C-034

マルチコア上での Fork/Join Framework を用いた再帰プログラムの粗粒度並列処理

Coarse Grain Parallel Processing Using Fork/Join Framework for Recursive Programs on Multicore Processors

橋本 里菜 †　　吉田 明正 †
Rina Hashimoto　　Akimasa Yoshida

1　はじめに

Java 言語の並列処理フレームワークとして利用されている Fork/Join Framework[1] は，ワークスティーリングを伴う低オーバヘッドなスケジューリング環境を提供しており，分割統治プログラム等の並列化に利用されてきた．この Fork/Join Framework は，粗粒度タスク間の並列性を引き出す粗粒度並列処理のスケジューラとして利用することも可能であり，並列化コンパイラを用いたコード生成手法が提案されている [2]．一方，Java 言語の再帰プログラムの並列処理に関する研究 [3][4] が提案されているが，Fork/Join Framework を対象としていない．

本稿では，再帰メソッド外部の粗粒度タスクと，再帰メソッド内部の粗粒度タスクを一括して管理し，これらの粗粒度タスクを Fork/Join Framework により均質なコアに割り当てる並列コードの生成手法を提案する．Intel Xeon E5-2680 上で性能評価を行ったところ，高い実効性能が得られ，提案手法の有効性が確認された．

2　タスク駆動型実行による粗粒度並列処理

マルチコア上での Java Fork/Join Framework 環境において，階層統合型粗粒度並列処理 [5] を実現する方法として，タスク駆動型実行 [2] が提案されている．Fork/Join Framework を用いたタスク駆動型粗粒度並列処理コードは，並列化コンパイラにより生成することが可能であり，そのコードを Java 仮想マシンで実行することにより，Fork/Join Framework のスケジューラを用いて粗粒度並列処理を効率よく行うことができる．

具体的には，並列化コンパイラ [2] が生成した粗粒度並列処理コード（以後，並列 Java コードと呼ぶ）によってマクロタスク（MT）の終了状態と分岐状態を管理し，当該マクロタスクの状態変化により実行可能になるマクロタスクを Fork する．但し，Fork されたマクロタスクは，各スレッドが所有するワーカーキューに投入された後，Fork/Join Framework のスケジューラが，ワーカーキューのマクロタスクを取り出してワーカースレッドで実行する．このとき必要に応じてワークスティーリングを行う．なお，後続マクロタスク候補に対して最早実行可能条件の判定を行い，実行可能な場合にその後続マクロタスクを Fork する．

3　再帰プログラムのための並列 Java コード生成

再帰メソッドを含むプログラムにタスク駆動型粗粒度並列処理 [2] を適用する場合，従来は再帰メソッド外部の粗粒度タスクに対して並列処理を行っており，再帰メソッド自身は単一コアで実行されていた．それゆえ，再帰メソッド内部の並列性の利用が十分ではなかった．

†明治大学総合数理学部ネットワークデザイン学科
School of Interdisciplinary Mathematical Sciences, Meiji Univ.

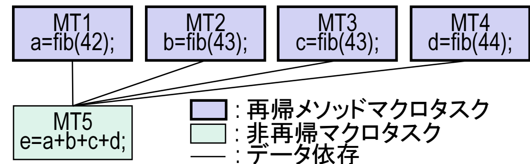

図 1　再帰プログラムのマクロタスクグラフ．

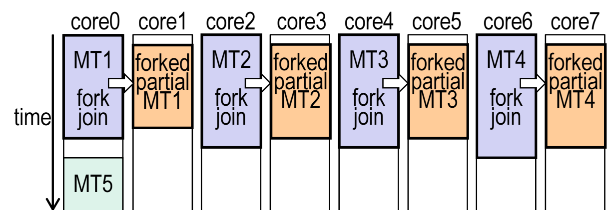

図 2　MT 内 Fork を伴うタスク駆動型粗粒度並列処理．

本稿では，このような問題点を解決するために，再帰メソッド外部の粗粒度タスクと，再帰メソッド内部の粗粒度タスクを一括管理して，再帰メソッド内部の並列性も十分に利用することを可能にしている．これらの粗粒度タスクを均質なコアに割り当てるために，Fork/Join Framework のスケジューラを用いており，コア数の増大に柔軟に対応することができる．

例えば，図 1 のマクロタスクグラフで表現される再帰呼び出しを含むプログラムの場合，図 2 の 8 コア実行イメージに示されるように，MT1〜MT4 はそれぞれが再帰呼び出しに対応しており，再帰呼び出しの一部分を Fork して，別コアで実行することにより，並列処理時間を短くすることが可能である．

3.1　再帰メソッド外部に対応する並列 Java コード

並列 Java コードの基本構造 [2] は，共通データのための Data クラス，並列 Java コードの main() メソッド（図 3 の 42〜47 行目）を含む Mainp クラスからなる．

Data クラスは図示されていないが，マクロタスク最早実行可能条件のチェックを行う eeccheck() メソッドが定義されている．一方，Mainp クラス（図 3）では，内部クラスの Layer0 クラス（32〜41 行目）が定義されている．main() から Layer クラスのコンストラクタを invoke()（45 行目）で呼び出すことにより，Layer クラス内部の compute() メソッド（図 3 の 36 行目）を処理する．これにより，RecursiveAction クラスを継承した ForkTemplate_main クラスが呼び出され，Fork/Join による並列処理が開始される．Fork された Main クラスのマクロタスクは，まず，compute() メソッド（図 3 の 18 行目）を処理し，eeccheck() の最早実行可能条件をもとに，どのマクロタスクを実行すればよいのかを判定する．その後，各マクロタスクに対応するメソッド（例，25 行目）を呼び出して処理が行われる．

Copyright © 2016 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

```
01: class Mainp { //並列メイン
02:     static class Recur extends RecursiveTask { //ForkJoin開始
03:         Recur() { … }//コンストラクタ
04:         @Override protected Integer compute() { mycompute(n); }
05:         int mycompute(int n) {
06:           …
07:           if (処理時間が基準以上) {
08:              final ForkJoinTask<Integer>f1 = new Recur(n-2).fork();
09:              return mycompute(n-1) + f1.join();
10:           }
11:           return mycompute(n-1) + mycompute(n-2);
12:         }
13:     }
14:     public static class ForkTemplate_main extends RecursiveAction{
15:         ForkTemplate_main(各引数){  //コンストラクタ
16:            該当MTで実行すべきMT番号を設定
17:         }
18:         protected void compute(){
19:            設定されたMT番号に該当するマクロタスクを実行
20:         }
21:         public void mtStart(){
22:            マクロタスク実行管理テーブルの更新
23:            データ依存後続マクロタスクのFork
24:         }
25:         public void mt1() { //実行管理条件付きマクロタスク処理
26:             Recur t=new Recur(42); t.fork(); Data.a=t.join();
27:             マクロタスク実行管理条件テーブルの更新
28:             データ依存後続マクロタスクのFork
29:         }
30:         ...
31:     }
32:     static class Layer0 extends RecursiveAction {
33:          Layer0(){ //コンストラクタ
34:             Dataクラスのフィールド変数の初期化;
35:         }
36:         protected void compute(){
37:             ForkTemplate_mainクラスのmtStartのforkを行う;
38:             helpQuiesce()でタスク処理へ移行;
39:             joinを行う;
40:         }
41:     }
42:     public static void main(…){
43:         ForkJoinPool pool = new ForkJoinPool(poolnum);
44:         Layer0 layer0 = new Layer0();
45:         pool.invoke(layer0); //内部のcompute()メソッドを処理
46:         Fork/Joinによる並列処理が開始される
47:     }
48: }
```

図 3　タスク駆動型並列 Java コード（Mainp クラス）.

3.2　再帰メソッド内部に対応する並列 Java コード

本稿では，粗粒度並列処理の適用されるマクロタスク内部において，その再帰メソッド呼び出しに Fork/Join を適用し，並列性を高める．再帰メソッドの呼び出しは，マクロタスク処理内の Recur（図 3 の 26 行目）コンストラクタを呼び出だすことにより実行される．Recur クラス（RecursiveTask クラスを継承している）では，mycompute() メソッドが実行され，処理時間が基準以上あれば，フィボナッチ数 fib(n) の計算のうち，処理時間の小さい fib(n-2) の部分を Fork して別コアで実行し，fib(n-1) は自コアで計算する．なお，これらの計算結果の和を Recur() の返却値とする．

4　再帰プログラムの粗粒度並列処理の性能評価

本章では，並列サーバ DELL PowerEdge R730 上で，再帰プログラムを用いてタスク駆動型粗粒度並列処理の性能評価を行う．

4.1　性能評価に用いる並列処理環境

DELL PowerEdge R730 は，CPU:Intel Xeon E5-2680 v3 2.5GHz 12core*2，メモリ：64GB，OS：CentOS6.7，Java 処理系：JDK1.8 からなる構成となっている．

4.2　再帰プログラムを用いた性能評価

本性能評価では，再帰プログラムとして，図 1 のマクロタスクグラフで表されるフィボナッチ数を計算するプログラムを用いる．本プログラムは，46 番目のフィボナッチ数（fib(46) と表記）の計算を MT1〜MT4 のマクロタスクに分けて計算しており，これらの合計を求める部分を MT5 と定義した．

表 1　Xeon E5-2680 でのタスク駆動型粗粒度並列処理.

実行方式	並列実行時間 [ms]				
	1core	4core	8core	12core	24core
粗粒度	9773	3736	3740	3744	3737
(1 コア比)	(1.00)	(2.62)	(2.61)	(2.61)	(2.62)
粗粒度・再帰	10523	3301	1665	1128	632
(1 コア比)	(1.00)	(3.19)	(6.32)	(9.33)	(16.66)

まず，このプログラムを並列化コンパイラ [2] に入力し，Fork/Join スケジューラを用いた並列 Java コードを生成する．この並列 Java コードは，マクロタスクのダイナミックスケジューリングに Fork/Join スケジューラが使用されており，マクロタスクは各スレッドに割り当てられて実行される．実行結果は表 1 のように，4 つのマクロタスク間の並列性が引き出されており，4 コアで 1 コアの 2.62 倍の速度向上が得られている．但し，利用できる並列性は粗粒度並列性に限られているため，24 コアにおける速度向上は 2.62 倍に留まっている．

次に，前述の並列 Java コードにおいて，MT1〜MT4 の計算部分において，提案する RecursiveTask を用いた Fork/Join コードを手動で挿入した．実行結果は，図 2 に示されるように，MT1〜MT4 のそれぞれにおいて，計算処理の一部分（再帰処理の一部）を Fork により Fork/Join フレームワークのワーカーキューに投入する．即ち，ワーカーキューは，通常のマクロタスクと RecursiveTask により新たに投入された再帰処理の一部を統一して管理し，アイドルなコアで実行される．それゆえ，提案手法による実行結果は，表 1 の粗粒度・再帰に記載されているように，4 コアで 1 コアの 3.19 倍，24 コアで 16.66 倍の速度向上が得られており，高い実効性能を達成できることが確かめられた．

5　おわりに

本稿では，再帰メソッド外部の粗粒度タスクと再帰メソッド内部の粗粒度タスクを一括管理し，これらの粗粒度タスクを Fork/Join Framework により均質なコアに割り当てる並列 Java コードの生成手法を提案した．

本手法では，再帰呼び出しにより動的に生成される内部の粗粒度タスクにも Fork/Join を適用しており，高い並列性能を引き出すことが可能になっている．マルチコア Xeon E5-2680 上で行った性能評価の結果から，タスク駆動型粗粒度並列処理は再帰プログラムに対しても高い性能を達成できることが確認された．

本研究の一部は，JSPS 科研費基盤研究 (C) 課題番号 16K00174 の助成により行われた．

参考文献

[1] 湊隆行.　Fork/Join Framework の性能について. https://software.fujitsu.com/jp/technical/interstage/apserver/guide/pdf/forkjoin-20130115.pdf 富士通 Interstage Application Server 手引き/ガイド集, 2012.

[2] 吉田明正, 神山彰. Android プラットフォームにおける Java Fork/Join Framework を用いた粗粒度並列処理. 情報処理学会研究報告, 2016-ARC-218(11), 2016.

[3] B. J. Bradel, T. S. Abdelrahman. The Use of Hardware Transactional Memory for the Trace-Based Parallelization of Recursive Java Programs. *PPPJ*, 2009.

[4] 遠藤佑太, 山内長承, 吉田明正. マルチコア上での再帰プログラムの階層間並列性を利用した粗粒度タスク並列処理．情報処理学会研究報告, 2014-ARC-208(24), 2014.

[5] A. Yoshida, Y. Ochi, N. Yamanouchi, Parallel Java Code Generation for Layer-unified Coarse Grain Task Parallel Processing. 情報処理学会論文誌, コンピューティングシステム (ACS), Vol.7, No.4, 2014.

Copyright © 2016 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.