修士 (工学) 学位論文

Master's Thesis of Engineering

Code Segment と Data Segment を持つ Gears OS の設計

Design of Gears OS with Code and Data Segment

2016 年 3 月

March 2016

小久保 翔平

Shohei KOKUBO

琉球大学

大学院理工学研究科

情報工学専攻

Infomation 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)　　高良　富夫

印

(副　査)　　長田　智和

印

(副　査)　　河野　真治

要 旨

本研究では Code/Data Segment によって構成される Gears OS の設計を行った。Gears OS は Cerium の開発を通して得られた知見を元に開発する。Cerium はオブジェクト指向言語である C++ を用いて開発した並列プログラミングフレームワークである。Cell, マルチコア CPU, GPU を用いた並列実行をサポートしている。並列処理の単位として Task を記述し、Task に他の Task との依存関係を設定することで並列実行を実現する。データは汎用ポインタで Task に渡されるため、データの依存関係を保証できない。Cerium を用いて並列処理を行うことでプロセッサの性能を十分に引き出すことができるしかし、データの正しさや依存関係を保証できていないので信頼性が低い。また、Cerium を用いて並列処理を記述するのは慣れが必要で難しい。オブジェクト指向自体も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性が低く並列処理と相性が悪い。今回設計した Gears OS は本研究室で開発している Continuation base C(CbC) を用いて実装する。CbC は Code Segment と Data Segment という単位でプログラムを記述する。Code Segment は並列処理の単位として利用でき、Data Segment はデータのそのもので型情報を持つ。Gears OS ではメタレベルの処理をサポートする。Gears OS の各スレッドは Context と呼ばれるメタレベルの Data Segment を持つ。Context には接続可能な Code/Data Segment のリスト、Data Segment の型情報、独立したメモリ空間、スレッドの情報等が格納されている。Code Segment の遷移には goto を用いる。ノーマルレベルの Code Segment からノーマルレベルの Code Segment に遷移するときメタレベルの Code Segment を間に挟む。CbC は Code/Data Segment, メタレベルの処理、並列処理を記述できる言語である。Gears OS 自体が Gears OS を用いて並列処理を記述する際の指針となるように実装する。本論文では Gears OS のプロトタイプを実装する。また、並列処理に必要な機能である Red-Black Tree, Synchronized Queue も実装し、簡単な並列処理の例題を用いて評価する。

Abstract

In this thesis we have designed Gears OS based on Code/Data Segment. It is a reimplementation of Cerium TaskManager. Cerium is parallel programming framework developed by C++, which suports Cell Broadband Engine, Shared Memory Multi CPU and GPU. Parallel computing of Cerium describes by Tasks, which have Task-Dependency. Data structures are passed to a Task as input/output parameters but the structures have no type constraints nor Data-Dependencies. Cerium gives good performances, but without type correctness and Data-Dependency, its programming very difficult and unreliable. Cerium is implement in C++, which is not suitable in parallel machine because objects in C++ has no referential transparency. Newly designed Gears OS is written in Continuation base C(CbC). CbC has Code Segments which are suitable as Tasks, it also has Data Segments with Type-Signature. Gears OS has Meta Computation supports. For each thread of Gears OS there is a Meta Data Segment called Context. The Context contains a set of Code Segments and Data Segments in the threads, it also has a Meta Data Segment such as Data Segment Type-Signature, Memory Allocation and Threads information. A Code Segment passes the control to another Code Segment using a goto statement. A Meta Computation can be inserted between the Code Segments. Using CbC, Code Segments, Data Segments and Meta Computation, parallel computation can be described in reliable style. Since Gears OS itself is written CbC it can be a guide to Gears OS Parallel Programming. We show a prototype implementation of Gears OS and show some examples including Red-Black Tree, Synchronized Queue, Simple parallel computation example.

目 次

図 目 次

表 目 次

第1章　並列環境下における プログラミング

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。これはアムダールの法則により、並列化できない部分が並列化による性能向上を制限することから言える。つまり、プログラムを並列処理に適した形で記述するためのフレームワークが必要になる。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 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ので型情報を失う。Task 側で正しく明示的に型変換する必要があり、誤った型変換を行うとデータ構造自体を破壊する可能性がある。型システムによって検査することも出来ず、型に基づく一連の不具合が常に付きまとう。

Gears OS は Code Segment と Data Segment によって構成される。Code Segment は処理の単位、Data Segment はデータの単位となる。Gears OS では Code/Data Segment 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 Segment 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 Segment と実行に必要な Input Data Segment, 出力される Output Data Segment

の組で表現される。Input/Output Data Segment によって依存関係が決定し、それに沿って並列実行する。本論文では基本的な機能として 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。

第2章　並列プログラミングフレームワーク Cerium

Gears OS は Cerium を開発して得られた知見を元に設計を行う。

Cerium は並列実行に Cell, マルチコア CPU, GPU を用いることができる並列プログラミングフレームワークである。高い並列度を保ったまま複雑な並列処理を記述可能で、プロセッサの性能を十分に引き出すことができる。しかし、いくつかの問題点が存在する。Gears OS はそれらの問題点を解決することを目的とする。

本章では Cerium の設計・実装について説明し、評価を行う。

2.1　Cerium の概要

Cerium は、TaskManager, SceneGraph, Rendering Engine の 3 つの要素から構成される。Cell 用のゲームフレームワークとして開発されたが、現在では Multi-Core CPU, GPU も計算資源として利用可能な汎用計算フレームワークとなっている。

2.2　TaskManager

TaskManager は、Task と呼ばれる分割されたプログラムを管理する。サブルーチンまたは関数が Task の単位となる。TaskManager が提供する API を表:2.1 に示す。

TaskManager は ActiveTaskList と WaitTaskList の 2 種類の Queue を持つ。依存関係を解決する必要がある Task は WaitTaskList に入れられる。TaskManger によって依存関係が解決されると ActiveTaskList に移され、実行可能な状態となる。実行可能な状態となった Task は set_cpu で指定された Device に対応した Scheduler に転送し実行される。図:2.1 は Cerium が Task を生成/実行する場合のクラスの構成である。

create_task	Task の生成
allocate	環境のアライメントに考慮した allocator
set_inData	Task への入力データのアドレスを追加
set_outData	Task からのデータ出力先アドレスを追加
set_param	Task のパラメータ (32 bits)
wait_for	Task の依存関係を設定
set_cpu	Task を実行する Device の設定
spawn	Task を Queue に登録
iterate	データ並列で実行する Task として Queue に登録

表 2.1: TaskManager API

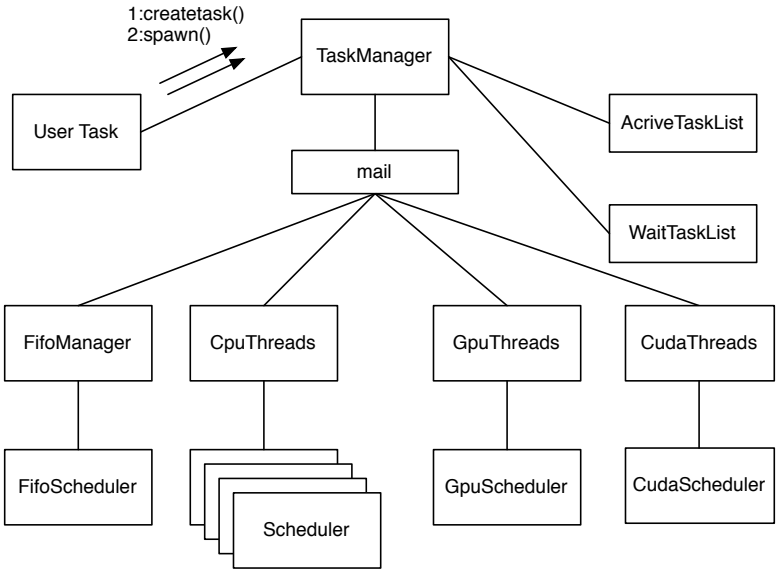

図 2.1: TaskManager

2.3　Cerium における Task

Task は TaskManager の API を利用して生成する。生成された Task には以下の要素を設定することができる。

- input data
  set_inData を用いて設定する Task が実行する処理に必要なデータの入力元となるアドレス。関数を呼び出す際の引数に相当する。汎用ポインタ (void* 型) なので Task 側で適切なキャストを行う必要がある。

- output data
  set_outData を用いて設定する Task が処理したデータの出力先となるアドレス。関数の戻り値に相当する。

- parameter
  set_param を用いて設定するデータの処理に必要な実数値 (index 等)。

- cpu type
  set_cpu を用いて設定する Task が実行される Device の組み合わせ。Cell, Multi-Core CPU, GPU またはこれらの組み合わせを指定することができる。

- dependency
  wait_for を用いて設定する他の Task との依存関係。依存関係が解決された Task は実行可能な状態となる。

ソースコード:2.1 に Task を生成する例題を示す。

input data として int 型の配列を受け取り、各要素を 2 倍にして output data に格納する twice という例題である。CPU を用いてデータ並列で実行する Task を生成している。set_cpu で GPU を指定することで GPU を用いて実行される。

```
1  void
2  twice_init(TaskManager *manager, int* data, int length)
3  {
4      /**
5       * Create Task
6       * create_task(Task ID);
7       */
8      HTask* twice = manager->create_task(TWICE_TASK);
9
10     /**
11      * Set of Device
12      * set_cpu(CPU or GPU)
13      */
14     twice->set_cpu(SPE_ANY);
15
16     /**
17      * Set of Input Data
18      * set_inData(index, address of input data, size of input data);
19      */
20     twice->set_inData(0, data, sizeof(int)*length);
21
22     /**
23      * Set of OutPut area
24      * set_outData(index, address of output area, size of output area);
25      */
26     twice->set_outData(0, data, sizeof(int)*length);
27
28     /**
29      * Enqueue Task
30      * iterate(Number of Tasks)
31      */
```

```
32     twice->iterate(length);
33 }
```

ソースコード 2.1: Task の生成

CPU 上で実行される Task, GPU 上で実行される kernel はソースコード:2.2, ソースコード 2.3 の通りになる。

Task には実行時に必要なデータが格納されている SchedTask, 設定した Input/Output Data が格納されている Buffer が渡される。

```
1  static int
2  twice(SchedTask *s,void *rbuf, void *wbuf)
3  {
4      /**
5       * Get Input Data
6       * get_input(input data buffer, index)
7       */
8      int* input = (int*)s->get_input(rbuf, 0);
9
10     /**
11      * Get Output Data
12      * get_output(output data buffer, index)
13      */
14     int* output = (int*)s->get_output(wbuf, 0);
15
16     /**
17      * Get index(x, y, z)
18      * SchedTask member
19      * x : SchedTask->x
20      * y : SchedTask->y
21      * z : SchedTask->z
22      */
23     long i = s->x;
24
25     output[i] = input[i]*2;
26
27     return 0;
28 }
```

ソースコード 2.2: 実行される Task

```
1  __global__ void
2  twice(int* input, int* output)
3  {
4      /**
5       * Get index(x, y, z)
6       * kernel built-in variables
7       * x : blockIdx.x * blockDim.x + threadIdx.x
8       * y : blockIdx.y * blockDim.y + threadIdx.y
9       * z : blockIdx.z * blockDim.z + threadIdx.z
10      */
11     long i = blockIdx.x * blockDim.x + threadIdx.x;
12
13     output[i] = input[i]*2;
14
15     return 0;
16 }
```

ソースコード 2.3: 実行される kernel

2.4　Task のパイプライン実行

Cell(図:2.2) や GPU(図:2.3) のように異なるメモリ空間を持つ Device を計算資源として利用するにはデータの転送が必要になる。このデータ転送がボトルネックとなり、並列度が低下してしまう。転送処理をオーバーラップし、並列度を維持するために Cerium では Task のパイプライン実行をサポートしている。

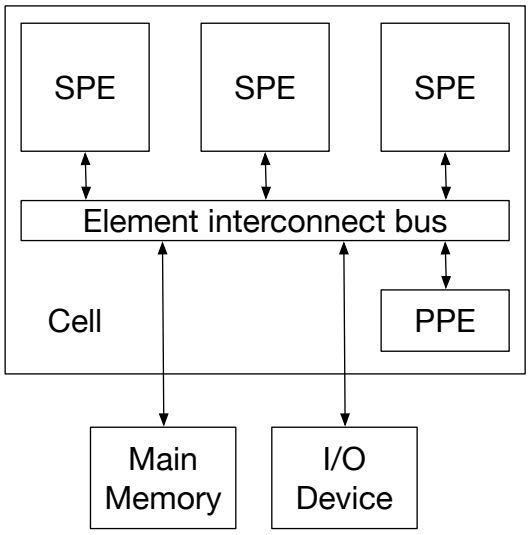

図 2.2: Cell Architecture

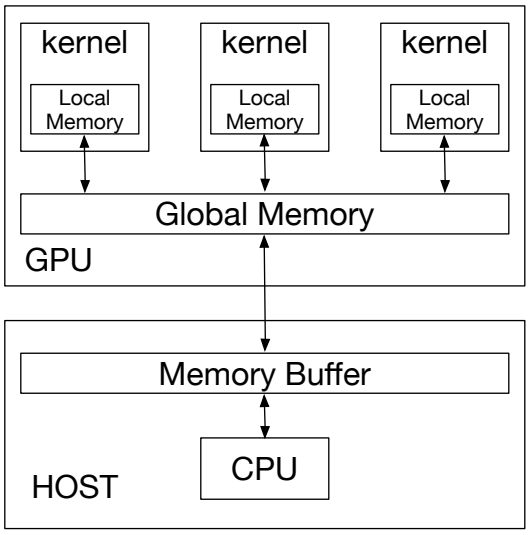

図 2.3: GPU Architecture

TaskManager である程度の Task をまとめた TaskList を生成し、実行する Device に対応した Scheduler に転送する。受け取った TaskList に沿ってパイプラインを組み Task を実行していく。TaskList でまとめられている Task は依存関係が解決されているので自由にパイプラインを組むことが可能である。実行完了は TaskList 毎ではなく、Task 毎に通知される。図:2.4 は TaskList を受け取り、Task をパイプラインで処理していく様子である。

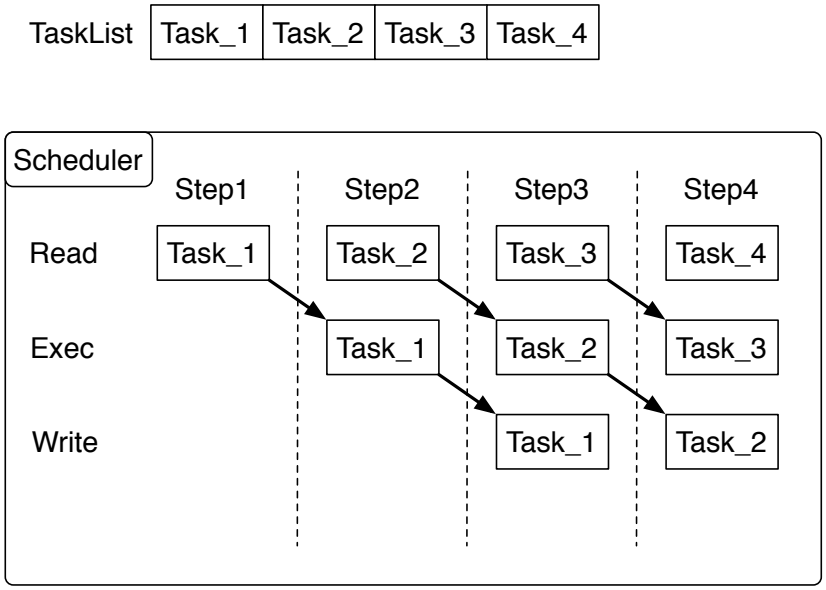

図 2.4: Scheduler

2.5　マルチコアへの対応

Cell には MailBox という機能がある。MailBox を用いることで双方向のデータの受け渡しが可能になる。FIFO キュー構造を持つ MailBox に対応させる形で Synchronized Queue 用いて Multi Core CPU 用の TaskManager に MailBox を移植した。Synchronized Queue は Queue を操作しているスレッドが常に 1 つになるようにバイナリセマフォを用いて制御する。

Cell では MailBox 以外に DMA 転送を使用してデータの受け渡しすることができる。DMA 転送は CPU を介さずに周辺装置とメモリ間でデータ転送を行う方式である。Cerium では DMA 転送を用いて Cell で実行することが可能である。Multi Core CPU 上で実行する場合、メモリ空間を共有しているので DMA 転送を行なっている部分をポインタ渡しを行うように修正し、直接アクセスさせることでデータ転送の速度の向上が見込める。

2.6　データ並列による実行

並列処理の方法としてタスク並列とデータ並列の 2 つがある。

タスク並列は Task 毎にデータを準備し、管理スレッドが個別に生成した Task を CPU に割り当てることで並列処理する方法である。異なる処理を同時に実行することができるというメリットがあるが、データ群の各要素に対して同じ処理をしたいときタスク並列では要素毎に同じ処理をする Task を生成する必要があり、ほとんど同一な大量の Task によってメモリを圧迫する場合がある。また、大量な Task の生成自体が大きなオーバーヘッドになる。

データ並列はあるデータ群を大量な Task で共有し、Task 実行時に処理範囲を計算し、その範囲にのみ処理を行うことで並列処理する方法である。実行スレッドで Task の生成・実行が行われるので、メモリの圧迫や Task 生成によるオーバーヘッドを抑えられる。並列化部分が全て同じ処理である場合、データ並列による実行のほうがタスク並列より有効である。

いままで Cerium における並列処理はタスク並列だったが、データ並列のよる実行もサポートした。

データ並列による実行では処理範囲を決定するための情報として index が必要になる。CPU による実行では SchedTask を参照 (ソースコード:2.2 23 行目)、GPU による実行では組み込み変数を参照 (ソースコード:2.3 11 行目) することで index を取得することができる。

データの長さが 10、CPU の数が 4 でデータ並列による実行をした場合の index の割当は表 2.2 の通りになる。

stage	CPU0	CPU1	CPU2	CPU3
1	0	1	2	3
2	4	5	6	7
3	8	9		

表 2.2: index の割り当て

2.7　GPGPU への対応

GPU の演算資源を Cerium から利用するために OpenCL, CUDA を用いた GpuScheduler, CudaScheduler を実装した。OpenCL, CUDA 単体を用いて GPGPU を行う場合、依存関係を記述する必要があるしかし、Cerium には依存関係を解決する TaskManager があるので GpuScheduler, CudaScheduler は受け取った TaskList を元に GPU を制御して GPGPU を行えばよい。

GPU はメモリ空間が異なる (図 2.3) のでデータ転送が大きなオーバーヘッドになる。なので、kernel 実行中にデータ転送を行うなどしてデータ転送をオーバーラップする必要がある。CUDA で GPU を制御するには同期命令を使う方法と非同期命令を使う方法があるが、同期命令ではデータ転送をオーバーラップすることが出来ないので非同期命令を利用して GPU を制御する。非同期命令は Stream に発行することで利用することができる。Stream に発行された命令は発行された順序で実行される。非同期命令と Stream を利用してデータ転送をオーバラップするには複数の Stream を準備して、Host から Device への転送・kernel の実行・Device から Host への転送を 1 セットとして各 Stream に発行することで実現できる。同期命令を使う場合と非同期命令を使う場合の実行の様子は図:2.5 の通りである。

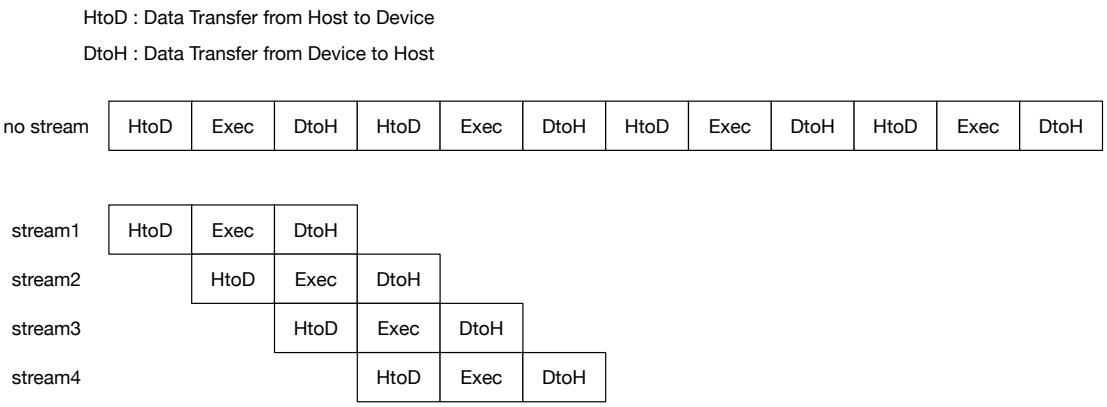

図 2.5: Overlap Data Transfer

2.8 Cerium の評価

Bitonic Sort, Word Count, Fast Fourier Transform(FFT) の 3 つの例題を用いて Cerium を評価する。

測定環境は表:2.3、測定に用いる GPU は表 2.4 の通りである。

Model	MacPro Mid 2010
OS	Mac OS X 10.10.
Memory	16GB
CPU	2 x 6-Core Intel Xeon 2.66GHz
GPU	NVIDIA Quadro K5000

表 2.3: 測定環境

Cores	1536
Clock Speed	706MHz
Memory Size	4GB GDDR5
Memory Bandwidth	173 GB/s

表 2.4: Quadro K5000

2.8.1 Bitonic Sort

Bitonic Sort は並列処理に向いた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 も並列処理することが、Quick Sort はソートの過程で並列度が変動するので自明な台数効果が出づらい。一方、Bitonic Sort は最初から最後まで並列度が変わらずに並列処理による恩恵を得やすい。図:2.6 は要素数 8 のデータに対する Bitonic Sort のソーティングネットワークである。

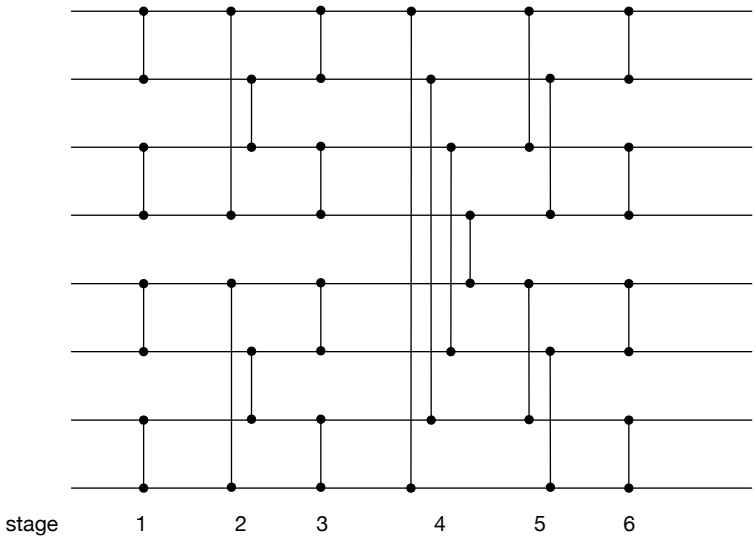

図 2.6: Sorting Network : bitonic sort

Bitonic Sort の並列処理に用いられる Task は 2 点間のの比較・交換を行うだけの小さい処理なので、1 コア当たりのクロック数よりもコアの数が結果に与える影響が大きいと考えられる。よって、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。

Cerium を用いて Bitonic Sort を実装し、要素数 2^{20} のデータに対してコア数・プロセッサの種類を変更して測定を行なった結果は表 2.5、図 2.7 の通りである。

Processor	Time(ms)
1 CPU	6143
2 CPUs	4633
4 CPUs	2557
8 CPUs	1630
12 CPUs	1318
GPU	155

表 2.5: 要素数 2^{20} に対するソート

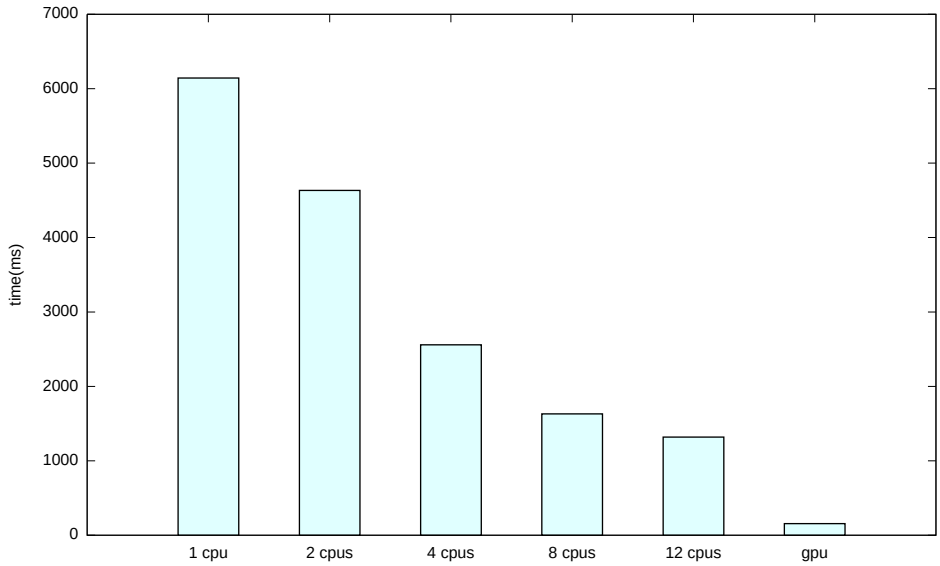

図 2.7: 要素数 2^{20} に対するソート

1 CPU と 12 CPU では約 4.6 倍の速度向上が見られた。これは Task の粒度が小さいため 1 コア当たりのクロック数の高さが活かしづらく、並列化によるオーバーヘッドが結果に影響を与えたと考えられる。CPU を用いた並列化には Task の粒度をある程度大きくし 1 コア当たりの仕事量を増やして CPU のクロック数の高さを活かすことが重要であることがわかる。

12 CPU と GPU では約 8.5 倍の速度向上が見られた。GPU の特徴であるコア数の多さによって CPU より高い並列度を発揮した結果だと考えられる。GPU の場合はその超並列性を活かすため Task を細かく分割することが重要であることがわかる。

測定結果から CPU と GPU で並列化の方法を変更する必要があることがわかった。Cerium を用いてヘテロジニアス環境で並列実行する場合、混在しているプロセッサの特徴に合わせたスケジューリングを行い並列実行するように Scheduler を改良する必要がある。

次に要素数も変更して測定を行なった。結果は図:2.8、図:2.9 の通りである。

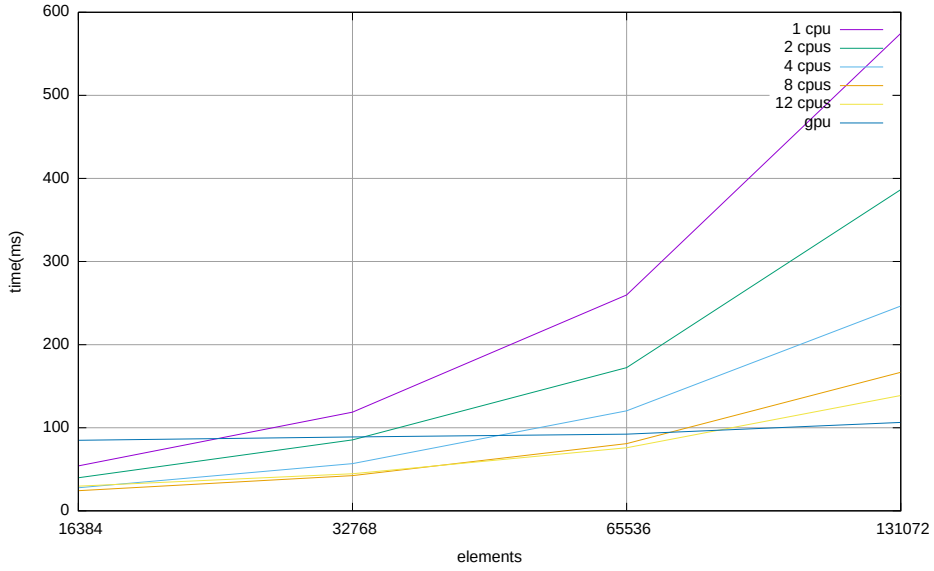

図 2.8: Bitonic Sort(from 2^{14} to 2^{17})

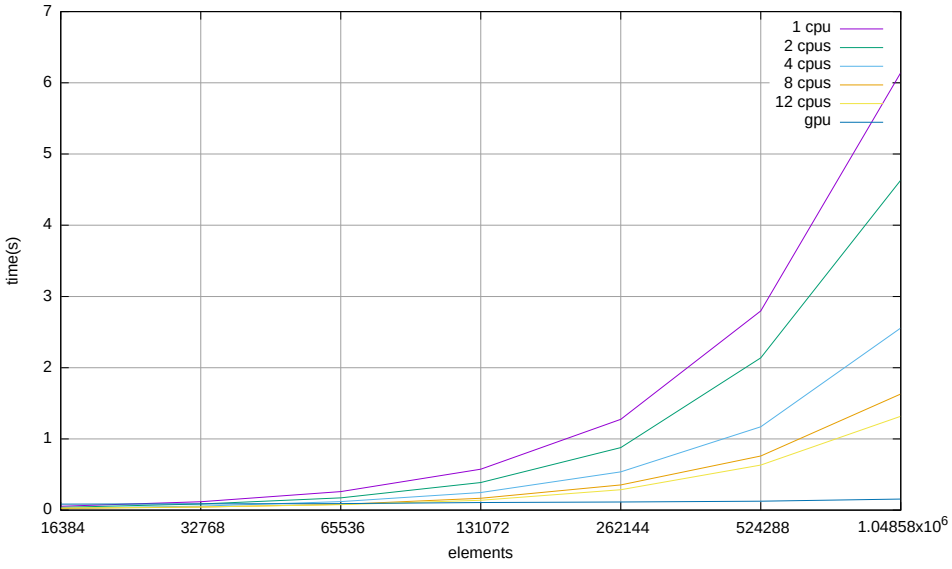

図 2.9: Bitonic Sort(from 2^{14} to 2^{20})

GPGPU では通信時間を考慮する必要がある。図:2.8 を見ると要素数 2^{14} のソートでは GPU が一番遅い。これはソート処理の時間より通信時間が大きいことが原因であると考えられる。通信時間を含めた処理時間が GPU が CPU を上回るのは要素数 2^{17} を超えてからである。

2.8.2 Word Count

並列処理を行う際に Task を大量に生成する場合がある。一度に大量の Task を生成してしまうと Task がメモリを圧迫して処理速度が著しく低下する。改善策としては Task の生成と実行を平行して行えばよい。Cerium では Task を生成する Task を記述することが可能なので Task の生成と実行を平行して行うことができる。

Word Count を並列処理する場合、与えられたテキストを分割して、分割されたデータごとに並列処理を行う。分割したデータの数だけ Task が必要なのでテキストサイズによっては一度に Task を生成するとメモリを圧迫する可能性がある。よって、Task を生成する Task が必要になる。Word Count の処理の流れは図 2.10 の通りである。

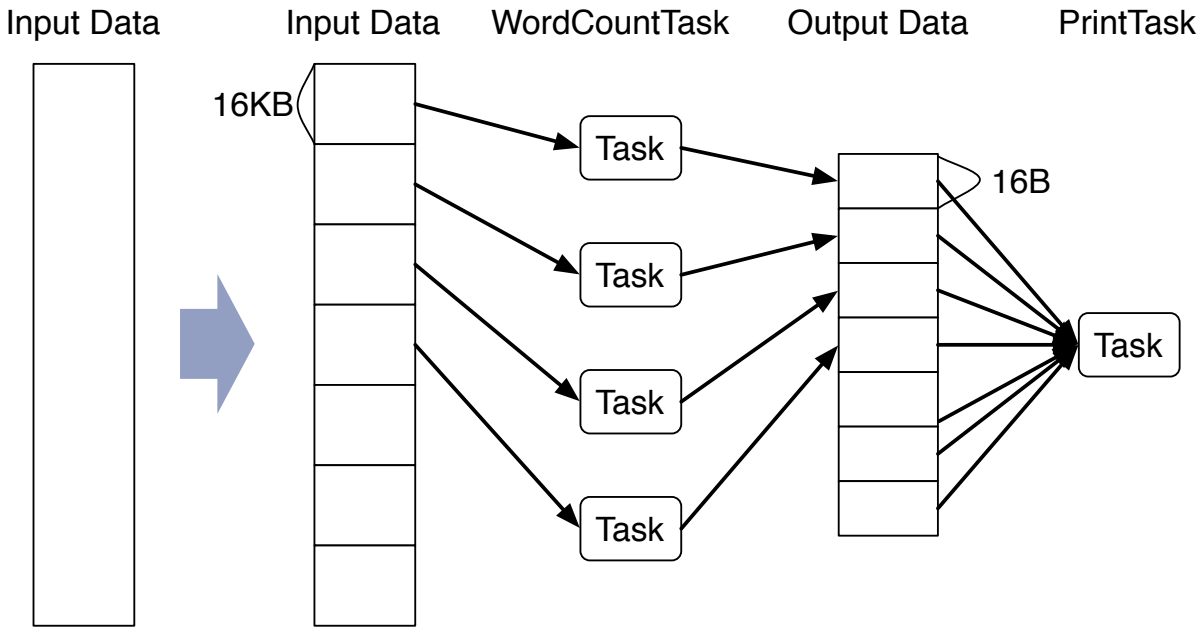

図 2.10: Word Count の流れ

Cerium が複雑な並列処理を記述可能でその上、高い並列度を保てること示すため Cerium 上に Word Count を実装し、100MB のテキストデータに対して測定を行なった。結果は表:2.6, 図:2.11 の通りである。

Processor	Time(ms)
1 CPU	716
2 CPUs	373
4 CPUs	197
8 CPUs	105
12 CPUs	87
GPU	9899
GPU(Data Parallel)	514

表 2.6: 100MB のテキストデータに対する WordCount

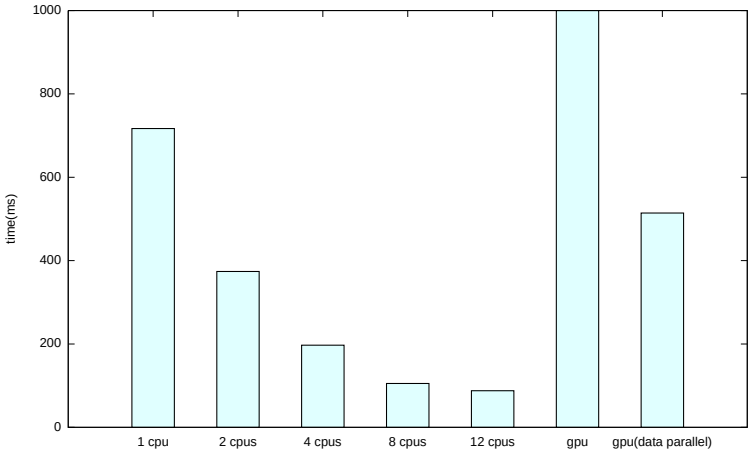

図 2.11: 100MB のテキストデータに対する WordCount

1 CPU と 12 CPU では約 8.2 倍の速度向上が見られた。複雑な並列処理でも高い並列度が保てていることがわかる。

GPU を用いたタスク並列による実行は実用に耐えない速度である。これはタスク並列による実行では小さなデータを十数回 GPU に転送する必要があるからで、GPU で高速に処理するためにはデータ転送を如何にして抑えるかが重要かわかる。一方、GPU を用いたデータ並列による実行速度は 1 CPU の約 1.4 倍となった。元々 WordCount は GPU に不向きな例題ではあるが、データ並列による実行ではデータ転送の回数を抑えることができるので GPU でもある程度の速度を出せることがわかる。

2.8.3　FFT

FFT は信号処理や画像処理、大規模シミュレーションに至るまで幅広い分野で活用されている計算である。バタフライ演算などの計算の性質上、大量の演算資源を持つ GPU と相性が良い。Cerium に実装した GPU 実行機構の評価を行うために適切な例題であると考えられる。

Cerium 上に FFT を実装し、測定を行なった結果は表:2.7, 図:2.12 の通りである。測定には 1MB の画像データを用いた。

Processor	Time(ms)
1 CPU	1958
2 CPUs	1174
4 CPUs	711
8 CPUs	451
12 CPUs	373
GPU	418

表 2.7: 1MB の画像データに対する FFT

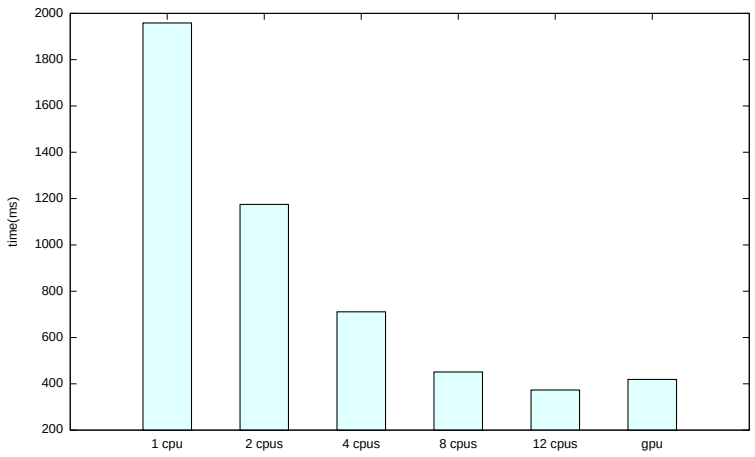

図 2.12: 1MB の画像データに対する FFT

1 CPU に対して 12 CPU では約 5.2 倍、GPU では約 4.7 倍の速度向上が見られる。ある程度の速度向上が見られたが、CPU に劣る結果となった。データ転送の最適化が十分に成されていない可能性があるので、GPU の実行機構を見直す必要がある。

2.9 Cerium の問題点

- Task 間の依存関係
  Cerium では Task 間の依存関係を記述することで並列処理を実現する。しかし、本来 Task はデータが揃えば実行可能になるものである。Task 間の依存関係だけでは待っている Task が不正な処理を行いデータがおかしくなっても Task の終了は通知され、そのまま処理が続行されてしまう。その場合、どこでデータがおかしくなったのか特定するのは難しくデバッグに多くの時間が取られてしまう。

- データの型情報
  Cerium の Task は汎用ポインタでデータを受け取るので型の情報がない。型の情報がないので Task を実行するまで正しい型かどうか判断することが出来ない。不正な型でも強制的に型変換され実行されるのでデータの構造を破壊する可能性がある。型システムによってプログラムの正しさを保証することも出来ず、バグが入り込む原因になる。

- Allocator
  Cerium の Allocator は Thread 間で共有されている。共有されているので、ある Thread がメモリを確保しようとすると他の Thread は終了を待つ必要があるその間メモリを確保することができないので処理が止まり、なにもしない時間が生まれてしまう。これが並列度の低下に繋がり、処理速度が落ちる原因になる。

- オブジェクト指向と並列処理
  一般的に同じ入力に対して、同じ出力を返すことが保証される参照透過な処理は並列化を行いやすい。一方、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や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並列処理との相性が悪いと言える。

- Cerium の実装
  Ceirum の実装自体が並列処理を意識したものになっていない。並列プログラミングフレームワークはそれ自体が並列処理を記述するための指針となるように実装されているべきである。

今回設計した Gears OS はこれらの問題を解決することを目的としている。

第3章　CbC

Gears OS の実装には LLVM/Clang 上に実装した CbC を用いる。

CbC は C から for 文、while 文といったループ制御構文や関数呼び出しを取り除き、Code Segment と goto による軽量継続を導入している。図:3.1 は goto による Code Segment の遷移を表したものである。

本章では CbC の特徴である Code Segment と Gears OS に対するサポートについて説明する。

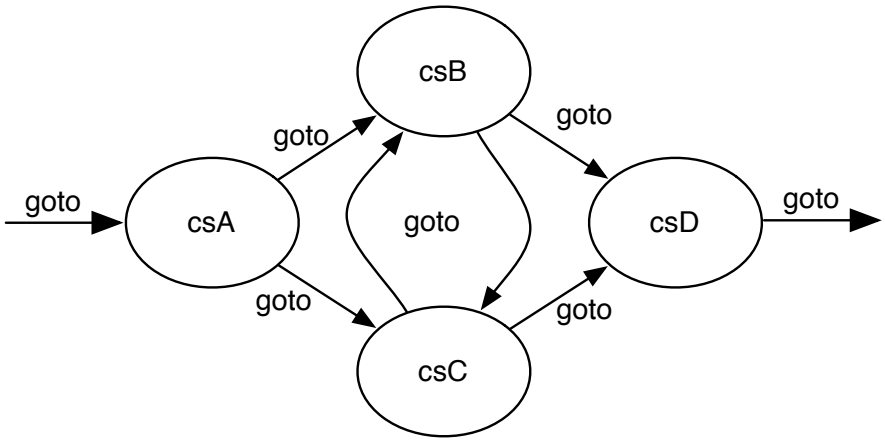

図 3.1: goto による Code Segment 間の継続

3.1　Code Segment

CbC では処理の単位として Code Segment を用いる。Code Segment は CbC における最も基本的な処理単位であり、C の関数と異なり戻り値を持たない。Code Segment の宣言は C の関数の構文と同じように行い、型に __code を用いる。前述した通り、Code Segment は戻り値を持たないので __code はそれが関数ではなく Code Segment であることを示すフラグのようなものである。Code Segment の処理内容の定義も C の関数同様

に行うが、CbC にはループ制御構文が存在しないのでループ処理は自分自身への再帰的な継続を行うことで実現する。

現在の Code Segment から次の Code Segment への処理の移動は goto の後に Code Segment 名と引数を並べて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。C において関数呼び出しを繰り返し行う場合、呼び出された関数の引数の数だけスタックに値が積まれていくが、戻り値を持たない Code Segment ではスタックに値を積んでいく必要が無くスタックは変更されない。このようなスタックに値を積まない継続を軽量継続と呼ぶ。この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なる。

3.2　プロトタイプ宣言の自動化

Code Segment の処理単位は小さく、目的の計算を実現するためには多くの Code Segment を書く必要がある。Code Segment と同じ数だけプロトタイプ宣言を書く必要があり、好ましくない。また、tail call elimination を強制するためにはプロトタイプの宣言を正確に記述することを要求するためプログラマに対する負担が大きい。つまり、プロトタイプ宣言を自動的に行うようにすることで tail call elimnation の条件を安定して満たすことができ、プログラマの負担も減らすことができる。

プロトタイプ宣言の自動化は、パーサーが Code Segment への継続の解析を行なった際にプロトタイプ宣言の有無を確認し、存在しない場合に接続先の Code Segment のプロトタイプ宣言を生成するというようにして行う。

3.3　Gear OS の構文サポート

Gears OS では Context から必要なデータを取り出して処理を行う。しかし、Context を直接扱うのはセキュリティ的に好ましくない。そこで Context から必要なデータを取り出して Code Segment に接続する stub を定義する。stub は接続される Code Segment から推論することが可能である。また、Code Segment の遷移には Meta Code Segment を挟む。Meta Code Segment への接続も省略して記述できるようにする。ノーマルレベルのソースコード:3.1 から実際にコンパイルされるメタレベルを明示したソースコード:3.2 へ変換される。

```
1  // Code Gear
2  __code code1(struct Allocate* allocate) {
3      allocate->size = sizeof(struct Data1);
4
5      goto allocator(allocate, Code2);
6  }
7
8  // Code Gear
9  __code code2(struct Data1* data1) {
10     // processing
11 }
```

ソースコード 3.1: ノーマルレベル

```
1  // Code Gear
2  __code code1(struct Context* context, struct Allocate* allocate) {
3      allocate->size = sizeof(struct Data1);
4      context->next = Code2;
5
6      goto meta(context, Allocator);
7  }
8
9  // Meta Code Gear(stub)
10 __code code1_stub(struct Context* context) {
11     goto code1(context, &context->data[Allocate]->allocate);
12 }
13
14 // Code Gear
15 __code code2(struct Context* context, struct Data1* data1) {
16     // processing
17 }
18
19 // Meta Code Gear(stub)
20 __code code2_stub(struct Context* context) {
21     goto code2(context, &context->data[context->dataNum]->data1);
22 }
```

ソースコード 3.2: メタレベル

第4章　Gears OS

Cerium と Alice の開発を通して得られた知見から並列分散処理には Code の分割だけではなく Data の分割も必要であることがわかった。当研究室で開発している Code Segment を基本的な処理単位とするプログラミング言語 Continuation based C(CbC) を用いて Data Segment を定義し、Gears OS の設計と基本的な機能の実装を行なった。

本章では Gears OS の設計と実装した基本的な機能について説明する。

4.1　Code Gear と Data Gear

Gears OS で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になる。これは OpenCL/CUDA の kernel, Cerium の Task に相当する。Code Gear は任意の数の Data Gear を参照し、処理が完了すると任意の数の Data Gear に書き込む。Code Gear は接続された Data Gear 以外にアクセスできない。Code Gear から次の Code Gear への処理の移動は goto の後に Code Gear の名前と引数を指定することで実現できる。Code Gear は Code Segment そのものである。

Data Gear はデータそのものを表す。int や文字列などの Primitive Data Type を持っている。

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測可能なものにすることが可能になる。

4.2 Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
  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おり、Context を通してアクセスすることができる。メインとなる Context と Worker 用の 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できない。Persistent Data Tree への書き込みのみで相互作用を発生させ目的の処理を達成する。

- TaskQueue
  ActiveTaskQueue と WaitTaskQueue の 2 つの TaskQueue を持つ。先頭と末尾の Element へのポインタを持つ Queue を表す Data Gear である。Element は Task を表す Data Gear へのポインタと次の Element へのポインタを持っている。Compare and Swap(CAS) を使ってアクセスすることでスレッドセーフな Queue として利用することが可能になる。

- TaskManager
  Task には Input Data Gear, Output Data Gear が存在する。Input/Output Data Gear から依存関係を決定し、TaskManager が解決する。依存関係が解決された Task は WaitTaskQueue から ActiveTaskQueue に移される。TaskManager はメインとなる Context を参照する。

- Persistent Data Tree
  非破壊木構造で構成された Lock-free なデータストアである。Red-Black Tree として構成することで最悪な場合の挿入・削除・検索の計算量を保証する。

- Worker
  TaskQueue から Task の取得・実行を行う。Task の処理に必要なデータは Persistent Data Tree から取得する。処理後、必要なデータを Persistent Data Tree に書き出して再び Task の取得・実行を行う。

図:4.1 は Gears OS の構成図である。

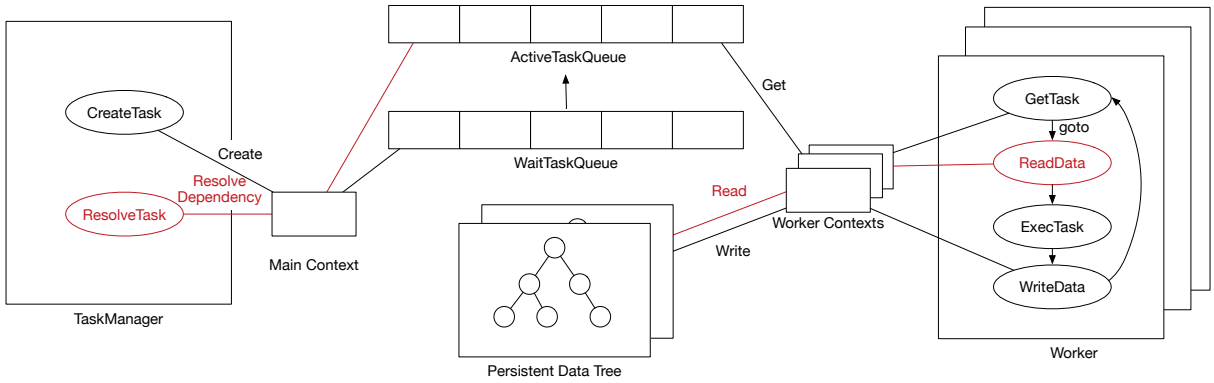

図 4.1: Gears OS

4.3 Allocator

Gears OS では Context の生成時にある程度の大きさのメモリ領域を確保する。Context には確保したメモリ領域を指す情報が格納される。このメモリ領域を利用して Task の実行に必要な Data Gear を生成する。

Context の定義と生成はソースコード:4.1, ソースコード:4.2 の通りである。

```
1  /* Context definition example */
2  #define ALLOCATE_SIZE 1000
3
4  // Code Gear Name
5  enum Code {
6      Code1,
7      Code2,
8      Allocator,
9      Exit,
10 };
11
12 // Unique Data Gear
13 enum UniqueData {
14     Allocate,
15 };
16
17 struct Context {
18     enum Code next;
19     int codeNum;
20     __code (**code) (struct Context*);
21     void* heapStart;
22     void* heap;
23     long heapLimit;
24     int dataNum;
25     union Data **data;
26 };
```

```
27
28 // Data Gear definition
29 union Data {
30     // size: 4 byte
31     struct Data1 {
32         int i;
33     } data1;
34     // size: 5 byte
35     struct Data2 {
36         int i;
37         char c;
38     } data2;
39     // size: 8 byte
40     struct Allocate {
41         long size;
42     } allocate;
43 };
```

ソースコード 4.1: Context

```
1  #include <stdlib.h>
2
3  #include "context.h"
4
5  extern __code code1_stub(struct Context*);
6  extern __code code2_stub(struct Context*);
7  extern __code allocator_stub(struct Context*);
8  extern __code exit_code(struct Context*);
9
10 __code initContext(struct Context* context, int num) {
11     context->heapLimit = sizeof(union Data)*ALLOCATE_SIZE;
12     context->heapStart = malloc(context->heapLimit);
13     context->heap = context->heapStart;
14     context->codeNum = Exit;
15
16     context->code = malloc(sizeof(__code*)*ALLOCATE_SIZE);
17     context->data = malloc(sizeof(union Data*)*ALLOCATE_SIZE);
18
19     context->code[Code1] = code1_stub;
20     context->code[Code2] = code2_stub;
21     context->code[Allocator] = allocator_stub;
22     context->code[Exit] = exit_code;
23
24     context->data[Allocate] = context->heap;
25     context->heap += sizeof(struct Allocate);
26
27     context->dataNum = Allocate;
28 }
```

ソースコード 4.2: initContext

Context はヒープサイズを示す heapLimit, ヒープの初期位置を示す heapStart, ヒープの現在位置を示す heap を持っている。必要な Data Gear のサイズに応じて heap の位置を動かすことで Allocation を実現する。

allocate を行うには allocate に必要な Data Gear に情報を書き込む必要がある。この Data Gear は Context 生成時に生成する必要があり、ソースコード:4.1 14 行目の Allocate がそれに当たる。UniqueData で定義した Data Gear は Context と同時に生成される。

Temporal Data Gear にある Data Gear は基本的には破棄可能なものなので heapLimit を超えたら heap を heapStart の位置に戻し、ヒープ領域を再利用する (図:4.2)。必要な Data Gear は Persistent Data Tree に書き出すことで他の Worker からアクセスすることが可能になる。

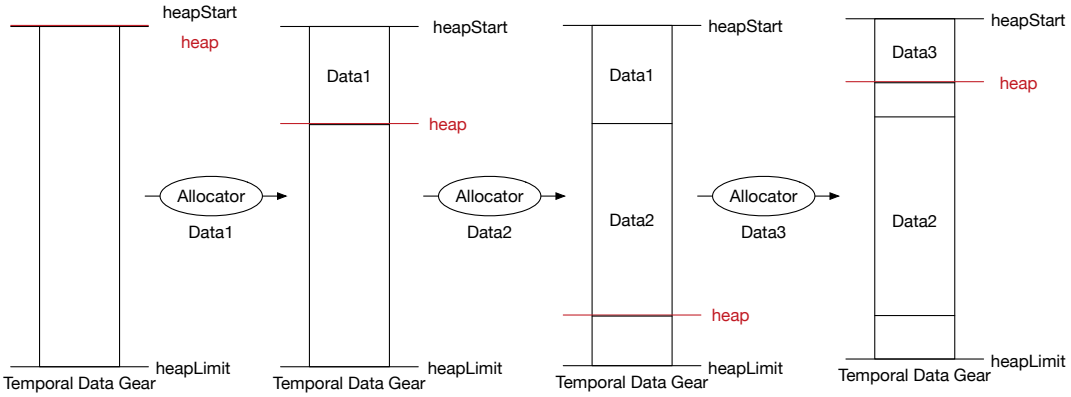

図 4.2: Allocation

実際に allocate を行う Code Gear はソースコード:4.3 の通りである。

Context 生成時に実行可能な Code Gear と名前が対応付けられる。その対応付けられた Code Gear が Context の code に格納される。この code を介して遷移先の Code Gear を決定する。

Code Gear には Context が接続されるが Context を介して Data Gear にアクセスすることはない。stub を介して間接的に必要な Data Gear にアクセスする。

```
1  // Code Gear
2  __code start_code(struct Context* context) {
3      // start processing
4      goto meta(context, context->next);
5  }
6
7  // Meta Code Gear
8  __code meta(struct Context* context, enum Code next) {
9      // meta computation
10     goto (context->code[next])(context);
```

```
11 }
12
13 // Code Gear
14 __code code1(struct Context* context, struct Allocate* allocate) {
15     allocate->size = sizeof(struct Data1);
16     context->next = Code2;
17
18     goto meta(context, Allocator);
19 }
20
21 // Meta Code Gear(stub)
22 __code code1_stub(struct Context* context) {
23     goto code1(context, &context->data[Allocate]->allocate);
24 }
25
26 // Meta Code Gear
27 __code allocator(struct Context* context, struct Allocate* allocate) {
28     context->data[++context->dataNum] = context->heap;
29     context->heap += allocate->size;
30
31     goto meta(context, context->next);
32 }
33
34 // Meta Code Gear(stub)
35 __code allocator_stub(struct Context* context) {
36     goto allocator(context, &context->data[Allocate]->allcate);
37 }
38
39 // Code Gear
40 __code code2(struct Context* context, struct Data1* data1) {
41     // processing
42 }
43
44 // Meta Code Gear(stub)
45 __code code2_stub(struct Context* context) {
46     goto code2(context, &context->data[context->dataNum]->data1);
47 }
```

ソースコード 4.3: allocate

4.4 Synchronized Queue

Gears OS における Synchronized Queue は TaskQueue として利用される。メインとなる Context と Worker 用の Context で共有され、Woker が TaskQueue から Task を取得し実行することで並列処理を実現する。

Gears OS での Queue を Queue を表す Data Gear と Queue の構成要素である Element によって表現する。Queue を表す Data Gear には先頭の 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 には Task を示す task, 次の Element を示す next が格納される。

ソースコード:4.4 は Context の定義 (ソースコード:4.1) に追加する Queue と Element の定義である。

```
1  // Code Gear Name
2  enum Code {
3      PutQueue,
4      GetQueue,
5  };
6
7  // Unique Data Gear
8  enum UniqueData {
9      Queue,
10     Element,
11 };
12
13 // Queue definication
14 union Data {
15     // size: 20 byte
16     struct Queue {
17         struct Element* first;
18         struct Element* last;
19         int count;
20     } queue;
21     // size: 16 byte
22     struct Element {
23         struct Task* task;
24         struct Element* next;
25     } element;
26 }
```

ソースコード 4.4: Context: queue

新たに Queue に対する操作を行う Code Gear の名前を追加し、UniqueData には Queue の情報が入る Queue(ソースコード:4.4 9 行目) と Enqueue に必要な情報を書き込む Element(ソースコード:4.4 10 行目) を定義している。

通常の Enqueue, Dequeue を行う Code Gear はソースコード:4.5 と ソースコード:4.6 の通りである。

```
1  // allocate Element
2  __code putQueue1(struct Context* context, struct Allocate* allocate) {
3      allocate->size = sizeof(struct Element);
4      allocator(context);
5
6      goto meta(context, PutQueue2);
7  }
8
9  // Meta Code Gear(stub)
10 __code putQueue1_stub(struct Context* context) {
11     goto putQueue1(context, &context->data[Allocate]->allocate);
12 }
13
14 // write Element infomation
15 __code putQueue2(struct Context* context, struct Element* new_element, struct
       Element* element, struct Queue* queue) {
16     new_element->task = element->task;
17
18     if (queue->first)
19         goto meta(context, PutQueue3);
20     else
21         goto meta(context, PutQueue4);
```

```
22 }
23
24 // Meta Code Gear(stub)
25 __code putQueue2_stub(struct Context* context) {
26     goto putQueue2(context,
27                    &context->data[context->dataNum]->element,
28                    &context->data[Element]->element,
29                    &context->data[ActiveQueue]->queue);
30 }
31
32 // Enqueue(normal)
33 __code putQueue3(struct Context* context, struct Queue* queue, struct Element*
       new_element) {
34     struct Element* last = queue->last;
35     last->next = new_element;
36
37     queue->last = new_element;
38     queue->count++;
39
40     goto meta(context, context->next);
41 }
42
43 // Meta Code Gear(stub)
44 __code putQueue3_stub(struct Context* context) {
45     goto putQueue3(context,
46                    &context->data[ActiveQueue]->queue,
47                    &context->data[context->dataNum]->element);
48 }
49
50 // Enqueue(nothing element)
51 __code putQueue4(struct Context* context, struct Queue* queue, struct Element*
       new_element) {
52     queue->first = new_element;
53     queue->last = new_element;
54     queue->count++;
55
56     goto meta(context, context->next);
57 }
58
59 // Meta Code Gear(stub)
60 __code putQueue4_stub(struct Context* context) {
61     goto putQueue4(context,
62                    &context->data[ActiveQueue]->queue,
63                    &context->data[context->dataNum]->element);
64 }
```

ソースコード 4.5: Enqueue

```
1  // Dequeue
2  __code getQueue(struct Context* context, struct Queue* queue, struct Node* node) {
3      if (queue->first == 0)
4          return;
5
6      struct Element* first = queue->first;
7      queue->first = first->next;
8      queue->count--;
9
10     context->next = GetQueue;
11     stack_push(context->code_stack, &context->next);
12
```

```
13      context->next = first->task->code;
14      node->key = first->task->key;
15
16      goto meta(context, GetTree);
17  }
18
19  // Meta Code Gear(stub)
20  __code getQueue_stub(struct Context* context) {
21      goto getQueue(context,
22                    &context->data[ActiveQueue]->queue,
23                    &context->data[Node]->node);
24  }
```

ソースコード 4.6: Dequeue

ソースコード:4.5 とソースコード:4.6 はシングルスレッドでは正常に動作するが、マルチスレッドでは期待した動作を達成できない可能性がある。並列実行すると同じメモリ位置にアクセスされる可能性があり、データの一貫性が保証できないからである。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 の操作を行うように変更する必要がある。CAS はデータの比較・置換をアトミックに行う命令である。メモリからのデータの読み出し、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ていないということを保証することができる。CAS に失敗した場合は置換は行わず、再びデータの読み出しから始める。

ソースコード:4.5 44 行目の putQueue3, 51 行目の putQueue4, ソースコード:4.6 2 行目の getQueue が実際に Queue を操作している Code Gear である。これらの Code Gear から CAS を利用したソースコード:4.7, ソースコード:4.8 の Code Gear に接続を変更することでスレッドセーフな Queue として扱うことが可能になる。Code Gear は Gears OS における最小の処理単位となっており、接続を変更することでプログラムの振る舞いを柔軟に変更することができる。

```
1   // Enqueue(normal)
2   __code putQueue3(struct Context* context, struct Queue* queue, struct Element*
        new_element) {
3       struct Element* last = queue->last;
4
5       if (__sync_bool_compare_and_swap(&queue->last, last, new_element)) {
6           last->next = new_element;
7           queue->count++;
8
9           goto meta(context, context->next);
10      } else {
11          goto meta(context, PutQueue3);
12      }
13  }
14
15  // Enqueue(nothing element)
16  __code putQueue4(struct Context* context, struct Queue* queue, struct Element*
        new_element) {
17      if (__sync_bool_compare_and_swap(&queue->first, 0, new_element)) {
18          queue->last = new_element;
```

```
19          queue->count++;
20
21          goto meta(context, context->next);
22      } else {
23          goto meta(context, PutQueue3);
24      }
25  }
```

ソースコード 4.7: Enqueue using CAS

```
1  // Dequeue
2  __code getQueue(struct Context* context, struct Queue* queue, struct Node* node) {
3      if (queue->first == 0)
4          return;
5
6      struct Element* first = queue->first;
7      if (__sync_bool_compare_and_swap(&queue->first, first, first->next)) {
8          queue->count--;
9
10         context->next = GetQueue;
11         stack_push(context->code_stack, &context->next);
12
13         context->next = first->task->code;
14         node->key = first->task->key;
15
16         goto meta(context, Get);
17     } else {
18         goto meta(context, GetQueue);
19     }
20 }
```

ソースコード 4.8: Dequeue using CAS

4.5 Persistent Data Tree

Gears OS では Persistent Data Gear の管理に木構造を用いる。この木構造は非破壊で構成される。非破壊木構造とは一度構築した木構造を破壊することな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 (図:4.3) ため、読み書きを平行して行うことが可能である。赤色で示したノードが新しく追加されたノードである。非破壊木構造の基本的な戦略は、変更したいノードへのルートノードからのパスを全てコピーする。そして、パス上に存在しないノードはコピー元の木構造と共有することである。

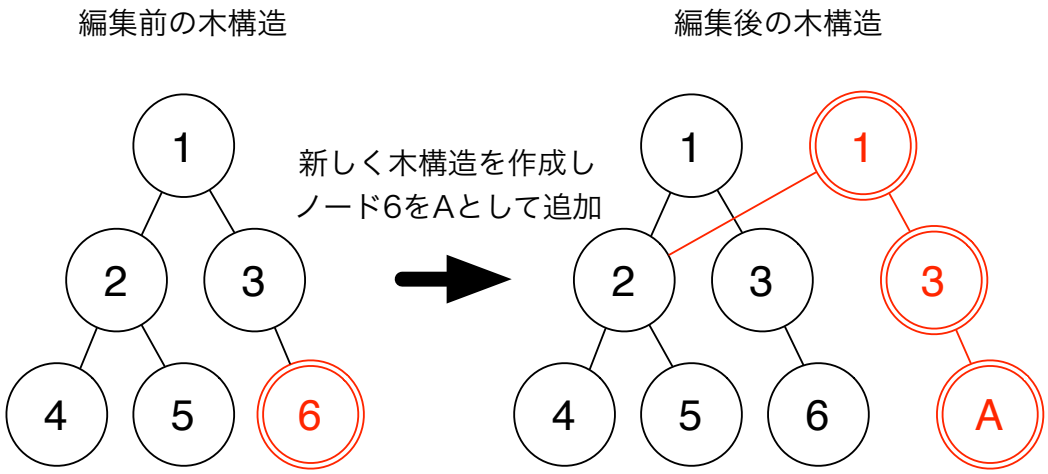

図 4.3: 木構造の非破壊的編集

木構造はディレクトリツリー、構文木など階層構造を持つデータを表現する。またはデータベースのインデックスなど情報を探索しやすくするための探索木としても用いられる。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。最悪なケースでは事実上の線形リストになり、計算量が O(n) となる。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。

Red-Black Tree は挿入・削除を行ったあとに変更したノードからルートへのパスを辿りながら Red-Black Tree の条件を満たすように色の変更や木の回転を行う。関数呼び出しが可能なプログラミング言語では戻り値でパスを辿ることができるが、CbC は末尾呼び出し最適化が行われるように記述する必要があるのでパスを辿るにはノードに親への参照を持たせるか挿入・削除時に辿ったパスを記憶するしかない。ノードが親への参照を持つと非破壊木構造を構築することが出来ないので、辿ったパスを記憶する方法を用いる。辿ったパスを記憶するため Context にスタックを持たせる。

ソースコード:4.9 は Context に追加する Tree, Node および Tree の操作を行う Code Gear 名の定義である。

```
1  // Code Gear Name
2  enum Code {
3      PutTree,
4      Replace,
5      Insert,
6      Compare,
7      RotateL,
8      RotateR,
9      SetTree,
10     InsertCase1,
11     InsertCase2,
12     InsertCase3,
13     InsertCase4,
14     InsertCase4_1,
15     InsertCase4_2,
16     InsertCase5,
17     StackClear,
18     Get,
19     Search,
20 };
21
22 // Compare Result
23 enum Relational {
24     EQ,
25     GT,
26     LT,
27 };
28
29 // Unique Data Gear
30 enum UniqueData {
31     Tree,
32     Traverse,
33     Node,
34 };
35
36 // Context definication
37 struct Context {
38     stack_ptr node_stack;
39 };
40
41 // Red-Black Tree definication
42 union Data {
43     // size: 8 byte
44     struct Tree {
45         struct Node* root;
46     } tree;
47     // size: 12 byte
48     struct Traverse {
49         struct Node* current;
50         int result;
51     } traverse;
52     // size: 32 byte
53     struct Node {
54         int key;
55         union Data* value;
56         struct Node* left;
57         struct Node* right;
```

```
58          enum Color {
59              Red,
60              Black,
61          } color;
62      } node;
63  };
```

ソースコード 4.9: Context: Red-Black Tree

Tree は参照する木を格納する Code Gear である。この Code Gear は Context の生成時に生成される。Traverse は木の探索に用いられる Code Gear である。Code Gear は末尾最適化されるので呼び出し元の情報が残らない。参照しているノードの情報を Code Gear 間で持ち歩くためには Traverse のような Data Gear が必要になる。

赤ノードが続かないという Red-Black Tree の条件を満たすか判定する Code Gear はソースコード:4.10 の通りである。まず、親の情報が必要なのでパスを記憶しているスタックから親ノードを取得する。親ノードが黒である場合、木を回転する必要はなく木は平衡を保っているので木に対する操作を終了する。

```
1  // Code Gear
2  __code insertCase2(struct Context* context, struct Node* current) {
3      struct Node* parent;
4      stack_pop(context->node_stack, &parent);
5
6      if (parent->color == Black) {
7          stack_pop(context->code_stack, &context->next);
8          goto meta(context, context->next);
9      }
10
11     stack_push(context->node_stack, &parent);
12     goto meta(context, InsertCase3);
13 }
14
15 // Meta Code Gear(stub)
16 __code insert2_stub(struct Context* context) {
17     goto insertCase2(context, context->data[Traverse]->traverse.current);
18 }
```

ソースコード 4.10: Insert Case

木の左回転を行う Code Gear はソースコード:4.11 の通りである。自分、親、兄弟の 3 点のノードの回転である。回転を行ったあとにも Red-Black Tree の条件を満たしているか確認する必要があるので回転後に変更された親ノードを再びスタックに記憶する。また、回転の際に現在見ているノードが変更する必要がある。

```
1  // Code Gear
2  __code rotateLeft(struct Context* context, struct Node* node, struct Tree* tree,
       struct Traverse* traverse) {
3      struct Node* tmp = node->right;
4      struct Node* parent = 0;
5
6      stack_pop(context->node_stack, &parent);
7
8      if (parent) {
9          if (node == parent->left)
10             parent->left = tmp;
11         else
12             parent->right = tmp;
13     } else {
14         tree->root = tmp;
15     }
16
17     stack_push(context->node_stack, &parent);
18
19     node->right = tmp->left;
20     tmp->left = node;
21     traverse->current = tmp;
22
23     stack_pop(context->code_stack, &context->next);
24     goto meta(context, context->next);
25 }
26
27 // Meta Code Gear(stub)
28 __code rotateLeft_stub(struct Context* context) {
29     goto rotateLeft(context,
30                     context->data[Traverse]->traverse.current,
31                     &context->data[Tree]->tree,
32                     &context->data[Traverse]->traverse);
33 }
```

ソースコード 4.11: Rotate Left

4.6 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 を参照している。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 はソースコード:4.8 の通りである。TaskQueue 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。Task の実行後に再び Task の取得を行う Code Gear に戻

る必要がある。Context は実行する Code Gear のスタックを持っているのでそのスタックに積む (ソースコード:4.8 11 行目) ことで戻ることができる。

Task に格納され Worker で実行される Code Gear はソースコード:4.12 の通りである。ソースコード:4.12 は指定された要素の値を 2 倍する Twice という例題である。Twice は並列実行される。

```
1  // Code Gear
2  __code twice(struct Context* context, struct LoopCounter* loopCounter, int index,
       int alignment, int* array) {
3      int i = loopCounter->i;
4
5      if (i < alignment) {
6          array[i+index*alignment] = array[i+index*alignment]*2;
7          loopCounter->i++;
8
9          goto meta(context, Twice);
10     }
11
12     loopCounter->i = 0;
13
14     stack_pop(context->code_stack, &context->next);
15     goto meta(context, context->next);
16 }
17
18 // Meta Code Gear(stub)
19 __code twice_stub(struct Context* context) {
20     goto twice(context,
21                &context->data[LoopCounter]->loopCounter,
22                context->data[Node]->node.value->array.index,
23                context->data[Node]->node.value->array.alignment,
24                context->data[Node]->node.value->array.array);
25 }
```

ソースコード 4.12: Task Sample

並列処理される Code Gear と言っても他の Code Gear と完全に同じである。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すべて並列に動作させることができるということを意味する。

4.7 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。

TaskManager は Worker の管理も行う。メインとなる Context には Worker の情報が格納されており、TaskManager はこの Context を参照して Worker の起動・停止を行う。ソースコード 4.13 は Worker を起動する Code Gear である。

```
1  // Code Gear
2  __code createWorker(struct Context* context, struct LoopCounter* loopCounter, struct
       Worker* worker) {
3      int i = loopCounter->i;
4
5      if (i < worker->num) {
6          struct Context* worker_context = &worker->contexts[i];
7          worker_context->next = GetQueue;
8          worker_context->data[Tree] = context->data[Tree];
9          worker_context->data[ActiveQueue] = context->data[ActiveQueue];
10         pthread_create(&worker_context->thread, NULL, (void*)&start_code,
               worker_context);
11         worker_context->thread_num = i;
12         loopCounter->i++;
13
14         goto meta(context, CreateWorker);
15     }
16
17     loopCounter->i = 0;
18     goto meta(context, TaskManager);
19 }
20
21 // Meta Code Gear
22 __code createWorker_stub(struct Context* context) {
23     goto createWorker(context, &context->data[LoopCounter]->loopCounter, &context->
           data[Worker]->worker);
24 }
```

ソースコード 4.13: InitWorker

第5章　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。

5.1　Cerium

Cerium ではサブルーチンまたは関数を Task の単位としてプログラムを分割する。Task には依存関係のある Task を設定することができ、TaskManager が依存関係を解決することで並列処理を実現している。実行に必要なデータのアドレスを Task の生成時に設定することで Task はデータにアクセスすることが可能になる。データは汎用ポインタとして渡されるので Task 側で型変換して扱うことになる。ここで問題となるのが Task 間だけにしか依存関係がないことと Task 実行時にデータの型情報がないことである。

本来 Task は必要なデータが揃ったときに実行されるべきものである。不正なデータが渡された場合、実行せずに不正なデータがであることを実行者に伝えることが望ましい。Cerium では Task の終了のみに着目して依存関係を解決するので途中で不正なデータになっても処理を続けてしまい不正な処理を特定することが難しい。

複雑なデータ構造を持つ場合、間違った型変換でデータの構造を破壊する可能性がある。型システムは正しい型に対して正しい処理が行われることを前提にしてプログラムの正しさを保証する。型情報がない Cerium では型システムによる安全性を保証できず、型に基づくバグが入り込む可能性がある。

Gears OS では Code Gear, Data Gear という単位でプログラムを分割する。Code Gear は処理の単位、Data Gear はデータそのものである。Code Gear には Input/Output Data Gear が設定されており、Input と Output の関係が Code Gear 間の依存関係となる。Gears OS の TaskManager は Data Gear が格納されている Persistent Data Tree を監視して依存関係を解決する。Data Gear は Context に構造体として定義されており、型情報を持つ。

5.2 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。

5.3 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 (ソースコード:5.1) 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。

```
1 #pragma omp parallel for
2 for(int i=0;i<N;i++) {
3     // Processing
4 }
```

ソースコード 5.1: 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。

5.4　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。

第6章　Gears OS の評価

現在の Gears OS には非破壊木構造を Red-Black Tree アルゴリズムに基づいて構築する Persistent Data Tree, CAS を用いてデータの一貫性を保証する TaskQueue, TaskQueue から Task を取得し並列に実行する Worker が実装されている。つまり、依存関係のない処理ならば並列処理することが可能である。

本章では依存関係のない簡単な例題を用いて Gears OS の評価を行う。また、Gears OS の実装自体への評価も行う

6.1　Twice

Twice は与えられた整数配列を 2 倍にする例題である。
以下の流れで処理は行われる。

- 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。

Gears OS 上に Twice を実装し、要素数 2^{17}*1000 のデータを 640 個の Task に分割してコア数を変更して測定を行なった。結果は表:6.1, 図:6.1 の通りである。

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

表 6.1: 要素数 2^{17}*1000 のデータに対する Twice

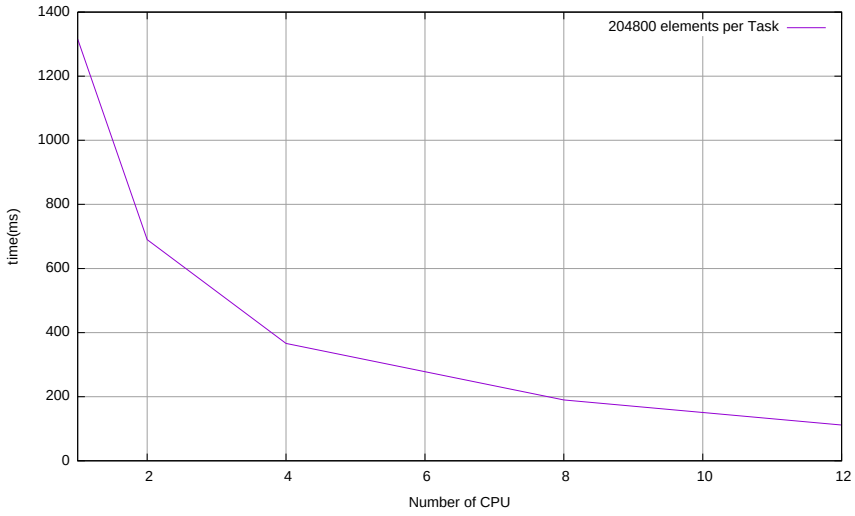

図 6.1: 要素数 2^{17}*1000 のデータに対する Twice

1 CPU と 12 CPU では約 11.8 倍の速度向上が見られた。十分な台数効果が出ていることがわかる。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出ないことがある。Code Gear には実行時間を予測可能なものにするという特徴があるので、その性質を利用してタスクが最適な粒度なのか検査する機能が必要になると考えられる。

今回、例題に用いた Twice は依存関係のない並列処理である。本来、並列処理には複雑な依存関係が存在するのが一般的である。並列フレームワークには複雑な依存関係を解決しながら十分な並列度を保てることが必須なので依存関係を解決するための TaskManager の実装が必要である。

6.2　Gears OS の実装

- Code Segment
  Code Segment は分割・統合を容易に行うことができる。巨大な Code Segment を記述することも可能だが、それは好ましくない。今回の実装では制御構文で Code Segment を分割するようにコーディングを行った。制御構文で分割することで Code Segment のサイズを小さくすることには繋がったが、Code Segment の数が増加した。Code Segment には必ず stub が付属するので記述量が 2 倍ほどになる。CbC の構文サポートを利用することで記述量を減らすことはできるが、正しいコードに変換できない場合もある。Code Segment の継続ではスタックに値が積まれないので、デバッグの際にどの Code Segment から接続されたか特定できない。

- Data Segment
  Data Segment は共用体と構造体を用いて表現した。現状ではすべての Context が同じ定義を持つことになる。必要ない Data Segment を持つ場合もあるので好ましくない。定義されているべき Data Segment は実行可能な Code Segment のリストから推論することが可能である。専用の構文を準備し、必要な Data Segment のみ持つようにするべきである。

第7章　結論

先行研究である Cerium の開発を通して得られた知見を元に Code Segment, Data Segment によって構成される Gears OS の設計・実装を行なった。実装には本研究室で開発している CbC(Continuation based C) を用いた。

Code Segment は処理、Data Segment はデータの単位である。Code Segment 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。このようなスタックに積まない継続を軽量継続と呼び、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。プログラムを Code/Data Segment で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 を Code/Data Segment の考えに基づいて設計を行なった。Gears OS には Code/Data Segment と同等なものとして Code/Data Gear を定義した。Code Gear はプログラムの処理そのもので、Data Gear は int や文字列などの Primitive Data Type を複数持っている構造体として表現する。Code Gear は任意の数の Data Gear を参照し、任意の数の Data Gear に書き込みを行う。Gear の特徴として処理やデータ構造が Code/Data Gear に閉じている。これにより実行時間、メモリ使用量などを予測可能なものにする。

Gears OS の基本的な機能として Allocator, TaskQueue, Persistent Data Tree, Worker の実装を行なった。Gears OS では Context に情報が格納される。格納される情報には接続可能な 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 を確保するためのメモリ空間などがある。Context はスレッドごとに存在し、それぞれが異なる Context を参照している。Allocator は Context が持っているメモリ空間のアドレスを変更し、Temporal Data Gear の確保を行う。確保される Data Gear は処理後には必要なくなるものなのでリニアに確保するだけの単純な処理である。TaskQueue は並列処理される Task を管理する。Gears OS で Task は実行する Code Gear と実行に必要な Data Gear の組で表現する。TaskQueue はすべての Context で共有され、マルチスレッドでデータの一貫性を保つために Compare and Swap(CAS) を用いた。Persistent Data Tree は Data Gear を管理する。非破壊木構造で構成され、Red-Black Tree アルゴリズムによって平衡性が保たれる。Persistent Data Tree はすべての Context で共有される。非破壊木構造なので読み書きを平行して行うことができる。Gears OS では Persistent Data Tree への書き込みのみで相互作用を発生させ目的の処理を達成する。Worker は Task を並列処理する。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。CAS を用いて TaskQueue にアクセスし、Task を取得する。取得した Task の情報を元に Persistent Data Tree から Data Gear を取得し、Code Gear を実行する。また、Gears OS 自体が Code/Data Segment を用いたプログラミングの指針となるように実装を行った。

Gears OS を用いて簡単な例題を実装し、評価を行った。与えられた要素を 2 倍にする Twice という依存関係がない並列処理の例題を Gears OS 上に実装した。1 CPU と 12 CPU で約 11.8 倍の速度向上を確認し、Gears OS を用いることで十分な並列処理性能を引き出せることを示した。

7.1　今後の課題

例題として Twice を用いて並列処理の性能を示したが、Twice は依存関係が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。複雑な並列処理を行えるようにするために依存関係を解決する TaskManager の実装が必要である。

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。Code/Data Segment を用いて各プロセッサのアーキテクチャにマッピングした実行機構を実装し、演算に利用できるようにする必要がある。

Data Segment は共用体と構造体によって定義したが、Data Segment 専用の構文を準備するべきである。Context は必要な Data Segment のみを持っているべきであり、すべての Data Segment を知っている必要はない。必要な Data Segment は実行可能な Code Segment のリストを参照することで推論することができる。

型情報を残すために Data Segment を定義しているが Data Segment 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 Segment の型情報を検査する型システムを Gears OS 上に実装する必要がある。

謝辞

本研究を行うにあたりご多忙にも関わらず日頃より多くのご助言、ご指導をいただきました河野真治准教授に心より感謝いたします。また、Cerium の先行研究がなければ本研究は成り立つことはありませんでした。Cerium の設計や実装に関わった全ての先輩方に感謝いたします。

研究を行うにあたり、研究に対する意見、実装、実験に協力いただいた並列信頼研究室の皆さまに感謝いたします。

最後に、長年に渡り理解を示し、支援してくださった家族に感謝いたします。

参考文献

[1] 河野真治, 島袋仁. C with continuation と、その playstation への応用. 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 2000.

[2] 河野真治, 揚挺. C 言語の continuation based c への変換. *SWoPP 2001*, July 2001.

[3] 河野真治. Continuation based c を使ったソースコードのリファクタリング手法. 日本ソフトウェア科学会第 21 回大会論文集, Sep 2004.

[4] 宮國渡, 河野真治. Continuation base c 言語による os システムコールの意味記述. 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), April 2007.

[5] 下地篤樹, 河野真治. 線形時相論理による continuation based c プログラムの検証. 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), April 2007.

[6] 河野真治. 検証を自身で表現できるハードウェア、ソフトウェア記述言語 continuation based c と、その cell への応用. 電子情報通信学会 VLSI 設計技術研究会, March 2008.

[7] 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), April 2008.

[8] 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, Jan 2013.

[9] 渡真利勇飛, 小久保翔平, 河野真治. Cerium task manager における gpu と multicore cpu の同時実行. 第 55 回プログラミング・シンポジウム, Jan 2014.

[10] 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), May 2014.

[11] 小久保翔平, 河野真治. 並列プログラミングフレームワーク cerium の opencl, cuda への対応. 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 2014.

[12] 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), May 2015.

[13] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[14] OpenCL. https://www.khronos.org/registry/cl/sdk/2.1/docs/man/xhtml.

[15] CUDA. http://docs.nvidia.com/cuda/index.html.

発表履歴

- 小久保翔平, 河野真治. 並列プログラミングフレームワーク Cerium の OpenCL, CUDA への対応. 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 2014.
- 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), May 2015.

付録

並列プログラミングフレームワーク Cerium の OpenCL, CUDA 対応

小久保 翔平[†1]　河　野　真　治[†2]

当研究室では、PS3, Linux 及び MacOS X 上で動作する並列プログラミングフレームワーク Cerium を提案している。MacOS X 上で GPGPU を行うには、OpenCL または CUDA を用いる方法が考えられる。OpenCL, CUDA の API に対応した API を Cerium に用意することでデータ並列に対応した。タスク並列で実行する場合、データ転送がオーバーヘッドになる。このオーバヘッドを解決するためには、kernel の実行中にデータ転送を行うことでデータ転送をオーバーラップする必要がある。OpenCL では CommandQueue、CUDA では Stream を複数用いることでデータ転送や kernel の実行を並列に行うことができる。複数の CommandQueue, Stream を用いて、自動で並列実行を行うスケジューラーを実装した。実装したスケジューラーを WordCount、FFT を例題に測定し、考察を行う。

Supporting OpenCL, CUDA in Parallel programming framework Cerium

Shohei KOKUBO[†1] and Shinji KONO [†2]

We are developing parallel programming framework Cerium, that is running on the PS3, Linux, Mac OS X and GPGPU. As GPGPU support on Mac OS X, OpenCL or CUDA can be used in an API set. Data Parallel on GPUGPU is also working on CPUs. In Task Parallel, data transfer causes overhead. To resolve this, pipeline data transfer is used both on GPGPU and Many Cores. OpenCL and CUDA have slightly different behavior on Data Parallel and out of order task execution. In WordCount and FFT examples, we show the different and its analysis.

1. はじめに

GPU の普及と高性能化にともない、GPU の演算資源を画像処理以外の目的にも利用する GPGPU(GPU による汎目的計算) が注目されている。[1] GPU 以外にも Cell[2], SpursEngine, Xeon Phi など様々なプロセッサが存在する。それぞれのプロセッサを利用するにはそれぞれ異なる API を利用する必要があり、それらの対応に多くの時間を取られてしまいプログラムの性能改善に集中することができない。また、GPU や Cell などメモリ空間が異なるプロセッサはデータの転送がオーバーヘッドとなるので、データ転送を効率的に行えるかどうかで処理時間が大きく変わる。

当研究室で開発・改良が行われている並列プログラミングフレームワーク Cerium[3] は様々なプロセッサを統合して扱えるフレームワークを目指している。様々なプロセッサを統合して扱えるフレームワークとしてフランス国立情報学自動制御研究所 (INRIA) が開発している StarPU[4] がある。StarPU は Cerium と同じタスクベースの非同期フレームワークである。タスクという単位で記述することで処理とデータを分離し、より効率的に処理を行うことができる。StarPU にはパイプラインでの実行機構は入ってなく、パイプライン処理を行いたい場合は自分で実装するしかない。しかし、パイプライン処理を書くことは非常に煩雑で難しい。そこで、今回 Cerium に OpenCL, CUDA を用いた Scheduler を新たに実装した。Scheduler は自動でデータ転送をオーバーラップし、パイプラインで処理を行うように設計した。

本論文では、まず OpenCL, CUDA について説明する。その後、既存の Cerium の実装および新たに実装した GPU 実行の機構について説明する。最後に WordCount, FFT を例題として測定し、評価を行う。

†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.

2. Parallel Computing Platform

2.1 OpenCL

OpenCL とは、Multi Core CPU と GPU のよう

なヘテロジニアスな環境を利用した並列計算を支援するフレームワークである。演算用のプロセッサ (本研究では GPU) 上で動作するプログラム OpenCL C という言語で記述する。OpenCL C で記述したプログラムを GPU 上で実行させるために OpenCL Runtime API を利用する。OpenCL ではオペレーティングシステムなどが処理されるメイン CPU のことを host、GPGPU が可能なグラフィックボードなどのことを device と定義している。OpenCL Application は host 側のプログラムと device 側のプログラムが一体となって動作する。この device 上で動作するプログラムを特別に kernel と呼ぶ。

2.1.1 CommandQueue

OpenCL では、device の操作に CommandQueue を使用する。CommandQueue は device に Operation を送るための仕組みである。kernel の実行、input buffer の読み込み、output buffer への書き込みなどが Operation となる。

CommandQueue に投入された Operation は投入された順序で実行される。CommandQueue を生成するときプロパティを指定することで Operation を投入された順序を無視して (out of order) 実行することが可能になる。また複数の CommandQueue を生成し、device に投入することでも our of order で実行することが可能である。

out of order で実行する場合、データの依存関係を設定する必要がある。各 Operation を発行する関数には event_wait_list と event を指定することができ、これらを利用してデータの依存関係を設定することができる。out of order 実行を可能にするプロパティをサポートしている device が少ないため、今回は複数の CommandQueue を用いる方法で実装を行なった。

2.1.2 OpenCL におけるデータ並列

3D グラフィックのような多次元のデータを処理する場合に高い並列度を保つには、データを分割して並列に実行する機能が必要である。これを OpenCL ではデータ並列と呼んでいる。OpenCL では次元数に対応する index があり、OpenCL は 1 つの記述から index の異なる複数の kernel を自動生成する。その添字を global_id と呼ぶ。このとき入力されたデータは WorkItem という処理単位に分割される。

OpenCL は WorkItem に対して、それぞれを識別する ID(global_id) を割り当てる。kernel は get_global_id という API によって ID を取得し、取得した ID に対応するデータに対して処理を行うことでデータ並列を実現する。

データ並列による kernel 実行の場合、clEnqueueNDRangeKernel を使用する。この関数の引数として WorkItem の数と次元数を指定することでデータ並列で実行することができる。

2.2 CUDA

CUDA とは、半導体メーカー NVIDIA 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コンパイラ、ライブラリ、デバッガなどから構成される。プログラミング言語である CUDA C は C 言語ベースに拡張を加えたものである。

CUDA には CUDA Runtime API と CUDA Driver API の 2 種類がある。Driver API は Runtime API に比べてプログラマが管理すべきリソースが多い。しかし、Runtime API より柔軟な処理を行うことができる。今回は Driver API を使用して実装した。

CUDA も OpenCL と同様に、制御を行う CPU 側を host、GPU 側を device と定義している。また、device 上で動作するプログラムも OpenCL と同様に kernel と呼ぶ。

2.2.1 Stream

CUDA には OpenCL の CommandQueue と似たような仕組みとして Stream がある。Stream は host 側で発行された Operation を一連の動作として device で実行する。Stream に発行された Operation は発行された順序で実行されることが保証されている。異なる Stream に発行された Operation に依存関係が存在しない場合、Operation を並列に実行することができる。

Stream は cuStreamCreate という Driver API で生成される。引数に Stream を指定しない API はすべて host 側をブロックする同期的な処理となる。複数の Stream を同時に走らせ Operation を並列に実行するためには非同期な処理を行う API を利用する必要がある。

2.2.2 CUDA におけるデータ並列

CUDA では OpenCL の WorkItem に相当する単位を thread と定義している。この thread をまとめたものを block と呼ぶ。CUDA でデータ並列による kernel 実行をする場合、cuLaunchKernel API を使用する。この関数は引数として各座標の block 数と各座標の block 1 つ当たりの thread 数を指定することでデータ並列で実行できる。

cuLaunckKernel で kernel を実行すると各 thread に対して block ID と thread ID が割り当てられる。CUDA には OpenCL とは異なり、ID を取得する API は存在しない。代わりに、kernel に組み込み変数が準備されており、それを参照し、対応するデータに対し処理を行うことでデータ並列を実現する。組み込み変数は以下の通りである。

- uint3 blockDim
- uint3 blockIdx
- uint3 threadIdx

各組み込み変数はベクター型で、blockDim.x とすると x 座標の thread 数を参照することができる。

blockIdx.x とすると x 座標の block ID が参照でき、threadIdx.x とすると x 座標の thread ID を参照することができる。blockDim.x * blockIdx.x + threadIdx.x で OpenCL の get_global_id(0) で取得できる ID に相当する ID を算出することができる。例として、ある kernel で get_global_id(0) の返り値が 13 の場合、CUDA では図:1 のようになる。

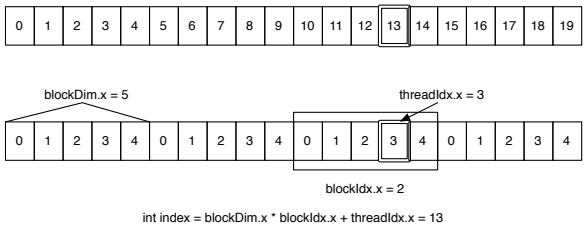

図 1 Calculate Index

3. Cerium

Cerium は、当初 Cell 用の Fine-Grain Task Manager[5] として当研究室で開発された。TaskManager, SceneGraph, Rendering Engine の 3 つの要素から構成されており、今では、PS3 および Linux, MacOS X 上で動作する。GPGPU の Data Parallel を含めて、同じ形式で並列プログラムを記述することができる。

3.1 Cerium TaskManager

TaskManager は、Task と呼ばれる分割されたプログラムを管理する。関数やサブルーチンを Task として扱い、Task 同士の依存関係を考慮しながら実行される。Task は TaskManager を使って生成する。Task を生成する際に、以下のような要素を設定することができる。

- input data
- output data
- parameter
- cpu type
- dependency

input, output, parameter は関数でいうところの引数に相当する。cpy type は Task がどのようなの Device の組み合わせで実行されるかを示す。dependency は他の Task との依存関係を示している。

図:2 は Cerium が Task を生成/実行する場合のクラスの構成である。TaskManager で依存関係が解消され、実行可能になった Task は ActiveTaskList に移される。ActiveTaskList に移された Task は依存関係が存在しないのでどのような順序で実行されても問題ない。Task は Scheduler に転送しやすい TaskList に変換してから cpy type に対応する Scheduler に Synchronized Queue である mail を通して転送される。Scheduler ではパイプラインで task が処理される (図:3)。Task が終了すると Scheduler から TaskManager に mail を通して通知される。その通知に従って依存関係が処理され、再び TaskManager から Scheduler に Task が転送される。

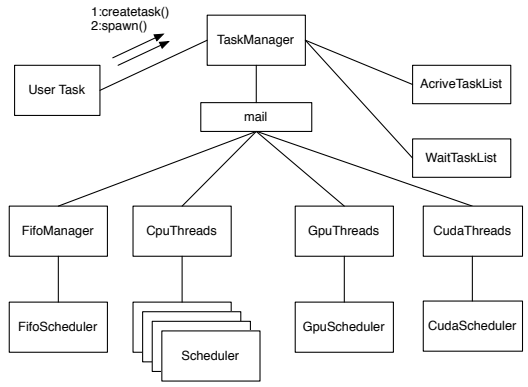

図 2 Task Manager

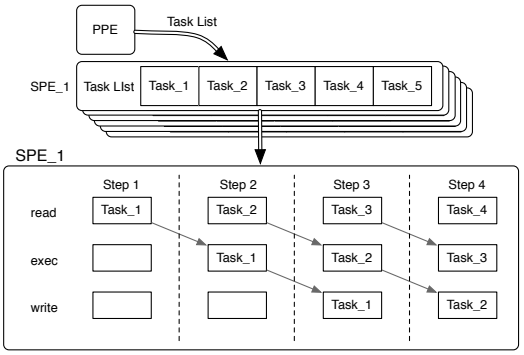

図 3 Task Scheduler

以下に Task を生成する例題を示す。表:1 は Task を生成に用いる API を示している。input データを 2 つ用意し、input データの各要素同士を乗算し、output に格納する multiply という例題である。

```
void
multi_init(TaskManager *manager)
{
    A = new float[length];
    B = new float[length];
    C = new float[length];
    for(int i=0; i<length; i++) {
        A[i]=(float)(i+1000);
        B[i]=(float)(i+1)/10.f;
    }

    // create task
    HTask* multiply = manager->create_task(
        MULTIPLY_TASK);
    // set cputype
    multiply->set_cpu(spe_cpu);
```

```
    // set indata
    multiply->set_inData(0,(memaddr)A,
        sizeof(float)*length);
    multiply->set_inData(1,(memaddr)B,
        sizeof(float)*length);
    // set outdata
    multiply->set_outData(0,(memaddr)C,
        sizeof(float)*length);
    multiply->set_param(0,(long)length);
    // spawn task
    multiply->iterate(length);
}
```

Code 1　multiply

create_task	Task を生成する
set_inData	Task への入力データのアドレスを追加
set_outData	Task からのデータ出力先アドレスを追加
set_param	Task のパラメータ
wait_for	Task の依存関係
set_cpu	Task を実行する Device の設定
spawn	Task を登録する
iterate	データ並列で実行する Task として登録する

表 1　Task 生成に用いる API

CPU で実行される Task(OpenCL, CUDA でいう kernel) の記述は以下のようになる。表:2 は Task 側で使用する API である。

```
static int
run(SchedTask *s)
{
    // get input
    float* A = (float*)s->get_input(0);
    float* B = (float*)s->get_input(1);
    // get output
    float* C = (float*)s->get_output(0);
    // get parameter
    long length = (long)s->get_param(0);

    for(int i=0;i<length;i++)
        C[i]=A[i]*B[i];

    return 0;
}
```

Code 2　task

get_input	入力データのアドレスを取得
get_output	データ出力先のアドレスを取得
get_param	パラメータを取得

表 2　Task 側で使用する API

3.2　Cerium におけるデータ並列

Cerium でデータ並列による実行をサポートするために、OpenCL の API に合わせた iterate という API を用意した。iterate は length を引数として受け取り、Scheduler で length の値と受け取った引数の個数を次元数として Task 数を計算し、データ並列で実行する Task として生成する。

例として、CPU 数 4、一次元で 10 個のデータに対してデータ並列実行を行なった場合、各 CPU が担当する index は表:3 のようになる。

	CPU0	CPU1	CPU2	CPU3
1	0	1	2	3
2	4	5	6	7
3	8	9		

表 3　Data 並列実行時の index の割り当て

各 CPU が担当する index は SchedTask に格納してある。データ並列で実行する Task の記述は以下のようになる。

```
static int
run(SchedTask *s)
{
    // get input
    float* A = (float*)s->get_input(0);
    float* B = (float*)s->get_input(1);
    // get output
    float* C = (float*)s->get_output(0);
    // get index
    long i = (long)s->x;

    C[i]=A[i]*B[i];

    return 0;
}
```

Code 3　example

並列プログラムでは、並列化する Task が全部同一であるということは少なくない。iterate を実装したことで、Task を生成する部分をループで回す必要はなくなり、OpenCL と同様に 1 つの記述で異なる index を持つ Task を Multi Core CPU 上で実行することが可能になった。

4.　Cerium の GPGPU への対応

本章では、まずはじめに GPU プログラミングの特徴および問題について述べ、Cerium への実装でどのように対応したかについて説明する。

4.1　GPU プログラミングの特徴および問題

まず Multi Core CPU に対するプログラミングと同様に性能を向上させるためには、プログラム全体を対象とした並列度を高くしなければならない。明示的な並列化部分はループ部分である。GPU は数百個のコアを有しており、ループ部分に対してデータ並列で処理を行うことで CPU より高速で演算を行うことができる。プログラムの大部分がループであれば、データ並列による実行だけでプログラムの性能は向上する。しかし、多くのプログラムはその限りではない。GPGPU においてネックになる部分はデータ転送である。GPU の Memory 空間 (図:4) は CPU(図:5) とは異なり、Shared Memory ではないため host と device 間でデータの共有ができない。データにアクセスするためには Memory 空間ごとコピーするしかない。これが大きなオーバーヘッドになるので、データ転送をオーバーラップする必要がある。今回新たに、デー

タ転送を自動でオーバーラップするように OpenCL および CUDA を用い Scheduler を実装した。

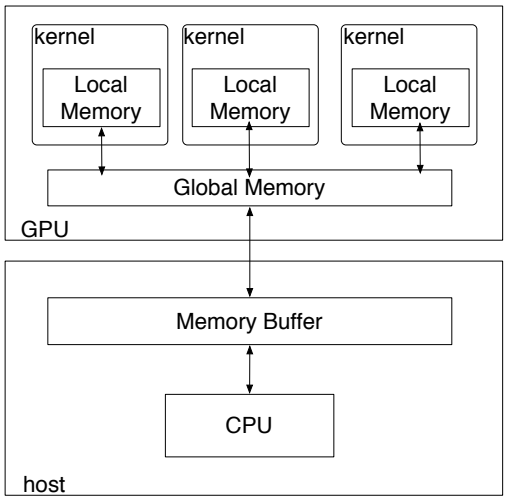

図 4 Gpu Architecture

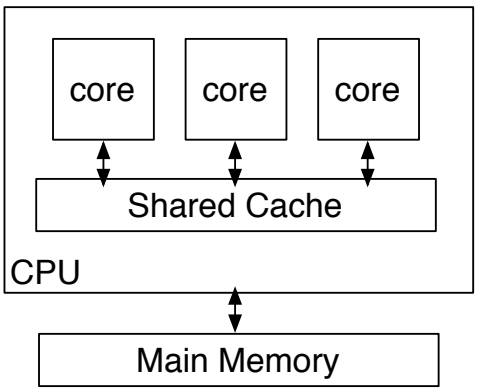

図 5 Cpu Architecture

4.2 OpenCL および CUDA を用いた Scheduler の実装

Scheduler と CpuThreads に対応させる形で OpenCL を用いた GpuScheduler, GpuThreads、CUDA を用いた CudaScheduler, CudaThreads を実装した。TaskManager から転送された TaskList の情報をもとに device 上のメモリ領域を確保する。その後、OpenCL ならば CommandQueue、CUDA ならば Stream に Operation を発行していく。Operation は発行された順序で実行されるので、host から device へのデータ転送、kernel の実行、device から host へのデータ転送の順に発行する。非同期 API を用いることでデータ転送や kernel の実行を並列に行うことができる。通常、非同期 API を用いる場合は依存関係を考慮した同期が必要になるが転送されてくる Task の依存関係は TaskManager ですべて解消されているので Scheduler 側では順番を考えず Task を実行して問題ない。host から device へのデータ転送は、OpenCL では clEnqueueWriteBuffer、CUDA では cuMempcyHtoDAsync を用いて行われる。clEnqueueWriteBuffer は第三引数に CL_FALSE を指定することで非同期なデータ転送を行う。転送されてきた TaskList からデータ並列またはタスク並列で実行するか決定する。データ並列で実行する場合は、OpenCL では clEnqueueTaskNDRangeKernel、CUDA では cuLaunchKernel を用いる。タスク並列で実行する場合は、OpenCL では clEnqueueTask、CUDA では cuLaunckKernel の引数を 1 に設定することで実行することができる。device から host へのデータ転送は、OpenCL では clEnqueuReadBuffer、CUDA では cuMemcpyDtoHAsync を用いて行われる。clEnqueueReadBuffer も clEnqueueWriteBuffer と同様に第三引数に CL_FALSE を指定することで非同期実行となる。転送されてきた Task がすべて終了すると Synchronized Queue である mail を通して TaskManager に Task の終了を通知する。終了が通知されると TaskManager で依存関係が解消し、再び TaskList を転送する。GpuScheduler および CudaScheduler は複数の CommandQueue および Stream を持っており、パイプラインで実行される。

kernel の記述は以下のようになる。

```
__kernel void
multi(__global const long *params, __global
      const float* A, __global const float* B,
      __global float* C)
{
    // get index
    long id = get_global_id(0);
    C[id]=A[id]*B[id];
}
```

Code 4 multiply(OpenCL)

```
extern "C" {
    __global__ void multi(long* params, float*
          A, float* B, float* C) {
        // calculate index
        int id = blockIdx.x * blockDim.x +
             threadIdx.x;
        C[id]=A[id]*B[id];
    }
}
```

Code 5 multiply(CUDA)

修飾子など若干の違いはあるが、ほぼ同じ記述で書くことができるが CPU, OpenCL, CUDA のどれか 1 つの記述から残りのコードも生成できるようにすることが望ましい。

5. Benchmark

本章では、WordCount, FFT を例題として用い、

本研究で実装した GpuScheduler および CudaScheduler の測定を行う。

測定環境

- OS : MacOS 10.9.2
- CPU : 2*2.66GHz 6-Core Intel Xeon
- GPU : NVIDIA Quadro K5000 4096MB
- Memory : 16GB 1333MHz DDR3
- Compiler : Apple LLVM version 5.1 (clang-503.0.40) (based on LLVM 3.4svn)

6. WordCount

今回は 100MB のテキストファイルに対して WordCount を行なった。表:4 は実行結果である。

	Run Time
1 CPU	0.73s
2 CPU	0.38s
4 CPU	0.21s
8 CPU	0.12s
OpenCL(no pipeline)	48.32s
OpenCL(pipeline)	46.74s
OpenCL Data Parallel	0.38s
CUDA(no pipeline)	55.71s
CUDA(pipeline)	10.26s
CUDA Data Parallel	0.71s

表 4 WordCount

パイプライン処理を行うことで CUDA では 5.4 倍の性能向上が見られた。しかし、OpenCL ではパイプライン処理による性能向上が見られなかった。OpenCL と CUDA を用いたそれぞれの Scheduler はほぼ同等な実装である。OpenCL でパイプライン処理を行うために実行機構を見直す必要がある。一方で、データ並列による実行は 1CPU に対して OpenCL では 1.9 倍、CUDA では 1.02 倍という結果になった。どちらもタスク並列による実行よりは優れた結果になっている。CUDA によるデータ並列実行の機構を見直す必要がある。

6.1 FFT

次に、フーリエ変換と周波数フィルタによる画像処理を行う例題を利用し測定を行う。使用する画像のサイズは 512*512 で、画像に対して High Pass Filter をかけて変換を行う。表:6.1 は実行結果である。

	Run Time
1 CPU	0.48s
2 CPU	0.26s
4 CPU	0.17s
8 CPU	0.11s
OpenCL	0.09s
CUDA	0.21s

表 5 FFT

1CPU に対して OpenCL ではの 5.3 倍、CUDA では 2.2 倍の性能向上が見られた。しかし、WordCount の場合と同様に OpenCL と CUDA で差がある。WordCount と FFT の結果から CudaScheduler によるデータ並列実行機構を見直す必要がある。また、FFT の OpenCL の kernel は cl_float2 というベクター型を用いている。CUDA では cl_float2 を float に変換して演算している。OpenCL ではベクターの演算なので、その部分に最適化がかかっており結果が良くなっている可能性がある。

7. ま と め

本研究では並列プログラミングフレームワーク Cerium を OpenCL および CUDA に対応させた。OpenCL および CUDA に対応させたことで Cerium は単一の記述から CPU および GPU 上での実行が可能になった。WordCount, FFT を例題に用い、Scheduler の測定も行なった。OpenCL と CUDA で異なる結果が出たことからそれぞれで最適なチューニングの方法が違うことがわかる。どちらもチューニングを行えば同等な結果が出ると考えられるのでプロファイラなどを用いて、実装を見直すことが今後の課題となる。また、Cerium は CPU と GPU の同時実に対応している。しかし、スケジューリングを行わず Task を CPU, GPU に対し交互に割り振っているため CPU 単体、GPU 単体で実行するより結果が悪くなる。Task の割り当てを最適化することで性能を向上させることが予想される。スケジューリングの方法として、一度 Task を CPU のみ、GPU のみで実行し、プロファイルを取ることで Task の割り当てを決定するなどが考えられる。

参 考 文 献

1) Yasuhiko OGATA, Toshio Endo, Naoya MARUYAMA, Satoshi MATSUOKA: 性能モデルに基づく CPU 及び GPU を併用する効率的な FFT ライブラリ, 情報処理学会論文誌コンピューティングシステム (2008).
2) Sony Corporation: Cell broadband engine architecture (2005).
3) : SourceForge.JP: Cerium Rendering Engine, https//sourceforge.jp/projects/cerium/.
4) Cédric Augonnet, Samuel Thibault, Raymond Namyst, Pierre-André Wacrenier: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, Vol.23, pp. 187–198 (2011).
5) 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 (2008).
6) Chiaki SUGIYAMA: SceneGraph と StatePattern を用いたゲームフレームワークの設計と実装 (2008).
7) 金城裕, 河野真治, 多賀野海人, 小林佑亮 (琉球大学)：ゲームフレームワーク Cerium TaskManager の改良, 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) (2011).
8) 當眞大千, 河野真治：Cerium Task Manager におけるマルチコア上での並列実行機構の実装, 第53 回プログラミング・シンポジウム (2012).
9) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
10) Khronos OpenCL Working Group: *OpenCL 1.2 Reference Pages* (2012).
11) : CUDA, https://developer.nvidia.com/category/zone/cuda-zone/.

Monad に基づくメタ計算を基本とする Gears OS の設計

小久保 翔平[†1]　伊 波 立 樹[†2]　河 野 真 治[†2]

本研究室では Code Gear, Data Gear を用いた並列フレームワークの開発を行なっている。Code Gear, Data Gear は処理とデータの単位である。並列実行に必要な Meta な機能を関数型言語における Monad の原理に基づいて、実現する。今回設計した Gears OS では Code Gear, Data Gear それぞれに Meta Code Gear と Meta Data Gear を対応させる。Code Gear が実行されるとそれに対応する Meta Code Gear が実行され、Meta Computation が行われる。Meta Computation は OS が行うネットワーク管理、メモリ管理等の資源制御を行う。本論文では基本的な機能を設計し、CbC(Continuation based C) で実装する。

Design of Gears OS with Meta Computation based on Monad

Shohei KOKUBO ,[†1] Tatsuki Iha [†2] and Shinji KONO [†2]

We are developing parallel framework using a Code/Data Gear. Code/Data Gear are unit of processing and data. Meta function for parallel execution based on a Monad in Functional Language is used in Geas OS. A Meta Code/Data Gear attached to a Code/Data Gear as a Monad. Meta Computation performs Network Management, Memory Management and more. We show same implemetation of Gears OS using CbC(Continuation based C).

1. Cerium と Alice

本研究室では並列プログラミングフレームワーク Cerium[1] と分散ネットフレームワーク Alice[2] の開発を行なってきた。

Cerium と Alice を開発して得られた知見から Inherent Parallel, Distributed Open Computation をキーワードとして並列分散フレームワーク Gears OS の設計・開発を行う。

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実現する。依存関係はプログラマ自身が意識して記述する必要がある。Task の種類が増えると記述が煩雑になり、プログラマの負担が大きくなる。Task の依存関係がデータの依存関係を正しく保証しない場合があるという問題がある。また、Task の取り扱うデータには型情報がない。汎用ポインタをキャストして利用するしかなく、型の検査が行われていない。Cerium は C++ で実装されているが、オブジェクトと並列処理が直接対応していないのでオブジェクト指向で記述する利点が少ない。Cell[3], Many Core CPU, GPU といった様々なプロセッサをサポートしている。しかし、それぞれの環境でプログラムを高速に動作させるためにはそれぞれに合わせた実行機構を必要としている。

Alice は本研究室で開発を行なっている分散管理フレームワークである。Alice では処理の単位である Code Segment, データの単位である Data Segment を用いてプログラムを記述[4] する。Code Segment 使用する Input Data Segment, Output Data Segment を指定することで処理とデータの関係を記述する。Alice は Java で実装されており、実行速度が遅いという問題がある。また、Data Segment にアクセスする API のシンタックスが特殊で Alice を用いてプログラムを作成するためには慣れが必要になる。

2. Gears OS

Cerium と Alice の例題から Code の単位だけでなく、Data の単位も必要であることがわかった。Gears OS では Gear という単位を用いてプログラムを Code Gear, Data Gear に細かく分割する。Code Gear は Input Data Gear から Output Data Gear を生成する。Input と Output の関係から Code Gear 同士の依存関係を解決し、並列実行するフレームワークの開発を行う。Code Gear はそれに接続された Data Gear のみを扱う。Code/Data Gear 同士の関係は Meta

†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.

Code/Data Gear によって表現される。この Meta Code/Data Gear を用いることで機能やデータ自体を拡張することができる。

Cerium は初め Cell 向けのフレームワークとして設計されたという経緯からプロセッサ毎の実行機構が異なる。Gears OS では Many Core CPU, GPU をはじめとする様々なプロセッサを同等な実行機構でサポートする。

本研究室で開発している CbC(Continuation based C)[5] を用いて、Gears OS を実装する。CbC はプログラムを Code Segment, Data Segment という単位で記述する。CbC において Code Segment 間の処理の移動は function call ではなく、goto を用いた軽量継続を用いる。CbC のコンパイルには LLVM をバックエンドとしたコンパイラ[6] を用いる。

従来の OS が行う排他制御、メモリ管理、並列実行などは Meta Computation に相当する。関数型言語では Meta Computation に Monad を用いる手法[7] がある。Gears OS では、Meta Code/Data Gear を Monad として定義し、Meta Computation を実現する。

Gears OS では並列実行をサポートするだけでなく、信頼性も確保する。そのために Gears OS を用いて作成されたプログラムに対する Model Checking を行う機能[8] を提供する。並列プログラムに Model Checking を行うことでそのプログラムがとり得る状態を列挙する。これにより、並列実行時のデッドロックの検出などを行うことでプログラムの信頼性を確保する。Model Checking も Meta Code/Data Gear を用いて実現する。

Gears OS は Many Core CPU, GPU といった並列実行環境に合わせた設計・実装を行う。また、接続する Gear を変更することでプログラムの振る舞いを変更することを可能にする柔軟性、Monad に基づくメタ計算による並行制御、Model Checking を用いた信頼性の確保を目的とする。Gears OS の構成は図 1 の通りである。

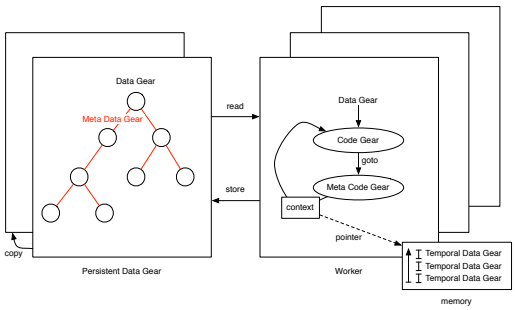

図 1 Gears OS の構成

3. Monad とメタ計算

関数型言語では入力から出力を得る通常の計算の他にメタ計算と呼ばれるものがある。メタ計算の例として、失敗する可能性がある計算、並行処理、入出力などの副作用、メモリ管理などがある。メタ計算の理論的な表現として、Monad を用いることが Moggi らにより提案[7] されている。Gears OS ではメタ計算を表現するのに、Monad と軽量継続を用いる。

Monad は関数が返す通常の値を含むデータ構造であり、メタ計算を表現するのに必要な情報を格納している。失敗する可能性がある計算の場合は、計算が失敗したかどうかのフラグが Monad に含まれる。並行処理の場合は、Monad は可能な計算の interleaving(並び替え) になるが、実際に並び替えを持っているわけではなく、マルチプロセッサで実行する環境そのものが Monad に対応する。

通常の関数を Monad を返すように変更することにより、メタ関数が得られる。逆に Monad の中にある通常の戻り値のみに着目すると通常の関数に戻る。このように、Monad を用いたメタ計算の表現では通常の計算とメタ計算が一対一に対応する。

一般的には複数の Monad の組み合わせが Monad になることを示すのは難しい。Gears OS では Code と Data を分離して、Code から他の Code への呼び出しを継続を用いて行う。Gears OS での Monad は Meta Code と Meta Data になる。

4. Code Gear と Data Gear

Gears OS ではプログラムの実行単位として様々な Gear を使う。Gear が平行実行の単位、データ分割、Gear 間の接続などになる。

Code Gear はプログラムの実行コードそのものであり、OpenCL[9]/CUDA[10] の kernel に相当する。Code Gear は複数の Data Gear を参照し、一つまたは複数の Data Gear に書き込む。Code Gear は接続された Data Gear 以外には触らない。Code Gear はサブルーチンコールではないので、呼び出し元に戻る概念はない。その代わりに、次に実行する Code Gear を指定する機能 (軽量継続) を持つ。

Data Gear には、int や文字列などの Primitive Data Type が入る。自分が持っていない Code Gear, Data Gear は名前で指し示す。

Gear の特徴の一つはその処理が Code Gear, Data Gear に閉じていることにある。これにより、Code Gear の実行時間、メモリ使用量を予測可能なものにする。

Code Gear, Data Gear はポインタを直接には扱わない。これにより、Code と Data の分離性を上げて、ポインタ関連のセキュリティフローを防止する。

Code Gear, Data Gear はそれぞれ関係を持っている。例えば、ある Code Gear の次に実行される Code Gear、全体で木構造を持つ Data Gear などである。Gear の関連付けは Meta Gear を通して行う。Meta Gear は、いままでの OS におけるライブラリや内部のデータ構造に相当する。なので、Meta Gear は Code Gear, Data Gear へのポインタを持っている。

5. 継　　続

ある Code Gear から継続するときには、次に実行する Code Gear を名前で指定する。Meta Code Gear が名前を解釈して、処理を対応する Code Gear に引き渡す。これらは、従来の OS の Dynamic Loading Library や Command 呼び出しに対応する。名前と Code Gear へのポインタの対応は Meta Data Gear に格納される。この Meta Data Gear を Context と呼ぶことにする。これは従来の OS の Process や Thread に対応する。

Context には以下のようなものが格納される。

- Code Gear の名前とポインタの対応表
- Data Gear の Allocation 用の情報
- Code Gear が参照する Data Gear へのポインタ
- Data Gear に格納される Data Type の情報

```
/* Context definition */

#define ALLOCATE_SIZE 1024

enum Code {
    Code1,
    Code2,
    Allocator,
};

enum UniqueData {
    Allocate,
    Tree,
};

struct Context {
    int codeNum;
    __code (**code) (struct Context *);
    void* heap_start;
    void* heap;
    long dataSize;
    int dataNum;
    union Data **data;
};

union Data {
    struct Tree {
        union Data* root;
        union Data* current;
        union Data* prev;
        int result;
    } tree;
    struct Node {
        int key;
        int value;
        enum Color {
            Red,
            Black,
        } color;
        union Data* left;
        union Data* right;
    } node;
    struct Allocate {
        long size;
        enum Code next;
    } allocate;
};
```

ソースコード 1　Context

Code Gear の名前とポインタの対応表

Code Gear の名前とポインタの対応は enum と関数ポインタによって表現される。これにより、実行時に比較ルーチンなどを動的に変更することが可能になる。

Data Gear の Allocation 用の情報

Context の生成時にある程度の領域を確保する。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。そのポインタを必要な Data Gear のサイズに応じて、インクリメントすることによって Data Gear の Allocation を実現する。

Code Gear が参照する Data Gear へのポインタ

Context には 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 union と 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。この情報から確保する Data Gear のサイズなどを決定する。

```
#include <stdlib.h>

#include "context.h"

extern __code code1(struct Context*);
extern __code code2(struct Context*);
extern __code allocate(struct Context*);

__code initContext(struct Context* context) {
    context->dataSize = sizeof(union Data)*
          ALLOCATE_SIZE;
    context->code = malloc(sizeof(__code*)*
          ALLOCATE_SIZE);
    context->data = malloc(sizeof(union Data*)*
          ALLOCATE_SIZE);
    context->heap_start = malloc(context->dataSize);
    context->heap = context->heap_start;

    context->codeNum = 3;
    context->code[Code1] = code1;
    context->code[Code2] = code2;
    context->code[Allocator] = allocate;

    context->dataNum = 2;
    context->data[Allocate] = context->heap;
    context->heap += sizeof(struct Allocate);
    context->data[Tree] = context->heap;
    context->heap += sizeof(struct Tree);

    context->root = 0;
```

```
    context->current = 0;
}
```

ソースコード 2　initContext

6. Persistent Data Gear

Data Gear の管理には木構造を用いる。この木構造は非破壊で構築される。非破壊的木構造では、図 2 のように編集元の木構造を破壊することなく新しい木構造を構成する。破壊的木構造と異なりロックの必要がなく、平行して読み書き、参照を行うことが可能である。また、変更前の木構造をすべて保持しているので過去のデータにアクセスすることができる。

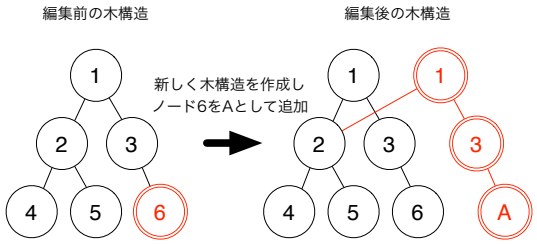

図 2　木構造の非破壊的編集

7. Allocator

Gears OS では Context の生成時にある程度の領域を確保し、その領域を指すポインタをインクリメントすることで Allocation を実現する。

Context には Allocation 用の Data Gear が格納されている。この Data Gear に確保するサイズと確保後に接続する Code Gear の名前を書き込み、Allocation を行う Code Gear に接続することで必要な領域を確保する。

```
// Code Gear
__code code1(struct Context* context) {
    context->data[Allocate]->allocate.size = sizeof(
        struct Node);
    context->data[Allocate]->allocate.next = Code2;
    goto meta(context, Allocate);
}

// Meta Code Gear
__code meta(struct Context* context, enum Code next)
    {
    // meta computation
    goto (context->code[next])(context);
}

// Meta Code Gear
__code allocate(struct Context* context) {
    context->data[++context->dataNum] = context->heap
        ;
    context->heap += context->data[Allocate]->
        allocate.size;
    goto (context->code[context->data[Allocate]->
        allocate.next])(context);
}
```

```
// Code Gear
__code code2(struct Context* context) {
    // processing content
}
```

ソースコード 3　Allocator

ソースコード 3 では Code Gear である code1 でポインタを扱っており、Code Gear でポインタを扱わないという設計思想に合っていない。そこで、ソースコード 4 をソースコード 3 として解釈するようにコンパイラを改良する。

```
// Code Gear
__code code1(Allocate allocate) {
    allocate.size = sizeof(long);
    allocate.next = Code2;
    goto Allocate(allocate); // goes through meta
}

// Meta Code Gear
__code meta(struct Context* context, enum Code next)
    {
    // meta computation
    goto (context->code[next])(context);
}

// Meta Code Gear
__code allocate(struct Context* context) {
    context->data[++context->dataNum] = context->heap
        ;
    context->heap += context->data[Allocate]->
        allocate.size;
    goto (context->code[context->data[Allocate]->
        allocate.next])(context);
}

// Code Gear
__code code2(Allocate allocate, Count count) {
    // processing
}
```

ソースコード 4　SyntaxSugar

8. List

通常 List は要素と次へのポインタを持つ構造体で表現される。Gears OS の場合、Meta レベル以外でポインタは扱わないので図 3 のように任意の要素を持つ Data Gear と次へのポインタを持つ Meta Data Gear の組によって List は表現される。

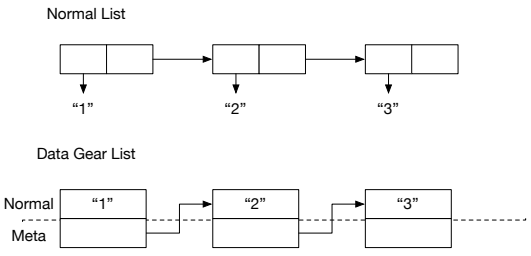

図 3　List の表現

9. Synchronized Queue

Gears OS では List を表現する Code/Data Gear に CAS(Compare and Swap) を行う Meta Code/-Data Gear を接続することで Synchronized Queue を実現する。Gears OS の機能は状態遷移図とクラスダイアグラムを組み合わせた図で表現する。この図を GearBox と呼ぶことにする。図 4 は Synchronized Queue の GearBox である。M:receiver/sender が CAS を行う Meta Code Gear となる。

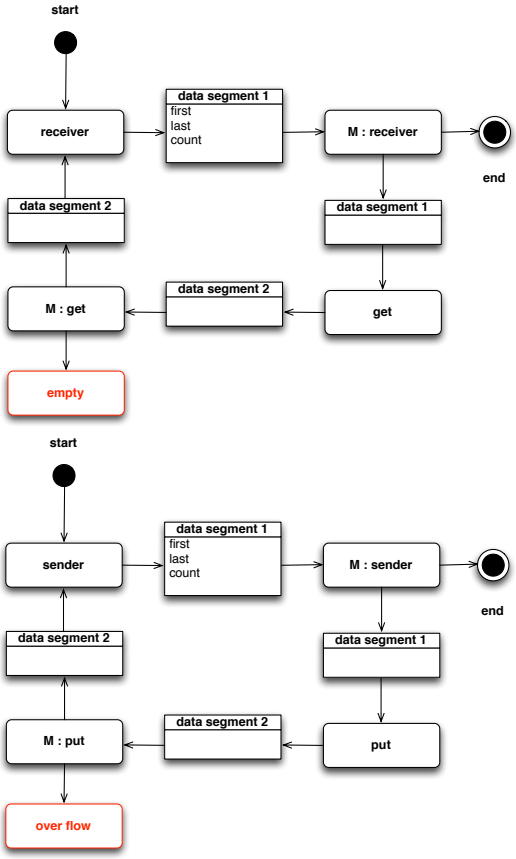

図 4 Synchronized Queue

10. 比　　較

Cerium/Alice, OpenCL/CUDA, 従来の OS との比較を以下に示す。

Cerium/Alice

Gears OS の Code Gear は Cerium の Task, Context は HTask に相当する。Cerium とは異なり、Gears OS は処理とデータが分離している。Gears OS では分離したデータを Data Gear と呼称する。これは Alice の Data Segment と同等のものである。Gears OS では Alice と同様に Code と Data の関係から依存関係を解決する。

Alice は Data Segment を MessagePack[11] を利用して通信することで分散実行を実現する。Gears OS 上での分散実行も Alice に沿って設計・実装する。

OpenCL/CUDA

Code Gear は OpenCL/CUDA の kernel に相当する。OpenCL/CUDA には Data Gear に相当する仕組みはない。接続された複数の Code Gear は接続された順番通りに実行される。これは、OpenCL の CommandQueue, CUDA の Stream と同等のものである。OpenCL/CUDA では kernel の依存関係を複雑に記述する必要があるが、Gears OS では Code と Data の関係から自動的に依存関係を解決する。

従来の OS

従来の OS が行な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 Gear, Meta Data Gear を用いて行う。このメタ計算は Monad に基づいて実現される。

11. ま　と　め

Gears OS は Inherent Parallel をキーワードとして、Gears OS 上で実行されるプログラムが自動的に並列で処理されるように設計した。Gear を他の Gear に接続することで機能およびデータの拡張を行える柔軟性を持つ。Meta な機能や並行制御を関数型言語における Monad に基づいて実現する。また、機能として Model Checking を持ち、Gears OS 上で実行されるプログラムの信頼性を保証する。本論文では必要な機能の一部である Context, Allocator, List, Non-Destructive Red-Black Tree を CbC を用いて実装した。今後は、Synchronized Queue, Woker を実装する予定である。これにより、Cerium と同等の例題を動かすことが可能となる。例題としては Bitonic Sort, Word Count を予定している。例題が動くことを確認し次第、Gears OS の測定・評価を行う。

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) 赤嶺一樹, 河野真治：DataSegment API を用いた分散フレームワークの設計, 日本ソフトウェア科学会第 28 回大会論文集 (2011).
3) Sony Corporation: Cell broadband engine architecture (2005).
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) 河野真治, 島袋　仁：C with Continuation と、

その PlayStation への応用，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) (2000).
6) 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).
7) Moggi, E.: Computational lambda-calculus and monads, *Proceedings of the Fourth Annual Symposium on Logic in computer science* (1989).
8) 下地篤樹，河野真治：線形時相論理による Continuation based C プログラムの検証，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) (2007).
9) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
10) : CUDA, https://developer.nvidia.com/category/zone/cuda-zone/.
11) : MessagePack, http://msgpack.org/.