Shared-Memory Communication for Containerized Workflows

Tanner Hobson, Orcun Yildiz, Bogdan Nicolae, Jian Huang, Tom Peterka

Shared-Memory Communication for Containerized Workflows

Tanner Hobson*, Orcun Yildiz†, Bogdan Nicolae†, Jian Huang*, Tom Peterka†

* *University of Tennessee*
EECS Department
Knoxville, TN, USA
thobson2@vols.utk.edu and huangj@utk.edu

†*Argonne National Laboratory*
Mathematics and Computer Science
Lemont, IL, USA
{oyildiz,bnicolae,tpeterka}@anl.gov

***Abstract*—Scientific computation increasingly consists of a workflow of interrelated tasks. Containerization can make workflow systems more manageable, reproducible, and portable, but containers can impede communication due to their focus on encapsulation. In some circumstances, shared-memory regions are an effective way to improve performance of workflows; however sharing memory between containerized workflow tasks is difficult. In this work, we have created a software library called Dhmem that manages shared memory between workflow tasks in separate containers, with minimal code change and performance overhead. Instead of all code being in the same container, Dhmem allows a separate container for each workflow task to be constructed completely independently. Dhmem enables additional functionality: easy integration in existing workflow systems, communication configuration at runtime based on the environment, and scalable performance.**

***Index Terms*—shared memory, workflow systems, containers**

I. Introduction

Scientific computing typically makes use of complex workflows that are composed of a large number of smaller tasks, each independently responsible for different types of computations: collection of input/output, simulations, and analytics. By splitting the workflow into a set of smaller tasks and managing the connections between them, different parts of the workflow can run concurrently at different scales. While individual tasks specialize, communication is the glue that turns a set of discrete tasks into a coherent and efficient workflow.

With increasing complexity of the tasks, composing a workflow is challenging for several reasons. First, each task may depend on a large number of libraries and runtimes that are not easy to deploy, nor to reconcile when multiple tasks need to share the same compute nodes. Second, some tasks may be implemented using legacy codebases that rely on old or unmaintained dependencies that are incompatible with the default installation. Third, the reproducibility of the scientific results is nontrivial if the ecosystem of libraries and runtimes is significantly different between runs.

It would be helpful to be able to isolate each task in its own environment, ideally without sacrificing performance and scalability. Initially developed in the context of cloud computing for the purpose of providing lightweight virtualization as an alternative to virtual machines, containers have recently seen adoption in HPC workflows as well, due to promising potential to solve the aforementioned challenge [16].

At its core, a container is an operating system virtualization approach that enables multiple isolated process groups to share the same kernel. The kernel exposes a mechanism to isolate both the environment (namespaces) and the performance, i.e., limit access to resources (cgroups). Of particular interest in the context of workflows is the isolation of the environment as a container image. While the kernel does not provide direct support in this regard—it assumes the environment is available as a mount point—tools such as Docker can be used to automatically build and share container images as a set of deterministic scripts that can import and modify other container images in a layered fashion. Such an approach can greatly increase the portability and reusability of container images, which was the main reason for their wide adoption.

Both individual tasks and groups of tasks, i.e. sub-workflows, can be packaged into container images, which in turn can be used to deploy a large number of container instances at scale. However, such an approach also introduces an important challenge: *efficient cross-container communication*. Specifically, if the container instances are co-located on the same node, then the inputs and outputs of the tasks can be passed between them through shared memory instead of messages in order to reduce communication overheads. However, containers do not share the same virtual address space. Therefore, a naive solution that simply serializes the data structures into shared memory is not significantly more efficient than sending the serialized data structures as a message. Furthermore, the locality of the containers also influences the performance of the synchronization: instead of messages, there are other, potentially more efficient alternatives: position-independent data structures, inter-process OS primitives.

This paper introduces *Dhmem*, an abstraction specifically designed for efficient cross-container communication. The key idea is to present a uniform data sharing and synchronization service that transparently implements a dynamic strategy to choose the optimal data sharing and synchronization primitives based on task location and isolation requirements. To this end, it exposes high-level data structures whose representation is specifically optimized to avoid serialization overheads, even

for tasks running in different co-located containers.

Using this approach, applications are freed from having to worry about the location of containers, which effectively helps scientists cross over containerized encapsulations of workflows as well as other system boundaries. In addition, Dhmem is easy to adopt because the code changes required to share a data structure are solely related to changing a type declaration: the kind of change required is simply to switch from one STL-style C++ class to another STL-style class. Furthermore, another advantage unlocked by the use of shared memory (in addition to lower overhead) is the reduction of the total in-core memory footprint, a crucial system resource in future exascale systems. We summarize our contributions as follows:

- We introduce Dhmem, a novel workflow abstraction specifically built to optimize data sharing between inter-container tasks, insisting on several key design principles (Section III-B).
- We present an implementation of the design principles as a cross-platform C++ library (Section III-C).
- We position Dhmem in the ecosystem of state-of-art HPC workflow solutions and discuss how it can be integrated with two of them (Section III-D).
- We perform a series of experiments using benchmarks that compare Dhmem both with a traditional solution based on serialized messages that sacrifices performance to achieve isolation and with a thread-based solution that sacrifices isolation to achieve performance (Section IV).

II. Background & Related Work

Dhmem builds upon several foundations of scientific workflow systems ranging from workflow system design, their definition, and their means of communication. It extends these foundations through the use of shared memory which has had success in other areas. Dhmem also builds on container research.

A. Workflow Systems

Scientific workflow systems manage the composition of workflow tasks along with their data dependencies. These dependencies instruct the workflow system how the communication between tasks should be done. In practice, there are two main types of workflows: distributed and in situ [22]. This distinction primarily characterizes whether tasks are collocated on the same machines or whether a disparate set of compute resources are to be used.

The communication mechanisms behind distributed and in situ workflows also differ. Distributed workflow systems often communicate via files or sockets and separate parts of the workflow can be run at different times. In situ workflows instead are always run at the same time and have a dedicated communication channel between them. Some examples of each include: (distributed) Pegasus [10] and Parsl [1]; (in situ) ADIOS [18], Decaf [13], Henson [20], and Damaris [12]. The former are mostly focused on coordinating data movement across heterogeneous sets of machines, while the latter can focus on fast communication within one set of homogeneous compute machines.

In situ workflow systems, the focus of this paper, communicate by passing messages between tasks through memory or the system interconnect. Communication can also happen via shared memory when tasks are on the same node [22]. This happens automatically and transparently by the library.

The design of Dhmem is agnostic to the choice of workflow system. In this paper, we illustrate its use in two workflow systems: Decaf [13] and Henson [20]. Decaf is based on MPI communication and features transparent data distribution to workflow tasks. Henson is instead based on `dlopen` system calls and runs each task as a coroutine with minimal memory copying overhead.

B. Containerization

While containerization started with cloud computing [2, 8] and not with HPC, scientific workloads have started adopting containers to manage complex toolchains and software dependencies to ease deployment in multiple Linux variants [15]. Widely deployed container systems that have found use in HPC include Docker [5, 9, 19], Singularity [16, 17, 24], and Shifter [3, 4].

These improvements are applicable and important for workflow systems, e.g., to help leverage both HPC and cloud resources for in situ processing [7], and to help envision unified resource management in the context of exascale computing [21].

Although some approaches allow sharing memory across processes [14], sharing memory across containers is in practice not done because an entire application is built in a single container. It is only in the context of coupling multiple applications in a workflow that the situation arises, and to the best of our knowledge, our paper is the first to develop a flexible and reusable solution to this problem.

Existing research in containerized workflows targets distributed workflows, producing a single, reusable container to run the scientific codes [25]. In contrast, in situ workflows usually require special integration into their codebases, so a single container is less useful. One project that makes use of containers for in situ workflows is SENSEI [23], which packages a complex toolchain of scientific codes into one Singularity container for ease of use.

As most in situ workflow systems are based on MPI, a compatible container system must support it easily. Currently, the easiest path to using MPI communication between containers is via Singularity. Singularity is built and optimized for scientific tools, compared to Docker which targets enterprise and businesses. This is due to Singularity's execution model, where each container acts as a simple executable, whereas running Docker containers requires a separate executable.

Dhmem is designed to be agnostic to its environment so that a task can easily run inside of a container as well as outside. In either case, the developer's code is the same.

C. Communication and Shared Memory

Communication for in situ workflow systems is usually based on message passing interfaces like MPI. This is due to their

encapsulation guarantees that make combining different parts of a workflow easier, but not necessarily faster.

All message passing communication methods have a drawback: the data being transferred must exist in at least two places at once—one from the sending side, one from the receiving side—plus any extra copies the library makes internally. This limitation imposes an upper bound on communication.

An alternative form of communication exists and has been a part of modern computing systems for decades: shared memory. Shared memory is exposed via POSIX `mmap` system calls, OpenSHMEM libraries [6], MPI one-sided communication [11], and many other tools. Shared memory enables a form of zero-copy communication from which workflow systems can benefit.

Two forms of shared memory exist that we call "automatic" and "directed." Automatic shared memory has a drawback in its inability to choose to have some data in its own memory space. In the automatic method, all memory allocated is immediately shared with another process. This is the case for multithreaded programs and workflow systems that dynamically load executables (e.g. with `dlopen`). Directed methods, instead, offer the developer more control over what data are shared, such as OpenSHMEM's `shmalloc`.

More recently, the use of automatic shared memory for workflow systems has been explored in Process-in-Process [14] using the `dlopen` system call. With the newer `clone` system call, this general principle could be applied to arbitrary executables rather than specially compiled libraries like with `dlopen`. Regardless of the method, both approaches enable processes to share the exact same address space, meaning any newly allocated data are automatically shared and making individual containers for each workflow task impossible.

Dhmem uses the Boost Interprocess library, making it easier to manage shared memory resources in a directed way. Boost Interprocess is built on top of `mmap` system calls and provides not just a single shared memory region, like OpenSHMEM does, but also includes a custom allocator to dynamically allocate data structures into shared memory. This dynamic allocation makes integrating Dhmem into an existing workflow that uses familiar STL data structures easier.

Our objective is to use shared memory in workflow systems to enable safe, zero-copy, directed communication. The main research questions, then, are how efficient is communicating across containers via shared memory, and how can such functionality be integrated into existing workflows.

III. Design

This section introduces Dhmem, an abstraction and runtime library for workflows that is specifically designed for efficient cross-container communication. We detail the design principles, architecture and implementation details.

A. Deployment model

The use of containers in workflows is illustrated in Figure 1. Most existing workflow literature follows either (a) or (b). In (a), no containers are used at all, and everything is based on how the entire system is structured. In (b), one container

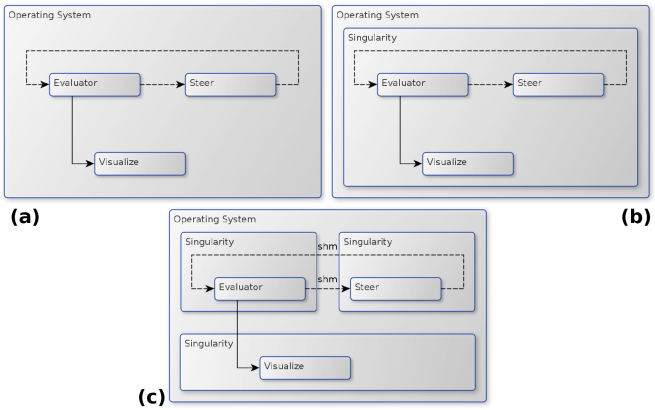

Fig. 1: Containerization of workflows using Singularity for an example with three tasks (Evaluator, Steer, Visualize). (a) provides no encapsulation from the host system. (b) provides some encapsulation from the host system, but none from other tasks. (c) provides the best encapsulation of the host system and each other task, but is most efficient using shared memory.

is used for the entire workflow, and every workflow task's dependencies need to be installed in that container.

Our use case is based on Figure 1(c), where every workflow task is in its own container. In this scenario, sharing memory is made more challenging because of the encapsulation that containers provide. The benefit is that different workflow tasks can depend on different and possibly conflicting dependencies, while being combined together into a single workflow. Dhmem also provides value in (a) and (b) for MPI-based workflows when tasks are collocated on the same compute node, thanks to a series of design principles detailed below.

B. Design principles

Dhmem is based on the following general design principles:

a) ***Unified, locality and isolation-aware model for data sharing and synchronization****:* If two tasks reside on different compute nodes, then the communication between them necessarily involves the transmission of messages that encapsulate a serialized version of the data structures to be exchanged. However, if two tasks reside on the same compute node, a better solution would be to simply put the data structures directly into a shared memory space in order to avoid additional overheads, just like threads. While this is straightforward if the tasks are implemented as threads (and still possible if they are implemented as processes that share the same address space), alternative approaches are needed for tasks running in co-located containers, because sharing the same address space is not possible due to isolation constraints. Similarly, based on location and isolation requirements, tasks may use messages, inter-process or thread-specific synchronization primitives. However, this creates a high level of complexity for workflow developers, both because the locality and the isolation of the tasks may not be known in advance (e.g. it may be decided by the scheduler) and because they may not be system-level experts that can understand and fine-tune their tasks for

all possible combinations. To this end, we propose a unified model that hides the complexity from the workflow developers, transparently optimizing the data sharing and synchronization based on task location and isolation requirements.

*b) **Position-independent high-level data structures**:* Modern HPC workflow systems need to run tasks that consume and produce complex inputs and outputs. Therefore, in a typical scenario, the communications between tasks involve complex data structures that maintain pointers to many contiguous memory regions scattered across the memory space (e.g., sparse multi-dimensional data structures, trees, dictionaries, etc.). If two tasks running in co-located containers need to exchange such data structures, they cannot simply use a shared memory segment for this purpose, because it would be mapped at different virtual addresses, therefore invalidating any pointers. Given the large size and number of contiguous memory regions, a naive solution that serializes such data structures into a shared memory segment introduces a significant overhead due to the need to assemble and copy all fragments into a single contiguous region. To address this issue, we propose the use of position-independent, high-level data structures that are based on relative offsets instead of pointers. Thanks to this approach, Dhmem automatically avoids serialization overheads for a large number of common high-level data structures without sacrificing isolation constraints.

*c) **Ease of integration with existing workflow systems**:* Dhmem is designed as a standalone service that integrates into the workflow as a complement, without changing the core functionality of the workflow itself. It is architected as combination of a core Dhmem library and a support library for each different workflow system. The core library provides base shared memory functionality. This functionality is used in each of the support libraries to integrate into workflow systems, thereby extending the available synchronization methods available to Dhmem. This separation of concerns enables two advantages simultaneously: (1) the core Dhmem library can be independently optimized and improved, thereby benefiting all workflow systems for which a support library exists; (2) containerization can be flexibly added or removed in any configuration (even dynamically during runtime) without impacting the structure of the workflow or the task implementation.

C. Dhmem Core Library

Dhmem's core library implements everything needed to work with shared memory regions and data structures. The core library enables tasks to connect to a shared-memory region, allocates data structures, and saves/loads process-independent pointers to those data structures.

Dhmem builds on top of the Boost Interprocess library, adding discoverability and usability in the context of a workflow. For example, Boost Interprocess can allocate and reference data structures by name, and Dhmem helps generate consistent names and interacts with the structures.

Several programming language constructs are used in Dhmem, enabling the use of shared-memory structures in a uniform way. We leverage C++ references to interact with a variable as if it were stack-allocated, when in reality the variable is stored in a shared-memory region. In most codes, this means only the variable definition needs to be updated, while its use remains the same.

In the simplest case, replacing ordinary data structures with shared-memory structures is as simple as replacing the `std::` prefix with `dhmem::` and calling a different constructor as shown in Listing 1.

```
+dhmem::Dhmem dhmem("shmem_namespace");

-int n;
+int &n = dhmem.simple<int>("my_n");
 n = 123;

-std::vector<int> v;
+auto &v = dhmem.container<dhmem::vector<int>>("v");
 v.resize(VSIZE);

-MPI_Send(v.data(), VSIZE, MPI_INT, 0, 0, comm);
+dhmem::handle h = dhmem.save(v);
+MPI_Send(&h, 1, MPI_DHMEM_HANDLE, 0, 0, comm);

-MPI_Recv(v.data(), VSIZE, MPI_INT, 0, 0, comm,
-         MPI_STATUS_IGNORE);
+dhmem::handle h;
+MPI_Recv(&h, 1, MPI_DHMEM_HANDLE, 0, 0, comm,
+         MPI_STATUS_IGNORE);
+auto &v = dhmem.load<dhmem::vector<int>>(h);
```

Lst. 1: Comparison of code changes to use Dhmem. Red lines prefixed with "-" indicate code before using Dhmem while green lines prefixed with "+" indicate newly added code.

The remainder of this section details how to add Dhmem to an existing workflow task, in order from the simplest case to the more complex edge cases.

Shared Memory Region. The first step to add Dhmem to a workflow task is to open or create a new shared memory region. This is accomplished with the `Dhmem` class which takes 2 or more parameters: the mode for the shared-memory region and its name.

The name uniquely identifies the memory region on a single machine and must be the same for all workflow tasks. A reasonable default is the name of the workflow itself. The mode tells Dhmem whether to open an existing region, create a new one, delete an existing one and then create a new one, or some combination thereof. An example is demonstrated in Listing 2.

Shared Primitive Variables. The simplest way to use Dhmem is when allocating primitive variables or arrays of them. These allocations are most commonly used for sending

```
if (mpi_rank == 0) {
  dhmem::Dhmem dhmem(create_only, "my_shm", 65536);
  MPI_Barrier(comm);
} else {
  MPI_Barrier(comm);
  dhmem::Dhmem dhmem(open_only, "my_shm");
}
```

Lst. 2: Connecting to shared-memory regions with Dhmem.

timestamps or other scalar values, though arrays can be leveraged to send several values at once. Simple allocations differ from other allocations by being a fixed size and not allocating any more data at runtime.

Dhmem exposes simple allocations via its `simple` function which takes a template parameter for the type of the data to allocate. The function also takes a name for this variable that can be used in other tasks to get a pointer to the same primitive variable. The type parameter can be simple types (`float`, `int`, etc) or structures of simple types. An example is given in Listing 3.

```
int &n = dhmem.simple<int>("my_n");
n = 3 * n + 1;  // usable like any other integer

struct S { int i; int j; int k; };
S &s = dhmem.simple<S>("my_s");
s.i = 1; s.j = 2; s.k = 3;

auto &arr = dhmem.simple<float[512]>("my_arr");
arr[0] = 1;
```

Lst. 3: Demonstration of shared primitive variables.

Shared Data Structures. Resizable arrays or other complex data structures in shared memory pose challenges because normal STL structures internally store and use regular pointer addresses that are not valid across process boundaries. Shared-memory data structures use position-independent pointers instead. These data structures also need to be able to dynamically allocate more memory at runtime needing a reference to the shared-memory allocator.

For most workflow tasks, simply replacing all instances of STL structures with Dhmem ones and using Dhmem to do the allocation suffices. In practice, this means replacing `std::vector` with `dhmem::vector` and `new` with `dhmem.container<>()`.

For `structs` and `classes` of shared data structures, it is necessary to forward the memory allocator that Dhmem provides to the other shared structures. A simplified code example of both regular vectors and `structs` of vectors is shown in Listing 4.

```
dhmem::vector<int> &v =
  dhmem.container<dhmem::vector<int>>("my_v");
v.push_back(5);  // allocates more shared memory

struct S {
  S(dhmem::allocator<void> alloc)
    : v1(alloc), v2(alloc) {}
  dhmem::vector<int> v1, v2;
};
S &s = dhmem.container<S>("my_s");
s.v1.push_back(5);
```

Lst. 4: Demonstration of shared data structures.

Shared Memory Handles. In Dhmem, there are two ways of referring to the same data in multiple tasks: by name and by handle. Handle-based references are used when referencing variables by names is inconvenient, such as when a workflow system is already being used to send other data structures. The handle is just another primitive variable that the workflow system sends like any other. Handles are essentially offsets into a shared memory region, though in the future, they could also contain the name of the shared memory region itself.

Dhmem handles are manipulated via two methods: `save` and `load`. The save method takes an already allocated shared variable and returns a handle, while load does the inverse. In both cases, the name of the variable itself is never included in the handle. In order to facilitate handle usage with MPI, a special constant `MPI_DHMEM_HANDLE` is exported that is compatible with `MPI_Send` and `MPI_Recv`. An example using handles is shown in Listing 5.

```
// producer
int &n = dhmem.simple<int>("my_n");
dhmem::handle h = dhmem.save(n);
MPI_Send(&h, 1, MPI_DHMEM_HANDLE, dest, tag, comm);

// consumer
dhmem::handle h;
MPI_Recv(&h, 1, MPI_DHMEM_HANDLE, src, tag, comm,
         MPI_STATUS_IGNORE);
int &n = dhmem.load<int>(h);
```

Lst. 5: Demonstration of Dhmem handle usage.

D. Dhmem Support Libraries

Dhmem supports multiple different workflow systems in its "support libraries." In this paper, we demonstrate 3 different support libraries—Decaf, Henson, and a standalone library for use outside of a workflow system—although Dhmem is designed to be easy to integrate into other workflow systems. For this reason, the core library is larger than any of its support libraries. In each of the cases, it is important that Dhmem works together with the workflow system rather than try to completely reinvent its communication. The most useful Dhmem construct for support libraries is the Dhmem `handle`.

In each of the following sections, each supported workflow system is discussed including how their data model works, how Dhmem fits into that data model, and what kind of source code changes are necessary. Each of the following cases require minimal changes to the workflow task source code. In each of them, the following steps are needed to integrate Dhmem.

(Step 1) The Dhmem library is built and installed.

(Step 2) The workflow task is linked with the Dhmem library.

(Step 3) The workflow task code allocates a `Dhmem` object to create and connect to the shared memory region.

(Step 4) The variables that should be shared are changed from `std::` data structures to `dhmem::` data structures and allocated with Dhmem.

(Step 5) The workflow-specific support libraries are used to save and load Dhmem handles between tasks.

Steps 1-4 are the same for every workflow, while Step 5 varies according to the workflow system. This is by design and means that the core usage of Dhmem is transferable between workflow systems. Because each workflow system uses a different mechanism for communication between tasks, each support library needs to accommodate this difference.

1) Decaf Support Library: Decaf [13] is a workflow system for high-performance, decoupled, in situ scientific workflows. Its tasks communicate via messages consisting of a hierarchical name-value mapping. These messages can then be serialized

into a flat memory buffer, sent over MPI, and later deserialized back into normal data structures.

Decaf's data model revolves around `pConstructData` objects. These objects have a name-value mapping where each value is a `Field` object. `Field`s contain a pointer to the data and also (de)serialization methods. Some fields are predefined, such as the `SimpleField` (for primitive variables) or `ArrayField` (for arrays of primitives). Fields are added to the mapping using the `appendData` method.

Dhmem can improve Decaf performance due to the nature and inherent overhead of (de)serialization. In essence, Dhmem's Decaf support library serializes shared data structures as handles rather than as their actual contents. To integrate easily into Decaf's API, this special serialization logic is exposed as a new `SharedField`, added to Decaf.

An example of integrating Dhmem into Decaf is shown in Listing 6. It is important to note that from a developer perspective, Dhmem is entirely optional and used only where it would improve performance, such as large arrays of data.

```
// send data
int &n = dhmem.simple<int>("my_n");
SharedField<int> nfield(n, dhmem);
pConstructData out_msg;
out_msg->appendData("n", nfield);
decaf->put(out_msg, "out_port");

// receive data
std::vector<pConstructData> in_msgs;
decaf->get(in_msgs, "in_port");
SharedField<int> field =
  in_msgs[0]->getFieldData<SharedField<int>>("n");
int &also_n = field.getData(dhmem);
```

Lst. 6: Using the Dhmem SharedField API. Blue lines indicate changes from ordinary Decaf code.

2) Henson Support Library: Henson [20] is a workflow system that loads multiple workflow tasks into the same address space. Due to tasks being in the same address space, all tasks must run inside of the same container. Henson does not communicate via messages, but instead via cooperative multitasking using coroutines. Each task is loaded by the Henson process and any allocated data allocated is available to any other task running under the same process.

Henson's data model is a one-level name-value mapping where each value is either a primitive variable or a pointer to an array. These pointers are passed verbatim between workflow tasks because they share the same address space. Data in Henson are either `saved` or `loaded` via type-dependent methods like `henson_save_int` or `henson_load_pointer`. This mapping is global to the entire workflow so that the same name refers to the same values throughout.

Henson's execution model prevents direct usage when every workflow task is containerized, but Dhmem offers a path forward for Henson tasks to work transparently within containers. In essence, rather than saving and loading pointers to arrays of data, Dhmem can save and load handles to shared arrays of data. For this support library, Dhmem exposes two new methods: `henson_save_handle` and `henson_load_handle`. A simplified example of these new methods is shown in Listing 7.

```
// save data
int &n = dhmem.simple<int>("my_n");
henson_save_handle("n", n, dhmem);

// load data
int &also_n = henson_load_handle<int>("n", dhmem);
```

Lst. 7: Using the Henson Support Library API.

3) Standalone Support Library: Dhmem is also intended to be usable without an underlying workflow system. There are two supported communication methods: low-level and MPI. In low-level communication, access to a shared Dhmem handle is controlled by a shared mutual exclusion lock. In MPI communication, Dhmem handles are sent using MPI functions. These two methods can be used either instead of a workflow system or in addition to one.

Standalone communication involves individual `Port`s that control access to a single variable (or a `struct` of variables). A `Port` is allocated based on whether the current process is on the producing or consuming end. An example is in Listing 8.

```
// send data
dhmem::Port port =
  dhmem.port("n_port", dhmem::producer, dhmem::mpi);
int &n = dhmem.simple<int>("my_n");
port.send(n);

// receive data
dhmem::Port port =
  dhmem.port("n_port", dhmem::consumer, dhmem::mpi);
int &n = port.recv<int>();
```

Lst. 8: Using the Standalone Support Library API.

IV. Evaluation

This section evaluates the performance and scalability of Dhmem in a series of workflow benchmarks that emphasize the low data sharing overhead between co-located containers. We compare our proposal with several state-of-art approaches.

A. Experimental Setup

a) Platform: Our experiments were performed on a single node of a shared compute-optimized machine, featuring a 16-core Intel Xeon Gold 6130 processor running at 2.10 GHz with 128 GB of RAM. In terms of software, this machine is running Ubuntu 18.04.5 with Linux kernel 4.15.0 and MPI support is from MPICH 3.3.2.

b) Benchmarks: To minimize the variability of our experiments and to isolate the aspects we are interested in evaluating as best as possible, we designed a series of synthetic benchmarks that focus on common workflow data sharing patterns, while minimizing the interference from everything else. These synthetic benchmarks can be parameterized and automatically generated (Section IV-B). Furthermore, since the isolation provided by containers is orthogonal to our evaluation and the overhead of containers is minimal compared with normal processes, we simplify our setup to use normal processes that make use of shared memory segments (mapped at different virtual addresses).

c) Compared approaches and metrics: We compare our approach with two other approaches: (1) *Threads*: it encapsulates tasks into threads, thereby featuring no isolation (and extra complexity) but in exchange enabling the tasks to take advantage of a shared virtual address space, where pointers are passed directly between tasks and a producer-consumer pattern is implemented with OS-level primitives (mutexes and condition variables); (2) *MPI*: it encapsulates tasks run in separate processes and relies on serialization/deserialization of the data structures into separate contiguous memory regions as a means of communication, while using MPI primitives to synchronize. These approaches are compared in a variety of scenarios based on the achieved throughput for accessing the shared data structures, which is calculated as the total size of the data transferred between the processes divided by the runtime of the benchmark instance. The higher the throughput, the better the performance.

B. Synthetic Workflow Generation

To evaluate Dhmem in a variety of workflow configurations, we developed a synthetic workflow generation script, which generates C++ source code to start and run the workflow from a succinct workflow description. The workflow description is a list of workflow-, task-, and port-level definitions. The workflow can be configured to use either Dhmem or MPI and this configuration can apply between any two tasks. In addition, tasks can be customized based on how much data they send per iteration and how long each iteration should take. These parameters allow modeling of a range of workflow tasks including simulation, analysis, and visualization.

We can synthesize three types of workflow graphs, shown in Figure 2: simple, pipeline, and scatter-gather. Simple workflows model simplistic producer-consumer pairs, where one task produces data that the other task consumes. Pipeline workflows model long chains of workflow tasks that all produce some data that the next task processes. Scatter-gather workflows model a single producer task that sends the same data to several other intermediate tasks which later send their data to a single consumer task.

Synthetic workflow options include: the maximum number of iterations to run (e.g. 1024); the size of the shared memory region (e.g. 1 GB). Task options include: the time between sending data (e.g. 1 second); the size of the data sent between tasks (e.g. 1 MB). Port options include: the communication method used (e.g. Dhmem or MPI).

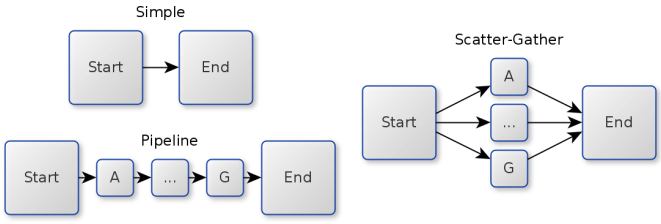

Fig. 2: Example types of workflows. Simple, pipeline, and scatter-gather workflows model real scientific workflows in a repeatable and testable way.

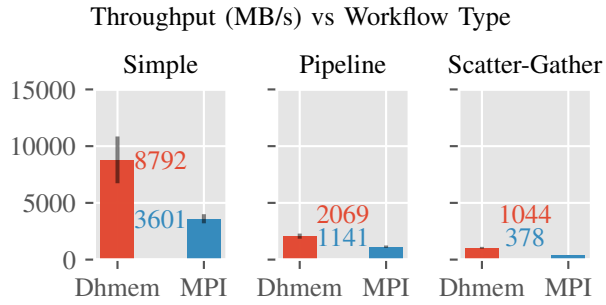

Fig. 3: Memory throughput for Simple, pipeline, and scatter-gather synthetic workflows using Dhmem and MPI communication strategies.

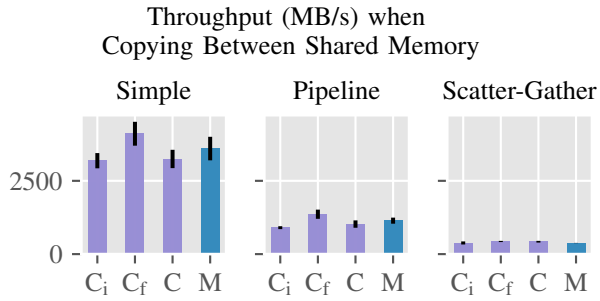

Fig. 4: The memory throughput for workflow tasks whose main algorithm cannot be modified to use Dhmem. Instead, Dhmem is used at the boundaries where tasks copy into (C_i), copy from (C_f), or both (C). These are compared with MPI (M). In each case, copying data from shared memory yields the best results as this can happen in parallel with the generation of new data.

C. Results: Dhmem vs MPI

In this test, we compare the memory throughput of synthetic workflows that communicate using Dhmem or MPI. The main distinction comes from how the data is allocated and referenced: using `dhmem::` vectors and using `std::` vector. This test serves as a baseline to compare other tests with. Each test is run for 1024 and 2048 iterations and their throughput is calculated based on the wall clock time, the number of iterations, and the amount of data sent between tasks. These trials are then repeated 5 times and their results are averaged.

As can be seen in Figure 3, Dhmem offers a significant improvement in simpler workflows like simple and pipeline, and a more modest improvement for complex workflows like scatter-gather. This is mostly due to the fact that, in the larger workflows, MPI has to repeatedly send a 10 MB buffer while Dhmem can simply copy the 8 byte Dhmem handle around. Although the difference in transmitted data size is large, the design of the synthetic workflows is such that all data is fully read and written in each task.

D. Results: Copy-into-Shared

Not all scientific codes can be easily modified to use Dhmem. In such cases, it still can make sense to use Dhmem at the

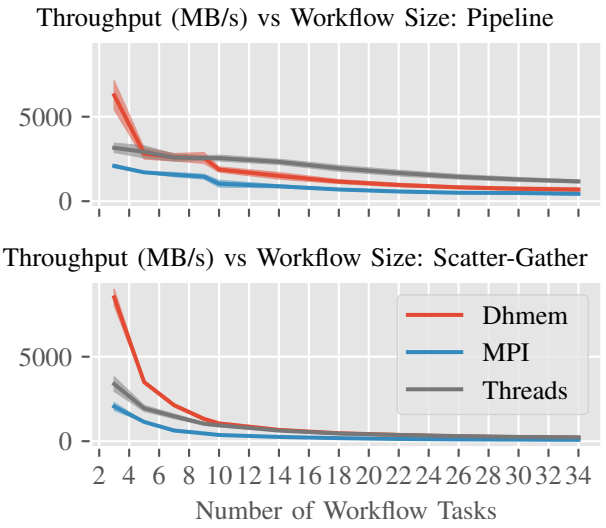

Fig. 5: Running workflows under threads provides a good target for the performance of multiprocess workflows. Compared to running with threads, Dhmem provides better performance for scatter-gather workflows and slightly worse for pipeline workflows.

boundaries between tasks. A minimal example of this idea is to use STL data structures within the task and afterwards copy the data into a shared-memory data structure for use in later tasks. This use case is predicated on the theory that performance can be improved using shared memory with minimal or even zero code change within the task. Then, the question is whether this improves performance compared with simply sending the data via MPI.

To test this, we consider the following cases: a sending task copies data into Dhmem (copy into), a receiving task copies data from Dhmem (copy from), both sending and receiving tasks copy data to and from Dhmem (copy). This copy is done by a direct `memcpy` between the two structures. In effect, this `memcpy` is doing the actual communication between tasks.

The results are shown in Figure 4 where the labels represent copy into (C_i), copy from (C_f), copy (C), and MPI (M). The copy from results exhibits the best performance. This is because the workflow cannot progress while data is being copied from the sending process, but the receiving process can begin copying from Dhmem. The copying methods get the most utility when the same data is used by multiple tasks, like in the scatter-gather use case.

These results indicate that even when Dhmem cannot be fully integrated into tasks and can only be used at boundaries, certain workflows can still see improved throughput. These workflows are typified either by producing large amounts of data that would be expensive to send using MPI, or by producing data that is consumed by lots of tasks.

E. Results: Threads vs MPI

Our next study is especially important for workflows that are considering the advantages of isolation (e.g., isolation is not mandatory but would simplify development) but need to understand the performance penalty before committing to such an approach. To this end, we add a comparison with the *Threads* approach, described in Section IV-A. We note that, in practice, scientific workflows would not use the threads approach to put the entire workflow within one process. Hence, this threading comparison serves to better highlight the overhead of shared memory methods both within and between processes.

For completeness, we also include a comparison with the *MPI* approach. We study the scalability of all three approaches for an increasing number of processes, ranging between 3 and 34. Each trial is run 5 times and the results are averaged.

The results are shown in Figure 5. For up to 10 tasks, Dhmem offers better performance than both *MPI* and *Threads*. This is a surprising result, because intuitively the *Threads* approach should always be the fastest. The explanation lies in the overhead of the synchronization primitives: Dhmem uses MPI messages, which are implemented using busy waiting, thereby being faster than OS primitives, which incur context switches, as long as spare hardware threads are available. Therefore, even if position-independent data structures have slightly higher overhead, this is negated by the lower synchronization overhead. However, with increasing number of processes, the situation reverses as expected. Based on these observations, we can draw two important conclusions: (1) unified data sharing and synchronization enables Dhmem to apply internal optimizations that can outperform even thread-based solutions under certain circumstances; (2) position-independent data structures enables Dhmem to scale well: it follows the *Threads* approach closely and consistently outperforms the *MPI* approach.

F. Results: Data Size Scalability

To evaluate how Dhmem scales with the amount of data transferred between tasks, we measured the throughput for different data sizes. The data is scaled between 1 MB and 1 GB and tested on the 3 workflow types with the pipeline and scatter-gather configured to use 9 total workflow tasks.

The results are shown in Figure 6 on a log-log scale. In general, Dhmem consistently outperforms MPI for same-node communication. As the data size increases, Dhmem becomes I/O bound by the testing machine. This causes the throughput for each test to stay the same despite sending more data each iteration.

Another important conclusion that can be drawn from this test is the latency of sending and receiving data with Dhmem. This latency can be summarized as the maximum number of iterations per second. Across all tests, the maximum iterations per second for Dhmem compared with MPI is: simple 1537.3 vs 1726.3 (0.89×), pipeline 854.5 vs 702.5 (1.21×), and scatter-gather 666.8 vs 360.6 (1.85×). These results show that Dhmem achieves lower latency than MPI for more complex workflows than for simple workflows.

G. Results: Containers

As Dhmem is built and targeted towards the containerized use case, it is necessary to compare performance inside a

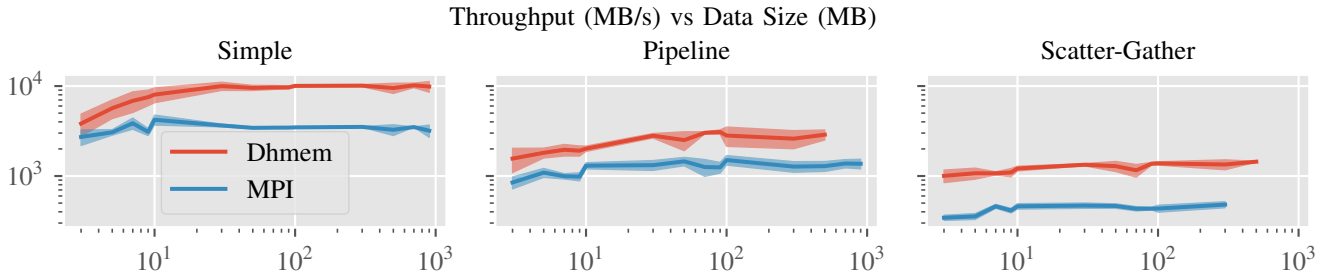

Fig. 6: Scalability test of Dhmem as data size increases. Eventually workflows become I/O bound and cannot send data faster.

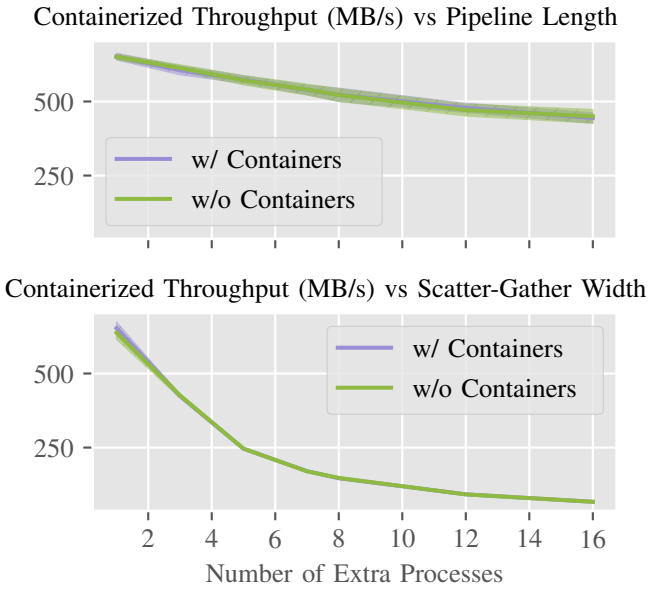

Fig. 7: This test compares Dhmem's performance inside of containers and its performance on the machine directly. The performance in the two cases is nearly identical and demonstrates the ability to Dhmem within containers without penalties.

container with performance outside of one. In this test, we reuse the same setup as in Section IV-E which exercises the different parts of the system the best. The main difference comes from where the test is run, and how the code is built.

Because Singularity requires an HPC environment, this test uses a different and slower machine than the other tests. This machine has a 36-core Intel Xeon E5-2695 processor running at 2.10 GHz with 128 GB of RAM. The code is first built as a Docker container image which is then converted into a Singularity container image on the testing machine and executed there. These results are then compared with running on the machine directly.

The results are shown in Figure 7. As shown, Dhmem offers equivalent performance inside of a container and outside of one. This is due to its design and focus towards running in containerized environments. Because the performance is so similar in the tested cases, the other experiments are representative of the performance in containers as well.

	Dhmem Setup	Task-Specific
Lines of Code	10 (1.9%)	8 (1.5%)

TABLE I: Number of lines of code changed in order to integrate Dhmem into an existing Decaf workflow. In this case, a large floating point array is stored and referenced via shared memory rather than serialized and transferred using MPI.

H. Code Impact for an Existing Workflow

In order to evaluate Dhmem's ease of integration, an existing workflow system was modified to share large data structures directly using Dhmem. This workflow is based on the previously published tessellation and density estimation workflow using Decaf [13] and is most similar to the pipeline workflow. The main data the workflow sends between tasks are large sets of 3D points, i.e. floating point data.

This test is primarily focused on developer productivity and as such is intended to illustrate the ease of modifying code to use Dhmem. For this reason, the primary metric is number of lines of changed code. These lines fall into two categories: Dhmem setup and task-specific setup. The former encompasses including header files, setting up the Dhmem shared-memory region, and passing the Dhmem region to each task. The latter includes creating shared data structures and sending them between tasks.

The results are shown in Table I. The Dhmem specific code, which is universal and required in every workflow task, is and should only ever be around 10 lines. The task-specific code will depend on exactly what the workflow is doing and what data needs to be in shared memory. In this workflow, a large floating point array is stored inside of a `dhmem::vector` and totals 1 MB. These results demonstrate that Dhmem can be integrated into workflows with a minimal amount of code change.

V. Conclusion

Containerization has the potential to make workflows more manageable, reproducible, and portable; however, containerization implies isolation, which may hinder optimized communication and integration that is possible when tasks are co-located on the same node and thus can take advantage of shared memory. Hence, we want enable isolation with minimal loss of performance.

Features. Dhmem offers a unified data sharing and synchronization model that enables tasks to easily share data without having to worry about isolation and locality, which are the main factors that affect performance and are used by Dhmem to transparently apply optimizations.

Benefits. When tasks are encapsulated in containers co-located on the same node, Dhmem takes advantage of position-independent data structures to enable performance levels close to direct sharing in the same address space without sacrificing isolation. It can even surpass direct sharing using OS-level synchronization primitives under certain circumstances. It performs better than message passing approaches that rely on extra copies and serialization.

Limitations. Position-indepenent high-level data structures require a specialized implementation that replaces pointers with offsets. While Dhmem can cover wide range of standard C++ data structures through the Boost library, custom user data structures that do not respect this model cannot take advantage of Dhmem's sharing optimizations and need to fall back to serialization at the cost of performance.

Future Work. In the future, we plan to extend Dhmem support for cross-workflow integration. The approach of sharing data structures between tasks in a workflow extends naturally to sharing between different types of workflow system with a minimal amount of interfacing code. Each workflow system only needs to use Dhmem to enable them to communicate together. As a second direction, we plan to evaluate Dhmem for real-world HPC workflows and assess its effect on the performance.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

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

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[3] M. Belkin, R. Haas, G. W. Arnold, H. W. Leong, E. A. Huerta, D. Lesny, and M. Neubauer. Container solutions for HPC systems: a case study of using Shifter on Blue Waters. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pp. 1–8. 2018.

[4] L. Benedicic, M. Gila, S. Alam, and T. Schulthess. Opportunities for container environments on Cray XC30 with GPU devices. In *Cray Users Group Conference (CUG16)*, 2016.

[5] R. Chamberlain and J. Schommer. Using Docker to support reproducible research. *DOI: https://doi. org/10.6084/m9. figshare*, 1101910:44, 2014.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pp. 1–3, 2010.

[7] J. Chen, Q. Guan, Z. Zhang, X. Liang, L. Vernon, A. McPherson, L.-T. Lo, P. Grubel, T. Randles, Z. Chen, et al. BeeFlow: A workflow management system for in situ processing across HPC and Cloud systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1029–1038. IEEE, 2018.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, 2005.

[9] J. Cook. *Docker for Data Science: Building Scalable and Extensible Data Infrastructure Around the Jupyter Notebook Server*. Apress, 2017.

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

[11] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 28(17):4385–4404, 2016.

[12] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf. Damaris: Addressing performance variability in data management for post-petascale simulations. *ACM Transactions on Parallel Computing (TOPC)*, 3(3):1–43, 2016.

[13] M. Dreher and T. Peterka. Decaf: Decoupled dataflows for in situ high-performance workflows. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[14] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa. Process-in-process: techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 131–143, 2018.

[15] J. Kowalkowski, A. Lyon, and M. Paterno. Study of a Docker use-case for HEP. Technical report, Fermi National Accelerator Lab, Batavia, IL, United States, 2016.

[16] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[17] E. Le and D. Paz. Performance analysis of applications using Singularity container on SDSC Comet. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, pp. 1–4. 2017.

[18] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the Adaptable IO System (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pp. 15–24, 2008.

[19] S. Martín-Santana, C. J. Pérez-González, M. Colebrook, J. L. Roda-García, and P. González-Yanes. Deploying a scalable data science environment using Docker. In *Data Science and Digital Business*, pp. 121–146. Springer, 2019.

[20] D. Morozov and Z. Lukic. Master of puppets: Cooperative multitasking for in situ processing. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 285–288, 2016.

[21] S. Perarnau, J. A. Zounmevo, M. Dreher, B. C. Van Essen, R. Gioiosa, K. Iskra, M. B. Gokhale, K. Yoshii, and P. Beckman. Argo NodeOS: Toward unified resource management for exascale. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 153–162. IEEE, 2017.

[22] T. Peterka, D. Bard, J. C. Bennett, E. W. Bethel, R. A. Oldfield, L. Pouchard, C. Sweeney, and M. Wolf. Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources. *The International Journal of High Performance Computing Applications*, p. 1094342020913628, 2020.

[23] S. Shudler, N. Ferrier, J. Insley, M. E. Papka, and S. Rizzi. Spack meets Singularity: creating movable in-situ analysis stacks with ease. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pp. 34–38, 2019.

[24] J. Zhang, X. Lu, and D. K. Panda. Is singularity-based container technology ready for running MPI applications on HPC clouds? In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pp. 151–160, 2017.

[25] C. Zheng and D. Thain. Integrating containers into workflows: A case study using Makeflow, Work Queue, and Docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, pp. 31–38, 2015.