情報処理学会論文誌　コンピューティングシステム　Vol.12　No.3　1–13 (July 2019)

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

岡 宏樹[1,†1]　吉田 明正[1,2,3,a)]

受付日 2018年12月1日, 採録日 2019年3月20日

概要：メニーコアプロセッサはサーバからスーパーコンピュータに至るまで，高性能な並列処理を実現するために広く利用されている．メニーコア環境においてプロセッサの性能を最大限に引き出すためには高い並列性が求められており，従来のマルチコア向け並列プログラムではその性能を十分に引き出せない可能性がある．また，最近の Java 並列処理環境には Fork/Join Framework が導入されており，そのフレームワークを用いて粗粒度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．本論文では，従来のタスク駆動型粗粒度並列処理で利用していたマクロタスク（粗粒度タスク）レベルの並列性に加えて，ローカルタスク（マクロタスクを構成するループイタレーション集合あるいは再帰メソッド呼び出し文）レベルの並列性を利用可能にするローカルタスク協調実行手法を提案する．本論文の性能評価では，開発した並列化コンパイラにより提案手法の並列コードを生成する．メニーコア Intel Xeon Phi Knights Landing 上で Java Grande Forum Benchmark Suites による性能評価を行ったところ，68 コア実行においてローカルタスク協調実行手法の適用により，タスク駆動型粗粒度並列処理の実行時間が 36.5～50.6%短縮された．また，再帰プログラムを用いた性能評価においても，従来の Fork/Join 実行と比べて実行時間が 11.1～14.5%短縮された．これらの結果より，提案手法の有効性が確認された．

キーワード：メニーコア，タスク駆動，粗粒度並列処理，コンパイラ，Java Fork/Join

Task-Driven Coarse Grain Parallel Processing with Cooperative Execution of Local Tasks on Many-core Processors

Hiroki Oka[1,†1]　Akimasa Yoshida[1,2,3,a)]

Received: December 1, 2018, Accepted: March 20, 2019

Abstract: Many-core processors are used to realize high-performance parallel processing for supercomputers as well as servers. In order to maximize the performance of many-core processors, it is required to use the sufficient parallelism. Thus, conventional parallel program toward multicores could not attain the maximum performance. Also, the recent Java parallel processing platform adopts the Fork/Join Framework. Such a framework can be used for coarse grain task parallel processing by means of the task-driven coarse grain parallel processing scheme. This paper proposes a local task cooperative execution method to use not only coarse grain parallelism but also local task parallelism that is either a set of loop-iterations within a macro-task or a recursive method invocation. In the performance evaluation, our developed parallelizing compiler can generate parallel codes corresponding to the proposed scheme. From the result of evaluation using Java Grande Forum Benchmark Suites on the many-core Intel Xeon Phi Knights Landing, the proposed local task cooperative execution scheme could reduce the execution time of the task-driven coarse grain task parallel processing by 36.5–50.6% compared to the execution without the proposed scheme. Furthermore, in the performance evaluation using recursive programs, the execution time with the proposed scheme could be reduced by 11.1–14.5%. Consequently, the effectiveness of the proposed scheme was confirmed.

Keywords: many-core, task-driven, coarse grain parallel processing, comipler, Java Fork/Join

1. はじめに

メニーコアプロセッサは高性能な並列処理環境として，サーバからスーパーコンピュータに至るまで幅広い場面で利用されている．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高い並列性が必要であり，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] を利用すること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，プラットフォームに依存しない 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）の実効性能が，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．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．

一方，複数階層にまたがった粗粒度タスクの並列実行手法として階層統合型粗粒度並列処理 [8] が提案されている．階層統合型粗粒度並列処理は，粗粒度タスク並列性を階層型マクロタスクグラフ [2] として表現しており，Thread クラス（Runnable インタフェース）を用いた並列コードにより，階層を越えたマクロタスクの統一的制御と並列性利用が可能になっている [9]．また，近年の Java Fork/Join Framework 環境上で階層統合型粗粒度並列処理を実現するために，タスク駆動型実行手法 [10] が提案されており，Fork/Join Framework を利用した粗粒度並列処理が実現されている．ただし，このタスク駆動型実行手法では並列化可能ループの分割によりマクロタスク数および並列コード長の増加が生じ，実行時のダイナミックスケジューリングのオーバヘッドの増加や，HotSpot 最適化の効果が十分に得られない可能性がある．加えて，再帰メソッド内部の並列性が有効に利用されていなかった．それゆえ，メニーコアのように高い並列性を必要とする環境では，その性能を十分に発揮できない場合がある．

本論文では，メニーコアプロセッサ上でタスク駆動型実行による粗粒度並列処理を効率よく実現するために，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（粗粒度タスク）を構成するループイタレーション集合あるいは再帰メソッド呼び出し文を新たに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方式をとる．これにより，スケジューリング対象となるマクロタスク数の減少ならびに並列コード長の短縮により，実行時オーバヘッドの削減および HotSpot 最適化の効果が十分に得られることによって，メニーコア環境においても高い実効性能を達成することが可能になる．また，再帰メソッド内部の並列性もマクロタスク並列性と共存して利用できるため，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．

本論文の構成は以下のとおりとする．2 章では，Java Fork/Join Framework 環境におけるタスク駆動型粗粒度並列処理について述べる．3 章では，ローカルタスク協調実行をともなうタスク駆動型並列処理について述べる．4 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．5 章では，メニーコア上でローカルタスク協調実行をともなうタスク駆動型並列処理の性能評価について述べる．6 章で関連研究について述べ，7 章でまとめを述べる．

2. Java Fork/Join Framework 環境におけるタスク駆動型粗粒度並列処理

本章では，提案手法の基盤となる Java Fork/Join Framework を用いたタスク駆動型粗粒度並列処理手法 [10] について述べる．

2.1 Java Fork/Join Framework の概要

Java Fork/Join Framework [7] は，Java SE 7 以降に導入されている 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．生成された各ワーカースレッドはそれぞれ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，ワーカーキュー内のタスクがワーカースレッドによって取り出され，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．

[1] 明治大学大学院先端数理科学研究科
Graduate School of Advanced Mathematical Sciences, Meiji University, Nakano, Tokyo 164–8525, Japan
[2] 明治大学総合数理学部
School of Interdisciplinary Mathematical Sciences, Meiji University, Nakano, Tokyo 164–8525, Japan
[3] 早稲田大学グリーンコンピューティングシステム研究機構
Green Computing Systems Research Organization, Waseda University, Shinjuku, Tokyo 162–0042, Japan
[†1] 現在，ピー・シー・エー株式会社
Presently with PCA CORPORATION
[a)] akimasay@meiji.ac.jp

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

図 1　並列化指示文付 Java コード（ローカルタスクなし）
Fig. 1　Java code including parallelizing directives without local tasks.

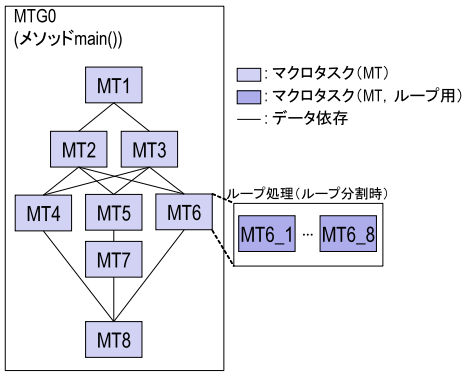

図 2　階層型マクロタスクグラフ（ローカルタスクなし）
Fig. 2　Hierarchical macro-task-graph without local tasks.

表 1　タスク駆動型実行の最早実行可能条件
Table 1　Earliest-executable-conditions for the task-driven execution.

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

2.2　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当てを行う．また，タスク駆動型実行 [10] では並列化コンパイラが生成する Fork/Join Framework を用いた並列 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．

たとえば，**図 1** の Java コードは**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．この表において，MT6 のループ分割後のマクロタスクは省略されている．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（MT6_1〜MT6_8）を fork してワーカーキューに投入する．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，マクロタスク間の並列性が利用されていることが分かる．ただし，図 3 では提案するローカルタスクが導入されていないため，再帰メソッド呼び出しが行われる MT4 は 1 スレッド実行となる．

2.3　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（メソッド呼び出し）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

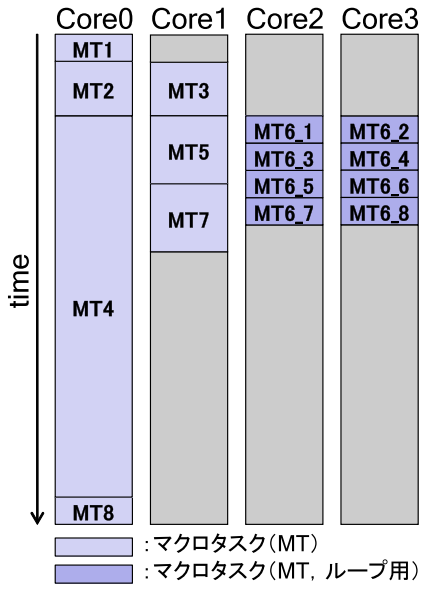

図 3　4 コアでのタスク駆動型実行の並列処理イメージ（ローカルタスクなし）

Fig. 3　An image of the task-driven execution without local tasks on 4 cores.

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．ただし，上位階層で実行時に利用できるすべてのコアを充足するのに十分な並列性が得られる場合には，下位階層の並列性を利用する必要がないため，下位階層のマクロタスクを定義する必要はない．

2.4　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ともな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，3 章で提案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ともなうスケジューラによってのみ管理される．

2.5　タスク駆動型並列処理コード

図 1 に示す並列化指示文付 Java コードを並列化コンパイラへの入力すると，**図 4** に示すタスク駆動型並列処理 Java コードが生成される．この並列処理プログラムでは，初めに Main_p クラス内の main() メソッド（図 4 の 52〜56 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が生成される．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，Layer0 クラスのインスタンスが invoke() メソッドにより呼び出されることで，Fork/Join による並列処理が開始される．

```
01:  class Main_p{ // 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:        ForkTemplateMain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 ForkTemplateMain extends RecursiveAction{
13:      マクロタスク間共有変数等の宣言;
14:      ForkTemplateMain(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(){
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_1(){
30:        for(i=0; i<M*(1/8); i++){ ... }
31:        ...
32:      }
33:      public void mt6_2(){
34:        for(i=M*(1/8); i<M*(2/8); i++){ ... }
35:        ...
36:      }
37:      ...
38:      public void mt6_8(){
39:        for(i=M*(7/8); i<M; i++){ ... }
40:        ...
41:      }
42:      public void mt7(){ ... }
43:      public void mt8(){ ... }
44:    }
45:    static int recurFunc(int n){
46:      ...
47:      y1 = recurFunc(n-1);
48:      y2 = recurFunc(n-2);
49:      y = y1 + y2;
50:      ...
51:    }
52:    public static void main(String[] args){
53:      ForkJoinPool pool = new ForkJoinPool(ワーカースレッド数);
54:      Layer0 layer0 = new Layer0();
55:      pool.invoke(layer0); // ForkJoin開始
56:    }
57:  }
```

図 4　タスク駆動型並列 Java コード

Fig. 4　Task-driven parallel Java code.

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，Main_p クラス内部の ForkTemplateMain クラスのマクロタスク mtStart（図 4 の 20〜23 行目）が実行される．このとき，マクロタスク識別情報を元に ForkTemplateMain クラスの compute() メソッド（図 4 の 17〜19 行目）が実行され，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 24〜43 行目）を実行する．

2.6　マクロタスクを用いたループ並列処理

タスク駆動型並列処理において並列化可能ループの並列

処理を行う場合，ループ制御変数の範囲を分割し，分割された範囲のイタレーション集合を部分ループとして扱う．この分割されたそれぞれの部分ループは，4.2 節で述べる並列化指示文によって定義されたマクロタスクと同様に，マクロタスクとして扱われる．

マクロタスク実行によるループ並列処理は mt6_1～mt6_8（図 4 の 29～41 行目）によって行われる．ここではループを 8 分割しており，mt6_1～mt6_8 のそれぞれの内部で部分ループを実行する．

ただし，マクロタスク実行によるループ並列処理においてはループの分割数の増加によってマクロタスク数および並列コード長の増加を招くため，ダイナミックスケジューリングのオーバヘッドの増加や HotSpot 最適化の効果の低下により，十分な実効性能が得られない場合がある．特に，メニーコア環境のように高い並列性を持つプログラムが必要になる場合は，これらの影響が顕著になる．なお，提案手法によるループ並列処理は 3.2 節，3.3 節で述べる．

2.7　マクロタスクを用いた再帰メソッド処理

タスク駆動型粗粒度並列処理では，再帰メソッドは 1 つのマクロタスクとして扱われ，メソッド内部の処理は逐次に実行される．並列処理コードでは，再帰メソッドである recurFunc()（図 4 の 45～51 行目）は 1 スレッドで実行される．ただし，再帰メソッドを実行するマクロタスクと他のマクロタスクを並列に実行することは可能である．なお，提案手法による再帰メソッドの並列処理は 3.4 節，3.5 節で述べる．

3.　ローカルタスク協調実行をともなうタスク駆動型並列処理

本章では，ローカルタスク協調実行を用いたタスク駆動型の並列処理手法について述べる．

3.1　ローカルタスク協調実行の概念

本論文で提案する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では，マクロタスクに加えてローカルタスクという新たなタスクの定義を導入する．ローカルタスクはマクロタスクのような実行管理は行わず，Fork/Join Framework のスケジューラによってのみ管理される．これにより，ローカルタスクではマクロタスクのような複雑な依存関係を考慮した実行制御はできないが，マクロタスク実行管理による実行時オーバヘッドはともなわなくなる．加えて，再帰メソッド内部の並列性も引き出すことができる．

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ともなわない従来のタスク駆動型並列処理 [10] 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，本論文で提案するローカルタスク協調実行をともなうタスク駆動型並列処理は，マクロタスク間，ローカルタスク間，およびマクロタスクとローカルタスクの間の並列性を利用することができる．

表 2　Fork/Join Framework を用いた並列処理方式
Table 2　Parallel processing scheme using Fork/Join Framework.

実行方式	マクロ タスク間並列	ローカル タスク間並列
通常の Fork/Join による実行	×	○
タスク駆動型並列処理 （ローカルタスク協調実行なし）	○	×
タスク駆動型並列処理 （ローカルタスク協調実行あり）	○	○

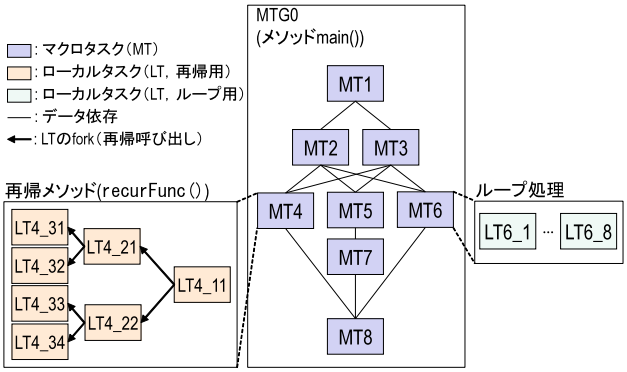

図 5　ローカルタスク協調実行のための階層型マクロタスクグラフ
Fig. 5　Hierarchical macro-task-graph for local task cooperative execution.

3.2　ローカルタスクを用いたループ並列処理

提案手法によるループ並列処理では，ループ制御変数の範囲を分割し，分割された範囲のイタレーション集合をローカルタスクと定義し，並列に実行する．

図 5 は，図 2 のマクロタスクグラフに対して提案手法を適用した場合のマクロタスクグラフである．また，**図 6** はそのプログラムを 4 コア（4 スレッド）上で実行したイメージである．これらの図のように，従来マクロタスクとして扱われていた部分ループは，ローカルタスク（LT6_1～LT6_8）として扱われる．

ローカルタスクによる実行は Fork/Join Framework のスケジューラによってのみ管理されるため，分割数の増加にともなうマクロタスク実行管理によるオーバヘッドは発生せず，HotSpot 最適化の効果を十分に利用することができる．特に，メニーコア環境のように高い並列性が要求される場合においても，十分な実効性能を得ることが可能になる．

3.3　ローカルタスクを用いたループ並列処理コード

ローカルタスクを用いたループ並列処理コードを生成す

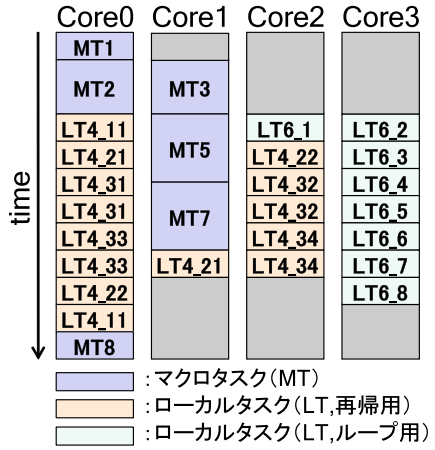

図 6　4 コアでのローカルタスク協調実行をともなうタスク駆動型実行の並列処理イメージ

Fig. 6　An image of the task-driven execution with local task cooperative execution on 4 cores.

```
01: /*mt fork fdecomp=8 private(i) (1 2)&(1 3)*/{
```

(a) ローカルタスクを用いたループ並列処理のための並列化指示文付Javaコード

```
01: /*recur*/
02: int recurFunc(int n){
03:   ...
04:   x = pre();
05:   if (n > 0){
06:     /*recur fork(1) if(n-1 > 10)*/
07:     { y1 = recurFunc(n-1); }
08:     /*recur fork(2) if(n-2 > 10)*/
09:     { y2 = recurFunc(n-2); }
10:     /*recur join(1)*/
11:     /*recur join(2)*/
12:     y = y1 + y2;
13:   }
14:   else{
15:     y = 0;
16:   }
17:   z = post();
18:   return x + y + z;
19: }
```

(b) ローカルタスクを用いた再帰メソッドの並列処理のための並列化指示文付Javaコード

図 7　ローカルタスク協調実行のための並列化指示文付 Java コード

Fig. 7　Java code with parallelizing directives for local task cooperative execution.

るためには，図 1 の並列化指示文付 Java プログラムの 29 行目を**図 7** (a) のコードに置き換えたコードを，並列化コンパイラに入力する．生成された並列処理コードは，図 4 の 29〜41 行目を**図 8** のコードに置き換えたものとなる．

図 8 において，ループ処理を行うマクロタスク mt6() メソッド（図 8 の 1 行目）が呼び出されると，Mt6 クラスの execute() メソッド（図 8 の 3〜5 行目）が実行される．これにより，Mt6 クラスのインスタンスが並列化指示文付 Java コード（図 7 (a)）において定義されたループ処理の分割数と同数生成され，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，fork されたインスタンスは compute() メソッド（図 8 の 9〜12 行目）を実行する．compute() メソッドで

```
01: public void mt6(){ Mt6.execute(); ... }
02: static class Mt6 extends RecursiveAction{
03:   static void execute(){
04:     分割数に従ってMt6をfork;
05:   }
06:   private Mt6(int forkId){
07:     forkIdをフィールド変数に設定;
08:   }
09:   protected void compute(){
10:     ...
11:     for(i=M*(forkId/8); i<M*((forkId+1)/8); i++){ ... }
12:   }
13: }
```

図 8　ローカルタスク協調実行をともなうループ並列処理のための Java コード

Fig. 8　Parallel Java code for loop paralleization with local task cooperative execution.

はループ処理が行われ，それぞれのインスタンスは forkId に従って異なるイタレーションの範囲を実行する．

3.4　ローカルタスクを用いた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文献 [11] により提案されているが，再帰呼び出しのたびにダイナミックスケジューラの管理対象となるマクロタスクが増加する傾向がある．そこで，本論文ではローカルタスクを用いて再帰メソッドの並列処理を行う手法を提案する．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帰メソッドが再び呼び出される際には再帰的に fork が行われる．

図 5 に示すように，従来は 1 つのマクロタスクにより実行されていた再帰メソッド（MT4）は，ローカルタスク（LT4_11〜LT4_34）によって実行される．これにより，図 6 に示すように従来の図 3 と比べてマルチコアを効率的に利用することが可能になり，実行時間の短縮が実現できていることが分かる．

ローカルタスクによる実行は Fork/Join Framework のスケージューラによって管理されるため，ローカルタスクが過剰に再帰的に fork された場合には，Fork/Join Framework のスケージューラのパフォーマンスが低下する場合がある．そこで，提案手法では再帰メソッドの fork の際に条件を設定可能にしている．実行中，この条件を満たさない場合には同一スレッド内でメソッドの再帰呼び出しが行われ，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 によるオーバヘッドを最小限に留めることができる．

3.5　ローカルタスクを用いた再帰メソッドの並列処理コード

図 9 はローカルタスク実行による再帰メソッドの並列処理コードであり，図 4 において 45〜51 行目をこれに置

```
01:  static int recurFunc(int n){ RecurFuncをfork; }
02:  private static class RecurFunc
                          extends RecursiveTask<Integer>{
03:    RecurFunc(int n){
04:      nをフィールド変数に設定;
05:    }
06:    protected Integer compute(){
07:      ...
08:      boolean isRecurTask1Forked = (n-1 < 10);
09:      ForkJoinTask<Integer> recurTask1 = null;
10:      if(isRecurTaks1Forked)
11:        recurTask1 = new RecurFunc(n-1).fork();
12:      ...
13:      if(isRecurTask1Forked)
14:        y1 = recurTask1.join();
15:      else
16:        y1 = selfCompute(n-1);
17:      ...
18:    }
19:    private static int selfCompute(int n){ ... }
20:  }
```

図 9　ローカルタスク協調実行をともなう再帰メソッドのための並列 Java コード

Fig. 9　Parallel Java code for recursive method with local task cooperative execution.

き換えることで，ローカルタスク協調実行を用いた再帰メソッドの並列処理を実現することができる．並列化コンパイラでは，この置き換えが行われた後の並列コードを生成することができる．

再帰メソッドである recurFunc() メソッド（図 9 の 1 行目）が実行されると，RecurFunc クラス（図 9 の 2〜20 行目）のインスタンスの生成および fork が行われ，クラス内部の compute() メソッド（図 9 の 6〜18 行目）が実行される．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．ただし，並列化指示文付 Java コードにおいて再帰メソッドの fork 条件が定義されている場合（図 7 の 6，8 行目）には，定義された条件を満たすときにのみ fork が行われる．条件を満たさない場合には逐次実行用の再帰メソッドである selfCompute() メソッド（図 9 の 19 行目）が呼び出され，同一スレッド内で再帰メソッドが実行される．

4. 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実行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 コード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 コードは，本コンパイラの字句解析・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 コードにも対応可能であるが，本コンパイラでは並列化指示文を挿入した Java コードは 1 つのファイルにまとめておく仕様となっており，並列化指示文を付加していない Java コードは本コンパイラによってコンパイルする必要がない．なお，使用するメソッドはあらかじめスレッドセーフな形にリストラクチャリングされていることを前提とする．

本並列化コンパイラでは，並列化指示文を付加した Java コード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．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）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．

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，本手法では入力する Java コードにおいて**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，入力する 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，`/*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

表 3　並列化指示文
Table 3　Parallelizing directives.

並列化指示文の表記	意味
/*mt fork 第 1 指示節 第 3 指示節 */	マクロタスクの定義
/*mt fork inner 第 3 指示節 */	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
第 1 指示節として，decomp=分割数 第 2 指示節	指定した分割数にマクロタスクとしてループ分割
第 1 指示節として，fdecomp=分割数 第 2 指示節 †	指定した分割数にローカルタスクとしてループ分割
第 2 指示節として，private(変数名)	ループ分割における変数のプライベート化
第 2 指示節として，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
第 3 指示節として，論理式	最早実行可能条件を論理式で指定
/*recur*/ †	再帰メソッドの定義
/*recur fork(ID) 第 4 指示節*/ †	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 に対する join）
第 4 指示節として，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

†：提案手法により追加した指示文

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．

4.3　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 コード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後，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を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後，図 4 のような ForkTemplateMain のような形式の並列 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後，JVM 上で実行することが可能となる．

5.　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並列処理の性能評価

本章では，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の性能評価について述べる．

5.1　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，1.40 GHz）を搭載し，メモリは 48 GB，OS は CentOS7.4，Java 処理系は JDK1.8 のマシンを利用した．また，Xeon Phi の MCDRAM はキャッシュモードとしている．

性能評価プログラムとして，5.2 節では**表 4** に示す Java Grande Forum Benchmark Suite Version 2.0 [12] から 4 つの Java プログラムを用い，5.3 節では**表 5** に示す 2 つの再帰プログラムを用いる．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あらかじめ行っており，表 3 の並列化指示文を加えたソースプログラムは 1 ファイルにまとめられている．なお，実行時間の測定においてはそれぞれ 5 回の計測を行い，最大と最小の値を除いた 3 回の平均値を用いている．

5.2　Java Grande Forum Benchmark Suite を用いた性能評価

本節では，表 4 に示す Java Grande Forum Benchmark Suite Version 2.0 [12] 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

表 4　Java Grande Forum Benchmark Suite の性能評価プログラム
Table 4　Performance evaluation programs in Java Grande Forum Benchmark Suite.

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=5,000 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
MT 数（ローカルタスク協調なし，ループ分割を含む）	104	1,006	1,007	502
MT 数（ローカルタスク協調あり）	4	6	7	2
ループ分割数	50	1,000	1,000	500
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
Table 5　Performance evaluation programs including recursive methods.

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
MT 数（ローカルタスク協調なし，ループ分割を含む）	9	39
MT 数（ローカルタスク協調あり）	9	7
ループ分割数	—	16
通常の Fork/Join による並列 Java コード（MT 間並列の利用なし）[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あり）[行]	415	607
逐次実行時間 [ms]	50,508	20,892

らに対して提案するローカルタスク協調実行をともなう並列処理を適用している．また，ループ分割数は事前に行った予備実行の結果を考慮して決定している．

Crypt　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．開発した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，従来のローカルタスク協調実行手法を用いない場合（マクロタスクとしてのループ分割あり）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実行手法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，実行時間が 46.3%短縮された．

次に，HotSpot 最適化の影響を調べるために，Crypt のタスク駆動型粗粒度並列処理プログラムを HotSpot 最適化を無効にして実行したところ，**図 11** に示すようにローカルタスク協調実行手法の有無による速度向上の差は HotSpot 最適化を有効にした場合と比べて小さくなった．また，表 4 に示すように，ローカルタスク協調実行を用いない従来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実行を用いたコード長は 584 行となっている．これらの結果から，コード長の短縮 [13] に起因して HotSpot 最適化の効果が大きく発揮され，図 10 に示すように提案手法において高い速度向上率が得られた．

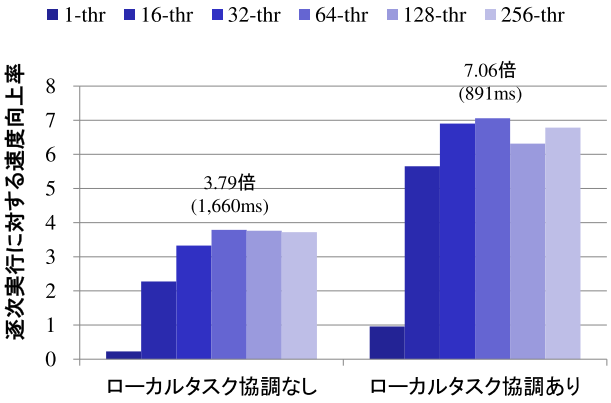

図 10　Crypt におけるタスク駆動型粗粒度並列処理
Fig. 10　Task-driven coarse-grain parallelization for Crypt.

Series　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．開発した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 12** に示す

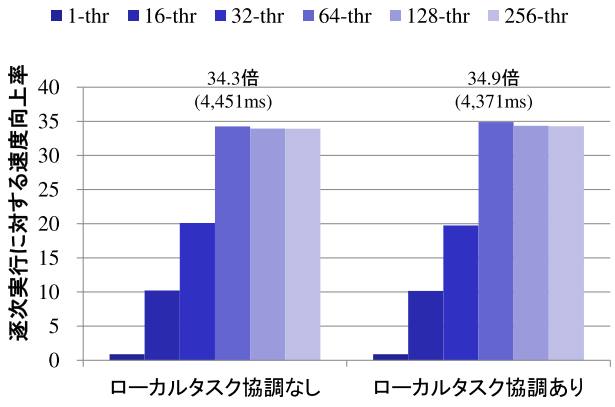

図 11　Crypt における HotSpot 最適化を無効にしたタスク駆動型粗粒度並列処理

Fig. 11　Task-driven coarse-grain parallelization without HotSpot optimization for Crypt.

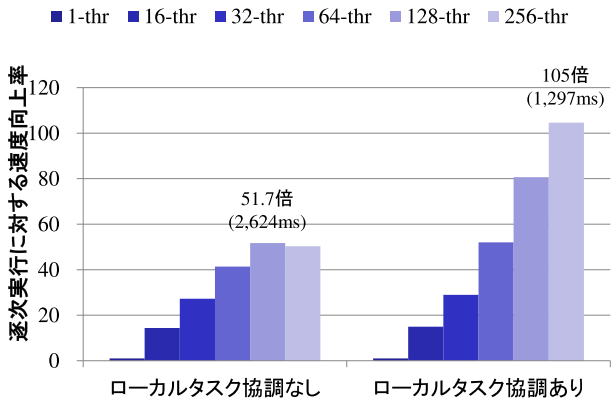

図 12　Series におけるタスク駆動型粗粒度並列処理

Fig. 12　Task-driven coarse-grain parallelization for Series.

ように，ローカルタスク協調実行手法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 51.7 倍の速度向上であるのに対し，ローカルタスク協調実行手法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 105 倍の速度向上を達成し，実行時間が 50.6%短縮された．

次に，HotSpot 最適化を無効にして実行したところ，**図 13** に示すようにローカルタスク協調実行手法の有無による速度向上の差は HotSpot 最適化を有効にした場合と比べて小さくなった．また，表 4 に示すように，ローカルタスク協調実行を用いない従来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実行を用いたコード長は 761 行となっている．これらの結果から，図 12 に示すように提案手法において HotSpot 最適化を有効にした場合にはその効果が十分に発揮され，高い速度向上率が得られた．

Monte Carlo　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，2,585 行（11 ファイル）の

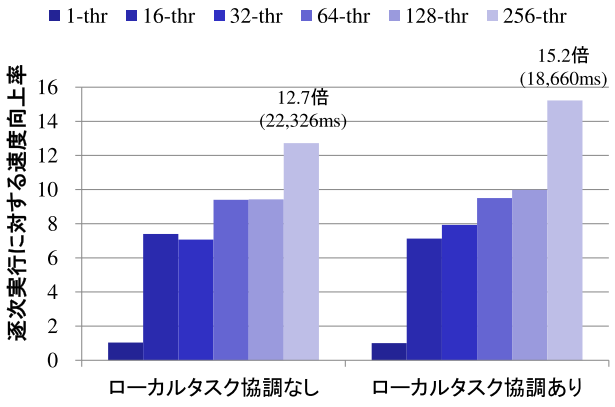

図 13　Series における HotSpot 最適化を無効にしたタスク駆動型粗粒度並列処理

Fig. 13　Task-driven coarse-grain parallelization without HotSpot optimization for Series.

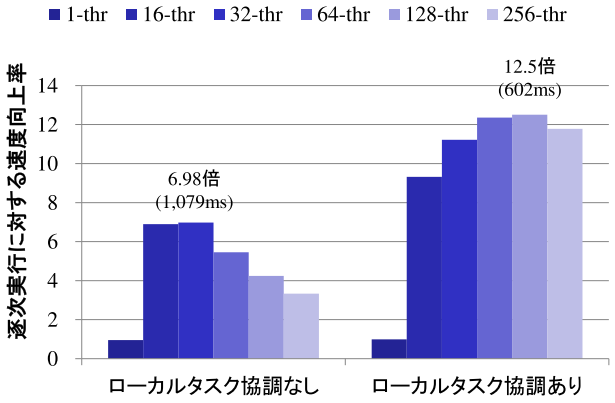

図 14　Monte Carlo におけるタスク駆動型粗粒度並列処理

Fig. 14　Task-driven coarse-grain parallelization for Monte Carlo.

並列化対象外ソースコードからなる．開発した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 14** に示すように，ローカルタスク協調実行手法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 6.89 倍の速度向上であるのに対し，ローカルタスク協調実行手法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 12.5 倍の速度向上を達成し，実行時間が 45.9%短縮された．また，表 4 に示すように，ローカルタスク協調実行手法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実行手法を用いた場合のコード長は 856 行となっている．

RayTracer　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．開発した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 15** に示すように，ローカルタスク協調実行手法を

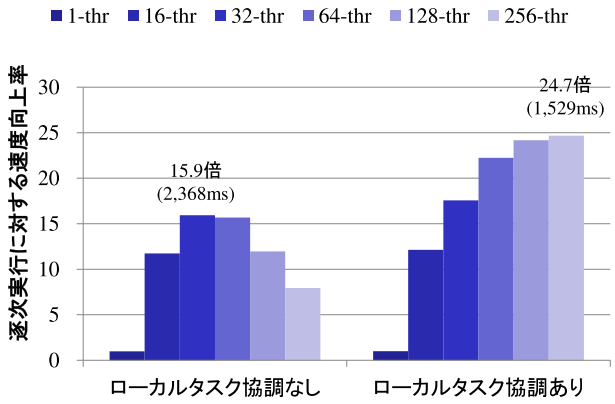

図 15　Raytracer におけるタスク駆動型粗粒度並列処理
Fig. 15　Task-driven coarse-grain parallelization for Raytracer.

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実行手法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，実行時間が 36.5%短縮された．また，表 4 に示すように，ローカルタスク協調実行手法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実行手法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，ローカルタスク協調実行によるループ並列処理をともなうタスク駆動型粗粒度並列処理の有効性が確かめられた．

5.3　再帰プログラムを用いた性能評価

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協調実行をともなう並列処理を適用している．また，後に述べる再帰メソッドの fork 条件は，事前に行った予備実行の結果を考慮して決定している．

フィボナッチ　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．また，再帰メソッドの fork 条件は $n > 27$（n はメソッドが求めるフィボナッチ数）となっている．開発した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 16** 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，マクロタスクのみ（ローカルタスク協調を用いない粗粒度並列処理）の場合は 256 スレッド実行の場合に最大で 7.45 倍の速度向上であった．これに対し，ローカルタスク協調実行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，実行時間が 14.5%短縮され，ローカルタスク間（通常の Fork/Join），マクロタスク間両方の並列性を利用することによる実効性能の向上が確認された．

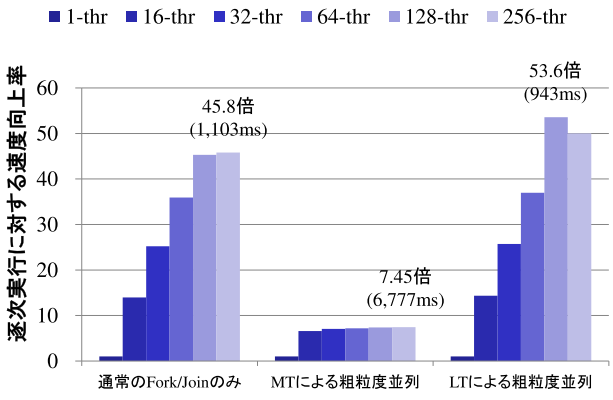

図 16　フィボナッチプログラムによる性能評価
Fig. 16　Performance evaluation of parallelization for Fibonacci program.

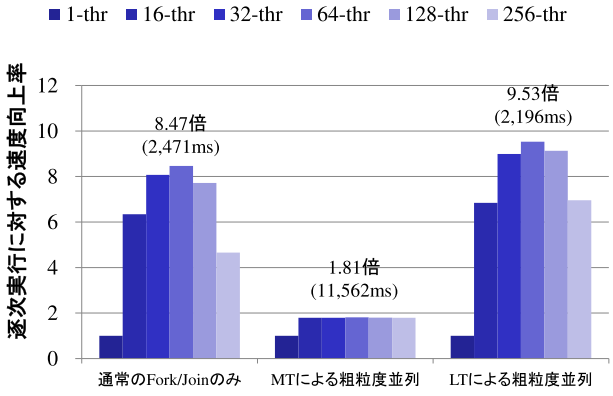

図 17　マージソートプログラムによる性能評価
Fig. 17　Performance evaluation parallelization for Merge-sort program.

マージソート　マージソートプログラムは，サイズが 5,000 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．また，再帰メソッドの fork 条件は $n > 10{,}000$（n はメソッドがソートする配列の大きさ）となっている．開発した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 17** 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 8.47 倍，マクロタスクのみの場合は 64 スレッド実行の場合に最大で 1.81 倍の速度向上であった．これに対し，ローカルタスク協調実行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，実行時間が 11.1%短縮され，ローカルタスク間（通常の Fork/Join），マクロタスク間両方の並列性を利用することによる実効性能の向上が確認された．

これらの 2 つのプログラムの性能評価の結果から，ローカルタスク間，マクロタスク間，およびローカルタスクとマクロタスクの間の並列性を利用する提案手法の有効性が

確かめられた．

6. 関連研究

Java による並列処理に関する研究として，HPF のような配列分散を取り入れた HPJava [14]，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 [15]，ランタイムサポートによりスレッド間並列性を利用する zJava [16] や Jrpm [17] 等が提案されている．しかし，これらはいずれも複数階層の粗粒度タスク間並列性を統合的に利用することは困難であった．

Java Fork/Join Framework で利用されているワークスティールをともなうスケジューラは，Java 言語だけでなく，Cilk 言語 [18] や C++言語の Intel TBB（Threading Building Blocks）[19] にも組み込まれており有望といえる．また，X10 言語を対象としたワークスティールスケジューラも提案されている [20]．Habanero-Java では，コンパイラサポートをともなって async-finish 並列性をサポートするワークスティールの実行時拡張技術が提案されている [21]．Tascell と呼ばれるワークスティールフレームワークでは，分割統治グラフ探索時の負荷バランスをとるために，ワークスペースの長時間排他利用技術が提案されている [22]．これらの研究はワークスティール機能の向上を目指しており，本論文が対象とする Java Fork/Join Framework 環境でタスク駆動型粗粒度並列処理を実現するアプローチとは異なる．

再帰プログラムの階層統合型粗粒度並列処理に関する研究として，Runnable インタフェースを用いたマクロタスクスケジューリングコードの実装方法 [11] が提案されているが，再帰メソッド内部のそれぞれの再帰呼び出し文をマクロタスクとして定義するため，メニーコアのような高並列性を必要とする環境では再帰呼び出しにともなうオーバヘッドが大きくなる可能性がある．

メニーコアプロセッサ上での並列処理に関連する研究として，Intel Xeon Phi Knights Corner を対象とした分割統治法における並列処理についての研究 [23] がある．この研究では分割ツリーの並列性とマージ処理内の並列性をともに利用しているが，同一レベルのマージ処理間にて同期が行われており，並列性は最大限には利用されていない．また，Fork/Join Framework を用いたメニーコアの組み込みシステム向けフレームワークとして ARTM が提案されている [24]．このフレームワークでは軽量 Fork/Join Framework を API として提供しているが，再帰メソッドの並列化には対応しておらず，また 16 より多いコア数では性能評価が行われていない．

7. おわりに

本論文では，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ともな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 コードを入力として，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．

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手法を利用したタスク駆動型並列 Java コードを生成した．この並列 Java プログラムを，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 105 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ともな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. 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. 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] 遠藤佑太，山内長承，吉田明正：マルチコア上での再帰プログラムの階層間並列性を利用した粗粒度タスク並列処理，情報処理学会研究報告，Vol.2014-ARC-208, No.24, pp.1–8 (2014).
[12] EPCC: Java Grande benchmark suite, EPCC (online) (2018), available from ⟨https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite⟩.
[13] 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).
[14] 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).
[15] 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).
[16] Chan, B. and Abdelrahman, T.S.: Run-Time Support for the Automatic Parallelization of Java Programs, *J. Supercomputing*, Vol.28, pp.91–117 (2004).
[17] Chen, M.K. and Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs, *Proc. ISCA-30*, pp.434–446 (2003).
[18] Frigo, M., Leiserson, C. and Randall, K.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pp.212–223 (1998).
[19] Reinders, J.: *Intel Threading Building Blocks*, O'Reilly & Associates, Inc. (2007).
[20] Tardieu, O., Wang, H. and Lin, H.: A Work-Stealing Scheduler for X10's Task Parallelism with Suspension, *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'12* (2012).
[21] Raman, R., Zhao, J., Budimlic, Z. and Sarkar, V.: Compiler Support for Work-Stealing Parallel Runtime Systems, Rice Technical Report TR10-02 (2010).
[22] Yasugi, M., Hiraishi, T., Umatani, S. and Yuasa, T.: Parallel Graph Traversals using Work-Stealing Frameworks for Many-core Platforms, 情報処理学会論文誌, Vol.20, No.1, pp.128–139 (2012).
[23] 廣田悠輔，今村俊幸：メニーコアプロセッサ向け分割統治法の実装技術，研究報告ハイパフォーマンスコンピューティング（HPC），Vol.2017-HPC-158, No.20, pp.1–9 (2017).
[24] Ojail, M., David, R., Lhuillier, Y. and Guerre, A.: ARTM: A Lightweight Fork-Join Framework for Many-core Embedded Systems, *Design, Automation & Test in Europe Conference & Exhibition* (2013).

岡 宏樹

2017 年明治大学総合数理学部ネットワークデザイン学科卒業．2019 年明治大学大学院先端数理科学研究科ネットワークデザイン専攻博士前期課程修了．在学中，Java の並列化コンパイラの研究に従事．2019 年よりピー・シー・エー株式会社に勤務．

吉田 明正　（正会員）

1991 年早稲田大学理工学部卒業．1993 年同大学大学院理工学研究科修士課程修了．1996 年同大学大学院博士後期課程修了．博士（工学）．1993 年日本学術振興会特別研究員（DC1）．1995 年早稲田大学理工学部助手．1997 年東邦大学理学部情報科学科講師，助教授，准教授を経て，2013 年明治大学総合数理学部ネットワークデザイン学科准教授，2016 年より同教授．2016 年早稲田大学グリーンコンピューティングシステム研究機構研究院客員教授．主に，並列分散処理，並列化コンパイラ，データローカリティ最適化の研究に従事．電子情報通信学会，電気学会，IEEE，ACM 各会員．本会シニア会員．