C-004

タスク駆動型粗粒度並列処理における インスタンスメソッドの並列ライブラリ化

Parallel Librarization of Instance Method for Task-Driven Coarse Grain Parallel Processing

山端 大揮 †　　吉田 明正 †‡
Daiki Yamahata　　Akimasa Yoshida

1　はじめに

マルチコア上での粗粒度並列処理手法として，ループ並列性に加えて，ループやサブルーチン等の並列性を利用したタスク駆動型粗粒度並列処理が提案されている．タスク駆動型粗粒度並列処理では，粗粒度タスクの実行終了時に後続データ依存タスクを fork することにより，Java Fork/Join Framework 環境 [1] で並列処理を実現する [2]．この方法では従来，インスタンスメソッドは 1 つの粗粒度タスクとして扱われており，その内部の並列性が有効に利用されていなかったが，本手法ではインスタンスメソッド内部においてローカルタスク [3] を生成し，ローカルタスク間の並列性と上位の粗粒度並列性を同時に利用する．

近年，汎用的なプログラムコードをライブラリ化することによって，マルチプラットフォームに対応可能なアプリケーションとして開発する研究が行われており，OpenMP によってマルチスレッド並列化された係数行列生成プロセスの最適化 [4] が挙げられる．本稿では，タスク駆動型粗粒度並列処理 [2] を適用し，ローカルタスクを含むマクロタスクをインスタンスメソッドコードとして実装し，並列ライブラリとする．

本手法による性能評価では，ヤコビ法プログラムに対して，ローカルタスクを含むマクロタスクを並列ライブラリ化し，Intel Xeon プロセッサ上でのタスク駆動型粗粒度並列処理による実行結果からその有効性を確認した．

2　タスク駆動型粗粒度並列処理

本節では，Java Fork/Join Framework 環境 [1] におけるタスク駆動型粗粒度並列実行手法 [2] について述べる．

2.1　Fork/Join Framework によるタスク駆動型実行

タスク駆動型実行とは，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，各階層の粗粒度タスク (マクロタスク，MT) 間のデータ依存と制御依存を解析し，最早実行可能条件 [5] の形で並列性を実現する実行手法である．

Fork/Join Framework によるタスク駆動型実行では，その後にマクロタスクの終了状態と分岐状態を管理し，新たに実行可能になるマクロタスクを fork し，ワーカーキューに投入する．このマクロタスクは，Fork/Join Framework のスケジューラによって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，必要に応じてワークスティーリングが行われる．

†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

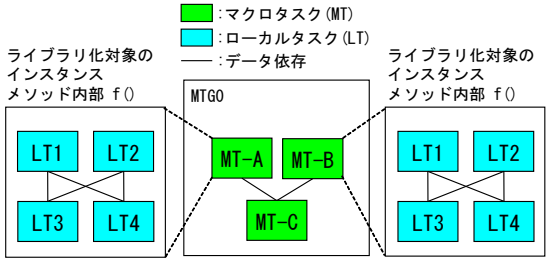

図 1　階層型マクロタスクグラフ

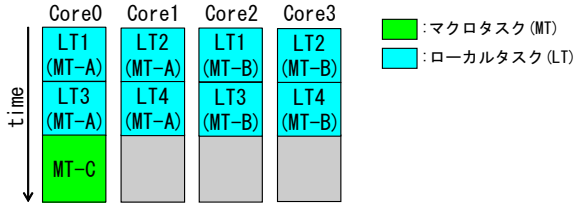

図 2　4 コアでのタスク駆動型実行の並列処理イメージ

2.2　ローカルタスク協調実行による並列処理

ローカルタスク協調実行 [3] では，タスク駆動型粗粒度並列処理のスケジューラがマクロタスクの実行を管理し，Fork/Join Framework のスケジューラが，マクロタスク内部となるインスタンスメソッド内部のローカルタスク (LT) を低オーバーヘッドで実行管理できる．

従来のローカルタスク協調実行 [3] では，部分ループ，再帰メソッドのみをローカルタスクの対象としていた．本稿では通常のデータ依存タスクをローカルタスクの対象とし，タスクに対して fork と join を行い，その後別のタスクに対して fork と join を行う．

例として，LT1 と LT2 の 2 タスクと，LT3 と LT4 の 2 タスクをそれぞれ並列化の対象とする図 1 のようなマクロタスクグラフを 4 コア上で並列実行したイメージは図 2 となる．このとき，並列 Java コードは図 3 のように表現でき，図 3(a) はタスク駆動型並列 Java コードを表し，図 3(b) はローカルタスクを含む，マクロタスク内部のインスタンスメソッドコードを表す．なお，図 3(b) のコードはライブラリ化の対象となる．

2.3　インスタンスメソッドの並列コードとライブラリ化

Java プログラムのコンパイルによる実行手法として，汎用する並列 Java コードを一つのファイルに圧縮して実行する並列ライブラリ化が挙げられる．これによって開発されたライブラリはマルチプラットフォームに対応することができ，アプリケーションを開発者の負担を抑えつつ効率的に実装することが望まれている．

本手法では，マクロタスクとして呼び出されたインスタンスメソッドコードに対して並列ライブラリ化を適用し，インスタンスメソッドコードを呼び出すクラス側で再利用することにより，効率的にコードの記述やプログラムの実行が行える．

```
01: class Mainp { //並列メイン
02:     static ParaRegion paraRegion = new ParaRegion();
03:     static class Layer0 extends RecursiveTask<Void> { //ForkJoin開始
04:         Layer0() { Dataクラスのフィールド変数の初期化; }
05:         protected void compute() {
06:             ForkTemplate_mainクラスのmtStartのforkを行う;
07:             helpQuiesce()でタスク処理へ移行;
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:         ForkTemplate_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() { 該当するマクロタスクを実行; }
17:         public void mtStart() { //mtStart
18:             マクロタスク実行管理テーブル更新;
19:             後続マクロタスクのforkを試みる;
20:         }
21:         public void mt_A() { paraRegion.f(); ... }
22:         public void mt_B() { paraRegion.f(); ... }
23:         public void mt_C() { ... }
24:     }
25:     public static void main(String[] args) {
26:         ForkJoinPool pool = new ForkJoinPool(ワーカースレッド数);
27:         Layer0 layer0 = new Layer0();
28:         pool.invoke(layer0); //ForkJoin開始
29:     }
30: }
```

(a) タスク駆動型並列Javaコード

```
01: class ParaRegion extends RecursiveTask<Void> {
02:     public void f() {
03:         分割数に従ってLT1,LT2をfork;
04:         LT1,LT2をjoin;
05:         分割数に従ってLT3,LT4をfork;
06:         LT3,LT4をjoin;
07:     }
08:     ParaRegion(int forkId) { forkIdをフィールド変数に設定; }
09:     protected void compute() { LT1,2の場合，LT3,4の場合で内容を場合分け; }
10: }
```

(b) ライブラリ化の対象となる，
マクロタスク内部のインスタンスメソッドコード

図 3　タスク駆動型並列 Java コード

図 1 の例において，MT-A と MT-B から同一のメソッド f() を呼び出す場合は，f() の並列ライブラリのみで，図 3(b) のような MT-A と MT-B の内部の並列コードを生成することができる．

3　Intel Xeon 上での粗粒度並列処理の性能評価

本性能評価では，メニーコアプロセッサ Intel Xeon Phi Processor 7250 を搭載したマシン Intel Xeon Phi Knights Landing Server を利用した．本マシンは，Intel Xeon Phi Processor 7250（1.40GHz，68 コア，272 スレッド），メモリ 48GB で構成される．また，OS は CentOS7.4，Java 処理系は JDK1.8 である．

3.1　ヤコビ法プログラムを用いた性能評価

本節では，ヤコビ法プログラムを用いて性能評価を行う．本プログラムは反復ループとその内部のマクロタスクから構成される．また，行列サイズは 16384 × 16384 とした．マクロタスクの分割数を 1，4，8，ローカルタスクの分割数を 1，4，8 と変化させ，計 9 つの実行条件において，Intel Xeon プロセッサの 64 コア実行で性能評価を行った．

JVM による HotSpot 最適化を有効とした通常実行での性能評価結果を図 4 に示す．並列化を行っていない逐次実行時間は 18.0[s] であり，提案手法による，逐次実行に対する速度向上率は，マクロタスク数が 8，マクロタスク内のローカルタスク数が 8 のときの実行条件において 15.5 倍を達成した．

また，HotSpot 最適化を無効とした実行での性能評価結果を図 5 に示す．逐次実行時間は 531.2[s] であり，提案手法による，逐次実行に対する速度向上率は，マクロタスク数が 8，マクロタスク内のローカルタスク数が 8 のときの実行条件において 46.4 倍を達成した．

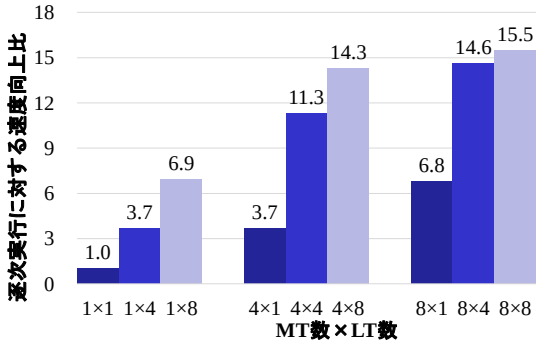

図 4　Xeon Phi における性能評価 (HotSpot 最適化有り)

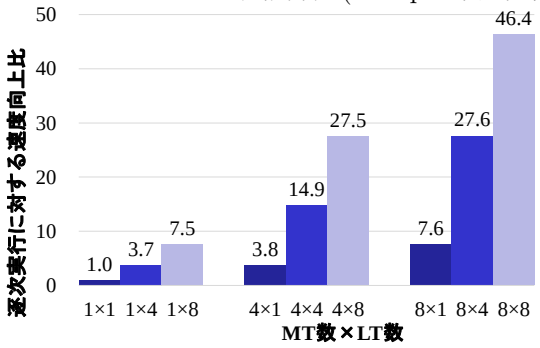

図 5　Xeon Phi における性能評価 (HotSpot 最適化無し)

4　おわりに

本稿では，粗粒度タスク間とローカルタスク間の並列性を同時に利用可能な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手法において，ローカルタスクを含むマクロタスクをインスタンスメソッドコードとして実装し，並列ライブラリ化を行った．性能評価では，メニーコアプロセッサ Intel Xeon Phi Processor 7250 を搭載した Intel Xeon Phi Knights Landing Server の 64 コア上で並列実行を行い，ヤコビ法プログラムにおいて，HotSpot 最適化を有効とした通常実行で最大 15.5 倍の速度向上が得られ，HotSpot 最適化を無効とした実行で最大 46.4 倍の速度向上が得られた．

以上の結果から，Intel Xeon プロセッサにおいて，Java Fork/Join Framework を利用した，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有効性，並びにローカルタスクを含むインスタンスメソッドコードの並列ライブラリ化の有用性が確認された．

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伴う Java プログラムのタスク駆動型粗粒度並列処理, 研究報告システム・アーキテクチャ(ARC), Vol.2018-ARC-232, No.24, pp.1-9, 2018.

[4] 中島研吾, 大島聡史, 塙敏博. 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.

[5] A.Yoshida, Y.Ochi, and N.Yamanouchi. Parallel Java Code Generation for Layer-unified Coarse Grain Task Parallel Processing, IPSJ Transactions on Advanced Computing Systems, Vol.7, No.4, pp.56-66, 2014.