2021.05.13版

クラウドシステム向けのプロセス・スレッド配置方法

（Type IIでIntelコンパイラ+Intel MPIを使う場合も参考にしてください）

更新履歴

- 2020.08.27　初版公開
- 2021.03.05　一部更新
- 2021.03.26　細かい修正
- 2021.05.11　cx-shareおよびcl-share利用時の注意点を追加
- 2021.05.13　I_MPI_DEBUG出力情報を追加

cx-shareおよびcl-share利用時の注意点

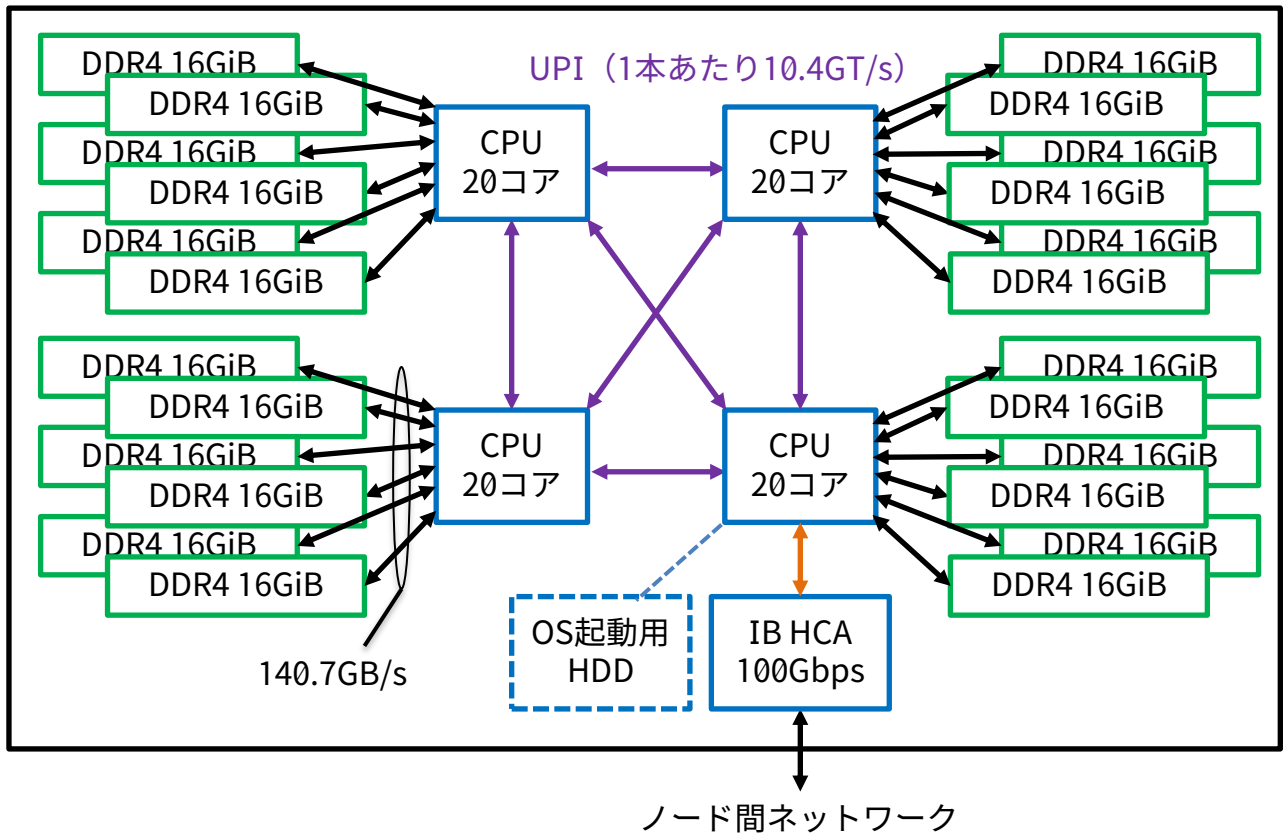
- Type IIサブシステム向けのcx-shareおよびクラウドシステム向けのcl-shareでIntelコンパイラやIntelMPIを利用する場合は、バッチジョブ実行時に割り当てられる資源情報と環境変数等で指定する資源割り当て情報に齟齬があると問題が起きます
  - 性能が低下したり、プログラムが実行できなかったりすることがあります
- cx-shareやcl-shareを利用する際にはスレッドやプロセスの割り当てを細かく指定しないようにしてください
- ジョブ実行時にはプログラム実行前に　`unset I_MPI_PIN_DOMAIN`　を実行するようにしてください

```
#!/bin/bash -x
#PJM -L rscgrp=cx-share
#PJM -L elapse=00:10:00
#PJM -j
#PJM -S

module load intel
unset I_MPI_PIN_DOMAIN
mpiexec -n ……
```

```
#!/bin/bash -x
#PJM -L rscgrp=cl-share
#PJM -L elapse=00:10:00
#PJM -j
#PJM -S

module load intel
unset I_MPI_PIN_DOMAIN
mpiexec -n ……
```

クラウドシステム向けのプロセス・スレッド配置方法

- Intel Xeon CPUのみ搭載のクラウドシステムではIntelコンパイラ+Intel MPIの利用が基本
- 20コアXeon×4ソケット環境のため
  「4プロセス」×「1プロセスあたり20スレッド」
  が基本の実行形態となる
  - もちろん、80スレッドOpenMP並列実行や80プロセスMPI実行を行っても構わないが、1プロセス中のスレッドが複数のCPUソケットにまたがらない実行が基本となる
- 他のソケットにつながったメモリへのアクセスは遅くなるため、ソケットローカルのメモリを優先して使いたい
  → `numactl -l` の指定は基本
- ネットワークにつながっているCPUがCPU0として扱われる



Intelコンパイラを用いて作成したOpenMPプログラムのスレッド配置

- 環境変数OMP_NUM_THREADSとKMP_AFFINITYの組み合わせにより指定する

- 基本の配置方法は以下の三つ
  - KMP_AFFINITY=granularity=fine,compact
  - KMP_AFFINITY=granularity=fine,scatter
  - KMP_AFFINITY=granularity=fine,balanced

	ソケット0	ソケット1	ソケット2	ソケット3
KMP_AFFINITY=granularity=fine,compact	0,1,2,3,4,5,6,7			
KMP_AFFINITY=granularity=fine,scatter	0, 4	1, 5	2, 6	3, 7
KMP_AFFINITY=granularity=fine,balanced	0, 1	2, 3	4, 5	6, 7

  - compactは、まずソケット0の全コアを埋める、次はソケット1、……という配置
  - scatterとbalancedは全ソケットに均等になるような配置、ただし配置順序が図のように異なる
  - 実行するプログラムにあわせて選べば良い：balancedの方が隣接プロセス間通信の性能が期待できる配置と言える

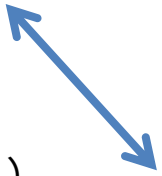
※コンパイラのドキュメントを見るとbalancedは使えないようなのだが、試してみたら使えた

- 「This affinity type is supported on the CPU only for single socket systems.」と書かれているため、1ソケットの環境でないと使えない気がするのだが？

配置状況の確認方法

- verboseオプション指定時の情報出力の例
  - KMP_AFFINITY=verbose,granularity=fine,compact

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: 0-79
OMP: Info #213: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #276: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #156: KMP_AFFINITY: 80 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #191: KMP_AFFINITY: 4 sockets x 20 cores/socket x 1 thread/core (80 total cores)
OMP: Info #215: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to socket 0 core 0
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to socket 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to socket 0 core 2
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to socket 0 core 3
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to socket 0 core 4
（省略、全80の割り当て情報が並ぶ、coreの数字は一部抜けていたりする）
OMP: Info #251: KMP_AFFINITY: pid 70 tid 70 thread 0 bound to OS proc set 0
OMP: Info #251: KMP_AFFINITY: pid 70 tid 71 thread 1 bound to OS proc set 1
OMP: Info #251: KMP_AFFINITY: pid 70 tid 72 thread 2 bound to OS proc set 2
OMP: Info #251: KMP_AFFINITY: pid 70 tid 73 thread 3 bound to OS proc set 10
```



対応する部分を読み合わせれば
配置がわかる

Intel MPIのプロセス配置方法

- 環境変数や引数で配置方法を指定する
- よく使う設定
  - 引数 -n 数字：プロセス数
  - 引数 -ppn 数字、もしくは環境変数 I_MPI_PERHOST：ノードあたり何プロセス配置したら次のノードへ行くか
  - 環境変数 I_MPI_PIN_DOMAIN：MPIプロセス1つあたりいくつの連続したコアを確保するか
    - 具体的な数字以外にomp（OMP_NUM_THREADSを参照）やauto（総コア数÷プロセス数を参照）も指定できる。OpenMPとのハイブリッド実行を行う場合はKMP_AFFINITYもあわせて指定しないとプロセス内の複数スレッドが同じCPUコアに配置されることがあるため注意。
    - フラットMPIの場合はautoを指定すると均等間隔で配置してくれる。
  - 環境変数 I_MPI_PIN_ORDER：プロセスを配置する順番（compact, scatter, spread, bunch）
    - compactとscatterはKMP_AFFINITYと同様。spreadとbunchはbalanced的な挙動。
      - 複数コアがL2キャッシュを共有しているとspreadとbunchの配置に差が生じる（同じL2キャッシュを使うコアに優先的に配置するのがbunch、しないのがspread）が、本環境では同じになると思われる。
  - 環境変数 I_MPI_DEBUG：デバッグレベル（数字を大きくするほど詳細情報が出力される、よく使われるのは5）

--mpiオプションと-n指定

- mpiexecのプロセス数に直接影響するのは-nで指定した値であるため、常にmpiexecに-nオプションで本当に起動したいプロセス数を指定するようにしてください

- 利用手引書では--mpi procとmpiexecの-nでプロセス数を指定しているが、本当に重要な（優先される）のは後者のmpiexecの-nオプション
- $PJM_MPI_PROCには--mpi procで指定した値が入ってくる
- mpiexecの-nオプションを指定しないと、想定外の数のプロセス数が起動することがあるため注意が必要

実践例：OpenMP 1/2

- 1ソケット利用時の並列化性能を比較したい場合

```
# 1 socket OpenMP
export OMP_NUM_THREADS=2
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

export OMP_NUM_THREADS=4
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

export OMP_NUM_THREADS=8
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

export OMP_NUM_THREADS=16
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

export OMP_NUM_THREADS=20
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}
```

- 1ソケット20スレッドを基準にソケットを増やした際の性能を比較したい場合

```
# 1 socket OpenMP
export OMP_NUM_THREADS=20
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

# 2 socket OpenMP
export OMP_NUM_THREADS=40
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

# 3 socket OpenMP
export OMP_NUM_THREADS=60
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}

# 4 socket OpenMP
export OMP_NUM_THREADS=80
export KMP_AFFINITY=granularity=fine,compact
numactl -l ${BIN}
```

※${BIN}には実行ファイル名が入っているとする
export BIN=./a.out　などで設定しておく

実践例：OpenMP 2/2

- 1ソケット16スレッドを基準に
  ソケットを増やした際の性能を比較したい場合

```
# 1 socket OpenMP
export OMP_NUM_THREADS=16
export KMP_AFFINITY=granularity=fine,balanced
numactl -l ${BIN}

# 2 socket OpenMP
export OMP_NUM_THREADS=32
export KMP_AFFINITY=granularity=fine,balanced
numactl -l ${BIN}

# 3 socket OpenMP
export OMP_NUM_THREADS=48
export KMP_AFFINITY=granularity=fine,balanced
numactl -l ${BIN}

# 4 socket OpenMP
export OMP_NUM_THREADS=64
export KMP_AFFINITY=granularity=fine,balanced
numactl -l ${BIN}
```

- compactのままでスレッド数を20以下に減らすとソケット内のコアを埋め尽くすまで次のソケットに配置してくれないため、balancedを指定
- balancedでは総コア数÷指定スレッド数だけのスレッドを配置したら次のソケットを割り当て対象にしてくれる

実践例：1ノード、フラットMPI

- 1ソケット内フラットMPIの性能を比較したい場合

```
# 1 socket MPI
export I_MPI_PIN_DOMAIN=1
export I_MPI_PIN_ORDER=compact
mpiexec -n 1 numactl -l ${BIN}
mpiexec -n 2 numactl -l ${BIN}
mpiexec -n 4 numactl -l ${BIN}
mpiexec -n 8 numactl -l ${BIN}
mpiexec -n 16 numactl -l ${BIN}
mpiexec -n 20 numactl -l ${BIN}
……
```

↑
I_MPI_PIN_ORDER=compactなのでソケット0のコア0から順番に使われる。20を超える場合はソケット1へ、40を超えるとソケット2へ、以下同様。

- 4ソケットにまたがるフラットMPIの性能を比較したい場合

```
# 4 socket MPI scatter
export I_MPI_PIN_DOMAIN=1
export I_MPI_PIN_ORDER=scatter
mpiexec -n 4 numactl -l ${BIN}
mpiexec -n 8 numactl -l ${BIN}
mpiexec -n 16 numactl -l ${BIN}
mpiexec -n 20 numactl -l ${BIN}
……
```

↑
I_MPI_PIN_ORDER=scatterなので、ソケット0のコア0→ソケット1のコア0→ソケット2のコア0……と分散して使われる。

```
# 4 socket MPI, balanced?
export I_MPI_PIN_DOMAIN=10
export I_MPI_PIN_ORDER=compact
mpiexec -n 8 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=5
export I_MPI_PIN_ORDER=compact
mpiexec -n 16 numactl -l ${BIN}
```

↑
I_MPI_PIN_DOMAINはプロセスを配置する際に1プロセスが占有するコアの数に相当する。これをうまく指定すれば、ソケット0内で均等分散してからソケット1へ……など様々な割り当てが可能。MPI+OpenMPハイブリッド実行の場合はこれを用いてスレッド配置範囲を調整する。

実践例：1ノード、MPI+OpenMPハイブリッド 1/2

- 1ソケット20スレッドを基準にソケットを増やして性能を比較したい場合

```
# 20 thread * 1 socket
export OMP_NUM_THREADS=20
export I_MPI_PIN_DOMAIN=20
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 1 numactl -l ${BIN}

# 20 thread * 2 socket
export OMP_NUM_THREADS=20
export I_MPI_PIN_DOMAIN=20
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 2 numactl -l ${BIN}

# 20 thread * 3 socket
export OMP_NUM_THREADS=20
export I_MPI_PIN_DOMAIN=20
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 3 numactl -l ${BIN}

# 20 thread * 4 socket
export OMP_NUM_THREADS=20
export I_MPI_PIN_DOMAIN=20
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 4 numactl -l ${BIN}
```

- おまけ：40スレッド×2プロセス

```
# 40 thread * 2 process
export OMP_NUM_THREADS=40
export I_MPI_PIN_DOMAIN=40
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 2 numactl -l ${BIN}
```

- I_MPI_PIN_DOMAIN=20にしているため1プロセスが占有するコア数は必ず20。プロセスあたりのOpenMPスレッド数を減らしたい場合はOMP_NUM_THREADSだけを減らす。（20コア分の枠を確保し、そのうちのxxコアだけを使う、という動作になる。）

- 補足（気になったので試してみた）
  - I_MPI_PIN_ORDER=scatterを追加してもプロセスの配置順序だけがscatterになったりはしないようだ。
  - KMP_AFFINITYをcompactからscatterにしても違いは生じないようだ。（プロセス内のスレッド配置順だけscatter的になったりはしなかった。）
  - 他の環境変数等との組み合わせのせいという可能性は残る。

実践例：1ノード、MPI+OpenMPハイブリッド 2/2

- もっとプロセス数を増やしたハイブリッド実行の例

```
# 10 thread * 8 process
export OMP_NUM_THREADS=10
export I_MPI_PIN_DOMAIN=10
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 8 numactl -l ${BIN}

# 8 thread * 8 process
export OMP_NUM_THREADS=8
export I_MPI_PIN_DOMAIN=10
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 8 numactl -l ${BIN}

# 5 thread * 16 process
export OMP_NUM_THREADS=5
export I_MPI_PIN_DOMAIN=5
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 16 numactl -l ${BIN}

# 4 thread * 16 process
export OMP_NUM_THREADS=4
export I_MPI_PIN_DOMAIN=5
export KMP_AFFINITY=granularity=fine,compact
mpiexec -n 16 numactl -l ${BIN}
```

- 各ソケットに2プロセス配置するなら80コア÷8プロセス=10でプロセスあたり10スレッドとなるが、スレッド数を2の冪乗数にしたいのであればI_MPI_PIN_DOMAINを10のままでOMP_NUM_THREADSを8にするのが良い。

- 各ソケットに4プロセス配置するなら80コア÷16プロセス=5でプロセスあたり5スレッドとなるが、スレッド数を2の冪乗数にしたいのであればI_MPI_PIN_DOMAINを5のままでOMP_NUM_THREADSを4にするのが良い。

実践例：複数ノード、フラットMPIまたはMPI+OpenMPハイブリッド

- ノード数とプロセス数さえ指定すれば均等に分散配置してくれる。
- 配置方法を調整したい場合にはパラメタで指定する必要がある。
- OMP_NUM_THREADSも指定されていれば、各プロセスは指定されたスレッド分のコア群に配置される。

- 配置情報を特に指定しないフラットMPI実行の場合、まず各ノードのコア0に1プロセスずつ配置、次に各ノードのコア1に1プロセスずつ配置……となる。
- これはI_MPI_PIN_DOMAINとI_MPI_PERHOSTにともに1を指定した状態。
  - I_MPI_DEBUG=5で確認できる

```
export OMP_NUM_THREADS=1
export I_MPI_PIN_DOMAIN=1
export I_MPI_PERHOST=1
mpiexec -n 16 numactl -l ${BIN}
```

```
export OMP_NUM_THREADS=1
export I_MPI_PIN_DOMAIN=1
export I_MPI_PERHOST=4
mpiexec -n 16 numactl -l ${BIN}
```

- I_MPI_PERHOSTを変更すると、この環境変数で指定した数のプロセスを配置してから次のノードへプロセスを配置するようになる。
- I_MPI_PIN_ORDERをscatterにすると、2周目のコア配置がコア1ではなくコア20など離れた（scatter的な）配置になる。
- I_MPI_PIN_DOMAINを増やすと、1プロセス配置する際に確保するコア数が増える。OpenMPのスレッドはその確保した枠内に配置する。
  - 1ノードの場合と同様の挙動

- 次ページ以降に4ノード×8プロセスのバリエーション例を示す。

ジョブスクリプトと配置の対応の例 1/4

```
#!/bin/bash -x
#PJM -L rscgrp=cl-extra
#PJM -L node=4
#PJM --mpi proc=32
#PJM -L elapse=00:10:00
#PJM -j
#PJM -S

module load intel
export OMP_DISPLAY_ENV=true
export I_MPI_DEBUG=5

BIN=./a.out

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}
```

- 4ノード、合計32プロセス利用することの宣言。
- OpenMPに関する環境変数の情報を細かく表示させるオプション
- MPIに関する環境変数の情報を細かく表示させるオプション

- I_MPI_PIN_DOMAIN=10のためプロセスが配置される際の単位は10コア、OMP_NUM_THREADS=8のためプロセス内のOpenMPに使われるのは8コア。
- I_MPI_PERHOST=1のため、1プロセス配置したら次のノードへ。
- I_MPI_PIN_ORDER=compactのため、ノードにプロセスを配置する際は前回の配置から近いところ。（ノード内のプロセス配置順序だけみたらcompact的。）
- numactl -lのため、ソケットから近いメモリだけを使う。

I_MPI_DEBUG=5出力情報の一部の例

```
[0] MPI startup(): Rank    Pid      Node name  Pin cpu
[0] MPI startup(): 0       78       dl097      {0,1,2,3,4,5,6,7,8,9}
[0] MPI startup(): 1       11       dl098      {0,1,2,3,4,5,6,7,8,9}
[0] MPI startup(): 2       11       dl099      {0,1,2,3,4,5,6,7,8,9}
[0] MPI startup(): 3       11       dl100      {0,1,2,3,4,5,6,7,8,9}
[0] MPI startup(): 4       79       dl097      {10,11,12,13,14,15,16,17,18,19}
[0] MPI startup(): 5       12       dl098      {10,11,12,13,14,15,16,17,18,19}
[0] MPI startup(): 6       12       dl099      {10,11,12,13,14,15,16,17,18,19}
[0] MPI startup(): 7       12       dl100      {10,11,12,13,14,15,16,17,18,19}
[0] MPI startup(): 8       80       dl097      {20,21,22,23,24,25,26,27,28,29}
[0] MPI startup(): 9       13       dl098      {20,21,22,23,24,25,26,27,28,29}
[0] MPI startup(): 10      13       dl099      {20,21,22,23,24,25,26,27,28,29}
[0] MPI startup(): 11      13       dl100      {20,21,22,23,24,25,26,27,28,29}
[0] MPI startup(): 12      81       dl097      {30,31,32,33,34,35,36,37,38,39}
[0] MPI startup(): 13      14       dl098      {30,31,32,33,34,35,36,37,38,39}
[0] MPI startup(): 14      14       dl099      {30,31,32,33,34,35,36,37,38,39}
[0] MPI startup(): 15      14       dl100      {30,31,32,33,34,35,36,37,38,39}
[0] MPI startup(): 16      82       dl097      {40,41,42,43,44,45,46,47,48,49}
[0] MPI startup(): 17      15       dl098      {40,41,42,43,44,45,46,47,48,49}
[0] MPI startup(): 18      15       dl099      {40,41,42,43,44,45,46,47,48,49}
[0] MPI startup(): 19      15       dl100      {40,41,42,43,44,45,46,47,48,49}
[0] MPI startup(): 20      83       dl097      {50,51,52,53,54,55,56,57,58,59}
[0] MPI startup(): 21      16       dl098      {50,51,52,53,54,55,56,57,58,59}
[0] MPI startup(): 22      16       dl099      {50,51,52,53,54,55,56,57,58,59}
[0] MPI startup(): 23      16       dl100      {50,51,52,53,54,55,56,57,58,59}
[0] MPI startup(): 24      84       dl097      {60,61,62,63,64,65,66,67,68,69}
[0] MPI startup(): 25      17       dl098      {60,61,62,63,64,65,66,67,68,69}
[0] MPI startup(): 26      17       dl099      {60,61,62,63,64,65,66,67,68,69}
[0] MPI startup(): 27      17       dl100      {60,61,62,63,64,65,66,67,68,69}
[0] MPI startup(): 28      85       dl097      {70,71,72,73,74,75,76,77,78,79}
[0] MPI startup(): 29      18       dl098      {70,71,72,73,74,75,76,77,78,79}
[0] MPI startup(): 30      18       dl099      {70,71,72,73,74,75,76,77,78,79}
[0] MPI startup(): 31      18       dl100      {70,71,72,73,74,75,76,77,78,79}
```

Node nameはノードのID、今回は以下のように割り当たっている：
- ノード0：dl097
- ノード1：dl099
- ノード2：dl099
- ノード3：dl100

・Rank0から順に、ノード0,1,2,3……と1つずつ割り当たっている
・1周するとPin cpuが10増えている
といった具合に、前ページの割り当て方式に対応していることが確認できる。
Pid(プロセスID)はノードローカルなプロセスIDであるため、特に気にしなくて良い。

ジョブスクリプトと配置の対応の例 2/4

```
#!/bin/bash -x
#PJM -L rscgrp=cl-extra
#PJM -L node=4
#PJM --mpi proc=32
#PJM -L elapse=00:10:00
#PJM -j
#PJM -S

module load intel
export OMP_DISPLAY_ENV=true
export I_MPI_DEBUG=5

BIN=./a.out

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}
```

- 4ノード、合計32プロセス利用することの宣言。
- OpenMPに関する環境変数の情報を細かく表示させるオプション
- MPIに関する環境変数の情報を細かく表示させるオプション

- I_MPI_PIN_DOMAIN=10のためプロセスが配置される際の単位は10コア、OMP_NUM_THREADS=8のためプロセス内のOpenMPに使われるのは8コア。
- I_MPI_PERHOST=2のため、2プロセス配置したら次のノードへ。
- I_MPI_PIN_ORDER=compactのため、ノードにプロセスを配置する際は前回の配置から近いところ。（ノード内のプロセス配置順序だけみたらcompact的。）
- numactl -lのため、ソケットから近いメモリだけを使う。

ジョブスクリプトと配置の対応の例 3/4

```
#!/bin/bash -x
#PJM -L rscgrp=cl-extra
#PJM -L node=4
#PJM --mpi proc=32
#PJM -L elapse=00:10:00
#PJM -j
#PJM -S

module load intel
export OMP_DISPLAY_ENV=true
export I_MPI_DEBUG=5

BIN=./a.out

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}
```

- 4ノード、合計32プロセス利用することの宣言。
- OpenMPに関する環境変数の情報を細かく表示させるオプション
- MPIに関する環境変数の情報を細かく表示させるオプション

- I_MPI_PIN_DOMAIN=10のためプロセスが配置される際の単位は10コア、OMP_NUM_THREADS=8のためプロセス内のOpenMPに使われるのは8コア。
- I_MPI_PERHOST=1のため、1プロセス配置したら次のノードへ。
- I_MPI_PIN_ORDER=compactのため、ノードにプロセスを配置する際は前回の配置から遠いところ。（ノード内のプロセス配置順序だけみたらscatter的。）
- numactl -lのため、ソケットから近いメモリだけを使う。

- 一般的なscatterの割り当て的にはソケット0の次はソケット2に配置されることが期待される気もするが、ソケット間の距離は均等であり、ソケットを埋め尽くす前に次のソケットへ行っているという点で確かにscatter。
- 実際に配置されるコア番号を見てみると、ソケット内で完全に連続したコアが確保されるとは限らない。
- 実例
- プロセス(ランク)0は「0,1,2,10,11,3,4,12」で8スレッド（ノード0のソケット0上）
- プロセス(ランク)1も「0,1,2,10,11,3,4,12」で8スレッド（ノード1のソケット0上）

ジョブスクリプトと配置の対応の例 4/4

```
#!/bin/bash -x
#PJM -L rscgrp=cl-extra
#PJM -L node=4
#PJM --mpi proc=32
#PJM -L elapse=00:10:00
#PJM -j
#PJM -S

module load intel
export OMP_DISPLAY_ENV=true
export I_MPI_DEBUG=5

BIN=./a.out

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=compact
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=1
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}

export I_MPI_PIN_DOMAIN=10
export I_MPI_PERHOST=2
export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=8
mpiexec -n 32 numactl -l ${BIN}
```

- 4ノード、合計32プロセス利用することの宣言。
- OpenMPに関する環境変数の情報を細かく表示させるオプション
- MPIに関する環境変数の情報を細かく表示させるオプション

- I_MPI_PIN_DOMAIN=10のためプロセスが配置される際の単位は10コア、OMP_NUM_THREADS=8のためプロセス内のOpenMPに使われるのは8コア。
- I_MPI_PERHOST=2のため、2プロセス配置したら次のノードへ。
- I_MPI_PIN_ORDER=compactのため、ノードにプロセスを配置する際は前回の配置から遠いところ。（ノード内のプロセス配置順序だけみたらscatter的。）
- numactl -lのため、ソケットから近いメモリだけを使う。

- 一般的なscatterの割り当て的にはソケット0の次はソケット2に配置されることが期待される気もするが、ソケット間の距離は均等であり、ソケットを埋め尽くす前に次のソケットへ行っているという点で確かにscatter。
- 実際に配置されるコア番号を見てみると、ソケット内で完全に連続したコアが確保されるとは限らない。
- 実例
- プロセス(ランク)0は「0,1,2,10,11,3,4,12」で8スレッド（ノード0のソケット0上）
- プロセス(ランク)1は「20,21,22,30,31,23,24,32」で8スレッド（ノード0上のソケット1上）

まとめ

- クラウドシステムで多く使われると思われるIntelコンパイラ + Intel MPIに絞ってプロセスとスレッドの配置を解説した
  - Type IIでIntelコンパイラ + Intel MPIを使う場合も参考になるはずである
  - OMP_環境変数、KMP_環境変数、I_MPI_環境変数が出てくるため若干わかりにくいかも知れないが、「よくある割り当て方法」であれば難しくないはずである
- 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
```

```c
#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