プロセスとスレッドの適切な割り当て
ノードグループB編

この資料について

- 玄界の各計算ノードグループには複数のCPUやGPUやNICが搭載されている
- 最大性能を得るためには、それらの配置を考えたプログラムの実行が必要

- この資料ではプロセスやスレッドの割り当てを最適化するためのヒントについて説明する
  - ノードグループB編では、前半はCPUを用いたOpenMPやMPIプログラムについて（ノードグループA編と類似の内容）、後半はMPIとGPUを用いたマルチGPUプログラム実行について解説する

- 2025年3月
  - ノードグループBのNUMA構成に誤りがあったため修正

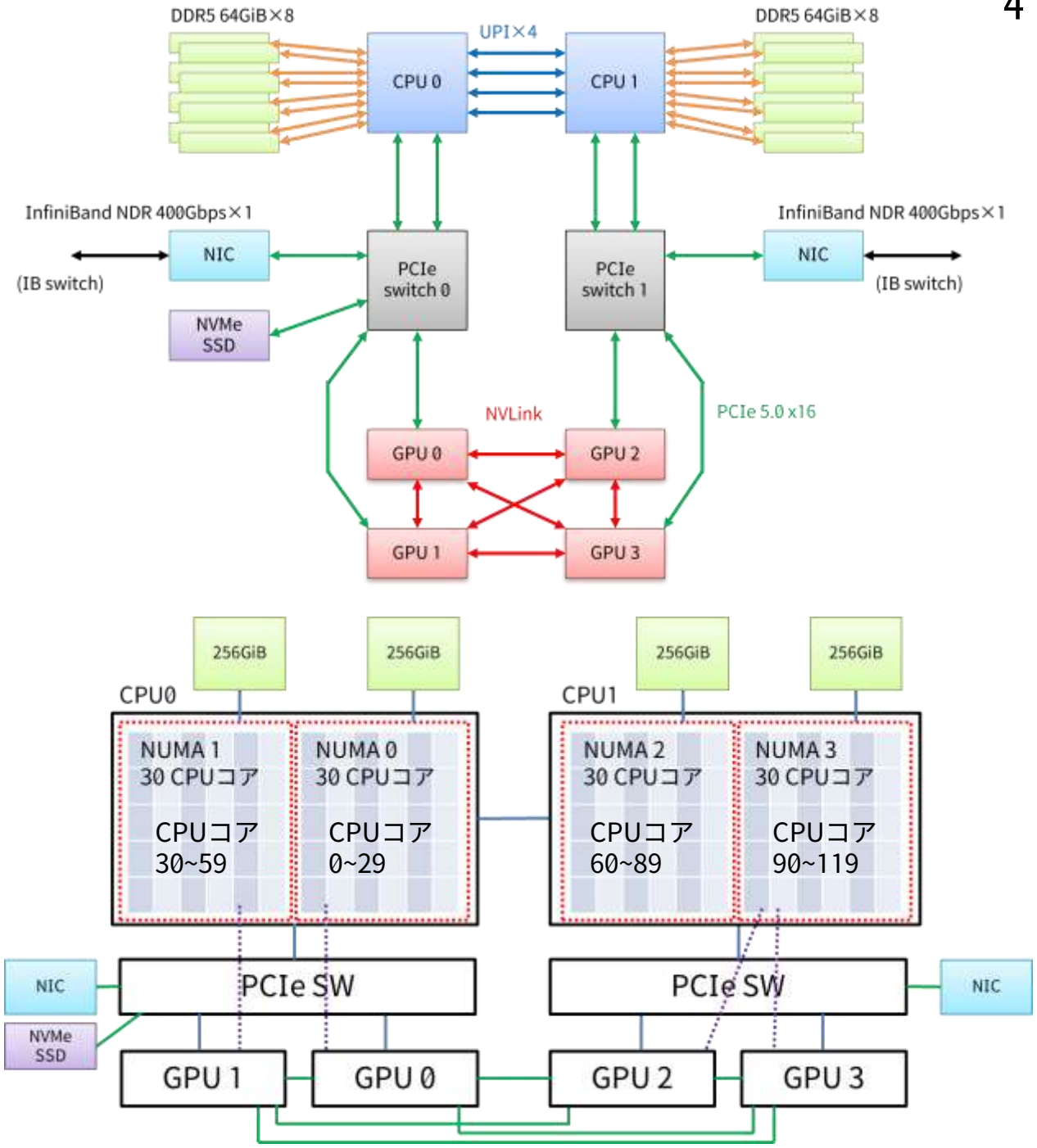
ハードウェア構成とNUMA構成の基礎知識

最適なプログラム実行のためには対象ハードウェアに関するある程度の知識が必要。

はじめに、計算ノードグループのハードウェア構成の概要と、ユーザ（プログラム）からどのように見えるのかについて説明する。

基本的な構成の理解

- 上図：ノードグループBのCPU・GPU・メモリ・NICの物理的な構成
  - メモリはそれぞれのCPUソケットにつながっているため、異なるCPUソケットにつながっているメモリにアクセスするには時間がかかる。
  - NICは各CPUに接続されている
- 下図：CPUの内部はHW的には4つのグループ（NUMAノード）に分かれているが、2つに分かれて見えるように設定している
  - 1NUMAノード＝1GPUの対応にしてある
  - NUMAノードをまたぐよりもまたがない方がメモリアクセスが高速（ソケットをまたぐのと比べると小さな差）
  - PCI-Expressの構成上、GPU2と3がNUMA3に接続（CPU-GPU間のデータ転送速度に10%程度の差）
- 実行時には `numactl -H`, `numactl -s`, `lstopo`, `nvidia-smi topo -m` などで構成を確認可能



ユーザからはどう見える？（numactl、1ノード占有ジョブ実行時）

- CPU2ソケットで合計4つのNUMAノードが確認できる

```
[ku40000105@b0001 ~]$ numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29
node 0 size: 257264 MB
node 0 free: 255033 MB
node 1 cpus: 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
node 1 size: 258040 MB
node 1 free: 255596 MB
node 2 cpus: 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
node 2 size: 258040 MB
node 2 free: 255749 MB
node 3 cpus: 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115 116
117 118 119
node 3 size: 257993 MB
node 3 free: 255938 MB
node distances:
node   0   1   2   3
  0:  10  12  21  21
  1:  12  10  21  21
  2:  21  21  10  12
  3:  21  21  12  10
```

0-1がCPU0、2-3がCPU1

```
[ku40000105@b0001 ~]$ numactl -s
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106
107 108 109 110 111 112 113 114 115 116 117 118 119
cpubind: 0 1 2 3
nodebind: 0 1 2 3
membind: 0 1 2 3
```

120コア、4NUMAノード

ユーザからはどう見える？（numactl、1GPUジョブ実行時）

- 確認方法によって見え方が違う

```
[ku40000105@b0031 ~]$ numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29
node 0 size: 257263 MB
node 0 free: 193697 MB
node 1 cpus: 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
node 1 size: 258040 MB
node 1 free: 255778 MB
node 2 cpus: 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
node 2 size: 257996 MB
node 2 free: 253796 MB
node 3 cpus: 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115 116
117 118 119
node 3 size: 258037 MB
node 3 free: 254286 MB
node distances:
node   0   1   2   3
  0:  10  12  21  21
  1:  12  10  21  21
  2:  21  21  10  12
  3:  21  21  12  10
```

```
[ku40000105@b0031 ~]$ numactl -s
policy: default
preferred node: current
physcpubind: 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89
cpubind: 2
nodebind: 2
membind: 0 1 2 3
```

CPU 30コア分の情報。
具体的なIDはジョブによって変わる。
（NUMAノードに対応。4パターン。
今回はCPU1の前半コア。）

numactl -s では使える資源のみが見える（30コア見える）
メモリノードだけはノード全体が見えてしまう（仕様）

numactl -Hではハードウェア情報が見えてしまうためノード占有ジョブと変化なし

nvidia-smi topo -m

```
[ku40000105@b0001 ~]$ nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    NIC0    NIC1    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      NV6     NV6     NV6     SYS     SYS     0-29    0               N/A
GPU1    NV6      X      NV6     NV6     PIX     SYS     30-59   1               N/A
GPU2    NV6     NV6      X      NV6     SYS     SYS     90-119  3               N/A
GPU3    NV6     NV6     NV6      X      SYS     PIX     90-119  3               N/A
NIC0    SYS     PIX     SYS     SYS      X      SYS
NIC1    SYS     SYS     SYS     PIX     SYS      X

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks

NIC Legend:

  NIC0: mlx5_0
  NIC1: mlx5_1
```

PCI-Expressの構成上、GPU2と3がNUMA3に接続されている
（GPU2のCPU-GPU間データ転送速度は他より10%程度低い。
実例は本資料の末尾にて紹介。）

```
[ku40000105@b0031 work]$ nvidia-smi topo -m
        GPU0    NIC0    NIC1    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      SYS     SYS             3               N/A
NIC0    SYS      X      SYS
NIC1    SYS     SYS      X
```

MIG利用時の注意事項

- MIGでは1GPUが7つのサブGPUに分割され、1ノード全体では28のサブGPUが利用可能
- ノードあたり物理CPUコアは60×2=120コア
- 単純計算すると1サブGPUあたり物理CPU数は120÷28=4.28……と中途半端になる
- 計算資源の分割と割り当てを行う際の利便性のため、120コア中の8コア（60コアあたり4コア）を無効化し、1サブGPUあたり4CPUコアが確保されるようにしている
- そのため、b-{batch|inter}-migで1ノードまるごと資源を確保すると、120コアではなく112コアのみ見える

ユーザからはどう見える？（numactl、MIG利用時）

- 確認方法によって見え方が違う：1サブGPUジョブの例

```
[ku40000105@b0036 ~]$ numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27
node 0 size: 257264 MB
node 0 free: 256140 MB
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54 55
node 1 size: 257996 MB
node 1 free: 256214 MB
node 2 cpus: 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83
node 2 size: 258040 MB
node 2 free: 254954 MB
node 3 cpus: 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111
node 3 size: 258037 MB
node 3 free: 255962 MB
node distances:
node   0   1   2   3
  0:  10  12  21  21
  1:  12  10  21  21
  2:  21  21  10  12
  3:  21  21  12  10
```

```
[ku40000105@b0036 ~]$ numactl -s
policy: default
preferred node: current
physcpubind: 0 1 2 3
cpubind: 0
nodebind: 0
membind: 0 1 2 3
```

1サブGPUジョブのためCPU 4コア分の情報。
具体的なIDはジョブによって変わる。
（今回はCPU0の先頭のコア群が割り当たった。）

numactl -s では使える資源のみが見える
メモリノードだけはノード全体が見えてしまう（仕様）

numactl -Hではハードウェア情報が見える（112コアの存在を確認できる）

GPUはどう見える？：ノード占有ジョブの場合

- ノード数を指定したジョブの場合はノード内のGPUが全て見える

```
[ku40000105@b0001 ~]$ nvidia-smi
Tue Jul 23 10:44:45 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.154.05             Driver Version: 535.154.05   CUDA Version: 12.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA H100                    On  | 00000000:1C:00.0 Off |                    0 |
| N/A   21C    P0             69W / 700W |      0MiB / 95830MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA H100                    On  | 00000000:2B:00.0 Off |                    0 |
| N/A   21C    P0             66W / 700W |      0MiB / 95830MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   2  NVIDIA H100                    On  | 00000000:AC:00.0 Off |                    0 |
| N/A   21C    P0             66W / 700W |      0MiB / 95830MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   3  NVIDIA H100                    On  | 00000000:BC:00.0 Off |                    0 |
| N/A   21C    P0             66W / 700W |      0MiB / 95830MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
[ku40000105@b0001 ~]$ nvidia-smi -L
GPU 0: NVIDIA H100 (UUID: GPU-c8bdcec7-1a54-f351-5167-2f65805df073)
GPU 1: NVIDIA H100 (UUID: GPU-44e7e22b-ac0f-6689-c571-e404bbdccc0b)
GPU 2: NVIDIA H100 (UUID: GPU-4974e6d2-5213-77b3-2873-41b45a6893f6)
GPU 3: NVIDIA H100 (UUID: GPU-090b1def-01ef-6f70-f713-faa425de01ea)
```

↑

このUUID情報（ GPU-c8bdcec7-1a54-f351-5167-2f65805df073 など）をCUDA_VISIBLE_DEVICESに与えてGPUの指定に使うことができる（単純にデバイス番号を与えても良い）

もちろん複数ノードの場合はノードごとに情報が見える

GPUはどう見える？：ノード非占有ジョブの場合

- GPU数を指定したジョブの場合は指定した数分だけのGPUが見える

```
[ku40000105@b0033 ~]$ nvidia-smi
Tue Jul 23 10:48:23 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.154.05             Driver Version: 535.154.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA H100                    On  | 00000000:1C:00.0 Off |                    0 |
| N/A   19C    P0              67W / 700W |      0MiB / 95830MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA H100                    On  | 00000000:2B:00.0 Off |                    0 |
| N/A   19C    P0              67W / 700W |      0MiB / 95830MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

```
[ku40000105@b0033 ~]$ nvidia-smi -L
GPU 0: NVIDIA H100 (UUID: GPU-962c241e-6966-98e3-d3d9-92a1ca6cdcfa)
GPU 1: NVIDIA H100 (UUID: GPU-5dc63e64-463d-2733-59be-fd08109eafd5)
```

↑

このUUID情報（ GPU-962c241e-6966-98e3-d3d9-92a1ca6cdcfa など）をCUDA_VISIBLE_DEVICESに与えてGPUの指定に使うことができる（単純にデバイス番号を与えても良い）

GPUはどう見える？：サブGPU（MIG）利用の場合

- サブGPU利用の場合も指定した数だけのGPU情報が見えるが、やや見た目が異なる

```
[ku40000105@b0036 ~]$ nvidia-smi
Tue Jul 23 10:54:17 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.154.05             Driver Version: 535.154.05   CUDA Version: 12.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA H100                    On  | 00000000:1C:00.0 Off |                   On |
| N/A   20C    P0              68W / 700W |                  N/A |     N/A      Default |
|                               |                      |               Enabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| MIG devices:                                                                |
+------------------+--------------------------------+-----------+-----------------------+
| GPU  GI  CI  MIG |                     Memory-Usage |        Vol|      Shared           |
|      ID  ID  Dev |                       BAR1-Usage | SM     Unc| CE ENC DEC OFA JPG    |
|                  |                                |        ECC|                       |
|==================+================================+===========+=======================|
|  0    7   0   0  |              12MiB / 11008MiB  | 16      0 |  1   0    1    0    1 |
|                  |               0MiB / 16383MiB  |           |                       |
+------------------+--------------------------------+-----------+-----------------------+
|  0    8   0   1  |              12MiB / 11008MiB  | 16      0 |  1   0    1    0    1 |
|                  |               0MiB / 16383MiB  |           |                       |
+------------------+--------------------------------+-----------+-----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

–L rscgrp=b-inter,gpu=2 の例

```
[ku40000105@b0036 ~]$ nvidia-smi -L
GPU 0: NVIDIA H100 (UUID: GPU-7e4e9dde-5f0a-b8ef-3606-0e7abd9c3b6d)
  MIG 1g.12gb     Device  0: (UUID: MIG-f7119b15-147a-5b68-a51b-a79cb75dec00)
  MIG 1g.12gb     Device  1: (UUID: MIG-dc2d8148-9fe3-5dce-845f-137297baf2eb)
```

↑

このUUID情報（ MIG-f7119b15-147a-5b68-a51b-a79cb75dec00 など）をCUDA_VISIBLE_DEVICESに与えてGPUの指定に使うことができる（単純にデバイス番号を与えても良い）

これは2サブGPUを指定した場合の例。nvidia-smiでサブGPUの状況を確認できる。複数サブGPUを要求した場合、同じ物理GPU上のサブGPUが確保されやすい構成にしてあるが、GPU資源の空き状況によっては異なる物理GPUにまたがってGPUが確保されることがある。

以下は4サブGPUを要求したら3サブGPUと1サブGPUに分かれていた例。

```
[ku40000105@b0036 ~]$ nvidia-smi -L
GPU 0: NVIDIA H100 (UUID: GPU-7e4e9dde-5f0a-b8ef-3606-0e7abd9c3b6d)
  MIG 1g.12gb     Device  0: (UUID: MIG-ef7aca2e-4ae7-5b10-a035-d354b2880e25)
  MIG 1g.12gb     Device  1: (UUID: MIG-8e1d20e1-52d5-56ae-8f80-cac7730966f6)
  MIG 1g.12gb     Device  2: (UUID: MIG-2f36b203-7660-56a7-9809-2eff92b70e71)
GPU 1: NVIDIA H100 (UUID: GPU-3861bd5b-d7fd-c1b0-0e0f-5edff930b7a5)
  MIG 1g.12gb     Device  0: (UUID: MIG-163f9ae0-e06f-5ac5-b977-3fb458ae9862)
```

何を考える必要がある？　1/2

- CPU関係
  - CPUとメモリを同じNUMAノードに配置すること
    - （プログラムによっては、あえてこれをやらない方が良いこともある）
  - ノードごとにデータを集約してからノード間通信をする場合は、NICが接続されている側のCPUに集約すること
  - プロセスやスレッドの配置順序
    - NUMAノード内で連続になるように配置するか、広く分散させるか
- GPU関係
  - GPUをどのように使うか。例えば……
    - 1MPIプロセスあたり1GPUを担当　→　MPIプロセスの配置方法を考える必要がある
      - ↑本資料ではこの使い方を想定
    - 1OpenMPスレッドあたり1GPUを担当　→　OpenMPスレッドの配置方法を考える必要がある
    - 1スレッドで複数GPUを操作　（最適な配置を行いにくい。どちらかというと非推奨。）

何を考える必要がある？　2/2

- ファーストタッチ（主にOpenMPプログラム）
  - 対象変数（配列）に対するキャッシュは、対象変数に最初にアクセスしたスレッドの存在する計算コアのキャッシュに割り当てられる
  - 並列計算ループ（主な計算処理）で行うのと同じ配列アクセスパターンの初期化ループを用意して実行しておくと、主な計算時に適切にキャッシュが使われて最大性能が得られる（低下を防げる）
  - 例

```
// 対象配列に対するアクセスが初めて生じる位置に追加する。
// 実際に計算をするループと同じスレッド割り当てが行われる必要があるため、
// schedule(dynamic)などを使うと効果が得られない。
#pragma omp parallel for
for(i=0; i<N; i++){
  dst[i] = srt[i] = 0;
}

// 本来のデータ初期化処理（省略）

// 実際に計算をするループ
#pragma omp parallel for
for(i=0; i<N; i++){
  dst[i] = src[i] * value;
}
```

考えられるプログラム実行パターン 1/2

1. ノード全体を占有するジョブ
   - node=ノード数 指定を行ってジョブを実行（pjsub）
     - もしくはb-{batch|inter}でgpu=4、b-{batch|inter}-migでgpu=28を指定してジョブを実行
2. ノードの一部のみを使う
   - gpu=GPU数 指定を行ってジョブを実行（pjsub）
   - ノードの一部しか使えないため、細かい資源割り当ての最適化は不可能（本資料では扱わない）

考えられるプログラム実行パターン 2/2

- CPU利用の観点から
  1. OpenMPのみ
     1. ノード全体に均等にスレッドを配置して実行
     2. CPUソケットやNUMAノードを限定し、その範囲内で均等にスレッドを配置して実行
        （※ノードを占有する意味があまりない。性能比較用などに。）
  2. MPIのみ
     1. ノード全体に均等にプロセスを配置して実行
  3. MPI + OpenMP
     1. NUMAノードごとにMPIプロセスを配置、NUMAノード内でOpenMP並列化
     2. CPUソケットごとにMPIプロセスを配置、CPUソケット内でOpenMP並列化
- GPU利用の観点から
  - NUMAノードごとにMPIプロセスを配置し、各MPIプロセスが1GPUを制御する実行形態を想定

numactlによるプログラムの最適化（実行方法の調整）

- numactl
  - 対象プログラムをどのNUMAノードやコア上で実行させるかや、どのメモリを使わせるかなどを指定するプログラム
  - MPIプログラムやOpenMPプログラムにも影響し、挙動を細かく調整することができる

- numactlの使い方
  - numactl オプション 対象プログラム　の形式でプログラムを実行する
  - 主なオプション
    - --cpunodebind/-N　スレッドを配置するNUMAノード番号を指定
    - --physcpubind/-C　スレッドを配置するCPUコア番号を指定
    - --membind/-m　指定したNUMAノードに接続されたメモリを使う
    - --interleave/-i　指定した範囲のメモリを均等に使う
    - --localalloc/-l　計算コアに近いメモリのみを使う
      - 多くの場合は--localallocが高速と思われるが、大容量のメモリを使う場合などに--interleave=allなどを活用すると良い

numactlの利用例　1/2

- NUMAノード0上で実行、メモリもNUMAノード0に接続されたもののみ利用
  - NUMAノード単位で指定する例
    - numactl -N 0 -l ./a.out
    - numactl --cpunodebind=0 --localalloc ./a.out
  - CPUコア単位で指定する例
    - numactl -C 0-14 -l ./a.out
    - numactl --physcpubind=0-14 --localalloc ./a.out
- CPUソケット0上で実行、メモリもCPUソケット0に接続されたもののみ利用
  - numactlで指定できるのはコアとNUMAノードのみのため、ソケット単位の指定は対応するコアやNUMAノードの組み合わせで指定する
  - numactl -N 0-1 -l ./a.out
  - numactl --cpunodebind=0-1 --localalloc ./a.out
  - numactl -C 0-59 -l ./a.out
  - numactl --physcpubind=0-59 --localalloc ./a.out

numactlの利用例　2/2

- CPUソケット0上で実行するが、メモリはノード全体を使う
  - 多くの容量のメモリを使いたい場合に有効
  - numactl -N 0-1 -i all ./a.out
  - numactl --cpunodebind=0-1 --interleave=all ./a.out

- 対象プログラムとして「numactl -s」を実行すると割り当て情報を確認しやすい
  - numactl -N 0 -l numactl -s
  - 結果

```
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
cpubind: 0
nodebind: 0
membind: 0 1 2 3
```

- NUMAノード0のCPU情報のみが見えるようになった
- localalloc指定してもmembindの表示は変わらない点には注意（実際には0だけが使われるはずである）

OpenMPプログラムの具体的な実行方法

（CPUによるスレッド並列化について。ノードグループBではGPUの利用がメインとなると思うが、CPUも全く利用しないわけではないはずなので説明する。）

OpenMPプログラムの実行を考える

- 問題設定（想定する実行条件）
  - ノードを占有したプログラム実行を対象としているため、ある程度多くのスレッドを用いた実行を想定
  - メモリは numactl --localalloc を指定しCPUコアに近いものを使用（必要に応じて変更して良い）

- 考えること
  - スレッドをどのCPUコアに割り当てるか　および　割り当てる順序をどうするか

- どのような最適化が可能か（最適化に使える推奨の手段は何があるか）
  - OpneMPの仕様に定められた環境変数
  - コンパイラごとの環境変数
  - numactlコマンド（解説済み）

OpenMPプログラムの制御方法

- OpenMPの仕様に定められた環境変数
  - OMP_PROC_BIND、OMP_PLACES
    - OMP_PROC_BIND=CLOSE　スレッドを隣接するCPUコアに割り当てる
    - OMP_PROC_BIND=SPREAD　スレッドを全体的に均等に割り当てる
    - OMP_PLACES　コア番号を指定するなどして詳細に割り当てる

- コンパイラごとの主要な環境変数
  - GNU：GOMP_CPU_AFFINITY
    - 使いたいコアの番号を指定する
  - Intel：KMP_AFFINITY
    - 割りあての方針を指定する
    - verboseを加えておくとデバッグに便利
      - 例：`export KMP_AFFINITY=verbose`
  - NVIDIA(PGI)：MP_BIND, MP_BLIST
    - 使いたいコアの番号を指定する

- ノードグループBではGNUコンパイラやNVIDIAコンパイラを使うことが多いと思うため、これらを中心にいくつか例を示す

OpenMPプログラムの実行例を考える

- ノード全体でOpenMPプログラムを実行する場合、基本的には各NUMAノードに同程度の数のスレッドを配置して実行したいはずである
  - たくさんのメモリを使いたいためにノード単位でジョブを実行しているとしても、一部のNUMAノードだけ集中的に使う必要性は低いだろうと仮定
  - もしNUMAノードへの配置を全く気にしないのであれば、スレッド数のみ指定して実行すれば十分
  - この場合、どのNUMAノード・どのCPUコアから利用するかはシステム依存となる（制御不能）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
export OMP_NUM_THREADS=32
./a.out
```

そもそもどのようなスレッド配置順が適切か

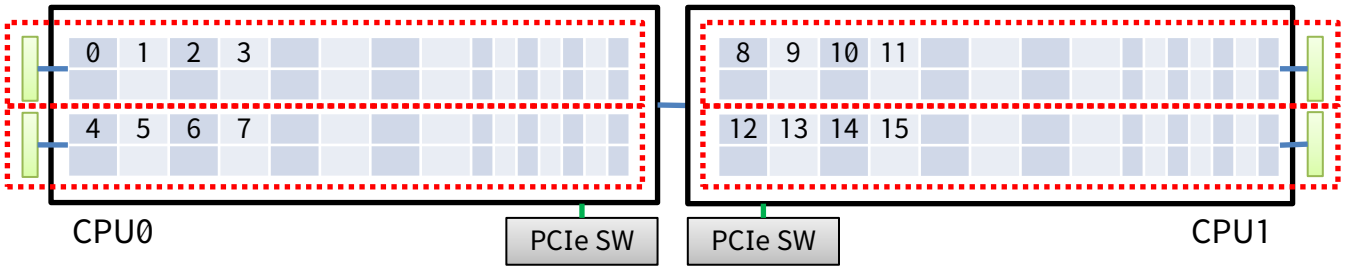
- 究極的には問題依存
- 一般的には……
  - スレッド数が少ない場合、メモリ性能を活かすためには全体に分散して配置、L3キャッシュを活かすには同一CPUソケットに配置するのが良い
  - スレッド数が多い場合、全体的に均等に配置するのが良い
    - 連続したスレッドIDを有するスレッドを近い位置に配置するか、
      連続したスレッドIDを有するスレッドを分散配置するか、の選択肢がある

- 近い位置に配置するか分散配置するかを考える際に、CPUソケットやNUMAノードを意識する必要がある

OpenMPプログラムの実行例：配置例1

- 配置例1
  - NUMAノードに必要数分のスレッドを配置してから、次のNUMAノードに配置する
  - 16スレッド実行を例に考える（全120コアに対して少ないが、図で例示する手間の都合上）
    - 注意点：均等な距離にスレッドを配置するような指定ではうまくいかないことがある
      - export OMP_PROC_BIND=SPREAD　などではうまくいったりいかなかったりする
      - （120をスレッド数で割った際に余りが出ると配置がズレてしまうのが原因と思われる）

狙いたい配置イメージ→



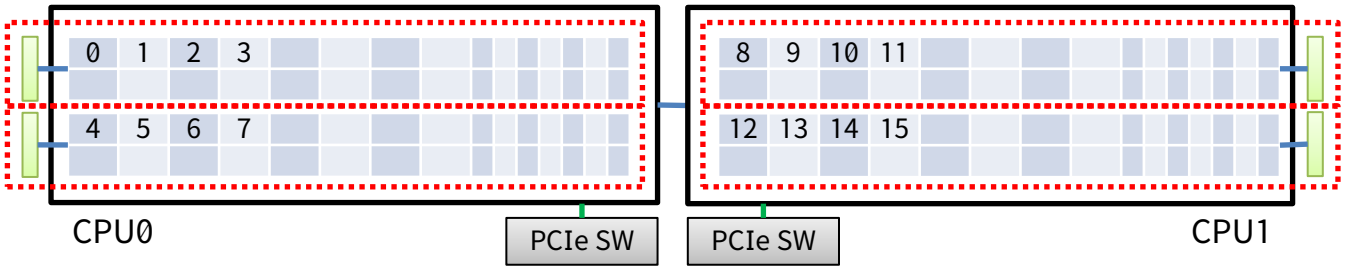
- NUMAノード内のスレッドの配置や順序は上図とある程度異なっていても良いし、NUMAノード毎に違っても構わない（有意な性能差は生じないはず）

- （ハードウェア的には最大4NUMAノードのCPUをGPU配置にあわせて2NUMAノードで利用しているため、CPUの性能を最大限に発揮させるなら4NUMAノードに対応したCPUコアを均等に使うのがベストと思われる）

OpenMPプログラムの実行例：配置例1　実行例1

- 実行例1：環境変数 OMP_PLACES で明示的に指定する
  - export OMP_PLACES="{0},{1},{2},{3},{30},{31},{32},{33},{60},{61},{62},{63},{90},{91},{92},{93}"
  - Intelコンパイラ、GNUコンパイラ、NVIDIAコンパイラ共通に使える
  - {} で括ったコア番号にスレッドが1つずつ配置される
  - やや面倒だが確実で自由度も高い

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
必要に応じmodule load
export OMP_NUM_THREADS=16
export OMP_PLACES="{0},{1},{2},{3},{30},{31},{32},{33},{60},{61},{62},{63},{90},{91},{92},{93}"
numactl --localalloc./a.out
```



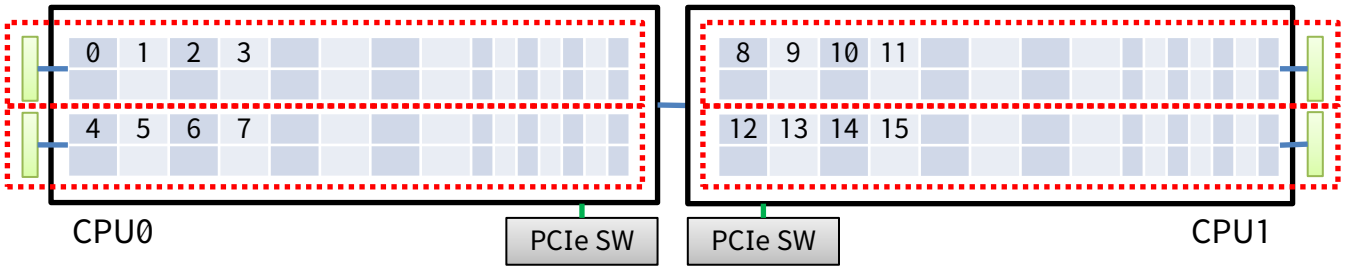
スレッド0から順に、OMP_PLACESで指定された番号のCPUコアに割り当てられる

スレッドID	0-3	4-7	8-11	12-15
CPUコアID	0,1,2,3	30,31,32,33	60,61,62,63	90,91,92,93
NUMAノード番号	0	1	2	3

OpenMPプログラムの実行例：配置例1　実行例2

- 実行例2：環境変数 GOMP_CPU_AFFINITY で明示的に指定する
  - export GOMP_CPU_AFFINITY="0-3,15-18,30-33,45-48,60-63,75-78,90-93,105-108"
    - 0,1,2,3, 30,31,32,33, 60,61,62,63, 90,91,92,93 のようにべた書きしても同様
  - Intelコンパイラと GNUコンパイラで共通して使える
  - IntelコンパイラではKMP_AFFINITYで書いても良い（内部的にはGOMP~はKMP~のエイリアス）
  - export KMP_AFFINITY=verbose,granularity=fine,proclist=[0-3,30-33,60-63,90-93],explicit

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load intel
export OMP_NUM_THREADS=16
export GOMP_CPU_AFFINITY="0-3,30-33,60-63,90-93"
export KMP_AFFINITY=verbose
numactl --localalloc ./a.out
```



スレッド0から順に、GOMP_CPU_AFFINITYで指定された番号のCPUコアに割り当てられる

スレッドID	0-3	4-7	8-11	12-15
CPUコアID	0,1,2,3	30,31,32,33	60,61,62,63	90,91,92,93
NUMAノード番号	0	1	2	3

KMP_AFFINITYにverboseを指定した場合の出力例 1

```
export OMP_NUM_THREADS=16
export OMP_PLACES="{0},{1},{2},{3},{30},{31},{32},{33},{60},{61},{62},{63},{90},{91},{92},{93}"
export KMP_AFFINITY=verbose
numactl --localalloc ./a.out
```
の出力例

```
OMP: Info #172: KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 1 maps to socket 0 core 1 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 2 maps to socket 0 core 2 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 3 maps to socket 0 core 3 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 4 maps to socket 0 core 4 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 5 maps to socket 0 core 5 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 6 maps to socket 0 core 6 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 7 maps to socket 0 core 7 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 8 maps to socket 0 core 8 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 9 maps to socket 0 core 9 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 10 maps to socket 0 core 10 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 11 maps to socket 0 core 11 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 12 maps to socket 0 core 12 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 13 maps to socket 0 core 13 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 14 maps to socket 0 core 14 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 15 maps to socket 0 core 15 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 16 maps to socket 0 core 16 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 17 maps to socket 0 core 17 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 18 maps to socket 0 core 18 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 19 maps to socket 0 core 19 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 20 maps to socket 0 core 20 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 21 maps to socket 0 core 21 thread 0
……
```

「proc N」が「コア番号N」に対応しているというシンプルな関係性

```
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 70 thread 0 bound to OS proc set 0
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 71 thread 1 bound to OS proc set 1
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 72 thread 2 bound to OS proc set 2
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 73 thread 3 bound to OS proc set 3
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 74 thread 4 bound to OS proc set 30
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 75 thread 5 bound to OS proc set 31
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 76 thread 6 bound to OS proc set 32
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 77 thread 7 bound to OS proc set 33
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 78 thread 8 bound to OS proc set 60
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 79 thread 9 bound to OS proc set 61
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 80 thread 10 bound to OS proc set 62
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 81 thread 11 bound to OS proc set 63
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 82 thread 12 bound to OS proc set 90
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 83 thread 13 bound to OS proc set 91
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 84 thread 14 bound to OS proc set 92
OMP: Info #255: OMP_PROC_BIND: pid 70 tid 85 thread 15 bound to OS proc set 93
……
```

スレッド0がproc set 0 = コア番号0上で実行される
スレッド4がproc set 30 = コア番号30上で実行される　といった関係性がわかる

KMP_AFFINITYにverboseを指定した場合の出力例 2

```
export OMP_NUM_THREADS=16
export GOMP_CPU_AFFINITY="0-3,30-33,60-63,90-93"
export KMP_AFFINITY=verbose
numactl --localalloc ./a.out
```

の出力例

```
OMP: Info #172: KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 1 maps to socket 0 core 1 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 2 maps to socket 0 core 2 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 3 maps to socket 0 core 3 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 4 maps to socket 0 core 4 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 5 maps to socket 0 core 5 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 6 maps to socket 0 core 6 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 7 maps to socket 0 core 7 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 8 maps to socket 0 core 8 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 9 maps to socket 0 core 9 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 10 maps to socket 0 core 10 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 11 maps to socket 0 core 11 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 12 maps to socket 0 core 12 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 13 maps to socket 0 core 13 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 14 maps to socket 0 core 14 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 15 maps to socket 0 core 15 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 16 maps to socket 0 core 16 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 17 maps to socket 0 core 17 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 18 maps to socket 0 core 18 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 19 maps to socket 0 core 19 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 20 maps to socket 0 core 20 thread 0
OMP: Info #172: KMP_AFFINITY: OS proc 21 maps to socket 0 core 21 thread 0
……
```

「proc N」が「コア番号N」に対応しているというシンプルな関係性

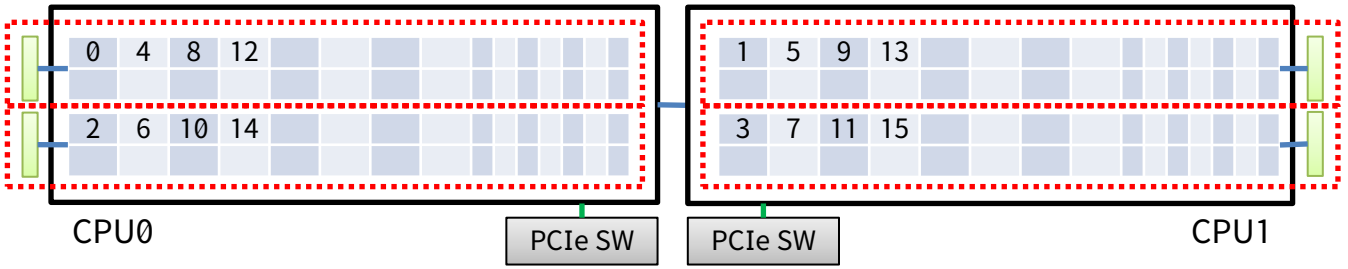
```
OMP: Info #255: KMP_AFFINITY: pid 70 tid 70 thread 0 bound to OS proc set 0
OMP: Info #255: KMP_AFFINITY: pid 70 tid 71 thread 1 bound to OS proc set 1
OMP: Info #255: KMP_AFFINITY: pid 70 tid 72 thread 2 bound to OS proc set 2
OMP: Info #255: KMP_AFFINITY: pid 70 tid 73 thread 3 bound to OS proc set 3
OMP: Info #255: KMP_AFFINITY: pid 70 tid 74 thread 4 bound to OS proc set 30
OMP: Info #255: KMP_AFFINITY: pid 70 tid 75 thread 5 bound to OS proc set 31
OMP: Info #255: KMP_AFFINITY: pid 70 tid 76 thread 6 bound to OS proc set 32
OMP: Info #255: KMP_AFFINITY: pid 70 tid 77 thread 7 bound to OS proc set 33
OMP: Info #255: KMP_AFFINITY: pid 70 tid 78 thread 8 bound to OS proc set 60
OMP: Info #255: KMP_AFFINITY: pid 70 tid 79 thread 9 bound to OS proc set 61
OMP: Info #255: KMP_AFFINITY: pid 70 tid 80 thread 10 bound to OS proc set 62
OMP: Info #255: KMP_AFFINITY: pid 70 tid 81 thread 11 bound to OS proc set 63
OMP: Info #255: KMP_AFFINITY: pid 70 tid 82 thread 12 bound to OS proc set 90
OMP: Info #255: KMP_AFFINITY: pid 70 tid 83 thread 13 bound to OS proc set 91
OMP: Info #255: KMP_AFFINITY: pid 70 tid 84 thread 14 bound to OS proc set 92
OMP: Info #255: KMP_AFFINITY: pid 70 tid 85 thread 15 bound to OS proc set 93
……
```

スレッド0がproc set 0 = コア番号0上で実行される
スレッド4がproc set 30 = コア番号30上で実行される　といった関係性がわかる

OpenMPプログラムの実行例：配置例2

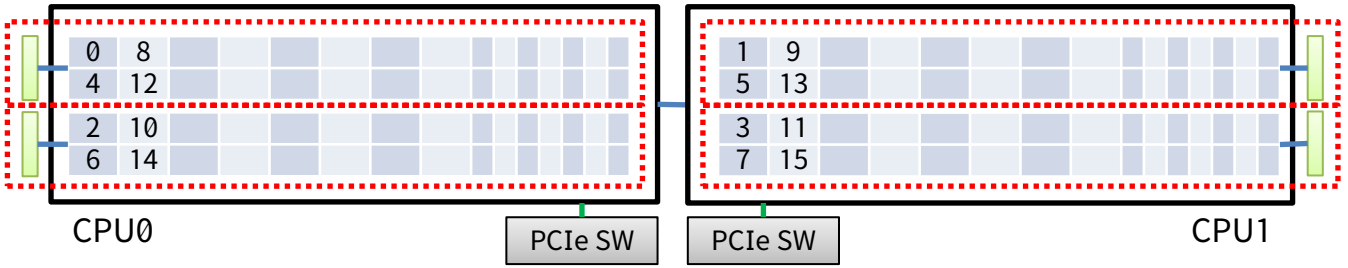
- 実行例2
  - 各NUMAノードに1スレッドずつ配置し、配置し終えたら2スレッド目を配置する
  - やはり配置するコアを明示的に指定するのが確実

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
必要に応じてmodule load
export OMP_NUM_THREADS=16
export OMP_PLACES="{0},{60},{30},{90},{1},{61},{31},{91},{2},{62},{32},{92},{3},{63},{33},{93}"
numactl --localalloc /a.out
```



本来の4NUMAを考慮するとこのような
割り当ての方が良い可能性がある？
（未調査、配置例1も同様の可能性あり？）

```
export OMP_PLACES="{0},{60},{30},{90},{15},{75},{45},{105},{1},{61},{31},{91},{16},{76},{46},{106}"
```



OpenMPプログラムの実行例：補足1

- もっと簡単な指定方法はないのか？
  - 配置例1について
    - export OMP_PLACES="{0:4}:4:30" （16スレッドではない場合は4をNUMAあたり利用コア数（切り上げ）に変更）でも同様の配置ができるようだが、割りきれない数の場合などに気持ち悪い配置がされてしまうようなので利用時には注意（確認）が必要
  - 配置例2について
    - いまのところ不明
    - KMP_AFFINTYでscatterを使うなどが良さそうに思ったが、NUMAノード単位ではなくソケット単位で分散配置されてしまった
- 特定のNUMAノードやCPUソケット内にのみスレッドを配置したい場合は？
  - やはり OMP_PLACES や GOMP_CPU_AFFINITY でコア番号を明確に指定するのがシンプルで確実
  - numactlでNUMAノードやCPUコアを指定しても良い
    - CPUコアを指定すれば完全に同じ配置ができる（記述量もほぼ変わらない）
- Intelコンパイラ以外で（プログラム内から）配置を確認する方法は？
  - 資料末尾の参考情報を参照

OpenMPプログラムの実行例：補足2

- CPU内のNUMAの構成まで細かく考えなくて良いのであれば
  OMP_PROC_BINDでcloseやspread、
  KMP_AFFINITYでcompactやscatterを指定する程度でも良い

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00

export OMP_PROC_BIND=close
export OMP_PROC_BIND=spread

module load intel
export KMP_AFFINITY=verbose,granularity=fine,compact
export KMP_AFFINITY=verbose,granularity=fine,scatter
export KMP_AFFINITY=verbose,granularity=fine,balanced

export OMP_NUM_THREADS=60

numactl --localalloc ./a.out
```

※ closeやspreadはgnuもintelも共通に利用可能。scatter的かつNUMAノードに均等に配置したい場合などはOMP_PLACESで細かく指定する必要があるようだ。

- CPUコアを端から順に埋める
- CPUコアを端から全体的に均等に埋める（小さな番号のコアに小さな番号のスレッドが配置される）

- CPUコアを端から順に埋める
- scatterやbalancedではCPUソケット単位で均等な配置になってしまうようなので注意が必要（NUMAノード0と4に優先して配置されてしまう）

ノード共有実行の場合は？

- （gpu<4 を指定して実行したノードの一部の資源のみが使えるジョブの場合）
  - NUMAノード単位で資源が割り当てられる
  - ジョブスクリプト内でnumactl -sの結果などを参照すれば実際に使えるCPUコア番号などが確認できる
  - 得られた情報をもとに細かい資源割り当てを行っても良いが、実行の度に割り当てが異なる可能性が高いため注意が必要
  - スレッド数だけ指定して実行すれば十分なのでは

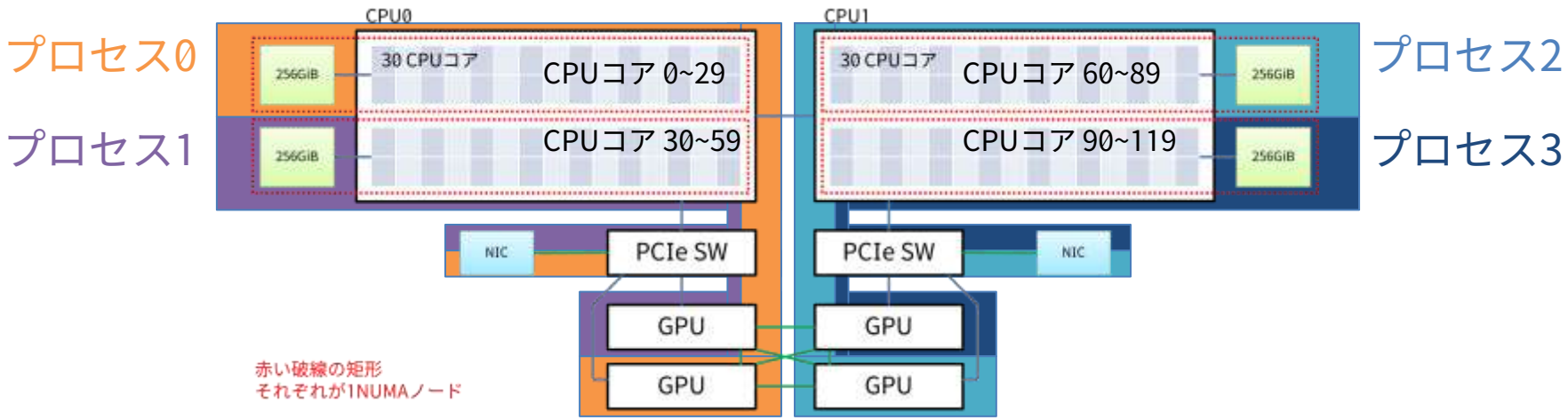
- （MIGを使ったジョブの場合）
  - NUMAノードをまたいで資源が割り当てられることもある
  - 細かいことは考えずにスレッド数だけ指定して実行するのが妥当

GPU+MPIプログラムの具体的な実行方法（Open MPI）

MPIプログラムおよびMPI+OpenMPプログラムの実行を考える

- ノードグループBではGPUを使うジョブが多く、GPUを利用する際はOpen MPIを使うことが多いと思われるため、本資料ではOpen MPIについてのみ解説する。

- 前提（問題設定）
  - ノードグループBは「1NUMAあたり1プロセスを配置、1プロセスあたり1GPUを担当」という実行形態を想定した設定にしてあるため、これにあわせた実行のみ扱う。
  - 配置例イメージ
    - 同じ色の枠の部分が1つのプロセス



（通信パターンによってはプロセス1と2は逆順が良いこともあるかもしれない）

MPIを実行するための準備　1/4

- CUDA Fortran, OpenACC, GPU向けOpenMPを使う場合はnvidiaモジュールのloadが必要。さらにOpen MPIを使う場合はnvhpcxモジュールのloadも必要。さらにNCCLも使いたい場合はcudaとncclのloadも必要。

```
$ module load nvidia nvhpcx cuda nccl
$ which mpicc
/home/app/nvhpc/23.9/Linux_x86_64/23.9/comm_libs/12.2/hpcx/hpcx-2.16/ompi/bin/mpicc
$ mpicc -v
Export NVCOMPILER=/home/app/nvhpc/23.9/Linux_x86_64/23.9
Export PGI=/home/app/nvhpc/23.9
nvc-Warning-No files to process
```

MPIを実行するための準備　2/4

- CUDA Cを使う場合はcudaのloadが必要。さらにOpen MPIを使う場合はgccとhpcxのloadが必要。さらにNCCLも使いたい場合はncclのloadも必要。

```
$ module load cuda gcc hpcx nccl
$ which mpicc
/home/app/hpcx/2.17.1/ompi/bin/mpicc
$ mpicc -v
Using built-in specs.
COLLECT_GCC=/usr/bin/gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/8/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-redhat-linux
Configured with: ../configure --enable-bootstrap --enable-languages=c,c++,fortran,lto --prefix=/usr --mandir=/usr/share/man --
infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared --enable-threads=posix --enable-
checking=release --enable-multilib --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-
object --enable-linker-build-id --with-gcc-major-version-only --with-linker-hash-style=gnu --enable-plugin --enable-initfini-
array --with-isl --disable-libmpx --enable-offload-targets=nvptx-none --without-cuda-driver --enable-gnu-indirect-function --
enable-cet --with-tune=generic --with-arch_32=x86-64 --build=x86_64-redhat-linux
Thread model: posix
gcc version 8.5.0 20210514 (Red Hat 8.5.0-18) (GCC)
```

MPIを実行するための準備　3/4

- NGCのHPC-benchmarksコンテナを使う場合にhpcx系ではなく gcc ompi 系をloadしないと動かないケースがあった

https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks

```
$ module load nvidia nvhpcx cuda nccl          NG
$ module load gcc hpcx cuda                    NG
$ module load cuda gcc hpcx nccl               NG
$ module load cuda/11.8.0 gcc/8 ompi/4.1.6     OK
$ module load cuda/12.2.2 gcc/8 ompi/4.1.6     OK
# HPC-benchmarksコンテナに含まれるHPLベンチマークを実行
$ mpirun -v -n 4 --map-by numa --bind-to numa --rank-by numa ¥
singularity exec --nv ~/container/hpc-benchmarks_24.03.sif /workspace/hpl-linux-x86_64/hpl.sh ¥
--dat /workspace/hpl-linux-x86_64/sample-dat/HPL-4GPUs.dat
```

- NGのmodule構成だと何も表示されずに時間だけが経過してしまう
- 24.06コンテナだとメモリアドレス関係のエラーが出てしまう（おそらくコンテナの期待しているCUDAやドライバよりも玄界側が古いため）

MPIを実行するための準備　4/4

- GPU向けOpen MPIについてはいくつかの構成が利用可能
- どれが最適かは調査中
- 基本的な使い方は変わらないはず

- 基本的には hpcx/nvhpcx 系が推奨だが、問題がある場合は ompi 系も試すのが良い？

Open MPIを使う：基礎編

- MPIプログラムの実行はmpirunまたはmpiexecで行う（どちらでも一緒）
- 主にmpirunへの引数で動作を制御する
  - プロセス数は実行時オプション-n
  - ノードあたりプロセス数は実行時オプション-npernode
  - プロセスの配置方法は実行時オプション--map-byや--bind-to
  - etc.
- -display-mapオプションや-display-devel-mapを使うと割り当て情報を確認可能

Open MPIを使う：出力ファイルの制御

- 標準出力・エラー出力をプロセスごとに別のファイルに書き出すには？
  - -output-filename 引数で出力ファイルを変更可能
    - 引数にはディレクトリ名を指定する。その下に
      1/rank.ランク番号/stdout　標準出力の出力先
      1/rank.ランク番号/stderr　標準エラー出力の出力先
      が作られる。
  - 例：TCS（バッチジョブシステム）のジョブIDを用いて重複しない名前のディレクトリに出力

```
mpirun -display-map -display-devel-map ¥
 -n 4 --map-by socket --bind-to socket ¥
 --rank-by node ¥
 -output-filename out_${PJM_JOBID} ./a.out
```

- ジョブスクリプト（bashスクリプト）は行末にバックスラッシュ
  （PowerPoint上では表示の都合で円記号）を書くことで行の折り返しが可能

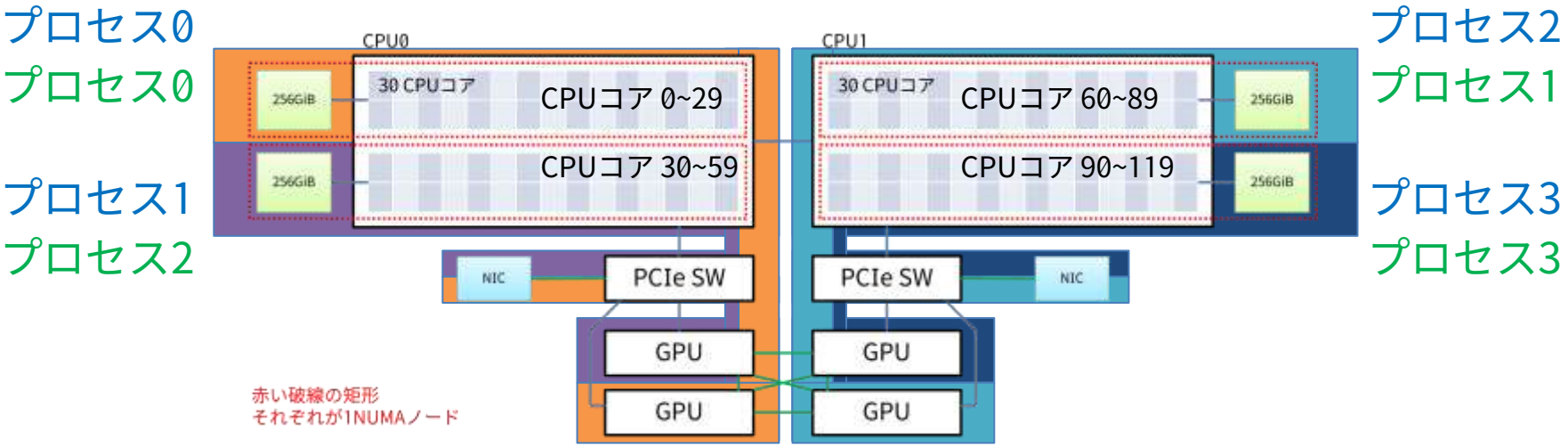
Open MPIによる1ノード内複数プロセス実行：基本的な考え方

- プロセスの配置方法は主にmpirunの引数で制御
  - --map-by, --rank-by：どのような順序でプロセスを配置するか
    - --map-by numa：各NUMAノードに横断的にプロセスを配置する
    - --map-by ppr:X:numa：NUMAノードへXプロセスを配置してから次のNUMAノードへ
    - --rank-by socket, --rank-by node：MPIランク番号の振り方の指定
  - --bind-to：プロセスをどのような単位で配置するか
    - core, numa, socket などの単位でプロセスを割り付ける

例：4プロセスフラットMPI

- 各NUMAノードに1プロセスずつ配置し、各プロセスがNUMAノード内のメモリや近いGPUにアクセスする

- 主なプロセス配置戦略
  - 青文字（先にCPU0側を埋める） or 緑文字（CPUに交互に配置する）



ジョブスクリプトの書き方例

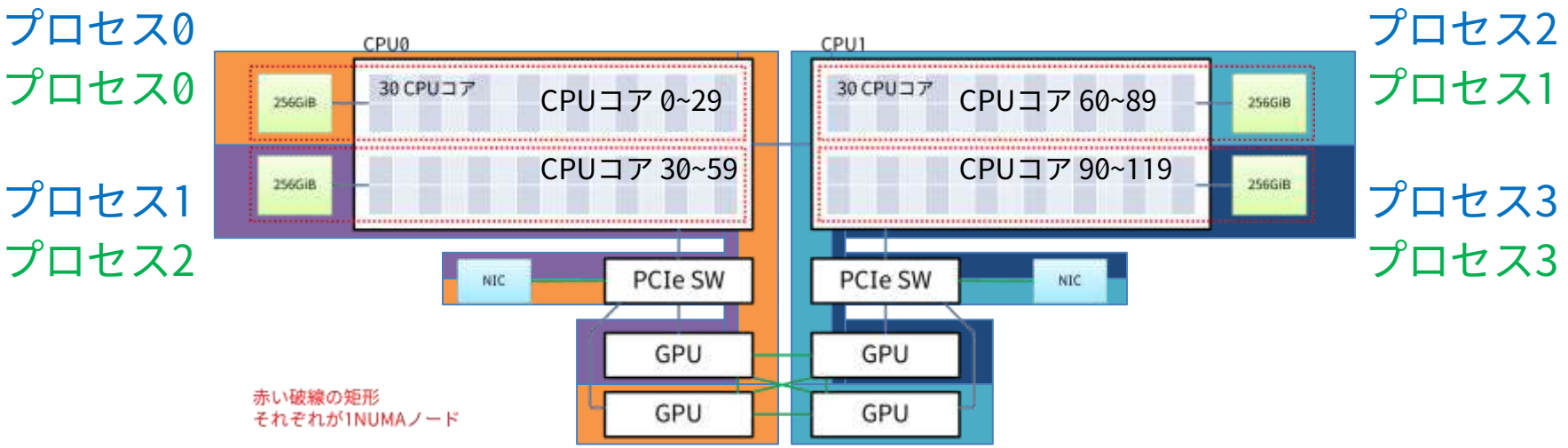
青文字の例（先にCPU0を埋める）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=1
mpirun -display-map -display-devel-map -n 4 ¥
--map-by numa --bind-to core --rank-by numa ¥
numactl -l ./a.out
```

※ --rank-by オプションを書かなかった場合もこちらの挙動

緑文字の例（CPUに交互に配置する）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=1
mpirun -display-map -display-devel-map -n 4 ¥
--map-by numa --bind-to core --rank-by socket ¥
numactl -l ./a.out
```



-display-devel-map 指定時に得られる出力（割り当て情報）

CPUソケット0 | CPUソケット1

NUMA 0 | NUMA 1 | NUMA 2 | NUMA 3

プロセス0（ランク0）＝コア0
プロセス1（ランク1）＝コア30
プロセス2（ランク2）＝コア60
プロセス3（ランク3）＝コア90

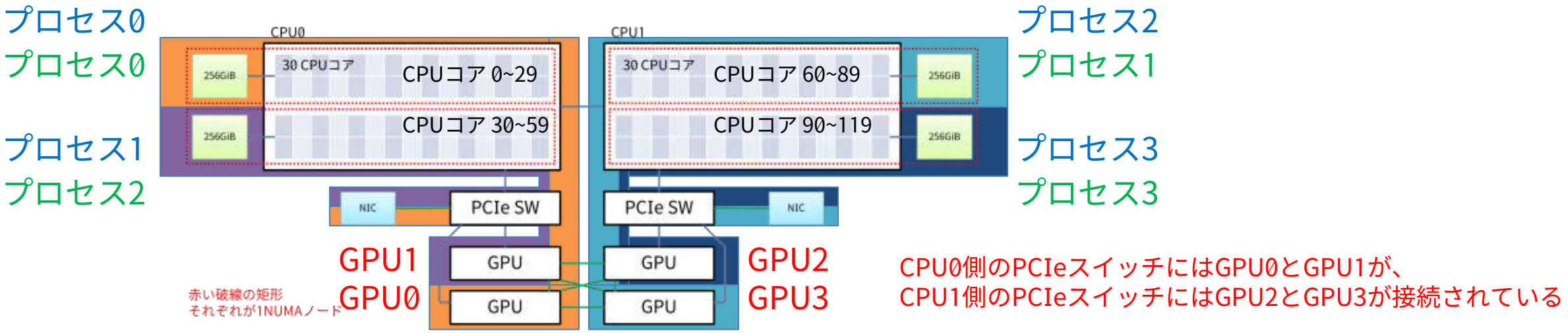
```
Data for node: b0003	State: 3	Flags: 11
	Daemon: [[38146,0],0]	Daemon launched: True
	Num slots: 120	Slots in use: 4	Oversubscribed: FALSE
	Num slots allocated: 120	Max slots: 0
	Num procs: 4	Next node_rank: 4
	Data for proc: [[38146,1],0]
		Pid: 0	Local rank: 0	Node rank: 0	App rank: 0
		State: INITIALIZED	App_context: 0
		Locale:  [B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././././././././././././././././.][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
		Binding: [B/././././././././././././././././././././././././././././././././././././././././././././././././././././././.][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
	Data for proc: [[38146,1],1]
		Pid: 0	Local rank: 1	Node rank: 1	App rank: 1
		State: INITIALIZED	App_context: 0
		Locale:  [././././././././././././././././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
		Binding: [././././././././././././././././././././././././././././././B/./././././././././././././././././././././././././././.][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
	Data for proc: [[38146,1],2]
		Pid: 0	Local rank: 2	Node rank: 2	App rank: 2
		State: INITIALIZED	App_context: 0
		Locale:  [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././././././././././././././././.]
		Binding: [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][B/././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
	Data for proc: [[38146,1],3]
		Pid: 0	Local rank: 3	Node rank: 3	App rank: 3
		State: INITIALIZED	App_context: 0
		Locale:  [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][././././././././././././././././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
		Binding: [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][././././././././././././././././././././././././././././././B/./././././././././././././././././././././././././././.]
```

プロセス0（ランク0）＝コア0
プロセス2（ランク2）＝コア30
プロセス1（ランク1）＝コア60
プロセス3（ランク3）＝コア90

```
Data for node: b0002	State: 3	Flags: 11
	Daemon: [[43984,0],0]	Daemon launched: True
	Num slots: 120	Slots in use: 4	Oversubscribed: FALSE
	Num slots allocated: 120	Max slots: 0
	Num procs: 4	Next node_rank: 4
	Data for proc: [[43984,1],0]
		Pid: 0	Local rank: 0	Node rank: 0	App rank: 0
		State: INITIALIZED	App_context: 0
		Locale:  [B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././././././././././././././././.][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
		Binding: [B/././././././././././././././././././././././././././././././././././././././././././././././././././././././.][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
	Data for proc: [[43984,1],2]
		Pid: 0	Local rank: 2	Node rank: 2	App rank: 2
		State: INITIALIZED	App_context: 0
		Locale:  [././././././././././././././././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
		Binding: [././././././././././././././././././././././././././././././B/./././././././././././././././././././././././././././.][./././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
	Data for proc: [[43984,1],1]
		Pid: 0	Local rank: 1	Node rank: 1	App rank: 1
		State: INITIALIZED	App_context: 0
		Locale:  [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././././././././././././././././.]
		Binding: [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][B/././././././././././././././././././././././././././././././././././././././././././././././././././././././.]
	Data for proc: [[43984,1],3]
		Pid: 0	Local rank: 3	Node rank: 3	App rank: 3
		State: INITIALIZED	App_context: 0
		Locale:  [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][././././././././././././././././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
		Binding: [./././././././././././././././././././././././././././././././././././././././././././././././././././././././.][././././././././././././././././././././././././././././././B/./././././././././././././././././././././././././././.]
```

GPUの割り当て方法

- 各MPIプロセスに対して近いGPUを割り当てたい
  - PCIeスイッチを経由して近くにつながっているGPUを使うようにしたい
- 各プロセスが利用するGPUは環境変数CUDA_VISIBLE_DEVICESで指定する
  - GPUの通し番号（0から3）を指定する、もしくはnvidia-smi -Lコマンドで表示されるUUID（ユニークなID）を指定する
  - 青文字のプロセス番号を使うのであればGPU番号にそのまま使える
  - 緑文字のプロセス番号を使う場合は番号の再計算などが必要



ジョブスクリプト内でプロセス番号（MPIランク番号）を使う方法

- （もちろんMPIプログラム内ではMPI_Comm_rank関数でランク番号を取得できる）
- ジョブスクリプト内で使いたい場合は2段階でスクリプトを実行する
  - mpiexecでスクリプトを呼び出すと、Open MPIにより追加された環境変数をスクリプト内で使用可能
    - OMPI_ で始まる変数などが追加される
  - 環境変数や実行時引数などに利用できる
  - numactlでより詳細なコア割り当てをしたい場合や、ノードグループB,CでGPU番号を指定する際に有用
    - MPIレベルでノード単位の割り当てだけ済ませておき、あとはnumactlで配置する、ということも可能

ジョブスクリプトの書き方例（GPU番号まで考慮）　1/2

青文字の例（先にCPU0を埋める）

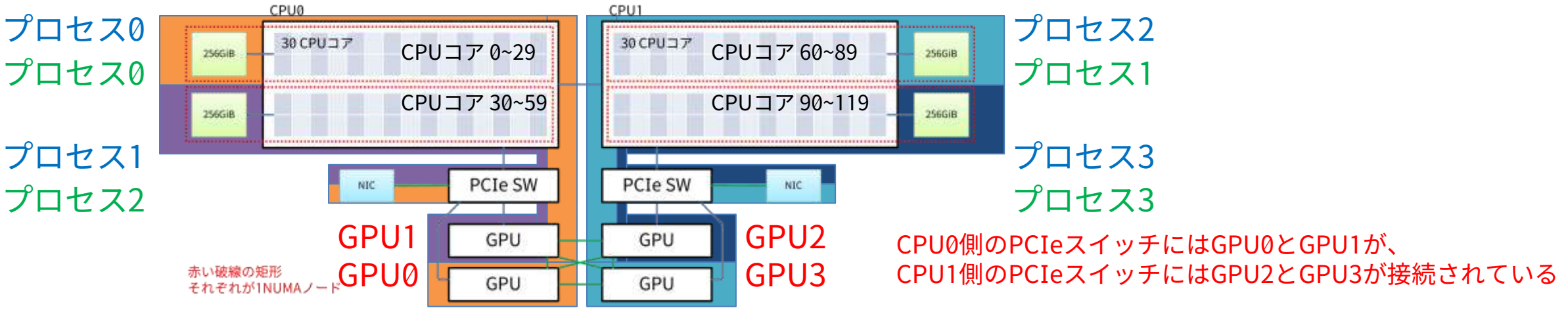
```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=1
mpirun -display-map -display-devel-map -n 4 --map-by numa ¥
--bind-to core --rank-by numa ./runB.sh ./a.out
```

※ --rank-by オプションを書かなかった場合もこちらの挙動

runB.sh

```
#!/bin/bash
export
CUDA_VISIBLE_DEVICES=${OMPI_COMM_WORLD_LOCAL_RANK}
numactl -l $@
```

- 利用するGPU番号としてMPIランクを利用
- LOCALなランクなので複数ノードでも利用可能（後述）
- $@にはスクリプトの引数全体が入っているためこの例で最終的に実行されるコマンドは
  numactl -l ./a.out



CPU0側のPCIeスイッチにはGPU0とGPU1が、
CPU1側のPCIeスイッチにはGPU2とGPU3が接続されている

ジョブスクリプトの書き方例（GPU番号まで考慮）　2/2

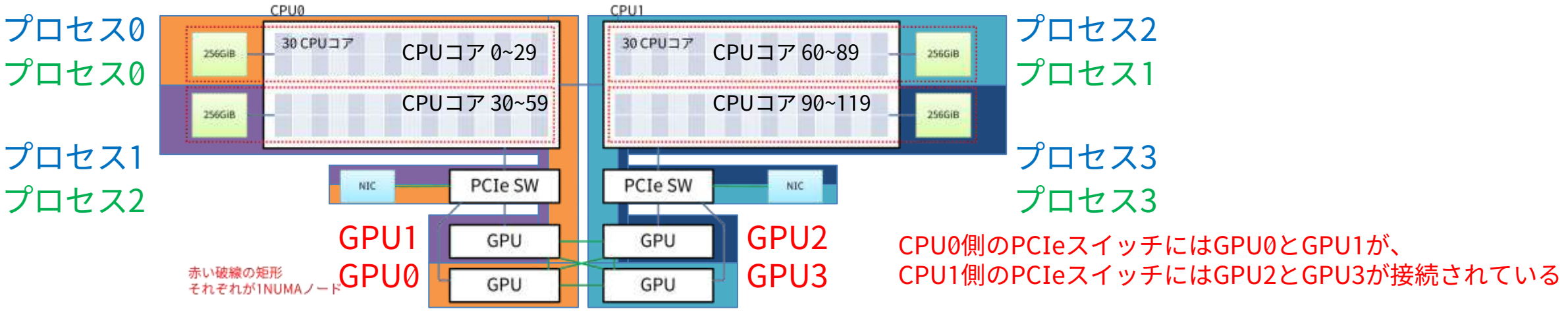
緑文字の例（CPUに交互に配置する）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=1
mpirun -display-map -display-devel-map -n 4 --map-by numa ¥
--bind-to core --rank-by socket ./runG.sh ./a.out
```

runG.sh

```
#!/bin/bash
case ${OMPI_COMM_WORLD_LOCAL_RANK} in
    0)
        export CUDA_VISIBLE_DEVICES=0
        numactl -l $@;;
    1)
        export CUDA_VISIBLE_DEVICES=2
        numactl -l $@;;
    2)
        export CUDA_VISIBLE_DEVICES=1
        numactl -l $@;;
    3)
        export CUDA_VISIBLE_DEVICES=3
        numactl -l $@;;
esac
```

- MPIランクで単純に分岐
- あとは前ページと同様



CPU0側のPCIeスイッチにはGPU0とGPU1が、
CPU1側のPCIeスイッチにはGPU2とGPU3が接続されている

CPUによるOpenMP並列化とGPU利用を考慮した割り当て

- プロセスを配置する際にCPUによるOpenMP並列実行も考慮するにはどうすればよいか
- 1NUMAあたり1MPIプロセスで考えてきたため、大きな変更は必要ない
- 変更点
  - 環境変数OMP_NUM_THREADSでスレッド並列数を指定する
  - --bind-to coreを--bind-to numaに変更する
    - プロセスをCPUコア単位ではなくNUMAノード単位で割り当てておき、NUMAノード内でスレッド並列実行する

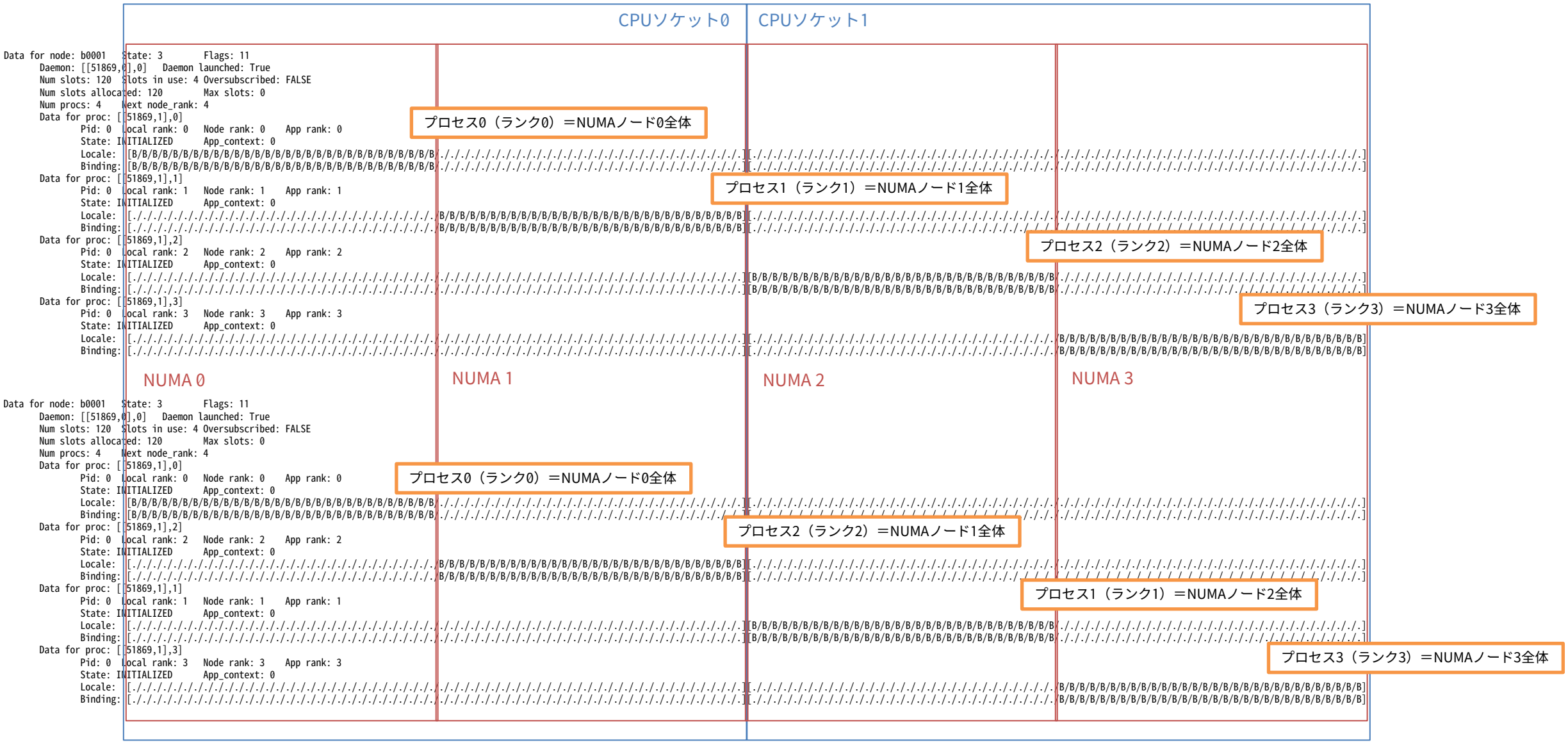
青文字の例（先にCPU0を埋める）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=任意スレッド数
mpirun -display-map -display-devel-map -n 4 --map-by numa ¥
--bind-to numa --rank-by numa ./runB.sh ./a.out
```

緑文字の例（CPUに交互に配置する）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=1
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=任意スレッド数
mpirun -display-map -display-devel-map -n 4 --map-by numa ¥
--bind-to numa --rank-by socket ./runG.sh ./a.out
```

-display-devel-map 指定時に得られる出力の例



複数ノード実行

- #PJM -L node=ノード数 により利用ノード数を指定（必須）
- これまでの例を単純に複数ノード化すると、はじめにノード0へ4プロセス配置し、次はノード1へ4プロセス配置……という振る舞いになる
  - --map-by numa で割り当てているため、ノードあたり4プロセス配置になる

青文字の例（先にCPU0を埋める）

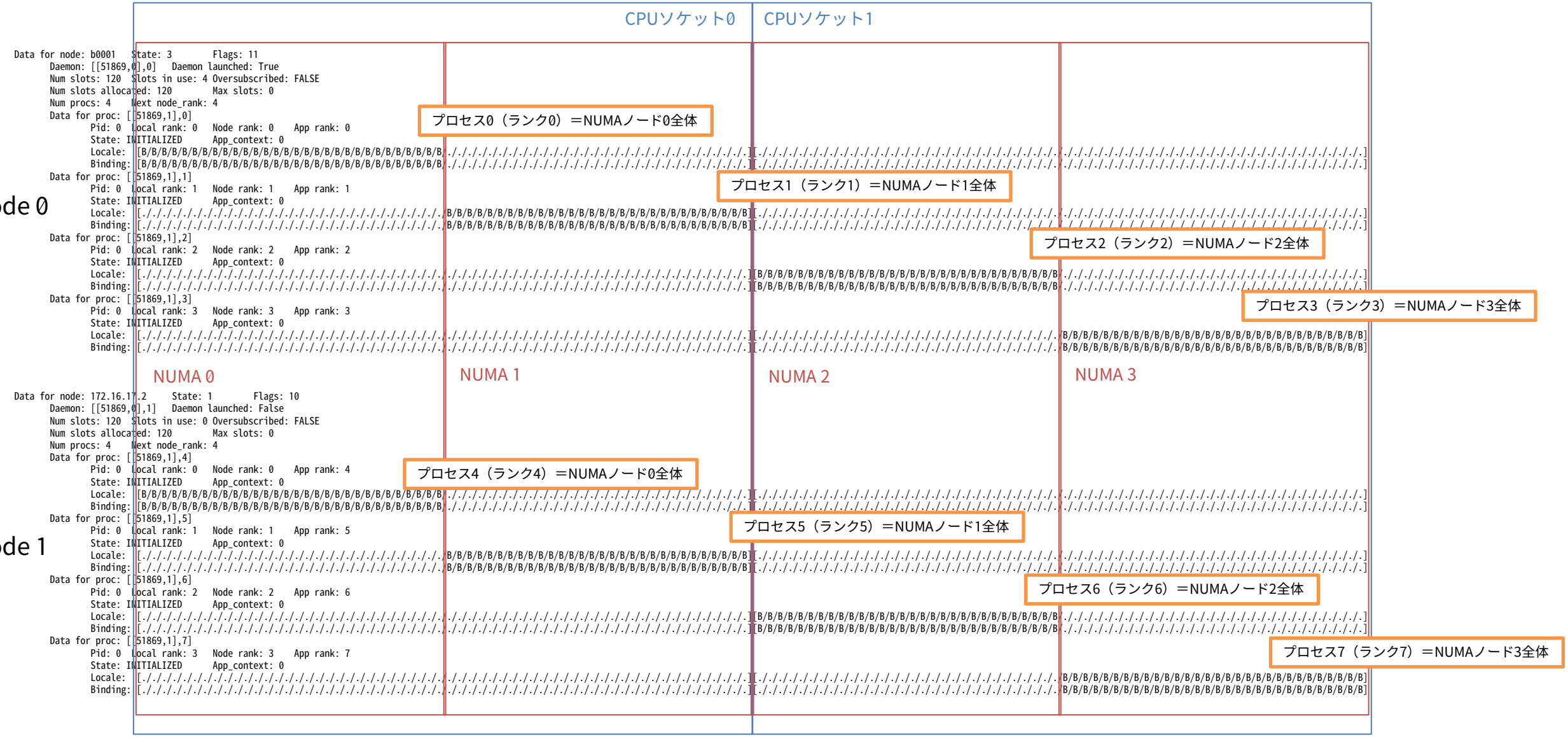
```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=2
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=任意スレッド数
mpirun -display-map -display-devel-map -n 8 --map-by numa ¥
--bind-to numa --rank-by numa ./runB.sh ./a.out
```

緑文字の例（CPUに交互に配置する）

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=2
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=任意スレッド数
mpirun -display-map -display-devel-map -n 8 --map-by numa ¥
--bind-to numa --rank-by socket ./runG.sh ./a.out
```

  - この例の場合は runB.sh runG.sh 側の修正は不要
  - -machinefile オプションによる利用ノードの指定は不要（module load時に環境変数が設定されているため）

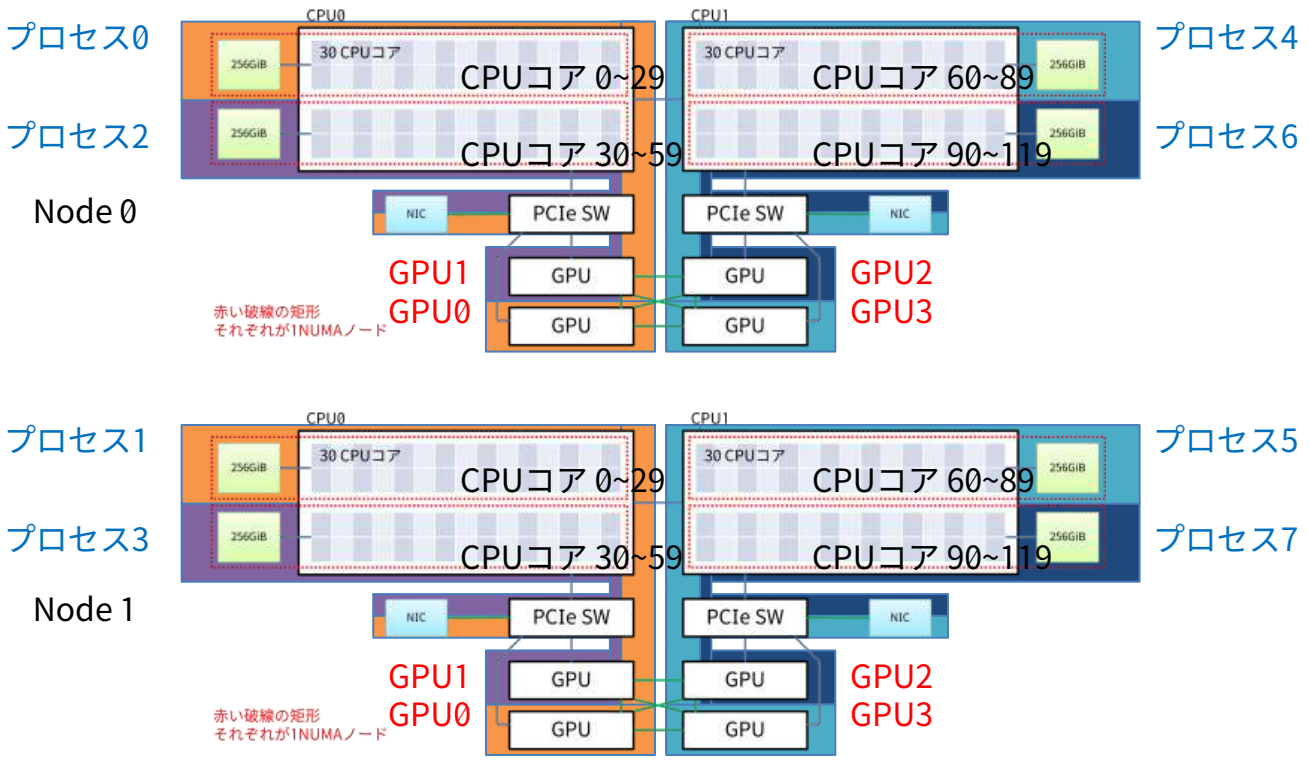
-display-devel-map 指定時に得られる出力の例



複数ノード実行

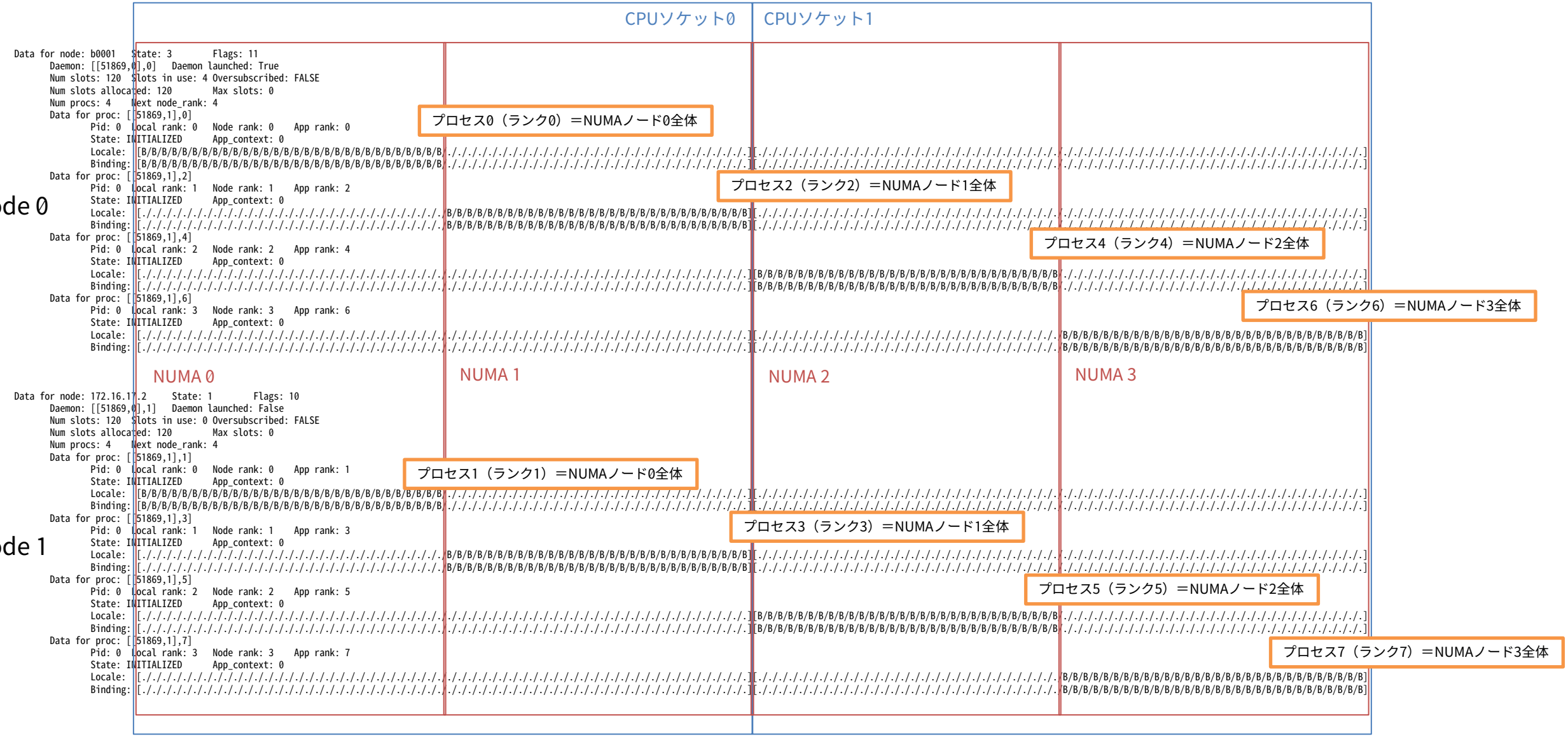
- --rank-by node を指定すると、連続するランクのプロセスが別のノードに配置される

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L node=2
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=任意スレッド数
mpirun -display-map -display-devel-map -n 8 --map-by numa ￥
--bind-to numa --rank-by node ./runB.sh ./a.out
```



– ノード内のプロセス割り当て順序を --rank-by socket と同じにしたい場合は、mpirunによるプロセスの配置はノード単位だけ考えておき、呼び出されるスクリプト側でnumactlによる配置調整を行うなどの対応が必要

-display-devel-map 指定時に得られる出力の例



ノード共有実行の場合は？

- gpu<4 を指定して実行した、ノードの一部の資源のみが使えるジョブの場合

- 計算コアがどのように提供されてくるかわからないが、コアをどのように配置するかをまったく指定しないで実行するとエラーしてしまう
  - 例

```
#!/bin/bash
#PJM -L rscgrp=b-batch
#PJM -L gpu=2
#PJM -L elapse=10:00
module load nvidia nvhpcx cuda nccl
export OMP_NUM_THREADS=4
mpirun -display-map -display-devel-map -n 2 ./a.out
```

```
Your job has requested more processes than the ppr for
this topology can support:

  App: ./a.out
  Number of procs:  2
  PPR: 1:node

Please revise the conflict and try again.
```

- --map-by ppr:プロセス数:node:PE=プロセスあたりスレッド数 --bind-to core が妥当
  - 例

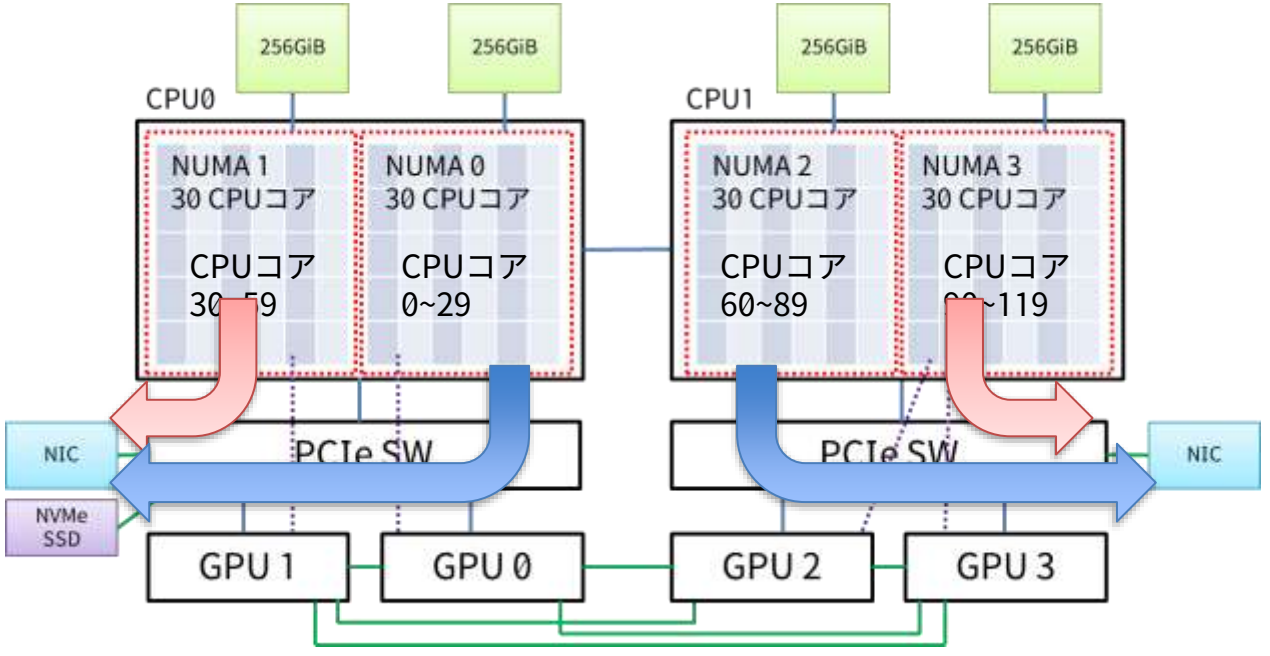
```
#PJM -L gpu=2
export OMP_NUM_THREADS=4
mpirun -display-map -display-devel-map -n 2 --map-by ppr:2:node:PE=4 --bind-to core ./a.out
```

PCI-Express接続まで考えた最適化

性能に及ぼす影響の確認

- NICに近いNUMAノードにプロセスを配置するかいなかでMPI通信に性能差が生じる

赤矢印：近い
青矢印：遠い（赤より時間がかかる）



その他・補足

参考：プログラム側から実行されているコアを確認する方法

- /proc/プロセスID/task/スレッドID/stat を確認する
  - 38番目のエントリがコア番号
- sched_getaffinity で得られる情報を確認する

/proc/プロセスID/task/スレッドID/stat を確認する例

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <omp.h>
#define min(a,b) a<b?a:b
void check_core(int rank){
  char buf[0xff], buf2[4], hostname[0xff];
  FILE *fp;
  int pid, tid, ompid, count, c1, c2;
  pid = getpid();
  tid = (pid_t) syscall(SYS_gettid);
  ompid = omp_get_thread_num();
  sprintf(buf, "/proc/%d/task/%d/stat", pid, tid);
  if ((fp = fopen(buf, "r")) != NULL) {
    fgets(buf, 0xff, fp);
    fclose(fp);
    count = 0;
    c1 = c2 = 0;
    while(buf[c1]!='￥0'){
      if(buf[c1]==' ')count++;
      c1++;
      if(count==38)break;
    }
    c2 = c1;
    while(buf[c2]!='￥0'){
      if(buf[c2]==' ')break;
      c2++;
    }
    strncpy(buf2, &buf[c1], min(c2-c1, 4));
    buf2[min(c2-c1, 4)] = '￥0';
    gethostname(hostname, 0xff);
    printf("hostname=%s rank=%d thread-id=%2d omp-tid=%2d core-id=%s %d￥n",
           hostname, rank, tid, ompid, buf2, atoi(buf2)/15);
  }
```

右上へ続く

```c
  FILE *fout;
  char fname[0xff];
  sprintf(buf, "/proc/%d/task/%d/stat", pid, tid);
  snprintf(fname, 0xff, "stat_%d_%d.txt", pid, tid);
  if ((fout = fopen(fname, "w")) != NULL) {
    if ((fp = fopen(buf, "r")) != NULL) {
      while(fgets(buf, 0xff, fp)!=NULL){
        fprintf(fout, "%s", buf);
      }
      fclose(fp);
    }
    fclose(fout);
  }
}

int main()
{
  #pragma omp parallel
  check_core(0);
}
```

sched_getaffinity で得られる情報を確認する例

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <omp.h>
#include <sched.h>

void check_core(int rank){
  int pid, tid, ompid;
  pid = getpid();
  tid = (pid_t) syscall(SYS_gettid);
  ompid = omp_get_thread_num();

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
#ifdef _OPENMP
  int ret = sched_getaffinity(tid, sizeof(cpuset), &cpuset);
#else
  int ret = sched_getaffinity(pid, sizeof(cpuset), &cpuset);
#endif
  char affinity[0xfff];
  affinity[0] = '￥0';
  int ncore = sysconf(_SC_NPROCESSORS_CONF);
  for (int i = 0; i < ncore; i++) {
    if (CPU_ISSET(i, &cpuset) == 1) {
      snprintf(affinity, 0xfff, "%s %d", affinity, i);
    }
  }
  printf("getaffinity rank=%d omp-tid=%2d core-id=%s￥n", rank, ompid, affinity);
}
```

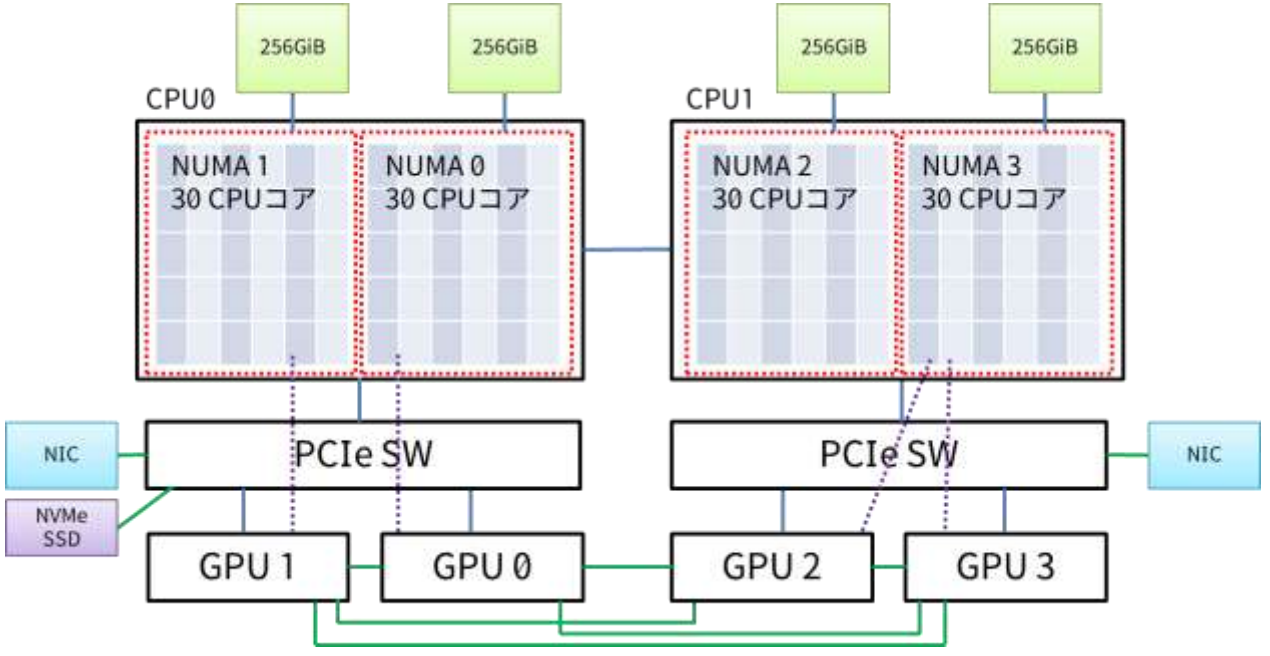
```c
int main()
{
  #pragma omp parallel
  check_core(0);
}
```

右上へ続く

NUMA構成の偏り（？）について

- PCI-Expressの構成上、GPU2はNUMA2ではなくNUMA3に接続されている
- 各NUMAとGPUの間でデータ転送を行うと、NUMA 2とGPU2の間だけ他より少し遅い可能性がある

- 実際に測定して確認した
  - 単純にcudaMemcopyを繰り返し実行
  - Device to Host, Host to Device,
    Pinned Host to Device,
    Device to Pinned Host の性能を比較
    - Device : cudaMallocで確保
    - Host : mallocで確保
    - Pinned Host : cudaMallocHostで確保



実験方法（中身）

- ソースコード抜粋
  - warm upしてから測定
  - array_bytes=1000M, iter=10

```
printf("Host to Device¥n");
for(int i=0; i<iter; i++){
  cudaMemcpy(d_d, h_d, array_bytes, cudaMemcpyHostToDevice);
}
t1 = omp_get_wtime();
for(int i=0; i<iter; i++){
  cudaMemcpy(d_d, h_d, array_bytes, cudaMemcpyHostToDevice);
}
t2 = omp_get_wtime();
t = t2 - t1;
printf("%lf sec %lf GB/s¥n", t, (double)array_bytes*(double)iter/t/1000.0/1000.0/1000.0);
```

- コンパイル方法

```
nvcc -O3 -Xcompiler -fopenmp -c bench.cu          module load cuda/12.6.1 を利用
nvcc -O3 -Xcompiler -fopenmp -o bench bench.o
```

実験方法（ジョブスクリプト）

- 1ノードを確保
- NUMAノードとGPUの総当たりで測定
- PCI Express Gen5 x16の理論性能は
  片方向あたり63.02GB/s

```
#!/bin/bash
#PJM -L rscgrp=b-batch-low
#PJM -L node=1
#PJM -L elapse=5:00
#PJM -j
#PJM -S

module load cuda/12.6.1

numactl -H
numactl -s

for i in 0 1 2 3
do
        for j in 0 1 2 3
        do
                export CUDA_VISIBLE_DEVICES=$j
                numactl -N $i ./bench 1000000000 10
        done
done
```

測定結果

- Device to Pinned Hostの転送性能が安定して同じ傾向
  - NUMA 0,1：赤=55、青=50、その他=45
  - NUMA 2：赤=50、青=50、その他=45
  - NUMA 3：赤=55、青=55、その他=45
  - 転送性能に最大10%程度の差が生じている
  - （Device to Pinned Host以外ではあまり影響がないのは何故だろう？）

※55/63.02=87%なので理論性能比的には十分まともな性能が出ていると思われる

- 赤文字：1GPUジョブで割り当たる想定の組み合わせ
- 青文字：同じCPUソケット内の転送

NODE ID 0x02FF0003

		Host to Device	Device to Host	Pinned Host to Device	Device to Pinned Host
numa 0	G 0	18.85785	15.48991	55.44735	55.45828
	G 1	18.68886	14.16233	53.22433	50.89731
	G 2	18.78489	13.77854	55.37305	44.68847
	G 3	18.86184	13.85467	55.40504	44.66958
numa 1	G 0	18.88373	14.38641	53.52818	50.46938
	G 1	18.97024	15.45923	55.4556	55.46419
	G 2	18.68695	13.12183	55.37095	44.60962
	G 3	18.48288	12.981	55.40664	44.63325
numa 2	G 0	18.58385	10.97836	55.39991	44.57844
	G 1	18.96006	11.56483	55.42331	44.59313
	G 2	19.15337	15.5199	53.76401	50.44027
	G 3	18.81034	15.28588	53.34937	49.32371
numa 3	G 0	15.16296	13.34552	55.38755	44.45958
	G 1	15.13728	12.90575	55.42084	44.69609
	G 2	15.29499	16.32306	55.40802	55.46356
	G 3	15.28514	16.41893	55.47968	55.46118

NODE ID 0x02FF0004

		Host to Device	Device to Host	Pinned Host to Device	Device to Pinned Host
numa 0	G 0	18.64805	13.97974	55.43118	55.39325
	G 1	18.92391	15.84694	53.06435	50.91793
	G 2	18.75765	12.17527	55.3955	44.78512
	G 3	18.44147	12.26705	55.39527	44.82183
numa 1	G 0	18.88768	15.4023	53.6259	50.47075
	G 1	18.99958	15.44231	55.17212	55.46414
	G 2	18.68909	12.03862	55.39738	44.77385
	G 3	18.46153	11.49279	55.40014	44.76233
numa 2	G 0	18.15502	11.20013	55.382	44.53951
	G 1	18.63463	11.59474	55.38045	44.6401
	G 2	14.62844	15.37351	53.85164	50.41127
	G 3	19.0305	15.91484	53.42734	49.2544
numa 3	G 0	19.36996	12.14226	55.38405	44.603
	G 1	19.284	12.5091	55.37677	44.50389
	G 2	18.88895	14.32608	55.43174	55.4623
	G 3	19.39469	15.73937	55.47552	55.46171

NODE ID 0x02FF0006

		Host to Device	Device to Host	Pinned Host to Device	Device to Pinned Host
numa 0	G 0	18.72736	15.39645	55.4668	55.35977
	G 1	18.5093	15.2545	53.42744	50.65789
	G 2	18.65588	12.29475	55.38887	44.84119
	G 3	18.29056	11.75759	55.36784	44.82964
numa 1	G 0	19.03542	15.37754	52.8724	50.42982
	G 1	19.08912	14.24617	55.43197	55.37363
	G 2	18.84233	12.41847	55.38739	44.91415
	G 3	18.63283	12.29901	55.37003	44.80122
numa 2	G 0	18.32881	10.70053	55.41504	44.52663
	G 1	14.64927	12.1871	55.39664	44.61277
	G 2	14.80216	15.3208	53.43778	51.05015
	G 3	18.69794	15.31816	53.34714	49.70487
numa 3	G 0	19.24442	12.86058	55.40103	44.59503
	G 1	19.19986	12.92374	55.38274	44.65259
	G 2	19.15909	15.8811	55.42886	55.22288
	G 3	19.34607	15.83183	55.44511	55.25808

NUMA構成の偏り（？）について：結論

- GPU2がNUMA2ではなくNUMA3につながっていることは性能にある程度影響している
  - Pinned Memoryを使ったcudaMemcpyに影響
- MPI通信性能についてもいくつかテストしてみたが、いまのところNUMAの偏り自体の影響はなさそう
  - 利用するCPUやGPUがNICと同じPCI-Expressスイッチにつながっているかどうかが重要