インテル® Xeon® CPU マックス・シリーズの設定およびチューニング・ガイド

この記事は、インテル® デベロッパー・ゾーンに公開されている「Intel® Advisor Cookbook」の「Intel® Xeon® CPU Max Series Configuration and Tuning Guide」(バージョン June 2023) の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

資料番号: 354227-002JA
June 2023

法務上の注意書き

インテルのテクノロジーを使用するには、対応したハードウェア、ソフトウェア、またはサービスの有効化が必要となる場合があります。絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

実際の費用と結果は異なる場合があります。

本資料に記載されているインテル製品に関する侵害行為または法的調査に関連して、本資料を使用または使用を促すことはできません。本資料を使用することにより、お客様は、インテルに対し、本資料で開示された内容を含む特許クレームで、その後に作成したものについて、非独占的かつロイヤルティー無料の実施権を許諾することに同意することになります。

ここに記載されているすべての情報は、予告なく変更されることがあります。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

開発コード名は、インテルが開発中で一般に公開されていない製品、テクノロジー、サービスを識別するために使用されます。これらは「商用」の名称ではなく、商標として機能することを意図したものではありません。

本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスも許諾するものではありません。唯一の例外として、a) 改変されていないコードは公開できます、b) 本資料に含まれるコードは、http://opensource.org/licenses/0BSD のとおり、ゼロ条項 BSD オープンソース・ライセンス (0BSD) の対象になるものとします。本資料に従って、本資料で言及されているインテル製品上で実行されるソフトウェア実装を作成することができます。本資料を変更、または派生物を作成する権利は付与されていません。

改訂履歴

日付	改訂番号	説明
2023 年 2 月	001	初版リリース
2023 年 6 月	002	最初の改訂

1. はじめに

1.1 対象者

本資料は、インテル® Xeon® CPU マックス・シリーズ上でアプリケーションを実行および最適化するシステム管理者とアプリケーション・エンジニア向けのドキュメントです。

1.2 用語集

表 1. 略語の定義

略語	用語	定義
BIOS	Basic Input/Output Service	
HBM	高帯域幅メモリー	
1LM	1 レベル・メモリー・モードまたはフラットモード	HBM と DDR が個別のアドレス空間としてソフトウェアに公開されるモード。
2LM	2 レベル・メモリー・モードまたはキャッシュモード	HBM が DDR のメモリー・サイド・キャッシュとして使用されるモード。ソフトウェアは DDR アドレス空間のみを見ることができ、HBM は DDR の透過的なメモリー・サイド・キャッシュとして機能します。
	「Fake (偽)」 NUMA モード	この機能は、Linux* カーネルのブートオプション (numa=fake) を使用すると有効になり、システムの物理メモリーを Fake NUMA ノードに分割できます。つまり、Fake NUMA を使用すると、一様な物理メモリー領域である物理 NUMA ノードを、複数の NUMA ノードとしてアプリケーションに公開できます。

1.3 参考文献

表 2. 参考文献

説明	URL
インテル® アーキテクチャー命令セット拡張プログラミング・リファレンス	https://software.intel.com/content/www/us/en/develop/down-load/intel-architecture-instruction-setextensions-programming-reference.html (英語)
memkind ライブラリー	http://memkind.github.io/memkind/ (英語)
libnuma API	https://man7.org/linux/man-pages/man3/numa.3.html (英語)
hbwmalloc API	http://memkind.github.io/memkind/man_pages/hbwmalloc.html (英語)
インテル® メモリー・レイテンシー・チェッカー (インテル® MLC)	https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html (英語)
STREAM ベンチマーク	https://www.cs.virginia.edu/stream/ (英語)
インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL)	https://www.xlsoft.com/jp/products/intel/perflib/mkl/index.html
インテル® oneMKL デベロッパー・ガイド (Linux* 版)	https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/top/intel-oneapi-math-kernel-library-benchmarks/intel-distribution-for-linpack-benchmark-1/overview-intel-distribution-for-linpack-benchmark.html (英語)

2. インテル® Xeon® CPU マックス・シリーズ

2.1 プロセッサーのブロック図

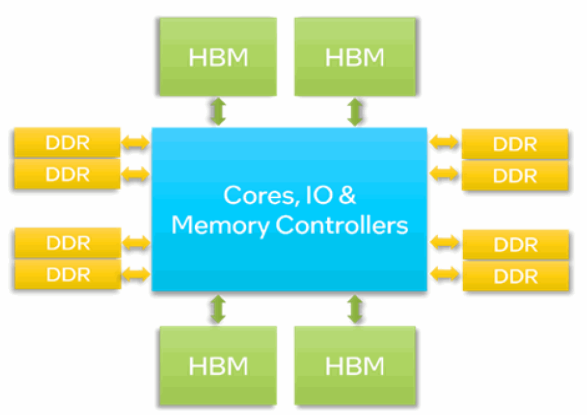

図 1: プロセッサーのブロック図

インテル® Xeon® CPU マックス・シリーズには、8 チャネルの DDR メモリーに加え、プロセッサーあたり合計 64GB の高帯域幅メモリー (HBM) 容量を持つ 4 つの HBM2e スタックが搭載されています。

2 ソケットのシステムでは、2 つのプロセッサーは最大 4 つのインテル® ウルトラ・パス・インターコネクト (インテル® UPI) リンクで接続されます。2 ソケットのシステムの HBM 容量は合計 128GB です。

2.1.1 HBM スタックとその仕様

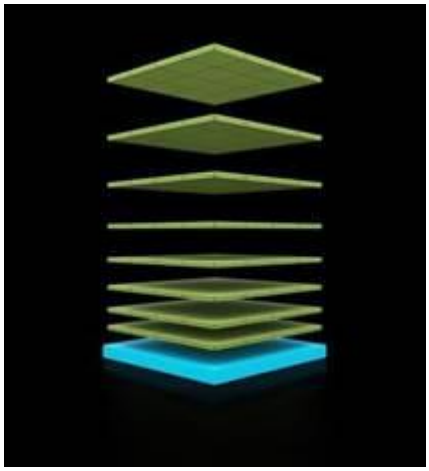

図 2: HBM 内のメモリースタック

HBM メモリーは、ワイドバスを持つ複数の DRAM メモリースタックで構成されます。各スタックには、下部のロジックダイ上にスタックされた 8 つの DRAM が含まれます。インテル® Xeon® CPU マックス・シリーズには、合計 64GB の HBM を持つ 4 つのスタックがあります。

3. ハードウェア構成

3.1 CPU 構成

インテル® Xeon® CPU マックス・シリーズ (パッケージまたはソケット) の HBM および DDR メモリーは、3 つのメモリーモードと 2 つのクラスターモードで構成できます。

このセクションでは、ハードウェアの観点から各モードについて説明します。OS でこれらのモードを設定する方法についてはセクション 5 で、アプリケーションで使用する方法についてはセクション 6 で説明します。

3.1.1 メモリーモード

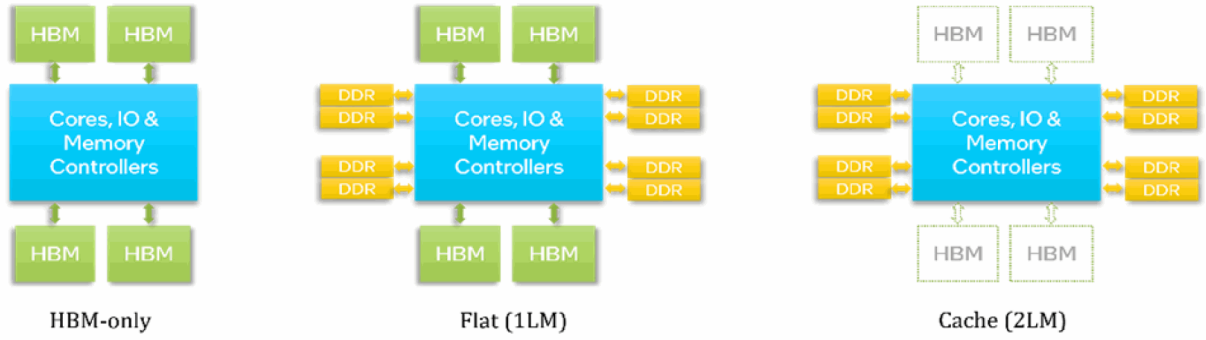

図 3: HBM メモリーモード

プロセッサーは、3 つのメモリーモードを使用して、ソフトウェア (OS とアプリケーション) に HBM を公開します。

3.1.1.1 HBM 専用

DDR が装着されていない場合、HBM 専用モードが選択されます。このモードでは、OS とアプリケーションが使用できるメモリーは HBM のみです。OS はインストールされているすべての HBM を見ることができ、アプリケーションは OS が公開する HBM を参照できます。したがって、OS とアプリケーションは HBM を容易に利用できます。ただし、OS、バックグラウンド・サービス、アプリケーションは、利用可能な HBM 容量 (プロセッサーごとに 64GB) を共有しなければなりません。

3.1.1.2 フラット (1LM) モード

DDR が装着されている場合、ブート時に BIOS メニューから フラット (1 レベルメモリー (1LM) とも呼ばれる) モードを選択することで、HBM と DDR の両方をソフトウェアに公開できます。このモードでは、HBM と DDR は個別のアドレス空間 (NUMA ノード) としてソフトウェアに公開されます。HBM を利用するには、セクション 6.2 で説明するように、ユーザーは NUMA 対応ツール (`numactl` など) やライブラリーを使用する必要があります。通常のメモリープールの一部として HBM にアクセスするには、追加の OS 設定が必要です (セクション 5.2 を参照)。

3.1.1.3 キャッシュ (2LM) モード

DDR が装着されている場合、起動時に BIOS メニューからキャッシュ (2 レベルメモリー (2LM) とも呼ばれる) モードを選択することで、HBM を DDR のメモリー・サイド・キャッシュとして使用できます。ソフトウェアは DDR アドレス空間のみを見ることができ、HBM は DDR の透過的なメモリー・サイド・キャッシュとして機能します。キャッシュモードを使用する場合、アプリケーションやコマンドラインを変更する必要はありません。HBM はダイレクト・マップ・キャッシュであり、競合ミスを最小限に抑えるには、追加の設定が必要になる場合があります (セクション 5.2.1 を参照)。

3.1.2 クラスター (パーティション) モード

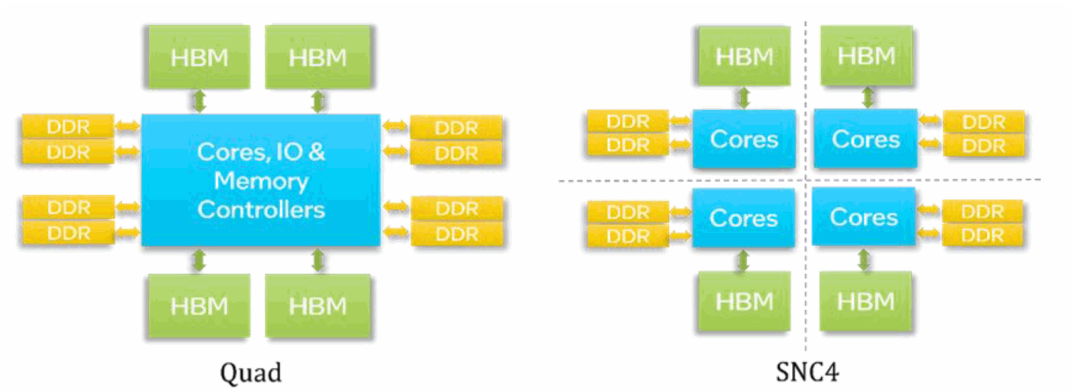

図 4: クラスターモード

クラスターモードは、プロセッサーが異なるアドレス空間 (NUMA ノード) にどのようにパーティショニング (分割) されるかを決定します。

クラスタリング (パーティショニング) により、コアは同じパーティション内のメモリー (HBM と DDR の両方) に、より高い帯域幅とより低いレイテンシーでアクセスできます。クラスターモードはメモリーモードとは異なります。インテル® Xeon® CPU マックス・シリーズには、2 つのクラスターモードがあります。

3.1.2.1 クワドラント

このモードは、単一のアドレス空間 (NUMA ノード) をソフトウェアに公開します。したがって、アプリケーションは NUMA に対応する追加手順を必要としません。このモードは、プロセッサーのすべてのコア間で大きなデータ構造を共有するアプリケーション (例えば、すべてのコアで実行され、大きなデータ構造を共有する OpenMP* アプリケーション) に適しています。

3.1.2.2 SNC4 (Sub-NUMA Clustering-4) - デフォルトのクラスターモード

このモードでは、各 CPU を 4 つのサブ NUMA クラスター・パーティションに分割します。各パーティションは 1 つまたは複数の NUMA ノードとしてソフトウェアに公開されます。したがって、各プロセッサーには少なくとも 4 つの NUMA ノードが存在します。このモードを使用するには、アプリケーションが NUMA に対応する必要があります。このモードは、クワドラント・モードよりも高い帯域幅と低いレイテンシーを提供します。このモードは、NUMA 対応のアプリケーション (MPI または MPI + OpenMP* アプリケーションなど) に適しています。

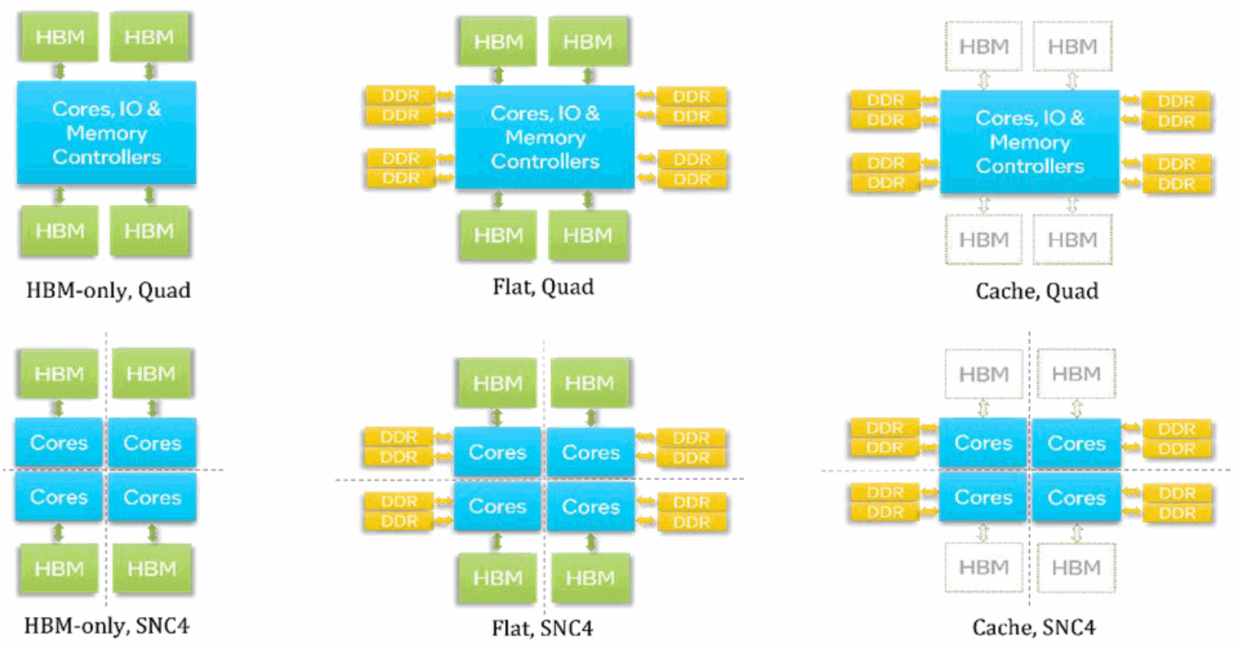

図 5: メモリーモード設定

インテル® Xeon® CPU マックス・シリーズには、図 5 に要約するように、3 つのメモリーモードと 2 つのクラスターモードを組み合わせた 6 つの構成オプションがあります。

3.2 マルチソケット構成

インテル® Xeon® CPU マックス・シリーズは、最大 4 つのインテル® UPI リンクで接続された 2 ソケット構成で利用できます。各ソケット (プロセッサー) は独立したアドレス空間 (NUMA ノード) です。したがって、クワドラント・モードの 2 ソケットのシステムには、少なくとも 2 つの NUMA ノードがあり、SNC4 モードの 2 ソケットのシステムには、少なくとも 8 つの NUMA ノードがあります。

3.3 DIMM 構成

各プロセッサーには 4 つの DDR メモリー・コントローラーがあり、各メモリー・コントローラーは 2 つのチャネルをサポートし、プロセッサーごとに合計 8 つのチャネルをサポートします。

3.3.1 HBM 専用モード

HBM 専用モードを利用するには、DIMM が取り外されている必要があります。一部の BIOS バージョンでは、物理的に取り外さなくても、BIOS オプションで DIMM を無効にすることができます。

3.3.2 フラットモード

図 6 は、単一 CPU のフラットモードにおけるすべての DIMM 構成オプションと、各 DIMM 構成が SNC4 をサポートしているかどうかをまとめたものです (HBM、DDR、および HBM + DDR)。

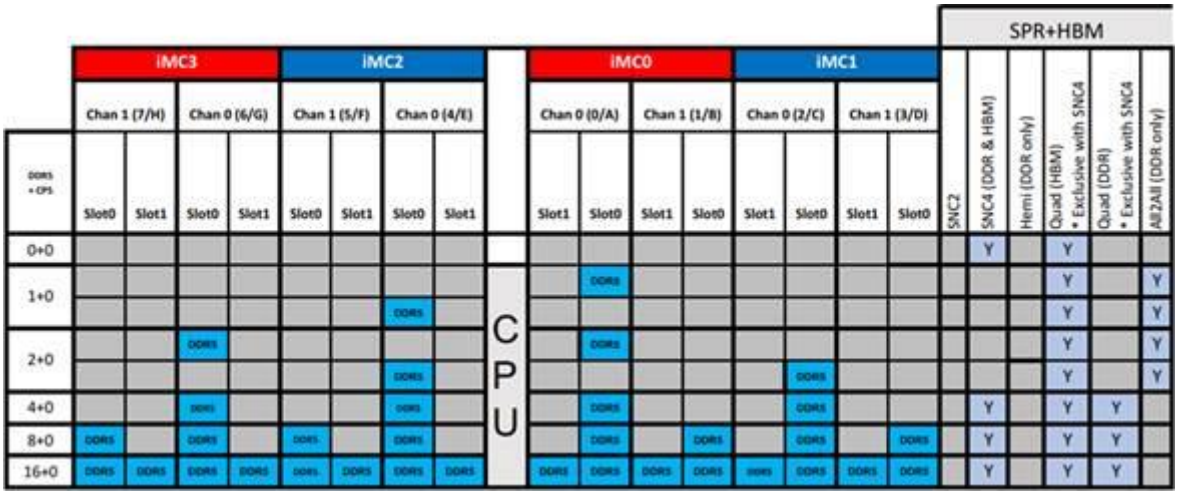

図 6: DIMM 構成オプション

すべてのモードは HBM でクワドラントまたは SNC4 をサポートします。ただし、HBM と DDR の両方で SNC4 をサポートするのは、対称 DIMM 構成の最後の 3 つのモードのみです。非対称 DIMM 構成のモードでは、DDR 空間は「All-to-All」と呼ばれる、非対称の低い帯域幅と非対称の高いレイテンシーの特殊なクラスターモードで構成されます。

チャネルの両方の DDR スロットを使用する場合 (最後の行)、チャネルあたり 1 スロットのみを使用する場合 (最後から 2 番目の行) よりも、帯域幅は低く、レイテンシーは高くなります。

3.3.3 キャッシュモード

キャッシュモードでは、4 つのメモリー・コントローラーすべてで対称 DIMM 構成が必要です。したがって、上図の最後の 3 行のみがキャッシュモードをサポートしています。

デュアルランク DDR DIMM はシングルランク DIMM よりも帯域幅が広いため、最高のパフォーマンスを得るには、デュアルランク DIMM を使用する必要があります。

3.4 BIOS 設定

このセクションでは、メモリーモードとクラスターモードを選択する BIOS オプションについて説明します。このセクションで示すメニューオプションは、インテル® ソフトウェア開発プラットフォーム用のものであり、BIOS プロバイダーによっては、特定のメニューオプションが**異なる**場合があります。

最新の BIOS には機能やパフォーマンスの拡張が含まれている可能性があるため、システムが古い BIOS バージョンを使用している場合は、最新の BIOS バージョンにアップグレードしてください。

3.4.1 メモリーモードの選択

メモリーモードを選択するには、BIOS で以下のメニューを選択します。

EDKII -> Socket Configuration -> Memory Configurations -> Memory Map -> Volatile Memory Mode -> 1LM/2LM

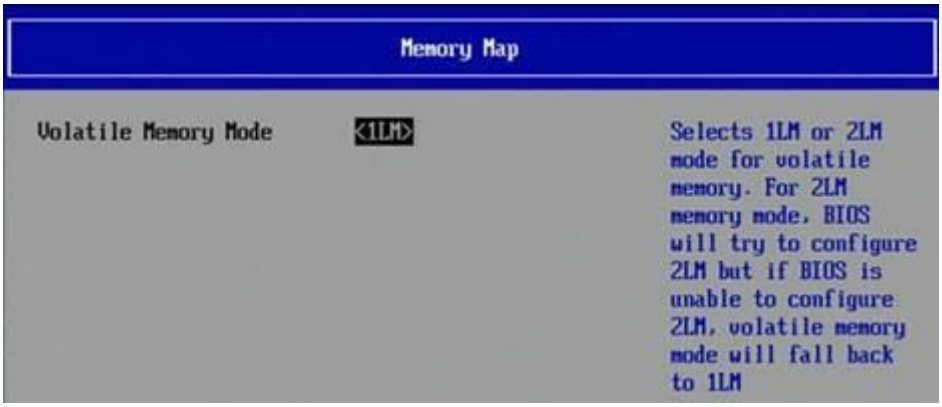

図 7: メモリーモードの選択

3.4.2 クラスターモードの選択

クラスターモードを選択するには、BIOS で以下のメニューを選択します。

EDKII → Socket configuration → Uncore configuration → Uncore General Configuration → SNC (Sub Numa)

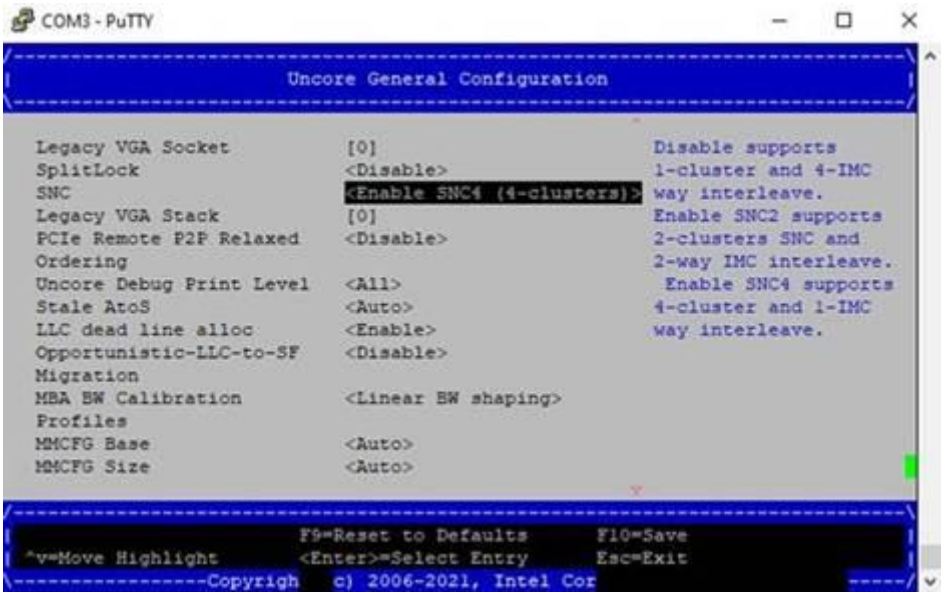

図 8: クラスターモードの選択

注: インテルの SYSCFG ユーティリティー (英語) を使用して、インテルのサーバー・プラットフォームの BIOS 設定の保存/復元と検査を行うことができます。

4. Linux* システム設定

4.1 共通の設定オプション

すべてのメモリーモードで以下の設定オプションを検討してください。

- スワップを無効にします。これは、容量が制限されている HBM 専用モードで特に有効です。スワップはパフォーマンスを大幅に低下させます。アプリケーションでスワップが発生する場合は、(ファイルシステムのキャッシュをクリアするなどして) メモリーを解放するか、ノード数を増やすことを検討してください。
- ゾーン再利用モードを有効にして、NUMA ミスを減らします。このモードは、NUMA ノードのサイズが小さい場合 (SNC4 クラスターモードなど) に有効です。Linux* ページ・アロケーターは、別の NUMA ノードに割り当てる前に、要求された NUMA ノードでページを簡単に再利用できるようになり、パフォーマンスを低下させる NUMA 交差が減少します。ただし、再利用アクティビティーによって、パフォーマンスにわずかなばらつきが生じる可能性があります。ゾーン再利用オプションは、以下のコマンドで有効にできます。これは再起動するたびに実行する必要があるため、初期化スクリプトを使用して自動化することを推奨します。

```
echo 2 > /proc/sys/vm/zone_reclaim_mode
```

- 各実行の前に、以下のコマンドを使用して、以前の実行でキャッシュされた内容が有用でない場合はファイルシステムのキャッシュをフラッシュし、メモリーを圧縮することを検討してください。これらのコマンドは root 権限を必要とするので、システム管理者は、バッチシステムのジョブプロローグの一部にするか、`setuid` バイナリーとして提供することを検討すべきです。

```
sync; echo 3 > /proc/sys/vm/drop_caches;
echo 1 > /proc/sys/vm/compact_memory
```

- Transparent Huge Pages (THP) を有効にすることを検討してください。ほとんどの HPC アプリケーションは THP の恩恵を受けます。ラージページを作成するためメモリーの圧縮が必要な場合、THP はオーバーヘッドを引き起こす可能性がありますが、管理者は、上記のように各実行前にメモリーを圧縮することでオーバーヘッドを軽減できます。
- `/dev/shm` (`tmpfs`) は利用可能なメモリーを減少させるため、ファイルの保存には使用しないでください。システム管理者は、ジョブ間の干渉を減らすジョブのプロローグの一部として `/dev/shm` をクリアすることを検討すべきです。
- 最新の安定版 Linux* カーネル (本資料の執筆時点では 5.15) の使用を検討してください。

4.2 役立つ Linux* ツール

4.2.1 numactl

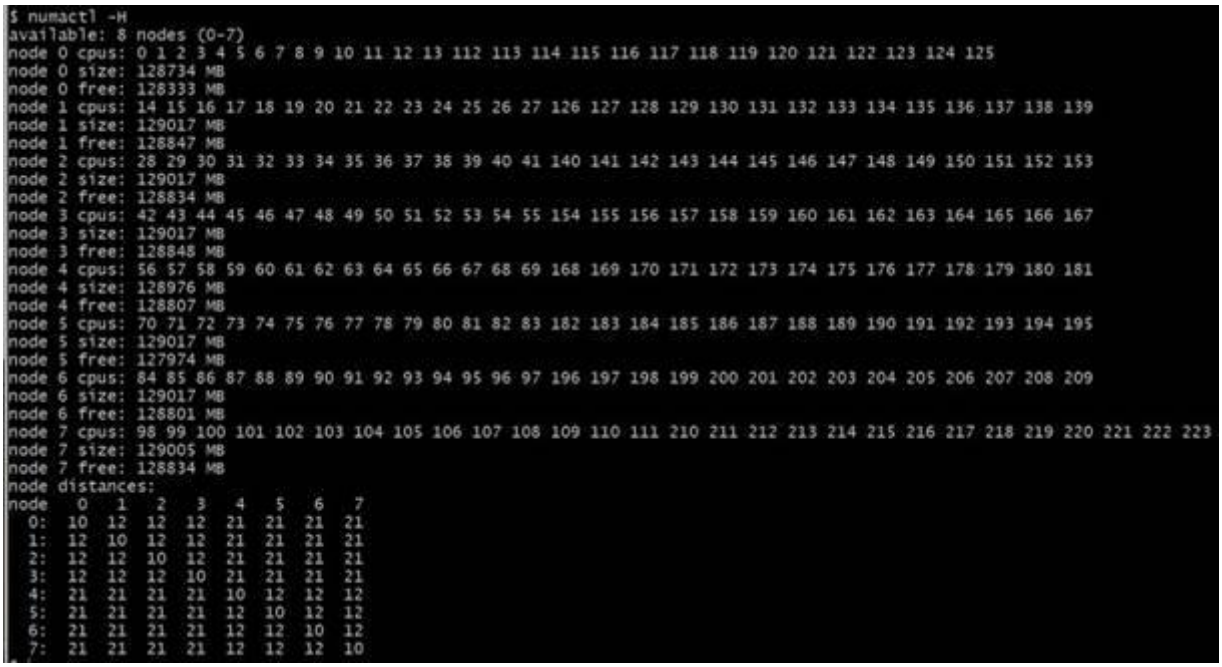

図 9: `numactl -H` の出力例

Linux* ユーティリティーの `numactl` は、システムの NUMA 構成を観察し、特定の NUMA ノードでアプリケーションを実行する場合に使用されます。システムの NUMA 構成を観察するには、`numactl -H` を使用します。以下は、`numactl -H` の出力例です。NUMA ノード数、CPU コア数、各 NUMA ノードのメモリー容量が表示され、その後に各ノードと他のノードの間の距離を表す行列が続いています。詳細は、`man numactl` を参照してください。

4.2.2 numastat

Linux* ユーティリティー `numastat` は、NUMA メモリー使用量に関するさまざまな統計情報を提供します。特に、以下のコマンドが便利です (詳細は `man numastat` を参照)。

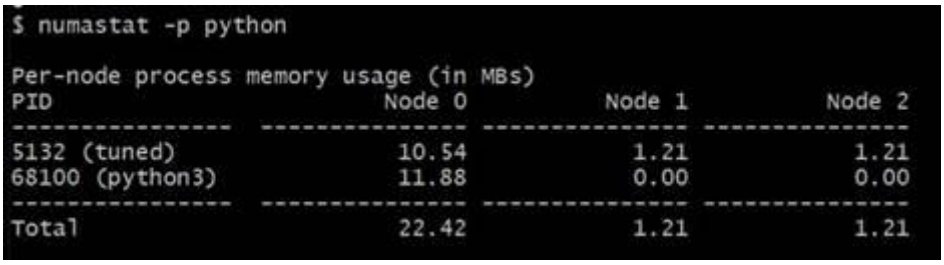

図 10: `numastat -p` の出力例

- `numastat -p <process_name>` は、図 11 に示すように特定のプロセスのメモリー使用量を表示します。

- `numastat -m` は、システム全体のメモリー使用量を表示します。

```
$ numastat
                           node0           node1
numa_hit                7038233469      7552949520
numa_miss                 31491495               0
numa_foreign                     0        31491495
interleave_hit           254896206       254893381
local_node              6905622525      7430407252
other_node               164102439       122542268
```

図 11: `numastat -m` の出力例

- `numastat` (引数なし) は、NUMA ヒット/ミス (ブートからの累積) を表示します。これは、予期しないパフォーマンス低下につながる NUMA ノードの交差 (`numa_miss`) を特定するのに役立ちます。これらの統計はブートからの累積であるため、ある実行が NUMA ミスに遭遇したかどうかを確認するには、各実行の前後に `numastat` を実行する必要があります。

4.2.3 `turbostat`

`turbostat` を使用して、root として (または setuid バイナリーとして) 実行した場合の、x86 アーキテクチャー・プロセッサーの消費電力、周波数、温度を調べることができます。`turbostat` は、システム冷却の問題を特定するのに役立ちます。例えば、以下はシステムの電力、周波数、温度を表示します。

```
• turbostat -qS                    # average for both CPUs
• turbostat -qS --cpu package      # separately for each CPU
```

```
$ turbostat -qS
Avg_MHz Busy%  Bzy_MHz TSC_MHz IPC    IRQ    SMI  POLL   C1ACPI C2ACPI POLL%  C1ACPI% C2ACPI% CPU%c1 CPU%c6 CoreTmp PkgTmp  PkgWatt RAMWatt PKG_%  RAM_%
1337    49.92  2679    1800    1.68   599693 0    5      669    8110   0.00   0.05    49.92   50.08  0.00   75      75      697.92  172.62  190.03  0.00
1341    49.93  2676    1807    1.70   601152 0    7      1038   8791   0.00   0.07    50.07   50.07  0.00   73      73      702.72  174.93  191.08  0.00
```

図 12: `turbostat` の例

4.2.4 `lscpu`

この標準 Linux* ユーティリティーは、NUMA ノード、コア数、ベース周波数、キャッシュサイズ、CPU フラグ (機能) など、高レベルのシステム設定の詳細を表示します。

4.2.5 `dmidecode` と `lshw`

これらの Linux* ユーティリティーを (root 権限で) 使用することで、HBM や DDR モジュールなど、搭載されているハードウェア・コンポーネントを検査することができます。

4.2.6 `htop`

`htop` ユーティリティーは、標準 Linux* の `top` ユーティリティーのようなツールですが、個々の CPU コア/スレッドの使用状況を視覚的に表示します。これは通常、NUMA の使用状況や MPI ランク、OpenMP* スレッドの配置を特定するのに役立ちます。さらに、システムのメモリー使用量も表示します。

4.2.7 `lstopo`

`lstopo` ユーティリティーは `hwloc` ライブラリーの一部であり、標準的なパッケージ・マネージャー (`dnf install hwloc` など) を使用して、パッケージとしてインストールできます。`lstopo` ユーティリティー (または `lstopo-no-graphics`) は、システムのハードウェア・トポロジーを表示します。

5. メモリーモード固有の設定

このセクションでは、各メモリーモードの OS 設定オプションについて説明します。

5.1 HBM 専用メモリーモード

HBM 専用モードで HBM を使用する場合、追加の設定は必要ありません。ただし、HBM 容量には限りがあるため、管理者はメモリー容量のオーバーヘッドを削減する追加の手順を実行できます。以下を検討してください。

- 不要なサービス (デーモン)、およびブート時に起動するドライバー (VNC サーバー、印刷/メール/その他のデーモン、パフォーマンス・プロファイル・ドライバーなど) を減らします。
- OS のファイルキャッシュと MPI バッファーのサイズを最小にします。
- 各実行の前に、ファイルシステムのキャッシュを消去し、メモリーを圧縮します (セクション 4.1 を参照)。

5.1.1 NUMA ノードのエミュレーション

図 13 は、HBM 専用モードの 2 ソケットのシステムにおける、クワドラントおよび SNC4 の NUMA ノード構成を要約したものです。クワドラント・モードでは 2 つの NUMA ノード (各ソケットに 1 つずつ) が作成され、SNC4 モードでは 8 つの NUMA ノード (各ソケットに 4 つずつ) が作成されます。各 NUMA ノードには、コアと HBM メモリーの両方が含まれます。

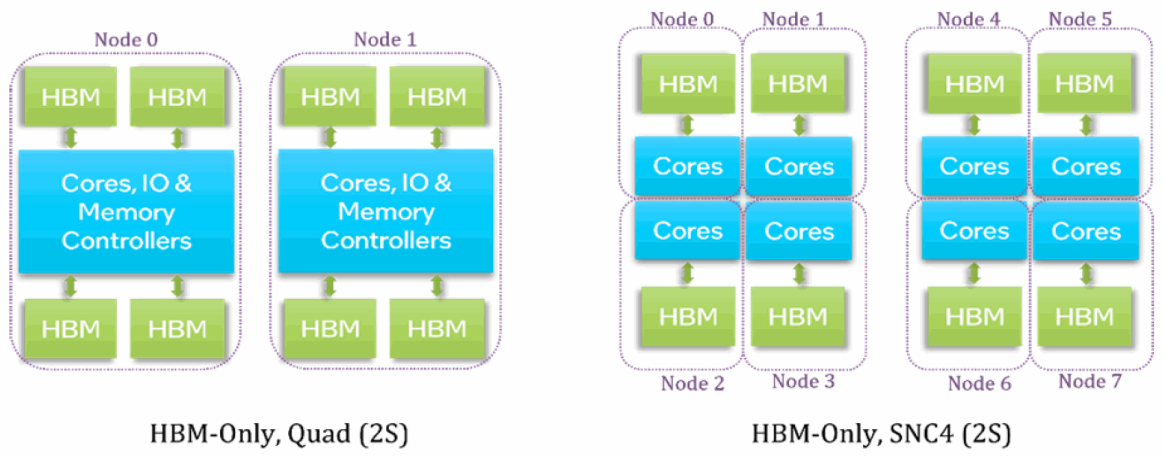

図 13: HBM 専用モードの NUMA ノード構成

`numactl -H` を使用して、NUMA ノードの構成と、各 NUMA ノード上の合計/空きメモリー量を確認します。

5.2 フラット・メモリー・モード

フラットモードは、セクション 3.4.1 で説明したように、DDR が装着されている場合に BIOS で 1LM を選択することで有効になります。しかし、これだけでは、デフォルトのメモリープールを介して OS やアプリケーションに HBM を公開することはできません。

デフォルトのメモリープールで HBM を有効にするには、追加の設定が必要です。

BIOS で 1LM を選択した後、システムは起動して DDR だけを OS とアプリケーションに公開します。HBM は特殊用途メモリーとしてマークされているため、デフォルトのメモリープールには HBM はまだ表示されていません。これは、OS がブートプロセス中に貴重な HBM メモリーを割り当てたり、予約するのを防ぐためです。ブートプロセス中、HBM は「隠されて」いて OS からは見えないため、OS は HBM メモリーを割り当てたり、予約することはできません。

以下は、フラットモードで起動し、HBM を公開する手順です。

1. BIOS メニューで 1LM を選択し (セクション 3.4.1 を参照)、OS を起動します。システムが起動すると、デフォルトのメモリープールには DDR だけが表示されます。`numactl -H` を使用して、これを確認できます。
2. 以下の Linux* パッケージをインストールします。

```
dnf install daxctl ndctl
```

3. 2 ソケットのシステムの場合、以下の `daxctl` コマンドを実行します (クワドラント・モードでは上の 2 つのコマンドのみが必要ですが、SNC4 ではすべてのコマンドが必要です)。これらのコマンドは root 権限で実行する必要があります。

```
## Base commands for both Quadrant andSNC4 cluster modes
##
daxctl reconfigure-device -m system-ram dax0.0
daxctl reconfigure-device -m system-ram dax1.0

## For SNC4 cluster mode, use the following additional commands:
##
daxctl reconfigure-device -m system-ram dax2.0
daxctl reconfigure-device -m system-ram dax3.0
daxctl reconfigure-device -m system-ram dax4.0
daxctl reconfigure-device -m system-ram dax5.0
daxctl reconfigure-device -m system-ram dax6.0
daxctl reconfigure-device -m system-ram dax7.0
```

ステップ 3 は、**システムが起動するたびに**実行する必要があります。そのため、上記のコマンドを OS 起動時に実行するスクリプトに記述しておくと便利です。

`numactl -H` を使用して、HBM ノードが表示されていること、および HBM の全容量に空きがあることを確認します。

5.2.1 フラットモードの NUMA ノードのエミュレーション

図 14 は、フラットモードの 2 ソケットのシステムにおける、クワドラント・モードと SNC4 モードの NUMA ノード構成を要約したものです。

- クワドラント・モードでは、4 つの NUMA ノード (コアに接続された 2 つの DDR ノードと、コアに接続されていない 2 つの HBM ノード) になります。
- SNC4 モードでは、16 個の NUMA ノード (コアに接続された 8 つの DDR ノードとコアに接続されていない 8 つの HBM ノード) になります。

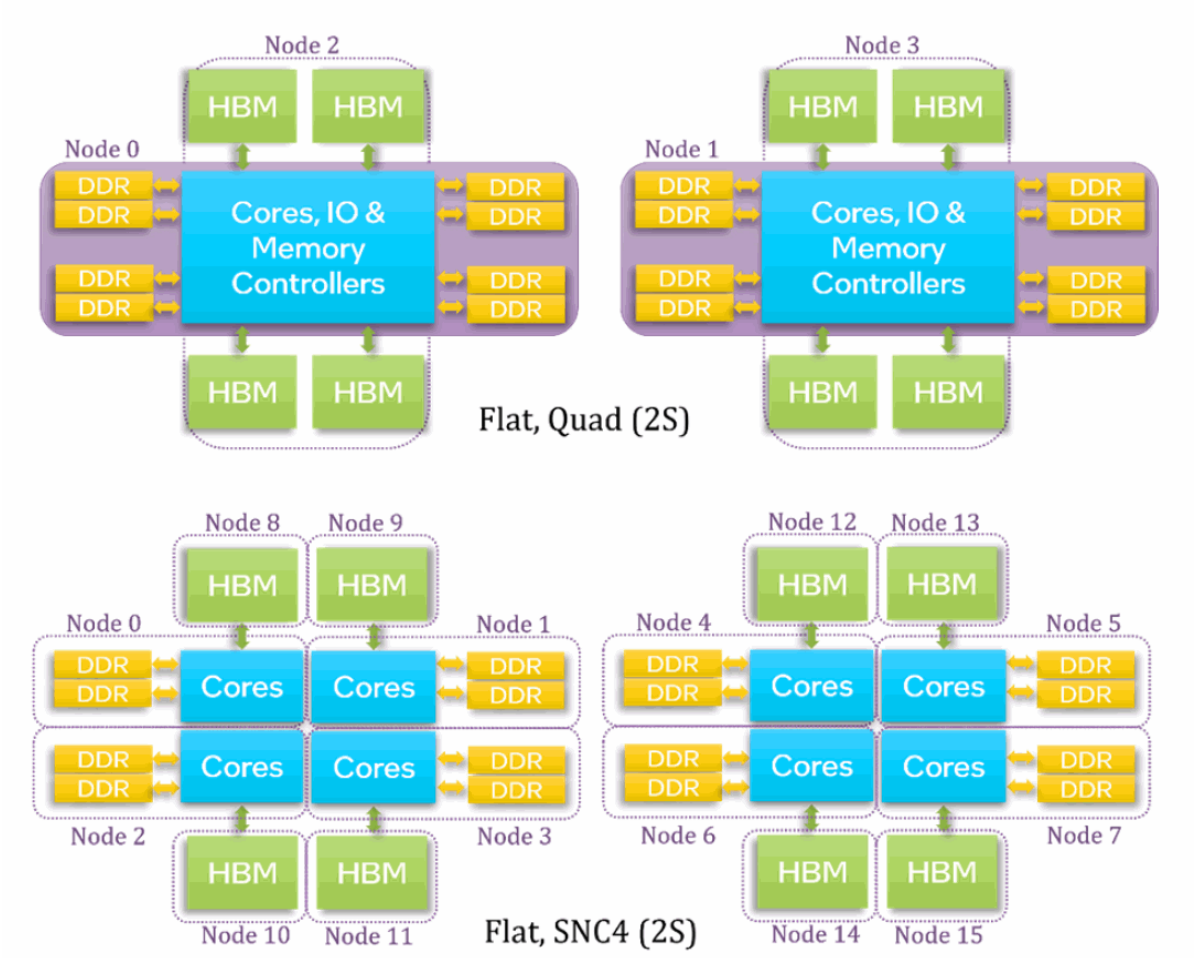

図 14: フラットモードの NUMA ノード構成

`numactl -H` を使用して、NUMA ノードの構成と、各 NUMA ノード上の合計/空きメモリー量を確認します。

5.3 キャッシュ・メモリー・モード

キャッシュモードを使用する場合、追加の設定は必要ありません。ただし、HBM キャッシュはダイレクトマップされたメモリー・サイド・キャッシュであるため、競合ミスによるアプリケーションへの影響を軽減するため、Fake NUMA を使用した OS の追加設定を強く推奨します。

5.3.1 キャッシュ・メモリー・モードでの Fake NUMA の使用

この機能は、Linux* カーネルのブートオプション (`numa=fake`) を使用すると有効になり、システムの物理メモリーを Fake NUMA ノードに分割できます。つまり、Fake NUMA を使用すると、一様な物理メモリー領域である物理 NUMA ノードを、複数の NUMA ノードとしてアプリケーションに公開できます。

例えば、キャッシュモードの 64GB の HBM と 128GB の DDR メモリーの構成について考えてみます。図 15 の左の図では、128GB の DDR アドレス空間の 2 つのラインが HBM の同じ場所にマップされ、64GB の HBM キャッシュに競合が生じています。HBM キャッシュはダイレクトマップされるため、2 つのラインのうち 1 つしかキャッシュに存在できません。

図 15 の右の図では、2 つの Fake NUMA ノードを作成した場合の効果を示しています。アプリケーションが Fake NUMA ノード (ノード 0 とします) 内に収まる場合、HBM キャッシュで競合ミスが発生しないことが保証されます。

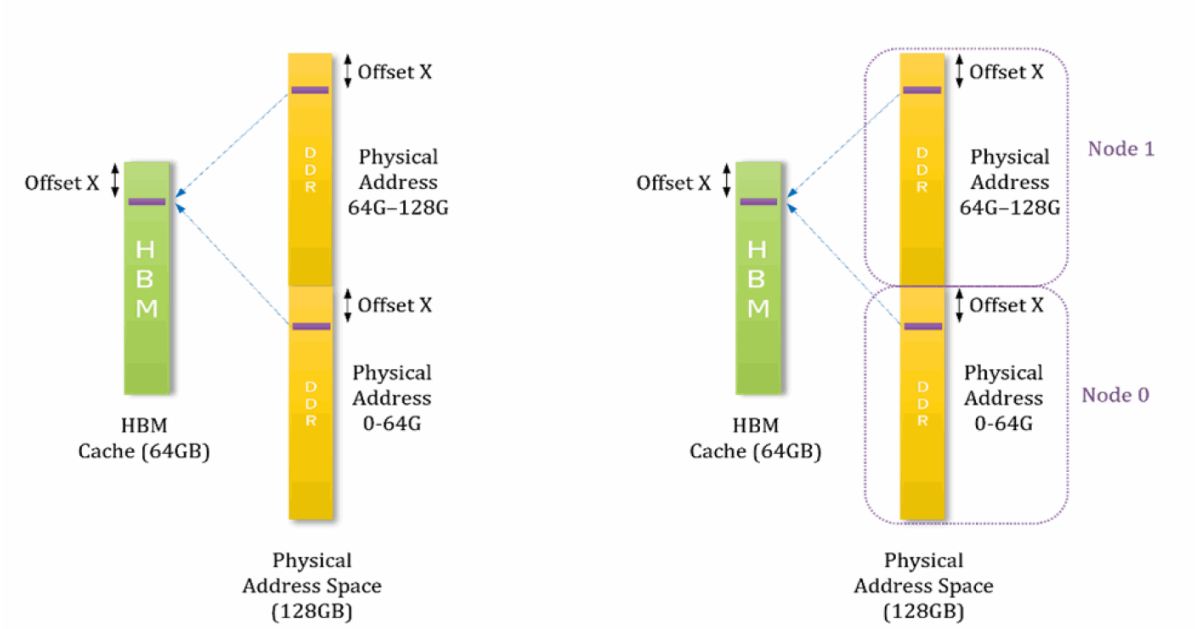

図 15: Fake NUMA ノードの例

128GB の DDR を持つシステム上に、それぞれ 64GB の容量を持つ 2 つの Fake NUMA ノードを作成するには、カーネルのブートオプション `numa=fake=2U` を使用します。これにより、各物理 NUMA ノードに対して 2 つの Fake NUMA ノードが作成されます。

Fake NUMA を使用しない場合、HBM 容量より小さいフットプリントのアプリケーションでも、物理メモリーの断片化により HBM キャッシュの競合ミスを引き起こす可能性があります。Fake NUMA を使用すると、Linux* カーネルはまず NUMA ノード 0 に、次に Fake NUMA ノード 1 に、というようにメモリーをシーケンシャルに割り当てます。これにより、Fake NUMA ノード内に収まるアプリケーションでは、HBM キャッシュで競合が発生しない割り当てが保証されます。そのようなアプリケーションは、ばらつきが小さくなり、最高のパフォーマンスを発揮します。

Fake NUMA ブートオプションでカーネルを起動した後、`numactl -H` を使用してノードが適切に分割されていることを確認します。Fake NUMA ノードが表示されない場合は、カーネルが `CONFIG_NUMA_EMU=y` オプションでビルドされていることを確認してください。

Fake NUMA ノードのサイズは、クワドラント・クラスター・モードでは約 64GB、SNC4 モードでは約 16GB にすべきです。

物理 NUMA ノードに属するすべての Fake NUMA ノードは、同じ CPU コアを共有します。そのため、Fake NUMA はアプリケーションの起動コマンドには影響しませんが、NUMA ノードの数は総 DDR 容量と総 HBM 容量の比率によって増加します。

Fake NUMA ノードが一杯になるとスワップを引き起こす可能性があるため、Fake NUMA を使用する場合は**スワップを無効にする**ことを強く推奨します。

Fake NUMA ノードは容量が小さいため、`zone_reclaim` を有効にすると、Fake NUMA ノードが一杯になったときに再利用アクティビティーが頻発して、パフォーマンスにわずかなばらつきが生じる場合があります。

Fake NUMA ノードでは、すべての標準 NUMA ツールを使用できます。例えば、`numactl -m 2 ./a.out` は、Fake NUMA ノード 2 のメモリーを使用してアプリケーションを起動します。同様に、`numastat` は Fake NUMA ノードのプロパティーを表示します。

5.3.2 ページシャッフル (ページのランダム化)

Linux* には、ページ割り当てをランダム化する機能があります。Fake NUMA を使用しない場合、ページシャッフルは、より一貫性のあるパフォーマンス結果 (例えば、起動したばかりのシステムと長期間稼働しているシステムの間で) の達成に有用である可能性があります。ページシャッフルを有効にすると、ページは物理メモリー内のランダムなページアドレスに割り当てられ、アプリケーションが起動するたびに、HBM キャッシュ内で競合するページが変わります。

この機能は、Linux* カーネル v5.4 以降でカーネルのブートオプション `page_alloc.shuffle=y` を使用して有効にできます。この機能の有無は、`/sys/module/page_alloc/parameters/shuffle` ファイルで確認できます。

5.3.3 キャッシュ・メモリー・モードの NUMA ノードのエミュレーション

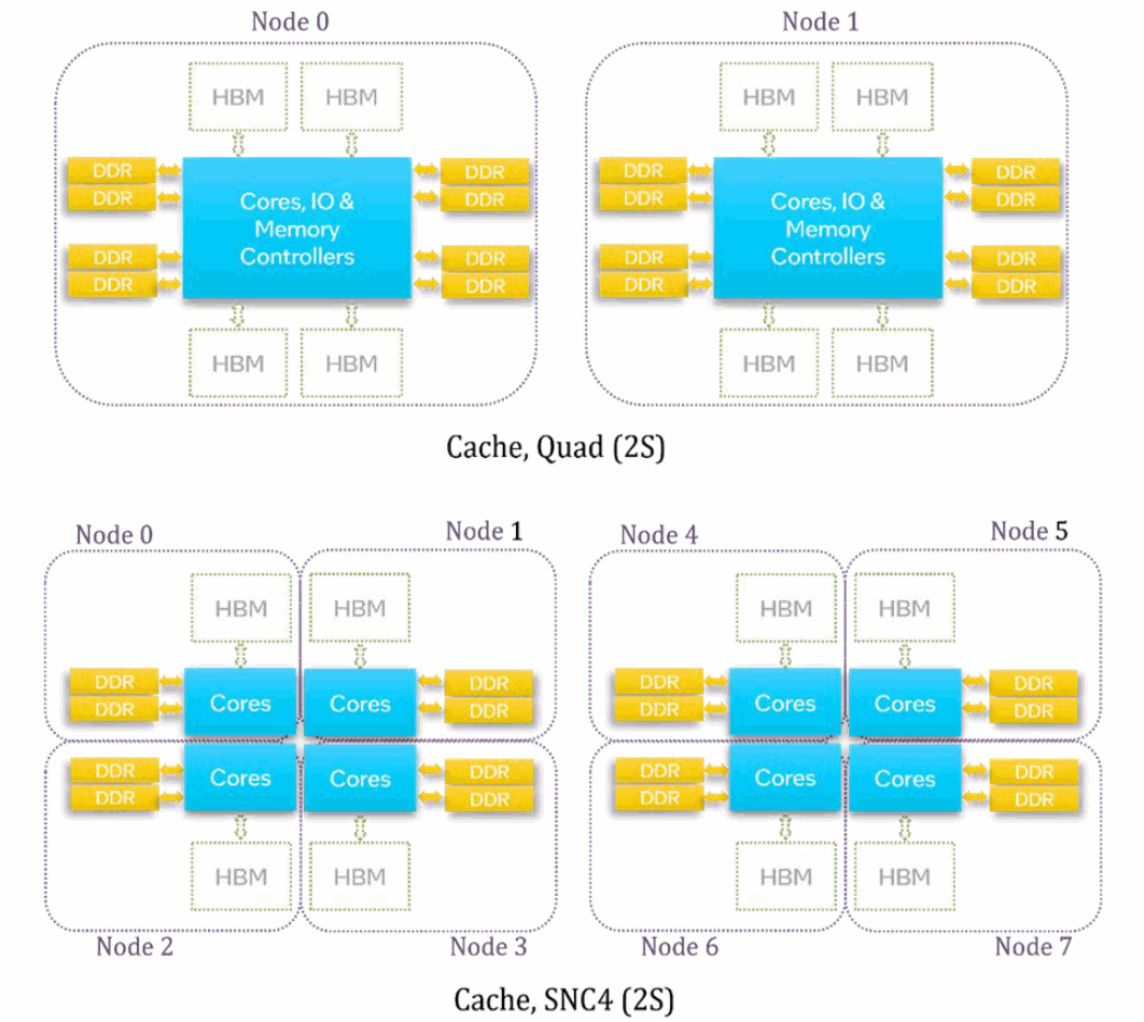

図 16: キャッシュモードの NUMA ノード構成

図 16 は、キャッシュモードの 2 ソケットのシステムにおける、クアドラントと SNC4 の NUMA ノード構成を要約したものです。クワドラント・モードでは少なくとも 2 つの NUMA ノード (ソケットごとに 1 つのノード) が作成され、SNC4 モードでは少なくとも 8 つの NUMA ノード (ソケットごとに 4 つ) が作成されます。各 NUMA ノードにはコアとメモリーの両方が含まれています。

`numactl -H` を使用して、NUMA ノードの構成と、各 NUMA ノード上の合計/空きメモリー量を確認します。

6. メモリーモードとクラスターモードの使用

このセクションでは、エンドユーザーがメモリーモードとクラスターモードを使用する方法について説明します。

6.1 HBM 専用メモリーモード

HBM 専用モードを使用する場合、ソースコードやコマンドライン構文を変更する必要はありません。OS とアプリケーションはどちらも、使用可能な唯一のオプションである HBM メモリーを使用します。ただし、アプリケーションを利用可能なメモリーに適合させるため、セクション 5.1 で説明した OS 設定手順に加えて、以下の手順が必要になる場合があります。

- OpenMP* スレッド数と MPI ランクのバランスをとります。OpenMP* スレッドはメモリーを共有するため、より多くの OpenMP* スレッドを利用すると、全体のメモリー・フットプリントを削減できます。
- アプリケーションが HBM 容量に収まらない場合、OpenMP* スタックサイズと MPI 通信バッファーサイズを適切なサイズに変更します。
- セクション 4.1 で説明したように、各実行前にファイルシステムのキャッシュをフラッシュし、メモリーを圧縮します。
- `/dev/shm` (`tmpfs`) は利用可能なメモリーを減少させるので、ファイルの保存には使用しないでください。以前のジョブのファイルが存在する場合は、`/dev/shm` にあるファイルを消去します。
- NUMA ミスがないことを確認します (実行の前後に `numastat` を実行します)。
- 上記の手順を実行してもアプリケーションが HBM 容量に収まらない場合は、ノード数を増やすことを検討してください。

6.2 フラット・メモリー・モード

フラットモードでは、前述のように、DDR と HBM が個別のアドレス空間 (NUMA ノード) としてユーザーに公開されます。図 17 は、SNC4 クラスターモードのフラット・メモリーモード・システムの `numactl -H` の出力例です。

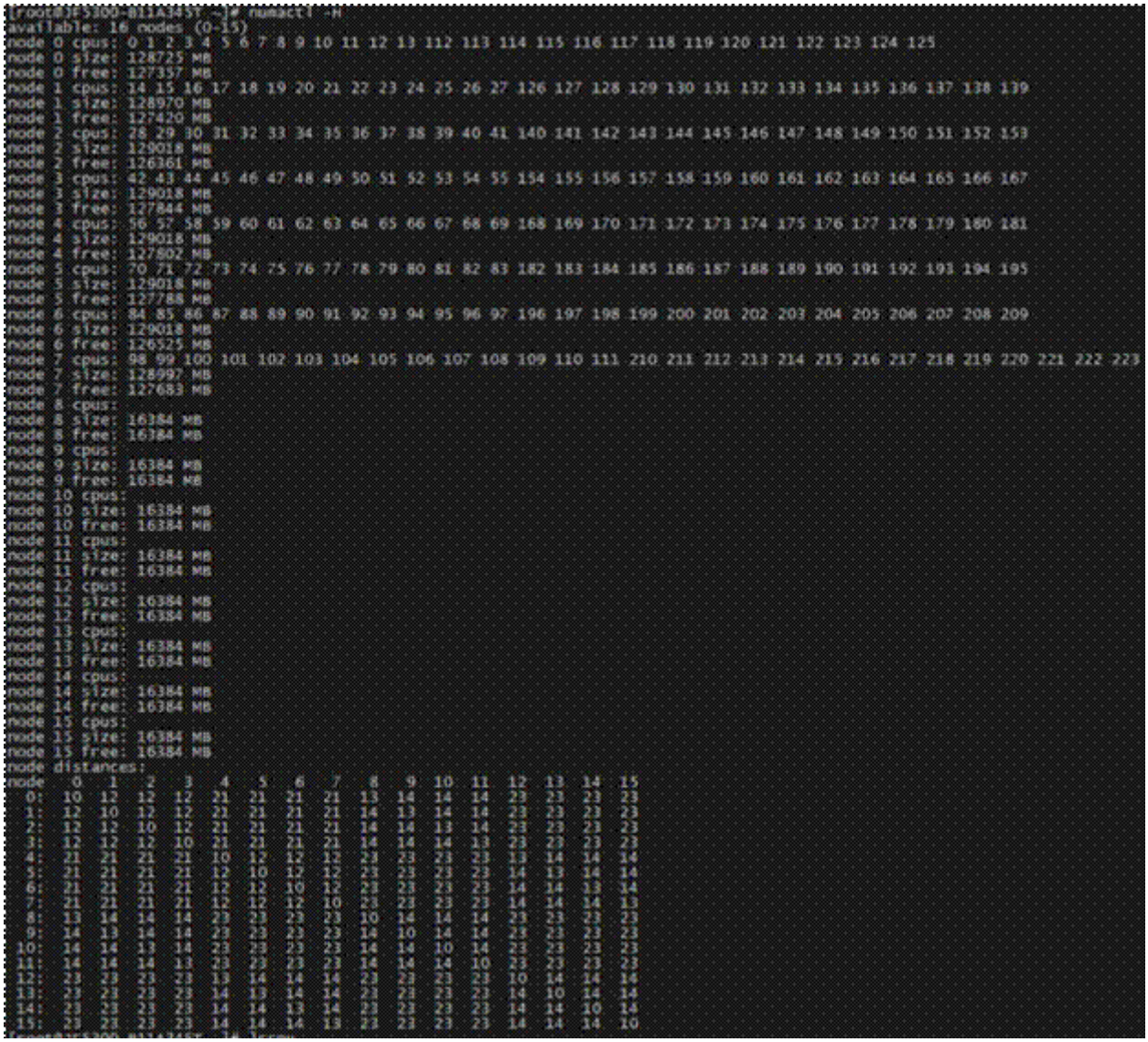

図 17: `numactl -H` の出力例

`numactl -H` の出力例に示されているように、ソケットごとに 2 種類のノードがあります。

- CPU が接続された DDR ノード (ノード 0 ~ 7)
- CPU が接続されていない HBM ノード (ノード 8 ~ 15)

アプリケーションが CPU コアで起動されると、「ノード距離」によって決定された、アプリケーションに最も近い NUMA ノードにメモリーが割り当てられます。例えば、図 17 のノード 0 の CPU の場合、最も近いメモリー (距離 10) はノード 0 に接続された DDR です。別の種類のメモリーを使用するには、次の 4 つの方法で libuma の機能を使用する必要があります。

- `numactl` ユーティリティー
- インテル® MPI ライブラリー
- OpenMP* ライブラリー
- `libnuma` API (英語)
- `memkind` ライブラリー (英語)

最初の 2 つの方法は、アプリケーション全体 (プログラムコード、静的データ、ヒープ、スタック) を HBM に配置し、最後の 2 つの方法は、動的に割り当てられた (ヒープに割り当てられた) 個々のデータ構造を HBM に配置します。

6.2.1 `numactl` を使用したアプリケーション全体の HBM 配置

標準 Linux* ユーティリティーの `numactl` を使用して、アプリケーションのメモリーを NUMA ノードに配置できます。以下のポリシーを考慮する必要があります。

- `membind` (`numactl --membind hbm_node1, hbm_node2,..... /a.out`): アプリケーションのすべてのメモリーが強制的に指定されたノードに割り当てられます。アプリケーションが指定されたノードの容量を超えると、アプリケーションは終了します。そのため、ユーザーはアプリケーションが **利用可能な** HBM 容量を超えないことを保証する必要があります。利用可能な HBM 容量は最大 HBM 容量よりも小さい場合があります。
- `preferred` (`numactl --preferred hbm_node ./a.out`): アプリケーションは、最初に指定された優先ノードにメモリーを割り当てます。優先ノードが一杯になったら、残りはデフォルトノードに割り当てられます。フラット・メモリー・モードでは、デフォルトノードは常に DDR ノードです。指定できる優先ノードは 1 つだけであることに注意してください。Linux* はファーストタッチ・ポリシーを採用しているため、アプリケーションはページを優先ノードに配置するため、ページを割り当ててタッチ (初期化など) する必要があります。例えば、クワドラント・クラスター・モードの 2 ソケットのシステムでは、ユーザーは MPI コロン構文を使用して `mpiexec` コマンドで異なるランクに異なる HBM ノードを指定できます。

```
mpiexec -n 1 numactl ---preferred hbm_node1 ./a.out : -n 1 numactl --
preferred hbm_node2 ./a.out
```

- `preferred-many` (`numactl --preferred-many hbm_node1, hbm_node2,..... /a.out`): `--preferred` オプションと似ていますが、複数の優先ノードを指定できます。このオプションは SNC4 とマルチソケットで特に有用ですが、Linux* カーネル 5.15 以上と `numactl` 2.0.15 以上が必要です。
- `interleaved` (`numactl --interleave hbm_node, DDR_node`): 任意の 2 つの NUMA ノード間でメモリー・インターリーブを可能にし、メモリー・フットプリントが HBM サイズの約 2 倍である場合に特に有用です。DDR と HBM の間でインターリーブする場合、期待できる最大帯域幅は DDR の帯域幅の 2 倍になります。

最良のアプローチは、HBM 容量内に収まる場合は、`membind` ポリシーを使ってアプリケーション全体を HBM 内に配置することです。そうでない場合は、`preferred` ポリシーや `interleaved` ポリシーを使用する前に、ノード数を増やすことを検討します。

6.2.1.1 SNC4 での特別な考慮事項

SNC4 で `numactl` を使用して複数の NUMA ノードに対応する場合、`membind` または `preferred` のどちらを使用するかに応じて、特別な注意を払う必要があります。

- membind: SNC4 モードで MPI アプリケーションを実行する場合、ユーザーは numactl の引数としてすべての HBM ノードを指定でき (例えば、mpiexec -np 8 numactl -m 4-7 ./a.out)、HBM メモリーは各ランク (プロセス) に最も近いノードから割り当てられます。
- preferred または interleaved: フラットモードで SNC4 を使用する場合、前述の preferred または interleaved のいずれかの方法でアプリケーションの一部を HBM に配置するには、MPI のコロン構文を使用する必要があります。なぜなら、preferred は (-- preferred-many が利用可能な場合を除き) 1 つの NUMA しか受け付けず、interleaved は対応する HBM と DDR ノードで行う必要があるからです。例えば、SNC4 のシングルソケットで 56 の MPI ランクを実行する場合に preferred を使うには、次のようにします。

```
mpirun -n 14 numactl -N 0 -p 4 ./a.out : -n 14 numactl -N 1 -p 5 ./a.out : -n 14
numactl -N 2 -p 6 ./a.out : -n 14 numactl -N 3 -p 7 ./a.out
```

同様に、同じシステムでインターリーブするには、次のようにします。

```
mpirun -n 14 numactl -N 0 -i 0,4 ./a.out : -n 14 numactl -N 1 -i 1,5 ./a.out :
-n 14 numactl -N 2 -i 2,6 ./a.out : -n 14 numactl -N 3 -i 3,7 ./a.out
```

6.2.2 インテル® MPI を使用したアプリケーション全体の HBM 配置

MPI アプリケーションでは、numactl の代わりに、I_MPI_HBW_POLICY 環境変数を使用して MPI ランクの HBM を割り当てることができます。この環境変数の詳細は、I_MPI_HBW_POLICY のリファレンス・ページを参照してください。

```
mpirun -genv I_MPI_HBW_POLICY hbw_bind -n 2 ./a.out
mpirun -genv I_MPI_HBW_POLICY hbw_preferred -n 2 ./a.out
mpirun -genv I_MPI_HBW_POLICY hbw_interleave -n 2 ./a.out
```

I_MPI_HBW_POLICY 環境変数は、MPI 自身によって割り当てられるメモリー (MPI バッファーなど) に対する割り当てポリシーも受け付けます。例えば、以下の例では、hbw_bind ポリシーをユーザー割り当てと MPI ライブラリー割り当ての両方に使用しています。

```
mpirun -genv I_MPI_HBW_POLICY hbw_bind,hbw_bind -n 2 ./a.out
```

6.2.3 個々のデータ構造の HBM 配置 (フラットモード)

より細かく制御するため、以下の方法で、動的に割り当てられた個々のデータ構造を HBM に配置することが可能です。これらの方法は、ユーザーが細かい制御を必要とし、`numactl` や MPI 環境変数を使用してもアプリケーション・メモリー全体を HBM に配置できない場合に使用します (例えば、アプリケーションが HBM の総容量を超えている場合など)。

これらの方法はソースコードの修正を必要とします。HBM に配置できるのは、動的に割り当てられたデータ構造 (つまり、ヒープ上に割り当てられたもの) のみです。スタック、静的データ、およびコードは、これらの方法で HBM に配置することはできません。

6.2.3.1 OpenMP* を使用した HBM 配置

これは、インテル® コンパイラー・クラシック (最近のバージョンすべて) およびインテル® oneAPI コンパイラー (バージョン 2021.3 以降) で利用できます。コンパイラーの OpenMP* プラグマとディレクティブは、`libnuma` へのインターフェイスとして `memkind` ライブラリー (英語) に依存しています。

この OpenMP* 機能は、gcc バージョン 11 以降でも使用できます。

C/C++

```
      #include <omp.h>
      float *x = (float *)omp_aligned_alloc(64, N*sizeof(float), omp_high_bw_mem_alloc);
      omp_free(x, omp_high_bw_mem_alloc);
```

`membind` が動作するように、フォールバックを `null_fb` または `abort_fb` に設定します。

```
      omp_alloctrait_t traits[2] = { {omp_atk_alignment, 64}, {omp_atk_fallback,
omp_atv_null_fb} };
      omp_allocator_handle_t my_high_bw_mem_alloc = omp_init_allocator(omp_high_bw_mem_space,
2, traits);
      float *x = (float *)omp_alloc(N*sizeof(float), my_high_bw_mem_alloc);
      omp_free(x, my_high_bw_mem_alloc);
      omp_destroy_allocator(my_high_bw_mem_alloc);
```

Fortran

```
       real, allocatable ::x(:)
       !dir$ omp allocate(x) allocator(omp_high_bw_mem_alloc) align(64)
       allocate(x(N))
```

6.2.3.2 memkind ライブラリーの hbwmalloc を使用した HBM 配置

memkind ライブラリー (英語) の hbwmalloc API (英語) を使用して個々のデータ構造を HBM に配置できます。

-lmemkind を使用してリンクします。

```
#include <hbwmalloc.h>
float* x = (float *)hbwmalloc(N * sizeof(float));
hbw_free(x);
```

hbw_posix_mem_align を使用することもできます。

```
#include <hbwmalloc.h>
float* x; hbw_posix_memalign((void**) &x, 64, N * sizeof(float));
hbw_free(x);
```

allocator を使用することもできます。

```
#include <hbw_allocator.h>
std::vector<float, hbw::allocator<float>> x;
```

Fortran の 2 つの例

```
!dir$ attributes memkind:hbw :: x
real, allocatable :: x(:)
allocate(x(N))


real, allocatable :: x(:)
!dir$ memkind : hbw, align:64
allocate(x(N))
```

6.3 キャッシュ・メモリー・モード

キャッシュモードを使用する場合、ソースコードやコマンドライン構文を変更する必要はありません。HBM キャッシュはソフトウェアに対して透過的であるため、アプリケーションには DDR メモリー空間しか見えません。したがって、ユーザーは DDR のみのマシンを使用しているかのようにアプリケーションを実行できます。

キャッシュモードでのみ存在する `/sys/devices/system/node/node0/memory_side_cache` ディレクトリーにより、システムがキャッシュ・メモリー・モードであることを確認できます。

最高のパフォーマンスを達成するため、HBM は DDR のキャッシュとして機能し、この HBM キャッシュがダイレクトマップされることに注意してください。その結果、HBM キャッシュで競合ミスが発生することがあります。競合ミスは、2 つの DDR アドレスが HBM キャッシュ内の同じ場所 (セット) にマップされた場合に発生します。HBM はダイレクトマップのメモリー・サイド・キャッシュであるため、ある時点でキャッシュできるのは、これらのアドレスのうち 1 つだけです。これは、頻繁なキャッシュミス (キャッシュに必要な行がない) につながる可能性があります。HBM キャッシュミスが発生するたびに、HBM アクセス (キャッシュラインが HBM キャッシュ内にあるかどうかを決定するため) と DDR アクセスが必要になるため、メモリー・レイテンシーが増加し、有効帯域幅が減少します。

競合ミスを減らすには、アプリケーションのワーキングセットが HBM キャッシュ内に収まるようにする必要があります。アプリケーションのワーキングセットが HBM キャッシュ内に収まらない場合、HBM と DDR の両方にアクセスすることでレイテンシーが増加し、総帯域幅が DDR の帯域幅を下回る可能性があります。そのため、フラットモードで DDR を直接使用するか、ノード数を増やしてノードあたりのワーキングセット・サイズを縮小することを検討してください。

6.2.3 Fake NUMA の使用

HBM 容量はソケットあたり 64GB なので、この容量に収まるアプリケーションであれば競合ミスは発生しないはずです。しかし実際には、物理メモリーの断片化により、そうはなりません。オペレーティング・システムは、物理アドレス範囲のどこからでも物理メモリーを割り当てることができます。物理メモリーは常にアドレス 0 から連続的に割り当てられるわけではなく、頻繁に割り当てと割り当て解除が繰り返されると、割り当てられたメモリーは連続したものではなくなります。これはメモリーの断片化と呼ばれます。物理メモリーが断片化されると、アプリケーションの総メモリー・フットプリントが HBM サイズより小さくても、HBM ダイレクト・マップ・キャッシュ内でメモリーアドレスが競合してしまう (つまり、アドレスがキャッシュ内の同じ場所にマップされてしまう) 可能性があります。

メモリー・フットプリントが 64GB 未満のアプリケーションでは、セクション 5.3.1 で説明したように、Fake NUMA と呼ばれる Linux* カーネル機能を使用することで、物理メモリーの断片化によって発生する競合を回避できます。Fake NUMA を使用すると、物理メモリーアドレス空間を連続した 64GB 領域 (Fake NUMA ノード) に分割できます。64GB NUMA ノード内のアドレスは、競合がないことが保証されています (つまり、HBM キャッシュ内の同じ場所にマップすることはできません)。そのため、アプリケーションは 1 つの Fake NUMA ノード内で実行できる場合、競合ミスを回避できます。

システムが Fake NUMA で構成されている場合、HBM サイズ内に収まるフットプリントを持つアプリケーションでは、競合ミスを回避するため追加ステップは必要ありません。アプリケーションが起動すると、ほかの Fake NUMA ノードに割り当てる前に、Fake NUMA ノード 0 から自動的にメモリー割り当てが開始されます。`numactl` を使用することで、ユーザーは任意の Fake NUMA ノードにアプリケーションを配置することも可能です。アプリケーションが 64GB メモリーのほぼすべてを使用するまれなケースでは、Fake NUMA ノード 0 ではなく、例えば Fake NUMA ノード 1 に配置することで、パフォーマンスがわずかに向上する場合があります。OS は各ソケット上の 1 番最初の Fake NUMA ノードのメモリーの一部を確保するため、Fake NUMA ノード 0 は通常、ほかの Fake NUMA ノードよりもメモリーの空き容量がやや少なくなります。`numactl -H` ですべての Fake NUMA ノードを確認できます。

SNC4 は Fake NUMA ノードの数を増やしますが、デフォルトの動作では、HBM 容量内に収まるアプリケーションは競合のない配置が保証されているため、ユーザーは Fake NUMA を使用するため特別な手順は必要ありません。Fake NUMA を使用するシステム上でアプリケーションをバインドする場合、NUMA ノードへのバインドではなく、**コア**へのバインド (OpenMP* や MPI 環境変数の使用など) を検討すると便利です。適切なコアにバインドされると、想定どおりに、コアに最も近い Fake NUMA ノードから自動的にメモリーが割り当てられます。

アプリケーションのメモリー・フットプリントが HBM 容量を超える場合、収まらなかったメモリーは次の Fake NUMA ノードから順次割り当てられます。これは、予測可能な動作です。

注: `hwloc` ライブラリーを使用する場合、Fake NUMA は不正なノードカウントにつながる可能性があります。回避策として、環境設定 `HWLOC_DEBUG_ALLOW_OVERLAPPING_NODE_CPUSETS=1` を使用してください。2023.1 より前のバージョンのインテル® MPI では、`I_MPI_HYDRA_TOPOLIB=ipl` を使用して `hwloc` をバイパスしない限り、この環境設定が必要です。

7. アプリケーション構成

このセクションでは、インテル® Xeon® CPU マックス・シリーズの一般的なベンチマークとツールの設定方法について説明します。

7.1 ソフトウェア環境

インテル® oneAPI ベース・ツールキット (ベースキット) やインテル® oneAPI HPC ツールキット (HPC キット) などのインテル® oneAPI ツールキットは、インテル® Xeon® CPU マックス・シリーズをサポートするコンパイラー、プロファイラー (インテル® VTune™ プロファイラー、インテル® Advisor など)、ライブラリー (インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL)、インテル® MPI ライブラリーなど) を提供します。

7.2 スモークテスト

以下のベンチマークは、システムの適切なパフォーマンスを検証するスモークテストとして使用できます。これらのベンチマークは、パフォーマンスを検証するため、システムの起動直後に実行すべきです。また、実行にそれほど時間がかからないため、各バッチジョブの前または後に実行して、各システムで期待されるパフォーマンスを検証することもできます。

最初の 2 つのテスト (インテル® メモリー・レイテンシー・チェッカー (インテル® MLC) と STREAM) はメモリーシステムのパフォーマンス (帯域幅とレイテンシー) を測定し、HPL は浮動小数点演算パフォーマンス (GFLOPs) を測定します。HPCG は主にメモリー帯域幅の影響を受けます。

7.2.1 インテル® メモリー・レイテンシー・チェッカー (インテル® MLC)

インテル® MLC (英語) は、単一システムの詳細なレイテンシーと帯域幅を測定します。以下のコマンドは、システムのテストに役立ちます。

ピーク帯域幅: `mlc --peak_injection_bandwidth -Z -X -t60`

帯域幅メトリックとレイテンシー: `mlc`

7.2.2 STREAM

STREAM ベンチマーク (英語) は、単一ノードのさまざまなルーチン (`Copy`、`Scale`、`Add`、`Triad` など) の帯域幅を測定します。

インテル® Xeon® CPU マックス・シリーズで最高のパフォーマンスを達成するには、以下のコマンドでソフトウェア・プリフェッチを有効にします。

```
icc -O3 -xCORE-AVX512 -qopt-zmm-usage=high -mcmodel=large -qopenmp
-qopt-streaming-stores=always -fno-builtin -qopt-prefetch-distance=128,16
-DSTREAM_ARRAY_SIZE=500000000 -DNTIMES=500 stream.c -o stream
```

そして、以下のコマンドで生成されたバイナリーを実行します。

```
KMP_HW_SUBSET=1t KMP_AFFINITY=balanced,granularity=core,verbose ./stream
```

注: キャッシュモードでは、メモリー・フットプリント (3 つのアレイの合計) が HBM キャッシュサイズを超える場合、最高のパフォーマンスを得るには、ソフトウェア・プリフェッチ・フラグ (`-qopt-prefetch-distance`) を省略する必要があります。

7.2.3 HPL

インテル® ディストリビューションの LINPACK ベンチマークは、High-Performance LINPACK (HPL) に変更と追加を加えたものです。インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) で、またはインテル® oneAPI ベース・ツールキット (ベースキット) の一部として、手順 (英語) とともに提供されています。倍精度のランダムな密連立一次方程式を因数分解して解を得るのにかかる時間を測定し、測定した時間をパフォーマンス・レート (GFLOPS) に変換し、演算結果が正確かどうかをテストします。

このベンチマークは、単一のノードまたはノードのクラスターで実行できます。ベンチマークの実行方法は、上記のリンク先を参照してください。例として、2 つのソケットを持つ単一ノードの SNC4 クラスターモードで、HBM 専用モードで実行するには、`runme_intel64_dynamic` ファイルの以下の定義を変更します。

```
export MPI_PROC_NUM=2
export MPI_PER_NODE=2
export NUMA_PER_MPI=4
```

その後、実行します。

```
./runme_intel64_dynamic -p 2 -q 1 -b 384 -n 120000
```

注: Fake 偽 NUMA を使用するキャッシュ・メモリー・モードでは、`NUMA_PER_MPI` はソケット上の Fake NUMA ノードの数と等しくなければなりません。

7.2.4 HPCG

インテル® CPU 向けに最適化された HPCG ベンチマーク (ソースコードおよびビルド済みバイナリー) は、最新のインテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) で、またはインテル® oneAPI ベース・ツールキット (ベースキット) の一部として、デベロッパー・ガイドとともに提供されています。

ソースコードからビルドするには、以下のコマンドを使用します。

```
# source C/C++ compiler, MPI compiler, and MKL library
#

export MKLROOT=/path/to/mkl

export LD_LIBRARY_PATH=${MKLROOT}/lib/intel64:${LD_LIBRARY_PATH}

# build binary for Intel AVX-512 -- bin/xhpcg_skx will be created
#

./configure IMPI_IOMP_SKX
make -j4 MKLROOT=${MKLROOT} MKL_INCLUDE=${MKLROOT}/include
```

インテル® Xeon® CPU マックス・シリーズ (各 56 コア) を搭載した 2 ソケットのシステムで HBM 専用メモリーモードと SNC4 クラスターモードで実行するには、以下のコマンドを使用します。

```
# Note: Select the best MPI x OMP decomposition for your case
#       Following assumes SNC4 (8 NUMA nodes on 2S),
#       and 14 cores (28 threads) on a NUMA node
#

export MKL_NUM_THREADS=28
export OMP_NUM_THREADS=28

nprocs_per_node=8
nnodes=1
nprocs=$((nnodes*nprocs_per_node))

problem_size=168         # options: 168, 192, 256
run_time_in_seconds=100  # 100 used as smoke test.
                         # 1800 is min for official HPCG submission

export I_MPI_SHM=spr-hbm
export I_MPI_FABRICS=shm:ofi
export I_MPI_PIN_DOMAIN=numa
export I_MPI_DEBUG=10    # print out mpi configuration mapping data

# for 1 hyper-thread, use 'compact,1,0' instead of 'compact'
#
export KMP_AFFINITY=granularity=fine,compact

echo " ===         nnodes: ${nnodes}"
echo " ===            ppn: ${nprocs_per_node}"
echo " ===         nprocs: ${nprocs}"
echo " === n_omp_per_proc: ${MKL_NUM_THREADS}"
echo " ===      prob_size: ${problem_size}"
echo " ===       run_time: ${run_time_in_seconds}"

# run bin/xhpcg_skx binary (either prebuilt or built by user)
#
mpiexec.hydra -genvall -n ${nprocs} -ppn ${nprocs_per_node}  bin/xhpcg_skx -
n$problem_size -t$run_time_in_seconds
```

7.3 アプリケーションのメモリー使用量の特定

最高のパフォーマンスを達成するには、通常、アプリケーションを HBM 容量内に収める必要があります。アプリケーションとワークロードのメモリー・フットプリントの特定には、以下のツールを利用できます。

- `top` と `htop` (指定された時点のシステムのメモリーの使用量/空き容量の合計を提供)
- `numastat -m` (各 NUMA ノードの合計メモリーを提供)
- `numastat -p <binary_name>` (指定された時点の指定されたプロセスのメモリー使用量を提供)
- `/usr/bin/time -v <app_cmd_line>` (**最大レジデント・セット・サイズ**を含む、アプリケーションに関するさまざまな統計情報を提供)。これは、bash のビルトイン `time` コマンドとは異なるため、パスを指定する必要があります。

アプリケーションのフットプリントが利用可能な HBM 容量を超える場合は、ノード数を増やすか、キャッシュモードを使用することを検討してください。

7.4 アプリケーションをメモリー帯域幅に最適化

インテル® Xeon® CPU マックス・シリーズは、以前のインテル® Xeon® プロセッサーと比較してメモリー帯域幅が大幅に向上しているため、以前のプロセッサーではメモリー帯域幅により制限されていたアプリケーションやルーチンであっても、インテル® Xeon® CPU マックス・シリーズではメモリー帯域幅が制限要因ではなくなる場合があります。

アプリケーションの実際の帯域幅使用量を確認する最善の方法は、インテル® VTune™ プロファイラーのメモリーアクセス解析 (英語) を使用することです。この解析により、アプリケーションの特定のフェーズまたはルーチンでメモリー帯域幅が十分に使用されていないことが示された場合は、次のオプションを使用してメモリー帯域幅を最適化できます。

計算の最適化: ルーチンはメモリー帯域幅に十分に使用するほど高速に計算を行っていない可能性があります。アドレス計算などの計算を最適化し、計算スループットを向上するためベクトル化を検討してください。インテル® Advisor を使用して、ベクトル化が可能な領域を特定し、ベクトル化を改善できます。

メモリー・レイテンシーの最適化: ルーチンで最終レベル (L3) キャッシュミスになるメモリーアクセスが頻発しているにもかかわらず、HBM 帯域幅を飽和させることができない場合、そのルーチンはメモリー・レイテンシー依存の可能性があります。ハードウェア・プリフェッチャーの有効性を低下させる不規則なアクセスパターン (間接アクセス、ギャザー、スキャッターなど) は、しばしば高いメモリー・レイテンシーにつながります。このようなアクセスパターンには、ソフトウェア・プリフェッチを使用することを検討してください。インテル® oneAPI コンパイラーは、`-qopt-prefetch` (英語) などのコンパイラー・フラグを提供し、ソースコード内で使用できる明示的な プリフェッチ・ディレクティブ (英語) と組込み関数 (C では `_mm_prefetch` (英語)、Fortran では `mm_prefetch` (英語)) をサポートしています。

インテル® Xeon® プロセッサーおよびインテル® Xeon® CPU マックス・シリーズの全体的なアーキテクチャーと最適化の詳細については、『Intel® Software Developer's Manuals and Software Optimization and Reference Manuals』(英語) と GitHub* (英語) を参照してください。

8. 関連情報

インテル® VTune™ プロファイラーを使用したインテル® Xeon® CPU マックス・シリーズとインテル® データセンター GPU マックス・シリーズ上でのワークロード最適化 (英語)