 Vol.2018-ARC-232 No.24
2018/8/1

メニーコア上でのローカルタスク協調実行を伴うJavaプログラムのタスク駆動型粗粒度並列処理

岡 宏樹[1]　吉田 明正[1,2,a)]

概要：マルチコア上での Java 並列処理環境として Fork/Join Framework が導入されており，メニーコア上においても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，マルチコア上でマクロタスクレベルの並列処理を実現するタスク駆動型粗粒度並列処理が提案されている．本稿では，従来のタスク駆動型粗粒度並列処理によるマクロタスクレベルの並列性利用に加えて，マクロタスク内部のイタレーション集合あるいは再帰メソッド呼び出し文のようなローカルタスクレベルの並列性利用を可能にするタスク駆動型粗粒度並列処理手法を提案する．本手法では，開発した並列化コンパイラにより，ローカルタスク協調実行を伴うタスク駆動型粗粒度並列処理コードを自動生成する．メニーコアプロセッサである Intel Xeon Phi 上でベンチマークプログラムによる性能評価を行ったところ，68 コア実行において逐次実行比で最大 106 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が一般的となった現在，更なる並列性を利用できるプラットフォームとしてメニーコアプロセッサが期待されている．メニー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，ループ並列処理 [1] 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 [2], [3], [4], [5] を利用した高い並列性が必要不可欠である．

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 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．

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．

粗粒度並列処理 [2] では，粗粒度タスク間の並列性を抽出し，階層型マクロタスクグラフを生成する．その後，各階層の粗粒度タスクを，グルーピングしたコアに階層ごとに割り当てて並列処理を行う．一方，粗粒度並列処理の階層型マクロタスクグラフ [2] を利用しつつ，対象プログラムの各階層に存在する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の並列性を最大限に利用する階層統合型実行制御手法 [8], [9] が提案されている．

階層統合型粗粒度並列処理ではプログラムの並列性を最大限に利用した並列処理が可能となる一方，タスク数の増加に伴うスケジューリングオーバーヘッドの増加や，再帰メソッド内部の並列性が利用できない問題があった．特に，メニーコアプロセッサの性能を最大限に利用するためには高い並列性が必要となるため，オーバーヘッドの増加によって処理速度が大幅に低下する場合がある．

本稿では，Java Fork/Join Framework を用いたタスク駆動型のマクロタスク並列処理 [10] において，ローカルタスク協調実行を伴う実行手法を提案し，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．本手法ではマクロタスク内部のイタレーション集合，または再帰メソッド呼び出しをローカルタスクと定義し，マクロタスクのスケジューラから独立して実行する．これにより，スケジューリングによるオーバーヘッドの削減が可能となり，メニーコア環境においても高い実行性能を達成することができる．また，再帰メソッド呼び出しの並列化が可能となるため，幅広いプログラムにおいて並列性を最大限に引き出すことが可能となる．

[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

© 2018 Information Processing Society of Japan

本稿の構成は以下の通りとする．第 2 章では，タスク駆動型粗粒度並列処理について述べる．第 3 章では，Java Fork/Join Framework によるローカルタスク実行について述べる．第 4 章では，Fork/Join Framework によるタスク駆動型の粗粒度並列処理について述べる．第 5 章では，並列化コンパイラによる並列コード生成について述べる．第 6 章では，メニーコアプロセッサ上でローカルタスク協調実行を伴うタスク駆動型粗粒度並列処理の性能評価について述べる．第 7 章でまとめを述べる．

2. タスク駆動型粗粒度並列処理

本章では，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実行手法について述べる．

2.1 タスク駆動型実行の概念

タスク駆動型実行とは，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，MT ）実行手法である．

タスク駆動型実行において，マクロタスクの定義およびマクロタスク間の並列性抽出については，粗粒度並列処理のための階層統合型実行制御 [8], [9] を採用する．階層統合型実行制御では，階層型マクロタスクグラフ (MTG)[2] を生成し，マクロタスクを階層的に定義する．その後，最早実行可能条件 [8] を満たしたマクロタスクに対して，並列化コンパイラが生成したダイナミックスケジューラがコアに割り当てを行う．これに対してタスク駆動型実行では，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状態を管理し，Fork/Join Framework のスケジューラが，マクロタスクをワーカースレッドに割り当てる方式をとる．

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（制御用マクロタスクは図示していない）に対応しており，各マクロタスクの最早実行可能条件は，**表 1** に示される．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 3** のようになっており，マクロタスクおよびローカル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．ここで，図 2 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の MT2, MT3 に対して，最早実行可能条件の判定が行われ，MT2, MT3 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，MT3 をそれぞれ取り出して実行する．同様に MT2，MT3 の中で最後に終了したマクロタスクが，MT4，MT5，MT6 を fork してワーカーキューに投入する．

```
01: public class Main {
02:     ...
03:     /*recur*/
04:     int recurFunc(int n) {
05:         ...
06:         x = pre();
07:         if (n > 0) {
08:             /*recur fork(1)*/
09:             { y1 = recurFunc(n-1); }
10:             /*recur fork(2)*/
11:             { y2 = recurFunc(n-2); }
12:             /*recur join(1)*/
13:             /*recur join(2)*/
14:             y = y1 + y2;
15:         }
16:         else {
17:             y = 0;
18:         }
19:         z = post();
20:         return x + y + z;
21:     }
22:     public static void main(String[] args) {
23:         ...
24:         /*mt fork*/
25:         { func1(); }
26:         /*mt fork (1 1)*/
27:         { func2(); }
28:         /*mt fork (1 1)*/
29:         { func3(); }
30:         /*mt fork (1 2)&(1 3)*/
31:         { res = recurFunc(N); }
32:         /*mt fork (1 2)&(1 3)*/
33:         { func5(); }
34:         /*mt fork fdecomp=8 private(i) (1 2)&(1 3)*/ {
35:             for (i=0; i<M; i++) {
36:                 array[i] = f(i);
37:             }
38:         }
39:         /*mt fork (1 5)*/
40:         { func7(); }
41:         /*mt fork (1 4)&(1 6)&(1 7)*/
42:         { func8(); }
43:     }
44: }
```

図 1 並列化指示文付 Java プログラム

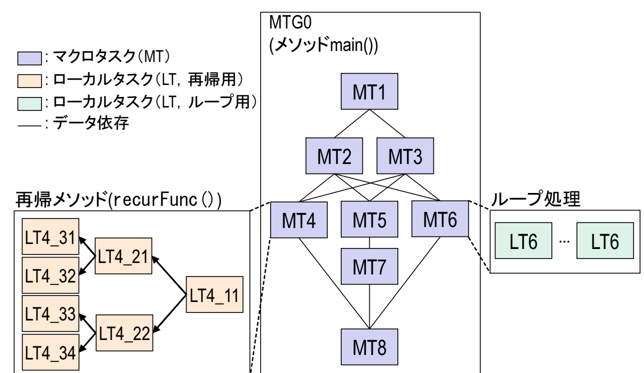

図 2 階層型マクロタスクグラフ (MTG)

2.2 階層的なマクロタスク定義と最早実行可能条件

階層統合型実行制御 [8] 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 文によるループ），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，第 1 階層マクロタスク内部に複数の

© 2018 Information Processing Society of Japan

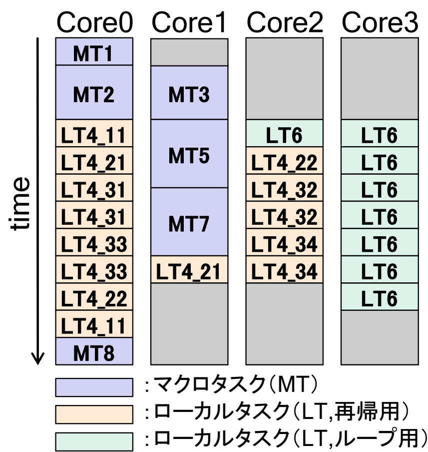

図 3　4 コアでのタスク駆動型実行の並列処理イメージ

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
	2	1	2	4,5,6
	3	1	3	4,5,6
	4	2∧3	4	8
	5	2∧3	5	7
	6	2∧3	6	8
	7	5	7	8
	8	4∧6∧7	8	End
	End†	8	—	—

†：制御用マクロタスク

サブマクロタスクを含む場合は，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とができるが，上位階層で十分な並列性が得られる場合には下位階層のマクロタスクは定義する必要がない．

2.3　タスク駆動型実行によるスケジューリング

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）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，ワークスティール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．ただし，本手法で導入するローカルタスクはマクロタスクと同様の実行管理は行われず，ワークスティールを伴うスケジューラによってのみ管理される．

表 2　Fork/Join Framework を用いた並列処理の実行方式

実行方式	LT 並列	MT 並列
通常の Fork/Join による実行	○	×
タスク駆動型並列処理（LT 協調なし）	×	○
タスク駆動型並列処理（LT 協調あり）	○	○

3.　Java Fork/Join Framework によるローカルタスク実行

本章では，Java Fork/Join Framework[7] と本稿で提案するローカルタスク実行について述べる．

3.1　Java Fork/Join Framework の概要

Java Fork/Join Framework[7] は，Java SE 7 以降に導入された 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により，小規模タスクレベルで複数コアによる並列処理を行うことが可能となる．

Fork/Join Framework では，初めにスレッド数を指定してスレッドプールを作成すると，その中に指定された数だけワーカースレッドが生成される．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キューに投入される．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() メソッドを実行する．

本フレームワークは，Thread クラスを用いた場合に比べて並列処理の記述が容易であり，ワークスティールを用いたスケジューラによって，大規模システムにおいてもロック競合の問題に対処することが可能である．

表 2 に示すように，通常の Fork/Join による実行はローカルタスクのみによる並列実行と同等であり，ローカルタスク協調実行を伴わない従来のタスク駆動型並列処理はマクロタスクのみを対象とした並列実行となっている．これに対し，本稿で提案するローカルタスク協調実行を伴うタスク駆動型並列処理は，ローカルタスクとマクロタスクの両方を対象とした並列実行となる．

3.2　ローカルタスク実行によるループ並列処理

従来のタスク駆動型並列処理では，分割された部分ループを含め，全てのタスクがマクロタスクとして定義されていた．分割された部分ループをマクロタスクとして定義し，メニーコアを対象とした並列化を行った場合，並列 Java コードのコード長が非常に長くなってしまうという問題がある．これにより，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い，実行時間が大幅に遅くなってしまう場合がある．

これに対して本稿で提案する手法では，分割された部分ループはローカルタスク実行により並列処理される．

© 2018 Information Processing Society of Japan

この場合，ループ並列処理の対象となるマクロタスクは，RecursiveAction クラスを継承したローカルタスクをループ分割数だけ fork し，ワーカーキューに直接投入される．ローカルタスクは Fork/Join スケジューラによってのみを管理されているため，分割数の増加に応じたマクロタスク実行管理によるオーバーヘッドの増加が発生しない．これにより，メニーコア環境のような高い並列性が要求される場合でも，高い実行性能を維持することが可能となる．

3.3 ローカルタスク実行による再帰メソッドの並列処理

マクロタスク管理による再帰メソッドの並列処理はオーバーヘッドが大きくなるため、提案するローカルタスクを用いて並列処理を実現する．再帰メソッドを呼び出すマクロタスクは，RecursiveAction クラス，または RecursiveTask クラスを継承したローカルタスクを fork し，メソッドの処理を行う．ローカルタスク内で再び再帰メソッドが呼び出される際には，改めて fork が行われる．ローカルタスクの管理はループ並列処理の場合と同様に Fork/Join スケジューラによって行われる．

また，本手法のよる再帰メソッドの fork には条件を付けることが可能となっている．この条件を満たさない場合には，通常の再帰メソッドと同様に同一スレッド上でメソッドの再帰呼び出しが行われ，それ以降の再帰呼び出しでは fork は行われなくなる．これにより，fork によるオーバーヘッドを最小限に留めることができる．

4. Fork/Join Framework によるタスク駆動型粗粒度並列処理

本稿で提案するタスク駆動型実行では，マクロタスク処理とマクロタスク実行管理は後述する ForkTemplate_main クラスにより実装し，最早実行可能条件を伴って管理される．また，ワーカーキューからのマクロタスクの実行については，Java Fork/Join Framework のワークスティーリングを伴う Fork/Join スケジューラを利用している．これに対し，ローカルタスクはマクロタスク実行管理とは独立しており，Fork/Join スケジューラによってのみ管理される．本章では，このようなタスク駆動型の粗粒度並列処理コードの生成手法について述べる．

4.1 タスク駆動型粗粒度並列処理の並列 Java コード

図 4 は，図 1 の Java プログラムに対して生成されるローカルタスク協調実行を伴うタスク駆動型並列 Java コードである．

このプログラムでは初めに，Mainp クラス内の main() メソッド（図 4 の 59〜63 行目）の処理が行われる．main() メソッドでは，ワーカースレッド数を指定してスレッドプールを生成する．ワーカースレッド数は，プログラムの実行時にコマンドライン引数として指定する．その後，

```
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 RecursiveTask<Void> {
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:         public void mt3() { ... }
27:         public void mt4() { res = recurFunc(N); ... }
28:         public void mt5() { ... }
29:         public void mt6() { Mt6.execute(); ... }
30:         static class Mt6 extends RecursiveTask<Void> {
31:             static void execute() {
32:                 分割数に従ってMt6をfork;
33:             }
34:             private Mt6(int forkId) { // コンストラクタ
35:                 forkIdをフィールド変数に設定;
36:             }
37:             protected Void compute() { ... }
38:         }
39:         public void mt7() { ... }
40:         public void mt8() { ... }
41:     }
42:     static int recurFunc(int n) { RecurFuncをfork; }
43:     private static class RecurFunc extends RecursiveTask<Integer> {
44:         RecurFunc(int n) { //コンストラクタ
45:             nをフィールド変数に設定;
46:         }
47:         protected Integer compute() {
48:             ...
49:             if (n > 0) {
50:                 ForkJoinTask<Integer> recurTask1 = new RecurFunc(n-1).fork();
51:                 ForkJoinTask<Integer> recurTask2 = new RecurFunc(n-2).fork();
52:                 y1 = recurTask1.join();
53:                 y2 = recurTask2.join();
54:                 y = y1 + y2;
55:             }
56:             ...
57:         }
58:     }
59:     public static void main(String[] args) {
60:         ForkJoinPool pool = new ForkJoinPool(ワーカースレッド数);
61:         Layer0 layer0 = new Layer0();
62:         pool.invoke(layer0); //ForkJoin開始
63:     }
64: }
```

図 4 ローカルタスク協調実行を伴うタスク駆動型並列 Java コード

Layer0 クラスのインスタンスを invoke() メソッドにより呼び出すことで，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，実行される．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きマクロタスクコード（図 4 の 24〜40 行目）を実行する．

4.2 ローカルタスク協調実行による並列ループコード

並列ループとして処理されるマクロタスク mt6() メソッド（図 4 の 29 行目）が呼び出されると，Mt6 クラス（図 4 の 29〜38 行目）の execute() メソッドが実行される．これにより，Mt6 クラスのインスタンスがループの分割数だけ

```
01: /*recur fork(1) if(n-1 > 10)*/ {
02:     y1 = recurFunc(n-1);
03: }
04: /*recur fork(2) if(n-2 > 10)*/ {
05:     y2 = recurFunc(n-2);
06: }
```

(a) 並列指示文付Javaプログラム（一部）

```
01: boolean isRecurTask1Forked = (n-1 > 10);
02: ForkJoinTask<Integer> recurTask1 = null;
03: if (isRecurTask1Forked)
04:     recurTask1 = new RecurFib(n-1).fork();
05: ...
06: if (isRecurTask1Forked)
07:     y1 = recurTask1.join();
08: else
09:     y1 = selfCompute(n-1);
10: ...
```

(b) ローカルタスク協調を伴うタスク駆動型並列Javaコード（一部）

図 5 条件付きで fork する場合の並列化指示文付 Java プログラムと並列 Java コード

生成され，それぞれ fork される．このとき，forkId として 0〜分割数-1 がコンストラクタの引数として渡される．fork されたインスタンスは，内部の compute() メソッドを実行する．compute() メソッド内部ではループ処理が行われるが，forkId によってそれぞれのインスタンスは異なるループ範囲を処理する．

4.3 ローカルタスク協調実行による再帰メソッドコード

再帰メソッドを含むマクロタスク mt4() メソッド（図 4 の 27 行目）が呼び出されると，recurFunc() メソッド（図 4 の 42 行目）が実行される．これにより，RecurFunc クラス（図 4 の 43〜58 行目）のインスタンスが生成され，fork される．fork されたインスタンスは，内部の compute() メソッドを実行する．compute() メソッド内部では本来の再帰メソッドの処理が行われるが，再帰的にメソッド呼び出しを行う場合には，自身のインスタンスを再度生成し，fork する（図 4 の 50，51 行目）．

また，再帰メソッドの fork に条件を付ける場合には，並列指示文付 Java プログラムにおいて，図 1 の 8〜11 行目を図 **5**(a) のように書き換えることができる．この場合，生成される並列 Java コードにおいて，図 4 の 50〜53 行目が図 5(b) のように変更される．これにより，指示文で記述された条件に従って fork が行われ，条件を満たさない場合には selfCompute() メソッドによって同一スレッド上で再帰メソッドが実行される．

5. 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自動生成する並列化コンパイラについて述べる．

5.1 並列化コンパイラの仕様と実行手順

本研究で開発した並列化コンパイラは，並列化指示文を加えた Java プログラムを入力ソースファイルとして，Java Fork/Join Framework を利用したタスク駆動型の粗粒度並列処理コード（並列 Java コード）を出力する．入力対象となる Java プログラムは，字句解析・構文解析が対応している J2SE 1.2 の文法で記述されているものとする．

複数ファイルからなる Java プログラムにも対応可能であるが，本並列化コンパイラでは並列化指示文を挿入した Java プログラムは 1 つのファイルにまとめておく仕様となっている．なお，並列化指示文を付加していない Java プログラムや class ファイルは，並列化コンパイラによってコンパイルする必要がないため，別ファイルのままで取り扱うことができる．ただし，並列化指示文を付加していない別ファイルのメソッドに対して，並列化対象となるマクロタスクからメソッド呼び出しが行われる可能性がある．そのような場合には，本並列化コンパイラではスレッドセーフなメソッドに予めリストラクチャリングされていることを前提とする．また，try-catch 文のような例外処理は逐次に実行されるが，科学技術計算等のプログラムでは大きな問題ではない．

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 [8] 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．

この並列 Java プログラムおよび並列化指示文を付加していない Java プログラムを通常の Java コンパイラでコンパイルすることにより，JVM 上で Fork/Join Framework を用いたタスク駆動型の粗粒度並列処理を実現することができる．

5.2 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，**表 3** の並列化指示文を記述し，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（例，図 1）．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．

また，入力対象の Java プログラムにおいて，粗粒度並列処理を適用しない部分（前処理部分や後処理部分）については，`/*premt*/`および`/*postmt*/`の並列化指示文を記述する．

並列化可能ループ（リダクションループを含む）は，

© 2018 Information Processing Society of Japan

表 3　並列化指示文

並列化指示文の表記	意味
/*mt fork 指示節 1*/	マクロタスクの定義
指示節 1 として，論理式	最早実行可能条件を論理式で指定
/*mt fork inner*/	内部でサブマクロタスクを定義しているマクロタスクの定義
指示節 1 として，decomp=分割数 指示節 2*/	指定した分割数にマクロタスクとしてループ分割
指示節 1 として，fdecomp=分割数 指示節 2*/	指定した分割数にローカルタスクとしてループ分割
指示節 2 として，private(変数名)	ループ分割における変数のプライベート化
指示節 2 として，reduction(リダクション演算子:変数名)*/	ループ分割におけるリダクション処理
/*recur*/	再帰メソッドの定義
/*recur fork(ID) 指示節 3*/	再帰メソッド呼び出し
/*recur join(ID)*/	再帰メソッドの join 位置の定義（同一 ID の fork を join する）
指示節 3 として，if(条件)	再帰メソッド呼び出しの fork 条件
/*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 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．このとき，private(変数名) によりプライベート変数の指定や，reduction(リダクション演算子:変数名) によりリダクション変数の指定ができる．

最早実行可能条件はメソッド内で自動的に求められるが，並列化指示文/*mt fork 論理式*/を記述することにより，より高度な並列性の利用が可能になる．

再帰メソッドは，/*recur*/のような並列化指示文を付加することにより，並列処理が可能な再帰メソッドとして定義される．再帰メソッド内部において，/*recur fork(ID)*/のような並列化指示文を付加することにより，fork される再帰メソッド呼び出しとして定義される．このとき，if(条件) により fork を行う条件を定義することができる．また，/*recur join(ID)*/により，fork された同一 ID の再帰メソッド呼び出しの join を行う位置を定義することができる．

5.3　並列化コンパイラの実装

本並列化コンパイラは 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．入力対象となる Java プログラムを並列化コンパイラへ入力すると，字句解析と構文解析が行われ，各ステートメントをノードとする抽象構文木が生成される．このとき，表 3 の並列化指示文により指定された部分はマクロタスクや再帰メソッドとして認識され，その情報がノードに保存される．

字句解析と構文解析で抽象構文木が作成されたあと，並列化指示文で定義されたマクロタスクにおいて変数の定義と使用を求め，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 のような最早実行可能条件を生成する．単一メソッド内のマクロタスクの最早実行可能条件は自動的に求められるが，本並列化コンパイラはプロトタイプであり，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，メソッド間で高度な並列性を引き出すためには，部分的に並列化指示文による最早実行可能条件の記述することが効果的である．

次に，タスク駆動型実行のために，各マクロタスクの最早実行可能条件から後続マクロタスク候補を解析する．最早実行可能条件と後続マクロタスク候補は，並列化コンパイラの生成する並列 Java コードに反映される．

本並列化コンパイラでは，並列化の解析が行われたあと，図 4 のような ForkTemplate ベースの並列 Java コードを生成する．生成された並列 Java プログラムと並列化指示文を含まない Java プログラム（別ファイル）に対して，通常の Java コンパイラでコンパイルしたあと，JVM 上で実行することが可能となる．

6.　メニーコア上での性能評価

本章では，ローカルタスク協調実行を伴うタスク駆動型粗粒度並列処理の性能評価について述べる．

6.1　性能評価環境

性能評価には，Intel Xeon Phi Processor 7250（68 コア，272 スレッド，1.40GHz）を搭載し，メモリは 48GB，OS には CentOS7.4 を導入し，Java 処理系は JDK1.8 のマシンを利用した．

6.2 節では，**表 4** に示す Java Grande Forum Benchmark Suite Version 2.0[11] から 4 つの Java プログラムを用いて性能評価を行う．6.3 節では，**表 5** に示す 2 つの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などのリストラクチャリングを予め行っており，表 3 の並列化指示文を加え，並列化指示文を含むソースプログラムは 1 ファイルにまとめている．

© 2018 Information Processing Society of Japan

表 4　Java Grande Forum Benchmark Suite の性能評価プログラム

プログラムの特性	Crypt	Series	MonteCarlo	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 [行]	308	508	553	448
並列化対象外のソースコード長 [行](ファイル数)	—	—	2,585(11)	998(12)
MTG 数	1	2	2	1
タスク駆動型並列 Java コード長（ローカルタスク協調なし）[行]	9,302	29,708	34,799	36,471
タスク駆動型並列 Java コード長（ローカルタスク協調あり）[行]	584	761	856	549
逐次実行時間 [ms]	6,289	135,668	7,528	37,722

表 5　再帰メソッドを含む性能評価プログラム

プログラムの特性	フィボナッチ	マージソート
並列化対象のソースコード長 [行]	64	95
並列化対象外のソースコード長 [行](ファイル数)	—	—
MTG 数	1	1
タスク駆動型並列 Java コード長（通常の Fork/Join のみ）[行]	408	333
タスク駆動型並列 Java コード長（ローカルタスク協調）[行]	415	607
逐次実行時間 [ms]	50,508	20,892

タスク駆動型実行を行う際には，開発した並列化コンパイラに並列化指示文を付加したソースプログラムを入力して，タスク駆動型並列 Java コードを生成し，Java コンパイラでコンパイルしてから JVM 上で実行する．

6.2 Java Grande Forum Benchmark Suite によるタスク駆動型粗粒度並列処理の性能評価

本節では，表 4 に示す Java Grande Forum Benchmark Suite Version 2.0[11] から 4 つの Java プログラムを用いて性能評価を行う．これらの 4 つのベンチマークプログラムはループ並列処理の対象となるループを含んでおり，それらに対して提案するローカルタスクを伴う並列処理を適用している．

Crypt は，IDEA(International Data Encryption Algorithm) と呼ばれる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 (encrypt) と復号化 (decrypt) を行うプログラムであり，308 行のソースコードからなる．本並列化コンパイラでタスク駆動型並列 Java コードを生成して実行した結果，**図 6** に示すように，従来のローカルタスク協調を用いない場合は最大で 64 スレッド実行の場合に逐次実行比で 3.79 倍の速度向上であるのに対し，ローカルタスク協調を用いた場合には最大で 64 スレッド実行の場合に逐次実行比で 7.06 倍の速度向上を達成している．また，表 4 に示すように，ローカルタスク協調を用いない従来の場合のタスク駆動型並列 Java コード長は 9,302 行であるのに対し，ローカルタスク協調を用いた場合のコード長は 584 行となっており，実行時間の短縮に寄与していると考えられる [12].

Series は，関数 $f(x) = (x+1)^x$ のフーリエ係数を求めるプログラムであり，508 行のソースコードからなる．本

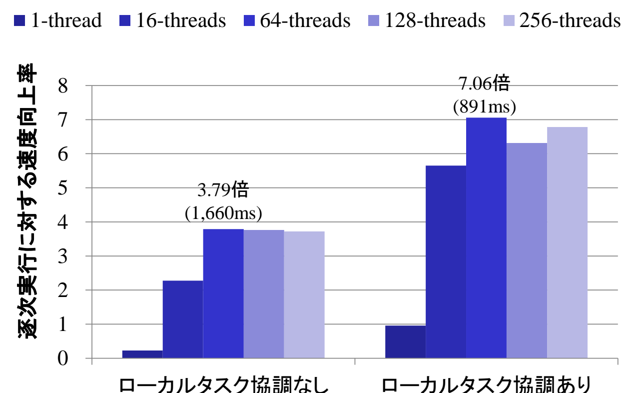

図 6　Crypt におけるタスク駆動型粗粒度並列処理

並列化コンパイラでタスク駆動型並列 Java コードを生成して実行した結果，**図 7** に示すように，ローカルタスク協調を用いない場合は最大で 128 スレッド実行の場合に逐次実行比で 52.3 倍の速度向上であるのに対し，ローカルタスク協調を用いた場合には最大で 256 スレッド実行の場合に逐次実行比で 106 倍の速度向上を達成している．また，表 4 に示すように，ローカルタスク協調を用いない場合のタスク駆動型並列 Java コード長は 29,708 行であるのに対し，ローカルタスク協調を用いた場合のコード長は 761 行となっている．

Monte Carlo はモンテカルロ法による金融シミュレーションのプログラムであり，553 行の並列化対象ソースコードと，2585 行（11 ファイル）の並列化対象外ソースコードからなる．本並列化コンパイラでタスク駆動型並列 Java コードを生成して実行した結果，**図 8** に示すように，ローカルタスク協調を用いない場合は最大で 16 スレッド実行の場合に逐次実行比で 6.91 倍の速度向上であるのに対し，

© 2018 Information Processing Society of Japan

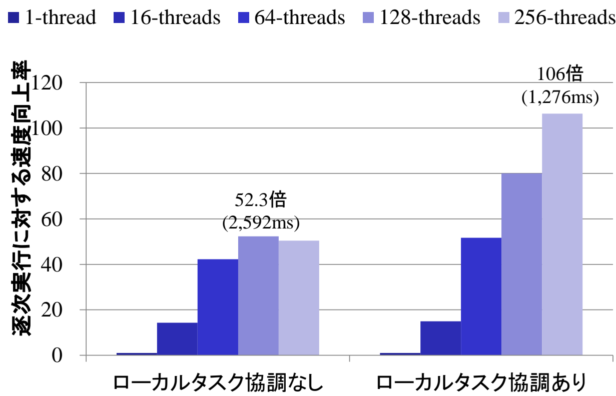

図 7　Series におけるタスク駆動型粗粒度並列処理

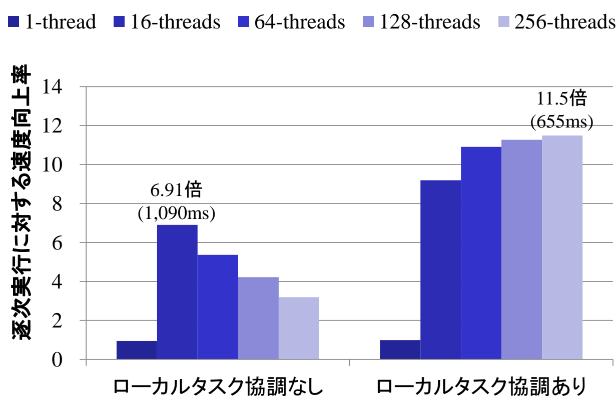

図 8　Monte Carlo におけるタスク駆動型粗粒度並列処理

ローカルタスク協調を用いた場合には最大で 256 スレッド実行の場合に逐次実行比で 11.5 倍の速度向上を達成している．また，表 4 に示すように，ローカルタスク協調を用いない場合のタスク駆動型並列 Java コード長は 34,799 行であるのに対し，ローカルタスク協調を用いた場合のコード長は 856 行となっている．

RayTracer は 3 次元の光線追跡法のプログラムであり，64 個の球を含むシーンがレンダリングされる．このプログラムは 448 行の並列化対象ソースコードと，998 行（12 ファイル）の並列化対象外ソースコードからなる．本並列化コンパイラでタスク駆動型並列 Java コードを生成して実行した結果，図 8 に示すように，ローカルタスク協調を用いない場合は最大で 64 スレッド実行の場合に逐次実行比で 15.7 倍の速度向上であるのに対し，ローカルタスク協調を用いた場合には最大で 256 スレッド実行の場合に逐次実行比で 24.7 倍の速度向上を達成している．また，表 4 に示すように，ローカルタスク協調を用いない場合のタスク駆動型並列 Java コード長は 36,471 行であるのに対し，ローカルタスク協調を用いた場合のコード長は 549 行となっている．

これらの 4 つのプログラムの性能評価の結果から，ローカルタスク実行によるループ並列処理を伴うタスク駆動型

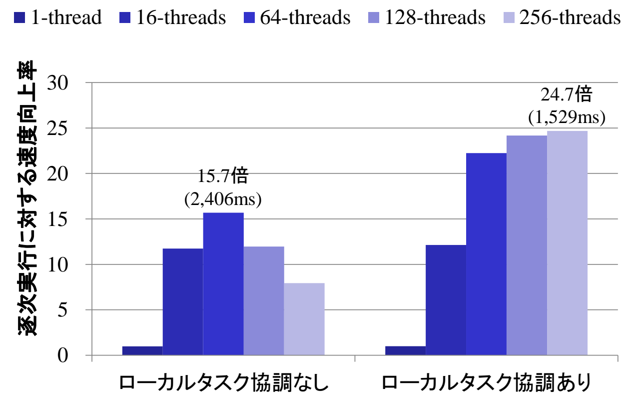

図 9　Raytracer におけるタスク駆動型粗粒度並列処理

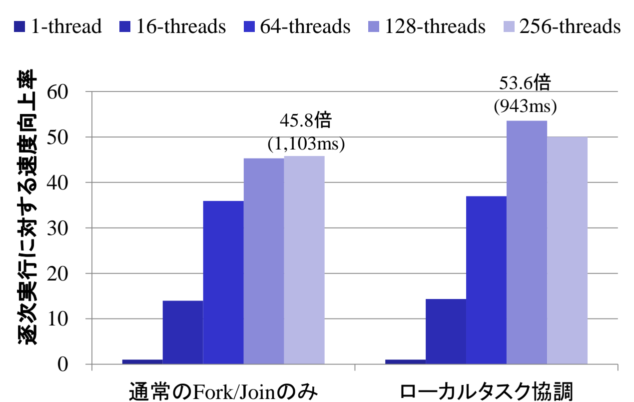

図 10　フィボナッチプログラムにおけるタスク駆動型粗粒度並列処理

粗粒度並列処理の有効性が確かめられた．

6.3　再帰プログラムによるタスク駆動型粗粒度並列処理の性能評価

本節では，表 5 に示す 2 つのプログラムを用いて性能評価を行う．これらの 2 つのプログラムは再帰メソッドを含んでおり，それらに対して提案するローカルタスクを伴う並列処理を適用している．

フィボナッチプログラムは，43 番目のフィボナッチ数を 8 回求めるプログラムであり，64 行のソースプログラムからなる．本並列化コンパイラでタスク駆動型並列 Java コードを生成して実行した結果，**図 10** に示すように，通常の Fork/Join のみを用いた場合は最大で 256 スレッド実行の場合に逐次実行比で 45.8 倍の速度向上であるのに対し，ローカルタスク協調を用いた場合には最大で 128 スレッド実行の場合に逐次実行比で 53.6 倍の速度向上を達成している．

マージソートプログラムは，サイズが 5000 万の配列をマージソートによってソートするプログラムであり，95 行のソースプログラムからなる．本並列化コンパイラでタスク駆動型並列 Java コードを生成して実行した結果，**図 11**

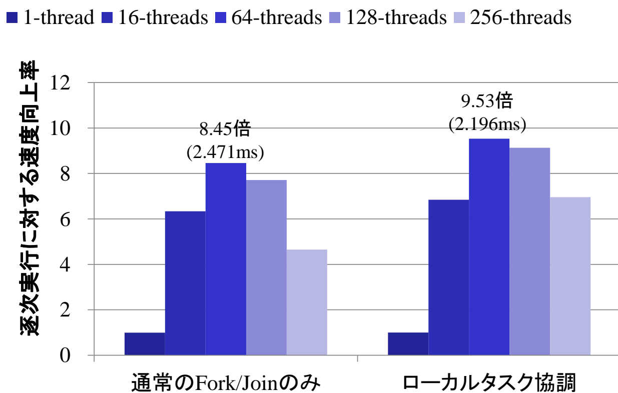

図 11 マージソートプログラムにおけるタスク駆動型粗粒度並列処理

に示すように，通常の Fork/Join のみを用いた場合は最大で 64 スレッド実行の場合に逐次実行比で 8.45 倍の速度向上であるのに対し，ローカルタスク協調を用いた場合には最大で 64 スレッド実行の場合に逐次実行比で 9.53 倍の速度向上を達成している．

これらの 2 つのプログラムの性能評価の結果から，表 2 に示すように，ローカルタスク並列のみでなく，マクロタスク並列を併用する提案手法の有効性が確かめられた．

7. おわりに

本稿では，Java Fork/Join Framework 環境における，メニーコア上でのローカルタスク協調実行を伴う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マクロタスクとローカルタスクの協調実行を伴う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プログラムを並列化コンパイラへ入力し，ローカルタスク協調実行を伴うタスク駆動型並列 Java コードを生成した．この並列 Java コードを，メニーコアプロセッサ Intel Xeon Phi Processor 7250 上で実行したところ，並列ループからなるローカルタスクを含むプログラムでは逐次実行比で最大 106 倍，再帰メソッドからなるローカルタスクを含むプログラムでは逐次実行比で最大 53.6 倍の速度向上が得られ，提案手法の有効性が確認された．

以上の結果から，Java Fork/Join Framework 環境におけるメニーコア上でのローカルタスク協調実行を伴うタスク駆動型実行による粗粒度並列処理の有効性，ならびに開発した並列化コンパイラの有用性が確認された．

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] Eigenmann, R., Hoeflinger, J. and Padua, D.: On the automatic parallelization of the Perfect benchmarks, *IEEE Trans. on Parallel and Distributed System*, Vol. 9, No. 1, pp. 5–23 (1998).

[2] 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).

[3] 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).

[4] Yoshida, A.: An Overlapping Task Assignment Scheme for Hierarchical Coarse Grain Task Parallel Processing, *Journal Concurrency and Computation: Practice and Experience*, Vol. 18, No. 11, pp. 1335–1351 (2006).

[5] 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).

[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] 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).

[9] 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).

[10] 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).

[11] EPCC: The Java Grande Forum Benchmark Suite, *http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/* (2014).

[12] 岡宏樹，吉田明正：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，研究報告システム・アーキテクチャ (ARC)，Vol. 2017-ARC-227, No. 38, pp. 1–7 (2017).

© 2018 Information Processing Society of Japan