The Parallel Universe

intel Software

MPI (メッセージ・パッシング・インターフェイス):
分散メモリーシステムの標準規格

MPI-3 共有メモリー・プログラミング

メモリー消費の推定

インテル® MPI ライブラリーの条件付き再現性

Issue
21
2015

目次

編集者からのメッセージ

James Reinders インテル コーポレーションの並列プログラミング・エバンジェリスト兼ディレクター。新しい書籍『High Performance Programming Pearls』の共著者で、このほかに、『Multithreading for Visual Effects』(2014)、『Intel® Xeon Phi™ Coprocessor High Performance Programming』(2013、日本語:『インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング』)、『Structured Parallel Programming』(2012、日本語:『構造化並列プログラミング』)、『Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism』(2007、日本語:『インテル スレッディング・ビルディング・ブロック ―マルチコア時代の C++ 並列プログラミング』、中国語、韓国語の翻訳版があります)、『VTune™ Performance Analyzer Essentials』(2005) などの文献を発表しています。

MPI の誕生から 23 年

コンピューティング世界の英知を結集して産み出された MPI (メッセージ・パッシング・インターフェイス) 規格の誕生から早くも 23 年が経過しました。1992 年 4 月 29 日にバージニア州ウィリアムズバーグで開催された分散メモリー環境におけるメッセージパッシング規格の標準化についてのワークショップには、政府、学界、民間企業の 40 の組織から多くのプログラマーとコンピューター業界の専門家が集まり、現在メッセージ・パッシング・プログラムを記述する際に広く用いられている MPI 規格の基本的な機能について議論が行われました。

MPI は、実用性、可搬性、効率性、利便性を含む、多くの利点を兼ね備えた並列プログラミング環境を提供します。この号の注目記事「MPI-3 共有メモリー・プログラミング入門」で説明されているように、最新の MPI 規格、MPI-3 では柔軟性がさらに向上しました。記事では、共有メモリー・プログラミングを導入して一般的な MPI の送信 / 受信パターンを変更することにより、パフォーマンスを向上させます。

パフォーマンスの向上にはトレードオフがつきものです。この号の別の記事では、**並列プログラミング**のさまざまな局面で MPI 規格を活用する方法を説明します。例えば、数値コードの浮動小数点演算では、各反復で演算精度の差が生じることがあります。「**インテル® MPI ライブラリー**の条件付き再現性」では、簡単な例を使用して、特定の条件が満たされる場合に、再現性のある結果を生成するインテル® MPI ライブラリーの集合操作を紹介します。

ハイパフォーマンス・コンピューティング・アプリケーションはノードで利用可能なメモリーの大部分を使用する傾向があり、MPI ライブラリーのメモリー消費を推定することは困難です。「インテル® MPI のメモリー消費」では、インテル® MPI ライブラリーのメモリー消費を推定し、ライブラリーの設定を微調整してメモリー・フットプリントを減らす方法を説明します。

MPI 規格は、長期にわたり優れた基盤を提供しています。規格に沿ってコーディングすることで、信頼性の高いアプリケーションを作成できます。また、幅広く受け入れられているため、継続的な機能の追加や多くのツールのサポートを受けられます。この号で、長期にわたり発展を続ける MPI 規格の素晴らしさを示す 3 つの例を紹介できることを嬉しく思います。

James Reinders
2015 年 5 月

MPI-3 共有メモリー・プログラミング入門

MPI/OpenMP* プログラミングに代わるハイブリット並列プログラミング

Mikhail Brinskiy インテル コーポレーション ソフトウェア開発エンジニア
Mark Lubin インテル コーポレーション テクニカル・コンサルティング・エンジニア

概要

メッセージ・パッシング・インターフェイス (MPI) 標準は、分散メモリーシステムでよく使用されるプログラミング・インターフェイスです。メニーコアシステムにおけるハイブリッド並列プログラミングは、そのほとんどが MPI と OpenMP* を組み合わせたものです。この MPI/OpenMP* アプローチは、インテル® Xeon® プロセッサーやインテル® Xeon Phi™ コプロセッサーのようなマルチコア / メニーコア・アーキテクチャーを活用するため、ノード間の通信に MPI モデルを使用し、各計算ノードでスレッドのグループを実行します。

MPI-3 規格は、ハイブリッド・プログラミングに新しい MPI 共有メモリー (SHM) モデルを使用するアプローチをもたらします。[1] インテル® MPI ライブラリー 5.0.2[2] でサポートされた MPI SHM モデルは、既存の MPI コードをベースにわずかな変更を加えることで、共有メモリーノードのプロセス間の通信を高速化できます。[3]

この記事では、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー・ベースのマルチノード・システムで MPI SHM を利用する方法を紹介します。記事では、1D Ring アプリケーションにコードを追加し、一般的な MPI 送信 / 受信パターンを変更することで MPI SHM インターフェイスを利用する方法を説明します。ノード間とノード内の通信に必要な MPI 関数についても説明します。インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー・ベースのクラスターで異なる同期メカニズムを利用した場合の MPI SHM モデルのパフォーマンス評価には、変更を加えた MPPTEST ベンチマークを使用しました。MPI-3 規格に対応したインテル® MPI ライブラリー 5.0.2 で共有メモリーアプローチを利用すると、MPI 送信 / 受信モデルよりも大幅にパフォーマンスが向上します。

1D Ring: 標準 MPI ポイントツーポイントから MPI SHM へ

各 MPI ランクで MPI-1 非ブロッキング・メッセージを左右のランクと交換する、1D Ring の例[4] を変更して、MPI SHM API のセマンティクスを説明します。


```
MPI_Irecv (&buf[0],…, prev,…, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv (&buf[1],…, next,…, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend (&rank,…, prev,…, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend (&rank,…, next,…, MPI_COMM_WORLD, &reqs[3]);
      {do some work}
MPI_Waitall (4, reqs, stats);
```

1 1D Ring トポロジーにおける最も近いパートナーとのメッセージ交換に対応する MPI-1 コード

すべての MPI ランクが各ノードのメモリーを共有し、複数のマルチコアノードでコードを実行します。プログラマーは、**MPI_Comm_split_type** 関数を利用して、メモリー共有が可能な MPI ランクの最大グループ数を判断できます。この関数は、各ノードに出力コミュニケーター **shmcomm** に属するプロセスの「島」を作成する強力な機能を備えています。

```
MPI_Comm shmcomm;
MPI_Comm_split_type (MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED,0, MPI_INFO_NULL,
&shmcomm);
```

次に、コンパニオン集合関数が各ノードに MPI-3 リモート・メモリー・アクセス (RMA) 用のメモリーウィンドウを割り当てます。プロセスのメモリーのどの部分が他のプロセスで利用可能になるかを MPI が制限するため、これらはウィンドウと呼ばれます。

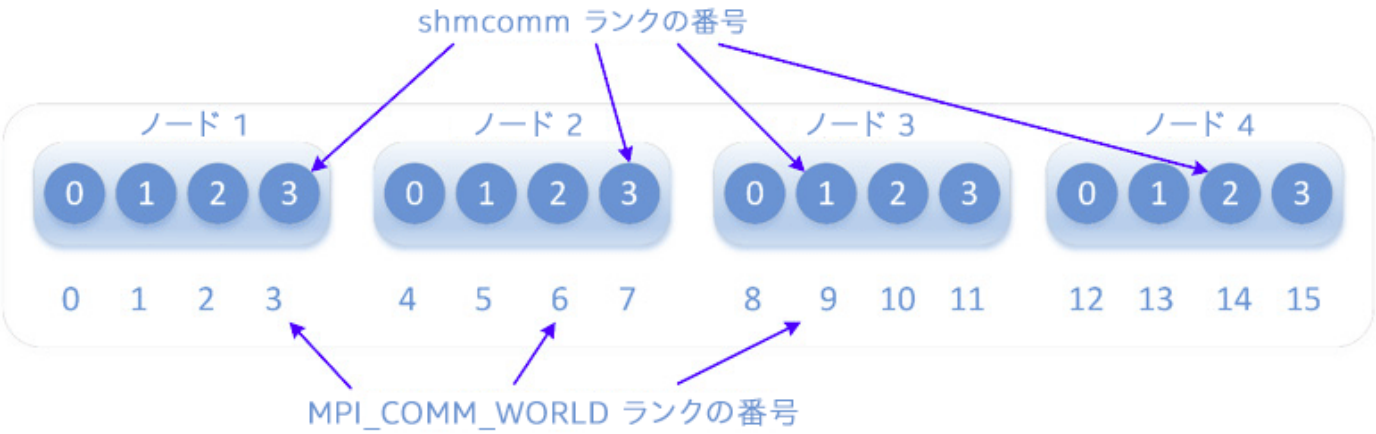
```
MPI_Win_allocate_shared (alloc_length, 1,info, shmcomm, &mem, &win);
```

ノード間で MPI 送信 / 受信ポイントツーポイント操作を行い、各ノード内で MPI SHM 関数を実行するには、同じノードに収まるランクと異なるノードに属するランクを識別するメカニズムが必要です。そこで、まず、グローバル・コミュニケーターと共有メモリー・コミュニケーター **shmcomm** により MPI グループを分けました。

```
MPI_Comm_group (MPI_COMM_WORLD, &world_group);
MPI_Comm_group (shmcomm, &shared_group);
```

次に、グローバルランクの番号を **shmcomm** ランクの番号にマップし、このマッピングを **partners_map** 配列に格納しました (**図 2**)。

```
MPI_Group_translate_ranks (world_group, n_partners, partners, shared_group,
partners_map);
```



2 グローバルランクと shmcomm ランクのマッピング。隣接するランクの一部が異なるノードに存在する場合、生成される partners_map 配列でそのマッピングは MPI_UNDEFINED (事前定義定数) になります。

MPI_Win_shared_query API は、条件付きテスト **partners_map[j]!= MPI_UNDEFINED** (現在のランクと通信パートナーが同じノードに存在してメモリーを共有する場合は true) を使用して共有メモリーセグメントのプロセスローカルなアドレスを見つけ出します。返されるメモリーポインター配列 **partners_ptrs** を利用して、共有メモリードメイン内のコストがかかる MPI 送信 / 受信関数を単純なロードとストアに置換します (**図 3**)。

```
for (j=0; j<n_partners; j++)
{
  if (partners_map[j] != MPI_UNDEFINED)
      MPI_Win_shared_query (win, partners_map[j],…, &partners_ptrs[j]);
}
```

3 MPI_Win_shared_query は異なるプロセスで同じ物理メモリーの異なるプロセスローカルのアドレスを返します。

ポイントツーポイント・メッセージ・パッシング・モデルと異なり、MPI SHM インターフェイスは、メモリーの一貫性を保証するため明示的な同期を仮定し、メモリーの変化がほかのプロセスに見えると仮定します。場合によっては、各開発者が理解して管理することで、より複雑なコードでもより高いパフォーマンスを実現できます。この記事では、これらの新しい同期のセマンティクスとパフォーマンスへの影響に焦点を当てます。

インテル® MPI ライブラリー 5.0.2 でサポートされた MPI SHM モデルは、既存の MPI コードをインクリメンタルに変更して、共有メモリーノードのプロセス間の通信を高速化できます。

最もパフォーマンス効率に優れた同期の 1 つとして、RMA ウィンドウを共有するすべてのプロセス用の **MPI_Win_lock_all** 関数と **MPI_Win_unlock_all** 関数のペアで定義される、いわゆるパッシブターゲット MPI RMA 同期があります。[5] ここでの用語「ロック」には、mutex におけるロックのように共有メモリー・プログラマーにとって一般的な意味は含まれていません。リモートメモリー操作が可能な場合、**MPI_Win_lock_all** と **MPI_Win_unlock_all** のペアは、RMA アクセスエポックと呼ばれる時間区間を示します。このケースでは、**MPI_Win_sync** 関数を使用してメモリー更新の完了を保証し、**MPI_Barrier** 関数を使用してノードのすべてのプロセスを同期します (**図 4**)。

```
// パッシブ RMA エポックを開始
MPI_Win_lock_all (MPI_MODE_NOCHECK, win);

// hello_world の情報をメモリー配列に書き込む
mem[0] = rank;
mem[1] = numtasks;
memcpy(mem+2, name, namelen);

MPI_Win_sync (win);     // メモリーフェンス - 同期ノード交換
MPI_Barrier (shmcomm); // タイムバリア
```

4 MPI SHM の更新にはパッシブ RMA 同期が必要です。パフォーマンス・アサーション MPI_MODE_NOCHECK はターゲットでエポックが直ちに開始可能なことを示します。プラットフォームによっては、読み取り側でメモリーの一貫性を保証するため MPI_Barrier の後に MPI_Win_sync がもう 1 つ必要になります。

特定のパートナー (隣接するプロセス) ごとに **MPI_Win_lock** を呼び出す手法は有効なアプローチであり、より優れたパフォーマンスが得られることもありますが、多くのコードを記述する必要があります。あるいは、メモリー更新を囲む一組の **MPI_Win_fence** 操作を利用して、アクティブターゲット MPI RMA 通信モードを使用する手法もあります。**MPI_Win_fence** 手法にはバリア同期がすでに含まれているため、エポックのロック / アンロックと比較するとそれほど冗長ではありませんが、検証ではより遅くなりました。

正しい同期を行うと、すべてのプロセスは、共有メモリーを使用して (パートナーが異なるノードに存在する場合は標準ポイントツーポイント通信を使用して) パートナーの情報を取得することができます (**図 5**)。

```
for (j=0; j<n_partners; j++){
  if (partners_map[j] != MPI_UNDEFINED)
  {
    i0 = partners_ptrs[j][0]; // MPI SHM からのロード処理!
    i1 = partners_ptrs[j][1];
    i2 = partners_ptrs[j]+2;

  } else { // ノード間非ブロッキング MPI
    MPI_Irecv (&rbuf[j],…, partners[j], 1 , MPI_COMM_WORLD, rq++);
    MPI_Isend (&rank,…, partners[j], 1 , MPI_COMM_WORLD, rq++);

  }
}
```

5 ノードで MPI SHM を使用し、ノード間の通信に標準非ブロッキング MPI 送信/受信を使用した Halo 交換

MPI SHM 通信が完了したら、**MPI_Win_unlock_all** を使用してアクセスエポックを閉じます。通常どおり、ノード間の通信は **MPI_Waitall** で同期します。

変更後のコードは**こちらから**ダウンロードできます。

MPI SHM を使用するように MPPTEST Halo 交換を変更

インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー・ベースのクラスターで、インテル® MPI ライブラリー 5.0.2 がサポートする MPI SHM のパフォーマンスを評価するため、プロトタイプ 1D Ring を使用して MPPTEST ベンチマーク[6] の Halo 交換アルゴリズムを変更しました。MPPTEST Halo テストには実際のアプリケーションに存在するような計算カーネルは含まれていませんが、異なる順の Halo 交換、メッセージサイズ、MPI 同期がパフォーマンスにどのように影響するか知ることができます。

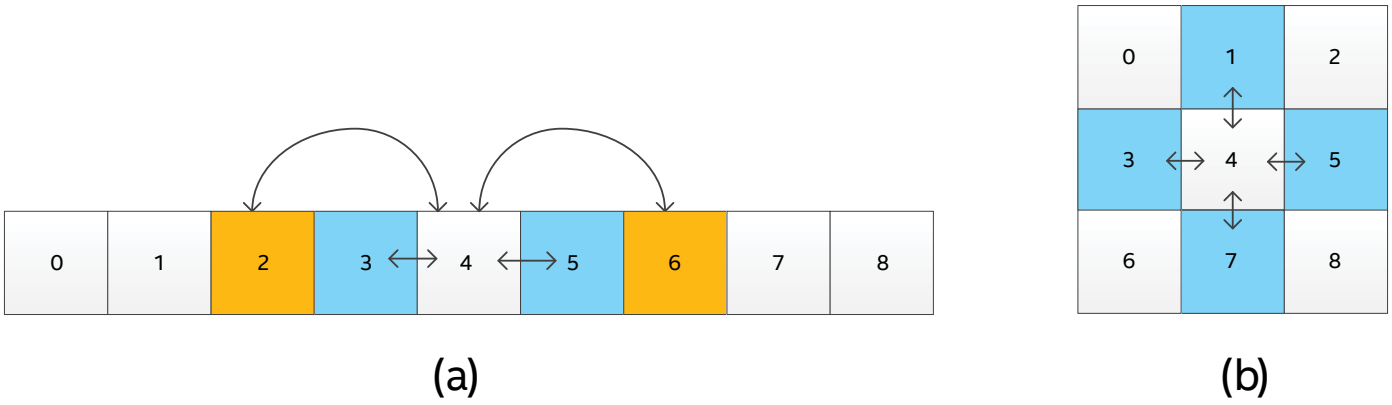
通常の送信 / 受信メモリーコピー操作、MPI スタックのレイテンシー、タグのマッチングを回避することにより、MPI SHM モデルは優れたパフォーマンスをもたらすことが知られています。[7] 従来の MPI メカニズムをメモリーコピー操作などの高速なノード内通信に置き換えると、残りの部分 (最後のセクションで簡単に説明する MPI SHM 同期) がノード内全体のパフォーマンス向上に役立ちます。

MPPTEST スイートに、設定パラメーターとして使用可能な 3 つの新しい Halo パターン (**mpi3shm_lockall**、**mpi3shm_lock**、**mpi3shm_fence**) を実装しました 。これらはすべて同じ MPI SHM 通信スキームを使用しますが、共有メモリー同期プリミティブは異なります。

- **mpi3shm_lockall。** アクセスエポックの開始と終了に **MPI_Win_lock_all** と **MPI_Win_unlock_all** を使用し、プロセスの同期 (メモリーおよび時間) に **MPI_Barrier** と **MPI_Win_sync** を使用します。
- **mpi3shm_lock。** **mp3shm_lockall** と同じですが、Halo 交換の各パートナーごとに **MPI_Win_lock** と **MPI_Win_unlock** を呼び出します。
- **mpi3shm_fence。** 一組の **MPI_Win_Fence** 呼び出しの間に実行される共有メモリーへのローカルストアはすべて一貫していることが保証されるため、ほかの同期プリミティブは必要ありません。

Halo 交換の異なるプロセストポロジーを調査するため、MPPTEST Halo ベンチマークに新しい設定パラメーター **-dimension** を追加しました。このパラメーターを指定すると、MPPTEST は 2 つのプロセス分解 (1D または 2D) のいずれかを使用します (デフォルトは 2D)。指定したパートナーの数が最も近いパートナーの交換に必要な数よりも多い場合、より深い密度の分解が使用されます。9 つのプロセスと 4 つのパートナー

に基づく例を、図 6 に示します。1D 分解の場合、ランク 4 のパートナーはランク 2、3、5、6 です。一方、2D 分解の場合はランク 1、3、5、7 になります。



6 プロセス分解: (a) 1D の 4 つのパートナー; (b) 2D の 4 つのパートナー

1D プロセス分解に MPI SHM を使用すると、ポイントツーポイント通信を使用した場合よりもパフォーマンスが (メッセージサイズに応じて) 最大 20 パーセント向上しました。

最後に、最も長い実行時間のプロセスに合わせて、レポートされるタイミングを変更しました。現在の MPPTEST アプローチでは、特にランク 0 のパートナーがほかのプロセスよりも少ない場合であってもランク 0 のタイミングが全体のタイミングとしてレポートされます。

環境と結果の評価

パフォーマンスの検証には、各ノードにデュアル インテル® Xeon® プロセッサー E5-2697、インテル® Xeon Phi™ コプロセッサー 7120P-C0 (1 枚)、そして同じソケットに接続された Mellanox* Connectx*-3 InfiniBand* アダプター (1 枚) が搭載された、インテル® Endeavor クラスターを使用しました。また、動作環境として、Red Hat* Enterprise Linux* 6.5、インテル® MPSS 3.3.30726、OFED* 1.5.4.1、インテル® MPI ライブラリー 5.0.2、インテル® C++ コンパイラー 15.0.1、および前のセクションで説明した変更を加えた MPPTEST ベンチマークを使用しました。

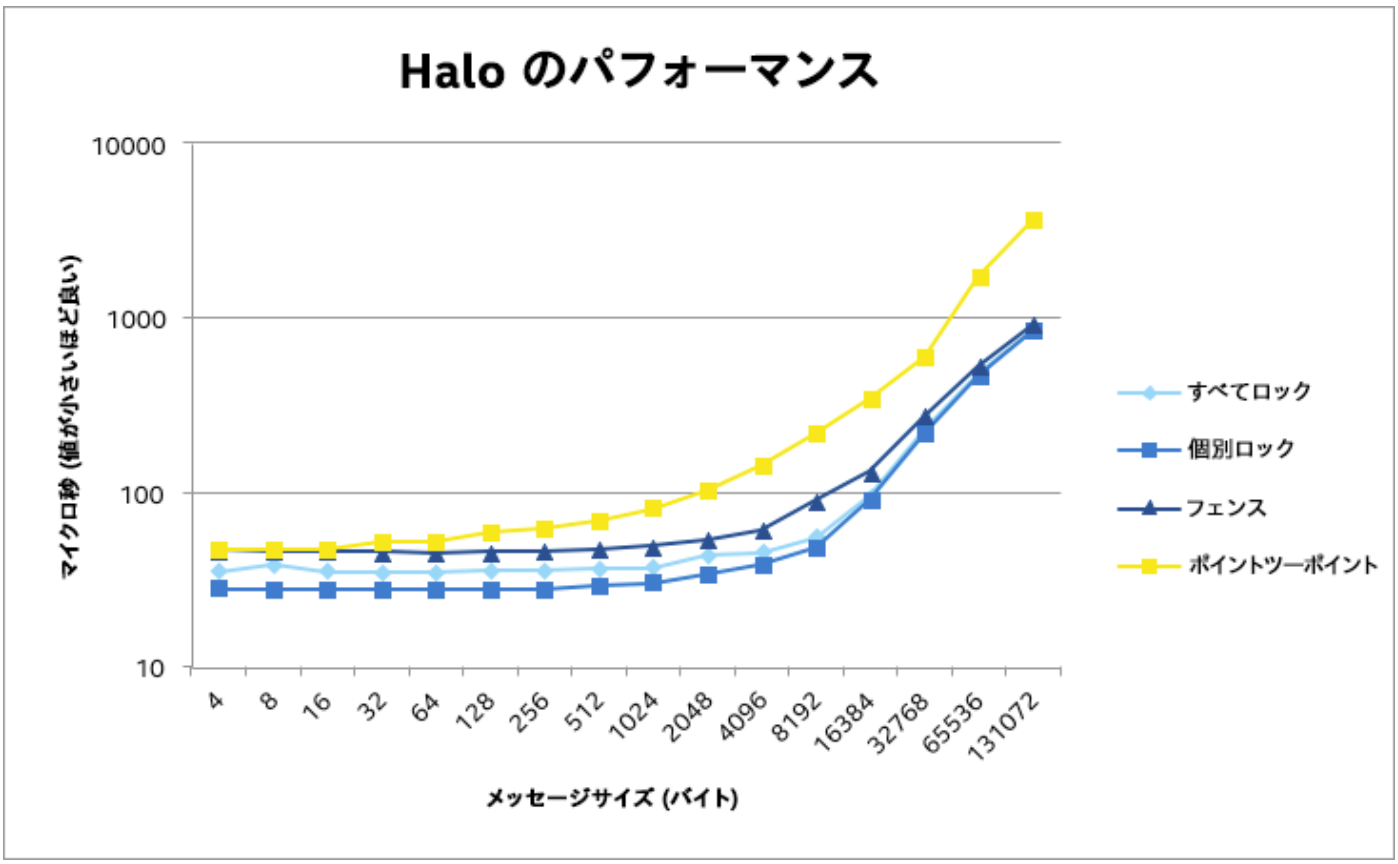
パフォーマンス・データの取得には、次のコマンドラインを使用しました。

```
mpirun -n 64 -machinefile hostfile ./mpptest -halo -waitall -logscale -n_avg 1000
-npartner 8 -dimension 2
```

-halo の後の引数は、ゴーストセル交換の通信パターンを指定します (**-waitall** はポイントツーポイント・メッセージの場合に使用されます。**-logscale** は 4 バイトから 128 KB の範囲の 2 の累乗のメッセージサ

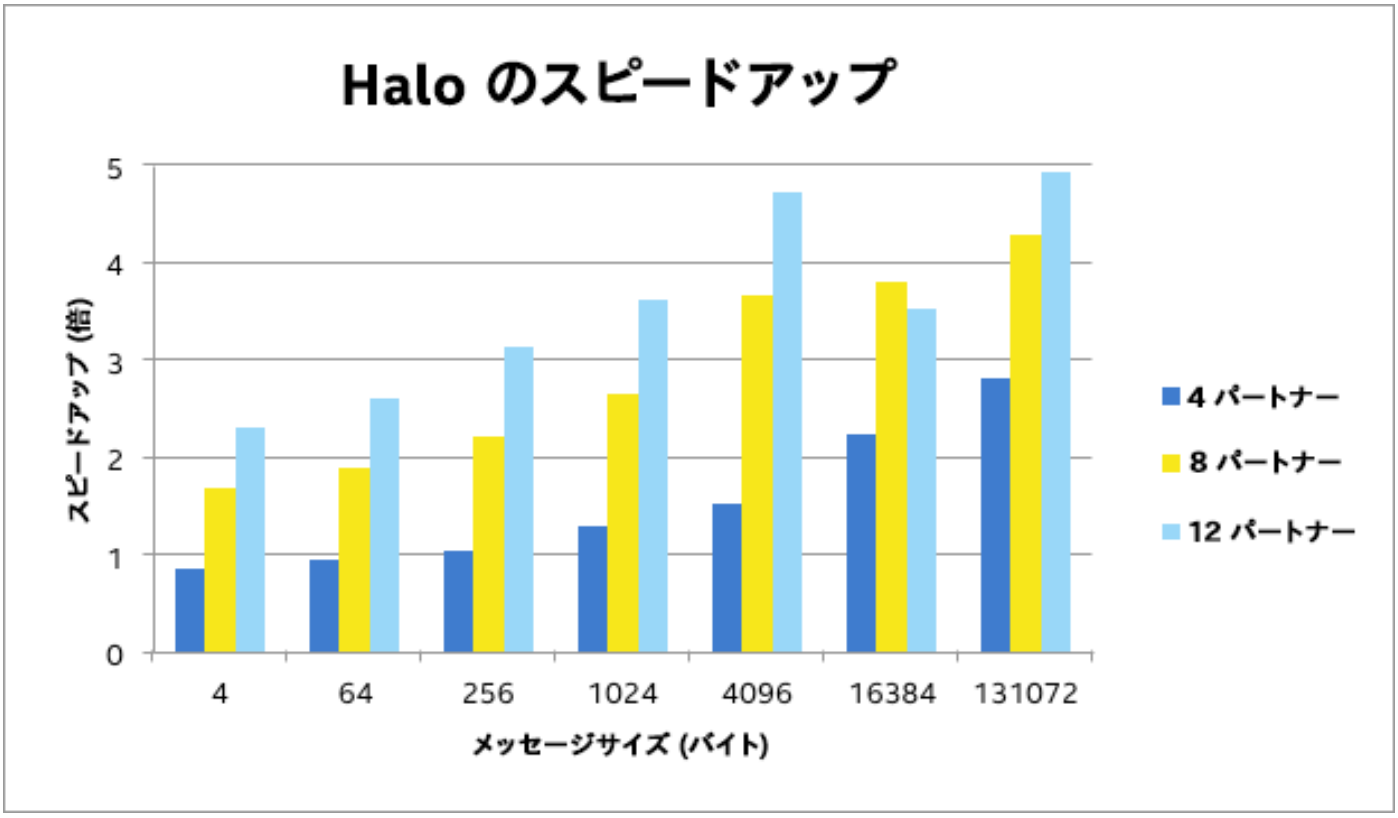
イズのテストを実行することを示します。**-n_avg** は使用する反復数を指定します。**-npartner** はプロセスごとのパートナーの数を決定します)。前述のように、**-waitall** の代わりに、変更後のベンチマークに対応する 3 つの新しいパラメーター (**-mpi3shm_lock**、**-mpi3shm_lockall**、**-mpi3shm_fence**) を実装しました。**-dimension** パラメーターはオプションです (デフォルトは 2D)。

図 7 は、1 枚のコプロセッサー、32 プロセス、8 パートナーの場合の結果を示しています。MPI SHM 機能は、同期の種類にかかわらず、通常のポイントツーポイント・パターンよりも大幅にパフォーマンスが優れています (y 軸は対数目盛であることに注意)。しかし、更新 (つまり、MPPTEST では反復) の量が相対的に小さい場合、ロックにより生じる同期のオーバーヘッドが大きくなることに注意すべきです。テストごとに 1 回ロックを行っているため、全体時間に対する割合は反復数に反比例します。個別にロックを使用するほうがプロセスすべてをロックするよりもパフォーマンスが向上することも分かります。特に、データを交換するノードのパートナーの数がウィンドウにバインドされたプロセスの数よりも大幅に少ない場合 (つまり、**MPI_Win_lock_all/MPI_Win_unlock_all** を呼び出すと、隣接パートナーのみではなくすべてのプロセスとの不必要な通信が行われる場合)、個別にロックを使用することが重要になります。また、**MPI_Win_fence** を使用した場合、同期プリミティブの中で最低の結果になることが分かります。



7 1 枚のコプロセッサー、32 プロセス、8 パートナーでの Halo パターンの比較

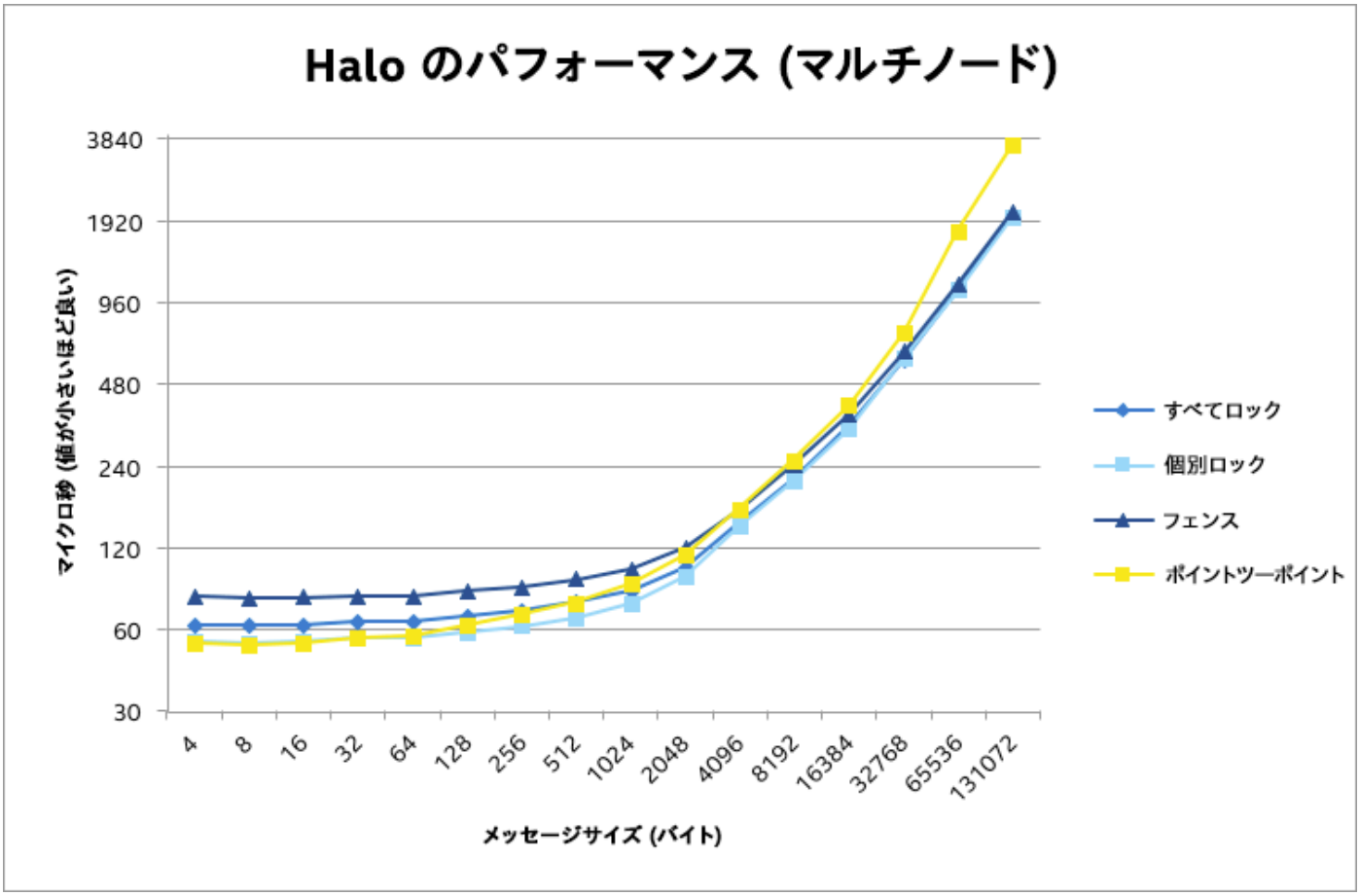
次に、Halo 交換のパートナーの数が全体的なパフォーマンスに与える影響を調べました。図 8 は、一般的な **MPI_Isend/MPI_Irecv** アプローチと比較した MPI SHM (ロック同期) のスピードアップを示しています。プロセスのパートナー数の増加に伴って、このアプローチのパフォーマンスが向上することが分かります。単純なメモリーコピーのパフォーマンス上の利点は交換とともに高まる一方、MPI SHM 同期の相対的なコストはパートナー数にかかわらず同じであることから、この結果は予想されたものでした。プロセスあたり 12 パートナーでは、小さなメッセージサイズで最大 2.6 倍、大きなメッセージサイズで最大 4.9 倍になりました。



8 ポイントツーポイント・ベースのアプローチと比較した場合の MPI SHM アプローチのスピードアップ (1 コプロセッサー、32 プロセスで測定)

異なるノードに接続された 2 枚のインテル® Xeon Phi™ コプロセッサーで測定を繰り返しました。プロセス数は 64 (コプロセッサーあたり 32 プロセス) でした。**図 9** の結果は、単一ノードの場合よりもスピードアップが低くなっています。これは、一部の交換はネットワーク経由で行われ、ノード内の通信コストは全体的なコストの一部にすぎないためです。メッセージサイズが非常に小さい場合 (標準ポイントツーポイント手法が適切) を除いて、ほとんどのメッセージサイズで個別のロックベースの共有メモリーアプローチが優れていることが

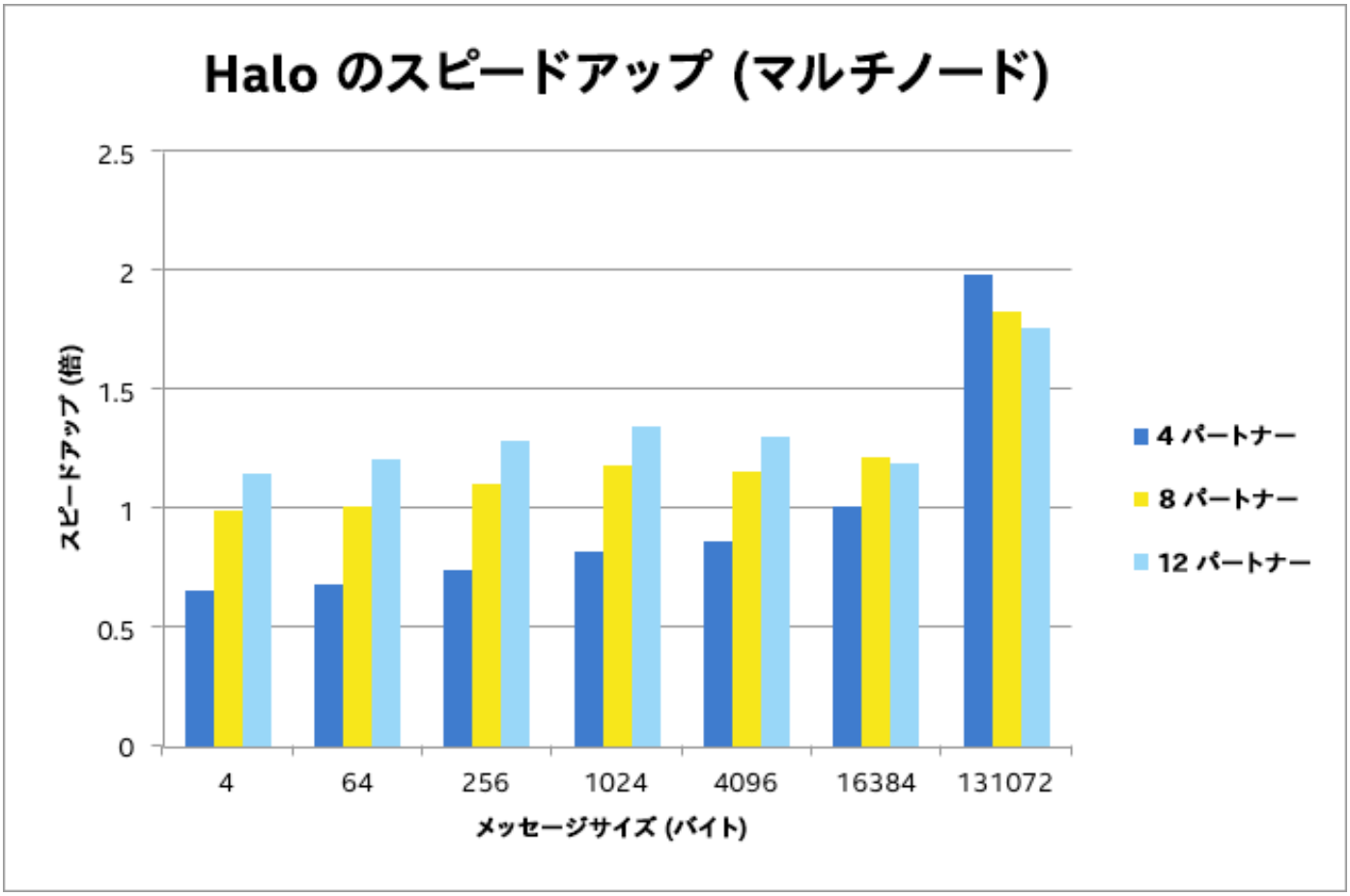
分かります。ここまでの測定は、デフォルトの 2D プロセストポロジーで行われました。1D プロセストポロジーを使用した個別のロックベースの MPI SHM アプローチは、小さなメッセージサイズでもほかのアプローチよりパフォーマンスが優れています。また、4KB 以上のメッセージサイズでは、すべての共有メモリーパターンはポイントツーポイント・ベースのパターンよりもパフォーマンスが優れています。



9 2 枚のインテル® Xeon Phi™ コプロセッサー、64 プロセス (カードあたり 32)、8 パートナーでの Halo パターンの比較

図 10 は、異なるパートナー数におけるロックベースのアプローチとポイントツーポイント・アプローチの比較を示しています。プロセスあたり 4 パートナーでは、ロックベースのアプローチは中規模以上のサイズでのみ利点があります。しかし、1 ノードの場合と同様に、パートナー数の増加に伴いパフォーマンスが向上します。プロセスあたり 8 パートナーおよび 12 パートナーでは、小さなメッセージサイズで最大 1.2 倍、大きなメッセージサイズで最大 1.8 倍パフォーマンスが向上しています。

インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーの両方を使用した 4 ノードと 8 ノードの予備調査では、同様の結果が示されていました。より多くのノード数での調査、およびハイブリッド MPI/OpenMP* コードの比較は今後の課題です。



10 ポイントツーポイント・ベースのアプローチと比較した場合の MPI SHM アプローチのスピードアップ
(2 枚のコプロセッサー、64 プロセスで測定)

まとめ

この記事では、MPI-3 規格に追加された共有メモリー機能について説明しました。この機能を使用するにはアプリケーションを変更する必要があるため、単純な 1D Ring サンプルを用いて変更方法を紹介し、複数ノードでの実行用に拡張しました。変更後の MPPTEST ベンチマークで、1 枚のインテル® Xeon Phi™ コプロセッサーでのパフォーマンスは、標準ポイントツーポイント・アプローチの最大 4.7 倍になりました。さらに、同じアプローチがマルチノードの Halo 交換にも有効であることを示しました。2 枚のインテル® Xeon Phi™ コプロセッサーでのパフォーマンスは最大 1.8 倍になりました。

我々の調査では、ゴーストセル交換ベースのアプリケーション (特に Halo 交換パートナーの数が多い場合) でも MPI SHM を使用するメリットがあることが分かっています。

謝辞

マルチノード MPI SHM コードに MPI グループを適用する方法について協力してくれた Charles Archer、多くの有用な議論を交わしてくれた Jim Dinan、そしてこの記事の草稿をレビューしてくれた Robert Reed と Steve Healey に感謝します。

参考文献

1. T. Hoefler et al., “Leveraging MPI’s One-Sided Communication Interface for Shared-Memory Programming: Recent Advances in the Message Passing Interface,” Proceedings of the 19th European MPI Users’ Group Meeting (EuroMPI 2012), Vienna, Austria, Vol. 7490, Sept. 23–26, 2012.

2. T. Hoefler et al., “MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory,” Computing (2013), Vol. 95, No. 12, p. 1,121.

3. M. Brinskiy et al., **“Mastering Performance Challenges with the new MPI-3 Standard,”** Parallel Universe Magazine Issue 18 (日本語訳:「**新しい MPI-3 標準でパフォーマンスの課題に対処する**」)

4. B. Barney, **“Message Passing Interface Tutorial: Non-Blocking Message Passing Routines.”**

5. W. D. Gropp et al., “Using Advanced MPI. Modern features of the Message-Passing Interface,” MIT Press, November 2014.

6. W. D. Gropp and Rajeev Thakur, “Revealing the Performance of MPI RMA Implementations, PVM/MPI’07,” Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 272–280.

7. Message Passing Interface Forum, **MPI: A Message-Passing Interface Standard Version 3.0**, University of Tennessee (Knoxville), Sept. 21, 2012.

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
software.intel.com/en-us/articles/optimization-notice/#opt-jp
を参照してください。

インテル® MPI ライブラリーの条件付き再現性

Michael Steyer インテル コーポレーション開発製品部門ソフトウェア & サービスグループ テクニカル・コンサルティング・エンジニア

はじめに

ハイパフォーマンス・コンピューティング (HPC) ユーザーなら、コードを実行したときに浮動小数点演算でわずかに異なる結果が生成された経験があることでしょう。通常、これは問題とは見なされませんが、アプリケーションの性質上、反復により違いが大きくなることがあります。

この問題に取り組むため、インテル® コンパイラーには浮動小数点の精度を操作するいくつかのスイッチが用意されています。インテル® マス・カーネル・ライブラリー (インテル® MKL) の条件付き数値再現性 (CNR) 機能[1] は、再現可能な浮動小数点結果を得るための機能を提供します。また、インテルの OpenMP* ランタイムと **インテル® スレッディング・ビルディング・ブロック** (インテル® TBB) ランタイムでは、決定性のあるリダクション・アルゴリズムが利用できます。しかし、**インテル® MPI ライブラリー**の一部の集合操作では、結果が多少異なることがあります。この記事では、インテル® MPI ライブラリーの集合操作から条件付きで再現性のある結果を得るための手法を紹介します。

目的

64 の MPI ランクで **MPI_Reduce** 操作を呼び出して倍精度値を集計する例を見てみましょう。

図 1 は、**MPI_Reduce** 操作を呼び出す Fortran コードを示しています。ランク 16 (インデックス 15) が 1.0 を書き込み、ランク 17 (インデックス 16) が -1.0 を書き込むことを除いて、各 MPI ランクは非常に小さな数 (2^-60) を **local_value** 変数に書き込みます。次に、**MPI_Reduce** を使用して、異なるランクのすべての **local_value** フィールドを集計します。リダクション操作の後、ランク 0 は小数点以下 20 桁までの数を **global_sum** に書き込みます。

```
program rep
  use mpi
  implicit none
  integer :: n_ranks,rank,errc
  real*8 :: global_sum,local_value

  call MPI_Init(errc)
  call MPI_Comm_size(MPI_COMM_WORLD, n_ranks, errc)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, errc)

  local_value = 2.0 ** -60

  if(rank.eq.15) local_value= +1.0
  if(rank.eq.16) local_value= -1.0

  call MPI_Reduce(local_value,global_sum,1,MPI_DOUBLE_PRECISION, &
         MPI_SUM,0,MPI_COMM_WORLD, errc)

  if(rank.eq.0) write(*,'(f22.20)') global_sum

  call MPI_Finalize(errc)
end program rep
```

1 Fortran 90 累積の例

各システムで 32 のプロセッサー・コアを備えた 4 つのノードが利用可能であると仮定します。アプリケーションは 2 つのシステムでのみ実行できるため、2 つの異なる MPI ランクの分散スキームを考えます。

A) 4 つのノードで 64 ランク => ノードあたり 16 ランク
B) 2 つのノードで 64 ランク => ノードあたり 32 ランク

高度な最適化のため、インテル® MPI ライブラリーは分散および共有メモリーリソースを可能な限り効率的に活用しようとします。実行サイズ (MPI ランク数) と交換するメッセージサイズに応じて、ライブラリーは各集合操作で利用可能なアルゴリズムの中から適切なアルゴリズムを選択します。レデュース操作にトポロジーを意識したアルゴリズムを選択すると、ケース A と B で演算の順序が異なります。

クラスター・インターコネクトで負荷を減らすには、最初にローカルな (ノードごとの) 演算を集計した後、クラスター・ネットワーク経由でそれらの結果を 1 回のみ送信し、最終的な結果を集計します。

A) Reduce(Reduce(#1 – #16) + Reduce(#17 – #32) + Reduce(#33 – #48) + Reduce(#49 – #64))
B) Reduce(Reduce(#1 – #32) + Reduce(#33 – #64))

結合則「 (a + b) + c = a + (b + c)」は、正確な計算と事実上無制限の精度を仮定するため、制限付きの精度表現を使用する場合には適用されません。浮動小数点数は限られたビット数で値を表現して近似されるため、これらの値に対する演算では丸め誤差がよく発生します。浮動小数点演算のシーケンスでは、丸め誤差の合計は演算を実行する順序に依存します。[2]

ケース A と B で演算の順序が異なると、最後の Reduce でわずかに異なる値が生成されます。

インテル® MPI ライブラリーには、MPI ランク分散環境が実行ごとに異なる場合でも条件付きで再現可能な結果を得ることができるアルゴリズムが用意されています。

結果がわずかに異なっても、IEEE 754 浮動小数点規格によれば結果はまだ有効です。[3] 純粋な浮動小数点の観点から、ケース A と B におけるランクの分散を分類すると、実際の問題がより分かりやすくなります。

A) ((… + 2^-60 + (+1)) + ((-1) + 2^-60 + …) + …
B) ((… + 2^-60 + (+1) + (-1) + 2^-60 + …) + …

ケース A では、+1 と -1 を非常に小さな値 (2^-60) と集計しなければなりません。ケース B では、+1 と -1 が同じステップで計算されるため、これらの値は排除されます。

インテル® MPI ライブラリーのランタイム設定 (**表 1**) に応じて、**図 2** のように結果が出力されます。

```
$ cat ${machinefile_A}
ehk248:16
ehs146:16
ehs231:16
ehs145:16
$ cat ${machinefile_B}
ehk248:32
ehs146:32
ehs231:0
ehs145:0
$ mpiifort -fp-model strict -o ./rep.x ./rep.f90
$ export I_MPI_ADJUST_REDUCE=3
$ mpirun -n 64 -machinefile ${machinefile_A} ./rep.x
0.00000000000000000000
$ mpirun -n 64 -machinefile ${machinefile_B} ./rep.x
0.00000000000000004163
```

2 異なる浮動小数点結果

前処理

インテル® MPI ライブラリーの再現性に取り組む前に、アプリケーションのほかの部分が数値的に安定した結果を生成することを確認します。

例えば、MPI のハイブリッド・スレッディング拡張によく使用される OpenMP* 標準規格では、部分和の組み合わせ順を定義しません。そのため、OpenMP* のリダクション操作の結果は、ランタイム・パラメーターに応じて実行ごとに異なることがあります。インテルの OpenMP* ランタイムには、ランタイムの動作を制御する環境変数、**KMP_DETERMINISTIC_REDUCTION** が用意されています。[4] また、インテル® TBB ライブラリーは、**parallel_deterministic_reduce** 関数を使用した決定性のあるリダクションをサポートしています。[5]

インテル® コンパイラーとインテル® MKL の使用についての詳細は、「Using the Intel Math Kernel Library and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results」[6] (英語) を参照してください。

再現性

期待値を明示的に設定するには、「再現可能」と「繰り返し可能」を区別する必要があります。「再現可能」を使用する場合は、常に条件付き再現性を意味するものとします。

繰り返し可能	アプリケーションを全く同じ条件で起動した場合 (同じマシンと条件で実行を繰り返した場合)、一貫した結果が得られます。
再現可能 (条件付き)	含まれるランク数 (およびハイブリッド・アプリケーションの場合はスレッド数) は一定でなければなりませんが、ランクの分散が異なる場合でも一貫した結果が得られます。さらに、マイクロアーキテクチャーを含むランタイムが一貫している必要があります。[7]

インテル® MPI ライブラリーのすべての操作は繰り返し可能な結果を保証

インテル® MPI ライブラリーの操作の再現性は、次の条件の下で保証されます。

1. 集合リダクション操作の内部でトポロジーを意識したアルゴリズムを使用しない。
2. `MPI_Allreduce` 操作に Recursive Doubling アルゴリズムを使用しない。
3. `MPI_Reduce_scatter_block` (および MPI-3 非ブロッキング) 集合操作を使用しない。

再現性の **1 つ目の条件**は、`I_MPI_ADJUST_` 環境変数を使用して対応する集合リダクション操作アルゴリズムを明示的に設定することで満たされます。
詳細は、『インテル® MPI ライブラリー リファレンス・マニュアル』[8] の「集合操作の制御」を参照してください。マニュアルの情報には、どのアルゴリズムがトポロジーを意識していて回避すべきであるか明白に記述されています。

表 1 に、リダクションを使用する 5 つの集合操作と対応するインテル® MPI ライブラリー環境変数を示します。(上記の 1 つ目の条件を満たす) トポロジーを意識しないアルゴリズムを利用するには、この表にしたがって環境変数を設定します。

表 1

リダクションを使用する MPI 集合操作	インテル® MPI 集合操作制御用環境変数	トポロジーを意識しないアルゴリズム
MPI_Allreduce	I_MPI_ADJUST_ALLREDUCE	(1)[a], 2 , 3 , 5 , 7 , 8, 9[b]
MPI_Exscan	I_MPI_ADJUST_EXSCAN	1
MPI_Reduce_scatter	I_MPI_ADJUST_REDUCE_SCATTER	1 , 2 , 3 , 4
MPI_Reduce	I_MPI_ADJUST_REDUCE	1 , 2 , 5, 7[a]
MPI_Scan	I_MPI_ADJUST_SCAN	1

a `MPI_Allreduce` の最初のアルゴリズムはトポロジーを意識しませんが、条件付きで再現可能な結果が保証されないことに注意してください (詳細は 2 つ目の条件を参照)。

b Knomial アルゴリズム (IMPI ≥ 5.0.2) は、`I_MPI_ADJUST_<COLLECTIVE-OP-NAME>_KN_RADIX` 環境変数が一定または変更されない場合にのみ再現可能な結果が得られます。

選択されているアルゴリズムを確認するには、環境変数 **I_MPI_DEBUG=6** を設定して出力を調べます。集合操作のデフォルトのアルゴリズムは、実行のサイズ (ランク数) と転送するメッセージサイズに応じて異なります。**図 3** は、前に紹介した単純な MPI レデュース・アプリケーションで使用した集合操作のデバッグ出力を示しています。

```
...
[0] MPI startup(): Reduce_scatter: 4: 0-2147483647 & 257-512
[0] MPI startup(): Reduce_scatter: 4: 0-5 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 5: 5-307 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 1: 307-1963 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 3: 1963-2380781 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 4: 0-2147483647 & 513-2147483647
[0] MPI startup(): Reduce: 1: 0-2147483647 & 0-2147483647
[0] MPI startup(): Scan: 0: 0-2147483647 & 0-2147483647
[0] MPI startup(): Scatter: 1: 1-494 & 0-32
[0] MPI startup(): Scatter: 2: 495-546 & 0-32
[0] MPI startup(): Scatter: 1: 547-1117 & 0-32
[0] MPI startup(): Scatter: 3: 0-2147483647 & 0-32
[0] MPI startup(): Scatter: 1: 1-155 & 33-2147483647
...
```

3 選択した集合操作の例

例えば、**MPI_Reduce** 集合操作では、すべてのメッセージサイズ (0–2147483647) および MPI ランクの範囲 (0–2147483647) に対して、アルゴリズムがデフォルトで選択されています。このため、MPI リダクションで異なる結果を得るには、上記の例に異なるトポロジーを意識したアルゴリズム (3) を選択する (**I_MPI_ADJUST_REDUCE=3** 以外に設定する) 必要がありました。

2 つ目の条件は、**MPI_Allreduce** 操作に Recursive Doubling アルゴリズムを使用しない (**I_MPI_ADJUST_ALLREDUCE=1** 以外に設定する) ことで満たされます。MPI ランクの順序が一定であることは保証されていますが、各 MPI ランク内部のオペランドの順序は適用される最適化により異なります。

しかし、演算に可換法則「a + b = b + a」が当てはまる場合、Recursive Doubling アルゴリズムを使用して再現可能な結果を得ることもできます。

3 つ目の条件は、**MPI_Reduce_scatter_block** (および新しい MPI-3[9] 非ブロッキング集合操作) がトポロジーを意識したアルゴリズムを使用して実装されるため必要です。これらの集合操作は (バージョン 5.0.2 の時点では) インテル® MPI ライブラリーのユーザーが変更することはできません。特定の操作パラメーターに基づいてランタイムに決定されます。

図 4 は、この記事の「目的」セクションで使用した単純なリダクションの例で再現可能な結果を得るための方法を示しています。ここでは、インテル® MPI ライブラリー環境でトポロジーを意識しない集合操作アルゴリズムを適用します。

図 3 で示したように、最初のアルゴリズムはすでにデフォルトで選択されていました。この別のオプションは、**I_MPI_ADJUST_REDUCE** 環境変数を指定しないで、デフォルト設定をそのまま使用しています。

```
$ cat ${machinefile_A}
ehk248:16
ehs146:16
ehs231:16
ehs145:16
$ cat ${machinefile_B}
ehk248:32
ehs146:32
ehs231:0
ehs145:0
$ mpiifort -fp-model strict -o ./rep.x ./rep.f90
$ export I_MPI_ADJUST_REDUCE=1
$ mpirun -n 64 -machinefile ${machinefile_A} ./rep.x
0.00000000000000004163
$ mpirun -n 64 -machinefile ${machinefile_B} ./rep.x
0.00000000000000004163
```

4 再現可能な浮動小数点結果

ノードの MPI ランクの分散は変更されていますが、条件付き再現性の定義によりランタイム環境は同じでなければならないため、ほかのすべてのパラメーター (ランク数や使用するアーキテクチャーなど) は一定であることに注意してください。

インテル® Xeon Phi™ コプロセッサー

インテル® MPI ライブラリーの条件付き再現性において、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーで処理に違いはありません。同じ概念が両方に適用されるため、ユーザーは違いを意識することなくインテル® Xeon Phi™ コプロセッサーを HPC ソリューションに統合できます。

しかし、マイクロアーキテクチャー / 命令セットやハードウェアの丸めサポートの違いから、2 つのマイクロアーキテクチャー間で結果が異なる場合があることに注意してください。この記事の「再現性」セクションで定義したように、条件は同じでなければならないため、スレッド数と MPI ランクは一定でなければなりません。

まとめ

この記事では、異なる MPI 集合操作で決定性のあるリダクションを保証するアルゴリズムをインテル® MPI ライブラリーで使用するいくつかの方法を説明しました。

また、アプリケーションのソースコードを変更することなく、繰り返し可能から条件付きで再現可能な結果に変更する簡単な MPI レデュース操作の例を用いて、アルゴリズムの影響を説明しました。

インテル® MPI ライブラリーには、MPI ランク分散環境が実行ごとに異なる場合でも条件付きで再現可能な結果を得ることができるアルゴリズムが用意されています。条件付きで再現可能な結果の条件を満たすには、ほかのすべてのパラメーター (ランク数やマイクロアーキテクチャーなど) は実行ごとに等しくなければならないことを理解することが重要です。

参考文献および注記

1. T. Rosenquist, "Introduction to Conditional Numerical Reproducibility (CNR)," Intel Corporation, 2012.
2. D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," Association for Computing Machinery, Inc., 1991.
3. IEEE Standard for Binary Floating-Point Arithmetic, The Institute of Electrical and Electronics Engineers, Inc., 1985.
4. M.J. Corden and D. Kreitzer, "Consistency of Floating-Point Results using the Intel® Compiler," Intel Corporation, 2012. (日本語訳 :「インテル® コンパイラーの浮動小数点演算における結果の一貫性」)

BLOG HIGHLIGHTS

計算をインテル® プロセッサー・グラフィックスへオフロードするチューニングのヒント

ANOOP MADHUSOODHANAN PRABHA »

プログラマーがプロセッサー・グラフィックスのパフォーマンスを引き出すためにカーネルをチューニングする際のヒントを以下に示します。

- オフロードされる入れ子構造のループにはプロセッサー・グラフィックスで利用可能なすべてのハードウェア・スレッド数以上の反復を含めます。完全な入れ子構造の _Cilk_for 並列ループを使用すると並列ループの入れ子の次元を並列化できます。
- オフロードコードをベクトル化するにはプラグマやコードの再構成を利用します。
- __restrict と __assume_aligned キーワードもベクトル化に役立ちます。
- offload プラグマの pin 節を使用すると、GPU とのデータコピーが制限されます。
- スカラーのメモリーアクセスはベクトルのメモリーアクセスほど効率的ではありません。メモリーアクセスにインテル® Cilk™ Plus の配列表記を使用すると、計算のベクトル化に役立ちます。単一のメモリーアクセスで 128 バイトまで処理できます。4 バイト要素のギャザー/スキャッター操作は非常に効率的ですが、2 バイト要素では遅くなります。ギャザー/スキャッター操作は、非ユニットストライドの部分配列により生じることがあります。

この記事の続きはこちら (英語) でご覧になれます。 >

5. A. Katranov, "**Deterministic Reduction: A New Community Preview Feature in Intel® Threading Building Blocks**," 2012.

6. T. Rosenquist and S. Story, "**Using the Intel Math Kernel Library and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results**," Intel® Parallel Universe Magazine, 2012. (日本語訳 : Parallel Universe マガジン第 11 号の「**インテル® マス・カーネル・ライブラリー (インテル® MKL) とインテル® コンパイラーを使用し、実行するたびに数値再現性のある結果を取得**」)

7. ターゲット・アプリケーションがインテル® AVX のようなベクトル命令セット向けにコンパイルされた場合でも、Sandy Bridge (開発コード名) や Haswell (開発コード名) のような異なるマイクロアーキテクチャー上で実行したときに、利用可能なマイクロアーキテクチャーに基づく異なるベクトル命令セットが使用されることがあります。詳細は、「Consistency of Floating-Point Results using the Intel® Compiler」[3] を参照してください。

8. **Intel® MPI Library—Documentation**, Intel Corporation, 2015.

9. "**MPI: A Message-Passing Interface Standard—Version 3.0**," Message-Passing Interface Forum, 2012.

インテル® MPI のメモリー消費

Dmitry Durnov インテル コーポレーション開発製品部門ソフトウェア & サービスグループ シニア・ソフトウェア・エンジニア
Michael Steyer インテル コーポレーション開発製品部門ソフトウェア & サービスグループ テクニカル・コンサルティング・エンジニア

はじめに

ハイパフォーマンス・コンピューティング (HPC) アプリケーションはノードで利用可能なシステムメモリーの大部分を使用する傾向があるため、クラスター上の限られたメモリーリソースを注意して扱うことが大切です。最大量の専用メモリーをアプリケーションに提供するには、クラスター上の他のメモリー消費を考慮する必要があります。特に、使用するオペレーティング・システムとライブラリーについて理解することが重要です。メッセージ・パッシング・インターフェイス (MPI) ライブラリーのメモリー消費がジョブサイズと MPI ランク数により増加するのに伴って、メモリー・フットプリントの推定は複雑になります。

この記事では、異なるファブリックを使用した**インテル® MPI ライブラリー**のメモリー消費の推定に関する基本的な情報を提示します。(実際のメモリー消費はオペレーティング・システム環境に依存するため、ここではバイト単位の正確な予測モデルは提示しません。) また、インテル® MPI ライブラリーの設定を微調整してメモリー・フットプリントを減らす方法についても説明します。

ライブラリーのメモリー

多くの要因がメモリー・フットプリントに影響を与えるため、メモリー消費の解析は複雑な作業です。割り当てられる仮想メモリーのサイズを予測することは可能でも、使用される物理メモリーの量はライブラリーを利用しているアプリケーションの特性とオペレーティング・システムの構成に依存します。

インテル® MPI ライブラリーがメモリーを消費する部分は 2 つ (MPI プロセス数を含むスケールする部分と固定のメモリー・フットプリントを含む部分) に分けることができます。メモリーを最大限に消費する部分は、ジョブサイズによりメモリー消費をスケールする必要性から発生します。

BLOG HIGHLIGHTS

インテル® マス・カーネル・ライブラリーの検査 - 実行スパース BLAS ルーチン

ZHANG Z »

2015 年 4 月にリリースされたベータ版 インテル ® マス・カーネル・ライブラリー (インテル® MKL) 11.3 では、スパース BLAS (SpMV 2) の検査 - 実行のための API が追加されました。この API は、操作を 2 つのステップに分割します。最初の解析段階で、API は行列の疎性パターンを検査し、行列構造の変更を適用します。その後のルーチン呼び出しで、この情報はパフォーマンス向上のために再利用されます。

この検査 - 実行 API は、反復法スパースソルバーの主要なスパース BLAS 操作をサポートし、インテル® MKL に含まれている従来のスパース BLAS 実装で利用可能なすべての機能に対応しています。

- 疎行列 - ベクトルの乗算
- 疎行列 - 行列の乗算 (結果が疎または密)
- 三角方程式の解法
- 疎行列の加算

この記事の続きはこちら (英語) でご覧になれます。 >

ライブラリーのメモリー消費の予測がプラットフォーム (オペレーティング・システムとアーキテクチャー) に依存する理由はいくつかあります。1 つの要因は、オペレーティング・システムが採用するページサイズです。ページサイズは、異なる MPI プロセス間のメッセージの送受信に使用するバッファーのアライメントとサイズに影響を与えます。別の要因は、導入しているミドルウェアです。一部のミドルウェア・ライブラリーは非常に低水準で使用されるため、ジョブサイズにより増加するメモリー消費の一部となります。異なるバージョンのライブラリーでは、接続のバッファーサイズが異なることがあります。この違いも、インテル® MPI ライブラリーのメモリー・フットプリントが変動する原因です。

これらの困難なメモリー消費の予測に適切に取り組むため、この記事では最悪のシナリオである全対全 (All-to-All) 接続に注目します。このシナリオでは、各 MPI ランクはほかのすべての MPI ランクへ接続を行うため、全体の接続数は **n^2** になります。メッセージサイズは最大の内部 MPI バッファーとほぼ同じになるように設定され、より大きなメッセージ転送のメモリー・フットプリントはアプリケーションに反映されます。これにより、割り当てられたメモリーのほぼすべてが使用されます。仮想メモリーのバッファーが一杯になっていない場合、マシンの物理メモリーのサイズは同じである必要はありません。

この記事で示すグラフはすべて、最悪のケースにおける、ランクあたりのインテル® MPI ライブラリーのメモリー消費を想定しています。そのため、ノードごとのメモリー消費の量は、各ノードで使用されているランク数に依存します。各グラフには、Haswell EP (開発コード名) のようなノードあたり 28 MPI ランクのシステムの基準値として、ノードあたり 64GB のメモリー消費を示す点線が含まれています。メモリー消費の推定はインテル® MPI ライブラリー 5.0.3.048 を基にしています。

全対全接続パターンが必要ない実際のアプリケーションではインテル® MPI のメモリー・フットプリントははるかに小さくなるため、この記事で提示した推定はあくまでも最悪のケースを仮定したものと考えてください。ほとんどのアプリケーションでは、インテル® MPI ライブラリーの動的な接続確立ロジック (サポートしているほぼすべてのトランスポート手法に含まれます) は、ランクごとに必要な最小限の接続のみ保持します。

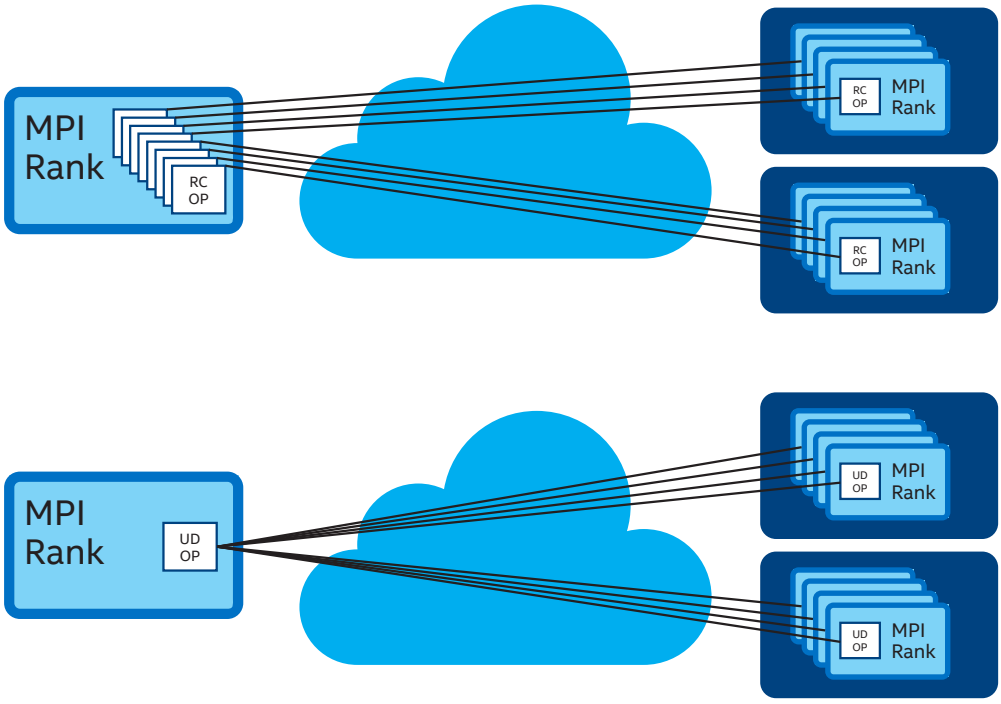
インテル® MPI のファブリックとメモリー

インテル® MPI ライブラリーは、次の低水準トランスポート・メカニズムをサポートしています。

- **SHM (共有メモリー)。**ノード間のメッセージ・トランスポートに使用します。メモリー消費 / レイテンシーのトレードオフとなるいくつかの設定を提供し、最新のプロセッサーのアーキテクチャー固有の最適化をサポートします。
- **DAPL (Direct Access Programming Library) (RC (Reliable Connection) および UD (User Datagram))。**uDAPL (User Direct Access Programming Library) に基づきます。uDAPL は、異なるベンダーのハードウェア利用に高いレベルの柔軟性を提供します。ベンダー固有およびテクノロジー固有の機能は uDAPL レベルでは意識する必要はありません。
- **OFA。**このトランスポート・レイヤーは、IB-verbs RC の直接使用に基づきます。
- **TMI (Tag Matching Interface)。**インテル® True Scale ファブリック・ハードウェアのメイン API である PSM (Performance Scaled Messaging) API に基づきます。
- **TCP。**TCP ソケットに基づき、IPoIB のようなソリューションにも適用できます。

また、**インテル® MPI ライブラリー**はこれらのファブリックの組み合わせをサポートしていて、ノード間通信とノード内通信のファブリックを分けるために使用できます。

しかし、インテル® MPI ライブラリーのメモリー消費状況について言えば、これらのファブリックの動作は異なります。各ランクでほかのランク (n) の QP (Queue Pair) バッファーとトランスポート・バッファーを保持する DAPL RC の特性により、DAPL RC のランクあたりのメモリー消費は MPI ランク数とともに直線的に増加します (n * 接続あたりのバッファー数)。これに対して、DAPL UD はすべてのランクで 1 つの QP とバッファーの 1 つの共通プールのみを利用するため、スケール時に優れています。**図 1** は、DAPL RC と DAPL UD の違いを示しています。

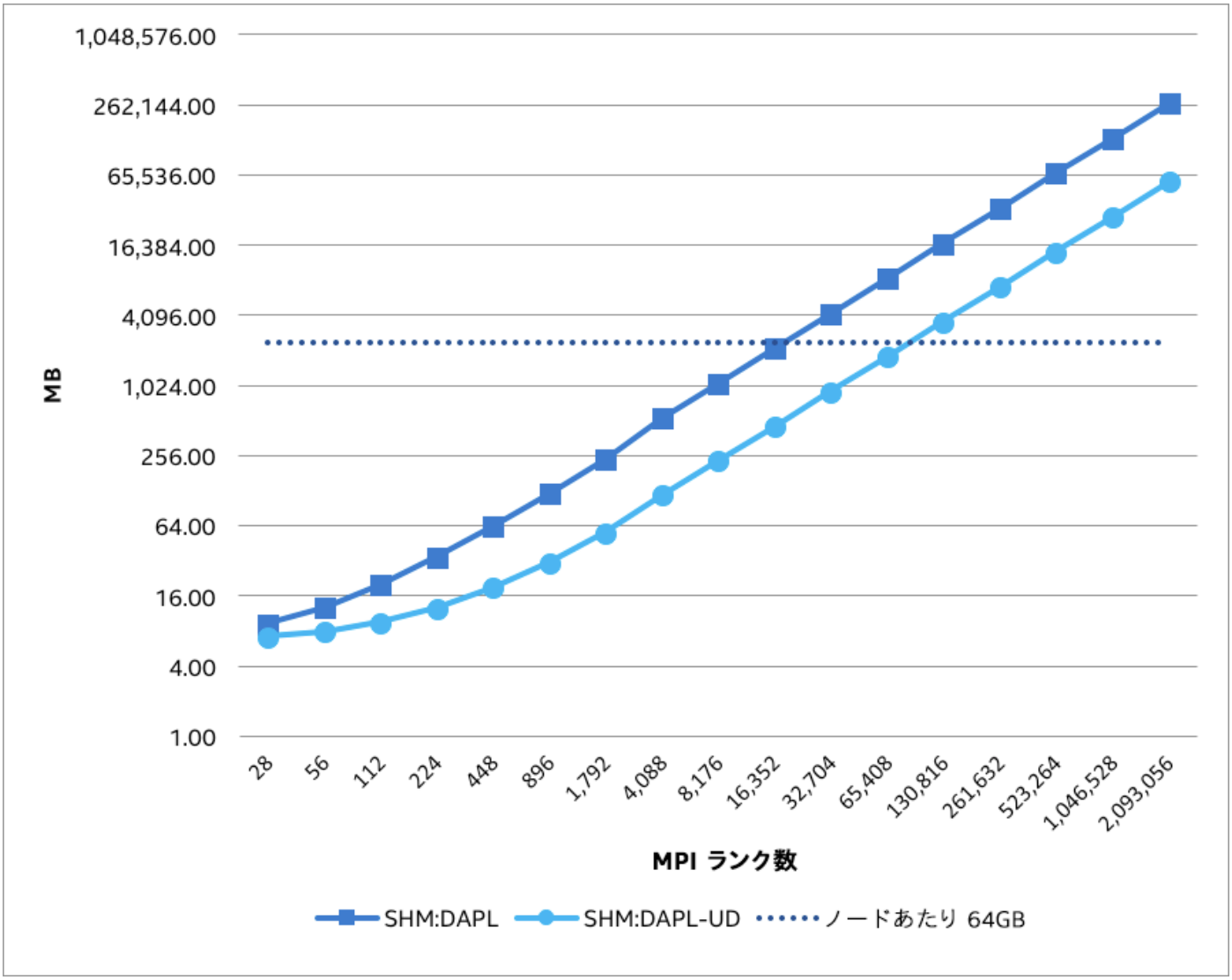


1 DAPL RC (上) と DAPL UD (下)

DAPL UD のコネクションレス通信の使用には、RDMA サポートが利用できないなどの短所もあります (DAPL UD + DAPL RC 混在モードでは利用できます)。各メッセージの区分化と再アセンブリーによる転送の遅延も短所になります。しかし、DAPL RC よりもメモリーが節約されることにより、ラージスケールの MPI ランクではパフォーマンスが向上します。

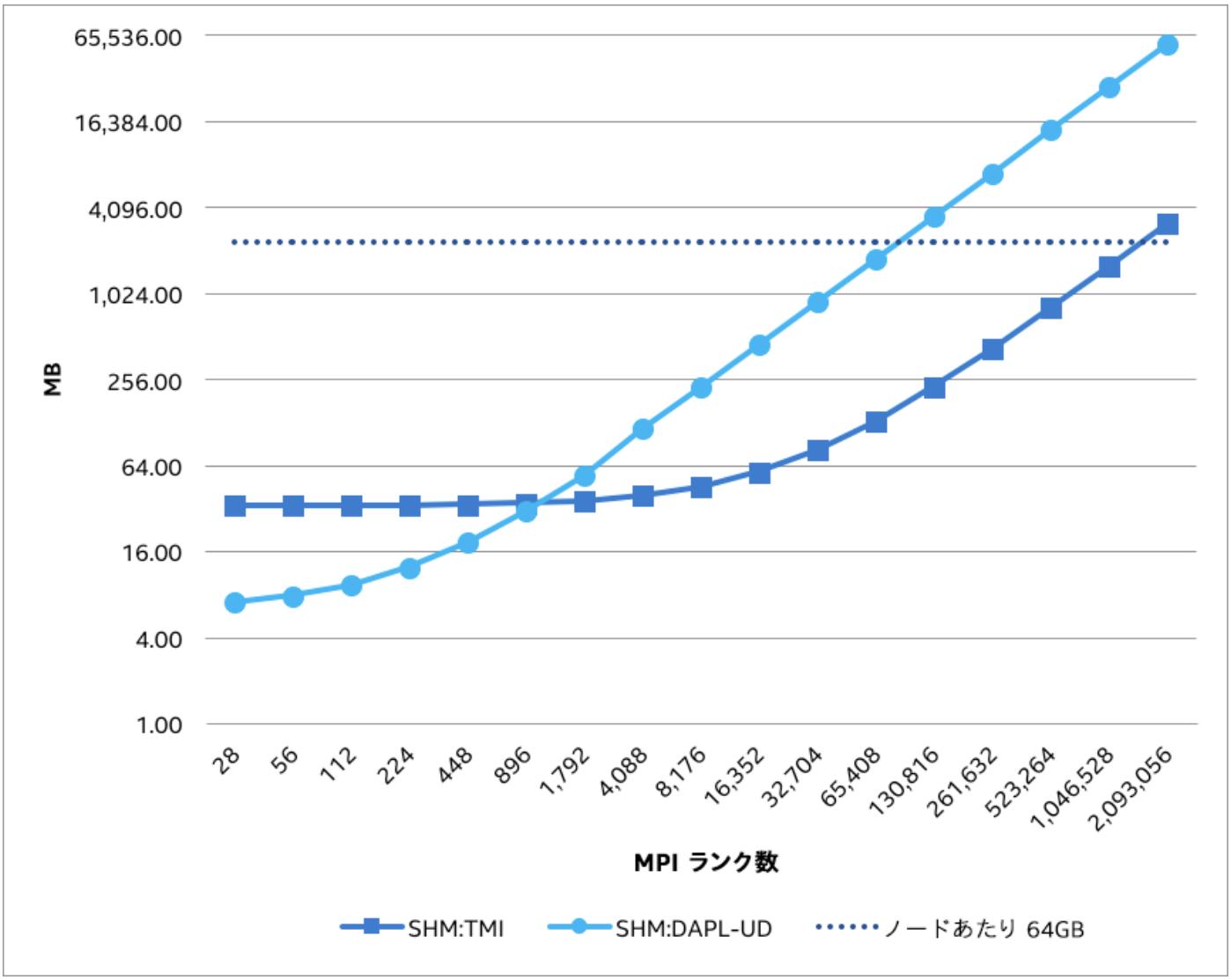
図 2 は、全対全接続シナリオにおけるインテル® MPI ライブラリーの DAPL RC と DAPL UD 間のメモリー消費の違いを示しています。前述したように、あくまでも最悪のケースを想定したものであるため、これらの数字はインテル® MPI ライブラリーの正確なメモリー消費モデルではなく、上限と見なすようにしてください。

検証では、最高のメモリー・フットプリントが得られるように、**I_MPI_ADJUST_ALLTOALL** 環境変数を「3」に設定して **MPI_Alltoall** 操作を行い、ペア交換アルゴリズムを利用しました。



2 DAPL RC と DAPL UD のメモリー消費 (値が小さいほど良い)

TMI は、低水準トランスポートを活用するためにインテル® MPI ライブラリーにより使用される API で、ロジックと一致する独自のメッセージを提供します。TMI の下で現在使用されている主要テクノロジーは PSM (インテル® True Scale ファブリックとインテル® Omni-Path アーキテクチャー・ファミリーのほかのアダプターで利用可能) です。このテクノロジーは IB-verbs レイヤーをバイパスし、トランスポート・バッファーのメモリー・フットプリントが固定されます。接続ごとに必要なメモリー量は、スケーラブルなファブリック・ソリューションの PSM に合わせるため、最新バージョンのインテル® MPI ライブラリーではさらに減少しています。



3 DAPL UD と TMI のメモリー消費 (値が小さいほど良い)

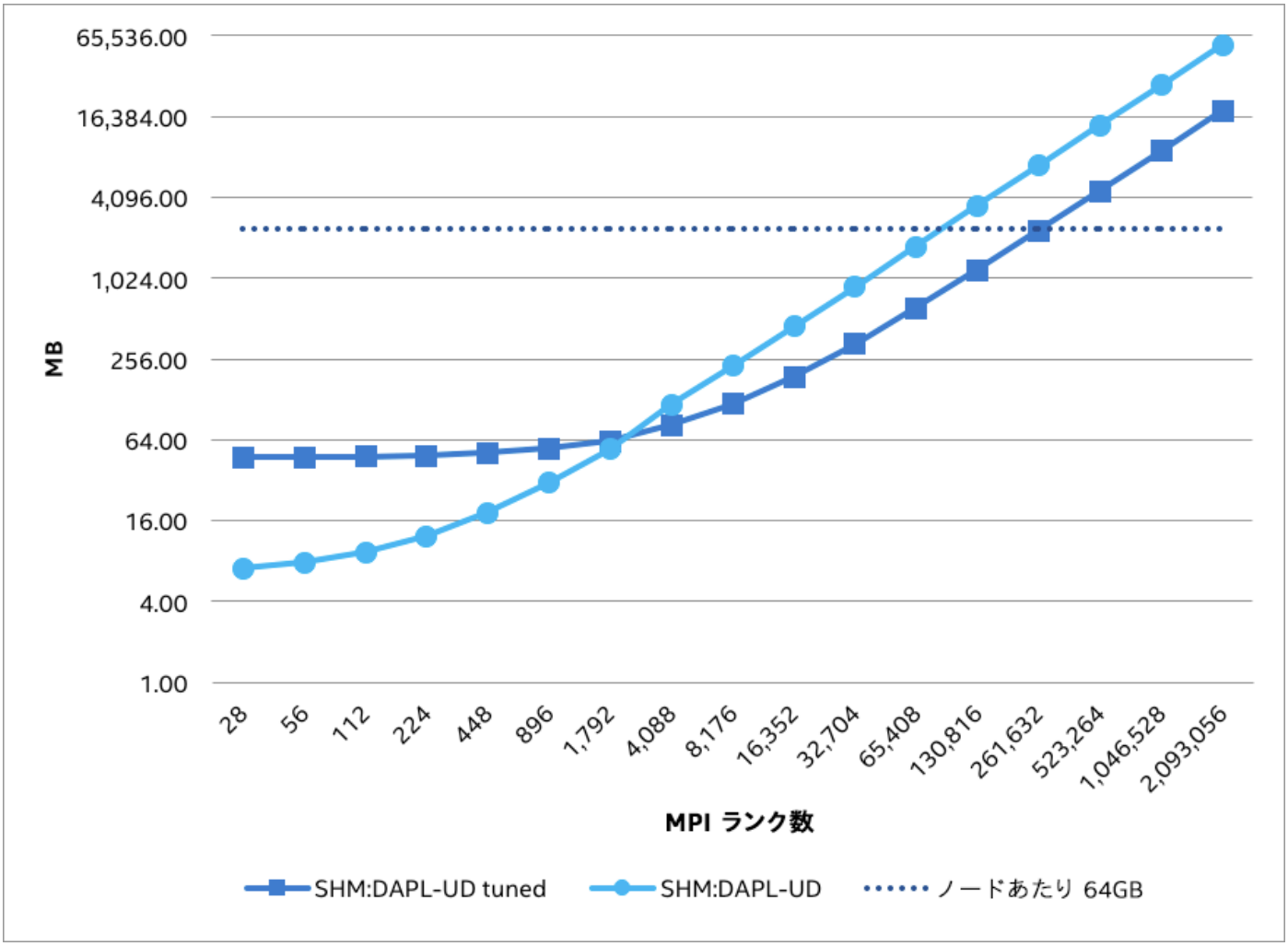
ラージスケールのメモリー・チューニング

DAPL UD はすでにメモリー効率に優れた手法で動作していますが、ファブリックをさらにチューニングすることによりラージスケール MPI 実行のメモリー効率を向上できます。**表 1** は、一部の環境変数のデフォルト値とチューニング後の値を示しています。

表 1 DAPL UD 環境変数のデフォルト値とチューニング後の値

環境変数	デフォルト値	チューニング後の値
I_MPI_DAPL_UD_SEND_BUFFER_NUM	ランタイムに依存	8208
I_MPI_DAPL_UD_RECV_BUFFER_NUM	ランタイムに依存	8208
I_MPI_DAPL_UD_ACK_SEND_POOL_SIZE	256	8704
I_MPI_DAPL_UD_ACK_RECV_POOL_SIZE	ランタイムに依存	8704
I_MPI_DAPL_UD_RNDV_EP_NUM	4	2

DAPL UD バッファープールのデフォルトサイズは MPI ランク数によってスケールしますが、チューニング後の設定では内部 DAPL UD バッファーの数を固定値にしています。同時に、低水準 QP 関連バッファーに必要なメモリーのチューニング・パラメーターも固定値にしています。これらの設定の影響は、特にラージスケール MPI を実行したときに顕著です (**図 3**)。



4 DAPL UD のデフォルト設定と DAPL UD のチューニング後のメモリー消費 (値が小さいほど良い)

まとめ

この記事では、異なるファブリックにおけるインテル® MPI ライブラリーのメモリー消費の最悪のケースを紹介しました。この情報は、厳密な予測ではありませんが、推定されるメモリー消費の上限値として利用できます。この情報から、HPC アプリケーションの特定のスケール条件下における残りのメモリー量を判断できるでしょう。

クラスター上で Mellanox* インターコネクトを使用するユーザーは、ラージスケールの実行でインテル® MPI のメモリー・フットプリントが減少する DAPL UD ファブリックに注目すべきです。この記事では OFA ファブリックを使用したメモリー消費を調査していませんが、このファブリックも優れたメモリー・スケーラビリティーが得られます (DAPL RC と DAPL UD の間になります)。

ユーザーは、インテル® MPI の内部バッファー構造を変更することにより、DAPL UD ファブリックのメモリー消費をさらに減らすことができます。インテル® MPI ライブラリーの最高のメモリー消費スケーラビリティーは、インテル® True Scale ファブリックとインテル® Omni-Path インターコネクト上で TMI ファブリックを使用したときに得られます。

Software

The Parallel Universe