2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS) | 978-1-6654-5191-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/WORKS56498.2022.00009

# RADICAL-Pilot and Parsl: Executing Heterogeneous Workflows on HPC Platforms

Aymen Alsaadi[1], Logan Ward[2], Andre Merzky[1], Kyle Chard[2,4], Ian Foster[2,4], Shantenu Jha[1,3], Matteo Turilli[1,3]
[1]*Rutgers, the State University of New Jersey, Piscataway, NJ 08854, USA*
[2]*Data Science and Learning Division, Argonne National Laboratory, Lemont, IL 60439, USA*
[3]*Brookhaven National Laboratory, Upton, NY 11973, USA*
[4]*Department of Computer Science, University of Chicago, Chicago, IL, USA*

***Abstract*—Workflows applications are becoming increasingly important to support scientific discovery. That is leading to a proliferation of workflow management systems and, thus, to a fragmented software ecosystem. Integration among existing workflow tools can improve development efficiency and, ultimately, increase the sustainability of scientific workflow software. We describe our experience with integrating RADICAL-Pilot (RP) and Parsl as a way to enable users to develop and execute workflow applications with heterogeneous tasks on heterogeneous high performance computing resources. We describe our approach to the integration of the two systems and detail the development of RPEX, a Parsl executor which uses RP as its workload manager. We develop a RP executor that executes heterogeneous MPI Python functions on CPU cores and GPUs. We measure the weak and strong scaling of RPEX, RP and Parsl when providing new capabilities to two paradigmatic use cases: Colmena and Ice Wedge Polygons.**

***Index Terms*—Workflows, HPC, MPI executor, middleware integration.**

## I. Introduction

Workflow systems are becoming ubiquitous as they effectively abstract the complexity of orchestrating the execution of heterogeneous tasks across diverse computing resources. This has led to the development of hundreds of workflow systems [1] with significant overlap between their goals and capabilities. The development of these systems is inefficient: there is significant duplication of functionality and a lack of robustness as it is infeasible for a single workflow system to meet all application requirements on all potential resources. Recent summits organized by the workflows community [2] highlighted the need for workflow system interoperability as a way of reducing development inefficiency; improving robustness, performance, and portability; and ultimately enhancing the sustainability of the workflows community.

Task heterogeneity is fundamental to high performance computing (HPC) scientific workflows. Tasks may be standalone executables or functions implemented in diverse programming languages. Both executable and function tasks have diverse requirements: from single-core utility functions to multi-node MPI simulation executables. Supporting such task heterogeneity requires: (1) a workflow system for users to express and execute applications with diverse types of tasks; and (2) a workload manager capable of interpreting and managing the execution of those tasks at scale and on diverse HPC platforms.

Here, we present the integration of the Parsl [3] workflow system and the RADICAL-Pilot (RP) [4] workload manager, independently developed by different research groups. We adopt a loosely-coupled integration approach, developing a RP Executor (RPEX) for Parsl and a new RP executor to add the capability to distribute and execute MPI Python functions concurrently to (non)MPI executable tasks. Our integration brings new capabilities to both systems: Parsl users can benefit from the scalable and performant RP runtime capabilities with minimal or no changes to their code, while RP users gain a larger choice when deciding what workflow system to use for their applications, e.g., Parsl, EnTK, Swift.

We describe how our integration brings new capabilities to two use cases. Colmena [5], a Python package that uses Parsl to execute ensemble applications, gains new MPI capabilities via RPEX, RP and its new executor. Ice Wedge Polygons (IWP) benefits from Parsl's dataflow capabilities and Python API to implement a workflow application that uses RPEX and RP to concurrently execute multi-node MPI Python functions on CPUs and GPUs.

We measure strong and weak scaling of the new RP executor, and of RPEX for both Colmena and IWP, showing that overheads are small or invariant of scale. We compare the resource utilization of Colmena with RPEX, showing that is analogous to that obtained in Ref. [5] with Parsl's HTEX executor. The insight gained by our analysis shows the viability of the proposed integrative approach and offers useful information about how to improve the execution of heterogeneous tasks on HPC resources at scale.

## II. Related Work

The integration of workflow and runtime systems, and the building blocks approach to workflow middleware [6] extend functionalities and interfaces, enabling different programming paradigms and the execution of different applications on different platforms at a variety of scales.

978-1-6654-5191-8/22/$31.00 ©2022 IEEE
DOI 10.1109/WORKS56498.2022.00009



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

There have been multiple approaches to integrating the traditional big-data middleware stack with HPC workflow and resource management tools [7], [8]. Here the primary focus is on integration of traditional HPC software systems, for example, the integration of PyCOMPS [9] with other frameworks, or Swift [10] with RP. In the later, the integration was based on their application programming interfaces (API) and special-purpose connectors [11]. This reduced the engineering effort spent on each system, centering the development on a small and independent component that translates computational requirements between the workflow application layer, and the resource management and task execution layers.

Shaffer et al. [12] use the Parsl API in their integration of Parsl with the WorkQueue framework [13]. They achieve lightweight function monitoring across HPC resources by forking a new process for every executing function. However, this approach introduces: (i) additional resource requirements due to the creation of a monitoring process for each Python function executed, and (ii) additional overhead associated with launching that extra process.

Merlin [14] is designed to enable the execution of large ensembles simulations and machine learning analyses on HPC platforms. Merlin uses the Maestro [15] interface to define workflows of millions of tasks, and deploys Flux [16] to scale such workflows on HPC systems. However, its use of Maestro's YAML-based interface to define tasks restricts it to a shell syntax, making it challenging for the end-user to take advantage of clearer and more powerful programming languages, such as Python.

## III. Use Cases

We present two exemplar use cases that require the capabilities of both RP and Parsl to execute (non)MPI executables and Python function tasks on HPC platforms.

### A. Colmena: Intelligent Steering of Ensemble Simulations

Colmena [5] is a Python package for machine learning-based steering of ensemble computations on HPC platforms, for such purposes as fitting interatomic potentials [17]. A Colmena application is organized as a *Thinker* process that implements a strategy for selecting computations that are then submitted to a *Task Server* process for execution. Currently, Colmena Task Server uses the Parsl workflow engine to dispatch tasks to multiple processors.

The computations managed by Colmena applications are frequently MPI programs of modest scale. Thus, in order to make efficient use of large parallel computers, Colmena needs to run efficiently many MPI applications at once—something that existing Parsl executors are not able to do. Parsl deploys MPI tasks using a single worker on an HPC launch node that is responsible for pre- and post-processing tasks and invoking the MPI launcher using subprocesses. Large ensemble can lead to overheads of minutes, as processing tasks compete for resources and requests overwhelm the MPI launcher. Colmena is thus an excellent use case for the new RPEX executor.

### B. Ice Wedge Polygons

Ice wedges are common permafrost subsurface attributes that evolved by accumulated frost cracking and ice-vein growth over long periods of time. These wedge-shaped ice masses create polygonized land surface patterns called Ice Wedge Polygons (IWP) across large Arctic areas. Observing IWP requires processing very high spatial resolution (VHSR) satellite imagery at multiple spatial scales [18].

IWP is implemented via MPI Python functions that require the concurrent use of both GPUs and CPUs. Each image is processed by performing two operations—tiling and inference. Tiling uses CPUs to divide each image into 360×360 pixels tiles; inference uses a GPU to extract the surface patterns from each tile.

The RPEX executor offers the required workload runtime capabilities via RADICAL-Pilot and a flexible programming model via Parsl to execute IWP multi-node MPI Python functions concurrently on CPUs and GPUs.

## IV. RADICAL-Pilot And Parsl Integration

We integrate RP and Parsl into a system that we name, for simplicity, RPEX. Importantly, we implement and extend an existing interface between the two systems 'as they are,' providing users with the sum of the two systems' capabilities, without engineering a whole new system. This integration allows RP to benefit from Parsl flexible programming model and its workflow management capabilities to build dynamic workflows. Additionally, RPEX will benefit Parsl by offering the heterogeneous runtime capabilities of RP to support many MPI computations more efficiently than with other Parsl executors.

### A. RADICAL-Pilot (RP)

RP is a scalable, modular, and interoperable pilot system, coded in Python, that enables the execution of heterogeneous workloads on heterogeneous HPC resources. RP has four main components [4]: The *Pilot Manager* and *Task Manager*, which are executed on a user resource or on the login node of an HPC platform; the *Agent*, which is executed on the compute nodes of the target HPC platform; and a *MongoDB* database, which is hosted on resources accessible via network by the other components.

RP provides methods for efficiently and effectively scheduling, placing, and launching independent tasks across multiple nodes. RP uses the pilot abstraction [19] to support the concurrent execution of up to $10^6$ tasks on $10^3$ compute nodes with low overheads [4].

Different from other pilot systems, RP supports tasks that may vary simultaneously along four dimensions; (1) programming model, including single/multi cores/GPUs with MPI, OpenMP, and single/multi-[threaded|process] tasks; (2) scale, from 1 to 27,000 GPUs and/or 1 to 467,000 CPU cores; (3) task duration, from >1 second to 24 hours or more; and (4) task packaging method: both standalone executables and Python functions.



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

### B. The Parsl parallel Python programming library

Parsl [3] is a Python module that enables parallel execution of Python functions and orchestration of functions into dataflow workflows. Parsl users decorate Python functions to indicate opportunities for concurrent execution. Parsl relies on *futures* to abstract asynchronous execution: invocation of a Parsl app returns a future to the calling program. The future's state is set only when the app completes execution; an attempt to read the future before that time causes the application to block. Parsl enables dataflow semantics by allowing developers to pass futures between apps.

Fig. 1 shows the three main components of the Parsl implementation: the Data Flow Kernel (DFK), Executor, and Provider. Parsl applications start when a Python program calls a Parsl app and passes either input arguments or a future from another app. The DFK wraps each task with a Python future object [20]. Throughout execution, the DFK maintains a directed acyclic graph (DAG) with nodes representing each invocation of an app (called a 'task') and edges representing futures passed between apps. Once a task's dependencies are resolved, the DFK submits it to one or more user-specified executors. The DFK tracks every task's state, updating the task graph.

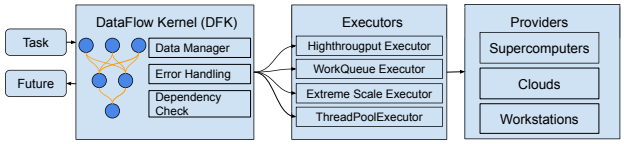


Fig. 1. Parsl architecture and execution model.

Parsl relies on Python's standard `concurrent.futures` executor interface to dispatch tasks for execution. Parsl includes two in-built executors and an external executor that implement this interface, each designed for a specific type of workload. The high-throughput executor (HTEX) is a pilot-based executor for rapid execution of many tasks. The Extreme-Scale Executor (EXEX) executes tasks on a pool of multi-node processes, using the Python package `mpi4py` [21] to build and manage communications between managers and workers. The WorkQueue Executor (WQEX) uses WorkQueue to provide managed task execution with dynamic resource sizing.

### C. Design

RP's architecture [4] and Parsl's architecture (Fig. 1) suggest two integration points: using RP to submit tasks to an existing Parsl executor or using the RP Agent as a new Parsl executor. While the former integration point would extend Parsl's HPC resource acquisition capabilities, it would not allow users to benefit from most of RP's Agent capabilities. The latter integration point allows us to maintain Parsl API and workflow-related capabilities while benefiting also of RP runtime capabilities.

RP can provide MPI support across multiple HPC platforms, supporting multiple dimensions of task and resource heterogeneity. Further, RP supports Single Program Multiple Data (SPMD) and its performance is tailored to extreme-scale on HPC resources [4]. It also supports the concurrent execution of multiple pilots on multiple HPC platforms [22] and the scheduling of a single workload across those pilots [11].

Integrating RP as a Parsl executor requires aligning the two systems' task execution models. RP's tasks are fully-decoupled, i.e., they have no data dependencies or those dependencies have been already satisfied out-of-band. Each task is assumed to be self-contained, executed by RP as a black-box that either returns or fails. RP has no knowledge of the code each task executes, enabling a separation of concern between resource and execution management, and task executables. Consistently, at application level, RP implements a 'batch-like' programming model in which groups of tasks (i.e., workloads) are described and submitted for execution. Concurrency is implicit: once submitted, RP executes tasks with the maximum concurrency allowed by the available resources.

Unlike RP tasks, Parsl tasks have dynamic data dependencies that must be respected before execution. At the application level, Parsl allows for the expression of nested parallelism within a single task or across a batch of tasks. Parsl programming model enables various parallel computing paradigms such as procedural and dynamic workflow execution and interactive parallel programming.

Parsl's tasks are Python functions while RP tasks are Python dictionaries that are dynamically updated to reflect the state of the tasks. The difference in the task object's type is a communication barrier that we overcame by implementing a mid-point component called "Task Translator", with the following capabilities: (i) detect whether Parsl task is a pure Python function or a Python call to a Bash command; (ii) translate Parsl tasks into RP tasks; and (iii) update the status of Parsl tasks, according to callbacks from RP tasks.

Fig. 2 illustrates the translation of Parsl tasks into RP tasks. Each Parsl task is translated via a direct (1:1) mapping in accordance to the task submission criteria of Parsl's DFK. Thus, tasks are created at application level and submitted to the executor one by one, iteratively.

### D. Implementation

We implement a new Parsl executor for RP shown in Fig. 2 and we call it RADICAL-Pilot Executor (RPEX). RPEX is a Python class that bootstraps RP when initialized by Parsl. To make RPEX consistent with other Parsl executors, we based RPEX's implementation on the Parsl HTEX executor class.

Note that Parsl does not require resource specification at task level, while RP requires specification of the number of cores and threads for both CPU cores and GPUs for every task. To enable the use of RP's resource management



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

capabilities in RPEX, we extended Parsl's API to allow users to define those parameters.

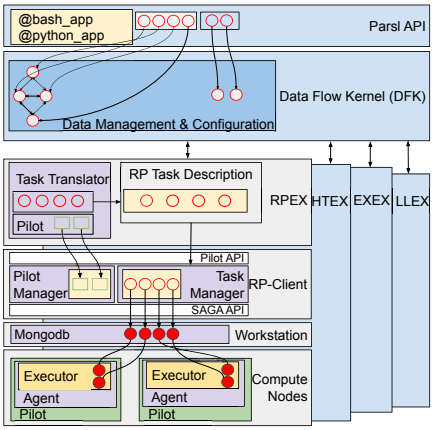


Fig. 2. RPEX integration architecture.

Once Parsl starts, it initializes DFK and RPEX simultaneously. Upon initialization, the DFK: (i) obtains the tasks from Parsl API; (ii) builds the tasks table; (iii) solves each task dependencies; and (iv) passes each task object to RPEX for execution (Fig. 2). Once initialized, RPEX: (i) obtains via its interface the HPC platform on which to execute the tasks and the amount of walltime for which to hold the resources; (ii) starts a new RP session and creates the Pilot Manager and the Task Manager; and (iii) obtains the number of CPU cores and GPUs required by each task submitted by the DFK.

Submitting and executing Parsl tasks via RP require translating those tasks into task objects that can be interpreted by RP. Each task object has a set of properties, e.g., the executable's name, its type, its arguments (if any), the number and type of resources, the number of processes, etc. Once the DFK submits the Parsl task to RPEX, the Task translator unpacks, translates and maps that task to the corresponding RP task object.

Parsl's DFK monitors the status of RPEX and, once ready, it starts submitting Parsl tasks one by one to RP (Fig.2, RP-Client). Eventually, RP submits the task to its Task Manager to be scheduled and executed on the pilot resource (Fig.2, Compute Nodes).

### *E. RP MPI Function Executor*

To support launching and executing of MPI Python functions, we implemented a RP single/multi-node MPI function executor shown in Fig. 3. Not to be confused with RPEX, that is a Parsl executor, the RP executor uses `mpi4py` to implement a task-based SPMD master-worker paradigm to concurrently execute heterogeneous MPI Python functions.

The MPI executor communicates with other RP's component via ZeroMQ, sending and receiving MPI Python functions, until terminated by RP's Agent. Once scheduled by RP, the MPI function executor: launches itself

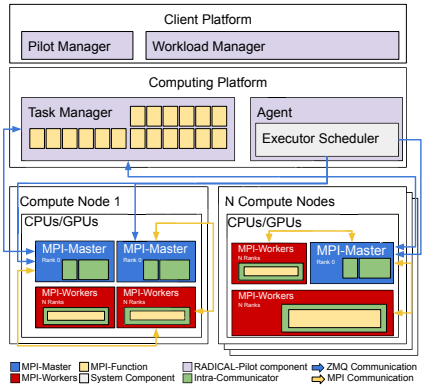


Fig. 3. RADICAL-Pilot multi-node MPI function executor.

via a user-specified MPI launch method; loads the `mpi4py` environment once for all incoming tasks; and spawn the MPI-Master and MPI-Workers.

The MPI-Master decomposes the main MPI-Communicator into several Intra-Communicators to serve as a private communicator for every Python function. Once the MPI-Master receives the functions via a ZMQ channel, it sends them to the designated workers for execution. Workers can concurrently run on single or multiple nodes. Every MPI-Master is responsible for coordinating the execution of a set of Python functions and performing MPI collective communications among the workers (see Fig. 3).

## V. Experiments and Evaluation

Table I shows the setup of our experiments. We use both SDSC Expanse and TACC Frontera for Experiment 1, and Frontera only for Experiment 2. Expanse compute nodes have 128 CPU cores, while Frontera has two types of nodes: "normal" with 56 CPU cores and no GPUs, and "rtx" with 16 CPU cores and 4 GPUs.

We use three metrics: Total Processing Time (TPT); Throughput (TS); and Total Time to Execution (TTX). TPT is the time spent by our executor to finish executing all the tasks of a workload. TS is the number of tasks executed per second, calculated by dividing the total number of tasks by TPT. TTX is the total amount of time taken by all tasks to finish executing. Note that TPT measures the time in which the executor kept the resources busy, excluding any idle or wait time. In contrast, TTX measures the time the workload spent to finish the execution of all tasks on those resources, including idle and wait time.

Experiment 1 measures the TPT and TS of the MPI-function executor presented in §IV, as a function of the number of tasks. Experiment 2 measures TTX and RPEX integration overheads with the two use cases described in §III. Together, these experiments enable us to characterize the performance of the integrated RPEX and of the Python MPI function executor with different resources, task heterogeneity, and scale on HPC resources.



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SETUP FOR EXPERIMENTS 1 AND 2. WS/SS = WEAK/STRONG SCALING; COL = COLMENA; IWP = ICE WEDGE POLYGONS.

| ID | Experiment Type | Platform | Nodes | Task Type | CPUs(cores) | GPUs |
|---|---|---|---|---|---|---|
| 1 | MPI executor WS/SS | Expanse<br>Frontera | $2-32$<br>$8-512$ | MPI-Homogeneous | #nodes $\times$ 128<br>#nodes $\times$ 56 | N/A |
| 2 | COL/IWP WS/SS | Frontera | $32-256$ | MPI-Heterogeneous | #nodes $\times$ 56 | $8-32$ |

TABLE II
EXPERIMENT 1 STRONG AND WEAK SCALING RESULTS ON EXPANSE AND FRONTERA. N = NUMBER OF COMPUTE NODES.

| System | Scaling | N | Total processing time (seconds) | Throughput (tasks/second) |
|---|---|---|---|---|
| Expanse | Strong | 2 | 6752.4 ±153.9 | 4.7 ±0.1 |
| | | 4 | 3494.4 ±199.0 | 9.2 ±0.4 |
| | | 8 | 1758.4 ±88.5 | 18.3 ±0.8 |
| | | 16 | 911.3 ±43.0 | 35.3 ±1.6 |
| | Weak | 2 | 409.5 ±4.9 | 4.8 ±0.05 |
| | | 4 | 423.1 ±9.4 | 9.4 ±0.2 |
| | | 8 | 412.1 ±2.5 | 19.4 ±0.1 |
| | | 16 | 430.5 ±4.1 | 37.1 ±0.3 |
| | | 32 | 423.5 ±4.8 | 75.5 ±0.8 |
| Frontera | Strong | 8 | 14173.1 ±375.2 | 36.1 ±0.9 |
| | | 16 | 7458.4 ±341.9 | 69.0 ±2.8 |
| | | 32 | 3546.8 ±105.6 | 144.7 ±4.0 |
| | | 64 | 2035.3 ±97.8 | 235.2 ±11.5 |
| | | 128 | 1236.8 ±150.6 | 431.6 ±51.4 |
| | | 256 | 509.1 ±8.6 | 1005.8 ±17.1 |
| | Weak | 8 | 231.3 ±6.1 | 34.6 ±0.8 |
| | | 16 | 228.8 ±5.2 | 70.0 ±1.6 |
| | | 32 | 221.9 ±4.4 | 144.2 ±2.8 |
| | | 64 | 238.5 ±14.0 | 270.0 ±15.6 |
| | | 128 | 258.3 ±14.5 | 498.3 ±26.7 |
| | | 256 | 309.4 ±50.3 | 868.9 ±129.3 |
| | | 512 | 303.7 ±17.5 | 1696.7 ±94.2 |

### A. Experiment 1: MPI-Function Executor Scalability

We characterize the performance of the MPI-function executor on Expanse and Frontera, measuring its strong and weak scaling in terms of TPT and TS. We summarize the results in Table II. Note that the processing time of the MPI function executor measured by TPT includes the aggregated overheads of launching the MPI infrastructure and of the MPI communications.

We use a homogeneous workload of Python `no-op` functions. We launch the executor with `mpirun` and configure it to execute each MPI function across two compute nodes, using `mpi4py` for each function. Every function is configured to run on 256 ranks on Expanse (128 cores per node) and 112 ranks on Frontera (56 cores per node). In this way, each task tests our executor's multi-node capability, scaling to sizable portions of the HPC platforms.

*1) SDSC Expanse:* Fig. 4 shows the strong (a) and the weak (b) scaling of our MPI executor in terms of TPT (blue) and TS (orange). For strong scaling, TPT decreases linearly while TS increases linearly as a function of the number of nodes across all runs (see Table II). Both TPT and TS show a consistent behavior across the experiment's scale while maintaining small error bars, indicating an efficient scaling behavior.

In the weak scaling case, the TPT shows consistent scaling with small error bars across all runs. Further, the TS increases linearly with the number of nodes, reaching a maximum of 75.5 task/s on 32 nodes. The linear increase of TS in Fig. 4 (a,b) show a positive correlation between the number of nodes and tasks, indicating that the executor can achieve higher TS at larger scales.

Fig. 4 (a,b) show that our executor scales efficiently on the assigned resource, using a homogeneous workload in which each function requires the same number of ranks. We use a homogeneous workload as it allows us to study the baseline performance of the executor since this type of workload is more generalizable and easier to measure across scales.

*2) TACC Frontera:* We used the same experiment setting of Expanse to characterize strong and weak scaling of the MPI function executor on Frontera at larger scale (see Table II). Fig. 4 (c,d) show the strong and the weak scaling of the MPI executor's TPT (blue) and TS (orange).

In the strong scaling case, TPT decreases linearly and TS increases exponentially with the number of nodes, with small error bars across all runs. In the weak scaling case, TPT remains stable from 8 to 64 nodes and then increases sub-linearly from 128 to 512 nodes. Note that the error bars of 8–64 nodes overlap with the ones of 128–512 nodes, making the difference between TPTs statistically less significant. The sub-linear increase in TPT between 128 and 512 nodes is due to the MPI collective communication overheads, leading to TPT deterioration with increasing numbers of resources [23]. TS shows an exponential increase between 8 and 512 nodes while maintaining relativity small error bars across all runs. This confirms our analysis in §V-A1 which shows that our executor reaches higher TS on larger scales consistently.

Comparing results between Expanse and Frontera we see that (i) as expected, the cost of constructing and launching an MPI task grows with the number of ranks (256 ranks vs. 112 ranks), since MPI takes more time to group and construct a larger MPI-communicator [24]; (ii) TPT increases proportionally with the number of resources due to the increased number of MPI communication overheads, and the number of ranks per task.

The MPI executor scales efficiently between 8 and 64 nodes on Frontera and 2 and 32 nodes on Expanse, but given the proportional increase of the overheads between 128 and 512 nodes, performance starts to deteriorate at a larger scale on Frontera (see Table II). Constructing an MPI Intra-communicator for every function is expensive



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

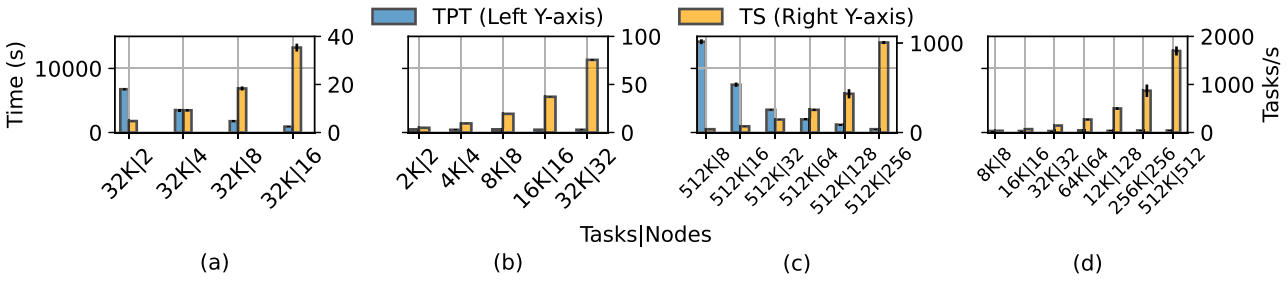


Fig. 4. Experiment 1: Scaling properties of MPI Function executor. (a) and (b) characterize strong and weak scaling on Expanse, respectively; (c) and (d) characterize strong and weak scaling on Frontera, respectively.

but necessary in presence of heterogeneous functions. The impact of that overhead on the overall workload execution depends on the duration of the functions execution. For short-running homogeneous functions, a more performant design would: (i) set up the Intra-communicator only once to be reused by every task; (ii) take advantage of caching capabilities for MPI Intra- and Inter-communicator.

### *B. Experiment 2: Use Case Scalability*

Next we study RPEX strong and weak scaling by running the Colmena and IWP use cases (§III) on Frontera while varying both problem size and number of nodes.

Both use cases require capabilities that can be provided only by RPEX and not by either RP or Parsl alone. Colmena requires execution of a workflow of concurrent heterogeneous single-node MPI executables and single core non-MPI Python functions. IWP requires instead a workflow of heterogeneous MPI Python functions that run concurrently on multiple nodes.

We summarize our experiment results in Table III, using three metrics: RP overheads, RPEX integration overheads, and total time to completion (TTX). Note that we measure RPEX overhead as the sum of Parsl and RP overheads. Parsl's overhead includes the amount of time taken to: (1) start the executor; (2) build the DAG of tasks; (3) solve the data dependencies among all tasks; (4) submit the tasks to the executor; and (5) shutdown and cleanup both the executor and the integration components. RP's overhead consists of the amount of time taken to: (1) start the runtime system; and (2) manage the tasks' execution.

*1) Colmena:* We created a heterogeneous synthetic workflow based on a real-world Colmena application [5] to evaluate the new capabilities provided by RPEX. The workflow consists of three tasks: Python "pre-process" and "post-process" functions, and a C MPI "simulation" executable that runs for ∼100s. The pre-process function prepares the execution environment for the simulation MPI task, while the post-processing function collects results from the simulation tasks, storing them in a Python class object. Each pre-process and post-process function requires one CPU core, while each simulation task requires a full node (56 CPU cores). We used the TACC-specific MPI launcher `Ibrun` to launch the MPI executables.

TABLE III
EXPERIMENT 2 STRONG AND WEAK SCALING RESULTS ON FRONTERA. N = NUMBER OF COMPUTE NODES. TIMES ARE IN SECONDS.

| App | Scaling | N | Total time to execution | RP overhead | RPEX overhead |
|---|---|---|---|---|---|
| Colmena | Strong | 32 | 8725.6 ±7.0 | 104.4 ±4.5 | 724 ±7.5 |
| | | 64 | 3961.9 ±2.0 | 116.4 ±11.8 | 744.6 ±13.5 |
| | | 128 | 1929.2 ±19.0 | 99.7 ±5.0 | 769.1 ±10.5 |
| | | 256 | 3263.9 ±94.7 | 176.2 ±56.4 | 855.2 ±56.2 |
| | Weak | 32 | 620.9 ±1.3 | 173.1 ±0.2 | 217.5 ±2.1 |
| | | 64 | 629.1 ±0.9 | 118.8 ±1.4 | 207.8 ±1.7 |
| | | 128 | 636.5 ±0.6 | 233.7 ±34.7 | 253.0 ±37.1 |
| | | 256 | 1891.8 ±61.6 | 134.8 ±2.0 | 388.2 ±74.3 |
| Ice Wedge Polygons | Strong | 2 | 10620.3 ±34.0 | 6.3 ±2.1 | 7.1 ±2.1 |
| | | 4 | 4895.9 ±1.4 | 5.5 ±3.0 | 6.4 ±2.1 |
| | | 8 | 2460.9 ±4.9 | 4.6 ±2.3 | 5.4 ±2.4 |
| | | 16 | 1344.6 ±51 | 5.5 ±2.3 | 6.4 ±0.1 |
| | Weak | 2 | 211.6 ±5.6 | 5.7 ±1.5 | 5.7 ±1.6 |
| | | 4 | 236.1 ±0.1 | 6.0 ±1.0 | 6.0 ±1.1 |
| | | 8 | 259.9 ±1.9 | 7.4 ±1.9 | 7.4 ±1.9 |
| | | 16 | 275.0 ±11.8 | 6.2 ±1.4 | 6.3 ±1.4 |

Fig. 5 shows strong (a) and weak (b) scaling with RPEX executing the Colmena workflow. TTX (red) decreases linearly between 32 and 128 nodes in strong scaling and maintains a consistent scale in the weak scaling. RP overheads (purple) are essentially invariant of scale, while RPEX overheads (blue) increase with scale in the weak scaling and maintains a relativity consistent behavior in the strong scaling. The scaling behavior changes at 256 nodes for both strong and weak scaling, showing a linear increase of TTX compared to the 32–128 nodes run. We investigated the TTX increase by measuring the resource utilization of Colmena workflows while executing 450/32, 900/64, 1800/128 and, 3600/256 tasks/nodes.

In Fig. 6, we measure and break down the resource utilization of Colmena's TTX based on four task-related events: "Scheduled" indicates resources being assigned to tasks and ready for execution; "Launching" indicates the resources occupied while waiting for `Ibrun` to launch the scheduled tasks by RP and also represents `Ibrun` overheads; "Running", which shows the resources occupied by



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

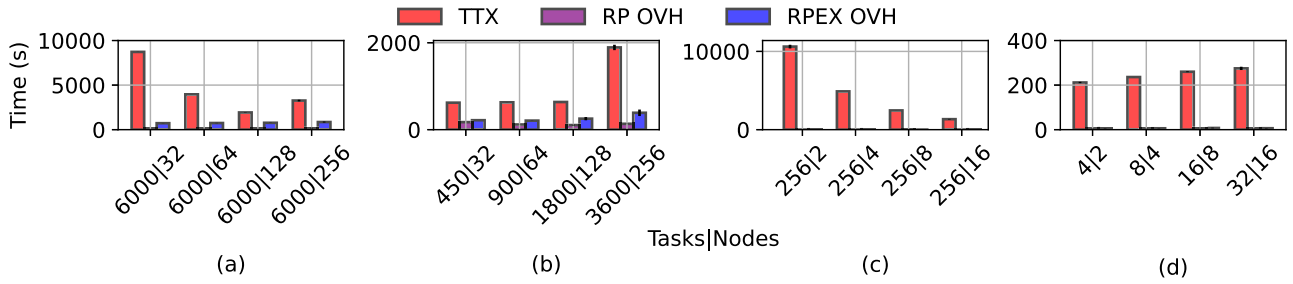


Fig. 5. Experiment 2: Scaling properties of Colmena and Ice Wedge Polygons (IWP). (a) and (b) characterize strong and weak scaling of Colmena; (c) and (d) characterize the strong and weak scaling in §III. RPEX overheads include RP overheads and represent the total overhead of the execution. RP overheads represent the time spent by RP and its executor to manage the execution of the workload/workflow.

RP while executing the launched tasks; and "Idle" the time in which the available resources are occupied but not busy.

Fig. 6 (a,b,c) show that Running (dark green) occupies ∼98% of the available resources with average Launching time of 30s, 65s and 215.3s. Fig. 6 (d) shows instead that Launching (light green) becomes the dominant activity, occupying most of the resources for ∼1791.2s. Launching creates a 'busy wait' condition that prevents tasks from executing. This explains the increase in Colmena's TTX shown in Fig. 5 (a,b): RP takes longer to execute on 256 nodes compared to 32–128 nodes because it has to wait longer for `Ibrun` to launch the tasks.

Comparing Colmena's resource utilization from prior work [5]—while executing only Python functions without RP—to Fig. 6 shows comparable resource utilization. RPEX reaches a resource utilization of ∼ 99% while executing both MPI-executables and Python functions, maintaining the performance measured when Colmena executed only non-MPI functions via Parsl. RPEX results in Fig. 5 also show that RPEX has the potential to reach large scales with low and constant overheads (RP overhead). Substituting `Ibrun` with more performant MPI libraries has the potential to lower the task launching overheads at scale [4], [25].

*2) Ice Wedge Polygons (IWP):* We implemented the IWP use case of §III by using the Single Program Multiple Data (SPMD) MPI pattern, where tasks are split up and concurrently executed on multiple cores with different inputs [26]. We configured RPEX to use RP's MPI function executor, executing the IWP workload with the SPMD pattern. We used 2 GPUs and 8 CPU cores per task, with up to 256 concurrent tasks on Frontera.

Fig. 5 shows the strong (c) and weak (d) scaling of RPEX with the IWP use case workload. In the strong scaling case, TTX (red) decreases exponentially. In the weak scaling case, TTX shows a sublinear increase across all runs (see Table III). As with the Colmena use case, RP overheads shows a consistent behavior across all the runs. Further, RPEX overheads in the strong and weak scaling maintains a consistent behavior while executing on $\leq$ 128 nodes. The scaling of the TTX confirms that `Ibrun` overheads increases marginally with the number of tasks and resources when executing on $\leq$ 128 nodes and becomes intolerable with $>$ 128 node runs. Overall, RP and RPEX overheads are low compared to the TTX of the use case and the scale of the experiments.

## VI. Conclusions and Future Work

We described an integration of RADICAL-Pilot's pilot capabilities with Parsl's workflow management capabilities to enable the execution of production workflows on diverse HPC platforms. The four main contributions of this paper are: (1) a case study of the engineering process used to integrate two independently developed middleware systems; (2) an analysis of the design of an executor for MPI Python functions tailored to HPC resources; (3) a performance characterization of both the integrated system and the proposed executor on two HPC platforms; and (4) a description of how the integrated system and executor have been used for two exemplar use cases.

The RPEX integration shows that integrating independent middleware systems requires an analysis of their capabilities, execution and state models, private and public APIs, and performance bottlenecks. We showed that, based on that analysis, the integration's engineering effort can be reduced, while avoiding expensive rewriting of existing code bases. Nonetheless, we also showed that user-facing APIs might have to be extended to make specific information available across the integrated systems.

We successfully supported two classes of use case. First, we used RPEX in Colmena, enabling efficient execution of both MPI and single node Python functions. Importantly, no changes were required to Colmena to make use of RPEX. Second, by supporting IWP, we showed how RPEX can be used out of the box to code specific workflows and run them at scale. Together, these results confirm that the integration of independent middleware components can be a viable approach to reducing capability duplication across middleware, especially when considering the challenges posed by executing workflows in production on Exascale HPC platforms.

Our experiments show that RPEX overheads increase only marginally with the number of tasks and resources.



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.

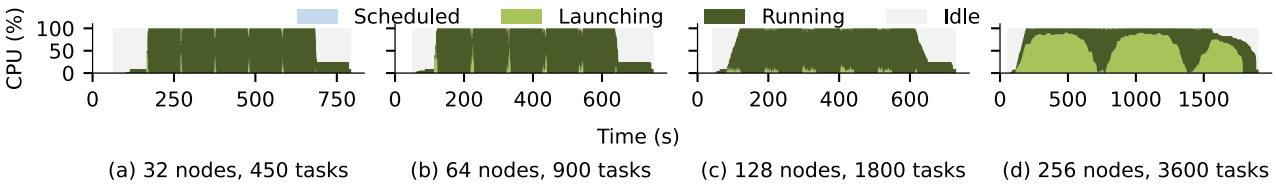


Fig. 6. Experiment 2: Colmena resource utilization with RPEX on 32, 64, 128 and 256 nodes

This result shows that our integration approach, based on the weak coupling of the two systems via a light-weight, stand-alone interface, does not introduce major overheads when executing heterogeneous workflows on HPC resources. It also highlights highlights the limitations of the MPI library used to launch tasks at scale and establishes the need for developing workflow-specific low-level communication libraries.

The performance of RPEX can be improved by adopting a different task submission logic. Currently, RPEX submits a stream of tasks to the runtime engine which increases overhead at scale, especially with short running tasks. We are developing a `bulk submission` mode for RPEX, which can reduce overheads, improving task submission, scheduling, and execution throughput.

This work was supported by the ECP ExaWorks and ExaLearn projects, as well as NSF-1931512 (RADICAL-Cybertools). HPC access on XSEDE was provided by allocation TG-MCB090174.

## References

[1] M. R. Crusoe *et al.*, "Computational data analysis workflow systems," 2022, https://s.apache.org/existing-workflow-systems.

[2] R. Ferreira da Silva, H. Casanova *et al.*, "Workflows community summit: Advancing the state-of-the-art of scientific workflows management systems research and development," Tech. Rep., 2021. [Online]. Available: https://zenodo.org/record/4915801

[3] Y. Babuji, A. Woodard *et al.*, "Parsl: Pervasive parallel programming in Python," in *28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2019, p. 25–36.

[4] A. Merzky, M. Turilli *et al.*, "Design and performance characterization of RADICAL-Pilot on leadership-class platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 818–829, 2021.

[5] L. Ward, G. Sivaraman *et al.*, "Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing," in *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments*, 2021, pp. 9–20.

[6] M. Turilli, V. Balasubramanian *et al.*, "Middleware building blocks for workflow systems," *Computing in Science & Engineering*, vol. 21, no. 4, pp. 62–75, 2019.

[7] J. Wang, D. Crawl, and I. Altintas, "Kepler + Hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems," in *4th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2009.

[8] A. Luckow, I. Paraskevakos *et al.*, "Hadoop on HPC: Integrating Hadoop and pilot-based dynamic resource management," in *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2016, pp. 1607–1616.

[9] R. M. Badia, J. Conejero *et al.*, "Pycompss as an instrument for translational computer science," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 79–84, 2022.

[10] M. Wilde, M. Hategan *et al.*, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[11] M. Turilli, Y. N. Babuji *et al.*, "Evaluating distributed execution of workloads," in *13th International Conference on e-Science*. IEEE, 2017, pp. 276–285.

[12] T. Shaffer, Z. Li *et al.*, "Lightweight function monitors for fine-grained management in large scale Python applications," in *Int. Parallel and Distributed Processing Symposium*. IEEE, 2021, pp. 786–796.

[13] P. Bui, D. Rajan *et al.*, "Work Queue + Python: A framework for scalable scientific ensemble applications," in *Workshop on Python for High Performance and Scientific Computing at SC11*. Citeseer, 2011.

[14] J. L. Peterson, R. Anirudh *et al.*, "Merlin: Enabling machine learning-ready HPC ensembles," 2019, arXiv 1912.02892.

[15] F. D. Natale, "Maestro workflow conductor," https://github.com/LLNL/maestrowf, 2019.

[16] D. H. Ahn, J. Garlick *et al.*, "Flux: A next-generation resource management framework for large HPC centers," in *43rd International Conference on Parallel Processing Workshops*, 2014, pp. 9–17.

[17] J. Guo, L. Ward *et al.*, "Composition-transferable machine learning potential for LiCl-KCl molten salts validated by high-energy x-ray diffraction," *Physical Review B*, vol. 106, no. 1, p. 014209, 2022.

[18] C. Witharana, M. A. E. Bhuiyan, and A. K. Liljedahl, "An object-based approach for mapping tundra ice-wedge polygon troughs from very high spatial resolution optical satellite imagery," *Remote Sensing*, vol. 13, no. 4, 2021.

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

[20] "Python 3.9.4 documentation: Concurrent.futures - launching parallel tasks," 2020.

[21] L. Dalcín, R. Paz, and M. Storti, "MPI for Python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[22] M. Turilli, F. Liu *et al.*, "Integrating abstractions to enhance the execution of distributed applications," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 953–962.

[23] S. Höfinger, T. Ruh, and E. J. Haunschmid, "Fast approximate evaluation of parallel overhead from a minimal set of measured execution times," *Parallel Processing Letters*, vol. 28, no. 1, pp. 1 850 003:1–1 850 003:12, 2018.

[24] J. Dinan, D. Goodell, and W. Gropp, "Efficient multithreaded context ID allocation in MPI," in *19th European Conference on Recent Advances in the Message Passing Interface*. Springer-Verlag, 2012, p. 57–66.

[25] M. Turilli, A. Merzky *et al.*, "Characterizing the performance of executing many-tasks on Summit," in *IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*. IEEE, 2019, pp. 18–25.

[26] F. Darema, "SPMD computational model," in *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 1933–1943.



Authorized licensed use limited to: Rutgers University. Downloaded on August 04,2023 at 14:12:57 UTC from IEEE Xplore. Restrictions apply.