Code Gear、 Data Gear に基づく OS のプロトタイプ

伊波立樹[†1]　東恩納　琢偉[†2]　河野真治[†2]

当研究室では 処理の単位を Code Gear、 データの単位を Data Gear を用いて並列実行を行う Gears OS を開発している Gears OS では 並列実行をするための Task を Code Gear と Data Gear の組で表現する。Task の依存関係は Code Gear を実行するために必要な Input Data Gear と Code Gear で作られる Output Data Gear によって決定し、それにそって並列実行を行う。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並列処理機構を設計し、 CbC(Continuation based C) で実装する。

Tatsuki IHA ,[†1] Takui HIGASHIONNA [†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。

Cerium[1)] 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、本要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[2)] 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

†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.

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[3] で使われている単位である Code Segment、 Data Segment[4] それぞれに対応する。

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)[5] 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 1 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。

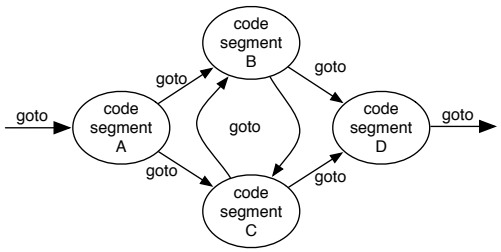

図 1 goto による Code Segment 間の接続

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 2 に Gears OS の構成図を示す。

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。

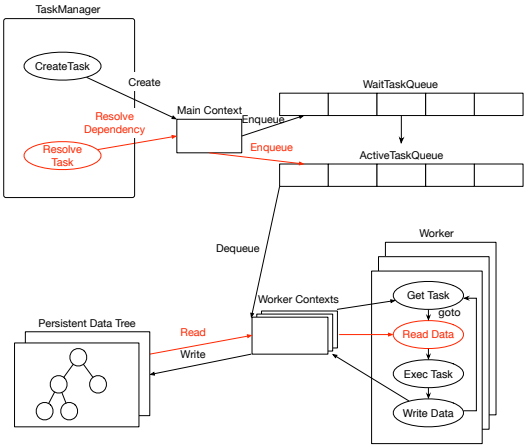

図 2 goto による Code Segment 間の接続

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出来ない。Worker 間の相互作用は Persistent Data Tree への読み書きのみで行う。

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 1 に CAS を使用した Task 挿入を示している。Code 1 は 2 つの Code Gear を定義しており、putQueue3 は要素がある場合、 putQueue4 は要素がない場合の 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 1 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 3 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。

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。

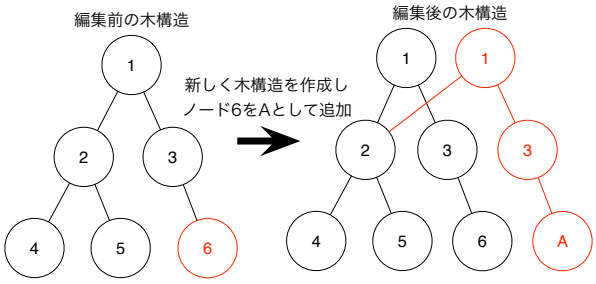

図 3 木構造の非破壊的編集

- 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 Task の取得を行う Code Gear を Code 2 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 2 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. 評　　価

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 3 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 3 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 1、図 4 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

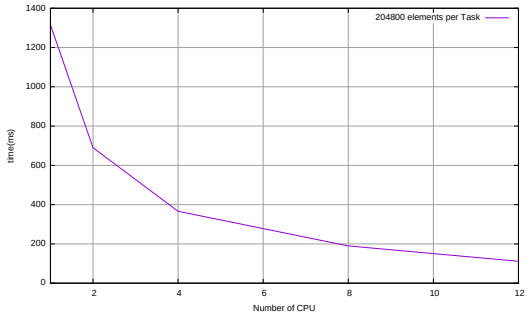

図 4 要素数 2^{17}*1000 のデータに対する Twice

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. 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。

参 考 文 献

1) 宮國 渡, 河野真治, 神里 晃, 杉山千秋：Cell 用の Fine-grain Task Manager の実装, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2008).
2) 小久保翔平, 伊波立樹, 河野真治：Monad に基づくメタ計算を基本とする Gears OS の設計, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2015).
3) 照屋のぞみ, 河野真治：分散フレームワーク Alice の PC 画面配信システムへの応用, 第 57 回プログラミング・シンポジウム (2016).
4) 河野真治, 杉本 優：Code Segment と Data Segment によるプログラミング手法, 第 54 回プログラミング・シンポジウム (2013).
5) TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
6) Sony Corporation: Cell broadband engine architecture (2005).
7) 徳森海斗, 河野真治：Continuation based C の LLVM/clang 3.5 上の実装について, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2014).
8) Moggi, E.: Computational lambda-calculus and monads, *Proceedings of the Fourth Annual Symposium on Logic in computer science* (1989).
9) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
10) : CUDA, https://developer.nvidia.com/category/zone/cuda-zone/.