マルチコア向け並列プログラミングの 8 つのルール

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

James Reinders The Parallel Universe 創刊編集長および名誉編集長

[編集者注 : この記事は The Parallel Universe 1 号 (2009 年 4 月) に掲載されたものです。前編集長 James Reinders のアドバイスは 11 年経過した現在でも適切なものです。その先見性に敬意を表して、この号で再掲することにしました。]

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。

ルール 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。

ルール 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・ブロックなどは、すべて抽象化を使用している例です。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。ネイティブ・スレッド・ライブラリーは、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。

ルール 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。

ルール 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して実行し、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分かれば、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。

ルール 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多くの不具合の原因になります。問題の解決には、暗黙的な同期を使用します。明示的な同期が必要な場合は、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。ロックの必要がないプログラムを設計してください。

ルール 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提示し、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。

ルール 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。

ルール 8: ワークロードに応じてスケーリングするように設計する

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、増加していくワークロードを処理する場合に有利な設計でなければなりません。

マルチコア・プロセッサーから最大限の性能を引き出す

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・プロセッサー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ー層高くなることでしょう。例えば、ヘテロジニアス・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。**[編集者注 : 太字は編集者が追加したものです。James は The Parallel Universe 1 号ですでにヘテロジニアス並列処理について言及していました。]

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。