Java Streams, Part 4: 並行処理から並列処理へ

並列化効率に影響を与える要因を理解する

Brian Goetz
Java Language Architect
Oracle

2016年 10月 27日

この全 5 回からなるシリーズ「Java Streams」の第 4 回では、並列処理の効率性を決定付ける要因を特定し、歴史的および技術的観点から説明します。並列実行を目的とした Streams ライブラリーを最適に利用するには、これらの決定要因を理解することが基礎となります (次回の記事で、今回説明する原則をそのまま Streams に当てはめます)。

このシリーズの他の記事を見る

この全 5 回からなるシリーズ「Java Streams」の第 4 回では、並列処理の効率性を決定付ける要因について、歴史的および技術的観点から説明します。並列実行を目的とした Streams ライブラリーを最適に利用するには、これらの決定要因を理解することが基礎となります (次回の記事で、今回説明する原則をそのまま Streams に当てはめます)。

コアが高速化されるのではなく数が増えていた時代

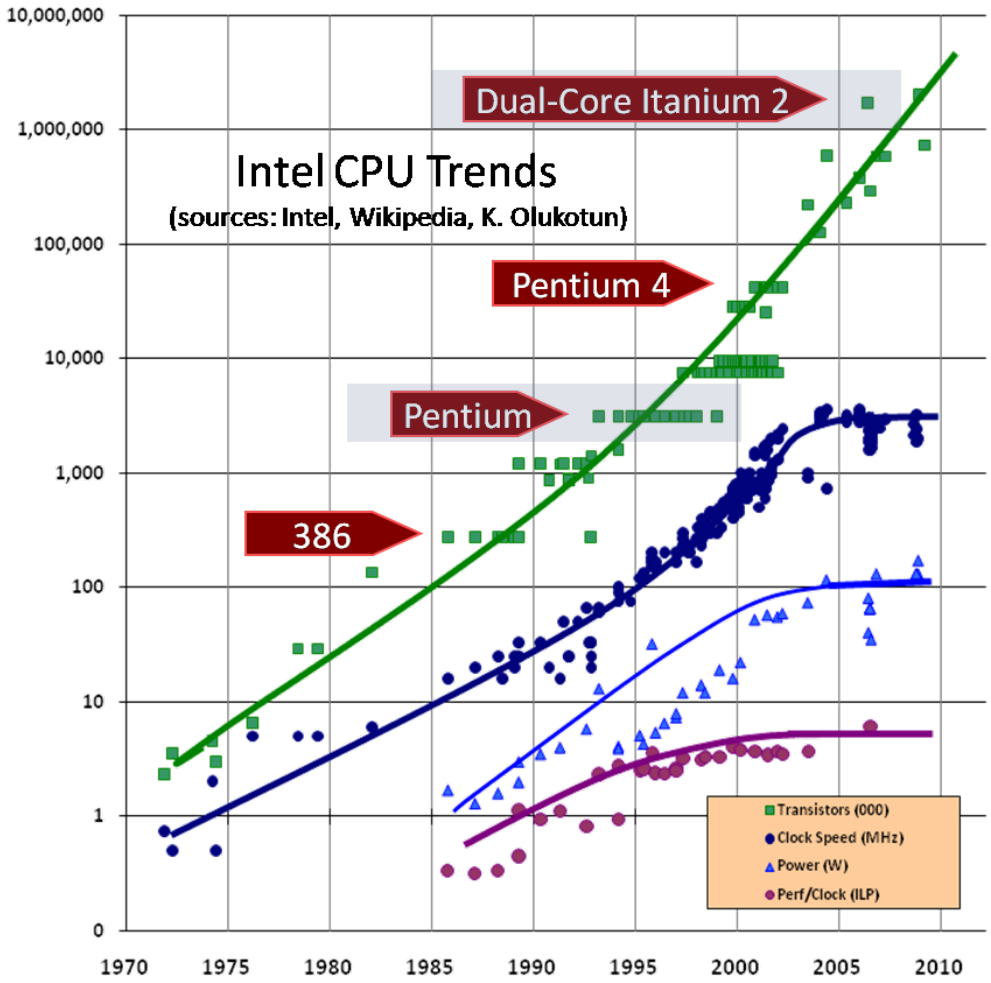
2002 年頃までに、チップの設計者がパフォーマンスを飛躍的に向上させるために使ってきた手法は底をつき始めてきました。電力消費量や放熱などのさまざまな理由から、クロック速度の大幅な向上は実現困難であり、サイクルごとの処理量を増やすための (命令レベルの並列処理) 手法も収穫逓減点に近づいていました。

このシリーズについて
`java.util.stream` パッケージを使用すると、コレクションや配列などのさまざまなデータ・ソースに対する一括処理を、おそらく並列でも実行できる、簡潔な宣言型の処理として表現することができます。このシリーズでは、Java 言語アーキテクトである Brian Goetz が、Streams ライブラリーについて包括的に解説し、このライブラリーを最大限活用するにはどのようにするのかを説明します。

ムーアの法則では、1 つのダイに載せられるトランジスターの数は約 2 年ごとに倍増すると予測しています。2002 年にチップの設計者が「周波数の壁」に突き当たったわけは、このムーアの法則の勢いが失速したからではありません。その当時も、トランジスターの数は着実かつ勢いよく増加し続けていました。チップの設計者はこの増え続けるトランジスター割り当て量を、**コアの高速化**につなげるための力を出し切った一方で、その割り当て量の増加をダイあたりの**コア数**

の増加に充てることはまだ可能でした。この傾向は、Intel プロセッサーのデータを対数スケールで表した図 1 に示されています。一番上のまっすぐに伸びた線は、トランジスター数の急増を示しています。その一方で、クロック速度、電力消費、命令レベルの並列処理のそれぞれを表す線は、いずれも 2002 年あたりで明らかに横ばいになっています。

図 1. Intel CPU のトランジスター数と CPU パフォーマンス (出典: Herb Sutter)



コアの数が増えれば、電力効率の向上につながります (アクティブに使用されていないコアは単独で電源を切ることができるため)。けれども、すべてのコアを有用な処理によってビジー状態に保てるのでなければ、コア数が多いからといって、必ずしもプログラムのパフォーマンスが向上す

るわけではありません。確かに、最近のチップのコア数は、ムーアの法則で予測されているほど多くなっていません。その主な理由は、最近のソフトウェアでは多数のコアをコスト効率良く活用できないからです。

並行処理から並列処理へ

コンピューティングの歴史のほぼ全体を通して、並行処理の目標はほとんど変わっていません。それは、CPU 使用率を上げてパフォーマンスを向上させることです。ただし、そのための手法 (そしてパフォーマンスの尺度) は変化しています。単一コア・システムの時代、並行処理は主に非同期処理に関連して理解されていました。アクティビティーが I/O の完了を待っている間、CPU を解放できるようにすることに目が向けられていたのです。非同期処理によって、応答性とスループットの両方を向上させられる可能性があります。具体的には、バックグラウンド・アクティビティーの進行中に UI をフリーズさせなければ応答性が向上し、あるアクティビティーが I/O の完了を待っている間、別のアクティビティーが CPU を使用できるようにすればスループットが向上するからです。一部の並行処理モデル (例えば、アクターと CSP (Communicating Sequential Process)) 内では、並行処理モデル自体がプログラムの構造に組み入れられていましたが、ほとんどのところ (良し悪しはともかく)、並行処理はパフォーマンスを向上させなければならない場合にのみ適用されていました。

> “ 並列処理の効率性に影響を与える要因には、解決する問題そのもの、問題を解決するために使用するアルゴリズム、タスクの分解とスケジューリングに対するランタイム・サポート、そしてデータ・セットのサイズおよびメモリー局所性が含まれます。 ”

マルチコア時代に入ると、並行処理は主に、ワークロードを独立したタスク (ユーザー要求など) に大まかに分解するために適用されるようになりました。その目的は、複数の要求を同時に処理することによってスループットを向上させることです。同じ頃、Java ライブラリーにはタスク・ベースの並行処理に最適なツールとして、スレッド・プール、セマフォー、Future などが加わりました。

コア数が増加を続ける一方で、「自然に発生する」タスクの数は、すべてのコアをビジー状態しておくために必要な数に達しないことがあります。コア数がタスクの数を超えるという状況の中、別のパフォーマンス目標が注目を集めるようになってきました。それは、複数のコアを使用して 1 つのタスクをより短時間で完了させることにより、遅延を削減することです。すべてのタスクがこの類の分解に適しているわけではありません。このように分解すると功を奏するタスクは、大規模なデータ・セット上で同じ処理が繰り返し行われる、データ集約型のクエリーです。

残念ながら、「並行性 (concurrency)」と「並列性 (parallelism)」という 2 つの用語には標準的な定義がなく、同じ意味で (誤って) 使われることがよくあります。歴史的に、並行性という用語は**プログラム**の特性を表現していました。具体的には、協力し合う計算アクティビティーのやりとりとしてプログラムがどの程度構造化されているかという特性です。一方、並列性という用語は、どの程度同時に物事が行われるのかという、プログラムの**実行**特性を表現していました (この定義の下では、並行性は**潜在的**並列性であると言えます)。この区別は、真の並行実行が主に理論的懸念であった頃は役立ちましたが、そのうち意味を持たなくなってきました。

最近のカリキュラムでは、並行性については、共有リソースへのアクセスを正しく効果的に制御することであると説明し、並列性については、問題をより短時間で解決するために、より多くのリソースを使用することであると説明しています。スレッド・セーフなデータ構造体を構成するのは、並行処理の分野になります。スレッド・セーフは、ロック、イベント、セマフォー、コルーチン、ソフトウェア・トランザクション・メモリー (STM) などのプリミティブによって可能になるためです。一方、並列処理では、区分化やシャーディングなどの手法を使用して、1 つのタスクに対して複数のアクティビティーを調整せずに進行させることを可能にします。

この区別は、なぜ重要なのでしょうか？結局のところ、並行処理と並列処理には、複数のことを同時に行うという共通の目標があります。けれども、その目標を達成するのがどれだけ容易であるかという点では、この 2 つの間に大きな違いがあります。ロックなどの調整プリミティブを使用して並行処理コードを正しく作成するのは難しく、エラーが発生しやすく、しかも不自然です。それぞれのワーカーが問題の異なる担当部分を処理するような仕組みにするのは、比較的単純であり、安全性が高く、そしてより自然なことです。

並列処理

並列処理の構造は、大抵は単純なものですが、これを使用すべき「状況」を把握するのはそう簡単ではありません。並列処理とは、厳密には最適化であり、解をより短時間で得ることを目的に、特定の計算により多くのリソースを使用するという選択を意味します。けれどもその一方で、計算を順次実行するという別の選択もあります。残念ながら、より多くのリソースを使ったとしても、解を得るまでの時間を短縮化できるとは限りません。まったく短縮できないこともあります。さらに、並列処理で追加のリソースを消費してもメリットがなければ (あるいは悪影響を及ぼすとしたら)、並列処理を使用すべきではありません。当然、並列処理によるメリットは状況に大きく依存するため、すべてに共通する 1 つの答えはありませんが、並列処理が特定の状況内で効果を発揮するかどうかを評価するためのツールはあります。それは、分析、測定、そしてパフォーマンス要件です。

この記事では主に分析に焦点を当て、並列化に適している計算のタイプと、そうではないタイプを探ります。並列処理によって処理の高速化を図れると確信する理由がない限り、順次実行を選ぶことを経験上お勧めします。また、高速化が実際に重要であるかどうかは、パフォーマンスの要件次第です (多くのプログラムはすでに十分高速なので、最適化に労力を費やすよりも、有用性や信頼性の向上などといった、より大きな価値をもたらすアクティビティーに取り組むほうが賢明な場合もあります)。

「高速化 (speedup)」と呼ばれる並列処理の有効性を測る尺度は、単に並列実行時間と順次実行時間の比に過ぎません。(高速化できるという前提で) 並列処理を選択するということは、よく考えた上で、CPU と電力消費より時間を優先することを意味します。並列実行には必ず、順次実行よりも多くの作業が伴います。それは、問題を解決するだけでなく、問題の分解、部分問題を記述するタスクの作成と管理、それらのタスクのディスパッチと待機、結果のマージが必要になってくるからです。したがって、並列実行は常に順次実行より「遅れ」をとって開始され、その最初の遅れを規模の経済によって取り戻すことを期待することになります。

並列処理が順次実行よりも有利な選択となるには、いくつかの条件があります。そもそも、問題自体が並列処理ソリューションを適用できるものでなければなりません (すべての問題に並列処理を適用できるわけではありません)。次に、その問題に備わっている並列性を活用するソリュー

ションの実装が必要です。また、並列処理を実装するために使用する手法に、問題にあてがうサイクルを無駄にするようなオーバーヘッドが伴わないことを確実にする必要もあります。さらに、高速化に達するために必要となる規模の経済を実現できるだけの十分な量のデータが必要です。

並列処理の活用

すべての問題に同じように並列処理を適用できるわけではありません。問題の例として、関数 `f` があるとします。これを計算するにはコストがかかることが想定されるため、次のように `g` を定義します。

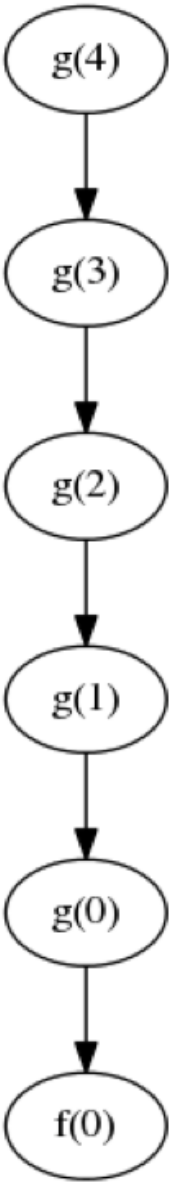
```
g(0) = f(0)
g(n) = f( g(n-1) ), for n > 0
```

`g` の並列処理アルゴリズムを実装すれば、g の高速化を測定できますが、その必要はありません。この問題が順次処理であることは一目瞭然だからです。順次処理であることを確認するために、`g(n)` を少々作り替えます。

```
g(n) = f( f( ... n times ... f(0) ... )
```

上記のように作成し直すと、`g(n-1)` の計算が完了してからでないと、`g(n)` を計算できないことがわかります。図 2 に、`g(4)` を計算する際のデータフローの依存性を示します。この図を見るとわかるように、`g(n)` の各値は前の値 `g(n-1)` に依存しています。

図 2. 関数 **g** のデータフロー依存性グラフ

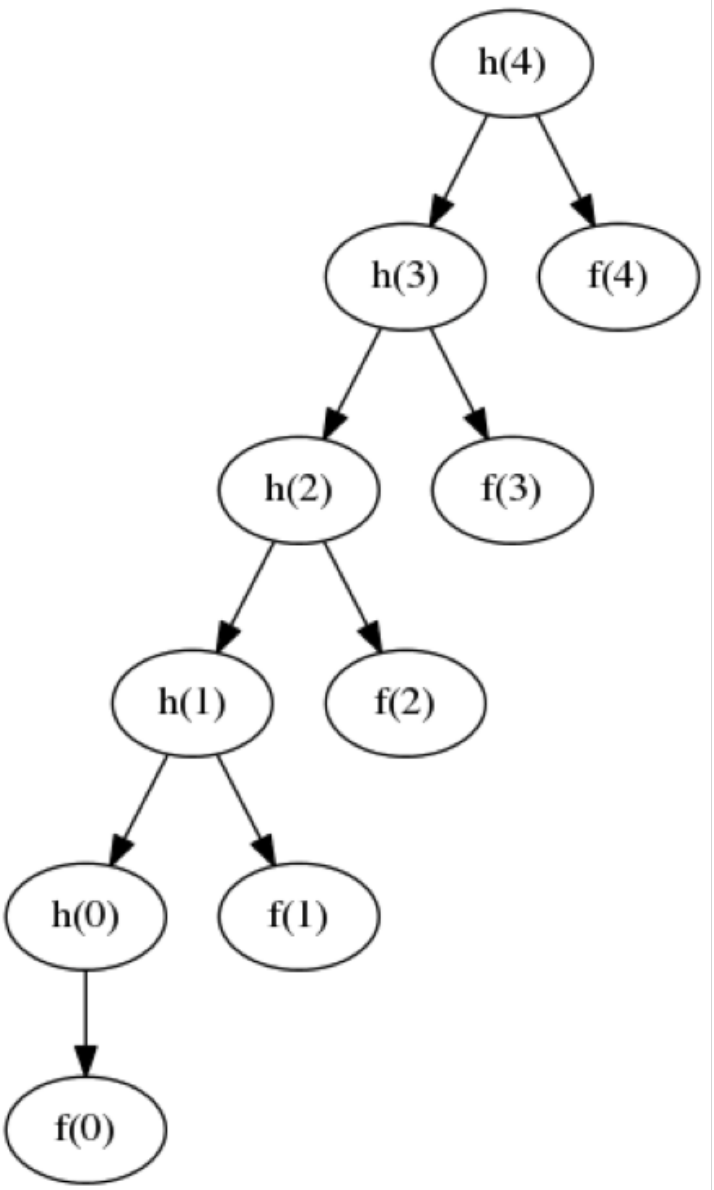


この問題の原因は `g(n)` が再帰的に定義されていることにあると結論したくなるかもしれませんが、それが原因ではありません。やや異なる問題として、以下に記載する関数 `h(n)` の計算を取り上げます。

```
h(0) = f(0)
h(n) = f(n) + h(n-1)
```

`h(n)` を計算するのに当たり前の実装を作成すると、図 3 のデータフロー依存性グラフに示されているように、それぞれの `h(n)` が `h(n-1)` に依存することになります。

図 3. 関数 **h** を単純に実装した場合のデータフロー依存関係グラフ

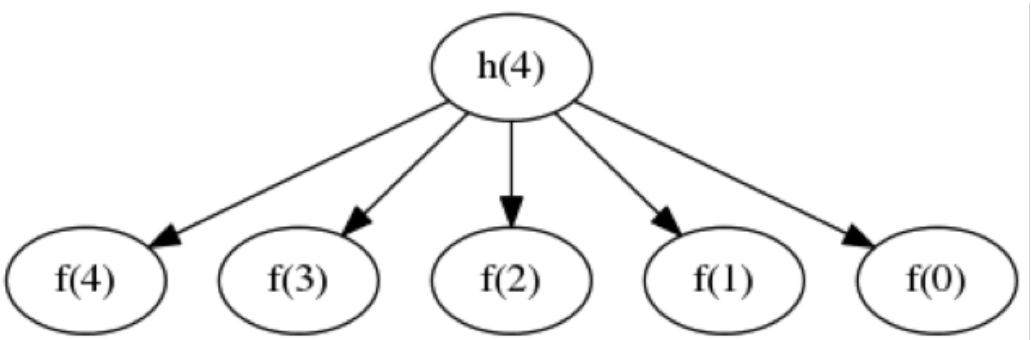


ただし、別の方法を使用して `h(n)` を作り替えると、この問題には並列処理を活用できることがすぐにわかります。以下のように作成し直せば、項のそれぞれを独立して計算した後、それらの結果をまとめることができます (つまり、並列処理を適用できるようになります)。

```
h(n) = f(0) + f(1) + .. + f(n)
```

この結果、図 4 のデータフロー依存性グラフ内で表されているように、各 `h(n)` を独立して計算することができます。

図 4. 関数 h のデータフロー依存性グラフ



以上の例から、2 つのことがわかります。1 つは、同じように見える問題でも、並列処理を活用できるかどうかには大きな違いがあることです。もう 1 つは、並列処理を活用できる問題に対して、「当たり前の」ソリューションを実装するのでは、その並列処理が活用されることにはならないことです。高速化のチャンスを手に入れるには、この 2 つの点を考慮しなければなりません。

分割統治

活用できる並列処理に取り組む際の標準的な手法は、再帰的分解 (recursive decomposition) または分割統治 (divide and conquer) と呼ばれる手法です。この手法では、問題を部分問題に分割していき、順次解決に適した大きさの部分問題になるまで分割を繰り返します。そして部分問題を順次解決した後、それらの部分問題の部分結果を結合して全体的な結果を生成します。

リスト 1 に、疑似コード内での典型的な分割統治ソリューションを記載します。このソリューションでは、並列実行に架空の `CONCURRENT` プリミティブを使用しています。

リスト 1. 再帰的分解

```
R solve(Problem<R> problem) {
    if (problem.isSmall())
        return problem.solveSequentially();
    R leftResult, rightResult;
    CONCURRENT {
        leftResult = solve(problem.leftHalf());
        rightResult = solve(problem.rightHalf());
    }
    return problem.combine(leftResult, rightResult);
}
```

再帰的分解が魅力的なのは、単純だからです。特に、すでに再帰的に定義されているデータ構造体 (ツリーなど) を扱う場合は、有用な手法になります。リスト 1 に記載したような並列処理コードは、幅広いコンピューティング環境にわたって移植できます。このようなコードは単一コアでもマルチコアでも効率的に機能します。また、共有されたミュータブルな状態 (shared mutabble state) を調整する複雑さを懸念する必要もありません。なぜなら、共有されたミュータブルな状態はないからです。したがって、通常はスレッド間で調整しなくても、問題を部分問題に分割して、各タスクが特定の部分問題のデータだけにアクセスするよう手配することができます。

パフォーマンスに関する考慮事項

ここからは、リスト 1 を見本に、並列処理がメリットをもたらすための条件を分析します。分割統治によって、問題の分割と、部分結果の結合という 2 つのアルゴリズム・ステップが追加されますが、そのそれぞれが、多少とも並列処理との親和性を備えることができます。並列化効率に影響を与える別の要因として、並列処理のプリミティブ自体の効率性も挙げられます (リスト 1 の疑似コード内で使用されている架空の `CONCURRENT` メカニズムを参照)。さらに、サイズおよびメモリー局所性というデータ・セットの 2 つのプロパティーも、パフォーマンスに対する影響要因に加わります。これらの条件について、順に説明します。

問題によっては、分解する余地がまったくないものもあります。「並列処理の活用」セクションで取り上げた `g(n)` 関数は、その一例です。問題を分解できるとしても、それにはコストが伴う場合があります (少なくとも、問題を分解するには、部分問題の記述を作成する必要があります)。例えば、クイックソート・アルゴリズムの分解ステップでは、ピボットとするポイントとして、問題のサイズにおける O(n) を見つけなければなりません。ピボットでデータを検査し、場合によってはすべてのデータを更新する必要があるためです。この要件には、問題の区分化ステップより遥かに大きなコストがかかります。例えば「要素の配列の合計を調べる」といった問題の区分化ステップは、O(1) です。つまり「先頭インデックスと終了インデックスの平均を調べる」だけに過ぎません。また、問題を効率的に分解できるとしても、2 つの部分問題のサイズに大きな差がある場合、活用できる並列処理を十分に活用していることにはなりません。

2 つの部分問題を解決したら、それらの部分結果を結合する必要もあります。「重複する要素を削除する」という問題の場合、結合ステップで 2 つのセットをマージしなければなりません。その場合、結果のフラットな表現が必要だとしたら、このステップは O(n) になります。一方、「配列の合計を調べる」という問題での結合ステップは、「2 つの数値を合計する」ことなので、O(1) になります。

並列処理で実行されるタスクの管理にも、効率性を損なう原因がいくつか考えられます。例えば、あるスレッドから別のスレッドにデータを渡す際に伴う遅延、共有されるデータ構造体に対するコンテンションのリスクなどです。フォーク/ジョイン・フレームワーク (粒度の細かい計算集約型のタスクを管理するために Java SE 7 内で導入されました) は、並列ディスパッチにおける多くの一般的な非効率性の原因を最小限に抑えるよう意図されています。`java.util.stream` ライブラリーは、このフォーク/ジョイン・フレームワークを使用して並列実行を実装します。

最後に、データについて考えなくてはなりません。データ・セットが小さいと、並列実行が課すスタートアップ・コストにより、どのような類の高速化でも、それを導くのは困難です。同じように、メモリー局所性に欠けるデータ・セット (配列とは対照的に、ポインター・リッチなグラフなどのデータ構造体) の場合、メモリーが制約された最近の典型的なシステムを使用する場合は、より短時間で解を得られるよう複数のコアを効率的に使用して並列で実行したとしても、多数のスレッドがキャッシュからのデータを待機するだけの結果になりがちです。

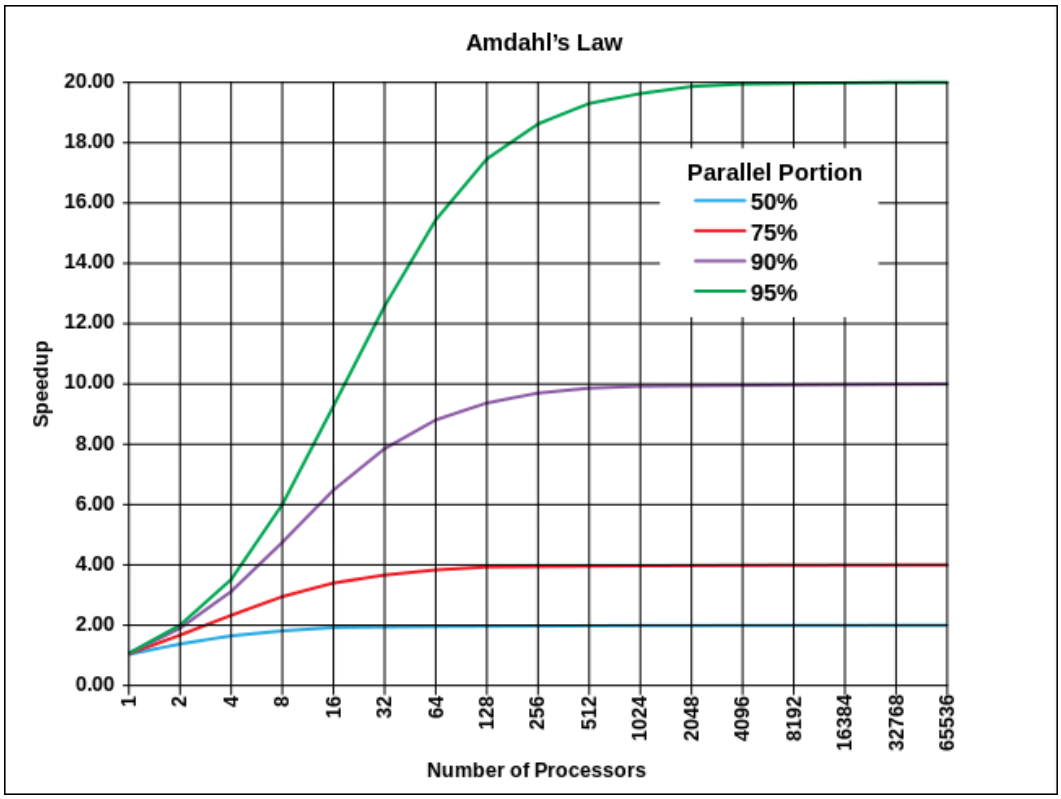
これらの要因のそれぞれが高速化を阻み、場合によっては高速化どころか処理速度を低下させる原因になることさえあります。

アムダールの法則

アムダールの法則は、計算の中で順次に実行される部分が、並列処理による高速化を制限する仕組みを表したものです。ほとんどの問題には、順次実行部分 (serial fraction) と呼ばれる、並列化できない処理がある程度あります。例えば、ある配列のデータを別の配列にコピーする場合、コピーするための処理は並列化できますが、ターゲットの配列の割り当て (本質的に順次) は、データをコピーする前に行われていなければなりません。

図 5 に、アムダールの法則による影響を示します。図中では、使用できるプロセッサーの数を関数として、並列実行部分 (順次実行部分の補完) を変えた場合に、アムダールの法則で予測される最大限の高速化がさまざまな曲線で表されています。例えば、並列実行部分が 5 (問題の半分を順次実行しなければならないことを意味) の場合、無制限の数のプロセッサーを使用できるとしても、アムダールの法則によると、期待できる最大限の高速化は 2 倍にとどまります。なぜなら、直感的に理解できるように、並列処理によって並列化可能な部分のコストをゼロにまで削減したとしても、問題の半分は順次実行しなければならないためです。

図 5. アムダールの法則 (出典: Wikimedia Commons)



アムダールの法則によって暗黙に定義される、処理の一部分を完全に順次実行しなければならない場合でも、残りの部分は完全に並列化できるというモデルはあまりにも単純ですが、並列処理を妨げる要因を理解する上では役立ちます。より効率的な並列処理アルゴリズムを作り出せるかどうかを左右する大きな要因は、順次実行部分を特定し、削減できるかどうかです。

アムダールの法則に別の解釈を適用してみましょう。例えば、コア数が 32 のマシンがあるとします。その場合、並列計算をセットアップするために費やす全サイクルのうち、31 のサイクルは問

題の解決に適用できません。問題を 2 つに分割したとしても、常に 30 のサイクルは無駄にされることになります。すべてのプロセッサーをビジー状態に維持できるだけの処理に分割できて、しかも短時間で完全に稼働中の状態にする (またはその状態を長時間維持する) ことができない限り、十分な高速化を達成するのは困難です。

第 4 回のまとめ

並列処理は、高速化を図るために、より多くの計算リソースを使用するというトレードオフです。理論上は、N 個のコアを使用することで処理を N 倍高速化できますが、実際は、この目標には遥かに及びません。並列処理の効率性に影響を与える要因には、解決する問題そのもの、問題を解決するために使用するアルゴリズム、タスクの分解とスケジューリングに対するランタイム・サポート、そしてデータ・セットのサイズおよびメモリー局所性が含まれます。シリーズ「Java Streams」の第 5 回では、これらの概念を Streams ライブラリーに当てはめて考え、一部のストリーム・パイプラインがどのようにして他のストリーム・パイプラインよりも優れた並列化を実現しているのか、およびなぜ優れているのかを明らかにします。

関連トピック： The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software ムーアの法則 アムダールの法則 Java並行処理プログラミング —その「基盤」と「最新API」を究める— (Brian Goetz 他著、ソフトバンククリエイティブ) java.util.stream のパッケージ・ドキュメント Java fork-join ライブラリー

著者について

Brian Goetz

Brian Goetz is the Java Language Architect at Oracle and has been a professional software developer for nearly 30 years. He is the author of the best-selling book Java Concurrency In Practice (Addison-Wesley Professional, 2006).