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

越智 佑樹[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 プログラムの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．本手法では階層ごとに粗粒度タスク間の並列性を抽出した後，複数階層の粗粒度タスクをダイナミックスケジューラが統一的に各コアに割り当て，全階層にまたがった並列性を利用することが可能である．本稿では階層統合型粗粒度タスク並列処理の並列 Java コードを，コンパイラにより生成する際，ユーザデータと実行制御データの管理において，静的データ構造を選択的に取り入れる方法を提案する．本手法では，並列 Java コードに静的／動的データ構造を取り入れ，データアクセス時間の軽減を実現する．本稿では Sun UltraSPARC T1 および Intel Xeon E5-2660 上で提案手法の性能評価を行っており，高い実効性能を達成している．

キーワード：並列化コンパイラ，粗粒度タスク並列処理，データ管理，マルチコア，Java プログラム

Parallel Java Code Generation Scheme Using Selective Static Data Structure for Layer-Unified Coarse Grain Task Parallel Processing

Yuki Ochi[1]　Nagatsugu Yamanouchi[1]　Akimasa Yoshida[2]

Abstract: In parallel processing for Java programs on multicore processors, the layer-unified coarse grain task parallel processing scheme, which extracts coarse grain task parallelism between different layers, has been proposed. In this scheme, after coarse grain parallelism of each layer was exploited, dynamic scheduler assigns ready tasks of all layers to cores. This paper proposes a data management method to efficiently handle shared data, such as user data and execution control data, in layer-unified coarse grain task parallel processing. By using both static data structure and dynamic data structure, the proposed method can reduce data access time remarkably. This paper also describes the performance evaluations on Sun UltraSPARC T1 and Intel Xeon E5-2660.

Keywords: Parallelizing Compilers, Coarse Grain Task Parallel Processing, Data Management, Multicore, Java Programs

1. はじめに

近年, 計算速度を向上させるために, スーパーコンピュータ，PC，組み込みシステム，スマートフォンに至るまでマルチコアプロセッサが用いられている．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様々な研究が行われており，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期待されている．

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

1 東邦大学理学部情報科学科
Department of Information Science, Toho 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, Meiji University

© 2013 Information Processing Society of Japan

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．

並列処理の対象言語は Fortan 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 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 [9]，HPF のような配列分散を取り入れた HPJava[10]，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[11] や Jrpm[12] 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の並列性を統一的に利用することは困難である．

本稿では，階層統合型実行制御を伴う粗粒度タスク並列処理を，Java プログラムに対して実現する並列 Java コード生成手法を提案し，その並列化コンパイラを開発した．また，本手法では階層統合型粗粒度タスク並列処理の並列 Java コードにおいて静的／動的データ構造を導入し，データアクセスにおけるオーバーヘッドを軽減する．それゆえ，本コンパイラにより生成された並列 Java コードは，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，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 4 章では，階層統合型粗粒度タスク並列処理における選択的静的データ構造について述べる．第 5 章では，並列化コンパイラについて述べる．第 6 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 7 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][8] 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（またはプロセッサ）に割り当てるダイナミックスケジューリングルーチンを生成する．

例えば，**図 1** の Java プログラムは，階層統合型粗粒度

```
class Other {                 //ユーザ定義クラス
  public static void func() {
    /*mt*/ {
    マクロタスク処理;                      //MT3-1
    }
    /*mt*/ {
    マクロタスク処理;                      //MT3-2
    }
  }
}
public class Main {
  public static void main(String[] args) {
    /*mt*/ {
    マクロタスク処理;                      //MT1-1
    }
    /*mt*/ {
    for (int i=0; i<2; i++) { //for文:MT1-2
      /*mt*/ {
      マクロタスク処理;                    //MT2-1
      }
      /*mt*/ {
      Other.func();  //メソッド呼出し:MT2-2
      }
    }
    }
    /*mt*/ {
    マクロタスク処理;                      //MT1-3
    }
  }
}
```

図 1　並列化指示文を伴う Java プログラム

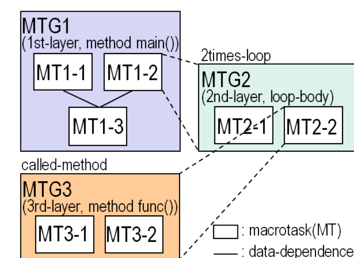

図 2　階層型マクロタスクグラフ（MTG）

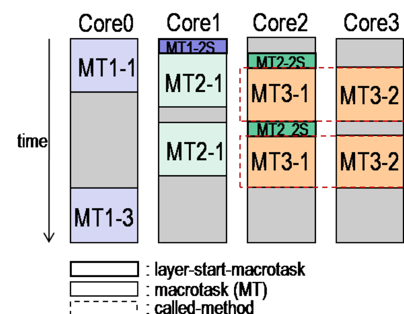

図 3　階層統合型粗粒度タスク並列処理の実行イメージ

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 コア上での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，MT1-1，MT2-1，MT3-1，MT3-2 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

© 2013 Information Processing Society of Japan

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，MT2-2 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）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．

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．

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]．

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	1-1
1	MT1-2 †	true	**1-2S**
1	MT1-3	1-1∧1-2	1-3
1	MT1-4(EndMT)	3	1-4
2	MT2-1	**1-2S**	2-1
2	MT2-2 ††	**1-2S**	**2-2S**
2	MT2-3(CtrlMT)	2-1∧ 2-2	2-3
2	MT2-4(RepMT)	2-3$_{2\text{-}4}$	2-4
2	MT2-5(ExitMT)	2-3$_{2\text{-}5}$	**1-2**
3	MT3-1	**2-2S**	3-1
3	MT3-2	**2-2S**	3-2
3	MT3-3(CtrlMT)	3-1∧3-2	3-3
3	MT3-4(ExitMT)	3-3$_{3\text{-}4}$	**3-4, 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.

2.4 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

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（Critical-Path[7] の大きい）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（プロセッサ）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，それぞれのコアまたはプロセッサ（グルーピングなし）に割り当てられ実行される．

3. 階層統合型粗粒度タスク並列処理の並列 Java コード

本章では，並列化コンパイラにより生成される並列 Java コードの構成について述べる．

3.1 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．このプログラムを後述の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，変数管理クラス（VARmanage$i$，ただし i は MTG 毎に生成）と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（クラス名はユーザの定義したもの），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．変数管理クラスとマクロタスク管理クラスに関しては 4 章で詳しく述べる．

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

```
class VARmanage1 {                   //変数管理クラス
  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 {
  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 MTtable {            //マクロタスク管理テーブル
  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;
  レディMTキュー宣言;
}
class MTmanage{            //マクロタスク管理クラス
   MTtable staticfield;
   ArrayList<MTtable>
        dynamicfield = new ArrayList<MTtable>();
}

class Other {            //ユーザ定義クラスとメソッド
  static void mt3_0() { varm3とmtmに追加 ... }
                                    //階層開始MT3-0
  static void mt3_1() { ... }               //MT3-1
  static void mt3_2() { ... }               //MT3-2
  ...
}

class Mainp {            //並列JavaコードのMainpクラス
  static VARmanage1 varm1 = new VARmanage1;
  static VARmanage2[] varm2 = new VARmanage2[P];//静的用
  static ArrayList<VARmanage2> varm2
                = new ArrayList<VARmanage2>();//動的用
                     :
  static MTmanage[] mtm = new MTmanage[N];
  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) {
    マスタースレッドがレディMTキューに投入;
    while (EndMTが未終了) {
      レディMTキューからCP大の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() { ... }               //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() { Other.mt3_0(); }  //MT2-2
  ...
  public static void main(String[] args) {
    varm（動的用）とmtmにMainp内MT用を追加;
    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

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．

3.2　ダイナミックスケジューラ

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，コアあるいはプロセッサを有効利用するため，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．実装には Java 言語の Runnable インタフェースによるマルチスレッドを用いる．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()` と `notifyAll()` により制御する．

各スレッドのコードでは，コア（プロセッサ）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（プロセッサ）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．

4.　選択的静的データ構造を用いた並列 Java コード

本章では，階層統合型粗粒度タスク並列処理における選択的静的データ構造を用いた並列 Java コード生成手法について述べる．

本手法では，ユーザデータとして，変数管理クラス（VARmanage*i*），ダイナミックスケジューリングのための制御データとして，マクロタスク管理クラス（MTmanage）という二つの共有データクラスを作成し，階層統合型粗粒度タスク並列処理を行っている．従来はこれらのデータに対して，並列 Java コード内において実行時に動的に生成・管理を行う動的データ構造による共有データ管理を採用していたが，本手法ではデータアクセス時間を軽減するため選択的静的データ構造を用いる．

4.1　動的データ構造によるユーザ／実行制御データの管理

階層統合型粗粒度タスク並列処理では，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．この際，内部を並列化対象とするメソッドが複数同時に呼び出される可能性がある．そこで，図 4 の並列 Java コードのようにメソッド内部の変数を管理する VARmanage*i*（i=1〜N，N は MTG 数）クラスと，メソッド内部のマクロタスクを管理する MTmanage クラスのインスタンスを，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している．

VARmanage*i* のインスタンスが生成された後，そのインスタンスの引数用変数に，呼び出し元の実引数のコピーを行う．VARmanage*i* のインスタンスへのアクセスは，各マクロタスクに付加された動的生成識別子によって適切に行われる．MTmanage のインスタンスにも，インスタンスを生成した後 VARmanage*i* のインスタンスと同じ動的生成識別子を付加する．階層開始マクロタスクによって動的に

© 2013 Information Processing Society of Japan

生成されたMTmanageのインスタンスを動的生成識別子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．

Java言語ではListインタフェースのサイズ変更が可能な配列としてArrayListクラスが提供されている．ArrayListクラスは同様のListインタフェースとして提供されているLinkedListクラスに比べ，インデックスを指定しての要素の読み出し，書き換えが高速に行う事が可能となっており，従来の動的データ構造に用いられていた．

しかしながら，このArrayListによる実装では静的な配列と比較した場合，次の二つの点でオーバーヘッドが発生する．(1)ArrayListにインスタンスを追加する際には並列処理領域でVARmanage$i$およびMTmanageのインスタンスの追加を行うため排他制御が必要となる．また，要素の追加の際には要素の増加に伴いメモリ領域の再取得が必要となる場合がありその際にはオーバーヘッドが生じる．(2)静的な配列と比較すると，データアクセス時間が大きい．

4.2 静的データ構造によるユーザ／実行制御データの管理

静的データ構造によるユーザ／実行制御データの管理では，VARmanage$i$クラスとMTmanageクラスのインスタンスをあらかじめ静的な配列として必要となる領域分を用意する．図4のVARmanage$i$クラスはMTGi内部の引数用変数，戻り値用変数，MT間共有変数の宣言で構成されており静的データ構造による管理ではこれらの要素をクラス変数とする．階層開始マクロタスクでは，予め用意された領域のユーザー／実行制御データの初期化を行い，領域の再利用を行う．静的データ構造によるユーザ／実行制御データの管理では4.1節で述べたような動的データ構造へのアクセスによるオーバヘッドを削減することが可能である．静的配列によるデータ管理では，再帰メソッド内部のMTのようにMTの同時呼び出し回数が静的に定まらない場合，VARmanage$i$クラスとMTmanageクラスのインスタンスの個数が静的に求まらず対処することができない．

4.3 静的／動的データ構造を伴う並列Javaコード

本手法では階層統合型粗粒度タスク並列処理の階層ごとに静的／動的データ構造を選択し共有データ管理を行う選択的静的データ構造を用いた並列Javaコード生成手法を提案する．

メソッド内部の変数を管理するVARmanage$i$クラスは選択的静的データ構造では従来と同様にそれぞれのMTGiごとにVARmanage$i$クラスを作成する．VARmanage$i$クラスのインスタンスはMainpクラス内に作成される．静的データ構造による管理を行う階層のVARmanage$i$クラス（例えば図4のVARmanage1）は静的な要素（図4のvarm1)として，動的データ構造により管理を行う階層のVARmanage$i$クラス（例えば図4のVARmanage2）はArrayListとしてインスタンス(図4のvarm2)を作成する．

マクロタスク管理用変数を管理するMTmanageクラスは選択的静的データ構造では静的データ構造による管理を行うための要素staticfield，動的データ構造による管理を行うための要素dynamicfieldを持っている．この二つの要素はマクロタスクのステート管理テーブル宣言，レディ管理テーブル（MTレディ）宣言を要素として持つMTtableクラスのインスタンスとなっている．MTmanageクラスのインスタンスはMainpクラス内でMTG数（図4ではN)の要素を持つ静的な配列として宣言される．

本手法では各階層のMTGごとに静的／動的データ構造を選択して共有データ管理を行い，動的データ構造での管理によるオーバーヘッドの削減を目指す．各MTGでの共有データ管理のデータ構造選択は同一MTの複数同時呼び出しが行われるかどうかである．同一MTの複数同時呼び出しが行われる場合には4.1節で述べたように，VARmanage$i$クラスとMTmanageクラスのインスタンスが同時呼び出し回数分だけ必要となるため，当該MTGを動的データ構造によるデータ管理とする．上記の条件に当てはまらないMTGはすべて静的データ構造による共有データ管理を行う．

図4は静的／動的データ構造を伴う並列Javaコードである．図中の変数NはMTG数，Pは同一MT同時呼び出し回数をそれぞれ表している．変数管理クラスVARmanage$i$に関しては，Mainpクラス内のVARmanage$i$クラスのインスタンス内の変数をアクセスする．VARmanageクラスのインスタンスはMainp内において動的データ構造ではArrayListとして領域を用意し，静的データ構造では静的配列として領域を確保する．但し，1要素の静的配列の場合にはスカラ変数にする．マクロタスク管理クラスMTmanageに関しては，静的データ構造による管理のMTGではMTmanageクラス内のstaticfield，動的データ構造による管理のMTGではMTmanageクラス内のdynamicfieldをアクセスする．階層開始マクロタスクにおいては動的データ構造による管理を行うMTGは従来と同様にインスタンスを追加し，一方，静的データ構造による管理を行うMTGにおいては当該MTGのマクロタスク管理クラスのインスタンスの初期化を行う．

5. 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．

5.1 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．

© 2013 Information Processing Society of Japan

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 Critical-Path 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．選択的静的データ構造については，/*mt stmn*/のように記述された MT の属する MTG に適用する．

5.2 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．

5.3 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．

6. マルチコア上での階層統合型粗粒度タスク並列処理の性能評価

階層統合型粗粒度タスク並列処理用の並列 Java コード生成の性能評価を，マルチコアプロセッサシステム Sun Fire T1000，および，DELL PowerEdge R620 上で行う．

表 2　性能評価プログラム

プログラムの種類	Turb3d	Swim	Jacobi	Pi
全行数	2290	431	27	99
並列化指示文数	68	30	9	2
階層数	7	5	4	1
全 MTG 数	21	8	5	1
全 MT 数（分割後）	470	199	148	36

6.1 性能評価環境

Sun Fire T1000 は，UltraSPARC T1（8 コア，1.0GHz），8GB のメモリから構成されており，OS は Solaris10，Java 処理系は JDK1.6 となっている．16 スレッドの実行時には，8 コアにてハイパースレッディングを用いている．JVM における実行では-Xint オプションをつけている．

一方，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．JVM における実行では-Xint オプションをつけている．

性能評価を行うプログラムは**表 2** に示す 4 種類であり，4.3 節で述べたデータ構造の選択基準に合わせて並列化指示文を加えている．

6.2 ベンチマークプログラムによる性能評価

本性能評価では，表 2 に示す SPECfp95 ベンチマークの Turb3d プログラム，Swim プログラムを用いて評価を行った．Turb3d，Swim プログラムは f2j[13] を用いて Fortran から Java に変換し，それぞれ並列化指示文を加え，本並列化コンパイラでコンパイルする．並列化可能ループは並列化指示文を用いて 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，並列 Java コードを javac でコンパイルし，JVM で実行した．

性能評価ではそれぞれのプログラムに対して，(1) ループ並列処理（階層統合型粗粒度タスク並列処理によるループ並列性のみを利用），(2) 階層統合型粗粒度タスク並列処理（動的データ構造），(3) 選択的静的データ構造を用いた階層統合型粗粒度タスク並列処理の 3 種類の並列実行にて評価を行った．

Turb3d プログラムでの実行結果を**図 5**，**図 6** に示す．図 5 の UltraSPARC T1 上での 16 スレッド（8 コア）の実行結果によると，ループ並列処理（階層統合型並列処理によるループ並列性のみ利用）では逐次処理に対して 7.51 倍の速度向上であるが，階層統合型粗粒度タスク並列処理では，粗粒度並列性を利用することにより 9.44 倍の速度向上が得られた．さらに，提案する選択的静的データ構造を適用すると 10.40 倍の速度向上が得られた．

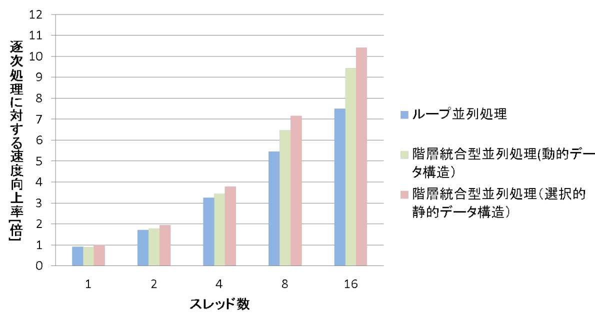

図 5 UltraSPARC T1 上での Turb3d プログラムの階層統合型粗粒度タスク並列処理（8 コア）

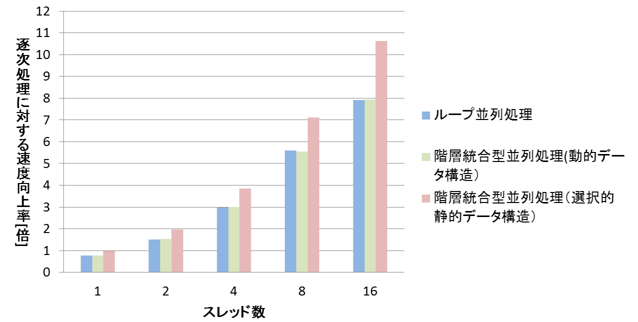

図 7 UltraSPARC T1 上での Swim プログラムの階層統合型粗粒度タスク並列処理（8 コア）

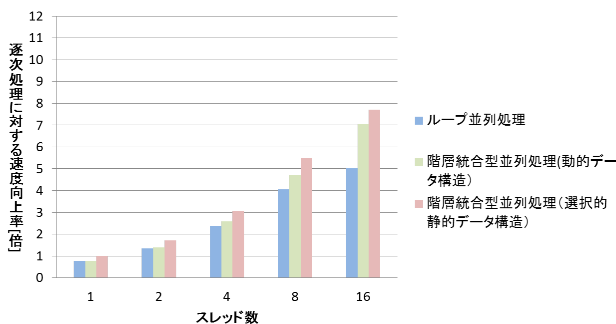

図 6 Xeon E5-2660 上での Turb3d プログラムの階層統合型粗粒度タスク並列処理（8 コア*2 ソケット）

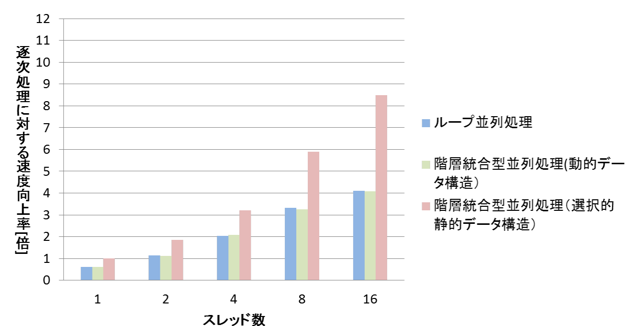

図 8 Xeon E5-2660 上での Swim プログラムの階層統合型粗粒度タスク並列処理（8 コア*2 ソケット）

図 6 の Xeon E5-2660 上での 16 スレッド（8 コア*2 ソケット）の実行結果によると，ループ並列性のみの利用では 5.01 倍の速度向上であるが，階層統合型粗粒度タスク並列処理では 7.04 倍の速度向上が得られた．提案する選択的静的データ構造を適用すると 7.71 倍の速度向上が得られた．

Swim プログラムでの実行結果を**図 7**，**図 8** に示す．このプログラムはループ並列性が十分にあるため，図 7 の UltraSPARC T1 上での 16 スレッド（8 コア）の実行結果によると，ループ並列処理（階層統合型並列処理によるループ並列性のみ利用）では逐次処理に対して 7.91 倍の速度向上であり，階層統合型粗粒度タスク並列処理においても 7.93 倍の速度向上が得られた．さらに，提案する選択的静的データ構造を適用すると 16 スレッド（8 コア）で 10.62 倍の速度向上が得られた．

図 8 の Xeon E5-2660 上での 16 スレッド（8 コア*2 ソケット）の実行結果によると，ループ並列性のみの利用では 4.09 倍の速度向上，階層統合型粗粒度タスク並列処理では 4.08 倍の速度向上が得られた．提案する選択的静的データ構造を適用すると 8.49 倍の速度向上が得られた．

これらの結果から，本並列化コンパイラの生成した並列 Java コードは，Java プログラムの粗粒度並列性とループ並列性を十分に引き出しており，また，選択的静的データ構造を用いたデータ管理によりデータアクセスオーバーヘッドが軽減され，実効性能を向上できることが確認された．

6.3 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 2 つの数値計算を用いる．1 つ目は，ヤコビ法による連立一次方程式求解のプログラム（表 2 の Jacobi）では，対象とする行列サイズは 10000 × 10000 であり，このプログラムでは，収束ループの中に，3 つのループ（for 文）と基本ブロックから構成される．2 つ目は，円周率を求める積分プログラム（表 2 の Pi）であり，$4/(1+x^2)$ 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 5000 万である．各プログラムには，並列化指示文を加えた後，本並列化コンパイラでコンパイルする．並列化可能ループは並列化指示文を用いて 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，この並列 Java コードを javac でコンパイルし，JVM で実行した．

性能評価では選択的静的データ構造による速度向上が，実行制御データとユーザデータに対して，それぞれどの程度の効果があるかを測定した．8 スレッドでの実行において逐次処理に対する速度向上率を次の 4 通りの並列実行で評価する．(1) 階層統合型並列処理で動的データ構造，(2) 階層統合型並列処理で実行制御データの静的化，(3) 階層統合型並列処理でユーザデータの静的化，(4) 階層統合型並列処理で選択的静的データ構造管理．

ヤコビ法による連立一次方程式求解のプログラムでの実行結果を**図 9** に示す．図 9 の実行結果によると，E5-2660 の 8 スレッド（8 コア）の並列実行では，階層統合型粗粒

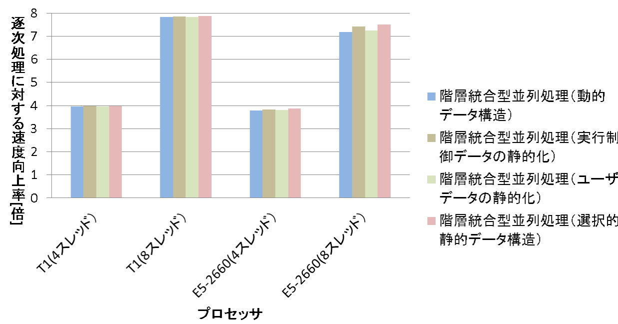

図 9 ヤコビ法プログラムの階層統合型粗粒度タスク並列処理（8 コア）

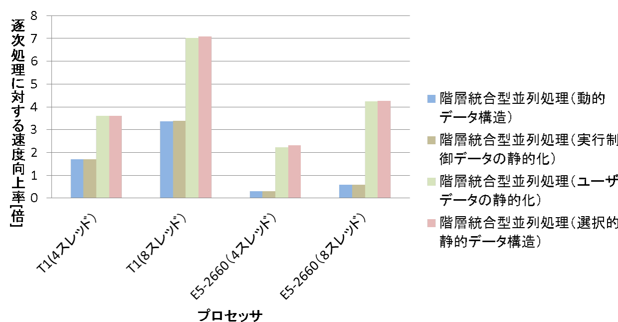

図 10 円周率を求める台形積分の階層統合型粗粒度タスク並列処理（8 コア）

度タスク並列処理の場合に逐次処理の 7.18 倍の速度向上が得られた．実行制御データの静的化の場合には逐次処理の 7.42 倍の速度向上が得られ，ユーザデータの静的化の場合に 7.23 倍の速度向上が得られ，選択的静的データ構造の場合に 7.50 倍の速度向上が得られている．それゆえ実行制御データの静的化による短縮効果が大きい．T1 でも同様の傾向がある．

次に，円周率を求める台形積分プログラムでの実行結果を**図 10** に示す．図 10 の実行結果によると，E5-2660 の 8 スレッド（8 コア）での並列実行では，階層統合型粗粒度タスク並列処理の場合に逐次処理の 0.58 倍の速度向上が得られた．実行制御データの静的化の場合には逐次処理の 0.59 倍の速度向上が得られ，ユーザデータの静的化の場合に 4.23 倍の速度向上が得られ，選択的静的データ構造の場合に 4.26 倍の速度向上が得られている．それゆえ，ユーザデータの静的化による短縮効果が大きい．T1 でも同様の傾向がある．

7. おわりに

本稿では，階層統合型粗粒度タスク並列処理のための並列 Java コードの選択的静的データ構造を用いたデータ管理手法を提案した．本手法では，並列化指示文付 Java プログラムを入力とし，開発した並列化コンパイラを用いて，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，階層統合型並列処理はループ並列処理と比べて高い実効性能を達成しており，また，選択的静的データ構造によりデータアクセス時間を大幅に短縮できた．それゆえ，選択的静的データ構造を用いた階層統合型粗粒度タスク並列処理の並列 Java コード生成手法の有効性が確認された．

今後の課題としては，並列化指示文を Java プログラムに自動挿入するプリプロセッサの開発があげられる．

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，” 情処理学会論文誌，vol.45，no.12，pp.2732–2740，2004.
[8] 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.
[9] 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.
[10] 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.
[11] B. Chan and T.S. Abdelrahman, “Run-time support for the automatic parallelization of Java programs,” J. Supercomputing, vol.28, pp.91–117, 2004.
[12] M.K. Chen and K. Olukotun, “The Jrpm system for dynamically parallelizing Java programs,” Proc. ISCA-30, pp.434–446, 2003.
[13] K. Seymour and J. Dongarra, “User’s guide to f2j version 0.8,” Innovative Computing Lab., Dept. of Computer Science, Univ. of Tennessee, 2007.

© 2013 Information Processing Society of Japan