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 Hyrbid 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 Hyrbid 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

Chapter 1

Overview

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

This MPI side document defines the memory allocation kinds and their associated restrictors that users can use to query the support for different memory kinds provided by the MPI library. These definitions supplement those found in section 11.4.3 of the MPI-4.1 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).

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

2.3 Level Zero memory kind

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

Restrictors

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

2.4 OpenCL memory kind

We define opencl as a memory kind that refers to the memory allocated by the OpenCL runtime system [?]. Example 3.1 showcases its usage.

Restrictors

- host: Support for memory that is owned by the host and is accessible by the host and by any OpenCL devices.
- device: Support for memory that is owned by a specific OpenCL device.
- shared: Support for memory that has shared ownership between the host and one or more OpenCL 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.

```

#include <iostream>
#include <optional>
#include <sycl.hpp>
#include <mpi.h>

enum class InteractionMethod
{
    begin = -1,

    // most preferred
    ComputeUsingQueue_CommunicationUsingDeviceMemory,
    ComputeUsingQueue_CommunicationUsingSharedMemory,
    ComputeUsingQueue_CommunicationUsingHostMemory,

    ComputeWithoutQueue_CommunicationUsingSystemMemory,
    // least preferred

    end
};

int main(int argc, char* argv[]) {
    try {
        sycl::queue q; // might use a CPU or a GPU or an FPGA, etc

        // information for the user only
        std::cout << "SYCL reports device name: "
            << q.get_device().get_info<sycl::info::device::name>()
            << std::endl;
```

```
        std::cout << "SYCL reports device backend: "
            << q.get_backend() << std::endl;

        // query SYCL for the backend and the features it supports
        const auto [qBackendEnum, qSupportsDeviceMem,
                    qSupportsSharedUSM, qSupportsHostUSM] =
        [&q]() {
            const sycl::device& dev = q.get_device();
            return std::make_tuple(
                q.get_backend(),
                dev.has(sycl::aspect::usm_device_allocations),
                dev.has(sycl::aspect::usm_shared_allocations),
                dev.has(sycl::aspect::usm_host_allocations)
            );
        }();

        // translate the backend reported by the SYCL queue
        // into a "memory allocation kind" string for MPI
        // and the feature support reported by the SYCL queue
        // into "memory allocation restrictor" strings for MPI
        const auto [queue_uses_backend_defined_by_mpi,
                    backend_from_sycl_translated_for_mpi,
            valid_mpi_restrictors_for_backend] = [qBackendEnum] () {
            typedef struct { bool known; std::string kind; struct {
                             std::string device;
                             std::string sharedOrManaged;
                             std::string host; } restrictors; } retType;
            switch (qBackendEnum) {
            case sycl::backend::ext_oneapi_level_zero:
                return retType{ true, "level_zero",
                                {"device", "shared", "host"} };
                break;
            case sycl::backend::ext_oneapi_opencl:
                return retType{ true, "opencl",
                                {"device", "shared", "host"} };
                break;
            case sycl::backend::ext_oneapi_cuda:
                return retType { true, "cuda",
                                 {"device", "managed", "host"} };
                break;
            case sycl::backend::ext_oneapi_hip:
                return retType { true, "rocm",
                                 {"device", "managed", "host"} };
                break;
            default:
                // means fallback to using "system" memory kind for MPI
                return retType{ false };
                break;
            }
        }();
        std::cout << "SYCL queue backend ('" << qBackendEnum
                  << "'), translated for MPI: "
                  << (queue_uses_backend_defined_by_mpi
                   ? backend_from_sycl_translated_for_mpi
```

```
                  : "NOT DEFINED BY MPI (will tell MPI 'system')")
                 << std::endl;

        MPI_Session session = MPI_SESSION_NULL;
        MPI_Comm comm = MPI_COMM_NULL;
        int my_rank = MPI_PROC_NULL;

        // repeatedly request memory allocation kind:restrictor support
        // in preference order until we find an overlap
        // between what the SYCL backend supports and what MPI provides
        InteractionMethod method;
        for (method = InteractionMethod::begin;
             method < InteractionMethod::end;
             method = static_cast<InteractionMethod>(
                                                 ((size_t)method) + 1)) {

            const auto requested_mem_kind_for_mpi =
            [=]() -> std::optional<std::string> {
                switch (method) {
                case InteractionMethod
                     ::ComputeUsingQueue_CommunicationUsingDeviceMemory:
                    if (!queue_uses_backend_defined_by_mpi)
                        // method cannot work because
                        // MPI does not define this backend
                        return std::nullopt;
                    else if (!qSupportsDeviceMem)
                        // method cannot work
                        // SYCL queue does not support this memory kind
                        return std::nullopt;
                    else
                        return backend_from_sycl_translated_for_mpi +
                               ":" + valid_mpi_restrictors_for_backend
                                                                  .device;
                    break;
                case InteractionMethod
                     ::ComputeUsingQueue_CommunicationUsingSharedMemory:
                    if (!queue_uses_backend_defined_by_mpi)
                        // method cannot work because
                        // MPI does not define this backend
                        return std::nullopt;
                    else if (!qSupportsSharedUSM)
                        // method cannot work
                        // SYCL queue does not support this memory kind
                        return std::nullopt;
                    else
                        return backend_from_sycl_translated_for_mpi +
                               ":" + valid_mpi_restrictors_for_backend
                                                         .sharedOrManaged;
                    break;
                case InteractionMethod
                     ::ComputeUsingQueue_CommunicationUsingHostMemory:
                    if (!queue_uses_backend_defined_by_mpi)
                        // method cannot work because
                        // MPI does not define this backend
```

```
                        return std::nullopt;
                    else if (!qSupportsHostUSM)
                        // method cannot work
                        // SYCL queue does not support this memory kind
                        return std::nullopt;
                    else
                        return backend_from_sycl_translated_for_mpi +
                               ":" + valid_mpi_restrictors_for_backend
                                                                 .host;
                    break;
                case InteractionMethod
                   ::ComputeWithoutQueue_CommunicationUsingSystemMemory:
                    // this method MUST work because the "system" memory
                    // kind must be provided by MPI when requested
                    return "system";
                    break;

                case InteractionMethod::begin:
                case InteractionMethod::end:
                default:
                    return std::nullopt;
                }
            }();
            if (!requested_mem_kind_for_mpi.has_value())
                continue; // this method cannot work, try the next one

            MPI_Info info = MPI_INFO_NULL;
            std::string key_for_mpi("mpi_memory_alloc_kinds");

            // usage mode: REQUESTED
            MPI_Info_create(&info);
            MPI_Info_set(info, key_for_mpi.c_str(),
                         requested_mem_kind_for_mpi.value().c_str());
            MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
            MPI_Info_free(&info);
            std::cout << "Created a session, requested memory kind: "
                      << requested_mem_kind_for_mpi.value()
                      << std::endl;

            // usage mode: PROVIDED
            bool provided = false;
            if (requested_mem_kind_for_mpi.value() == "system") {
                // kind "system" must be provided by MPI when requested
                provided = true; // we have a winner: exit the for loop
            } else {
                MPI_Session_get_info(session, &info);
                int len = 0, flag = 0;
                MPI_Info_get_string(info, key_for_mpi.c_str(), &len,
                                    nullptr, &flag);
                if (flag && len > 0) {
                    size_t num_bytes_needed = (size_t)len*sizeof(char);
                    char* val = static_cast<char*>(
                                          malloc(num_bytes_needed));
                    if (nullptr == val) std::terminate();
```

```
                    MPI_Info_get_string(info, key_for_mpi.c_str(),
                                        &len, val, &flag);
                    std::string val_from_mpi(val);
                    std::cout << "looking for substring: "
                              << requested_mem_kind_for_mpi.value()
                              << std::endl;
                    std::cout << "within value from MPI: "
                              << val_from_mpi << std::endl;
                    if (std::string::npos != val_from_mpi.find(
                                  requested_mem_kind_for_mpi.value())) {
                        provided = true; // we have a winner: assert
                    } else {
                        std::cout << "Not found -- this MPI_Session"
                                   + "does NOT provide the requested"
                                   + "support!" << std::endl;
                    }
                    free(val);
                } else {
                    std::cout << "Info key '" << key_for_mpi << "' "
                               + "not found in MPI_Info from session!"
                              << std::endl;
                }
                MPI_Info_free(&info);
            }
            if (!provided)
                MPI_Session_finalize(&session);
            else {
                // usage mode: ASSERTED
                std::string assert_key_for_mpi(
                                       "mpi_assert_memory_alloc_kinds");
                std::cout << "MPI says it provides the requested memory"
                           + " kind ("
                          << requested_mem_kind_for_mpi.value()
                          << ")--will assert during MPI_Comm creation"
                          << std::endl;
                MPI_Info_create(&info);
                MPI_Info_set(info, assert_key_for_mpi.c_str(),
                            requested_mem_kind_for_mpi.value().c_str());

                MPI_Group world_group = MPI_GROUP_NULL;
                std::string pset_for_mpi("mpi://world");
                MPI_Group_from_session_pset(session,
                                   pset_for_mpi.c_str(), &world_group);
                std::string tag_for_mpi("org.mpi-forum.mpi-side-doc."
                                     + "mem-alloc-kinds.sycl-example");
                MPI_Comm_create_from_group(world_group,
                                           tag_for_mpi.c_str(), info,
                                           MPI_ERRORS_ARE_FATAL, &comm);
                MPI_Group_free(&world_group);
                MPI_Comm_rank(comm, &my_rank);

                break;
            }
        } // end of 'for (InteractionMethod)'
```

```
        if (MPI_SESSION_NULL == session) {
            std::cout << "FAILED to create a usable MPI session"
                      << std::endl; //  (should not happen)
            std::terminate();
        } else
            std::cout << "SUCCESS -- for this session, MPI says the"
                       + " requested memory kind is provided"
                      << std::endl;

        // allocate a data buffer on GPU or CPU
        int* data_buffer = [&q, &method, &my_rank] {
            switch (method) {
            case InteractionMethod
                     ::ComputeUsingQueue_CommunicationUsingDeviceMemory:
                std::cout << "[rank:" << my_rank << "] MPI says this"
                           + " communicator can accept device memory --"
                           + " allocating memory on device"
                          << std::endl;
                return malloc_device<int>(6, q);
                break;
            case InteractionMethod
                     ::ComputeUsingQueue_CommunicationUsingSharedMemory:
                std::cout << "[rank:" << my_rank << "] MPI says this"
                           + " communicator can accept shared/managed"
                           + " memory -- allocating USM shared memory"
                          << std::endl;
                return malloc_shared<int>(6, q);
                break;
            case InteractionMethod
                       ::ComputeUsingQueue_CommunicationUsingHostMemory:
                std::cout << "[rank:" << my_rank << "] MPI says this"
                           + " communicator can accept host memory --"
                           + " allocating USM host memory" << std::endl;
                return malloc_host<int>(6, q);
                break;
            case InteractionMethod
                   ::ComputeWithoutQueue_CommunicationUsingSystemMemory:
                std::cout << "[rank:" << my_rank << "] MPI says this"
                           + " communicator CANNOT accept device memory"
                           + " -- allocating memory on system"
                          << std::endl;
                return static_cast<int*>(malloc(6 * sizeof(int)));
                break;

            case InteractionMethod::begin:
            case InteractionMethod::end:
            default:
                std::cout << "ERROR: invalid interaction method"
                          << std::endl; //  (should not happen)
                std::terminate();
                break;
            }
        }();

```

```
        // define a simple work task for GPU or CPU
        auto do_work = [=]() {
            for (int i = 0; i < 6; ++i)
                data_buffer[i] = (my_rank + 1) * 7;
            };

        // execute the work task using the data buffer on GPU or CPU
        if (method != InteractionMethod
                 ::ComputeWithoutQueue_CommunicationUsingSystemMemory) {
            q.submit([&](sycl::handler& h) {
                h.single_task(do_work);
            }).wait_and_throw();
            std::cout << "[rank:" << my_rank << "] finished work on GPU"
                        << std::endl;
        } else {
                do_work();
            std::cout << "[rank:" << my_rank << "] finished work on CPU"
                        << std::endl;
        }

        MPI_Allreduce(MPI_IN_PLACE, data_buffer, 6, MPI_INT, MPI_MAX,
                                                                 comm);
        std::cout << "[rank:" << my_rank << "] finished reduction"
                    << std::endl;

        MPI_Comm_disconnect(&comm);
        MPI_Session_finalize(&session);

        int answer = std::numeric_limits<int>::max();
        if (method == InteractionMethod
                   ::ComputeUsingQueue_CommunicationUsingDeviceMemory) {
            q.memcpy(&answer, &data_buffer[0], sizeof(int))
                                                   .wait_and_throw();
        } else {
            answer = data_buffer[0];
        }
        std::cout << "[rank:" << my_rank << "] The answer is: "
                    << answer << std::endl;
        if (method != InteractionMethod
                 ::ComputeWithoutQueue_CommunicationUsingSystemMemory) {
            free(data_buffer, q);
        } else {
            free(data_buffer);
        }
    }
    catch (sycl::exception const& e) {
        std::cout << "An exception was caught.\n";
        std::terminate();
    }
    return 0;
}
```

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <mpi.h>
#include <cuda_runtime.h>

#define CUDA_CHECK(mpi_comm, call)                                  \
    {                                                               \
        const cudaError_t error = call;                             \
        if (error != cudaSuccess)                                   \
        {                                                           \
            fprintf(stderr,                                         \
                    "An error occurred: \"%s\" at %s:%d\n",         \
                    cudaGetErrorString(error),                      \
                    __FILE__, __LINE__);                            \
            MPI_Abort(mpi_comm, error);                             \
        }                                                           \
    }

int main(int argc, char *argv[])
{
    int cuda_device_aware = 0;
    int cuda_managed_aware = 0;
    int len = 0, flag = 0;
    int *managed_buf = NULL;
    int *device_buf = NULL, *system_buf = NULL;
    int nranks = 0;
    MPI_Info info;
    MPI_Session session;
    MPI_Group wgroup;
    MPI_Comm system_comm;
    MPI_Comm cuda_managed_comm = MPI_COMM_NULL;
    MPI_Comm cuda_device_comm = MPI_COMM_NULL;

    // Usage mode: REQUESTED
    MPI_Info_create(&info);
    MPI_Info_set(info, "mpi_memory_alloc_kinds",
                       "system,cuda:device,cuda:managed");
    MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
    MPI_Info_free(&info);

    // Usage mode: PROVIDED
    MPI_Session_get_info(session, &info);
    MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
                        &len, NULL, &flag);

    if (flag) {
        char *val, *valptr, *kind;
```

```
        val = valptr = (char *) malloc(len);
        if (NULL == val) return 1;

        MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
                            &len, val, &flag);

        while ((kind = strsep(&val, ",")) != NULL) {
            if (strcasecmp(kind, "cuda:managed") == 0) {
                cuda_managed_aware = 1;
            }
            else if (strcasecmp(kind, "cuda:device") == 0) {
                cuda_device_aware = 1;
            }
        }
        free(valptr);
    }

    MPI_Info_free(&info);

    MPI_Group_from_session_pset(session, "mpi://WORLD" , &wgroup);

    // Create a communicator for operations on system memory
    // Usage mode: ASSERTED
    MPI_Info_create(&info);
    MPI_Info_set(info, "mpi_assert_memory_alloc_kinds", "system");
    MPI_Comm_create_from_group(wgroup,
        "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.system",
        info, MPI_ERRORS_ABORT, &system_comm);
    MPI_Info_free(&info);

    MPI_Comm_size(system_comm, &nranks);

    /** Check for CUDA awareness **/

    // Note: MPI does not require homogeneous support
    // across all processes for memory allocation kinds.
    // This example chooses to use
    // CUDA managed allocations (or device allocations)
    // only when all processes report it is supported.

    // Check if all processes have CUDA managed support
    MPI_Allreduce(MPI_IN_PLACE, &cuda_managed_aware, 1, MPI_INT,
                  MPI_LAND, system_comm);

    if (cuda_managed_aware) {
        // Create a communicator for operations that use
        // CUDA managed buffers.
        // Usage mode: ASSERTED
        MPI_Info_create(&info);
        MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
                          "cuda:managed");
        MPI_Comm_create_from_group(wgroup,
          "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.managed",
```

```
          info, MPI_ERRORS_ABORT, &cuda_managed_comm);
        MPI_Info_free(&info);
    }
    else {
        // Check if all processes have CUDA device support
        MPI_Allreduce(MPI_IN_PLACE, &cuda_device_aware, 1, MPI_INT,
                      MPI_LAND, system_comm);
        if (cuda_device_aware) {
            // Create a communicator for operations that use
            // CUDA device buffers.
            // Usage mode: ASSERTED
            MPI_Info_create(&info);
            MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
                         "cuda:device");
            MPI_Comm_create_from_group(wgroup,
            "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.device",
            info, MPI_ERRORS_ABORT, &cuda_device_comm);
            MPI_Info_free(&info);
        }
        else {
            printf("Warning: cuda alloc kind not supported\n");
        }
    }

    MPI_Group_free(&wgroup);

    /** Execute according to level of CUDA awareness **/
    if (cuda_managed_aware) {
        // Allocate managed buffer and initialize it
        CUDA_CHECK(system_comm,
                   cudaMallocManaged((void**)&managed_buf, sizeof(int),
                   cudaMemAttachGlobal));
        *managed_buf = 1;

        // Perform communication using cuda_managed_comm
        // if it's available.
        MPI_Allreduce(MPI_IN_PLACE, managed_buf, 1, MPI_INT,
                      MPI_SUM, cuda_managed_comm);

        assert((*managed_buf) == nranks);

        CUDA_CHECK(system_comm,
                   cudaFree(managed_buf));
    }
    else {
        // Allocate system buffer and initialize it
        // (using cudaMallocHost for better performance of cudaMemcpy)
        CUDA_CHECK(system_comm,
                   cudaMallocHost((void**)&system_buf, sizeof(int)));
        *system_buf = 1;

        // Allocate CUDA device buffer and initialize it
        CUDA_CHECK(system_comm,
                   cudaMalloc((void**)&device_buf, sizeof(int)));
```

```
        CUDA_CHECK(system_comm,
                   cudaMemcpyAsync(device_buf, system_buf, sizeof(int),
                   cudaMemcpyHostToDevice, 0));

        CUDA_CHECK(system_comm,
                   cudaStreamSynchronize(0));
        if (cuda_device_aware) {
            // Perform communication using cuda_device_comm
            // if it's available.
            MPI_Allreduce(MPI_IN_PLACE, device_buf, 1, MPI_INT,
                          MPI_SUM, cuda_device_comm);

            assert((*device_buf) == nranks);
        }
        else {
            // Otherwise, copy data to a system buffer,
            // use system_comm, and copy data back to device buffer
            CUDA_CHECK(system_comm,
                       cudaMemcpyAsync(system_buf, device_buf,
                       sizeof(int), cudaMemcpyDeviceToHost, 0));

            CUDA_CHECK(system_comm,
                       cudaStreamSynchronize(0));
            MPI_Allreduce(MPI_IN_PLACE, system_buf, 1, MPI_INT,
                          MPI_SUM, system_comm);
            CUDA_CHECK(system_comm,
                       cudaMemcpyAsync(device_buf, system_buf,
                       sizeof(int), cudaMemcpyHostToDevice, 0));

            CUDA_CHECK(system_comm,
                       cudaStreamSynchronize(0));
            assert((*system_buf) == nranks);
        }

        CUDA_CHECK(system_comm, cudaFree(device_buf));
        CUDA_CHECK(system_comm, cudaFreeHost(system_buf));
    }

    if (cuda_managed_comm != MPI_COMM_NULL)
        MPI_Comm_disconnect(&cuda_managed_comm);
    if (cuda_device_comm != MPI_COMM_NULL)
        MPI_Comm_disconnect(&cuda_device_comm);
    MPI_Comm_disconnect(&system_comm);

    MPI_Session_finalize(&session);

    return 0;
}
```

3.3 MPI plus ROCm

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <mpi.h>
#include <hip/hip_runtime_api.h>

#define HIP_CHECK(condition) {                              \
        hipError_t error = condition;                       \
        if(error != hipSuccess){                            \
            fprintf(stderr,"HIP error: %d line: %d\n", \
                    error, __LINE__);                       \
            MPI_Abort(MPI_COMM_WORLD, error);               \
        }                                                   \
    }

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    int rocm_device_aware = 0;
    int len = 0, flag = 0;
    int *device_buf = NULL;
    MPI_File file;
    MPI_Status status;
    MPI_Info info;

    // Usage mode: REQUESTED
    // Supply mpi_memory_alloc_kinds to the MPI startup
    // mechanism (not shown)
    MPI_Init(&argc, &argv);

    // Usage mode: PROVIDED
    // Query the MPI_INFO object on MPI_COMM_WORLD to
    // determine whether the MPI library provides
    // support for the memory allocation kinds
    // requested via the MPI startup mechanism
    MPI_Comm_get_info(MPI_COMM_WORLD, &info);
    MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
                        &len, NULL, &flag);
    if (flag) {
        char *val, *valptr, *kind;

        val = valptr = (char *) malloc(len);
        if (NULL == val) return 1;

        MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
                            &len, val, &flag);

```

```
        while ((kind = strsep(&val, ",")) != NULL) {
            if (strcasecmp(kind, "rocm:device") == 0) {
                rocm_device_aware = 1;
            }
        }
        free(valptr);
    }

    HIP_CHECK(hipMalloc((void**)&device_buf, BUFSIZE * sizeof(int)));

    // The user could optionally create an info object,
    // set mpi_assert_memory_alloc_kind to the memory type
    // it plans to use, and pass this as an argument to
    // MPI_File_open. This approach has the potential to
    // enable further optimizations in the MPI library.
    MPI_File_open(MPI_COMM_WORLD, "inputfile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL, &file);

    if (rocm_device_aware) {
        MPI_File_read(file, device_buf, BUFSIZE, MPI_INT, &status);
        printf("Using if part\n");
    }
    else {
        int *tmp_buf;
        printf("Using else part\n");
        tmp_buf = (int*) malloc (BUFSIZE * sizeof(int));
        MPI_File_read(file, tmp_buf, BUFSIZE, MPI_INT, &status);

        HIP_CHECK(hipMemcpyAsync(device_buf, tmp_buf,
                                 BUFSIZE * sizeof(int),
                                 hipMemcpyDefault, 0));
        HIP_CHECK(hipStreamSynchronize(0));

        free(tmp_buf);
    }

    // Launch compute kernel(s)

    MPI_File_close(&file);
    HIP_CHECK(hipFree(device_buf));

    MPI_Finalize();
    return 0;
}
```

Bibliography

[1] CUDA Runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/.

[2] HIP Programming Manual. https://rocm.docs.amd.com/en/latest/reference/hip.html.

[3] Level Zero Programming Guide. https://spec.oneapi.io/level-zero/latest/core/PROG.html.