マルチコア CPU が並行処理にもたらす変化

スレッド・ベースのプログラムでアプリケーションを並列処理する方法がマルチコアの時代に適さない理由

Vasudevan Thiyagarajan
Architect
Royal Caribbean Cruises Ltd.

2012年 9月 06日

マルチコア・チップ・アーキテクチャーになっても、個々のコアのパフォーマンスはほとんど改善されていません。この傾向は続いているため、ハードウェア・リソースを最大限に利用するための重責はオペレーティング・システム開発者や、プログラミング言語開発者、アプリケーション開発者などの肩にかかってきています。アプリケーション開発コミュニティーの多くの人々は、スレッド・ベースの並行プログラミングを利用してアプリケーションの並列処理を実装しています。この記事では、スレッド・ベースのプログラミング手法がマルチコア時代のアプリケーションの並列処理には最適ではない理由を説明します。

ムーアの法則 (1965年に Gordon Moore 氏が行った「1 個の IC に含まれる部品点数は 18 ヶ月から 24 ヶ月ごとに倍になるだろう」という予言) は 2012年の現在まで真実であり、今後も 2015年から 2020年頃までは相変わらず当てはまると見込まれています (「参考文献」を参照)。2005年までは、CPU のクロック周波数も一定の割合で高くなってきました。それだけでも、CPU 上で実行されるすべてのアプリケーションのパフォーマンスを向上させるには十分でした。そのため、アプリケーション開発コミュニティーは、アルゴリズムの改良に関する投資をほとんど、あるいはまったくしなくても、パフォーマンスが向上するという結果を享受することができました。

しかし 2005年以降、クロック周波数とトランジスター数の増加は鈍っています。プロセッサーの材料の物理的性質により、クロック周波数の増加は止まり (それどころか低下しています)、プロセッサー・メーカーは、より多くの実行ユニット (コア) を 1 つのチップ (ソケット) に搭載するようになりました。この傾向は今後もしばらく続きそうですが、この傾向により、大まかに言うと以下の 2 つの点で、アプリケーション開発やプログラミング言語開発のコミュニティーに対する圧力が高まっています。

今後はどうなるのか

アクターによる並行処理が次第によく使われるようになり、従来の Java による並行処理に代わる必要な手段となりつつある理由を、knowledge path「Actor concurrency for Java applications」(developerWorks、2012年 5月) で学んでください。

この developerWorks ポッドキャストでは、Andy Glover 氏が並行処理の専門家、Alex Miller 氏にインタビューしています。

- シングル・スレッド・アプリケーションの場合、より強力な CPU にアップグレードしただけでは、2005年以前のようにパフォーマンスを高められるわけではありません。CPUの中にコアが何個あっても、シングル・スレッド・アプリケーションのパフォーマンスは同じです。つまり CPU のコア数に関わらず、コア当たりのスループットはほとんど同じです (ただし、コンパイラー、仮想マシン、オペレーティング・システムなどのレベルで、自動並列処理の手法にブレークスルーが起きないという前提です)。
- マルチコア CPU にアップグレードしても、効果があるのはシステムに追加される負荷に対してのみであり、既存の負荷に対しては効果がありません。

使用可能な CPU コアを効率的に利用する唯一の方法は並列処理です。これまで、並列処理は主にオペレーティング・システムのプロセス・レベルで使用され、シームレスなマルチタスク処理や、マルチプロセスのエクスペリエンスを実現してきました。アプリケーション開発においては、並列処理を実装するためのメカニズムとしてはスレッド・ベースの並行プログラミングが主流です。

スレッド・ベースのプログラミング・モデル

スレッドは軽量のプロセスであり、OS によってスケジューリングされる最小の実行単位です。1 つのプロセス内のすべてのスレッドはメモリー内の同じアドレス空間を共有し、その結果すべてのスレッドはメモリー内のオブジェクトを共有します。この記事では、スレッドの動作の技術的な詳細については説明しません。

スレッド・ベースの並列処理には以下の 3 つの利点があります。

- 十分に確立されたプログラミング・モデルです。
- アプリケーション開発コミュニティーは、スレッドの作成、スケジューリング、実行、管理に関して十分に理解しています。
- 開発者はシーケンシャルな方法でアルゴリズム開発を考えるように訓練されています。スレッド・モデルは単純にそれと同じ手法を並列処理に拡張しているにすぎません。

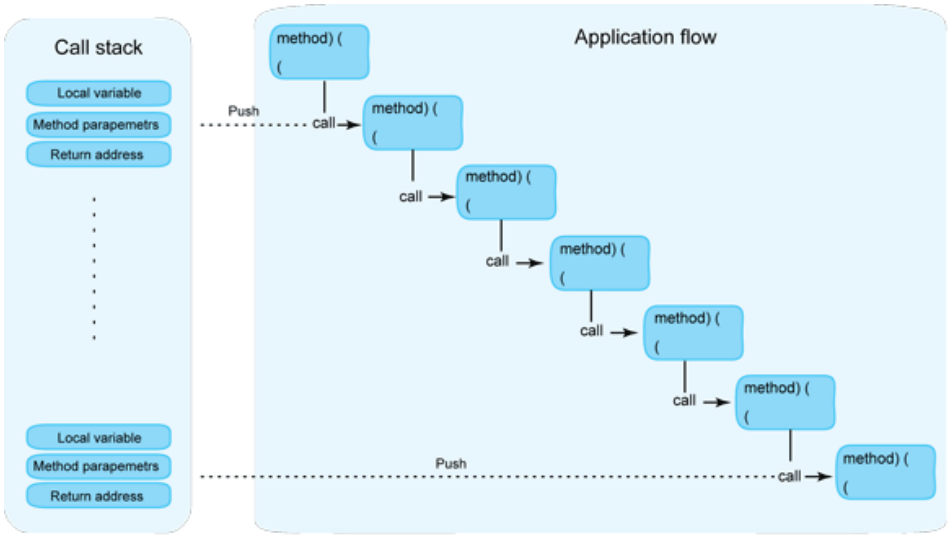
ただし、スレッド・ベースのプログラムでアプリケーションを並列処理する方法には、これらの利点を上回る重要な問題があります。この記事では、なぜ明確にスレッドを使用したプログラムでアプリケーションを並列処理する方法が CPU コアを利用する最善の方法ではないのか、またなぜ別のプログラミング・パラダイムが必要なのか、その理由を説明します。

コール・スタックの深さ

コール・スタックは、OS または仮想マシンによって管理される内部的な構造であり、あらゆるメソッドの呼び出しを処理します。スレッド実行中にメソッドが呼び出されるたびに、その呼び出しによって 1 つのスタック・フレームがプッシュされます (スタック・フレームは現在のメソッド呼び出しに関する詳細 (引数、戻りアドレス、ローカル変数など) で構成されます)。

図 1 はメソッドを呼び出した場合の内部動作を示しています。

図 1. コール・スタックの内部構造と深化



いかにアプリケーションを複数の論理層にモジュール化しても (例えばコントローラー層、ファサード層、コンポーネント層、DAO (Data Access Object) 層など)、実行時には最終的にスレッドが処理を行い、スレッドには 1 つのスタックしかありません。コール・スタックは、モジュール化されたソース・コードをアプリケーションの実行時に扱うための素晴らしい発明です。しかしアプリケーションの複雑さやシステムの負荷が増してくると、現在のコール・スタック構造モデルによってアプリケーションのスケーラビリティーが制限され、メモリー・サイズやオブジェクトへの到達可能性に関連するこのモデル特有の問題を生じます。

オブジェクトへの到達可能性

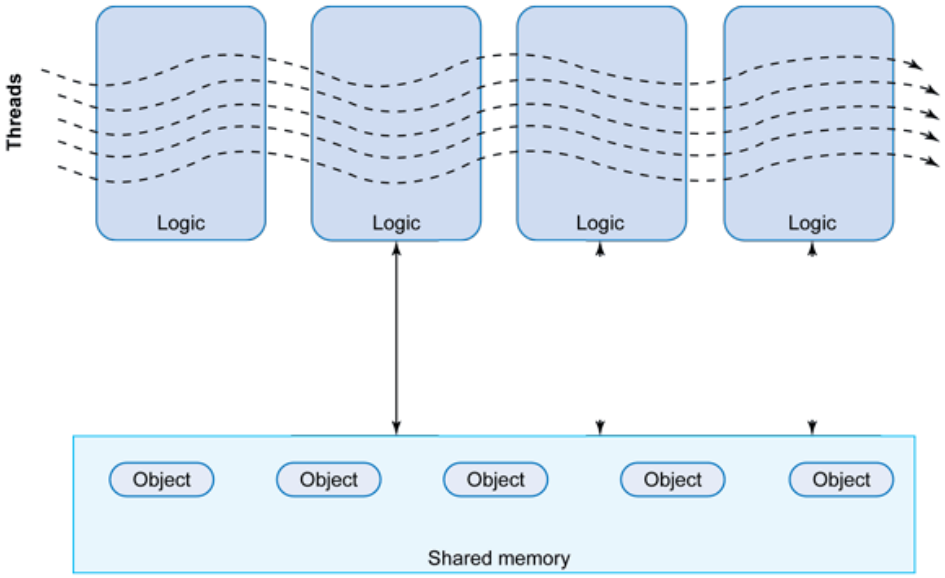
コール・スタックが深くなることによる別の問題として、コール・スタックの中にオブジェクト参照が保持されるものの結局は使用されない、という問題があります。例えば図 1 で、実行フローの最も深いところにあるメソッドをスレッドが実行している場合に、コール・スタックの中のすべてのメソッドのローカル引数とローカル変数がすべて必要になることはあり得ません (例えば、ある 1 つのスレッドが DAO 層のコードを実行する際に、サーブレット層、コントローラー層、ファサード層、そしてその他の層でのメソッド呼び出しによってコール・スタックにプッシュされたすべてのローカル引数とローカル変数がアプリケーションに必要になるということはまずあり得ません)。しかし、それらの変数や引数には有効な参照が含まれているため、解放されたりガーベッジ・コレクションの対象にされたりすることはありません。

Java コール・スタックの実装は、メソッドの呼び出しから戻るとすべての参照が自動的に解放されるように設計されています。この設計は JVM の負荷が高くない場合には許容されるかもしれません。しかし大量のスレッドがアクティブな状態で JVM が動作している場合には、問題になる可能性があります。例えば、コール・スタック内で使用してはいないものの有効な参照のために各スレッドが最大 5MB を使用し、アクティブなスレッドが 100 個ある場合、ヒープ領域の 500MB はコール・スタック内の変数や引数によって参照されているため、JVM はその 500MB に対してガーベッジ・コレクションを実行することはできません。これは 32 ビット・マシンの場合、JVM で利用可能な全メモリーの少なくとも 25 パーセントに達する可能性があり、無視できない大きさです。

共有オブジェクト

スレッド・ベースの並列処理によるもう 1 つの重要な問題は、複数のスレッドで可変オブジェクトを共有することから同期が必要になることです (図 2)。

図 2. 共有メモリー



同期の概念は新しいものではなく、広く使用されていますが、同期によってアプリケーションのパフォーマンスは低下します。これはロックを獲得するためのシーケンスによって、ロックが解放されるまでスレッドが強制的に待機状態やスリープ状態にされる場合があり、それによって、内部でスレッド・コンテキスト・スイッチがトリガーされることになるからです。通常はコンテキスト・スイッチによって、スレッドが処理を完了するまでの時間は長くなります。またコンテキスト・スイッチによってコアの中にあるパイプライン命令やキャッシュはすべてフラッシュされます。JVM に大量の並列スレッドがある場合、同期とロックのためにスレッドのコンテキスト・スイッチが頻繁に発生する可能性があります。

シーケンシャル・プログラミング

シーケンシャル・プログラミングは必ずしもスレッドそのものに関する問題というわけではありませんが、アプリケーションにおけるスレッドの使い方に関連します。コンピューティングの黎明期に、(ユーザーが実行依頼したジョブの中で) 命令をシーケンシャルに実行するための論理概念として OS プロセスが考案されました。しかし当時に比べて一部のプロセスは何倍も複雑になったにもかかわらず、シーケンシャル・プログラミングの考え方は相変わらず広く採用されています。複雑さが増すにつれ、さまざまなシステム層 (バックエンド、ミドルティア、フロントエンド) が存在するようになりました。しかし、1 つの層内ではアプリケーションのユースケースは相変わらずシーケンシャルに実行されており、さまざまなコンポーネントにわたるあらゆるロジックを 1 つのスレッドが処理しています。

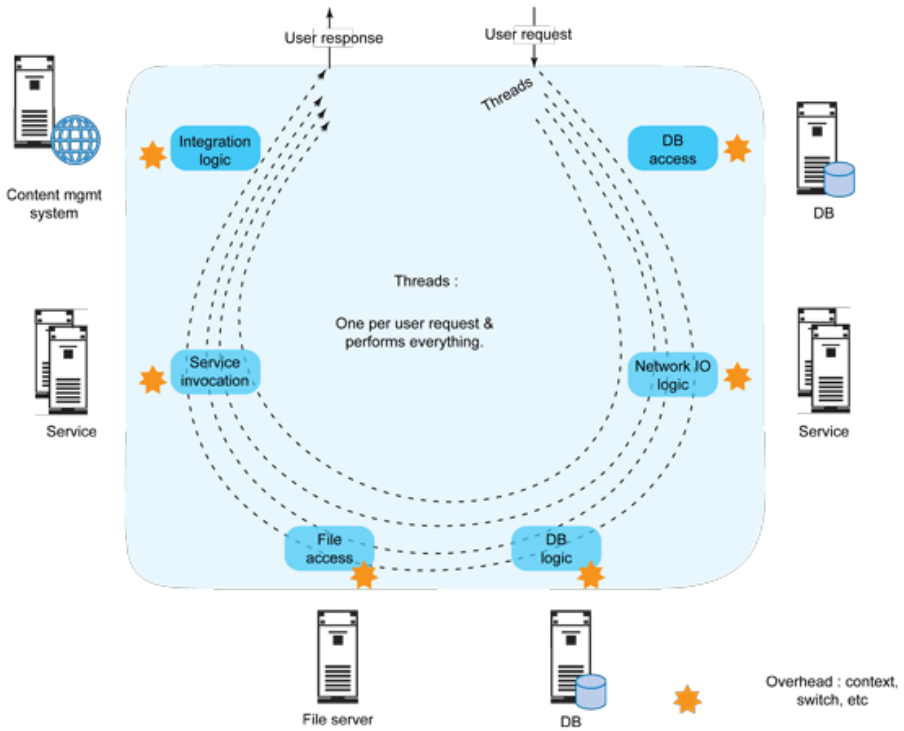
この状況は、ヘンリー・フォード (Henry Ford) による組立ラインが導入される以前の時代の製造プロセスにたとえることができます。その当時は 1 人のワーカーまたは 1 つのワーカー・チームが 1 つの製品全体を作っていました。組立ラインにより、ワーカーは製造プロセス全体のなかのある特定のサブタスクに集中できるようになりました。また、ワーカーが製品を製造するための

ある工程から次の工程へ移るために費やすはずであった時間が節約され、生産性は何倍にも高まりました。

今日の世界では、組立ラインはファーストフード店の注文処理にたとえることができます。ファーストフード店の注文処理では、あらかじめ決められた人数のワーカー (各ワーカーは一連のサブタスクに特化しています) が注文を処理します。各ワーカーは全体の作業の一部のみを実行します。あるワーカーの担当作業が終了すると、完成半ばの商品は流れ作業の次のワーカーに渡され、最終的な商品になるまでそれが続きます。対照的に、各ワーカーが一度に 1 人の顧客の注文を最初から最後まで処理するシステムを考えてみてください。どちらも注文を処理する上では有効な方法ですが、ファーストフード方式の方がより生産的です。1 人のワーカーが 1 つの注文をすべて処理すると、そのワーカーは実際に商品を作るための時間以外にも、作業ごとに場所を移動することに時間をかけすぎてしまいます。複数のワーカーの間を動き回ることで、場所の競合が生じて時間が余計にかかるなどの別の問題も発生します。

今度は、最近の JEE アプリケーション・サーバーがユーザー・リクエストを実行する場合を考えてください。サーバーは 1 つのユーザー・リクエストに対して専用のスレッドを割り当てます。図 3 に示すように、そのスレッドは、ロギングからデータベースとのやり取り、Web サービスの呼び出し、ネットワークとの通信、ロジックの計算処理、等々に至るまですべての命令を実行します。

図 3. スレッドのフロー



コントローラー、モデル、ビュー、ファサード、そしてその他の層に分割することで、いかに適切にソース・コードをモジュール化しても、そのソース・コードを実行するのは 1 つのスレッドです。シーケンシャルに実行するこの形式では、コンテキスト・スイッチなど、ハードウェア・リソースの競合が内部で大量に発生します。

まとめ

マルチスレッドは、CPU リソースを可能な限り効率的に活用するための優れた方法です。しかしシステムの進化に伴い、開発コミュニティーや OS コミュニティーはアプリケーション・レベルの並列処理にもマルチスレッドを使用するようになりました。そして、アプリケーション開発コミュニティーは、スレッド・ベースのプログラミングを使用してすべてのアプリケーション・ロジックをシーケンシャルに実行するようになりました。マルチコアの CPU が一般的になり始めており、コアの数が次第に増加していることから、明確にスレッドを使用したシーケンシャルなプログラミングは従来よりも効率の悪いものになりました。

マルチコア・ハードウェア上で実行されるスケーラブルでハイパフォーマンスのアプリケーションのためには、相互に依存する複数のワーク・ユニットのスライスにアプリケーション・ロジックを分割し、それらを透過的に連結する並列処理の手法 (1 つのスレッドによって明示的にワーク・ユニットを結合する手法とは対照的です) が必要です。それにより、個々のワーク・ユニットを効率的に実行できるようになります。

組立ラインが製造プロセスに革命を起こし、すべてのプロセスに効率化がもたらされたのと同じように、将来の適切なプログラミング・モデルはアプリケーション・ソフトウェアの設計方法を変えるはずです。そうした抽象化モデルの 1 つであるアクター・ベースのプログラミング (「参考文献」を参照) は、アプリケーション全体を複数のスライスに分割することで、コアをこれらのスライスに割り当てて、効率的な方法で並列に実行させることができます。

お断り

この記事で紹介した意見や見解は完全に著者自身のものであり、必ずしも著者が所属する企業の意見や見解ではありません。

謝辞

貴重な助言をくださった私の同僚である Jesus Bello 氏と Olga Raskin 氏に感謝いたします。

著者について

Vasudevan Thiyagarajan

Vasu Thiyagarajan は IT アーキテクトとして Royal Caribbean Cruises Ltd. に勤務しています。以前はバンガロールの IBM ソフトウェア研究所に勤務していました。彼は主に、高度にスケーラブルなシステムや分散コンピューティングの構築に関心を持っています。