Vol.2014-ARC-208 No.24
2014/1/24

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

遠藤 佑太[1]　山内 長承[1]　吉田 明正[2]

概要：マルチコアプロセッサ上での Java プログラムを対象とした粗粒度タスク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，階層ごとにタスク間並列性を抽出した後，ダイナミックスケジューラが全階層のタスクをコアに割り当て，階層を越えたタスク間並列性を利用することが可能である．本稿では，再帰メソッドを伴う Java プログラムに対して，再帰レベルを考慮した階層統合型粗粒度タスク並列処理を実現する並列 Java コードの生成手法を提案する．本手法を実装した並列化コンパイラを用いて並列 Java コードを生成し，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，再帰モデルプログラムとマージソートの場合に高い実効性能が達成されることが確認された．

キーワード：並列化コンパイラ，階層統合型実行制御，粗粒度タスク並列処理，マルチコア，再帰プログラム

1. はじめに

マルチコアプロセッサは，スーパーコンピュータ，PC，組み込みシステムに至るまで広く用いられている．この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ためには，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．

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並列化コンパイラが抽出して階層型マクロタスクグラフを生成し，各階層の粗粒度タスクを，グルーピングしたコアに階層的に割り当て並列処理を行っていた．この場合，対象プログラム中の各階層の粗粒度タスクは，その階層を処理すべきコアグループに割り当てられて実行されるため，十分な台数のコアを確保できない場合には，対象プログラムに内在する全階層の粗粒度タスク間並列性を利用できない可能性がある．

そこで，粗粒度タスク並列処理で用いられている階層型マクロタスクグラフ [3] を利用しつつ，対象プログラム中の異なる階層の粗粒度タスクを統一的に取り扱い，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階層統合型実行制御手法 [7], [8] が提案されている．

[1] 東邦大学理学部情報科学科
[2] 明治大学総合数理学部ネットワークデザイン学科

並列処理の対象言語は Fortran 言語や C 言語が主流であったが，最近では PC や組み込みシステムのソフトウェア開発において Java 言語が広く用いられてきており，Java プログラムによる並列処理の期待が高まっている．Java 言語の再帰プログラムの並列処理に関する研究は，トレース間並列処理 [9], [10] が提案されているが，ハードウェアトランザクションメモリ環境を前提としている．また，C 言語の再帰プログラムの Cell BE 用の並列コード生成に関しては，分割統治プログラムを対象としたものが提案されている [11]．

本稿では，再帰メソッドを伴う Java プログラムに対して，再帰レベルを考慮した階層統合型粗粒度タスク並列処理を実現する並列 Java コードの生成手法を提案し，本研究室で開発した並列化コンパイラに提案手法を実装した．本コンパイラにより生成された並列 Java コードは，再帰メソッドにまたがる階層間並列性を利用し，マルチコアプロセッサ上で高い実効性能を達成することが確認されている．

本稿の構成は以下の通りとする．第 2 章では階層統合型粗粒度タスク並列処理の概要を述べる．第 3 章では，再帰レベルを考慮し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 コードについて述べる．第 4 章では，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 5 章では，並列化コンパイラにより生成した並列 Java コードを用いて性能評価を行う．第 6 章でまと

めを述べる．

2. 階層統合型粗粒度タスク並列処理

階層統合型粗粒度タスク並列処理 [7] では，粗粒度タスク並列処理手法 [3] で用いられている並列性抽出技術を用いて，階層型マクロタスクグラフ（MTG）を生成し，その階層型マクロタスクグラフに対して階層開始マクロタスク [7] を導入する．その後，全階層のマクロタスクを統一的に取り扱い，最早実行可能条件を満たした粗粒度タスク（マクロタスク）から順に，コアに割り当てるダイナミックスケジューリングコードを生成する．

例えば，**図 1** の Java プログラムは，階層統合型粗粒度タスク並列処理を適用する場合，**図 2** の階層型マクロタスクグラフ（制御用のダミーマクロタスクは図示していない）として表現される．このプログラムを 4 コア上で実行したイメージは **図 3** のようになり，全階層のマクロタスク間並列性（例，MT2-1，MT2-2，MT3-3，MT3-4 間の並列性）が最大限に利用される．

2.1 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，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 の Java プログラムを階層的にマクロタスクに分割する場合，Main クラスの main() メソッドにおいて，第 1 階層マクロタスク（MT1-1，MT1-2，MT1-3）が定義される．次に，MT1-2 の繰り返し文（for 文）の内部において，第 2 階層マクロタスク（MT2-1，MT2-2）が定義される．さらに，MT1-1 のメソッド呼び出し Other.func() の内部において，第 3 階層マクロタスク（MT3-1，MT3-2, MT3-3, MT3-4）が定義される．

2.2 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

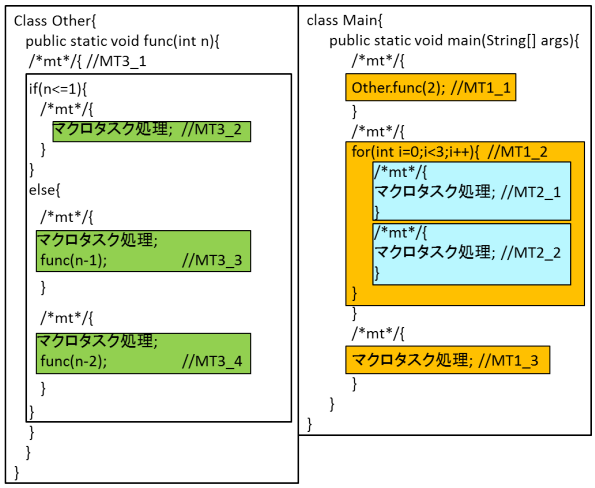

図 1 再帰メソッドを含む並列化指示文付き Java プログラム

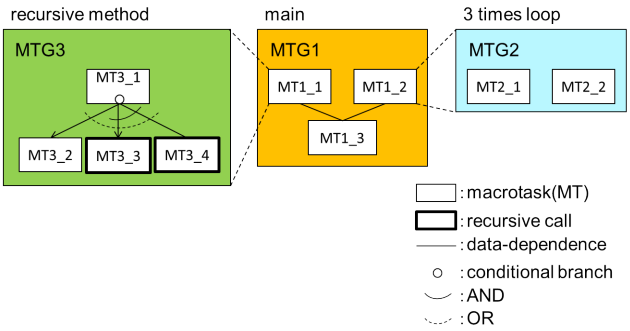

図 2 階層型マクロタスクグラフ（MTG）

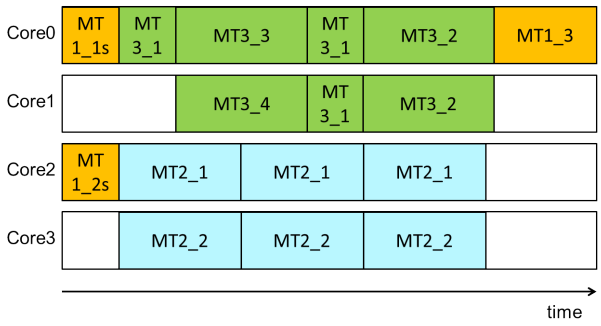

図 3 階層統合型粗粒度タスク並列処理の実行イメージ

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．

例えば，図 2 の繰り返し文の MT1-2 の場合，内部に第 2 階層マクロタスク（MT2-1，MT2-2）を含んでおり，MT1-2 は第 2 階層用の階層開始マクロタスクとして扱われる．同様に，メソッド呼び出しの MT1-1 の場合，内部に第 3 階層マクロタスク（MT3-1，MT3-2, MT3-3, MT3-4）を含んでおり，MT1-1 は第 3 階層用の階層開始マクロタスクとして扱われる．また，MT3-3，MT3-4 はメソッド func() を再帰呼び出ししており，これらも第 3 階層用の階層開始マクロタスクとして扱われる．

2.3 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

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	MT1-1 ††	true	1-1S
1	MT1-2 †	true	1-2S
1	MT1-3	1-1∧1-2	1-3
1	MT1-4(EndMT)	1-3	1-4
2	MT2-1	1-2S	2-1
2	MT2-2	1-2S	2-2
2	MT2-3(CtrlMT)	2-1∧ 2-2	2-3
2	MT2-4(RepMT)	$2\text{-}3_{2\text{-}4}$	2-4
2	MT2-5(ExitMT)	$2\text{-}3_{2\text{-}5}$	1-2
3	MT3-0	1-1S∨3-3S∨3-4S	3-0
3	MT3-1	3-0	3-1
3	MT3-2	3-1	3-2
3	MT3-3	3-1	3-3S
3	MT3-4	3-1	3-4S
3	MT3-5(CtrlMT)	3-2∨(3-3∧3-4)	3-5
3	MT3-6(RepMT)	$3\text{-}5_{3\text{-}6}$	3-6
3	MT3-7(ExitMT)	$3\text{-}5_{3\text{-}7}$	3-7, 上位 MT

(注) † 繰り返し文内部の第 2 階層 MTG2 の階層開始 MT.
†† メソッド内部の第 3 階層 MTG3 の階層開始 MT.

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最大限に表しており，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．

図 2 の各マクロタスクの最早実行可能条件と終了通知は **表 1** の通りとなる．最早実行可能条件において，i は MT_i の終了，$(i)_j$ は MT_i から MT_j への分岐，i_j は MT_i から MT_j への分岐と MT_i の終了を表している．また，EndMT（終了処理），CtrlMT（当該階層の繰り返し判定処理），RepMT（当該階層の繰り返し更新処理），ExitMT（当該階層の終了処理）は制御に用いられるダミーマクロタスクである．

次に，階層開始マクロタスクの導入により，従来の階層ごとに求めた最早実行可能条件を階層統合型実行制御用に変換する [7].

2.4 階層統合型実行制御によるスケジューリング

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 2.3 節の最早実行可能条件が満たされた後，レディマクロタスクキューに投入され，プライオリティの高いマクロタスクから順にレディマクロタスクキューから取り出されてコアに割り当てられる．

階層統合型実行制御では，全ての階層のマクロタスクが統一的に取り扱われ，それぞれのコア（グルーピングなし）に割り当てられ実行される．

3. 再帰プログラムの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を実現するための並列 Java コードおよびデータ管理について述べる．なお，並列 Java コードは後述の並列化コンパイラにより自動生成される．

3.1 再帰呼び出しを考慮した並列 Java コードの構成

図 1 のプログラムは，並列化指示文を加えた Java プログラムであり， 図 2 の階層型マクロタスクグラフに対応している．このプログラムを，本研究室で開発した並列化コンパイラに入力すると， **図 4** の並列 Java コードが生成される．並列 Java コードは，後述の変数管理クラス（VARmanage），ダイナミックスケジューリングのためのマクロタスク管理テーブルクラス（MTtable）とマクロタスク管理クラス（MTmanage），ユーザ定義クラスとメソッドのための Other クラス，並列 Java コードの main() メソッドを含む Mainp クラスから構成される．

Mainp クラスにおいて，内部クラスの Scheduler クラスが定義されており，scheduler() メソッドが呼び出される．eeccheck() メソッドでは，引数で与えられたマクロタスクが最早実行可能条件を満たしているかを判定している．scheduler() メソッドでは，レディマクロタスクキューからマクロタスクを取り出して実行し，新たなレディマクロタスクをレディマクロタスクキューに投入する手順を，EndMT が終了するまで繰り返す．

各マクロタスクのコードは，Mainp クラスのクラスメソッドとして実装される．なお，ユーザ定義クラスのメソッド内のマクロタスクのコードに関しては，ユーザ定義クラス（Other クラス）の中に，クラスメソッドとして実装される．ユーザ定義クラスは複数あっても構わない．

本手法では，再帰呼び出しに対応するために，動的生成識別子 `dynamicid` を使用する．階層開始マクロタスクにおいて変数管理クラスとマクロタスク管理テーブルクラスのインスタンスを生成する際，同時にその階層のマクロタスク管理テーブルクラスに対応した `dynamicid` を生成する．各々の管理クラスへのアクセスはこの `dynamicid` を用いて行われ，再帰メソッド内のマクロタスクは適切に実行される．

3.2 ダイナミックスケジューラの Java 実装

本手法では，階層統合型実行制御を伴うダイナミックスケジューラの実装方式として，コアを有効利用するため，

```
class MTtable {        //マクロタスク管理テーブルクラス
  ステート管理テーブル（MT終了分岐）宣言;
  レディ管理テーブル（MTレディ）宣言;
}

class MTmanage {              //マクロタスク管理クラス
  ArrayList<MTtable> dynamicfield
                          = new ArrayList<MTtable>();
}

class VARmanage1 {            //変数管理クラス //MTG1
  引数用変数の宣言;
  戻り値用変数の宣言;
  MT間共有変数の宣言;
}
class VARmanage2 {            //変数管理クラス //MTG2
}
class VARmanage3 {            //変数管理クラス //MTG3
}

class Other {               //ユーザ定義クラスとメソッド
  static void mt3_0() {
   if(設定した再帰レベルに達していない){
   動的生成識別子dynamicid
                  = Mainp.mtm[2].dynamicfield.size()-1;
   varm3とmtm[2].dynamicfieldのdynamicid番目に追加...}}
                                        //階層開始MT3-0
  static void mt3_1(){ ... }                    //MT3-1
  static void mt3_2(){mt3_0();}     //再帰呼び出しMT3-2
  ...

}

class Mainp {             //並列JavaコードのMainpクラス
  static MTmanage  mtm
           = new MTmanage[MTMMAX];            //MT管理
  static ArrayList<VARmanage1> varm1
           = new ArrayList<VARmanage1>();   //変数管理
  static ArrayList<VARmanage2> varm2
           = new ArrayList<VARmanage2>();   //変数管理
  static ArrayList<VARmanage3> varm3
           = new ArrayList<VARmanage3>();   //変数管理
  レディMTキュー宣言;

  static class Scheduler implements Runnable {
    int threadid;
    Scheduler(int thrid) { threadid = thrid;    }
    public void run()    { scheduler(threadid); }
  }

  static boolean eeccheck(int mt) {
    mt番MTの最早実行可能条件を満たすか判定;
  }
  static void scheduler(int threadid) {
    レディMTキューにレディMTを投入;
    while (EndMTが未終了) {
      レディMTキューからMTiを取り出す;
      MTiを実行し、MTiの終了を登録;
      新たなレディMTをレディMTキューに投入;
    }
  }
  static void mt1_1() { Other.mt3_0(); }        //MT1-1
  static void mt1_2() { ... }             //階層開始MT1-2
  static void mt1_3() { ... }                   //MT1-3
  static void mt2_1() { ... }                   //MT2-1
  static void mt2_2() { ... }                   //MT2-2
  ...

  public static void main(String[] args) {
   varm1,varm2,mtm[0],mtm[1]にMainp内MT用instance追加;
   varm3,mtm[2].dynamicfieldにinstance追加;
   PE(コア)数のスレッドをScheduler()で生成;
   スレッド合流;
  }
}
```

図 4 コンパイラ生成による並列 Java コード

分散型ダイナミックスケジューリング方式を採用している．この場合，各コアでは，スケジューリング処理部とマクロタスク処理部から構成されるスレッドコードをそれぞれ実行する．

本手法では，Java 言語の Runnable インタフェースを用いてマルチスレッドコードを実現する．また，ダイナミックスケジューリング時に，レディマクロタスクキューが空の場合には，ダイナミックスケジューラは，`wait()` によりウェイトセットに入る．一方，他コアで，マクロタスク終了に伴い新たに実行可能なマクロタスクがレディキューに投入された場合には，`notifyAll()` により，ウェイトセットに入っているスレッドは実行状態に戻る．

各スレッドコードでは，コア上でマクロタスクの処理を終える度に，スケジューリング処理部でスケジューリングを行い，自コアに新たに割り当てられたマクロタスクの処理を行う．なお，レディマクロタスクキューのアクセスに対しては，Java 言語の `synchronized()` により排他制御を行っている．

3.3 変数管理クラス（VARmanage）

階層統合型粗粒度タスク並列処理では，異なる階層のマクロタスクを統一的に扱うため，メソッド内部のマクロタスクとメソッド呼び出し側のマクロタスクをスケジューラが一元管理する．メソッドを複数の箇所で同時に呼び出す可能性を考慮し，本手法では，メソッド内部の変数を管理する VARmanage クラスのインスタンスを，対応する階層のメソッド呼び出しの階層開始マクロタスクで動的に生成する．インスタンスへのアクセスは，3.1 節の `dynamicid` を用いて適切に行われる．

VARmanage クラスは階層の数だけ生成され，それぞれの階層で使用される変数が内部で定義されている． 図 4 の場合，階層の数は 3 つであるため，VARmanage クラスは VARmanage1〜VARmanage3 まで生成される．VARmanage クラスでは，呼び出すメソッドの引数用変数，戻り値用変数，マクロタスク間共有変数を管理する．VARmanage のインスタンスは呼び出すメソッドごとに用意するため，並列化対象メソッドが複数あれば VARmanage のインスタンスも複数となる．並列化対象メソッド内部のマクロタスクで使われる変数は，VARmanage のインスタンスの中にある変数に対してアクセスすることになる．

VARmanage のインスタンスは，呼び出す階層（メソッド本体）に対応したもののみが生成される．例えば，呼び出すメソッドが第 3 階層にあたる場合は，VARmanage3 のインスタンスが生成される．

3.4 マクロタスク管理クラス（MTmanage）

並列化対象のメソッドを呼び出すと，その階層開始マクロタスクが実行され，VARmanage クラスと同様にマクロタスクを管理する MTtable クラスのインスタンスが生成される．生成された MTtable のインスタンスを用いて，メソッド内部のマクロタスクのステート管理等を行う．

階層開始マクロタスクによって動的に生成された MTtable のインスタンスを `dynamicid` でアクセスすることにより，スケジューラは適切にマクロタスクの管理を行うことができる．最早実行可能条件においても `dynamicid` およびマクロタスク番号によって判別することができる．

MTtable のインスタンスの管理は，マクロタスク管理

クラス（MTmanage）が行う．Mainp クラスにおいて，階層の数を要素数とした MTmanage クラスの配列を用意する．これにより，MTtable のインスタンスの管理は各階層（メソッド本体）毎によって行われ，メソッドの呼び出し元となる第 1 階層の無駄な管理テーブルクラスの生成を避けることが可能となる．MTtable のインスタンスへのアクセスも，VARmanage のインスタンスと同様に 3.1 節の dynamicid を用いて適切に行われる．

3.5 再帰レベルを考慮した再帰メソッド内のマクロタスク管理

本手法では，再帰メソッド内部の各マクロタスクに対して階層統合型粗粒度タスク並列処理を適用するために，変数管理クラス（VARmanage）とマクロタスク管理テーブルクラス（MTtable）のインスタンスを動的に生成し，再帰メソッド内部のマクロタスクの管理を行う．

呼び出し側のマクロタスクは再帰メソッドを呼び出した時，即ち，その階層開始マクロタスクが実行された時に，VARmanage および MTtable のインスタンスを生成する．呼び出される側のマクロタスク ExitMT の終了通知が発行されると，その再帰メソッドのマクロタスクは全て終了したことになり，上位のメソッド呼び出しのマクロタスクが終了したことになる．ここで，呼び出される側のマクロタスクにおいて戻り値があるならば，上位のメソッド呼び出しのマクロタスクはその値を受け取る．

再帰呼び出しを行う際，最初に再帰呼び出しを行うマクロタスクの再帰レベルを 0 とする．呼び出されたマクロタスクの再帰レベルは 1 となり，以降のマクロタスクの再帰レベルは呼び出し側の再帰レベルに 1 を足したものになる．

本手法では，再帰メソッドを呼び出す際，同時に再帰レベルを表す引数を渡す．再帰呼び出し時に一定のレベルに達している場合は，管理クラスのインスタンスの生成を行わず，マクロタスク化していない元の再帰メソッドを呼び出す．これにより，並列性を十分満たした場合には，その再帰レベル以上の再帰呼び出しは 1 コア上で実行することにより，ダイナミックスケジューリングのオーバーヘッドを軽減している．このレベルは，4.1 節の並列化指示文によりユーザが指定可能である．

このように，再帰メソッド内部のマクロタスクは，階層型マクロタスクグラフ（MTG）の階層レベルに加えて再帰レベルも考慮して扱われる．これにより再帰メソッド内部のマクロタスクに対しても，階層統合型粗粒度タスク並列処理を適用することが可能となる．

4. 並列化コンパイラ

本章では，階層統合型粗粒度タスク並列処理の並列化コンパイラについて述べる．

4.1 並列化指示文

本手法では，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並列化指示文を記述し，並列化コンパイラにより並列 Java コードを生成する．

粗粒度タスク（マクロタスク）として定義したい部分に，以下のような並列化指示文を加える．

```
/*mt*/ {
  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やクラスメソッド内部においても，並列化指示文（/*mt*/）を入れ子にすることにより記述できる．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の適用しない部分（前処理部分や後処理部分）は，/*premt*/，/*postmt*/の並列化指示文を記述する．

図 1 のプログラム（ 図 2 の MTG に対応）は，本コンパイラの並列化指示文を加えたソースプログラムである．本コンパイラでは，並列化可能ループは並列化指示文で分割数を指定（/*mt decomp=値*/）することにより，複数のループに分割して，それぞれをマクロタスクとして定義する．各マクロタスクの CP 長は，/*mt cp=値*/により記述できる．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 論理式*/を用いて，最早実行可能条件の論理式を直接記述することも可能である．3.5 節の再帰レベルは，並列化指示文/*mt relv=値*/を再帰メソッド呼び出しの部分に記述することで設定できる．

4.2 並列化コンパイラの仕様

本並列化コンパイラでは，並列化指示文を加えた Java プログラムを入力とし，階層統合型粗粒度タスク並列処理の並列 Java プログラムを出力する．

本コンパイラの対象となる入力 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．

4.3 並列化コンパイラの実装

本節では，並列 Java コードを生成する並列化コンパイラについて述べる．本コンパイラは Java 言語により開発されており，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，並列化指示文により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．この最早実行可能条件は本コンパイラの生成するダイナミックスケジューラに反映され，ダイナミックスケジューラを含む並列 Java コード

(Mainp.java) が生成される．

現インプリメントでは，メソッド内におけるマクロタスクの最早実行可能条件の自動生成は可能であるが，高度なインタープロシージャ解析は実装されていないため，メソッド間の高い並列性を引き出すためには，並列化指示文にて最早実行可能条件を直接記述する方法もある．

5. マルチコアプロセッサ上での性能評価

再帰プログラムの階層統合型粗粒度タスク並列処理の性能評価を，DELL PowerEdge R620 サーバ上で行う．

5.1 性能評価環境

DELL PowerEdge R620 は，Intel Xeon E5-2660（8 コア，2.20GHz）を 2 ソケット（16 コア），64GB のメモリから構成されており，OS は CentOS6.4，Java 処理系は JDK1.7 となっている．性能評価を行うプログラムは，5.2 節で述べる 4 種類の再帰モデルとマージソートプログラムである．並列化指示文を加えたこれらのプログラムを並列化コンパイラに入力し，並列 Java コードを生成する．この並列 Java コードを JDK1.7 の javac でコンパイルし，JVM で実行して性能評価を行う．なお，マージソートプログラムの性能評価では，OpenMP の task 構文による実装との比較評価も行う．

5.2 再帰モデルプログラムによるループ並列処理と階層統合型粗粒度タスク並列処理の比較

本節では，4 種類の再帰モデル (R 再帰 P 並列) を用意し，それぞれにおいてのループ並列処理と階層統合型粗粒度タスク並列処理との性能比較を行う．

用意する再帰モデルを **図 5** に示す．図中の A，$C_j(1 \leqq j \leqq P)$，D は一定の処理時間を持つマクロタスクであり，かかる負荷はすべて同一とする．$B_i(1 \leqq i \leqq R)$ は再帰呼び出しを行うマクロタスクであり，再帰メソッド func() を呼び出している． 図 5 の (a) のモデルでは，B1 と C1 のマクロタスクが並列に実行される形となっており，これを 1 再帰 1 並列モデルと呼ぶ．(b) のモデルは，B1，C1，C2 のマクロタスクが並列に実行される形であり，1 再帰 2 並列と呼ぶ．(c) のモデルは，B1，B2，C1 のマクロタスクが並列に実行される形であり，2 再帰 1 並列と呼ぶ．(d) のモデルは，B1，B2，C1，C2 のマクロタスクが並列に実行される形であり，2 再帰 2 並列モデルと呼ぶ．これらの計 4 種類の再帰モデルで性能評価を行う．

4 種類の再帰モデル (R 再帰 P 並列) での実行において，利用する並列性の異なる 3 通りの実行条件を設定する．

(1) ループ並列性のみ利用：
A，D を 16 分割し，$B_i(1 \leqq i \leqq R)$ と $C_j(1 \leqq j \leqq P)$ を逐次的に実行する．

(2) 粗粒度並列性のみ利用：
A，D を分割せず，$B_i(1 \leqq i \leqq R)$ と $C_j(1 \leqq j \leqq P)$ を並列に実行する．

(3) ループ並列性と粗粒度並列性の同時利用：
A，D を 16 分割し，$B_i(1 \leqq i \leqq R)$ と $C_j(1 \leqq j \leqq P)$ を並列に実行する．

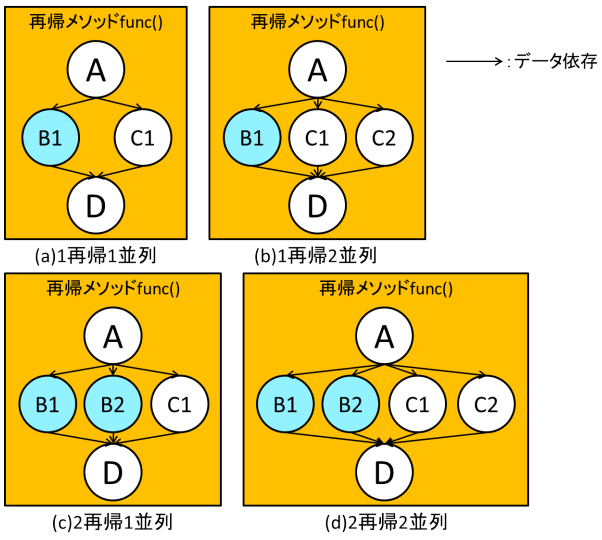

図 5 4 種類の再帰モデル

メソッド呼び出し回数を 4 回，再帰レベルを 3 に設定し，それぞれの再帰モデルを上記の条件で実行した結果を **表 2** に示す．また，16 スレッドで実行した結果をまとめたグラフを **図 6** に示す．本節で実行するプログラムは，すべて-Xint オプションを付け，JVM の HotSpot を適用せずに実行している．

1 再帰 1 並列モデルの並列 Java コードを Intel Xeon E5-2660 上で 16 スレッドで実行した結果，(1) のループ並列性のみの利用では 2.35 倍（逐次実行比）の速度向上，(2) の粗粒度並列性のみの利用では 1.29 倍（逐次実行比）の速度向上であるが，(3) のループ並列性と粗粒度並列性の同時利用では 5.89 倍（逐次実行比）の速度向上が得られた．

1 再帰 2 並列モデルの並列 Java コードを Intel Xeon E5-2660 上で 16 スレッドで実行した結果，(1) のループ並列性のみの利用では 1.77 倍の速度向上，(2) の粗粒度並列性のみの利用では 1.71 倍の速度向上であるが，(3) のループ並列性と粗粒度並列性の同時利用では 8.33 倍の速度向上が得られた．

2 再帰 1 並列モデルの並列 Java コードを Intel Xeon E5-2660 上で 16 スレッドで実行した結果，(1) のループ並列性のみの利用では 2.19 倍の速度向上，(2) の粗粒度並列性のみの利用では 4.64 倍の速度向上であるが，(3) のループ並列性と粗粒度並列性の同時利用では 13.57 倍の速度向上が得られた．

2 再帰 2 並列モデルの並列 Java コードを Intel Xeon E5-2660 上で 16 スレッドで実行した結果，(1) のループ並列性のみの利用では 1.71 倍の速度向上，(2) の粗粒度並列性

表 2 再帰モデルの階層統合型粗粒度タスク並列処理による速度向上率 (逐次実行比)

再帰モデル	並列性利用 ‡	スレッド数 1	2	4	8	16	理論値 †
1 再帰 1 並列	(1)	0.99	1.39	1.85	2.08	2.35	2.67
	(2)	0.99	1.29	1.29	1.29	1.29	1.33
	(3)	0.99	1.72	2.96	4.67	5.89	8.00
1 再帰 2 並列	(1)	0.99	1.26	1.53	1.64	1.77	1.88
	(2)	0.99	1.29	1.72	1.72	1.71	1.77
	(3)	0.99	1.88	3.43	5.79	8.33	10.67
2 再帰 1 並列	(1)	0.99	1.41	1.84	1.99	2.19	2.67
	(2)	0.99	1.78	2.92	4.29	4.64	5.00
	(3)	0.99	1.98	3.89	7.65	13.57	15.65
2 再帰 2 並列	(1)	0.99	1.28	1.53	1.61	1.71	1.88
	(2)	0.99	1.85	3.25	4.72	6.38	6.67
	(3)	0.99	1.98	3.95	7.75	13.76	15.48

(注) † 16 スレッドでの理論値.
‡ 並列性利用 (1)：ループ並列性のみ.
並列性利用 (2)：粗粒度並列性のみ.
並列性利用 (3)：ループ並列性と粗粒度並列性.

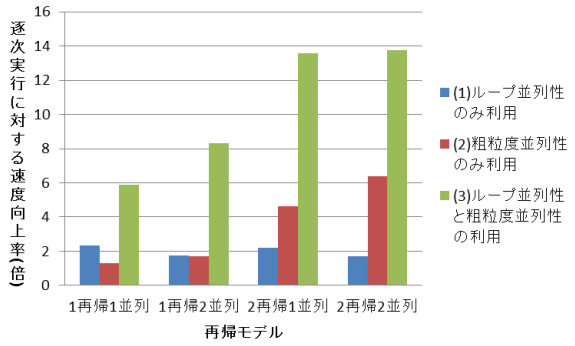

図 6 再帰モデルの 16 スレッドによる階層統合型粗粒度タスク並列処理

のみの利用では 6.38 倍の速度向上であるが，(3) のループ並列性と粗粒度並列性の同時利用では 13.76 倍の速度向上が得られた．

これらの結果から，再帰モデルプログラムにおいて，ループ並列性と粗粒度並列性の両方を利用できる階層統合型粗粒度タスク並列処理は，理論値に匹敵する速度向上率を示しており，高い実効性能を達成できることが確認された．

5.3 マージソートの階層統合型粗粒度タスク並列処理

本節では，マージソートプログラムの並列 Java コードの性能評価について述べる．マージソートのデータ数は 5000 万個とし，実行する再帰メソッドのマクロタスクの再帰レベルを 2，3，4 と変えて実行した．また，マージソートのデータ列の分割及び併合を行うマクロタスクはループ文となっており，これを 16 個に分割した場合の性能評価も行った．

図 7 は，再帰レベル 3 で，-Xint オプションを付けて

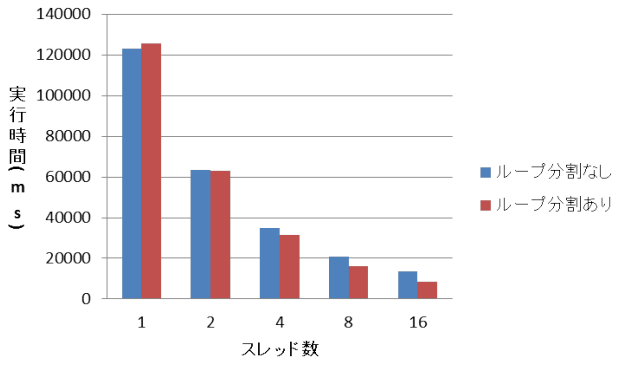

図 7 マージソートの階層統合型粗粒度タスク並列処理 (再帰レベル 3，-Xint オプションあり)

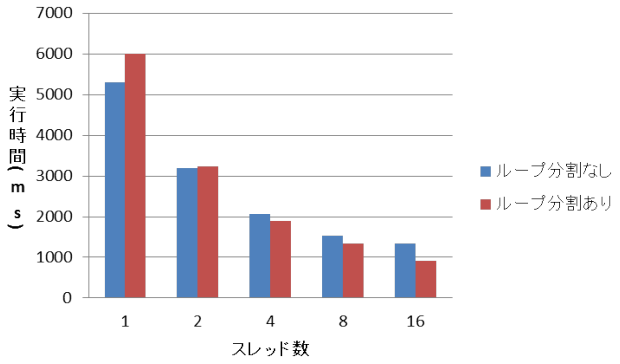

図 8 マージソートの階層統合型粗粒度タスク並列処理 (再帰レベル 3，-Xint オプションなし)

Intel Xeon E5-2660 上で実行した結果である．逐次実行では 122731[ms] の実行時間であるのに対し，ループ分割なしでは 16 スレッドで 13459[ms]，分割ありでは 16 スレッドで 8302[ms] の実行時間となった．

次に，**図 8** は，再帰レベル 3 で，-Xint オプションを付けずに (JVM の HotSpot を適用して) 実行した結果である．逐次実行では 5263[ms] の実行時間であるのに対し，ループ分割なしでは 16 スレッドで 1333[ms]，分割ありでは 16 スレッドで 924[ms] の実行時間となった．

再帰レベルはマクロタスクを生成する階層を，再帰メソッドの呼び出しの深さに応じて限定することを意味しており，再帰レベルより深い呼び出し部分は 1 コア実行となる．16 スレッドの-Xint オプションなしの実行の場合，再帰レベル 2 では 1423[ms]，再帰レベル 4 では 1309[ms] となっており，再帰レベル 3 の 924[ms] が本環境の最適なレベルとなる．

比較評価として，並列プログラミング用 API である OpenMP の task 構文を用いた C 言語によるマージソートの実行結果を **図 9** に示す．最適化オプション-O0 で実行した場合，逐次実行では 14202[ms] の実行時間であるのに対し，16 スレッドでは 6718[ms] の実行時間となった．最適化オプション-fast で実行した場合，逐次実行では 4977[ms]

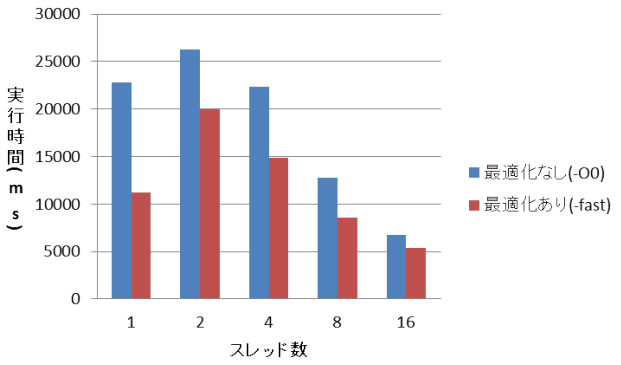

図 9 マージソートの OpenMP の task 構文による実装

の実行時間であるのに対し，16 スレッドでは 5360[ms] の実行時間となった．OpenMP のタスクの起動や同期には大きなオーバーヘッドがかかるため，マージソートにおける並列処理の台数効果は，OpenMP では期待できないことがわかる． 図 8 と 図 9 の 16 スレッドを比較すると，本手法は 924[ms]，OpenMP は 5360[ms] であり，本手法は OpenMP の task 構文に対して 5.80 倍の速度向上を実現している．

この結果から，本並列化コンパイラの生成した並列 Java コードは，Java プログラムの並列性を十分に引き出しており，階層統合型粗粒度タスク並列処理により高い実効性能を達成できることが確認された．

6. おわりに

本稿では，マルチコアプロセッサを対象とし，再帰メソッドを伴う Java プログラムに対して，再帰レベルを考慮した階層統合型粗粒度タスク並列処理を実現する並列 Java コードの生成手法を提案した．本手法を実装した並列化コンパイラは，並列化指示文付 Java プログラムを入力とし，再帰レベルを考慮した階層統合型粗粒度タスク並列処理の並列 Java コードを自動生成することが可能である．

並列化コンパイラにより生成された並列 Java コードをマルチコアプロセッサ上で実行したところ，マージソートの場合には OpenMP の task 構文と比較して 16 スレッドで 5.80 倍の速度向上を実現しており，再帰レベルを考慮した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，再帰レベルの自動決定や，他の再帰プログラムによる性能評価が挙げられる．

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] B. J. Bradel, T. S. Abdelrahman. A study of potential parallelism among traces in Java programs. *Science of Computer Programming*, Vol.74, Issue 5-6, pp.296-313, 2009.
[10] B. J. Bradel, T. S. Abdelrahman. The Use of Hardware Transactional Memory for the Trace-Based Parallelization of Recursive Java Programs. *Proc. Int. Conference on Principles and Practice of Programming in Java*, pp.101-110, 2009.
[11] R. L. Collins, B. Vellore, and L. P. Carloni. Recursion-Driven Parallel Code Generation for Multi-Core Platforms. *Proc. the Conference on Design, Automation and Test in Europe*, pp.190-195, 2010.
[12] K. Seymour and J. Dongarra, “User’s guide to f2j version 0.8,” Innovative Computing Lab., Dept. of Computer Science, Univ. of Tennessee, 2007.