Vol.2018-ARC-232 No.25
2018/8/1

マルチコア/GPU環境における階層統合型粗粒度タスク並列処理

渡辺 智之[1]　吉田 明正[1,2,a)]

概要：GPU を伴うマルチコアシステムの普及に伴い，GPU を有効利用して粗粒度タスク並列処理の性能を向上させることが期待されている．従来の階層統合型粗粒度タスク並列処理では，マルチコアシステムを対象とし，階層的に定義された各粗粒度タスクを，ダイナミックスケジューラが CPU コアに割り当てる方式を取っている．しかしながら，一部の処理時間の大きい粗粒度タスクは，プログラム全体の実行時間に大きな影響を及ぼす可能性がある．そこで，本稿では，CPU コアの処理能力では十分でない粗粒度タスクの実行に GPU を使用し，粗粒度タスクの実行時間の短縮を目指す．各粗粒度タスクは，CPU 実行あるいは GPU 実行がユーザにより指定されており，ダイナミックスケジューラによって適切な CPU コアあるいは GPU に割り当てられる．性能評価では，Intel Xeon E5-2680 および NVIDIA Tesla K80 からなるシステム上で，粒子法等のプログラムにより性能評価を行っており，提案手法の有効性を確認した．

1. はじめに

マルチコアシステムにおける並列処理手法として，ループ並列性に加えて，粗粒度タスク並列性 [1][2] を最大限に利用する階層統合型粗粒度並列手法 [1][3][4] が提案されている．階層統合型粗粒度タスク並列手法では，OpenMP あるいは Java マルチスレッド実装により，ダイナミックスケジューリングコードを含む並列コードを生成する方法をとる．

近年，GPU を伴うマルチコアシステムの普及に伴い，階層統合型実行制御により粗粒度タスク間の並列性を利用しつつ，処理時間の大きい粗粒度タスクを GPU に割り当て，実行時間を短縮するアプローチが考えられる．

本稿では，複数の GPU アクセラレータを伴うマルチコア環境において，ダイナミックスケジューリングを用いて粗粒度タスク間の並列性を利用しつつ，ユーザの指定した粗粒度タスクを任意 GPU または指定 GPU に割当て，高速化を実現する方法を提案する．性能評価では，2 台の NVIDIA Tesla K80 搭載並列システム上で，ヤコビ法及び粒子法のアプリケーションプログラムにより提案手法の有効性を示す．

本稿の構成は以下の通りとする．第 2 章では，階層統合型粗粒度並列処理について述べる．第 3 章では，マルチコア/GPU を対象としたマクロタスクスケジューリングを述べる．第 4 章では，マルチコア/GPU 環境における階層統合型粗粒度並列処理の性能評価を述べる．第 5 章では，まとめを述べる．

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

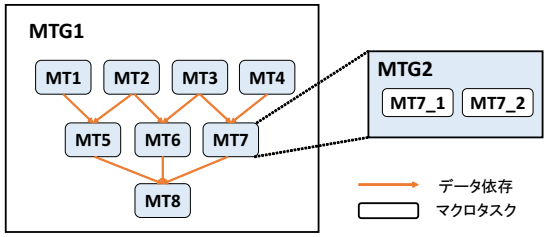

図 1　階層型マクロタスクグラフ（MTG）．

2. 階層統合型粗粒度並列処理

本章では，本手法の基盤として用いる階層統合型粗粒度並列処理について述べる．

2.1 階層統合粗粒度並列処理の概要

階層統合型粗粒度並列処理では，階層型マクロタスクグラフ（MTG）[5] を生成し，マクロタスク（MT）を階層的に定義する．その後，最早実行可能条件 [1] を満たした全階層のマクロタスクを，ダイナミックスケジューラが統一的にコアに割り当てて実行する．例えば，図 1 のような階層型マクロタスクグラフで表されるプログラムを 4 コアで実行したイメージは図 2 のようになる．この場合，複数階層の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．ここで，図 1 の MT8 の最早実行可能条件は

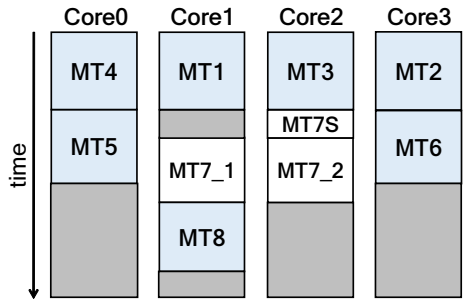

図 2 4 コア上での階層統合型粗粒度並列処理の実行イメージ.

MT5∧MT6∧MT7 と求めることができる．これは，MT5 と MT6 と MT7 の実行が終了した後に，MT8 の実行が可能になるということを表している．表 1 に示す各マクロタスクの最早実行可能条件は，図 1 の階層型マクロタスクグラフに対応している．

2.2 マクロタスクの階層的定義

粗粒度タスク並列処理では，まず，与えられたプログラム（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（MT）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（メソッド呼び出し）の 3 種類から構成される [1]．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んでいる場合は，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．階層統合型実行制御 [1] を適用する場合，全階層の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．第 L 階層マクロタスクをサブマクロタスクとして内部に持つ上位の第 $(L-1)$ 階層マクロタスクを，第 L 階層用の階層開始マクロタスクとして取り扱う．この階層開始マクロタスクは，内部の第 L 階層マクロタスクの実行を開始するために使用される．階層開始マクロタスクの導入により，当該階層のマクロタスクの実行が可能になったことが保証され，全階層のマクロタスクを同時に取り扱うことが可能となる．

本手法では，処理時間の大きいマクロタスクを事前に GPU 実行対象として決定しておき，それらのマクロタスクを GPU 上で実行するための GPU（CUDA）コードを用意しておく．

2.3 最早実行可能条件を用いたマクロタスクスケジューリング

マクロタスクの生成後，各階層におけるマクロタスク間の制御フローとデータ依存を解析し，階層型マクロタスクグラフ [5] を生成する．その後，制御依存とデータ依存を考慮したマクロタスク間並列性を最大限に引き出すため，各マクロタスクの最早実行可能条件 [5] を解析する．これらは，図 1 のようなマクロタスクグラフとして表現できる．

表 1 階層統合型実行制御の最早実行可能条件

MTG 番号	MT 番号	最早実行可能条件	終了通知
1	1	true	1
	2	true	2
	3	true	3
	4	true	4
	5	1∧2	5
	6	2∧3	6
	7†	3∧4	7S
	8	5∧6∧7	8
	9(EndMT)	8	9
2	71	7S	71
	72	7S	72
	73(ExitMT)	71∧72	73, 上位 MT(7)

†：メソッド内部の第 2 階層 MTG の階層開始 MT

表 1 のような最早実行可能条件は，マクロタスクの実行制御に用いられる．ここで，MT7 は図 1 に示すように MT71 と MT72 で構成されたメソッドの呼び出しに対応する．それゆえ，MT7 は階層開始マクロタスクとして動作しており，階層開始マクロタスクの処理を終了した時に 7S の終了通知を発行する．一方，MT7 のメソッド呼び出しの終了通知は，メソッド内の MT73 が終了通知 7 を発行する．ダイナミックスケジューリングの際には，ステート管理テーブルに保存された各マクロタスクの終了通知，分岐通知，最早実行可能条件を調べることにより，新たに実行可能なマクロタスクを検出することが可能となる [1]．階層統合型実行制御によるマクロタスクスケジューリングでは，各マクロタスクは最早実行可能条件を満たした後，レディキューに投入される．その後，レディキューから順に取り出されてコア（プロセッサ）に割り当てられ実行される．

3. マルチコア/GPU を対象としたマクロタスクスケジューリング

本章では，マルチコア/GPU を対象としてマクロタスクスケジューリングを行う際に実装されるレディキューの構成，分散型マクロタスクスケジューリング（各コアは MT 実行後にスケジューリングを行い自コアに割当てる）の実行手順，粗粒度並列コードについて述べる．

3.1 CPU キューと GPU キューの構成

階層統合型粗粒度並列処理をマルチコア/GPU 環境で実現するためには，前述のマクロタスクスケジューリングにおいて，各マクロタスクが最早実行可能条件を満たした後にレディキューに投入される際，GPU 実行対象のマクロタスクを管理するレディキューを用意する必要がある．提案手法では，CPU 実行対象マクロタスクが投入される CPU

表 2　CPU/GPU ダイナミックスケジューリングの実装方法.

	CPU への割当て	CPU キュー	GPU への割当て	個別 GPU キュー	共通 GPU キュー
任意 GPU 割当て	任意コア	○	任意 GPU	×	○
指定 GPU 割当て	任意コア	○	指定 GPU	○	×
任意・指定 GPU 割当て	任意コア	○	任意・指定 GPU	○	○

キュー，GPU 実行対象マクロタスクが投入される GPU キューを導入する．ここで，GPU キューは GPU 番号が指定されていない（任意 GPU 割当て）マクロタスクを投入するキュー（共通 GPU キュー）と GPU 番号が指定された（指定 GPU 割当て）マクロタスクを投入するキュー（個別 GPU キュー）の 2 種類を用意する．個別 GPU キューは使用する GPU の台数分用意する．

マクロタスクスケジューリングにおける GPU 割当て方式の比較を表 2 に示す．任意 GPU 割当てを伴うマクロタスクスケジューリングはレディキューとして共通 GPU キューと CPU キューを用いて，任意 GPU と任意コアへマクロタスクを割当てる．一方，指定 GPU 割当てを伴うマクロタスクスケジューリングは個別 GPU キューと CPU キューを用いて，指定 GPU と任意コアに割当てる．また，任意 GPU 割当てと指定 GPU 割当てを併用したスケジューリングも可能である．

3.2　任意 GPU 割当てを伴うマクロタスクスケジューリング

任意 GPU 割当てを伴うマクロタスクスケジューリングの割当て方式と使用するレディキューは，表 2 のようになる．

例として図 3 (a) のマクロタスクグラフを 4 コア＋ 4GPU のシステムで実行する場合を取り上げる．MT1 の実行が終了した段階で，レディキューは図 3 (b) 左側のようになっている．この時，図 3 (c) の Core0 で動作しているスケジューラは，MT3 を共通 GPU キューから取り出して GPU0 で実行する．その後，Core1，Core2 は共通 GPU キューから MT4，MT5 をそれぞれ取り出して，GPU1，GPU2 で実行する．

次に MT2 の実行が終了した段階で，レディキューは図 3 (b) 右側のようになっている．この時，図 3 (c) の Core3 で動作しているスケジューラは，MT7 を GPU キューから取り出して GPU3 で実行する．但し，Core3 は GPU3 の終了まで待機している．その後，Core2 と Core1 は，それぞれ MT6 と MT8 を実行する．このように 2 つのレディキュー（CPU 用，GPU 用）にマクロタスクが投入されている場合，GPU キュー即ち共通 GPU キューを優先する．

3.3　指定 GPU 割当てを伴うマクロタスクスケジューリング

指定 GPU 割当てを伴うマクロタスクスケジューリング

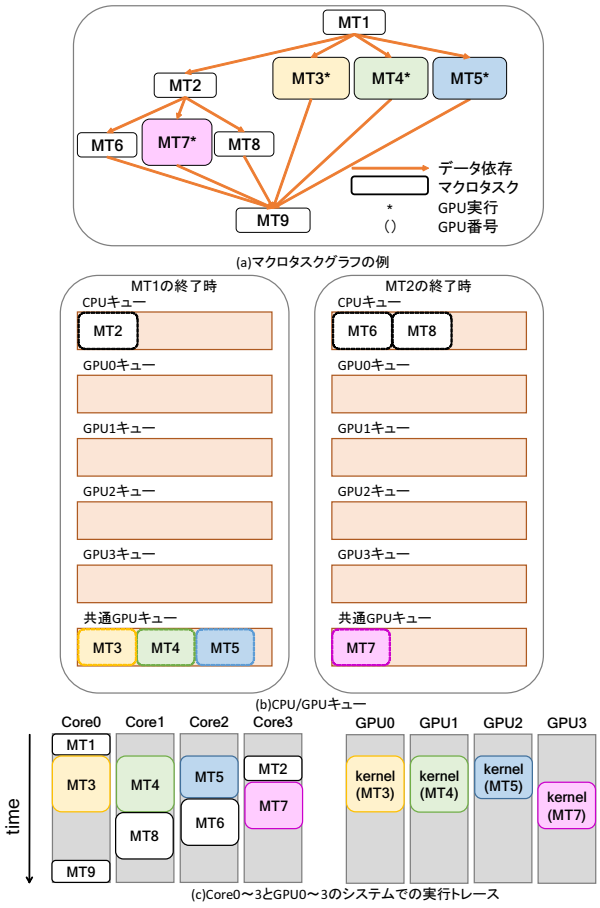

図 3　任意 GPU 割当てによるマクロタスクスケジューリング.

の割当て方式と使用するレディキューは，表 2 のようになる．この実装では，指定 GPU のデバイスメモリにデータ保持が可能であり，GPU と CPU のデータ転送を省略することが可能になる．

例えば，図 4 (a) のマクロタスクグラフを 4 コア＋ 4GPU のシステムで実行する場合を取り上げる．MT1 の実行が終了した段階で，レディキューは図 4 (b) 左側のようになっている．この時，図 4 (c) の Core0 で動作しているスケジューラは，MT3 を GPU0 キューから取り出して GPU0 で実行する．但し，Core0 は GPU0 の MT3 終了まで待機している．その後，Core1，Core2，Core3 は MT4，MT5，MT2 を実行する．

次に，MT2 の実行が終了した段階でレディキューは図 4 (b) 右側のようになっている．この時，図 4 (c) の Core3 で動作しているスケジューラは，全てのレディキューを確認する．この際，図 4 (b) 右側の GPU0 キューに MT7

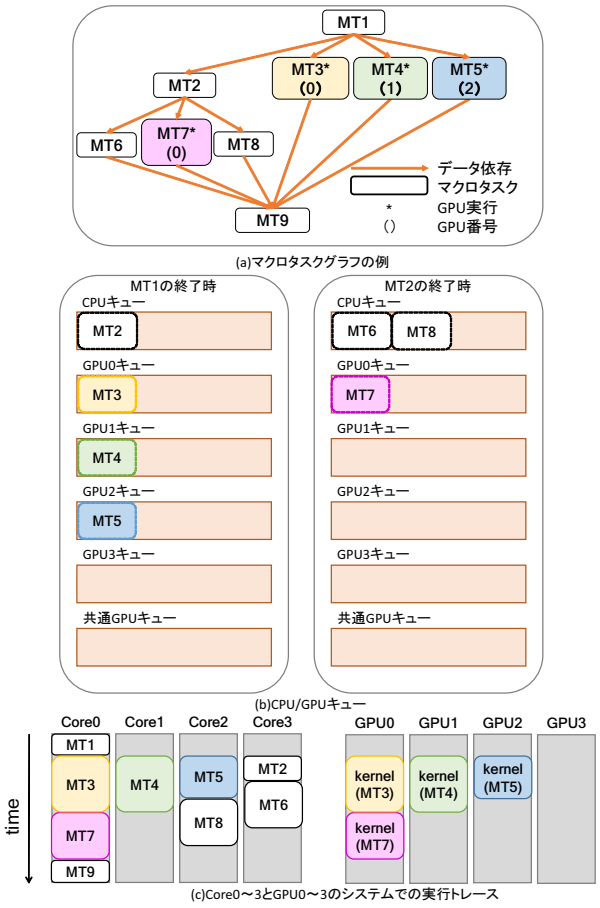

図 4 指定 GPU 割当てによるマクロタスクスケジューリング.

```
01:  #define BLOCK   //GPUのブロック数
02:  #define THREAD  //GPUのスレッド数
03:  #define CORE    //CPUのコア数
04:  スケジューラ用変数の宣言;
05:  MT間共有変数の宣言;
06:  /*kernel関数*/
07:  void kernel(){
08:      GPUコード;
09:  }
10:  /*MT7のコード (GPU指定の場合) */
11:  void MT7(){
12:
13:      cudaSetDevice(dev);  //GPUの実行デバイスをdevに設定
14:      cudaMemcpy(&GPU変数, &CPU変数, …);  //CPUからGPUへメモリ転送
15:      kernel<<<BLOCK, THREAD>>>();  //kernel関数呼び出し
16:      cudaMemcpy(&CPU変数, &GPU変数, …);  //GPUからCPUへメモリ転送
17:      cudaThreadSynchronize();  //スレッド同期
18:  }
19:  …
20:  void MT8(){
21:      CPUコード;
22:  }
23:  …
24:  /*最早実行可能条件*/
25:  void EEC(int mt){
26:      mtの最早実行可能条件をチェック;
27:  }
28:  /*ダイナミックスケジューラ*/
29:  void SCHEDULER(){
30:      while (全MTが終了するまで){
31:          if(GPU0がアイドル状態&&GPU0キューの投入MT数>=1)
32:              GPU0キューからMTを取り出す;
33:          else if(GPU1がアイドル状態&&GPU1キューの投入MT数>=1)
34:              GPU1キューからMTを取り出す;
35:          else if(GPU2がアイドル状態&&GPU2キューの投入MT数>=1)
36:              GPU2キューからMTを取り出す;
37:          else if(GPU3がアイドル状態&&GPU3キューの投入MT数>=1)
38:              GPU3キューからMTを取り出す;
39:          else if(アイドルGPU数>=1&&共通GPUキューの投入MT数>=1){
40:              GPUキューからMTを取り出す;
41:              アイドルGPUからGPUを選択;
42:          }else if(CPUキューの投入MT数>=1)
43:              CPUキューからMTを取り出す;
44:          if (取り出したMTの属性==GPU){
45:              GPUでMTを実行する;
46:              MTの終了・分岐通知;
47:          }else{
48:              本コアでMTを実行する;
49:              MTの終了・分岐通知;
50:          }
51:          各MTのEECを満たしたらMTをキューに投入;
52:      }
53:  }
54:  /*main関数*/
55:  void main(){
56:      データの初期化;
57:      #pragma omp parallel
58:      {
59:          SCHEDULER(omp_get_thread_num());
60:      }
61:  }
```

図 5 OpenMP/CUDA 実装による並列コード.

が投入されているが，GPU0 は MT3 を実行しているので MT7 を取り出すことはできない．よって，Core3 のスケジューラが，MT6 を CPU キューから取り出して Core3 で実行する．同様に Core2 のスケジューラは MT5 が終了した段階で MT8 を CPU キューから取り出し，実行する．その後，MT3 の実行が終了した段階で Core0 で動作しているスケジューラは，MT7 を GPU0 キューから取り出して GPU0 で実行する．但し，Core0 は GPU0 の終了まで待機している．このように 2 種類のレディキュー（CPU 用，GPU 用）にマクロタスクが投入されている場合，個別 GPU キュー，共通 GPU キュー CPU キューの順に優先度をつけて，キューからマクロタスクの取り出しを行う．

3.4 OpenMP/CUDA 実装による粗粒度並列処理コード

マルチコア/GPU 環境における階層統合型粗粒度並列処理コードの構成を図 5 に示す．

3.4.1 OpenMP によるマクロタスクスケジューリングコード

図 5 の main() 関数では，まず，OpenMP の指示文によりコア数分のスレッドを生成し，SCHEDULER() 関数を実行する．この関数ではマクロタスクの最早実行可能条件を EEC() 関数（図 5 の 25 行目）で確認した後に，51 行目で最早実行可能条件を満たすマクロタスクをレディキューへ投入する．一方，31 行目から 38 行目は指定 GPU 割当て対象のマクロタスクを，個別 GPU キュー（GPU0，GPU1，GPU2，GPU3 キュー）から取り出している．また，39 行目から 41 行目は，任意 GPU 割当て対象のマクロタスクを，共通 GPU キューから取り出しており，42 行目で CPU 実行対象のマクロタスクを，CPU キューから取り出す．最後に，44 行目から 49 行目でレディキューから取り出したマクロタスクに対応する関数（マクロタスクコード）を実行する．これらの一連の手順によって，全てのマクロタスクが実行される．

3.4.2 CUDA によるマクロタスク処理コード

GPU 実行をするマクロタスクの処理コードは，図 5 の 11 行目の MT7() のように記述される．この関数では，13 行目で cudaSetDevice() により GPU デバイスの指定を行い，15 行目で GPU のブロックとスレッド数を指定して 7 行目の kernel 関数を実行し，14 行目と 16 行目でホストメモリとデバイスメモリ間の転送を行う．これは，cudaMemcpy() を

表 3　並列システム Dell PowerEdge R730 の構成.

CPU	Intel Xeon E5-2680 v3, 2.5GHz，12 コア× 2 個
メモリ	64GB
GPU	NVIDIA Tesla K80 × 2 個（GK210 × 4）
OS	CentOS 6.9
処理系	GCC 4.4.7，CUDA Toolkit 9.1

用いて実装する．最後に，17 行目で cudaThreadSynchronize() でスレッド同期を行う．

3.4.3　ホストメモリとデバイスメモリの転送コード

図 5 の 14 行目で，GPU での演算に必要なデータを CPU から GPU に転送する．その後，16 行目で CPU で必要となるデータを GPU から CPU に転送している．任意 GPU 割当てを伴うマクロタスクスケジューリングの際，GPU 実行対象のマクロタスク処理コードでは，ホストメモリとデバイスメモリの転送コードは，kernel() 関数の前処理と後処理として，cudaMemcpy() により実装される．

一方，指定 GPU 割当てを伴うマクロタスクスケジューリングの際には，どの GPU で実行するかユーザが指定するため，指定された GPU のデバイスメモリ上にデータを保持することが可能であり，ホストメモリとデバイスメモリの転送コードは最小限に抑えられる．

4.　マルチコア/GPU 環境における階層統合型粗粒度並列処理の性能評価

本章では，GPU 搭載のマルチコアサーバにおいて連立 1 次方程式反復解法のヤコビ法プログラム，流体解析の 1 手法である粒子法プログラムを用いて性能評価を行う．

4.1　Tesla K80 搭載マルチコアサーバの構成

本性能評価では，表 3 に示す Dell PowerEdge R730 サーバで並列実行を行う．本サーバは，Intel Xeon E5-2680（12 コア）の CPU を 2 個，Tesla K80 を 2 個，メモリ 64GB を搭載している [6]．各 Tesla K80 には 2496CUDA コアからなる GK210 を 2 台搭載しており，本サーバでは GK210 デバイスを 4 台使用することが可能である．OS は CentOS6.9，処理系は GCC4.4.7，CUDA Toolkit 9.1 である．

4.2　ヤコビ法プログラムを用いた任意 GPU 割当ての性能評価

性能評価プログラムとして連立 1 次方程式反復解法のヤコビ法を用いた．ヤコビ法の反復計算は，収束条件を満たすまで繰り返され，行列サイズは 40,000 × 40,000 とした．ヤコビ法プログラムのマクロタスクグラフを図 6 に示す．本性能評価では，18 個のマクロタスクからなるヤコビ法プログラムで性能評価を行っている．また，GPU 実行対象とするマクロタスクは予めユーザが指定する．本性能評価では収束ループ内において，逐次処理時間の大きいマクロタスク（MT1_1，MT1_2，MT1_3，MT1_4）を GPU 実行対象とした．本性能評価では，任意 GPU 割当てを伴うマクロタスクスケジューリングを用いる．TeslaK80 搭載マルチコアサーバで実行した結果は図 7 に示す通りである．

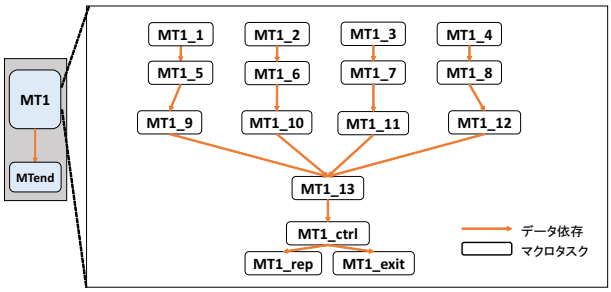

図 6　ヤコビ法プログラムのマクロタスクグラフ.

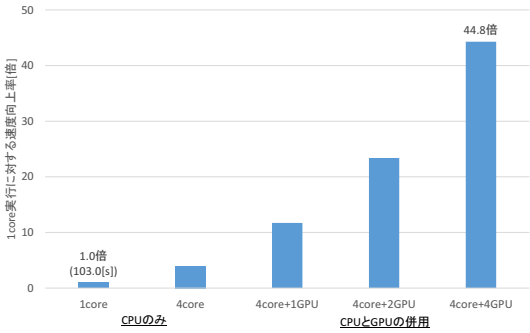

図 7　ヤコビ法プログラムによる任意 GPU 割当ての性能評価.

まず，CPU の 1 コアの実行時間は 103.0[s]，CPU の 4 コアによる実行時間は 25.8[s] となり 4.0 倍の速度向上が得られている．次に，処理時間の大きいマクロタスクに CPU 実行ではなく GPU 実行を適用する．GPU 実行には 4 台の GPU（GK210）を使用する．4 コア＋ 1GPU で並列実行を行うと実行時間は 8.8[s]，4 コア＋ 2GPU の場合の実行時間は 4.4[s]，4 コア＋ 4GPU で 2.3[s] という実行結果となり，これらの結果は 1 コアのみの場合と比べて 11.7 倍，23.4 倍，44.8 倍の速度向上が得られている．各 GPU デバイスでは 2496CUDA コアが利用可能であり，kernel() 関数の実行には 10 ブロック× 1000 スレッドを指定している．

4.3　粒子法プログラムを用いた指定 GPU 割当ての性能評価

次に，粒子法プログラム [7] を用いて，指定 GPU 割当ての性能評価を行う．粒子法は計算対象の流体を複数の粒子の集まりとして表し，数値的に解くための離散化手法の一つである．

粒子法プログラムのマクロタスクグラフは図 8 の通りであり，46 個のマクロタスクから構成される．処理時間の大きいマクロタスク，計 20 個を GPU 実行対象とした．本性能評価では，指定 GPU 割当てを伴うマクロタスクスケジューリングを採用しており，同一データを使用する MT を同一 GPU に指定する．具体的には，

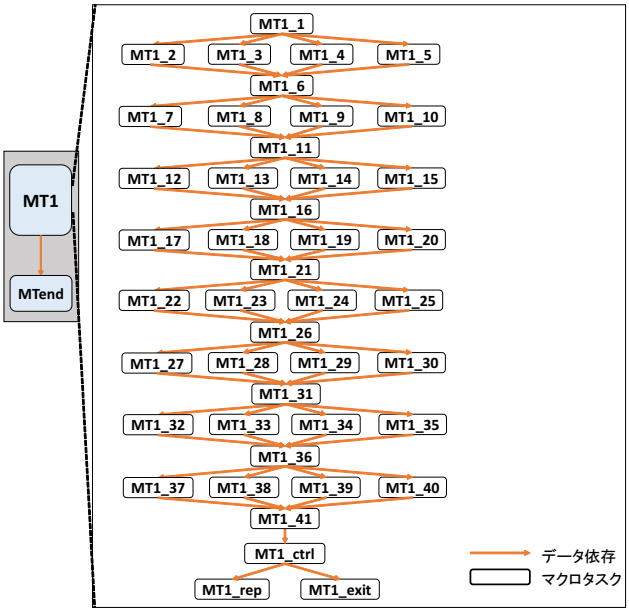

図 8 粒子法プログラムのマクロタスクグラフ.

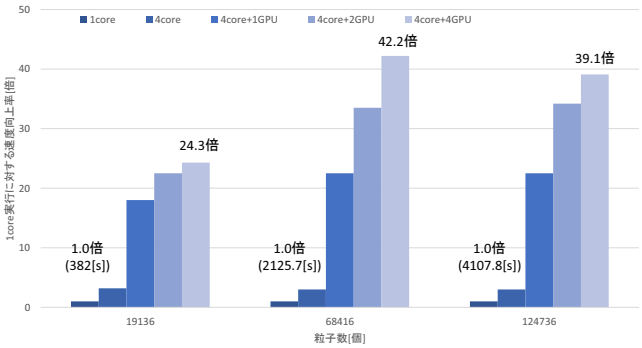

図 9 粒子法プログラムによる指定 GPU 割当ての性能評価.

MT1_2, MT1_12, MT1_22, MT1_27, MT1_37 を GPU0 とし, MT1_3, MT1_13, MT1_23, MT1_28, MT1_38 を GPU1 とし, MT1_4, MT1_14, MT1_24, MT1_29, MT1_39 を GPU2 とし, MT1_5, MT1_15, MT1_25, MT1_30, MT1_40 を GPU3 とした.

Tesla K80 搭載マルチコアサーバで実行した結果を図 9 に示す. 本性能評価では, 粒子数が 19,136 個, 68,416 個, 124,736 個の 3 種類で取り扱う. まず, 粒子数が 19,136 個の場合, CPU の 1 コアの実行時間は 382.0[s], CPU の 4 コアによる実行時間は 119.1[s] となり 3.2 倍の速度向上が確認できる. 次に処理時間の大きいマクロタスクに CPU 実行の代わりに GPU 実行を適用する. GPU 実行には GK210 からなる GPU を利用する. 4 コア＋ 1GPU による実行時間は 21.2[s] となり, 1 コアのみの場合と比べて 18.0 倍の速度向上が得られている. また, 4 コア＋ 2GPU では実行時間が 16.8[s], 4 コア＋ 4GPU で 15.7[s] という実行結果となり, これらの結果は 22.7 倍, 24.3 倍の速度向上が得られている.

次に, 粒子数が 68,416 個の場合では, CPU の 1 コアの実行時間は 2,125.7[s], 4 コア＋ 1GPU による実行時間は 94.6[s] (1 コア比 22.5 倍), 4 コア＋ 2GPU では 63.5[s] (1 コア比 33.5 倍), 4 コア＋ 4GPU で 50.4[s] (1 コア比 42.2 倍) となり, 最大の速度向上率が得られている.

粒子数が 124,736 個の場合では, CPU の 1 コアの実行時間は 4,107.8[s], 4 コア＋ 1GPU による実行時間は 182.8[s] (1 コア比 22.5 倍), 4 コア＋ 2GPU では 120.0[s] (1 コア比 34.2 倍), 4 コア＋ 4GPU で 105.1[s] (1 コア比 39.1 倍) となった. 以上の結果から, 粒子法プログラムにおいて, 指定 GPU 割当てを伴う粗粒度並列処理の有効性が確認された.

5. おわりに

本稿では, マルチコア/GPU 環境における階層統合型粗粒度並列処理のための並列コードの生成手法を提案した.

提案手法は, 任意 GPU 割当て及び指定 GPU 割当てを伴うマクロタスクスケジューリングを実現しており, その並列コードは OpenMP/CUDA 処理系を用いて実装される.

ヤコビ法プログラムにおける性能評価では, Xeon E5-2680 の 4 コアと Tesla K80 の 4GPU からなるサーバ上で実行したところ, 最大で 44.8 倍の速度向上が得られた. また, 粒子法プログラムでは, 4 コアと 4GPU で実行したところ, 42.2 倍の速度向上が得られ, 本手法の有効性が確認された.

今後の課題としては, 提案するマルチコア/GPU 環境に対応した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の開発が挙げられる.

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] 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.
[2] 林明宏, 和田康孝, 渡辺岳志, 関口威, 間瀬正啓, 白子準, 木村啓二, 笠原博徳: ヘテロジニアスマルチコア向けソフトウェア開発フレームワークおよび API, 情報処理学会論文誌, Vol.5, No.1 pp.68-79, 2012.
[3] Yoshida, A., Ochi, Y., Yamanouchi, N.: 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.
[4] Yoshida, A., Kamiyama, A., Oka, H.: A Task-Driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform, IPSJ Transactions on Advanced Computing Systems, Vol.10, No.1 pp.1-12, 2017.
[5] 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.
[6] NVIDIA: TESLA K80 GPU ACCELERATOR, https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf, 2015.

[7] 越塚誠一，柴田和也，室谷浩平: 粒子法入門，丸善出版株式会社，2014.