WELCOME TO THE PARALLEL UNIVERSE

THINK PARALLEL or PERISH
マルチコア時代を生き抜くソフトウェア開発者とは
James Reinders、インテル® ソフトウェア開発製品部門チーフ・エバンジェリスト

安全な並列コードの記述
Peter Varhol、Technology Strategy Research, LLC 代表

Parallel Universe ～並列処理の世界～へ 足を踏み入れる準備はできていますか
マルチコア向けのアプリケーションの最適化
Levent Akyil、インテル® ソフトウェア開発製品部門ソフトウェア・エンジニア

マルチコア向け並列プログラミングの 8 つのルール

Software Products

マルチコア時代を生き抜くソフトウェア開発者のための特別冊子

Think Parallel or Perish

～マルチコア時代を生き抜くソフトウェア開発者とは～

James Reinders
インテル® ソフトウェア開発製品部門のチーフ・エバンジェリストである James Reinders は、将来、ソフトウェア開発者は誰もがプログラミングする際、最初に並列化を考えなければならなくなると予測しています。本冊子では 3 年前に『Think Parallel or Perish』を出版した当時の考えを再度取り上げ、ソフトウェア開発の「これまで」と「これから」を考察します。

3 年前、私はふとソフトウェア開発における自分の将来について考え、思い至ったことを書き留めました。「ソフトウェア開発者として、我々すべてが自ら “Think Parallel” (並列化を考える) しなければならない。さもなければ、並列化を考え、取り入れている他の開発者に取り残されて競争力を失うであろう」と。私はいまなお、2016 年のソフトウェア開発の世界では「Think Parallel」を習得していないプログラマーは厳しい状況に陥るとの予測に自信を持っています。

並列プログラムを記述する開発者がいれば、記述しない開発者もいるでしょう。また、新しい言語を使用して、従来とは全く異なる方法で記述する開発者も登場してくることでしょう。しかしながら、ほとんどの開発者は従来の並列プログラムの記述方法を踏襲するでしょう。

我々、すべての開発者がすべきことは、「Think Parallel」の習得です。コンピューターやアプリケーションなど我々を取り巻く環境において、並列化が現在どのように使用され、今後どのように発展していくのかを理解することは我々の将来において非常に重要なことです。この新しい世界では「良いもの」と「悪いもの」を直感的に把握していくことが、我々の目の前にある「Think Parallel」への挑戦です。

私のデスクにある間仕切りの壁には、今でも当時のメモ書きが貼られています。「自分へのメモ: 失業への近道、それは並列化を活用するプログラミング・スキルを磨かないことだ」と。

2009 年は、インテルがマルチコア・プロセッサーを出荷してから 5 年目の年です。この 5 年間は、単にマルチコア・プロセッサーが登場してから 5 年が経過したというだけではなく、サーバー、デスクトップ、そしてラップトップ・コンピューターに搭載されている x86 プロセッサーのマルチコア化が進んだ 5 年間です。この期間、いくつかのマルチコア・プロセッサーの登場により、シングルコア・プロセッサーがその役割を終えました。

そして 2009 年、我々はもう 1 つの記念すべき出来事を経験することになります。インテルが製品版メニーコアの発表を予定している年だからです。メニーコア・プロセッサーはマルチコア・プロセッサーとは異なり、プロセッサー、キャッシュ、メモリー、そして I/O を相互接続する洗練されたシステムを備えています。

この 16 コア程度のプロセッサーから必要とされるシステムにより、メニーコア・プロセッサーではマルチコア・プロセッサーよりも多くのコアを持つことが期待できます。今後数年の間に、我々はかつてないほど多数のコアが搭載されたメニーコア・プロセッサーを見ることになるでしょう。

優れた設計は、ツールからではなく、人である開発者から生まれるものである。

このメニーコア・プロセッサー時代の幕開けは重要ではありますが、2009 年はそれほど多くのプログラマーに影響を与えるものではないでしょう。しかしながら、このメニーコアの出現は、マルチコアが出現したとき以上に新しい並列化時代が到来したことを現実のものとして感じさせます。ハードウェアを桁外れに並列化できるという可能性には圧倒されるものがあります。今、その終わりを迎えようとしている「非並列化」時代は、人々が何百年後かにコンピューターの歴史を振り返るとき、非常に原始的に映るのかもしれません。世界中の物事が並列に機能しているように、コンピューター・プログラムも並列に動作すべき時代なのです。我々は並列化がほとんど行われていない時代に別れを告げようとしています。「そろそろ時期がきた。1 つずつ処理するなんて、もう時代遅れだ」と。

今後 10 年も満たないうちに、まず「Think Parallel」を意識しないプログラマーは、プログラマーではなくなるでしょう。

インテルの取り組み

並列化について数年前に執筆したとき、インテルはすでに OpenMP* 規格 (www.openmp.org) の長きに渡る推進者であり、MPI に対応した多くのツールをリリースしてきました。パフォーマンス解析ツールとデバッガーは、時代に先駆けて包括的に並列化に対応していました。また、並列プログラムのデッドロックや競合状態を発見する新興ツールもすでに提供していました。

その後、インテルでは OpenMP 規格のバージョン 3.0 へのアップデートを支援し、他に先駆けてコンパイラーでサポートしました。さらに、より移植性に優れた高性能な MPI ライブラリーを独自に発表しています。

新たにリリースしたインテル® スレッディング・ビルディング・ブロック (インテル® TBB) は、C++ プログラムにおける最も優れた、最もポピュラーな並列化の方法となりました。

そして先日、インテル® Parallel Studio の提供が開始されました。これまでの経験をすべて取り入れ、より直感的で信頼性の高い魅力的な製品となっています。これらの新しい製品が、インテルの既存製品に加わりました。インテルのツールは、並列化の使用を進める顧客をサポートするために向上と拡張を続けています。並列化サポートを拡張する一方で、インテルは従来の技術の陳腐化を強いることはしません。これは顧客からも喜ばれている点です。

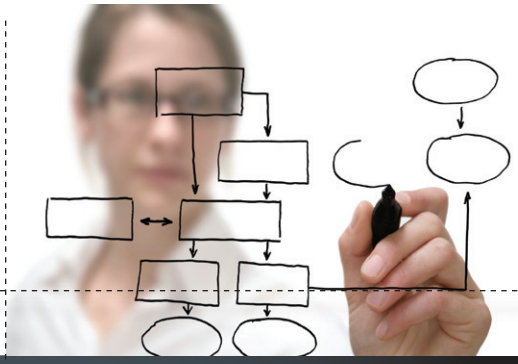
インテルの Ct テクノロジー (注: IDF 2009 で発表) については私自身も非常に楽しみにしており、ますますこのテクノロジーについての情報は増えていくことでしょう。このプロジェクトは、インテルが長年に渡って検証を続け、顧客と意見を交わしてきたプロジェクトの 1 つです。

現在インテルでは、並列プログラミングを支援するツールの開発に力を入れています。スケーラビリティー、正当性、保守性を達成するために、標準規格の採用、言語の拡張、ライブラリー API の使用を積極的に行っています。

これらの素晴らしいツールで何をするべきかを理解することは、ソフトウェア開発者である我々の役目です。並列化が始まる直前もそうであったように、優れた設計はツールからではなく人間である開発者から生まれるものです。並列化もまた同じです。我々が「Think Parallel」しなければならないのです。

James Reinders
オレゴン州ポートランド
2009 年 4 月

著者紹介: **James Reinders** は、Intel Corporation のソフトウェア開発製品部門のチーフ・エバンジェリスト兼ディレクターです。『*Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*』などの並列化に関する文献を発表しています。詳細は、www.go-parallel.com (英語) を参照してください。



並列化手法

インテル® Parallel Studio は、並列アプリケーション開発における 4 つのステップに対応します。

1. **インテル® Parallel Composer によるコーディングとデバッグ:** インテル® Parallel Composer のソースコード解析機能を使用して、コンパイル時にソースコードのエラーを特定します。ソースコード解析を有効にするには、コンパイラー・オプションとして /Qopenmp とともに /Qdiag-enable:sc-parallel{1|2|3} を使用する必要があります。オプション内の数字は、深刻度の診断レベルを指定します (1=すべての致命的なエラー、2=すべてのエラー、3=すべてのエラーと警告)。以下のシナリオでは、インテル® Parallel Composorがが並列アプリケーションのコーディングをサポートするさまざまなパターンを紹介しています。このようなシナリオは、他のケースでも同様の段階を踏んでいくでしょう。

シナリオ 1: OpenMP* 3.0 を使用して並列化を実装したい。

シナリオ 2: すでに使用している暗号化コードを並列化したい。インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) を使用して暗号化コードを置換したい。OpenMP を使用して並列化を実装したい。

シナリオ 3: 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) を追加したい。

シナリオ 4: すでに使用しているオーディオ/ビデオコードを並列化したい。適切なドメインにインテル® IPP を追加したい。インテル® TBB を追加して、その他の領域を並列化したい。

シナリオ 5: Windows スレッドによるコードを使用している。ラムダ関数をサポートするインテル® TBB を追加して、新しい領域を並列化したい。

OpenMP を使用して並列化を導入するシナリオでは、Microsoft* Visual Studio* 統合のインテル® Parallel Debugger Extension (http://software.intel.com/en-us/articles/preview/parallel-debugger-extension) を使用して OpenMP コードをデバッグできます。デバッガーには Microsoft Visual Studio の [デバッグ] メニューからアクセス可能な機能セットが用意されています。デバッガーでは、スレッドデータの共有違反についてのヒント、再入可能関数呼び出しの検出、SSE レジスターへのアクセス、タスク、タスク・ウェイト・リスト、タスク・スポーン・ツリー、バリア、ロック、スレッドチームなどの OpenMP データ構造へのアクセスなどが提供されます。インテル® Parallel Debugger でアプリケーションのデバッグを行うには、/debug:parallel オプションを使用して、インテル® C++ コンパイラーでビルドする必要があります。

3. **インテル® Parallel Amplifier によるチューニング:** インテル® Parallel Amplifier では、利用可能なプロセッサー・コアから最大限にパフォーマンスを引き出すことができます。インテル® Parallel Amplifier の並列処理分析により、プロセッサー・リソースを十分に活用していないコード部分を特定します。パフォーマンスを向上させる変更を行ったら、アプリケーションをインテル® Parallel Composer でリビルドします。そして、インテル® Parallel Inspector を使用して可能性のあるスレッド化エラーを確認し、インテル® Parallel Amplifier でパフォーマンスの向上度合いを確認します。

4. **コードを確認します。**

5. **ステップ 1 に戻り、並列化を追加する次の領域を特定します。**

インテル® Parallel Composer に付属する『Parallelizing N-Queens with the Intel® Parallel Composer (インテル® Parallel Composer による N クイーンの並列化)』チュートリアルでは、このドキュメントで説明されている各並列化手法を、N クイーン問題 (Eight Queens Puzzle (http://en.wikipedia.org/wiki/N-queens/) のより一般的な問題) に適用して、並列ソリューションを実装するハンズオン・トレーニングを提供します。チュートリアルは、インテル® Parallel Composer のメイン・インストール・フォルダーの “Samples\NQueens” フォルダー以下にインストールされます。

テクニカルサポート

インテルのオンライン・コミュニティー・サポートのユーザー・フォーラムとナレッジベースでは、インテルのツールや並列化の専門家、他の開発者から必要なヘルプを得られます。www.intel.com/software/support/ (英語) を参照してください。

安全な 並列コードの記述

マルチプロセッサー、マルチコア・プロセッサーをフルに活用するマルチスレッド・コードを記述することは簡単ではありません。新製品のインテル® Parallel Studio は、この問題の溝を埋める製品です。

Peter Varhol

マルチコア・プロセッサーの急増に伴い、コンピューターの処理能力を効率的に使用するという課題に対する重圧は、今やアプリケーション開発者に重くのしかかっています。アプリケーション開発において、並列化および独立した並列処理を実現できる箇所を特定し、コードに並列化手法を用いて実際に並列化を実装する必要性が高まってきています。

しかし、これは単純な作業でありません。並列処理中にデータとシステムリソースを保護する方法を理解するには、専門的な技術と細部への注意が必要です。競合状態やデータの上書きなどのエラーの発生率は非常に高いものです。

並列実行が可能なコード領域を特定することは、大変な課題です。コードに対する深い理解が必要なだけではなく、アプリケーション全体の実行フローも理解している必要があります。これは、現在のツールや技術では非常に難しいことです。

そのため、マルチプロセッサーやプロセッサー・コアを活用する並列コード作成に意欲を示す開発者や、技術的に準備ができている開発者はごく少数です。実際には、最新のサーバーや多くのワークステーション、ラップトップでマルチコア・プロセッサーが搭載されているにもかかわらず、この種のコーディングは、特殊でニッチだと考えられているのです。

しかしながら、マルチプロセッサー・システムやマルチコア・プロセッサーは消滅するわけではありません。事実、業界では今後数年の間にコア数が増加したプロセッサーが出現することでしょう。そして、数年のうちにデスクトップ・システムでさえも、16 コア以上のプロセッサーが搭載されることになるでしょう。商用ソフトウェアとカスタムソフトウェアの両方で、これらのプロセッサーの持つ処理能力を活用する必要があるのです。

そして、それは難題です。ただ、朗報はマルチコア・プロセッサーやマルチプロセッサー向けの並列化アプリケーションの開発に差をつけるツールが登場していることです。例えば、インテル® Parallel Studio は、マルチスレッド・コードでの作業に不可欠なツールが提供されています。開発者は、スレッドやメモリーのエラーを簡単に発見でき、また、ボトルネックの特定や分析を行うことができます。

この製品は、インテル® Parallel Composer、インテル® Parallel Inspector、インテル® Parallel Amplifier の 3 つのコンポーネントから構成されています。これらのツールを合わせて使用することで、並列実行を行うマルチスレッド・アプリケーションの開発において、今日のプロセッサーを効率良く活用するアプリケーションの構築とデバッグをすばやく行うことができます。

インテル® Parallel Composer は、並列コードのデバッグを単純化し、スレッドの正当性を確実にする並列デバッガープラグインが含まれています。また、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) とインテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) も含まれ、並列化をアプリケーションに簡単

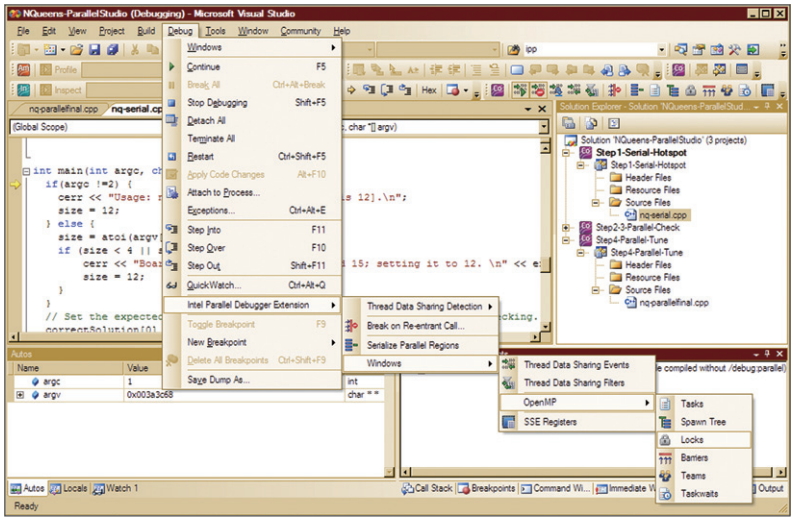


図 1: インテル® Parallel Composer には、並列アプリケーションの共有データやデータの依存関係について詳しい洞察やアクセスを提供するインテル® Parallel Debugger Extension が含まれています。Visual Studio の [デバッグ] メニューからアクセスできます。

に実装できる各種の並列化された汎用関数やアプリケーション固有の関数を提供します。

インテル® Parallel Inspector は、検出の困難なスレッド化エラーやメモリーエラーを検出し、アプリケーションの信頼性の確保に役立つ方向性を示します。インテル® Parallel Amplifier は、プロセッサー・アーキテクチャーやアセンブリー・コードの知識がなくても、マルチコア・パフォーマンスのボトルネックを簡単にすばやく発見できるパフォーマンス・プロファイラーです。

並列構造の課題

アプリケーション・コードが最新のマルチコア・プロセッサーを十分に活用していないことを認識する開発者は次第に増加するでしょう。多くの場合、そのようなコードはシングルコアでは高いパフォーマンスを達成できるものの、多大な労力を費やしてコード処理を並列にしなければ、利用可能なコアの使用に応じてスケーリングすることはできません。

並列コードとは、ほかのコードから独立して実行が可能なコードのことです。シングル・プロセッサー・システムやシングル・コア・システムでは、あるスレッドのプロセッサー・ダウンタイムを別のスレッドの実行に使用できるため、少なくともパフォーマンスが高速化したという幻想を抱くことはできます。マルチプロセッサーやマルチコア・システムでは、複数の個別の実行パイプラインを十分に活用することが必要不可欠です。

そして、それは簡単なことではありません。大抵のコードは、わかりやすい実行形式か、またはトップダウン形式で記述されています。いずれにしても、開発者には、並列化が可能な処理かどうかという見地からコードを検証する良い機会はなかなかありません。今日の一般的な開発プロセスでは、単純にそれを行うことができない状況です。

その代わり、開発者はアプリケーション実行の観点からコードを検証する必要があります。アプリケーションの実行と、ユーザーがアプリケーションを実行したら、どのようにソースコードが実行されるかというシーケンスを想像することができなければなりません。ここで、ほかに依存せずに実行できる処理こそが、並列化の候補となります。

しかし、この検証作業は、並列化工程の中でほんの最初のステップに過ぎません。開発者が並列に実行する実行パスを決定したら、それを実現しなければならないのです。それにはコードをスレッド化するだけでは不十分です。スレッドが開始され、独立して実行されたら、そのスレッドを保護しなければなりません。これには、コード内でクリティカル・セクションを設定しますが、該当セクションが実行されている間はそのセクションに関連する操作が実行されない (またはクリティカルの度合いによっては、何も実行されない) ように設定する必要があります。

しかし、クリティカル・セクションはリソースをロックする傾向があり、実行速度を減少させ、ときにデッドロックを発生させることがあります。リソースがロックされていることにより実行できない、あるスレッドが、そのロックしているスレッドで必要なリソースを保持している場合、それ以上の実行ができなくなります。

そのようなデッドロックは、複数のスレッドからの異なるリソースが関わっているため、リソースを特定し、適時にそのリソースをロックすることは、非常に細かな、エラーの発生しやすい作業になります。

スレッドのメモリーリークや同様のメモリーエラーもまた、並列実行において特定と診断が難しいものです。使用するメモリーが割り当てられても、自動で解放メモリーリストに戻らないため、意識しないと C や C++ コードでは非常に一般的に発生します。エラーは、あらゆる実行スレッドで発生する可能性があるため、非常に検出が難しく、通常、開発者やテスターが行う短時間のテスト中にはなかなか現れません。むしろ、時間とともにロストメモリーが蓄積され、ヒープが大きくなるにつれてアプリケーション実行速度が遅くなり、最終的にヒープが割り当てられたメモリーを超過するとプログラムのクラッシュを引き起こします。

競合状態

複数のスレッドが同時に実行され、どのスレッドが最初に実行を終了するかによって実行結果が異なるという状態は、競合状態として知られています。スレッドは実行中にデータをやりとりしますが、特定のスレッドが実行されるタイミングや、どの程度その実行が進んでいるかによって、そのデータは異なります。

もちろん、スレッドが独立して実行しているという仮定の下では、データのやりとりが発生しないことになっています。しかし、コード中に保護機能 (通常は排他機能やクリティカル・セクション) がなかったり、保護機能が不十分な場合は、クリティカル・セクション中でもスレッド間でデータのやりとりが発生してしまいます。

これは、スレッドの競合によって結果が異なるという、競合状態と呼ばれるものです。競合状態が最も潜行性が高く、またプログラミングでは発見しにくいエラーであるという理由の 1 つに、問題が常に発生するとは限らないという点があげられます。これは、コンピューターの実行はすべて確定している、という最も根本的なコンピューター実行の原理に反するものです。

現実には、常にエラーが発生ない限り、コード中のエラーを特定して分析することは不可能に近いことです。開発者は、何かがおかしいとわかっていても、問題の箇所を特定することができないため、競合状態が偶発的にしか検出できないと開発が頓挫してしまいます。スレッド実行を深く詳細に観察できるツールが、このような問題や、マルチスレッド・コードにおける非常にテクニカルで発見が困難な問題を特定できる唯一の方法と言えます。

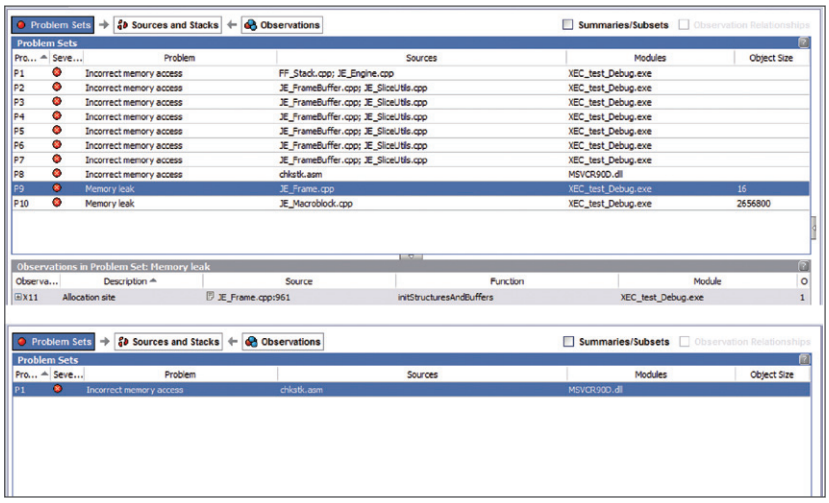


図 2: インテル® Parallel Studio に含まれるインテル® Parallel Inspector で、実行マルチスレッド・アプリケーション中のメモリーリークやメモリーエラーを特定することができます。

並列化のバグ発見をインテルが支援

インテル® Parallel Studio に含まれる 3 つのツールを使用すると、マルチプロセッサーやプロセッサー・コアをより良く活用できるマルチスレッド・アプリケーションをビルドすることができます。この製品は、Microsoft* Visual Studio* にプラグインされ、メモリー解析、スレッド解析、デバッグ、スレッド・ライブラリー、複雑なプログラムのパフォーマンス解析機能を提供します。

インテル® Parallel Inspector では、マルチスレッド・アプリケーションにおけるメモリー使用を分析できます。例えばメモリーが割り当てられても、処理の完了後にメモリーを解放しないなど、メモリーは、シングルスレッド・アプリケーションではよく誤用されるリソースです。マルチスレッド・プログラムでこのようなエラーを発見して修正することは、困難で非常に時間のかかる作業です。ツールを使用しなければ、妥当な時間でエラーを発見し、分析し、修正することは不可能です。

インテル® Parallel Inspector は、デッドロックなどのスレッド化エラーの検出や特定を支援し、また競合状態の発見にも役立ちます。これは、個々のスレッドの動作とスレッドがどのようにメモリーを使用し解放しているかを調査することによって行います。競合状態の場合、どのスレッドが最初に完了し、それが結果にどう影響するかを調べることができます。

インテル® Parallel Amplifier は、パフォーマンス・プロファイラーで、コードが最も時間を費やしている箇所をレポートします。さらに、プログラムやスレッドが待機している場所や、並列処理が十分ではない場所を検知します。つまり、アイドル状態のスレッド、プログラムでの待機、異常に長い時間アイドル状態にあるリソースを特定します。

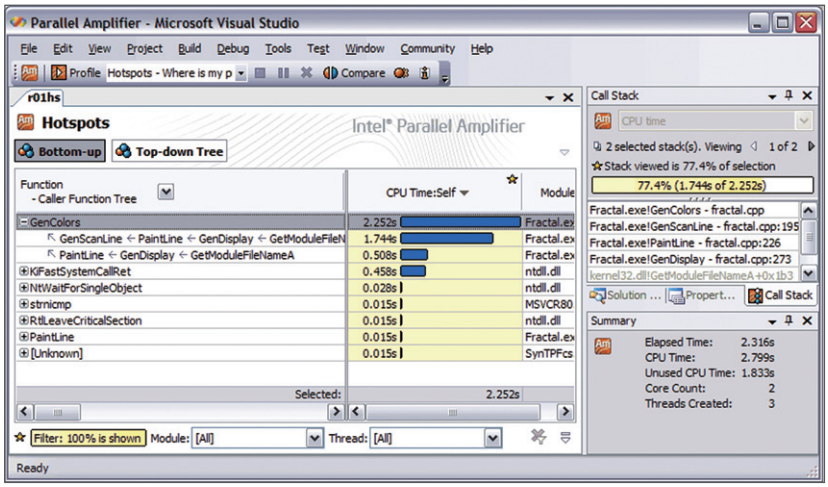


図 3: インテル® Parallel Amplifier は、マルチスレッド実行の観点から、開発アプリケーションのパフォーマンス分析を支援します。パフォーマンス・プロファイラーおよび、アイドル時間やボトルネックを特定する機能を提供します。

インテル® Parallel Composer は、パッケージの要とも言うべきコンポーネントです。並列ライブラリーとインテル® スレッディング・ビルディング・ブロックの C++ テンプレート・ライブラリーにより、開発者はスレッドをタスクに抽象化して、並列実行機能を備えたマルチスレッド・アプリケーションをより簡単に作成することができます。これらのライブラリーは、各種の並列化された汎用関数やアプリケーション固有の関数を提供し、アプリケーションに容易に並列化を追加で実装できるようになります。

インテル® Parallel Composer は、マルチスレッド・アプリケーション用に設計されたデバッガーのほか、コードをビルドする C++ コンパイラーも提供します。

これらのツールはいずれも、1 つのツールだけでは並列実行のマルチスレッド・コードの開発を幅広く支援するには不十分です。マルチスレッド・コードの記述は、並列実行用にデバッグされた既存のライブラリーがなくても行うことができます。マルチスレッド・メモリー・ツールがなくてもメモリーリークの発見は可能です。デッドロックや競合状態の分析も、スレッドの実行を観察する特別なツールがなくてもできるでしょう。しかし、マルチコア・プロセッサー上でハイパフォーマンスな処理を実行するように意図してマルチスレッド・コードを記述し、分析し、デバッグするには、簡単に並列関数を追加でき、それらの関数をデバッグし、デッドロックや競合状態を調査し、パフォーマンスを分析できるようなツールが必要です。

インテル® Parallel Studio は、これらすべての作業を Microsoft Visual Studio 内で行うソフトウェア・ツールです。インテル® Parallel Studio を使用することで、並列実行が求められる環境で使用するためのマルチスレッド・アプリケーションを迅速にビルド、分析、デバッグすることができます。マルチスレッド・アプリケーションで包括的に作業できる製品はほかにはありません。

将来、開発者は並列コードを記述する日が来るでしょう。そして、インテルは今、その作業に役立つツールを提供しているのです。これは、今まで私たちにもたらされていたコンピューターの処理能力を十分に活用し得るための唯一の方法です。開発者がコンピューターの処理能力を活用しなければ、インテルが提供する新しい革新的なテクノロジーの恩恵をユーザーは受けることはできないのです。

著者紹介：**Peter Varhol** 氏は、業界分析、コンサルティング会社の Technology Strategy Research, LLC 代表です。

Parallel Universe ～並列処理の世界～へ足を踏み入れる準備はできていますか：マルチコア向けの**アプリケーションの最適化**

「4 頭の馬に荷車を引かせることはできるが、1024 羽のにわとりに荷車を引かせる方法はわからない。」
—Enrico Clementi

Levent Akyil
Microsoft* Visual Studio* C/C++ を使用してWindows* アプリケーションを作成する開発者向けに設計された新しいインテル® Parallel Studio で実現可能な並列化手法について検証します。

はじめに

マルチコア・プロセッサーにより、消費者と開発者の両方にとって新しい時代が到来しました。消費者にさまざまな可能性が提供される一方で、開発者は、新しいマルチコア・プロセッサーの優れた処理能力を効率的に使用する製品を開発する必要性に迫られています。インテルは、開発に費やした莫大な投資を無駄にすることなく、急速に普及が進むマルチコア・システムを生産的に活用するツールをソフトウェア・コミュニティーに提供することを使命としています。この取り組みの一環として、インテルは Microsoft Visual Studio C/C++ を使用する開発者向けにインテル® Parallel Studio を発表し、幅広い分野のソフトウェア開発者に Parallel Universe への扉を開きました。インテル® Parallel Studio は、競争の激しいソフトウェア市場で高い地位が得られるアプリケーションの開発を支援するだけでなく、マルチコア向けの並列化を実現し、メニーコア・プロセッサー向けに優れたスケーリングを提供します。

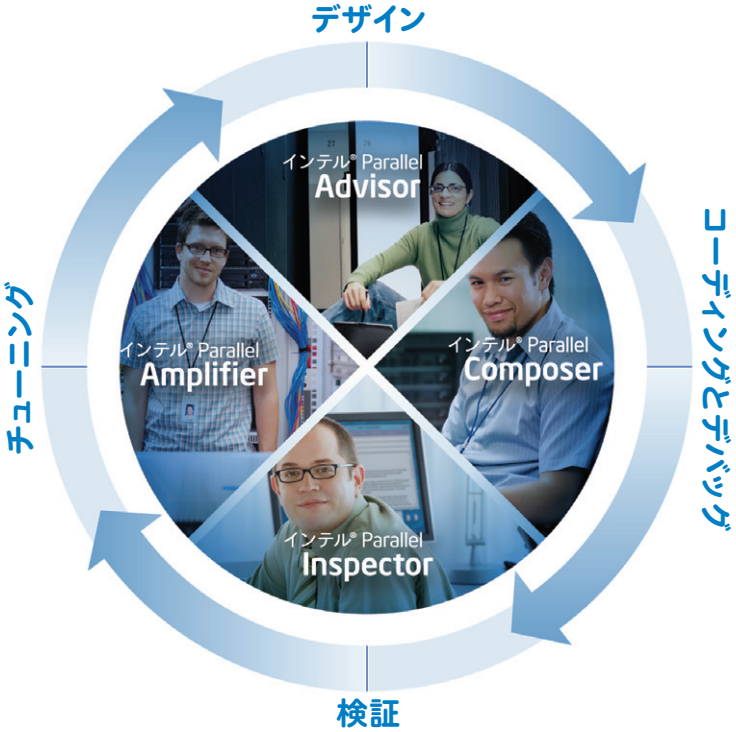


図 1: 並列ソフトウェア開発の 4 つのステップ

ここでは、インテル® Parallel Studio とソフトウェア開発の異なる側面に対応した主要なツール (図 1) について紹介します。インテル® Parallel Studio は、インテル® Parallel Advisor (設計)、インテル® Parallel Composer (コーディング/デバッグ)、インテル® Parallel Inspector (検証)、およびインテル® Parallel Amplifier (チューニング) で構成されています。

インテル® Parallel Composer では、コンパイラーとスレッド化ライブラリーによって、ソフトウェア開発における並列化の導入を迅速に

行うことができます。広範囲の並列プログラミング・モデルがサポートされているため、開発アプリケーションに最も適したコーディング方法に合致するモデルを見つけることができます。インテル® Parallel Inspector は、事前に「バグを発見」します。並列化プログラミング・モデルを問わず、アプリケーションの信頼性を向上する柔軟なツールです。従来のデバッガーとは異なり、このインテル® Parallel Inspector は、マルチスレッド化された C/C++ Windows アプリケーションで発見が困難なスレッド化エラーを検出し、データ競合やデッドロックのような不具合の元となる原因を分析します。インテル® Parallel Amplifier は、スケーリングを制限してしまう予期しない直列化の発見に役立ち、マルチコア・プロセッサー向けに最適なパフォーマンスを発揮する並列アプリケーションのためのきめ細かな調整を行います。

インテル® Parallel Composer

インテル® Parallel Composer を使用することで、開発者は並列処理を簡単に表現し、マルチコア・アーキテクチャーを活用することができます。インテル® Parallel Composer は、並列処理の迅速な導入を支援する並列プログラミング拡張を提供します。Microsoft Visual Studio 環境に統合され、OpenMP* 3.0、ラムダ関数、自動ベクトル化、自動並列化、スレッド化ライブラリーのサポートなど、アプリケーション・レベルで並列化を行うための機能が追加されます。受賞歴もあるインテル® スレッディング・ビルディング・ブロック (インテル® TBB) も、インテル® Parallel Composer の重要なコンポーネントの 1 つです。C/C++ において、使いやすく、ハイパフォーマンスで移植性の高い並列プログラミングを行う方法を提供します。

インテル® Parallel Composer により提供される並列化の主な拡張機能は次のとおりです。

■ **ベクトル化サポート:** インテル® コンパイラーのコンポーネントである自動ベクトライザーは、MMX® 命令、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE、SSE2、SSE3 および SSE4 ベクトル化コンパイラー命令およびメディア・アクセラレーター命令) とストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) を自動的に使用します。

■ **OpenMP 3.0 サポート:** インテル® コンパイラーは、開発者がソースプログラムで指定した OpenMP 宣言子に従ってコード変換を実行し、マルチスレッド・コードを生成します。インテル® コンパイラーは、現在の業界標準である OpenMP 宣言子のすべてに対応し、OpenMP 宣言子の注釈のある並列実行プログラムをコンパイルします。また、OpenMP バージョン 3.0 仕様に、ランタイム・ライブラリー・ルーチンおよび環境変数を含むインテル独自の拡張機能を提供します。/Qopenmp スイッチを使用すると、コンパイラーは OpenMP 宣言子に基づいてマルチスレッド・コードを生成します。生成されるコードは、単一プロセッサー・システムおよびマルチプロセッサー・システムの両方で並列実行が可能です。

■ **自動並列化機能:** インテル® コンパイラーの自動並列化機能は、入力プログラムの直列部分を同等のマルチスレッド・コードに自動的に変換します。自動並列化機能は、ワークシェアリング候補のループを特定し、正しい並列実行を確認するためにデータフロー解析を行います。また、OpenMP 宣言子のプログラミングに必要な場合には、スレッド化コード生成のデータをパーティショニングします。/Qparallel を使用すると、コンパイラーはアプリケーションの自動並列化を試みます。

■ **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):** 受賞歴もあるランタイムベースのプログラミング・モデルです。マルチコア・プロセッサーの潜在的なパフォーマンスを活用できるように支援する、テンプレート・ベースのランタイム・ライブラリーで構成されています。開発者は並行収集と並列アルゴリズムを活用して、スケーラブルなアプリケーションを作成することができます。

■ **単純並列関数:** 4 つのキーワード (__taskcomplete、__task、__par、および __critical) を文のプリフィックスとして使用して、簡単に並列プログラムを作成することができます。キーワードは、OpenMP ランタイムサポートを使用して実装されます。プログラムの並列化を細かく制御する場合は、OpenMP 機能を直接使用してください。この機能を使用する場合は、/Qopenmp コンパイラー・スイッチを指定して並列実行機能を有効にします。コンパイラー・ドライバーは、自動的に OpenMP ランタイム・サポート・ライブラリーにリンクします。実際の並列処理は、ランタイムシステムによって管理されます。

例

```
void sum (int length, int *a, int *b, int *c )
{
 int i;
 for (i=0; i<length; i++)
            c[i] = a[i] + b[i];
}

//並列関数を使用した場合
sum(1000, a, b, c);
```

```
// シリアル関数を使用した場合
__taskcomplete
{
 __task sum(500, a, b, c);
 __task sum(500, a+500, b+500, c+500);
}
```

■ **C++ ラムダ関数サポート:** C++ ラムダ式は、関数オブジェクトを定義する基本式です。このような式は、関数オブジェクトが想定される場所ではどこでも使用できます (例: Standard Template Library (STL) アルゴリズムの引数)。インテル® C++ コンパイラーは、ISO C++ ドキュメント (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/) (英語) で定められているラムダ式を実装しています。/Qstd=c++0x スイッチを使用すると、ラムダサポートが有効になります。

■ **インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP):** インテル® Parallel Composer に含まれており、マルチメディア、データ処理、通信アプリケーション向けに高度に最適化された、ソフトウェア関数の広範囲なマルチコア対応ライブラリーです。インテル® コンパイラーおよびパフォーマンス最適化ツールとともに使用できる、最適化されたソフトウェア・ビルディング・ブロックを提供します。インテル® IPP には、信号処理、オーディオ・コーディング、音声認識、音声コーディング、画像処理、ビデオ・コーディング、小行列の演算、3D データ処理などの分野でアプリケーションを作成するための基本的な低レベルの関数が用意されています。

■ **valarray:** ハイパフォーマンス・コンピューティング向けの配列演算を含む配列の C++ STL (標準テンプレート・ライブラリー) コンテナークラスです。これらの演算は、ベクトル化などのハードウェア機能を活用します。インテル® コンパイラーは valarray を組み込み型として認識し、インテル® IPP ライブラリーの呼び出しに置換します。

例

```
// 整数型の valarray を作成する
valarray_t::value_type ibuf[10] = {0,1,2,3,4,5,6,7,8,9};
valarray_t vi(ibuf, 10);

// ブール型の valarray のマスクを作成する
maskarray_t::value_type mbuf[10] = {1,0,1,1,1,0,0,1,1,0};
maskarray_t mask(mbuf,10);

// マスクされた配列の値を倍にする
vi[mask] += static_cast<valarray_t> (vi[mask]);
```

■ **インテル® Parallel Debugger Extension:** Microsoft Visual Studio に統合される、インテル® C++ コンパイラーの並列コード開発機能向けのアドオンデバッガーです。Microsoft Visual Studio のデバッグ機能を置換するのではなく、既存の機能を拡張します。

- **異なるスレッドからの同一データ要素へのアクセスを検出するスレッドデータ共有解析**
- **再入可能な関数呼び出しでプログラムの実行を停止するスマート・ブレークポイント機能**
- **OpenMP の並列ループで動的な追加作業スレッドの作成を有効/無効にするシリアル実行モード**
- **高度な OpenMP プログラム状態解析用の OpenMP ランタイム情報ビュー**
- **SIMD (Single Instruction Multiple Data) 命令セットを使用して並列データをデバッグするように、形式オプションと編集オプションが拡張されたストリーミング SIMD 拡張命令 (SSE) レジスタービュー**

このように、インテル® Parallel Debugger Extension は、スレッドデータ共有問題を特定するのに便利です。インテル® Parallel Debugger Extension は、ソース・インストルメンテーションを使用して、データ共有問題を特定します。この機能を有効にするには、[構成プロパティ] > [C/C++] > [General (全般)] > [Debug (デバッグ)] > [Enable Parallel Debug Checks (並列デバッグ検証を有効にする)] を選択して、/debug:parallel を設定します。図 2 は、インテル® Parallel Debugger Extension が、同じデータにアクセスする 2 つのスレッドを検出して、アプリケーションの実行を停止したことを示します。

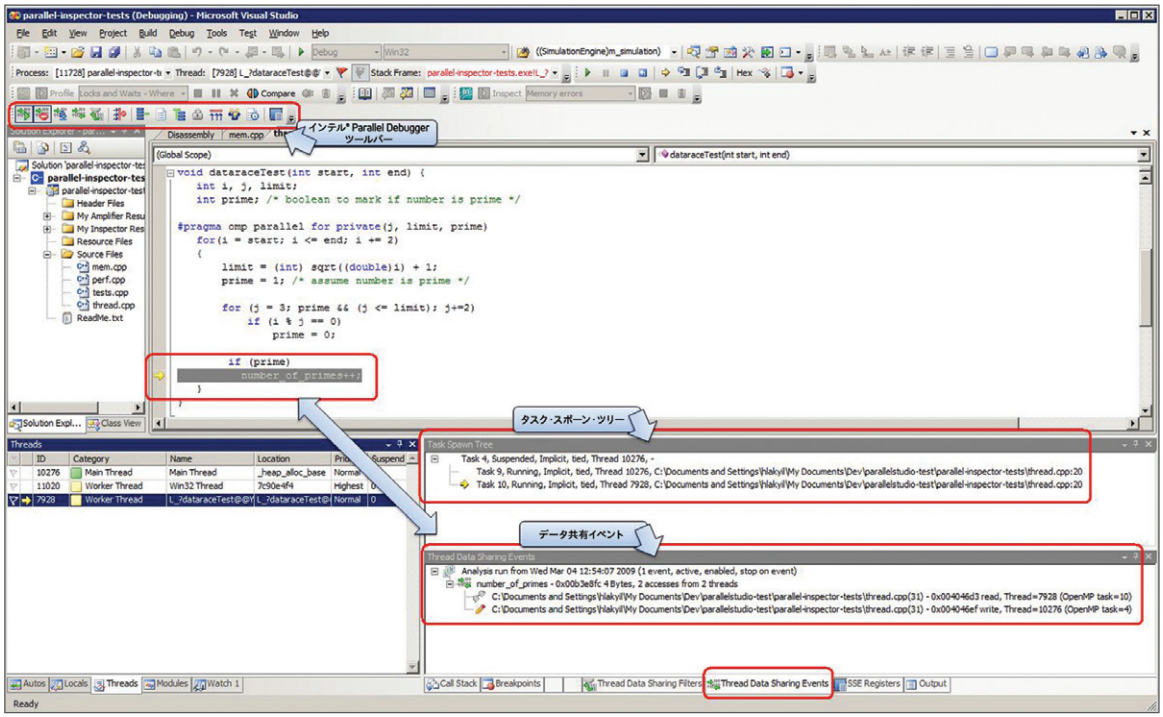


図 2: インテル® Parallel Debugger Extension はデータ共有問題を検出すると実行を中断することが可能

インテル® PARALLEL INSPECTOR

「この問題は、発生するまでその形跡すらありませんでした。非常に奥深くに埋もれていて、数百万行ものコードとデータを何週間もかかって詳細に調べ上げる必要がありました。」

—Ralph DiNicola
2003 年北米北東部大停電のアメリカ-カナダ合同調査委員会スポークスマン

マルチスレッド・アプリケーションでエラーの原因を特定することは困難です。インテル® Parallel Studio ツールの 1 つであるインテル® Parallel Inspector は、マルチスレッド・アプリケーションにおけるスレッド化エラーとメモリーエラーの元となる原因を分析し、事前にバグを発見します。

インテル® Parallel Inspector は、C および C++ アプリケーション開発者向けに、次の機能を提供しています。

- リーク、バッファー・オーバーラン・エラー、ポインターの問題など、メモリーとリソースに関するさまざまな問題を特定
- スレッド関連のデッドロック、データ競合、およびその他の同期化問題を予測し特定
- 並列アプリケーションの潜在的なセキュリティー問題の特定
- サイズ、頻度、種類順でエラーをすばやくソートし、重大な問題を認識して優先順位を設定

インテル® Parallel Inspector (図 3) は、Pin と呼ばれるバイナリー・インストルメンテーション・テクノロジーを使用して、メモリーエラーとスレッド化エラーを検証します。Pin は、インテルより提供されている動的なインストルメンテーション・システムです (http://www.pintool.org)。実行中の実行ファイルの任意の場所に C/C++ コードを挿入して、プログラムの動作を監視できます。

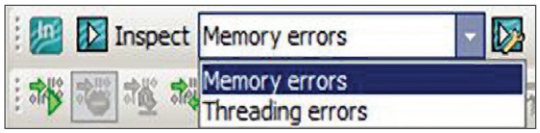


図 3: インテル® Parallel Inspector のツールバー

インテル® Parallel Inspector のメモリー分析レベル

インテル® Parallel Inspector は、Pin の設定を変更して、図 4 に示すように、設定とオーバーヘッドが異なる 4 つのレベルで分析を行うことができます。最初の 3 つの分析レベルはヒープ上で発生するメモリーエラーを分析し、4 つめのレベルはスタック上で発生するメモリーエラーも分析します。インテル® Parallel Inspector では、これらの分析レベルのサポートに、リーク検出テクノロジー (レベル 1) とメモリー・チェック・テクノロジー (レベル 2 ～ 4)を使用しています。また、これらのテクノロジーでは、さまざまな方法で Pin を使用しています。

レベル 1 レベル 1 分析は、アプリケーションのメモリーリークを見つけるのに役立ちます。メモリーリークは、メモリーブロックが割り当てられ、解放されない場合に発生します。

レベル 2 レベル 2 分析は、初期化されていないメモリーへのアクセス、不正な割り当て解除、割り当てと解除の不一致など、アプリケーションの不正なメモリーアクセスを検出します。不正なメモリーアクセスは、読み取り操作や書き込み操作が、論理的または物理的に不正なメモリーを参照した場合に発生します。このレベルでは、不正な部分メモリーアクセスも認識されます。不正な部分アクセスは、読み取り操作が、論理的に不正な部分を含むメモリーブロック (2 バイト以上) を参照した場合に発生します。

レベル 3 レベル 3 分析は、コールスタックの深さが 1 から 12 に変更され、ぶら下がりポインターの拡張チェックが有効であることを除き、レベル 2 に似ています。ぶら下がりポインターとは、すでに存在しないデータにアクセスしたり、そのようなデータを指定しているポインターです。インテル® Parallel Inspector は、割り当て解除を遅延して、メモリーが再割り当てされないように (別の割り当て要求によって返されないように) します。そのため、割り当て解除以降の参照は、ぶら下がりポインターによる不正な参照であることがわかります。この手法には追加のメモリーが必要で、遅延された割り当て解除リストに使用されるメモリーは制限されているため、インテル® Parallel Inspector は遅延した参照の割り当てをいずれ解除する必要があります。

レベル 4 レベル 4 分析は、すべてのメモリーエラーを見つけようとします。コールスタックの深さは 32 に変更され、ぶら下がりポインターの拡張チェック、システム・ライブラリーの分析、およびスタック上で発生するメモリーエラーの分析も行います。スタック分析は、このレベルでのみ有効になります。

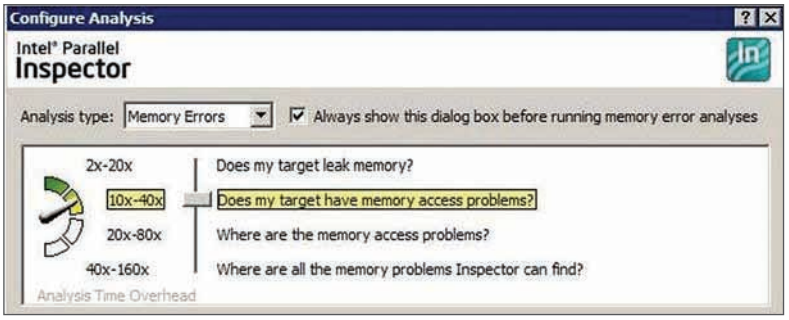


図 4: メモリーエラーの分析レベル

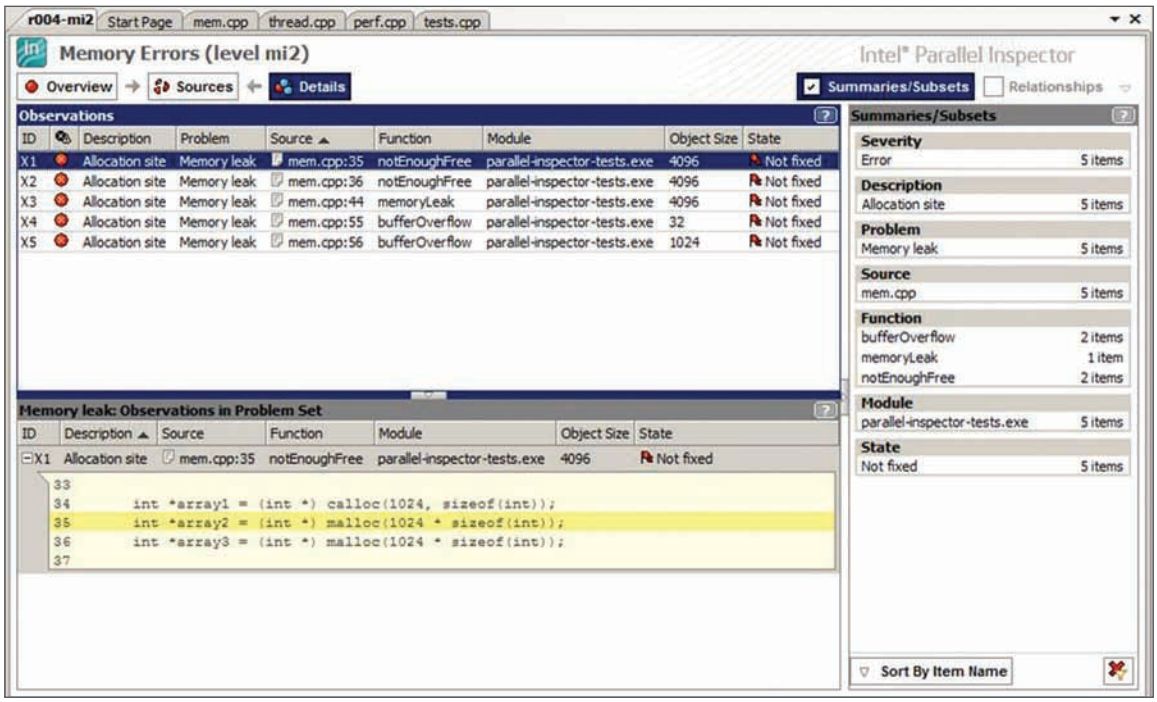


図 5: 検出されたメモリーエラーを示すインテル® Parallel Inspector の分析結果

図 5 に示すように、重大度、説明、原因、関数名、モジュールで結果をフィルターできます。

インテル® Parallel Inspector のスレッド化エラー分析レベル

インテル® Parallel Inspector には、スレッド化エラーの分析レベルも 4 つあります (図 6)。インテル® Parallel Inspector で認識される主なスレッド化エラーには、データ競合、デッドロック、ロック階層違反、および潜在的なプライバシーの侵害があります。

レベル 1 レベル 1 分析は、アプリケーションのデッドロックを見つけるのに役立ちます。デッドロックは、2 つ以上のスレッドが、他のスレッドがリソース (MUTEX、クリティカル・セクション、スレッドハンドルなど) を解放するのを待機しているのに、どのスレッドもリソースを解放しない場合に発生します。この場合、どのスレッドも先に進むことができません。コールスタックの深さは 1 に設定されます。

レベル 2 レベル 2 分析は、アプリケーションのデータ競合とデッドロックを検出します。データ競合は、最も一般的なスレッド化エラーの 1 つで、適切な同期処理が行われずに複数のスレッドが同じメモリー位置にアクセスした場合に発生します。このレベルでも、コールスタックの深さは 1 に設定されます。このレベルのバイト・レベルの粒度は 4 です。

レベル 3 レベル 3 分析は、レベル 2 のデータ競合とデッドロックを検出に加えて、それらの発生場所も検出します。より詳細な分析を行うために、コールスタックの深さは 12 に設定されます。このレベルのバイト・レベルの粒度は 1 です。

レベル 4 レベル 4 分析は、すべてのスレッド化エラーを見つけようとします。コールスタックの深さは 32 に変更され、スタック上で発生するエラーの分析も行います。スタック分析は、このレベルでのみ有効になります。このレベルのバイト・レベルの粒度は 1 です。

データ競合はさまざまな場合に発生します。インテル® Parallel Inspector は、write-write (書き込み-書き込み)、read-write (読み取り-書き込み)、および write-read (書き込み-読み取り) データ競合を検出します。

- write-write データ競合は、2 つ以上のスレッドが同じメモリー位置に書き込みを行う場合に発生します。
- read-write データ競合は、1 つのスレッドがメモリー位置から読み取りを行う際に、別のスレッドが同時に同じメモリー位置に書き込みを行う場合に発生します。
- write-read データ競合は、1 つのスレッドがメモリー位置に書き込みを行う際に、別のスレッドが同時に同じメモリー位置から読み取りを行う場合に発生します。

すべての場合において、実行順序が共有データに影響を与えます。

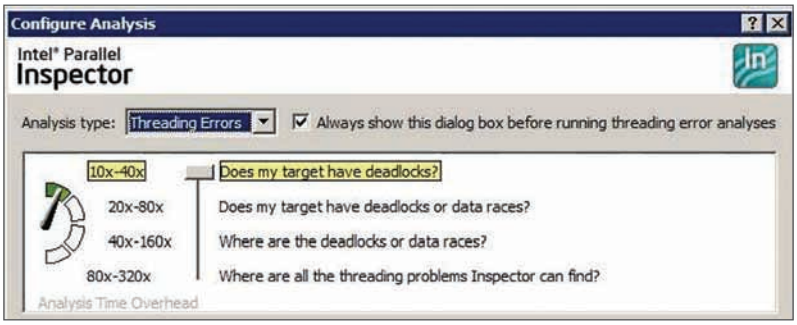


図 6: スレッド化エラーの分析レベル

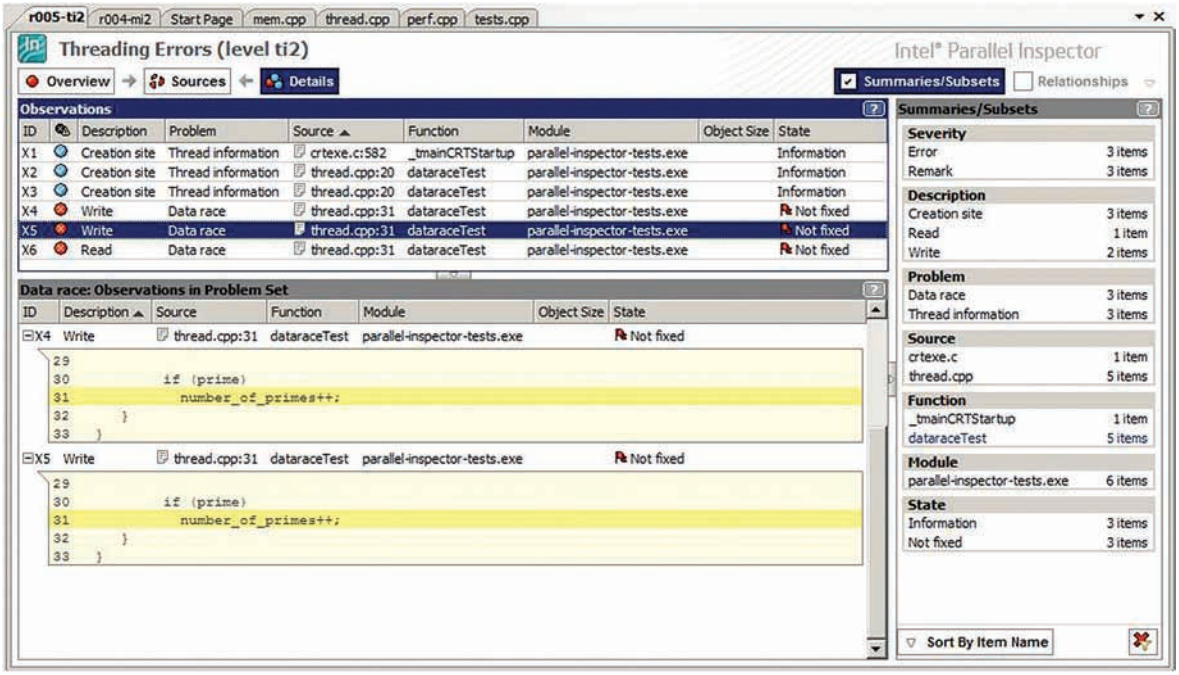


図 7: 検出されたデータ競合問題を示すインテル® Parallel Inspector の分析結果

インテル® PARALLEL AMPLIFIER

「プログラマーは、プログラムの重要ではない部分の速度について膨大な時間を無駄にしています。約 97 %の場合において、小さな効率は無視すべきです。中途半端な最適化は諸悪の根源と言えます。」

—*Donald Knuth*（出典：C. A. R. Hoare）

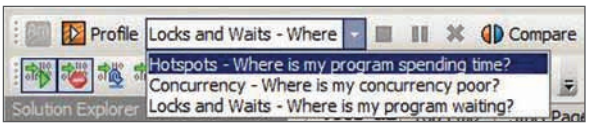


図 8: インテル® Parallel Amplifier のツールバー

マルチスレッド・アプリケーションは、並列化による複雑性のため、固有の問題セットを持つ傾向にあります。シリアルコードをスレッドセーフなコードに変換することは簡単ではありません。通常は、開発期間に影響を与え、既存のシリアル・アプリケーションよりも複雑になります。マルチスレッド・アプリケーション特有のパフォーマンス問題は、次のように要約できます。

- 複雑性（データ再構成、同期化の使用）
- パフォーマンス（最適化とチューニングの必要性）
- 同期オーバーヘッド

Knuth 氏のアドバイスに従って、インテル® Parallel Amplifier（図 8）は、開発者が投資収益率（ROI）の最も高い最適化を実現できるように、コードのボトルネックを特定するのを支援します。アプリケーションのパフォーマンス問題を特定し、適切に排除することは、効率的な最適化を行う上で重要です。

マウスをシングルクリックするだけで、インテル® Parallel Amplifier は、hotspot 分析、並列性分析、およびロックと待機の分析という 3 種類の強力なパフォーマンス解析を行うことができます。各分析レベルについて説明する前に、インテル® Parallel Amplifier で使用する指標について理解しておいたほうが良いでしょう。

経過時間　アプリケーションの実行時間です。一定の作業の実行において、アプリケーションの経過時間を減らすことは、重要な目的の 1 つです。アプリケーションの経過時間は、サマリービュー（図 11）に表示されます。

CPU 時間　プロセッサーでスレッドが実行に消費した時間です。複数のスレッドの場合は、すべてのスレッドの CPU 時間が合計されます。合計 CPU 時間とは、分析中に実行されたすべてのスレッドの CPU 時間の合計です。

待機時間　指定されたスレッドが、同期待機や I/O 待機などのイベントが発生するまで待機に要した時間です。

hotspot 分析

オーバーヘッドの低い統計サンプリング (スタック・サンプリングとも呼ばれる) アルゴリズムを使用することで、hotspot 分析は、開発者がアプリケーション・フローを理解し、実行に時間のかかるコードセクション (hotspot) を特定できるようにします。hotspot 分析中、インテル® Parallel Amplifier は、OS タイマーを使用して一定の間隔でサンプリングを行い、アプリケーションをプロファイルします。すべてのアクティブな命令アドレスとその呼び出しシーケンスのサンプルが収集されます。そして、格納されたサンプリング結果の命令ポインター (IP) と関連する呼び出しシーケンスを分析し、表示します。統計的に収集された IP サンプルと呼び出しシーケンスを使用して、インテル® Parallel Amplifier はコールツリーを生成し、表示します。

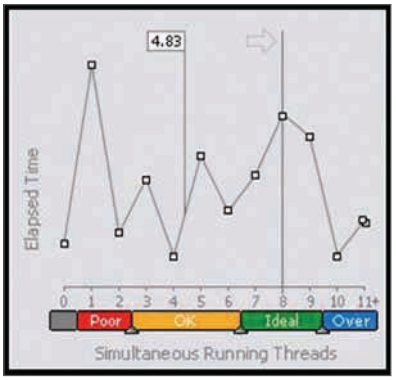


図 11: インテル® Parallel Amplifier の並列分析結果 - サマリービュー

並列性分析

並列性分析は、アプリケーションがシステムで利用可能なプロセッサーをどれだけ活用しているかを測定します。図 10 に示すように、プロセッサーが有効に利用されていない hotspot 関数を特定するのに役立ちます。並列性分析中、インテル® Parallel Amplifier は、アクティブなスレッド数を収集して、提供します。アクティブなスレッドとは、実行中のスレッド、または定義されている待機 API やブロッキング API により待機中ではないキューイングされているスレッドです。実行中のスレッド数は、アプリケーションの並列レベルに対応します。並列レベルとプロセッサー数を比較することで、インテル® Parallel Amplifier は、アプリケーションによるシステムで利用可能なプロセッサーの利用状況を分類します。

[Concurrency (並列性)] ウィンドウと [Locks and Waits (ロックと待機)] ウィンドウの時間は、次の利用状況を示します (図 11)。

Idle: プログラム中のすべてのスレッドが待機中です。実行中のスレッドはありません。[Summary (サマリー)] タブのグラフには、アイドリング中であることを示す 1 つのノードだけが表示されます。

Poor: 十分に利用されていません。デフォルトでは、スレッド数が並列化ターゲットの 50% 以下の場合です。

OK: 許容可能範囲で利用されています。デフォルトでは、スレッド数が並列化ターゲットの 51 ～ 85% の場合です。

Ideal: 理想的に有効利用されています。デフォルトでは、スレッド数が並列化ターゲットの 86 ～ 115% の場合です。

Hotspots: Bottom-up | Hotspots: Top-down Tree

Function - Bottom-up Tree	Module	CPU Time
ompsched2omp1	parallel-inspector-tests.exe	3.078s
ompsched3	parallel-inspector-tests.exe	2.874s
ompsched1omp1	parallel-inspector-tests.exe	2.874s
multiply_d	parallel-inspector-tests.exe	2.016s
dgemm	parallel-inspector-tests.exe	1.936s
multiply_b	parallel-inspector-tests.exe	0.516s
ompsched1	parallel-inspector-tests.exe	0.469s

図 9: インテル® Parallel Amplifier の hotspot 分析結果

Concurrency: Bottom-up | Concurrency: Top-down Tree

Function - Thread - Bottom-up Tree	Module	CPU Time by Utilization (Poor / Ok / Ideal / Over)
ompsched1	parallel-ins ...	5.004s
mainCRTStartup (1)	parallel-ins ...	3.187s
OMP Worker Thread #1 (3)	parallel-ins ...	1.817s
multiply_b	parallel-ins ...	4.937s
multiply_d	parallel-ins ...	4.266s
dgemm	parallel-ins ...	3.156s
ompsched2	parallel-ins ...	3.028s
ompsched3	parallel-ins ...	2.782s

図 10: インテル® Parallel Amplifier の並列性分析結果 (粒度が Function-Thread の場合)

ロックと待機の分析

並列性分析は、アプリケーションの並列化されていない場所や利用可能なプロセッサーを十分に活用していない場所を特定し、ロックと待機の分析は、プロセッサーが有効に利用されていない原因を特定するのに役立ちます。プロセッサーが有効に利用されていない最も一般的な原因は、スレッドが同期オブジェクト (ロック) を長時間待機しているためです。多くの場合、この間に有効な作業が行われないため、パフォーマンスが低下し、プロセッサーが有効に使用されません。

ロックと待機の分析中、開発者は、各同期オブジェクトの影響を予測することができます。分析結果は、各同期オブジェクトまたはスリープやブロッキング I/O などのブロッキング API における、アプリケーションの待機時間を理解するのに役立ちます。

Locks and Waits: Bottom-up | Locks and Waits: Top-down Tree

Sync Object Name - Wait Thread - Wait Function - Bottom-up Tree	Sync Object Type	Creation Source File	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count
Thread 0x63941c14	Thread	N_ThreadEvent1.cpp	28.621ms	1
mainCRTStartup (1)	Thread	N_ThreadEvent1.cpp	28.621ms	1
Semaphore 0xfbc2360c	Semaphore	N_ThreadEvent1.cpp	17.011ms	308
threadProc (3)	Semaphore	N_ThreadEvent1.cpp	8.998ms	156
threadProc (2)	Semaphore	N_ThreadEvent1.cpp	8.013ms	152
Stream 0x2a975393	Stream	ioinit.c	0.206ms	4
mainCRTStartup (1)	Stream	ioinit.c	0.206ms	4

図 12: ロックと待機の分析結果

分析結果を基に、同期オブジェクトを mutex (相互排他オブジェクト)、セマフォー、クリティカル・セクションおよび fork-join 処理にすることができます。ほとんどの場合、待機時間が最も長く、並列レベルの高い同期オブジェクトがアプリケーションのボトルネックになっています。

インテル® Parallel Inspector とインテル® Parallel Amplifier は、両方ともソース・コード・レベルまでドリルダウンすることができます。これは非常に重要なことです。例えば、図 13 の行をダブルクリックして、ソースコードにドリルダウンし、問題を引き起こしている同期オブジェクトを確認できます。

まとめ

並列プログラミングは新しいものではありません。長年の間、ハイパフォーマンス・コンピューティング (HPC) の分野でよく研究され、使用されてきました。今日、マルチコア・プロセッサーの普及に伴い、並列プログラミングは主流となりつつあります。今こそ、まさにインテル® Parallel Studio の出番と言えるでしょう。インテル® Parallel Studio はソフトウェア開発者が並列化プログラミングへ挑戦する機会をもたらすツールであり、並列化というパラダイムシフトをサポートすることにより、ソフトウェア開発者が並列処理の世界に足を踏み入れる際の障壁を劇的に減らします。PARALLEL UNIVERSE へようこそ。

著者紹介：**Levent Akyil** は、Intel Corporation のソフトウェア開発製品部パフォーマンス、分析、スレッド化研究室のソフトウェア・エンジニアです。

Line	Source	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count
26	`return numerator/n;`		
27	`}`		
28			
29	`DWORD WINAPI threadProc(LPVOID par)`		
30	`{`		
31	`int threadIndex = *((int *)par);`		
32	`for(int i=threadIndex; i<SERIES_MEMBER_COUNT;i+=NUMTHREADS) {`		
33			
34	`WaitForSingleObject(hSem1, INFINITE); // access to input`	17.011ms	308
35	`res += getMember(i+1, x);`		
36	`ReleaseSemaphore(hSem1, 1, NULL);`		
37			
38	`}`		
39			
40	`delete par;`		
41	`return 0;`		
42	`}`		
43			
	Total Selected:	17.011ms	308

図 13: ロックと待機の分析のソース・コード・ビュー

マルチコア向け 並列プログラミングの 8 つのルール

James Reinders

並列化の課題の解決とマルチコアの潜在能力を引き出すのに役立つ一貫したルールを紹介します。

マルチコア・プロセッサー向けのプログラミングは新しい挑戦です。ここでは、マルチコア・プログラミングで成功するための 8 つのルールを紹介します。

1 Think Parallel (並列化を考える)。並列処理を意識しながら、すべての問題にアプローチします。並列化の場所を理解し、それを適用する考えを整理します。その他の設計や実装の決定を下す前に、最も良い並列アプローチを決定します。「Think Parallel」並列処理の思考を身につけます。

2 抽象化を使用してプログラミングする。並列化を表現するコードの記述に重点を置き、スレッドやプロセッサー・コアを管理するコードの記述を避けます。ライブラリー、OpenMP、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) などは、すべて抽象化を使用している例です。raw ネイティブスレッド (pthreads、Windows スレッド、Boost スレッドなど) は使用しないようにしてください。スレッドと MPI は、並列化のアセンブリー言語です。これらは最も柔軟性がありますが、記述、デバッグ、保守に多大な時間が必要です。スレッド管理やコア管理を行うプログラミングではなく、問題に対応するプログラミングを行えるよう、コードは十分にレベルの高いものである必要があります。

3 スレッド (core: コア) ではなく、タスク (chore: 仕事) をプログラミングする。スレッドやプロセッサー・コアへのタスクのマッピングは、プログラムの中で明確に分離された処理にしておきます。使用する抽象化は、プログラマーの代わりにスレッド/コア管理を行ってくれるものであることが望ましいでしょう。プログラムには多くのタスク、またはプロセッサー・コア間で自動で処理されるタスク (OpenMP ループなど) を作成してください。タスクを作成することで、オーバーサブスクリプションを心配することなく、自由に可能な限り多くの処理を作成できます。

4 並列処理をオフにするオプションを付けて設計する。デバッグ作業を簡単にするには、並列処理を行わずに実行できるプログラムを作成します。デバッグの際に、最初は並列処理をオンにて実行し、その後オフにすることで、両方の実行で失敗するかどうかを調査できます。プログラムが並列実行されていない場合、一般的な問題は診断がしやすく、従来のツールでもサポートされているため、デバッグが簡単です。並列実行時のみ不具合が生じることがわかれば、追跡中の不具合がどのような種類のものかを特定する手がかりになります。このルールを無視して、シングルスレッドで実行できないプログラムを作成した場合、デバッグ作業に過度の時間を費やすことになります。シングルスレッドの実行はデバッグ用にのみ必要なため、効率的である必要はありません。「Producer-Consumer (生産者-消費者)」モデルなど、並行処理が正しく動作しなければならない並列プログラムの作成を避ければ良いだけです。MPI プログラムは、しばしばこのルールに反しています。これは、MPI プログラムが、実装とデバッグに問題がある理由の 1 つです。

5 ロックの使用を避ける。ロックは極力使用しないでください。ロックはプログラムの実行速度を低下させ、スケーラビリティーを減少させ、並列プログラム中の不具合の原因になります。問題の解決には、暗黙的な同期化を使用します。明示的な同期化が必要な場合は、アトミック操作を使用します。ロックは、最終手段としてのみ使用します。ロックの必要性を完全に排除できるようなプログラム設計をしてください。

6 並列化に役立つツールとライブラリーを使用する。古いツールで「頑張らない」ようにします。並列化をどのように呈示し、そして並列化とどのようにかかわるか、という観点から、ツールに対して批判的になってください。ほとんどのツールは、並列化の準備ができていません。スレッドセーフなライブラリーを探してください。並列化を活用するよう設計されたものが理想的です。

7 スケーラブル・メモリー・アロケーターを使用する。スレッド化プログラムでは、スケーラブル・メモリー・アロケーターを使用する必要があります。多くのソリューションがありますが、私は、それらすべてが malloc() よりも優れていると考えます。スケーラブル・メモリー・アロケーターを使用することで、グローバル・ボトルネックが排除され、スレッド間でメモリーを再利用してキャッシュを有効利用し、適切なパーティショニングによってキャッシュラインの共有が回避され、アプリケーションの速度が向上します。

8 作業負荷に応じてスケーリングするよう設計する。年々、プログラムが処理すべき作業は増加していきます。そのための準備が必要です。スケーリングを念頭において設計しておくことで、プロセッサー・コアが増えてもより多くの作業を処理することができます。毎年、インテルのコンピューターの処理能力は向上しています。将来、増大していく作業負荷を処理する場合に有利な設計でなければなりません。

この 8 つのルールでは、スレッド化が至るところで暗黙的に言及されています。ルール 7 のみが、明確にスレッド化について言及したものです。スレッド化が、マルチコアの価値を引き出す唯一の方法ではありません。マルチプログラムやマルチプロセスの実行は、特にサーバー・アプリケーションではよく使用されています。

ここで紹介したルールは、マルチコアから最大限の性能を引き出すのに役立ちます。プロセッサー・コアは増え、コア自体の多様性も増しているため、今後 10 年間に、この 8 つのルールのいくつかはその重要性が一層高くなることでしょう。例えば、異機種のプロセッサーや NUMA の到来は、ルール 3 の重要性をより高いものにしています。

ここで紹介した 8 つのルールをよく理解し、検討してください。この 8 つのルールや並列化全般に関して皆さんからのご意見をお待ちしております。

製品情報および購入情報については、次の Web サイトを参照してください。
http://www.intel.co.jp/jp/software/products/intel-parallel-studio-home/

©2009 Intel Corporation. 無断での引用、転載を禁じます。この文書は現状のまま提供され、いかなる保証もいたしません。記載内容は予告なしに変更されることがあります。
本資料に掲載されている情報は、現状のまま提供されます。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を許諾するものではありません。
インテルはいかなる責任を負うものではなく、また本資料に掲載されている情報に関する明示または黙示の保証 (特定目的への適合性、商品適格性、あらゆる特許権、著作権、その他、知的財産権の侵害への保証を含む) に関してもいかなる責任も負いません。
また、ソフトウェア開発製品については http://www.intel.com/software/products/ を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国およびその他の国における Intel Corporation の商標です。*その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。