Vol.2016-ARC-218 No.11
Vol.2016-SLDM174 No.11
2016/1/19

Androidプラットフォームにおける Java Fork/Join Frameworkを用いた粗粒度並列処理

吉田 明正[1,a)]　神山 彰[2]

概要：Androidプラットフォームにおける Javaプログラムの並列処理環境として，Android API Level 21より Fork/Join Framework が導入されている．Fork/Join Framework は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おり，再帰や分割統治のプログラムにおいて利用されることが多いが，一般的なプログラムに適用して粗粒度タスク間の並列性を引き出すことは困難であった．本稿では，このような問題点を解決するために，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 を用いた粗粒度並列処理コードを自動生成する．生成された粗粒度並列処理コードでは，最早実行可能条件を用いて粗粒度タスクの実行管理とエンキューが行われており，粗粒度タスクのデキューは Fork/Join Framework のスケジューラが行う．NVIDIA Tegra K1 を搭載した Androidタブレット Shield 上で性能評価を行ったところ，Java Grande Forum Benchmark Suite の 4 つのプログラムに対して，4 スレッド実行の場合に逐次実行比で 3.0 倍から 3.8 倍の速度向上が得られ，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による並列処理は，スーパーコンピュータから Android タブレットや組み込みシステムに至るまで幅広く利用されている．マルチコアプロセッサによる並列処理において高い実効性能を実現するためには，ループ並列性の利用に加えて，ループやサブルーチン間の粗粒度並列性を引き出す粗粒度並列処理 [1], [2], [3] の利用が必要とされている．粗粒度並列処理の一手法として，異なる階層にまたがった粗粒度タスク間の並列性を利用する階層統合型実行制御手法 [4], [5] が提案されており，本研究の並列処理のベースとする．

High Performance Computing における並列処理の対象言語としては，従来より C/C++言語や Fortran 言語が多用されてきたが，プラットフォームフリーの Java 言語への関心が近年増加している [6]．これは Java 仮想マシン（JVM）の実行性能が，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．また，モバイル端末 OS として用いられる Android5.1.1 では，Java 処理系が JIT コンパイラ技術をベースとした Dalvik から，Ahead-Of-Time（AOT）コンパイラ技術をベースとした ART 処理系に変わっており，性能が向上している．

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[7] 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．Fork/Join Framework ではワークスティーリングを伴うスケジューラが導入されており，今後の普及が期待されている．しかしながら，Java Fork/Join Framework による並列処理は，再帰や分割統治等のプログラムへの適用が多く，さまざまな科学技術計算プログラムに対して粗粒度並列処理を実現することは困難であった．

一方，Java 拡張による並列処理の研究としては，HPF のような配列分散を取り入れた HPJava[8]，OpenMP/MPI のような共有メモリ・クラスタ並列プログラミングの API を提供する Parallel Java[9]，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[10] 等が提案されている．

本研究では，Java SE 7 以上に導入されている Fork/Join Framework 環境において，粗粒度並列処理を実現するためのタスク駆動型実行手法を提案し，その並列 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，Java Fork/Join Framework を用いたタスク駆動型実行により粗粒度並列処理を実現しており，NVIDIA Tegra K1 の Android タブ

[1] 明治大学総合数理学部
[2] ソフトバンク株式会社
[a)] akimasay@meiji.ac.jp

Vol.2016-ARC-218 No.11
Vol.2016-SLDM174 No.11
2016/1/19

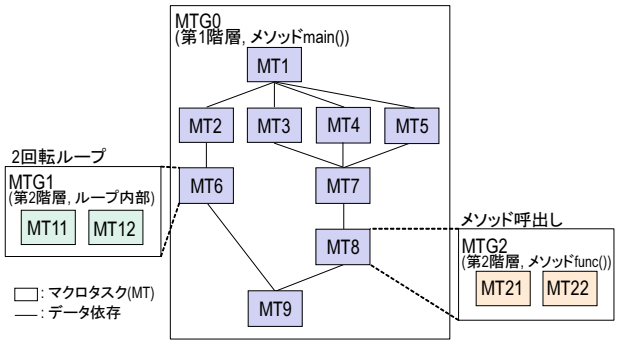

図 1 マクロタスクグラフ (MTG).

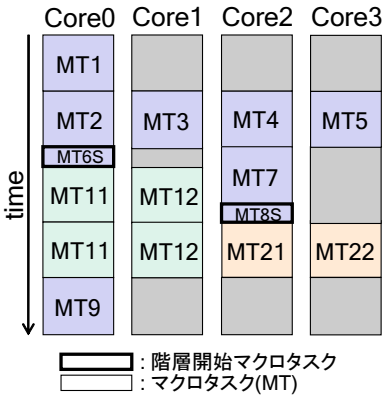

図 2 4 コアでのタスク駆動型実行の並列処理イメージ.

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

レットおよび組込みボードを用いて性能評価を行った.

本稿の構成は以下の通りとする．第 2 章では，タスク駆動型実行による粗粒度並列処理について述べる．第 3 章では，Fork/Join Framework を用いたタスク駆動型の粗粒度並列処理コードについて述べる．第 4 章では，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．第 5 章では，Android プラットフォームにおけるタスク駆動型実行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 [4], [5] 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 [4], [5] 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 [4] 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．

その後，提案するタスク駆動型実行においては，並列化コンパイラが生成した並列 Java コードによって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変化により実行可能になるマクロタスクを Fork する．但し，Fork されたマクロタスクは，直ちに実行される訳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，ワーカーキューの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．このとき必要に応じてワークスティーリングを行う．

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（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は**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は**図 2** のようになっており，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 のマクロタスク 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の 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，各コアのワーカースレッドがワーカーキューから MT2〜MT5 のマ

クロタスクをそれぞれ取り出して実行する．一方，MT3，MT4，MT5 の場合，各々の後続マクロタスク候補は MT7 であり，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 [4] 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，全階層のマクロタスク，即ち，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] を解析する．

例えば，表 1 の MT6 の最早実行可能条件は，MT2 の実行が終了した段階で，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としての終了（6S）を記録する．これにより，後続マクロタスク候補の 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 の実行後に終了（10）を記録し，MT10 の後続マクロタスク候補（MT11 と MT12）が実行可能になる．繰り返しブロックの繰り返し判定は，表 1 の MT13（mtCtrl）が行っており，繰り返しを継続する場合には，MT13（mtCtrl）が MT14（mtRepeat）に分岐し，繰り返しを終了する場合には，MT13（mtCtrl）が 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は，後続マクロタスク候補に対して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．

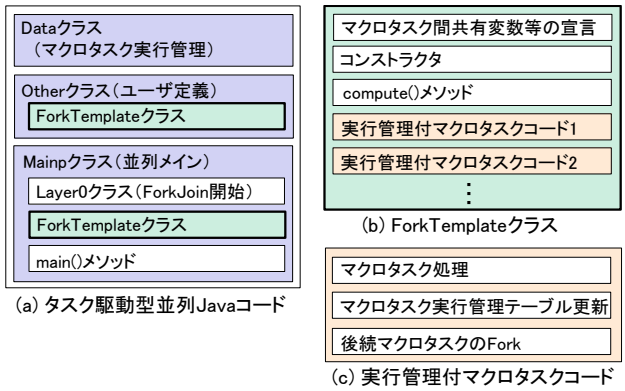

図 3 並列化コンパイラにより生成されるタスク駆動型並列 Java コードの構成．

3. Fork/Join Framework を用いたタスク駆動型の粗粒度並列処理コード

本章では，マクロタスク実行管理を伴うタスク駆動型の粗粒度並列処理コードの構成について述べる．

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，**図 3**(a) の構成をしている．本研究で生成する ForkTemplate クラスは，ソースプログラムのメソッドに対応しており，図 3(b) の構成をしている．この ForkTemplate クラス内部では，図 3(c) の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る．例えば，図 1 のマクロタスクグラフ（MTG）に対応した並列 Java コードの場合，Mainp クラス（並列メイン）は**図 4** のような形式をとる．

タスク駆動型並列 Java コードの実行では，Mainp クラス内に存在する main() メソッド（図 4 の 40〜44 行目）の処理が行われる．main() メソッドでは，スレッドプールを生成しワーカースレッド数を指定する．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 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 4 の 2〜11 行目）のインスタンスは，内部の compute() メソッドを実行する．これにより，Mainp クラス内部の ForkTemplate_main クラスのマクロタスク mtStart（図 4 の 20〜23 行目）が fork される．このとき，マクロタスク識別情報を基に，ForkTemplate_main クラスの compute() メソッド（図 4 の 17〜19 行目）を実行し，該当するマクロタスクの実行管理付マクロタスクコード（図 4 の 24〜38 行目）を実行する．

3.2 ForkTemplate クラス

ForkTemplate ベースのクラスは図 3(b) の構成をしており，並列化前のソースプログラムにおいて，並列化指示文

```
01: class Mainp { //並列メイン
02:     static class Layer0 extends RecursiveAction { //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のforkを行う;
08:             helpQuiesce()でタスク処理へ移行;
09:             joinを行う;
10:         }
11:     }
12:     public static class ForkTemplate_main extends RecursiveAction {
13:         マクロタスク間共有変数等の宣言;
14:         ForkTemplate_main(MT識別情報) { //コンストラクタ
15:             MT識別情報をフィールド変数に設定;
16:         }
17:         protected void compute() {
18:             該当するマクロタスクを実行;
19:         }
20:         public void mtStart(){ //mtStart
21:             マクロタスク実行管理テーブル更新;
22:             後続マクロタスクのforkを試みる;
23:         }
24:         public void mt1() { ... }
25:         public void mt2() { ... }
26:         ...
27:         public void mt13() { //mtCtrl
28:             繰り返し判定によりマクロタスク実行管理テーブル更新;
29:             mt14(mtRepeat)あるいはmt15(mtExit)のforkを行う;
30:         }
31:         public void mt14() { //mtRepeat
32:             マクロタスク実行管理テーブル更新;
33:             mt10(mtLoop)のforkを試みる;
34:         }
35:         public void mt15() { //mtExit
36:             マクロタスク実行管理テーブル更新;
37:             上位階層の後続マクロタスクのforkを試みる;
38:         }
39:     }
40:     public static void main(String[] args) {
41:         ForkJoinPool pool = new ForkJoinPool(ワーカースレッド数);
42:         Layer0 layer0 = new Layer0();
43:         pool.invoke(layer0); //ForkJoin開始
44:     }
45: }
```

図 4 タスク駆動型並列 Java コードの Mainp クラス（並列メイン）.

を含むメソッド毎に生成される．即ち，複数のメソッドが存在する場合には，各メソッドに対応した ForkTemplate クラスが生成される．そのメソッド内のマクロタスクの実行は，ForkTemplate ベースのクラスのインスタンスを Fork することにより実現される．

3.3 実行管理付マクロタスクコード

ForkTemplate ベースのクラスに含まれる実行管理付マクロタスクコードは，各マクロタスクに対応しており，図 3(c) のような構成をしている．このコードを実行すると，まず，マクロタスク処理が行われ，マクロタスク処理の終了後，マクロタスク実行管理テーブル（終了・分岐）を更新する．次に，後続マクロタスク候補の最早実行可能条件が満たされた場合に，その後続マクロタスク候補を Fork する．

4. タスク駆動型実行用の並列化コンパイラ

本章では，Fork/Join Framework を用いた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

本研究で開発した並列化コンパイラ [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，マクロタスクの最早実行可能条件 [4] を求めた後，タスク駆動型実行の並列 Java コードを生成する．

表 2 性能評価環境.

マシン	タブレット Shield	組込みボード JetsonTK1
プロセッサ	NVIDIA Tegra K1 2.3GHz	
CPU コア	Quad-Core ARM Cortex-A15 ”r3”	
メモリ	2GB	
OS	Android5.1.1	Ubuntu14.04
Java 処理系	ART	JVM1.7

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，本手法では入力対象となる Java プログラムにおいて，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，マクロタスク（MT）の定義は必須であり，`/*mt fork*/`のような並列化指示文を記述する．また，繰り返し文やクラスメソッド等のマクロタスク内部において，サブマクロタスクを階層的に定義する場合には，`/*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 decomp=`分割数`*/`のような並列化指示文を付加することにより，指定された分割数にループ分割され，それぞれがマクロタスクとして定義される．

5. Android プラットフォームでのタスク駆動型実行による粗粒度並列処理の性能評価

本章では，**表 2** に示すタブレット NVIDIA Shield と組込みボード NVIDIA JetsonTK1 において，タスク駆動型実行による粗粒度並列処理の性能評価を行う．

5.1 Java Grande Forum Benchmark Suite

本性能評価では，Java Grande Forum Benchmark Suite Version 2.0[12] より，**表 3** に示す 4 つの Java プログラムを用いて性能評価を行う．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．提案するタスク駆動型実行による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，タスク駆動型並列 Java コードを生成する．

5.2 タブレット NVIDIA Shield による性能評価

本節では，Android プラットフォームにおけるタスク駆

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

特性	Crypt	Series	MolDyn	MonteCarlo
プログラムの種類	暗号化処理	フーリエ級数	原子間相互作用	モンテカルロ法
データセット	C(N=5000 万)	B(N=10 万)	B(N=8788)	A(N=1 万)
並列化対象のソースコード長	308	505	561	553
タスク駆動型並列 Java コード長	1,823	919	4,695	1,073
並列化対象外のソースコード長 (ファイル数)	—	—	—	2,585(11)
並列化指示文数	5	6	23	5
ループ分割数	8	8	13	8
タブレット Shield の逐次実行時間 [ms]	28,094	51,446	183,448	8,450
組込みボード JetsonTK1 の逐次実行時間 [ms]	25,513	57,532	77,097	5,618

動型並列 Java コードの実行方法と，タブレット Shield 上での性能評価について述べる．

5.2.1 Android プラットフォームにおける実行方法

性能評価に用いる表 2 のタブレット NVIDIA Shield は，Tegra K1 プロセッサを搭載しており，OS は Android 5.1.1 (Lollipop，API レベル 22)，Java 処理系は ART ランタイムが用いられている．並列 Java コードの Build と Android 端末への Load には，Android Studio 1.3 を用いている．

Android Studio の Build により，java ファイルは，class ファイルを経て，dex（Dalvik Executable）ファイルに変換される．ART ランタイムの Android 端末では，AOT（Ahead-Of-Time）コンパイル方式が採用されており，dex ファイルは Android 端末に Load する際に，ネイティブコードに変換される．これにより，ART ランタイムの Android 端末では，ネイティブコードによる実行が可能になり，従来の Dalvik ランタイムの JIT に伴う処理時間を短縮することができる．

本性能評価では，Android Studio のプロジェクトファイル群の MainActivity.java において，onClick() メソッドとスレッド数選択を可能にしたテンプレートコードを用意する．次に，評価プログラムごとに，このテンプレートコードに，並列化コンパイラで生成した並列 Java コードを埋め込むことにより，Android 端末上で並列処理を行うことが可能になる．

5.2.2 タスク駆動型実行による粗粒度並列処理の性能評価

本節では，Android タブレット Shield 上で，表 3 の 4 種類のベンチマークプログラムを性能評価した結果を示す．逐次実行時間は表 3 の通りであり，並列処理による速度向上率（逐次実行比）は**図 5** に示されている．以下に個々のプログラムの性能について説明する．

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り，308 行のコード長となる．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タスク駆動型並列 Java コードは，ループ分割数=8 でループ分割が行われており，コード長は 1823 行となっている．このプログラムの逐次処理時間は，表 3 に示す通り 28,094[ms] となる．タスク駆動型並列 Java コードによる実行結果は図 5 に示すように，4 スレッド実行では逐次実行比で 3.62 倍の速度向上となっている．なお，実行の際には largeHeap を用いている．

Series はフーリエ級数を求めるプログラムであり，505 行のソースコードからなる．タスク駆動型並列 Java コードによる実行結果は，図 5 に示すように，4 スレッド実行では逐次実行比で 3.76 倍の速度向上となっている．

MolDyn はアルゴン原子の相互作用をモデル化した N 体問題のプログラムであり，561 行のソースコードからなる．タスク駆動型並列 Java コードによる実行結果は，図 5 に示すように，4 スレッド実行では逐次実行比で 3.11 倍の速度向上となっている．

MonteCarlo はモンテカルロ法による金融シミュレーションのプログラムであり，並列化対象部分は 553 行のソースコードからなり，並列化対象外のソースコードは 2,585 行 (11 ファイルは MainActivity.java と同じフォルダに配置する）になる．本並列化コンパイラでは，並列化対象部分のソースコードに対して 1,073 行のタスク駆動型並列 Java コードを生成する．タスク駆動型並列 Java コードによる実行結果は，図 5 に示すように，4 スレッド実行では逐次実行比で 2.96 倍の速度向上となっている．

以上の結果から，本並列化コンパイラで生成したタスク駆動型並列 Java コードを用いて，Android タブレット Shield 上で並列処理を行った場合，全てのプログラムにおいて，2.96 倍〜3.76 倍の速度向上が得られた．これによりタスク駆動型並列 Java コードは，Android プラットフォーム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．

5.3 組込みボード NVIDIA JetsonTK1 による性能評価

次に，Android タブレットと同様のプロセッサ NVIDIA Tegra K1 を搭載した組込みボード JetsonTK1 を用いて性能評価を行う．この組込みボードは，表 2 に示すように，Ubuntu14.04 と JVM1.7 がインストールされており，並列化コンパイラで生成されたタスク駆動型並列 Java コードを，HotSpot 最適化を伴う JVM 上で並列処理することができる．

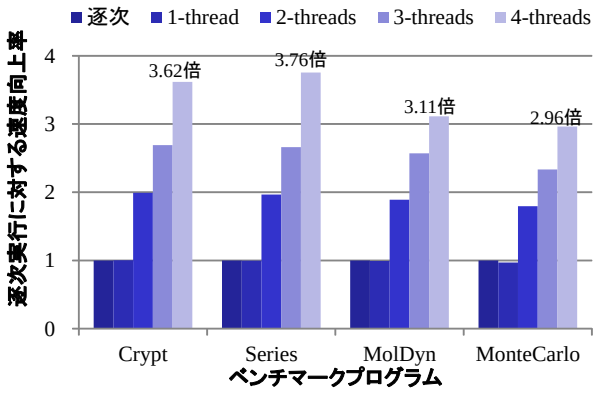

図 5 Android タブレット Shield による並列処理.

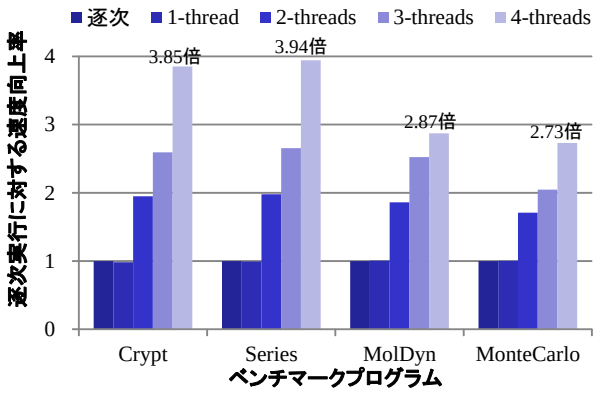

図 6 組込みボード JetsonTK1 による並列処理.

組込みボードの逐次実行時間は表 3 に示す通りであり，概ねタブレット Shield と同等であるが，MolDyn の場合はタブレット Shield より実行時間が短くなっている．これは Java 処理系の違いによるものと思われる．

一方，並列処理による速度向上率（逐次実行比）は**図 6** に示されており，並列化コンパイラで生成したタスク駆動型並列 Java コードを用いて，JetsonTK1 上で並列処理を行った場合，全てのプログラムにおいて 2.73 倍〜3.94 倍の速度向上が得られている．

これらの結果から，提案するタスク駆動型並列 Java コードによる粗粒度並列処理は，図 5 のタブレット Shield，及び，図 6 の組込みボード JetsonTK1 の両方において，高い実効性能が得られており，その有効性が確認された．

6. おわりに

本稿では，マルチコア上での Java Fork/Join Framework 環境において，粗粒度並列処理を実現するためのタスク駆動型並列 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 を用いたタスク駆動型並列 Java コードを容易に生成することができる．生成されたタスク駆動型並列 Java コードには，ForkTemplate ベースのクラスが導入されており，マクロタスク実行管理を行いつつ，Fork/Join Framework のスケジューラを用いて粗粒度並列処理を効率よ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，タスク駆動型並列 Java コードを生成した．Android タブレット Shield の 4 スレッド実行では 3.0 倍から 3.8 倍の速度向上，組込みボード JetsonTK1 の 4 スレッド実行では 2.7 倍から 3.9 倍の速度向上が得られた．

以上の結果から，Android プラットフォームにおいて，Java Fork/Join Framework を用いたタスク駆動型実行による粗粒度並列処理の有効性が確認された．

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] Mase, M., Onozaki, Y., Kimura, K. and Kasahara, H.: Parallelizable C and Its Performance on Low Power High Performance Multicore Processors, *Proc. of 15th Workshop on Compilers for Parallel Computing* (2010).
[3] Thies, W., Chandrasekhar, V. and Amarasinghe, S.: A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs, *Proc. IEEE/ACM Int. Symposium on Microarchitecture*, pp. 356–368 (2007).
[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, pp. 2732–2740 (2004).
[5] 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コンピューティングシステム， Vol. 7, No. 4, pp. 56–66 (2014).
[6] 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).
[7] Lea, D.: A Java Fork/Join Framework, *Proc. ACM conference on Java Grande, JAVA'00*, pp. 36–43 (2000).
[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] Raman, R., Zhao, J., Budimlic, Z. and Sarkar, V.: Compiler Support for Work-Stealing Parallel Runtime Systems, *Rice Technical Report TR10-02* (2010).
[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. 6, pp. 1–10 (2015).
[12] EPCC: The Java Grande Forum Benchmark Suite, *http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/* (2014).