第91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：下川辺 隆史
shimokawabe @ 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
  - ✓ 12月4日（月） 10:00 – 18:00

- プログラム
  - ✓ 09:30 - 10:00 受付
  - ✓ 10:00 – 10:40 Reedbush-Hへのログイン
  - ✓ 10:40 – 12:00 GPUとOpenACC基礎（座学）
  - ✓ （昼休み）
  - ✓ 13:00 – 14:00 OpenACC基礎（演習）
  - ✓ 14:00 – 14:50 MPI復習（座学、簡単な演習）
  - ✓ （休憩）
  - ✓ 15:00 – 16:30 OpenACCとMPIによるマルチGPUプログラミング（座学、演習）
  - ✓ 16:40 – 18:00 複数GPUを用いたFDTD法による電磁波伝搬計算（座学、演習）

# 講習会について

- 本講習会では
  - ✓ GPUとOpenACCプログラミングの基礎
  - ✓ 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/support/kosyu/
  - ✓ OpenACC、MPIの機能全般について学びたい場合は、他の講習会も受講することをお勧めします。

- 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。

# 事前準備

# Reedbush にログインする

- 別資料「Reedbush利用の手引き」を参照し、ログインします。

# Reedbush 利用上の注意（1）

- ディレクトリについて（home と lustre)
  - ✓ 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 /lustre 以下のディレクトリ（/lustre/gt00/txxxxx）に置く
  - ✓ /home は計算ノードからは参照できない
  - ✓ cdw コマンドで Lustreファイルシステムへ移動できる。

```
$ cdw
```

# Reedbush 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。

# モジュールの切り替え

- PGIコンパイラ（OpenACCやCUDA Fortran）を使う場合

```
$ module load pgi
```

- CUDA開発環境を使う場合

```
$ module load cuda
```

- Intelコンパイラを使う場合

```
$ module load intel
```

- MPIを使う場合

```
$ module load openmpi-gdr/2.1.1/{gnu,intel,pgi}
$ module load mvapich2-gdr/2.2/{gnu,intel,pgi}
```

  - ✓ PGIなどのコンパイラに追加して load

- モジュールはジョブ実行時にもコンパイル時と同じものを load する。
- 組み合わせて利用できる

# サンプルコードのコンパイル

- Reedbush へのログイン
```
$ ssh -Y txxxxx@reedbush.cc.u-tokyo.ac.jp
```
  - ✓ txxxxx 各自の利用者番号（アカウント）に置き換えてください。
  - ✓ -Y をつけてください。
- cdw コマンドで Lustreファイルシステムへ移動する。
```
$ cdw
```
- 自分のディレクトリにサンプルコードをコピーする。
```
$ cp /lustre/gt00/share/openacc_mpi_samples.tar.gz .
```
- サンプルコードを展開する。
```
$ tar zxvf openacc_mpi_samples.tar.gz
```
- サンプルコードへ移動する。
```
$ cd openacc_mpi_samples
```
- モジュールをロードする。
```
$ module load pgi
```
- コンパイルする。
```
$ cd openacc_hello/01_hello_acc
$ make
```
- 実行ファイルができていることを確認する。
```
$ ls
```

# プログラムの実行

- ジョブとして投入し、実行する。

```
$ qsub ./run.sh
```

- 投入されたジョブを確認する。

```
$ rbstat
```

- 実行が終了すると、以下のファイルが生成される。

```
run.sh.o??????
run.sh.e??????
```
（?????? は数字）

- 上記の標準出力ファイルの中身を確認する。

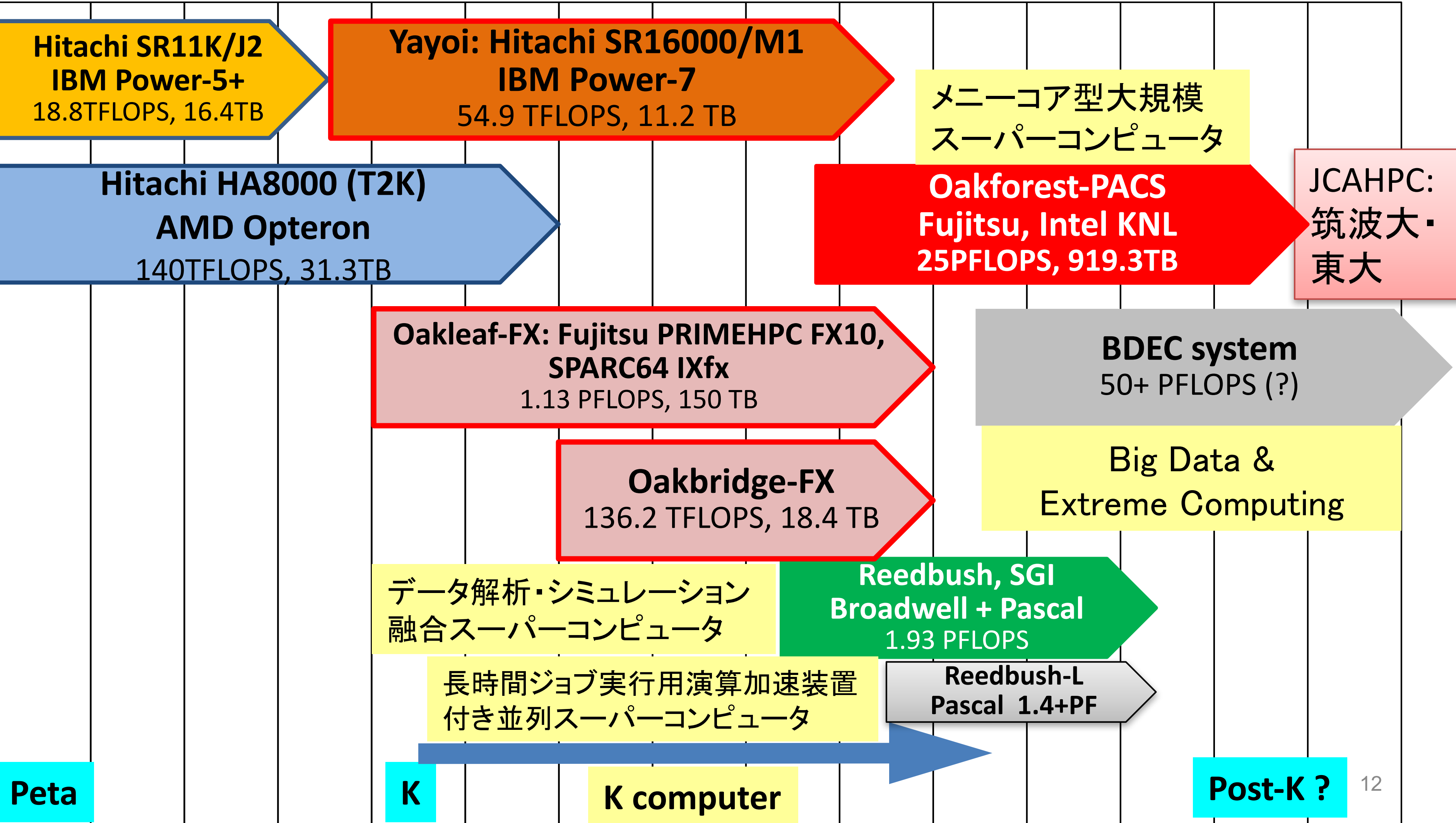
```
$ cat run.sh.o??????
```

- 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 run.sh.e??????
```

# 東大情報基盤センター<br>スーパーコンピュータの概略

# 東大センターのスパコン



# 4システム運用中

- Oakleaf-FX (富士通 PRIMEHPC FX10)
  - ✓ 1.135 PF, 京コンピュータ商用版, 2012年4月 ～ 2018年3月

- Oakbridge-FX (富士通 PRIMEHPC FX10)
  - ✓ 136.2 TF, 長時間実行用（168時間）, 2014年4月 ～ 2018年3月
- 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
  - ✓ 3.361 PF, 2016年7月～ 2020年6月
  - ✓ 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))
  - ✓ JCAHPC (筑波大CCS＆東大ITC)
  - ✓ 25 PF, TOP 500で6位 (2016年11月) (日本で1位)
  - ✓ Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

# 東京大学情報基盤センター スパコン（1/3）

Fujitsu PRIMEHPC FX10 （FX10スーパーコンピュータシステム）

| | |
|---|---|
| Total Peak performance | : 1.13 PFLOPS |
| Total number of nodes | : 4,800 |
| Total memory | : 150TB |
| Peak performance per node | : 236.5 GFLOPS |
| Main memory per node | : 32 GB |
| Disk capacity | : 2.1 PB |
| **SPARC64 IXfx 1.848GHz** | |

2012年7月~2018年3月（予定）

Oakbridge-FX

：長時間ジョブ用のFX10
ノード数：24～576
制限時間：最大168時間
（1週間）

# 東京大学情報基盤センター スパコン（2/3）

**Reedbush** （SGI Rackable　クラスタシステム ）

Reedbush-U (2016/7/1 ~ )

- 理論性能 : 508TFlops
- ノード数 : 420
- ノード構成 : Intel Xeon Broadwell x2

Reedbush-H (2017/3/1 ~ )

- 理論性能 : 1418TFlops
- ノード数 : 120
- ノード構成 : Intel Xeon Broadwell x2 + **NVIDIA P100 GPU x2**

Reedbush-L (**2017/10/1 ~** )

- 理論性能 : 1435TFlops
- ノード数 : 64
- ノード構成 : Intel Xeon Broadwell x2 + **NVIDIA P100 GPU x4**

# 東京大学情報基盤センター スパコン（3/3）

筑波大学計算科学研究センターと共同運用

Oakforest-PACS （Fujitsu PRIMERGY CX600）

| | |
|---|---|
| Total Peak performance | : 25 PFLOPS |
| Total number of nodes | : 8,208 |
| Total memory | : 897.7 TB |
| Peak performance per node | : 3.046 TFLOPS |
| Main memory per node | : 96 GB (DDR4)<br>+ 16 GB(MCDRAM) |
| Disk capacity | : 26.2 PB |
| File Cache system (SSD) | : 960 TB |
| **Intel Xeon Phi 7250 1.4 GHz 68 core x1 socket** | |

2016年12月1日試験運転開始

2017年4月3日正式運用開始

# FX10計算ノードの構成

1ソケットのみ

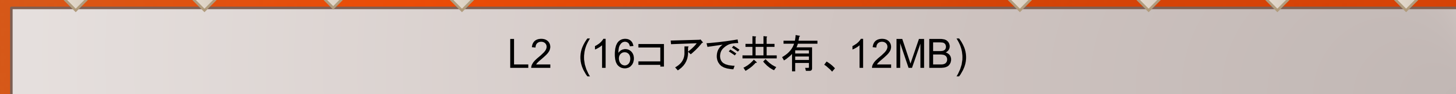
TOFU Network

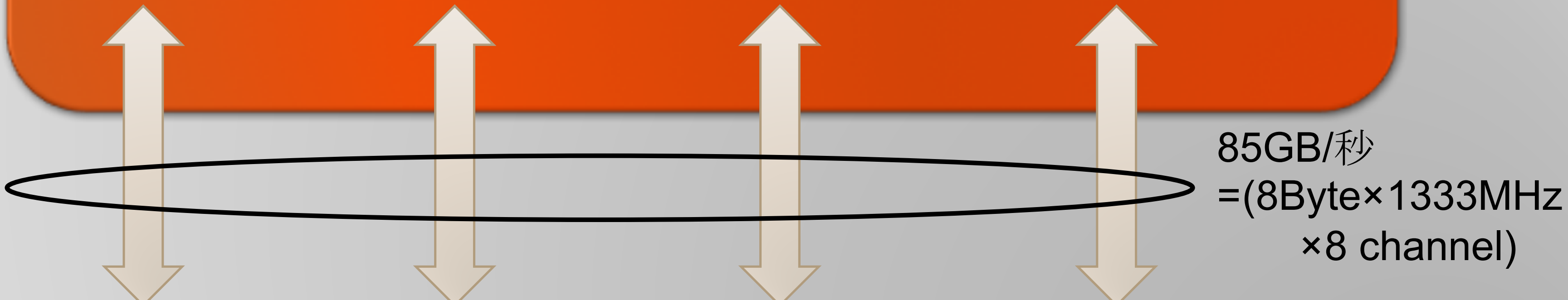
各CPUの内部構成

20GB/秒

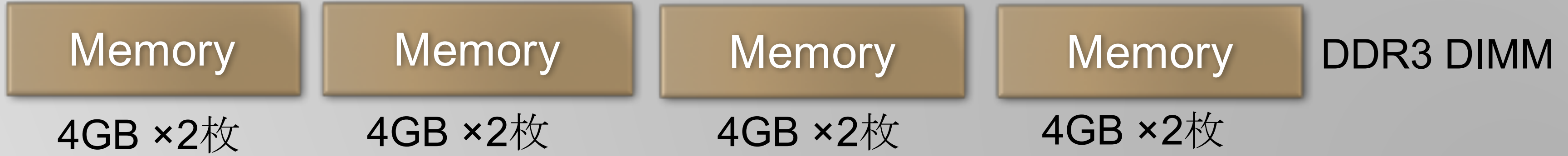
ICC

: L1データキャッシュ32KB



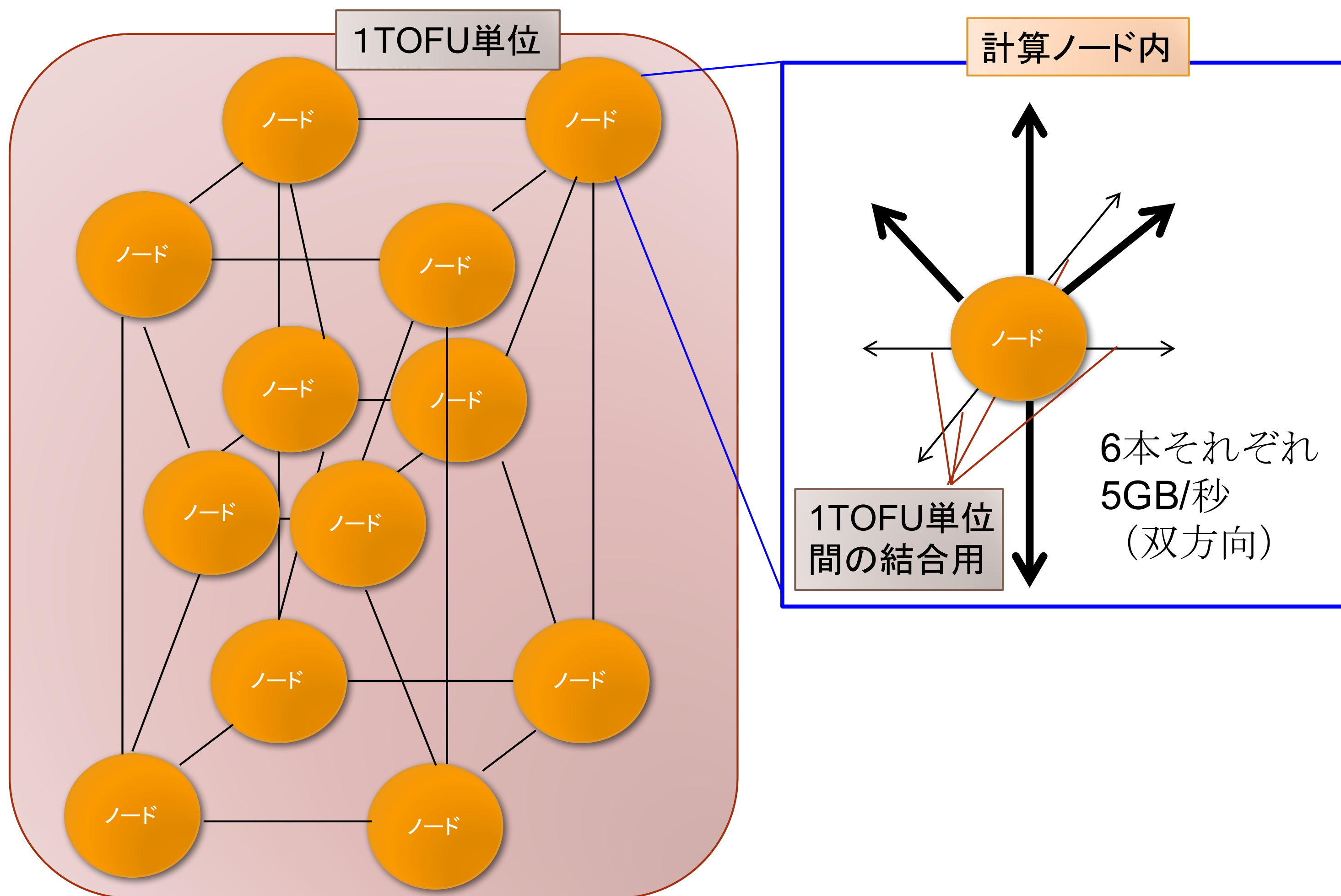
85GB/秒
=(8Byte×1333MHz
×8 channel)



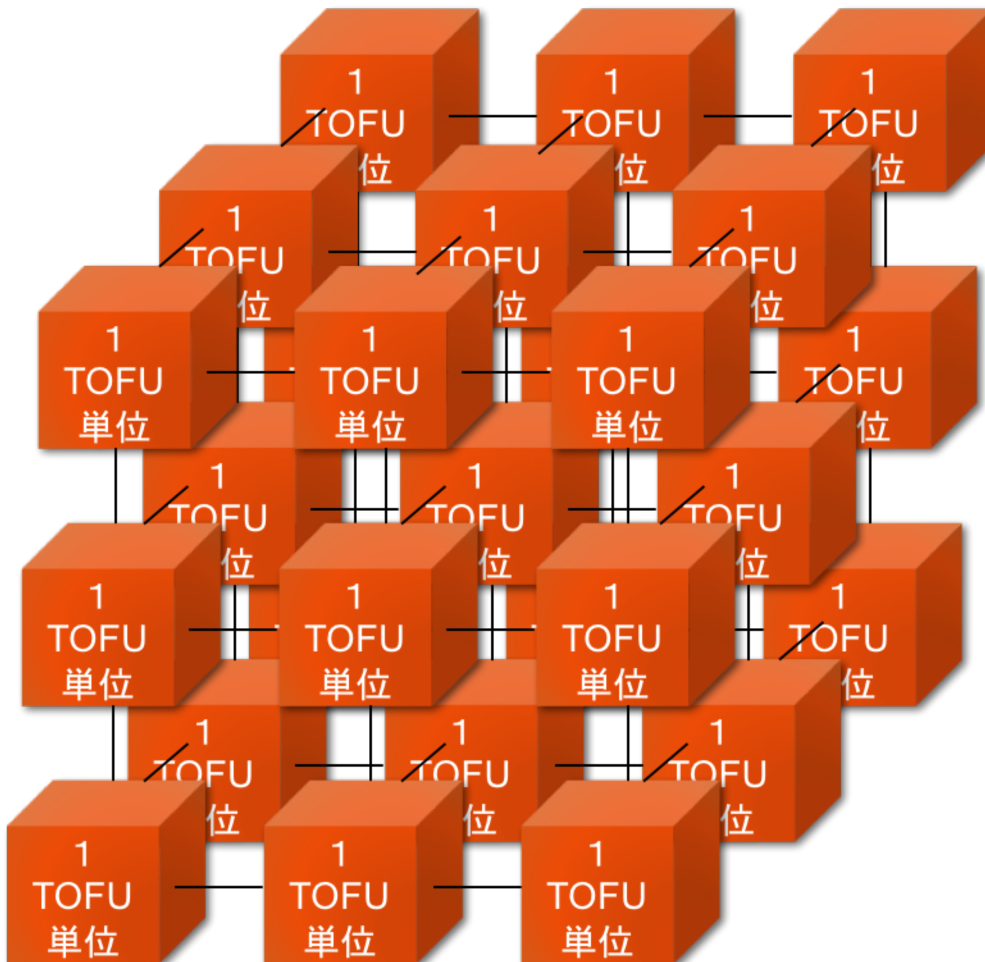

DDR3 DIMM

ノード内合計メモリ量：8GB×4＝32GB

# FX10の通信網



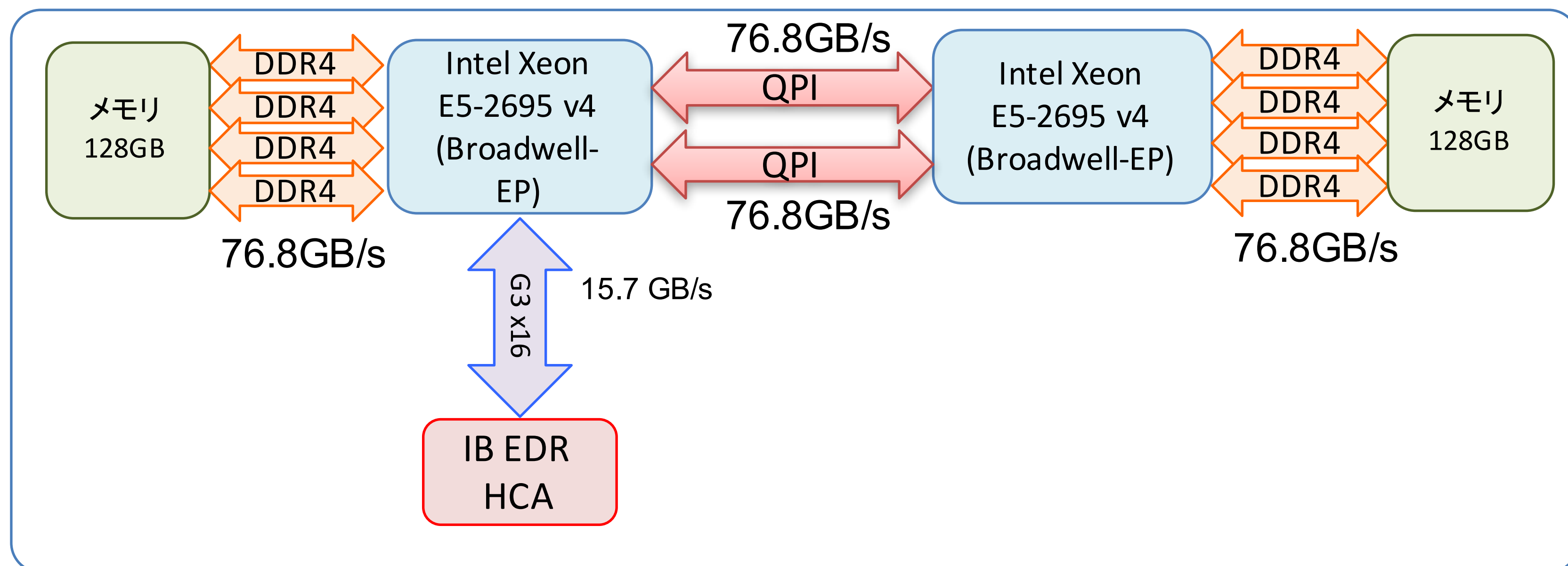
# FX10の通信網（１TOFU単位間の結合）

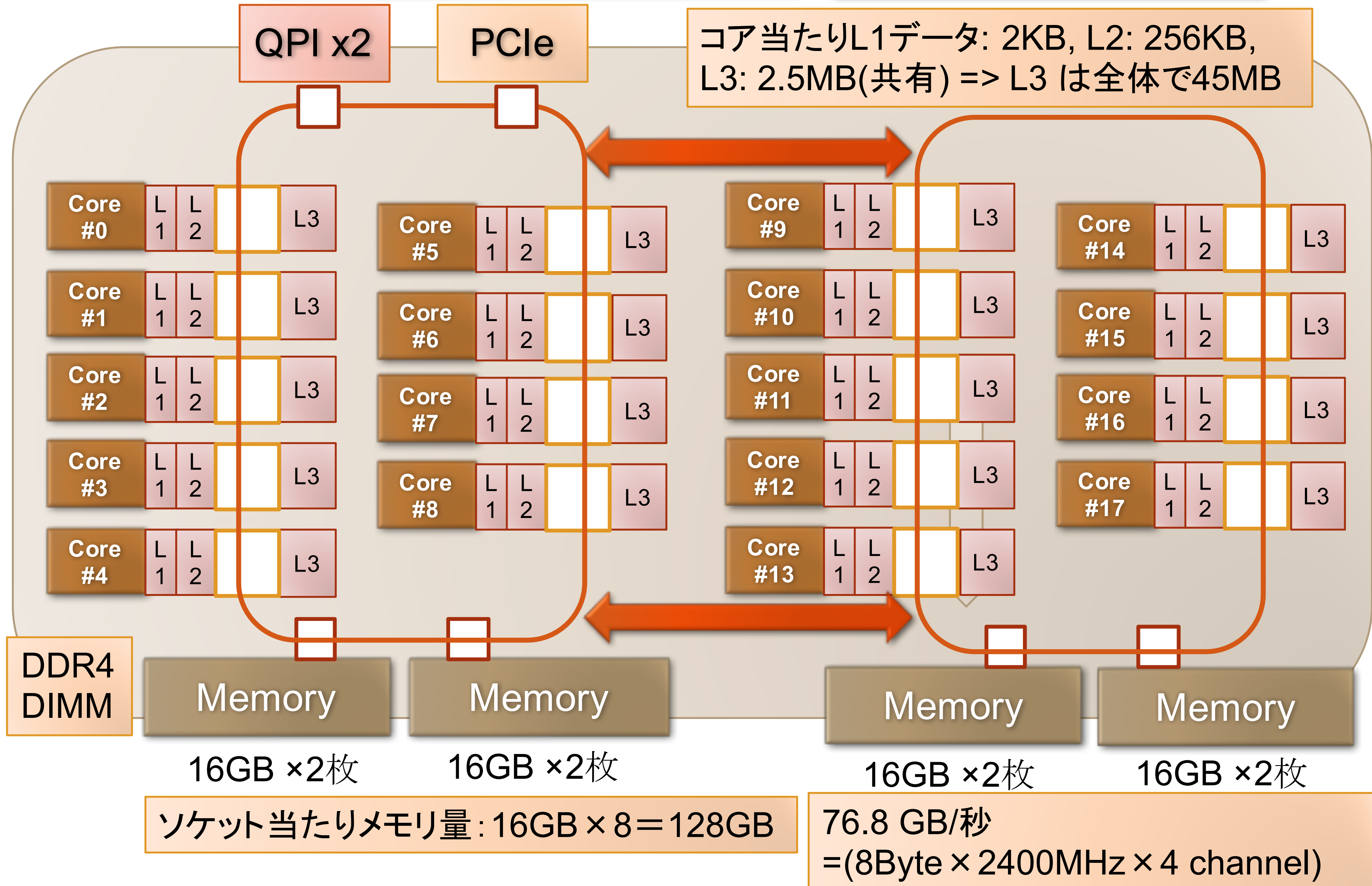



- ユーザから見ると、X軸、Y軸、Z軸について、奥の1TOFUと、手前の１TOFUは、繋がってみえます（3次元トーラス接続）
- ただし物理結線では
  - X軸はトーラス
  - Y軸はメッシュ
  - Z軸はメッシュまたは、トーラス
    になっています

# Reedbush-Uノードのブロック図

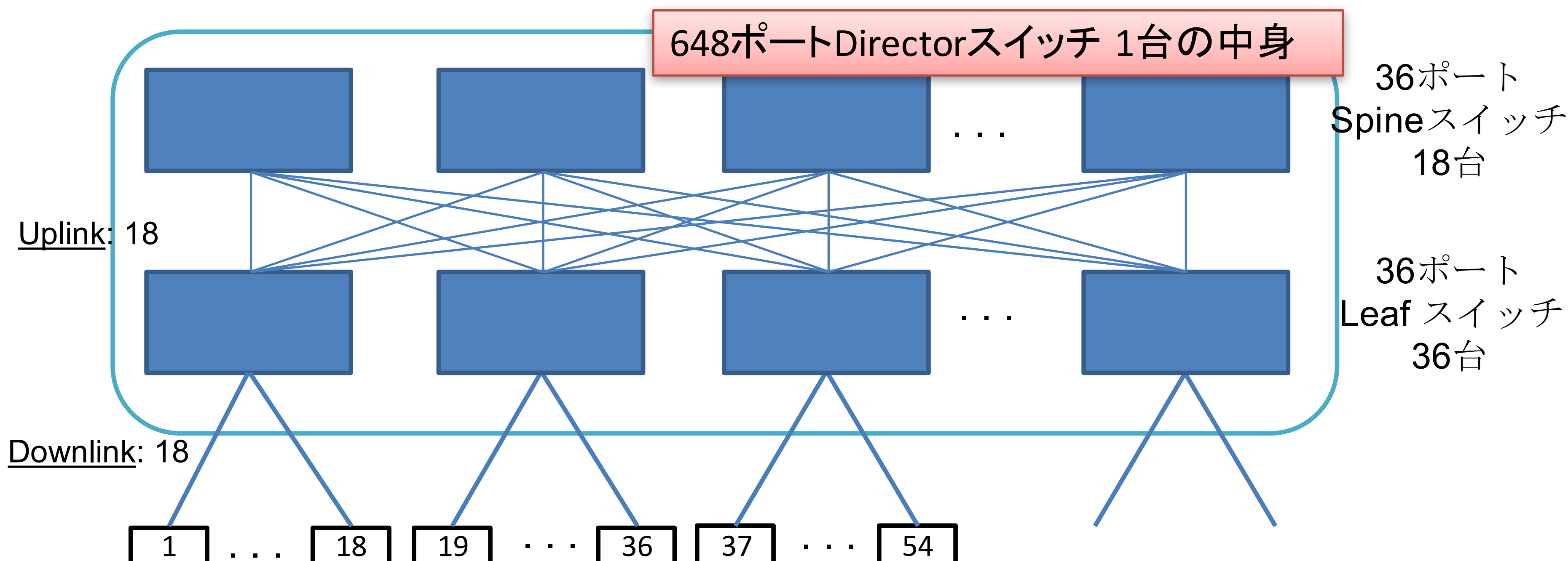
- メモリのうち、「近い」メモリと「遠い」メモリがある
  => NUMA (Non-Uniform Memory Access)
  (FX10はフラット)



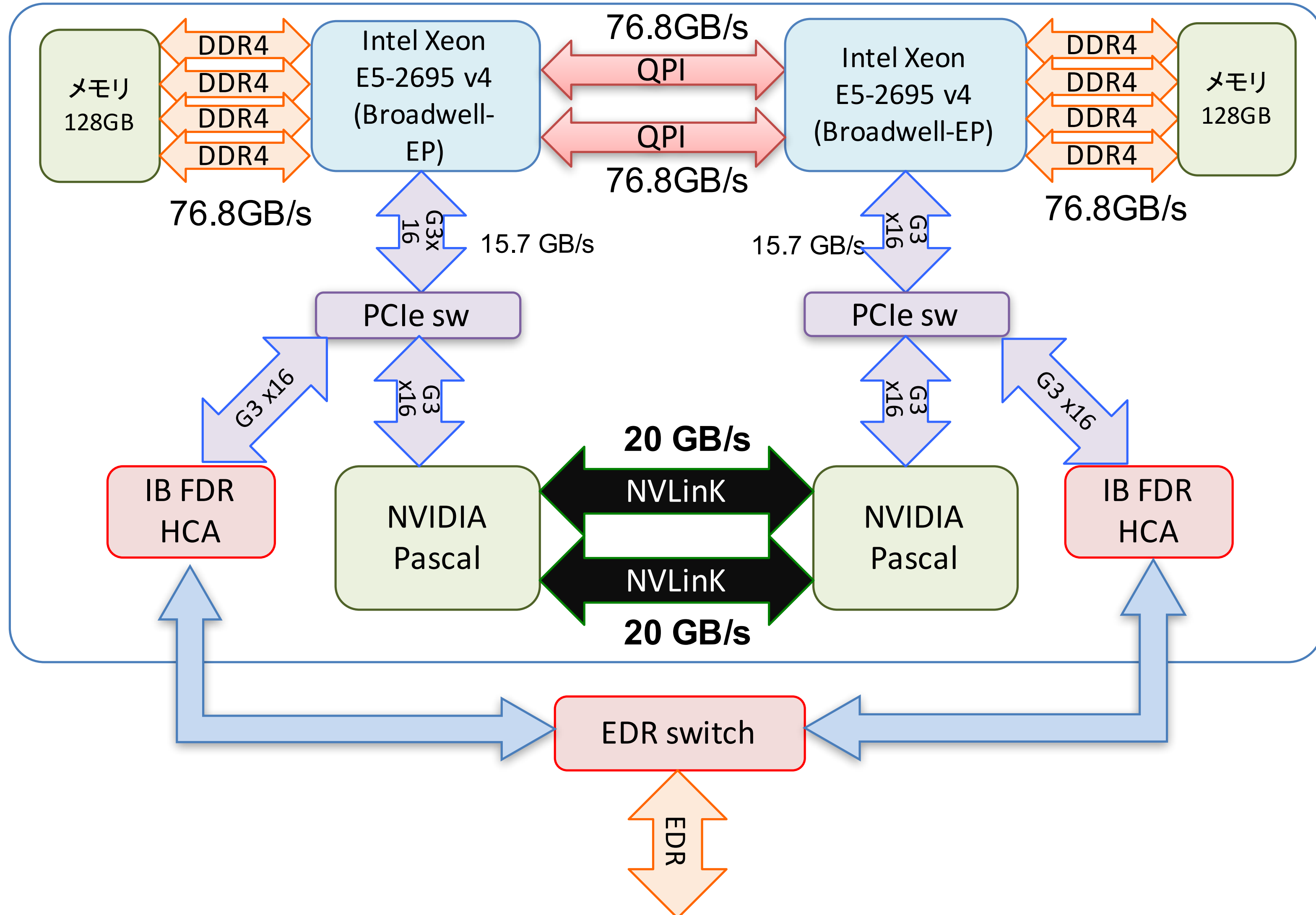

Broadwell-EPの構成
1ソケットのみを図示
QPI x2
PCIe
コア当たりL1データ: 2KB, L2: 256KB,
L3: 2.5MB(共有) => L3 は全体で45MB
Core #0
Core #1
Core #2
Core #3
Core #4
Core #5
Core #6
Core #7
Core #8
Core #9
Core #10
Core #11
Core #12
Core #13
Core #14
Core #15
Core #16
Core #17
L1
L2
L3
DDR4 DIMM
Memory
16GB ×2枚
ソケット当たりメモリ量：16GB×8＝128GB
76.8 GB/秒
=(8Byte×2400MHz×4 channel)

# Reedbush-Uの通信網

- フルバイセクションバンド幅を持つFat Tree網
  - ✓ どのように計算ノードを選んでも互いに無衝突で通信が可能
- Mellanox InfiniBand EDR 4x CS7500: 648ポート
  - ✓ 内部は36ポートスイッチ (SB7800)を (36+18)台組み合わせたものと等価

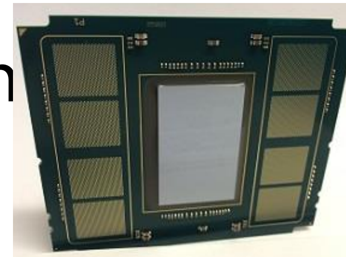


# Reedbush-Hノードのブロック図



# Oakforest-PACS 計算ノード

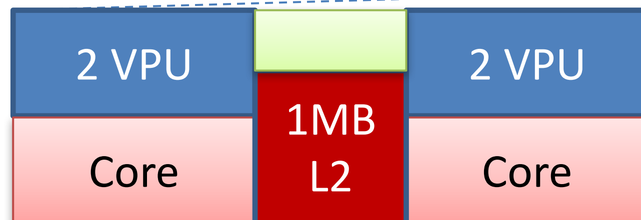
- Intel Xeon Phi (Knights Landin
  - 1ノード1ソケット
- MCDRAM: オンパッケー
  の高バンド幅メモリ16G
  + DDR4メモリ

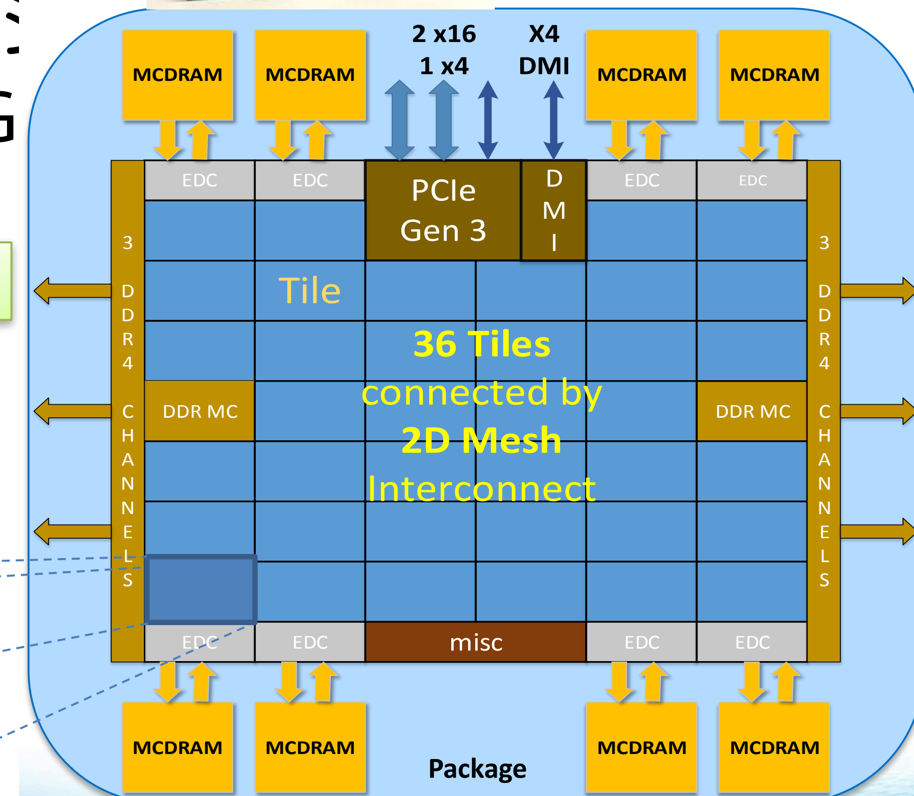


HotChips27
KNLスライドより

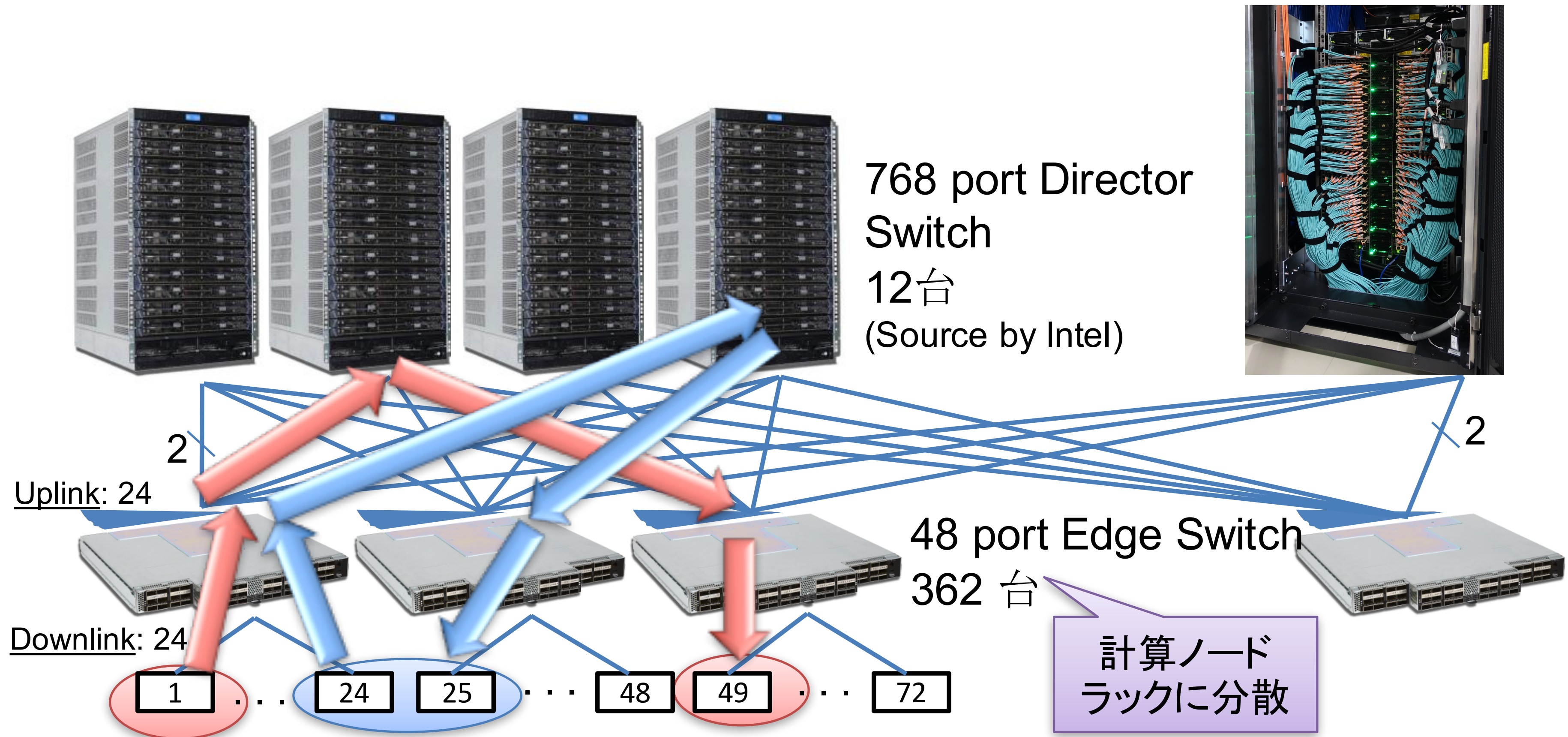
ソケット当たりメモリ量 : 16GB×6＝96GB

MCDRAM: 490GB/秒以上（実測）
DDR4: 115.2 GB/秒
=(8Byte×2400MHz×6 channel)



# Oakforest-PACS: Intel Omni-Path Architecture による フルバイセクションバンド幅Fat-tree網




コストはかかるがフルバイセクションバンド幅を維持
- システム全系使用時にも高い並列性能を実現
- 柔軟な運用：ジョブに対する計算ノード割り当ての自由度が高い

# 東大情報基盤センターOakforest-PACSスーパーコンピュータシステムの料金表（2017年4月1日）

- パーソナルコース（年間）
  - コース１：　100,000円　：　8ノード(基準)、最大16ノードまで
  - コース２：　200,000円　：　16ノード(基準)、最大64ノードまで
- グループコース
  - 400,000円 (企業 480,000円)　：1口 8ノード（基準）、最大128ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはFX10、Reedbushとの相互トークン移行も可能

# 東大情報基盤センターReedbushスーパーコンピュータシステムの料金表（2017年4月1日）

- パーソナルコース（年間）
  - 150,000円　：　RB-U: 4ノード（基準）、最大16ノードまで
    RB-H: 1ノード（基準）、最大2ノードまで
- グループコース
  - 300,000円：1口 4ノード（基準）、最大128ノードまで、
    RB-H：1ノード（基準）、最大32ノードまで（トークン係数はUの2.5倍）
  - RB-Uのみ 企業 360,000円　：1口 4ノード（基準）、最大128ノードまで
  - RB-Hのみ 企業 216,000円　：1口 1ノード（基準）、最大32ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはFX10, Oakforest-PACSとの相互トークン移行も可能
  - ノード固定もあり

# 東大情報基盤センターFX10スーパーコンピュータシステムの料金表（2017年4月1日）

- パーソナルコース（年間）
  - コース１： 90,000円 ： 12ノード(基準)、最大24ノードまで
  - コース２： 180,000円 ： 24ノード(基準)、最大96ノードまで
- グループコース
  - 360,000円 (企業 432,000円) ：1口、12ノード、最大1440ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはReedbush, Oakforest-PACSとの相互トークン移行も可能

# JPY (=Watt)/GFLOPS Rate

Smaller is better (efficient)

| System | JPY/GFLOPS |
|---|---|
| Oakleaf/Oakbridge-FX (Fujitsu) (Fujitsu PRIMEHPC FX10) | 125 |
| Reedbush-U (SGI) (Intel BDW) | 62.0 |
| Reedbush-H (SGI) (Intel BDW+NVIDIA P100) | 17.1 |
| Oakforest-PACS (Fujitsu) (Intel Xeon Phi/Knights Landing) | 16.5 |

# トライアルユース制度について

- 安価に当センターのOakleaf/Oakbridge-FX, Reedbush-U/H, Oakforest-PACSシステムが使える「無償トライアルユース」および「有償トライアルユース」制度があります。
  - アカデミック利用
    - パーソナルコース、グループコースの双方（1ヶ月～3ヶ月）
  - 企業利用
    - パーソナルコース（１ヶ月～３ヶ月）（FX10: 最大24ノード、最大96ノード,
      RB-U: 最大16ノード, RB-H: 最大2ノード、OFP: 最大16ノード, 最大64ノード）
      講習会いずれかの受講が必須、審査無
    - グループコース
      - 無償トライアルユース：（１ヶ月～３ヶ月）：無料（FX10:最大1,440ノード、 RB-U: 最大128ノード、RB-H: 最大32ノード、OFP: 最大2048ノード）
      - 有償トライアルユース：（１ヶ月～最大通算９ヶ月）、有償（計算資源は無償と同等）
      - スーパーコンピュータ利用資格者審査委員会の審査が必要（年2回実施）
    - 双方のコースともに、簡易な利用報告書の提出が必要

# スーパーコンピュータシステムの詳細

- 以下のページをご参照ください
  - 利用申請方法
  - 運営体系
  - 料金体系
  - 利用の手引

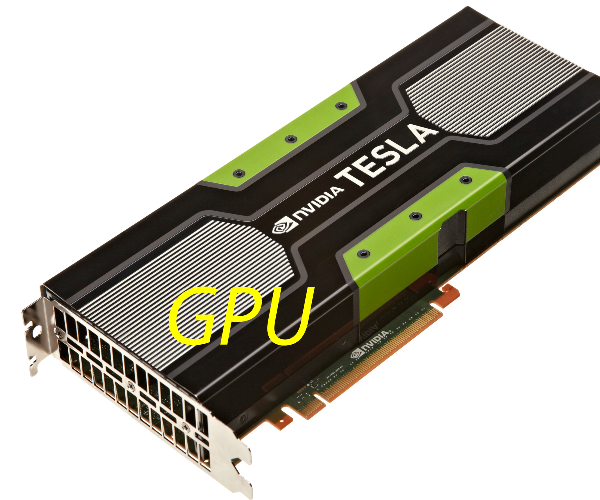
  などがご覧になれます。

  http://www.cc.u-tokyo.ac.jp/system/ofp/
  http://www.cc.u-tokyo.ac.jp/system/reedbush/
  http://www.cc.u-tokyo.ac.jp/system/fx10/

# GPU入門

# What's GPU ?

- Graphics Processing Unit
- もともと PC の3D描画専用の装置
- パソコンの部品として量産されてる。= 非常に安価



http://www.nvidia.co.jp

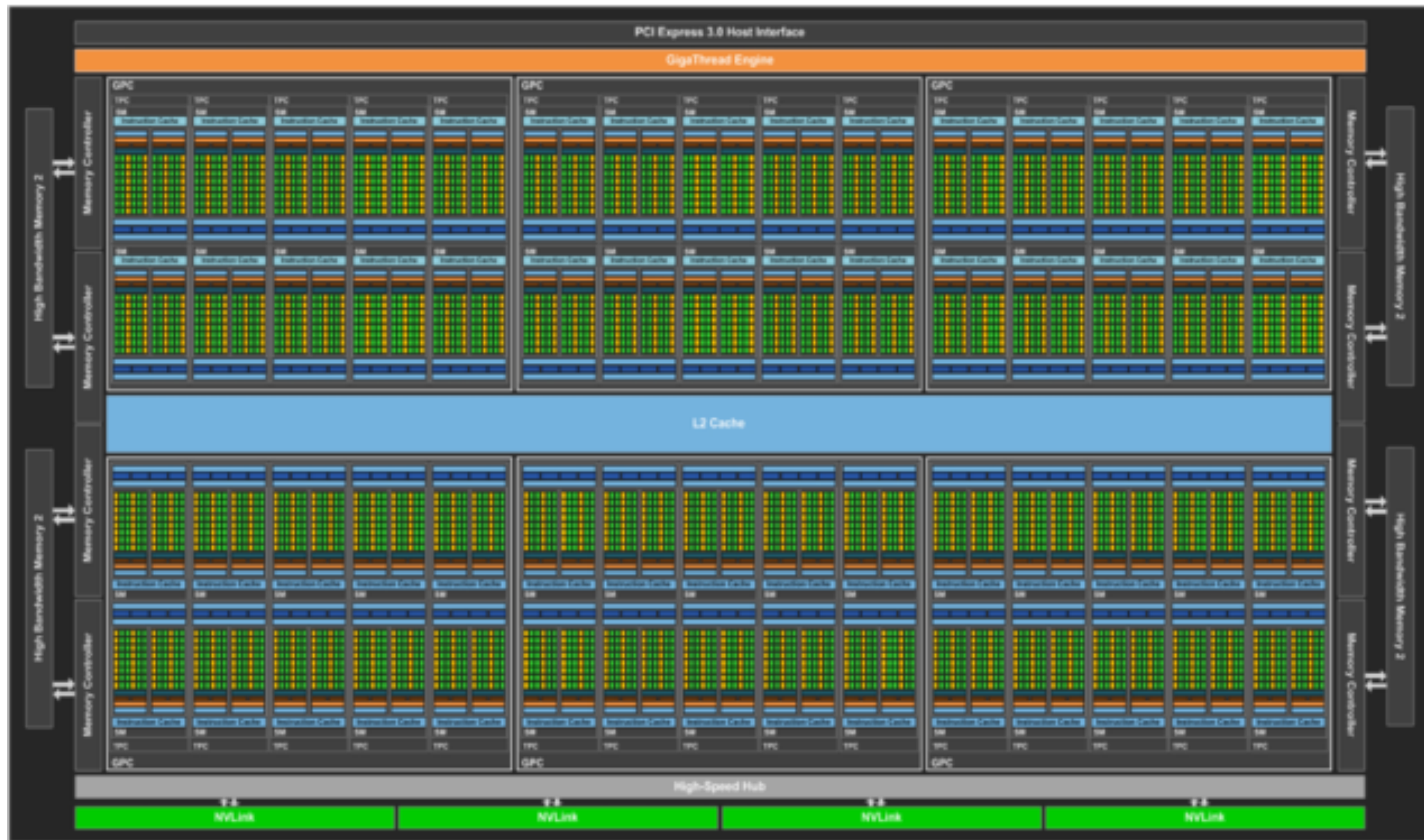
# GPUコンピューティング

- GPUはグラフィックスやゲームの画像計算のために進化を続けている。
- CPUがコア数が2-12個程度に対し、GPUは1000以上のコアがある。
- GPUを一般のアプリケーションの高速化に利用することを「GPUコンピューティング」「GPGPU (General Purpose computation on GPU)」などという。
- 2007年にNVIDIA社のCUDA言語がリリースされて大きく発展
- ここ数年、ディープラーニング（深層学習）、機械学習、AI（人工知能）などでも注目を浴びている。

# GPUの特徴

- なぜGPUが使われるのか？
  - ✓ 性能高い
    - ✓ NVIDIA P100 (Reedbush-H) 5,304 GFlops
    - ✓ Intel Xeon Phi (Oakforest-PACS) 3,046.4 GFLops
  - ✓ 消費電力低い
    - ✓ スパコンに搭載されている
- コンピュータに取り付ける増設ボード
  - ✓ 単体では動作できず、CPUからの指令が必要。
  - ✓ CPUとGPUはメモリが異なるため、メモリ間のデータ移動もプログラミングする必要がある。
- 多数の小さなコア（1000以上）を搭載。これを有効に活用するため「並列計算」が必須。
  - ✓ CPUは大きなコア：分岐予測、パイプライン処理などできる。逐次処理が得意。
  - ✓ GPUは小さなコア：CPUの持つ機能がほとんどないか、全くない。

# NVIDIA Tesla P100

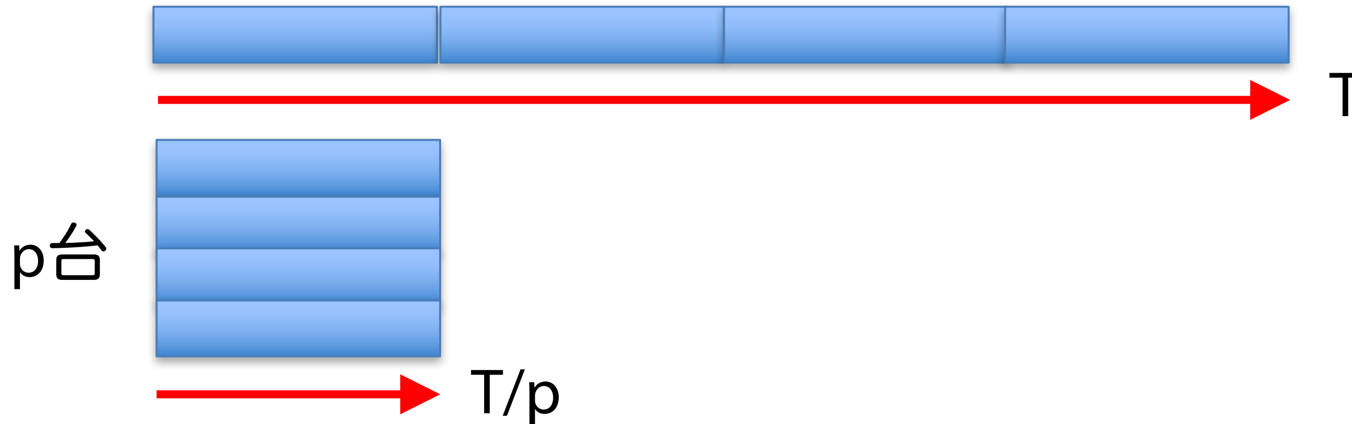



- 56 SMs, 3584 CUDAコア, 16 GByte

Tesla P100 whitepaper より

# 並列計算

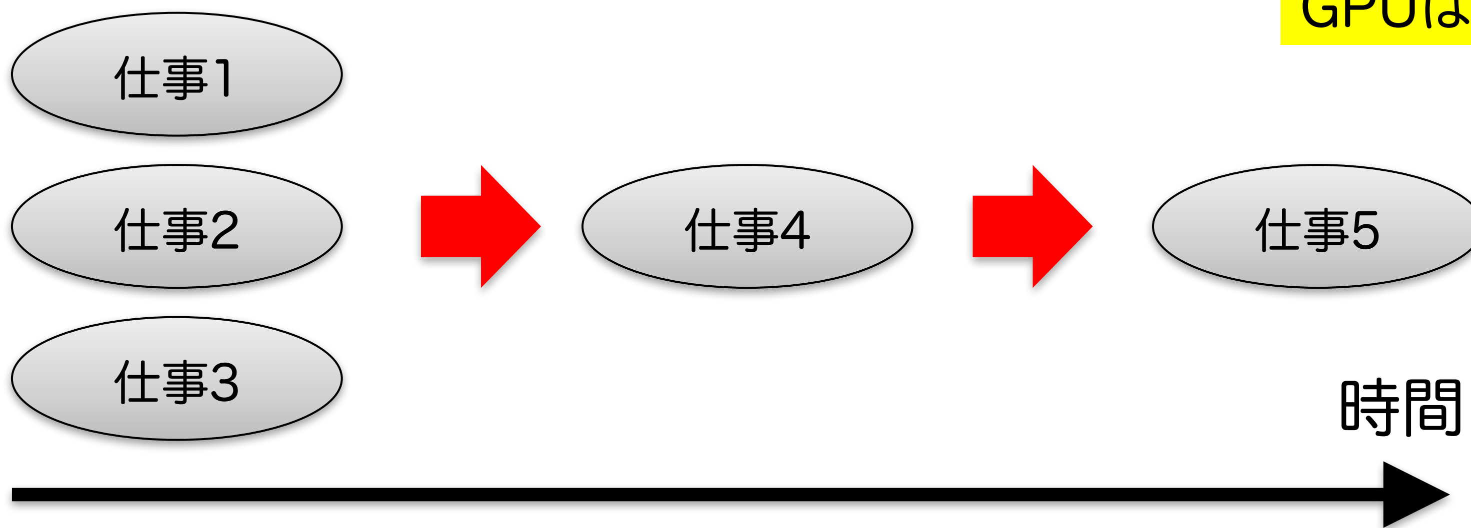
- 実行時間Ｔの逐次処理のプログラムをｐ台の計算機で並列計算することで、実行時間をＴ/ｐ にする。




- 実際にできるかどうかは、処理内容（アルゴリズム）による。アルゴリズムによって難易度は異なる。
  - ✓ 並列化できないアルゴリズム、通信のオーバーヘッド
  - ✓ 部分的にでも並列化できないアルゴリズムがあると、どれだけ並列数を上げても、その時間は短縮されない。
- 並列処理（計算）の種類
  - ✓ 「タスク並列」と「データ並列」

# タスク並列

- タスク（仕事）を分割することで並列化する。
- タスク並列の例：カレーを作る
  - ✓ 仕事１：野菜を切る
  - ✓ 仕事２：肉を切る
  - ✓ 仕事３：水を沸騰させる
  - ✓ 仕事４：野菜と肉を入れて煮込む
  - ✓ 仕事５：カレーのルウを入れる
- 並列化



# データ並列

- データを分割することで並列化する。
  - ✓ データは異なるが計算の手続きは同じ。
- データ並列の例：手分けをして算数ドリルを解く
  - ✓ 数字だけ異なるが計算の手続きは同じ。

| | | | |
|---|---|---|---|
| 2 + 1 = | 12 – 88 = | 34 + 3 = | 2 + 4 = |
| 3 + 19 = | - 20 + 29 = | 1 + 2 = | 99 – 72 = |
| 4 - 6 = | 4 + 10 = | -3 + 2 = | 2 + 10 = |
| 5 + 3 = | 1 + 10 = | -10 – 10 = | 3 – 11 = |
| -2 + 10 = | 1 + 13 = | 2 + 1 = | 12 – 34 = |
| 1 + 2 = | 0 + 0 = | 1 -10 = | 45 + 19 = |

GPUの並列計算はこれが原則。

# GPUプログラミングでの注意点（1）

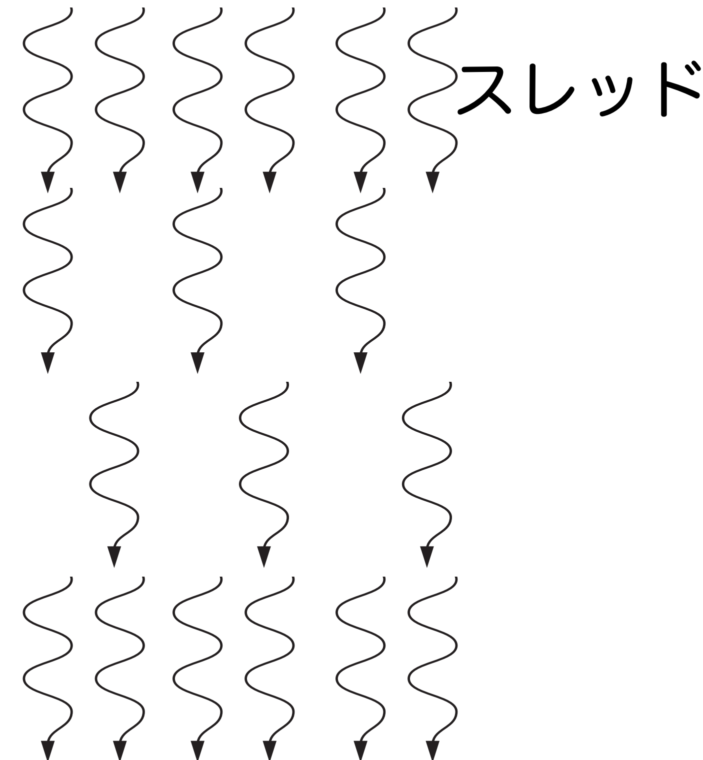
これらはプログラミング言語が CUDA か OpenACCに関わらず、GPUプログラミングでは考慮する必要がある。

- CPUメモリとGPUメモリが独立
  - ✓ 特にOpenACCでは、これを意識する必要がある
- 高い性能を得るには、スレッド数 >> コア数（P100 では 3584）
  - ✓ 数万から数百万スレッド程度
  - ✓ メモリアクセスによる暇な時間（ストール）を、他のスレッドが埋めることができる
  - ✓ Intel CPU では Hyperthread に相当。ただしこちらは物理コア x 2

# GPUプログラミングでの注意点（2）

- スレッド間での異なる処理（分岐）は避ける
  - ✓ 連続した32スレッド（Warp と呼ぶ）が同じ命令を実行
  - ✓ Warp内で分岐先が異なると、全ての分岐先を実行し、真になる条件の処理のみ採用
  - ✓ branch divergence, divergent branch などという。

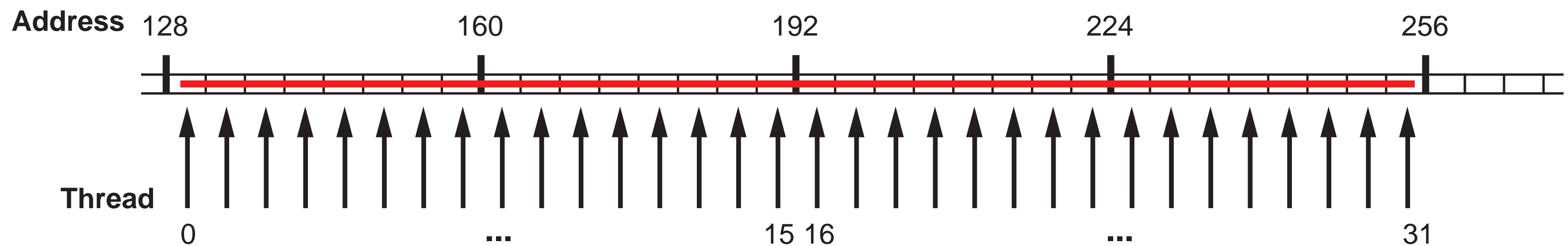
```
...
...
if (奇数番スレッド) {

    処理 A;

} else {

    処理 B;

}
...
...
```



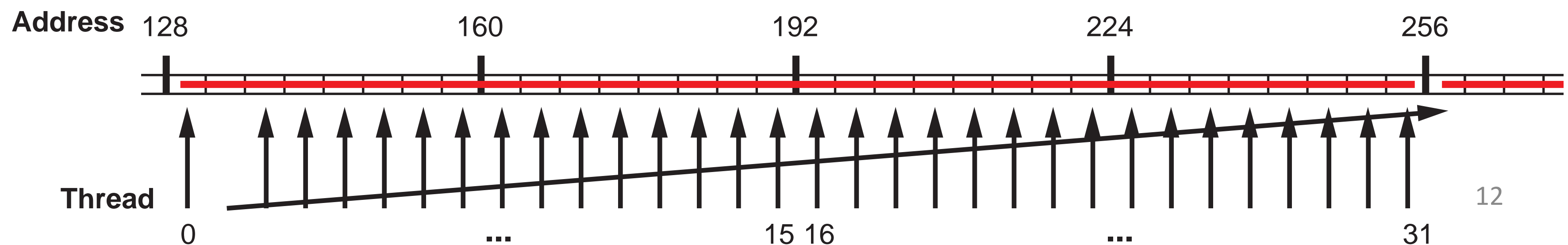
# GPUプログラミングでの注意点（3）

- 同じWarp内のスレッド（連続するスレッド）は近いメモリアドレスへアクセスすると効率的
  - ✓ コアレスドアクセス（coalesced access）と呼ぶ
  - ✓ メモリアクセスは128 Byte 単位で行われる。128 Byte に収まれば1回のアクセス、超えれば128 Byte アクセスをその分繰り返す。

128 byte x 1回のメモリアクセス




128 byte x 2回のメモリアクセス



# OPENACC入門

# GPUコンピューティングの方法

- ライブラリの利用（CUFFT, CUBLAS など）
  - ✓ GPU用ライブラリを呼ぶだけで、すぐに利用できる。
  - ✓ ライブラリ以外の部分は高速化されない。
- 指示文ベース（OpenACC）
  - ✓ 指示文（ディレクティブ）を挿入するだけである程度高速化。
  - ✓ 既存のソースコードを活用できる。
- プログラミング言語（CUDA、OpenCLなど）
  - ✓ GPUの性能を最大限に活用。
  - ✓ プログラミングにはGPGPU用言語を使用する必要あり。

簡単

難しい

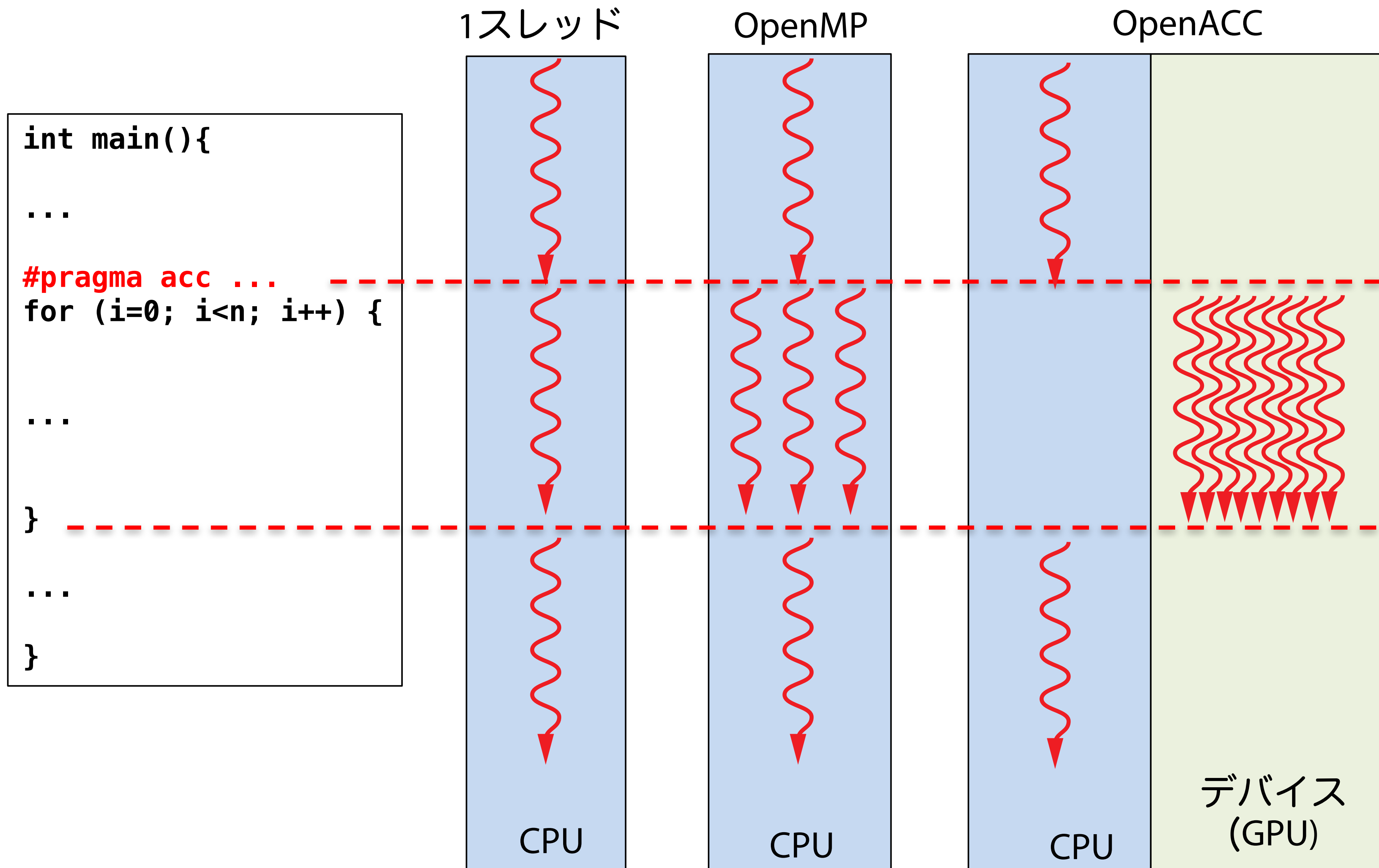
# OpenACC

- OpenACCとは
  - 新しいアクセラレータ用プログラミングインターフェース
  - OpenMP のようなディレクティブ（指示文）・ベース
  - C 言語/C++, Fortran に対応
  - 2011年秋に OpenACC1.0、最新は 2.5
  - コンパイラ：［商用］PGI, Cray,［フリー］GCC（PGIは無料版あり）
  - WEBサイト：http://www.openacc.org/
- 指示文ベースの利点
  - 指示文：コンパイラへのヒント
  - アプリケーションの開発や移植が比較的簡単
  - ホスト（CPU）用コード、複数のアクセラレータ用コードを単一コードとして記述。メンテナンスが容易。高生産性。

C言語
```
#pragma acc directive-name [clause, …]
{
    // C code
}
```

Fortran
```
!$acc directive-name [clause, …]
    ! Fortran code
!$acc end directive-name
```

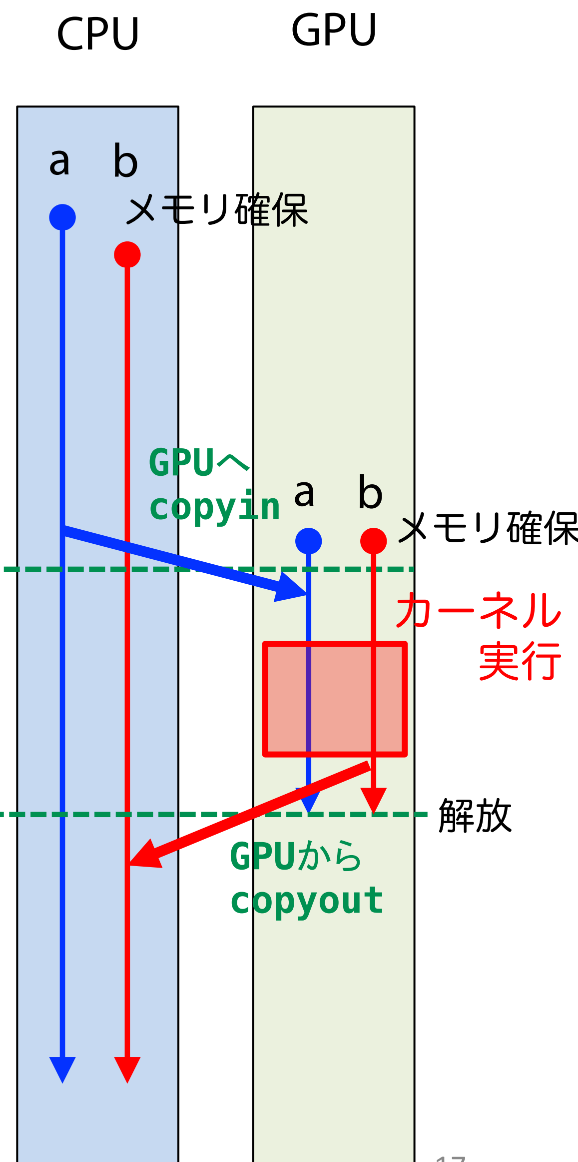
# 逐次、OpenMP、OpenACCの実行イメージ



# はじめてのOpenACCコード

openacc_hello/01_hello_acc

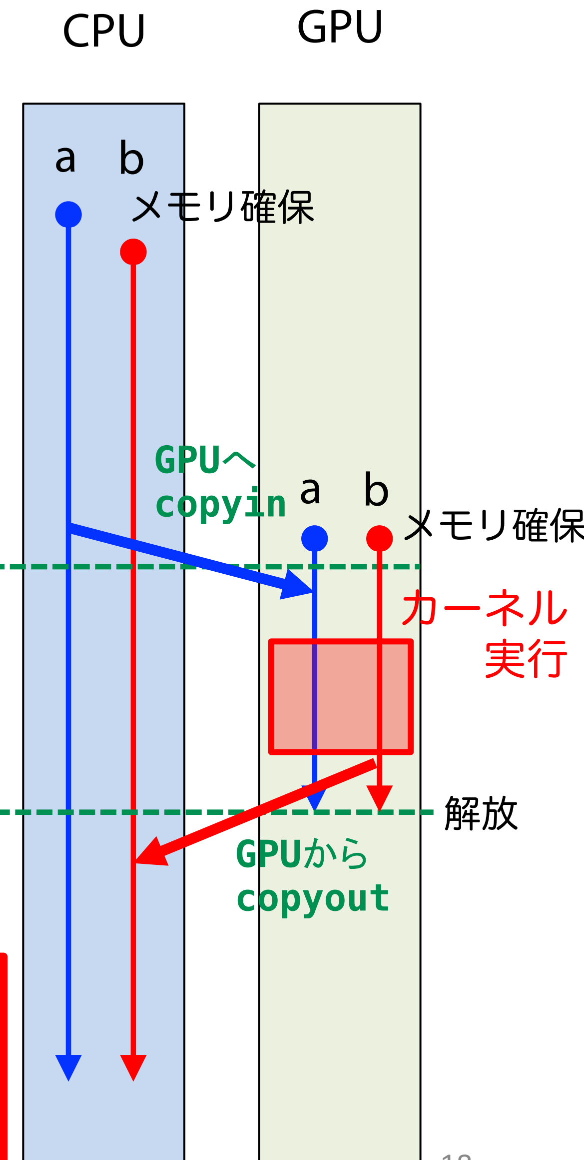
```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```



# はじめてのOpenACCコード

openacc_hello/01_hello_acc

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

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




コード上同じ a, b であっても、原則として
ホストコードはホストメモリで確保された a, b 、GPUで実行される並列領域（カーネル）はデバイスメモリで確保された a, b
を参照しにいく。

# OpenACCの主な指示文

- 並列領域指定指示文
  - ✓ kernels, parallel
- データ管理・移動指示文
  - ✓ data, enter data, exit data, update
- 並列処理の指定
  - ✓ loop
- その他
  - ✓ host_data, atomic, routine, declare

赤字：この講習会で扱うもの

# OpenACCによるアクセラレータでの実行

- kernels 指示文
  - ✓ 指定された領域がアクセラレータで実行されるカーネルへ
  - ✓ 一般には、それぞれのループが別々のカーネルへコンパイル

```
int main() {

#pragma acc kernels
{
    for (int i=0; i<n; i++) {
        A;
    }

    for (int i=0; i<n; i++) {
        B;
    }
}

}
```

kernel 1

kernel 2

- ✓ 同様な指示文として、領域内が一つのカーネルとして生成される parallel 指示文もある

# CPUコードのOpenACC化

openacc_hello/01_hello_acc

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }




for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

- ループのOpenACC 化
  - ✓ 並列化したいループにkernels, loop 指示文を追加

# CPUコードのOpenACC化

openacc_hello/01_hello_acc

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }


#pragma acc kernels
#pragma acc loop independent
for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

- ループのOpenACC 化
  - ✓ 並列化したいループにkernels, loop 指示文を追加

# CPUコードのOpenACC化

openacc_hello/01_hello_acc

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }


#pragma acc kernels
#pragma acc loop independent
for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

- ループのOpenACC 化
  - ✓ 並列化したいループにkernels, loop 指示文を追加

カーネルとしてコンパイルされ、
GPU上で実行される
配列の1要素が1スレッドで処理されるイメージ

# OpenACCのデフォルトの変数の扱い

- スカラ変数
  - ✓ firstprivate または private
  - ✓ ホストからデバイスへコピーが渡され初期化。ホストに戻せない。
- 配列
  - ✓ shared
  - ✓ デバイスメモリに動的に確保され、スレッド間で共有。
  - ✓ デバイスからホストへコピーすることが可能。

- kernels 構文に差し掛かると、
  - ✓ OpenACCコンパイラは実行に必要なデータを自動で転送する。
  - ✓ 配列はデバイスメモリに確保され、shared になる。
  - ✓ 構文に差し掛かるたびに転送を行う。data 指示文で制御できる。

# データ管理・移動

■ data 指示文

- ✓ デバイス(GPU)メモリの確保と解放、ホスト(CPU)とデバイス(GPU)間のデータ転送を制御

  kernels指示文では、データ転送は自動的に行われる。data指示文でこれを制御することで、不要な転送を避け、性能向上できる

- ✓ CUDA で言うところの cudaMalloc, cudaMemcpy に相当

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }
```

openacc_hello/01_hello_acc

変数 c はスカラ変数のため、自動的に デバイスへコピーされ、プライベート変数となる。

# data 指示文の指示節

- copy
  - ✓ allocate, memcpy(H->D), memcpy(D->H), deallocate
- copyin
  - ✓ allocate, memcpy(H->D), deallocate
  - ✓ 解放前にホストへデータをコピーしない
- copyout
  - ✓ allocate, memcpy(D->H), deallocate
  - ✓ 確保後にホストからデータをコピーしない
- create
  - ✓ allocate, deallocate
  - ✓ コピーしない
- present
  - ✓ 何もしない。既にデバイス上で確保済みであることを伝える。

- copy/copyin/copyout/create は既にデバイス上確保されているデータに対しては何もしない。present として振る舞う。（OpenACC2.5以降）

# データの移動範囲の指定

- ホストとデバイス間でコピーする範囲を指定
  - 部分配列の転送が可能
  - Fortran と C言語で指定方法が異なるので注意

- 二次元配列A転送する例
  - Fortran: 下限と上限を指定

```
!$acc data copy(A(lower1:upper1, lower2:upper2) )
...
!$acc end data
```

  - C言語: 始点とサイズを指定

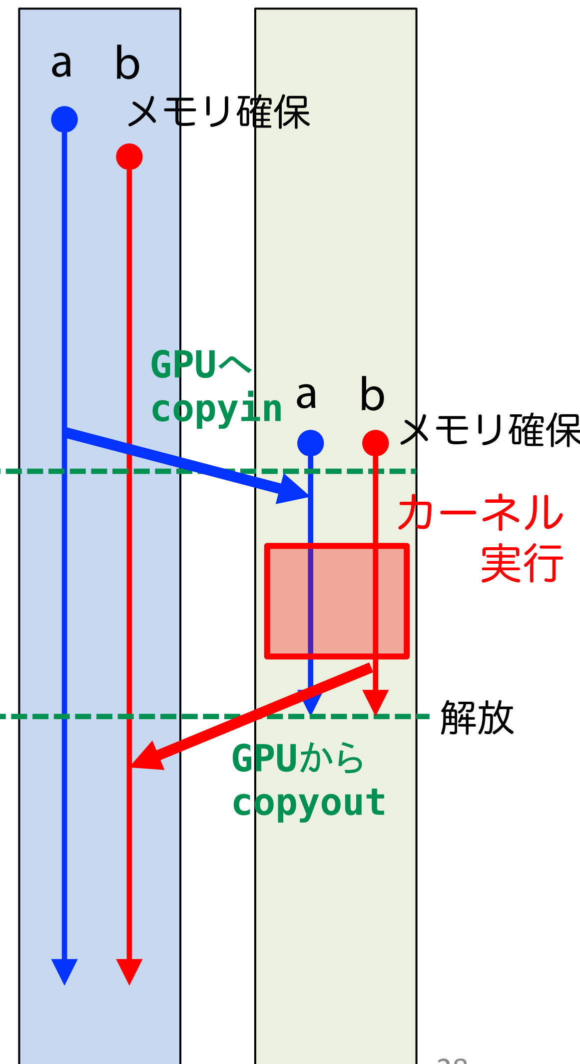
```
#pragma acc data copy(A[begin1:length1][begin2:length2])
...
```

# 並列処理の指定

openacc_hello/01_hello_acc

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

loop指示文



# 並列処理の指定：loop 指示文

- loop 指示文
  - ✓ ループマッピングのパラメータの調整（CUDAのスレッドブロック数を指定）

    gang, worker, vector を用いて指定する。大まかに以下のように考えると良い。

    - gang:  CUDA の thread block 数の指定
    - vector: CUDA の block 内の threads 数の指定
  - ✓ ループがデータ独立であることを指定 (independent clause)

    データ独立でないと並列化できない。C言語ではコンパイラがしばしばデータ独立であることを判断できないので、その場合はこれを指定。
  - ✓ リダクション処理 (reduction clause)
  - ✓ 逐次処理 (seq clause)

# データの独立性

- independent 指示節 により指定
  - ✓ ループがデータ独立であることを明示する
  - ✓ コンパイラが並列化できないと判断したときに使用する

```
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }
```

並列化可能（データ独立）なので、independent を指定
（コンパイラは並列化可能とは判断してくれなかった）

- データ独立でない（並列化可能でない）例

```
// これは正しくない

#pragma acc kernels
#pragma acc loop independent
    for (int i=1; i<n; i++) {
        d[i] = d[i-1];
    }
```

# 参考：OpenACC 化とCUDA化の比較

```
// OpenACC

void calc(int n, const float *a,
const float *b, float c, float *d)
{
#pragma acc kernels present(a, b, d)
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        d[i] = a[i] + c*b[i];
    }
}

int main()
{
    ...
#pragma acc data copyin(a[0:n], b[0:n]) copyout(d[0:n])
    {
        calc(n, a, b, c, d);
    }
    ...
}
```

kernel

- ✓ kernels 指示文でGPUでの実行領域を指定。
- ✓ loop 指示文で並列処理の最適化。
- ✓ data 指示文でデータ転送を制御。kernels 指示文でデータ転送を自動的に行うこともできる。

```
// CUDA

__global__
void calc_kernel(int n, const float *a, const float *b,
float c, float *d)
{
    const int i   = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        d[i] = a[i] + c*b[i];
    }
}

void calc(int n, const float *a, const float *b, float c,
float *d)
{
    dim3 threads(128);
    dim3 blocks((n + threads.x - 1) / threads.x);

    calc_kernel<<<blocks, threads>>>(n, a, b, c, d);
    cudaThreadSynchronize();
}

int main()
{
    ...

    float *a_d, *b_d, *d_d;
    cudaMalloc(&a_d, n*sizeof(float));
    cudaMalloc(&b_d, n*sizeof(float));
    cudaMalloc(&d_d, n*sizeof(float));

    cudaMemcpy(a_d, a, n*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(b_d, b, n*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(d_d, d, n*sizeof(float), cudaMemcpyDefault);

    calc(n, a_d, b_d, c, d_d);

    cudaMemcpy(d, d_d, n*sizeof(float), cudaMemcpyDefault);
    ...
}
```

# OpenACCコードのコンパイル

## ■ PGIコンパイラによるコンパイル

✓ ReedbushではOpenACCはPGIコンパイラで利用できます。

```
$ module load pgi/17.1
$ pgcc -O3 -acc -Minfo=accel -ta=tesla,cc60 -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,cc60:**

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 6.0 (cc60) 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 pgi/17.1
$ make
```

# 簡単なOpenACCコード

- サンプルコード: openacc_basic/
  - ✓ OpenACC指示文 kernels, data, loop を利用したコード
  - ✓ 計算内容は簡単な四則演算

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    for (unsigned int j=0; j<ny; j++) {
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

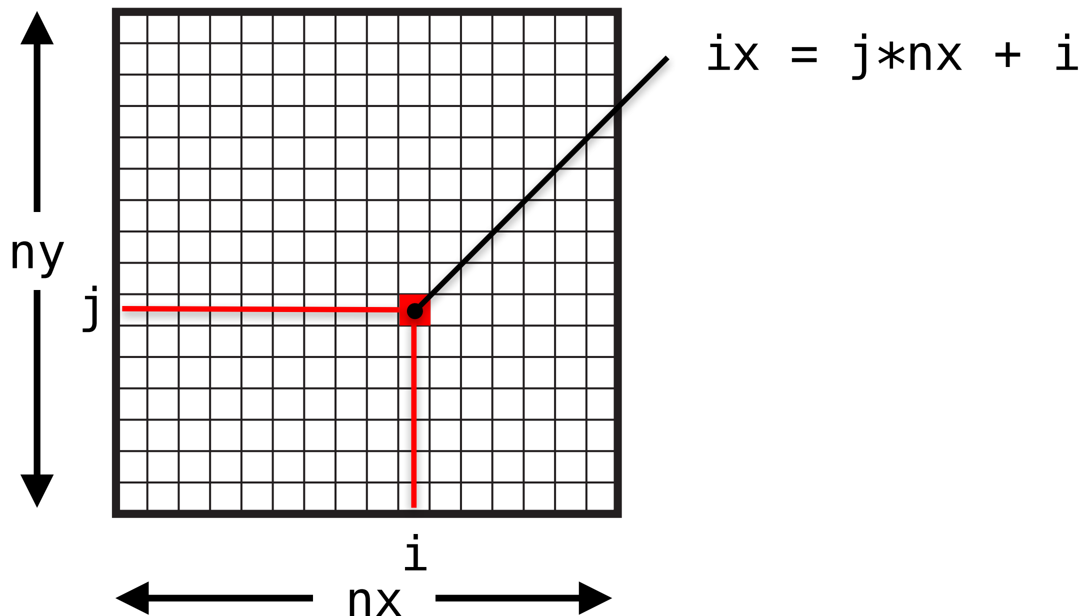
- ✓ ソースコード

| | |
|---|---|
| openacc_basic/01_original | CPUコード。 |
| openacc_basic/02_kernels | OpenACCコード。上にkernels/loop指示文を追加。 |
| openacc_basic/03_data | OpenACCコード。上にdata指示文を明示的に追加。 |
| openacc_basic/04_present | OpenACCコード。上でpresent指示節を使用。 |
| openacc_basic/05_reduction | OpenACCコード。上にreduction指示節を使用。 |

# 配列のインデックス計算

- サンプルコード: openacc_basic/
  - ✓ OpenACC指示文 kernels, data, loop を利用したコード
  - ✓ 計算内容は簡単な四則演算

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    for (unsigned int j=0; j<ny; j++) {
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```



# 簡単なOpenACC: CPUコード

- 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_basic/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    19.886 [sec]
```

？の数字はジョブごとに変わります。

答えは常に3000.0

openacc_basic/01_original

- 計算内容
  - ✓ 配列 a、b、cをそれぞれ 1.0, 2.0, 0.0 で初期化
  - ✓ calc関数内で c += a *b を nt(=1000)回実行。
  - ✓ この実行時間を測定

# 簡単なOpenACC: kernels 指示文（1）

- 02_kernelsコード: calc関数
  - ✓ CPUコードにkernels 指示文の追加

openacc_basic/02_kernels

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
    for (unsigned int j=0; j<ny; j++) {
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

allocate, H -> D

D->H, deallocate

  - ✓ kernels 指示文では data 指示文が使える
  - ✓ 上の場合は、copy を指定
    - ✓ カーネル前後でGPUとCPU間のメモリ転送が行われる。

# 簡単なOpenACC: kernels 指示文（２）

- 02_kernelsコード:初期化
  - ✓ CPUコードにkernels 指示文の追加

openacc_basic/02_kernels

```
int main(int argc, char *argv[])
{
...
#pragma acc kernels copyout(b[0:n], c[0:n])
    {
        for (unsigned int i=0; i<n; i++) {
            b[i] = b0;
        }
        for (unsigned int i=0; i<n; i++) {
            c[i] = 0.0;
        }
    }
...
}
```

allocate

b0 はスカラ変数のため自動的に各スレッドへコピーが渡される。

D->H, deallocate

- ✓ kernels 指示文では data 指示文が使える
- ✓ 上の場合は、copyout を指定
  - ✓ GPU上で値を初期化するため、CPUからGPUへのコピーは不要。

# 簡単なOpenACC: kernels 指示文（3）

- コンパイル
  - ✓ データの独立性がコンパイラにはわからず、並列化されない。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
calc:
     13, Generating copy(a[:n],c[:n],b[:n])
     14, Complex loop carried dependence of a-> prevents parallelization
         Loop carried dependence due to exposed use of c[:n] prevents parallelization
         Complex loop carried dependence of c->,b-> prevents parallelization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         14, #pragma acc loop seq
         15, #pragma acc loop seq
     15, Complex loop carried dependence of a->,c->,b-> prevents parallelization
         Loop carried dependence due to exposed use of c[:i1+n] prevents parallelization
main:
     43, Generating copyout(c[:n],b[:n])
     45, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         45, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     48, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60   main.o   -o run
```

# 簡単なOpenACC: loop 指示文（1）

- 03_loopコード
  - ✓ 02_kernelsコードにloop independent の追加

openacc_basic/03_loop

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
#pragma acc loop independent
    for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

```
// main 関数内
#pragma acc kernels copyout(b[0:n], c[0:n])
    {
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            b[i] = b0;
        }
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            c[i] = 0.0;
        }
    }
```

# 簡単なOpenACC: loop 指示文（2）

openacc_basic/03_loop

- コンパイル
  - ✓ ループが並列化され、カーネルが生成された。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
calc:
     13, Generating copy(a[:n],c[:n],b[:n])
     15, Loop is parallelizable
     17, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
main:
     45, Generating copyout(c[:n],b[:n])
     48, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     52, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         52, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

# 簡単なOpenACC: loop 指示文（3）

openacc_basic/03_loop

- 03_loopコードの実行
  - ✓ 答えは正しいが、実行時間が大変長い。

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =   70.414 [sec]
```

  - ✓ ソースコードをみると、calc関数でカーネル前後にGPUとCPU間のデータ転送が発生する。これが性能低下させている。

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])       allocate, H -> D
#pragma acc loop independent
    for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }                                                   D->H, deallocate
}
```

# 簡単なOpenACC: data指示文（１）

- 04_dataコード
  - ✓ 03_loopにdata指示文追加

openacc_basic/04_data

```
// main関数内
#pragma acc data copyin(a[0:n]) create(b[0:n]) copyout(c[0:n])
    {
#pragma acc kernels copyout(b[0:n], c[0:n])
        {
#pragma acc loop independent
            for (unsigned int i=0; i<n; i++) {
                b[i] = b0;
            }
#pragma acc loop independent
            for (unsigned int i=0; i<n; i++) {
                c[i] = 0.0;
            }
        }

        for (unsigned int icnt=0; icnt<nt; icnt++) {
            calc(nx, ny, a, b, c);
        }
    }
```

a: allocate, H -> D
b: allocate
c: allocate

present として振舞う。

a: deallocate
b: deallocate
c: D->H, deallocate

- ✓ copy/copyin/copyout/create は既にデバイス上確保されているデータに対しては何もしない。present として振舞う。（OpenACC2.5以降）
- ✓ 配列 a, b, c は利用用途に合わせた指示節を指定。

# 簡単なOpenACC: data指示文（２）

- 04_dataコードの実行
  - ✓ 答えは正しく、速度が上がった。

openacc_basic/04_data

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    1.174 [sec]
```

# 簡単なOpenACC: present指示節

- 05_presentコード
  - ✓ 04_dataコードで present 指示節を使用

openacc_basic/05_present

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels present(a, b, c)
#pragma acc loop independent
    for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
```

present へ変更

```
// main 関数内
#pragma acc kernels
    {
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            b[i] = b0;
        }
```

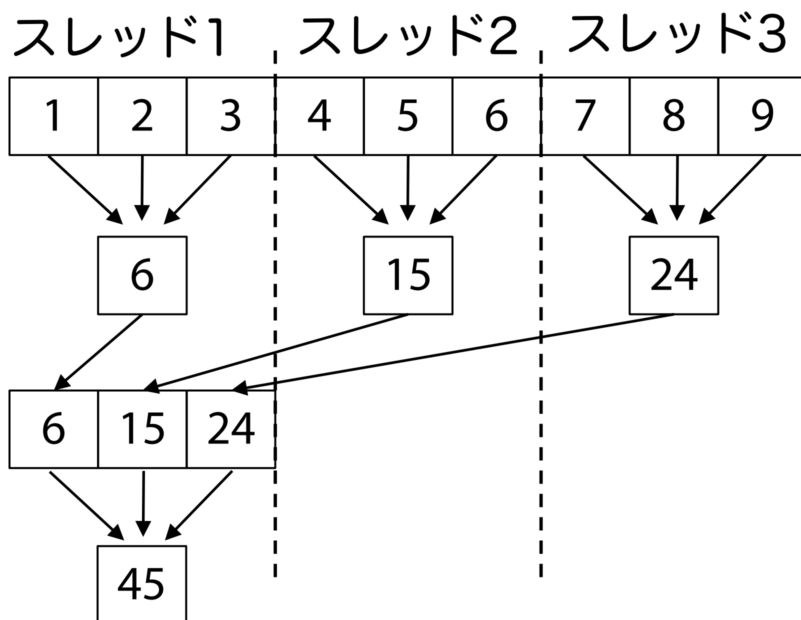
指示節を削除

- ✓ データ転送の振る舞いは変化しないため、性能変化はなし。
- ✓ present ではメモリ確保、データ転送をしないため、配列サイズの指定は不要。
- ✓ コードとしては見通しがよい。

# リダクション計算（1）

- リダクション計算
  - ✓ 配列の全要素から一つの値を抽出
  - ✓ 総和、総積、最大値、最小値など
  - ✓ 出力が一つのため、並列化に工夫が必要（CUDAでの実装は煩雑）

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




1. 各スレッドが担当する領域をリダクション
2. 一時配列に移動
3. 一時配列をリダクション
4. 出力を得る

# リダクション計算(2)

- loop 指示文に reduction 指示節を指定
  - ✓ reduction 演算子と変数を組み合わせて指定

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

- Reduction 指示節
  - ✓ **acc loop reduction(+:sum)**
    - ✓ 演算子と対象とする変数（スカラー変数）を指定する。
- 利用できる主な演算子と初期値
  - ✓ 演算子: +, 初期値: 0
  - ✓ 演算子: *, 初期値: 1
  - ✓ 演算子: max, 初期値: least
  - ✓ 演算子: min, 初期値: largest

# 簡単なOpenACC: reduction指示節（１）

- 06_reductionコード
  - ✓ 05_presentコードで reductionを使用

openacc_basic/06_reduction

```
// main 関数内
        for (unsigned int icnt=0; icnt<nt; icnt++) {
            calc(nx, ny, a, b, c);
        }

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

  - ✓ data 指示文で c を create に変更。
- 06_reductionコード
  - ✓ リダクションコードが生成された。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
(省略)
main:
(省略)
67, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         67, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:sum)
```

# 簡単なOpenACC: reduction指示節（2）

- 06_reductionコードの実行
  - ✓ 答えは正しく、速度が上がった。
  - ✓ 配列 c の転送が削減されたこと、リダクションがGPU上で行われることによる性能向上。

openacc_basic/06_reduction

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    1.089 [sec]
```

# OpenACC化のステップのまとめ

- OpenACC化のための3つの指示文の適用
  - ✓ kernels 指示文を用いてGPUで実行する領域を指定
  - ✓ data 指示文を用い、ホスト-デバイス 間の通信を最適化
  - ✓ loop 指示文を用い、並列処理の指定

```
#pragma acc data copyin(a[0:n]) create(b[0:n], c[0:n])
    {

#pragma acc kernels
        {
#pragma acc loop independent
            for (unsigned int i=0; i<n; i++) {
                b[i] = b0;
            }
#pragma acc loop independent
            for (unsigned int i=0; i<n; i++) {
                c[i] = 0.0;
            }
        }

        for (unsigned int icnt=0; icnt<nt; icnt++) {
            calc(nx, ny, a, b, c);
        }

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

openacc_basic/06_reduction

# OPENACC入門実習

# 実習

- 3次元拡散方程式のOpenACC化
  - ✓ サンプルコード：openacc_diffusion/01_original
- 3次元拡散方程式のCPUコードにOpenACC の kernels, data, loop 指示文を追加し、GPUで高性能で実行しましょう。

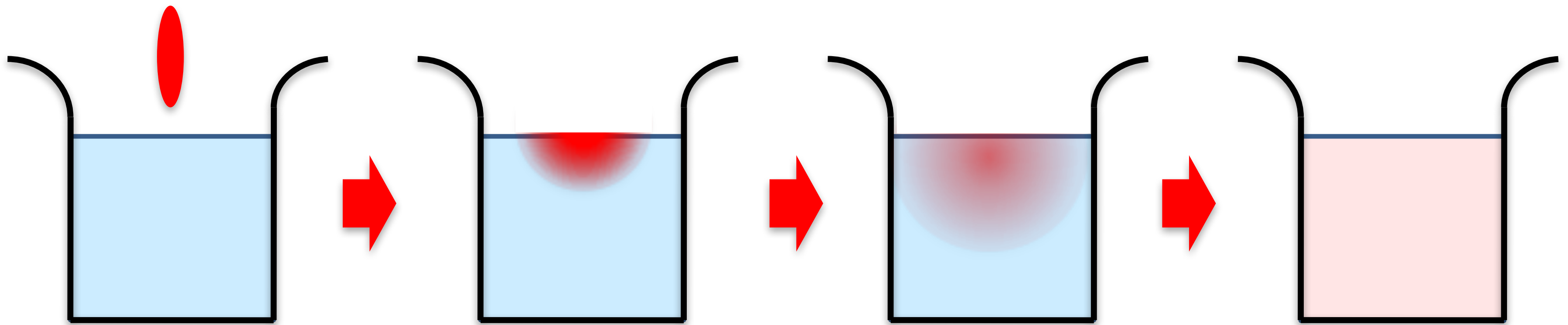
diffusion.c, diffusion3d 関数内

```
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 + 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 = k == nz - 1 ? ix : ix + nx*ny;
            const int km = 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_diffusion/01_original

# 拡散現象シミュレーション（1）

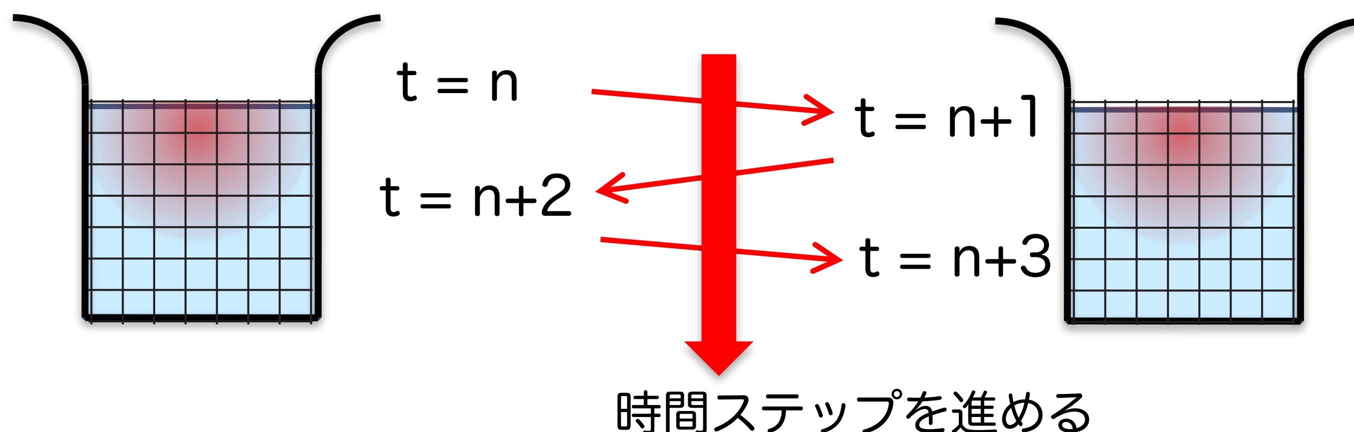
- 拡散現象
  - ✓ コップの中に赤インクを落とすと水中で拡がる
  - ✓ 次第に拡散し赤インクは拡がり、最後は均一な色になる。



- 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

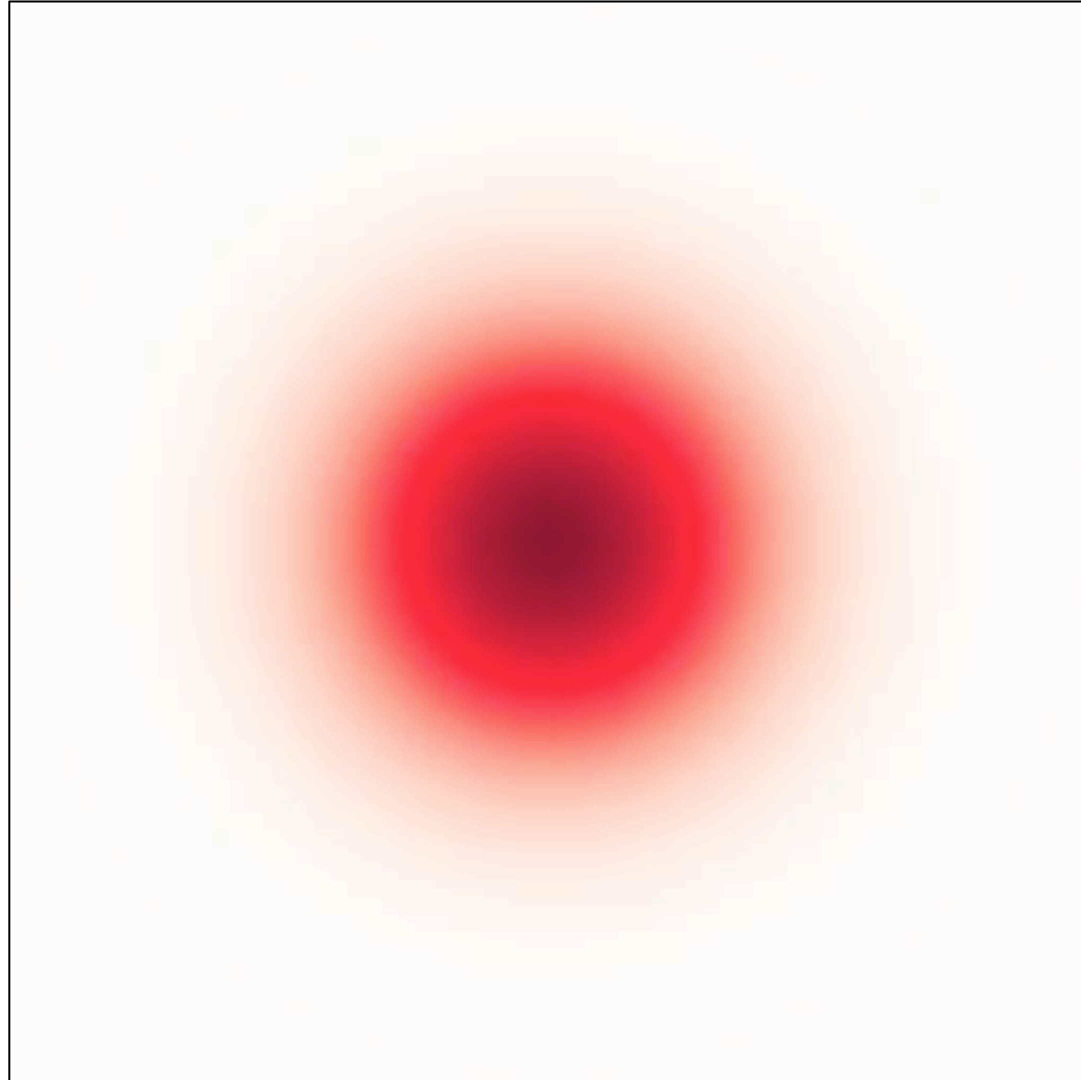
| j \ i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 2 | 8 | 2 | 0 |
| 3 | 1 | 8 | 20 | 8 | 1 |
| 4 | 0 | 2 | 8 | 2 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 |

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



# CPUコード

- CPUコードのコンパイルと実行

```
$ cd openacc_diffusion/01_original
$ make
$ qsub ./run.sh
# cat run.sh.o??????
time(   0) = 0.00000
time( 100) = 0.00610
time( 200) = 0.01221
...
time(1000) = 0.06104
time(1100) = 0.06714
time(1200) = 0.07324
time(1300) = 0.07935
time(1400) = 0.08545
time(1500) = 0.09155
time(1600) = 0.09766
Time =   20.564 [sec]
Performance=    2.17 [GFlops]
Error[128][128][128] = 4.556413e-06
```

← 実行性能

← 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    = pgcc
CXX   = pgc++
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): kernels

- diffusion3d関数に kernelsを追加しましょう

```
#pragma acc kernels copyin(f[0:nx*ny*nz]) copyout(fn[0:nx*ny*nz])
    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 + 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 = k == nz - 1 ? ix : ix + nx*ny;
                const int km = 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

make して実行してみましょう。

# OpenACC化(2): loop

- diffusion3d関数に loopを追加しましょう

```
#pragma acc kernels copyin(f[0:nx*ny*nz]) copyout(fn[0:nx*ny*nz])
#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 + 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 = k == nz - 1 ? ix : ix + nx*ny;
                const int km = 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

高速化よりも、まずは正しい計算を行うコードを保つことが大事です。
末端の関数から修正を進めます。

make してジョブ投入 qsub ./run.shしてみましょう。遅いですが実行できます。

# OpenACC化(3): データ転送の最適化(1)

- diffusion3d関数で present とし、main関数で data を追加

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 + 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 = k == nz - 1 ? ix : ix + nx*ny;
                const int km = 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;
}
```

なお、 present にしなくても期待通りに動作します。

# OpenACC化(4): データ転送の最適化(2)

- diffusion3d関数で present とし、main関数で data を追加

main.c, main 関数内

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

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

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

            swap(&f, &fn);

            time += dt;
        }

        elapsed_time = get_elapsed_time();
    }
```

copy/create など適切なものを選びます。

make して実行してみましょう。 どのくらいの実行性能が出ましたか？

OpenACC化の例は、openacc_diffusion/02_openacc

# PGI_ACC_TIME によるOpenACC 実行の確認

- PGI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PGI_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PGI_ACC_TIME を1に設定し、プログラムを実行する。

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

- Reedbush でジョブに環境変数PGI_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
...
 . /etc/profile.d/modules.sh
module load pgi/17.1
export PGI_ACC_TIME=1

./run
```

サンプルコードは、 openacc_diffusion/03_openacc _pgi_acc_time

# PGI_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.e??????
Accelerator Kernel Timing data
/lustre/pz0115/z30115/lecture_samples/openacc_diffusion/03_open
acc_pgi_acc_time/main.c
  main  NVIDIA  devicenum=0
    time(us): 6,359
    38: data region reached 2 times
        38: data copyin transfers: 1
             device time(us): total=3,327 max=3,327 min=3,327 avg=3,327
        55: data copyout transfers: 1
             device time(us): total=3,032 max=3,032 min=3,032 avg=3,032
/lustre/pz0115/z30115/lecture_samples/openacc_diffusion/03_open
acc_pgi_acc_time/diffusion.c
  diffusion3d  NVIDIA  devicenum=0
    time(us): 101,731
    19: compute region reached 1638 times
        25: kernel launched 1638 times
            grid: [4x128x32]  block: [32x4]
             device time(us): total=101,731 max=64 min=62 avg=62
            elapsed time(us): total=136,255 max=540 min=81 avg=83
    19: 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

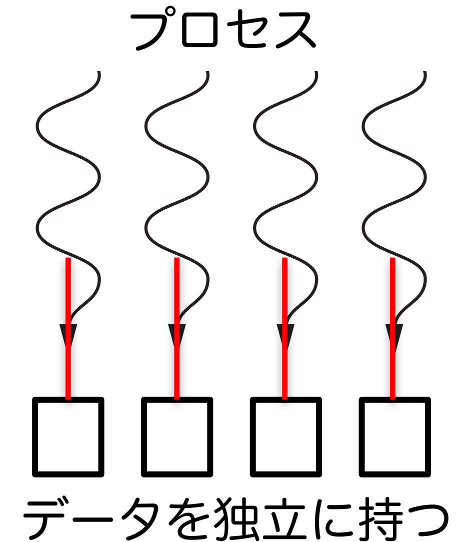
# MPI復習

# MPI

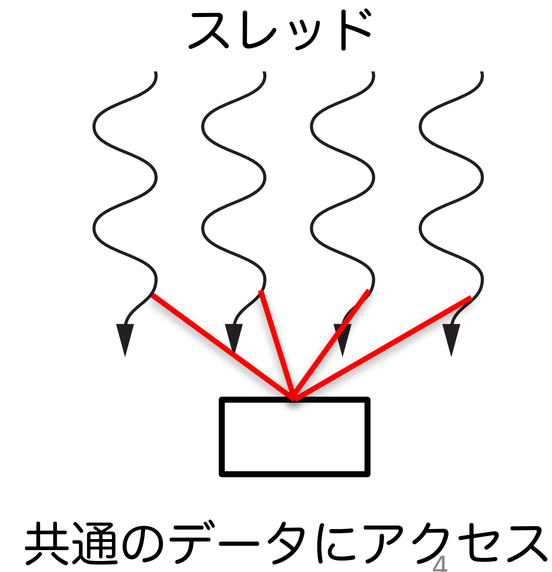
- MPI (Message Passing Interface)
- メッセージパッシング用のライブラリの規格の一つ
  - ✓ C, C++, Fortran に対応
  - ✓ コンパイラの規格、特定のソフトウェアやライブラリではない。
- 分散メモリ並列計算機での並列実行に向く
  - ✓ 分散メモリモデル
  - ✓ プロセス間のデータ移動はメッセージ
- 大規模計算が可能
  - ✓ 1プロセッサにおけるメモリサイズを超え、計算時間を短縮
  - ✓ 利用可能な並列度はノード数を超えられる

# MPIとOpenMP

- MPI
  - ✓ 分散メモリモデル
  - ✓ プロセス間のデータ移動はメッセージ
  - ✓ Critical section の代わりにメッセージによる同期
  - ✓ 利用可能な並列度はノード数を超えられる
  - ✓ 逐次プログラムからプログラム構造が変更することが多い




- OpenMP
  - ✓ 共有メモリモデル
  - ✓ スレッド間のデータ移動は共有変数で行う
  - ✓ Critical section による排他処理
  - ✓ メモリ共有できるノード内並列に限られる
  - ✓ 逐次プログラムに必要な部分に指示文（#pragma omp）を入れ簡単に高速化できる（可能性がある）



# MPIの実装

- MPICH
  - ✓ 米国アルゴンヌ国立研究所が開発
- MVAPICH
  - ✓ 米国オハイオ州立大学で開発
  - ✓ MPICHをベース
  - ✓ Reedbush では、以下などでロードできる

```
$ module mvapich2/2.2/{gnu,intel,pgi}
```

- OpenMPI
  - ✓ オープンソース
  - ✓ Reedbush では、以下などでロードできる

```
$ module openmpi/1.10.7/{gnu,intel,pgi}
```

- ベンダMPI
  - ✓ Intel MPI など
  - ✓ 大抵、上のどれかがベース

# MPIプロセスとメモリ

- SPMD(Single Program Multiple Data)
  - ✓ 複数のプロセスが同一プログラムを実行
- プロセスごとに別のメモリ空間
  - ✓ すべての変数は各プロセスで別々
  - ✓ 協調動作させるには明示的に通信する
- プロセスには固有の番号（ランク, rank）がつく

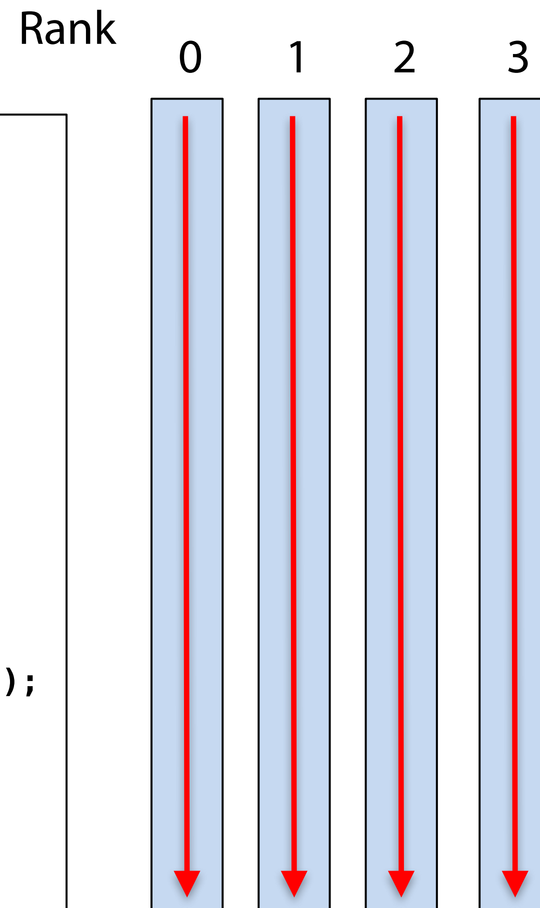
```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);     // 全プロセス数取得
MPI_Comm_rank(MPI_COMM_WORLD, &rank);       // ランク取得
```

- ✓ 0 <= rank < nprocs
- ✓ MPI_COMM_WORLD は全プロセスを含むコミュニケータ。
- ✓ コミュニケータは操作対象となるプロセッサ群を定める。
- ✓ メッセージの送信先、受信先は rank で指定

# MPIプログラムの大きな構造

mpi_basic/01_hello_mpi

```
#include <stdio.h>
#include <mpi.h>


int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int nprocs = 1;
    int rank   = 0;

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    fprintf(stdout, "rank = %d, nprocs = %d¥n", rank, nprocs);

    MPI_Finalize();

    return 0;
}
```




変数は各プロセスで独立

MPIの初期化: MPI_Init

MPIの終了: MPI_Finalize

をプログラムの先頭と最後で行う。

# MPIコードのコンパイル

## ■ MPIコードのコンパイル

✓ ここではPGIコンパイラとOpenMPIを利用します。

```
$ module load pgi/17.1
$ module openmpi/1.10.7/pgi
$ mpicc –O3 –c main.c
```

**mpicc**

MPIコードをコンパイラできる。オプションは元のコンパイラ（今回はPGI）と同じ。

## ■ 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 pgi/17.1 openmpi/1.10.7/pgi
$ make
```

# MPIプログラムの実行

- ジョブスクリプト

```
$ cat run.sh
#! /bin/sh
#PBS -q h-tutorial
#PBS -l select=1:mpiprocs=2:ompthreads=0
#PBS -W group_list=gt00
#PBS -l walltime=00:05:00

cd $PBS_O_WORKDIR

. /etc/profile.d/modules.sh
module load pgi/17.1
module load openmpi/1.10.7/pgi

mpirun -np 2./run
```

`select=1`:

使用するノード数を指定。この場合は1ノード。

`mpiprocs=2`:

各ノードに割り当てるMPIプロセス数。この場合は2プロセス。

`mpirun -np 2 ./run`

-np に全並列数（全MPIプロセス数）を指定。select * mpiprocs となることが多い。./run が実行したいプログラム。

# MPI関数

- システム制御
  - ✓ MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Finalize, MPI_Barrier
- 1対1通信
  - ✓ ブロッキング通信
    - ✓ MPI_Send, MPI_Recv
  - ✓ ノンブロッキング通信
    - ✓ MPI_Isend, MPI_Irecv, MPI_Wait
- 集団通信
  - ✓ MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather,  MPI_Alltoall, MPI_Reduce, MPI_Allreduce
- 時間計測
  - ✓ MPI_Wtime

# C言語とFortranインターフェースの違い（１）

- エラーコードの扱い
  - ✓ C言語版は、整数の戻り値(err)

```
err = MPI_Xxxx(...);
```

  - ✓ Fortran版は、最後に整数引数 err を渡して取得

```
call MPI_XXXX(..., err)
```

- システム用配列の確保の方法
  - ✓ C言語版

```
MPI_Status status;
```

  - ✓ Fortran版は、最後に整数引数 err を渡して取得

```
integer status(MPI_STATUS_SIZE)
```

# C言語とFortranインターフェースの違い（２）

- MPIデータ型
  - C言語版
  - ✓ MPI_CHAR（文字型）、MPI_INT（整数型）、MPI_FLOAT（単精度実数型）、MPI_DOUBLE（倍精度実数型）
  - Fortran版
  - ✓ MPI_CHARACTER（文字型）、MPI_INTEGER（整数型）、MPI_REAL （単精度実数型）、MPI_DOUBLE_PRECISION（倍精度実数型）、MPI_COMPLEX（複素数型）

以降はC言語インターフェースで説明します。

# 基礎的なMPI関数：MPI_Recv

```
err = MPI_Recv(recvbuf, count, datatype, src, tag, comm, status);
```

src ランクを持つプロセスから送られた識別番号 tag で連続した count 個のメッセージを recvbuf に受信する。メッセージが到着するまで待たされる（ブロッキング）。

- **recvbuf** 任意 受信領域の先頭アドレス。
- **count** 整数 受信するデータの個数。
- **datatype** MPIデータ型 受信領域のデータ型。
  - ✓ MPI_CHAR（文字型）、MPI_INT（整数型）、MPI_FLOAT(単精度実数型）、MPI_DOUBLE（倍精度実数型）など
- **src** 整数 送信元プロセスのランク。
- **tag** 整数 受信したいメッセージの識別番号。
- **comm** コミュニケータ コミュニケータ。
  - ✓ 通常は MPI_COMM_WORLD を指定する。
- **status** MPI_Status型 受信状況に関する情報が入る。
  - ✓ 必ず専用の型で受け取る。
- **err** 整数 エラーコードが入る。

# 基礎的なMPI関数：MPI_Send

```
err = MPI_Send(sendbuf, count, datatype, dst, tag, comm);
```

sendbuf から連続した count 個のメッセージを、識別番号 tag をつけて、dst ランクをもつプロセスへ送信する。メッセージ送受信が完了するまで待たされる（ブロックキング）。

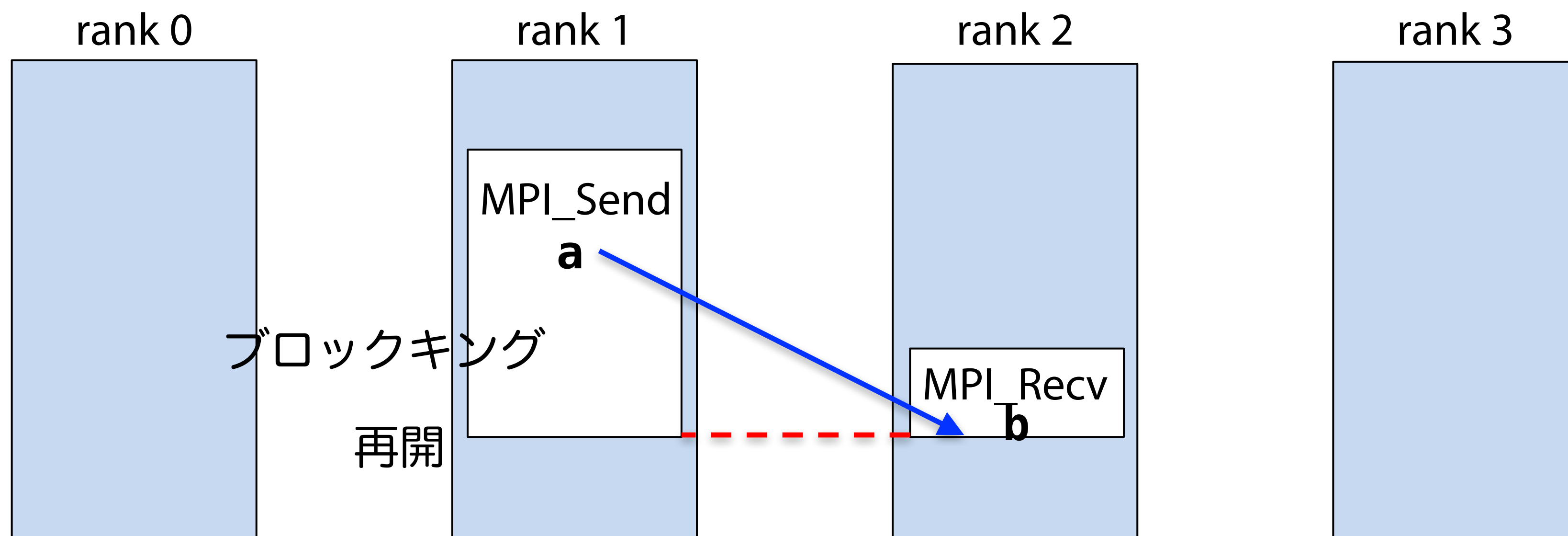
- **sendbuf**　任意　送信領域の先頭アドレス。
- **count**　整数　送信するデータの個数。
- **datatype**　MPIデータ型　送信領域のデータ型。
- **dst**　整数　送信先プロセスのランク。
- **tag**　整数　送信するメッセージの識別番号。
- **comm**　コミュニケータ　コミュニケータ。
- **err**　整数　エラーコードが入る。

なお、MPI_Send/MPI_Recvはブロックキングするため、注意深く順番を並べて関数を呼ばないとプログラムがデッドロックする。

# 1対1通信（ブロッキング）

mpi_basic/02_recv_send

```
const int n = 16;
int a[n];
int b[n];
for (int i=0; i<n; i++) {
    a[i] = i;
    b[i] = 0;
}
if (rank == 1) {
    MPI_Send(a, n, MPI_INT, 2, 100, MPI_COMM_WORLD);
} else if (rank == 2) {
    MPI_Status status;
    MPI_Recv(b, n, MPI_INT, 1, 100, MPI_COMM_WORLD, &status);
}
```



# 基礎的なMPI関数：MPI_Irecv

```
err = MPI_Irecv(recvbuf, count, datatype, src, tag, comm, request);
```

MPI_Recvのノンブロッキング版。送受信が完了しなくても関数は直ちに終了する。受信を完了したい箇所で request を渡して MPI_Wait を呼ぶ。

- **recvbuf** 任意 受信領域の先頭アドレス。
- **count** 整数 受信するデータの個数。
- **datatype** MPIデータ型 受信領域のデータ型。
- **src** 整数 送信元プロセスのランク。
- **tag** 整数 受信したいメッセージの識別番号。
- **comm** コミュニケータ コミュニケータ。
- **request** MPI_Request型 受信を要求したメッセージにつけられる識別子
- **err** 整数 エラーコードが入る。

MPI_Irecvが完了する段階では受信が終わっていない可能性があるため、受信情報を保持する MPI_Status は引数に取らない。

送受信が完了しなくても、 MPI_Irecv 直後から送受信と関係ない作業を再開できる。

# 基礎的なMPI関数：MPI_Isend

```
err = MPI_Isend(sendbuf, count, datatype, dst, tag, comm, request);
```

MPI_Sendのノンブロッキング版。送受信が完了しなくても関数は直ちに終了する。
送信を完了したい箇所で request を渡して MPI_Wait を呼ぶ。

- **sendbuf** 任意 送信領域の先頭アドレス。
- **count** 整数 送信するデータの個数。
- **datatype** MPIデータ型 送信領域のデータ型。
- **dst** 整数 送信先プロセスのランク。
- **tag** 整数 送信するメッセージの識別番号。
- **comm** コミュニケータ コミュニケータ。
- **request** MPI_Request型 送信を要求したメッセージにつけられる識別子
- **err** 整数 エラーコードが入る。

送受信が完了しなくても、 MPI_Isend 直後から送受信と関係ない作業を再開できる。

# 基礎的なMPI関数：MPI_Wait

```
err = MPI_Wait(request, status);
```

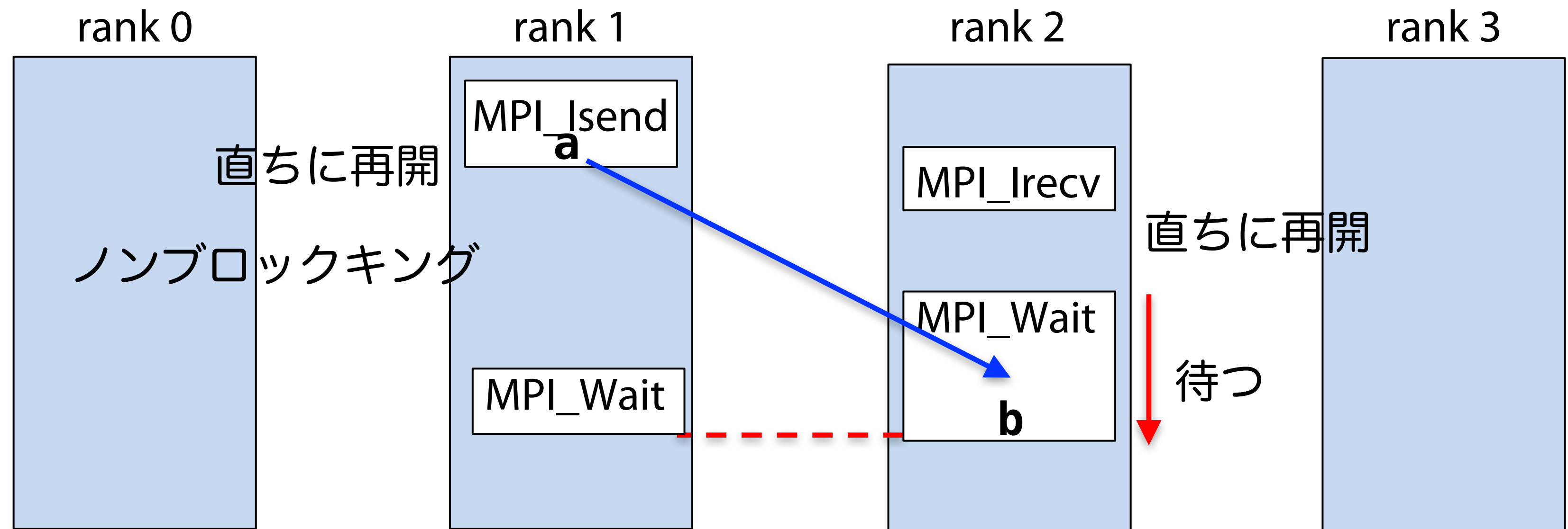
MPI_Isend/MPI_Irecvの送信/受信リクエストを受け取り、その送受信を待つ。

- **request**　MPI_Request型　送信を要求したメッセージにつけられる識別子
- **status**　MPI_Status型　受信状況に関する情報が入る。
- **err**　整数　エラーコードが入る。

# 1対1通信（ノンブロッキング）

mpi_basic/03_irecv_isend

```
for (int i=0; i<n; i++) {
    a[i] = i + rank * 10;
    b[i] = 0;
}
const int dst = (rank + 1) % nprocs;
const int src = (rank - 1 + nprocs) % nprocs;

MPI_Status sstatus, rstatus;
MPI_Request sreq, rreq;
MPI_Isend(a, n, MPI_INT, dst, 100, MPI_COMM_WORLD, &sreq);
MPI_Irecv(b, n, MPI_INT, src, 100, MPI_COMM_WORLD, &rreq);

MPI_Wait(&rreq, &rstatus);
MPI_Wait(&sreq, &sstatus);
```



# 1対1通信（ブロッキング、デッドロック）

mpi_basic/04_recv_send_deadlock

```
    //const int n = 16;
    const int n = 1024 * 1024;
    int a[n];
    int b[n];

    for (int i=0; i<n; i++) {
        a[i] = i + rank * 10;
        b[i] = 0;
    }

    const int dst = (rank + 1) % nprocs;
    const int src = (rank - 1 + nprocs) % nprocs;

    MPI_Status status;
    MPI_Send(a, n, MPI_INT, dst, 100, MPI_COMM_WORLD);
    MPI_Recv(b, n, MPI_INT, src, 100, MPI_COMM_WORLD, &status);
```

mpi_basic/03_irecv_isend を MPI_Send/MPI_Recvで実装すると、通信が循環しているため、デッドロックします。
MPI_Sendがバッファリングされる場合には実行されてしまいますが、されるかどうかは実装依存であるため、それを前提にしてはいけません。

# 基礎的なMPI関数：MPI_Gather

```
err = MPI_Gather(sendbuf, sendcount, sendtype,
                 recvbuf, recvcount, recvtype, root, comm);
```

comm 内のランク0から順に送られる sendbuf から連続した sendcount 個のメッセージをランク root の recvbuf へ recvcount個ずつ入れる。

- **sendbuf** 任意 送信領域の先頭アドレス。
- **sendcount** 整数 送信するデータの個数。
- **sendtype** MPIデータ型 送信領域のデータ型。
- **recvbuf** 任意 受信領域の先頭アドレス。
  - ✓ 原則、送受信領域は別の領域としなければならない。
- **recvcount** 整数 受信するデータの個数。
  - ✓ ここでは各ランクから送られてくる個数（同一の数）を指定する。各ランクから異なる数のデータは送れない。一般には sendcount と同数。
- **recvtype** MPIデータ型 受信領域のデータ型。
- **root** 整数 受信するランク。
- **comm** コミュニケータ コミュニケータ。
- **err** 整数 エラーコードが入る。

# 集団通信：収集

mpi_basic/05_gather

```
    const int n = 8;

    int a[n];
    int b[n];
    for (int i=0; i<n; i++) {
        a[i] = i + rank * 10;
        b[i] = 0;
    }

    MPI_Gather(a, 2, MPI_INT, b, 2, MPI_INT, 1, MPI_COMM_WORLD);
```

array a

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| rank 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| rank 1 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| rank 2 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| rank 3 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |

array b

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| 0 | 1 | 10 | 11 | 20 | 21 | 30 | 31 |
| | | | | | | | |
| | | | | | | | |

# 基礎的なMPI関数：MPI_Scatter

```
err = MPI_Scatter(sendbuf, sendcount, sendtype,
                  recvbuf, recvcount, recvtype, root, comm);
```

ランクroot の sendbuf から sendcount個ずつのメッセージを comm 内のランク0から順に全ランクの recvbuf へ recvcount 個入れる。

- **sendbuf** 任意 送信領域の先頭アドレス。
- **sendcount** 整数 送信するデータの個数。
  - ✓ ここでは各ランクへ送る個数（同一の数）を指定する。各ランクへ異なる数のデータは送れない。一般には recvcount と同数。
- **sendtype** MPIデータ型 送信領域のデータ型。
- **recvbuf** 任意 受信領域の先頭アドレス。
  - ✓ 原則、送受信領域は別の領域としなければならない。
- **recvcount** 整数 受信するデータの個数。
- **recvtype** MPIデータ型 受信領域のデータ型。
- **root** 整数 受信するランク。
- **comm** コミュニケータ コミュニケータ。
- **err** 整数 エラーコードが入る。

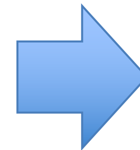
# 集団通信：分配

mpi_basic/06_scatter

```
    const int n = 8;

    int a[n];
    int b[n];
    for (int i=0; i<n; i++) {
        a[i] = i + rank * 10;
        b[i] = 0;
    }

    MPI_Scatter(a, 2, MPI_INT, b, 2, MPI_INT, 1, MPI_COMM_WORLD);
```

array a

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| rank 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| rank 1 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| rank 2 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| rank 3 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |



array b

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 11 | | | | | | |
| 12 | 13 | | | | | | |
| 14 | 15 | | | | | | |
| 16 | 17 | | | | | | |

# 基礎的なMPI関数：MPI_Reduce

```
err = MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);
```

comm 内の全ランクのsendbuf から連続した count 個のメッセージに対して、演算 op を適用し、ランク root の recvbuf へ送る。

- **sendbuf**　任意　送信領域の先頭アドレス。
- **recvbuf**　任意　受信領域の先頭アドレス。
- **count**　整数　送受信するデータの個数。
- **datatype**　MPIデータ型　送受信領域のデータ型。
- **op**　MPIオペランド　演算の種類
  - ✓ MPI_SUM（総和）、MPI_PROD（積）、MPI_MAX（最大）、MPI_MIN（最小）など
- **root**　整数　受信するランク。
- **comm**　コミュニケータ　コミュニケータ。
- **err**　整数　エラーコードが入る。

# 集団通信：リダクション

mpi_basic/07_reduce

```
const int n = 8;

int a[n];
int b[n];
for (int i=0; i<n; i++) {
    a[i] = i + rank * 10;
    b[i] = 0;
}

MPI_Reduce(a, b, 2, MPI_INT, MPI_SUM, 1, MPI_COMM_WORLD);
```

array a

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| rank 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| rank 1 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| rank 2 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| rank 3 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |

array b

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| 60 | 64 | | | | | | |
| | | | | | | | |
| | | | | | | | |

# 基礎的なMPI関数：MPI_Bcast

```
err = MPI_Bcast(buffer, count, datatype, root, comm);
```

ランク root の buffer から連続した count 個のメッセージを comm 内の全ランクの buffer へ送る。

- **buffer**　任意　送信および受信領域の先頭アドレス。
  - ✓ ランク root のプロセスにとっては送信領域、それ以外には受信領域
- **count**　整数　送受信するデータの個数。
- **datatype**　MPIデータ型　送受信領域のデータ型。
- **root**　整数　送信するランク。これ以外は受信となる。
- **comm**　コミュニケータ　コミュニケータ。
- **err**　整数　エラーコードが入る。

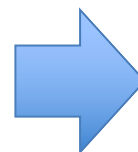
# 集団通信：ブロードキャスト

mpi_basic/08_bcast

```
    const int n = 8;

    int a[n];
    for (int i=0; i<n; i++) {
        a[i] = i + rank * 10;
    }

    MPI_Bcast(a, 4, MPI_INT, 1, MPI_COMM_WORLD);
```

array a

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| rank 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| rank 1 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| rank 2 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| rank 3 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |



array a

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 11 | 12 | 13 | 4 | 5 | 6 | 7 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 10 | 11 | 12 | 13 | 24 | 25 | 26 | 27 |
| 10 | 11 | 12 | 13 | 34 | 35 | 36 | 37 |

# その他関数と定数

- プロセス間同期

```
MPI_Barrier(comm);
```

  - ✓ コミュニケータ comm 内のすべてのプロセスで同期をとる。comm 内のプロセスがこれを呼ばないと次へ進まない。

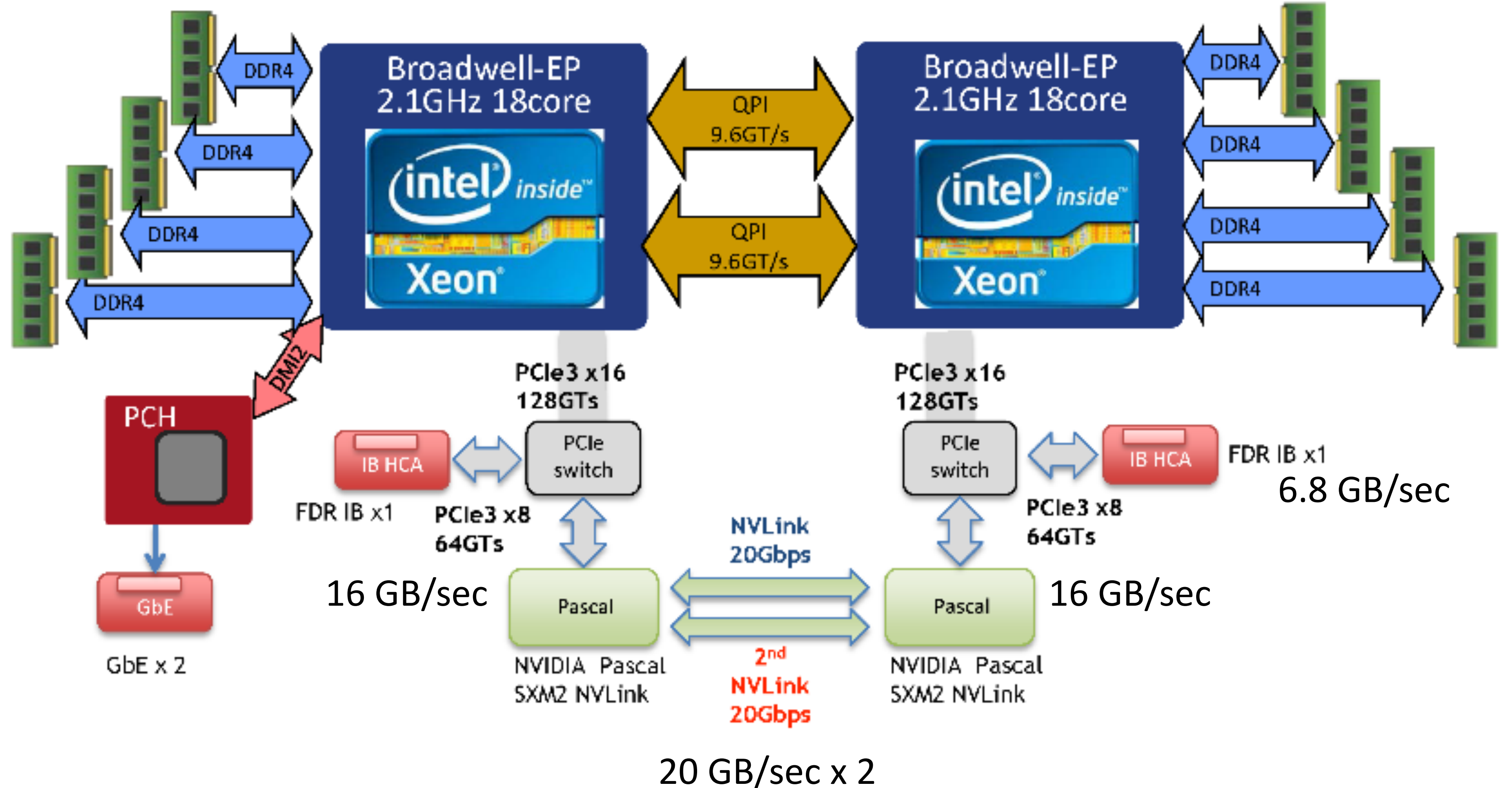
- MPI_PROC_NULL
  - ✓ MPI_Send の dst や MPI_Recv の src に MPI_PROC_NULLを指定すると、何も起こらず（バッファを書き変えずに）、関数が終了する。端領域などで通信をしない場合に、利用できる。

# OPENACCとMPIによる
# マルチGPUプログラミング

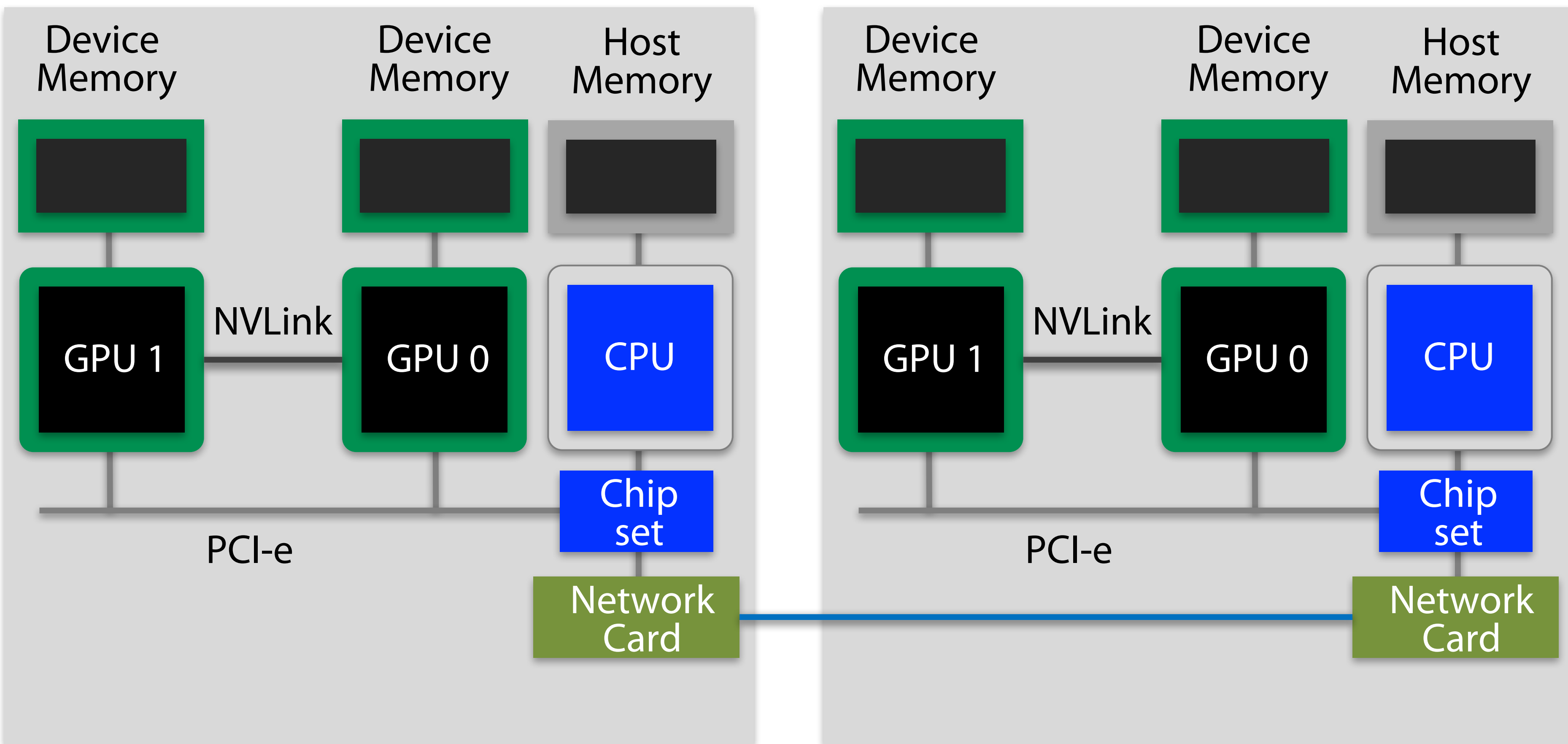
# マルチGPUコンピューティング

- 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
  - ✓ 複数GPUをMPIで並列化
  - ✓ 複数GPUをOpenMPで並列化
  - ✓ 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。

# Reedbush-H の計算ノード



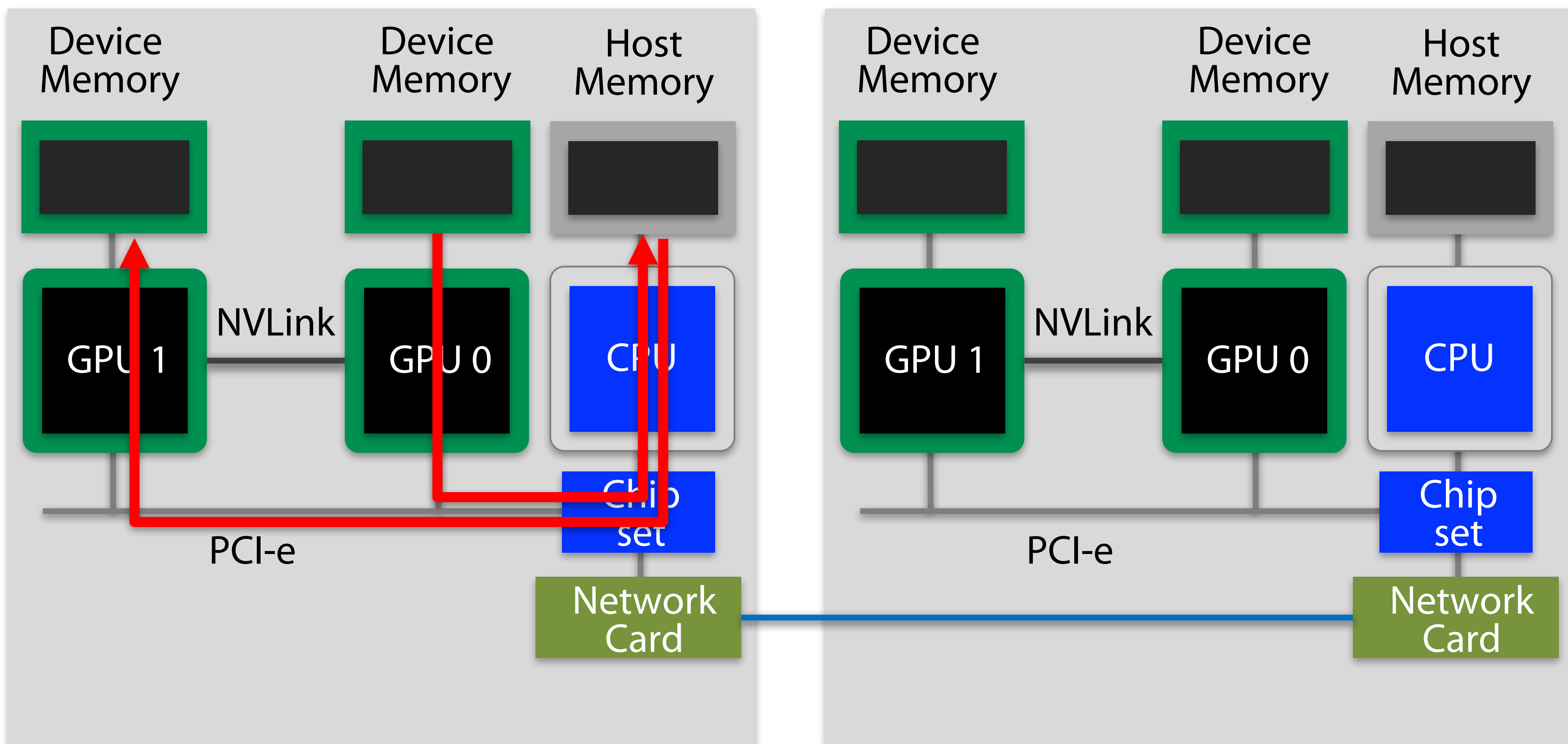
# Reedbush-H の計算ノード模式図




- GPU-GPU通信にはいくつかの種類がある

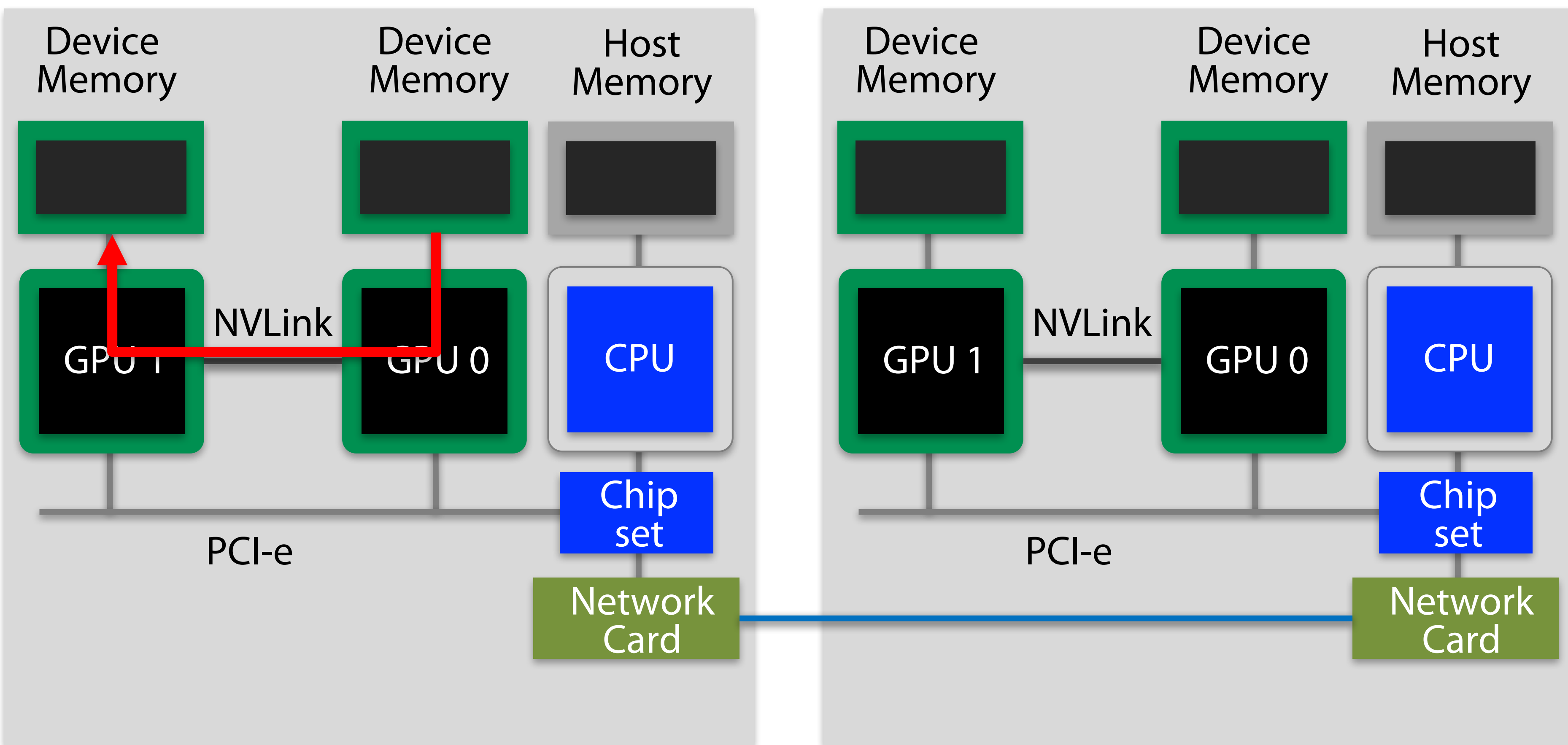
# ノード内GPU間通信（1）

- 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。



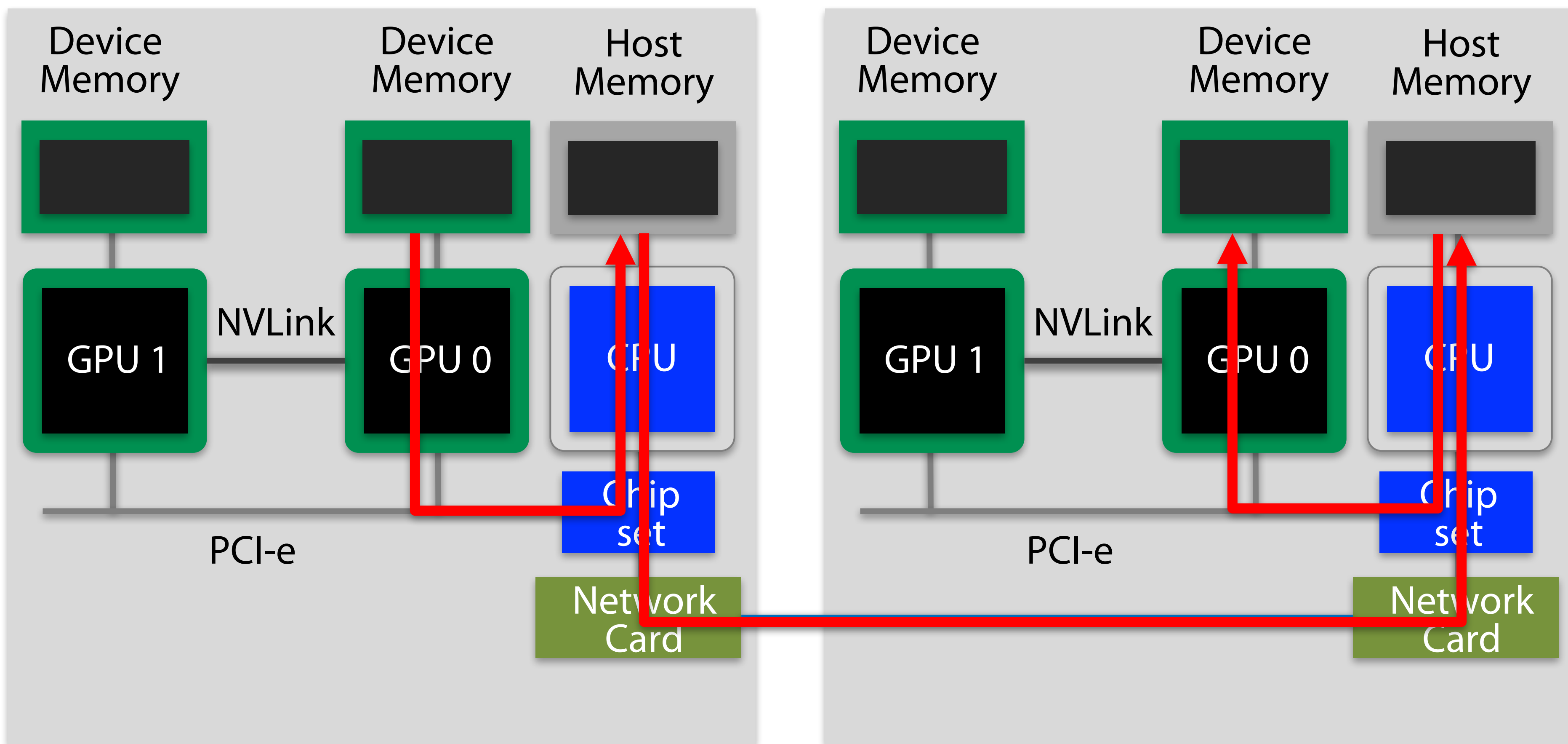
# ノード内GPU間通信（2）

- 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。ReedbushではGPU間をつなぐNVLink経由で高速通信する。



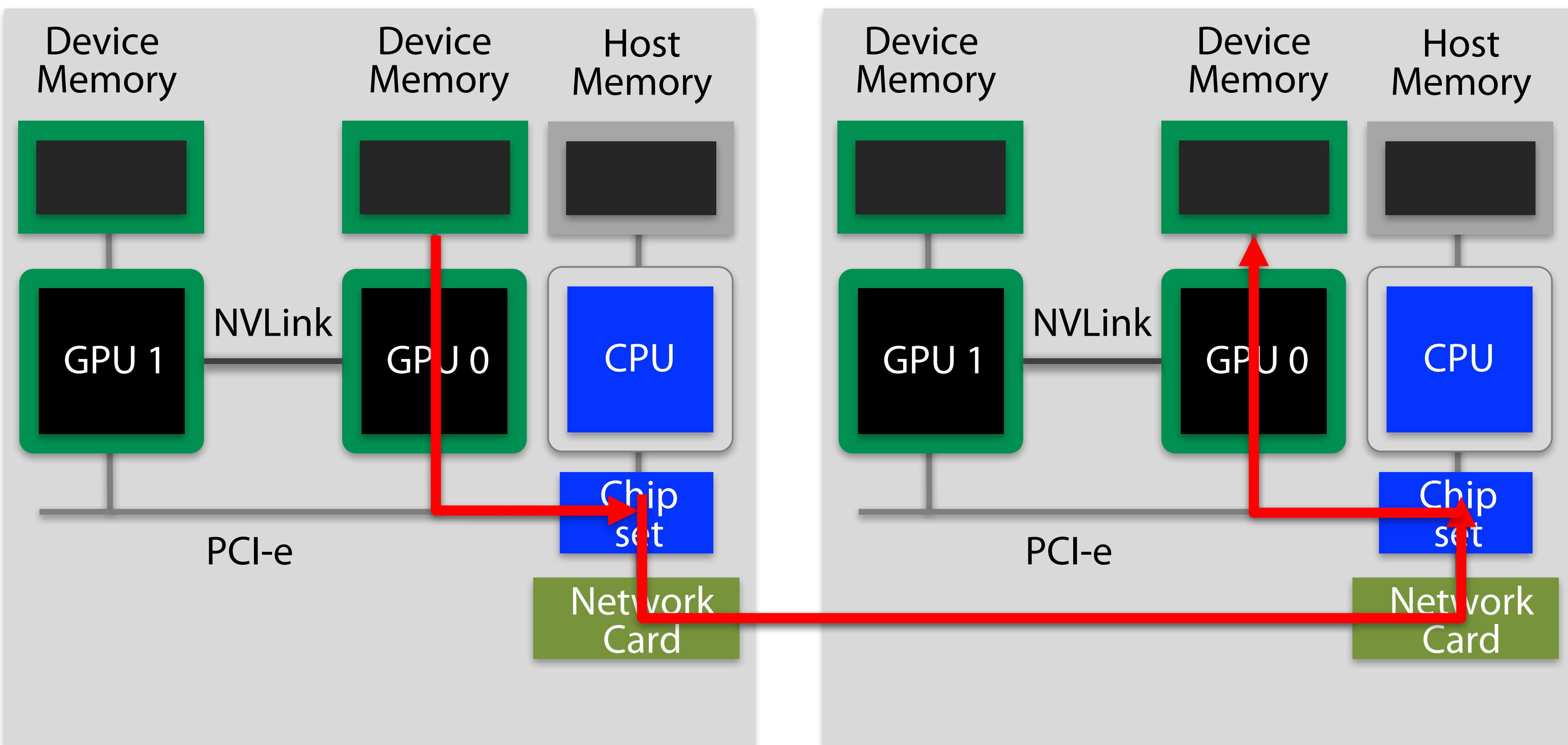
# ノード間のGPU間通信（1）

- 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。
  - ✓ Reedbush では、 CUDA-aware で GPUDirectRDMA(GDR)に対応した
    OpenMPI (モジュール名: openmpi-gdr/1.10.7/pgi)
    MVAPICH2-GDR (モジュール名: mvapich2-gdr/2.2/pgi)
  が利用できる。

# 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PGIコンパイラとOpenMPIを利用します。

```
$ module load pgi/17.1
$ module openmpi-gdr/1.10.7/pgi
$ mpicc -O3 -acc -Minfo=accel -ta=tesla,cc60 -c main.c
```

GDR対応であることに注意

**-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,cc60:**

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 6.0 (cc60) 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 pgi/17.1 openmpi-gdr/1.10.7/pgi
$ 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

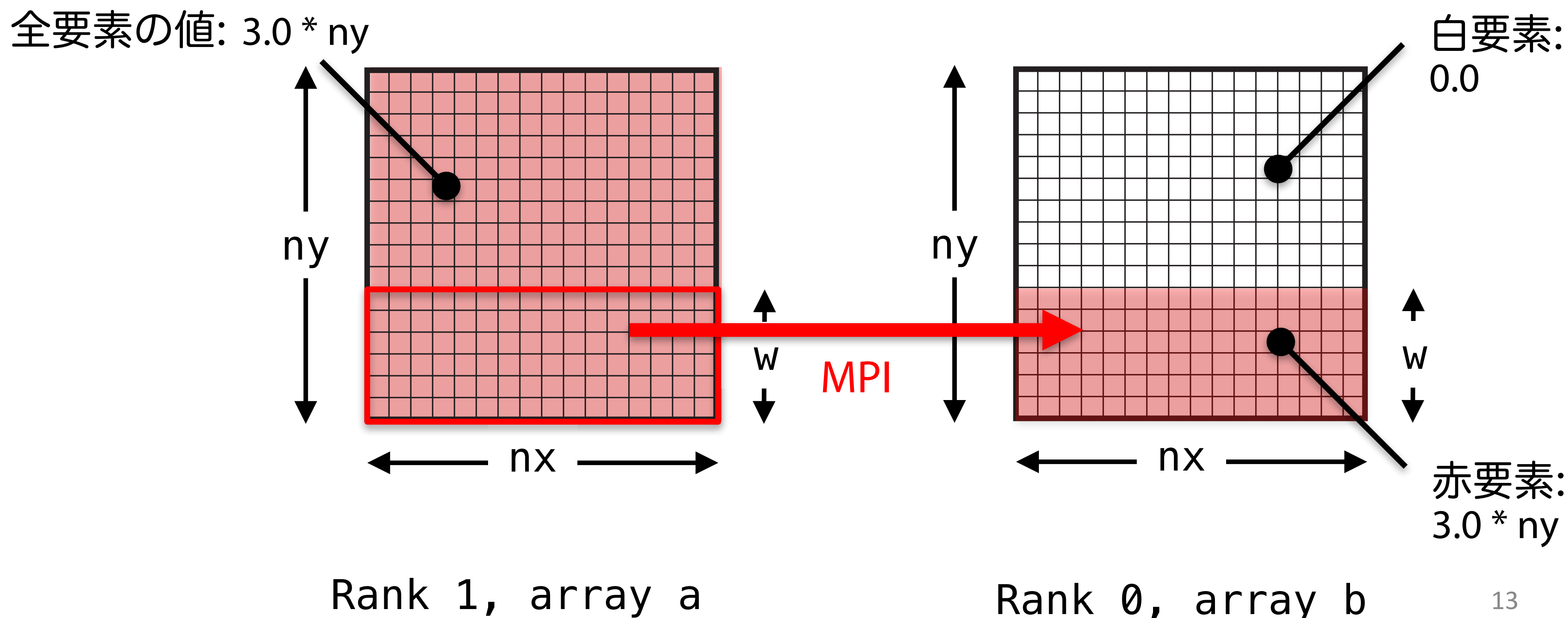
rank = 1 では a = 3.0 * ny

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 | MPI+OpenACCコード。上を2ノードにある2GPU使用。 |
| openacc_mpi_basic/06_gdr | 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
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 30.00
Time =     0.078 [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/sh
#PBS -q h-tutorial
#PBS -l select=1:mpiprocs=2:ompthreads=0
#PBS -W group_list=gt00
#PBS -l walltime=00:05:00

cd $PBS_O_WORKDIR

. /etc/profile.d/modules.sh
module load pgi/17.1
module load openmpi/1.10.7/pgi

mpirun -np 2 ./run
```

**select=1:**

使用するノード数を指定。この場合は1ノード。

**mpiprocs=2:**

各ノードに割り当てるMPIプロセス数。この場合は2プロセス。

**mpirun -np 2**

-np に全並列数（全MPIプロセス数）を指定。 select * mpiprocs となることが多い。./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

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

- ノード上のGPU数の取得

```
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

```
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。

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

D->H, deallocate

MPI(H)

MPI(H)

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
$ qsub ./run.sh
$ cat run.sh.o??????
num of GPUs = 2
mean = 30.00
Time =    0.233 [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])                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(H)
            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])                   H->D
        } else {
#pragma acc update host(a[0:w *nx])                     D->H
            MPI_Send(a, w * nx, MPI_FLOAT, dst_rank, tag, MPI_COMM_WORLD);
        }                                               MPI(H)

#pragma acc kernels copyin(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
$ qsub ./run.sh
$ cat run.sh.o??????
num of GPUs = 2
mean = 30.00
Time =    0.127 [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])              allocate, H->D
{
...
#pragma acc host_data use_device(a)
{
    cuda_func(a, n)
}
...
}                                          deallocate
...
```

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。

# 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])
    {
#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];
        }
    }
```

allocate

MPI(D)

MPI(D)

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
$ qsub ./run.sh
$ cat run.sh.o??????
num of GPUs = 2
Rank 0: hostname = a086, GPU num = 0
Rank 1: hostname = a086, GPU num = 1
mean = 30.00
Time =    0.131 [sec]
```

- CUDA IPC の無効化
  - ✓ OpenMPI では、デフォルトで CUDA IPC は有効となる。これを無効とするには、実行時に btl_smcuda_use_cuda_ipcを0とする。

```
mpirun -np 2 --mca btl_smcuda_use_cuda_ipc 0 ./run
```

# 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。

```
$ cat run.sh
#! /bin/sh
#PBS -q h-tutorial
#PBS -l select=2:mpiprocs=1:ompthreads=0
#PBS -W group_list=gt00
#PBS -l walltime=00:05:00

cd $PBS_O_WORKDIR

. /etc/profile.d/modules.sh
module load pgi/17.1
module load openmpi-gdr/1.10.7/pgi

mpirun -np 2 ./run
```

**select=2:**

使用するノード数を指定。この場合は2ノード。

**mpiprocs=1:**

各ノードに割り当てるMPIプロセス数。この場合は1プロセス。

**mpirun -np 2:**

-np に全並列数（全MPIプロセス数）を指定。この場合は2プロセス。

# GPUDirectRDMA無効コード

openacc_mpi_basic/ 05_no_gdr

- 05_no_gdr コード
  - ✓ 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
$ qsub ./run.sh
$ cat run.sh.o??????
num of GPUs = 2
Rank 0: hostname = a086, GPU num = 0
Rank 1: hostname = a087, GPU num = 1
mean = 30.00
Time =    0.034 [sec]
```

# GPUDirectRDMA有効コード（1）

openacc_mpi_basic/ 06_gdr

- 06_gdr コード
  - main.c 自体は 04_cuda_aware と同じ。run.sh のみ変更。
- GPUDirectRDMA の有効化
  - ✓ OpenMPI では、デフォルトで GPUDirectRDMA は無効である。これを有効にするには、実行時に btl_openib_want_cuda_gdrを1とする。

```
mpirun -np 2 -mca btl_openib_want_cuda_gdr 1 ./run
```

- 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。

# GPUDirectRDMA有効コード（2）

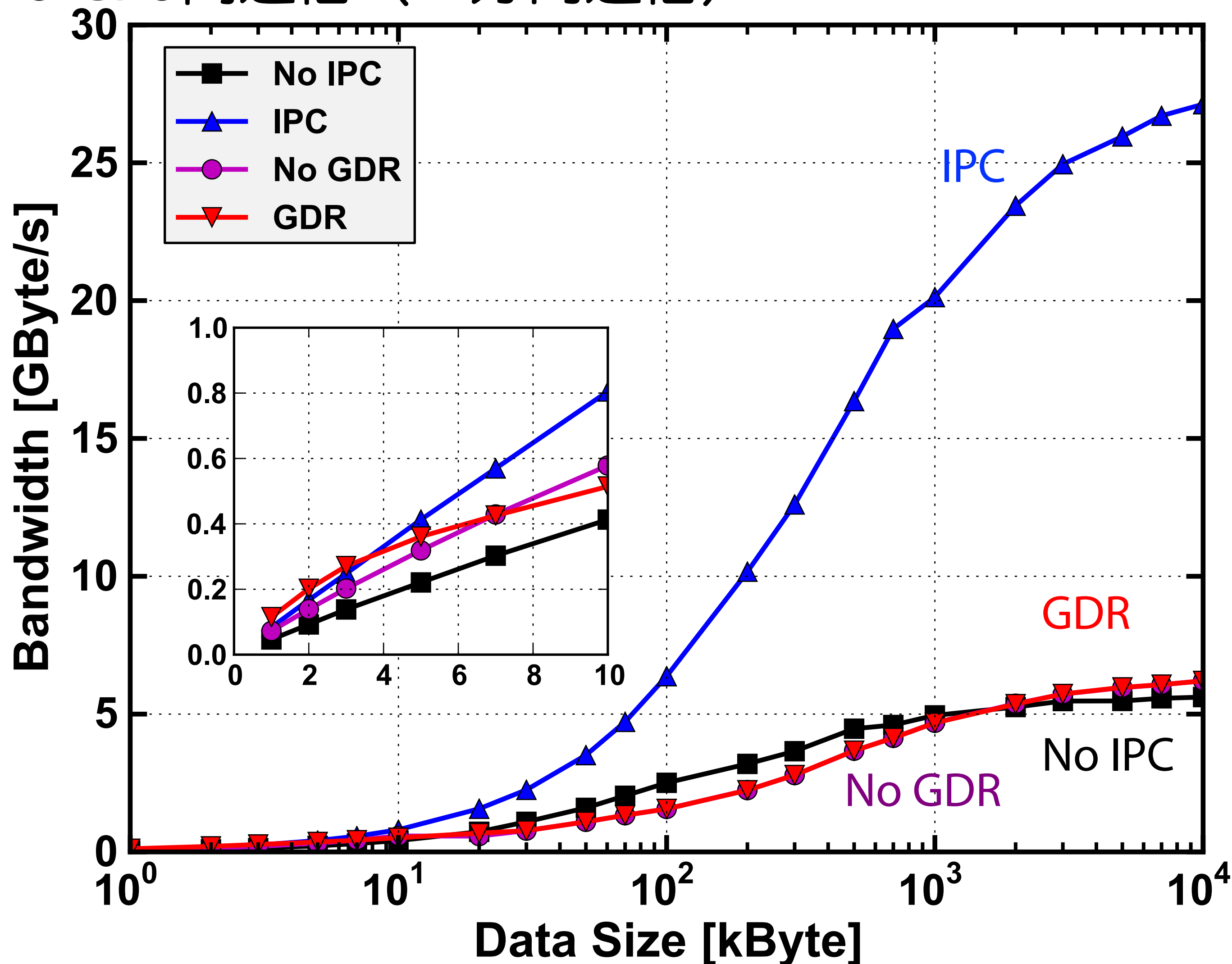
openacc_mpi_basic/ 06_gdr

■ コンパイルと実行

```
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
num of GPUs = 2
Rank 1: hostname = a087, GPU num = 1
Rank 0: hostname = a086, GPU num = 0
mean = 30.00
Time =    0.051 [sec]
```

# 通信方法によるバンド幅比較

- GPU-GPU間通信（一方向通信）




✓ openmpi-gdr/1.10.7/pgi による測定

# 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。

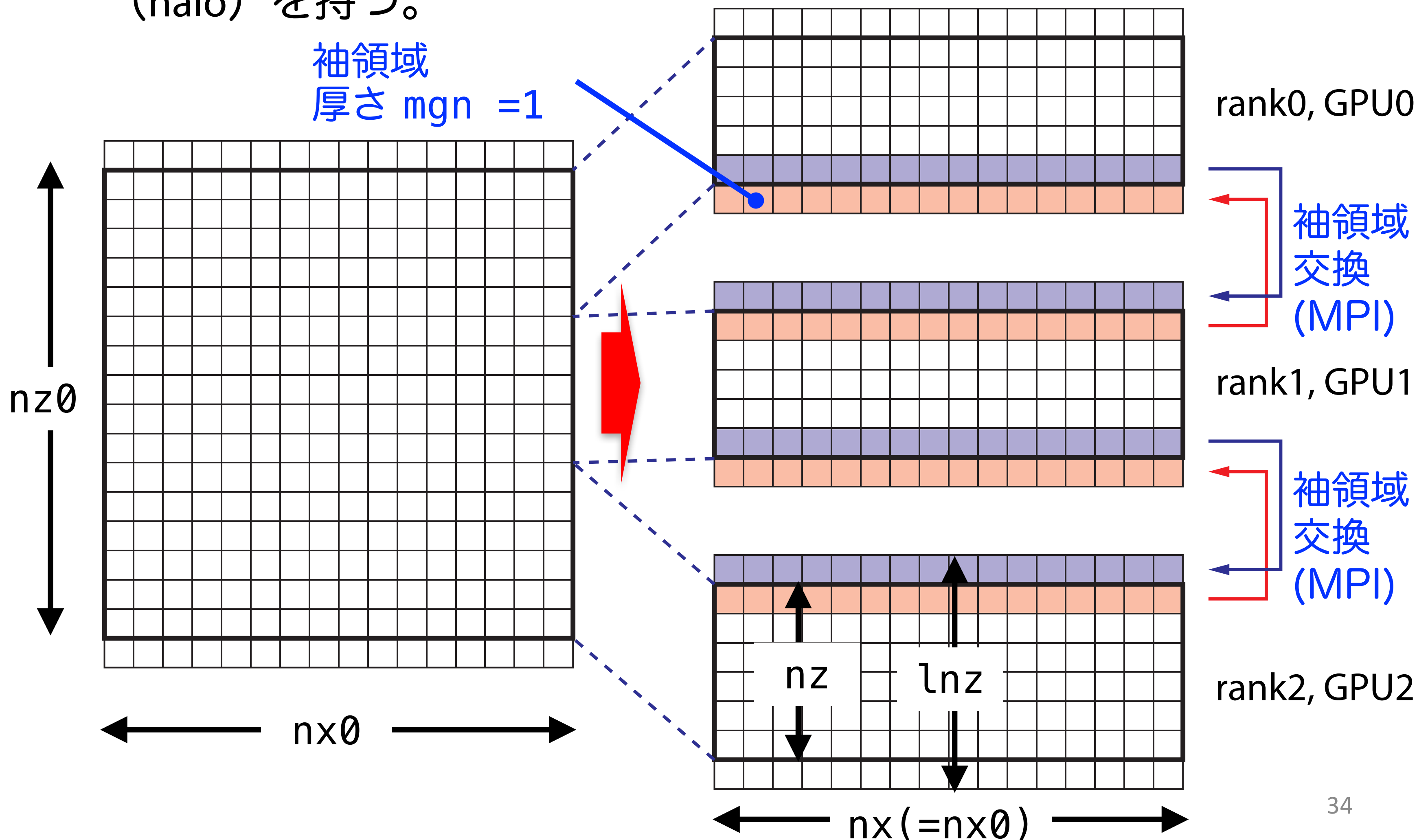
diffusion.c, diffusion3d 関数内

```
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

# 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。



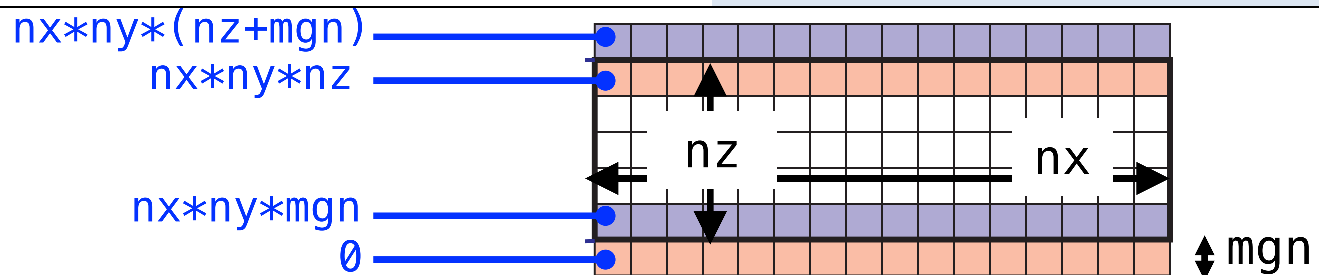
# 時間発展

- 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
$ qsub ./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);
    }

    if (rank == 0) {
        fprintf(stdout, "OMPI_MCA_btl_smcuda_use_cuda_ipc  = %s¥n",
                getenv("OMPI_MCA_btl_smcuda_use_cuda_ipc"));
        fprintf(stdout, "OMPI_MCA_btl_openib_want_cuda_gdr = %s¥n",
                getenv("OMPI_MCA_btl_openib_want_cuda_gdr"));
    }
...
```

OMPI_MCA_btl_smcuda_use_cuda_ipc環境変数などを確認することでCUDA IPC や GDRが有効/無効か確認できる

# 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();

    }
```

# 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 x 2 node から 1 GPU x 2 node または 2 GPU x 1 node へ変更
  - ✓ CUDA IPC や GDR の有効/無効の変更
- PGI_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

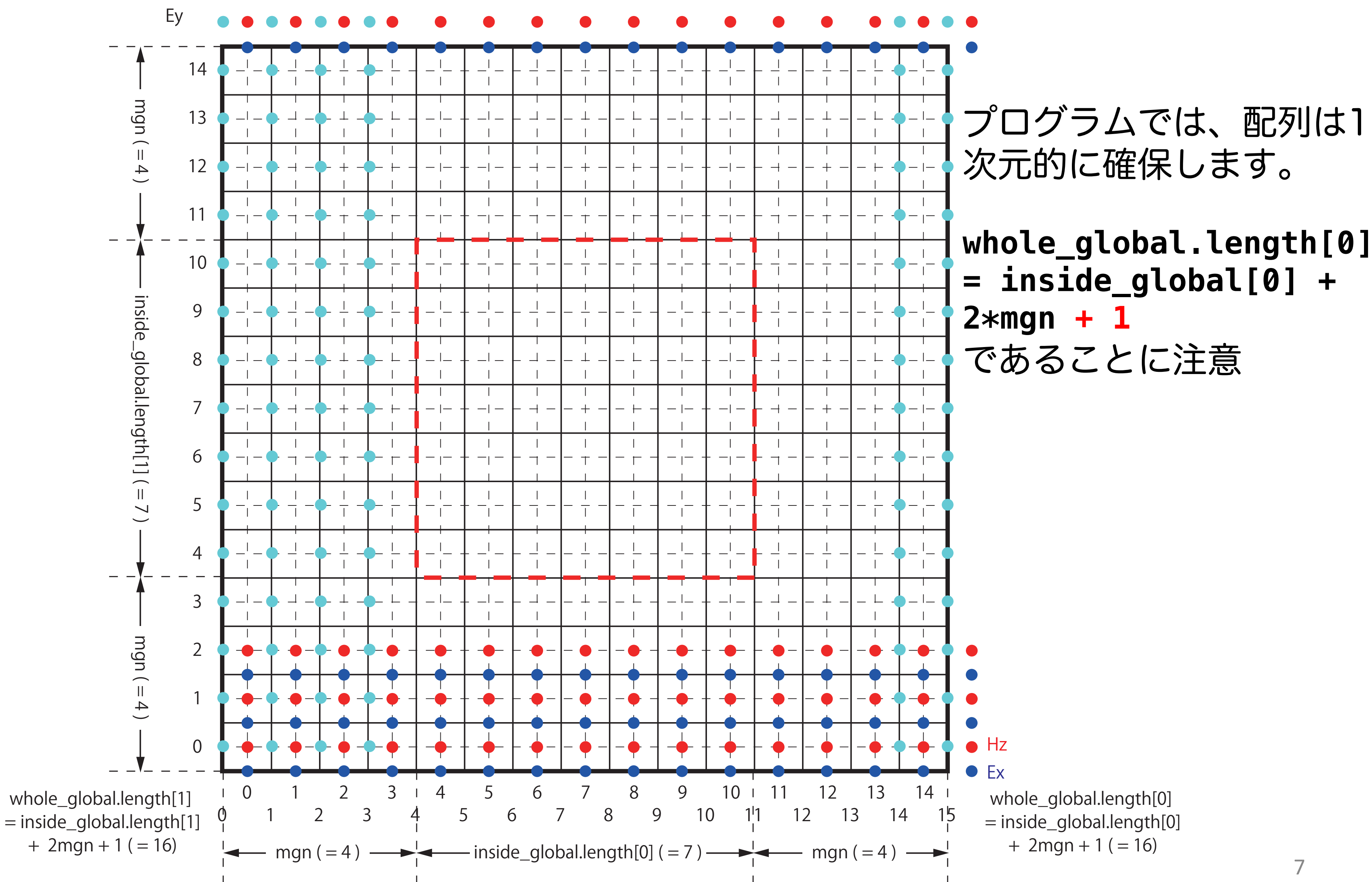
$$\frac{\partial \boldsymbol{E}}{\partial t} = \frac{1}{\varepsilon}\nabla \times \boldsymbol{H} \qquad \frac{\partial \boldsymbol{H}}{\partial t} = -\frac{1}{\mu}\nabla \times \boldsymbol{E}$$

( $\varepsilon$ : 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

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

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

変形して、

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

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

# 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_fdtd/
  - OpenACCとMPIを利用したFDTD法（電磁波解析）

| | |
|---|---|
| openacc_mpi_fdtd/01_original | MPI並列化されたCPUコード。 |
| openacc_mpi_fdtd/02_openacc1 | calc_ex_ey, pml_boundary_ex, pml_boundary_ey, がOpenACC。 |
| openacc_mpi_fdtd/03_openacc2 | 時間更新ループ全体が OpenACC。 |
| openacc_mpi_fdtd/04_openacc3 | 初期化を含め OpenACC。 |
| openacc_mpi_fdtd/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

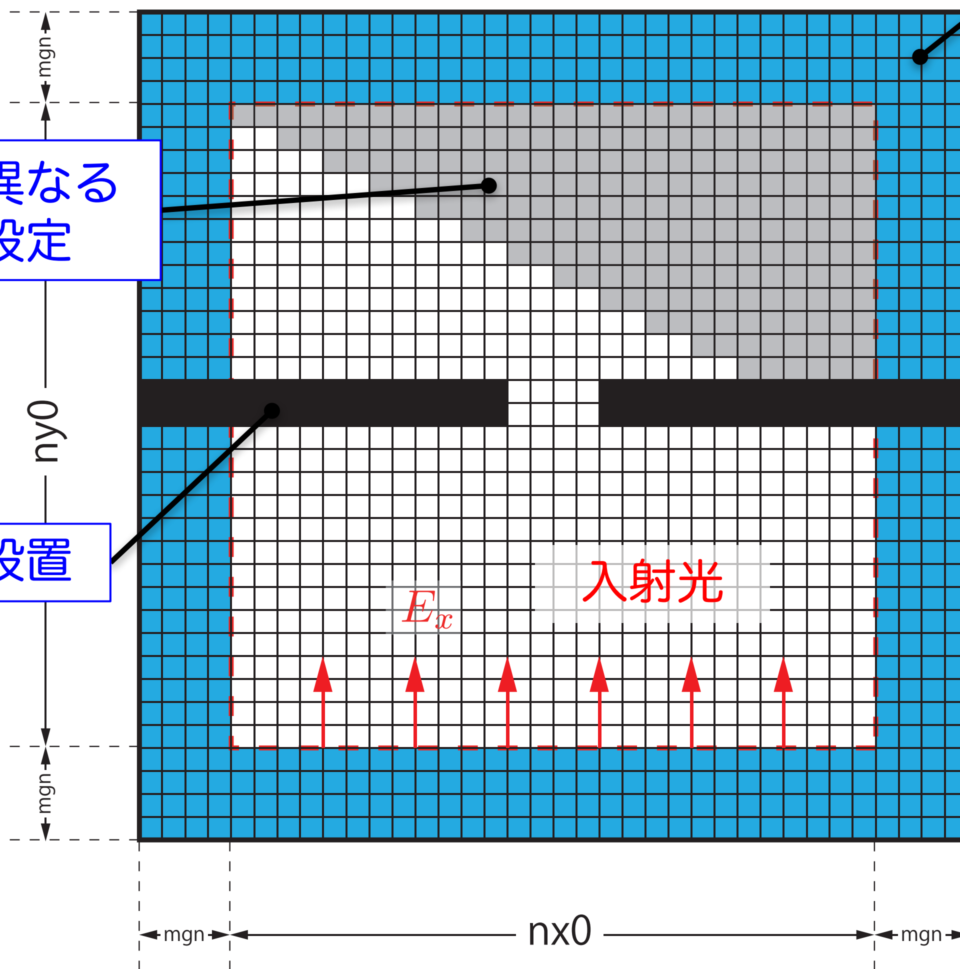
| | |
|---|---|
| main.c | プログラムのメインコード |
| fdtd2d.{c, h} | 2次元 FDTD の 計算コード |
| fdtd2d_sources.{c, h} | 入射光設定のための関数 |
| setup.c | 計算条件の設定と変数の初期化 |
| config.{c, h} | 物理定数の定義 |
| output.{cc, h} | 計算結果出力のための関数 |
| bitmap* | BMPファイル作成のための関数 |

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: $Ex$、$Ey$、$Hz$
  - ✓ $y$ 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)




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

デフォルト設定:

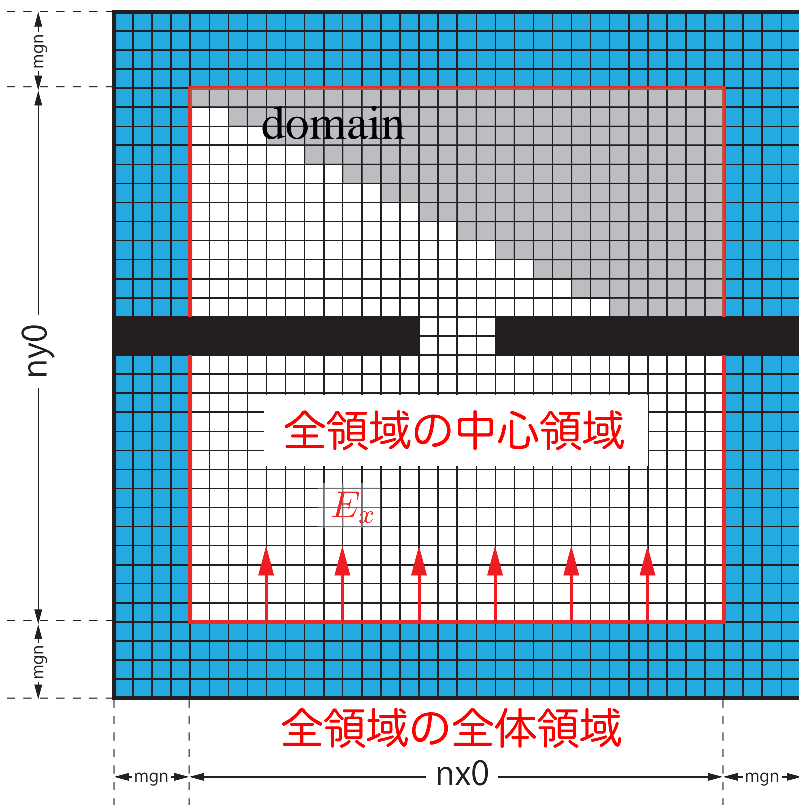
```
nx = 512
ny = 512
mgn = 8
lnx = 529
lny = 529
```

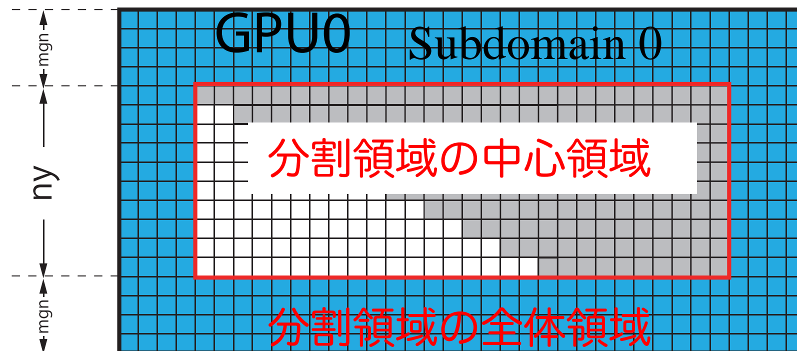
プログラム中では下記の変数が使われているので注意

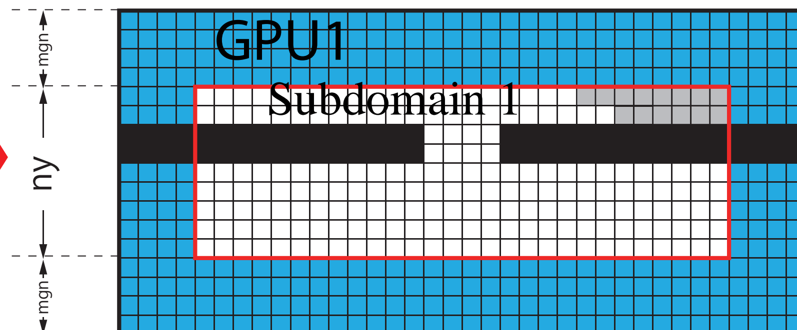
```
inside_global.length[0] = nx0
inside_global.length[1] = ny0
whole_global.length[0]
            = nx0 + 2*mgn + 1
whole_global.length[1]
            = ny0 + 2*mgn + 1
```

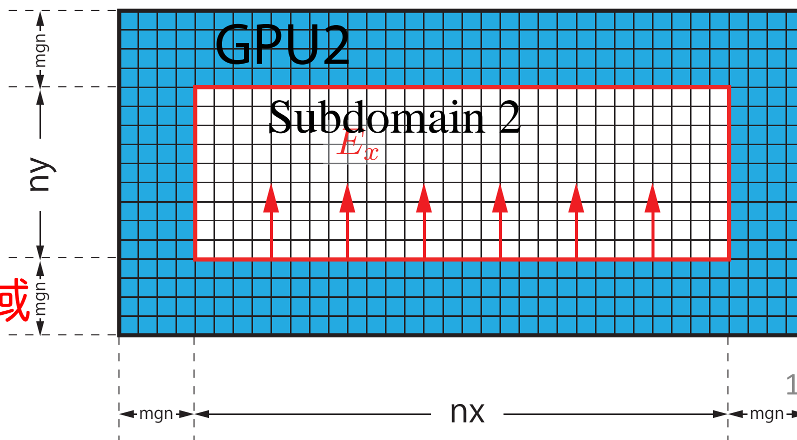
# マルチGPU化のための領域分割

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。



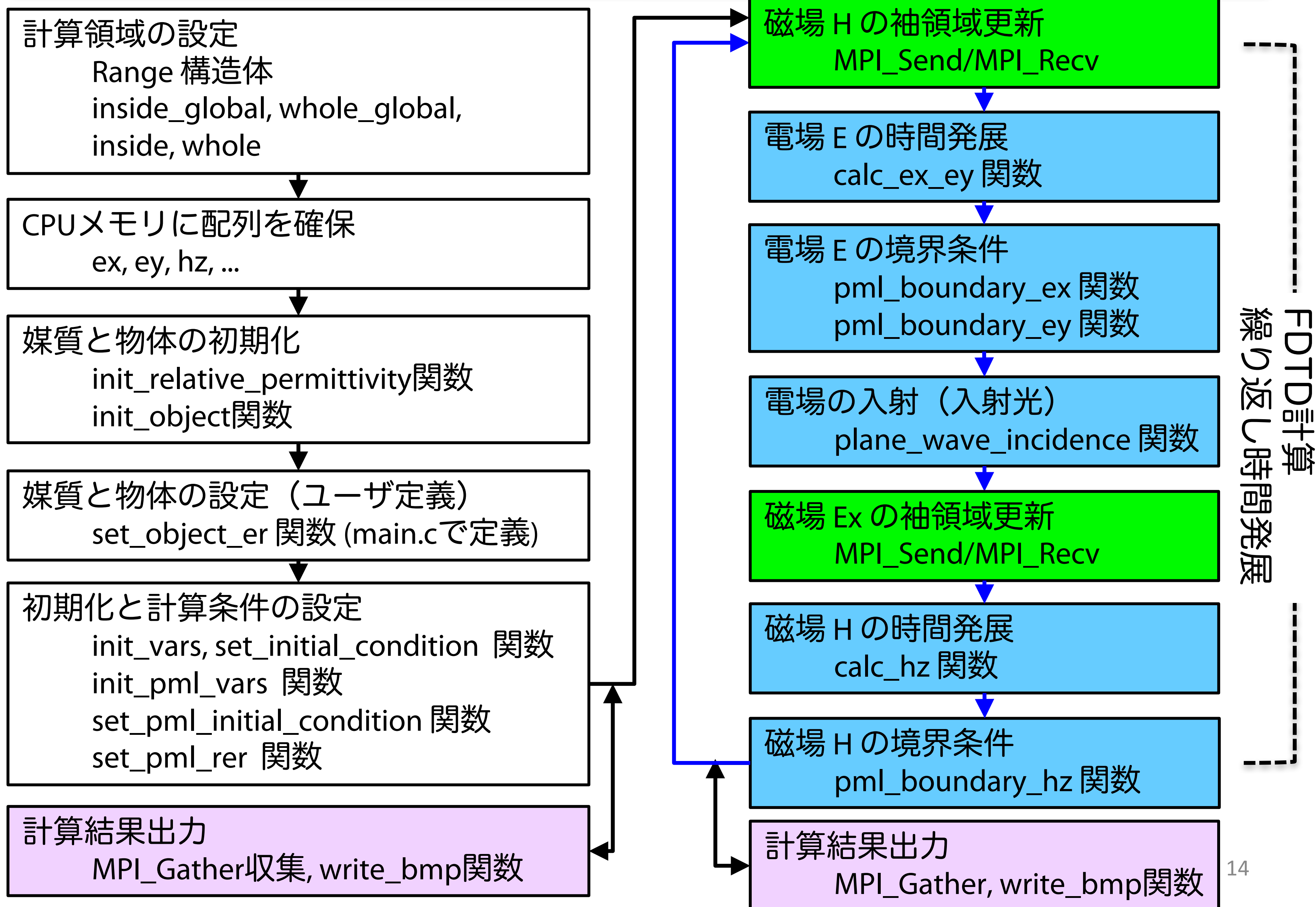



全体領域：袖領域（**PML**、青格子）を含む領域
中心領域：袖領域を含まない領域

# コード全体の流れ（main.c 内）



# 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]) },
                                     { 0, 0 } };
const struct Range whole_global  = { { inside_global.length[0] + 2*mgn + 1,
                                       inside_global.length[1] + 2*mgn + 1},
                                     { inside_global.begin[0]  - mgn      ,
                                       inside_global.begin[1]  - mgn    } };

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    } };
```

全領域の中心領域

全領域の全体領域

分割領域の中心領域

分割領域の全体領域

# 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    } };
```

分割領域の中心領域

分割領域の全体領域

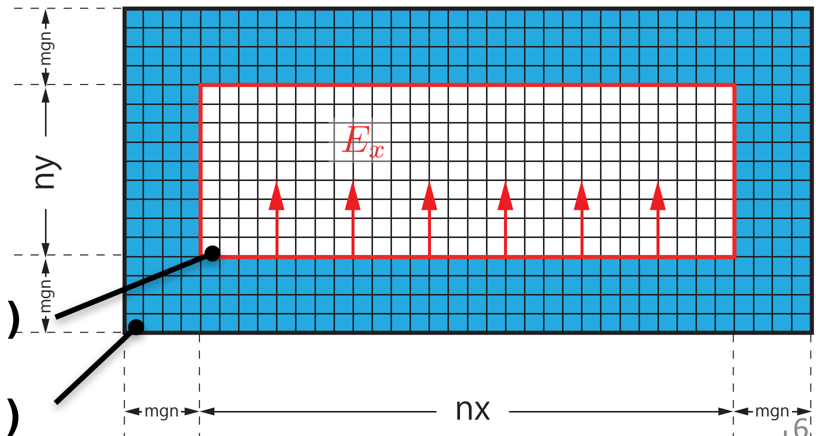
プログラムでは下記の通り

**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);
```

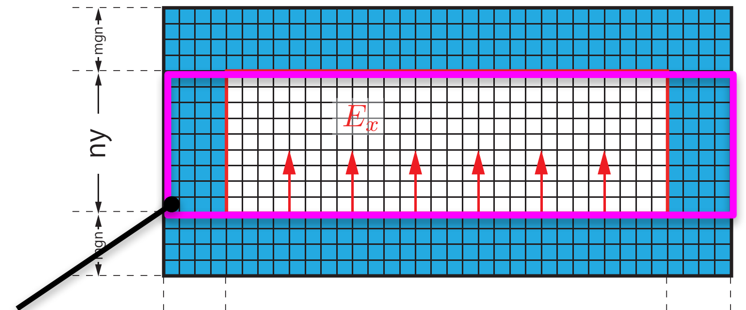
- 多くの配列は whole.length[0] * whole.length[1]
- ex_global, ey_global, hz_global はファイル出力に使うため、**whole_global.length[0] * whole_global.length[1]**

# 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, ceyxl, 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)
```

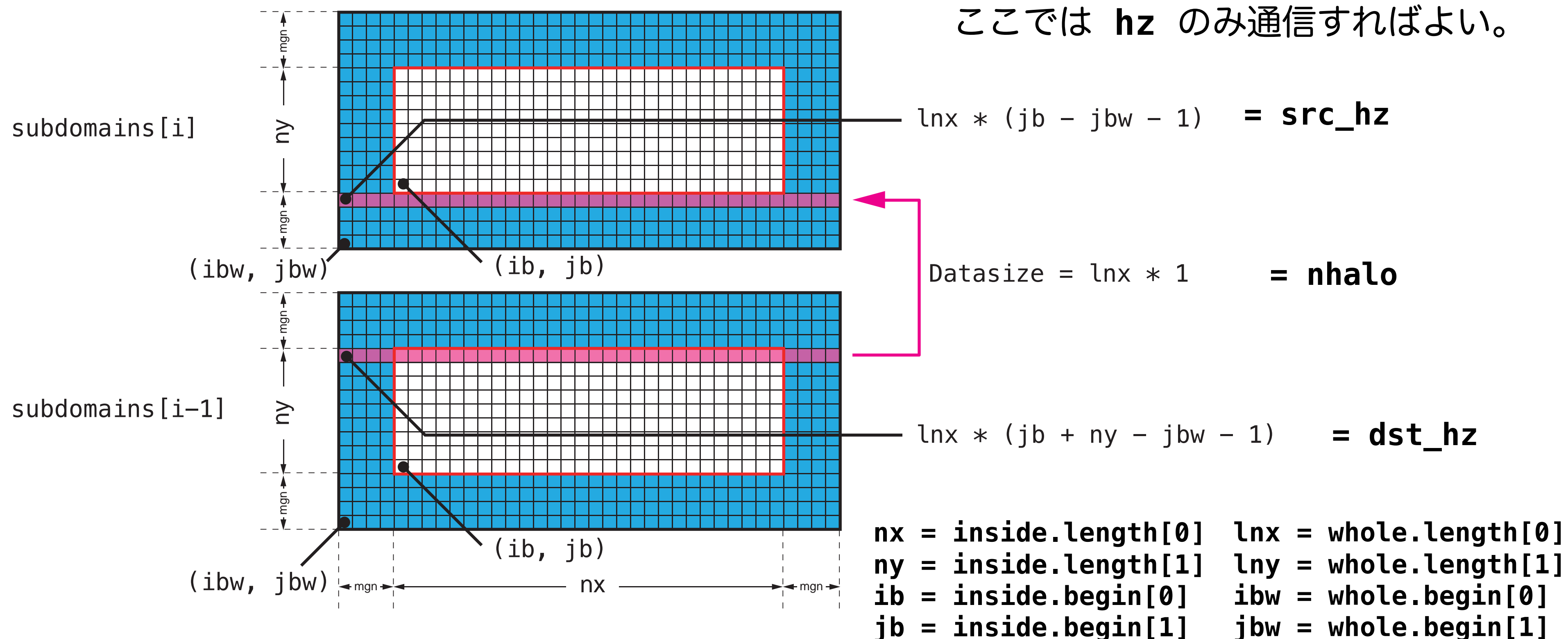
# 時間発展（2）

- 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）

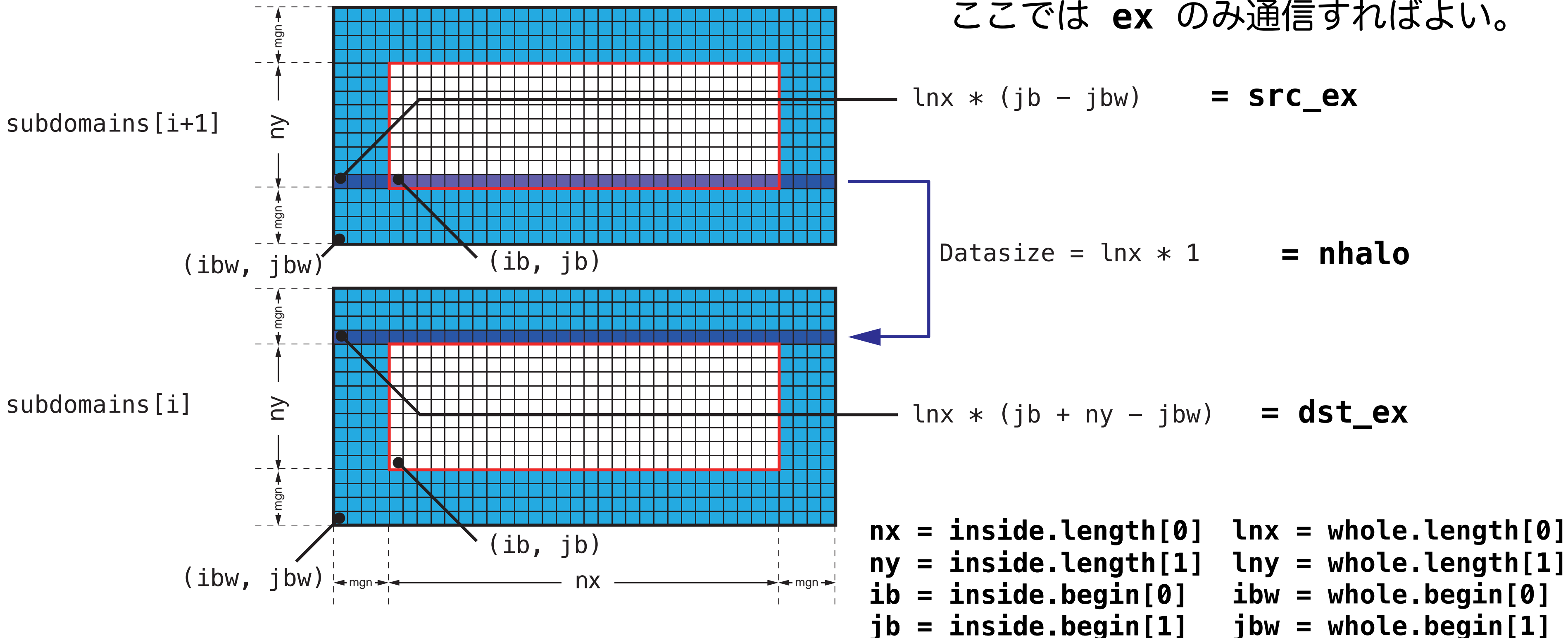
- hz の境界領域のデータ交換




```
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




```
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_fdtd/01_original

```
$ cd openacc_mpi_fdtd/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
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、qsub ./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

openacc_mpi_fdtd/01_original

```
$ cat run.sh
#! /bin/sh
#PBS -q h-tutorial
#PBS -l select=2:mpiprocs=2:ompthreads=0

（省略）

mkdir -p sim_run
cd sim_run

nprocs=4
mpirun -np $nprocs ../run 512 512 $nprocs 5000 50
```

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

nprocs: 全ランク数（=分割数）

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_fdtd/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
```

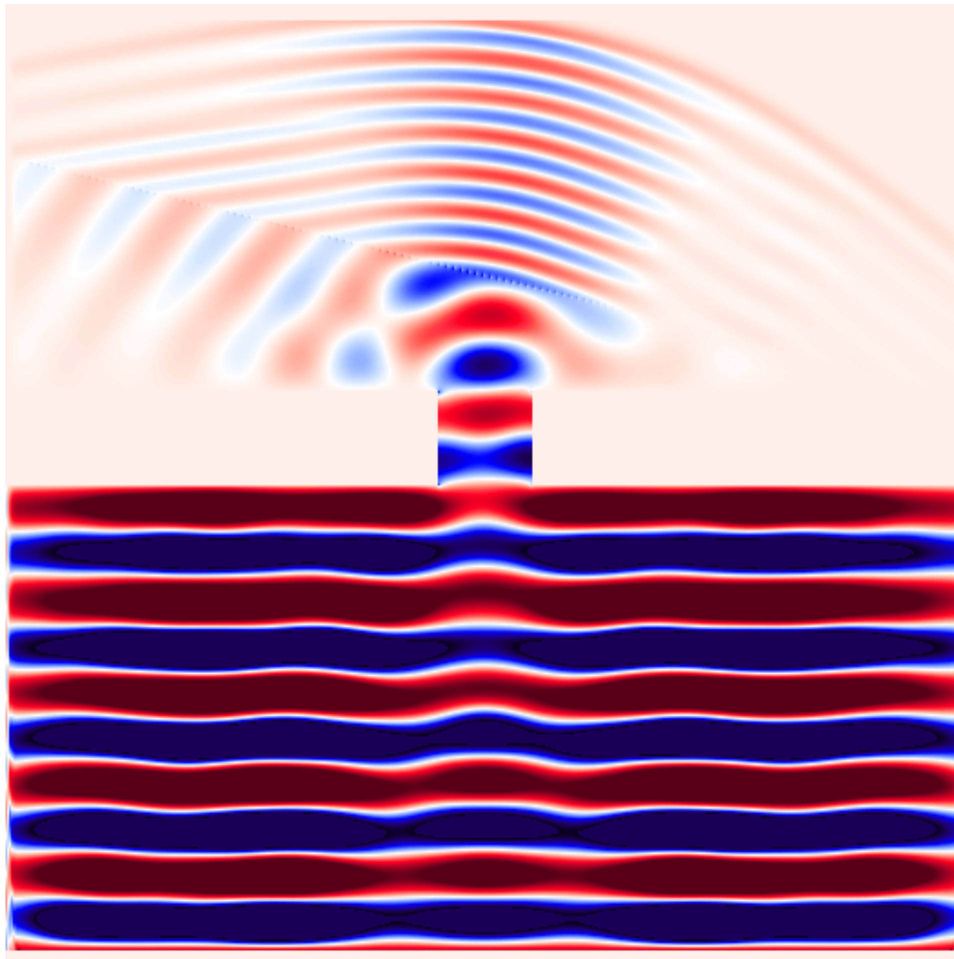
なお

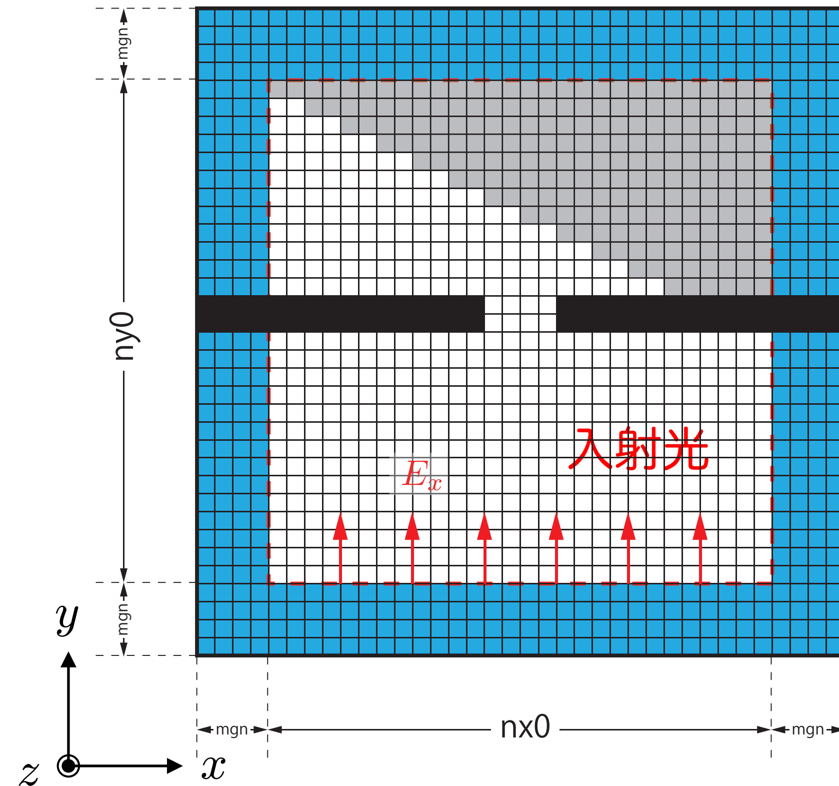
```
ssh –Y txxxxx@reedbush.cc.u-tokyo.ac.jp
```

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
  - ✓ $Ex$（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
- fdtd2d.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_fdtd/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], ceyxl[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, ceyxl, 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
- fdtd2d.c
  - ✓ 残りの関数にkernels 指示文、loop 指示文の追加
- fdtd2d_sources.c
  - ✓ kernels 指示文、loop 指示文の追加

解答例は、openacc_mpi_fdtd/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], ceyxl[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 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]);

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

# 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_fdtd/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ょう。
  - ✓ 計算格子サイズを 4096 x 4096 など大きくしてみましょう。
- OpenACCコードをさらに最適化しましょう。
  - ✓ PGI_ACC_TIMEも活用しましょう。
  - ✓ 実は単純に fdtd2d.c に kernels と loop を入れても、いくつかの関数で暗黙の copyin が発生します。これも修正していきましょう。

```
$ 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_fdtd/05_openacc4