Dhmem: Shared-Memory Communication for Containerized Workflows

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

Overview

- Background
  - Workflow systems, containerization, and shared memory
- Dhmem
  - Architecture, usage, and results
- Future Opportunities
- Resources & Links

Background

Background

- Scientific workflows fall into two primary categories:
  - Distributed: Different machines; different times; Ex. Pegasus, Parsl
  - In situ: Same machines; same time; Ex. ADIOS, Decaf, Henson, Damaris
- Dependency management is challenging:
  - Complex environments often with competing packages
- Containers offer a solution:
  - Each task can be compiled into different container with their own dependencies
  - Distributed: Tasks compiled once and reused in different workflows; file communication is fast
  - In situ: Tasks need to be compiled separately for each workflow; communication overhead

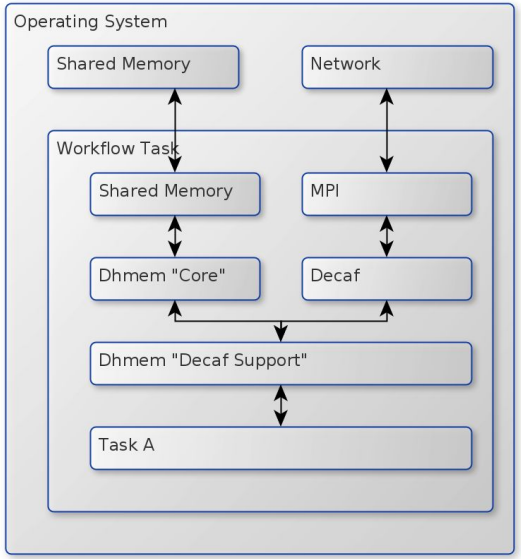
Background

- Improving the communication performance of containerized in situ workflows is the goal
  - Shared memory is the key
- Shared memory improves performance by allocating data structures once
  - Contrast with MPI
  - Difficult for individually containerized tasks
- By abstracting the usage of shared memory, it:
  - Can be easily integrated into workflow systems
  - Can be used identically within or without containers
  - Can offer scalable performance between tasks

Dhmem

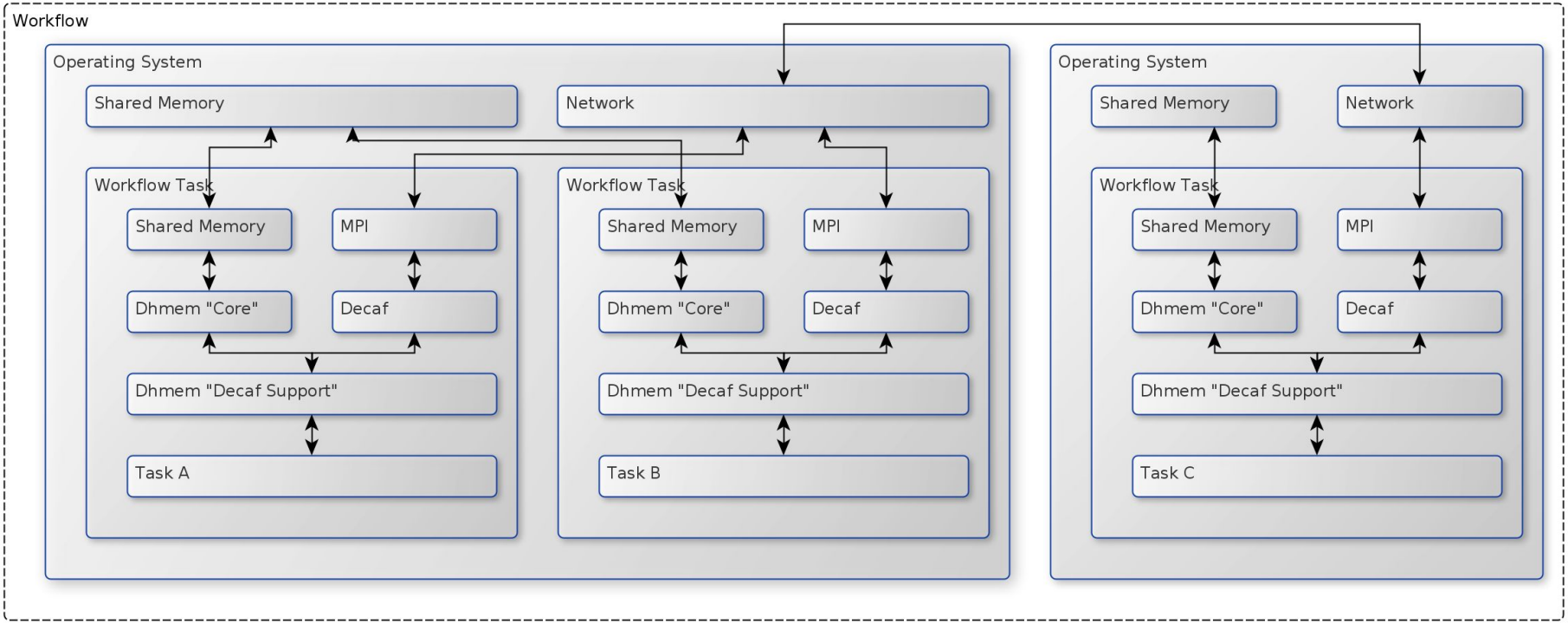
Dhmem

- We created a library called Dhmem to solve the problem of:
  - Shared-memory communication between containerized in situ workflow tasks
- Dhmem is based on the design principles of enabling:
  - Unified model for shared-memory data
  - Position-independent data structures
  - Ease of integration with existing workflow systems
- Split into two parts:
  - The “core” Dhmem library for working with shared memory
  - Several “support” libraries for bridging workflow systems with the core library
- Currently supports:
  - Decaf[†], Henson[‡], and a standalone mode without a workflow system



[†] *Dreher, Matthieu, and Tom Peterka. “Decaf: Decoupled dataflows for in situ high-performance workflows.” 2017.*

[‡] *Morozov, Dmitriy, and Zarija Lukic. "Master of puppets: Cooperative multitasking for in situ processing." 2016.*



Dhmem Architecture Overview

Usage: Allocating Data into Shared Memory

- Dhmem “Core” primarily:
  - Allocates data structures
  - Shares data structure pointers
- There are two modes of allocation:
  - “Simple” allocations for primitives
  - “Container” allocations for data structures (i.e. vectors and maps)
- User code should use references (`int &n`) instead of bare variables (`int n`)

```
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;
```

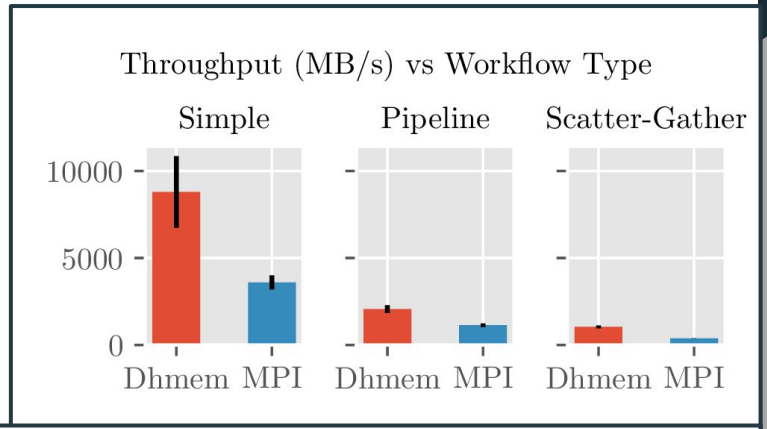
```
dhmem::vector<int> &v =
  dhmem.container<dhmem::vector<int>>("my_v");
v.push_back(5);  // allocates more shared memory

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);
```

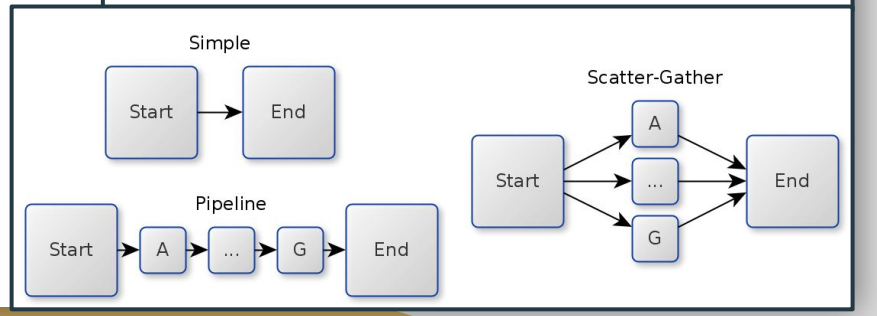
```
1  // send data
2  int &n = dhmem.simple<int>("my_n");
3  SharedField<int> nfield(n, dhmem);
4  pConstructData out_msg;
5  out_msg->appendData("n", nfield);
6  decaf->put(out_msg, "out_port");
7
8  // receive data
9  std::vector<pConstructData> in_msgs;
10 decaf->get(in_msgs, "in_port");
11 SharedField<int> field =
12   in_msgs[0]->getFieldData<SharedField<int>>("n");
13 int &also_n = field.getData(dhmem);
```

Example: Dhmem + Decaf integration that mimics original Decaf API.

Results: Fast Communication between Tasks

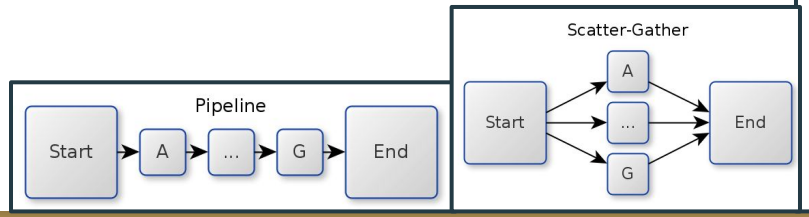
- Dhmem is intended as a replacement for MPI in certain situations
  - Hence, we evaluated performance differences
- The test was designed as:
  - Number of workflow tasks: 2 or more
  - Data being transferred: 10 MB of integers
  - Number of workflow iterations: 1024 and 2048
  - Number of trials: 5
  - Reported metric: average throughput
- Three different types of workflows are tested
  - Simple, pipeline, and scatter-gather

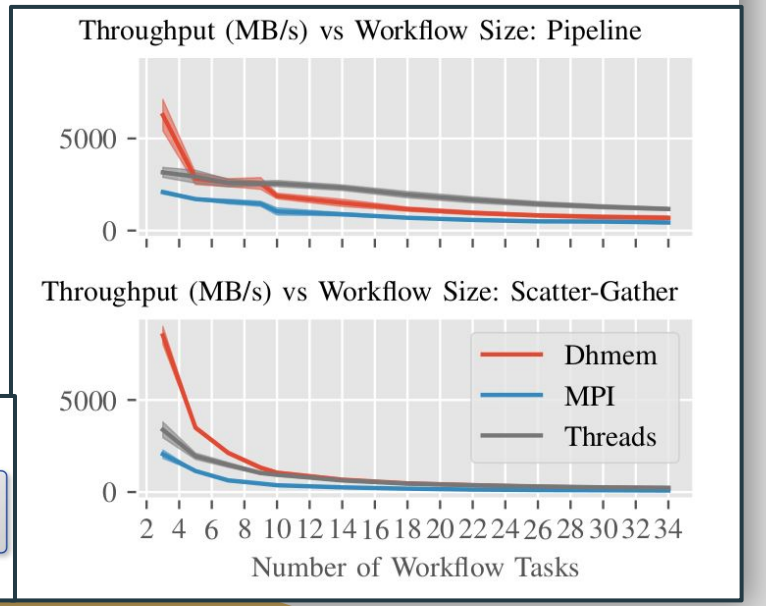




Results: Scalable to Many Tasks

- Dhmem is intended to scale to many tasks
- Setup same as before, but:
  - Varied the number of tasks for pipeline and scatter-gather
  - Compare multi-process Dhmem with multi-process MPI
  - Compare multi-process Dhmem with multi-thread Dhmem
- Dhmem provides better performance than MPI in each test





Opportunities

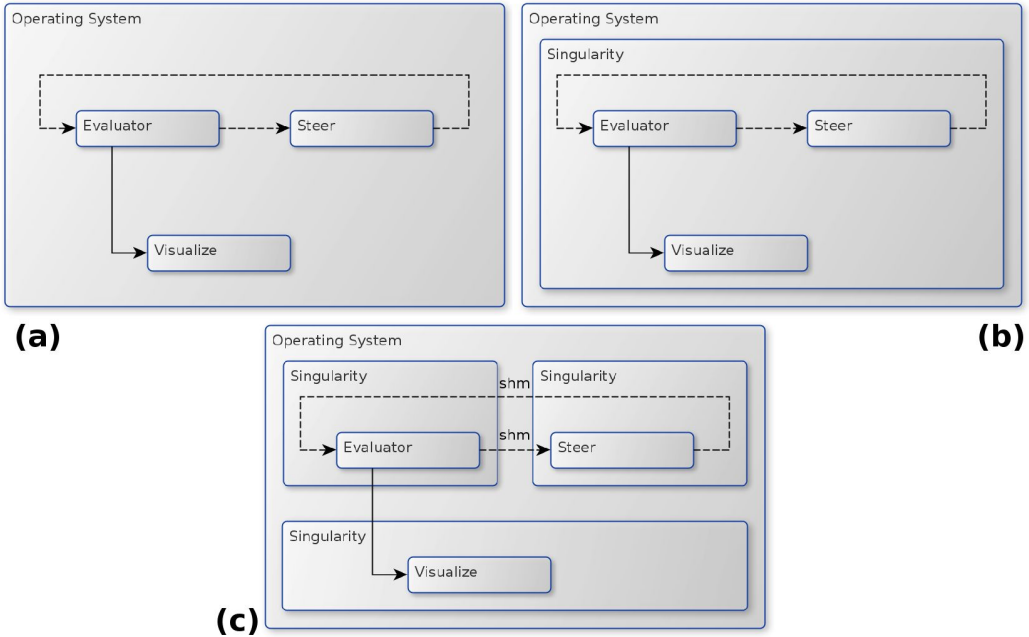
Opportunities

- Different Workflow Task Lifetimes
  - Being able to keep one task running long before or long after the workflow has benefits
  - Consider: TensorBoard for TensorFlow
- Dynamic Data Access
  - Variables can only be accessed in other tasks if they have been exported
  - Consider: Any calculated variable within the task is accessible in any other task
- More Workflow System Integrations
  - Currently integrating two workflow systems requires one to embed the other, or both to be aware of each other
  - Consider: Integration could be as simple as integrating each with Dhmem

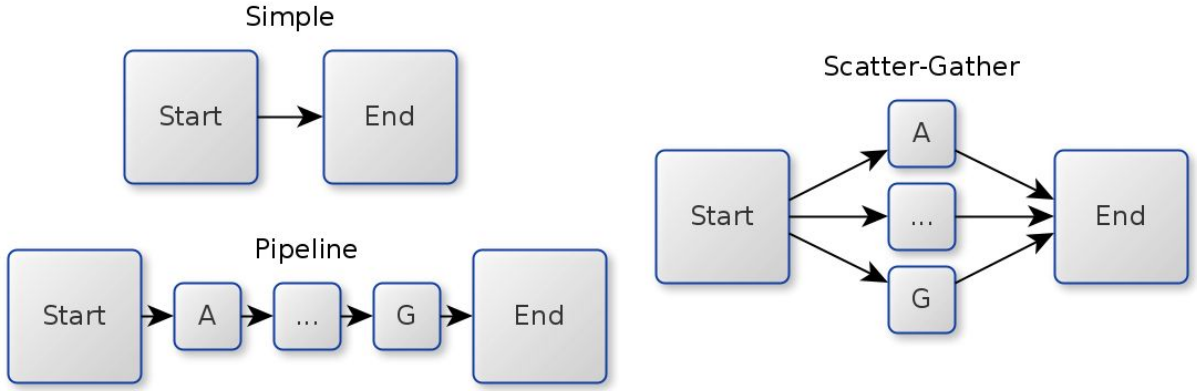
Conclusion & More Info

- Workflow tasks increasingly use containers to manage dependencies
  - This is at odds with in situ workflows due to communication overheads
- Communication can be more performant using shared memory
  - This is at odds with using containers due to isolation
- Dhmem is a library that makes it easy to use shared memory and containers together for workflows
  - There are several exciting opportunities for collaboration with others
- Code Repository: https://bitbucket.org/seelabutk/dhmem
- *Email thobson2@vols.utk.edu for a copy of the paper*

Extras



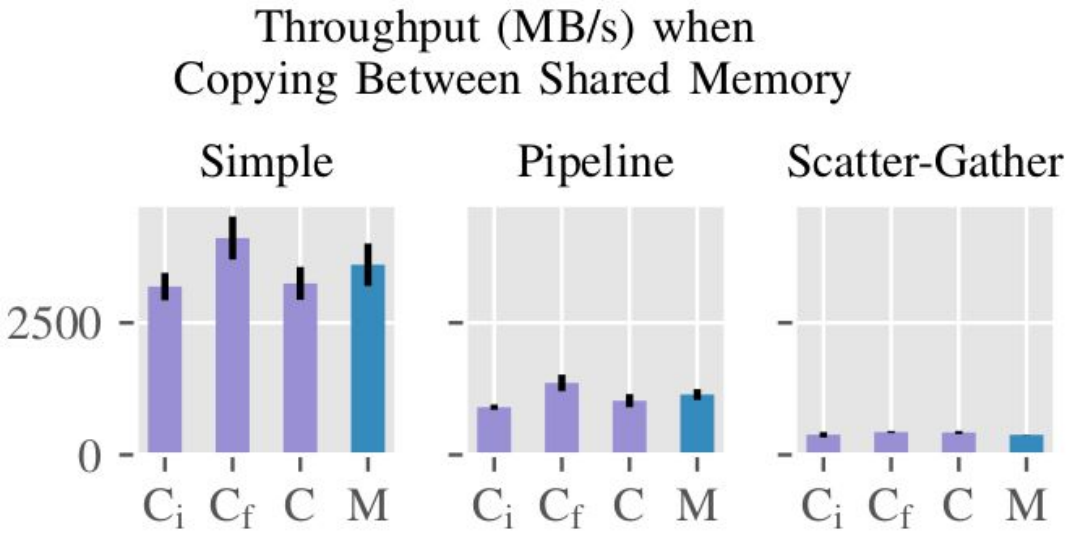
Modes of containerization. (a) is without containers. (b) is with one container for the whole workflow. (c) is with one container for each task.



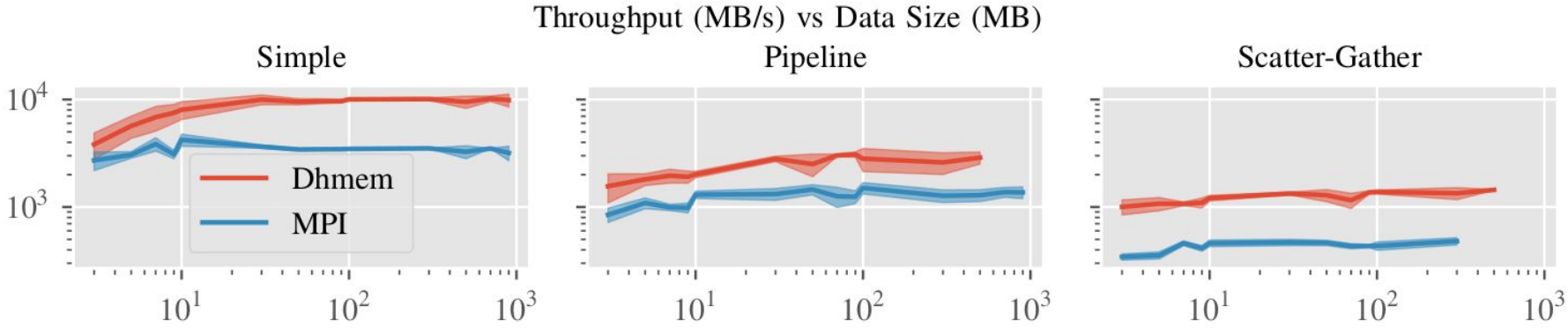
The three types of workflows evaluated with Dhmem. Pipeline and scatter-gather can vary the number of processes.

```
+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);
```

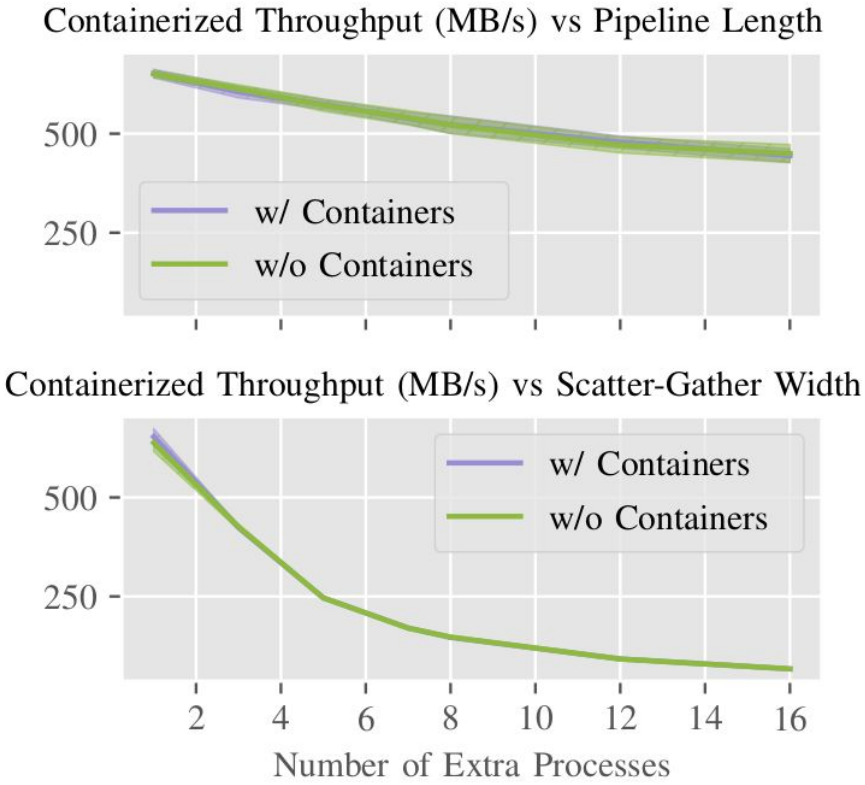
A comparison of sending data via MPI (red, “-” prefix) and via Dhmem (green, “+” prefix).



A simplistic adaptation of MPI communication to Dhmem is to simply copy data from virtual memory into shared memory and vice versa. C_i is “copy into shared-memory.” C_f is “copy from.” C is “copy from and into.” M is “MPI.”



A comparison between Dhmem and MPI performance as the size of the data is varied.



A comparison of Dhmem performance with and without using Singularity containers.

```
1  // send data
2  int &n = dhmem.simple<int>("my_n");
3  SharedField<int> nfield(n, dhmem);
4  pConstructData out_msg;
5  out_msg->appendData("n", nfield);
6  decaf->put(out_msg, "out_port");
7
8  // receive data
9  std::vector<pConstructData> in_msgs;
10 decaf->get(in_msgs, "in_port");
11 SharedField<int> field =
12   in_msgs[0]->getFieldData<SharedField<int>>("n");
13 int &also_n = field.getData(dhmem);
```

Example of using Dhmem with the Decaf workflow system.

```
// 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);
```

Example of using Dhmem with the Henson workflow system.

```
1  // send data
2  dhmem::Port port =
3    dhmem.port("n_port", dhmem::producer, dhmem::mpi);
4  int &n = dhmem.simple<int>("my_n");
5  port.send(n);
6
7  // receive data
8  dhmem::Port port =
9    dhmem.port("n_port", dhmem::consumer, dhmem::mpi);
10 int &n = port.recv<int>();
```

Example of using Dhmem without a workflow system.