Fork/Join Framework の性能について

２０１２年１２月３日
湊　隆行

最近、CPU クロック数の上昇が望めなくなりマルチコア化された CPU が登場するようになると、マルチスレッド処理の高度化が求められるようになりました。しかし、スレッドはスタック領域などのメモリを必要とするので、大量のスレッドを生成するとメモリが枯渇する問題に直面しやすくなります。また、スレッドのプリエンプションなどにかかるオーバーヘッドなども含めると、大量のスレッドを生成すると逆にスケーラビリティが低下する恐れもあります。

その対策として、コア数と同数程度のスレッドをプールする「スレッドプール」と、タスクを溜めるための「ワークキュー」を用意しておき、ワークキューから取り出したタスクを、プールしているスレッドを使い回して処理する手法があります。この手法では、無駄に大量のスレッドを生成せずに、タスクを効率的に並行処理できるメリットがあります。

そこで、並行処理をサポートする Concurrency Utilities が JSR-166 で策定されました。Java SE 5.0 で追加された java.util.concurrent パッケージには、スレッドプールとワークキューを使ったクラスライブラリが含まれています。また Java SE 7 で同パッケージに追加された Fork/Join Framework も同じ手法を使っています。

一方、Java EE では Servlet/EJB のコンテナがスレッドを含む各種リソースを管理する関係で、アプリケーションから Thread クラスや java.util.concurrent パッケージを直接使うことを推奨していません。そのため現時点では、Fork/Join Framework の利用場面としては、スタンドアロン Java アプリケーションを起動してバッチ処理などを行うなどの限定的なものに限られるでしょう。しかし JSR-236 の Concurrency Utilities for Java EE により、将来的には Java EE アプリケーションから並行処理を実行できるようになると期待されます。

本書では、性能測定を通して Fork/Join Framework のチューニング方法を検証します。Fork/Join Framework を F/J と略記します。

1. 分割統合法

F/J が得意とされているアルゴリズムとして分割統合法があり、具体例として、フィボナッチ数の計算、クイックソートやマージソートがあります。本書ではフィボナッチ数の計算とクイックソートについて、F/J を使ったベンチマークプログラムを作成し性能比較を行います。

2. フィボナッチ数の計算

n のフィボナッチ数を計算する関数を F(n)とすると、フィボナッチ数は図 2-1 の漸化式で計算します。F(n)関数は、n=0 なら 0 を、n=1 なら 1 を返し、2 以上なら F(n-2)と F(n-1)の関数を再帰的に呼び出してその合計値を返します。フィボナッチ数をもっと高速に計算するアルゴリズムがありますが、今回は F/J の性能測定が目的なので分割統合法を使います。

図 2-1 フィボナッチ数の漸化式

```
F(0) = 0
F(1) = 1
```

F(n) = F(n-2) + F(n-1) (n≧2)

F(40)を計算するプログラムを3種類作ります。各プログラムをそれぞれ、図2-1～図2-3に示します。

- プログラム1： 単一スレッドでフィボナッチ数を計算する基本的なプログラム
- プログラム2： F/Jを使用してフィボナッチ数を計算するプログラム
- プログラム3： F/Jを使用してfork実行回数を抑制しながらフィボナッチ数を計算するプログラム

図2-1 フィボナッチ数計算プログラム1（単一スレッドで計算）

```
class FibonacciTask1 {
    int compute(int n){
        return (n <= 1) ? n : compute(n-2)+compute(n-1);
    }

    public static void main(String[] args){
        long start = System.nanoTime();
        int fibonacci = new FibonacciTask1().compute(40);
        long end = System.nanoTime();
        System.out.println("F(40)="+fibonacci+", elapsed="+(end-start));
    }
}
```

図2-2 フィボナッチ数計算プログラム2（Fork/Join Framework使用）

```
class FibonacciTask2 extends RecursiveTask<Integer> {
    int n;

    FibonacciTask2(int n){
        this.n = n;
    }

    @Override
    protected Integer compute(){
        return compute(n);
    }

    int compute(int n){
        if(n <= 1){
            return n;
        }
        ForkJoinTask<Integer> f1 = new FibonacciTask2(n-2).fork();
        return compute(n-1) + f1.join();
    }

    public static void main(String[] args){
        int multiplicity = Integer.parseInt(args[0]);
        long start = System.nanoTime();
        ForkJoinPool pool = new ForkJoinPool(multiplicity); //多重度を指定
        int fibonacci = pool.invoke(new FibonacciTask2(40));
        long end = System.nanoTime();
        System.out.println("F(40)="+fibonacci+", elapsed="+(end-start));
        pool.shutdown();
    }
}
```

図2-3 フィボナッチ数計算プログラム3（Fork/Join Framework使用＋fork実行回数抑制）

```
class FibonacciTask3 extends RecursiveTask<Integer> {
    private static final AtomicInteger taskCounter = new AtomicInteger(1);
    private static int multiplicity;
    private final int n;

    FibonacciTask3(int n) {
        this.n = n;
    }

    @Override
    protected Integer compute() {
        final int result = compute(n);
        taskCounter.decrementAndGet();
        return result;
    }

    int compute(int n){
        if(n <= 1){
            return n;
        }
        if(taskCounter.get() < multiplicity){
            taskCounter.incrementAndGet();
            final ForkJoinTask<Integer> f1 = new FibonacciTask3(n-2).fork();
            return compute(n-1) + f1.join();
        }
        return compute(n-2)+compute(n-1);
    }

    public static void main(String[] args){
        multiplicity = Integer.parseInt(args[0]);
        final long start = System.nanoTime();
        final ForkJoinPool pool = new ForkJoinPool(multiplicity); //多重度を指定
        final int fibonacci = pool.invoke(new FibonacciTask3(40));
        final long end = System.nanoTime();
        System.out.println("FibonacciTask4: F(40)="+fibonacci+", elapsed="+(end-start));
        pool.shutdown();
    }
}
```

プログラム1では、単一スレッド（多重度1）で再帰的にcompute(int)を呼び出します。プログラム2では、ForkJoinPoolコンストラクタでワーカースレッドの個数（多重度）を指定し、各ワーカースレッドでcompute(int)を再帰的に呼び出しながら、子タスクをfork()で起動しjoin()で計算結果を統合します。プログラム3は、プログラム2よりfork()の実行回数を減らしたものですが、詳しくは後で説明します。

コア数4（スレッド数8）のマシンでJava SE 7 Update 9のHotSpot Client VMでこれらのプログラムを起動し、多重度を変えながら実行時間を採取した結果を表2-1に示します。また、多重度1におけるガーベジコレクションの実行状況を表2-2に示します。

表2-1 フィボナッチ数計算プログラムの実行時間（単位：ミリ秒）

多重度	プログラム1	プログラム2	プログラム3
1	1,955	30,836	3,227
2	-	5,122	505
4	-	3,370	337
8	-	2,712	417

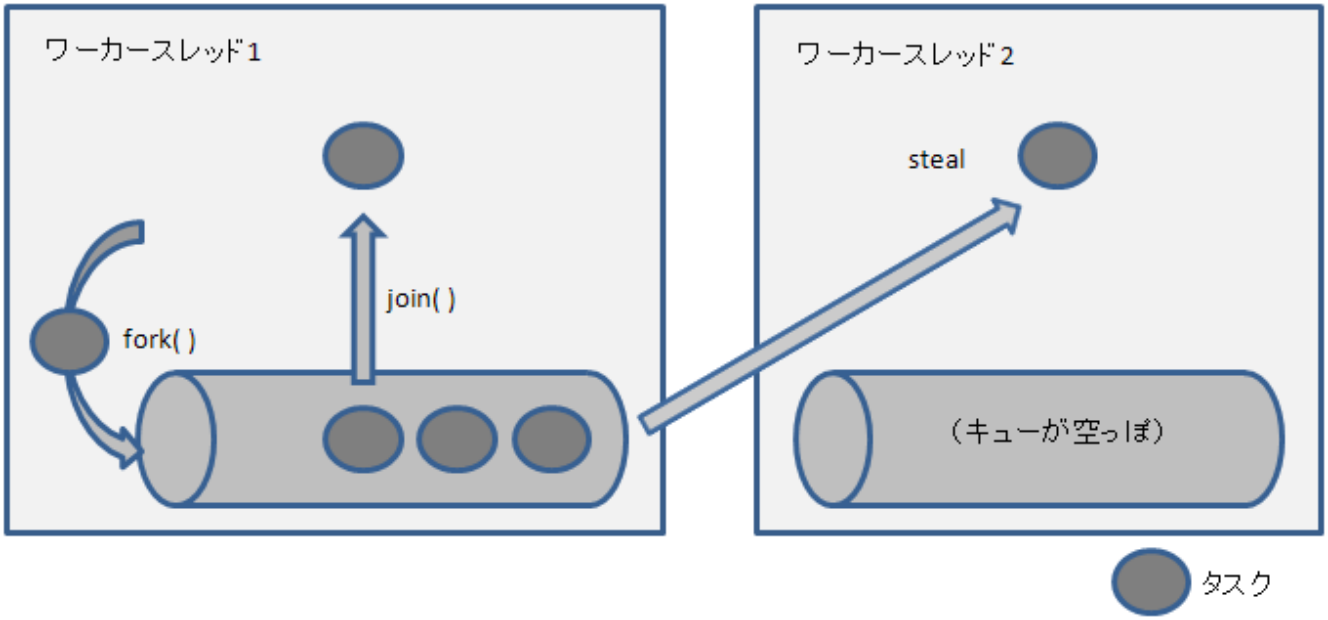
表2-2 多重度1におけるフィボナッチ数計算プログラムのガーベジコレクション実行状況（時間の単位：ミ

リ秒）

ガーベジコレクション名	プログラム1		プログラム2		プログラム3	
	回数	時間	回数	時間	回数	時間
マイナーGC	0	0	885	252	10	9
Full GC	0	0	0	0	0	0

ForkJoinPoolのAPIドキュメントに「work-stealing」の簡単な説明があります。F/Jでは指定した多重度分だけワーカースレッドを生成し、各ワーカースレッドはforkされたタスクを蓄積するワークキューを所有します。基本的にワーカースレッドは所有しているワークキューからタスクを取り出して実行しますが、ワークキューが空っぽになると、他ワーカースレッドのワークキューからタスクを取り出して実行します。この処理方式をwork-stealingといい、java.util.concurrencyパッケージの他のExecutorServiceにないF/Jの特長です。work-stealingによりスレッドが「遊んでいる」時間を極力減らして、並行処理の密度を上げることを狙いとしています。work-stealingのイメージを、図2-4に示します。

図2-4 work-stealingのイメージ図



しかし、表2-1のプログラム1とプログラム2を見比べると、F/Jを使うプログラム2の多重度を8まで増やしても、フィボナッチ数の計算の実行時間は2,712ミリ秒でした。F/Jを使わないプログラム1の実行時間（1,955ミリ秒）より長く、F/Jを使うメリットが得られない結果となりました。

多重度1におけるプログラム1とプログラム2の実行時間の差は28,881ミリ秒（=30,836−1,955）です。表2-2のマイナーGC（New世代領域に対するガーベジコレクション）の実行時間を差し引いても、28,629ミリ秒（=28,881-252）も開きがあります。この約29秒をF/Jによるオーバーヘッドとみなすと、どの処理が主なボトルネックになるでしょうか。

ForkJoinPoolとForkJoinWorkerThreadなどの各メソッドの処理時間を採取したところ、ワークキューに対する処理に多くの時間を消費していることが判明しました。またワークキューにタスクが多く蓄積されているときに、ワークキューに対する処理時間が長くなる傾向がありました。

そこで、ワークキューに蓄積するタスクを減らしてボトルネックの軽減を狙ったのが、プログラム3です。プログラム2ではcompute(int)の引数が2以上なら必ずfork()を実行しているのに対し、プログラム3ではア

トミック変数taskCounterを使ってforkの実行回数を抑えています。「if(*taskCounter*.get() < *multiplicity*)」と「*taskCounter*.incrementAndGet()」を同期しないと正しいカウンタになりませんが、synchronizedなどのロック機構は性能が著しく低下するため使わないことにしました。

プログラム3を実行したところ、表2-1にあるように、プログラム2より実行時間が大幅に短縮し、多重度2以上ではプログラム1より実行時間が短くなり、F/Jを使うメリットが得られる結果となりました。

プログラム2とプログラム3との間で、どのような変化が起きたのでしょうか。表2-2のガーベジコレクションの実行状況を見比べると、タスクの生成量（new FibonacciTask3(n-2)の実行回数）を抑制しているため、マイナーGCの回数と時間がそれぞれ「885回→10回」と「252ミリ秒→9ミリ秒」と減りました。

また多重度8におけるプログラム2とプログラム3の子タスクの処理状況を、表2-3に示します。

表2-3 多重度8における子タスクの処理状況

	プログラム 2	プログラム 3
fork 実行回数	16,558 万	181 万
join 実行回数	16,558 万	181 万
steal 実行回数	83	61 万
総計	33,116 万	423 万

fork実行回数が16,558万回→181万回に減り、joinもforkと同じ回数だけ実行しているので、同じように減ります。その代わり、ワークキューが空っぽになりやすいため、ワーカースレッド間でタスクを取り合う回数（steal発生回数）が83回→61万回と増えました。fork、joinとstealは、すべてワークキューに対する処理を行いますが、その実行回数を合計すると33,116万回→241万回と大幅に減りました。ワークキューに対する処理回数を減らしたことが、実行時間の短縮につながったわけです。

今回はカウンタを使いましたが、何らかの「しきい値」を使ってfork実行回数を調整するチューニングが有効であるとわかります。また、多くのコアを搭載しているマシンであれば、各ワーカースレッドが所有するワークキューにタスクが溜まりすぎないように、F/Jの多重度を上げるのも効果的と言えます。

3. クイックソート

クリックソートの処理内容を説明します。ソート対象のリスト中の 2 つの先頭要素のうち大きい要素を軸要素として決めます。そして、リスト中の先頭位置と最終位置それぞれから逆方向に向かって探しながら、「軸要素以上の要素」と「軸要素より小さな要素」を見つけては入れ替えます。「先頭位置から探した位置」と「最終位置から探した位置」が交差したときに、この位置をもとにリストを 2 つに分割します。すると、左側のリストには軸要素よりも小さな要素しかなく、右側のリストには軸要素以上の要素しかない状態になります。こうして分割したリストに対し、リストの長さが 1 になるまで再帰的に同じ処理を繰り返します。

クイックソートの処理イメージを図 3-1 に示します。またF/J を使ってクイックソートを行うタスクを、図 3-2 に示します。このタスクでは、pivot()で軸要素を決め、process()でリスト内の要素を入れ替えます。

図 3-1 クイックソートの処理イメージ

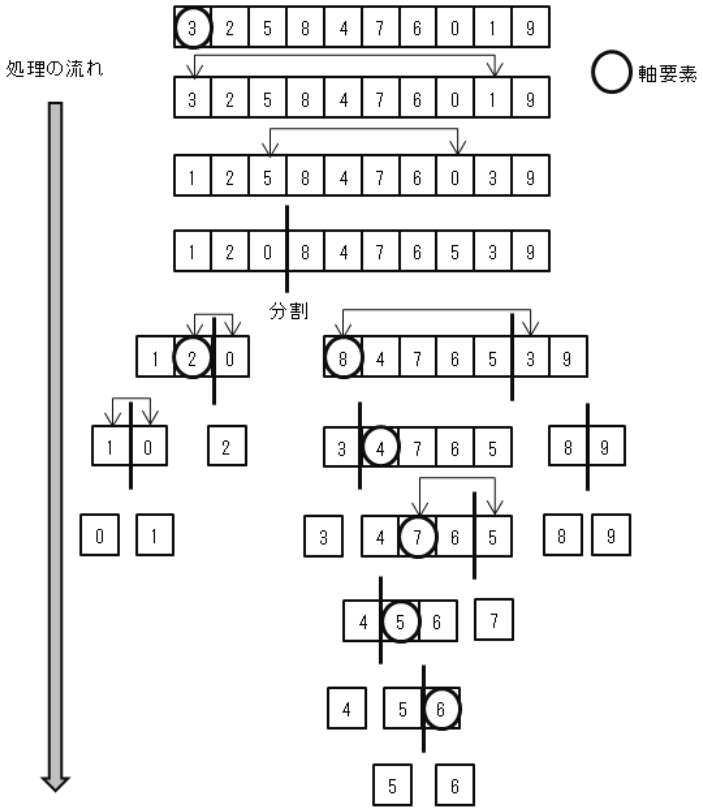


図 3-2 クイックソートを行う Fork/Join Framework 用タスク

```
class QuickSortTask extends RecursiveAction {
    private final List<T> list;

    QuickSortTask(List<T> list){
        this.list = list;
    }

    @Override
    protected void compute(){
        final int pos = process(list);
        if(pos > 0){
```

```
            new QuickSortTask(list.subList(0, pos)).fork();
            new QuickSortTask(list.subList(pos, list.size())).compute();
        }
    }

    private int process(List<T> list){
        int size = list.size();
        //リストのサイズが1なら終了
        if(size <= 1){
            return -1;
        }
        //軸要素を決める
        int pivot = pivot(list);
        if(pivot < 0){
            return -1;
        }
        //軸要素
        T e1 = list.get(pivot);
        //leftとrightの位置が交差するまで処理
        int left = pivot;
        int right = size-1;
        do{
            //軸要素以上の位置を探す
            while(left<size && e1.compareTo(list.get(left))>0){
                left++;
            }
            //軸要素より小さい位置を探す
            while(right>=0 && e1.compareTo(list.get(right))<=0){
                right--;
            }
            //要素を入れ替える
            if(left < right){
                Collections.swap(list, left++, right--);
            }
        }while(left <= right);
        return left;
    }

    /**
     *軸要素の位置を決めます。
     *要素0より大きい要素の位置を返します。
     *ただし要素0が一番大きい場合は0、全要素が同値の場合は-1を返します。
     */
    private int pivot(List<T> list){
        int size = list.size();
        T e1 = list.get(0);

        int i = 1;
        while(i < size) {
            T e2 = list.get(i);
            int c = e1.compareTo(e2);
            if(c < 0){
                return i;
            }else if(c > 0){
                return 0;
```

```
            }
            i++;
        }
        return (i==size) ? -1 : i;
    }
}
```

フィボナッチ数の計算と同様、次の 3 種類のベンチマークプログラムを作成します。

- プログラム 1： 単一スレッドでクイックソートを行う基本的なプログラム
- プログラム 2： F/J を使用してクイックソートを行うプログラム
- プログラム 3： F/J を使用して fork 実行回数を抑制しながらクイックソートを行うプログラム

コア数4（スレッド数8）のマシンでJava SE 7 Update 9のHotSpot Client VMで、これらのプログラムを起動し、200万個の要素を持つリストに対するクイックソートの性能を採取しました。多重度を変えながら実行時間を採取した結果と、多重度4における子タスクの処理状況を採取した結果を、それぞれ表3-1と表3-2に示します。またプログラム2とプログラム3にてjava.lang.managementパッケージのThreadMXBeanを使って、多重度2～8におけるワーカースレッドの状況を採取した結果を、それぞれ表3-3と表3-4に示します。なお、プログラム2とプログラム3ではマイナーGCが発生しましたが、フィボナッチ数の計算と同様、その実行時間は無視できる程度に短かいものでした。

表 3-1 クイックソートプログラムの実行時間（単位：ミリ秒）

多重度	プログラム 1	プログラム 2	プログラム 3
1	1,943	4,613	4,810
2	-	1,181	1,113
4	-	916	1,090
8	-	1,333	1,184

表3-2 多重度4における子タスクの処理状況

	プログラム 2	プログラム 3
fork 実行回数	1,999,999	141
steal 実行回数	252	80
総計	2,000,251	221

表3-3 プログラム2におけるワーカースレッドの処理状況の平均値（時間の単位：ミリ秒）

多重度	CPU 時間	BLOCKED		WAITING	
		回数	時間	回数	時間
2	2,465	0	0	2	89
4	1,689	0	0	1	82
8	1,453	0	0	17	273

表3-4 プログラム3におけるワーカースレッドの処理状況の平均値（時間の単位：ミリ秒）

多重度	CPU 時間	BLOCKED		WAITING	
		回数	時間	回数	時間
2	2,340	0	0	1	934
4	1,244	0	0	20	743
8	735	0	0	77	1,409

4 つの表を見比べると、フィボナッチ数の計算と異なる点が 2 つあります。

まず、表 3-1 と表 3-2 を見比べてください。プログラム 2 の方がプログラム 3 より fork と steal の実行回数が多いにも関わらず、実行時間が短い結果となりました。フィボナッチ数の計算と異なり、fork 実行回数のチューニングが効かないことを示しています。

次に、表 3-3 と表 3-4 を見てください。プログラム 2 とプログラム 3 ともに WAITING 状態が発生しています。表 3-1 と見比べると、WAITING 状態になった時間と実行時間に関連性が見られ、WAITING 状態が短いほど実行時間も短くなり、多重度 4 の性能が最も実行時間が短い結果となりました。

この 2 点からわかることは、ワークキューにタスクが溜まりにくく、ワーカースレッドが待機状態になりやすいということです。フィボナッチ数の計算と違ってクイックソートでは、fork 実行回数を減らしても多重度を上げすぎても逆効果になるわけです。

4. まとめ

フィボナッチ数の計算とクイックソートは同じ分割統合法ですが、ワークキューへのタスクの溜まり方によってチューニング方法が異なることを紹介しました。まとめとして、表 4-1 に示します。

表 4-1 処理内容と Fork/Join Framework のチューニング方法の違い

処理内容	チューニング方法
ワークキューにタスクが溜まりやすい処理 （例：フィボナッチ数の計算）	・多重度を増やす ・コア数が少ない場合は、fork 実行回数を抑制
ワークキューにタスクが溜まりにくい処理 （例：クイックソート）	多重度を減らす

Fork/Join Framework は work-stealing 方式でタスクを回す分だけオーバーヘッドが高くなります。しかし、各ワーカースレッドで細粒度タスク（処理時間が短いタスク）を効率的に消化できるようにチューニングを行えば、他の ExecutorService よりも速度性能が向上します。

商標について

・Java、Java HotSpot は、Oracle Corporation およびその子会社、関連会社の米国およびその他の国における登録商標です。

・その他の会社名および製品名は、それぞれの会社の登録商標もしくは商標です。

－以上－