並列計算量の形式的証明を伴う BSP プログラム用 Coq ライブラリ

田中 匠海[1,a)]　江本 健斗[1]

概要：一般に，プログラムは，正しくかつ効率的に動作することが望まれる．特に，複数の計算機を同時に用いて計算時間の短縮を図る並列プログラムに対しては，その効率，すなわち，計算量が想定どおりであることが強く期待される．近年，プログラムの性質に関する形式的な保証を与える手法として，Coq 等の定理証明支援系の利用が広く認知されてきた．しかしながら，そこではプログラムの正しさが主な焦点となっており，並列プログラムの計算量についての保証は考えられていない．本研究は，BSP モデルに従う並列プログラムを対象に，計算量についての形式的証明を伴う並列プログラムを構築するための Coq ライブラリを提案する．

キーワード：並列プログラミング，定理証明支援系，Coq，形式化，BSP，計算量

A Coq Library for BSP Programs with Certified Parallel Complexity

Tanaka Takumi[1,a)]　Emoto Kento[1]

Abstract: In general, a program is desired to be correct and run efficently. Especially, a parallel program, which uses multiple computers to reduce the computation time, is strongly expected to be efficient, i.e., to have the expected parallel time complexity. Recently, proof assistants such as Coq have become widely used to certify that a developed program satisfies its expected properties. However, this mainly focuses on the correctness and not on the parallel complexity of a parallel program. In this research, targeting on BSP model, we propose a Coq library to support building a parallel program with its certified parallel complexity.

Keywords: Parallel Programming, Proof Assistant, Coq, Formalization, Complexity

1. はじめに

一般に，プログラムは，正しくかつ効率的に動作することが望まれる．特に，複数の計算機を同時に用いて計算時間の短縮を図る並列プログラムに対しては，その効率，すなわち，計算量が想定どおりであることが強く期待される．

近年，プログラムの性質に関する形式的な保証を与える手法として，Coq [1] や Agda [3] 等の定理証明支援系の利用が広く認知されてきた．この手法では，定理証明支援系の提供するプログラミング言語で対象プログラムやその性質を記述し，その性質の証明を定理証明支援系の確認のもとで対話的に行っていく．証明が完了した際には，その証明が見落としなく正しいものであると保証される（人手での証明では見落としがないことの保証はない）．

しかしながら，既存の定理証明支援系を用いた手法では，プログラムの正しさが主な焦点となっており，並列プログラムの計算量についての保証は考えられていない．

本研究は，BSP モデル [8] に従う並列プログラムを対象に，計算量についての形式的証明を伴う並列プログラムを構築するための Coq ライブラリを提案する．具体的には，BSP モデルに従う並列プログラムを記述するためのプリミティブを与えるライブラリ BSML [5] に対し，その既存の Coq での形式化 SyDPaCC [2], [4] をベースとして，

[1] 九州工業大学
Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[a)] tanaka@pl.ai.kyutech.ac.jp

そこに既存の逐次計算量に関する証明支援の Coq ライブラリ [7] を利用した計算量記述を追加する形で，計算量証明を伴う BSML プリミティブの形式化を提案する．また，提案ライブラリの評価として，実用される並列計算である「接頭和計算」の並列プログラムを実際に記述し，その計算量証明が出来ることを確認した．

以降の本論文の構成は以下のとおりである．まず，第 2 節にて，既存の事柄についての導入を行う．次に，第 3 節にて，本研究の提案する BSML プリミティブの形式化を示す．そして，第 4 節にて，「接頭和計算」の並列プログラムを用いた実際の記述を通して提案手法を評価する．最後に，第 5 節にて，本論文をまとめる．

2. 準備

本節では，本論文で用いる既存結果について導入する．

2.1 定理証明支援系 Coq

定理証明支援系 Coq [1] は，依存型とカリー・ハワード同型対応に基づき，形式的証明の正しさを保証するシステムである．ユーザは，Coq の提供する関数型言語 Gallina を用いて，プログラムを含む数学的オブジェクトやその性質に関する定理（命題）を記述する．そして，Coq の型システムによる確認のもとで，ユーザはその定理の証明をタクティックというコマンドを用いて対話的に進めていく．証明が完了すれば，その証明が見落としなく正しいことが保証される．

例として，自然数（nat）を受け取って 2 を掛けて返す関数 dbl を考える．次のコードは，dbl の定義とその性質「任意の自然数 n に対して，dbl n [*1] は n + n に等しい」の証明を Coq で記述したものである：

```
1 Definition dbl (n : nat) := 2 * n.
2 Lemma dbl_n_is_n_plus_n:
3   ∀n : nat, dbl n = n + n.
4 Proof.
5   intros. unfold dbl. cbn.
6   rewrite plus_0_r. reflexivity.
7 Qed.
```

コードの 1 行目で関数 dbl を定義している．関数名 dbl の後ろは引数 n とその型 nat であり，:= の後ろの 2 * n が関数本体である．続く 2–3 行目に，上記の dbl の性質が補題として記述されている．2 行目の : の直前が補題名であり，3 行目は性質の命題である．それ以降の 4–7 行目が，Coq との対話によって完了した証明である．詳細は割愛するが，intros などのタクティックが . で区切られて並んでいる．証明の途中では，既存の定理を利用することも出来る（6 行目で定理 plus_0_r の与える等式 ∀n : nat, n + 0 = n による書き換えをしている）．

[*1] 関数型言語のため，関数の引数に括弧をつけないことに注意する．

2.2 逐次プログラムの計算量証明支援ライブラリ

逐次プログラムの計算量に関しては，その証明支援を行う Coq ライブラリが提案されている [7]．その基本アイデアは，「戻り値に計算量の証明を付ける」というものである．

例として，次のリストの和を求める関数 sum を考える：

```
1 Fixpoint sum(x : list nat) : nat
2   := match x with
3     | nil ⇒ 0
4     | cons a x' ⇒ a + sum x'
5     end.
```

関数本体（2–5 行目）は，match 構文により入力リスト x の場合分けを行い，空リスト（nil）なら 0 を返し，そうでなく先頭要素 a と残りの部分 x' に分けられる（cons a x'）なら，残りの部分の和 sum x'に先頭要素を足したものを返している．なお，1 行目の末尾の : nat は，この関数の戻り値の型が nat であることを明示したものである．

この sum に計算量記述を追加したのが次の sum' である：

```
1  Program Fixpoint sum'(x : list nat) :
2    {! ret !:! nat !<! c !>! c <= 1 + 2 *
       length x !}
3    := match x with
4      | nil ⇒ += 1;
5               <== 0
6      | cons a x' ⇒ s <- sum' x';
7                     += 2;
8                     <== a + s
9      end.
10 Next Obligation. lia. Defined.
```

まず，2 行目において関数の戻り値の型が nat から複雑な型 {! ret !:! nat !<! c !>! c <= 1 + 2 * length x !} に書き換えられている．これは Coq の記法拡張を用いた記述であるが，意味は「この関数の戻り値 ret は nat 型であり，その計算にかかる計算量 c は c <= 1 + 2 * length x を満たす」である．すなわち，入力リストに比例する計算量がかかることを記述したものである．この型の書き換え以外には，match での場合分けの各々における結果部分（⇒ の右）が書き換わっている．この部分では，+= n;が計算量が n 追加されることを，v <- exp;が計算 exp の結果を変数 v に束縛することを，<== e; が純粋な結果 e を返すことを意味している．これらも記法拡張を用いた記述であるが，その背景にモナド構造があることだけを述べて詳細は割愛する．そして，最後の行にある Next Obligation 以下において，今定義しようとしている関数 sum' の計算量 c が c <= 1 + 2 * length x を満たすことの証明を行っている（この証明が必要なため，1 行目の冒頭に Program のキーワードがある）．今回の証明は一次式の操作で済むため，一次式に関する証明を強力に自動化する lia タクティックで証明を終えている．

2.3 BSP モデル

BSP モデルは Valiant によって提唱された並列計算モデルである [8]．このモデルでは，各プロセッサが自身のローカルのメモリを持ち，それらプロセッサがネットワークを介して接続されるというハードウェア構成を仮定する．また，このモデルでの計算は，「スーパーステップ」の繰り返しとして構成される．各スーパーステップでは，全プロセッサが並列にローカル計算と他プロセッサへのデータ送信を行った後，全プロセッサでバリア同期を行う．その際，ローカル計算では，各プロセッサは，自身のローカルメモリ上のデータと直前のスーパーステップで自身に向けて送信されたデータのみを用いて計算を行う．また，送信されたデータはバリア同期により受信先に配送される．すなわち，送信データは次のスーパーステップにならないと利用できない．この制限により，デッドロックの問題を回避する．BSP モデルは幅広い並列計算を表現できる．例えば，Google の提唱した大規模グラフ並列計算モデルである Pregel [6] も BSP モデルに従ったものである．

BSP モデルの特徴のひとつは，その簡潔な計算量モデルにある．BSP モデルは計算量に関する三つのパラメータ，プロセッサ数 p，バリア同期時間 L，単位メッセージあたりの転送時間 g を持つ．これらを用いることで，BSP モデルに従う並列プログラム（以下，BSP プログラム）の計算量は次式となる：

$$\sum_{i=1}^{N}\left[L+\max_{0\le j<p} w_i^{(j)}+g\max_{0\le j<p} h_i^{(j)}\right]$$

ここで，N は計算に要したスーパーステップの数，$w_i^{(j)}$ はスーパーステップ i におけるプロセッサ j の計算時間，$h_i^{(j)}$ は同送受信メッセージ総量である．この式から分かることは，「BSP プログラムは，各プロセッサにおける逐次計算量を知ることができれば全体の並列計算量を表現可能である」ということである．本研究はこの特徴を活用する．

2.4 BSML と SyDPaCC

BSML [5] は，BSP プログラムを記述するための OCaml ライブラリであり，以下の五つのプリミティブを提供する．

$$\begin{aligned}
&\mathsf{mkpar}\ h && = \langle h\ 0,\ldots,h\ (p-1)\rangle\\
&\mathsf{apply}\ \langle f_0,\ldots,f_{p-1}\rangle\\
&\qquad \langle v_0,\ldots,v_{p-1}\rangle && = \langle f_0\ v_0,\ldots,f_{p-1}\ v_{p-1}\rangle\\
&\mathsf{put}\ \langle f_0,\ldots,f_{p-1}\rangle && = \langle g_0,\ldots,g_{p-1}\rangle\\
&\qquad && \quad \text{where } g_i\ j = f_j\ i\\
&\mathsf{get}\ \langle v_0,\ldots,v_{p-1}\rangle\ i && = v_i\\
&\mathsf{proj}\ \langle v_0,\ldots,v_{p-1}\rangle && = g \text{ where } g\ i = v_i
\end{aligned}$$

ここで，$\langle x_0,\ldots,x_{p-1}\rangle$ は「プロセッサ i が自身のローカルメモリに x_i を保持している」ということを意味するベクトルである（以下，並列ベクトルと呼ぶ）．プリミティブ mkpar は，関数 h を受け取り，それぞれのプロセッサ上で h に自身のプロセッサ番号 i を与えて計算した結果 $h\ i$ の並列ベクトルを作る．プリミティブ apply は，関数の並列ベクトルと値の並列ベクトルを受け取り，各プロセッサ上で自身の持つ関数 f_i を自身の持つ値 v_i に適用し，計算結果 $f_i\ v_i$ の並列ベクトルを作る．これらふたつは，BSP モデルにおけるローカル計算に対応する．プリミティブ put は，通信内容を表現する関数の並列ベクトルを受け取り，通信を実行する．この put が受け取る並列ベクトルの i 番目の要素，すなわち，i 番目のプロセッサのもつ関数 f_i は，i 番目のプロセッサが送信したいデータの情報を保持した関数であり，「プロセッサ番号 j を受け取ると，j 番目のプロセッサに送るデータを返す」という関数である．そして，put の返す並列ベクトルの関数 g_i は，「j を渡すと，j 番目のプロセッサから自身（i 番目のプロセッサ）に送信されたデータを返す」という関数である．プリミティブ get は，並列ベクトルの指定されたプロセッサ番号 i の値を（マスタープロセスに）取り出す．同様に，プリミティブ proj は，並列ベクトルの全情報を保持した関数を取り出す．これら三つのプリミティブは通信を伴うため，スーパーステップを消費することになる．

BSML のプリミティブは SyDPaCC [4] として Coq 上に形式化されている．具体的には，以下の様なモジュールとして，それぞれのプリミティブの型と満たすべき仕様とが記述されている．

```
1  Module Type BSP_PARAMETERS.
2   Parameter p : N.
3   Axiom p_spec : 0 < p.
4  End BSP_PARAMETERS.
5
6  Module Type BSML.
7   Declare Module Bsp : BSP_PARAMETERS.
8   Notation pid :={ n:N | N.ltb n Bsp.p = true}.
9   Parameter par : Type →Type.
10
11  Parameter get:
12    ∀{A: Type}, par A →pid →A.
13  Axiom par_eq : ∀{A:Type} (v v': par A),
14    (∀ (i: pid), get v i = get v' i) →
15     v = v'.
16  Parameter mkpar:
17    ∀{A:Type} (f:pid →A), par A.
18  Axiom mkpar_spec:
19    ∀(A:Type)(f:pid→A)(i:pid),
20      get (mkpar f) i = f i.
21  (* others are omitted for spece limitation *)
22 End BSML.
```

まず，BSP モデルのパラメータ p と，その仕様 $0 < p$ がモジュール BSP_PARAMETERS に定義されている（1–4 行目）．そして，BSML プリミティブの形式化として，モジュー

ル BSML が定義されている（6 行目以降）．このモジュールは，プロセッサ番号の型を pid と書けるように記述を定義（8 行目）し，並列ベクトルを表現する型 par を定義（9 行目）した後，プリミティブの型とその仕様を記述している．例えば，get であれば，上に示した定義に従いその型が ∀{A: Type}, par A →pid →A とされている（11–12 行目）．すなわち，「(暗黙の引数として要素の型 Aと）並列ベクトルとプロセッサ番号を受け取り，その並列ベクトルの要素を返す」という型である．また，その仕様は，「並列ベクトル v と v' が，任意のプロセッサ番号 i に対して get $v\ i =$ get $v'\ i$ となるならば，v と v' は等しい」というものである（13–15 行目）．同様に，mkpar についても型の指定（16–17 行目）と，その仕様（18–20 行目）が記述されている．紙面の都合で他のプリミティブについては割愛するが，以上と同様である．

これらのモジュールを使って Coq 上に BSP プログラムを記述すれば，Coq による支援のもとでそのプログラムの正しさを証明できる．また，証明が完了したコードから正しさを保ったまま OCaml コードを抽出し，BSML/OCaml の並列プログラムとして動作させることができる [2]．これらにより，正しさの保証された，実用可能な BSP プログラムを作ることが出来る．

3. BSML プリミティブの計算量を伴う形式化

前節で導入した SyDPaCC での BSML の形式化に対し，計算量記述の拡張を行う．具体的には，計算量記述に必要なパラメータの追加と，いくつかの補助関数等の追加と，プリミティブの型・仕様の書き換えを行う．

3.1 BSP パラメータの形式化の追加

SyDPaCC は計算量に関連する BSP パラメータを省いている．そのため，本研究では次のようにパラメータ L と g を追加する．いずれも，適当な単位時間を考えてその整数倍であるとし，nat 型の正の値であるとする：

```
1  Module Type BSP_PARAMETERS.
2    Parameter p : nat.
3    Axiom p_spec : 0 < p.
4
5    Parameter g : nat. (* added *)
6    Axiom g_spec : 0 < g.
7
8    Parameter L : nat. (* added *)
9    Axiom L_spec : 0 < L.
10 End BSP_PARAMETERS.
```

3.2 補助関数等の追加

計算量に関する記述を行う際に必要となる補助的な関数等をモジュール BSML に追加する．

まず，通信に関する計算量を記述する際に必要となる「通信されるもののサイズ」を取得するための仕組みを導入する．これには，次の型クラス sized を用いた：

```
1 Class sized(A : Set) := {size : A →nat}.
```

このクラスのインスタンスは，A 型の要素のサイズを計算する関数 size を提供する．例えば，自然数 nat を実際の動作時に 32 ビット整数で代用するといった場合には，次にようなインスタンスをユーザが定義することになる：

```
1 Instance nat_sized :
2   Bsml.sized (nat) := {size n := 4}.
```

また，簡便のため，並列ベクトルの要素になり得る型 A については，必ず sized のインスタンスが与えられるということを仕様として追加する：

```
1 Axiom par_sized : ∀{A:Set}, par A →sized A.
```

BSP の計算量では，全プロセッサの持つ値の最大を取る操作が現れるため，デフォルト値付きの最大値計算関数 maximum を次のように定義する：

```
1 Fixpoint maximum (e:nat) (l:list nat) : nat :=
2   match l with
3   | nil ⇒ e
4   | (x::xs) ⇒ max x (maximum e xs)
5   end.
```

同様に，リストの和を取る sum と二つのリストの対応する要素同士をペアとしたリストを返す zip を定義する：

```
1 Definition sum (x : list nat) :=
2   fold_right plus 0 x.
3 Fixpoint zip {A B : Type}
4  (xs : list A) (ys : list B) : list (A*B) :=
5   match xs, ys with
6   | x::xs', y::ys' ⇒ (x,y)::zip xs' ys'
7   | _,_ ⇒ nil
8   end.
```

本研究の形式化では，多くの関数が証明付きの値を戻り値とする．しばしばそれらから純粋な値だけを取り出したい場面が生じるため，そのための記法 valOf を用意する：

```
1 Notation valOf := proj1_sig.
```

同様に，しばしばプロセサ番号 pid を nat として扱いたいことがあるため，その変換関数を追加する（pid は「p 未満である」という証明の付いた nat である）：

```
1 Definition pid2nat (p:pid): nat := valOf p.
```

さらに，プロセッサ番号を並べたリスト $[0,\ldots,p-1]$ もしばしば必要となるため，その定義を pid_list として与える．具体的には，「q 以下の自然数 n に対して $[0,\ldots,n-1]$ を生成する」という関数 list_gen を介して定義する：

```
Program Fixpoint list_gen (n : nat) (q : nat)
  (x : list ({n : nat | n < q})) (H : n <= q)
    : list ({n : nat | n < q}) :=
  match n with
  | 0 ⇒ x
  | S n' ⇒ list_gen n' q (exist _ n' _::x) _
  end.
Next Obligation.
  apply le_Sn_le. assumption.
Defined.
Program Definition pid_list : list pid :=
    list_gen Bsp.p Bsp.p nil _.
```

また，pid_list のサイズに関する以下の補題も追加する：

```
Lemma length_of_pid_list :
  length pid_list = Bsp.p.
Lemma pid_list_is_non_empty :
  0 < length pid_list.
```

なお，これらは，以下の補題で示される（証明は割愛）．

```
Lemma length_of_gen_list :
 ∀(n q : nat) (H : n <= q) (xs : list _),
  length (list_gen n q xs H) = n + length xs.
```

以上のほか，計算量に関する記述や証明においては並列ベクトル（par 型）をリストとして扱いたいことが多いため，par を list に変換する関数を仮定する．

```
Parameter par2list: ∀{A:Set}, par A →list A.
```

3.3 プリミティブの形式化

以上の準備のもと，五つのプリミティブの型と仕様を計算量記述を伴う形に拡張した．以下，各々について述べる．

3.3.1 プリミティブ get

拡張後の get の型と仕様を以下に示す：

```
1 Parameter get :
2  ∀{A : Set}(v : par A), ∀(p : pid),
3  {! a !:! A !<! c !>!
4    c <= Bsp.L
5         + Bsp.g * @size A (par_sized v) a !}.
6 Axiom par_eq : ∀{A:Set} (v v': par A),
7  (∀(i:pid), valOf(get v i) = valOf(get v' i))
8   →v = v'.
```

プリミティブ get は，通信を行うため，その計算量にはバリア同期時間 L が含まれる．また，get が返す値 a を通信する時間が必要となるため，そのサイズ @size A (par_sized v)a に g を掛けた時間も含まれることになる．上記の形式化では，これらの時間で計算量 c が押さえられるという証明の付いた結果を返すように返り値の型が拡張されている（3–5 行目）．また，仕様の公理（6–8 行目）については，get の返す証明付きの値から純粋な値を取り出すために valOf を適宜挿入している以外は，既存の仕様と同じである．

3.3.2 プリミティブ mkpar

拡張後の mkpar の形式化を以下に示す：

```
1 Parameter mkpar:
2  ∀{A:Set},∀(fc : pid →nat)
3  (f : ∀p : pid,
4       {! a !:! A !<! c !>!c <= fc p !}),
5  {! va !:! par A !<! c !>!
6    c <= maximum 0 (map fc pid_list) !}.
7 Axiom mkpar_spec:
8  ∀(A : Set)(fc : pid →nat)
9  (f : ∀p : pid ,
10   {! a !:! A !<! c !>! c <= fc p!}),
11 ∀i : pid,
12 valOf(get(valOf(mkpar fc f))i) = valOf(f i).
```

プリミティブ mkpar はユーザ定義関数 f を使用するため，その計算量は f に依存する．そのため，f についてもその計算量の情報を戻り値に付加するよう，f の型が拡張されている（3–4 行目）．具体的には，f の計算量 c は，プロセッサ番号 p を引数とした関数 fc で押さえられるとしている．そして，mkpar の全体としての計算量は，BSP モデルの計算量に基づき，各プロセッサでの f の計算量（map fc pid_list）の最大値で押さえられるとしている（6 行目）．なお，この各プロセッサでの計算量を得るために，本研究では計算量を押さえるための関数（fc）を陽に受け取る形で形式化している．これを陽な引数として受け取らないように工夫することは今後の課題である．

仕様に関する公理部分（7–12 行目）は，上記の mkpar 自体の拡張に伴って引数が変わっていることと，証明付きの値から純粋な値を取り出すために valOf を適宜挿入していること以外は，既存の仕様と同じである．

3.3.3 プリミティブ apply

拡張後の apply の形式化を以下に示す：

```
1 Parameter apply:
2  ∀{A B:Set} (fc : (A*pid) →nat)
3  (vf : par ( ∀a : A, {! b !:! B !<! c !>!
4              ∀p : pid, c <= fc (a,p) !} ))
5  (vx: par A),
6  {! vb !:! par B !<! c !>!
7    c <= maximum 0
8       (map fc (zip (par2list vx) pid_list))!}.
9 Axiom apply_spec:
10 ∀(A B: Set)(fc : (A*pid) →nat)
11 (vf: par( ∀a : A, {! b !:! B !<! c !>!
12   ∀p : pid, c <= fc (a,p) !} ))
13 (vx: par A) (i: pid),
14  valOf(get(valOf(apply fc vf vx)) i)
15  =valOf((valOf(get vf i))(valOf(get vx i))).
```

プリミティブ apply も mkpar と同様に，ユーザ定義関数の計算量に全体の計算量が依存する．そのため，引数として受け取るユーザ定義関数の並列ベクトル vf の型が，計算量の証明を伴う値を返す関数の並列ベクトルとなっている（3–4 行目）．具体的には，その計算量は，同じく引数として与えられる fc に各プロセッサにおける入力 a とプロセッサ番号 p のペアを与えた結果の値で押さえられるとしている．また，全体の計算量がそれらの最大で押さえられるという点も mkpar と同様である（7–8 行目）．仕様に関する公理部分（9–15 行目）も同様に，純粋な値を取り出すために valOf を適宜挿入している程度の変更である．

3.3.4 プリミティブ put

拡張後の put の形式化を以下に示す：

```
1  Parameter put:
2   ∀{A: Set}(fc: (A*pid)→nat)(szdA: sized A)
3   (vf : par ( ∀p : pid, {! a !:! A !<! c !>!
4     c <= fc (a ,p) !} )),
5   {! vg !:! par (∀ p : pid,
6     {! a !:! A !<! c !>! c <= 1 !}) !<! c !>!
7     c <= Bsp.L + maximum 0 (map
8       (fun i ⇒ Bsp.g *
9         ((sum (map (@size A (szdA))
10                (map (fun j ⇒ valOf((valOf(get
                      vf i) j))) pid_list)))
11          +((sum (map (@size A (szdA))
12                (map (fun j ⇒ valOf((valOf(get
                      vf j) i))) pid_list)))))
13         +(sum (map (fun j ⇒ fc (valOf(valOf(
              get vf i) j),i)) pid_list)))
14       pid_list) !}.
15  Axiom put_spec:
16    ∀(A:Set)(fc: (A*pid)→nat)(szdA: sized A)
17    (vf:par ( ∀p : pid, {! a !:! A !<! c !>!
18      c <= fc (a ,p) !} ))(i j:pid),
19    valOf(valOf(get(valOf(put fc szdA vf))i)j)
20     = valOf(valOf(get vf j) i).
```

プリミティブ put の形式化は少々複雑になる．まず，これまでのプリミティブと同様，ユーザ定義関数の計算量を押さえる関数 fc を陽に受け取る．また，通信される要素そのものの並列ベクトルが手に入らないため，sized のインスタンスも陽に受け取るとした（2 行目）．そして，ユーザ定義関数については，やはり送信されるデータの生成に計算時間がかかるため，計算量証明を伴うようにしている（3–4 行目）．そして，put の戻り値の型のうちの純粋な値の部分は，関数の並列ベクトル vg であり，その要素の型は ∀p : pid, {! a !:! A !<! c !>! c <= 1 !} である（5–6 行目）．すなわち，与えられたプロセッサ番号 p に対し，単位計算時間で値 a を返すとしている．そして，put 全体の計算時間の記述が 7–14 行目となる．プリミティブ put の計算量は，BSP モデルの計算量に基づき，送信を完了するためのバリア同期時間 L と各プロセッサの消費する時間の最大値との和で押さえられる．プロセッサ i の消費時間は 8–13 行目の部分に記述されており，それは，送信する全データのサイズ（9–10 行目）と受信する全データのサイズ（11–12 行目）に g を掛けたものと，送信データの生成にかかる時間（13 行目）との和である．

仕様に関する公理部分（15–20 行目）は，純粋な値を取り出すために valOf を適宜挿入している程度の変更である．

3.3.5 プリミティブ proj

拡張後の proj の形式化を以下に示す：

```
1  Parameter proj:
2  ∀{A:Set}(v: par A),
3  {! f !:! pid →{! a !:! A !<! c !>! c <= 1 !}
4    !<! c !>! c <= Bsp.L + Bsp.g * (sum(map
5       (@size A (par_sized v))(par2list v)))!}.
6  Axiom proj_spec:
7   ∀(A:Set)(v: par A)(i: pid),
8    valOf (valOf(proj v) i) = valOf(get v i).
```

通信を伴う proj の計算時間は，バリア同期時間 L と，各プロセッサからの送信データを受信するのにかかる時間との和で押さえられる（4–5 行目）．また，proj の返す関数の計算量は，与えられたプロセッサ番号でテーブルを引くだけであるので，定数時間で押さえられるとしている（3 行目）．仕様に関する公理部分は，純粋な値を取り出すために valOf を適宜挿入しているのみで本質的な変更はない．

4. 評価

前節で提案した形式化を評価するために，実用で使われる並列計算である「接頭和計算」を本研究で拡張したプリミティブで記述して計算量の証明を行った．以下，まずは接頭和計算の説明を行い，その後に具体的な記述を示す．

4.1 接頭和計算

接頭和計算 $psum$ は，リストや配列を入力にとり，その各要素に対して先頭からの和を計算するものである：

$$psum\ [x_0, \ldots, x_n] = [x_0, x_0 + x_1, \ldots, x_0 + x_1 + \cdots + x_n]$$

例えば，$psum\ [3,1,4,1,5,9] = [3,4,8,9,14,23]$ である．

BSP モデルにおける接頭和計算の実現として，スーパーステップ 2 つからなる方法が知られている（例えば [9] を参照）．前提として，入力のリストや配列がプロセッサと同数の均等なチャンクに分割され，先頭のプロセッサから順にそれらのチャンクを保持しているとする．例えば入力が $[3,1,4,1,5,9]$ でプロセッサ数が 3 であれば，プロセッサ 0 が $[3,1]$ を，プロセッサ 1 が $[4,1]$ を，プロセッサ 2 が $[5,9]$ を保持する．

まず，最初のスーパーステップで，各プロセッサは，自分の保持するチャンクの和を求め，それを自分より後ろの

プロセッサに送信する．上記の例では，プロセッサ 0 がその和 4 をプロセッサ 1 とプロセッサ 2 に送る．同様に，プロセッサ 1 はプロセッサ 2 に自身の和 5 を送信する．そして，バリア同期を行いデータの配送を確定する．

そして，次のスーパーステップで，各プロセッサは，自身の受け取った値の総和を初期値として，自身のチャンクに対して接頭和計算を行う．例えば，プロセッサ 0 は何も値を受け取らなかったため，初期値 0 からの接頭和を計算して $[3, 4]$ を得る．プロセッサ 1 は 4 を受け取っているため，これを初期値として接頭和計算を行い，$[8, 9]$ を得る（すなわち，$[4 + 4, 4 + 4 + 1]$ である）．そして，プロセッサ 2 は 4 と 5 を受け取っているため，9 を初期値として接頭和を計算して $[14, 23]$（$= [9 + 5, 9 + 5 + 9]$）を得る．以上により，接頭和計算の出力全体が並列に計算された．

この接頭和計算の計算量は，バリア同期時間 L と，ローカル計算の時間 $O(n/p+p)$ と，通信にかかった時間 $O(gp)$ の和となるから，$O(L + n/p + gp)$ である．

4.2 BSML プリミティブに依る記述

前節で導入した接頭和計算を，本研究で拡張した BSML プリミティブで記述した結果を示す．紙面の都合により，本節では各定義で要求される証明部分を割愛する．

各ステップの説明は後ほど順に述べることとして，まずはプログラムの全体像を示す：

```
1  Program Definition
2    p_sum (cs : Bsml.par (list nat))
3      : {! pa !:! Bsml.par (_) !<! c !>!
4           c <= p_sum_calc cs !}
5    := mks <- Bsml.mkpar fc_mksump mk_sum;
6       ss <- Bsml.apply fc_sum mks cs;
7       mps <- Bsml.mkpar fc_send send;
8       fs <- Bsml.apply fc_send' mps ss;
9       gs <- Bsml.put fc_fs nat_sized fs;
10      mkg <- Bsml.mkpar fc_gj mk_gj;
11      hs <- Bsml.apply fc_gj' mkg gs;
12      ress <- Bsml.apply _ hs cs;
13      <== ress.
```

この関数 p_sum は，チャンクの並列ベクトル cs を受け取り（2 行目），接頭和計算の結果を同じ形のチャンクの並列ベクトル ress として返す（13 行目）．そして，その計算量 c が，次で定義される p_sum_calc で押さえられる（4 行目）ということが証明できた．

```
1  Definition p_sum_calc cs :=
2   8 + Bsml.Bsp.L
3    + 4 * Bsml.maximum 0 (map (length (A:=nat))
4                              (Bsml.par2list cs))
5    + (Bsml.Bsp.g * 8 + 3) * Bsml.Bsp.p.
```

この p_sum_calc は，バリア同期時間 L（2 行目）と，チャンクの最大サイズ（3–4 行目）と，単位メッセージの通信時間 g とプロセッサ数 p との積（5 行目）の線形結合である．よって，チャンクが均等に分けられているという仮定のもと，このプログラムの計算量が前節で述べた計算量 $O(L + n/p + gp)$ であることが形式的に保証された．

以下，p_sum のプログラム全体の流れを説明する．

まず，mkpar を用いて，各プロセッサ上に「チャンク内の要素の和を計算する関数」を用意する（5 行目）．そこで使われている関数 mk_sum は，計算量証明付きの sum 関数を返すだけの関数であり，その計算量は定数関数 fc_mksump で押さえられる（定数コスト）：

```
1  Definition fc_mksump (p : Bsml.pid) := 1.
2  Program Definition mk_sum(p : Bsml.pid) :
3  {! a !:! ∀x' : list nat,
4    {! ret !:! nat !<! c !>!∀ p : Bsml.pid,
5                          c <= fc_sum (x',p) !}
6   !<! c !>! c <= fc_mksump (p) !}
7    := <== sum.
```

また，計算量証明付きの sum 関数とその計算量 fc_sumは次のように定義した（リスト長に比例するコスト）：

```
1  Definition fst{A B:Type}(ab : A*B) : A
2    := match ab with (a,b) ⇒ a end.
3  Definition snd{A B:Type}(ab : A*B) : B
4    := match ab with (a,b) ⇒ b end.
5  Definition fc_sum (ap: (list nat) * Bsml.pid):
       nat := 1 + 2 * (length (fst ap)).
6  Program Fixpoint sum(x' : list nat) :
7  {! ret !:! nat !<! c !>!
8    ∀p : Bsml.pid, c <= fc_sum (x',p) !}
9  := match x' with
10     | nil ⇒ +=1; <== 0
11     | cons a x ⇒ s <- sum x; += 2; <== a + s
12     end.
```

次いで，apply を用いてその関数を各プロセッサのチャンクに適用し，チャンクの内の要素の和からなる並列ベクトル ss を作る（6 行目）．

続く 7–9 行目で，put を用いてその総和を自身の後ろのプロセッサに送信する．この部分での処理の本質は次の send 関数であり，これは，送信元プロセッサ番号 i とそのチャンクの和 s と送信先プロセッサ番号 j を受け取り，i<j なら s を，そうでなければ単位元 0 を返すというものである（12 行目）：

```
1  Definition fc_send (p : Bsml.pid) := 1.
2  Definition fc_send'(ap : nat * Bsml.pid) := 1.
3  Definition fc_fs(ap : nat * Bsml.pid) := 1.
4  Program Definition send (i : Bsml.pid) :
5  {! a !:! ∀s : nat ,
6   {! fs !:! ∀p :Bsml.pid ,
7    {! ret !:! nat !<!c!>! c <= fc_fs(ret,p)!}
```

```
8    !<!c!>! ∀p: Bsml.pid, c <= fc_send'(s,p)!}
9  !<!c!>!c <= fc_send i !}
10 := <== (fun s : nat ⇒
11    (<== (fun p : Bsml.pid ⇒
12     (<== if Nat.ltb i p then s else 0)))).
```

この関数は，動作はとても単純であるものの，三つの引数を必要とする．BSML のプリミティブでは引数を一つずつしか与えることが出来ないため，「引数を一つ受け取る毎に計算量証明付きの関数を返す」という構造が三回ネストすることになる．結果として，戻り値の型が尋常ならざる面倒さとなっている（5–9 行目）．この面倒の解消は今後の課題である．

最後に，続く 10–12 行目で，各プロセッサが，前方のプロセッサから受け取った値の和を初期値として自身のチャンクに対する接頭和の計算を行う．この計算の主要部分は，送られてきた値の総和を取る sum と，以下に定義されるローカルの接頭和を計算する psum であり，これらはさらに以下に定義される gj_sum で連続して使用される：

```
1  Definition fc_psum(ap :(list nat) * Bsml.pid)
2    := 1 + length (fst ap) * 2 .
3  Program Fixpoint psum(n : nat)(x' : list nat)
4   : {! ln !:! list nat !<! c !>!
5      ∀p : Bsml.pid, c <= fc_psum (x',p)!}
6   := match x' with
7      | nil ⇒ +=1; <== nil
8      | a::xs ⇒ p <- psum (a+n) xs;
9                +=2; <== ((a + n)::(p))
10     end.
11 Definition fc_pssum (ap : (list nat)* Bsml.pid)
12  := fc_sum(map Bsml.pid2nat Bsml.pid_list,
       snd(ap)) + fc_psum(ap).
13 Definition fc_gj (p : Bsml.pid) : nat := 1.
14 Program Definition gj_sum
15  (g : Bsml.pid →{!_ !:! nat !<!c!>! c<=1!}):
16  {!f_p!:! ∀x' : list nat,
17    {! ln !:! list nat !<! c !>!
18      ∀p : Bsml.pid, c <= fc_pssum (x',p)!}
19    !<! c !>! ∀p : Bsml.pid, c <= fc_gj p!}
20  := fun x' : list nat ⇒ (
21       n <- sum (map (@valOf nat _)
22                     (map g Bsml.pid_list));
23       psum n x').
```

我々の目的はこの gj_sum に必要な引数を与えて動作させることであるが，プリミティブを用いた記述では引数を一つずつしか渡せないため，さらに以下の関数 mk_gjを定義してプリミティブに渡している．これは gj_sum を返すだけの極単純な関数であるが，send と同様，型の記述が非常に複雑で面倒である：

```
1  Program Definition mk_gj (p : Bsml.pid ) :
2  {!a!:!∀ g:(Bsml.pid→{!_!:!nat!<!c!>!c<=1!}),
       {!f_p!:!∀ x:list nat,{!ln!:!list nat
3        !<!c!>!∀ p:Bsml.pid,c<=fc_pssum(x,p)!}
4      !<!c!>!∀ p:Bsml.pid,c<=fc_gj p!}
5    !<!c!>!c <= fc_gj p !}
6  := <== gj_sum.
```

5. おわりに

本論文は，BSP モデルを対象として，並列計算量の形式的証明を伴う並列プログラムを記述するための Coq ライブラリを提案した．実用的な並列計算である「接頭和計算」について，実際に並列プログラムを計算量証明付きで記述できることが確認された．

今後の課題として，ユーザ定義関数の型が複雑になってしまう問題への対処が考えられる．この複雑さの原因の一つは，BSML プリミティブを用いる際に，ユーザ定義関数が引数を一つずつしか受け取れないことにある．この制限を緩和するよう，プリミティブよりも抽象度の高い並列スケルトンを提供することで，ある程度の解決が期待される．また，プリミティブが計算量に関する記述のための関数を陽に受け取ってしまっている点も，抽出されるコードの効率の観点からは解消したい．この点も今後の課題となる．

謝辞　本研究は JSPS 科研費 JP19K11903 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*, Springer (2004).
[2] Emoto, K., Loulergue, F. and Tesson, J.: A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction, *Interactive Theorem Proving*, pp. 258–274 (2014).
[3] Kokke, W., Siek, J. G. and Wadler, P.: Programming Language Foundations in Agda, *Science of Computer Programming*, Vol. 194, p. 102440 (2020).
[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, pp. 300–319 (2017).
[5] Loulergue, F., Hains, G. and Foisy, C.: A Calculus of Functional BSP Programs, *Science of Computer Programming*, Vol. 37, No. 1–3, pp. 253–277 (2000).
[6] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G.: Pregel: a system for large-scale graph processing, *SIGMOD '10*, pp. 135–146 (2010).
[7] McCarthy, J., Fetscher, B., New, M., Feltey, D. and Findler, R. B.: A Coq Library For Internal Verification of Running-Times, *FLOPS 2016*, pp. 144–162 (2016).
[8] Valiant, L. G.: A Bridging Model for Parallel Computation, *Communications of the ACM*, Vol. 33, No. 8, pp. 103–111 (1990).
[9] 松崎公紀，江本健斗：BSP（Bulk Synchronous Parallel）モデル再訪–BSP によるマルチコアプログラミング大規模グラフフレームワーク–，情報処理，Vol. 56, No. 5, pp. 482–488 (2015).