Gears OS における並列処理

東恩納 琢偉[†1]　伊 波　立 樹[†2]　河 野 真 治[†2]

Gears OS は継続を中心とした言語で記述されており、メタ計算をノーマルレベルと分けて記述することができる。並列処理はメタ計算によって記述されており、CbC 自体には並列処理の機能はない。Gears OS のプログラムは Code Gear と Data Gear の集まりである interface によって行われる。Gears OS でのスレッドは interface の集合で出来ており、code gear data gear を接続する context という meta data gear を持つ。 並行実行する場合は新しく context を生成し、それを時分割または、物理的な CPU に割り当てることによって実現される。 つまり、context そのものがスレッドとなる。

Gears OS での同期機構は data gear を待ち合わせることによって行われる。例えば、GPU 上で実行する場合は必要な data gear を GPU 内部に転送し、それらが揃った時点で並列実行される。data gear の待ち合わせはメモリ上の data gear の meta data gear に待ち合わせ用のキューを作ることによって行われる。キューには Gears OS のスレッドつまり context meta data gear が入る。

本論文では Gears OS での並列処理の構成方法について述べる。並列処理をメタレベルで行うことにより、並列処理で重要なチューニングや性能測定あるいはデバッグをメタ計算を切り替えることにより、ノーマルレベルの計算を変更することなく行うことができることを示す。

Takui HIGASHIONNA ,[†1] Tatsuki IHA [†2] and Shinji KONO [†2]

1. Gears OS

CPU の処理速度の向上のためクロック周波数の増加は発熱や消費電力の増大により難しくなっている。そのため、クロック周波数を上げる代わりに CPU のコア数を増やす傾向にある。マルチコア CPU の性能を発揮するには、処理をできるだけ並列化しなければならない。また、PC の処理性能を上げるためにマルチコア CPU 以外にも GPU や CPU と GPU を複合したヘテロジニアスなプロセッサが登場している。並列処理をする上でこれらのリソースを無視することができない。しかし、これらのプロセッサで性能を出すためにはこれらのアーキテクチャに合わせた並列プログラミングが必要になる。並列プログラミングフレームワークではこれらのプロセッサを抽象化し、CPU と同等に扱えるようにすることも求められる。本研究では Cerium を開発して得られた知見を元にこれらの性質を持つ並列プログラミングフレームワークとして Gears OS の設計・実装を行う。

†1 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

Cerium[?)] は本研究室で開発していた並列プログラミングフレームワークである。Cerium では Task と呼ばれる分割されたプログラムを依存関係に沿って実行することで並列実行を可能にする。Cerium では依存関係を Task 間で設定するが、本来 Task はデータに依存するもので Task 間の依存関係ではデータの依存関係を保証することができない。また、Task には汎用ポインタとしてデータの受け渡しを行うため、型情報がない。そのため、汎用ポインタをキャストして利用するしか無く、型の検査を行う事ができない。

Gears OS[?)] は Code Gear と Data Gear によって構成される。Code Gear は処理の単位、Data Gear はデータの単位となる。Gears OS では Code/Data Gear を用いて記述することでプログラム全体の並列度を高めて、効率的に並列処理することが可能になることを目的とする。また、Gears OS の実装自体が Code/Data Gear を用いたプログラミングの指針となるように実装する。Gears OS における Task は実行する Code Gear と実行に必要な Input Data Gear, 出力される Output Data Gear の組で表現される。Input/Output Data Gear によって依存関係が決定し、それに沿って並列実行する。依存関係の解決などの Meta Computation の実行は Meta Code Gear で行われる。Meta Code Gear は Code Gear に対応しており、 Code Gear が実行した後にそれに対応した

Meta Code Gear が実行される。本論文では Gears OS のプロトタイプとして Data Gear を管理する Persistent Data Tree, Task を管理する TaskQueue, 並列処理を行う Worker を実装し、簡単な例題を用いて Gears OS の評価を行う。

2. Code Gear と Data Gear

Gears OS はプログラムの単位として Gear を用いる。Gear は並列実行の単位、データの分割、Gear 間の接続等になる。

Code Gear はプログラムの処理そのものである。Code Gear は任意の数の Input Data Gear を参照し、処理が完了すると任意の数の Output Data Gear に書き込む。Code Gear は接続された Data Gear 以外には参照を行わない。

Data Gear は Data そのものを表しており、int や文字列などの Primitive Data Type が入っている。

Code Gear、 Data Gear は 本研究室で開発されている Alice[?)] で使われている単位である Code Segment、 Data Segment[?)] にそれぞれ対応する。

Gears OS では Code Gear と Input / Output Data Gear の対応から依存関係を解決し、Code Gear の並列実行を可能とする。

Gear の特徴として処理やデータの構造が Code Gear、 Data Gear に閉じていることにある。これにより、実行時間、メモリ使用量などを予想可能なものにする事が可能になる。

3. Meta Computation

Gears OS では通常の処理を Computation、Computation のための Computation を Meta Computation として扱う。Meta Computation の例として並列処理の依存関係の解決や、OS が行うネットワーク管理、メモリ管理等の資源制御などが挙げられる。

Gears OS では Meta Computation を Meta Code Gear、 Meta Data Gear で表現する。Meta Code Gear は通常の Code Gear 直後に遷移され、 Meta Computation を実行する。Meta Computation の実行後は通常の Code Gear で指定した Code Gear を実行する。つまり Code Gear の実行後は何かしらの Meta Code Gear を実行する。

4. Continuation based C

Gears OS の実装は本研究室で開発している CbC(Continuation based C)[?)] を用いて行う。CbC は処理を Code Segment を用いて記述することを基本としているため、 Gears OS の Code Gear を記述するのに適している。

CbC のプログラムでは C の関数の代わりに Code Segment を用いてい処理を記述している。 Code Segment は C の関数と異なり戻り値を持たない。Code Segment の宣言は C の関数の構文と同様に行い、 型に __code を使うことで宣言できる。

Code Segment から Code Segment への移動は goto の後に Code Segment 名と引数を並べた記述するという構文を用いて行う。この goto による処理の遷移を継続と呼ぶ。 図?? は Code Segment 間の継続関係を表している。

C では関数呼び出しを行うたび、関数の引数の値がスタックに積まれていくが、Code Segment では戻り値を持たないため、スタックに値を積んでいく必要がなくスタックを変更する必要が無い。このようなスタックに値を積まない継続、つまり呼び出し元の環境を持たない継続を軽量継続と呼ぶ。軽量継続により、並列化、ループ制御、関数コールとスタックの操作を意識した最適化がソースコードレベルで行えるようにする。

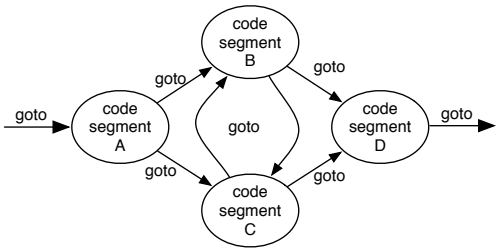

図 1 goto による Code Segment 間の接続

5. CbC での Gears OS の構文サポート

CbC は Gears OS の構文のサポートを行う。

Gesrs OS では Context という接続可能な Data Gear のリストからデータを取り出して処理行う。しかし、 Context を直接扱うのはセキュリティ上好ましくない。そこで Gears OS では Context から必要なデータを取り出して Code Gear に接続する stub を定義する。stub は Code Gear から推論することが可能のため、 CbC は自動的に stub の生成を行う。

また、Code Gear の遷移には Meta computation を行うために Meta Code Gear を挟む。CbC では Meta Code Gear への接続も自動的に行うようにする。

6. Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
- TaskQueue
- TaskManager
- Persistent Data Tree
- Worker

図?? に Gears OS の構成図を示す。

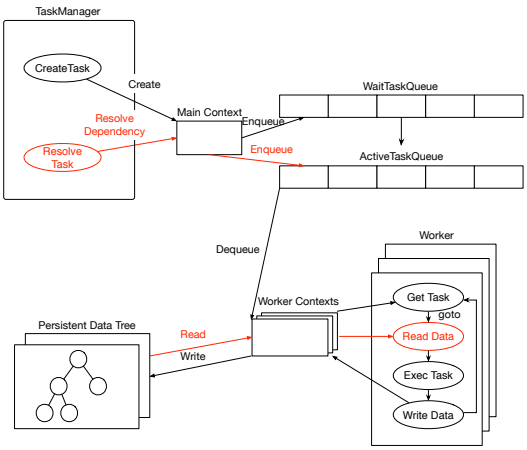

図 2 Gears OS の構成図

7. Context

Context は接続可能な Code/Data Gear のリスト、TaskQueue へのポインタ、 Persistent Data Tree へのポインタ、 Temporal Data Gear のためのメモリ空間等を持っている Meta Data Gear である。Gears OS では必要な Code/Data Gear に参照したい場合、この Context を通す必要がある。

メインとなる Context と Worker 用 Context があり、 TaskQueue と Persistent Data Tree は共有される。Temporal Data Gear のためのメモリ空間は Context 毎に異なり、互いに干渉することはない。Worker 間の相互作用は Persistent Data Tree への読み書きのみで行う。

Code ??, Code ?? に実際の Context の定義と生成を示す。

```
/* define context */

#define ALLOCATE_SIZE 20000000
#define NEWN(n, type) (type*)(calloc(n, sizeof(type
    )))
#define ALLOC_DATA(context, dseg) ({ context->data[
    dseg] = context->heap; context->heap += sizeof(
    struct dseg); (struct dseg *)context->data[dseg
    ]; })

enum Code {
    Code1,
    Code2,
    Code3,
};

enum UniqueData {
    Allocate,
    Tree,
    Queue,
    Worker,
};
```

```
struct Context {
    enum Code next;
    int codeNum;
    __code (**code) (struct Context*);
    void* heapStart;
    void* heap;
    long heapLimit;
    pthread_t thread;
    int thread_num;
    int dataNum;
    union Data **data;
};

union Data {
    struct Worker {
        int num;
        struct Context* contexts;
    } worker;
    struct Tree {
        struct Node* root;
    } tree;
    struct Node {
        // need to tree
        enum Code next;
        int key; // comparable data segment
        union Data* value;
        struct Node* left;
        struct Node* right;
        // need to balancing
        enum Color {
            Red,
            Black,
        } color;
    } node;
    struct Allocate {
        enum Code next;
        long size;
    } allocate;
};
```

Code 1 Context

```
#include <stdlib.h>

#include "context.h"

extern __code code1_stub(struct Context*);
extern __code code2_stub(struct Context*);
extern __code code3_stub(struct Context*);

__code initContext(struct Context* context) {
    context->heapLimit = sizeof(union Data)*
        ALLOCATE_SIZE;
    context->code = (__code(**) (struct Context*))
        NEWN(ALLOCATE_SIZE, void*);
    context->data = NEWN(ALLOCATE_SIZE, union Data*);
    context->heapStart = NEWN(context->heapLimit,
        char);
    context->heap = context->heapStart;

    context->codeNum = Code3;

    context->code[Code1] = code1_stub;
    context->code[Code2] = code2_stub;
    context->code[Code3] = code3_stub;

    struct Worker* worker = ALLOC_DATA(context,
        Worker);
    worker->num = 0;
    worker->contexts = 0;

    struct Allocate* allocate = ALLOC_DATA(context,
```

```
        Allocate);
    allocate->size = 0;

    struct Tree* tree = ALLOC_DATA(context, Tree);
    tree->root = 0;

    struct Node* node = ALLOC_DATA(context, Node);
    node->key = 0;
    node->value = 0;
    node->left = 0;
    node->right = 0;
}
```

Code 2　initContext

Code **??**, Code **??** は以下の事を定義している。

Code Gear の名前とポインタのリスト

Code Gear の名前とポインタの対応は Code **??** の enum Code と 関数ポインタによって表現される。実際に Code Gear に接続する際は enum Code を指定することで接続を行う。これにより、実行時のルーチンなどを動的に変更することが可能となる。

Data Gear の Allocation 用の情報

Context の生成時 (Code **??**)、 Allocation 用に Code **??** の ALLOCATE_SIZE 分の領域を確保する。Context にはその領域へのポインタとサイズが格納されている (Code **??**の struct Context 内の heap, heapLimit)。実際に Allocation する際は heap を 必要な Data Gear のサイズに応じてインクリメントすることで Data Gear の Allocation を実現する。

Data Gear へのポインタ

Context には Allocation 等で生成した Data Gear へのポインタが格納されている。Code Gear は Context を通して Data Gear へアクセスする。

Data Gear に格納される Data Type の情報

Data Gear は Code **??** の union Data と その中の struct によって表現される。Context には Data Gear の Data Type の情報が格納されている。この情報から確保される Data Gear のサイズなどを決定する。

8. TaskQueue

Gears OS における Task Queue は Synchronized Queue で実現される。メインとなる Context と Worker 用 の Context で共有され、 Worker が TaskQueue から Task を取得し、実行することで並列処理を行う。

Gears OS の Queue は Queue を表す Data Gear と List を表現する Element という名前の Data Gear を組み合わせて表現する。Queue を表す Data Gear には List 構造の先頭の Element を指す first, 末尾の Element を指す last, Element の個数を示す count が格納される。Element を表す Data Gear は、Task を示す task、 次の Element を示す next が格納される。

Queue に対して操作を行う場合、 Queue 自体の Data Gear を書き換える。Task を 挿入する場合、新しく Element を生成し、 Queue の last から List 構造の末尾に新しい Element を追加し、 Queue の last を書き換える。Task を 取得する場合、 Queue の first から List 構造を最初の要素を取り出し、 取り出した要素の次の要素の参照を Queue の first に書き込む。

Gears OS の TaskQueue はマルチスレッドでの操作を想定しているため、データの一貫性を保証する必要がある。そのため、データの一貫性を並列実行時でも保証するために Compare and Swap(CAS) を利用して Queue の操作を行っている。CAS はデータの比較・置換をアトミックに行う命令である。メモリからデータの読みだし、変更、メモリへのデータの書き出しという一連の処理を CAS を利用することで処理の間に他のスレッドがメモリに変更を加えないことを保証することができる。CAS に失敗した場合は置換を行わず、再びデータの呼び出しから始める。

Code **??** に CAS を使用した Task 挿入を示している。Code **??** は 2 つの Code Gear を定義しており、putQueue3 は Queue に要素がある場合、putQueue4 は Queue に要素がない場合の Task 挿入を示している。

```
// Enqueue(normal)
__code putQueue3(struct Context* context, struct
      Queue* queue, struct Element* new_element) {
    struct Element* last = queue->last;

    if (__sync_bool_compare_and_swap(&queue->last,
         last, new_element)) {
        last->next = new_element;
        queue->count++;

        goto meta(context, context->next);
    } else {
        goto meta(context, PutQueue3);
    }
}

// Enqueue(nothing element)
__code putQueue4(struct Context* context, struct
     Queue* queue, struct Element* new_element) {
    if (__sync_bool_compare_and_swap(&queue->first,
          0, new_element)) {
        queue->last = new_element;
        queue->count++;

        goto meta(context, context->next);
    } else {
        goto meta(context, PutQueue3);
    }
}
```

Code 3　Enqueue

9. Persistent Data Tree

Gears OS は Persistent Data Gear の管理に木構造を用いる。この木構造は非破壊的で構成される。非

破壊木構造とは図??のように一度構築した木構造を破壊すること無く新しい木構造を構築することで、木構造を編集する方法である。非破壊木構造は木構造を書き換えることなく編集を行うため、読み書きを平行して行うことが可能である。

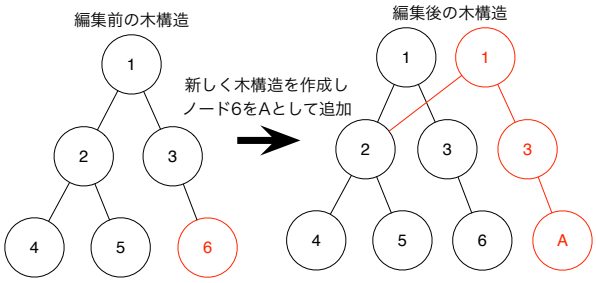

図 3 木構造の非破壊的編集

Gears OS では Data Tree として木構造を利用する。その場合、普通に木構造を構築するだけでは偏った木構造が構築される可能性がある。最悪なケースでは線形リストになり、計算量が O(n) となる。

そのため、挿入・削除・検索における処理時間を保証するため Red-Black Tree を用いて木構造の平衡性を保証する。Red-Black Tree は通常の二分探索木としての条件の他に以下の条件を持つ。

- 各ノードは赤または黒の色を持つ。
- ルートの色は黒である。
- 赤ノードは 2 つの黒ノードを子として持つ (赤ノードが続くことはない)。
- ルートから最下位ノードへのパスに含まれる黒ノードの数はどの最下位ノードでも一定である。

これらの条件によってルートから最も遠い最下位ノードへのパスの長さはルートから最も近い最下位ノードへのパスの長さの 2 倍に収まることが保証される。

10. Worker

Worker は TaskQueue から Task を取得し、実行する。Task には実行する Code Gear と実行に必要な Code Gear の key が格納されている。実行に必要な Code Gear は Persistent Data Tree から key を使って取得する。

各 Worker は個別の Context を参照しており、メモリ空間も独立しているのでメモリを確保する処理で他の Thread を止めることはない。ただし、Persistent Data Tree への書き出しは競合する可能性があるので CAS を利用してデータの一貫性を保証する必要がある。

Worker が TaskQueue から Task の取得を行う Code Gear を Code ?? に示す。Task Queue から取得した Task から実行する Code Gear と必要な Data Gear の key を Worker Context に書き込むことで実行される。

```
// Dequeue
__code getQueue(struct Context* context, struct Queue
      * queue, struct Node* node) {
    if (queue->first == 0)
        return;

    struct Element* first = queue->first;
    if (__sync_bool_compare_and_swap(&queue->first,
          first, first->next)) {
        queue->count--;

        context->next = GetQueue;
        stack_push(context->code_stack, &context->next
              );

        context->next = first->task->code;
        node->key = first->task->key;

        goto meta(context, Get);
    } else {
        goto meta(context, GetQueue);
    }
}
```

Code 4 GetTask

Worker から取得された Task の Code Gear は並列実行される。並列実行される Code Gear と言っても他の Code Gear と同じである。これは Gears OS 自体が Code Gear によって構成されていることに起因する。つまり、Gears OS を利用して書かれたプログラムで定義されている Code Gear に依存関係がないとき、全て並列に実行することができる。

11. TaskManager

Gears OS の TaskManager は WaitTaskQueue に入っている Task の依存関係を解決する。Task には Input/Output Data Gear の情報が格納されている。Input Data Gear は Task に必要な Data Gear で揃ったら Task は実行可能な状態になる。Output Data Gear は Task が Persistent Data Tree に書き出す Data Gear である。この Input と Output の関係が依存関係となる。TaskManager は Persistent Data Tree を監視しており、WaitTaskQueue に入っている Task の Input Data Gear が揃っているのを確認したら実行可能な Task として AcitiveTaskQueue へ移動させる。

12. プロトタイプの動作

Gears OS の評価として依存関係のない例題の並列実行を行った。

今回使用した例題は Twice という整数配列を 2 倍にする例題である。Code ?? に Twice の処理を行う Code Gear を示す。

```
// Twice
```

```
__code twice(struct Context* context, struct
     LoopCounter* loopCounter, int index, int
     alignment, int* array) {
    int i = loopCounter->i;

    if (i < alignment) {
        array[i+index*alignment] = array[i+index*
              alignment]*2;
        loopCounter->i++;

        goto meta(context, Twice);
    }

    loopCounter->i = 0;

    stack_pop(context->code_stack, &context->next);
    goto meta(context, context->next);
}
```

Code 5　Twice

以下に今回の処理の流れを示す。

- 配列サイズを元に index, alignment, 配列へのポインタを持つ Data Gear に分割。
- Data Gear を Persistent Data Tree に挿入。
- 実行する Code Gear(Twice) と実行に必要な Data Gear への key を持つ Task を生成。
- 生成した Task を TaskQueue に挿入。
- Worker の起動。
- Worker が TaskQueue から Task を取得。
- 取得した Task を元に必要な Data Gear を Persistent Data Tree から取得。
- 並列の処理される Code Gear(Twice) を実行。

要素数 2^{17}*1000 のデータを 640 個の Task に分割し、コア数を変更して測定を行った結果を表??、図??に示す。

Processor	Time(ms)
1 CPU	1315
2 CPUs	689
4 CPUs	366
8 CPUs	189
12 CPUs	111

表 1　要素数 2^{17}*1000 のデータに対する Twice

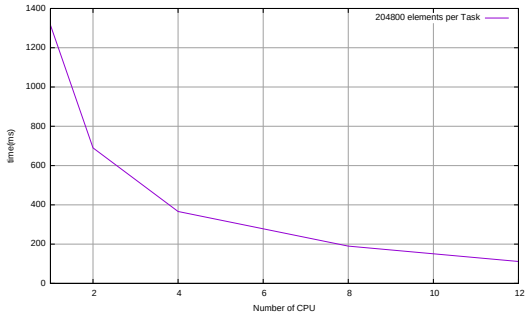

図 4　要素数 2^{17}*1000 のデータに対する Twice

結果から、 1 CPU と 12 CPU で約 11.8 倍の速度向上が見られた。しかし、 タスクの粒度が小さすぎると CAS の失敗が多くなり、性能がでないことがある。Code Gear には実行時間を予想可能なものにするという特徴があるため、タスクが最適な粒度なのかを検査する機能が必要になると考えられる。

13. 比　　較

本章では今回設計・実装した Gears OS と既存の並列フレームワークとの比較を行う。また、Gears OS は以下のような性質を有している。

- リソース管理
  Context 毎に異なるメモリ空間を持ち、それを管理する。Meta Code Gear, Meta Data Gear を用いてネットワーク管理、並行制御等を行う。
- 処理の効率化
  依存関係のない Code Gear は並列実行することが可能である。また、Code Gear 自体が処理の最小単位となっており Code Gear を利用してプログラムを記述するとプログラム全体の並列度を高めることに繋がる。
- プロセッサ利用の抽象化
  Multi Core CPU, GPU を同等の実行機構で実行可能である。

これらの性質を有する Gears OS はオペレーティングシステムであると言えるので既存の OS との比較も行う。

Cerium

- 依存関係
  Cerium では Task 間で依存関係を設定する。Task の途中でデータが破損しても完了を TaskManager に通知し、依存関係を解決して次の Task が実行される。これではデータの正しさを保証することができない。
  Gears OS では Task に Input/Output Data Gear を設定することで Input と Output の関係から依存関係を決定する。TaskManager は Persistent Data Tree を監視し、必要な Data Gear が揃っていることを確認すると依存関係を解決する。
- データの型情報
  Cerium では Task にデータを引き渡すとき汎用ポインタを用いる。このときデータの型情報が落ちるので Task の組み合わせが型的に安全なのか保証することができない。
  Gears OS では型情報を持つ分割されたデータとして Data Gear を定義し、Data Gear 単位でデータを Task に引き渡す。Data Gear を型シグネチャとして Task の組み合わせが正しいことを保証する。

- Allocator
  Cerium では Thread 間で Allocator を共有している。ある Thread がメモリ確保を行うとその間、他の Thread はメモリを確保することができず並列度が低下する。
  Gears OS では Thraed ごとに Context を割り当てる。Context は独立したメモリ空間を持つので他の Thread と干渉することないメモリの確保を行うことができる。
- 並列処理との相性
  Cerium はオブジェクト指向言語である C++ で実装されている。オブジェクト指向は保守性と再利用性を高めるためにカプセル化とポリモフィズムを重視する。オブジェクトの状態によって振る舞いが変わるため参照透過な処理でなくなり並列処理との相性が悪い。
  Gears OS は本研究で開発している CbC を用いて実装する。CbC は Code Segment という単位でプログラムを記述する。Code Segment はスタックに値を積まない軽量継続を用いて他の Code Segment に遷移する。この軽量継続により並列化、ループ制御などを意識した最適化がソースコードレベルで行うことができる。

OpenCL/CUDA

OpenCL[?)]/CUDA[?)] では並列処理に用いる関数を kernel として定義する。OpenCL では CommandQueue, CUDA では Stream という命令キューに命令を発行することで GPU を利用することができる。命令キューは発行された順番通りに命令が実行されることが保証されている。複数の命令キューを準備して、各命令キューに命令を発行することで命令を並列に実行することができる。命令キュー単位で依存関係を設定することができる。つまり、命令キューに入っている最後の命令次第でデータを待っているのか kernel の実行を待っているのか変わるので依存関係の記述が複雑になる。データは kernel の引数の定義に型変換され渡される。データ転送の際には型情報が落として渡す必要があり、型を意識したプログラミングが必要になる。

一方、Gears OS ではデータによって依存関係が決定する。また、データを Data Segment という単位で分割して管理しており型情報を保ったままデータの受け渡しを行うことができる。

OpenMP

OpenMP ではループ制御構文の前にアノテーションを付ける (Code **??**) ことでコンパイラが解釈し、スレッド処理を行うように変換して並列処理を行う。

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    // Processing
}
```

Code 6　OpenMP

他の並列化手法に比べて既存のコードに対する変更が少なくて済む。しかし、この方法ではプログラム全体の並列度が上がらずアムダールの法則により性能向上が頭打ちになる。

一方、Gears OS では初めから Code Gear, Data Gear という単位でプログラムを分割して記述するのでプログラム全体の並列度を高めることができる。

従来の OS

従来の OS が行ってきたネットワーク管理、メモリ管理、平行制御などのメタな部分を Gears OS では Meta Code/Data Gear として定義する。通常の Code Gear から必要な制御を推論し、Meta Code Gear を接続することで従来の OS が行ってきた制御を提供する。このメタ計算は関数型言語で用いられる Monad に基づいて実現する。

14. ま と め

本論文では Code Gear、 Data Gear によって構成される Gears OS のプロトタイプの設計、実装を行った。

Code Gear は処理、 Data Gear はデータの単位である。Code Gear は戻り値を持たないので、関数呼び出しのようにスタックに値を積む必要がなく、スタックは変更されない。そのため並列化、ループ制御、関数コールとスタックの操作を意識した最適化をソースコードレベルで行える。また、プログラム Code/Data Gear に分割して記述することで並列度を高めることができる。

Gears OS の基本的な機能として、 TaskQueue, Persistent Data Tree, Worker の実装を Code/Data Gear に基づいて行った。Gears OS では Context というデータ構造に Code/Data Gear のリスト、TaskQueue へのポインタ Persistent Data Tree へのポインタ、Temporal Data Gear を確保するためのメモリ空間などがある。Context はスレッド毎に存在し、それぞれが異なる Context を参照している。TaskQueue は並列処理される Task を管理する。TaskQueue はすべての Context で共有され、マルチスレッドで動作する必要がある。そのためデータの一貫性を保つために Compare and Swap(CAS) を用いた実装を行った。Persistent Data Tree は 並列処理 での Data Gear の管理を行う。そのためすべての Context で共有される。Persistent Data Tree は非破壊木構造で構成することで読み書きのを平行して行う事が可能となった。また、Red-Black Tree アルゴリズムを用いて実装することで木の平衡性が保たれる。Worker は TaskQueue から Task を取り出し、 Persistent Data Tree から Data Gear を取得し、 Task 内の Code Gear の並列実行を行う。また、個別の Context を参照している

ので、メモリ空間が独立しており、メモリを確保する処理で他の Worker を止めることはない。

今後の課題として、 依存関係のある並列処理の実現、 GPU などのプロセッサ等での実行、 デバック手法、 型検査が上げられる。

今回の例題では Twice を用いて並列処理の性能を示したが、 Twice は依存関係のない並列処理である。本来、並列処理には依存関係が存在するため、 複雑な並列処理でも安定した実行ができることを依存関係がある並列処理の例題を作成し、評価する必要がある。

Gears OS 上でマルチコア CPU を用いた実行を可能にしたが、 GPU などの他のプロセッサを演算に用いることができない。そのため、 Data Gear 等のデータを GPU などの各プロセッサにマッピングするための機構を用意する必要がある。

Gears OS は 関数呼び出しではなく、スタックを積まない軽量継続を使用して実装されているため、スタックトレースが見えず、従来のデバック手法が使えない。そのため、Gears OS 用の 新しい デバック手法を考案する必要がある。Gears OS は Context から Data Gear の情報を取得できるため、そこから今の状況を把握することができる。その性質から、 Context を見ることができるコードを Meta Code Gear に入れることで、 Code Gear を止めて Data Gear の状況を見ることが可能となる。しかし、この方法では並列で動いている Code Gear に対しては綺麗にデバック出来ない。そのため、並列処理でのデバック手法も考案する必要がある。

また、型情報を残すために Data Gear を定義しているが、 Data Gear の型情報を検査していない。プログラムの正しさを保証するために Data Gear の型情報を検査するシステムを 実装する必要がある。