Vol.2014-ARC-211 No.8
2014/7/28

マルチコア上での Java Fork/Join Framework を用いた粗粒度タスク並列処理

神山 彰 [†1, a)]　吉田 明正 [†1, †2, †3, b)]

概要：マルチコアプロセッサ上での Java プログラムに対する並列処理手法として，階層統合型粗粒度タスク並列処理が提案されている．階層統合型粗粒度タスク並列処理では，異なる階層の粗粒度タスクをダイナミックスケジューラが各コアに割り当てることで，複数階層にまたがった粗粒度タスク間並列性を利用することが可能である．本稿では，Java Fork/Join Framework を導入することにより，階層統合型粗粒度タスク並列処理の並列 Java コードを生成する方法を提案する．本手法による並列 Java コードでは，各スレッドはワーカーキューから粗粒度タスクを取り出して実行し，その後，新たに実行可能な粗粒度タスクを Fork してワーカーキューに投入する．マルチコアプロセッサ Intel Xeon E5-2660 上で性能評価を行ったところ，ベンチマークプログラムにおいて高い実効性能を達成することが確認された．

キーワード：粗粒度タスク並列処理，階層統合型実行制御，Fork/Join Framework，コード生成，Java プログラム

1. はじめに

近年，コンピュータの計算速度性能のさらなる向上のために，マルチコアプロセッサを用いることが一般的となっている．マルチコアプロセッサは一般的な PC に限らず，組み込みシステムやスマートフォン，さらにはスーパーコンピュータに至るまで，広く利用されている．こうしたマルチコアプロセッサを利用した並列処理技術に関する研究は数多く行われており，従来のループ並列処理 [1], [2] に加えて，ループやサブルーチン等の粗粒度タスクレベルの並列性 [3], [4], [5], [6] を利用する粗粒度タスク並列処理が必要とされている．これにより，マルチコアプロセッサ上での並列処理において，高い実効性能を達成することが可能になる．

粗粒度タスク並列処理 [3] では，粗粒度タスク間の並列性を抽出することで階層型マクロタスクグラフを生成する．その後，各階層の粗粒度タスクを，グルーピングしたコアに階層ごとに割り当てて並列処理を行う手法である．また，粗粒度タスク並列処理で生成された階層型マクロタスクグラフ [3] を用いるとともに，対象プログラム中の各階層に存在する粗粒度タスクを統一的に取り扱うことで，異なる階層にまたがった粗粒度タスク間並列性を最大限に利用することのできる階層統合型実行制御手法 [7], [8] が提案されている．

本稿では，階層統合型粗粒度タスク並列処理を実現する並列 Java コードを，Java Fork/Join Framework[9] を用いて生成する手法を提案する．本手法を実装した並列 Java コードを生成し，マルチコアプロセッサ上で性能評価を行った結果，ベンチマークプログラムにおいて高い実効性能が達成され，提案手法の有効性が確認された．

本稿の構成は以下の通りとする．第 2 章では，関連研究について述べる．第 3 章では，階層統合型粗粒度タスク並列処理について述べる．第 4 章では，本手法で用いる Java Fork/Join Framework について述べる．第 5 章では，Java Fork/Join Framework を用いた階層統合型粗粒度タスク並列処理を実現するための並列 Java コードの構成について述べる．第 6 章では，本手法を実装した並列 Java コードによるマルチコア上での性能評価について述べる．第 7 章でまとめを述べる．

[†1] 明治大学大学院先端数理科学研究科
Graduate School of Advanced Mathematical Sciences, Meiji University
[†2] 明治大学総合数理学部ネットワークデザイン学科
Department of Network Design, School of Interdisciplinary Mathematical Sciences, Meiji University
[†3] 早稲田大学グリーンコンピューティングシステム研究機構
Green Computing Systems Research Organization, Waseda University
[a)] cs31006@meiji.ac.jp
[b)] akimasay@meiji.ac.jp

2. 関連研究

並列処理の対象言語としては，近年では PC や組み込みシステム等のソフトウェア開発の現場において Java 言語が広く利用されるようになってきており，Java プログラムによる並列処理への期待がより一層高まっている．Java プログラムの並列処理に関する研究は，ループのリストラクチャリングコンパイラ [10] や HPF のような配列分散を取り入れた HPJava[11]，ランタイムサポートによりスレッド間並列性を利用する zJava[12] や Jrpm[13]，マルチコアとヘテロジニアスアーキテクチャを利用するための Habanero-Java[14] 等が提案されている．これらはいずれも，複数階層の粗粒度タスク間並列性を統一的に利用することは困難であった．

3. 階層統合型粗粒度タスク並列処理

本章では，階層統合型粗粒度タスク並列処理 [7], [15] について述べる．

3.1 階層統合型実行制御の概要

階層統合型粗粒度タスク並列処理では，階層型マクロタスクグラフ（MTG）を生成し，マクロタスク（MT）を階層的に定義する．その後，ダイナミックスケジューラが最早実行可能条件を満たしたすべての階層のマクロタスクを統一的にコアに割り当てて処理する．

例えば，**図 1** のような階層型マクロタスクグラフとして表現されるプログラムを 4 コア上で実行したイメージは**図 2** のようになり，マクロタスク間の並列性が最大限に利用されている．ここで，図 1 の MT8 の最早実行可能条件は，MT5∧MT6∧MT7 と求めることができ，MT8 は MT5 と MT6 と MT7 の実行が終了した後に実行可能になるということを表している．**表 1** のマクロタスクの最早実行可能条件は，図 1 の階層型マクロタスクグラフ（MTG）に対応している．

3.2 階層的なマクロタスク生成

粗粒度タスク並列処理による実行では，まず，与えられたプログラム（全体を第 0 階層マクロタスクとする）を第 1 階層マクロタスク（MT）に分割する．マクロタスクは，基本ブロック，繰り返しブロック（for 文等のループ），サブルーチンブロック（メソッド呼び出し）の 3 種類から構成される [7]．次に，第 1 階層マクロタスク内部に複数のサブマクロタスクを含んでいる場合には，それらのサブマクロタスクを第 2 階層マクロタスクとして定義する．同様に，第 L 階層マクロタスク内部において，第 $(L+1)$ 階層マクロタスクを定義する．

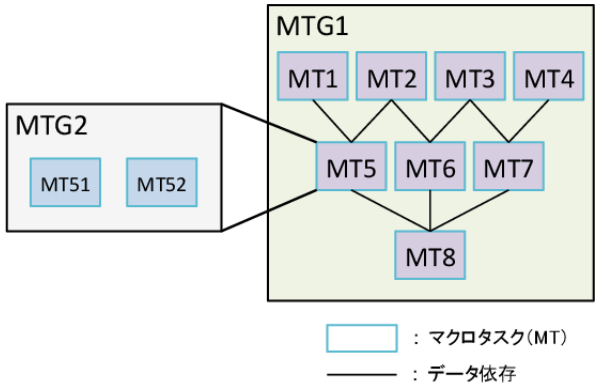

図 1 階層型マクロタスクグラフ（MTG）．

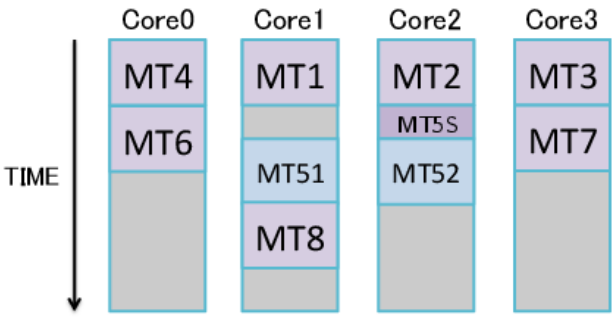

図 2 4 コア上での階層統合型粗粒度タスク並列処理の実行イメージ．

3.3 階層開始マクロタスク

階層統合型実行制御 [7] を適用する場合，全階層のマクロタスクを統一的に取り扱うため，階層開始マクロタスクを導入する．第 L 階層マクロタスクを内部に持つ上位の第 $(L-1)$ 階層マクロタスクを，第 L 階層用の階層開始マクロタスクとして取り扱う．この階層開始マクロタスクは，内部の第 L 階層マクロタスクの実行を開始するために使用される．この階層開始マクロタスクの導入により，当該階層のマクロタスクの実行が可能になったことが保証され，全階層のマクロタスクを同時に取り扱うことが可能となる．

3.4 階層統合型実行制御の最早実行可能条件

マクロタスク生成後，各階層のマクロタスク間の制御フローとデータ依存を解析し，階層型マクロフローグラフ [3] を生成する．次に，制御依存とデータ依存を考慮したマクロタスク間並列性を最大限に引き出すために，各マクロタスクの最早実行可能条件 [3] を解析する．最早実行可能条件は，制御依存とデータ依存を考慮したマクロタスク間の並列性を表しており，マクロタスクの実行制御に用いられる．ダイナミックスケジューリングの際には，ステート管理テーブルに保存された各マクロタスクの終了通知，分岐通知，最早実行可能条件を調べることにより，新たに実行可能なマクロタスクを検出することが可能となる [7]．

表 1　階層統合型実行制御の最早実行可能条件

MTG 番号	MT 番号	最早実行可能条件	終了通知
1	1	true	1
	2	true	2
	3	true	3
	4	true	4
	5†	1∧2	**5S**
	6	2∧3	6
	7	3∧4	7
	8	5∧6∧7	8
	9(EndMT)	8	9
2	51	**5S**	51
	52	**5S**	52
	53(CtrlMT/ExitMT)	51∧52	**53, 上位 MT**

注）†メソッド内部の第 2 階層 MTG の階層開始 MT

3.5　階層統合型マクロタスクスケジューリング

階層統合型実行制御によるマクロタスクスケジューリングでは，各マクロタスクは 3.4 節の最早実行可能条件を満たした後，レディマクロタスクキューに投入される．その後，レディマクロタスクキューから順に取り出されて，コア（プロセッサ）に割り当てられ実行される．なお，本手法ではレディマクロタスクキューとして，Java Fork/Join Framewrok のワーカーキューを用いる．

4.　Java Fork/Join Framework

Java Fork/Join Framework[9] は，Java SE 7[16] から導入された ExecutorService インタフェースを実装した並列処理フレームワークである．このフレームワークを利用することで，処理を小さな単位（タスク）に分割することができ，それらを複数のプロセッサを用いて処理することが可能となる．

Fork/Join では，main() メソッドにおいてスレッドプールを生成し，その際に，生成するワーカースレッド数を設定する．スレッドプール内に生成された各ワーカースレッドは独自のワーカーキューを持ち，それぞれが Fork されたタスクを実行することによって，Fork/Join による並列処理が行われる．実行されるタスクは，それぞれ compute() メソッドを処理しており，Java の Thread クラスや Runnable インタフェースの run() メソッドによる処理と同等なものと考えることができる．このフレームワークは，スレッド並列処理に比べて並列処理の記述が容易であり，また，コア数が増大した場合においても，ワークスティーリングによりロック競合の問題に対応することができる．

4.1　RecursiveAction クラスと RecursiveTask クラス

RecursiveAction クラス，もしくは RecursiveTask クラスは，Java の抽象基底クラスである ForkJoinTask クラスを継承する抽象クラスである．RecursiveAction クラスは，結果の出ない処理や計算の場合に用いられ，一方の RecursiveTask クラスは計算後に戻り値を返す場合に用いられる．Fork/Join Framework を使用する際には，どちらかのクラスを継承（extends）することにより，実行可能となる．これにより Fork/Join による並列処理を行うタスクとして使用できる．

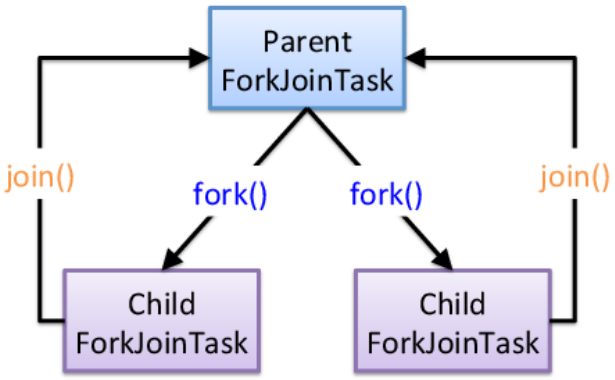

図 3　Fork/Join の動作.

4.1.1　compute() メソッド

このメソッドは，RecursiveAction クラス，もしくは RecursiveTask クラスでは抽象メソッドとして宣言されているため，継承後に本メソッド内にこのタスクで実行すべき処理を記述する．後述する fork() メソッドによってこのメソッドが処理される．

4.1.2　fork() メソッドと join() メソッド

fork() メソッドは，指定したタスクを非同期で実行するための調整を行う．ここで，同じタスクを再度 Fork する場合は，そのタスクの処理が終了し再初期化（reinitialize()）しなければ，再度 Fork することはできない．また，join() メソッドは，そのタスクが fork() された後，処理が終了するまで待機する（戻り値がある場合には，それを返す）．

例えば，図 3 に示すように，親タスクが fork() することにより指定されたタスクが実行される．子タスクは処理が終了すると，return により戻り値を返す．この間，親タスクは join() によって子タスクの終了を待機することとなる．

4.2　ワーカースレッドとワーカーキュー

Fork されたタスクは，各ワーカースレッド内に存在するワーカーキュー（両端キュー）の先頭に投入（プッシュ）される．ワーカースレッドが現在実行中のタスクを終了すると，自分のワーカーキューの先頭からタスクを取り出して実行する．ワーカーキュー内に存在するすべてのタスクはすでに実行可能状態となっているため，ワーカースレッドにおいてすぐに実行することが可能である．この際，自分のワーカーキューの先頭にアクセスするのは自分のみであるため，競合が起きることはない．

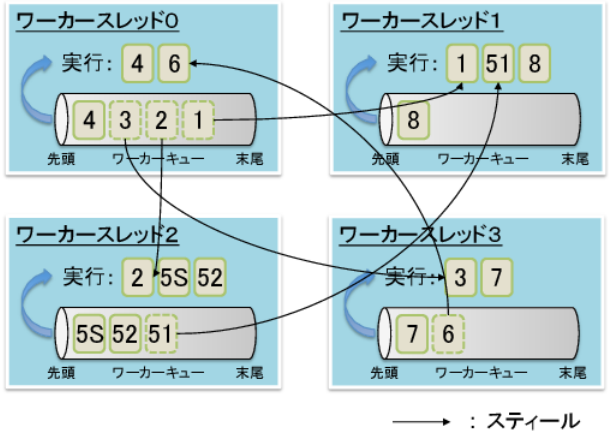

図 4　タスク実行とワークスティーリングの概念図.

4.3　ワークスティーリング

自分の持っているワーカーキューが空になり，処理するタスクがない場合は，他のワーカースレッドをランダムに選択し，そのワーカースレッドが持っているワーカーキューの末尾からタスクを取り出す．この仕組みをワークスティーリングと呼び，こうした操作のことをスティールするという．自ワーカースレッドはスティールしたタスクを即座に実行する．例えば，図 1 の階層型マクロタスクグラフ（MTG）で表されるプログラムを実行すると，**図 4** で示すように，それぞれのワーカースレッドによるタスクの実行とワーカーキューへのタスクの投入，また各ワーカースレッドによるワークスティーリングが行われる．ワークスティーリングは，状況によっては複数タスクに対してまとめて行われるため，ワーカースレッド間での競合は稀である．

5.　Java Fork/Join Framework による並列 Java コード生成

Java Fork/Join Framework を用いた粗粒度タスク並列処理に関する研究として，ループを含まないプログラムを対象とした手法が提案されている [17]．本章では，前章で述べた Java Fork/Join Framework を用いた階層統合型粗粒度タスク並列処理を実現する並列 Java コードの生成手法について述べる．

5.1　Fork/Join 型並列 Java コードの構成

Java Fork/Join Framework を用いて生成した並列 Java コードの例を**図 5** に示す．本並列 Java コード（Fork/Join 型並列 Java コード）は，(1) マクロタスク管理テーブルクラスである Data クラス（図 5 の 1〜7 行目），(2) ユーザ定義クラスやメソッドを含む Other クラス（クラス名はユーザが定義する，同 9〜35 行目），(3) 並列 Java コードの main() メソッドを含む Mainp クラス（同 39〜93 行目）から構成される．Mainp クラス内部には，さらに，Fork/Join 処理を開始するための第 0 階層 MTG 専用クラスである MTG0 クラスや，実際の処理でタスクとして Fork されることとなる Main クラス（オリジナルプログラムの main() メソッドに対応）が含まれている．

5.1.1　Data クラス

ここでは，Fork されるタスクの ArrayList への登録や管理，Fork されたタスクの状態管理を行うマクロタスク管理テーブルが宣言されている．また，後続マクロタスクを Fork することが可能かを判定する forkCheck() メソッド（図 5 の 4〜6 行目）があり，後続マクロタスクの番号を引数として渡すことで，最早実行可能条件のチェックを行っている．

5.1.2　Other クラス

ユーザが独自に定義したクラス内において Fork 対象となるタスクが存在する場合，コード構成としては Main クラスと同じ構造となるため，Other クラス内部のタスクについても同じような操作で Fork 処理を行うことができる．ユーザ定義クラスは複数あっても問題ない．

5.1.3　Mainp クラス

Mainp クラスでは，main() メソッド（図 5 の 88〜92 行目）においてスレッドプールの生成とスレッド数の指定を行う．その後，MTG0 クラスを invoke() メソッド（同 91 行目）で呼び出すことにより，Fork/Join による並列処理が開始される．第 0 階層として用意された MTG0 クラス（同 40〜48 行目）内部の compute() メソッド（同 43〜47 行目）が，今度は Mainp クラス内部の Main クラス（同 49〜86 行目）の階層開始マクロタスクを fork() することで，第 1 階層（MTG1）の処理が開始される．ここで，MTG0 は helpQuiesce() メソッド（同 45 行目）により処理を移し，また join() メソッド（同 46 行目）により，すべてのタスクの処理が終了するのを待機する．Fork された Main クラスの各タスクは，まず compute() メソッド（同 56〜58 行目）を実行し，自分のマクロタスク番号を基にどのマクロタスクを処理すればよいのかを判定する．その後，各マクロタスクごとのメソッドを呼び出して処理が行われる．

5.2　Fork-Template 形式

本節では，1 つのマクロタスク実行とそのマクロタスクに付随したスケジューリング管理の一連の動作を，Fork-Template と呼ぶ．図 5 の Mainp クラス内の Main クラスや，Other クラス内部の各クラスは，Fork-Template 形式であるため，本実装によって Fork 処理を行うことができる．Fork-Template 形式の内部には，このタスクで処理すべきマクロタスクの番号を格納する変数が用意されており，Fork-Template 形式のインスタンスが生成される際，コンストラクタに引数として値を渡すことで設定される．また，fork() されたタスクは，内部に存在する compute() メソッドを実行する．ここでは，処理すべきマクロタスク

```
01: class Data{
02:     static ArrayList<ArrayList<RecursiveAction>> mtg
:                                   = new ArrayList<ArrayList<RecursiveAction>>();
03:     ステート管理テーブル(MT終了・分岐)宣言;
04:     synchronized static boolean forkCheck(int mtnum){
05:         最早実行可能条件によるmtnum番MTのフォークチェック;
06:     }
07: }
08:
09: class Other{ //ユーザ定義クラスとメソッド
10:     public static class Other_inner extends RecursiveAction{ //MTG2
11:         //オリジナルプログラムではメソッドだった部分
12:         自MTG・MT, 上位MTG・MT, 当該MTで実行すべきMT番号用変数宣言;
13:         Other_inner( 各引数 ){ //コンストラクタ
14:             自分の処理すべきMT番号を設定;
15:                 自MTG・MT, 上位MTG・MTの設定;
16:         }
17:         protected void compute(){
18:                 MT番号のMTを実行;
19:         }
20:         public void mtStart(){ //階層開始MT
21:                 後続MTのreinitialize()とfork();
22:         }
23:         public void mt1(){ //MT1
24:                 このMTで処理すべき内容;
25:                 該当ステートをtrueにする;
26:                 後続MTのforkCheck()を行い, trueならばreinitialize()してfork()する;
27:         }
28:         public void mt2(){ ・・・} //MT2
29:         ・・・
30:         public void mtExit(){ //ExitMT
31:                 上位階層後続MTのreinitialize()とfork();
32:         }
33:     }
34:     ・・・
35: }
36:
37: ・・・
38:
39: class Mainp{ //Mainpクラス
40:     static class MTG0 extends RecursiveAction{ //Fork実行開始MTG, MTG0
41:         コンストラクタにおいて, 新たなMTGを生成する;
42:         Mainクラスの階層開始MTをmtgで管理する;
43:         protected void compute(){
44:                 Mainクラス(第1階層)の階層開始MTをフォークする;
45:                 helpQuiesce()でタスク処理へ移行;
46:                 join()でFork実行の終了を待機する;
47:         }
48:     }
49:     public static class Main extends RecursiveAction{ //MTG1
50:         //オリジナルプログラムではmainメソッドだった部分
51:         自MTG・MT, 上位MTG・MT, 当該MTで実行すべきMT番号用変数宣言;
52:         Main( 各引数 ){ //コンストラクタ
53:                 自分の処理すべきMT番号を設定;
54:                 自MTG・MT, 上位MTG・MTの設定;
55:         }
56:         protected void compute(){
57:                 MT番号のMTを実行;
58:         }
59:         public void mtStart(){ //階層開始MT
60:                 後続MTをmtgで管理しreinitialize()とfork();
61:         }
62:         public void mtForStart(){ //for文開始MT
63:                 for文内MTのインスタンスを生成しmtgで管理する;
64:                 生成した後続MTのreinitialize()とfork();
65:         }
66:         public void mt1(){
67:                 このMTで処理すべき内容;
68:                 該当ステートをtrueにする;
69:                 後続MTのforkCheck()を行い, trueならばreinitialize()してfork()する;
70:         }
71:         public void mt2(){
72:                 メソッド内MTのインスタンスを生成しmtgで管理する;
73:                 生成した後続MTのreinitialize()とfork();
74:                 //Otherクラス内Other_innerクラスのmtStart()をForkする
75:         }
76:         public void mt2_return(){ //return用MT
77:                 //Otherクラス内Other_innerクラスのmtExit()はここに返ってくる
78:                 後続MTのreinitialize()とfork();
79:         }
80:         public void mtForCtrl(){ //for文制御MT
81:                 繰り返し判定;
82:                 mt1(), もしくは後続MTのreinitialize()とfork();
83:         }
84:         ・・・
85:         public void mtEnd(){ } //EndMT
86:     }
87:     ・・・
88:     public static void main(String[] args){ //mainメソッド
89:         実行時にスレッド数を指定;
90:         スレッドプール生成;
91:         invoke()によりFork処理開始MTGを実行;
92:     }
93: }
```

図 5　Fork/Join 型並列 Java コード.

番号を基に該当のメソッドを呼び出す処理を行っている．呼び出されたメソッドによって，それぞれのマクロタスクとしての個別の処理が行われる．もし，Other クラスが複数あったとしても，プログラム構成は同じである．

Fork-Template 形式に基づいたクラスは，オリジナルプログラムの各メソッド毎に作られ，そのメソッド内のマクロタスクの実行は，Fork-Template 形式のクラスより生成されるインスタンスを Fork することにより実現される．

5.3　マクロタスク処理

Fork-Template 形式のクラスにある各マクロタスク用メソッドにおいて，当該マクロタスクとしての実際の処理が行われる．マクロタスクとしての処理が終了すると，MT 管理テーブルを更新し処理の終了を通知する．また，forkCheck() を行うことで後続 MT の最早実行可能条件を判定し，実行可能であるかチェックを行う．この際，synchronized() による排他制御をかけているため，最早実行可能条件の判定ミスやタスクの実行に齟齬が生じないようにしている．最早実行可能条件によって実行可能と判定された後続 MT は，このマクロタスクによる再初期化（reinitialize()）が行われ fork() される．すると，Fork された後続 MT は現在実行中のワーカースレッド内にあるワーカーキューの先頭に投入される．その後，ワーカーキューから取り出されるか，もしくはワークスティーリングで横取りされることにより，後続 MT として処理される．

5.3.1　繰り返し・メソッド呼び出しへの対応

for 文等の繰り返しやメソッド呼び出しを行う部分については，各イタレーションや呼び出しごとに新たな MTG を生成してしまい，MTG 数の増大が懸念される．この問題に対処するため，一つ前のイタレーションや呼び出しの処理で終了した MTG を reinitialize() により再利用している．reinitialize() を行うことで内部データは初期化されるため，データの不具合が発生することはない．これにより，MTG 数の増加を防ぐことができる．

5.3.2　マクロタスク融合

制御用のダミーマクロタスク（CtrlMT，RepMT，ExitMT）はオーバヘッド軽減のため，マクロタスク融合により 1 つのマクロタスクとして扱う．

6.　マルチコア上での粗粒度タスク並列処理の性能評価

Java Fork/Join Framework による階層統合型粗粒度タスク並列処理を実現する並列 Java コードの性能評価を，マルチコアプロセッサシステム DELL PowerEdge R620 上で行う．

6.1　性能評価環境

性能評価環境として使用する DELL PowerEdge R620 は，

表 2　各性能評価プログラムの概要

プログラムの種類	Jacobi	Crypt	MolDyn
並列化後のコード長	1015	2537	5407
MTG 数	4	3	1
分割後の MT 数	40	74	205
逐次処理時間［ms］（HotSpot 最適化あり）	5523	1239	27583

Intel Xeon E5-2660（8 コア，2.20GHz）を搭載し，64GB のメモリから構成されている．OS は CentOS6.5 を導入し，Java 処理系は JDK1.7 となっている．本手法の性能評価には，6.2 節で述べる数値計算プログラムと，6.3 節で述べるベンチマークプログラムを用いる．それぞれのプログラムの概要を**表 2** に示す．各プログラムの並列 Java コードは，Java Fork/Join Framework による実装がなされており，階層統合型粗粒度タスク並列処理を実現できる．これらの並列 Java コードを JDK1.7 のコンパイラ javac でコンパイルし，JVM 上で実行して性能評価を行う．

6.2　数値計算プログラムによる性能評価

本性能評価では，Java で作成した数値計算プログラムとして，ヤコビ法の計算プログラム（表 2 の Jacobi）を用いる．ヤコビ法は一般的に n 元連立一次方程式の求解法として用いられる．本プログラムでは，対象とする行列のサイズを 10000 × 10000 としており，収束ループ内部は 3 つのループ（for 文）と基本ブロックから構成される．このプログラムに対して提案手法を実装し，Java Fork/Join Framework による階層統合型粗粒度タスク並列処理を実現する並列 Java コードを生成する．その際，並列化可能ループ部分については 8 分割し，それぞれをマクロタスクとして定義している．また，ループやメソッド呼び出しの部分は，Fork-Template クラスのインスタンスとして生成されたタスクの再利用を行うコードとして実装されている．その後，生成した並列 Java コードを javac でコンパイルし，JVM 上で実行する．実行時には HotSpot 最適化を適用している．

ヤコビ法による n 元連立一次方程式の求解プログラムの実行結果を**図 6** に示す．図 6 に示すとおり，Intel Xeon E5-2660 上での並列処理では 1 スレッドでの実行に比べて，8 スレッドで 6.92 倍の速度向上が得られた．この場合，逐次処理（1 コアで並列化を行っていないプログラムの処理）と比べて 6.72 倍の速度向上が得られた．これより，提案手法の有効性が確認された．

6.3　ベンチマークプログラムによる性能評価

次に，Java Grande Forum Benchmark Suite[18] より提供されている表 2 の Crypt プログラムと MolDyn プログラムを並列化したプログラム（並列 Java コード）を用いて性能評価を行う．

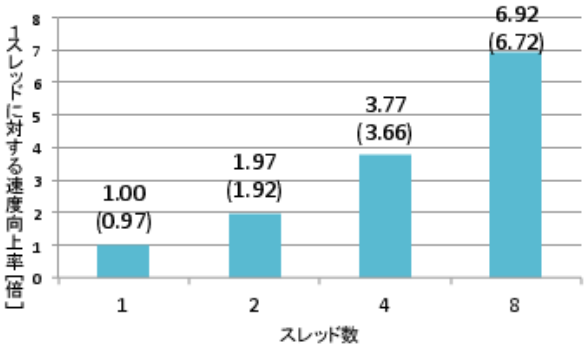

図 6　Intel Xeon E5-2660 上でのヤコビ法プログラムの階層統合型粗粒度タスク並列処理（逐次比を括弧内に記載）．

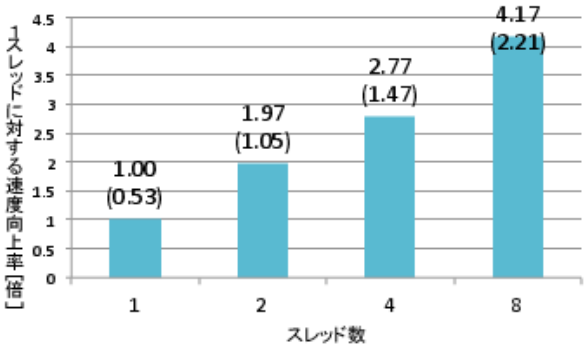

図 7　Intel Xeon E5-2660 上での Crypt プログラムの階層統合型粗粒度タスク並列処理（逐次比を括弧内に記載）．

6.3.1　Crypt プログラム

Crypt プログラムは，IDEA(International Data Encryption Algorithm) と呼ばれる，共通鍵暗号方式によるデータ暗号化アルゴリズムを用いた暗号化（encrypt）と復号化（decrypt）の処理を，N バイトの配列上で行うプログラムである．これは共通鍵によるブロック暗号法の一方式であり，鍵は 128 ビットとなっている．1992 年にスイス工科大学の James L.Massey と Xuejia Lai，スイスの Ascom 社によって開発された．

このプログラムに対して提案手法を実装し，Java Fork/Join Framework による階層統合型粗粒度タスク並列処理を実現する並列 Java コードを生成する．その際，並列化可能ループ部分については 32 分割し，それぞれをマクロタスクとして定義している．その後，生成した並列 Java コードを javac でコンパイルし，JVM 上で実行する．性能評価ではクラス B のプログラムを使用し，配列の大きさを N=2000 万としている．また，JVM 実行時には HotSpot 最適化を適用している．

Crypt プログラムの実行結果を**図 7** に示す．図 7 に示すとおり，Intel Xeon E5-2660 上での並列処理では 1 スレッドでの実行に比べて，8 スレッドで 4.17 倍の速度向上が得られた．この場合，逐次処理と比べて 2.21 倍の速度向上が得られた．

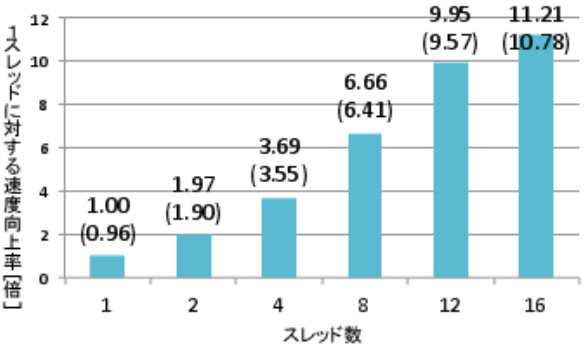

図 8 Intel Xeon E5-2660 上での MolDyn プログラムの階層統合型粗粒度タスク並列処理（逐次比を括弧内に記載）.

6.3.2 MolDyn プログラム

MolDyn プログラムは，周期境界条件において，3 次元空間体積での Lennard-Jones ポテンシャル下における N 個のアルゴン原子の相互作用の挙動をモデル化するシンプルな N 体問題のコードである．このプログラムに対して提案手法を実装し，Java Fork/Join Framework による階層統合型粗粒度タスク並列処理を実現する並列 Java コードを生成する．その際，並列化可能ループ部分については 26 分割し，それぞれをマクロタスクとして定義している．また，ループやメソッド呼び出しの部分では生成したタスクのインスタンスを再利用しており，MTG 数や MT 数の増大を防いでいる．その後，生成した並列 Java コードを javac でコンパイルし，JVM 上で実行する．性能評価ではクラス B のプログラムを使用し，データサイズを N=8788 としている．また，JVM 実行時には HotSpot 最適化を適用している．

MolDyn プログラムの実行結果を**図 8** に示す．図 8 に示すとおり，Intel Xeon E5-2660 上での並列処理では 1 スレッドでの実行に比べて，8 スレッドで 6.66 倍（逐次処理と比べて 6.41 倍），16 スレッドで 11.21 倍（逐次処理と比べて 10.78 倍）の速度向上が得られた．これより，提案手法の有効性が確認された．

7. おわりに

本稿では，階層統合型粗粒度タスク並列処理において，Java Fork/Join Framework を用いた並列 Java コードの生成手法を提案した．これにより，階層統合型粗粒度タスク並列処理におけるダイナミックスケジューリングを，Java Fork/Join Framework により実装することを可能にした．

Java Fork/Join Framework を用いた並列 Java コードを生成し，マルチコアプロセッサ DELL PowerEdge R620 上で実行したところ，ヤコビ法プログラムの場合は 8 スレッドで 6.92 倍，Crypt プログラムでは 8 スレッドで 4.17 倍，MolDyn プログラムでは 16 スレッドで 11.21 倍の速度向上が得られ，高い実効性能が達成された．これらの結果より，Java Fork/Join Framework による階層統合型粗粒度タスク並列処理の並列 Java コードの有効性が確認された．

今後の課題としては，本手法による並列 Java コードを自動生成する並列化コンパイラの開発が挙げられる．

参考文献

[1] M. Wolfe: “High performance compilers for parallel computing”, *Addison-Wesley Publishing Company* (1996).
[2] R. Eigenmann, J. Hoeflinger and D. Padua: “On the automatic parallelization of the Perfect benchmarks”, *IEEE Trans. on Parallel and Distributed System*, Vol. 9, No. 1, pp. 5–23 (1998).
[3] 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).
[4] 間瀬正啓，木村啓二，笠原博徳：“マルチコアにおける Parallelizable C プログラムの自動並列化”，情報処理学会研究報告, 2009-ARC-184-15 (2009).
[5] W. Thies, V. Chandrasekhar and S. Amarasinghe: “A practical approach to exploiting coarse-grained pipeline parallelism in C programs”, *Proc. IEEE/ACM Int. Symposium on Microarchitecture*, pp. 356–368 (2007).
[6] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, Gonzalez M. and J. Labarta: “Thread Fork/Join techniques for multi-level parallelism exploitation in NUMA multiprocessors”, *Proc. Int. Conference on Supercomputing*, pp. 294–301 (1999).
[7] 吉田明正：“粗粒度タスク並列処理のための階層統合型実行制御手法”，情報処理学会論文誌，Vol. 45, No. 12, pp. 2732–2740 (2004).
[8] 越智佑樹，山内長承，吉田明正：“階層統合型粗粒度タスク並列処理のための選択的静的データ構造を用いた並列 Java コード生成手法”，情報処理学会研究報告, 2013-ARC-206-2 (2013).
[9] Oracle: “The Java™ Tutorials > Essential Classes > Concurrency > Fork/Join”, [http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html].
[10] A.J.C. Bik and D.B. Gannon: “Javar a prototype Java restructuring compiler”, *Concurrency: Practice and Experience*, Vol. 9, No. 11, pp. 1181–1191 (1997).
[11] S.B. Lim, H. Lee, B. Carpenter and G. Fox: “Runtime support for scalable programming in Java”, *J. Supercomputing*, Vol. 43, pp. 165–182 (2008).
[12] B. Chan and T.S. Abdelrahman: “Run-time support for the automatic parallelization of Java programs”, *J. Supercomputing*, Vol. 28, pp. 91–117 (2004).
[13] M.K. Chen and K. Olukotun: “The Jrpm system for dynamically parallelizing Java programs”, *Proc. ISCA-30*, pp. 434–446 (2003).
[14] V. Cavé, J. Zhao, J. Shirako and V. Sarkar: “Habanero-Java: the new adventures of old X10”, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ACM, pp. 51–61 (2011).
[15] A. Yoshida and T. Ozawa: “Layer-Unified Coarse Grain Task Parallel Processing for Java Programs”, *Proc. 16th International Workshop on Compilers for Parallel Computing* (2012).
[16] Oracle: “Java™ Platform, Standard Edition 7 API Specification”, [http://docs.oracle.com/javase/7/docs/api/].
[17] 笠松拓史，吉田明正：“階層統合型粗粒度タスク並列処理のための Fork/Join を用いた並列 Java コード生成”，

情報処理学会第 73 回全国大会講演論文集，pp. 123–125 (2011).

[18] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty and R. A. Davey: “A benchmark suite for high performance Java”, *Concurrency - Practice and Experience*, Vol. 12, No. 6, pp. 375–388 (2000).