MULTI-GPU PROGRAMMING WITH MPI

Jiri Kraus, Senior Devtech Compute, NVIDIA
MPI+CUDA

Node 0

MEM

GPU

IB

Node 1

MEM

GPU

IB

Node N-1

MEM

GPU

IB
MPI+CUDA
MPI+CUDA

//MPI rank 0
MPI_Send(s_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d, size, MPI_CHAR, n-1, tag, MPI_COMM_WORLD, &stat);
YOU WILL LEARN

What MPI is
How to use MPI for inter GPU communication with CUDA and OpenACC
What CUDA-aware MPI is
What Multi Process Service is and how to use it
How to use NVIDIA tools in an MPI environment
How to hide MPI communication times
MESSAGE PASSING INTERFACE - MPI

Standard to exchange data between processes via messages

Defines API to exchanges messages

Point to Point: e.g. MPI_Send, MPI_Recv

Collectives: e.g. MPI_Reduce

Multiple implementations (open source and commercial)

Bindings for C/C++, Fortran, Python, ...

E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, ...
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    /* Initialize the MPI library */
    MPI_Init(&argc, &argv);
    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
MPI
Compiling and Launching

$ mpicc -o myapp myapp.c
$ mpirun -np 4 ./myapp <args>
A SIMPLE EXAMPLE
EXAMPLE: JACOBI SOLVER

Solves the 2D-Laplace Equation on a rectangle

$$\Delta u(x, y) = 0 \ \forall \ (x, y) \in \Omega \setminus \delta \Omega$$

Dirichlet boundary conditions (constant values on boundaries)

$$u(x, y) = f(x, y) \in \delta \Omega$$

2D domain decomposition with n x k domains
EXAMPLE: JACOBI SOLVER

Single GPU

While not converged

Do Jacobi step:

```c
for (int iy=1; iy < ny-1; ++iy)
  for (int ix=1; ix < nx-1; ++ix)
    u_new[ix][iy] = 0.0f - 0.25f*( u[ix-1][iy] + u[ix+1][iy]
                                 + u[ix][iy-1] + u[ix][iy+1]);
```

Swap u_new and u

Next iteration
EXAMPLE: JACOBI SOLVER

Multi GPU

While not converged

Do Jacobi step:

```c
for (int iy=1; iy < ny-1; ++iy)
    for (int ix=1; ix < nx-1; ++ix)
        u_new[ix][iy] = 0.0f - 0.25f*( u[ix-1][iy] + u[ix+1][iy] + u[ix][iy-1] + u[ix][iy+1] );
```

Exchange halo with 2 4 neighbors

Swap $u_{\text{new}}$ and $u$

Next iteration
MPI_Sendrecv(u_new+offset_first_row m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```c
#pragma acc host_data use_device ( u_new ) {
    MPI_Sendrecv( u_new + offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
                  u_new + offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv( u_new + offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
                  u_new + offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 1,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

MPI_Sendrecv( u_new_d + offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
              u_new_d + offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv( u_new_d + offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
              u_new_d + offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 1,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

**EXAMPLE JACOBI**

**Top/Bottom Halo**

---

**CUDA**

---

**OpenACC**

---
EXAMPLE: JACOBI

Left/Right Halo

```
//right neighbor omitted
#pragma acc parallel loop present ( u_new, to_left )
for ( int i=0; i<n-2; ++i )
    to_left[i] = u_new[(i+1)*m+1];

#pragma acc host_data use_device ( from_right, to_left ) {
    MPI_Sendrecv( to_left, n-2, MPI_DOUBLE, l_nb, 0,
                  from_right, n-2, MPI_DOUBLE, r_nb, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}

#pragma acc parallel loop present ( u_new, from_right )
for ( int i=0; i<n-2; ++i )
    u_new[(m-1)+(i+1)*m] = from_right[i];
```
EXAMPLE: JACOBI

Left/Right Halo

//right neighbor omitted
pack<<<gs,bs,0,s>>>(to_left_d, u_new_d, n, m);
cudaStreamSynchronize(s);

MPI_Sendrecv( to_left_d, n-2, MPI_DOUBLE, l_nb, 0,
from_right_d, n-2, MPI_DOUBLE, r_nb, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE );

unpack<<<gs,bs,0,s>>>(u_new_d, from_right_d, n, m);
LAUNCH MPI+CUDA/OPENACC PROGRAMS

Launch one process per GPU

**MVAPICH:** `MV2_USE_CUDA`

```
$ MV2_USE_CUDA=1 mpirun -np ${np} ./myapp <args>
```

**Open MPI:** CUDA-aware features are enabled per default

**Cray:** `MPICH_RDMA_ENABLED_CUDA`

**IBM Platform MPI:** `PMPY_GPU_AWARE`
HANDLING MULTIPLE MULTI GPU NODES
HANDLING MULTIPLE MULTI GPU NODES
How to determine the local rank? - MPI-3

MPI_Comm local_comm;
MPI_Info info;
MPI_Info_create(&info);
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank, info, &local_comm);
int local_rank = -1;
MPI_Comm_rank(local_comm,&local_rank);
MPI_Comm_free(&local_comm);
MPI_Info_free(&info);
HANDLING MULTIPLE MULTI GPU NODES
HANDLING MULTIPLE MULTI GPU NODES
GPU-affinity

Use local rank:

```c
int local_rank = -1;
MPI_Comm_rank(local_comm,&local_rank);
int num_devices = 0;
cudaGetDeviceCount(&num_devices);
cudaSetDevice(local_rank % num_devices);
```
JACOBI RESULTS (1000 STEPS)
MVAPICH2-2.0b FDR IB - Weak Scaling 4k x 4k per Process

![Bar chart showing runtime for different numbers of MPI ranks. The bars are labeled with 'Tesla K20X' and 'Xeon E5-2690 v2 @ 3.0Ghz'. The x-axis represents the number of MPI ranks, ranging from 1 to 8. The y-axis represents runtime in seconds, ranging from 0 to 14 seconds.]
EXAMPLE JACOBI
Top/Bottom Halo

```c
#pragma acc update host(u_new[offset_first_row:m-2], u_new[offset_last_row:m-2])
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
#pragma acc update device(u_new[offset_top_boundary:m-2], u_new[offset_bottom_boundary:m-2])
```

```
//send to bottom and receive from top top bottom bottom omitted

cudaMemcpy(u_new+offset_first_row,
            u_new_d+offset_first_row, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
cudaMemcpy(u_new_d+offset_bottom_boundary,
            u_new+offset_bottom_boundary, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
```
THE DETAILS
UNIFIED VIRTUAL ADDRESSING

No UVA: Separate Address Spaces

UVA: Single Address Space

System Memory

GPU Memory

PCI-e

0x0000
0xFFFF

0x0000
0xFFFF

CPU

GPU

System Memory

GPU Memory

PCI-e

0x0000

0xFFFF

CPU

GPU

nVIDIA
UNIFIED VIRTUAL ADDRESSING

One address space for all CPU and GPU memory

Determine physical memory location from a pointer value

Enable libraries to simplify their interfaces (e.g. MPI and cudaMemcpy)

Supported on devices with compute capability 2.0+ for

64-bit applications on Linux and Windows (+TCC)
NVIDIA GPUDIRECT™

Peer to Peer Transfers
NVIDIA GPUDIRECT™
Support for RDMA
CUDA-AWARE MPI

Example:

MPI Rank 0 \texttt{MPI\_Send} from GPU Buffer

MPI Rank 1 \texttt{MPI\_Recv} to GPU Buffer

Show how CUDA+MPI works in principle

Depending on the MPI implementation, message size, system setup, ... situation might be different

Two GPUs in two nodes
MPI_GPU_TO_REMOTE_GPU

Support for RDMA

MPI Rank 0

GPU

Host

MPI Rank 1

MPI_Send(s_buf_d, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
MPI GPU TO REMOTE GPU
Support for RDMA

Time

MPI_Sendrecv

32
**REGULAR MPI GPU TO REMOTE GPU**

```c
cudaMemcpy(s_buf_h, s_buf_d, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
MPI_Recv(r_buf_h, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
cudaMemcpy(r_buf_d, r_buf_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(s_buf_h, s_buf_d, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
MPI_Recv(r_buf_h, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
cudaMemcpy(r_buf_d, r_buf_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(s_buf_h, s_buf_d, size, cudaMemcpyDeviceToHost);
```
REGULAR MPI GPU TO REMOTE GPU

memcpy D->H  

MPI_Sendrecv

memcpy H->D

Time
MPI_GPU_TOREMOTE_GPU
without GPUDirect

MPI_Send(s_buf_h, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
MPI_Recv(r_buf_h, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
MPI GPU TO REMOTE GPU

without GPUDirect
PERFORMANCE RESULTS GPUDIRECT RDMA
OpenMPI 1.10.4 MLNX FDR IB (4X) Tesla P100 PCI-E 16GB

Latency (1 Byte)  15.65 us  16.50 us  4.87 us
PERFORMANCE RESULTS GPUDIRECT P2P
OpenMPI 1.10.4 DGX-1 Tesla P100

![Graph showing performance results for CUDA-aware MPI with GPUDirect P2P, CUDA-aware MPI, and regular MPI.](image)
MULTI PROCESS SERVICE (MPS) FOR MPI APPLICATIONS
GPU ACCELERATION OF LEGACY MPI APPS

Typical legacy application

- MPI parallel
- Single or few threads per MPI rank (e.g. OpenMP)

Running with multiple MPI ranks per node

GPU acceleration in phases

- Proof of concept prototype, ...
- Great speedup at kernel level

Application performance misses expectations
MULTI PROCESS SERVICE (MPS)
For Legacy MPI Applications

Multicore CPU only

GPU-accelerated

N=1  N=2  N=4  N=8

N=1  N=2  N=4  N=8

GPU parallelizable part
CPU parallel part
Serial part

With Hyper-Q/MPS
Available on Tesla/Quadro with CC 3.5+
(e.g. K20, K40, K80, M40,...)
PROCESSES SHARING GPU WITHOUT MPS

No Overlap

Process A
Context A

Process B
Context B

GPU
PROCESSES SHARING GPU WITHOUT MPS

Context Switch Overhead

Context Switch

[Image of NVIDIA profiling tool with highlighted context switch]
PROCESSES SHARING GPU WITH MPS

Maximum Overlap

- Process A
  - Context A
- Process B
  - Context B

MPS Process

GPU

Kernels from Process A

Kernels from Process B
PROCESSES SHARING GPU WITH MPS

No Context Switch Overhead
HYPER-Q/MPS CASE STUDY: UMT

Enables overlap between copy and compute of different processes.

GPU sharing between MPI ranks increases utilization.
HYPER-Q/MPS CASE STUDIES

CPU Scaling Speedup

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup vs. 1 Rank/GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>HACC</td>
<td>3</td>
</tr>
<tr>
<td>MP2C</td>
<td>4.7</td>
</tr>
<tr>
<td>VASP</td>
<td>1.5</td>
</tr>
<tr>
<td>ENZO</td>
<td>2.1</td>
</tr>
<tr>
<td>UMT</td>
<td>1.3</td>
</tr>
</tbody>
</table>

CPU Scaling Speedup
HYPER-Q/MPS CASE STUDIES

Additional Speedup with MPS

![Speedup vs. 1 Rank/GPU](image)

- **HACC**: 
  - CPU Scaling Speedup: 4
  - Overlap/MPS Speedup: 1
- **MP2C**: 
  - CPU Scaling Speedup: 5
  - Overlap/MPS Speedup: 1
- **VASP**: 
  - CPU Scaling Speedup: 2
  - Overlap/MPS Speedup: 1
- **ENZO**: 
  - CPU Scaling Speedup: 3
  - Overlap/MPS Speedup: 1
- **UMT**: 
  - CPU Scaling Speedup: 2
  - Overlap/MPS Speedup: 1
No application modifications necessary
Not limited to MPI applications

MPS control daemon

Spawn MPS server upon CUDA application startup

#Typical Setup

nvidia-smi -c EXCLUSIVE_PROCESS

nvidia-cuda-mps-control -d

#On Cray XK/XC systems

export CRAY_CUDA_MPS=1
MPS SUMMARY

Easy path to get GPU acceleration for legacy applications

Enables overlapping of memory copies and compute between different MPI ranks

Remark: MPS adds some overhead!
DEBUGGING AND PROFILING
TOOLS FOR MPI+CUDA APPLICATIONS

Memory checking: cuda-memcheck

Debugging: cuda-gdb

Profiling: nvprof and the NVIDIA Visual Profiler (nvvp)
**MEMORY CHECKING WITH CUDA-MEMCHECK**

`cuda-memcheck` is a tool similar to Valgrind’s memcheck

Can be used in a MPI environment

```
mpiexec -np 2 cuda-memcheck ./myapp <args>
```

**Problem:** Output of different processes is interleaved

**Solution:** Use save or log-file command line options

```
mpirun -np 2 cuda-memcheck
   --log-file name.%q{OMPI_COMM_WORLD_RANK}.log
   --save name.%q{OMPI_COMM_WORLD_RANK}.memcheck
./myapp <args>
```

OpenMPI: `OMPI_COMM_WORLD_RANK`

MVAPICH2: `MV2_COMM_WORLD_RANK`
MEMORY CHECKING WITH CUDA-MEMCHECK
MEMORY CHECKING WITH CUDA-MEMCHECK

Read Output Files with `cuda-memcheck --read`
DEBUGGING MPI+CUDA APPLICATIONS

Using \texttt{cuda-gdb} with MPI Applications

Use \texttt{cuda-gdb} just like \texttt{gdb}

For smaller applications, just launch \texttt{xterms} and \texttt{cuda-gdb}

\texttt{mpiexec -x -np 2 xterm -e cuda-gdb ./myapp <args>
DEBUGGING MPI+CUDA APPLICATIONS

cuda-gdb Attach

```c
if ( rank == 0 ) {
    int i=0;
    printf("rank %d: pid %d on %s ready for attach\n.", rank, getpid(),name);
    while (0 == i) { sleep(5); }
}
```

> mpiexec -np 2 ./jacobi_mpi+cuda
Jacobi relaxation Calculation: 4096 x 4096 mesh with 2 processes and one Tesla M2070 for each process (2049 rows per process).
rank 0: pid 30034 on judge107 ready for attach
> ssh judge107
j kraus@judge107:~> cuda-gdb --pid 30034
DEBUGGING MPI+CUDA APPLICATIONS

CUDA_DEVICE_WAITS_ON_EXCEPTION

The application encountered a device error and CUDA_DEBUG_NOT_SUPPORTED can now attach a debugger to the application. The application encountered a device error and CUDA_DEBUG_DATABASE can now attach a debugger to the application. The application encountered a device error and CUDA_DEBUG_DATABASE can now attach a debugger to the application. The application encountered a device error and CUDA_DEBUG_DATABASE can now attach a debugger to the application. The application encountered a device error and CUDA_DEBUG_DATABASE can now attach a debugger to the application. The application encountered a device error and CUDA_DEBUG_DATABASE can now attach a debugger to the application.
DEBUGGING MPI+CUDA APPLICATIONS

With CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 core dumps are generated in case of an exception:

Can be used for offline debugging

Helpful if live debugging is not possible

CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION: Enable/Disable CPU part of core dump (enabled by default)

CUDA_COREDUMP_FILE: Specify name of core dump file

Open GPU: (cuda-gdb) target cudacore core.cuda

Open CPU+GPU: (cuda-gdb) target core core.cpu core.cuda
DEBUGGING MPI+CUDA APPLICATIONS

CUDA_ENABLE_COREDUMP_ON_EXCEPTION
DEBUGGING MPI+CUDA APPLICATIONS

CUDA_ENABLE_COREDUMP_ON_EXCEPTION
DEBUGGING MPI+CUDA APPLICATIONS

Third Party Tools

Allinea DDT debugger

Rogue Wave TotalView
PROFILING MPI+CUDA APPLICATIONS

Using nvprof+NVVP

Embed MPI rank in output filename, process name, and context name

```
mpirun -np $np nvprof --output-profile profile.%q{OMPI_COMM_WORLD_RANK} \
  --process-name "rank %q{OMPI_COMM_WORLD_RANK}" \
  --context-name "rank %q{OMPI_COMM_WORLD_RANK}"
```

Alternatives:

- Only save the textual output (`--log-file`)
- Collect data from all processes that run on a node (`--profile-all-processes`)

OpenMPI: OMPI_COMM_WORLD_RANK

MVAPICH2: MV2_COMM_WORLD_RANK
PROFILING MPI+CUDA APPLICATIONS
Using \texttt{nvprof}+\texttt{NVVP}
PROFILING MPI+CUDA APPLICATIONS
Using \texttt{nvprof+NVVP}

\texttt{nvvp jacobi.*.nvprof}

Or use the import Wizard
Multiple parallel profiling tools are CUDA-aware

Score-P
Vampir
Tau

These tools are good for discovering MPI issues as well as basic CUDA performance inhibitors.
ADVANCED MPI ON GPUS
BEST PRACTICE: USE NON-BLOCKING MPI

```c
#pragma acc host_data use_device ( u_new ) {
  MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
               u_new+offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
               u_new+offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

MPI_Request t_b_req[4];
#pragma acc host_data use_device ( u_new ) {
  MPI_Irecv(u_new+offset_top_boundary,m-2,MPI_DOUBLE, t_nb, 0,
            t_b_req+1);
  MPI_Irecv(u_new+offset_bottom_boundary,m-2,MPI_DOUBLE, b_nb, 1,
            t_b_req+1);
  MPI_Isend(u_new+offset_last_row,m-2,MPI_DOUBLE, b_nb, 0,
            t_b_req+2);
  MPI_Isend(u_new+offset_first_row,m-2,MPI_DOUBLE, t_nb, 1,
            t_b_req+3);
}
MPI_Waitall(4, t_b_req, MPI_STATUSES_IGNORE);
```

**Non-blocking**

- MPI
- Acc

Gives MPI more opportunities to build efficient pipelines
COMMUNICATION + COMPUTATION OVERLAP
MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

Runtime (s)

<table>
<thead>
<tr>
<th>Local problem size</th>
<th>Nooverlap</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096x4096</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1024x1024</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>
COMMUNICATION + COMPUTATION OVERLAP

No Overlap

Process Whole Domain

Overlap

Boundary and inner domain processing can overlap

Process inner domain

Process boundary domain

Dependency

MPI

Possible gain
COMMUNICATION + COMPUTATION OVERLAP
CUDA with Streams

process_boundary_and_pack<<<gs_b,bs_b,0,s1>>>(u_new_d,u_d,to_left_d,to_right_d,n,m);

process_inner_domain<<<gs_id,bs_id,0,s2>>>(u_new_d,u_d,to_left_d,to_right_d,n,m);

cudaStreamSynchronize(s1);  //wait for boundary
MPI_Request req[8];

//Exchange halo with left, right, top and bottom neighbor
MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
unpack<<<gs_s,bs_s,0,s2>>>(u_new_d, from_left_d, from_right_d, n, m);

cudaDeviceSynchronize();  //wait for iteration to finish
#pragma acc parallel loop present ( u_new, u, to_left, to_right ) async(1)
for ( ... )
   //Process boundary and pack to_left and to_right
#pragma acc parallel loop present ( u_new, u ) async(2)
for ( ... )
   //Process inner domain
#pragma acc wait(1) //wait for boundary
MPI_Request req[8];
#pragma acc host_data use_device ( from_left, to_left, form_right, to_right, u_new )
   //Exchange halo with left, right, top and bottom neighbor
}
MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
#pragma acc parallel loop present ( u_new, from_left, from_right ) async(2)
for ( ... )
   //unpack from_left and from_right
#pragma acc wait //wait for iteration to finish

COMMUNICATION + COMPUTATION OVERLAP
OpenACC with Async Queues
COMMUNICATION + COMPUTATION OVERLAP
MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

Speedup (Overlap vs. Nooverlap)

Runtime (s)

Local problem size

4096x4096
2048x2048
1024x1024
HIGH PRIORITY STREAMS

Improve scalability with high priority streams

cudaStreamCreateWithPriority

Use-case: MD Simulations

Stream 1
- Comp. Local Forces

Stream 2
- Ex. Non-local Atom pos.
- Comp. Non-Local Forces
- Ex. Non-local forces

Stream 1 (LP)
- Comp. Local Forces

Stream 2 (HP)
- Ex. Non-local Atom pos.
- Comp. Non-Local Forces
- Ex. Non-local forces

Possible gain
MPI AND UNIFIED MEMORY

CAVEAT

Using Unified Memory with a non Unified Memory-aware MPI might fail with errors or even worse silently produce wrong results, e.g. when registering Unified Memory for RDMA.

Use a Unified Memory-aware MPI,
e.g. OpenMPI since 1.8.5 or MVAPICH2-GDR since 2.2b

Unified Memory-aware: CUDA-aware MPI with support for Unified Memory
MPI AND UNIFIED MEMORY

Performance Implications

Unified Memory can be used by any processor in the system

Memory pages of a Unified Memory allocation may migrate between processors memories to ensure coherence and maximize performance

Different data paths are optimal for performance depending on where the data is: e.g. NVLink between peer GPUs

The MPI implementation needs to know where the data is, but it can‘t!
MPI AND UNIFIED MEMORY

Performance Implications - Simple Example

cudaMallocManaged( &array, n*sizeof(double), cudaMemAttachGlobal );

while( ... ) {
    foo(array,n);
    MPI_Send(array,...);
    foo(array,n);
    }
MPI AND UNIFIED MEMORY
Performance Implications - Simple Example

- If foo is a CPU function pages of array might migrate to System Memory
- If foo is a GPU function pages of array might migrate to GPU Memory
- The MPI implementation is not aware of the application and thus doesn't know where array is and what's optimal

```c
while( ... ) {
    foo(array,n);
    MPI_Send(array,...);
    foo(array,n);
}
```
MPI AND UNIFIED MEMORY

The Future with Data Usage Hints

Tell where the application intends to use the data

cudaMallocManaged( &array, n*sizeof(double), cudaMemAttachGlobal );

cudaMemAdvise(array,n*sizeof(double),cudaMemAdviseSetPreferredLocation,device);

while(...){
    foo(array,n);
    MPI_Send(array,...);
    foo(array,n);
}

Remark: Data Usage Hints are available since CUDA 8, but currently not evaluated by any Unified Memory-aware MPI implementation.
Tell where the application intends to use the data

```c
cudaMallocManaged( &array, n*sizeof(double), cudaMemcpyTimeout );
cudaMemAdvise(array,n*sizeof(double),cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);
while( ... ) {
    foo(array,n);
    MPI_Send(array,...);
    foo(array,n);
}
```

Remark: Data Usage Hints are available since CUDA 8, but currently not evaluated by any Unified Memory-aware MPI implementation.

Array is intended to be used on the CPU
Data usage hints can be queried by the MPI Implementation and allow it to take the optimal data path.

If the application lies about the data usage hints it will run correctly but performance will be affected.

Performance tools help to identify missing or wrong data usage hints.

Data usage hints are general useful for the Unified Memory system and can improve application performance.

Remark: Data Usage Hints are only hints to guide the data usage policies of the Unified Memory system. The Unified Memory system might ignore them, e.g. to ensure coherence or in oversubscription scenarios.
MPI AND UNIFIED MEMORY

Current Status

Available Unified Memory-aware MPI implementations

- OpenMPI (since 1.8.5)
- MVAPICH2-GDR (since 2.2b)
  - Performance improvements with 2.2RC1 for Intranode GPU to GPU communication

Currently both don’t evaluate Data Usage Hints, i.e. all Unified Memory is treated as Device Memory

Good performance if all buffers used in MPI are touched mainly on the GPU.
MPI AND UNIFIED MEMORY

Without Unified Memory-aware MPI

Only use non Unified Memory Buffers for MPI: cudaMalloc, cudaMallocHost or malloc

Application managed non Unified Memory Buffers also allow to work around current missing cases in Unified Memory-aware MPI Implementations.
DETECTING CUDA-AWARENESS

OpenMPI (since 2.0.0):

Macro:

```c
MPIX_CUDA_AWARE_SUPPORT
```

Function for runtime decisions

```c
MPIX_Query_cuda_support()
```

Include `mpi-ext.h` for both.

See [http://www.open-mpi.org/faq/?category=runcuda#mpi-cuda-aware-support](http://www.open-mpi.org/faq/?category=runcuda#mpi-cuda-aware-support)
THANK YOU FOR YOUR ATTENTION

Questions?

Monday, May 8, 12:30 PM - 2:30 PM: L7114 - MULTI GPU PROGRAMMING WITH MPI AND OPENACC - Room LL21E

Wednesday, May 10, 2:00 PM - 3:00 PM: H7119 - CONNECT WITH THE EXPERTS: MULTI-GPU PROGRAMMING - LL Pod B

Thursday, May 11, 2:00 PM - 2:50 PM: S7356 - MVAPICH2-GDR: PUSHING THE FRONTIER OF HPC AND DEEP LEARNING - Room 211B