Optimization towards Efficiency and Stateful of dispel4py

Liang Liang*, Heting Zhang†, Guang Yang*, Thomas Heinis*, Rosa Filgueira†
* Imperial College London
† University of St Andrews

ABSTRACT

Scientific workflows bridge scientific challenges with computational resources. While dispel4py, a stream-based workflow system, offers mappings to parallel enactment engines like MPI or Multiprocessing, its optimization primarily focuses on dynamic process-to-task allocation for improved performance. An efficiency gap persists, particularly with the growing emphasis on conserving computing resources. Moreover, the existing dynamic optimization lacks support for stateful applications and grouping operations.

To address these issues, our work introduces a novel hybrid approach for handling stateful operations and groupings within workflows, leveraging a new Redis mapping. We also propose an auto-scaling mechanism integrated into dispel4py's dynamic optimization. Our experiments showcase the effectiveness of auto-scaling optimization, achieving efficiency while upholding performance. In the best case, auto-scaling reduces dispel4py's runtime to 87% compared to the baseline, using only 76% of process resources. Importantly, our optimized stateful dispel4py demonstrates a remarkable speedup, utilizing just 32% of the runtime compared to the contender.

KEYWORDS

scientific workflow, stream-based workflow, workflow optimization, auto-scaling, stateful application, dispel4py

ACM Reference Format:
Liang Liang*, Heting Zhang†, Guang Yang*, Thomas Heinis*, Rosa Filgueira†, * Imperial College London, † University of St Andrews . 2023. Optimization towards Efficiency and Stateful of dispel4py. In *Proceedings of 18th Workshop on Workflows in Support of Large-Scale Science (WORKS 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Powerful computing infrastructures and platforms have evolved to handle the increase in large-scale data and highly computing-intensive applications [14]. However, navigating these computing resources to tackle data-intensive scientific problems can be overwhelming for scientists from various disciplines. The challenge lies not just in choosing the proper infrastructure but also in understanding and managing it. Therefore, workflow communities are promoted as a means to design and propose various workflow systems, effectively concealing intricate technical details, in order to bridge the divide between scientific challenges and computing technologies [3]. Workflow systems automatically handle low-level computing processing, allowing scientists to focus on their research.

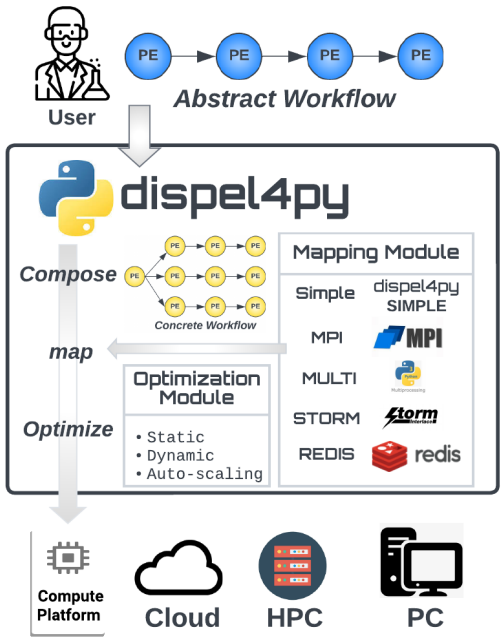


Figure 1: dispel4py overview. From abstract workflow design to automatic concrete workflow generation and execution.

When building workflows within dispel4py, users engage in the design, composition, and interconnection of various processing elements (PEs), which serve as the fundamental computational building blocks of the system [8]. These PEs are linked together by users to form graphs, also known as abstract workflows.

Subsequently, dispel4py smoothly converts the composed abstract workflows into concrete implementations using the chosen mapping (or enactment engine). These concrete implementations are then executed on the specified computing platform, as depicted in Figure 1. This process is underscored by the strategic decoupling of abstract workflows from underlying communication mechanisms, empowering adaptability across diverse computing resources and resonating with dispel4py's intuitive processing methodology that aligns harmoniously with the principles of streaming computing, capturing significant interest from researchers for its adeptness in managing dynamic and time-sensitive data [7, 8].

While dispel4py has substantial merits, performance limitations are evident. A notable concern is its basic and static workload allocation, where processing elements (PEs) are pre-assigned to available *computing resources* in a static deployment of workflows.

WORKS 2023, November 12 2023, Denver, CO

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To address these challenges, previous efforts [12, 13] introduced static optimizations (*naive assignment* and *staging*) and a dynamic optimization (*dynamic scheduling*). *naive assignment* and *staging* strategies optimize resource allocation for static deployment, while *dynamic scheduling* enhances adaptability by dynamically distributing resources among PEs, eliminating operational halts, collectively enhancing scalability and adaptability.

In the current context of increasing significance towards efficiency, encompassing performance and cost-effectiveness, consideration of monetary and energy costs is crucial. Aligning with sustainability and green computing principles, efficient systems lead to significant energy savings and reduced carbon footprint [4, 17]. Auto-scaling emerges as a pertinent solution, dynamically adjusting resources based on real-time demands, ensuring avoidance of underutilization or over-provisioning [20]. Driven by these principles, this work introduces auto-scaling techniques to elevate `dispel4py`'s capabilities, dynamically adjusting resources in response to real-time requirements. This advancement empowers `dispel4py` to enhance its efficiency, conserve energy, and bolster its adaptability to contemporary demands.

Moreover, a rising trend in stateful applications [5] presents a challenge. While `dispel4py`'s static deployment accommodate such applications, compatibility with *dynamic scheduling* becomes complex when processes randomly handle tasks, potentially causing inconsistent task states. To address this, we propose the *hybrid* method, enabling *dynamic scheduling* mappings to proficiently manage both stateless and stateful applications within `dispel4py`. In essence, our contributions seamlessly integrate the *auto-scaling* and *hybrid* techniques, enhancing `dispel4py`'s versatility, and leveraging three real-world workflows for performance evaluation.

The rest of the paper is structured as follows. Section 2 provides the necessary background context. Section 3 delves into the introduced mappings and optimizations. The use cases are presented in Section 4, while Section 5 highlights the potential and effectiveness of our introduced techniques. Lastly, Section 6 concludes with final thoughts.

2 BACKGROUND

2.1 `dispel4py`

`dispel4py` is a stream-based data-intensive workflow written in Python [8]. Key concepts in `dispel4py` include:

- *Processing elements (PEs)* represent computational entities responsible for task processing or data transformation within the workflow graph. In the context of `dispel4py`, various types of PEs are available. Unlike file-based interactions common in task-based workflow systems, data is seamlessly exchanged between connected PEs in a streaming fashion.
- *Instance* denotes an executable copy of a PE. A single PE can have multiple instances depending on the configuration and the number of processes.
- *Connection* transmits data from one PE instance's output port to one or more input ports of another PE instance.
- *Mapping* is the process of 'translating' workflows onto execution systems. This encompasses *Simple* mapping for sequential workflow execution, *MPI*, alongside parallel alternatives like *Multiprocessing*[1], *MPI* [15] and *Storm* [18]. Those mappings eliminate the need for manual workflow modifications.
- *Abstract workflow* consists of several PEs in the form of a directed acyclic graph (DAG), which is designed and defined by the user for solving specific problems.
- *Concrete Workflow*, also referred to as the executable workflow, is a directed acyclic graph that `dispel4py` automatically constructs from the abstract workflow taking into account the selected mapping specified by the user. The concrete workflow is the actual workflow executed by the compute infrastructure.
- *Grouping* governs how processing elements (PEs) communicate during input connections in `dispel4py`. It offers a range of grouping choices available, each with distinct behaviors. For instance, `group-by` operates akin to 'MapReduce,' directing data units with matching values (e.g 'state' in Figure 7) in the specified element to the same PE instance. Employing these grouping strategies bolsters both data distribution and the efficiency of communication within the workflow.
- *Stateless & Stateful*: In the context of `dispel4py`, processing elements (PEs) can exhibit either stateless or stateful behavior. A stateless PE relies solely on its current input for operation, while a stateful PE retains information from previous inputs to influence subsequent outputs.
- *Workload Allocation*: Originally, `dispel4py` supported only static deployment, distributing workloads without considering workflow features. In our previous work [12], we introduced *dynamic scheduling* for adaptive resource allocation to PEs without halting their execution, addressing data-rate and workload variations. Nonetheless, *dynamic scheduling* exclusively manages stateless PEs and lacks support for grouping. Further details regarding *dynamic scheduling* are elaborated in Section 2.2.

Creating `dispel4py` workflows entails user-designed PEs and connections within graphs. PEs are defined with Python classes and linked by specifying inputs and outputs. Upon composition, the (abstract) workflow forms a DAG, where nodes signify PEs and edges depict data flow. Users then choose a mapping to execute the (abstract) workflow on a computing platform, with `dispel4py` automatically generating the suitable concrete workflow.

To illustrate `dispel4py`'s mapping mechanism, we are going to examine the abstract workflow and concrete workflow shown in Figure 1. In this example, the abstract workflow is mapped using *Multiprocessing* mapping across 12 cores, with the first PE exclusively assigned to a single process. Subsequently, each of the remaining PEs is allocated 3 ($\lfloor \frac{12-1}{3} \rfloor$) instances, leaving 2 cores unutilized (as shown in the concrete workflow). This inefficient allocation, leading to two idle cores, prompted our exploration into the *hybrid* and *auto-scaling* optimizations explained in section 3.

2.2 Adaptive Optimizations for `dispel4py`

In [13], we pioneered several optimization for `dispel4py`, leading to the inclusion of an optimization module within the `dispel4py` framework. The module includes two static optimizations and one

[1] https://docs.python.org/3/library/multiprocessing.html

dynamic optimization. The static methods, namely, *naive assignment* and *staging*, are designed to ensure a compact PE allocation to minimize communication costs while maintaining workflow balance. Specifically, *naive assignment* consolidates all interconnected PEs whose communication times surpass their execution times by analyzing execution logs, while *staging* clusters operations that do not require data shuffling based on the abstract workflow.

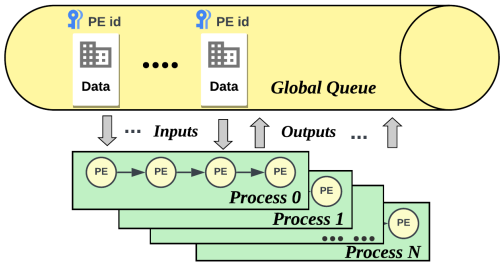


Figure 2: `dispel4py`'s *dynamic scheduling* mapping. Processes retrieve PEs dynamically from 'Global Queue' and return results.

Dynamic optimization, in contrast, targets the execution phase. Unlike static optimizations that assign PEs to distinct processes upfront, the dynamic approach allocates the entire workflow graph to all processes, without predefined tasks. This shift transforms the fixed one-to-one mapping into a dynamic PE-Process mode, where task execution depends on both the PE ID and the process. Processes possess an abstract workflow map, fetching tasks and data from a global queue, referencing their maps for operations, and returning results to the global queue (as depicted in Figure 2). Although this shift does not affect `dispel4py` end-users, it holds vital implications for developers seeking to advance the framework.

Though the optimization module functions independently of the mapping module, the ideal scenario is for the optimization methods to be compatible with various mapping techniques after certain modifications. While static optimizations seamlessly fit with different mappings since they optimize the abstract workflow prior to actual mapping, dynamic optimizations depend on the capabilities and design of the chosen mapping method. For instance, *dynamic scheduling* is ineffective with *Simple* mapping, where tasks are executed sequentially. MPI was primarily designed for SIMD (Single Instruction, Multiple Data) parallel processing, emphasizing efficient and predictable communication patterns among processes. However, this structure is at odds with the adaptive and unpredictable communication inherent to *dynamic scheduling*. Traditional MPI lacks support for a queue-based system crucial for dynamic task assignments. While *dynamic scheduling* involves regular inter-process communication for task balancing, MPI's protocols aim to minimize such communications for efficiency. Consequently, the *MPI* mapping is not suited for the demands of *dynamic scheduling*.

The existing dynamic optimization of `dispel4py` encounters challenges when adapted to novel computational demands, particularly in handling stateful applications. These applications necessitate maintaining consistent states across tasks, a task that `dispel4py`'s random task and data selection approach struggles with. As a solution, this work introduces a hybrid strategy (see Section 3.1.2) to effectively address this limitation and enable seamless handling of stateful applications.

2.3 Redis

Redis [6] is as an open-source, in-memory data structure store, renowned for its efficiency in managing a diverse array of data tasks. Supporting various data structures such as strings, lists, sets, and hashes, Redis has found widespread application in caching, real-time analytics, and messaging systems due to its rapid data access and low latency. Its minimalist yet high-performance design philosophy enables it to handle substantial data volumes seamlessly. Notably, Redis offers advanced functionalities like replication, clustering, and pub/sub messaging, bolstering its adaptability and robustness for modern data-intensive applications.

This work delves into Redis's potential (see Section 3.1) to amplify the performance and resource management of `dispel4py` workflows. Particularly, the integration of Redis Stream[2], a novel data type introduced in Redis 5.0, emerges as a focal point. Redis Stream empowers Redis with dynamic capabilities, enabling streamlined management of message and event streams. This fosters collaborative message consumption by multiple clients from a unified stream. With distinctive message entries and real-time data processing, Redis Stream seamlessly aligns with `dispel4py`'s *dynamic scheduling* objectives.

2.4 Related Work

Within this subsection, we delve into relevant literature, with a particular focus on two key dimensions of optimization: auto-scaling and the management of stateful applications.

2.4.1 Auto-scaling. Auto-scaling mechanisms find application in various domains: (1) Cloud service providers[16] like AWS, Google Cloud, and Azure offer auto-scaling for efficient resource allocation of VM instances while ensuring optimal user experience; (2) Big Data processing[2, 19]: Apache Spark and Apache Flink dynamically allocate resources during runtime; (3) Databases[9]: Amazon Aurora and Azure Cosmos DB leverage auto-scaling to maintain peak performance amidst varying workloads; (4) Workflow systems: Tools like Celery[3] possess auto-scaling capabilities, extendable with external tools. While `dispel4py` and Celery share the domain of workflow systems, their distinct objectives lead to different considerations when integrating auto-scaling. `dispel4py` targets data-flow applications, while Celery caters to task-based ones.

2.4.2 Stateful Applications. Stateful computing in distributed computing remains a popular topic. There are multiple areas that concern statefulness: (1) Traditional relational databases such as MySQL and PostgreSQL support states across transactions. (2) NoSQL databases such as MongoDB [10] support states across nodes, ubiquitous in modern web and mobile application demands. (3) Middleware and Message Brokers: systems such as RabbitMQ maintain stateful information about messages, ensuring reliable and ordered message delivery. (4) Stream processing frameworks: Checkpointing is a

[2]https://redis.io/docs/data-types/streams/
[3]https://docs.celeryq.dev/en/stable/

prevalent strategy for state management. Global snapshot is a state-of-the-art periodic checkpointing solution which captures the holistic state of execution. For instance, Apache Flink has introduced a sophisticated distributed asynchronous snapshot mechanism [5]. This ensures a low-cost state management mechanism; however, it depends on Apache Flink's distinct data flow, which guarantees the ordering. Another check-pointing method for state management is the localized state used by Apache Storm Trident [11]. Both checkpointing methods require ordering, a guarantee that dynamic `dispel4py` cannot provide.

3 MAPPINGS AND OPTIMIZATIONS

In this work, we have directed our efforts towards two primary objectives: leveraging the potential of the Redis framework to integrate the *dynamic scheduling* optimization and introducing an innovative auto-scaling strategy. The upcoming subsections explain in detail the proposed techniques.

3.1 Redis Mappings

The incorporation of Redis into `dispel4py` brings the potential for optimizing data-intensive workflows, as discussed in Section 2.3. Redis, recognized for its effective data management, introduces transformative features through Redis Stream. This real-time sequence maintenance aligns with the `dispel4py` dynamic optimizations.

In our pursuit of integrating `dispel4py` with Redis, we initially introduced the *dynamic Redis* mapping. Yet, to enhance support for groupings and stateful applications within dynamic optimization, we have introduced the *hybrid Redis* mapping. This addition further extends `dispel4py`'s capabilities to accommodate a broader range of optimization scenarios

3.1.1 Dynamic Redis Mapping. The *dynamic Redis* mapping draws inspiration from the original *dynamic scheduling* for *Multiprocessing* mapping. In this adaptation, the multiprocessing queue is replaced with the powerful Redis stream, seamlessly incorporating Redis's inherent features. This alteration is exemplified in Figure 2, where the previous 'Global Queue' utilizing a multiprocessing queue has been replaced by the Redis stream mechanism for this new mapping.

3.1.2 Hybrid Redis Mapping. To address the requirements of stateful applications and dynamic optimization, we introduce the *hybrid Redis* mapping (illustrated in Figure 3). This novel mapping strategy is designed to efficiently handle both stateless and stateful tasks. Key to its operation is the direct mapping of stateful PE instances to dedicated processes, ensuring the maintenance of local states and private task input queues (refereed as 'Private Queues' in Figure 3). Meanwhile, outputs from these stateful PE instances can be routed to different queues based on connections, eliminating the need for continuous state synchronization and enhancing performance in comparison to traditional global state management approaches.

In various stateful applications, tasks can exhibit either a stateful or stateless nature. To maximize the advantages offered by the *dynamic scheduling* mapping strategy, the *hybrid* mapping introduces a mechanism where stateless PE instances are assigned to the available processes that are not dedicated to stateful tasks. This allocation is determined by calculating $N-$number of stateful PE instances, where N represents the total number of available processes. These

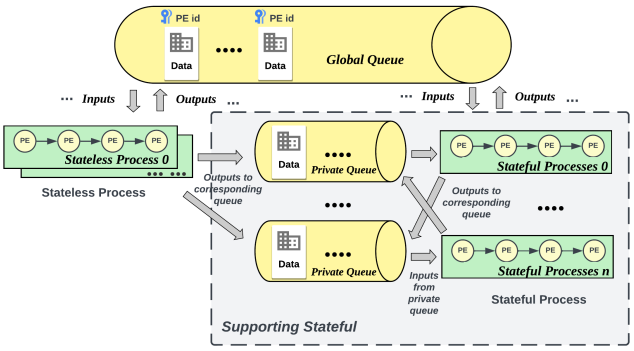


Figure 3: `dispel4py`'s *hybrid Redis* mapping. Processes categorized into stateless and stateful manage corresponding PEs, optimizing workflow execution. Stateful processes maintain dedicated private queues.

stateless processes function similarly to the conventional dynamic methods, acquiring inputs from the 'Global queue' and returning completed tasks to it. However, a subtle distinction arises: these stateless tasks possess the additional capability of depositing their outputs into private queues specifically designated for stateful tasks, enhancing the efficiency of the overall workflow execution.

3.2 Auto-scaling Optimizations

The *auto-scaling* optimization addresses the efficient allocation of computational resources in response to the varying workload. *auto-scaling* makes the system responsible for workload spikes, and reduces resource wastage during low workload periods, thereby improving the efficiency of `dispel4py`. We implement *auto-scaling* optimization for *Multiprocessing* and *dynamic Redis* (see Section 3.1.1) mappings for handling stateless workflow, namely *dynamic auto-scaling Multiprocessing* and *dynamic auto-scaling Redis*.

Within the framework of `dispel4py`, *auto-scaling* extends the capabilities of *dynamic scheduling*, as depicted in Figure 4. This enhancement introduces two processor statuses: active and idle. Active processes are allocated to the abstract workflow and actively participate in task execution. They retrieve PE IDs and associated data from the queue, process them, and return the results, similar to the original *dynamic scheduling* behavior.

However, a notable distinction arises when the queue experiences a reduced workload. In the conventional *dynamic scheduling* framework (introduced in Section 2.2), certain processes continuously monitor the queue, anticipating new tasks. This operation, however, proves to be both resource-intensive and redundant. *auto-scaling* introduces an optimized approach to address this issue. Processes without immediate tasks are transitioned into an idle state, operating in a low-energy standby mode. This efficient mechanism not only curbs energy consumption but also offers the potential for reallocating these idle processes elsewhere. When a surge in workload is detected, *auto-scaling* can swiftly activate these dormant processes. Conversely, during periods of low demand, surplus processes are deactivated and returned to an idle state. This dynamic activation and deactivation process is orchestrated by the

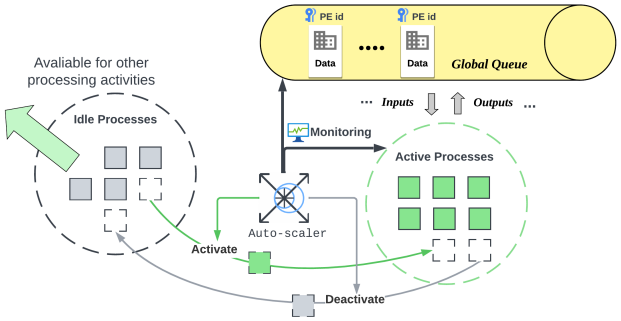


Figure 4: dispel4py's *auto-scaling*. Idle processes conserve energy and can be reactivated when workload surges.

auto-scaler, working in tandem with a monitoring framework and various auto-scaling strategies.

3.2.1 Auto-scaler. Algorithm 1 shows how the *auto-scaler* works. It sets parameters like the maximum pool size and workload threshold upon initialisation; by default, the *active_size* is half of the maximum of total processes (*max_pool_size*). The configurable *workload_threshold* is used in auto-scaling strategies. The `shrink` and `grow` methods decrease and increase the active processes, respectively, and are controlled by the central logic methods `auto_scale`. It implements auto-scaling strategies (introduced in the Section 3.2.2), which monitor the system state, and control the resource adjustments. Tasks are dispatched through the `start` method and finished by the done methods; those two methods are also responsible for updating the active count denoted as *active_count* used to avoid over-use of processes. The entry method `process` continually evaluates and scales resources before activating processes by calling `start`. Overall, the *auto-scaler* can schedule the resources dynamically, ensuring efficiency and responsiveness.

3.2.2 Auto-scaling and Monitoring Strategy. Auto-scaling is not just about dynamically reallocating resources; striking the right balance between performance and efficiency is also the crux of chosen auto-scaling strategies. These strategies have two important decisions: 'when to scale' and 'how to scale'. The former is about metrics from the monitoring framework, while the latter determines the magnitude of scaling. For instance, a question might be whether we should rapidly scale up when we observe a task burst. In this work, we adopt a simple incremental approach: incrementing the active size by 1 or -1. Given *dynamic auto-scaling Multiprocessing* and *dynamic auto-scaling Redis* have different monitoring frameworks, we use a different strategy for each:

- *dynamic auto-scaling Multiprocessing*: This approach employs queue size to gauge the current workload. When the queue size increases compared to the previous state, indicating higher task volume, additional processes are activated. Conversely, processes are deactivated during reduced workload, while a minimum threshold prevents unnecessary scaling during low demand.
- *dynamic auto-scaling Redis*: Here we utilize Redis's consumer group's average idle time as a metric. Unlike queue size, idle

Algorithm 1 Auto_scaler for Dynamic Optimization

```
1:  Class Auto_scaler:
2:  Parameters : max_pool_size, pool, threshold, queue, active_size, active_count
3:
4:  procedure CONSTRUCTOR(Parameters)
5:      Initialize member parameters
6:      By default active_size ← max_pool_size/2
7:  end procedure
8:
9:  procedure SHRINK(size_to_shrink)
10:     Decrease active_size by size_to_shrink (with a minimum of 1)
11: end procedure
12:
13: procedure GROW(size_to_grow)
14:     Increase active_size by size_to_grow (with a maximum of max_pool_size)
15: end procedure
16:
17: procedure IS_TERMINIATED
18:     Return BOOL depending on Termination Methods
19: end procedure
20:
21: procedure AUTO_SCALE
22:     curr_states ← Monitor
23:     if curr_states > threshold then
24:         GROW(1)
25:     else
26:         SHRINK(1)
27:     end if
28: end procedure
29:
30: procedure START(func, args)
31:     while active_count >= active_size do
32:         Wait
33:     end while
34:     Increment active_count
35:     return Pool.apply_async(func, args, callback=DONE())
36: end procedure
37:
38: procedure DONE(result)
39:     Decrement active_count
40: end procedure
41:
42: procedure PROCESS(graph)
43:     while True do
44:         AUTO_SCALE
45:         if IS_TERMINIATED( )then
46:             Get results from results
47:             Return
48:         else
49:             cp_graph ← DeepCopy(graph)
50:             Initialize worker's args with queue and cp_graph
51:             START(worker.process, args)
52:         end if
53:     end while
54: end procedure
55: end Class
```

time directly reflects process states. We employ a threshold for the average idle time of active processes. If a process's idle time exceeds the time needed for reactivation and redeployment, it is logically deactivated. The reactivation time is influenced by computational resources and the specific workflow, requiring proper configuration for practical use

The experimental results in Section 5 show that these preliminary strategies are effective. While they currently serve our purpose, we will dive deeper to refine and optimize them in future work.

3.2.3 Termination Strategy. As mentioned earlier, with the shift to dynamic optimization, how tasks are executed and terminated significantly changes. Instead of the static pre-assignment of PEs to specific processes, dynamic optimization assigns the entire workflow graph to all processes and takes a task from the queue without order. Such changes results in the failure of traditional static termination methods. In the static context, the *"poison pills"* termination method was employed, where the source PE would signal the end of its input to all subsequent instances. However, in the dynamic setting, the task processing order is not reserved; it is based on availability rather than any specific order in the abstract workflow. This makes the *"poison pills"* method ineffective, as it can unexpectedly halt processes and leave tasks unfinished in the queue. To solve this, the native dynamic approach relies on checking the emptiness of the global queue for termination. While this method is generally effective, it is not foolproof and could lead to unexpected exits in some extreme cases. Additionally, constant checks from all processes on the queue's status could be inefficient.

We use a retry mechanism combined with *"poison pills"* to mitigate these challenges. If the queue appears empty, processes will wait for a configurable threshold duration and retry a specified number of times before deciding on termination. Once a process determines to stop based on these parameters, it broadcasts *"poison pills"* to other processes, speeding up their termination check, thereby reducing overall waiting time.

4 USE CASES

In this section, we introduce three real scientific `dispel4py` workflows, including two stateless workflows and one stateful workflow. These workflows will be used in the experiment to evaluate our proposed mappings and optimizations.

4.1 Internal Extinction of Galaxies

The Internal Extinction of Galaxies workflow has been implemented to calculate the extinction metric within galaxies, which is a significant property in astrophysics. This property reflects the dust extinction of the internal galaxies and is used for measuring optical luminosity. The workflow contains four PEs, as shown in Figure 5. The first PE, `read RaDec`, reads the coordinator data for galaxies in an input file. Then, these data are used in `getVO Table` to download the VOTables. These VOTables in `filter Columns` are filtered by specified columns used in the internal extinction computation. The last PE (`internal Extinction`) will perform the computation. It is important to note that all PEs here are stateless.

To introduce variability in the workload, we adjusted the `read RaDec` PE. For a standard workload (denoted as 1X), it reads data for 100 galaxies. This reading scales to 300 galaxies for 3X, 500 for 5X, and 1000 for 10X. In addition to varying the stream length, we also varied the PE's workload. By using a random sleep time sampled from a beta(2,5) distribution, we added delays ranging from 0 to 1 second within the `getVO Table` PE and `filter Columns` PE. This modification is labeled as "heavy". Therefore, the experiment based on Internal Extinction of Galaxies now has both standard and heavy workloads varying from 1X to 10X.

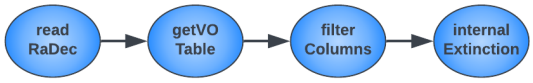


Figure 5: Internal Extinction of Galaxies workflow.

4.2 Seismic Cross-Correlation

The Seismic Cross-Correlation workflow is engineered to monitor and analyze the vast geological waveform data gathered from FDSN[4] stations. Its primary goal is to evaluate and predict the risk and likelihood of volcanic eruptions and earthquakes in real-time. The original workflow consists of two phases (in Figure 6): the initial stage involves pre-processing data collected from stations, and the second phase deals with reading the pre-processed data and executing the cross-correlation computations.

Notably, the second phase has a *grouping* mechanism. Given that the *auto-scaling* cannot handle stateful applications, we want to focus on the first phase of our experiments (all PEs in the first phase are stateless). For testing the *hybrid*, we selected another representative workflow. The first phase comprises nine interconnected PEs: the initial PE reads the data, the intermediate PEs process the raw data, and the final PE writes the data to disk. There are more imbalanced workloads among PEs; for example, the intermediate PEs only do calculations in main memory, but the last PE writes data into the disk, which involves IO operations. A detailed description of the setup can be found in [12, 13].

4.3 Sentiment Analyses for News Articles

This workflow analyses newspaper articles data by implementing a sentiment analysis of the news[5]. This workflow applies two distinct sentiment analyses on articles, subsequently aggregating these sentiment scores based on the published location. Such workflow complicates the graph by having various types of *grouping* compared with the first two workflows. The news articles employed the source data from public *News Articles* datasets on Kaggle[6].

The workflow shown in Figure 7 unfolds as follows. The initial PE (`read Articles`) sequentially reads and parses articles from input files. Each parsed article then undergoes dual processing by two downstream PEs. The `sentiment AFINN` PE computes sentiment scores utilizing the AFINN lexicon[7], while concurrently, `tokenize WD` and `sentiment SWN3` PEs tokenize the articles and derive sentiment using the SWN3 lexicon [1]. Post-processing, the data from both pathways convene within their respective `find`

[4] https://www.fdsn.org
[5] https://github.com/NoPuzzle/dispel4py_autoscaling/tree/main/dispel4py/examples/article_sentiment_analysis
[6] https://www.kaggle.com/datasets/asad1m9a9h6mood/news-articles
[7] https://github.com/fnielsen/afinn

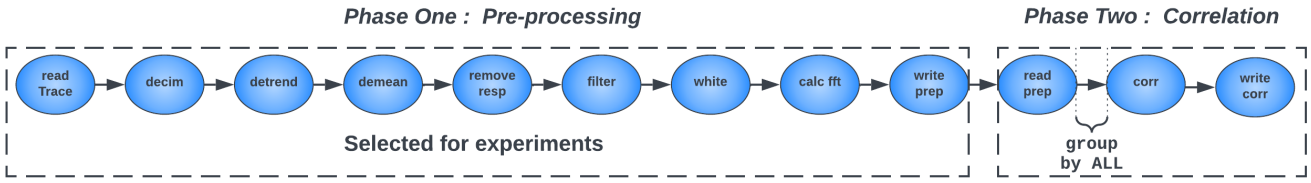


Figure 6: Seismic Cross-Correlation workflow.

`State` - `happy State` - `top 3 happiest` sequence. These three PEs identify, group, and display the top three happiest locations with their scores.

Stateful PEs play a crucial role in this workflow. For instance, the `happy State` PE is strategically distributed across four instances grouped by their 'state' (`group-by`), while the `top 3 happiest` PE is confined to one instance under the `global` grouping [8]. In contrast, other PEs lack these constraints and are classified as stateless. By blending stateless and stateful PEs, this workflow stands as an ideal testbed to explore the behavior of enhanced dynamic deployment within the realm of a real stateful application.

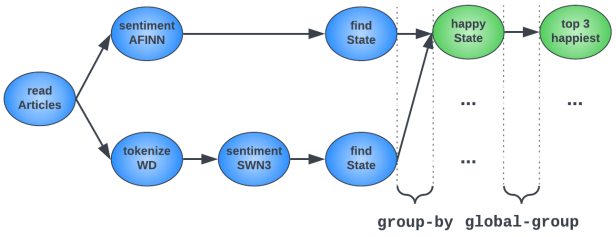


Figure 7: Sentiment Analyses for News Articles workflow.

5 EVALUATIONS

In this section, we conduct a comparative experimental evaluation of the workflows outlined in Section 4. Our primary focus lies in assessing the performance and efficiency of the mappings and optimization techniques proposed in Section 3. We introduce abbreviations and provide key insights into the evaluated techniques:

- *multi*: This represents the native *Multiprocessing* mapping [7]. this approach statically assigns PE instances to processes as detailed in the background Section 2.1. Benefiting from its inherent state maintenance capabilities, *multi* can effectively manage both stateful and stateless applications, establishing itself as an appropriate baseline for all experimentation.
- *dyn_multi*: This denotes the *dynamic Multiprocessing* mapping [12] introduced in the background Section 2.2 . It enables processes to dynamically share the workload on-the-fly. Being built upon the *Multiprocessing* mapping, *dyn_multi* serves as a baseline for comparing the *auto-scaling* optimization based on the same mapping.

[8]This grouping enforces all instances of the previous PE (characterized by the 'happy state') are directed towards a singular instance of the same PE (representing the 'top 3 happiest')

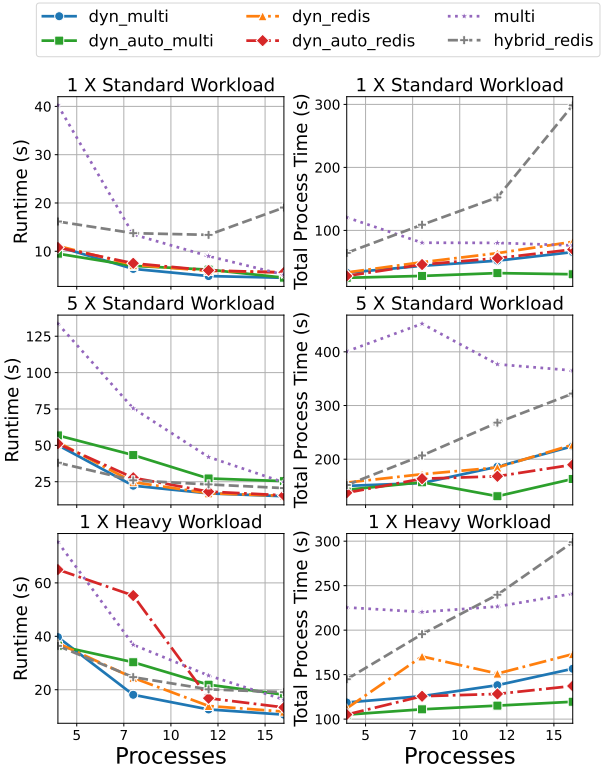


Figure 8: Workload Performance for Internal Extinction of Galaxies using the *server* with up to 16 processes.

- *dyn_auto_multi*: This signifies the new *dynamic auto-scaling Multiprocessing* mapping presented in Section 3.2.2. It will be compared primarily with *dyn_multi*, aiming to evaluate the efficiency gains achieved by incorporating *auto-scaling*.
- *dyn_redis*: This refers to the new *dynamic Redis* mapping explained in Section 3.1.1. The performance of *dyn_redis* is horizontally compared with *dyn_multi*, as both employ the *dynamic scheduling* optimization.
- *dyn_auto_redis*: This stands for the new *dynamic auto-scaling Redis* mapping introduced in Section 3.2.2. *dyn_auto_redis* is mainly compared with *dyn_redis* to evaluate *auto-scaling* in a vertical comparison, given their shared mapping foundation. Additionally, it is compared with *dyn_auto_multi* for a horizontal assessment over different mappings.

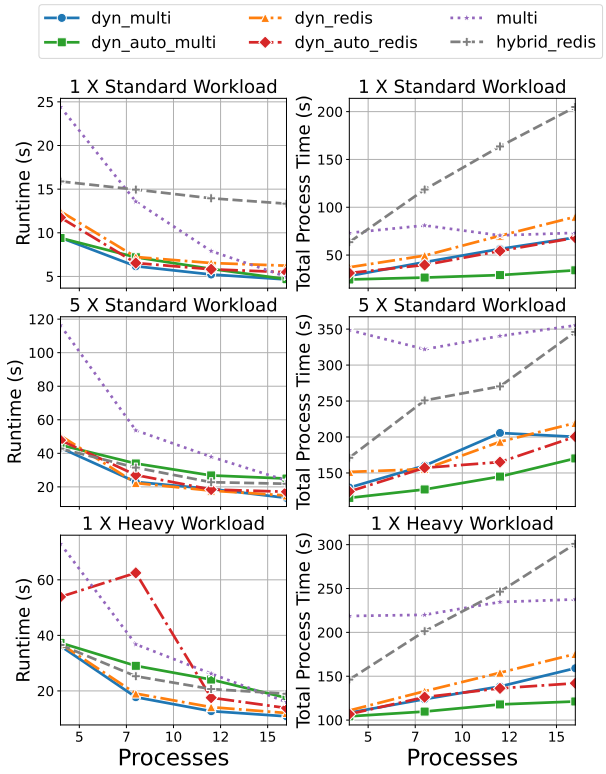


Figure 9: Workload Performance for Internal Extinction of Galaxies using the *cloud* with up to 16 processes.

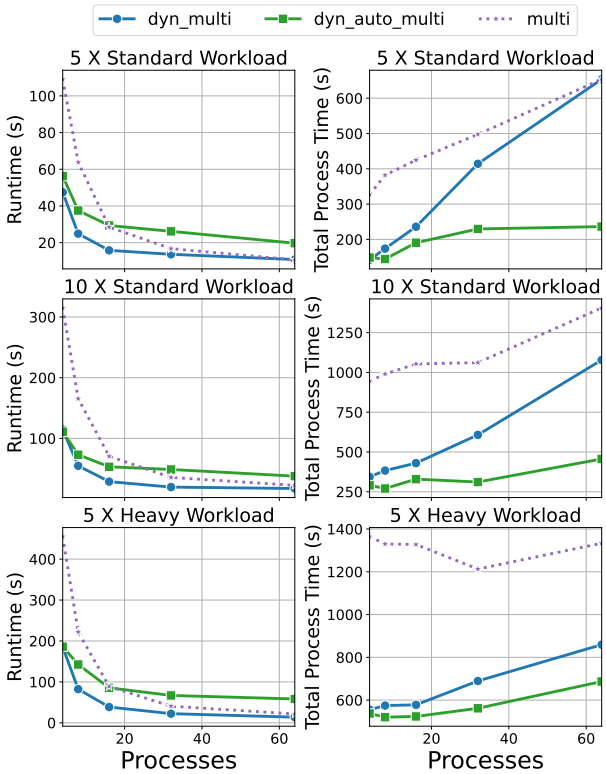


Figure 10: Workload Performance for Internal Extinction of Galaxies using the *HPC* with up to 64 processes.

- *hybrid_redis*: This denotes the new *hybrid Redis* mapping explained in Section 3.1.2, exclusively designed to support stateful applications. The performance of *hybrid_redis* will be comparatively evaluated with *multi*.

5.1 Experiment Setup

5.1.1 Platform. We have conducted our experiments on multiple infrastructures/platforms, each with its distinct configurations:

- *server*: This is a virtual server for research groups supported by Department of Computing, Imperial College London [9]. It consists of 16-core with Intel E5-2690@2.60GHz processor paired with 64GB of RAM and runs on Ubuntu 14.04. We denote it as *server* for short. It runs all three workflows on various processes: 4, 8, 12, and 16.
- *cloud*: This server is provided by Google Cloud Platform [10]. It has 4 Intel(R) Xeon(R) CPU @ 2.20GHz (8vCPUs), and 16GB of RAM running Ubuntu 20.04. For our experiments, we refer this server to *cloud*, and its configuration for running the experiment is identical to *server*.
- *HPC*: The HPC cluster we used is provided by HPC at Imperial [11]. The HPC servers provide multi-job classes, and we use the *short* class in which the CPU and OS mode are Intel E5-2680 v3 @ 2.50GHz and Centos 8, respectively. We request up to 64 CPUs and 64 GB memory. Since Redis cannot be deployed on the HPC, no mapping is based on Redis running on HPC. We employed 4, 8, 16, 32, and 64 CPUs for other experiments.

5.1.2 Metrics. We use two main metrics for evaluation: runtime (*runtime*) and total process time (*process time*). *runtime* represents the real-world execution time, while *process time* accounts for all active process durations, reflecting overall efficiency. Typically, fewer processes lead to shorter *process time* due to reduced synchronization overhead, but may extend *runtime*. Our results are presented as *runtime* and *process time* pairs for a comprehensive view.

Moreover, we calculate *runtime* and *process time* ratios between methods to provide intuitive insights. For example, a *runtime* ratio below 1 (shown in Table 1) between *dyn_auto_multi* and *dyn_multi* implies that *dyn_auto_multi* completes tasks faster. Similarly, a *process time* ratio below 1 suggests *dyn_auto_multi* is more efficient than *dyn_multi*. To maintain consistency, we only include our proposed optimizations (*auto-scaling* or *hybrid_redis*) in the numerator. If both ratios are below 1, the proposed approach excels in both performance and efficiency, although practical trade-offs may affect ideal outcomes.

[9] https://www.imperial.ac.uk/admin-services/ict/self-service/research-support/private-cloud/
[10] https://cloud.google.com/
[11] https://www.imperial.ac.uk/computational-methods/hpc/

5.2 Ev. Internal Extinction of Galaxies

Different workloads based on Internal Extinction of Galaxies workflow are used on multiple platforms to provide a comprehensive evaluation. On both *server* and *cloud*, we use three different workloads: the 1X standard workload (100 galaxies as input), the 5X standard workload (500 galaxies), and the 1X heavy workload (100 galaxies with sleep synthetically added to the PEs). On *HPC*, which offers more cores, we deploy a heavier workload than *server* and *cloud*. *HPC* will be experimented with 5X, 10X standard and 5X heavy workload.

Figure 8 shows the performance metrics in terms of *runtime* (left) and *process time* (right) for all six techniques introduced in Section 5. All techniques show a decreasing trend for *runtime* with an increase in the number of processes, indicating they share good scalability in `dispel4py`. With lower workloads, *auto-scaling* techniques achieve shorter *runtime*, as they dynamically adjust the number of activated processes, thereby lowering unnecessary synchronization costs. However, as the workload increases, particularly in heavy-workload scenarios, *auto-scaling* technique lags slightly behind pure *dynamic scheduling* optimizations. This minor fault could be optimized with a more refined auto-scaling strategy. Apart from this, *auto-scaling* techniques still outperform native *multi* mapping in most cases. Regarding *process time*, as the number of processes increases, the *process time* rises due to the accumulated synchronization overhead from more active processes. Unsurprisingly, both *dyn_auto_multi* and *dyn_auto_redis* increase slightly, with both excelling over their corresponding pure *dynamic scheduling* competitors.

Notably, in the race track of versatile methods (supporting both stateless and stateful), *hybrid_redis* presents strong *runtime* performance, but given we did not equip *auto-scaling* optimization to it, *hybrid_redis* does not achieve the same efficiency, compared with *dyn_auto_redis*. In terms of comparison between *multi* and *Redis*, there is a common pattern trend across various experimental setups: both *runtime* and *process time* metrics for the optimization using the *Redis* mapping are larger than those of the *multi* mapping. The reason can be attributed to the inherent characteristics of the two mappings. Compared with *Redis*, *multi* is designed to be lightweight, offering outstanding performance. However, *Redis* supports more features regarding monitoring, reliable messaging and robust data persistence, which render *Redis* more resource-intensive, thereby affecting its performance.

Performance trends on the *cloud* platform are similar to those observed on the *server*. However, since there are only 8 cores in *cloud*, the performance slightly dips with 12 and 16 processes compared to *server*. Despite this, the overall trends remain consistent across different platforms, indicating the reproducibility and portability of our experiments.

Experiments on *HPC* show how the performance metrics (both *process time* and *runtime*) change with a larger scale in processes. Experiments on *HPC* show how the performance metrics (both *process time* and *runtime*) change with a larger scale in processes. The *runtime* of all three methods based on *multi* mapping show a quick drop when increasing the number of processes to 16, then gradually becoming steady with a slight decrease. However, the *process time* of *dyn_multi* and *dyn_auto_multi* show a linear increase with the increase in the number of processes. In the meanwhile, the increase of *dyn_auto_multi* is at a slight upward slope. This difference strongly supports the effectiveness of *auto-scaling*, especially when a large number of processes are involved.

Table 1: Performance comparison based Internal Extinction of Galaxies workflow between techniques (A and B). The *runtime* ratio is the *runtime* of mapping A over the *runtime* of mapping B. Similarly, the *process time* ratio is the total process time elapsed ratio between A and B. The ratios are prioritized by the metric in the "Prioritized By" column, and the [Mean, Std] shows the average and standard deviation for all *runtime* and *process time* ratios.

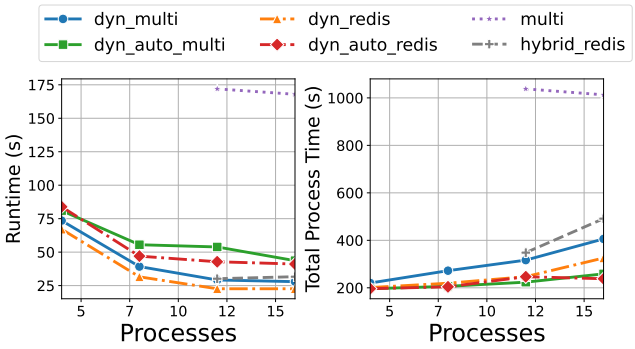
Platform	Comparison between A/B	Prioritized By	Runtime Ratio	Process Time Ratio
server	$\frac{dyn\_auto\_multi}{dyn\_multi}$	*runtime*	0.87	0.76
		process time	1.01	0.46
		[Mean, Std]	[1.39, 0.37]	[0.77, 0.15]
	$\frac{dyn\_auto\_redis}{dyn\_redis}$	*runtime*	0.97	0.83
		process time	2.26	0.74
		[Mean, Std]	[1.21, 0.39]	[0.86, 0.06]
cloud	$\frac{dyn\_auto\_multi}{dyn\_multi}$	*runtime*	1.00	0.87
		process time	1.01	0.50
		[Mean, Std]	[1.36, 0.34]	[0.77, 0.15]
	$\frac{dyn\_auto\_redis}{dyn\_redis}$	*runtime*	0.88	0.76
		process time	0.88	0.76
		[Mean, Std]	[1.26, 0.66]	[0.86, 0.08]
HPC	$\frac{dyn\_auto\_multi}{dyn\_multi}$	*runtime*	0.97	0.85
		process time	1.83	0.36
		[Mean, Std]	[1.95, 0.83]	[0.75, 0.20]

5.2.1 Summary of the evaluation on Internal Extinction of Galaxies with runtime and process time ratios. To provide an intuitive view of the results, we calculate *runtime* and *process time* ratios to intuitively compare *auto-scaling* and *dynamic scheduling* techniques: *dyn_auto_multi* and *dyn_multi*, *dyn_auto_redis* and *dyn_redis*. Table 1 displays these ratios across various platforms, allowing analysis from different perspectives. For example, in the best *runtime* case, *dyn_auto_multi* requires only 87% *runtime* and 76% *process time* of *dyn_multi*. Prioritizing *process time*, the most efficient ratio achieved is 0.46, with a 1.01 *runtime* ratio. Overall, *auto-scaling* techniques demonstrate efficiency by slightly extending *runtime* while reducing *process time*, highlighting their trade-offs and benefits.

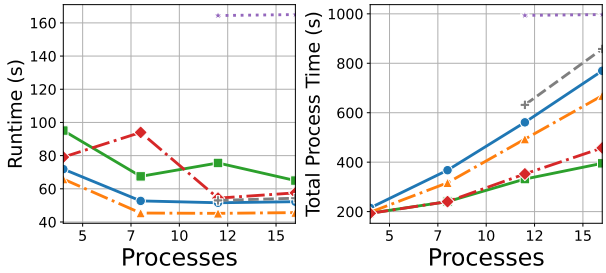
5.3 Ev. Seismic Cross-Correlation

In the Seismic Cross-Correlation workflow, we adopt a consistent workload (50 stations as input) across all platforms. Compared with Internal Extinction of Galaxies, Seismic Cross-Correlation has more number of PEs and characterises a more heterogeneously distributed workload. It is worth pointing out that, unlike other methods, which start with 4 processes, *multi* initiates with 12 processes. Since the workflow contains 9 PEs, given the fixed allocation of *multi*, 9 processes is the minimum requirement. This is a constraint of native static mappings.

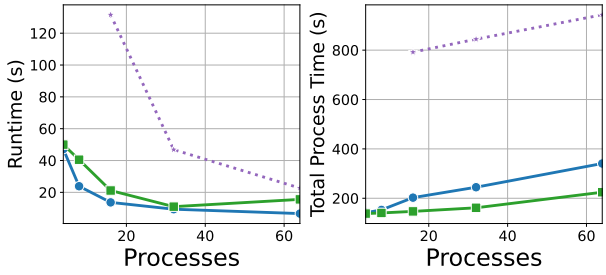
As illustrated in Figure 11a and Figure 11b, *runtime* of all techniques show a downward trend as the number of processes increases. Conversely, *process time* exhibits an increased trend. Notably, the overall performance on *server* is slightly better than *cloud* due to the different capacities of these two platforms. The pattern observed includes different mappings and optimizations, various platforms, and different metrics closely aligned with the experiment



(a) Workload Performance for Seismic Cross-Correlation using the *server* with up to 16 processes.



(b) Workload Performance for Seismic Cross-Correlation using the *cloud* with up to 16 processes.



(c) Workload Performance for Seismic Cross-Correlation using the *HPC* with up to 64 processes.

Figure 11: Experiments on Seismic Cross-Correlation over different platforms.

results from Internal Extinction of Galaxies. Given this consistency, we avoid repeatedly mentioning these patterns. Instead, we will conclude with insights from the findings in the subsequent summary section (Section 5.6).

5.3.1 Summary of the evaluation on Seismic Cross-Correlation with runtime and process time ratios. Statistics from Table 2 reveal that the overall *process time* of this workflow surpasses that of Internal Extinction of Galaxies. Notably, while optimal *runtime* ratios in the previous workflow were generally under 1, they exceed 1 in this case. This indicates that the *auto-scaling* optimization faces challenges with *runtime* in complex workflows. This could stem from the limitations of the naive auto-scaling algorithm in accurately gauging demand for intricate workflows. However, the *process time* ratios, consistently below 1, affirm the efficiency of *auto-scaling* even in complex scenarios.

Table 2: Performance comparison based Seismic Cross-Correlation workflow between techniques (A and B). The *runtime* ratio is the *runtime* of mapping A over the *runtime* of mapping B. Similarly, the *process time* ratio is the total process time elapsed ratio between A and B. The ratios are prioritized by the metric in the "Prioritized By" column, and the [Mean, Std] shows the average and standard deviation for all *runtime* and *process time* ratios.

Platform	Comparison between A/B	Prioritized By	Runtime Ratio	Process Time Ratio
server	$\frac{dyn\_auto\_multi}{dyn\_multi}$	*runtime*	1.10	0.89
		process time	1.56	0.64
		[Mean, Std]	[1.48, 0.31]	[0.75, 0.11]
	$\frac{dyn\_auto\_redis}{dyn\_redis}$	*runtime*	1.25	0.98
		process time	1.82	0.73
		[Mean, Std]	[1.61, 0.30]	[0.91, 0.12]
cloud	$\frac{dyn\_auto\_multi}{dyn\_multi}$	*runtime*	1.18	0.62
		process time	1.46	0.61
		[Mean, Std]	[1.30, 0.13]	[0.69, 0.14]
	$\frac{dyn\_auto\_redis}{dyn\_redis}$	*runtime*	1.05	0.90
		process time	1.50	0.60
		[Mean, Std]	[1.47, 0.40]	[0.73, 0.13]
HPC	$\frac{dyn\_auto\_multi}{dyn\_multi}$	*runtime*	1.06	0.98
		process time	2.34	0.66
		[Mean, Std]	[1.56, 0.51]	[0.79, 0.15]

5.4 Ev. Sentiment Analyses for News Articles

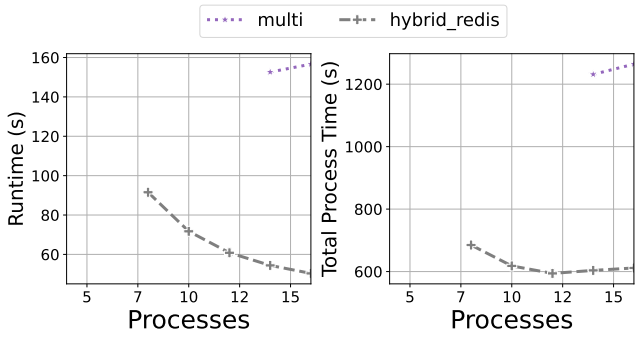
The goal of this workflow evaluation is to assess *hybrid_redis*'s performance in stateful applications compared to the baseline *multi*. For this purpose, the `happy state` and `top 3 happiest` stateful PEs have 4 and 2 instances, respectively. With 1 process reserved for stateless PEs, *hybrid_redis* starts with 8 instances in the experiment. In contrast, *multi* demands a minimum of 14 processes due to its one-to-one instance-to-process mapping. As the upper process limit is 16, we use finer increments of 8, 10, 12, 14, and 16 processes.

The results on the *server*, as shown in Figure 12a, demonstrates that *hybrid_redis* significantly outperforms the *multi* in terms of *runtime*. Additionally, *hybrid_redis* exhibits a speed-up as the number of processes increases. This can be attributed to the fact that as more processes are added, the stateless workload can be shared by more processes, thereby reducing the overall execution time. Unexpectedly, the trend of *process time* is similar to *runtime*. Given that *auto-scaling* is not applied into *hybrid_redis*, this efficiency gain likely comes from the more efficient processing of stateless tasks, which reduces the idle time for stateful processing awaiting outputs from stateless PEs. Thus, even with the additional synchronization overhead caused by the increasing number of processes, this reduction in idle time still results in a net efficiency gain.

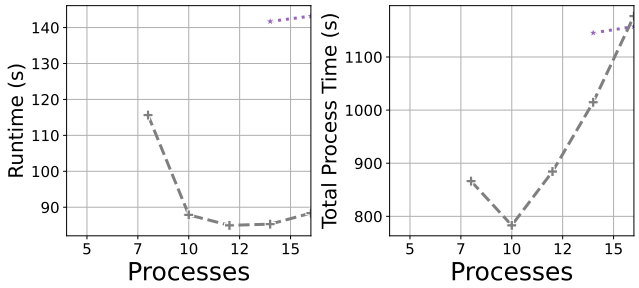
For the results on *cloud* which is limited to 8 cores, the drawbacks of over-allocating processing become evident. While *hybrid_redis* shows a downtrend in the *runtime*, its *process time* tells a different story. Since available cores have to keep shifting to support an oversized number of processes, the latency introduced by the switching, especially when stateful instances are left idle, leads to a dramatic increase in total idle time. However, despite these, both *runtime* and *process time* for *hybrid_redis* remain superior to the *multi*.

Table 3: Performance comparison based Sentiment Analyses for News Articles workflow between techniques (A and B). The *runtime* ratio is the *runtime* of mapping A over the *runtime* of mapping B. Similarly, the *process time* ratio is the total process time elapsed ratio between A and B. The ratios are prioritized by the metric in the "Prioritized By" column, and the [Mean, Std] shows the average and standard deviation for all *runtime* and *process time* ratios.

Platform	Comparison between A/B	Prioritized By	Runtime Ratio	Process Time Ratio
server	$\frac{hybrid\_redis}{multi}$	*runtime*	0.32	0.48
		process time	0.32	0.48
		[Mean, Std]	[0.34, 0.02]	[0.49, 0.01]
cloud	$\frac{hybrid\_redis}{multi}$	*runtime*	0.60	0.89
		process time	0.60	0.89
		[Mean, Std]	[0.61, 0.01]	[0.95, 0.09]



(a) Workload Performance for Sentiment Analyses for News Articles using the *server* with up to 16 processes.



(b) Workload Performance for Sentiment Analyses for News Articles using the *cloud* with up to 16 processes.

Figure 12: Experiments on Sentiment Analyses for News Articles over different platforms.

5.4.1 Summary of the evaluation on Sentiment Analyses for News Articles with runtime and process time ratios. In terms of the *runtime* and *process time* ratios from various platforms and priorities, all ratios are smaller than 1. This indicates that, with *dynamic scheduling* optimization, *hybrid_redis* outperforms *multi*. This is especially noteworthy, based on the observation that the *Redis* mapping is overall slower than *Multiprocessing* with the same settings.

5.5 Analysis on *auto-scaling*

We experiment with the *auto-scaler* mechanism using the Internal Extinction of Galaxies and Seismic Cross-Correlation workflows on both *server* and *HPC*. In Figure 13, the left y-axis depicts the active process count throughout runtime. The right y-axis represents the queue size for *dyn_auto_multi* and the average idle time for *dyn_auto_redis*. The x-axis indicates iteration counts, recorded when monitored metrics (right y-axis) change. Please note that these iterations are not uniformly spaced in terms of time interval.

For *dyn_auto_multi* in Figure 13a, 13c, 13d and 13f, there is noticeable positive correlation between the number of active processes and queue size. It is in line with our expectations: a larger workload in the queue needs more active processes. Notably, in *HPC*, especially, for the simple workflow, the active size rarely reaches the maximum limit (64), even though the queue size consistently remains at a high level. This is probably because the naive *auto-scaler* of *dyn_auto_multi* adjusts the active size only by considering the changes in the queue size without the absolute workload.

For *dyn_auto_redis* in Figure 13b and Figure 13e, there is an inverse relationship between the number of active processes and the average idle time. This means that the *auto-scaler* reduces the active size when active processes have larger idle periods, indicating a reduced workload. The sub-figures reveal a consistent trend: active process numbers lag behind metric changes due to inertia in the naive *auto-scaling* strategy. This can result in mismatches between actual needs and active process count. We recognize the need for optimizing the auto-scaling strategy, enhancing its ability to accurately capture real requirements and predict workload changes.

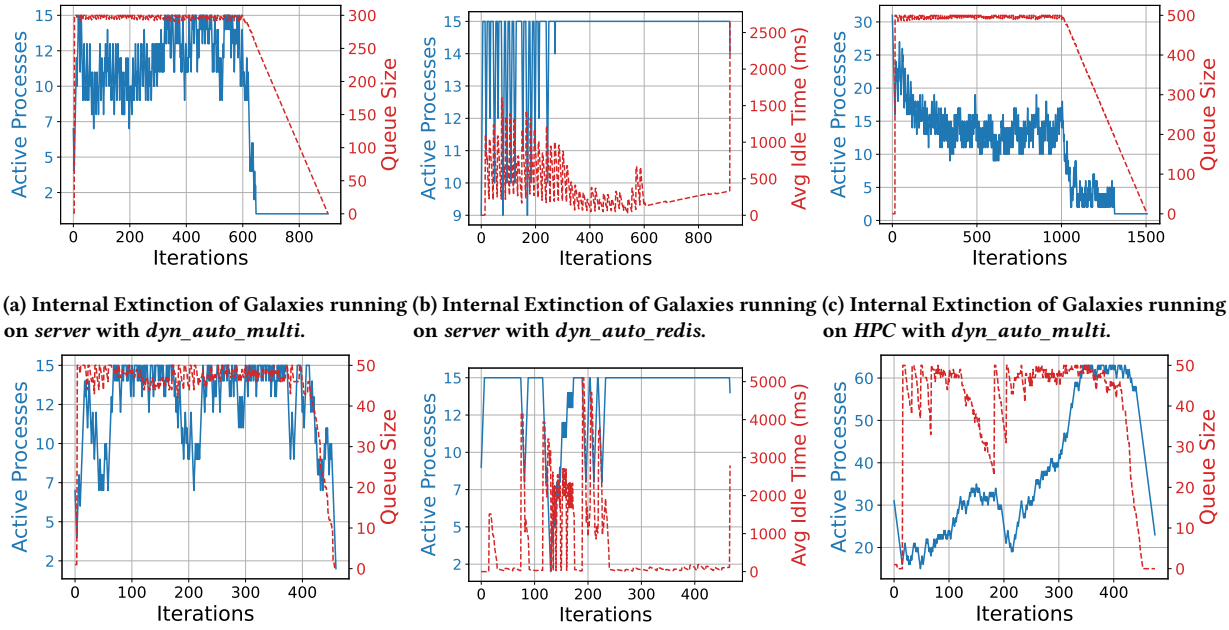
5.6 Key Insights

Summarizing our extensive experiments, key findings include:

- **Consistent *auto-scaling* Efficiency**: *auto-scaling* consistently demonstrates efficiency across diverse platforms and workflows. It achieves 87% *runtime* and 76% *process time* of *dynamic scheduling*'s performance in optimal cases
- **Complex Workflow Challenges**: While generally effective, *auto-scaling* faces challenges with complex workflows. Its auto-scaler can struggle to accurately predict needs, causing slight *runtime* extensions. However, *process time* efficiency remains strong.
- **Stateful Mapping Superiority**: In the context of workflows involving stateful applications, *hybrid_redis* surpasses its counterpart *multi*, achieving as low as 32% of *runtime* compared to *multi*.
- ***Multiprocessing* vs. *Redis* Performance**: The performance achieved with the *Multiprocessing* optimizations (*dyn_multi* and *dyn_auto_multi*) outperforms those of *Redis* (*dyn_redis* and *dyn_auto_redis*), primarily attributed to the lightweight nature of *Multiprocessing* - despite employing the same optimization strategies for both *Multiprocessing* and *Redis*.

6 CONCLUSIONS

This paper presents our work harnessing the capabilities of the *Redis* framework to incorporate the *dynamic scheduling* optimization, complemented by a new *auto-scaling* strategy. Furthermore, we



(a) Internal Extinction of Galaxies running on *server* with *dyn_auto_multi*. **(b) Internal Extinction of Galaxies running on *server* with *dyn_auto_redis*.** **(c) Internal Extinction of Galaxies running on *HPC* with *dyn_auto_multi*.**

(d) Seismic Cross-Correlation running on *server* with *dyn_auto_multi*. **(e) Seismic Cross-Correlation running on *server* with *dyn_auto_redis*.** **(f) Seismic Cross-Correlation running on *HPC* with *dyn_auto_multi*.**

Figure 13: Active Size and the monitoring metrics (Queue Size, and Idle Time) running with Internal Extinction of Galaxies workflow and Seismic Cross-Correlation workflow.

have expanded the horizon of *dynamic scheduling* optimizations to accommodate stateful applications through the introduction of a novel *hybrid_redis* approach. Through experiments across diverse workflows and platforms, we demonstrated that *auto-scaling* achieves efficiency while preserving performance in most cases. Notably, the stateful optimization (*hybrid_redis*) outperformed native mappings. While initial *auto-scaling* strategies have limitations, our study lays a foundation for refining and enhancing auto-scaling within `dispel4py`. This integration benefits `dispel4py` and offers insights for enhancing other scientific workflows. Our contributions pave the way for future advancements and provide valuable insights to the scientific community.

REFERENCES

[1] Muhammad Abdul-Mageed and Mona Diab. 2012. Toward building a large-scale Arabic sentiment lexicon. In *Proceedings of the 6th international global WordNet conference*. 18–22.

[2] Safaa Alkatheri, Samah Anwar Abbas, and Muazzam Ahmed Siddiqui. 2019. A comparative study of big data frameworks. *International Journal of Computer Science and Information Security (IJCSIS)* 17, 1 (2019), 66–73.

[3] Malcolm Atkinson, Sandra Gesing, Johan Montagnat, and Ian Taylor. 2017. Scientific workflows: Past, present and future. , 216–227 pages.

[4] Salil Bharany, Sandeep Sharma, Osamah Ibrahim Khalaf, Ghaida Muttashar Abdulsahib, Abeer S Al Humaimeedy, Theyazn HH Aldhyani, Mashael Maashi, and Hasan Alkahtani. 2022. A systematic survey on energy-efficient techniques in sustainable cloud computing. *Sustainability* 14, 10 (2022), 6256.

[5] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 31–42.

[6] Dirk Eddelbuettel. 2022. A Brief Introduction to Redis. arXiv:2203.06559 [stat.CO]

[7] Rosa Filgueira, Amrey Krause, Malcolm Atkinson, Iraklis Klampanos, Alessandro Spinuso, and Susana Sanchez-Expósito. 2015. dispel4py: An agile framework for data-intensive escience. In *2015 IEEE 11th International Conference on e-Science*. IEEE, 454–464.

[8] Rosa Filguiera, Amrey Krause, Malcolm Atkinson, Iraklis Klampanos, and Alexander Moreno. 2017. dispel4py: A python framework for data-intensive scientific computing. *The International Journal of High Performance Computing Applications* 31, 4 (2017), 316–334.

[9] Michael A Georgiou, Aristodemos Paphitis, Michael Sirivianos, and Herodotos Herodotou. 2019. Towards auto-scaling existing transactional databases with strong consistency. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 107–112.

[10] Amit Gupta and Sushant Jain. 2022. Optimizing performance of Real-Time Big Data stateful streaming applications on Cloud. In *2022 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 1–4.

[11] Ankit Jain. 2017. *Mastering apache storm: Real-time big data streaming using kafka, hbase and redis*. Packt Publishing Ltd.

[12] Liang Liang, Rosa Filgueira, Yan Yan, and Thomas Heinis. 2022. Scalable adaptive optimizations for stream-based workflows in multi-HPC-clusters and cloud infrastructures. *Future Generation Computer Systems* 128 (2022), 102–116.

[13] Liang Liang, Rosa Filguiera, and Yan Yan. 2020. Adaptive optimizations for stream-based workflows. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 33–40.

[14] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. 2015. A survey of data-intensive scientific workflow management. *Journal of Grid Computing* 13 (2015), 457–493.

[15] openmpi [n. d.]. OpenMPI: Open Source High Performance Computing. https://www.open-mpi.org.

[16] EG Radhika and G Sudha Sadasivam. 2021. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. *Materials Today: Proceedings* 45 (2021), 2793–2800.

[17] Biswajit Saha. 2014. Green computing. *International Journal of Computer Trends and Technology (IJCTT)* 14, 2 (2014), 46–50.

[18] storm [n. d.]. Apache Storm. http://storm.apache.org.

[19] Kundjanasith Thonglek, Kohei Ichikawa, Chatchawal Sangkeettrakarn, and Apivadee Piyatumrong. 2021. Auto-scaling system in apache spark cluster using model-based deep reinforcement learning. *Heuristics for Optimization and Learning* (2021), 347–360.

[20] Shveta Verma and Anju Bala. 2021. Auto-scaling techniques for IoT-based cloud applications: a review. *Cluster Computing* 24, 3 (2021), 2425–2459.