Gears OS における並列処理

東恩納琢偉[†1]　伊波　立樹[†2]　河野真治[†1]

現代の OS では拡張性と信頼性を両立させることが要求されている。信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に Gears OS を設計中である。Gears OS は継続を中心とした言語で記述されており、メタ計算をノーマルレベルと分けて記述することができる。並列処理はメタ計算によって記述されており、CbC 自体には並列処理の機能はない。メタレベルでの並列処理は新規に実行環境を作り引数を設定するなどの煩雑な処理であり、ノーマルレベルでは簡潔な並列構文があることが望ましい。本論文では並列構文と並列処理の実装について記述する。

Gears OS のプログラムは Code Gear と Data Gear の集まりである interface によって行われる。Gears OS でのスレッドは interface の集合で出来ており、code gear data gear を接続する context という meta data gear を持つ。並行実行する場合は新しく context を生成し、それを時分割または、物理的な CPU に割り当てることによって実現される。 つまり、context そのものがスレッドとなる。

Gears OS での同期機構は data gear を待ち合わせることによって行われる。例えば、GPU 上で実行する場合は必要な data gear を GPU 内部に転送し、それらが揃った時点で並列実行される。data gear の待ち合わせはメモリ上の data gear の meta data gear に待ち合わせ用のキューを作ることによって行われる。キューには Gears OS のスレッドつまり context meta data gear が入る。

並列処理をメタレベルで行うことにより、並列処理で重要なチューニングや性能測定あるいはデバッグをメタ計算を切り替えることにより、ノーマルレベルの計算を変更することなく行うことができることを示す。

Takui HIGASHIONNA ,[†1] Tatsuki IHA [†2] and Shinji KONO [†1]

1. Gears OS

OS に要求される信頼性はテストやモデル検査あるいは証明によって保証される。一方で新しいアプリケーションや新しいアーキテクチャへの対応を OS が提供する必要がある。様々な拡張のたびに信頼性の保証をゼロからやり直すのは現実的ではない。本研究室で開発している Gears OS は信頼性の保証を比較的固定されたノーマルレベルの計算に対しておこない、拡張性をメタレベルの計算で対応する。メタ計算の一つは並列処理であり、OS の機能の重要な部分である。本論文では Gears OS の並列構文と並列処理の実装について述べる。

Gears OS は本研究室で開発している CbC(Continuation based C) を用いて行われている。CbC は処理を Code Segment を用いて分割して記述することを基本としている。Gears OS では Code Gear が CbC の Code Segement にあたる。Gears OS のプログラムは C の関数の単位で Code Gear を用いて分割し、処理を記述している。Code Gear から Code Gear への移動は goto の後に移動先の Code Gear 名と引数を並べた記述する構文を用いて行う。この goto による処理の遷移を継続と呼び、C での関数呼び出しにあたり、C では関数の引数の値がスタックに積まれていくが、Code Gear の goto では戻り値を持たないため、スタックに値を積んでいく必要がなくスタックを変更する必要がない。このようなスタックに積まない継続を軽量継続と呼び、呼び出し元の環境を持たない。

Code Gear は処理の基本として、Input Data Gear を参照し、一つまたは複数の Output Data Gear に書き込む。また、接続された Data Gear 以外には参照を行わない。(図 1) Input Data Gear と Output Data Gear の 2 つによって、Code Gear の Data に対する依存関係が解決し Code Gear の並列実行を可能とする。

Gears OS の通常の処理を Computation、Computation のための Computation を Meta Computation として扱う。例として、 Code Gear が次に実行する

†1 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.
†2 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.

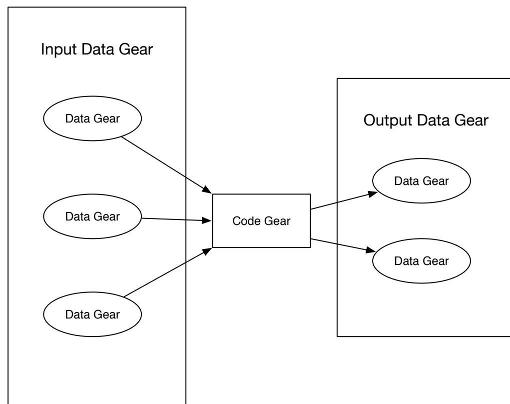

図 1　Gears OS の依存関係

Code Gear を goto で名前指定する。この継続処理に対して Meta Code Gear が名前を解釈して、処理を対応する Code Gear に引き渡す。これらは、従来の OS の Dynamic Loading Library や Command 呼び出しに対応する。名前と Code Gear へのポインタの対応は Meta Data Gear に格納される。この Meta Data Gear を Context と呼び、これは従来の OS の Process や Thread を表す構造体に対応する。Meta Computation を使用することで以下のことが可能になる。

- 元の計算を保存したデータ拡張や機能の追加
- GPU 等のさまざまなアーキテクチャでの動作
- 並列処理や分散処理の細かいチューニングや信頼性の制御
- Meta Computation は 通常の Computaiton の間に挟まれる

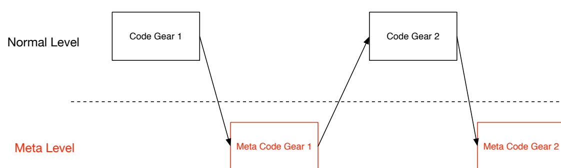

図 2　Meta_code_gear

2. GearsOS の構成

Gears OS の以下の要素で並列処理を行う。

- Context(Task)
- TaskManager
- Worker

Gears OS では、 Context という Meta Data Gear を通して Code Gear と Data Gear の接続を行う。また、 並列実行を行う際はこの Context を生成し、それを Task として実行する。そのため、 Context は 接続に必要な Code/Data Gear のリスト、 Data Gear を確保するためのメモリ空間、 実行する Code Gear、 Code Gear の実行に必要な Input Data Gear のカウンタ等をもっている。

TaskManager は Task、 Worker の生成、 Worker に生成した Task を送信する。また、生成した Worker の終了処理等を行う。

Worker は thread と実行する Task が入っている Queue を持っている。Worker は TaskManager から送信された Task を Queue から取り出し、Code Gear を実行する。Task は Context なので、Code Gear の実行に必要な Input Data Gear はその Context から参照される。Code Gear を実行した後は出力される Output Data Gear から依存関係を解決する。

Gears OS の並列処理の構成を図 3 に示す。

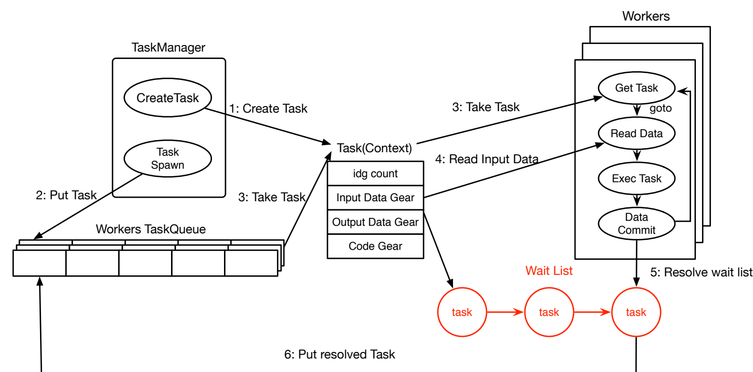

図 3　並列処理の構成

3. 並列処理の依存関係の解決

Gears OS の並列処理の依存関係の解決は Data Gear に依存関係解決のための Queue をもたせることで行う。Queue にはその Data Gear を Input Data Gear として使用する Task(Context) が入っている。依存関係の解決の流れは図 4 に示す。Worker は Task の Code Gear を実行後、 書き出された Output Data Gear の Queue から, 依存関係にある Task を参照する。参照した Task には 実行に必要な Input Data Gear のカウンタをもっているので、そのカウンタのデクリメントを行う。カウンタが 0 になったら Task が待っている Data Gear が揃ったことになるので、その Task を Worker に送信する。

4. GPGPU

GPGPU とは、元々は画像出力や画像編集などの画像処理に用いられる GPU を画像処理以外に利用する技術の事である。画像の編集はピクセル毎に行われ

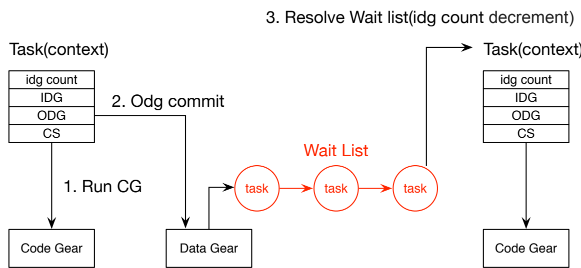

図 4 依存関係の解決

るため多大な数の処理を行う必要があるが、 GPU は CPU に比べコア数が多数あり、多数のコアで同時に計算することによって CPU よりも多数の並列な処理を行う事が出来る。これによって GPU は画像処理のような多大な処理を並列処理することで、 CPU で処理するよりも高速に並列処理することが出来る。しかし、GPU のコアは CPU のコアに比べ複雑な計算は出来ない構造であるため単純計算しか出来ない、また一般的にユーザーから GPU 単体に直接命令を書き込むことも出来ないなどの問題点も存在する。GPGPU は CPU によって単純計算の Task を GPU に振り分ける事によって、 GPU の問題点を解決しつつ、高速な並列処理を行うことである。また Data Gear へのアクセスは接続された Code Gear からのみであるから、処理中に変数が書き変わる事がない。図??では以下の流れで処理が行われる。

- Data Gear を Persistent Data Tree に挿入。
- TasMannager で実行する Code Gear と実行に必要な Data Gear への Key を持つ Task を生成。
- 生成した Task を TaskQueue に挿入。
- Worker の起動。
- Worker が TskQueue から Task を取得。
- 取得した Task を元に必要な Data Gear を Persistent Data Tree から取得。
- 並列処理される Code Gear を実行。

5. CUDA

CUDA とは NVIDIA 社が提供している並列コンピューティング用の統合開発環境で、コンパイラ、ライブラリなどの並列コンピューティングを行うのに必要なサポートを提供している。一般的にも広く使われている GPU の開発環境である。Task(kernel) は .ptx という GPU 用のアセンブラに変換され、プログラム内部から直接 kernel を呼び出す構文を持つ API である。さらにメモリ転送と kernel 呼び出しを自分で制御する DriverAPI の 2 種類をもつ。

6. CPUWoker

Worker thread で動く Task スケジューラーである。synchronized queue から Task の List を読み込み実行する。Data Gear の待ち合わせの管理を行う。CPUWorker は receive Task という API を持ち、Task がなくなるまで繰り返す。

7. CUDAWorker の実装

CPUWorker を再利用して作成する Task スケジューラー。CUDA ライブラリの初期化を行う以外の動作は CUDAWorker と全く同じになる。GPU へのデータ転送及び GPU 側での Task の実行は Task の Meta Code Gear で行われる。

8. Gears OS の並列構文

Gears OS では 並列実行する Task の設定をメタレベルで Code1 のように行っている。Code1 では 実行する CodeGear、待ち合わせ中の Input Data Gear の数、Input/Output Data Gear への参照等の設定を記述している。これらの記述は Context などを含む煩雑な記述であるが、並列実行されることを除けば通常の CbC の goto 文と同等である。そこで、Context などを直接用いないノーマルレベルの並列構文を用意することが望ましい。Code2 のような par goto を用いる記述方法を新たに考案した。この記述は メタレベル生成スクリプトにより、 Code1 に変換される。また、メタレベルの記述を変換することで、 CPU、GPU での実行の切り替え等を行うことが出来るようになる。

```
__code createTask(TaskManager* taskManager, Context*
      task, Integer *integer1, Integer *integer2,
      Integer *output) {
    task->next = C_add; // set Code Gear
    task->idgCount = 2; // set Input Data Gear Counter
    task->data[task->idg] = (union Data*)integer1; //
         set Input Data Gear reference
    task->data[task->idg+1] = (union Data*)integer2;
    task->maxIdg = task->idg + 2;
    task->odg = task->maxIdg; // Output Data Gear
        index
    task->data[task->odg] = (union Data*)output; //
        set Output Data Gear reference
    task->maxOdg = task->odg + 1;
    taskManager->next = C_createTask1;
    goto meta(context, taskManager->taskManager->
        TaskManager.spawn); // spawn task
}

// code gear
__code add(Integer *integer1, Integer *integer2,
      Integer *output) {
    ....
}
```

Code 1 createTask

```
__code createTask(Integer *integer1, Integer *
      integer2, Integer *output, __code next(...)) {
    par goto add(integer1, integer2, output, __exit);
    goto next(...);
}
```

Code 2　parGoto

par goto で生成された Task は`__exit`と言う continuation に goto することにより終了する。終了時には Task の Context が解放される。Gears OS では Task は output Data Gear を生成した時点で終了するので、par goto では直接に`__exit`に継続するのではなく、Data Gear に書き込み依存関係の処理をおこなう継続を指定する。これらの継続は interface での Data Gear の定義に従って生成される。

これにより Code Gear と Data Gear の依存関係をノーマルレベルで記述できるようになる。

9. 考　　察

従来の OS の並列処理は、例えば Unix の fork 、pthread の`pthread_create`で記述される。これらはプロセス ID や thread 構造体を含むメタレベルの記述であり、並列処理の煩雑さを解消できなかった。これに対して、プログラミング言語レベルで並列処理構文を用意する方法がある。例えば Java の thread あるいは Go 言語の go routine などである。これらは OS によって直接サポートされているわけではない。言語レベルでの並列処理を細かく制御する場合は言語機能そのものに変更を加える必要がある。例えば、 Java の thread や go routine を GPU で動かすためには言語のサポートを待つ必要がある。

並列処理ではチューニングが重要であり、どの Task を どの CPU とメモリに割り当てるかを細かく制御する必要がある。これらを可能にするためには言語レベルでのサポートを非常に複雑にするか、言語レベルでの記述を諦めて OS レベルでの記述をおこなう必要がある。

Gears OS ではノーマルレベルとメタレベルの記述を分離することにより、並列処理の煩雑な記述を避け、チューニングなどをメタレベルの記述で行うことが可能になる。メタレベルの記述の変更は現状では CbC レベルで行う。goto 文の行き先を実行時に変更することも可能であり (goto meta)、この場合は実行時のチューニングなどのメタ計算を行うことに相当する。

プログラムの信頼性をノーマルレベルで示しておけば、メタ計算がノーマルレベルの意味を保存すれば信頼性はそのまま維持される。このように Gears OS ではノーマルレベルでのプログラムの信頼性を再利用しながら、メタ計算により様々な拡張を行うことができる。拡張部分の信頼性は独立して示す必要がある。これらは Gears OS のモデル検査の機能などを用いて示すことになる。

10. ま と め

Code Gear Data Gear を用いて CUDA を利用した並列処理プログラムを記述した。CUDA 専用のコンパイラの nvcc と Code Gear Data Gear のコンパイラを CMake を用いる事で両立させた。Gears OS での GPU の基本的な実行を確認することができた。これらの部分は Gears OS のメタ計算として実装されている。par goto 構文の実装は今後の課題である。

参 考 文 献

1) 河野真治，杉本　優：Code Segment と Data Segment によるプログラミング手法，第 54 回プログラミング・シンポジウム (2013).
2) : CUDA, https://developer.nvidia.com/category/zone/cuda-zone/.
3) TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
4) 伊波立樹，東恩納琢偉，河野真治：Code Gear、Data Gear に基づく OS のプロトタイプ，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) (2016).