社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

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 ba sed C と、そのCell への応用

河野　真治†

† 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
E-mail: †kono@ie.u-ryukyu.ac.jp

あらまし　近年、CPU の性能向上は、クロックサイクルをあげることよりも、複数の CPU コア (Many Core Architecture) を導入することにより得られるようになって来ている。しかし、Many Core Architecture のプログラムは複雑であり、その信頼性を確保することは難しい。本論文では、本研究室で開発した C のサブセット である Contination based C を用いて、Many Core Architecture のプログラミングと、その検証を行う手法について考察する。ここでは例題として、Cell Broadband Engine を用いたレンダリングエンジンを用いる。
キーワード　Cell，マルチコア，　継続

Self descriptive verfication in Continuation based C and it's appli cation to Cell architecture

Shinji KONO†

† Information Engineering, University Of Ryukyus　Senbaru 1, Nishihara , Okinawa, 903-0213 Japan
E-mail: †kono@ie.u-ryukyu.ac.jp

Abstract　CPU performance is achieved in Many Core Architecture rather than high clock speed recently. The complicated nature of this architecture makes reliable program difficult. We present a subset of C Programming Language with continuation: Continuation based C to make a program for Many Core Architecture. We use this method for a rendering engine "Cerium" for Cell Broadband Engine.
Key words　multicore , Cell

1. Multi core system

複数の CPU を載せたコンピュータは昔から使われて来たが、最近の傾向は、一つの Chip に複数の CPU コアを載せたものの登場である。従来のマルチプロセッサは同期をサポートしたキャッシュを経由しメインメモリにアクセスすることが多いが、最近開発された Multi Core では、CPU 間の通信に特別なポートを用意している。例えば、Intel は Quick Path と言う通信ポートががある。これにより、メインメモリへのアクセスによる競合を避けることができる。しかし、その分、複雑なプログラミングが必要となる。

Cell Broadband Engine [2] は、SCEI と IBM によって開発された PS3 ゲーム機用の CPU であり、2 thread の PPU(PowerPC Unit) と、8 個の SPU (Synergetic Processing Unit) を持つ (図 1.)。本研究で用いた PS3Linux (FedoreCore 6) では、6 個の SPU を使うことが出来る。SPU はそれぞれ 256kb のローカルメモリを持ち、バスに負担をかけることなく並列に計算を進めることが出来る。SPU からメインメモリへは、SPU の機械語から直接アクセスすることは出来ず、Cell の MFC(Memory Flow Controller) へ DMA (Direct Memory Access) 命令を送ることで行われる。SPU はグラフィックスに適した、4 つの固定小数点、浮動小数点を同時に演算する命令などを持ち、PPU に比べて高速な演算が可能であり、ほとんどの演算を SPU 上で進めることが推奨されている。

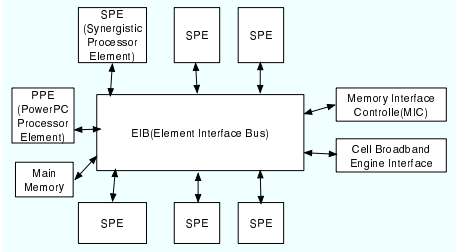


図 1　cellarch

2. Many Core 上のプログラムの特徴

従来の逐次型のプログラムでは、Many Core の性能を十分に引き出すことは出来ない。ここでは、その性能を活かす Many Core プログラムの特徴について考察する。ここでは、定常的な並列度の必要性、データ並列、パイプライン実行、プログラムとデータの分割、同期の問題、マルチスレッド、デバッグに関して考察を行う。

2.1 定常的な並列度の必要性

並列実行には Amdahl 則 [3] があり、プログラムの並列化率が低ければ、その性能を活かすことは出来ない。0.8 程度の並列化率では、6CPU でも 3 倍程度の性能向上しか得られない (2.1)。

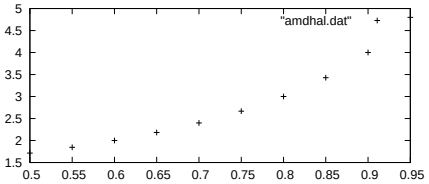


図 2 amdahl

高い並列度ではなくとも、恒常的に並列度を維持する必要がある。このため、

逐次型のプログラムの一部を並列化する

という手法では不十分である。LSI などのハードウェア場合は演算の対象がもともと多量の演算とデータパスを持つので並列計算の効果を定常的に得られることが多いが、C 等で記述されたプログラムでは、for 文や配列のアクセスなどに並列性が隠されてしまい、それを引き出すことが難しい。

プログラムの中の並列度は、主に二つの形で取り出すことが出来る。一つは、配列や木の中の個々の要素に対して並列に実行する

データ並列

である。もう一つは、複数の逐次処理の隣同士を重ねて実行する

パイプライン処理

である。この二つを同時に用いることで定常的な並列度を維持することが可能となることがある。パイプライン処理は、プログラム中で階層的に使われることが多い。

2.2 プログラムとデータの分割

データ並列とパイプライン処理を可能にするためには、プログラムとデータの適切な分割を行う必要がある。for 文、あるいは、木をだとって処理する個々のステートメントがプログラムの分割の対象となる。データは自明に分割できるわけではなく、分割できるデータ構造を採用し、必要ならばコピーを行う。

分割されたデータは、メモリ上に置かれるが、Cell の場合は SPU のローカルメモリ上に置かれることになる。共有メモリベースの場合でもキャッシュを考慮した配置をする必要がある。具体的には、データのアライメントをそろえる必要がある。メインメモリ上で計算を行う逐次型プログラムと異なり、コピーのコストを払ってでもデータを分割し、複数の CPU で独立に処理する必要がある。特に、DMA 中心のアクセスになる Cell の場合は、コピーしやすいように、数 Kbyte 毎の配列にする方が良い。

Cell の場合はさらにコードもローカルメモリ上に置かれるために、コードをロードする仕組みも必要になる。

2.3 同期の問題

ここで言う同期は、複数の CPU がデータの待ち合わせ、または、整合性を維持するために、他の CPU との待ち合わせを行うことである。従来のマルチ CPU では、並列度が低いために同期は希であり、キャッシュ上の Spin lock を用いることが多かった。これは、メモリの特定の場所を test and set 等の特殊な命令で繰り返しアクセスして待ち合わせる手法である。Unix OS の kernel、POSIX Thread など同期機構の実装に使われている。

Many Core では、待ち合わせを行うと並列度が下がってしまうので、同期自体を減らす必要がある。そのためには、各 CPU が独立に (Lock 無しに) データにアクセス出来るようにデータを分割すれば良い。Cell では、SPU がローカルメモリを持っているので、そちらにコピーすることになる。しかし、SPU はメインメモリからデータを取得する必要があるので、その取得の際には同期を取る必要がある。Cell では、SPU と PPU 間の同期には Mail box という FIFO メッセージキューが用意されていて、readch,writech という命令で SPU からアクセスする。Spin lock と違って、メッセージ交換なので待ち合わせを避けることが可能である。

共有メモリベースのシステムの場合でも、同じ場所をアクセスする場合はキャッシュの競合が生じるので、コピーなどを用いて領域の分割を行う必要がある。Thread local などを用いる場合もある。

複数の CPU が出力する結果を一つのキューに挿入するようなことをすると、挿入時に必ず同期が必要になるので同期のコストが高い。

2.4 マルチスレッド

従来の共有メモリ型のマルチ CPU では、POSIX Thread を用いて並列実行を実現することが多い。特に Hyper Thread では、複数の命令ストリームを使って、メモリアクセス時等のの命令実行パイプラインのストールをスレッドを切替えて隠すことが出来る。しかし、Many Core の場合に、個々の Core に複数の Thread を割り当てるとワーキングセット (Thread が使用するデータ) の大きさが大きくなりキャッシュやローカルメモリに入らなくなる場合がある。

スレッド (Hyper Thread) は本来、I/O 待ちやメインメモリアクセス等に対して有効であり、ほとんどのデータがキャッシュやローカルメモリにあると考えられる Many Core には向いていない。個々のタスクを実行する CPU 上で複数のスレッドを使用するメリットはほとんどないと思われる。一方で、個々の

CPU は細分化されたタスクを十分に持っていなければ恒常的な並列度を維持できない。

一方で、同期機構で待ち合わせを行う場合に、Spin lock を避けるとすれば、条件付変数などのスレッドのタスクの待ち合わせ機構が必要となる。

Many Core の台数にも寄るが、実行するタスクの管理を行うマネージャーを複数スレッドで構成し、そのうちの一つが、ポーリングベースで複数の Core に対する待ち合わせを行うようにするのが良いと思われる。Cell では、SPURS [7] という仕組みが提案されている。

2.5 デ バッグ

複数の Core を走らせた状態でのデバッグは難しい。並列プログラムの特徴として、実行が非決定的 (同じ状態で実行しても結果が異なる) ことがあり、バグの状態を再現することが難しいことがある。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。

3. Many Core 上のプログラムの作り方

本論文では、本研究室で作成した PS3 上のレンダリングエンジンである Cerium Rendering Engine を例題として使う。詳細は別な論文 [12] に譲り、ここでは簡単に記述する。

Cerium は、Scene Graph (3D オブジェクトを木構造にしたもの) をフレームバッファ上に SPU を用いて描画する Rendering Engine であり、教育用としてシンプルな構成を持っている。Cerium は、

1. Scene Graph の Polygon の座標から表示する
2. 座標の計算を行い PolygonPack を生成する
3. PolygonPack から、同じ Y 座標を持つ線分の集合 SpanPack を生成する
4. SpanPack を (Texture を読込ながら)Z buffer を用いて描画する。

という 4 つのタスクを持つ。並列実行は、Scene Graph の木、PolygonPack, SpanPack に対してデータ並列実行を行う。さらに、この 4 つが表示画面毎にパイプライン的に実行される。

これらのタスクは、SPU で実行するために小さなセグメントに分割される必要がある。分割されたタスクを、PPU または SPU で実行するのは Cerium task manager である。Task manager はタスク依存を解決する機能を持っていて、タスク依存が満たされたものをアクティブキューに入れ、SPU を起動する。SPU はアクティブキューから、処理するコードとデータを取得し自律的に実行を行う。

Cerium Rendering engine を作るには、以下の手順に実装とテストを行う。

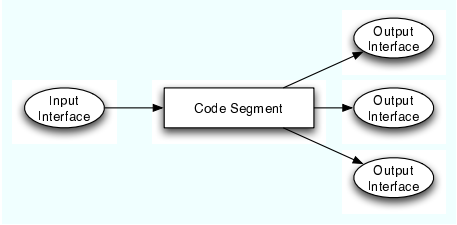
1. 普通の C で実装した Rendering algorithm
2. データ構造 (PolygonPack, SpanPack) を導入したもの
3. コードをタスクに分割し、FIFO キューでシーケンシャルに実行する
4. タスクを SPU に割り当て並列実行する

これにより、Algorithm の正しさを確認した上で並列実行に移行することが出来る。

4. Continuation based C

CbC (Continuation based C) は、C からサブルーチンやループ構造を除いた C の下位言語であり、ハードウェアとソフトウェア、特に組込みシステムの記述言語として本研究室で提案している言語である。

C の関数の代わりに、たコード・セグメントという単位を持つ。コードセグメントには、入口に相当する Input interface と、出口に相当する Parameterized goto 文が存在する。



以下は簡単な CbC のプログラムである。

```
code fact(int n,int result,
          code print()){
    if(n>0){
        result *= n;
        n--;
        goto fact(n,result,print);
    } else
        goto (*print)(result);
}
```

間接 goto と、直接 goto が、プログラムの単位の結び付きをボトムアップに規定して、自然なグループを構成する。

CbC の継続は、Scheme などの継続とは異なり環境 (関数の呼び出し履歴) を持たない。コードセグメントは、関数呼び出しではないので環境を持つ必要がない。

C にコードセグメントと引数つき goto 文を付加した CwC は、C のスーパセットであり、コードセグメント内から通常の C の関数を呼び出すことも可能ある。また、通常の関数からコードセグメントへ goto したり、コードセグメントから、呼び出された関数へ値を戻すことも出来る。

コードセグメントは状態遷移記述と対応しているので、ハードウェア記述とも相性が良い。

CbC コンパイラは、micro-C base のものと GCC base, TCC base のものが動いており、実用的に使うことができる。sourceforge.jp 上 [10] で公開されている。

4.1 C から CbC への変換

C のスタックを明示的に構成することにより C のプログラムを CbC に変換することが可能である。

```
j = g(i+3);
```

のような C の関数呼び出しは、 struct f_g0_save などの明示的なスタックの中身を表す構造体を用いて、

```
struct f_g0_interface *c =
  (struct f_g0_save *)(sp -=
  sizeof(struct f_g0_save));
c = sp;
c->ret = f_g1;
goto g(i+3,sp);
```

のような形で、明示的なスタック操作に変換される。これは変換の一例であり、他の方法、例えばリンクトリストなどを用いても良い。f_g1 は、関数呼び出しの後の継続であり、g では、

```
code g(int i,stack sp) {
    goto (* ((struct
     f_g0_save *)sp)->ret)
      (i+4,sp);
}
```

のように間接的に呼び出される。スタックの中は、継続と中間変数などを格納する構造体である。スタックそのものは、これらの構造体を格納するメモリである。

これらの変換は自動的に行うことが出来き、試作変換系を実装した報告 [11] もあるが、実用レベルの変換系はまだ存在しない。現在は変換と、その後の最適化はは手動で行う必要がある。

4.2 CbC での並列実行記述

並列実行記述ではタスクの生成と同期機構の記述が問題になる。Verilog や VHDL などでも並列タスクの記述があり、POSIX Thread ではライブラリコールとして pthread_create や pthread_mutex_lock などがある。

これらの動作記述は、マニュアルや Formal Description で示されている。例えば、以下のような記述である。The pthread_mutex_lock() function locks mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.

このようは記述は曖昧で誤解を招きやすい。しかし、同期機構の検証では、これらの動作の正確な意味を知る必要がある。

CbC では ad-hoc な並列記述プリミティブは必要ではなく、自分自身で並列実行を記述することが可能である。これは、コードセグメントには隠された情報が存在しないので、実行に必要な情報をすべて Input interface から取得できるからである。

実行キューを作成し CbC 自体で Scheduler を記述することにより、並列実行を記述する。この場合の並列実行単位はコードセグメントとなる。goto 文を規則的に scheduler への goto 文に書き換えることにより、並列実行を記述することが出来る。

以下は、CbC で記述した Dining Philosopher の状態の一部である。

```
code pickup_rfork(PhilsPtr self)
{
    if (self->right_fork->owner == NULL) {
            self->right_fork->owner = self;
            goto eating(self);
    } else {
            goto hungry2(self);
    }
}
```

これを並列実行するためには、

```
code pickup_rfork(PhilsPtr self, TaskPtr current_tas
{
    if (self->right_fork->owner == NULL) {
            self->right_fork->owner = self;
            self->next = eating;
            goto scheduler(self, current_task);
    } else {
            self->next = hungry2;
            goto scheduler(self, current_task);
    }
}
```

という形に記述を変える。Scheduler の実装は例えば、FIFO であれば、

```
code scheduler(PhilsPtr self, TaskPtr list)
{
    TaskPtr t = list;
    TaskPtr e;
    list = list->next;
    goto list->phils->next(list->phils,list);
}
```

という形になる。このように自分自身で並列実行スケジューラを記述できることが CbC の特徴である。同期機構であるが、ここでは right_fork の排他制御は、コードセグメントが同時に実行されることはないので、自動的に保証されている。条件付変数のような待ち合わせを実現したい場合は、TaskPtr への操作として実現すれば良い。

4.3 CbC での並列実行の実装

FIFO scheduler を例えば Cell の SPU の active task queue への挿入とし、active task queue を各 SPU が自律的に取得するようにする (SPU scheduler) と、実際に CbC のコードセグメントを並列実行することが出来る。

FIFO scheduler と実際の並列実行の同期機構が一致するようにするのは、一般的には自明ではない。同期機構をコードセグメントで記述して、SPU scheduler によって実現しても良いし、コードセグメント内部で、例えば、pthread_mutex_lock を呼び出しても良い。

5. CbC での Cell のプログラムの流れ

CbC を用いて、Many Core Architecture のプログラムを作成する流れは以下のようになる。

1. C によりアルゴリズムをシーケンシャルに記述する
2. データを並列用に分割した構造に変更する
3. C の記述を CbC の記述に変換する (必要な部分のみ。手動)

4．コードセグメントを並列実行用に分割する
5．FIFO スケジューラにより動作を確認する
6．SPU スケジューラにより Cell 上での動作を確認する

これらの各過程すべてでエラーが入る可能性がある。また、論理的なエラーだけなく、仕様に沿った十分な性能が出るかどうかも検証する必要がある。

1 の段階は通常の C のプログラミングであり、この部分がちゃんと動くことが必須である。この段階では、入力に対して出力が一意に決まる状況であり、テストは容易である。バグも実行トレースの二分法により容易に行うことが出来る。

4 段階まではプログラム変換の問題であり、一つ前の段階と同じ結果を得られるかどうか検証する必要がある。

5 段階以前はアーキテクチャに依存しないので、ターゲットが開発途中の段階でも記述することが可能である。しかし、5 段階では、FIFO スケジューラの替わりに、Random スケジューラなどを使うことができ、並列実行特有の非決定的な実行が導入される。

非決定的な実行は、クロックベースのハードウェアでは入力の任意性から生じることが多い。ハードウェアでも複数のタスクを使用したり、外界と相互作用する場合は非決定的な実行が現れる。

段階 5 では、これらの非決定的な実行でも 4 段階までと同じ仕様を満たすことを検証する必要がある。これは、逐次型のプログラムでは出て来ない問題である。

段階 6 では、段階 5 まできちんと動いていれば、問題なく動作すると期待される。しかし、FIFO スケジューラと SPU スケジューラでは、同期機構の実現が異なることがある。これは、並列実行と同期機構のの粒度と意味論が異なるために起きると考えられる。

ここで、段階 1 が仕様であり段階 5 が実装であると考えることもできる。実際のプログラムとは別に、実行時に満たして欲しい仕様の記述がある場合もある。これらの記述は、例えば、「計算がいつか終る」等の時相論理的な記述になる。時相論理としては、LTTL [9], CTL* [1], ITL [8] などを使うことができる。

6. CbC での Cell のプログラムの検証

CbC での検証は、プログラム作成の各段階で行われる。CbC では実行要素はコードセグメントであり、その Input interface の値により動作は一意に決まる。従って、検証はコードセグメント単位良い。コードセグメント内部の正当性はテストあるいは証明こ行われるべきである。

6.1 コードセグメントの入出力テスト

これは、通常のプログラムのテスト手法と同じであり、FIFO スケジューラを導入する前の段階では、入力と出力に対応するテストを行う。

コードセグメント毎にテストデータを作成するべきであるが、結果の正しさを確認するプログラムを書く必要とする場合もある。

データ及びコードの分割が終った段階では、データを Multi Core CPU がアクセスできるようにコピーするコードが入ることがある。このコピーは、FIFO スケジューラレベルでは、ポインタ渡しに避けても良い。しかし、コピーコード自体にエラーが出る場合も多い。例えば、Cell では、MFC による DMA は 64bit alignment が必要であり、これが満たされないと例外が発生してしまう。

6.2 FIFO スケジューラレベルでのテスト

FIFO スケジューラレベルのテストでは、非決定性が導入される。Cell では組み込まれた SPU は、すべて決定的に動作するが、データによって SPU の演算の終了結果は異なり、結果的に非決定性が生じる。

これらの非決定性を、網羅的に調べることも可能であり、モデル検査と呼ばれる。

CbC では、状態を記録しながら、すべての可能な非決定的実行を行うスケジューラを実装することが可能である。

メモリ状態をデータベースから調べる
すでにあれば、一つ前の状態に戻して他の実行を探す
なければ、一段深い実行に進み状態を探す

という形である。実際の実装は以下のようになっている。

```
code tableau(TaskPtr list)
{
    if (lookup_StateDB(&st, &state_db, &out)) {
        // found in the state database
        printf("found %d\n",count);
        while(!(list = next_task_iterator(task_iter)
            // no more branch, go back to the previou
            TaskIteratorPtr prev_iter = task_iter->pr
            if (!prev_iter) {
                printf("All done count %d\n",count);
                ....
            }
            printf("no more branch %d\n",count);
            depth--;
            free_task_iterator(task_iter);
            task_iter = prev_iter;
        }
        // return to previous state
        // here we assume task list is fixed, we don
        // recover task list itself
        restore_memory(task_iter->state->memory);
        printf("restore list %x next %x\n",(int)list
    } else {
        // one step further
        depth++;
        task_iter = create_task_iterator(list,out,tas
    }
    goto list->phils->next(list->phils,list);
}
```

この検証系は、SPU を使った実機上で動かすには、SPU 内部のメモリを記録するなどの工夫が必要となる。また、探索空間が膨大になる場合もある。

探索空間を小さくするには、並列実行の粒度を大きくしたり、メモリの状態を抽象化したりする方法が考えられる。これらの手法は、CbC 自身で記述することが可能である。

7. 本手法の利点と欠点

CbC という特殊な処理系を使うことになるので、ソースの変更が必要となる。現状では、C++には対応していない。

コードセグメントのテストを Single core 上と Multi core 上の両方でテスト出来るので、Multi core になれていなくても、動作のテストが容易である。また、Multi core になれるための準備、教育的ツールとして使うことも出来る。アルゴリズムの正しさを並列実行とは別にテスト出来るのが利点である。

Cell はヘテロな Multi Core であり、SPU では、その性能を活かすためには、特殊なアセンブラ命令、例えば 4 つの浮動小数点の値の同時演算などを使用する必要がある。これらは、gcc の拡張あるいは asm ステートメントなどで使用することができるが、他のアーキテクチャ上では動作しない。従って、同じ機能を持つコードセグメントで代用してテストすることになる。異なるアーキテクチャでの異なるコードセグメントの同等性を直接テストすることは出来ない。

Many Core 向けのデータ分割、コード分割は自動ではないので、試行錯誤することになる。必要な性能が出るかどうかは、分割のために生じるコピーのコストなどの要素が関係し、アーキテクチャに依存するので Single Core 上のテストで見積もることは難しい。

データ分割、コード分割が手動なので間違いが入りやすい。FIFO スケジューラレベルで、Single Core と同等な動作をする場合は、このようなエラーを見つけるのは容易である。

しかし、非決定的な動作をする場合は、自明な動作比較をすることは出来ず、モデル検査などのコストの高い方法を使う必要が出て来る。コードセグメントレベルのモデル検査は実機上の同期動作との差があるので、正確ではない。

モデル検査のコストが重い場合は、スケジューラを挟む部分を大まかにして状態数を減らす手法が使える。正確さは落ちるが、高速に検査することが出来る。

スケジューラを挟む部分の変更は手動であり、マクロあるいはスクリプトで生成する必要がある。

8. 学生による実際の実装の現状

本手法を、PS3 Linux 上の SPU を用いた 3D Rendering Engine の作成に適用した。詳細は別な論文で述べるので、ここでは学生の反応について報告する。

Single Core でプログラムを動作させることは容易である。これは、従来のプログラムと差がないからであり、学生にとっては入りやすい。データとコードの分割を自力で求めるのは、並列処理の経験がない学生には難しい。なんらかのひな型か指示が必要である。

FIFO スケジューラから、SPU スケジューラを実装する時点で、なるべく早く動かそうとするために ad-hoc なコードが入りやすい。特に、一旦、SPU スケジューラに移ってしまうと、元のソースを FIFO スケジューラ/Single Core 上で動かすことを考慮せずに変更を加えてしまう。そのような場合は、この方法のメリットである、並列実装部分のバグと、アルゴリズム/データ分割のバグを分離することが出来なくなってしまう。しかし、常に両方で動作させるように変更するのは、手間がかかることも確かである。

データやコードの分割において、Cell では 6 個の SPU を使うことが出来るが、必ず 6 個に分割することを仮定してプログラムを書いてしまうという問題も見られた。分割したコードにどれだけの Core を割り振るかは、後で調整する必要があるので、必ず最大値を使うとは限らない。

9. CbC を用いた検証手法と他の手法との比較

コードセグメントを使わずに、C の関数による分割を行っても良い。この場合は、分割された関数を順々に呼び出すマネージャを記述する必要がある。CbC のようにモデル検査を自分で記述するようなことは出来ないが、分割に関しては同等である。

モデル検査では SPIN [6] が有名だが、SPIN では問題を Promela という仕様記述言語で記述する必要があり、Promela を直接実装に使うことは出来ない。

実装言語を直接モデル検査するシステムとしては、Java PathFinder [5]、CBMC [4] などが知られている。これらはオブジェクト指向記述に対しても適用できるという利点がある。これらで並列処理を記述する場合は、一般的には Thread の機能を用いることになる。

CbC では、スケジューラ及びモデル検査器を自分で記述出来るので、検査する並列性や抽象度を自分で制御できると言う利点がある。現状では、そういう可能性があるというだけで実装は行っていない。

現在用いている CbC 用のモデル検査器は比較的単純に作られており、大きなプログラムの検証をするのには向かない。2.8GHz Pentinum 4 では、Dining Philosphers の例題に対して以下のような性能が出ている。

表 1 Dining Philosophers Problem の CbC によるモデル検査

プロセス数	状態数	実行時間 (秒)
3	1,340	0.01
4	6,115	0.08
5	38,984	0.66
6	159,299	3.79
7	845,529	27.59
8	3915,727	199.80

Java Path Finder と SPIN の丁度間の性能となっているが、プロセス数が多くなると、Java Path Finder よりも性能が落ちる。これは、モデル検査のための状態格納のアルゴリズムに問題があると考えている。SPIN の性能が良いのは、元々の状態記述の抽象度が高く状態数そのものが小さいからであると考

えられる。

CbC の記述を変更して、スケジューラを呼び出す頻度を下げると、状態するが下がり SPIN に近い性能を得ることも可能である。Java Path Finder では、そのような変更を行うことは原理的に出来ない。

10. ま と め

May Core Architecture のプログラムを作成する手法として、シーケンシャルな C の記述を、CbC に分解する手法を提案し実践してみた。また、その手法に対するテスト及び検証法について述べた。

CbC では、モデル検査部と実装部分を混在し、同時にプログラムすることが出来るので、並列実行部分を作成しながら、それをテストするためのモデル検査部を作成していくことが可能である。

現在のモデル検査部で、SPIN と Java Path Finder の中間的な性能を得ることが出来ているが、より高性能なアルゴリズムを実装する必要がある。

Cerium Rendering Engine は Cell 上で動作しているが、テスト及びモデル検査は行っていないので、これらを実装して、実際の効果を調べることが必要である。

表 2　SPIN による Dining Philosophers Problem のモデル検査

プロセス数	状態数	実行時間 (秒)
5	94	0.008
6	212	0.01
7	494	0.03
8	1,172	0.04

表 3　JPF による Dining Philosophers Problem のモデル検査

プロセス数	状態数	実行時間 (秒)
5	不明	3.98
6	不明	7.33
7	不明	26.29
8	不明	123.16

文　献

[1] E.M. Clarke and E.A. Emerson. “synthesis of synchronizatoin skeletons from branching time temporal logic”. In *Proc. of the Workshop on Logics of Programs, LNCS-131, Springer-Verlag*, 1982.

[2] Sony Corporation. Cell broadband engine architecture, 2005.

[3] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[4] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain, 2004.

[5] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[6] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.

[7] Keisuke Inoue. Spu centric execution model, 2006.

[8] Shinji Kono. A Combination of Clasual and Non Clausal Temporal Logic Program. *IJCAI-93 Workshop on Executable Modal and Temporal Logics*, Aug, 1993.

[9] P. Wolper. Synthesis of communicating processes from temporal logic specifications. Technical Report STAN-CS-82-925, Stanford University, 1982.

[10] 河野　真治 . CbC, March 2008.

[11] 河野真治 (琉球大/科学技術振興事業団), 揚 挺 (琉球大). C 言語の Continuation based C への変換. In *SwoPP 2001*, July 2001.

[12] 神里 晃, 河野　真治 . C から Cell Architecture を利用した CbC への変換. VLD 研究会, March 2008.