C-013

マルチコア上での Java インスタンスメソッドの タスク駆動実行による粗粒度並列処理

Coarse Grain Prallelization Using Task-driven Execution for Java Instance Method on Multicore

村上 茜 †　　吉田 明正 †
Akane Murakami　　Akimasa Yoshida

1　はじめに

近年，高性能計算を実現するために，マルチコアプロセッサやメニーコアプロセッサが幅広い分野で用いられている．このマルチコアプロセッサの性能を最大限に引き出すためには，対象プログラムのループ並列性に加えて，粗粒度並列性 [1][2][3] を利用することが必要である．マルチコアにおける Java プログラムの並列処理環境は，従来の Thread クラス (Runnable インタフェース) に加えて，Fork/Join Framework[4] が利用可能であり，粗粒度並列処理の実装方式も，Runnable インタフェースを用いる並列コード生成手法 [5][6] と Fork/Join Framework を用いるタスク駆動型並列コード生成手法 [3] が提案されている．

本稿では，タスク駆動型並列コードを用いた粗粒度並列処理において，インスタンスメソッド外部の並列性 (コール文レベル) に加えて，インスタンスメソッド内部の並列性を効果的に利用する並列実行手法を提案する．Intel Xeon E5-2680 上で行った性能評価の結果から，その有効性が確かめられた．

2　タスク駆動型実行による粗粒度並列処理

本章ではタスク駆動型実行の概念とその粗粒度並列処理コードについて述べる．

2.1　Fork/Join Framework を用いたタスク駆動実行の概念

本手法のベースとするタスク駆動型実行は，入力プログラムの構造に対応してマクロタスク (MT) を階層的に定義し，各マクロタスク間のデータ依存と制御依存を解析して，最早実行可能条件の形で並列性を抽出する [6]．次に，並列性を表した図 1 のようなマクロタスクグラフを用いてマクロタスクの終了・分岐状態を管理し，実行可能マクロタスクを Fork によりワーカキューに投入する．その後，Fork/Join Framework のスケジューラがワーカキューのマクロタスクを取り出して，ワーカ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が行われる．

2.2　タスク駆動型実行のための粗粒度並列処理コード

Fork/Join Framework を用いたタスク駆動型粗粒度並列処理コードは，指示文付 Java プログラムを入力として並列化コンパイラ [3] により生成される．但し，後述するメソッドローカルスケジューラのコードは手動で埋め込んでいる．

3　インスタンスメソッド外部/内部の粗粒度並列処理

本章では，インスタンスメソッド外部 (コール文レベル) のマクロタスクをスケジューリングするためのグローバルスケジューラと，インスタンスメソッド内部のマクロタスクをスケジューリングするためのメソッドローカルスケジューラについて述べる．

†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.

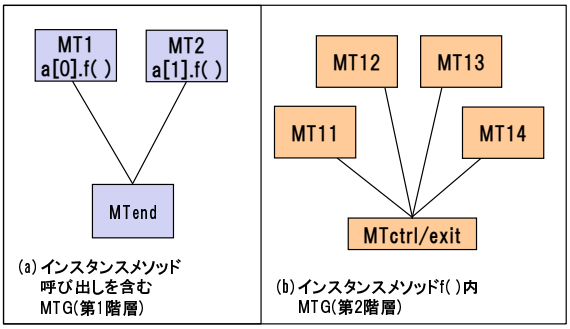

図 1　インスタンスメソッドを含むプログラムのマクロタスクグラフ

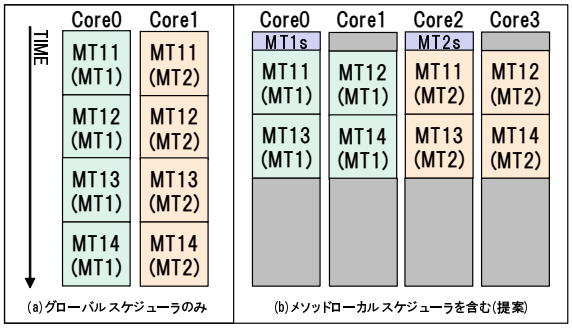

図 2　従来手法と提案手法の実行イメージ

3.1　グローバルスケジューラ

main メソッド内で実行されるマクロタスクの Fork 処理はすべてグローバルスケジューラで管理されている．

グローバルスケジューラはあるマクロタスクの実行が終了すると，タスク駆動型実行方式により，新たに実行可能になる後続マクロタスクを Fork する．但し，グローバルスケジューラはインスタンスメソッドをコールすることはできるが，インスタンスメソッド内部のマクロタスクのスケジューリングは後述のメソッドローカルスケジューラに依頼する．例えば図 1 の場合，グローバルスケジューラは (a) 第 1 階層の 2 つのマクロタスク (メソッドコール文) をスケジューリングの対象とする．

なお，図 2(b) の MT1s と MT2s のように，開始マクロタスク [3] はインスタンスメソッド内部から独立しているため，メソッドローカルスケジューラを起動した後にワーカを占有することはない．

3.2　メソッドローカルスケジューラ

グローバルスケジューラにより起動したメソッドローカルスケジューラは，当該インスタンス内部のスケジューリングを担当する．起動の際に，グローバルスケジューラで管理されていた main メソッド側のマクロタスク番号を受け取っておき，メソッドローカルスケジューラの終了時点で受け取った番号に対する終了状態および分岐状態の操作を行う (図 3 の 48 行目)．

また，本手法ではタスク駆動型実行用いているため，

```
01: class Sample {
02:     public static void main(String args[]){
03:         ForkJoinPool pool = new ForkJoinPool(poolnum);
04:         MTG0 mtg0 = new MTG0();
05:         pool.invoke(mtg0);
06:     }
07:     public static class Main extends RecursiveAction {
08:         Main(MT番号) { } //コンストラクタ
09:         protected void compute() { } //実行
10:         public void mtStart() { } //開始タスク
11:         public void mt1()  {
12:             //外部メソッドのインスタンス化
13:             //Func( 戻る番号, グローバルスケジューラ,
14:                     後続MTのインスタンス )
15:             Func f1 = new Func(renum, test.Data, new Main(...));
16:             //初期化, 最初のタスクのfork
17:             Func.initStbr fs1 = f1.new initStrb();
18:         }
19:     }
20:         public void mt2()  { }
21:         public void mtEnd()  { }
22: }
23:
24: class FuncData extend RecursiveAction {
25:     void initReverseEEC() { } //後続候補のMT
26:     boolean eeccheck(int unimt) { } //実行可能かどうか調べる
27: }
28:
29: class Func extends RecursiveAction {
30:     Func( 戻る番号, グローバルスケジューラ,
31:           後続MTのｲﾝｽﾀﾝｽ ) { } //コンストラクタ
32:     //スケジューラの初期化と開始
33:     public class initStbr() { }
34:     public class Methid extends RecursiveAction {
35:         Method ( MT番号 ); //コンストラクタ
36:         protected void compute() { } //実行
37:         int calc() { }  //計算
38:
39:         public void mt2_1() {
40:             funcdata.st[1]=true;
41:             calc();
42:             synchronized ( FuncData.class ) {
43:                 //後続MTをfork
44:             }
45:         }
46:         public void mt2_2() { }
47:         public void mt2Exit() {
48:             data.st[renum] = true; //戻る場所の準備
49:             synchronized ( Data.class ) {
50:                 //親側のMTをfork
51:                 test.Data.mtg.add(後続MTのインスタンス).fork;
52:             }
53:         }
54: }
```

図 3　グローバルスケジューラとメソッドローカルスケジューラを含む並列 Java コード

インスタンスメソッド側において，呼び出し元の後続マクロタスクを Fork する必要がある．このマクロタスクについてもスケジューラの起動時に引数として受け取ることで，スケジューラの終了時に後続候補のマクロタスクを Fork することができる (図 3 の 50〜51 行目)．

実行イメージは図 2 のようになり，従来法では (a) のようにインスタンスメソッドを 1 つのマクロタスクとして扱っていたが，(b) では内部の並列性を利用できており，より有効な手段であるといえる．

4　インスタンスメソッドを含むプログラムによる粗粒度並列処理の性能評価

本章では，マルチコア Xeon E5-2680 上で行った性能評価について述べる．

4.1　性能評価に用いるマルチコアサーバ

性能評価に用いる DELL PowerEdge R370 は，CPU:Xeon E5-2680 v3 (2.5GHz) 12core × 2，メモリ:64GB，OS:CentOS6.9，Java 処理系:JDK1.8 から構成される．

4.2　マンデルブロー図形生成プログラムを用いた性能評価

マンデルブロー図形はフラクタル図形であり，複素平面の各点 C において，$Z = Z^n + C$ を計算し，$|Z|^2 > 4$ になるまでの回数を求めることにより，その発散までの回数を色別に図示したものである．本評価で用いたプログラムでは，次数 $n = 60$，画像のピクセル数を 2560×2560 とした．main メソッド側のマクロタスクグラフ (第 1 階層) は 8 つのインスタンスメソッドコール文から構成し，さらに各インスタンスメソッド内部に 8 つのマクロタスクを持たせた．実行した総 MT 数は 64 となる．結果は表 1 に示す．

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	2core	4core	8core	16core
提案手法	8078	4120	2231	1323	907
(1 コア比)	(1.00x)	(1.96x)	(3.62x)	(6.11x)	(8.91x)
従来手法	7939	4001	2496	2354	2373
(1 コア比)	(1.00x)	(1.98x)	(3.37x)	(3.37x)	(3.34x)

インスタンスメソッドのコール文を一つのマクロタスクとして扱う従来手法では，16 コアで 3.34 倍のように 4 コア以上使用した場合において速度向上が得られていなかった．提案手法では 16 コアを使用した場合にもインスタンスメソッド外部および内部の並列性を利用することができており．8.91 倍の速度向上が得られており，その有用性が確認できた．

5　おわりに

本稿では，マルチコア上での Java プログラムのタスク駆動型実行による粗粒度並列処理において，インスタンスメソッド外部 (コール文レベル) の並列性に加えて，インスタンスメソッド内部の並列性を効果的に利用する並列実行手法を提案した．本手法の並列 Java コードは，グローバルスケジューラとメソッドローカルスケジューラから構成されており，Fork/Join Framework 環境下で効率よく並列処理を行えることが確認された．

また，マルチコア Xeon E5-2680 上で行った性能評価により，16 コアで 8.91 倍の速度向上が得られており，インスタンスメソッド外部および内部の並列性を利用する提案手法の有効性が確認された．

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] 笠原博徳，小幡元樹，石坂一久: 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理．情報処理学会論文誌，Vol. 42, No. 4, pp. 910-920 (2001).

[2] Mase,M., Onozaki,Y., Kimura,K., Kasahara,H.: Parallelizable C and Its Performance on Low Power HighPerformance Multicore Processors. Proc. of 15th Work-shop on Compilers for Parallel Computing (2010).

[3] Yoshida,A., Kamiyama,A., Oka,H.: A Task-driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform. Journal of Information Prosessing，Vol.25，2017.

[4] Lea,D.: A Java Fork/Join Framework Proc.ACM conference on Java Grande, JAVA'00, pp. 36-43 (2000).

[5] 岸央希，吉田明正: インナークラススケジューラを伴う階層統合型粗粒度タスク並列処理．情報処理学会第 73 回全国大会，5H-4，2011.

[6] Yoshida,A., Ochi,Y., Yamanouchi,N.: 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.