プログラムスライスを用いた分散プログラムのデッドロック原因究明方式

岡田 達彦，太田 剛，水野 忠則
静岡大学情報学部

本稿では，分散プログラムにおけるデッドロックの原因をプログラムスライシング技術を用いて究明する方式について述べる．スライスを用いてデッドロックの原因を究明するためには，スライスに含まれる文を時間順に整列する．このとき，デッドロックを起こすプログラムでは，時間順序に不整合を生じる．この不整合を起こす部分を手掛りにして原因を究明する．これによって，ユーザが処理しなければならない情報を減らすことができ，誤っている部分を限定できる．

1　はじめに

分散処理環境下で稼働する分散プログラムの開発には，従来の逐次型プログラムにはない難しさが存在する．それは，プログラムの挙動の非決定性，プローブ効果，絶対時間の欠如等に起因する [1, 2]．また，デッドロックのような分散プログラム固有の問題がデバッグを困難にする．これらの要因のため，分散プログラムのデバッグは，試行錯誤で進められることが多い．

逐次型プログラムのデバッグ技術の1つとして，プログラムスライシング技術がある．プログラムスライシング技術は，プログラム中の文の間に存在する依存関係によって，プログラム中のある特定の文に影響を与える文を抽出する技術である [4, 5]．抽出された文集合をスライスと呼ぶ．スライスには，プログラムテキストを解析した静的スライスと，ある1つの入力に対してその実行系列を解析した動的スライスの2つがある [5]．スライスを求めることで，プログラムテキストを削減でき，プログラムテキスト上でのバグの探索範囲を狭めることができる．また，スライスを求めることによって，プログラム中の文の間の依存関係を明らかにすることができる [4, 5]．現在，プログラムスライシング技術は，プログラムのデバッグに限らず，テスト，保守，プログラム理解へと広範囲に応用されている [5]．

プログラムスライシング技術を逐次型プログラムのデバッグに適用した研究は，Weiser のデータフロー分析によるスライス計算法に始まる [4]．他のスライス計算方法として，Horwiz がプログラムテキストの解析によって求めるプログラム依存グラフ (Program Dependence Graph) を用いる方法 [8] を，Bergeretti がμ関係行列を用いる方法 [9] を示している．下村は，クリティカルスライス (Critical Slice) を定義し，それを用いたアルゴリズミックなバグ究明方式を提案している [11]．

一方，分散プログラムのデバッグにおいても，プログラムスライシング技術を適用

A Fault Localization Scheme for Deadlocks in Distributed Programs using Program Slicing
Tatsuhiko Okada, Tsuyoshi Ohta and Tadanori Mizuno
Faculty of Information, Shizuoka University

した研究が行なわれている．分散プログラムのデバッガは，プロセスの内部状態やプロセス間の同期など，複数個の要素を観測しなければならず，必要な情報が非常に多くなる．この大量の情報は，プログラムスライシングを行なうことで，逐次型プログラムと同じように削減できる．これによって，ユーザは問題の解決のために必要な情報だけを取得できる．これを目的として，Cheng は並行プログラムのスライスを定義した [10]．また，Taylor はデータフロー分析により，シングルプロセスと並行プロセスの両方に対して，プログラムの異常の有無を発見できることを示した [7]．

だが，分散プログラムのデッドロックの原因を究明しようとするときには，文の依存関係をたどり，文の記述の誤りや変数値の誤りを修正するだけでは解決できない．なぜなら，個々のプロセスが正しく動作したとしても，複数個のプロセスが同時に動作したときの相互作用が，デッドロックを起こす原因となるからである．このため，デッドロックを解決するためには，ユーザが試行錯誤でテストを繰り返してデッドロックに至る実行過程を追及することになり，非常に労力が要る．

この問題を解決するために，本稿では分散プログラムを対象に，動的スライスを用いたデッドロックの原因究明方式について述べる．本方式を適用することで，デッドロックの原因となるサイクルを形成する送受信文の組み合わせを半自動的に求めることができ，ユーザが処理しなければならない情報を減らすことができる．対象とするプログラミング言語は，構造化プログラミングが可能な手続き型言語とし，通信方式は同期式メッセージパッシング方式とする．

空文：	
代入文：	識別子=式
if文：	if (条件式) |文|
if-else文：	if (条件式) |文| else |文|
while文：	while (条件式) |文|
入力文：	read (識別子)
出力文：	write (式)
送信文：	send (送り先プロセス名, 式)
受信文：	recv (受け元プロセス名, 識別子)
変数宣言：	int 識別子
プロセス定義：	process プロセス名 |変数宣言| |文|

表 1: モデル言語の構成

2 分散プログラムのプログラムスライシング

この節では，本稿で取り扱うプログラミング言語を定義し，動的スライスを計算するために必要な，プログラム中の文の間の依存関係を説明する．そして，分散プログラムのスライス計算に必要な，送受信文の解釈法について述べる．

2.1 プログラム言語のモデル

本稿で対象とするプログラミング言語は，構造型プログラミング言語に最低限必要な文にプロセス定義，送信文，受信文を追加した手続き型言語とする．すなわち，この言語は，空文，代入文，if 文，if-else 文，while 文，入力文，出力文，送信文，受信文，変数宣言，プロセス定義から構成される (表 1)．この文法は，プロセスの生成 (fork)，結合 (join) を行なう文を含まない．また，関数，手続きの定義はない．変数は個々のプロセス内で有効なスカラ型に限り，共有変数は使用できない．プロセス間の通信方式は，同期式メッセージパッシング方式とする．

2.2 プログラムスライシング技術

本稿で述べる手法は，動的スライス (以下スライス) を用いる．スライスは，ある入力に基づいてプログラムを実行したとき，ある文の実行に実際に影響を与えた文の集合である [5, 6]．プログラムの実行系列内の文を X^p(p 番目の文の実行時に，実行された文 X) とする．このプログラム実行系列からスライスを抽出するためには，プログラム中の文の間に存在する依存関係を用いる．それは，次の 2 種類である．

- Data Influence
  文 X^pから文 Y^qへの Data Influence があるとは，文 Y^qの実行で使用したある変数を最後に定義した文が文 X^p である場合である．すなわち，

  1. $p < q$.
  2. 文 X^pの実行において変数vを定義した．
  3. 文 Y^qの実行において変数vを使用した．
  4. $p < k < q$となる任意の kに対して，文^{k}は変数vを定義しなかった．

  以上の 4 つを満たす場合である．

- Control Influence
  文 X^pから文Y^qへの Control Influence があるとは，文 X^pが if 文か while 文であり，文 Y^qの実行の有無を直接決定した文である場合．

この 2 つの依存関係をたどって得られる文集合がスライスである．スライス基準は，$C(i, V)$ と定義される．ここで，iはプログラム実行系列内の文，V はプログラム中の変数の集合である．

次にプログラムスライシングを分散プログラムに適用するためにプロセス間通信を次のように解釈する [12]．

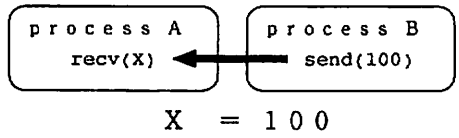

図 1: プロセス間通信と代入文

2.1節で述べたように，通信方式を同期式メッセージパッシング方式とした．これによって，プロセス間通信をプロセス間にまたがる代入文と考えることができる．すなわち，送信動作は情報を保持している変数を使用することであり，受信動作は変数に値を定義することである．このように解釈することによって，送信動作は代入文の右辺，受信動作は左辺に相当することになり，プロセス間通信を単純にプロセス間にまたがる代入文と考えることができる．図 1はこの様子を示している．

この解釈によって，Data Influence は 1 つのプロセスだけでなく，2 つのプロセスにまたがる場合が生じる．このとき，1 つの受信文に対して複数個の送信文がメッセージを送る可能性があるが，依存関係をたどるときに参照するのは，プログラム実行時に実際に対応付けられた送受信文の組み合わせとする．

3 デッドロック原因究明方式

プログラム実行の結果，デッドロックが生じたとする．このとき，プログラムの実行系列からスライスを計算する．計算したスライスを用いて，デッドロックの原因を究明するためには，スライスに含まれる文を時間順に整列する．このとき，デッドロックを起こすプログラムでは，時間順序に不整合 (サイクル) を生じる．これは，デッドロックを起こすプログラムがサイクルを含むという事実に基づく [13]．このサイクルを手掛りにして原因を究明する．これによっ

```
     process A                    process B
     {                            {
       int x,y,z;                   int p,q,r;
A1:    read(x);             B1:     read(p,q);
A2:    recv(B,y);           B2:     send(A,p);
A3:    i = x+y;             B3:     send(C,q);
A4:    if (i > 0) {         B4:     recv(A,r);
A5:      send(B,x);         B5:     send(C,r);
A6:      recv(C,z);         B6:     write(p+q+r);
       } else {                   }
A7:      recv(C,z);
A8:      send(B,x);           process C
       }                          {
A9:    write(z*y*z);                int l,m,n;
     }                        C1:    read(l);
                              C2:    recv(B,m);
                              C3:    recv(B,n);
                              C4:    send(A,l);
                              C5:    write(l-m-n);
                                   }
```

図 2: プログラムの例

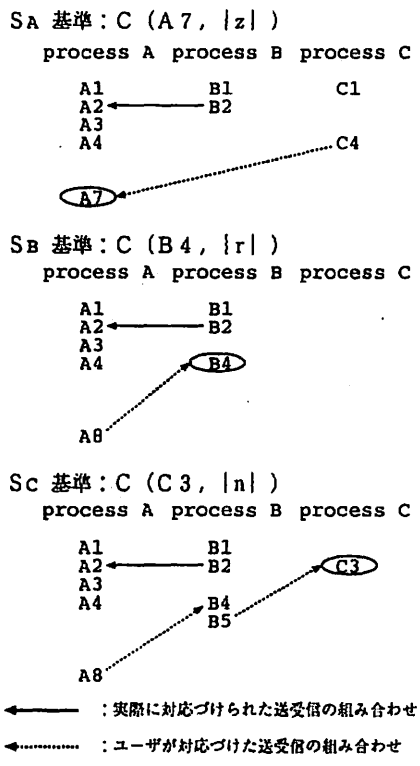

図 3: 図 2 のプログラムの各プロセスごとのスライス例

て，ユーザが処理しなければならない情報を減らすことができ，誤っている部分を限定できる．

この節では，スライスの計算法，文の整列法について説明し，デッドロックの原因を究明する方式を述べる．

3.1 スライスの計算

最初に，各プロセスごとにスライスを計算する．スライス基準は，プロセス中にあるデッドロックを起こした受信文 (i) とその受信文に含まれている変数 (V) である．

スライス基準の決定後，Data Influence, Control Influence の 2 つによって，i に影響を与えた文を抽出していく．他プロセスからの受信文が抽出されたならば，その受信文に対して実際にメッセージを送出した送信文を抽出する．そして，抽出された送信文を基準として，同じように 2 つの依存関係をたどり，文を抽出していく．

図 2で示したプログラムを例に，上記の操作を説明する．図中の recv(B,y) は，プロセス B から送信された値を変数 y に代入する受信文で，send(B,x) は，プロセス B へ変数 x の値を送信する送信文である．このプログラムは，A7,B4,C3 においてデッドロックを起こし得る．このときの実行系列が，例えば，[A1,A2,A3,A4,A7]，[B1,B2,B3,B4]，[C1,C2,C3] であったとする．この場合の各プロセスごとのスライスを図 3 に示す．

プロセス A のスライス基準は C(A7,{z})，プロセス B は C(B4,{r})，プロセス C は C(C3,{n}) である．次に基準の文に対応する送信文を求める．A7 に対応する送信文は，プログラムテキストから，C4 だけであることがわかる．これにより，A7 が待ち合わせることになった送信文は，C4 であったといえる．同様に，C3 が待ち合わせることになった送信文は，B5 となる．よって，(A7,C4)，(C3,B5) を送受信文の組み合わせとして対応付け，C4,B5 を抽出する．だが，B4 には，対応する可能性を持つ送信文が 2 つある (A5 と A8)．これら 2 つのうち，ど

```
     process A
     {
       int x,y,z;
A1:    read(x);
A2:    recv(B,y);
A3:    i = x+y;
A4:    if (i > 0) {

       } else {
A7:      recv(C,z);
A8:      send(B,x);
       }
     }
```

```
     process B
     {
       int p,q,r;
B1:    read(p,q);
B2:    send(A,p);

B4:    recv(A,r);
B5:    send(C,r);

     }

     process C
     {
       int l,m,n;
C1:    read(l);

C3:    recv(B,n);
C4:    send(A,l);

     }
```

図 4: 図 2 のプログラムのスライス例

ちらか一方を選択し抽出する．図 3は，A8 を選んだ場合のスライスである．このとき選ぶ送信文は，ユーザが持つプログラムの知識に基づいて，ユーザが判断する．計算したスライスを以下に述べる方法によって分析しても原因が判明しない場合，選択した送受信文の組み合わせでは，デッドロックは起きない．この場合には，残りの可能性の中から 1 つ選択し，スライスを計算し直す．

このようにして，プロセスごとに計算したスライスには，デッドロックを起こした文に影響を与えた文をすべて含んでいる．このプロセスごとのスライスの和集合をプログラムのスライスとし (図 4)，本方式ではこれを分析する．

3.2 文の整列

プログラム中のプロセス間通信の時間的な不整合を発見するために，計算したスライスに含まれる文を時間順に並び換える．だが，プロセスごとに時間軸が独立しているため，スライスに含まれる文を 1 つの時間軸上に並べることはできない．この問題を解決するために Lamport のベクトルクロック (vector clock) [3] を用いて，時間半順序関係に従って並び換える．

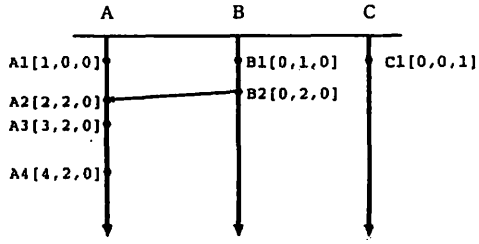

図 5: ローカル時間ベクトル

文を整列するために，ローカル時間ベクトルを用いる．ローカル時間ベクトル Tは，プログラムに n 個のプロセスがあるとき，n 個の要素から構成される．

$$T_k^P = [t_1, t_2, \cdots, t_n]$$

ここでの Pはプロセス名，kはプロセス内の実行時点を示している．$t_1, t_2, \cdots, t_n$は，プロセス Pからみた各プロセスのローカル時間を示した整数値である．プロセスの開始時点では，

$$T_0^P = [0, 0, \cdots, 0]$$

となる．図 5は，図 4のスライスの前半部分のローカル時間ベクトルを表している．図中の各ローカル時間ベクトルは，3 つの要素から成り立つ．これは，図 4のスライスが 3 つのプロセスを持つためである．これらの要素は各プロセスに対応しており，プロセスにおける内部処理や通信の実行により 1 増加する．図 5では，A1 から A2 の変化がこれに相当する．さらに，本稿では同期通信のみを扱うので，プロセス間で通信が行なわれるときには，送信側のプロセスと受信側のプロセスが持っているベクトルの各要素を比較し，各要素ごとに大きい値を選ぶことによって，受信側の新しいローカル時間ベクトルが求められる．図 5では，B2 から A2 への通信がこれに相当する．

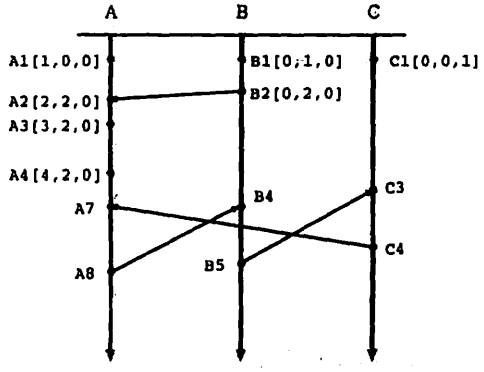

図 6: サイクル

3.3 原因の究明

デッドロックが起きないプログラムでは，スライス中のすべての実行時点において，ベクトルクロックを矛盾なく求めることができる．だが，デッドロックを起こしたプログラムのスライスでは，ベクトルクロックを求めることができない．これは，プログラムの実行系列にサイクルがあるときに起こる．

図 4のスライスはサイクルを含んでいる(図 6)．この例では，A7,B4,C3 のローカル時間ベクトルを求めることができない．なぜなら，プロセスごとの時間軸上の後方に位置する文のローカル時間ベクトル値 (A8,B5,C4) を必要とするからである．この時点で，A7,C4,C3, B5,B4,A8 がサイクルを形成していることがわかる．

ここで発見されたサイクルを形成する送受信の組み合わせが，デッドロックの原因として挙げられる．このとき，プログラム中にサイクルを形成する送受信文の組み合わせがあることをユーザに提示できれば，ユーザはそのような送受信文の対応付けを行なう制御移行をしないようにプログラムを修正することで，プログラム実行時に起こるデッドロックを防げる．プログラムの制御移行が正しいとユーザが判断した場合には，サイクルを形成している送受信文の実行順序を入れ替えることでサイクルを解除する．

例で示したプログラムにおいては，問題となっている A7,A8 が A4 の制御下にあるため，A4 の式が誤っていることが考えられる．もし，ユーザが，A4 が誤っていると判断すれば，else 部に制御が移らないように A4 を修正することでデッドロックを解除できる．逆にユーザが，A4 から else 部への制御移行が正しいと判断したときは，(A7,A8), (B4,B5), (C3,C4) のいずれかの文の実行順序が誤っていることがデッドロックの原因である．この場合はいずれかの実行順序を入れ替えることで，サイクルを解除する†．

4 おわりに

本方式をデッドロックを起こし得る分散プログラムに適用することで，プログラム中のサイクルを形成する送受信文の組み合わせが判明する．だが，現段階では，ユーザがプログラム中の送受信文の組み合わせの正誤を判断できると仮定した場合に限り有効である．なぜなら，発見されたサイクルがデッドロックの原因になるかどうかの判断をユーザに委ねているためである．また，発見されたサイクルの解除方法も複数考えられるため，どの方法が最も適しているかの判断もユーザに委ねている．このため，プロセスの相互関係をユーザに視覚的に示す機能が重要になる．特に，ユーザが複雑なプロセス間通信の様子を容易に理解できるようにしなければならない．

また，本稿では，通信方式として同期式メッセージパッシングを用いているが，非同期通信や共有メモリによるプロセス間通信は，単純にプロセスをまたがる代入文と解釈することができず，検討が必要である．

†ただし，この例では，B4 から B5 への Data Influence があるので，B4 と B5 とを入れ換えることはできない．

今後，実用的なプログラムに対して本方式を適用し，評価を行なう予定である．そして，本方式を形式的に記述し，分散プログラムのアルゴリズミックなデバッグ手法への拡張を考えている．

参考文献

[1] McDowell,C.E. and Helmbold,D.P.: "Debugging Concurrent Programs", ACM Computing Surveys, Vol.21, No.4, pp.593–622 (1989).

[2] 山田剛: "並列処理におけるプログラムデバッグ", 情報処理, Vol.34, No.9, pp.1170–1178 (1993).

[3] Lamport,L.: "Time, Clocks, and the Ordering of Events in a Distributed System", Comm. ACM, Vol.21, No.7, pp.558–pp.565 (1978).

[4] Weiser,M.: "Program Slicing", IEEE Trans. on Software Engineering, Vol.SE–10, No.4, pp.352–357 (1984).

[5] 下村隆夫: "Program Slicing 技術とテスト, デバッグ, 保守への応用", 情報処理, Vol.33, No.9, pp.1078–1086 (1992).

[6] Korel,B. and Laski,J.: "Dynamic Program Slicing", Information Processing Letters, Vol.29, No.10, pp.155–163 (1988).

[7] Taylor,R.N. and Osterweil,L.J.: "Anomaly Detection in Concurrent Software by Static Data Flow Analysis", IEEE Trans. on Software Engineering, Vol.SE–6, No.3, pp.265–278 (1980).

[8] Horwiz,S. and Reps,T.: "The Use of Program Dependence Graphs in Software Engineering", the 14th International Conference on Software Engineering, pp.392–pp.411 (1992).

[9] Bergeretti,J.-F. and Carré,B.A.: "Information-Flow and Data-Flow Analysis of while-Programs", ACM Trans. Program. Lang. & Sys., Vol.7, No.1, pp.37–61 (1985).

[10] Jingde Cheng: "Slicing Concurrent Programs", in "Automated and Algorithmic Debugging" (Lecture Notes in Computer Science 749), Springer-Verlag, pp.223–240 (1993).

[11] 下村隆夫: "変数値エラーにおける Critical Slice に基づくバグ究明戦略", 情報処理, Vol.33, No.4, pp.501–pp.511 (1992).

[12] 太田剛, 渡辺尚, 水野忠則: "プログラムスライシングの分散プログラムへの適用", 情処研報, マルチメディア通信と分散処理,65–8, pp.43–pp.48 (1994).

[13] 前川守, "オペレーティングシステム", 岩波書店 (1993).