Gears OS の並列処理

2018/02/20

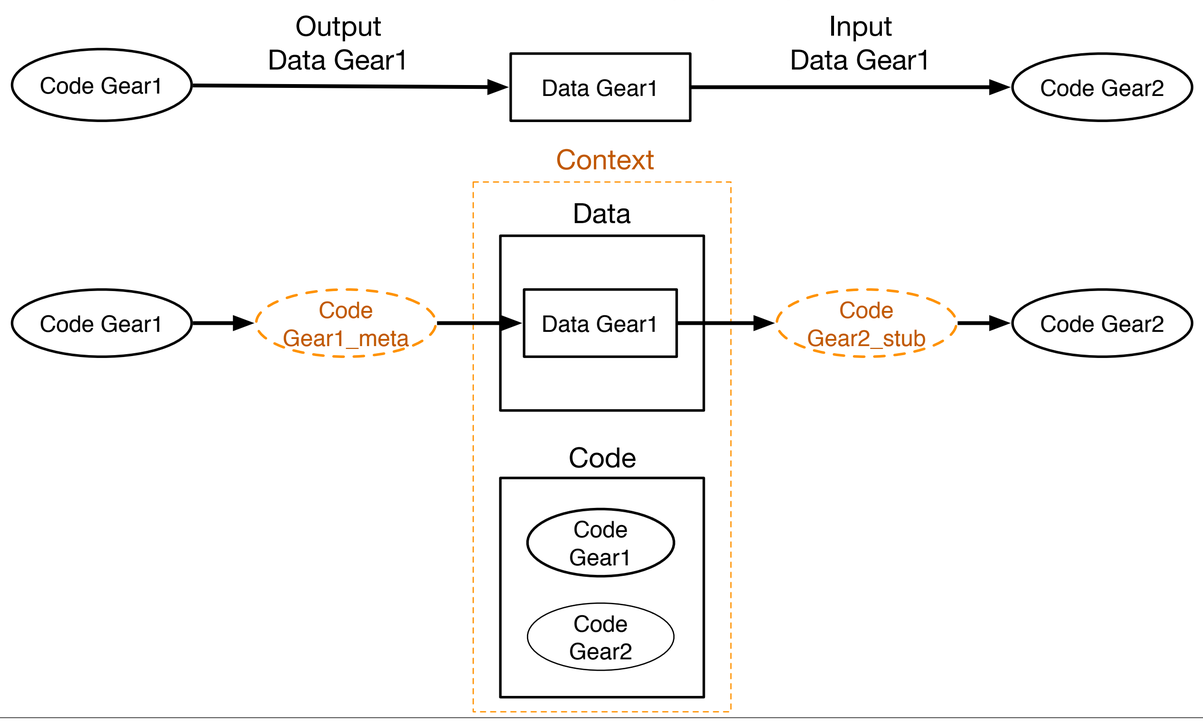
伊波立樹　並列信頼研

メタ計算を使った並列処理

・並列処理のチューニングや信頼性を保証するのは難しい
  - 従来のテストやデバッグでは非決定的な実行の対処が困難
  - GPU などのアーキテクチャに合わせたプログラミング
・Gears OS は通常の計算をノーマルレベル、信頼性と拡張性の計算をメタレベルに階層化することを目指して開発している
・Gears OS の並列処理機構、並列処理構文(par goto)の実装、Gears OS を実装するにつれて必要なったモジュール化の導入を行う

Gears OS の概念

・Gears OS は処理の単位として Code Gear、 データの単位として Data Gear を用いて構成される
・Code Gear は必要な Input Data Gear が揃ったら実行し、 Output Data Gear を生成する
・Code Gear 間の移動は軽量継続という呼び出し元の環境を持たない継続で行う。 軽量継続は goto 構文で記述する
・メタレベルの計算は Code Gear の接続間で実行され、 Code/Data Gear に対応した Meta Code/Data Gear で構成される
・Gears OS では Context という 全ての Code Gear と Data Gear を参照できる Meta Data Gear をメタ計算(stub Code Gear)で参照し、メタレベルからノーマルレベルへの継続を行う



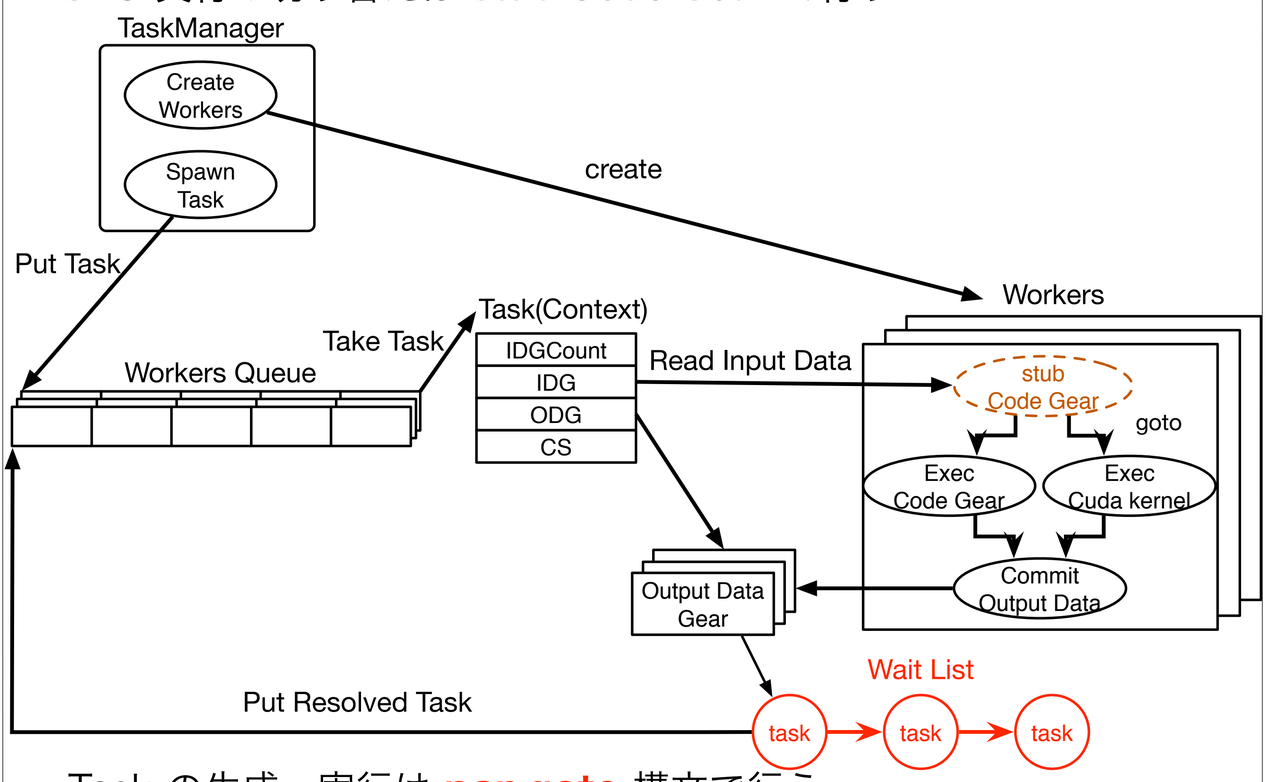
Interface

・Gears OS を実装するにつれて、stub Code Gear の記述が煩雑になることがわかった
・モジュール化の仕組みとしてInterface を導入した
・Interface はある Data Gear とそれに対するAPI である Code Gear の集合を表現する Data Gear
・Interface を導入することで、Stack や Queue などのデータ構造を仕様と実装に分けて記述できる
・Interface の定義には API の引数群の型、API 自体の Code Gear の型を記述し、 定義から stub Code Gear を自動生成する

```
typedef struct Queue<Impl>{
        // Data Gear parameter
        union Data* queue;
        union Data* data;
        __code next(...);
        __code whenEmpty(...);
        // Code Gear
        __code clear(Impl* queue, __code next(...));
        __code put(Impl* queue, union Data* data, __code
next(...));
        __code take(Impl* queue, __code next(union Data*, ...));
        __code isEmpty(Impl* queue, __code next(...), __code
whenEmpty(...));
} Queue;
```

並列処理の構成

・今回、並列処理機構をInterface を用いて実装を行った
・Task(Context)
  - Context はTaskに相当し、従来のプロセスやスレッドとして扱う
・TaskManager
  - CPU、 GPU 分 の Worker を生成し、管理を行う
  - 依存関係を解決した Task を各 Worker の Queue に送信する
・Worker
  - 送信された Task を取得し、Code Gear の実行を行う
・ Synchronized Queue
  - TaskManager と Worker 間の通信を行うための Queue
  - マルチスレッド間でのデータの同期を CAS を使用して行う
・並列処理の依存関係の解決は Data Gear がメタレベルで持っている Queue を使用して行う
・GPU 実行の切り替えは stub Code Gear で行う

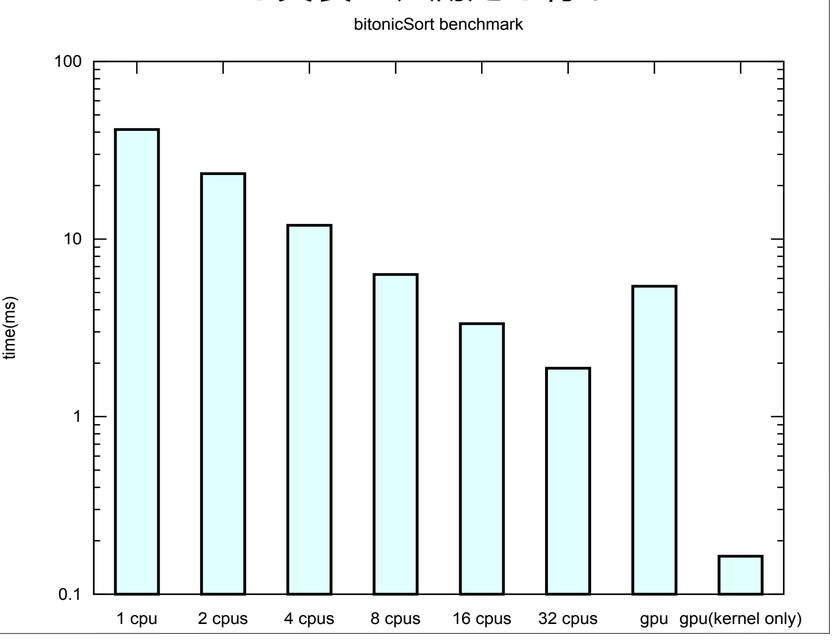


・ Task の生成、実行は **par goto** 構文で行う
・ par goto 構文は Context を参照するためメタレベルに変換される

```
__code code1(Integer *integer1, Integer *integer2, Integer
*output) {
    par goto add(integer1, integer2, output, __exit);
    goto code2();
}
```

・並列処理の例題として BitonicSort を実装し、測定を行った
  - 要素数 2^24

・1 CPU から 32 CPU で約22.12倍の速度向上
・GPU では kernel のみの実行では 32 CPU の約11.48倍になった



まとめと今後の課題

・Gears OS の並列処理機構を Interface を用いて実装を行った
  - Interface を導入することで見通しの良い Gears OS のプログラミングが可能になった
  - par goto 構文を導入することで、ノーマルレベルで並列処理の記述が可能になった
・今後の課題
  - 並列処理の信頼性を証明とモデル検査で保証する
  - 並列処理のオーバーヘッドの解消