COMP4901Q: High Performance Computing (HPC)

Tutorial 4:
Hybrid Parallel Programming Models

TA: Mingkai TANG(mtangag@connect.ust.hk)

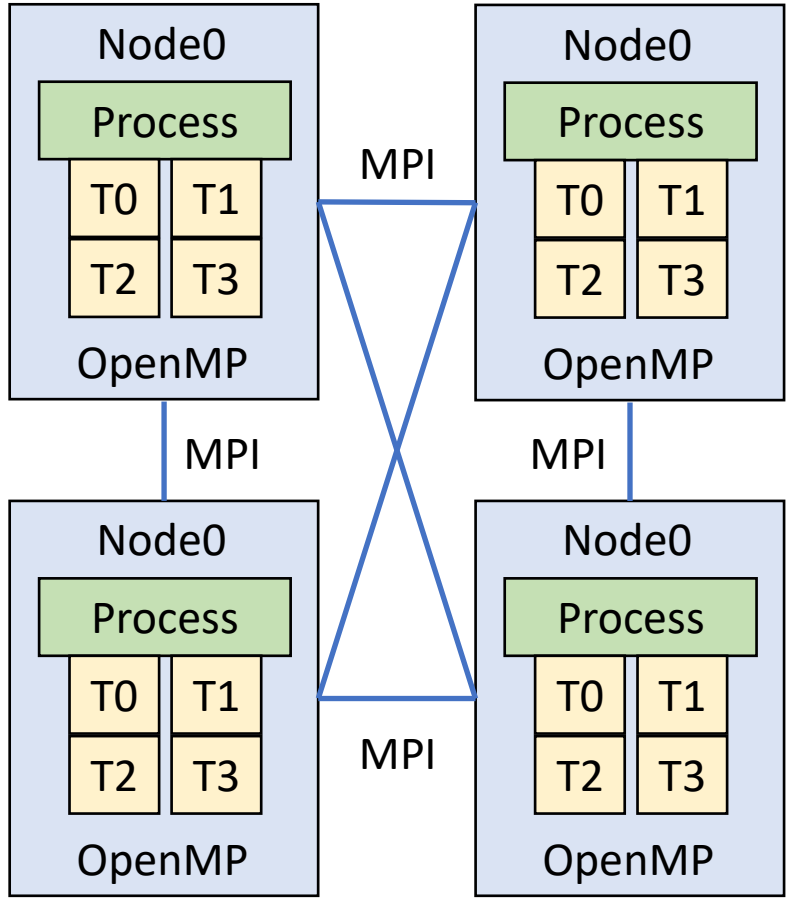
Contents

- Part 1: MPI + OpenMP
- Part 2: MPI + CUDA

Part 1: MPI + OpenMP

MPI+ OpenMP: Motivation

- Two-level Parallelization
  - Mimics hardware layout of cluster
  - MPI between nodes or CPU sockets
  - OpenMP within shared-memory nodes or processors
- Pros
  - No message passing inside of the shared-memory processor (SMP) nodes
  - No topology problem
- Cons
  - Should be careful with sleeping threads
  - Not always better than pure MPI or OpenMP



MPI Rules with OpenMP

- Special MPI init for multi-threaded MPI processes:

```
int MPI_Init_thread( int* argc, char** argv[],
                     int thread_level_required,
                     int* thead_level_provided);
int MPI_Query_thread( int* thread_level_provided);
int MPI_Is_main_thread(int* flag);
```

  - `thread_level_required` specifies the requested level of thread support.
  - Actual level of support is then returned into `thread_level_provided`.

Four Options for Thread Support

- MPI_THREAD_SINGLE
  - Only one thread will execute, EQUALS to MPI_Init
- MPI_THREAD_FUNNELED
  - Only master thread will make MPI-calls
- MPI_THREAD_SERIALIZED
  - Multiple threads may make MPI-calls, but only one at a time
- MPI_THREAD_MULTIPLE
  - Multiple threads may call MPI with no restrictions
- In most cases MPI_THREAD_FUNNELED provides the best choice for hybrid programs

Hybrid Hello

- mpi_omp_hello.cpp

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(processor_name, &namelen);

#pragma omp parallel default(shared) private(iam, np)
{
   np = omp_get_num_threads();
   iam = omp_get_thread_num();
   printf("Hybrid: Hello from thread %d out of %d from process %d out of %d on %s\n", iam, np, rank,
numprocs, processor_name);
}
```

Hybrid Array Sum: Funneled MPI calls

- mpi_omp_SumArray.cpp: Process 0

```
#pragma omp parallel
{
   if (pid == 0) {
   …
   #pragma omp master
   {
              for (int i = 1; i < np; i++) {
              MPI_Send(&elements_per_process, …);
              MPI_Send(&a[i * elements_per_process…);
     }
   }
   #pragma omp barrier
   #pragma omp for reduction(+:local_sum)
              for (int i = 0; i < elements_per_process; i++)
                 local_sum += a[i];
              }
…
}
```

Hybrid Array Sum

- mpi_omp_SumArray.cpp: Other Processes

```
#pragma omp parallel
{
…
   else {
      #pragma omp master
      {
                  MPI_Recv(&n_elements_recieved, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
                  MPI_Recv(a2, n_elements_recieved, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
               }
              #pragma omp barrier
              #pragma omp for reduction(+:local_sum)
                 for (int i = 0; i < n_elements_recieved; i++)
                    local_sum += a2[i];
   }
}
```

Hybrid Array Sum

- mpi_omp_SumArray.cpp: All Processes

```
MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Environment Setup

- Setup SSH passwordless login between nodes
  - Refer to lab3 slides

- Check & Install OpenMPI, OpenMP if not
  - OpenMPI-3.0.0 (/usr/local/software/openmpi-3.0.0)
  - OpenMP 3.0

Compilation

- OpenMPI wrapper script with OpenMP -fopenmp switch
  - mpic++ -fopenmp -o mpi_omp_hello mpi_omp_hello.cpp
  - mpic++ -fopenmp -o mpi_omp_SumArray mpi_omp_SumArray.cpp

Execution

- Nearly same with pure MPI. Use mpiexec or mpirun
- With default thread num in OMP sections
  - mpiexec --host csl2wk01:2,csl2wk02:2 ./mpi_omp_hello
- Specify OMP_NUM_THREADS
  - mpiexec --host csl2wk01,csl2wk02 -x OMP_NUM_THREADS=3 ./mpi_omp_hello
  - -x: Export an environment variable to the remote nodes before executing the program, optionally specifying a value
- Specify OMP_NUM_THREADS for different hosts
  - mpiexec -n 1 --host csl2wk01 -x OMP_NUM_THREADS=3 ./mpi_omp_hello : -n 2 --host csl2wk02:2 -x OMP_NUM_THREADS=2 ./mpi_omp_hello

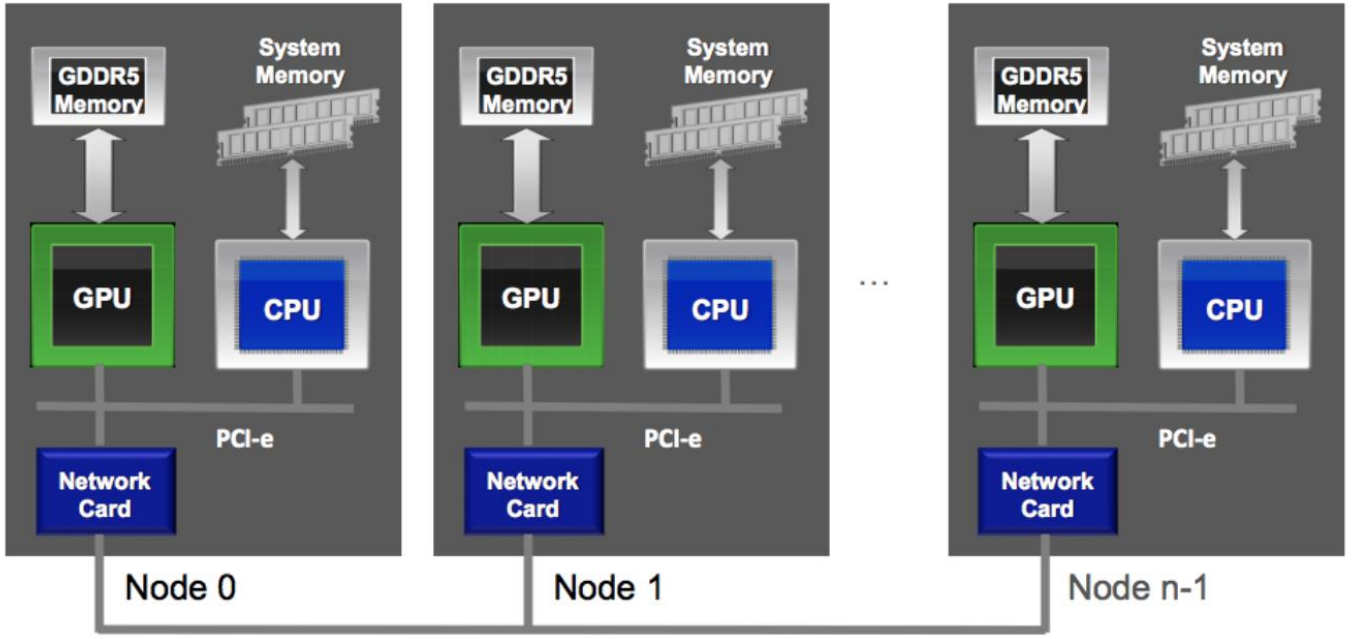
Practice

- Implement the code of vector addition using MPI and OpenMP
- Sample code：　./practice/mpi_openmp/vector_addition.cpp
- Solution: ./practice/mpi_openmp/vector_addition_solution.cpp

Part 2: MPI + CUDA

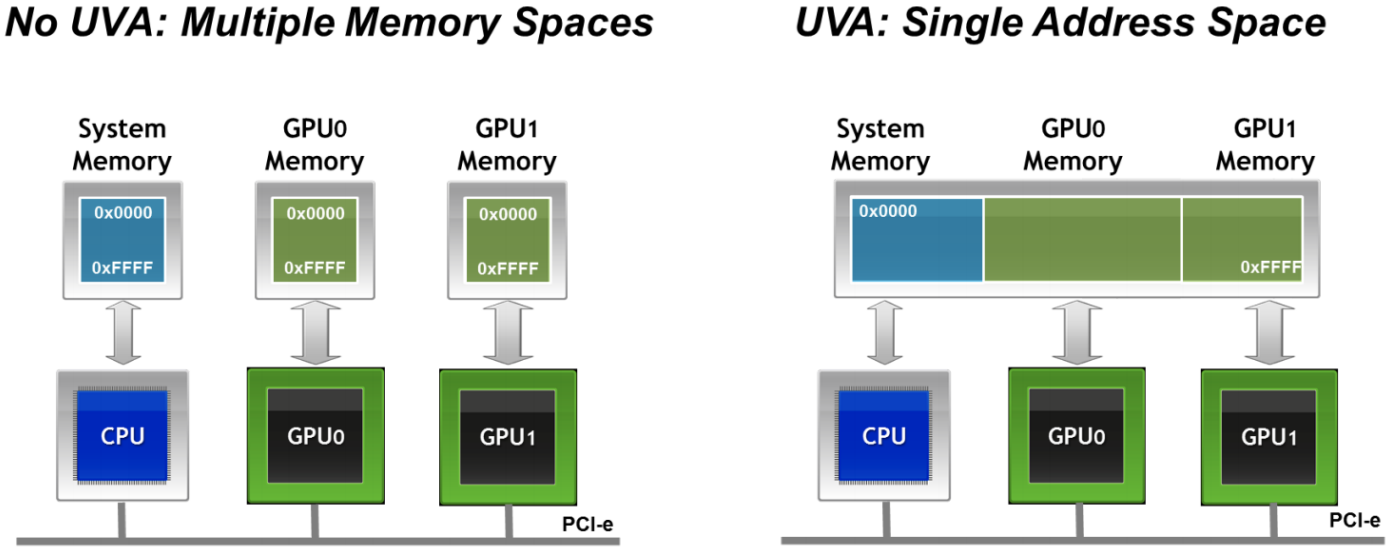
Hybrid CUDA and MPI: Motivation

- MPI is easy to exchange data located at different processors
  - CPU <-> CPU: Traditional MPI
  - GPU <-> GPU: CUDA-Aware MPI
- MPI+CUDA makes the application run more efficiently
  - All operations that are required to carry out the message transfer can be pipelined
  - Acceleration technologies like GPUDirect can be utilized by the MPI library transparently to the user.



Unified Virtual Addressing (UVA)

- No UVA: Separate Address Spaces vs. UVA



- UVA: 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

UVA Data Exchange with MPI

UVA

```
//MPI Rank 0
MPI_Send(s_buf_d, size, …);

//MPI Rank n-1
MPI_Recv(r_buf_d, size, …);
```

CUDA-aware MPI is required!

Non-UVA

```
//MPI Rank 0
cudaMemcpy(s_buf_h, s_buf_d, size,…);
MPI_Send(s_buf_h,size,…);

//MPI Rank n-1
MPI_Recv(r_buf_h, size, …);
cudaMemcpy(r_buf_d, r_buf_h, size,…);
```

Example: Matrix Multiplication

- The root process generates two random matrices of input size and stores them in a 1-D array in Row-major order.
- The first matrix is divided into columns depending on the number of input processors and each part is sent to a separate GPU (MPI_Scatter)
- The second matrix (Matrix B) is broadcasted to all nodes and copied on all GPUs to perform computation. (MPI_Bcast)
- Each GPU computes its own part of the result matrix and sends the result back to the root process
- Results are gathered into a resultant matrix. (MPI_Gather)

Code

- Without UVA. Send the data in the host memory.
  - matvec.cu

- With UVA. Send the data in the device memory.
  - matvec_uva.cu

matvec.cu(Without UVA)

- **1. Generate the data in the master process:**
- Status = IntializingMatrixVectors(&MatrixA, &MatrixB, &ResultVector, RowsNo, ColsNo, RowsNo2, ColsNo2);

- **2. Send data to different processes in host memory:**
- MPI_Bcast(MatrixB, matrixBsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
- MPI_Scatter(MatrixA, ScatterSize * ColsNo, MPI_FLOAT, MyMatrixA, ScatterSize * ColsNo, MPI_FLOAT, 0, MPI_COMM_WORLD);

matvec.cu(Without UVA)

- **3. Allocate the memory in the device memory in each process:**
- cudaMalloc((void **)&DeviceMyMatrixA, ScatterSize * ColsNo * sizeof(float)));
- cudaMalloc((void **)&DeviceMatrixB, matrixBsize*sizeof(float)));
- cudaMalloc((void **)&DeviceMyResultVector, elements * sizeof(float)));

- **4. Copy the Data from host to device in each process:**
- cudaMemcpy((void *)DeviceMyMatrixA, (void *)MyMatrixA, ScatterSize * ColsNo * sizeof(float), cudaMemcpyHostToDevice);
- cudaMemcpy((void *)DeviceMatrixB, (void *)MatrixB, matrixBsize*sizeof(float), cudaMemcpyHostToDevice);

- **5. Do the calculation in each process:**
- MatrixVectorMultiplication<<<1, 256>>>(DeviceMyMatrixA, DeviceMatrixB, DeviceMyResultVector, RowsNo, ColsNo, RowsNo2, ColsNo2, ColsNo, ScatterSize, BLOCKSIZE, MyRank, NumberOfProcessors);

matvec.cu(Without UVA)

- **6. Copy the result from device to host in each process :**
- cudaMemcpy((void *)MyResultMatrix, (void *)DeviceMyResultVector, elements * sizeof(float), cudaMemcpyDeviceToHost);

- **7. Gather the result:**
- MPI_Gather(MyResultMatrix,elements, MPI_FLOAT, ResultVector, elements, MPI_FLOAT, 0, MPI_COMM_WORLD);

matvec_uva.cu(With UVA)

- **1. Generate the data in the master process:**
- Status = IntializingMatrixVectors(&MatrixA, &MatrixB, &ResultVector, RowsNo, ColsNo, RowsNo2, ColsNo2);

- **2. Allocate the memory on the device memory in the master process:**
- cudaMalloc((void **)&DeviceRootMatrixA, RowsNo * ColsNo * sizeof(float)) ;
- cudaMalloc((void **)&DeviceRootResultVector, RowsNo * ColsNo2 * sizeof(float)) ;

- **3. Copy the Data from host to device in the master process :**
- cudaMemcpy((void *)DeviceRootMatrixA, (void *)MatrixA, RowsNo * ColsNo * sizeof(float), cudaMemcpyHostToDevice);

matvec_uva.cu(With UVA)

- **4. Allocating the memory in the device memory in each process:**
- cudaMalloc((void **)&DeviceMyMatrixA, ScatterSize * ColsNo * sizeof(float)) ;
- cudaMalloc((void **)&DeviceMatrixB, matrixBsize*sizeof(float)));
- cudaMalloc((void **)&DeviceMyResultVector, elements * sizeof(float)) ;

- **5. Send data to different processes in device memory:**
- MPI_Bcast(DeviceMatrixB, matrixBsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
- MPI_Scatter(DeviceRootMatrixA, ScatterSize * ColsNo, MPI_FLOAT, DeviceMyMatrixA, ScatterSize * ColsNo, MPI_FLOAT, 0, MPI_COMM_WORLD);

matvec_uva.cu(With UVA)

- **6. Do the calculation in each process:**
- MatrixVectorMultiplication<<<1, 256>>>(DeviceMyMatrixA, DeviceMatrixB, DeviceMyResultVector, RowsNo, ColsNo, RowsNo2, ColsNo2, ColsNo, ScatterSize, BLOCKSIZE, MyRank, NumberOfProcessors);

- **7. Gather the result in the device memory in the master process:**
- MPI_Gather(DeviceMyResultVector, elements, MPI_FLOAT, DeviceRootResultVector, elements, MPI_FLOAT, 0, MPI_COMM_WORLD);

- **8. Copy the result from device to host in the master process :**
- cudaMemcpy((void *)ResultVector, (void *)DeviceRootResultVector, RowsNo * ColsNo2 * sizeof(float), cudaMemcpyDeviceToHost);

Environment Setup

- CUDA 8 and OpenMP 3.0
  - `setenv PATH "${PATH}:/usr/local/cuda-8.0/bin/“`
- G++ 4.8.5

Compilation

- 1. Put both MPI and CUDA code in a single file, matvec.cu.
- This program can be compiled using *nvcc*, which internally uses gcc/g++ to compile the C/C++ code, and linked to MPI library:
  - `/usr/local/cuda/bin/nvcc -Xcompiler -g -w -I.. -I /usr/local/software/openmpi/include/  -L /usr/local/software/openmpi/lib –lmpi matvec.cu -o newfloatmatvec`

Compilation

- 2. Have MPI and CUDA code separate in two files: *main.c* and *multiply.cu* respectively. These two files can be compiled using *mpicc/mpic++*, and *nvcc* respectively into object files (.o) and combined into a single executable file using *mpicc/mpic++*.
- 3. This third option is an opposite compilation of the first one, using *mpicc/mpic++*, meaning that you have to link to your CUDA library.

Execution

- Use mpiexec. If compiled with nvcc, include the OpenMPI lib path in LD_LIBRARY_PATH
  - `mpiexec --host csl2wk26:1,csl2wk25:1 -x LD_LIBRARY_PATH=/usr/local/software/openmpi/lib:$LD_LIBRARY_PATH ./newfloatmatvec 4 3 3 4 -p -v`

Practice

- Implement the code of vector addition using MPI and CUDA
- Without UVA
  - Sample code: ./practice/mpi_cuda/vector_addition.cu
  - Solution: ./practice/mpi_cuda/vector_addition_solution.cu
- With UVA
  - Sample code: ./practice/mpi_cuda/ vector_addition_uva.cu
  - Solution: ./practice/mpi_cuda/ vector_addition_uva.cu

Thank You!