1H-05

タスク駆動型粗粒度並列処理におけるコンパイラによる並列ライブラリ生成

Parallel Library Generation for Task-Driven Coarse Grain Parallel Processing by Compiler

山端 大揮 †　　吉田 明正 †‡
Daiki Yamahata　　Akimasa Yoshida

1 はじめに

メニーコア上での並列処理手法として，Java Fork/Join Framework[1] を利用したタスク駆動型粗粒度並列処理 [2] が提案されている．本手法では，並列化対象となるプログラムを階層的にマクロタスク (MT) に分割し，さらにマクロタスク内部をローカルタスク (LT) に分割する．ダイナミックスケジューリング環境下でマクロタスクとローカルタスクの協調実行を行うことにより，高い並列性を利用することができる [3].

近年，GPU を対象とした行列計算の cuBLAS ライブラリ [4] や，ベクトル化と SIMD 化を行った有限要素法のライブラリ [5] を用いて並列実行を実現する研究が行われている．本稿では，タスク駆動型粗粒度並列処理の並列コードを，開発した並列化コンパイラにより，並列ライブラリとして生成する方法を提案し，Intel Xeon Phi での性能評価から，提案手法の有効性を確認する．

2 タスク駆動型粗粒度並列処理

タスク駆動型粗粒度並列処理 [2] とは，入力対象となる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および制御依存を解析し，最早実行可能条件を求めることによって，マクロタスク間の並列性を引き出す並列実行手法である．

2.1 Fork/Join Framework によるタスク駆動型実行

本並列処理手法は，Java Fork/Join Framework 環境 [1] において，実行中のマクロタスクの終了状態・分岐状態を管理し，そのマクロタスクの状態変化から，次に実行可能となるマクロタスクを fork する．その後，fork されたマクロタスクはワーカーキューに投入され，Fork/Join Framework のスケジューラによりワーカーキューから取り出され，ワーカースレッドで実行される．

2.2 ローカルタスク協調実行による並列処理

本並列処理手法では，対象プログラムにおいて分割されたマクロタスク，サブマクロタスク，ローカルタスクを低オーバーヘッドで協調実行ができ，高い並列性を得ることができる [3]．本手法における例として，図 1 のマクロタスクグラフを 4 コア上で実行したイメージは図 2 のようになる．また，図 1 のマクロタスクグラフを表すタスク駆動型並列コードは図 3 として表される．

3 並列化コンパイラ

並列化コンパイラは，並列化指示文を付加した Java プログラムを入力対象として，Java Fork/Join Framework を利用したタスク駆動型並列コードを出力し [2]，シェルスクリプトを使って並列コードから並列ライブラリとなる jar ファイルを生成する．図 4 は，本コンパイラの構成を示す．

† 明治大学大学院先端数理科学研究科
Graduate School of Advanced Mathematical Sciences, Meiji University
‡ 明治大学総合数理学部ネットワークデザイン学科
Department of Network Design, School of Interdisciplinary Mathematical Sciences, Meiji University

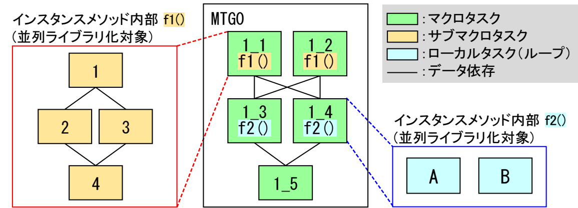

図 1　階層型マクロタスクグラフ.

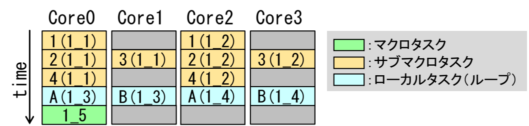

図 2　図 1 の 4 コアでの並列処理イメージ.

3.1 並列化コンパイラの実装

本並列化コンパイラは Java 言語で開発されており，LALR(1) のパーサジェネレータである JFlex と Jay を，それぞれ字句解析と構文解析にて用いている．例として，表 1 に示すような並列化指示文が付加された図 5 の Java コードを並列化コンパイラへ入力すると，まず字句解析と構文解析が行われて抽象構文木が作成される．その後，並列化指示文で定義されたマクロタスク間のデータ依存と制御依存を解析し，最早実行可能条件を生成し，後続マクロタスク候補を解析することによって，図 1 に示すマクロタスクグラフを構成する．以上の並列化の解析が行われた後，図 3 の並列 Java コードを生成する．

3.2 タスク駆動型実行コードの並列ライブラリ化

並列ライブラリ化については，まず生成された図 3 のような並列コードから，コンパイルにより個々のクラスファイルを生成する．次に，並列化されたタスク集合のクラスファイルを対象として，jar コマンドによるファイル圧縮を行い，並列ライブラリとなる jar ファイルを生成する．例えば，図 3 ではサブマクロタスク集合の Smt_p(27〜43 行目) とローカルタスク集合の Lt_p(44〜48 行目) のそれぞれが並列ライブラリ化される．

プログラムの実行時には，main メソッド含むクラスファイルと，生成した jar ファイルを用いることにより，タスク駆動型実行が可能となる．

4 Intel Xeon 上での粗粒度並列処理の性能評価

本研究では，メニーコアプロセッサ搭載マシン Intel Xeon Phi Knights Landing Server を用いて性能評価を行った．本マシンでは，搭載プロセッサが Intel Xeon Phi Processor 7250 (1.40GHz，68 コア)，メモリが 48GB，OS が CentOS7.4，Java 処理系が JDK1.8 で構成される．

性能評価プログラムは，共通鍵暗号方式 AES プログラム [6] を用いた．本プログラムでは，4 種類の 16[MiB] の入力文を用いて処理を行う．

本プログラムにおいて，第 1 階層においては，4 つのマクロタスクを用いて 4 つの入力文の並列実行を行う．そのマクロタスクの内部の第 2 階層では，各入力文の暗号化を行うサブマクロタスクと，復号化を行うサブマク

Copyright ©2021 Information Processing Society of Japan.
All Rights Reserved.

表 1　並列化指示文.

並列化指示文の表記	意味
/*mt fork Instance(クラス名:インスタンス名) 論理式*/	インスタンスメソッド内部でサブマクロタスク集合およびローカルタスク集合を構成しているマクロタスクの定義
/*mtset*/	サブマクロタスク集合を構成するインスタンスメソッドの定義
/*smt fork Instance(クラス名:インスタンス名) 論理式*/	インスタンスメソッド内部でサブマクロタスク集合およびローカルタスク集合を構成しているサブマクロタスクの定義
/*ltset*/	ローカルタスク集合を構成するインスタンスメソッドの定義
/*lt fork fdecomp=分割数 private(変数名)*/	指定した分割数にローカルタスクとしてループ分割 ループ分割における変数は private(変数名) でプライベート化

```
01: class Data_Sample {}  // Sample_pクラスのマクロタスク管理クラス
02: class Sample_p {
03:   public static void main(String[] args) {
04:     ForkJoinPool pool = new ForkJoinPool(ワーカースレッド数);
05:     Layer0 layer0 = new Layer0();
06:     pool.invoke(layer0); //ForkJoin開始
07:   }
08:   static class Layer0 extends RecursiveAction { //ForkJoin開始
09:     Layer0() { Data_Sampleクラスのフィールド変数の初期化; }
10:     protected void compute() {
11:       Main0クラスのmtStartのforkを行う;
12:       helpQuiesce()でタスク処理へ移行;
13:       joinを行う;
14:     }
15:   }
16:   public static class Main0 extends RecursiveAction {
17:     Main0(マクロタスク番号) { }  //コンストラクタ
18:     protected void compute() { 該当するマクロタスクを実行; }
19:     public void mtStart() { }  //開始タスク
20:     public void mt1_1() { Smt_p s1 = new Smt_p();  s1.f1(); }
21:     public void mt1_2() { Smt_p s2 = new Smt_p();  s2.f1(); }
22:     public void mt1_3() { Lt_p  l3 = new Lt_p();   l3.f2(); }
23:     public void mt1_4() { Lt_p  l4 = new Lt_p();   l4.f2(); }
24:     public void mt1_5() { System.out.println("mt1_5"); }
25:   }
26: }
27: class Data_Smt {}  // Smt_pクラスのサブマクロタスク管理クラス
28: class Smt_p {
29:   public void f1() { Layer1 layer1 = new Layer1();   layer1.invoke(); }
30:   class Layer1 extends RecursiveAction {
31:     Layer1() { Data_Smtのフィールド変数の初期化; }
32:     protected void compute() { Main1クラスのsmtStartのforkを行う; }
33:   }
34:   public class Main1 extends RecursiveAction {
35:     Main1(サブマクロタスク番号) { }  //コンストラクタ
36:     protected void compute() { 該当するサブマクロタスクを実行; }
37:     public void smtStart() { }  //開始タスク
38:     public void smt_1() { System.out.println("smt_1"); }
39:     public void smt_2() { System.out.println("smt_2"); }
40:     public void smt_3() { System.out.println("smt_3"); }
41:     public void smt_4() { System.out.println("smt_4"); }
42:   }
43: }
44: class Lt_p extends RecursiveAction {
45:   public void f2() { 分割数に従ってLt_pをfork; }
46:   private Lt_p (int LT番号) { LT番号をフィールド変数に設定; }
47:   protected void compute() { ループ並列処理; }
48: }
```

図 3　タスク駆動型並列 Java コード.

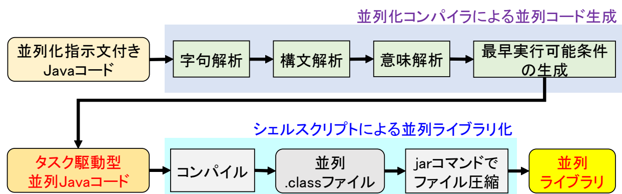

図 4　並列化コンパイラの構成.

ロタスクの逐次実行を行う．さらに，各サブマクロタスクの内部の第 3 階層では，ローカルタスクとして入力文の暗号化および復号化のループ並列処理を行う．この際，ローカルタスク数は 1 の場合 (ループ分割無し)，64 の場合 (ループを 64 分割) の 2 通りとする．

性能評価では，開発した並列化コンパイラでタスク駆動型並列処理コードを生成し，生成されたローカルタスク集合クラスに対して並列ライブラリ化を行い，Intel Xeon Phi プロセッサ上で並列実行を行った．また，HotSpot 最適化は有効としており，並列化を行っていない逐次実行時間は 60,417[ms] である．

性能評価結果は図 6 の通りとなった．ローカルタスク数が 1 の場合の性能評価結果では，最大の速度向上比が 32 スレッド実行の時に 3.73 倍となり，ローカルタスク数が 16 の場合では，最大の速度向上比が 64 スレッド実行の時に 16.63 倍となった．

```
01: class Sample {
02:   public static void main(String[] args) {
03:     /*mt fork Instance(Smt:s1)*/ { Smt s1 = new Smt();  s1.f1(); }
04:     /*mt fork Instance(Smt:s2)*/ { Smt s2 = new Smt();  s2.f1(); }
05:     /*mt fork Instance(Lt:l3) (1 1)&(1 2)*/ { Lt l3 = new Lt();  l3.f2(); }
06:     /*mt fork Instance(Lt:l4) (1 1)&(1 2)*/ { Lt l4 = new Lt();  l4.f2(); }
07:     /*mt fork (1 3)&(1 4)*/ { System.out.println("mt1_5"); }
08:   }
09: }
10: class Smt {
11:   /*mtset*/
12:   public void f1() {
13:     /*smt fork (0)*/ { System.out.println("smt_1");}
14:     /*smt fork (1)*/ { System.out.println("smt_2");}
15:     /*smt fork (1)*/ { System.out.println("smt_3");}
16:     /*smt fork (2)&(3)*/ { System.out.println("smt_4");}
17:   }
18: }
19: class Lt {
20:   /*ltset*/
21:   public void f2() {
22:     /*lt fork fdecomp=分割数 private(変数名)*/ { ループ処理; }
23:   }
24: }
```

図 5　並列化コンパイラにおける入力対象 Java コード.

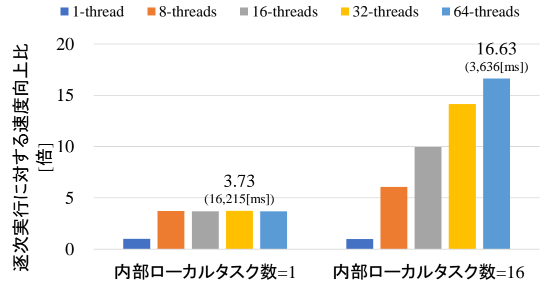

図 6　Intel Xeon Phi 上での AES の性能評価.

5　おわりに

本稿では，並列化コンパイラを用いてタスク駆動型粗粒度並列 Java コード生成を行い，並列化可能なタスク集合をライブラリ化する手法を提案した．

性能評価では，Intel Xeon Phi の 64 コア上で並列実行を行い，共通鍵暗号 AES プログラムにおいて最大 16.63 倍の速度向上が得られ，並列化コンパイラによる並列ライブラリ生成の有効性が確認された．

参考文献

[1] Lea.D.　A Java Fork/Join Framework.　*Proc. ACM conference on Java Grande, JAVA'00*, pp.36-43, 2000.

[2] A.Yoshida, A.Kamiyama, and H.Oka.　A Task-driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform.　*Journal of Information Processing*, Vol.25, pp.426-437, 2017.

[3] 岡宏樹, 吉田明正. メニーコア上でのローカルタスク協調実行をともなうタスク駆動型粗粒度並列処理. 情報処理学会論文誌コンピューティングシステム (ACS), Vol.12, No.3, pp.1-13, 2019.

[4] NVIDIA.　cuBLAS :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cublas/index.html, 2020.

[5] 中島研吾, 星野哲也, 成瀬彰, 塙敏博, 三木洋平. 有限要素法における係数行列生成部のマルチコア・メニィコア向け最適化. 研究報告ハイパフォーマンスコンピューティング (HPC), Vol.2018-HPC-163, No.28, pp.1-8, 2018.

[6] 神永正博, 山田聖, 渡邊高志. Java で作って学ぶ暗号技術. 森北出版株式会社, 2008.

Copyright ©2021 Information Processing Society of Japan.
All Rights Reserved.