情報処理学会論文誌　コンピューティングシステム　Vol.13　No.3　1–12 (Nov. 2020)

マルチGPU環境における ハイブリッド粗粒度タスク並列処理

渡辺 智之[1,†1]　吉田 明正[1,2,3,a)]

受付日 2020年3月26日, 採録日 2020年7月8日

概要：高性能な GPU を複数搭載したマルチコアシステムの普及にともない，ループや関数間の並列性を利用する粗粒度タスク並列処理において，GPU アクセラレータを用いた高速化が期待されている．本論文の基盤とする階層統合型粗粒度タスク並列処理では，従来粗粒度タスクをマルチコアのみで実行する方式を採用してきたが，実行対象とするアプリケーションプログラムによっては，一部の粗粒度タスクの実行に膨大な時間を要しており，並列処理時間の短縮を妨げてきた．そこで，本論文ではマルチコアの処理能力では十分ではない粗粒度タスクに対して GPU を活用し，並列処理時間の短縮を実現するハイブリッド粗粒度タスク並列処理手法を提案する．加えて，本論文では提案するハイブリッド粗粒度タスク並列処理の並列コードを自動的に生成するために，LLVM/Clang を用いた並列化コンパイラを開発した．本並列化コンパイラは，並列化指示文付き C プログラムをソースプログラムとし，OpenMP と CUDA をともなうハイブリッド粗粒度タスク並列処理コードを生成する．性能評価では，NVIDIA Tesla K80 を搭載した Intel Xeon サーバを使用し，ヤコビ法プログラムの 24 コア・4 GPU 実行において 52.59 倍，粒子法プログラムの 4 コア・4 GPU 実行において 45.67 倍の速度向上を得た．これらの結果から提案手法の有効性が確認された．

キーワード：マルチ GPU，ハイブリッド粗粒度タスク並列処理，並列化コンパイラ，LLVM，Clang．

Hybrid Coarse Grain Task Parallelization on Multi-GPU

Tomoyuki Watanabe[1,†1]　Akimasa Yoshida[1,2,3,a)]

Received: March 26, 2020, Accepted: July 8, 2020

Abstract: With the spread of multicore systems with multi-GPU, in the coarse-grain task parallel processing that extracts parallelism among loops and functions, it is expected to enhance the performance by GPU-accelerators. Though the layer-unified coarse grain task, which is used as our platform, has targeted on only multicore, a certain application program demanded vast execution time for several coarse grain tasks and was prevented from reducing parallel execution time. Therefore, this paper proposes the hybrid coarse grain task parallelization scheme that utilizes GPUs for coarse grain tasks whose multicore performance is insufficient. Moreover, in order to generate hybrid coarse grain parallel codes automatically, this paper developed the parallelizing compiler that is implemented by using LLVM/Clang. This parallelizing compiler treats the C program with parallelizing directives as a source program, and generates the hybrid coarse grain task parallel processing code including OpenMP and CUDA. In the performance evaluation on Intel Xeon server equipped with NVIDIA Tesla K80, the Jacobi method program attained 52.59 times faster speedup and the particle method program attained 45.67 times faster speedup compared to sequential processing. From these results, effectiveness of the proposed scheme was confirmed.

Keywords: multi-GPU, hybrid coarse grain task parallelization, parallelizing compiler, LLVM, Clang

1. はじめに

マルチコアシステム上での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] を利用する階層統合型粗粒度タスク並列処理 [1], [3] が提案されている．階層統合型粗粒度タスク並列処理の並列コードは，現在までに Fortran OpenMP による実装 [1]，Java Thread による実装 [3]，Java Fork/Join Framework による実装 [4] が提案されている．

最近では，GPU の計算性能の飛躍的な進歩により，スーパーコンピュータからサーバに至るまで，GPU をマルチコアシステムのアクセラレータとして利用するようになってきており，ヘテロジニアスな並列システムにおける研究がさかんに行われている．たとえば，アクセラレータとして DRP コアを利用するソフトウェア開発フレームワークと API の研究 [5]，PGAS モデルにおいて OpenMP 拡張指示文を用いてタスクとデータの分割および複数 GPU へのマップを指示文で指定する研究 [6]，CPU・GPU コアが単一の仮想共有メモリにアクセスするプログラミングモデルを対象にコンパイラによるデータ転送コード生成に関する研究 [7]，OpenACC アプリケーションをマルチ GPU で実現するためのソース・トゥ・ソースコンパイラの研究 [8]，流体モデリングの SPH シミュレーションにおけるマルチ GPU 上での負荷バランス化と同期オーバヘッド軽減の研究 [9] があげられる．これらの関連研究は，本手法の対象とする階層統合型粗粒度タスク並列処理をベースとしたマルチ GPU 向け並列コードを並列化コンパイラで生成する方法とは異なる．

本論文では，マルチ GPU 環境，すなわち単一ノードからなるマルチコア/マルチ GPU システムにおいて，ダイナミックスケジューリングを用いた階層統合型粗粒度タスク並列処理と GPU によるループ並列処理を組み合わせたハイブリッド粗粒度タスク並列処理手法を提案する．本手法では，処理時間の大きい粗粒度タスクを GPU に割り当て，粗粒度タスク内部を GPU の CUDA コアを用いて並列処理することにより，大幅な実行時間短縮を実現する．

加えて，本論文では提案するハイブリッド粗粒度タスク並列処理の並列コードを自動生成するために，LLVM/Clang [10], [11] をベースとした並列化コンパイラを開発した．本並列化コンパイラは，並列化指示文を付加した C プログラムをソースプログラムとすることで，OpenMP/CUDA をともなう並列 C コードを自動生成することが可能である．本並列化コンパイラにより生成された並列 C コードは，マルチ GPU 環境においてハイブリッド粗粒度タスク並列処理を実現しており，NVIDIA Tesla K80 搭載 Intel Xeon サーバ上での性能評価から，高い実効性能を達成することが確認されている．

本論文の構成は以下のとおりとする．2 章では関連研究について述べる．3 章ではマルチコアシステム上での階層統合型粗粒度タスク並列処理について述べる．4 章ではマルチ GPU 環境におけるハイブリッド粗粒度タスク並列処理の実装手法について述べる．5 章では，ハイブリッド粗粒度タスク並列処理のための並列化コンパイラについて述べる．6 章では，Tesla K80 搭載 Xeon サーバにおける性能評価について述べる．7 章でまとめを述べる．

2. 関連研究

本章では，GPU 環境におけるタスク並列処理の関連研究について述べる．StarPU [12], [13] は，ヘテロジニアスな並列システムにおいて，コアあるいはアクセラレータで実行される抽象的なタスクを codelet と定義し，1 つのアプリケーションはデータ依存関係にある codelet 集合として記述される．この方式では，ユーザは codelet と呼ぶ構造体を関数ごとに記述する必要があり，タスク粒度は関数レベルとなる．それに対して，本論文では独自の構造体を用いることなく，並列化指示文を挿入するだけで，複数階層のタスクをヘテロジニアスな並列システム上で並列実行できる点で異なる．

OmpSs [14], [15] はヘテロジニアスなマルチコアアーキテクチャにおける指示文ベースのプログラミングモデルである．OmpSs の Runtime は，Xeon Phi，GPU，FPGA などのアクセラレータを対象としており，オフローディングするタスクを helper スレッドにより管理する．このようなモデルは，HPC システムにおけるプログラムの高生産性を目指したものであり，本論文のように低オーバヘッドなダイナミックスケジューリングを用いてマルチコア/マルチ GPU 上で複数階層のタスク並列性を利用するアプローチとは異なる．

Kokkos [16] の C++ライブラリは，さまざまなメニーコアアーキテクチャ上でアプリケーションの性能可搬性を目指しており，細粒度データ並列性とメモリアクセスパターンを抽象化して統一的に管理している．具体的には，計算をメニーコアデバイスに割り当て，多次元配列をポリモーフィックレイアウトで管理する．Kokkos の配列は View と呼ばれる C++テンプレートで実装されており，本論文が対象としているような C プログラムに並列化指示文を加えて粗粒度タスク並列処理を実現するアプローチとは異なる．

[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，電源開発株式会社
Presently with Electric Power Development Co., LTD.
[a)] akimasay@meiji.ac.jp

マルチ GPU 環境向けタスク並列処理の機能に関して，OpenMP [17] は OpenMP4.0 において dependency-aware なタスク並列モデルが導入されており，OpenMP4.5 では，GPU を含むアクセラレータをサポートしており，データとタスクをアクセラレータにオフロードすることができる．マルチ GPU 環境においても teams と distribute の指示文を使うことにより実装可能である．しかしながら，マルチ GPU 環境で高度にタスク並列性を利用するプログラムを OpenMP で作成することはユーザの負担が大きく，本論文のような並列化コンパイラによる並列コード生成のアプローチが期待されている．

OpenACC2.5 [18] では，parallel 構文を用いてユーザ責任で高度に並列化を行う方法，あるいは，kernels 構文を用いてコンパイラにより並列化を行う方法がある．GPU へのオフロードをともなう高度な記述は，parallel，data，loop の構文を利用して可能である．単一ノード内のマルチ GPU 計算は OpenACC のみで記述可能であるが，複数ノードの場合には MPI を併用する必要がある．この方法は，本論文のようにダイナミックスケジューリング環境下で複数階層の粗粒度タスクをマルチコア/マルチ GPU に割り当てるアプローチとは異なる．

3. マルチコアシステム上での階層統合型粗粒度タスク並列処理

本章では，本手法の基盤として用いる階層統合型粗粒度タスク並列処理について述べる．

3.1 階層統合型粗粒度タスク並列処理の概念

階層統合型粗粒度タスク並列処理 [1] では，対象プログラムに対して，ループ（繰り返しブロック），関数（サブルーチンブロック），基本ブロックからなる 3 種類のマクロタスク（MT）を階層的に定義し，データ依存と制御依存を表した階層型マクロタスクグラフを生成する．その後，全階層のマクロタスクのなかで最早実行可能条件 [1] を満たしたものを，ダイナミックスケジューラが統合的管理の下でコアに割り当て実行する．たとえば，**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 コアの CPU 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分かる．ここで，図 1 の MT8 の最早実行可能条件は MT5∧MT6∧MT7 と表現され，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．MT8 は MT8_1 と MT8_2 を内部に含む繰り返しブロック（2 回転の do-while 文）を表しており，図 2 においては，MT8_1 と MT8_2 がそれぞれ 2 回実行される．なお，**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．

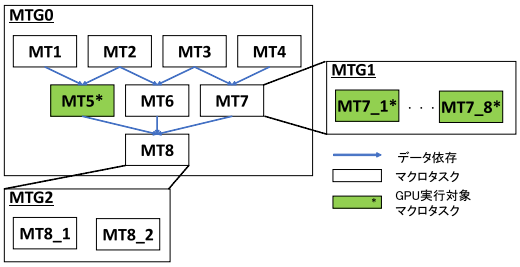

図 1　階層型マクロタスクグラフ（MTG）
Fig. 1　Hierarchical macro-task-graph.

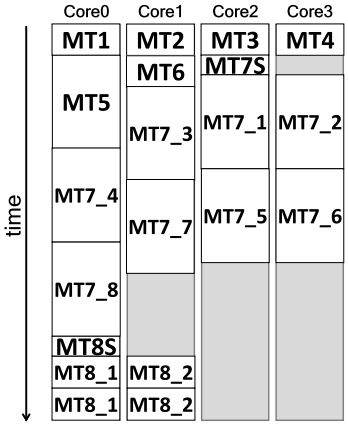

図 2　マルチコアのみにおけるマクロタスクのコア割当て
Fig. 2　Assignment of macrotasks on multicore.

3.2 マクロタスクの階層的定義

階層統合型粗粒度タスク並列処理では，まず，ソースプログラムを第 1 階層マクロタスク（MT）に分割する．マクロタスクは，繰り返しブロック（for 文や while 文など），サブルーチンブロック（関数呼び出し），基本ブロックの 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．

3.3 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

マクロタスクの定義後，マクロタスク間の制御依存とデータ依存を解析して，表 1 のような最早実行可能条件を求める．最早実行可能条件は，図 1 のような階層型マクロタスクグラフ [2] で表現することができる．

表 1　階層統合型実行制御の最早実行可能条件
Table 1　Earliest-executable-conditions for layer-unified execution control.

MTG 番号	MT 番号	最早実行可能条件	終了通知
0	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	8S
	9(end)	8	9
1	7_1	7S	7_1
	7_2	7S	7_2
	7_3	7S	7_3
	7_4	7S	7_4
	7_5	7S	7_5
	7_6	7S	7_6
	7_7	7S	7_7
	7_8	7S	7_8
	7_9(exit)	7_1∧7_2∧7_3∧7_4 ∧7_5∧7_6∧7_7∧7_8	7_9, 7
2	8_1	8S	8_1
	8_2	8S	8_2
	8_3(ctrl)	8_1∧8_2	8_3
	8_4(rep)	$8\_3_{8\_4}$	8_4
	8_5(exit)	$8\_3_{8\_5}$	8_5，8

†：階層開始 MT
$8\_3_{8\_4}$：MT8_3(ctrl) が MT8_4(rep) に分岐かつ終了を表す
$8\_3_{8\_5}$：MT8_3(ctrl) が MT8_5(exit) に分岐かつ終了を表す

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．たとえば，MT7 は図 1 に示すように MT7_1 から MT7_8 で構成された関数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ときに 7S の終了通知を発行する．一方，MT7 の関数呼び出しの終了処理は，関数内の制御用 MT である MT7_9 の終了通知 7 の発行により実現される．図 1 では制御用 MT である 7S と MT7_9 を省略している．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マクロタスクが取り出されてコアに割り当てられ実行される．

4. マルチ GPU 環境におけるハイブリッド粗粒度タスク並列処理

本章では，ハイブリッド粗粒度タスク並列処理のプログラミングモデルと実行モデルについて述べる．

4.1 ハイブリッド粗粒度タスク並列処理のプログラミングモデル

マルチ GPU システムにおけるハイブリッド粗粒度タスク並列処理を実現する場合，入力 C プログラムにおいて**表 2** の並列化指示文を記述し，本並列化コンパイラで並列コードを自動生成する．

ここで，**図 3** の C プログラム例（図 1 に対応）を用いて並列化指示文を説明する．まず，マクロタスクの定義は 15 行目と 16 行目のように#pragma para mt(MT 番号) によって MT 番号とともに範囲を指定する．繰り返し文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，66 行目から 76 行目のように#pragma para mt(MT 番号) inner で範囲を指定する．これにより，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．各マクロタスクの最早実行可能条件は 26 行目のように#pragma mt eec(out:MT 番号) eec(in:論理式) を記述する．GPU 実行対象マクロタスクの定義は，27 行目から 33 行目のように#pragma para mt(MT 番号) kernel で範囲を指定する．その内部において，29 行目と 32 行目のように#pragma para copy(配列名，初期値，型，サイズ，HostToDevice |DeviceToHost) を記述することでデータ転送と転送方向を指定する．

4.2 ハイブリッド粗粒度タスク並列処理の実行モデル

ハイブリッド粗粒度タスク並列処理では，基盤として前述の階層統合型粗粒度タスク並列処理を用いる．CPU を対象とした従来の階層統合型粗粒度タスク並列処理は，階層型マクロタスクグラフ上の全マクロタスクを統一的に取り扱い，単一の CPU キューを用いてコアへの割当てを行っていた．すなわち，最下位層（内部にサブマクロタスクを持たないもの）のマクロタスクは CPU の 1 コア上で実行される．

提案するハイブリッド粗粒度タスク並列処理においても，各計算資源（コアあるい GPU）の負荷を均衡に保つため，ダイナミックスケジューリングを採用しており，共有データはホストのメモリに配置する．ユーザは表 2 の並列化指示文により処理時間の大きい最下位層のマクロタスク（リダクションループを除く並列化可能ループ）を GPU 割当て対象として指定することが可能であり，ダイナミックスケジューラは GPU 割当てを指示されたマクロタスクがレディになると，CPU キューとは別に用意された GPU キューにそのマクロタスクを投入する．投入された GPU

表 2　並列化コンパイラのための並列化指示文
Table 2　Parallelizing directives for parallelizing compiler.

表記	意味
#pragma para mt(MT 番号) {...}	MT 番号のマクロタスクの定義
#pragma para mt(MT 番号) inner {...}	内部にサブマクロタスクを含む MT 番号のマクロタスクの定義
#pragma para mt(MT 番号) kernel {...}	GPU 実行対象とする MT 番号のマクロタスクの定義
#pragma para eec(out:MT 番号) eec(in:論理式)	out:で MT 番号を指定し，最早実行可能条件の論理式を in:で指定する．
#pragma para copy(変数名，初期値，型，サイズ， HostToDevice|DeviceToHost)	GPU で使用するデータの転送と転送方向

キューのマクロタスクはアイドルコアによって取り出され，コアは並列化コンパイラが生成した CUDA コードをアイドル GPU で実行する．この CUDA コードには，ホストからデバイスへのデータ転送コード，GPU 実行のためのカーネルコード，デバイスからホストへのデータ転送コードが含まれている．

なお，処理時間の大きい最下位層のマクロタスク（並列化可能ループ）は，プログラム全体のクリティカルパス長に大きな影響を与えるため，これらのマクロタスクの GPU 割当てによる処理時間短縮は，プログラム全体のクリティカルパス長，すなわち実行時間を短縮するうえで大きな効果が期待できる．

4.2.1　スケジューリングのための CPU キューと GPU キュー

ハイブリッド粗粒度タスク並列処理をマルチコア/GPU 環境で実現するために，CPU 実行対象マクロタスクを扱う CPU キューと GPU 実行対象マクロタスクを扱う GPU キューの 2 種類をレディキューとして用意する．例として図 1 のマクロタスクグラフを 4 コア・4 GPU のシステムで実行する場合を取り上げる．MT1，MT2，MT3，MT4 の実行が終了した段階で，各レディキューは**図 4** (b) のようになっている．このとき，図 4 (a) の Core0 で動作しているスケジューラは，MT5 を GPU キューから取り出してアイドル状態の GPU0 で実行する．その後，Core1，Core2 は CPU キューから MT6，MT7S をそれぞれ取り出して，自コアで実行する．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 キューを優先することで GPU を効率的に利用することができる．

次に MT7S の実行が終了した段階で，レディキューは図 4 (c) のようになっている．このとき，図 4 (a) の Core2 で動作しているスケジューラは，MT7_1 を GPU キューから取り出してアイドル状態の GPU1 で実行する．ここで，コア数は GPU 数以上であることを前提として，GPU のカーネル呼び出しは同期的に行っており，Core2 は GPU1 の終了まで待機している．その後，Core3 のスケジューラは，MT7_2 をアイドル状態の GPU2 で実行する．これら一連の流れを終了条件を満たすまで繰り返す．

4.2.2　OpenMP によるマクロタスクスケジューリングコード

マルチコア/GPU 環境におけるハイブリッド粗粒度タスク並列処理コードの構成を**図 5** に示す．図 5 の 88 行目の 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 55 行目）で確認した後に，73 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，61 行目から 63 行目は，GPU 実行対象のマクロタスクを GPU キューから取り出しており，65 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，66 行目から 72 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，すべてのマクロタスクが実行される．

4.2.3　CUDA によるマクロタスク処理コードとホスト・デバイス間転送コード

GPU 実行対象のマクロタスク処理コードは，図 5 の 24 行目の MT5() のように記述される．この関数では，26 行目で cudaSetDevice() によりアイドル状態の GPU デバイスの指定を行い，28 行目で GPU のブロック数とスレッド数を指定して 10 行目で宣言された kernel() 関数を実行し，30 行目で cudaDeviceSynchronize() を用いてスレッド同期を行う．図 5 の 27 行目で，GPU 上の演算に必要なデータをホストからデバイスに転送する．その後，30 行目でホストで必要となるデータをデバイスから転送している．GPU 実行対象のマクロタスク処理コードでは，ホスト・デバイス間転送コードは，kernel() 関数の前処理と後処理として，cudaMemcpy() により実装される．

5.　ハイブリッド粗粒度タスク並列処理のための並列化コンパイラ

本章では，LLVM/Clang [10], [11] を用いて実装したハ

```
01: #define N 1024
02: int *array;
03: double *data;
04: int loop;
05: void init() {
06:   array = (int *)malloc(sizeof(double)*N);
07:   data = (double *)malloc(sizeof(double)*N);
08:   ...
09: }
10: ...
11: void func8_2() { ... loop++; }
12: int main() {
13:   init();
14:   #pragma para eec(out:1)
15:   #pragma para mt(1)
16:   { func1(); }
17:   #pragma para eec(out:2)
18:   #pragma para mt(2)
19:   { func2(); }
20:   #pragma para eec(out:3)
21:   #pragma para mt(3)
22:   { func3(); }
23:   #pragma para eec(out:4)
24:   #pragma para mt(4)
25:   { func4(); }
26:   #pragma para eec(out:5) eec(in:1&&2)
27:   #pragma para mt(5) kernel
28:   {
29:     #pragma para copy(array,0,int,N,HostToDevice)
30:     for(int i=0; i<N; i++)
31:     { array[i] = i; }
32:     #pragma para copy(array,0,int,N,DeviceToHost)
33:   }
34:   #pragma para eec(out:6) eec(in:2&&3)
35:   #pragma para mt(6)
36:   { func6(); }
37:   #pragma para eec(out:7) eec(in:3&&4)
38:   #pragma para mt(7) inner
39:   {
40:     #pragma para eec(out:7_1)
41:     #pragma para mt(7_1) kernel
42:     {
43:       #pragma para copy(data,0,double,N/8,HostToDevice)
44:       for(int i=0; i<N/8; i++)
45:       { data[i] = i; }
46:       #pragma para copy(data,0,double,N/8,DeviceToHost)
47:     }
48:     #pragma para eec(out:7_2)
49:     #pragma para mt(7_2) kernel
50:     {
51:       #pragma para copy(data,N/8,double,N/8,HostToDevice)
52:       for(int i=N/8; i<2*N/8; i++)
53:       { data[i] = i; }
54:       #pragma para copy(data,N/8,double,N/8,DeviceToHost)
55:     }
56:     ...
57:     #pragma para eec(out:7_8)
58:     #pragma para mt(7_8) kernel
59:       #pragma para copy(data,N/8,double,N/8,HostToDevice)
60:       for(int i=7*N/8; i<N; i++)
61:       { data[i] = i; }
62:       #pragma para copy(data,7*N/8,double,N/8,DeviceToHost)
63:     }
64:   }
65:   #pragma para eec(out:8) eec(in:5&&6&&7)
66:   #pragma para mt(8) inner
67:   {
68:     do{
69:       #pragma para eec(out:8_1)
70:       #pragma para mt(8_1)
71:       { func8_1(); }
72:       #pragma para eec(out:8_2)
73:       #pragma para mt(8_2)
74:       { func8_2(); }
75:     }while(loop<2);
76:   }
77:   return 0;
78: }
```

図 3　並列化指示文をともなう C プログラム
Fig. 3　C program including parallelizing directives.

イブリッド粗粒度タスク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．

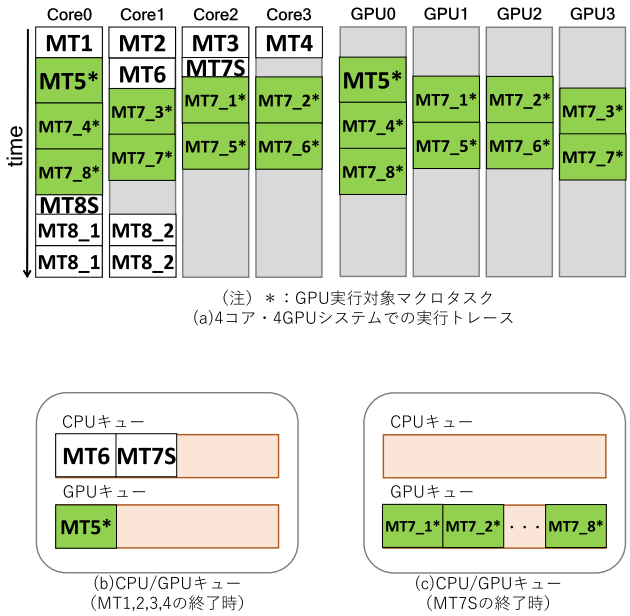

図 4　マルチコア/マルチ GPU におけるマクロタスクのコア/GPU 割当て
Fig. 4　Assignment of macrotasks on multicore/multi-GPU.

5.1　並列化コンパイラの仕様と LLVM/Clang による実装

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 C プログラムをソースプログラムとし，OpenMP/CUDA をともなうハイブリッド粗粒度タスク並列処理の並列コードを出力する．入力対象となる C プログラムは，本コンパイラの開発に用いた LLVM/Clang3.7.1 が対応している文法で記述されているものとする．また，本コンパイラでは，並列化指示文を加えた C コードは 1 つのファイルにまとめておく必要がある．一方，分割コンパイルを行う際に，並列化指示文を付加しない C プログラムは本コンパイラによって並列コードを生成する必要がない．マクロタスクの対象となる範囲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している．

本並列化コンパイラは LLVM/Clang を用いて C++言語で開発されており，その構成は**図 6** のとおりである．

LLVM はコンパイラ開発に必要なライブラリ，プラグインなどの一連のソフトウェア群である．Clang は LLVM のプロジェクトの 1 つであり，Objective-C や Objective-C++，C++，C 言語を対象とした LLVM のフロントエンドである．加えて，Clang はコンパイルだけでなく，Clang をライブラリとして利用することで機能の一部をユーザが開発したソフトウェアに受け渡すことができるインタフェース（LibTooling [11]）を有している．本論文では，LibTooling を用いることで Clang で行った字句解析から構文解析，意味解析，生成された抽象構文木の情報を受け取り，並列コードの生成を行う．このように，Clang に解析部分を完

```
01: #define N 1024
02: #define devMAX 4 //デバイス数
03: スケジューラ用変数の宣言;
04: MT間共有変数の宣言;
05: int *array;
06: int *d_array;
07: double *data;
08: double *d_data[devMAX];
09: int loop;
10: __global__ void kernel5(int *array){
11:  int i = 0 + blockIdx.x*blockDim.x + threadIdx.x;
12:   if( i < N ) {
13:     array[i] = i;
14:   }
15: }
16: __global__ void kernel7_1(double *data){ GPUコード; }
17: ...
18: __global__ void kernel7_8(double *data){ GPUコード; }
19: void MT1(){ ... }
20: void MT2(){ ... }
21: void MT3(){ ... }
22: void MT4(){ ... }
23: /*マクロタスクコード (GPU) */
24: void MT5(int dev){
25:   ...
26:   cudaSetDevice(dev);  //GPUの実行デバイスをdevに設定
27:   cudaMemcpy(d_array[dev], array, sizeof(int)*N, cudaMemcpyHostToDevice);
28:   kernel5<<<1, 1024>>>(d_array[dev]);
29:   cudaDeviceSynchronize();
30:   cudaMemcpy(array, d_array[dev], sizeof(int)*N, cudaMemcpyDeviceToHost);
31:   ...
32: }
33: void MT6(){ ... }
34: void MT7(){ 階層開始処理; }
35: void MT7_1(int dev){ ... }
36: ...
37: void MT7_8(int dev){ ... }
38: void MT8(){ 階層開始処理; }
39: void MT8_1(){ ... }
40: void MT8_2(){ ... loop++; }
41: void MT8_ctrl(){
42:   if(loop<2)
43:     MT8_repへ分岐通知;
44:   else
45:     MT8_exitへ分岐通知;
46: }
47: void MT8_rep(){
48:   MT8内のマクロタスクの終了通知, 分岐通知の初期化;
49: }
50: void MT8_exit(){
51:   MT8の終了通知;
52: }
53: ...
54: /*最早実行可能条件*/
55: void EEC(int mt){
56:   mtの最早実行可能条件をチェック;
57: }
58: /*ダイナミックスケジューラ*/
59: void SCHEDULER(int thread){
60:   while (全MTが終了するまで){
61:     if(アイドルGPU数>=1&&共通GPUキューの投入MT数>=1){
62:       GPUキューから1MTを取り出す;
63:       アイドルGPUから1GPUを選択;
64:     }else if(CPUキューの投入MT数>=1)
65:       CPUキューから1MTを取り出す;
66:     if (取り出したMTの属性==GPU){
67:       GPUでMTを実行する;
68:       MTの終了・分岐通知;
69:     }else{
70:       本コアでMTを実行する;
71:       MTの終了・分岐通知;
72:     }
73:     各MTのEECを満たしたらMTをキューに投入;
74:   }
75: }
76: void init(){
77:   /*ホストにデータ確保*/
78:   array = (int *)malloc(sizeof(double)*N);
79:   data = (double *)malloc(sizeof(double)*N);
80:   /*デバイスにデータ確保*/
81:   for(int i=0; i<devMAX; i++) {
82:     cudaSetDevice(i);
83:     cudaMalloc((void **)&d_array, sizeof(int)*N);
84:     cudaMalloc((void **)&d_data, sizeof(double)*N);
85:   }
86:   ホストとデバイスデータの初期化;
87: }
88: void main(){
89:   init();
90:   #pragma omp parallel
91:   {
92:     SCHEDULER(omp_get_thread_num());
93:   }
94:   return 0;
95: }
```

図 5　OpenMP/CUDA をともなうハイブリッド粗粒度タスク並列処理の並列コード

Fig. 5　Hybrid coarse-grain task parallel code using OpenMP/CUDA.

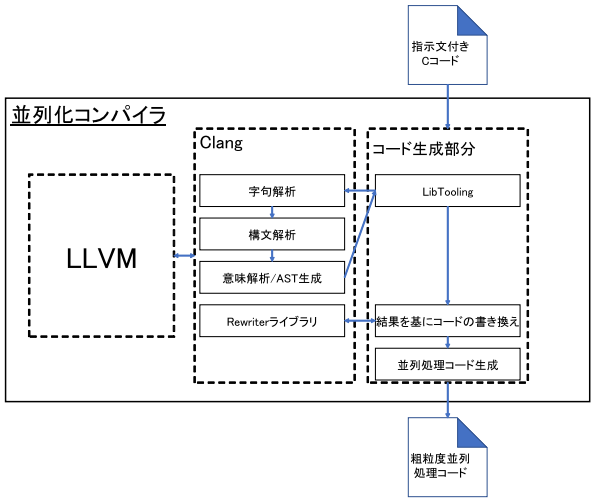

図 6　LLVM/Clang を用いた並列化コンパイラの構成

Fig. 6　Structure of parallelizing compiler using LLVM/Clang.

全に任せることで低コストで開発を進めることが可能になる．本並列化コンパイラでは Clang の Rewriter ライブラリを用いて，ソースプログラムから並列コードへの書き換えを広範囲で行っている．

並列化指示文を付加した C プログラムを入力すると，LibTooling を介することで Clang によって字句解析と構文解析が行われ，抽象構文木が生成される．構文解析では，字句解析の結果から並列化指示文の範囲であると認識すると，対象のステートメントノード（for 文や代入文など）や制御文などの情報を個別に収集する．次に，構文解析によって得られた情報から意味解析を行い，抽象構文木を生成する．これらの解析結果は Clang から LibTooling を介して受け取る．その後，Rewriter ライブラリを利用して並列コードの書き換えが行われる．

最後に，OpenMP/CUDA をともなうハイブリッド粗粒度タスク並列処理コードが生成される．マクロタスクの管理を行う変数やデバイスデータ領域の確保，初期化を追加した init() 関数，粗粒度並列処理に必要なコードを追加した main() 関数を出力するとともに，並列化指示文から収集した情報からマクロタスクやマクロタスクを実行するスケジューラ関数，最早実行可能条件を判定する EEC() 関数を出力する．なお，本コンパイラで生成された OpenMP/CUDA をともなう並列コードは，nvcc コンパイラでコンパイルして実行コードを生成することにより，マルチ GPU システム上でハイブリッド粗粒度タスク並列処理を実現することができる．

5.2　OpenMP/CUDA をともなう並列コードの生成

開発した並列化コンパイラは，マルチコア/GPU 環境におけるハイブリッド粗粒度タスク並列処理の並列コードを自動生成することが可能である．図 3 をソースプログラムとして並列化コンパイラに与えた場合，自動生成された並

表 3　性能評価プログラム
Table 3　Performance evaluation programs.

プログラムの特性	ヤコビ法（オリジナル）	ヤコビ法（24 コア・4 GPU 用）	粒子法
逐次実行のソースコード長（並列化指示文なし）[行]	158	577	1379
並列化指示文の行数	34	104	128
定義した MT 数	13	49	37
並列コード長（コンパイラにより生成）[行]	515	1146	2174

列コードは図 5 となり，対応する階層型マクロタスクグラフは図 1 のように表現される．

図 5 の 19 行目は，図 3 の 15 行目，16 行目で定義したマクロタスクである．ここで，図 3 における 66 行目の #pragma para mt(MT 番号) inner で定義したマクロタスクは，図 5 の 38 行目のような階層開始マクロタスクを生成した後に，図 3 の 75 行目の do-while 文の終了条件（loop<2）をコード生成の際に抽象構文木から抽出し，図 5 の 41 行目から 46 行目のマクロタスクの終了条件を判定するマクロタスクを生成する．加えて，47 行目から 49 行目に示す内部のマクロタスクの終了通知と分岐通知を制御するマクロタスク，および 50 行目から 52 行目に示す終了通知処理を行うマクロタスクを自動生成する．

図 5 において 10 行目から 15 行目の kernel() 関数の宣言では，図 3 の 30 行目の最外側ループである for 文を崩壊させた後，31 行目の処理を kernel() 関数の処理とする．for 文の初期値は図 5 の 11 行目の kernel() 関数のインデックス番号に対応する．そして，for 文の範囲（i<N）は図 5 の 12 行目のように設定することで for 文の 1 イタレーションを 1 スレッドとして実行する．また，kernel() 関数で扱う変数の内，kernel() 関数の内部で宣言された変数以外を抽象構文木から求め，引数として指定する．図 5 の 28 行目のようにマクロタスク内で kernel() 関数を呼び出す場合，本コンパイラでは 1 イタレーションを 1 スレッドとして実行するために，スレッド数はユーザが設定ファイルにより指定し，kernel() 関数の対象となったループのイタレーション数/スレッド数（割り切れない場合はイタレーション数/スレッド数+1）をブロック数とする．その後，cudaDeviceSynchronize() を行う．kernel 関数の前後には，cudaMemcpy() を挿入することで，ホスト・デバイス間のデータ転送を行う．この際，#pragma para copy() から情報を収集し，適切な配列にデータを転送する．

6. NVIDIA Tesla K80 搭載 Intel Xeon サーバにおける性能評価

本章では，GPU 搭載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，流体解析の一手法である粒子法プログラムを用いて性能評価を行う．各プログラムの特性は**表 3** に示すとおりである．

表 4　並列システム Dell PowerEdge R730 の構成
Table 4　Specification of parallel system Dell PowerEdge R730.

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

6.1　性能評価環境

本性能評価では，**表 4** 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，NVIDIA Tesla K80 を 2 個，メモリ 64 GB を搭載している [19]．各 Tesla K80 には 2496CUDA コアからなる GK210 を 2 個搭載しており，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，CUDAToolkit 9.1 である．

6.2　ヤコビ法プログラムを用いたマルチ GPU 環境における性能評価

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 40,960 × 40,960 としている．ヤコビ法プログラムの逐次実行のソースコード長は，表 3 に示すように並列化指示文なしで 158 行である．このソースコードに並列化指示文 34 行を追加したプログラムを入力対象とし，本並列化コンパイラでコンパイルを行うと，13 個の MT を有する並列コード（515 行）を生成することができる．ヤコビ法プログラムの階層型マクロタスクグラフは，**図 7** (a) のとおりであり，2 階層からなる 13 個のマクロタスクで構成される．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（MT2_2，MT2_3，MT2_4，MT2_5）を GPU 実行対象とした．

NVIDIA Tesla K80 搭載マルチコアサーバで実行した結果は**図 8** に示すとおりである．まず，CPU の逐次実行時間は 112.010 [s]，CPU の 4 コアによる実行時間は 28.300 [s] となり逐次比で 3.96 倍の速度向上が得られている．

次に，提案するハイブリッド粗粒度タスク並列処理により，処理時間の大きいマクロタスクに CPU 実行では

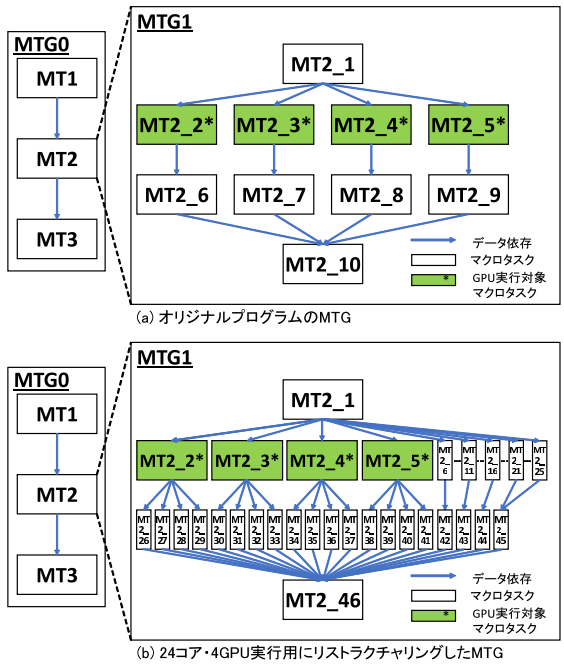

図 7　ヤコビ法プログラムの階層型マクロタスクグラフ

Fig. 7　Hierarchical macro-task-graph of Jacobi method program.

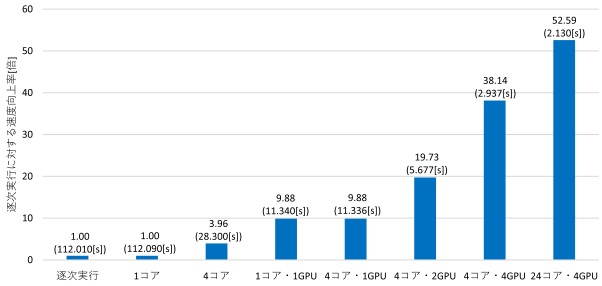

図 8　マルチ GPU 上でのヤコビ法プログラムによる性能評価

Fig. 8　Performance evaluation using Jacobi method program on multi-GPU.

表 5　1 コア・1 GPU 上でのヤコビ法プログラムの実行時間

Table 5　Execution time of Jacobi method program on one core and one GPU.

ブロック数	スレッド数	実行時間 [s]
80	128	12.520
40	256	13.111
20	512	11.340
10	1,024	15.190

なく GPU 実行を適用する．単一 GPU（すなわち 1 台の GK210）を使用した場合，2496CUDA コアが利用可能である．ここで，ブロック数とスレッド数を変更した場合の 1 コア・1 GPU における実行時間の比較を**表 5** に示す．ブロック数 × スレッド数 ≥40,960/4 に設定されており，表 5 から実行時間が短くなるブロック数・スレッド数の組合せは 20 ブロック・512 スレッドであるため，kernel() 関数の実行には 20 ブロック・512 スレッドを指定する．単

表 6　ヤコビ法における従来手法との性能比較

Table 6　Performance comparison with conventional schemes for Jacobi method.

実行方法	コア数	GPU 数	実行時間 [s]	速度向上率 [倍]（逐次比）
OpenMP	24	0	4.778	23.44
OpenMP・CUDA	24	4	2.991	37.45
提案手法	24	4	2.130	52.59

一 GPU 実行（1 コア・1 GPU 実行）の場合，実行時間は 11.340 [s] であり，逐次比で 9.88 倍の速度向上が得られた．

マルチコア/マルチ GPU を利用すると，4 コア・4 GPU による実行時間は 2.937 [s] であり，逐次比で 38.14 倍の速度向上が得られている．さらに，計算資源を最大限に利用するために，図 7 (a) の MTG を図 7 (b) のように，GPU と CPU の処理能力を考慮してリストラクチャリング（GPU で実行される MT2_2～MT2_5 の処理量と CPU で実行される MT2_6～MT2_25 の処理量を 4:1 に調整）を行った．その後，本並列化コンパイラで並列コードを生成し，24 コア・4 GPU により実行すると実行時間は 2.130 [s] まで短縮され，逐次比で 52.59 倍の速度向上が得られた．

一方，本手法を用いない従来手法との性能比較は**表 6** に示すとおりである．OpenMP（parallel for 構文によるループ並列処理）による CPU のみの 24 コア実行では 4.778 [s] である．手動生成による OpenMP・CUDA コードでは，図 7 (a) の MT2_2，MT2_3，MT2_4，MT2_5 に相当する処理を GPU 実行対象としており，OpenMP の sections 構文の各 section 指示句において，ホストから GPU への cudaMemcpy()，GPU のカーネル関数呼び出し，GPU からホストへの cudaMemcpy() を行う．図 7 (a) の MT2_6，MT2_7，MT2_8，MT2_9 に相当する処理については CPU（24 コア）実行対象とし，OpenMP の parallel for 構文と reduction 指示句により並列コードを作成した．この手動生成の並列コードによる 24 コア・4 GPU 実行時間は 2.991 [s] であった．

これらの結果から，提案するハイブリッド粗粒度タスク並列処理は，マルチコア/マルチ GPU を最大限に利用して，従来手法より 29%の実行時間短縮を実現しており，その有効性が確認された．

6.3　ヤコビ法プログラムを用いたスケジューリングオーバヘッド性能評価

本節では，提案手法の並列コードのダイナミックスケジューリングのオーバヘッドを調べるため，**表 7** のように，ヤコビ法プログラムの行列サイズ N × N における N を 40,960（マルチ GPU 環境の性能評価のデータサイズ），20,480，10,240 と変化させ，逐次実行時間（ソースプログラムの 1 コア実行時間）と 1 コア実行時間（並列コードのダイナミックスケジューリング環境下での 1 コア実行時

表 7　1 コア上でのヤコビ法プログラムの逐次/並列コード実行時間
Table 7　Execution time of sequential/parallel codes for Jacobi method program on one core.

行列サイズ N	10,240	20,480	40,960
逐次実行時間 [ms]	6,566.308	27,998.050	112,012.200
1 コア実行時間 [ms]	6,570.875	28,005.120	112,094.700
収束回数 [回]	15	16	16
実行 MT 数 [個]	153	163	163

間）を測定した．いずれの行列サイズにおいても並列コードによる 1 コア実行時間は，逐次実行時間との差はきわめて小さく，実行 MT 数個のダイナミックスケジューリングに要するオーバヘッドは小さいことが分かる．

6.4　粒子法プログラムを用いたマルチ GPU 環境における性能評価

本節では，粒子法プログラム [20] を用いて性能評価を行う．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 1 つである．粒子法の代表的なものとして SPH 法と MPS 法があり，本性能評価では MPS 法を対象としている．シミュレーションには水柱崩壊を利用した．

表 3 に示すように粒子法プログラムの逐次実行のソースコード長は，並列化指示文なしで 1,379 行である．このソースコードに並列化指示文 128 行を追加したプログラムを入力対象としてコンパイルを行うと，37 個の MT を有する並列コード（2,174 行）のプログラムを生成することができる．粒子法プログラムの階層型マクロタスクグラフは**図 9** のとおりであり，37 個のマクロタスクから構成される．また，処理時間の大きいマクロタスク，すなわち，MT2_2 から MT2_5，MT2_10 から MT2_13，MT2_18 から MT2_21，MT2_22 から MT2_25，MT2_30 から MT2_33 を GPU 実行対象マクロタスクとした．

NVIDIA Tesla K80 搭載マルチコアサーバで実行した結果を**図 10** に示す．本性能評価では粒子数を 68,416 個に設定しており，CPU の逐次実行時間は 2127.202 [s]，4 コアによる実行時間は 718.300 [s] であり，逐次比 2.96 倍の速度向上にとどまっている．

ここで，1 コア・1 GPU におけるブロック数とスレッド数を変更した場合の実行時間の比較を**表 8** に示す．ブロック数×スレッド数 ≥68,416/4 に設定されており，表 8 から実行時間が短くなるブロック数・スレッド数の組合せは 134 ブロック・128 スレッドであるため，kernel() 関数の実行には 134 ブロック・128 スレッドを指定する．

提案するハイブリッド粗粒度タスク並列処理の並列コードによる実行において，1 コア・1 GPU の実行時間は 108.175 [s] となり，逐次比 19.66 倍の速度向上が得られている．さらに，マルチコア/マルチ GPU を活用すると 4 コア・4 GPU の実行時間は 46.577 [s] となり，逐次比で 45.67 倍の速度向上が得られた．

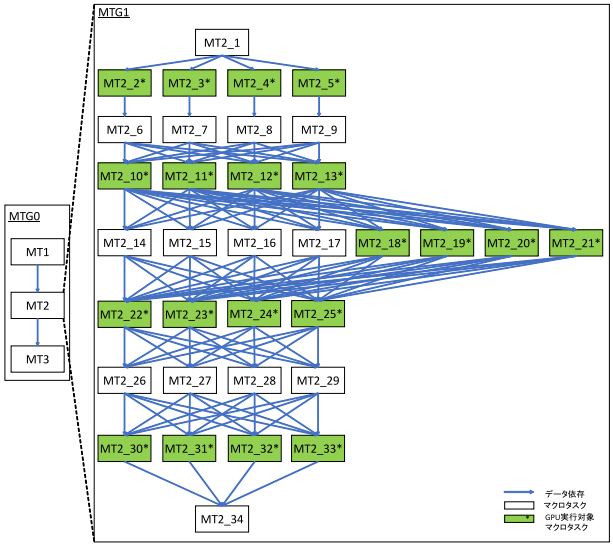

図 9　粒子法プログラムの階層型マクロタスクグラフ
Fig. 9　Hierarchical macro-task-graph particle method program.

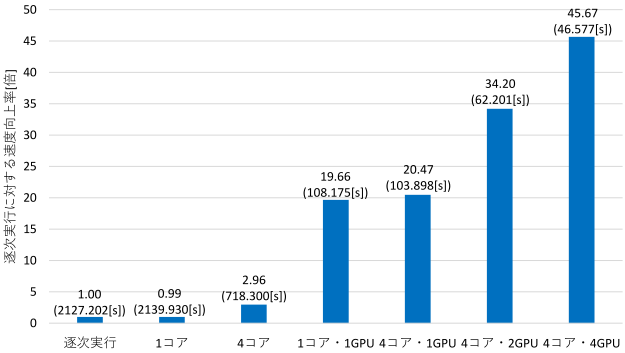

図 10　マルチ GPU 上での粒子法プログラムによる性能評価
Fig. 10　Performance evaluation using particle method program on multi-GPU.

表 8　1 コア・1 GPU 上での粒子法プログラム実行時間
Table 8　Execution time of particle method program on one core and one GPU.

ブロック数	スレッド数	実行時間 [s]
268	64	119.821
134	128	108.175
67	256	118.589
34	512	123.744
17	1,024	147.720

一方，従来手法との性能比較を**表 9** に示す．OpenMP コード（parallel for 構文によるループ並列処理）と単一 GPU 用の CUDA コードは，文献 [20] において提供されており，それらを利用した．OpenMP による 24 コア実行では 101.369 [s]，CUDA による 1 GPU 実行では 82.483 [s] であった．次に，提案手法を用いない複数 GPU のための OpenMP・CUDA コードでは，ホストのメモリにデータを

表 9　粒子法における従来手法との性能比較
Table 9　Performance comparison with conventional schemes for particle method.

実行方法	コア数	GPU数	実行時間 [s]	速度向上率 [倍]（逐次比）
OpenMP	24	0	101.369	20.98
CUDA	1	1	82.483	25.79
OpenMP・CUDA	4	4	47.001	45.26
提案手法	4	4	46.577	45.67

配置し，図 9 の GPU 実行対象マクロタスクに相当する処理に対して OpenMP の sections 構文を使用し，各 section 指示句においてはホストから GPU への cudaMemcpy()，GPU のカーネル関数呼び出し，GPU からホストへの cudaMemcpy() を行う．図 9 の CPU 実行対象マクロタスクに相当する処理については，CPU（4 コア）実行対象として OpenMP の sections 構文により並列コードを作成した．この手動生成の並列コードによる 4 コア・4 GPU 実行時間は 47.001 [s] であった．

これらの結果から，提案するハイブリッド粗粒度タスク並列処理は，並列化指示文の追加のみで並列化コンパイラが並列コードを生成することが可能であり，また，手動生成の並列コードと比べても実行時間を短縮しており，提案手法の有効性が確かめられた．

7. おわりに

本論文では，マルチ GPU システムにおけるハイブリッド粗粒度タスク並列処理手法を提案した．本手法では複数階層にわたる粗粒度タスク並列性を抽出し，処理時間の大きい粗粒度タスクの実行に GPU を利用して，並列実行時間の短縮を図っている．また，本論文では，提案手法による並列コードを自動生成する並列化コンパイラを開発しており，並列化指示文付き C プログラムから，OpenMP/CUDA をともなうハイブリッド粗粒度タスク並列処理の並列コードを容易に生成することができる．

マルチ GPU 搭載サーバによる性能評価では，並列化指示文を加えた C プログラムを並列化コンパイラの入力とし，生成された OpenMP/CUDA をともなうハイブリッド粗粒度タスク並列処理の並列コードを使用した．ヤコビ法プログラムの場合，Xeon サーバの 24 コア・4 GPU 実行において逐次実行比で 52.59 倍の速度向上が得られた．また，粒子法プログラムにおいては 4 コア・4 GPU 実行において逐次実行比で 45.67 倍の速度向上が得られた．

これらの結果から，マルチ GPU システムにおけるハイブリッド粗粒度タスク並列処理の有効性，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．

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] 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] Yoshida, A., Ochi, Y. and Yamanouchi, N.: Parallel Java Code Generation for Layer-unified Coarse Grain Task Parallel Processing, *IPSJ Trans. Advanced Computing Systems*, Vol.7, No.4, pp.56–66 (2014).
[4] Yoshida, A., Kamiyama, A. and Oka, H.: A Task-Driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform, *IPSJ Trans. Advanced Computing Systems*, Vol.10, No.1, pp.1–12 (2017).
[5] 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).
[6] 中尾昌広，村井　均，佐藤三久：PGAS モデルによるマルチ GPU 対応 OpenMP コンパイラ，情報処理学会研究報告，Vol.2018-HPC-165, No.40, pp.1–7 (2018).
[7] 山本　怜，大野和彦：GPGPU フレームワーク MESI-CUDA のマルチ GPU 環境への対応，情報処理学会論文誌，Vol.9, No.1, p.12 (2016).
[8] Matsumura, K., Sato, M., Boku, T., Podobas, A. and Matsuoka, S.: MACC: An OpenACC Transpiler for Automatic Multi-GPU Use, *Asian Conference on Supercomputing Frontiers*, No.40, pp.109–127 (2018).
[9] Verma, K., Szewc, K. and Wille, R.: Advanced Load Balancing for SPH Simulations on Multi-GPU Architectures, *IEEE High Performance Extreme Computing Conference*, pp.1–7 (2017).
[10] LLVM: The LLVM Compiler Infrastructure Project (2019), available from ⟨https://llvm.org/⟩.
[11] LLVM: Clang C Language Family Frontend for LLVM (2019), available from ⟨https://clang.llvm.org⟩.
[12] StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures (2020), available from ⟨http://starpu.gforge.inria.fr⟩.
[13] Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P.A.: StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures, *CPE Special Issue: Euro-Par 2009*, Vol.23, pp.187–198 (2011).
[14] Center, B.S.: Ompss programming model (2020), available from ⟨https://pm.bsc.es/ompss⟩.
[15] Fernandez, A., Beltran, V., Martorell, X., Badia, R.M., Ayguade, E. and Labarta, J.: Task-Based Programming with OmpSs and Its Application, *Springer International Publishing Switzerland* (2014).
[16] Edwards, H.C., Trott, C.R. and Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, *Journal of Parallel and Distributed Computing* (2014).
[17] OpenMP: The OpenMP API specification for parallel programming (2020), available from ⟨https://www.openmp.org/⟩.
[18] OpenACC: OpenACC More Science, Less Programming (2020), available from ⟨https://www.openacc.org/⟩.
[19] NVIDIA: NVIDIA TESLA GPU ACCELERATORS (2014), available from ⟨https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf⟩.
[20] 越塚誠一，柴田和也，室谷浩平：粒子法入門，丸善出版 (2014).

渡辺 智之

2018 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．2020 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 GPU 環境における粗粒度タスク並列処理と並列化コンパイラの研究に従事．2020 年より電源開発株式会社に勤務．

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．