第228回　お試しアカウント付き
並列プログラミング講習会

OpenACCとMPIによる
マルチGPUプログラミング入門

東京大学 情報基盤センター

担当：星野哲也

hoshino@cc.u-tokyo.ac.jp

（内容に関するご質問はこちらまで）

講習会スケジュール

- 開催日時
  - ✓ 7月21日（月） 13:00 – 17:00
- プログラム
  - ✓ 13:00 – 13:30 スパコンの使い方など
  - ✓ 13:30 – 14:30 OpenACC + MPI混合プログラミング（座学）
  - ✓ 14:45 – 17:00 OpenACC + MPI演習

講習会について

- 本講習会は
  - ✓ OpenACCとMPIを使った複数GPUプログラミング入門

  を中心に扱います。

- その他の講習会

  https://www.cc.u-tokyo.ac.jp/events/lectures/

- スパコンイベント情報メール配信サービス

  https://regist.cc.u-tokyo.ac.jp/announce/
  - ✓ 講習会や研究会の案内、トライアルユースの実施のお知らせなどを配信しています。

Youtubeにて過去の講習会を配信中！

https://www.youtube.com/channel/UC2CHaGp1AO-vqRlV7wmU0-w/videos?view=0&sort=p&flow=grid

講習会の進め方

- Zoomを利用したオンライン講習会です
  - ✓ この講義は録画されています
  - ✓ 質問があるとき以外はミュートでお願いします
  - ✓ ビデオもオフを推奨します
- slackを使って質問に対応します
  - ✓ slackはリンクを知っている人は誰でも使える設定になっています
  - ✓ slackのリンクをzoomのチャットに貼るので、未登録の場合は今のうちに登録お願いします
    - ✓ slackの登録メールの配送に小一時間かかることがあります
  - ✓ スクリーンショットなどで画像を共有することで、質問対応します
    - ✓ Windows：Alt + PrtScn で作業中ウィンドウのスクショがクリップボードにコピーされます。slackのチャット部分で貼り付け(Ctrl + V)することで画像をアップロードできます
    - ✓ Mac : command + shift + control + 4 の同時押し、その後撮りたいウィンドウ上でspaceを押すことで、スクリーンショットがクリップボードにコピーされます。slackのチャット部分で貼り付け(command + V)することで画像をアップロードできます

Zoom: 「手を挙げる」方法

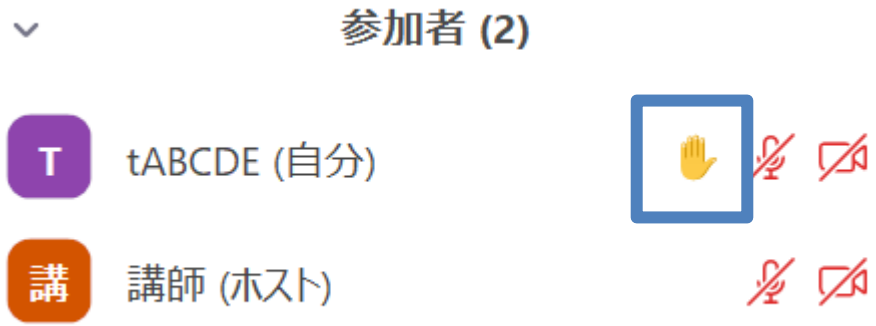
1. Zoomメニュー中の「リアクション」をクリック

2. ポップアップで表示された「手を挙げる」をクリック

Zoom: 手が挙がっていることの確認方法

1. Zoomメニュー中の「参加者」をクリックして，参加者一覧を表示

2. 表示された参加者一覧の，自分のところを見ると手が挙がっている

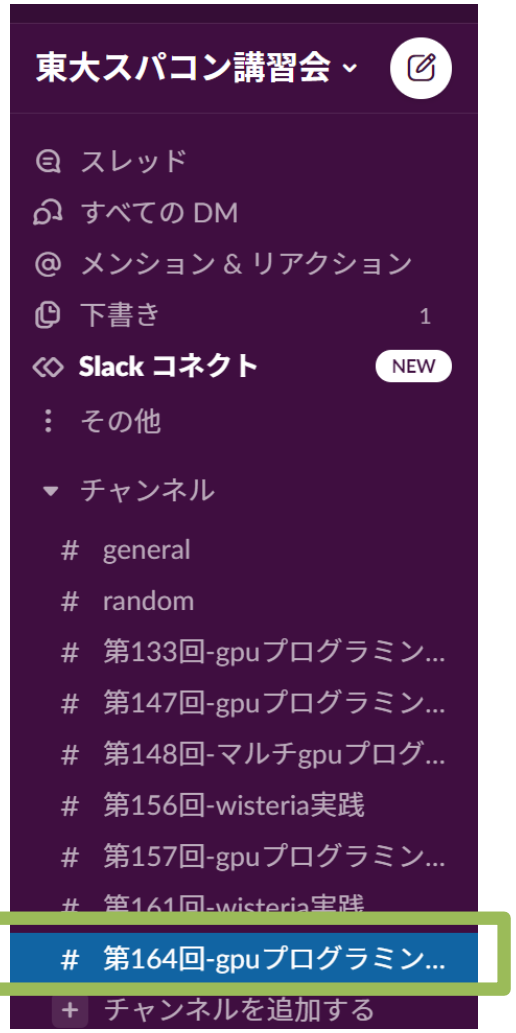


Zoom: 「手を降ろす」方法

1. Zoomメニュー中の「リアクション」をクリック

2. ポップアップで表示された「手を降ろす」をクリック

Slack: 質疑応答チャンネルへの移動



- 左側のメニューバーのチャンネル一覧内に「第164回-gpuプログラミング入門」があるので，クリック
- 表示されていない場合
  1. 「チャンネルを追加する」をクリック
  2. 「チャンネル一覧を確認する」をクリック
  3. 「第228回-マルチgpuプログラミング入門」があるので，「参加する」をクリック

Slack: メッセージの入力方法

- 最下部に入力欄があるので，質問内容を記載して Ctrl+Enter
  - 入力後に右下の「メッセージを送信する」をクリックしても同じ
    （メッセージ入力前には，「メッセージを送信する」は押せない）

- コードを入力する際には，「コードブロック」がおすすめ
  - 枠が生成されるので，この中にコピペするのが簡単かつ見やすい
  - ```（JIS配列ならばShift+@を3連打）しても枠が生成される

サンプルコードのダウンロード

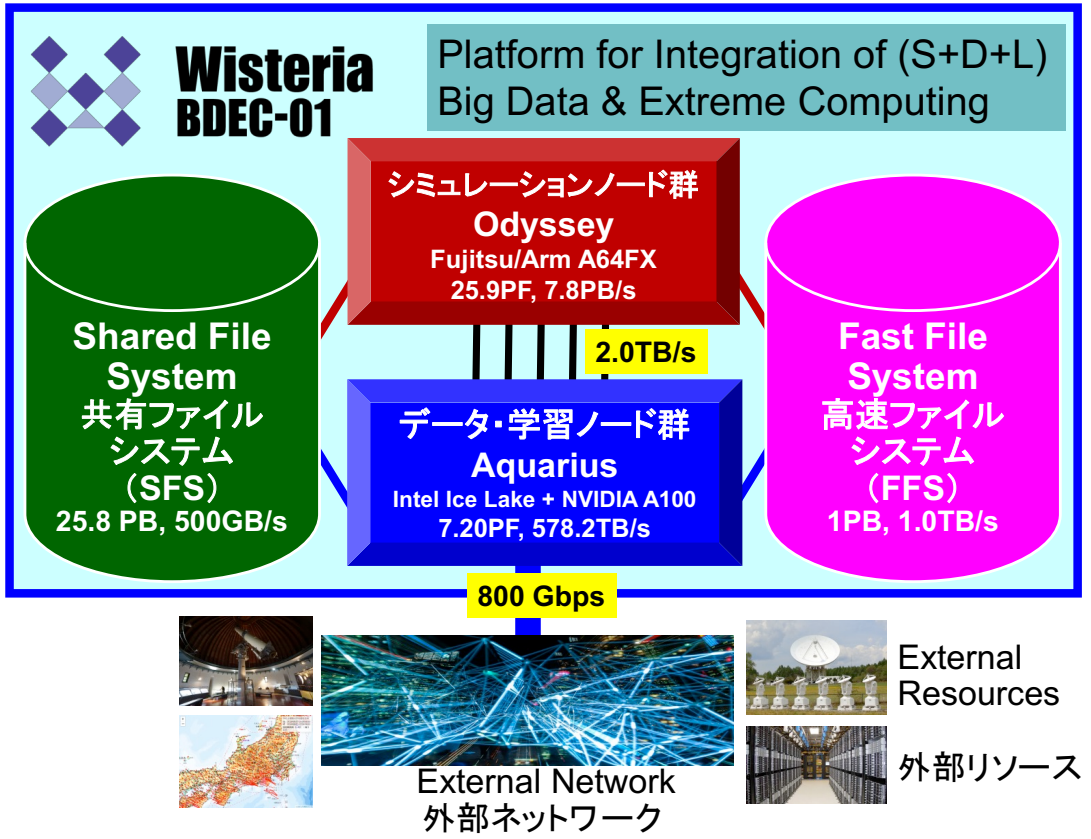
- 講習会で使うサンプルコードはGitHubにあります
  - https://github.com/hoshino-UTokyo/lecture_openacc_mpi.git

- ダウンロード手順
  1. $ ssh tXXXXX@wisteria.cc.u-tokyo.ac.jp
     - wisteriaへのssh。tXXXXXはアカウント名
  2. $ cd /work/gt00/tXXXXX
  3. $ git clone https://github.com/hoshino-UTokyo/lecture_openacc_mpi.git
  4. $ cd lecture_openacc_mpi/
  5. $ cd C or F
     - CまたはFortran好きな方を利用してください。

Wisteria/BDEC-01

BDEC:「計算・データ・学習（S+D+L）」融合のためのプラットフォーム（Big Data & Extreme Computing）

- 2021年5月14日運用開始
  - 東京大学柏Ⅱキャンパス
- 33.1 PF, 8.38 PB/sec.， **富士通製**
  - ~4.5 MVA（空調込み）, ~360m^2
- Hierarchical, Hybrid, Heterogeneous (h3)
- **2種類のノード群**
  - **シミュレーションノード群 (S, SIM)：Odyssey**
    - **従来のスパコン**
    - **Fujitsu PRIMEHPC FX1000 (A64FX), 25.9 PF**
      - **7,680ノード（368,640 コア），20ラック，Tofu-D**
  - **データ・学習ノード群（D/L, DL）：Aquarius**
    - **データ解析，機械学習**
    - **Intel Xeon Ice Lake + NVIDIA A100, 7.2 PF**
      - **45ノード（Ice Lake：90基，A100：360基），IB-HDR**
    - **一部は外部リソース（ストレージ，サーバー，センサーネットワーク他）に直接接続**
  - ファイルシステム：共有（大容量）＋高速



Wisteria
BDEC-01

Wisteria 利用上の注意（１）

- ディレクトリについて（home と work)
  - ✓ ログイン時のディレクトリ（/home/gt00/txxxxx）にはログイン時に必要なファイルのみを置く
  - ✓ プログラム作成や実行などに必要なファイルは /work 以下のディレクトリ（/work/gt00/txxxxx）に置く
  - ✓ /home は計算ノードからは参照できない

Wisteria 利用上の注意（2）

- コンパイルおよび実行のための環境準備
  - ✓ コンパイルおよび実行のための環境を準備するために module コマンドを使用する。これによって様々な環境を簡単に切り替えて使用できる。

$ module load <*module_name*>

モジュール名 **<*module_name*>** のモジュールをロードして環境を準備。環境変数PATHなどが設定される。

$ module avail

使用可能なモジュール一覧を表示する。

$ module list

使用中のモジュールを表示する。

Wisteriaでのプログラムの実行

- ジョブスクリプト(〇〇.sh)を作成し、ジョブとして投入、実行する。

```
$ pjsub ./〇〇.sh
```

- 投入されたジョブを確認する。（qstatではないので注意）

```
$ pjstat
```

- 実行が終了すると、以下のファイルが生成される。

```
〇〇.sh.??????.out
〇〇.sh.??????.err　（?????? はジョブID）
```

- 上記の標準出力ファイルの中身を確認する。

```
$ cat 〇〇.sh.??????.out
```

- 必要に応じて、上記のエラー出力ファイルの中身を確認する。

```
$ cat 〇〇.sh.??????.out
```

コンパイラの種類と実行(Aquarius)

- ログインノードとAquarius計算ノードとでは、CPUの命令セットが（ほぼ）同じ
  - ログインノード：命令セットアーキテクチャ Intel CascadeLake＋AVX512, **x86_64**
  - Aquarius計算ノード： 命令セットアーキテクチャ Intel IceLake＋AVX512, **x86_64**
- 様々なコンパイラが利用可能: GPU向けには gcc+CUDAか NVIDIAを推奨

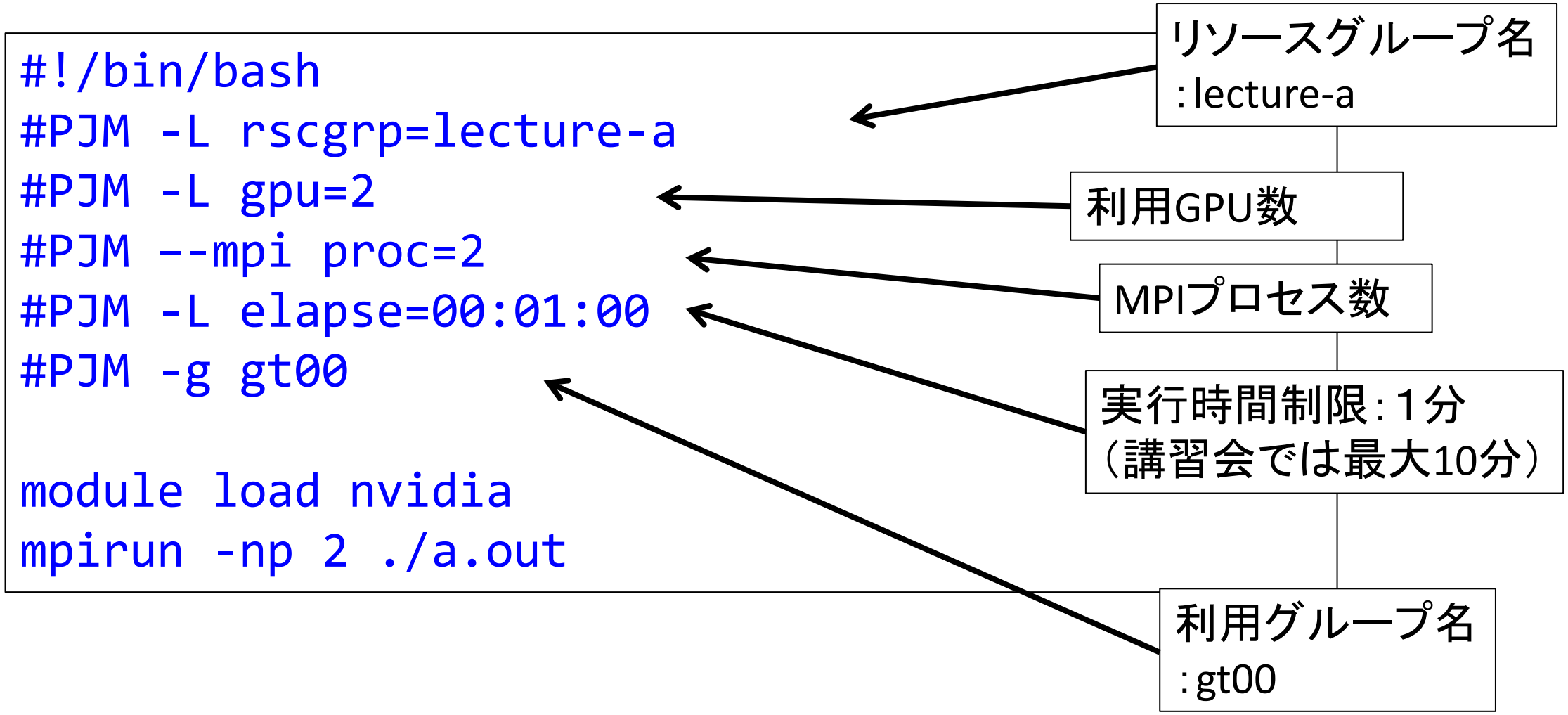
```
$ module load gcc cuda ompi-cuda   または
$ module load nvidia cuda ompi-cuda    #OpenACCならこちら
```

言語	GNUコンパイラ	Intelコンパイラ	NVIDIA コンパイラ (旧PGI)	CUDAコンパイラ
C	gcc	icc	nvc (pgcc)	nvcc
C++	g++	icpc	nvc++(pgc++)	
Fortran	gfortran	ifort	nvfortran (pgfortran)	
OpenACC			○	

JOBスクリプトサンプルの説明（Aquarius, MPIあり）

```
#!/bin/bash
#PJM -L rscgrp=lecture-a
#PJM -L gpu=2
#PJM --mpi proc=2
#PJM -L elapse=00:01:00
#PJM -g gt00

module load nvidia
mpirun -np 2 ./a.out
```



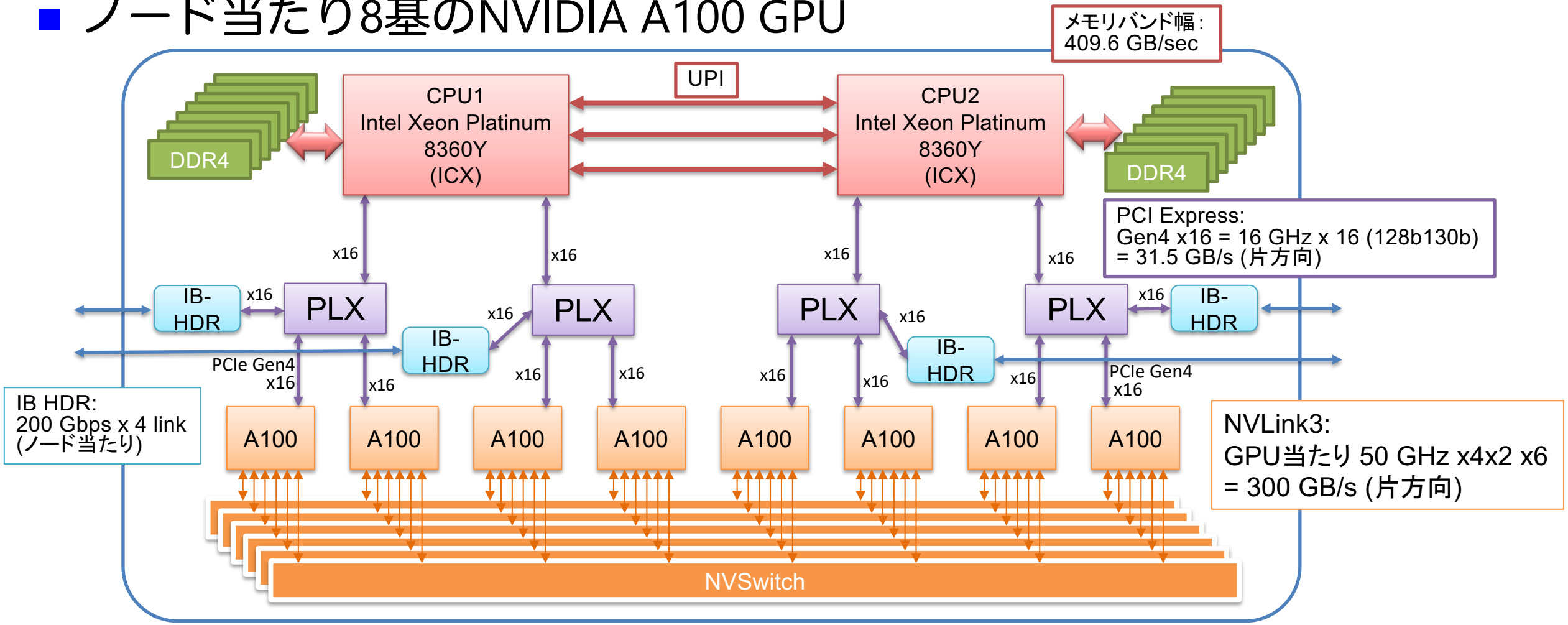
OPENACCとMPIによる
マルチGPUプログラミング

マルチGPUコンピューティング

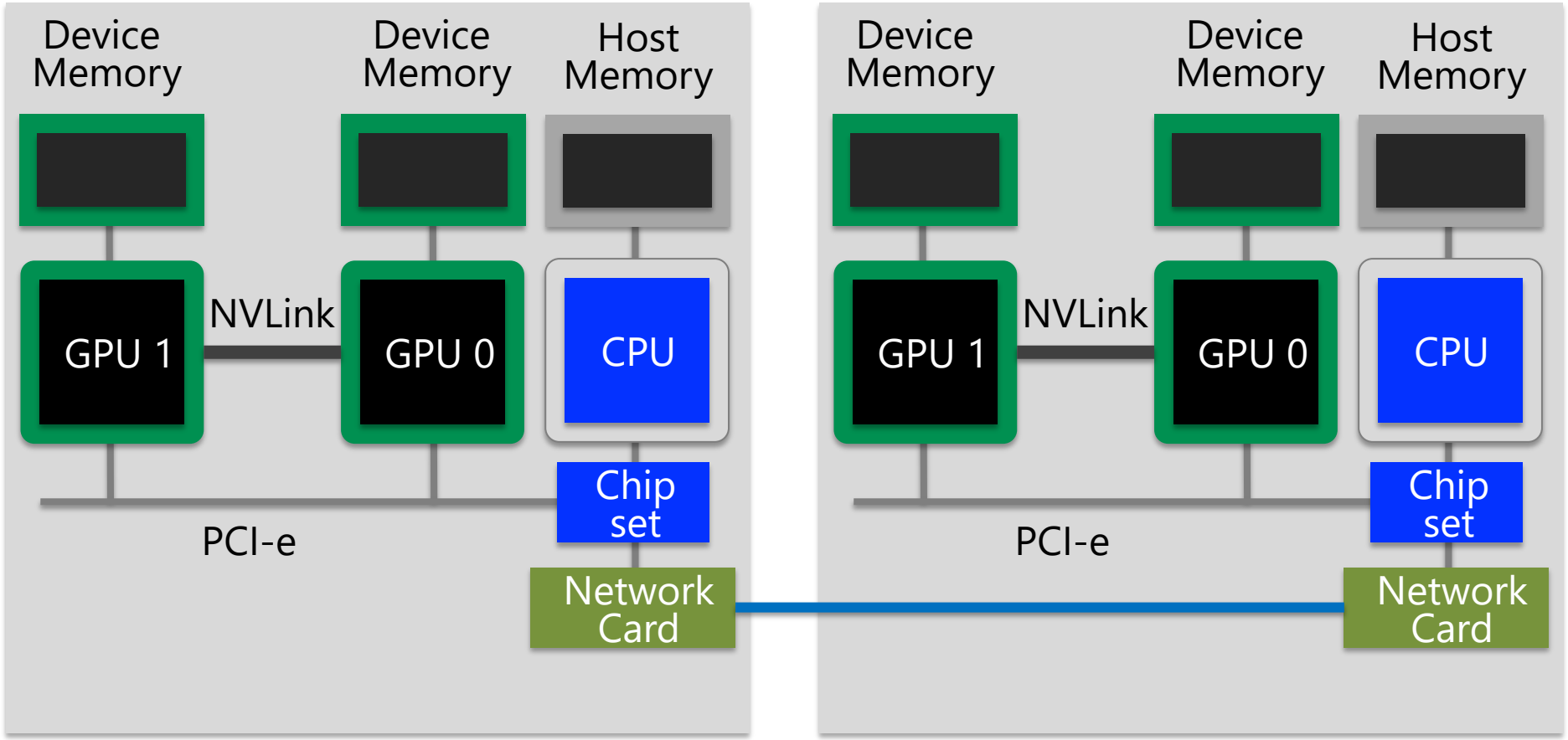
- 複数GPU計算する目的
  - ✓ 1個のGPUに搭載されたメモリよりも大きい問題を解きたい。
  - ✓ 1個のGPUで計算するよりも高速に計算したい。
- 複数GPU計算の方法
  - ✓ 推奨：複数GPUをMPIで並列化
    - ✓ 最大の難点は環境構築の難しさ。
  - ✓ 複数GPUをOpenMPで並列化
    - ✓ できなくはない。OpenMPを使う場合、OpenACCと同じループに指示文の適用はできないため、omp_get_thread_num()を用いて並列化することになる。
  - ✓ 複数GPUを acc_set_device_num (OpenACC) や cudaSetDevice (CUDA) で切り替えながら計算
- 本講習会では、複数ノードに搭載された複数のGPUを活用できるMPI による並列化を行う。

Aquariusの計算ノード

- Intel Xeon Platinum 8360Y (36c 2.4GHz) x 2ソケット, 512GBメモリ
- ノード当たり8基のNVIDIA A100 GPU



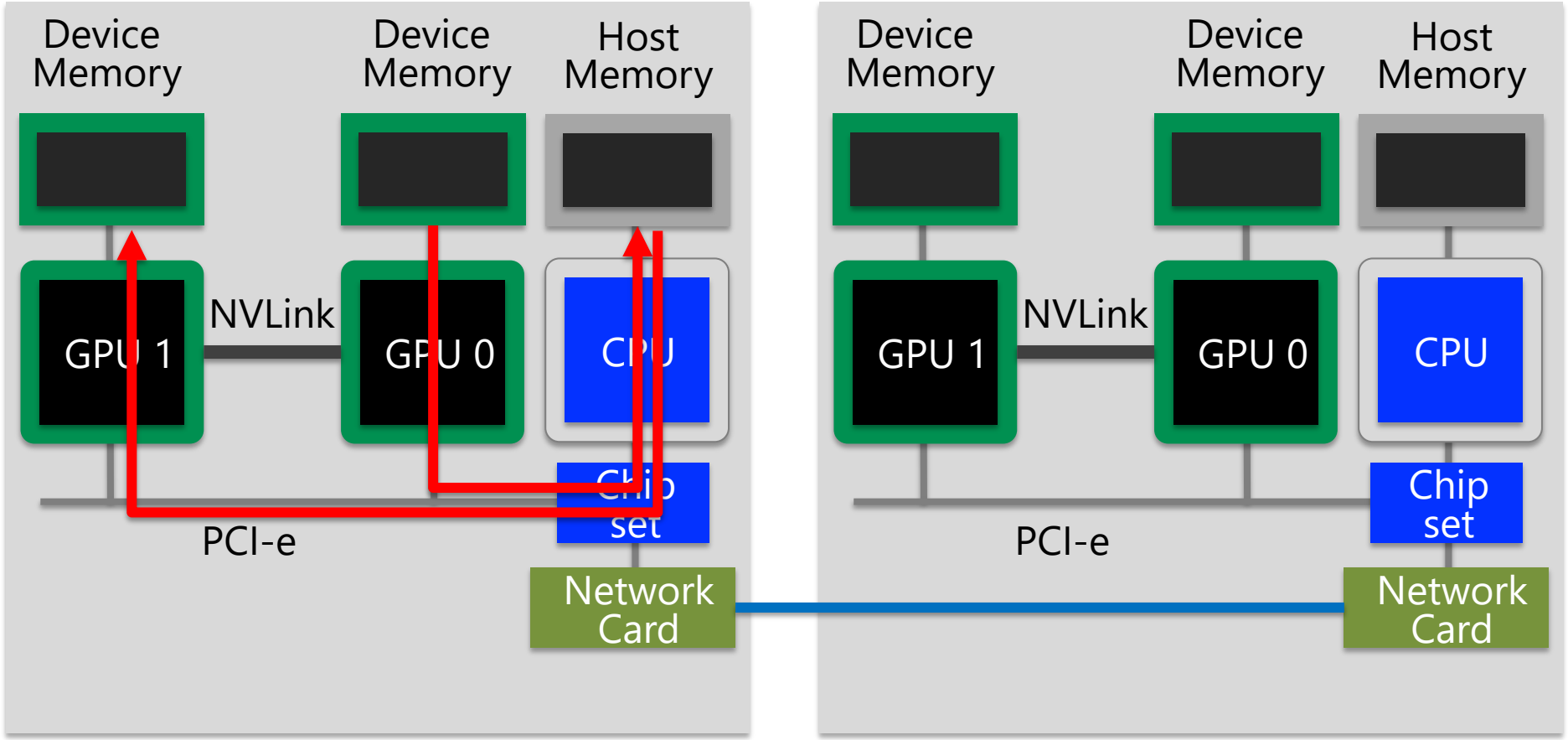
GPUx2 の計算ノード模式図



- GPU-GPU通信にはいくつかの種類がある

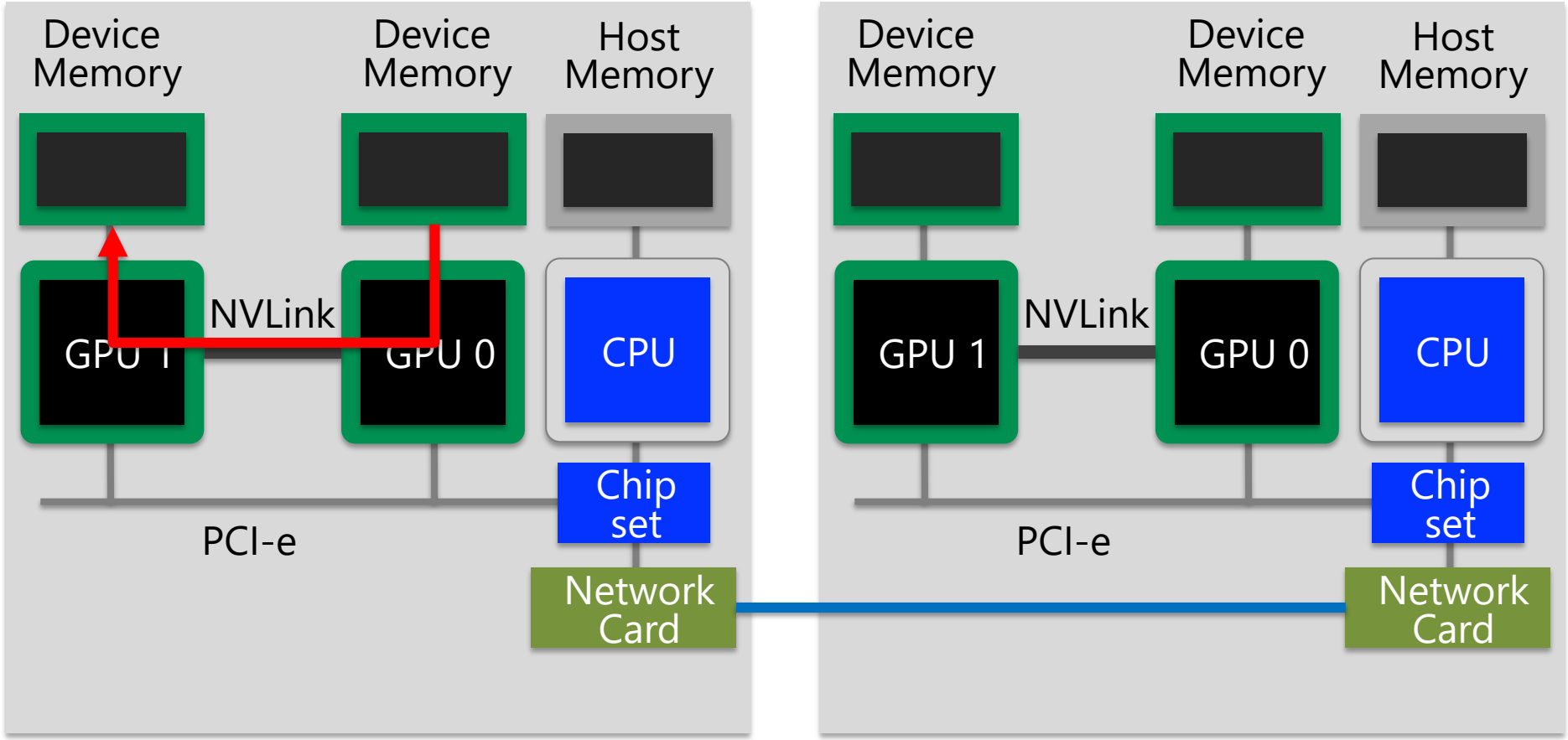
ノード内GPU間通信（１）

- ホストメモリ経由のノード内GPU間通信
  - ✓ デバイスメモリとホストメモリの間の PCI-eを通る。



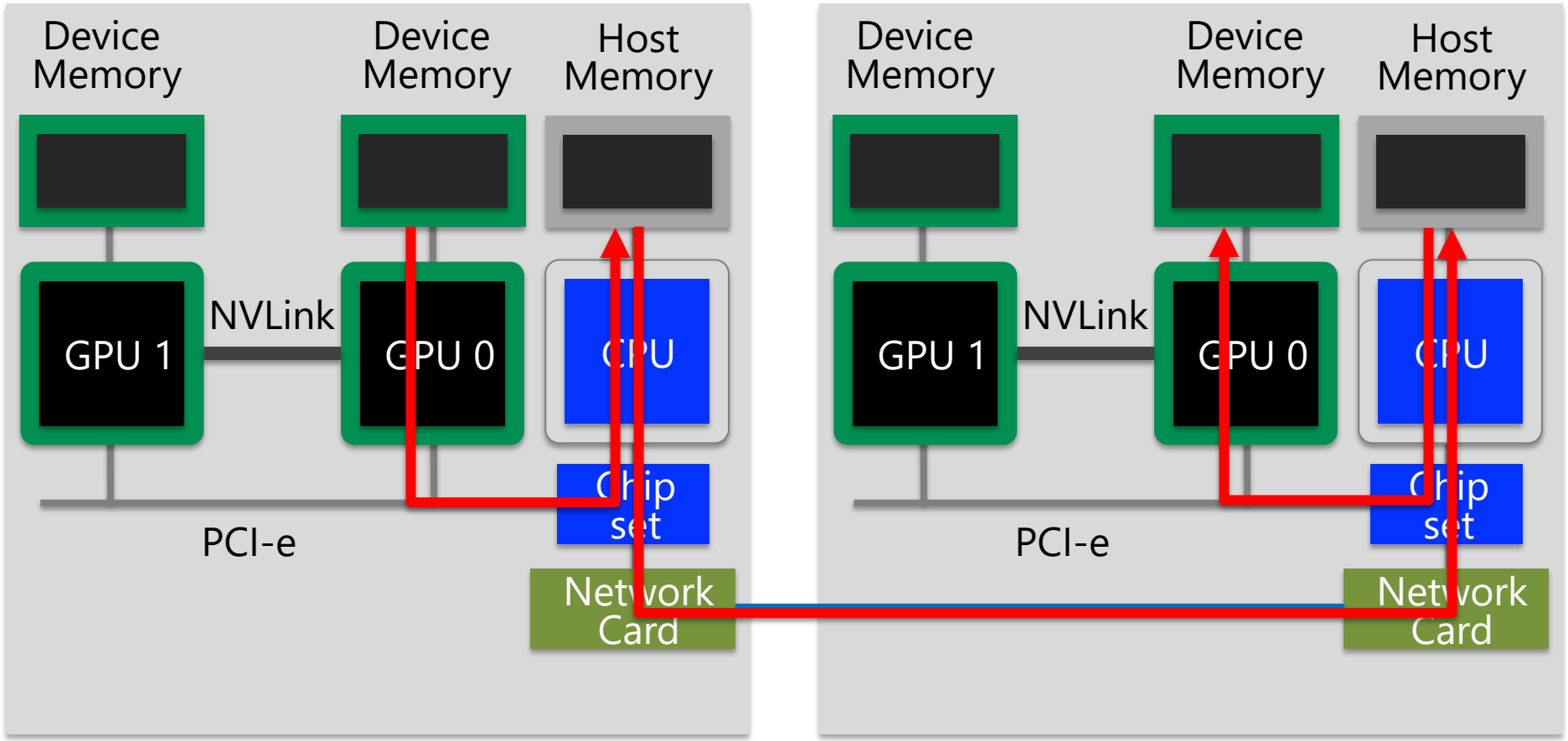
ノード内GPU間通信（2）

- NVLink経由のGPU間通信
  - ✓ CUDA Inter Process Communication (IPC) を利用し、同じノード内にあるプロセスではホストを経由せずGPU間で直接通信できる。最近のサーバではGPU間をつなぐNVLink経由で高速通信する。



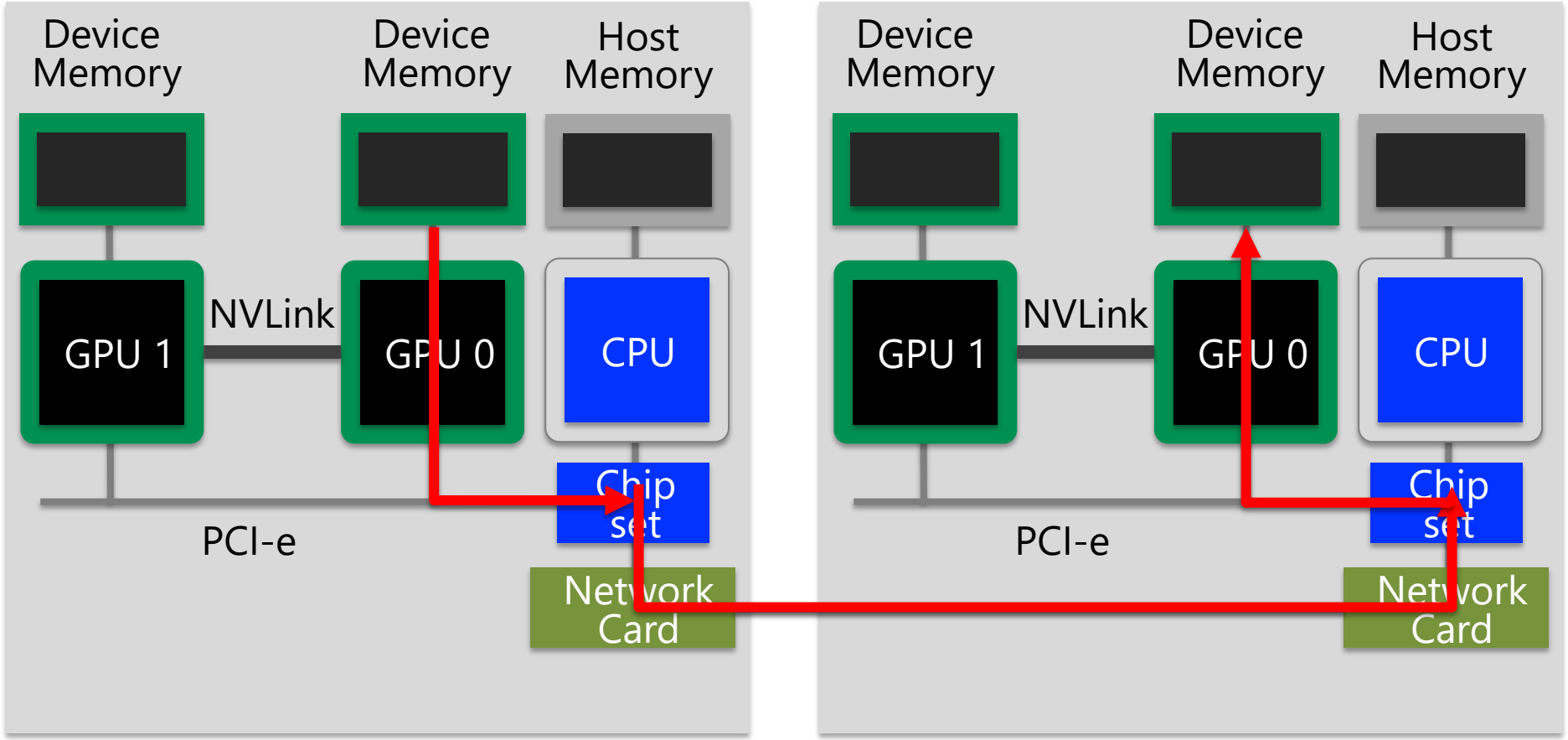
ノード間のGPU間通信（１）

- ホストメモリ経由のノード間GPU間通信
  - ✓ あるノードのGPU0から別ノードのGPU0へ転送する際に、それぞれのホストメモリを経由して転送する。



ノード間のGPU間通信（2）

- GPUDirectRDMAによるノード間GPU間通信
  - ✓ ホストメモリを経由することなく、GPUとInfiniBand (Network Card) 間で直接データ転送 (RDMA) をすることにより異なるノードのGPU間で高速通信する。



CUDA-aware MPI

- 従来のMPI
  - ✓ MPIの受信領域や送信領域にホストメモリ上のアドレスのみ指定可能。
  - ✓ デバイスメモリのデータを転送する際には、一度ホストメモリへコピーが必要。
- CUDA-aware MPI
  - ✓ CUDA（NVIDA GPU向けの開発環境）とMPIによる複数GPU計算では、しばしばデバイスメモリの内容を他ノードのGPUへMPIで転送する。CUDA-aware MPIでは受信領域や送信領域にデバイスメモリ上のアドレスも指定可能。MPIライブラリ内部ではGPUDirectなどが利用され、MPIライブラリを利用するだけでGPU 間の高速通信技術を利用することができる。
  - ✓ OpenACC からも利用できる。
    - ✓ acc host_data 指示文で、デバイスメモリのアドレスを渡せば良い
  - ✓ Unified Memory 使用時には利用できない
  - ✓ Wisteria では、 nvidia, cuda module の2つをloadした後に出てくるompi-cuda module が CUDA-aware で GPUDirectRDMA(GDR)に対応したOpenMPIです。

OpenACCとMPIコードのコンパイル

- OpenACCとMPIコードのコンパイル
  - ✓ ここではnvidiaコンパイラとOpenMPIを利用します。

```
$ module load nvidia/22.7 cuda/11.4 ompi-cuda/4.1.4-11.4
$ mpicc -O3 -acc -Minfo=accel -gpu=cc80 -c main.c
```

←どうやらこのバージョンが推奨

-acc: OpenACCコードであることを指示

-Minfo=accel:

OpenACC指示文からGPUコードが生成できたかどうか等のメッセージを出力する。このメッセージがOpenACC化では大きなヒントになる。

-gpu=cc80:

compute capability 8.0 (cc80) のコードを生成する。

- Makefileでコンパイル

講習会のサンプルコードには Makefile がついているので、コンパイルするためには、単純に下記を実行すれば良い。

```
$ module load nvidia/22.7 cuda/11.4 ompi-cuda/4.1.4-11.4
$ make
```

簡単なOpenACCとMPIコード（1）

- サンプルコード: openacc_mpi_basic/
  - ✓ OpenACCとMPIを利用したコード
  - ✓ 計算内容は簡単な四則演算と2プロセス間の通信

```
// main.c 内

    for (unsigned int i=0; i<n; i++) {
        a[i] = 3.0 * rank * ny;
        b[i] = 0.0;
    }

    const int dst_rank = (rank + 1) % nprocs;
    const int tag      = 0;
    if (rank == 0) {
        MPI_Status status;
        MPI_Recv(b, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD, &status);
    } else {
        MPI_Send(a, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD);
    }

    double sum = 0.0;
    for (unsigned int i=0; i<n; i++) {
        sum += b[i];
    }
```

次のスライドにも説明あります

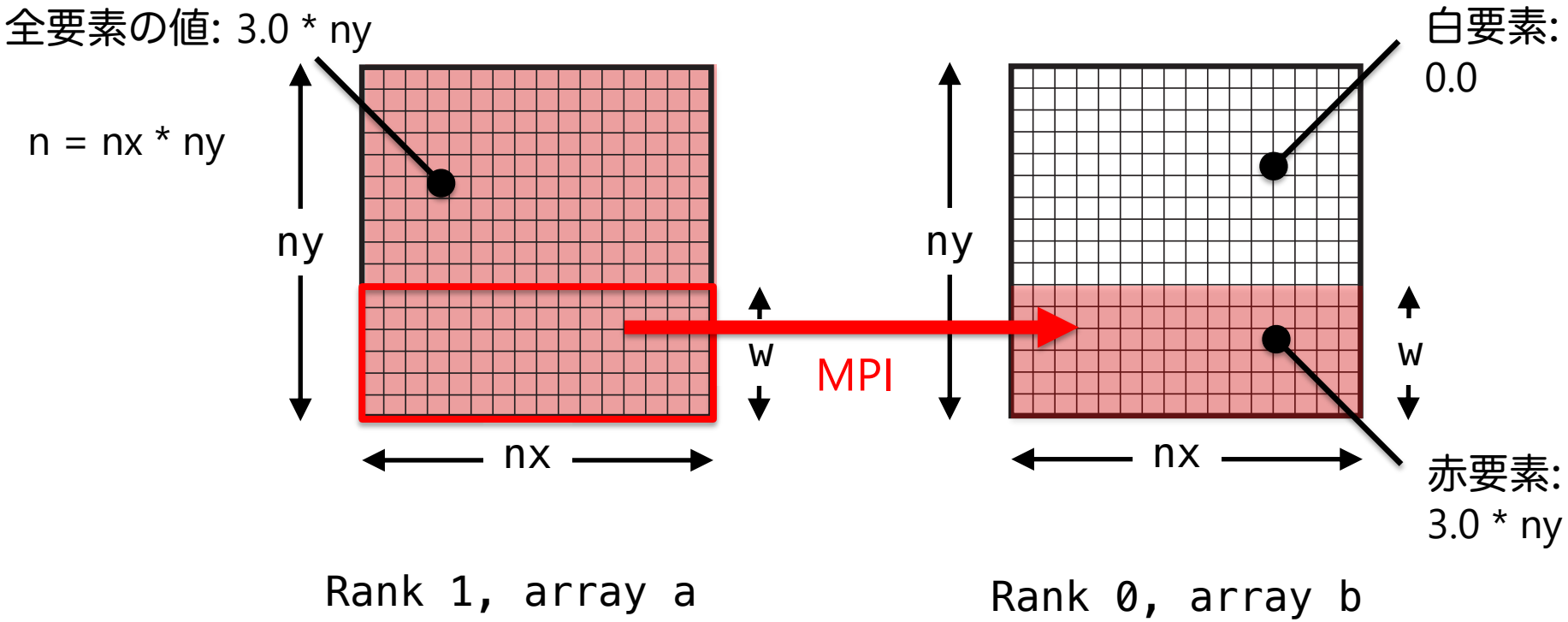
rank = 1 では a = 3.0 * ny

rank1の a から rank0 の b へ
w * nx = 10nx が転送される。

sum/n = (3.0*ny) * (w*nx) / (nx*ny)
= 3.0*w
= 30.0

簡単なOpenACCとMPIコード（2）

- 計算内容
  - ✓ rank = 1 では、a = 3.0 * ny で初期化
  - ✓ rank = 1 の a から rank = 0 の b へ w * nx = 10nx 個の要素が転送される。
  - ✓ sum/n = (3.0∗ny) ∗ (w∗nx) / (nx∗ny) = 3.0∗w = 30.0 となる



簡単なOpenACCとMPIコード（３）

- サンプルコード: openacc_mpi_basic/
  - ✓ OpenACCとMPIを利用したコード
  - ✓ 計算内容は簡単な四則演算と2プロセス間の通信

openacc_mpi_basic/01_original	MPI並列化されたCPUコード。
openacc_mpi_basic/02_kernels	MPI+OpenACCコード。上にkernels/loop指示文を追加。単一ノード内2GPU使用。
openacc_mpi_basic/03_update	MPI+OpenACCコード。上にupdate指示文を追加。単一ノード内2GPU使用。
openacc_mpi_basic/04_cuda_aware	MPI+OpenACCコード。MPI通信にデバイスポインタを渡す。単一ノード内2GPU使用。
openacc_mpi_basic/05_no_gdr	（参考）MPI+OpenACCコード。上を2ノードにある2GPU使用。
openacc_mpi_basic/06_gdr	（参考）MPI+OpenACCコード。上にGPUDirectRDMAを適用。2ノードにある2GPU使用。

簡単なOpenACCとMPIコード: CPUコード (1)

- CPUコードのコンパイルと実行
  - ✓ 配列の平均値と実行時間が出力されています。

```
$ cd openacc_mpi_basic/01_original
$ make
$ pjsub ./run.sh
$ cat run.sh.??????.out
mean = 30.00
Time =    0.034 [sec]
```

? の数字はジョブごとに変わります。

答えは常に30.0

openacc_mpi_basic/01_original

簡単なOpenACCとMPIコード: CPUコード (2)

- 1ノード内の2GPUを利用
  - ✓ ジョブスクリプトを確認します。

```
$ cat run.sh
#!/bin/bash
#PJM -L rscgrp=lecture-a
#PJM -L gpu=2
#PJM --mpi proc=2
#PJM -L elapse=00:10:00
#PJM -g gt00

module purge
module load nvidia cuda ompi-cuda
export UCX_MEMTYPE_CACHE=n

mpirun -np 2 ./run
```

gpu=2:

使用するGPU数を指定。Lecture-aキューは、他のユーザと共有の環境になっており、最大4GPUまで利用できる。講習会では使えないが、ノード占有のキューではnode=2などと書く。

--mpi proc=2:

MPIのプロセス数を書く

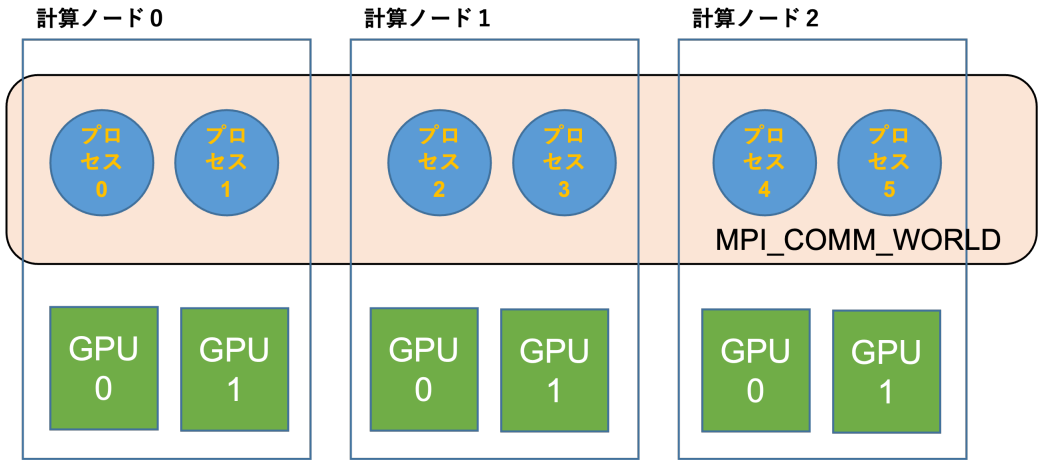
mpirun -np 2

-np に全並列数（全MPIプロセス数）を指定。 ./run が実行したいプログラム。

MPIランクとGPU割り当て

- OpenACC関数を呼ぶためヘッダーを追加

```
C: #include <openacc.h>
Fortran: use openacc
```



- ノード上のGPU数の取得(acc_get_num_devices)

```
const int ngpus = acc_get_num_devices(acc_device_nvidia);
```

acc_device_nvidiaを指定することで、NVIDIA GPUを数える。

- あるプロセスに特定のGPUを割り当て(acc_set_device_num)

```
int rank   = 0;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
acc_set_device_num(rank % ngpus, acc_device_nvidia);
```

device_num を rank % ngpus とすることで、ノード内のプロセスは異なるGPUを利用することとなる。**これを呼ばないと、0番を使いに行く。**

openacc_mpi_basic/02_kernels

kernels 指示文追加コード（１）

- 02_kernelsコード
  - ✓ kernels, loop を利用したコード。MPIはホストメモリを参照。

openacc_mpi_basic/02_kernels

```
#pragma acc kernels copyout(a[0:n], b[0:n])
#pragma acc loop independent
    for (unsigned int i=0; i<n; i++) {
        a[i] = 3.0 * rank * ny;
        b[i] = 0.0;
    }

    const int dst_rank = (rank + 1) % nprocs;
    const int tag      = 0;
    if (rank == 0) {
        MPI_Status status;
        MPI_Recv(b, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD, &status);
    } else {
        MPI_Send(a, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD);
    }

    double sum = 0.0;
#pragma acc kernels copyin(b[0:n])
#pragma acc loop reduction(+:sum)
    for (unsigned int i=0; i<n; i++) {
        sum += b[i];
    }
```

allocate

GPU->CPU, deallocate

MPI(CPU)

MPI(CPU)

allocate, CPU->GPU

deallocate

kernels 指示文追加コード（2）

- 通信方法
  - ✓ MPI前後で配列の全領域をデバイスメモリとホストメモリ間でコピー
  - ✓ ホストメモリ経由のノード内GPU間通信

- コンパイルと実行

```
$ make
$ pjsub ./run.sh
$ cat run.sh.??????.out
num of GPUs = 2
mean = 30.00
Time =    0.078 [sec]
```

update 指示文

- data 指示文などで既にデバイスメモリ上に確保されたデータに対して、それ自体または対応するホストメモリを対となるメモリの値で更新する。
  - ✓ memcpy(CPU->GPU), memcpy(GPU->CPU) の機能を有すると思えば良い。
- update 指示文の 主な指示節
  - host
    - ✓ memcpy(GPU->CPU)
  - device
    - ✓ memcpy(CPU->GPU)
- update 指示文の 使い方例

```
...
#pragma acc data copy(a[0:nx * ny])                    allocate, CPU->GPU
{
...
#pragma acc update host(a[0:nx])                       GPU->CPU
    host_func(a, nx)
#pragma acc update device(a[0:nx])                     CPU->GPU
...
}                                                      GPU-> CPU, deallocate
...
```

update指示文追加コード（1）

- 03_update コード

  openacc_mpi_basic/03_update

  ✓ data, update を追加したコード。MPIはホストメモリを参照。

```
double sum = 0.0;
#pragma acc data create(a[0:n], b[0:n])                    allocate
    {
#pragma acc kernels present(a[0:n], b[0:n])
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            a[i] = 3.0 * rank * ny;
            b[i] = 0.0;
        }

        const int dst_rank = (rank + 1) % nprocs;
        const int tag      = 0;
        if (rank == 0) {
            MPI_Status status;                                 MPI(CPU)
            MPI_Recv(b, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD, &status);
#pragma acc update device(b[0:w *nx])                          CPU->GPU
        } else {
#pragma acc update host(a[0:w *nx])                            GPU->CPU
            MPI_Send(a, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD);
        }                                                      MPI(CPU)

#pragma acc kernels present(b[0:n])
#pragma acc loop reduction(+:sum)
        for (unsigned int i=0; i<n; i++) {
            sum += b[i];
        }
    }                                                          deallocate
```

MPI通信前後に挿入

update指示文追加コード（2）

- 通信方法
  - ✓ MPI前後で配列の必要な領域のみを update 指示文でデバイスメモリとホストメモリ間でコピー
  - ✓ 02_kernels と比較し、メモリの確保・解放とデータのコピー量が削減されている。
  - ✓ ホストメモリ経由のノード内GPU間通信

- コンパイルと実行

```
$ make
$ pjsub ./run.sh
$ cat run.sh.??????.out
num of GPUs = 2
mean = 30.00
Time =    0.037 [sec]
```

host_data 指示文

- data 指示文などで既にデバイスメモリ上に確保されたデータに対して、並列領域の外でデバイスメモリ側のアドレスを使うための指示文
- デバイス側のアドレスを使いたい例
  - ✓ GPU用のライブラリの呼び出し
  - ✓ CUDA で書かれた関数を呼ぶ
  - ✓ CUDA-aware MPIによる通信（GPUDirectの利用）

```
...
#pragma acc data copy(a[0:n])
{
...
#pragma acc host_data use_device(a)
{
    cuda_func(a, n)
}
...
}
...
```

allocate, CPU->GPU

host_data 内ではホストコードにも関わらず a はデバイス側のアドレスが使われる。

CPU->GPU, deallocate

CUDA-aware MPIコード（1）

- 04_cuda_aware コード

openacc_mpi_basic/ 04_cuda_aware

  - ✓ host_data を利用したコード。MPIはデバイスメモリを参照。

```
double sum = 0.0;
#pragma acc data create(a[0:n], b[0:n])                    allocate
    {
#pragma acc kernels present(a[0:n], b[0:n])
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            a[i] = 3.0 * rank * ny;
            b[i] = 0.0;
        }

        const int dst_rank = (rank + 1) % nprocs;
        const int tag      = 0;
        if (rank == 0) {
            MPI_Status status;
#pragma acc host_data use_device(b)                        MPI(GPU)
            MPI_Recv(b, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD, &status);
        } else {
#pragma acc host_data use_device(a)                        MPI(GPU)
            MPI_Send(a, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD);
        }

#pragma acc kernels present(b[0:n])
#pragma acc loop reduction(+:sum)
        for (unsigned int i=0; i<n; i++) {
            sum += b[i];
        }
    }                                                      deallocate
```

CUDA-aware MPIコード（2）

- 通信方法
  - ✓ MPI_Send, MPI_Recv に対してhost_data 指示文でデバイス側のアドレスを渡している。CUDA-aware MPI であれば正しく動作する。
  - ✓ 03_update と比較し、CUDA IPC が利用され、ホストメモリを介さない通信となる。NVLink経由でGPU間通信する。
- コンパイルと実行

```
$ make
$ pjsub ./run.sh
$ cat run.sh.??????.out
num of GPUs = 2
Rank 0: hostname = wa36, GPU num = 0
Rank 1: hostname = wa36, GPU num = 1
mean = 30.00
Time =    0.024 [sec]
```

（参考）2ノードの2GPUでの実行

- 各ノード1GPUで2ノードを利用

openacc_mpi_basic/ 05_no_gdr

✓ ジョブスクリプトを確認します。(講習会では利用できません)

```
$ cat run.sh
#!/bin/bash
#PJM -L rscgrp=regular-a
#PJM -L node=2
#PJM --mpi proc=2
#PJM -L elapse=00:10:00
#PJM -g gz00

module purge
module load nvidia cuda ompi-cuda

export UCX_MEMTYPE_CACHE=n
export UCX_IB_GPU_DIRECT_RDMA=n

mpiexec -machinefile $PJM_O_NODEINF -n $PJM_MPI_PROC -npernode 1 ./run
```

node=2:

使用するノード数を指定。この場合は2ノード。

--mpi proc=2:

総MPIプロセス数。この場合は2プロセス。

mpiexec -np -machinefile $PJM_O_NODEINF $PJM_MPI_PROC -npernode 1:

-n に全並列数（全MPIプロセス数）を指定。$PJM_MPI_PROCが—mpi procの数字になる。-npernodeで1ノードあたりのプロセス数を指定する。この場合は１。

（参考）GPUDirectRDMA無効コード

- 05_no_gdr コード

  openacc_mpi_basic/ 05_no_gdr

  - ✓ main.c 自体は 04_cuda_aware と同じ。run.sh のみ変更。
- 通信方法
  - ✓ MPI_Send, MPI_Recv に対してhost_data 指示文でデバイス側のアドレスを渡している。CUDA-aware MPI であれば正しく動作する。
  - ✓ 04_cuda_aware と比較し、2つのGPUはノードが異なるため、MPI通信は InfiniBand を通過する。Wisteria-Aquariusのデフォルトでは、GPU Direct RDMA通信となる。
- コンパイルと実行

```
$ make
$ pjsub ./run.sh
$ cat run.sh.o??????
num of GPUs = 8
Rank 0: hostname = wa07, GPU num = 0
Rank 1: hostname = wa10, GPU num = 1
mean = 30.00
Time =    0.099 [sec]
```

（参考）GPUDirectRDMA有効コード（１）

- 06_gdr コード

  openacc_mpi_basic/ 06_gdr

  - main.c 自体は 04_cuda_aware と同じ。run.sh のみ変更。
- GPUDirectRDMA の有効化
  - ✓ WisteriaのOpenMPI では、デフォルトで GPUDirectRDMA は有効である。これを無効にするには、05_no_gdrのスクリプトに出てきた、UCX_IB_GPU_DIRECT_RDMAをnoにする。

```
export UCX_IB_GPU_DIRECT_RDMA=n   #無効の時
export UCX_IB_GPU_DIRECT_RDMA=y   #有効の時
```

- 通信方法
  - ✓ 05_no_gdr と比較し、 GPUDirectRDMA を有効とする。ホストメモリを経由することなく、直接データ転送 (RDMA) をする。

（参考）GPUDirectRDMA有効コード（２）

- コンパイルと実行

openacc_mpi_basic/ 06_gdr

```
$ make
$ pjsub ./run.sh
$ cat run.sh.??????.out
num of GPUs = 8
Rank 1: hostname = wa10, GPU num = 1
Rank 0: hostname = wa07, GPU num = 0
mean = 30.00
Time =    0.026 [sec]
```

OpenACC+MPI化のステップのまとめ

- OpenACC+MPI化
  - ✓ acc_set_device_numによるMPIランクとGPUの対応付け
  - ✓ kernels, loop, data 指示文を用いてOpenACC化
  - ✓ host_data指示文を用い、MPIにデバイス側のアドレスを渡す

```
double sum = 0.0;
#pragma acc data create(a[0:n], b[0:n])
    {
#pragma acc kernels copyout(a[0:n], b[0:n])
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            a[i] = 3.0 * rank * ny;
            b[i] = 0.0;
        }

        const int dst_rank = (rank + 1) % nprocs;
        const int tag      = 0;
        if (rank == 0) {
            MPI_Status status;
#pragma acc host_data use_device(b)
            MPI_Recv(b, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD, &status);
        } else {
#pragma acc host_data use_device(a)
            MPI_Send(a, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD);
        }

#pragma acc kernels copyin(b[0:n])
#pragma acc loop reduction(+:sum)
        for (unsigned int i=0; i<n; i++) {
            sum += b[i];
        }
    }
```

openacc_mpi_basic/06_gdr

OPENACCとMPIによる
マルチGPUプログラミング実習

実習

- MPI並列化された3次元拡散方程式のOpenACC化
- サンプルコード： openacc_mpi_diffusion/01_original
  - ✓ 3次元拡散方程式のCPUコードにOpenACC のacc_set_device_numでGPUを割り当て、kernels, data, loop, host_data 指示文を追加し、複数GPUで高性能で実行しましょう。

diffusion.c, diffusion3d 関数内

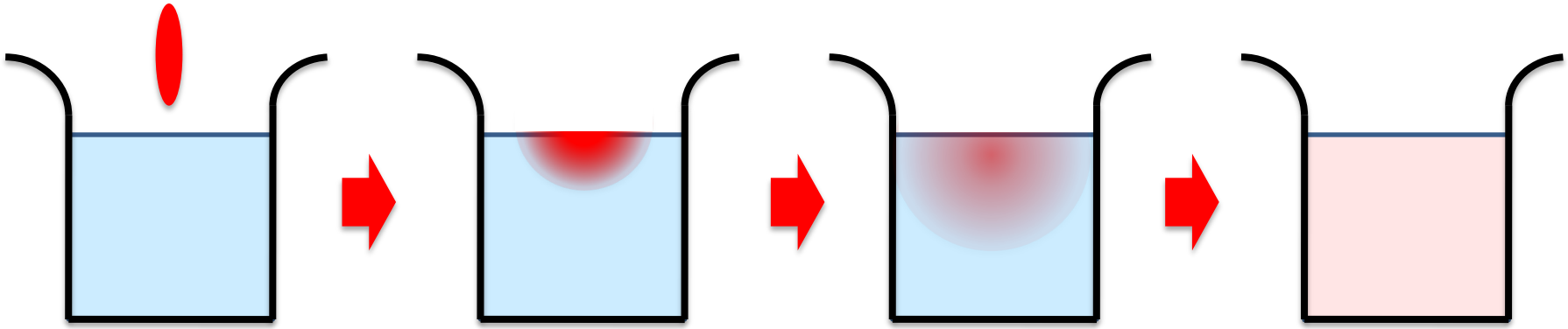
```
for(int k = 0; k < nz; k++) {
    for (int j = 0; j < ny; j++) {
        for (int i = 0; i < nx; i++) {
            const int ix = nx*ny*(k+mgn) + nx*j + i;
            const int ip = i == nx - 1 ? ix : ix + 1;
            const int im = i == 0      ? ix : ix - 1;
            const int jp = j == ny - 1 ? ix : ix + nx;
            const int jm = j == 0      ? ix : ix - nx;
            const int kp = (rank == nprocs - 1 && k == nz - 1) ? ix : ix + nx*ny;
            const int km = (rank == 0          && k == 0     ) ? ix : ix - nx*ny;

            fn[ix] = cc*f[ix]
                   + ce*f[ip] + cw*f[im]
                   + cn*f[jp] + cs*f[jm]
                   + ct*f[kp] + cb*f[km];
            }
        }
    }
```

openacc_mpi_diffusion/01_original

拡散現象シミュレーション（1）

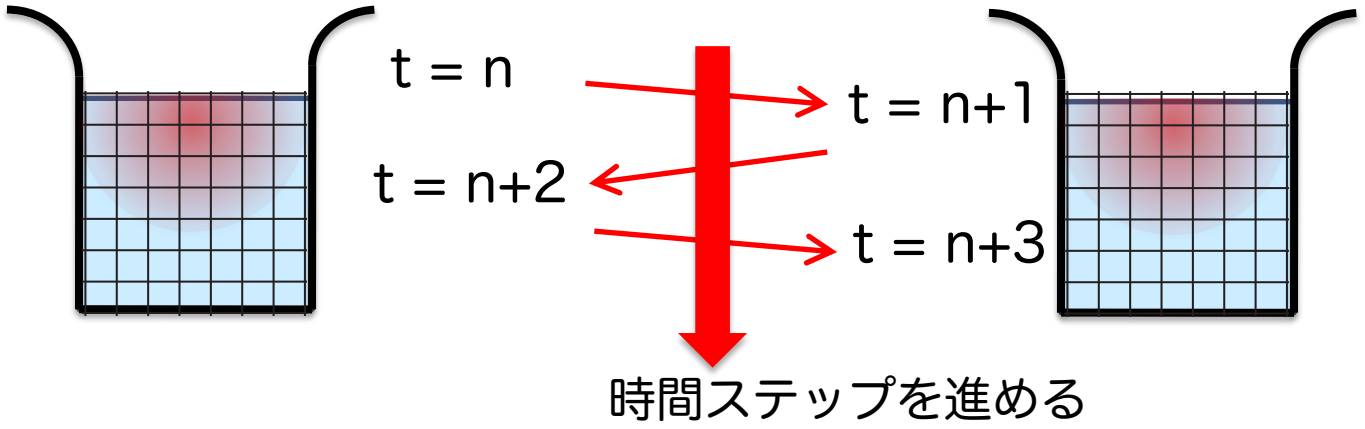
- 拡散現象
  - ✓ コップの中に赤インクを落とすと水中で拡がる
  - ✓ 次第に拡散し赤インクは拡がり、最後は均一な色になる。



- 拡散方程式のシミュレーション
  - ✓ 各点のインク濃度の時間変化を計算する

拡散現象シミュレーション（2）

- データ構造
  - ✓ 計算したい空間を格子に区切り、一般に配列で表す。
  - ✓ 計算は3次元であるが、C言語では1次元配列として確保することが一般的。
  - ✓ 2ステップ分の配列を使い、タイムステップを進める（ダブルバッファ）。



- サンプルコードは、
  - ✓ 計算領域: nx * ny * nz （3次元）
  - ✓ 最大タイムステップ: nt

  となっている。

拡散現象シミュレーション（3）

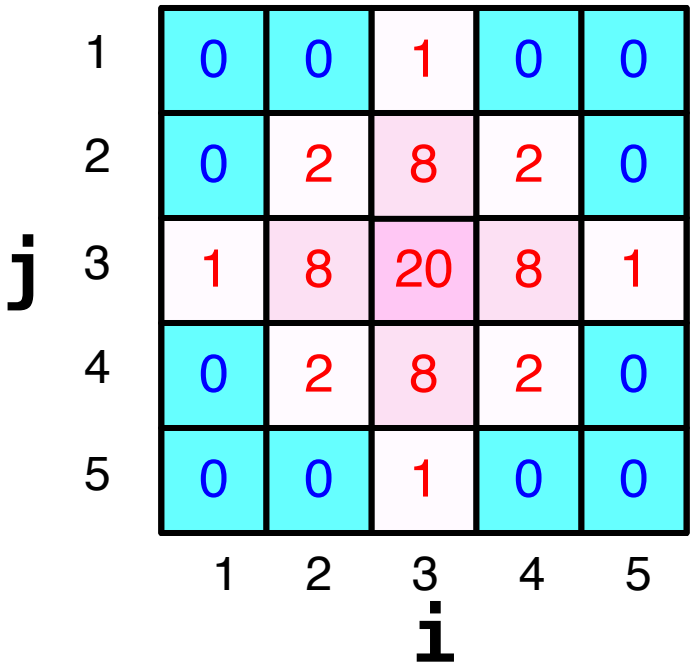
- 2次元拡散方程式の離散化の一例

$$f_{i,j}^{n+1} = (f_{i-1,j}^{n} + f_{i+1,j}^{n} + f_{i,j-1}^{n} + f_{i,j+1}^{n} + 4f_{i,j}^{n})/8$$

平均後の
自分自身の値

上下左右の値

自分自身の値の4倍

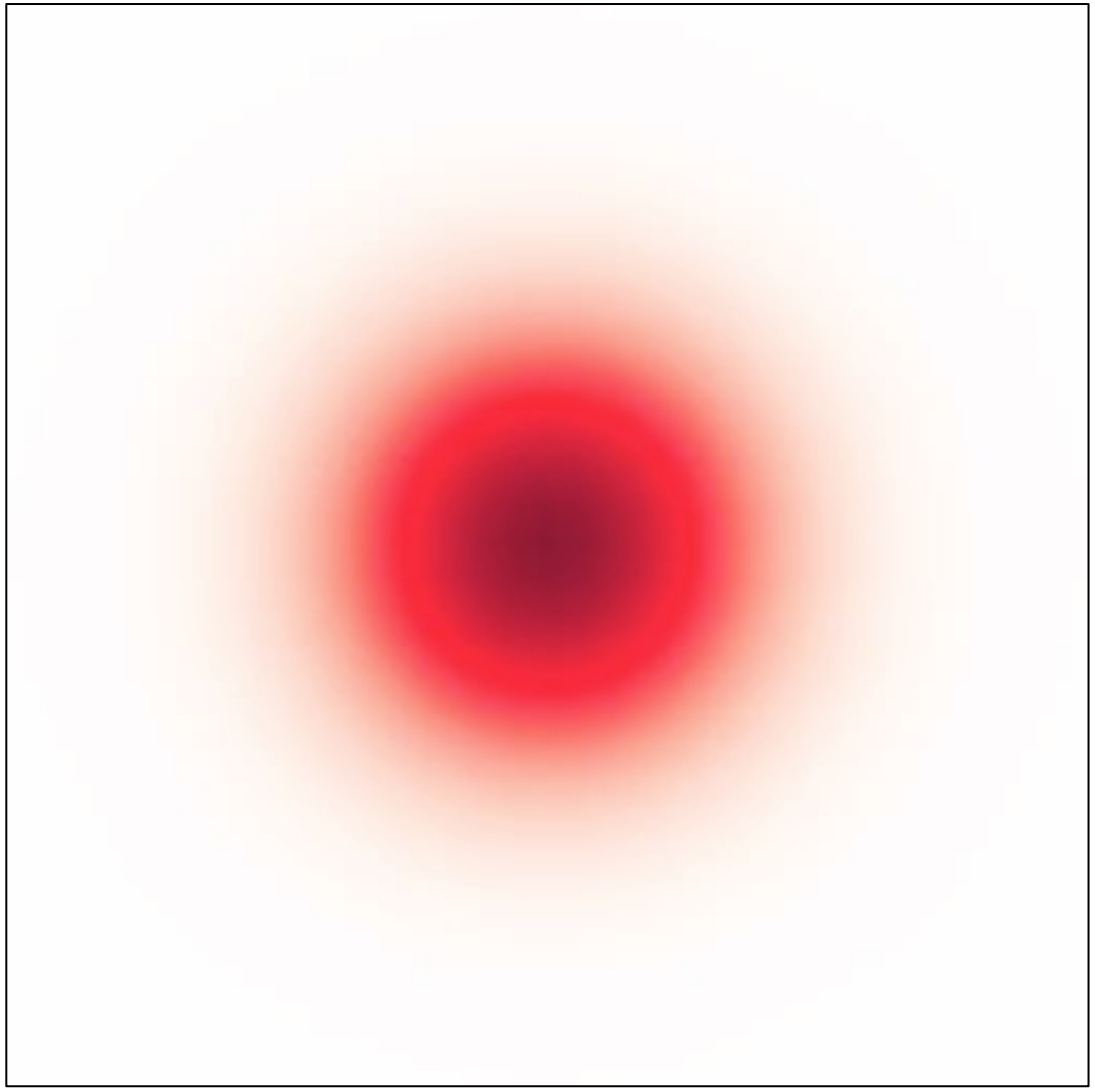


2回目の平均後

繰り返し平均化を行うと、インクが拡散します。

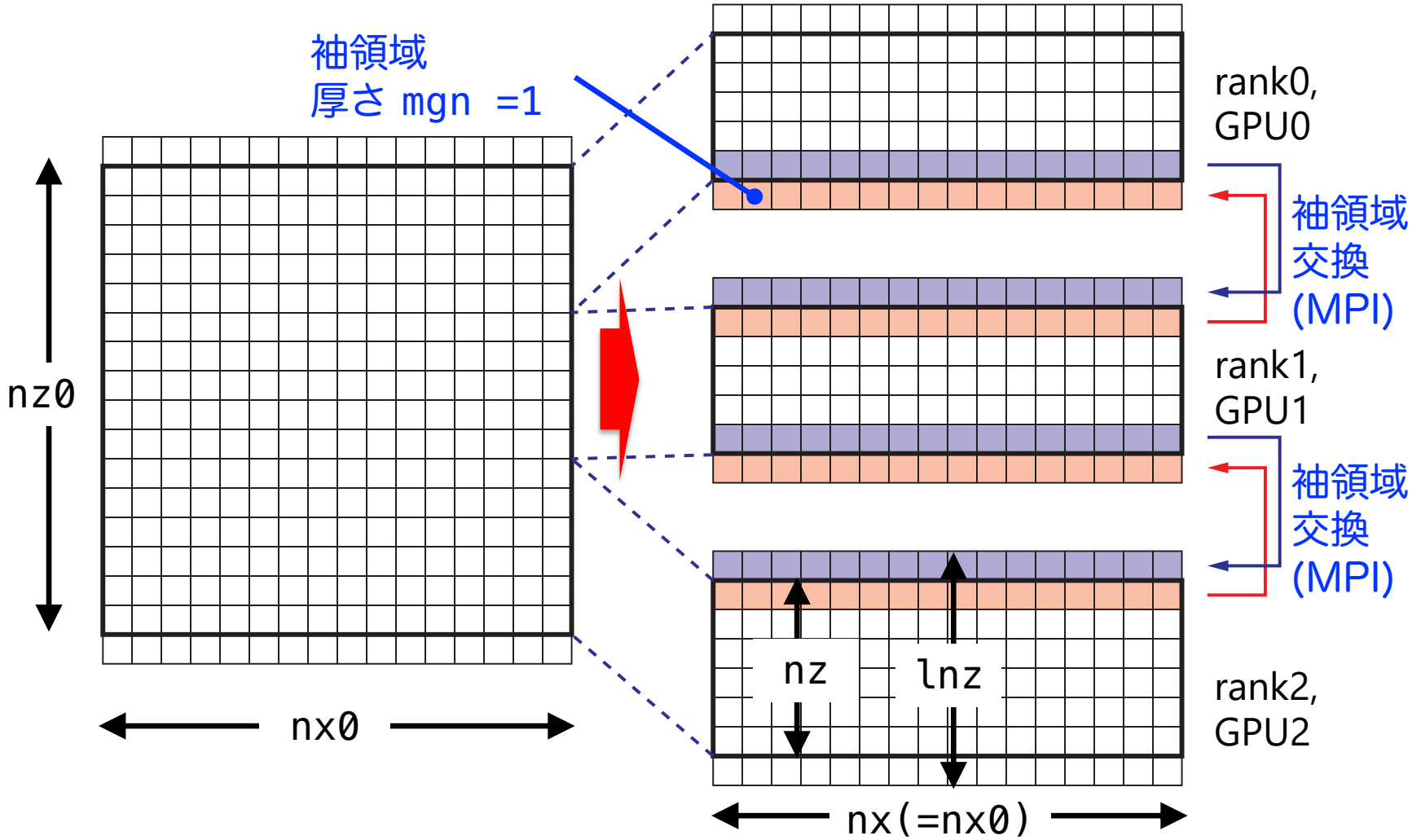
拡散現象シミュレーション（４）

- 2次元拡散方程式の計算例



マルチGPUのための領域分割

- 並列計算するために計算領域を分割する
  - ✓ z 方向で分割する。各計算領域はデータ交換のための袖領域（halo）を持つ。



時間発展

- diffusion3d 関数の前に袖領域のデータ交換のために MPI_Send/MPI_Recv が実行される。

main.c, main 関数内

```
for (; icnt<nt && time + 0.5*dt < 0.1; icnt++) {
    if (rank == 0 && icnt % 100 == 0)
        fprintf(stdout, "time(%4d) = %7.5f¥n", icnt, time);

    const int tag = 0;
    MPI_Status status;

    MPI_Send(&f[nx*ny*nz]      , nx*ny, MPI_FLOAT, rank_up  , tag, MPI_COMM_WORLD);
    MPI_Recv(&f[0]             , nx*ny, MPI_FLOAT, rank_down, tag, MPI_COMM_WORLD, &status);

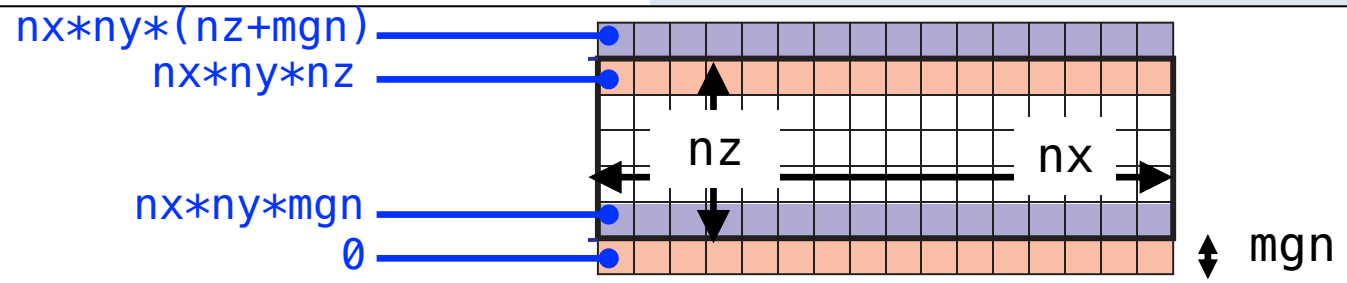
    MPI_Send(&f[nx*ny*mgn]     , nx*ny, MPI_FLOAT, rank_down, tag, MPI_COMM_WORLD);
    MPI_Recv(&f[nx*ny*(nz+mgn)], nx*ny, MPI_FLOAT, rank_up  , tag, MPI_COMM_WORLD, &status);

    flop += diffusion3d(nprocs, rank, nx, ny, nz, mgn, dx, dy, dz, dt, kappa, f, fn);

    swap(&f, &fn);

    time += dt;
}
```

openacc_mpi_diffusion/01_original



CPUコード

- CPUコードのコンパイルと実行

```
$ cd openacc_mpi_diffusion/01_original
$ make
$ pjsub ./run.sh
# cat run.sh.o??????
nprocs = 4
Rank 1: hostname = a086
Rank 0: hostname = a086
Rank 2: hostname = a087
Rank 3: hostname = a087
time(   0) = 0.00000
time( 100) = 0.00610
...
time(1500) = 0.09155
time(1600) = 0.09766
Time =    6.208 [sec]
Performance=    1.80 [GFlops/nprocs]
                7.19 [GFlops]
Error[128][128][128] = 4.556413e-06
```

実行性能（プロセスあたり）

実行性能（合計）

解析解との誤差

- OpenACCコードでは、どのくらいの実行性能が達成できるでしょうか？

OpenACC化(0): Makefile の修正

- Makefile に OpenACC をコンパイルするよう –acc などを追加しましょう

```
CC   = mpicc
CXX  = mpic++
GCC  = gcc
RM  = rm -f
MAKEDEPEND = makedepend

CFLAGS    = -O3 -acc -Minfo=accel -gpu=cc80
GFLAGS    = -Wall -O3 -std=c99
CXXFLAGS  = $(CFLAGS)
LDFLAGS   =
...
```

OpenACC化(1): GPU割り当て

- ヘッダー追加とmain関数に acc_set_device_num等を追加

main.c, main 関数内

```
...
#include <openacc.h>
...

int main(int argc, char *argv[])
{
...
    const int ngpus = acc_get_num_devices(acc_device_nvidia);
    if (rank == 0) {
        fprintf(stdout, "num of GPUs = %d¥n", ngpus);
    }
    const int gpuid = ngpus > 0 ? rank % ngpus : -1;
    if (gpuid >= 0) {
        acc_set_device_num(gpuid, acc_device_nvidia);
    }

...
```

OpenACC化(2): kernels, loop

- diffusion3d関数に kernels, loopを追加。present 指定。

diffusion.c, diffusion3d 関数内

```
#pragma acc kernels present(f, fn)
#pragma acc loop independent
    for(int k = 0; k < nz; k++) {
#pragma acc loop independent
        for (int j = 0; j < ny; j++) {
#pragma acc loop independent
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*(k+mgn) + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = (rank == nprocs - 1 && k == nz - 1) ? ix : ix + nx*ny;
                const int km = (rank == 0          && k == 0     ) ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix] + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm] + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

OpenACC化(3): data, host_data

- main関数で data, host_data を追加

main.c, main 関数内

```
#pragma acc data copy(f[0:ln]) create(fn[0:ln])
    {
        start_timer();

        for (; icnt<nt && time + 0.5*dt < 0.1; icnt++) {
            if (rank == 0 && icnt % 100 == 0)
                fprintf(stdout, "time(%4d) = %7.5f¥n", icnt, time);

            const int tag = 0;
            MPI_Status status;

#pragma acc host_data use_device(f)
            {
                MPI_Send(&f[nx*ny*nz]      , nx*ny, MPI_FLOAT, rank_up  , tag, MPI_COMM_WORLD);
                MPI_Recv(&f[0]             , nx*ny, MPI_FLOAT, rank_down, tag, MPI_COMM_WORLD, &status);

                MPI_Send(&f[nx*ny*mgn]     , nx*ny, MPI_FLOAT, rank_down, tag, MPI_COMM_WORLD);
                MPI_Recv(&f[nx*ny*(nz+mgn)], nx*ny, MPI_FLOAT, rank_up  , tag, MPI_COMM_WORLD, &status);
            }

            flop += diffusion3d(nprocs, rank, nx, ny, nz, mgn, dx, dy, dz, dt, kappa, f, fn);

            swap(&f, &fn);

            time += dt;
        }

        MPI_Barrier(MPI_COMM_WORLD);
        elapsed_time = get_elapsed_time();

    }
```

NVCOMPILER_ACC_TIME によるOpenACC 実行の確認

- NVIDIAコンパイラを利用する場合、OpenACCプログラムがどのように実行されているか、環境変数NVCOMPILER_ACC_TIMEを設定すると簡単に確認することができる。
- Linuxなどでは、環境変数NVCOMPILER_ACC_TIME を1に設定し、プログラムを実行する。

```
$ export NVCOMPILER_ACC_TIME=1
$ ./run
```

- Wisteria でジョブに環境変数NVCOMPILER_ACC_TIME を設定する場合は、ジョブスクリプト中に記載する。

```
$ cat run.sh
...

export NVCOMPILER_ACC_TIME=1
./run
```

NVCOMPILER_ACC_TIME によるOpenACC 実行の確認

- ジョブ実行が終わると、標準エラー出力にメッセージが出力される。

```
$ cat run.sh.??????.err
Accelerator Kernel Timing data
/work/01/gt00/z30108/openacc_samples_test/openacc_samples/C/openacc_dif
fusion/03_openacc_pgi_acc_time/main.c
  main  NVIDIA  devicenum=0
    time(us): 725
    39: data region reached 2 times
        39: data copyin transfers: 1
             device time(us): total=342 max=342 min=342 avg=342
        53: data copyout transfers: 1
             device time(us): total=383 max=383 min=383 avg=383
/work/01/gt00/z30108/openacc_samples_test/openacc_samples/C/openacc_dif
fusion/03_openacc_pgi_acc_time/diffusion.c
  diffusion3d  NVIDIA  devicenum=0
    time(us): 0
    17: compute region reached 1638 times
        25: kernel launched 1638 times
            grid: [16384]  block: [128]
            elapsed time(us): total=67,084 max=54 min=40 avg=40
    17: data region reached 3276 times
```

データ移動の回数

起動したスレッド

カーネル実行時間

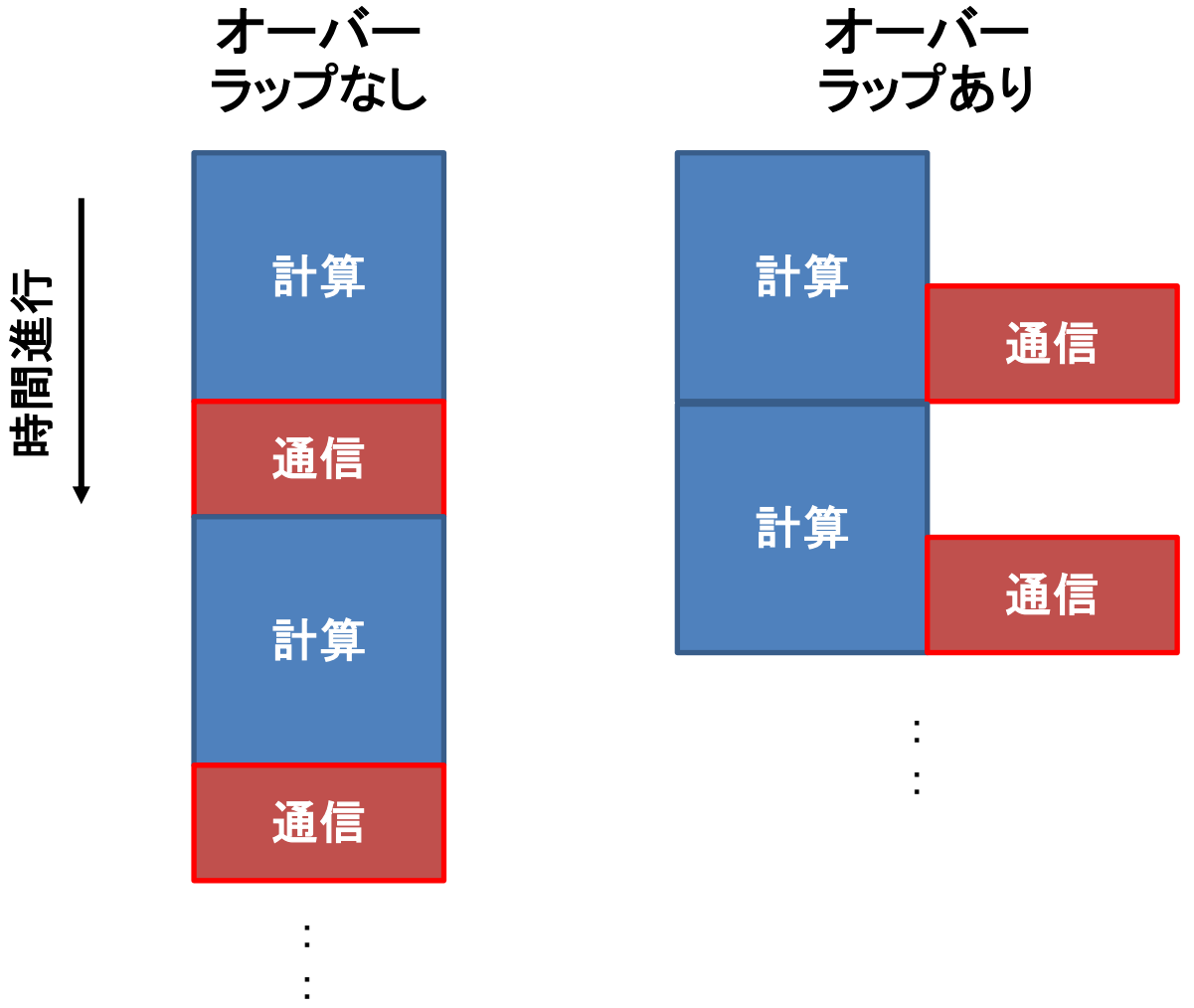
OpenACC化(4): 性能測定

- まずはそのまま make して実行してみましょう。
- 条件を変更して性能測定してみましょう。
  - ✓ 計算格子サイズを 128^3 から 512^3 へ変更
    - ✓GPUでは計算領域が大きいほうが性能が出やすい。
  - ✓ 2 GPU -> 4 GPU
- NVCOMPILER_ACC_TIME によるOpenACC 実行の確認

OpenACC化の例は、openacc_mpi_diffusion/02_openacc

通信と計算のオーバラップ(1)

- よりスケーリングさせるためには、計算と通信のオーバーラップを目指す必要がある
- 実装すべきこと
  1. 袖領域計算と内点の計算の分離
  2. OpenACCのasync 指示節とwait 指示文の利用
  3. MPIの非同期関数による通信



async指示節とwait指示文

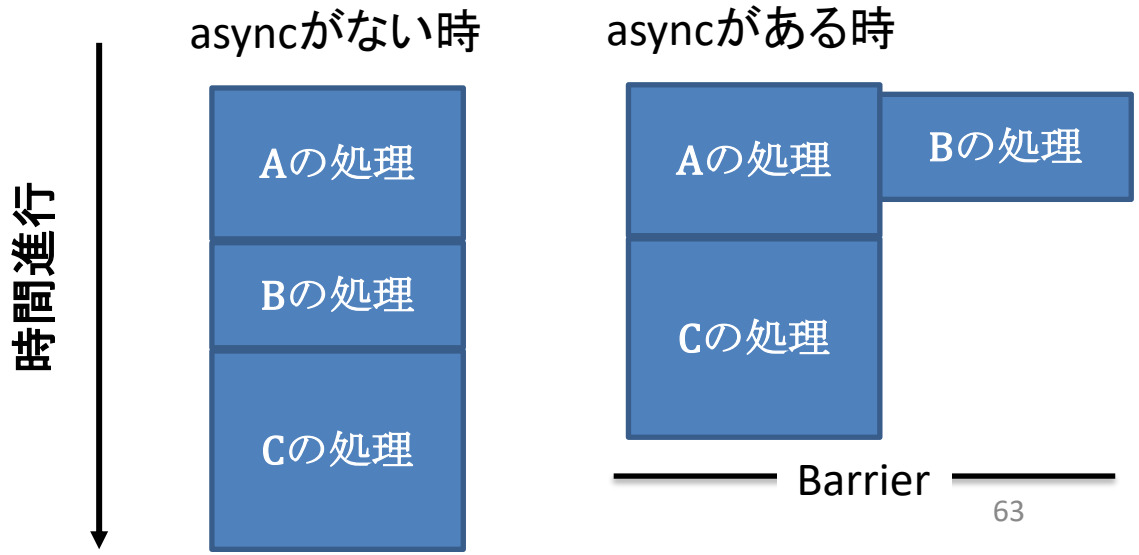
- async指示節
  - 主にkernels指示文に付与する
  - async(整数)と指定する
  - (整数)が違うkernelが非同期実行される
    - GPUのリソースに空きがあれば同時に実行される
    - (整数)が同じなら、上から順番に実行される
  - CPU-GPU間の同期を減らす目的でも使われる
- wait指示文
  - wait(整数)で、対応するasync(整数)の同期処理を行う
  - 単にwaitを指定すると、wait allの意味

```
#pragma acc kernels async(0)
for(…){
  /* Aの処理 */
}

#pragma acc kernels async(1)
for(…){
  /* Bの処理 */
}

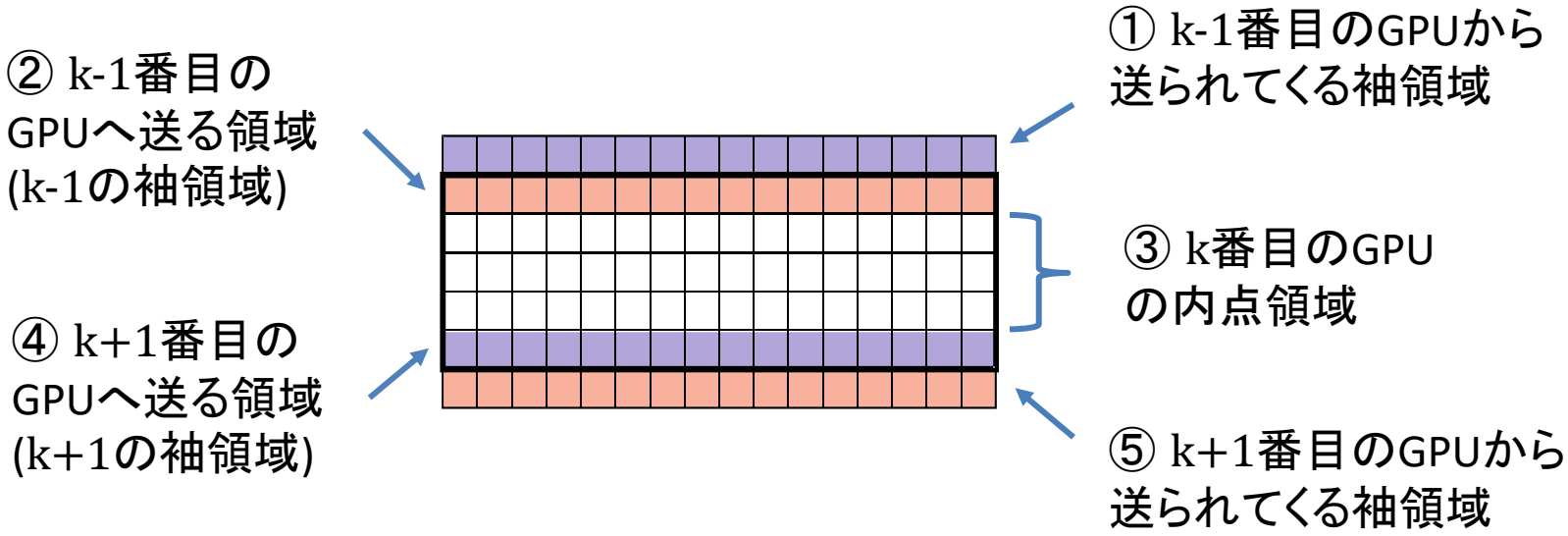
#pragma acc kernels async(0)
for(…){
  /* Cの処理 */
}

#pragma acc wait
```

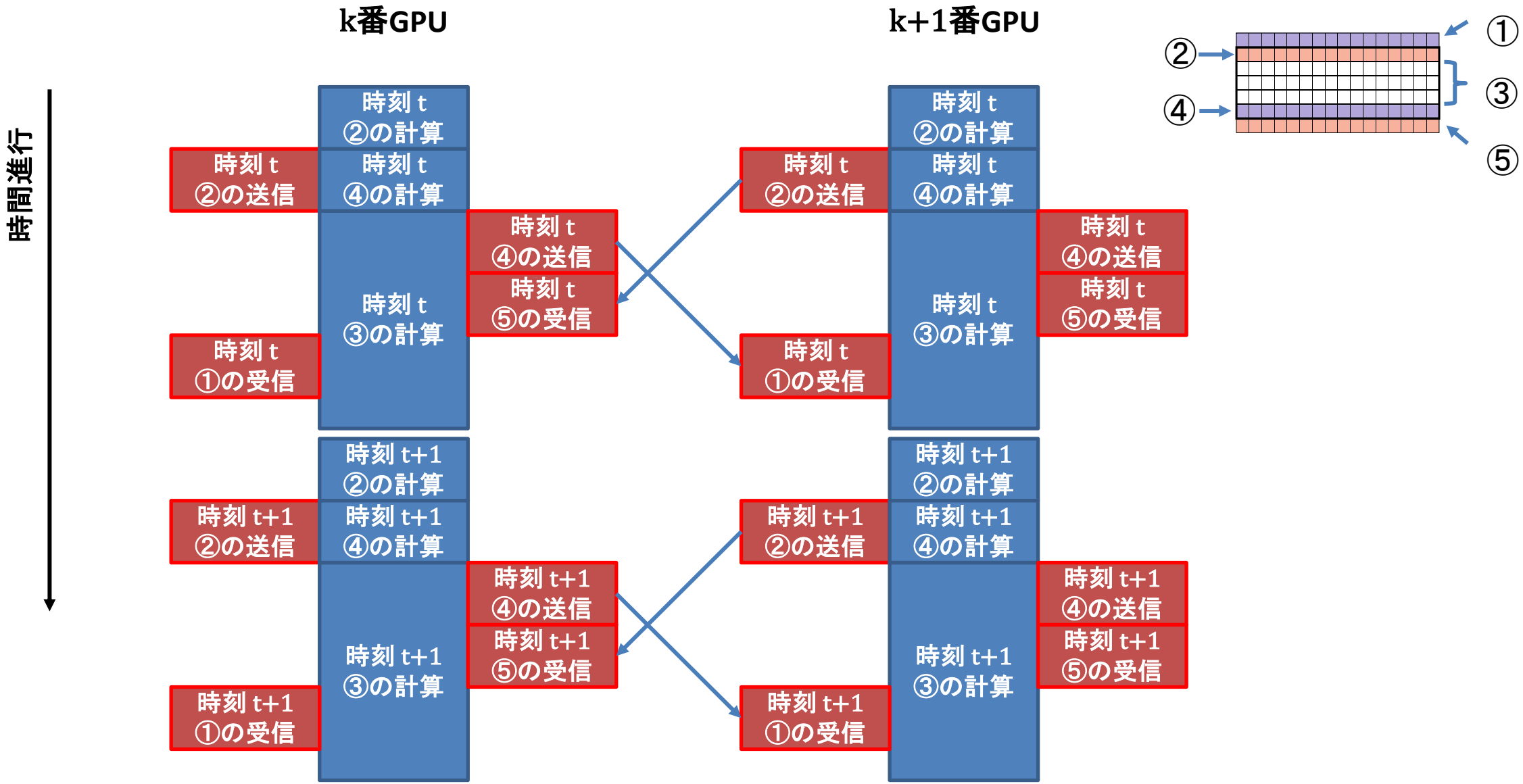


通信と計算のオーバーラップ(2)

- まずはasync・wait無しで考える
  1. 図の②と④の領域をループから分離し、先に計算する
  2. ②・④の計算が終わり次第、Isend・Irecvで通信する
  3. 通信している間に、③の計算を行う
  4. MPI_Barrier()し、次のタイムステップへ

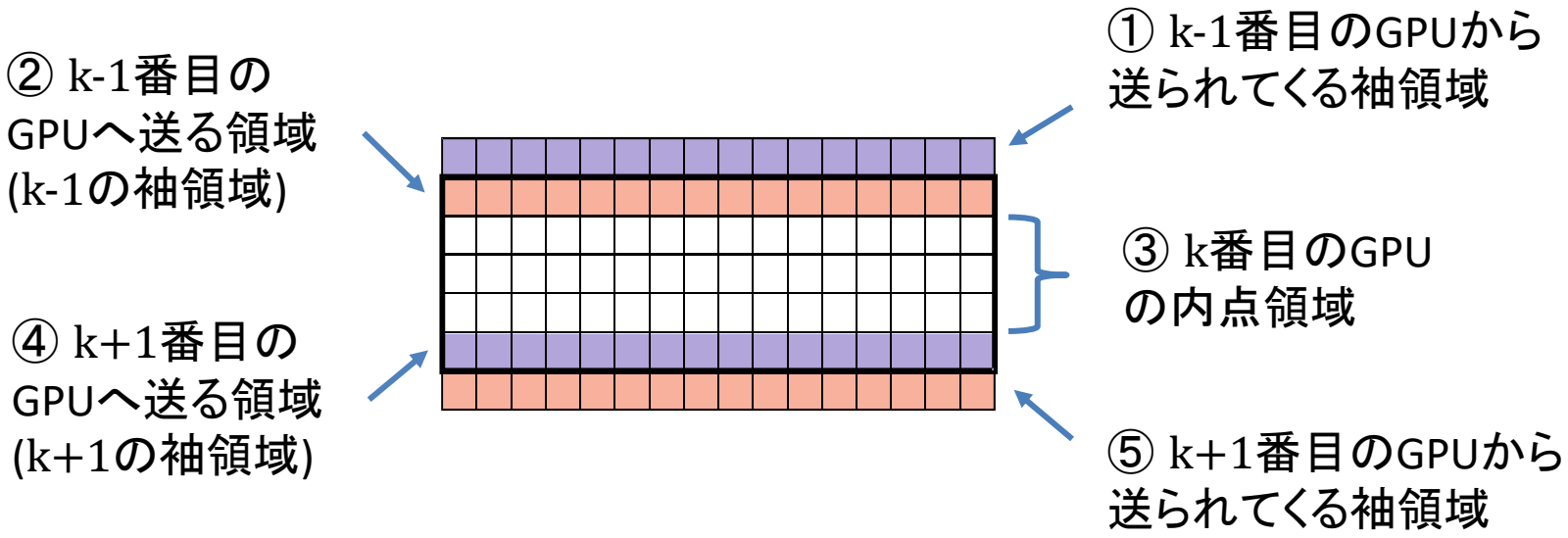


通信と計算のオーバーラップ(2)

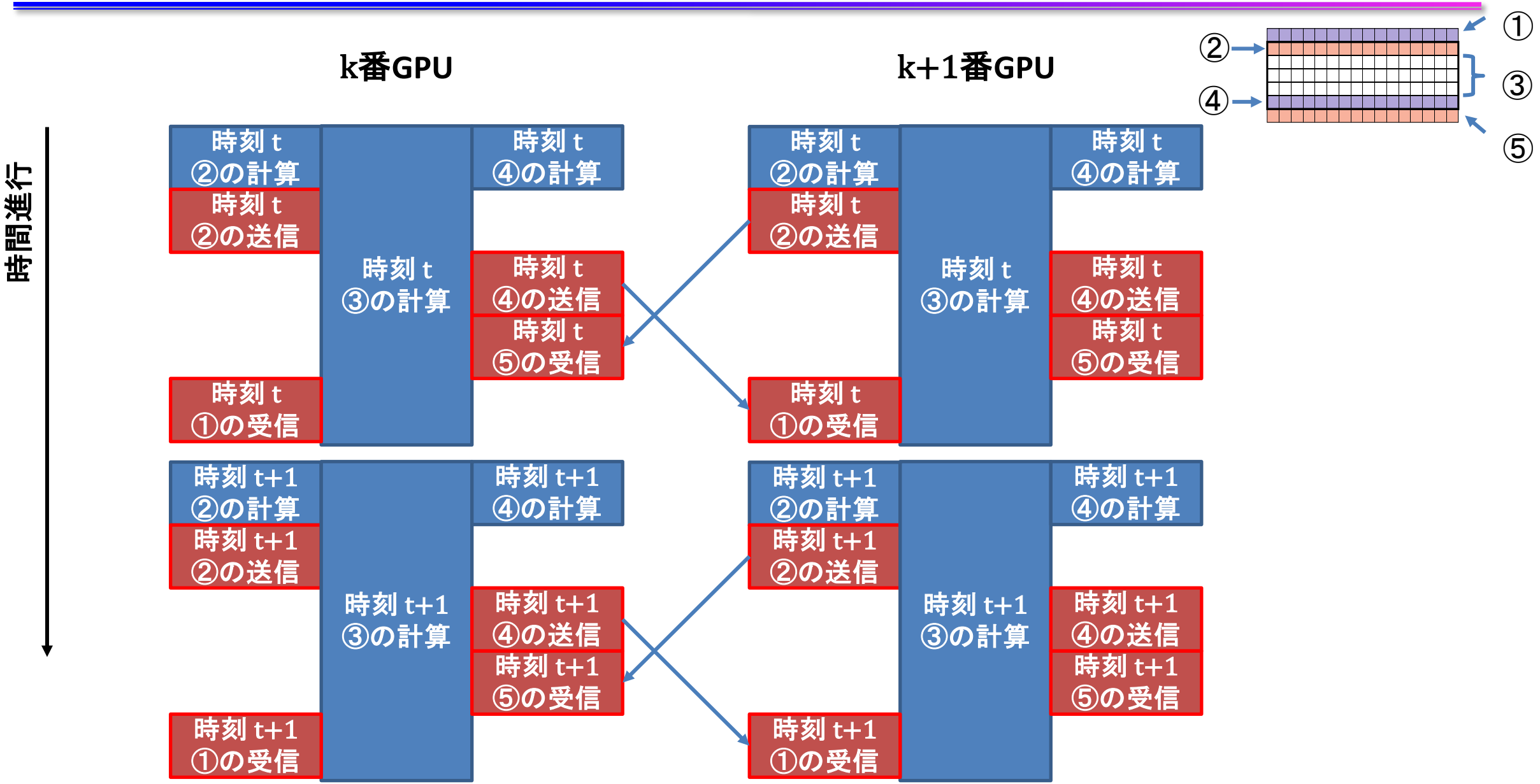


通信と計算のオーバーラップ(3)

- Async・waitありで考える
  1. 図の②と③と④の領域を同時に計算する
     - GPUへ投入する順番は、③が一番最後
  2. ②・④の計算が終わり次第、Isend・Irecvで通信する
  3. MPI_Barrier()、acc waitをして、次のタイムステップへ



通信と計算のオーバーラップ(3)



```
103         for (; icnt<nt && time + 0.5*dt < 0.1; icnt++) {
104             if (rank == 0 && icnt % 100 == 0) fprintf(stdout, "time(%4d) = %7.5f\n", icnt, time);
105
106             const int tag = 0;
107             MPI_Status stat[4];
108             MPI_Request req[4];
109
110 #pragma acc wait(0)  //②の計算が終わるのをここで待つ
111 #pragma acc wait(1)  //④の計算が終わるのをここで待つ
112 #pragma acc host_data use_device(f)
113             {
114                 MPI_Irecv(&f[0]             , nx*ny, MPI_FLOAT, rank_down, tag, MPI_COMM_WORLD, &req[0]); //①の受信準備
115                 MPI_Irecv(&f[nx*ny*(nz+mgn)], nx*ny, MPI_FLOAT, rank_up  , tag, MPI_COMM_WORLD, &req[1]); //⑤の受信準備
116                 MPI_Isend(&f[nx*ny*nz]      , nx*ny, MPI_FLOAT, rank_up  , tag, MPI_COMM_WORLD, &req[2]); //②の送信開始
117                 MPI_Isend(&f[nx*ny*mgn]     , nx*ny, MPI_FLOAT, rank_down, tag, MPI_COMM_WORLD, &req[3]); //④の送信開始
118             }
119             MPI_Waitall(4,req,stat); //①、②、④、⑤の送受信が終わるのをここで待つ
120 #pragma acc wait(2)  //③の計算が終わるのをここで待つ
121
122             flop += diffusion3d(nprocs, rank, nx, ny, nz, mgn, dx, dy, dz, dt, kappa, f, fn); //次ステップの計算
123
124             swap(&f, &fn);
125
126             time += dt;
127         }
```