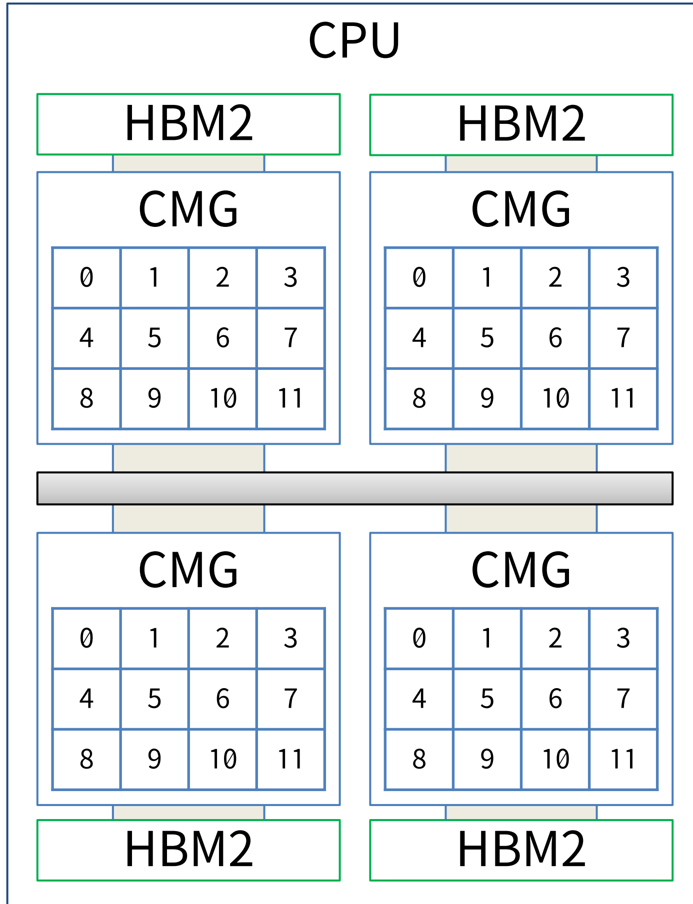
2020.08.26　初版公開
2020.09.07　補足2を追加、その他細かい修正
2021.03.26　ファーストタッチの例を追加、その他細かい修正

Type Iサブシステム向けのプロセス・スレッド配置方法

Type Iサブシステム向けのプロセス・スレッド配置方法

- Type Iサブシステムは1CPUが4つのCMGに別れているため、4以上のプロセスを用いた実行を基本とする
  - OpenMPのみを用いた並列化や、ノードあたり4プロセス未満のOpenMP/MPIハイブリッド並列化も可能であるが、推奨しない
    - CMGを跨いだメモリアクセスが発生するため、十分な性能を発揮しにくい
  - フラットMPI並列化を行う場合は、以下のようなプロセス配置が推奨される
    - 各CMGにnプロセスずつ、合計4nプロセス配置
  - OpenMP/MPIハイブリッド並列化を行う場合は、以下のようなプロセス配置が推奨される
    - 各CMGにnプロセスずつ、合計4nプロセス配置、さらにCMG内でOpenMP並列化
- 補足
  - メモリが4つのCMGに分散して配置されていることから、プロセスやスレッドも4つのCMGに均等に配置され、さらに各CMGに直結したメモリを使わないと良い性能が得られないということ
  - 4ソケットCPUだと思って使うのが良い
  - より細かい情報を得たい場合はプログラミングガイドプロセッサ編やMPI使用手引書を確認してください（HPCポータルから入手可能）

A64FXのコア構成の確認

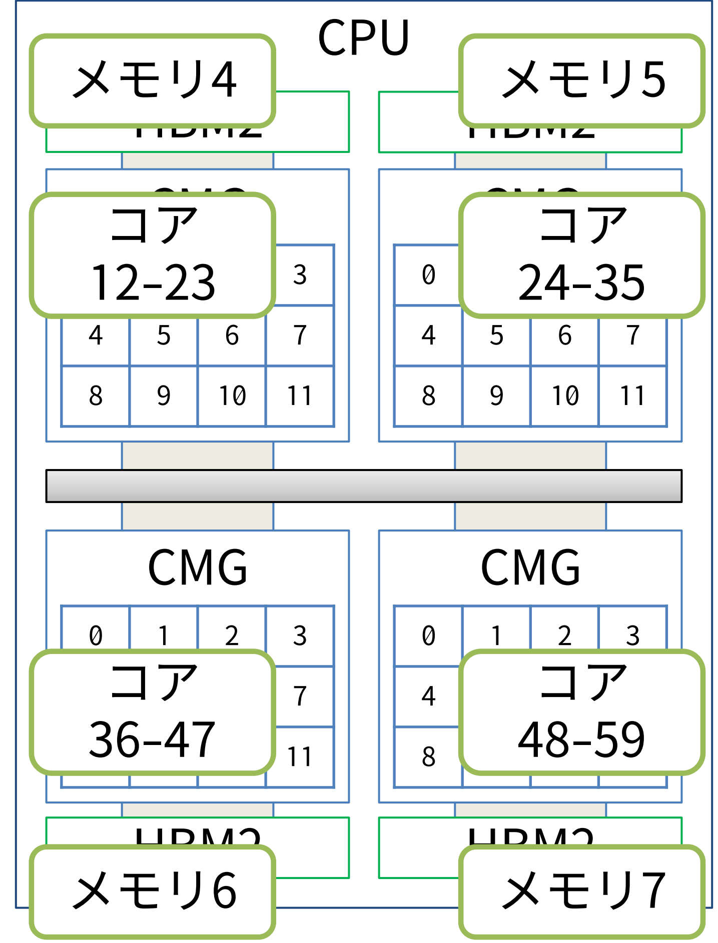


- 1CPUは4つのCMGから構成されている
- 各CMGは12の計算コアから構成されている
- メモリ(HBM2)は各CMGごとに接続されているため、CPU全体でみると計算コアからメモリへのアクセス性能は不均一
  - 自CMGに接続されたHBM2へのアクセスは高速
  - 他CMGに接続されたHBM2へのアクセスは低速
- 1プロセスが複数のCMGにまたがって配置される使い方は非推奨
  - 問題なく動作するが、良い性能が得やすい実行形態ではない
- 推奨される「プロセス数」×「スレッド数」の例
  - 基本：「4プロセス」×「プロセスあたり12スレッド以下」
    - 各CMGに1プロセスを割りあてる
    - 各プロセスは同一CMG内のコアにスレッドを割りあてる
    - 2のべき乗の数が良いなら「4プロセス」×「8スレッド」
  - より多くのプロセスを利用したい場合は各CMGに配置するプロセス数を増やし、その分だけスレッド数を減らす
    - 「8プロセス」×「6スレッド以下」
    - 「12プロセス」×「4スレッド以下」
    - 「16プロセス」×「3スレッド以下」　など

numactlによる詳細なプロセス配置の指定とOpenMP並列実行

- 「プログラミングガイド プロセッサ編」に掲載されているとおり、numactlでコアの指定（-C）とメモリの指定（-m）が可能

CMG	コア番号	メモリ番号
CMG0	12-23	4
CMG1	24-35	5
CMG2	36-47	6
CMG3	48-59	7



- MPI+OpenMPハイブリッド実行では
  `mpiexec numactl (options) ./a.out`
  - numactlで調整できるのはmpiexecで割り当てられたコアの範囲だけ
    - numactl -sを実行するだけのスクリプトをmpiexecで呼び出すと簡単に割り当て状況を確認できる
  - 一般的な割り当てはmpiexecだけで十分に対応が可能
- 1つのCMGだけを使ったOpenMP実行の例
  - CMG0に対応するコア番号とメモリ番号を指定

```
#PJM -L node=1
export OMP_NUM_THREADS=12
numactl -C 12-23 -m 4 ./a.out
```

参考：前ページのスクリプト例の完全な例

```
#!/bin/bash -x
#PJM -L rscgrp=fx-extra
#PJM -L node=1
#PJM --mpi proc=1
#PJM -j
#PJM -S

export OMP_NUM_THREADS=12
numactl -C 12-23 -m 4 ./a.out
```

1/4CPU（=1CMG）のみを用いて
OpenMPによる並列処理が行われる

1ノード、フラットMPI実行におけるプロセスの配置

- 基本：ジョブスクリプトの最初の#PJMの部分で指定したノード数とプロセス数に基づき、全CMGにまたがってプロセスが均等に配置される
- 8以上のプロセス数を指定すると1CMGに複数のプロセスが配置される
  - 引数として
    `-mca plm_ple_numanode_assign_policy share_cyclic`
    または
    `-mca plm_ple_numanode_assign_policy share_band`
    を指定することで配置順序を調整できる
    - share_cyclic：各CMGに1つずつプロセスを配置、を繰り返す（いわゆるscatterな配置）
    - share_band：全プロセス数÷CMG数（4）のプロセスをCMGに配置したら次のCMGへ（いわゆるbalancedな配置）
    - share_cyclicがデフォルト
  - 配置例はMPI+OpenMPハイブリッド実行とあわせて紹介する

1ノード、MPI+OpenMPハイブリッド実行におけるプロセスの配置

- 基本の実行形態である4プロセス×12以下スレッド実行であればプロセス数とスレッド数を普通に指定するだけで良い
  - 以下、具体的な指定の例（割り当てイメージは次頁）

```
#PJM -L node=1
#PJM --mpi proc=4
export OMP_NUM_THREADS=12
mpiexec ./a.out
```

```
#PJM -L node=1
#PJM --mpi proc=4
export OMP_NUM_THREADS=8
mpiexec ./a.out
```

```
#PJM -L node=1
#PJM --mpi proc=4
export OMP_NUM_THREADS=4
mpiexec ./a.out
```

フルバージョン例

```
#!/bin/bash -x
#PJM -L rscgrp=fx-extra
#PJM -L node=1
#PJM --mpi proc=4
#PJM -j
#PJM -S

export OMP_NUM_THREADS=12
mpiexec ./a.out
```

numactlを実行するスクリプトをmpiexecから実行すれば、mpiexecによって割り当てられた資源の情報を確認できる

```
#!/bin/bash -x
#PJM -L rscgrp=fx-extra
#PJM -L node=1
#PJM --mpi proc=4
#PJM -j
#PJM -S

export OMP_NUM_THREADS=12
mpiexec ./run2.sh
```

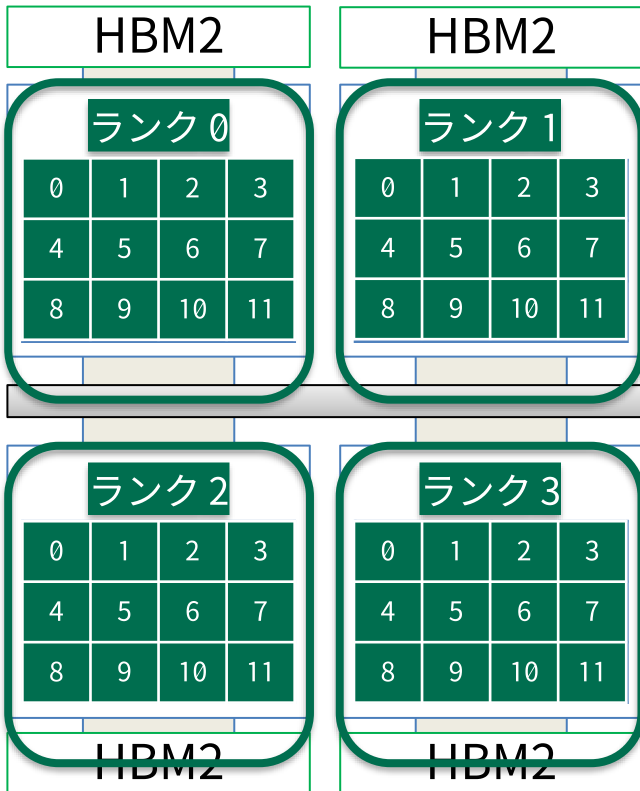
↓run2.sh

```
#!/bin/bash -x
numactl -s
./a.out
```

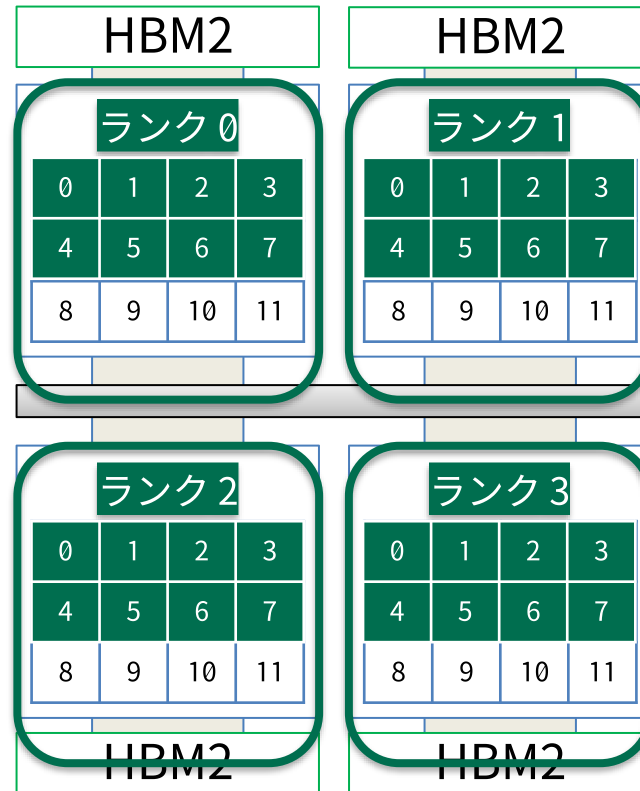
※右例のようにmpiexecでスクリプトを実行したい場合はchmod u+xなどで実行権を与えておく必要がある

1ノード、MPI+OpenMPハイブリッド実行の例

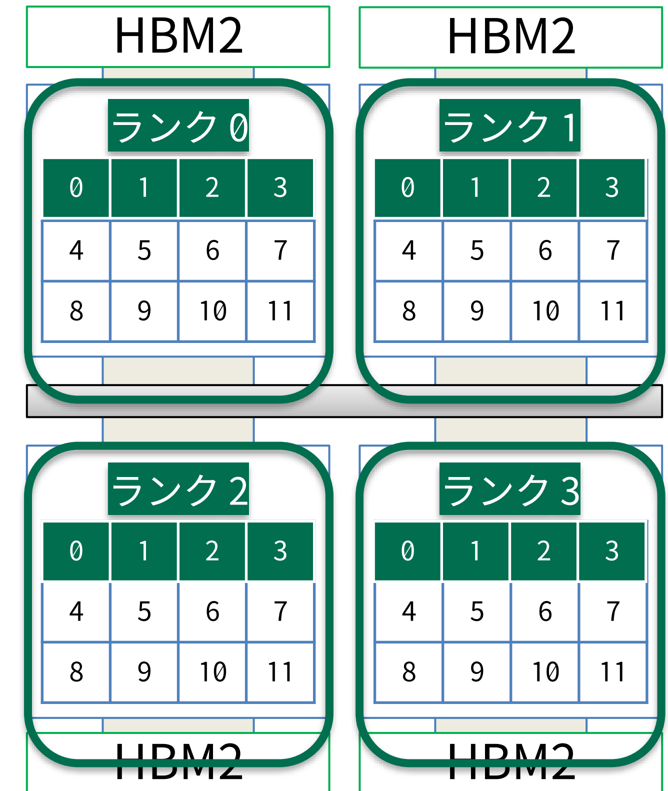
```
#PJM -L node=1
#PJM --mpi proc=4
export OMP_NUM_THREADS=12
mpiexec ./a.out
```




```
#PJM -L node=1
#PJM --mpi proc=4
export OMP_NUM_THREADS=8
mpiexec ./a.out
```




```
#PJM -L node=1
#PJM --mpi proc=4
export OMP_NUM_THREADS=4
mpiexec ./a.out
```



※どのCMGから埋めるかが厳密に決まっているわけではないため、実際に試すと実行のたびに変わることがある

1ノード、MPI+OpenMPハイブリッド実行、プロセス数が多い場合の例

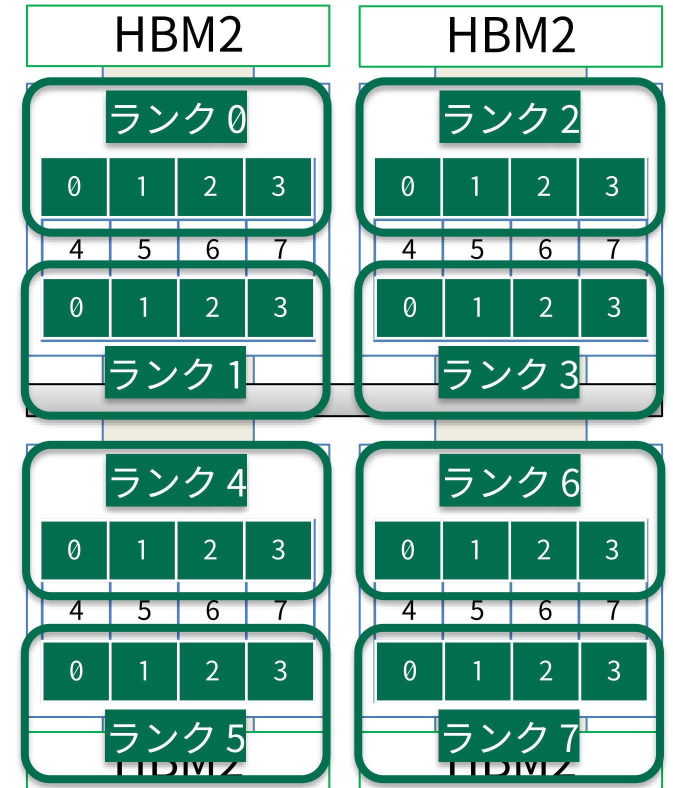
CMG数よりも多くのプロセスを配置するとどうなるだろうか？
例：8プロセス×4スレッドを指定

```
#PJM -L node=1
#PJM --mpi proc=8
export OMP_NUM_THREADS=4
mpiexec ./a.out  もしくは
mpiexec -mca plm_ple_numanode_assign_policy share_cyclic ./a.out
```

← 全CMGにプロセスを配置してから
各CMGに2つ目のプロセスを配置し始める

CMGに2つ目のプロセスを配置できる場合は
積極的に配置する →

```
#PJM -L node=1
#PJM --mpi proc=8
export OMP_NUM_THREADS=4
mpiexec -mca plm_ple_numanode_assign_policy share_band ./a.out
```



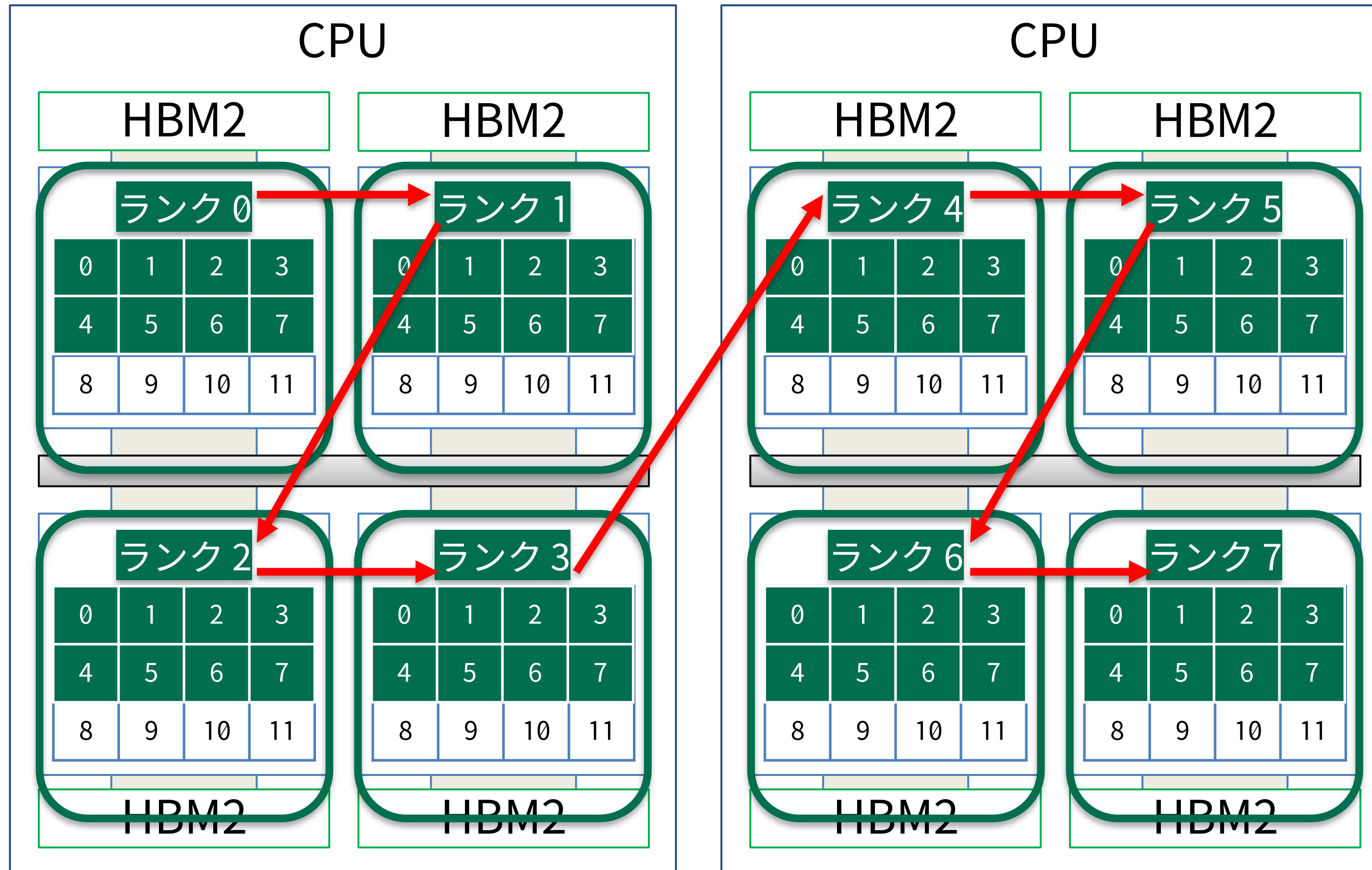
- あくまで割り当てポリシの指定であり、物理コアに配置される順序が常に同じになるとは限らない模様。
- メモリについては、デフォルトでコアの存在するCMG上のメモリが使われる。（容量の都合などにより）どうしてもCMG外のメモリまで使いたい場合は、numactlのinterleave指定を使う。例：--interleave=4-7

複数ノード実行時におけるプロセスの配置

- 複数ノードの場合は、ジョブに与えたノード数とプロセス数を元に均等なプロセス配置が行われる
- 連続するIDのプロセスをなるべく同じノードに配置したいかどうか（いわゆるcompactな配置かscatterな配置か）を指定可能
- rank-map-bychip（デフォルト）
  - ノードに十分な数（全プロセス数÷ノード数、切り上げ）のプロセスを配置してから次のノードへ
  - compactな配置
- rank-map-bynode
  - 1プロセス配置したら次のノードへ
  - scatterな配置

- ノード数を指定する際に確保する形状（torus, mesh, noncont）も指定できる
  - 大規模実行する場合やTofu-Dの能力をフル活用したい場合には調整する余地がある

2ノード実行の例：rank-map-bychip

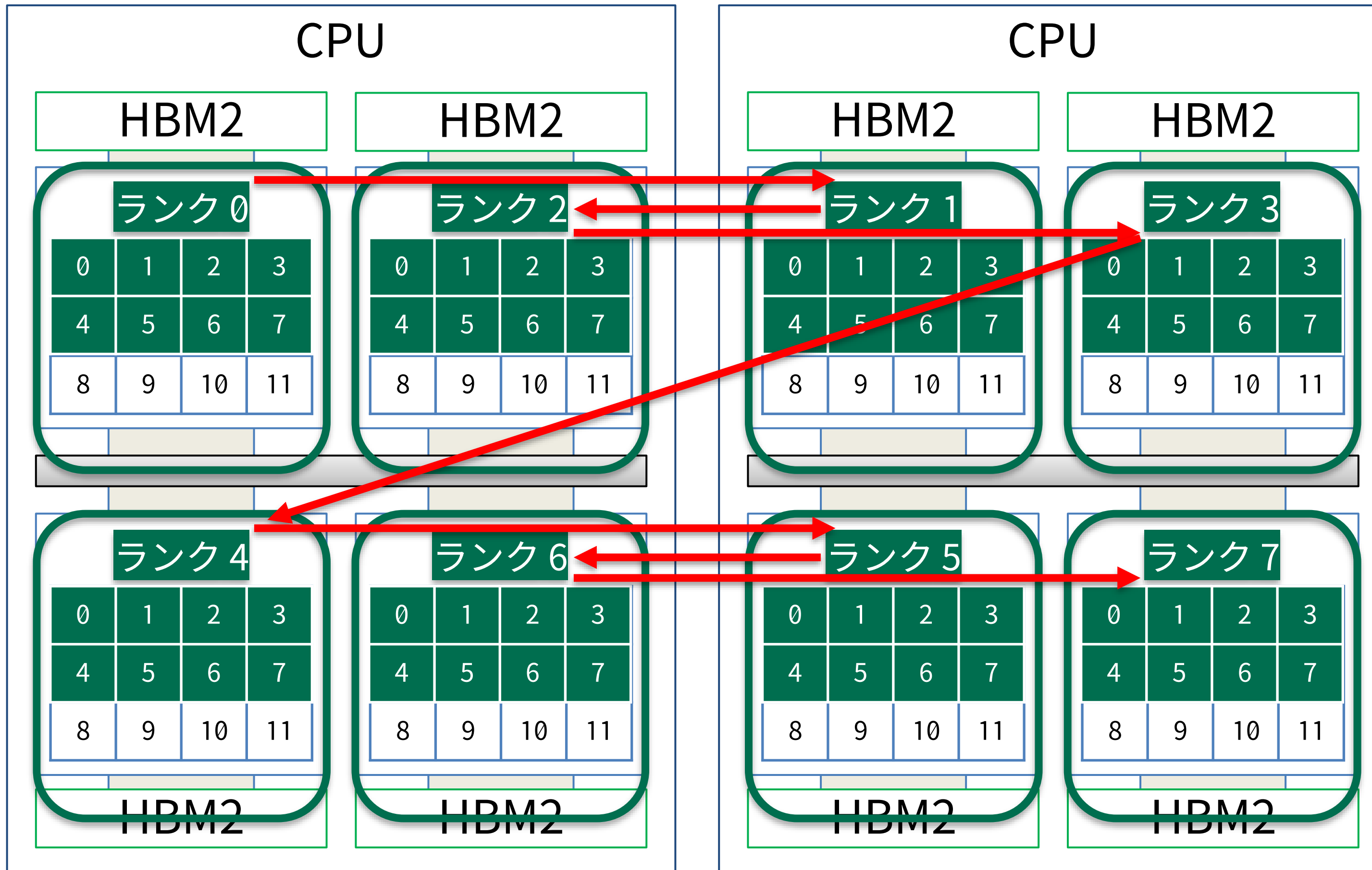



```
#PJM -L node=2
#PJM --mpi proc=8
#PJM --mpi rank-map-bychip
export OMP_NUM_THREADS=8

mpiexec ./a.out
```

- rank-map-bychipでは、プロセス数8÷ノード数2=4プロセスを最初のノードに配置してから次のノードにプロセスを配置する。
- 隣接プロセス間の通信が多い場合はこの配置が適する。

2ノード実行の例：rank-map-bynode




```
#PJM -L node=2
#PJM --mpi proc=8
#PJM --mpi rank-map-bynode
export OMP_NUM_THREADS=8

mpiexec ./a.out
```

- rank-map-bynodeでは、1プロセス配置したら次は別のノードへ、と確保したノード全体にまたがって均等にプロセスを配置。
- ノード間の負荷を均等化することを重視する場合はこちらの配置も試す余地がある。

まとめ

- Type Iサブシステムで実用的なプロセス・スレッド配置については、ノード数指定、スレッド数指定、rank-map-*パラメタの指定で十分カバーできると思われる
- もし「○○な配置はどうすれば良いのか？」と具体的に分からない事例があればQ&Aシステムへご質問ください
  - ある程度需要がありそうな事例であると思われる場合には本資料に追記させていただきます

参考：プロセスの配置情報を確認する方法（プログラム側から）

- /proc/{PID}/task/{TID}/stat の39列目（processor情報）にコア番号が入っている
- 確認用テストプログラムの例　→
- サブシステムを問わず利用可能、以下はType Iにおける4プロセス×4スレッドでの実行例（ソート済み）

```
hostname=fx2271 rank=0 thread-id=63 omp-tid= 0 core-id=12
hostname=fx2271 rank=0 thread-id=77 omp-tid= 1 core-id=13
hostname=fx2271 rank=0 thread-id=80 omp-tid= 2 core-id=14
hostname=fx2271 rank=0 thread-id=84 omp-tid= 3 core-id=15
hostname=fx2271 rank=1 thread-id=65 omp-tid= 0 core-id=24
hostname=fx2271 rank=1 thread-id=75 omp-tid= 1 core-id=25
hostname=fx2271 rank=1 thread-id=79 omp-tid= 2 core-id=26
hostname=fx2271 rank=1 thread-id=83 omp-tid= 3 core-id=27
hostname=fx2271 rank=2 thread-id=64 omp-tid= 0 core-id=36
hostname=fx2271 rank=2 thread-id=78 omp-tid= 1 core-id=37
hostname=fx2271 rank=2 thread-id=82 omp-tid= 2 core-id=38
hostname=fx2271 rank=2 thread-id=86 omp-tid= 3 core-id=39
hostname=fx2271 rank=3 thread-id=66 omp-tid= 0 core-id=48
hostname=fx2271 rank=3 thread-id=76 omp-tid= 1 core-id=49
hostname=fx2271 rank=3 thread-id=81 omp-tid= 2 core-id=50
hostname=fx2271 rank=3 thread-id=85 omp-tid= 3 core-id=51
```

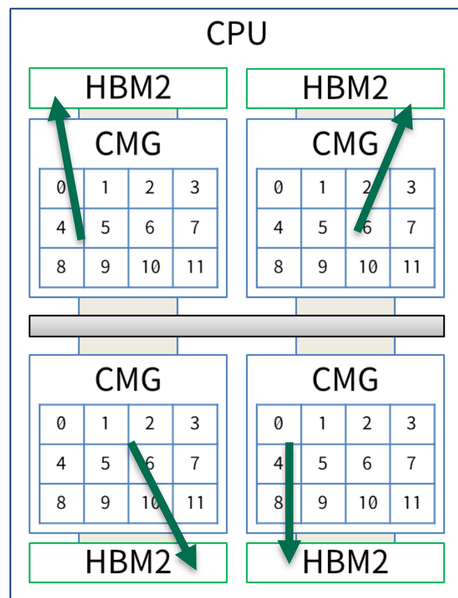
```
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <mpi.h>
#include <omp.h>

#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¥n",
        hostname, rank, tid, ompid, buf2);
  }
}
```

※strtokで分解したらNULLを喰らうことが度々あったため、手動で分割している

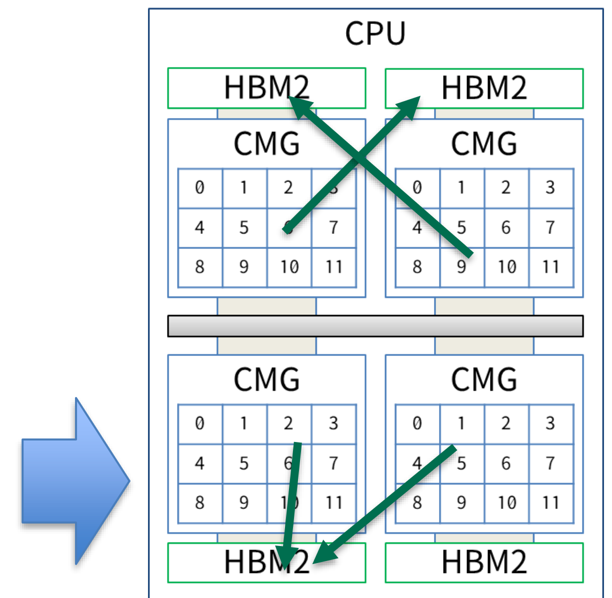
補足2：1プロセスOpenMPプログラムで十分なメモリ転送性能を得るには？

- 1プロセスOpenMP実行は推奨されていないが、もちろん適切に実行すれば妥当な性能が得られる
  - STREAMベンチマークは1プロセスOpenMP実行で800MB/sを超える性能が出せる
  - ファーストタッチ（First Touch）をしっかりやれば妥当な性能が得られる



1ノードに1プロセスの実行でも、CMG上のコアから直結したメモリへのアクセスのみが行われている分には問題ない。メモリアクセス範囲が限定されており、ファーストタッチがしっかり行われているプログラムであれば、メモリ性能をフルに発揮できる。

異なるCMGへのメモリアクセスがあったり、メモリアクセス量が均等でなかったりすると、メモリ性能をフルに発揮できない。



  - 1プロセスOpenMPプログラム以外の場合でも同様に、CMGをまたいだメモリアクセスが起きないように注意すること

典型的なFirst Touchの例

- STREAMベンチマークの例

```
// 初期化部分
#pragma omp parallel for
for (j=0; j<N; j++) {
    a[j] = 1.0;
    b[j] = 2.0;
    c[j] = 0.0;
}
```

（省略）

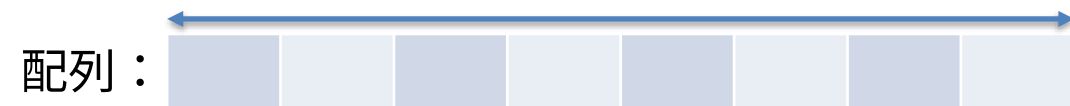
```
// 測定対象
#pragma omp parallel for
for (j=0; j<N; j++) {
    a[j] = b[j] + scalar * c[j];
}
```

初期化部分がOpenMP化されていない場合

- 初期化時

マスタースレッド（スレッド0）が初期化する

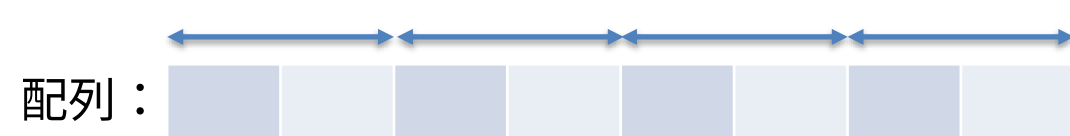
配列：



- 計算時

スレッド0　スレッド1　スレッド2　スレッド3

配列：



この場合、配列全体のキャッシュ情報がマスタースレッドに置かれた状態で計算が行われるため、計算時にスレッド0以外のメモリアクセスが遅くなる。
初期化時に計算時と同じ並列化を施しておけば、遅くならない。
コード例はC言語版だが、Fortranでも同様。