社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－HPC－116 (32)
2008／8／7

階層統合型粗粒度タスク並列処理のための並列Javaコード生成手法

吉田 明正[†] 小澤 智弘[†]

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，マルチプロセッサやマルチコアプロセッサ上で効率よく実現するために，ダイナミックスケジューリングコードを伴う並列 Java コードの生成手法を提案する．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統一的にプロセッサ（またはコア）に割り当てることにより，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．また，本稿では提案手法を実装した並列化コンパイラを用いて性能評価を行っており，並列化指示文付 Java プログラムから並列 Java コードを効率よく生成できることが確認された．生成された並列 Java コードを Sun Fire T1000 上で並列実行した結果，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できることが確認された．

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

AKIMASA YOSHIDA[†] and TOMOHIRO OZAWA[†]

This paper proposes a parallel Java code generation scheme with the dynamic scheduling code to realize the layer-unified coarse grain task parallel processing on multiprocessors or multi-core processors effectively. In the layer-unified coarse grain task parallel processing, after task parallelism of each layer was exploited, dynamic schedulers assign the ready tasks of all layers to processors or cores simultaneously, so that the parallelism between different layers can be utilized. In this paper, the performance evaluations, using a parallelizing compiler to implement the proposed scheme, showed that the parallel Java code was generated effectively from the Java program with the parallel directives. The generated parallel Java code could achieve better performance on Sun Fire T1000.

1. はじめに

近年，マルチプロセッサやマルチコア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)~5)]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)]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粗粒度タスク間並列性を効果的に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[6)]が提案されている．Java 並列化コンパイラに関しては，ループ並列化[7)]を対象としたものが研究されているが，粗粒度タスク並列処理は対象としてない．

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，開発した並列化コンパイラによる並列 Java コード生成について述べる．第 4 章では，SPECfp95 の SWIM プログラム（f2j[8)] で java に変換）に並列化指

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

```
class Other {
  public static void func() {
    /*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.func(); //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.

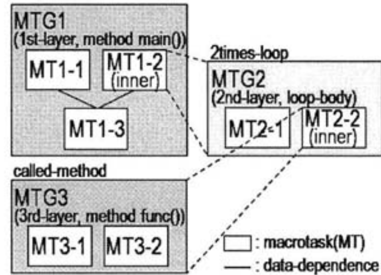

図 2　階層型マクロタスクグラフ（MTG）.

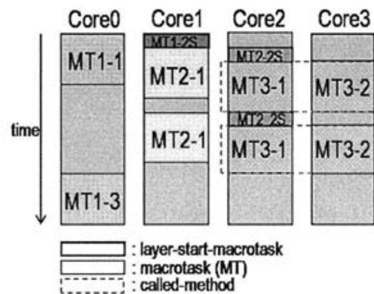

図 3　階層統合型粗粒度タスク並列処理の実行イメージ.

示文を加えて，並列 Java コードをコンパイラで生成し性能評価を行う．第 5 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[6] 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，プロセッサ（またはコア）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．

階層統合型粗粒度タスク並列処理[6] 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 Pthread を用いて実装されてきた[6]．本稿では，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並列化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．

2.3 階層開始マクロタスク

階層統合型実行制御[6] 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．

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）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，$1_1 \wedge 1_2$ 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．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

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	$1_1 \wedge 1_2$	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)	$2_1 \wedge 2_2$	2_3
2	MT2_4(RepMT)	$2_3_{2_4}$	2_4
2	MT2_5(ExitMT)	$2_3_{2_5}$	**1_2**
3	MT3_1	**2_2S**	3_1
3	MT3_2	**2_2S**	3_2
3	MT3_3(CtrlMT)	$3_1 \wedge 3_2$	3_3
3	MT3_4(ExitMT)	$3_3_{3_4}$	**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.

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[11] 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プロセッサ（またはコア）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，それぞれのプロセッサ（コア）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，それらを各プロセッサ（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．分散型ダイナミックスケジューリングの場合は，各プロセッサ（コア）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．以下にその手順を示す．

(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，自プロセッサ（コア）に割り当てる．
(iii) 自プロセッサ（コア）に割り当てられたマクロタスクの処理を行う．
(iv) EndMT が終了していない間は (i) に戻る．

3. コンパイラによる並列 Java コード生成

階層統合型粗粒度タスク並列処理は，既に OpenMP, Pthread, MPI 等の並列化 API を用いて実装されてきた[6),10)]．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，階層統合型粗粒度タスク並列処理のための並列 Java コードを生成する並列化コンパイラを開発した．

本章では，繰返し文やメソッド呼出しを含む Java プログラムに対して，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し，並列化コンパイラにより並列 Java コードを生成する．

並列処理適用部分の指示文は次のように記述する．

```
/*mt MT番号 最早実行可能条件*/ {
  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 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．

データ入力のような前処理部分の指示文は次のように記述する．

```
/*premt*/ {
  処理;
}
```

データ出力のような後処理部分の指示文は次のように記述する．

```
/*postmt*/ {
  処理;
}
```

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．並列化指示文（/*mt MT 番号 最早実行可能条件*/）の最早実行可能条件には，AND（&）と OR（|）と分岐方向（*）を記述することができ，制御依存とデータ依存の条件を合わせて記述することができる．

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可能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．

同様に，図 1 の MT3_1 のように，メソッド呼出し文の内部のマクロタスク（例，MT3_1）において，階層開始マクロタスクを最早実行可能条件にする場合には，MT3_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，MT3_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 MT2_2S）に置き換えられる．

3.2 並列 Java コードのダイナミックスケジューラ

本稿では，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，各プロセッサ（またはコア）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．

Java 言語によるマルチスレッド実装では，Runnable インタフェースと Thread クラスを用いる方法が可能であるが，本手法では，Runnable インタフェースを採用した．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，他プロセッサ（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．

なお，生成するスレッドコードは，プロセッサ（コア）数とし，実行開始時に main() メソッドより，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実行は，最初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，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．Java 言語実装の場合には，`synchronized()` により排他制御を行っている．

3.3 並列 Java コードの構成

図 1 のプログラムは，図 2 の階層型マクロタスクグラフに対応している並列化指示文を加えた Java プログラムである．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，階層統合型ダイナミックスケジューリングの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．

```
class Data { //共通データ
  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() {...} //階層開始MT
  static void mt3_1() {...} //MT3_1
  static void mt3_2() {...} //MT3_2
  ...
}

class Main_p { //並列Javaコード
  static Data data = new Data(); //共通データ
  MT間共有変数の宣言;
  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) {
    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() {...} //MT1_2階層開始MT
  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) {
    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．

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，Main_p 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．マクロタスク間の共有変数は，後述の並列化コンパイラによりリネームを行い Main_p クラスのクラス変数に変換している．

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() 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．

3.4 並列化コンパイラの実装

本節では，3.3 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．本コンパイラは Java 言語を用いて開発されており，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[9]/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（Main_p.java）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．複数クラスを取り扱う場合，連結して 1 つの入力ファイルにしておく必要がある．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．インスタンスメソッド内部では，階層的マクロタスク定義が実装されていないため並列化の対象とはならないが，上位階層のインスタンスメソッド呼出し部分では，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．

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を用いた．T1000は，UltraSPARC T1（8core, 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），

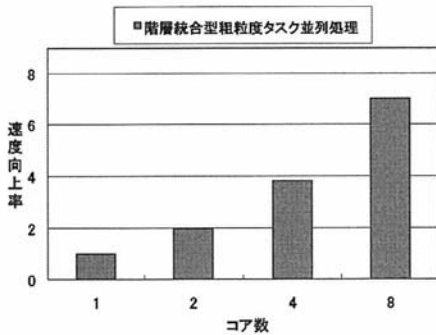

図 5 Sun T1000 上での SWIM プログラムの並列処理.

3MB の L2 キャッシュ（プロセッサ単位）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[8] を用いて Fortran から Java に変換し Java プログラムを用意する．オリジナルの SWIM プログラムは，429 行の Fortran プログラムであり，4 つ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，その内部の条件分岐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．4 つのサブルーチンには，それぞれ，2 重ループ，1 重ループ，基本ブロックから構成される．ここで，処理時間の大きい Doall ループはコア数と同数に分割しておく．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，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 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 5** に示す通り，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．

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，開発した並列化コンパイラで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 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, 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) 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.
7) A. J.C. Bik, D. B. Gannon. Javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, Vol. 9, No. 11, 1997.
8) K. Seymour, J. Dongarra. User's Guide to f2j Version 0.8. *Innovative Computing Lab., Dept. of Computer Science, Univ. of Tennessee*, 2007.
9) Jay: A LALR(1) parser generator. http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html, 2006.
10) 吉田明正. PC クラスタ上での階層統合型粗粒度タスク並列処理. 情報処理学会研究報告, 2006-HPC-107-22, 2006.
11) A. Yoshida. An Overlapping Task Assignment Scheme for Hierarchical Coarse Grain Task Parallel Processing. *Journal Concurrency and Computation: Practice and Experience, Wiley*, Vol. 18, Issue 11, 2006.