マルチプラットフォーム対応
並列プログラミングフレームワーク

平成26年度 3月 学位論文(修士)

琉球大学大学院　理工学研究科
情報工学専攻

渡真利 勇飛

要 旨

Abstract

目 次

図 目 次

表 目 次

第1章　研究目的と背景

PCやタブレットの一般的な利用方法として動画の編集や再生、ゲームといったアプリケーションの利用が挙げられる。これらのアプリケーションはグラフィックや物理演算等、要求する処理性能が上がってきている。

しかし消費電力や発熱、クロックの限界から CPU の性能自体を上げることによる処理性能の向上は不可能となっている。その事からプロセッサメーカーはマルチコアやヘテロジニアス構成の路線を打ち出している。そういったアーキテクチャ上でリソースを有効活用するにはプログラムの並列化は必須と言える。

しかしプログラムを並列化するのみではリソースの有効活用としては不充分であり、実行の順番やタスクをどのリソース上で実行するかといった Scheduling も行わなければならない。こういった部分をサポートするプログラミングフレームワークが必要である。

当研究室では Cerium というプログラミングフレームワークを開発している。Cerium をマルチプラットフォームに対応させ、高い並列度を維持したプログラミングを可能にする。本研究ではマルチプラットフォーム上でのプログラムの実行における最適化について、Sort 、Word Count 、FFT を例題に考察していく。

1.1　本論文の構成

第2章　既存のマルチプラットフォームフレームワーク

マルチプラットフォームでプログラムを動作させる場合、そのアーキテクチャを意識する必要がある。マルチプラットフォームにはマルチコア CPU 、 GPU や Cell といったヘテロジニアスマルチコアのような様々な構成がある。

2.1　Architecutre

本研究では、 CPU の他に GPU 上でのプログラミング (GPGPU) にも対応する。

GPU(Graphics Processing Unit) は PC の画像処理を担当するユニットで、レンダリングに特化したプロセッサが多く集まった構造を持つ。一つ一つのプロセッサの構造は単純で、その機能は CPU に比べて限定的ではあるが大量のデータを複数のプロセッサで並列処理することに長けている。

GPGPU (General Purpose computing on Graphics Processing Units) とは、GPU の高い演算能力を画像処理ではなく汎用計算に使用することである。

2.2　Shared Memory

計算機にはメモリ空間が別の計算機と、共有メモリ (Shared Memory) な計算機がある。GPU のメモリ空間 (図:2.1) はマルチコア CPU (図:2.2) と違い、共有メモリ (shared mermoy) でないので Host と Device 間で Data の共有ができない。そのためマルチプラットフォーム環境に対応したフレームワークには、Device と Host 間でデータの転送を行う API 備わっている。しかし、異なる Device 間でデータの転送を行うとネックになる。そのためデータの入出力を行う回数を減らす、入出力の処理をパイプライン処理にするなどの工夫が必要になる。

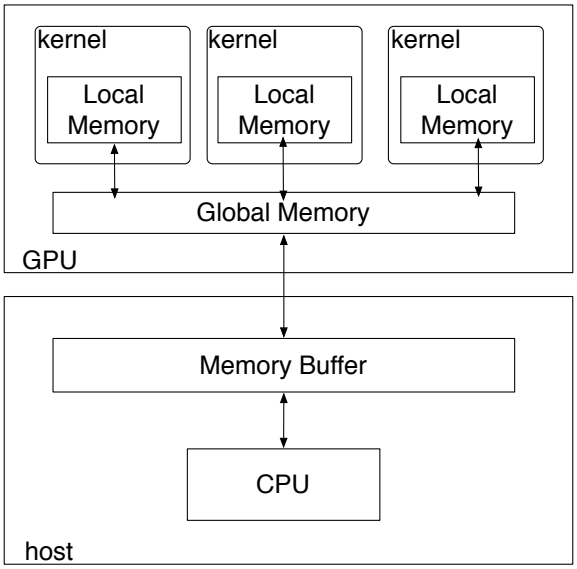

図 2.1: GPU Architecture

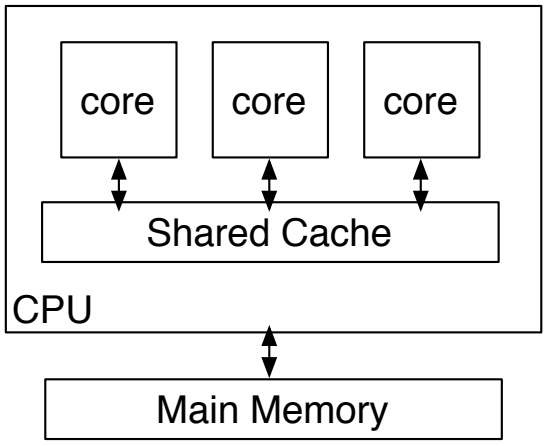

図 2.2: CPU Architecture

2.3 OpenCL

OpenCL とは、 Khronos Group の提供するマルチコア CPU と GPU といった、ヘテロジニアス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。

OpenCL では演算用プロセッサ側を Device 、制御用デバイス側を Host として定義する。また、 Device 上で動作するプログラムの事を kernel と呼ぶ。

OpenCL では、デバイスの操作に Command Queue を使用する。Command Queue は Device に命令を送るための仕組みである。Command Queue は clCreateCommandQueue という OpenCL API で作成され、Command Queue が所属するコンテキストや実行対象となる Device を指定する。

kernel の実行、input data への書き込み、 output data の読み込みといったメモリ操作はこの Command Queue を通して行われる。

OpenCL には主に 2 つの仕様がある。

- OpenCL C 言語
- OpenCL Runtime API

OpenCL C は演算用プロセッサ上で動作する、 C 言語を拡張したプログラミング言語である。一方で OpenCL Runtime API は OpenCL C で記述した kernel を Queuing するために Host が利用する API である。

Host では主に Data を input/output するメモリ資源の確保を行う。OpenCL は host 側で memory buffer を作成してメモリのコピーを行う。これらの処理や Task は Command Queue に enqueue することで実行される。

多次元のデータ構造を扱う計算において高い並列度を保つには、多次元データを分割して並列に実行する機能が必要である。これをデータ並列実行という。 OpenCL はデータ並列実行もサポートしている。OpenCL は次元数に対応する index があり、OpenCL は一つの記述から異なる index を持つ複数の kernel を自動生成する。その添字を global_id と呼ぶ。この時入力されたデータはワークアイテムという処理単位に分割される。

OpenCL はワークアイテム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 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い、データ並列を実現する。この ID によって取得してきたワークアイテムをグローバルワークアイテムという。また、ワークアイテムは 3 次元までのデータを渡すことができる。

データ並列による kernel 実行の場合は clEnqueueNDRangeKernel API を使用するが、この関数の引数としてワークアイテムのサイズと次元数を指定することでデータ並列で実行できる。

前節でワークアイテムという処理単位について述べたが、さらに複数個のグローバルワークアイテムを work_group という単位にまとめることができる。work_group 内では同期やローカルメモリの共有が可能となる。

グローバルワークアイテム (ワークアイテム全体) の個数と、ローカルワークアイテム (グループ一つ辺りのアイテム) の個数を指定することでワークアイテムを分割する。なお、このときグローバルワークアイテム数はローカルアイテム数の整数倍でなければ clEnqueueNDRangeKernel API 呼び出しは失敗する。

ローカルアイテム数は 0 を指定することで、コンパイル時に最適化させることができる。したがってローカルアイテムのサイズは 0 を指定するのが一般的である。

なお、 work_group を設定した場合は global_id の他に work_group_id 、local_id がそれぞれの kernel に割り当てられる (図:2.3)。

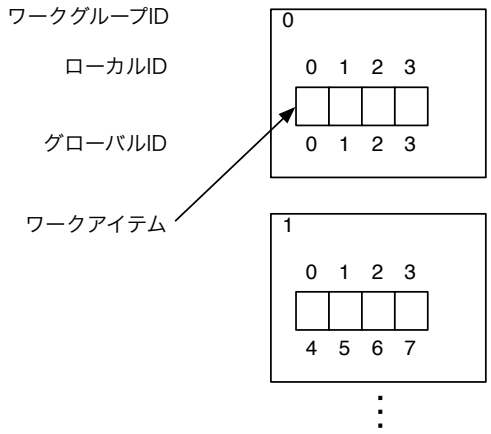

図 2.3: WorkItem ID

なお、work_group を設定した場合は global_id の他に work_group_id、local_id がそれぞれの kernel に割り当てられる (図:2.3)。

kernel 側からそれぞれ ID に対応した API を使用して、各 ID を取得する。取得した ID から自分が担当する index を計算して導く。表:2.1 は kernel 側で使用できる、 ID を取得するための API となる。 なお、 local_id 、global_id を取得する API は引数に 0、

get_group_id	work_group_id を取得
get_local_id	local_id を取得
get_global_id	global_id を取得

表 2.1: kernel で使用する ID 取得の API

1、2 の値を set することができる。id は x, y, z 座標があり、それぞれが 0, 1, 2 に対応している。例えば get_global_id(1) と呼び出した場合は y 座標の、get_global_id(1) と呼び出した場合は z 座標の global_id を取得する。

2.4 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ある。

CUDA も OpenCL と同様、演算用プロセッサ (GPU) を Device 、制御用デバイス側を Host として定義する。また、 Device 上で動作するプログラムの事も kernel と呼ぶ。

OpenCL における Command と CommandQueue に対応するものとして、 CUDA には Operation と 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。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。更に、異なる Stream に発行された Operation も依存関係が存在しない場合、Operation は並列に実行される。更に依存関係が存在しない、異なる Stream に発行された Operation は並列に実行される。

CUDA には主に 3 つの仕様がある。

- CUDA C
- CUDA Runtime API
- CUDA Driver API

CUDA C は GPU 上で動作する、C 言語を拡張したプログラミング言語である。CUDA Runtime API も CUDA Driver API も CUDA C で記述した Kernel を Queueing するために Host が利用する API である。Driver API は Runtime API に比べ、プログラマが管理しなければならないリソースが多くなる代わり、より柔軟な処理を行う事ができる。

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 は全て 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的な処理を行う API を利用する必要がある。

CUDA では OpenCL の WorkItem に相当する単位を thread として定義している。この thread をまとめた単位として block がある。

CUDA でデータ並列による kernel 実行を行う場合、cuLaunchKernelAPI を使用する。この関数は引数として各座標の block 数、各座標の block 1 つ辺りの thread 数を指定することによりデータ並列実行を行う。

cuLaunchKernel で kernel を実行すると各 thread に対して blockID と threadID が割り当てられる。CUDA には OpenCL と異なり、ID を取得する API が存在しない。それに代わり、 kernel に組み込み変数が準備されている。その組み込み変数を参照し、対応するデータに対し処理を行うことでデータ並列実行を実現する。組み込み変数は以下の 3 つである。

- uint3 blockDim
- uint3 blockIdx

- uint3 threadIdx

3 つの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。同じように blockID 、 threadID の x 座標を参照することができる。blockDim.x * blockIdx.x + threadIdx.x とする事で OpenCL における get_global_id(0) で取得できる ID に相当する値を算出する事ができる。

例としてある kernel で get_global_id(0) の値が 8 の時、CUDA では 図 2.4 のように算出する。

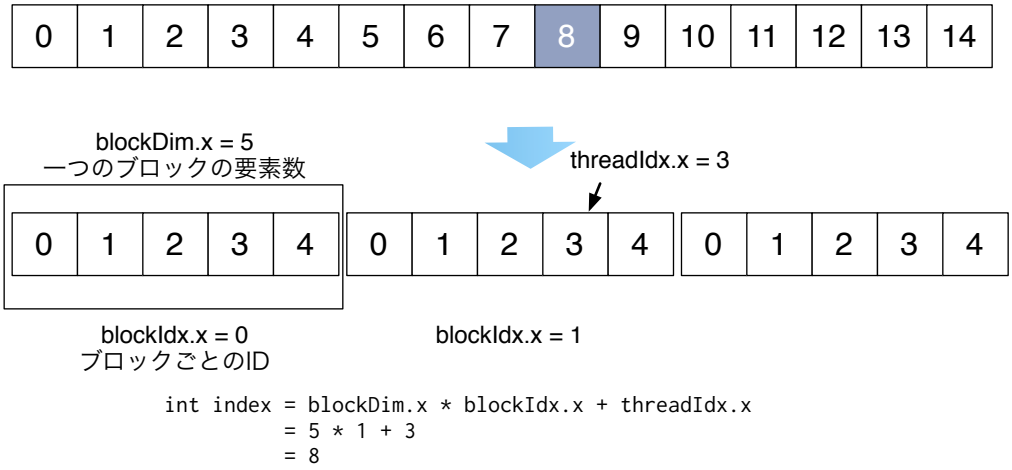

図 2.4: Calculate Index example

2.5 StarPU

StarPU はフランス国立情報学自動制御研究所 (INRIA) の StarPU 開発チームの提供する、ヘテロジニアス環境向けのフレームワークである。GPU の制御に OpenCL と CUDA を用いており、どちらかを選択することで GPU 上で実行することができる。

OpenCL と CUDA における実行の単位は kernel だったが、 StarPU では実行の単位を Task と定義している。

StarPU では Task を制御するために codelet と呼ばれる構造体を使う。codelet を Task 生成時にポインタ渡しすることで、演算を行うリソースや実行する関数等を指定することができる。CPU と GPU で並列に実行する例を 2.1 に示す。

ソースコード 2.1: codelet の例

```
1 starpu_codelet codelet = {
2       .where = STARPU_CPU|STARPU_CUDA,
3       .cpu_func = cpu_function,
4       .cuda_func = cuda_function,
5 };
```

計算に必要なデータは、StarPU のデータプールに登録されている必要がある。StarPU ではデータを starpu_data_handle という型で登録する。Task はこの handle を参照することで値を参照することができる。

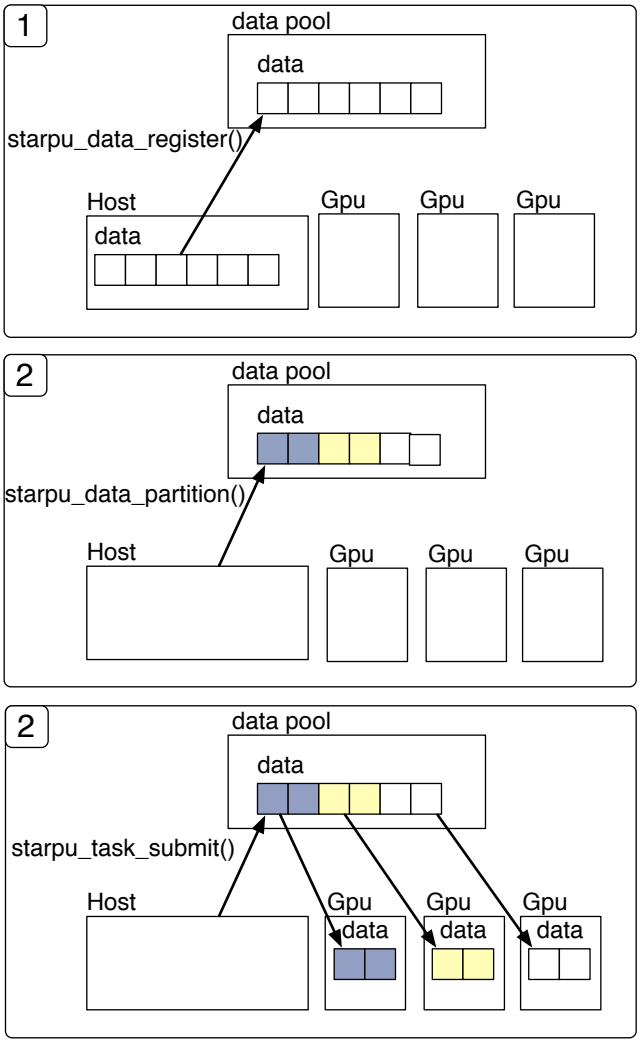

図 2.5: StarPU におけるデータ分割

図:2.5 に StarPU におけるデータ並列実行の流れを示す。StarPU では配列の初期化や代入を行った後、starpu_data_register 関数を使って StarPU のデータプールに登録する。

データ並列で実行する場合、更にデータを分割する必要がある。starpu_data_partition 関数を用いる事で分割を行うことができる。分割数を指定することで、データプールに登録したデータを chunk と呼ばれる単位に分割する。starpu_task_submit 関数により chunk を CPU や GPU に割り当てることができる。

第3章　Cerium

Cerium は、当初 Cell 用の Fine-Grain TaskManager として当研究室で開発された。本章では Cerium の実装について説明する。

3.1　Cerium の概要

Cerium は当初 Cell 用であったが、現在では Linux、 MaxOS X 上で動作する。GPGPU の Data Parallel を含めて同じ形式で記述できる。

Cerium は TaskManager、 SceneGraph、Rendering Engine の 3 つの要素から構成される。本研究では Cerium の TaskManager を汎用計算フレームワークとして改良を行う。これによりヘテロジニアス環境に対応したシステムやフレームワークに必要な API や機構について考察していく。

3.2　Cerium TaskManager

Cerium TaskManager では、処理の単位を Task としてプログラムを記述していく。関数やサブルーチンを Task として扱い、 Task 間の依存関係を考慮しながら実行される。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 data, parameter は関数で言うところの引数に相当する。cpu type は Task が動作する Device を示し、 dependency は他の Task との依存関係を表す。

3.3 Cerium における Task

図:3.1 は Cerium が Task を生成/実行する場合のクラスの構成図である。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。Active-TaskList 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番で実行されても良い。Task は転送を行いやすい TaskList に変換され、cpu type に対応した Scheduler に転送される。なお、転送は Synchronozed Queue である mail を通して行われる。

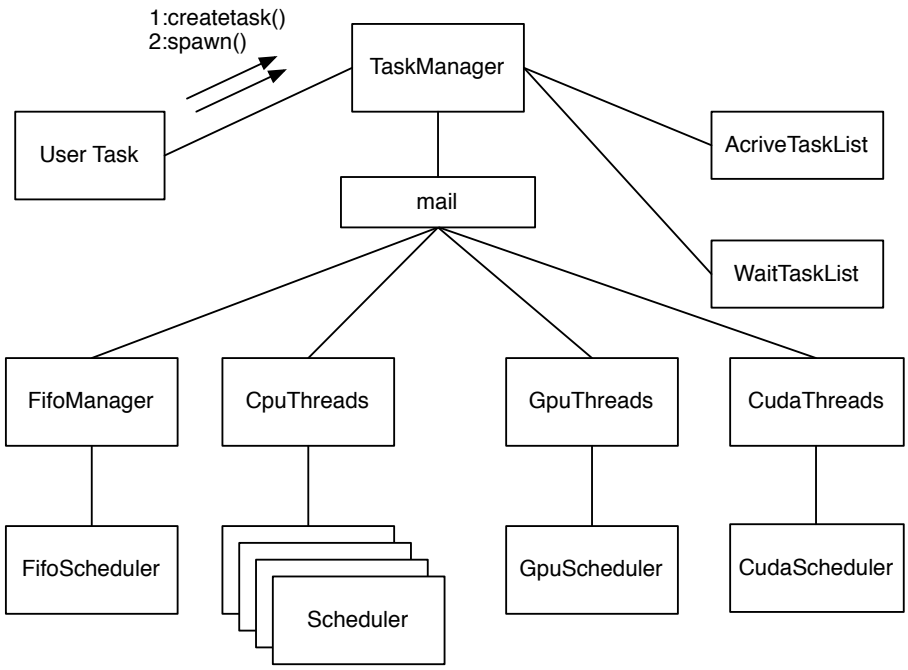

図 3.1: Task Manager

3.4 Task の Scheduling

GPU や Cell のような Shared Memory でない環境でのプログラミングを行う場合、Task の入出力となるデータを転送し、転送が終わってから Task を起動しなければならない。転送処理がボトルネックとなり、並列度が低下してしまう。そのため、Cerium はパイプライン実行をサポートしている。

Scheduler に転送された Task はパイプラインで処理される (図:3.2)。Task が全て終了すると Scheduler から TaskManager に mail を通して通知される。通知に従い依存関係を解決した Task が再び TaskManager から Scheduler に転送される。

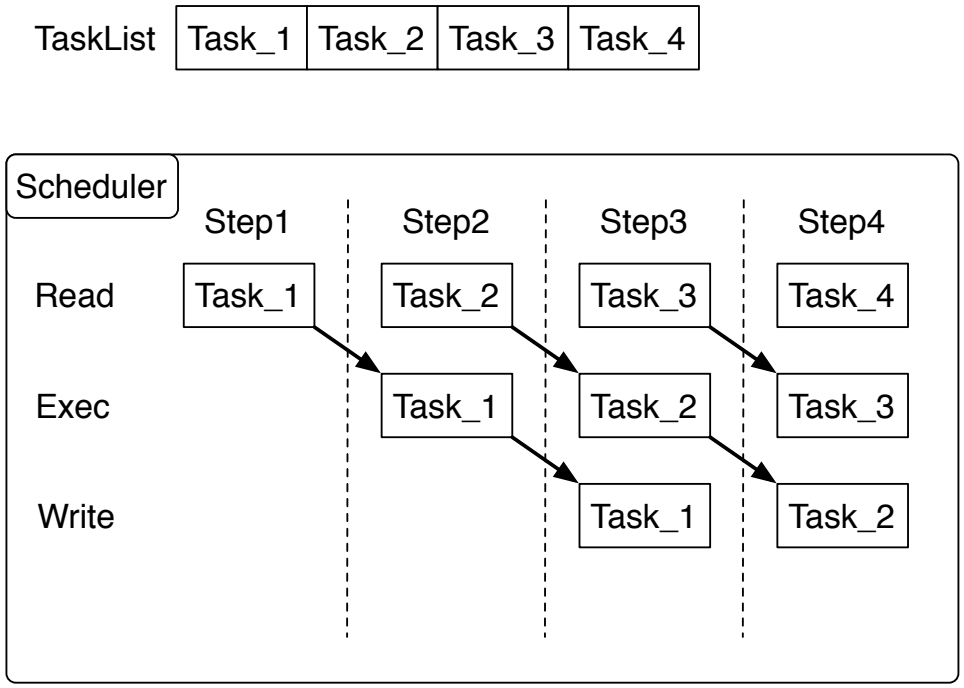

図 3.2: Scheduler

3.5 Task 生成の例

ソースコード:3.1 に Task を生成する例題を示す。input data を 2 つ用意し、 input data 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。

ソースコード 3.1: Task の生成

```
1  void
2  multiply_init(TaskManager *manager, float *i_data1, float *i_data2, float *o_data) {
3
4      // create task
5      HTask* multiply = manager->create_task(MULTIPLY_TASK);
6      multiply->set_cpu(spe_cpu);
7
8      // set indata
9      multiply->set_inData(0, i_data1, sizeof(float) * length);
10     multiply->set_inData(1, i_data2, sizeof(float) * length);
11
12     // set outdata
13     multiply->set_outData(0, o_data, sizeof(float) * length);
14
15     // set parameter
16     multiply ->set_param(0,(long)length);
17
18     // set device
19     multiply->set_cpu(SPE_ANY);
20
21     // spawn task
22     multiply ->spawn();
23 }
```

表:3.1 は Task 生成時に用いる API の一覧である。create された Task は各種パラメタを設定し、spawn/iterate することで TaskManager に登録される。

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 からの出力データのアドレスを追加
set_param	Task へ値を一つ渡す。ここでは length を渡している
set_cpu	Task を実行する Device の設定
spawn	生成した Task を ActiveTaskList に登録する

表 3.1: Task 生成おける API

ソースコード:3.1 は Host 側で Task を生成しているプログラムである。Device 側で実行される Task (OpenCL、CUDA でいう kernel) の記述はソースコード:3.2 のようになる。

ソースコード 3.2: Task

```
1  static int
2  run(SchedTask *s) {
3      // get input
4      float *i_data1 = (float*)s->get_input(0);
5      float *i_data2 = (float*)s->get_input(1);
6      // get output
7      float *o_data = (float*)s->get_output(0);
8      // get parameter
9      long length = (long)s->get_param(0);
10
11     // calculate
12     for (int i=0; i<length; i++) {
13         o_data[i] = i_data1[i] * i_data2[i];
14     }
15     return 0;
16 }
```

表:3.2 は Task 側で使用する API である。

get_input	入力データのアドレスを取得
set_output	出力先データのアドレスを取得
set_param	パラメータを取得

表 3.2: Task 側で使用する API

第4章　Ceriumを用いた例題

Cerium は様々な例題を含んでいる。本論文では Bitonic Sort、 Word Count、 FFT の 3 つの例題を扱う。

Bitonic Sort は、ベンチマークをとる際の一般的な例題として選択した。

Word Count は、計算自体は条件に合う word をカウントアップしていくシンプルな内容である。シンプルな計算でも並列化する事で大きな性能向上を狙える事を示す。

FFT:Fast Fourier Transform(高速フーリエ変換) は、信号処理や画像処理から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、データ並列と相性がよく、 GPGPU で高い並列度を維持できる事が知られている。

以上 3 つの例題を用いてベンチマークを行っていく。本論文で使用する各種例題について紹介する。

4.1　Bitonic Sort

Cerium Task Manager を使った Sort である。Bitonic Sort は配列の分割を行い、分割した部分に対して sort を行う。分割後の Sort には QuickSort を使用している。Task の構成は以下のようになる。

- SortSimpleTask
- QuickSortTask

指定された数の乱数を生成し、sort する例題である。SortSimpleTask は Task の割り当てを行う Task である。QuickSortTask は割り当てられた範囲を QuickSort により Sort する Task である。図:4.1 に Bitonic Sort の例を示す。SimpleSortTask は乱数列を分割し、QuickSortTask に割り当てる。QuickSortTask は割り当てられた部分を Sort する。分割した部分を QuickSortTask に割り当て、繰り返し起動していく事で Sort を行う。

1. SimpleSortTask が乱数列を分割し、 QuickSortTask に割り当てる
2. QuickSortTask が割り当てられた部分を Sort する
3. SimpleSortTask が最初に割り当てた範囲の中間から次の範囲の中間までを QuickSortTask に割り当てる

4. QuickSortTask が割り当てられた部分を Sort する

このような Task の分割 → Sort を分割数分繰り返し実行することで全体を Sort する。
本論文では Bitonic Sort による測定を行う場合、10 万入力を Input とするベンチマークを行う。

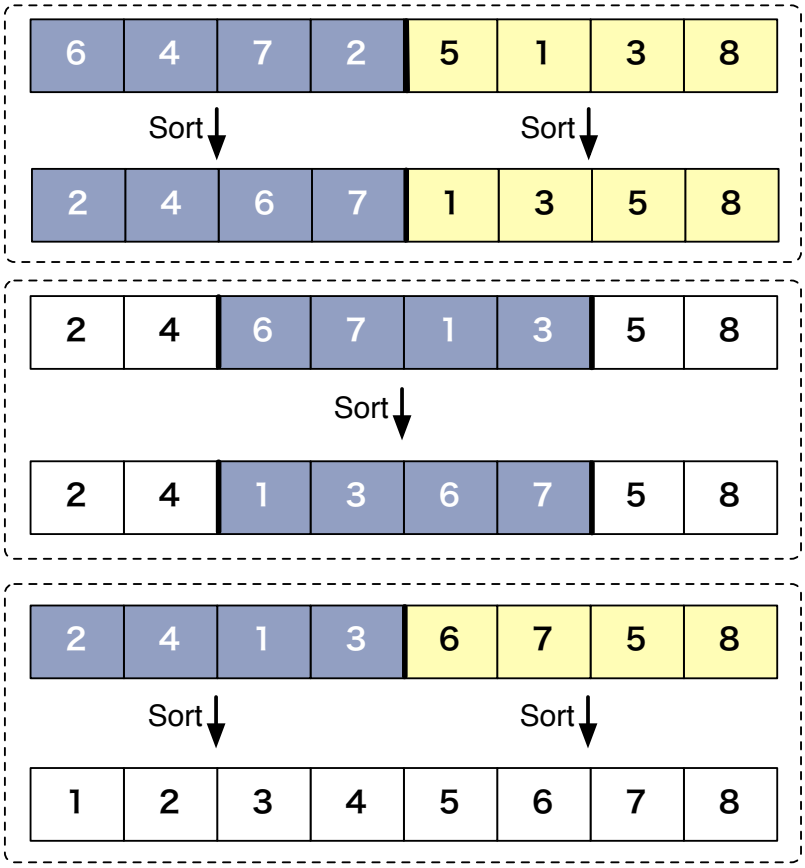

図 4.1: Bitonic Sort の例

4.2 Word Count

WordCount は Input としてファイルを受け取り、ファイルの単語数と行数を集計し、表示する。空白で区切られたものを単語として扱う。

Word Count の Task の構成は以下の通りである。

- WordCountTask
- PrintTask

WordCountTask は Input された data を Word Count し、単語数と行数を Output として指定された Data 領域に書き込む Task である。

Task には分割されたテキストが送られてくるため、送られてきたテキストの前後の状態によっては振る舞いを変える必要がある。分割により Word の途中で切れてしまう場合があり、その場合だと word 数-1 する処理が必要になる。そのため、 WordCountTask は自分が割り当てられた範囲である data の先頭と末尾のパラメータを返す。

PrintTask は WordCountTask によって書き出された単語数と行数を集計し、出力する Task である。集計時は全 WordCount の結果を照らし合わせ、分割されたテキストを正しく整合する。PrintTask は WordCountTask を wait する設定で、全ての WordCountTask が終了したあとに動作する。

WordCount の対象として入力されたファイルは、 mmap を用いてメモリに展開する。その後データを 16KByte の大きさに分割しながら WordCountTask に割り当てていく。

各 Task 間の data の流れを図:4.2 に示す。

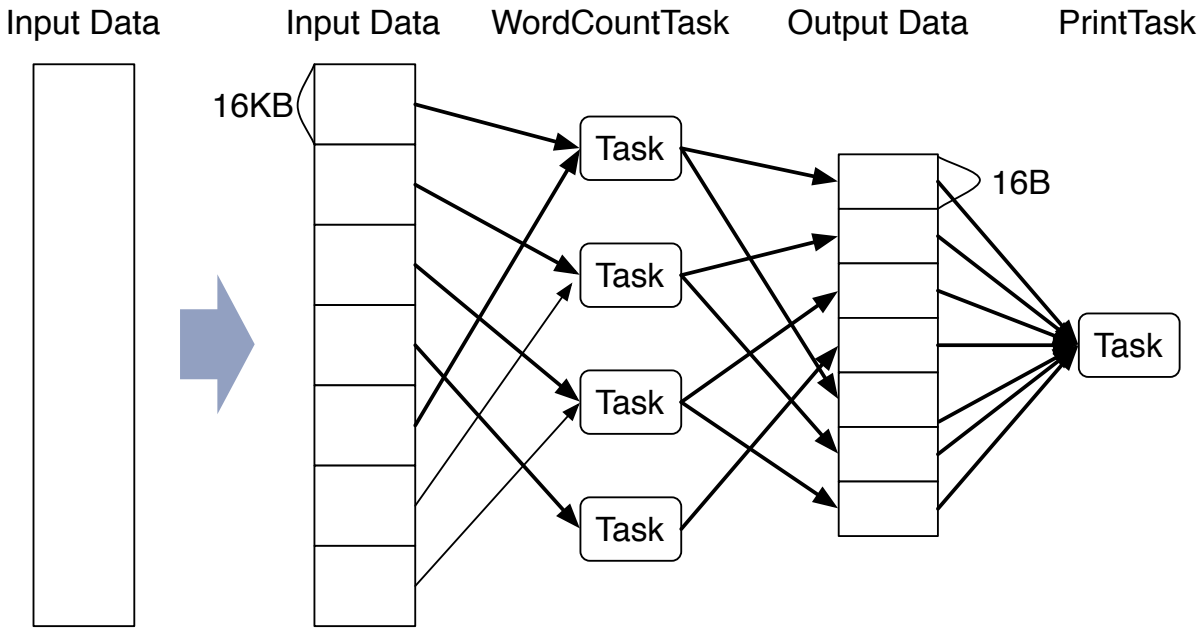

図 4.2: WordCount のフロー

4.3　FFT

FFT:Fast Fourier Transform(高速フーリエ変換) は、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である。今回は入力として受け取った画像に対してハイパスフィルターを行う例題である。

FFT の Task の構成は以下のとおりである。

- BitReverse
- Butteerfly
- HighPassFilter
- SpinFact
- Transpose

FFT は画像 Data に対して様々な Task を割り当てる。入力は比較的大きなサイズを想定している。Input と Output を繰り返し行うと、特に GPU だとボトルネックになってしまう。このベンチマークで並列度を維持するにはデータ並列実行に対応し、データ依存で並列化を可能にする必要がある。

第5章　マルチコアへの対応

Cerium は Cell 上で並列に動作するフレームワークであったが、 Mac OS X 、 Linux 上でも並列に実行させることを可能にした。

5.1　マルチコア上での実行の機構

Cell には MailBox という機能がある。Cell は Shared Memory でないので、 Memory に直接アクセスできない。そこで MailBox を用いて双方向のデータの受け渡しを可能にしている。MailBox は FIFO キュー構造になっており、Device と Host でこの MailBox に対応させる形で Synchronized Queue を用いて MacOSX 、Linux 用の TaskManager へ MailBox を移植した。

Syncronized Queue はバイナリセマフォで管理されており、Queue を操作しているスレッドは常に一つになる。各スレッドは Input 用と Output 用として Synchronized Queue を 2 つ持っており、管理スレッドから Task を受けて並列に実行する。

5.2　DMA

Cell ではデータの受け渡しとして MailBox 以外に DMA 転送を使用する方法がある。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 も DMA 転送を用いている箇所がある。しかしマルチコア CPU 上で実行する場合は各 CPU で同じメモリ空間を利用できる。よって DMA 転送を用いていた部分をポインタ渡しを行うように修正し、メモリに直接アクセスさせることで速度の向上が見込める。

第6章　GPGPU への対応

Cerium の新たな演算資源として GPU の使用を可能にした。現在、GPU のような異なる種類のアーキテクチャを搭載した CPU 、つまりヘテロジニアスな CPU が増えている。特定の計算に特化した Task の生成やスケジューリングを行い、 GPGPU により高い並列度を出す研究は様々な分野で行われている。本研究では Cerium を特定の計算に限らず、 GPU を用いて汎用計算を行えるフレームワークに改良する。

6.1　OpenCL および CUDA による実装

OpenCL 、CUDA による GPGPU 対応を行った。Scheduler と CpuThreads に対応させる形で OpenCL を用いた GpuScheduler と GpuThreads、CUDA を用いた CudaScheduler と CudaThreads を実装した。それぞれの Scheduler 内で各フレームワークの API を用いて GPU の制御を行っている。

TaskManager から受け取った TaskList をもとに Device 上のメモリバッファを作成する。その後 CommandQueue、 Stream といったそれぞれの Queue に Device 制御用の Command を Queueing していく。

Command は Queueing した順に実行されるので、以下のように Command を Queueing する。

1. Host から Device へのデータ転送
2. kernel の実行
3. Device から Host へのデータ転送

データの転送や kernel の実行は非同期 API を用いることで並列に行うことができる。

通常、フレームワークが依存関係を解決して実行するが、非同期 API を用いる場合はユーザが依存関係を考慮する必要がある。しかし Task の依存関係は TaskManager が既に解決した状態で送ってくるので、Scheduler は依存関係を考慮せずに実行して問題ない。

GPGPU 用の Scheduler は CommandQueue を 2 つ持っており、Task をパイプライン的に実行する。

転送されてきた Task が全て終了すると、TaskManager 間の通信を担当する同期キューである 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 でその TaskList に関する依存関係が解消され、

GPGPU の Scheduler 内で Platform や Device ID の取得、 Context の生成、 Kernel の Build と Load 等も行っており、OD 並列処理したい計算のみに集中できる。

6.2　データ並列

並列プログラミングにおいて、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 より高速に演算を行う事ができる。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。OpenCL 、 CUDA ともにデータ並列をサポートしている。Task を実行する際にデータをどう分割するか指定し、kernel をデータ並列実行用に書き換えることで実現する。データ並列実行用の kernel は以下のように記述する。2 つの input データの積を output データに格納する例題、 multiply を用いる。

ソースコード 6.1: Multiply(OpenCL)

```
1  __kernel void
2  multiply(__global const long *params,
3           __global const float *input1,
4           __global const float *input2,
5           __global const float *output) {
6
7      long id = get_global_id(0);
8
9      output[id] = input1[id] * input2[id];
10 }
```

ソースコード 6.2: Multiply(CUDA)

```
1  __global__ void
2  multiply(__global const long *params,
3           __global const float *input1,
4           __global const float *input2,
5           __global const float *output) {
6
7      int id = blockIdx.x * blockDim.x + threadIdx.x;
8
9      output[id] = input1[id] * input2[id];
10 }
```

このような kernel を分割数分生成する。分割数は kernel の生成時にそれぞれのフレームワークが 用意している API を用いて指定する。いずれの kernel も

- 自分の計算する範囲を取得 (ソースコード 6.1、ソースコード 6.2 の 7 行目)
- 取得した範囲を計算 (ソースコード 6.1、ソースコード 6.2 の 9 行目)

という手順で処理する。計算する範囲については OpenCL では取得用の API を用い、CUDA では kernel の持つ組み込み変数から算出する。

Cerium でも データ並列実行をサポートする。GPU におけるデータ並列実行だけでなくマルチコア CPU 上でのデータ並列実行にも対応する。なお、マルチコア CPU 上で実行する場合も GPU 実行時の kernel (ソースコード 6.1、ソースコード 6.2) となるべく近

い形式で記述できるようにする。マルチコア CPU 上でデータ並列実行する場合、 kernel は以下のように記述する。

ソースコード 6.3: Multiply(CPU)

```
1  static int
2  run(SchedTask *s, void *rbuf, void *wbuf) {
3      float *indata1, *indata2, *outdata;
4
5      indata1 = (float*)s->get_input(rbuf, 0);
6      indata2 = (float*)s->get_input(rbuf, 0);
7      outdata = (float*)s->get_output(wbuf, 0);
8
9      long id = (long)s->get_param(0);
10     outdata[id] = indata1[id] * indata2[id];
11     return 0;
12 }
```

OpenCL 、CUDA と違い値を引数として直接渡すのではなく、メモリバッファから Load し、計算を行う。値渡しや修飾子等若干の違いはあるが、ほぼ同じ形式で kernel を記述することができる。CPU、 OpenCL、 CUDA いずれか 1 つの記述から残りのコードも生成できるようにする事が望ましい。

Cerium でデータ並列実行を行う場合、Task を spwan API でなく iterate API で生成すればよい。iterate API は複数の length を引数とし、length の値がデータ分割後に各 Task が担当するサイズ、length の個数がデータの次元数となる。これを元に Scheduler が各 Task が担当する index を計算し、Task に set_param する。

Task は実行時に get_param することで set_param した値を取得し、担当範囲をデータ並列を実行する。この get_param が OpenCL における get_global_id API に相当する。

index の割り当ての例を示す。データ数 10 個の入力を持つ Task に対して CPU 数 4、一次元における分割でデータ並列実行した場合の index の割り当ては表:6.1 になる。

この例だと各 CPU に対する index の割り当ては CPU0 は index 0、4、8、 CPU1 は index 1、5、9、CPU2 は index 2、6、CPU3 は index 3、7 となる。

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

表 6.1: データ並列実行時の index の割り当て

並列プログラミングだと、並列化部分が全て同一の Task であるということは少なくない。その際、 Task 生成部分をループで回すことなく、簡単な Syntax で記述することができる。

データ並列で実行する場合、 Input と Output を各 Task 間で共有するため、少ないコピーに抑えられる。CPU ではメモリ領域を節約する事ができるが、 Task と Manager でメモリ領域が同じ (2.2 節) なため、コピーによるオーバーヘッドは少ない。

しかし GPU は SharedMemory ではなく、データの転送がオーバーヘッドとなる。コピーを減らす事で並列度の向上が見込める。

第7章　並列処理向けI/O

ファイル読み込みなどの I/O を含むプログラムは、読み込み時間が Task の処理時間と比較してオーバーヘッドになることが多い。プログラムの並列化を行ったとしても、 I/O がボトルネックになってしまうと処理は高速にならない。本項では Cerium に並列処理用の I/O の実装を行う。これにより I/O 部分の高速化を図る。

7.1　mmap

Cerium ではファイルの読み込みを mmap により実装していた。しかし、mmap や read によりファイルを読み込んでから処理を実行させると、読み込んでいる間は他の CPU が動作せず、並列度が落ちてしまう。そこで、 I/O 部分も並列に動作するよう実装した。

Read を並列に行うには、File Open ではなく mmap を使う方法がある。mmap はすぐにファイルを読みに行くのではなく、まず仮想メモリ空間にファイルの中身を対応させる。メモリ空間にアクセスが行われると、OS が対応したファイルを読み込む。

mmap で読み込んだファイルに Task1、Task2 がアクセスし、それぞれの処理を行う際の例を図:7.1 に示す。

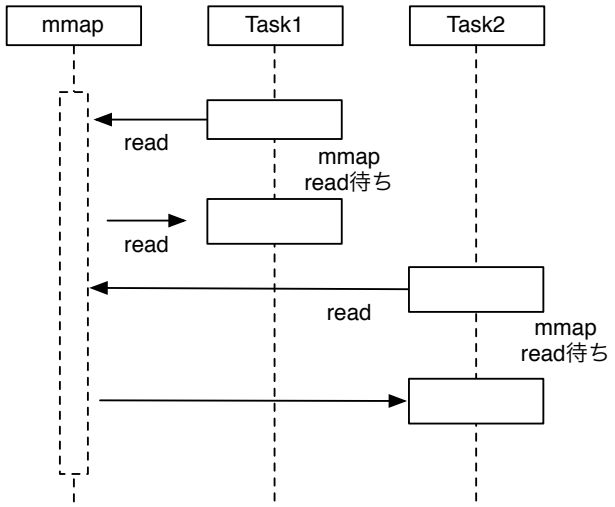

図 7.1: mmap の Model

Task1 が実行される時、仮想メモリ上 Open されたファイルの読み込みを行い、Task1

の処理を行う。その後、同様に Task2 も読み込みを行ってから処理を行う。この Task1 と Task2 の間に待ちが入る。

ファイルの読み込みが起こると、アクセスした Thread/Process には wait がかかってしまう。しかし mmap による Read は Task と並列に実行されるべきである mmap は逐次アクセスを仮定しているので、OS 内部で自動的にファイルの先読みを行う事も期待できる。しかしそれは OS の実装に依存してしまう。読み込みが並列に実行されない場合、Task が読み込み待ちを起こしてしまう。読み込みが OS 依存となるため、環境によって左右されやすく、汎用性を損なってしまう。

そこで、mmap を使わず、read を独立したスレッドで行い、読み込まれた部分に倒して並列に Task を起動する。これを Blocked Read と呼ぶ。Blocked Read によるプログラミングは複雑になるが、高速化が期待できる。

7.2 Blocked Read による I/O の並列化

Blocked Read を実装するに辺り、WordCount を例に考える。Blocked Read はファイル読み込み用の Task(以下、ReadTask) と読み込んだファイルに対して処理を行う Task(今回は WordCount) を別々に生成する。ReadTask はファイル全体を一度に全て読み込むのではなく、ある程度の大きさで分割してから読み込みを行う。分割後の読み込みが終わると、読み込んだ範囲に対して WordCount を行う。

BlockedRead による WordCount を行う場合、図:7.2 のようになる。

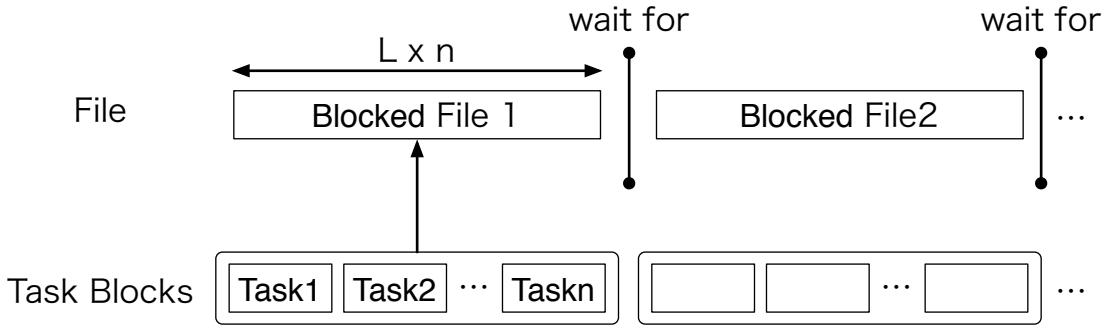

図 7.2: BlockedRead による WordCount

Task を一定数まとめた単位で生成し、起動を行っている。この単位を Task Block と定義する。TaskBlock が BlockedRead を追い越して実行してしまうと、まだ読み込まれてない領域に対して処理を行ってしまう。その問題を解決するため、依存関係を設定する。BlockedRead による読み込みが終わってから TaskBlock が起動されるよう、Cerium の API である wait_for により依存関係を設定する。

以上を踏まえ、BlockedRead の実装を行った。BlockedRead Task の生成はソースコード:7.1 のように行う。

ソースコード 7.1: BlockedRead を行う Task の生成

```
1 HTaskPtr readTask = manager->create_task(READ_TASK)
2 readTask->set_cpu(DEVICE_TYPE);
3 readTask->set_outData(0, file_map + task_num * division_size, task_blocks * division_size);
4 readTask->set_param(0, fd);
5 readTask->set_param(1, task_num * division_size);
6 runTask();
7 readTask->set_param(2, task_num * division_size)
8 readTask->spawn();
```

- 3 行目、set_outData(0): ファイルを読み込んだ際の格納場所を設定
- 4 行目、set_param(0): 読み込むファイルディスクリプタを設定
- 5 行目、set_param(1): BlockedRead Task で読み込むファイルの範囲の先頭のポジションを設定
- 7 行目、set_param(2): BlockedRead Task で読み込むファイルの範囲の末尾のポジションを設定

Cerium において、 WordCount で必要な Task を全て生成してしまうと、その Task のデータ構造自体がメモリを消費してしまう。そこである程度の量の Task を起動し、それが終了してから (正確には終了する前に) 次の Task を生成するようになっている。それらの機能を持った関数が 6 行目にあたる run_tasks である。run_tasks に wait_for による ReadTask との待ち合わせの処理を入れれば良い。

BlockedRead の Task をいかに示す。

ソースコード 7.2: BlockedRead Task

```
1  static int
2  read_task(SchedTask *s, void *rbuf, void *wbuf) {
3      long fd = (long)s->get_param(0);
4      long start = (long)s->get_param(1);
5      long end = (long)s->get_param(2);
6      char txt = (char*)s->get_output(wbuf, 0);
7      long size = end - start;
8
9      pread(fd, txt, size, start);
10     return 0;
11 }
```

Cerium の API により、生成部分で設定したパラメタをそれぞれ受け取る。ファイル読み込みの先頭・末尾のポジションが渡されているので、ファイルから読み込むサイズは求められる。受け取ったパラメタをそれぞれ pread 関数に渡すことで Blocked Read を実現している。

7.3　I/O 専用 Thread の実装

Cerium Task Manager では、各種 Task にデバイスを設定することができる。SPE_ANY 設定を使用すると、 Task Manager で CPU の割り振りを自動的に行う。BlockedRead は連続で読み込まれなければならないが、SPE_ANY 設定で実行すると BlockedRead 間に別の Task を割り込んでしまう場合がある。(図:7.3)

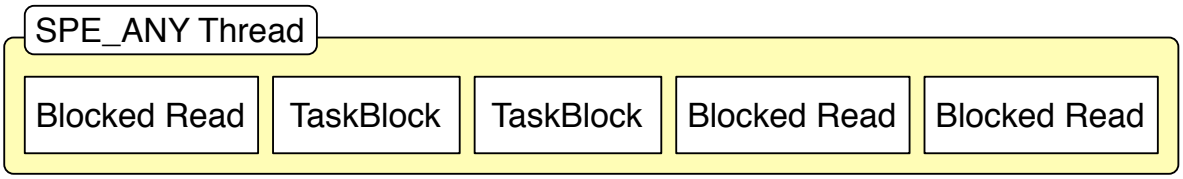

図 7.3: BlockedRead と Task を同じ thread で動かした場合

そこで I/O 専用の Thread である IO_0 の追加を行った。

IO_0 は SPE_ANY とは別 Thread の Scheduler で動くので、SPE_ANY で動いている Task に割り込まれることはない。しかし、読み込みの終了を通知し、次の read を行う時に他の Task がスレッドレベルで割り込んでしまう事がある。pthread_getschedparam() で IO_0 の priority の設定を行う必要がある (図:**??**)。

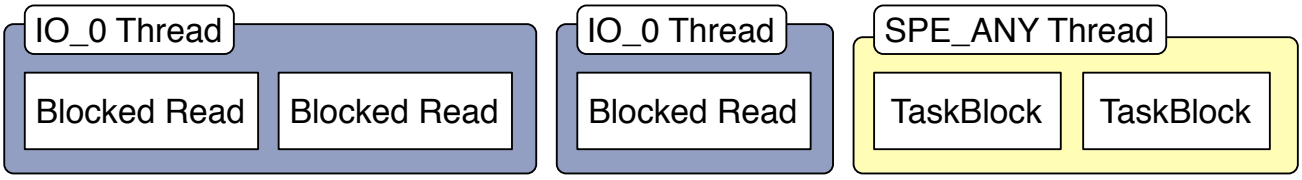

図 7.4: IO Thread による BlockedRead

IO_0 で実行される Task は BlockedRead のみなので、IO_0 の priority を高く設定することで Blocked Read は連続で実行される。

また、以上の事から I/O を含む処理では、 I/O を行う Thread の priority を高くする必要があるという知見を得られた。

第8章　Memory Allocator

8.1　現状のMemory Allocator

8.2　新しいMemory Allocator

第9章　ベンチマーク

9.1　実験環境

今回使用する実験環境を表:9.1、表:9.2 に示す.

名前	概要
Model	MacPro Mid 2010
CPU	6-Core Intel Xeon @2.66GHz
Serial-ATA Device	HDD ST4000VN000-1H4168
Memory	16GB
OS	MacOSX 10.10.1
Graphics	NVIDIA Quadro K5000 4096MB

表 9.1: Cerium を実行する実験環境 1

名前	概要
Model	MacPro Late 2013
CPU	6-Core Intel Xeon E5@3.5GHz
Serial-ATA Device	Apple SSD SM0256
Memory	16GB
OS	MacOSX 10.10.1
Graphics	AMD FirePro D700 6144MB

表 9.2: Cerium を実行する実験環境 2

なお、表:9.1 と表:9.2 はCPU クロック数の他にも Strage や GPU の性能にも違いがある。実験環境 1(表:9.1) は実験環境 2(表:9.2) に比べてクロック数が低く、Strage は HDD を使用している。なお、 GPU は NVIDIA 製の高性能なものを使用している。実験環境 2(表:9.2) はクロック数が高く、 Strage に SSD を使用している。

以上の環境で今回新たに実装したマルチコア、GPGPU 、並列 I/O のベンチマークを行う。

9.2　マルチコア

マルチコア GPU における並列実行について、Sort(図:9.1) と WordCount(図:9.2) によるベンチマークを行った。

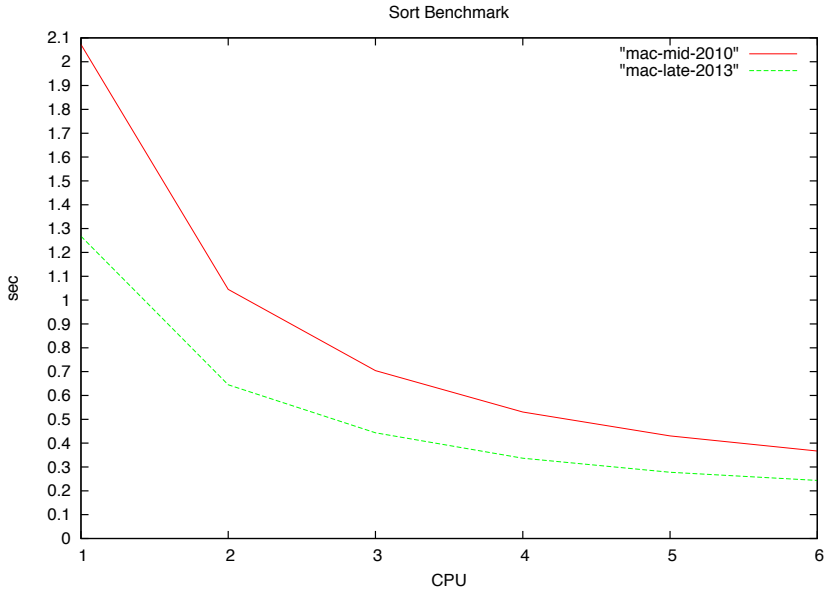

図 9.1: マルチコア CPU における Sort

MacPro 2013 Late Model において、6CPU を使用した場合、1CPU を利用した場合と比較して、Sort は 5.2 倍、WordCount は 4.9 倍の速度向上が見られる。MacPro 2010 Mid Model においても Sort は 5.65 倍、WordCount は 5.0 倍の速度向上が見られた。

9.3　GPGPU

9.4　並列 I/O

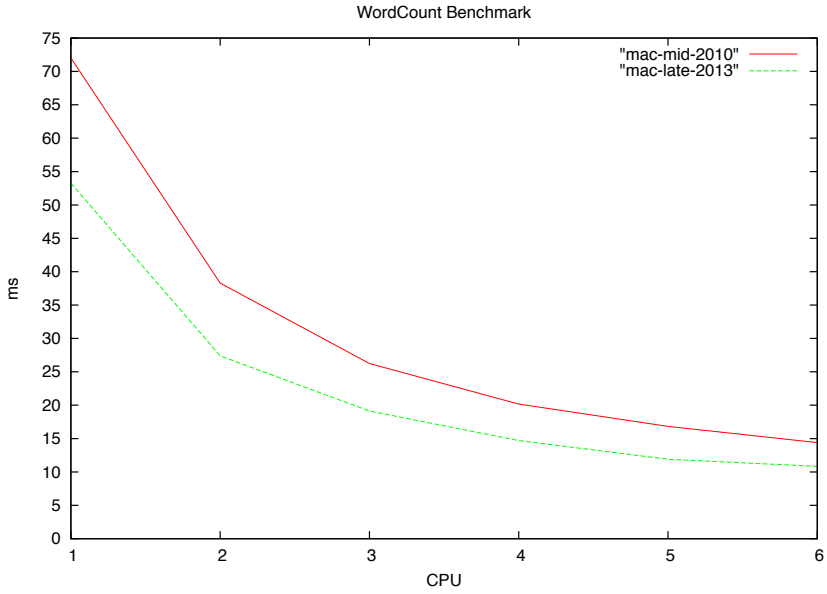

図 9.2: マルチコア CPU における WordCount

第10章　結論

10.1　まとめ

10.2　今後の課題

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.
研究を行うにあたり, 並列計算環境の調整, 意見, 実装に協力いただいた小久保翔平さん, 並びに並列信頼研究室の全てのメンバーに感謝いたします.
最後に, 大学の修士まで支えてくれた家族に深く感謝します.

参考文献

[1] Messagepack. http://msgpack.org/.

[2] 大城信康, 河野真治. Data segment の分散データベースへの応用. 日本ソフトウェア科学会, September 2013.

[3] 玉城将士, 河野真治. Cassandra を使ったスケーラビリティのある cms の設計. 情報処理学会, March 2011.

[4] 玉城将士, 河野真治. Cassandra と非破壊的構造を用いた cms のスケーラビリティ検証環境の構築. 日本ソフトウェア科学会, August 2011.

[5] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, Mar 2003.

[6] Fay Changand Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storege system for structured data.

[7] Nancy Lynch and Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly avaliable key-value store.

[9] 所眞理雄. DEOS プロジェクト研究成果集 Dependability Engineering for Open Systems, 2013.

[10] 永山 辰巳, 横手 靖彦. オープンシステムディペンダビリティと D-Case を繋ぐリポジトリ, 2013.

発表履歴

- Java による授業向け画面共有システムの設計と実装, 大城信康, 谷成雄 (琉球大学), 河野真治 (琉球大学), オープンソースカンファレンス 2011 Okinawa, Sep, 2011
- Continuation based C の GCC 4.6 上の実装について,
  大城信康, 河野真治 (琉球大学),
  第 53 回プログラミング・シンポジウム, Jan, 2012
- GraphDB 入門 TinkerPop の使い方,
  大城信康, 玉城将士 (琉球大学),
  第 15 回 Java Kuche, Sep, 2012
- ディペンダブルシステムのための木構造を用いた合意形成データベースの提案と実装,
  大城信康, 河野真治 (琉球大学), 玉城将士 (琉球大学), 永山 辰巳 (株式会社 Symphony),
  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, 2013
- Data Segment の分散データベースへの応用,
  大城信康, 杉本優 (琉球大学), 河野真治 (琉球大学),
  日本ソフトウェア科学会 30 回大会 (2013 年度) 講演論文集, Sep, 2013