

第167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](mailto: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

- ✓ 10月18日（火） 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Ⅱ

# 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-vqRIV7wmU0-w/videos?view=0&sort=p&flow=grid>

# 講習会の進め方

---

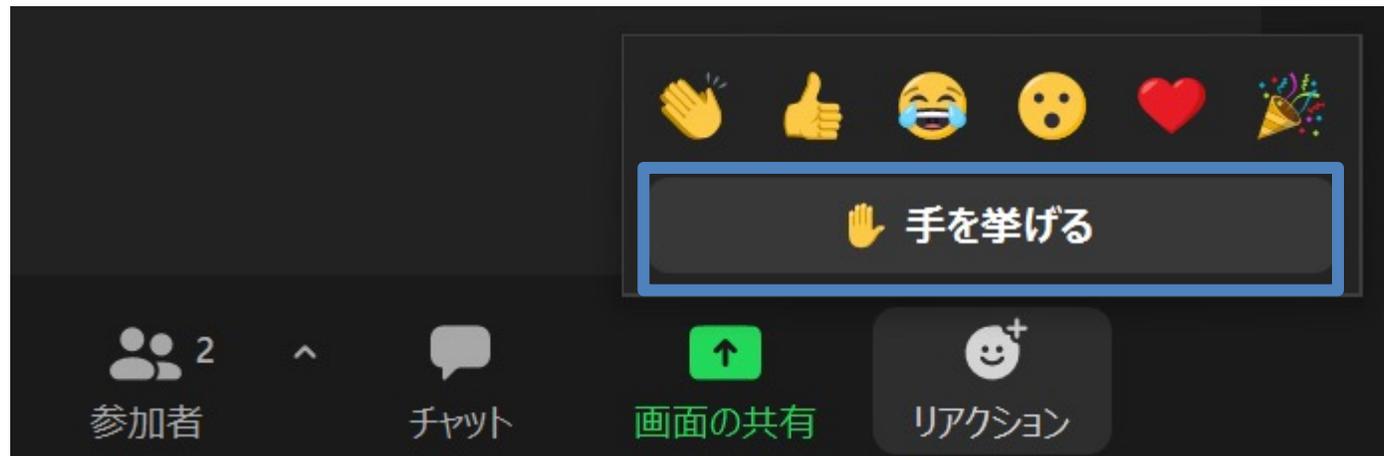
- Zoomを利用したオンライン講習会です
  - ✓ この講義は録画されています
  - ✓ 質問があるとき以外はミュートでお願いします
  - ✓ ビデオもオフを推奨します
- slackを使って質問に対応します
  - ✓ slackはリンクを知っている人は誰でも使える設定になっています
  - ✓ slackのリンクをzoomのチャットに貼るので、未登録の場合は今のうちに登録をお願いします
    - ✓ slackの登録メールの配送に小一時間かかることがあります
  - ✓ スクリーンショットなどで画像を共有することで、質問対応します
    - ✓ Windows : Alt + PrtScn 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

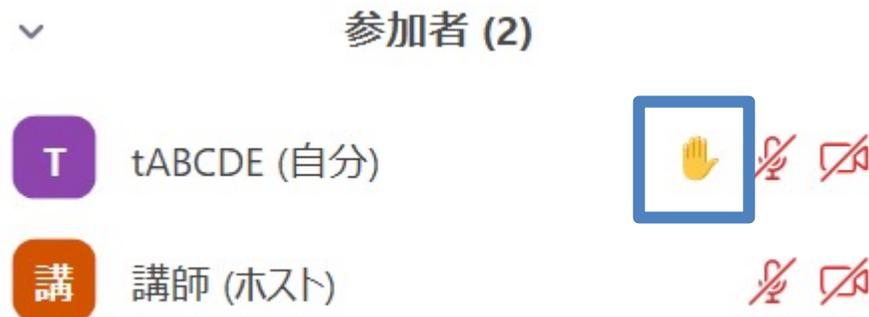


# Zoom: 手が挙がっていることの確認方法

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164回-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

---

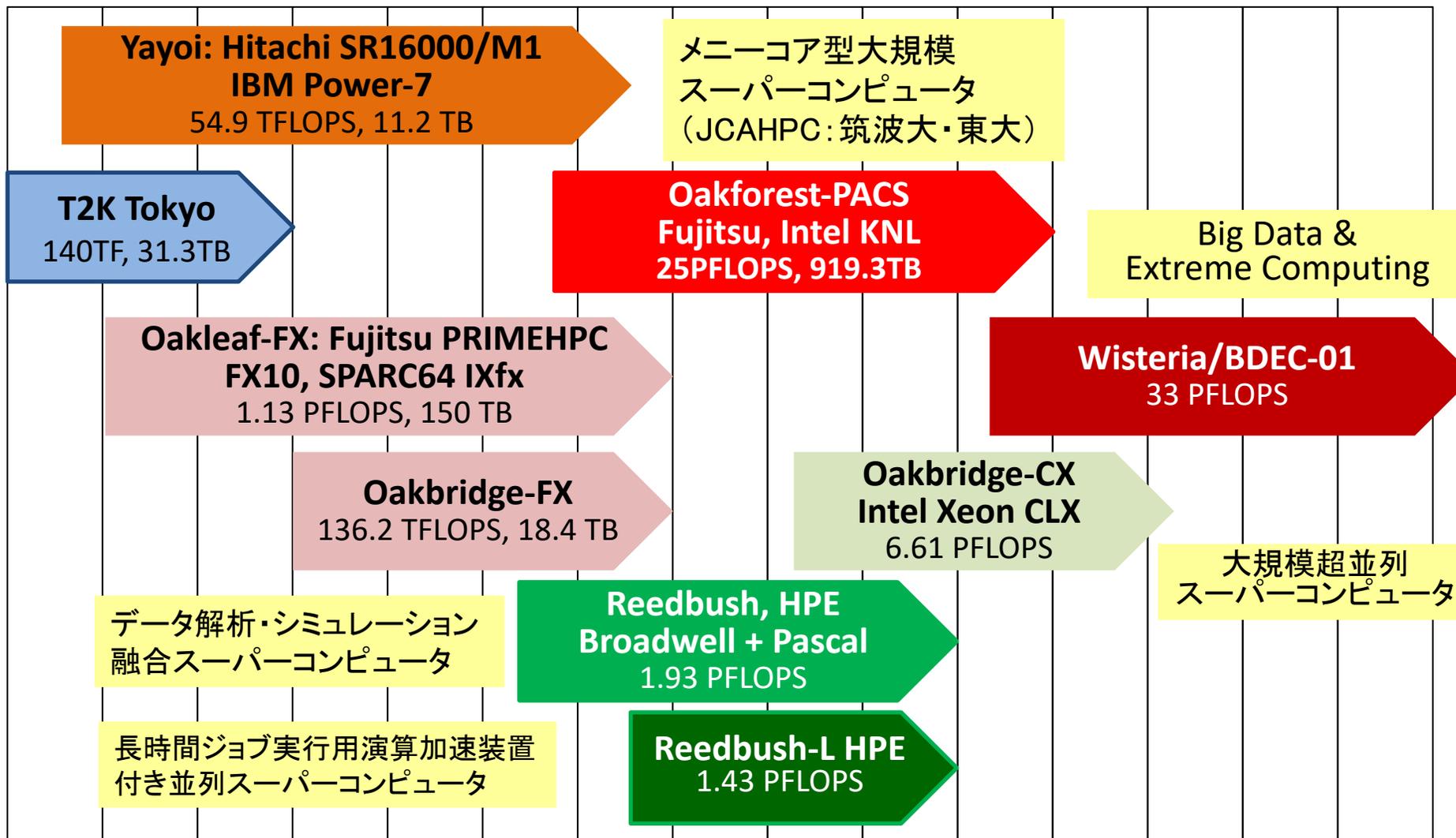
- 講習会で使うサンプルコードはGitHubにあります
  - [https://github.com/hoshino-UTokyo/lecture\\_openacc\\_mpi.git](https://github.com/hoshino-UTokyo/lecture_openacc_mpi.git)
- ダウンロード手順
  1. `$ ssh tXXXXX@wisteria.cc.u-tokyo.ac.jp`
    - wisteriaへのssh。tXXXXXはアカウント名
  2. `$ cd /work/gt00/tXXXXX`
  3. `$ git clone https://github.com/hoshino-UTokyo/lecture_openacc_mpi.git`
  4. `$ cd lecture_openacc/`
  5. `$ cd C or F`
    - CまたはFortran好きな方を利用してください。

# 東大情報基盤センターの スパコン概要

# 東大センターのスパコン

FY 2基の大型システム, 6年サイクル(だった)

11 12 13 14 15 16 17 18 19 20 21 22 23 24 25



# 既存3システム：利用者2,000+, 学外50+%

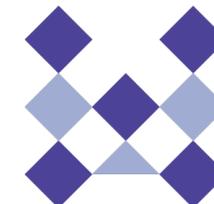
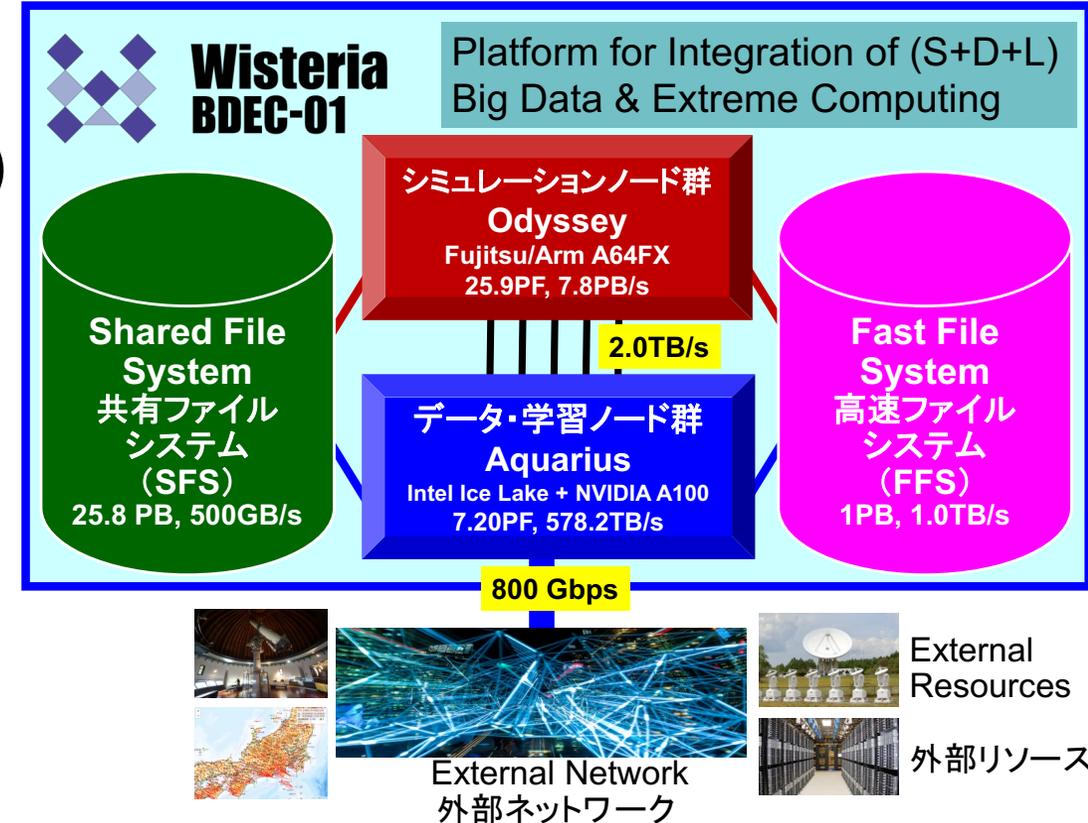
- Reedbush (SGI, Intel BDW + NVIDIA P100 (Pascal))
  - データ解析・シミュレーション融合スーパーコンピュータ
  - 3.36 PF, 2016年7月～**2021年11月末**
  - 東大ITC初のGPUシステム (2017年3月より), DDN IME (Burst Buffer)
- Oakforest-PACS (OFP) (富士通, Intel Xeon Phi (KNL))
  - 2016年10月～**2022年3月末**
  - JCAHPC (筑波大CCS&東大ITC)
  - 25 PF, TOP 500で6位 (2016年11月) (日本1位) (初登場時)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)
- Oakbridge-CX (富士通, Intel Xeon Platinum 8280)
  - 大規模超並列スーパーコンピュータシステム
  - 6.61 PF, 2019年7月～2023年6月
  - 全1,368ノードの内128ノードにSSDを搭載



# Wisteria/BDEC-01

- 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<sup>2</sup>
- 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

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)



**Wisteria**  
**BDEC-01**

# スパコン料金表 (2021年4月時点)

## ■ 最小セット料金表

	トークン※1	料金(大学・公共機関)	ストレージ容量	利用期間
Wisteria/BDEC-01	720 ※2	5,000円	2TB	2021/8/2以降～年度末まで
Reedbush	720 ※3	6,300円	1TB	2021/11/30まで
Oakforest-PACS	720	4,200円	1TB	年度末まで (2021年度で運用終了予定)
Oakbride-CX	720	8,400円	4TB	年度末まで

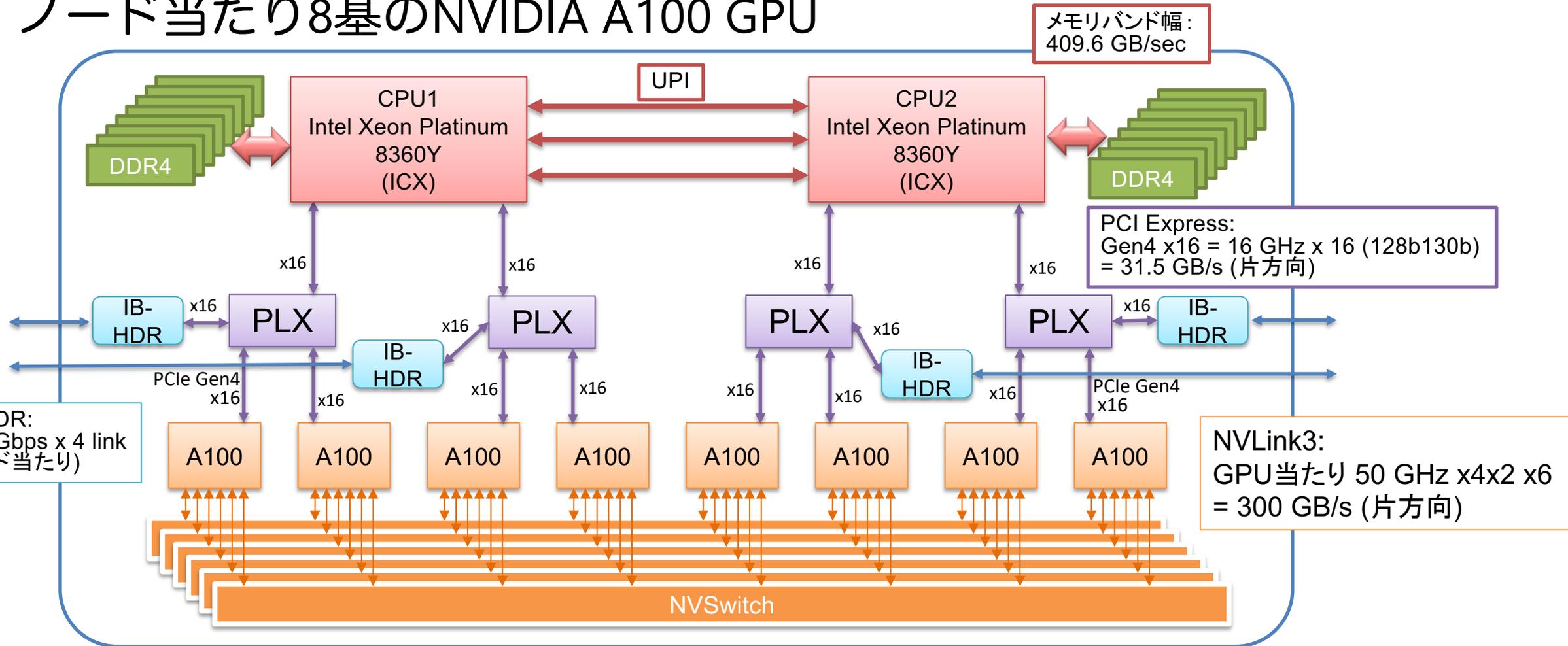
※1 トークン≒ノード時間 ※2, 3。720トークンなら、1ノードを720時間利用できる。

※2 Odyssey(CPUノード)基準。Aquarius(GPUノード)を使う場合、利用するGPU数x3.0倍のトークンを消費する。つまり1ノード(8GPU)を利用する場合、 $720/24=30$ 時間しか利用できない。

※3 Reedbush-H,L はそれぞれ2.5, 4.0倍のトークンを消費する。つまり、Hの場合は $720/2.5=288$ 時間, Lの場合は $720/4=180$ 時間しか利用できない

# Aquariusの構成

- 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



# 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)	
<b>OpenACC</b>			○	

# 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
./a.out
```

リソースグループ名  
:lecture-a

利用GPU数

MPIプロセス数

実行時間制限: 1分  
(講習会では最大10分)

利用グループ名  
:gt00

# 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

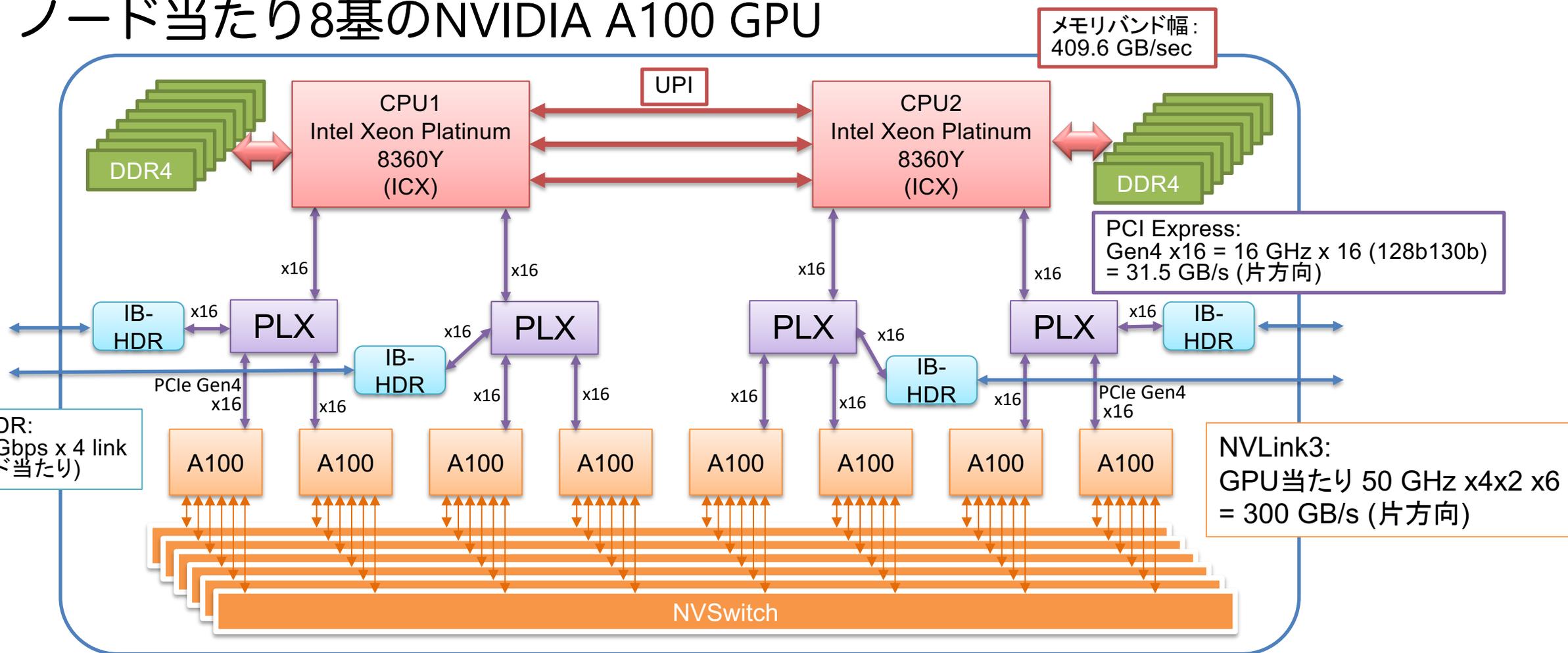
- ✓ できなくはない。OpenMPを使う場合、OpenACCと同じループに指示文の適用はできないため、`omp_get_thread_num()`を用いて並列化することになる。

- ✓ 複数GPUを `acc_set_device_num` (OpenACC) や `cudaSetDevice` (CUDA) で切り替えながら計算

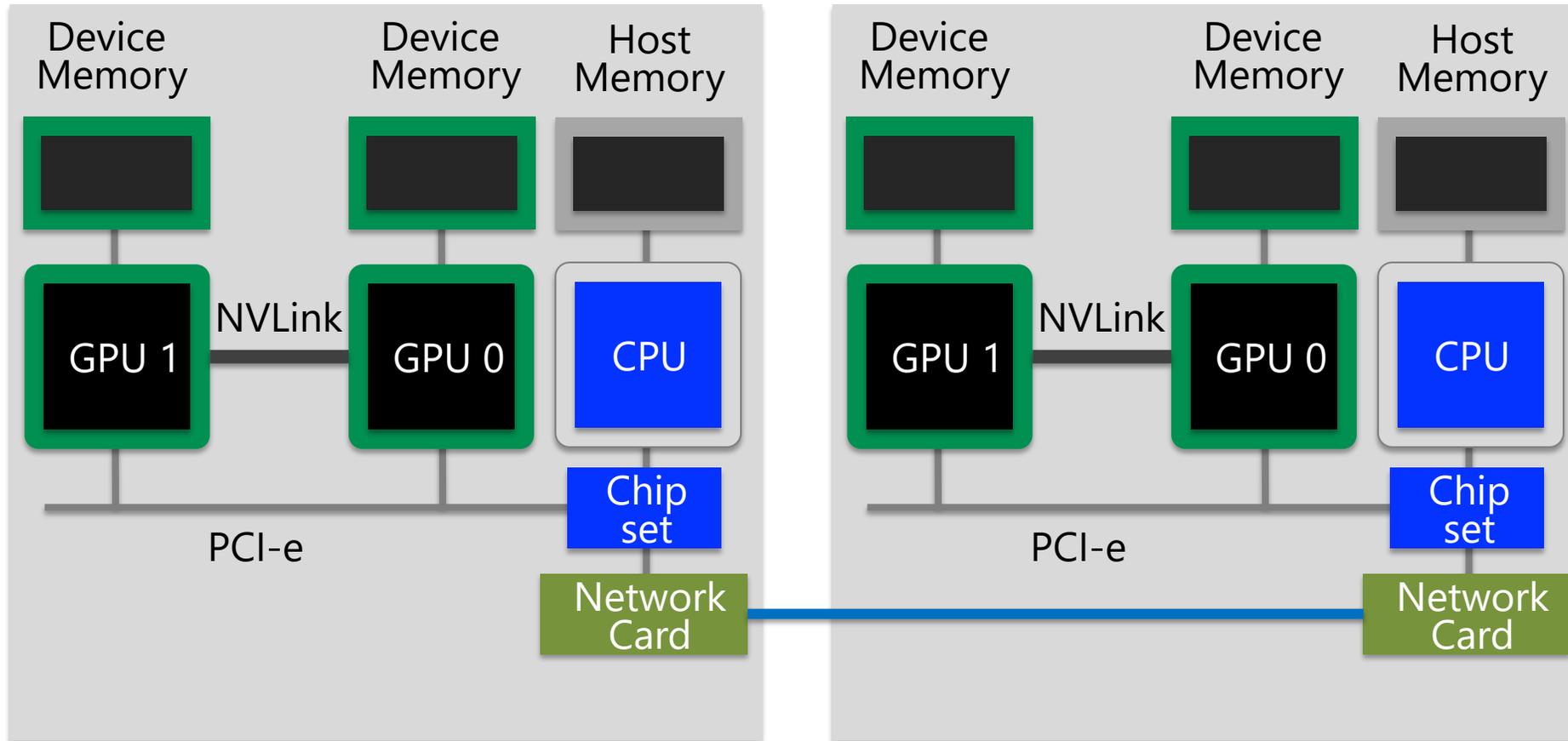
- 本講習会では、複数ノードに搭載された複数のGPUを活用できるMPIによる並列化を行う。

# 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



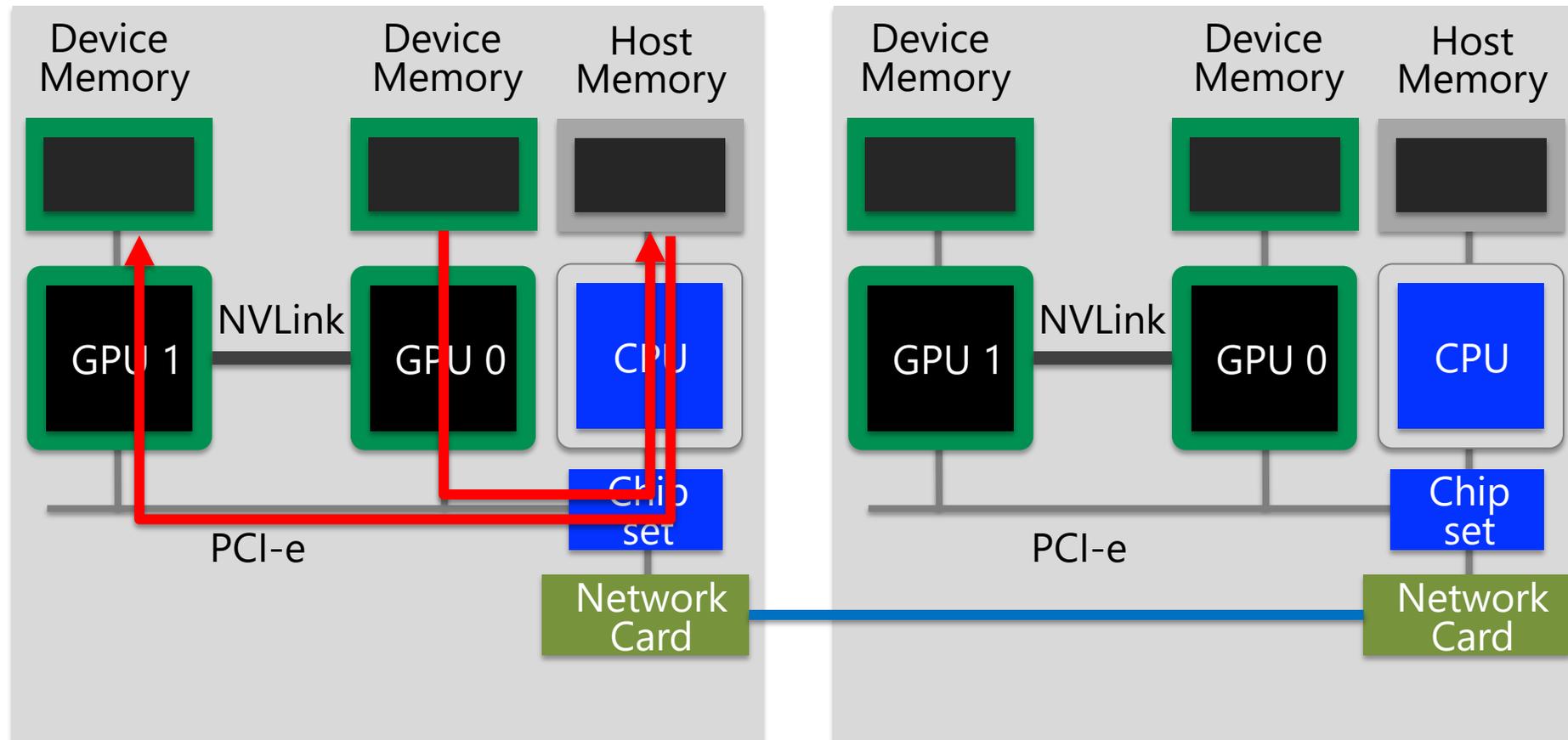
# GPUx2 の計算ノード模式図



- 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)

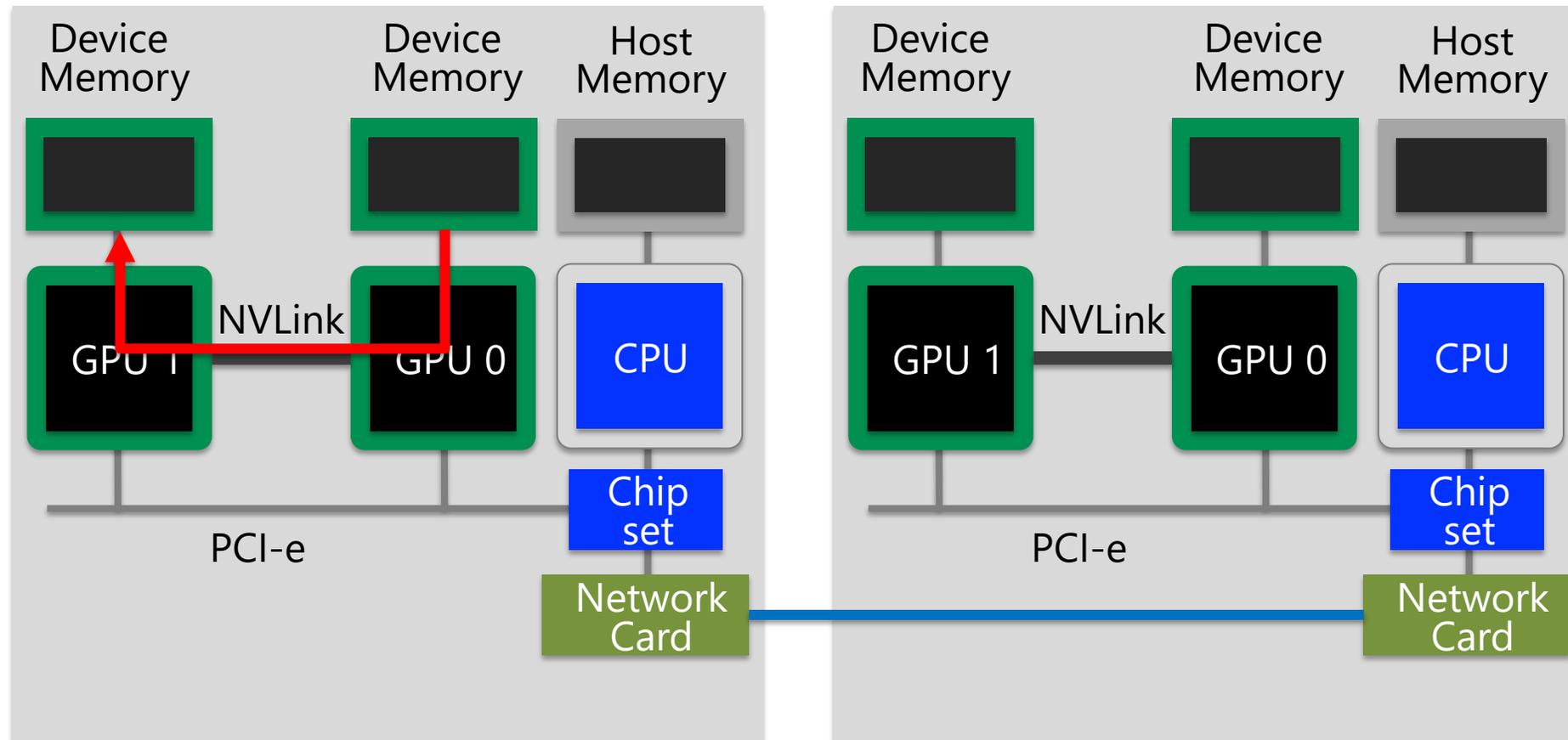
- ホストメモリ経由のノード内GPU間通信
  - ✓ デバイスメモリとホストメモリの間のPCI-eを通る。



# ノード内GPU間通信 (2)

## ■ NVLink経由のGPU間通信

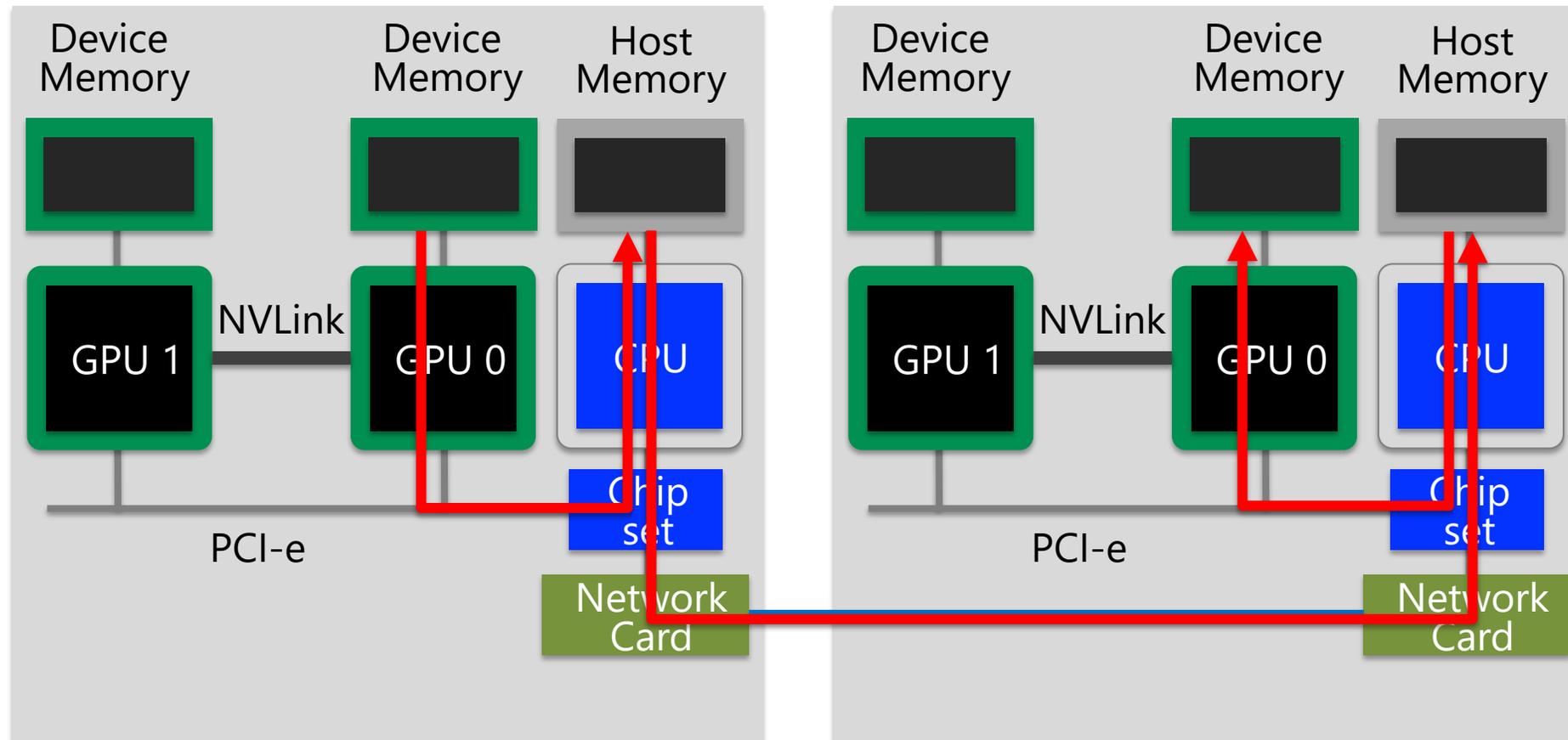
- ✓ CUDA Inter Process Communication (IPC) を利用し、同じノード内にあるプロセスではホストを経由せずGPU間で直接通信できる。最近のサーバではGPU間をつなぐNVLink経由で高速通信する。



# 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

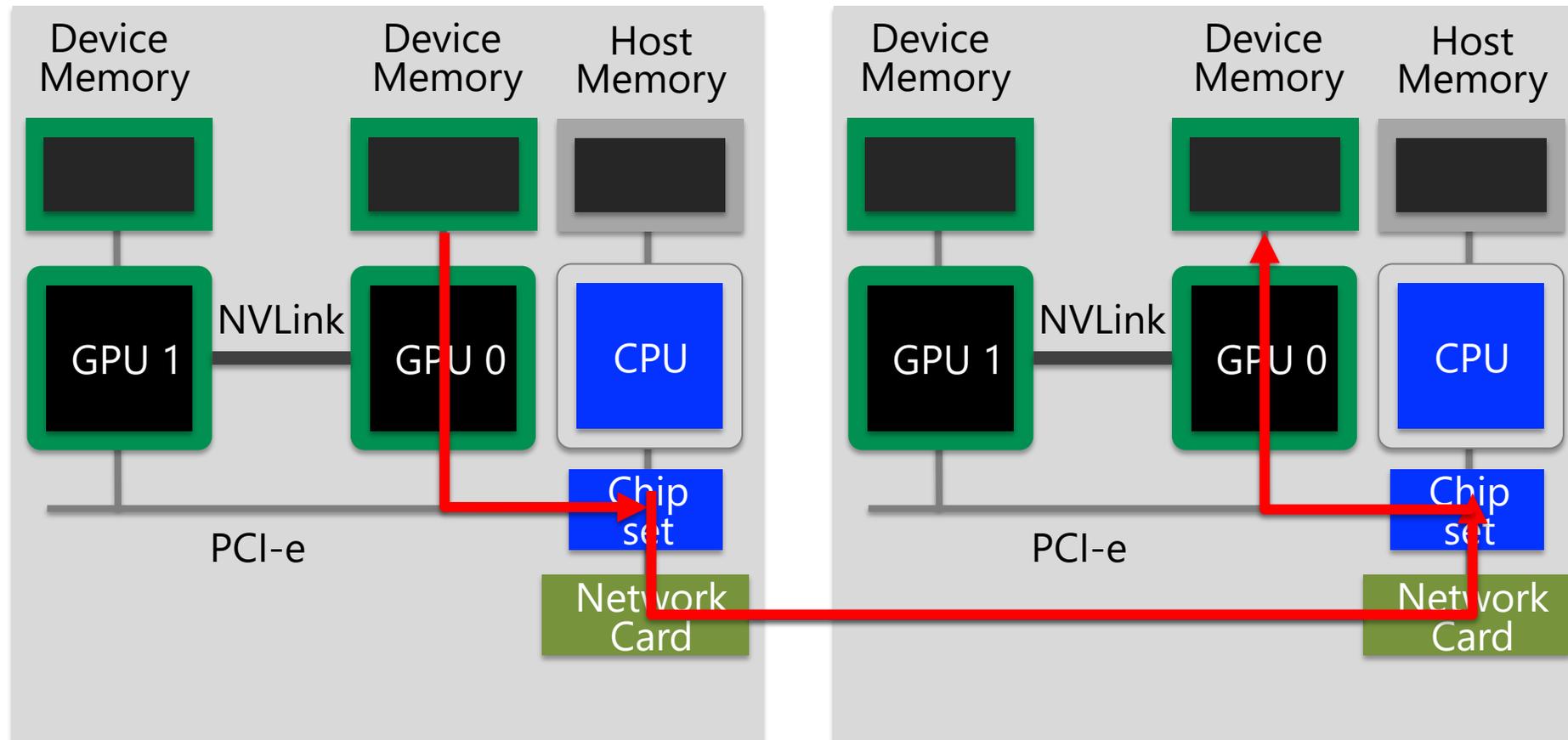
- ✓ あるノードのGPU0から別ノードのGPU0へ転送する際に、それぞれのホストメモリを経由して転送する。



# 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。
- ✓ デバイスマ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 (NVIDIA GPU向けの開発環境) とMPIによる複数GPU計算では、しばしばデバイスマ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マ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、デバイスマ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 cuda ompi-cuda  
$ mpicc -O3 -acc -Minfo=accel -ta=tesla,cc80 -c main.c
```

**-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。

**-ta=tesla,cc80:**

ターゲット・アーキテクチャの指定。NVIDIA GPU Teslaをターゲットとし、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 cuda ompi-cuda  
$ 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  $a = 3.0 * ny$

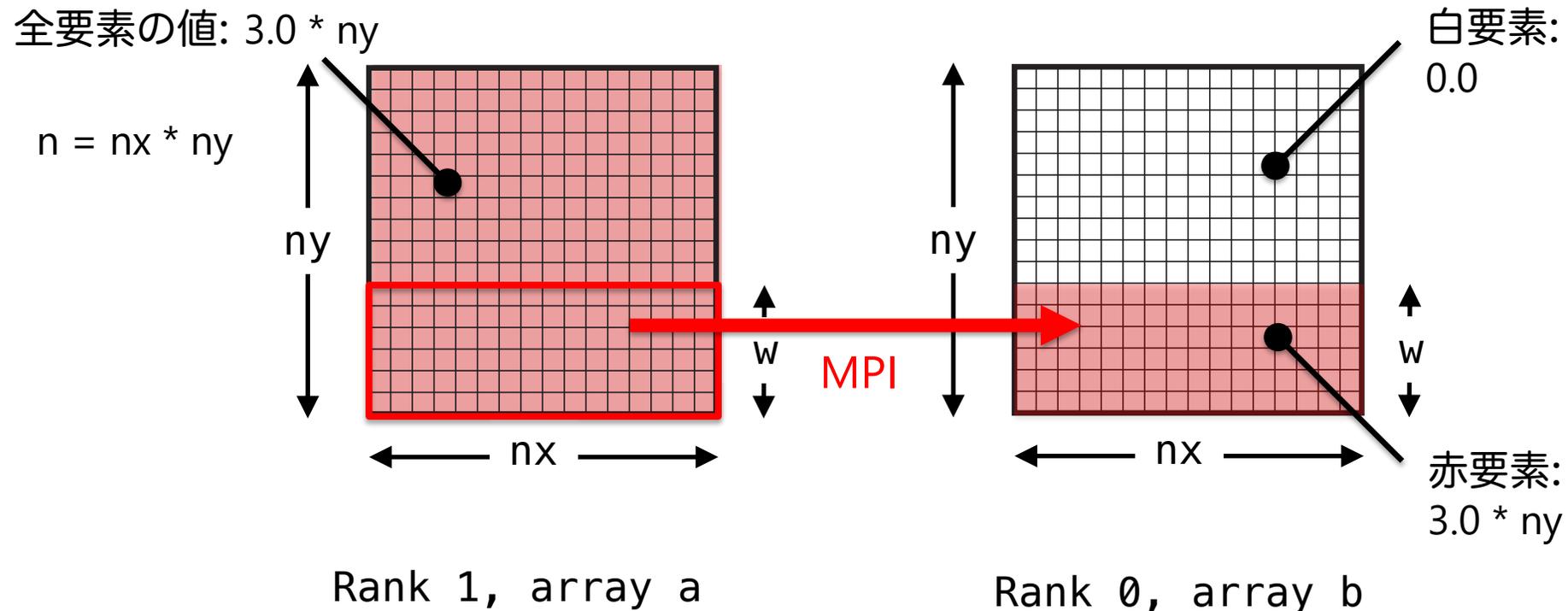
rank1の a から rank0 の b へ  
 $w * nx = 10nx$  が転送される。

$$\begin{aligned} \text{sum}/n &= (3.0 * ny) * (w * nx) / (nx * ny) \\ &= 3.0 * w \\ &= 30.0 \end{aligned}$$

# 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  $b \wedge w * nx = 10nx$  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

---

<code>openacc_mpi_basic/01_original</code>	MPI並列化されたCPUコード。
<code>openacc_mpi_basic/02_kernels</code>	MPI+OpenACCコード。上に <code>kernels/loop</code> 指示文を追加。単一ノード内2GPU使用。
<code>openacc_mpi_basic/03_update</code>	MPI+OpenACCコード。上に <code>update</code> 指示文を追加。単一ノード内2GPU使用。
<code>openacc_mpi_basic/04_cuda_aware</code>	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。
<code>openacc_mpi_basic/05_no_gdr</code>	(参考)MPI+OpenACCコード。上を2ノードにある2GPU使用。
<code>openacc_mpi_basic/06_gdr</code>	(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)

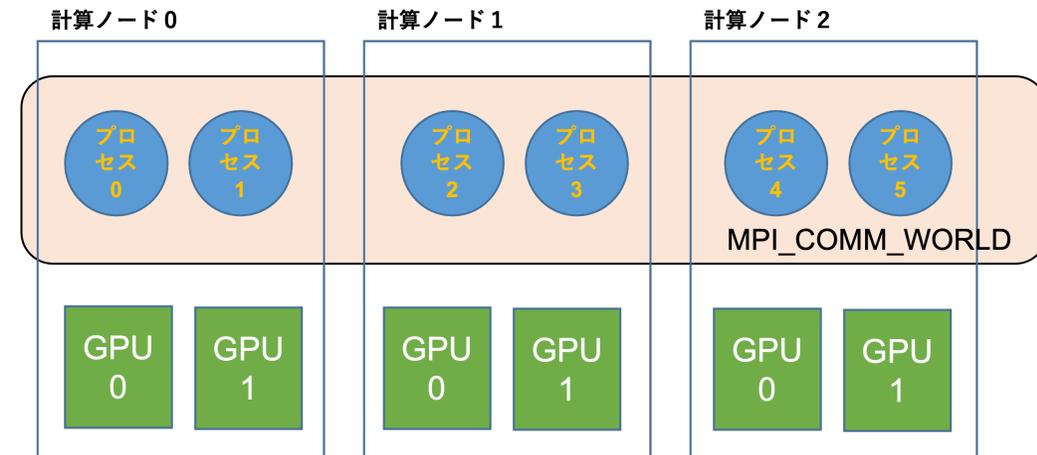
```
const int ngpus = acc_get_num_devices(acc_device_nvidia);
```

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。



# 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。

```
#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;  
}
```

openacc\_mpi\_basic/02\_kernels  
allocate

D->H, deallocate

```
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);
```

```
}
```

MPI(H)

MPI(H)

```
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, H->D

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(H->D), memcpy(D->H) 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(D->H)
  - device
    - ✓ memcpy(H->D)
- update 指示文の 使い方例

```
...  
#pragma acc data copy(a[0:nx * ny])           allocate, H->D  
{  
...  
#pragma acc update host(a[0:nx])              D->H  
    host_func(a, nx)  
#pragma acc update device(a[0:nx])           H->D  
...  
}  
...  
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])
{
  #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);
  #pragma acc update device(b[0:w * nx])
  } else {
  #pragma acc update host(a[0:w * nx])
    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];
  }
}
```

allocate

MPI(H)  
H->D  
D->H  
MPI(H)

MPI通信前後に挿入

dealloc

# update指示文追加コード (2)

---

## ■ 通信方法

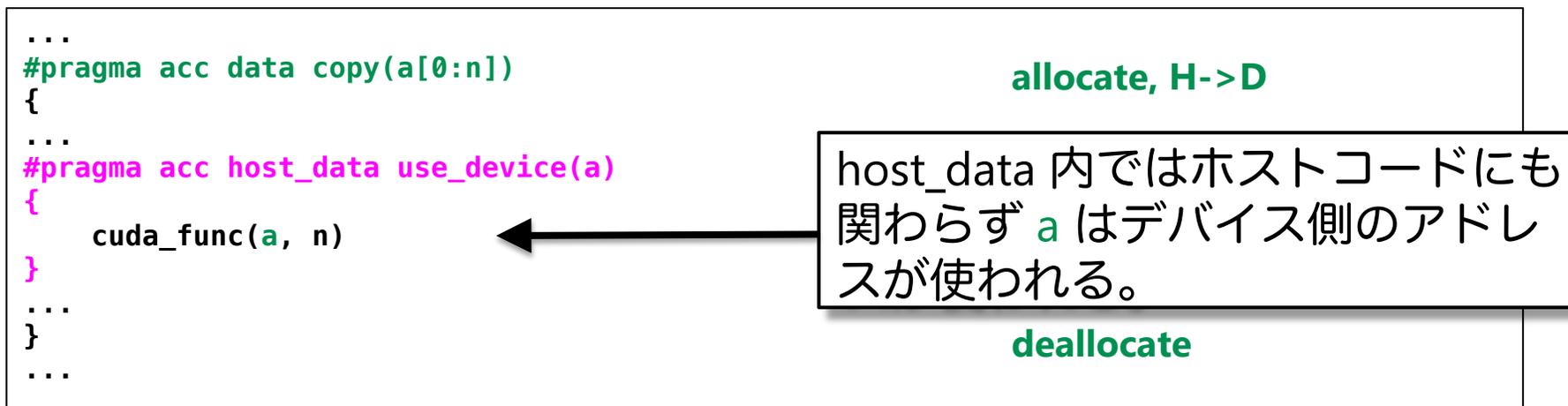
- ✓ MPI前後で配列の必要な領域のみを update 指示文でデバイスメモリとホストメモリ間でコピー
- ✓ 02\_kernels と比較し、メモリの確保・解放とデータのコピー量が削減されている。
- ✓ ホストメモリ経由のノード内GPU間通信

## ■ 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)



# 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 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(D)
    MPI_Recv(b, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD, &status);
  } else {
    #pragma acc host_data use_device(a) MPI(D)
    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];
  }
} 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

mpixexec -machinefile $PJM_0_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。

**mpixexec -np -machinefile \$PJM\_0\_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。デフォルトでは、ホストメモリ経由のノード間GPU間通信となる。

## ■ 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립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。

```
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;

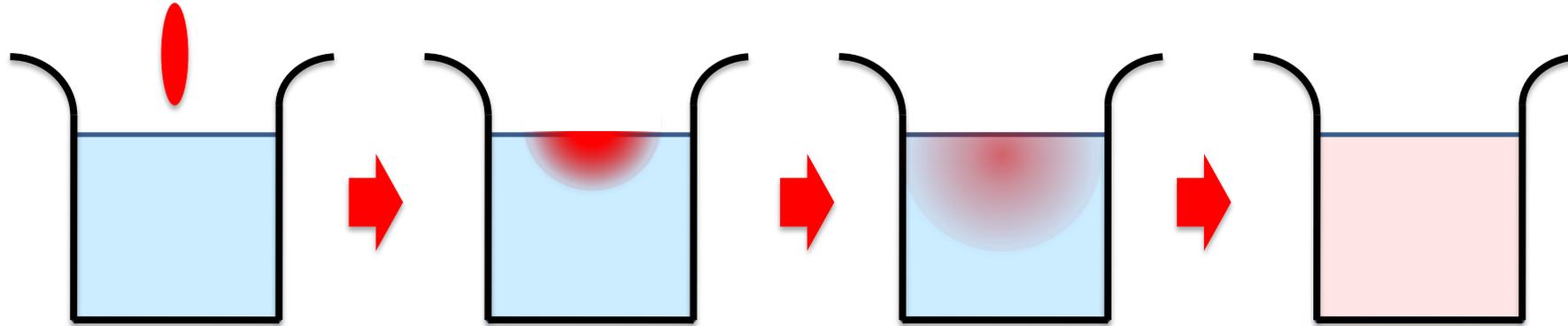
            fn[ix] = cc*f[ix]
                + ce*f[ip] + cw*f[im]
                + cn*f[jp] + cs*f[jm]
                + ct*f[kp] + cb*f[km];
        }
    }
}
```

diffusion.c, diffusion3d 関数内

# 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。



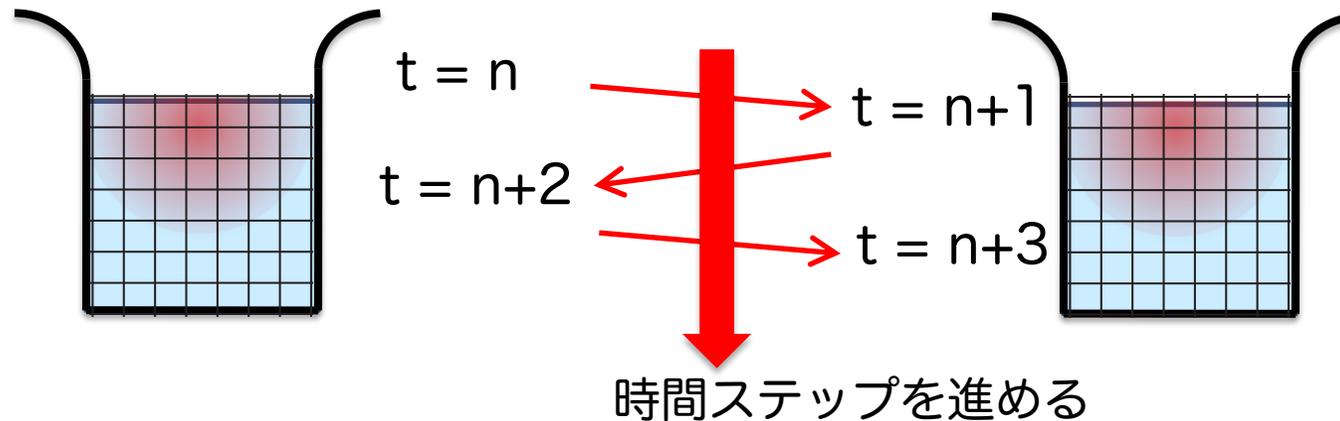
## ■ 拡散方程式のシミュレーション

- ✓ 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:  $n_x * n_y * n_z$  (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

1	0	0	1	0	0
2	0	2	8	2	0
<b>j</b> 3	1	8	20	8	1
4	0	2	8	2	0
5	0	0	1	0	0
	1	2	<b>i</b> 3	4	5

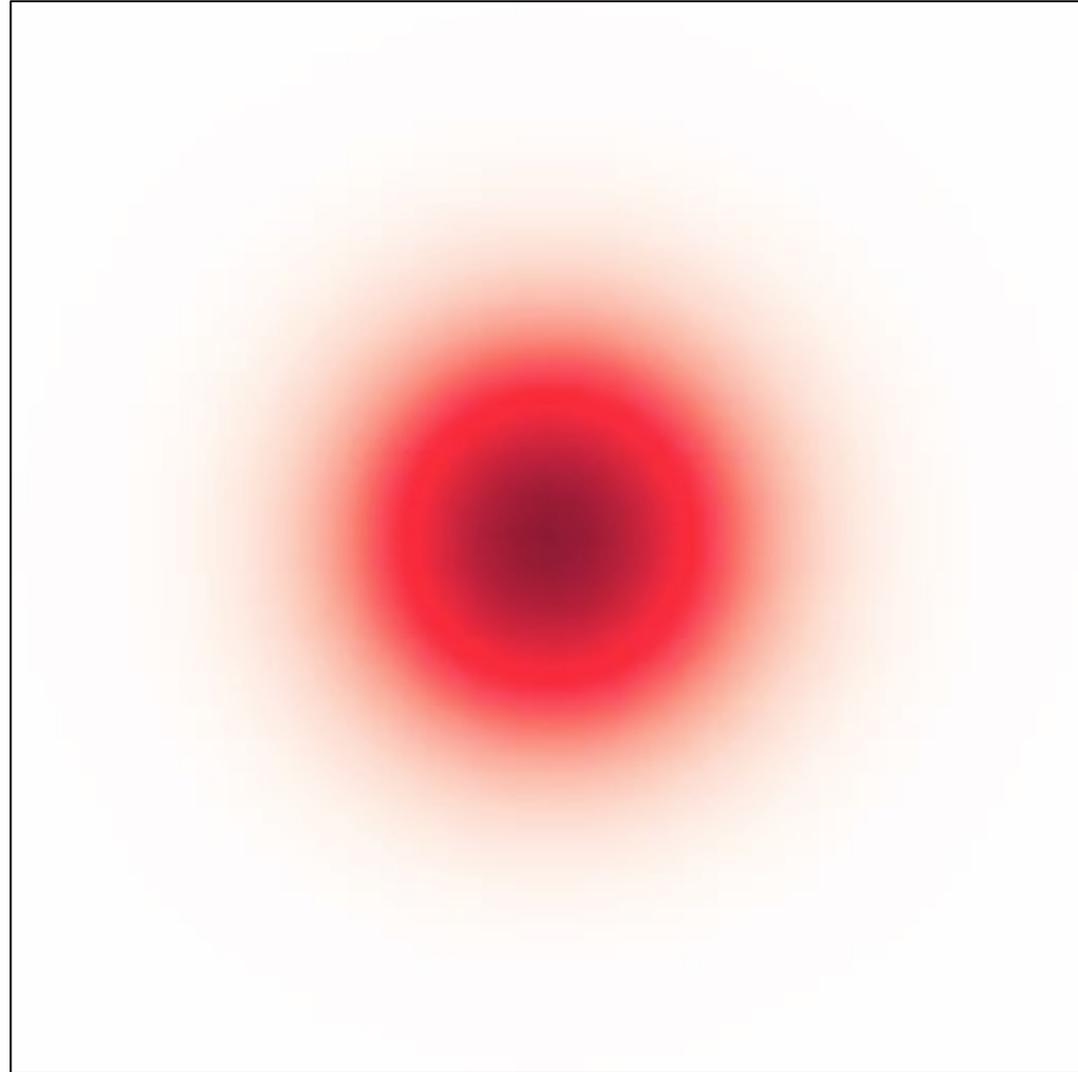
2回目の平均後

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)

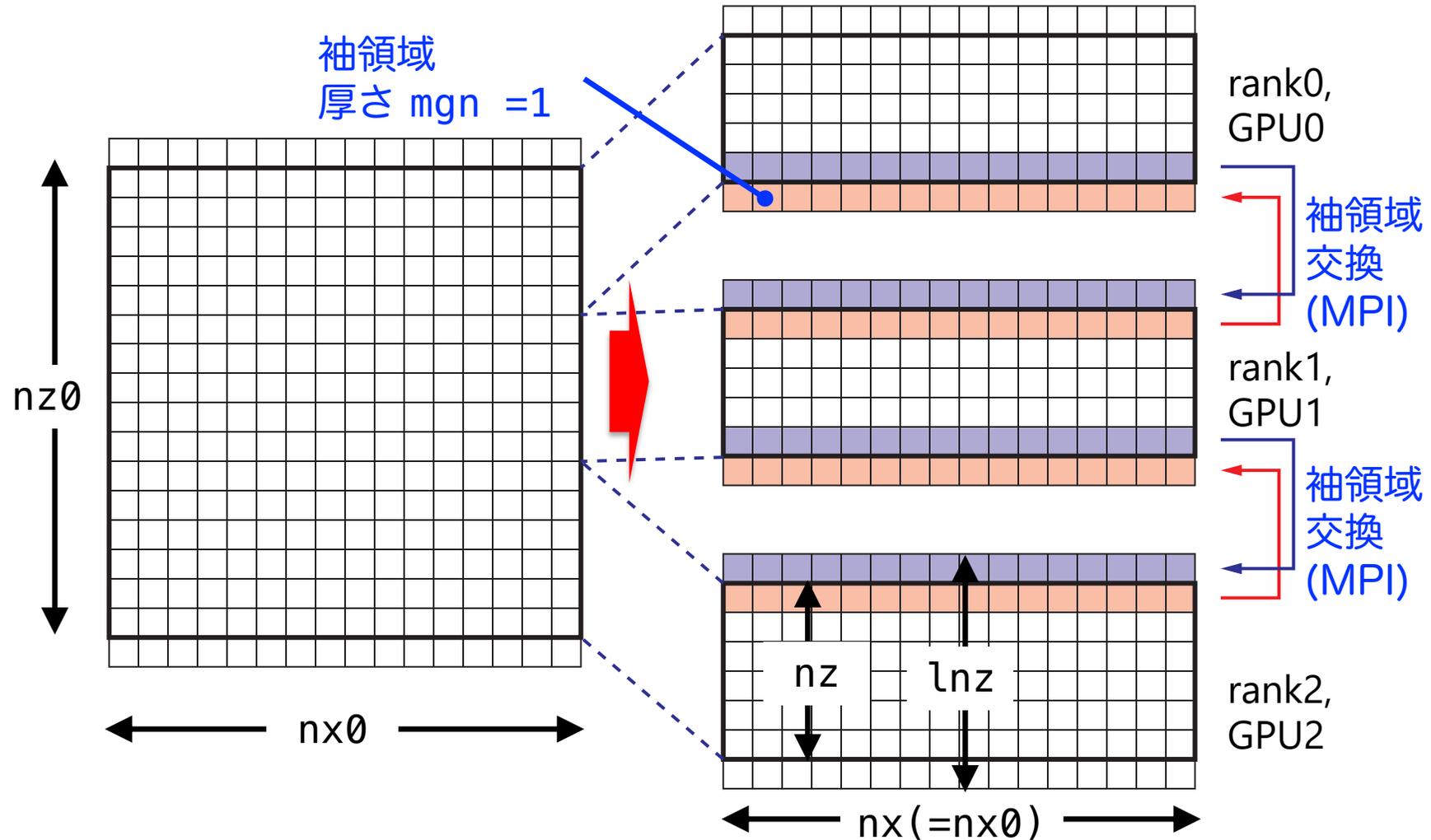
---

- 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。

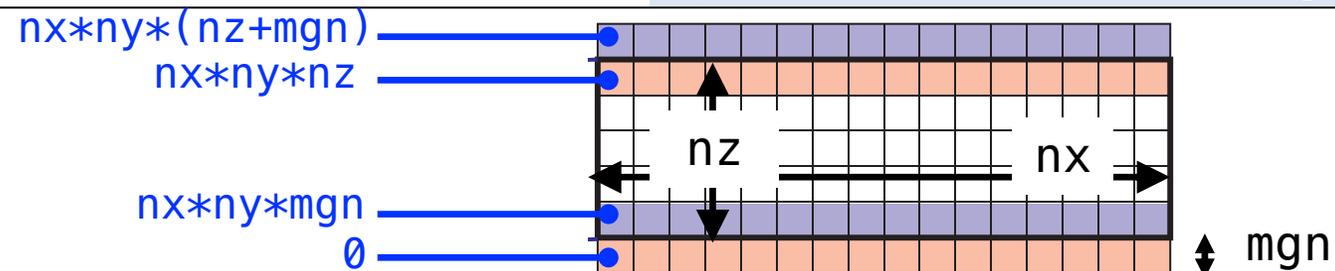


# 時間発展

- 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。

```
for (; icnt<nt && time + 0.5*dt < 0.1; icnt++) {  
    if (rank == 0 && icnt % 100 == 0) main.c, main 関数内  
        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 -ta=tesla,cc60
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。

```
#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;
}
```

diffusion.c, diffusion3d 関数内

# OpenACC化(3): data, host\_data

## ■ main関数で data, host\_data を追加

```
#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();
}
```

main.c, main 関数内

# 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。

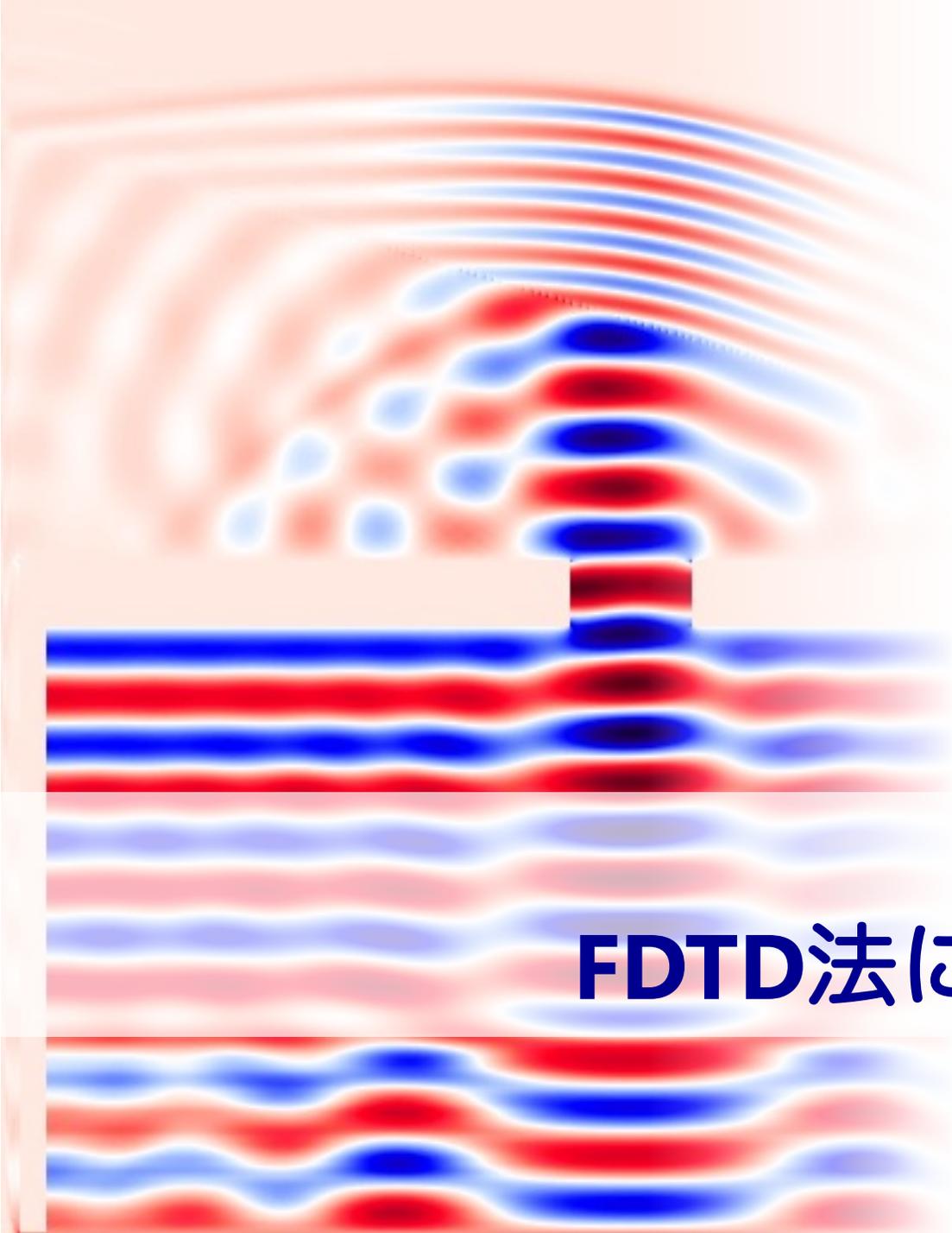
```
$ cat run.sh.?????.err
Accelerator Kernel Timing data
/work/01/gt00/z30108/openacc_samples_test/openacc_samples/C/openacc_diffusion/03_openacc_pgi_acc_time/main.c
main NVIDIA devicenum=0
time(us): 725
39: data region reached 2 times ← データ移動の回数
39: data copyin transfers: 1
device time(us): total=342 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_diffusion/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 ← 起動したスレッド
grid: [16384] block: [128]
elapsed time(us): total=67,084 max=54 min=40 avg=40 ←
17: data region reached 3276 times
```

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よう。
- 条件を変更して性能測定してみましよう。
  - ✓ 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`



# 複数GPUを用いた FDTD法による電磁波伝搬計算

# 電磁波の方程式

## ■ 真空での電場Eと磁場Hの時間発展

Maxwell 方程式の一部

$$\frac{\partial \mathbf{E}}{\partial t} = \frac{1}{\varepsilon} \nabla \times \mathbf{H} \qquad \frac{\partial \mathbf{H}}{\partial t} = -\frac{1}{\mu} \nabla \times \mathbf{E}$$

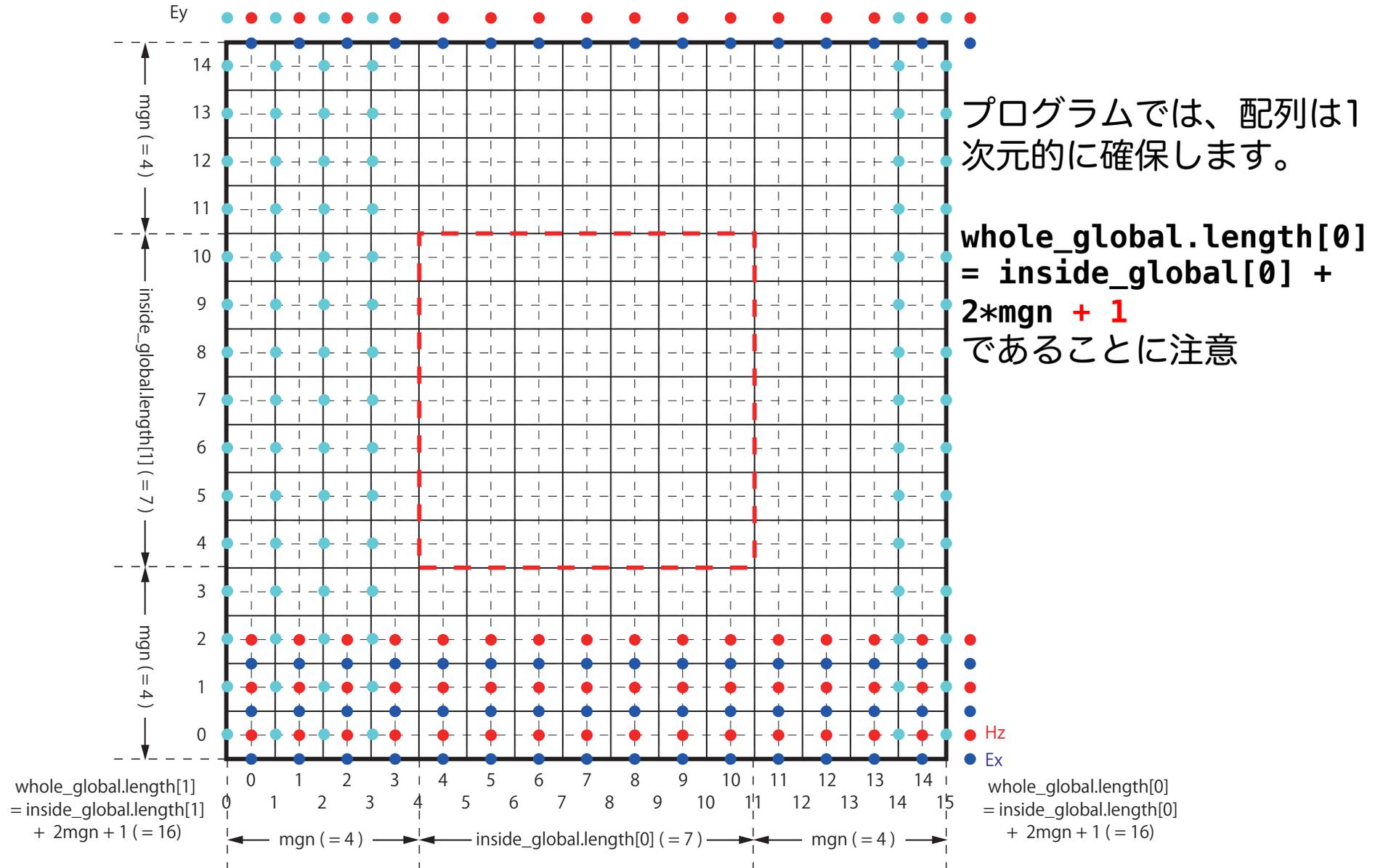
( $\varepsilon$  : 誘電率)

( $\mu$  : 透磁率)

この方程を、2次元FDTD法 (Finite-difference time-domain 法) \*を用いて解いて行きます。

\* K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," IEEE Trans. on Antennas and Propagat., vol. 14, pp. 302-307, May 1966.

# 2次元FDTD法の変数配置



# FDTD法 (1)

## ■ EとHの時間発展

$$\frac{\mathbf{E}^n - \mathbf{E}^{n-1}}{\Delta t} = \frac{1}{\varepsilon} \nabla \times \mathbf{H}^{n-\frac{1}{2}}$$

$$\frac{\mathbf{H}^{n+\frac{1}{2}} - \mathbf{H}^{n-\frac{1}{2}}}{\Delta t} = -\frac{1}{\mu} \nabla \times \mathbf{E}^n$$

変形して、

$$\mathbf{E}^n = \mathbf{E}^{n-1} + \frac{\Delta t}{\varepsilon} \nabla \times \mathbf{H}^{n-\frac{1}{2}}$$

$$\mathbf{H}^{n+\frac{1}{2}} = \mathbf{H}^{n-\frac{1}{2}} - \frac{\Delta t}{\mu} \nabla \times \mathbf{E}^n$$

# FDTD法 (2)

## ■ 例えば、

$$E_x^n(i + \frac{1}{2}, j) = E_x^{n-1}(i + \frac{1}{2}, j) + \frac{\Delta t}{\varepsilon(i + \frac{1}{2}, j)} \left( \frac{H_z^{n-\frac{1}{2}}(i + \frac{1}{2}, j + \frac{1}{2}) - H_z^{n-\frac{1}{2}}(i + \frac{1}{2}, j - \frac{1}{2})}{\Delta y} \right)$$

$$H_z^{n+\frac{1}{2}}(i + \frac{1}{2}, j + \frac{1}{2}) = H_z^{n-\frac{1}{2}}(i + \frac{1}{2}, j + \frac{1}{2}) - \frac{\Delta t}{\mu(i + \frac{1}{2}, j + \frac{1}{2})} \left( \frac{E_y^n(i + 1, j + \frac{1}{2}) - E_y^n(i, j + \frac{1}{2})}{\Delta x} - \frac{E_x^n(i + \frac{1}{2}, j + 1) - E_x^n(i + \frac{1}{2}, j)}{\Delta y} \right)$$

# ソースコード (1)

---

- サンプルコード: openacc\_mpi\_fdttd/
  - ✓ OpenACCとMPIを利用したFDTD法 (電磁波解析)

---

openacc_mpi_fdttd/01_original	MPI並列化されたCPUコード。
openacc_mpi_fdttd/02_openacc1	calc_ex_ey, pml_boundary_ex, pml_boundary_ey, がOpenACC。
openacc_mpi_fdttd/03_openacc2	時間更新ループ全体が OpenACC。
openacc_mpi_fdttd/04_openacc3	初期化を含め OpenACC。
openacc_mpi_fdttd/05_openacc4	データ移動の最適化。

---

# ソースコード (2)

---

## ■ それぞれのファイルの内容

---

main.c	プログラムのメインコード
fdtd2d.{c, h}	2次元 FDTD の 計算コード
fdtd2d_sources.{c, h}	入射光設定のための関数
setup.c	計算条件の設定と変数の初期化
config.{c, h}	物理定数の定義
output.{cc, h}	計算結果出力のための関数
bitmap*	BMPファイル作成のための関数

---

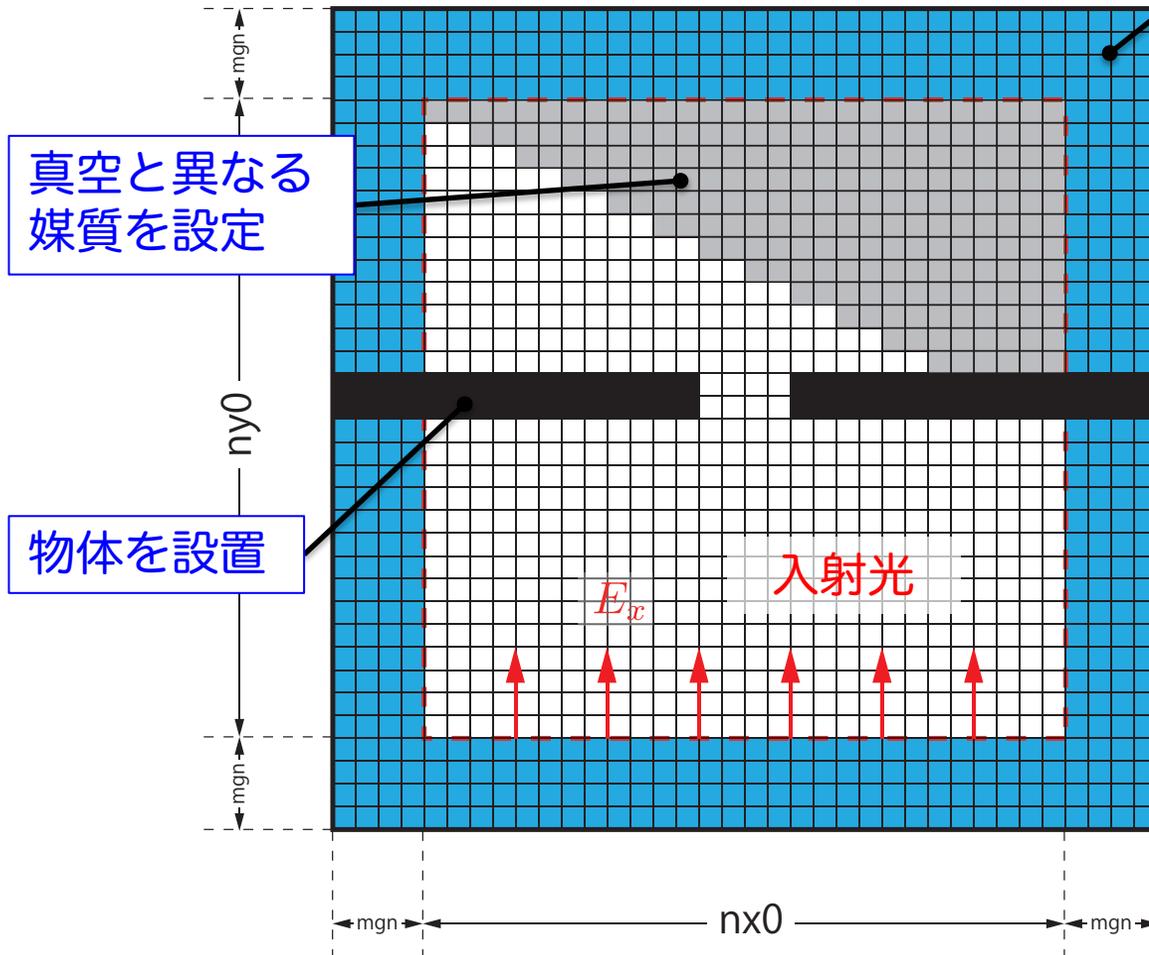
本講習では、“main.c”、“fdtd2d.c”、“fdtd2d\_sources.c”、“setup.c” のソースコードを追記・修正していきます。

# 計算条件

## ■ 2次元波動伝搬

- ✓ 成分:  $E_x$ 、 $E_y$ 、 $H_z$
- ✓  $y$  方向下側から平面波を入射

電磁波の境界での非物理的な反射を防ぐための吸収境界条件 (PML)



真空と異なる  
媒質を設定

物体を設置

$$dx = lx/nx$$
$$dy = ly/ny$$

デフォルト設定:

$$nx = 512$$

$$ny = 512$$

$$mgn = 8$$

$$lx = 529$$

$$ly = 529$$

プログラム中では下記の変数が使われているので注意

$$\text{inside\_global.length}[0] = nx0$$

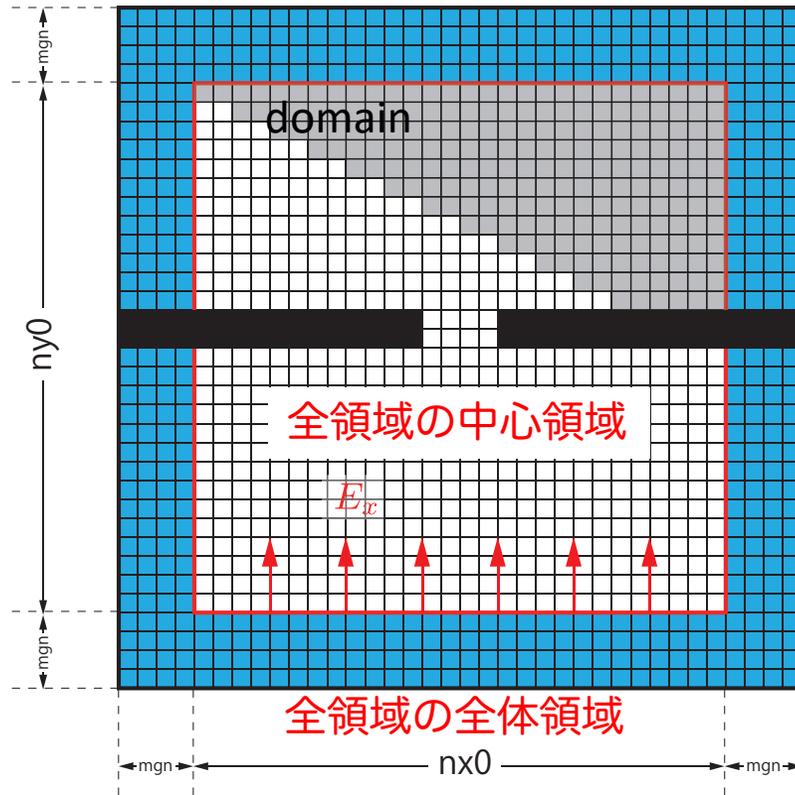
$$\text{inside\_global.length}[1] = ny0$$

$$\text{whole\_global.length}[0] = nx0 + 2*mgn + 1$$

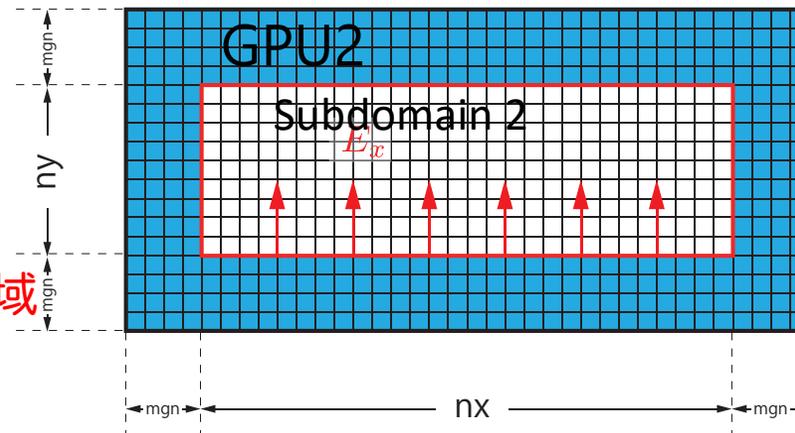
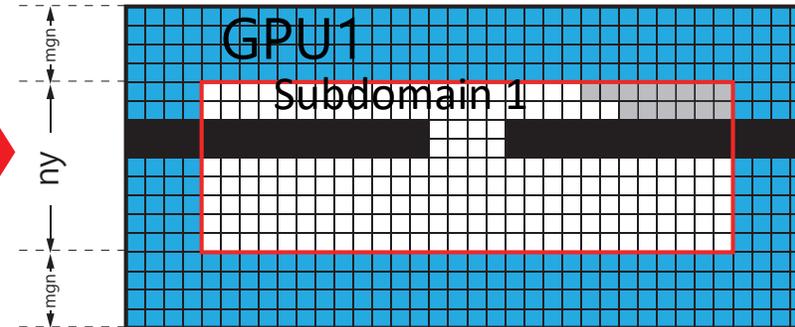
$$\text{whole\_global.length}[1] = ny0 + 2*mgn + 1$$

# マルチGPU化のための領域分割

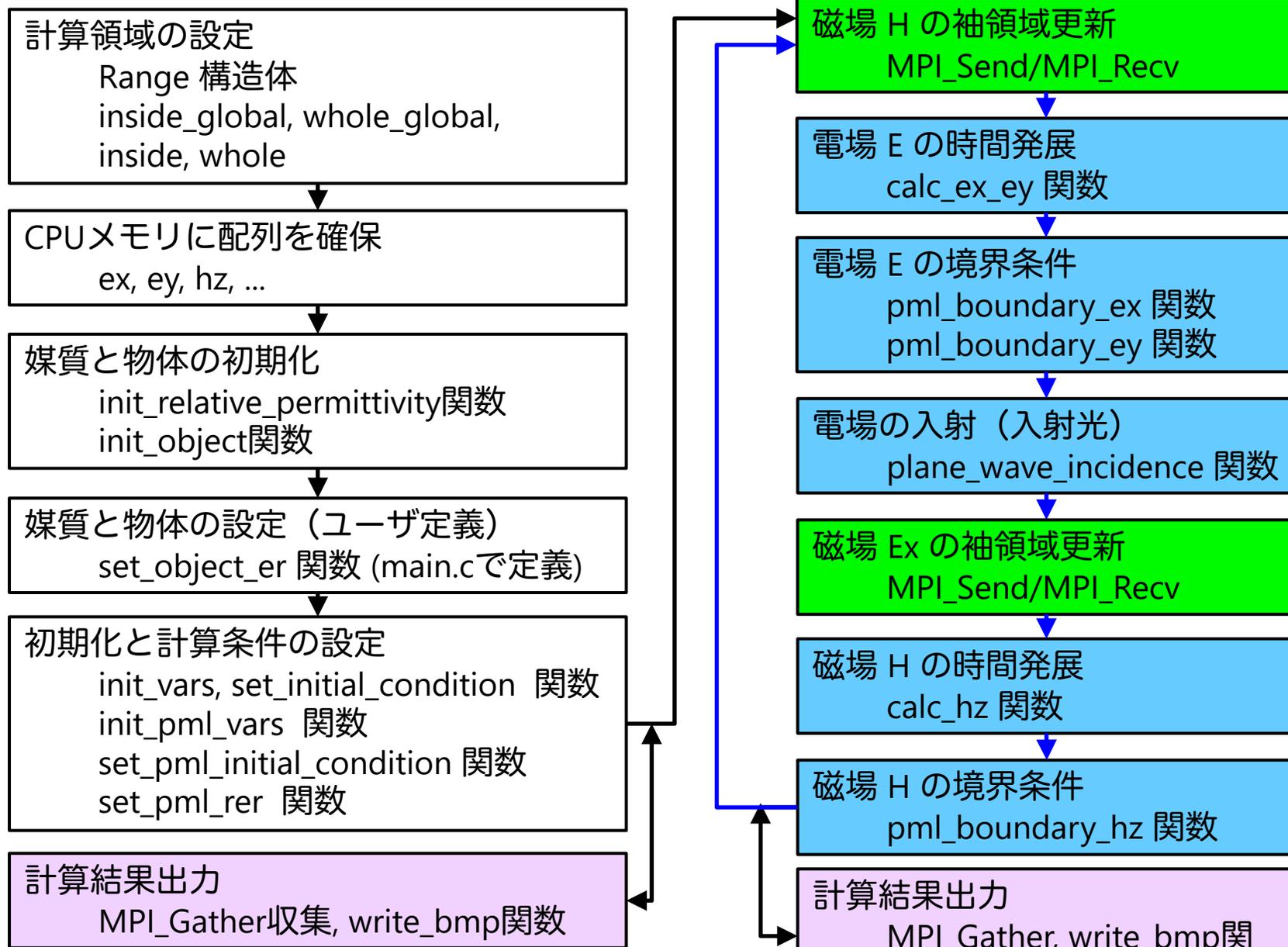
y 方向で全領域を分割  
各分割領域に袖領域を持たせます。



全体領域：袖領域（PML、青格子）を含む領域  
中心領域：袖領域を含まない領域



# コード全体の流れ (main.c 内)



FDTD計算  
繰り返し時間発展

# 計算領域の設定 (1)

## ■ Range 構造体

- ✓ 計算領域の始点と大きさを保持

```
// config.h
struct Range {
    int length[2];
    int begin [2];
};

// main.c
const struct Range inside_global = { { atoi(argv[1]), atoi(argv[2]) },      全領域の中心領域
                                     { 0, 0 } };
const struct Range whole_global  = { { inside_global.length[0] + 2*mgn + 1,  全領域の
                                     inside_global.length[1] + 2*mgn + 1},    全体領域
                                     { inside_global.begin[0] - mgn           ,
                                     inside_global.begin[1] - mgn           } };

const struct Range inside        = { { inside_global.length[0],              分割領域の
                                     inside_global.length[1]/nsubdomains },    中心領域
                                     { 0,
                                     inside_global.length[1]/nsubdomains * rank } };
const struct Range whole         = { { inside.length[0] + 2*mgn + 1,
                                     inside.length[1] + 2*mgn + 1},          分割領域の
                                     { inside.begin[0] - mgn                   , 全体領域
                                     inside.begin[1] - mgn                   } };
```

# 計算領域の設定 (2)

## ■ Range 構造体

- ✓ 計算領域の始点と大きさを保持

```
struct Range {  
    int length[2];  
    int begin [2];  
};  
  
const struct Range inside = { { inside_global.length[0],  
                                inside_global.length[1]/nsubdomains },  
                                { 0,  
                                inside_global.length[1]/nsubdomains * rank } };  
  
const struct Range whole = { { inside.length[0] + 2*mgn + 1,  
                                inside.length[1] + 2*mgn + 1},  
                                { inside.begin[0] - mgn      ,  
                                inside.begin[1] - mgn      } };
```

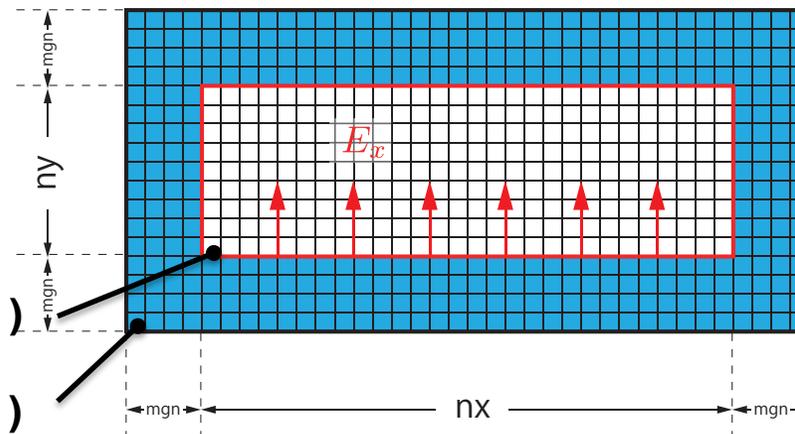
プログラムでは  
下記の通り

```
inside.length[0] = nx  
inside.length[1] = ny
```

```
whole.length[0] = nx + 2*mgn + 1  
whole.length[1] = ny + 2*mgn + 1
```

座標(inside.begin[0], inside.begin[1])

座標(whole.begin[0], whole.begin[1])



# 配列の確保

## ■ 物理変数配列は main.c で確保

```
// main.c
const int    nelems      = whole.length[0] * whole.length[1];
const int    nelems_x    = whole.length[0];
const int    nelems_y    = whole.length[1];
const size_t size        = sizeof(FLOAT)*nelems;
const size_t size_x      = sizeof(FLOAT)*nelems_x;
const size_t size_y      = sizeof(FLOAT)*nelems_y;
const size_t size_global = sizeof(FLOAT)* whole_global.length[0] * whole_global.length[1];

FLOAT *ex      = (FLOAT *)malloc(size); // 電場 Ex
FLOAT *ey      = (FLOAT *)malloc(size); // 電場 Ey
FLOAT *hz      = (FLOAT *)malloc(size); // 磁場 Hz
...
// For output
FLOAT *ex_global = (FLOAT *)malloc(size_global);
FLOAT *ey_global = (FLOAT *)malloc(size_global);
FLOAT *hz_global = (FLOAT *)malloc(size_global);
```

- 多くの配列は `whole.length[0] * whole.length[1]`
- `ex_global, ey_global, hz_global` はファイル出力に使うため、`whole_global.length[0] * whole_global.length[1]`

# 計算結果出力

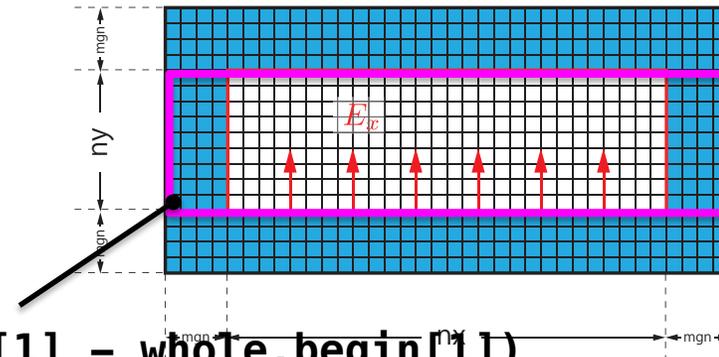
- 各ランクの  $ex$  を  $ex\_global$  へMPI\_Gatherで収集
  - ✓  $ey, hz$  も同様

```
// main.c
const int rank_root = 0;
const int sendnelems = whole.length[0] * inside.length[1];
const int src = whole.length[0] * (inside.begin[1] - whole.begin[1]);
const int dst = whole.length[0] * (inside.begin[1] - whole.begin[1]);

MPI_Gather(&ex[src], sendnelems, MPI_FLOAT_T, &ex_global[dst],
          sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
MPI_Gather(&ey[src], sendnelems, MPI_FLOAT_T, &ey_global[dst],
          sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
MPI_Gather(&hz[src], sendnelems, MPI_FLOAT_T, &hz_global[dst],
          sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);

if (rank == rank_root) {
    write_bmp(icnt, time, whole_global.length, dx, dy, ex_global, ey_global, hz_global);
}
```

袖領域を落とすため、 $y$ 方向には  
中心領域のみ切り出して、  
MPI\_Gather する



$src = whole.length[0] * (inside.begin[1] - whole.begin[1])$

# 時間発展 (1)

## ■ 前半

- ✓ hz の袖領域更新 (MPI\_Send/MPI\_Recv)、
- ✓ 電場Eの時間発展 (calc\_ex\_ey)、境界条件(pml\_boundary\_...)
- ✓ 入射光 (plane\_wave\_incidence)

```
while (icnt < nt) {  
  
    MPI_Status status;  
    const int tag = 0;  
    const int nhalo      = whole.length[0];  
    const int inside_end1 = inside.begin[1] + inside.length[1];  
  
    const int src_hz      = whole.length[0] * (inside_end1      - whole.begin[1] - 1);  
    const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);  
  
    MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD);  
    MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);  
  
    calc_ex_ey(&whole, &inside, hz, cexly, ceylx, ex, ey);  
    pml_boundary_ex(&whole, &inside, hz, cexy, cexyl, rer_ex, ex, exy);  
    pml_boundary_ey(&whole, &inside, hz, ceyx, ceysl, rer_ey, ey, eyx);  
  
    const int j_in = 0;  
    plane_wave_incidence(&whole, &inside, time, j_in, wavelength, ex, ey);  
    time += 0.5*dt;
```

(後半へ)

# 時間発展 (2)

## ■ 後半

- ✓ ex の袖領域更新 (MPI\_Send/MPI\_Recv)、
- ✓ 磁場Hの時間発展 (calc\_hz)、境界条件(pml\_boundary\_hz)

(前半から)

```
const int src_ex      = whole.length[0] * (inside.begin[1] - whole.begin[1]);
const int dst_ex      = whole.length[0] * (inside_end1      - whole.begin[1]);

MPI_Send(&ex[src_ex], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD);
MPI_Recv(&ex[dst_ex], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD, &status);

calc_hz(&whole, &inside, ey, ex, chzlx, chzly, hz);
pml_boundary_hz(&whole, &inside, ey, ex, chzx, chzxl, chzy, chzyl, hz, hzx, hzy);
time += 0.5*dt;

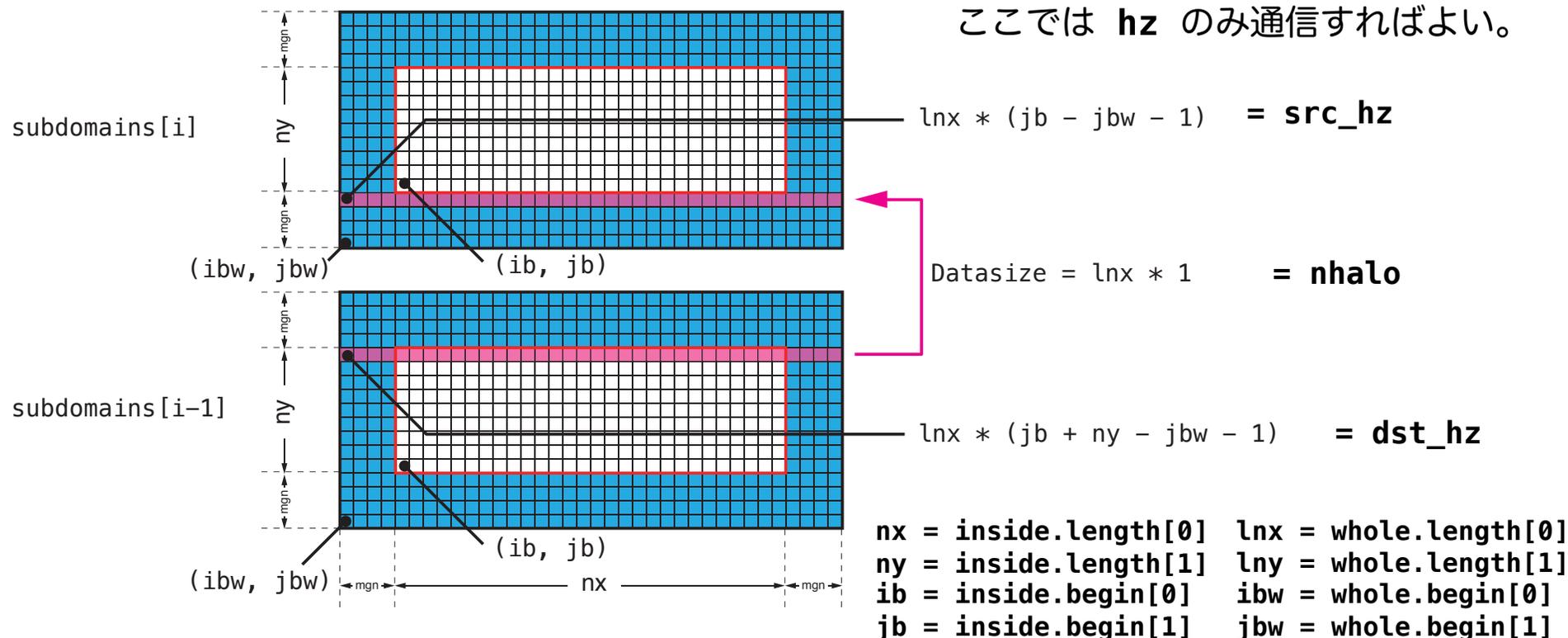
icnt++;

(出力など)
}
```

# 境界領域のデータ交換 (1)

## ■ hz の境界領域のデータ交換

データアクセスが非対称のため、  
ここでは **hz** のみ通信すればよい。



```
const int nhalo      = whole.length[0];
const int inside_end1 = inside.begin[1] + inside.length[1];

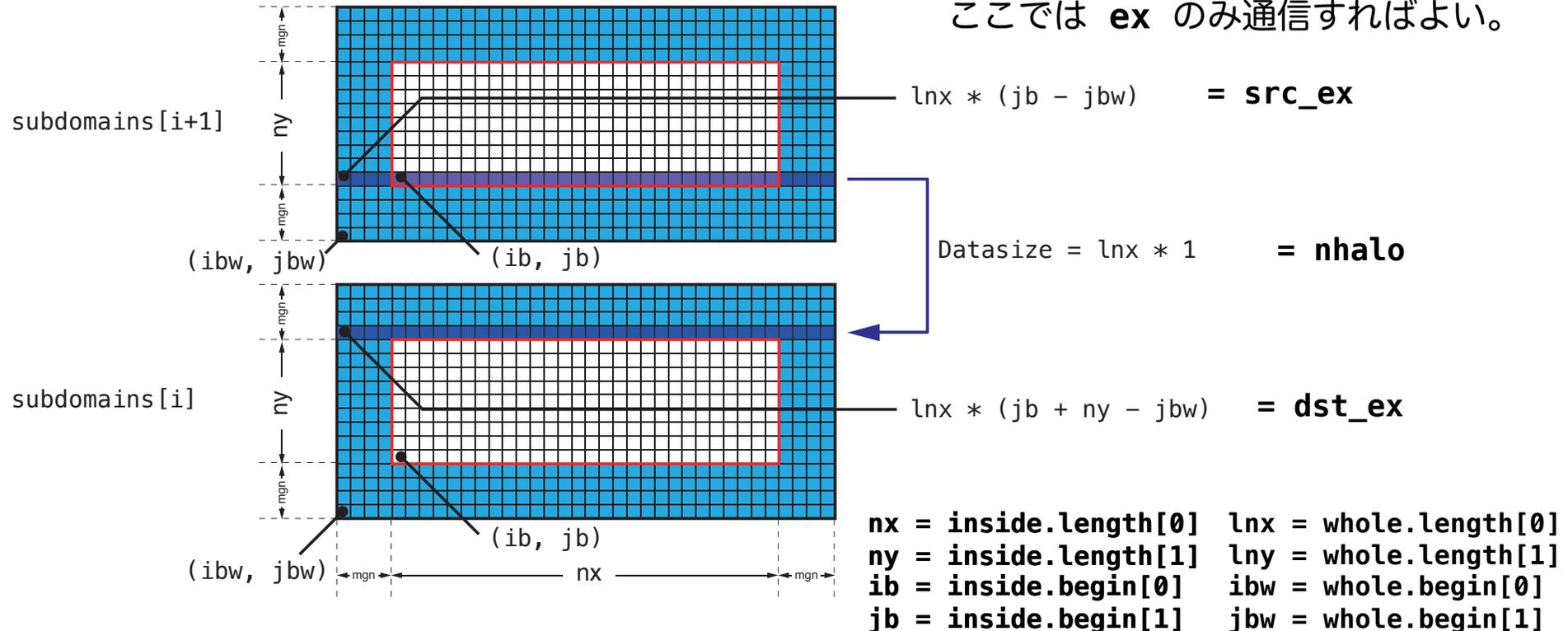
const int src_hz     = whole.length[0] * (inside_end1 - whole.begin[1] - 1);
const int dst_hz     = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up, tag, MPI_COMM_WORLD);
MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);
```

# 境界領域のデータ交換 (2)

## ■ ex の境界領域のデータ交換

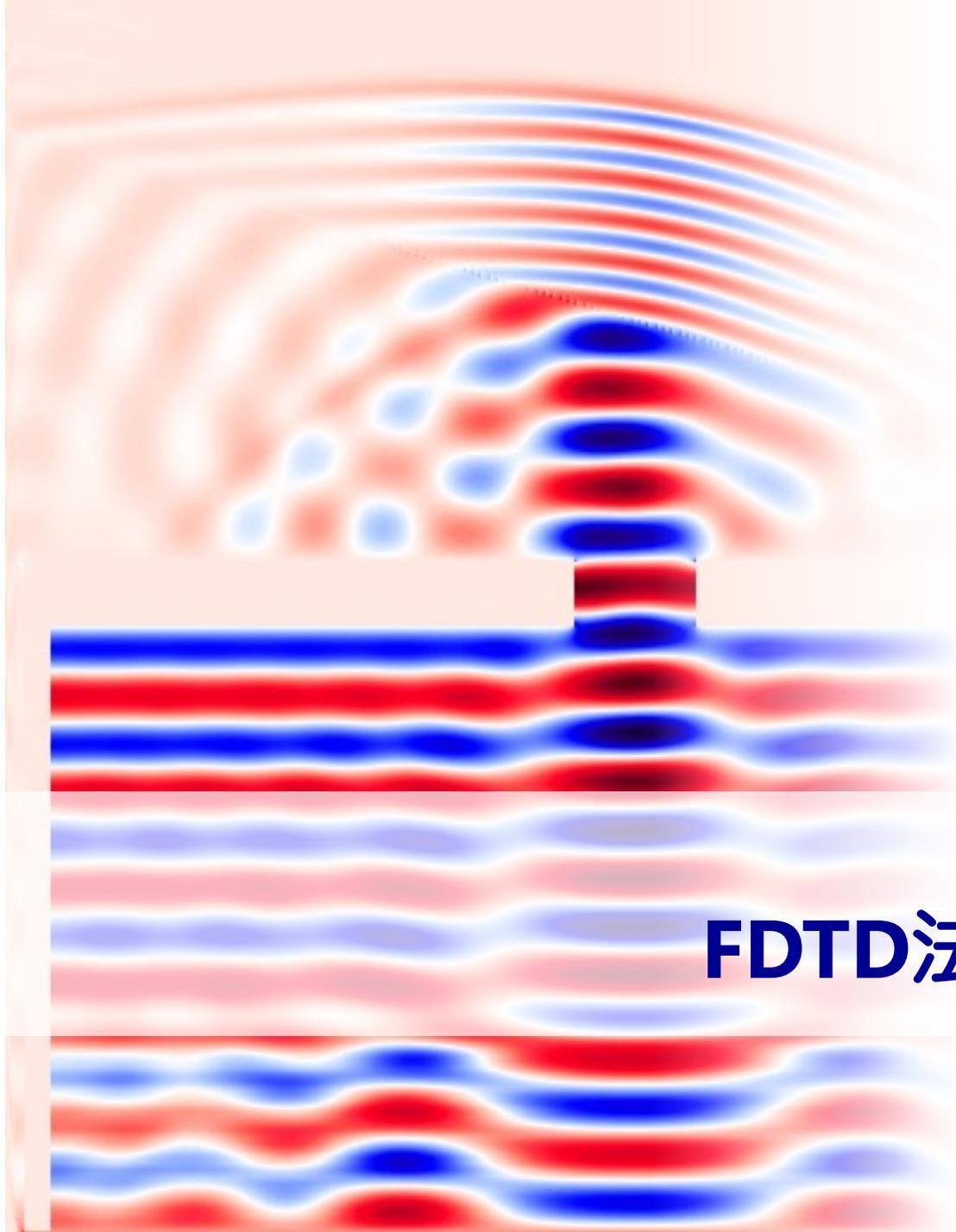
データアクセスが非対称のため、  
ここでは **ex** のみ通信すればよい。



```
const int nhalo      = whole.length[0];
const int inside_end1 = inside.begin[1] + inside.length[1];

const int src_ex     = whole.length[0] * (inside.begin[1] - whole.begin[1]);
const int dst_ex     = whole.length[0] * (inside_end1 - whole.begin[1]);

MPI_Send(&ex[src_ex], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD);
MPI_Recv(&ex[dst_ex], nhalo, MPI_FLOAT_T, rank_up, tag, MPI_COMM_WORLD, &status);
```



複数GPUを用いた  
FDTD法による電磁波伝搬計算  
の実習

# プログラムのコンパイルと実行 (1)

## ■ CPUコードのコンパイルと実行

openacc\_mpi\_fdttd/01\_original

```
$ cd openacc_mpi_fdttd/01_original
$ make
$ pjsub ./run.sh
$ cat run.sh.?????.out
Rank 0: hostname = a090
Rank 1: hostname = a090
Rank 2: hostname = a091
Rank 3: hostname = a091
Calculation condition
  nx_global      = 512
```

(省略)

```
icnt = 4900, time = 2.3115e-14 [sec]
icnt = 5000, time = 2.3587e-14 [sec]
```

```
-----
Domain      = 512 x 512
nsubdomains = 4
output_file = 1
Time        = 4.103535 [sec]
-----
```

? の数字はジョブごと  
に変わります。

利用したノード

計算領域サイズ、領域  
分割数、出力の有無、  
計算時間

なお、`pjsub ./run_no_out.sh` すると出力なしで実行する。性能測定用。

# プログラムのコンパイルと実行 (2)

## ■ プログラムの実行時オプション

```
$ cat run.sh
#!/bin/sh
#PJM -L rscgrp=lecture-a
#PJM -L gpu=2
#PJM -mpi proc=2
(省略)

mkdir -p sim_run
cd sim_run

mpiexec -machinefile $PJM_0_NODEINF -n $PJM_MPI_PROC ../run 512 512 $PJM_MPI_PROC 5000 50
```

openacc\_mpi\_fdttd/01\_original

`mpirun -np <nprocs> ../run <nx> <ny> <nprocs> <nt> <nout>`

`$PJM_MPI_PROC`: 全ランク数 (=分割数) (= --mpi proc の値)

`nx, ny`: 計算領域サイズ

`nt`: 全時間ステップ

`nout`: 出力を行うタイムステップ数。50 の場合、50ステップに1回出力する。0 を指定すると出力しない。

# 計算結果の表示

- 計算結果は `sim_run` に BMP として出力される

```
$ cd sim_run/
```

```
openacc_mpi_fdttd/01_original
```

- 計算結果の表示

- ✓ 1枚のBMPを見る

```
$ display e05000.bmp
```

- ✓ 複数のBMPファイルをアニメーションで表示

```
$ animate *.bmp
```

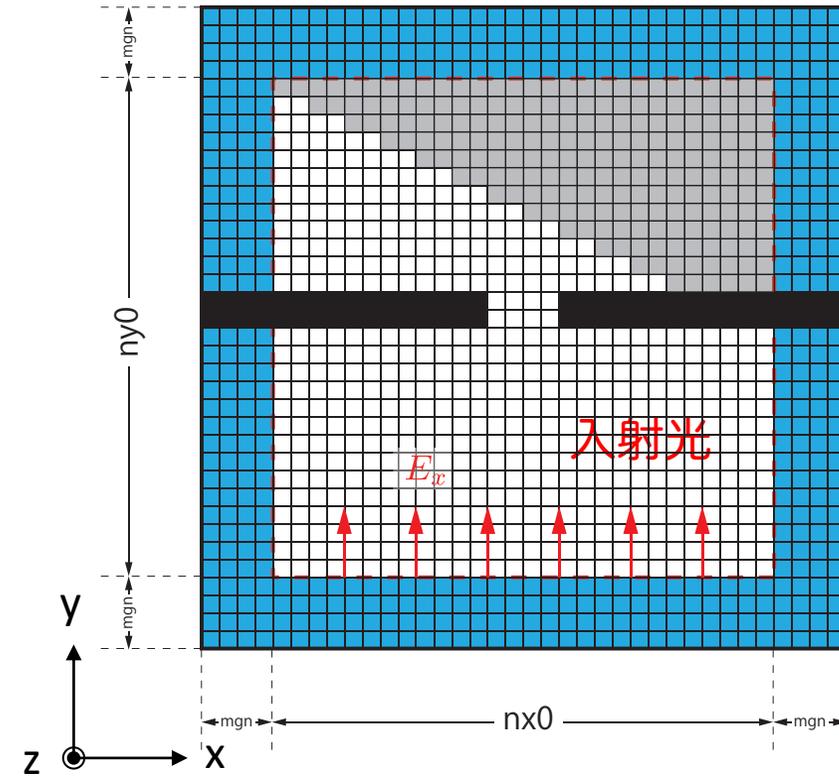
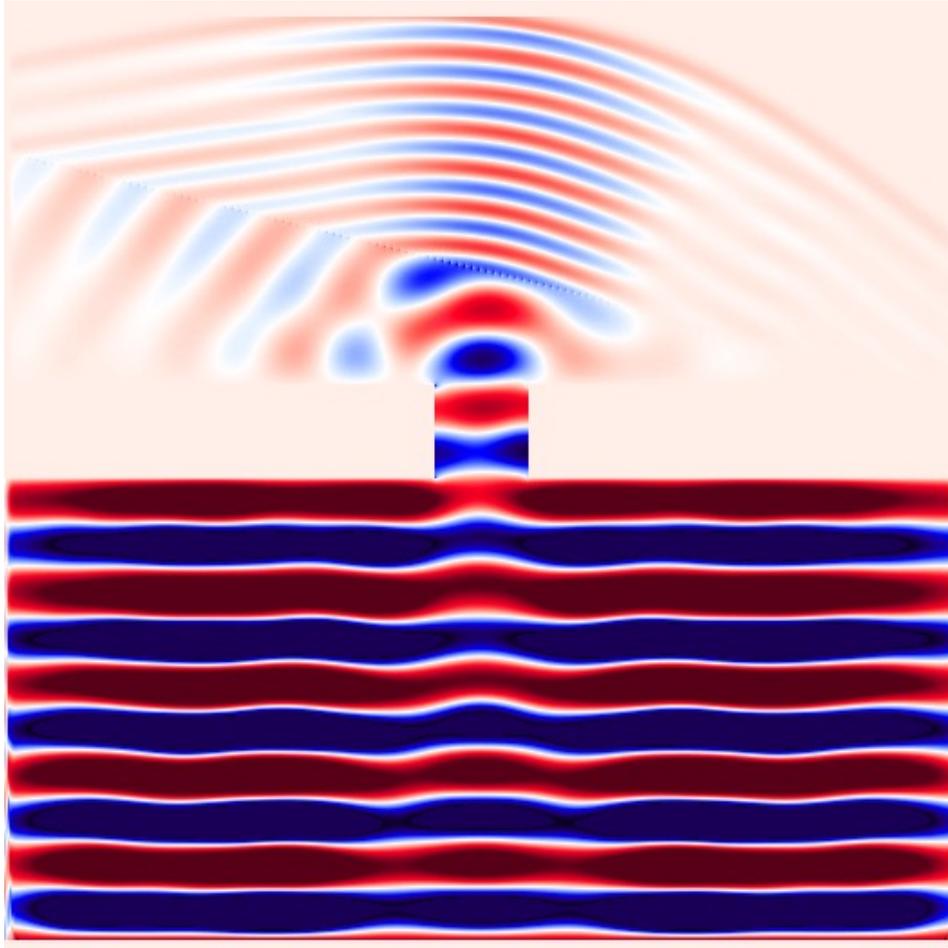
なお

```
ssh -Y txxxxx@wisteria.cc.u-tokyo.ac.jp
```

と `-Y` をつけていないと表示されない。うまく表示できない場合は画像を手元にコピーして表示してください。

# 計算結果の例

- 出力されたBMPファイルの一例
  - ✓  $E_x$  (電場の x 成分) の出力



# 実習1

- `calc_ex_ey`, `pml_boundary_ex`, `pml_boundary_ey` を OpenACC化しましょう。
- Makefile
  - ✓ コンパイルオプションの修正
- `main.c`
  - ✓ OpenACCヘッダーの追加
  - ✓ GPUの割り当て
  - ✓ `data` 指示文の追加
  - ✓ `MPI_Send/MPI_Recv`に対する`host_data`指示文の追加
- `fdd2d.c`
  - ✓ `kernels` 指示文、`loop` 指示文の追加
- `run.sh`
  - ✓ GPUDirectRDMA の有効化

実行速度が遅くても、動くプログラムである状態を保ちながらOpenACC化します。末端の関数からOpenACC化するのがよいでしょう。

解答例は、`openacc_mpi_fdd2d/02_openacc1`

# data , host\_data指示文

## ■ main関数のwhile 内で data, host\_dataを追加

```
#pragma acc data ¥
copy(ex[0:nelems], ey[0:nelems], hz[0:nelems]) ¥
copyin(cexly[0:nelems], ceylx[0:nelems], chzlx[0:nelems], chzly[0:nelems]) ¥
copy(exy[0:nelems], eyx[0:nelems], hzx[0:nelems], hzy[0:nelems]) ¥
copyin(cexy[0:nelems_y], ceyx[0:nelems_x], chzx[0:nelems_x], chzy[0:nelems_y]) ¥
copyin(cexyl[0:nelems_y], ceysl[0:nelems_x], chzxl[0:nelems_x], chzyl[0:nelems_y]) ¥
copyin(obj[0:nelems], er[0:nelems]) ¥
copyin(rer_ex[0:nelems], rer_ey[0:nelems])
{

#pragma acc host_data use_device(hz)
{
MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up , tag, MPI_COMM_WORLD);
MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag,
MPI_COMM_WORLD, &status);
}

calc_ex_ey(&whole, &inside, hz, cexly, ceylx, ex, ey);
pml_boundary_ex(&whole, &inside, hz, cexy, cexyl, rer_ex, ex, exy);
pml_boundary_ey(&whole, &inside, hz, ceyx, ceysl, rer_ey, ey, eyx);

} // acc data

const int j_in = 0;
plane_wave_incidence(&whole, &inside, time, j_in, wavelength, ex, ey);
time += 0.5*dt;
```

# kernels, loop指示文

## ■ fdtd2d.c 内の関数

```
void calc_ex_ey(const struct Range *whole, const struct Range *inside,
               const FLOAT *hz, const FLOAT *cexly, const FLOAT *ceylx, FLOAT *ex, FLOAT *ey)
{
    const int nx    = inside->length[0];
    const int ny    = inside->length[1];
    const int mgn[] = { inside->begin[0] - whole->begin[0],
                       inside->begin[1] - whole->begin[1] };
    const int lnx   = whole->length[0];

    #pragma acc kernels present(hz, cexly, ex)
    #pragma acc loop independent
        for (int j=0; j<ny+1; j++) {
    #pragma acc loop independent
        for (int i=0; i<nx; i++) {
            const int ix = (j+mgn[1])*lnx + i+mgn[0];
            const int jm = ix - lnx;
            //ex[ix] += cexly[ix]*(hz[ix]-hz[jm]) - cexlz[ix]*(hy[ix]-hy[km]);
            ex[ix] += cexly[ix]*(hz[ix]-hz[jm]);
        }
    }

    (省略)
}
```

# 実習2

---

- main 関数内の while 内をすべて OpenACCにしましょう。
- main.c
  - ✓ data 指示文の移動と copyin などの最適化
  - ✓ MPI\_Send/MPI\_Recvに対するhost\_data指示文の追加
  - ✓ MPI\_Gatherに対する host\_data 指示文の追加
- fdttd2d.c
  - ✓ 残りの関数にkernels 指示文、loop 指示文の追加
- fdttd2d\_sources.c
  - ✓ kernels 指示文、loop 指示文の追加

解答例は、`openacc_mpi_fdttd/03_openacc2`

# data 指示文

## ■ main関数のwhile 外に data を移動

```
#pragma acc data ¥
copyin(ex[0:nelems], ey[0:nelems], hz[0:nelems]) ¥
copyin(cexly[0:nelems], ceylx[0:nelems], chzlx[0:nelems], chzly[0:nelems]) ¥
copyin(exy[0:nelems], eyx[0:nelems], hzx[0:nelems], hzy[0:nelems]) ¥
copyin(cexy[0:nelems_y], ceyx[0:nelems_x], chzx[0:nelems_x], chzy[0:nelems_y]) ¥
copyin(cexyl[0:nelems_y], ceysl[0:nelems_x], chzxl[0:nelems_x], chzyl[0:nelems_y]) ¥
copyin(obj[0:nelems], er[0:nelems]) ¥
copyin(rer_ex[0:nelems], rer_ey[0:nelems])
{

while (icnt < nt) {

    MPI_Status status;
    const int tag = 0;
    const int nhalo      = whole.length[0];
    const int inside_end1 = inside.begin[1] + inside.length[1];

    const int src_hz      = whole.length[0] * (inside_end1      - whole.begin[1] - 1);
    const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

#pragma acc host_data use_device(hz)
    {
    MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up, tag, MPI_COMM_WORLD);
    MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);
    }
}
```

# host\_data 指示文

## ■ MPI\_Gather に対する host\_data の追加

```
const int rank_root = 0;
const int sendelems = whole.length[0] * inside.length[1];
const int src       = whole.length[0] * (inside.begin[1] - whole.begin[1]);
const int dst       = whole.length[0] * (inside.begin[1] - whole.begin[1]);

#pragma acc host_data use_device(ex)
MPI_Gather(&ex[src], sendelems, MPI_FLOAT_T, &ex_global[dst],
          sendelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
#pragma acc host_data use_device(ey)
MPI_Gather(&ey[src], sendelems, MPI_FLOAT_T, &ey_global[dst],
          sendelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
#pragma acc host_data use_device(hz)
MPI_Gather(&hz[src], sendelems, MPI_FLOAT_T, &hz_global[dst],
          sendelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);

if (rank == rank_root) {
    write_bmp(icnt, time, whole_global.length, dx, dy, ex_global, ey_global, hz_global);
}
```

# 実習3

---

- 初期化を含めて全てOpenACCにします。ただし、`set_object_er` がCPU上のユーザ定義関数のため、これ以降の初期化関数をOpenACCにします。
- `main.c`
  - ✓ `data` 指示文の移動と最適化（多くが `create` になるはずです）
  - ✓ 初期化後のMPI\_Gatherに対する `host_data` 指示文の追加
- `setup.c`
  - ✓ `kernels` 指示文、`loop` 指示文の追加

解答例は、`openacc_mpi_fdttd/04_openacc3`

# 実習4

- 計算領域のサイズやGPU数も変更して性能測定してみましよう。
  - ✓ 計算格子サイズを 4096 x 4096 など大きくしてみましよう。
- OpenACCコードをさらに最適化しましよう。
  - ✓ NVCOMPILER\_ACC\_TIMEも活用しましよう。
  - ✓ 実は単純に ftd2d.c に kernels と loop を入れても、いくつかの関数で暗黙の copyin が発生しましよう。これも修正していきましよう。

```
$ make
calc_ex_ey:
 25, Generating present(ex[:],cexly[:])
    Generating implicit copyin(mgn[:])
    Generating present(hz[:])
 27, Loop is parallelizable
 29, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 27, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 29, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 37, Generating present(ey[:],ceylx[:])
    Generating implicit copyin(mgn[:])
```

解答例は、openacc\_mpi\_ftdd/05\_openacc4