Gears OS のモジュール化と並列 API

宮城光希[†1]　桃原優[†1]　河野真治[†2]

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 Continuation based C によってアプリケーションと OS そのものを記述する。CbC はこの Code Gear と Data Gear の単位でプログラムを記述する。システムやアプリケーションを記述するために Code Gear と Data Gear を柔軟に再利用する必要がある。このときに機能を接続する API と実装の分離が可能であることが望ましい。Gears OS の信頼性を保証するために、形式化されたモジュールシステムを提供する必要がある。本論文では、Interface を用いたモジュールシステムの説明とその応用としての並列 API について考察する。並列 API は継続を基本とした関数型プログラミングと両立する必要がある。ここでは、CbC の goto 文を拡張した par goto 文を導入する。par goto では並列実行のための実行 Context を生成し、TaskScheduler に登録する。Gears OS での同期機構は Data Gear の待ち合わせとして定義する。メタレベルではこれらの同期機能は CAS とそれを用いて実装した Synchronized Queue になる。これらの Queue も Interface を用いてモジュール化されている。モジュール化の詳細と、有効性について考察する。

Mitsuki MIYAGI ,[†1] Yu TOBARU [†1] and Shinji KONO [†2]

1. OS の拡張性と信頼性の両立

さまざまなコンピュータの信頼性の基本はメモリなどの資源管理を行う OS である。OS の信頼性を保証する事自体が難しいが、時代とともに進歩するハードウェア、サービスに対応して OS 自体が拡張される必要がある。OS は非決定的な実行を持ち、その信頼性を保証するには、従来のテストとデバッグでは不十分であり、テストしきれない部分が残ってしまう。これに対処するためには、証明を用いる方法とプログラムの可能な実行をすべて数え上げるモデル検査を用いる方法がある。モデル検査は無限の状態ではなくても巨大な状態を調べることになり、状態を有限に制限したり状態を抽象化したりする方法が用いられている図 1。

証明やモデル検査を用いて OS を検証する方法はさまざまなものが検討されている。検証は一度ですむものではなく、アプリケーションやサービス、デバイスが新しくなることに検証をやり直す必要がある。つまり信頼性と拡張性を両立させることが重要である。

†1 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.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

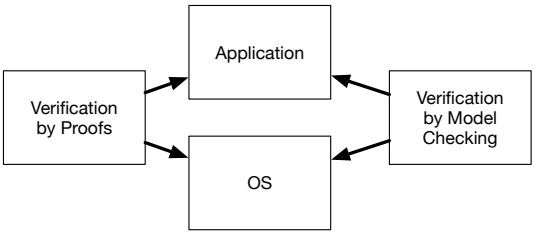

図 1: 証明とモデル検査による OS の検証

コンピュータの計算はプログラミング言語で記述されるが、その言語の実行は操作的意味論の定義などで規定される。プログラミング言語で記述される部分をノーマルレベルの計算と呼ぶ。コードが実行される時には、処理系の詳細や使用する資源、あるいは、コードの仕様や型などの言語以外の部分が服属する。これをメタレベルの計算という。この二つのレベルを同じ言語で記述できるようにして、ノーマルレベルの計算の信頼性をメタレベルから保証できるようにしたい。ノーマルレベルでの正当性を保証しつつ、新しく付け加えられたデバイスやプログラムを含む正当性を検証したい。

ノーマルレベルとメタレベルを共通して表現できる言語として Continuation based C(以下 CbC)[1] を用いる。CbC は関数呼出時の暗黙の環境 (Environment) を使わずに、コードの単位を行き来できる引数

付き goto 文 (parametarized goto) を持つ C と互換性のある言語である。これを用いて、Code Gear と Data Gear、さらにそのメタ構造を導入する。これらを用いて、検証された Gears OS を構築したい。図 2。検証には定理証明支援系である Agda を用いる。Gears の記述をモジュール化するために Interface を導入した。さらに並列処理の記述用に par goto 構文を導入する。これらの機能は Agda 上で継続を用いた関数型プログラムに対応するようになっている。

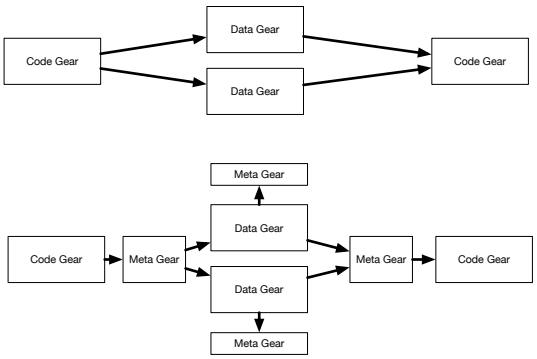

図 2: Gears のメタ計算

本論文では Interface と par goto の実装の詳細を記述し評価を行った。マルチ CPU と GPU 上での par goto 文の実行を確認した。

2. Gears におけるメタ計算

Gears OS ではメタ計算を柔軟に記述するためのプログラミング言語の単位として Code Gear、Data Gear という単位を用いる。Code Gear、Data Gear にはそれぞれメタレベルの単位である Meta Code Gear、Meta Data Gear が存在し、これらを用いてメタ計算を実現する。

CbC は軽量継続 (goto 文) による遷移を行うので、継続前の Code Gear に戻ることはなく、状態遷移ベースのプログラミングに適している。CbC は LLVM[2] と GCC[?] 上で実装されている。メタレベルの記述は Perl スクリプトによって生成される stub と goto meta によって Code Gear で記述される。

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface 作成時に Code Gear の集合を指定することにより複数の実装を持つことができる。Interface の操作に対応する Code Gear の引数は Interface に定義されている Data Gear を通して指定される。一つの実行スレッド内で使われる Interface の Code Gear と Data Gear は Meta Data Gear に格納される。この Meta Data Gear を Context という。ノーマルレベルでは Context を直接見ることはできない。

Code Gear の継続は関数型プログラミングからみると継続先の Context を含む Closure となっている。これを記述するために継続に不定長引数を追加する構文をスクプリトの変換機能として用意した図。1 メタ計算側ではこれらの Context を常に持ち歩いているので goto 文で引数を用いることはなく、行き先は Code Gear の番号のみで指定される。

```
__code popSingleLinkedStack(struct SingleLinkedStack*
      stack, __code next(union Data* data, ...)) {
   if (stack->top) {
       data = stack->top->data;
       stack->top = stack->top->next;
   } else {
       data = NULL;
   }
   goto next(data, ...);
}
```

Code 1: 不定長引数を持つ継続

これにより Interface 間の呼び出しを C++ のメソッド呼び出しのように記述することができる。Interface の実装は、Context 内に格納されているので、オブジェクトごとに実装を変える多様性を実現できている。呼び出された Code Gear は必要な引数を Context から取り出す必要がある。これは スクリプトで生成された stub Meta Code Gear で行われる。Gears OS でのメタ計算は stub と goto のメタ計算の 2 箇所で実現される。

Context を複製して複数の CPU に割り当てることにより並列実行を可能になる。これによりメタ計算として並列処理を記述したことになる。Gears のスレッド生成は Agda の関数型プログラミングに対応して行われるのが望ましい。そこで、par goto 構文を導入し、Agda の継続呼び出しに対応させることにした。par goto では Context の複製、入力の同期、タスクスケジューラーへの Context の登録などが行われる。par goto 文の継続として、スレッドの join に相当する __exit を用意した。__exit により par goto で分岐した Code Gear の出力を元のスレッドで受け取ることができる。

関数型プログラムではメモリ管理は GC などを通して暗黙に行われる。Gears OS ではメモリ管理は stub などのメタ計算部分で処理される。例えば、寿命の短いスレッドでは使い捨ての線形アロケーションを用いる。

3. Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
- TaskQueue

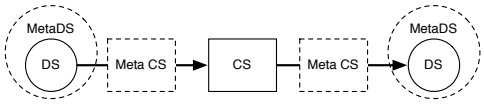

図 3: Gears でのメタ計算

- TaskManager
- Worker

図 4 に Gears OS の構成図を示す。

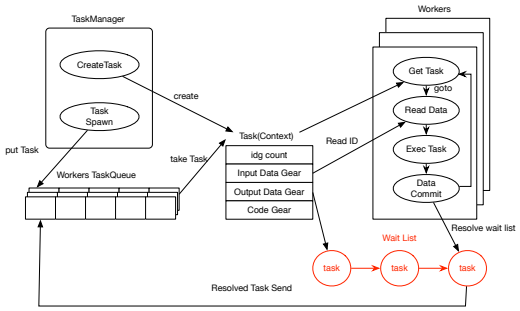

図 4: Gears OS の構成図

Data Gear は union と struct によって表現される。Context には Data Gear の Data Type の情報が格納されている。この情報から確保する Data Gear のサイズなどを決定する。

Context は Task でもあり、Task は通常の OS のスレッドに対応する。Task は実行する Code Gear と Data Gear をすべて持っている。TaskManager は Task を実行する Worker の生成、管理、Task の送信を行う。Gears OS における Task Queue は Synchronized Queue で実現される。Worker は TaskQueue から Task である Context を取得し、Task の Code Gear を実行し、Output Data Gear の書き出しを行っている。Input/Output Data Gear の依存関係が解決されたものから並列実行される。

4. Interface

Interface は呼び出しの引数になる Data Gear の集合であり、そこで呼び出される Code Gear のエントリである。呼び出される Code Gear の引数となる Data Gear はここで全て定義される。

Code2 は stack の Interface である。Code Gear、Data Gear に参照するために Context を通す必要があるが、Interface を記述することでデータ構造の api と Data Gear を結びつけることが出来る。

```
typedef struct Stack<Impl>{
    union Data* stack;
    union Data* data;
    union Data* data1;
    __code whenEmpty(...);
    __code clear(Impl* stack,__code next(...));
    __code push(Impl* stack,union Data* data, __code
          next(...));
    __code pop(Impl* stack, __code next(union Data*,
          ...));
    __code pop2(Impl* stack, union Data** data, union
          Data** data1, __code next(union Data**,
          union Data**, ...));
    __code isEmpty(Impl* stack, __code next(...),
          __code whenEmpty(...));
    __code get(Impl* stack, union Data** data, __code
          next(...));
    __code get2(Impl* stack,..., __code next(...));
    __code next(...);
} Stack;
```

Code 2: Stack の Interface

Code3 は stack の実装の例である。createImpl は関数呼び出しで呼び出され、初期化と Code Gear のスロットに対応する Code Gear の番号を入れる。

```
Stack* createSingleLinkedStack(struct Context*
      context) {
    struct Stack* stack = new Stack();
    struct SingleLinkedStack* singleLinkedStack = new
          SingleLinkedStack();
    stack->stack = (union Data*)singleLinkedStack;
    singleLinkedStack->top = NULL;
    stack->push = C_pushSingleLinkedStack;
    stack->pop = C_popSingleLinkedStack;
    stack->pop2 = C_pop2SingleLinkedStack;
    stack->get = C_getSingleLinkedStack;
    stack->get2 = C_get2SingleLinkedStack;
    stack->isEmpty = C_isEmptySingleLinkedStack;
    stack->clear = C_clearSingleLinkedStack;
    return stack;
}

__code clearSingleLinkedStack(struct
     SingleLinkedStack* stack,__code next(...)) {
    stack->top = NULL;
    goto next(...);
}

__code pushSingleLinkedStack(struct SingleLinkedStack
     * stack,union Data* data, __code next(...)) {
    Element* element = new Element();
    element->next = stack->top;
    element->data = data;
    stack->top = element;
    goto next(...);
}
```

Code 3: SingleLinkedStack の実装

5. stub Code Gear の生成

stub Code Gear は Code Gear 間の継続に挟まれる Code Gear が必要な Data Gear を Context から取り出す処理を行うものである。Code Gear 毎に記述する必要があり、その Code Gear の引数を見て取り出す Data Gear を選択する。stub Code Gear を

自動生成する generate stub を Perl スクリプトで作成することによって Code Gear の記述量を約半分にすることができる。

stub を生成するために generate_stub は指定された cbc ファイルの __code 型である Code Gear を取得し、引数から必要な Data Gear を選択する。generate_stub は引数と interface を照らし合わせ、Gearef または GearImpl を決定する。また、この時既に stub Code Gear が記述されている Code Gear は無視される。

cbc ファイルから、生成した stub Code Gear を加えて stub を加えたコードに変換を行う。(Code4)

```
__code clearSingleLinkedStack(struct Context *context
     ,struct SingleLinkedStack* stack,enum Code next
     ) {
    stack->top = NULL;
    goto meta(context, next);
}

__code clearSingleLinkedStack_stub(struct Context*
     context) {
    SingleLinkedStack* stack = (SingleLinkedStack*)
         GearImpl(context, Stack, stack);
    enum Code next = Gearef(context, Stack)->next;
    goto clearSingleLinkedStack(context, stack, next
         );
}
```

Code 4: stub Code Gear を加えたコード

6. Context の生成

generate_context は Context.h、Interface.cbc、generate_stub で生成された Impl.cbc を見て Context を生成する Perl スクリプトである。

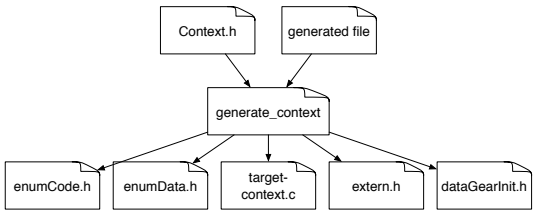

図 5: generate_context による Context の生成

Context は Meta Data Gear に相当し、Code Gear や Data Gear を管理している。

generate_context は context の定義 (Code??) を読み宣言されている Data Gear を取得する。Code Gear の取得は指定された generate_stub で生成されたコードから __code 型を見て行う。取得した Code Gear、Data Gear の enum の定義は enumCode.h、enumData.h に生成される。

Code/Data Gear の名前とポインタの対応は generate_context によって生成される enum Code、enum Data を指定することで接続を行う。また、generate context は取得した Code/Data Gear から initContext の生成も行う。

Context には Allocation 等で生成した Data Gear へのポインタが格納されている。Code Gear は Context を通して Data Gear へアクセスする。Data Gear の Allocation を行うコードは dataGearInit.c に生成される。

7. Gears OS の並列処理

Gears OS では実行の Task を Code Gear と Input/Output Data Gear の組で表現する。Input/Output Data Gear によって依存関係が決定し、それにそって並列実行を行う。Gears OS では 並列実行する Task を Context で表現する。Context には Task 用の情報として、実行される Code Gear、Input/Output Data Gear の格納場所、待っている Input Data Gear のカウンタ等を持っている Task の Input Data Gear が揃っているかを TaskManager で判断し、 実行可能な Task を Worker に送信する。Worker は送信された Task が指定した Code Gear を実行し、Output Data Gear を書き出す。

8. SynchronizedQueue

SynchronizedQueue は Worker の Queue として使用される。Worker の Queue は TaskManager を経由して Task を送信するスレッドと Task を取得する Worker 自身のスレッドで扱われる。そのため SynchronizedQueue はマルチスレッドでもデータの一貫性を保証する Queue を実装する必要がある。

データの一貫性を保証する解決例としての 1 つとしてロックを使った解決方法がある。しかし、ロックを行ってデータを更新した場合、同じ Queue に対して操作を行う際に待ち合わせが発生し、全体の並列度が下がってしまう。そこで、Gears OS ではデータの一貫性を保証するために CAS(Check and Set、Compare and Swap) を利用した Queue を実装している。CAS は値の比較、更新をアトミックに行う命令である。CAS を使う際は更新前の値と更新後の値を渡し、渡された更新前の値を実際に保存されているメモリ番地の値と比較し、同じならデータ競合がないため、データの更新に成功する。異なる場合は他に書き込みがあったとみなされ、値の更新に失敗する。

Gears OS ではこの CAS を行うための Interface を定義している (Code5)。この Interface では、Data Gear 全てを内包している Data 共用体のポインタの値を更新する CAS を定義している (Code5 6 行目)。

```
typedef struct Atomic<Impl>{
    union Data* atomic;
    union Data** ptr;
    union Data* oldData;
```

```
    union Data* newData;
    __code checkAndSet(Impl* atomic, union Data** ptr
          , union Data* oldData, union Data* newData,
          __code next(...), __code fail(...));
    __code next(...);
    __code fail(...);
} Atomic;
```

Code 5: AtomicInterface

AtomicInterface での CAS の実際の実装を Code6 に示す。実際の実装では __sync_bool_compare_and_swap 関数を呼び出すことで CAS を行う (Code6 2 行目)。この関数は第一引数に渡されたアドレスに対して第二引数の値から第三引数の値へ CAS を行う。CAS に成功した場合、true を返し、失敗した場合は false を返す。Code6 では CAS に成功した場合と失敗した場合それぞれに対応した Code Gear へ継続する。

```
__code checkAndSetAtomicReference(struct
     AtomicReference* atomic, union Data** ptr,
     union Data* oldData, union Data* newData,
     __code next(...), __code fail(...)) {
   if (__sync_bool_compare_and_swap(ptr, oldData,
        newData)) {
      goto next(...);
   }
   goto fail(...);
}
```

Code 6: CAS の実装

SynchronizedQueue の Data Gear の定義を Code7 に示す。SynchronizedQueue はデータのリストの先頭と、終端のポインタを持っている。Queue を操作する際はこのポインタに対して CAS をすることでデータの挿入と取り出しを行う。

```
struct SynchronizedQueue {
    struct Element* top;
    struct Element* last;
    struct Atomic* atomic;
};

// Singly Linked List element
struct Element {
    union Data* top;
    struct Element* next;
};
```

Code 7: SynchronizedQueue の定義

9. 並 列 構 文

Gears OS の並列構文は par goto 文で用意されている (Code8)。

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

Code 8: par goto による並列実行

par goto の引数には Input/Output Data Gear と実行後に継続する __exit を渡す。par goto で生成された Task は __exit に継続することで終了する。par goto 文でも 通常の goto 分と同様にメタへの goto 文へ置き換えられるが、par goto 文では通常の goto 文とは異なるメタへと継続する。Gears OS の 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 __exit に継続するのではなく、Output Data Gear への書き出し処理 (Commit) に継続される。

Code Gear は Input に指定した Data Gear が全て書き込まれると実行され、実行した結果を Output に指定した Data Gear に書き出しを行う。Code Gear の引数である__next が持つ引数の Data Gear が Output Data Gear となる。

書き出し処理は Data Gear の Queue から、依存関係にある Task を参照する。参照した Task が持つ実行に必要な Input Data Gear カウンタのデクリメントを行う。カウンタが 0 になると Task が待っている Input Data Gear が揃ったことになり、その Task を TaskManager 経由で 実行される Worker に送信する。

この par goto 文は通常のプログラミングの関数呼び出しのように扱うことができる。

10. Gears OS の評価

10.1 実 験 環 境

今回 Twice、 BitonicSort をそれぞれ CPU、GPU 環境で Gears OS の測定を行う。

使用する実験環境を表 1、 GPU 環境を表 2 に示す。また、今回は LLVM/Clang で実装した CbC コンパイラを用いて Gears OS をコンパイルする。

Model	Dell PowerEdgeR630
OS	CentOS 7.4.1708
Memory	768GB
CPU	2 x 18-Core Intel Xeon 2.30GHz

表 1: 実行環境

GPU	GeForce GTX 1070
Cores	1920
Clock Speed	1683MHz
Memory Size	8GB GDDR5
Memory Bandwidth	256GB/s

表 2: GPU 環境

10.2 Twice

Twice は与えられた整数配列のすべての要素を 2 倍にする例題である。

Twice の Task は Gears OS のデータ並列で実行される。CPU の場合は配列ある程度の範囲に分割して Task を生成する。これは要素毎に Task を生成するとその分の Context を生成するために時間を取ってしまうからである。

Twice は並列実行の依存関係もなく、データ並列での実行に適した課題である。そのため、 通信時間を考慮しなければ CPU よりコア数が多い GPU が有利となる。

要素数 2^{27} のデータに対する Twice の実行結果を 表**??**、図 6 に示す。CPU 実行の際は 2^{27} のデータを 64 個の Task に分割して並列実行を行っている。GPU では 1 次元の block 数を 2^{15}、 block 内の thread 数を 2^{10} で kernel の実行を行った。ここでの “GPU“ は CPU、GPU 間のデータの通信時間も含めた時間、 “GPU(kernel only)“ は kernel のみの実行時間である。

Processor	Time(ms)
1 CPU	1181.215
2 CPUs	627.914
4 CPUs	324.059
8 CPUs	159.932
16 CPUs	85.518
32 CPUs	43.496
GPU	127.018
GPU(kernel only)	6.018

表 3: 2^{27} のデータに対する Twice

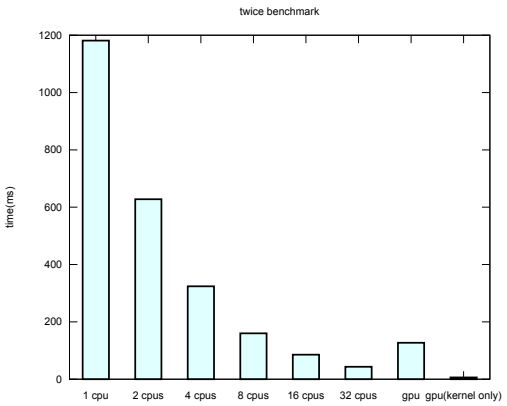

図 6: 2^{27} のデータに対する Twice

ある程度の台数効果があると考えられる。

GPU での実行は kernel のみの実行時間は 32 CPU に比べて 約 7.2 倍の速度向上が見られた。しかし、通信時間を含めると 16 CPU より遅い結果となってしまった。CPU、GPU の通信時間かオーバーヘッドになっている事がわかる。

10.3 BitonicSort

BitonicSort は並列処理向けのソートアルゴリズムである。代表的なソートアルゴリズムである Quick Sort も並列処理 を行うことが可能であるが、 Quick-Sort では ソートの過程で並列度が変動するため、台数効果が出づらい。一方で Bitonic Sort は最初から最後まで並列度が変わらずに並列処理を行う。図 7 は要素数 8 のデータに対する BitonicSort のソートネットワークである。

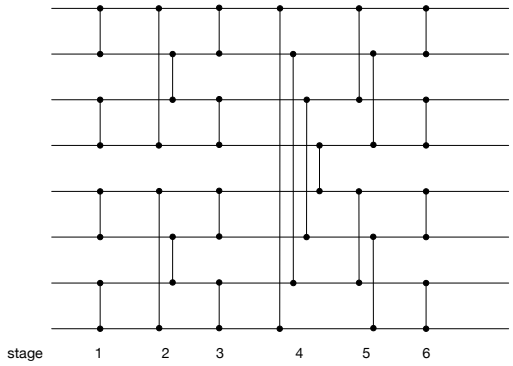

図 7: 要素数 8 の BitonicNetwork

BitonicSort はステージ毎に決まった 2 点間の要素の入れ替えを並列に実行することによってソートを行う。Gears OS ではこのステージ毎に Output Data Gear を書き出し、 次のステージの Code Gear の Input Data Gear として記述することで BitonicSort を実現する。

要素数 2^{24} のデータに対する BitonicSort の実行結果を 表 4、図 8 に示す。こちらも Twice と同じく CPU 実行の際は 2^{24} のデータを 64 個の Task に分割して並列実行を行っている。つまり生成される Task は 64 * ステージ数 となる。GPU では 1 次元の block 数を 2^{14}、 block 内の thread 数を 2^{10} で kernel の実行を行った。

Processor	Time(s)
1 CPU	41.416
2 CPUs	23.340
4 CPUs	11.952
8 CPUs	6.320
16 CPUs	3.336
32 CPUs	1.872
GPU	5.420
GPU(kernel only)	0.163

表 4: 2^{24} のデータに対する BitonicSort

1 CPU と 32 CPU で 約 22.12 倍 の速度向上が見られた。GPU では通信時間を含めると 8 CPU の約 1.16 倍となり、 kernel のみの実行では 32 CPU

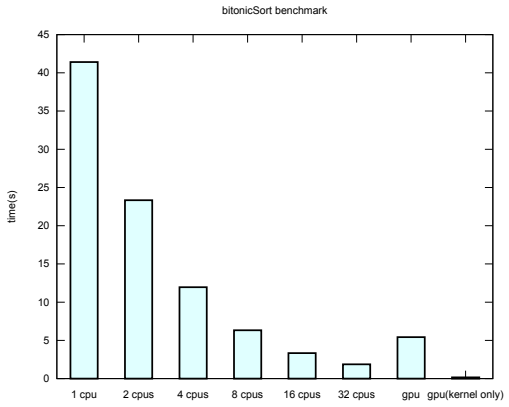

図 **8**: 2^{24} のデータに対する BitonicSort

の約 11.48 倍となった。現在の Gears OS の CUDA 実装では、 Output Data Gear を書き出す際に一度 GPU から CPU へ kernel の実行結果の書き出しを行っており、その処理の時間で差が出たと考えられる。GPU で実行される Task 同士の依存関係の解決の際は CuDevicePtr などの GPU のメモリへのポインタを渡し、CPU でデータが必要になったときに初めて GPU から CPU へデータの通信を行うメタ計算の実装が必要となる。

10.4 OpenMP との比較

OpenMP[?] は C、 C++ のプログラムにアノテーションを付けることで並列化を行う。アノテーションを Code 9 のように for 文の前につけることで、ループの並列化を行う。

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    // Processing
}
```

Code 9: OpenMP での Twice

OpenMP は既存のコードにアノテーションを付けるだけで並列化を行えるため、変更が少なくて済む。しかし、 ループのみの並列化ではプログラム全体の並列度が上がらずアムダールの法則により性能向上が頭打ちになってしまう。OpenMP はループの並列化ではなくブロック単位での並列実行もサポートしているが、アノテーションの記述が増えてしまう。また、OpenMP はコードとデータを厳密に分離していないため、データの待ち合わせ処理をバリア等のアノテーションで記述する。

Gears OS では Input Data Gear が揃った Code Gear は並列に実行されるため、プログラム全体の並列度を高めることが出来る。また 並列処理のコードとデータの依存関係を par goto 文で簡潔に記述することが出来る。

Gears OS と OpenMP で実装した Twice の実行結果の比較を図 9 に示す。実行環境は 表 1、 2^{27} のデータに対して行い、Gears OS 側は配列を 64 個の Task に分割し、OpenMP は for 文を static スケジュールで並列実行した。static スケジュールはループの回数をプロセッサーの数で分割し、並列実行を行う openMP のスケジュール方法である。

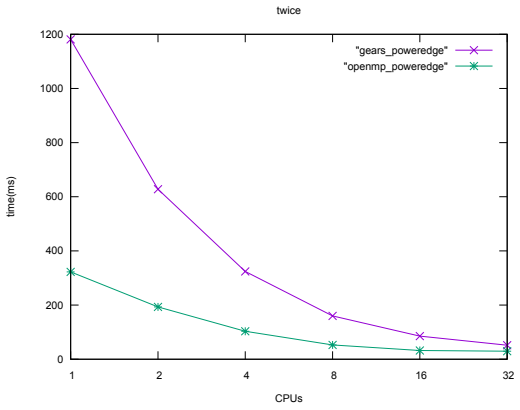

図 **9**: vs OpenMP

実行結果として OpenMP は 1CPU と 32CPU で約 10.8 倍の速度向上がみられた。一方 Gears OS では約 27.1 倍の速度向上のため、台数効果が高くなっている。しかし、Gears OS は 1CPU での実行時間が OpenMP に比べて大幅に遅くなっている。

10.5 Go 言語との比較

Go 言語 は Google 社が開発しているプログラミング言語である。Go 言語による Twice の実装例を code 10 に示す。

```
func main() {
    c := make(chan []int)
    for i :=0; i < *split; i++ {
        // call goroutine
        go twice(list, prefix, i, c);
    }

    for i :=0; i < *split; i++ {
        // join twice routins
        <- c
    }
}

func twice(list []int, prefix int, index int, c chan
      []int) {
    for i := 0; i < prefix; i++ {
        list[prefix*index+i] = list[prefix*index+i] *
             2;
    }
    c <- list
}
```

Code 10: Go 言語での Twice

Go 言語は並列実行を “go function(argv)” のような構文で行う。この並列実行を goroutine と呼ぶ。

Go 言語は goroutin 間のデータ送受信をチャネルというデータ構造で行う。チャネルによるデータの送受信は “<-” を使って行われる。例えばチャネルのデータ構造である channel に対して “channel <- data” とすると、 data を channel に送信を行う。“<- channel” とすると、 channel から送信されたデータを 1 つ取り出す。channel にデータが送信されていない場合は channel にデータが送信されるまで実行をブロックする。Go 言語はチャネルにより、データの送受信が簡潔に書ける。しかし、チャネルは複数の goroutine で参照できるためデータの送信元が推測しづらい。

Gears OS では goroutine は par goto 文とほぼ同等に扱うことが出来る。また、Code Gear は par goto 文で書き出す Output Data Gear を指定して実行するため、Data Gear の書き出し元が推測しやすい。

Go 言語での OpenMP と同様に Twice を実装し Gears OS と比較を行う。こちらも実行環境は 表 1、2^{27} のデータに対して行い、Gears OS、Go 言語両方とも配列を 64 個の Task、 goroutine に分割して並列実行を行った。

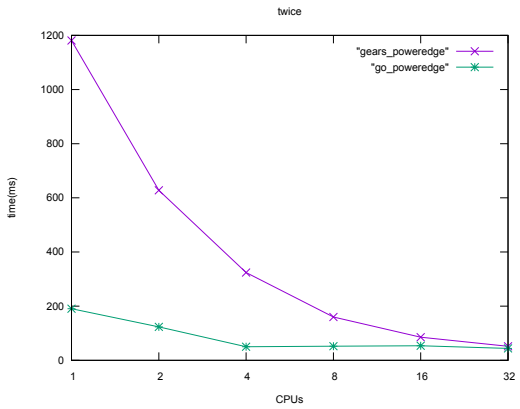

図 10: vs Go

実行結果として Go 言語は 1CPU と 32CPU で約 4.33 倍の速度向上が見られた。

11. 結　　論

本論文では Gears OS のプロトタイプの設計と実装、メタ計算である Context と stub の生成を行う Perl スクリプトの記述、並列実行機構の実装を行った。Code Gear 、Data Gear を処理とデータの単位として用いて Gears OS を設計した。Code Gear、Data Gear にはメタ計算を記述するための Meta Code Gear、Meta Data Gear が存在する。メタ計算を Meta Code Gear、によって行うことでメタ計算を階層化して行うことができる。Code Gear は関数より細かく分割されてるためメタ計算を柔軟に記述できる。Gears OS は Code Gear と Input/Output Data Gear の組を Task とし、並列実行を行う。

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface は複数の実装をもち、Meta Data Gear として定義される。従来の関数呼び出しでは引数をスタック上に構成し、関数の実装アドレスを Call するが、Gears OS では引数は Context 上に用意された Interface の Data Gear に格納され、操作に対応する Code Gear に goto する。

Context は使用する Code Gear、Data Gear をすべて格納している Meta Data Gear である。通常の計算から Context を直接扱うことはセキュリティ上好ましくない。このため Context から必要なデータを取り出して Code Gear に接続する Meta Code Gear である stub Code Gear を定義した。stub Code Gear は Code Gear 毎に記述され、Code Gear 間の遷移に挿入される。

並列処理を行う際は Context を生成し、 Code Gear と Input/Output Data Gear を Context に設定して TaskManager 経由で各 Worker の SynchronizedQueue に送信される。Context の設定はメタレベルの記述になるため、ノーマルレベルでは par goto 文という CbC の goto 文に近い記述で並列処理を行える。この par goto は通常のプログラミングの関数呼び出しのように扱える。

これらのメタ計算の記述は煩雑であるため Perl スクリプトによる自動生成を行なった。これにより Gears OS のコードの煩雑さは改善され、ユーザーレベルではメタを意識する必要がなくなった。

Twice と BitonicSort の例題の測定結果では 1CPU と 32CPU で Twice では約 27.1 倍、BitonicSort では 約 22.12 倍の速度向上が見られた。また、GPU 実行の測定も行い、kernel のみの実行時間では 32 CPU より Twice では約 7.2 倍、BitonicSort では約 11.48 倍の速度向上がみられ、GPU の性能を活かすことができた。

今後の課題は、Go、OpenMP との比較から、Gears OS が 1CPU での動作が遅いということがわかった。Gears OS は par goto 文を使用することで Context を生成し、並列処理を行う。しかし、Context はメモリ空間の確保や使用する全ての Code/Data Gear を設定する必要があり、生成にある程度の時間がかかってしまう。そこで、 par goto のコンパイルタイミングで実行する Code Gear のフローをモデル検査で解析し、処理が軽い場合は Context を生成せずに、関数呼び出しを行う等の最適化を行¿うといったチュー

ニングが必要である。

参 考 文 献

1) TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
2) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California (2004).
3) 河野真治，伊波立樹，東恩納琢偉：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).
4) Sigurbjarnarson, H., Bornholt, J., Torlak, E. and Wang, X.: Push-button Verification of File Systems via Crash Refinement, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, USENIX Association, pp. 1–16 (2016).
5) Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E. and Wang, X.: Hyperkernel: Push-Button Verification of an OS Kernel, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, ACM, pp.252–269 (2017).
6) Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V. and Costanzo, D.: CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, USENIX Association, pp. 653–669 (2016).
7) Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp.207–220 (2009).
8) Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. and Heiser, G.: Comprehensive Formal Verification of an OS Microkernel, *ACM Trans. Comput. Syst.*, Vol.32, No.1, pp.2:1–2:70 (2014).
9) Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F. and Zeldovich, N.: Using Crash Hoare Logic for Certifying the FSCQ File System, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 18–37 (2015).
10) Yang, J. and Hawblitzel, C.: Safe to the Last Instruction: Automated Verification of a Type-safe Operating System, *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, New York, NY, USA, ACM, pp. 99–110 (2010).
11) Moggi, E.: Computational Lambda-calculus and Monads, *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Piscataway, NJ, USA, IEEE Press, pp. 14–23 (1989).