THE PARALLEL UNIVERSE

Issue 2

intel Software

Think Parallel:

優れたプログラミングは開発者から始まる

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。最新のアプリケーションやツール、2010 年の注目すべき動向、そして今後の展開について紹介します。

目次

インテル® TBB での成功を受け、インテルはその設計サイクル全体をインテル® Parallel Studio でサポートしました。

並列プログラミングを支援するソフトウェアとして雑誌の記事や解析レポートでインテル® Parallel Studio が絶えず取り上げられることは驚くことではありません。インテルには OpenMP* や MPI* への取り組みを通して、ハイパフォーマンス・コンピューティング (HPC) 分野での長年の実績があります。最近では、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) やインテル® Parallel Studio などの HPC 以外のプロジェクトでも優れた成果をあげています。

マルチコア・プログラミングにおける本当の支援とは、既存のアプリケーションを変更する取り組みへの対応、「使いやすさ」の追求、そしてプログラムの設計プロセス全体で役立つツールを提供すること、を意味します。これらの目的を達成するためにインテルはまずインテル® TBB を提供し、シングルプロセッサー・システムの時代に設計された C++ 言語をマルチプロセッサー / マルチコア・プロセッサーで使用する上での多くの課題に取り組んできました。

このインテル® TBB での成功を受け、インテルは新たにインテル® Parallel Studio を提供し、その設計サイクル全体に対応しました。並列化用の言語拡張などの注目を集める話題に重点を置くだけではなく、並列プログラミングにおける長年の課題であるデバッグやチューニングの問題でも大幅な進歩を遂げました。さらに、最も一般的な並列プログラミングの問題であるデッドロックやデータ競合のデバッグ支援にとどまらず、並列プログラムではデバッグの難しいメモリーリークの対応についても重要な進歩を遂げました。

ここでの大きな前進が開発スケジュールを明確化することに大いに役立つことが証明されています。このようなインテルの新しいツールがなければ、並列プログラミングへの取り組みが成功する可能性は低いといっても過言ではありません。

私自身も参加したインテルと Microsoft* 共催の並列化に関するユーザー討論会では、レガシー、教育、そしてツールが主要な 3 つのニーズとして明らかになりました。我々の活動とこのようなニーズに合致したインテルの取り組みは、業界の中でも最も強力なものです。

取り組みの一例 :

- 大幅な変更を行うことなく既存のプログラムに並列化を追加できるように支援する
- 後で追加の教育が必要にならないよう、本当に必要なときや開発中、あるいは可能な時点でソフトウェア開発者を教育することによりマルチコアの機会への取り組みを支援する

インテル® ソフトウェア開発者ネットワークの Go-Parallel サイトとマルチコアポータルの両方で、多くの開発者が自身の知識とスキルの研鑽に利用している充実したフォーラムやトレーニング資料が提供されています。さらに、ハイパフォーマンス・ソリューションとして長年認知されているインテル® ソフトウェア・ツールは、並列プログラミングを強力にサポートしています。

その一方で、インテルは継続的な効果をもたらす投資も行っています。

2010 年は、インテルがマルチコア・プロセッサーとインテル® TBB の出荷を開始してから 6 年目の年にあたり、インテル® Parallel Studio も 2 年目を迎えます。また Cilk Arts* と RapidMind* のチームが現在活動中です。「インテルは次は何に着手するのか」とお思いになるでしょう。

インテル® TBB は受賞歴もあり、また多くの採用実績を誇る、並列化のための抽象化手法として紹介されています。我々はインテル® TBB への強い関心と、本製品を使用して開発された多くの製品を誇らしく思っています。2010 年、インテル® TBB は Microsoft の新しい Concurrency Runtime をサポートする予定です。これは、インテル® TBB のような並列モデルによる使用を意図して設計されており、新しいレベルの互換性が保証されます。この動きは、Windows* オペレーティング・システムで実行される並列化機能の連携に向けた重要なステップとなります。

インテル® Parallel Studio は最初のサービスパックがリリースされ、Windows* 7 にも対応し、インテル® Parallel Inspector / インテル® Parallel Amplifier のコマンドライン機能が追加されました。また、インテル® Parallel Studio には並列化を追加する設計と設計評価ステップを支援するインテル® Parallel Advisor も含まれるようになります。インテル® Parallel Advisor が提示するアドバイスが、プログラム・アーキテクトによって、並列化を追加する方法の決定というミスの起こりやすい煩雑なステップをどのように向上させるかという課題を解決する際に有効に活用されることを期待しています。

インテル® Parallel Studio は、Microsoft* Visual Studio* 2005 と 2008 をサポートしており、2010 はリリース後に間もなくサポートされる予定です。今後も開発者に幅広い選択肢を提供できるよう VS バージョンのサポートを拡張していきます。また、昨年、インテルは優れたチームを擁する数社を買収し、その具体的な成果も見えてきています。特に、インテル® コンパイラーは Cilk テクノロジーとインテルの Ct テクノロジーから恩恵を受けており、ベータ版でフィールドテストが行われています (2010 年 4 月現在)。

もちろん、ハイパフォーマンス・コンピューティング向けの並列化にも継続して取り組みます。インテルは、Microsoft* Concurrency Runtime に対応する最初でかつ唯一の OpenMP、スプリットレール対応の MPI ライブラリー、MPI ツール、コンパイラー、そして更新 / 拡張されたインテル® VTune™ パフォーマンス・アナライザーを提供する予定です。並列プログラミングは、今後ますます進化していくことでしょう。

私が常に強調しておきたいことは、従来のアプリケーションへの適合の支援や開発者の教育、非常に役立つツールの提供など輝かしい実績があったとしても、これらの素晴らしいツールで何をするかはソフトウェア開発者である我々自身にかかっているということです。並列プログラミングの時代が来る前と変わらず、優れた設計は人である開発者から生まれるものであり、ツールから生まれるものではありません。それは並列プログラミングも例外ではありません。我々が「Think Parallel」(並列化を考える) 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
2010 年 3 月

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.』等の並列化に関する文献を発表しています。

Levent Akyil

並列化した場合に処理 (ワークロード) がどのようにコア / プロセッサーに分散またはスケジュールされるかを考えたことはありますか？インテル® VTune™ パフォーマンス・アナライザーを利用すれば、この分析を簡単に実施できます。

Where Are My Threads?

インテル® VTune™ パフォーマンス・アナライザーを利用したスレッド化問題と並列化問題の検出

イベント・ベース・サンプリング (EBS) テクノロジーは、タイマー割り込みやキャッシュミスのようなプロセッサー・イベントをサンプリングして、システム全体のソフトウェア・パフォーマンス問題を識別します。

並列のワークロードがどのようにコア / プロセッサーに分散またはスケジュールされているかという質問を開発者から受けることがよくあります。

インテル® VTune™ パフォーマンス・アナライザーを利用すれば、この分析はとても簡単になります。イベント・ベース・サンプリング (EBS) テクノロジーは、タイマー割り込みやキャッシュミスのようなプロセッサー・イベントをサンプリングして、システム全体のソフトウェア・パフォーマンス問題を識別します (図 1)。EBS データから、イベントを生成したプロセス、スレッド、モジュール、関数、およびアプリケーションのソース行を特定できます。このテクノロジーを導入することで、各コアでサンプリングされたイベント数と生成されたスレッド数を確認できます。

サンプリング・ツールバーの CPU 情報の表示 / 非表示ボタン (図 1) で、プロセス、スレッド、モジュール、hotspot サンプリング・ビュー (図 2) に、プロセッサーごとに収集されたサンプル数とイベント数を表示できます。

表示された情報から、この特定のプログラム (sort_mt1.exe) は 2 つのコアで実行されていることと、各コアで収集されたサンプル数がわかります。しかし、このアプリケーションがいくつのスレッドを作成して、それぞれのコアでどのように実行されているかはわかりません。この情報を確認するには、スレッドビュー (図 2) と CPU ボタンの両方を選択します。図 3 は、sort_mt1.exe が 2 つのスレッド (thread18 と thread13) を作成し、分析中、各スレッドが両方のコアで実行されている (OS がこれらのスレッドを各コアで実行するようにスケジュールした) ことを示しています。thread18 のクロック間隔 (CPU_CLK_UNHALTED.CORE) を確認すると、このスレッドが各コアで実行され、そのほとんどの時間が Processor 0 で費やされていることがわかります。

スレッドごと、またはコアごとにサンプルがどのように分散されているかを確認したい場合は、サンプリング・オーバー・タイム (SOT) ビューが参考になります。スレッドビュー (またはその他のビュー) で SOT ビュー (図 3) を選択すると、収集されたサンプルがスレッド / コアごとに表示されます (図 4)。図 4 のビューはさまざまな場合に役立ちます。

SOT ビュー :

- オペレーティング・システム (OS) が実行するスレッドのスケジューリングを確認できます。
- スケジューリング問題を特定できます (図 5)。
- スレッド間のロードバランス (負荷分散) 問題を識別できます (図 6)。
- マイクロアーキテクチャー上の問題を関係付けることができます。

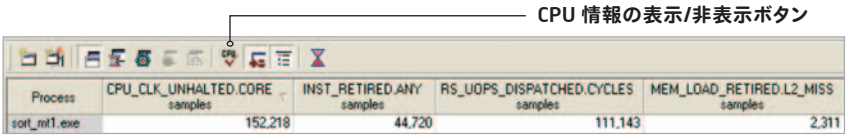


図 1

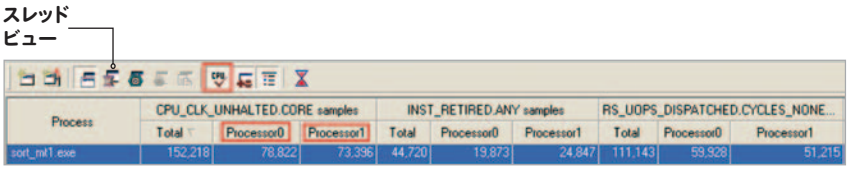


図 2

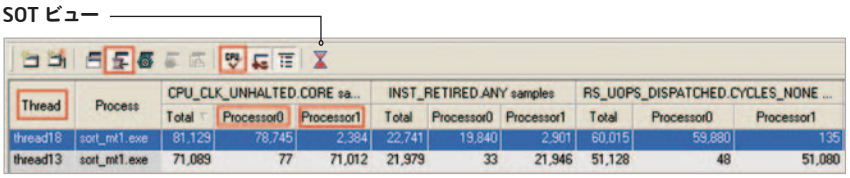


図 3

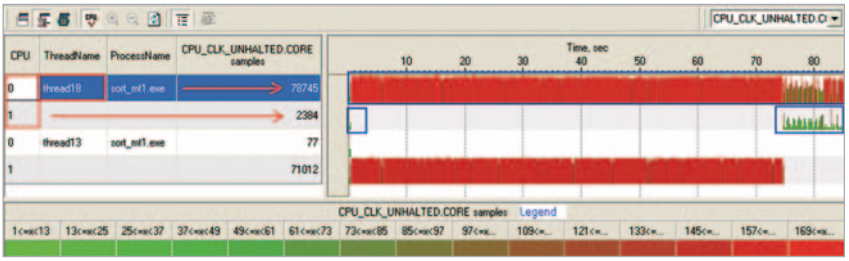


図 4

スレッドビュー (図 2) と CPU ボタンを同時に選択すると、これらのスレッドの実行とスケジューリングについての状況を確認できます (図 7)。図 7 で示されているように、thread9 と thread59 が各コアでどのように実行されているか詳細を確認することで、OS (この例では Windows* XP SP3) が両方のコアでスレッドをどのようにスケジューリングしているかを把握できます。ここでは、それぞれのスレッドが各コアでほぼ同じ時間実行されていることがわかります。

注 : RS_UOPS_DISPATCHED.CYCLES_NONE イベント (図 7) は、μ ops がディスパッチされなかったサイクル数 (ストールサイクル数) をカウントします。式全体は、CPU_CLK_UNHALTED.CORE (クロック間隔) ~= RS_UOPS_DISPATCHED.CYCLES_ANY + RS_UOPS_DISPATCHED.CYCLES_NONE (ストールサイクル数) で表されます。

マウスで領域を選択した後、コンテキスト・メニュー (右クリックメニュー) で [Zoom In (拡大)] を選択すると、その領域を拡大できます。図 8 の拡大された領域 (0-1.8 秒) では、スレッドがコア間でどのように渡されているかを確認できます。OS のスケジューラーは、同じコアに複数のスレッドを配置しません (例えば、コア 0 に thread9、コア 1 に thread59、またはその逆)。このシステムではコアが同じ 2 次キャッシュを共有しているため、この特定のスケジューリング・パターンは問題にはなりません。ただし、マルチソケット・システムではこのパターンは問題になります。

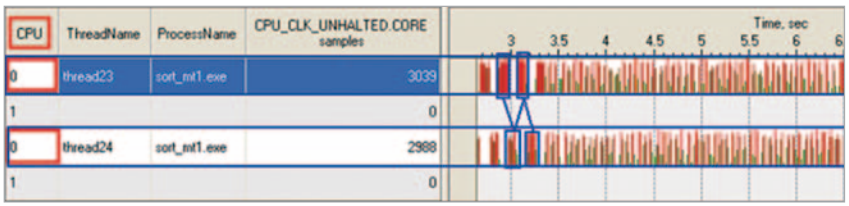


図 5: 手動でスレッド・アフィニティーを設定したことで問題が発生しています。各スレッドが Core/Processor 0 にスケジュール/ピンされています。

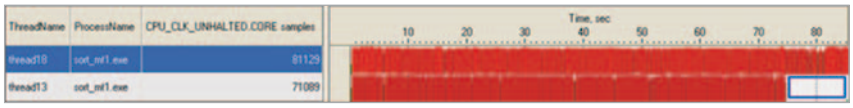


図 6: SOT はロード・インバランス問題を示しています。

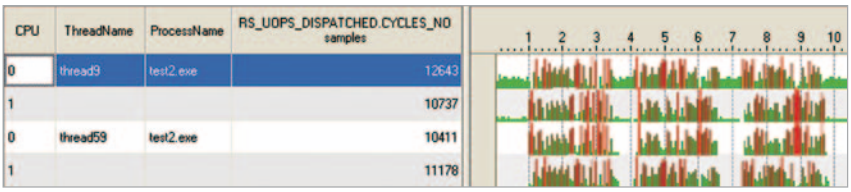


図 7: CPU/スレッドごとのストールサイクル数を識別します。

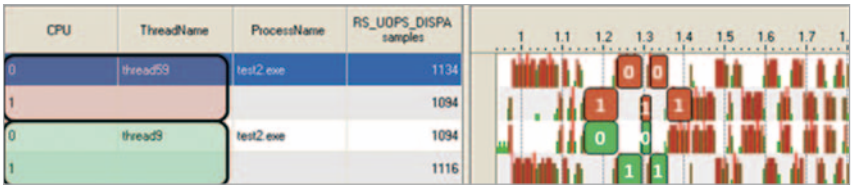


図 8: SOT ビューは OS がどのように特定のアプリケーションをスケジュールしているかを示します。

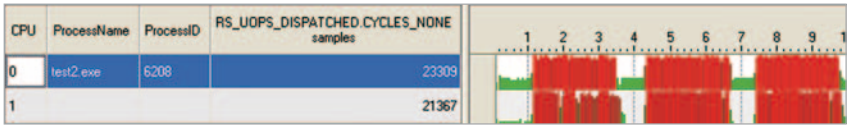


図 9: スレッド・アフィニティーの設定後の SOT ビューを示しています。

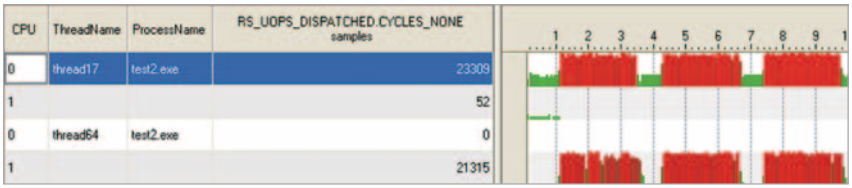


図 10: スレッド・アフィニティーの設定がアプリケーションの実行時間に与えた影響を示しています。

マウスで領域を選択した後、コンテキスト・メニュー (右クリックメニュー) で [Zoom In (拡大)] を選択すると、その領域を拡大できます。

スレッド・アフィニティー

ここで、スレッド・アフィニティーの概念を理解することが重要です。スレッド・アフィニティーは、特定のスレッドの実行をマルチプロセッサー・コンピューターの物理プロセシング・ユニットのサブセットに限定します。マシンのトポロジーにより、スレッド・アフィニティーはアプリケーションの実行速度に大きな影響を与えます。

ある特定の条件で、複数のプロセッサー / コアでスレッドを効率的にスケジュールするように OS スケジューラーの動作を変更してください。最新の OS とそのスケジューラーでは大幅に改善されており、一般的に最新のスケジューラーであるほど効率的に動作します。

インテル® コンパイラーの OpenMP* ランタイム・ライブラリーには、OpenMP スレッドを物理プロセシング・ユニットにバインドする機能があります。スレッド・アフィニティーは、Windows* OS システムとスレッド・アフィニティー対応カーネルを持つ Linux* OS システムのバージョンでサポートされています。このバインド機能は、3 種類のインターフェイスで使用できます。これらのインターフェイスは総称して、インテル® OpenMP スレッド・アフィニティー・インターフェイスと呼ばれます。インテルの OpenMP ランタイム・ライブラリーでサポートしているアフィニティー・タイプは、none (デフォルト) / compact / disabled /explicit / scatter です。

アフィニティー設定後

この例では、アフィニティーを scatter や compact に設定しても違いはありません。

図 10 は、thread17 と thread64 の両方が最初にスケジュールされた同じコアに残っていることを示しています。thread17 は、最初に Core 0 と Core 1 で実行するようにスケジュールされていましたが、途中から、残りの実行は Core 0 で行われています。

ロックと待機の分析 - パフォーマンス・チューニング

DAVID MACKAY, PH.D.

スレッド化されたソフトウェアにおいて、クリティカル・セクション、mutex、ロックが必要な場合があります。開発者はコードの最も影響力のあるオブジェクトを常に認識しているのでしょうか？ 影響を最小限に抑えるためにソフトウェアを調査する場合や、これらの同期オブジェクトの一部を削除してパフォーマンスが向上するようにデータを再構成する場合、最もパフォーマンスを向上できる箇所をどのように見つければ良いでしょうか？ そのような場合は、インテル® Parallel Amplifier の出番です。

インテル® Parallel Amplifier は、hotspot (およびコールスタック情報)、並列性、ロックと待機の 3 つの基本的な分析タイプを提供します。ロックと待機の分析は、スレッドを最も長い間ブロックしている同期オブジェクトを特定します。ソフトウェアの同期ポイントが多すぎたり、少なすぎたりすることはよくあることです。同期ポイントが少ない場合、競合状態が発生したり、結果は確定的ではありません (この問題が発生した場合、インテル® Parallel Amplifier ではなく、インテル® Parallel Inspector が必要です。インテル® Parallel Inspector の詳細は、MSDN Web セミナー「Got Memory Leaks? Find Errors Using Intel® Parallel Studio (Level 300)」を参照してください)。同期オブジェクトが非常に多い場合、削除することでパフォーマンスが最も向上するオブジェクトを特定できます。

Go-Parallel.com

「Go Parallel: マルチコアのパワーをアプリケーション・パフォーマンスに変える」 Web サイトで関連分野のブログを是非ご覧ください。

インテル® Parallel Inspector

サンプル・プロジェクトを通じて機能を理解する

Bradley J. Werth

インテル® Parallel Inspector は、正当性の検証作業における負担を軽くします。メモリーとスレッドの整合性をインテル® Parallel Inspector がどのように確保するのかを見ていきましょう。

インテル® Parallel Inspector は、メモリー整合性の確認とスレッド正当性の確認という 2 つの重要な機能を実行するツールを組み合わせたものです。メモリー整合性は、これまで常に、ソフトウェア開発において重要な課題の 1 つでした。あるプログラムがバッファー・オーバーランによってハックされる場合、それはメモリー整合性の問題です。注意していても、C のような複雑なメモリー構文を使用したときにプログラマーが間違いを犯すことはよくあります。同様に、マルチスレッドのセマンティクスでも、プログラミングのミスはよく起こります。インテル® Parallel Inspector はメモリーとスレッドの整合性を確実に保つことで、プログラマーの正当性をチェックする負担を軽くします。

この記事では、インテル® Parallel Inspector で検出できるすべてのメモリー問題とスレッド化問題を含むサンプル C++ プロジェクトを例として使用しています。独自のプロジェクトでも同様の問題を検出して解決できるように、インテル® Parallel Inspector の設定方法についても説明します。

グラフのデータは、インテル® Core™ i7 プロセッサー エクストリーム・エディションを搭載した Windows* 7 システム上でサンプル・プロジェクトを使用して測定したものです。インテル® Parallel Inspector の分析による時間の増加や速度の低下は、プラットフォーム、システム構成、分析するプロジェクトによって異なります。

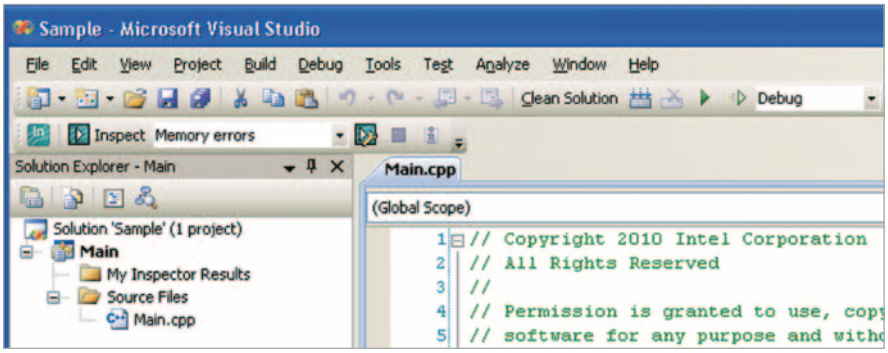


図 1: インテル® Parallel Inspector は Microsoft* Visual Studio* 2008 IDE のツールバーとして表示されます。

インテル® Parallel Inspector の使い方

インテル® Parallel Inspector は、Microsoft* Visual Studio* 2008 IDE にインストールされ、ツールバーとして表示されます。図 1 は、インストール後に Visual Studio* にプラグインされたインテル® Parallel Inspector を示しています。

一般的な使い方としては、デバッグ情報付きでアプリケーションをビルドした後、コンボボックスから分析の種類を選択し、[Inspect (検証)] ボタンをクリックして、メモリー分析やスレッド化分析を実行します。

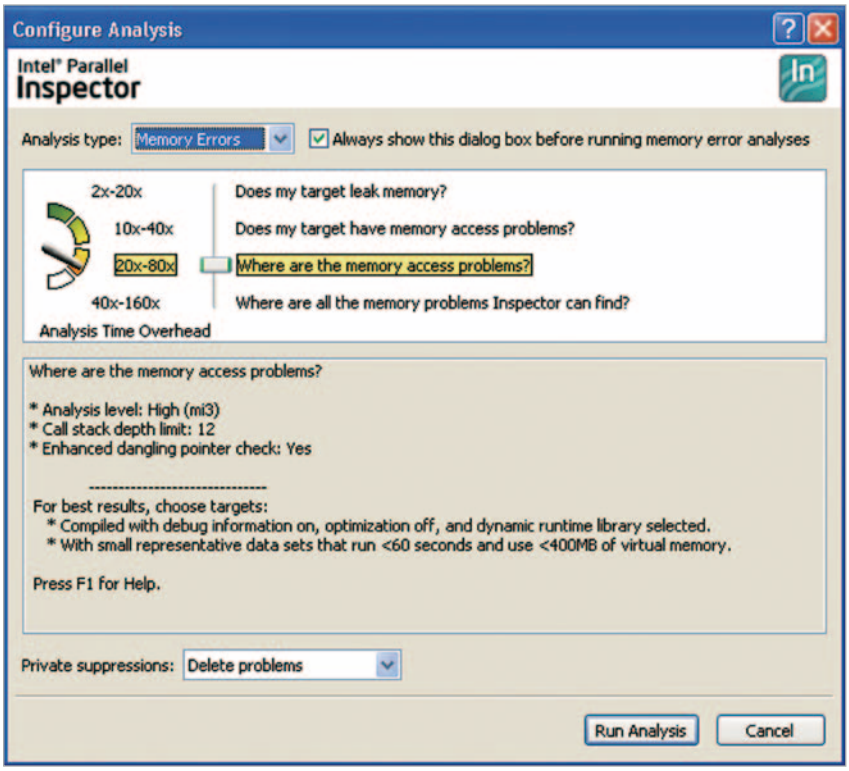


図 2: インテル® Parallel Inspector にはメモリー分析とスレッド化分析用の設定オプションが用意されています。

インテル® Parallel Inspector は、動的分析ツールです。ソースコードではなく、実行中のプログラムそのものを検証します。この手法には長所と短所があります。長所は、静的分析では検出できない多くの問題を検出できることです。短所は、プログラムの実行速度が大幅に低下することです。長所と短所のバランスを選択できるように、インテル® Parallel Inspector では、メモリーとスレッド化分析の両方で 4 つのレベルを用意しています。レベルは 1 から 4 で、レベルが高くなるほど精度は高くなり、実行時間は低下します。図 2 は、メモリー分析のレベルを指定する設定画面です。

この設定画面で [Run Analysis (分析の実行)] ボタンをクリックすると、インテル® Parallel Inspector は分析を開始し、完了するまで、あるいはツールバーの [Stop (停止)] ボタンが押されるまでプログラムを実行します。[Stop (停止)] ボタンを押した場合、一部のプログラムは正しく検出されないことがあります。分析が完了すると、インテル® Parallel Inspector は、分析結果を Microsoft* Visual Studio* IDE のタブに表示します。図 3 は、同じプロジェクトでレベル 3 のメモリー分析を実行した結果を示しています。

エラーの 1 つをダブルクリックすると、問題のソースと問題の診断に役立つ詳細な情報が表示されます。

図 3: 分析結果は IDE のタブに表示されます。

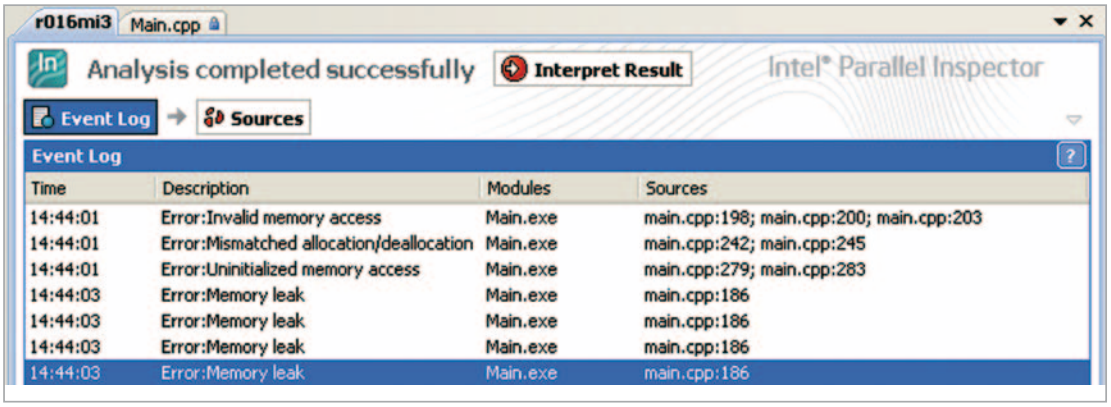
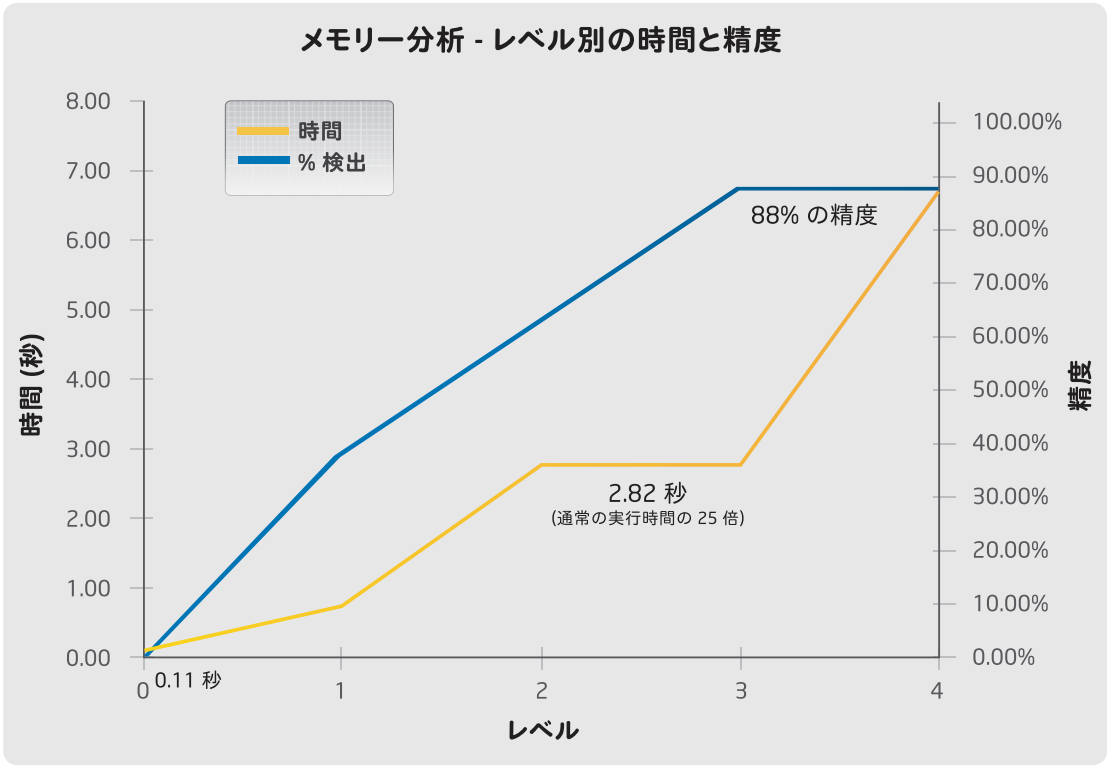


図 4: メモリー分析の精度は分析レベルに応じて向上します。



メモリー分析 : 機能、精度、パフォーマンス

サンプル・プロジェクトには、インテル® Parallel Inspector の最も高い分析レベル (レベル 4) で検出できる 11 のメモリーエラーが含まれています。この中の 3 つのエラーは、ヒープが壊れたり任意のメモリーへ書き込みを行うなど、基本的に安全ではありません。このため、これらのエラーはデフォルトでは無効になっているプリプロセッサー定義を有効にした場合のみ含まれます。レベル 1 では、インテル® Parallel Inspector はコードのメモリーリークをすべて検出しますが、深い階層のメモリーリークについては完全なコールスタック情報を得られません。レベル 2 では、不正なメモリーの読み取り / 書き込みが検出されます。レベル 3 とレベル 4 では、問題のコールスタックの深さがより深くなります。

サンプル・プロジェクトで検出された「安全性に関する」メモリー問題 :

- 異なるコールスタックの深さでのメモリーリーク 4
- 不正なメモリーの読み取り 2
- 初期化されていないメモリーへのアクセス 1
- 割り当てと解除の不一致 1

図 5: スレッド化分析の精度は分析レベルに応じて向上します。

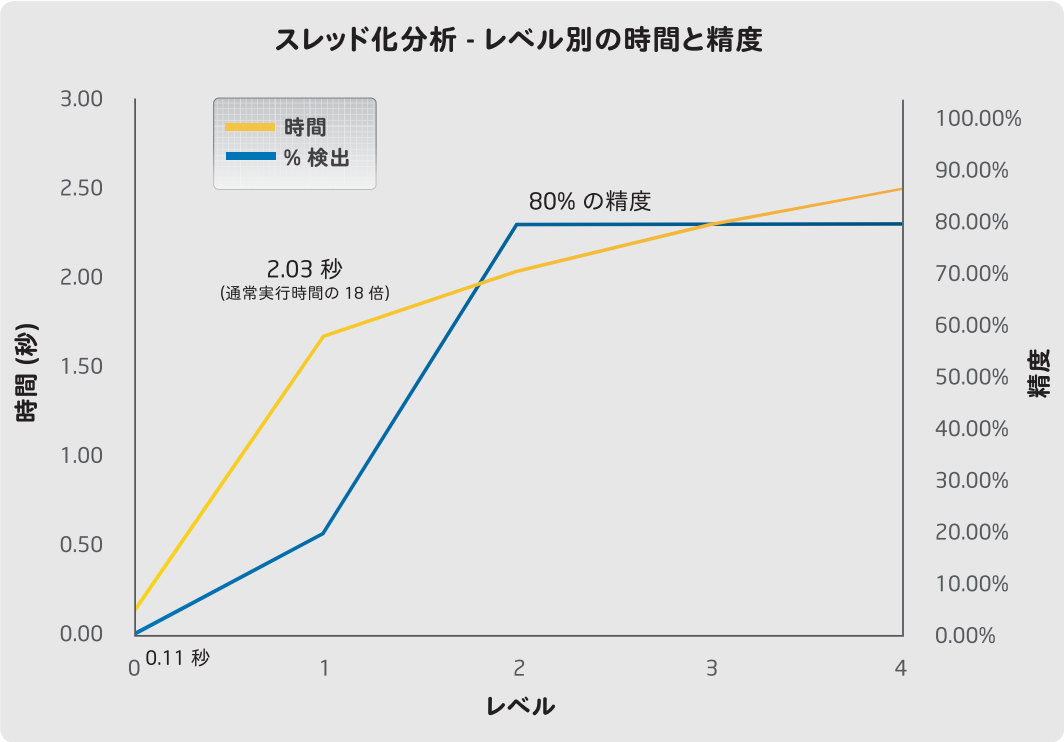


図 4 は、インテル® Core™ i7 プロセッサー エクストリーム・エディションを搭載した Windows* 7 システム上で、サンプルプログラムのメモリー分析を行ったときの精度と実行時間の関係を示しています。分析レベル 1 から 4 は、各レベルで検出した問題の割合を示しています。分析レベル 0 は、インテル® Parallel Inspector が全く実行されなかった場合 (つまり、コードの本来のパフォーマンス) を表しています。

このデータからは、精度と実行時間のバランスという点で、レベル 3 のメモリー分析が非常に優れていることがわかります。レベル 3 では、標準の実行時間の 25 倍で、このプロジェクトの問題の 88% が検出されています。レベル 4 の分析では追加情報 (たとえば、より深いコールスタックとスレッドスタック分析) が提供されますが、追加のメモリー問題は検出されていません。一部のプロジェクトには、レベル 4 のメモリー分析が必要な、より複雑なメモリー・アクセス・パターンが含まれていますが、ほとんどのプロジェクトではレベル 3 で十分です。

スレッド化分析 : 機能、精度、パフォーマンス

サンプル・プロジェクトには、インテル® Parallel Inspector の最も高い分析レベル (レベル 4) で検出できる 5 つのスレッド化エラーが含まれています。メモリー分析と異なり、すべてのスレッド化エラーはメモリーを壊さないという点で安全です。レベル 1 では、インテル® Parallel Inspector は潜在的なデッドロックのみ検出します。レベル 2 では、データ競合も検出されます。レベル 3 とレベル 4 では、データ競合のコールスタックの深さがより深くなります。

サンプル・プロジェクトで検出されたスレッド化問題 :

- ヒープデータの競合 3
- スタックデータの競合 1
- デッドロック 1

図 5 は、インテル® Core™ i7 プロセッサー エクストリーム・エディションを搭載した Windows* 7 システム上で、サンプルプログラムのスレッド化分析を行ったときの精度と実行時間の関係を示しています。分析レベル 1 から 4 は、各レベルで検出した問題の割合を示しています。分析レベル 0 は、インテル® Parallel Inspector が全く実行されなかった場合 (つまり、コードの本来のパフォーマンス) を表しています。

スレッド化分析のデータは、メモリー分析と同じ傾向を示しています。レベル 2 では、標準の 18 倍の実行時間で、問題の 80% が検出されています。レベル 4 の分析は、スタック変数のデータ競合をチェックします。通常、スタック変数は共有されないため、データ競合は一般的ではありません。レベル 2 や 3 のような低レベルの分析は、一般的なテストに利用します。

関連情報と資料

インテル® Parallel Inspector のヘルプファイル (Microsoft* Visual Studio* の [ヘルプ] メニューから [Intel® Parallel Studio] - [Parallel Studio Help] - [Inspector Help] でアクセス) の「Problem Type Reference」セクションに、検出可能なエラーの例が含まれています。

Paul Petersen

並列化のためにアプリケーションを変更するとき、テストスイートが非常に役立ちます。インテル® Parallel Advisor は、既存のシーケンシャル実装の分析を支援するように設計されています。

ADVISOR

真っ白な紙。 恐ろしくもあり、また計り知れない力を秘めたものでもあります。それは制限のない自由な創造への切符です。ソフトウェア開発者として、我々がこのような切符を手に入れる機会はどのくらいあるでしょうか。そのような機会は滅多に訪れないものです。

ソフトウェア開発者の多くは、既存のソリューションを適合させて新しい目的の達成を目指します。それは既存のアルゴリズムの最適化であり、顧客がすでに価値を認めたソリューションの新しい実行モデルの実現です。

もしかしたら、あなたは並列化への移行の過程で現在のソースコードを作成し直すチャンスに恵まれた開発者の 1 人かもしれません。だとしたら大きなやりがいを感じられるでしょう。なぜなら、並列実行の利点を最大化するアルゴリズムの設計と実装において、無限の自由があるからです。

あるいは、既存の実装の大部分を利用しなければならないかもしれません。それでも大きなチャンスと言えます。この場合は作業が簡単です。もしかしたら、すでに「ダイヤモンドの原石」を持っているかもしれません。実装しようとしている「正しい」定義がわかっており、既存の逐次アルゴリズムの外部動作によって正しい動作が定義されている可能性があります。

ORIGINS

テストスイートは、この正しい動作と照合してアプリケーションを確認します。並列化のためにアプリケーションを変更するとき、テストスイートが非常に役立ちます。実行したそれぞれの変更ステップの後で、アプリケーション中で実行された部分をテストスイートで検証することで、変更の妥当性をチェックできます。

アプリケーションの動作に合うようにアルゴリズムの変更を行う場合、この変更が望ましいものであるかどうかを早めに判断することが重要です。そのような場合、この新しい動作に沿ってテストスイートを更新する必要があります。動作に問題が発生した場合は、変更を元に戻して、前のバージョンで作業を行います。

インテル® Parallel Advisor (インテル® Parallel Studio に同梱) は、既存のシーケンシャル実装を分析して、アプリケーションの並列実行を有効に活用するためのリファクタリングまたは再設計の可能性を探るのに役立ちます。本記事では、インテル® Parallel Advisor の設計の基になっている原理をいくつか紹介します。

緩和逐次実行を有効にするリファクタリング

アプリケーションですでに「正しい」定義が具体化されている場合は、それを維持するようにします。これは、リファクタリング手法を使用して、アプリケーションの潜在的な並列性を見出すことを意味します。この並列化は潜在的です。なぜなら、アルゴリズムを表現するために逐次言語を使用することは、正しい実行に必要な依存性を過剰制約するからです。リファクタリングとは、外部からみた機能的動作を変更せずにアプリケーションの内部構造を変更するプロセスのことです。これにより、本来の意図がさらに明確になり、アルゴリズムのシーケンシャル実装に存在する暗黙的な逐次依存性を排除することができます。

```
T1 = F1(A + B)
T2 = F2(C * D)
```

図 1

```
parallel {
    task { T1 = F1(A + B) }
    task { T2 = F2(C * D) }
}
```

図 2

図 1 で、逐次セマンティクスは、まず A+B の足し算を行い、関数 F1 を適用して、その結果を T1 に格納します。その後に C*D の乗算を行い、関数 F2 を適用して、その結果を T2 に格納します。F1 と F2 が純関数と仮定すると、最初に T2 を計算し、それから T1 を計算しても数学的にセマンティクスは等価です。

これらの 2 つの文の記述によって、記述される前には存在しなかった制約が生まれます。最適化コンパイラー (およびアウトオブオーダー実行ハードウェア) は、より迅速な実行を実現するために真の依存性と見せかけの依存性を理解しようとします。タスク境界 (図 2 で task として示されている並列動作など) を宣言することで、同じ手法をソースコードに静的に使用して、正しい実行を保証するために暗黙の制御やデータ依存を実施する必要がないことを示します。

図 3 は、似たような状況を表しています。このループは、変数 i が 0:N-1 の範囲にある代入文セットの省略表現です。このループによって生成される結果では、B と C の要素の合計を含む A の要素がそれぞれ作成されます。逐次プログラムでは、帰納変数 i=i+1 で計算されるインデックス変数を指定することによって、このループが過剰制約されます。図 4 のタスク境界の宣言では、すべてのループの反復処理が論理的なタスクの分割 (各タスクの作成時に i の値を取得) として定義されます。

リファクタリングによる並列化の導入では、主に解決しなければならない問題を逐次セマンティクスが過剰制約している場所を特定する作業を行います。リファクタリングによる並列アプリケーションの設計においては、元の逐次実行が並列プログラムの 1 つの単なる実行に過ぎないキー・プロパティーを作成します。並列プログラムを考えてみましょう。このプログラムをシングルスレッドで実行すると何が起こるでしょうか。緩和あるいは削除された人工的な逐次依存性の場合、それらを追加して元に戻し、シングルスレッドで並列プログラムを実行すると、同じ動作が保たれます。

実行順を考えなくても良い場合は、プログラムは並列に実行できます。実行順が問題となる場合 (前の計算に次の文が依存するなど) は、逐次実行のままにします。

マルチレベルの並列タスクの実行

緩和逐次実行の機会を特定するリファクタリングは、一般に fork-join 並列化によって実装されます。逐次実行の制約を緩和するときに fork を実行し、逐次実行の制約を実施するときに join を実行 (バリアーを使用) します。join ポイントのバリアーでは、バリアーの前からバリアーの後までのあらゆる依存性を満たすことができます。

単純な fork-join 並列化では、アプリケーションは逐次実行されるか、または並列に実行されるかのいずれかになります。アムダールの法則によると、潜在的な速度向上はプログラムの逐次実行の割合に制限されます。そのため、並列実行から逐次実行に移行する場合は常にパフォーマンスが損なわれます。

この問題を克服するためには階層的に考えると良いでしょう。ここでのポイントは、より内側のレベルで逐次化アルゴリズムを複製し、それぞれの作業を独立して行える外側のレベルで複数のタスクを作成する方法を見出すことです。QuickSort アルゴリズムが良い例です (図 5)。

このアルゴリズムは、逐次フェーズ (小規模配列のソート、ピボットの選択、配列のパーティショニングなど) と並列フェーズ (二分割された配列のソート) から成ります。ここで有効に利用できる階層は、QuickSort 関数への再帰呼び出しです。

階層的に (時に再帰的に) 考えることで、並列実行を有効利用できる独立した作業を特定できるようになります。上位 2 つの独立した QuickSort 呼び出しからのみ並列化を作成した場合、速度向上は暗黙的に制限され、係数 2x 程度の速度向上にとどまります。このようなタスクを再帰的に分割して、小さなタスクにすることによって並列性が増します。

階層的並列化設計が役立つもう 1 つの理由は、起動できるタスク数が増えることにあります。並列化に最適な問題は、大規模なオブジェクト群があり、関数の適用によってそれぞれが独立して変換されるものです。アルゴリズムをこのようなフォームに変換できる場合には最良の結果が達成されます。

しかし、オブジェクトがこのように独立していることはあまりなく、多くの場合、「トップレベル」のわずかなオブジェクト群のみが独立しています。並列化をこの「トップレベル」のオブジェクト群のみに適用する場合、効果的な並列実行が行われるのに十分な粒度は得られるものの、スケーラビリティーが「トップレベル」のオブジェクト数に限定されます。

並列化を行うのに十分なアルゴリズムの実行時間がある場合には、ほかにも独立したデータ項目の小規模なセットに対して独立した計算を行う、別のレベルの入れ子アルゴリズムがあるかもしれません。並列化を両方のレベルに適用することで、並列アルゴリズムのスケーラビリティーが向上します。

```
for (int i = 0; i < N; ++i)
      A[i] = B[i] + C[i]
```

図 3

```
parallel {
      for (int i = 0; i < N; ++i)
           task { A[i] = B[i] + C[i] }
}
```

図 4

```
void QuickSort( Value A[], int L, int H ) {
      if (H-L < TooSmallLimit) {
              SerialSort(A, L, H);
              return;
      }
      Value Pivot = A[ L+(H-L)/2 ];
      int L1 = L; int H1 = H;
      while (L1 < H1) {
              if (A[L1] < Pivot)
                      ++L1;
              else if (A[H1] >= Pivot)
                      --H1;
              else
                      Swap(A[L1], A[H1]);
      }
      parallel {
              task { QuickSort(A, L, L1-1); }
              task { QuickSort(A, L1, H); }
      }
}
```

図 5

タスクの選択

プログラムタスクの逐次実行は複数のレベルで行えます。極端な例では、アプリケーション全体を 1 つのタスクと仮定できます。あるいは、文や表現のそれぞれを 1 つのタスクと仮定することもできます。しかし、いざアプリケーションにタスクを追加する場所を選択しようとすると、たいていはどちらも当てはまりません。

アプリケーションの潜在的な並列性は、作成するタスクの数によって制限されます。2 つのタスクのみを作成した場合 (メインプログラムも論理的にはタスク)、理想的な速度向上は、利用可能なプロセッサー数に関わらず、逐次 (逐次キャッシュやメモリー帯域幅の効果は無視) と比べて 2 倍の速度にとどまります。さまざまな形で現れるオーバーヘッドもまた、並列タスクの作成による潜在的な利点を損ないます。そのため、優れたパフォーマンスを達成するためには、利用可能なプロセッサー数よりも多くのタスクを識別する必要があります。

利用可能なタスクが多いと、ほかの点でも役立ちます。タスクが 2 つあり、1 つのタスクが短時間 (1 秒など) で、もう一方のタスクが長時間 (59 秒など) という極端な場合、この 2 つのタスクを順次に実行すると 1 分かかります。並列で実行する場合はどうでしょうか。2 倍の速度向上が達成されて、30 秒で両方のタスクを完了できると予想するかもしれません。

しかしながら、両方のタスクの完了には 59 秒かかります。両方のタスクを実行する最短時間は、実行時間の長いほうのタスクが完了する時間と同じです。2 番目のタスクの実行時間が 59 秒のため、少なくともその分は待つ必要があります。可能であれば、2 番目のタスクを細かく分割してより大きなタスク群にします。例えば、59 秒かかるタスクの代わりに、それぞれの実行が 1 秒の 59 個のタスクを作成すると、2 台のプロセッサーを使用して 2 倍の速度向上を達成し、60 個のタスクを 30 秒で完了できる可能性があります。

目標のパフォーマンスを達成するために十分な並列性の可能性を引き出すには、多数の小さなタスクが必要であることがわかります。次の 2 つの作用により、バランスを保ちながら、タスクを小さくし過ぎないようにできます。

1 つ目は、1 つのスレッドに 1 つのタスクをスケジューリングすると、高くはないもののコストがかかります。それぞれが単一動作の 10 個のタスクを 1 つのスレッドにスケジューリングすることは、10 の動作すべてを 1 つのタスクに結合して、そのタスクを 1 つのスレッドにスケジューリングするよりもわずかにコストがかかります。ほとんどの並列フレームワークには、タスクを結合してより大きなチャンクを作成し、スレッドにスケジューリングして実行する能力が備わっています。これは単純な機械的処理です。しかし逆の場合、つまり一枚岩の大きなタスクを小さなタスクに分割することは、並列フレームワークでは非常に難しいものの、開発者が簡単に指定できることがあります。

2 つ目として、優れたタスクはオブジェクトやオブジェクト・セットの一貫した計算をカプセル化します。1 つのタスクによってアクセスされる変数をすべて調査すると、次の 3 つのケースに分類できることがわかります。

3 つのケース：

1. 変数はタスクに対してすでに private であるか、またはすべてのタスクにとって読み取り専用である。
2. 変数をタスクに対して private にできる。
3. 変数はタスク間で値をやりとりする。

この記事の Part 2 では、これらのケースを検証することによって、変数の使用方法がアプリケーションを変換する際の選択肢にどのように影響を与えるかを見ていきます。記事の Part 2 は、Go-Parallel.com で読むことができます。

intel®
Software

マルチコア・プラットフォームの性能を引き出す

インテル® コンパイラー、ライブラリー、デバッグおよびチューニングの各ツールにより、今日のマルチコア・イノベーションにおいてスケーラビリティーを備えた、信頼のおけるアプリケーションの開発に必要な機能のすべてが揃います。スーパー・コンピューターからラップトップ、組み込みシステムから携帯端末にいたるまで、インテル® ソフトウェア・ツールで従来のシリアルコードおよびスレッド化コードを最適化して、マルチコアに対応できます。

インテル® ソフトウェア・ツールであなたもロックスターに。
無料評価版のダウンロード: http://www.intel.co.jp/jp/software/products/

© 2010 Intel Corporation. 無断での引用、転載を禁じます。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。