Practical Parallel Computing (実践的並列コンピューティング)

Part 3: MPI

No 4: Communication Overlap etc.

June 3, 2024

Toshio Endo

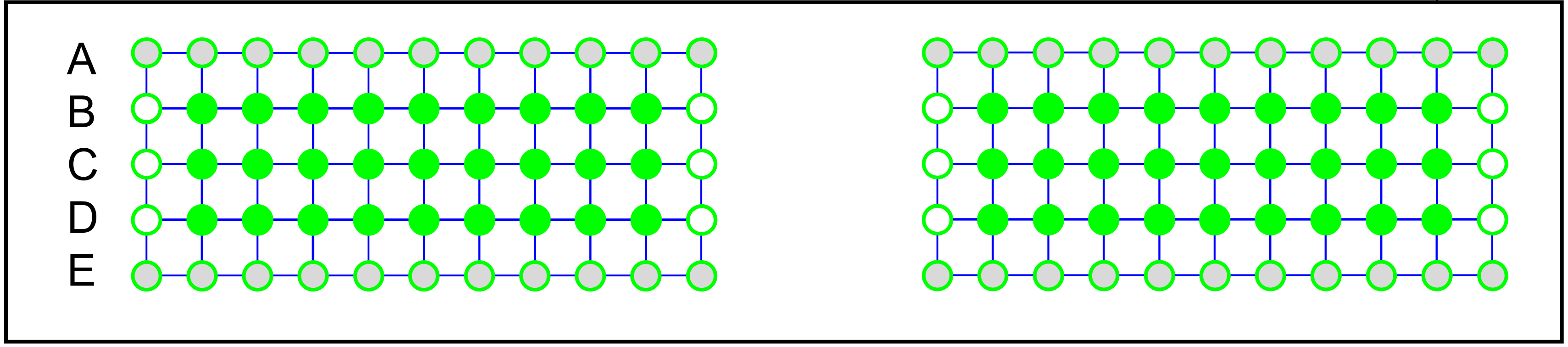
School of Computing & GSIC

endo@is.titech.ac.jp

Improving MPI diffusion by Overlapping of Communication

related to [M1], but optional

Overview of MPI "diffusion" (See MPI (2) Slides)



```
for (t = 0; t < nt; t++) {
  if (rank > 0) Send B to rank-1
  if (rank < size-1) Send D to rank+1
  if (rank > 0) Recv A from rank-1
  if (rank < size-1) Recv E from rank+1
  Computes points in rows B-D
  Switch old and new arrays
}
```

(1) Communication in "old" array

(2) Computation "old" array ⇒ "new" array

Actually this should be fixed to avoid deadlock

Considering Performance of MPI Programs

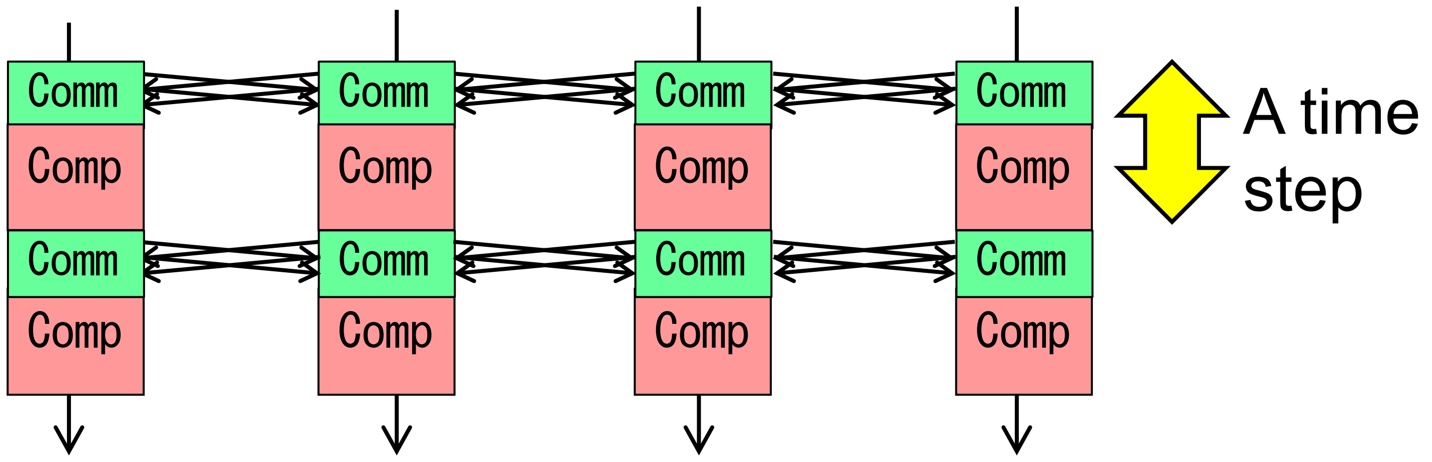
(Simplified) Execution time of an MPI program =

Computation time ← including memory access

+ Communication time ← including congestion

+ Others ← load imbalance, I/O…

Behavior of stencil computations on MPI



Computation Time & Communication Time

- Let us compare them for some samples

Sample Program	Computation Cost	Communication Cost
mm	O(mnk/p)	O(0)
mm (memory-reduced)	O(mnk/p)	O(mk) ← When A is sent
diffusion	O(NX NY NT /p)	O(NX NT) ← When NY is divided

per process

- In these samples, communication costs look smaller?

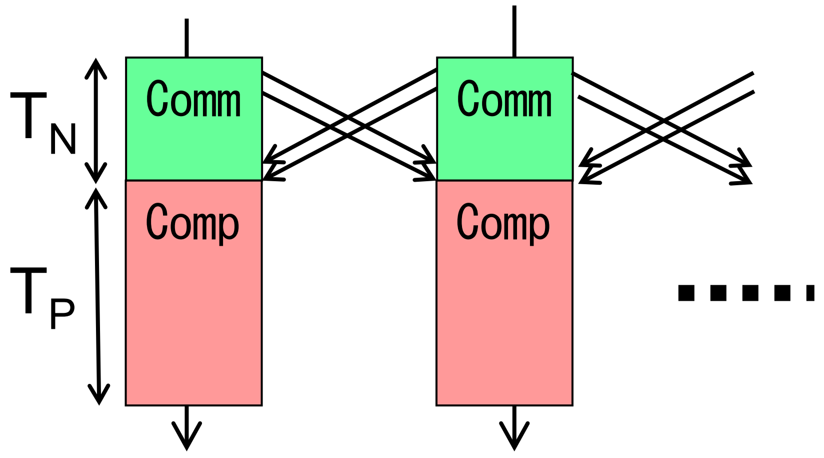
➔ In most computer systems,
O(N) communication is much slower than O(N) computation

➔ Reducing effects of communication is important

Idea of Overlapping

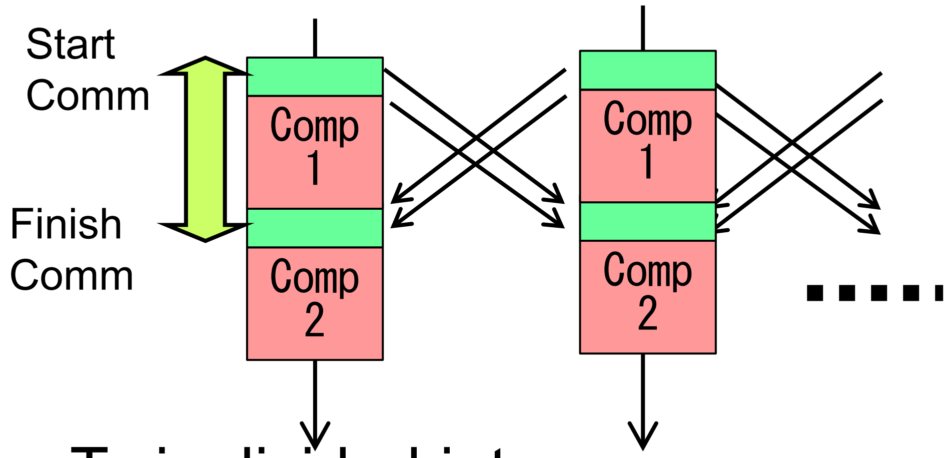
If "some computations" do not require contents of message, we may start them beforehand

Without overlap



$T=T_N+T_P$

With overlap



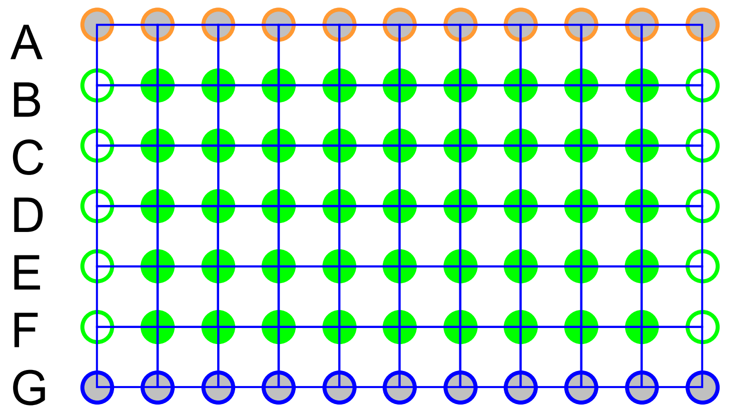
T_P is divided into

- T_{P1}: can be overlapped
- T_{P2}: cannot be overlapped

$T=\max(T_N,T_{P1})+T_{P2}$

Overlapping in Stencil Computation (related to [M1], but not requied)

When we consider data dependency in detail, we can find computations that do not need data from other processes



Rows C, D, E do not need data from other processes
→ They can be computed without waiting for finishing communication

On the other hand, rows B, F need received data

For such purposes, non-blocking communications (MPI_Isend, MPI_Irecv…) are helpful again

Implementation without Overlapping (Not Fast!)

```
for (t = 0; t < nt; t++) {
   Start Send B to rank-1, Start Send F to rank+1
    (MPI_Isend)
   Start Recv A from rank-1, Start Recv G from
    rank+1 (MPI_Irecv)
   Waits for finishing all communications
    (MPI_Wait for 4 times)
  Compute rows B--F
  Switch old and new arrays
}
```

T_N

T_P

$$T=T_N+T_P$$

Implementation with Overlapping

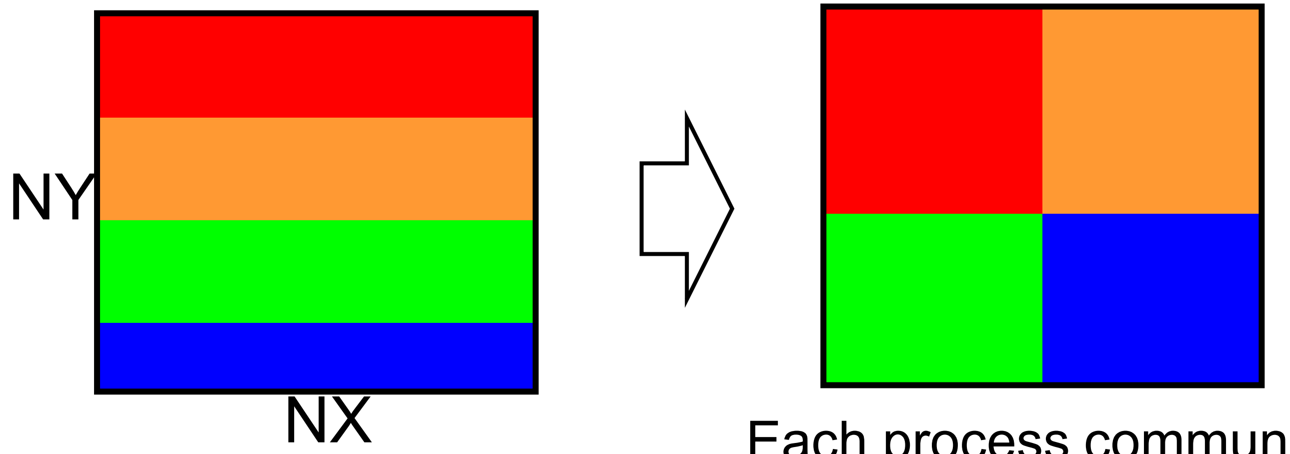
```
for (t = 0; t < nt; t++) {
   Start Send B to rank-1, Start Send F to rank+1
   (MPI_Isend)
   Start Recv A from rank-1, Start Recv G from
   rank-1 (MPI_Irecv)
  Compute rows C--E
   Waits for finishing all communications
   (MPI_Wait)
  Compute rows B, F
  Switch old and new arrays
}
```

computations are divided

$T = \max(T_N,T_{P1})+T_{P2} < T_N+T_{P1}+T_{P2} = T_N+T_P$

Another Improvement: Reducing Communication Amounts

Multi-dimensional division may reduce communication



Each process communicate with upper/lower/right/left processes

- Comp: O(NY NX NT/p)
- Comm: O(NX NT)

per process

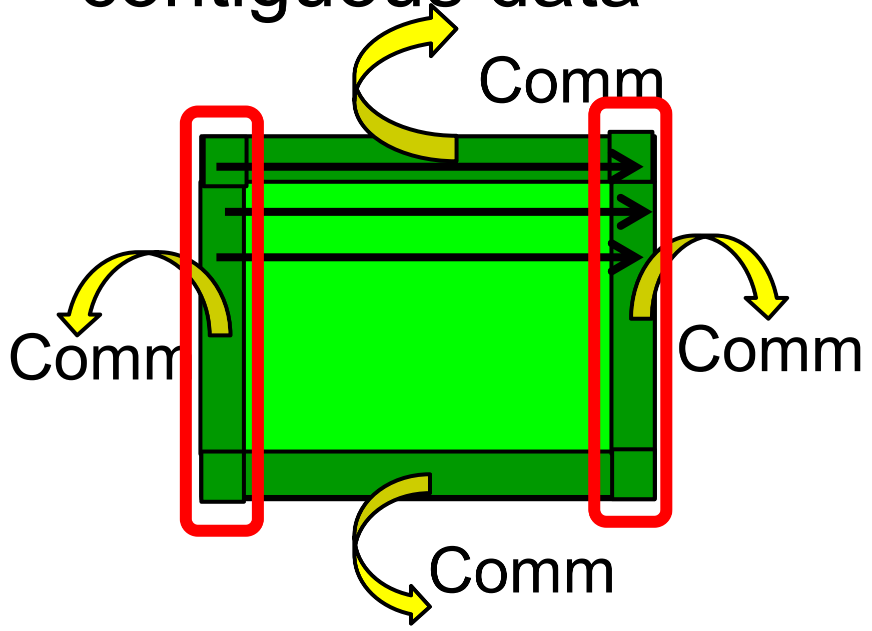
- Comp: O(NY NX NT/p)
- Comm: O((NY+NX)/$p^{1/2}$NT)

per process

→ Comm is reduced

Multi-dimensional division and Non-contiguous data (1)

- MD division may need communication of non-contiguous data



In Row-major format,
we need send/recv of non-contiguous data for left/right borders

But "fragmented communication" degrades performance! (since Latency > 0)
How do we do?

Multi-dimensional division and Non-contiguous data (2)

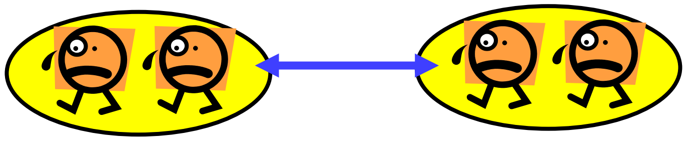
Solution (1):

- Before sending, copy non-contiguous data into another contiguous buffer
- After receiving, copy contiguous buffer to non-contiguous area

Solution (2):

- Use MPI_Datatype
  - Skipped in the class; you may use Google

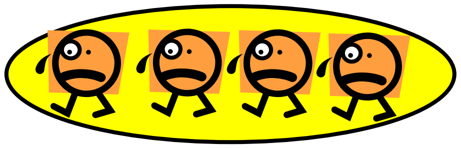
Hybrid Programming with MPI+OpenMP



Speed of mm-mpi is improved

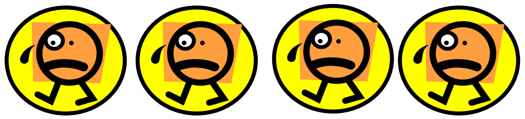
MPI+OpenMP

OpenMP:
1process has multiple threads



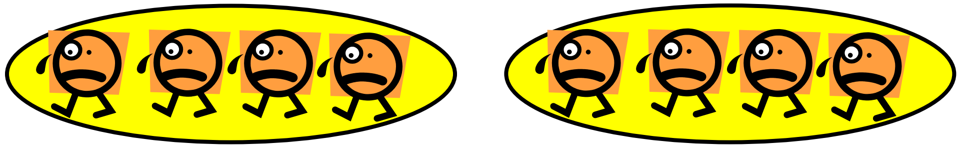
Only 1 node can be used

MPI :
Multiple processes are used
(usually) Each has 1 thread



Multiple nodes can be used

MPI +OpenMP:
Multiple processes are used
Each has multiple threads



Multiple nodes can be used

Sample: /gs/bs/tga-ppcomp/24/mm-mpi-omp/

Compiling mm-mpi-cuda Sample

```
module load intel-mpi    [Do once after login]
cd ~/ppc24
cp -r /gs/bs/tga-ppcomp/24/mm-mpi-omp  .
cd mm-mpi-omp
make
[An executable file “mm” is created]

export OMP_NUM_THREADS=8
mpiexec -n 2 ./mm 2000 2000 2000
```

Number of threads per process

Number of processes

Messages between Processes

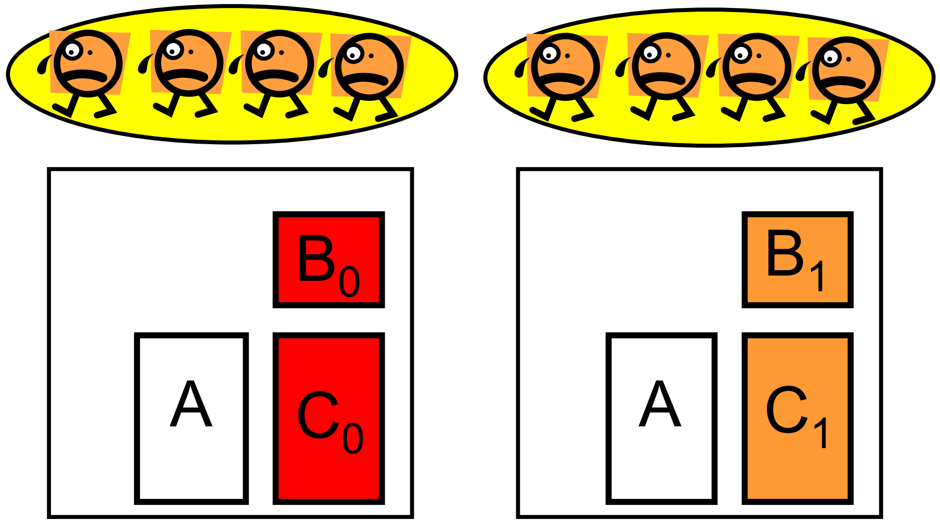
- Basically MPI communication should be out of OpenMP parallel region
  - In mm-mpi, processes call MPI_Barrier

- If you want to do communication freely, please try google MPI_Init_thread()

Speed of mm-mpi-cuda Is Better Than mm-mpi

- In (simple) mm-mpi, each process has entire A
- mm-mpi-cuda works similarly, but the number of processes can be reduced

➔ Due to cache locality, speed may be improved than mm-mpi



There are 2 copies of A, not 8

Job Submission of mm-mpi-omp

In job.sh sample, 2 cpu_16 node partitions are allocated

job.sh

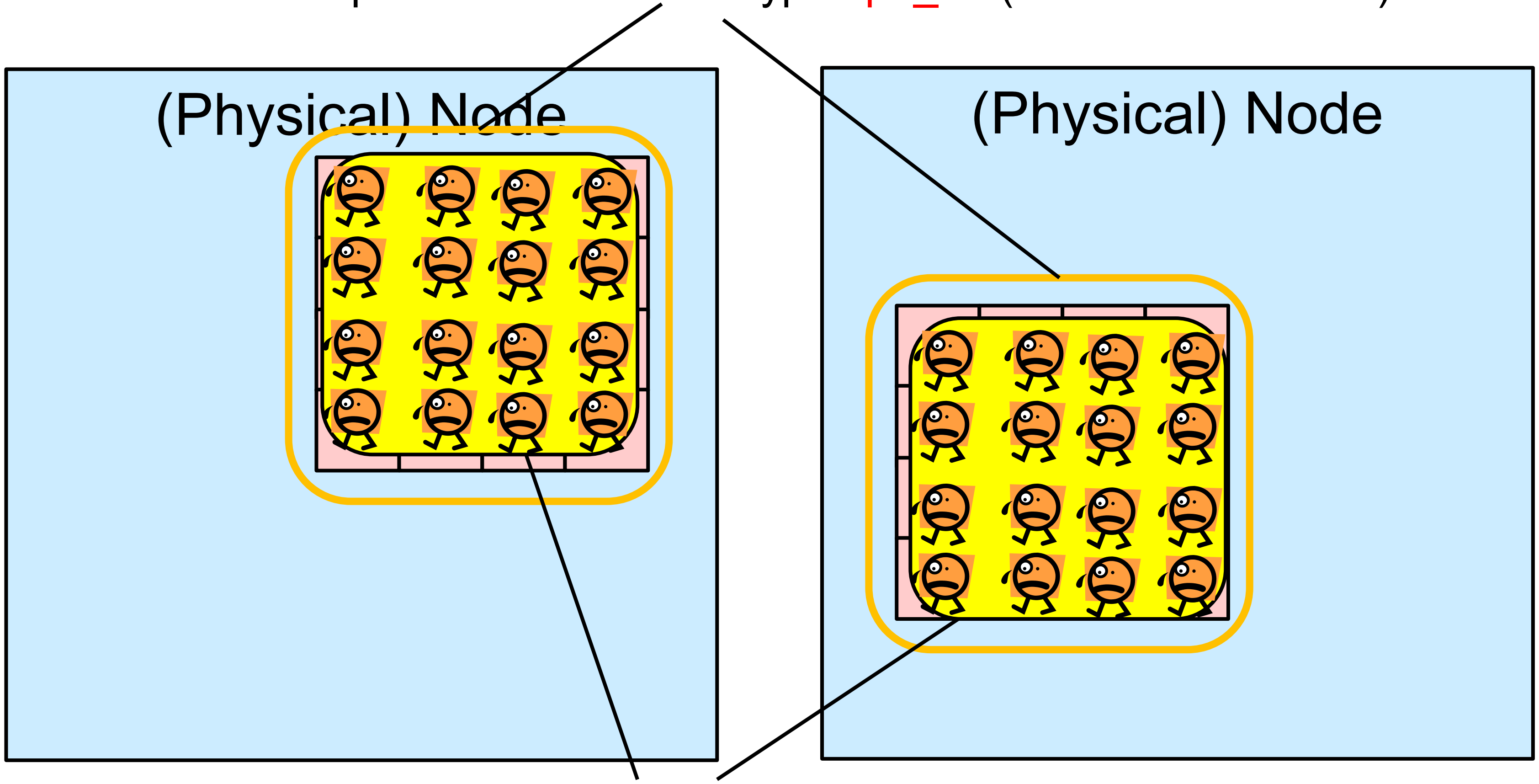
```
#!/bin/sh
#$ -cwd
#$ -l cpu_16=2
#$ -l h_rt=0:10:00

module load intel-mpi

export OMP_NUM_THREADS=16
mpiexec -n 2 -ppn 1 ./mm 2000 2000 2000
```

16 threads per process

2 processes, 1 processes per node

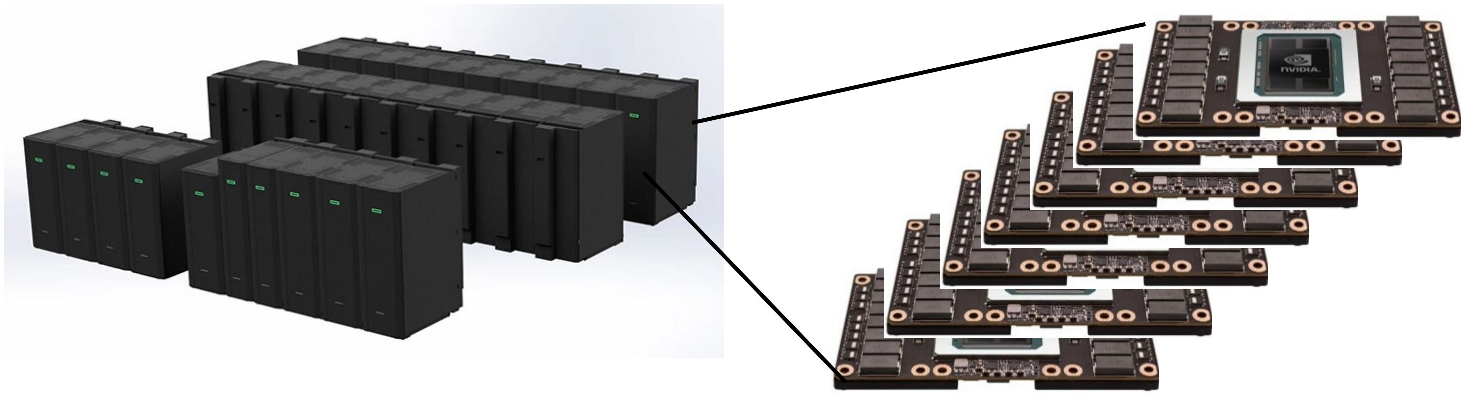
What Happens on TSUBAME with mm-mpi-omp/job.sh

(1) -l cpu_16=2 ➔ Job scheduler allocates 2 node partitions. Each has type cpu_16 (with 16 CPU cores)

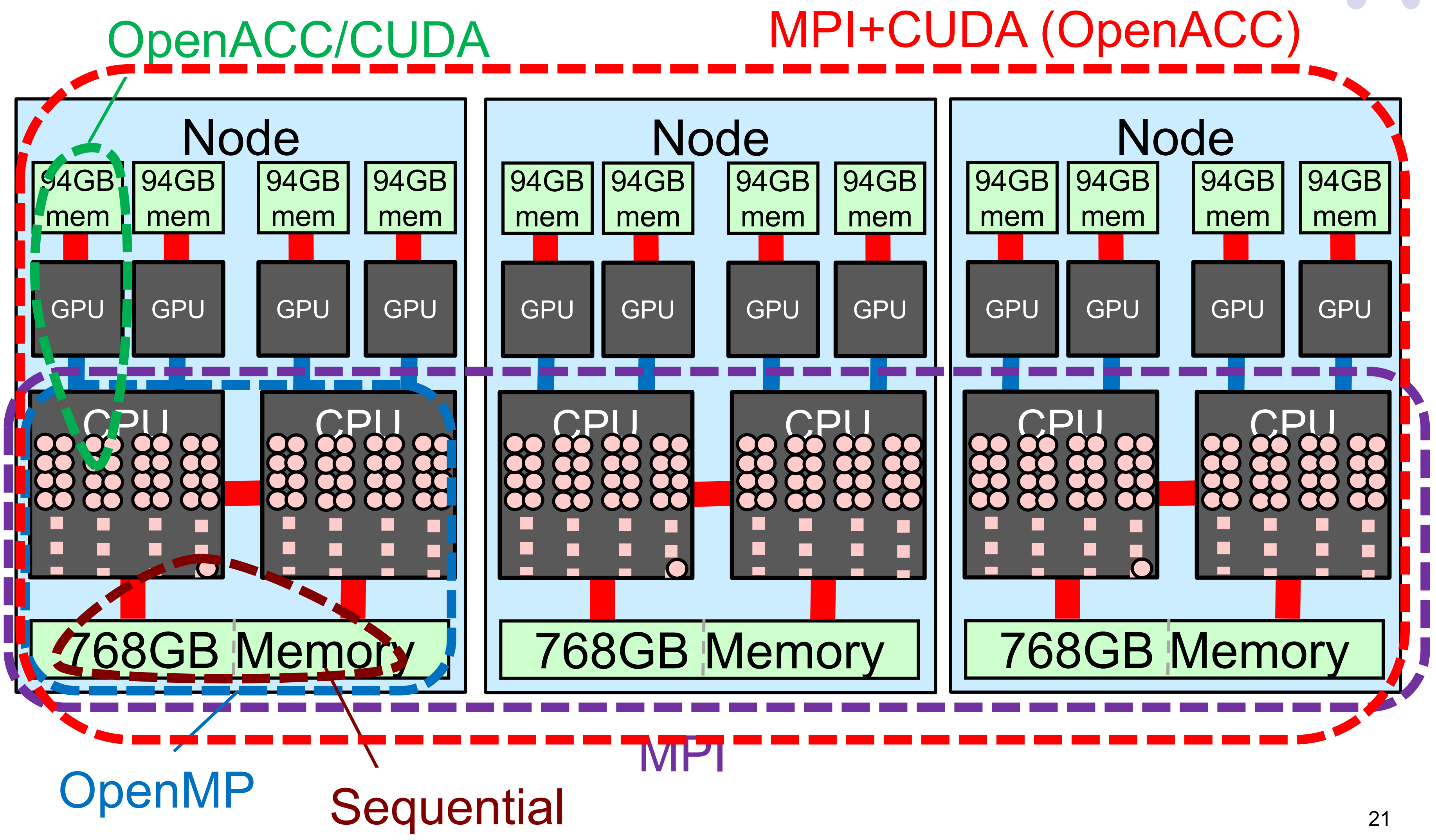


(2) -n 2 -ppn 1 ➔ mpiexec invokes 2 process, 1 processes per node (partition)

Using Multiple GPUs with MPI+CUDA



Parallel Programming Methods on TSUBAME



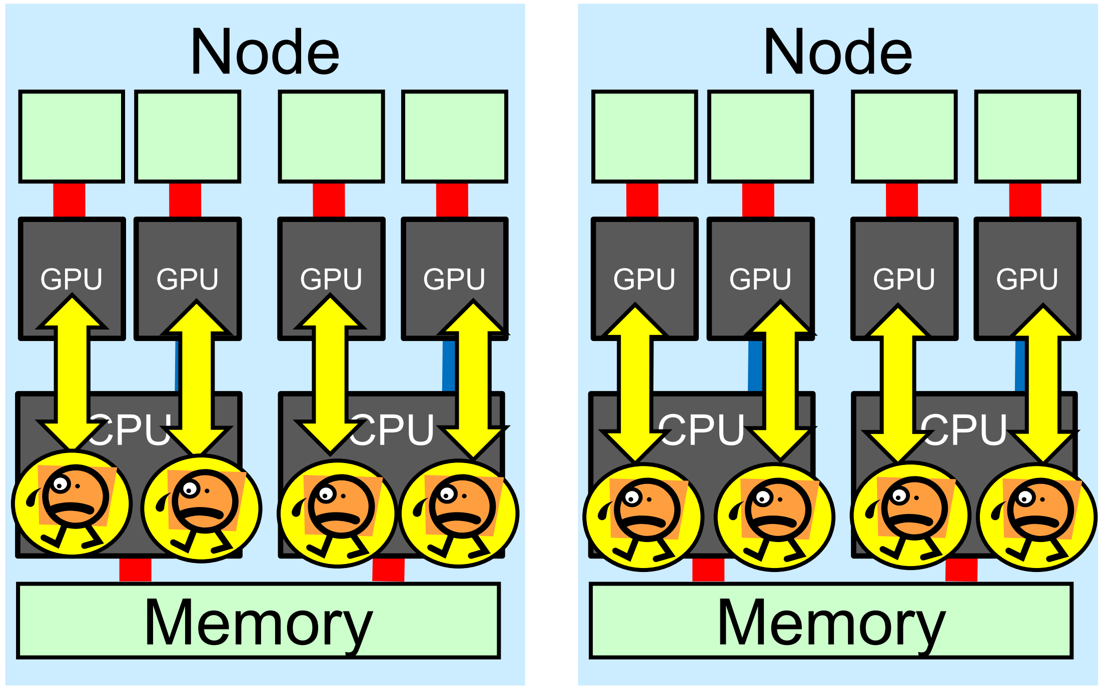
Using Multiple GPUs

- GPUs on multiple nodes
  - MPI + CUDA
    - 1 process uses 1 GPU (mm-mpi-cuda sample)
- GPUs on a single node

  (Up to 4 GPUs on a TSUBAME4.0 node_f)
  - MPI+CUDA
  - OpenMP + CUDA
    - 1 thread uses 1 GPU
  - 1 thread switches multiple GPUs
    - cudaSetDevice() is called many times

Using Multiple GPUs with MPI

- Basic idea:
  (1) Start processes on multiple nodes by MPI
  (2) Each process uses its local GPU by CUDA



Sample: /gs/bs/tga-ppcomp/24/mm-mpi-cuda/

Compiling mm-mpi-cuda Sample

```
module load cuda intel-mpi    [Do once after login]
cd ~/ppc24
cp -r /gs/bs/tga-ppcomp/24/mm-mpi-cuda  .
cd mm-mpi-cuda
make
[An executable file “mm” is created]
```

In this Makefile,

- nvcc is used as the compiler
- mpicxx is used as the linker, with CUDA libraries

This Makefile is for current TSUBAME, so you will need to modify it for other systems

Executing mm-mpi-cuda

- Interactive use is only for one node

→ To use multiple nodes, job submission is required

qsub job2q.sh ➔ node_q (1GPU) x 2 are used ➔ 2GPUs in total

qsub job2f.sh ➔ node_f (4GPU) x 2 are used ➔ 8 GPUs in total

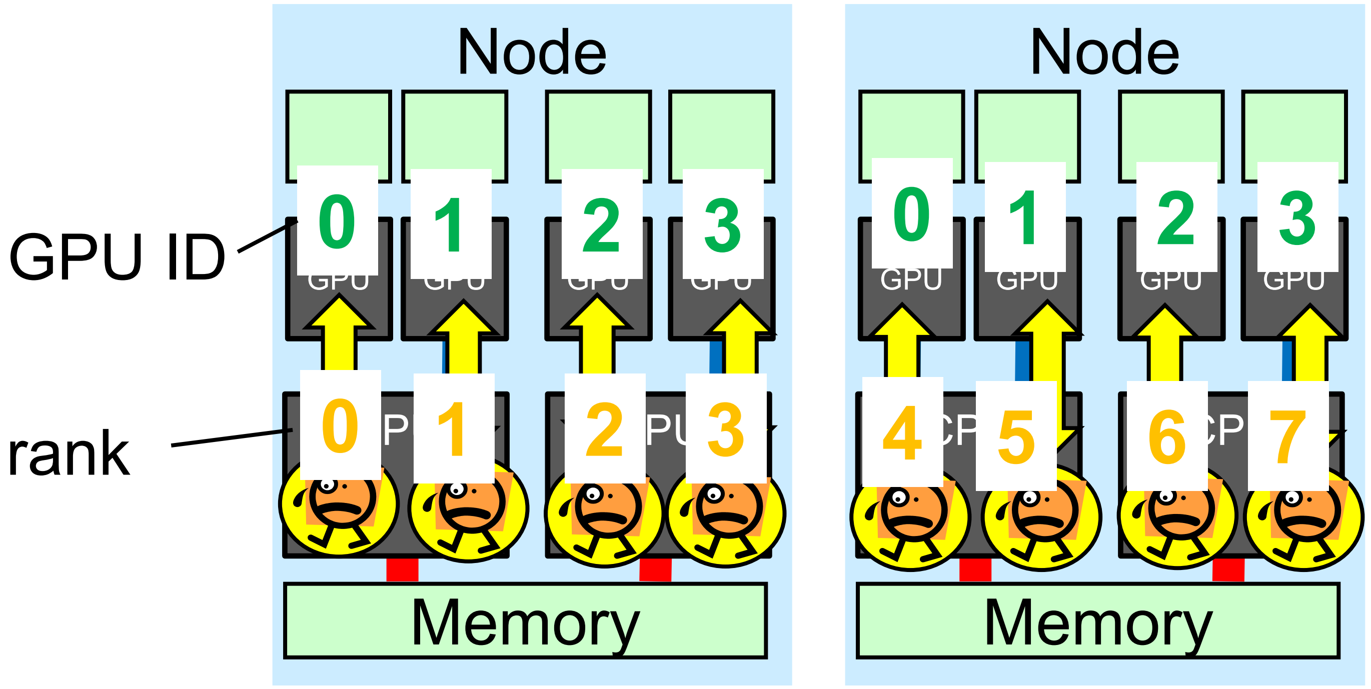
job2f.sh

```
#!/bin/sh
#$ -cwd
#$ -l node_f=2
#$ -l h_rt=0:10:00

module load cuda openmpi

mpiexec -n 8 -ppn 4 ./mm 2048 2048 2048
```

8 processes, 4 processes per node

Using Multiple GPUs per node (1)

- In case of “node_f=2”, each node has 4 GPU
  - In default, all processes use “GPU 0” on the node → slow ☹
- Each process should determine GPU ID by (rank%4)



Using Multiple GPUs per Node (2)

- node_f or node_n has multiple GPUs (4 or 2)

- Each process should use distinct GPUs

➔ In mm.cu, cudaSetDevice(int dev) is called first
  - specifies the GPU to be used
  - dev: GPU ID in the node (0, 1, 2…)
    - In this sample, GPU ID is computed as (rank % num of devices)

From cudaGetDeviceCount()
➔ 1 on node_q
➔ 2 on node_h
➔ 4 on node_f

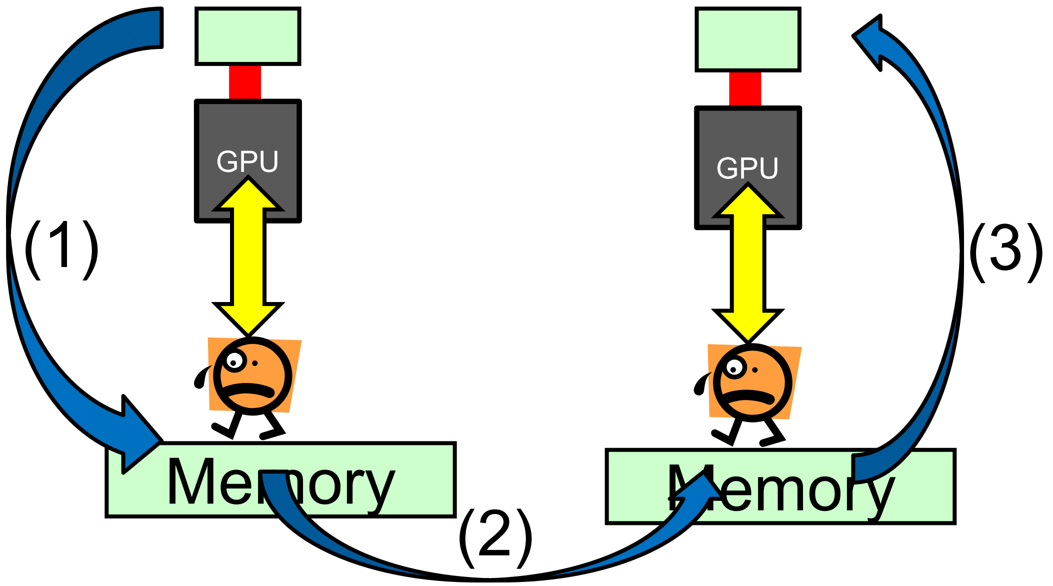
Data Transfer (1)

- mm sample does not use communication
- If we want to do, the basic method is

  (1) Copy data on GPU memory to CPU (cudaMemcpy)

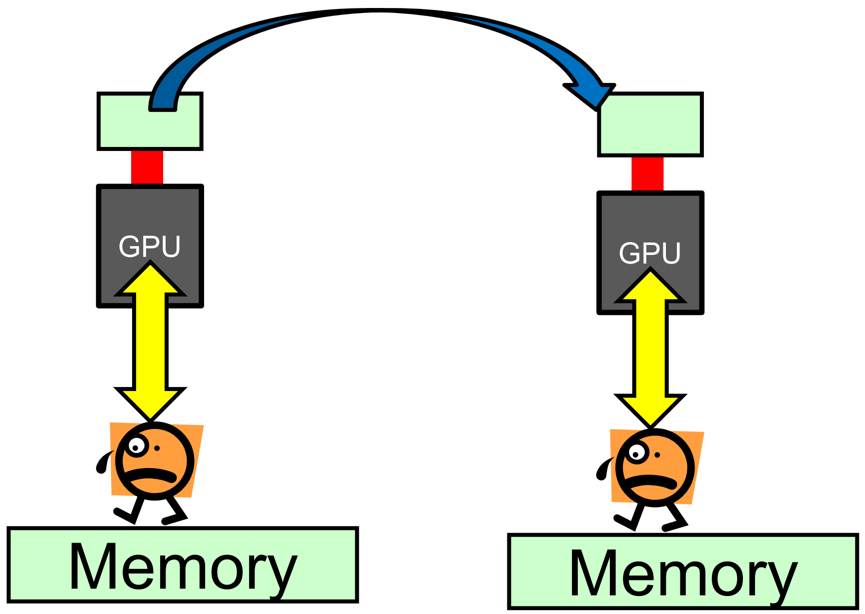
  (2) Transfer between processes (MPI_Send/MPI_Recv)

  (3) Copy data on CPU memory to GPU (cudaMemcpy)



Data Transfer (2)

- Recent MPI supports GPU direct
- For direct communication on GPU memory
  - MPI_Send(DP, …) and MPI_Recv(DP, ….) can use pointers on device memory



All Parts are Almost Finished

- Part 1: Shared memory parallel programming with OpenMP
- Part 2: GPU programming with OpenACC and CUDA
- Part 3: Distributed memory parallel programming with MPI

Many common strategies towards faster software:

- To understand source of bottleneck
- Reducing computation and communication
- Overlapping computation and communication
- To understand property of architecture

Assignments in MPI Part (Abstract)

Choose *one of* [M1]—[M3], and submit a report

Due date: June 13 (Thursday) (sorry, not June 14!)

[M1] Parallelize “diffusion” sample program by MPI.

- Be careful for deadlock

[M2] Improve mm-mpi sample in order to reduce memory consumption.

[M3] (Freestyle) Parallelize *any* program by MPI.

For more detail, please see 3-1 slides

- Thank you for participating in practical parallel computing

Today, we will go to the TSUBAME tour