並列プログラム計算量の系統的機械証明手法の開発

白水駿，江本健斗

近年，処理データの巨大化やマルチコアプロセッサの普及に伴い，正しく効率の良い並列プログラムの需要が高まっている．一方，プログラムの正しさが定理証明支援系によって証明されるようになってきており，プログラムの効率（計算量）についてもその系統的証明手法が逐次プログラムを対象に研究されてきた．本研究は，既存の逐次プログラムを対象とした証明手法を拡張し，定理証明支援系 Coq を用いた並列プログラムの計算量の系統的証明手法を提案する．一般に，Coq におけるプログラムの性質の証明では，証明に関連した要素が最終的に抽出される本来の計算コードに影響を与えないよう注意深く証明を構築する面倒が伴う．既存研究はこの問題を拡張したモナドを用いることで解決を測った．本研究はその性質を継承しつつ,「2 つの計算を並列に行う」というプリミティブを提供することで並列計算量の証明を本来の計算の邪魔すること無く行える環境を構築する．

1 はじめに

現在の情報化社会ではプログラムは様々な場所で利用されるようになっている. これに伴いプログラムは正しく効率よく動作することが強く期待される. その期待に対し，プログラムの正しさが広く定理証明支援系などで機械的に証明されるようになってきた [2].

正しさだけでなく，その効率も証明できれば効率がよく正しいプログラムであることを示すことができる. 近年，プログラムの効率についてもその機械証明が注目されるようになり，逐次プログラムの計算量を系統的に機械証明する手法がいくつか提案されてきている [3–7].

最近，マルチコアプロセッサの普及や処理対象データの巨大化により並列処理の需要が高まっている. そのため，並列プログラムの計算量を系統的に機械証明する手法の開発が望まれている.

本研究は，先行研究 [5] の逐次プログラムの計算量証明の手法を発展させることによって，並列プログラムの計算量を証明する手法を開発することを目的とする.

本研究では，具体的にはに以下のことを行った．まず,「2 つの計算を並列に行う」プリミティブを追加した．また，このプリミティブは，複数プロセッサーが使えれば二つの計算を並列に行い，そうでないなら逐次で行うようにし，それに応じた計算量を算出できるようにした.

各節の構成について示す．2 節では定理証明支援系の Coq についておよび，先行研究の逐次プログラムの計算量証のためのライブラリについて，3 節では今回対象とする並列プログラムの計算量について，4 節では並列プログラム計算量証明のためのライブラリについて，5 節では並列プログラムの計算量の証明について，説明する.

2 準備

2.1 定理証明支援系 Coq

Coq [1] はフランス国立情報学自動制御研究所（INRIA）が開発した定理証明支援システムの一つである Coq は対話形式で，様々な論理的命題を機械的に証明することができる．例えば，プログラムの性質を証明することができる．リストの要素の順番を逆にする

Towards Systematic Proof of Parallel Running-time on Coq
Shun Shiromizu，Kento Emoto, 九州工業大学, Kyushu Institute of Technology.

```
1  Program Fixpoint pow2 (n:nat)
2  : {! res !:! nat !<! c !>!
3       pow2_result n c !} :=
4    match n with
5      | 0 =>
6        += 1;
7        <== 1
8      | S n' =>
9         a <- pow2 n';
10        b <- pow2 n';
11        += 1;
12        <== (a + b)
13   end.
```

図 2　逐次プログラムの計算量証明用 pow2 のコード

```
1  Definition pow2_result
2   (n:nat)  (c:nat) :=
3      c = (2^(n+1))-1.
```

図 3　pow2 の計算量に関する命題

関数 reverse であれば，reverse を二回適用したリストは二回適用する前のリストと等しいといったものである．

Coq で書かれたプログラムは Haskell や OCaml の通常の関数型言語に証明された性質を保ったまま抽出して実行できる．

2.2　逐次プログラムの計算量証明手法

先行研究 [5] は，逐次プログラムの計算量証明ができる Coq ライブラリを提供している．例として自然数 n を受け取り，自然数 2^n を返す関数 pow2 を考える．n=0 のとき 1 を返し，0 < n のとき pow2(n-1)+pow2(n-1) を返すことで 2^n を計算する．

まず，通常の Coq 関数として pow2 を記述すると次のようになる．

```
1  Program Fixpoint pow2 (n:nat):(nat):=
2    match n with
3    | 0 => 1
4    | S n' =>  let a = pow2 n'
5                   b = pow2 n'
6                   in a + b
7    end.
```

このコードをベースにして計算量証明のためのライブラリ [5] の書式に直したものが図 2 のコードである．このライブラリでは，計算量に関する部分を関数の結果の型の部分に追加しつつ，計算量の算出のための仕組みが動くように計算本体の書式を少しだけ変更する．

以下，このコードを用いて先行研究の計算量証明手法について説明する．

まず，関数の結果の型である”{!res !:! nat !<! c !>! pow2_result n res c !}”には次のような情報が記述される．”!res !:! nat !”の部分は計算の結果の変数が res で nat がその型が自然数型であることを表しており，”<! c !>”はそのときの計算量が c であることを示している．最後の”pow2_result n res c !” はこれら n,res,c の満たすべき性質を表す命題である．この命題は図 3 に示されており．この命題は計算量 c が関数の入力 n に対して $2^{n+1}-1$ であることを示している．実際には，図 2 コードを定義するためには図 3 の命題を証明する必要がある．

次に，関数本体部の書き直しについて説明する．基本的には，各部にかかる計算量を追記しつつ，計算の流れを”;”でつなげて表記する．例えば，n=0 の場合は計算は”+=1”と”<== 1”が”;”で繋がれたものである（6,7 行目）．前者の”+=1;”の部分は計算量（c）に 1 を加算することを意味する．この関数の場合は 1 である．この値は結果の 1 という値を生成するコストが 1 であるためである．”<== 1”の部分は純粋な計算結果 (res) として 1 を返すこと意味する．

同様に n > 0 の場合計算は，”a <- pow2 n’ ”, ”b <- pow2 n’ ”, ”+=1”, ”<== (a+b)”を続けて行う (9〜12 行目)．ここで”a <- pow2 n’ ”は pow2 n’ の計算結果を a に束縛することを意味する．一般にこのライブラリにおいて”x <- Y ; Z”という記述は Y の計算を実行し計算 Y の結果を x に束縛して Z の部分を実行するといった計算をする．この時，全体の計算量は Y の計算量に Z の部分の計算量を加えたものになる．つまり，関数 pow2 の 11 行目から 14 行目の部分の計算は，a に pow2(n’) の結果を，b に pow2(n’) の結果を代入し最終的に a + b を返す．計算量は pow2(n’)(a <- pow2 n’; の部分) の計算量

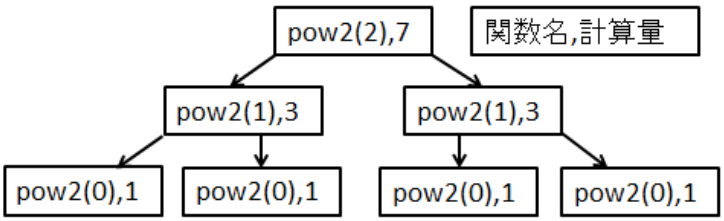

図 1　逐次の pow2(4) の計算量

```
1  Definition
2  C (A:Set) (P:A -> nat -> Prop) :
3   Set :=
4   {a : A | exists (an:nat), (P a an)}.
```

図 4　モナド C のコード

```
1  Definition
2  ret (A:Set) (P:A -> nat -> Prop)
3   (a:A) (Pa0:P a 0)
4  : C A P.
```

図 5　ret 関数のコード

に，pow2(n')(`b <- pow2 n'`; の部分) の計算量を加え，さらに 1(`+=1` の部分) を加えた値になる．

さらに，計算量証明に関する余計な部分を排除した Ocaml のコードが抽出できる．一般に，愚直に抽出してしまうと抽出後のコードに余計な処理が入ってしまう問題がある．

```
1  Definition
2  inc (A:Set) k (PA : A -> nat -> Prop)
3        (x:C A (fun x xn => forall xm,
4                xn + k = xm -> PA x xm))
5  : C A PA.
```

図 6　inc 関数のコード

2.3　逐次プログラムの計算量証明のためモナド構造

この節では前節で示した計算量算出等の動作が内部でどうなっているかを説明する．計算量の算出および計算結果の指定は計算結果の型を関数の型の中で操作することによって実現している．計算結果の型は図 4 のようなモナド構造を用いて表現されている．

”`C A P`”という型をもつ構造は，A 型の出力 a に対して計算量 an を述語 P で関連づけた集合である．つまり，計算結果と計算量との対応のデータ構造のようなものである．

先に示した関数の結果の型”`{!res !:! nat !<! c !>! pow2_result n c !}`”はこの C を用いた”`C nat (fun (res:nat) (c:nat) => pow2_result n c )`”の略記である．

ここで，P の第一引数が純粋な結果 (res)，第二引数が計算量 (c) に対応している．その他の書式も，この構造を操作するものである．

```
1  Definition
2    bind (A:Set) (PA:A -> nat -> Prop)
3      (B:Set) (PB:B -> nat -> Prop)
4      (am:C A PA)
5      (bf:forall (a:A)
6        (pa:exists an, PA a an),
7           C B (fun b bn => forall an,
8           PA a an -> PB b (an+bn)))
9  : C B PB.
```

図 7　bind 関数のコード

まず純粋な結果の指定をする”`<== x`”の操作は，図 5 の ret を用いて”`ret _ _ x _`”に対応する．”`_`”の引数は Coq が補完する． この ret は”`Pa0`”の型の部分で純粋な結果に対して計算量は 0 を対応づけている．

次に計算量を指定する”`+= k ; Z`”の操作である．

は図 6 の inc を用いて”inc _ k _ Z”に対応している．”Z”の部分は次に記述されている操作を表している．例えば，関数 pow2 の”+= 1; <== 1”では”<== 1”が Z である．

この inc は，3 行目と 4 行目の”x”の型の部分で”Z”の計算結果の計算量に”k”を加算して”PA”の引数としている．そして，この PA を C A PA としてモナドでまとめている．

最後に，二つの操作を繋げる ”x <- Y; Z”の操作である．図 7 の bind を用いて”bind _ _ _ _ Y (fun (x : _) (am : _) => Z)”に対応している．inc 同様”Z”の部分は後述の操作を表している．

”C A PA”は Y 部分の計算に対応している．その中の PA を用い，Y の (結果の値 a とその) 計算量 an が，Z の (結果の値 b とその) 計算量 bn と加算され，全体の計算量とされている (8 行目)．

このように型の内部に計算量に関する操作を記述することによって，証明したコードを OCaml に抽出する際に計算に関係のない証明用の部分が省かれ計算量の性質が保たれたコードを抽出できるようになる．

3 並列プログラムの計算量

並列プログラムの計算量のライブラリを設計するために，具体的な例を使いつつ並列プログラムの計算量の考え方を整理する．

フィボナッチ数を求める関数 fib について考える．関数 fib は n=0 の時 0 を返し，n=1 の時 1 を返し，n>1 の時は fib(n-1)+fib(n-2) を返す．この関数 fib は，fib(n-1) と fib(n-2) が独立な計算であるため，これらの二つの関数を並列に同時に呼び出すことができる．以降ではこのように二つの計算を並列に呼び出す (呼び出しは繰り返しても構わない) ことを考える．このような並列呼び出しを”a & b <-- X # Y;Z”の書式で考える．この書式は計算 X と計算 Y を並列に呼び出しそれぞれの結果を a と b で束縛し，Z の計算を行うものである．この書式で fib を記述したものが次のコードである．

```
1  Program Fixpoint fib (n:nat)
2  : {! res !:! nat !<! c !>! !;! pn !;!
3       fib_result n res pn c !} :=
4   match n with
5     | 0 =>
6      += 1;
7      <== 0
8     | S n' =>
9       match n' with
10      | 0 =>
11        += 1;
12        <== 1
13      | S n'' =>
14        a & b <-- (fib n'') # (fib n');
15        += 1;
16        <== (a + b)
17      end
18    end.
```

まず，無限にプロセッサがある場合の計算量を考える．並列呼び出し毎に計算をあたらしいプロセッサに割り振るので図 8 のような実行の様子になる．fib(n) の計算では fib(n-1) と fib(n-2) が並列に実行できるので，fib(n) を実行しているプロセッサは引き続き fib(n-1) を実行し，同時に，空いているプロセッサで fib(n-2) の計算の実行を依頼する．この時の計算量は，実行開始して最後のプロセッサが終了するまでの計算時間となる．例えば，図 8 では各関数の計算量が等しいとき，プロセッサー 1 の fib(1) が終了するまでの計算時間である．

並列度無限の時の計算量は，並列呼び出しした 2 つの部分関数の計算量の最大値を取り，呼び出し元の関数の計算量を追加することによって求まる．具体的には，図 8 の fib(4) に注目すると fib(3) の計算量 3 と fib(2) の計算量 2 の最大値 3 に fib(4) 自体の計算量 1 を加えた 4 が fib(4) 全体の計算量になる．

以上は並列度を無限とした議論であったが，現実ではプロセッサに限りがあるため並列度を有限であると考える必要がある．プロセッサが有限だった場合．計算へのプロセッサの割り当て方は多くのものが考えられるが，本研究では簡単のため，単純にプロセッサ集合を分割していく方法を考えることにする．図 9 は並列度を 2 としたときの fib（4）である．並列度が 2 なので，プロセッサ 1 で fib(4) が fib(3) 実行しつつ,fib(2) プロセッサ 2 に並列に実行することができる．しかし，これ以降は並列に呼び出すことができなず，それぞれが逐次的に実行を行う．このように空きプロセッサー数によって動作を分ける必要がある．

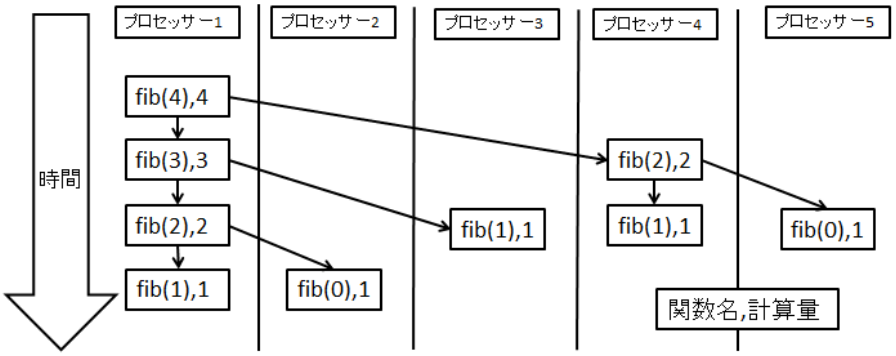

図 8　並列度無限の **fib(4)** の計算量　並列度無限大の環境で実行．**2** つの再帰のうち一つを新たなプロセッサで実行を依頼する．

```
1  Program Fixpoint pow2_par (n:nat)
2  : {! res !:! nat !<! c !>! !;!lbp !;!
3       pow2_par_result n lbp c !} :=
4    match n with
5      | 0 =>
6         += 1;
7        <== 1
8      | S n' =>
9       a & b <--
10           pow2_par n' # pow2_par n' ;
11     += 1; <== (a + b)
12 end.
```

図 10　並列プログラムの計算量証明用 **pow2_par** のコード

```
1  Fixpoint pow2_par_time
2   (n:nat) (lbp:nat) :=
3   match n with
4    | 0 =>
5        1
6    | S _ =>
7  if n <? lbp then n + 1
8  else (((2 ^ ( (n+1)-lbp)) - 1) + lbp)
9  end.
10
11 Definition pow_par_result
12  (n:nat) (lbp:nat) (c:nat) :=
13     c =  pow2_par_time n lbp.
```

図 11　**pow2_par_resurlt** と **pow2_g_time** のコード

この際の計算量は次のようになる．並列度に余裕 (空きプロセッサー) があれば並列呼び出しとし計算量は二つの計算の大きい方とする．並列度に余裕 (空きプロセッサー) がなければ逐次呼び出しとして，計算量は二つの関数の計算量の合計とする．

以降，簡単のため，並列度は 2 のべき乗であると考え，並列呼び出しをした際は，その二つ関数にプロセッサー数を半分ずつ割り当てるとする．例えば，プロセッサー数 8 で並列呼び出しをしたら場合，二つの関数はプロセッサー数 4 となる．

なお，プロセッサの再利用も考えられるが，それらは今後の課題とする．

4　並列計算量証明のためのライブラリ

2 節で述べた先行研究のライブラリ [1] を 3 節で述べた並列計算量の考えに基づき，並列プログラムの計算量を扱うように拡張した．本ライブラリでは並列度 p は 2 のべき乗であるとする．そして，ライブラリ内部でその 2 の対数 lbp を主に扱う．つまり, 並列度を p と lbp の関係は次の式となる．

$$p = 2^{lbp}$$

並列プログラムの計算量の証明のために加えた拡張は以下のとおりである．

- 並列度を表す変数 lbp のモナド C への追加.
- 並列呼び出しを指定する書式の追加.
- 並列度に応じた計算量算出の追加.

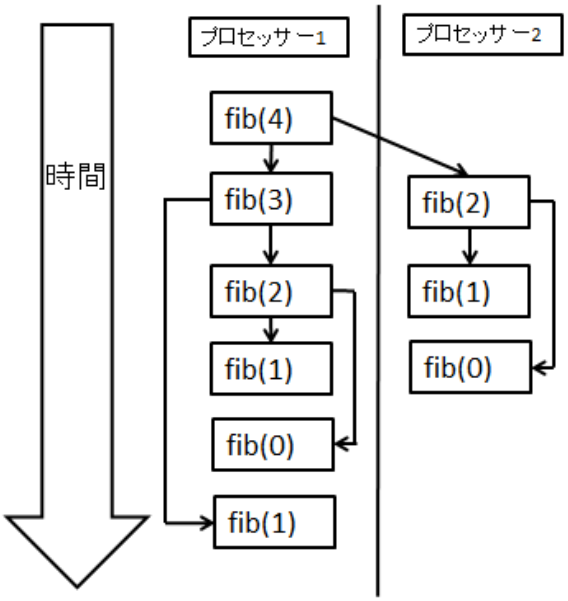

図 9　並列度 2 の fib(4) の計算量　fib(3)，fib(2) 以降はプロセッサに空きがないので逐次的に動く．

関数 pow2 の並列版 pow2_par を図 10 に示す．pow2 の ”a <- pow2 n’;” と ”b <- pow2 n’;” の部分は並列に呼び出すことができるため，それらを新たな書式 a & b <-- pow2_par n’ # pow2_par n’ ; で並列実行するように記述している．

関数 pow2_par の計算量 c は，lbp 回までの並列呼び出しではプロセッサ数に余裕があるがそれ以降は逐次で動くため，入力 n と lbp を用いて次の式で表すことができる．

$$c = \begin{cases} 1 & (n = 0) \\ n + 1 & (0 < n \wedge n < lbp) \\ 2^{n+1-lbp} - 1 + lbp & (\text{otherwise}) \end{cases}$$

この計算量の性質を Coq で書いたものが図 11 の pow2_par_resurlt と pow2_par_time である．この pow2_par_result が関数 pow2_par に対して成り立つことを証明することによって計算量の証明となる．

以降で拡張の内容を詳しく説明する．

4.1　モナド C への並列度に関する情報の追加

並列呼び出しの際，並列度によって算出する計算量を変化させるために利用できる並列度に関する情報

```
1  Definition
2   C
3   (A:Set) (P:A -> nat -> nat -> Prop)
4   Set :=
5   {a : A | forall (lbp:nat),
6     exists (an:nat), (P a lbp an)}.
```

図 12　並列拡張後の C のコード

```
1  Definition inc
2   (A:Set)
3   k
4   (PA : A -> nat -> nat -> Prop)
5   (x:C A (fun x lbp xn =>
6      forall xm, xn + k = xm ->
7      PA x lbp xm))
8   : C A PA.
```

図 13　並列拡張後の inc のコード

を変数として保持する必要がある．そのため，lbp を C に追加した．

図 12 は拡張を加え lbp を追加したモナド C である．このように，入力と計算量の関係を表す述語 P の引数に lbp を追加した．ただし，lbp は forall で限

```
1  Definition ret
2   (A:Set)
3   (P:A -> nat -> nat -> Prop)
4   (a:A)
5   (Pa0: forall lbp, P a  lbp 0)
6   : C A P.
```

図 14　並列拡張後の ret のコード

定する．これにより，並列度の具体値を限定せずに各操作を定義することができる．また，図 11 の関数 pow2_par_result ように，計算量の性質の中に lbp を加えることができる．

inc 関数および，ret 関数にも lbp の追加に対応した拡張を加えた．まず，図 13 が拡張後の inc 関数のソースコードである．逐次の手法との違いは，4 行目 PA の型の部分と 5 行目からの a の型の部分に lbp を追加している．実行前と実行後の lbp の値に変化はなく動作は逐次の手法と変わらない．次に，図 14 が拡張後の ret 関数のソースコードである．逐次の手法との違いは，3 行目 P の型の部分と 4 行目からの Pa0 の型の部分に lbp を追加している．実行前と実行後の lbp の値に変化はなく動作は逐次の手法と変わらない．

4.2　並列呼び出し操作の追加

並列呼び出しを行うために bind 操作を拡張した parbind を追加した．ユーザープログラム上の書式は”`a & b <-- X # Y ; Z`”とする．これは計算 X と計算 Y を並列呼び出しを行い，それぞれの結果を a,b に入れ，Z の計算を行うというものである．

図 15 は parbind 関数のソースコードである．bind と違い，並列に動かす可能性のある X と Y とそれに続く Z の三つの計算を繋ぐことになる．6 行目からの zf の内部で計算Xと計算の結果から Z の計算を行い，計算量の指定や結果の指定をしている．計算 X の計算量が xn，並列度の対数 lbp が lbx，計算 Y の計算量が yn，並列度の対数 lbp が lby，計算 Z の計算量が zn，並列度の対数 lbp が lbz である．

9 行目と 10 行目は lbz が 0, つまり使えるプロセッサが一つの時の動作を示している．この時は逐次呼び出しと同等の動作をする．lbx と lby が 0 で計算量は xn と yn と zn の合計である．

11 行目と 12 行目は lbz が 0 より大きいとき, つまり並列度に余裕があるときである．この時は並列呼び出しをする．lbx と lby は lbz より 1 少ない値である．これは，lbp が並列度の 2 の対数であるため，X と Y の計算に全体で使えるプロセッサ集合の半分ずつ割り当てることに対応する．計算量は xn と yn の最大値に zn を加えた値になる．つまり，より計算に時間がかかった方が全体の計算時間を支配することに対応する．

5　計算量の証明と抽出

計算量の証明の例として pow2_par の計算量の証明を説明する．付録 A に証明全体を示した．pow2_par の計算量の証明は関数 pow2_par が図 11 の pow2_par_resurt を満たすこと証明する．Coq が obligation として証明を要求してくるのは以下の 2 点である．

- n=0 の時に pow2_par_result が成り立つ．
- それ以外の時に pow2_par_result が成り立つ．

二つ目について詳しく説明する．pow2_par(S m) が pow2_par(m) を二つ呼び出す場合，二つの pow2_par(m) が pow2_par_result を満たし，pow2_par(S m) と二つの pow2_par(m) の計算量と並列度の関係が成り立つ前提で pow2_par_result が成り立つことを証明することになる．また，並列呼び出しは lbp が 0 である場合とそうでない場合で分岐するため，lbp が 0 である場合と，そうでない場合両方を示す．この証明によって，関数 pow2_par が計算量が満たすべき pow2_par_resurt を満たすこと証明できる．

最後に，先行研究同様，証明したコードを計算量に影響を及ぼさずにソースコードを抽出することができた．図 16 は pow2_par の抽出後のソースコードである．このコードの par は並列計算を行うためのプリミティブであり，抽出後の言語で実装されているとする．その操作は，上で述べたように並列度に余裕があるならば二つの計算を並列実行し，並列度を二つのプロセッサーに半分ずつに分ける，並列度に余裕がな

```
Definition parbind
(X:Set) (PX:X -> nat -> nat -> Prop)
 (Y:Set) (PY:Y -> nat -> nat -> Prop)
 (Z:Set) (PZ:Z -> nat -> nat -> Prop)
 (xm:C X PX) (ym:C Y PY)
 (zf:forall (x:X)  (px:forall lbx, exists xn, PX x lbx xn)
              (y:Y)  (py:forall lby ,exists yn, PY y lby yn),
  C Z ((fun z lbz zn =>
    (lbz = 0 ->(forall lbx lby xn yn,  (lbx = 0 /\ lby = 0) ->
  PX x 0 xn /\ PY y 0 yn -> PZ z 0 ((xn+yn)+zn)))/\
     (0 < lbz ->(forall lbx lby xn yn, (lbx + 1 = lbz /\ lby + 1 = lbz ) ->
            PX x lbx xn /\ PY y lby yn -> PZ z lbz ((max xn yn)+zn))))))
: C Z PZ.
```

図 15　parbind 関数のコード

```
let rec pow2_par n =
  (fun f0 fS n -> if (eq_big_int n zero_big_int)
    then f0 () else fS (pred_big_int n))
    (fun _ -> succ_big_int
    zero_big_int)
    (fun n' ->
    let a,b = par (pow2_par n') (pow2_par n') in add a b)
    n
```

図 16　pow2_par の抽出後のコード

ければ二つの計算を逐次実行するものである．

6　まとめ

本研究では逐次プログラムの計算量証明の手法をベースに並列プログラムの計算量の証明手法を開発した．新たに，並列度によって計算量を分岐させるため，並列度の要素を加え，また，並列呼び出しのための書式を追加して，並列度に応じた計算量を算出できるように拡張した．この拡張によって並列プログラムの計算量を算出し，計算量の性質を証明できるようになった．

今後の課題として以下を考えている．今回の拡張はプロセッサの再利用を考慮しない．実用上は，ふたつに分岐した計算の一方が早期に終了した場合，そちらに割り当てていたプロセッサを他方に追加して割り当てることで高速化が望めるかもしれない．そのようなプロセッサの再利用ができる計算量の算出への対応が今後の課題である．また，証明の際に頻出する部分が存在する．この部分は複数のタクティクが必要がなため，それを一つにまとめることによって，証明を簡単にすることができる．今回 pow2_par の証明を行ったが，並列度が無限大である場合にマージソートの証明を行っている．マージソート関数も本研究のライブラリで証明が可能であると想定できる．しかし，pow2_par で証明した図 11 のような計算量が満たす性質が何であるかが考えずらい問題点がある．また，マージソート関数は，並列呼び出しする関数が他の関数であり，計算量が入力サイズ，並列度によって定まらないため可変である．このような場合オーダーを求める必要がある．先行研究ではオーダーの証明対応しているが，この点の並列拡張ができていないのでこの点も今後の課題である．

謝辞 本研究は JSPS 科研費 JP15K15974 の助成を受けたものです．

参 考 文 献

[1] Bertot, Y. and Castéran, P.: *Interactive Theorem Proving and Program Development - Coq'Art:*

The Calculus of Inductive Constructions, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

[2] Emoto, K., Loulergue, F., and Tesson, J.: A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014, pp. 258–274.

[3] Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., and Tesson, J.: Systematic Development of Correct Bulk Synchronous Parallel Programs, *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, IEEE, 2010, pp. 334–340.

[4] Loulergue, F., Bousdira, W., and Tesson, J.: Calculating Parallel Programs in Coq Using List Homomorphisms, *International Journal of Parallel Programming*, Vol. 45, No. 2(2017), pp. 300–319.

[5] McCarthy, J. A., Fetscher, B., New, M. S., Feltey, D., and Findler, R. B.: A Coq Library for Internal Verification of Running-Times, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, 2016, pp. 144–162.

[6] Mu, S.-C., Ko, H.-S., and Jansson, P.: Algebra of Programming Using Dependent Types, *Mathematics of Program Construction*, Audebaud, P. and Paulin-Mohring, C.(eds.), LNCS, Vol. 5133, Springer Berlin / Heidelberg, 2008, pp. 268–283.

[7] Tesson, J., Hashimoto, H., Hu, Z., Loulergue, F., and Takeichi, M.: Program Calculation in Coq, *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST2010)*, LNCS 6486, Springer, 2010, pp. 163–179.

A pow2par の計算量の証明

図 17 は pow2_par の証明のソースコードである．また，図 18 は pow2_par の証明に用いた補題のソースコードである．

```
Fixpoint pow2_par_time (n:nat) (lbp:nat) :=
 match n with
  | 0 => 1
  | S _ =>if n <? lbp then n + 1 else (((2 ^ ( (n+1)-lbp)) - 1) + lbp)
end.

Definition pow2_par_result (n:nat) (lbp:nat) (c:nat) := c = pow2_par_time n lbp.

Program Fixpoint pow2_par (n:nat) :
  {! res !:! nat !<! c !>! !;! lbp !;! pow2_par_result n lbp c !} :=
  match n with
    | 0 => += 1; <== 1
    | S n' => a & b <-- pow2_par n' # pow2_par n' ; += 1; <== (a + b)
   end.
Next Obligation.
Proof.
  unfold pow2_par_result, pow2_par_time; reflexivity.
Qed.
Next Obligation.
Proof.
  intros; unfold pow2_par_result; intros; remember n' as m.
  induction lbp as [ |lbp].
  - split.
    -- intros; destruct H1; rewrite H1,H2, (pow2_par_time_s m 0); reflexivity.
    -- intros; apply False_ind, (lt_n_0 0 H).
  - split.
    -- intros; discriminate.
    -- intros; destruct H1; subst.
       destruct H0; rewrite Nat.add_1_r in H0,H1.
       apply (eq_add_S lbx lbp) in H0; apply (eq_add_S lby lbp) in H1; subst.
       rewrite (Max.max_idempotent).
       induction n' as [ | m].
       --- inversion H.
           ---- simpl; omega.
           ---- unfold pow2_par_time, "<?".
                rewrite (leb_correct 2 (S lbp) ); omega.
       --- unfold pow2_par_time, "<?".
           case_eq (S (S m) <=? lbp).
           ---- intros; apply (leb_complete (S (S m)) lbp) in H0.
                rewrite (leb_correct  (S( S(S m))) (S lbp) ); try omega.
           ---- intros; apply (leb_iff_conv  lbp (S(S m)) ) in H0.
                rewrite (leb_correct_conv (S lbp) (S (S (S m))) ); try omega.
                rewrite(Nat.add_1_r (S (S m))),(Nat.sub_succ ((S (S m))) (lbp)).
                rewrite (Nat.add_1_r  (S m)); omega.
Qed.
```

図 17　pow2_par 計算量証明コード

```
Lemma move1: forall x y: nat, 0 < x -> 0 < y -> x - 1 + (y - 1) + 1 = x + y - 1.
Proof.
  intros x y Hx Hy.
  rewrite (Nat.sub_1_r y).
  rewrite (Nat.sub_1_r x).
  specialize (S_pred x 0 Hx); intro Hx2.
  rewrite Hx2 at 2.
  rewrite plus_Sn_m.
  assert (1 <= pred x + y) as H1xy.
  { rewrite <- (Nat.succ_pred_pos y Hy).
    rewrite <- plus_n_Sm.
    apply le_n_S.
    apply le_0_n. }
  rewrite <- (minus_Sn_m); [auto | auto].
  rewrite <- (Nat.add_1_r (pred x + y - 1)).
  rewrite (Nat.sub_1_r).
  assert (y <> 0) as Hy0.
  { rewrite Nat.neq_0_lt_0.
    apply Hy. }
  rewrite <- Nat.add_pred_r; [reflexivity | auto].
Qed.
Lemma le_0_pow2: forall x: nat, 0 < 2 ^ x.
Proof.
  induction x as [ | x].
  - unfold pow.
    apply lt_0_Sn.
  - unfold pow.
    fold pow.
    apply Nat.mul_pos_pos;auto.
Qed.

Lemma pow2_par_time_s :
  forall n lbp : nat, lbp = 0
    -> pow2_par_time n lbp + pow2_par_time n lbp + 1 = pow2_par_time (S n) lbp.
Proof.
  intros; induction n.
  - subst; unfold pow2_par_time, "<?".
    rewrite (leb_correct_conv 0 2 ); simpl; auto.
  - unfold pow2_par_time; unfold "<?".
    rewrite (leb_correct_conv lbp ( S(S n)) ).
    rewrite (leb_correct_conv lbp (S (S (S n)))).
    subst; simpl.
    rewrite (Nat.add_0_r (2 ^ (n+1))).
    rewrite (Nat.add_0_r (2 ^ (n+1) + 2 ^ (n+1) - 1)).
    rewrite (Nat.add_0_r (2 ^ (n+1) + 2 ^ (n+1))).
    rewrite (Nat.add_0_r (2 ^ (n+1) + 2 ^ (n+1) + (2 ^ (n+1) + 2 ^ (n+1)) - 1)).
    rewrite (move1 (2 ^ (n+1) + 2 ^ (n+1))(2 ^ (n+1) + 2 ^ (n+1))).
    reflexivity.
    apply (Nat.add_pos_l ( 2 ^ (n+1)) ( 2 ^ (n+1)) ), (le_0_pow2 (n+1)).
    apply (Nat.add_pos_l ( 2 ^ (n+1)) ( 2 ^ (n+1)) ), (le_0_pow2 (n+1)).
    subst; apply (Nat.lt_0_succ (S (S n))).
    subst; apply (Nat.lt_0_succ ( S n)).
Qed.
```

図 **18**　**pow2_par** 計算量証明の補題のコード