Vol.2017-ARC-227 No.38
2017/7/28

メニーコア上での粗粒度並列処理における コードコンパクション

岡 宏樹[1]　吉田 明正[1,2,a)]

概要：Java プログラムの並列処理環境として Fork/Join Framework が導入されており，ワークスティーリングを伴うスケジューラが利用できるようになっている．この Fork/Join Framework を用いて，タスク駆動型実行を伴う並列 Java コードを実装する方法が提案されている．この方法をメニーコア環境に適用する場合，並列ループの分割数に起因して並列コードが長くなる．しかしながら，並列 Java コードの増大は，JVM 上での Java プログラムの実行時間を増加させる傾向がある．そこで本稿では，タスク駆動型実行の並列 Java コードを短縮するコードコンパクション手法を提案する．本手法では，指示文付 Java プログラムを入力として，開発した並列化コンパイラにより Fork/Join Framework を用いたタスク駆動型実行コードを自動生成する．Intel Xeon Phi Knights Landing 上で性能評価を行ったところ，Java Grande Forum Benchmark Suite 2.0 のプログラムに対して，68 コア実行において最大 103 倍の高い速度向上が得られ，提案手法の有効性が確認された．

1. はじめに

マルチコアプロセッサやメニーコアプロセッサは，現在スーパーコンピュータからスマートフォン，組込みシステムに至るまで幅広い機器で利用されている．マルチコアプロセッサを利用した並列処理に関する研究は数多く行われてきたが，更なる実行性能の向上を目指すためには従来のループ並列処理に加えて，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を利用した粗粒度タスク並列処理 [1], [2], [3] が必要である．

High Performance Computing における並列処理の対象言語は，従来より C/C++言語や Fortran 言語が多用されてきたが，プラットフォームを選ばない Java 言語への関心が近年高まっている．これは Java 仮想マシン (JVM) の実行性能が，Oracle の HotSpot 最適化による Just-In-Time(JIT) コンパイラ技術の進歩により向上していることに起因する [4]．

Java 言語における並列処理は，従来より Thread クラス（Runnable インタフェース）が用いられてきたが，Java SE 7 以降では Fork/Join Framework[5] が導入されており，小規模タスクに対して Fork/Join Framework による並列処理を行うことが可能になっている．そこで，Java Fork/Join Framework を用いて Java プログラムの粗粒度並列処理を実現するタスク駆動型実行手法 [6] が提案されている．一方，Java 拡張による並列処理の関連研究としては，コンパイラサポートを伴って async-finish 並列性をサポートする Habanero-Java[7]，配列分散を取り入れた HPJava[8]，共有メモリ/クラスタ並列プログラミングの API を提供する ParallelJava[9] 等が提案されている．

マルチコアプロセッサが広く普及している中，さらに多くのコアを搭載するメニーコアが現在注目されている．例えば，Intel Xeon Phi Knights Landing[10] では，64〜72 コアが搭載されている．しかしながら，従来の粗粒度並列処理コード [6] ではメニーコア向けにループ分割数を増加させた場合にコード長が大きくなり，処理時間が長くなることがある．

そこで本研究では，メニーコアを対象として，Fork/Join Framework を用いたタスク駆動型実行の並列 Java コードを生成する並列化コンパイラを開発した．本並列化コンパイラにより生成された並列 Java コードは，提案するコードコンパクション手法によりメニーコア環境に対応したスレッド数においても低オーバーヘッドで並列性能を引き出すことができる。コードコンパクション手法では，従来の並列 Java コードと比較してコード長の短い並列 Java コードを生成することで，JVM が備える HotSpot 機能をメニーコア環境で十分に活かすことができ，より高い並列処理性能を発揮することが可能となる．

本稿の構成は以下の通りとする．第 2 章では，タスク駆

[1] 明治大学大学院先端数理科学研究科
[2] 明治大学総合数理学部
[a)] akimasay@meiji.ac.jp

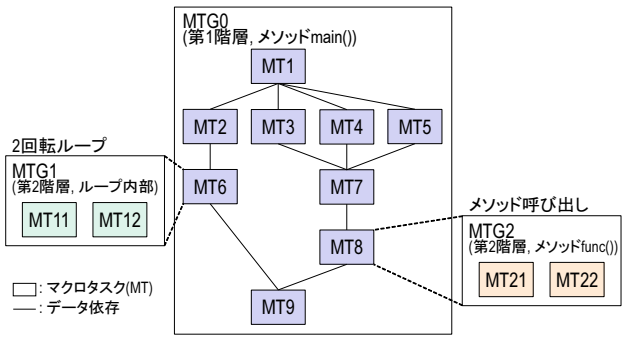

図 1　マクロタスクグラフ (MTG).

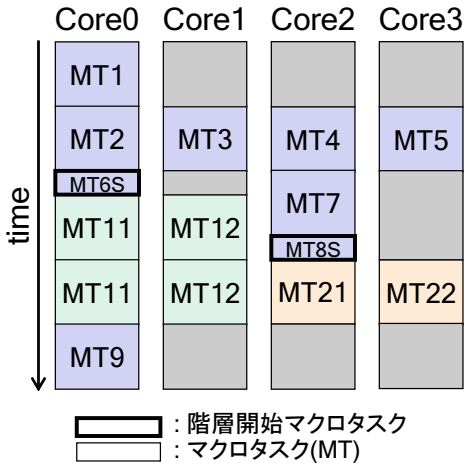

図 2　4 コアでのタスク駆動型実行の並列処理イメージ.

動実行による粗粒度並列処理について述べる．第 3 章では，コードコンパクションを伴う粗粒度並列処理コードについて述べる．第 4 章では，コードコンパクションを実装した並列化コンパイラについて述べる．第 5 章では，Intel Xeon Phi Knights Landing Server 上での性能評価について述べる．第 6 章でまとめを述べる．

2. タスク駆動型実行による粗粒度並列処理

本章では，階層統合型実行制御による粗粒度並列処理 [2], [3] を，Fork/Join Framework 環境で実現するためのタスク駆動型実行手法について述べる．

2.1 タスク駆動型実行の概念

本稿で提案するタスク駆動型実行 [6] は，Java Fork/Join Framework 環境で粗粒度並列処理を実現するための，データ依存と制御依存を考慮した粗粒度タスク（マクロタスク）の実行手法である．

タスク駆動型実行において，マクロタスクの階層的定義およびマクロタスク間の並列性抽出に関しては，階層統合型実行制御 [2], [3] を採用する．具体的には，入力プログラムの構造に対応した階層を定義し，各階層のマクロタスク間のデータ依存と制御依存を解析して，最早実行可能条件 [2] の形で並列性を表現する．これはマクロタスクグラフ (MTG)[1] として表現される．

表 1　タスク駆動型実行の最早実行可能条件.

MTG	MT	最早実行可能条件	終了・分岐の記録	後続 MT 候補
0	1	true	1	2,3,4,5
	2	1	2	6
	3	1	3	7
	4	1	4	7
	5	1	5	7
	6†	2	**6S**	10
	7	3∧4∧5	7	8
	8†	7	**8S**	21,22
	9	6∧8	9	End
	End‡	9	—	—
1	10‡(Loop)	**6S**	10	11,12
	11	10	11	13
	12	10	12	13
	13‡(Ctrl)	11∧12	$13_{14} \vee 13_{15}$	14,15
	14‡(Repeat)	13_{14}	14	10
	15‡(Exit)	13_{15}	**6**	9
2	21	**8S**	21	23
	22	**8S**	22	23
	23‡(Ctrl,Exit)	21∧22	**8**	9

†：階層開始マクロタスク
‡：制御用マクロタスク
13_{14}：MT13 が終了して MT14 に分岐
13_{15}：MT13 が終了して MT15 に分岐

その後，提案するタスク駆動型実行においては，並列化コンパイラによって生成された並列 Java コードがマクロタスクの終了状態と分岐状態を管理し，当該マクロタスクの状態の変化により実行可能になるマクロタスクを Fork/Join Framework を用いて fork する．ただし，fork されたマクロタスクは直ちに実行されるわけではなく，各スレッドが所有するワーカーキューに投入される．その後，Fork/Join Framework のスケジューラはワーカーキューに投入されたマクロタスクを取り出して，ワーカースレッドで実行する．また，このときワーカースレッドおよびワーカーキューの状態に応じてワークスティーリングが行われる．

ここで，プログラムの並列性を表現した**図 1** のマクロタスクグラフ（ただし，制御用マクロタスクは図示していない）を用いて実行手順を説明する．各マクロタスクの最早実行可能条件が**表 1** に示されている．このプログラムを 4 コア（4 スレッド）上で実行したイメージが**図 2** に示されており，マクロタスク間の並列性が最大限に利用されていることがわかる．例えば，図 1 の MT1 の実行が終了すると，表 1 に示される後続マクロタスク候補 MT2〜MT5 に対して最早実行可能条件の判定が行われ，MT2〜MT5 が fork によりワーカーキューに投入される．そして，各ワーカースレッドがワーカーキューから MT2〜MT5 をそれぞ

れ取り出して実行する．一方，MT3，MT4，MT5 の場合，それぞれの後続マクロタスク候補は MT7 である．この場合は，MT3，MT4，MT5 のうち，最後に実行が終了したマクロタスクが，MT7 を fork してワーカーキューに投入することになる．

2.2 階層的なマクロタスク定義

階層統合型実行制御 [2] による粗粒度並列処理では，まず与えられた対象プログラム（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．次に，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は，階層的にマクロタスクを定義する．

階層統合型実行制御を適用する場合，全ての階層のマクロタスク，すなわち繰り返しブロック内部やメソッド呼出しブロック内部のサブ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．例えば，図 2 の MT6S は図 1 の MT6（2 回転ループ）の階層開始マクロタスク，図 2 の MT8S は図 1 の MT8（メソッド呼出し）の階層開始マクロタスクである．

2.3 タスク駆動型実行における最早実行可能条件

マクロタスクの生成後，マクロタスク間の並列性を最大限に引き出すため，制御依存とデータ依存を考慮した最早実行可能条件 [1], [2] を解析する．

例えば，表 1 において MT6 の最早実行可能条件を考えると，MT2 の実行が終了した後に MT6 は実行可能となる．MT6 は繰り返しブロックであるため，MT6 の階層開始マクロタスクである MT6S の終了を記録する．これにより，後続マクロタスク候補である MT10（制御用マクロタスク，mtLoop）の最早実行可能条件の判定が行われ，MT10 は実行可能となる．MT10 が終了すると，その終了を記録し，MT10 の後続マクロタスク候補である MT11，MT12 が実行可能となる．繰り返しブロックの繰り返し判定は表 1 の MT13(mtCtrl) が行っている．繰り返しを継続する場合には MT13 から MT14(mtRepeat) に分岐し，繰り返しを終了する場合には MT13 から MT15(mtExit) に分岐する．

2.4 タスク駆動型実行によるスケジューリング

提案するタスク駆動型実行におけるマクロタスクの実行管理では，後続マクロタスク候補に対してそれぞれ最早実行可能条件の判定を行い，条件を満たしている（実行可能な）場合にはその後続マクロタスクを fork する．fork された後続マクロタスクは Fork/Join Framework 環境のワーカーキュー（各ワーカースレッドが保持）に投入され，ワークスティーリングを伴うスケジューラにより取り出されて，ワーカースレッドで実行される．

2.5 Java Fork/Join Framework

Java Fork/Join Framework は Java SE 7 以降で導入されている，ExecutorService インタフェースを実装した並列処理フレームワークである．本フレームワークを利用することで処理を小さな単位（タスク）に分割することができ，それらを複数のコアを用いて処理することが可能となる．

Fork/Join Framework では，まずスレッドプールを作成し，その中に任意の数のワーカースレッドを生成する．スレッドプール内に生成された各ワーカースレッドはそれぞれ独自のワーカーキューを持っており，fork されたタスクは各々のワーカースレッドのワーカーキューへ投入される．その後，各ワーカースレッドがワーカーキューからタスクを取り出して実行する．実行されるタスクは内部で compute() メソッドを処理している．

本フレームワークを用いることで，スレッド並列処理に比べて並列処理の記述が容易になるというメリットがある．また，本フレームワークの特徴的な機能であるワークスティーリングの仕組みにより，ロック競合の問題に対応することが可能である．

3. コードコンパクションを伴う粗粒度並列処理コード

本章では，タスク駆動型の粗粒度並列処理コードと提案するコードコンパクション手法について述べる．

3.1 タスク駆動型並列 Java コード

Java Fork/Join Framework を用いたタスク駆動型並列 Java コードは，主に Data クラス，Other クラス（ユーザ定義），Mainp クラスによって構成される．そのうち，並列処理を行う Mainp クラスが**図 3** に示されている．図 3 の ForkTemplate_main クラスはソースプログラムのメソッドに対応しており，その内部では実行管理付マクロタスクコードがマクロタスク数分用意されている．

並列 Java コードの実行では，まず Mainp クラス内に存在する main() メソッド（図 3 の 42〜46 行目）の処理が行われる．main() メソッドでは，指定された数だけワーカースレッドを持つスレッドプールを生成する．その後，Layer0 クラスのインスタンスを invoke() メソッドで呼び出すことにより，Fork/Join Framework による並列処理が開始される．

第 0 階層 MTG として用意された Layer0 クラス（図 3 の 2〜10 行目）のインスタンスは内部の compute() メソッドを実行し，ForkTemplate_main クラスの mtStart を fork する．これにより，ForkTemplate_main クラスの compute() メソッド（図 3 の 16〜18 行目）が実行され，マクロタスク識別情報を基に mtStart() メソッド（図 3 の 19〜22 行目）

```
01: class Mainp { // 並列メイン
02:     static class Layer0 extends RecursiveTask<Void> { //ForkJoin開始
03:         Layer0() { //コンストラクタ
04:             Dataクラスのフィールド変数の初期化;
05:         }
06:         protected void compute(){
07:             ForkTemplate_mainクラスのmtStartをforkする;
08:             joinを行う;
09:         }
10:     }
11:     public static class ForkTemplate_main extends RecursiveTask<Void> {
12:         マクロタスク間共有変数等の宣言;
13:         Main(MT識別情報) { //コンストラクタ
14:             MT識別情報をフィールド変数に設定;
15:         }
16:         protected void compute() {
17:             該当するマクロタスクを実行;
18:         }
19:         public void mtStart(){
20:             マクロタスク実行管理テーブル更新;
21:             後続マクロタスクのforkを試みる;
22:         }
23:         public void mt1() {
24:             ForkTemplate_mt1.execute();
25:             マクロタスク実行管理テーブル更新;
26:             後続マクロタスクのforkを試みる;
27:         }
28:         static class ForkTemplate_mt1 extends RecursiveTask<Void> {
29:             static void execute() {
30:                 MTをサブタスクに分割してforkする;
31:             }
32:             private ForkTemplate_mt1(サブタスク識別情報) { //コンストラクタ
33:                 サブタスク識別情報をフィールド変数に設定;
34:             }
35:             protected Void compute() {
36:                 サブタスクを実行する;
37:             }
38:         }
39:         public void mt2() { ... }
40:         ...
41:     }
42:     public static void main(String[] args) {
43:         ForkJoinPool pool = new ForkJoinPool(ワーカースレッド数);
44:         Layer0 layer0 = new Layer0();
45:         pool.invoke(layer0); //ForkJoin開始
46:     }
47: }
```

図 3 サブタスクレベル分割を伴うタスク駆動型並列 Java コードの Mainp クラス．

が実行される．そして，mtStart() メソッドの終了直前に，後続マクロタスク候補の中から実行可能なマクロタスクを fork することにより，後続のマクロタスクが実行可能となる．これらの一連の手順の繰り返しによって，全てのマクロタスクが実行される．

3.2 コードコンパクションのためのサブタスクレベル分割

並列化可能ループの並列処理のために，従来は他のマクロタスクと同様の分割（マクロタスクレベル分割）を行っていた．しかしながら，この手法を適用してメニーコア環境に対応した，例えば 272 スレッド用に並列化可能ループを 272 分割したコードを生成すると，並列 Java コードのコード長が非常に長くなってしまい，JVM が備える HotSpot 最適化による JIT コンパイラの機能を十分に活かすことができなくなってしまう．

提案するコードコンパクション手法では，並列化可能ループに対して，マクロタスクレベル分割ではなくサブタスクレベル分割を適用する．サブタスクレベル分割では，並列化可能ループを一つのマクロタスクとして扱い，マクロタスクの内部でサブタスクに分割する．また，全イタレーションを予め指定された分割数に分割したイタレーションの集合をサブタスクとして定義し，マクロタスクスケジューリングの対象とせずに Fork/Join Framework のワーカーキューに直接投入する．

mt1() メソッド（図 3 の 23〜27 行目）および ForkTemplate_mt1 クラス（図 3 の 28〜38 行目）では一つの並列化可能ループの処理を行っている．ForkTemplate_main クラスの compute() メソッドから実行された mt1() メソッドは，ForkTemplate_mt1 クラス内部の execute() メソッド（図 3 の 29〜31 行目）を実行する．execute() メソッドは予め決められた数のサブタスクを fork し，それぞれのサブタスクはサブタスク識別情報を基に与えられた並列化可能ループの範囲を計算して実行する．全てのサブタスクの実行が終了すると execute() メソッドも終了し，後続マクロタスク候補の中から実行可能なマクロタスクを fork する．

4. コードコンパクションを実装した並列化コンパイラ

本章では，コードコンパクションを伴うタスク駆動型の粗粒度並列処理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 並列化コンパイラ

本研究で開発基盤とした並列化コンパイラ [6], [11] は Java 言語を用いて開発されており，字句解析と構文解析においては LALR(1) のパーサジェネレータである Jay/JFlex を用いている．本並列化コンパイラでは，並列化指示文を付加した Java プログラムからマクロタスクを定義し，各マクロタスクの最早実行可能条件 [2] を求めた後，並列 Java コードを生成する．

4.2 並列化指示文

タスク駆動型実行による粗粒度並列処理を実現する場合，入力対象となる Java プログラムに対して適切な並列化指示文を付加し，本並列化コンパイラで並列 Java コードを生成する．この際，/*mt fork*/のような並列化指示文を付加することで，マクロタスクの定義をすることができる．また，繰り返し文やクラスメソッド等のマクロタスクの内部においてサブマクロタスクを階層的に定義する場合には，マクロタスクに/*mt fork inner*/の並列化指示文を付加し，内部のサブマクロタスクに/*mt fork*/を付加することにより，複数階層のマクロタスク間の並列性を利用することが可能になる．並列化可能ループは，/*mt fork fdecomp=分割数*/のような並列化指示文を付加することにより，指定された分割数のサブタスクレベル分割を行うマクロタスクとして定義される．一方で，並列化可能ループに/*mt fork decomp=分割数*/のような並列化指示文を付加する場合には，並列化可能ループに対して従来のマクロタスクレベル分割を伴う並列 Java コードを生成する．

表 2　性能評価マシン.

マシン	Intel Xeon Phi Knights Landing Server
プロセッサ	Intel Xeon Phi Processor 7250
コア	68cores (272threads), 1.4GHz
メモリ	64GB
OS	CentOS 7.2
Java 処理系	JVM1.8

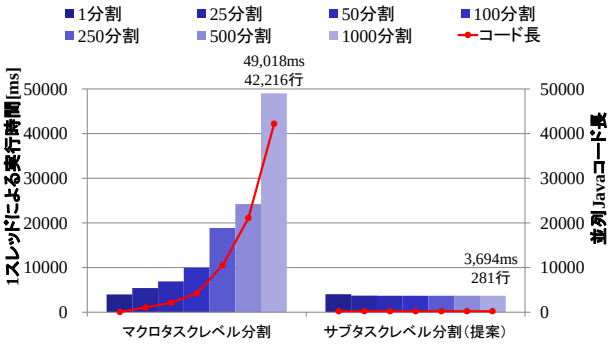

図 4　台形積分プログラムにおける並列 Java コード長と 1 スレッド実行時間.

5. Intel Xeon Phi Knights Landing Server 上での性能評価

本章では，Intel Xeon Phi Knights Landing Server において，コードコンパクションを伴う粗粒度並列処理コードの性能評価を行う．

5.1 性能評価の環境

性能評価マシンには**表 2** の Intel Xeon Phi Knights Landing Server を用いている．これは 68 コア搭載の Intel Xeon Phi Processor 7250 を搭載しており，OS は CentOS 7.2，Java 処理系は JVM1.8 が用いられている．

5.2 節の性能評価では台形積分プログラムを使用し，5.3 節では Java Grande Forum Benchmark Suite Version 2.0[12] の Crypt，Series，Monte Carlo，Raytracer を使用した．それぞれのプログラムには，定数伝播，インライン展開，変数のプライベート化等のリストラクチャリングを予め行っている．提案するコードコンパクションを伴う粗粒度並列処理を行うためには，並列化指示文を加えた並列化対象のソースプログラムを並列化コンパイラの入力とし，並列 Java コードを生成する．

5.2 並列 Java コード長と 1 スレッド実行時間の評価

本節では，台形積分プログラムを用いて，1 スレッド実行においてコード長が実行時間に与える影響についての評価を行う．この台形積分プログラムは，定積分 $\int_0^1 \frac{4}{1+x^2}dx$ を，積分領域を 10^8 個に分割して近似計算する円周率計算プログラムである．コード長は 35 行であり、このコードには 2 つの並列化指示文を挿入している．台形積分プログラムの Xeon Phi での逐次実行時間は 4,039 ミリ秒であった．

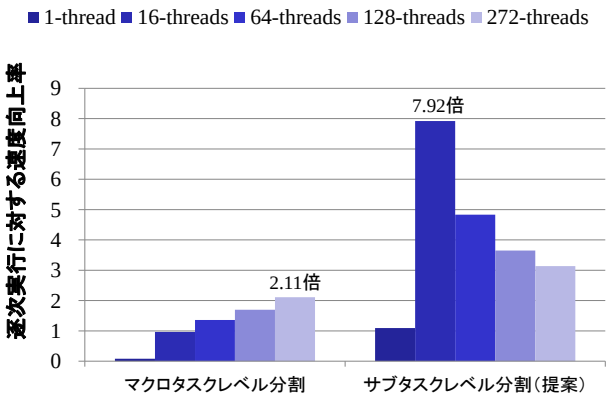

図 5　台形積分プログラムの Xeon Phi 上での速度向上率.

台形積分プログラムの並列化可能ループに対して，従来のマクロタスクレベル分割を適用した場合と，提案するサブタスクレベル分割を適用した場合について，並列 Java プログラムの 1 スレッドの実行時間とコード長を**図 4** に示している．ただし，それぞれの場合に対して分割数を 1，25，50，100，250，500，1000 の 7 種類で測定している．

図 4 に示す通り，マクロタスクレベル分割の場合にはマクロタスクの分割数の増加に伴って実行時間が長くなっており，1,000 分割のときには 49,018 ミリ秒である．また，このとき並列 Java コード長も分割数の増加に伴って長くなっており，1,000 分割のときには 42,216 行である．

それに対し，サブタスクレベル分割の場合には 1,000 分割のときの実行時間は 3,694 ミリ秒であり，分割数の増加に伴って実行時間が長くなることはない．また，このとき並列 Java コード長は分割数に関わらず 281 行で一定である．

図 5 には，分割数が 1,000 のときの台形積分プログラムの Xeon Phi 上での速度向上率を示している．図 5 に示すように，マクロタスクレベル分割では最大 272 スレッドで逐次実行比 2.11 倍，提案するサブタスクレベル分割では最大 16 スレッドで逐次実行比 7.92 倍と，提案手法が従来手法を大きく上回る速度向上となっている．

5.3 Java Grande Benchmark Suite による性能評価

本節では，**表 3** の 4 種類のベンチマークプログラムを用いた性能評価を行う．

Crypt は，共通鍵暗号方式によるデータ暗号化アルゴリズムを用いた暗号化と復号化を行うプログラムである．コード長は 308 行であり，このコードには 5 つの並列化指示文を挿入している．並列化コンパイラにより生成された並列 Java コードは，ループ分割数 500 にてループ分割が行われており，コード長は 600 行である．このプログラムの逐次実行時間は 6,202 ミリ秒である．並列 Java コードによる実行結果は**図 6** に示すように，マクロタスクレベル

表 3　性能評価に用いた Java Grande Forum Benchmark Suite の特性.

特性	Crypt	Series	Monte Carlo	Raytracer
プログラムの種類	暗号化処理	フーリエ級数	モンテカルロ法	光線追跡法
データセット	C（N = 5000 万）	B（N = 10 万）	A（N = 1 万）	B（N = 500）
並列化対象のソースコード長	308	505	553	448
タスク駆動型並列 Java コード長	600	743	860	563
並列化対象外のソースコード長（ファイル数）	—	—	2,585 (11)	998 (12)
並列化指示文数	5	6	5	3
ループ分割数	500	1,000	1,000	500
Xeon Phi 上での逐次実行時間 [ms]	6,202	134,920	7,525	37,668

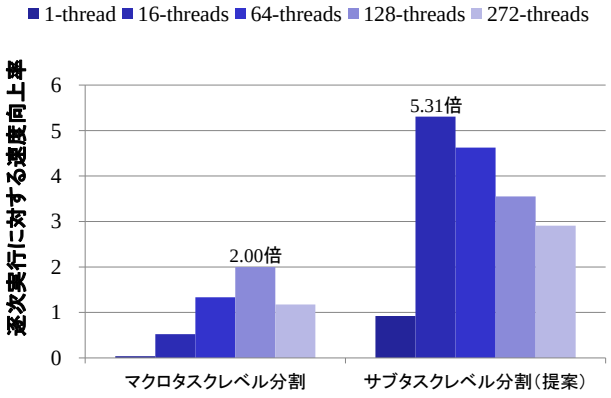

図 6　Crypt の Xeon Phi 上での速度向上率.

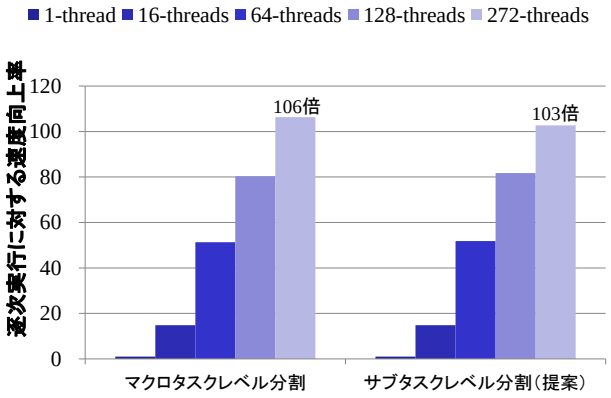

図 7　Series の Xeon Phi 上での速度向上率.

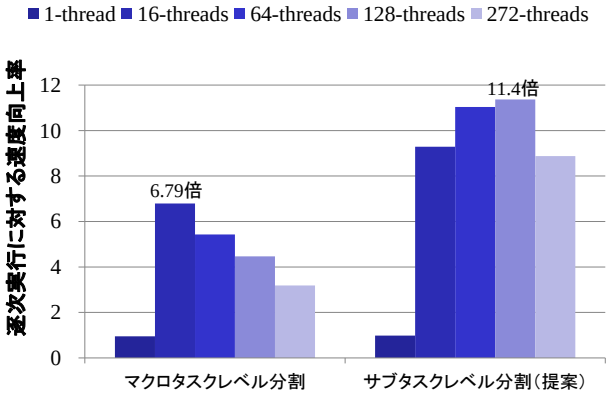

図 8　Monte Carlo の Xeon Phi 上での速度向上率.

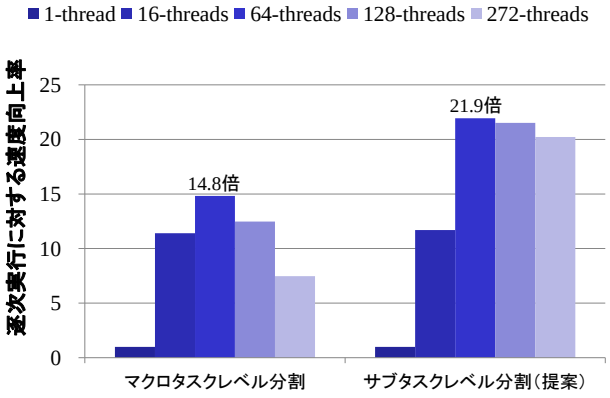

図 9　Raytracer の Xeon Phi 上での速度向上率.

分割では 128 スレッド実行で逐次実行比 2.00 倍，提案するサブタスクレベル分割では 16 スレッド実行で逐次実行比 5.31 倍と，提案手法が従来手法を大きく上回る速度向上となっている．

Series は，フーリエ級数を求めるプログラムである．並列 Java コードによる実行結果は**図 7** に示すように，マクロタスクレベル分割では 272 スレッド実行で逐次実行比 106 倍，提案するサブタスクレベル分割では 272 スレッド実行で逐次実行比 103 倍と，提案手法と従来手法は同程度の速度向上となっている．

Monte Carlo は，モンテカルロ法による金融シミュレーションを行うプログラムである．並列 Java コードによる実行結果は**図 8** に示すように，マクロタスクレベル分割では 16 スレッド実行で逐次実行比 6.79 倍，提案するサブタスクレベル分割では 128 スレッド実行で逐次実行比 11.4 倍と，提案手法が従来手法を大きく上回る速度向上となっている．

Raytracer は，光線を追跡することである点において観測される像のシミュレーションを行うプログラムである．並列 Java コードによる実行結果は**図 9** に示すように，マクロタスクレベル分割では 64 スレッド実行で逐次実行比 14.8 倍，提案するサブタスクレベル分割では 64 スレッド実行で逐次実行比 21.9 倍と，提案手法が従来手法を大きく上回る速度向上となっている．

以上の結果，本並列化コンパイラで生成したコードコンパクションを伴う並列 Java コードは，従来手法と同程度，またはそれを上回る高い速度向上が得られた．これにより，コードコンパクションを伴う並列 Java コードはメニーコア環境において高い実効性能を達成できることが確かめられた．

6. おわりに

本稿では，メニーコア環境において，コードコンパクションを伴うタスク駆動型実行の粗粒度並列処理コードの生成手法を提案し，その並列化コンパイラを開発した．本並列化コンパイラは，並列化指示文を加えた Java プログラムを入力として，Fork/Join Framework を用いた並列 Java コードを容易に生成することができる．生成された並列 Java コードは，マクロタスク実行管理を行いつつ，Fork/Join Framework のスケジューラを利用した粗粒度並列処理を効率良く行うことができる．

性能評価では，並列化指示文を加えたベンチマークプログラムを並列化コンパイラへ入力し，コードコンパクションを伴う並列 Java コードを生成した．提案するサブタスクレベル分割では、従来のマクロタスクレベル分割に比べて並列 Java コード長を短縮し，さらに Intel Xeon Phi Knights Landing Server 上での実行時間を大幅に短縮することができた．

以上の結果，メニーコア環境において，Java Fork/Join Framework を利用したコードコンパクションを伴うタスク駆動型実行の粗粒度並列処理の有効性が確認された．

本研究の一部は，JSPS 科研費基盤研究 (C) 課題番号 16K00174 の助成により行われた．

参考文献

[1] 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).
[2] 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).
[3] Yoshida, A., Ochi, Y. and Yamanouchi, N.: “Parallel Java Code Generation for Layer-unified Coarse Grain Task Parallel Processing”, 情報処理学会論文誌 ACS, Vol. 7, No. 4, pp. 56–66 (2014).
[4] Taboada, G. L., Ramos, S., Expósito, R. R., Touriño, J. and Doallo, R.: Java in the High Performance Computing Arena: Research, Practice and Experience, *Science of Computer Programming*, Vol. 78, No. 5, pp. 425–444 (2011).
[5] Lea, D.: A Java Fork/Join Framework, *Proc. ACM conference on Java Grande, JAVA'00*, pp. 36–43 (2000).
[6] Yoshida, A., Kamiyama, A. and Oka, H.: “A Task-driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform”, *Journal of Information Processing*, Vol. 25, pp. 426–437 (2017).
[7] Imam, S. and Sarkar, V.: “Habanero-Java Library: a Java 8 Framework for Multicore Programming”, *Proc. International Conference on Principles and Practices of Programming on the Java platform*, pp. 75–86 (2014).
[8] Lim, S. B., Lee, H., Carpenter, B. and Fox, G.: Runtime support for scalable programming in Java, *J. Supercomputing*, Vol. 43, pp. 165–182 (2008).
[9] Kaminsky, A.: Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java, *Proc. IEEE Int. Parallel and Distributed Processing Symposium* (2007).
[10] Jeffer, J., Reinders, J. and Sodani, A.: *“Intel Xeon Phi Processor High Performance programming Knights Landing Edition”*, Morgan Kaufmann (2016).
[11] 神山彰，吉田明正：“Java Fork/Join Framework を用いた粗粒度並列処理コードの自動生成”，情報処理学会研究報告，Vol. 2015-ARC-214, No. 5 (2015).
[12] EPCC: “Java Grande at EPCC”, ⟨https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index.php⟩ (2017).