学生用 PS3 GameFrameWork Cerium での Pipeline ストールの検出

小林佑亮[†1]　河野真治[†1]
多賀野海人[†1]　金城裕[†1]

我々は PlayStation3 上でのゲームプログラミングをサポートする Cerium Engine を開発した。Cerium におけるゲーム開発ではオブジェクトの描画や動作を Task という単位で管理しており、この Task を SPE に送り、Pipeline で実行することによってプログラムの並列化を図っている。この Pipeline におけるストールによって実行速度の低下が発生することが分かっている。本研究ではこのストールを検出することを考える。

Detecting pipeline stall on PS3 GameFrameWork Cerium

Yusuke Kobayashi,[†1] Shinji Kono,[†1] Kaito Tagano[†1]
and Yutaka Kinjo[†1]

We developed Cerium Engine which support Game Programming on PlayStation3. Cerium managed drawing objects and movement in unit called Task. By sending SPE a Task and executing on pipeline,we attempt parallelize program. Stalling of this pipeline reduce execution speed. Ins this paper, we detect this stall.

†1 琉球大学 理工学研究科 情報工学専攻 並列信頼研究室
Concurrency Reliance Laboratory, Information Engineering Course, Faculty of Engineering Graduate School of Engineering and Science, University of the Ryukyus.

1. はじめに

当研究室ではこれまで家庭用ゲーム機上でのゲームプログラミングの開発を行ってきた。過去には PlayStation や PlayStation2、GameBoyAdvance を用いており、現在は PlayStation3(以下 PS3) で 動作するゲーム開発を行っている。PS3 では Fedora や Yellow Dog Linux といった Linux O を動作させることができるので (現在の公式のサポートは終了している)C や C++ といったプログラム言語を用いて Linux 上でプログラミングすることが可能となっている。しかし、PS3 の Architecture である CellBroadbandEngine は複数の SPE を使用する並列プログラミングが求められている。そこで我々は Cell のような ManyCoreArchitecture を用いた、並列プログラムの開発をサポートするフレームワークとして FineGrainTaskManager を開発した。この TaskManager を用いたゲーム開発フレームワークが Cerium である。

Cerium を用いたゲーム開発ではプログラムの動作を Task という単位で管理し、この Task と必要なデータを複数の SPE に転送して処理させる。転送された各 TASK は並列に処理され、Pipeline で実行される。本研究では Cerium に対して様々なテストを行い、Cerium の チューニングやゲームプログラミングを補助するテスト手法を提案する。

2. Cell Broadband Engine

CellBroadbandEngine は SCEI と IBM、東芝によって開発された CPU である。2 thread の PPE(PowerPC Processor Element) と 8 個の SPE(Synergistic Processor Element) からなる非対称なマルチコアプロセッサであり、高速リングバスである EIB(Element Interface Bus) で構成されている。PS3 Linux では 6 個の SPE を使うことが出来る。(図 1)

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。

2.2 SPE(Synergistic Processor Element)

SPE には 256KB の Local Store(LS) と呼ばれる直接参照できるメモリ領域があり、バス

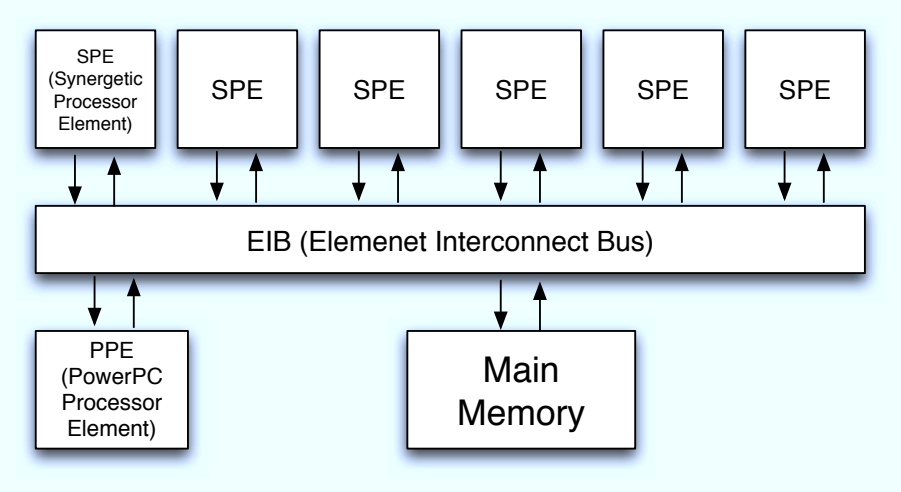

図 1 Cell Broadband Engine

に負担をかける事な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。

3. Cerium Engine

Cerium は 独自の Rendering Engine と Scene Graph、Task Manager の 3 つによって構成される。ゲーム中のオブジェクトの振る舞いやルール、Polygon Data は SceneGraph によって管理され、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。Cerium は C++ で実装されており、画像の読み込みや入力デバイスは SDL[⋆1] を用いて行っている。

3.1 Scene Graph

Cerium ではゲーム中の一つの場面 (Scene) を構成するオブジェクトの頂点座標やテクスチャー、そしてその振る舞いの集合を SceneGraph としている。SceneGraph のノードは親子関係を持つ tree で構成される。親子関係とは、親オブジェクトの回転や並列移動等の

⋆1 Simple DirectMedia Layer。グラフィックやサウンドなどの API を提供するフリーのライブラリ。

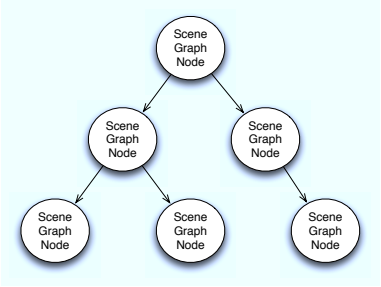

図 2 Scene Graph Tree

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。(図**??**)

3.2 Task Manager

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する。Task は主に PPE 内で生成され、処理する内容と必要なデータを設定する。実行可能になった Task から各 SPE 割り当てられ、Task を受け取った SPE はその情報を元に処理を実行する。

3.3 Rendering Engine

現在、PlayStation3 の GPU にアクセスする API が公開されていないため、Cerium の Rendering Engine では、オブジェクトの描画を Cell で並列処理している。描画処理を段階別に以下の 3 種類の Task に分けている。

- SceneGraph が持つ Polygon の座標から、実際に画面に表示する座標の計算を行い、それを格納した PolyPack を生成する SceneGraph2PolygonPack Task
- PolygonPack から同じ Y 座標を持つ線分の集合である SpanPack を生成する PolygonPack2SpanPack Task
- SpanPack を Texture を読み込みながら Z Buffer を用いて描画する DrawSpan Task

この 3 つの Task が Pipeline 状に動作することにより画面の描画を実現している。

4. Cerium の Task

4.1 Task ID

Task ID とは、各 Task に割り振られたグローバル ID である。通常の逐次型プログラム

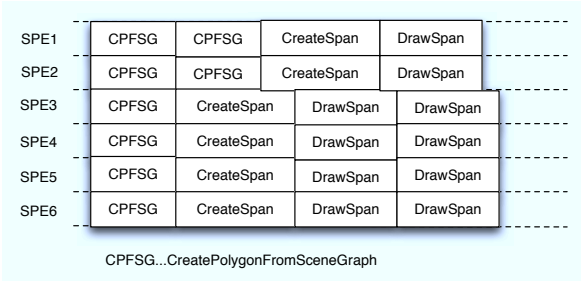

図 3 RenderingEngine の Pipeline

```
//SPE 1 で実行する
task1->set\_cpu(SPE\_1);

//SPE のどれかで実行する
task2->set\_cpu(SPE\_ANY);

//PPE で実行する
task3->set\_cpu(PPE);
```

図 4 set_cpu による実行 CPU の選択

では Task を選択して実行する場合、その関数ポインタを指定して実行すればいい。しかし Cell の場合、PPE と SPE 上ではアドレス空間が異なるため、SPE 上で実行される Task を PPE から直接アドレス指定することはできない。そこで、Task 毎に ID を割り振り、その ID から、SPE 上にある Task が格納された配列を参照して実行する。

4.2 Task を実行させる CPU の選択

Task Manager の **set_cpu()** により、Task をどの CPU で実行するか選択することが出来る。(図**??**)

これにより、PPE 内でも Task を実行することが可能になるため、環境依存によるプログラム変換は Task の部分だけとなる。

4.3 Task で入出力されるデータ

Task に渡すデータとして、**add_inData** と **add_param** がある。**add_inDara(addr, size)** は、Task に渡すデータのアドレスと、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 命令により転送される。**add_param(param)** は、Task に 32 bit のデータを渡す。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 命令による転送は行わない。

4.4 Task 間の共用領域

各 Task は独立して動作するため、使用するメモリ領域も他の Task と干渉することは無い。しかし、処理によっては Task 間で同じデータを使用する場合がある。その場合、Task 毎に同じデータを持つよりも、LS 上に一つロードしてそのデータを各 Task が参照することが望ましい。共用領域を allocate するには、**global_alloc(int id, int size)** を使う。id は allocate した領域に割り振る番号で、size が allocate するバイト数を表す。allocate した領域のアドレスは **glbal_get(int id)** で、global_alloc() 時に指定した id を渡すことで取得できる。 global_alloc() した領域は、TaskManager 側で解放することな無いのでユーザが **global_free(int id)** を実行する必要がある。共用領域は各 SPE 上に置かれるので、違う SPE の領域を参照することはできない。

4.5 Task の Dependency

Task に依存関係 (Dependency) を持たせるには **wait_for** を使用する。

```
//task2 は task1 が終了してから開始する
task2->wait_for(task1);
```

上記の場合には task1 は Active Queue へ、task2 は Wait Queue へ格納される。Active Queue に格納された Task は各 SPE に送られ処理される。終了した Task から PPE の Task Manager へタスク終了のコマンドを発行する。コマンドを受け取った Task Manager は Wait Queue を調べ、依存関係を満たした Task から Active Queue に移し、SPE へ送る。(図 5)

4.6 Task 終了時に実行される関数

set_post を使うことで Task が終了した際に PPE で実行される関数と、その引数を指定できる。(図 6)

set_post により、ユーザ側でも Task が終了したということを検知できる。また、大量の Task に依存関係を設定する場合、一度に bf create_task で生成し、**wait_for** で繋げるというのは難しい。その場合、ある一定数の Task だけ生成しておき、**set_post** を使って終了したことを確認して、その中で新たに Task を生成していくという方法により依存関係を設定することができる。

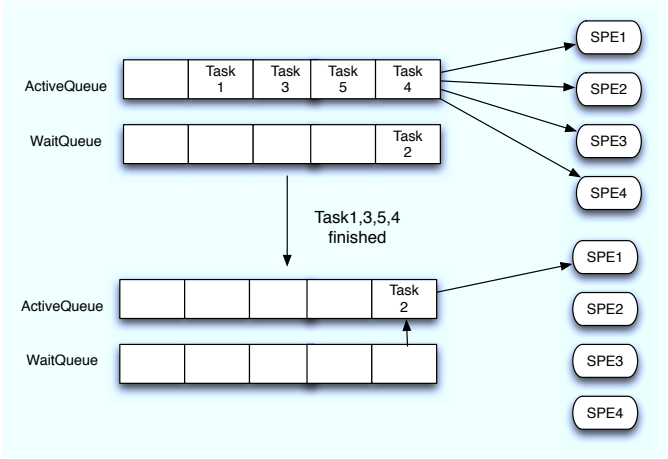

図 5 Task の Dependency

```
int data = 3;
task->set_post(func1, (void*)data);

void
func1(void *data)
{
  printf("func1: data = %d\n", (int)data);
}
//実行結果
func1: data = 3
```

図 6 set_post による関数の設定

5. Super Dandy

SuperDandy は当研究室が PlayStation でのゲーム開発を行っていた時期に開発されたシューティングゲームである。**SuperDandy** は開発する環境が変わる度に移植されており、過去には PlayStation2、MacOSX 用も作られた。今回、新たに MacOSX バージョンをベースとして Cerium で動作するバージョンを作成した。SuperDandy が伝統的に移植されてきた背景には、ある程度のボリュームのあるゲームであること、動作結果を過去の環境と比較することで新たな環境のチューニングが行えることが挙げられる。

5.1 Super Dandy の Cerium への移植

Cerium ではオブジェクトの管理は SceneGraph を用いて行っているが、Super Dandy は独自にオブジェクトの管理を行なっている。また PutSprite という描画用の関数が存在する。この関数は OpenGL[⋆2] を使用しており、XY 座標、テクスチャの指定を行うことによってオブジェクトの描画を行うことができる。また、PutSprite の他にオブジェクトの拡大縮小が可能な PutSpriteEx が存在する。Super Dandy を Cerium へ移植する場合には、この PutSprite, PutSpriteEx と同名の関数を用意し、その中で SceneGraph tree への登録を行うことで Cerium 上への移植に成功した。また、この方法で移植することによって Super Dandy 内で設定された細かなパラメータを再利用でき、旧バージョンとの比較によって Cerium の機能の洗い出しをすることができる。具体的にはテクスチャの表示のズレ、光源処理のチューニングなどである。また、元の PutSprite,PutSpriteEX を使うことにより旧バージョンを動かすことができる。これにより異なる環境での動作の確認が容易に行える。

5.2 Super Dandy on PlayStation3

現在、この Super Dandy を Task ベースで動かすための移植作業を行っている。Super Dandy を Task 化して PlayStation3 上で動かす場合、Cell Broadband Architecture を意識する必要がある。Super Dandy は move(動作) や collision(衝突判定) を関数ポインタで持っているが、それを Task 化して並列で動かす場合、PPE と SPE のメモリ領域は異なるので関数ポインタのアドレスは使用できない。Cerium では Task を ID で管理しているのでそれを使用して move や collision を行うのが望ましい。Super Dandy を始め、ゲームには相互に干渉するパラメータが多数存在する。共通のパラメータを必要とする Task 群を global_alloc を使用して同じ Task 内で処理させたり、Task 同士に依存関係を持たせる事が必要となる。しかし、Task に複雑な依存関係を持たせると、Pipeline 処理がストールしてしまう原因となる。

⋆2 2D, 3D オブジェクトの描画の為のプログラムインターフェース

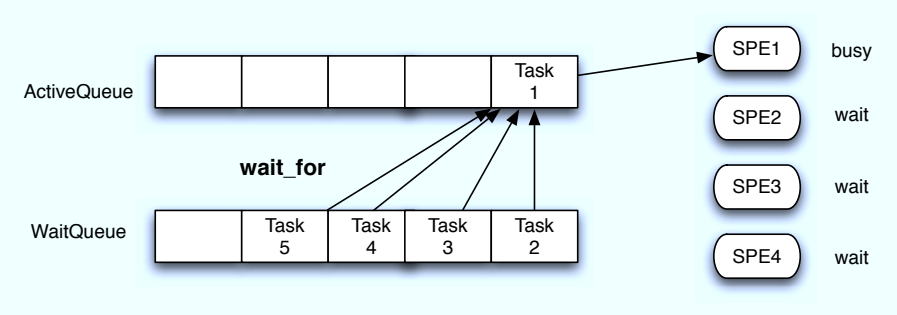

図 7 dependency によるストール

6. Pipeline のストール

Task は SPE で並列処理させることで全体的な処理速度が増す。しかし、PPE による処理が挟まれたり、Task の依存関係を解決するために時間がかかるとパフォーマンスが低下する。例えばある Task の処理を複数の Task が待つような dependency が設定されていたとき、依存している Task の処理に時間がかかると、動作していない SPE が発生してしまう。(図 7) これにより Task 処理の並列度が低下し、全体的なパフォーマンスの低下に繋がる。Cerium で実行される各 Task は Pipeline 状に処理することによって全ての SPE を常に busy 状態することにより、並列度を維持することが望ましい。そこで Pipeline 上のストールを検出し、並列度の低い箇所を発見するテスト機構を実装した。

6.1 Pipeline ストールの検出

SPE の稼働率が低い場合、Task の依存関係の問題や PPE 処理が挟まれている可能性がある。そこで SPE の稼働率が低下している時の Task Queue を調べる。この Task Queue には PPE で実行される Task を格納するキューと SPE で実行される Task を格納するキューがあり、それぞれ activeTaskQueue と waitTaskQueue を持つ。ここで大きなパフォーマンスの低下を引き落としていると思われる PPE 側の TaskQueue を検証した。また、キューの長さと格納されている Task ID を表示し、どの Task がどれくらい積まれているかを調べた。

6.2 ball_bound

ball_bound は 3D のボールが画面内で跳ねる初歩的なプログラムである。このボールの描画には Cerium の機能の一つである Rendering Engine が使われている。

```
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = StartTask ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = CreatePolygonFromSceneGraph ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 6
task_name = CreateSpan ,task_name = CreateSpan ,task_name = CreateSpan ,
task_name = CreateSpan ,task_name = CreateSpan ,task_name = CreateSpan ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = StartTask ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = CreatePolygonFromSceneGraph ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 6
...
4.073489 FPS
```

図 8 実行結果 1

図 8 を見ると 3 種類の Task が表示されているのが分かる。**StartTask** は全ての Task の一番初めに PPE で実行される Task である。**CreatePolygonFromSceneGraph** と **CreateSpan** は 3.3 節で述べたように Rendering Engine の Task であるが、PPE 側の TaskQueue に格納されてしまっている。

CreateSpan と CreatePolygonFromSceneGraph, DrawSpan は本来 Pileline で実行される Task であるが CreatePolygonFromSceneGraph と CreateSpan が Sequential に動作してしまっていると考えられる。

上記の実行結果を検証したところ、SPE に送られる予定の Task が PPE で実行される Task を待っている時、PPE 側の TaskQueue に格納されてしまうバグがあることがわかった。このバグを修正した結果が (図 9) である。

CreateSpan が正しく SPE に送られ、処理されている。(図**??**) しかし、実行速度は低下してしまっている。これは SPE に送られた Task が必要なデータを DMA 転送により取得しているため、その待ち時間と送信時間が結果的に PPE 上での実行処理より遅延を引き起こしていると考えられる。

```
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = StartTask ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = CreatePolygonFromSceneGraph ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = StartTask ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = CreatePolygonFromSceneGraph ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = StartTask ,
spu_idle! spe_running = 0 : activeTaskQueue->length = 1
task_name = CreatePolygonFromSceneGraph ,
...
3.059250 FPS
```

図 9 実行結果 2

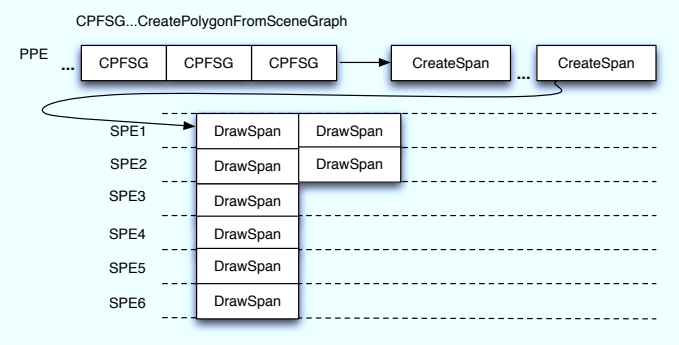

図 10 RenderingEngine の Pipeline

7. ま と め

TaskQueue に格納された Task と SPE の稼働率により、並列度の低い箇所を知ることができた。しかし、処理速度は改善されていない。同じ Task でも PPE 上での処理と SPE 上での処理にかかる時間が異なる。そこで Task の 両環境での平均的な処理時間を取得することで、CPU に適切な Task 処理の振り分けをすることができる。また、今回の方法だけでは局所的な情報だけであり Task がどのように展開されているのか、どの程度並列度が落ちているのかまではわからない。登録された Task には set_post や wait_for で設定された依存関係が保存されており、この情報のログを取り、依存関係の樹形図を可視化することができれば Cerim における並列プログラミングを強力にバックアップすることができる。

参 考 文 献

1) 宮國 渡: Cell 用の Fine-Grain Task Manager の実装. 琉球大学大学院 理工学研究科 情報工学専攻 平成 20 年度 学位論文 2008.
2) 河野 真治: PS3 上でのゲームプログラミング第 51 回プログラミング・シンポジウム 2010.
3) 多賀野 海人: Cell Task Manager Cerium の SPU 内データ管理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 2009.