GPUを用いたマルチスレッドアルゴリズムの効率的な実現に関する一考察

岡田, 圭太 / OKADA, Keita

(開始ページ / Start Page)
1
(終了ページ / End Page)
22
(発行年 / Year)
2017-03-24
(学位授与年月日 / Date of Granted)
2017-03-24
(学位名 / Degree Name)
修士(工学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)

2016 年度修士論文

GPU を用いたマルチスレッドアルゴリズムの効率的な実現に関する一考察

指導教員　和田　幸一

法政大学大学院理工学研究科

応用情報工学専攻修士課程

学生証番号　15R4109

氏名　岡田　圭太

目次

1. はじめに

1.1 研究背景・目的

GPU(Graphics Processing Unit)は一般的に画像処理を専門とする演算装置であり,多くの場合では,CPU と呼ばれる主演算装置の制御のもとで用いられている動画信号生成専用の補助演算用 IC である.高速の VRAM と接続され,グラフィクスシェーディングに特化したプロセッサが多く集まった構造を持っている.一つ一つのプロセッサの構造は単純なため,その機能は CPU に比べて限定されたものであるが,大量のデータを複数のプロセッサで同時かつ並行処理することができる.[1]

近年では,この GPU の機能を画像処理のみならず,GPU の持つ特定処理の高いパフォーマンスに着目し,その処理能力を HPC(ハイパフォーマンス・コンピューティング)のコンピューティングパワーに生かす GPGPU(General Purpose GPU)に注目が集まっている.このことから,本論文では GPU を用いたマルチスレッドアルゴリズムに着目する.

マルチスレッドアルゴリズム[1]とは,複数のスレッドを使用して並列に処理を行うことができるアルゴリズムである.小さな粒度（スレッドレベル）で計算処理を行うことができ,現実的に実現しやすい.マルチスレッドアルゴリズムについての研究では,行列乗算やマージソートのマルチスレッド化,Cilk によるマルチスレッドランタイムシステム[1]など,様々な考案が生み出されてきた.

本論文では、マルチスレッドアルゴリズムを GPU 上で実現するための方法論を開発することを目的とする。特に、GPU で再帰手続きの並列化が可能な DP(Dynamic Parallelism)によるマルチスレッドアルゴリズムの実現について考察する。

ここでは,フィボナッチ数列とハノイの塔という代表的な再帰関数のアルゴリズムによって実現できる問題を取り上げる.これらの問題を DP(Dynamic Parallelism)で実現するときの問題点を明らかにし,実用サイズで実現できるアルゴリズムを与える.また,ハノイの塔に対してはこれまでよりも真に高速なアルゴリズムを与えた.

1.2 本論文の構成

本論文の構成は以下のとおりである.まず始めに,2 章では GPU について説明する.3 章では本論文のキーワードとなるマルチスレッドアルゴリズムの概念やその理論的な効率は「仕事量」と「スパン」という 2 つの尺度を用いて測れることを述べる.4 章では DP を用いることで GPU のカーネル関数内から,そのまま再帰関数を呼び出せることを示す.5 章では,フィボナッチ数列が GPU を用いて実用的なサイズで実現できる並列化手法を説明し,その理論評価を行う.6 章では,ハノイの塔が GPU を用いて実用的なサイズで実現できる並列化手法を説明し,その理論評価を行う.7 章では 5,6 章で得られた結果をもとに,その考察を行う.最後に 8 章では本論文の結論と,今後マルチスレッドアルゴリズムの開発を行う上での課題について述べる.

2. GPU[2]

GPU は,具体的には,パソコンの拡張スロットに装着するビデオカード（グラフィクス・ボード）に搭載される,描画処理機能に特化したチップや,CPU に付随するチップセットの中に組み込まれる,描画処理機能の部分のハードウェアのことである.大まかにいうと,ビデオカード上には GPU とビデオ･メモリ(VRAM)が置かれている.そして,ビデオカードは,PCI Express バスのスロットに装着して使う.

ここで,NVIDIA 社の GPU のアーキテクチャを図 1 に示す.さらに,GPU チップの内部にある,ストリーミング・マルチプロセッサ(SM)の内部の構造を図 2 に示す.GPU の構造はコアの世代と型番で異なるが,アーキテクチャはほぼ同じである.

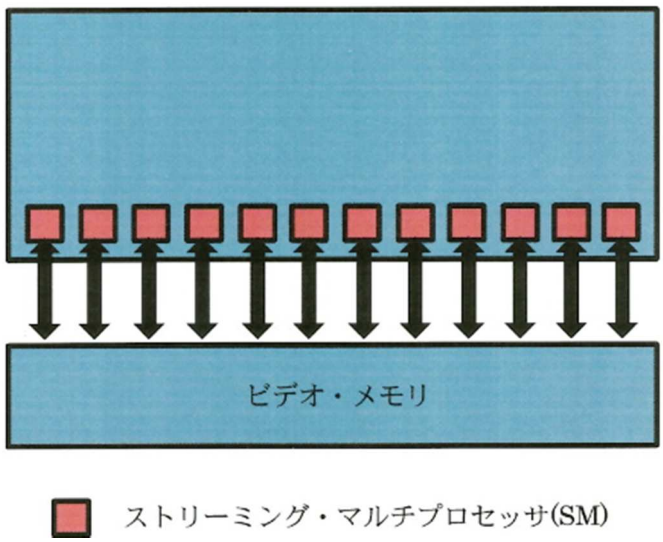

図 1 .NVIDIA 製の GPU のアーキテクチャ

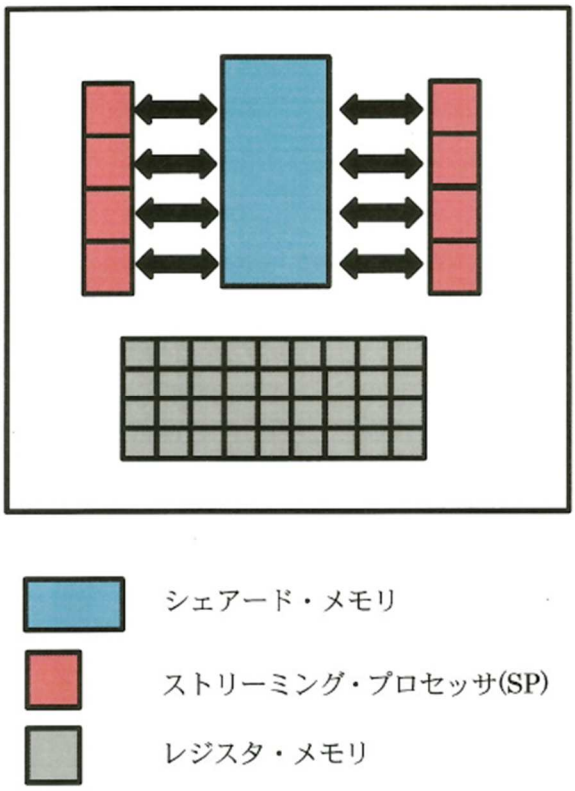

図 2.ストリーミング・マルチプロセッサ(SM)の内部構造

ビデオ・メモリと GPU とは,メモリ・インターフェイスで接続されている.

CPU の場合はメイン・メモリとせいぜい 64bit 幅で接続されているが,GPU の場合はハイエンドでは 512bit であるなど,ビデオ・メモリと高速にデータ転送ができるアーキテクチャになっている.

GPU チップの内部には,ストリーミング・マルチプロセッサ(SM)が多数入っている.さらに SM の内部には,ストリーミング・プロセッサ(SP)という最小単位の演算処理ユニットが 8 個ある.

3.　マルチスレッドアルゴリズム[1]

チップマルチプロセッサやほかの共有記憶並列コンピュータ上でプログラムを書くとき,共通して利用できる道具の１つが静的スレッドである.しかし,直接に静的スレッドを用いて共有記憶並列コンピュータのプログラムを書くことは難しい.理由の１つは,各スレッドの負荷がおおよそ等しくなるように仕事を動的に分割する必要があるからである.極端に簡単な場合を除けば,仕事の負荷分散を行うスケジューラを実現するために,プログラマは複雑な通信プロトコルを使わなければならない.このような事情から,並列計算資源を調整し,スケジュールし,管理するための並行性プラットフォームが構築されることとなった.

並行性プラットフォームの重要なクラスの１つが動的マルチスレッドであり,このモデルを本研究では採用する.動的マルチスレッドを用いると,通信プロトコル,負荷分散などに悩むことなくプログラマは並列性を指定できる.並行性プラットフォームは計算を自動的に負荷分散するスケジューラを含むので,プログラマの作業は劇的に減ることになる.動的マルチスレッド環境が持つ機能は進化しつつも,ほぼすべての環境では入れ子並列性と並列ループが提供されている.入れ子並列性によってサブルーチンを“産み出す”ことができる.並列ループは通常の for 文に似ているが,ループのすべての繰り返しが並列に実行できるところが違う.

この 2 つの並列性が本論で述べる動的マルチスレッドモデルの基礎である.このモデルで最も重要な点は,プログラマは計算の中の論理的並列性を指定するだけでよく,スケジューラがスレッドをスケジュールし,負荷分散をする.

本研究の動的マルチスレッドモデルには以下に述べる重要な利点がある.

・逐次プログラミングモデルの単純な拡張になっている.擬似コードに 3 つの並行性に関するキーワード parallel, spawn, sync を導入することで,マルチスレッドアルゴリズムが記述できる.また,マルチスレッド化された擬似コードからこれらの並行キーワードを削除すると,残されたコードは同じ問題に対する逐次擬似コードになる.これをマルチスレッドアルゴリズムの「逐次化」と呼ぶ.
・「仕事量」と「スパン」という概念に基づいて並列度を定量化する用法がある.
・マルチスレッドアルゴリズムは分割統治法から自然に生成されることが多い.さらに,逐次型の分割統治アルゴリズムと同様,マルチスレッドアルゴリズムも漸化式を用いる解析に適している.

次にフィボナッチ数列を例にとり,動的マルチスレッドについて説明する.

まずはじめに,フィボナッチ数列 FIB(n)を漸化式

$$\mathrm{FIB}(0) = 0$$
$$\mathrm{FIB}(1) = 1$$
$$\mathrm{FIB}(\mathrm{i}) = \mathrm{FIB}(\mathrm{i}-1) + \mathrm{FIB}(\mathrm{i}-2)(i \geq 2) \cdots ①$$

によって定義する.n 番目のフィボナッチ数を求める再帰的な逐次アルゴリズムを図 3 に示す.

```
FIB(n)
1  if n ≤ 1
2        return n
3  else x = FIB(n-1)
4        y=FIB(n-2)
5        return x+y
```

図 3.n 番目のフィボナッチ数を求める再帰的な逐次アルゴリズム

次に,FIB(6)の計算が呼び出す再帰の木を図 4 に示す.

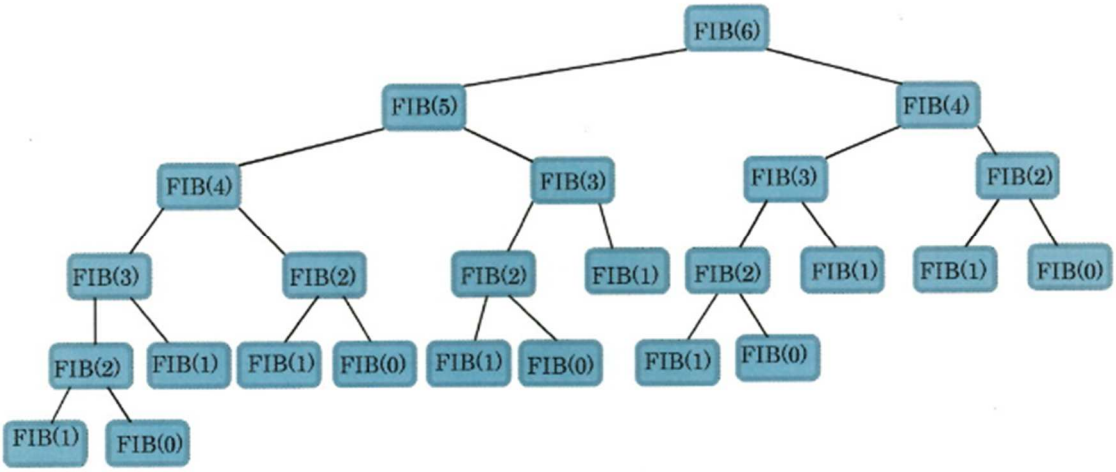

図 4.FIB(6)を計算するときの再帰木

図 4 から,FIB(6)を呼び出すと,FIB(5)と FIB(4)が再帰的に呼び出される.しかし,FIB(5)を呼び出したときにも,FIB(4)が再び呼び出される.そして,2 つの FIB(4)は同じ結果(FIB(4) = 3)

を返す.手続き FIB では履歴管理をしないことから, 2 回目の FIB(4)の呼び出しでも,最初の呼び出しが行った仕事をそのまま繰り返す.

図 3 の FIB(n)は n に関して指数関数的に増加する.また,いくつも同じ関数が生成されるので,これはフィボナッチ数を計算するうえで極めて非効率な方法となる.しかし,マルチスレッドアルゴリズムの解析における概念を説明するには適している.図 3 の手続き FIB(n)の第 3 行と第 4 行における 2 回の再帰呼び出し,FIB(n - 1)と FIB(n - 2)は任意の順序で呼び出すことができ,一方の計算が他方に影響を与えることはない.したがって,これらの呼び出しを並列に実行することができる.

次に,私たちは並行性キーワードである spawn と sync を,並列性を指示するために擬似コードに付加する.図 5 に FIB 手続きの動的マルチスレッド化である P-FIB 手続きを示す.

```
P-FIB(n)
1  if n ≤ 1
2        return n
3  else x = spawn P-FIB(n-1)
4        y=P-FIB(n-2)
5        sync
6        return x+y
```

図 5.FIB 手続きの動的マルチスレッド化である P-FIB 手続き

並行性キーワード spawn と sync を P-FIB から除去すると FIB と同じ擬似コードが残る.あるマルチスレッドアルゴリズムの逐次化を,そのアルゴリズムからマルチスレッド化キーワード,すなわち,spawn と sync,そして並列ループを含むときには parallel を除去することで構成される逐次アルゴリズムと定義する.マルチスレッド擬似コードは,その逐次化が同じ問題を解く普通の逐次擬似コードになるという性質を持つ.

図 5 の第 3 行のように,spawn が手続き呼び出しの前にあるとき,入れ子並列性が発生する.生成（spawn）と普通の呼び出し（call）の違いは,生成を実行する親は,逐次実行で通常発生するように,生成したサブルーチンである子の実行が終了するのを待つのではなく,並列に実行を続けることができる点にある.P-FIB では,生成された子が P-FIB(n - 1)を計算している間に,子の実行と並列に,親は第 4 行の P-FIB(n - 2)の計算に進むことができる.P-FIB 手続きは再帰的だから,これら 2 つのサブルーチン呼び出しは,入れ子になった並列性を生成する.このように,部分計算の巨大な木を生成し,これらすべての部分計算を並列に実行する.

しかし,キーワード spawn は,手続きが生成した子らを同時に実行することができると言っているだけである.並列性キーワードは,計算のどの部分が並列に実行可能かを示す論理

的並列性を表現する.実際に実行する部分計算を決定する役割はスケジューラに任されている.

手続きは,図 5 の第 5 行にある sync 文を実行するまでは,生成した子が返す値を安全に利用できない.つまり,キーワード sync があると,この手続きが生成した子がすべての計算を終了するまで,この手続きは sync の次の文の実行に進めない.P-FIB 手続きでは,x が計算される前に x と y の和を求めるエラーを回避するために,第 6 行の return の前に sync が必要となる.sync による明示的な同期操作のほかに,任意の手続きは値を返す前に(明示されていない)sync を実行し,それが生成した子が既に終了していることを保証する.

フィボナッチ数列を動的マルチスレッド化したプログラムを実行するときに,プロセッサ群が実行する命令の集合であるマルチスレッド計算は,図 6 のような計算ダグと呼ぶ有効非巡回グラフとして考えると理解しやすい.

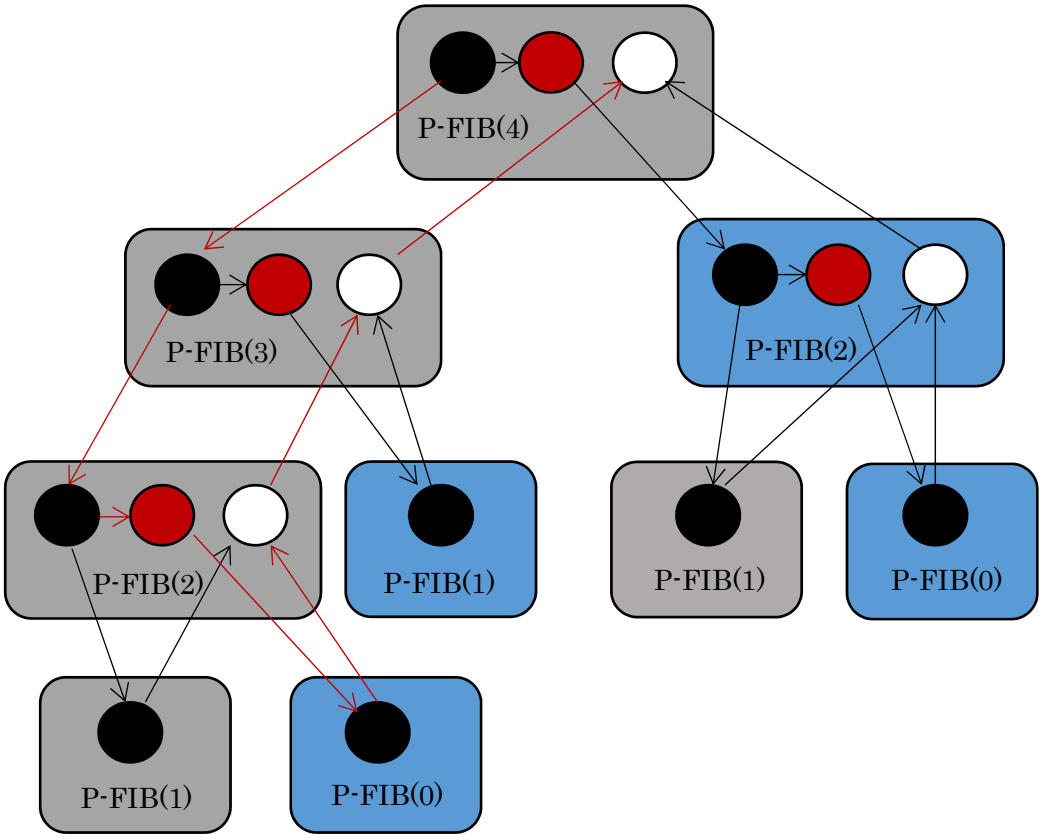

図 6.P-FIB(4)の計算を表す有効非巡回グラフ.各円は 1 つのストランドを表す.黒い円は図 5 の第 3 行の P-FIB(n - 1)の spawn までの部分を表す.赤い円は,図 5 の第 4 行で P-FIB(n - 2)を呼び出してから第 5 行で sync を実行するまでの手続きの部分を表す.ただし,P-FIB(n - 1)が値を返すまで実行は中断する.白い円は図 5 の sync の後で,手続きが x と y の和を計算し,

その結果を返す部分を表す.同じ手続きに属するストランドのグループを,それぞれ角の丸い長方形で囲み,生成された手続きを灰色の長方形,呼び出された手続きを青い長方形で示す.生成,または呼び出しを示す辺は下,継続を示す辺は右,戻り先を示す辺は上を向いている.

マルチスレッドアルゴリズムの理論的な効率は「仕事量」と「スパン」という 2 つの尺度を用いて測ることができる.マルチスレッドアルゴリズムの仕事量は,1 台のプロセッサで全体の命令を実行するのに必要な時間である.言い換えると,仕事量は各ストランドにかかる時間の総和である.各ストランドが単位時間で実行できる計算ダグの仕事量はそのダグの頂点数である.スパンは,ダグの任意のパスに沿ってストランドを実行するときにかかる時間の最大値である.各ストランドが単位時間で実行できるダグでは,スパンはダグの最長路,すなわち,クリティカルパス（赤矢印）の頂点数に等しい.たとえば,図 6 の計算ダグは全部で 17 個の頂点を持ち,その中の 8 個がクリティカルパス上にある.したがって,各ストランドが単位時間で実行できるなら,仕事量は 17 単位,スパンは 8 単位時間である.

しかし,マルチスレッドアルゴリズムの実際の実行時間は,仕事量とスパンだけで決まるのではない.利用できるプロセッサ台数とスケジューラによるストランドのプロセッサへの割り当てにも依存する.P 台のプロセッサ上でのアルゴリズムの実行時間を添字 P を用いてT_Pと表す.仕事量は 1 台のプロセッサ上での実行時間T_1である.スパンは各ストランドをそれぞれ別のプロセッサ上で実行できる場合,すなわち,無限個のプロセッサを持っている場合の実行時間なのでT_∞で表す.

ここで,P 台のプロセッサを持つ理想並列コンピュータ上で貪欲スケジューラ[1]は仕事量がT_1,スパンがT_∞であるマルチスレッド計算を

$$T_P \leq {}^{T_1}/_{P} + T_\infty \quad \cdots ②$$

時間で実行することができる.(2)の式を利用して,仕事量・スパンだけでなく,T_Pで比較する必要もある.

ここで$\phi = (1+\sqrt{5})/2$を黄金比としたときの,フィボナッチ数列の一般的な仕事量とスパン,そしてT_Pを表 1 に示す.

表 1.フィボナッチ数列の一般的な仕事量とスパン,T_P

仕事量(T_1)	スパン(T_∞)	T_P
O(ϕ^n)	O(n)	O(${}^{\phi^n}/_{p} + n$)

4.　Dynamic Parallelism

まずはじめに,従来の方法と DP を用いた場合とを比較した,CPU,GPU 間の内部動作を図 7 に示す.

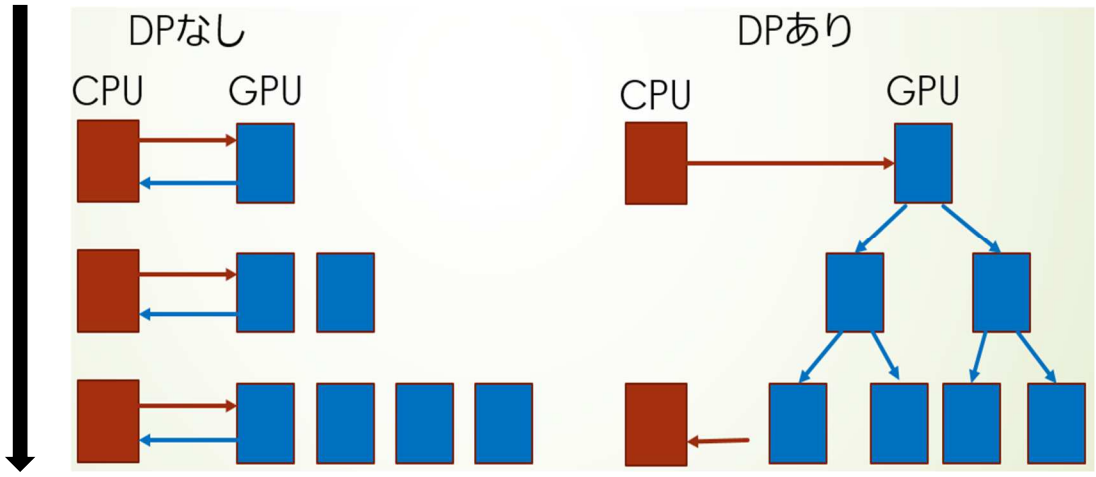

図 7.従来の方法と DP を用いた場合の内部動作

図 7 の左側（DP なし）のように,従来の方法だとカーネル関数（GPU 上で動作する関数）は CPU からでしか呼び出すことができなかった.このため,一度 GPU でカーネル関数の計算処理を行った後に次のカーネル関数を動作させるには,GPU から CPU にカーネルの実行終了を通知し,再び CPU 側からカーネルを実行させる必要があった.このため,カーネルの実行のたびに「CPU→GPU」と「GPU→CPU」の通信が必要である.

これに対し,図 7 の左側(DP あり)のように DP を用いることで,カーネル関数で計算処理を行い,再びカーネル関数を呼び出したいときには GPU のカーネル内からそのまま別のカーネルを呼び出すことができる.これにより CPU を介さずにすむので,「CPU→GPU」と「GPU→CPU」という通信のオーバーヘッドを省くことができる.

しかし,DP を用いる際に注意すべき点がある.カーネル関数内から再び別のカーネルを呼び出すプログラムでは,図 8 のように GPU 側で再帰木が形成される.そのときの再帰木の深さ(Depth)には制限があり,現在はその上限が 24 となっている.そのことから,GPU 側で再帰を行う際にはそのここに数式を入力します。制限を考慮して再帰プログラムを作成しなければならない.Depth が 24 を超えてしまうと正常に処理が行われない可能性がある.

Depth(≤24)

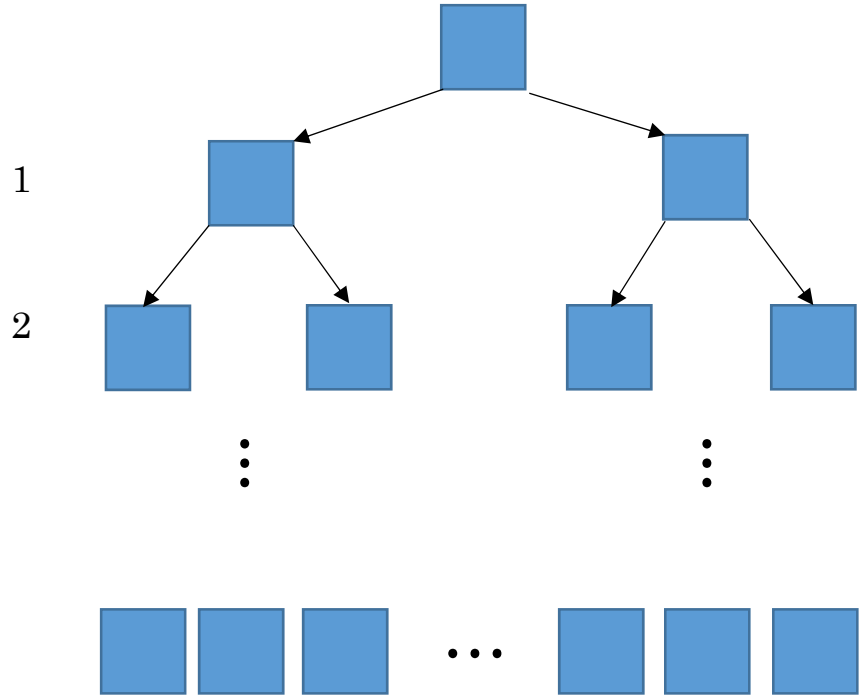

図 8.DP の Depth

5. フィボナッチ数列

次に,フィボナッチ数列の別の計算処理方法を示し,それらの理論評価を行う.

表 1 のT_Pは指数関数的に増加するので,①のフィボナッチ数列の計算効率は悪いことが分かる.また,Depth の上限が 24 であることから,DP で処理できる n の値は$n \leq 25$となる.しかしこの n のサイズでは実用的ではない.ここでは,フィボナッチ数列を表現する別の漸化式にもとづいて,実用的な n のサイズでフィボナッチ数列が計算できることを示す.

5.1 奇数・偶数で漸化式を分けたフィボナッチ数列

まずはじめに,n の値が奇数と偶数で処理する漸化式が異なるフィボナッチ数列について説明する.奇数の場合では

$$\mathrm{FIB}(2n+1) = 2(FIB(n))^2 + 2FIB(n)FIB(n-1) + (FIB(n-1))^2 \cdots ③$$

の漸化式でフィボナッチ数列の計算を行う.また,偶数では

$$\mathrm{FIB}(2n) = \left(FIB(\boldsymbol{n})\right)^2 + 2FIB(n)FIB(n-1) \cdots ④$$

の漸化式で処理を行う.ここで,②と③の漸化式を利用したアルゴリズムを図 9 に示す.

if n ≤ 5
　初期設定された値を出力

if(n % 2 == 0){
　　x = FIB$\left(\frac{n}{2}\right)$
　　y = FIB$(\frac{n}{2} - 1)$
　　return $x^2 + 2xy$
}

if(n % 2 == 1){
　　x = FIB$\left(\frac{n}{2} - \frac{1}{2}\right)$
　　y = FIB$(\frac{n}{2} - \frac{3}{2})$
　　return $2x^2 + 2xy + y^2$
}

図 9. 奇数・偶数で漸化式を分けたフィボナッチ数列のアルゴリズム

この方法で処理を行うと,偶数では FIB(n)に対して FIB(n / 2)と FIB(n / 2 - 1)の再帰関数を生成・呼び出すことになる. このことから,フィボナッチ数列の仕事量を T(n)とすると,その式は

$$T(n) = 2T\left(n/2\right) + C(C \text{ は定数}) \cdots ⑤$$

となる.また,③,④のフィボナッチ数列の仕事量T_1,スパンT_∞,そしてT_Pを表 2 に示す.

表 2. 奇数・偶数で漸化式を分けたフィボナッチ数列の仕事量,スパン,T_P

仕事量(T_1)	スパン(T_∞)	T_P
O(n)	O($\log n$)	O($n/p + \log n$)

表 1 と表 2 を比較すると,どの計算時間においても③,④のフィボナッチ数列の方が効率が良いことが分かる.また,DP で処理できる n の値は$\log_2 n \leq 24$となり,より実用的な n

のサイズでフィボナッチ数列を計算することができる.

また、③、④のフィボナッチ数列は再帰を用いず、繰り返し処理により$O(n)$で求めることができるが、今回の並列化アルゴリズムでは、P 台のスレッドで$O(n)$の並列アルゴリズムを実現できる。

6　ハノイの塔

次に,ハノイの塔について考える.まずはじめに,ハノイの塔の問題の定義を行う.初期の状態では,3 つの柱 A,B,C があり,いま,柱 A に n 枚の円盤が大きい順に下から積まれていることとする.

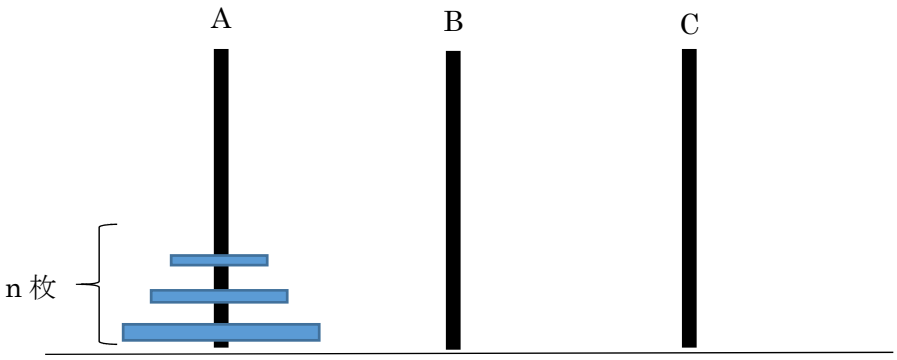

図 11.ハノイの塔の初期状態

図 11 の状態から,柱 A にある円盤すべてを柱 C に移すことを目的とする.ただし,以下のルールを守らなくてはならない.

(1)1 回に 1 枚しか動かしてはならない.また,小さな円盤にそれより大きな円盤を載せてはならない.
(2)すべてこの棒を使って動かすこと.棒以外のところに円盤を置いてはならない.

次に,ハノイの塔の一般的な解法とそのアルゴリズムを述べる.初期の状態で柱 A に n 枚の円盤があるとすると,柱 C にすべての円盤を動かすためには,まずはじめに一番大きな円盤を柱 C に積まなくてはならない.そのために,残りの n − 1 枚の円盤を柱 A から柱 B に移す必要があり,そうすることで一番大きな円盤を柱 A から柱 C に移動させることができる.

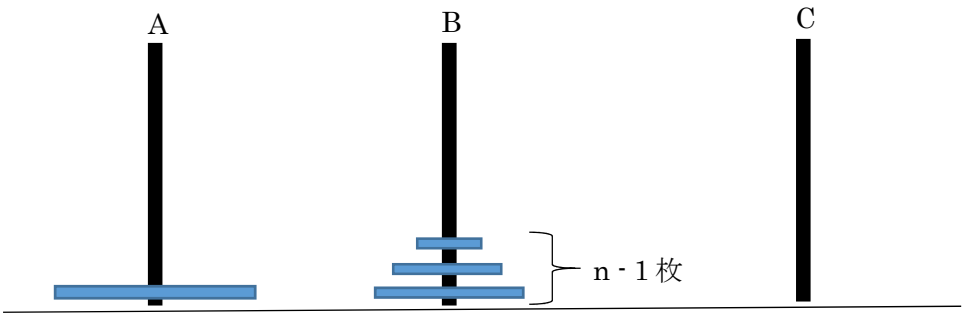

図 12.n – 1 枚の円盤を柱 B に移動させたハノイの塔

図 12 の状態になれば,柱 A にある一番大きな円盤を柱 C に動かすことができる.そして次の動作としては,柱 B にある残りの n – 1 枚の円盤を柱 C に移動させなければならない.そのために,さきほどと同様に,柱 B にある円盤で一番大きなものを除く n – 2 枚を柱 A に移動させれば,n 枚の中で 2 番目に大きい円盤は柱 C に移動することができる.以降同様の操作を繰り返し,最終的に n 枚の円盤すべてが柱 C に積まれることになる.

初期状態から,柱 A にある n 枚の円盤を柱 B を利用し,すべて柱 C に移動させるハノイの塔の関数を$\mathrm{h(n,a,b,c)(A=a,B=b,C=c}$ とする)とする.その時の上記のハノイの塔のアルゴリズムを図 13 に示す.

```
if(n > 0){
   call hanoi(n − 1, a, c, b)
   n 番目の円盤を柱 A から柱 C へ移動
   call hanoi(n − 1, b, a, c)
}
```

図 13.一般的なハノイの塔のアルゴリズム

図 13 から一般的なハノイの塔の仕事量を $\mathrm{H(n)}$とすると,その漸化式は

$$\mathrm{H(n)} = 2\mathrm{H(n-1)} + \mathrm{C}\big(\mathrm{C}\text{ は定数}\big)\cdots⑥$$

と表すことができる.⑥の漸化式も①のフィボナッチ数列と同じように,n に関して H の関数が指数関数的に増加し,その計算量は$\mathrm{O}(2^n)$となる.⑥の方法では,n の値が大きくなると計算するのに莫大な時間を要する.また,再帰の深さが n – 1 であることから,DP を利用できる n の値は$\mathrm{n-1 \le 24(n \le 25)}$となる.この n のサイズではハノイの塔を処理するのにあまり実用的とはならない.$\mathrm{H(n)} = \mathrm{o}(2^n)$となる解を与え,かつ DP で実装するときに n を実用的

なサイズまで扱えるようにしたい.

6.1　改良したハノイの塔の一般化漸化式

⑤の式を改良し,一般化した漸化式を提示することを試みる.まずはじめに$n = 4$とし,さきほどのハノイの塔の関数$h(n, a, b, c)$を計算した時の再帰的手続きのインスタンスの木を図14に示す.ここで図14では,再帰木の深さをその木の段数とする.

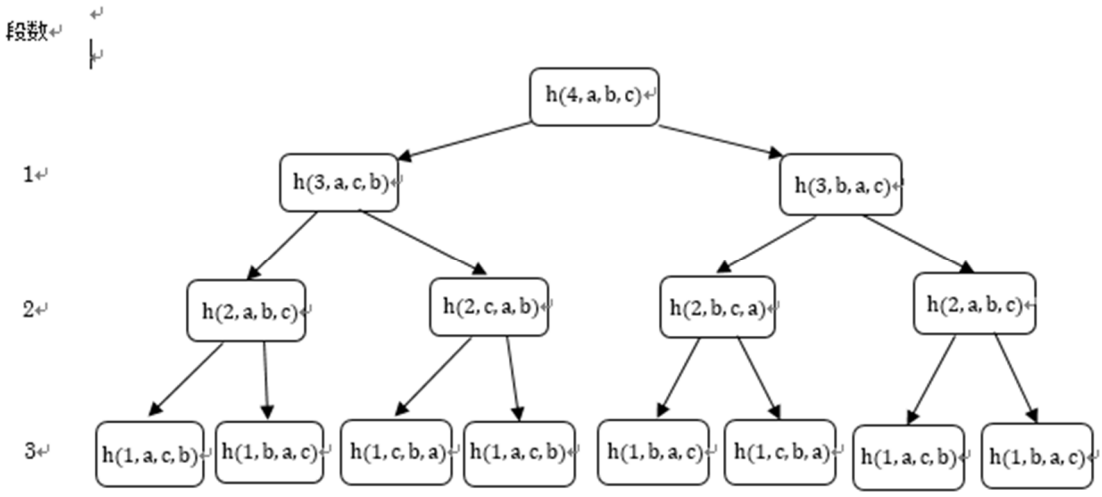

図 14. $n = 4$ の時のハノイの塔$h(n, a, b, c)$の再帰木

図14を見ると,段数が3である奇数の時には3つの関数$h(1, a, c, b), h(1, b, a, c), h(1, c, b, a)$が生成・呼び出しされている.さらにその3つの関数が$h(1, a, c, b) \to h(1, b, a, c) \to h(1, c, b, a)$という順に$8(= 2^3)$回繰り返し実行されていることが分かる.また,段数が偶数である2の場合には,奇数の時とは違う$h(2, a, b, c), h(2, c, a, b), h(1, b, c, a)$の3つの関数が生成・呼び出しされている.さらにその3つの関数が$h(2, a, b, c) \to h(2, c, a, b) \to h(2, b, c, a)$の順に$4(= 2^2)$回繰り返し実行されていることが分かる.

上記のことから,段数 k が奇数の時には 3 つの関数が$h(n - k, a, c, b) \to h(n - k, b, a, c) \to h(n - k, c, b, a)$という順に$2^k$回繰り返し実行される.また,段数 k が偶数の時には 3 つの関数が$h(n - k, a, b, c) \to h(n - k, c, a, b) \to h(n - k, b, c, a)$という順に$2^k$回繰り返し実行されることが分かる.各段数では高々3種類の関数が出現するので,始めの3つの関数を再帰で呼び出せば,残りはそれらの関数を参照すればよい.また,図14から,段数が3である場合の関数を再帰で呼び出すことになると,段数が2以下の時の7つ$(= 2^3 - 1)$の手続きが行われていないことになる.ゆえにその部分は再帰部とは別に非再帰部として計算しなければならない.

上記のことから,私たちは図 15 に示すハノイの塔のアルゴリズムを記述することができる.

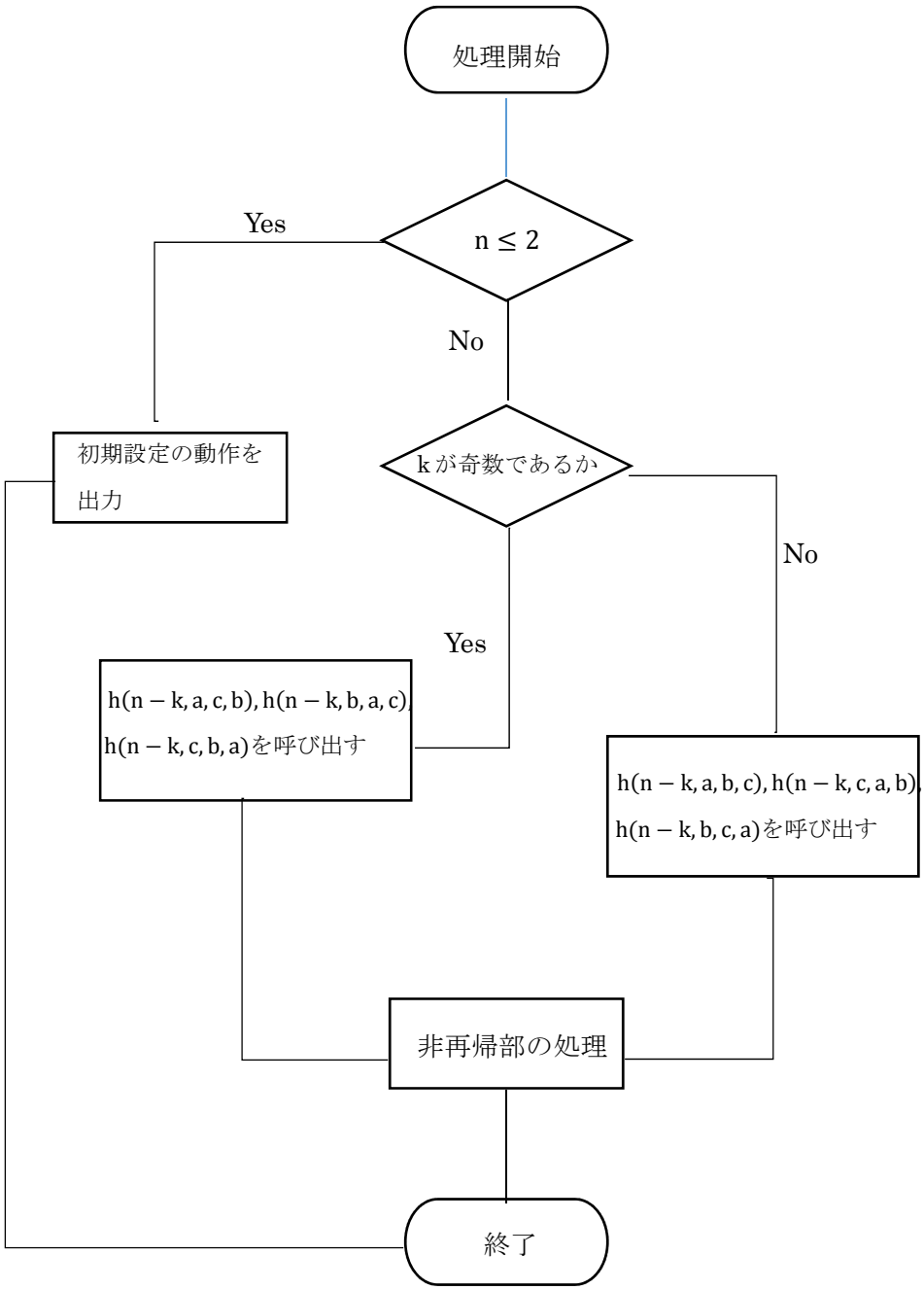

図 15.改良したハノイの塔のアルゴリズム

また,図 15 のアルゴリズムより,以下の改良したハノイの塔の仕事量 H(n)の式を提示することができる.

$$\mathrm{H(n)} = 3\mathrm{H(n-k)} + \mathrm{C}\cdot 2^{k}\left(\mathrm{C}\text{ は任意の定数}\right)\cdots ⑦$$

⑦の式より,k の値を大きくすることで非再帰部の計算量が増え,その代わり扱える n のサイズが大きくなる.また,k の値を小さくすると扱える n のサイズが小さくなるが,非再帰部の計算量が少なくなる.

例として,まずはじめに$\mathrm{k} = \frac{3}{4}n$の場合の⑦の式を考える.その時の⑦の漸化式は

$$\mathrm{T(n)} = 3\mathrm{T}\left({}^{n}/_{4}\right) + \mathrm{C}\cdot 2^{\frac{3}{4}n}\left(\mathrm{C}\text{ は定数}\right)\cdots ⑧$$

となる.⑧の時の再帰木の最大の深さは$\log_4 n$となり,扱える n の範囲が$\log_4 n \leq 24$となるので,非再帰部の計算量が$\mathrm{O}(2^{\frac{3}{4}n})$となる分,⑥の漸化式よりも実用的な n のサイズでハノイの塔を処理できる.

次に,$\mathrm{k} = {}^{n}/_{10}$の場合を考える.その時に与えられる漸化式は

$$\mathrm{T(n)} = 3\mathrm{T}\left(\frac{9}{10}n\right) + \mathrm{C}\cdot 2^{{}^{n}/_{10}}\left(\mathrm{C}\text{ は定数}\right)\cdots ⑨$$

となる.ここで,⑨の場合の再帰木の深さは$\log_{\frac{10}{9}} n$となり,n の範囲が$\log_{\frac{10}{9}} n \leq 24$となるので,⑧の時よりも大きい n のサイズは扱うことができない.そのかわり,非再帰部の計算量が$\mathrm{O}(2^{{}^{n}/_{10}})$となるため,⑧よりも少ない計算時間でハノイの塔を処理できる.

6.2　マルチスレッドアルゴリズムの性能の尺度を用いたハノイの塔の理論評価

次に,ハノイの塔の仕事量,スパンを評価した.

まずはじめに,⑤の漸化式によるハノイの塔の仕事量とスパン,そしてT_Pを表 3 に示す.

表 3.一般的なハノイの塔の仕事量とスパン,T_P

仕事量(T_1)	スパン(T_∞)	T_P
$\mathrm{O}(2^n)$	O(n)	$\mathrm{O}({}^{2^n}/_{p} + n)$

⑥の漸化式を逐次で行うと,H(n)を計算するときに 2 つの H(n - 1)の関数を生成・呼び出ししていたため,計算時間（仕事量）は指数関数的に増加していたが,⑥の漸化式を並列化させることで,2 つの関数は並列に処理することができ,その漸化式は

$$\mathrm{H(n)} = \mathrm{H(n-1)} + \mathrm{C}\big(\mathrm{C}\text{ は任意の定数}\big)\cdots ⑩$$

となる.よって⑩の漸化式の計算量（スパン）は n のサイズに比例して増加する.

次に,⑦の漸化式からハノイの塔の仕事量,スパン,そしてT_Pを求めた. $\mathrm{k} = \frac{n}{c}$(c はc > 1となる任意の整数)としたときの⑦の漸化式の仕事量,スパン,T_Pを表 4 に示す.

表 4.改良したハノイの塔の仕事量,スパン, T_P

仕事量	スパン	T_P
$\mathrm{O}(2^{\frac{n}{c}})$	$\mathrm{O}(2^{\frac{n}{c}})$	$\mathrm{O}(2^{\frac{n}{c}}/p)$

⑩の時と同じように,再帰関数の部分は並列に処理できるのでその時の漸化式は,

$$\mathrm{H(n)} = \mathrm{H(n-k)} + \mathrm{C}\cdot 2^{k}\big(\mathrm{C}\text{ は定数}\big)\cdots ⑪$$

となる. 表 3 と表 4 を比較すると,スパンは改良前の方が効率が良いが,T_Pは改良後の方が計算効率は良くなる.実際にはプロセッサの台数も考慮しないといけないので,実質的な計算効率は改良後のほうが良いことが分かる.また,仕事量の比較から,DP を利用しなくても⑦の式の方が計算効率は良いことが分かる.

表 4 を見ると,改良したハノイの塔の仕事量,スパンは漸近的には同じであることが分かる.この理由としては,逐次と並列でそれぞれ再帰部の計算は違うものの,非再帰部の計算量が再帰部の計算量を支配しているので,オーダー的には同じになる.

また、このアルゴリズムの再帰の段数は$\mathrm{O}(\log_{\frac{c}{c-1}} n)$となり、GPU を用いて実現する場合実用サイズまで計算可能である。

7. あとがき

本研究では GPU における DP を利用して、マルチスレッドアルゴリズムの効率的な実現に関して考察を行った。今後はそれぞれのアルゴリズムを実際に実現し、本論文の方法の妥当性を検証することが残されている。

謝辞

本研究の遂行にあたり,ご指導をいただいた和田幸一教授および研究室の方々に深く感謝いたします.また,大阪府立大学大学院の藤本典幸教授に心より厚く御礼申し上げます.

参考文献

[1]Tomas H. Cormen, et al.,"Introduction to Algorithms", MIT Press, 1990

[2]青木尊之,額田彰『はじめての CUDA プログラミング』工学社,2009 年

[3]Robert D. Blumofe, et al., "Cilk: An Efficient Multithreaded Runtime System"" PPOPP '95 Proc, pp. 207-216, 1995

[4]S. Lakshmivarahan, et al., "PARALLEL ALGORITHMS FOR SOLVING CERTAIN CLASSES OF LINEAR RECURRENCES" the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science proc., pp. 457-478, 1985

[5]Ronald L. Knuth et al., "Concrete Mathematics", Addison-Wesley, 1994