Partition Communication and SYCL Bindings

Maria Garzaran

intel®

A detour to SYCL

oneAPI for Cross-Architecture Performance

Optimized Applications

Optimized Middleware & Frameworks

oneAPI Product

Direct Programming

SYCL/DPC++

API-Based Programming

Libraries

Analysis & Debug Tools

CPU

GPU

FPGA

AI

Get functional quickly, then analyze and tune

SYCL

- SYCL is an industry-driven Khronos standard that **adds data parallelism to C++ for heterogeneous systems** (https://www.khronos.org/sycl/)
- There are SYCL compilers that support different vendor's hardware:

1. Intel compiler generates code for Intel CPUs, GPUs, and FPGAs
2. ComputeCpp from CodePlay generates code for any CPU, Intel GPUs and FPGAs, AMD GPUs, ARM Mali, Nvidia GPUs
3. triSYCL from Xilinx generates code Xilinx FPGAs
4. hipSYCL from University of Heidelberg generates code for AMD GPUs and Nvidia GPUs.

- Libraries such as Kokkos and Raja have SYCL backends

Hello data-parallel SYCL code

```cpp
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;

const std::string secret {
  "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
  "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01" };
const auto sz = secret.size();

int main() {
  queue Q;

  char *result = malloc_shared<char>(sz, Q);
  std::memcpy(result,secret.data(),sz);

  Q.parallel_for(sz,[=](auto& i) {
     result[i] -= 1;
     }).wait();

  std::cout << result << "\n";
  return 0;
}
```

- Work is submitted to a queue
- A queue is associated with a single device
- Work in the device is asynchronous

- Allocate memory on shared memory
- (both host and device can access)

- Submit work to the queue
- The work is expressed as a lambda function here

- Piece of work the kernel performs

Hello data-parallel SYCL code

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;

const std::string secret {
  "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
  "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01" };
const auto sz = secret.size();

int main() {
  queue Q;

  char *result = malloc_shared<char>(sz, Q);
  std::memcpy(result,secret.data(),sz);

  Q.parallel_for(sz,[=](auto& i) {
      result[i] -= 1;
      }).wait();

  std::cout << result << "\n";
  return 0;
}
```

Host code

Device code

Host code

Where the code executes depends on where we setup the queue

queue Q { default_selector{} };
queue Q { host_selector{} };
queue Q { cpu_selector{} };
queue Q { gpu_selector{} };
queue Q { accelerator_selector{} };
queue Q { INTEL::fpga_selector{} };

Partition Communication

Example 4.2 MPI standard (OpenMP)

```
if (myrank == 0) {
  MPI_Psend_init(message, partitions, count, xfer_type,
     dest, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);

  #pragma omp parallel for shared(request)
num_threads(NUM_THREADS)
  for (int i = 0; i < partitions; i++) {
      /* compute and fill partition #i, then mark ready: */
      MPI_Pready(i, request);
  }
  while (!flag) {
     /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
}
```

```
if (myrank == 1) {
   MPI_Precv_init(message, partitions, count, xfer_type
     source, tag, info, MPI_COMM_WORLD, &request);
   MPI_Start(&request);

 while (!flag) {
  /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
 MPI_Request_free(&request);
}
```

Partition Communication Ex. 4.2 MPI Standard (SYCL)

```
if (myrank == 0) {
  MPI_Psend_init(message, partitions, count, xfer_type,
    dest, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);

  mQueue.submit([&](sycl::handler &h) {
    h.parallel_for(partitions, [=](id<1> i) {
      MPI_Pready(i, request);
    });
  });
  mQueue.wait();
  while (!flag) {
    /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
}
```

```
if (myrank == 1) {
  MPI_Precv_init(message, partitions, count, xfer_type
    source, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);

  while (!flag) {
   /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
}
```

Partition Communication: MPI_Pready executes from Device

```
if (myrank == 0) {
  MPI_Psend_init(message, partitions, count, xfer_type,
    dest, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);
  MPI_Pbuf_prepare(&request);

  mQueue.submit([&](sycl::handler &h) {
    h.parallel_for(partitions, [=](id<1> i) {
      MPI_Pready(i, request);
    });
  });
  mQueue.wait();
  while (!flag) {
    /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
}
```

```
if (myrank == 1) {
  MPI_Precv_init(message, partitions, count, xfer_type
    source, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);
  MPI_Pbuf_prepare(&request);

  while (!flag) {
  /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
}
```

Partition Communication: MPI_Prequest for device

```
if (myrank == 0) {
  MPI_Psend_init(message, partitions, count, xfer_type,
     dest, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);
  MPI_Pbuf_prepare(&request);
  // Create a request to pass to the kernel
  MPI_Prequest_create(request, info, &device_request);

  mQueue.submit([&](sycl::handler &h) {
    h.parallel_for(partitions, [=](id<1> i) {
      MPI_Pready(i, device_request);
    });
  });
 mQueue.wait();
 while (!flag) {
    /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
 }
 MPI_Request_free(&request);
}
```

```
if (myrank == 1) {
   MPI_Precv_init(message, partitions, count, xfer_type
     source, tag, info, MPI_COMM_WORLD, &request);
   MPI_Start(&request);
   MPI_Pbuf_prepare(&request);

  while (!flag) {
   /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
   }
  MPI_Request_free(&request);
}
```

Partition Communication: MPI_Prequest for device (notes)

```
if (myrank == 0) {
  MPI_Psend_init(message, partitions, count, xfer_type,
     dest, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);
  MPI_Pbuf_prepare(&request);
  // Create a request to pass to the kernel
  MPI_Prequest_create(request, info, &device_request);

  mQueue.submit([&](sycl::handler &h) {
     h.parallel_for(partitions, [=](id<1> i) {
        MPI_Pready(i, device_request);
     });
  });
 mQueue.wait();
 while (!flag) {
     /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
 }
 MPI_Request_free(&request);
}
```

```
if (myrank == 1) {
   MPI_Precv_init(message, partitions, count, xfer_type
     source, tag, info, MPI_COMM_WORLD, &request);
   MPI_Start(&request);
   MPI_Pbuf_prepare(&request);
   }
  MPI_Request_free(&request);
}
```

- We are making the implicit assumption that the device request is allocated on the same device as the message
- We also assume that the message is only allocated in a single device
- Device information is not needed, because we assume the runtime is able to recover the device where message is allocated from the request created during the MPI_Psend_init() call
- An additional reason to have the device_request is that in some devices (e.g. FPGAs), the kernel might not be able to access the host memory, so it would not be able to access the request in the host memory

MPI_Prequest

```
int MPI_Prequest_create (MPI_Request req,
                         MPI_Info info,
                         MPI_Prequest *preq)
int MPI_Prequest_free(MPI_Prequest *preq)
```

Executes in the host

- MPI_Prequest should be created on the same device as the message in MPI_Send_Init().
- The MPI implementation is free to allocate the MPI_Prequest on USM shared, device or host.
- Each MPI implementation can determine the information that is needed on the MPI_Prequest
- MPI_Prequest objects are only valid for use in device functions

Device Ready

SYCL_EXTERNAL int MPI_Pready(MPI_Prequest prequest, int partition)

SYCL_EXTERNAL int MPI_Pready_range(int part_low, int part_high, MPI_Prequest prequest)

SYCL_EXTERNAL int MPI_Pready_list(int length, const int array_of_partitions[], MPI_Prequest prequest)

Execute in the device

Device Arrived

SYCL_EXTERNAL int MPI_Parrived(MPI_Prequest preq, int partition, int *flag)

Execute in the device

Others

- For efficiency reasons we should define variants of MPI_Ready()/MPI_Parrived() for work items (warp) and/or work groups (thread block)

Example 4.2 MPI standard (OpenMP offload)

```
if (myrank == 0) {
  MPI_Psend_init(message, partitions, count, xfer_type,
     dest, tag, info, MPI_COMM_WORLD, &request);
  MPI_Start(&request);

  #pragma omp target teams distribute parallel for
  for (int i = 0; i < partitions; i++) {
     /* compute and fill partition #i, then mark ready: */
      MPI_Pready(i, request);
  }
  while (!flag) {
     /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
}
```

```
if (myrank == 1) {
   MPI_Precv_init(message, partitions, count, xfer_type
     source, tag, info, MPI_COMM_WORLD, &request);
   MPI_Start(&request);

  while (!flag) {
   /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
   }
  MPI_Request_free(&request);
}
```

OpenMP declare target directive

declare target directive is used to indicate that the corresponding call inside a **target** region is to a function that can execute on the default target device

```
#pragma omp declare target
int MPI_Pready(MPI_Request request, int partition) {
    ...
}
#pragma omp end declare target
```