

再配布禁止

※ 本資料に記載されている内容の無断転載・複製を禁じます

「富岳」セミナー 中級編第二部 講習会資料



登録施設利用促進機関 / 文科省委託事業「HPCIの運営」代表機関
一般財団法人 高度情報科学研究機構（RIST）【著】



国立研究開発法人理化学研究所 計算科学研究センター（R-CCS）

本セミナー中級編の目的

- 「富岳」でプログラムを効率的に走らせるためには、チューニングによりプログラムの高速化を目指す必要があります。そのためには、性能情報を採取し、自分のプログラムがどのような特性を持っているかを知ることが重要となります。
- 本中級編は、チューニング未経験の人を対象に、性能向上策実施の補助となるプログラム性能情報の採取方法や最適化メッセージの見方、重要な最適化手法の概略、および最適化手法の適用例等の基本事項を説明します。

中級編の開催予定

本中級編は今後開催予定も含めて三部構成を予定しています。

■ 第一部（適時開催中）：プロファイラ・重要な最適化手法概略編

第一部では**CPU**単体性能最適化の中で特に重要な最適化手法である**SIMD**化、ソフトウェアパイプライン、プリフェッチについて、概念説明やプロファイラを用いて最適化の効果を確認する方法を説明しています。

■ 第二部（今回開催）：MPI・LLIO編

第二部では並列性能最適化に向けて、通信や**I/O**の最適化の際に考慮すべき事項を取り上げます。効率的な通信や**I/O**のためのノードやプロセスの配置に関する各種事項、「富岳」のストレージ構成、**I/O**性能を最適化・高速化する技術である**LLIO**の機能概略、**MPI**通信性能情報（**MPI**統計情報）の採取方法、及び「富岳」の**MPI**通信に固有な事項を説明します。

■ 第三部（今後開催予定）：Fortranでの各種最適化手法編

第一部で取り上げなかった**CPU**単体性能の各種最適化手法を取り上げる予定です。

第二部（今回開催）の内容

■ 「富岳」の概略

CPUとメモリ等、Tofuインターコネクトの概略、計算ラックの構成概略を説明します。

■ MPIジョブ実行に関連する各種基本事項

ノード資源の指定、Tofu座標（物理座標）におけるノードの配置方法、MPIジョブ投入時の指定など、効率的な通信やI/Oのためのノードやプロセスの配置に関する各種事項について説明します。

■ 「富岳」のストレージ構成とLLIO

I/Oのための基礎知識として「富岳」のストレージ構成を説明し、I/O性能を最適化・高速化する技術であるLLIOの機能概略を説明します。

■ 「富岳」のMPI通信

「富岳」のMPI通信に固有な事項として、MPI性能情報採取機能であるMPI統計情報、「富岳」の通信モード、バリア通信機能という高速化技術を用いた集団通信、一対一通信での通信方式であるイーガー通信方式とランデブー通信方式、メモリ使用量を抑えるための考慮点を説明します。

内容

■ 「富岳」の概略

- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

内容

■ 「富岳」の概略

■ CPUとメモリ等

- Tofuインターコネクトの概略

- 計算ラックの構成

■ MPIジョブ実行に関連する各種基本事項

■ 「富岳」のストレージ構成とLLIO

■ 「富岳」のMPI通信

■ 【付録】参考文献

■ CPU（ノード）の構成

CPU内の構成を下図に示す。

■ CPUチップ：A64FX

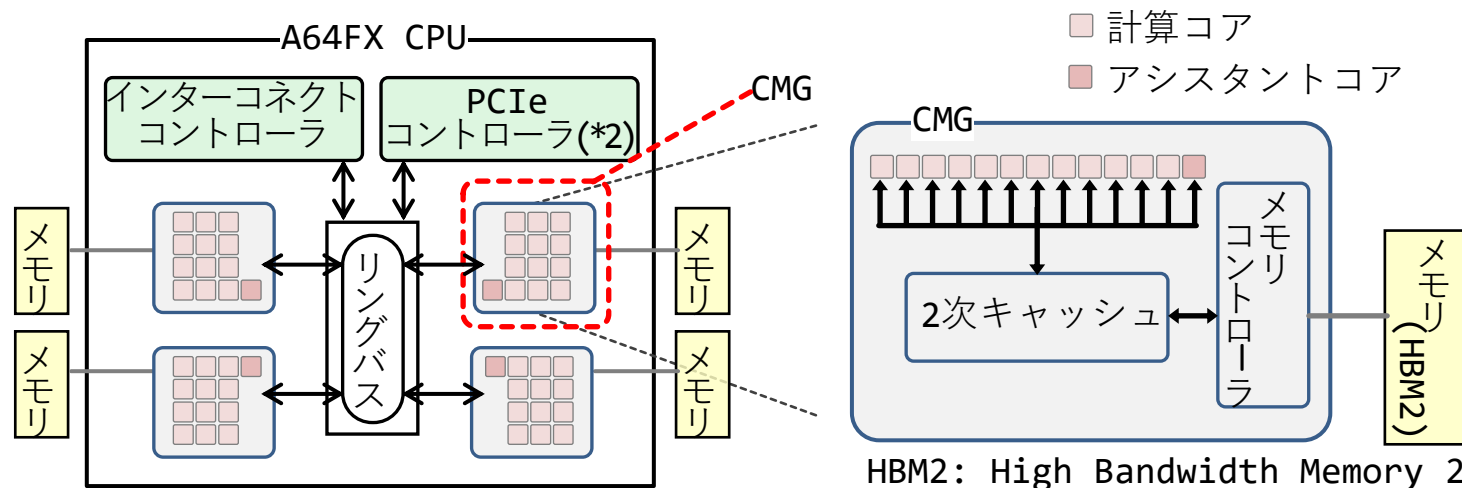
■ CPU（ノード）の総数：158,976

■ CPU当たりのコア数：

□ 計算ノード：計算コア 48 個 + アシスタントコア 2 個(*1)

□ 計算兼I/Oノード：計算コア 48 個 + アシスタントコア 4 個(*1)

(*1) 12個の計算コアと1個（または0個）のアシスタントコア、2次キャッシュ、およびメモリコントローラで、1個のCMG（Core Memory Group）を構成する。一つのCPU内には四つのCMGが含まれる。



(*2) PCIeコントローラは計算兼I/Oノードのみに存在する。

■ CPUの演算性能等

■ 動作周波数：

□ 計算ノード：通常モード 2.0 GHz①、ブーストモード 2.2 GHz

□ 計算兼I/Oノード：常に2.2 GHz

■ 浮動小数点演算器：512-bit SIMD② FMA③ × 2個④

■ ピーク演算性能：コア当たり、倍精度で 64 GFLOPS(*3)、
ノード当たり、倍精度で 3072 GFLOPS

(*3) 1秒間にコア当たり、① 2 G サイクル × ② 倍精度 8 要素 × ③ 2 積和演算
× ④ 2 個 = 64 G 回の浮動小数点演算、即ち、64 GFLOPS

■ 従って、「富岳」のCPU性能を発揮させるためには、SIMDとFMAの活用が重要である。

■ レジスタ

- 浮動小数点レジスタ：コア当たり 32 個

■ キャッシュ

- 1次データキャッシュ：コア当たり 4 ウェイ（4 ウェイの合計 64 KiB）
- 2次データキャッシュ：CMG当たり 16 ウェイ（16 ウェイの合計 8 MiB）

■ メモリ

- 容量：CPU当たり 32 GiB（CMG当たり 8 GiB）
- 転送速度：CPU当たり 1024 GB/s

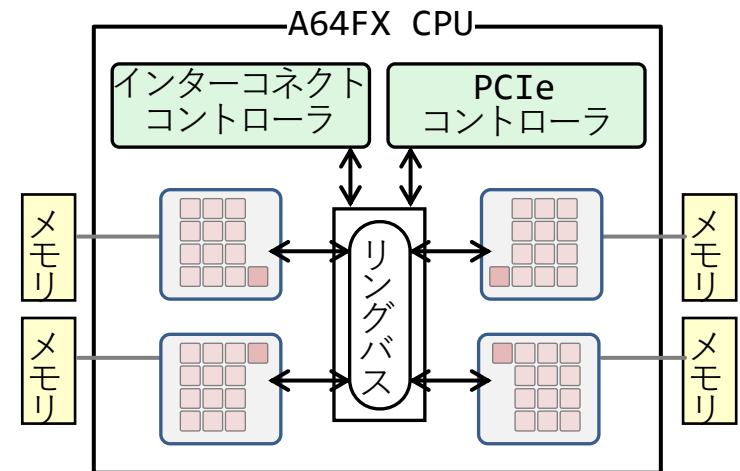
■ 計算ノード間ネットワーク

- Tofu Interconnect D
(28 Gbps x 2 lane x 10 port)

■ ストレージ

■ 階層化ストレージ：

- 第1階層：LLIO（Lightweight Layered IO-Accelerator）
- 第2階層：FEFS（Fujitsu Exabyte File System）
- 第3階層：商用クラウドストレージ（要別途契約）、HPCIストレージ



□ 計算コア
■ アシスタントコア

内容


- 「富岳」の概略
 - CPUとメモリ等
 - Tofuインターコネクトの概略
 - 計算ラックの構成
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ Tofuインターコネクト、Tofu単位

- 大規模並列計算では多数のノードを利用して**MPI**プログラムを実行し、一般に多数のノード間で大量のデータのやり取り（通信）が必要となる。そのため、大規模並列計算では一般に通信時間が大きくなるので通信性能が非常に大事である。
- 通信時間は通信を行うノード間に介在するネットワークの本数（ホップ数）に依存し、ホップ数が少ないほど通信時間は短くなる。
- 通信性能を向上させるためにはホップ数を最小化することが重要であり、それを実現するために、「富岳」ではTofuインターコネクトという技術を導入している。
- Tofuインターコネクトの、大きさ $2 \times 3 \times 2$ のa、b、c軸（次スライドで説明）で構成される単位をTofu単位という。

■ Tofu単位の構成と物理座標

- Tofuインターコネクトの、大きさ $2 \times 3 \times 2$ のa、b、c軸で構成される単位をTofu単位という。

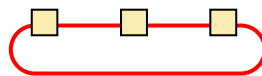
- 一つのTofu単位には、ノード（図の  の部分）が12個含まれる。

- 各Tofu単位内の：

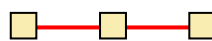
- 各ノードの座標はa、b、c軸で表される。

- 各ノード間の結合は

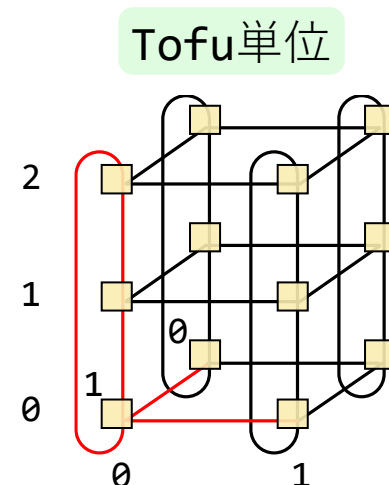
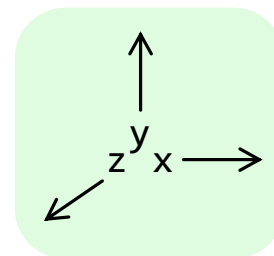
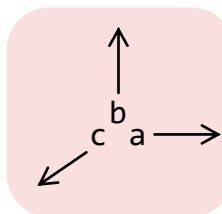
- b軸方向はトーラス結合
（ノードがリング状に結合）。



- a、c軸方向はメッシュ結合
（ノードが3個以上の場合、両端が結合していない）。



- 各Tofu単位の座標はx、y、z軸で表される。



- 以後、x、y、z軸およびa、b、c軸を物理座標と呼ぶ。

■ 一つのノードと、隣接するTofu単位内のノードの結合方法

- x、y、z軸について隣接するTofu単位同士では、a、b、c座標を同じくするノード同士が接続されている。

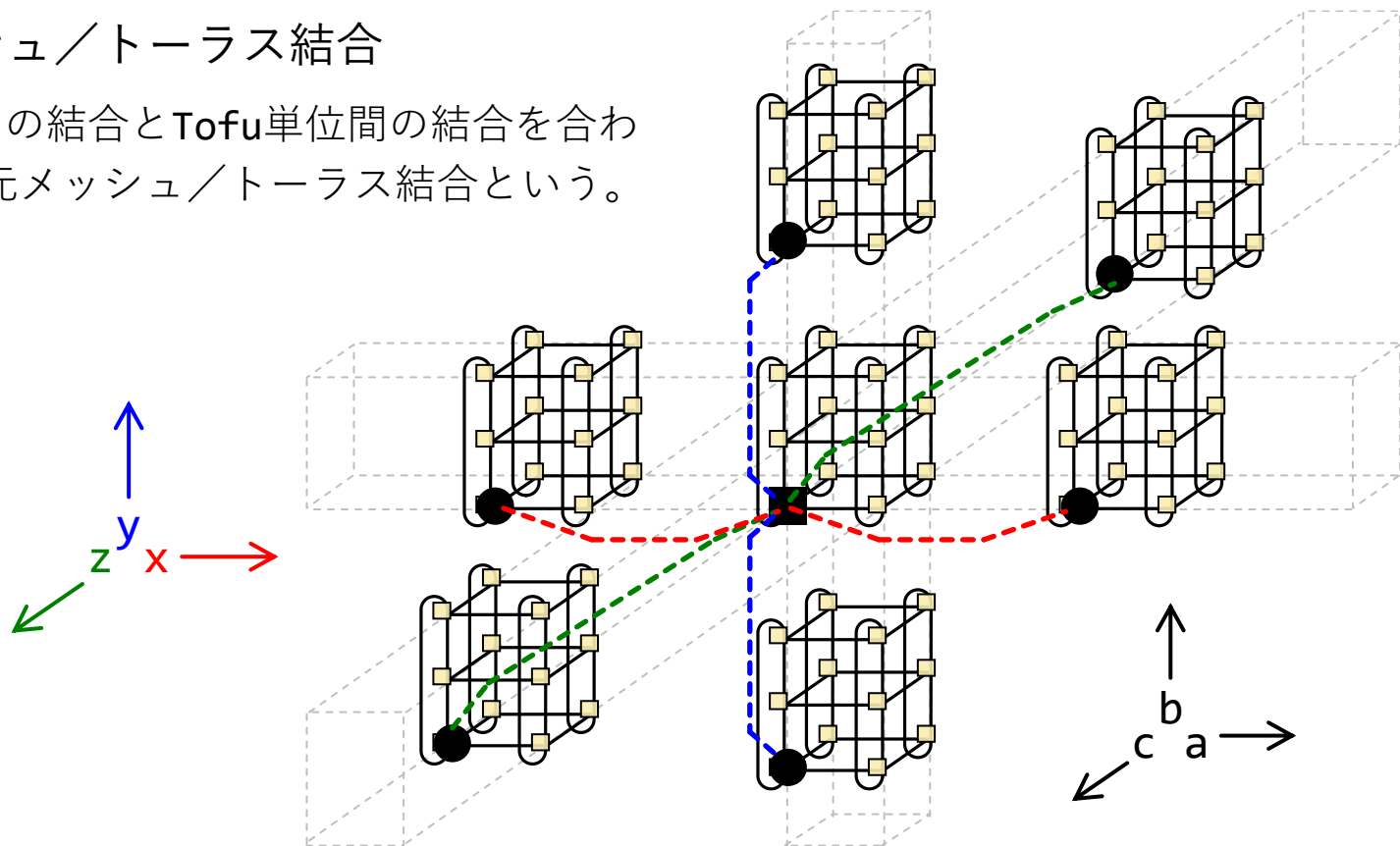
- 例えば、下の図の中央のTofu単位内の■のノードは、左右、上下、前後のTofu単位の同じ位置にある●のノードと、それぞれ赤・青・緑の点線で結合している。



- ■以外のノードについても同様である。

■ 6次元メッシュ／トーラス結合

- Tofu単位内の結合とTofu単位間の結合を合わせて、6次元メッシュ／トーラス結合という。



■ 論理座標

- MPIジョブの実行時に、ユーザはジョブスクリプトにおいて、（例えば「`#PJM -L "node=2x6x4"`」のように）使用したいノード数を、**X**、**Y**、**Z**軸の各方向の個数で指定できる。
- 使用したいノード数の指定における**X**、**Y**、**Z**軸を論理座標と呼ぶ。

■ 論理座標と物理座標の対応付け

- 論理座標のノードは、実際には**Tofu**の物理座標上のノードに対応付けられる。
- この対応付けはシステムが自動的に行うので、ユーザは関知する必要はない。

内容

- 「富岳」の概略
 - CPUとメモリ等
 - Tofuインターコネクトの概略
 - 計算ラックの構成
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ チューニングにおけるノード形状やI/Oの重要性

- 実行時のノード数やノード形状を決めるにあたっては、多数の計算ノードを収めた計算ラックの構成を踏まえて性能に悪影響が出にくい値にするのが良い。

- 計算ラック内には計算に加えてI/Oを担当するノードというものがあり、I/O性能の改善が求められる場合には計算ラック内でのI/Oノードの位置を意識してノード形状を決める必要がある。

※ 具体的には「MPIジョブ実行に関連する各種基本事項」の章の「I/Oノードを意識した資源の割り当て」で説明します。

- I/Oノードのうち、ストレージI/OノードはI/O性能向上を目的とするLLIOの構成要素である。

※ I/OノードとLLIOについては、「「富岳」のストレージ構成」の章で説明します。

- I/Oを分散する場合にも計算ラックの構成を踏まえて検討すると良い。

■ 計算ラックの構成 [1/3]：シェルフ

- 一つのシェルフにTofu単位が4個含まれる。
 - 1個のシェルフに48ノード入る。

※ シェルフについては、「「富岳」のストレージ構成」の章でさらに説明します。

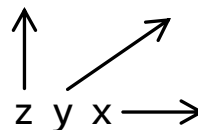
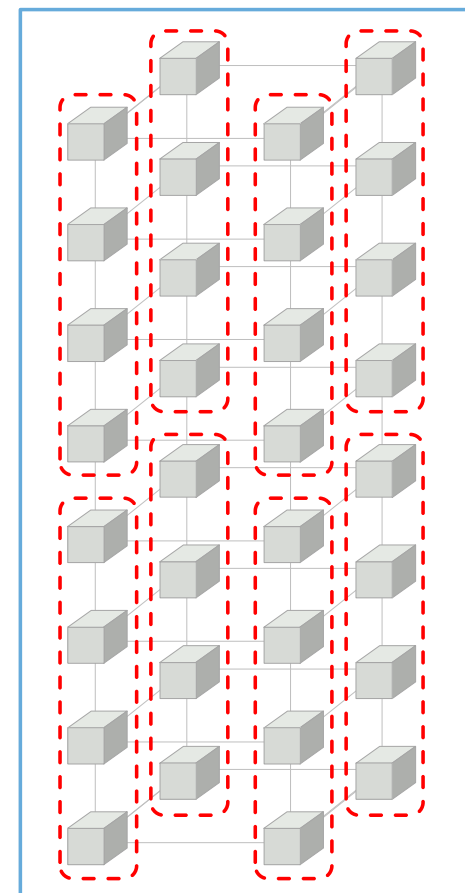
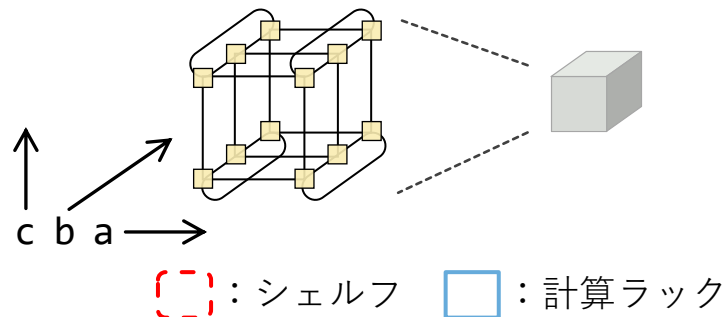
- 8個のシェルフが一つの計算ラックに収められている。
 - 一つの計算ラックに384ノード入る(*1)。

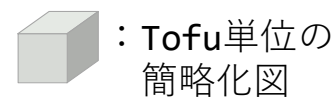
(*1) ただし、36個の計算ラックには192ノード入る。

- 384ノードのラック数：396、
192ノードのラック数：36、
よって、ノードの総数は、**158,976**となる。
- 384ノード以下を使用する場合はジョブスクリプトにおいて「#PJM -L "rscgrp=small"」を、385ノード以上を使用する場合は「#PJM -L "rscgrp=large"」を指定する。
385ノード以上のジョブはシェルフ単位でノードが割り当てられる。

Tofu単位
(12ノード)

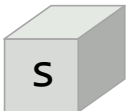
Tofu単位の
簡略化図





■ 計算ラックの構成 [2/3]：ストレージI/Oノード

- 一つのシェルフ に三つのストレージI/Oノードを含むTofu単位(*2)が一つ含まれる。

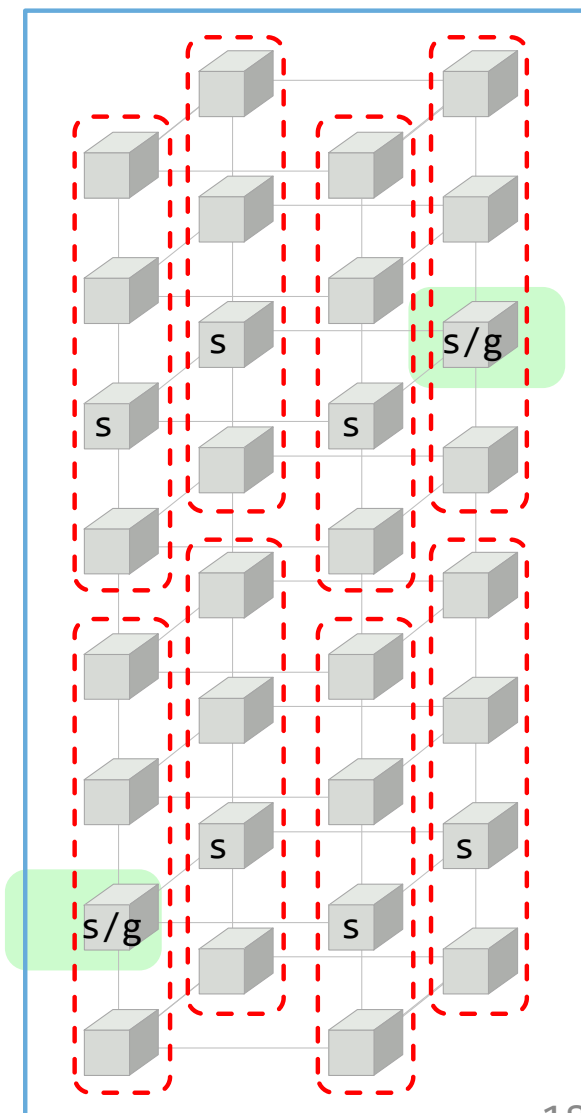
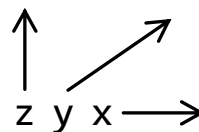
(*2) 、および後述の  で表す。

■ ストレージI/O (SIO) ノード：

- 第1階層ストレージを構成するSSDが接続されていて、第1階層ストレージに対する入出力を担う。
- 計算ノードも兼ねる。
- 16ノード当たりについて存在する。

※ ストレージI/Oノードおよび第1階層ストレージについては、「「富岳」のストレージ構成」の章でさらに説明します。

- 一つのシェルフ内のノードはそのシェルフ内のSIOノードを利用して第1階層ストレージへの入出力を行う。

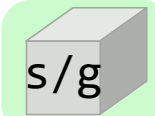




[Red dashed box] : シェルフ [Blue solid box] : 計算ラック

■ 計算ラックの構成 [3/3] : グローバルI/Oノード

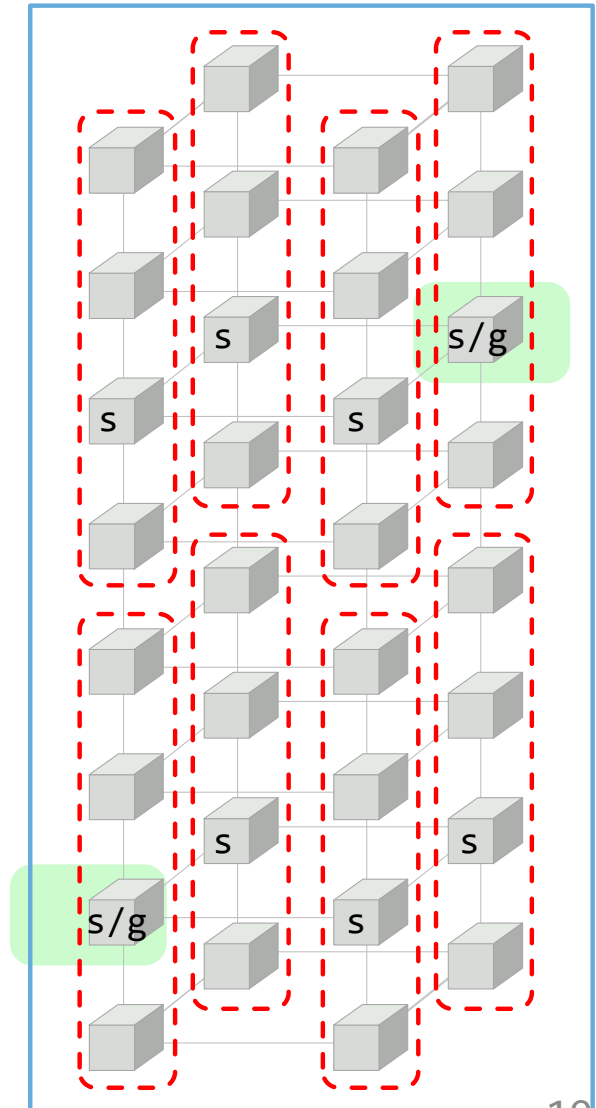
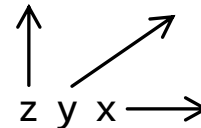
- 一つの計算ラック に三つのストレージI/Oノードと一つのグローバルI/Oノードを含むTofu単位(*3)が二つ含まれる。

(*3)  で表す。

■ グローバルI/O (GIO) ノード :

- 第2階層ストレージに対する入出力を中継する。
- 計算ノードも兼ねる。
- 192ノード当たりについて存在する。

※ グローバルI/Oノードおよび第2階層ストレージについては、「「富岳」のストレージ構成」の章でさらに説明します。



内容

- 「富岳」の概略

- MPIジョブ実行に関連する各種基本事項

- 「富岳」のストレージ構成とLLIO

- 「富岳」のMPI通信

- 【付録】参考文献

■ MPIジョブ実行に関連する各種基本事項

本章の内容

自分のMPIプログラムの通信やI/Oを効率的にするためには、ジョブ投入時に、複数のノードをどのような形状でどのように割り当てるか、ノードにプロセスをどのように割り当てるかを意識することが重要となります。通信に対しては、介在するネットワークの本数が少なくなるようにプロセスを配置するのが良いです。I/Oに対しては、I/Oノードを意識した割り当てによりI/Oの変動が抑えられる可能性があります。

本章ではこれらの検討の際に有用となる各種事項について説明します。

- ノード資源の指定
- Tofu座標（物理座標）におけるノードの配置方法
- MPIジョブ投入時の指定
- mpiexecコマンドのオプション概略
- mpiexecコマンドの `--vcoordfile` オプション

■ MPIジョブ実行に関連する各種基本事項

本章の内容

自分のMPIプログラムの通信やI/Oを効率的にするためには、ジョブ投入時に、複数のノードをどのような形状でどのように割り当てるか、ノードにプロセスをどのように割り当てるかを意識することが重要となります。通信に対しては、介在するネットワークの本数が少なくなるようにプロセスを配置するのが良いです。I/Oに対しては、I/Oノードを意識した割り当てによりI/Oの変動が抑えられる可能性があります。

本章ではこれらの検討の際に有用となる各種事項について説明します。

■ ノード資源の指定

- Tofu座標（物理座標）におけるノードの配置方法
- MPIジョブ投入時の指定
- mpiexecコマンドのオプション概略
- mpiexecコマンドの `--vcoordfile` オプション

■ ノード資源の指定：リソースグループの指定

- ユーザは「富岳」ポータルサイトを参照して、実行したいジョブの規模に応じてリソースグループを選択する。利用可能なリソースグループ名は「富岳」ポータルサイトで確認して下さい。
- リソースグループの指定は、ジョブ投入時にpjsubコマンドの-L（または--rsc-list）オプションのrscgrpパラメータで行う。

◆ コマンドラインでpjsubコマンドのオプションで指定する場合：

```
pjsub {-L | --rsc-list} "rscgrp=リソースグループ名"
```

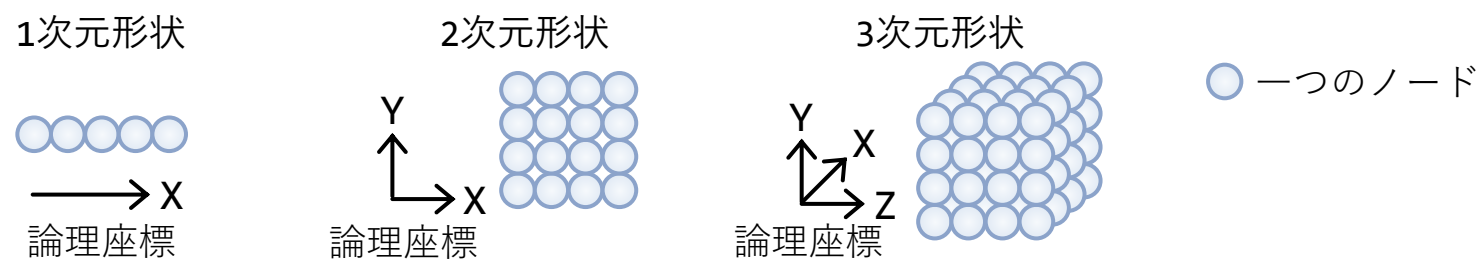
◆ ジョブスクリプト内で指定する場合：

```
#PJM {-L | --rsc-list} "rscgrp=リソースグループ名"
```

■ ノード資源の指定：割り当てるノードの数および形状の指定 [1/2]

ユーザはジョブに割り当てるノードの数および形状を以下のように指定する。

- ユーザは割り当てるノードの集まりを、論理的な1次元、2次元、または3次元の空間（**論理座標**の空間）に配置される形状として（リソースグループ内で可能な範囲で）決める。



- ノード形状の指定をpjsubコマンドの-L（または--rsc-list）オプションのnodeパラメータで行う。

◆ コマンドラインでpjsubコマンドのオプションで指定する場合：

```
{-L | --rsc-list} "node=shape"
```

◆ ジョブスクリプト内で指定する場合：

```
#PJM {-L | --rsc-list} "node=shape"
```

- **shape**に指定する値はノード形状の次元に応じて、論理座標の各軸方向のノード数として、**N1**、**N1xN2**、**N1xN2xN3**のように指定する。
- **shape**に指定された合計**N1**個、**N1xN2**個または**N1xN2xN3**個のノードがジョブに割り当てられる。
- ⚠ **mpiexec**コマンドにより静的に生成されるプロセスのみならず、**MPI_Comm_spawn**関数や**MPI_Comm_spawn_multiple**関数により動的に生成される**MPI**プロセスも、割り当てられたノード上に配置される。従って、ユーザは必要なプロセス数を考慮して割り当てノード数を決める必要がある。

■ ノード資源の指定：割り当てるノードの数および形状の指定 [2/2]

■ nodeパラメータを指定しなかった場合のデフォルト値

nodeパラメータを指定しなかった場合、割り当てるノード形状はジョブACL機能で定義されているデフォルト値に従う。デフォルト値はログインノードでpjac1コマンドの出力の"pjsub option parameters" の項目「node=」のdefault値で確認できる。

※ pjac1コマンドでは--rscgrpもしくは--rgオプションで確認したいリソースグループを指定する。例えばpjac1 --rg largeとすればリソースグループlargeにおけるデフォルト値を確認することができる。--rgを指定しなかった場合はデフォルトのリソースグループsmallの値が出力される。

【pjac1コマンドでnodeパラメータのデフォルト値を確認する例】

```
$ pjac1 --rg small
(中略)
pjsub option parameters
  (-L/--rsc-list)          lower      upper      default
(中略)
      (node=)              1          384        12
(以下略)
```

■ ノード数およびノード形状の確認方法

ユーザが要求したノード数・ノード形状と、実際に割り当てられたノード数・ノード形状は、ジョブ統計情報(*1)内に表示される。

- (*1) ・ ジョブ統計情報はpjsubコマンドの-sオプションまたは-Sオプションの指定によりジョブ終了時に出力される。コマンドラインで指定する場合は、pjsub {-s|-S}であり、ジョブスクリプト内で指定する場合は#PJM {-s|-S}である。
- ・ ジョブ統計情報を格納するファイルのデフォルトの名前は「ジョブ名.ジョブ番号.stats」である（会話型ジョブの場合は「STDIN.ジョブ番号.stats」である）。

ジョブ統計情報内のノード数・ノード形状の情報の出力例を以下に示す。

NODE NUM (REQUIRE)	: 12:2x3x2
NODE NUM (ALLOC)	: 12:2x3x2
NODE NUM (USE)	: 12

- ・ 「NODE NUM (REQUIRE)」はジョブ投入時に指定したノード数とノード形状である。
- ・ 「NODE NUM (ALLOC)」は実際に割り当てられたノード数とノード形状である。
- ・ 「NODE NUM (USE)」は使用されたノード数である。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
 - ノード資源の指定
 - Tofu座標（物理座標）におけるノードの配置方法
 - MPIジョブ投入時の指定
 - mpiexecコマンドのオプション概略
 - mpiexecコマンドの `--vcoordfile` オプション
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ 3種類の配置方法

指定されたノード形状を、Tofu座標（ x 、 y 、 z 、 a 、 b 、 c 軸で構成する6次元の物理座標）にどのように配置するかによって、トーラスモード、メッシュモード、および離散割り当ての3種類の配置方法がある。

※ 運用上リソースグループによってどのノード配置が利用できるかが決められているので、ポータルサイトの「リソースグループ」で確認して下さい。

□ トーラスモード

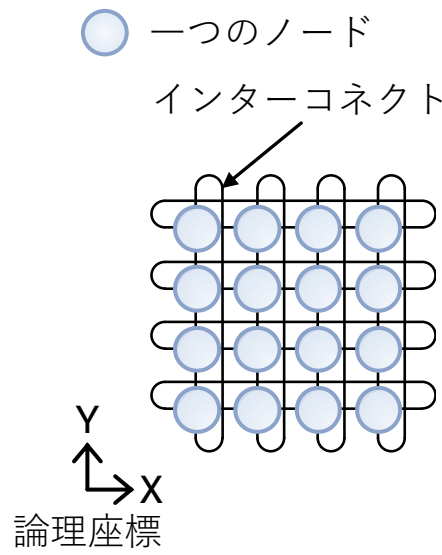
□ メッシュモード

□ 離散割り当て

以降のスライドで順に説明する。

□ トーラスモード

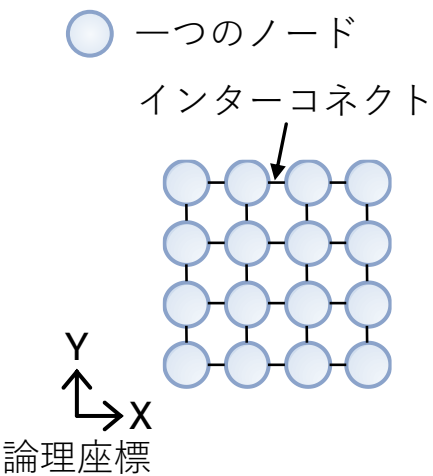
- トーラスモードでは、ノードを接続するインターコネクトは、各軸方向のトーラスを構成する。ノード形状の各軸で両端のノードが隣り合うノードとして接続され、ループを描くようにインターコネクトで接続される。
 - 1次元形状では、X 軸方向のトーラス。
 - 2次元形状では、X、Y軸方向のトーラス。
 - 3次元形状では、X、Y、Z軸方向のトーラス。
- ノードの最小割り当て単位はTofu単位（12ノード）である。
 - ノード数1のジョブの割り当てノード数：
 - 1次元形状（node=1）と指定した場合、割り当てられるノードは1ノードになる。
 - 2次元形状（node=1x1）または3次元形状（node=1x1x1）と指定した場合、割り当てられるノード数はTofu単位に切り上げられる。



MPIジョブ実行に関連する各種基本事項 ▶ Tofu座標（物理座標）におけるノードの配置方法
▶ 3種類の配置方法

□ メッシュモード

- メッシュモードでは、ノード形状の各軸で両端のノードは隣り合うノードとしては接続されない。
- 割り当てられるノードは、Tofu座標上で隣接するノードが選択される。
- ノードの最小割り当て単位は1ノードである。
- メッシュモードで2次元形状（node=N1xN2）を指定した場合、割り当てられるノード形状は3x1単位に切り上げられる。



NODE NUM (REQUIRE)	: 1:1x1:mesh
NODE NUM (ALLOC)	: 3:1x3x1:mesh

NODE NUM (REQUIRE)	: 2:1x2:mesh
NODE NUM (ALLOC)	: 3:3x1x1:mesh

NODE NUM (REQUIRE)	: 2:2x1:mesh
NODE NUM (ALLOC)	: 3:1x3x1:mesh

□ メッシュモードについての補足事項

2022年6月7日時点で、以下の制限事項②は、①の条件付きで解除されている。

① 制限内容：実行開始予定時刻が表示される**mesh**指定（後述）によるメッシュモードのジョブ数はシステム全体で**12**ジョブに制限される。

■ 制限により実行開始予定時刻が表示されないジョブにおいては、**pjstat**コマンドの**REASON**に「**MESHLIMIT EXCEED**」と表示される。

■ **mesh**指定のジョブの投入数には制限はない。

② 2022年6月7日以前までの制限事項

■ **mesh**指定をしたジョブのスケジューリングが非常に時間を要し、運用に影響がでることが判明したため、**mesh**指定でのジョブ投入は一時的に禁止される。

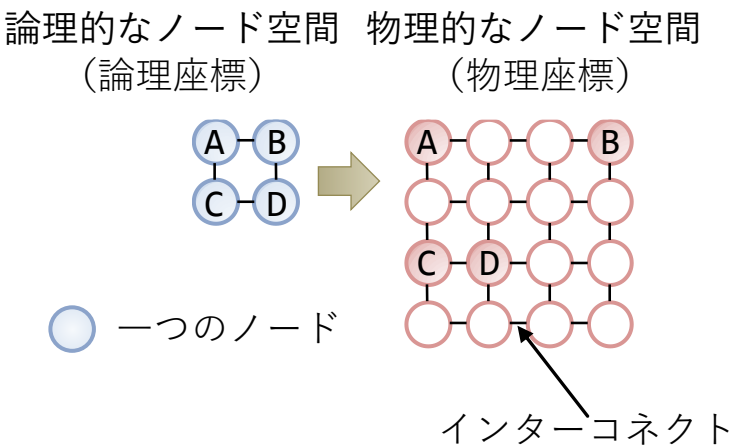
■ 離散割り当てまたはトラスモードの指定をお願いします。

詳細は以下のURLを参照して下さい。

https://www.fugaku.r-ccs.riken.jp/restriction/20220303_01

❑ 離散割り当て

- ノードの最小割り当て単位は1ノードである。
- 割り当てられるノードは、できるだけTofu座標上で隣接するように選択される。
- 離散割り当てでは、ノードが物理的な空間で隣り合うことは保証されない。つまり、離散割り当てではジョブ内の論理的なノード空間におけるノード間の距離と、物理的なノード間の距離は一致しない場合がある。
- ジョブに割り当てられていないノードであっても、複数のジョブの通信経路としてTofuインターコネクトが使用される場合がある。このため、通信経路として使用されているノードのTofuインターコネクトが故障した場合、複数のジョブが影響を受けやすくなる。
- 以下の場合には隣接しないノードが選択される。
 - ・ 隣接する空きノードがない場合。
 - ・ 隣接しないノードを選択することでジョブの実行開始を早められる場合。



■ トーラスモード、メッシュモード、および離散割り当て配置方法の比較

■ ジョブの通信性能の観点

- トーラスモードが有利。

対して、メッシュモードでは、ノードの配置結果によってはノード間の通信経路の長さが変わるため、通信性能が常に同じにならない可能性がある。離散割り当てでは、割り当てるノードが隣接するとは限らないため、各ジョブの通信が干渉しやすくなる。

※ なお、適用される通信アルゴリズムに3次元用が使われないと通信性能が出ない。そのため、**shape**で3次元形状を指定しないと通信性能が出ない。

■ ノードの割り当てやすさの観点

- ノードの最小割り当て単位が**1**ノードである離散割り当てが有利。

離散割り当てのほうが、トーラスモードよりもジョブの実行開始が早くなる可能性がある。

※ 現在の運用だと、メッシュモードがトーラスモードより有利とは必ずしも言えない。
トーラスモードを使用すると、**small**ジョブは**large**ジョブの隙間で実行される可能性がある（**large**ジョブの隙間で実行されるのは**small**ジョブのみ）。メッシュモードではそのような配慮はない。

■ 通信経路故障時の影響の観点

- トーラスモードやメッシュモードが有利。

対して、離散割り当てでは、ジョブに割り当てられていないノードであっても、通信経路として**Tofu**インターコネクトが使用される場合がある。このため、通信経路として使用されているノードの**Tofu**インターコネクトが故障した場合、ジョブが影響を受ける可能性がある。

■ ノードの配置方法を指定するオプション

- Tofu座標内でのノードの配置方法（トーラスモード、メッシュモード、離散割り当て）は、**pjsub**コマンドの**node**パラメータの**shape**の後に**:torus**、**:mesh**、または**:noncont**パラメータで指定される。

```
pjsub {-L | --rsc-list} "node=shape[:torus|:mesh|:noncont]"
```

- **:torus**
トーラスモードと指定する。
 - **:mesh**
メッシュモードと指定する。
 - **:noncont**
離散割り当てと指定する。
-
- 省略時はジョブ**ACL**機能で定義されているデフォルト値に従う。**pjac1**コマンドで、**"defines"**の項目**default allocation mode**の値で確認できる。

```
$ pjac1 --rg small  
  (中略)  
defines  
  (中略)  
      default allocation mode      noncont  
  (以下略)
```

■ 軸の回転の抑止

- ノード形状が3次元でノードの配置方法が**torus**の追加指定によるトーラスモードの場合、**strict**を追加指定することで、論理座標の軸が物理座標に配置される際の軸の回転を抑止することができる。

```
pjsub {-L | --rsc-list} "node=N1xN2xN3:strict:torus"
```

- **strict**指定をしない場合の例
node=3x1x1と指定し、**strict**を指定しない場合は、論理座標のX軸を（物理座標のyおよびb軸に対応する）Y軸に回転することで、割り当てるべきTofu単位は1個で済む。

```
pjsub -L node=3x1x1:torus
```

NODE NUM (REQUIRE)	:	3:3x1x1
NODE NUM (ALLOC)	:	12:2x3x2

- **strict**指定をした場合の例
node=3x1x1と指定し、**strict**を指定した場合は、軸の回転が抑止される。回転の抑止により、（物理座標のxおよびa軸方向に対応する）X方向に3個のノードが必要なため、この方向にTofu単位が2個分必要となる。その結果割り当てるTofu単位は2個となる。

```
pjsub -L node=3x1x1:strict:torus
```

NODE NUM (REQUIRE)	:	3:3x1x1:strict
NODE NUM (ALLOC)	:	24:4x3x2:strict

- ⚠ **strict**指定はトーラスモードでのみ可能である。メッシュモードや離散割り当てでは指定できない。トーラスモード以外で指定すると以下のpjsubエラーとなる。

```
[ERR.] PJM 0002 pjsub Invalid combination: 'strict' and 'mesh'.
```

```
[ERR.] PJM 0002 pjsub Invalid combination: 'strict' and 'noncont'.
```

■ I/Oノードを意識した資源の割り当て

ジョブの入出力処理をするI/Oノード（ストレージI/OノードまたはグローバルI/Oノード）のTofu座標（物理座標）上の位置を意識して計算ノードを割り当てると、ジョブのI/O性能の変動の抑止や向上が見込める。
I/Oノードを意識した割り当てには、以下がある。

□ I/Oノードと計算ノードの距離を毎回同じにする割り当て

この割り当て方法は、ジョブを実行するたびにI/O性能が変動することを抑える効果がある。

□ I/Oノードをジョブに専有させる割り当て

この割り当て方法は、一つのジョブの入出力処理のためにI/Oノードを専有させることでほかのジョブの入出力の影響を受けないようにする効果がある。

以降のスライドではそれぞれについて説明する。

MPIジョブ実行に関連する各種基本事項 ▶ Tofu座標（物理座標）におけるノードの配置方法
▶ I/Oノードを意識した割り当て

□ I/Oノードと計算ノードの距離を毎回同じにする割り当て

- ジョブを実行する計算ノードとその入出力を処理するI/OノードのTofu座標上の距離（座標の差）はジョブのI/O性能に影響する。

ジョブを投入すると、どの計算ラック内のどのノードに割り当てられるかはジョブ運用ソフトウェアが決定する。ノード形状指定のみのジョブ投入では、割り当てられるノードの計算ラック内での相対位置は、ジョブを実行するたびに変化し、常に同じとは限らない。そのため、ジョブを実行する計算ノードとI/OノードとのTofu座標上の距離はジョブ実行で常に同じとは限らず、ジョブ実行のたびにI/O性能が変動する可能性がある。

- ジョブ投入時にパラメータstrict-ioを指定すると、
 - 軸の回転をせずにノードがTofu座標上で割り当てられ、かつ、
 - 割り当てるノードの位置は常に計算ラックの原点（ラック内の最小の座標）が始点になる。

これにより、各計算ノードに対するI/Oノードの位置は毎回同じになり、ジョブ実行のたびにI/O性能が変動することを抑えられる。

```
$ pjsub -L "node=N1xN2xN3:torus:strict-io" job.sh
```

- パラメータstrict-ioは、torusモードかつ3次元形状のノードを割り当てるときだけ指定できる。

【確認例】

- パラメータstrict-ioは指定できないようにジョブACL機能によって管理者が設定している場合がある。指定できるかどうかはpjac1コマンドで確認できる。確認例を右の図に示す。

```
login1$ pjac1 --rg small
(中略)
execute
(中略)
    pjsub(torus)                enable
    pjsub(strict-io)            enable
```




- （次スライドに説明続く）

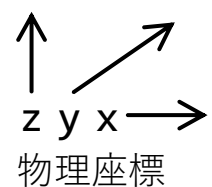
■ strict-ioの指定例

「-L "node=4x3x4:torus:strict-io"」を指定すると、物理xyz座標の中で軸の回転をせずにx方向に2個、y方向に1個、z方向に2個に相当するTofu単位4個の直方体が割り当てられ、かつ、計算ラックの原点（ラック内でのxyzが最小の点）が始点となる。

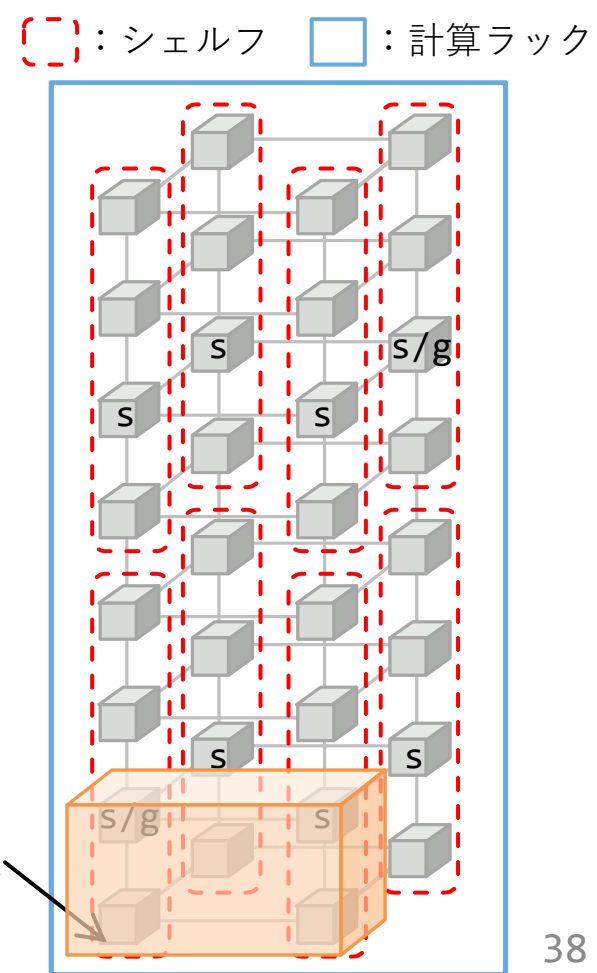
これにより、各計算ノードに対するI/Oノードの位置は毎回同じになり、ジョブを実行するたびにI/O性能が変動することを抑えられる(*1)。

(*1) 「富岳」では計算ラックが432個あり、どの計算ラックのノードに割り当てられるかはジョブ運用ソフトウェアが決定する。strict-ioを指定すると、選択されたノードが属する計算ラックが異なったとしても、ラックの原点が常に始点となる。ラックの原点に対するそのラック内のI/Oノードの相対的位置は各計算ラックで同じであるため、各計算ノードに対するI/Oノードの位置は毎回同じになる。

-  Tofu単位を簡略化した図（12ノード含まれる）
-  ストレージI/Oノードを含むTofu単位
-  ストレージI/OノードとグローバルI/Oノードを含むTofu単位



ラックの原点



MPIジョブ実行に関連する各種基本事項 ▶ Tofu座標（物理座標）におけるノードの配置方法
▶ I/Oノードを意識した割り当て

□ I/Oノードをジョブに専有させる割り当て

I/O性能を最大限に引き出すには、I/Oノードの位置を考慮するだけでなく、投入するジョブにだけI/Oノードを専有させる必要もある。このためには以下で説明するように、I/Oノードが入出力を担う範囲のノードを単位として割り当てる。割り当てる範囲の大きさに対応して、ストレージI/Oノードを専有する方法と、ストレージI/Oノードに加えグローバルI/Oノードも専有する方法がある。

- ◇ ストレージI/Oノードの専有

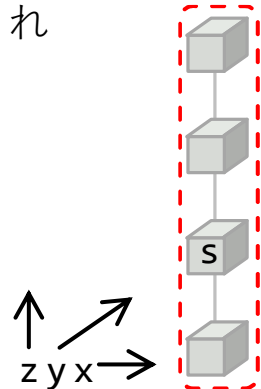
- ◇ グローバルI/Oノードの専有

以降のスライドで順に説明する。

MPIジョブ実行に関連する各種基本事項 ▶ Tofu座標（物理座標）におけるノードの配置方法
▶ I/Oノードを意識した割り当て ▶ I/Oノードをジョブに専有させる割り当て

◇ ストレージI/Oノードの専有

- ストレージI/Oノードを一つのジョブに専有させることで、（後の章で説明する）**LLIO**が管理する第**1**階層ストレージの**I/O**性能を最大限に引き出せる。これを**I/O専有モード**と呼び、**I/O専有モード**のジョブを**I/O専有ジョブ**と呼ぶ。
- **I/O専有モード**では、**シェルフ**単位（z軸方向に 4 Tofu単位分の2x3x8=48ノード）でジョブにノードを割り当てる。割り当てられたシェルフ内に存在するストレージ**I/O**ノードをそのジョブだけが専有でき、そのジョブは他のジョブの入出力の影響を受けない。
- **I/O専有モード**は、ジョブ投入時にパラメータ**io-exclusive**で指定される。



```
$ pjsub -L "node=shape:torus:io-exclusive" job.sh
```

- パラメータ**io-exclusive**は、ノード割り当てモードがトーラスモード（**torus**）の場合に指定でき、メッシュモード（**mesh**）と離散割り当て（**noncont**）では指定できない。
- パラメータ**io-exclusive**は指定できないようにジョブ**ACL**機能によって管理者が設定している場合がある。指定できるかどうかは**pjac1**コマンドで確認できる。確認例を右の図に示す。

【確認例】

```
login1$ pjac1 --rg small
(中略)
execute
(中略)
pjsub(io-exclusive)      disable
```

右の確認例ではパラメータ**io-exclusive**は指定できないように管理者が設定しているようである。

MPIジョブ実行に関連する各種基本事項 ▶ Tofu座標（物理座標）におけるノードの配置方法
▶ I/Oノードを意識した割り当て ▶ I/Oノードをジョブに専有させる割り当て

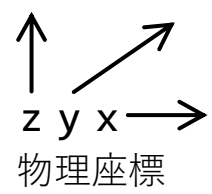
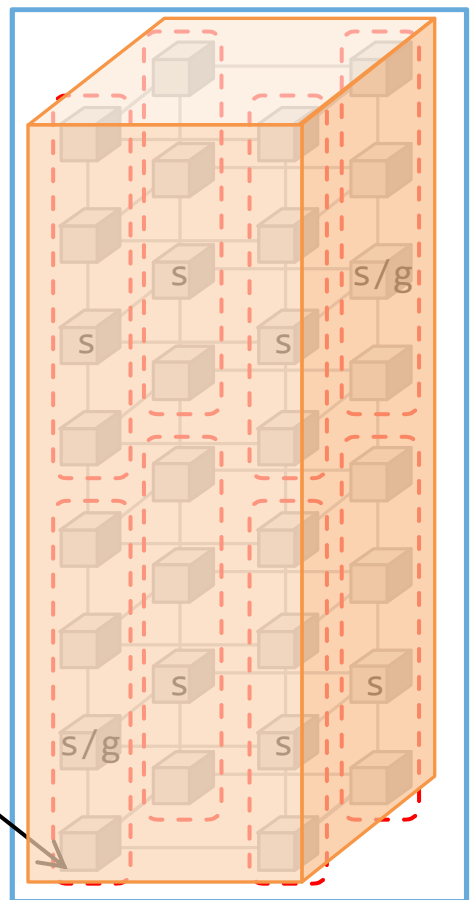
◇ グローバルI/Oノードの専有

- 一つの計算ラック内全体のノードを割り当てることで、その計算ラック内に存在する全てのストレージI/OノードとグローバルI/Oノードを専有することができる。このような専有は、一つの計算ラック内のノードの範囲 **4x6x16** を単位とし、さらにパラメータ **strict-io**を指定することで可能である。

- 一つの計算ラック全体のノードを割り当て、結果としてそのラック内の全てのSIOノードと二つのGIOノードを専有する例を以下のコマンドおよび右の図に示す。

```
$ pjsub -L "node=4x6x16:torus:strict-io" job.sh
```

 : シェルフ : 計算ラック



ラックの原点

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
 - ノード資源の指定
 - Tofu座標（物理座標）におけるノードの配置方法
 - MPIジョブ投入時の指定
 - mpiexecコマンドのオプション概略
 - mpiexecコマンドの `--vcoordfile` オプション
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ MPIジョブ投入時に指定できるパラメータ

MPIジョブ投入時には、**pjsub**コマンドの**--mpi**オプションを利用して、静的に起動するプロセス形状、起動する静的プロセスの最大値、**1**ノードあたりに生成する**MPI**プロセス数の上限値、ランク割付ルールを指定することができる。

■ 静的に起動するプロセス形状

```
pjsub --mpi shape
```

■ 起動する静的プロセスの最大数

```
pjsub --mpi proc
```

■ 1ノードあたりに生成するMPIプロセス数の上限値

```
pjsub --mpi max-proc-per-node
```

■ 生成するプロセスのランク割り当てルール

```
pjsub --mpi rank-map-bychip
```

```
pjsub --mpi rank-map-bynode
```

```
pjsub --mpi rank-map-hostfile
```

以降で各オプションパラメータについて説明する。

■ 静的に起動するプロセス形状

`pjsub` コマンドオプションの `--mpi shape` パラメータを使用することで、`mpiexec` コマンドにより静的に起動されるプロセスの形状を指定できる。

■ `pjsub --mpi shape=shape`

静的に起動するプロセスの形状を *shape* に指定する。プロセスの形状は、1次元、2次元、3次元の形状で、`pjsub -L node=～` で指定するノード形状と同じ次元数を指定する必要がある。

`shape` パラメータ指定例

[1次元形状] `--mpi "shape=NX"`

[2次元形状] `--mpi "shape=NXxNY"`

[3次元形状] `--mpi "shape=NXxNYxNZ"`

- 本 `shape` パラメータが省略された場合は、`pjsub -L node=～` で指定された値が使用される。

⚠ `MPI_Comm_spawn` 関数や `MPI_Comm_spawn_multiple` 関数により動的に生成されるプロセスは、`--mpi shape` で指定された範囲外で生成される。

■ 起動する静的プロセスの最大数

プログラム起動時に静的に生成する最大のプロセス数は、`pjsub --mpi proc`で指定する。

■ `pjsub --mpi proc=n`

プログラム起動時に静的に生成する最大のプロセス数を n に指定する。

- `--mpi proc`で指定可能なプロセス数は、`--mpi shape`で指定した値×48" 以下となる（48は一つのノード内のCPUコア数である）。
 - `--mpi proc`に、`--mpi shape`で指定した値の積×48"より大きい値を指定した場合、ジョブの受付が拒否される。
- `--mpi proc`を省略した場合、`--mpi shape`で指定した値の積（形状から求まるノード数）が使用される。
 - 例えば`--mpi shape=N1xN2`と指定し、`--mpi proc`を指定しない場合、プログラム起動時に静的に生成する最大のプロセス数は $N1 \times N2$ となる。
 - `--mpi shape`も省略した場合は、`-L node`で指定したノード数を採用する。

【node、shape、procパラメータの指定例】

```
#PJM -L "node=4"          # 1次元形状で4ノードをジョブに割り当てる
#PJM --mpi "shape=4"       # 静的プロセス形状を1次元形状で4ノードとする
#PJM --mpi "proc=4"        # mpiexecで生成する静的プロセスの最大数を4と指定する
mpiexec ./a.exe            # 静的プロセス数4でMPIプログラムa.exeを実行する
```

mpiexecコマンドにより、一つのノードに `proc`の指定値/`shape`の指定値 = $4/4 = 1$ 個ずつ、計4個の静的MPIプロセスが生成される。

■ 1ノードあたりに生成するMPIプロセス数の上限値

■ pjsub --mpi max-proc-per-node=*n*

1ノードあたりに生成するMPIプロセス数の上限値を*n*に指定する。

- 指定された値が48（一つのノード内のCPUコア数）を超える場合、ジョブの受付が拒否される。
- 本パラメータを省略した場合、または--mpi procパラメータと同時に指定した場合の動作は以下のようになる。

--mpi proc= <i>n1</i> の指定	--mpi max-proc- per-node= <i>n2</i> の指定	[プログラム開始時] 動作	[動的プロセス生成時] 動作
指定あり	指定なし	ランク割り当てルールに従って、一つのノードに <i>n1/shape</i> 個(*1)または <i>n1/node</i> 個(*2)ずつ（小数点以下切り上げ）、計 <i>n1</i> 個に達するまで、MPIプロセスが生成される。	一つのノード内に最大 <i>n1/shape</i> 個(*1)または <i>n1/node</i> 個(*2)（小数点以下切り上げ）のMPIプロセスが生成できる。
指定なし	指定あり	各ノードに <i>n2</i> 個のMPIプロセスが生成される。	各ノードに <i>n2</i> 個まで、MPIプロセスを生成できる。
指定あり	指定あり	ランク割り当てルールに従って、一つのノードに <i>n2</i> 個まで、計 <i>n1</i> 個に達するまで、MPIプロセスが生成される。(*3)	各ノードに <i>n2</i> 個まで、MPIプロセスを生成できる。

(*1) *shape*: -L *shape* パラメータで指定したノード数。
(*2) *node*: -L *node* パラメータで指定したノード数（-L *shape*パラメータを指定しない場合）。
(*3) *n2*が*n1/shape*より小さい場合、ジョブの受付が拒否される。

■ 生成するプロセスのランク割り当てルール

MPIプロセスの各ランクにノードをどのように割り当てるか（ランク割り当てルール、ランクマップ）は、`pjsub`コマンドの以下の`--mpi`オプションパラメータにより指定することができる。

ランクマップを指定して特定のプロセス間の通信距離が短くなるようにプロセスを配置することで、通信の効率化を図ることができる。

□ `--mpi rank-map-bychip`

連続したランクに対して、同一のノードから順に割り当てる（後述）。

□ `--mpi rank-map-bynode`

連続したランクに対して、別のノードを順に割り当てる（後述）。

- ※ `rank-map-bychip`と`rank-map-bynode`は互いに排他である。
- ※ どちらも指定しない場合、`rank-map-bychip`が指定されたものとみなす。
- ※ 1ノードに一つのランク（プロセス）を生成する場合は、どちらのパラメータを指定してもノードの割り当て方は同じになる。

□ `--mpi rank-map-hostfile=hostfile`

各ランクに対して割り当てるノードをユーザがファイル`hostfile`で指定する（後述）。

□ `--mpi "rank-map-bychip,rank-map-hostfile=hostfile"`

□ `--mpi "rank-map-bynode,rank-map-hostfile=hostfile"`

□ `hostfile`の記述方法に関する注意事項（後述）

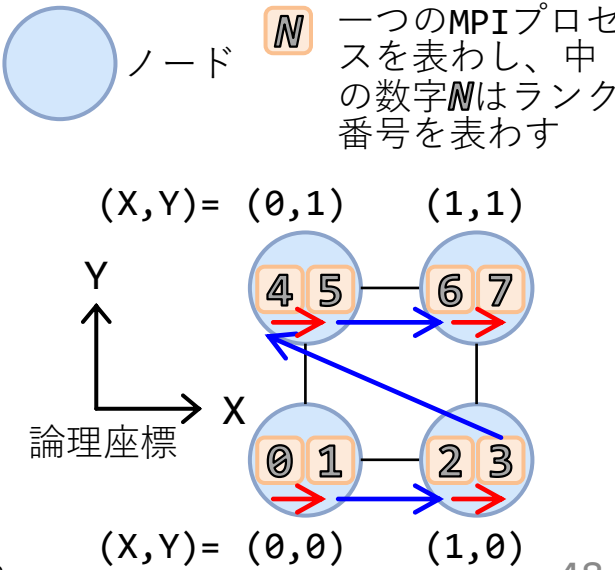
MPIジョブ実行に関連する各種基本事項 ▶ MPIジョブ投入時の指定
▶ 生成するプロセスのランク割り当てルール

```
❑ --mpi "rank-map-bychip[:rankmap]"  
rankmap={XY|YX|XYZ|XZY|YXZ|YZX|ZXY|ZYG}
```

ノードは以下のように割り当てられる。

- 1. 一つのノードに、指定された数のランクを割り当てる（図の赤い矢印）。
 - 一つのノードに割り当てるランクの数は、
"「--mpi proc=」に指定したプロセス数" / "「--mpi shape=」に指定したノード数"
で指定される（小数点以下は切り上げる）。
 - 「--mpi shape=」が指定されない場合は、「-L node=」
で指定された数を使用する。
- 2. 次のノードを選択する。選択するノードの順番は rankmap で指定される（図の青い矢印）。
 - rankmap には「--mpi shape=」で指定した次元数と同じものを指定する：2次元ジョブの場合XYまたはYXを、3次元ジョブの場合XYZ、XZY、YXZ、YZX、ZXYまたはZYGを指定する。1次元の場合、rankmap の指定はできない。
 - rankmap の文字列中で左にある文字に対応する座標成分が先に増す順番でノードを移動する。
 - :rankmap を指定しなかった場合のデフォルトは以下となる。
 - 1次元ジョブの場合は、1次元座標がランク番号になる。
 - 2次元ジョブの場合は、:XY の指定と同じ。
 - 3次元ジョブの場合は、:XYZ の指定と同じ。

```
【例】  
#PJM --mpi "rank-map-bychip:XY"  
#PJM --mpi shape=2x2  
#PJM --mpi proc=8
```



MPIジョブ実行に関連する各種基本事項 ▶ MPIジョブ投入時の指定

▶ 生成するプロセスのランク割り当てルール

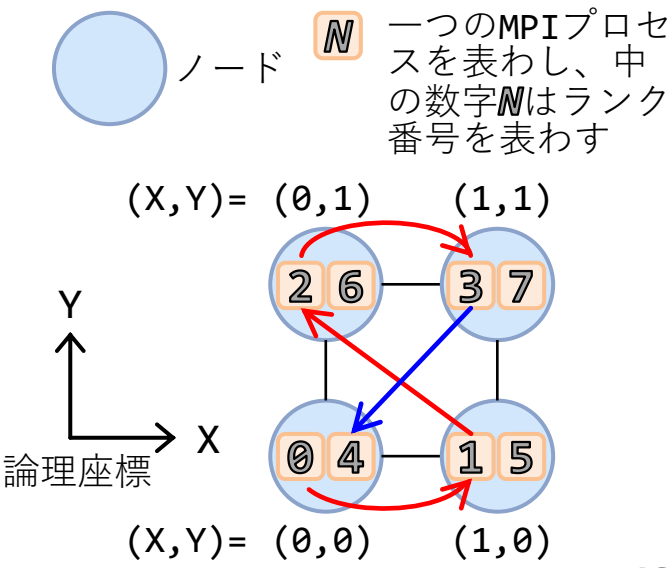
```
❑ --mpi "rank-map-bynode[=rankmap]"  
rankmap={XY|YX|XYZ|XZY|YXZ|YZX|ZXY|ZYG}
```

ノードは以下のように割り当てられる。

- 1. 一つのランクにノードを割り当てると、次のランクには別のノードを割り当てる。割り当てるノードの順番は、rankmapで指定される（図の赤い矢印）。
- rankmapには「--mpi shape=」で指定した次元数と同じものを指定する：2次元ジョブの場合XYまたはYXを、3次元ジョブの場合XYZ、XZY、YXZ、YZX、ZXYまたはZYGを指定する。1次元の場合、rankmapの指定はできない。
- rankmapの文字列中で左にある文字に対応する座標成分が先に増す順番でノードを移動する。
- =rankmap を指定しなかった場合のデフォルトは以下となる。
 - 1次元ジョブの場合は、1次元座標がランク番号になる。
 - 2次元ジョブの場合は、=XY の指定と同じ。
 - 3次元ジョブの場合は、=XYZ の指定と同じ。
- 2. rankmapで指定されたノードを一巡したら、最初に割り当てたノードに戻る（図の青い矢印）。
- 3. 1と2を繰り返して、すべてのランクにノードを割り当てる。

【例】

```
#PJM --mpi "rank-map-bynode=XY"  
#PJM --mpi shape=2x2  
#PJM --mpi proc=8
```



MPIジョブ実行に関連する各種基本事項 ▶ MPIジョブ投入時の指定

▶ 生成するプロセスのランク割り当てルール

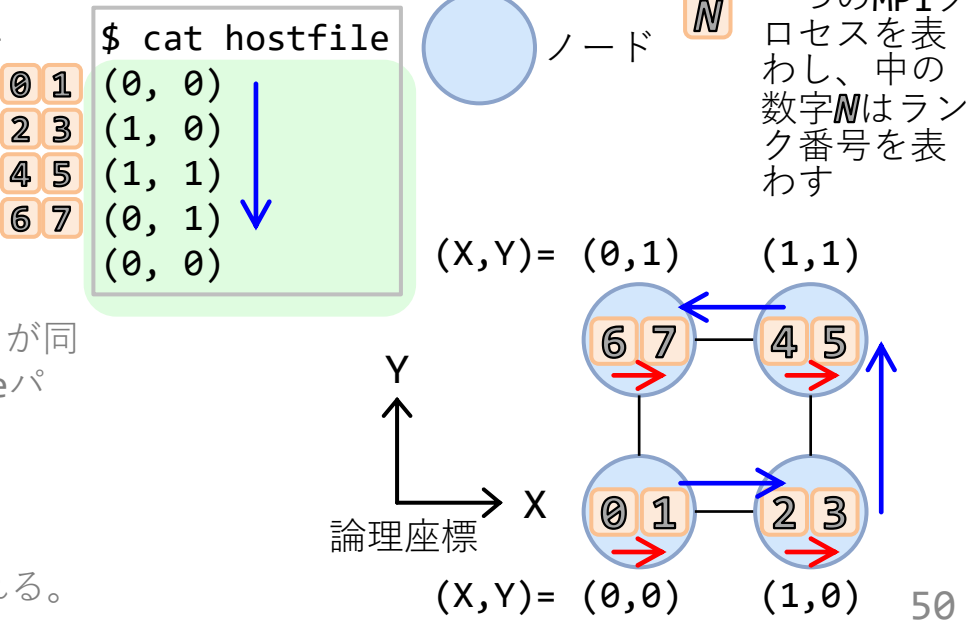
□ `--mpi "rank-map-bychip,rank-map-hostfile=hostfile"`

ノードは以下のように割り当てられる。

- 1. 一つのノードに、指定された数のランクを割り当てる (図の赤い矢印)。
 - 一つのノードに割り当てるランクの数は、
"「`--mpi proc=`」に指定したプロセス数" / "「`--mpi shape=`」に指定したノード数"
で指定される (小数点以下は切り上げる)。
 - 「`--mpi shape=`」が指定されない場合は、
「`-L node=`」で指定された数を使用する。
- 2. 次のノードを選択する。選択するノードの順番は、`rank-map-hostfile`で指定されるファイル`hostfile`に記述された順となる (図の青い矢印)。
 - 一行につき一つの座標を記述する。
 - 割り当てるノードはプロセスの形状に合わせて1次元、2次元、または3次元の論理座標で指定する。

【例】

#PJM -L "node=2x2"
#PJM --mpi "proc=8"
#PJM --mpi "rank-map-hostfile=hostfile"
#PJM --mpi "rank-map-bychip"



※ `rankmap`と`rank-map-hostfile`パラメータが同時に指定された場合、`rank-map-hostfile`パラメータが優先する。

- 3. 1と2を繰り返して、すべてのランクにノードを割り当てる。
 - ※ `hostfile`ファイル中の余った行は無視される。

MPIジョブ実行に関連する各種基本事項 ▶ MPIジョブ投入時の指定

▶ 生成するプロセスのランク割り当てルール

❑ `--mpi "rank-map-bynode,rank-map-hostfile=hostfile"`

ノードは以下のように割り当てられる。

1. 一つのランクにノードを割り当てると、次のランクには別のノードを割り当てる。割り当てるノードの順番は、`rank-map-hostfile`で指定されるファイル`hostfile`に記述された順となる（図の赤い矢印）。

- 一行につき一つの座標を記述する。
- 割り当てるノードはプロセスの形状に合わせて1次元、2次元、または3次元の論理座標で指定する。

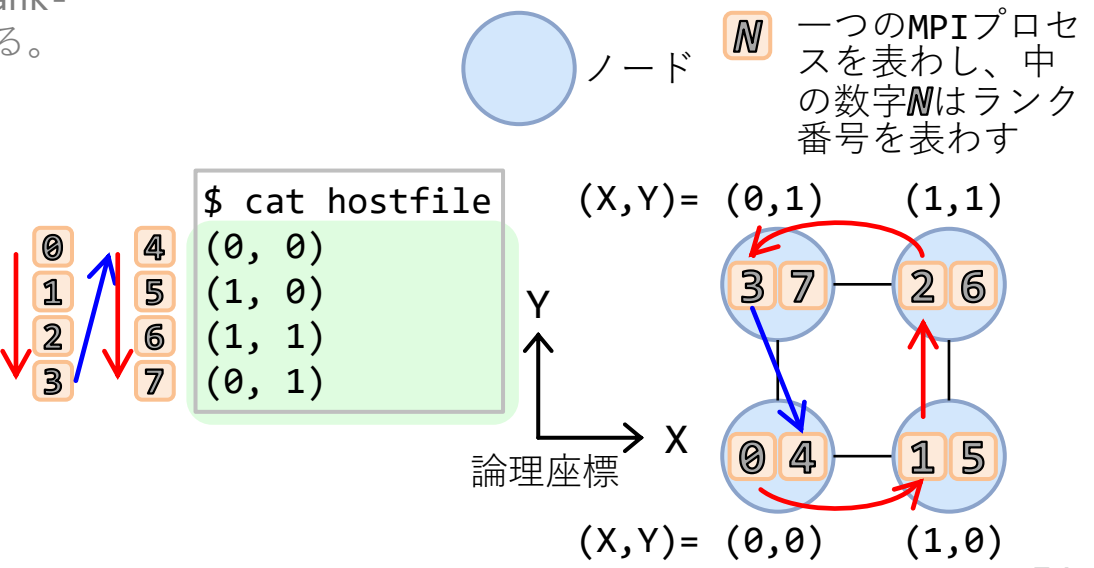
※ `rankmap`と`rank-map-hostfile`パラメータが同時に指定された場合、`rank-map-hostfile`パラメータが優先する。

2. `hostfile`の行数（座標の数）がプロセス数より少なく`hostfile`で指定されたノードを一巡した場合、最初に割り当てたノードに戻る（図の青い矢印）。

3. 1と2を繰り返して、すべてのランクにノードを割り当てる。

【例】

```
#PJM -L "node=2x2"
#PJM --mpi "proc=8"
#PJM --mpi "rank-map-hostfile=hostfile"
#PJM --mpi "rank-map-bynode"
```



MPIジョブ実行に関連する各種基本事項 ▶ MPIジョブ投入時の指定

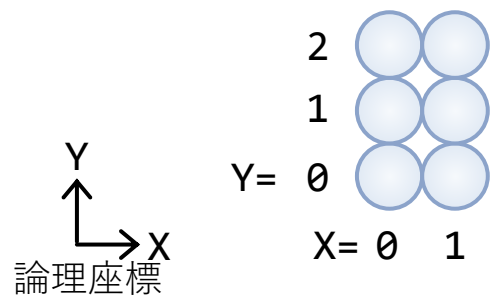
▶ 生成するプロセスのランク割り当てルール

□ *hostfile*の記述方法に関する注意事項 [1/3]

■ 一般的な注意事項

- ファイル*hostfile*内の空行は無視される。
- ファイル*hostfile*内に記述する座標はノードの形状を表す*shape*パラメータで指定した範囲内の値でなければならない。
 - 例えば、*shape=2x3* が指定された場合、記述できる座標は (0,0)、(0,1)、(0,2)、(1,0)、(1,1)、(1,2) である。

```
#PJM --mpi "shape=2x3"  
#PJM --mpi "rank-map-hostfile=hostfile"  
...
```



```
$ cat hostfile  
(0, 0)  
(0, 1)  
(0, 2)  
(1, 0)  
(1, 1)  
(1, 2)
```

□ *hostfile*の記述方法に関する注意事項 [2/3]

■ rank-map-hostfile=*hostfile*をrank-map-bychipと共に指定する場合の注意事項

rank-map-hostfileパラメータをrank-map-bychipパラメータと共に指定する場合、ファイル*hostfile*内の記述は以下に従う必要がある。

- *hostfile*内に記述する座標の個数はshapeパラメータで指定された形状が示すノード数と同じにする必要がある。
 - 例えば、shape=3x2の場合、ノード数は6台なので、ファイル*hostfile*内には6個の座標を記述する。

```
#PJM --mpi "shape=3x2"
#PJM --mpi "rank-map-hostfile=hostfile"
#PJM --mpi "rank-map-bychip"
...
```

```
$ cat hostfile
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

- 記述した座標の数がshapeパラメータで指定した形状が示すノード数よりも少ない場合、pjsubコマンドはジョブの受付けを拒否する。
- 記述した座標の数がshapeパラメータで指定した形状が示すノード数よりも多い場合、残りの座標は無視される。
- *hostfile*内には、複数の同じ座標は記述できない。同じ座標を記述した場合は、pjsubコマンドがエラーになる。

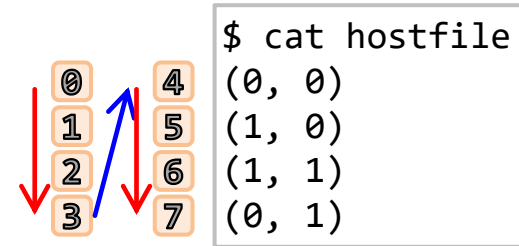
▶ 生成するプロセスのランク割り当てルール

□ *hostfile*の記述方法に関する注意事項 [3/3]

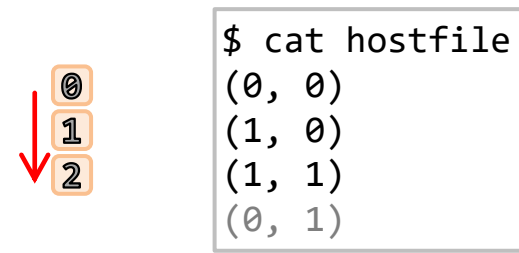
■ rank-map-hostfile=*hostfile*をrank-map-bynodeと共に指定する場合の注意事項

rank-map-hostfileパラメータをrank-map-bynodeパラメータと共に指定する場合、ファイル*hostfile*内の記述は以下に従う必要がある。

- *hostfile*内に記述した座標の数がprocパラメータで指定したプロセス数よりも少ない場合、最後の座標のノードまで割り当てたら最初の座標のノードに戻って割り当てる。



- *hostfile*内に記述した座標の数がprocパラメータで指定したプロセス数よりも多い場合、残りの座標は無視される。



- *hostfile*内には、同じ座標は計算ノード当たりのCPUコア数以下であれば記述できる。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
 - ノード資源の指定
 - Tofu座標（物理座標）におけるノードの配置方法
 - MPIジョブ投入時の指定
 - **mpiexecコマンドのオプション概略**
 - mpiexecコマンドの `--vcoordfile` オプション
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ MPI並列プログラミングのモデル

MPIを使用した並列プログラミングのモデルはプログラムの実行コマンドの形式の観点で以下の二つに分類される。

1. SPMDモデル (Single Program/Multiple Dataモデル)

このモデルでは全てのプロセスで同一のMPIプログラムを使用する。また、プロセスごとに異なるデータを使用し処理を分担して動作する。

2. MPMDモデル (Multiple Program/Multiple Dataモデル)

このモデルでは二つ以上の異なるMPIプログラムを使用する。また、プロセスごとに異なるデータを使用し処理を分担して動作する。

■ 実行コマンドの形式

実行コマンドは計算ノードで実行する。実行コマンドの形式は、SPMDモデルで実行する場合とMPMDモデルで実行する場合とで異なる。

1. SPMDモデルで実行する場合

```
mpiexec global_options local_options execfile execfile_arguments
```

指定例 `$ mpiexec -of-proc procfile -mca mpi_print_stats 2 ./a.exe`

MPIプログラムa.exeを実行する。プロセスごとに、プログラムの出力結果と統計情報（後述）が、指定の方法で生成される名前のファイルに出力される。

2. MPMDモデルで実行する場合

```
mpiexec global_options local_options execfile1 execfile1_arguments
      : local_options execfile2 execfile2_arguments
      [ : local_options execfile3 execfile3_arguments ] ...
```

指定例 `$ mpiexec -n 2 ./a.exe : -n 4 ./b.exe : -n 6 ./c.exe`

三つのMPIプログラムa.exe、b.exe、およびc.exeを、それぞれ2、4、および6の並列プロセス数で実行する。

- mpiexec コマンドのコマンド文字列に、1個のコロン「:」だけからなる文字列を指定した場合、区切り文字としてみなされる。例えば、execfile または execfile_argument として、1個のコロンだけからなる文字列は指定できない。
- Java プログラムの場合、MPMDモデルでの実行は保証されない。

■ global_options のオプション概略（抜粋）

mpiexec コマンドの global_options に指定可能な主なオプション（抜粋）を表に示す。

```
mpiexec global_options ~
```

なお、これらのオプション文字列の最初のハイフンは二つにすることも可能（例：-stdin の代わりに --stdin と指定することも可能）。

オプションの名前と形式	概略
-stdin STDIN_FILE	全ての並列プロセスの標準入力を単一のファイルから読み込む。
{ -ofout-proc -stdout-proc } OUT_PROC_FILE	各並列プロセスの標準出力をプロセスごとに出力する。
{ -oferr-proc -stderr-proc } ERR_PROC_FILE	各並列プロセスの標準エラー出力をプロセスごとに出力する。
{ -of-proc -std-proc } PROC_FILE	各並列プロセスの標準出力と標準エラー出力をプロセスごとに出力する。
{ -ofprefix -stdprefix } date,rank,nid	並列プロセスの標準出力と標準エラー出力の行の先頭に、それぞれ出力時刻、MPI_COMM_WORLD でのランク番号、ノード ID を出力する。
{ -vcoordfile --vcoordfile } VCOORD	VCOORD ファイル内に記述されたプロセスに割り当て情報に基づいて並列プロセスを割り当てることを指定する。
{ -debuglib --debuglib }	デバッグ用の MPI ライブラリを結合することを指定する。実行時の動的デバッグ機能の一つである引数チェック機能が使用できる。本オプションを指定した場合、デバッグ用の MPI ライブラリが結合されるため、MPI プログラムの実行時間が非常に遅くなるので、デバッグ時に使用する。

□ 標準出力関連オプションの注意事項

- 運用上の制限事項として、2022年10月20日以降、ジョブの標準出力・標準エラー出力の出力方法が以下のように変更されています。
- デフォルトでは、並列プロセスからの標準出力・標準エラー出力はプロセス（ランク）ごとに出力され、1000ランクごとにディレクトリを分割して配置される。

```
$ pjacl --rg small
(中略)
defines
(中略)
    mpiexec-stdouterr-unit
    mpiexec-stdout
    mpiexec-stderr
(以下略)
```

%jはジョブID、%/1000rはランク番号やspawn
番号を1000単位で切り捨て、%mはジョブ内での
mpiexecコマンドの実行回数を表すメタ文字。

```
proc
"./output.%j/%/1000r/%m/stdout"
"./output.%j/%/1000r/%m/stderr"
```

- 全並列プロセスからの標準出力および標準エラー出力をそれぞれ単一のファイル（`mpiexec`単位）でまとめて出力することはできない。全並列プロセスからの出力先を単一のファイルに指定するオプション`-stdout`（標準出力）、`-stderr`（標準エラー出力）、または`-std`（標準出力および標準エラー出力）を指定しても警告メッセージが出力され、その指定は無視され、デフォルト設定が適用される。
- 詳細は以下のURLを参照して下さい。

https://www.fugaku.r-ccs.riken.jp/restriction/20221017_01

- 出力するプロセスをユーザプログラムが限定している場合であっても、想定外のエラーが発生し全てのプロセスからエラーメッセージが出力されることがある。

□ 標準出力に関連する環境変数 PLE_MPI_STD_EMPTYFILE

- 並列プロセスの標準出力/標準エラー出力への出力がない場合に、空ファイルを作成するかどうかを指定する。
 - on : 作成する
 - off : 作成しない
- on、off以外の値が指定された場合は、ジョブ運用ソフトウェアのジョブACL機能 (pjaclコマンドの“mpiexec-std-emptyfile”)の設定に従います。

```
$ pjacl --rg small
(中略)
defines
(中略)
    mpiexec-std-emptyfile      off
(以下略)
```

【例】 環境変数PLE_MPI_STD_EMPTYFILEの値を"off"にする指定により標準出力や標準エラー出力がない場合に空ファイルは作成しない。

```
export PLE_MPI_STD_EMPTYFILE="off"
```

□ 大規模MPIジョブを実行する場合の注意点

1万以上のMPIプロセスを生成するジョブでは、ファイルに書き出す処理のシステム負荷を考慮して、以下のように実行して下さい。なお、運用上の制限事項として、下記①と②はデフォルトとして適用されます。③もデフォルトで適用されます。

- ① 標準出力や標準エラー出力をプロセス（ランク）ごとに別ファイルへ出力するオプション（`-std-proc`または、`-stdout-proc`と`-stderr-proc`）を使用する。
- ② 各プロセスの標準出力／標準エラー出力ファイルは同じディレクトリに出力せず、いくつかのランクごとにディレクトリを分割して出力する。
- ③ プロセスからの標準出力／標準エラー出力がない場合に、環境変数 `PLE_MPI_STD_EMPTYFILE` の値を `"off"` に指定することにより、空ファイルを作成しない。

■ local_options のオプション概略（抜粋） [1/2]

mpiexec コマンドの local_options に指定可能な主なオプション（抜粋）を表に示す。

```
mpiexec global_options local_options ~
```

オプションの名前と形式	概略
{ -c -np --np -n --n } N	対応するMPIプログラムの並列プロセス数をNに指定する。 SPMDモデルでの実行では、本オプションの指定が省略された場合、生成できる最大値が指定されたものとみなされる。 MPMDモデルでの実行では、本オプションを必ず指定する必要がある。 本オプションを重複して指定した場合、最後に指定したパラメータが優先する。
-x NAME=VALUE	対応するMPIプログラムに対して環境変数に値を設定することを指定する。 NAMEは環境変数名を表わす。VALUEはその環境変数に設定する値を表わす。 指定に空白を含めたい場合、"NAME=VALUE"のように引用符で括る。 複数の環境変数を指定したい場合、 -x OMP_NUM_THREADS=8 -x THREAD_STACK_SIZE=4096 のように指定する。 同一の環境変数名を重複して指定した場合、最後に指定した内容が優先する。
(次スライドへ続く)	

■ local_options のオプション概略（抜粋） [2/2]

mpiexec コマンドの local_options に指定可能な主なオプション（抜粋）を表に示す。

```
mpiexec global_options local_options ~
```

オプションの名前と形式	概略
(前スライドからの続き)	
{ -mca --mca } MCA パラメータ名 値	対応する MPI プログラムに対して MCA パラメータを指定する。すべてのプログラムに対する MCA パラメータの指定内容が同じ値になるように指定する必要がある。異なる値が指定された場合には動作は保証されない。
-tune MCA パラメータの設定ファイル	対応する MPI プログラムに対して、MCA パラメータの設定ファイルのパス名を指定する。 なお、MCA パラメータの設定ファイルを「AMCA パラメータファイル」と呼ぶ。 すべてのプログラムに対する MCA パラメータの指定内容（値）が同じになるように指定する必要がある。 異なる値が指定された場合、動作は保証されない。 ファイル内での指定方法は、次のとおりである。 <ul style="list-style-type: none">各行には、以下の形式で指定する。 MCA パラメータ名=値同一の MCA パラメータに複数の値を指定する場合には、以下のようにコンマ(,)で区切って指定する。 MCA パラメータ名=値1, 値2 本オプションを重複して指定した場合、最後に指定したパラメータが優先する。

□ MCAパラメータとは

「富岳」では、MPIプログラムの実行の際、MPIライブラリ内部の変数の値を一時的に変更することによって、MPIライブラリの動作条件を変更できる。このような変数をMCAパラメータと呼ぶ。

本講義では重要なMCAパラメータについて、以降の各章で随時説明します。

□ MCAパラメータを設定するmpiexecのオプション `-mca`

{ `-mca` | `--mca` } MCAパラメータの名前 MCAパラメータの値

- 対応するMPIプログラムに対してMCAパラメータを指定する。
- すべてのプログラムに対するMCAパラメータの指定内容が同じ値になるように指定する必要がある。異なる値が指定された場合には動作は保証されない。
- `-x` オプションと異なり、名前と値の間はスペースであることに注意して下さい。

□ MCAパラメータの設定ファイルを指定：-tune オプション

-tune MCAパラメータの設定ファイル

- 対応するMPIプログラムに対して、MCAパラメータの設定ファイルのパス名を指定する。なお、MCAパラメータの設定ファイルを「AMCAパラメータファイル」と呼ぶ。
- すべてのプログラムに対するMCAパラメータの指定内容（値）が同じになるように指定する必要がある。異なる値が指定された場合、動作は保証されない。
- ファイル内での指定方法は、次のとおりである。
 - 各行には、以下の形式で指定する。

MCAパラメータ名=値

- 同一のMCAパラメータに複数の値を指定する場合には、以下のようにコンマ(,)で区切って指定する。

MCAパラメータ名=値1, 値2

- 本オプションを重複して指定した場合、最後に指定したパラメータが優先する。

□ MCA パラメータの各種指定方法と優先順位

「富岳」における MCA パラメータの指定方法には、**-mca** オプションで指定する方法を含めて以下の三通りの方法がある。項番の数字が小さいほど優先順位が高い。

1. mpiexec コマンドの **-mca** オプションを使用して、MCA パラメータを直接指定する。

例： `$ mpiexec -mca btl_toFU_eager_limit 4096 ./a.exe`

2. MCA パラメータを環境変数によって指定する。

例： `$ export OMPI_MCA_btl_toFU_eager_limit=4096`
`$ mpiexec ./a.exe`

3. mpiexec コマンドの **-tune** オプションを使用して、MCA パラメータの設定ファイル（AMCA パラメータファイル）内に指定する。

例： `$ cat mca_file`
`btl_toFU_eager_limit=4096`

`$ mpiexec -tune mca_file ./a.exe`

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
 - ノード資源の指定
 - Tofu座標（物理座標）におけるノードの配置方法
 - MPIジョブ投入時の指定
 - mpiexecコマンドのオプション概略
 - mpiexecコマンドの `--vcoordfile` オプション
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ VCOORDファイルを使用したノードの割り当て

ランクに対するノードの割り当ての指定方法には、前述のrank-map-hostfileパラメータの他、VCOORDファイルを使用する方法がある。VCOORDファイルは以下のように指定する（詳細は後述する）。

□ 静的に生成するプロセスに対する指定：

- mpiexec コマンドの--vcoordfileオプションの引数にVCOORDファイルを指定する。

□ 動的に生成するプロセスに対する指定：

- MPI_Comm_spawn関数やMPI_Comm_spawn_multiple関数のinfoキーvcoordfileにVCOORDファイルを指定する。

■ VCOORDファイルによる指定の特徴

■ rank-map-hostfileによる指定とVCOORDファイルによる指定の特徴を以下の表に示す。

比較する特徴	pjsubコマンドのrank-map-hostfileパラメータによる指定	VCOORDファイルによる指定
指定の適用範囲	ジョブ内で実行する全てのMPIプログラムに共通の指定になる。	実行するMPIプログラムごとに指定を変えることができる。つまり、mpiexecコマンドの実行ごと、あるいはMPI_Comm_spawnやMPI_Comm_spawn_multiple関数の実行ごとに、指定を変えることができる。
mpiexecによるMPMD方式での異なるMPIプログラムのプロセスを、同じノードへ割り当てられるかどうか	異なるノードへの割り当てとなる。	同じノードへの割り当てが可能である。
静的に生成するMPIプロセスと動的に生成するMPIプロセスを、同じノードへ割り当てられるかどうか	異なるノードへの割り当てとなる。	同じノードへの割り当てが可能である。

■ ランクに対するノードの割り当ては、rank-map-hostfileパラメータによる指定よりも --vcoordfileオプションによる指定が優先する。

□ 静的に生成するプロセスに対するVCOORDファイルの指定方法

静的に生成するプロセスに対するVCOORDファイルの指定は、`mpiexec` コマンドの `--vcoordfile` オプションの引数にVCOORDファイルを指定することで行う。

- `pjsub` コマンドの実行前あるいは`mpiexec` コマンドの実行前に、プロセス割り当て情報を記述したVCOORDファイルを用意する。記述方法は後のスライドで示す。

VCOORD ファイル

(記述形式は後述)

- `mpiexec` コマンドの `--vcoordfile` オプションの引数に上記のVCOORDファイルのパスを指定する。

`mpiexec { -vcoordfile | --vcoordfile } VCOORD ファイル名`

- 本オプションはMPIプログラムを実行する際、VCOORDファイル内に指定されたプロセス割り当て情報に基づいて並列プロセスを割り当てることを指定する。
- ファイル名だけまたは相対パスを指定した場合、`mpiexec` コマンド実行時のカレントディレクトリからの相対パスとなる。
- `mpiexec` のバックグラウンド実行 (`&` を用いた実行) を利用して、複数の`mpiexec` コマンドを同時に実行させる場合、本オプションを必ず指定する必要がある。なお、バックグラウンド実行で同時に実行可能な個数は128個までである。
- 本オプションを重複して指定した場合、最後に指定したパラメータが優先する。

□ 動的に生成するプロセスに対するVCOORDファイルの指定方法

動的に生成するプロセスに対するVCOORDファイルの指定は、生成元のMPIプログラム内のMPI_Comm_spawn関数やMPI_Comm_spawn_multiple関数のinfoキーvcoordfileにVCOORDファイルを指定することで行う。

- MPI_Comm_spawn{_multiple}関数の実行前に、プロセス割り当て情報を記述したVCOORDファイルを用意する。記述方法は後のスライドで示す。

VCOORD ファイル
(記述形式は後述)

- MPI_Comm_spawn{_multiple}関数の引数infoのキーvcoordfileに上記のVCOORDファイルを指定する。

- 生成元のプログラムa.exeからb.exeを動的に生成する場合の指定例を以下に示す。

生成元のプログラムa.exeのソースファイルa.cpp (抜粋)

```
MPI_Info info;
MPI_Info_create(&info); // MPI_Infoオブジェクトの生成
MPI_Info_set(info, "vcoordfile", VCOORDファイル名); // VCOORDファイルを指定
int nprocs_b = 2; // 生成するプロセス数
MPI_Comm intercomm_b; // 生成元と生成されたプロセスで作るコミュニケーター
MPI_Comm_spawn("./b.exe", MPI_ARGV_NULL, nprocs_b, info,
               0, MPI_COMM_SELF, &intercomm_b, MPI_ERRCODES_IGNORE);
```

■ VCOORDファイルの記述形式 [1/2]：3種類の形式

VCOORDファイルには、MPIプロセスを割り当てるノードの論理座標、各プロセスで使用するCPUコア数、メモリ割り当て方法、CPUコア割り当て方法を指定することができる。以下の形式のいずれかの形式で指定する。

■ 形式1. 論理座標とCPUコア数を指定する

ノードの論理座標 使用コア数 [NUMAメモリ割り当て方法] [CPUコア割り当て方法]

■ 形式2. 論理座標だけ指定する

ノードの論理座標 [NUMAメモリ割り当て方法] [CPUコア割り当て方法]

■ 形式3. CPUコア数だけ指定する

使用コア数 [NUMAメモリ割り当て方法] [CPUコア割り当て方法]

- 指定する項目について記述する内容は次のスライドで示す。
- ファイルの1行目がランク0、2行目がランク1、...に対する指定である。
- 一つのVCOORDファイルにつき、形式1～3のいずれかの形式で指定する。
- 形式1と2において「ノードの論理座標」の指定は必須である。「ノードの論理座標」の指定は行の最初の項目でなければならない。同一の座標を複数行に記述することで、同一座標に複数のプロセスを生成することが可能である。
- 形式1と3において「使用コア数」の指定は必須である。「使用コア数」、「NUMAメモリ割り当て方法」、「CPUコア割り当て方法」の二つ以上を指定する場合、順番に制約はない。
- 各形式において、[NUMAメモリ割り当て方法]と[CPUコア割り当て方法]は、指定をする場合としない場合を、各行で選択できる。

■ VCOORDファイルの記述形式 [2/2]：記述する内容

指定したい各項目に対して記述する内容は以下の表の通りである。

指定する項目		記述する内容
ノードの論理座標	1次元形状の場合：	(X)
	2次元形状の場合：	(X,Y)
	3次元形状の場合：	(X,Y,Z)
使用コア数		core=N
NUMAメモリ割り当て方法		memory_allocation_policy=設定値(*1)
CPUコア割り当て方法		numanode_assign_policy=設定値(*2)

- (*1) 設定値にはNUMAメモリの割り当てポリシーを指定するMCAパラメータ `plm_ple_memory_allocation_policy`に指定する以下の値が指定可能（本講義では説明しません。詳細は「MPI使用手引書」を参照して下さい）：

`localalloc`、`interleave_local`、`interleave_nonlocal`、`interleave_all`、`bind_local`、`bind_nonlocal`、`bind_all`、`prefer_local`、`prefer_nonlocal`。

VCOORDファイルでの本指定と本MCAパラメータでの指定とではVCOORDファイルの指定が優先する。どちらも指定がない場合は`localalloc`がデフォルトである。
- (*2) 設定値にはNUMAノードへのCPUコア割り当てポリシーを指定するMCAパラメータ `plm_ple_numanode_assign_policy`に指定する以下の値が指定可能（本講義では説明しません。詳細は「MPI使用手引書」を参照して下さい）：

`simplex`、`share_cyclic`、`share_band`。

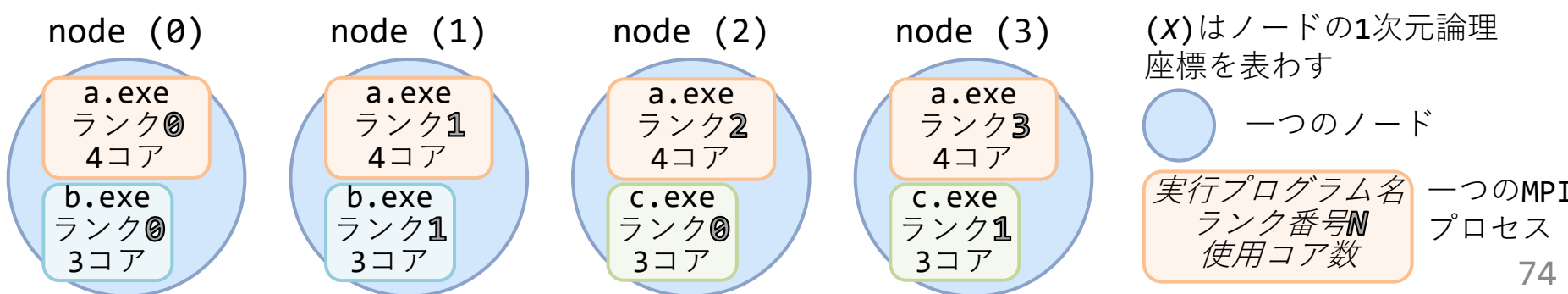
VCOORDファイルでの本指定と本MCAパラメータでの指定とではVCOORDファイルの指定が優先する。どちらも指定がない場合は`share_cyclic`がデフォルトである。

■ VCOORDファイルの使用例：同一ノード上で複数MPIプログラムを実行（静的プロセス） [1/2]

四つのノード上に3種類のMPIプログラムa.exe、b.exe、およびc.exeを、下図のように配置したいとする：

- a.exeのプロセスを1ノード当たり1プロセス生成するとし、合計4プロセス生成するとする。論理座標 $X=N$ ($N=0,1,2,3$) のノードに、ランク N のプロセスを配置する。各プロセスは4コアを使用する。
- b.exeのプロセスも1ノード当たり1プロセス生成するとし、合計2プロセス生成するとする。論理座標 $X=N$ ($N=0,1$) のノードに、ランク N のプロセスを配置する。各プロセスは3コアを使用する。
- c.exeのプロセスも1ノード当たり1プロセス生成するとし、合計2プロセス生成するとする。論理座標 $X=N$ ($N=2,3$) のノードに、ランク N のプロセスを配置する。各プロセスは3コアを使用する。

次のスライドで、上記のように配置する場合のVCOORDファイルの指定方法と実行方法を示す。



■ VCOORDファイルの使用例：同一ノード上で複数MPIプログラムを実行（静的プロセス） [2/2]

- a.exeに対するvcoordfileファイルvfile_aは、論理座標X=N (N=0,1,2,3) のノードに、a.exeのランクNのプロセスを、各四つのコアを使用して生成することを指定する。
- b.exeに対するvcoordfileファイルvfile_bは、論理座標X=N (N=0,1) のノードに、b.exeのランクNのプロセスを、各三つのコアを使用して生成することを指定する。
- c.exeに対するvcoordfileファイルvfile_cは、論理座標X=N (N=2,3) のノードに、c.exeのランクNのプロセスを、各三つのコアを使用して生成することを指定する。
- 一度に複数のMPIプログラムを実行するには & を付けてmpiexecコマンドをバックグラウンド実行する。

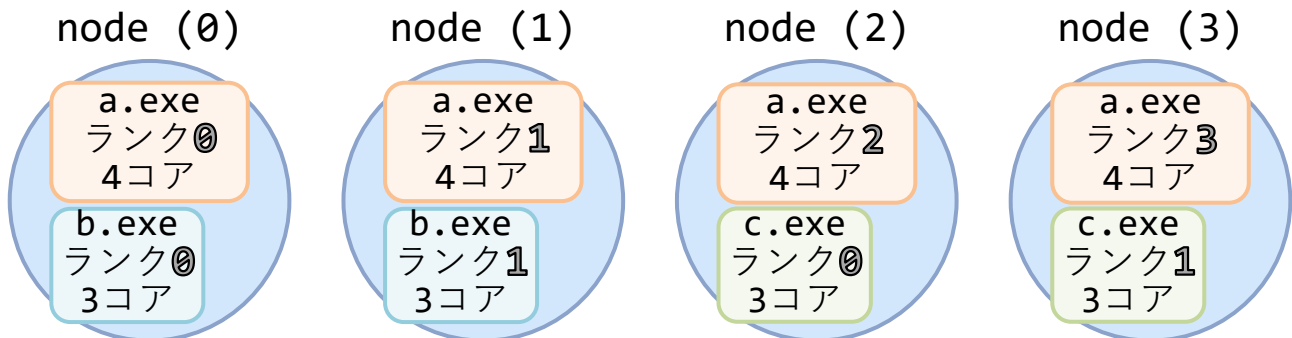
```
#!/bin/bash
#PJM -L node=4
#PJM --mpi max-proc-per-node=4
mpiexec --vcoordfile vfile_a -std-proc stdouterr ./a.exe &
mpiexec --vcoordfile vfile_b -std-proc stdouterr ./b.exe &
mpiexec --vcoordfile vfile_c -std-proc stdouterr ./c.exe
```

```
$ cat vfile_a
(0) core=4
(1) core=4
(2) core=4
(3) core=4
```

```
$ cat vfile_b
(0) core=3
(1) core=3
```

```
$ cat vfile_c
(2) core=3
(3) core=3
```

N N N ランク番号
(X)はノードの1次元論理座標を表わす



(X)はノードの1次元論理座標を表わす

● 一つのノード

実行プログラム名 一つのMPI
 ランク番号N プロセス
 使用コア数

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ 「富岳」のストレージ構成とLLIO

本章の内容

多数のMPIプロセスを必要とするMPIプログラムを実行し、各プロセスがI/Oを行う場合、プログラムの効率的な実行のためには、I/O効率の向上が重要となることがあります。I/Oの効率化を検討するためには「富岳」のストレージ構成を理解しておくことが必要です。本章では、そのための基礎知識として、まず「富岳」のストレージ構成の概要について説明します。

「富岳」では、ストレージに対するアプリケーション・プログラムのI/O特性に応じてI/O性能を最適化・高速化する目的でLLIOという技術が導入されています。本章ではLLIOの機能の概略、および幾つかの機能を取り上げて説明します。

- 「富岳」ストレージ構成の概要
- LLIOの機能概略
- 第1階層ストレージの三つの領域
- 第1階層ストレージのサイズ
- 共通ファイル配布機能
- 【参考】LLIO使用サンプル

■ 「富岳」のストレージ構成とLLIO

本章の内容

多数のMPIプロセスを必要とするMPIプログラムを実行し、各プロセスがI/Oを行う場合、プログラムの効率的な実行のためには、I/O効率の向上が重要となることがあります。I/Oの効率化を検討するためには「富岳」のストレージ構成を理解しておくことが必要です。本章では、そのための基礎知識として、まず「富岳」のストレージ構成の概要について説明します。

「富岳」では、ストレージに対するアプリケーション・プログラムのI/O特性に応じてI/O性能を最適化・高速化する目的でLLIOという技術が導入されています。本章ではLLIOの機能の概略、および幾つかの機能を取り上げて説明します。

■ 「富岳」ストレージ構成の概要

- LLIOの機能概略
- 第1階層ストレージの三つの領域
- 第1階層ストレージのサイズ
- 共通ファイル配布機能
- 【参考】LLIO使用サンプル

■ 「富岳」のストレージ構成

「富岳」のストレージは、第1階層ストレージ、第2階層ストレージ、および第3階層ストレージから構成される。（第3階層ストレージは商用クラウドストレージおよびHPCIストレージであり、本講義では以下省略します。）

■ 第1階層ストレージ

第1階層ストレージは富士通が開発したLLIO（**L**ightweight **L**ayered **I**O-**A**ccelerator）によって管理されるSSDストレージである。

LLIOは、ジョブ用の一時ファイルを第2階層ストレージに書き出さない機能、または第1階層ストレージから第2階層ストレージへの書き出しを計算処理中に非同期に行う機能により、高速化を実現する。

第1階層ストレージはジョブ専用であり、ジョブ内からアクセスされジョブ実行時に一時的なストレージとして利用される。ジョブごとに存在し、他のジョブからは利用できない。

■ 第2階層ストレージ

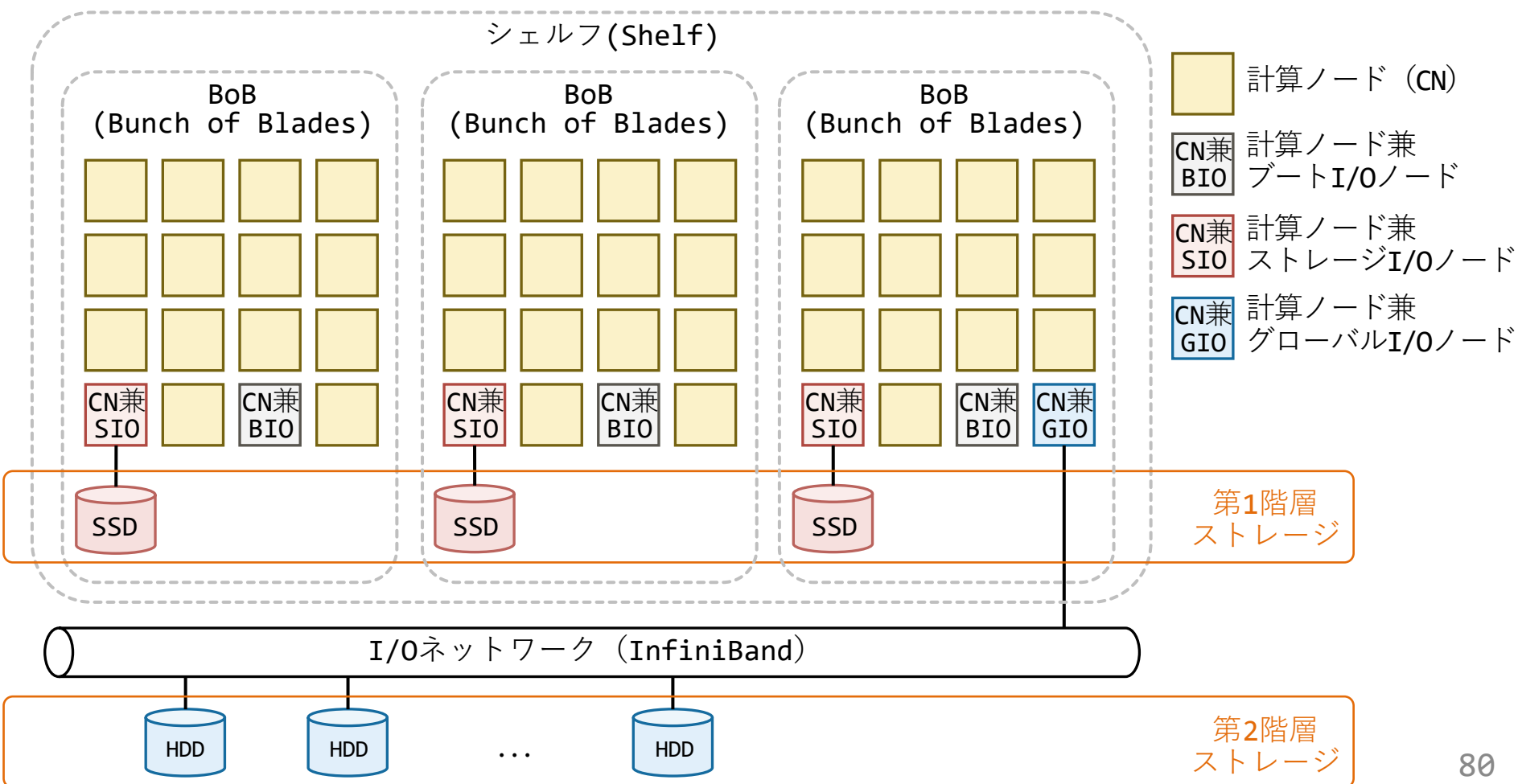
第2階層ストレージは富士通が開発した並列分散ファイルシステムであるFEFS（**F**ujitsu **E**xabyte **F**ile **S**ystem）により管理されるHDDストレージである。

第2階層ストレージはログインノードおよび各計算ノードで共有され、ジョブの入出力データを保管するためのストレージとして利用される。

■ 「富岳」ストレージ構成の各要素の説明 [1/7]

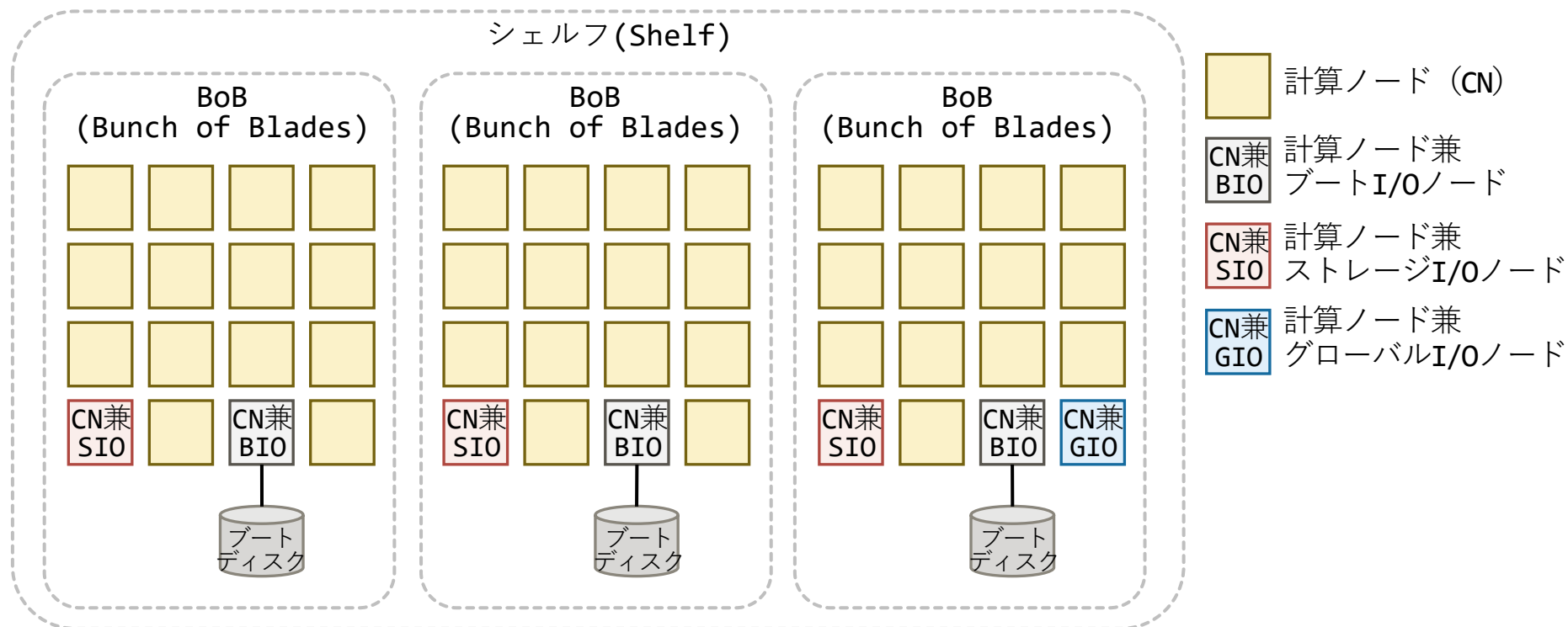
「富岳」ストレージ構成の概要を図に示す。以降の数ページに渡って図の各要素について説明する。

- 計算ノード同士は、概略の章で記述したように、TofuインターコネクタDで接続されている（図ではTofuインターコネクタDによる接続は省略されている）。



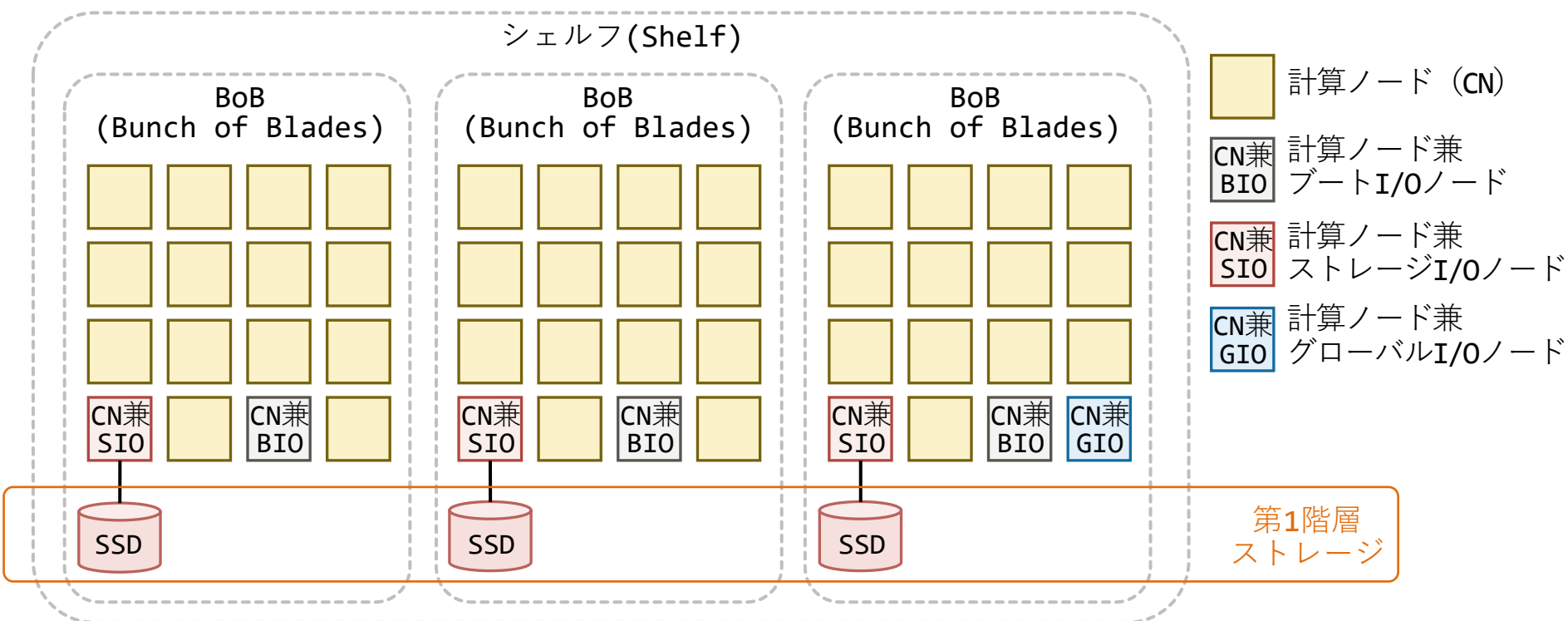
■ 「富岳」ストレージ構成の各要素の説明 [2/7]

- 物理的には、一つのBoB(Bunch of Blades)に16個の計算ノードがグループ化されて存在する。グループ内の一つの計算ノードに、16個の計算ノードのネットワークブートを担うブートサーバが稼働し、ブートディスクが接続されている。この役割を受け持つノードを、計算ノード兼ブートI/O (BIO) ノード、あるいは単にBIOノード、という。(以降のスライドの図ではBIOノードにつながるブートディスクは説明の便宜上省略する。)



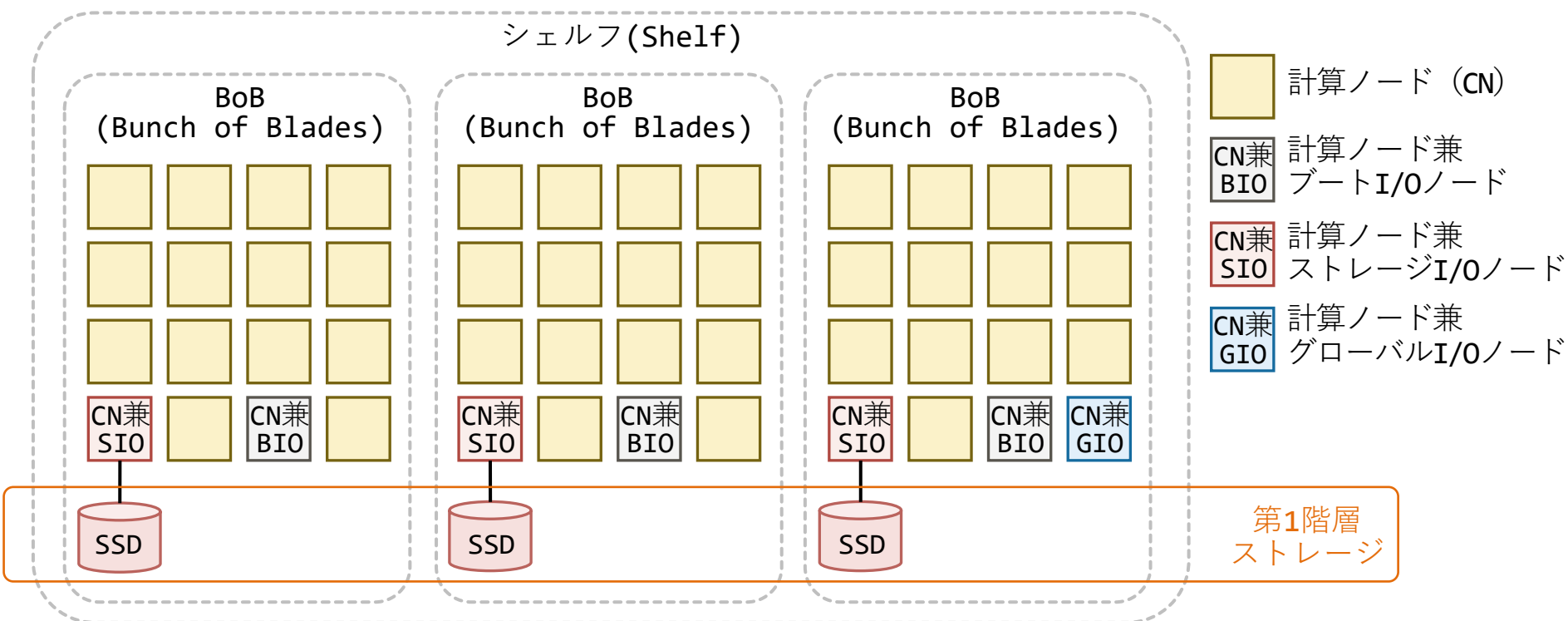
■ 「富岳」ストレージ構成の各要素の説明 [3/7]

- 各BoB内に第1階層ストレージを構成する一つのSSDが存在する。このSSDはそのBoB内の一つのノードに接続し、そのノード上でSSDストレージのサーバが稼働する。このサーバが稼働するノードを、計算ノード兼ストレージI/O(SIO)ノード、あるいは単にSIOノード、という。
- SIOノードは一つのBoBごと（16ノードごと）に一つ存在する。



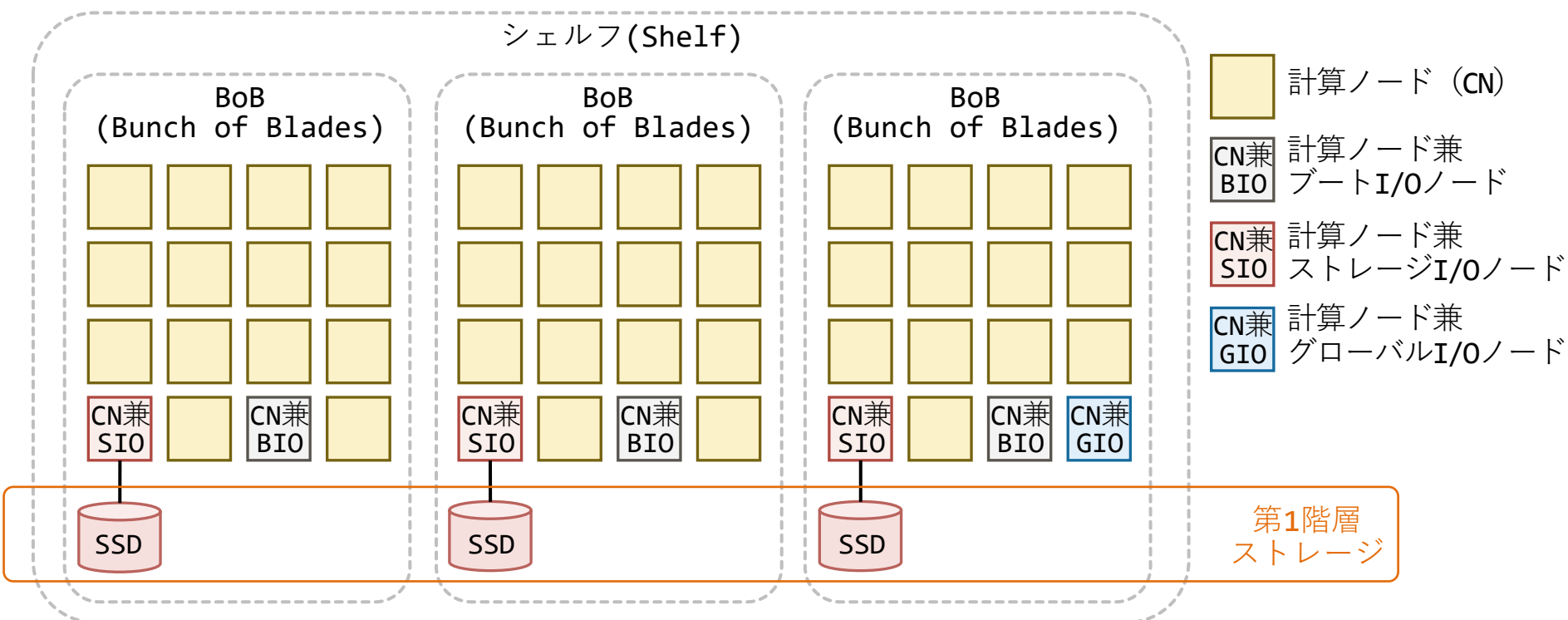
■ 「富岳」ストレージ構成の各要素の説明 [4/7]

- ユーザのジョブが開始すると、ジョブに割り当てられた計算ノードが属するBoBのSIOノードのみを用いて、一時的なLLIOのファイルシステムが生成される。ジョブに割り当てられた計算ノードは、そのジョブに割り当てられたBoB内のSIOノードのみを経由し、生成されたLLIOのファイルシステムにアクセスする。ジョブが完了すると、このファイルシステムは解放される。



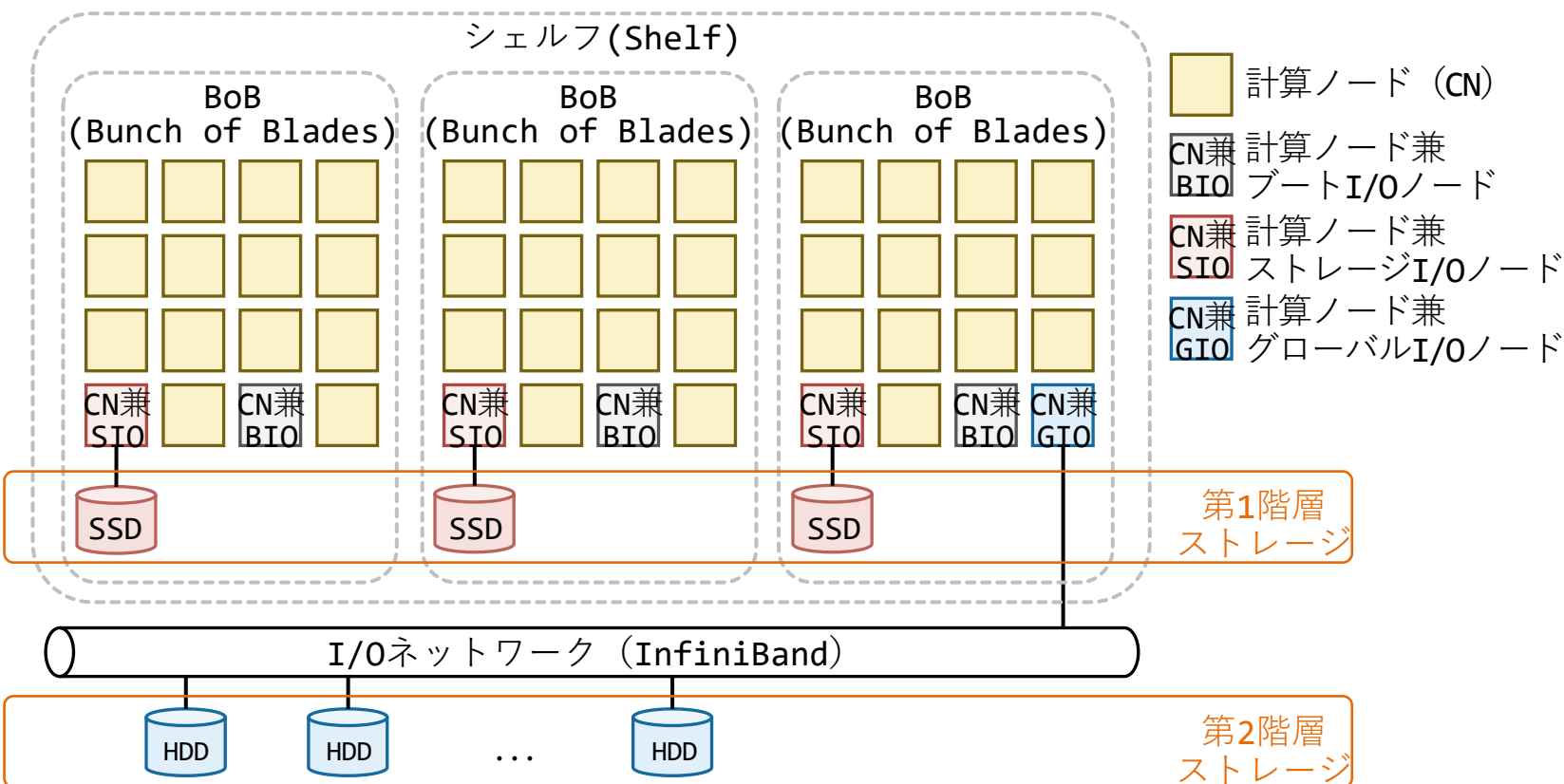
■ 「富岳」ストレージ構成の各要素の説明 [5/7]

- 各計算ノードからのファイルアクセス要求はSIOノードのアシスタントコアで処理される。
- (SIOノード内でのユーザジョブの計算は48個の計算コアで処理され、ファイルアクセス要求がユーザジョブの計算の妨げとならない仕組みとなっている。)



■ 「富岳」ストレージ構成の各要素の説明 [6/7]

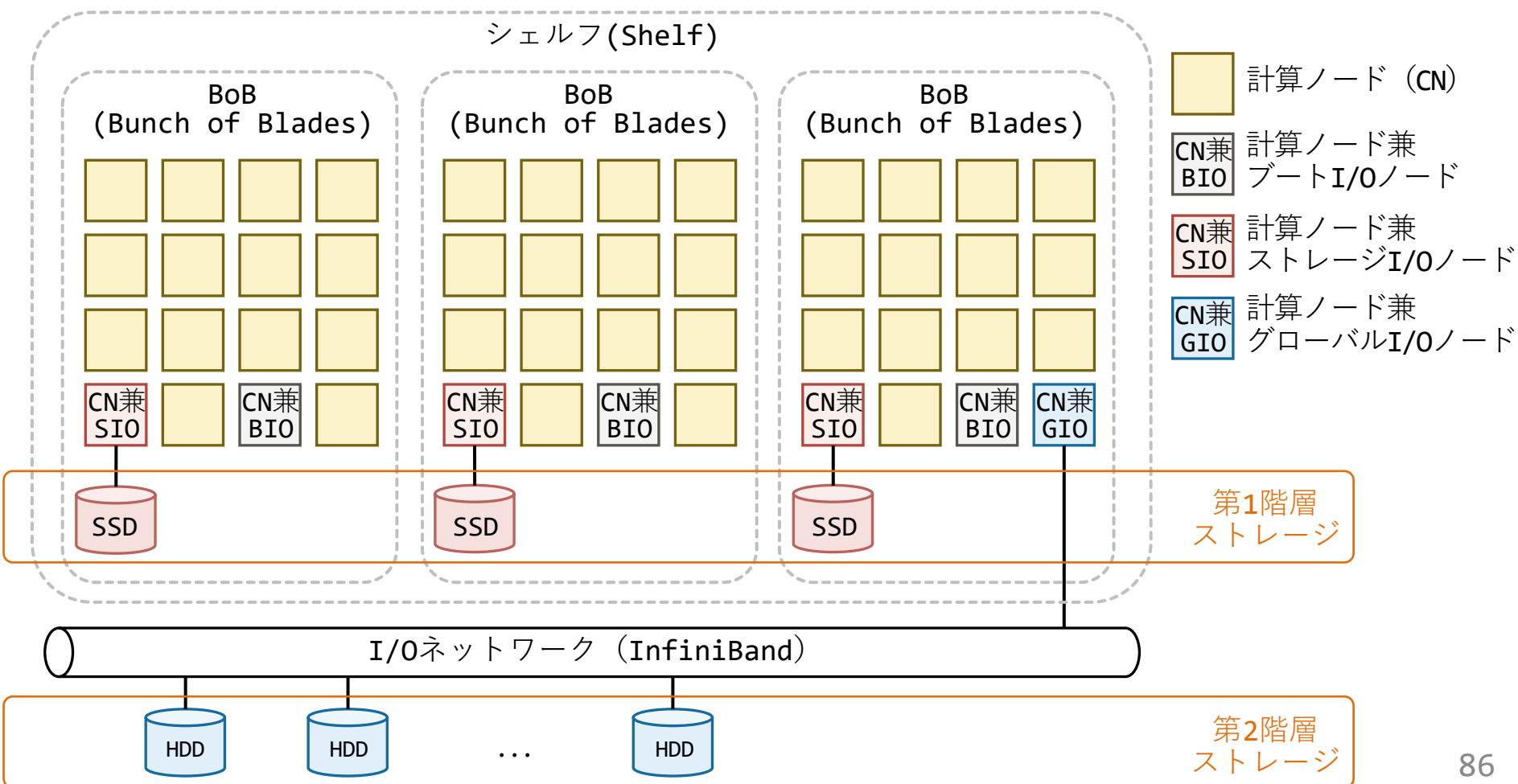
- 三つのBoBで一つのシェルフを構成する。（一つのシェルフには48個のノードが存在する。）
- 各シェルフ内の1個または0個のノードがInfiniBandを使用したI/Oネットワークにつながる。
- I/Oネットワークにはまた、第2階層ストレージを構成するHDDが接続されている。
- シェルフ内のノードのうち、I/Oネットワークに（従って第2階層ストレージに）つながっているノードを、計算ノード兼グローバルI/O(GIO)ノード、あるいは単にGIOノードという。
- GIOノードは192ノード当たり1個存在する。
- 第1階層と第2階層ストレージ間のデータ転送はSIOノードとGIOノードを経由して行われる。



■ 「富岳」ストレージ構成の各要素の説明 [7/7]

- なお、後の【参考情報】のスライドで示されるように、BIO、SIO、GIOノードが同一ノードとなることはない。

以上で図の説明を終わる。



■ 【参考】 ノードがBIO、SIO、またはGIOノードであるかの確認方法 [1/3]

- ノードがBIO、SIO、またはGIOノードであるかはノードIDから以下のようにして確認することができる。

- ノードIDは「`pjsub -S`」により、ジョブ終了後に生成されるジョブ統計情報に16進数で出力される。

- ノードがBIO、またはSIOノードであるかは16進数のノードIDの末尾により以下のように判定される。

- 末尾が01はBIOノードである。
- 末尾が02はSIOノードである。

- GIOノードのリストは以下のURLで示されている：

`https://www.fugaku.r-ccs.riken.jp/doc_root/ja/contents/0603-2020/GIO_LIST.txt`

上記URLのファイルGIO_LIST.txt中の16進数ノードIDの末尾はどれも03である。

■ 【参考】 ノードがBIO、SIO、またはGIOノードであるかの確認方法 [2/3]

- ノードIDから、ノードがBIO、SIO、またはGIOノードのいずれであるかを確認する
idcheckコマンドがログインノードで利用できる。参考URLを以下に示す。

https://www.fugaku.r-ccs.riken.jp/faq/20200604_01

idcheckコマンドの使用例を以下に示す。

【使用例1】 -h オプションでヘルプ内容が表示される。

```
login7$ idcheck -h
Usage: idcheck [OPTION]...
  -h Display help
  -n nodeid[,...] Search node type from node ID
  -f <FILE> This option executes the process for node IDs written in the file that
is specified by filename.
[Example]
0x01010001
0x01010002
0x01010003
0x01010004
```

次スライドへ次の使用例が続く。

■ 【参考】 ノードがBIO、SIO、またはGIOノードであるかの確認方法 [3/3]

【使用例2】 -n オプションで、コンマ区切りで複数のNODEIDを指定して確認できる。

```
login7$ idcheck -n 0x01010001,0x01010002,0x01010003,0x01010004
0x01010001 BIO
0x01010002 SIO
0x01010003 GIO
0x01010004 CN
```

【使用例3】 -f オプションでファイルに記載されたNODEID群を確認できる。

- aaa という名前のファイルには16進数のノードIDが一行につき一つ記述されている。

```
login7$ cat aaa
0x01010001
0x01010002
0x01010003
0x01010004
```

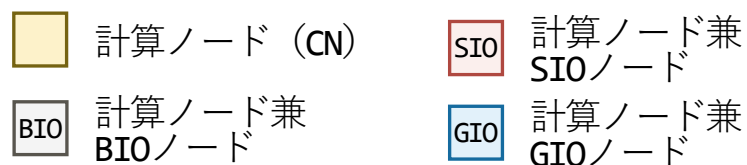
- idcheck コマンドの -f オプションにファイルaaaを指定して実行すると、各ノードIDの種別が表示される。

```
login7$ idcheck -f aaa
0x01010001 BIO
0x01010002 SIO
0x01010003 GIO
0x01010004 CN
```

■ Tofu単位との関係

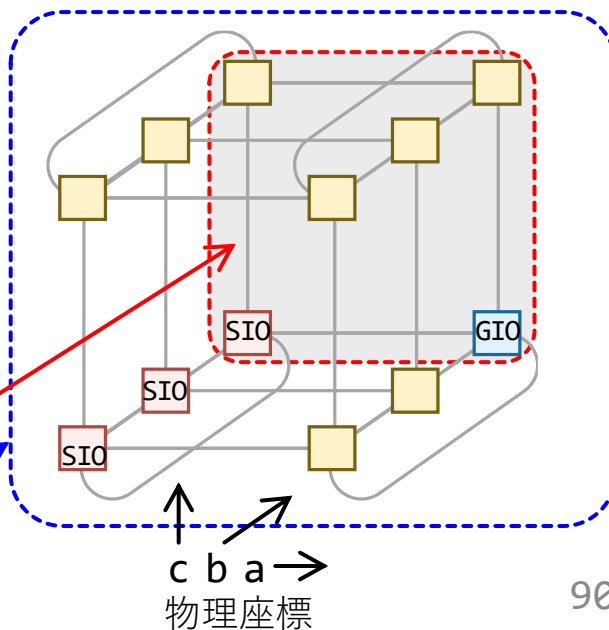
シェルフ内のノードがどのように集まりTofu単位を構成しているかについて記す。

- 図の青い点線に示すように、一つのシェルフ内の三つのBoBそれぞれから四つずつのノードが集まり一つのTofu単位を構成する。
 - 三つのBoBからの合計3個のSIOノードと1個または0個(*1)のGIOノードは一つのTofu単位に入る。
(*1) GIOノードは192ノード当たりについて存在し、GIOノードを含まないシェルフと一つ含むシェルフが存在する。
- 図の赤い点線に示すように、一つのBoBからの四つのノードは同一のac面内に配置される。
- 三つのSIOノードはTofu単位内で、同一のac座標で、b軸に沿う $b=0, 1, 2$ に、配置される。
- 同様に、一つのシェルフ内の各BoBのGIOノードはSIOノードが入るTofu単位とは別の一つのTofu単位に入る。
- 一つのシェルフ内の48個のノードが合計4個のTofu単位を構成する。



シェルフ

SIOノードを3個、GIOノードを1個含むTofu単位の例



内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
 - 「富岳」ストレージ構成の概要
 - LLIOの機能概略
 - 第1階層ストレージの三つの領域
 - 第1階層ストレージのサイズ
 - 共通ファイル配布機能
 - 【参考】LLIO使用サンプル
- 「富岳」のMPI通信
- 【付録】参考文献

第1階層ストレージはLLIOによって管理される領域である。LLIOは以下の機能を提供することにより、第1階層ストレージに対する高性能な入出力を可能にする。

ユーザはジョブ投入時にコマンドの--llioオプションでLLIOに関するパラメータを指定することで、これらの機能の使用の有無または動作を指定することができる。

■ 第1階層ストレージの三つの領域（後の小節で詳細を説明します）

- 第1階層ストレージは、計算ノードからの参照範囲やファイルの生存期間等に関して異なる特性を持つ3種類の領域を持つ。
 - ノード内テンポラリ領域
 - 共有テンポラリ領域
 - 第2階層ストレージのキャッシュ領域
- ユーザは自分のジョブの特性に応じてこれらの領域を使い分けることができる。

■ 第1階層ストレージのサイズ（後の小節で詳細を説明します）

- ユーザはジョブ投入時にそれぞれの領域に対して最適な大きさを指定できる。

```
$ pjsub --llio localtmp-size=size
```

```
$ pjsub --llio sharedtmp-size=size
```

■ （次スライドへ続く）

■ 共通ファイル配布機能：llio_transferコマンド（後の小節で詳細を説明します）

- ジョブの実行開始時、第2階層ストレージ上の読み込み専用ファイル（実行プログラムや入力ファイル）は負荷の集中が想定される。
- 共通ファイル配布機能は、第2階層ストレージ上のこれら入力ファイルのコピーを第1階層ストレージの複数のSSDに作成する機能である。
- 各計算ノードは第1階層ストレージに作成された複数のコピーに分散してアクセスすることで負荷分散を実現することができる。

■ （次スライドへ続く）

■ 計算ノード内キャッシュ機能

- 本機能はアプリケーションが使用していない計算ノードの空きメモリをページキャッシュとして使用する機能である。本機能を利用することで、ジョブからのI/Oを計算ノード内で折り返し、I/Oの高速化を実現することができる。
- ユーザは計算ノード内キャッシュの動作について、以下を指定することができる。

■ 計算ノード内キャッシュのメモリサイズ

```
$ pjsub --llio cn-cache-size=size
```

■ read時における計算ノード内キャッシュへのキャッシュ機能

```
$ pjsub --llio cn-read-cache={on|off}
```

■ 計算ノード内キャッシュへのファイルの自動先読み機能

```
$ pjsub --llio auto-readahead={on|off}
```

- ⚠ パラメータcn-read-cache=offを指定（計算ノードへの読み込み時に計算ノード内キャッシュにキャッシュしないと指定）している場合、自動先読みは無効になる。auto-readaheadを有効（on）にするにはcn-read-cacheを有効（on）にする必要がある。

■ 計算ノード内キャッシュの使用有無を切り替えるwriteサイズの閾値

```
$ pjsub --llio cn-cached-write-size=size
```

■ （次スライドへ続く）

■ 読み込み時における第1階層ストレージへのキャッシュ機能

- 本機能は第2階層ストレージから計算ノード内キャッシュに読み込んだファイルを第1階層ストレージにキャッシュする機能である。本機能を使用することで、ジョブが同じファイルを複数回読み込む場合に性能向上が実現できる。

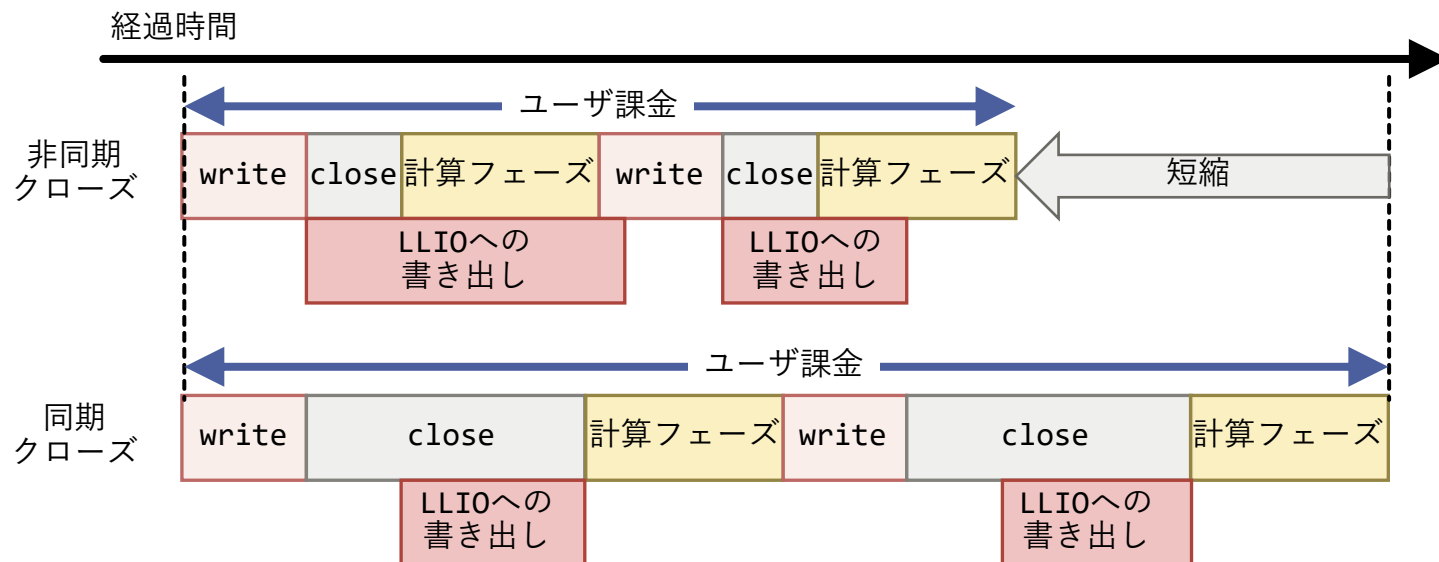
```
$ pjsub --llio sio-read-cache={on|off}
```

■ （次スライドへ続く）

「富岳」のストレージ構成とLLIO・LLIOの機能概略

■ 非同期クローズ機能

- 本機能は、計算ノード内キャッシュから第1階層ストレージ、および第2階層ストレージのキャッシュから第2階層ストレージへの書き出しにおいて、ファイルクローズ時の書き出しを非同期に行う機能である。この機能を使用することで、書き出し完了を待たされることがなくなり、ジョブの終了が早くなる効果がある。



- この機能が有効でも、ジョブ終了までのファイルの書き出しは保証される。ただし、計算ノードがダウンした場合やジョブの経過時間制限を超過した場合、キャッシュから第2階層ストレージへの転送は中断され、キャッシュから第2階層ストレージへの書き出しは保証されない。
- ユーザはジョブ投入時に、非同期クローズ機能を使用するかどうかを指定できる。

```
$ pjsub --llio async-close={on|off}
```


■ 第1階層ストレージへのストライプ機能

- ストライプ（またはストライピング）とは、データを均等なサイズのブロック単位に分割し、複数の**SSD**または複数の**HDD**に分散してラウンドロビンで読み書きすることである。
 - 一つのブロックをストライプといい、一つのブロックのサイズをストライプサイズ、分散する個数をストライプカウント、ストライプの総数をストライプ数という。
- **LLIO**は、一つのファイルを各ストレージ**I/O**ノードにつながる**SSD**にストライピングする、第1階層ストレージへのストライプ機能を提供する。本機能には以下の効果がある。
 - 一つの**SSD**の物理的な容量を超えるサイズのファイルを作成できる。
 - 一つのファイルを複数の**SSD**に分散して格納することで、ファイルアクセスの帯域幅が向上する。
- ⚠ 第1階層ストレージへのストライプ機能が使用できるのは、共有テンポラリ領域と第2階層ストレージのキャッシュ領域である。ノード内テンポラリ領域には使用できない。
- ユーザはジョブ投入時にストライプカウントとストライプサイズを指定することができる。

```
$ pjsub --llio stripe-count=count,stripe-size=size
```

■ LLIOの利用をサポートする機能の概略 [1/2]

LLIOの利用をサポートするための機能として、以下の機能が提供される。これらの機能の使用はジョブ投入時に指定できる。

■ LLIO性能情報の採取機能

- 本機能は第1階層ストレージへのアクセス状況に関する情報（LLIO性能情報）を採取しジョブ終了時にファイルとして出力する機能である。
- LLIO性能情報を参照することで、自分のジョブに関する第1階層ストレージへのアクセス状況を後から分析することができる。
- ユーザはジョブ投入時にpjsubコマンドの--llio オプションのパラメータperfを指定することで、LLIO性能情報ファイルを出力することができる。

```
$ pjsub --llio perf
```

⚠ LLIO性能情報の採取に関する注意事項

- LLIO性能情報の採取はジョブがRUNOUT状態の時に実施している。
- LLIO性能情報は、ジョブを実行したノード数が多くなるほどその採取に時間がかかり、これにともなってジョブがRUNOUT状態となっている時間も増えていく。
- 1万ノードを超えるジョブにおいては、ジョブ投入時のオプション(--llio perf)を指定しないでください。ジョブスクリプトで指定している場合は、削除もしくはコメントアウトしてください。1万ノードを超えるジョブでオプション(--llio perf)を指定した場合、ジョブ運用全体に影響を与える可能性があります。

https://www.fugaku.r-ccs.riken.jp/restriction/20220304_01

■ LLIOの利用をサポートする機能の概略 [2/2]

■ 未書き出しファイル一覧取得機能

- 本機能は、ジョブ実行中に計算ノードに異常が発生した場合や、ジョブの経過時間制限に達した場合などで、計算ノード内キャッシュから第1階層ストレージへ、および第1階層ストレージから第2階層ストレージへの書き出しが完了しなかったファイルの一覧を出力する機能である。
- 未書き出しファイルの情報を分析することで、ジョブの経過時間制限値を調整するための目安にできる。
- ユーザはジョブ投入時にpjsubコマンドの--llio オプションのパラメータ **uncompleted-fileinfo-path**を以下のように指定することで、未書き出しファイルの情報を *path*で指定されたファイルに出力することができる。

```
$ pjsub --llio uncompleted-fileinfo-path=path
```

- 本パラメータを指定しなかった場合、未書き出しファイルの情報は標準エラー出力に出力される。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
 - 「富岳」ストレージ構成の概要
 - LLIOの機能概略
 - 第1階層ストレージの三つの領域
 - 第1階層ストレージのサイズ
 - 共通ファイル配布機能
 - 【参考】LLIO使用サンプル
- 「富岳」のMPI通信
- 【付録】参考文献

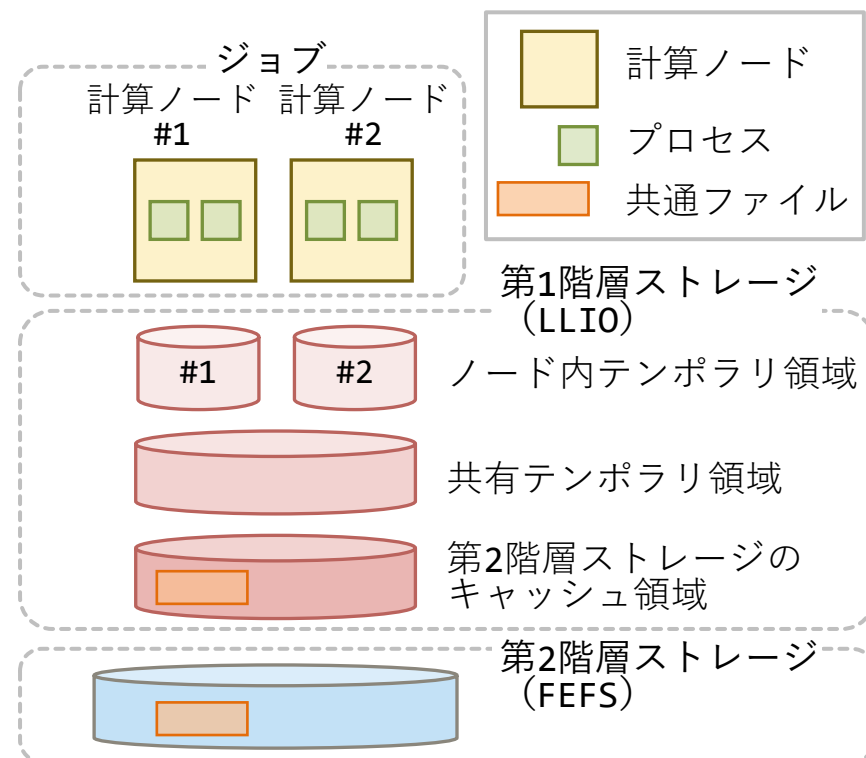
「富岳」のストレージ構成とLLIO▶第1階層ストレージの三つの領域

■ 三つの領域の概要

■ LLIOは第1階層ストレージに次の三つの領域を提供する。

- ・ ノード内テンポラリ領域
- ・ 共有テンポラリ領域
- ・ 第2階層ストレージのキャッシュ領域

■ 上記三つの領域の特徴は以下の通りである。

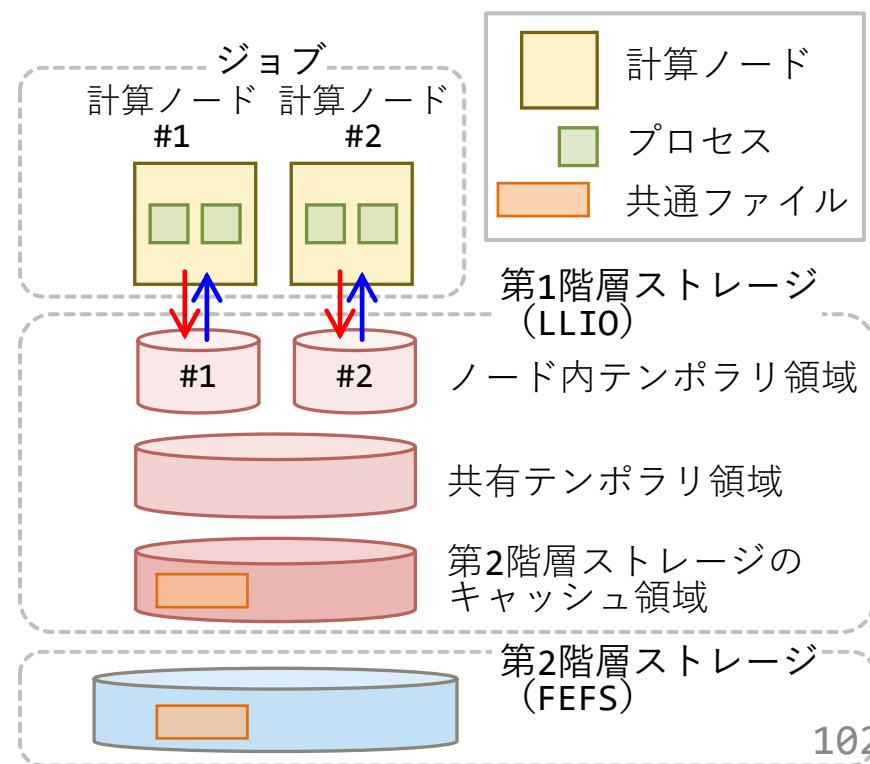


項番	領域名	読み書きできる範囲	適用事例
1	ノード内テンポラリ領域	同一ジョブの同一ノード内	ランク・ノードごとに独立した中間ファイルや一時ファイルの作成・参照。
2	共有テンポラリ領域	同一ジョブの全ての計算ノード	ランク・ノード間でのファイルを参照、巨大なファイルの操作。
3	第2階層ストレージのキャッシュ領域	同一ジョブの全ての計算ノード	全ての計算ノードに共通な実行プログラムや入力ファイルの読み込み。標準出力・標準エラー出力など、第2階層に保存されるファイルの出力。

■ 一般に、項番の小さいほど、より良いIO性能スケールを持つので、この順に利用を検討すると良い。

■ ノード内テンポラリ領域とその利用方法の概略

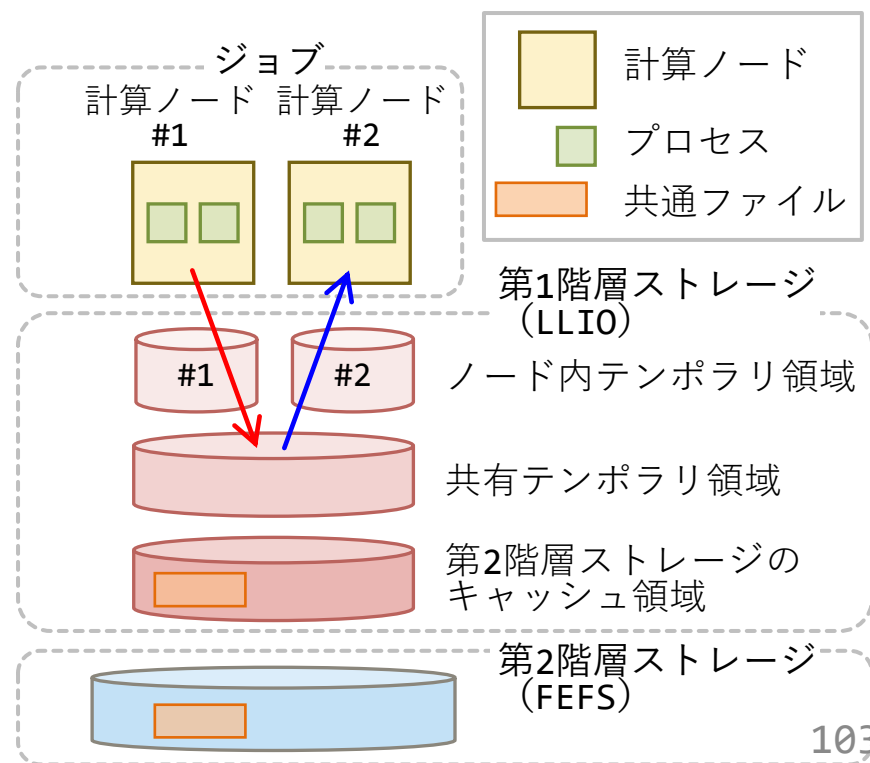
- ノード内テンポラリ領域は同ジョブにおいて同一計算ノード内で動作するプロセスからのみ参照可能な一時領域である。各計算ノードに対して一つのノード内テンポラリ領域が割り当てられ、一つの計算ノードに割り当てられたノード内テンポラリ領域はジョブ内の他の計算ノードからは利用できない。
 - 本領域を使用するには、ユーザがジョブ投入時にpjsubコマンドのオプション**--llio**
localtmp-size=sizeで割り当てたい領域サイズを**size**に指定する必要がある（詳細は後述）。
 - ノード内テンポラリ領域は、ジョブ起動前に初期化され、ジョブ終了時に削除される。
 - ノード内テンポラリ領域のパス名はジョブ内で環境変数`${PJM_LOCALTMP}(*1)`により参照することができる。
- (*1) 本環境変数は計算ノードでのみ利用できる。ログインノードでは空文字列である。
- ジョブから本領域へ書き出した実行結果などの出力ファイルは、第2階層ストレージへは自動的に書き出されない。そのため、ユーザがジョブスクリプト内で手動で第2階層ストレージへコピーする必要がある。



※ 手動でコピーする例を後の節「【参考】LLIO使用サンプル」で示します。

■ 共有テンポラリ領域とその利用方法の概略

- 共有テンポラリ領域は同一ジョブに割り当てられた全ての計算ノードから参照できる一時領域である。ジョブ内の全ての計算ノードにわたって共通な一つの共有テンポラリ領域が割り当てられ、ジョブ内の一つの計算ノードがこの領域に書き出したファイルをそのジョブ内の他の計算ノードが読み込むことができる。また、ジョブ内の複数の計算ノードが共有テンポラリ領域内に共通のファイルを読み書きできる。
- 本領域を使用するには、ユーザがジョブ投入時に **pjsub** コマンドのオプション **--llio** **sharedtmp-size=size** で割り当てたい1ノード当たりの領域サイズを **size** に指定する必要がある（詳細は後述）。共有テンポラリ領域のサイズは **size** × 割り当てノード数となる。
- 共有テンポラリ領域は、ジョブ起動前に初期化され、ジョブ終了時に削除される。
- 共有テンポラリ領域のパス名はジョブ内で環境変数 **\${PJM_SHARED TMP}(*2)** により参照することができる。
(*2) 本環境変数は計算ノードでのみ利用できる。ログインノードでは空文字列である。
- ジョブから本領域へ書き出した実行結果などの出力ファイルは、第2階層ストレージへは自動的に書き出されない。そのため、ユーザがジョブスクリプト内で手動で第2階層ストレージへコピーする必要がある。



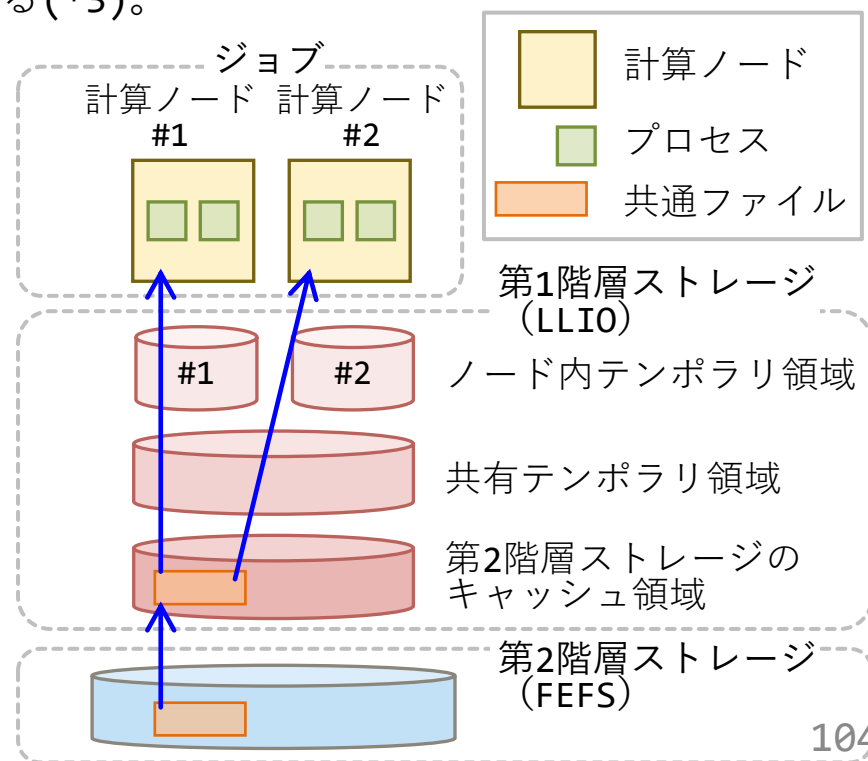
「富岳」のストレージ構成とLLIO▶第1階層ストレージの三つの領域

■ 第2階層ストレージのキャッシュ領域とその利用方法の概略 [1/3]

- 第2階層ストレージのキャッシュ領域とは、第1階層ストレージ上に第2階層ストレージがキャッシュされた領域である。
- ユーザはジョブ投入時に、共有テンポラリ領域の大きさとノード内テンポラリ領域の大きさを指定することで第2階層ストレージのキャッシュ領域の大きさを間接的に指定できる（詳細は後述）。
- ユーザは、本領域を第2階層ストレージと同じように扱うことができる。ジョブ（計算ノード）から第2階層ストレージ（パス名：`/vol0x0y/data/<groupname>`）へのI/Oは、第2階層ストレージのキャッシュ領域を経由してアクセスされる(*3)。

(*3) 本キャッシュ領域を介さずに第2階層ストレージに直接アクセスしたい場合は、`2ndfs`領域（パス名：`/2ndfs/<groupname>`）を用いることができる。

- 第2階層ストレージのキャッシュ領域は同一ジョブに割り当てられた全ての計算ノードから参照できる。
- （次スライドへ続く）



■ 第2階層ストレージのキャッシュ領域とその利用方法の概略 [2/3]

- コマンド `llio_transfer` を利用した 共通ファイル配布機能（詳細は後述）により、アクセスが集中するファイルの複数のコピーを、第2階層ストレージのキャッシュ領域に共通ファイルとして配布することができる。

```
#!/bin/bash
```

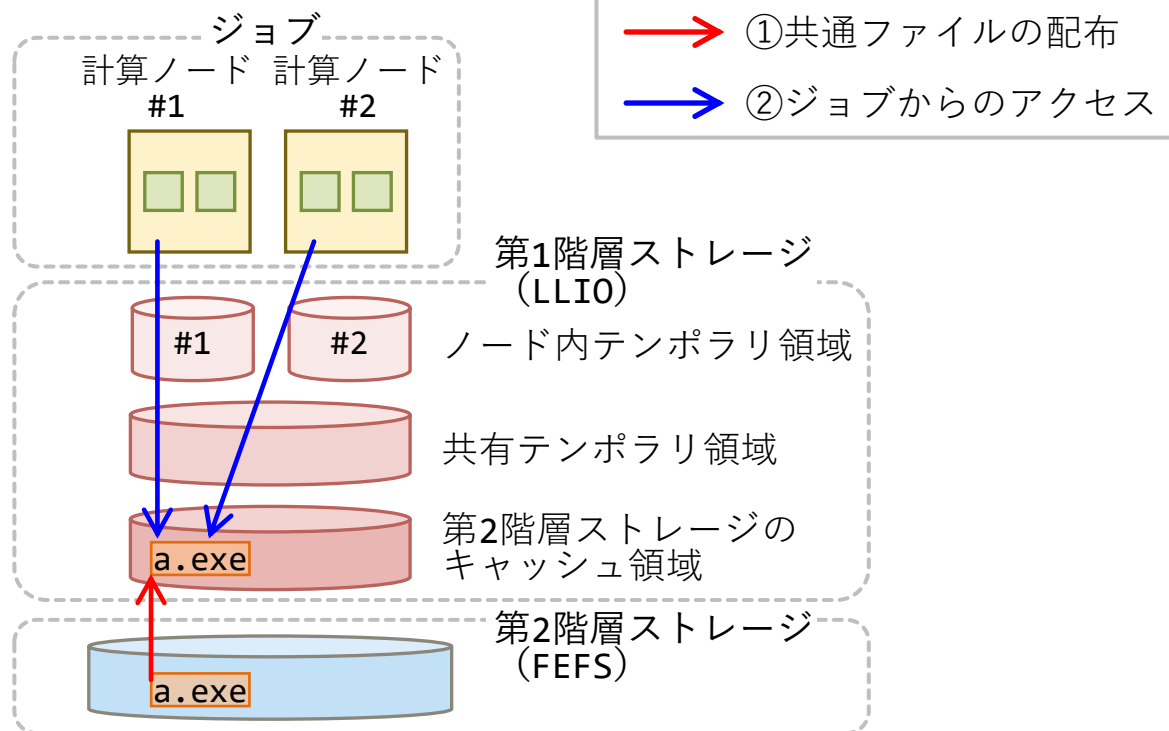
```
...
```

```
llio_transfer ./a.exe
```

①

```
mpiexec ./a.exe
```

②



■ 第2階層ストレージのキャッシュ領域とその利用方法の概略 [3/3]

■ 生存期間

- 第2階層ストレージのキャッシュ領域は、ジョブ起動前に初期化され、ジョブ終了時に削除される。
- 第2階層ストレージへの書き出し途中でジョブ実行可能時間制限を超えるなどしてジョブが中断された場合は、第2階層ストレージのキャッシュ領域は書き出しが完了していない場合(*4)でも削除される。
(*4) ジョブ投入時にpjsubコマンドのオプション--llio_uncompleted-fileinfo-path=*path*を指定すると、書き出されなかったファイルの一覧を*path*に指定したファイルに出力することができる。
- 上記以外で第2階層ストレージのキャッシュ領域のキャッシュが削除されるタイミングは以下のとおりである。
 - ファイルを削除した場合。
 - 第2階層ストレージのキャッシュ領域が一杯になった場合（この場合は LRU 方式で古いものから削除される）。
 - Direct I/O (write または read) が実行された場合（この場合は Direct I/O した箇所のキャッシュが削除される）。

※ pjdelコマンドでジョブを削除する場合、pjdelのオプション--llio-flushによって、経過時間制限値の範囲内で、ファイルへの書き出し完了を待ち合わせることができる。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
 - 「富岳」ストレージ構成の概要
 - LLIOの機能概略
 - 第1階層ストレージの三つの領域
 - 第1階層ストレージのサイズ
 - 共通ファイル配布機能
 - 【参考】LLIO使用サンプル
- 「富岳」のMPI通信
- 【付録】参考文献

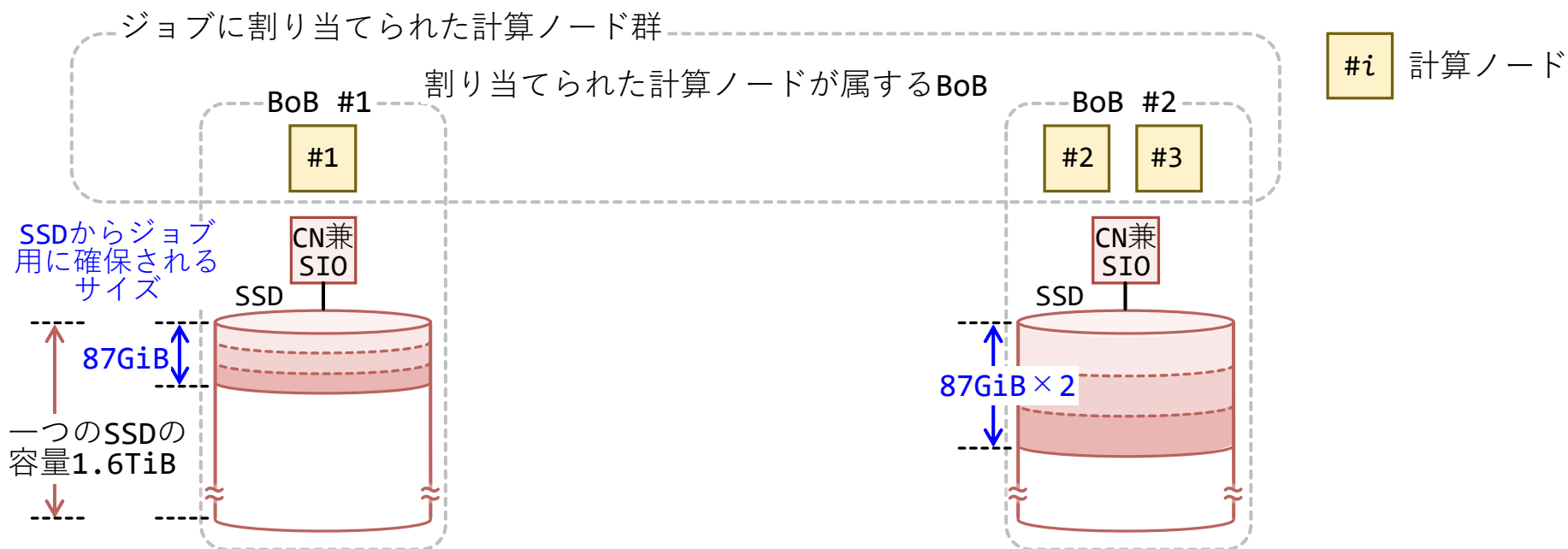
「富岳」のストレージ構成とLLIO ▶ 第1階層ストレージのサイズ

■ 第1階層ストレージの領域の確保 [1/3]

- 第1階層ストレージの領域はストレージI/O (SIO) ノードに接続されたSSDから確保され、ジョブに割り当てられたノード一つ当たり**87GiB**が確保される。

より詳しくは、ジョブに割り当てられた**各ノードについて**、そのノードが属するBoB内のSIOノードに接続されたSSDから**87GiBが確保される**。

- (次スライドに説明が続く)

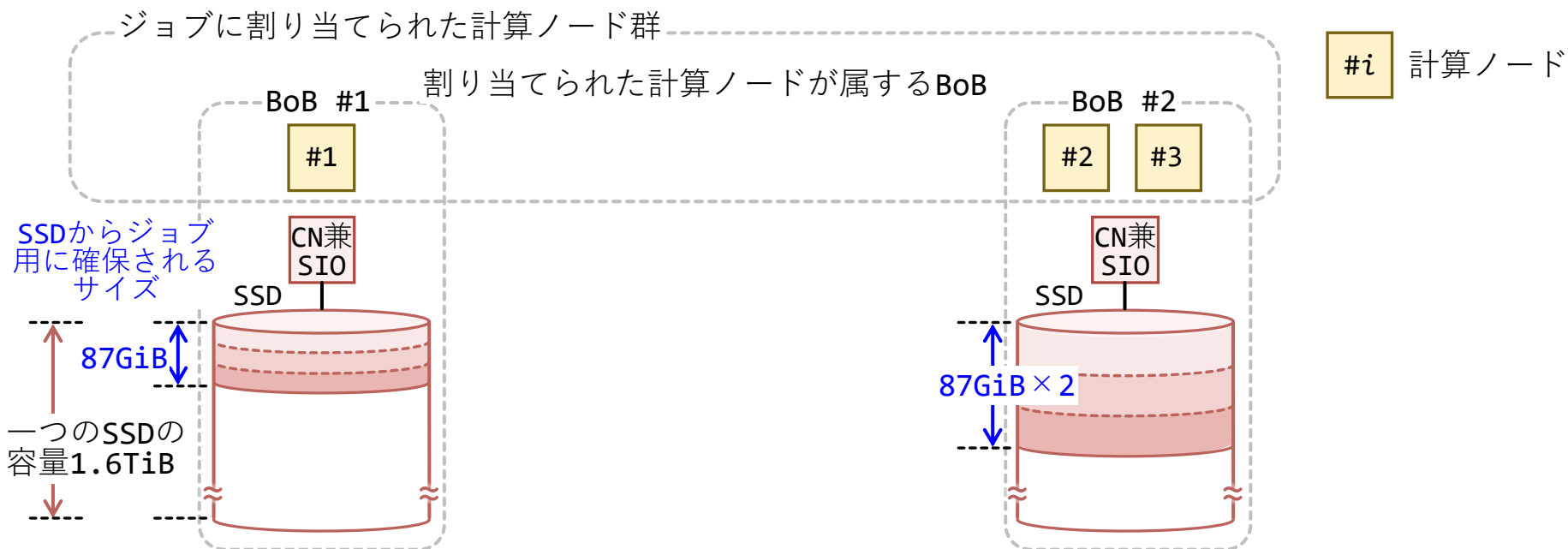


■ 第1階層ストレージの領域の確保 [2/3]

- 例えばジョブスクリプトで「`#PJM -L node=3`」の指定により割り当てノード数が3とする。さらに、下図に示すように、一つのノードはあるBoB（BoB #1とする）に属し、残り二つのノードはもう一つのBoB（BoB #2とする）に属するとする(*1)。

(*1) ジョブにいくつかのSIOノードが確保されたかは、LLIO性能情報で出力されるファイル（ジョブ名.ジョブ番号.llio_perf）中の「SIO Infomation」から確認できる。

三つのノードが下図のように割り当てられた場合、BoB #1のSIOノードに接続されたSSDから1ノード分の 87 GiB が確保され、BoB #2のSIOノードに接続されたSSDから2ノード分の 87×2 GiB が確保される。

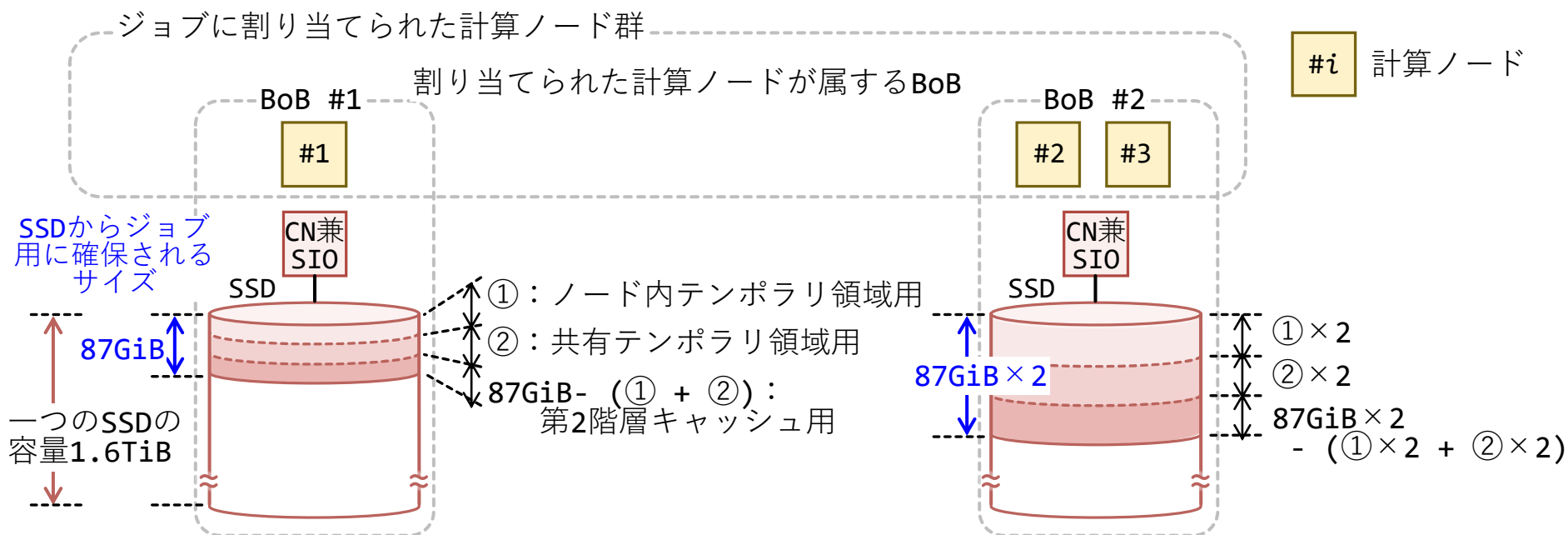


■ （次スライドに説明が続く）

■ 第1階層ストレージの領域の確保 [3/3]

- 確保された1ノード当たり **87 GiB**について、ノード内テンポラリ領域用と共有テンポラリ領域用のサイズ（下図の①と②のサイズ）を、ユーザはジョブ実行時に指定する。

その指定のためのオプションを次スライドで示す。



■ 第1階層ストレージのサイズを指定するオプション

- 第2階層ストレージのキャッシュ領域の容量は1計算ノードあたり最低**128MiB**必要である。
- 1計算ノードあたり「**87GiB - 128MiB**」に対して、ノード内テンポラリ領域と共有テンポラリ領域のサイズを以下の二つのパラメータで指定する。どちらも1ノード当たりのサイズを指定する。

■ `pjsub --llio localtmp-size=size_L` ①

「1ノード当たりのノード内テンポラリ領域のサイズ」を`size_L`に指定する。

■ `pjsub --llio sharedtmp-size=size_S` ②

「ジョブ全体の共有テンポラリ領域のサイズ÷割当てノード数」を`size_S`に指定する。

- ジョブスクリプト内で指定する場合は、以下のように指定する。

```
#PJM --llio localtmp-size=size_L
#PJM --llio sharedtmp-size=size_S
```

- 各領域のサイズに指定できる値の下限、上限、およびデフォルト値は、`pjac1`コマンドで表示される項目`localtmp-size`と`sharedtmp-size`で確認できる。ただし、①と②の値の合計は、最大 **87GiB - 128MiB** まで指定できる (① + ② ≤ **87GiB - 128MiB**) 。

```
$ pjac1 --rg small
(中略)
pjsub option parameters
(中略)
      (--llio)                lower      upper      default
      (sharedtmp-size=)      0Mi        89278Mi    0Mi
      (localtmp-size=)       0Mi        89278Mi    0Mi
```

- 上記二つのパラメータより、第2階層ストレージのキャッシュ領域のサイズは1ノードあたり **87GiB - (①の値 + ②の値)** と定まる。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
 - 「富岳」ストレージ構成の概要
 - LLIOの機能概略
 - 第1階層ストレージの三つの領域
 - 第1階層ストレージのサイズ
 - 共通ファイル配布機能
 - 【参考】LLIO使用サンプル
- 「富岳」のMPI通信
- 【付録】参考文献

■ 共通ファイル配布機能を使用しない場合

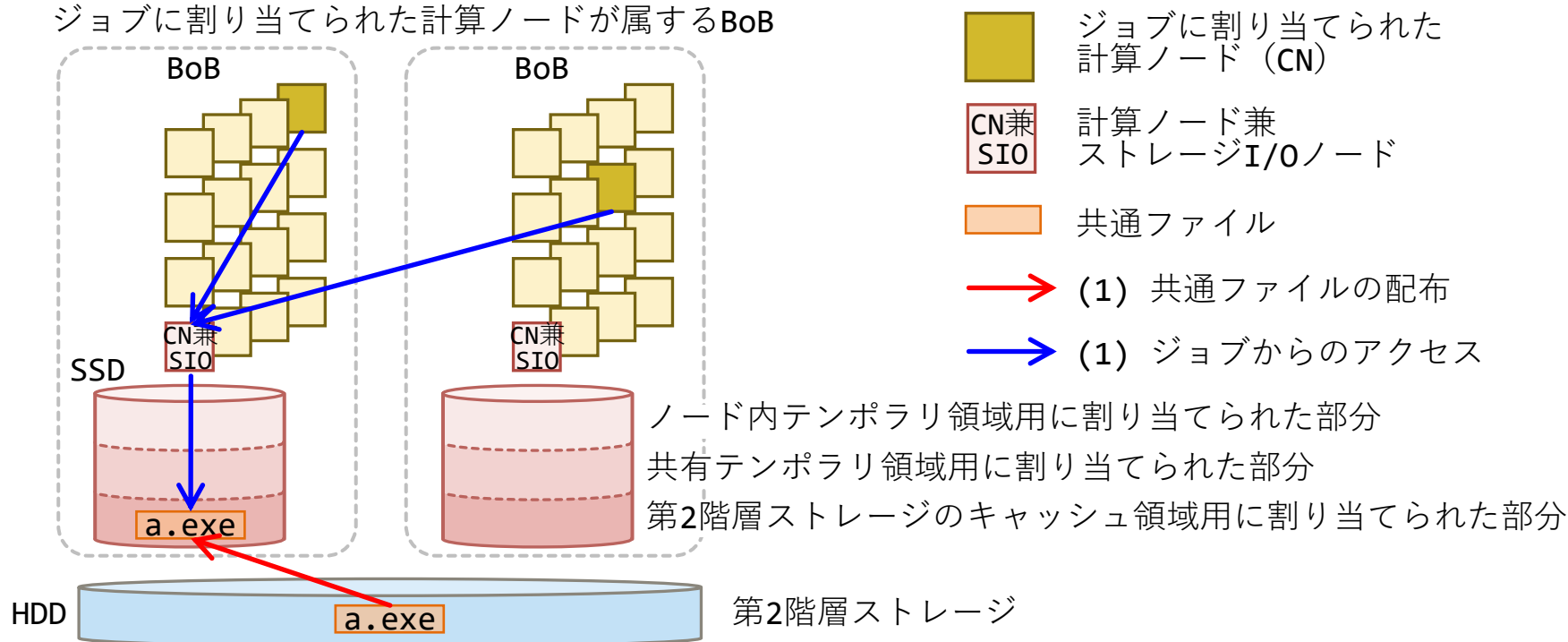
- ジョブから第2階層ストレージへのアクセスは、基本的に第2階層ストレージのキャッシュ領域を介して行われる。
 - 第2階層ストレージに直接アクセスしたい場合、2ndfs領域（パス名：/2ndfs/<groupname>）を用いることができる。

- 共通ファイル配布機能を使用しない場合、計算ノード上のジョブは、一つのストレージI/Oノードにアクセスが集中し、性能劣化を引き起こす可能性がある。

```
#!/bin/bash
#PJM -L "node=2"
#PJM --llio localtmp-size=10Gi,sharedtmp-size=5Mi
mpiexec ./a.exe
```

(1)

ジョブに割り当てられた計算ノードが属するBoB



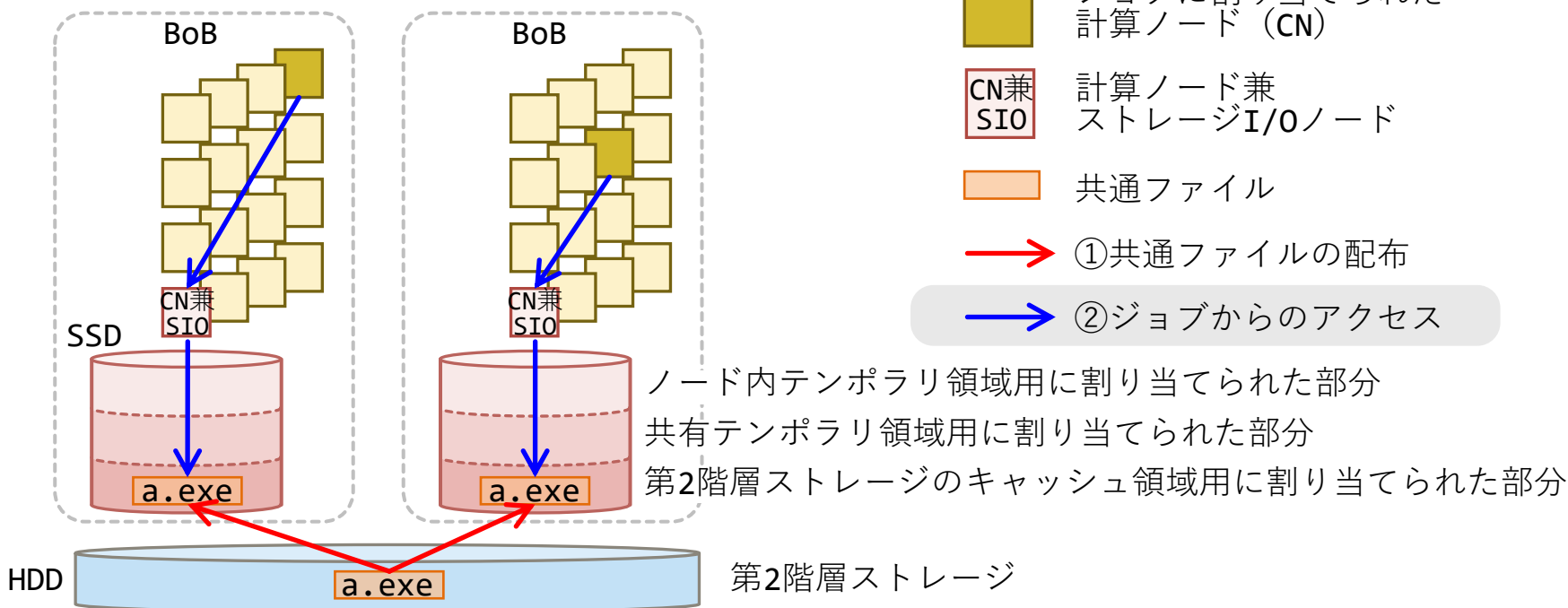
■ 共通ファイル配布機能を使用する場合 [1/2]

- 共通ファイル配布機能を使用する場合、ジョブに割り当てられた各計算ノードが属するBoBのストレージI/O (SIO) ノードにつながる各SSDの第2階層ストレージキャッシュ領域に共通ファイルが配布される。
- 各計算ノード上のジョブは自ノードが属するBoBのSIOノードにアクセスすることで、SIOノードのアクセス負荷が分散される。

```
#!/bin/bash
#PJM -L "node=2"
#PJM --llio localtmp-size=10Gi,sharedtmp-size=5Mi
llio_transfer ./a.exe ← ファイルa.exeを第2階層ストレージのキャッシュへコピー
mpiexec ./a.exe ← 第2階層ストレージキャッシュ上のa.exeを実行
llio_transfer --purge ./a.out
```

①
②
③

ジョブに割り当てられた計算ノードが属するBoB



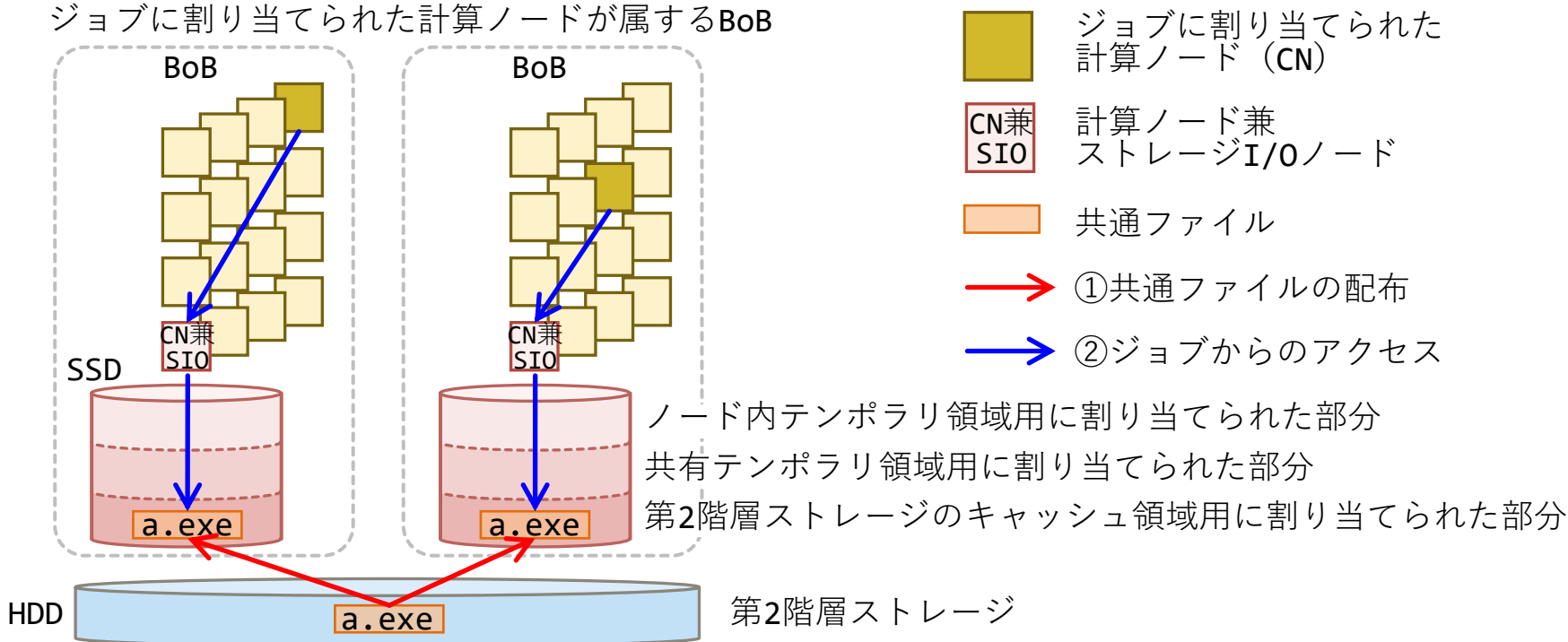
■ 共通ファイル配布機能を使用する場合 [2/2]

- 第2階層ストレージのキャッシュ領域にコピーされた共通ファイルは、この領域が一杯になるまで削除されることはない。キャッシュ領域にコピーした共通ファイルがジョブスクリプトの途中で不要となる場合は、キャッシュ容量確保のため、③のように`llio_transfer --purge` で適宜削除するとよい。
- 第2階層ストレージのキャッシュ領域の共通ファイルは、ジョブ終了後には自動的に削除される。

```
#!/bin/bash
#PJM -L "node=2"
#PJM --llio localtmp-size=10Gi,sharedtmp-size=5Mi
llio_transfer ./a.exe ← ファイルa.exeを第2階層ストレージのキャッシュへコピー
mpiexec ./a.exe ← 第2階層ストレージキャッシュ上のa.exeを実行
llio_transfer --purge ./a.out ← 第2階層ストレージキャッシュ上のa.exeを削除
```

①
②
③

ジョブに割り当てられた計算ノードが属するBoB



■ 共通ファイル配布機能に関する注意事項

`llio_transfer`コマンドで第2階層ストレージのキャッシュへコピーした共通ファイルに関して、以下の点に注意してください。

```
#!/bin/bash
```

```
...
```

```
llio_transfer ./a.exe
```

```
mpiexec ./a.exe
```

```
llio_transfer --purge ./a.out
```

① (注1~5)

②

③ (注6,7)

(注1) `llio_transfer`コマンドは読み込みを行うだけのファイルが対象である。

(注2) 共通ファイルのコピー先である、第1階層ストレージの"第2階層ストレージのキャッシュ領域"には共通ファイル全体を格納できるだけの容量が必要である。

(注3) `llio_transfer`コマンド使用前に対象のファイルをオープンすると、キャッシュデータが作成され`llio_transfer`コマンドがエラーになる場合がある。

(注4) 共通ファイルに対するファイルロックは未サポートである。

(注5) ジョブ実行中は第2階層ストレージにある共通ファイルのコピー元を変更したり削除したりしてはならない。

- コピー元を変更した場合は、第2階層ストレージのキャッシュにコピーした共通ファイルの内容が不定になる可能性がある。
- コピー元を削除した場合は、共通ファイルを`llio_transfer`コマンドの`--purge`オプションで削除できなくなり、ジョブが終了するまで第2階層ストレージのキャッシュ領域が共通ファイルに占有されたままになる。

(注6) ジョブスクリプト内で共通ファイルを削除する場合は、`llio_transfer --purge`で削除する。`rm`コマンドはエラーで失敗し、削除できない。

(注7) `llio_transfer --purge`で第2階層ストレージのキャッシュから共通ファイルを削除した直後は、ジョブからの共通ファイルのコピー元ファイルの削除、ファイルデータ更新やファイル属性更新の操作がエラーで失敗する場合がある。その場合は、60秒待ってから操作を再実行してください。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
 - 「富岳」ストレージ構成の概要
 - LLIOの機能概略
 - 第1階層ストレージの三つの領域
 - 第1階層ストレージのサイズ
 - 共通ファイル配布機能
 - **【参考】LLIO使用サンプル**
- 「富岳」のMPI通信
- 【付録】参考文献

本節ではLLIOの使用例として、共通ファイル配布機能、ノード内テンポラリ領域、およびLLIO性能情報を取得するサンプルプログラム、サンプルジョブスクリプト、および取得したLLIO性能情報の集計例を示します。

サンプルプログラムは（時間ステップループに相当する）ループの各反復で、MPI並列の各プロセスが自プロセス固有のファイルに書き出し・読み込みを行うFortranプログラムです。ファイルのオープン時に第2階層ストレージのファイルパスを指定すると、第2階層ストレージのキャッシュ領域を介してファイルにアクセスします。この方法をケース(a)とします。「第1階層ストレージの三つの領域」の節で説明したように、ランク・ノードごとに独立した中間ファイルや一時ファイルの作成・参照には、ノード内テンポラリ領域が適しています。そこで、ケース(b)として、ファイルオープン時にファイルパスとしてノード内テンポラリ領域を使用するように指定します。

サンプルジョブスクリプトとして、ケース(b)のジョブスクリプトを示します。ノード内テンポラリ領域および共通ファイル配布機能を使用し、LLIO性能情報を取得をするように指定します。（なお、ケース(a)ではノード内テンポラリ領域および共通ファイル配布機能は使用せず、LLIO性能情報のみを取得をするように指定します。ケース(a)と**(b)**で異なる部分は適宜コメントします。）

取得したLLIO性能情報の集計例では、ケース(a)と**(b)**の実行で得られたLLIO性能情報から集計を行うawkスクリプトを利用して集計し、集計結果を比較する例を示します。

なお、本節の目的は、LLIOの使用方法を示すことであり、機能の比較結果から性能を判断することではありません。

- プログラム例
- ジョブスクリプト例
- LLIO性能情報の集計例

■ プログラム例 [1/3]

以下のスライド数ページに渡って、LLIOのノード内テンポラリ領域を使用する場合のFortranのプログラム例を示す。

- （時間ステップループに相当する）ループの各反復で、MPI並列の各プロセスが自プロセス固有のファイルに書き出し・読み込みを行う。
 - ②の行の配列Aの要素数を①の行のように設定し、要素数134217728の倍精度の配列(1GB)の書き出し・読み込みを行う。
- 次スライドへ続く

PROGRAM MAIN

プログラム例 (Fortran)

IMPLICIT NONE

INCLUDE 'mpif.h'

INTEGER,PARAMETER::NLOOP=100

INTEGER,PARAMETER::N=134217728 ! 1 GB FOR REAL*8 ①

DOUBLE PRECISION::A(N) ②

INTEGER::IERR,NPROCS,MYRANK,NF,LL

CHARACTER(LEN=8)::CRANK

CHARACTER(LEN=256)::FILENAME

DOUBLE PRECISION::T1,T2

CALL MPI_INIT(IERR)

CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)

(次のスライドへ続く)

■ プログラム例 [2/3]

■ ファイルオープン時のパスの指定：

- ケース(a)：第2階層ストレージのファイルパスを指定し、第2階層ストレージのキャッシュ領域を介してファイルにアクセスする。
 - ③の行をコメントにして、④の行を復活させる。
- ケース(b)：ノード内テンポラリ領域を使用するようにファイルパスを指定する。
 - ④で入出力ファイルのパスを指定する際、LLIOのノード内テンポラリ領域を使用するよう、環境変数PJM_LOCALTMPの値/localをパス名の先頭に付ける。

■ 次スライドへ続く

(前のスライドからの続き)

プログラム例 (Fortran)

```
A=0D0
!:: 出力ファイルの拡張子をランク番号にするための処理
WRITE(CRANK,'(I0.6)') MYRANK
WRITE(*,602) 'NPROCS= ',NPROCS,'MYRANK= ',MYRANK,'N= ',N,&
& DBLE(N)*8D0/1024D0/1024D0,'MB','NLOOP= ',NLOOP
602 FORMAT(1X,2(A,I6,2X),(A,I10,2X),(F8.3,1X,A,2X),(A,I6,2X))
!:: LLIOのノード内テンポラリ領域を使用する場合
FILENAME = '/local/outfile.'//TRIM(ADJUSTL(CRANK)) ③
!:: 第2階層ストレージのパスを使用する場合
!!!! FILENAME = 'outfile.'//TRIM(ADJUSTL(CRANK)) ④
(次のスライドへ続く)
```

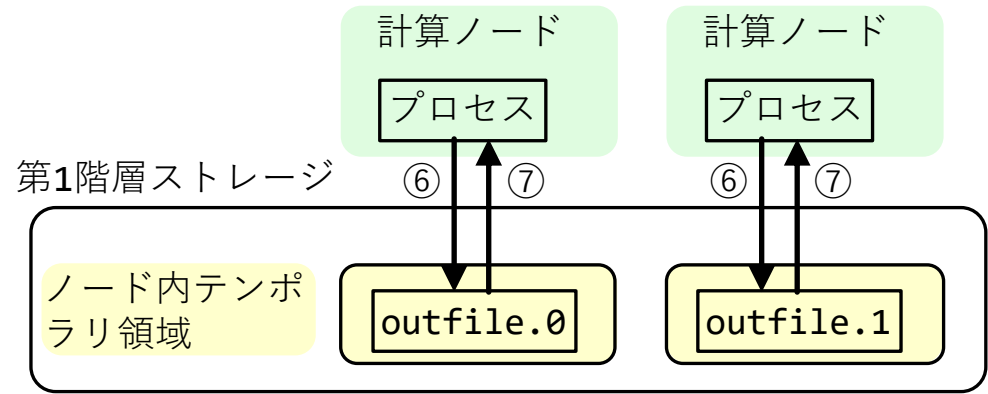

※本図は1ノードに1プロセスを配置した例

■ プログラム例 [3/3]

■ ⑤のDOループ内の処理：

- ⑥：ファイルオープン、バイナリで書き出し、クローズ。
- ⑦：ファイルオープン、バイナリで読み込み、クローズ。
- 以上を⑤のDOループにより100回繰り返す。

以上でプログラム例の説明を終わる。



(前のスライドからの続き)

プログラム例 (Fortran)

```

NF=10
DO LL = 1,NLOOP
  CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)
  T1 = MPI_WTIME()
  OPEN(UNIT=NF, FILE=FILENAME, FORM='UNFORMATTED', &
    & STATUS='UNKNOWN', IOSTAT=IERR)
  WRITE(NF) A
  CLOSE(NF)
  OPEN(UNIT=NF, FILE=FILENAME, FORM='UNFORMATTED', &
    & STATUS='UNKNOWN', IOSTAT=IERR)
  READ(NF) A
  CLOSE(NF)
  T2 = MPI_WTIME()
  WRITE(*,601) 'MYRANK= ',MYRANK,&
    & 'LL= ',LL,'ELAPSED(SEC)= ',T2-T1
601 FORMAT(1X,2(A,I6,2X),(A,1PE15.6,2X))
ENDDO ! LL
(以下、省略)
    
```

■ ジョブスクリプト例 [1/5]

以下のスライド数ページに渡って、ジョブスクリプト例を示す。

- ①②のPJMオプションで第1階層ストレージ上の使用サイズを指定する（ケース**(b)**に対してのみ必要）。
 - ①「`--llio localtmp-size=size_L`」：1ノード当たりのノード内テンポラリ領域のサイズを指定（本スクリプトで使用する）。
 - ②「`--llio sharedtmp-size=size_S`」：「共有テンポラリ領域全体のサイズ」÷「割当てノード数」を指定（本スクリプトでは不使用）。
 - 注意：`size_S`には、1ノード当たりのサイズを指定する。共有テンポラリ領域全体のサイズは、「`size_S`の値」×「ジョブに割り当てるノード数」である。
- ③のPJMオプションでLLIO性能情報を出力すると指定する（ケース**(a)**と**(b)**に共通）。

■ 次スライドへ続く

```
#!/bin/bash
```

（中略）

```
#PJM --llio localtmp-size=10Gi
```

```
#PJM --llio sharedtmp-size=1Mi
```

```
#PJM --llio perf
```

（次のスライドへ続く）

①

②

③

■ ジョブスクリプト例 [2/5]

- 第1階層ストレージ上のノード内テンポラリ領域と共有テンポラリ領域のアクセスパスは④⑤のechoコマンドで確認できる。

- ④⑤のechoコマンドの結果は以下となる。

PJM_LOCALTMP=/local

PJM_SHAREDTMP=/share

- 次スライドへ続く

(前のスライドからの続き)

ジョブスクリプト

echo "PJM_LOCALTMP=\${PJM_LOCALTMP}"

④

echo "PJM_SHAREDTMP=\${PJM_SHAREDTMP}"

⑤

(次のスライドへ続く)

■ ジョブスクリプト例 [3/5]

- ロードモジュールとジョブスクリプトは、⑥の`${PJM_O_WORKDIR}`で指定されるジョブ実行用ディレクトリに置かれていると仮定する。⑥でロードモジュールのパスを設定する。
- [ケース(b)の場合のみ] ⑦の`llio_transfer`コマンドでロードモジュールを第2階層ストレージのキャッシュ領域に置く。
 - ⑦はジョブで使用する全てのSIOに接続されているSSDにロードモジュールをコピーする（共通ファイル配布機能）。
 - ケース(a)の場合は⑦の行をコメントアウトする。

(前のスライドからの続き)

```
LD=${PJM_O_WORKDIR}/a.exe
```

```
llio_transfer ${LD}
```

(次のスライドへ続く)

ジョブスクリプト

⑥

⑦

■ 次スライドへ続く

■ ジョブスクリプト例 [4/5]

■ ⑧のmpirexecコマンドにより、ロードモジュールを実行する。

- ケース(a)の場合、第2階層ストレージの上のロードモジュールを（第2階層キャッシュ領域を介して）実行する。
- ケース(b)の場合、共通ファイル配布機能によりジョブに割り当てられたノードが属する各SSD内の第2階層キャッシュ領域上にコピーされたロードモジュールを実行する。
- ケース(b)の場合、**a.exe**の実行により出力ファイル**outfile.***が第1階層ストレージ上のノード内テンポラリ領域に出力されると仮定する（前小節のプログラム例参照）。

（前のスライドからの続き）

ジョブスクリプト

```
rm -f outfile.* ; rm -rf zzoutput
mpirexec -stdout-proc ¥                                     ⑧
        ./zzoutput/%/1000r/stdout_LLIO ¥
        -stderr-proc ¥
        ./zzoutput/%/1000r/stderr_LLIO ${LD}
```

（次のスライドへ続く）

※ 「¥」 記号は「\」を入力してください。

- ⑧におけるディレクトリ**zzoutput**は第2階層ストレージ上の最終的な実行結果格納用ディレクトリとする。

■ 次スライドへ続く

■ ジョブスクリプト例 [5/5]

- [ケース(b)の場合のみ] ⑨の2回目の**mpirexec**コマンドで第1階層ストレージから第2階層ストレージに出力ファイルをコピーする。

- 環境変数**PLE_RANK_ON_NODE**の値を参照し、**if**文によりノード内の1プロセスのみを利用してコピーを行う。

(前のスライドからの続き) ジョブスクリプト

```
mpirexec (標準出力関連オプションは省略) sh -c ¥ ⑨
'if [ ${PLE_RANK_ON_NODE} == 0 ]; then ¥
    cp -p ${PJM_LOCALTMP}/outfile.* ¥
    ./zzoutput/; ¥
fi'
(下の図へ続く)
```

※「¥」記号は「\」を入力してください。また、表示の簡略化のため標準出力関連オプションは省略しました。前スライドの⑧の行を参考に適宜指定して下さい。

- [ケース(b)の場合のみ] ジョブスクリプトの途中でロードモジュールを第2階層ストレージのキャッシュ領域から削除したい場合など必要に応じて⑩の **llio_transfer --purge** を実行する。
- [ケース(b)の場合のみ] ノード内テンポラリ領域上のファイル、共有テンポラリ領域上のファイル、および**llio_transfer**でコピーした第2階層キャッシュ領域上のファイルはジョブ終了後に自動的に削除される。

以上でジョブスクリプト例の説明を終わる。

(上の図からの続き) ジョブスクリプト

```
llio_transfer --purge ${LD} ⑩
exit
```

■ LLIO性能情報の集計例 [1/5]

前前小節のプログラムと前小節のジョブスクリプトについてケース(a)とケース(b)をジョブ実行し取得されたLLIO性能情報ファイルからファイルアクセス時間を集計する例を示します。さらに、集計されたケース(a)とケース(b)の結果を比較する例を示します。

ケース(a)：共通ファイル配布機能不使用、第2階層ストレージのファイルパスを指定（第2階層ストレージのキャッシュ領域を介してファイルにアクセス）、ノード内テンポラリ領域不使用。

ケース(b)：共通ファイル配布機能使用、ノード内テンポラリ領域使用。

- ケース(a)とケース(b)について、前小節のジョブスクリプトをpjsubコマンドでジョブ実行する。ジョブ終了後、各ケースについて、拡張子が.llio_perfのLLIO性能情報ファイル（ジョブ名.ジョブ番号.llio_perf）が出力されていることを確認する。

■ LLIO性能情報の集計例 [2/5]

- 以下の集計スクリプト **sample.awk(*1)** を使用して、LLIO性能情報からノード内テンポラリ領域、共有テンポラリ領域、第2階層キャッシュ領域それぞれで要した時間を算出することとする。

(*1) 以下のURLからの引用：「利用手引書 利用およびジョブ実行編 8.7.1. LLIO性能情報」https://www.fugaku.rccs.riken.jp/doc_root/ja/user_guides/use_latest/LayeredStorageAndLLIO/IOprofiling.html#llio

```
$ cat sample.awk
/^SIO Infomation/,/END/ {
    sumflg=1
}
/^I¥/O      2ndLayerCache    NodeTotal/,/^I¥/O      2ndLayerCache    ComputeNode/ {
    if($3 == "Sum" && sumflg == 1){sum1 += $6}
}
/^I¥/O      SharedTmp       NodeTotal/,/^I¥/O      SharedTmp       ComputeNode/ {
    if($3 == "Sum" && sumflg == 1){sum2 += $6}
}
/^I¥/O      LocalTmp        NodeTotal/,/^I¥/O      LocalTmp        ComputeNode/ {
    if($3 == "Sum" && sumflg == 1){sum3 += $6}
}
(次のスライドへ続く)
```

集計スクリプト sample.awk

■ LLIO性能情報の集計例 [3/5]

■ 集計スクリプトsample.awk (続)

(前のスライドからの続き)

集計スクリプトsample.awk

```
/^Meta      2ndLayerCache      NodeTotal/,/^Meta      SharedTmp      NodeTotal/ {  
    if(NF == 3 && sumflg == 1){sum4 += $3}  
}  
/^Meta      SharedTmp      NodeTotal/,/^Meta      LocalTmp      NodeTotal/ {  
    if(NF == 3 && sumflg == 1){sum5 += $3}  
}  
/^Meta      LocalTmp      NodeTotal/,/^Resource      2ndLayerCache  
CacheOperation/ {  
    if(NF == 3 && sumflg == 1){sum6 += $3}  
}  
END {  
    total = sum1+sum2+sum3+sum4+sum5+sum6;  
    printf("%-20s %-20s %15s¥n", "      Area", "Time(us)", "% of Time");  
    printf("%-20s %20d %15.1f¥n", "Meta 2ndLayerCache", sum4, (sum4/total)*100);  
    printf("%-20s %20d %15.1f¥n", "      SharedTmp", sum5, (sum5/total)*100);  
    printf("%-20s %20d %15.1f¥n", "      LocalTmp", sum6, (sum6/total)*100);  
    printf("%-20s %20d %15.1f¥n", "I/O 2ndLayerCache", sum1, (sum1/total)*100);  
    printf("%-20s %20d %15.1f¥n", "      SharedTmp", sum2, (sum2/total)*100);  
    printf("%-20s %20d %15.1f¥n", "      LocalTmp", sum3, (sum3/total)*100);  
}
```

- ケース(a)とケース(b)のそれぞれについて、集計スクリプトの引数にLLIO性能情報ファイルを指定してスクリプトを実行する。

```
$ awk -f sample.awk ジョブ名.ジョブ番号.llio_perf
```

■ LLIO性能情報の集計例 [4/5]

- 集計スクリプトで算出した時間（us単位）の比較結果を以下に示す。本例では、**2ndLayerCache**で示される第2階層ストレージのキャッシュ領域について、**Meta**欄で示されるメタアクセスの時間が大きく異なっていることが確認される。

ケース(a)：共通ファイル配布機能不使用、第2階層ストレージのファイルパスを指定（第2階層ストレージのキャッシュ領域を介してファイルにアクセス）、ノード内テンポラリ領域不使用。

ケース(b)：共通ファイル配布機能使用、ノード内テンポラリ領域使用。

		ケース(a)		ケース(b)	
	Area	Time(us)	% of Time	Time(us)	% of Time
Meta	2ndLayerCache	566,764,563	76.2	6,728,606	3.7
	SharedTmp	1,322	0.0	1,522	0.0
	LocalTmp	253	0.0	61,185	0.0
I/O	2ndLayerCache	177,465,699	23.8	832,333	0.5
	SharedTmp	0	0.0	0	0.0
	LocalTmp	0	0.0	173,373,732	95.8

※ 数値データ簡素化のため、ノード数1、ノード当たりのプロセス数1で実行。

- 次のスライドでメタアクセスの時間の内訳を調べる。

■ LLIO性能情報の集計例 [5/5]

- メタアクセスの時間の内訳を調べるため、LLIO性能情報ファイル内の、見出しが以下の項目に着目する。これらの項目に続く数行には、内訳としてopen、close等のメタアクセスの回数（Count）及び時間（Time）が記録されている。

Meta (中略)	2ndLayerCache	NodeTotal	Count	Time(us)
--------------	---------------	-----------	-------	----------

Meta (中略)	LocalTmp	NodeTotal	Count	Time(us)
--------------	----------	-----------	-------	----------

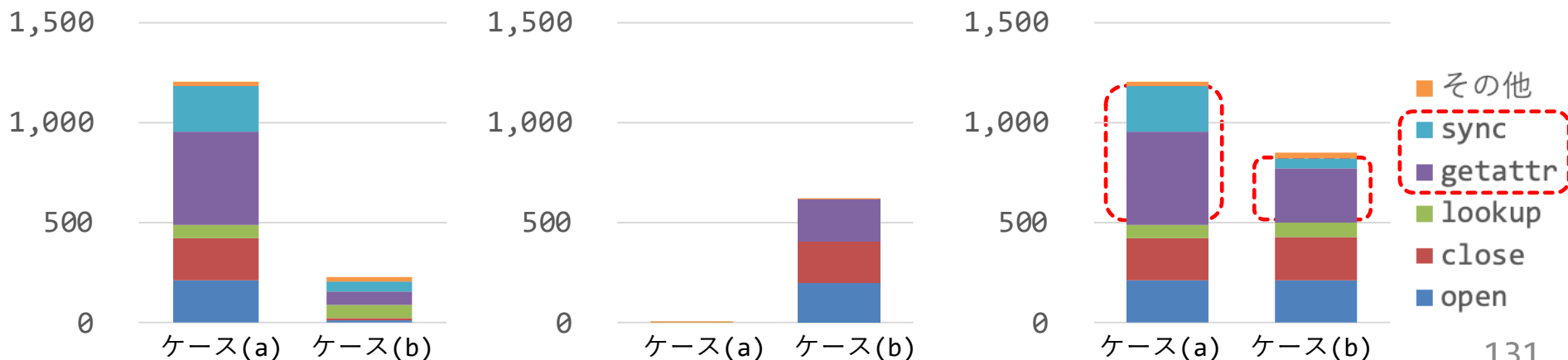
- LLIO不使用の場合とLLIO使用した場合の比較結果を以下のグラフに示す。一番右のグラフから、LLIOのノード内テンポラリ領域を使用したケース(b)はメタアクセスの内訳getattrとsyncの回数が減少した事が確認できる。

メタアクセスの内訳（Countの値）

第2階層キャッシュ領域

ノード内テンポラリ領域

第2階層キャッシュ領域と
ノード内テンポラリ領域の合計



内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- 【付録】参考文献

■ 「富岳」のMPI通信

本章の内容

- ◇ 多数のMPIプロセス間で通信するMPIプログラムを実行する場合、効率的な実行のために、通信性能の改善が必要となることがあります。
- ◇ 通信性能改善の検討のためには、対象となるMPIルーチンの通信方式や通信回数等の情報が必要です。「富岳」ではMPI通信に関する統計情報を採取する機能が提供されています。MPI通信に関する統計情報をMPI統計情報と呼びます。本章ではMPI統計情報の採取方法を説明します。（MPI通信時間および待ち時間の調査には、詳細プロファイラの利用も有効です。詳細プロファイラは本セミナー中級編の第一部で説明していますので、そちらを参照して下さい。）
- ◇ MPI統計情報に出力された内容を理解し改善策を検討するには、「富岳」のMPI通信に固有な事項を理解しておくことが必要です。本章ではその目的で、通信モード、集団通信を高速化する技術であるバリア通信機能とそれを用いた集団通信、一対一通信での通信方式であるイーガー通信方式とランデブー通信方式について説明します。
- ◇ 「富岳」では、MPIプログラム実行時に、ユーザがMPIライブラリの動作をパラメータ（MCAパラメータ）である程度変更することができる機能が提供されています。本章ではこれらについても説明します。
- ◇ 大規模なMPIプログラムの実行ではメモリ使用量が問題となることがあります。メモリ使用量について、ユーザはMCAパラメータによりある程度制御することができます。本章の最後では、メモリ使用量を抑えるための考慮点について説明します。

■ 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル

■ MPI統計情報

■ 「富岳」の通信モード

■ 高速な集団通信

■ イーガー通信方式とランデブー通信方式

■ メモリ使用量を抑えるための考慮点

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
 - 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル
 - MPI統計情報
 - 「富岳」の通信モード
 - 高速な集団通信
 - イーガー通信方式とランデブー通信方式
 - メモリ使用量を抑えるための考慮点
- 【付録】参考文献

■ 「富岳」のMPIライブラリのMPI規格への対応レベル

「富岳」で提供されるMPIライブラリは、MPI-3.1規格およびMPI-4.0（仮称）規格の一部に準拠している。

C++ bindingは、MPI-2.2規格の範囲でサポートされている。

■ マルチスレッド環境での動作

「富岳」で提供されるMPIライブラリはマルチスレッド環境での動作に対応している。MPI規格でのマルチスレッドのサポートレベルの値はMPI_THREAD_SERIALIZEDである。

■ MPI_THREAD_SERIALIZED

- 複数のスレッドからMPI関数の呼出しは可能であるが、注意点として、その場合には同時に呼び出すことはできず、スレッド間で逐次に呼び出さなければならない。
- スレッド間のMPI呼び出しの逐次化はユーザが行う必要がある。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
 - 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル
 - **MPI統計情報**
 - 「富岳」の通信モード
 - 高速な集団通信
 - イーガー通信方式とランデブー通信方式
 - メモリ使用量を抑えるための考慮点
- 【付録】参考文献

■ 「富岳」のMPI通信

■ MPI統計情報

本節の内容

「富岳」ではMPIプログラム内で使用しているMPI通信ルーチンの通信回数などの統計情報を採取する機能が提供されています。このMPI通信に関する統計情報をMPI統計情報と呼びます。ユーザはMPIプログラム実行時にMPI統計情報を採取するかどうかをMCAパラメータで指定できます。

本節ではMPI統計情報を採取する方法を説明します。

- MPI統計情報の概要
- 全体出力モード
- 区間出力モード

■ 「富岳」のMPI通信

■ MPI統計情報

本節の内容

■ MPI統計情報の概要

- 全体出力モード
- 区間出力モード

■ 統計情報の種類

MPI統計情報で取得できる統計情報の種類を以下の表に示す。

対象ルーチン		統計情報の種類
MPI通信ルーチン		通信回数、ホップ数、通信量、通信時間。
	MPIの <u>集団</u> 通信ルーチン	Tofuインターコネクトの高機能バリア通信機能を使用しているかどうか、およびそれらの通信回数。
		Tofuインターコネクト向けにチューニングされたアルゴリズムを使用しているかどうか、およびそれらの通信回数。
	MPIの <u>一対一</u> 通信ルーチン	イーガー通信方式とランデブー通信方式のどちらで通信しているか、およびそれらの通信回数。

■ MPI統計情報の出力モード

MPI統計情報の出力方法には全体出力モードと区間指定出力モードの2種類がある。

■ 全体出力モード

MPIプログラム全体を対象としてMPI統計情報を採取し出力する。

■ 区間出力モード

ユーザが指定した区間を対象としてMPI統計情報を採取し出力する。

■ MPI統計情報の出力方法

全体出力モードで出力するか、区間出力モードで出力するかは、MCAパラメータ `mpi_print_stats` によって制御する。MCAパラメータは以下のいずれかの方法で指定することができる。

■ `mpiexec` コマンドの `-mca` オプションの引数で指定する。

例：`mpiexec -mca mpi_print_stats 値`

↑
= でなく スペース

■ 環境変数で指定する。

環境変数で指定する場合は、MCAパラメータ名の前に「OMPI_MCA_」を付ける。

例：`export OMPI_MCA_mpi_print_stats=値`

本MCAパラメータで指定できる 値 については次のスライドで説明する。

■ MCAパラメータmpi_print_statsの値による出力内容および出力先

- mpi_print_statsの値が1または2の場合全体出力モードと指定され、3または4の場合区間出力モードと指定される。
- 出力先は、値が1または3の場合ランク0のプロセスのみの標準エラー出力となり、値が2または4の場合それぞれのMPIプロセス自身の標準エラー出力となる。

mpi_print_statsの値	モード	出力内容	区間指定ルーチン(*1) (後述)	出力先
1	全体出力モード	すべてのMPIプロセスのMPI統計情報が総括される。	不要	MPI_COMM_WORLDのランク0プロセスの標準エラー出力
2		各MPIプロセスごとのMPI統計情報。		それぞれのMPIプロセス自身(*2)の標準エラー出力
3	区間出力モード	すべてのMPIプロセスのMPI統計情報が総括される。	必要 (後述)	MPI_COMM_WORLDのランク0プロセスの標準エラー出力
4		各MPIプロセスごとのMPI統計情報。		それぞれのMPIプロセス自身(*2)の標準エラー出力

(*1) FJMPI_Collection_clear、FJMPI_Collection_start、FJMPI_Collection_stop、およびFJMPI_Collection_print (後述)

(*2) 特定のMPIプロセスを出力対象としたい場合、MCAパラメータmpi_print_stats_ranksによって指定できる。

■ 「富岳」のMPI通信

■ MPI統計情報

本節の内容

■ MPI統計情報の概要

■ 全体出力モード

■ 区間出力モード

■ 全体出力モードのためのプログラム例

以下のプログラム例について、プログラム内の全MPI通信ルーチンの統計情報を取得する方法を説明する。本例ではMPI_InitとMPI_Finalizeの間にMPI_Bcastを2回呼んでいる。

プログラム例

```
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1] = {};
    MPI_Bcast(&A[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
    (中略)
    MPI_Bcast(&A[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
    (中略)
    MPI_Finalize(); return 0;
}
```

■ 以降のスライドで、全体出力モードの出力例として以下の例を説明する。

- 全体出力モードの例：ランク0のみ出力

■ 全体出力モードの例：ランク0のみ出力：環境変数の設定

- ジョブスクリプトにおいて、①の環境変数 `export OMPI_MCA_mpi_print_stats=1` を指定すると、すべてのMPIプロセスのMPI統計情報が総括され、MPI_COMM_WORLDのランク0の標準エラー出力に出力される。

ジョブスクリプト

```
#!/bin/bash
(中略)
#PJM -L node=2x2x2
#PJM --mpi max-proc-per-node=4
#PJM -S
export OMPI_MCA_mpi_print_stats=1
export PLE_MPI_STD_EMPTYFILE=off
mpirexec -stdout-proc stdout -stderr-proc stderr ./a.exe
```

①

標準エラー出力を格納するファイル

```
$ ls stderr.1.0
stderr.1.0
```


■ 全体出力モードの例：ランク0のみ出力：統計情報の内容 [1/3]

ランク0の標準エラー出力に書き出された統計情報の内容（抜粋）を以降のスライドにわたって説明する。

- 「Tofu Barrier Collective Communication Count」欄には、Tofuインターコネクトの高機能バリア通信機能（後の節で説明）を利用した各集団通信ルーチンの呼び出し回数が表示される。

- 各項目について、全プロセス中の
 - ・ 最大値 [とそのランク値]、
 - ・ 最小値 [とそのランク値]、
 - ・ 平均値
 が表示される。

```
=====
/***** MPI Statistical Information *****/
=====
(中略)
----- Tofu Barrier Collective Communication Count -----
                                MAX                MIN                AVE
Bcast                          2 [  0]            2 [  0]            2.0
Reduce                         0 [  0]            0 [  0]            0.0
Allreduce                      0 [  0]            0 [  0]            0.0
(以下、省略)
```

プログラム例のMPI_Bcastの情報

※ ReduceとAllreduceは本プログラム例内に使用されていなくても項目としては表示される。

■ 全体出力モードの例：ランク0のみ出力：統計情報の内容 [2/3]

- 「MPI Information」欄には、MPI_COMM_WORLDに属する並列プロセスが配置されているトーラス構成の、次元数(Dimension)、および、プロセス形状(Shape)が表示される。

```
=====
/***** MPI Statistical Information *****/
=====
----- MPI Information -----
Dimension                3
Shape                    2x2x2
----- MPI Memory Usage (MiB) -----
(以下、省略)
```

- 「MPI Memory Usage (MiB)」欄には、MPIライブラリのメモリ使用量の概算値が表示される。

```
=====
/***** MPI Statistical Information *****/
=====
(中略)
----- MPI Memory Usage (MiB) -----
              MAX              MIN              AVE
Estimated_Memory_Size  89.03 [  0]  89.03 [  0]  89.03
(以下、省略)
```

■ 全体出力モードの例：ランク0のみ出力：統計情報の内容 [3/3]

- 「Process Mapping」欄には、MPI_COMM_WORLDに属するすべての並列プロセスのランク番号と論理座標の対応一覧が表示される。本情報は、ランクが0の並列プロセスだけによって出力される。

```
=====
/***** MPI Statistical Information *****/
=====
(中略)
----- Process Mapping -----
(0,0,0)      0,1,2,3
(1,0,0)      4,5,6,7
(0,1,0)      8,9,10,11
(1,1,0)      12,13,14,15
(0,0,1)      16,17,18,19
(1,0,1)      20,21,22,23
(0,1,1)      24,25,26,27
(1,1,1)      28,29,30,31
```

論理座標

ランク番号

以上で例を終わる。

■ 「富岳」のMPI通信

■ MPI統計情報

- MPI統計情報の概要

- 全体出力モード

- 区間出力モード

本節の内容

■ 指定方法の概略

- ソースプログラムに計測したい区間と計測結果を出力する地点を指定する：

- ◆ Fortranの場合：

- 計測したい区間を

- `CALL FJMPI_COLLECTION_START` と

- `CALL FJMPI_COLLECTION_STOP`

- で囲む。複数の区間を指定することができる（その場合リセットするまで計測結果が累積される）。

- 計測結果を出力したい地点で`CALL FJMPI_COLLECTION_PRINT`を実行する。
 - 必要に応じて計測結果のリセットを`CALL FJMPI_COLLECTION_CLEAR`で行う。

- ◆ C/C++の場合：

- ヘッダファイルの読み込み`#include <mpi-ext.h>`を指定する。
 - Fortranの場合と同様に`FJMPI_collection_{clear, start, stop, print}`を指定する。

- ジョブスクリプトに

- 環境変数 `export OMPI_MCA_mpi_print_stats=3` または `4` を指定する。

以降のスライドで指定方法の例を示す。

■ 区間出力モードの例 [1/7]

図の①と②に示すように複数のMPIルーチンを使用している場合、各ルーチンごとに別々に統計情報を取得する方法を以降のスライドにわたって説明する。

プログラム例

```
#include <mpi.h>
#include <mpi-ext.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1] = {};
    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT, ①
                0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_1"));

    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT, ②
                0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_2"));
    (中略)
    MPI_Finalize(); return 0;
}
```

■ 区間出力モードの例 [2/7]

- ソースプログラムにヘッダファイルの読み込みの指定

```
#include <mpi-ext.h> (※)
```

をする。

※ C/C++の場合のみ

- ソースプログラムで別々にMPI統計情報を取得したい区間を囲むように
FJMPI_Collection_
{clear, start, stop, print}
を指定する。

プログラム例

```
#include <mpi.h>
#include <mpi-ext.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1] = {};
    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT,
                  0, MPI_COMM_WORLD); ①
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_1"));

    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT,
                  0, MPI_COMM_WORLD); ②
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_2"));
    (中略)
    MPI_Finalize(); return 0;
}
```

■ 区間出力モードの例 [3/7]

③：その時点までの統計情報をリセット。

④：統計情報の取得を開始。

⑤：統計情報の取得を中断。

プログラム例

```
#include <mpi.h>
#include <mpi-ext.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1] = {};
    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT,
                  0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_1"));

    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT,
                  0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_2"));
    (中略)
    MPI_Finalize(); return 0;
}
```

③
④
①
⑤
②

■ 区間出力モードの例 [4/7]

⑥：この時点での統計情報が
下の図の **1** に書き出される。

Time(sec): 区間指定ごとの経過時間（秒）。
MPI通信ルーチン以外に計算部分があればその時間も含む。

プログラム例

```
#include <mpi.h>
#include <mpi-ext.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1] = {};
    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT,
                  0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_1"));
    FJMPI_Collection_clear();
    FJMPI_Collection_start();
```

(省略)

----- MPI Information -----

(中略)

Section	1 REGION_1	MAX	MIN	AVE
Time(Sec)	0.00 [18]	0.00 [2]	0.00	0.00

----- Per-peer Communication Count -----

(以下、省略)

1

■ 区間出力モードの例 [5/7]

⑦：その時点までの統計情報をリセット。

- ⑦を実行しない場合、以後の各測定値は累積される。

⑧：統計情報の取得を再開。

⑨：統計情報の取得を中断。

```
#include <mpi.h>
#include <mpi-ext.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1] = {};
    FJMPI_Collection_clear();
    FJMPI_Collection_start();
    {
        MPI_Bcast(&A[0], 1, MPI_INT,
                  0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop();
    FJMPI_Collection_print
        (const_cast<char *>("REGION_1"));

    FJMPI_Collection_clear(); ⑦
    FJMPI_Collection_start(); ⑧
    {
        MPI_Bcast(&A[0], 1, MPI_INT, ②
                  0, MPI_COMM_WORLD);
    }
    FJMPI_Collection_stop(); ⑨
    FJMPI_Collection_print
        (const_cast<char *>("REGION_2"));
    (中略)
    MPI_Finalize(); return 0;
}
```

■ 区間出力モードの例 [6/7]

```
#include <mpi.h>
#include <mpi-ext.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    (中略)
    int A[1000000];
    ...
}
```

(省略)

----- MPI Information -----

(中略)

Section	2 REGION_2	MAX	MIN	AVE
Time(Sec)		0.00 [18]	0.00 [2]	0.00

----- Per-peer Communication Count -----

(以下、省略)

2

⑩：この時点での統計情報が
上の図の 2 に書き出される。

```
FJMPI_Collection_clear();           ⑦
FJMPI_Collection_start();           ⑧
{
    MPI_Bcast(&A[0], 1, MPI_INT,     ②
              0, MPI_COMM_WORLD);
}
FJMPI_Collection_stop();             ⑨
FJMPI_Collection_print                ⑩
    (const_cast<char *>("REGION_2"));
    (中略)
MPI_Finalize(); return 0;
}
```

■ 区間出力モードの例 [7/7]

- ジョブスクリプトにおいて環境変数OMPI_MCA_mpi_print_statsに値「3」または「4」を指定する。
- 本環境変数に値「3」を指定した場合、統計情報がランク0の標準エラー出力に書き出される。

ジョブスクリプト

```
#!/bin/bash
(中略)
export OMPI_MCA_mpi_print_stats=3
mpiexec -stdout-proc stdout -stderr-proc stderr ./a.exe
```

⚠ 値「3」を指定した場合、FJMPI_Collection_printは、集団通信ルーチンと同様に全プロセスが実行する必要がある。

- 本環境変数に値「4」を指定した場合、各プロセスごとの統計情報が、各プロセスの標準エラー出力に書き出される。

ジョブスクリプト

```
#!/bin/bash
(中略)
export OMPI_MCA_mpi_print_stats=4
mpiexec -stdout-proc stdout -stderr-proc stderr ./a.exe
```

- 値「4」を指定した場合、FJMPI_Collection_printを全プロセスが実行する必要はない。

以上で区間出力モードの例を終わる。

■ 区間指定MPI統計情報インターフェースの注意事項

- 本インターフェースを利用するには、MCAパラメータ`mpi_print_stats`に値「3」または「4」を指定する必要がある。
- 複数区間の指定、および入れ子について：
 - `FJMPI_Collection_start`と`FJMPI_Collection_stop`の組を繰り返して複数の区間に指定できる。その場合、測定値は累積される。ただし入れ子にはできない。
 - `FJMPI_Collection_clear`を指定するまで、測定区間の統計情報は累積される。
- `FJMPI_Collection_{start,stop,clear}`は、非集団的動作であり並列プロセス間で同期はとられない。
- `FJMPI_Collection_print`のプロセス間での動作について：
 - MCAパラメータ`mpi_print_stats`に値「3」を指定した場合（即ち、`MPI_COMM_WORLD`に属するランク0のプロセスがMPI統計情報を総括して出力すると指定した場合）、関数`FJMPI_Collection_print`は`MPI_COMM_WORLD`に属する全プロセスでの集団的動作となり、全プロセスが同時に呼び出す必要がある。正しく呼び出しを行わないと、デッドロックの原因となるので注意して下さい。
 - `mpi_print_stats`の値が「4」の場合の`FJMPI_Collection_print`は、非集団的動作であり並列プロセス間で同期はとられない。
- 各区間で同期をとって計測を開始したい場合は、`FJMPI_Collection_start`の直前に任意で`MPI_Barrier`を入れてもよい。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
 - 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル
 - MPI統計情報
 - 「富岳」の通信モード
 - 高速な集団通信
 - イーガー通信方式とランデブー通信方式
 - メモリ使用量を抑えるための考慮点
- 【付録】参考文献

■ 「富岳」のMPI通信

■ 「富岳」の通信モード

本節の内容

- ◇ 「富岳」では、MPIプログラムが実行されると、MPIライブラリ自身が必要とするメモリ（例えば受信バッファなど）が内部的に確保されます。これらのメモリは、通信相手のプロセスごとに確保される必要があります。「富岳」では、これらのメモリを通信相手の各プロセスと初めて通信を行う時点で確保するようにしています（この方式を動的コネクションと呼びます）。
- ◇ 通常、MPIプログラムの実行が進み通信相手のプロセスの数が増加すると、結果的にメモリ使用量も多くなっていきます。「富岳」では、その際のメモリ使用量をできるだけ抑えるために、高速型通信モードと省メモリ型通信モードという二つの通信モードを用意し、通信相手のプロセスごとにこれら二つの通信モードを内部的に使い分けられるようになっています。
- ◇ 本節では、高速型通信モードと省メモリ型通信モード、およびそれらの切り替え方法について説明します。

■ 高速型通信モードと省メモリ型通信モードの概要

■ 通信モードが切り替わるタイミング

■ 受信バッファサイズの変更

■ 補足事項

■ 「富岳」のMPI通信

■ 「富岳」の通信モード

本節の内容

■ 高速型通信モードと省メモリ型通信モードの概要

- 通信モードが切り替わるタイミング
- 受信バッファサイズの変更
- 補足事項

■ 高速型通信モードと省メモリ型通信モード

■ 高速型通信モード

- Large受信バッファと呼ぶ比較的大きな受信バッファと**Small**受信バッファと呼ぶ追加の受信バッファを通信相手プロセスごとに用意し、可能な限り高速に通信を行う。

■ 省メモリ型通信モード

- 以下の二つの方式が用意され、メモリ使用量ができるべく少なくなるように通信を行う。

- Medium受信バッファを使用する方式

- **Medium**受信バッファは、通信相手プロセスごとに用意される。
- 通信性能がやや落ちる代わりにメモリ使用量が抑えられる。

- Shared受信バッファを使用する方式

- 自プロセスに一つだけ用意され、（省メモリ型通信モードで通信する）相手プロセスで共有して使用される。
- この方式では受信バッファの合計メモリ使用量が通信相手プロセス数によらず一定になる。
- 通信相手プロセス数が特に多い場合は、メモリ使用量をより抑えることができる。ただし、通信性能は大きく落ちるため注意が必要。

■ 省メモリ型通信モードで使う方式の切り替え

省メモリ型通信モードで使う二つの方式（**Medium**受信バッファを使用する方式と**Shared**受信バッファを使用する方式）はMCAパラメータ`common_tofu_memory_saving_method`によって選択できる。

`-mca common_tofu_memory_saving_method 1`（省略値は1）

省メモリ型通信モードでの通信時に、**Medium**受信バッファを使用する方式を使う。

`-mca common_tofu_memory_saving_method 2`

省メモリ型通信モードでの通信時に、**Shared**受信バッファを使用する方式を使う。

■ 「富岳」のMPI通信

■ 「富岳」の通信モード

本節の内容

- 高速型通信モードと省メモリ型通信モードの概要

■ 通信モードが切り替わるタイミング

- 受信バッファサイズの変更
- 補足事項

■ 高速型通信モードに切り替わる通信回数の基準値

- 実際にどのプロセスと高速型通信モードで通信するかは、MPIプログラムの通信パターンによって実行時に決まる。
- 通常、最初はどのプロセスが相手でも省メモリ型通信モードで通信する。
- あるプロセスとの通信回数が基準値(*1)に達すると、（そのプロセスとの）それ以降の通信では高速型通信モードが使用される。

(*1) 省メモリ型通信モードから高速型通信モードに切り替わる通信回数の基準値は通常16に設定されている。この基準値はMCAパラメータ `common_tofu_fastmode_threshold` によって変更できる（省略値は16）。

`-mca common_tofu_fastmode_threshold 0`

（高速型通信モードで通信可能な相手プロセス数の上限に達していない限り）最初から高速型通信モードで通信を行う。

本パラメータの適用例を「イーガー通信方式とランデブー通信方式」節の「通信方式の変更」で示します。

■ 高速型通信モードで通信可能なプロセス数の上限

- 高速型通信モードで通信可能なプロセス数には上限(*2)が設定できる。高速型通信モードで通信する相手プロセスの数がこの上限に達した場合、それ以降高速型通信モードへの切り替えは行われない。

(*2) 通常、高速型通信モードで通信を行うことができるプロセス数の上限は256に設定されている。この上限値はMCAパラメータ`common_tofu_max_fastmode_procs`によって変更できる（省略値は256）。

`-mca common_tofu_max_fastmode_procs 0`

高速型通信モードを使用せず、全てのプロセスとの通信を省メモリ型通信モードで行う。

`-mca common_tofu_max_fastmode_procs -1`

全てのプロセスとの通信を高速型通信モードで行う。

■ 「富岳」のMPI通信

■ 「富岳」の通信モード

本節の内容

- 高速型通信モードと省メモリ型通信モードの概要
- 通信モードが切り替わるタイミング

■ 受信バッファサイズの変更

高速型通信モードと省メモリ型通信モードでの受信バッファのサイズはMCAパラメータによって変更できます。ここでは受信バッファのサイズを変更するMCAパラメータについて説明します。

- 補足事項

■ 高速型通信モードの受信バッファサイズの変更

高速型通信モードのLarge受信バッファのサイズは MCAパラメータ `common_tofu_large_recv_buf_size`によって変更できる。

`-mca common_tofu_large_recv_buf_size` 1024から16777216までの整数値
(省略値は1048576)

Large受信バッファのサイズ（バイト数）を指定する。

MCAパラメータ`common_tofu_use_memory_pool`の値が1の場合は、本パラメータの省略値が1048064に変更される。

※ Small受信バッファのサイズを変更するMCAパラメータは存在しない。

■ 省メモリ型通信モードの受信バッファサイズの変更

省メモリ型通信モードの二つの方式に対する受信バッファのサイズは以下のMCAパラメータによって変更できる。

■ Medium受信バッファを使用する方式

- Medium受信バッファのサイズはMCAパラメータ
common_tofu_medium_recv_buf_sizeによって変更できる。

-mca common_tofu_medium_recv_buf_size 256から16777216までの整数値
(省略値は2048)

Medium受信バッファのサイズ（バイト数）を指定する。

■ Shared受信バッファを使用する方式

- Shared受信バッファのサイズはMCAパラメータ
common_tofu_shared_recv_buf_sizeによって変更できる。

-mca common_tofu_shared_recv_buf_size 65536以上の整数値
(省略値は16777216)

Shared受信バッファのサイズ（バイト数）を指定する。指定された値が2のべき乗でない場合は2のべき乗の値へと切り上げられます。

■ 「富岳」のMPI通信

■ 「富岳」の通信モード

本節の内容

- 高速型通信モードと省メモリ型通信モードの概要
- 通信モードが切り替わるタイミング
- 受信バッファサイズの変更

■ 補足事項

後の節「メモリ使用量を抑えるための考慮点」との関連で、補足事項を記します。

■ 補足事項

- プロセス数が多い場合、必要メモリ量および求められる通信性能に応じて、下記の①～⑥のMCAパラメータ（既出）によるチューニングが重要となる。

① `common_tofu_max_fastmode_procs`

高速型通信モードで通信可能なプロセス数の上限の基準値

② `common_tofu_fastmode_threshold`

省メモリ型通信モードから高速型通信モードに切り替わる通信回数の基準値

③ `common_tofu_large_recv_buf_size`

高速型通信モードのLarge受信バッファのサイズ

④ `common_tofu_medium_recv_buf_size`

省メモリ型通信モードのMedium受信バッファのサイズ

⑤ `common_tofu_memory_saving_method`

省メモリ型通信モードで使う方式の変更

⑥ `common_tofu_shared_recv_buf_size`

省メモリ型通信モードのShared受信バッファのサイズ

詳細は「メモリ使用量を抑えるための考慮点」の節で説明します。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
 - 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル
 - MPI統計情報
 - 「富岳」の通信モード
 - 高速な集団通信
 - イーガー通信方式とランデブー通信方式
 - メモリ使用量を抑えるための考慮点
- 【付録】参考文献

■ 「富岳」のMPI通信

■ 高速な集団通信

本節の内容

「富岳」では、**Tofu**インターコネクトのハードウェアによりバリア通信という機能が提供されています。特定の**MPI**集団通信ルーチンは、バリア通信機能が適用される条件を満たせば、高速に通信を行うことができます。

本節では、バリア通信機能と、それが適用される**MPI**集団通信ルーチン、およびその他の機能について説明します。

- **Tofu**バリア通信のためのハードウェア
- **Tofu**バリア通信機能が適用される集団通信ルーチン
- ランク間で要素数が同じ場合の**MPI_BCAST**および**MPI_IBCAST**

■ 「富岳」のMPI通信

■ 高速な集団通信

本節の内容

「富岳」では、**Tofu**インターコネクトのハードウェアによりバリア通信という機能が提供されています。特定の**MPI**集団通信ルーチンは、バリア通信機能が適用される条件を満たせば、高速に通信を行うことができます。

本節では、バリア通信機能と、それが適用される**MPI**集団通信ルーチン、およびその他の機能について説明します。

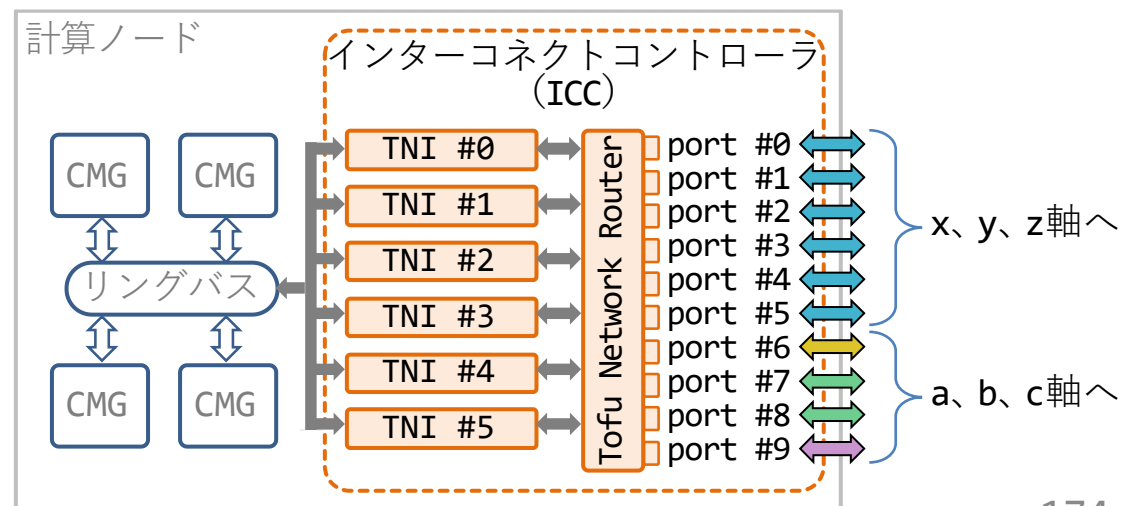
■ Tofuバリア通信のためのハードウェア

- Tofuバリア通信機能が適用される集団通信ルーチン
- ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

■ バリア通信のためのハードウェア [1/2]

- 各計算ノード内にはインターコネクト・コントローラ（Interconnect Controller、ICC）と呼ばれるLSIが存在する。ICCが他のノードとの通信を司っている。
- ICC内部には、6個のTofuネットワークインターフェース（Tofu Network Interface、TNI）というRDMAエンジン、10個のポート、およびTNIとポートの間にTofuネットワーク・ルーターが存在する。
- TNIはリングバスを通してノード内の4個のCMGと接続されている。各TNIは同時に1送信・1受信が可能であり、6個のTNIを合わせると、ICCは同時に6送信・6受信が可能である。
- 10個のポートはTofuインターコネクトにより、他のノードのポートと接続されている。

10個のポートのうち、6個のそれぞれは物理xyz軸のそれぞれの正負方向の送受信用、4個のそれぞれは物理abc軸方向（メッシュ結合のa軸とc軸にそれぞれ1個ずつ、トーラス結合のb軸の正負方向にそれぞれ1個ずつ）の送受信用である。



■ (次スライドへ続く)

■ バリア通信のためのハードウェア [2/2]

- バリア通信は**TNI**中に存在するバリアゲートと呼ばれるハードウェア・リソースがバリアネットワークを構成することにより行われる：
 - バリアゲートには、始点と終点の役目を果たす入力・出力ゲートと中継点の役目を果たす中継ゲートが存在する。
 - 各**TNI**には、入力・出力ゲート、中継ゲート合わせて**48**個存在する。
 - バリアネットワークを構成するには、各ノードに始点／終点用の**1**個のバリアゲートおよび中継点用の複数個のバリアゲートを確保する必要がある。これらのバリアゲートは、各ノードの同一**TNI**から確保される必要がある。
- 計算ノードごとの、**MPI**ライブラリで使用可能な最大ゲート数は、入力・出力ゲートが**96**（各ノードで**TNI**ごとに**16**個、最大総数**96**個まで利用できる）、中継ゲートが**192**（各ノードで**TNI**ごとに**32**個、最大総数**192**個まで利用できる）である（最大数はあくまでも目安である）。

■ 「富岳」のMPI通信

■ 高速な集団通信

本節の内容

■ Tofuバリア通信のためのハードウェア

■ Tofuバリア通信機能が適用される集団通信ルーチン

■ ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

■ 「富岳」のMPI通信

■ 高速な集団通信

■ Tofuバリア通信機能が適用される集団通信ルーチン

本小節の内容

- ◇ 「富岳」では、以下のMPI集団通信ルーチンについて、Tofuインターコネクタのハードウェア機能として提供されるバリア通信機能を利用し高速に通信を行うことができます。

MPI_BARRIER、MPI_BCAST、MPI_REDUCE、およびMPI_ALLREDUCE

- ◇ バリア通信機能はMPIルーチンごとに決められた条件を満たす場合に自動的に適用されます。
- ◇ バリア通信機能が適用されたかどうかはMPI統計情報で確認することができます。ここでは集団通信ルーチンごとの適用条件やMPI統計情報での確認の仕方について説明します。
- ◇ なお、適用条件には様々な項目が含まれるため、幾つかの細かな項目についてはルーチンごとの説明の後の【補足】で説明します。

■ MPI_BARRIER

■ MPI_BCAST

■ MPI_REDUCEおよびMPI_ALLREDUCE

■ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

■ 「富岳」のMPI通信

■ 高速な集団通信

■ Tofuバリア通信機能が適用される集団通信ルーチン

■ MPI_BARRIER

MPI_BARRIERに対してバリア通信機能が適用される条件の概略について説明します。MPI_BARRIERに対してバリア通信機能が適用されたかどうかをMPI統計情報で確認する仕方を具体例で示します。

■ MPI_BCAST

■ MPI_REDUCEおよびMPI_ALLREDUCE

■ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

■ MPI_BARRIERにバリア通信機能が適用される条件 [1/2]

以下の条件をすべて満たす場合にバリア通信機能が適用される。

- MCAパラメータ`coll`の値に`^tbi`が指定(*1)されていない。
- コミュニケータが、
グループ内コミュニケータ (`intra-communicator`) (*2)である、かつ、
`MPI_INTERCOMM_MERGE`ルーチンで作成されたものではない。
- (次スライドに続く)

(*1) `-mca coll ^tbi` の指定は、`MPI_BARRIER`、`MPI_BCAST`、`MPI_REDUCE`、および
`MPI_ALLREDUCE`ルーチンのいずれにおいても、バリア通信機能を適用しないことを意味する。

(*2) グループ内コミュニケータとは、`MPI_COMM_WORLD`から生成されたコミュニケータや、
`MPI_INTERCOMM_CREATE`以外のコミュニケータ生成ルーチンで生成されたコミュニケータを指す。

■ MPI_BARRIERにバリア通信機能が適用される条件 [2/2]

■ 以下の条件a.またはb.を満たす。

- 条件a. • MCAパラメータ`coll_tbi_intra_node_reduction(*3)`に指定された値が2である。
 - かつ、コミュニケータに割り当てられた計算ノード数が4以上である。
- 条件b. • `coll_tbi_intra_node_reduction(*3)`に指定された値が3または4である。
 - かつ、以下の条件1.または2.を満たす。
 - 条件1. • コミュニケータに属するプロセス数が4以上である。
 - かつ、コミュニケータに割り当てられた計算ノード数が3以下ならば、すべての計算ノードでノード内プロセス数が2以上である。
 - 条件2. • コミュニケータに属するプロセス数が4以上である。
 - かつ、コミュニケータに割り当てられた計算ノード数が4以上である。

■ バリアゲートの必要数(*4)が確保できる。

(*3) MCAパラメータ`coll_tbi_intra_node_reduction` (省略値: 3) は1ノード内に複数プロセスが割り当てられる場合に、バリア通信がノード内の浮動小数点型または複素数型データのリダクション演算で使用するアルゴリズムを指定する。(【補足】の節で説明します。)

(*4) バリアゲートとはバリア通信を行うためのハードウェア・リソースです (バリアゲートの必要数は【補足】の節で説明します。)

▶ MPI_BARRIER

■ MPI統計情報での表示例 [1/3]

- 以下のプログラム例の①のMPI_BARRIERについて、MPI統計情報を取得することとする。
「MPI統計情報」の節で説明した区間指定ルーチンで囲み（#include <mpi-ext.h>も指定し）、実行時にMCAパラメータmpi_print_statsに値「3」または「4」を指定（本例では「3」を指定）する。
- ジョブが終了すると、①のMPI_BARRIERにバリア通信機能が適用された場合、標準エラー出力内のMPI統計情報に、②に示す「Barrier Communication Count」のTofu欄にバリア通信機能を適用した回数が表示される。
- （次スライドにその他の説明続く。）

プログラム例

```
FJMPI_Collection_clear();  
FJMPI_Collection_start();  
MPI_Barrier(MPI_COMM_WORLD);  
FJMPI_Collection_stop();  
FJMPI_Collection_print(（省略）);
```

①

MPI統計情報

Barrier Communication Count				
	MAX		MIN	
				AVE
Tofu	1	[0]	1	1.0
Soft	0	[0]	0	0.0

②

Barrier Communication Count の各欄の説明	
Tofu	Tofuインターコネクトのバリア通信機能を利用したバリアの回数
Soft	ソフトウェア的に実行したバリアの回数

▶ MPI_BARRIER

■ MPI統計情報での表示例 [2/3]

- ①のMPI_BARRIERを実行した場合、MPI統計情報内の、下記図の③の「Collective Communication Information」のCOLLECTIVE欄に実行したブロッキング集団通信（本例ではMPI_BARRIER）に関する情報が表示される。

プログラム例（再掲）

```
FJMPI_Collection_clear();  
FJMPI_Collection_start();  
MPI_Barrier(MPI_COMM_WORLD);  
FJMPI_Collection_stop();  
FJMPI_Collection_print( (省略) );
```

①

MPI統計情報

----- Collective Communication Information -----								
COLLECTIVE	ALG	MSGMIN	MSGMAX		NODE	COMM	COUNT	AVE.TIME(MSEC)
Barrier	200	0	inf	2x	2x	2	32	1 0.072

③

ブロッキング集団通信ルーチンについて適用されたアルゴリズムの番号（次スライドで説明）。

ブロッキング集団通信ルーチンのメッセージサイズの範囲。

平均実行時間（単位はミリ秒）

▶ MPI_BARRIER

■ MPI統計情報での表示例 [3/3]

- **ALG**欄は各ブロッキング集団通信ルーチンについて適用されたアルゴリズムを番号で表している。番号の意味は下の表の通りである。
 - **MPI_BARRIER**にバリア通信機能が適用された以下の**MPI**統計情報表示例では**ALG**欄の番号が**200**となる。

MPI統計情報（再掲）

----- Collective Communication Information -----										③
COLLECTIVE	ALG	MSGMIN	-	MSGMAX		NODE	COMM	COUNT	AVE.TIME (MSEC)	
Barrier	200	0	-	inf	2x	2x	2	32	1 0.072	

ブロッキング集団通信ルーチンについて適用されたアルゴリズムの番号。

ALG欄の 番号	内容
1～6	Open MPIで公開されているアルゴリズム
100～103	Tofuインターコネクトや本システムのCPU向けにチューニングされたアルゴリズム
200	バリア通信
300	本システムで開発したアルゴリズム

以上でMPI_BARRIERについての説明を終わる。

■ 「富岳」のMPI通信

■ 高速な集団通信

■ Tofuバリア通信機能が適用される集団通信ルーチン

■ MPI_BARRIER

■ MPI_BCAST

MPI_BCASTに対してバリア通信機能が適用される条件の概略について説明します。MPI_BCASTに対してバリア通信機能が適用されたかどうかをMPI統計情報で確認する仕方を具体例で示します。

■ MPI_REDUCEおよびMPI_ALLREDUCE

■ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

■ MPI_BCASTにバリア通信機能が適用される条件 [1/2]

以下の条件をすべて満たす場合にバリア通信機能が適用される。

■ 「MPI_BARRIERにバリア通信機能が適用される条件」と同じ条件を満たす。

■ MCAパラメータ`coll_tbi_use_on_bcast(*5)`の値に0が指定されていない。

(*5) MCAパラメータ`coll_tbi_use_on_bcast`はMPI_BCASTルーチンにおいてバリア通信機能を適用するかどうかを指定する。

`-mca coll_tbi_use_on_bcast 1` (本パラメータの省略値は1)
バリア通信機能を適用することを指定する。

`-mca coll_tbi_use_on_bcast 0`
バリア通信機能を適用しないことを指定する。

本パラメータの説明と注意事項を条件の説明の後に説明します。

■ (次スライドに続く)

■ MPI_BCASTにバリア通信機能が適用される条件 [2/2]

■ メッセージの要素数が以下の全てを満たす。

- データ型、1要素あたりのバイト数、およびメッセージの要素数上限の組み合わせが、以下の表のどれかである。

データ型	1要素あたりのバイト数	メッセージの要素数上限
整数型 浮動小数点型 論理型	8バイト以内	6（各要素が8バイトの場合）
		12（各要素が4バイトの場合）
		24（各要素が2バイトの場合）
		48（各要素が1バイトの場合）
複素数型	4バイト（2バイト × 2）	12（各要素が4バイトの場合）
	または 8バイト（4バイト × 2）	または 6（各要素が8バイトの場合）
	または 16バイト（8バイト × 2）	または 3（各要素が16バイトの場合）
バイト型	1バイト	48
多言語型	8バイト	6

- 基本データ型ではないデータ型を利用する場合、メッセージの要素数は構成要素にした基本データ型の要素数が上の表におけるメッセージの要素数上限を超えない範囲である。
- MCAパラメータ`coll_tbi_repeat_max(*6)`と組み合わせた場合、メッセージの要素数はメッセージの要素数上限を`coll_tbi_repeat_max`倍した値を超えない範囲である。
(*6) 後の項「MPI_BCASTによる通信方法の例」で説明します。

以上でMPI_BCASTに対するバリア通信機能適用条件概略の説明を終わる。

■ MCAパラメータcoll_tbi_use_on_bcastの説明

本MCAパラメータはMPI_BCASTルーチンにおいてバリア通信機能を適用するかどうかを指定する。

`-mca coll_tbi_use_on_bcast 1` （本パラメータの省略値は1）

バリア通信機能を適用することを指定する。

⚠ 型仕様（次スライドで説明）が異なるMPI通信に適用するとデッドロックを起こす可能性がある(*1)。

`-mca coll_tbi_use_on_bcast 0`

バリア通信機能を適用しないことを指定する。

- 型仕様が異なるMPI通信に本指定をすることができる(*1)。

(*1) 次のスライドで説明します。

次スライドに本パラメータに関する注意事項を説明します。

■ MCAパラメータcoll_tbi_use_on_bcastの注意事項

- ⚠ 下図のプログラム例では、①のMPI_BCASTでランク0を送信元として4バイトのunsigned int型データを1個送信している。②のMPI_BCASTで0以外のランクは4バイトのfloat型データとして受信している。送受信するデータのバイト数は同じだが、送信側と受信側で基本データ型が異なっている。基本データ型の並びをMPI規格では型仕様 (type signature) という。型仕様が異なる場合、MPI規格ではプログラムの誤りであるとされる。

型仕様が異なる
プログラム例

```
if (myrank == 0){  
    unsigned int buf0 = 987654321U;  
    MPI_Bcast(&buf0, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD); ①  
}else{  
    float buf1;  
    MPI_Bcast(&buf1, 1, MPI_FLOAT, 0, MPI_COMM_WORLD); ②  
}
```

- ⚠ 「富岳」では型仕様が異なるメッセージに対してバリア通信を行うと、デッドロックや異常終了することがある。型仕様が異なるプログラムは本来MPIプログラムとしては正しくないが、MCAパラメータcoll_tbi_use_on_bcastの値を0に設定することでMPI_BCASTルーチンにおいてバリア通信機能を適用しないように指定して、実行することができる。（あるいはMCAパラメータcollの値を^tbiに設定することでMPIルーチンに関係なくバリア通信機能の適用を無効化することもできる。）

▶ MPI_BCAST

■ MPI統計情報での表示例 [1/3]

- 以下のプログラム例の①のMPI_BCASTについて、MPI統計情報を取得することとする。
「MPI統計情報」の章で説明した区間指定ルーチンで囲み（#include <mpi-ext.h>も指定し）、実行時にMCAパラメータmpi_print_statsに値「3」または「4」を指定（本例では「3」を指定）する。

プログラム例

```
double A[1] = {};  
FJMPI_Collection_clear();  
FJMPI_Collection_start();  
MPI_Bcast(&A[0], 1, MPI_DOUBLE, 0, MPI_COMM_WORLD); ①  
FJMPI_Collection_stop();  
FJMPI_Collection_print(（省略）);
```

- ジョブが終了すると、①のMPI_BCASTにバリア通信機能が適用された場合、標準エラー出力内のMPI統計情報に、②に示す「Tofu Barrier Collective Communication Count」のBcast欄にMPI_BCASTにバリア通信機能を適用した回数が表示される。

MPI統計情報

----- Tofu Barrier Collective Communication Count -----					②
	MAX		MIN		AVE
Bcast	1	[0]	1	[0]	1.0
Reduce	0	[0]	0	[0]	0.0
Allreduce	0	[0]	0	[0]	0.0

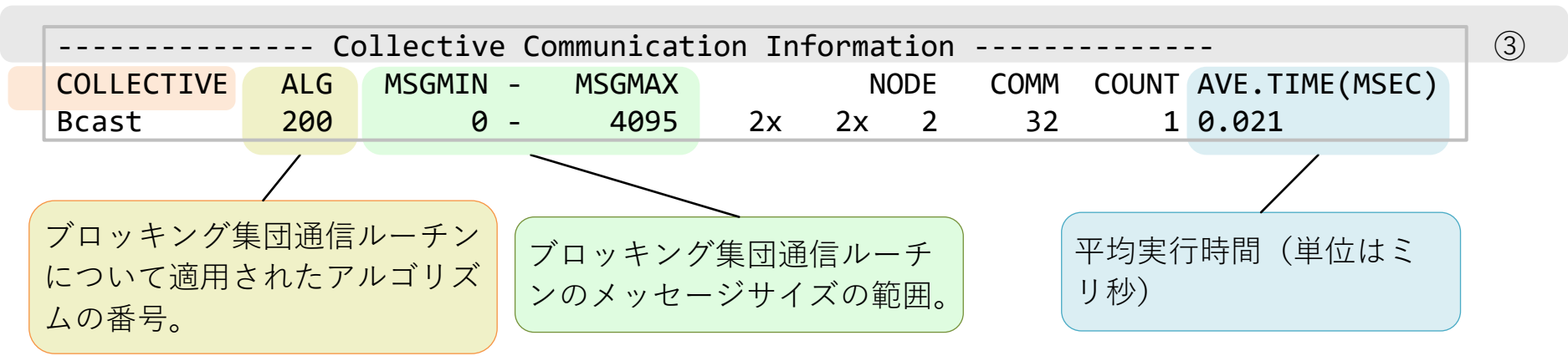
- なお、本例ではMPI_REDUCEおよびMPI_ALLREDUCEルーチンは計測対象区間に使用していないが、その場合でもReduce欄およびAllreduce欄が表示され、回数は0回と表示される。

▶ MPI_BCAST

■ MPI統計情報での表示例 [2/3]

- また、MPI統計情報内の、下記図の③「Collective Communication Information」のCOLLECTIVE欄に実行したブロッキング集団通信（本例ではMPI_BCAST）に関する情報が表示される。
 - MPI_BCASTにバリア通信機能が適用された以下のMPI統計情報表示例ではALG欄の番号が200となる。

MPI統計情報



ALG欄の 番号	内容
1～6	Open MPIで公開されているアルゴリズム
100～103	Tofuインターコネクトや本システムのCPU向けにチューニングされたアルゴリズム
200	バリア通信
300	本システムで開発したアルゴリズム

▶ MPI_BCAST

■ MPI統計情報での表示例 [3/3]

- MPI_BCASTにバリア通信機能が適用されなかった場合、MPI統計情報内の
 - ④「Tofu-specific Collective Communication Count」または
 - ⑤「Non-Tofu-specific Collective Communication Count」にその回数が表示される。④にはTofuインターコネクトや本システムのCPU向けにチューニングされたアルゴリズムが適用された回数が表示され、⑤にはそれらのアルゴリズムが適用されなかった回数が表示される。
- 前スライドのプログラム例でバリア通信機能が適用されなかった場合の以下の表示例では⑤の方に回数が計上されている。

MPI統計情報

----- Tofu Barrier Collective Communication Count -----				
	MAX	MIN	AVE	
Bcast	0 [0]	0 [0]	0.0	
(中略)				
----- Tofu-specific Collective Communication Count -----				④
	MAX	MIN	AVE	
Bcast	0 [0]	0 [0]	0.0	
(中略)				
----- Non-Tofu-specific Collective Communication Count -----				⑤
	MAX	MIN	AVE	
Bcast	1 [0]	1 [0]	1.0	

以上でMPI_BCASTについての統計情報での表示例の説明を終わる。

■ MPI_BCASTによる通信方法の例

ここでは、MPI_BCASTで複数の要素を通信する際、通信時間の観点で、通信の仕方について留意すると良い点について説明します。

「富岳」に限らず一般的に、MPIルーチンによる通信の時間はルーチンの立ち上がり時間と通信自体の時間の合計となります。通信を1回行うごとに、一定の立上り時間がかかります。従って、必要な全体としての通信量が同じならば通信する回数はなるべく少なくするというのが基本的指針です。これは1回で通信する量（例えば配列の要素数）を多くするという指針になります。

「富岳」のMPI集団通信ルーチンに対しても一般論としてはそうです。しかし、「富岳」でMPI_BCAST、MPI_REDUCE、MPI_ALLREDUCEルーチンで通信する場合、バリア通信機能が適用できる要素数には上限があるため、1回で通信する量がこの上限を超えて通信すると、バリア通信機能が適用されなくなります。

ここでは、そのような場合の例を示します。改善策として二つの方法を例で示します。一つ目は、通信する回数は増えるけれども、1回で通信する量を少なくし、バリア通信機能適用の要素数の上限以下に納まるようにする方法です。二つ目は、通信する回数はそのままで、バリア通信機能適用の要素数の上限を大きくするようMCAパラメータcoll_tbi_repeat_maxを設定する方法です。これらの方法の例によりバリア通信機能が適用されるようになり、通信時間が短くなることを確認します。（あくまでも可能性を示すものです。実行時のプロセス数などにより、これらの方法が常にうまく行くとは限りません。ケースバイケースで有効性を検討して下さい。）

▶ MPI_BCAST

■ MPI_BCASTによる通信方法の例：方法1 複数回に分けて通信する [1/2]

- 以下のプログラム1では、forループ内の1回のMPI_BCASTでdouble型の配列の12個の要素を通信している。
- プログラム2では、forループ内で12個を6個ずつ2回のMPI_BCASTに分けて通信している。
- 以下のプログラム1と2でforループ内の1反復で通信データ量は同じである。
- これらのBCASTの通信に要した経過時間を計測するとする。forループは経過時間計測のためのものである。
 - MPI統計情報の区間指定ルーチンにより、MPI統計情報を取得するとする。以下の図では区間指定ルーチンの記載は省略する。

プログラム1と2に共通：

```
double A[12] = {};
```

プログラム1：

```
for (int i=0; i<100000; i++){  
    MPI_Bcast(&A[0], 12, MPI_DOUBLE, 0, ~);  
}
```

プログラム2：

```
for (int i=0; i<100000; i++){  
    MPI_Bcast(&A[0], 6, MPI_DOUBLE, 0, ~);  
    MPI_Bcast(&A[6], 6, MPI_DOUBLE, 0, ~);  
}
```

- (次スライドへ続く)

▶ MPI_BCAST

■ MPI_BCASTによる通信方法の例：方法1 複数回に分けて通信する [2/2]

- MPI_BCASTで通信するメッセージの各要素がdouble型 8 バイトの場合、バリア通信が適用される要素数上限は 6 である。
- プログラム1では 12 個を指定していて要素数上限 6 を超えるのでバリア通信が適用されない。
- プログラム2では 6 個を指定しているので、要素数上限 6 以内でありバリア通信が適用される。

プログラム1と2に共通：

```
double A[12] = {};
```

プログラム1：

```
for (int i=0; i<100000; i++){  
    MPI_Bcast(&A[0], 12, MPI_DOUBLE, 0, ~);  
}
```

プログラム2：

```
for (int i=0; i<100000; i++){  
    MPI_Bcast(&A[0], 6, MPI_DOUBLE, 0, ~);  
    MPI_Bcast(&A[6], 6, MPI_DOUBLE, 0, ~);  
}
```

- ノード形状2x2x2、プロセス数32で実行した場合、forループの経過時間は以下の結果となった。

プログラム1：

0.79秒

>

プログラム2：

0.66秒

- A[0～11]の12要素を一度に通信するよりも、A[0～5]とA[6～11]の2回に分けて通信した方が速い可能性がある。

▶ MPI_BCAST

■ MPI_BCASTによる通信方法の例：方法2 バリア通信適用要素数上限を上げる

- バリア通信が適用される要素数の上限はMCAパラメータcoll_tbi_repeat_maxで変更することができる。

-mca coll_tbi_repeat_max 1以上の整数値 （本パラメータ無指定時の値は1）

- 本MCAパラメータに値 n を指定した場合、MPI_BCAST、MPI_REDUCE、およびMPI_ALLREDUCEルーチンにおいて、バリア通信が適用される要素数の上限が n 倍になる。
- MPI_BCASTで通信するメッセージの各要素がdouble型 8 バイトの場合、バリア通信が適用される要素数上限は（本MCAパラメータ無指定時は） 6 であるが、本MCAパラメータに2を指定すると上限は2倍された値 12 になる。この指定をして実行したプログラム1はバリア通信が適用されることになる。

プログラム1：

```
for (int i=0; i<100000; i++){  
    MPI_Bcast(&A[0], 12, MPI_DOUBLE, 0, ~);  
}
```

プログラム2：

```
for (int i=0; i<100000; i++){  
    MPI_Bcast(&A[0], 6, MPI_DOUBLE, 0, ~);  
    MPI_Bcast(&A[6], 6, MPI_DOUBLE, 0, ~);  
}
```

coll_tbi_repeat_maxの指定無し
プログラム1：
0.79秒

バリア通信適用要素数上限：6

バリア通信適用されない



coll_tbi_repeat_maxに値2の指定あり
バリア通信適用要素数上限：12
バリア通信適用される

0.62秒

プログラム2：

0.66秒

- プログラム1でcoll_tbi_repeat_maxの値2の指定により、無指定時0.79秒よりも速くなった。
- また、2回に分けてBCASTするプログラム2の0.66秒よりも若干速くなった。

■ 「富岳」のMPI通信

■ 高速な集団通信

■ Tofuバリア通信機能が適用される集団通信ルーチン

■ MPI_BARRIER

■ MPI_BCAST

■ MPI_REDUCEおよびMPI_ALLREDUCE

MPI_REDUCEおよびMPI_ALLREDUCEに対してバリア通信機能が適用される条件の概略について説明します。MPI_REDUCEおよびMPI_ALLREDUCEに対してバリア通信機能が適用されたかどうかをMPI統計情報で確認する仕方を具体例で示します。

■ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

「富岳」のMPI通信 ▶ 高速な集団通信 ▶ Tofuバリア通信機能が適用される集団通信ルーチン
▶ MPI_REDUCEおよびMPI_ALLREDUCE

■ MPI_REDUCEおよびMPI_ALLREDUCEにバリア通信機能が適用される条件 [1/3]

以下の条件をすべて満たす場合にバリア通信機能が適用される。

- 「MPI_BARRIERにバリア通信機能が適用される条件」と同じ条件を満たす。
- MCAパラメータ`coll_base_reduce_commute_safe(*7)`によるリダクション演算の順序を保証する指定がない。
- メッセージの要素数が以下の全てを満たす。
 - リダクション演算の演算、データ型、1要素あたりのバイト数、およびメッセージの要素数上限の組み合わせが以降の2ページに渡って記載するの表のどれかである。
 - MCAパラメータ`coll_tbi_repeat_max(*6)`と組み合わせた場合、メッセージの要素数は、メッセージの要素数上限を `coll_tbi_repeat_max` 倍した値を超えない範囲である。

(*6) MPI_BCASTに対するMPI統計情報の表示例の後の小節を参照して下さい。

(*7) MCAパラメータ`coll_base_reduce_commute_safe`は、MPI_REDUCE、MPI_ALLREDUCE、およびその他のリダクション演算をする集団通信について、リダクション演算の順序を保証するかどうかを指定するパラメータである。詳細は後の【補足】を参照して下さい。本パラメータの省略値は0（リダクション演算の順序を保証しない）である。

■ （次のスライドにつづく）

▶ MPI_REDUCEおよびMPI_ALLREDUCE

■ MPI_REDUCEおよびMPI_ALLREDUCEにバリア通信機能が適用される条件 [2/3]

■ リダクション演算の演算、データ型、1要素あたりのバイト数、およびメッセージの要素数上限の組み合わせが以下の表のどれかである。

⚠ リダクション演算がMPI_SUMの場合、1要素あたりのバイト数が整数型と浮動小数点型とで共に8バイトであるが、メッセージの要素数上限は整数型に対しては6、浮動小数点型は3であることに注意して下さい。

MPI定義済み演算 [C/C++/Fortran]	データ型	1要素あたりのバイト数	メッセージの 要素数上限
MPI_MAX MPI_MIN	整数型	8バイト以内	6
	浮動小数点型(*2)	8バイト以内	6
MPI_SUM	多言語型	8バイト	6
	整数型	8バイト以内	6
	浮動小数点型	8バイト以内	3
	複素数型	4バイト (2バイト × 2) または 8バイト (4バイト × 2) または 16バイト (8バイト × 2)	1
	多言語型	8バイト	6
(次のスライドへ続く)			

※ 演算の型及びデータ型のC++インターフェースは省略した。

(*2) MCAパラメータcoll_tbi_use_on_max_minに値1を指定する必要がある。本MCAパラメータの省略値は1 (MPI_REDUCEおよびMPI_ALLREDUCEルーチンの浮動小数点型MPI_MAX/MPI_MIN演算において、バリア通信機能を適用することを指定する) である。

「富岳」のMPI通信 ▶ 高速な集団通信 ▶ Tofuバリア通信機能が適用される集団通信ルーチン
 ▶ MPI_REDUCEおよびMPI_ALLREDUCE

■ MPI_REDUCEおよびMPI_ALLREDUCEにバリア通信機能が適用される条件 [3/3]

(前のスライドから続き)			
MPI定義済み演算 [C/C++/Fortran]	データ型	1要素あたりのバイト数	メッセージの 要素数上限
MPI_LAND MPI_LOR MPI_LXOR	整数型 論理型	8バイト以内	384
MPI_BAND MPI BOR MPI_BXOR	整数型	8バイト以内	6 (各要素が8バイトの場合) 12 (各要素が4バイトの場合) 24 (各要素が2バイトの場合) 48 (各要素が1バイトの場合)
	バイト型	1バイト	48
	多言語型	8バイト	6
MPI_MAXLOC MPI_MINLOC	[C/C++] MPI_2INT MPI_LONG_INT MPI_SHORT_INT [Fortran] MPI_2INTEGER	8バイト (4バイト + 4バイト) 12バイト (8バイト + 4バイト) 6バイト (2バイト + 4バイト)	3

※ 演算の型及びデータ型のC++インターフェースは省略した。

■ MPI統計情報での表示例 [1/2]：バリア通信機能が適用される場合

- 以下のプログラム例の①のMPI_ALLREDUCEについて、MPI統計情報を取得することとする。
「MPI統計情報」の章で説明した区間指定ルーチンで①の文を囲み（`#include <mpi-ext.h>`も指定し）、実行時にMCAパラメータ`mpi_print_stats`に値「3」または「4」を指定（本例では「3」を指定）する。

プログラム例

```
double A3[3] = {1.0, 2.0, 3.0};  
double B3[3];  
MPI_Allreduce(&A3[0], &B3[0], 3,  
              MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD); ①
```

- 上のプログラム例では①のMPI_ALLREDUCEでdouble型8バイトの要素を 3 個指定し、リダクション演算にMPI_SUMを指定している。
- 前小節で記したように、MPI_SUMで浮動小数点型の場合、バリア通信機能が適用される要素数の上限は 3 である。従って本例の場合はバリア通信機能が適用される。
- MPI_ALLREDUCEにバリア通信機能が適用された場合、MPI統計情報内の、②に示す「Tofu Barrier Collective Communication Count」のAllreduce欄にバリア通信機能を適用した回数が表示される。本例では実際適用されていることが確認される。

MPI統計情報

----- Tofu Barrier Collective Communication Count -----					②
	MAX		MIN		AVE
Bcast	0	[0]	0	[0]	0.0
Reduce	0	[0]	0	[0]	0.0
Allreduce	1	[0]	1	[0]	1.0

■ MPI統計情報での表示例 [2/2]：バリア通信機能が適用されない場合

- 以下のプログラム例では③のMPI_ALLREDUCEにMPI_SUMで倍精度実数 6 要素を指定している。
- MPI_SUMで浮動小数点型の場合バリア通信機能が適用される要素数の上限は 3 であるため、本例ではバリア通信機能が適用されない。

プログラム例

```
double A6[6] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};  
double B6[6];  
MPI_Allreduce(&A6[0], &B6[0], 6,  
              MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

③

- MPI_ALLREDUCEにバリア通信機能が適用されなかった場合、MPI統計情報内の
④「Tofu-specific Collective Communication Count」または
⑤「Non-Tofu-specific Collective Communication Count」
にその回数が表示される。④にはTofuインターコネクトや本システムのCPU向けにチューニングされたアルゴリズムが適用された回数が表示され、⑤にはそれらのアルゴリズムが適用されなかった回数が表示される。
- 上のプログラム例の場合、⑤の方に回数が計上されている。

MPI統計情報

----- Tofu-specific Collective Communication Count -----				④
	MAX	MIN	AVE	
(中略)				
Allreduce	0 [0]	0 [0]	0.0	
(中略)				
----- Non-Tofu-specific Collective Communication Count -----				⑤
	MAX	MIN	AVE	
(中略)				
Allreduce	1 [0]	1 [0]	1.0	

「富岳」のMPI通信 ▶ 高速な集団通信 ▶ Tofuバリア通信機能が適用される集団通信ルーチン
▶ MPI_REDUCEおよびMPI_ALLREDUCE

■ MPI_ALLREDUCEによる通信方法の例

MPI統計情報での表示例での二つの例で見たように、実数型のMPI_SUMのALLREDUCEで複数要素を通信する場合、バリア通信機能は、3要素では適用され、6要素では適用されませんでした。ここでは、「MPI_BCASTによる通信方法の例」で行ったように、バリア通信機能を適用させるための二通りの方法の検討結果を示します。

まず、6要素を3要素ずつ二つに分けて通信する方法の検討結果を示します。

次に6要素のままでMCAパラメータcoll_tbi_repeat_maxでバリア通信機能適用要素数の上限を上げる方法の検討結果を示します。

（なお、ここでの結果は、あくまでも可能性を示すものです。実行時のプロセス数などにより、これらの方法が常にうまく行くとは限りません。ケースバイケースで有効性を検討して下さい。）

▶ MPI_REDUCEおよびMPI_ALLREDUCE

■ MPI_ALLREDUCEによる通信方法の例

- 6要素一度に通信（プログラム1）
 - 先のプログラム例で見たように、実数型のMPI_SUMのALLREDUCEのバリア通信機能適用の要素数上限3を超えるため、バリア通信機能は適用されない。
- 3要素ずつ二つに分けて通信（プログラム2）
 - 先のプログラム例で見たようにバリア通信機能が適用される。
- 6要素一度に通信（プログラム1）し、coll_tbi_repeat_maxに値2を指定する
 - バリア通信機能適用の要素数上限が6となり、バリア通信機能が適用される。

※ FJMPI_Collection_{clear,start,stop,print}と#include <mpi-ext.h>の記載は省略する。

```
for (int i=0; i<NLOOP; i++){  
    MPI_Allreduce(&A[0], &B[0], 6,  
                  MPI_DOUBLE,MPI_SUM, (省略));  
}
```

プログラム1

```
for (int i=0; i<NLOOP; i++){  
    MPI_Allreduce(&A[0], &B[0], 3,  
                  MPI_DOUBLE, MPI_SUM, (省略));  
    MPI_Allreduce(&A[3], &B[3], 3,  
                  MPI_DOUBLE, MPI_SUM, (省略));  
}
```

プログラム2

バージョン	使用プログラム	要素数の指定方法	coll_tbi_repeat_maxの指定	バリア通信機能適用の有無	経過時間 (単位：秒) (*1)	【参考】 ALG欄の 番号(*2)
asis	プログラム1	6要素一度に通信	指定無し	適用されない	1.31	3
tune1	プログラム2	3要素ずつ 二つに分けて通信	指定無し	適用される	0.61	200
tune2	プログラム1	6要素一度に通信	値2を指定する	適用される	0.53	200

(*1) ノード形状2x2x2、プロセス数32、反復回数NLOOP=100000を指定した場合の累積時間。
(*2) 本例に対する結果。「Collective Communication Information」のALG欄の番号。
ALLREDUCEに対する番号「3」はOpen MPIで公開されているアルゴリズム (recursive doubling)、番号「200」はバリア通信を意味する。

■ 「富岳」のMPI通信

■ 高速な集団通信

■ Tofuバリア通信機能が適用される集団通信ルーチン

■ MPI_BARRIER

■ MPI_BCAST

■ MPI_REDUCEおよびMPI_ALLREDUCE

■ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

ここでは、バリア通信機能の適用条件に関与する以下の項目について説明します。

■ MCAパラメータcoll_tbi_intra_node_reduction

■ バリア通信を行うためのバリアゲートの必要数の目安

■ MCAパラメータcoll_base_reduce_commute_safe

- 「富岳」のMPI通信 ▶ 高速な集団通信 ▶ Tofuバリア通信機能が適用される集団通信ルーチン
- ▶ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

■ MCAパラメータ `coll_tbi_intra_node_reduction`

1 ノード内に複数プロセスが割り当てられる場合に、バリア通信がノード内の浮動小数点型または複素数型データのリダクション演算で使用するアルゴリズムを指定する。

`-mca coll_tbi_intra_node_reduction 2`

MPI_REDUCEおよびMPI_ALLREDUCEに対するバリア通信の適用条件に該当する場合に、ノード内のリダクション演算でバリア通信を使用しない。代わりにソフトウェアによる`recursive_doubling`アルゴリズムを使用する。

`-mca coll_tbi_intra_node_reduction 3`（本パラメータの省略値）

ノード内のリダクション演算でバリア通信を使用する。必要なバリアゲートの数を節約できる`binary_tree`アルゴリズムを使用する。

`-mca coll_tbi_intra_node_reduction 4`

ノード内のリダクション演算でバリア通信を使用する。本パラメータに3を指定した場合より多くのバリアゲート数を必要とするが、高速化が期待される`recursive_doubling`アルゴリズムを使用する。

- 「富岳」のMPI通信 ▶ 高速な集団通信 ▶ Tofuバリア通信機能が適用される集団通信ルーチン
- ▶ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

■ バリア通信を行うためのバリアゲートの必要数の目安

MPI_BARRIER、MPI_BCAST、MPI_REDUCE、およびMPI_ALLREDUCEルーチンでバリア通信が適用される場合の、中継点用バリアゲートの必要数の目安は以下のようになる。

MCAパラメータ coll_tbi_intra_node_reduction の値	各ノードでの中継点用バリアゲートの 使用数の目安
2 の場合：	最大で $\log_2 N$ 個程度
3 の場合：	最大で $(\log_2 N) + (3P - 3)$ 個程度
4 の場合：	最大で $(\log_2 N + \log_2 P) \times P$ 個程度

本パラメータの意味については前スライドを参照
して下さい

N: 1回のバリア通信において使用するノード数
P: 1ノード内のプロセス数

- 「富岳」のMPI通信 ▶ 高速な集団通信 ▶ Tofuバリア通信機能が適用される集団通信ルーチン
- ▶ 【補足】バリア通信機能が適用される条件に関連する幾つかの項目について

■ MCAパラメータ `coll_base_reduce_commute_safe`

リダクション演算を行う以下の集団通信ルーチンにおいて、リダクション演算の順序を保証するかどうかを指定する。

MPI_REDUCE、MPI_IREDUCE、
MPI_ALLREDUCE、MPI_IALLREDUCE、
MPI_REDUCE_SCATTER、MPI_IREDUCE_SCATTER、
MPI_REDUCE_SCATTER_BLOCK、MPI_IREDUCE_SCATTER_BLOCK、および
MPI_SCAN

`-mca coll_base_reduce_commute_safe 0`（本パラメータの省略値は0）

リダクション演算の順序を保証しない。各プロセスの通信順序は通信条件に応じて、通信時間になるべく短くなるように、最適化される。リダクション演算の順序が変わることで、結果的に計算結果の精度にも影響を与えることがある。

`-mca coll_base_reduce_commute_safe 1`

リダクション演算の順序を保証する。通信の最適化が行われないため、通信時間が長くなる可能性がある。

■ 「富岳」のMPI通信

■ 高速な集団通信

本節の内容

- Tofuバリア通信のためのハードウェア
- Tofuバリア通信機能が適用される集団通信ルーチン

■ ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

「富岳」では、MPI_BCASTまたはMPI_IBCASTルーチンにおいて、ランク間の要素数が同じである場合、MCAパラメータ `coll_tuned_bcast_same_count` に値1を指定して通信の高速化を実現する機能が提供されています。MPI使用手引書ではこの機能に名称は付けられていません。本講義ではこの機能を「MPI_BCASTの高速アルゴリズム」と呼ぶことにします。

本小節では、MCAパラメータ `coll_tuned_bcast_same_count` と、指定する際の注意事項を説明します。

「富岳」のMPI通信 ▶ 高速な集団通信

▶ ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

■ 本機能を使用するためのMCAパラメータ

「富岳」では、MPI_BCASTルーチンまたはMPI_IBCASTルーチンにおいて、ランク間の要素数が同じである場合、通信の高速化を実現する機能を用意している。この機能は、MCAパラメータcoll_tuned_bcast_same_countに値1を指定することで利用できる。

■ MCAパラメータcoll_tuned_bcast_same_count

MPI_BCASTまたはMPI_IBCASTルーチンにおいて、ランク間で同じ要素数を用いた通信において、通信を高速化する機能(*1)を用いるかどうかを指定する。

(*1) この機能を、本講義では「MPI_BCASTの高速アルゴリズム」と呼ぶことにする。

```
-mca coll_tuned_bcast_same_count 1
```

「MPI_BCASTの高速アルゴリズム」を使用することを指定する。

```
-mca coll_tuned_bcast_same_count 0
```

（本パラメータの省略値）

「MPI_BCASTの高速アルゴリズム」を使用しないことを指定する。

⚠ 本MCAパラメータは、次のスライドで述べる理由により、省略値は0に設定されている。

▶ ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

■ 本機能のMCAパラメータのデフォルト

■ MPI_BCASTまたはMPI_IBCASTルーチンについて、引数に指定したデータ型と要素数がランク間で異なる場合でも、MPIの規格では基本データ型の並びである型仕様 (type signature) が一致していれば正しいプログラムである。

- 例えば、以下のプログラム例では、ランク0は①のMPI_BCASTで、MPI_INTを2個、それ以外のランクは②で2個のMPI_INTから成る派生データ型INT_2を1個指定している。①と②でデータ型と要素数が異なるが、基本データ型の並び（本例ではMPI_INTの並び）が一致しているので、正しいMPIプログラムである。

プログラム例

```
MPI_Datatype INT_2;  
MPI_Type_contiguous(2, MPI_INT, &INT_2);  
MPI_Type_commit(&INT_2);  
if (myrank == 0){  
    int A[2] = {1, 2};  
    MPI_Bcast(&A[0], 2, MPI_INT, 0, MPI_COMM_WORLD); ①  
}else{  
    int A[2];  
    MPI_Bcast(&A[0], 1, INT_2, 0, MPI_COMM_WORLD); ②  
}
```

■ 上記のようなMPIプログラムに対して「MPI_BCASTの高速アルゴリズム」を適用すると、MPIライブラリ内で不整合が生じ誤動作する可能性がある。このためデフォルトでは、「MPI_BCASTの高速アルゴリズム」を使用しないように設定されている。この理由で、MCAパラメータcoll_tuned_bcast_same_countの省略値は0に設定されている。

▶ ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

■ 本機能を使用する場合の注意事項 [1/2]

■ プログラム内で使用している各MPI_BCASTやMPI_IBCASTルーチンにおいて、ランク間の要素数が同じであることが保証されている場合は、MCAパラメータ `coll_tuned_bcast_same_count` に1を指定して「MPI_BCASTの高速アルゴリズム」を使用することができる。

- バリア通信機能またはTofuインターコネクト向けにチューニングされたアルゴリズムが適用された場合でも、上記MCAパラメータを指定することができる。
- 上記MCAパラメータを指定しても速くならない場合や、却って遅くなる場合もある。
- MPI統計情報には、「MPI_BCASTの高速アルゴリズム」が使用されたことを示す項目はない。

■ (次スライドへ続く)

▶ ランク間で要素数が同じ場合のMPI_BCASTおよびMPI_IBCAST

■ 本機能を使用する場合の注意事項 [2/2]

- MPI_ALLGATHERおよびMPI_ALLGATHERVルーチンには、内部でMPI_BCASTルーチンの処理を呼ぶアルゴリズムが存在する。このため、`coll_tuned_bcast_same_count`に値1を指定した場合、MPI_ALLGATHERやMPI_ALLGATHERVでは以下の点に注意する必要がある。

- ◆ MPI_ALLGATHERの場合：送信側と受信側で要素数を同じにする。

- ◆ MPI_ALLGATHERVの場合：受信側の要素数の配列を全てのランク間で同じにする。

【引数の指定例】Cでの構文

```
int MPI_Allgatherv(  
    const void *sendbuf,          int sendcount,          MPI_Datatype sendtype,  
    void *recvbuf, const int *recvcounts, const int *displs, MPI_Datatype recvtype,  
    MPI_Comm comm)
```

を用いて、プロセス数が2、本MCAパラメータに1を指定した場合の、引数指定の仕方の誤りの例と正しい例を以下に示す。

誤りの例：

- 受信側の要素数の配列がランクごとで異なる。
- 型仕様が同じで正しいMPIプログラムだが、本MCAパラメータ値が1の場合、誤動作の可能性あり。

ランク 0:

```
sendcount: 2;          sendtype: MPI_INT  
recvcounts配列: 2, 4;  recvtype: MPI_INT
```

ランク 1:

```
sendcount: 4,          sendtype: MPI_INT  
recvcounts配列: 1, 2;  recvtype: INT_2
```

正しい例：

- 受信側の要素数の配列が全てのランク間で同じ。

ランク 0:

```
sendcount: 2;          sendtype: MPI_INT  
recvcounts配列: 2, 4;  recvtype: MPI_INT
```

ランク 1:

```
sendcount: 4,          sendtype: MPI_INT  
recvcounts配列: 2, 4;  recvtype: MPI_INT
```

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
 - 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル
 - MPI統計情報
 - 「富岳」の通信モード
 - 高速な集団通信
 - イーガー通信方式とランデブー通信方式
 - メモリ使用量を抑えるための考慮点
- 【付録】参考文献

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

本節の内容

「富岳」では、一対一通信でのメッセージの通信方式として、イーガー（**Eager**）通信方式とランデブー（**Rendezvous**）通信方式を実装しています。本節では、この二つの通信方式について説明します。

なお、「富岳」の通信モードの節では、高速型通信モードと省メモリ型通信モードを説明しましたが、これらは**MPI**ライブラリのメモリ使用量の観点からの分類です。これから説明するイーガー通信方式とランデブー通信方式は通信手順の実装方法（メッセージをどのような手順でやり取りするか）の観点からの分類です。

- 通信方式の概要
- 通信方式・ホップ数の確認方法
- 通信方式の変更
- 計算と通信のオーバーラップ

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

本節の内容

「富岳」では、一対一通信でのメッセージの通信方式として、イーガー（**Eager**）通信方式とランデブー（**Rendezvous**）通信方式を実装しています。本節では、この二つの通信方式について説明します。

なお、「富岳」の通信モードの節では、高速型通信モードと省メモリ型通信モードを説明しましたが、これらは**MPI**ライブラリのメモリ使用量の観点からの分類です。これから説明するイーガー通信方式とランデブー通信方式は通信手順の実装方法（メッセージをどのような手順でやり取りするか）の観点からの分類です。

■ 通信方式の概要

- 通信方式・ホップ数の確認方法
- 通信方式の変更
- 計算と通信のオーバーラップ

■ 通信方式の概要

本スライドではイーガー（**Eager**）通信方式とランデブー（**Rendezvous**）通信方式の概要を説明します。（以降のスライドで各通信方式でのメッセージのやり取りの仕方を図で説明します。）

■ イーガー通信方式

- 送信側と受信側との事前の連携処理は無い。

→ 連携処理のオーバーヘッドなし。

- MPIライブラリ内部にバッファ領域を予め確保する。

→ 利用者プログラムの受信先メモリ領域の情報なしで送信側プロセスがメッセージを送信する（即ち、送信側が受信側の状態にかかわらずメッセージを送信する非同期型）通信方式。

→ 内部バッファ領域と利用者プログラムのメモリ空間の間でのコピーのオーバーヘッドあり。

■ ランデブー通信方式

- 送信側と受信側とで事前の連携処理を行う。

→ 利用者プログラムの受信先メモリ領域が確定してから送信側プロセスがメッセージを送信する（即ち、送信側が受信側のメッセージ格納先が確定するまでメッセージを送信しない同期型）通信方式。

→ 連携処理のオーバーヘッドあり。

- MPIライブラリ内部にバッファ領域を必要としない。

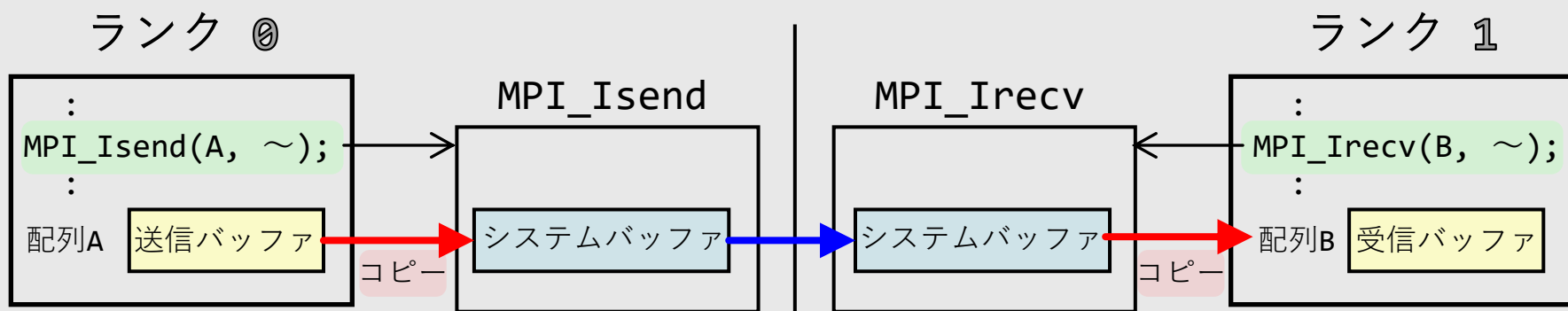
→ 利用者プログラムの送信側と受信側のメモリ空間の間で直接にコピーできる。

→ 内部バッファ領域と利用者プログラムのメモリ空間の間でのコピーのオーバーヘッドなし。

■ イーガー通信方式でのメッセージのやり取りの仕方

ランク⑩からランク①にMPI_Isend/MPI_Irecvで一対一通信を行う場合を想定する。

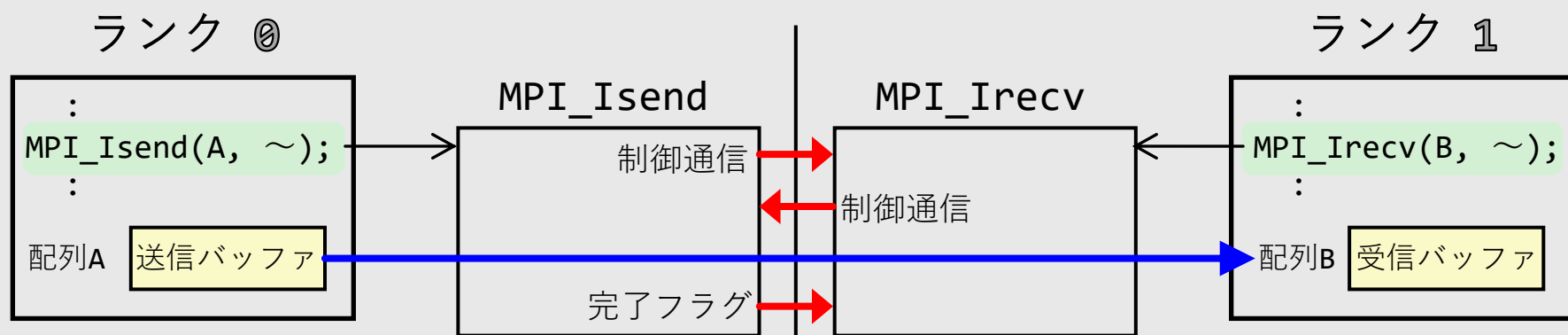
- ランク⑩の送信バッファのデータは一旦送信側のシステムバッファにコピーされてから送信される。
- 受信側で受信したデータは、一旦受信側のシステムバッファに入ってから、ランク①の受信バッファにコピーされる。
- 2回行うコピーの時間がオーバーヘッドとなる。



■ ランデブー通信方式でのメッセージのやり取りの仕方

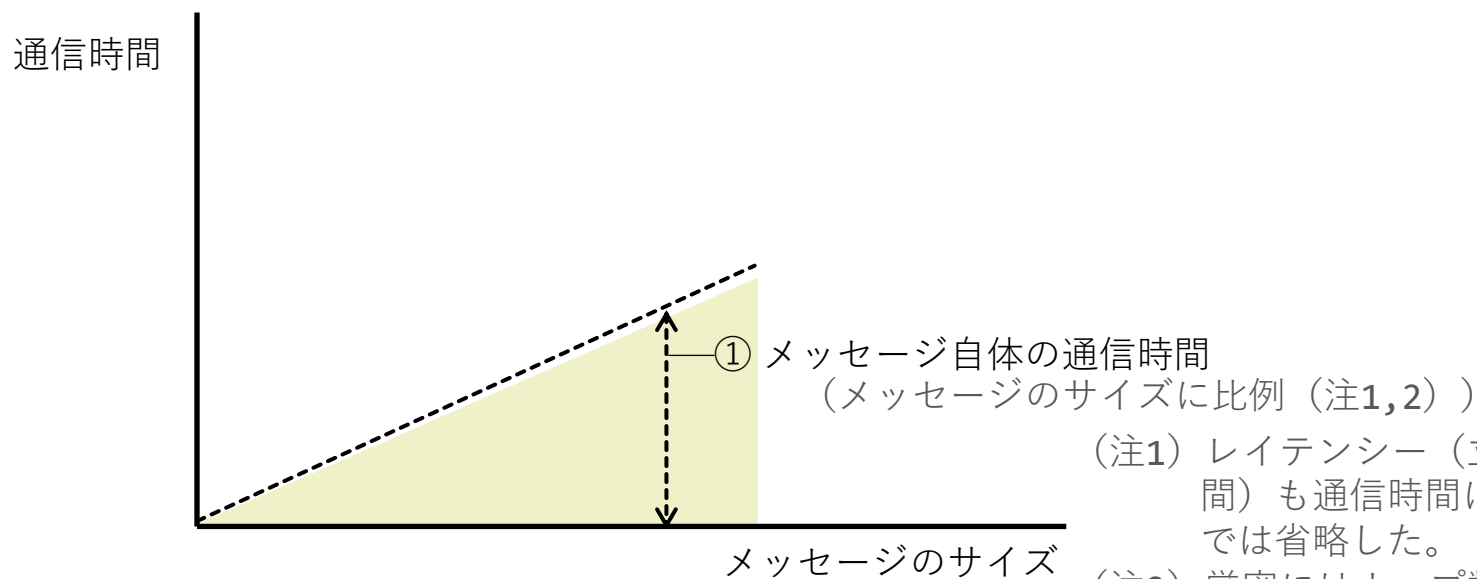
ランク④からランク①にMPI_Isend/MPI_Irecvで一対一通信を行う場合を想定する。

- ランク④の送信バッファのデータは、ランク①の受信バッファに送信される。
- このため、システムバッファとの間でコピーを行う必要はない。
- その代わり、メッセージの通信を行う前後に、ランク④とランク①の間で制御通信（制御信号のやりとり）を行う必要があり、この時間がオーバーヘッドになる。



■ 二つの通信方式の比較 [1/4]

◆ メッセージ自体の通信時間①は、メッセージのサイズに比例する（右下（注1,2）も参照）。

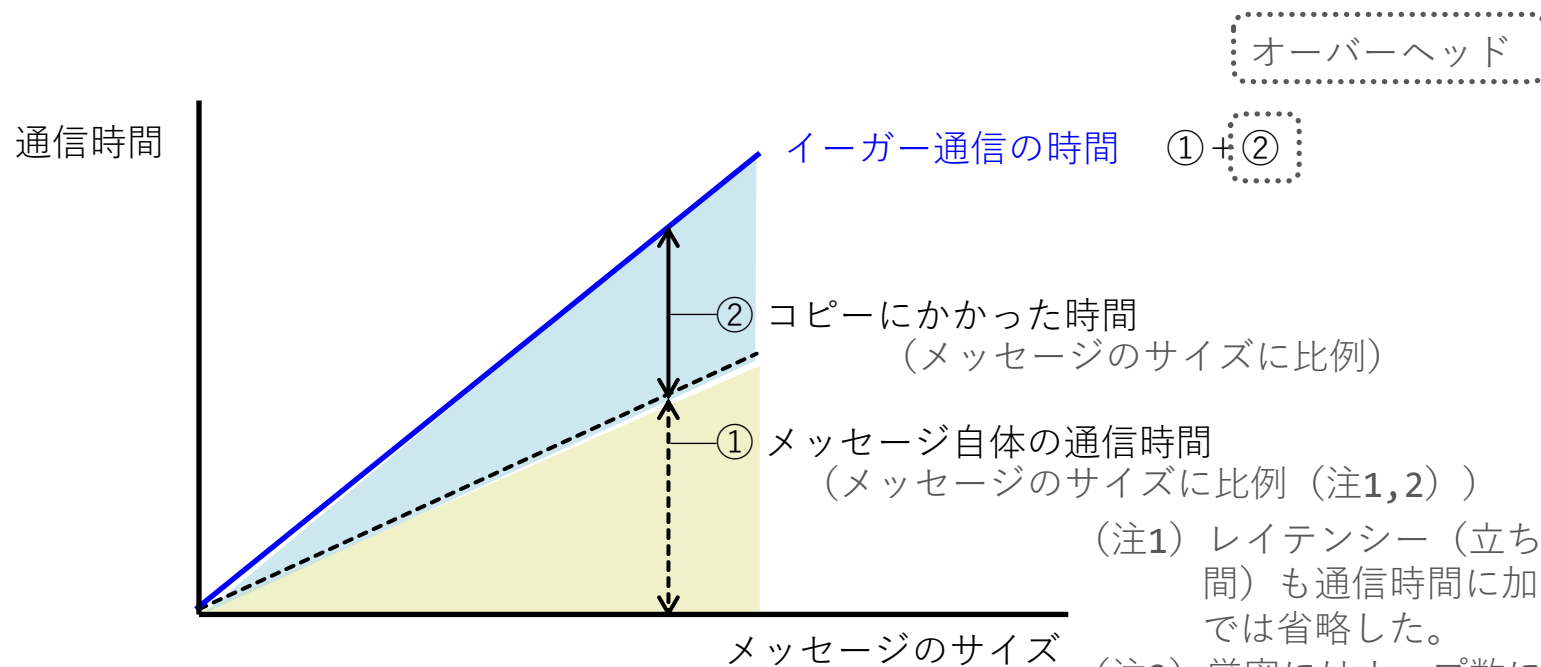


(注1) レイテンシー（立ち上がり時間）も通信時間に加わるが、図では省略した。

(注2) 厳密にはホップ数にも依存するが、図では省略した。

■ 二つの通信方式の比較 [2/4]

- ◆ メッセージ自体の通信時間①は、メッセージのサイズに比例する（右下（注1,2）も参照）。
- ◆ **イーガー通信**では、①の時間に、コピーにかかった時間②がオーバーヘッドとして加わる。
 - コピーにかかった時間②は、メッセージのサイズに比例する。

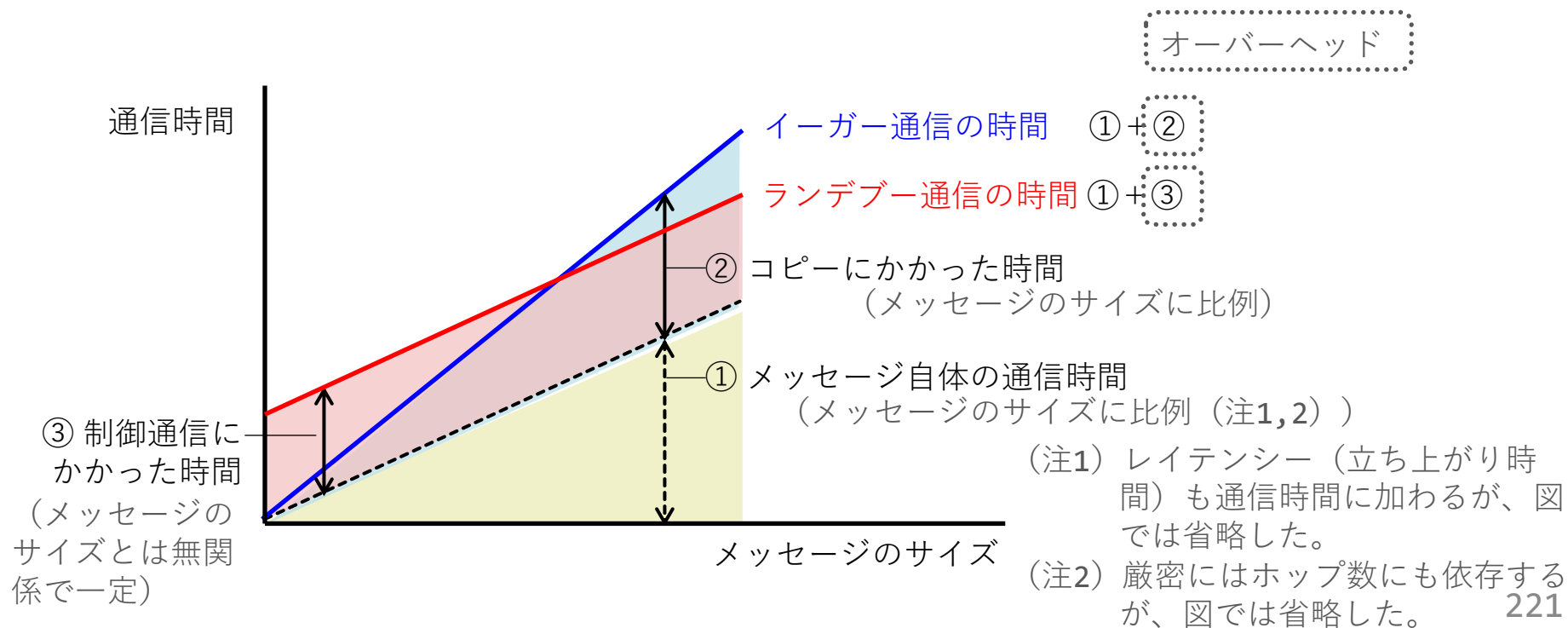


(注1) レイテンシー（立ち上がり時間）も通信時間に加わるが、図では省略した。

(注2) 厳密にはホップ数にも依存するが、図では省略した。

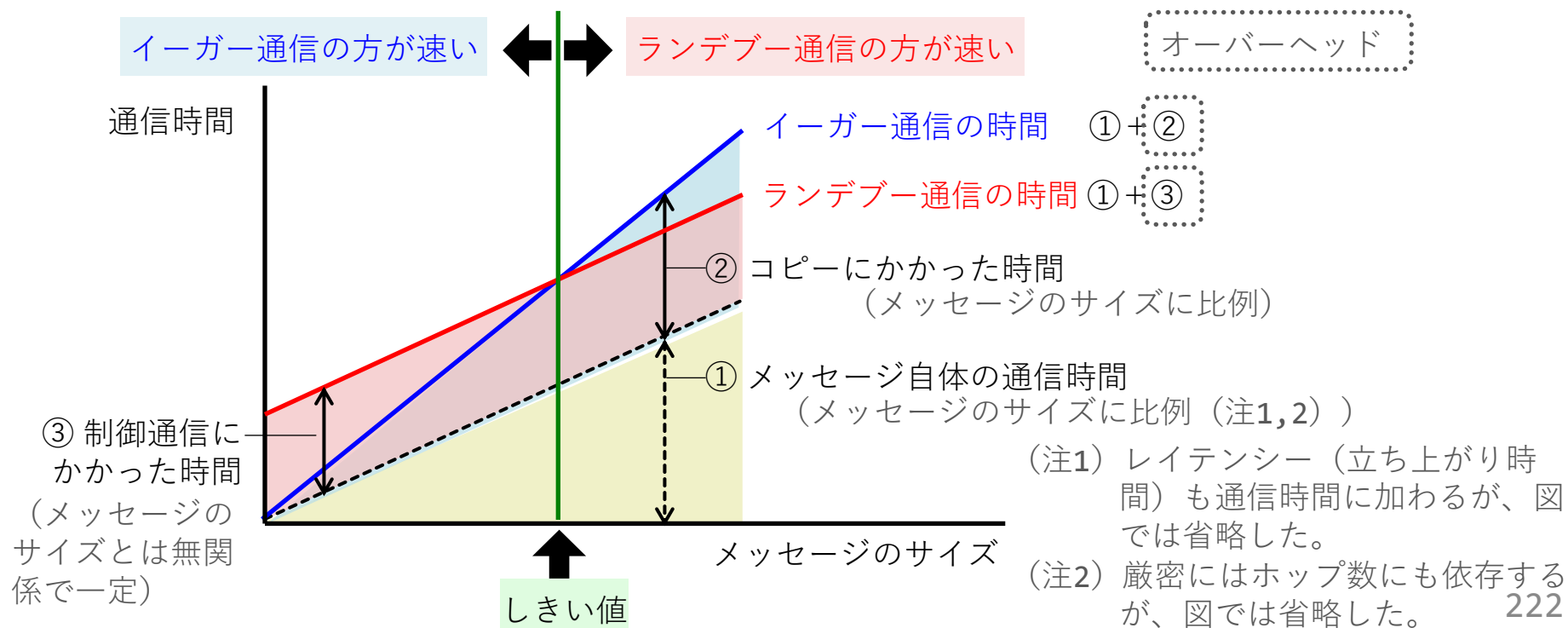
■ 二つの通信方式の比較 [3/4]

- ◆ メッセージ自体の通信時間①は、メッセージのサイズに比例する（右下（注1,2）も参照）。
- ◆ **イーガー通信**では、①の時間に、コピーにかかった時間②がオーバーヘッドとして加わる。
 - コピーにかかった時間②は、メッセージのサイズに比例する。
- ◆ **ランデブー通信**では、①に、制御通信にかかった時間③がオーバーヘッドとして加わる。
 - 制御通信にかかった時間③は、メッセージのサイズとは無関係で一定である（（注1,2））。



■ 二つの通信方式の比較 [4/4]

- ◆ メッセージ自体の通信時間①は、メッセージのサイズに比例する（右下（注1,2）も参照）。
- ◆ **イーガー通信**では、①の時間に、コピーにかかった時間②がオーバーヘッドとして加わる。
 - コピーにかかった時間②は、メッセージのサイズに比例する。
- ◆ **ランデブー通信**では、①に、制御通信にかかった時間③がオーバーヘッドとして加わる。
 - 制御通信にかかった時間③は、メッセージのサイズとは無関係で一定である（（注1,2））。
- ◆ 従って、メッセージのサイズが「しきい値」以下ではイーガー通信の方が速く、「しきい値」以上ではランデブー通信の方が速くなる。



■ 「しきい値」について

- 「富岳」では、短いメッセージの送信にはイーガー通信方式を選択し、長いメッセージの送信にはランデブー通信方式を選択するよう、送信するメッセージのサイズによって内部的に通信方式を切り替えている(*1)。

(*1) 厳密には、メッセージのサイズだけでなく、通信を行うノード間に介在するネットワークの本数（ホップ数）についても考慮されている。

- 高速型通信モードでは、イーガー通信方式とランデブー通信方式を切り替えるときの「しきい値」（バイト数）を次の式で決めている。

$$\text{しきい値} = 45056 + \text{ホップ数} \times 296$$

- 省メモリ型通信モードにおける「しきい値」については、メモリ使用量が少なくなるように、本処理系によって適切な値が自動的に設定される。

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

■ 通信方式の概要

■ 通信方式・ホップ数の確認方法

本小節の内容

- ◇ 前小節までで説明したように、一対一の通信方式は通信するメッセージのサイズ、ホップ数、および「しきい値」から自動的に選択されます。「しきい値」より短いメッセージはイーガー通信方式が、長いメッセージはランデブー通信方式が採用されます。
- ◇ どちらの通信方式が何回使用されたか、およびホップ数は、以前の節「MPI統計情報」で説明したMPI統計情報内に表示されます。
- ◇ ここでは、MPI統計情報により通信方式およびホップ数を確認する方法を説明します。

■ 通信方式の変更

■ 計算と通信のオーバーラップ

■ 通信方式の確認方法 [1/4]

- 図のプログラムを例にして通信方式を調べる方法を説明する。

- 本プログラム例では①でランク0が `MPI_Send` を呼び、②でランク `nprocs-1` が `MPI_recv` を呼んで、ランク0と `nprocs-1` で一対一通信する。
- 本例では、配列要素数 `N=1000` の `MPI_DOUBLE` の配列を送受信しているので、1回の送受信でのメッセージサイズ（バイト数）は `1000*8=8000` バイトであることに注意して下さい。
- 一対一通信をforループで10万回繰り返す。

```
int nprocs, myrank;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
constexpr int N = 1000, NLOOP = 100000;
double A[N] = {};
int src_rank = 0;
int dst_rank = nprocs-1;
MPI_Barrier(MPI_COMM_WORLD);

for (int i=0; i<NLOOP; i++) {
    if (myrank == src_rank){
        /* src_rank の値は 0 */
        MPI_Send(&A[0], N, MPI_DOUBLE,
                dst_rank, 99, MPI_COMM_WORLD);
    }
    if (myrank == dst_rank){
        /* dst_rank の値は nprocs-1 */
        MPI_Recv(&A[0], N, MPI_DOUBLE,
                src_rank, 99, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
    }
}
```

①

②

■ 通信方式の確認方法 [2/4]

- 通信方式はMPI統計情報を用いて調べることができる。

- 本例では問題の区間のみのMPI統計情報を調べたいとし、「MPI統計情報」の節で説明した区間指定ルーチンを使用する。

※ 図では区間指定ルーチンの使用に必要な `#include <mpi-ext.h>` の指定は記載を省略しているので、注意して下さい。

- ジョブスクリプトにおいて環境変数 `export OMPI_MCA_mpi_print_stats=3` を指定して実行する。
値「3」は、ランク0の標準エラー出力にMPI統計情報を出力するという指定である。

- 本例では、ノード数8、ノード内プロセス数4で実行する。

```
int nprocs, myrank;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
constexpr int N = 1000, NLOOP = 100000;
double A[N] = {};
int src_rank = 0;
int dst_rank = nprocs-1;
MPI_Barrier(MPI_COMM_WORLD);
FJMPI_Collection_clear();
FJMPI_Collection_start();
for (int i=0; i<NLOOP; i++) {
    if (myrank == src_rank){
        /* src_rank の値は 0 */
        MPI_Send(&A[0], N, MPI_DOUBLE,
                 dst_rank, 99, MPI_COMM_WORLD);
    }
    if (myrank == dst_rank){
        /* dst_rank の値は nprocs-1 */
        MPI_Recv(&A[0], N, MPI_DOUBLE,
                 src_rank, 99, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
}
FJMPI_Collection_stop();
FJMPI_Collection_print(const_cast<char
*>("N_1000_NLOOP_100000"));
```

①

②

■ 通信方式の確認方法 [3/4]

- ジョブが終了すると、ランク0の標準エラー出力にMPI統計情報が出力され、MPI統計情報内の③の「Per-protocol Communication Count」に、利用された通信方式と、通信方式ごとの通信回数が表示される。

③

----- Per-protocol Communication Count -----				
	MAX		MIN	AVE
Eager	99989	[0]	0 [1]	3124.7
Rendezvous	11	[0]	0 [1]	0.3
Persistent_Extended_IF	0	[0]	0 [0]	0.0
Unexpected_Message	1	[0]	0 [1]	0.1

- 各行の意味を以下の表に示す。Eager欄とRendezvous欄が、それぞれイーガー通信方式とランデブー通信方式の回数である。

Per-protocol Communication Count の各欄の説明	
Eager	一対一通信において、送信側でイーガー通信方式を利用した回数
Rendezvous	一対一通信において、送信側でランデブー通信方式を利用した回数
Persistent_Extended_IF	一対一通信において、送信側で拡張持続的通信要求インターフェースを利用した回数
Unexpected_Message	一対一通信において、内部バッファに一時的に退避したメッセージの最大数

- 通信回数は最大値（MAX）、最小値（MIN）、および平均値（AVE）が表示される。最大値および最小値については、それぞれ対応する並列プロセスのランクが括弧内に表示される。

■ 通信方式の確認方法 [4/4]

■ 本例では、

- 最初の**11**回は省メモリ型通信モードが用いられ(*2)、ランデブー通信方式で通信を行う。
- その後通信モードが切り替わり、残りの99989回は高速型通信モードが用いられ、イーガー通信方式で通信を行う。

(*2) 「通信モード」の節で説明したように、通常、最初は省メモリ型通信モードとなることに注意して下さい。

③

----- Per-protocol Communication Count -----				
	MAX		MIN	AVE
Eager	99989 [0]		0 [1]	3124.7
Rendezvous	11 [0]		0 [1]	0.3
Persistent_Extended_IF	0 [0]		0 [0]	0.0
Unexpected_Message	1 [0]		0 [1]	0.1

■ ホップ数の確認方法

- MPI統計情報内の「Per-peer Communication Count」に、**一対一通信したプロセス間の距離に関する情報**（同じノード内の通信であるか、異なるノード間の通信であるか）、および**Tofu通信に関する情報**（通信のためにプロセス間に確立されたコネクションの数、およびホップ数）が表示される。

----- Per-peer Communication Count -----				
	MAX		MIN	AVE
In_Node	0 [0]		0 [0]	0.0
Neighbor	0 [0]		0 [0]	0.0
Not_Neighbor	100000 [0]		0 [1]	3125.0
Total_Count	100000 [0]		0 [1]	3125.0
Connection	4 [0]		3 [1]	3.1
Max_Hop	2 [0]		1 [1]	1.5
Average_Hop	1.50 [31]		1.00 [1]	1.18

- 各行の意味を以下の表に示す。Max_Hop欄とAverage_Hop欄が、プロセス間のホップ数に関する情報である。

Per-peer Communication Count の各欄の説明	
In_Node	一対一通信におけるノード内の通信回数
Neighbor	一対一通信における隣接するノード間の通信回数
Not_Neighbor	一対一通信における隣接しないノード間の通信回数
Total_Count	一対一通信における通信回数の総計
Connection	Tofu通信のためのコネクション数
Max_Hop	Tofu通信によるプロセス間の最大ホップ数
Average_Hop	Tofu通信によるプロセス間の平均ホップ数

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

■ 通信方式の概要

■ 通信方式・ホップ数の確認方法

■ 通信方式の変更

本小節の内容

通信方式はMCAパラメータで変更できる場合があります。ここでは、MCAパラメータを用いて通信方式を変更する例を二つ示します。

■ 通信モードを変更することにより通信方式を間接的に変更する例

前小節の例では、最初の数回は省メモリ型通信モードが用いられ、ランデブー通信方式で通信を行っていました。その後通信モードが切り替わり、高速型通信モードが用いられ、イーガー通信方式で通信が行われました。

ここでは、MCAパラメータを用いて最初から高速型通信モードで通信を行うように指定し、その結果最初からイーガー通信方式で通信を行うようになる例を示します。

■ 通信方式を切り替えるしきい値を変更する例

デフォルトのしきい値によって選択された通信方式よりも、もう一方の通信方式の方が速い場合があります。ここではMCAパラメータによる高速型通信モードでのしきい値の変更方法と、しきい値の変更により通信方式が切り替わることの確認、および通信方式が切り替わることで速くなる例を示します。

■ 計算と通信のオーバーラップ

■ 通信モードを変更することにより、通信方式を間接的に変更する [1/2]

- 省メモリ型通信モードから高速型通信モードに切り替わる通信回数の基準値は通常**16**に設定されている。
- この基準値はMCAパラメータ`common_tofu_fastmode_threshold`によって変更できる。

`-mca common_tofu_fastmode_threshold` 0以上の整数値 (省略値は16)

省メモリ型通信モードから高速型通信モードに切り替えるときの条件となる通信回数(*1)を指定する。

(*1) この回数にはMPIライブラリ内部で行われる制御通信の回数も含まれる。そのため指定した回数より少ない回数のMPIルーチンの呼び出しでモードが切り替わることがある。

`-mca common_tofu_fastmode_threshold` 0

本MCAパラメータに値「0」を指定した場合、（高速型通信モードで通信可能な相手プロセス数の上限に達していない限り）最初から高速型通信モードで通信を行う。

■ 通信モードを変更することにより、通信方式を間接的に変更する [2/2]

- 前小節の例（MPI_Send/MPI_recvで一対一通信する例）において、ジョブスクリプト内で以下の④のようにMCAパラメータcommon_tofu_fastmode_thresholdに値「0」を指定した場合、省メモリ型通信モードを使わず、最初から高速型通信モードを使うように指示する。

```
export OMPI_MCA_common_tofu_fastmode_threshold=0 ④
```

- 上記④を指定した場合、標準エラー出力のMPI統計情報内の⑤に示すように、④の指定により最初から高速型通信モードが用いられ、イーガー通信方式で通信を行うようになったことが確認できる。なお、イーガー通信方式となるのは以下の(*2)に示す理由による。

----- Per-protocol Communication Count -----				
	MAX		MIN	AVE
Eager	100000 [0]		0 [1]	3125.0
Rendezvous	0 [0]		0 [0]	0.0

- (*2) イーガー通信方式とランデブー通信方式の「しきい値」についての節で説明したように、高速型通信モードでは、イーガー通信方式とランデブー通信方式を切り替えるときの「しきい値」（バイト数）は次の式で決まる。

$$\text{しきい値} = 45056 + \text{ホップ数} \times 296$$

前小節のMPI_Send/MPI_recvで一対一通信する例では、配列要素数 N=1000 の MPI_DOUBLE の配列を送受信しているので、メッセージサイズ（バイト数）は1000*8=8000バイトである。

$$8000 < 45056 \leq \text{しきい値 (バイト数)} = 45056 + \text{ホップ数} \times 296$$

であることから、メッセージサイズは「しきい値」より小さいためイーガー通信方式が選択される。

■ 通信方式を切り替えるしきい値を変更する [1/4]

- プログラムは前例で使用されたプログラムと、以下の点を除いて同じである。

- 送受信する配列の要素数を $N = 6000$ と前例よりも大きくしている。1回の通信でのメッセージサイズは $6000 \times 8 = 48000$ バイトである。
- 計測対象区間内の通信に二つの通信方式が混在するのを防ぐため、計測対象区間よりも前の箇所（ダミー走行と記した箇所）で、同一通信パターンでダミーの通信を十分な回数行い、計測対象区間内の通信の繰り返しが全て高速型通信モードとなるようにする。

```
constexpr int N = 6000, NLOOP = 100000;
double A[N] = {};
int src_rank = 0;
int dst_rank = nprocs-1;
(ダミー走行)
MPI_Barrier(MPI_COMM_WORLD);
FJMPI_Collection_clear();
FJMPI_Collection_start();
for (int i=0; i<NLOOP; i++) {
    if (myrank == src_rank){
        /* src_rank の値は 0 */
        MPI_Send(&A[0], N, MPI_DOUBLE,
                 dst_rank, 99, MPI_COMM_WORLD);
    }
    if (myrank == dst_rank){
        /* dst_rank の値は nprocs-1 */
        MPI_Recv(&A[0], N, MPI_DOUBLE,
                 src_rank, 99, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
}
FJMPI_Collection_stop(); (以下省略)
```

■ 通信方式を切り替えるしきい値を変更する [2/4]

■ 本例では、ノード数8、ノード内プロセス数4で実行する。

■ MCAパラメータとしてOMPI_MCA_mpi_print_stats=3のみ指定した場合、MPI統計情報よりランデブー通信方式で通信(*1)していることが確認される。

----- Per-protocol Communication Count -----				
	MAX		MIN	AVE
Eager	0 [0]		0 [0]	0.0
Rendezvous	100000 [0]		0 [1]	3125.0

①

(*1) MPI統計情報よりMax_Hopが2であると確認されることから、本例での高速型通信モードのしきい値は大きくても45,648と計算される。「しきい値」のこの値よりも本例での1回の通信のメッセージサイズ6000*8 = 48000バイト方が大きいことから、ランデブー通信方式となる。

■ MPI統計情報より通信時間は1.56秒であった。

■ 通信方式を切り替えるしきい値を変更する [3/4]

- 高速型通信モードでは、イーガー通信方式とランデブー通信方式を切り替えるときの「しきい値」（バイト数）を次の式で決めている。

$$\text{しきい値} = 45056 + \text{ホップ数} \times 296$$

- 高速型通信モードでの「しきい値」はMCAパラメータ**bt1_tofu_eager_limit**によって変更することができる。

-mca bt1_tofu_eager_limit 1以上の整数値

高速型通信モードにおけるイーガー通信方式とランデブー通信方式を切り替える「しきい値」となるメッセージのサイズ（バイト数）を指定する。指定したメッセージのサイズ（バイト数）(*2)よりも小さいメッセージについてはイーガー通信方式によって送信する。

(*2) 厳密には、実際のメッセージのサイズ（バイト数）に内部的に付加されるヘッダー分のサイズ（数十バイト）を加えた値となる。

本MCAパラメータを指定すると、ホップ数に関係なく指定した値を「しきい値」として使用する。

■ 通信方式を切り替えるしきい値を変更する [4/4]

- 本例では、1回の通信でのメッセージサイズは $6000 \times 8 = 48000$ バイトであることを考慮して、この値より大きくなるようにジョブスクリプトに以下の値と指定する。

```
export OMPI_MCA_btl_tofu_eager_limit=50000 ②
```

- 実行すると、MPI統計情報より、イーガー通信方式が採用されていることが確認される。

```
----- Per-protocol Communication Count -----
                MAX                MIN                AVE
Eager          100000 [  0]          0 [  1]          3125.0
Rendezvous      0 [  0]          0 [  0]           0.0
```

③

- また、通信時間は1.17秒となり、しきい値の変更前1.56秒より速くなった。

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

- 通信方式の概要
- 通信方式・ホップ数の確認方法
- 通信方式の変更

■ 計算と通信のオーバーラップ

本小節の内容

一対一通信を、非ブロッキング通信のMPI_ISENDとMPI_IRECVを使用して行うと、「富岳」では、計算と通信をオーバーラップ（同時実行）させ、通信のオーバーヘッドをある程度隠蔽することができます。イーガー通信方式の場合、特別な処理をしなくてもオーバーラップは可能です(*1)。ランデブー通信方式の場合、通常そのままでは計算と通信はオーバーラップされません(*2)。 (*1,*2) 後続のスライドで示します。

本小節では、一対一通信を非ブロッキング通信のMPI_ISENDとMPI_IRECVで行う場合に計算と通信をオーバーラップさせる方法を、イーガー通信方式とランデブー通信方式それぞれについて、具体的な例を用いて説明します。

- イーガー通信方式のオーバーラップ
- ランデブー通信方式のオーバーラップ

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

■ 計算と通信のオーバーラップ

■ イーガー通信方式のオーバーラップ

「通信方式の概要」で説明したように、イーガー通信方式では送信側と受信側との事前の連携処理は無く、MPIライブラリ内部にバッファ領域を予め確保し、送信側が受信側の状態にかかわらずメッセージを送信する非同期型の通信方式です。そのため、以下で示すように、通信要求の完了を待つ `MPI_Wait` と計算の実行順序にさえ注意すれば、他の特別な処理をしなくてもオーバーラップは可能です。

ここではイーガー通信方式の場合について、オーバーラップできないパターンとできるパターン、オーバーラップできるパターンについてプログラム例とMPI統計情報での確認例を示します。

■ オーバラップできないパターン

■ オーバラップできるパターン

■ プログラム例（オーバーラップあり）と実行結果

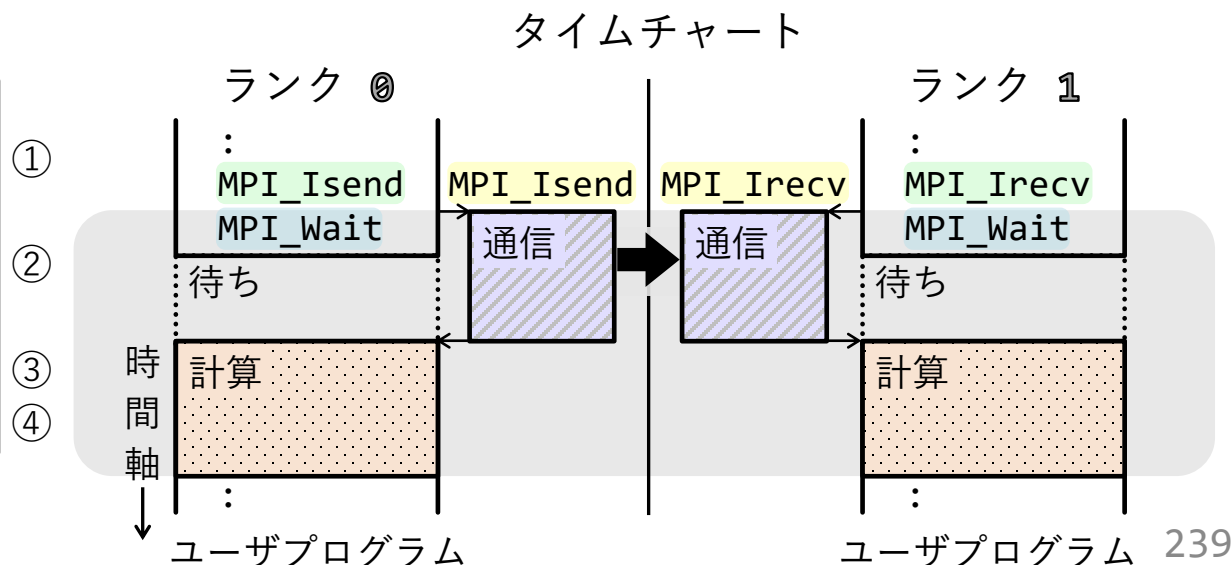
■ ランデブー通信方式のオーバーラップ

■ オーバラップできない例

下の左図のように、①でランク①がMPI_Isendを、②でランク②がMPI_Irecvする場合を例にする。

- ①MPI_Isendと②MPI_Irecvの後にすぐ③MPI_Waitを実行した場合、タイムチャートは下の右図のようになる。
- ①MPI_Isendと②MPI_Irecvで通信が開始し、制御はユーザプログラムにすぐに復帰するが、直後の③MPI_Waitで通信の完了を待つ（MPI_Waitの動作は通信相手プロセスに依存するブロッキング動作である）。
- ④の計算は通信が完了してから開始される。そのため、計算と通信はオーバーラップされない。

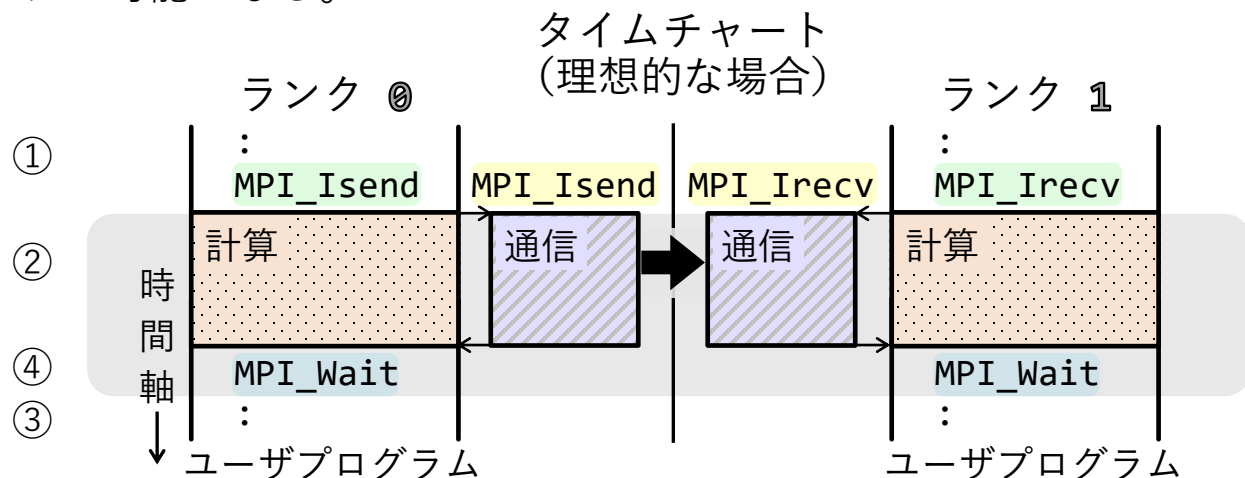
```
if (myrank == 0){  
    MPI_Isend(~);  
}else if (myrank == 1){  
    MPI_Irecv(~);  
}  
MPI_Wait(~);  
計算
```



■ オーバラップできる例

- ③MPI_Waitと④計算を逆にした場合、①②で通信が開始し、制御はユーザプログラムにすぐに復帰する。復帰後、④の計算が実行される。タイムチャートは下の右図のようになり、計算と通信のオーバーラップが可能となる。

```
if (myrank == 0){  
    MPI_Isend(~);  
}else if (myrank == 1){  
    MPI_Irecv(~);  
}  
計算  
MPI_Wait(~);
```



⚠ 【注意】 通信するメッセージと計算に関係がある場合(*1)は、オーバーラップさせることはできない (④③の順序にすることはできない)。

(*1) 例えば、①での送信バッファの内容を④の計算で書き換える場合や、②での受信バッファの内容を④の計算で参照するような場合。

【参考情報】 ①での送信バッファの内容を④の計算で書き換える場合、MCAパラメータ `mpi_check_buffer_write` に値「1」を指定 (本パラメータ無指定時の値は0) して実行すると、標準エラー出力にその旨のメッセージを出力して終了する。予めリンク時にリンカのオプション「-Wl,-export-dynamic」を指定して実行プログラムを作成し、実行時に本MCAパラメータに値「1」を指定をして実行すると、送信バッファの書き込み破壊が発生したサブルーチン名も表示される。

「富岳」のMPI通信 ▶ イーガー通信方式とランデブー通信方式 ▶ 計算と通信のオーバーラップ ▶ イーガー通信方式のオーバーラップ

■ プログラム例（オーバーラップあり）

```
constexpr int N = 1600;  
constexpr int M = 60;  
constexpr int NLOOP = 100000;  
double A[N] = {};  
double B[M][M] = {};  
int src_rank = 0;  
int dst_rank = 1;
```

```
MPI_Request req[1];
```

(ダミー走行)

```
for (int ll=0; ll<NLOOP; ll++) {
```

```
    if (myrank == src_rank){
```

```
        MPI_Isend(&A[0], N, MPI_DOUBLE, dst_rank, 999, MPI_COMM_WORLD, &req[0]);
```

```
    }else if (myrank == dst_rank){
```

```
        MPI_Irecv(&A[0], N, MPI_DOUBLE, src_rank, 999, MPI_COMM_WORLD, &req[0]);
```

```
    }
```

```
    for(int i=0; i<M; i++){
```

```
        for(int j=0; j<M; j++){
```

```
            B[i][j] += 1.0;
```

```
        }
```

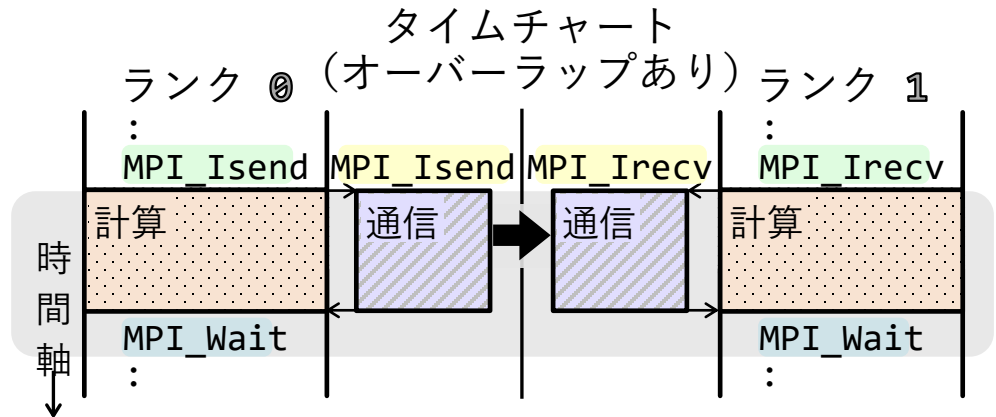
```
    }
```

```
    if (myrank == src_rank || myrank == dst_rank){
```

```
        MPI_Wait(req, MPI_STATUS_IGNORE);
```

```
    }
```

```
}
```



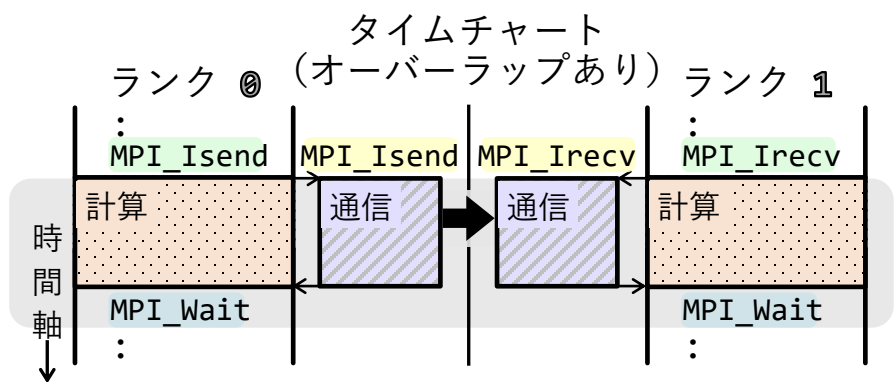
※ 計測対象区間内の通信で通信モードが切り替わるのを防ぐため、計測対象区間よりも前の箇所（ダミー走行と記した箇所）で同一通信パターンでダミーの通信を十分な回数行い、計測対象区間内の通信が全て同一の通信モードとなるようにする。

※ 本小節のオーバーラップのプログラム例ではオーバーラップできるか否かのみを問題にするとする。計算部分の処理量は配列Bの要素数を決める定数Mで自由に設定できるので、通信時間と計算時間の比は問題にしないこととする。

「富岳」のMPI通信 ▶ イーガー通信方式とランデブー通信方式 ▶ 計算と通信のオーバーラップ
 ▶ イーガー通信方式のオーバーラップ

■ 実行結果 (MPI統計情報)

- MPI統計情報から、本例ではイーガー通信方式で通信していることが確認できる。



----- Per-protocol Communication Count -----				
	MAX		MIN	AVE
Eager	100000 [0]	0 [1]	8333.3	
Rendezvous	0 [0]	0 [0]	0.0	

- 以下の上の表より、本例ではオーバーラップの有りと無しの経過時間の差は**0.10**秒である。これは以下の下の表の参考データの計算部分のみの時間に等しい。このことから、計算部分が通信部分に隠蔽され、計算と通信がオーバーラップしていることが確認できる。

オーバーラップの有無	経過時間(*2)
無し	0.58秒
有り	0.48秒

- 参考データ (各処理部分の事前の個別計測結果)

通信部分のみの時間	0.47秒
計算部分のみの時間	0.10秒

(*2) ノード2x3x2:torus、ノードあたりプロセス数1で実行。「for (int ll=0; ll<NLOOP; ll++) {...}」のループの経過時間。

■ 「富岳」のMPI通信

■ イーガー通信方式とランデブー通信方式

■ 計算と通信のオーバーラップ

■ イーガー通信方式のオーバーラップ

■ ランデブー通信方式のオーバーラップ

「通信方式の概要」で説明したように、ランデブー通信方式では通信するメッセージ本体の通信の前に、送信側と受信側とで連携処理（制御通信）を行います。

ここではランデブー通信方式では制御通信の動作しだいで計算と通信はオーバーラップされない可能性があることを説明します。次に具体例として、前述のイーガー通信方式でオーバーラップするプログラム例において通信するメッセージサイズが大きくランデブー通信方式で通信を行った場合、実際オーバーラップしない例を示します。そしてオーバーラップするように修正したプログラム例を示します。

■ ランデブー通信方式での通信の動作

- 制御通信がすぐに完了する場合
- 制御通信がすぐには完了しない場合

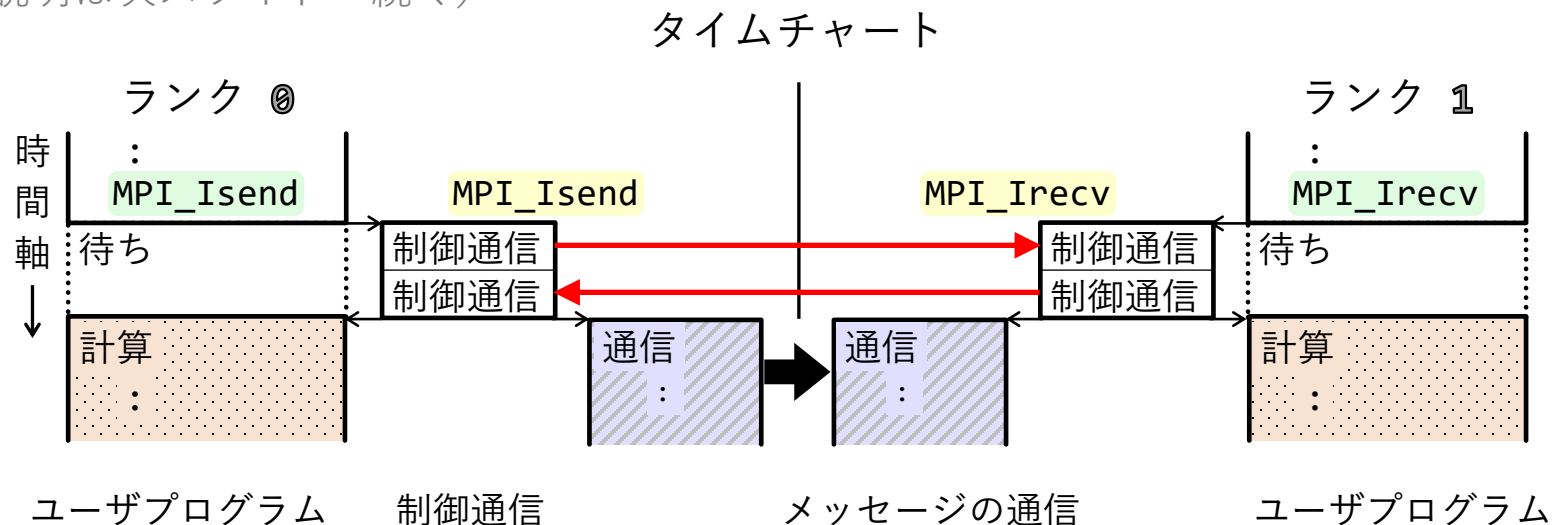
■ オーバーラップしないプログラム例と実行結果（MPI統計情報）

■ オーバーラップするよう修正したプログラム例と実行結果（MPI統計情報）

■ ランデブー通信方式での通信の動作（制御通信がすぐに完了する場合） [1/2]

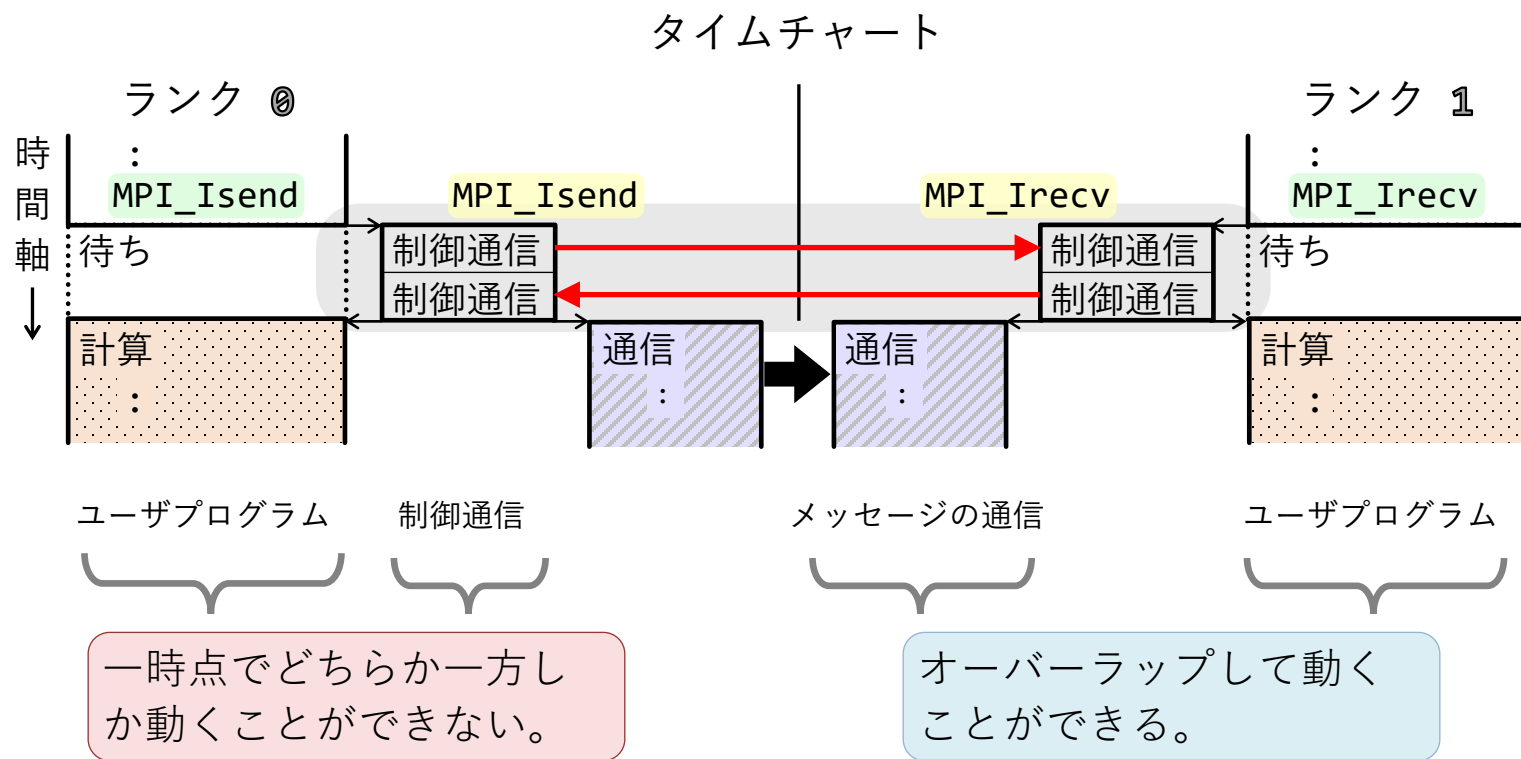
下の図のように、ランク④がMPI_Isendを、ランク①がMPI_Irecvする場合を例にとり動作を説明する。

- ランデブー通信方式の場合、MPIルーチンは、メッセージそのものの通信の前に、制御通信（制御信号のやり取り）を行う。
- （説明は次スライドへ続く）



■ ランデブー通信方式での通信の動作（制御通信がすぐに完了する場合） [2/2]

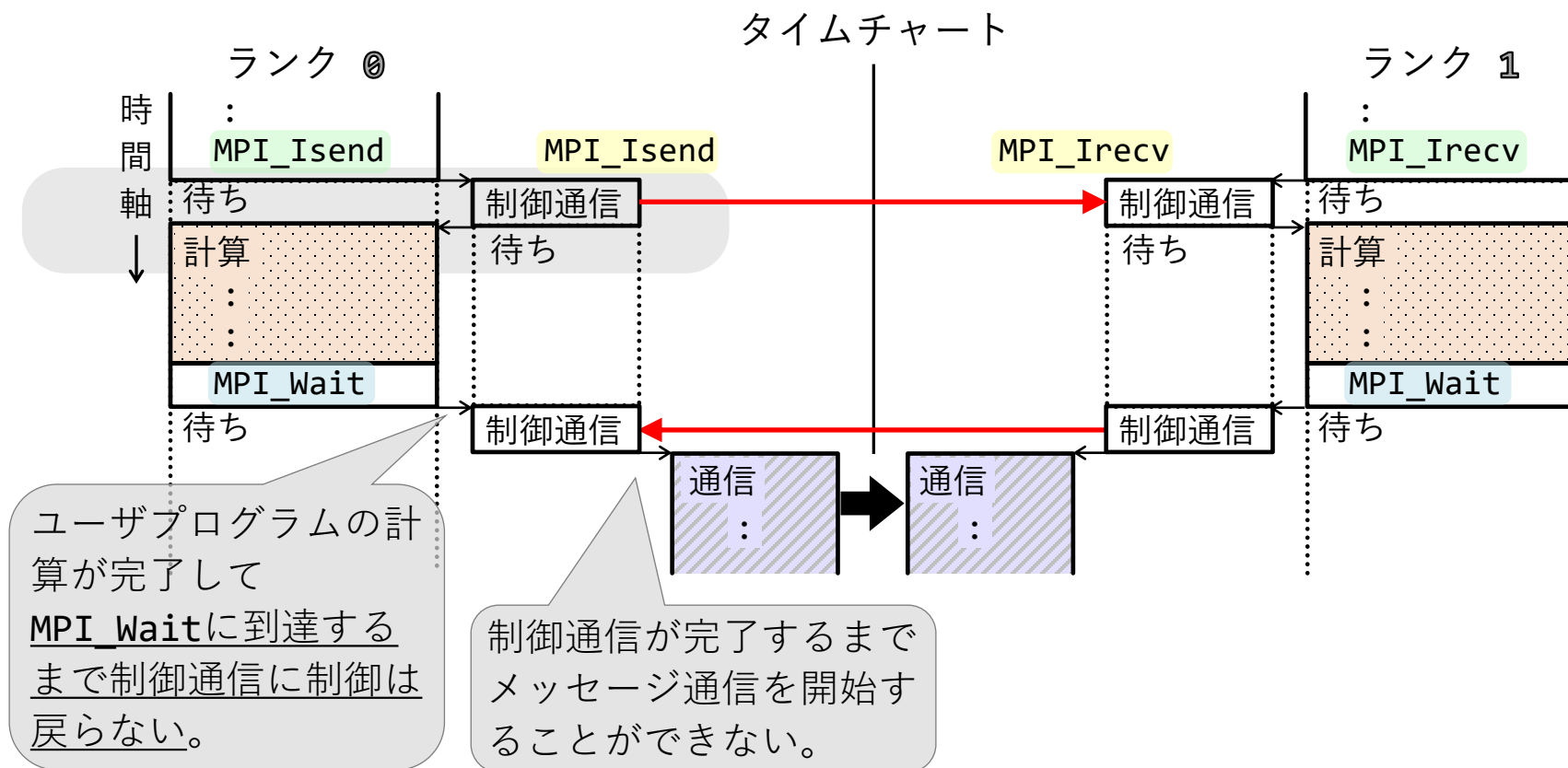
- ユーザプログラムと制御通信は一時点でどちらか一方しか動くことができない。
- メッセージの通信とユーザプログラムはオーバーラップして動くことができる。



➡ 従って、上の図のように制御通信が完了してからユーザプログラムに制御が移れば、計算とメッセージの通信はオーバーラップすることができる。

■ ランデブー通信方式での通信の動作（制御通信がすぐには完了しない場合） [1/2]

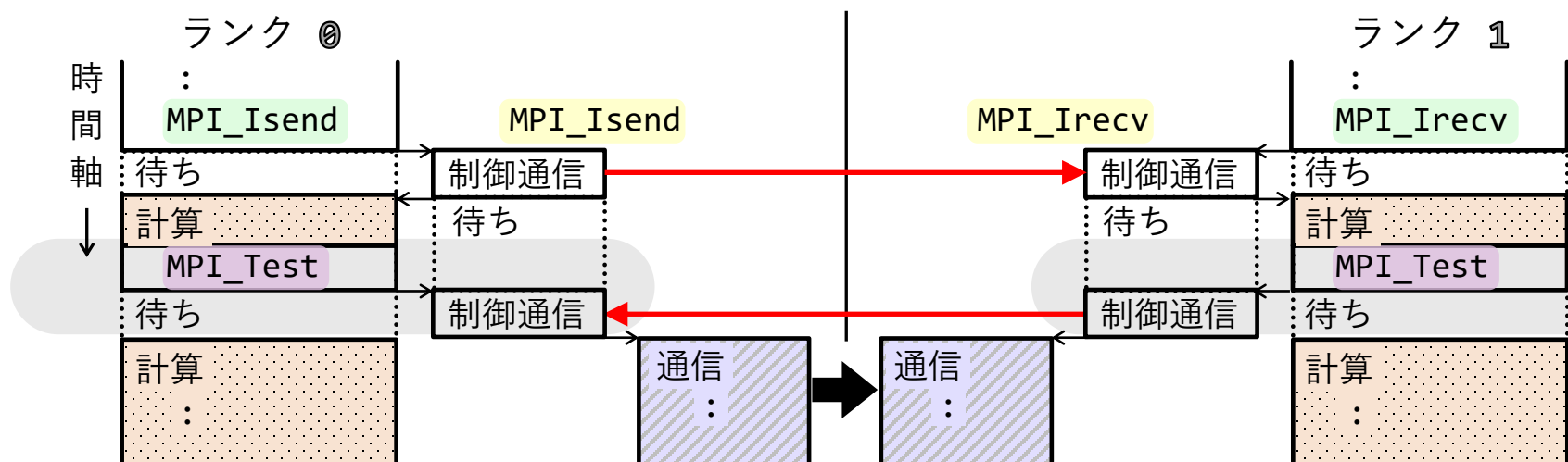
- ・ タイミングによっては、制御通信が完了しないうちに、ユーザプログラムに制御が移ることがある。



➡ 計算とメッセージの通信をオーバーラップさせることができない。

■ ランデブー通信方式での通信の動作（制御通信がすぐには完了しない場合） [2/2]

- 制御通信が完了しないうちにユーザプログラムに制御が移るような場合、計算の途中でダミーのMPIルーチン **MPI_Test** を実行すると、制御通信に制御が移って制御通信が完了し、以後は計算とメッセージの通信をオーバーラップさせることができる。



※ 図ではMPI_Testは1回だけ実行しているが、実際には定期的に複数回実行する（後のスライドでのプログラム例を参照）。

【補足】

- Cの場合の構文：`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
- MPI_testは通信要求requestが完了したかしないかを検査し、その結果を引数flagに返し、（図中の制御通信の待ちは別として）直ちに復帰する（非ブロッキング）。通信相手プロセスに依存せずに動作（局所的動作）するので、各プロセスで独立して挿入して良い。使用例を後のプログラム例において示す。

以上でランデブー通信方式での通信の動作の説明を終わる。

■ オーバーラップしないプログラム例

プログラム例として、前述のイーガー通信方式でオーバーラップするプログラム例を使用する。

- ただし、ここでは、ランデブー通信方式となることを期待して、通信する配列 **A** の要素数 **N** を大きくしていることに注意する。メッセージサイズは約 **8 MiB** である。
- なお、配列 **B** の要素数に関する寸法 **M** と、ループ反復回数 **NLOOP** の値も、イーガー通信方式でのプログラム例での値から変えている。

```
constexpr int N = 1000000;  
constexpr int M = 1000;  
constexpr int NLOOP = 1000;  
double A[N] = {};  
double B[M][M] = {};  
int src_rank = 0;  
int dst_rank = 1;
```

```
MPI_Request req[1];  
for (int ll=0; ll<NLOOP; ll++) {  
    if (myrank == src_rank){  
        MPI_Isend(&A[0], N, MPI_DOUBLE, dst_rank,  
                  999, MPI_COMM_WORLD, &req[0]);  
    }else if (myrank == dst_rank){  
        MPI_Irecv(&A[0], N, MPI_DOUBLE, src_rank,  
                  999, MPI_COMM_WORLD, &req[0]);  
    }  
    for(int i=0; i<M; i++){  
        for(int j=0; j<M; j++){  
            B[i][j] += 1.0;  
        }  
    }  
    if (myrank == src_rank || myrank == dst_rank){  
        MPI_Waitall(1, req, MPI_STATUSES_IGNORE);  
    }  
}
```

①

②

■ オーバーラップしないプログラム例の実行結果（MPI統計情報）

- 本プログラムで「N=」で指定した要素数が大きくメッセージが長い。MPI統計情報から本例での通信はランデブー通信方式で行われていることが確認される。

```
----- Per-protocol Communication Count -----
                MAX                MIN                AVE
Eager              0 [  0]              0 [  0]              0.0
Rendezvous        1000 [  0]              0 [  1]             83.3
```

- 本プログラム例の経過時間は、通信部分のみの時間と計算部分のみの時間の合計にほぼ等しいことから、計算と通信のオーバーラップが行われていないことが確認できる。

本プログラム例の経過時間(*1)	1.35秒
------------------	-------

- 参考データ（各処理部分の事前の個別計測結果）

通信部分のみの時間	1.27秒
計算部分のみの時間	0.07秒

(*1) ノード2x3x2:torus、ノードあたりプロセス数1で実行。「for (int ll=0; ll<NLOOP; ll++) {...}」のループの経過時間。

オーバーラップさせるためには、以前のスライドで説明したように、計算の途中でダミーのMPIルーチンMPI_Testを定期的に実行する必要がある。次のスライドでそのプログラム例を示す。

■ オーバーラップするよう修正したプログラム例 [1/2]

- ③のループ内で計算を行う。その際、
外側iループと内側jループの間で、
定期的に④ `if(!flag)`と、⑤
`MPI_Testall(~)`を実行する。
- (説明は次スライドへ続く)

```
MPI_Request req[1];
for (int ll=0; ll<NLOOP; ll++) {
    if (myrank == src_rank){
        MPI_Isend(&A[0], N, MPI_DOUBLE, dst_rank, ①
                  999, MPI_COMM_WORLD, &req[0]);
    }else if (myrank == dst_rank){
        MPI_Irecv(&A[0], N, MPI_DOUBLE, src_rank, ②
                  999, MPI_COMM_WORLD, &req[0]);
    }
    int flag = false; ③
    for(int i=0; i<M; i++){ ④
        if( !flag ){
            if (myrank==src_rank || myrank==dst_rank){
                MPI_Testall(1, req, &flag, ⑤
                           MPI_STATUSES_IGNORE);
            }
        }
        for(int j=0; j<M; j++){ B[i][j] += 1.0; }
    } // i
    if (myrank == src_rank || myrank == dst_rank){
        MPI_Waitall(1, req, MPI_STATUSES_IGNORE);
    }
}
```

■ オーバーラップするよう修正したプログラム例 [2/2]

- ⑤ `MPI_Testall(n, req, &flag, ~)`は 配列 `req` に格納された n 個（本例では各プロセスにつき一つ(*1)）の通信要求が完了しているかどうかを調べる。

- 少なくとも一つは未完了ならば、`flag`への返り値は、`false`
- 全て完了していれば、`flag`への返り値は、`true`

➡ 通信が完了するまで④のif文により定期的に⑤の`MPI_Testall`がコールされ、通信が完了すると⑤は実行されなくなる。

(*1) 本例では①②のように`MPI_Isend`と`MPI_Irecv`はそれぞれ一つだが、現実のプログラムでは多数の`MPI_Isend/Irecv`を実行することがある。そのような場合、通信要求は複数となるが、その場合複数の通信要求を配列に格納して`MPI_Testall`の引数に指定すれば1回のコールで複数の通信要求のテストができるので便利である。

```
MPI_Request req[1];
for (int ll=0; ll<NLOOP; ll++) {
    if (myrank == src_rank){
        MPI_Isend(&A[0], N, MPI_DOUBLE, dst_rank, ①
                  999, MPI_COMM_WORLD, &req[0]);
    }else if (myrank == dst_rank){
        MPI_Irecv(&A[0], N, MPI_DOUBLE, src_rank, ②
                  999, MPI_COMM_WORLD, &req[0]);
    }
    int flag = false;
    for(int i=0; i<M; i++){ ③
        if( !flag ){ ④
            if (myrank==src_rank || myrank==dst_rank){
                MPI_Testall(1, req, &flag, ⑤
                           MPI_STATUSES_IGNORE);
            }
        }
        for(int j=0; j<M; j++){ B[i][j] += 1.0; }
    }
    if (myrank == src_rank || myrank == dst_rank){
        MPI_Waitall(1, req, MPI_STATUSES_IGNORE);
    }
}
```

■ オーバーラップするよう修正したプログラム例の実行結果（MPI統計情報）

- MPI統計情報より、本プログラムの通信はランデブー通信方式で行われていることが確認できる。

```
----- Per-protocol Communication Count -----
                MAX                MIN                AVE
Eager              0 [    0]              0 [    0]              0.0
Rendezvous       1000 [    0]              0 [    1]             83.3
```

- 以下の上の表より本例ではMPI_Testall挿入の有無の経過時間の差は0.07秒となった。これは、以下の下の表より計算部分のみの時間に等しい。このことから、MPI_Testallを挿入したプログラムでは計算部分が通信部分に隠蔽され、計算と通信がオーバーラップしていることが確認できる。（*2）

MPI_Testall 挿入の有無	経過時間 (*3)	オーバー ラップ
なし	1.35秒	していない
あり	1.28秒	している

- 参考データ（各処理部分の事前の個別計測結果）

通信部分のみの時間	1.27秒
計算部分のみの時間	0.07秒

（*2） 計算の途中でMPI_Testallを複数回実行したことで、計算が実行されている0.07秒の間に制御通信が完了し、メッセージの通信が開始した。その結果、計算とメッセージの通信がオーバーラップした。

（*3） ノード2x3x2:torus、ノードあたりプロセス数1で実行。「for (int ll=0; ll<NLOOP; ll++) {...}」のループの経過時間。

- MPI_Testallを実行することで、ランデブー通信の場合でもオーバーラップが行われ、経過時間が短縮された。

以上でランデブー通信方式でのプログラム例を終わる。

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
 - 「富岳」のMPIライブラリのMPI規格およびマルチスレッドへの対応レベル
 - MPI統計情報
 - 「富岳」の通信モード
 - 高速な集団通信
 - イーガー通信方式とランデブー通信方式
 - メモリ使用量を抑えるための考慮点
- 【付録】参考文献

■ 「富岳」のMPI通信

■ メモリ使用量を抑えるための考慮点

本節の内容

本節では、各並列プロセス内で受信用のバッファなど、内部的に確保される**MPI**ライブラリ自身が必要とするメモリ使用量を抑えるための考慮点について説明します。

- メモリ使用量を抑えるためのMCAパラメータとそのチューニング
- MCAパラメータの手動チューニング
- MCAパラメータの自動チューニング

■ 「富岳」のMPI通信

■ メモリ使用量を抑えるための考慮点

本節の内容

本節では、各並列プロセス内で受信用のバッファなど、内部的に確保される**MPI**ライブラリ自身が必要とするメモリ使用量を抑えるための考慮点について説明します。

■ メモリ使用量を抑えるためのMCAパラメータとそのチューニング

- MCAパラメータの手動チューニング
- MCAパラメータの自動チューニング

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶メモリ使用量を抑えるためのMCAパラメータとそのチューニング

■ メモリ使用量を抑えるためのMCAパラメータ

- 「「富岳」の通信モード」の節において、メモリ使用量の観点からの分類として高速型通信モードと省メモリ型通信モード、およびそれらを切り替えるMCAパラメータや各通信モードでのバッファサイズを指定するMCAパラメータとして下記①～⑥を説明した。

①common_tofu_max_fastmode_procs

高速型通信モードで通信可能なプロセス数の上限の基準値

②common_tofu_fastmode_threshold

省メモリ型通信モードから高速型通信モードに切り替わる通信回数の基準値

③common_tofu_large_recv_buf_size

高速型通信モードのLarge受信バッファのサイズ

④common_tofu_medium_recv_buf_size

省メモリ型通信モードのMedium受信バッファのサイズ

⑤common_tofu_memory_saving_method

省メモリ型通信モードで使う方式の変更

⑥common_tofu_shared_recv_buf_size

省メモリ型通信モードのShared受信バッファのサイズ

- プロセス数が大きい場合、必要メモリ量および求められる通信性能に応じて、上記①～⑥のMCAパラメータによるチューニングが重要となる。

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶メモリ使用量を抑えるためのMCAパラメータとそのチューニング

■ メモリ使用量を抑えるためのMCAパラメータのチューニング

①～⑥のMCAパラメータによるチューニングには手動チューニングの方法と自動チューニングの方法がある。

◆ 手動チューニング

ユーザがMPIプログラムの通信相手プロセス数に着目して、MCAパラメータ①～⑥についてユーザ自身で値を指定する。

◆ 自動チューニング

ユーザはMPIライブラリが使用可能なメモリ使用量の制限値を指定するMCAパラメータ（後述）を指定し、処理系にMCAパラメータのチューニングを任せる。

①common_tofu_max_fastmode_procs

②common_tofu_fastmode_threshold

③common_tofu_large_recv_buf_size

④common_tofu_medium_recv_buf_size

⑤common_tofu_memory_saving_method

⑥common_tofu_shared_recv_buf_size

以降のスライドで、手動チューニングと自動チューニングの方法を順に説明する。

■ 「富岳」のMPI通信

■ メモリ使用量を抑えるための考慮点

本節の内容

- メモリ使用量を抑えるためのMCAパラメータとそのチューニング

■ MCAパラメータの手動チューニング

- MCAパラメータの自動チューニング

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶MCAパラメータの手動チューニング

■ MCAパラメータの手動チューニングのための分類

通信相手のプロセス数に関して以下の2種類の量に着目する。

N_tune: 通信性能を求められる通信相手プロセス数

N_all: 全通信相手プロセス数

上記2種類の量に関してMPIプログラムのパターンを以下のように分類する。

□ パターン1: $N_tune \ll N_all$:

通信性能を求められる通信相手プロセス数 (**N_tune**) が、全通信相手プロセス数 (**N_all**) と比較して、少ない場合

□ パターン2: $N_tune \approx N_all$:

通信相手となるほぼすべてのプロセス (**N_all**) に対し、均等に通信性能が要求される場合

パターン1や2に対して以降で示す手動チューニング結果が不十分な場合としてパターン3を導入する。

□ パターン3:

パターン1や2に対する手動チューニングで不十分な場合

上記パターンによって、チューニングの観点に違いがある。以降のスライドで、上記それぞれのパターンに応じたチューニングの指針を示す。

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶MCAパラメータの手動チューニング

■ パターン1に対する手動チューニング

□ パターン1: $N_{\text{tune}} \ll N_{\text{all}}$:

通信性能を求められる通信相手プロセス数 (N_{tune}) が、全通信相手プロセス数 (N_{all}) と比較して、少ない場合

■ パターン1に対するチューニングの指針

- MCAパラメータ①`common_tofu_max_fastmode_procs`により、高速型通信モードで通信を行うプロセス数の上限値を設定する。

※ 高速型通信モードで通信を行うプロセス数が上限値に達したことで、高速型通信モードによる通信に切り替えることができなかった通信相手（プロセス）との通信性能に注意する必要がある。

- 高速型通信モードおよび省メモリ型通信モードの割り当てプロセスが想定どおりにならない場合、MCAパラメータ②`common_tofu_fastmode_threshold`を使って、省メモリ型通信モードから高速型通信モードに切り替えられる通信回数を調整する。

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶MCAパラメータの手動チューニング

■ パターン2に対する手動チューニング

□ パターン2: $N_{\text{tune}} \approx N_{\text{all}}$:

通信相手となるほぼすべてのプロセス (N_{all}) に対し、均等に通信性能が要求される場合

■ パターン2に対するチューニングの指針

- MCAパラメータ③`common_tofu_large_recv_buf_size`により、高速型通信モードのLarge受信バッファのサイズを調整する。

※ ただし、データサイズが数KiBから数十KiBの場合は通信性能が一律に落ちることがあるので、注意してください。

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶MCAパラメータの手動チューニング

■ パターン3に対する手動チューニング

□ パターン3:

パターン1や2に対する手動チューニングで不十分な場合

■ パターン3に対するチューニングの指針

- MCAパラメータ①`common_tofu_max_fastmode_procs`および③`common_tofu_large_recv_buf_size`による調整に加え、MCAパラメータ④`common_tofu_medium_recv_buf_size`によって、省メモリ型通信モードのMedium受信バッファのサイズを調整する。
- それでも不十分な場合は、MCAパラメータ⑤`common_tofu_memory_saving_method`に2を指定して省メモリ型通信モードで使う方式としてShared受信バッファを使用する方式を選択し、必要に応じてMCAパラメータ⑥`common_tofu_shared_recv_buf_size`によって省メモリ型通信モードのShared受信バッファのサイズを調整する。

■ 「富岳」のMPI通信

■ メモリ使用量を抑えるための考慮点

本節の内容

■ メモリ使用量を抑えるためのMCAパラメータとそのチューニング

■ MCAパラメータの手動チューニング

■ MCAパラメータの自動チューニング

■ 自動チューニングを有効にするMCAパラメータ

ユーザがMPIプログラム自身の使用するメモリ量を知っている場合、MPIライブラリが使用可能なメモリ使用量をユーザが制限することで、処理系は内部的に各MCAパラメータ(*1)を自動的にチューニングし、可能な限り指定されたメモリ使用量の制限値の範囲で動作するようになる。

(*1) ①common_tofu_max_fastmode_procs、③common_tofu_large_recv_buf_size、
④common_tofu_medium_recv_buf_size、⑥common_tofu_shared_recv_buf_size

MPIライブラリが使用可能なメモリ使用量を制限するかどうかは、MCAパラメータ `common_tofu_memory_limit` で指定する。本パラメータに1以上の値を指定するとメモリ使用量の制限が有効になる。

■ `-mca common_tofu_memory_limit 1` 以上の整数値

MPIライブラリが使用可能なメモリ使用量の制限を有効にする。

本MCAパラメータの値にメモリ使用量の制限値 (MiB) を指定する。

上記(*1)のMCAパラメータ①③④⑥を自動チューニングの対象にする。

■ `-mca common_tofu_memory_limit 0`

メモリ使用量の制限を無効にする。-1以下の数値が指定された場合、0を指定したものとみなされる。

本パラメータの省略値は0MiBである。

「富岳」のMPI通信▶メモリ使用量を抑えるための考慮点

▶MCAパラメータの自動チューニング

■ 自動チューニング時の通信相手プロセス数

- MCAパラメータ`common_tofu_memory_limit`に1以上の値を指定した場合、MCAパラメータ`common_tofu_memory_limit_peers(*2)`で指定された値を通信相手プロセス数として使用する。

- (*2) ・ MCAパラメータ`common_tofu_memory_limit_peers`の省略値として、同じコミュニケーター`MPI_COMM_WORLD`に属するプロセスの数が設定される。
- ・ より正確な自動チューニングを行うためには、MPI統計情報によって得られるTofu通信のためのコネクション数（「Per-peer Communication Count」欄の「`Connection`」の行の値）を指定する必要がある。

■ 自動チューニング対象MCAパラメータの優先順位

■ MCAパラメータ①③④⑥は以下の優先順位で自動チューニングされる。

優先順位 (高→低)	MCAパラメータ(*3)
1	①common_tofu_max_fastmode_procs
2	③common_tofu_large_recv_buf_size
3	④common_tofu_medium_recv_buf_size(*4)
	⑥common_tofu_shared_recv_buf_size(*4)

(*3) これらMCAパラメータのうち二つ以下が同時に指定されている場合は、指定されたMCAパラメータについては指定値がそのまま有効となる。残りの指定されていないMCAパラメータだけを対象に、優先順位の高い順に自動チューニングされる。

(*4) MCAパラメータcommon_tofu_memory_saving_methodの値が1の場合は④が、2の場合は⑥が有効となる。

⚠ 自動チューニングの注意事項

- 処理系が最低限必要とするメモリ使用量は、並列プロセス数などの条件によって変わる。以下の場合メモリが使用されるが、これらのメモリ使用量について処理系は事前に把握できないため、ユーザが指定したメモリ使用量の制限値を超えてしまうことがある。

■ Unexpected messageが発生した場合

- この場合、Unexpected messageをMPIライブラリ内で待避させるために、Unexpected messageの量に応じたメモリが使用される。
- MPIライブラリは、MPIプログラム実行中にどれだけの量のUnexpected messageが発生するかを把握できないため、自動チューニングではUnexpected messageの発生が考慮されていない。

■ 動的プロセス生成を行った場合

■ コミュニケータを作成した場合

内容

- 「富岳」の概略
- MPIジョブ実行に関連する各種基本事項
- 「富岳」のストレージ構成とLLIO
- 「富岳」のMPI通信
- **【付録】 参考文献**

- [1] MPI使用手引書
- [2] ジョブ運用ソフトウェア エンドユーザ向けガイド
- [3] LLIO ユーザーズガイド
- [4] <https://www.r-ccs.riken.jp/fugaku/system/>

※ [1]～[3]はスーパーコンピュータ「富岳」のサイトにアクセス可能ならば以下のURLよりダウンロード可能です。

https://www.fugaku.r-ccs.riken.jp/docs/manuals_r01

以上