C-014

# 階層統合型粗粒度タスク並列処理のための並列Javaコード生成

Parallel Java Code Generation for Layer-Unified Coarse Grain Task Parallel Processing

小澤 智弘† 山本 義幸† 吉田 明正†
Tomohiro Ozawa Yoshiyuki Yamamoto Akimasa Yoshida

## 1 はじめに

マルチプロセッサやマルチコアプロセッサ上での並列化コンパイラを用いた並列処理では，従来よりループ並列化技術 [1, 2] が用いられてきたが，近年，高い実効性能を達成するために粗粒度タスク並列処理 [3, 4] が広まっている．粗粒度タスク並列処理では，対象プログラムをループやサブルーチン等の粗粒度タスクに階層的に分割して，粗粒度タスク間並列性を抽出し，粗粒度タスクをプロセッサ（コア）に割り当てる．本稿では，階層統合型実行制御 [4] を用いた粗粒度タスク並列処理を対象としており，提案する並列 Java コード生成手法を用いることにより，全階層の粗粒度タスク間並列性を有効利用することができる．本稿では，並列化指示文付 Java プログラムを入力とし，並列 Java コードを生成する並列化コンパイラも実装されており，Sun Fire T1000 上で性能評価を行っている．

## 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 [4] では，まず，粗粒度タスク並列処理 [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）を生成する．次に，階層開始マクロタスクを導入し，全階層のマクロタスクを統一的にプロセッサ（コア）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，図2の階層型マクロタスクグラフに変換される．このプログラムを4コアのプロセッサ上で並列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を利用できる．

### 2.1 マクロタスクと階層開始マクロタスク

粗粒度タスク並列処理による実行では，プログラムを階層的に，基本ブロック，繰返しブロック（for 文や while 文等），サブルーチンブロック（メソッド呼出し）の3種類のマクロタスク（MT）に分割する [3]．例えば，図1の Java プログラムを階層的にマクロタスクに分割し，制御依存とデータ依存を考慮した最早実行可能条件を解析することにより，図2のような階層型マクロタスクグラフ（MTG）が生成できる．

次に，階層統合型実行制御 [4] 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．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できる．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）

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

```
class Other {
  public static void func0 {
    /*mt 3_1 (3_0)*/ {
    MT3_1の処理;
    }
    /*mt 3_2 (3_0)*/ {
    MT3_2の処理;
    }
  }
}
public class Main {
  public static void main(String[] args) {
    /*mt 1_1*/ {
    MT1_1の処理;
    }
    /*mt 1_2 inner*/ {
    for (int i=0; i<2; i++) { //MT1_2:for文
      /*mt 2_1 (2_0)*/ {
      MT2_1の処理;
      }
      /*mt 2_2 inner (2_0)*/ {
      Other.func0; //MT2_2:メソッド呼出し
      }
    }
    }
    /*mt 1_3 (1_1)&(1_2)*/ {
    MT1_3の処理;
    }
  }
}
```

図1 並列化指示文を伴う Java プログラム．

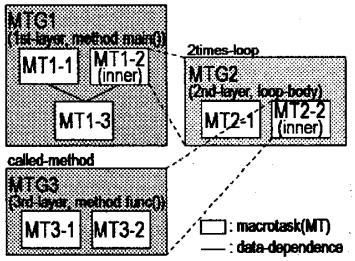


図2 階層型マクロタスクグラフ（MTG）．

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．

### 2.2 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．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．

次に，階層開始マクロタスクの導入により，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．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

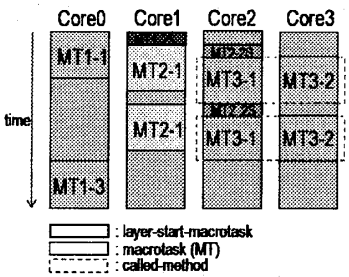


図 3　4 コア上での階層統合型粗粒度タスク並列処理.

表す終了通知 $i$S 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.

## 3　コンパイラによる並列 Java コード生成

本節では，並列化指示文付 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 コードを生成する方法について述べる.

### 3.1　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する．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 MT番号 最早実行可能条件*/`）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，`/*mt MT番号 inner 最早実行可能条件*/`のように inner を並列化指示文に加える．図 1 のプログラムは，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の一例である.

ここで，図 1 の MT2_1 のように，繰返し文の内部のマクロタスクにおいて，階層開始マクロタスクを最早実行開始条件にする場合には，MT2_0 のように「MTG 番号_0」として記述する．本コンパイラで並列 Java コードを生成した場合には，MT2_0 の部分は，実際の呼出し元の階層開始マクロタスク番号（この例では MT1_2S）に置き換えられる.

### 3.2　並列 Java コードの構成

図 1 のプログラムは，開発した並列化コンパイラに入力すると，階層統合型粗粒度タスク並列処理の並列 Java コードが生成される．各スレッドコードでは，プロセッサ（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，自プロセッサ（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排他制御を行う.

並列 Java コードは，ダイナミックスケジューリング用共通データのための Data クラス，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 Main_p クラスから構成される．Main_p 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，新たに実行可能となる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.

### 3.3　並列化コンパイラの実装

本節では，3.2 節の並列 Java コードを生成する並列化コンパイラについて述べる．開発した並列化コンパイラは Java 言語を用いて実装されており，字句・構文解析部分では，LALR(1) コンパイラコンパイラの Jay[5]/JFlex を用いて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含む並列 Java コードを生成する．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，JDK1.2 の文法で記述できるものとする．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，void 型のクラスメソッドに限り対象としている.

## 4　マルチコアプロセッサ Sun T1000 上での性能評価

本性能評価では，マルチコアプロセッサ Sun Fire T1000 を用いる．T1000 は，UltraSPARC T1（1.0GHz, 8 コア）と 8GB のメモリを備えており，OS は Solaris 10，Java コンパイラは JDK1.6 となっている.

本性能評価では，SPECfp95 のベンチマークの SWIM プログラムを，f2j[6] を用いて Fortran から Java に変換し Java プログラムを用意する．この Java プログラムに，3.1 節の並列化指示文を加え，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．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 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.

並列実行結果は，4 コアを用いた場合に 3.81 倍，8 コアを用いた場合に 7.03 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 プログラムに対して，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.

## 5　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 コード生成手法を提案した．本手法では，並列化指示文を Java プログラムに挿入することにより，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，並列化コンパイラで生成することが可能になる.

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 上で実行したところ，SWIM プログラムにおいて高い実効性能を達成することができ，並列 Java コード生成が効果的に行われていることがわかる.

### 参考文献

[1] M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley Publishing Company*, 1996.

[2] Aart J.C. Bik, Dennis B. Gannon. Javar a prototype Java restructuring compiler. *Concurrency: Practice and Experience*, Vol. 9, No. 11, 1997.

[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, 2001.

[4] 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.

[5] Jay: a LALR(1) parser generator. http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html, 2006.

[6] K. Seymour, J. Dongarra. User's Guide to f2j Version 0.8. *Innovative Computing Lab., Dept. of Computer Science, Univ. of Tennessee*, 2007.