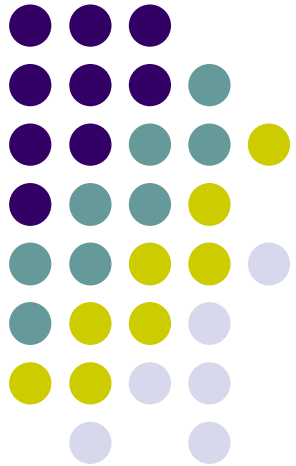
Practical Parallel Computing (実践的並列コンピューティング)



Part 3: MPI

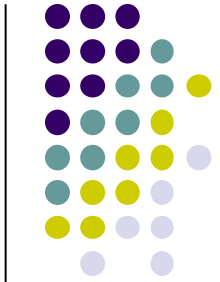
No 4: Communication Overlap etc.

June 1, 2023

Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

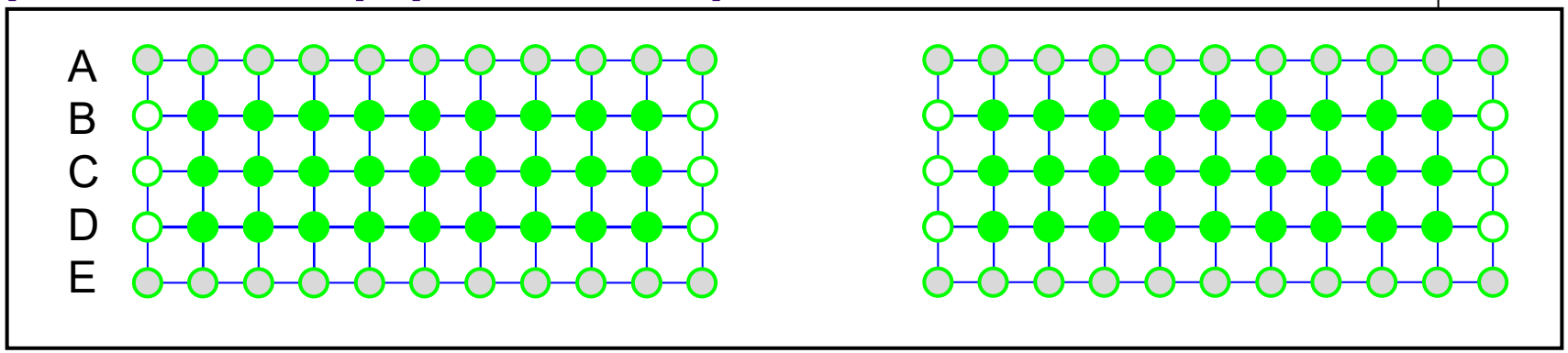


Improving MPI diffusion by Overlapping of Communication

related to [M1], but optional

Overview of MPI “diffusion”

(See MPI (2) Slides)



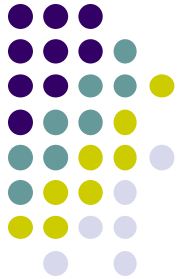

```
for (t = 0; t < nt; t++) {
  if (rank > 0) Send B to rank-1
  if (rank < size-1) Send D to rank+1
  if (rank > 0) Recv A from rank-1
  if (rank < size-1) Recv E from rank+1
  Computes points in rows B-D
  Switch old and new arrays
}
```

(1) Communication in “old” array

(2) Computation “old” array ⇒ “new” array

Actually this should be fixed to avoid deadlock

Considering Performance of MPI Programs



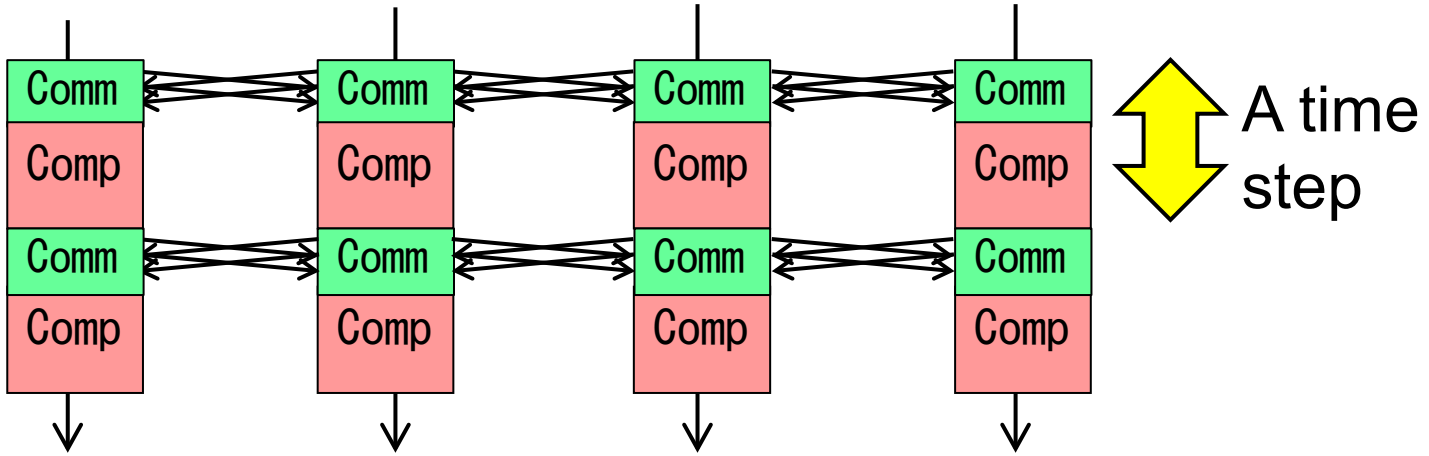
(Simplified) Execution time of an MPI program =

Computation time ← including memory access

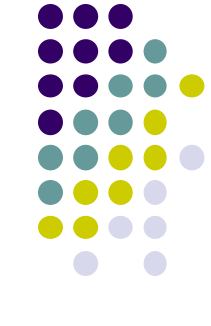
\+ Communication time ← including congestion

\+ 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?

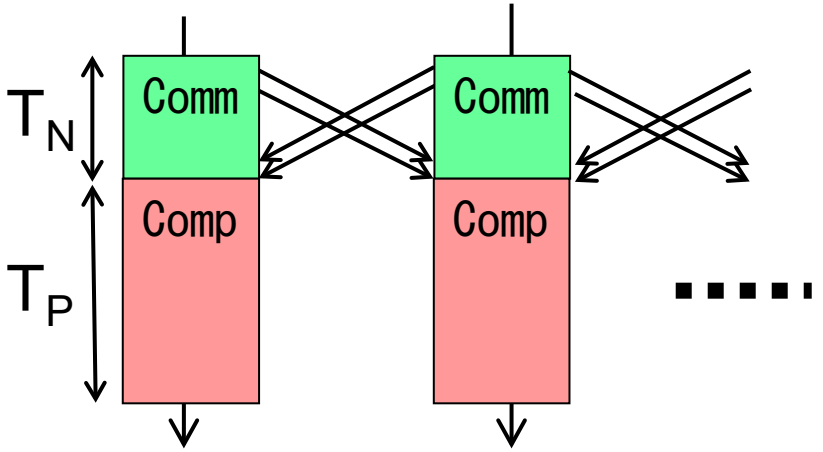
➔ In most computer systems,
O(N) communication is much slower than O(N) computation

➔ Reducing effects of communication is important

Idea of Overlapping

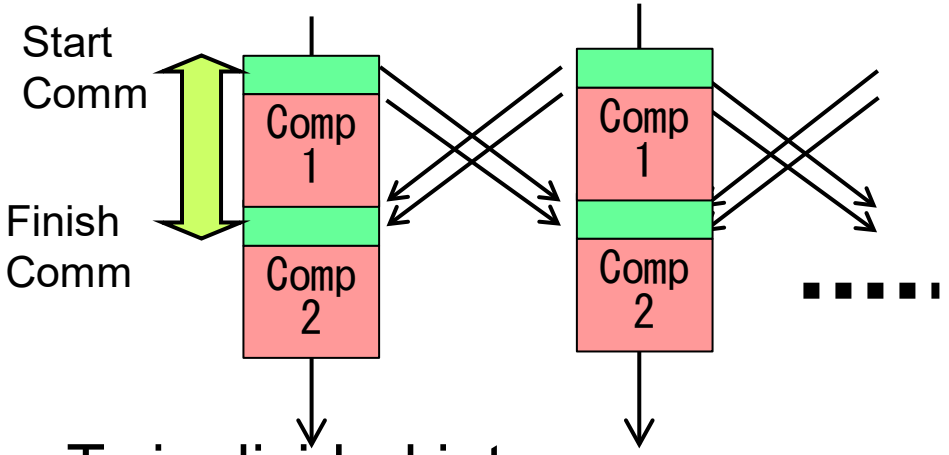
If "some computations" do not require contents of message, we may start them beforehand

Without overlap



$T=T_N+T_P$

With overlap



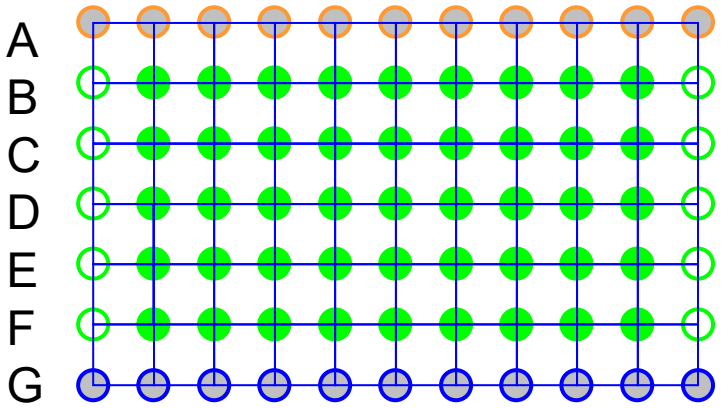
T_P is divided into

- 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

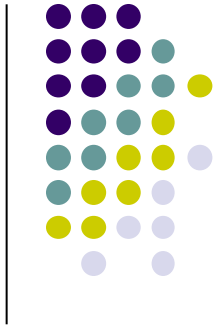


Rows C, D, E do not need data from other processes → They can be computed without waiting for finishing communication

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!)



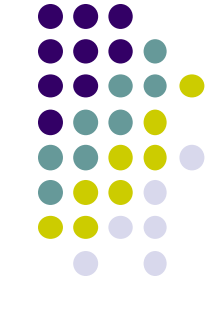
```
for (t = 0; t < nt; t++) {
  Start Send B to rank-1, Start Send F to rank+1
   (MPI_Isend)
  Start Recv A from rank-1, Start Recv G from
   rank+1 (MPI_Irecv)
  Waits for finishing all communications
   (MPI_Wait for 4 times)
 Compute rows B--F
 Switch old and new arrays
}
```

T_N

T_P

$$T=T_N+T_P$$

Implementation with Overlapping



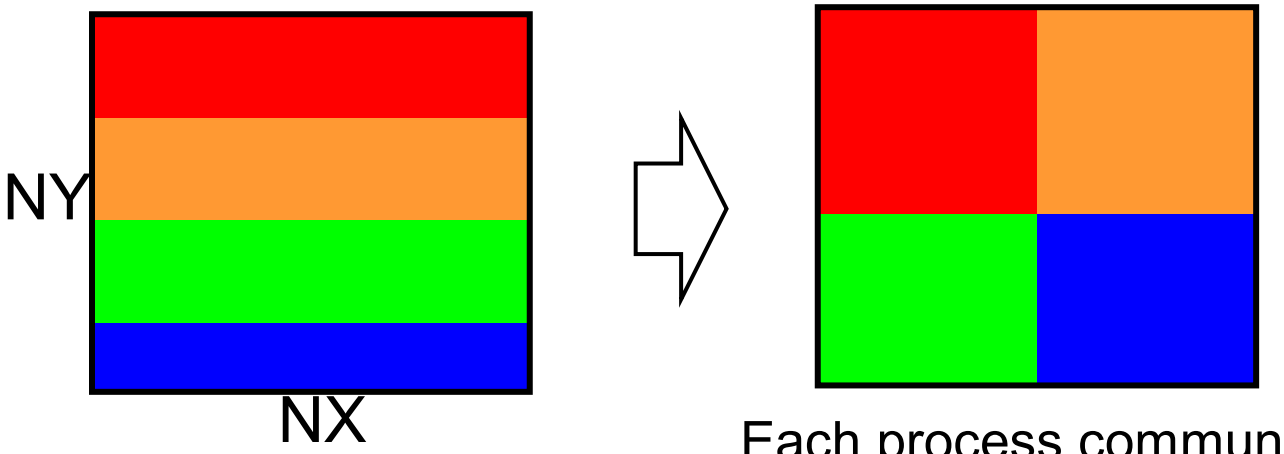
```
for (t = 0; t < nt; t++) {
  Start Send B to rank-1, Start Send F to rank+1
   (MPI_Isend)
  Start Recv A from rank-1, Start Recv G from
   rank-1 (MPI_Irecv)
  Compute rows C--E
  Waits for finishing all communications
   (MPI_Wait)
  Compute rows B, F
  Switch old and new arrays
}
```

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)

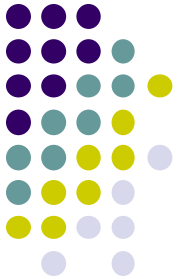
per process

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

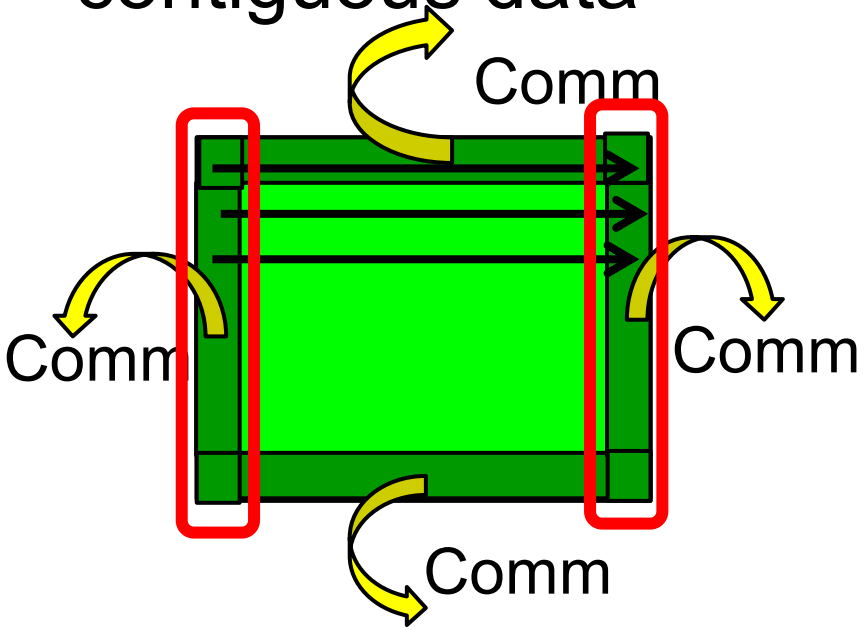
per process

→ Comm is reduced

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



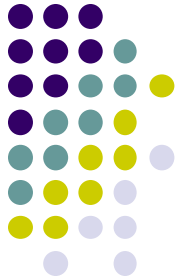
- MD division may need communication of non-contiguous data



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)

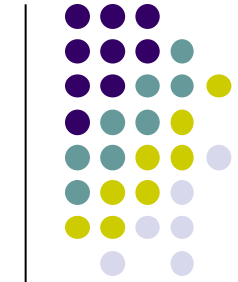


Solution (1):

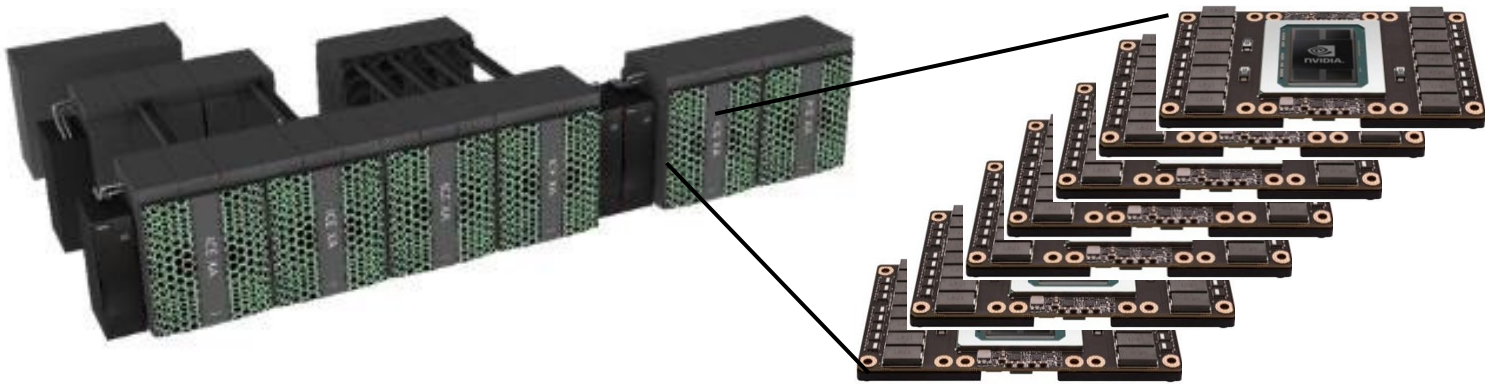
- Before sending, copy non-contiguous data into another contiguous buffer
- After receiving, copy contiguous buffer to non-contiguous area

Solution (2):

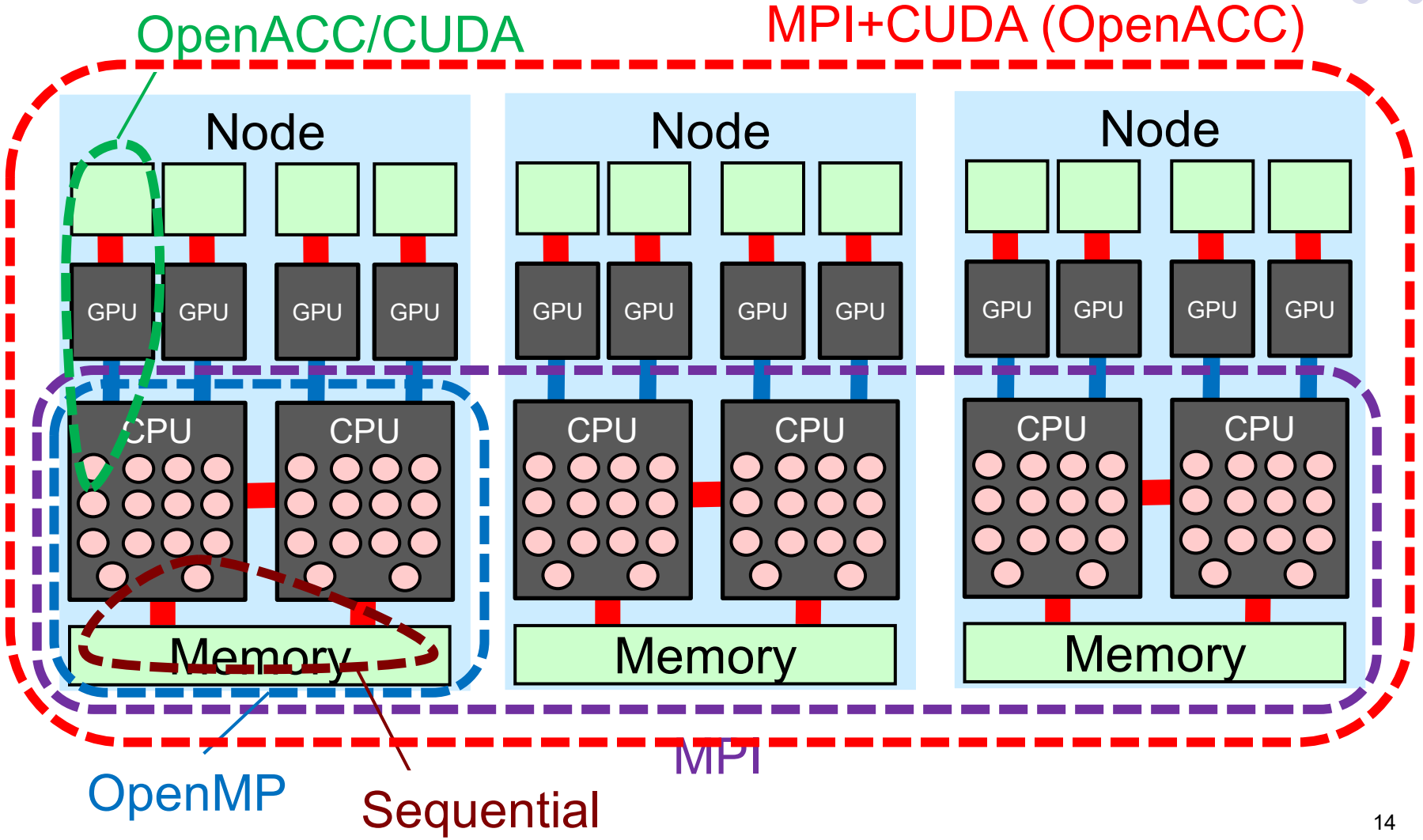
- Use MPI_Datatype
  - Skipped in the class; you may use Google



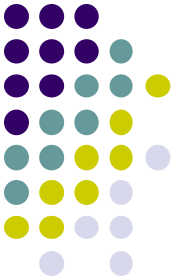
Using Multiple GPUs with MPI+CUDA



Parallel Programming Methods on TSUBAME



Using Multiple GPUs



- GPUs on a single node

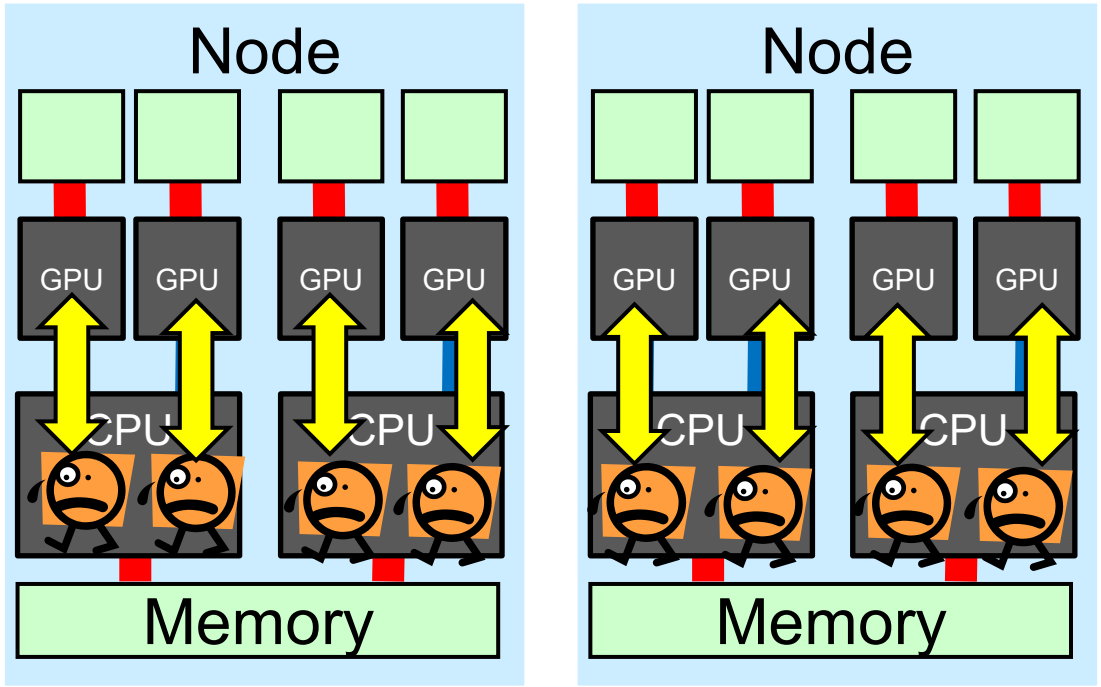
  (Up to 4 GPUs on a TSUBAME3.0 f_node)
  - OpenMP + CUDA
    - 1 thread uses 1 GPU
  - 1 thread switches multiple GPUs
    - cudaSetDevice() is called many times
- GPUs on multiple nodes
  - MPI + CUDA
    - 1 process uses 1 GPU (mm-mpi-cuda sample)

Using Multiple GPUs with MPI

- Basic idea:

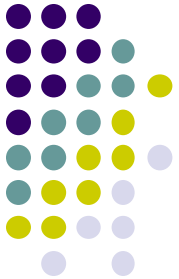
  (1) Start processes on multiple nodes by MPI

  (2) Each process uses its local GPU by CUDA



Sample: /gs/hs1/tga-ppcomp/23/mm-mpi-cuda/

Compiling mm-mpi-cuda Sample



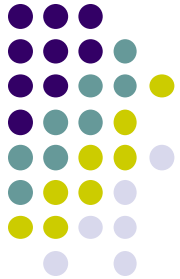
```
module load cuda openmpi    [Do once after login]
cd ~/t3workspace            [In web-only route]
cp -r /gs/hs1/tga-ppcomp/23/mm-mpi-cuda  .
cd mm-mpi-cuda
make
[An executable file “mm” is created]
```

In this Makefile,

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

On other systems than TSUBAME3, we may need try&error

Executing mm-mpi-cuda



- Interactive use is only for one node
- → To use multiple nodes, job submission is required

In standard route, Use qsub on the login node

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

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

job2f.sh

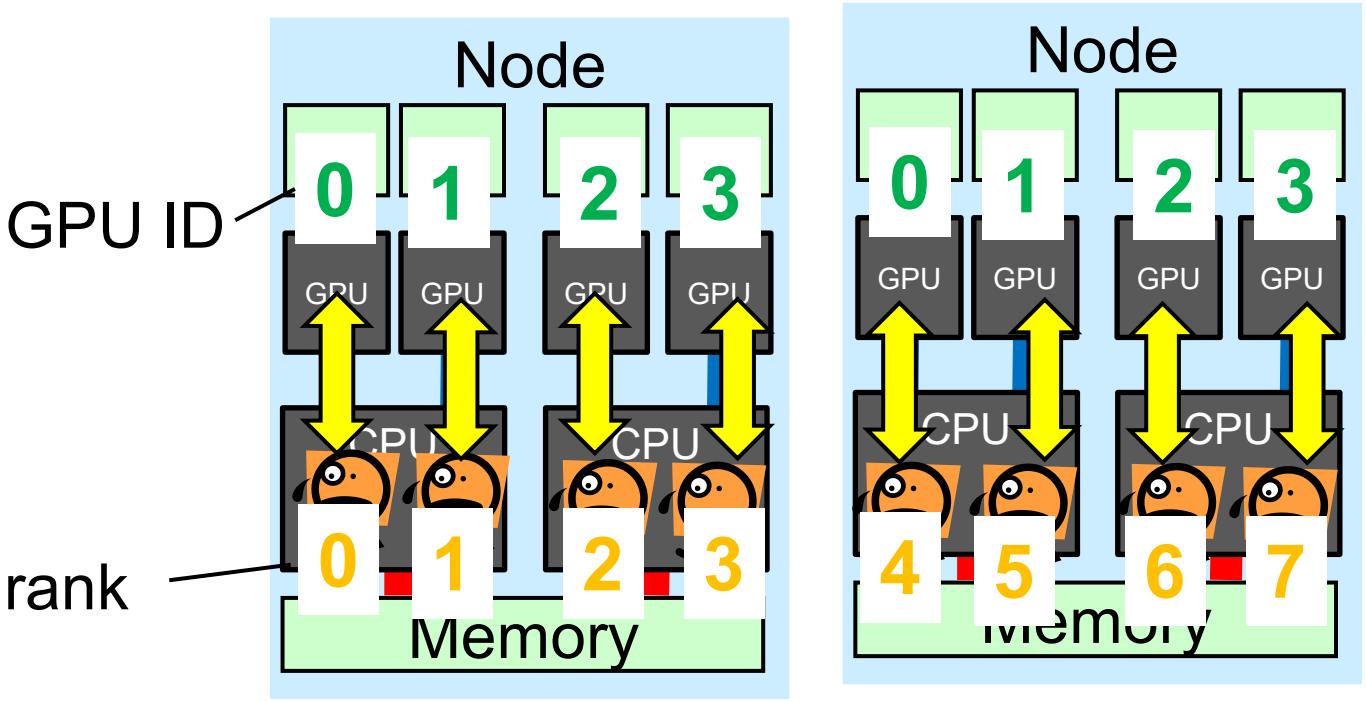
```
#!/bin/sh
#$ -cwd
#$ -l f_node=2
#$ -l h_rt=0:10:00

. /etc/profile.d/modules.sh
module load cuda openmpi

mpiexec -n 8 -npernode 4 -x LD_LIBRARY_PATH ./mm 2048 2048 2048
```

8 processes, 4 processes per node

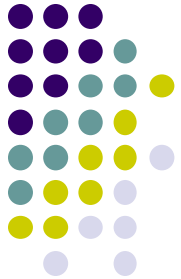
To avoid scheduler's problem

Using Multiple GPUs per node (1)

- Case of “f_node=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)



- f_node or h_node 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 q_node
➔ 2 on h_node
➔ 4 on f_node

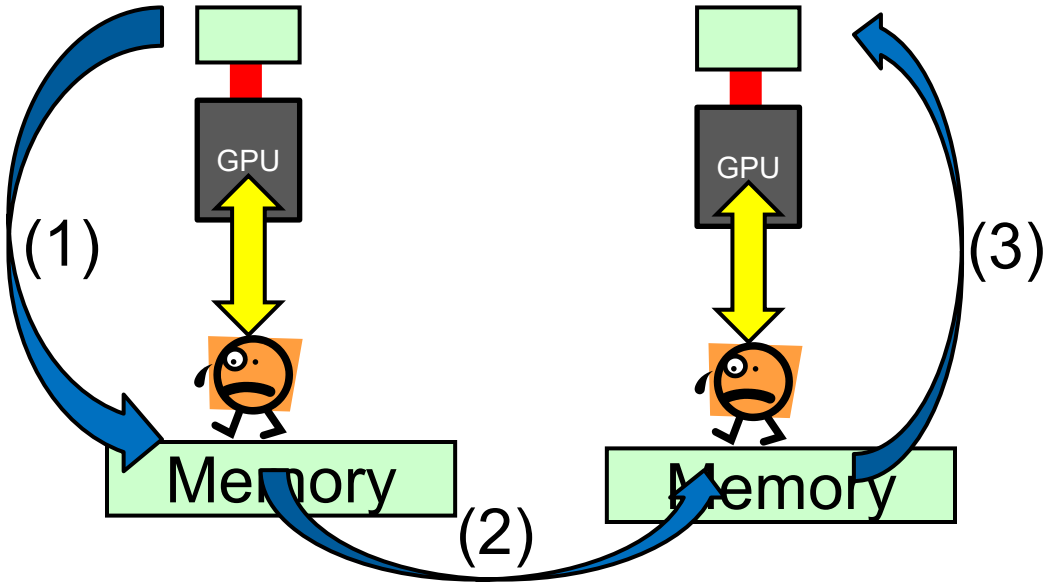
Data Transfer (1)

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

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

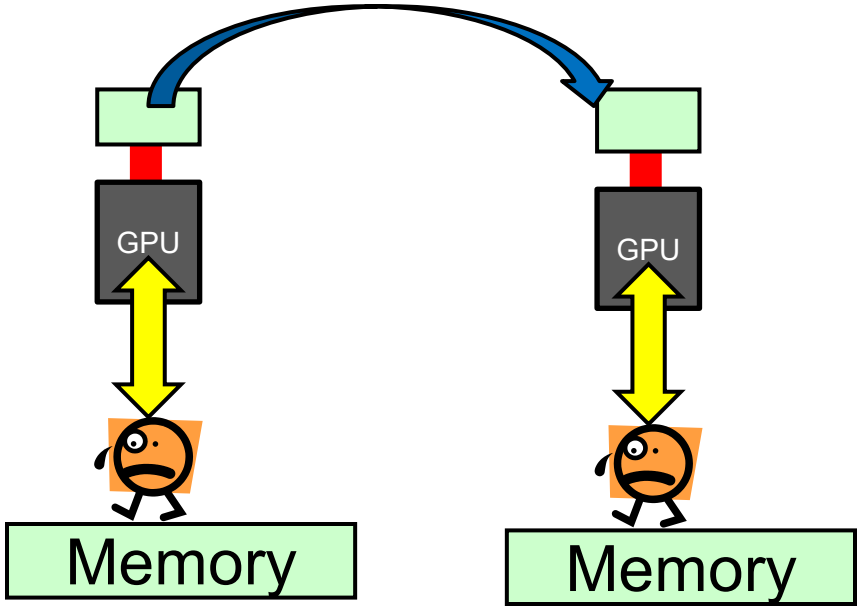
  (2) Transfer between processes (MPI_Send/MPI_Recv)

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

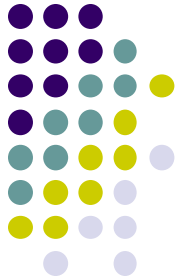


Data Transfer (2)

- 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

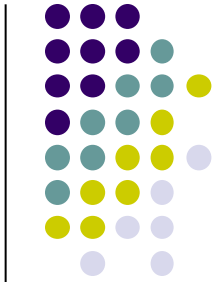


- Part 1: Shared memory parallel programming with OpenMP
- Part 2: GPU programming with OpenACC and CUDA
- Part 3: Distributed memory parallel programming with MPI

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

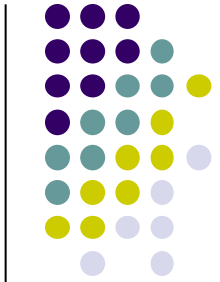
Due date: June 12 (Monday)

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

[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