並列プログラミングフレームワーク Cerium の改良

Improvement of parallel programming framework Cerium

平成23年度 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

金城 裕

要 旨

アブストラクト

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景と目的

本研究では、当研究室で開発した並列プログラミングフレームワーク Cerium の改良を行った。マルチコア CPU という複数のコアを持つ CPU の登場により、その台数効果を得るために並列プログラミングを行う必要がでてきた。複数のコア上でプログラムを動作させる場合、並列プログラミングが必要になる。並列プログラミングで台数効果を出すためには、プログラム全体を対象にした並列度が高くなければならない。そのため、並列化部分の特定を行う必要がある。明示的な並列化部分はループ部分であるが、それがプログラムの大部分でなければ、台数効果はでない。そのためループ部分以外ではプログラマが効率的な並列化を考えなければならない。また並列化の際には各コアへの通信、同期、排他制御等の要素が追加され、これらは逐次プログラミングにおいては考慮する必要がなかった。加えて非同期での処理が含まれるため、再現性のないバグが発生する可能性があり、それに対応するデバッグが必要である。またアーキテクチャを理解し、最適なプログラムを書く必要がある。これらのように、並列プログラミングでは、その信頼性を維持するために考慮すべき点が逐次プログラミングより多く、開発には技術と手間が必要で、時間がかかる問題がある。Cell 上でのゲーム作成を行う学生実験でも講義期間中に作品が一定のレベルに達しない問題が生じた。そこで、当研究室では並列プログラミングフレームワーク Cerium を開発した。Cerium は ゲームのシーンを記述する SceneGraph、ソフトウェアレンダリングを行う RenderingEngine、Task をスケジューリングする TaskManager から構成されている。Task とは Cerium が提供するデータ構造で、処理を Task として記述していくことで、TaskManager が自動的にスケジューリングを行い Task の情報に沿った、各コアへの処理の割り振り、同期を行う。現在 RenderingEngine 部分が Task で記述され並列処理にされている。Cerium の動作環境は Cell/PS3, Linux, MacOSX であり、どの環境でもほぼ同じコードで動作する。Cerium を用いることで講義期間中でのゲーム作成が可能になった。しかし 実験での使用や例題の実装を通してフレームワークとしての信頼性が十分でないことが明らかになった。それは 仕様通りの稼働率がでないこと、アーキテクチャ依存の記述が避けにくことである。これらの問題点を解決するためにソフトウェア MailQueue の実装、Task をグルーピングする TaskArray の実装、明示的な DMA ロードを隠蔽する MemorySegment の API の実装、また細かなチューニングを行った。これらの改良によって、信頼性のある並列プログラミングフレームワークを目指す。

第2章　Cell Broadband Engine

本研究で実験題材の対象となった Cell アーキテクチャについて説明する。

Cell Broadband Engine [?] は、ソニー・コンピュータエンタテインメント、ソニー、IBM , 東芝によって開発されたマルチコア CPU である。Cell は、1 基の制御系プロセッサコア PPE (PowerPC Processor Element) と 8 基の演算系プロセッサコア SPE (Synergistic Processor Element) で構成される。各プロセッサコアは、EIB (Element Interconnect Bus) と呼ばれる高速なバスで接続されている。また、EIB はメインメモリや外部入出力デバイスとも接続されていて、各プロセッサコアは EIB を経由してデータアクセスをおこなう。本研究で用いた PS3Linux (Yellow Dog Linux 6.2) では、6 個の SPE を使うことが出来る (図 2.1)

この PPE と SPE の 2 種類の CPU を、プログラマ自身が用途に合わせて適切に使い分けるように考慮する必要がある。

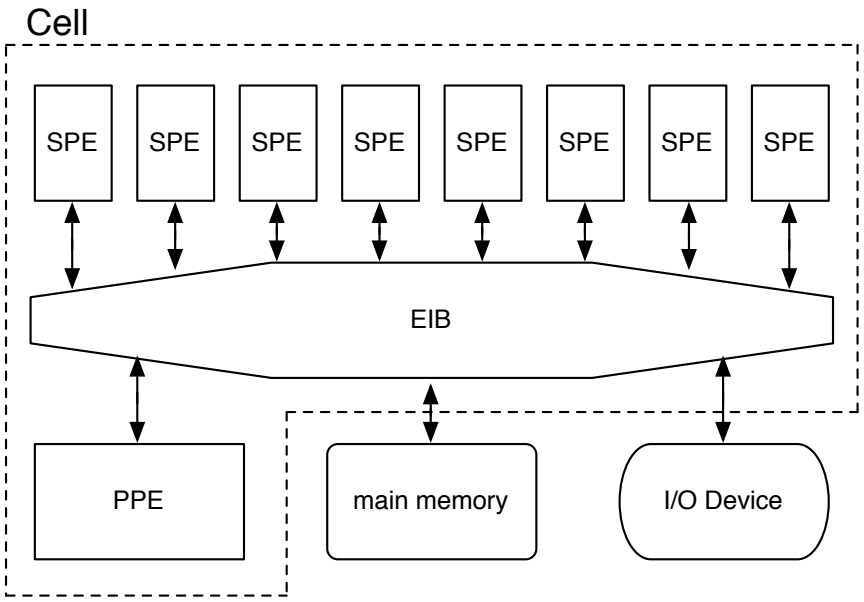

図 2.1: Cell Broadband Engine Architecture

2.1 PPE (PowerPC Processor Element)

PPE は Cell Broadband Engine のメインプロセッサで、複数の SPE をコアプロセッサとして使用することができる汎用プロセッサである。メインメモリや外部デバイスへの入出力、SPE を制御する役割を担っている。PPU (PowerPC Processor Unit) は、PPE の演算処理を行うユニットで、PowerPC アーキテクチャをベースとした命令セットを持つ。PPSS (PowerPC Processor Storage Subsystem) は PPU からメインメモリへのデータアクセスを制御するユニットである (図 2.2) 。

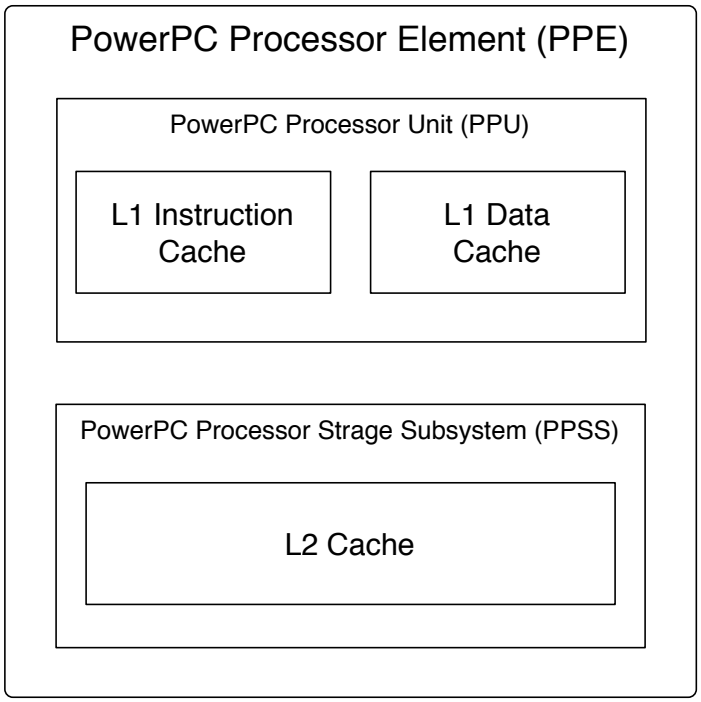

図 2.2: PPE (PowerPC Processor Element)

2.2 SPE (Synergistic Processor Element)

SPE には 256KB の Local Store (LS) と呼ばれる、SPE から唯一、直接参照できるメモリ領域があり、バスに負担をかける事無く並列に計算を進めることが出来る。SPE からメインメモリへは、直接アクセスすることは出来ず、SPE を構成する一つである MFC (Memory Flow Controller) へ、チャネルを介して DMA (Direct Memory Access) 命令を送ることで行われる (図 2.3) 。

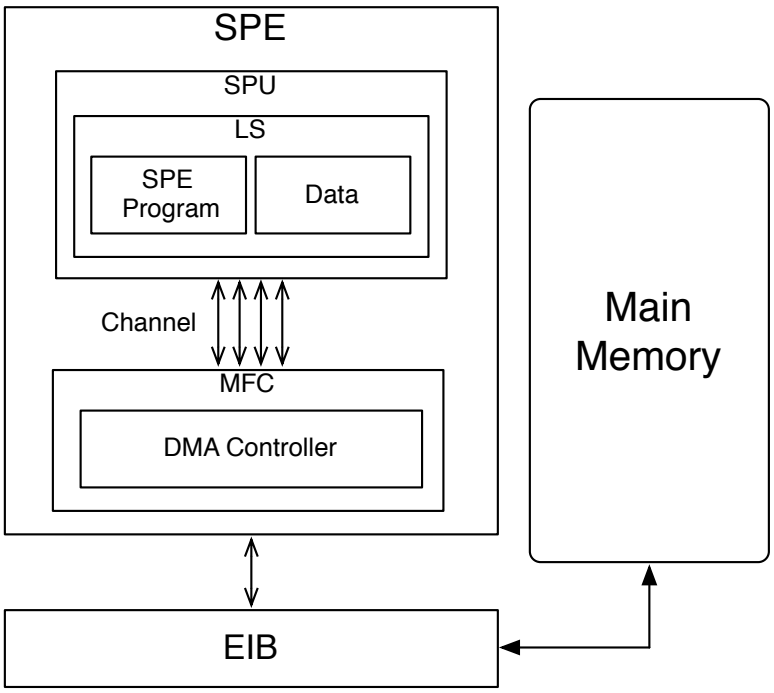

図 2.3: SPE (Synergistic Processor Element)

2.3 Cell の基本性能

2.3.1 DMA (Direct Memory Access)

2.2 節 で述べた通り、SPE は LS 以外のメモリに直接アクセスすることができず、PPE が利用するメインメモリ上のデータにアクセスするには DMA (Direct Memory Access) を用いる。DMA 転送とは、CPU を介さずに周辺装置とメモリとの間でデータ転送ことで、Cell の場合はメインメモリと LS 間でデータの転送を行う。手順としては以下の様になる。

1. SPE プログラムが MFC (Memory Flow Controller) に対して DMA 転送命令を発行
2. MFC が DMA Controller を介して DMA 転送を開始。この間、SPE プログラムは停止しない。
3. DMA 転送の終了を待つ場合、SPE プログラム内で転送の完了を待つ

この時、DMA 転送するデータとアドレスにはいくつか制限がある。転送データが 16 バイト以上の場合、データサイズは 16 バイトの倍数で、転送元と転送先のアドレスが 16 バイト境界に揃えられている必要がある。転送データが 16 バイト未満の場合、データサイズは 1,2,4,8 バイトで、転送サイズに応じた自然なアライメントである (転送サイズのバイト境界に揃えられている) ことが条件となる。

2.3.2 SIMD (Single Instruction Multiple Data)

Cell では、SPE に実装されている 128 ビットレジスタを用いて SIMD を行うことが出来る。SIMD とは、1 つの命令で複数のデータに対して処理を行う演算方式である (図 2.4) 。

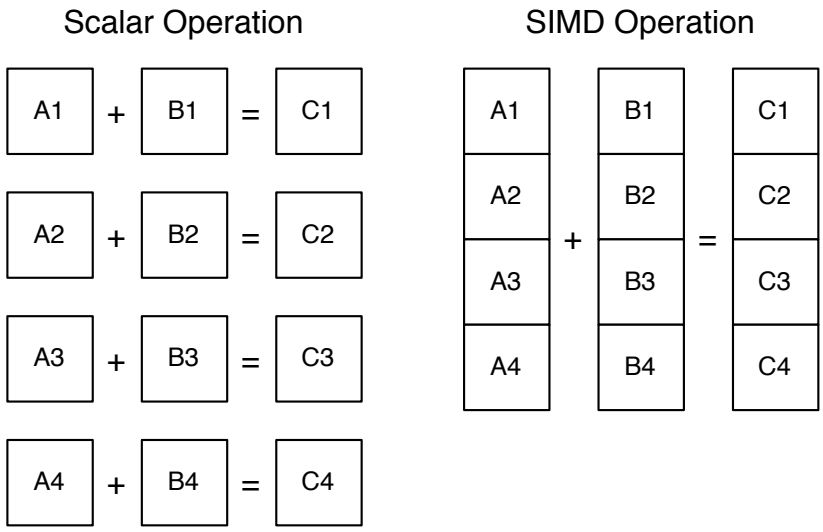

図 2.4: SIMD (Single Instruction Multiple Data)

図 2.4 のスカラ演算は以下のような式に当てはまる。

```
int a[4] = {1, 2, 3, 4};
int b[4] = {5, 6, 7, 8};
int c[4];

for (int i = 0; i < 4; i++) {
    a[i] + b[i] = c[i];
}
```

これに対し、SIMD 演算は以下のようになる。

```
vector signed int va = {1, 2, 3, 4};
vector signed int vb = {5, 6, 7, 8};
vector signed int vc;

vc = spu_add(va, vb);
```

Cell の SIMD 演算では、vector 型の変数を用いる。

このように、通常は 4 回計算するところを 1 回の計算で行うことができる反面、すべての演算を 128 ビットで計算するため、なるべく効果的に行う様に工夫する必要がある。

```
int a, b, c;

c = a + b;
```

この様な計算の場合でも 128 ビット同士の演算を行うため、無駄が生じる。

2.3.3 Mailbox

Mailbox とは SPE の MFC 内の FIFO キューであり、PPE と SPE 間の 32 ビットメッセージの交換に用いられる。Mailbox では 3 つの振る舞いが出来る様に設計されている (図 2.5) 。

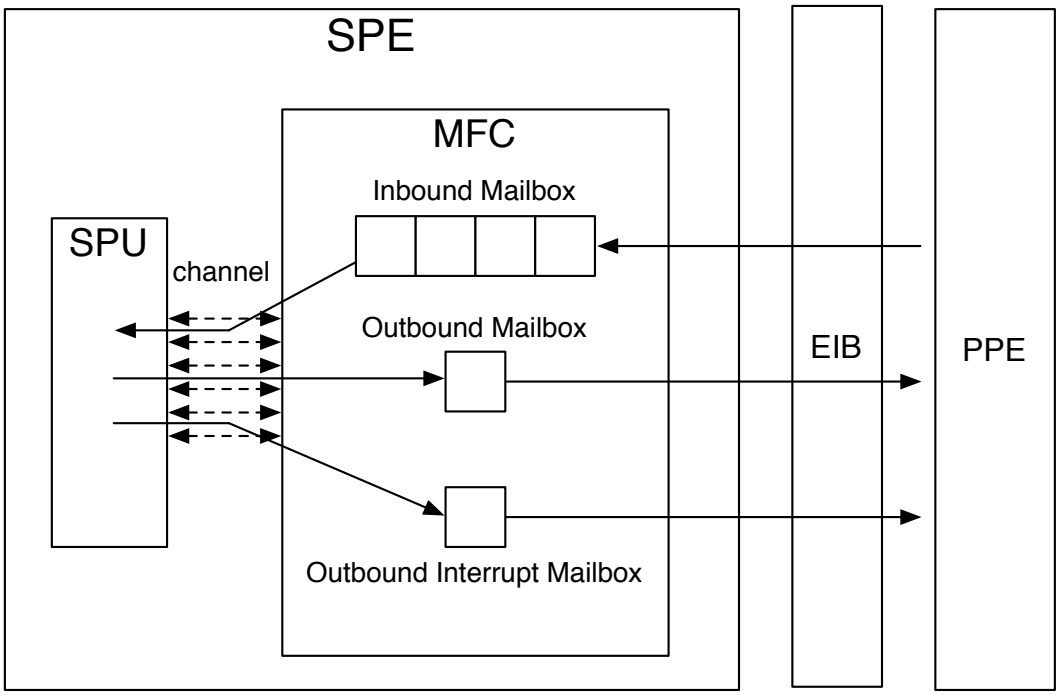

図 2.5: Mailbox

1. SPU Inbound Mailbox
   PPE から SPE へデータを渡すためのキュー。キューのエントリ数は実装依存による [?] が、研究環境では最大 4 個までのデータを蓄積できる。このキューが空の場合は、SPE は、データがメールボックスに書き込まれるまでは、命令でストールする。読み出すデータの順番は書き込んだ順番に保証されている。

2. SPU Outbound Mailbox
   SPE から PPE へのデータを渡すためのキュー。研究環境では最大 1 個までしかデータが蓄積できない。

3. SPU Outbound interrupt Mailbox
   SPU Outbound Mailbox とほとんど同じだが、このキューでは SPE からキューにデータが書き込まれると、PPE に対して割り込みイベントが発生し、データの読み出しタイミングを通知する事が出来る。

2.4 開発環境

2.4.1 libSPE2

libSPE2 とは、PPE が SPE を扱うためのライブラリ群である [?] 。libSPE2 は SPE Context Creation、SPE Program Image Handling、SPE Run Control、SPE Event Handling、SPE MFC Problem State Facilities、Direct SPE Access for Applications という基本構成でできている。Cell の基本プログラムは次の様になる。

1. create N SPE context
2. Load the appropriate SPE executable object into each SPE context's local store
3. Create N threads
4. Wait for all N threads to terminate

2.4.2 SPU C/C++ 言語拡張

SPE では基本的な C 言語の機能の他に、Cell 特有の拡張が行われている [?] 。表 2.1 に主な API を記す。

表 2.1: SPU C/C++ 言語拡張 API

spu_mfcdma32	DMA 転送を開始する
spu_read_in_mbox	PPE からの mail を取得する
spu_write_out_mbox	PPE へ mail を送信する
spu_add、spu_sub、spu_mul	SIMD 演算 (加算、減算、乗算)

SPE を効率よく使う上で 表 2.1 の様な Cell 特有の API や、SPE アセンブラ命令を学ぶことが必要となる。

2.4.3 SPURS

ここでは、現在発表されている Cell の開発環境である SPURS について説明する。

SPURS [?] は、閉じた並列分散と考えることができる Cell の環境で、いかに効率よく動作させるかということを考えたシステムである。

Cell の性能を存分に生かすためには SPE を効率よく使い切ることであり、SPE の動作を止めることなく、同期を最小限に行う必要がある。そこで SPURS では SPE を効率よく利用するために、PPE に依存せずに SPE コードを選択し、実行することと機能は効率重視で割り切ることを挙げている。そのため、SPE 上にカーネルを組み込んでいる。

アプリケーションを複数 SPE で実行するとき、アプリケーションプログラムを出来るだけ小さな単位 (タスク) に分割し、通信ライブラリを用いてタスク間を依存関係で結合する。LS 常駐のカーネルが、実行可能なタスクを選んで実行していく (図 2.6) 。

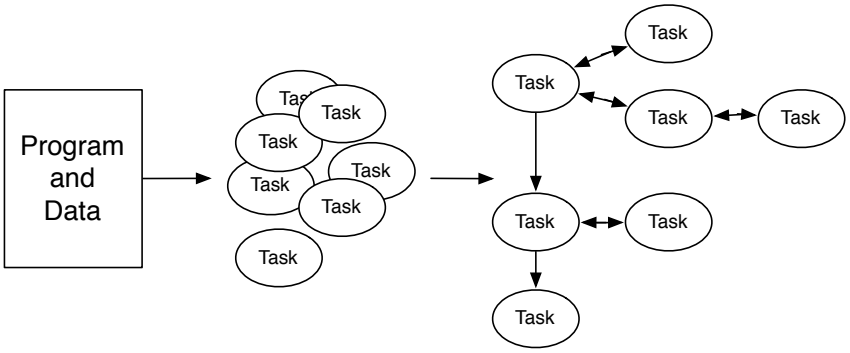

図 2.6: SPURS Task

これらの処理はデータを扱うため、SPURS はパイプラインで実行される (図 2.7) 。

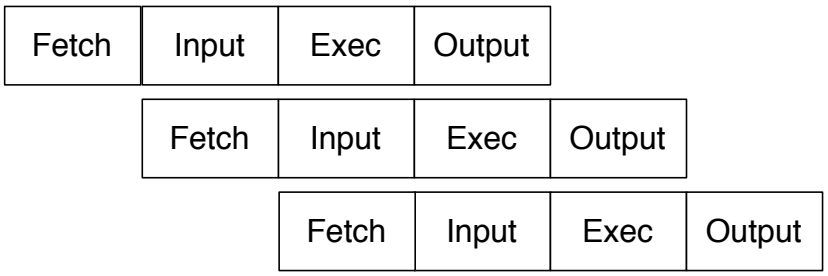

図 2.7: SPURS Pipeline

以上から、SPURS は複数の SPE を効率よく使うためのライブラリとして優れた物であると思われるが、現在一般には公開されていない。

第3章　Cerium

Cerium [?] は Cell 用の Fine-Grain Task Manager として同研究室の宮國が実装した。本章ではその実装について説明する

3.1　Cerium の概要

Cerium は、我々が提案したフレームワークで、TaskManager (3.2), SceneGraph (3.6) と Rendering Engine (3.6.2) の 3 つの要素から構成されており、PS3, Mac OS X, Linux 上でゲームフレームワークとして動作する。ゲーム中のオブジェクトの振る舞いやルールは SceneGraph で管理し、それらの動きや Rendering の処理を動的に SPE に割り振るカーネルとして、TaskManager が用いられる。PS3 の Graphics Engine の仕様は公開されておらず、Cerium は独自の Rendering Engine を有している。

3.2　Cerium TaskManager

Task Manager は、 Task と呼ばれる、分割された各プログラムを管理する。 Task の単位はサブルーチンまたは関数として、 Task 同士の依存関係を考慮しながら実行していく。先行研究において実装されていた TaskManager の API を 表 3.1 に示す。

表 3.1: 旧 Task Manager API

create_task	Task を生成する
run	実行 Task Queue の実行
allocate	環境のアライメントを考慮した 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 の依存関係の考慮
set_cpu	Task を実行する CPU の設定
set_post	Task が終了したら PPE 側で実行される関数の登録
spawn	Task を実行 Task Queue に登録する

3.3 Cerium における Task

Task は TaskManager を使って生成する。 Task を生成する際に、以下のような要素が設定可能である。

1. input data
2. output data
3. parameter
4. cpu type
5. dependency

input, output, data, parameter は関数でいうところの引数に価する。cpu type は Task が PPE または SPE のどちらで実行されるかを示している。 dependency は他の Task との依存関係を示している。依存関係の情報は PPE 側が持っており、 SPE, PPE の Task が終了すると、Task の終了が通知され、その通知に従って PPE が依存関係を処理していく（例: Task A の終了通知を受け、 PPE は Task B を実行可能状態にする)。Task の依存関係の処理を図を用いて説明する。

Task B は Task A の終了を待っている。他の Task の終了を待っている Task は、Wait Queue に、Task を待っていない Task は Active Queue に入れる。この時点で Task A が先頭にあるので Task A が SPE に送られる (図 3.1)。

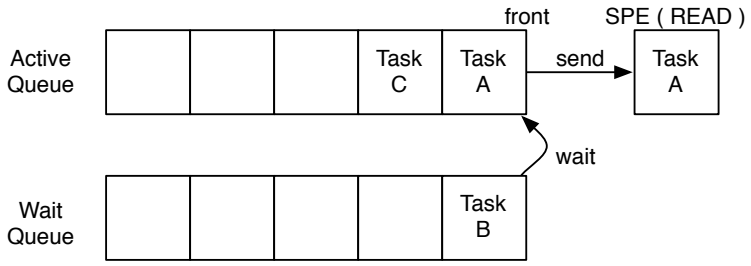

図 3.1: Task dependency 1

そして、SPE に送られた Task A は SPE で処理が行われる (図 3.2)。

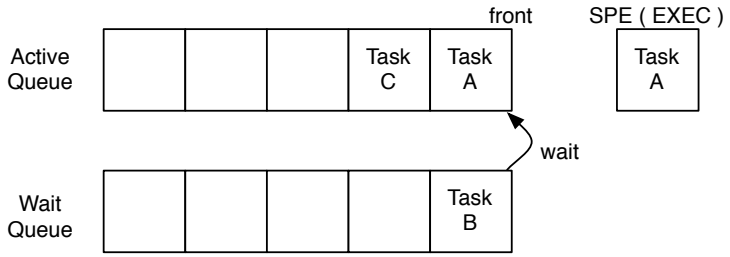

図 3.2: Task dependency 2

Task A の処理が終了すると mail で Task B へ通知される。Task B はその通知を受け取ると待ち状態が解除される (図 3.3)。

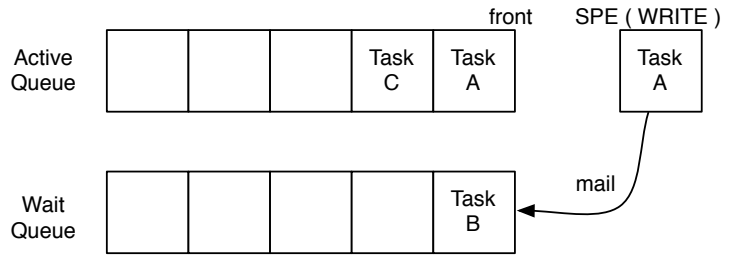

図 3.3: Task dependency 3

待ち状態が解除された Task B は、Active Queue に追加され、この図 (図 3.4) では、Task C 終了後に SPE に送られ処理が行われる。

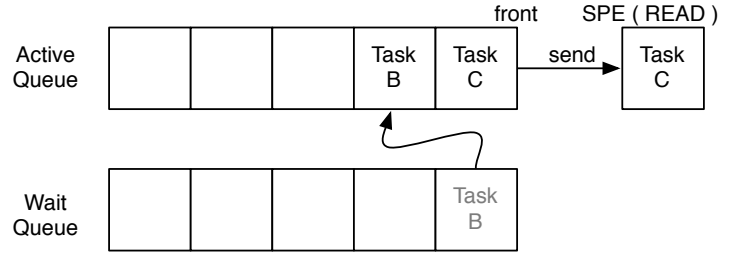

図 3.4: Task dependency 4

3.4 Task のスケジューリング

SPE は、Task を一つずつ受け取るのではなく、ある程度まとめて受け取る。それを TaskList と呼んでいる。 TaskList は各 SPE 用に PPE 側で生成される。(図 3.5) SPE は 受け取った TaskList に沿って Task を実行していき、 Task 毎に実行完了の Mail を送る。TaskList の Task をすべて実行すると、次の TaskList を要求する Mail を PPE 側に送る。

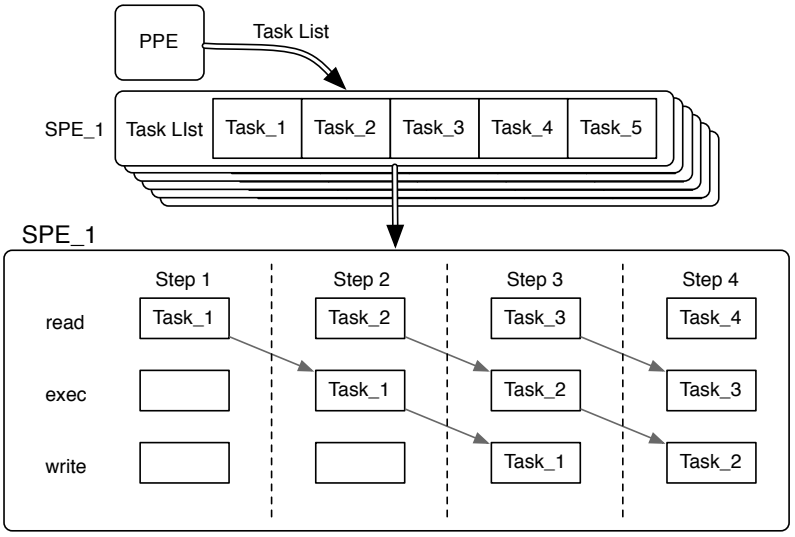

図 3.5: Task scheduler

3.5 Task の入出力

Task の入出力の API として、set_inData, set_param , set_oudData がある。

set_inData(index, addr, size) は、 データを受け取る buffer の配列番号とデータのアドレス、そのデータのサイズを引数として入力する。このデータは DMA 転送されるため、addr は 16 byte alignment が取れており、size は 16 byte の倍数である必要がある。

set_param(index, param) は、データを受け取る buffer の配列番号と 32bit のデータを渡す。set_inData で渡すには小さいデータを送るのに適している。 param はアドレスとしてではなく、値を Task オブジェクトが直接持っているので、 DMA 転送は行わない。

set_outData(index, addr, size) は、Task のデータの出力先を指定する。使用方法は set_inData と同じで、alignment, byte 数に気をつける必要がある。

3.6 SceneGraph

本研究では、ゲーム中の一つの場面 (Scene) を構成するオブジェクトやその振る舞い、ゲームのルールの集合を SceneGraph とする [?]。SceneGraph のノードは親子関係を持つ tree で構成される (図 3.6)。親子関係とは、親オブジェクトの回転や平行移動などの行列計算による頂点座標の変更が、子オブジェクトにも反映する関係のことである。これは子に対してスタックに積まれた親の変換行列を掛けることで実現できる。SceneGraph ノードは以下のようなデータと動作を持つ。

Vertex: ポリゴンオブジェクトの頂点座標

Texture: ポリゴンオブジェクトのテクスチャ座標

TextureImage: テクスチャイメージ

TransMatrix: ポリゴンオブジェクトの変換行列

Coordinates: オブジェクトの座標

Angle: オブジェクトの角度

Move: 自律的なオブジェクトの動き

Collision: 他のノードと衝突したときの動き

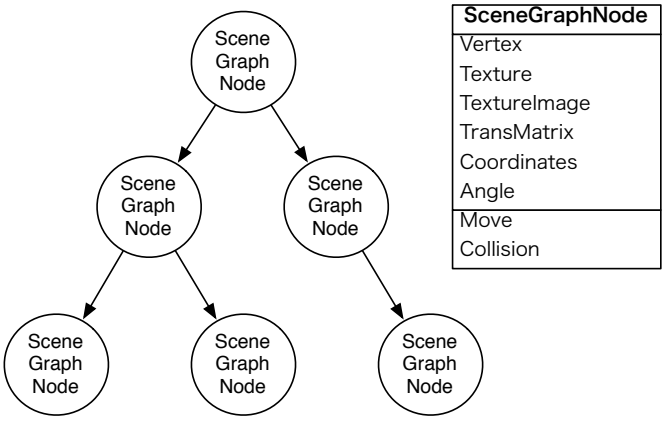

図 3.6: SceneGraph

3.6.1 Scene ポリゴンの生成

ゲーム中に登場するオブジェクトは、オープンソースの 3D モデリングツールである Blender [?] を用いる。Blender で記述したポリゴンオブジェクトを、座標やテクスチャイメージを埋め込んだ xml ファイルとして出力する。Blender には Python インタプリタがあり、xml ファイルを生成する Python スクリプトを作成している。xml には、オブジェクトの名前、ポリゴン情報、ポリゴンに対応するテクスチャ座標、テクスチャイメージがある。

xml ファイル形式を採用している理由は、Cerium が MacOSX や PS3 Linux などの複数の環境で動作することを目的としており、環境に依存しないテキスト形式でのデータ構造を構築できるからである。また、Cerium は将来的にネットワークを使用することを考えており、その際に有効なフォーマットであると考えたからである。

ネットワークを使用した例題として、Federated Linda [?] を用いた「水族館ゲーム」を赤嶺が作成した。

このゲームは、複数のクライアントのディスプレイを並べて使用する。各プレイヤーは 1 匹ずつ魚のオブジェクトが与えられ、それを自由に操作することができる。また、魚は画面の端まで移動すると、自分の画面上からは消え、接続している他のプレイヤーの画面の端から出てくる。(図 3.7) このゲームは、初めに xml ファイルをすべてのプレイヤーが共有した状態で開始される。

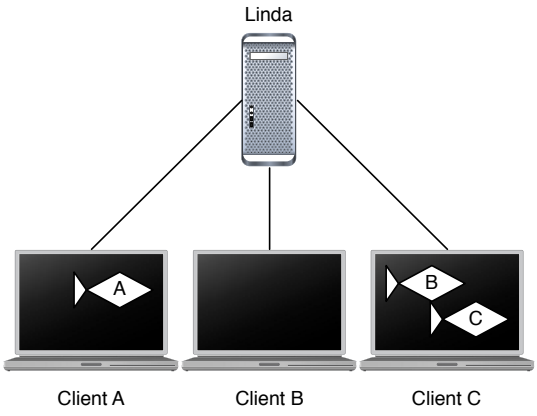

図 3.7: 水族館ゲーム

3.6.2 SceneGraph オブジェクトの生成

Cerium は生成された xml ファイルから、そのパラメータを持つ SceneGraph オブジェクトを生成する (Original SceneGraph)。ここで作られたオブジェクトはユーザには見せ

ず、ユーザが該当する SceneGraph を生成したい場合は Original を参照し、そのコピー (Copy SceneGraph) を渡す。

Original SceneGraph の情報は Cerium が配列として持っており、xml ファイルを読み込んで生成された SceneGraph を SceneGraph ID の位置に格納する。SceneGraph ID は SceneGraph に割り当てられるグローバル ID である。配列に格納する時点で登録された順に ID が割り当てられる。改良後では、同時に、 Scenegraph の名前を key, ID を value として hash に登録しているので、ユーザは ID と名前の両方から Scenegraph を生成することができる。

3.7 RenderingEngine

RenderingEngine は TaskManager から提供される Task から構成され、主に以下の 3 つの Task から成る。(図 3.8)

CreatePolygonTask: SceneGraph での処理結果を元に、モデリングデータの座標変換を行う

CreateSpanTask: 座標変換されたポリゴンを、水平な線 (Span) に分割する

DrawSpanTask: 分割された Span を実際に画面へと描画する。

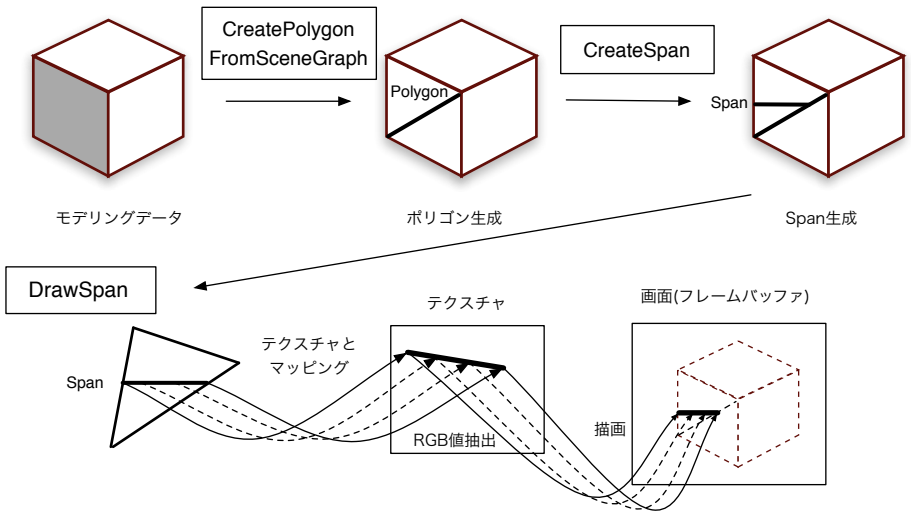

図 3.8: Rendering flow

これら Task の入出力のデータとして、PolygonPack, SpanPack がある。それぞれポリゴンの情報を格納する構造体、Span の情報を格納する構造体である。Task と合わせてその構造体について説明していく。

PolygonPack

PolygonPack は、SceneGraph から抽出されたポリゴンの集合である。CreatePolygonTask で扱う構造体である。以下に構造を示す。

```
// ポリゴンを構成する情報
typedef struct TrianglePack {
    // テクスチャ情報
    TriTexInfo tex_info;

    // ポリゴンを構成する頂点の情報
    VertexPack ver1, ver2, ver3;

    // 法線ベクトルの情報
    NormalPack normal1, normal2, normal3;

} TrianglePack, *TrianglePackPtr;

typedef struct PolygonPack {

    TrianglePack tri[MAX_SIZE_TRIANGLE];

    struct POLYGON_info {
        int size;
        int light_pos[3]; // 光源の位置
        int light_rgb[3]; // 光源の色
    }info;
    PolygonPack* next;
} PolygonPack, *PolygonPackPtr;
```

TriTexInfo は、テクスチャイメージのアドレス、幅、高さ、縮小率が格納されている

VertexPack は、ポリゴンの頂点座標と対応するテクスチャの座標が格納されている

NormalPack は、ポリゴンの法線ベクトルの情報が格納されている

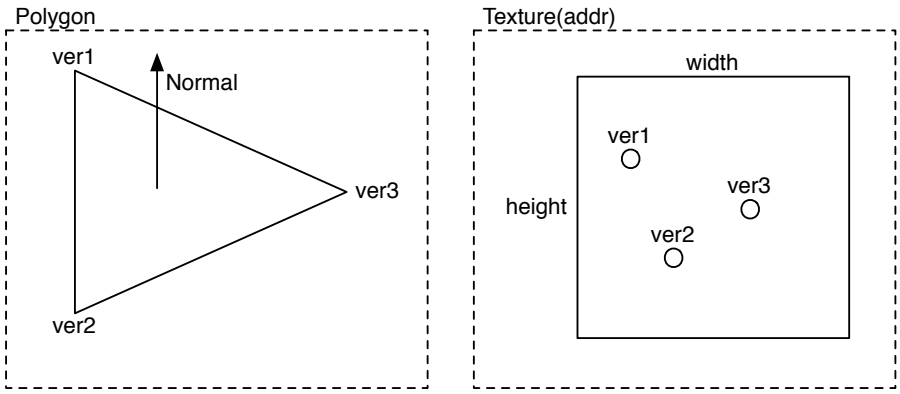

図 3.9: PolygonPack(TrianglePack)

PolygonPack は光源やテクスチャ、頂点の情報から構成される。テクスチャの縮小率に関しては 3.7.6 で詳しく説明する。

SpanPack

SpanPack は、ポリゴンから抽出された Span の集合である。CreateSpan で PolygonPack を元に SpanPack が生成される。以下に構造を示す。

```
class Span {
public:
    // テクスチャ情報
    uint32 *tex_addr;
    int tex_width, tex_height;

    // span の開始 x 座標、y 座標、長さ
    int x, y, length_x;

    // span の z 座標 (開始、終了)
    float start_z, end_z;

    // span に対応するテクスチャの座標
    float tex_x1, tex_x2, tex_y1, tex_y2;

    // span の法線ベクトル
    float normal_x, normal_y, normal_z;
};
```

```
class SpanPack {
public: /* fields */
    struct SpanInfo {
        int start;
        int size;
        int y_top;
        int light_pos[3];
        int light_rgb[3];
    } info;
    Span span[MAX_SIZE_SPAN];
    SpanPack *next;
};
```

Span は、その座標と対応するテクスチャの座標を持つ (図 3.10)

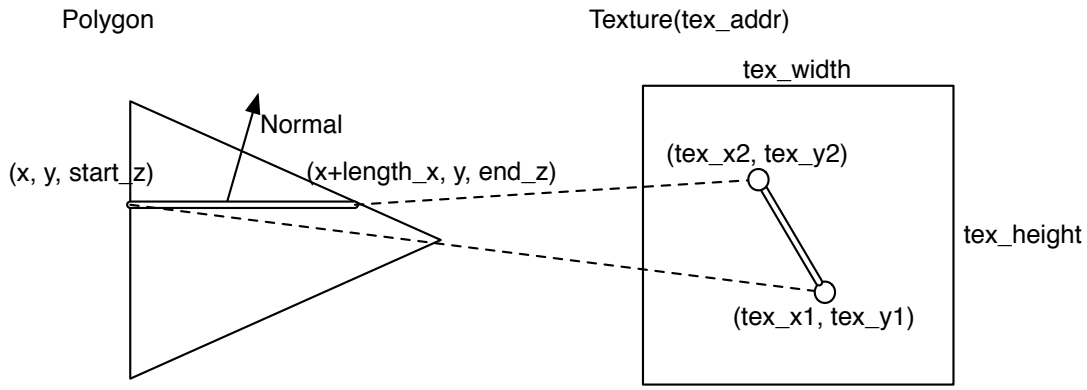

図 3.10: Span の構造

Rendering には SpanPack を用いる。ここからは、SceneGraph から Rendering までの流れを それぞれの Task を用いて説明する。

3.7.1 CreatePolygonTask

SceneGraph の move や collision を行い、各オブジェクトの変換行列を生成する。その後、SceneGraph が持つポリゴンの座標と法線ベクトルに変換行列をかけ、得られた座標を PolygonPack に格納していく。これまでは関数で書かれていたが、SPE に処理を割り振ることも視野に入れ、Task で書いている。新しい Rendering のデータ構造で述べた法線ベクトルの計算も追加している。以下がそのコードである。

```
SceneGraphPtr sg_top = sg_draw_tree;

// ポリゴンの 1 辺
xyz1[0] = sg->coord_xyz[(i+0)*3];
xyz1[1] = sg->coord_xyz[(i+0)*3+1];
xyz1[2] = sg->coord_xyz[(i+0)*3+2]*-1.0f;
xyz1[3] = 1.0f;

/* xyz2, xyz3 は省略 */

// ポリゴンの 1 辺の法線ベクトル
normal1[0] = sg->normal[(i+0)*3];
normal1[1] = sg->normal[(i+0)*3+1];
normal1[2] = sg->normal[(i+0)*3+2]*-1.0f;

/* normal2, normal3 は省略 */

// sg->matrix = 回転行列*透視変換行列
ApplyMatrix(xyz1, sg->matrix);
ApplyMatrix(normal1, sg->real_matrix);

// PolygonPack の TrianglePack  に計算した値とテクスチャの座標を格納する
PolygonPackPtr pp = new PolygonPack;
TrianglePack *triangle = &pp->tri[pp->info.size++];

triangle->ver1.x = xyz1[0];
triangle->ver1.y = xyz1[1];
triangle->ver1.z = xyz1[2];
triangle->ver1.tex_x = sg->coord_tex[(i+0)*3];
triangle->ver1.tex_y = sg->coord_tex[(i+0)*3+1];
triangle->normal1.x = normal1[0];
triangle->normal1.y = normal1[1];
triangle->normal1.z = normal1[2];

/* ver2, ver3, normal2. normal3 は省略 */

// テクスチャのアドレス、幅、高さ、縮小率を格納する
triangle->tex_info.addr   = sg->texture_info.pixels;
triangle->tex_info.width  = sg->texture_info.t_w;
```

```
triangle->tex_info.height = sg->texture_info.t_h;
triangle->tex_info.scale_max = sg->texture_info.scale_max;
```

これらの処理を全ての Scenegraph に行い、PolygonPack を生成していく。

3.7.2 CreateSpanTask

生成された PolygonPack に格納されているポリゴンから、Span を抽出していく。Span は x 軸に平行な線分を表しているため、初めにポリゴンを水平に分割して二つの三角形 (Triangle1, Triangle2) を作り、それぞれに対して Span を求める [?] (図 3.11)。

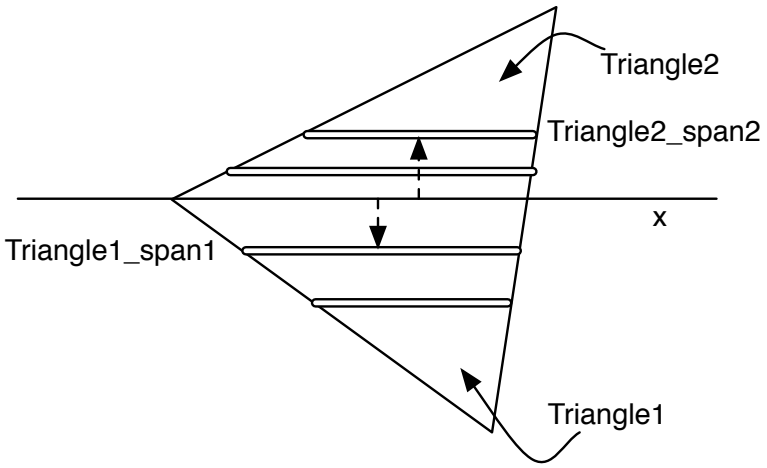

図 3.11: ポリゴンの分割と Span 生成

得られた Span を SpanPack に格納する場合、その SpanPack が持つ全ての Span の y 座標が一定範囲内に入るようにする。

Rendering する場合、画面を複数の領域に分割し、それぞれを一つの Task で担当する。Cerium では Rendering 時に Z Buffer を用いているため、同じ Task に違う領域の Span があると正常に描画できない。そこで、一つの SpanPack には決まった y 座標を持った Span だけを入れることにより、Rendering 時には独立して行うことができる。y の範囲は 8 としている (図 3.12)。

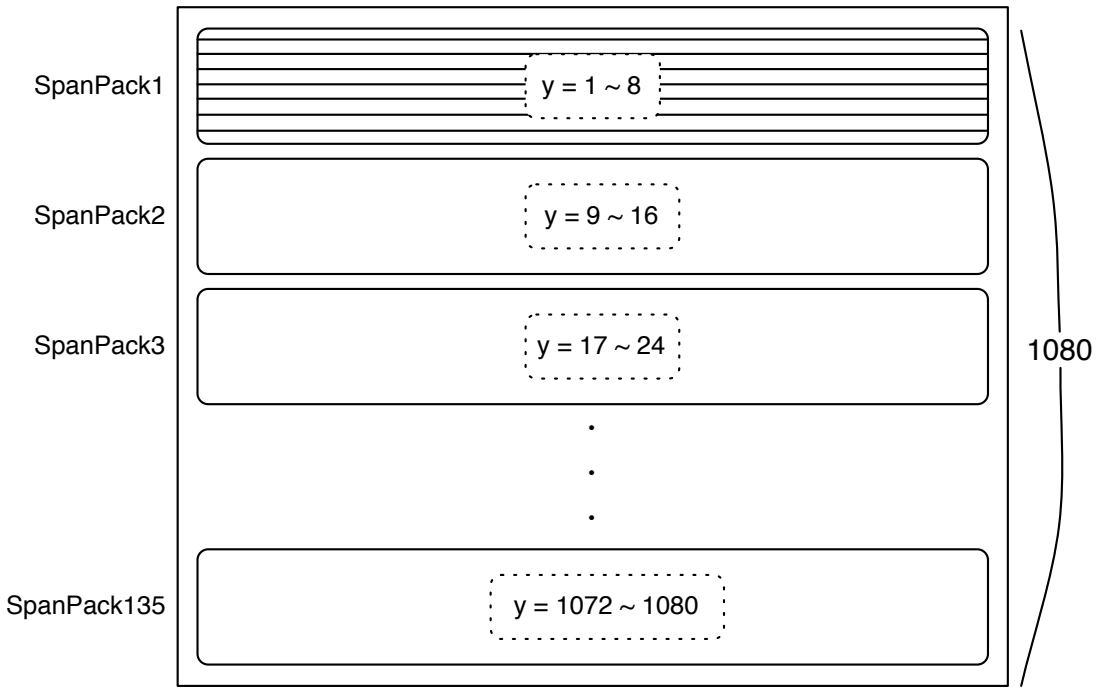

図 3.12: SpanPack の割り当て (height = 1080)

3.7.3 Rendering するための準備

画面に Rendering するためには、SpanPack の中の Span を辿り、対応する RGB 値をテクスチャイメージから取り出して frame buffer に書きこむ必要がある。しかし、SPE のメモリ領域は 256KB しかないため、Span が参照しているテクスチャイメージ全てを SPE に送ることはできない。

そこで我々は、テクスチャイメージに対して分割、Tile 管理、縮小という操作を行うことでこの問題を解決した。

3.7.4 Texture の分割

SPE のメモリ領域は 256KB しかないため、テクスチャイメージを全て転送すると動かなくなる可能性がある。そこで、そこで、テクスチャをブロックに分割し、ブロックごとに転送することにした。そのブロックを Tile と呼び、分割単位は 8 x 8 とする (図 3.13)。Tile は 図 3.13 の順番で配列に変換する。

Span が持つテクスチャのアドレスはこの配列を指しており、描画する Span 中の 1 pixel の座標から、目的の Tile を計算する。その Tile をメインメモリから DMA で持ってきて RGB 値を取り出す。

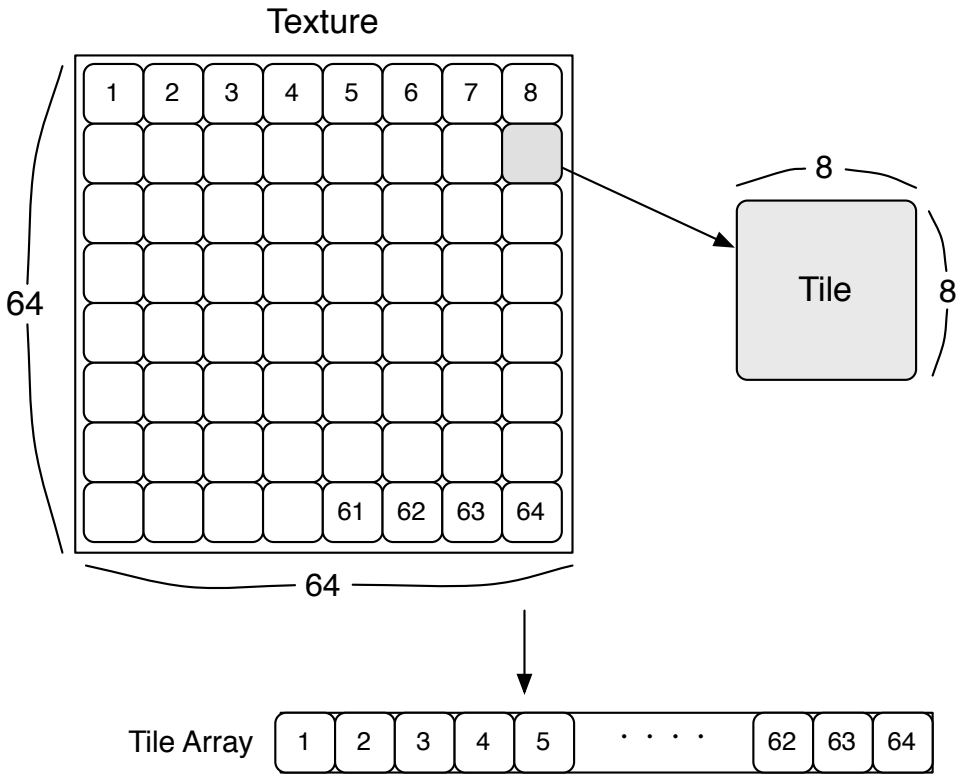

図 3.13: Texture の分割と Tile

3.7.5 SPE 上での Tile 管理

描画 Task が変わるたびに、新しい Tile を転送していては、DMA 転送のコストがかかりすぎる。そこで、すでに SPE 上に転送してある Tile を保存しておく。そして、次に描画する Span が保存してある Tile を参照する場合、新たに DMA 転送を行わずに保存してある Tile を使用する。新たに Tile を参照し且つ領域に空きが無い場合、 FIFO で Tile の入れ替えを行う。Tile の検索は、テクスチャのアドレスを key とし、Tile を value としてハッシュを用いる。

3.7.6 Texture の縮小

遠くにあるオブジェクトは小さく描画される。この場合、使用されるテクスチャは原寸大である必要がない。そこで、オリジナルのテクスチャの他に縮小したテクスチャを用意し、描画されるオブジェクトの大きさによって、使用するテクスチャを変更する。

テクスチャは Tile に分割しなければならないため、縦横ともに 8 の倍数を保つようにする。これまでは、テクスチャの縦横どちらかのサイズが最小 (8 ドット) になったとき、縮小画像生成を終了していたが、テクスチャのサイズが縦横で違う場合、長い方に合わせ

て空のデータを埋め込み 8 の倍数の正方形にするように改良した。この改良によってテクスチャの最大縮小率を正確に求めることが出来るようになったこの制約を守りながら 2 分の 1 ずつ縮小させたときの最大縮小率を求める。縮小したテクスチャ毎に Tile で分割し、すべての縮小率で求めた Tile を繋げて Tile Array とする (図 3.14)。

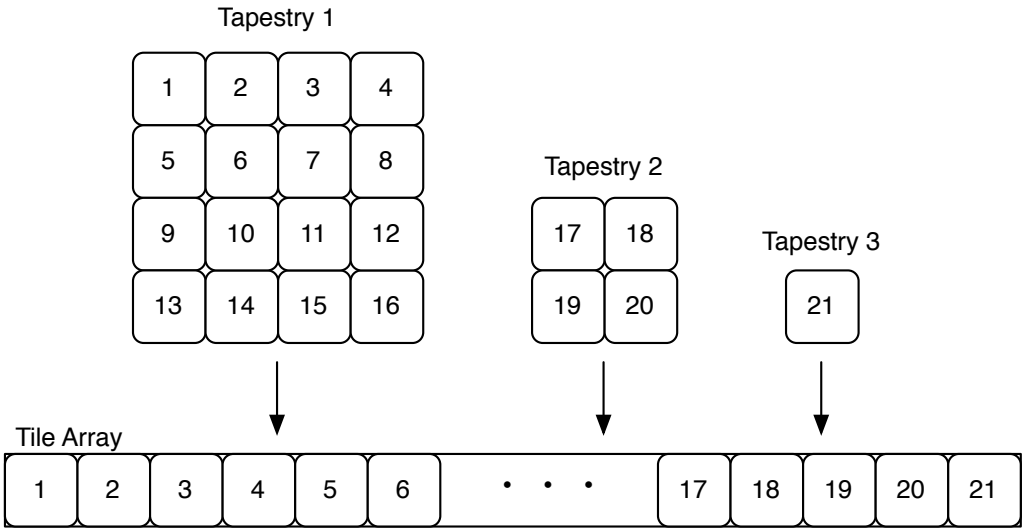

図 3.14: Tapestry の生成

縮小したテクスチャを Tapestry と呼ぶ。 SceneGraph や PolygonPack の時点ではテクスチャイメージとして Tile Array の先頭アドレスを持っている。CreateSpan の中で使用する Span の長さに適した scale を求め、その scale にあった Tapestry を span 生成たためのテクスチャとする。

3.7.4 でも話したとおり、テクスチャを全て SPE 内に送ることはできない。Tile 単位で分割し、更にハッシュを用いて SPE 内にキャッシュすることによって無駄な転送コストを減少させた。さらに Scale によって 描画に用いる Tapestry を変更することで、1 テクスチャの Tile の数を減少させることができる。必要 Tile のヒット率が上昇し、結果転送コストの削減に繋がる。

この Texture hash の効果を示す。(表 3.2) 例題として、ball bound を使用した。

表 3.2: キャッシュの効果 (ball bound)

キャッシュ	FPS	DMA 転送待ち時間	mail 待ちの割合	SPE 稼働率
なし	24.6FPS	5.5%	72.4%	22.1%
あり	27.2FPS	1.8%	76.2%	22.0%

3.7.7 DrawSpanTask

現在、PlayStation 3 の GPU にアクセスする API は公開されていないため、Cerium では Frame Buffer に直接描画する。Frame Buffer のアドレスは mmap() で取得できるため、 Task 出力として Frame Buffer を指定するか、Task 内で DMA 転送を行えば描画することができる。Mac OSX 上で動かす場合は SDL を用いて描画を行う。

Rendering は DrawSpan という Task で行う。受け取った SpanPack から Span を取り出す。Span の端から 1 pixel ずつ見ていき、その pixel の z 座標と Z Buffer を見比べ、描画するものと判断されると、対応する RGB 値を書きこむ。

また、DrawSpan は分割された画面領域の一部を担当するので、Span がその領域を越えている場合は描画しない。現在、一つの DrawSpan が描画する領域は 256x8 としている (図 3.15)。

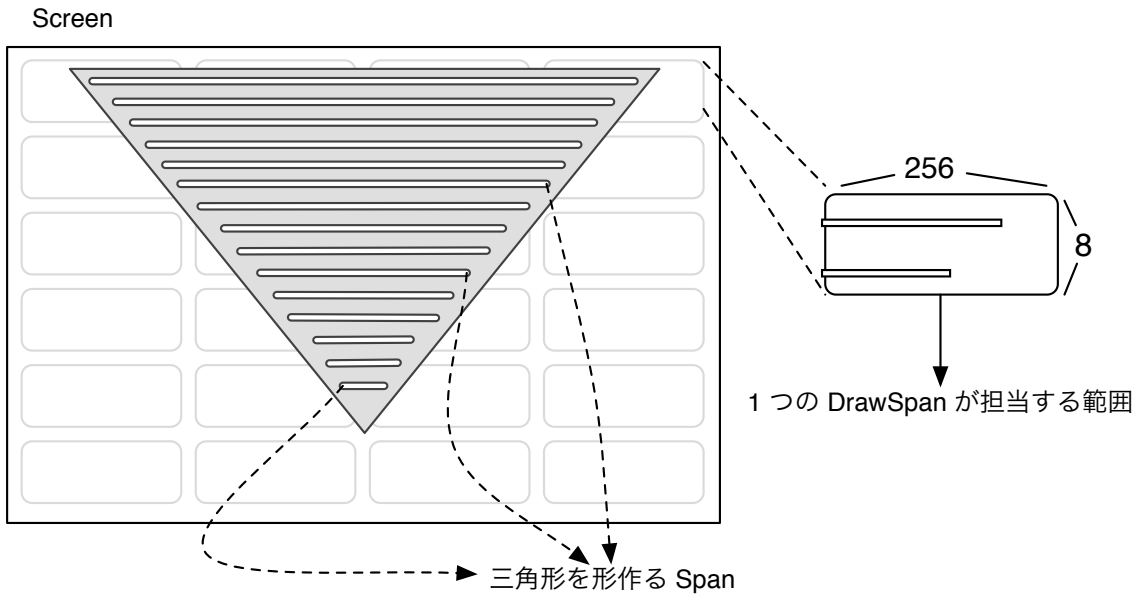

図 3.15: DrawSpan の担当領域

3.7.8 RenderingTask のパイプライン化

主に 3 つの Task から構成されいる RenderingEngine だが、それぞれの Task はバリア同期を行なっている。Cerium においてバリア同期を行う場合に、2 つの待ち時間がある。

- SPE が他の SPE の処理完了を待つ時間
- バリア同期が完了し、PPE 側で次の Task が作られる時間

この二つの時間の間SPE の処理が止まり、処理性能の低下につながる。この待ち時間を回避するためには、Task の粒度を下げる、他の SPE の処理完了を待っている SPE に、別の Task を割り当てる、等の方法がある。別の Task を割り当てるには Task の実行をパイプライン化する方法がある。そこで、特に 3 つの Task の中でも処理の重い DrawSpanTask と、CreatePolygonTask, CreateSpanTask の Task でパイプライン化を行っている。(図 3.16) 速度比較の対象として、ball bound を用いる。FPS は一秒あたりの RednderingEngine 全体の処理回数 (Frame per Scecond) であり、busy ration は SPE の稼働率である。(表 3.3)

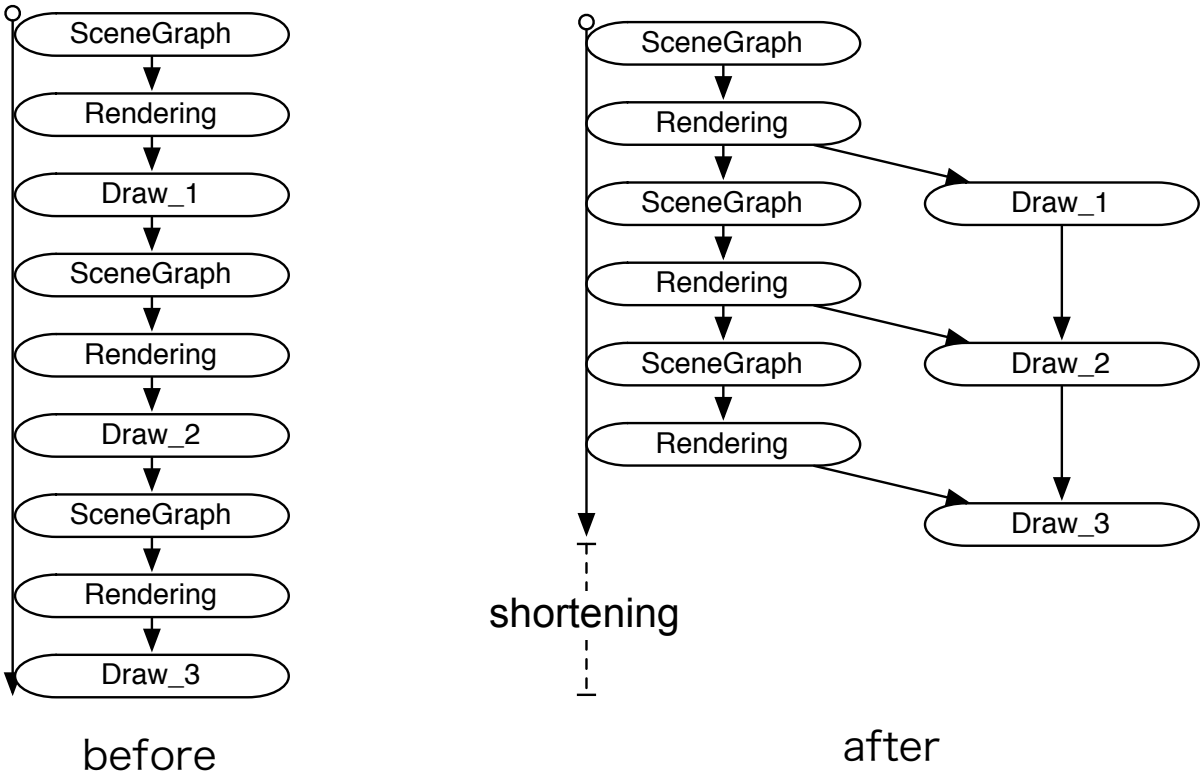

図 3.16: DrawSpan の担当領域

表 3.3: パイプライン化の効果 (ball bound)

PipeLine	FPS	DMA 転送待ち時間	mail 待ちの割合	SPE 稼働率
なし	27.2FPS	1.8%	76.2%	22.0%
あり	30.2FPS	1.8%	74.3%	23.7%

パイプライン化した結果（表 3.3)、SPE の稼働率が向上し、FPS も向上した。処理性能を維持するには、SPE はなるべく稼働させ続けなければならない。その為には処理を

Task に分割し、並列実行するだけでなく、バリア同期などで、SPE の待ち時間が発生することへ対処しないといけない。その対処の一つとしてそれにパイプライン化は有効である。

第4章　TaskManagerを使った例題

本章では、TaskManager を使った例題を紹介する

4.1　WordCount

例題とし TaskManager を使った WordCount を実装した。Task の構成は以下ののである。

1. WordCountTask
2. PrintTask

WordCountTask は、input された data を word count し、単語数と行数を output に指定された data 領域に書きこむ Task である。分割されたデータが送られてくるため、分割された前後のテキストがどうなっているかはわからない。そのため担当範囲であるデータの先頭と末尾のパラメータを単語数と行数の他に付け加える。後にそのデータを他の word count 結果と照らし合わせ、分割されたテキストを正しく整合する。PrintTask は WordCountTask 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。WordCountTask を wait する設定で、すべての WordCountTask が終了したあとに、動作する。word count 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。(図 4.1)

サイズを 100MB, 200MB としたテキストファイルを対象に、速度の測定を行った。Linux wc は PS3 上の Linux のが提供する wc コマンドを用いた結果で PPE 1 基を利用したものである。Cerium wc は SPE 6 基　を用い今回実装した word count の計測結果である。(表 4.1)

表 4.1: speed of WordCount

wc	file size(MB)	time(sec)
Linux	100	30.9
Linux	200	62.8
Cerium	100	2.2
Cerium	200	12.8

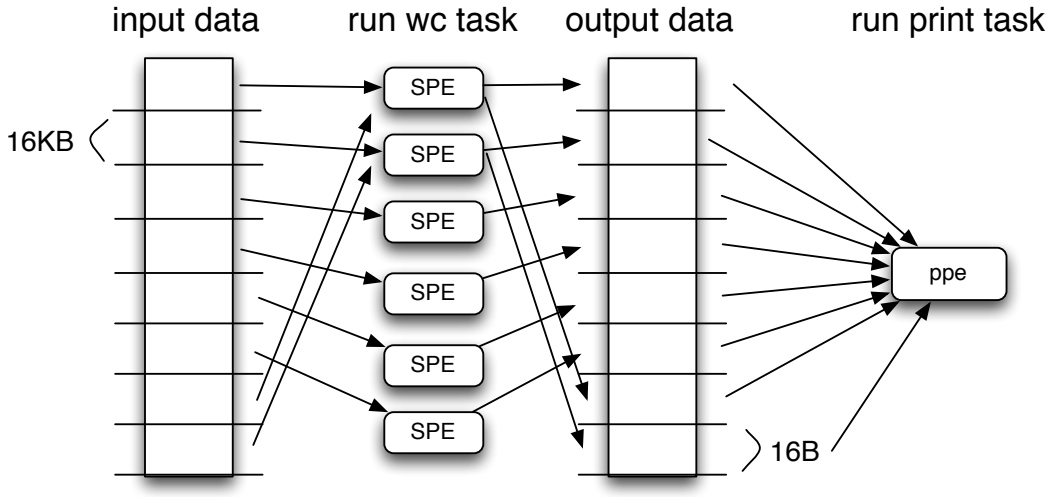

図 4.1: wordcount flow

100MB のテキストを扱う場合には、Linux と比べ 15 倍ほどの差が見られた。200MB の場合は約 5 倍ほどであるのは、PPE のメモリ容量が 256MB となっており、200MB のファイルを扱う場合には、頻繁にスワップなどが起きているため、ファイルの IO の部分でのオーバヘッドが高いと考えられる。通常の wc コマンドよりも Cerium を用いた wc が高い性能を示した。

4.2 Sort

4.3 Prime

第5章　Ceriumの改良

本章では、Cerium に行った改良について説明する。Cerium のレンダリングの例題として ball_bound を使用し、それを元に改良の効果を示していく。まず、改良前の 計測結果を示す。

表 5.1: Cerium 改良前 (ball bound)

FPS	DMA 転送待ち時間	mail 待ちの割合	SPE 稼働率
30.2FPS	1.8%	74.3%	23.7%

Mail の待ち時間が 約 70%あり、処理時間の大部分を占めている。この Mail の待ち時間を削減するために Task をグルーピングする TaskArray, ソフトウェア MailQueue の実装を行った。また RenderinEngine の Task 内には、明示的な DMA ロードが記述されており、アーキテクチャ依存となっている。そこでアーキテクチャ依存の記述を避けるために、MemorySegment を実装した。

5.1　TaskArray

本研究では新しい Task として Task Array を実装した。 Cerium では Task と TaskList 毎に Mail を通知し、SPE の Mail の待ち時間が発生する問題があった。PPE 側で実行される Task もあるなかで、SPE 側の Task を一つ一つ Task 毎に依存関係を処理しており、SPE への TaskList 要求や、次の Mail 書き込み待ちなど、Mail 通知の対応が遅れることが考えられる。Mail 通知の対応が遅れた分 SPE はアイドル状態となってしまい、稼働率が下がってしまう。この問題を解決するために、我々は Task Array を提案、実装した。 Task Array を使用することで、複数の Task をまとめる Task のグルーピングが行える。複数の Task を ひとつの Task Array として登録すると、依存関係の処理が Task Array ひとつ分で収まる。依存関係の処理には Mail 通知を行なっていたので 依存関係の処理が減少する分、 Mail 通知の回数も少なくなる。これによって、Mail 待ちによる SPE のアイドル状態の時間を減少させることができると考える。例えば TaskArray のサイズが 4、TaskList のサイズが 4、処理するべき Task の数が 16 の場合だとする。この時に TaskArray を使用すると、4 つ必要だった TaskList が 1 つで済み、さらに 16Task 分の依存関係の解決が 4 つの TaskArray 分で済む。(図 5.1) これによって Task 毎の Mail 通

知と TaskList 毎の Mail 通知の回数が合わせて 四分の一となる。このように TaskArray を用い Mail 自体の回数を減らし、待ち時間を削減できると考える。

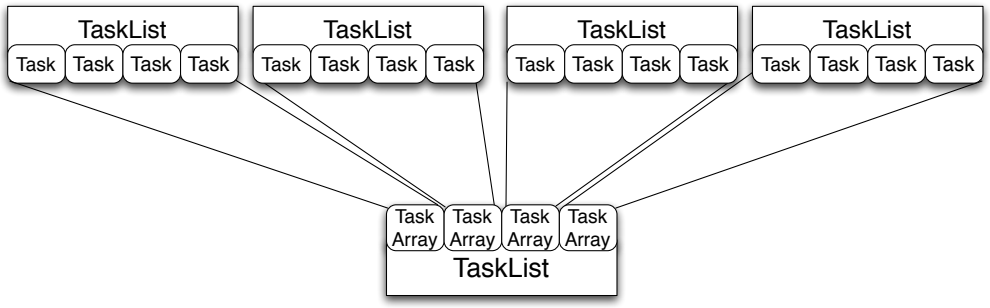

図 5.1: task array flow

以下に Task Array を用いた記述例を示す。このプログラムは Task Array に複数の同一 Task を登録して、まとめて実行するというプログラムである。

```
void
hello_init(TaskManager *manager)
{

    /* task id/task num/param/inData/outData の数を指定する*/
                                                    data_count,data_count);
    Task *t = 0;

    for(int i = 0;i<task_num;i++) {
        t = twice_main->next_task_array(Hello, t);
    }
    twice_main->spawn_task_array(t->next());
    twice_main->set_cpu(SPE_ANY);
    twice_main->spawn();
}

static int
run(SchedTask *s,void *rbuf, void *wbuf)
{
    s->printf("Hello World\n");
    return 0;
}

// 実行結果
```

```
% ./hello -task 6
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

プログラムに用いられてる新しい API について説明する。

create_task_array: 同一の Task を複数持つことのできる Task Array を生成する。この際に、登録する Task の ID, Task の数、設定する param, input data, output data の数を指定する。

next_task_array: Task Array に Task を実行順序を定める。実行順序は next_task_array を行った順になる。

spawn_task_array: Task Array の中のすべての Task が書きこまれたかどうかをチェックする。TaskArray では指定された Task の数と、実際に登録された Task の数が同一か計算し、異なってた場合にはエラー文を返す。これは DMA ロードの際のサイズを合わせる為、正確に数を合わせなければならない。

この TaskArray を RenderingEngine の DrawSpanTask に適応させた。レンダリングの例題は ball bound を用いた。その効果を示す。wor(表 5.2)。

表 5.2: Task Array の効果 (ball bound)

TaskArray	FPS	DMA 転送待ち時間	mail 待ちの割合	SPE 稼働率
未使用	30FPS	1.8%	76.2%	22.0%
使用	32.2FPS	2.5%	66.7%	30.8%

mail の待ち時間が 約 10% 削減され、FPS の向上があり、TaskArray による効果が見られた。

5.2 MailQueue

Task の完了通知や、アイドル状態の SPE に Task のリストの情報を通知するために Mail を使用している。Cell の設計では、その通知に使われる SPE からの書き出し Mail の Queue のサイズは 1 である。ハードウェアの設計として、Mail を書き出す場合、前回

の Mail が PPE 側から読み込まれていない場合に、PPE 側の読み出しで Mailbox が空き状態になるまで、SPE はアイドル状態となる。すると PPE 側の Mail 読み出し処理が追いつかない場合には、SPE 側に余計な待ち時間が入ってしまう。そこで、ソフトウェア MailQueue を実装した。ハードウェアの Mailbox への書き込みができない場合には、ソフトウェア MailQueue へ書き出し、Mail の書き出し待ちをなくす。代わりに割り当てられた TaskList をすべて処理したあとに、MailQueue に残っている Mail をすべて書き出す処理を行う。

MailQueue の大きさはメモリ容量の限り自動で拡張される。以下に ball bound の例題での MailQueue の有無における測定結果を示す (表 5.3)。

表 5.3: MailQueue の効果 (ball bound)

MailQueue	FPS	DMA 転送待ち時間	mail の待ち時間	SPE 稼働率
なし	32.2FPS	2.5%	66.7%	30.8%
あり	41.7FPS	3.3%	56.8%	40.0%

Mail の待ち時間の割合が減少し、8FPS の向上がみられた。ソフトウェア MailQueue によって Mail の書き出しタイミングを変更することで、Mail の待ち時間を削減することができた。PPE 側では、SPE からの Mail の確認は一度の ループ文で行なっている。Mail を確認しおえ、そのループ文を抜けてしまうと、次に Mail を確認するまでに PPE 側の Task 処理が挟まれる。よって、SPE 側の Mail 通知は一度に多く行った方が、PPE 側の Mail 確認がスムーズに行われ、結果 SPE の稼働率向上に繋がると言える。ソフトウェア MailQueue では Mail をキューイングし一度に書き出すので、この点でも効果がある。

5.3 MemorySegment

CreateSpanTask 内では明示的に DMA 転送命令を行なっている。これは処理するデータ構造上の理由である。しかし、DMA 転送は Cell のアーキテクチャに依存した機能である。また Task に登録された input data と output data は自動的に TaskManager によってパイプライン化されるが、明示的に DMA 転送を行う場合には、手動でのパイプライン処理を行う必要がある。そこで、Task 内でのデータの入出力の機能を 抽象化する MemorySegment を実装した。MemorySegment によって DAM 転送は隠蔽され、アーキテクチャ依存の記述を避けることができる。またメモリ操作も抽象化される。

CreateSpanTask 内の明示的な DMA 転送の例を示す。

```
create_span() {

   tmp_spack = spack;
   spack = send_spack;
```

```
    send_spack = tmp_spack;

    smanager->dma_wait(SPAN_PACK_STORE);
    smanager->dma_store(send_spack, (memaddr)spackList[prev_index],
                        sizeof(SpanPack), SPAN_PACK_STORE);

    smanager->dma_load(spack, (memaddr)spackList[index],
                        sizeof(SpanPack), SPAN_PACK_LOAD);

    prev_index = index;
    smanager->dma_wait(SPAN_PACK_LOAD);

    span_calc(spack);

}
```

CreateSpanTask では SpanPack というデータ構造を扱う。MainMemory から SpanPack を DMA ロードし、ロードした SpanPack に span_calc() で計算した Span を格納し また MainMemory へと書きだす。変数 tmp_spack, spack, send_spack はそれぞれ SpanPack へのポインタであり、それを入れ替えながら DMA 転送行うことで、メモリを節約を行なっている。それぞれの API の説明を行う。

dma_wait: 指定された dma タグの dma 転送の完了を待つ。

dma_store: 指定された LS のアドレスから、指定された サイズ分 の領域を MainMemory に書き出す。その際に dma_wait で用いる dma タグも指定する。例の場合では SPAN_PACK_STORE である。

dma_load: 指定された MainMmory のアドレスから 指定された サイズ分のデータを LS へ DMA ロードを行う。その際に dma_wait で用いる dma タグも指定する。 例の場合では SPAN_PACK_LOAD である。

dma タグは、enum によって整数が定義されている。次に上記のコードを MemorySegment API に変更した例を示す。

```
create_span() {

    smanager->wait_segment(span_put_ms);

    span_put_ms = span_get_ms;
    smanager->put_segment(span_put_ms);
```

```
    span_get_ms = smanager->get_segment((memaddr)spackList[index], span_ml);
    smanager->wait_segment(span_get_ms);

    prev_index = index;
    spack = (SpanPackPtr)span_get_ms->data;

    span_calc(spack);

}
```

変数 span_get_ms, span_put_ms はそれぞれ MemorySegment のデータ構造を指すポインタである。この MemorySegment というデータ構造で Task 内部での 必要なデータのやり取りを行う。MemorySegment の API の説明を行う

wait_segment: 指定された MemorySegment の処理完了を待つ。それは書き出し、読み込みどちらも当てはまる。dma_wait に相当する。

put_segment: 指定された MemorySegment を書きだす。書き出し先のアドレスは、MemorySegment の生成時に登録されている。

get_segment: 指定されたアドレス への書き出しを目的とした、MemorySegment を取得する。MemorySegment は予め MemorySegmentList によって管理されており、その List から使用可能な Segment を得ることができる。

SPE の LS は 256KB となっていて、一般的な CPU のメモリ容量と比べると小さい。LS 内部での頻繁な メモリのアロケーションなどは LS を圧迫することになる。そこで、一度確保したメモリを使いまわすことが必要である。MemorySegment は必要な構造体をはじめに List として登録する。List は LRU で管理し、指定されたメモリ容量を超えた場合には 最もアクセスされた時間が古いデータから順に開放され、新たなデータを List に加える。以下にそのコードを示す

```
    ml = smanager->createMemList(sizeof(SpanPack), SPANPACK_SEGMENT_NUM);
    smanager->global_set(GLOBAL_SPANPACK_LIST, (void *)ml);
```

コード上にある API の説明を行う。

createMemList: 指定されたサイズの、指定され数だけ メモリを確保し、そのメモリを List を生成する。

global_set: どの Task からでも 指定された アドレスを呼び出すことができる。Task 共通のメモリ空間を扱う時に用いる。

このように MemorySegment はメモリの List を持ち、それを抽象化した API によって LS のメモリを使い回しながら、Task 内でのデータ転送を可能にする。また 明示的な DMA 転送 API を隠蔽することができるので、他の分散メモリ環境などでの汎用性が期待できる。DrawSpanTask の他に、Texture cache でも使用している。

5.3.1 改良のまとめと比較

本研究で行った改良を加えた計測結果をまとめる (表 5.4)

表 5.4: 本研究の改良効果 (ball bound)

	FPS	DMA 転送待ち時間	mail 待ちの割合	SPE 稼働率
改良前	30.2FPS	1.8%	74.3%	23.7%
改良後	41.7FPS	3.3%	56.8%	40.0%

Mail の待ち時間、DMA 転送の待ち時間がともに削減され、稼働率と FPS の向上が見られた。

本研究で行った改良の加え、Cerium 開発後からの改良である、Task のパイプライン化、Texture cache の使用の効果をまとめる (表 5.5)

表 5.5: Cerium の改良の効果 (ball bound)

	FPS	DMA 転送待ち時間	mail 待ちの割合	SPE 稼働率
改良前	24.6FPS	5.5%	72.4%	22.1%
改良後	41.7FPS	3.3%	56.8%	40.0%

Cerium 開発後からの改良の結果、FPS が 17 上昇し、稼働率が 18%向上した。

OpenGL との比較

OpenGL (Open Graphics Library) とは、Silicon Graphics 社が開発した、3D グラフィックス処理のためのプログラミングインターフェースである。上記で紹介した SuperDandy を Task を用いない OpneGL 上で動作するバージョンを用意して、Cerium と性能を比較してみた。OpenGL は PPE だけで動作している。Cerium は今までの改良をすべて加えたもの。

コアを 1 つ用いている OpenGL 版に比べて、Cerium では 2.9 倍の性能向上が見られた。SPE を活用し、改良によって待ち時間の削減ができ、性能の向上ができた。

表 5.6: シューティングゲーム「dandy」の性能比較 (OpenGL, Cerium))

	OpenGL	Cerium	性能差
dandy	17.5 FPS	49.5 FPS	2.9 倍

第6章　デバック

並列プログラムの特徴として、デバッグが難しいことが挙げられる。実行が非決定的 (同じ状態で実行しても結果が異なる) な場合があり、バグの状態を再現することが難しい。また、個々の Core 上のデータを調べる必要があり、デバッガが複数の Core を取り扱えることが必須である。Cell の場合、動作している複数 の SPE の一つに対して gdb で breakpoint を掛ければ、PPE や他の SPE も同時にストップするが、それら全ての CPU を手動で管理するのは時間がかかる。また、PPE と SPE ではメモリ空間が違うため、SPE から直接 PPE のデータを見ることができない。Cerium での開発工程は、

1. C によるシーケンシャルな実装
2. 並列実行を考慮したデータ構造を持つ実装
3. コードを分割し、シーケンシャルに実行する実装
4. 分割したコードを並列実行する実装

となっている。逐次的な実行で正常な動作を確認した後、並列に実行した場合に正常な動作をしない場合がある。特に Cell 特有の問題としてデータ構造が DMA 転送に合っていないこと。つまり、DMA 転送される際に、16 アライメントに設定されていないか、データのサイズが 16 の倍数になっていない場合に注意しなければならない。また Cerium ではコードがそれぞれ PPE 用と SPE 用が別個に存在するので、互いのコードが等価であるかもチェックしなければならない。一つのコードに統一しても良いが、別個で対応したい問題がでた時に対応してる。バグの例として、本来 SPE で動くべき Task が PPE で動作するなど、Task のスケジューリングでのバグ、Task の設定の際のバグなどがある。

第7章　結論

7.1　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 の改良を行った。Cerium は並列プログラミングをサポートする TaskManager、ソフトウェアレンダリングを行う RenderinEngine, ゲームのシーンを記述していく、SceneGraph から構成されている。学生実験で行う Cell 上でのゲーム作成を考慮にいれた並列プログラミングフレームワークである。Ceirum 開発後に学生実験での使用や、例題を実装していく上で、仕様通りの稼働率がでないことや、アーキテクチャ依存の記述が避けづらいことなど、フレームワークとしての信頼性が十分でないことが明らかになった。特に Cell 上での実行の場合には、SPE との通信の待ち時間が約 70%と、処理性能に関わっている。そこで SPE と PPE の通信部分である Mail について、TaskArray の実装と使用、ソフトウェア MailQueue の実装をし改良を行った。その結果 FPS の向上、Mail 待ち時間の減少に効果が見られた。また、Task 内でのアーキテクチャ依存の記述を　MemorySegment によって、避けることに成功した。また MemorySegment を用いることでメモリ管理を抽象化することができる。その他の先行研究の改良として、Task のパイプライン化、Texutre のキャッシュなどがある。これらの改良によっても、FPS の向上や、DMA 転送の待ち時間が改善された。Cerium の改良を重ね、結果として約 17 FPS の向上に成功した。また Mail の待ち時間は約 18%削減できた。

7.2　今後の課題

各コアの稼働率の向上のためには、さらに Mail の待ち時間を削減する必要がある。Mail の通信の待ち時間は内訳は

PPE の対応が遅れる

SPE が他の SPE を待つ

の二つの要素がある。PPE の対応が遅れる場合には PPE 側に処理するべきプログラムがあり、Mail の対応が遅れる原因がある。そのため 制御用の PPE は処理するべきプログラムは排除し、Mail 通知の対応や、Task の割り振りを専門に PPE が行うことによって、SPE の稼働率向上を果たせると考える。また バリア同期を行うと 割り振られた処理によっては 他の SPE よりも早く処理が終了し、他の SPE を待つ時間が生じる場合がある。

7.2.1 プログラムの Task 化

PPE での Task 管理以外の処理を排除するため、具体的には、プログラムの Task 化 を行い Task の部分を SPE で処理する方法がある。現在 RenderingEngien 部分のほとんどが Task 化されているが、ScenenGraph 部分は Task 化されていない。SceneGraph 部分の Task 化が必要である。

SPE の LS の利用方法

Task の粒度

バリア同期の際に 他の SPE を待つ時間を削減するために、Task はなるべく粒度を細かく設定することで解決できると考える。粒度を細かくすることは、各 SPE への均等な Task の分散のために必要であり、並列化部分の特定にも繋がる。

7.2.2 自動的な依存関係の解決

7.2.3 DataSegment

7.2.4 CodeSegment

謝辞

ありがとうございました。

参考文献

[1] 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.

[2] 神里晃, 宮國渡, 杉山千秋, 河野真治. C から cell アーキテクチャを利用した cbc への変換. 電子情報通信学会 VLSI 設計技術研究会, March 2008.

[3] 神里晃, 河野真治. Continuation based c による ps3 cell のシミュレーション. 情報処理学会システムソフトウェアとオペレーティング・システム研究会, May 2006.

[4] 神里晃. Cell を用いたゲームフレームワークの提案. Master's thesis, 琉球大学理工学研究科情報工学専攻, Feb 2008.

[5] 宮國渡. Cell 用の fine-grain task manager の実装. Master's thesis, 琉球大学理工学研究科情報工学専攻, Feb 2009.

[6] 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, April 2008.

[7] 杉山千秋. Scenegraph と statepattern を用いたゲームフレームワークの設計と実装. 琉球大学工学部情報工学科 平成 19 年度卒業論文, 2008.

[8] 赤嶺一樹, 河野真治. Meta engine を用いた federated linda の実験. 日本ソフトウェア科学会第 27 会大会 (2010 年度), Sep 2010.

[9] 多賀野海人. Cell task manager cerium における task を用いたパイプラインの改良. Master's thesis, 琉球大学理工学研究科情報工学専攻, Feb 2011.

[10] 金城裕, 河野真治. Fine grain task manager cerium のチューニング. 日本ソフトウェア科学会第 27 会大会, Sep 2010.

[11] 金城裕, 河野真治. 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, Apr 2011.

[12] 金城裕, 河野真治. Cerium における datasegment api の設計. 日本ソフトウェア科学会第 28 会大会, Sep 2011.

発表履歴

-