Vol.2010-HPC-124 No.10
2010/2/23

階層統合型粗粒度タスク並列処理のための Java コンパイラ

小 澤 智 弘[†1]　吉 田 明 正[†1]

マルチコアプロセッサやマルチプロセッサ上での並列処理手法として，複数階層の粗粒度タスク間並列性を利用する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，階層ごとに粗粒度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粗粒度タスクを統一的にコア（またはプロセッサ）に割り当てることにより，階層を越え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，並列化指示文付の Java プログラムから，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並列化コンパイラを提案する．本並列化コンパイラを用いて，Sun Fire T1000 上で性能評価を行った結果，階層統合型粗粒度タスク並列処理は高い実効性能を達成しており，並列化コンパイラにより生成された並列 Java コードの有効性が確認された．

Java Compiler for Layer-Unified Coarse Grain Task Parallel Processing

Tomohiro Ozawa[†1] and Akimasa Yoshida [†1]

In the parallel processing schemes on multicore processors or multiprocessors, the layer-unified coarse grain task parallel processing scheme, which extracts the coarse grain parallelism between different layers, has been proposed. In the layer-unified coarse grain task parallel processing, after the coarse grain task parallelism of each layer was exploited, the dynamic schedulers assign the ready tasks of all layers to cores or processors, so that the parallelism between different layers can be utilized. This paper proposes a parallelizing compiler which generates parallel Java codes to enable the layer-unified coarse grain task parallel processing from Java programs with parallelization directives. The performance evaluations using the parallelizing compiler achieved higher performance on Sun Fire T1000. Consequently, it was confirmed that the parallel Java codes were generated by the parallelizing compiler effectively.

1. はじめに

近年，マルチコアプロセッサやマルチプロセッサ上での並列処理手法として，ループ並列処理[1),2)]や，ループやサブルーチン等の粗粒度タスク間の並列性[3)–8)]を利用する粗粒度タスク並列処理が広く用いられている．

粗粒度タスク並列処理[5),6)]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（プロセッサ）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（プロセッサ）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（プロセッサ）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[5)]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[9)]が提案されている．

並列処理の対象言語は，Fortan や C 言語を対象とするものが主流であったが，最近では，Java 言語における並列処理の期待が高まっている．Java 言語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)]，HPF のような配列分散を取り入れた HPJava[11)]，Container に起因するデータ依存を除去する変換技術[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)] 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実現するための並列 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．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

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

述べる．第 5 章では，SPECfp95 のベンチマークプログラム（f2j[15] で java に変換）に並列化指示文を加え，本並列化コンパイラで生成した並列 Java コードにより性能評価を行う．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では，粗粒度タスク並列処理手法[5] 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[9] 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[9]．階層統合型粗粒度タスク並列処理[9] では，**図 1** の Java プログラムは，**図 2** の階層型マクロタスクグラフ（制御用のマクロタスクは図示されていない）に変換される．このプログラムを 4 コアのプロセッサ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が最大限に利用される．

2.1 対象アーキテクチャ

階層統合型粗粒度タスク並列処理は，現在までに，共有メモリ型マルチプロセッサ（SMP）やマルチコアプロセッサにおいて，OpenMP や POSIX スレッドを用いて実装されてきた[9]．本手法では，階層統合型粗粒度タスク並列処理の Java マルチスレッドコードをコンパイラで生成しており，JVM の動作する各種アーキテクチャで実行可能となる．

2.2 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，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[5]．

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．ここで，繰り返しブロックが処理時間の大きい Doall ループ（あるいはリダクションループ）である場合には，粗粒度並列性を向上させるために，複数の部分 Doall ループに分割し，それらを別々のマクロタスクとして定義する．

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．

```
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 inner*/ {
    for (int i=0; i<2; i++) { //for文:MT1-2
      /*mt*/ {
      マクロタスク処理;                //MT2-1
      }
      /*mt inner*/ {
      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．

2.3 階層開始マクロタスク

階層統合型実行制御[9] 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

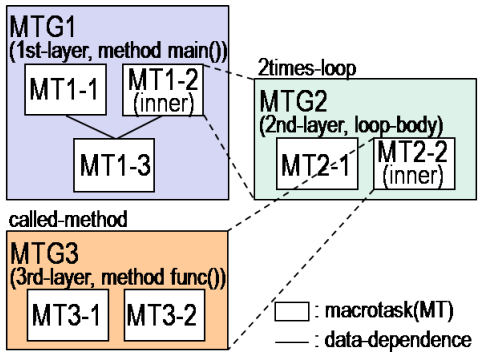

図 2 階層型マクロタスクグラフ（MTG）.

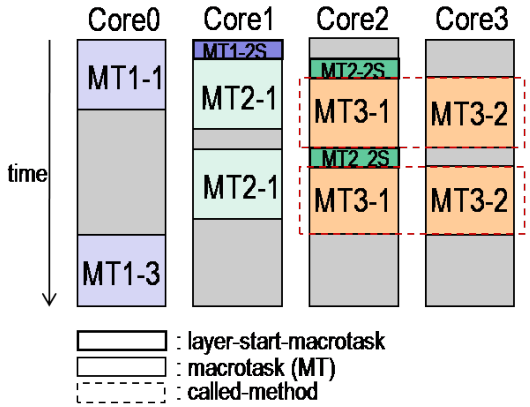

図 3 階層統合型粗粒度タスク並列処理の実行イメージ.

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．

例えば，図 2 の繰返し文の 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 MT2-2 の場合，内部に第 3 階層マクロタスク（MT3-1，MT3-2）を含んでおり，MT2-2 は第 3 階層用の階層開始マクロタスクとして扱われる．

2.4 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[5] 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[5] 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．例えば，図 2 の MT1-3 の最早実行可能条件は，1-1∧1-2 と求められ，MT1-3 は MT1-1 と MT1-2 の実行終了後に実行可能となることを表している．各マクロタスクの最早実行可能条件は，図 2 のような階層型マクロタスクグラフ（MTG）によって表すことが可能である．

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[9]．具体的には，第 L 階層マクロタスクの最早実行可能条件が「true」（即ちその階層が実行可能になればすぐに実行可能）である場合，その条件を「第 L 階層用の階層開始マクロタスク MT_i の終了」に置き換える．階層開始マクロタスクとしての MT_i の実行終了を表す終了通知 iS は，階層開始マクロタスク自身に発行させ，MT_i 内部の第 L 階層の実行終了を表す終了通知 i は，第 L 階層の ExitMT に発行させている．

表 1 において，第 2 階層の MT2-1 と MT2-2 の最早実行可能条件は,「その階層の階層開始マクロタスク MT1-2 の終了（1-2S)」となる．階層開始マクロタスク MT1-2 の終了通知は 1-2S であり，本来の MT1-2 の終了通知 1-2（MT1-2 内部の階層の終了を意味する）は，その内部の MT2-5（ExitMT）により発行される．同様に，階層開始マクロタスク MT2-2 の終了通知は 2-2S であり，本来の MT2-2 の終了通知である 2-2 は，その内部の MT3-4（ExitMT）により発行される．

2.5 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 2.4 節

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)	$2\text{-}3_{2\text{-}4}$	2-4
2	MT2-5(ExitMT)	$2\text{-}3_{2\text{-}5}$	**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)	$3\text{-}3_{3\text{-}4}$	**3-4, 2-2**

(注) † 繰返し文内部の第 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.

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，即ち，絶対 CP 長[16] 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．例えば，図 2 の階層型マクロタスクグラフの場合，図 3 に示すように，MT1-1〜MT1-3 の第 1 階層マクロタスク，MT1-2 内部の MT2-1〜MT2-2 の第 2 階層マクロタスク，MT2-2 内部の MT3-1〜MT3-2 の第 3 階層マクロタスクを統一的に取り扱い，それらを各コアに割り当てて実行することが可能となる．

階層統合型実行制御を伴うダイナミックスケジューリングでは，全階層のマクロタスクを対象としたレディマクロタスクキューを用意する．本手法で採用している分散型ダイナミックスケジューリングの場合は，各コア（プロセッサ）がスケジューリング処理とマクロタスク処理の両方を行う方式である．以下にスケジューリング処理の手順を示す．

(i) 全階層のマクロタスクの中から最早実行可能条件を満たすマクロタスクを，レディマクロタスクキューに投入する．
(ii) レディマクロタスクキューから CP 長の大きいマクロタスクを取り出し，自コア（プロセッサ）に割り当てる．
(iii) 自コア（プロセッサ）に割り当てられたマクロタスクの処理を行う．
(iv) EndMT が終了してない間は (i) に戻る．

3. 階層統合型粗粒度タスク並列処理の並列 Java コード

階層統合型粗粒度タスク並列処理は，既に OpenMP，POSIX スレッドの並列化 API を用いて実装されてきた[9]．本稿では，新たに，並列化指示文を伴う 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用の並列 Java コードを生成する．本章では，並列 Java コードの構成およびデータ管理について述べる．

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．このプログラムを本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）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，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．

ここで，図 4 の並列 Java コードに示すように，繰返し文に対応するマクロタスクコードは，mt1_2() メソッドが，階層開始マクロタスクとして階層開始の処理を行った後，終了通知（表 1 の 1-2S）を発行することにより，その内部の MT2-1 と MT2-2 が新たにレディマクロタスクキューに投入される．また，メソッド呼び出しに対応するマクロタスクコードは，mt2_2() メソッドより，階層開始マクロタスクとなるユーザ定義クラスの Other.mt3_0() が呼び出される．Other.mt3_0 が呼び出された際に，変数管理クラスとマクロタスク管理ク

ラスのインスタンスを動的に生成する．Other.mt3_0() は階層開始の処理を行った後，終了通知を発行することにより，MT3-1 と MT3-2 が新たにレディーマクロタスクキューに投入される．

```
class VARmanage {                    //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 MTmanage {                     //マクロタスク管理クラス
  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;
  レディMTキュー宣言;
}

class Other {                        //ユーザ定義クラスとメソッド
  static void mt3_0() {varm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 ArrayList<VARmanage> varm
                    = new ArrayList<VARmanage>();
  static ArrayList<MTmanage>  mtm
                    = new ArrayList<MTmanage>();
  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．

3.2 ダイナミックスケジューラ

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．この場合，各コア（プロセッサ）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（プロセッサ）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．

図 4 の main() メソッドにおいては，まず，Mainp 内部のマクロタスクのために，変数管理クラス（VARmanage）とマクロタスク管理クラス（MTmanage）のインスタンスを生成する．次に，スレッドコードが開始時に 1 回のみ生成される．以後，マクロタスクスケジューリングとマクロタスク処理は，開始時に生成されたスレッドにより行われるため，スレッド生成に伴うオーバヘッドは軽減される．

各スレッドコードでは，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．

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も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，本手法では，メソッド内部の変数を管理する VARmanage 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する．

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，MT 間共有変数を管理する．VARmanage クラス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 のインスタンスが動的に生成された後，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 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．戻り値が存在するのであれば return 文を含むマクロタスクの実行終了後に，VARmanage のインスタンスの戻り値用変数に保存し，ExitMT が発行された際にその値を呼び出し元に返却する．

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，前述の VARmanage クラスとマクロタスクを管理する MTmanage クラスのインスタンスが生成される．生成された MTmanage 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．生成された MTmanage のインスタンスにも，VARmanage 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．なお，最早実行可能条件においても動的生成識別子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．

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 Java コードを生成する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**表 2** の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）にしたい部分に，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 文等の繰返し文内部や，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．この場合，上位マクロタスクにおいては，内部階層が並列処理の対象となることを指示するため，/*mt inner*/のような並列化指示文を記述する．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．

表 2 並列化指示文.

表記	意味
/*mt*/	マクロタスクの定義
/*mt inner*/	内部並列化対象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 論理式*/	論理式を最早実行可能条件として設定
/*mt decomp=値*/	値の個数にループ（for 文）を分割
/*mt cp=値*/	値を CP 長として設定

図 1 のプログラム（図 2 の階層型マクロタスクグラフに対応）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，Doall 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．これにより，Doall ループの並列性を利用することができる．各マクロタスクの CP 長は，/*mt cp=値*/により記述できる．

本コンパイラでは，基本型変数に関するデータ依存は自動的に求められるが，エイリアスの発生するクラス型変数に関するデータ依存は全て依存を残している．データ依存解析後に最早実行可能条件は自動的に求められる．但し，並列化指示文/*mt 論理式*/のように，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．

4.2 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 言語を用いて開発されており，**図 5** の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 を用いており，抽象構文木を作成する．その後，並列化指示文の情報を用いて，ダイナミックスケジューリングコードを伴う並列 Java コード（Mainp.java）を生成する．

4.1 節の並列化指示文により定義されたマクロタスク間のデータ依存と制御依存を解析して表 1 の最早実行可能条件を自動生成する．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する．

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

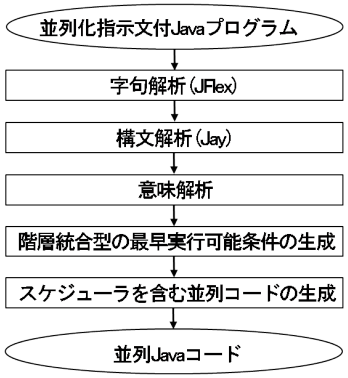

図 5 並列化コンパイラの構成.

高い並列性を引き出すためには，前述の並列化指示文にて最早実行可能条件を直接記述することで対処できる．

4.3 並列化コンパイラの仕様

本コンパイラでは，4.1 節に示す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．

並列化指示文/*mt*/の中で使用可能な制御文は for 文，if-else 文，do-while 文，while 文である．switch 文，および，try-catch 文については対応していない．

本コンパイラの対象となる入力 Java プログラムは，現段階で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．また，複数ファイルからなる Java プログラムの場合，連結して 1 つの入力ファイルにしておく必要がある．

また，メソッド内部の階層統合型粗粒度タスク並列処理については，クラスメソッド（再帰呼び出しも可能）を対象としている．インスタンスメソッドは，上位階層のインスタンスメソッド呼び出し部分をマクロタスクと定義することにより，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である．

5. マルチコアプロセッサ Sun Fire T1000 上での性能評価

本章では，マルチコアプロセッサ Sun Fire T1000 を用いて，提案する並列化コンパイラの性能評価を行う．

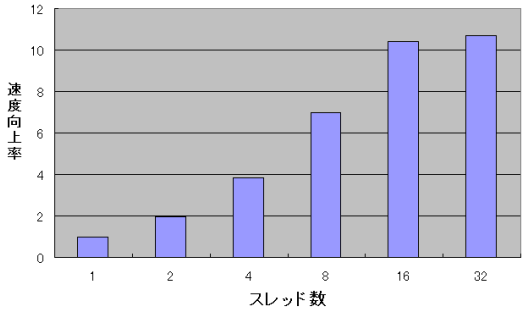

図 6 T1000 上での TURB3D の階層統合型粗粒度タスク並列処理 (8 コア).

5.1 性能評価の環境

Sun Fire T1000 は，UltraSPARC T1（8 コア，1.0GHz），8GB のメモリ，16KB の L1 命令キャッシュ（コア単位），8KB の L1 データキャッシュ（コア単位），3MB の L2 キャッシュ（プロセッサ単位）を備えている．各コアは 4 スレッドを実行可能であり，32 スレッドまでの並行処理が可能である．OS は Solaris 10，Java コンパイラは JDK1.6，Fortran コンパイラは Sun Studio 11 となっている．

5.2 TURB3D による性能評価

本性能評価では，SPECfp95 のベンチマークの TURB3D プログラムを，f2j[15] を用いて Fortran から Java に変換し Java プログラムを用意する．但し，f2j は Fortran の COMPLEX 型に対応していないため，Java プログラムに Complex クラスを手動で追記した．このプログラムは 2160 行（コメントを除く）の Java プログラムであり，21 個のサブルーチンとメインルーチンから構成される．メインルーチンは，繰返しループとなっており，その内部の条件文に応じて該当するサブルーチンが呼ばれる形となっている．21 個のサブルーチンには，それぞれ，3 重ネストループ，2 重ネストループ，1 重ループ，基本ブロックから構成される．f2j では，Fortran のサブルーチンに対して，Java のクラスとクラスメソッドが生成され，各クラスごとに別ファイルとして出力されるが，それらの複数ファイルを 1 つのファイルに連結しておく．

次に，この Java プログラムに，4.1 節の並列化指示文を 47 個加え，開発した並列化コンパイラを用いて，並列 Java コードを生成する．Doall 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．並列化コンパイラによって生成された

並列 Java コードは 13542 行である．その後，並列 Java コードを JDK1.6 の javac でコンパイルし，マルチコアプロセッサ Sun Fire T1000 の JVM で実行した．JVM では-Xint オプションをつけ，JIT コンパイルは適用していない．

実行結果は，**図 6** に示す通り，8 スレッドを用いた場合に 7.0 倍，32 スレッドを用いた場合に 10.7 倍の速度向上が得られた．それゆえ，本コンパイラは，並列化指示文を伴う 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ための並列 Java コードを効率よく生成しており，並列 Java コード生成手法の有効性が確かめられた．

6. おわりに

本稿では，階層統合型粗粒度タスク並列処理の並列 Java コード生成する並列化コンパイラを提案した．本並列化コンパイラは並列化指示文付 Java プログラムを入力対象とし，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，Sun Fire T1000 の 8 コア上で実行したところ，TURB3D プログラムの場合に 10.7 倍の速度向上が得られた．それゆえ，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，並列化コンパイラの制約を軽減することや，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, Jan. 1998.
3) C. J. Brownhill, A. Nicolau, S. Novack, and C. D. Polychronopoulos. Achieving multi-level parallellization. *Proc. of ISHPC'97*, 1997.
4) X. Martorell, E. Ayguade, N. Navarro, et. al. Thread Fork/Join techniques for multi-level parallelism exploitation in NUMA multi-processors. *Proc. of International Conference on Supercomputing*, 1999.
5) 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, 2001.
6) 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.
7) 池田倫久, N.T.Van, 田中雅俊, 福岡岳穂, 片桐孝洋, 本多弘樹, 弓場敏嗣. 粗粒度並列化コンパイラ CoCo の開発. 情報処理学会研究報告, 2004-HPC-98-7, 2004.
8) W. Thies, V. Chandrasekhar, S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. *Proc. IEEE/ACM Int. Symposium on Microarchitecture*, 2007.
9) 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, 2004.
10) A.J.C.Bik, D.B.Gannon. Javar a prototype Java restructuring compiler. *Concurrency: Practice and Experience*, Vol.9, No.11, 1997.
11) S.B.Lim, H.Lee, B.Carpenter, G.Fox. Runtime support for scalable programming in Java. *J. Supercomputing*, 43, pp.165-182, 2008.
12) P. Wu and D. Padua. Container on the Parallelization of General-Purpose Java Programs. *Int. J. Parallel Programming*, Vol.28, No.6, 2000.
13) B.Chan, T.S.Abdelrahman. Run-Time Support for the Automatic Parallelization of Java Programs. *J. Supercomputing*, 28, pp.91-117, 2004.
14) M.K. Chen, K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. *Proc. ISCA-30*, 2003.
15) K.Seymour, J.Dongarra. User's Guide to f2j Version 0.8. *Innovative Computing Lab., Dept. of Computer Science, Univ. of Tennessee*, 2007.
16) A. Yoshida. An Overlapping Task Assignment Scheme for Hierarchical Coarse Grain Task Parallel Processing. *J. Concurrency and Computation: Practice and Experience, Wiley*, Vol.18, Issue11, 2006.