Vol.2015-ARC-214 No.6
2015/1/30

Java Fork/Join Framework を用いた粗粒度並列処理コードの自動生成

神山 彰 [†1, a)]　吉田 明正 [†1, †2, †3, b)]

概要：Java 言語の並列プログラムは，Java スレッドによる実装が行われてきたが，最近では Java Fork/Join Framework による実装が可能となっている．Java プログラムのマルチコア上での並列処理手法として，階層統合型実行制御を用いた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により，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 実装によるダイナミックスケジューラを利用する並列 Java コードを，開発した並列化コンパイラを用いて自動生成する．並列化コンパイラにより生成された並列 Java コードは，実行可能な粗粒度タスクを Fork によりワーカーキューへ投入し，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する．性能評価では，並列化コンパイラにより生成された並列 Java コードを用いて，マルチコアプロセッサ Intel Xeon E5-2660 上で並列実行を行った．その結果，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，並列化コンパイラ，Java プログラム

1. はじめに

マルチコアプロセッサは計算速度をより向上させるために，スーパーコンピュータから PC，タブレット，スマートフォンに至るまで，幅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となる．

粗粒度タスク並列処理 [3] 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の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．

ここで，階層統合型粗粒度タスク並列処理の並列 Java コードにおけるダイナミックスケジューラの実装方法として，Java スレッドを用いた実装，あるいは，Java Fork/Join Framework[9] を用いた実装 [10] が提案されている．しかし，Java Fork/Join Framework を用いた実装の場合，そのコード生成はユーザが手動で行う必要があるため，並列化コンパイラによる自動化が望まれていた．

そこで，本稿では Java Fork/Join Framework を用いた粗粒度並列処理コードの自動生成手法を提案するとともに，その並列化コンパイラを開発した．本並列化コンパイラにより自動生成された並列 Java コードは，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し，マルチコアプロセッサ上での性能評価から，高い実効性能を達成することが確認されている．

[†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

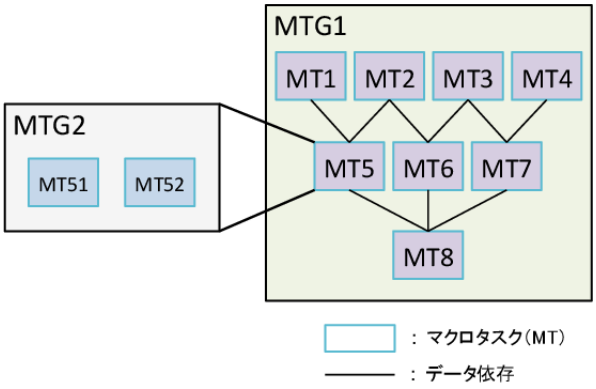

図 1　階層型マクロタスクグラフ（MTG）.

本稿の構成は以下の通りである．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，並列 Java コードに導入する Java Fork/Join Framework について述べる．第 5 章では，Java Fork/Join Framework を用いた粗粒度並列処理コードについて述べる．第 6 章では，その並列 Java コードを自動生成する並列化コンパイラについて述べる．第 7 章では，開発した並列化コンパイラにより生成された並列 Java コードを用いて性能評価を行う．第 8 章でまとめを述べる．

2. 関連研究

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いる．そのため，Java プログラムを用いた並列処理への期待がより一層高まっている．

Java による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 [11] や HPF のような配列分散を取り入れた HPJava[12]，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[13] や Jrpm[14]，科学技術計算のために複素数型を導入した Habanero-Java[15] 等が提案されている．しかし，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], [8] 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）[3] を生成し，マクロタスク（MT）を階層的に定義する．その後，最早実行可能条件 [7] を満たした全階層のマクロタスクを，ダイナミックスケジューラが統一的にコアに割り当てて実行する．

例えば，**図 1** のような階層型マクロタスクグラフで表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** のようになる．このとき，マクロタスク間の並列性が最大限に利用されていることがわかる．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る．これは，MT5 と MT6 と MT7 の実行が終了した後に，MT8 の実行が可能になるということを表している．**表 1** に示す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に対応している．

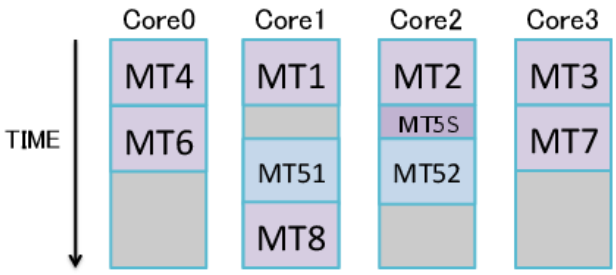

図 2　4 コア上での階層統合型粗粒度タスク並列処理の実行イメージ.

3.2 階層的なマクロタスク生成

粗粒度タスク並列処理では，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．

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サブマクロタスクとして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．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

マクロタスクの生成後，各階層における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．その後，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，各マクロタスクの最早実行可能条件 [3] を解析する．表 1

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(ExitMT)	51∧52	**53, 上位 MT(5)**

†: メソッド内部の第 2 階層 MTG の階層開始 MT

のような最早実行可能条件は，マクロタスクの実行制御に用いられる．ここで，MT5 は図 1 に示すように MT51 と MT52 で構成されたメソッドの呼び出しに対応する．それゆえ，MT5 は階層開始マクロタスクとして動作しており，階層開始マクロタスクの処理を終了した時に 5S の終了通知を発行する．一方，MT5 のメソッド呼び出しの終了通知は，メソッド内の MT53 が終了通知 5 を発行する．

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].

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，本研究においては，レディマクロタスクキューとして，後述する Java Fork/Join Framework のワーカーキューを用いる．

4.　Java Fork/Join Framework

本章では，後述する並列化コンパイラにより生成される並列 Java コードの実装で用いる Java Fork/Join Framework[9] について述べる．

4.1　Java Fork/Join Framework の概要

Java Fork/Join Framework[9] は，Java SE 7[16] に導入された 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．本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，それらを複数のコアを用いて処理することが可能となる．

Fork/Join Framework では，まずスレッドプールを作成し，その中にワーカースレッドを生成する．その際，生成するワーカースレッド数を指定する．スレッドプール内に生成された各ワーカースレッドは，それぞれ独自のワーカーキューを持っており，Fork されたタスクは各々のワーカースレッドのワーカーキューへ投入される．その後，各ワーカースレッドがワーカーキューからタスクを取り出して実行する．実行されるタスクは内部で compute() メソッドを処理しており，これは 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．

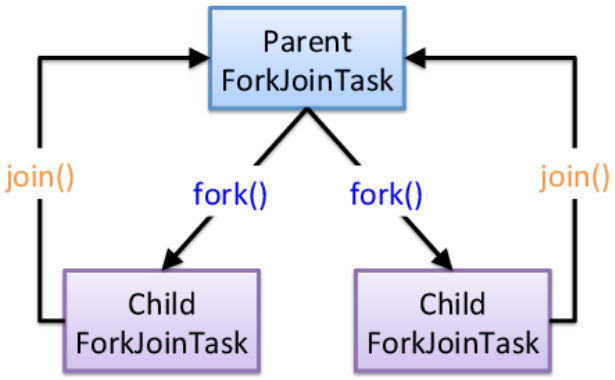

図 3　Fork/Join の動作.

本フレームワークを用いることで，スレッド並列処理に比べて並列処理の記述が容易になるというメリットがある．また，本フレームワークの特徴的な機能であるワークスティーリングの仕組みにより，ロック競合の問題に対応することが可能である．

4.2　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，Fork したタスクが戻り値を持たない場合に用いられ，一方の RecursiveTask クラスは Fork したタスクが戻り値を持つ場合に用いられる．Fork/Join Framework による実装を行う際は，どちらかのクラスを継承することにより実現可能となる．

4.2.1　compute() メソッド

compute() メソッドは，RecursiveAction クラスや RecursiveTask クラスにおいて，抽象メソッドとして宣言されている．そのため，継承後に，当該タスクで実行すべき処理を本メソッド内に記述する．このメソッドは fork() メソッドによって実行される．

4.2.2　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た後，再初期化（reinitialize()）しなければならない．また，join() メソッドは，

表 2　階層統合型粗粒度並列処理コードの実装比較

実 装 方 法	マクロタスク処理	スケジューリングのためのマクロタスク管理	スケジューリング処理
Fork/Join 実装（本研究）	並列 Java コードの Fork-Template	並列 Java コードの Fork-Template	ワークスティーリングを伴う Fork/Join スケジューラ
Runnable 実装（従来）	並列 Java コードのマクロタスク処理部分	並列 Java コードのマクロタスク管理部分	並列 Java コードのスケジューリング処理部分

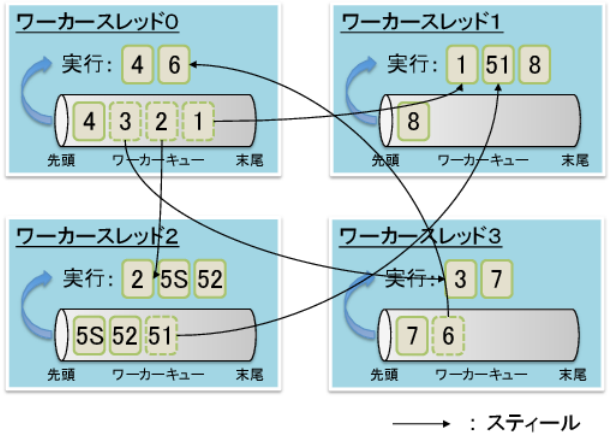

図 4　タスクの実行とワークスティーリングの概念図.

そのタスクが fork() された後，処理が終了するまで待機する（戻り値がある場合はそれを返す）.

例えば，図 3 に示すように，親タスクが子タスクを fork() することで子タスクの処理が行われる．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.

4.3　ワーカースレッドとワーカーキュー

ワーカースレッド内に存在するワーカーキューは両端キュー（deque）であり，Fork されたタスクは，各ワーカースレッドが持っているワーカーキューの先頭に投入（プッシュ）される．ワーカースレッドにおいて現在実行中のタスクが終了すると，自分のワーカーキューの先頭からタスクを取り出す．このとき，ワーカーキュー内に格納されているすべてのタスクは実行可能状態であるため，取り出したタスクをワーカースレッドで即座に実行することができる．ここで，ワーカーキューの先頭にアクセスできるのは，そのワーカーキューを持っているワーカースレッドのみであるため，競合が発生することはない.

4.4　ワークスティーリング

各ワーカースレッドは，自分の持っているワーカーキューが空になり実行するタスクがなくなると，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．こうした仕組みをワークスティーリングと呼び，この操作のことをスティール（横取り）という．他のワーカースレッドのワーカーキューからスティールしたタスクは，ワーカースレッド上で即座に実行される.

例えば，図 1 の階層型マクロタスクグラフで表されるプログラムを実行すると，図 4 のようになる．この図では，各ワーカースレッドによるワーカーキューへのタスクの投入と実行，また，ワークスティーリングの流れを示している．スティールは，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発生する競合は稀である.

5.　Java Fork/Join Framework を用いた粗粒度並列処理コード

粗粒度並列処理コードの構成として，従来は表 2 の Runnable 実装がなされていた．Runnable 実装でのマクロタスク処理やスケジューリングのためのマクロタスク管理，スケジューリング処理は，並列化コンパイラにより生成される並列 Java コードにおいて，それぞれマクロタスク処理部分，マクロタスク管理部分，スケジューリング処理部分にて行われていた [8].

一方，本研究における粗粒度並列処理コードでは，表 2 の Fork/Join 実装がなされており，マクロタスク処理とスケジューリングのためのマクロタスク管理は後述する Fork-Template を用いている．また，スケジューリング処理については，Java Fork/Join Framework のワークスティーリングを伴う Fork/Join スケジューラを利用している．本章では，このような Java Fork/Join Framework を用いた粗粒度並列処理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 を用いて生成される並列 Java コードの例を図 5 に示す．この並列 Java コードは図 6 のマクロタスクグラフに対応している．本並列 Java コード（Fork/Join 型並列 Java コード）は，以下の 3 つから構成される.

(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 1〜11 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 12〜35 行目）
(3) 並列 Java コードの main() メソッドを含む Mainp クラス（同 37〜97 行目）

ここで，(3) Mainp クラス内には，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

```
1:  class Data{ //マクロタスク管理テーブルクラス
2:      static ArrayList<ArrayList<RecursiveAction>> mtg
3:                                  = new ArrayList<ArrayList<RecursiveAction>>();
4:      static Map<Integer, ArrayList<Integer>> reverseEEC
5:                                  = new HashMap<Integer, ArrayList<Integer>>();
6:      マクロタスク実行管理テーブル(MT終了・分岐)宣言;
7:      逆読み最早実行可能条件の登録;
8:      static boolean forkCheck(int mtnum){
9:          mtnum番マクロタスクの最早実行可能条件チェック;
10:     }
11: }
12: class Other{ //ユーザ定義クラスとメソッド
13:     public static class Other_inner extends RecursiveAction{ //MTG2
14:         //並列化前の入力プログラムではメソッドに対応
15:         当該MTG・MT, 上位MTG・MT, 当該MTで実行すべきMT番号用変数宣言;
16:         Other_inner( 各引数 ){ //コンストラクタ
17:             当該MTで実行すべきMT番号を設定;
18:             当該MTG・MT, 上位MTG・MTの設定;
19:         }
20:         protected void compute(){
21:             設定されたMT番号に該当するマクロタスクを実行;
22:         }
23:         public void mt2_1(){
24:             マクロタスク処理;
25:             マクロタスク実行管理テーブルの更新;
26:             データ依存後続マクロタスクのFork;
27:         }
28:         public void mt2_2(){ ・・・ }
29:         ・・・
30:         public void mtExit(){ //ExitMT
31:             上位階層後続マクロタスクのFork;
32:         }
33:     }
34:     ・・・
35: }
36: ・・・
37: class Mainp{ //Mainpクラス
38:     static class MTG0 extends RecursiveAction{ //Fork/Join処理開始MTG
39:         Mainクラス(第1階層)の階層開始MTをmtgで管理;
40:         protected void compute(){
41:             Mainクラス(第1階層)の階層開始MTをFork;
42:             helpQuiesce()でタスク処理へ移行;
43:             join()でタスク実行の終了を待機;
44:         }
45:     }
46:     public static class Main extends RecursiveAction{ //MTG1
47:         //並列化前の入力プログラムではmain()メソッドに対応
48:         当該MTG・MT, 上位MTG・MT, 当該MTで実行すべきMT番号用変数宣言;
49:         Main( 各引数 ){ //コンストラクタ
50:             当該MTで実行すべきMT番号を設定;
51:             当該MTG・MT, 上位MTG・MTの設定;
52:         }
53:         protected void compute(){
54:             設定されたMT番号に該当するマクロタスクを実行;
55:         }
56:         public void mtStart(){ //階層開始MT
57:             マクロタスク実行管理テーブルの更新;
58:             データ依存後続マクロタスクのFork;
59:         }
60:         public void mtForStart(){ //for文開始MT
61:             条件判断;
62:             マクロタスク実行管理テーブルの更新;
63:             for文内後続マクロタスクのFork;
64:         }
65:         public void mt1_1(){
66:             マクロタスク処理;
67:             マクロタスク実行管理テーブルの更新;
68:             データ依存後続マクロタスクのFork;
69:         }
70:         public void mt1_2(){
71:             メソッド内後続マクロタスクのFork;
72:             //Otherクラス内Other_innerクラスの後続MTをForkする
73:         }
74:         public void mt1_3(){
75:             //Otherクラス内Other_innerクラスのExitMTによりForkされる
76:             データ依存後続マクロタスクのFork;
77:         }
78:         public void mtForCtrl(){ //for文制御MT
79:             繰り返し判定;
80:             RepeatMT, もしくはExitMTをFork;
81:         }
82:         public void mtForRepeat(){ //RepeatMT
83:             mt1_1()をForkしてfor文を繰り返す;
84:         }
85:         public void mtForExit(){ //ExitMT
86:             データ依存後続マクロタスクのFork;
87:         }
88:         ・・・
89:         public void mtEnd(){ } //EndMT
90:     }
91:     ・・・
92:     public static void main(String[] args){ //mainメソッド
93:         実行時にワーカースレッド数を指定;
94:         スレッドプールの生成;
95:         invoke()でFork/Joinによる並列処理を開始;
96:     }
97: }
```

図 5 並列化コンパイラにより生成される Fork/Join 型並列 Java コードの例.

スや，実際の並列処理においてタスクとして Fork される Main クラス（並列化前の入力プログラムにおける main() メソッドに対応）も含まれる．

5.1.1 Data クラス

Data クラスでは，Fork されるタスクの ArrayList への登録や管理，Fork されたタスクの実行管理を行うマクロタスク実行管理テーブルの宣言を行っている．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 8〜10 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う．

さらに，各マクロタスクの後続 MT として実行可能になる可能性のあるマクロタスクの候補（逆読み最早実行可能条件）を，HashMap へ登録し管理を行っている．これは，5.3 節で述べる実行管理付きマクロタスク処理コードにおいて，データ依存後続マクロタスクを Fork する際に，forkCheck() メソッドとともに用いられる．

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 対象となるマクロ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 クラス内部のマクロ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

プログラムの実行が開始されると，Mainp クラス内に存在する main() メソッド（図 5 の 92〜96 行目）の処理が行われる．main() メソッドでは，スレッドプールを生成しワーカースレッド数を指定する．その後，MTG0 クラスのインスタンスを invoke() メソッド（同 95 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 MTG として用意された MTG0 クラス（同 38〜45 行目）のインスタンスは，内部の compute() メソッド（同 40〜44 行目）を処理する．これにより，Mainp クラス内部の Main クラス（同 46〜90 行目）の階層開始マクロタスクが fork() される．その結果，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 42 行目）により処理を移し，join() メソッド（同 43 行目）によってすべてのタスクの実行が終了するのを待機する．

Fork された Main クラスのタスクは，まず compute() メソッド（同 53〜55 行目）を処理し，当該マクロタスク番号をもとに，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

1 つのマクロタスク実行と，そのマクロタスクに付随す

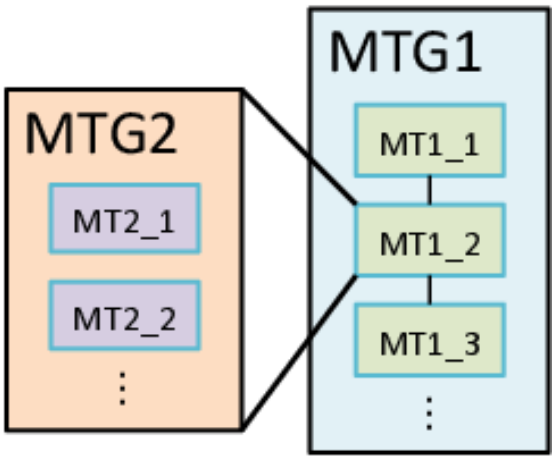

図 6　Fork/Join 型並列 Java コードに対応した MTG.

るスケジューリング管理の一連の操作を，本稿では 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 の内部には，当該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 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() メソッドを処理する．ここでは，実行すべきマクロタスク番号をもとに，該当のメソッドを呼び出す処理を行っている．こうして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本来の処理が行われる．もし，Other クラスに相当するクラスが複数存在しても，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，並列化前のプログラムで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　実行管理付きマクロタスク処理コード

本節では，Fork-Template 内に存在するマクロタスク処理コードの構成について述べる．

5.3.1　マクロタスク処理

まず，マクロタスクとして指定されたタスク本来の処理，即ち，マクロタスクとして定義した計算や分割後の for 文等の処理が行われる．

5.3.2　マクロタスク実行管理テーブルの更新

マクロタスク処理の終了後，マクロタスク実行管理テーブルを更新し，当該マクロタスクの処理が終了したことを通知する．

5.3.3　データ依存後続マクロタスクの Fork

その後，逆読み最早実行可能条件の情報（当該 MT の終了により実行可能になる可能性のある MT 候補）をもとに，当該マクロタスクの後続 MT について forkCheck() メソッドによる最早実行可能条件の判定を行い，実行可能であるかチェックする．このとき，synchronized() による排他制御を行っている．最早実行可能条件において実行可能であると判定されると，後続 MT は fork() され，実行可能でない場合は fork() されない．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．そして，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により横取りされることで，後続 MT としての実行が行われる．

5.4　ダイナミックスケジューリング

Java Fork/Join Framework を用いた粗粒度並列処理コードでの各マクロタスクに対するダイナミックスケジューリングの手順は，以下のようになる．

(1) マクロタスク処理
(2) 終了通知
(3) データ依存後続 MT の最早実行可能条件判定
(4) 実行可能ならばワーカーキューへ投入
(5) ワーカースレッドがワーカーキューから取り出す
(6) (1) に戻る

ここで（4）は，マクロタスクが後続 MT を Fork することにより行われ，当該マクロタスクを実行しているワーカースレッドのワーカーキューへ投入される．（5）は 4.4 節で述べた Java Fork/Join Framework のワークスティーリングの機能を伴って行われる．そのため，それぞれのワーカーキュー内のタスクは，自分のワーカースレッドによって取り出されるか，もしくは他のワーカースレッドがスティールすることによって取り出される．その後，当該タスクを取り出したワーカースレッド上でマクロタスクとして処理される．

Java Fork/Join Framework によるダイナミックスケジューリングでは，以上の流れを繰り返すことでマクロタスクの並列実行を可能としている．

6.　並列化コンパイラ

本章では，Java Fork/Join Framework を用いた粗粒度並列処理コードを自動生成する並列化コンパイラについて述べる．

6.1　並列化コンパイラの仕様

本研究で開発した並列化コンパイラは，並列化指示文を加えた Java プログラムを入力とすることで，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 コードを出力する．入力対象となる Java プログラムは，フロントエンドが対応している JDK1.2 の文法で記述されているものとする．

本コンパイラは Java 言語を用いて開発されており，そ

表 3　並列化指示文

表 記	意 味
/*mt fork*/	マクロタスクの定義
/*mt fork inner*/	内部にサブマクロタスクを定義する場合のマクロタスクの定義
/*premt*/	前処理マクロタスクの定義
/*postmt*/	後処理マクロタスクの定義
/*mt fork decomp=分割数*/	分割数で指定した個数にループを分割
/*mt fork decomp=分割数 private(変数名 1, 変数名 2,…)*/	分割後のループ内で使用する変数の private 化
/*mt fork decomp=分割数 reduction(リダクション演算子:変数名)*/	ループ分割によるリダクション処理
/*mt fork 論理式*/	最早実行可能条件を論理式で設定

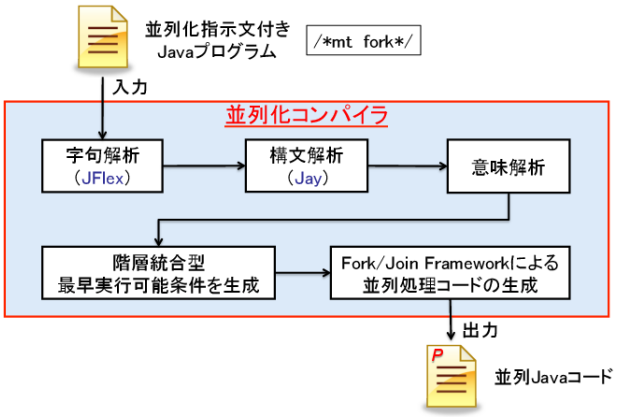

図 7　並列化コンパイラの構成.

の構成は図 7 のようになっている．字句解析と構文解析においては，LALR(1) のコンパイラコンパイラである Jay/JFlex を用いて抽象構文木を作成する．その後，並列化指示文で定義されたマクロタスクに対して，データ依存と制御依存を解析し，表 1 のような最早実行可能条件を生成する．最早実行可能条件は，本コンパイラが生成する Java Fork/Join Framework によるダイナミックスケジューリングを伴う並列 Java コードに反映される．生成された並列 Java コードは，JDK1.7 のコンパイラ javac でコンパイルすることにより，JVM 上で実行することが可能である．

6.2　並列化指示文

入力対象となる Java プログラムにおいて，階層統合型粗粒度タスク並列処理を実現する部分に表 3 の並列化指示文を記述することで，並列 Java コードを生成する．マクロタスク（MT）として定義したい部分に，以下のような並列化指示文を記述する．

```
/*mt fork*/ {
    マクロタスク処理;
}
```

マクロタスクは階層的に定義することが可能であり，for 文や while 文等の繰り返し文内部，クラスメソッド内部において，並列化指示文を入れ子状に記述する．この際，上位に位置するマクロタスクに/*mt fork inner*/と記述することで，内部並列化が可能となる．また，入力対象の Java プログラムにおいて，階層統合型粗粒度タスク並列処理を適用しない部分（前処理部分や後処理部分）については，/*premt*/および，/*postmt*/といった並列化指示文を記述する．Doall ループやリダクションループ等の並列化可能ループは，/*mt fork decomp=分割数*/のような指示文により，指定された分割数に分割する．これにより，ループを複数個に分割して，それぞれをマクロタスクとして定義することができる．また，ループ内で使用する各変数は，/*mt fork decomp=分割数 private(変数名 1, 変数名 2,…)*/と記述することにより，ループ分割後のそれぞれのマクロタスク内において，private な変数として利用することができる．さらに，リダクションループの場合，/*mt fork decomp=分割数 reduction(リダクション演算子:変数名)*/と記述することで，指定した変数に対するリダクション処理を行うことができる．マクロタスクの最早実行可能条件は自動的に求められるが，並列化指示文/*mt fork 論理式*/を記述することで，ユーザが任意に最早実行可能条件を設定することも可能である．

6.3　フロントエンドにおける抽象構文木生成

入力対象となる Java プログラムを並列化コンパイラへ入力すると，フロントエンドにおいて，抽象構文木の作成や字句解析，構文解析といった処理が行われる．抽象構文木の作成では，入力されたプログラムを木構造の形で表現し，また，マクロタスクを管理するテーブルの作成を行う．字句解析では，その文字列が並列化指示文かどうかということも判定しているため，表 3 の並列化指示文を事前に登録しておくことで，解析時に並列化指示文として認識してくれる．構文解析では，解析により並列化指示文の内部であると認識すると，内部のステートメントノード（for 文や代入文等）を構築したり，制御文等の特殊な処理が必要なものに関しては個別に情報収集を行う．その際，情報として MTG や MT，現在のクラス名，現在のメソッド名等を保存する．

6.4　ミドルパスにおける最早実行可能条件の解析

フロントエンドで収集された情報から，マクロタスク管

表 4　性能評価プログラムの概要

プログラムの種類	MolDyn	Crypt	Jacobi	台形積分
並列化指示文付き入力ソースコード長	561	308	113	35
コンパイラ生成の並列コード長（分割なし）	1337	495	466	219
コンパイラ生成の並列コード長（分割あり）	4695	1823	1231	547
MTG 数	4	1	2	1
ループ分割後の MT 数	147	21	36	12
逐次処理時間（HotSpot 最適化あり）[ms]	27992	3042	5539	1021

理や変数管理のデータを作成するとともに，変数間に存在するデータ依存の解析を行う．また，構文解析で集めた変数の定義や使用の情報を用いることで，マクロタスク間にデータ依存があるかどうかを判断する．そうして解析したデータ依存から，各マクロタスクの最早実行可能条件を生成する．それと同時に，後続 MT として実行する可能性のあるマクロタスクの情報を保持する逆読み最早実行可能条件の生成を行う．さらに，並列化可能なループ（リダクションループを含む）を分割する必要がある場合は，複数のマクロタスクに分割する．

6.5　バックエンドにおける並列 Java コードの出力

バックエンドでは，並列 Java コードの生成を行う．第 5 章で述べたマクロタスク管理テーブルクラスである Data クラスや，ユーザ定義クラスやメソッドを含む Other クラス，MTG0 クラスや main() メソッドを含む Mainp クラスとともに，Java Fork/Join Framework によるダイナミックスケジューリングを実現する際に必要となる import 文の宣言や Fork 命令の出力を行う．入力プログラムにおいて，並列化指示文によりマクロタスクとして定義された処理は，実行管理付きマクロタスク処理コードとしてそれぞれがメソッドの形で出力される．ここで，並列化可能ループについては，指定された分割数で分割してそれぞれをマクロタスクとして出力する．

こうして生成された並列 Java コードは，第 5 章で述べた Fork-Template に則ったコード構成になっており，JDK1.7 のコンパイラ javac でコンパイルした後，JVM 上で実行することが可能である．

7.　マルチコア上での粗粒度タスク並列処理の性能評価

本章では，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並列 Java コードを，開発した並列化コンパイラにより自動生成し，マルチコアプロセッサシステム DELL PowerEdge R620 上で実行して性能評価を行う．

7.1　性能評価環境

性能評価に使用する DELL PowerEdge R620 は，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．

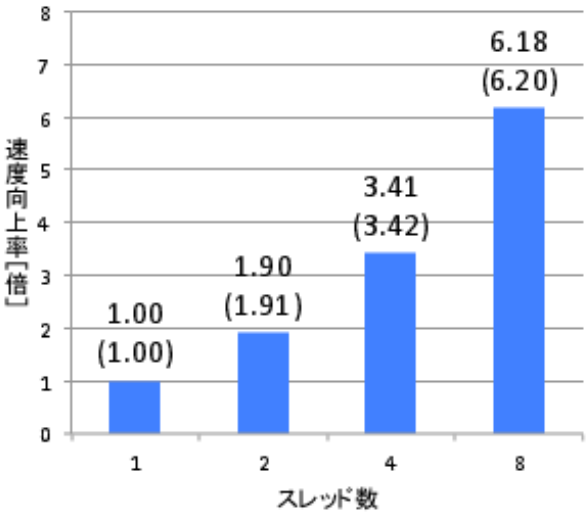

図 8　MolDyn プログラムの階層統合型粗粒度タスク並列処理（括弧内は逐次実行比）．

性能評価では，ベンチマークプログラムを含む 4 つの Java プログラムを用いる．それぞれのプログラムに表 3 の並列化指示文を加え，本研究において開発した並列化コンパイラへ入力して並列 Java コードを生成する．生成された並列 Java コードは，Java Fork/Join Framework による実装がなされている．各性能評価プログラムの概要を**表 4** に示す．自動生成された並列 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．

7.2　MolDyn プログラムによる性能評価

Java Grande Forum Benchmark Suite[17] より提供されている 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．本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．

このプログラムに対して並列化指示文を挿入し，本並列化コンパイラにより，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

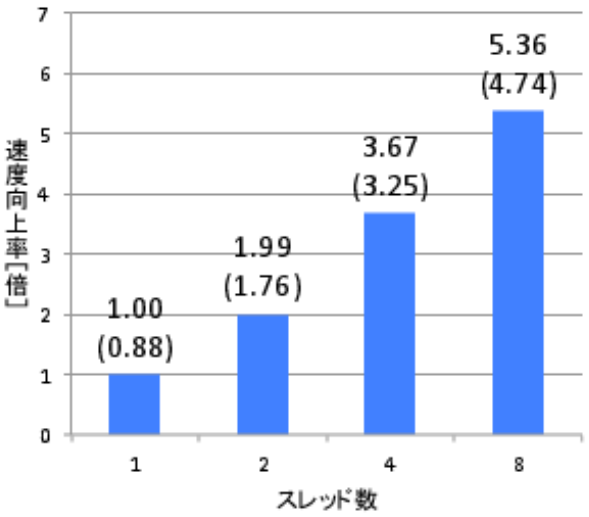

図 9　Crypt プログラムの階層統合型粗粒度タスク並列処理（括弧内は逐次実行比）.

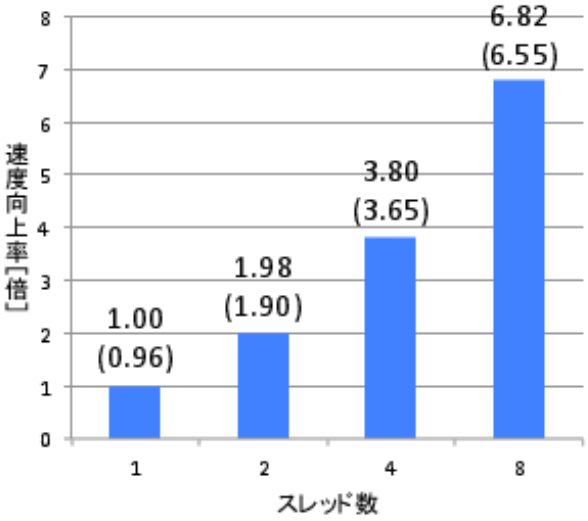

図 10　Jacobi プログラムの階層統合型粗粒度タスク並列処理（括弧内は逐次実行比）.

コードを生成する．その際，並列化可能ループについては並列化指示文を用いて 13 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，生成された並列 Java コードを javac でコンパイルし，JVM 上で実行する．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 に示すように，Intel Xeon E5-2660 上での並列処理では，並列化したプログラムを 1 スレッドで実行した場合と比べて，8 スレッドで 6.18 倍の速度向上が得られた．一方，逐次実行（並列化コンパイラによる並列化を行う前のプログラムの実行）と本手法による 1 スレッドでの実行を比較すると，本手法では Fork-Template を用いたデータ構造が導入されており，1 スレッド実行において逐次実行よりも処理時間が僅かに短縮されている．8 スレッドの場合では，逐次実行と比べて 6.20 倍の速度向上が得られている．

7.3　Crypt プログラムによる性能評価

同じく，Java Grande Forum Benchmark Suite より提供されている 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．本性能評価ではクラス C のプログラムを使用し，配列の大きさを N=5000 万としている．

このプログラムに対して並列化指示文を挿入し，本並列化コンパイラにより並列 Java コードを生成する．その際，並列化可能ループについては並列化指示文を用いて 8 分割している．その後，生成された並列 Java コードをコンパイルし，JVM 上で実行する．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 9** に示す．図 9 に示すように，並列化したプログラムを 1 スレッドで実行した場合と比べて，8 スレッドで 5.36 倍の速度向上が得られた．一方，逐次実行と比べると 4.74 倍の速度向上が得られている．

7.4　Jacobi プログラムによる性能評価

Jacobi プログラムは，ヤコビ法による連立一次方程式の求解プログラムである．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並列化指示文を挿入し，本並列化コンパイラにより並列 Java コードを生成する．その際，並列化可能ループについては 8 分割している．その後，生成された並列 Java コードをコンパイルし，JVM 上で実行する．JVM 実行時には HotSpot 最適化を適用している．

Jacobi プログラムの実行結果を**図 10** に示す．図 10 に示すように，並列化したプログラムを 1 スレッドで実行した場合と比べて，8 スレッドで 6.82 倍の速度向上が得られた．一方，逐次実行と比べると 6.55 倍の速度向上が得られている．

7.5　台形積分プログラムによる性能評価

台形積分プログラムは円周率を求めるプログラムで，$4/(1+x^2)$ を x=0〜1 の範囲で定積分するものである．積分は台形公式にて行っており，積分の分割数は N=5000 万としている．

このプログラムに対して並列化指示文を挿入し，本並列化コンパイラにより並列 Java コードを生成する．その際，並列化可能ループについては 8 分割している．その後，生成された並列 Java コードをコンパイルし，JVM 上で実行する．JVM 実行時には HotSpot 最適化を適用している．

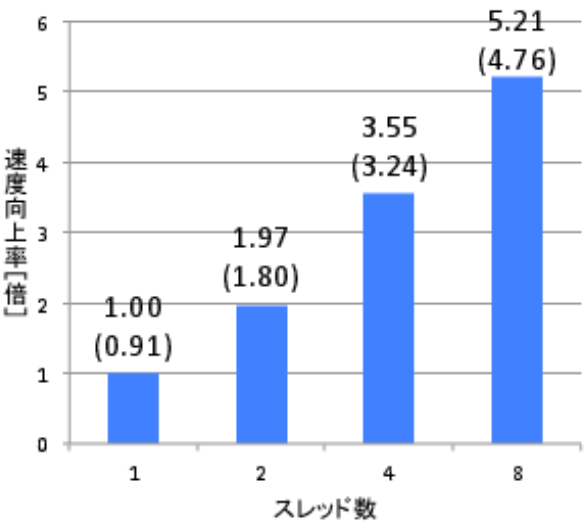

図 11 台形積分プログラムの階層統合型粗粒度タスク並列処理（括弧内は逐次実行比）.

台形積分プログラムの実行結果を**図 11** に示す．図 11 に示すように，並列化したプログラムを 1 スレッドで実行した場合と比べて，8 スレッドで 5.21 倍の速度向上が得られた．一方，逐次実行と比べると 4.76 倍の速度向上が得られている．

以上の結果より，本並列化コンパイラにより自動生成された並列 Java コードの有効性が確認された．

8. おわりに

本稿では，Java Fork/Join Framework を用いた粗粒度並列処理コードの自動生成手法を提案し，その並列化コンパイラを開発した．本並列化コンパイラは，並列化指示文を加えた Java プログラムを入力として，Fork/Join Framework を用いた並列 Java コードを生成することが可能である．生成された並列 Java コードでは Fork-Template 形式のデータ構造が導入されており，Java Fork/Join Framework による階層統合型粗粒度タスク並列処理が実現される．

性能評価では，並列化指示文を加えたベンチマークプログラムを並列化コンパイラへ入力し，並列 Java コードの自動生成を行った．生成された並列 Java コードを，マルチコアプロセッサ DELL PowerEdge R620 上で実行したところ，MolDyn プログラムの場合は 8 スレッドで 6.18 倍，Crypt プログラムでは 5.36 倍，Jacobi プログラムでは 6.82 倍，台形積分プログラムでは 5.21 倍の速度向上を実現し，高い実効性能を達成した．

以上の結果から，本研究で開発した並列化コンパイラの有用性が確認され，本並列化コンパイラにより生成された Fork/Join Framework を用いた粗粒度並列処理コードの有効性が確認された．

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， Vol. 2009-ARC-184, No. 15, pp. 1–10 (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] A. Yoshida, Y. Ochi and 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， Vol. 7, No. 4, pp. 56–66 (2014).
[9] Oracle: “The Java™ Tutorials > Essential Classes > Concurrency > Fork/Join”, [http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html].
[10] 神山 彰，吉田明正：“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”，情報処理学会研究報告，Vol. 2014-ARC-211, No. 8, pp. 1–8 (2014).
[11] 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).
[12] 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).
[13] B. Chan and T.S. Abdelrahman: “Run-time support for the automatic parallelization of Java programs”, *J. Supercomputing*, Vol. 28, pp. 91–117 (2004).
[14] M.K. Chen and K. Olukotun: “The Jrpm system for dynamically parallelizing Java programs”, *Proc. ISCA-30*, pp. 434–446 (2003).
[15] 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).
[16] Oracle: “Java™ Platform, Standard Edition 7 API Specification”, [http://docs.oracle.com/javase/7/docs/api/].
[17] 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).