修士 (工学) 学位論文

Master's Thesis of Engineering

Gears OS の並列処理

Parallel processing of Gears OS

2017 年 3 月

March 2018

伊波 立樹

Tatsuki IHA

琉球大学

大学院理工学研究科

情報工学専攻

Information Engineering Course

Graduate School of Engineering and Science

University of the Ryukyus

指導教員 : 教授　和田 知久

Supervisor: Prof. Tomohisa WADA

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　和田　知久

印

(副　査)　　岡崎　威生

印

(副　査)　　名嘉村　盛和

印

(副　査)　　河野　真治

要 旨

アブストラクト

Abstract

abstract

目 次

図 目 次

表 目 次

ソースコード目次

第1章　メタ計算を使った並列処理

1.1　本論文の構成

第2章　Gears OS

2.1　Code Gear と Data Gear

Gears OS はプログラムとデータの単位として 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、図 2.1 で示しているように任意の数の 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。また、Code Gear は接続された Data Gear 以外には参照を行わない。この 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。

Code Gear 間の移動は継続を用いて行われる。継続は関数呼び出しとは異なり、呼び出し元に戻らず、Code Gear 内で次の Code Gear への継続を行う。そのため Code Gear、Data Gear を使ったプログラミングは末尾再帰を強制したスタイルになる。

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。

2.2　Continuation based C

Gears OS の実装は本研究室で開発されている CbC (Continuation based C) を用いて行う。CbC は Code Gear を基本的な処理単位として記述できるプログラミング言語である。

CbC の記述例をソースコード 2.1 に, 実際にこのソースコードが実行される際の遷移を図 2.2 に示す。CbC の Code Gear は __code という型を持つ関数として記述する。Code Gear は継続で次の Code Gear に遷移する性質上、関数とは違い戻り値は持たない。そのため、__code は Code Gear の戻り値ではなく、Code Gear であることを示すフラグとなっている。Code Gear から次の Code Gear への遷移は goto による継続で処理を行い、次の Code Gear への引数として入出力を与える。ソースコード 2.1 内の goto cg1 (a+b); が継続にあたり、 (a+b) が cg1 への入力になる。

ソースコード 2.1: CodeSegment の軽量継続

```
1 __code cg0(int a, int b) {
2     goto cg1(a+b);
```

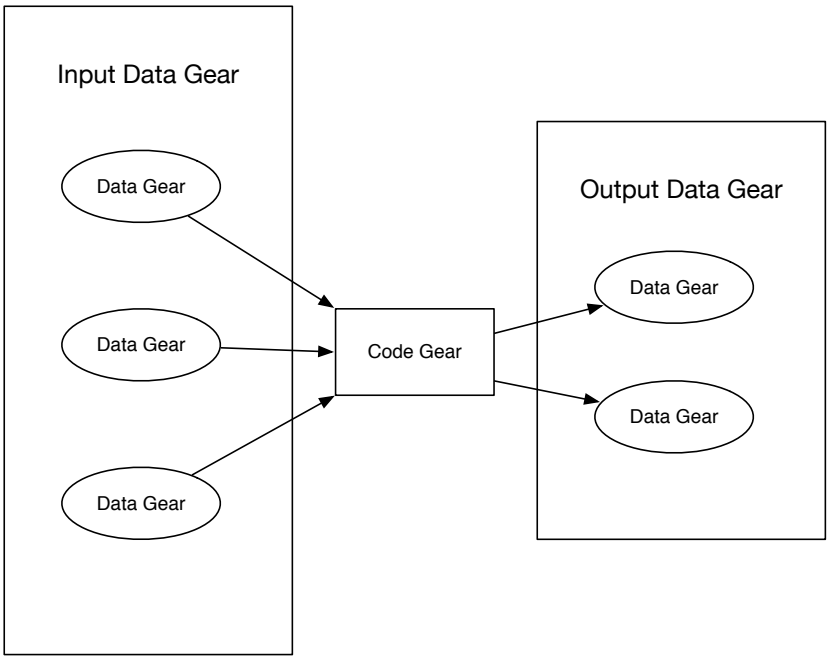

図 2.1: Code Gear と Data Gear の関係

```
3
4 }
5
6 __code cg1(int c) {
7     goto cg2(c);
8 }
```

CbC の goto による継続は Scheme の call/cc といった継続と異なり、呼び出し元の環境を必要とせず、行き先を指定すれば良い。この継続を軽量継続と呼ぶ。ソースコード 2.1 は cs0 から cs1 へ継続したあとには cs0 へ戻らずに処理を続ける。

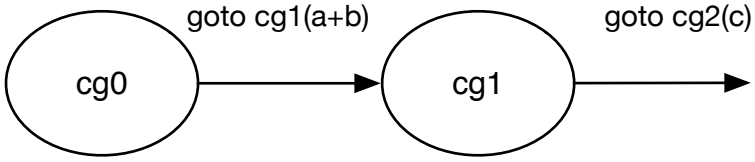

図 2.2: goto による Code Gear の軽量継続

2.3 メタ計算

プログラムの記述する際は、ノーマルレベルの計算の他に、メモリ管理、スレッド管理、CPU が GPU の資源管理等を記述しなければならない処理が存在する。これらの計算はノーマルレベルの計算と区別してメタ計算と呼ぶ。

メタ計算は関数型言語では Monad を用いて表現される。Monad は Haskell では実行時の環境を記述する構文として使われる。

従来の OS では、メタ計算はシステムコールやライブラリーコールの単位で行われる。実行時にメタ計算の変更を行う場合には OS 内部のパラメータの変更を使用し、実行されるユーザープログラム自体への変更は限定的である。しかし、メタ計算は性能測定あるいはプログラム検証、さらに並列分散計算のチューニングなど細かい処理が必要で実際のシステムコール単位では不十分である。例えば、モデル検査ではアセンブラあるいは バイトコード、インタプリタレベルでのメタ計算が必要になる。しかし、バイトコードレベルでは 粒度が細かすぎて扱いが困難になっている。具体的にはメタ計算の実行時間が大きくなってしまう。

2.4 Meta Gear

Gears OS の Code Gear は関数に比べて細かく分割されているため、メタ計算をより柔軟に記述できる。Code Gear と Data Gear にはそれぞれメタ計算の区分として Meta Code Gear、Meta Data Gear が存在し、これらを用いてメタ計算を実装する。Meta Gear は 制限された Monad に相当し、型付きアセンブラよりは大きな表現単位を提供する。Haskell などの関数型プログラミング言語では実行環境が複雑であり、実行時の資源使用を明確にすることができないが、Gears OS を記述している CbC はスタック上に隠された環境を持たないので、メタ計算で使用する資源を明確にできる利点がある。Meta Code Gear は図 2.3 に示すように通常の Code Gear の直後に遷移され、メタ計算を実行する。また、 Meta Code Gear は、その階層からさらにメタ計算を記述することが可能である。

2.5 Context

Context は接続可能な Code/Data Gear のリスト、Data Gear を確保するメモリ空間、実行される Task への Code Gear 等を持っている Meta Data Gear である。Gears OS では Code Gear と Data Gear への接続を Context を通して行う。

また、Context は並列実行の Task でもあり、従来のスレッドやプロセスに対応する。そのため Gears OS で並列実行を行うには Context を生成し、Task の実行を行う。

ソースコード 2.2 に Context の定義を示す。

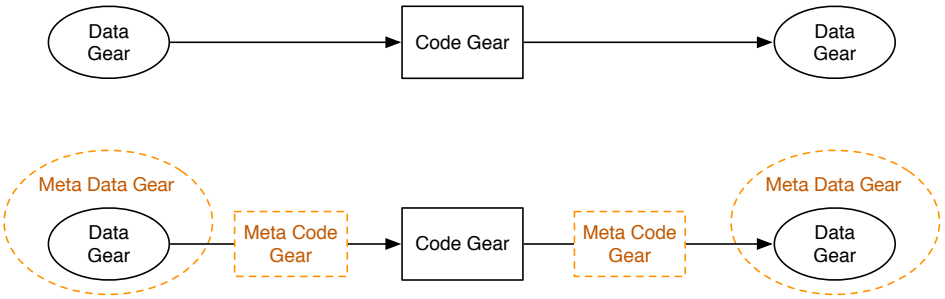

図 2.3: Meta Code Gear の実行

ソースコード 2.2: Context の定義

```
1  /* Context definition */
2  struct Context {
3      enum Code next;
4      int codeNum;
5      __code (**code) (struct Context*);
6      union Data **data;
7      void* heapStart;
8      void* heap;
9      long heapLimit;
10     int dataNum;
11
12     // task parameter
13     int idgCount; //number of waiting dataGear
14     int idg;
15     int maxIdg;
16     int odg;
17     int maxOdg;
18     int gpu; // GPU task
19     struct Worker* worker;
20     struct TaskManager* taskManager;
21     struct Context* task;
22     struct Element* taskList;
23 #ifdef USE_CUDAWorker
24     int num_exec;
25     CUmodule module;
26     CUfunction function;
27 #endif
28     /* multi dimension parameter */
29     int iterate;
30     struct Iterator* iterator;
31 };
32
33 union Data {
34     struct Meta {
35         enum DataType type;
36         long size;
37         long len;
```

```
38          struct Queue* wait; // tasks waiting this dataGear
39      } Meta;
40      struct Context Context;
41      struct Timer {
42          union Data* timer;
43          enum Code start;
44          enum Code end;
45          enum Code next;
46      } Timer;
47      struct TimerImpl {
48          double time;
49      } TimerImpl;
50      ....
51  }; // union Data end       this is necessary for context generator
```

ソースコード 2.2 は以下の内容を定義している。

- Code Gear の名前と関数ポインタとの対応表 Code Gear の名前とポインタの対応は Context 内の code(ソースコード 2.2 4 行目) に格納される。code は全ての Code Gear を列挙した enum と関数ポインタの組で表現される。Code Gear の名前は enum で定義され、コンパイル後には整数へと変換される。実際に Code Gear に接続する際は番号 (enum) を指定することで接続を行う。これにより、メタ計算の実行時に接続する Code Gear を動的に切り替えることが可能となる。

- Data Gear の Allocation 用の情報 Data Gear のメモリ空間は事前に領域を確保した後、必要に応じてその領域を割り当てることで実現する。実際に Allocation する際は Context 内の heap(ソースコード 2.2 8 行目) を Data Gear のサイズ分インクリメントすることで実現する。

- Code Gear が参照する DataGear へのポインタ Allocation で生成した Data Gear へのポインタは Context 内の data(ソースコード 2.2 6 行目) に格納される。Code Gear は data を参照して Data Gear へアクセスする。

- 並列実行用の Task 情報 Context は 並列実行の Task も兼任するため、待っている Input Data Gear のカウンタ、Input/Output Data Gear が格納されている場所を示すインデックス、GPU での実行フラグ等を持っている (ソースコード 2.2 13-30 行目)。

- Data Gear の型情報 Data Gear は構造体を用いて定義する (ソースコード 2.2 34-49 行目)。Timer や TimerImpl などの構造体が Data Gear に相当する。メタ計算では任意の Data Gear を一律に扱うため、全ての Data Gear の共用体を定義する (ソースコード 2.2 33-51 行目)。Data Gear を確保する際のサイズはこの型情報から決定する。

2.6 Interface

Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合を表現する Data Gear である。Interface には複数の実装を持つことができ、実装によって実行する Code Gear を切り替えることが可能になる。

Queue の Interface を ソースコード 2.3 に示す。Interface は API となる Code Gear を __code として 定義 (ソースコード 2.3 6-9 行目) する。この __code 実体は Code Gear への番号が格納される変数であり、Interface の実装毎に値は変化する。

ソースコード 2.3: Queue の Interface

```
1  typedef struct Queue<Impl>{
2          union Data* queue;
3          union Data* data;
4          __code next(...);
5          __code whenEmpty(...);
6          __code clear(Impl* queue, __code next(...));
7          __code put(Impl* queue, union Data* data, __code next(...));
8          __code take(Impl* queue, __code next(union Data*, ...));
9          __code isEmpty(Impl* queue, __code next(...), __code whenEmpty
       (...));
10 } Queue;
```

ソースコード 2.4 は Queue Interface を用いた SingleLinkedQueue の実装である。createSingleLinkedQueue(ソースコード 2.4 3-14 行目) は Queue Interface を実装した Data Gear の生成を行っている関数であり、データ構造の初期化と実装した Code Gear の番号 (enum) を Queue Interface で定義している Code Gear を示す変数に入れる (ソースコード 2.4 9-12 行目)。

ソースコード 2.4: SingleLinkedQueue の実装

```
1  #interface "Queue.h"
2
3  Queue* createSingleLinkedQueue(struct Context* context) {
4      struct Queue* queue = new Queue(); // Allocate Queue interface
5      struct SingleLinkedQueue* singleLinkedQueue = new SingleLinkedQueue()
       ; // Allocate Queue implement
6      queue->queue = (union Data*)singleLinkedQueue;
7      singleLinkedQueue->top  = new Element();
8      singleLinkedQueue->last = singleLinkedQueue->top;
9      queue->clear = C_clearSingleLinkedQueue;
10     queue->put  = C_putSingleLinkedQueue;
11     queue->take  = C_takeSingleLinkedQueue;
12     queue->isEmpty = C_isEmptySingleLinkedQueue;
13     return queue;
14 }
15
16 __code clearSingleLinkedQueue(struct SingleLinkedQueue* queue, __code
       next(...)) {
17     queue->top = NULL;
```

```
18      goto next(...);
19  }
20
21  __code putSingleLinkedQueue(struct SingleLinkedQueue* queue, union Data*
        data, __code next(...)) {
22      Element* element = new Element();
23      element->data = data;
24      element->next = NULL;
25      queue->last->next  = element;
26      queue->last = element;
27      goto next(...);
28  }
29
30  __code takeSingleLinkedQueue(struct SingleLinkedQueue* queue, __code next
        (union Data* data, ...)) {
31      struct Element* top = queue->top;
32      struct Element* nextElement = top->next;
33      if (queue->top == queue->last) {
34          data = NULL;
35      } else {
36          queue->top = nextElement;
37          data = nextElement->data;
38      }
39      goto next(data, ...);
40  }
41
42  __code isEmptySingleLinkedQueue(struct SingleLinkedQueue* queue, __code
        next(...), __code whenEmpty(...)) {
43      if (queue->top == queue->last)
44          goto whenEmpty(...);
45      else
46          goto next(...);
47  }
```

Interface での Code Gear 呼び出しは “goto interface-¿method” という構文で行う。ここの interface は createSingleLinkedQueue(ソースコード 2.4 3-14 行目) 等で初期化を行った Interface へのポインタ、method は実装した Code Gear の番号になる。

ソースコード 2.5 に Queue Interface を使用した Code Gear の呼び出し例を示す。この呼び出しでは SingleLinkedQueue の put 実装に継続される。

ソースコード 2.5: Queue Interface での Code Gear の呼び出し

```
1   #interface "Queue.h"
2
3   __code code1() {
4       Queue* queue = createSingleLinkedQueue(context);
5       Node* node = new Node();
6       node->color = Red;
7       goto queue->put(node, queueTest2);
8   }
```

ソースコード 2.5 は実際にはスクリプトによって ソースコード 2.6 に変換されコンパイ

ルされる。ソースコード 2.6 内の Gearef マクロは Context から Interface の引数格納用の Data Gear を取り出す。この引数格納用の Data Gear は Context の初期化の際に生成される。引数格納用の Data Gear を取り出した後は変換前の呼び出しの引数を Interface で定義した Code Gear の引数情報に合わせて格納し、 指定した Code Gear に継続する。

ソースコード 2.6: スクリプトによる変換後

```
1 __code code1(struct Context *context) {
2     Queue* queue = createSingleLinkedQueue(context);
3     Node* node = &ALLOCATE(context, Node)->Node;
4     node->color = Red;
5     Gearef(context, Queue)->queue = (union Data*) queue;
6     Gearef(context, Queue)->data = (union Data*) node;
7     Gearef(context, Queue)->next = C_queueTest2;
8     goto meta(context, queue->put);
9 }
```

2.7 stub Code Gear

第3章　Gears OSの並列処理

3.1　Allocator

3.2　TaskManager

3.3　Worker

3.4　SynchornizedQueue

3.5　Task

3.6　依存関係の解決

3.7　並列処理の記述

3.8　Iterator

3.9　待ち機構

第4章　GPGPU

第5章　Gears OS の評価

5.1　Twice

5.2　BitonicSort

第6章　まとめ的なやつ

謝辞

本研究の遂行、本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に心より感謝致します。そして、 共に研究を行い暖かな気遣いと励ましをもって支えてくれた並列信頼研究室の全てのメンバーに感謝致します。最後に、 有意義な時間を共に過ごした理工学研究科情報工学専攻の学友、並びに物心両面で支えてくれた家族に深く感謝致します。

2017 年 3 月
比嘉健太

参考文献

[1] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[2] Joachim (mathmaticien) Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, Cambridge, New York (N. Y.), Melbourne, 1986.

[3] Michael Barr and Charles Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice-Hall, 1990. Second edition, 1995.

[4] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.

[5] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993.

[6] Tokumori Kaito and Kono Shinji. The implementation of continuation based c compiler on llvm/clang 3.5. *IPSJ SIG Notes*, Vol. 2014, No. 10, pp. 1–11, may 2014.

[7] 信康大城, 真治河野. Continuation based c の gcc4.6 上の実装について. 第 53 回プログラミング・シンポジウム予稿集, 第 2012 巻, pp. 69–78, jan 2012.

[8] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2016/01/20(Fri).

[9] Welcome to agda's documentation! — agda 2.6.0 documentation. `http://agda.readthedocs.io/en/latest/index.html`. Accessed: 2016/01/31(Tue).

[10] Welcome! — the coq proof assistant. `https://coq.inria.fr/`. Accessed: 2016/01/20(Fri).

[11] Ats-pl-sys. `http://www.ats-lang.org/`. Accessed: 2016/01/20(Fri).

[12] Spin - formal verification. http://spinroot.com/spin/whatispin.html. Accessed: 2016/01/20(Fri).

[13] Nusmv home page. http://nusmv.fbk.eu/. Accessed: 2016/01/20(Fri).

[14] The cbmc homepage. http://www.cprover.org/cbmc/. Accessed: 2016/01/20(Fri).

[15] Opencl — nvidia developer. https://developer.nvidia.com/opencl. Accessed: 2016/02/06(Mon).

[16] Cuda zone — nvidia developer. https://developer.nvidia.com/cuda-zone. Accessed: 2016/02/06(Mon).

[17] 翔平小久保, 立樹伊波, 真治河野. Monad に基づくメタ計算を基本とする gears os の設計. Technical Report 16, 琉球大学大学院理工学研究科情報工学専攻, 琉球大学工学部情報工学科, 琉球大学工学部情報工学科, may 2015.

[18] 徳森海斗. Llvm clang 上の continuation based c コンパイラ の改良. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2016.

[19] 小久保翔平. Code segment と data segment を持つ gears os の 設計. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2016.

[20] 河野 真治比嘉 健太. Continuation based c を用いたプログラムの検証手法.

[21] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[22] B.C. Pierce. 型システム入門プログラミング言語と型の理論:. オーム社, 2013.

[23] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pp. 1–2, New York, NY, USA, 2009. ACM.

[24] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, Vol. 13, No. 8, pp. 165–180, August 1978.

[25] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Vol. 6, No. 4, pp. 308–320, January 1964.

[26] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jork, 1941.

[27] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, Vol. 27, No. 5, May 1992.

[28] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, Vol. 75, No. 5, pp. 381 – 392, 1972.

発表履歴

- 伊波立樹, 河野真治. 有線 LAN 上の PC 画面配信システム TreeVNC の改良. 第 57 回プログラミング・シンポジウム, Jan, 2016
- 伊波立樹, 東恩納琢偉, 河野真治. Code Gear、 Data Gear に基づく OS のプロトタイプ. 第 137 回 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, May, 2016