Java Streams, Part 5: 並列ストリームのパフォーマンス

並列処理を目的にストリーム・パイプラインを最適化する

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」の最終回となるこの記事では、前回の記事を引き継いで、並列処理の効率性に影響を与える要因について説明し、それらの概念を Streams ライブラリーに当てはめて考えます。一部のストリーム・パイプラインが他のパイプラインよりも並列化になぜすぐれているのかを明らかにし、独自のストリーム・コードを使用して並列化効率をどのように分析するのかを紹介します。

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、並列処理の効率性に影響を与える要因について説明しました。具体的には、問題の特性、ソリューションを実装するために使用するアルゴリズム、並列実行するタスクのスケジューリングに使用するランタイム・フレームワーク、そしてデータ・セットのサイズとメモリー・レイアウトが影響要因として挙げられます。今回の記事では、これらの概念を Streams ライブラリーに当てはめて、一部のストリーム・パイプラインが他のストリーム・パイプラインよりも並列化に優れているのはなぜなのかを考えます。

並列ストリーム

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。

第 3 回で説明したように、ストリーム・パイプラインは、ストリーム・ソース、ゼロ個以上の中間処理、そして単一の終端処理からなります。ストリーム・パイプラインを実行するには、中間処理と終端処理を実行する「マシン」を構成し、ストリーム・ソースの要素をマシンに取り込みます。ストリーム・パイプラインを並列で実行する場合は、`Spliterator` の `trySplit()` メソッドを使った再帰的分解によって、ソース・データをセグメントに区分します。その結果作成されるのは、二分計算木です。このツリーの各リーフ・ノードがソース・データのセグメントに対応し、それぞれの内部ノードが問題をサブタスクに分割したポイントに対応します。並列実行での

2 つのサブタスクの結果は、結合する必要があります。順次実行の場合はデータ・セット全体に対して 1 つのマシンを構成しますが、並列実行ではソース・データのセグメントごとに 1 つのマシンを構成し、各リーフの部分結果を生成します。部分結果を生成した後はツリーの上へと進み、部分結果を 1 つの結果にマージします。どのようにマージするのかは、終端処理に固有のマージ関数に応じて異なります。例えば、終端処理が `reduce()` の場合、集約に使用された二項演算子がマージ関数としても使用されます。`collect()` の場合、`Collector` には、結果コンテナーを別のコンテナーにマージするために使うマージ関数があります。

前回の記事では、並列実行の効率性を損なわせる可能性がある、以下の 7 つの要因を特定しました。

- ソースを分割するのに大きなコストがかかる、または均一に分割できないこと。
- 部分結果をマージするのに大きなコストがかかること。
- 並列処理を十分に活用できない問題であること。
- データ結果のレイアウトが原因で、アクセスの局所性が乏しいこと。
- 並列処理のスタートアップ・コストを補うだけの十分な量のデータがないこと。

ここからは、上記の考慮事項が並列ストリーム・パイプラインを使用した場合にどのように現れてくるのかに目を向けて、それぞれの考慮事項について考えます。

ソースの分割

並列ストリームの場合は、`Spliterator` の `trySplit()` メソッドを使用してソース・データのセグメントを 2 つに分割します。計算ツリーの各ノードは 2 つに分割されたそれぞれの部分に対応するため、二分木が形成されます。このツリーはバランスがとれたものであり (つまり、それぞれのノードがまったく同じ量の処理を表すこと)、分割のコストもゼロであることが理想です。

実際に達成することはできませんが、ソースによっては、この理想にかなり近づいたツリーが生成されます。その好例は、配列です。配列のセグメントは、配列のベース参照と、開始セグメントと終了セグメントの整数オフセットを使用して記述することができます。このセグメントを 2 つの均等なセグメントに分割するコストはほとんどかかりません。セグメントの中間点を計算して、前半部分の記述子を作成し、現在のセグメントの開始インデックスを後半部分の最初の要素に移動すればよいだけです。

リスト 1 に、`ArraySpliterator` 内で `trySplit()` を使用した場合のコードを記載します。配列の分割コストはわずかです (数少ない算術演算と、オブジェクトの作成にコストがかかるだけです)。さらに、配列は均等に分割できます (したがって、バランスのとれた計算ツリーになります)。これらの有利な特性は、`ArrayList` の `Spliterator` にもあります (しかも、配列を分割するときに、すべての部分セグメントの正確なサイズも既知となるため、一部のストリーム・パイプライン内でのコピー処理を最適化できます)。

リスト 1. ArraySpliterator.trySplit() の実装

```
public Spliterator<T> trySplit() {
    int lo = index, mid = (lo + fence) >>> 1;
    return (lo >= mid)
           ? null
           : new ArraySpliterator<>(array,
                                    lo, index = mid,
                                    characteristics);
}
```

一方、分割しにくいソースとしては、リンク付きリストが挙げられます。リンク付きリストの中間点を調べるには、一度に 1 つのノードずつ、リストの半分をトラバースしなければならないため、分割にコストがかかります。不均一な分割を受け入れて分割コストを抑えることもできますが、そうしたとしても大して役には立ちません。極端な場合、異様にバランスの崩れた (右側に偏った) ツリーになり、分割された部分セグメントの一方に最初の要素だけが含まれ、もう一方にリストの残りの要素がすべて含まれる結果になります。そこで、O(lg n) 分割ではなく、O(n) 分割を使用すると、今度は O(n) 結合ステップが必要になってきます。したがって、作成するのに非常にコストがかかる計算ツリーを取るか (アムダールの法則での順次実行部分が増えるため、並列性が限られます)、比較的並列処理に対応するものの、(バランスがとれていないために) 結合コストが大きい計算ツリーを取るかという、苦しい選択肢しか残されません。そうは言っても、各ノードに対して実行する処理にあまりにもコストがかかる場合は、リンク付きリストを使用して並列処理を実現することも不可能ではありません (「NQ モデル」を参照)。

二分木 (`TreeMap` など) とハッシュ・ベースのコレクション (`HashSet` など) は、リンク付きリストよりも分割しやすいとは言え、配列ほどではありません。二分木を 2 つに分割するコストはわずかであり、比較的バランスのとれたツリーであれば、生成される計算ツリーもバランスがとれたものになります。私たちはこのことから、各リンク付きリストを 1 つのバケットとした、バケットからなる配列として `HashMap` を実装しています。このハッシュ関数によってバケット全体に均等に要素を分散させられるとしたら、コレクションを比較的上手く分割できます (これは、単一のバケットに辿りつくまでの話です。その後は、リンク付きリストを分割することになります。理想的には、小さいリストでなければなりません)。ただし、ツリー・ベースのコレクションとハッシュ・ベースのコレクションはどちらも一般に、配列のように予測どおりに分割されることはありません。分割された部分セグメントのサイズを予測できないことから、場合によってはコピー処理を最適化できなくなります。

ソースとしてのジェネレーター

すべてのストリームがコレクションをソースとして使用するわけではありません。ストリームによっては、`IntStream.range()` などのジェネレーター関数を使用する場合もあります。コレクションのソースに適用される考慮事項は、ジェネレーターにもそのまま適用されます。

次の 2 つの例に、整数 0 から 99 からなるストリームを生成する 2 つの方法を示します。以下のコードでは、`Stream.iterate` を使用しています。この 3 つの引数を使用するバージョンの `iterate()` は、Java 9 内で導入されました。

```
IntStream stream1 = IntStream.iterate(0, n -> n < 100,
                                      n -> n + 1);
```

以下のコードでは、`IntStream.range()` を使用しています。

```
IntStream stream2 = IntStream.range(0, 100);
```

上記の 2 つの例が生成する結果は同じですが、分割特性は大きく異なります。そのため、並列化効率にも大幅な違いが現れます。

`Stream.iterate()` は `for`-loop と同じように、初期値と 2 つの関数を取ります。一方の関数は次の値を生成するため、もう一方は要素の生成を停止するかどうかを判断するためのものです。このジェネレーターが基本的に順次であることは直感的に明らかです。それは、要素 n-1 が生成されるまでは、要素 n を生成できないためです。したがって、順次ジェネレーター関数の分割には、リンク付きリストを分割する場合と同じ特性が伴うため (高い分割コストまたは極めて不均一な分割のどちらを取るかの苦しい選択になります)、同じように並列性に欠ける結果となります。

その一方、`range()` ジェネレーターは比較的配列と同じように分割できます。介在する要素が計算される前でも、範囲の中間点を簡単に計算することができ、そのコストもわずかです。

Streams ライブラリーの設計は、関数型プログラミングの原則に多大な影響を受けていますが、上記の 2 つのジェネレーター関数の間での並列化効率の特性における違いは、関数型プログラミングの経験者が陥る可能性がある罠を説明しています。関数型プログラミング内で、範囲を生成するのに一般的で自然なイディオムは、イテレーター関数を適用して構成される無限ストリームから、遅延して要素を取り込むことですが、このイディオムは、データ並列処理がまったくの理論的概念であった頃に生まれたものです。関数型プログラマーが、この手法に伴う順次性にすぐに気付かなければ、慣れ親しんだ繰り返し処理のイディオムを気軽に採用しようとするでしょう。

結果の結合

効率的かつ均等に分割できるかどうかに関わらず、ソースを分割することは、並列計算を可能にするには避けられないコストです。分割にかかるコストがわずかである好都合な場合は、早い段階で処理をフォークでき、アムダールの法則で予想された限界に達しないで済みます。

ソースを分割することで必ず必要となるのは、分割された部分セグメントの中間結果を結合することです。リーフ・ノードのそれぞれが入力の担当セグメントに対する処理を完了した後は、ツリーを上に向かって遡りながら結果を結合していきます。

結合処理の中には、コストがわずかなものもあれば (合計による集約など)、かなりのコストがかかるものもあります (2 つのセットのマージなど)。結合ステップに費やされる時間は、計算ツリーの深さに比例します。バランスのとれたツリーの深さは O(lg n) である一方、異様にバランスの崩れたツリー (リンク付きリストや反復型ジェネレーター関数を分割して生成されたツリーなど) の深さは O(n) です。

コストのかかるマージに関するもう 1 つの問題として、最終マージ (最終的に 2 つになった中間結果をマージする段階) は、順次処理として実行されます (他に残っている処理がないため)。このことが原因で、マージ・ステップが O(n) のストリーム・パイプライン (`sorted()` や `collect(Collectors.joining())` などの終端処理を使用するストリーム・パイプライン) では、並列処理の限界に突き当たる場合があります。

> “ 最適化が出現順に縛られる場合、並列実行にかかる時間とスペースのコストに驚かされることでしょう。一般に、順次実行を当たり前の方法で実装すると、入力を出現順にトラバースするという実装になるので、発生順に対する依存性が明らかになることも、高いコストを生じさせることもめったにありません。並列実行の場合、そのような依存性に非常に大きなコストがかかる可能性があります。 ”

処理のセマンティクス

一部のソース (リンク付きリストや反復型ジェネレーター関数など) には本質的に順次性が伴うのと同じく、一部のストリーム処理にも本質的に順次性の側面があり、それが並列処理の障害となります。そのようなストリーム処理に該当するのは、一般に、セマンティクスが出現順の観点から定義されている処理です。

例えば、`findFirst()` 終端処理の結果は、ストリームの最初の要素になります (一般に、この処理はフィルタリングと組み合わせられるため、「条件を満たす最初の要素を検出する」という意味になるのが通常です)。`findFirst()` を順次処理として実装するコストはごくわずかです。単に、何らかの結果が生成されるまで、データをパイプラインにプッシュして、結果が生成された時点でデータのプッシュを停止するだけに過ぎません。並列処理の場合、アップストリーム処理を並列化するのは簡単ですが、サブタスクによって結果が生成されても、そこで終わりというわけではありません。先行するすべてのサブタスクがその出現順で結果を生成するまで待機する必要があるからです (ただし少なくとも、出現順で後のほうに出現するサブタスクはキャンセルできます)。並列実行では分解およびタスク管理のコストをすべて払わなければなりませんが、それによってメリットを得られる見込みはあまりありません。その一方、終端処理 `findAny()` は、並列による高速化を達成する可能性が大いにあります。それはこの処理が、一致を探す間すべてのコアをビジー状態にして、一致が見つかった時点で直ちに終了できるからです。

セマンティクスが出現順に関連付けられている終端処理としては、`forEachOrdered()` もあります。この処理でも、中間処理を完全に並列化できることがよくありますが、最後に適用可能なステップは順次処理として実行されます。それとは対照的に、`forEach()` 終端処理は、出現順による制約はありません。要素が使用可能になったら、各要素に適用可能なステップを、いつでも、どのスレッド内でも実行できます。

`limit()` や `skip()` などの中間処理も、出現順の制約を受ける場合があります。`limit(n)` は、入力ストリームを最初の n 個の要素で切り捨てる処理です。`findFirst()` と同じように、要素があるタスクによって生成された後、`limit(n)` ではそれらの要素をパイプラインの残りにプッシュすべきかどうかを判断するために、そのタスクに先行するすべてのタスクをその出現順で待機しなければなりません。また、生成された要素が必要であるかどうかを把握するまで、それらの要素をバッファーに入れる必要もあります (順不同ストリームの場合、`limit(n)` では任意の n 個の要素を選択できるため、`findAny()` と同じく、並列処理に遥かに対応しやすくなります)。

最適化が出現順に縛られる場合、並列実行にかかる時間とスペースのコストに驚かされることでしょう。`findFirst()` および `limit()` を当たり前の方法で実装した順次実装は、単純で効率的であり、スペースのオーバーヘッドはほとんど必要になりませんが、並列実装となると複雑にな

り、かなりの待機とバッファリングが必要になります。一般に、順次実行を当たり前の方法で実装すると、入力を出現順にトラバースするという実装になるので、発生順に対する依存性が明らかになることも、高いコストを生じさせることもめったにありません。並列実行の場合、そのような依存性に非常に大きなコストがかかる可能性があります。

幸い、出現順に対する依存性は、パイプラインに小さな変更を加えることによって排除できることがよくあります。ほとんどの場合、`findFirst()` を `findAny()` で置き換えても正確さを失うことはありません。同様に、第 3 回で説明したように `unordered()` 処理によってストリームを順不同にすると、多くの場合は正確さを失うことなく、`limit()`、`distinct()`、`sorted()`、および `collect()` に伴う出現順に対する依存性を排除できます。

これまで説明してきた、並列処理による高速化を妨げる要因は、累積的に作用することもあります。3 つの引数を使用する `iterate()` ソースは、範囲コンストラクターと比べるとお勧めできない選択肢であるのと同様、`limit()` と 2 つの引数を使用する `iterate()` ソースを組み合わせて使用すると、順次生成ステップを、出現順に影響される処理と組み合わせることになるため、さらにお勧めできない選択肢になります。以下は、並列処理に最も不適切な方法で、整数の範囲を生成する例です。

```
IntStream stream3 = IntStream.iterate(0, n -> n+1).limit(100);
```

メモリー局所性

最近のコンピューター・システムでは、頻繁に使用されるデータを可能な限り CPU の近くに配置するために、高度なマルチレベル・キャッシュを採用しています (文字通り、光の速度を制限要因にするキャッシュです)。データをメイン・メモリーからフェッチするのではなく、L1 キャッシュからフェッチすれば、いとも簡単に速度を 100 倍にすることができます。CPU が次に必要なデータを、より効率的に予測できれば、CPU が計算に費やせるサイクルが増え、データを待機しなければならない時間が短縮されます。

データはキャッシュ・ラインの粒度でキャッシュにページングされます。最近の x86 チップで使用しているキャッシュ・ラインのサイズは 64 バイトです。この見返りを享受できるのは、メモリー局所性 (最近アクセスされた場所に近い場所でメモリーにアクセスする傾向を意味します) に優れたプログラムです。配列を線形に進むということは、優れた局所性があるというだけでなく、プリフェッチによる恩恵も受けることになります。メモリー・アクセスの線形パターンが検出されると、ハードウェアは、すぐに必要になるという前提で、次のキャッシュ・ライン分のメモリーをプリフェッチするようになります。

主流の Java 実装では、メモリー内でオブジェクトのフィールドと配列の要素を隣接させて配置します (フィールドは必ずしもソース・ファイル内で宣言されている順に配置されるわけではありません)。最後にアクセスしたフィールドまたは配列の要素の「近く」にあるフィールドまたは要素にアクセスするということは、すでにキャッシュに入れられているデータにヒットする可能性が十分にあることを意味します。その一方、他のオブジェクトへの参照はポインターとして表現されるため、オブジェクト参照を逆参照すると、まだキャッシュに入れられていないデータにヒットして、遅延が発生する可能性が高くなります。

プリミティブ型の配列を使うと、最大限の局所性を実現できます。配列参照が最初に逆参照された後、データはメモリー内に連続して保管されるため、データ・フェッチあたりの計算量を最大化することができます。オブジェクト参照の配列は、配列の次の参照をフェッチする際には局所性を発揮しますが、それらのオブジェクト参照を逆参照する際にキャッシュ・ミスを発生させるリスクがあります。同様に、複数のプリミティブ型フィールドが含まれるクラスはメモリー内でフィールドを別のフィールドの近くに配置する一方、多数のオブジェクト参照が含まれるクラスでは、その状態にアクセスするために多数の逆参照が必要になります。突き詰めるところ、データ構造体がポインター・リッチになればなるほど、そのようなデータ構造体をトラバースしてメモリー・フェッチ単位に配置する負担が大きくなるため、計算時間 (CPU がデータを待機する時間が加算されるため) と並列処理 (多数のコアがメモリーから同時にフェッチすると、メモリーからキャッシュにデータを転送するために使用できる帯域幅に負担がかかるため) の両方に悪影響を与える恐れがあります。

NQ モデル

並列処理によって高速化を実現できるかどうかを判断するために考慮すべき最後の要因は、使用可能なデータの量と、データ要素ごとに実行できる計算の量です。

並列分解について最初に説明した中で、セグメントの問題を順次処理で解決するには効率的でなくなるまでの大きさにソースを分割するという概念を強調しました。どれだけ小さいセグメントでなければならないかは、解決する問題と、特に要素ごとに行われる処理の量に依存します。例えば、文字列の長さを計算するための処理は、文字列の SHA-1 ハッシュを計算するための処理に比べるとほんのわずかです。要素ごとの処理量が多ければ多いほど、「並列処理を活用するのに十分な大きさ」のしきい値は低くなります。同様に、データの量が多ければ多いほど、「小さすぎ」に関するしきい値に到達しないで分割できるセグメントが多くなります。

NQ モデルは、単純ながらも、並列化効率を測定するのに役立つモデルです。NQ モデルの N はデータ要素の数を表し、Q は要素ごとの処理量を表します。N*Q の積が大きければ大きいほど、並列処理によって高速化できる可能性が高くなります。数値を合計するなど、明らかに Q の値が小さい問題では、高速化を図るには N が 10,000 を超える値でなければなりません。Q の値が大きくなれば、高速化を図るために必要なデータ・サイズはそれだけ小さくなります。

Q の値が大きくなると、並列処理を妨げる障害の多く (分割のコスト、結合のコスト、出現順の重要性など) は軽減されます。`LinkedList` の分割特性には手こずるかもしれませんが、Q の値が大きければ、並列処理によって高速化することは可能です。

まとめ

並列処理によって可能になるのは実行時間の短縮だけです。したがって、並列処理を使用するのは、実際に高速化が達成される場合のみでなければなりません。順次ストリームを使ったコードを開発してテストした後、パフォーマンス要件からさらなる改善が必要であることが推測される場合には、最適化戦略の候補として並列処理を検討してください。最適化の取り組みが逆効果にならないことを確認するには、測定が不可欠ですが、多くのストリーム・パイプラインは、並列処理の有力候補であるかどうかを検査によって判断できます。並列処理による潜在的な高速化の実現を妨げる要因には、分割しにくいソースや不均一に分割されるソース、結合に伴う大きなコスト、出現順への依存性、局所性の乏しさ、データ不足などがあります。その一方、要素ごとの計算量 (Q) が大きければ、これらの不備を補うことができます。

関連トピック： JMH Benchmarking Harness 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).