Javaの理論と実践: フォークを活用する、第 1 回

Java 7 で登場するフォーク/ジョインのフレームワークを使って細粒度並列処理の活用方法を学ぶ

Brian Goetz
Senior Staff Engineer
Sun Microsystems

2007年 11月 13日

Java™ 7 で登場する java.util.concurrent パッケージに追加されているものの 1 つが、フォーク/ジョインのスタイルによる並列分割のフレームワークです。フォーク/ジョインによる抽象化では、さまざまなアルゴリズムを分解し、ハードウェアによる並列処理を効果的に活用するための機構が提供されています。
このシリーズの次回の記事では、メモリー内のデータ構造に対する並列ソートや並列検索の操作を単純化することができる、ParallelArray クラスについて説明します。

このシリーズの他の記事を見る

ハードウェアの傾向によってプログラミングのイディオムが変化する

私達がプログラムを作成するための方法は、言語と、ライブラリー、そしてフレームワークによって決まります。Alonzo Church は 1934年に、既知のすべての計算フレームワークは、そうしたフレームワークで表現できるプログラム・セットの中で等価であることを示しました。しかし実際にプログラマーが作成する一連のプログラムはイディオムによって形成され、このイディオムは言語と、ライブラリー、そしてフレームワークによって決まるプログラミング・モデルで容易に表現することができます。

また一方、その時点で優勢なハードウェア・プラットフォームによって、言語と、ライブラリー、そしてフレームワークを作成するための方法が決まります。Java 言語は最初からスレッドと並行性をサポートしていました。つまり Java 言語には `synchronized` や `volatile` などの同期プリミティブがあり、またクラス・ライブラリーには `Thread` などのクラスが含まれていました。しかし、1995年には適切であった並行プリミティブは、当時のハードウェアの現実を反映したものでした。つまり商用利用できるシステムの大部分は並列処理をまったくサポートしておらず、最も高価なシステムさえ、限定的な並列処理しか提供していませんでした。この当時、スレッドは、並行性ではなく非同期を表現するために主に使われており、そのため一般的に、こうした機構は当時の要求に対しては十分だったのです。

マルチプロセッサー・システムが安くなり始めると、そうしたシステムが提供する、ハードウェアによる並列処理を活用しなければならないアプリケーションが増加しました。またプログラマーは、Java 言語やクラス・ライブラリーで提供される下位レベルのプリミティブを使って並行プログラムを作成する作業が容易ではなく、間違いを起こしやすいことに気付きました。Java 5 では、Java プラットフォームに `java.util.concurrent` パッケージが追加され、並行アプリケーションを構築するための便利なコンポーネント・セットが提供されています (並行コレクションやキュー、セマフォー、ラッチ、スレッド・プールなど)。これらの機構は、比較的粒度の粗いタスクを持つプログラムには適していました。つまりアプリケーションは、利用可能な (少数の) プロセッサーの数よりも並行タスクの数が常に下回るということがないように、作業を分割するだけでよかったのです。アプリケーションは、Web サーバーやメール・サーバー、あるいはデータベース・サーバーでの 1 つのリクエストの処理を作業単位として使うことで、通常はその要求を満たすことができました。そのため、ある程度の並列処理能力を持つハードウェアを活用する上では、こうした機構で事足りていたのです。

今後のハードウェアの傾向は明らかです。ムーアの法則 (Moore's Law) によってクロック・レートが高くなることはないでしょうが、チップ当たりのコア数は増えていくでしょう。ユーザー・リクエストなどの粒度の粗いタスク境界を使って十数個のプロセッサーをビジー状態に維持する方法は容易に想像できますが、この方法を何千ものプロセッサーに拡張することはできないでしょう。そこまで拡張するとトラフィックが短期間、指数関数的に増加するかもしれませんが、やがてハードウェアによる傾向が優勢になります。マルチ・コアの時代に入るのに合わせ、細粒度の並列処理を見つける必要があります。そうでないと、処理の必要な作業が大量にあるにもかかわらず、プロセッサーをアイドル状態に放置する羽目になります。優勢なハードウェア・プラットフォームが移り変わるにつれ、それに追いつくためにソフトウェア・プラットフォームも変わる必要があります。この点に関して、Java 7 には、ある種の細粒度並列アルゴリズムを表現するフレームワークが含まれています。それがフォーク/ジョインのフレームワークです。

細粒度の並列処理を公開する

今日の大部分のサーバー・アプリケーションは、作業単位としてユーザー・リクエストとレスポンスの処理を使っています。サーバー・アプリケーションは通常、利用可能なプロセッサー数よりもはるかに多い並行スレッド、つまりリクエストを実行しています。その理由は、大部分のサーバー・アプリケーションではリクエスト処理にかなりの量の入出力処理が含まれていますが、入出力処理はプロセッサーをあまり必要としないためです。(リクエストはソケットを通して受信されるため、すべてのネットワーク・サーバー・アプリケーションは大量のソケットの入出力処理を行います。また多くのネットワーク・サーバー・アプリケーションは、かなりの量のディスク (あるいはデータベース) の入出力処理も行います。) もし、入出力処理が完了するのを待つために各タスクが 90% の時間を費やすとすると、プロセッサーを完全に利用し続けるには、プロセッサーの数の 10 倍の並行タスクが必要です。プロセッサー数が増加すると、すべてのプロセッサーをビジー状態に保つために十分な並行リクエストがないかもしれません。しかし並列処理を使うことで、もう 1 つのパフォーマンス尺度、つまりレスポンスを得るまでユーザーが待つ時間を改善することは相変わらず可能なのです。

典型的なネットワーク・サーバー・アプリケーションの例として、データベース・サーバーを考えてみてください。データベース・サーバーでリクエストが受信されると、そのリクエストは一連のフェーズを通ります。最初に SQL 文が構文解析され、検証されます。次にクエリー・プランを選択する必要があります。複雑なクエリーの場合、データベース・サーバーは、想定される入

出力操作の数を最小限にするために数多くの候補プランを評価します。クエリー・プランの検索は CPU 負荷の高いタスクです。ある時点で、さらに多くの候補プランを検討しても成果はマイナスになります。しかし評価する候補プランの数が少なすぎると、ほぼ確実に必要以上の入出力処理をすることになります。ディスクからデータを取得した後、得られたデータ・セットに対して、さらに処理が必要かもしれません (クエリーには SUM や AVERAGE などの集約操作が含まれるかもしれず、あるいはデータ・セットをソートする必要があるかもしれません)。そして結果をエンコードして要求側に返す必要があります。

大部分のサーバー・リクエストと同じように、SQL クエリーの処理には計算と入出力の処理が混在しています。CPU パワーをいくら追加しても入出力の処理を完了するための時間を減らすことはできません (ただしメモリーを追加して以前の入出力操作の結果をキャッシュすることで入出力処理の数を減らすことはできます)。しかし、リクエスト処理のうち CPU 負荷の重い部分 (プランの評価やソートなど) を並列処理できるようにすることによって、その部分の処理に必要な時間を短縮することはできます。クエリー・プランの候補を評価する際には、さまざまなプランを並行して評価することができます。つまり、データ・セットをソートする際に、大きなデータ・セットを小さなデータ・セットに分割し、並行して個別にソートしてからマージすることができるのです。こうすることで早く結果を得られるため、(リクエストをサービスするために必要な作業量は増えるかもしれませんが) パフォーマンスに関するユーザーの印象を改善することができます。

分割統治法アルゴリズム

マージ・ソートは分割統治法アルゴリズムの一例です。このアルゴリズムでは、問題を再帰的に副問題に分割していき、副問題に対する解を組み合わせて最終的な結果を得ます。分割統治法アルゴリズムは逐次環境において効果的な場合が多いのですが、並列環境では副問題を並行して解決できるため、より一層効果的です。

典型的な並列型の分割統治法アルゴリズムは、リスト 1 のような形式をとります。

リスト 1. 汎用の分割統治法並列アルゴリズムの擬似コード

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

並列型の分割統治法アルゴリズムは最初に、問題が小さすぎて逐次型ソリューションの方が早く結果を得られることがないか評価します。通常これは、問題のサイズを何らかのしきい値と比較することで行われます。もし並列分割のメリットを生かせるほど問題が大きければ、その問題を 2 つ以上の副問題に分割し、これらの副問題に対して並行してこのアルゴリズムの再帰的な呼び出しを行い、副問題の結果を待ち、そしてそれらの結果を組み合わせます。逐次実行と並列実行のどちらを選ぶか、その理想的なしきい値は、並列タスクの調整コストの関数です。もし調整コストがゼロならば、大量の細粒度タスクによって高い並列処理を実現することができます。調整

コストが低ければ低いほど粒度を細かくすることができ、逐次型の方法に切り替える必要性が低くなります。

フォーク/ジョイン

リスト 1 の例は、存在しない INVOKE-IN-PARALLEL 操作を利用しています。この操作では、現在のタスクが中断されて、2 つのサブタスクが並行して実行されます。現在のタスクはその 2 つのサブタスクが完了するまで待機します。2 つのサブタスクが完了すると、その結果が組み合わされます。この種の並列分割は、よくフォーク/ジョインと呼ばれますが、それはタスクを実行するとタスクがフォーク (分岐) して複数のサブタスクが開始され、(サブタスクの完了を待って) サブタスクがジョインされる からです。

リスト 2 はフォーク/ジョインによるソリューションに適した問題の例として、大規模な配列を検索して最大の値を持つ要素を求めるものです。もちろん、これは非常に簡単な例ですが、フォーク/ジョインの方法は非常にさまざまな種類の検索やソート、データ分析などの問題に適しています。

リスト 2. 大規模な配列から最大の値を持つ要素を選択する

```
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size;

    // constructors elided

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++) {
            int n = numbers[i];
            if (n > max)
                max = n;
        }
        return max;
    }

    public SelectMaxProblem subproblem(int subStart, int subEnd) {
        return new SelectMaxProblem(numbers, start + subStart,
                                    start + subEnd);
    }
}
```

`subproblem()` メソッドが要素をコピーしないことに注意してください。このメソッドは単に配列参照とオフセットを既存のデータ構造の中にコピーするだけです。これはフォーク/ジョインの問題を実装するための典型的な方法です。なぜなら、問題を再帰的に分割するプロセスによって、新しい `Problem` オブジェクトが大量に作成される可能性があるからです。この場合、検索タスクは検索対象のデータ構造をまったく変更しません。そのため、基礎となる、各タスクのデータ・セットのプライベート・コピーを保持する必要がなく、従ってコピーのための余分なオーバーヘッドが発生しません。

リスト 3 は、Java 7 に含まれる予定のフォーク/ジョインのパッケージを使った、`SelectMaxProblem` のソリューションを示しています。このパッケージは JSR 166 Expert Group で jsr166y というコード名でオープンに開発されているので、このパッケージを単独でダウンロー

ドし、Java 6 以降で使用することができます。(これは最終的に、`java.util.concurrent.forkjoin` パッケージの中に入ります。) `invoke-in-parallel` という操作は `coInvoke()` メソッドで実装されています。このメソッドは複数のアクションを同時に呼び出し、そしてすべてのアクションが完了するまで待機します。`ForkJoinExecutor` はタスクを実行するために設計されたという意味で `Executor` に似ていますが、計算負荷の重いタスク専用に設計されており、同じ `ForkJoinExecutor` が別のタスクを処理するのを待機する以外にはブロックされないという点が異なります。

フォーク/ジョインのフレームワークは `ForkJoinTasks` のいくつかのスタイルをサポートしており、その中には明示的な完了を必要とするものや周期的に実行されるものがあります。ここで使用している `RecursiveAction` クラスは、何も結果を生成しないタスクに対して並列再帰分割を行うスタイルを直接サポートしており、`RecursiveTask` クラスは、結果を生成するタスクに対して同様の問題に対処します。(他のフォーク/ジョインのタスクのクラスには、`CyclicAction` と、`AsyncAction`、そして `LinkedAsyncAction` があります。これらの使い方の詳細に関しては Javadoc を参照してください。)

リスト 3. 最大の値を持つ要素を選択する問題をフォーク/ジョインのフレームワークで解決する

```
public class MaxWithFJ extends RecursiveAction {
    private final int threshold;
    private final SelectMaxProblem problem;
    public int result;

    public MaxWithFJ(SelectMaxProblem problem, int threshold) {
        this.problem = problem;
        this.threshold = threshold;
    }

    protected void compute() {
        if (problem.size < threshold)
            result = problem.solveSequentially();
        else {
            int midpoint = problem.size / 2;
            MaxWithFJ left = new MaxWithFJ(problem.subproblem(0, midpoint), threshold);
            MaxWithFJ right = new MaxWithFJ(problem.subproblem(midpoint +
              1, problem.size), threshold);
            coInvoke(left, right);
            result = Math.max(left.result, right.result);
        }
    }

    public static void main(String[] args) {
        SelectMaxProblem problem = ...
        int threshold = ...
        int nThreads = ...
        MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
        ForkJoinExecutor fjPool = new ForkJoinPool(nThreads);

        fjPool.invoke(mfj);
        int result = mfj.result;
    }
}
```

表 1 は、さまざまなシステムで、逐次型ソリューションの方が並列型ソリューションよりも好ましくなる敷居値をさまざまに変化させ、500,000 要素の配列の中から最大の値を持つ要素を選択した結果を示しています。ほとんどの実行において、フォーク/ジョインのプールのスレッド数は、利用可能なハードウェア・スレッドの数 (コアとコア当たりのスレッドの積) と等しくなっていま

した。表に示されている数字は、そのシステムで逐次型ソリューションを実行した場合と比較して、どれほど高速化されたかを示しています。

表 1. 50 万要素の配列に対して最大の値を持つ要素を選択する問題をさまざまなシステムで実行した結果

	しきい値 =500k	しきい値 =50k	しきい値 =5k	しきい値 =500	しきい値 =-50
Pentium-4 HT (2 スレッド)	1.0	1.07	1.02	.82	.2
Dual-Xeon HT (4 スレッド)	.88	3.02	3.2	2.22	.43
8-way Opteron (8 スレッド)	1.0	5.29	5.73	4.53	2.03
8コア Niagara (32 スレッド)	.98	10.46	17.21	15.34	6.49

広い範囲のパラメーターにわたって速度の向上が見られ、非常に心強い結果が得られています。つまり問題に対して、あるいはベースとなるハードウェアに対して、まったく不適切なパラメーターを選択しない限り、ほとんど調整せずに良い結果を得ることができます。チップによるマルチスレッドでは、どの程度が最適な高速化なのか、明確ではありません。もちろん、ハイパースレッディングなど CMT の方法では、実際に同じ数のコアがある場合ほど高いパフォーマンスは得られませんが、どの程度パフォーマンスが低いかは、実行されるコードがキャッシュ・ミスする割合を含めて、非常にさまざまな要因に依存します。

ここで選択した逐次処理のしきい値の範囲は、500K (配列のサイズ、つまり実質的に並列処理なし) から、下は 50 までです。この場合の逐次処理のしきい値 50 は、「桁外れに小さい」領域に十分入ります。また結果を見ると、逐次処理のしきい値を桁外れに低くすると、フォーク/ジョインのタスク管理のオーバーヘッドに占有されてしまいます。しかしこれらの結果から、「桁外れに高い」パラメーターや「桁外れに低い」パラメーターを避けさえすれば、調整をしなくても良い結果を得られることがよくわかります。一般的に、ワーカー・スレッドの数として `Runtime.availableProcessors()` を選択すると、最適に近い結果を得ることができます。これは、フォーク/ジョインのプールで実行されるタスクは CPU バウンドになるはずだからです。しかしこの場合も、プールを大きくしすぎたり小さくしすぎたりしない限り、結果はこのパラメーターにそれほど敏感ではないことが多いものです。

`MaxWithFJ` クラスでは、明示的な同期は必要ありません。このクラスによって操作されるデータは、問題が存在している間は一定であり、`ForkJoinExecutor` 内では十分な内部同期化が行われるので、サブタスクに対して問題データの可視性が保証され、またサブタスクを結合するタスクに対してサブタスクの結果の可視性が保証されます。

フォーク/ジョインのフレームワークを詳しく調べる

リスト 3 に示すようなフォーク/ジョインのフレームワークは、数多くの方法で実装することができます。スレッドをそのまま使う方法も選択肢の 1 つです (`Thread.start()` と `Thread.join()` は、必要なすべての機能を提供しています)。しかしこの方法では、VM がサポートできる以上のスレッドが必要になる可能性があります。サイズ Nの問題に対して、(逐次処理のしきい値が非常に小さいとすると) 問題を解決するために O(N) スレッドが必要になります (問題ツリーは深さ

$\log_2 N$ を持ち、また深さ k のバイナリー・ツリーは 2^k ノードを持ちます)。そしてこれらのスレッドの半分は、そのスレッドの存続期間のほとんどすべてをサブタスクが完了するのを待機することに費やします。スレッドは作成コストが高く、また大量のメモリーを使用するため、この方法を使用するのは困難です。(この方法を使用することはできますが、コードはより複雑になり、また問題のサイズやハードウェアに関するパラメーターを非常に注意深く調整する必要があります。)

従来のスレッド・プールを使ってフォーク/ジョインを実装する方法も、フォーク/ジョインのタスクはその存続期間の大部分を他のタスクを待機するのに費やしてしまうため、やはり簡単ではありません。作成されるタスクの数を制約するようにパラメーターを注意深く選ぶか、あるいはプール自体の制約をなくさない限り、この動作によってスレッド不足 (thread starvation) によるデッドロックが生じます。従来のスレッド・プールは、互いに独立しているタスクを対象に設計されており、またブロックされる可能性のある粒度の粗いタスクのことも想定して設計されていますが、フォーク/ジョインのソリューションで作成されるタスクは、そのどちらでもありません。また従来のスレッド・プールでは、細粒度のタスクによって、すべてのワーカーで共有されるタスク・キューに過度の競合が発生することにもなります。

ワーク・スティーリング

フォーク/ジョインのフレームワークは、ワーク・スティーリング (work stealing) として知られる手法を使うことで、ワーク・キューの競合を減少させることができます。各ワーカー・スレッドは、両端キューつまりデック (deque) を使って実装される独自のワーク・キューを持っています。(Java 6 は、`ArrayDeque` や `LinkedBlockingDeque` を含め、いくつかのデック実装をクラス・ライブラリーに追加しています。) あるタスクが新しいスレッドをフォークすると、そのタスクはそのスレッドをタスク自身のデックの先頭にプッシュします。あるタスクが、まだ完了していない別のタスクとのジョイン操作を実行すると、(`Thread.join()` のように) 対象とするタスクが完了するまでスリープするのではなく、デックの先頭から別のタスクをポップして、そのタスクを実行します。もしそのスレッドのタスク・キューが空の場合には、別のスレッドのデックの末尾から別のタスクをスティールしよう (奪おう) とします。

標準キューを使ってワーク・スティーリングを実装することはできますが、デックを使うと、競合を減らすことができ、またタスクのスティールを削減できるという、標準キューにはない 2 つの大きな利点を活用することができます。自分自身のデックの先頭にアクセスするのはワーカー・スレッドのみなので、デックの先頭に対する競合は起こりません。デックの末尾にアクセスするのは、スレッドの仕事がなくなった時だけなので、どのスレッドのデックの末尾でも、ほとんど競合は起こりません。(フォーク/ジョインのフレームワークに採用されているデックーの実装は、こうしたアクセス・パターンを利用して調整コストをさらに削減しています。) このように競合が減ることで、スレッド・プールをベースにした従来の方法と比較して、同期のコストが大幅に削減されます。さらに、こうした方法では暗黙的に LIFO (last-in-first-out) でタスクが順序付けされるため、最大のタスクがデックの末尾に置かれます。従って別のスレッドがタスクをティールしなければならない場合、そのスレッドは、小さなタスクに分解できる大きなタスクをスティールします。これによって、そのすぐ後に再度タスクをスティールする必要性が低くなります。つまりワーク・スティーリングによって、中央での調整なしに、また最小限の同期化コストで、適切なロード・バランシングを実現することができます。

まとめ

フォーク/ジョインの方法では、ターゲット・システムがどの程度の並列処理を提供するのかを事前に知らなくても並列化可能なアルゴリズムを表現するための、移植可能な方法を提供しています。あらゆる種類のソート・アルゴリズム、検索アルゴリズム、そして数値アルゴリズムは、並列分割が可能です。(将来は、`Arrays.sort()` のような標準ライブラリー機構がフォーク/ジョインのフレームワークを利用するようになり、それによってアプリケーションは並列分割の利点を存分に利用できるようになるかもしれません。) プロセッサーの数が増加し続けるにつれ、そうしたプロセッサーを効果的に利用するために、プログラムに特有の並列処理をもっと公開する必要があります。ソートなどの計算負荷の重い動作を並列分割することによって、プログラムは将来のハードウェアの利点を活用しやすくなります。

著者について

Brian Goetz

Brian Goetz はこれまで 20 年間、プロのソフトウェア開発者として活躍してきました。現在は Sun Microsystems のシニア・スタッフ・エンジニアであり、複数の JCP Expert Group の一員でもあります。2006年5月に Addison-Wesley から彼の著書『Java 並行処理プログラミング — その「基盤」と「最新 API」を究める —』が出版されています。人気の業界紙に掲載された、Brian のこれまでの記事、そして今後の記事を参照してください。