

## C から Cell アーキテクチャを利用した CbC への変換

神里 晃<sup>†</sup> 宮國 渡<sup>†</sup> 杉山 千秋<sup>††</sup> 河野 真治<sup>††</sup>

<sup>†</sup> 琉球大学理工学研究科情報工学専攻 〒903-0213 沖縄県西原町千原1番地

<sup>††</sup> 琉球大学工学部情報工学専攻 〒903-0213 沖縄県西原町千原1番地

E-mail: <sup>†</sup>{akira,gongo,chiaki}@cr.ie.u-ryukyu.ac.jp, <sup>††</sup>kono@ie.u-ryukyu.ac.jp

あらまし 我々は状態遷移記述に向けた C の下位言語である Continuation based C(CbC) を提案している。今回 Cell アーキテクチャを利用し、C 言語から CbC を利用した Cell プログラムを生成する手法について考察する。本変換で、信頼性の高い並列計算を行うシーケンシャルなプログラムを提供することが可能となる。

キーワード Cell, マルチコア,

## Conversion to CbC which used the Cell architecture from C

Akira KAMIZATO<sup>†</sup>, Wataru MIYAGUNI<sup>†</sup>, Chiaki SUGIYAMA<sup>††</sup>, and Shinji KONO<sup>††</sup>

<sup>†</sup> Information Engineering, University Of Ryukyus Senbaru 1, Nishihara, Okinawa, 903-0213 Japan

<sup>††</sup> Information Engineering, University Of Ryukyus Senbaru 1, Nishihara, Okinawa, 903-0213 Japan

E-mail: <sup>†</sup>{akira,gongo,chiaki}@cr.ie.u-ryukyu.ac.jp, <sup>††</sup>kono@ie.u-ryukyu.ac.jp

**Abstract** We are proposing Continuation based C(CbC), which is a low level language of C. In this paper, the technique which converted CbC which used the Cell Architecture from C is considered. In this conversion, it can provide sequential program which is reliable parallel calculate.

**Key words** multicore, Cell

### 1. はじめに

C から継続を基本とする言語 CbC による Cell 上の並列計算への変換手法について考察する。Cell Broadband Engine は一つの制御系プロセッサ Power Processor Element と 7 つのデータ処理演算プロセッサ Synergistic Processor Element(SPE) から構成されている。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、メインメモリや他の SPE の LS とのデータは DMA を通して行われる。ここでは信頼性の高い並列計算を行うプログラムを提供するために CbC を用いる。例題として我々が独自に開発したソフトウェアレンダリングエンジン Cerium を用いる。

### 2. CbC の概要

CbC は C 言語からループ制御構造とサブルーチンコールを取り除き、継続を導入した言語である [1]。code-segment は引数付き goto で接続することで継続を実現する。

code-segment はキーワード code を用いることで関数のように定義される。引数部分は interface と呼ぶ。code-segment からの脱出は引数付き goto である。よって CbC のプログラムは複数の code-segment が goto で接続された物になる。(図 1)

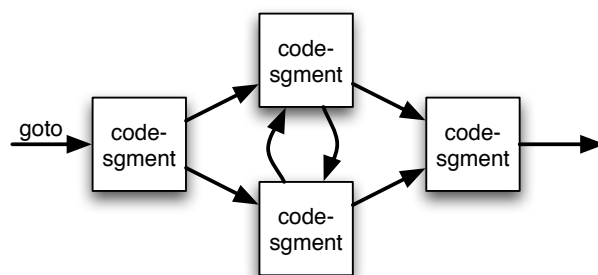


図 1 CbC プログラムの構成

### 3. Cell

Cell Broadband Engine はメインプロセッサである PowerPC Processor Element(PPE) と 6 個のデータ処理プロセッサアーキテクチャ Synergistic Processor Element(SPE) が使用できる非対称マルチコアプロセッサであり、EIB と呼ばれる高速リングバスで構成されている。(図 2)

PPE は複数の 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、オペレーティングシステムの役割で

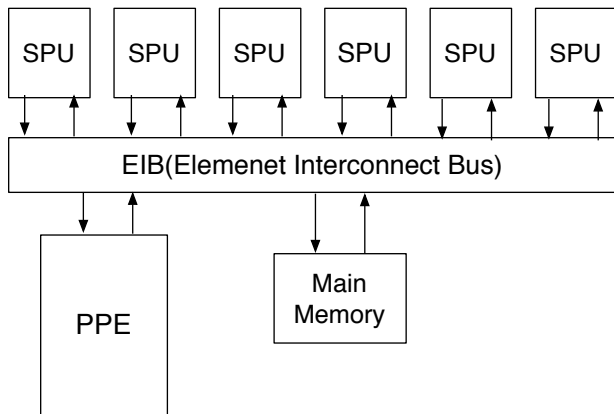


図 2 Cell の構成

あるメインメモリや外部デバイスへの入出力制御を行う。

SPE は PPE のような複雑な制御よりも計算を単純に繰り返すマルチメディア系の処理を得意とする演算系プロセッサコアである。(図 3)

図 3 SPE

SPE は SPU と MFC から構成され、独自規格の命令セットを持っている。各々の SPU は 256Kb のメモリを持ち、各 SPU から直接参照できる唯一のメモリとして存在する。また 128Kb のレジスタを 128 本持ち、SPE は各自が持っている LS 以外は参照することができない。メインメモリなどのデータにアクセスする場合は DMA を用いる。MFC はメインメモリや他の SPE などとデータをやりとりするためのユニットで、SPU はチャンネルというインターフェースを介して MFC に対してデータ転送などを依頼し、各々の SPU が持つ LS にメインメモリ上のデータなどを転送する。

#### 4. マルチコアシステム

一概にマルチコアアーキテクチャといっても様々なマルチコアアーキテクチャが存在する。簡単に分別するとホモジニアスマルチコア (図 4) とヘテロジニアスマルチコア (図 5) がある。ホモジニアスマルチコアはすべてのコアが同じコアで構成され、プログラマ側は CPU コアや命令セットの違いを意識する必要がないが、汎用的なコアですべての処理をこなすため、処理効率が悪いという特徴がある。それに対してヘテロジニアス

図 4 ホモジニアスマルチコア

図 5 ヘテロジニアスマルチコア

マルチコアは二種類の構造があり、単一命令セットで構成されたマルチコアと異種命令セットで構成されたマルチコアが存在する。単一命令セットで構成されたマルチコアは CPU コアをタスクにあわせて最適化することで、効率の高い CPU を作ることができる。しかし、異種命令セットのヘテロジニアスマルチコアはそれだけではなく、命令セットアーキテクチャレベルからタスクを最適化する必要がある。

必然的にシングルコアでは限られていたアルゴリズムがマルチコアの種類や並列化を考慮しアルゴリズムを考え直さなければいけない。

#### 5. レンダリングエンジン

ここでは例題として用いるレンダリングエンジン Cerium について説明する。Cerium はシーングラフ、レンダリングエンジン、タスクマネージャから構成される。(図 6)

図 6 Cerium の要素

SceneGraph は Blender3D モデリングツールから出力され

るポリゴン情報やテクスチャ情報などが記述された xml をパースし、XYZ の頂点座標を取得する。図 6 の SceneGraph の入力は XYZ の頂点座標となる。XYZ の頂点座標をキー入力にあわせて、拡大や縮小、移動などを行うのが Transform となる。XYZ の頂点をポリゴンにまとめるのが Gather となる。ポリゴンとは図 7 の三角形の各頂点の値のことである。

図 7 データ構造

レンダリングエンジンは SPAN を生成する部分と SPAN に RGB をマッピングし描画する部分とに分けることができる。SPAN とは図 7 のポリゴンに対するある特定の Y 座標に関するデータを抜き出した構造体である。SPAN を生成する部分は図 6 の create\_span の部分に相当する。Create\_SPAN ではポリゴンから SPAN を計算する部分 (CreateSPAN) とテクスチャを読み込む部分 (TextureImage) のみ行う。

SPAN に RGB をマッピングし描画する部分は図 6 の DRAW の部分に相当する。DRAW では Create\_SPAN で生成された SPAN を受け取り、ZBuffer をみながら描画するデータをメモリに書き込んでいく。ZBuffer とは画面サイズ分用意された Z のメモリ空間で、XY 座標に対する描画される Z の値が代入されている。SPAN の Z 座標と ZBuffer の Z を比較し、カメラからみてどちらが手前にあるかというのを判断するのが DRAW の Zcompare である。実際に ZBuffer と比較して描画する SPAN であるならば、XY 座標に対してのテクスチャの RGB 情報をメモリに書き込む。その役目が図 6 の Mapping RGB となる。RGB 情報をマッピングした後、実際に描画するのが WriteFB となる。

タスクマネージャはタスクを管理するライブラリで、タスクと呼ばれる分割された各プログラムを依存関係を考慮しながらメモリ上にマッピングし、SPU 上ではそのプログラムを DMA によりロードする。(図 8)

これは SPU の LS が 256Kb しかないため必要になる。タスクマネージャは次のような関数で実行することができる。

set_symbol	タスクの ID 登録
open	ID の取得
create_task	タスクを作る
spawn_task	実行タスク Queue に追加
set_depend	依存関係の考慮
set_cpu	タスクを行う CPU のセット
run	実行タスク Queue の実行

表 1 タスクマネージャの関数

タスクマネージャは登録されたタスクをみて、プログラムの

図 8 タスクマネージャ

ロードを行い、入力データの読み込み、計算、出力データの書き出しを行う。また create\_task のときに入力データのサイズやアドレスなどが登録される。またタスクマネージャは PPU で実行するか SPU で実行するかを明示的に書くことができる。また SPU を使う場合は SPU コアを使うことができる。

## 6. 開発過程

開発過程として次のような順で実装する。

- (1) C によるシーケンシャルな開発
- (2) SPU を考慮したデータ構造を持つシーケンシャルな開発
- (3) SPU を使った開発
- (4) CbC をもちいた開発

1 の C によるシーケンシャルな開発はタスクマネージャを使わず実際にプログラムのアルゴリズムの信頼性をみるために行われる。C によるシーケンシャルな開発ではタスクマネージャは使われない。

2 の SPU を考慮したデータ構造を持つシーケンシャルな開発はタスクマネージャを用いるが、このタスクマネージャは SPU の実行部分をシミュレーションしたタスクマネージャを使って、実装することができる。しかし、依存関係や SPU に送るデータのサイズなどを考慮する必要があり、またタスクの中ではポインターを使うことができないなど多少の煩わしさがある。

2 から 3 へ移行するのはタスクマネージャの set\_cpu を用いることによって簡単に移行することが可能である。

4 の CbC を用いた開発では改めて今まで書いてきたプログラムを CbC に書き直す作業が待っている。しかし、CbC への変換は今まで書いていた C のプログラムを逐次的に goto で code-segment を接続すればよい。

```
__code SceneGraph((void *)rbuf,(void *)wbuf)
{
    .....
    goto Scheduler((void *)wbuf,PPU_Memory1);
}
```

```
__code PPU_Memory1((void *)polygon) {
```

```
.....
```

```

    goto Scheduler((void *)wbuf,Create_SPAN);
}

__code Create_SPAN((void*)rbuf,(void*)wbuf)
{
    .....
    goto Scheduler((void *)wbuf,PPU_Memory2);
}

__code PPU_Memory2((void *)span) {
    .....
    goto Scheduler((void *)wbuf,DRAW);
}

__code DRAW((void*)rbuf,(void*)wbuf)
{
    .....
    goto Scheduler((void*)0,SceneGraph);
}

__code Scheduler((void*)rbuf,__code *next) {
if(...)
    goto *next(rbuf);
if(...)
    goto *next(rubf,wbuf);
}

```

## 7. 並列処理

Cellではあらゆるレベルで並列に動作させることが求められる。ダブルバッファがその一例として挙げられる。Cellではそれぞれのコアがメインメモリを直接参照することができない。そのためDMAによりデータをやり取りするのは前述した通りである。DMAはCPUを介さずに直接データ転送を行う方式のことである。そのためDMAしている間、SPUは何らかの処理を行うことができる。SPUは入力用のBufferと出力用のバッファを二つずつ用意する。そうすることにより図9のようなパイプライン処理が可能となる。またタスクマネージャは

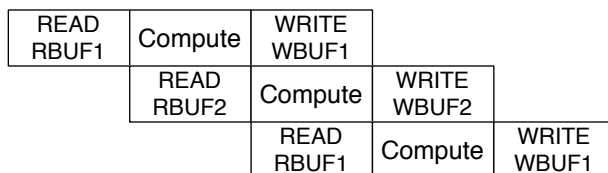


図9 パイプライン

PPUで実行するかSPUで実行するかを明示的に書くことができる。またSPUを使う場合は使うSPUの数を指定することができるようになる。そのため、ダブルバッファを利用した図10のようなことができる可能性もある。

図10 タスクマネージャが行うパイプライン

## 8. SPURS との比較

我々が作成したタスクマネージャに似た研究としてSPURS [3]が挙げられる。SPURSは我々が今回作成したCeriumのようなSPUに入力データを与えるプログラムに関してはほとんど同じ機能を持っている。しかし、タスクがSPURSの場合は関数に対し、Ceriumではcode-segmentになる。

## 9. まとめ

ここでは継続を基本とした言語CbCを使ってCellのようなマルチコアでの記述法について述べた。CbCは状態遷移を用いた記述になるので依存関係がはっきりしており、code-segment単位をタスクと考えることができる。code-segmentをスケジューラをもちいることにより並列的に動作させることが可能となる。

また、これらはシーケンシャルなアルゴリズムから並列計算に移行することが他の言語と比較して容易にできる。そのため、シーケンシャルな環境でのデバッグがそのまま並列分散のデバッグにもなる。

### 文 献

- [1] 河野真治. “継続を持つCの下位言語によるシステム記述”. 日本ソフトウェア科学会第17回大会, 2000.
- [2] 河野真治. “継続を基本とするプログラム単位を用いた組み込みシステム開発”. 組み込みソフトウェア工学シンポジウム, 2003
- [3] 井上 敬介 “「Cellプロセッサ向け実行環境 (SPU Centric Execution Model)」”. 先進的計算基盤システムシンポジウム SACSIS, 2006