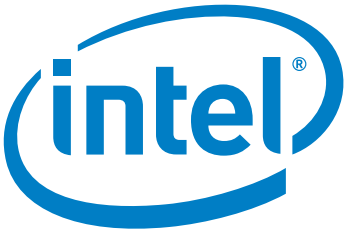


インテル® コンパイラー OpenMP* 入門

デュアルコア / マルチコア対応アプリケーション開発 ①

目 次

注記 :

『デュアルコア / マルチコア対応アプリケーション開発』は、次の 4 巻から構成されています。

① インテル® コンパイラー OpenMP* 入門
② インテル® C/C++ コンパイラー OpenMP* 活用ガイド
③ インテル® Fortran コンパイラー OpenMP* 活用ガイド
④ インテル® コンパイラー 自動並列化ガイド

1. はじめに

インテルが提供するソフトウェア開発ツール「インテル® コンパイラー バージョン 9.0」は、Linux* および Windows* プラットフォーム上で、自動並列化とユーザーによる明示的な並列化の指定をサポートします。これらの開発ツールを利用することで、デュアルコアとマルチコアを搭載したシステムをより有効に活用することができます。

インテル® Xeon® プロセッサーやインテル® Itanium® プロセッサーなどの従来のシングルコア・プロセッサー向けに開発されたアプリケーションは、同じプロセッサー・ファミリーのデュアルコア・プロセッサーおよびマルチコア・プロセッサー上で実行が可能です。デュアルコア・プロセッサーおよびマルチコア・プロセッサー上では同時に複数のプロセスの実行が可能であり、オペレーティング・システム (OS) がカーネルの実行、ドライバー、ライブラリー、アプリケーションのすべての処理を最適にスケジュールすることで、大幅な処理性能の向上を図ることが可能です。更に、アプリケーションを再コンパイルしたり、新しいライブラリーをリンクしたりすることで、その性能を大幅に向上させることが可能です。これは、アプリケーションの実行時に複数のスレッドを生成して、今まで逐次的に実行されていた処理を、各スレッドが同時・並列に実行し、より短時間で処理を終了することが可能となるためです。ただ、一般には従来のシングルプロセッサー、シングルコア上で開発、利用されてきたプログラムやアプリケーションは、デュアルコアおよびマルチコアをターゲットとして開発されていません。そのようなアプリケーションをデュアルコアとマルチコア上でいかに効率良く、より高速に実行できるようにプログラムすることが重要となります。

今後はより多くのコアが 1 つのプロセッサー上に実装され、デュアルコア・プロセッサーやマルチコア・プロセッサーが一般的になります。この資料では、デュアルコアとマルチコアの持つ能力を最大限に発揮するためのキーとなる、マルチスレッドでのアプリケーション開発のためのプログラミングについて説明します。本資料を通して、デュアルコアとマルチコアに対応した、インテルの充実したソフトウェア開発ツールについてもご理解いただければと思います。

2. 並列処理 : マルチプロセスとマルチスレッド

今後のデュアルコアやマルチコア構成のマイクロプロセッサーでは、並列処理技術がより重要になります。データベースや WEB などのシステムでは、一度に複数のユーザーからの大量のデータ処理が要求されます。このような場合には、これらの複数のタスクを同時に処理するマルチタスクが行われています。デュアルコアやマルチコア構成のマイクロプロセッサーを搭載したシステムでは、同時により多くのタスクを処理することが可能になります。これに対して、ある特定の問題の計算処理を複数のプロセッサー、複数のサーバを利用して高速に処理する並列計算という処理もあります。並列計算の目的は、対象となる問題を複数のコア、プロセッサーを同時に利用して短時間で解くことといえます。このような並列計算というのは、一部の科学技術計算分野だけで利用される特殊なものだという認識があるかもしれません。しかし、デュアルコアからマルチコアへと、今後のマイクロプロセッサーの進化を考えた場合、このような並列計算をごく一般的なプログラムでも利用することがより一般的になるでしょう。デュアルコアやマルチコアのもつ高いコストパフォーマンスをさらに引き出すためにも、このような並列計算の実現は、今後どのようなプログラムを開発するにあたっても、大きな意味を持ちます。

並列処理を理解するためには、プロセスとスレッドという言葉について理解することが必要です。先に述べたように、並列処理とは「同時に複数の処理（タスク）」を行うことであり、これらの複数の処理は OS によって効率良く処理されます。この OS による処理単位がプロセスとスレッドということになります。

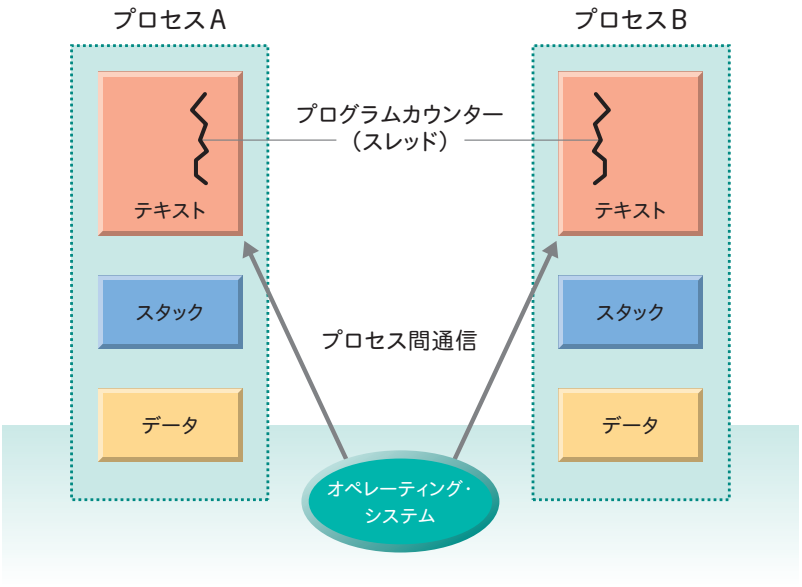
OS は、要求されたタスクに対して複数のプロセスを起動してその処理を行います。これらのプロセス内で生成されて実際の処理を行うのがスレッドとなります。プロセス内で複数のスレッドを生成して、並列処理を行うことをマルチスレッドと呼びます。

プロセスは、独立した仮想メモリー空間とスタックをもち、1 つ以上のスレッドから構成されます。これらの仮想メモリー空間は、OS によるメモリー管理機能により、各プロセス間での干渉が防止されています。言い換えれば、この独立した仮想メモリー空間がプロセスになります。複数のプロセスを利用して行う並列処理がマルチプロセスとなります。

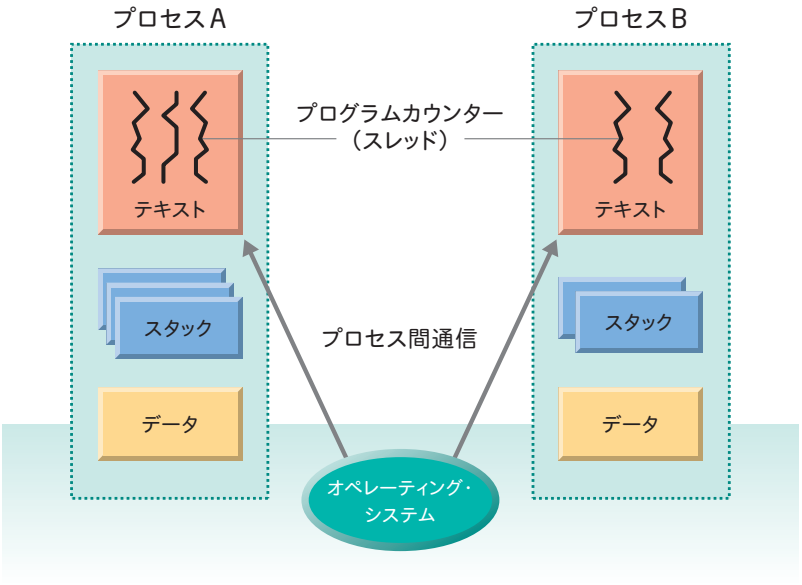
プロセスの起動やそのスケジュール管理など OS が提供する様々なサービスも、複数のタスクを同時に処理するマルチプロセスとなります。スケジューリングについては様々な技術が取り入れられており、複雑なワークロードへの対応も可能です。デュアルコアやマルチコアを搭載したシステムにおいては、OS がマルチプロセスの効率的な処理を行うことで大幅な処理性能の向上が可能となります。

このように複数のタスクを効率良く処理するにはマルチプロセスは最適です。しかし、このマルチプロセスで 1 つの処理・タスクを分割して複数の処理として並列に実行し、そのタスクの高速処理を行うには効率上の問題があります。例えば、データベースの処理などでは、非常に複雑な 1 つの処理要求（クエリー）があった場合、これらの処理要求を内部で複数のクエリーに分割し、親プロセスと親プロセスが起動する子プロセスで処理することにより、マルチプロセスでの処理が可能になります。この場合、親プロセスから起動される子プロセスは、自身のプログラムのロードやメモリーの確保、初期化などの作業を行う必要があります。これらの処理は OS にも大きな負担になります。

また、プロセスは先に示したように、メモリー管理機能によって相互に保護されます。従って、プロセス間でのデータのやり取りは、すべて OS が介在したプロセッサー間でのデータ通信を必要とします。このようなプロセッサー間通信は、計算スピードと比較して非常に遅い処理となるため、ボトルネックとなり並列処理での性能向上を十分に図ることが難しくなります。また発生させたプロセスについては、それらのスケジューリングと同期処理などの点で、OS の負担が大きくなります。



マルチプロセス / シングルスレッド



マルチプロセス / マルチスレッド

一方、スレッドはプロセス内に 1 つ以上生成されたもので、現在の OS は、ほとんどこのスレッドに対して CPU へのスケジューリングを行います。プロセス内で生成される複数のスレッドが、マルチスレッドとなります。マルチスレッドは、マルチプロセスとは違ってプロセス内の仮想メモリー空間といったリソースやコンテキストを共有し、固有のスタックとプログラムカウンター (PC) を個別に持つことになります。

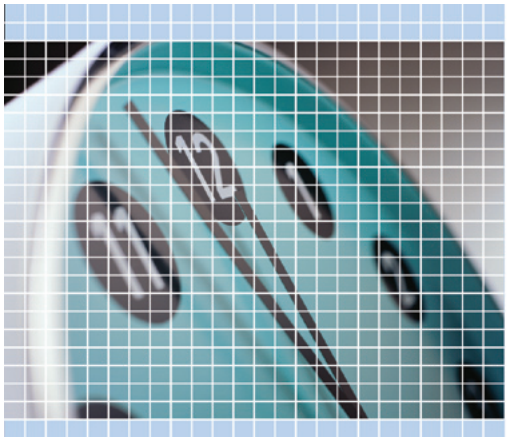
個々のプロセスが固有の仮想空間上で動作するプロセスと異なり、このメモリー空間を共有することで、スレッドは並列処理を行う上で非常に効率的な実行を可能としています。メモリー空間の共有により、スレッド間でのデータのやり取りは直接アクセスが可能となり、OS を介した通信のようなオーバーヘッドの大きなオペレーションを必要としません。同時に、スレッドの生成や切り替えはプロセスの場合と比較して高速であり、システムのリソースへのインパクトも非常に小さくなります。このようなスレッドの特性を最大限に活用することで、効率のよい並列処理を行うのが、マルチスレッド・プログラミングです。

3. 計算粒度と負荷分散

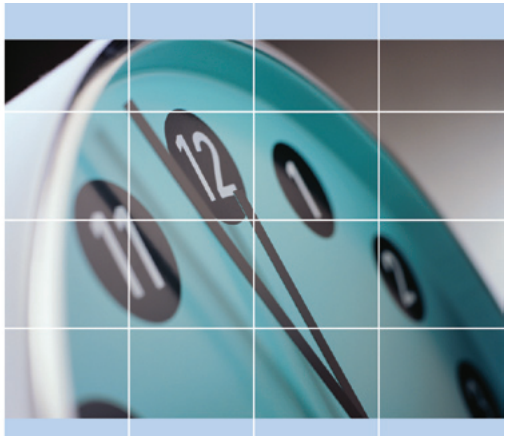
並列処理とは、逐次処理では順番に処理されていた一連の処理を複数の CPU で同時に処理することでもありますが、この並列処理を考える上で大事なことがあります。これは並列処理の効率にも関わることですが、同時に複数の処理を行う場合、各処理における並列計算の「粒度」が非常に重要になります。「粒度」とは、一般には並列計算における処理レベルを指しますが、実際には分類基準があるわけではありません。1 つの並列タスク内の仕事量と考えることもできます。プログラム中の計算ループで並列化するような場合を「細粒度」(粒度が細かい) と呼び、関数やサブルーチン呼び出しなどを含むプロシージャー間での並列処理を「疎粒度」(粒度が粗い) などと呼んでいます。

一般に、並列計算機上でプログラムを効率良く実行するためには計算の粒度を適切に設定することが必要になります。粒度が小さ過ぎると実際の計算よりもスレッドの切り替えや通信のオーバーヘッドが大きく、効率が良くないということになる場合もあります。粒度が粗すぎると、負荷の不均衡のためにパフォーマンスが低下する可能性もあります。並列タスクの適切な粒度を決定して (通常は粗い粒度の方が良いと言われています)、負荷の不均衡と通信オーバーヘッドを回避し、最適なパフォーマンスを得ることが必要です。

細粒度での領域分割



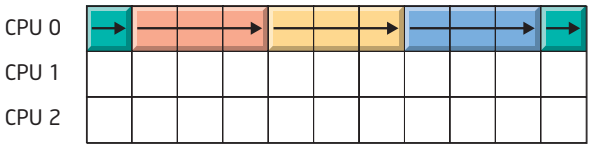
疎粒度での領域分割



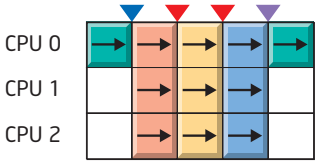
計算粒度を模式的に示した図 : 領域を例えばスレッド数に分割し、それぞれの領域を個々のスレッドが計算するような場合は疎粒度での領域分割となります。一方、領域を細かく分割し、各スレッドが複数の領域を順次処理するような場合は細粒度での領域分割となります。

この並列処理の粒度について模式的に示します。ここでは前処理の後、処理タスク 1、2、3 を順次実行することとして「粒度」について考えます。処理タスク 1、2、3 をそれぞれ並列化して同時に処理するような細粒度のケースでは、各処理を複数のプロセッサーで処理することになりますが、各処理の開始と終了時にスレッドの生成、終了、同期処理などを行う必要があります。これがスレッド切り替えのオーバーヘッドになります。一方、疎粒度な単位で処理タスク 1、2、3 をそれぞれ同時に複数のプロセッサーで処理するようなケースでは、スレッドの生成、終了、同期処理などを最小化できます。ただ、このような疎粒度での並列化は、細粒度での並列性を見つけるよりも、より深い洞察とプログラムの検討が必要になります。

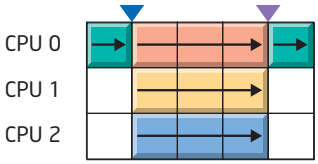
逐次処理



並列処理：細粒度での並列処理



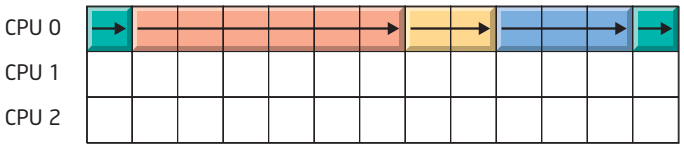
並列処理：疎粒度での並列処理



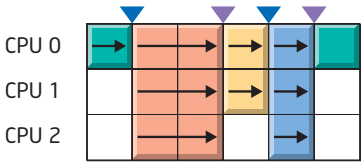
計算粒度：細粒度と疎粒度での並列処理

並列処理の効率を考える上でもう 1 つ重要なのが、スレッド間のアプリケーションのワークロード負荷分散です。この負荷分散は、アプリケーションのパフォーマンスにとってきわめて重要であり、プログラムの並列化を行う場合には十分な注意が必要です。負荷分散の目的はスレッドのアイドル時間を最小限に抑えることです。ワークロードが適切に各プロセッサーに割り振られない場合、アイドルしているプロセッサーは、他のプロセッサーの処理が終了するのをただ待つだけの状態になってしまいます。

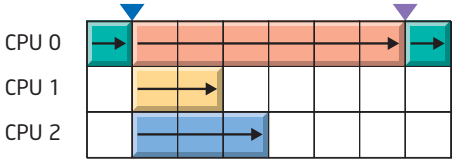
逐次処理



並列処理：細粒度での並列処理

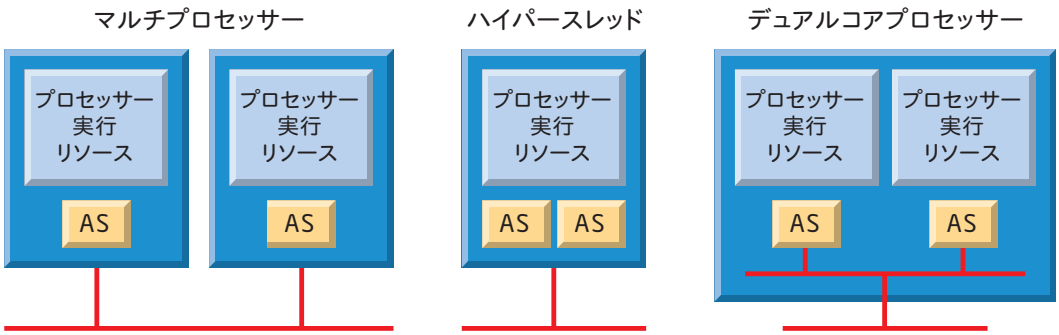


並列処理：疎粒度での並列処理



4. ハイパースレッドとマルチコアシステム

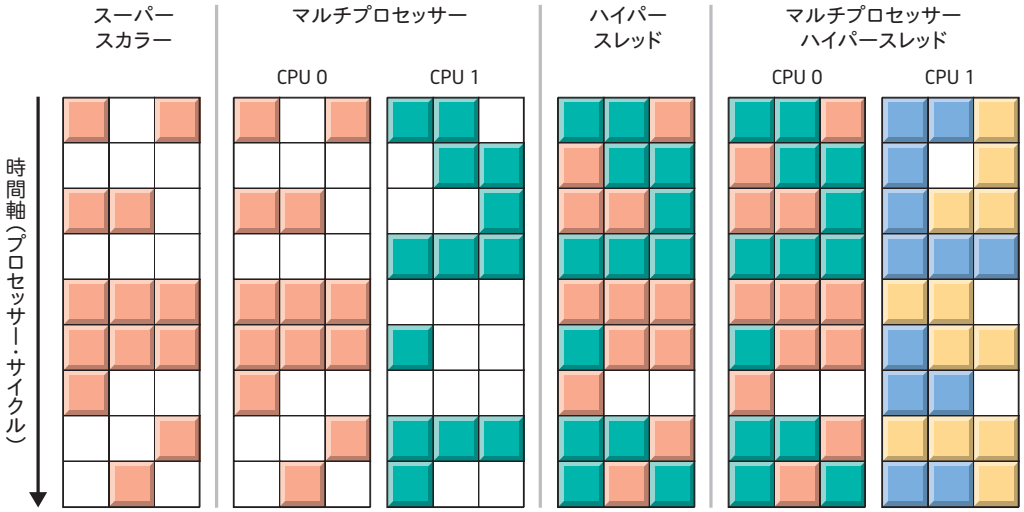
マルチスレッド並列処理は、デュアルコアとマルチコアシステムだけで有効なのではありません。SMP（対称型マルチプロセッシング）や NUMA（不均等メモリーアクセス）といったアーキテクチャーを持つ、より多くのプロセッサーとプロセッサー・コアの実装が可能なシステムでも、インテルのソフトウェア開発ツール「インテル® コンパイラー バージョン 9.0」を利用することで容易な並列処理が可能となります。また、インテル® コンパイラーによる並列処理機能は、ハイパースレッディング・テクノロジー（HT テクノロジー）を搭載したプロセッサーでも利用可能です。この HT テクノロジーとデュアルコアやマルチコアの技術は、ユーザーからするとほぼ同じ技術のように思われますが、プロセッサーの構成とそこから得られる性能は大きく違ってきます。



AS（Architecture State）は、汎用レジスターや制御レジスター、APIC（Advanced Programmable Interrupt Controller）レジスターなどプロセッサーの状態を保持するものです。

HT テクノロジーでは、2 つのスレッドを 1 つのプロセッサー・コア上で同時に実行することを実現しました。従来のプロセッサーでも複数のスレッドの処理は可能でしたが、同時に実行できるスレッドは 1 つに限られていました。HT テクノロジー対応のプロセッサーは、物理的には 1 つのプロセッサーでありながら、対応する OS やアプリケーションからは、あたかも 2 つのプロセッサーがあるかのように機能します。シングルコアのプロセッサーがプログラムを実行する場合、常にプロセッサー・コアのリソースを 100% 使い切ることはありません。HT テクノロジーはこのようなリソースの空きを活用してもう一つのインストラクション・ストリームを同時に実行できるようにする仕組みです。その結果、HT テクノロジー対応のプロセッサーを搭載したシステムでは、OS が複数のスレッドのスケジューリングを行うことが可能になっています。HT テクノロジーは、マルチスレッドの同時処理を可能としますが、実際に 2 つのスレッドを同時に実行するには、リソースが不足しています。そのため、マルチスレッド化されたプログラムは HT テクノロジーで受けることのできる性能上の恩恵は 20%-30% が限度といわれています。

デュアルコアとマルチコアシステムでは、プロセッサー上に完全に独立したプロセッサー・コアを搭載することで、各スレッドに対してその実行のための完全なリソースを提供します。デュアルコアとマルチコアでは、複数スレッドの実行に際して、一方のコアが他方の実行を阻害することのないように設計されています。そのためマルチスレッド化されたプログラムはコア数倍（デュアルコアの場合、2 倍）に近い性能向上を得ることも可能です。



各ボックスがプロセッサーの各演算機を示しています。1クロックあたり、複数の演算器が同時に並列に動作します。

5. マルチスレッド対応のための開発ツール

先に述べたようにデュアルコアとマルチコア上でのアプリケーションの高速化には、アプリケーションの実行時に、複数のスレッドが並列に処理を行うことが必要になります。ここで問題となるのはアプリケーション・プログラムに対して、並列処理を適用する為の特別な作業（OS が提供するスレッド制御の API を使用）やそのための開発工数が必要になるかということです。実際には、マルチスレッド化や並列化といった作業にはそれほどの時間を必要とするものではありません。マルチスレッド対応の開発ツールがあれば、これらの並列化は容易に行うことが可能です。プログラムの開発者やプログラマは、プログラムの本質的なロジックを記述することに専念し、並列化については、既に高度に最適化・並列化されたライブラリーを利用したり、並列化コンパイラーの支援をしたりすることによって、プログラムのマルチスレッド化を図ることが現在では可能になっています。

インテルのソフトウェア開発ツール「インテル® コンパイラー バージョン 9.0」は、優れたマルチスレッド対応の開発ツールです。32bit と 64bit Linux* および 32bit と 64bit Windows* のオブジェクト・コードを作成し、それぞれのプラットフォームでのプログラムの性能を最大限に引き出すことが可能です。

バージョン 9.0 でサポートされているプログラムのマルチスレッド化機能は、大きく 2 つに分けられます。1 つは自動並列化、そしてもう 1 つが OpenMP* のサポートです。自動並列化では、その名の通り、コンパイラーがソースコードを解析し、自動的にプログラムの構造に適したマルチスレッド実行が可能な実行モジュールを作成します。自動並列化のオプションによって、アプリケーション内で複数の実行スレッドや拡張機能が自動作成されるようになります。生成されるバイナリはマルチコア・プラットフォーム向けに最適化され、アプリケーション処理性能の効率化が図られます。

OpenMP* は、ユーザーがプログラムの並列化を指示する宣言子をプログラム中に記述することで、マルチスレッド並列プログラムを開発する枠組みを提供します。バージョン 9.0 では、OpenMP* 2.5 仕様の実装により、最新の OpenMP* の機能が利用可能となっています。

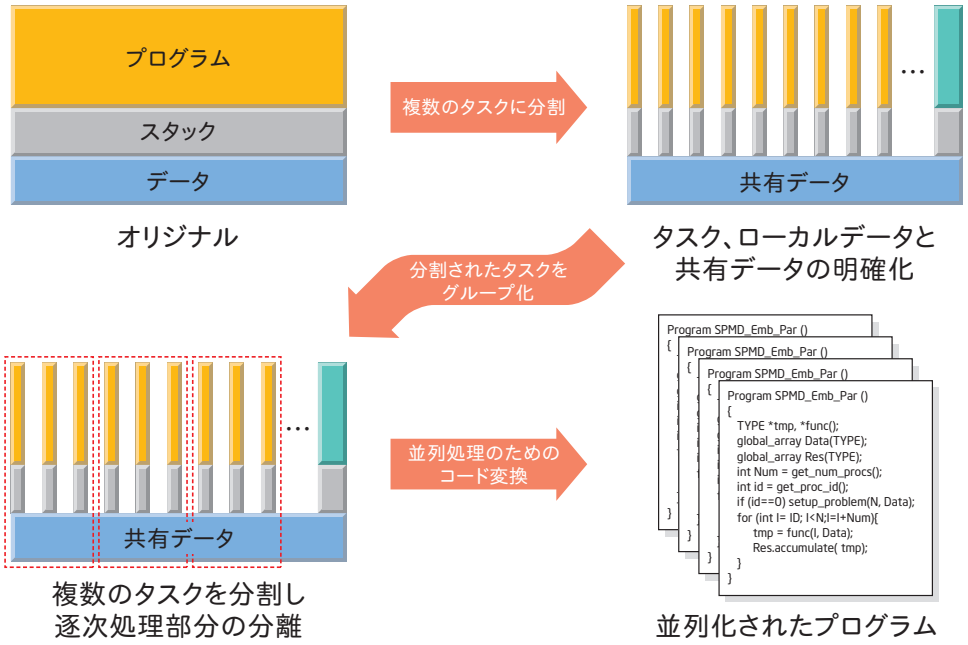
ソフトウェア開発者や研究者がプログラムを開発する目的は、そのプログラムの並列化を行うためではありません。ある処理、解析を目的にプログラムを書き、そのプログラムをプラットフォーム上で効率良く、高速に実行できることを目的としています。インテル® コンパイラーをはじめ、並列化に対応したコンパイル・ツールは、並列化という必要ではあるが本質的ではない手間のかかる作業を開発者の代わりに行います。これにより、開発者がプログラム開発における本来の目的である、アルゴリズムの実装やロジックの検証のための作業に専念することができます。

5.1 自動並列化

自動並列化は、ループとプログラム構造をコンパイラーが解析し、プログラム中で並列実行可能な領域を見つける、高度なプログラム解析機能に基づいた並列処理です。自動認識された並列実行領域は、その領域内でのマルチスレッド並列プログラムとして実行されます。ループの展開やデータのアライメントの共有、スレッドのスケジューリング、同期化といった、低レベルの細かい作業をプログラマーが行わなくても並列化が行われるようになり、マルチプロセッサー・システムやハイパースレッディング・テクノロジー対応システムが持つスレッド機能を十分に活かすことができます。

自動並列化は、コンパイル時にコンパイル・オプションとして、/Qparallel スイッチ（Windows*）、-parallel スイッチ（Linux*）を指定することで可能となります。これらのオプションが指定されると、コンパイラーは入力されたファイルに対してその並列処理の可能性の解析と並列化のための最適化を行います。

実際には、どのように変換されるか実際に見ていきます。



並列化コンパイルフロー

自動並列化コンパイラーでは、プログラム内でループとして形成される一連の処理における、その実行タスクとデータフローを解析します。実際には、プログラム中で利用される配列や変数がループ中でどのようにアクセスされるかを解析します。このデータアクセスのパターンにおいて、ループの実行時にそのデータ参照において競合が無い事を確認する依存性解析を行います。データアクセスのパターン解析では、プログラム内の配列や変数が、並列化した場合に各スレッドが「Private」として個々に利用するものであるか、「Shared」としてスレッド間で共有されるかを判定します。

また、自動並列化では、ループ処理のタスクで並列に実行できる部分と、逐次処理する部分も分離して、並列化処理のためのコードへの変換がなされます。実際には、スカラー最適化やループ最適化（レジスター内でのベクトル化やメモリー最適化）といった最適化機能と並列化機能の 2 つは緊密に統合されて、キャッシュの局所性を高め、スレッドレベル以外での並列性を効率的に利用しています。

自動並列化では、入れ子になった多重ループの場合、できるだけ外側のループを並列化の対象とするようになっており、並列処理のオーバーヘッドを最大限減らすことに寄与します。例えば、二重にネストされたループの場合、最内側ループについては、ベクトル化をはじめキャッシュの局所性を高め、レジスター利用の最適化を最大限に図り、計算回数とメモリー参照を最小化するような最適化の適用を目指し、外側ループでマルチスレッド・コードを生成するような最適化を行います。自動並列化での並列処理は、一般には、「細粒度」での並列処理となります。

例えば、次のようなプログラムの自動並列化を考えてみましょう。

```
for (i=1; i<100; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

このプログラムは、コンパイラーによって、自動的に次のようにスレッド毎のタスクに分割されます。

```
// Thread 1
for (i=1; i<50; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
// Thread 2
for (i=50; i<100; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

プログラムの開発者は、この例で示したようなループの分割やその分割したループのマルチスレッドでの実行の制御などについて意識する必要はありません。普通のループで必要な処理を記述すれば、コンパイラーが並列処理を行います。またこの例では、ループ回数は明示的に記述しましたが、実際にはループ回数はプログラムの実行時に初めて明らかになるケースがほとんどです。コンパイラーは、並列化したループの実行前に各スレッドが実行するループの反復回数を決定して実行します。また反復回数が少ない場合には、並列化処理を行わず逐次処理を行うような条件判断を設定することも可能です。

簡単なπの計算での自動並列化コンパイルを示します。πの値を計算するには、以下のようなプログラムが可能です。

```
 1  #define num_steps  1000000
 2  double step;
 3  main ()
 4  {  int i;   double x, pi, sum = 0.0;
 5
 6    step = 1.0/(double) num_steps;
 7
 8    for (i=1;i<= num_steps; i++){
 9            x = (i-0.5)*step;
10            sum = sum + 4.0/(1.0+x*x);
11    }
12    pi = step * sum;
13  }
```

このプログラムに対して、自動並列化コンパイルのためのコンパイル・オプションを指定してコンパイルすることで自動並列化コンパイルが可能です。(Linux* でのコンパイル例)

```
$ icc -parallel -par-report3 -par-threshold0 -O3 sample.c
   procedure: main
sample.c(9) : (col. 11) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 9
      shared     : { }
      private    : {"i", "x"}
      first priv.: {"step"}
      reductions : {"sum"}
```

ここでは、自動並列化に関するレポートを出力させているので、どのループで並列化されたかが出力されています。

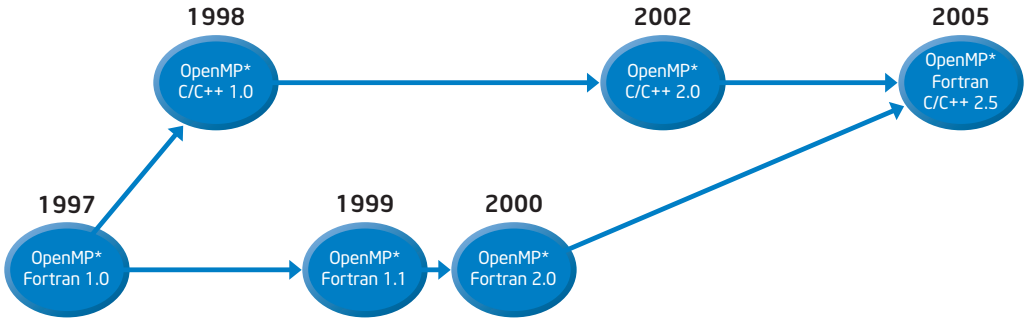
最初に示したように、自動並列化解析とは、コンパイラーによる並列実行可能なタスクの認識とそのタスク内でのデータの利用に関する解析です。コンパイラーは、プログラム内のループの構造をチェックし、その並列実行の可能性をチェックします。そのループを複数のタスクに分割できると判断した後で、並列処理が可能であるかどうかはそれらのループ内でのデータの依存性の判断になります。例えば、次のような例では、各ループの実行は、その前のループの反復計算の結果を必要としますので、依存性があることになります。従って、このようなループの自動並列化はできません。コンパイラーは、依存性解析を行った結果をメッセージとしても出力しますので、そのメッセージなどを参考に並列化のためのプログラム変更などの検討ができます。

```
$ cat -n sample.c
     1  #define N 1000
     2  main ()
     3  { int i;   double a[N], b[N], c[N];
     4    for (i=1;i<= N; i++){
     5            a[i] = a[i-1] + b[i] * c[i];
     6    }
     7  }
$ icc -parallel -par-report3 -par-threshold0 sample.c
   procedure: main
   serial loop: line 5
      flow data dependence from line 5 to line 5      flow data dependence from
stmt 2 to stmt 2, due to "a"
```

コンパイラーが生成するコードは、ハイレベルのマルチスレッド・ライブラリーを参照します。このマルチスレッド・ライブラリーは、この後で説明する OpenMP* でも共通に利用されます。これによって、自動並列化と OpenMP* の混在が可能となり、またオペレーティング・システムに関係なく、自動並列化の適用が可能になります。

5.2 OpenMP* : 共有メモリー並列プログラミング API

OpenMP* は、マルチスレッド並列プログラミングのための API (Application Programming Interface) です。OpenMP* は 1997 年に発表された業界標準規格であり、多くのハードウェアおよびソフトウェア・ベンダーが参加する非営利団体「OpenMP Architecture Review Board」によって管理されています。OpenMP* API は、Linux*、UNIX* および Windows* システムで利用可能です。OpenMP* は、C/C++ や Fortran* のようなコンパイラー言語ではなく、コンパイラーに対する並列処理の機能拡張を規定したものです。したがって、OpenMP* を利用するためには、「インテル® コンパイラー バージョン 9.0」のような OpenMP* をサポートするコンパイラーが必要になります。OpenMP* の詳細については、OpenMP* のホームページ http://www.openmp.org/ に詳細な情報があります。最新の OpenMP* のリリースは、2005 年 5 月に発表された OpenMP* 2.5 であり、この仕様で初めて C/C++ と Fortran* の双方の規格が統合されました。

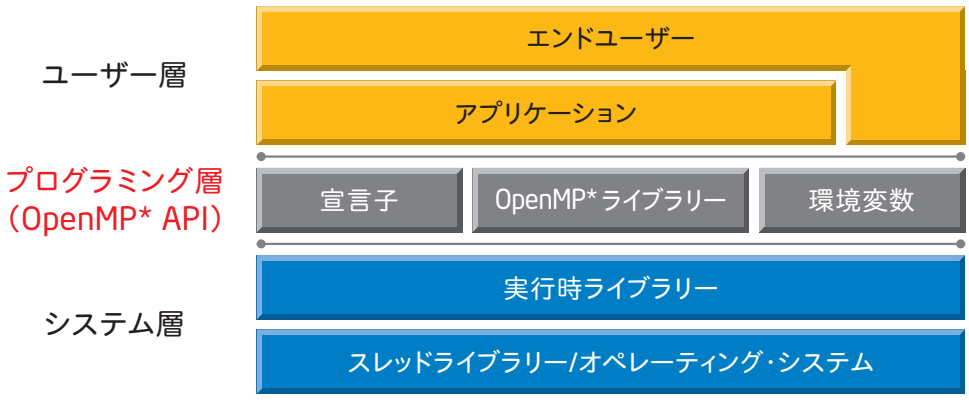


OpenMP* 開発とリリースの歴史

C/C++ や Fortran* には並列処理のための API が無かったために、それを補うものとして規定されたのが OpenMP* であるとも言えます。OpenMP* は C/C++ や Fortran* の言語規格に準拠しているため、OpenMP* を利用してもプログラムの移植性や互換性を損なうことなく、並列処理を容易に適用することができます。

プログラムの開発者は、コードの設計時から OpenMP* を利用した並列処理を実装することも、すでに開発されたプログラムを、OpenMP* を利用して段階的に並列化することも可能です。実際に OpenMP* での並列化を適用していて計算などが不正になった場合、簡単にその部分だけを逐次実行に切り替えることができますから、プログラムのデバッグが非常に容易です。さらに、先に説明した自動並列化と OpenMP* を併用したり、プログラムの一部だけを OpenMP* で並列化したりして、他の部分を自動並列化することもできます。

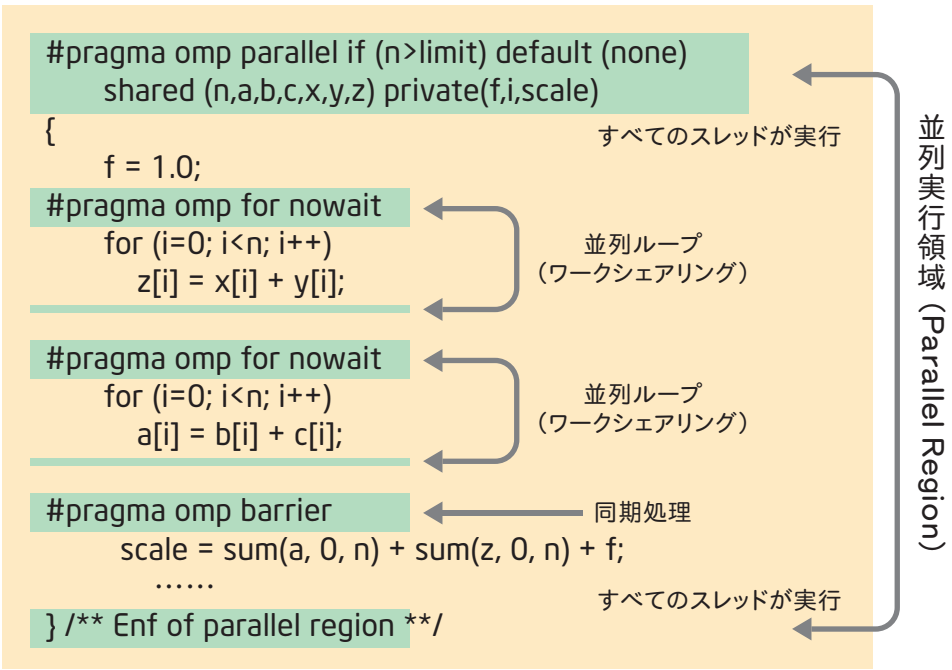
OpenMP* のアーキテクチャーは以下に示します。



OpenMP* アーキテクチャー

API はソフトウェアを開発する際に使用できる命令や関数群、あるいは、それらを利用するプログラム上の規約を総称したものです。OpenMP* API では、コンパイラー宣言子の規定と OpenMP* ライブラリー、および環境変数が API として規定されています。

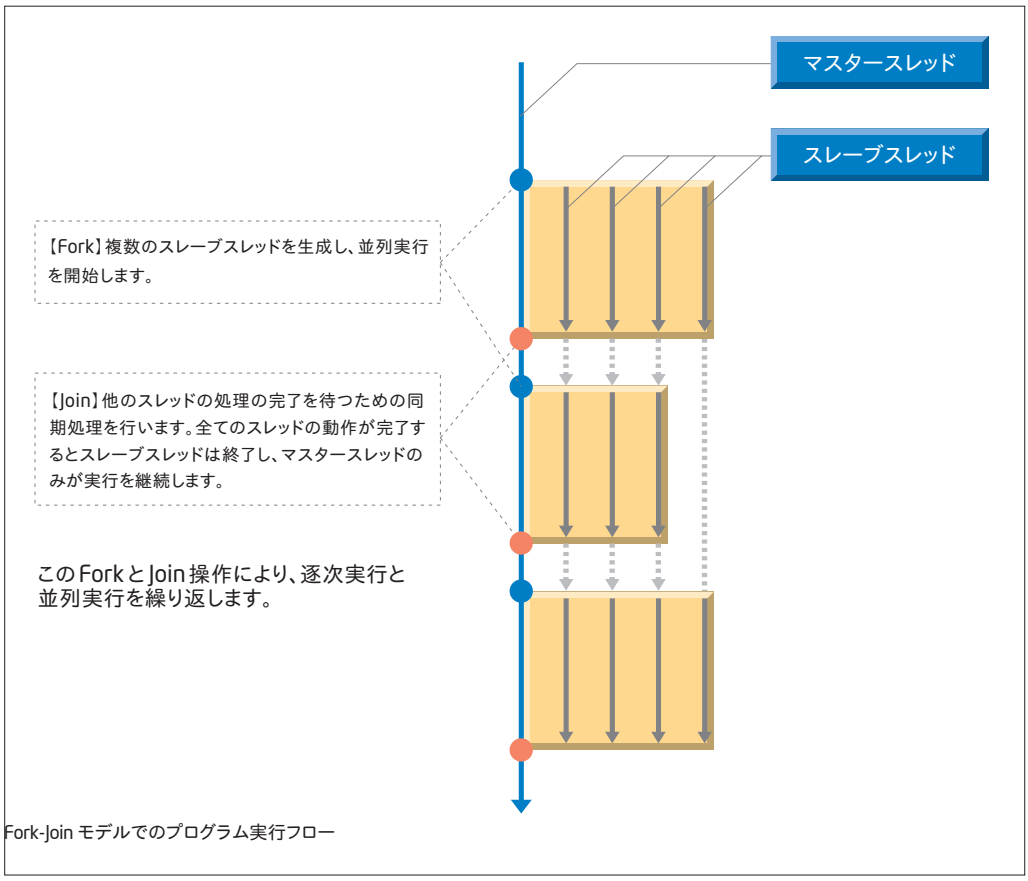
OpenMP* API は、プログラムの並列化領域や、データ属性などの宣言子と並列処理を補助するための OpenMP* ライブラリー、そして並列処理を行う場合の実行環境を指定する環境変数から構成されます。これらは、デスクトップ PC からスーパーコンピューターまで、統一された 1 つの API となっています。したがって、OpenMP* API でプログラムの並列化を行うことで、将来のシステムのスケールアップに合わせてプログラムを書き直す必要はありません。また .OpenMP* API をサポートしているプラットフォーム間であれば、将来のシステム変更の際のプログラムの移行は非常に容易になります。



OpenMP* によるプログラム構造

OpenMP* を利用した並列プログラミングは、プログラムに対して OpenMP* で規定された宣言子を挿入し、コンパイル時にコンパイル・オプションとして Windows* の場合 /Qopenmp スイッチを、また Linux* の場合は -openmp スイッチを指定することで可能になります。OpenMP* 宣言子は、コンパイラーに対して並列化のためのヒントを与えるのでなく、明示的に並列化を指示するという事に注意する必要があります。間違った宣言子を指定しても、コンパイラーはその指示に従って並列化を行います。また、データの依存性などがあっても、コンパイラーは警告メッセージやエラーメッセージを出し、その指示を無視することなく忠実に並列化を行いますので、依存性があるループなどを OpenMP* で並列化した場合には、計算結果が不正になります。一方、OpenMP* を使用した場合、スレッドの生成や各スレッドの同期コントロールといった制御については、ユーザーが意識する必要はありません。

OpenMP* では、一般に Fork-Join モデルと呼ばれる並列実行処理を行います。



Fork-Join モデルでのプログラム実行フロー

このプログラミング・モデルでは、逐次実行部分（シングルスレッドの実行）と並列実行（マルチプロセッサーやマルチコア・プロセッサー上でのマルチスレッドの実行）が交互に切り替わって実行されます。

1 OpenMP* の処理では、最初にマスタースレッドが起動されてプログラムの実行を開始します。このマスタースレッドは、逐次的にプログラムの処理を行います。

2 プログラムが OpenMP* での並列化宣言子　#pragma omp parallel (C/C++)、!$omp parallel (Fortran) の部分に到達すると、スレーブスレッドと呼ばれるスレッドを生成（スレッドを Fork する）し、分割されたプログラムのタスクを並列に処理します。

3 このマスタースレッドとスレーブスレッドの処理は、プログラム中での並列実行領域の終了を示す宣言子 #pragma omp end parallel (C/C++)、!$omp end parallel (Fortran) に到達すると終了します。この終了時には、全スレッドが各自の処理を終了するまで、先に終了したスレッドも待つことになり、同期処理を必要とします。

4 全スレッドが完了した時点（スレッドを join する）で、プログラムの実行処理は再びマスタースレッドだけが行うことになります。

このような処理を繰り返して、プログラムの実行を行うので、Fork-Join モデルと呼ばれています。

先ほどの自動並列化で示した、簡単な π の計算を例に、OpenMP* プログラムを示します。このプログラムに OpenMP* 宣言子を挿入し並列化した例は次のようになります。

```
 1  #define num_steps  1000000000
 2  double step;
 3  main ()
 4  {  int i;   double x, pi, sum = 0.0;
 5
 6    step = 1.0/(double) num_steps;
 7
 8  #pragma omp parallel for private(x) reduction(+:sum)
 9    for (i=1;i<= num_steps; i++){
10            x = (i-0.5)*step;
11            sum = sum + 4.0/(1.0+x*x);
12    }
13    pi = step * sum;
14    printf (" pi = %f \n",pi);
15  }
```

このプログラムに対して、OpenMP* での並列化をコンパイラーに指示することで、並列化が可能になります。Linux* でのコンパイル例と出力を示します。

```
$ icc -openmp –openmp-report2 -O3 sample1.c
sample1.c(8) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

OpenMP* での並列化では、自動並列化が難しい関数やサブルーチンの呼び出しを含むタスクでの並列化（疎粒度での並列化）が可能になります。コンパイラーの自動並列化では認識できないこのような粒度の大きな並列化では、OpenMP* を利用することによりオーバーヘッドの小さな並列化処理が可能となります。

次の例は OpenMP* ホームページ（http://www.openmp.org）に掲載されているサンプルプログラムです。ここでは並列実行領域内からサブルーチンを呼び出しています。このようなプログラムコードの粗粒度の並列性を最大限に活用することで高いスケーラビリティーの実現が可能となります。

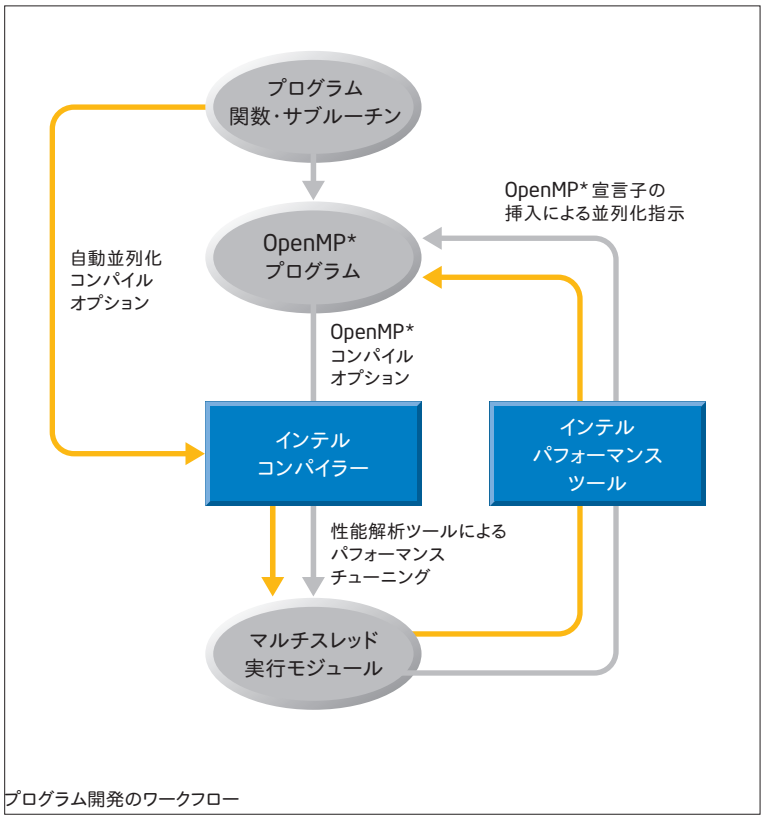
```
!$omp  parallel do
!$omp& default(shared)
!$omp& private(i,j,k,rij,d)
!$omp& reduction(+ : pot, kin)
       do i=1,np
        ! compute potential energy and forces
        f(1:nd,i) = 0.0
        do j=1,np
           if (i .ne. j) then
             call dist(nd,box,pos(1,i),pos(1,j),rij,d)
             ! attribute half of the potential energy to particle 'j'
             pot = pot + 0.5*v(d)
             do k=1,nd
               f(k,i) = f(k,i) - rij(k)*dv(d)/d
             enddo
           endif
        enddo
        ! compute kinetic energy
        kin = kin + dotr8(nd,vel(1,i),vel(1,i))
       enddo
!$omp  end parallel do
       kin = kin*0.5*mass
```

```
subroutine dist(nd,box,r1,r2,dr,d)
implicit none

integer i
      ....
      ....
d = 0.0
do i=1,nd
  dr(i) = r1(i) - r2(i)
 d = d + dr(i)**2.
enddo
d = sqrt(d)

return
end
```

http://www.specbench.org には SPEComp* という OpenMP* によるマルチスレッド・プログラムの標準ベンチマークの結果が掲載されています。このページに掲載されている OpenMP* を用いたベンチマーク・コードはいずれも高いスケーラビリティーを示しています。そのコードのほとんどは、単純なループレベルの並列化ではなく、より粒度の大きな並列度で並列化されています。

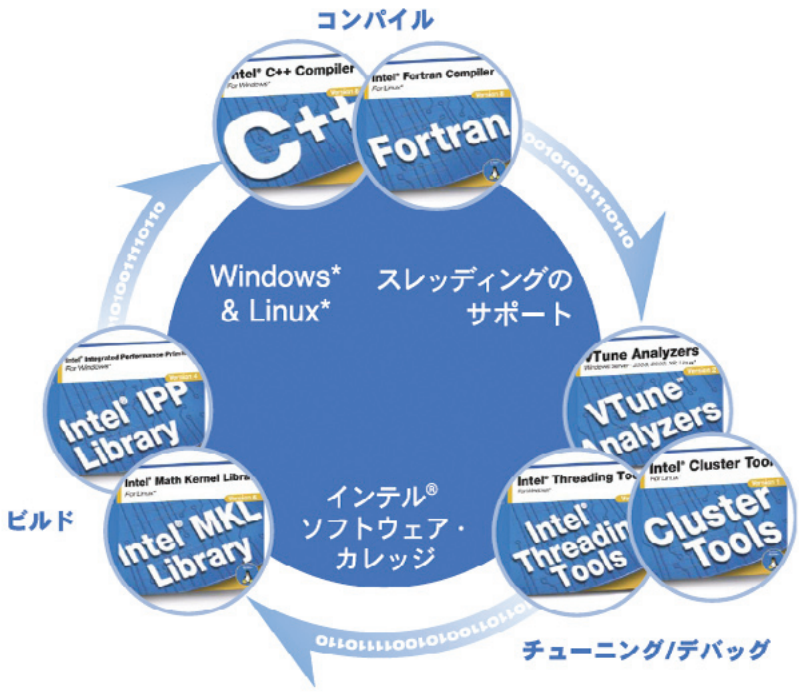
6. 開発環境

プログラムの開発には、多くの試行錯誤とプログラムの実行と検証、デバッグといった作業が必要になります。またプログラムについては、その実行性能の向上を図るためのソースコードの書き換えなどの作業を、プログラムの開発中も開発後も行う必要があります。このような作業を効率良く、また短時間で行うには、優れた開発環境が必要です。



プログラム開発のワークフロー

シングルスレッドのアプリケーションを OS が提供する API を使用してマルチスレッド化する場合、シングルスレッド環境では起こり得なかった問題が露呈することがあります。このような問題の多くはタイミングに依存し、毎回同じ条件で発生するとは限りません。そのため、マルチスレッド・プログラムの開発には、コンパイラー以外のツールの利用も不可欠です。開発者がアプリケーション開発のスピードを速め、その作業を単純化するためのツールが必要です。また、開発されたマルチスレッド・アプリケーションをより高速に実行するためのマルチスレッド最適化のためのツールも必要です。インテルのソフトウェア開発製品は、マルチスレッド・プログラミングに対応したインテル® コンパイラーでのプログラミングを支援する、豊富なツール群が用意されています。それらのツールを活用することで、プログラム開発サイクルを短縮し、信頼性を向上することができます。

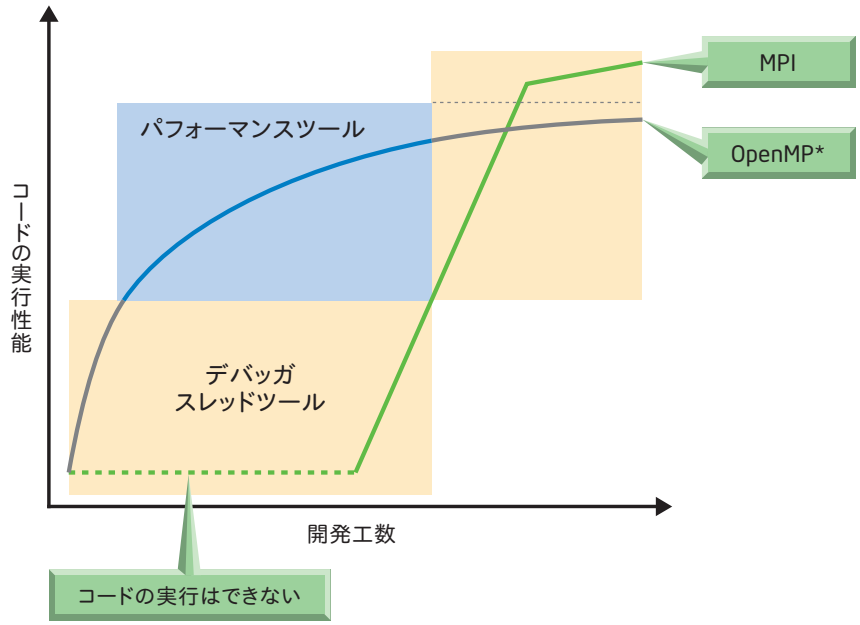


「Windows*/Linux* アプリケーションの高速パフォーマンスの追及」より

- インテル® スレッド・チェッカーは、データレースやデッドロックなど、不適切な同期処理などのスレッド間で不正にデータがアクセスされる問題を検知することが可能です。これにより、マルチスレッド・プログラムに関する問題点の識別をより容易にします。また、このツールは OpenMP* 宣言子にも対応しているため、OpenMP* を利用したマルチスレッド・プログラミングでの強力な解析ツールとなります。
- インテル® スレッド・プロファイラーは、スレッドのワークロードの不均衡を表示する新しいツールです。このツールを使用することで、OpenMP* によるマルチスレッド・アプリケーションのパフォーマンス・チューニングが効率よくできるようになります。スレッド・プロファイラーは、インテル® VTune™ アナライザーの機能として統合されています。

7. まとめ

「並列処理」と聞くと、何かプログラム上で特別なことを行っているような印象を受けるかもしれません。実際のところ、現在のシングルプロセッサー、シングルコアのマイクロプロセッサーでも、プロセッサー・コア内部で多くの並列処理が行われており、最新の高速マイクロプロセッサーでは、プロセッサー内に並列処理のための非常に多くのリソースを実装しています。これらのリソースを同時に利用することで、プログラムの高速実行を可能としています。インテル® Itanium® 2 プロセッサーなどはそのもっとも進んだ例です。コンパイラーは高度にプログラムを解析し、命令レベルでの並列化（ILP）を既に高度なレベルで実現しています。



MPI（Message Passing Interface）と OpenMP* でのプログラム開発工数

パフォーマンスに対する高度な要求に答える形で、プロセッサーは高速化の一途をたどってきました。しかし、現在プロセッサーとメモリーの性能格差が広がるにつれて、様々なアプリケーションにおいて、メモリー・レイテンシーがパフォーマンス面でのボトルネックになっています。またプロセッサーの消費電力と発熱量も大きな問題です。このような状況を打破するためにも、デュアルコアやマルチコアといった最新のプロセッサーの実装技術が求められています。自動並列化、OpenMP* でのプログラム処理の並列化は、このような時代の要求に応えるものです。コンパイラーの更なる進化によって、今後、これらの機能はさらに強化されていきます。

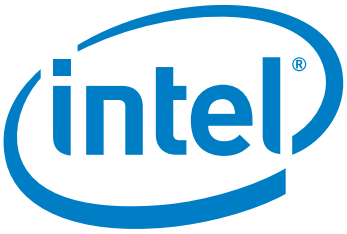
様々なレベルでの並列処理において、それらの並列処理技術を緊密に結合することで、アプリケーションの性能を大幅に向上させることが可能です。

執筆者プロフィール

スケーラブルシステムズ株式会社

代表取締役　　戸室　隆彦

約 20 年間にわたる HPC およびハイエンドコンピューティングの経験と、日本 SGI における CTO（チーフテクノロジーオフィサー）としての幅広い経験を基に、2005 年 6 月、HPC およびハイエンドコンピューティングに関する技術コンサルテーションを提供する「スケーラブルシステムズ株式会社」を設立。URL：http://www.sstc.co.jp/



分散並列処理、スーパーコンピューターなどに関する各種情報は、インテル HPC リソース・センターをご参照ください。http://www.intel.co.jp/jp/go/hpc/

インテル株式会社
〒 300-2635 茨城県つくば市東光台 5-6
http://www.intel.co.jp/

2006 年 3 月

525J-001
JPN/0603/PDF/SE/DEG/KS