ハイブリッド並列処理: 並列分散メモリーと共有メモリー・コンピューティング

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「Hybrid Parallelism: Parallel Distributed Memory and Shared Memory Computing」の日本語参考訳です。

並列コンピューティングには、主に 2 つの方法 (分散メモリー・コンピューティングと共有メモリー・コンピューティング) があります。科学および工学問題を解く大規模クラスターでは、プロセッサー・コア数の増加とともに、優れた分散メモリープログラムと共有メモリープログラムを組み合わせたハイブリッド・プログラミング手法がより一般的になっています。この傾向は、244 の仮想コアを備えたインテル® Xeon Phi™ プロセッサー製品の登場により加速され、多くの開発者がハイブリッド・プログラミング手法への移行を進めています。

この記事では、共有メモリー・プログラミング手法、分散メモリー MPI プログラミングを順に取り上げます。そして最後に、ハイブリッド共有メモリー/分散メモリー・プログラミングについて説明し、例を紹介します。

共有メモリー・コンピューティング

大規模な対称型マルチプロセッサー・システムは、大規模な計算集約型問題を解くために、より多くの計算リソースを提供します。科学者やエンジニアは、問題を単一プロセッサー・システムより高速に解くため、ソフトウェアをスレッド化しています。マルチコア・プロセッサーではスレッド化の恩恵を簡単に受けることができます。2 つのプロセッサーまたはコアを利用すると、理論的には半分の時間で問題を解くことができます。8 つのプロセッサーまたはコアでは、8 分の 1 になります。この恩恵により、マルチプロセッサーおよびマルチコアシステムで計算リソースを活用するようにソフトウェアを変更する価値は大いにあります。スレッド化は、最も一般的な共有メモリー・プログラミング手法です。スレッド化モデルでは、すべてのリソースは同じプロセスに属します。各スレッドには個別のアドレスポインターとスタックがありますが、アドレス空間とシステムリソースは共有します。一般的な共有メモリーアクセスを利用して、開発者はワーク、タスク、データを簡単に分割できます。欠点は、すべてのリソースがすべてのスレッドで利用可能であること、つまり、データ競合が発生する可能性があることです。

2 つ以上のスレッドが同じメモリーアドレスにアクセスし、少なくとも 1 つのスレッドがメモリーの値を変更すると、データ競合が発生します。計算の結果は、読み取りスレッドが値を読む前または後に書き込みスレッドが書き込みを完了するかどうかによって変わります。mutex、バリア、ロックは、実行フローの制御、メモリーの保護、競合状態の回避のために設計されたものですが、処理の進行を妨げるデッドロックや、mutex やロックの競合による実行フローの制限のような、ボトルネックとなるほかの問題が発生する原因にもなります。mutex とロックは万能ではありません。正しく使用されない場合、データ競合が発生します。特定のメモリー参照の周りではなくコードセグメントの周りにロックを配置することは、最も一般的な誤りです。さらに、すべての mutex、ロック、バリアにわたってスレッドフローの設計を追跡することは複雑であり、特に複数の共有オブジェクトや動的にリンクされたライブラリーを含む場合、開発者が保守および理解することは困難です。スレッドの抽象化は、プログラミングと制御が簡単になるように設計されています。

スレッドの抽象化

工学および科学分野で最も一般的な高レベルのスレッドの抽象化は OpenMP* です。オリジナルの OpenMP* の実装では、fork-join 並列構造で構築されていました。ワークは並列領域のスレッドプールでフォークされ、シーケンシャル領域でジョインされます。また、並列領域に対するフォークとジョインは何度も繰り返される可能性があります。そのため、共通のスレッドプールを利用することで、タスクごとに新しいスレッドを生成して破棄するオーバーヘッド・コストを回避していました。

スレッドプールでは、スレッドが生成され、プログラムが終了するまで存続します。DO ループまたは for ループは、OpenMP* で最も一般的な使用モデルです。DO ループまたは for ループが並列領域としてマークされると、OpenMP* ランタイム・ライブラリーは、自動的にループをタスクへ分解し、各 OpenMP* スレッドで実行できるようにします。表 1 に例を示します。OpenMP* のようなスレッドの抽象化を利用すると、スレッド・プログラミングの追跡、理解、保守がより簡単になります。

// プラグマは、直後の for ループが並列領域であることを定義して、その反復が複数の OpenMP* スレッドに分散されることを示します。

```
#pragma omp parallel
for for (i=0; i < n; i++)
{
    . . . ;
    computations to be completed ;
    . . . ;
}
```

C$ ディレクティブは並列領域の範囲を定義します。C$ 反復 (I) は複数の OpenMP* スレッドに分散 C$ されます。

```
!$ OMP PARALLEL
DO DO I=1,N
    . . .
    computation work is completed
    . . .
    ENDDO
!$ OMP END PARALLEL DO
```

表 1: C および Fortran の OpenMP* 並列領域の例

一部の新しい OpenMP* 構造には、並列タスクに加えて、ワークをコプロセッサーやアクセラレーターに割り当てる機能が含まれています。新しいスレッドの抽象化には、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) が含まれます。OpenMP* は C/C++ および Fortran で動作しますが、インテル® TBB は C++ テンプレートに基づくジェネリック・プログラミング・モデルであり、C++ のみで動作します。インテル® TBB は、明白な抽象化を提供しつつ、豊富な機能セットを含んでいるため、C++ プログラマーの間で広く利用されています。

別のスレッドの抽象化であるインテル® Cilk™ Plus は、コンパイラーとより緊密に連携して、いくつかのケースで優れたパフォーマンスを提供します。インテル® Cilk™ Plus は、並列 fork-join 構造に依存します。入れ子の並列処理はインテル® Cilk™ Plus とインテル® TBB では一般的ですが、OpenMP* では開発者が入れ子の並列処理を識別して宣言する必要があります。このため、インテル® TBB とインテル® Cilk™ Plus をライブラリーに含めることが理想的です。現在、インテルでは、(従来の) OpenMP* バージョンとインテル® TBB バージョンのインテル® マス・カーネル・ライブラリー (インテル® MKL) を提供しています (インテル® TBB バージョンはインテル® MKL 11.3 で登場しました)。

スレッドの抽象化	プロパティー
OpenMP*	構造化 fork-join 並列モデル (parallel for、タスク、セクションをサポート) オフロードをサポート 入れ子の並列処理は明示的に識別する必要がある C、C++、Fortran
インテル® TBB	parallel for、パイプライン、一般的なグラフと依存性、タスク、最適化されたリーダー/ライターロックなどをサポート 入れ子/再帰並列処理 テンプレート・ベース、C++ のみ
インテル® Cilk™ Plus	構造化 fork-join 並列モデル parallel for および fork コマンドをサポート 入れ子/再帰並列処理 C/C++

表 2: 一般的なスレッド化モデル

分散メモリー・プログラミング

多くのアプリケーションが単一のマルチプロセッサー・システムを上回る計算能力を求めるようになった結果、複数のシステムを接続してコンピューターのクラスターを形成し、1 つの計算ワークロードを解くようになりました。これらのシステムは専用の「ファブリック」でリンクされ、クラスターの各プラットフォームはクラスター内に個別のプライベート・メモリー領域を保持していました。プログラムは、別のプラットフォームと共有するデータを明示的に定義して、そのデータをクラスターの別のプラットフォームに送る必要がありました。

分散型計算アプローチでは、明示的なメッセージ・パッシング・プログラムが記述されました。プログラムはデータを明示的にパッケージして、そのデータをクラスターの別のシステムに送り、ほかのシステムはデータを明示的にリクエストして、プロセスで使用する必要がありました。その後、並列仮想マシンと呼ばれるワークステーションをリンクするアプローチが開発され、ワークステーションのネットワークを利用してプログラムを実行できるようになりました。ベンダーごとに専用のファブリックが提供され、個別のメッセージ・パッシング・ライブラリーがサポートされました。

間もなく、それらはコミュニティーにより、メッセージ・パッシング・インターフェイス (MPI) の下に統合されました。MPI では、プログラマーがクラスターへの問題の分解を明示的に制御して、メッセージが適切な順序で送信および受信されることを確認する必要があります。このアプローチでは、単一システムのメモリー制限がなくなり、解決できる科学問題のサイズが増加しました。多くのシステムの計算能力を組み合わせることで、非常に大きく、複雑で、計算負荷の高い問題を解くことも可能になりました。MPI で追加された主な機能は、片方向のデータ移動やリモート・ダイレクト・メモリー・アクセスのサポートです。これにより、送信プロセスと受信プロセスが、メッセージの送信と受信の呼び出しを同期しなくてもデータを移動できるようになりました。データアクセスの共有メモリー領域 (ウィンドウ) は明示的にセットアップして定義する必要があります。

MPI メッセージパッシングは、インクリメンタルな並列プログラミングには適していません。いったん、リモートシステムにメモリーを分散したら、アプリケーションの次のフェーズで中央のプラットフォームにワークを戻す理由がないため、MPI プログラムでは一般的に OpenMP* のようなほかのモデルよりも前処理が多くなります。しかし、MPI よりも OpenMP* のほうが、パフォーマンスがチューニングされた完全に並列のプログラムを高速に記述できるというわけではありません。一部の開発者は、最初に MPI でプログラムし、必要に応じて MPI をスレッドに変換しています。MPI の開発では、開発者が並列処理について熟考し、アプリケーションの並列アーキテクチャーや並列設計を考慮して、並列コードが始めから適切に設計されていることを確認する必要があります。

MPI プログラミングの利点は、1 つのシステムのメモリー量や 1 つのシステムのプロセッサー/コア数に制限されないことです。別の利点は、データとプログラムの適切な分解が開発者に求められることです。MPI ではスレッドほどデータ競合が発生することはありませんが、開発者が (不適切に設計された依存性により発生しないイベントやメッセージをすべての MPI プロセスが待つ) デッドロックを引き起こす MPI プログラムを記述する可能性があります。MPI では、開発者がメッセージの送信と受信を明示的に記述します。図 1 はこの様子を示しています。

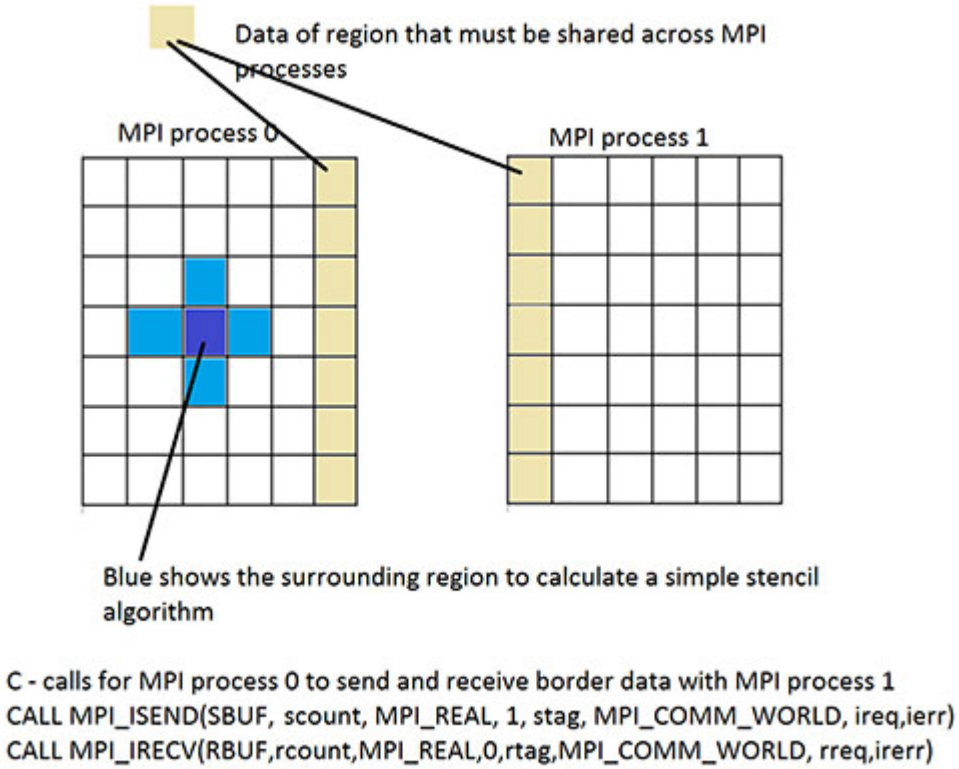

図 1: データの送受信の MPI プロセス

MPI プロセスは同じプラットフォームの複数のプロセスとして実行することも、複数のプラットフォームにわたって分割することもできるため、MPI でプログラムしたら単一のプラットフォームでも、複数のプラットフォームでも実行できるように思われるかもしれません。しかし、MPI ライブラリーのメモリー消費について考慮することが重要です。MPI ランタイム・ライブラリーは、送受信メッセージを制御するためバッファーを利用します。MPI プロセスの数が増加すると、MPI ライブラリーはアプリケーションのほかのプロセスとの送受信に対応する必要があります。これは、ランタイム・ライブラリーがより多くのメモリーを消費し、より多くのメッセージのデスティネーションと受け取り位置を追跡する必要があることを意味します。

クラスターサイズの増加とともに、MPI ランタイム・ライブラリーのメモリー使用量が増えています。さらに複雑なことに、メモリー使用量の増加は、共有空間ではなくクラスターのすべてのシステムで発生します。参考文献では、不要なメモリー消費を最小化するため、MPI ライブラリーを改善する方法を示しています [1]。

インテル® Xeon Phi™ コプロセッサー (開発コード名 Knights Corner (KNC)) は、4 ウェイ対称型マルチスレッド (SMT) に対応した 60 を超えるプロセッサーを搭載し、約 240 のアクティブなスレッドまたはプロセスが利用できるように設計されています。開発者が MPI のみを使用して 4 つのカードで MPI アプリケーションを実行した場合、MPI プロセスの数は 960 になります。インテル® MPI ライブラリーのメモリー消費のグラフに基づくと [1]、MPI ランクごとに約 50MB または合計 48GB (カードあたり 12GB) のメモリーが消費されます。このデータが MPI メモリー消費の上限であると考えられます。ここでは、MPI ライブラリーが 50MB の 34% (約 17MB) のメモリーを使用するとします。アプリケーションが KNC † で利用可能な SMT のうち 3 つのみを使用する場合 (つまり、720 の MPI プロセスでそれぞれ 17MB のメモリー、合計 12.2GB またはカードあたり 3GB のメモリーが必要) について考えてみます。上位モデルの KNC † カードに搭載されているメモリーは 16GB です。メモリー消費が多くない問題において、MPI ライブラリーは 16GB のうち 3GB (メモリーの 1/8 以上) を消費します (オペレーティング・システム、バイナリー、その他のサービスで消費されるメモリーは含みません)。これだけでも、問題を解くためにデータが利用可能なメモリー量は大幅に減ります。MPI ライブラリーのメモリー消費は、ハイブリッド・プログラミングへの移行を後押しする要因の 1 つです。Pavan Balaji 氏は、「コアあたりのメモリー量が減

るとともに、アドレス空間の通信には引き続き MPI を使用しながら、アプリケーションがマルチコアノードで共有メモリー・プログラミング・モデルを使用する傾向が高まるでしょう。」と述べています [2]。

コミュニティーは、この潜在的なメモリー消費とその対応に必要なステップを認識しています [3、4]。

1 つのプログラムでスレッド化と MPI を組み合わせて使用する場合、開発者はスレッドの安全性と MPI ライブラリーの認識にも注意する必要があります。MPI 標準はスレッド化されたソフトウェア向けに 4 つのモデルを定義しています。

- **MPI_THREAD_SINGLE**。コードはシーケンシャルで、1 つのスレッドのみ実行します (すべての MPI 呼び出しがシーケンシャル領域にある場合、OpenMP* でも動作します)。
- **MPI_THREAD_FUNNELED**。1 つのスレッドが MPI ライブラリーのすべての呼び出しを行います。OpenMP* では、並列領域の内部で呼び出しを行うことができますが、omp master ディレクティブ/プラグマを使用して、マスタースレッドがすべての MPI 呼び出しを行うように保証すべきです。
- **MPI_THREAD_SERIALIZED**。すべてのスレッドが MPI ライブラリーの呼び出しを行うことができますが、常に 1 つの MPI 呼び出しで 1 つのスレッドのみがアクティブになるように、開発者がコントロールを配置します。
- **MPI_THREAD_MULTIPLE**。任意のスレッドがいつでも任意の MPI 呼び出しを行います。

これまで私が調査したハイブリッド・コードはすべて、最初のモデル (MPI_THREAD_SINGLE) を使用していました。MPI 呼び出しは MPI コードのシーケンシャル領域で行われます。上記の最初の 3 つのモデルは簡単に使用できます。MPI_THREAD_MULTIPLE モデルは、メッセージがスレッドではなく MPI プロセスに送られるため、より多くのことを考慮する必要があります。2 つのスレッドが同じ MPI プロセスとの間でメッセージの送受信を行う場合、どちらのポイント (レシーバーまたはセンダー) からであっても、送信/受信呼び出しを行うスレッドに関係なく、メッセージの順番が正しくなるようにコードを設計する必要があります。

MPI_THREAD_MULTIPLE をメッセージパッシングに使用する場合、追加のオーバーヘッドが発生します。Linux* クラスターで測定されたデータでは、これらの差がわずかであることがレポートされています [5]。上手く設計されている MPI_THREAD_MULTIPLE は適切に動作します。上記のレポートは、アプリケーション・データではなく、ベンチマーク・データであることに注意してください。

ハイブリッドのサンプルコード

NAS Parallel Benchmarks は、サンプル並列コードの広く利用可能なリファレンス実装を提供します [7]。NAS Parallel Benchmarks の Multi-Zone バージョンには、ハイブリッド MPI/OpenMP* のリファレンス実装が含まれています。Multi-Zone ポートは、OpenFlow* などで使用される流体力学コードを表すことを目的としています。ここでは、各ゾーンのソリューションを求めてから、各時間ステップですべてのゾーンにわたってゾーン境界値を交換するようにコードを変更しています。ローカルゾーンのソリューション境界の交換は各時間ステップで繰り返されます。この記事では、さまざまな構成の MPI プロセスと OpenMP* スレッドを使用して、インテル® Xeon Phi™ コプロセッサー・カード上でクラス C 問題サイズについて結果を収集しました。クラス C 問題サイズは、1 つの KNC † コアの 1 つのプロセスで実行することも、240 コアで実行することもできます。1 つの MPI ランクとさまざまなスレッド数での実行では、良好なスケーラビリティーが示されました。1 つのスレッドに固定して MPI プロセスの数を増やすと、より優れたスケーラビリティーが示されました。これは、共有メモリー・プログラミングと分散メモリー・プログラミングの違いではなく、並列処理のレベルによるものです。

SMP/MLP モデル (マルチプロセス共有メモリー領域を使用する別の並列プログラミング・モデル) を使用したレポートでは、MPI + 1 OpenMP* スレッドと SMP/MLP + 1 OpenMP* スレッドを比較した場合、SP-MZ クラス C 問題では SMP/MLP コードのほうが MPI よりもパフォーマンスがやや高くなっています [6]。つまり、共有メモリーと分散メモリー・プログラミングでスタイル固有のパフォーマンス優位性はありません。インテル® Xeon Phi™ コプロセッサーで実行した SP-MZ ハイブリッド MPI/OpenMP* 分解では、約 100 MPI プロセスまではハイブリッ

ドよりも MPI のほうが適切にスケーリングしていますが、この値を超えると、ハイブリッド MPI/OpenMP* 分解のほうが結果は良くなっています。

グラフ (図 2 を参照) を 1 MPI/1 OpenMP* スレッドから開始すると、シーケンシャルに実行する時間の範囲が大きすぎて、スレッド/プロセス数の違いを判別できません。この理由により、グラフの表示は合計 50 スレッドから始まっています。スレッドの総数は、MPI プロセスの数にプロセスごとの OpenMP* スレッドの数をかけた値です。

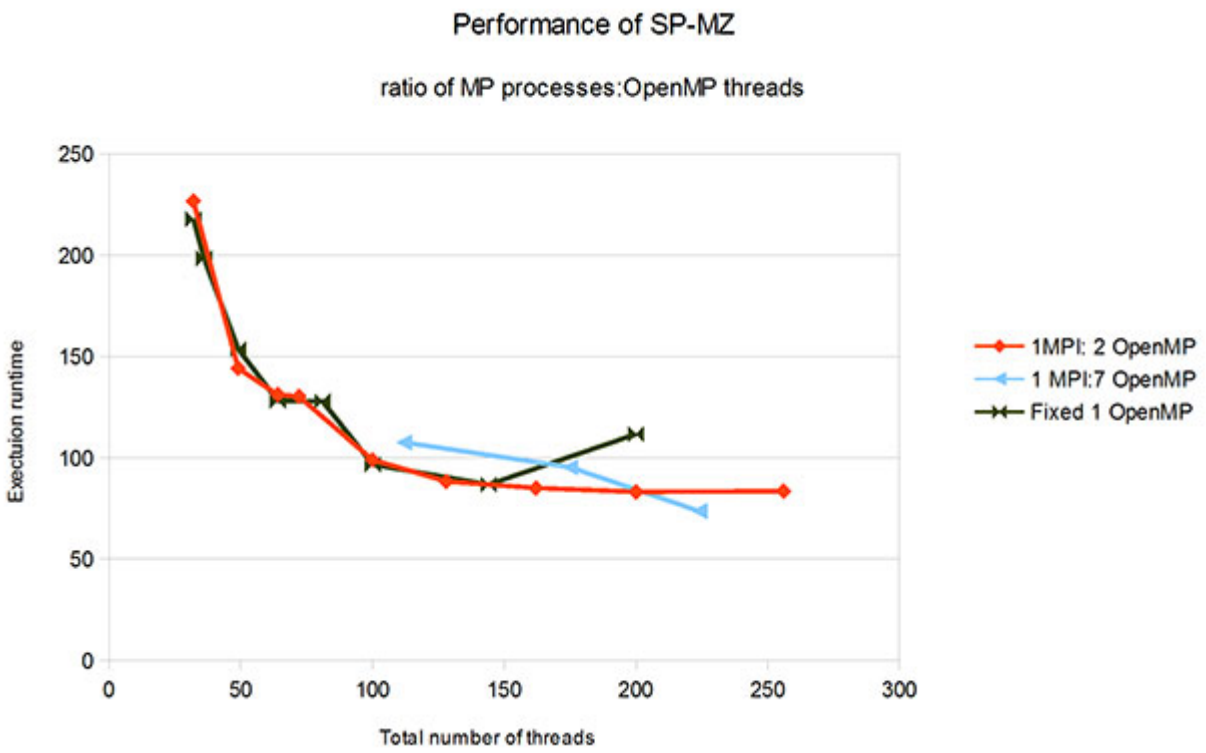

図 2: MPI プロセスと OpenMP* スレッドの比率および数によるパフォーマンスの変化

最高のパフォーマンスは、32 MPI プロセスと 7 つの OpenMP* スレッド (合計 224 スレッド) で達成されています。SP-MZ コードは、適切に定義されたメッセージ・インターフェイスで多くの独立した計算を行うことができます。メッセージパッシングが多いアプリケーションでは、最適な比率は異なります (1:2、1:4、1:7、1:8 など)。データを収集して測定し、最良の比率を決定するには、開発者は MPI のワークロード・バランスに加えてスレッドのワークロード・バランスも確認すべきです。グラフから分かる重要なポイントは、MPI プロセスを増やすだけではある地点でコードは高速に実行されなくなりますが、ハイブリッド・モデルを使用するとパフォーマンスは引き続き向上します。

この NAS SP-MZ コードは、並列処理を 2 つの異なるレベルで行います。MPI 並列処理の下で OpenMP* 並列処理を入れ子にするか、MPI 並列処理と同じレベルに OpenMP* 並列処理を配置します。ケースによっては、いずれかの設計のほうが優れていることもあるでしょう。また、いくつかのスレッドがタスクの MPI プロセスと同じレベルにあるケースや、高いレベルのワーカーが低いレベルで複数のワーカースレッドを使用するケースがあるかもしれません。

開発者を支援するため、ハイブリッド・プログラミングに関するルールとデータ測定のセットがあることが理想的ですが、開発環境はまだそのレベルに達していません。開発者は、適切な判断と優れた設計を行うことが推奨されます。場合によっては、最良のデザインパターンに従うためではなく、成り行きで OpenMP* が並列処理コードに追加されることがあります (つまり、コードを並列化する最良の方法としてではなく、DO または for ループだからという理由だけで OpenMP* プラグマが配置されます)。優れたスケーリングには、優れた並列処理が必要です。

優れた並列処理は、MPI またはスレッド化手法で表現できます。開発者を支援する多くのツールがあります [8、9]。インテル® Advisor は、並列処理を認識、設計、モデル化する手段を提供します。インテル® VTune™ Amplifier XE は、OpenMP* 並列領域のパフォーマンスと動作を測定します。インテル® Trace Analyzer は、MPI コードの動作を理解できるように、MPI パフォーマンス解析情報を表示します。TAU Performance System* および ParaProf も、OpenMP* と MPI のパフォーマンス・データとイベントを収集して、パフォーマンス・データを表示します。これらのツールはすべて、設計とパフォーマンスを向上するため、開発者がコードのパフォーマンスを理解できるように支援します。

まとめ

パフォーマンスを追求する開発者は、リソース (特にメモリー) 消費を向上する機会を提供する、ハイブリッド・プログラミング手法を調査することを推奨します。SP-MZ で示されたような複数レベルの並列処理も、優れたパフォーマンスを生成します。MPI プロセスとスレッドの比率はアプリケーションにより異なるため、テストと評価が必要です。ソフトウェアは、スレッドの抽象化のように、再コンパイルなしでスレッド数を制御できる方法で記述すべきです。OpenMP* およびインテル® TBB は、最もよく使用されているスレッドの抽象化です。開発者は、ハイブリッド並列プログラミング・モデルを採用し、さらなるパフォーマンスの向上に取り組むべきです。

参考文献

1. Durnov, D. and Steyer, M. インテル® MPI のメモリー消費 (The Parallel Universe Issue 21)
2. Balaji, P. et al. MPI on a Million Processors. (英語)
3. Goodell, D., Gropp, W., Zhao, X., and Thakur, R. Scalable Memory Use in MPI: A Case Study with MPICH2 (英語)
4. Thakur, R. MPI at Exascale (英語)
5. Thakur, R. and Gropp, W. Test Suite for Evaluating Performance of MPI Implementations That Support MPI THREAD MULTIPLE (英語)
6. Jin, H. and Van der Wijngart, R. Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks (英語)
7. NAS Parallel Benchmarks (英語)
8. インテル® VTune™ Amplifier XE、インテル® Advisor、インテル® MKL、インテル® Trace Analyzer (インテル® Parallel Studio XE Cluster Edition に同梱)
9. TAU Performance System* および ParaProf (tau.uoregon.edu から入手可能)

† 開発コード名

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。