

Memory Allocation Kinds  
An MPI Side Document  
Version 1.0

MPI Forum Hybrid and Accelerator Working Group  
(XXXX 2023)

This document defines memory allocation kinds that are compatible with the MPI-4.1 standard.

Version 1.0: XXXX 2023 This document defines the first set of memory allocation kinds. This and future versions of this side document to the MPI standard are ratified by the MPI Forum, but not an official part of the standard itself.

# Acknowledgments

This document represents the work of many people who have served on the MPI Forum Hybrid and Accelerator Working Group. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the Memory Allocation Kinds document. The technical development was carried out by subgroups, whose work was reviewed by the full committee.

Those who served as primary coordinators in Version 1.0 are:

- James Dinan, Rohit Zambre, NVIDIA CUDA
- Pedram Alizadeh, Edgar Gabriel, Michael Klemm, AMD ROCm
- Maria Garzaran, Dan Holmes, Intel Level Zero

The following list includes some of the active participants in the MPI Forum Hybrid and Accelerator Working Group.

Pedram Alizadeh	James Dinan	Dmitry Durnov	Edgar Gabriel
Maria Garzaran	Yanfei Guo	Khaled Hamidouche	Dan Holmes
Michael Klemm	Michael Knobloch	Guillaume Mercier	Christoph Niethammer
Howard Pritchard	Ken Raffenetti	Joseph Schuchart	Amirhossein Sojoodi
Yiltan Temuçin	Rohit Zambre	Hui Zhou	

The following institutions supported the Version 1.0 effort of the Memory Allocation Kinds through time and travel support for the people listed above.

Advanced Micro Devices, Inc.  
Amazon.com, Inc  
Argonne National Laboratory  
Jülich Supercomputing Center  
HLRS, University of Stuttgart  
Institut National de Recherche en Informatique et Automatique (Inria)  
Intel Corporation  
Los Alamos National Laboratory  
NVIDIA Corporation  
Queen's University  
University of Tennessee, Knoxville

# Contents

<b>Acknowledgments</b>	<b>iv</b>
<b>1 Overview</b>	<b>1</b>
<b>2 Definitions</b>	<b>2</b>
2.1 CUDA memory kind . . . . .	2
Restrictors . . . . .	2
2.2 ROCm memory kind . . . . .	2
Restrictors . . . . .	2
2.3 Level Zero memory kind . . . . .	3
Restrictors . . . . .	3
<b>3 Examples</b>	<b>4</b>
3.1 MPI plus SYCL . . . . .	4
3.2 MPI plus CUDA . . . . .	10
3.3 MPI plus ROCm . . . . .	14
<b>Bibliography</b>	<b>17</b>



# 1 Chapter 1

## 2 Overview

3 Modern computing systems contain a variety of memory types, each closely associated with  
4 a distinct type of computing hardware. For example, compute accelerators such as GPUs  
5 typically feature their own memory that is distinct from the memory attached to the host  
6 processor. Additionally, GPUs from different vendors also differ in their memory types.  
7 The differences in memory types influence feature availability and performance behavior of  
8 an application running on such modern systems. Hence, MPI libraries need to be aware  
9 of and support additional memory types. For a given type of memory, MPI libraries need  
10 to know the associated memory allocator, the memory's properties, and the methodologies  
11 to access the memory. The different memory kinds capture the differentiating information  
12 needed by MPI libraries for different memory types.

13 This MPI side document defines the memory allocation kinds and their associated  
14 restrictors that users can use to query the support for different memory kinds provided by  
15 the MPI library. These definitions supplement those found in section 11.4.3 of the MPI-4.1  
16 standard, which also explains their usage model.

# Chapter 2

## Definitions

This chapter contains definitions of memory allocation kinds and their restrictors for different memory types.

### 2.1 CUDA memory kind

We define `cuda` as a memory kind that refers to the memory allocated by the CUDA runtime system [1]. Examples 3.1 and 3.2 showcase its usage.

#### Restrictors

- `host`: Support for memory allocations on the host system that are page-locked for direct access from the CUDA device (e.g., memory allocations from the `cudaHostAlloc()` function). These memory allocations are attributed with `cudaMemoryTypeHost`.
- `device`: Support for memory allocated on a CUDA device (e.g., memory allocations from the `cudaMalloc()` function). These memory allocations are attributed with `cudaMemoryTypeDevice`.
- `managed`: Support for memory that is managed by CUDA's Unified Memory system (e.g., memory allocations from the `cudaMallocManaged()` function). These memory allocations are attributed with `cudaMemoryTypeManaged`.

### 2.2 ROCm memory kind

We define `rocm` as a memory kind that refers to the memory allocated by the ROCm runtime system [2]. Examples 3.1 and 3.3 showcases its usage.

#### Restrictors

- `host`: Support for memory allocated on the host system that is page-locked for direct access from the ROCm device (e.g., memory allocations from the `hipHostMalloc()` function).
- `device`: Support for memory allocated on the ROCm device (e.g., memory allocations from the `hipMalloc()` function).



- 1     • managed: Support for memory that is managed automatically by the ROCm runtime  
2       (e.g., memory allocations from the `hipMallocManaged()` function).

## 3   2.3   Level Zero memory kind

4   We define `level_zero` as a memory kind that refers to the memory allocated by the Level Zero  
5   runtime system [3]. Example 3.1 showcases its usage.

### 6   Restrictors

- 7     • host: Support for memory that is owned by the host and is accessible by the host and  
8       by any Level Zero devices.
- 9     • device: Support for memory that is owned by a specific Level Zero device.
- 10    • shared: Support for memory that has shared ownership between the host and one or  
11      more Level Zero devices.

# Chapter 3

## Examples

This chapter includes examples demonstrating the usage of memory kinds defined in Chapter 2.

### 3.1 MPI plus SYCL

**Example 3.1** This SYCL example demonstrates the usage of the different memory allocation kinds to perform communication in a manner that is supported by the underlying MPI library.

```
9
10 #include <iostream>
11 #include <optional>
12 #include <sycl.hpp>
13 #include <mpi.h>
14
15 enum class InteractionMethod
16 {
17     begin = -1,
18
19     // most preferred
20     ComputeUsingQueue_CommunicationUsingDeviceMemory,
21     ComputeUsingQueue_CommunicationUsingSharedMemory,
22     ComputeUsingQueue_CommunicationUsingHostMemory,
23
24     ComputeWithoutQueue_CommunicationUsingSystemMemory,
25     // least preferred
26
27     end
28 };
29
30 int main(int argc, char* argv[]) {
31     try {
32         sycl::queue q; // might use a CPU or a GPU or an FPGA, etc
33
34         // information for the user only
35         std::cout << "SYCL reports device name: "
36                 << q.get_device().get_info<sycl::info::device::name>()
37                 << std::endl;
```

```

1      std::cout << "SYCL reports device backend: "
2          << q.get_backend() << std::endl;
3
4      // query SYCL for the backend and the features it supports
5      const auto [qBackendEnum, qSupportsDeviceMem,
6          qSupportsSharedUSM, qSupportsHostUSM] =
7      [&q]() {
8          const sycl::device& dev = q.get_device();
9          return std::make_tuple(
10              q.get_backend(),
11              dev.has(sycl::aspect::usm_device_allocations),
12              dev.has(sycl::aspect::usm_shared_allocations),
13              dev.has(sycl::aspect::usm_host_allocations)
14          );
15      }();
16
17      // translate the backend reported by the SYCL queue
18      // into a "memory allocation kind" string for MPI
19      // and the feature support reported by the SYCL queue
20      // into "memory allocation restrictor" strings for MPI
21      const auto [queue_uses_backend_defined_by_mpi,
22          backend_from_sycl_translated_for_mpi,
23          valid_mpi_restrictors_for_backend] = [qBackendEnum] () {
24          typedef struct { bool known; std::string kind; struct {
25              std::string device;
26              std::string sharedOrManaged;
27              std::string host; } restrictors; } retType;
28          switch (qBackendEnum) {
29              case sycl::backend::ext_oneapi_level_zero:
30                  return retType{ true, "level_zero",
31                      {"device", "shared", "host"} };
32                  break;
33              case sycl::backend::ext_oneapi_cuda:
34                  return retType { true, "cuda",
35                      {"device", "managed", "host"} };
36                  break;
37              case sycl::backend::ext_oneapi_hip:
38                  return retType { true, "rocm",
39                      {"device", "managed", "host"} };
40                  break;
41              default:
42                  // means fallback to using "system" memory kind for MPI
43                  return retType{ false };
44                  break;
45          }
46      }();
47      std::cout << "SYCL queue backend ('" << qBackendEnum
48          << "')", translated for MPI: "
49          << (queue_uses_backend_defined_by_mpi
50              ? backend_from_sycl_translated_for_mpi
51              : "NOT DEFINED BY MPI (will tell MPI 'system')")
52          << std::endl;
53
54      MPI_Session session = MPI_SESSION_NULL;

```

```

1      MPI_Comm comm = MPI_COMM_NULL;
2      int my_rank = MPI_PROC_NULL;
3
4      // repeatedly request memory allocation kind:restricor support
5      // in preference order until we find an overlap
6      // between what the SYCL backend supports and what MPI provides
7      InteractionMethod method;
8      for (method = InteractionMethod::begin;
9           method < InteractionMethod::end;
10          method = static_cast<InteractionMethod>((size_t)method) + 1)) {
11
12
13      const auto requested_mem_kind_for_mpi =
14      [=]() -> std::optional<std::string> {
15          switch (method) {
16              case InteractionMethod
17                  ::ComputeUsingQueue_CommunicationUsingDeviceMemory:
18                  if (!queue_uses_backend_defined_by_mpi)
19                      // method cannot work because
20                      // MPI does not define this backend
21                      return std::nullopt;
22                  else if (!qSupportsDeviceMem)
23                      // method cannot work
24                      // SYCL queue does not support this memory kind
25                      return std::nullopt;
26                  else
27                      return backend_from_sycl_translated_for_mpi +
28                          ":" + valid_mpi_restricors_for_backend
29                          .device;
30                  break;
31              case InteractionMethod
32                  ::ComputeUsingQueue_CommunicationUsingSharedMemory:
33                  if (!queue_uses_backend_defined_by_mpi)
34                      // method cannot work because
35                      // MPI does not define this backend
36                      return std::nullopt;
37                  else if (!qSupportsSharedUSM)
38                      // method cannot work
39                      // SYCL queue does not support this memory kind
40                      return std::nullopt;
41                  else
42                      return backend_from_sycl_translated_for_mpi +
43                          ":" + valid_mpi_restricors_for_backend
44                          .sharedOrManaged;
45                  break;
46              case InteractionMethod
47                  ::ComputeUsingQueue_CommunicationUsingHostMemory:
48                  if (!queue_uses_backend_defined_by_mpi)
49                      // method cannot work because
50                      // MPI does not define this backend
51                      return std::nullopt;
52                  else if (!qSupportsHostUSM)
53                      // method cannot work
54                      // SYCL queue does not support this memory kind

```

```

1         return std::nullopt;
2     else
3         return backend_from_sycl_translated_for_mpi +
4             ":" + valid_mpi_restrictors_for_backend
5                 .host;
6
7         break;
8     case InteractionMethod
9         ::ComputeWithoutQueue_CommunicationUsingSystemMemory:
10        // this method MUST work because the "system" memory
11        // kind must be provided by MPI when requested
12        return "system";
13        break;
14
15    case InteractionMethod::begin:
16    case InteractionMethod::end:
17    default:
18        return std::nullopt;
19    }
20 }();
21 if (!requested_mem_kind_for_mpi.has_value())
22     continue; // this method cannot work, try the next one
23
24 MPI_Info info = MPI_INFO_NULL;
25 std::string key_for_mpi("mpi_memory_alloc_kinds");
26
27 // usage mode: REQUESTED
28 MPI_Info_create(&info);
29 MPI_Info_set(info, key_for_mpi.c_str(),
30             requested_mem_kind_for_mpi.value().c_str());
31 MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
32 MPI_Info_free(&info);
33 std::cout << "Created a session, requested memory kind: "
34           << requested_mem_kind_for_mpi.value()
35           << std::endl;
36
37 // usage mode: PROVIDED
38 bool provided = false;
39 if (requested_mem_kind_for_mpi.value() == "system") {
40     // kind "system" must be provided by MPI when requested
41     provided = true; // we have a winner: exit the for loop
42 } else {
43     MPI_Session_get_info(session, &info);
44     int len = 0, flag = 0;
45     MPI_Info_get_string(info, key_for_mpi.c_str(), &len,
46                        nullptr, &flag);
47     if (flag && len > 0) {
48         size_t num_bytes_needed = (size_t)len*sizeof(char);
49         char* val = static_cast<char*>(
50             malloc(num_bytes_needed));
51         if (nullptr == val) std::terminate();
52         MPI_Info_get_string(info, key_for_mpi.c_str(),
53                            &len, val, &flag);
54         std::string val_from_mpi(val);
55         std::cout << "looking for substring: "

```

```

1         << requested_mem_kind_for_mpi.value()
2         << std::endl;
3         std::cout << "within value from MPI: "
4         << val_from_mpi << std::endl;
5         if (std::string::npos != val_from_mpi.find(
6             requested_mem_kind_for_mpi.value())) {
7             provided = true; // we have a winner: assert
8         } else {
9             std::cout << "Not found -- this MPI_Session"
10             + "does NOT provide the requested"
11             + "support!" << std::endl;
12         }
13         free(val);
14     } else {
15         std::cout << "Info key '" << key_for_mpi << "' "
16         + "not found in MPI_Info from session!"
17         << std::endl;
18     }
19     MPI_Info_free(&info);
20 }
21 if (!provided)
22     MPI_Session_Finalize(&session);
23 else {
24     // usage mode: ASSERTED
25     std::string assert_key_for_mpi(
26         "mpi_assert_memory_alloc_kinds");
27     std::cout << "MPI says it provides the requested memory"
28     + " kind ("
29     << requested_mem_kind_for_mpi.value()
30     << ")--will assert during MPI_Comm creation"
31     << std::endl;
32     MPI_Info_create(&info);
33     MPI_Info_set(info, assert_key_for_mpi.c_str(),
34         requested_mem_kind_for_mpi.value().c_str());
35
36     MPI_Group world_group = MPI_GROUP_NULL;
37     std::string pset_for_mpi("mpi://world");
38     MPI_Group_from_session_pset(session,
39         pset_for_mpi.c_str(), &world_group);
40     std::string tag_for_mpi("org.mpi-forum.mpi-side-doc."
41         + "mem-alloc-kinds.sycl-example");
42     MPI_Comm_create_from_group(world_group,
43         tag_for_mpi.c_str(), info,
44         MPI_ERRORS_ARE_FATAL, &comm);
45     MPI_Group_free(&world_group);
46     MPI_Comm_rank(comm, &my_rank);
47
48     break;
49 }
50 } // end of 'for (InteractionMethod)'
51 if (MPI_SESSION_NULL == session) {
52     std::cout << "FAILED to create a usable MPI session"
53     << std::endl; // (should not happen)
54     std::terminate();

```

```

1     } else
2         std::cout << "SUCCESS -- for this session, MPI says the"
3             + " requested memory kind is provided"
4             << std::endl;
5
6     // allocate a data buffer on GPU or CPU
7     int* data_buffer = [&q, &method, &my_rank] {
8         switch (method) {
9             case InteractionMethod
10                 ::ComputeUsingQueue_CommunicationUsingDeviceMemory:
11                 std::cout << "[rank:" << my_rank << "]" MPI says this"
12                     + " communicator can accept device memory --"
13                     + " allocating memory on device"
14                     << std::endl;
15                 return malloc_device<int>(6, q);
16                 break;
17             case InteractionMethod
18                 ::ComputeUsingQueue_CommunicationUsingSharedMemory:
19                 std::cout << "[rank:" << my_rank << "]" MPI says this"
20                     + " communicator can accept shared/managed"
21                     + " memory -- allocating USM shared memory"
22                     << std::endl;
23                 return malloc_shared<int>(6, q);
24                 break;
25             case InteractionMethod
26                 ::ComputeUsingQueue_CommunicationUsingHostMemory:
27                 std::cout << "[rank:" << my_rank << "]" MPI says this"
28                     + " communicator can accept host memory --"
29                     + " allocating USM host memory" << std::endl;
30                 return malloc_host<int>(6, q);
31                 break;
32             case InteractionMethod
33                 ::ComputeWithoutQueue_CommunicationUsingSystemMemory:
34                 std::cout << "[rank:" << my_rank << "]" MPI says this"
35                     + " communicator CANNOT accept device memory"
36                     + " -- allocating memory on system"
37                     << std::endl;
38                 return static_cast<int*>(malloc(6 * sizeof(int)));
39                 break;
40
41             case InteractionMethod::begin:
42             case InteractionMethod::end:
43             default:
44                 std::cout << "ERROR: invalid interaction method"
45                     << std::endl; // (should not happen)
46                 std::terminate();
47                 break;
48         }
49     }();
50
51     // define a simple work task for GPU or CPU
52     auto do_work = [=]() {
53         for (int i = 0; i < 6; ++i)
54             data_buffer[i] = (my_rank + 1) * 7;

```

```

1         };
2
3         // execute the work task using the data buffer on GPU or CPU
4         if (method != InteractionMethod
5             ::ComputeWithoutQueue_CommunicationUsingSystemMemory) {
6             q.submit([&](sycl::handler& h) {
7                 h.single_task(do_work);
8             }).wait_and_throw();
9             std::cout << "[rank:" << my_rank << "]" finished work on GPU"
10                << std::endl;
11        } else {
12            do_work();
13            std::cout << "[rank:" << my_rank << "]" finished work on CPU"
14                << std::endl;
15        }
16
17        MPI_Allreduce(MPI_IN_PLACE, data_buffer, 6, MPI_INT, MPI_MAX,
18                      comm);
19        std::cout << "[rank:" << my_rank << "]" finished reduction"
20            << std::endl;
21
22        MPI_Comm_disconnect(&comm);
23        MPI_Session_Finalize(&session);
24
25        int answer = std::numeric_limits<int>::max();
26        if (method == InteractionMethod
27            ::ComputeUsingQueue_CommunicationUsingDeviceMemory) {
28            q.memcpy(&answer, &data_buffer[0], sizeof(int))
29                .wait_and_throw();
30        } else {
31            answer = data_buffer[0];
32        }
33        std::cout << "[rank:" << my_rank << "]" The answer is: "
34            << answer << std::endl;
35        if (method != InteractionMethod
36            ::ComputeWithoutQueue_CommunicationUsingSystemMemory) {
37            free(data_buffer, q);
38        } else {
39            free(data_buffer);
40        }
41    }
42    catch (sycl::exception const& e) {
43        std::cout << "An exception was caught.\n";
44        std::terminate();
45    }
46    return 0;
47 }

```

## 3.2 MPI plus CUDA

**Example 3.2** This CUDA example demonstrates the usage of the different kinds to perform communication in a manner that is supported by the underlying MPI library.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5  #include <mpi.h>
6  #include <cuda_runtime.h>
7
8  #define CUDA_CHECK(mpi_comm, call) \
9      { \
10         const cudaError_t error = call; \
11         if (error != cudaSuccess) \
12         { \
13             fprintf(stderr, \
14                 "An error occurred: \"%s\" at %s:%d\n", \
15                 cudaGetErrorString(error), \
16                 __FILE__, __LINE__); \
17             MPI_Abort(mpi_comm, error); \
18         } \
19     }
20
21 int main(int argc, char *argv[])
22 {
23     int cuda_device_aware = 0;
24     int cuda_managed_aware = 0;
25     int len = 0, flag = 0;
26     int *managed_buf = NULL;
27     int *device_buf = NULL, *system_buf = NULL;
28     int nranks = 0;
29     MPI_Info info;
30     MPI_Session session;
31     MPI_Group wgroup;
32     MPI_Comm system_comm;
33     MPI_Comm cuda_managed_comm = MPI_COMM_NULL;
34     MPI_Comm cuda_device_comm = MPI_COMM_NULL;
35
36     // Usage mode: REQUESTED
37     MPI_Info_create(&info);
38     MPI_Info_set(info, "mpi_memory_alloc_kinds",
39                     "system,cuda:device,cuda:managed");
40     MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
41     MPI_Info_free(&info);
42
43     // Usage mode: PROVIDED
44     MPI_Session_get_info(session, &info);
45     MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
46                         &len, NULL, &flag);
47
48     if (flag) {
49         char *val, *valptr, *kind;
50
51         val = valptr = (char *) malloc(len);
52         if (NULL == val) return 1;
53
54         MPI_Info_get_string(info, "mpi_memory_alloc_kinds",

```

```

1         &len, val, &flag);
2
3     while ((kind = strsep(&val, ",")) != NULL) {
4         if (strcasecmp(kind, "cuda:managed") == 0) {
5             cuda_managed_aware = 1;
6         }
7         else if (strcasecmp(kind, "cuda:device") == 0) {
8             cuda_device_aware = 1;
9         }
10    }
11    free(valptr);
12}
13
14MPI_Info_free(&info);
15
16MPI_Group_from_session_pset(session, "mpi://WORLD" , &wgroup);
17
18// Create a communicator for operations on system memory
19// Usage mode: ASSERTED
20MPI_Info_create(&info);
21MPI_Info_set(info, "mpi_assert_memory_alloc_kinds", "system");
22MPI_Comm_create_from_group(wgroup,
23    "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.system",
24    info, MPI_ERRORS_ABORT, &system_comm);
25MPI_Info_free(&info);
26
27MPI_Comm_size(system_comm, &n ranks);
28
29/** Check for CUDA awareness */
30
31// Note: MPI does not require homogeneous support
32// across all processes for memory allocation kinds.
33// This example chooses to use
34// CUDA managed allocations (or device allocations)
35// only when all processes report it is supported.
36
37// Check if all processes have CUDA managed support
38MPI_Allreduce(MPI_IN_PLACE, &cuda_managed_aware, 1, MPI_INT,
39    MPI_LAND, system_comm);
40
41if (cuda_managed_aware) {
42    // Create a communicator for operations that use
43    // CUDA managed buffers.
44    // Usage mode: ASSERTED
45    MPI_Info_create(&info);
46    MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
47        "cuda:managed");
48    MPI_Comm_create_from_group(wgroup,
49        "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.managed",
50        info, MPI_ERRORS_ABORT, &cuda_managed_comm);
51    MPI_Info_free(&info);
52}
53else {
54    // Check if all processes have CUDA device support

```

```

1      MPI_Allreduce(MPI_IN_PLACE, &cuda_device_aware, 1, MPI_INT,
2                    MPI_LAND, system_comm);
3      if (cuda_device_aware) {
4          // Create a communicator for operations that use
5          // CUDA device buffers.
6          // Usage mode: ASSERTED
7          MPI_Info_create(&info);
8          MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
9                        "cuda:device");
10         MPI_Comm_create_from_group(wgroup,
11                                   "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.device",
12                                   info, MPI_ERRORS_ABORT, &cuda_device_comm);
13         MPI_Info_free(&info);
14     }
15     else {
16         printf("Warning: cuda alloc kind not supported\n");
17     }
18 }
19
20 MPI_Group_free(&wgroup);
21
22 /** Execute according to level of CUDA awareness */
23 if (cuda_managed_aware) {
24     // Allocate managed buffer and initialize it
25     CUDA_CHECK(system_comm,
26               cudaMallocManaged((void*)&managed_buf, sizeof(int),
27                                 cudaMemAttachGlobal));
28     *managed_buf = 1;
29
30     // Perform communication using cuda_managed_comm
31     // if it's available.
32     MPI_Allreduce(MPI_IN_PLACE, managed_buf, 1, MPI_INT,
33                 MPI_SUM, cuda_managed_comm);
34
35     assert((*managed_buf) == nranks);
36
37     CUDA_CHECK(system_comm,
38               cudaFree(managed_buf));
39 }
40 else {
41     // Allocate system buffer and initialize it
42     // (using cudaMallocHost for better performance of cudaMemcpy)
43     CUDA_CHECK(system_comm,
44               cudaMallocHost((void*)&system_buf, sizeof(int)));
45     *system_buf = 1;
46
47     // Allocate CUDA device buffer and initialize it
48     CUDA_CHECK(system_comm,
49               cudaMalloc((void*)&device_buf, sizeof(int)));
50     CUDA_CHECK(system_comm,
51               cudaMemcpyAsync(device_buf, system_buf, sizeof(int),
52                               cudaMemcpyHostToDevice, 0));
53
54     CUDA_CHECK(system_comm,

```

```

1         cudaStreamSynchronize(0));
2     if (cuda_device_aware) {
3         // Perform communication using cuda_device_comm
4         // if it's available.
5         MPI_Allreduce(MPI_IN_PLACE, device_buf, 1, MPI_INT,
6                       MPI_SUM, cuda_device_comm);
7
8         assert((*device_buf) == nranks);
9     }
10    else {
11        // Otherwise, copy data to a system buffer,
12        // use system_comm, and copy data back to device buffer
13        CUDA_CHECK(system_comm,
14                   cudaMemcpyAsync(system_buf, device_buf,
15                                   sizeof(int), cudaMemcpyDeviceToHost, 0));
16
17        CUDA_CHECK(system_comm,
18                   cudaStreamSynchronize(0));
19        MPI_Allreduce(MPI_IN_PLACE, system_buf, 1, MPI_INT,
20                      MPI_SUM, system_comm);
21        CUDA_CHECK(system_comm,
22                   cudaMemcpyAsync(device_buf, system_buf,
23                                   sizeof(int), cudaMemcpyHostToDevice, 0));
24
25        CUDA_CHECK(system_comm,
26                   cudaStreamSynchronize(0));
27        assert((*system_buf) == nranks);
28    }
29
30    CUDA_CHECK(system_comm, cudaFree(device_buf));
31    CUDA_CHECK(system_comm, cudaFreeHost(system_buf));
32 }
33
34 if (cuda_managed_comm != MPI_COMM_NULL)
35     MPI_Comm_disconnect(&cuda_managed_comm);
36 if (cuda_device_comm != MPI_COMM_NULL)
37     MPI_Comm_disconnect(&cuda_device_comm);
38 MPI_Comm_disconnect(&system_comm);
39
40 MPI_Session_finalize(&session);
41
42 return 0;
43 }

```

### 44 3.3 MPI plus ROCm

45 **Example 3.3** This HIP example demonstrates the usage of memory allocation kinds with  
46 MPI File I/O.

```

47 #include <stdio.h>
48 #include <stdlib.h>
49 #include <string.h>
50 #include <assert.h>

```

```

1  #include <mpi.h>
2  #include <hip/hip_runtime_api.h>
3
4  #define HIP_CHECK(condition) { \
5      hipError_t error = condition; \
6      if(error != hipSuccess){ \
7          fprintf(stderr,"HIP error: %d line: %d\n", \
8              error, __LINE__); \
9          MPI_Abort(MPI_COMM_WORLD, error); \
10     } \
11 }
12
13 #define BUFSIZE 1024
14
15 int main(int argc, char *argv[])
16 {
17     int rocm_device_aware = 0;
18     int len = 0, flag = 0;
19     int *device_buf = NULL;
20     MPI_File file;
21     MPI_Status status;
22     MPI_Info info;
23
24     // Usage mode: REQUESTED
25     // Supply mpi_memory_alloc_kinds to the MPI startup
26     // mechanism (not shown)
27     MPI_Init(&argc, &argv);
28
29     // Usage mode: PROVIDED
30     // Query the MPI_INFO object on MPI_COMM_WORLD to
31     // determine whether the MPI library provides
32     // support for the memory allocation kinds
33     // requested via the MPI startup mechanism
34     MPI_Comm_get_info(MPI_COMM_WORLD, &info);
35     MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
36         &len, NULL, &flag);
37     if (flag) {
38         char *val, *valptr, *kind;
39
40         val = valptr = (char *) malloc(len);
41         if (NULL == val) return 1;
42
43         MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
44             &len, val, &flag);
45
46         while ((kind = strsep(&val, ",")) != NULL) {
47             if (strcasecmp(kind, "rocm:device") == 0) {
48                 rocm_device_aware = 1;
49             }
50         }
51         free(valptr);
52     }
53
54     HIP_CHECK(hipMalloc((void**)&device_buf, BUFSIZE * sizeof(int)));

```

```

1
2 // The user could optionally create an info object,
3 // set mpi_assert_memory_alloc_kind to the memory type
4 // it plans to use, and pass this as an argument to
5 // MPI_File_open. This approach has the potential to
6 // enable further optimizations in the MPI library.
7 MPI_File_open(MPI_COMM_WORLD, "inputfile",
8               MPI_MODE_RDONLY, MPI_INFO_NULL, &file);
9
10 if (rocm_device_aware) {
11     MPI_File_read(file, device_buf, BUFSIZE, MPI_INT, &status);
12     printf("Using if part\n");
13 }
14 else {
15     int *tmp_buf;
16     printf("Using else part\n");
17     tmp_buf = (int*) malloc (BUFSIZE * sizeof(int));
18     MPI_File_read(file, tmp_buf, BUFSIZE, MPI_INT, &status);
19
20     HIP_CHECK(hipMemcpyAsync(device_buf, tmp_buf,
21                             BUFSIZE * sizeof(int),
22                             hipMemcpyDefault, 0));
23     HIP_CHECK(hipStreamSynchronize(0));
24
25     free(tmp_buf);
26 }
27
28 // Launch compute kernel(s)
29
30 MPI_File_close(&file);
31 HIP_CHECK(hipFree(device_buf));
32
33 MPI_Finalize();
34 return 0;
35 }

```

# <sup>1</sup> Bibliography

- <sup>2</sup> [1] CUDA Runtime API. <https://docs.nvidia.com/cuda/cuda-runtime-api/>.
- <sup>3</sup> [2] HIP Programming Manual. <https://rocm.docs.amd.com/en/latest/reference/hip.html>.
- <sup>4</sup> [3] Level Zero Programming Guide. [https://spec.oneapi.io/level-](https://spec.oneapi.io/level-zero/latest/core/PROG.html)
- <sup>5</sup> [zero/latest/core/PROG.html](https://spec.oneapi.io/level-zero/latest/core/PROG.html).