修士 (工学) 学位論文

Master's Thesis of Engineering

Gears OS の並列処理

Parallel processing in 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)　　名嘉村　盛和

印

(副　査)　　長田　智和

印

(副　査)　　河野　真治

要 旨

現在の OS では信頼性と拡張性を両立させることが要求されている。本研究室では 処理を Code Gear、 データを Data Gear という単位を用いて信頼性と拡張性をメタレベルで保証する Gears OS を開発している。

Gears OS の信頼性と拡張性は ノーマルレベルの計算に対して別の階層のメタレベルの計算される。このメタ計算は CPU、GPU などの実行環境の切り替え, データ拡張等を提供する。

Gears OS では Task を Code Gear と入力の Input Data Gear と出力の Output Data Gear の組で表現される。この Input/Output Data Gear によって依存関係を解決し、Input Data Gear が揃った Code Gear が並列実行される。

Code Gear と Data Gear に Interface を使ったモジュール化を導入した。これにより、Stack や Queue などのデータ構造を仕様と実装に分けて記述することができ、見通しの良いプログラミングが可能になった。

Gears OS の Task は par goto 構文により生成され、TaskManager を通して CPU、GPU の Worker に送信され Code Gear の実行を行う。par goto 構文を導入し、並列実行の記述を簡素化することができた。

本論文では Gears OS の基本概念、並列処理機構の実装について述べる。モジュール化、par goto 構文の実装についても考察する。また、並列処理を行う例題を用いて評価を行う。

Abstract

Reliability and extensibility is necessary in computer operating system. We are developing Gears OS, which has Code Gear and Data Gear as units of computation, and it also has meta level computations. Meta computations include GPU interface, parallel processing, memory managements, and synchronizations.

A task of Gears OS is a pair of Code Gear and Input/Output Data Gears. The Inputs and Outputs determine dependencies of the tasks. Gears kernel resolve the dependencies and execute the task.

New Introduce module system using interface, which defines a group of Code Gears and Data Gears. The interface the programming of Gears system clear. We also introduce par goto syntax for task creations.

In this paper, we describe concept of Gears OS and implementation of parallelism execution structure, module system, and par goto syntax. We evaluate Gears OS by parallel computation examples.

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

並列処理は現代主流のマルチコア CPU の性能を発揮するには重要なものになっている。しかし、並列処理のプログラミングはスレッド間の共通資源の競合など非決定的な実行を持っており、その信頼性を保証するには従来のテストやデバッグでは不十分であり、テストしきれない部分が残ってしまう。

また、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も並列処理をする上で無視できない。これらのプロセッサで性能を出すためには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様々なプロセッサを抽象化し、CPU と同等に扱えるようにする柔軟性が求められる。

本研究室では通常の計算をノーマルレベルとし、並列処理の信頼性をノーマルレベルの計算に対して保証し、拡張性をノーマルレベルとは別の階層のメタレベルの計算で実現することを目標に Gears OS[1] を設計、開発中である。Gears OS では処理を Code Gear、データを Data Gear という単位を用いてプログラムを記述する。Code Gear は入力の Input Data Gear が揃ったら実行され、Output Data Gear を出力する。この Input/Output Data Gear の関係から依存関係を決定し、Input Data Gear が揃った Code Gear が並列に実行される。

Gears OS のプログラムの信頼性の確保はモデル検査、検証を使用して行う。この信頼性のための計算はメタ計算として記述される。このメタ計算は信頼性の他に CPU、GPU などのアーキテクチャに沿った実行環境の切り替え、データの拡張等の拡張性を提供する。メタ計算の処理は Meta Code Gear、Meta Data Gear で表現する。Meta Code Gear は 通常の Code Gear 間に実行される。

従来の研究では OS の実装言語として Python[2] や Haskell[3][4] をノーマルレベルとして採用し、メタレベルで検証を行う研究や、メタ計算の実装を型付きアセンブラ (Typed Assembler)[5] を用いる例もある。Gears OS は ノーマルレベルとメタレベルを共通して表現出来る Continuation Based C(CbC)[6] で実装を行い、証明支援系 Agda[7] を用いて証明を行う。CbC は Code Gear の単位でプログラムを記述し、軽量継続を用いてコード間を移動する。軽量継続は関数呼び出しとは異なり、呼び出し元に戻らないため、呼び出し元の環境を覚えずに行き先のみを指定する。この CbC は C と互換性のある言語で、型付きアセンブラに比べると大きな表現力を提供する。また Haskell などに比べて関数呼び

出しではなく軽量継続を採用しているため、スタック上に隠された環境を持たないため、メタレベルで使用する資源を明確にできる利点がある。

Code Gear 間の継続は次の Code Gear の番号と Context という全ての Code Gear と Data Gear を参照できる Meta Data Gear で行われる。Context からノーマルレベルの Code Gear へ接続する際は stub Code Gear という Meta Code Gear を用いる。この stub Code Gear はメタレベルの計算であるため、継続先の Code Gear から自動的に生成されるのが望ましい。生成に必要な情報は Code Gear と Data Gear の集まりから得る。 この集まりを Interface[8] として定義している。

Gears OS でのプロセス、スレッドは Context が担う。並列実行する際は新しく Context を生成し、それを CPU、GPU に割り振る事によって実現される。生成された Context には実行する Code Gear と対応する Input/Output Data Gear が登録され、割り振られた先で Context に設定された Code Gear を実行する。この Context を用いた並列処理は新規に実行環境を作り、引数を設定するなどの煩雑なメタレベルの処理であり、ノーマルレベルでは Go 言語 [9] の goroutine のような簡潔な並列構文があることが望ましい。

本研究では Gears OS の基本設計、マルチコア CPU と CUDA による GPU での実行機構、並列構文を実装し、例題を用いて Gears OS の並列処理の評価を行う。

第2章　Gears OS の概念

Gears OS は信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に開発している OS である。

Gears OS は処理の単位を Code Gear、データの単位を Data Gear と呼ばれる単位でプログラムを構成する。信頼性や拡張性はメタ計算として、通常の計算とは区別して記述する。

本章では Gears OS を構成する様々な要素について説明する。

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。

また Gears OS 自体もこの Code Gear、Data Gear を用いた CbC(Continuation based C) で実装される。そのため、Gears OS の実装は Code Gear、Data Gear を用いたプログラミングスタイルの指標となる。

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 の処理系として llvm[10]/clang と gcc による実装などが存在する [11][12]。

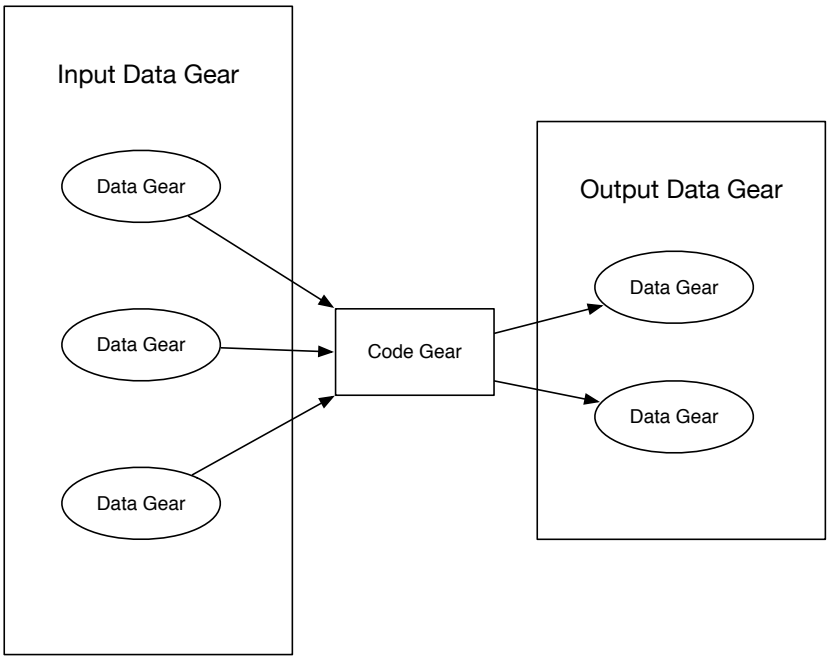

図 2.1: Code Gear と Data Gear の関係

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

CbC の goto 文による継続は 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。

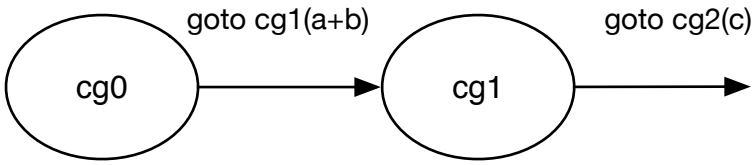

図 2.2: goto 文による Code Gear の軽量継続

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[13] を用いて表現される [14]。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。

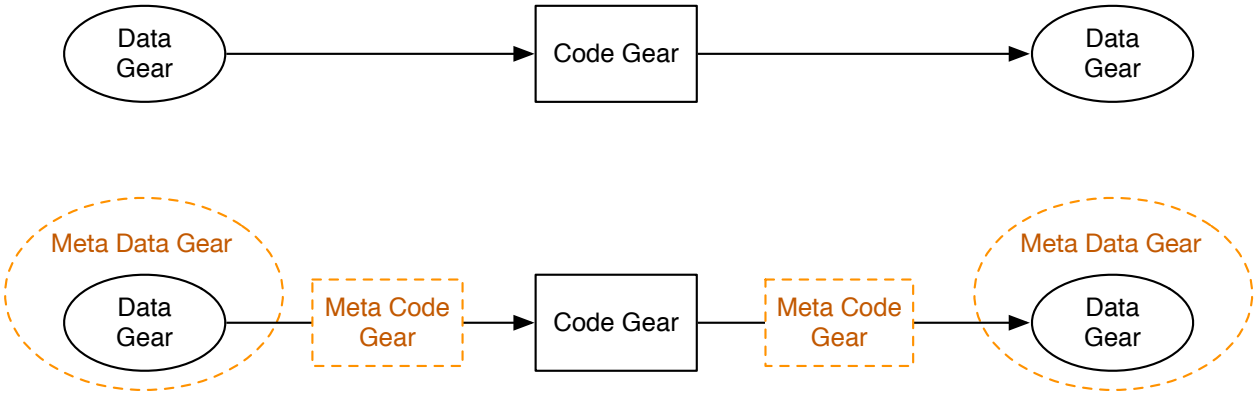

図 2.3: Meta Code Gear の実行

2.5 Context

Context とは一連の実行で使用される Code Gear と Data Gear の集合である。従来のスレッドやプロセスに対応する。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。

ソースコード 2.2 に Context の定義を示す。

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 へのポインタは番号を割り振り、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 から番号を指定して 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 stub Code Gear

stub Code Gear は Code Gear の接続の間に挟まれる Meta Code Gear である。ノーマルレベルの Code Gear から Meta Data Gear である Context を直接参照してしまうと、ユーザがメタ計算をノーマルレベルで自由に記述できてしまい、メタ計算を分離した意味がなくなってしまう。stub Code Gear はこの問題を防ぐため、Context から必要な Data Gear のみを ノーマルレベルの Code Gear に渡す処理を行っている。

stub Code Gear は使用される全ての Code Gear 毎に記述する必要がある。しかし、全ての Code Gear に対して stub Code Gear を記述するのは膨大な記述量になってしまうため、後述する Interface を実装した Code Gear の stub Code Gear はスクリプトで自動生成する。

stub Code Gear はユーザーが自前で記述することも可能である。つまり、ユーザーがメタ計算を記述することができる。stub Code Gear を用いたメタ計算の例として、本来 stub Code Gear は対応した Code Gear に継続するが、自前で stub Code Gear を記述することで、継続先を柔軟に変更できる。

第3章　Gears OS のモジュール化

Gears OS は stub Code Gear という Meta Code Gear で Context という全ての Code Gear と Data Gear を持った Meta Data Gear から値を取りだし、ノーマルレベルの Code Gear に値を渡す。しかし、Gears OS を実際に実装するにつれて、メタレベルからノーマルレベルへの継続の記述が煩雑になることがわかり、Code Gear と Data Gear のモジュール化が必要になった。

本章では Gears OS のモジュール化の仕組みである Interface について説明する。

3.1　Context を経由しての継続の問題点

Gears OS は Code Gear で必要な Input Data Gear を Context から番号を指定して取り出すことで処理を実行する。Context はプログラム全体でみると使用する全ての Code Gear と Data Gear の集合を表現する Meta Data Gear になっている。しかし、Gears OS を実装する上で Context から Code Gear と Data Gear の番号の組合せを全て展開すると Code Gear がどの Data Gear の番号に対応するかを stub Code Gear に書く必要があり、記述が煩雑になってしまった。また、stub Code Gear の記述の煩雑さを避けるために、決まった番号に決まった型の Data Gear を生成し、その Data Gear を複数の Code Gear で使いまわすという、Data Gear をグローバル変数のように扱う問題が多発した。

この問題点は Context が全ての Code Gear と Data Gear の集合を表現するために起こった問題である。そこで、Gears OS をモジュール化する仕組みとして Interface を導入した。Interface は ある Data Gear の定義と、それに対する操作 (API) を行う Code Gear の集合を表現する Meta Data Gear である。Context では全ての Code Gear と Data Gear の集合を表現してることに比べ、Interface は 一部の Data Gear と 一部の Code Gear の集合を表現する。この Interface は Java の インターフェース、Haskell の型クラスに対応する。

3.2　Interface の定義

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 Meta 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。この Code Gear は C++ の virtual 関数、Java の abstract メソッドに対応する。

ソースコード 3.1 に Queue の Interface を示す。Interface には以下の内容を定義する。

- 引数の Data Gear 群

  ソースコード 3.1 3-4 行目は 引数の Data Gear 群を定義している。ここで定義された Data Gear 名は、定義された Code Gear の引数に対応する。

  この Interface では 10 行目で Queue に要素を挿入する Code Gear を定義しており、引数として挿入する Queue の実装と挿入する要素を受け取る。この引数それぞれが 3-4 行目で定義した queue と data に対応する。

- Interface が所属する Code Gear の実行後に継続される Code Gear

  Interface の Code Gear は基本的には実行後の継続先は不定となっており、継続元から渡される。継続元から渡される値 は ソースコード 3.1 5-6 行目に定義している変数に格納される。“__code next(...)” の引数である “...” は複数の Input Data Gear を持つという意味である。この “...” は他のプログラミング言語では可変長引数のような扱いである。また、実行する Code Gear によってこの継続は複数設定される場合がある。

  例えば、この Interface では 12 行目で Queue が空かどうかを調べる Code Gear を定義しており、中身がある場合と空の場合で別の継続を渡す必要がある。

- 操作 (API) である Code Gear と Code Gear に渡す引数情報

  操作 (API) に対応する Code Gear は ソースコード 3.1 9-12 行目 のように __code として定義する。この __code の実体は Code Gear への番号が格納される変数であり、実装した Code Gear に対応する番号を代入する。Code Gear の引数には Data Gear と Code Gear 実行後に継続される Code Gear 等を記述する。引数の Data Gear はその Code Gear の Input Data Gear になり、引数の Code Gear の中の引数が Output Data Gear になる。Code Gear の第一引数には Interface を実装した Data Gear を渡す。これは、Code Gear の操作の対象となる Data Gear を設定しており、後述する継続構文では引数として記述を行わない。

  この Interface では 11 行目で Queue から要素の取り出しを行う Code Gear を定義しており、引数として取り出す Queue の実装と、Code Gear 実行後に継続される Code Gear を受け取る。引数の Code Gear である “__code next(union Data*, ...)“ の “(union Data*, ...)” は Queue の要素取り出しを行う Code Gear の Output Data Gear であり、実行後に継続される Code Gear の Input Data Gear になる。

ソースコード 3.1: Queue の Interface

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

3.3 Interface の実装

Interface は Data Gear に対しての操作 (API) を行う Code Gear とその Code Gear で扱われている Data Gear の集合を抽象的に表現した Meta Data Gear であり、実装は別に定義する。Interface の実装は、実装する Data Gear の初期化と実装した Code Gear を Interface で定義した Code Gear に代入することで行う。この代入する Code Gear を入れ替えることで操作 (API) は同じで処理は別の実装を複数表現することが出来る。また、実装された Code Gear は引数の Data Gear と 実装した Data Gear 以外にアクセスすることはない。

Interface で指定された Code Gear 以外の Code Gear も実装することが出来る。このような Code Gear は基本的に Interface で指定された Code Gear 内からのみ継続されるため、Java の private メソッドのように扱われる。この Code Gear も Interface で指定された Code Gear と同じく外から渡された Data Gear にアクセス出来る。

ソースコード 3.2 は Queue Interface(ソースコード 3.1) を用いた SingleLinkedQueue の実装である。Interface で実装した Data Gear の生成は関数呼び出しで行われる。createSingleLinkedQueue 関数 (ソースコード 3.2 3-14 行目) は実装した Data Gear の生成を行っている。この関数は生成する Data Gear の初期化 (ソースコード 3.2 7-8 行目) と、実装した Code Gear を Interface で定義した Code Gear の代入 (ソースコード 3.2 9-12 行目) を行う。実際に実装する Data Gear は Interface の型に包まれて生成される (ソースコード 3.2 6 行目)。このように生成することで実装した Data Gear は実装以外の場所からは Interface の型として扱う事ができる。

ソースコード 3.2: 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 .....
```

3.4　Interface を利用した Code Gear の継続

Gears OS では Interface を利用した Code Gear の継続用に “goto interface->method” という構文を提供している。Interface を実装した Data Gear は外からは Interface の型として扱われる。そのため構文の “interface“ は実装した Data Gear を Interface の型で包んだポインタ、method は実装した Code Gear に対応する。

ソースコード 3.3 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 3.3: 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 }
```

“goto interface->method” という構文は実際にはスクリプトで変換され、コンパイルされる。変換後のコードはメタレベルのコードとなるため、Context をマクロを経由し、直接参照を行う。ソースコード 3.4 は ソースコード 3.3 がスクリプトによってに変換されたソースコードを示しており、 図 3.1 は ソースコード 3.4 が実行された際の Queue Interface と Code Gear、 Data Gear の関係を示している。

ソースコード 3.4 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 の初期化の際に特別に生成され、型は使用される Interface の型と同じである。また、引数格納用の Data Gear はノーマルレベルでは参照されず、メタレベルの場合のみ参照される。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 3.4 では Queue Interface の put を継続しているため、6 行目で Input Data Gear として node Data Gear を 引数格納用の Data Gear の data に代入し、7 行目で実行後に継続する Code Gear として queueTest2 を 引数格納用の Data Gear の next に代入している。代入した引数は自動生成された stub Code Gear(ソースコード 3.5) で展開され、実装された Code Gear に Data Gear を渡す。

ソースコード 3.4: 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 }
```

ソースコード 3.5: スクリプトによって生成された put stub Code Gear

```
1  __code putSingleLinkedQueue(struct Context *context,struct
       SingleLinkedQueue* queue, union Data* data, enum Code next) {
2      Element* element = &ALLOCATE(context, Element)->Element;
3      element->data = data;
4      element->next = NULL;
5      queue->last->next  = element;
6      queue->last = element;
7      goto meta(context, next);
8  }
9
10 // generated by script
11 __code putSingleLinkedQueue_stub(struct Context* context) {
```

```
12      SingleLinkedQueue* queue = (SingleLinkedQueue*)GearImpl(context,
    Queue, queue);
13      Data* data = Gearef(context, Queue)->data;
14      enum Code next = Gearef(context, Queue)->next;
15      goto putSingleLinkedQueue(context, queue, data, next);
16  }
```

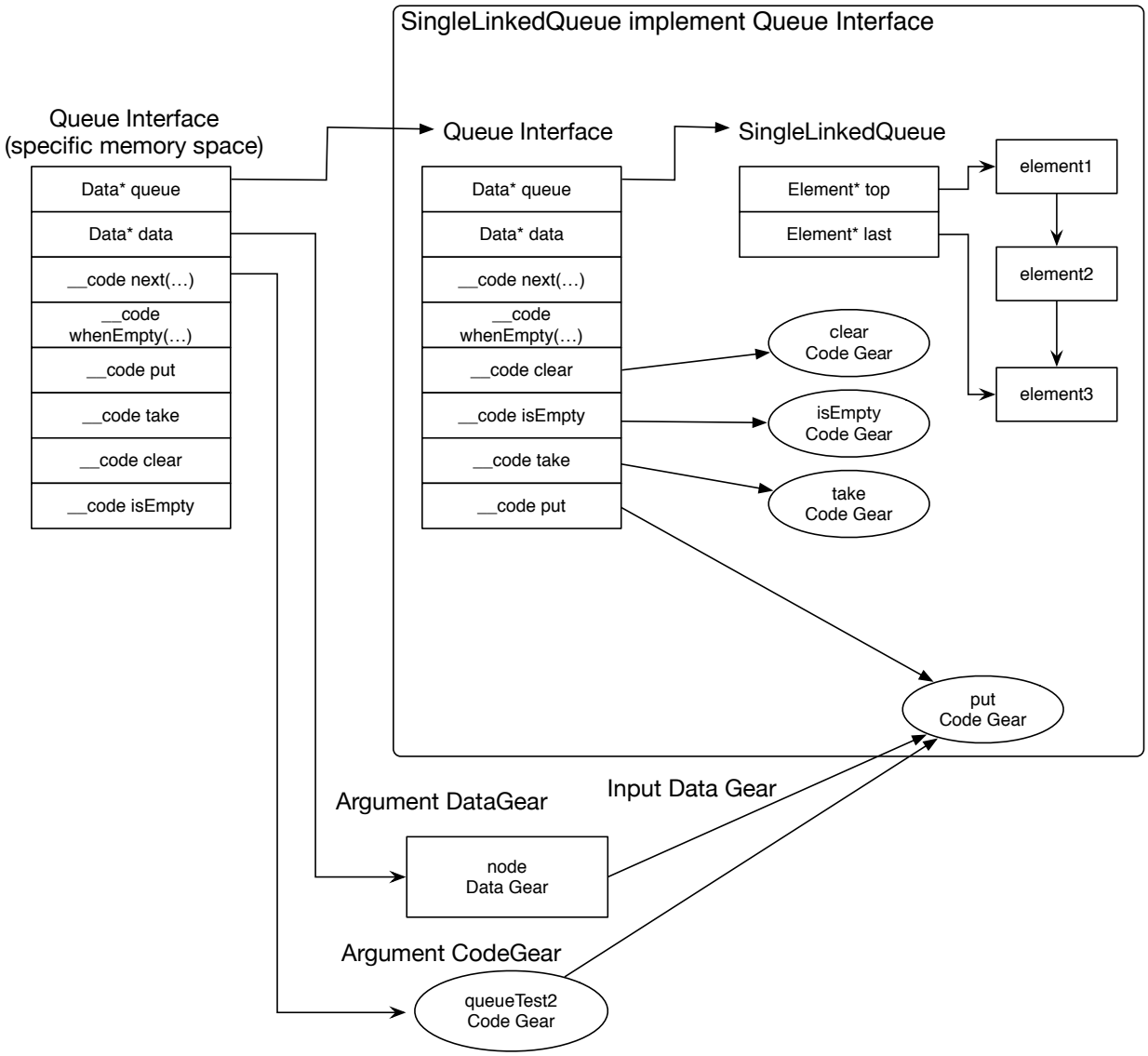

図 3.1: Queue Interface で実装した put Code Gear の呼び出し

第4章　Gears OSの並列処理

Gears OS では実行の Task を Code Gear と Input/Output Data Gear の組で表現する。Input/Output Data Gear によって依存関係が決定し、それにそって並列実行を行う。

本章では、Gears OS の並列処理の構成、機能について説明する。

4.1　Task

Gears OS では 並列実行する Task を Context で表現する [15]。Context には Task 用の情報として、実行される Code Gear、Input/Output Data Gear の格納場所、待っている Input Data Gear のカウンタ等を持っている。Task の Input Data Gear が揃っているかを TaskManager で判断し、 実行可能な Task を Worker に送信する。Worker は送信された Task が指定した Code Gear を実行し、Output Data Gear を書き出す。

実行される Code Gear の例を ソースコード 4.1 に示す。ソースコード 4.1 は Integer 型 の Input Data Gear を 2 つ受け取り、加算処理を行い、Integer 型 の Output Data Gear に書き出す。並列処理を行う Code Gear は Interface の Code Gear と同じく、引数に Input Data Gear、処理が終了した後に継続する Code Gear、引数の Code Gear の中に Output Data Gear を記述する (ソースコード 4.1 1 行目)。この Code Gear 実行後は通常 Output Data Gear を書き出す Code Gear に継続する。実際に Output Data Gear を書き出す場合、goto 文に Output Data Gear を引数に渡す (ソースコード 4.1 3 行目)。

ソースコード 4.1: 並列実行される Code Gear の例

```
1 __code add(struct Integer* input1, struct Integer* input2, __code next(
      struct Integer* output, ...)) {
2     output->value = input1->value + input2->value;
3     goto next(output, ...);
4 }
```

Task の Input/Output Data Gear の格納場所は Context の Task 情報が持っている。その為、Task に対応する Code Gear の stub Code Gear は context が持っている Input/Output Data Gear 用のインデックスから Data Gear を取り出す。ソースコード 4.2 に ソースコード 4.1 の stub Code Gear を示す。この stub Code Gear も Interface の stub Code Gear と同等にスクリプトによって自動生成される。

ソースコード 4.2: 並列実行される Code Gear の stub Code Gear

```
1  __code add_stub(struct Context* context) {
2      // Input Data Gear
3      Integer* input1 = &context->data[context->idg + 0]->Integer;
4      Integer* input2 = &context->data[context->idg + 1]->Integer;
5
6      // set Continuation
7      enum Code next = context->next;
8
9      // Output Data Gear
10     Integer** O_output = (Integer **)&context->data[context->odg + 0];
11     goto add(context, input1, input2, next, O_output);
12 }
```

4.2 TaskManager

Gears OS の TaskManager は Task を実行する Worker の生成、管理、Task の送信を行う。ソースコード 4.3 に TaskManager の Interface を示す。

ソースコード 4.3: TaskManager の Interface

```
1  typedef struct TaskManager<Impl>{
2      union Data* taskManager;
3      struct Context* task;
4      struct Element* taskList;
5      __code spawn(Impl* taskManager, struct Context* task, __code next
       (...));
6      __code spawnTasks(Impl* taskManagerImpl, struct Element* taskList,
       __code next1(...));
7      __code setWaitTask(Impl* taskManagerImpl, struct Context* task,
       __code next(...));
8      __code shutdown(Impl* taskManagerImpl, __code next(...));
9      __code incrementTaskCount(Impl* taskManagerImpl, __code next(...));
10     __code decrementTaskCount(Impl* taskManagerImpl, __code next(...));
11     __code next(...);
12     __code next1(...);
13 } TaskManager;
```

TaskManager は 以下の API を持っている。

- Task の実行 (spawn、spawnTasks)
- Task の依存関係の設定 (setWaitTask)
- TaskManager が管理している Task 数のインクリメントとデクリメント (increment/decrementTaskCount)
- TaskManager(shutdown) の終了処理

TaskManager は初期化の際に、指定した数の Worekr を生成する。その際 CPU、GPU の数を指定することができ、指定した分の CPUWorker と GPUWorker が生成される。

TaskManager は 図 4.1 に示すように spawn を呼び出した際、実行する Task の Input Data Gear が用意されているかを判断する。Input Data Gear が全て用意されている場合、その Task を Worker の Queue に送信する。送信する Worker は Task を実行する環境 (CPU、GPU) によって決定する。

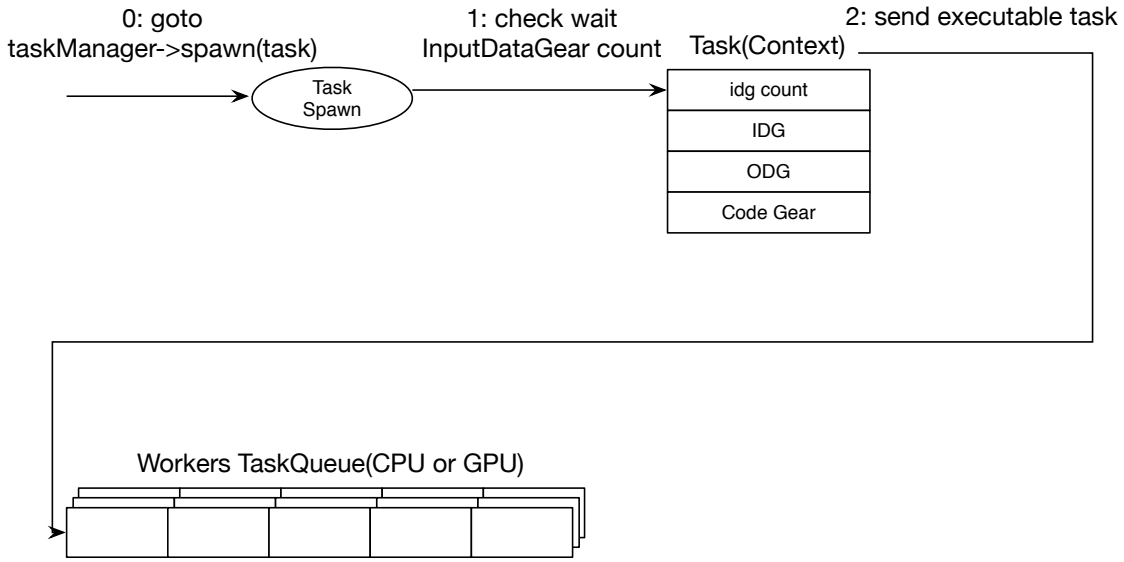

図 4.1: Worker への Task 送信

4.3 Worker

Worker は自身の Queue から Task を取得し、Task の Code Gear を実行し、Output Data Gear の書き出しを行っている。

ソースコード 4.4 に Task を CPU で実行する CPUWorker の初期化部分を示す。CPUWorker は初期化の際に スレッドを生成する (ソースコード 4.4 10 行目)。生成されたスレッドはまず startWorker 関数 (ソースコード 4.4 14-21 行目) を呼び出し、このスレッド用の Context を生成する。Context をスレッド毎に生成することで、メモリ空間をスレッドごとに持てるため Gearef マクロ で Interface の引数を取得する際の競合、メモリ確保の処理での他のスレッドの停止を防ぐ事ができる。

ソースコード 4.4: CPUWorker の初期化

```
1  Worker* createCPUWorker(struct Context* context, int id, Queue* queue) {
2      struct Worker* worker = new Worker();
3      struct CPUWorker* cpuWorker = new CPUWorker();
4      worker->worker = (union Data*)cpuWorker;
5      worker->tasks = queue;
6      cpuWorker->id = id;
7      cpuWorker->loopCounter = 0;
8      worker->taskReceive = C_taskReceiveCPUWorker;
9      worker->shutdown = C_shutdownCPUWorker;
10     pthread_create(&worker->thread, NULL, (void*)&startWorker, worker);
11     return worker;
12 }
13
14 static void startWorker(struct Worker* worker) {
15     struct CPUWorker* cpuWorker = &worker->worker->CPUWorker;
16     cpuWorker->context = NEW(struct Context);
17     initContext(cpuWorker->context);
18     Gearef(cpuWorker->context, Worker)->worker = (union Data*)worker;
19     Gearef(cpuWorker->context, Worker)->tasks = worker->tasks;
20     goto meta(cpuWorker->context, worker->taskReceive);
21 }
```

Context の生成後は Queue から Task を取得する Code Gear へ継続する。Task は Context なので、Worker は Context を入れ替えて Task の実行を行う。

CPUWorker での Task の実行をソースコード 4.5、 図 4.2 に示す。ソースコード 4.5 は Context の入れ替えを行うため、getTaskCPUWorker(ソースコード 4.5 1-9 行目) の引数に入れ替え後の Task(Context) を記述している。Worker は中身が NULL の task を取得すると Worker の終了処理を行う (ソースコード 4.5 2-4 行目)。Task が取得できた場合 Task の実行後に継続する Code Gear を格納し (ソースコード 4.5 7 行目)、Task を Context として Code Gear に継続する (ソースコード 4.5 8 行目)。Task の実行後に継続する Code Gear は Data Gear の書き出しと依存関係の解決を行う。Data Gear 書き出し後は Task の Context を Worker の Context に入れ替え、再び Task を取得する Code Gear に継続する。

ソースコード 4.5: CPUWorker での Task の実行

```
1  __code getTaskCPUWorker(struct CPUWorker* cpuWorker, struct Context* task
       , struct Worker* worker) {
2      if (!task) {
3          goto worker->shutdown(); // end thread
4      }
5      task->worker = worker;
6      enum Code taskCg = task->next;
7      task->next = C_odgCommitCPUWorker; // commit outputDG after task exec
8      goto meta(task, taskCg); // switch task context
9  }
```

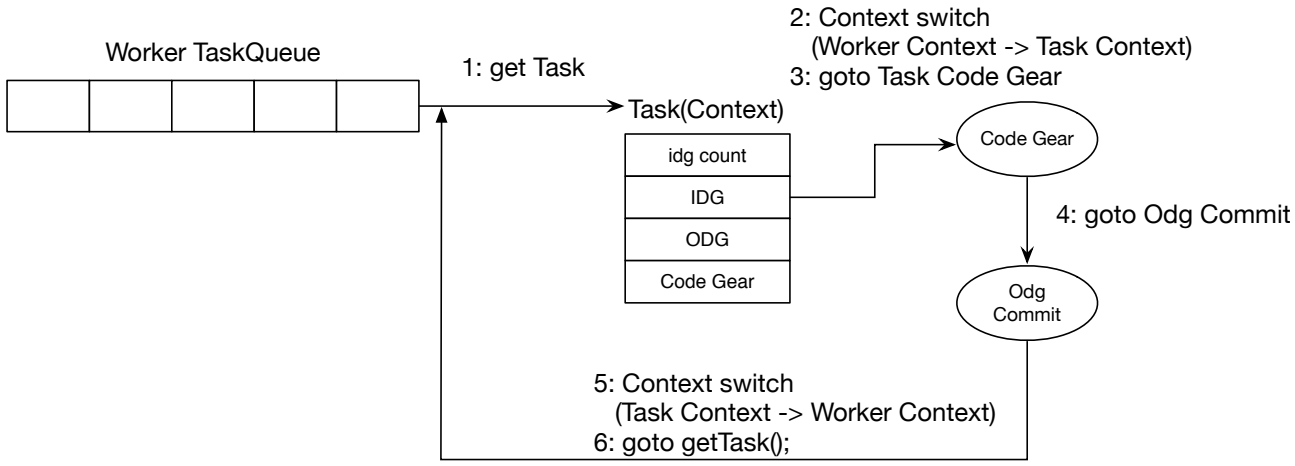

図 4.2: Worker での Task 実行

4.4 SynchronizedQueue

SynchronizedQueue は Worker の Queue として使用される。Worker の Queue は TaskManager を経由して Task を送信するスレッドと Task を取得する Worker 自身のスレッドで扱われる。そのため SynchronizeQueue はマルチスレッドでもデータの一貫性を保証する Queue を実装する必要がある。

データの一貫性を保証する解決例としての 1 つとしてロックを使った解決方法がある。しかし、ロックを行ってデータを更新した場合、同じ Queue に対して操作を行う際に待ち合わせが発生し、全体の並列度が下がってしまう。そこで、Gears OS ではデータの一貫性を保証するために CAS(Check and Set、Compare and Swap) を利用した Queue[16] を実装している。CAS は値の比較、更新をアトミックに行う命令である。CAS を使う際は更新前の値と更新後の値を渡し、渡された更新前の値を実際に保存されているメモリ番地の値と比較し、同じならデータ競合がないため、データの更新に成功する。異なる場合は他に書き込みがあったとみなされ、値の更新に失敗する。

Gears OS ではこの CAS を行うための Interface を定義している (ソースコード 4.6)。この Interface では、Data Gear 全てを内包している Data 共用体のポインタの値を更新する CAS を定義している (ソースコード 4.6 6 行目)。

ソースコード 4.6: AtomicInterface

```
1 typedef struct Atomic<Impl>{
2     union Data* atomic;
3     union Data** ptr;
4     union Data* oldData;
5     union Data* newData;
```

```
6     __code checkAndSet(Impl* atomic, union Data** ptr, union Data*
      oldData, union Data* newData, __code next(...), __code fail(...));
7     __code next(...);
8     __code fail(...);
9 } Atomic;
```

AtomicInterface での CAS の実際の実装を ソースコード 4.7 に示す。実際の実装では __sync_bool_compare_and_swap 関数を呼び出すことで CAS を行う (ソースコード 4.7 2 行目)。この関数は第一引数に渡されたアドレスに対して第二引数の値から第三引数の値へ CAS を行う。CAS に成功した場合、true を返し、失敗した場合は false を返す。ソースコード 4.7 では CAS に成功した場合と失敗した場合それぞれに対応した Code Gear へ継続する。

ソースコード 4.7: CAS の実装

```
1 __code checkAndSetAtomicReference(struct AtomicReference* atomic, union
      Data** ptr, union Data* oldData, union Data* newData, __code next(...)
      , __code fail(...)) {
2     if (__sync_bool_compare_and_swap(ptr, oldData, newData)) {
3         goto next(...);
4     }
5     goto fail(...);
6 }
```

SynchronizedQueue の Data Gear の定義を ソースコード 4.8 に示す。SynchronizedQueue はデータのリストの先頭と、終端のポインタを持っている。Queue を操作する際はこのポインタに対して CAS をすることでデータの挿入と取り出しを行う。

ソースコード 4.8: SynchronizedQueue の定義

```
1 struct SynchronizedQueue {
2     struct Element* top;
3     struct Element* last;
4     struct Atomic* atomic;
5 };
6
7 // Singly Linked List element
8 struct Element {
9     union Data* top;
10    struct Element* next;
11 };
```

図 4.3 は要素の取り出し (dequeue) を行った流れを示している。データを取り出す際はリストの先頭を次の要素へ CAS することででデータを取得する。この Queue では先頭に挿している要素はダミー扱いとする。その為、実際に取り出される値は CAS に成功した後の先頭の値となる。

図 4.4 は要素の挿入 (inqueue) を行った流れを示している。データを挿入する際は 2 度の CAS を行う。まず、末尾の要素の次の要素を新しい要素に CAS を行う。その後、末

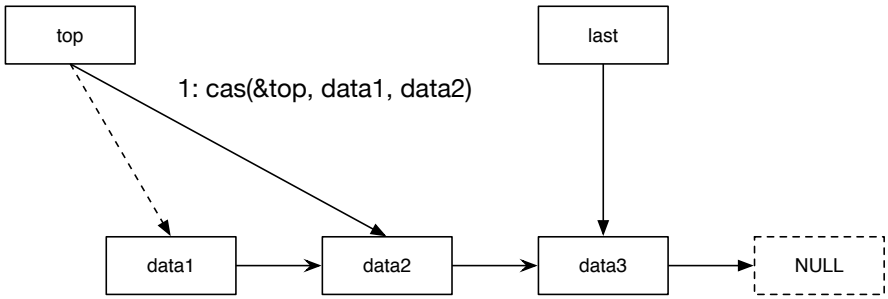

図 4.3: SynchronizedQueue による要素の取り出し

尾の要素が差している要素を挿入に成功した新しい要素に CAS を行う。

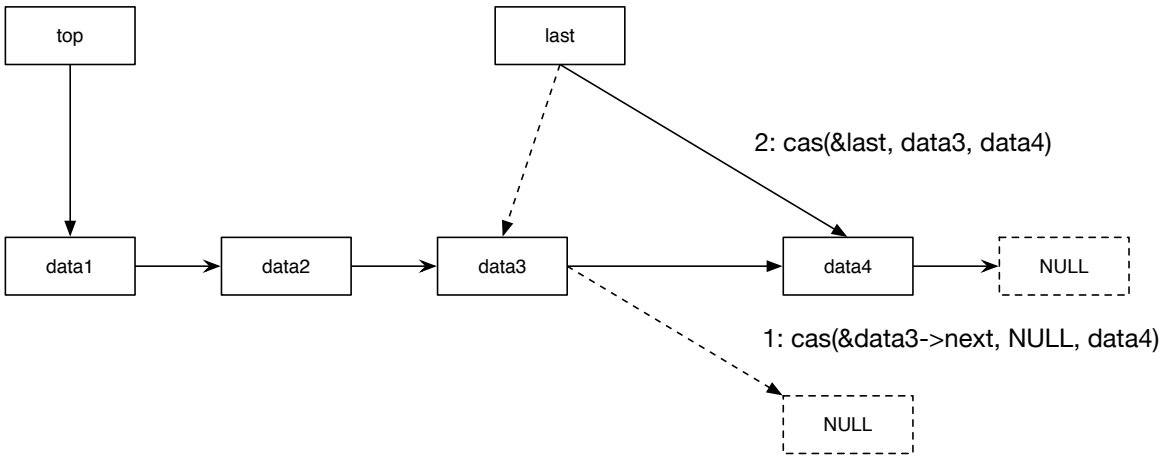

図 4.4: SynchronizedQueue による要素の挿入

しかし、データの挿入する際は 2 度の CAS の間に他のスレッドの操作が入り、Queue の構造が破綻する場合がある。例えば図 4.5 に示すように、あるスレッドが末尾を更新した際に、他のスレッドが更新処理を行うと Queue が破綻する。

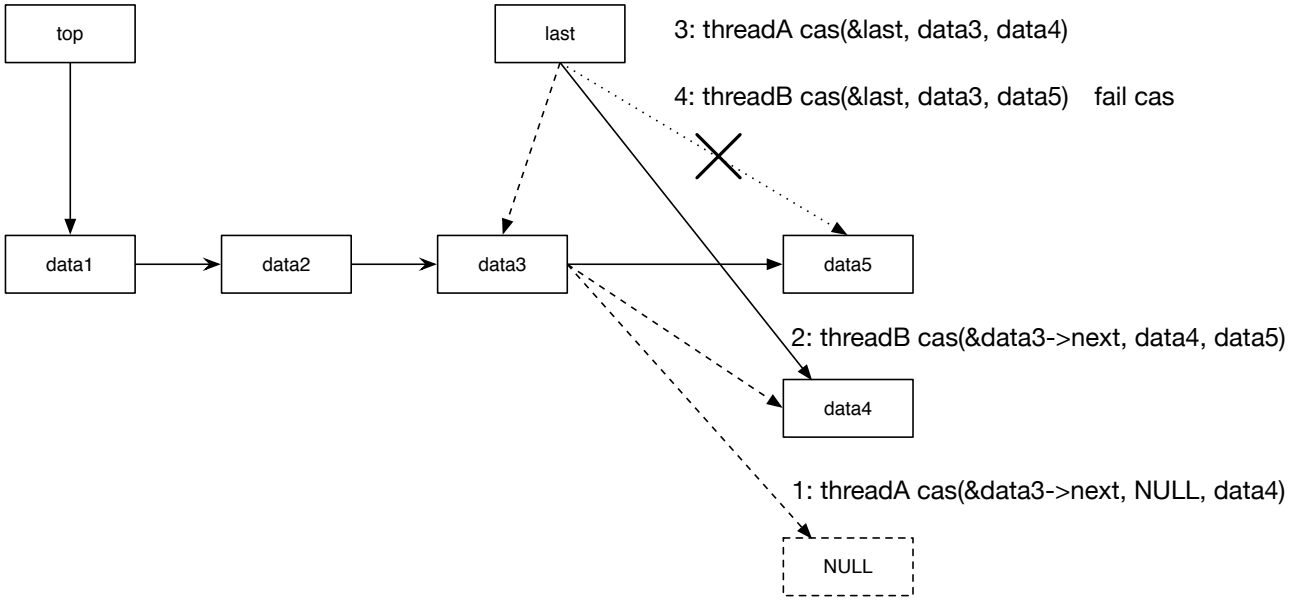

図 4.5: SynchronizedQueue によるデータの挿入時の破綻例

図 4.5 は 2 つのスレッドが以下の順番で処理を行っている。

1. threadA: 末尾の要素 (data3) の次の要素を新しい要素 (data4) に CAS を実行する
2. threadB: threadA での末尾更新をする前に末尾の要素 (data3) の次の要素を新しい要素 (data5) に CAS を実行する。この時の末尾が指す要素は、threadA が要素挿入を行う前の状態と同じなため、この操作を行うとリストが破綻する。
3. threadA: 末尾の要素 (data3) を挿入に成功した要素 (data4) に CAS を行う。
4. threadB: 末尾の要素 (data3) を挿入に成功した要素 (data5) に CAS を行う。threadB が挿入の操作を行ったときの末尾は threadA が末尾更新する前の末尾要素なので、この CAS は失敗する。

この破綻は末尾の要素が必ず末尾を示していると仮定しているためである。しかし、データ挿入の際は 2 度の CAS の間に他のスレッドが割り込む場合がある。そこで、末尾の要素は必ずしも末尾を示さないと仮定して要素の取出しと挿入の処理を記述する。ソースコード 4.9 は要素の挿入の処理の一部である。末尾の要素が末尾を示さない場合の処理はソースコード 4.9 の 13-16 行目に記述している。変数 nextElement は 末尾要素の次の要素を示しており、NULL ではない場合は末尾を差していない状態ということになる。

その場合は、末尾の要素を nextElement に CAS を行う。この処理は要素の取り出しを行う際にも行われる。

ソースコード 4.9: SynchronizedQueue による要素の挿入

```
1  __code putSynchronizedQueue(struct SynchronizedQueue* queue, union Data*
       data, __code next(...)) {
2      Element* element = new Element();
3      element->data = data;
4      element->next = NULL;
5      Element* last = queue->last;
6      Element* nextElement = last->next;
7      if (last != queue->last) {
8          goto putSynchronizedQueue();
9      }
10     if (nextElement == NULL) {
11         struct Atomic* atomic = queue->atomic;
12         goto atomic->checkAndSet(&last->next, nextElement, element, next
       (...), putSynchronizedQueue);
13     } else { // wrong last element
14         struct Atomic* atomic = queue->atomic;
15         goto atomic->checkAndSet(&queue->last, last, nextElement,
       putSynchronizedQueue, putSynchronizedQueue);
16     }
17 }
```

4.5 依存関係の解決

Gears OS は並列処理の依存関係を Input と Output の Data Gear と Code Gear の関係で解決する。Code Gear は Input に指定した Data Gear が全て書き込まれると実行され、実行した結果を Output に指定した Data Gear に書き出しを行う。

Data Gear はメタレベルで依存関係解決のための Queue を持っている。この Queue にはその Data Gear を Input Data Gear として使用する Task(Context) が入っている。

依存関係の解決の流れを図 4.6 に示す。Worker は Task の Code Gear を実行後、Output Data Gear の 書き出し処理 (Commit) を行う。書き出し処理は Data Gear の Queue から、依存関係にある Task を参照する。参照した Task には実行に必要な Input Data Gear のカウンタをもっているので、そのカウンタのデクリメントを行う。カウンタが 0 になったら Task が待っている Input Data Gear が揃ったことになるので、その Task を TaskManager 経由で 実行される Worker に送信する。

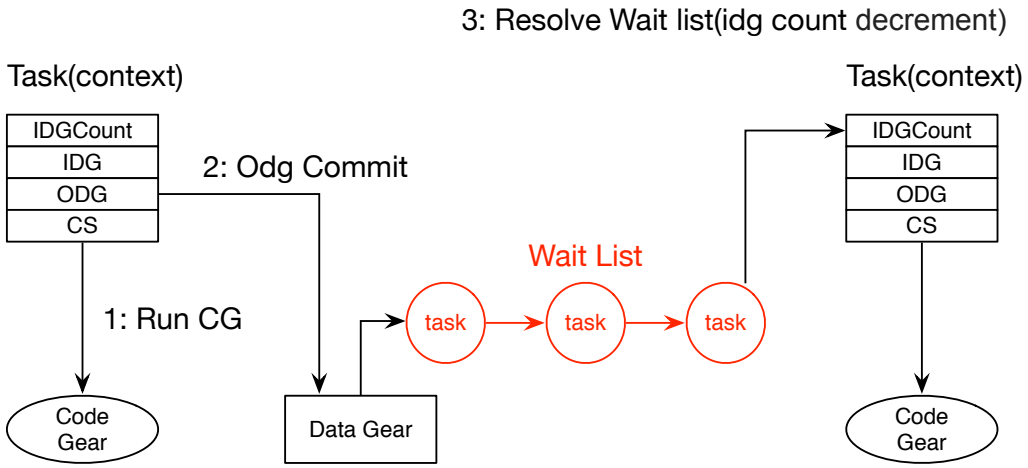

図 4.6: 依存関係の解決処理

4.6　並列構文

Gears OS では並列実行する Task の設定をメタレベルで ソースコード 4.10 のように行っている。ソースコード 4.10 では実行する Code Gear、待ち合わせ中の Input Data Gear の数、Input/Output Data Gear への参照等の設定を記述している。これらの記述は Context などを含む煩雑な記述であるが、並列実行されることを除けば通常の CbC の goto 文と同等である。そこで、Context を直接用いないノーマルレベルの並列構文である par goto 文を用意した。

ソースコード 4.10: メタレベルによる Task の生成

```
1  __code code1(struct Context *context, Integer *integer1, Integer *
       integer2, Integer *output) {
2      // create context
3      context->task = NEW(struct Context);
4      initContext(context->task);
5
6      // set task parameter
7      context->task->next = C_add;
8      context->task->idgCount = 2;
9      context->task->idg = context->task->dataNum;
10     context->task->maxIdg = context->task->idg + 2;
11     context->task->odg = context->task->maxIdg;
12     context->task->maxOdg = context->task->odg + 1;
13
14     // create Data Gear Queue
15     GET_META(integer1)->wait = createSynchronizedQueue(context);
16     GET_META(integer2)->wait = createSynchronizedQueue(context);
17     GET_META(integer3)->wait = createSynchronizedQueue(context);
```

```
18
19      // set Input Data Gear
20      context->task->data[context->task->idg+0] = (union Data*)integer1;
21      context->task->data[context->task->idg+1] = (union Data*)integer2;
22
23      // set Output Data Gear
24      context->task->data[context->task->odg+0] = (union Data*)integer3;
25
26      // add taskList Element
27      struct Element* element;
28      element = &ALLOCATE(context, Element)->Element;
29      element->data = (union Data*)context->task;
30      element->next = context->taskList;
31      context->taskList = element;
32
33      // set TaskManager->spawns parameter
34      Gearef(context, TaskManager)->taskList = context->taskList;
35      Gearef(context, TaskManager)->next1 = C_code2;
36      goto meta(context, C_code2);
37  }
38
39  // code gear
40  __code add(Integer *integer1, Integer *integer2, next(Integer *output,
        ...)) {
41      ....
42  }
```

ソースコード 4.11 に par goto 文による記述例を示す。この記述はスクリプトにより、Task で実行される Code Gear の Input/Output の数を解析し、ソースコード 4.10 に変換される。

ソースコード 4.11: par goto による並列実行

```
1  __code code1(Integer *integer1, Integer * integer2, Integer *output) {
2      par goto add(integer1, integer2, output, __exit);
3      goto code2();
4  }
```

par goto の引数には Input/Output Data Gear と 実行後に継続する Code Gear を渡す。par goto で生成された Task は __exit に継続することで終了する。Gears OS の Task は Output Data Gear を生成した時点で終了するので、par goto では直接 __exit に継続するのではなく、Output Data Gear への書き出し処理に継続される。これにより Code Gear と Data Gear の依存関係をノーマルレベルで記述できるようになる。この par goto 文は 通常のプログラミングの関数呼び出しのように扱える。

4.7 Task(Context) 間の同期処理

Gears OS では複数の Task(Context) から同じ Output Data Gear を修正する場合がある。その際に適切な同期処理を行わずそのまま実行すると Output Data Gear の整合性が取れない場合がある。

そこで 複数の Task 間の同期処理を行うために Semaphore の実装を行った。Semaphore の Interface を ソースコード 4.12 に示す。

ソースコード 4.12: Semaphore Interface

```
1 typedef struct Semaphore<Impl>{
2         union Data* semaphore;
3         __code next(...);
4
5         // method
6         __code p(Impl* semaphore, __code next(...));
7         __code v(Impl* semaphore, __code next(...));
8 } Semaphore;
```

Semaphore はある資源に対してアクセスできるスレッドの数を制限するものであり、P 命令 と V 命令がある。P 命令は資源の消費に相当し、V 命令が資源の開放に相当する命令である。P 命令を行う際、資源がなければ V 命令で開放されるまでそのスレッドは処理を停止する。

Gears OS の Context はスレッドに相当するため、 Gears OS 上で Semaphore を実装することは Context の停止を実装する必要がある。Gears OS の Semaphore は Context の停止を停止用の待ち Queue を使って行う。

図 4.7 に資源が 1 つの Semaphore に 2 つの Context が P 命令を実行しているシーケンス図を示す。

図 4.7 の処理の流れを以下に示す。

1. Context1 が Semaphore に対して P 命令を実行する。Semaphore には資源が残っているので資源を消費する

2. Context2 が Semaphore に対して P 命令を実行する。この時、Semaphore に資源が残っていないので Context を Seamphore が持っている待ち Queue に追加する。その後は Context2 を取得していた Worker に処理を移譲する。

3. Context1 が Semaphore に対して V 命令を実行する。Semaphore の資源を開放しつつ、 待ち Queue に Context があるかの確認を行う。Context があった場合 1 つ Context を 待ち Queue から取得し、TaskManager へ Context の実行を行う命令を実効する。

TaskManager に送られた Context は Worker で取得される。取得された Context は停止した時の状態を記録しているため、停止した Code Gear である P 命令を再び実行する。

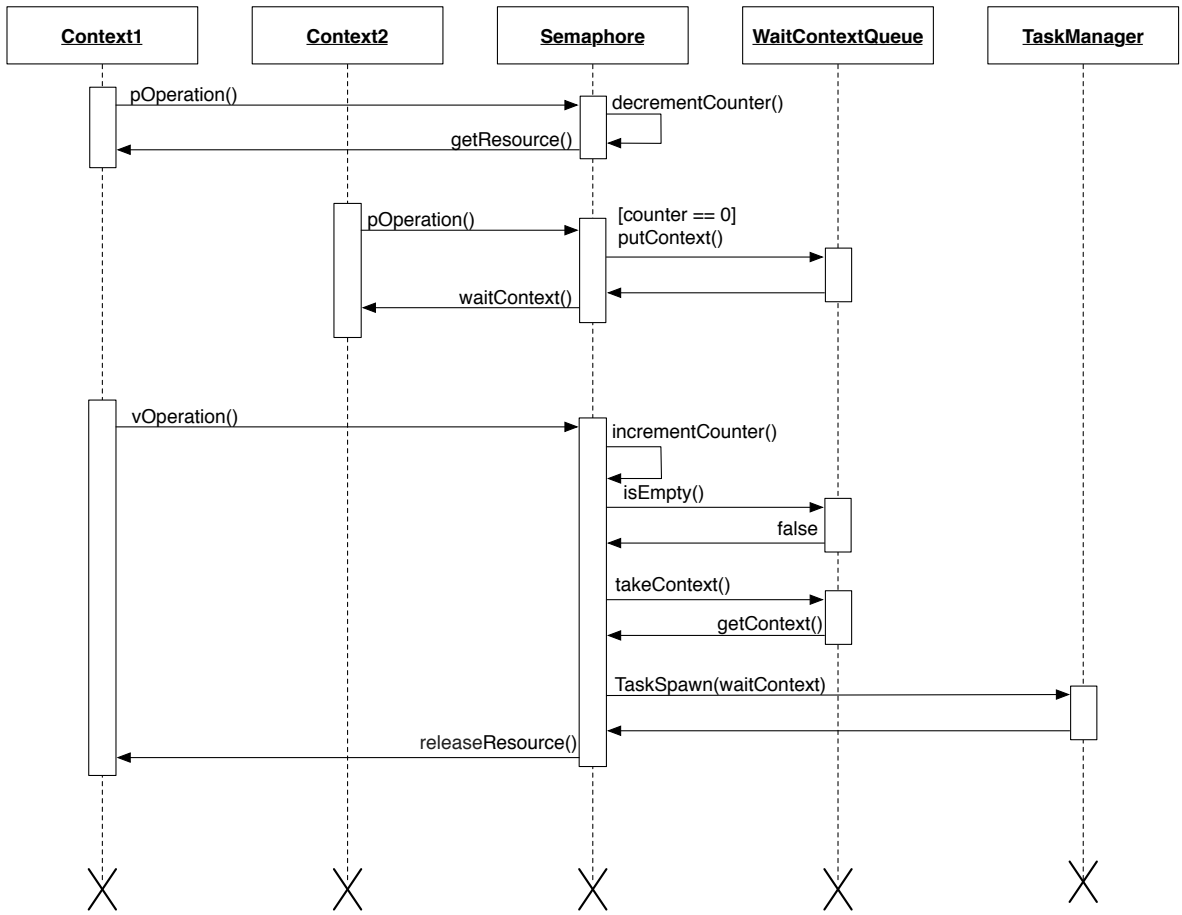

図 4.7: Gears OS 上 Semaphore

4.8 データ並列

並列プログラミングを行う際、並列化の方式としてタスク並列とデータ並列の 2 つがある。Gears OS の並列処理は Task(Context) を使用したタスク並列により実現されている。

タスク並列は処理をタスクに分割し、各タスク間に依存関係のないものを集め、それを並列化する。Gears OS では依存関係を Input/Output Data Gear から解析を行い、依存関係が解決された Code Gear から実行される。一方でデータ並列は処理対象のデータが十分な数のサブデータへ分割することが可能で、各サブデータに行う処理が同じ場合に有効な並列処理手法である。このデータ並列は GPGPU と相性が良く、GPU 環境でも実行できる Gears OS でもサポートを行った。

Gears OS でデータ並列実行を行う場合、ソースコード 4.13 のように par goto 文の引数にデータ並列用の構文として iterate を入れて実行する。iterate は複数の数値を引数とし、数値の値がデータの分割数、数値の個数が次元数になる。

ソースコード 4.13: par goto によるデータ並列

```
1 __code code1() {
2     par goto printIterator(input, output, iterate(2), __exit);
3     goto code2();
4 }
```

Gears OS は データ並列用の par goto を実行した場合、 データ並列用に Task が生成される。この Task には ソースコード 4.14 の Iterator Interface を実装した Data Gear を持たせる。このデータ並列用の Task は Input Data Gear が揃うまでは通常の Task 同等として扱う。依存関係が解決され、実行可能な Task になった際に、 Iterator Interface の exec を呼ぶ。exec では par goto で渡された次元、数値分 Task を コピーし、インデックスを割り当てる処理を行う。この index は コピーされた Task の Input Data Gear として扱われ、Code Gear 内では通常の Data Gear として利用される。図 4.8 は 1 次元で数値 4 を渡した場合の Task 実行を示している。コピーされた Task は通常の Task と同じように TaskManager を通して Worker に送信される。

ソースコード 4.14: Iterator Interface の定義

```
1  typedef struct Iterator<Impl>{
2          union Data* iterator;
3          struct Context* task;
4          int numGPU;
5          __code next(...);
6          __code whenWait(...);
7
8          // method
9          __code exec(Impl* iterator, struct Context* task, int numGPU,
       __code next(...));
10         __code barrier(Impl* iterator, struct Context* task, __code next
       (...), __code whenWait(...));
```

```
11 } Iterator;
```

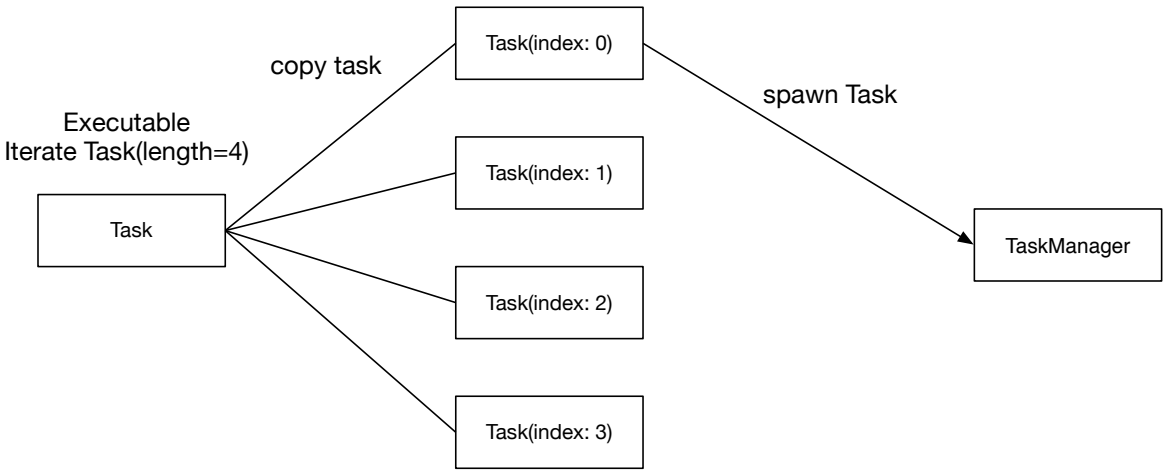

図 4.8: 1 次元、数値 4 のデータ並列用 Task の実行

通常の Task であれば、実行後に Output Data Gear を書き出す処理に入るが、 データ並列用の Task はコピーされた全ての Task 実行後に Output Data Gear の書き出しを行う。その判断と処理を行うのが Iterator Interface の barrier である。barrier は コピーされた Task 実行後に呼ばれ、Output Data Gear が書き出せる状態なら書き出し処理を行う Code Gear に継続し、書き出せる状態でないなら Worker に操作を移譲する Code Gear に継続する。

第5章　CUDA への対応

Gears OS では GPU での実行もサポートする [15]。また、CPU、GPU の実行環境の切り替えは Meta Code Gear、つまり stub Code Gear で切り替えを行う。

本章では、 Gears OS での CUDA 実行のサポートについて説明する。

5.1　CUDA

CUDA[17] とは NVIDA 社が提供する GPU コンピューティング向けの総合開発環境である。

CUDA は演算用プロセッサ (GPU) を Device、制御用デバイス側 (CPU) を Host として定義する。また、Device 上で実行するプログラムのことを kernel と呼ぶ。

また、CUDA には CUDA Runtime API と CUDA Driver API の 2 種類存在する。Driver API は Runtime API に比べて管理すべきリソースが多いが、 Runtime API より柔軟な処理を行うことが出来る。Gears OS では Driver API を用いて GPU 実行の実装を行う。

CUDA では処理の最小の単位を thread と定義しており、それをまとめたものを block と呼ぶ。block と thread は それぞれ 3 次元まで展開することが出来る。図 5.1 に thread、block を 2 次元で展開した例を示す。

kernel を起動すると、各 thread に対して block ID と thread ID が付与される。この ID は blockIdx、threadIdx といった組み込み変数で取得できる。これらの変数は 3 次元のベクター型になっており、blockIdx.x とすると x 座標の block ID が取得でき、 threadIdx.x とすると x 座標の thread Id を取得できる。また、block 内の thread 数は blockDim という組み込み変数で取得でき、これも 3 次元のベクター型になっている。CUDA では これらの組み込み変数から thread が対応するデータを割り出し、データ並列の処理を行う。

5.2　CUDAWorker

CUDAWorker は TaskManager から送信される CUDA 用の Task を取得し、実行を行う。

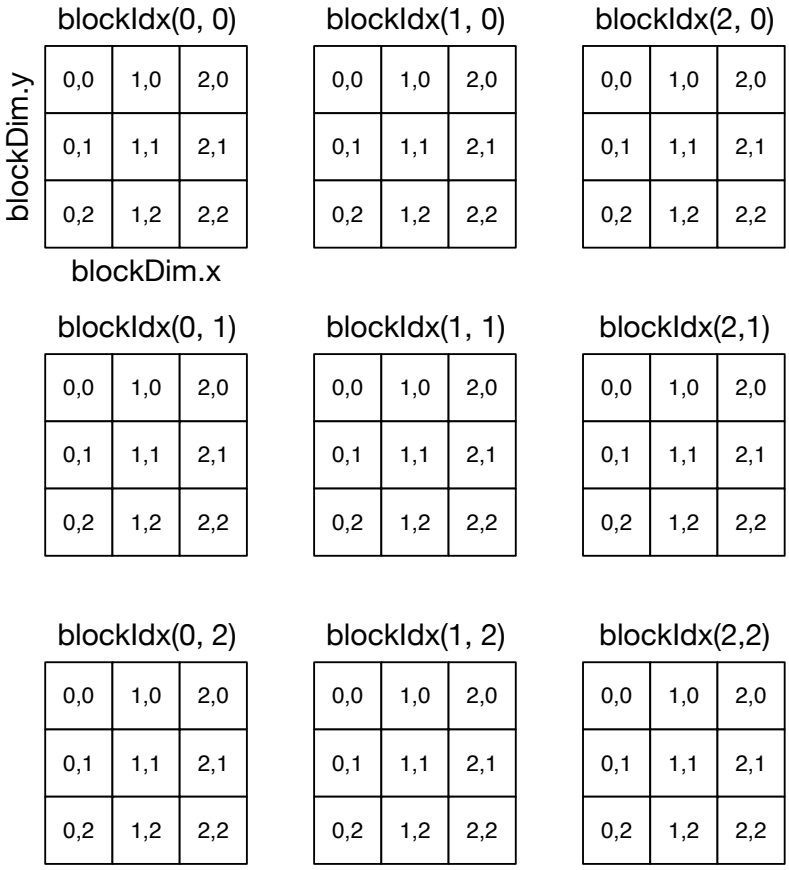

図 5.1: block サイズ (3,3)、 thread サイズ (3,3) に展開

CUDAWorker は CPUWorker と同じく初期化の際にスレッドが生成される。生成されたスレッドは CUDA ライブラリ初期化や後述する CUDAExectuor の生成を行う。

データ並列用の Task は CUDAWorker に送信する際は Task のコピーを行わず送信する。受け取ったデータ並列用の Task は Code Gear のメタレベルで kernel の実行を行う。

5.3 CUDAExectuor

CUDAExectuor は ソースコード 5.1 に示す ExecutorInterface を実装しており、 Host から Device へのデータの送信 (read)、kernel の実行 (exec)、Device から Host への データの書き出しを行う (write)。

ソースコード 5.1: executor Inteface

```
typedef struct Executor<Impl>{
    union Data* Executor;
    struct Context* task;
    __code next(...);
    // method
    __code read(Impl* executor, struct Context* task, __code next(...));
    __code exec(Impl* executor, struct Context* task, __code next(...));
    __code write(Impl* executor, struct Context* task, __code next(...));
}
```

Gears OS では データは Data Gear で表現される。つまり、Host、Device 間でデータのやり取りを行うということは Data Gear を GPU のデータ領域に沿った形に適用する必要がある。Host から Device へデータを送信する際、 CUDA では cuMemAlloc 関数を使用してサイズを指定し、Device 側のデータ領域を確保する。全ての Data Gear には Meta Data Gear として Data Gear のサイズを持っており、基本的にはこのサイズでデータ領域を取ればよい。しかし、Data Gear によっては内部に更にポインタで Data Gear を持っている場合がある。このような Data Gear は Data Gear の実際のサイズではなく、ポインタのサイズで計算されてしまうため、そのままでは Device 用のデータ領域を確保することができない。

この問題を解決するために、CUDABuffer という CUDA データ送信用の Data Gear を用意した. CUDABuffer には Data Gear の内部にポインタを持たない Data Gear まで展開した Input/Output Data Gear を格納される。Data Gear を CUDABuffer に格納する処理は CUDAExectuor では行わず、実行される Task の stub Code Gear で行われる。CUDABuffer に格納されている Data Gear のサイズを参照し、cuMemAlloc 関数で Device のデータ領域を確保する。

Host、Device、CUDABuffer 間の関係を図 5.2 に示す。

Host から Device にデータをコピーするには cuMemcpyHtoD 関数を使用して行う。この際に Host で指定するデータは CUDABuffer に格納されている Data Gear となる。

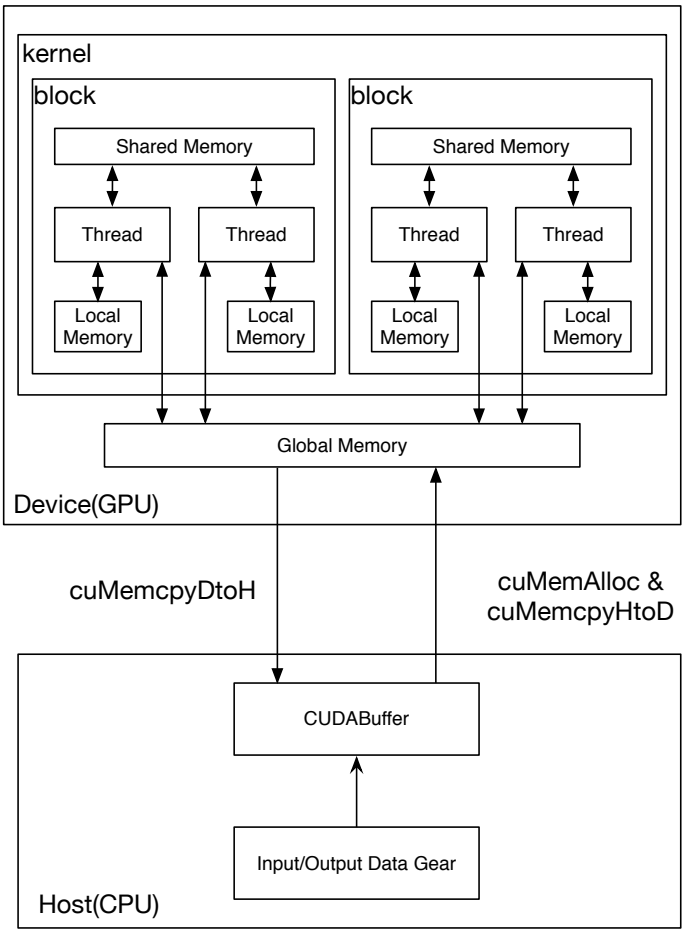

図 5.2: Host、 Device 間のデータの関係

kernel の実行後、結果を Device から Host にコピーする際は cuMemcpyDtoH 関数で行われる。Host のコピーされたデータは Output Data Gear も含んでいるため、 コピー後は Output Data Gear への書き出す処理に継続する。

kernel の実行は cuLaunchKernel 関数で行われる。cuLaunchKernel 関数には引数として各次元の block サイズ、thread サイズ、kernel への引数等を渡す。Gears OS ではデータ並列 Task の際は Iterator Interface を持っており、 そこで指定した長さ、次元数に応じて cuLaunchKernel の引数を決定する。

5.4 stub Code Gear による kernel の実行

Gears OS では stub Code Gear で CUDA による実行の切り替える。

stub Code Gear での切り替えの際は CUDABuffer への Data の格納、実行される kernel の読み込みを行う。実際に GPU で実行されるプログラムは ソースコード 5.2 のように記述する。

ソースコード 5.2: 配列の要素を二倍にする例題

```
1 extern "C" {
2     __global__ void twice(int* array) {
3         array[i+(blockIdx.x*blockDim.x+threadIdx.x)*prefix] = array[i+(
    blockIdx.x*blockDim.x+threadIdx.x)*prefix]*2;
4     }
5 }
```

stub Code Gear は通常はその stub に対応した Code Gear に継続するが、 CUDA で実行する際は CUDAExectuor の Code Gear に継続する。

第6章　Gears OS の評価

6.1　実験環境

今回 Twice、 BitonicSort をそれぞれ CPU、GPU 環境で Gears OS の測定を行う。使用する実験環境を表 6.1、 GPU 環境を表 6.2 に示す。

Model	Dell PowerEdgeR630
OS	CentOS 7.4.1708
Memory	768GB
CPU	2 x 18-Core Intel Xeon 2.30GHz

表 6.1: 実行環境

GPU	GeForce GTX 1070
Cores	1920
Clock Speed	1683MHz
Memory Size	8GB GDDR5
Memory Bandwidth	256GB/s

表 6.2: GPU 環境

6.2　Twice

Twice は与えられた整数配列のすべての要素を 2 倍にする例題である。

Twice の Task は Gears OS のデータ並列で実行される。CPU の場合は配列ある程度の範囲に分割して Task を生成する。これは要素毎に Task を生成するとその分の Context を生成するために時間を取ってしまうからである。

Twice は並列実行の依存関係もなく、データ並列での実行に適した課題である。そのため、 通信時間を考慮しなければ CPU よりコア数が多い GPU が有利となる。

要素数 2^{27} のデータに対する Twice の実行結果を 表 6.3、図 6.1 に示す。CPU 実行の際は 2^{27} のデータを 64 個の Task に分割して並列実行を行っている。GPU では 1 次元の block 数を 2^{15}、 block 内の thread 数を 2^{10} で kernel の実行を行った。ここでの “GPU“ は CPU、GPU 間のデータの通信時間も含めた時間、 “GPU(kernel only)“ は kernel のみの実行時間である。

Processor	Time(ms)
1 CPU	1181.215
2 CPUs	627.914
4 CPUs	324.059
8 CPUs	159.932
16 CPUs	85.518
32 CPUs	43.496
GPU	127.018
GPU(kernel only)	6.018

表 6.3: 2^{27} のデータに対する Twice

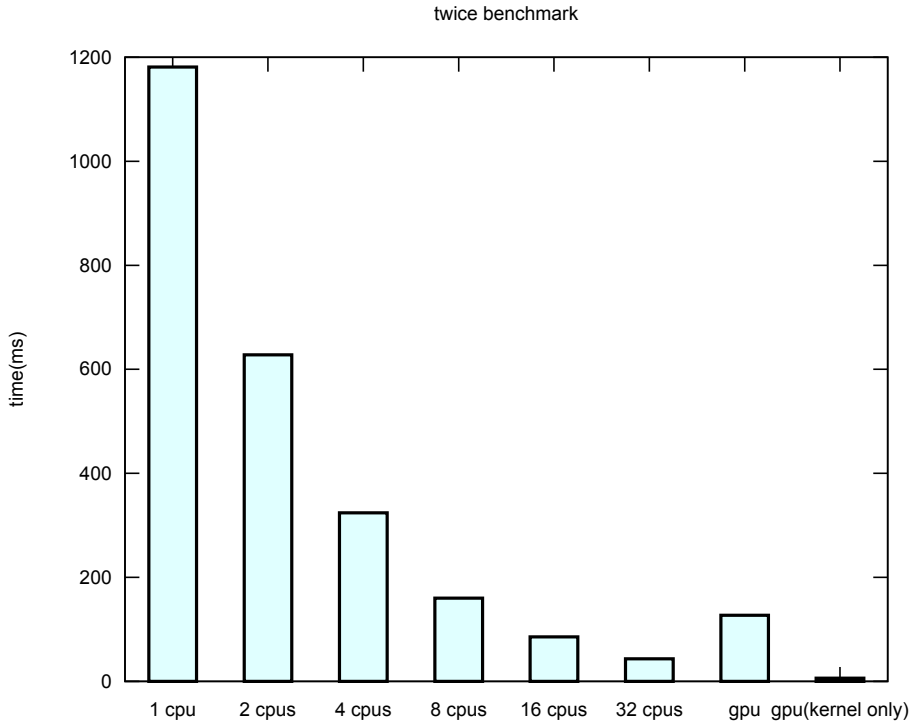

図 6.1: 2^{27} のデータに対する Twice

1 CPU と 32 CPU では 約 27.1 倍の速度向上が見られた。ある程度の台数効果があると考えられる。

GPU での実行は kernel のみの実行時間は 32CPU に比べて 約 7.2 倍の実行向上が見られた。しかし、通信時間を含めると 16CPU より遅い結果となってしまった。CPU、GPU の通信時間かオーバーヘッドになっている事がわかる。

6.3 BitonicSort

BitonicSort は並列処理向けのソートアルゴリズムである。代表的なソートアルゴリズムである Quick Sort も並列処理 を行うことが可能であるが、QuickSort では ソートの過程で並列度が変動するため、台数効果が出づらい。一方で Bitonic Sort は最初から最後まで並列度が変わらずに並列処理を行う。図 6.2 は要素数 8 のデータに対する BitonicSort のソートネットワークである。

BitonicSort はステージ毎に決まった 2 点間の要素の入れ替えを並列に実行することによってソートを行う。Gears OS ではこのステージ毎に Output Data Gear を書き出し、

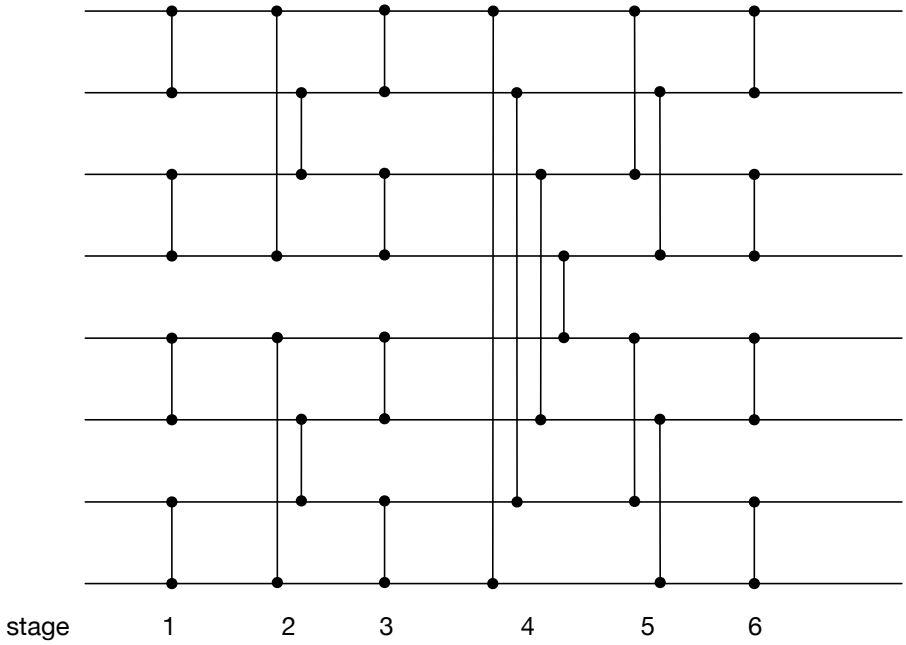

図 6.2: 要素数 8 の BitonicNetwork

次のステージの Code Gear の Input Data Gear として記述することで BitonicSort を実現する。

要素数 2^{24} のデータに対する BitonicSort の実行結果を 表 6.4、図 6.3 に示す。こちらも Twice と同じく CPU 実行の際は 2^{24} のデータを 64 個の Task に分割して並列実行を行っている。つまり生成される Task は 64 * ステージ数 となる。GPU では 1 次元の block 数を 2^{14}、 block 内の thread 数を 2^{10} で kernel の実行を行った。

Processor	Time(s)
1 CPU	41.416
2 CPUs	23.340
4 CPUs	11.952
8 CPUs	6.320
16 CPUs	3.336
32 CPUs	1.872
GPU	5.420
GPU(kernel only)	0.163

表 6.4: 2^{24} のデータに対する BitonicSort

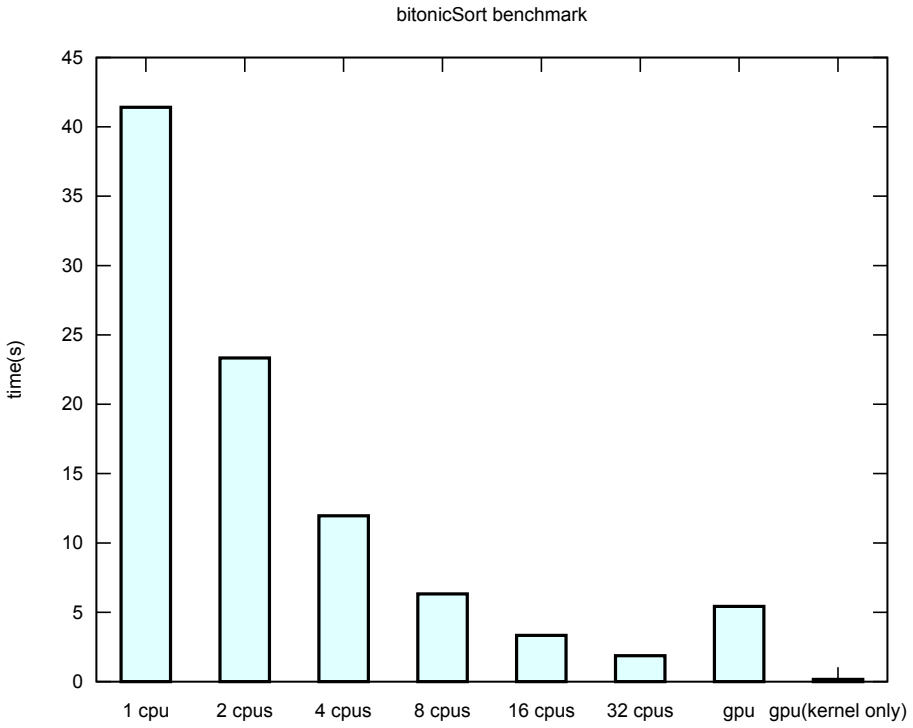

図 6.3: 2^{24} のデータに対する BitonicSort

1 CPU と 32 CPU で 約 22.12 倍 の速度向上が見られた。GPU では通信時間を含めると 8 CPU の約 1.16 倍となり、 kernel のみの実行では 32 CPU の約 11.48 倍となった。現在の Gears OS の CUDA 実装では、 Output Data Gear を書き出す際に一度 GPU から CPU へ kernel の実行結果の書き出しを行っており、その処理の時間で差が出たと考えられる。GPU で実行される Task 同士の依存関係の解決の際は CuDevicePtr などの GPU のメモリへのポインタを渡し、CPU でデータが必要になったときに初めて GPU から CPU へデータの通信を行うメタ計算の実装が必要となる。

6.4 OpenMP との比較

OpenMP[18] は C、 C++ のプログラムにアノテーションを付けることで並列化を行う。アノテーションを ソースコード 6.1 のように for 文の前につけることで、ループの並列化を行う。

ソースコード 6.1: OpenMP での Twice

```
1 #pragma omp parallel for
2 for(int i = 0; i < length; i++) {
3     a[i] = a[i] * 2;
4 }
```

OpenMP は既存のコードにアノテーションを付けるだけで並列化を行えるため、変更が少なくて済む。しかし、 ループのみの並列化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ってしまう。OpenMP はループの並列化ではなくブロック単位での並列実行もサポートしているが、アノテーションの記述が増えてしまう。また、 OpenMP はコードとデータを厳密に分離していないため、データの待ち合わせ処理をバリア等のアノテーションで記述する。

Gears OS では Input Data Gear が揃った Code Gear は並列に実行されるため、プログラム全体の並列度を高めることが出来る。また 並列処理のコードとデータの依存関係を par goto 文で簡潔に記述することが出来る。

Gears OS と OpenMP で実装した Twice の実行結果の比較を図 6.4 に示す。実行環境は 表 6.1、 2^{27} のデータに対して行い、Gears OS 側は配列を 64 個の Task に分割し、OpenMP は for 文を static スケジュールで並列実行した。static スケジュールはループの回数をプロセッサーの数で分割し、並列実行を行う openMP のスケジュール方法である。

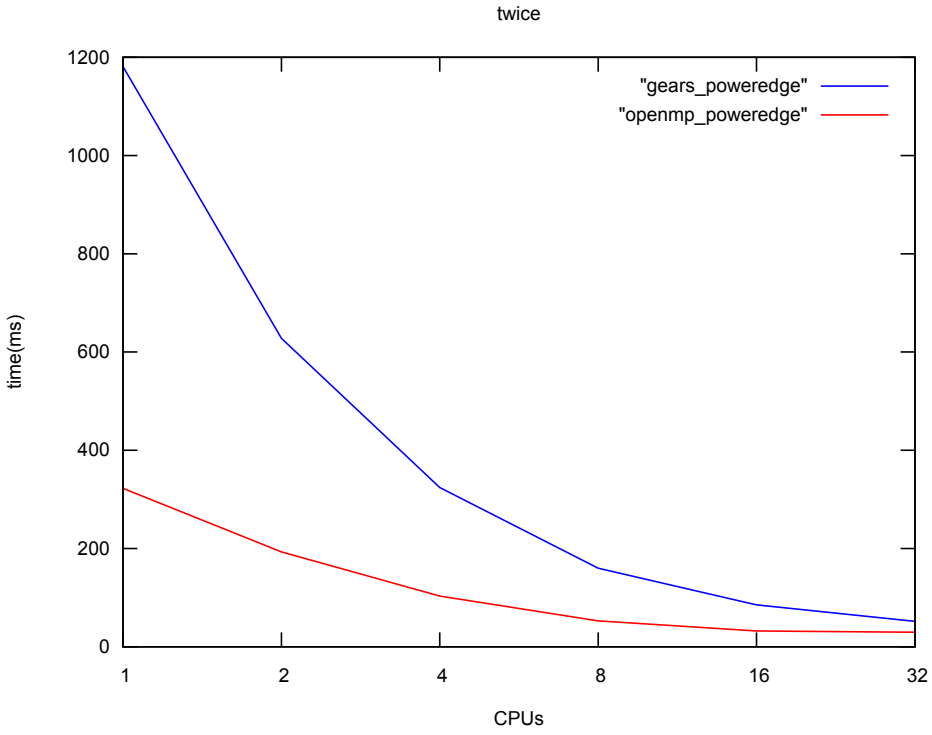

図 6.4: vs OpenMP

OpenMP は 1CPU と 32CPU で約 10.8 倍の速度向上がみられた。一方 Gears OS では約 27.1 倍の速度向上のため、台数効果が高くなっている。しかし、Gears OS は 1CPU での実行時間が OpenMP に比べて大幅に遅くなっている。

6.5　Go 言語との比較

Go 言語 は Google 社が開発しているプログラミング言語である。Go 言語による Twice の実装例をソースコード 6.2 に示す。

ソースコード 6.2: Go 言語での Twice

```
1 func main() {
2     c := make(chan []int)
3     for i :=0; i < *split; i++ {
4         // call goroutine
5         go twice(list, prefix, i, c);
6     }
7
8     for i :=0; i < *split; i++ {
```

```
9           // join twice routins
10          <- c
11      }
12  }
13
14  func twice(list []int, prefix int, index int, c chan []int) {
15      for i := 0; i < prefix; i++ {
16          list[prefix*index+i] = list[prefix*index+i] * 2;
17      }
18      c <- list
19  }
```

Go は並列実行を “go function(argv)” のような構文で行う。この並列実行を goroutine と呼ぶ。

Go は goroutin 間のデータ送受信をチャネルというデータ構造で行う。チャネルによるデータの送受信は “<-” を使って行われる。例えばチャネルのデータ構造である channel に対して “channel <- data” とすると、 data を channel に送信を行う。“<- channel” とすると、 channel から送信されたデータを 1 つ取り出す。channel にデータが送信されていない場合は channel にデータが送信されるまで実行をブロックする。Go 言語はチャネルにより、データの送受信が簡潔に書ける。しかし、チャネルは複数の goroutine で参照できるためデータの送信元が推測しづらい。

Gears OS では goroutine は par goto 文とほぼ同等に扱うことが出来る。また、Code Gear は par goto 文で書き出す Output Data Gear を指定して実行するため、Data Gear の書き出し元が推測しやすい。

Go 言語での OpenMP と同様に Twice を実装し Gears OS と比較を行う。こちらも実行環境は 表 6.1、 2^{27} のデータに対して行い、Gears OS Go 言語両方とも配列を 64 個の Task、 goroutine に分割して並列実行を行った。

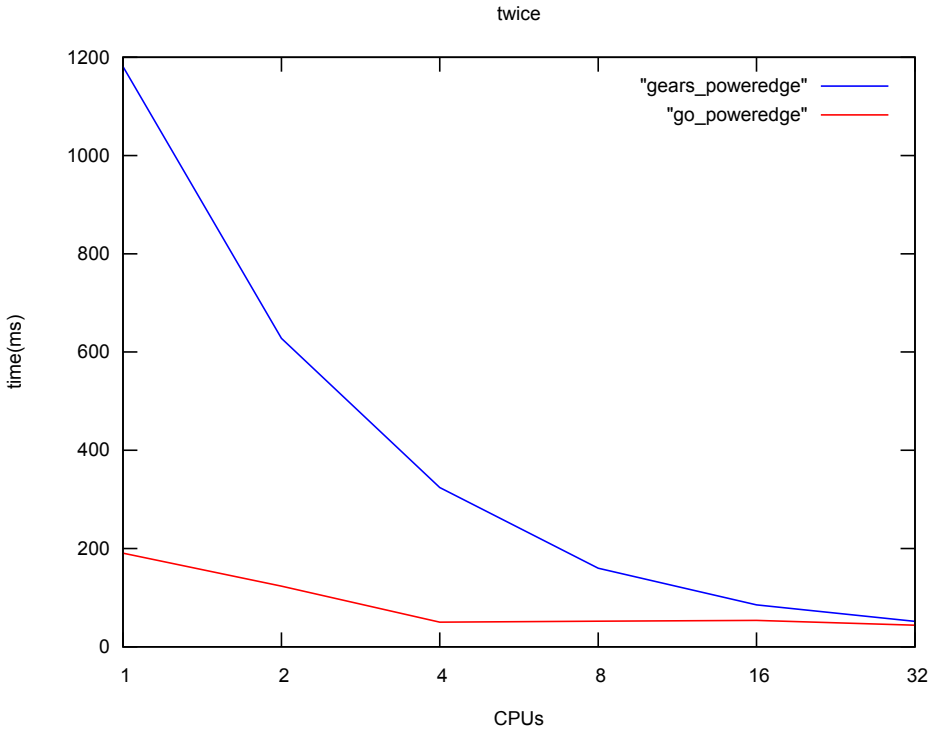

図 6.5: vs Go

Go 言語は 1CPU と 32CPU で約 4.33 倍の速度向上が見られた。こちらも OpenMP と同じく、台数効果自体は Gears OS が高いが、1CPU での実行時間は Go 言語が大幅に速くなっている。

第7章　結論

本研究では Gears OS の並列実行機構の実装を行った。Gears OS は処理を Code Gear、データを Data Gear を用いて処理を実行する。Code Gear と Input/Output Data Gear の組を Task とし、並列実行を行う。

Gears OS のプログラミングは Interface という 一部の Code Gear と Data Gear の集合を表現する Meta Data Gear を用いて行われる。この Interface は stub Code Gear という全ての Code Gear に対応する Meta Code Gear で Data Gear を参照する。

Gears OS の Task は Context という全ての Code/Data Gear を参照できる Meta Data Gear が対応する。並列処理を行う際は Context を生成し、 Code Gear と Input/Output Data Gear を Context に設定して TaskManager 経由で各 Worker の SynchronizedQueue に送信される。Context の設定はメタレベルの記述になるため、ノーマルレベルでは par goto 文という CbC の goto 文に近い記述で並列処理を行える。この par goto は通常のプログラミングの関数呼び出しのように扱える。

Gears OS の GPU 対応はアーキテクチャ用の Worker と Executor、 Buffer を用意することでアーキテクチャ毎に合わせた実行を行える。本研究では CUDA 実装として CUDAWorker、CUDAExecutor、CUDABuffer の実装を行い、実行確認を行った。また、CPU、 GPU の実行の切り替えは並列実行される Code Gear の stub Code Gear で継続先を切り替えることでメタレベルで記述することが可能となった。

Twice と BitonicSort の例題の測定結果では 1CPU と 32CPU で Twice では約 27.1 倍、BitonicSort では 約 22.12 倍の速度向上が見られた。また、GPU 実行の測定も行い、kernel のみの実行時間では 32 CPU より Twice では約 7.2 倍、BitonicSort では約 11.48 倍の速度向上がみられ、GPU の性能を活かすことができた。

7.1　今後の課題

今後の課題として、Gears OS の並列処理の信頼性の保証、 チューニングを行う。

Gears OS では証明とモデル検査をメタレベルで実行することで信頼性を保証する。

証明は CbC のプログラムを証明支援系の Agda に対応させて証明を行う。現在は Gears OS の Interface 部分の動作の証明を行っており、Stack や Tree の動作の証明を行ってい

る。Gears OS の並列処理の信頼性を証明するには Synchronized Queue の証明を行う必要がある。

モデル検査では CbC で記述されたモデル検査器である akasha [19] を使用して行う。モデル検査の方針としては、Code Gear の実行を擬似並列で実行し、全ての組合せを列挙する方法で行う。

Go、OpenMP との比較から、 Gears OS が 1CPU での動作が遅いということがわかった。Gears OS は par goto 文を使用することで Context を生成し、並列処理を行う。しかし、Context はメモリ空間の確保や使用する全ての Code/Data Gear を設定する必要があり、生成にある程度の時間がかかってしまう。そこで、 par goto のコンパイルタイミングで実行する Code Gear のフローをモデル検査で解析し、処理が軽い場合は Context を生成せずに、関数呼び出しを行う等の最適化を行うといったチューニングが必要である。

今回の CUDA 実装では Output Data Gear を書き出す際に一度 GPU から CPU にデータの送信する必要があった。しかし、CPU、 GPU 間のデータの通信はコストが高いことが例題の結果からわかった。GPU にあるデータは CPU 側ではポインタで持つことが出来る。そこで Meta Data Gear に Data Gear が CPU、GPU のどこで所持されているかを持たせる。GPU にある Data Gear が CPU で必要になったときに初めてデータの通信を行うことで、最低限のデータ通信で処理を実行できる。

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。

2018 年 3 月
伊波立樹

参考文献

[1] 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.

[2] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 1–16, Berkeley, CA, USA, 2016. USENIX Association.

[3] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 18–37, New York, NY, USA, 2015. ACM.

[4] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. ACM.

[5] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 99–110, New York, NY, USA, 2010. ACM.

[6] 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.

[7] 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.

[8] 宮城光希, 河野真治. Code gear と data gear を持つ gears os の設計. 第 59 回プログラミング・シンポジウム予稿集, 第 2018 巻, pp. 197–206, jan 2018.

[9] J. Meyerson. The go programming language. *IEEE Software*, Vol. 31, No. 5, pp. 104–104, Sept 2014.

[10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] Tokumori Kaito and Kono Shinji. Implementing continuation based language in llvm and clang. *LOLA 2015, Kyoto*, July 2015.

[12] 大城信康, 河野真治. 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.

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

[14] 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. 第 133 回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), May 2015.

[15] 東恩納琢偉, 伊波立樹, 河野真治. Gears os における並列処理. 第 140 回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), May 2017.

[16] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pp. 267–275, New York, NY, USA, 1996. ACM.

[17] Cuda zone — nvidia developer. `https://developer.nvidia.com/cuda-zone`. Accessed: 2018/02/05(Mon).

[18] Openmp: Simple, portable, scalable smp programming. `http://www.openmp.org,`. Accessed: 2018/02/05(Mon).

[19] 比嘉健太, 河野真治. Verification method of programs using continuation based c. 情報処理学会論文誌プログラミング（PRO）, Vol. 10, No. 2, pp. 5–5, feb 2017.

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, May, 2016