Enabling HPC Scientific Workflows for Serverless

Anderson Andrei Da Silva*, Rolando Pablo Hong Enriquez*, Gourav Rattihalli*, Vijay Thurimella*,
Rafael Ferreira da Silva‡, Dejan Milojicic*
*Hewlett Packard Labs, Milpitas, CA, USA
‡Oak Ridge National Laboratory, Oak Ridge, TN, USA
{da-silva, rhong, gourav.rattihalli, vijay.thurimella, dejan.milojicic}@hpe.com, silvarf@ornl.gov

***Abstract*—The convergence of edge computing, big data analytics, and AI with traditional scientific calculations is increasingly being adopted in HPC workflows. Workflow management systems are crucial for managing and orchestrating these complex computational tasks. However, it is difficult to identify patterns within the growing population of HPC workflows. Serverless has emerged as a novel computing paradigm, offering dynamic resource allocation, quick response time, fine-grained resource management and auto-scaling. In this paper, we propose a framework to enable HPC scientific workflows on serverless. Our approach integrates a widely used traditional HPC workflow generator with an HPC serverless workflow management system to create benchmark suites of scientific workflows with diverse characteristics. These workflows can be executed on different serverless platforms. We comprehensively compare executing workflows on traditional local containers and serverless computing platforms. Our results show that serverless can reduce CPU and memory usage respectively by 78.11% and 73.92% without compromising performance.**

***Index Terms*—HPC Serverless Workflows, Serverless Computing, Scientific Workflows**

I. Introduction

In modern research, scientific workflows are critical in managing and orchestrating complex computational tasks across diverse domains [1]. These workflows, composed of interdependent tasks, enable scientists to process large volumes of data, automate analyses, and derive meaningful insights with high efficiency [2]. As science evolves, so does the technology supporting these workflows. Recently, there has been a notable shift towards leveraging serverless computing for scientific workflows, particularly in High-Performance Computing (HPC) environments. This shift is driven by serverless computing's potential to simplify management, reduce costs, enhance scalability, and enable on-demand resource provisioning. Examples of such applications include genomics data processing, large-scale simulation workflows, and real-time data analysis in environmental sciences [3], where serverless paradigms can significantly streamline execution and resource allocation while maintaining the computational rigor required by HPC workloads.

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, worldwide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

Given the importance of HPC serverless workflows, it is thus not surprising that this class of workflows has become a focal point for numerous research and development efforts [4]–[10]. These activities span a wide range of objectives, including the design of resource management and scheduling algorithms, the development of runtime systems capable of executing workflows across various hardware and software stacks, and the analysis of workflow configurations to identify commonalities and differences across scientific domains. However, most experimental evaluations of these workflows rely on real-world workflow instances running on production platforms to ensure their relevance to current application domains or testbeds to explore hypothetical and future scenarios.

Despite these advances, the existing literature on HPC serverless workflows lacks a comprehensive framework that supports and bridges the theoretical and practical aspects of workflow managers' research and development. WfCommons [11] has been a pivotal framework for the scientific workflow community, offering tools and data to cater to research and development needs by making real-world workflow instances accessible through a common format and generating realistic workflow benchmark specifications. However, WfCommons has been tailored primarily for traditional workflow managers and is currently not equipped to advance the research and development of serverless workflows, especially those tailored for HPC environments. This paper addresses this gap by proposing an extension of WfCommons to support serverless computing. Our approach includes adapting the WfCommons framework to generate serverless-compatible HPC workflow benchmarks and developing a new workflow manager specifically designed for HPC serverless environments. This comprehensive framework will enable more extensive research into the performance and applicability of serverless computing within the scientific workflow domain. Our framework is available in a GitHub repository[1] and detailed in the AD/AE appendix.

In this work, we make the following contributions:

- A framework that enables HPC scientific workflows on serverless computing, assembling a workflows manager for serverless and the WfCommons framework;
- A prototype of a workflow management system for serverless, evaluated using Knative, yet designed to cor-

[1]https://github.com/HewlettPackard/wfm-serverless-hpc

respond to any serverless platform that handles HTTP requests;
- Extension to the WfCommons Framework with Knative as a serverless platform;
- Extension to the WfCommons, proposing WfBench as a Service (one of the key components of WfCommons);
- An extensive evaluation, comparing serverless and bare-metal containers in terms of granularity, execution time, power, CPU, and memory usage.

II. Background and Related Work

A. HPC Workflows

Since the moment we built the first computers, scientist has been one of their main power users. Along the way, the complexity of both, the computational facilities and the calculations themselves have co-evolved substantially, giving rise to our modern HPC facilities and HPC workflows respectively. In this respect, the sophistication and variety of our current scientific workflows have been driven by a variety of factors and as such running these calculations efficiently presents ever-increasing challenges [12]. Notably, the convergence of disruptive technologies such as edge computing, big data analytics, and AI with traditional scientific calculations are rapidly becoming mainstream HPC workflows [1], [13]. Furthermore, identifying patterns in the growing set of HPC workflows is challenging, partly due to their diverse origins across specialized scientific communities. Consequently, the convergence execution management tools for these workflows seem at the moment unlikely although these are ongoing discussions within the workflow communities [14]. In this context, an open-source framework like WfCommons [11] with its capability to generate diverse types of workflow instances, strikes us as an adaptable research tool to explore the use of new technologies like serverless.

B. Workflow Management Systems

Workflow Management Systems (WMS) are used to automate the orchestration of scientific computations [15]. By managing task dependencies, data transfer, and resource allocation, WMS's simplify distributed computing, enabling scientists to concentrate on research rather than infrastructure. To capture the complexity of computational workflows, WFM often supports at least one of the many workflow definition languages in existence such as Common Workflow Language (CWL) [16] or Yet Another Workflow Language (YAWL) [17]. Characteristically, individual scientific communities have their own preferences for workflow languages; for instance, Nextflow is very popular for the design of bioinformatics pipelines [18]. While the use of domain-specific languages facilitates the development of workflows, it comes with the risk of vendor lock-in and reduces the interoperability between WFMs. On the other hand, some other WFMs have rejected the idea of using workflow languages and instead natively integrate programmable APIs for popular languages like Python, so that users can write workflows as code. Pegasus [19], is an example of this last type of WMS; this strategy, although powerful, requires additional programming knowledge and experience by the workflow developers. In the context of domain-specific workflow languages, little has been done to provide WFM with serverless capabilities. A notable exception is the specification for an Abstract Function Choreography language (AFCL) [20], a YAWL-like language for serverless applications. Yet, for WFM that has no support for AFCL the feasibility of executing traditionally serverful workflows under the serverless model is challenging. Other sets of challenges also arise for WFM that implement workflows as code. One might think that, pragmatically, the execution of serverless workflows might be easier without the involvement of WFM at all [21] or with the implementation of a light-weight serverless-aware WFM, which is the kind of strategy we follow here.

C. Serverless Functions and Platforms

Serverless computing has emerged as a novel paradigm in cloud and cluster computing, offering a dynamic way to allocate resources based on the specific requirements of individual compute functions. Traditionally, users of large-scale computing systems were limited to allocating resources in coarse-grained virtual machines (VMs). The introduction of serverless computing aimed to improve resource utilization by dynamically allocating the resources for a user's application at runtime. This model simplifies resource management and provides enhanced performance for short-lived functions, which are common in many HPC workflows. In the context of serverless computing, a function is defined as a stateless task that is executed when an event invoking the function enters the serverless framework.

Various platforms support serverless computing on large clusters, with Knative [22] being a popular example. Knative is a Kubernetes [23] based platform that enables serverless computing, allowing developers to focus on writing code without the need to manage complex infrastructure. This approach is particularly beneficial for HPC workflows, as it can efficiently handle the bursty and variable nature of many scientific computations. Serverless platforms can automatically scale resources up or down based on demand, potentially leading to cost savings and improved resource efficiency. Additionally, the event-driven nature of serverless computing aligns well with the structure of many scientific workflows, where one computation often triggers subsequent analyses or simulations.

D. HPC and Serverless

The Cloud Native Computer Foundation (CNCF), which is part of Linux Foundation is hosting the development of a community-driven open source specification for a Serverless Workflow Domain Specific Language (serverlessworkflow.io) [24]. Additionally, under the same umbrella, there is an ongoing effort to develop a workflow manager called Synapse that is fully compliant with this specification [25]. Although both projects are still under active development and it might be too early to predict their final acceptance, there will likely be some resistance to replacing current traditional

HPC solutions with their cloud-native counterparts. Ideally, both cloud-native and HPC solutions might merge into a better general solution or develop enough interoperability to coexist, exploiting the best of both environments. Other attempts to combine workflow managers with serverless technology, though less comprehensive, have also been proposed [7], [26]. Like the $serverlessworkflow.io + Synapse$ pairing, these initiatives have primarily focused on cloud solutions rather than HPC. In some cases like in Mashup [6], although targeting HPC, the serverless capabilities are limited to offload computations to the cloud in the form of serverless tasks when the computer capabilities of HPC clusters are stretched. Spillner et al. discuss the adoption of Cloud Computing in HPC and further explore how FaaS allows for a faster and cheaper way to run some HPC workloads [8]. Copik et al. explore the integration of High Performance Computing with serverless, they discuss the benefits of using serverless in the context of HPC like better compute and memory disaggregation and the ability for dynamic scheduling [5]. Malla et al., in their work, compare the performance of HPC workloads in the Cloud and a more traditional setup like IaaS. Their results show that specific HPC workloads that can take advantage of auto-scaling can subjectively be more performant in a Cloud-like setting [27].

III. A Framework to Study HPC Scientific Workflows on Serverless

In the pertinent literature, we can find several workflow management systems with relatively broad adoption such as Pegasus [19], NextFlow [18], or AirFlow [28]. However, none of them were designed for managing serverless-composed workflows. Knative, a well-known and validated container orchestrator built on Kubernetes, receives invocations for serverless functions, creates the necessary processes (pods), and executes them taking into account several other mechanisms such as resource-management, auto-scaling, fault-tolerance, etc. Knative is broadly known in the literature as well as other serverless platforms such as Globus Compute (previously FuncX [29]), OpenWhisk [30], OpenFaas [31], OpenShift [32] and OpenShift Serverless [33] (also based on Knative). We have chosen Knative as our core serverless platform because of its relatively simple installation, setup, and ease of use. By doing so, we hope that our results can be reproduced by others effortlessly.

Despite all the benefits, neither Knative nor the other serverless platforms offer mechanisms to manage workflow invocations, as they are all stateless and meant for single invocations. Therefore, we developed a prototype workflow manager for serverless environments. Although we used Knative as the initial target for our experiments, our workflow manager is designed to work with any serverless platform that handles invocations through HTTP requests. For the first version of our prototype, we assume that all machines in the cluster have access to a common shared directory for storing I/O. With that, we enforce that all functions in the workflows can write to and read from the same place, hence the communication between the different functions and steps is guaranteed.

To compensate for the lack of HPC scientific workflows for serverless, we rely on WfCommons, a framework that generates realistic synthetic workflows. WfCommons can create and translate workflows to fit the requirements of several workflow managers such as Pegasus and NextFlow. In this work, we extend WfCommons by proposing a new Translator, for Knative. With that, we can produce several different scientific workflows and have all of them ready to be executed on serverless, using Knative.

Finally, we propose a framework illustrated in Figure 1, composed of 4 main components. By combining our workflow manager (component 3), which is capable of executing workflows on any serverless platform that handles HTTP requests from any cluster (component 4), with our contribution to WfCommons (component 1), we can perform an extensive set of experiments with scientific workflows on serverless platforms (component 2). In these experiments, we vary the size of the workflows, the amount of CPU each function should stress, the choice of keeping or not keeping memory allocated over the execution of the functions, and the granularity of the serverless processes. In the remainder of this section, we detail each of these components.

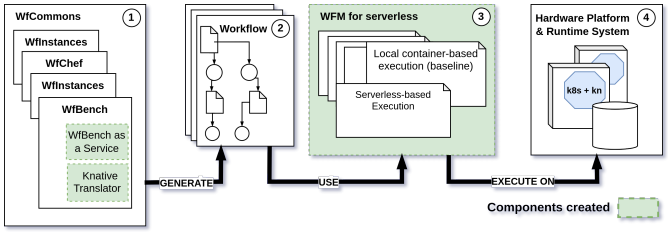

Fig. 1: Architecture of our framework for managing HPC scientific workflows on serverless.

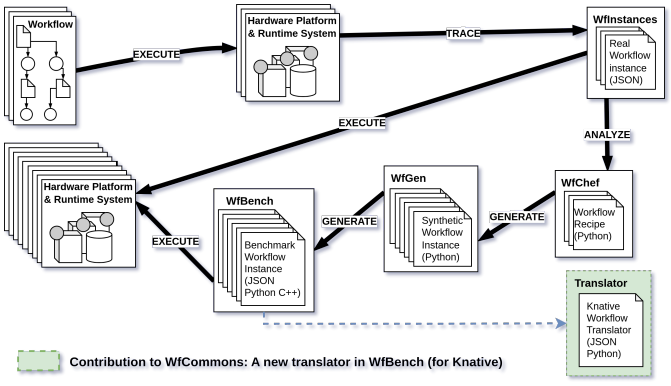

Fig. 2: WfCommons architecture and our contribution.

A. Extending WfCommons

WfCommons is a framework that comprises four main components: (1) WfInstances: gathers different scientific workflows and groups them by type; (2) WfChef: uses the groups of

workflow instances to generate recipes of scientific workflows for that type; (3) WfGen: uses the recipes to generate workflows; and (4) WfBench: creates benchmarks after the workflows are generated. In particular, WfBench has a component called *Translator* that converts a generated workflow that is written in the WfCommons pattern to different workflow managers [34]. Currently, WfCommons supports Translators for Pegasus and NextFlow. We propose a new Translator, targeting Knative. Figure 2 details WfCommon's main components and how they interact. Specifically, the figure highlights our extension to the framework. In addition, the excerpt of code below illustrates the output of our Knative Translator after generating workflows with WfCommons. Lines 7-14 and line 20 represent respectively two of our main modifications. The first modification converts the entry "arguments" from a list of parameters to a sub-entry with key-values, making it easier to build the HTTP request command in our workflow manager for serverless. The second modification includes the HTTP request endpoint of that specific function on the serverless platform.

```
{
"blastall_00000002": {
 "name": "blastall_00000002",
 "type": "compute",
 "command": {
  "program": "wfbench.py",
  "arguments": [
   {
    "name": "blastall_00000002",
    "percent-cpu": 0.9,
    "cpu-work": 100,
    "out": {
     "blastall_00000002_output.txt": 40161
    },
    "inputs": [
     "split_fasta_00000001_output.txt"
    ]
   }
  ],
  "api_url": "http://wfbench.knative-functions
     ↪ .00.000.000.000.sslip.io/wfbench"
 },
 "parents": [
  "split_fasta_00000001"
 ],
 "children": [
  "cat_blast_00000042",
  "cat_00000043"
 ],
 "files": [
  {
   "link": "output",
   "name": "blastall_00000002_output.txt",
   "sizeInBytes": 40161
  },
  {
   "link": "input",
   "name": "split_fasta_00000001_output.txt",
   "sizeInBytes": 40161
  }
 ],
 "runtimeInSeconds": 0,
 "cores": 1,
 "id": "00000002",
 "category": "blastall",
 "startedAt": "2024-07-12T17:09:21.522439+02:
     ↪ 00"
},
}
```

B. WfBench as a Service

One of the key aspects of our integration of WfCommons to a serverless framework is to turn WfBench [34] into a service. Through WfBench, the WfCommons framework offers an executable, coded in Python, that performs real computation for each function of the synthetic workflows generated, respecting their parameters of stressing CPU, memory, and producing and using I/O. This application was designed to be executed on bare metal, but we containerized it, and then deployed it on Knative, turning it into a service on a serverless platform. Afterward, WfBench executable responds to an HTTP request, where we send a POST request containing all the necessary parameters, including the *name* of the function, *CPU stress* percentage, *CPU workload*, *input* and *output* data, and the *directory* were I/O operations should be recorded. The request structure appears as follows:

```
curl localhost:8080/wfbench -X POST -H '
    ↪ Content-Type: application/json' -d '{"
    ↪ name":"split_fasta_00000001", "percent-
    ↪ cpu":0.6, "cpu-work":100, "out":{"
    ↪ split_fasta_00000001_output.txt": 204082
    ↪ }, "inputs":["split_fasta_00000001_input
    ↪ .txt"], "workdir":"../data/wfbench-
    ↪ knative"}'
```

C. Building a Serverless Workflow Manager

Our serverless workflow manager processes input in the form of a workflow description formatted as a JSON file. This file is structured according to the WfCommons pattern, where each entry represents a single function. Each function is linked to its predecessor (parent) and successor (child) functions, along with their associated input and output files. The key innovation in adapting WfCommons for serverless environments involves enhancing the workflow description to include a new entry for each function. This entry specifies the HTTP request required to execute the function using the WfBench service. Additionally, we introduced a header (starting function) at the beginning of the workflow and a tail (finishing function) at the end. These additions streamline the management of various workflow types, ensuring a more generic and flexible execution process. Upon invocation, the workflow is translated into a Directed Acyclic Graph (DAG). For each step in the DAG, all associated functions are collected and simultaneously executed by sending HTTP requests to their respective addresses.

In its initial version, our workflow manager assumes the presence of a shared drive within the serverless cluster to handle the input and output data exchanged between different

functions and phases. Before invoking each function, the workflow manager checks whether the required input files are available on the shared drive. This ensures they were generated by preceding functions. To guarantee the correct sequencing, a brief delay of one second is introduced between each workflow phase, allowing sufficient time for the preceding functions to complete and write the expected files to the shared drive. These outputs then serve as inputs for the subsequent functions.

D. A Bare-metal Local Container Baseline

Since standard HPC workflows are typically executed on workflow managers deployed on bare-metal platforms, we adopt this approach as our baseline computational paradigm. Consequently, our serverless workflow manager, as described above, also supports the execution of scientific workflows using a bare-metal local container setup. In this configuration, we assume the presence of a shared drive within the local cluster, and the workflow is managed as a DAG of functions executed in phases. The key distinction is that, instead of relying on the WfBench service deployed on Knative, we utilize a local container that hosts the same version of the WfBench application, allowing the workflows to be executed entirely within the local environment.

IV. Methodology

To address the lack of available scientific workflow instances and workflow managers for serverless, we propose and evaluate a framework that comprises both (see Section III). Using a workflow generator (see Section III-B), we create different types of synthetic workflows. Additionally, we extended this tool to convert the generated workflows to serverless platforms (see Section III-A). In doing so, we became equipped to study the different behaviors of the now *serverless workflows*. Particularly, the types of workflows under consideration differ not only in their number of functions, but also in other characteristics such as being burst or spread, and having many or few phases. To properly execute these workflows on top of serverless platforms, we use our first prototype of a workflow manager for serverless (see Section III-C). We emphasize that this approach is compatible with any serverless platform that uses HTTP requests for function invocation. Similarly, our baseline (see III-D) is applicable to any bare-metal platform capable of running Docker [35] containers.

To evaluate our framework and show its versatility, we performed a series of experiments following the design shown in Table I. Specifically, we evaluate two computational platforms: serverless computing and local bare-metal containers. We compare both platforms in terms of execution time, power consumption, CPU, and memory usage. We also evaluate different granularities for the computational processes, as well as the number of inner workers and threads of their execution. Finally, we vary the size of the workflows and the configurations of the auto-scaling mechanisms for the serverless setups. In total we perform 140 experiments, from where: a) 98 experiments are focused on fine-grained scenarios, varying 7 computational paradigms, 7 workflows and 2 sizes of workflows; b) 42 experiments are focused on coarse-grained scenarios, varying 2 computational paradigms, 7 workflows and three sizes of workflows. The computational paradigms are detailed on Table II.

TABLE I: Design of Experiments.

Parameter	Value
Platform	Bare-metal local containers, Knative
Workflows	Blast, Bwa, Cycles, Epigenomics, Genomes, Seismology, Srasearch
Workflow Sizes	250, 500, 1000 tasks
CPU stressing intensity	100%
Number of workers	1, 10 workers
Functions' granularity	Coarse-grained, fine-grained
Persistent Memory	With, without
Computational Paradigms	Kn1wPM, Kn1wNoPM, Kn10wNoPM, Kn1000wPM, LC1wPM, LC1wNoPM, LC10wNoPM , LC10wNoPmNoCR, LC1000wPM
Total	140 experiments

V. Experimental Evaluation

To ensure a more structured analysis and discussion, we have organized our experimental results into four groups:

A. An initial **workflow characterization**, where we aggregate different aspects of workflows, including their DAG structures, phase/step compositions, and their different functions.

B. An analysis of the **impact of different approaches and setups with fixed computational paradigms**, focusing on parameters such as the number of workers managing simultaneous parallel invocations, the use of persistent memory, and pre-allocated resources.

C. A study of the effectiveness of **coarse-grained approaches** within different computational paradigms.

D. A comparison between the performance of workflows executed in **serverless environments and local containers**.

To represent the computational paradigms presented in them, all the figures use the nomenclature detailed in Table II.

A. Workflow Characterization

WfCommons workflows represent a large set of real-world workflow instances from diverse application domains based on their executions using diverse workflow systems. WfCommons instances are derived from execution logs of real-world workflow applications. They represent domains such as Bioinformatics, Agroecosystems, Seismology and Astronomy. Detailed classification can be found in the WfInstances repository on GitHub [36]. They represent a good fraction of HPC scientific workflows. Additional workflows with similar structures but with different requirements could be generated using WfBench [34].

TABLE II: Nomenclature for the computation paradigms used in the experimental evaluation.

Computational Paradigm	Description
Kn1wPM	**Kn**ative, with **1 w**orker per process (pod), and **P**ersistent **M**emory over the functions
Kn1wNoPM	**Kn**ative, with **1 w**orker per process (pod), and **No P**ersistent **M**emory over the functions
Kn10wNoPM	**Kn**ative, with **10 w**orker per process (pod), and **No P**ersistent **M**emory over the functions
Kn1000wPM	**Kn**ative, with **1000 w**orker per process (pod), and **P**ersistent **M**emory over the functions (only coarse-grained for Kn)
LC1wPM	**L**ocal **C**ontainers, with **1 w**orker per process (container), and **P**ersistent **M**emory over the functions
LC1wNoPM	**L**ocal **C**ontainers, with **1 w**orker per process (container), and **NoP**ersistent **M**emory over the functions
LC10wNoPM	**L**ocal **C**ontainers, with **10 w**orker per process (container), and **NoP**ersistent **M**emory over the functions
LC10wNoPMNoCR	**L**ocal **C**ontainers, with **10 w**orker per process (container), **No P**ersistent **M**emory over the functions, and **No C**PU **R**equirement
LC1000wPM	**L**ocal **C**ontainers, with **1000 w**orker per process (container) and **P**ersistent **M**emory over the functions (only coarse-grained for LC)

Using our extension to WfCommons, we generate and translate seven HPC scientific workflows, namely Blast, BWA, Cycles, Epigenomics, Genomes, Seismology, and Srasearch. Figure 3 illustrates the composition and structure of Blast, BWA, Cycles, and Epigenomics workflows. The first subfigure depicts the overall workflow, the second shows the distribution of functions across the workflow phases, and the third shows the number of functions categorized by their specific types. The two first workflows, Blast and BWA, are more dense, featuring fewer steps but a high concentration of functions executed simultaneously. In contrast, Cycles and Epigenomics have a more complex structure, as evident in their visualization, with varying numbers of steps and a broader diversity of function types.

B. Impact of Fixed Paradigm Configurations

Figures 4 and 5 show the various workflows and metrics across multiple dimensions. The x-axis presents different setups of computational paradigms, while the y-axis represents the corresponding metrics, and the colors differentiate workflow sizes. Both figures emphasize the Blast and Epigenomics workflows, as they exemplify the two main behaviors identified across the different workflows. Blast shares similarities with BWA, Genome, Seismology, and SraSearch workflows, while Epigenomics is comparable to the Cycles workflow.

Figure 4 shows the different configurations for the serverless paradigm using Knative. The three setups vary by the number of workers (1w or 10w) and whether persistent memory is utilized (PM or NoPM). The results suggest that using 10 workers (10w) without persistent memory (NoPM) slightly improves execution time, power, and memory usage, though it does not significantly impact CPU usage. This outcome is expected since 10 workers per container within Knative allow for more parallel function execution. Overall, despite the less optimal CPU usage, the trade-offs across the different metrics indicate that the *10wNoPM* setup provides the most balanced performance. In addition, this is the closest scenario to real production platforms, with more than one worker per process. Therefore, this is the preferred configuration for the subsequent comparisons.

Figure 5 shows the different setups for the local container (LC) paradigm performed on bare metal. The four setups vary the number of workers (1w or 10w), whether to use persistent memory (PM or NoPM), and the option to specify in advance the number of cores and memory to be used by the containers (NoCR). Not surprisingly, the results suggest that using 10 workers (10w) without persistent memory (NoPM) and no CPU request (NoCR) slightly improves power efficiency and CPU usage, though it does not enhance execution time or reduce memory usage. This outcome is expected because when the container's memory requirements are specified in advance, the container enforces a hard limit, but without such constraints, it may consume more memory, as observed in this case.

C. Coarse-grained for Serverless

Figure 6 displays various workflows and metrics across multiple facets. The colors represent the computational paradigms (Knative and local containers, respectively), while the x-axis presents workflow sizes. The y-axis shows the measured metrics. The figure shows the performance of serverless versus local containers for a coarse-grained scenario. In this scenario, we had one unique process in serverless that specifically reserved all the resources on the machine. Therefore, there is no cold-start delay involved, nor scaling of the computational process. This scenario shows that with coarse-grained reservations, even for serverless, we can manage bigger applications and workflows composed of more functions (e.g 1000 functions) in an easier way. Because of auto-scaling the previous fine-grained scenarios did not conclude their execution without reaching memory and CPU limits. Also, it is possible to see in the first column of the figure that serverless can be close to or even faster than the local container approach. However, by reserving the entire machine in advance for one unique process, serverless loses in terms of resource utilization, having similar or worse performance in terms of power, memory, and CPU usage.

D. Serverless versus Local Containers

We select the *10wNoPM* computational paradigm as the best one for serverless, which despite the less optimal CPU usage, presents good trade-offs across the different metrics indicating the most balanced performance (see Figure 4). In addition, this is the closest scenario to real production platforms, with

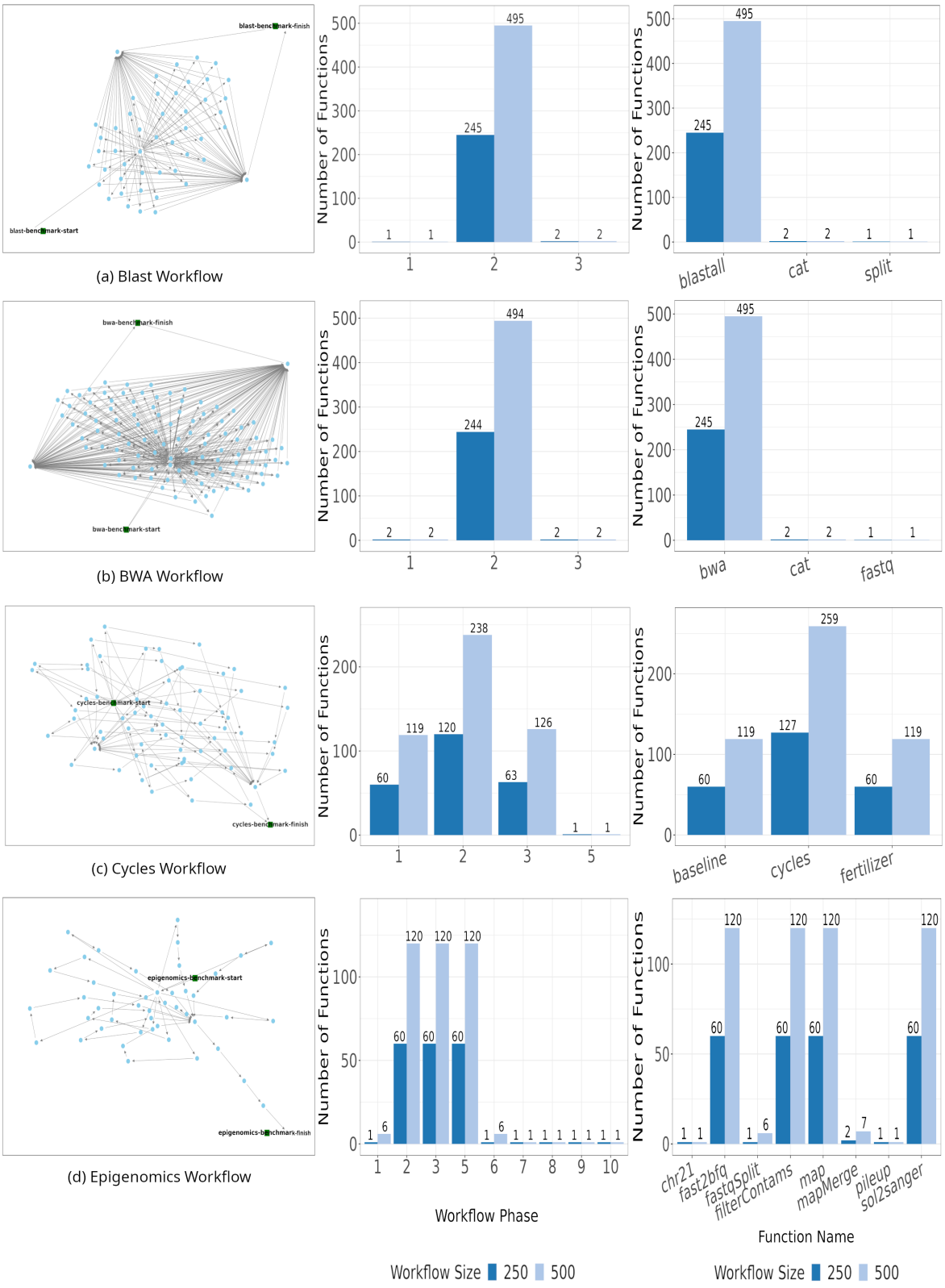

Fig. 3: Different workflows, its phase density in number of in functions, and its composition in function's name and quantity.

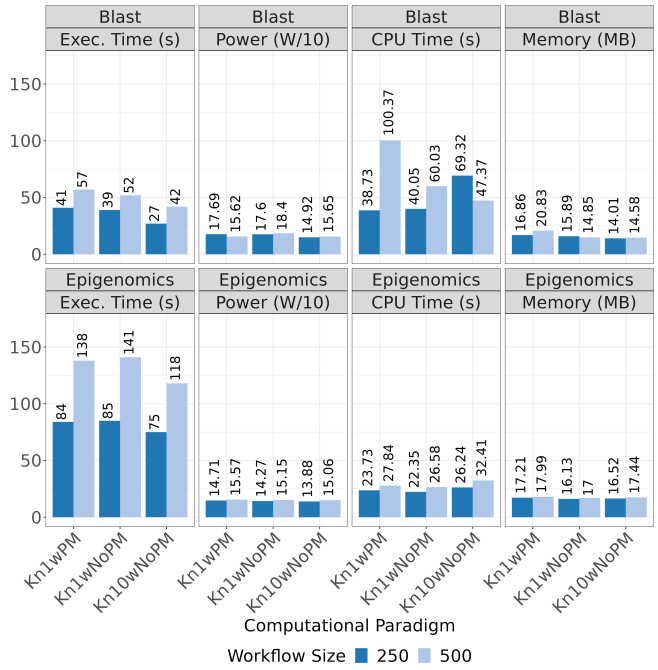

Fig. 4: Comparison between different setups for the Serverless Computational Paradigm.

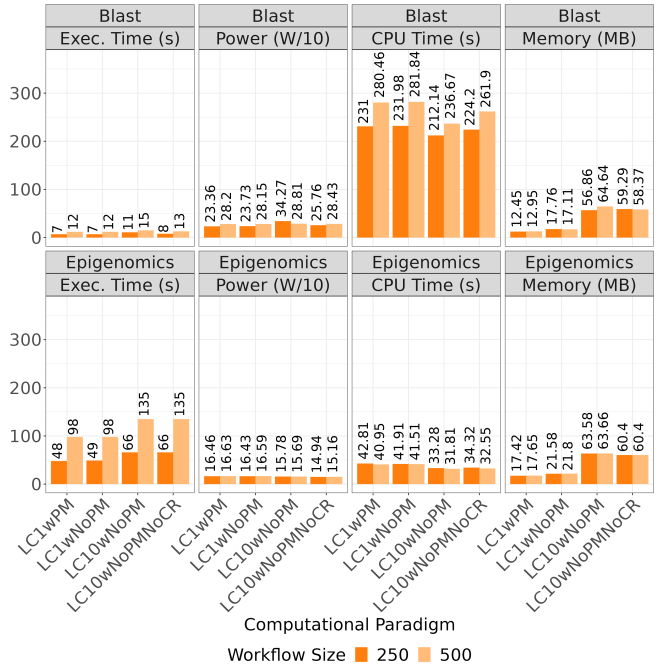

Fig. 5: Comparison between different setups for the Local Containers Computational Paradigm.

more than one worker per process. From the local container perspectives, the directly comparable computational paradigms is the *LC10wNoPM* (see Figure 5).

Figure 7 shows an overview of the various workflows and metrics across multiple facets. The x-axis presents the different workflow sizes, against the corresponding metrics on the y-

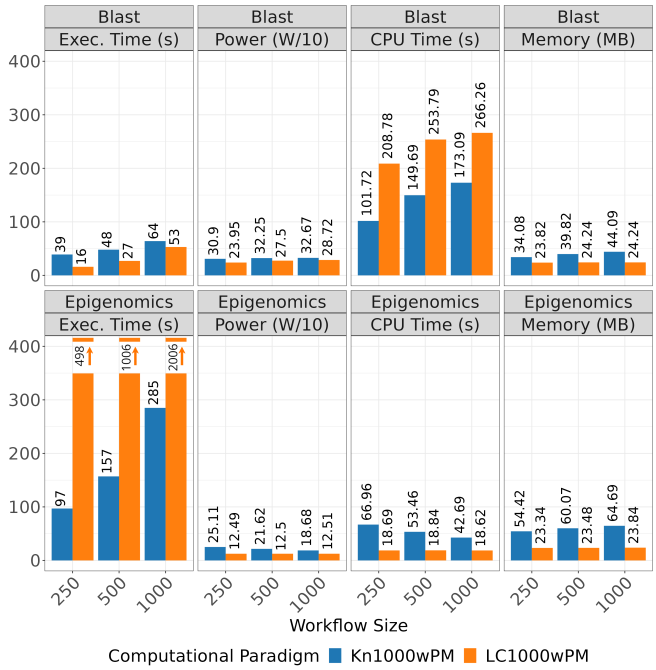

Fig. 6: Comparison for coarse-grained granularity between the Serverless and the Local Container Computational Paradigm.

axis, while the colors represent the computational paradigms (respectively *10wNoPM* for knative and *LC10wNoPM* for local containers). This figure shows that by developing a workflow manager for serverless and extending WfCommons to translate their generated workflows for execution on serverless platforms, our approach enables researchers to study the performance of different types of scientific workflows across various metrics. Additionally, our setup allows for the combination of multiple parameters. WfCommons facilitates the adjustment of CPU intensity and I/O operations for individual functions. With the Knative setup, we can explore the dynamics of cold/warm containers and the number of accepted requests. Finally, using Docker, we can control the number of workers per container to manage requests per core.

Upon further analysis, the workflows can be categorized into two main groups based on their behaviors: (1) **First group**: Blast, BWA, Genome, Seismology and SraSearch workflows exhibit longer execution time on serverless platforms compared to local containers, as expected; (2) **Second group:** the Cycles and Epigenomics workflows surprisingly present a different pattern where local containers generally outperform serverless in terms of execution. However, the performance gap is narrower for this group, especially when managing workflows containing a higher number of functions. Despite this, serverless platforms demonstrate advantages across other metrics, such as power consumption, where they match local containers while reducing CPU usage by up to 78.11% and memory usage by up to 73.92% during workflow execution.

The most significant gains are observed in the first group of workflows, suggesting that scientific workflows with dense

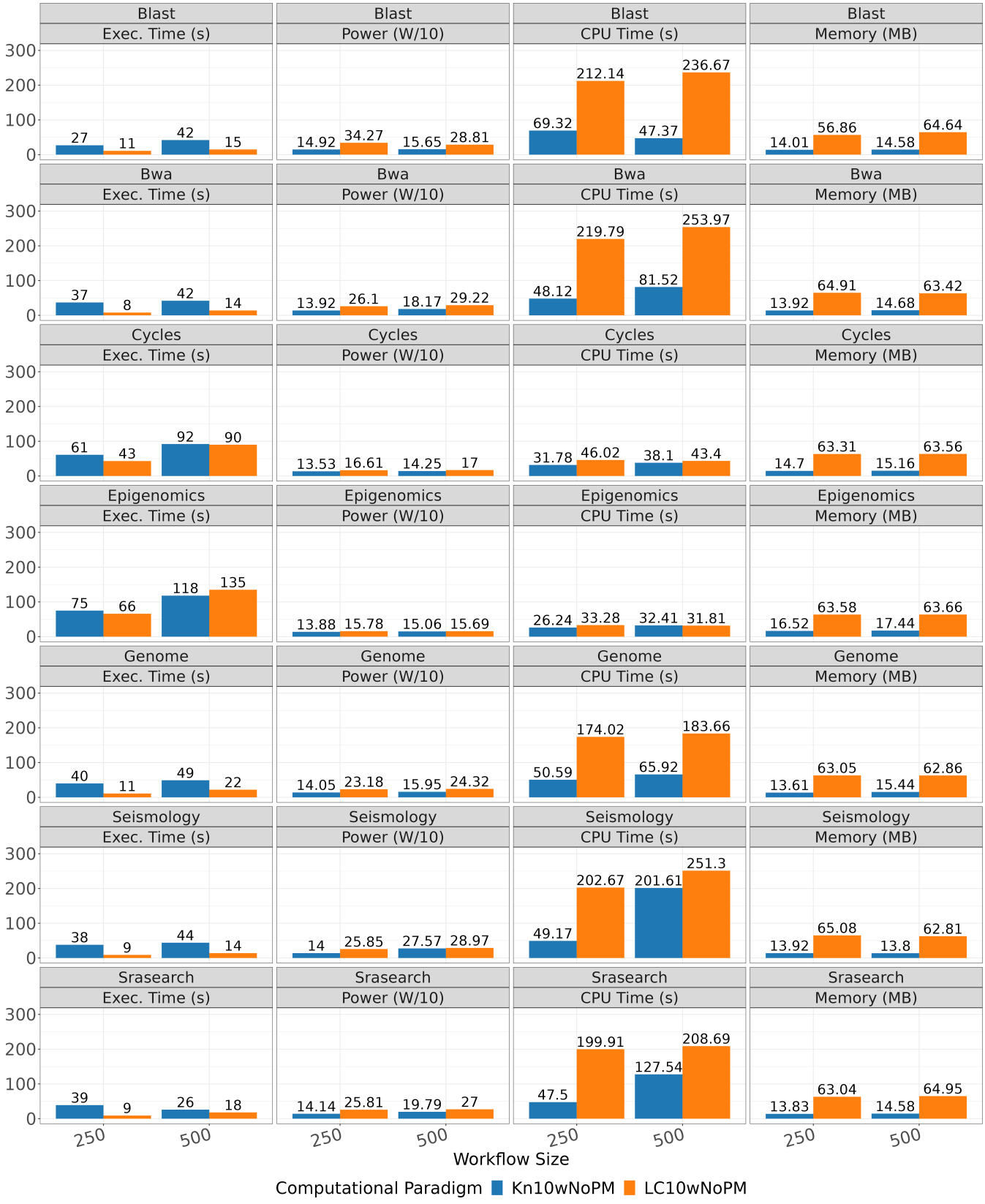

Fig. 7: Comparison between best setups for Serverless and Local Container Computational Paradigms.

steps, where many of the identical functions are invoked simultaneously using the same processes on serverless platforms, can reduce resource usage while maintaining execution performance comparable to local containers. Therefore, we emphasize the effectiveness of this paradigm for such workflows.

Although the second group of workflows does not show clear benefits from serverless execution, it's important to note that workflows often consist of different steps and types of functions. We believe that these workflows could benefit from serverless execution when combined with those from the first group, either by invoking them concurrently or by merging their functions. Furthermore, we argue that complex workflows may gain the most advantage from a hybrid approach, leveraging a combination of both computational paradigms, if applied strategically to different steps within the workflows.

VI. Conclusion

In this work, we propose a framework for executing and evaluating HPC scientific workflows on serverless platforms. Our approach combines (1) a serverless workflow manager and (2) an extension of the WfCommons framework to translate traditional workflows into serverless-compatible versions. By doing so, we empower researchers with new tools to study the performance of various HPC scientific workflows across multiple metrics. Our setup offers flexibility in configuring different parameters, such as modifying CPU intensity and I/O operations per task through WfCommons, exploring container dynamics like cold/warm startups and the number of accepted requests via Knative, and managing the number of workers per CPU core using Docker.

Our extensive experiments demonstrate that HPC scientific workflows can significantly benefit from serverless in terms of resource efficiency (CPU, memory, and power) while maintaining performance levels close to traditional execution times. Yet, not all evaluated workflows showed these benefits uniformly when executed on serverless platforms. We should recall that workflows can be composed of different steps and types of functions, not all of them are necessarily ideal for serverless execution. Therefore, it is likely that in some cases, a mapping of different execution paradigms per sub-workflow might be a better choice. Namely, the optimal strategy for complex workflows might be combining executions on serverless and bare-metal local containers for different tasks or groups of tasks.

We also evaluate coarse-grained scenarios for serverless. In that case, we understand that the management of CPU and memory is simpler than in fine-grained scenarios, as these resources are reserved in advance. The consequence is that the results from serverless are closer to the bare-metal local containers; however, the resource usage is not optimal for serverless as it is when using fine-grained resources. We understand that we can evaluate bigger workflows on coarse-grained scenarios. Still, with fine-grained resources and auto-scaling enabled, managing resources such as CPU and memory is more challenging. The auto-scaling configuration enables us to create new processes (pods) in advance whenever a specific load is achieved on the existing processes. However, between creating these new warm processes and invoking new functions, some functions executed on the older processes may be finished, enabling them to receive the newly submitted functions, making the new processes either empty or underutilized. In this case, more resources are used, and limits of memory and CPU may be reached. For that reason, bigger workflows were successfully executed on coarse-grained scenarios in our small setup. The experiments were not concluded for all the tests due to memory and CPU limits being reached.

We conclude that our framework and investigation shed light on how researchers can perform studies on HPC scientific workflows using serverless computing. We show that our framework is very flexible, enabling the usage of an already well-validated tool, WfCommon. Our workflow management system can be used on top of any serverless platform that uses HTTP requests for invoking functions. Therefore, we believe these results provide an important step toward the characterization of scientific, HPC-based serverless workflows

VII. Future Work

We intend to leverage this study and include more aspects of serverless, either by investigating other mechanisms of the current chosen platform or including new ones in the WfCommons' list of targets. For instance, we intend to investigate the impacts of using external distributed data storage for managing scientific workflows. We also plan to study the impacts of serverless on multi-cluster invocation scenarios. We believe that fine-grained resource management and the auto-scaling mechanism of serverless can improve even more aspects such as resource usage, when we consider the invocation of multiple concurrent functions by different workflows. In addition to Knative, we will explore other serverless platforms, such as Globus Compute [29], and those offered by Cloud providers (AWS Lambda, Google Cloud Functions, and Azure). Finally, we expect that all these directions can lead us to the characterization of HPC scientific workflows on serverless, being able to select the best types of functions for the different computational paradigms, and merging them with serverless computing.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] R. M. Badia Sala, E. Ayguadé Parra, and J. J. Labarta Mancho, "Workflows for science: A challenge when facing the convergence of HPC and big data," *Supercomputing frontiers and innovations*, vol. 4, no. 1, 2017.

[2] R. Ferreira da Silva, R. M. Badia, D. Bard, I. T. Foster, S. Jha, and F. Suter, "Frontiers in Scientific Workflows: Pervasive Integration with HPC," *IEEE Computer*, vol. 57, no. 8, pp. 36–44, 2024.

[3] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, "Serverless workflows for containerised applications in the cloud continuum," *Journal of Grid Computing*, vol. 19, pp. 1–18, 2021.

[4] R. Farahani, F. Loh, D. Roman, and R. Prodan, "Serverless Workflow Management on the Computing Continuum: A Mini-Survey," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 146–150.

[5] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler, "Software resource disaggregation for hpc with serverless computing," *arXiv preprint arXiv:2401.10852*, 2024.

[6] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–60. [Online]. Available: https://doi.org/10.1145/3503221.3508407

[7] A. John, K. Ausmees, K. Muenzen, C. Kuhn, and A. Tan, "Sweep: Accelerating scientific research through scalable serverless workflows," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '19 Companion. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–50. [Online]. Available: https://doi.org/10.1145/3368235.3368839

[8] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *High Performance Computing*, E. Mocskos and S. Nesmachnow, Eds. Cham: Springer International Publishing, 2018, pp. 154–168.

[9] P. Bruel, S. R. Chalamalasetti, A. Dhakal, E. Frachtenberg, N. Hogade, R. P. H. Enriquez, A. Mishra, D. Milojicic, P. Prakash, and G. Rattihalli, "Predicting heterogeneity and serverless principles of converged high-performance computing, artificial intelligence, and workflows," *Computer*, vol. 57, no. 1, pp. 136–144, 2024.

[10] T. Pfandzelter, A. Dhakal, E. Frachtenberg, S. R. Chalamalasetti, D. Emmot, N. Hogade, R. P. H. Enriquez, G. Rattihalli, D. Bermbach, and D. Milojicic, "Kernel-as-a-service: A serverless programming model for heterogeneous hardware accelerators," in *Proceedings of the 24th International Middleware Conference*, ser. Middleware '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 192–206. [Online]. Available: https://doi.org/10.1145/3590140.3629115

[11] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. Ferreira da Silva, "WfCommons: A Framework for Enabling Scientific Workflow Research and Development," *Future Generation Computer Systems*, vol. 128, pp. 16–27, 2022.

[12] T. Ben-Nun, T. Gamblin, D. S. Hollman, H. Krishnan, and C. J. Newburn, "Workflows are the new applications: Challenges in performance, portability, and productivity," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 57–69.

[13] J. Ejarque, R. M. Badia, L. Albertin, G. Aloisio, E. Baglione, Y. Becerra, S. Boschert, J. R. Berlin, A. D'Anca, D. Elia, F. Exertier, S. Fiore, J. Flich, A. Folch, S. J. Gibbons, N. Koldunov, F. Lordan, S. Lorito, F. Løvholt, J. Macías, F. Marozzo, A. Michelini, M. Monterrubio-Velasco, M. Pienkowska, J. de la Puente, A. Queralt, E. S. Quintana-Ortí, J. E. Rodríguez, F. Romano, R. Rossi, J. Rybicki, M. Kupczyk, J. Selva, D. Talia, R. Tonini, P. Trunfio, and M. Volpe, "Enabling dynamic and intelligent workflows for hpc, data analytics, and ai convergence," *Future Generation Computer Systems*, vol. 134, pp. 414–429, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22001364

[14] R. Ferreira da Silva, H. Casanova, K. Chard, I. Altintas, R. M. Badia, B. Balis, T. Coleman, F. Coppens, F. Di Natale, B. Enders, T. Fahringer, R. Filgueira, G. Fursin, D. Garijo, C. Goble, D. Howell, S. Jha, D. S. Katz, D. Laney, U. Leser, M. Malawski, K. Mehta, L. Pottier, J. Ozik, J. L. Peterson, L. Ramakrishnan, S. Soiland-Reyes, D. Thain, and M. Wolf, "A community roadmap for scientific workflows research and development," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2021, pp. 81–90.

[15] C. Schmitt, B. Yu, and T. Kuhr, "A workflow management system guide," 2023. [Online]. Available: https://arxiv.org/abs/2212.01422

[16] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and T. C. Community, "Methods included: standardizing computational reuse and portability with the common workflow language," *Commun. ACM*, vol. 65, no. 6, p. 54–63, may 2022. [Online]. Available: https://doi.org/10.1145/3486897

[17] W. van der Aalst and A. ter Hofstede, "Yawl: yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245–275, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437904000304

[18] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, Apr 2017. [Online]. Available: https://doi.org/10.1038/nbt.3820

[19] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. Ferreira da Silva, G. Papadimitriou, and M. Livny, "The evolution of the pegasus workflow management software," *Computing in Science & Engineering*, vol. 21, no. 4, pp. 22–36, 2019.

[20] S. Ristov, S. Pedratscher, and T. Fahringer, "Afcl: An abstract function choreography language for serverless workflow specification," *Future Generation Computer Systems*, vol. 114, pp. 368–382, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X20302648

[21] A. Elshamy, A. Alquraan, and S. Al-Kiswany, "A study of orchestration approaches for scientific workflows in serverless computing," in *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, ser. SESAME '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 34–40. [Online]. Available: https://doi.org/10.1145/3592533.3592809

[22] "Knative: Kubernetes-based platform to build, deploy, and manage modern serverless workloads." 2014, https://github.com/knative [Accessed: 05.08.2024].

[23] "Kubernetes," 2014, https://kubernetes.io/ [Accessed: 05.08.2024].

[24] "serverless.io - a vendor-neutral, open-source, and entirely community-driven ecosystem tailored for defining and executing dsl-based workflows in the realm of serverless technology." 2020, https://serverlessworkflow.io/ [Accessed: 08.08.2024].

[25] "Synapse - a vendor-neutral, free, open-source, and community-driven workflow management system (wfms) implementing the serverless workflow specification." 2020, https://github.com/serverlessworkflow/synapse [Accessed: 08.08.2024].

[26] A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," *Future Generation Computer Systems*, vol. 124, pp. 215–229, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21001989

[27] S. Malla and K. Christensen, "Hpc in the cloud: Performance comparison of function as a service (faas) vs infrastructure as a service (iaas)," *Internet Technology Letters*, vol. 3, no. 1, p. e137, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/itl2.137

[28] "Airflow - a platform created by the community to programmatically author, schedule and monitor workflows." 2020, https://airflow.apache.org/ [Accessed: 14.08.2024].

[29] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funcx: A federated function serving fabric for science," in *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, 2020, pp. 65–76.

[30] "Openwhisk - an open source, distributed serverless platform that executes functions (fx) in response to events at any scale.." 2024, https://openwhisk.apache.org/ [Accessed: 16.08.2024].

[31] D.-N. Le, S. Pal, and P. K. Pattnaik, "Openfaas," *Cloud computing solutions: architecture, data storage, implementation and security*, pp. 287–303, 2022.

[32] "Openshift container platform - a consistent hybrid cloud foundation for building and scaling containerized applications." 2024, https://docs.openshift.com/container-platform/4.16/welcome/index.html [Accessed: 16.08.2024].

[33] "Openshift serverless- kubernetes native building blocks that enable developers to create and deploy serverless, event-driven applications on openshift container platform." 2024, https://docs.openshift.com/serverless/ [Accessed: 16.08.2024].

[34] T. Coleman, H. Casanova, K. Maheshwari, L. Pottier, S. R. Wilkinson, J. Wozniak, F. Suter, M. Shankar, and R. F. Da Silva, "Wfbench: Automated generation of scientific workflow benchmarks," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2022, pp. 100–111.

[35] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, mar 2014.

[36] "Wfinstances: Collection and curation of open access production workflow executions from various scientific applications." 2021, https://github.com/wfcommons/wfinstances [Accessed: 05.08.2024].

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 A framework that enables HPC scientific workflows on serverless computing, assembling a workflow manager for serverless and the WfCommons framework;
C_2 A prototype of a workflow management system for serverless, evaluated using Knative, yet designed to correspond to any serverless platform that handles HTTP requests;
C_3 Extension to the WfCommons Framework with Knative as a serverless platform;
C_3 Extension to the WfCommons, proposing WfBench as a Service;
C_4 An extensive evaluation, comparing serverless and bare-metal containers in terms of granularity, execution time, power, CPU, and memory usage.

B. Computational Artifacts

A_1 https://github.com/HewlettPackard/wfm-serverless-hpc/tree/main/wfbench
A_2 https://github.com/HewlettPackard/wfm-serverless-hpc

Artifact ID	Contributions Supported	Related Paper Elements
A_1, A_2	$C_1 - C_5$	Figures 3-7 Table 2

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This is our contribution to WfCommons, which consists on creating a new Translator component, and the deployment of one of its modules as a service. They are included in our repository, organized as following:

- /wfbench/knative-translator/:
  - **knative.py**: this is our Knative translator component, designed to translate WfCommons workflows to be executed on the Knative serverless platform.
  - **generate_workflows_example.py**: this script examplifies how to import our Knative Translator module, and how to generate and translate workflows.
- /wfbench/services/:
  - **wfbench**: this folder contains the source-code that packs wfbench as a service to be deployed on Knative.
  - **wfbench-local**: this folder contains the source-code that packs wfbench as a local container.

Expected Results

The components are both used in combination with Artifact A_2 for the production of the results of the paper. WfCommon is now able to translate their generate workflows to the Knative serverless platform, and developers and scientists are equipped to use *WfBench as a Service*.

Expected Reproduction Time (in Minutes)

Artifact Setup: The expected computational time for deploying the setup, to install all dependencies, setting up the cluster and the serverless platform, is about 60 minutes. **Artifact Execution:** The expected computational time of this artifact depends mostly on the amount of workflows generated and their sizes. In our case, the total time, considering all workflows and all combination of different scenarios is 5 minutes. **Artifact Analysis:** The analysis process for this artifact is actually the testing of our scripts and the validation of the deployment of all components, on serverless or locally. It takes in order of 10 minutes.

Artifact Setup (incl. Inputs)

Hardware: There is no hardware requirements.

Software: The following programs, libraries and frameworks must be installed:

- Git: https://github.com/git-guides/install-git
- Kubernetes: https://kubernetes.io/docs/tasks/tools/
- Knative: https://knative.dev/docs/install/
- Python3: https://www.python.org/downloads/
- Workflow Commons (WfCommons): https://docs.wfcommons.org/en/latest/quickstart_installation.html
- Performance Co-Pilot (PCP): https://pcp.readthedocs.io/en/latest/QG/QuickReferenceGuide.html#installation

Datasets / Inputs: There are no datasets required for the execution of this artifact.

Installation and Deployment: Other than the software requirements, it is necessary to clone ours and the WfCommons GitHub repositories, as well as to install the Python libraries used by our framework.

Artifact Execution

This artifact's workflow consists of two tasks T_1 and T_2, that interact as T_1, T_2 (there is no dependcy between them). Task T_1 may deploy WfBench as a Service in the Knative serverless platform, and locally as a bare-metal local container. Task T_2 may clone the WfCommon repository, compile our Knative Translator component and make sure that it is working properly, using the example available.

Artifact Analysis (incl. Outputs)

The outputs of this artifact are the service alive in the Knative platform, and examples of workflows in *JSON* format and their respective I/O datasets in *txt* format.

B. Computational Artifact A_2

Relation To Contributions

This is our framework for generating and managing HPC scientific workflows on serverless computing. All the components for the paper and all the experiments conducted are included here, organized by folder, as following:

- /experiments/src/
  - **serverless-workflows-wfbench.py**: this is our prototype of WFM for serverless.
  - **run_all_wfbench.sh** : this is a bash script that calls each experiment on our serverless setup.
  - **run_all_wfbench_local.sh** : this is a bash script that calls each experiment on our local container setup.
- /experiments/workflows/
  - **generate_workflows.py**: this script imports our Knative Translator module and use it to generate and translate workflows.
- /experiments/results/
  - **workflow_executions**: this folder contains the main results of our experiments - the measurements of execution time, and CPU, memory and power usage. They are grouped by paradigm, described in the paper in Table 2. They are:
    - **knative-sequential**
    - **knative-level**
    - **knative-scaling-1w**
    - **knative-scaling-1w-novm**
    - **knative-scaling-10w-novm**
    - **local-level**
    - **local-container-96w**
    - **local-container-96w-novm**
    - **local-container-960w-novm**
  - **workflows_descriptions/**: this folder contains, by group, the analyses for the workflows we used. The groups are:
    - **functions_invocation/** : this folder containers the analysis of the number of invocations per phase;
    - **functions_invocation_name/** : this folder containers the analysis of the number of invocations per function name.
- /analysis/graphs_visualization/
  - **generate_visualization.py**: this script generates the visualization for the workflows in the /experiments/-workflows. The outputs of this script are placed in the *pdf* and *png* folders in this same directory, and some of them compose Figure 3.
- /analysis/jupyter_notebook/
  - **analysis_wfbench.ipynb**: this is the Jupyter Notebook script that produces the Figures 4-7.
  - **analysis_wfbench_invocation.ipynb**: this is the Jupyter Notebook script that produces some of the figures in Figure 3.

Expected Results

The outcomes of using this framework are the experiments with different HPC scientific workflows over serverless platforms and local-based containers. Serverless-based executions should show benefits from bare-metal local container-based executions in terms of power, CPU and memory usage.

Expected Reproduction Time (in Minutes)

Artifact Setup: This artifact uses the same setup as Artifact A_1. **Artifact Execution:** The expected computational time of this artifact depends mostly on the workflows that are being executed. In our case, the total time, considering all workflows and all combination of different scenarios is 800 minutes (13 hours). More specifically, to reproduce each figure of the paper, representing our different sets of experiments: Figure 4: 100 minutes; Figure 5: 130 minutes; Figure 6: 510 minutes (where only the Epigenomics workflow takes 410 minutes); Figure 7: 60 minutes. **Artifact Analysis:** The expected computational time for running our analysis scripts, is about to 10 minutes.

Artifact Setup (incl. Inputs)

Hardware: There is no hard-requirements for hardware. But we ran the experiments using a 2-nodes cluster connected to one shared-drive for storing data. We ran all experiments on a local cluster with two nodes with Ubuntu 22.04.4 LTS. The first node was the master, with 2x AMD EPYC 7443 24-Core Processor, 48 threads, with 256 GB of memory. The second node was the worker, with 2x AMD EPYC 7443 24-Core Processor, 48 threads, with 192 GB of memory.

Software: This artifact has the same software requirements as Artifact A_1.

Datasets / Inputs: The datasets required for our experiments are HPC scientific workflows in *JSON* format. All the datasets are generated using our framework, using the script **generate_workflows.py**.

Installation and Deployment: This artifact has the same installation and deployment requirements as Artifact A_1.

Artifact Execution

This artifact's workflow consists of three tasks, T_1, T_2, and T_3, that interact as $T_1 \rightarrow T_2 \rightarrow T_3$. Task T_1 may generate the datasets, which are the HPC scientific workflows that are going to be executed. These workflows are then used as input by our workflow manager for serverless in task T_2. Task T_2 manages and executes the workflows in the Knative serverless platform (or bare-metal local containers for baseline purposes), and it stores the inputs and outputs in the shared disk that was set-up. In addition, our workflow manager uses Performance Co-Pilot (PCP) to measure resource usage (CPU, memory, and power). These outputs are processed by task T_3, which produces the final results (Figures 3-7).

Artifact Analysis (incl. Outputs)

The outputs of the experiments are *csv* files containing the measurements of execution time, power, CPU, and memory usage for each executed workflow. To process these outputs, we use our two Jupyter Notebook scripts (**analysis-wfbench.ipynb**, **analysis-wfbench-invocation.ipynb**) to produce Figures 3-7. In addition, the **generate_visualization.py** script generates parts of the visualization of the workflows for Figure 3 (this script can be executed before or after the experiments).

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

This artifact's workflow consist of two tasks T_1 and T_2, that interact as T_1, T_2. Once all the software requirements are installed (including the deployment and configuration of the Kubernetes cluster and Knative), proceed to install the Python libraries required:

```
$ cd src/
$ pip install -r requirements.txt
```

Then we proceed to the setup for each task:

- **Task** T_1: to deploy our contribution to WfCommons (as Task T_1), it is needed to first clone their repository, to copy our Knative Translator to there, and to compile the entire repository as an unique python package, as following:

```
$ cd .. & git clone https://github.com/wfcommons
    ↪ /WfCommons.git
$ cd wfcommons & cp ../wfm-serverless/wfbench/
    ↪ knative.py ./wfcommons/wfbench/Translator
    ↪ /
$ pip install -r requirements.txt
$ pip install . --break-system-packages
```

- **Task** T_2: to deploy WfBench as a Service, we follow similar processes for serverless or for bare-metal local containers:
  - To deploy WfBench as a service on **serverless**, first it is needed to make sure that the container is uploaded on DockerHub then it is possible to deploy it as a service on Knative:

```
$ cd wfbench/services/wfbench/wfbench/
$ sudo docker build -t
<dockerhub_user_name>/<container_image>:<
    ↪ containet_tag> .
$ sudo docker push <dockerhub_user_name>/<
    ↪ container_image>:<containet_tag>
$ kubectl apply -f service.yaml
```

As an example, using ou DockerHub repository[2]:

```
$ cd wfbench/services/wfbench/wfbench/
$ sudo docker build -t andersonandrei/wfbench-
    ↪ knative:wfbench-local .
$ sudo docker push andersonandrei/wfbench-
    ↪ knative:wfbench-local
$ kubectl apply -f service.yaml
```

[2]https://hub.docker.com/r/andersonandrei/wfbench-knative/

We remark that we **update and re-deploy** this Dockerfile, modifying a few parameters in two key files, for each set of experiments **with each computational paradigm** listed on Table 2. These files and parameters are:

* **Dockerfile**: there are modifiable parameters in the last line of the *Dockerfile* in this directory, such as: *number_of_workers, number_of_threads, timeout_response*:

```
CMD exec gunicorn --bind :$PORT --workers <
    ↪ number_of_workers> --threads <
    ↪ number_of_threads> --timeout <
    ↪ timeout_response> app:app
```

* **wfbench.py**: the *line 118* of the wfbench.py program, attached to the *Dockerfile*, enables or disables the use of persistent memory between the functions of the workflow:

```
118 "--vm-bytes", f"{total_mem}", "--vm-
    ↪ keep"] # or "--vm-bytes", f"{
    ↪ total_mem}"]
```

For instance, for the computational paradigm *Kn10wNoPM* where we have *10 workers* running and *no persistent memory* between the different functions, the last line of the *Dockerfile* should be:

```
CMD exec gunicorn --bind :$PORT --workers
    ↪ 10 --threads 1 --timeout 0 app:app
```

And the *wfbench.py* attached program should have:

```
118 "--vm-bytes", f"{total_mem}"]
```

  - To deploy WfBench as a Service on **bare-metal local containers**, for the baseline experiments, we deploy the container similarly as detailed above. As an example:

```
$ cd wfbench/services/wfbench-local/wfbench/
$ sudo docker build -t andersonandrei/wfbench-
    ↪ knative:wfbench-local .
$ sudo docker push andersonandrei/wfbench-
    ↪ knative:wfbench-local
```

We remark that the modifications to perform the experiments for each computational paradigm in local containers listed on Table 2 are the same as described for the serverless variations, but performed in the *wfbench/services/wfbench-local/* directory.

Artifact Execution

- **Task** T_1: To test if our Knative Translator is correctly deployed withing WfCommons, creating workflows and translating them to Knative, run the following:

```
$ cd wfbench/knative-translator/
$ python3 generate_worklfows_example.py
```

- **Task** T_2: To execute the WfBench service deployed on **serverless**, run the following:

```
curl http://<server_address>:<port_number>/
    ↪ wfbench -X POST -H 'Content-Type:
    ↪ application/json' -d '{"name":<
    ↪ function_name>, "percent-cpu":<
    ↪ percent_cpu>, "cpu-work":<cpu_work>, "out
    ↪ ":<outputs_dictionay>, "inputs":[<
    ↪ list_of_inputs>], "workdir":<
    ↪ workflow_data_location>}'
```

Notice that the following parameters are modifiable: *server_address, port_number, function_name, percent_cpu, cpu_work, outputs_dictionay, list_of_inputs, workflow_data_location*. As an example:

```
curl http://localhost:80/wfbench -X POST -H '
    ↪ Content-Type: application/json' -d '{"
    ↪ name":"split_fasta_00000001", "percent-
    ↪ cpu":0.9, "cpu-work":100, "out":{"
    ↪ split_fasta_00000001_output.txt": 10010},
    ↪  "inputs":["split_fasta_00000001_input.
    ↪ txt"], "workdir":"../data/BlastRecipe
    ↪ -250-100"}'
```

To execute the WfBench service deployed on **bare-metal local containers**, run the following:

```
$ docker run -t -v /mnt/data:/data --name
    ↪ wfbench --cpus=2 -p 127.0.0.1:80:8080/tcp
    ↪  andersonandrei/wfbench-knative:wfbench-
    ↪ local
```

Then, in a second terminal, run the same command as for Knative, but using the address of the local container. As an example:

```
curl http://localhost:80/wfbench -X POST -H '
    ↪ Content-Type: application/json' -d '{"
    ↪ name":"split_fasta_00000001", "percent-
    ↪ cpu":0.9, "cpu-work":100, "out":{"
    ↪ split_fasta_00000001_output.txt": 10010},
    ↪  "inputs":["split_fasta_00000001_input.
    ↪ txt"], "workdir":"../data/BlastRecipe
    ↪ -250-100"}'
```

Artifact Analysis (incl. Outputs)

The output of our Knative translator will be a workflow description in a JSON file format with entries for the Knative HTTPs request API. The outcome of deploying WfBench as a Service will be having WfBench as service in the Knative platform, or as a service in a local-container. These components and the outcomes here are intermediate. They will be used by the Artifact A_2 for the main experiments, as components of the same framework, for creating and managing HPC scientific workflows for Serverless.

B. Computational Artifact A_2

Artifact Setup (incl. Inputs)

This artifact's workflow consists on three tasks, T_1, T_2, and T_3 and they interact as $T_1 \rightarrow T_2 \rightarrow T_3$. Once all the software requirements are installed (including the deployment and configuration of the Kubernetes cluster and Knative), proceed to install the Python libraries required for having the workflow manager and all the scripts running:

```
$ cd src/
$ pip install -r requirements.txt
```

Install the R packages for the analyses:

```
$ R
$ install.packages("dplyr", "tidyr", "ggplot2", "
    ↪ data.table", "gridExtra", "scales", "
    ↪ patchwork", "RColorBrewer")
```

Artifact Execution

- **Task** T_1: to generates the HPC scientific workflows:

```
$ cd experiments/workflows/
$ python3 generate_workflows.py
```

Notice that the following parameters are all modifiable: *service_name, service_namespace, container_tag, container_url, volume_mount_name, volume_mount_path, cpu_request, memory_request, cpu_limit, memory_limit, volume_name, pvc_name, workflow_data_locality, workflow_manager_data_locality, knative_function_url*. The **generate_workflows.py** has a first example.

- **Task** T_2: we use our workflow manager through two different scripts:
  - **Serverless-based execution:** the experiments on serverless can be performed running the following bash script:

```
$ cd experiments/src/
$ chmod +x run_all_wfbench.sh
$ ./run_all_wfbench.sh
```

Notice that this script runs a repetition of the following command:

```
python3 serverless-workflow-wfbench.py -r <
    ↪ workflow_description>.json <
    ↪ workflow_name> <number_of_cpus> <
    ↪ computational_paradigm>
```

Notice that the following parameters are all modifiable: *workflow_description, workflow_name, number_of_cpus, computational_paradigm*. In particularly, *computational_paradigm* can be *knative* or *local*. As an example:

```
python3 serverless-workflow-wfbench.py -r ../
    ↪ workflows/wfcommons/SrasearchRecipe
    ↪ -250-1000/SrasearchRecipe-250-1000.json
    ↪  SrasearchRecipe-250-1000 1 knative
```

  - **Local-container-based execution:** for running our baseline experiments:

```
$ cd experiments/src/
$ chmod +x run_all_wfbench_local.sh
$ ./run_all_wfbench_local.sh
```

Notice that this script runs a repetition of similar commands as mentioned above, setting the *computational_paradigm* parameter to *local*.

To collect the metrics we need, on Task T_2, the workflow manager runs the following PCP command, for both serverless-base and local-container-based executions:

```
pmdumptext -d <separator> -f <date_format> -t <
    ↪ timestamp> <metric_1> <metric_2> <
    ↪ rapl_endpoint_1> <rapl_endpoint_2> <
    ↪ file_to_be_saved.csv>
```

As an example:

```
pmdumptext -d \',\' -f \'%d/%m/%y %H:%M:%S\' -t
    ↪ 1sec kernel.all.cpu.user mem.util.used
    ↪ denki.rapl.rate[\"0-package-0\"] denki.
    ↪ rapl.rate[\"1-package-1\"] > ../../
    ↪ wfbench/" + measurements_file_name + '.
    ↪ csv'
```

For a multi-node scenario, we run:

```
ssh -l <user_name> <node_address> "<pmdumptext
    ↪ command>"
```

- **Task** T_3: To do the analyses of our results:
  - **Graphs' Visualization (DAGs)** :

    ```
    $ cd analysis/graphs_visualization/
    $ python3 generate_visualization.py
    ```

    Notice that all workflows generated should be at *./generated_workflows/* and the results of this script is going to be saved at *./generated_workflows/png/* and *./generated_workflows/pdf/*.
  - **Metrics Analysis**:

    ```
    $ cd analysis/jupyter-notebook/
    $ jupyter-notebook .
    ```

    It will open the Jupyter-Notebook in a browser and then there will be a button called "runn all".
    Notice that the results from Task T_2 should be placed at *./wfbench/csv/<computational_paradigm>*, and all the outputs are going to be generated at the *analysis/jupyter-notebook/* directory.

Artifact Analysis (incl. Outputs)

The expected results after task $T3$ are the Figures 3-7 of the paper, in addition to some other visualizations that were used as support, but not selected to be presented. All these figures should be present at */analysis/results/png* and */analysis/results/pdf*. Analyzing these figures, it should be clear that serverless-based executions show benefits from bare-metal local-container-based executions in terms of power, CPU and memory usage.