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 Cerium を開発して得られた知見から Code Segment と Data Segment を用いた並列フレームワークの開発を行なっている。Code Segment と Data Segment は処理とデータの単位である。今回設計した Gears OS ではプログラムを Code Segment と Data Segment で記述する。Code Segment と Data Segment で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を可能にする。本論文では Gears OS の基本的な機能を設計し、CbC(Continuation based C) を用いて実装する。

Abstract

目 次

図 目 次

表 目 次

第1章　並列分散環境下におけるプログラミング

第2章　並列プログラミングフレームワーク Cerium

Cerium は PlayStation 3(PS3) に搭載された Cell Broadband Engine(Cell) 向けの Fine-Grain TaskManager として当研究室で設計・開発されたフレームワークである。本章では Cerium の実装について説明する。

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。

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

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。

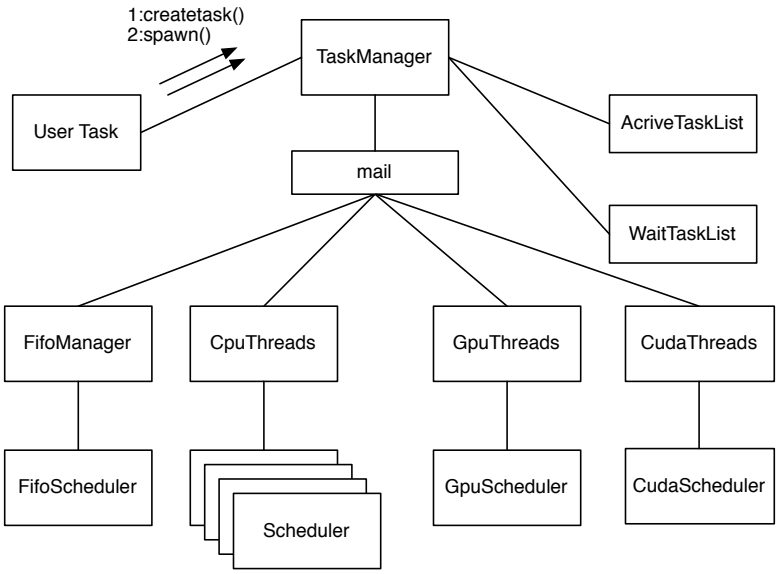

図 2.1: TaskManager

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。

```
static int
twice(SchedTask *s,void *rbuf, void *wbuf)
{
    /**
     * Get Input Data
     * get_input(input data buffer, index)
     */
    int* input = (int*)s->get_input(rbuf, 0);

    /**
     * Get Output Data
     * get_output(output data buffer, index)
     */
    int* output = (int*)s->get_output(wbuf, 0);

    /**
     * Get index(x, y, z)
     * SchedTask member
     * x : SchedTask->x
     * y : SchedTask->y
     * z : SchedTask->z
     */
    long i = s->x;

    output[i] = input[i]*2;

    return 0;
}
```

ソースコード 2.2: 実行される Task

```
__global__ void
twice(int* input, int* output)
{
    /**
     * Get index(x, y, z)
     * kernel built-in variables
     * x : blockIdx.x * blockDim.x + threadIdx.x
     * y : blockIdx.y * blockDim.y + threadIdx.y
     * z : blockIdx.z * blockDim.z + threadIdx.z
     */
    long i = blockIdx.x * blockDim.x + threadIdx.x;

    output[i] = input[i]*2;

    return 0;
}
```

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。

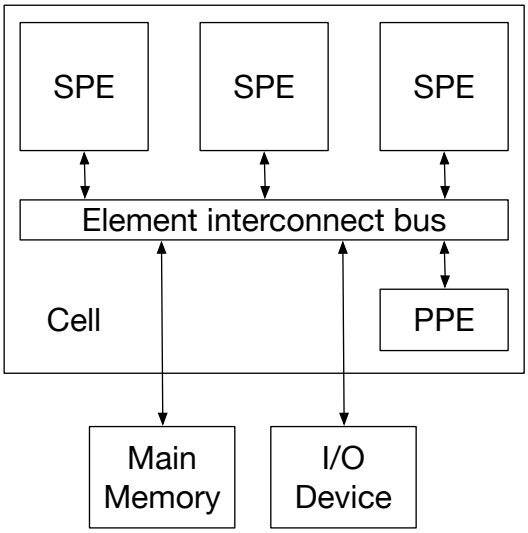

図 2.2: Cell Architecture

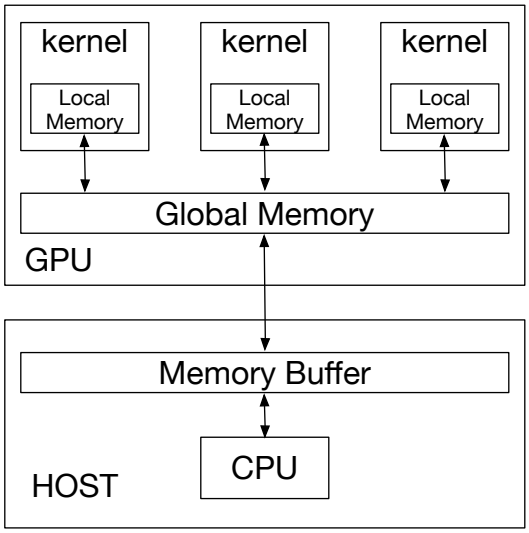

図 2.3: GPU Architecture

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。

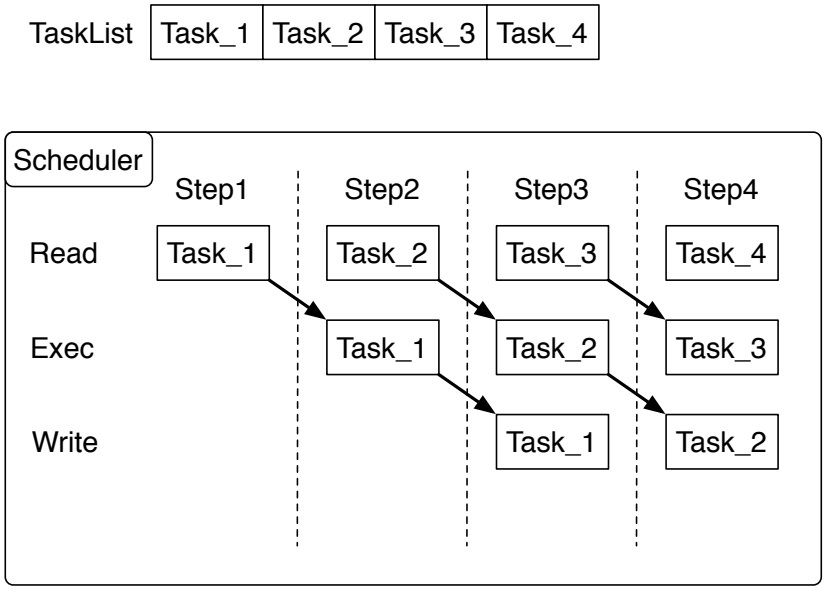

図 2.4: Scheduler

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。

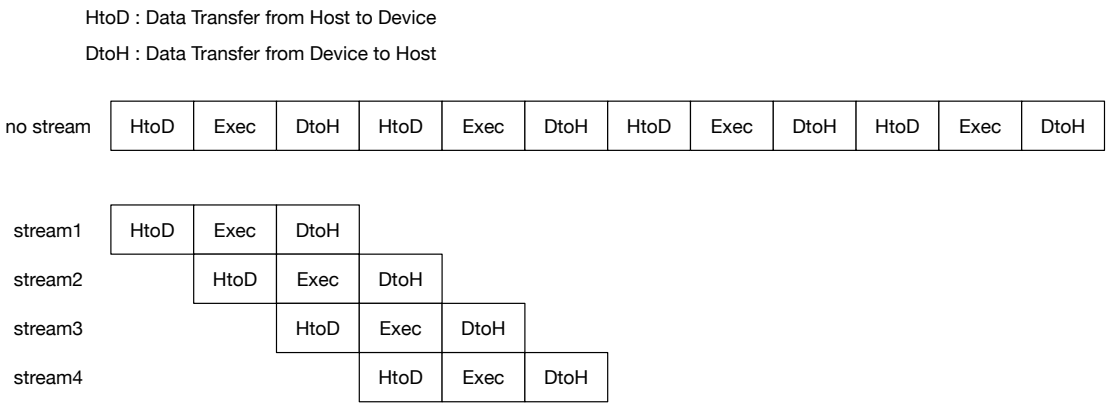

図 2.5: Overlap Data Transfer

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。

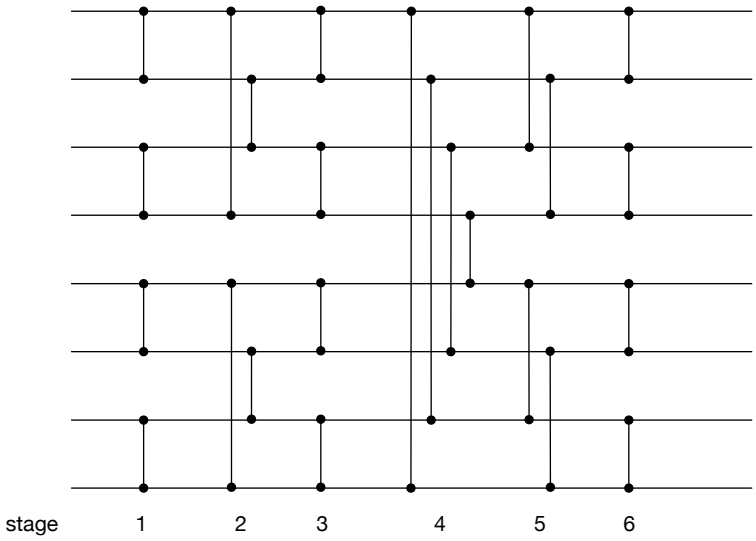

図 2.6: Sorting Network : bitonic sort

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

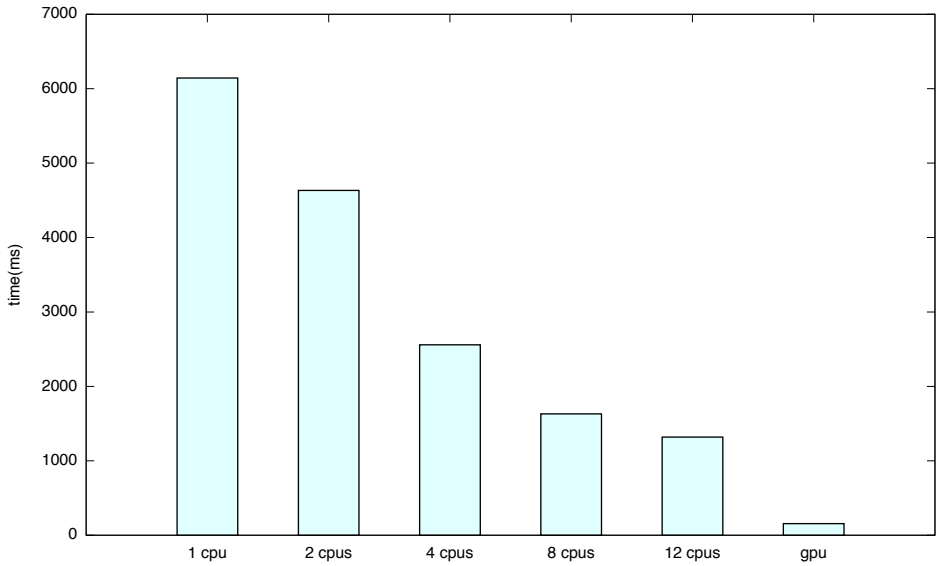

図 2.7: 要素数 2^{20} に対するソート

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。

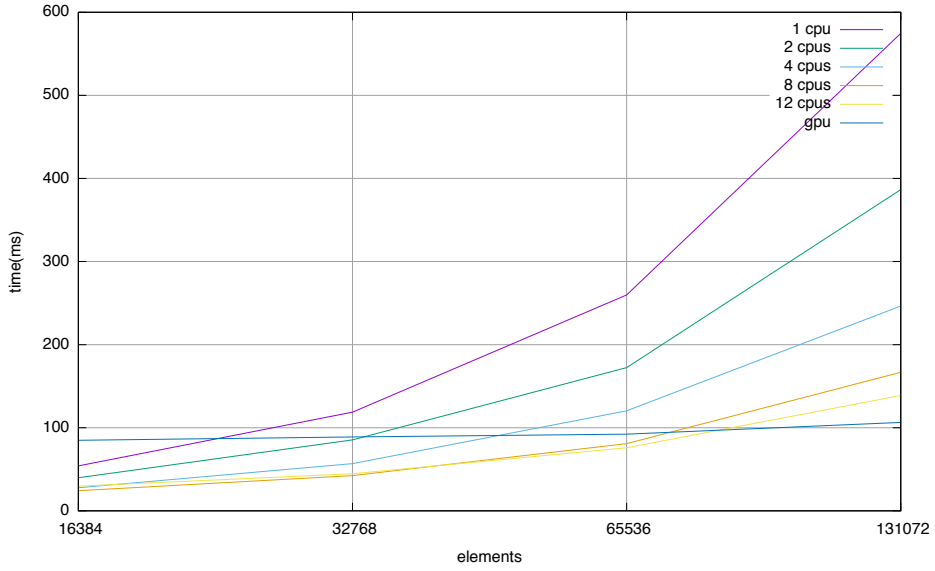

図 2.8: Bitonic Sort(from 2^{14} to 2^{17})

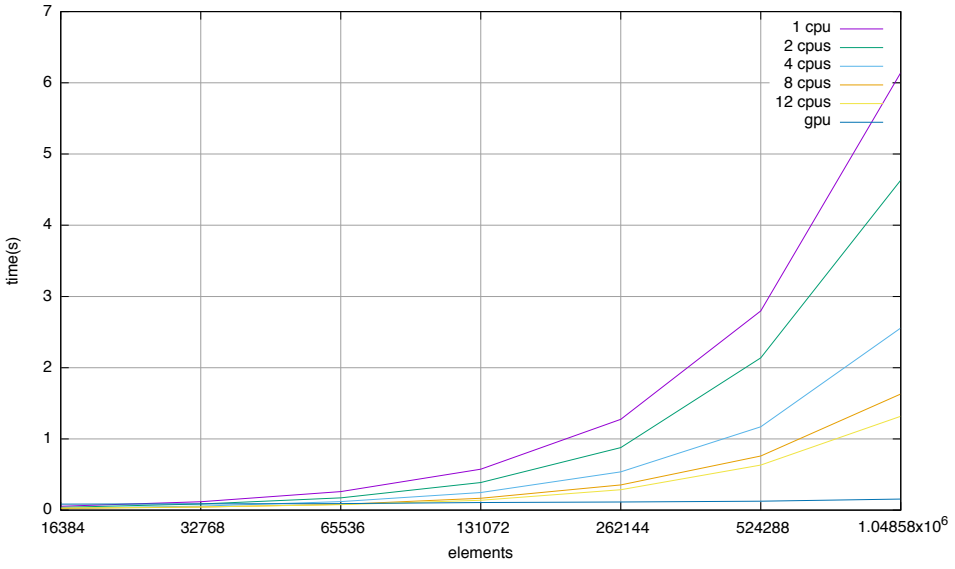

図 2.9: Bitonic Sort(from 2^{14} to 2^{20})

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。

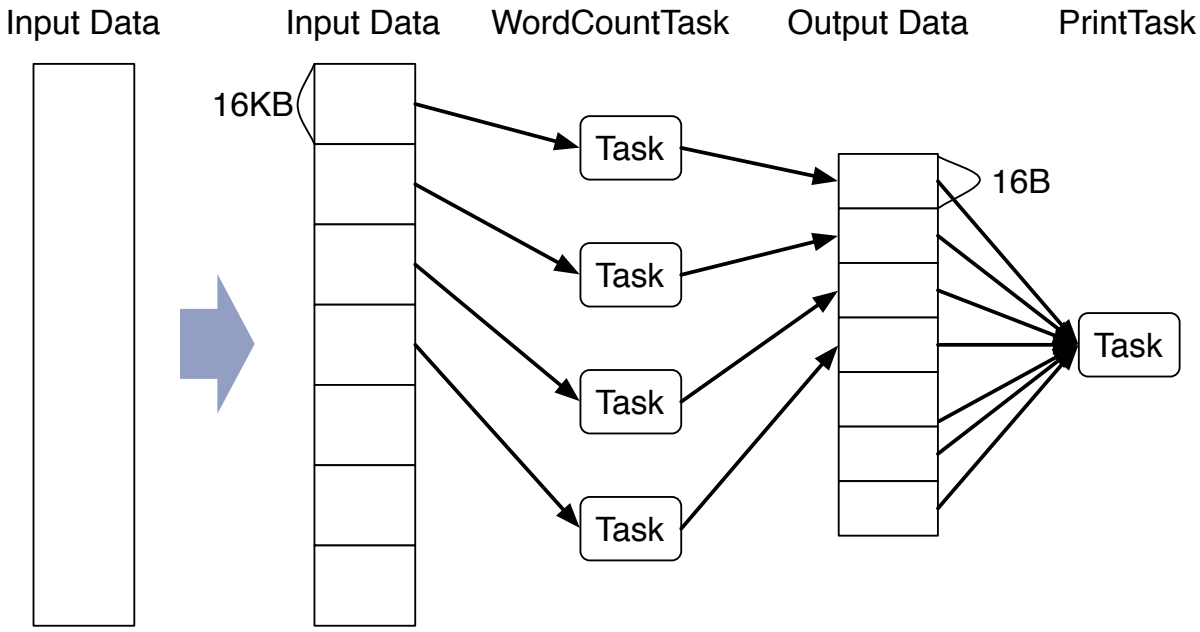

図 2.10: Word Count の流れ

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

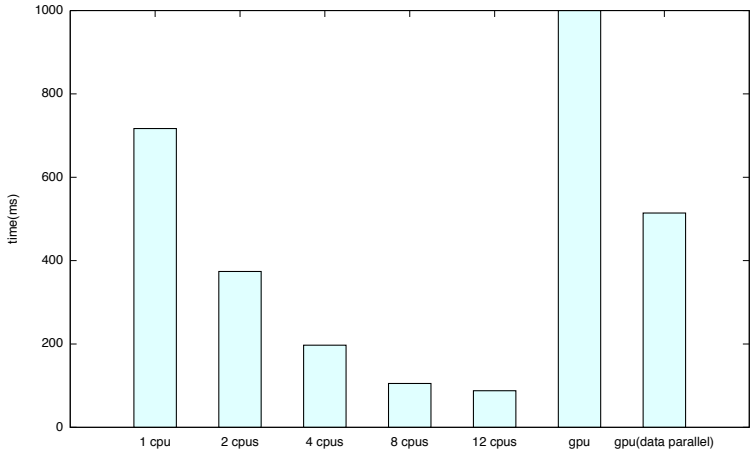

図 2.11: 100MB のテキストデータに対する WordCount

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

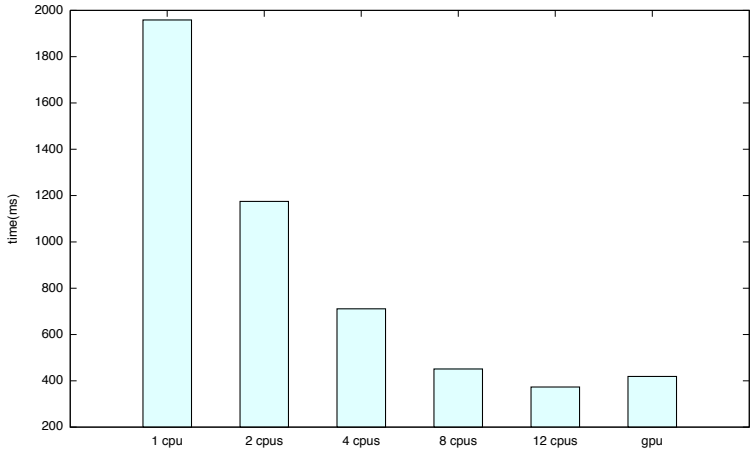

図 2.12: 1MB の画像データに対する FFT

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な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

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。また、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。

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。

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

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。

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

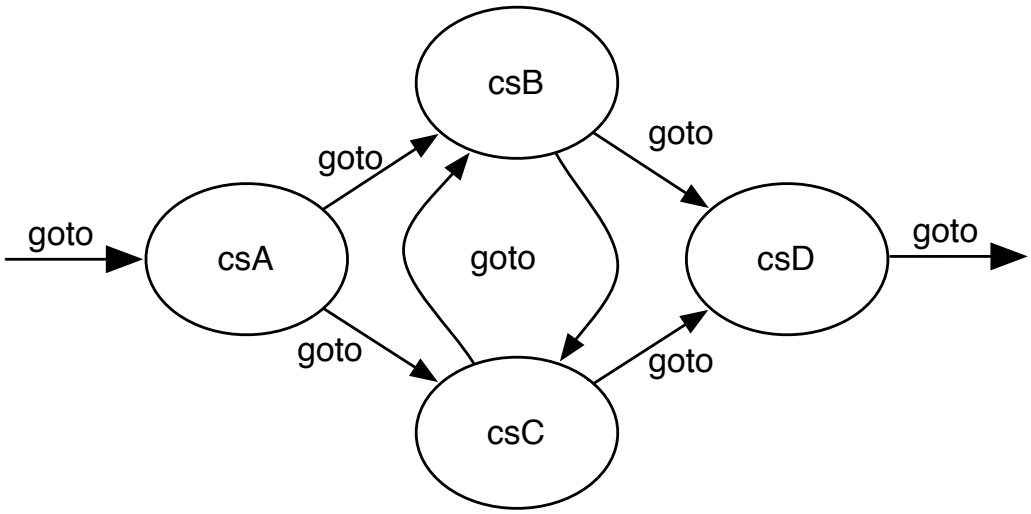

図 3.1: goto による Code Segment 間の継続

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。省略形のソースコード:3.1 から実際にコンパイルされるソースコード: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: 省略形

```
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: 変換後

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。

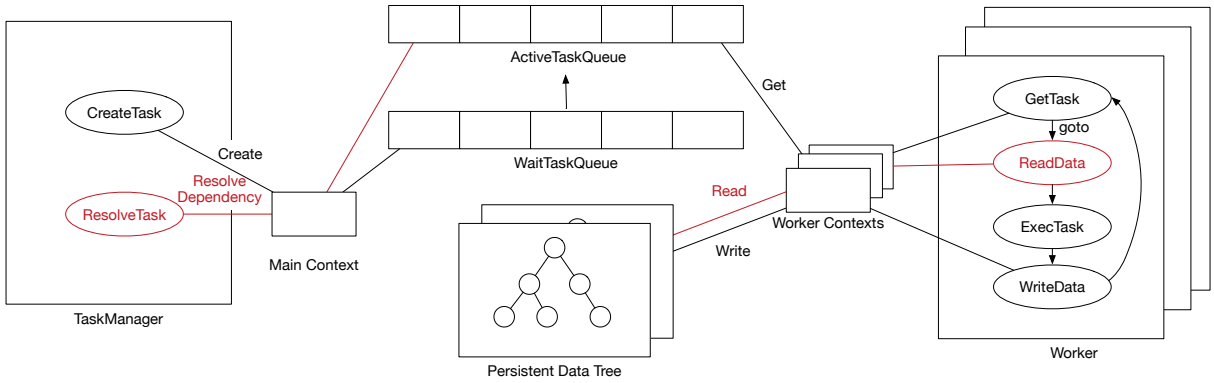

図 4.1: Gears OS

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。

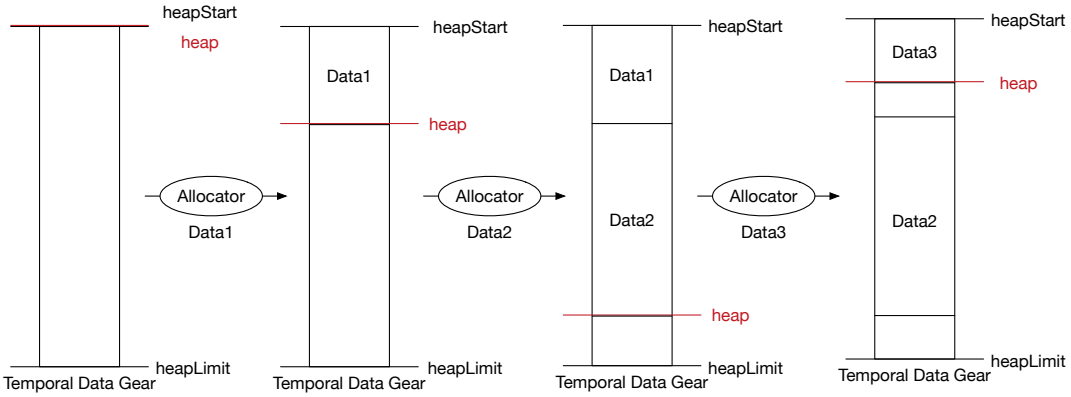

図 4.2: Allocation

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。

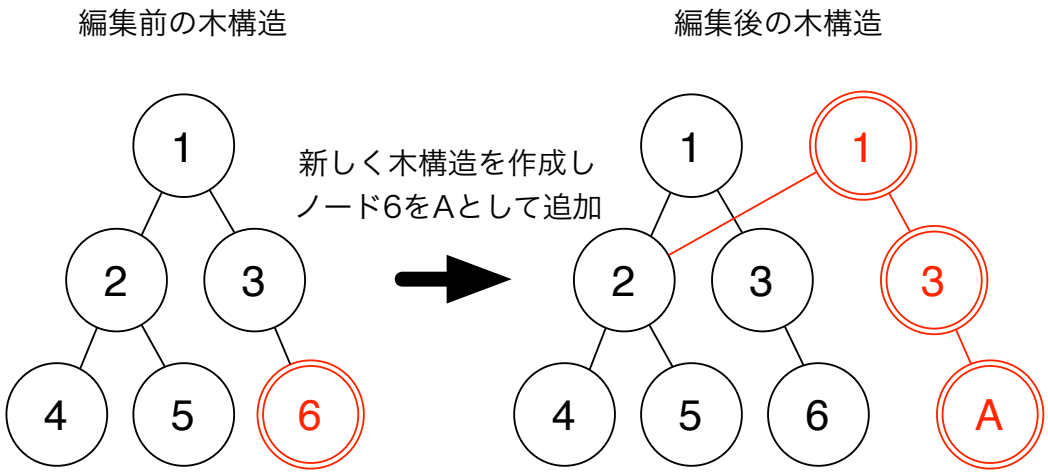

図 4.3: 木構造の非破壊的編集

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

この章では今回設計・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

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。

この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。

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 のデータに対してコア数・1 Task 当たりの仕事量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	64 Tasks(ms)	640 Tasks(ms)	6400 Tasks(ms)
1 CPU	1245	1315	1973
2 CPUs	629	689	1118
4 CPUs	326	366	610
8 CPUs	165	189	327
12 CPUs	121	111	114

表 6.1: 要素数 2^{17}*1000 のデータに対する Twice

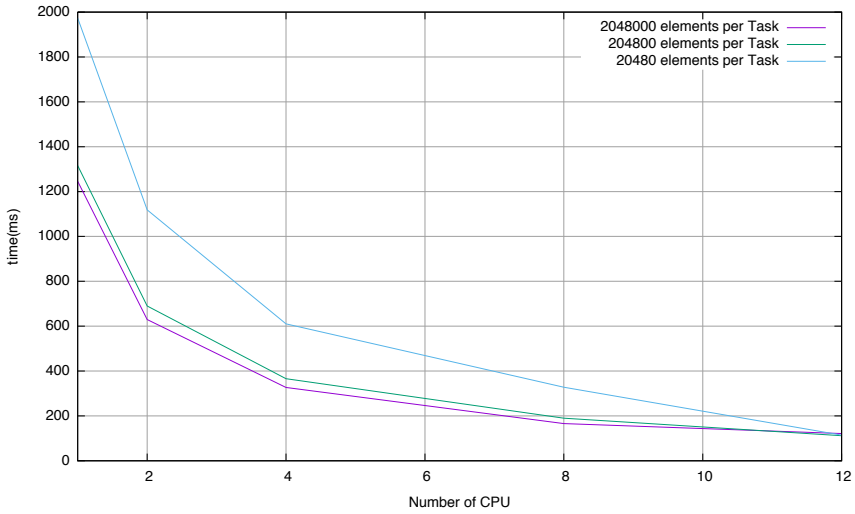

図 6.1: 要素数 2^{17}*1000 のデータに対する Twice

第7章　結論

謝辞

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), April 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, Sep 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, Jan 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), May 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), May 2014.

[7] Eugenio Moggi. 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), April 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/.