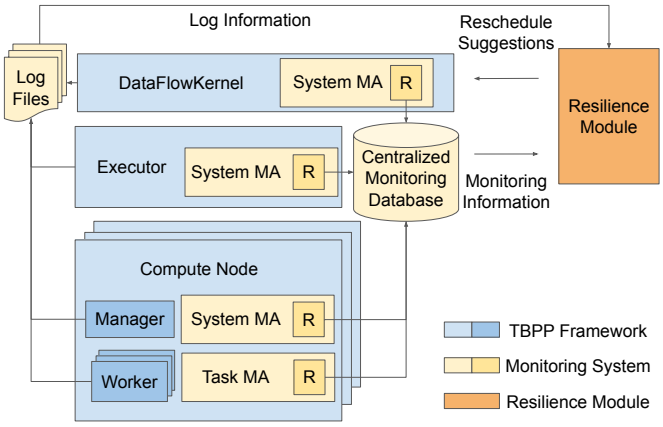


Fig. 3: WRATH system architecture diagram. Components in yellow and orange denote components of WRATH. MA: Monitoring Agent. R: Communication Radio.

We implement the resilience module as a *retry handler* in Parsl. When a task fails, the Parsl DFK automatically invokes the retry handler to determine how to handle the failed task. The failure root cause analyzer in WRATH uses a decision tree to classify errors. For unrecoverable errors, the analyzer combines task and system metrics with heuristics (e.g., error types and retry counts) to recommend fail-fast decisions. To handle Python package-related failures, WRATH dynamically collects package availability on compute nodes (via `pip freeze`) and matches task requirements (using static analysis) to identify suitable nodes. Non-Python library or package-related failures are classified as application-layer errors and flagged as non-recoverable, requiring user intervention.

By adopting a layered architecture for failure characterization, hierarchical monitoring, and a resilience module, the core ideas of WRATH are framework-agnostic and accommodate diverse environments. Users can define custom rules for failure categorization and retry strategies, enabling seamless integration with existing monitoring and resource management systems. As outlined in Section 2, this approach is generalizable to most TBPP frameworks, given their layered architectures and the reliance on retries for failure recovery.

VII. EVALUATION

We evaluate WRATH by applying it to a number of Parsl applications and aim to answer the following questions.

- Can WRATH accurately identify non-resolvable failures, stop retrying, and "fail fast" to minimize wasted time and resources?
- When resolvable errors occur, does WRATH enhance the application's success rate by making appropriate retry decisions?
- Does WRATH introduce significant overhead to application performance?
- How does WRATH perform as the failure rate and scale of the applications increase?

A. Experimental Setup

Testbed: We conduct our evaluation on an HPC cluster[2]. The cluster consists of 815 CPU nodes (each with 2 Xeon Gold 6148 CPUs and 192 GB of memory) and 2 large-memory nodes (each with 8 Xeon Platinum 8160 CPUs and 6 TB of memory).

Workloads: We evaluate WRATH's performance on five applications from the TaPS benchmark suite [17], which provides multiple real-world, DAG-based applications to benchmark the performance of TBPP frameworks: see Table II. We use Parsl as our TBPP framework here. To emulate failures, we created *failure-injection engines* in TaPS that replace selected tasks with failure tasks. For example, we replace the standard Parsl engine with a "Parsl-fail engine". This modified engine allows us to replace a specified fraction of the tasks in the benchmark applications with a failure task. The supported failure types are described in Table III. In the experiments that we describe below, we use this machinery to evaluate the performance of each of our five applications as we vary the types of failure.

Metrics: We evaluate the effectiveness of WRATH using the following metrics.

- Makespan: The total time taken to complete all tasks of an application, including TBPP framework initialization, task execution, retry, clean up, etc.
- Time to failure: The makespan at the point when any task in the application fails without remaining retry attempts, resulting in the application's failure.

[2]https://hpc.sustech.edu.cn/introduction/hardwareresource.html

TABLE II: The benchmark applications.

Application	Description	# Tasks	Configuration
Cholesky	Compute Cholesky decomposition of a randomly generated positive definite matrix	385	Matrix size: 10 000*10 000, Block Size: 1000*1000
Docking	Predict orientation and position of two molecules	160	Initial simulations: 8; Batch size: 8; Rounds: 3
FedLearn	Federated learning	5	Dataset: MNIST; Clients: 8; Batch size: 3, Rounds: 3, Epochs/Round: 3
MapReduce	Count words using a MapReduce strategy	101	Dataset: Randomly generated, Map task count: 100, Generated files: 100
MolDesign	Use ML to identify molecules with largest ionization energies	214	Initial simulations: 4; Batch size: 4; Search count: 16

TABLE III: Failure types that we inject during our WRATH experiments

Layer (Failure Type)	Injected Failures	Description	WRATH Solution
Application Layer (User Failure)	Zero-division Failure	Raise divide by zero error Raise runtime exception	Terminate Terminate
Framework Layer (System Failure)	Worker-killed Dependency	Kill current process Parent task exception leads to child task dependency failure	Reschedule to another worker Act according to the root cause of the dependent item
Runtime Layer (Resource Failure)	ulimit Memory	Open 1M files to simulate ulimit exceeded error Force out of memory error	Hierarchical retry Allocate sufficient memory
Environment Layer (Hardware & Environment Failure)	Import	Simulate import error due to bad environment	Hierarchical retry

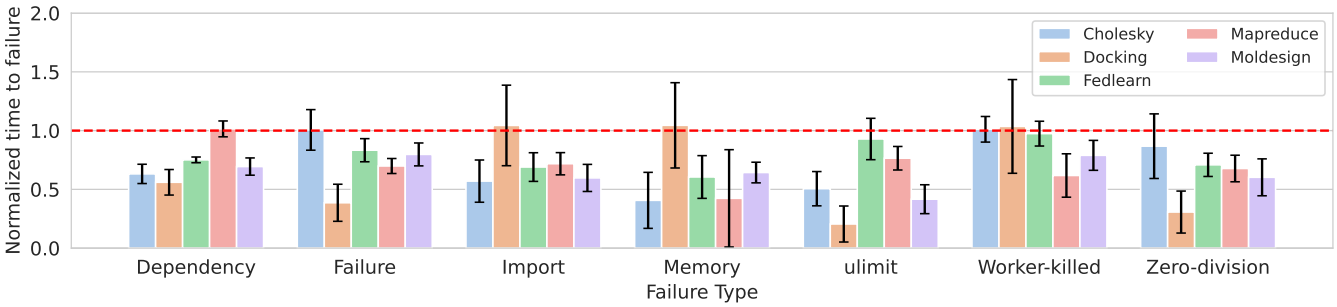


Fig. 4: Normalized time to failure for the applications with different failure types when WRATH is enabled. All results are normalized to those without WRATH. Failure rate = 0.3, Nodes = 32. Error bars represent the standard error of the mean (SEM) across 10 independent runs. All of the trials failed here, but those with WRATH failed fast.

- Overhead ratio: The proportion of time consumed by WRATH to analyze failures and decide retry strategies, relative to the total makespan of the workflow.
- Task success rate: The number of successful tasks divided by the total number of tasks.
- Retry success rate: The number of successfully retried tasks divided by the total number of tasks that were retried.
- Application success rate: The percentage of runs that are successfully completed without any failures during the application execution across multiple runs.

Baseline: We use Parsl and its default retry mechanism as the baseline for comparison. In this default mechanism, tasks are always retried on the same Parsl executor, regardless of the failure type or resource availability.

B. Overall Performance of WRATH

Time to failure: In this experiment, we run the benchmark applications on 32 nodes. To stress-test WRATH, we set the failure rate as 0.3, meaning that 30% of tasks in each application are replaced with failure tasks. Figure 4 shows the time to failure for different applications across various failure types with WRATH enabled. The results show that applications tend to fail more quickly with WRATH compared to without it. For most application and failure type combinations, WRATH reduces the time to failure by 20%–50%. This is because WRATH identifies the root cause of failures and makes more informed retry decisions, allowing tasks that are destined to fail to do so more rapidly.

However, the error bars indicate significant variability in our experimental results. This is because failures can be injected at any point within the application's DAG, and different

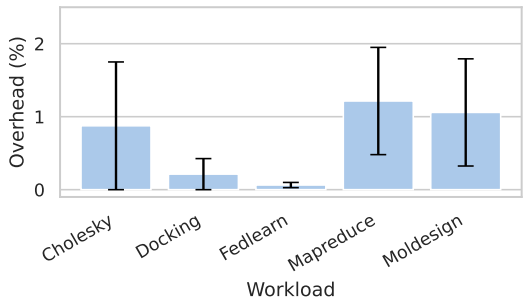


Fig. 5: Overhead ratio of WRATH on successful runs of each application with a pre-set failure rate of 0.1 on 32 nodes.

failure points can lead to substantial differences in application behavior. Take a MapReduce application as an example. a failure can occur during either the Map stage or the Reduce stage. If a failure occurs in the Map stage, the Reduce stage is not executed due to unmet dependencies. In contrast, a failure in the Reduce stage occurs after substantial time has already been spent executing tasks in the Map stage, resulting in a longer time to failure. For applications with more complex DAG structures, the variability between runs becomes even more pronounced.

Overhead: Figure 5 illustrates the overhead ratio for successful runs. In all experiments, the overhead ratio is less than 2%, and in most cases, less than 1%.

C. WRATH's Performance for Resolvable Failures

In this experiment, we aim to show how WRATH performs hierarchical retries for the MapReduce application when dealing with two types of resolvable failures, i.e., memory-insufficient failures and import failures. Note that similar results can be obtained for all the other applications, so we only show the results of MapReduce due to space limits. The settings for the two failure types are as follows:

- **Memory failure**: Each task requires 200 GB of memory and runs on a single node. We configure two executors in Parsl: one with nodes that have 192 GB of memory and another with nodes that have 6 TB of memory.
- **Import failure**: Each task requires a specific software package. We configure two executors in Parsl: one that has the required package installed and one that does not.

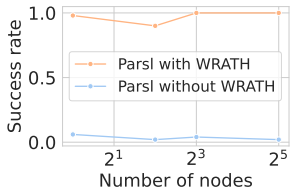
Table IV shows that WRATH significantly improves both the task success rate and the retry success rate. This is because the tasks can only succeed when they are allocated to (or retried on) the appropriate executor—either the one with sufficient memory or the one with the necessary package. Without WRATH, tasks are repeatedly retried on the same executor, meaning that if a task fails due to a memory or import failure, it will continuously fail. Thus, it's unsurprising that both the task success rate and retry success rate are low in the absence of WRATH. With WRATH, tasks are retried across different resource pools based on the success rate and resource availability of each executor, leading to a higher success rate during execution.

TABLE IV: Task success rate and retry success rate of MapReduce. SR: Success Rate.

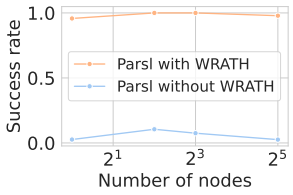
Configuration	Failure Type	Retry SR	Task SR
Parsl with WRATH	import	0.53	0.43
	memory	0.75	0.47
Parsl w/o WRATH	import	0.22	0.00
	memory	0.24	0.00

D. Scalability

In this experiment, we evaluate how WRATH performs when scaling the number of nodes. We inject either import failures or memory failures into the MapReduce application, using the same settings as those in §VII-C. We increase the number of nodes that either lack sufficient memory or do not have the required package. As shown in Figure 6, WRATH consistently maintains an application success rate exceeding 90%, regardless of the number of nodes with insufficient memory or missing packages. In contrast, without WRATH, tasks continuously fail. This enhancement is due to WRATH's ability to leverage hierarchical retries, enabling it to effectively identify and allocate tasks to the appropriate resources for successful execution.



(a) Results for import failures. X axis is the number of nodes without the required package. The number of nodes with the required package is fixed to one.



(b) Results for memory failures. X axis is the number of nodes with insufficient memory. The number of nodes with sufficient memory is fixed to one.

Fig. 6: Application success rate of MapReduce when being injected with different types of failures.

Figure 7 shows that the overhead ratio remains relatively constant as the number of nodes increases for both types of failures. The primary source of overhead is resource log analysis, as WRATH needs to process more logs when the number of nodes grows. This demonstrates that WRATH's mechanisms for detecting failures and reallocating tasks are both efficient and scalable, with most of the overhead attributed to log processing rather than the failure handling or task redistribution itself.

E. Varying the Failure Rate

In this experiment, we vary the failure rate within the range of $[0.1, 0.3]$ to evaluate how WRATH performs. We use the Cholesky application, randomly replacing tasks with a pre-set failure rate for memory-intensive tasks that require 200 GB of memory. All other settings remain consistent with those in

§VII-C. Figure 8 shows that WRATH maintains a high task success rate due to the hierarchical retry mechanism, which monitors the resource usage of failed tasks and schedules them to the appropriate resources for successful execution. In contrast, the task success rate without WRATH continuously decreases as the failure rate rises, demonstrating the effectiveness of WRATH in handling increasing failure rates.

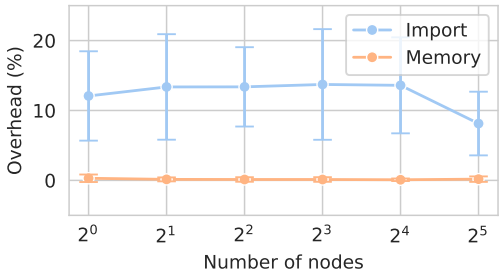


Fig. 7: Overhead ratio of WRATH with a varying number of nodes for the MapReduce application.

VIII. Related Work

A. Failure Categorization

Failures in HPC systems have been widely studied at various granularities. Schroeder et al. analyzed a decade of failure data collected from Los Alamos National Laboratory (LANL) clusters and categorized failures into hardware, software, environmental, network, and human error categories while excluding user application issues [12], [18], [19]. In contrast, we consider user errors to be a significant failure source, and our categorization methodology is more refined, targeting the TBPP stack and decomposing the software layer into Framework and Runtime layers.

Di Martino et al. [11] provided a similar failure categorization for Blue Waters, a Cray supercomputer, highlighting how different failures impacted workload executions, from interrupting with failover to multiple node failures. They found that only 2% of missing compute node heartbeats were due to actual node problems; in most cases, heartbeats resumed without operator intervention. Other researchers [20] have also noted a lack of clear evidence linking health faults to node

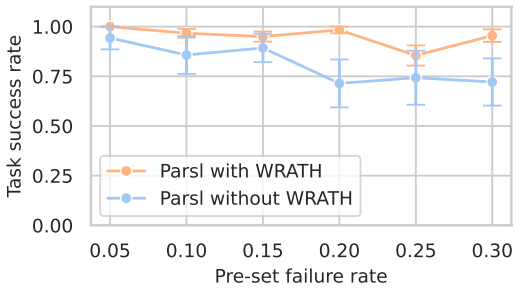


Fig. 8: Task success rate of Cholesky when being injected different rates of memory failure. The number of small memory nodes is 16 and the number of large memory nodes is 1. Error bars represent the standard error of the mean (SEM) across ten independent runs.

faults. WRATH's resilience module also treats service heartbeat as a key indicator of errors, but we avoid relying solely on this signal to inform our resilience strategies.

Several papers [21], [22] consider more refined failure categories that align with the nature of their data sources, such as job scheduling logs and Reliability, Availability, and Serviceability (RAS) logs. They observe that many failures (31–99%) are due to user and application behavior, such as coding bugs, incorrect configurations, operational errors, and memory exhaustion. Our resilient module mitigates such user failures through robust error matching and automated recovery strategies. For recoverable failures, such as missing dependencies, the module informs the scheduler to initiate a hierarchical retry. In the case of syntax errors, it terminates the execution without triggering unnecessary retries.

B. Failure Detection

Detecting failures across multiple layers in HPC and TBPP systems remains a challenge. Wintermute offers an online Operational Data Analytics (ODA) framework to monitor system metrics such as CPU cycles and power usage [23]. WRATH monitors more than numerical readings, leveraging system logs at multiple levels (application, workload, runtime) to identify failures. Furthermore, WRATH incorporates resilient strategies that allow for best-effort recovery without operator intervention.

Huang et al. explored minimally invasive fault detection in MPI applications using side-band network and independent hardware coress [24]. While similar in goal, WRATH's cloud-based event fabric (Octopus) can be configured to be used for monitoring, enhancing reliability by reducing dependence on local hardware.

Other monitoring systems focus on detecting performance degradation [25] , using time series data of resource usage and numerical metrics such as CPU, memory, I/O, and network utilization, alongside with systems and hardware error logs. These systems apply statistical and machine learning methods to compare and contrast data from normal and abnormal runs to identify anomaly signatures. However, these approaches are not directly applicable to our lightweight resilient module, which neither relies on historical data nor requires significant computational power for model training or fine-tuning. Instead, it focuses on addressing categorized failures in the TBPP workload by identifying recoverable failures and performing hierarchical retries where feasible.

C. Failure Handling

Failure handling techniques in distributed systems and HPC can be categorized as reactive [8], proactive [26], or resilient [27]. Reactive methods employ techniques like replication, checkpoints, and retry to mitigate the impact of failures, particularly for long-running tasks, but repeatedly resubmitting tasks without addressing the underlying cause of failure will lead to a huge waste of resources. Proactive methods monitor a system and make predictions to maximize availability, assuming accurate fault predictions. While they aim to prevent

failures through early detection, their effectiveness hinges on the accuracy of fault predictions. Resilience methods leverage machine learning or adaptive learning models to recover from faults quickly by continuously interacting with the environment, but will become inaccurate when encountering a rapidly changing environment. Apart from these, Zhang et al. [28] proposed Trua, which handles failures by employing a historical failure data-based task replication strategy and using anomaly detection to filter out unusual failures.

We focus on reactive strategies such as resource reallocation and hierarchical retries. Traditional methods [29], [30] often conduct checkpointing and replication before retry mechanisms, while our approach explores work placement as an alternative mitigation strategy for atomic tasks. Implementing this approach requires extensive coordination across TBPP layers and the monitoring system, making our work a unique contribution to failure handling in HPC. Kola et al. [31] discussed silent failures in distributed systems—failures that either produce incorrect results without any error status or cause processes to hang. In contrast, we focus on failures that generate exceptions across various layers of the TBPP framework.

IX. Conclusion

We have proposed WRATH, an approach for detecting and handling failures in distributed TBPP. We surveyed common failure-handling mechanisms, including reactive, proactive, and resilient methods, and identified their limitations in dynamic and large-scale parallel systems. To improve TBPP robustness, we developed WRATH with a scalable monitoring system and an intelligent resilient module. Together these components detect, report, and reschedule failed tasks, leading to a reduction in makespan and better resource utilization. Our evaluation results demonstrate that WRATH is effective in enhancing TBPP frameworks by offering automatic failure recovery, improving task execution efficiency, and minimizing performance overhead. However, WRATH currently has limitations in supporting compiled languages. These limitations stem from fundamental differences in error handling between interpreted and compiled languages. In future work, we plan to extend WRATH to support compiled languages by developing language-specific recovery mechanisms and exploring cross-language interoperability for heterogeneous environments. Additionally, we will investigate its integration with other distributed computing frameworks and scalability in large-scale systems.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China Grant No. 62202216, the Guangdong Basic and Applied Basic Research Foundation Grant No. 2023A1515010244, the Shenzhen Science and Technology Program Grant 20231121101752002, NSF awards 2209919 and 2004894, and the Diaspora project funded by the U.S. Department of Energy under Contract DE-AC02-06CH11357. This work was also supported by Center for Computational Science and Engineering at Southern University of Science and Technology.

References

[1] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *SciPy*, 2015, pp. 126–132.

[2] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in Python," in *28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.

[3] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 561–577.

[4] "DAGman: The Directed Acyclic Graph Manager," http://www.cs.wisc.edu/condor/dagman. Accessed Oct 2024.

[5] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, 2017.

[6] "Apache Airflow," https://airflow.apache.org. Accessed Oct 2024.

[7] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, 2018.

[8] A. Iamnitchi and I. Foster, "A problem-specific fault-tolerance mechanism for asynchronous, distributed systems," in *International Conference on Parallel Processing*. IEEE, 2000, pp. 4–13.

[9] M. Turilli, M. Santcroos, and S. Jha, "A comprehensive perspective on pilot-job systems," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–32, 2018.

[10] "Molecular design in Parsl," https://github.com/ExaWorks/molecular-design-parsl-demo. Accessed October 2024.

[11] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 610–621.

[12] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2009.

[13] "HTCondor," https://htcondor.org/. Accessed October 2024.

[14] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[15] "psutil: Python system and process utilities," https://psutil.readthedocs.io/en/latest/. Accessed October 2024.

[16] H. Pan, R. Chard, S. Zhou, A. Kamatar, R. Vescovi, V. Hayot-Sasson, A. Bauer, M. Gonthier, K. Chard, and I. Foster, "Octopus: Experiences with a hybrid event-driven architecture for distributed scientific computing," in *IEEE/ACM 14th Workshop on Fault Tolerance for HPC at eXtreme Scale*. IEEE, 2024.

[17] J. G. Pauloski, V. Hayot-Sasson, M. Gonthier, N. Hudson, H. Pan, S. Zhou, I. Foster, and K. Chard, "TaPS: A Performance Evaluation Suite for Task-based Execution Frameworks," in *IEEE 20th International Conference on e-Science (e-Science)*. IEEE, 2024, pp. 1–10.

[18] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007.

[19] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail," in *43rd annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2013, pp. 1–12.

[20] A. Das, F. Mueller, and B. Rountree, "Systemic assessment of node failures in HPC production platforms," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2021, pp. 267–276.

[21] S. Di, H. Guo, E. Pershey, M. Snir, and F. Cappello, "Characterizing and understanding HPC job failures over the 2K-day life of IBM BlueGene/Q system," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2019, pp. 473–484.

[22] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[23] A. Netti, M. Müller, C. Guillen, M. Ott, D. Tafani, G. Ozer, and M. Schulz, "DCDB Wintermute: Enabling online and holistic operational data analytics on HPC systems," in *29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 101–112.

[24] B. Huang, A. G. Schmidt, A. A. Mendon, and R. Sass, "Investigating resilient high performance reconfigurable computing with minimally-invasive system monitoring," in *4th International Workshop on High-performance Reconfigurable Computing Technology and Applications*. IEEE, 2010, pp. 1–8.

[25] S. Jha, J. Brandt, A. Gentile, Z. Kalbarczyk, G. Bauer, J. Enos, M. Showerman, L. Kaplan, B. Bode, A. Greiner, A. Bonnie, M. Mason, R. K. Iyer, , and W. Kramer, "Holistic measurement-driven system assessment," in *IEEE International Conference on Cluster Computing*. IEEE, 2017, pp. 797–800.

[26] M. Melo, J. Araujo, R. Matos, J. Menezes, and P. Maciel, "Comparative analysis of migration-based rejuvenation schedules on cloud availability," in *IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 2013, pp. 4110–4115.

[27] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1521–1526.

[28] Z. Zhang, B. Bockelman, D. Weitzel, X. Zhang, H. Vakilzadian, and D. Swanson, "Trua: Efficient task replication for flexible user-defined availability in scientific grids," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 2020, pp. 360–369.

[29] J. Zhao, Y. Xiang, T. Lan, H. H. Huang, and S. Subramaniam, "Elastic reliability optimization through peer-to-peer checkpointing in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 491–502, 2016.

[30] R. A. Ashraf, S. Hukerikar, and C. Engelmann, "Shrink or substitute: handling process failures in HPC systems using in-situ recovery," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2018, pp. 178–185.

[31] G. Kola, T. Kosar, and M. Livny, "Faults in large distributed systems and what we can do about them," in *11th International Euro-Par Conference*. Springer, 2005, pp. 442–453.