2G−6

# 分散プログラムデバッガのためのチェックポイントアルゴリズム

青柳滋己　　真鍋義文
NTTソフトウェア研究所

## 1 はじめに

並列・分散プログラムでは、あるプロセスで生じたエラーがプロセス間通信によって他のプロセスに伝播しそこで顕在化することもあり、プログラムのデバッグを非常に困難なものにしている。従来の並列・分散デバッガは個々のプロセスの実行を止めてその時点での変数の値などを調べたり、あるいはプロセス間通信の様子を調べたりする機能は持っているがそれだけではエラーが他のプロセスに伝播して顕在化するバグの原因を発見するには不十分である。メッセージの流れに沿ってデバッグの視点を順方向・逆方向に移動できるcausal debugging[1]を用いると、このようなバグの原因を発見するのに有効と考えられる。メッセージの流れを逆方向にたどる視点移動のとき、causal debuggingではreverse execution法[2]を用いる。reverse execution法とは実行の途中で随時チェックポイントをとり、それをもとに目的の場所に戻る方法である。本稿ではcausal debuggingのために考案したチェックポイント・ロールバックのアルゴリズムを報告する。

## 2 Causal Debugging

分散プログラムの実行中あるプロセスでエラーが発生し、それがその場所で顕在化せずにプロセス間通信によって他のプロセスにエラーを含んだ通信がなされ、別のプロセスでエラーが顕在化する場合を考える。

従来のデバッガでは一般にブレークポイントをエラーが顕在化した場所付近に設定し実行を止め、プロセスの様子を詳細に調査し、その原因を求め、先に止めた地点よりも前に別のブレークポイントを設定して再実行し直し原因を求め…を繰り返す。しかしこの方法では、エラーの原因がエラーが顕在化した場所よりはるか以前のことであった場合、原因となった箇所を発見するために最初からの再実行を繰り返す必要があるため、非常に手間がかかる。しかしcausal debuggingではメッセージの流れ(causality)に沿い、送信したメッセージの受信プロセスでの扱われ方、あるいは受信したメッセージが送られてきた原因を順次たどって調べることができる。

causal debuggingでは注目プロセスを1つ決め、それを変更していく。注目プロセス以外のプロセスは真鍋ら[3]の最小実行のアルゴリズムを用いて常に最小限の部分まで実行させる。注目プロセスがメッセージ送信を実行中に、その受信側プロセスでの処理を見たいときには注目プロセスを受信側プロセスに移し、そのメッセージを受信したところまで受信側プロセスの実行を進める。その他のプロセスは、最小限の地点まで実行を進める。図1で、$P_1$ の $t_4$ の地点まで実行していたとする。$P_1$ がその後送ったメッセージ $M_1$ が $P_2$ でどのように処理されるか調べる場合には、causal debuggingでは注目プロセスを $P_2$ に移し、$P_2$ を現在の $t_5$ から $t_8$ まで実行させる。また、$P_1, P_3$ は $t_8$ とconcurrentな最初の場所 $t_7, t_9$ まで実行させる。

また、注目しているプロセスの受信メッセージが、送信側プロセスでどのような処理をされて送られてきたのかを調べたいときには、送信側プロセスの、メッセージ送信を行なった手前に戻る。具体的にはそのメッセージ送信より前で、一番最後に他のプロセスからメッセージをもらった直後の地点まで戻る。これは、他からのメッセージにより、このプロセスがメッセージを送ることになった可能性が最も高いからである。

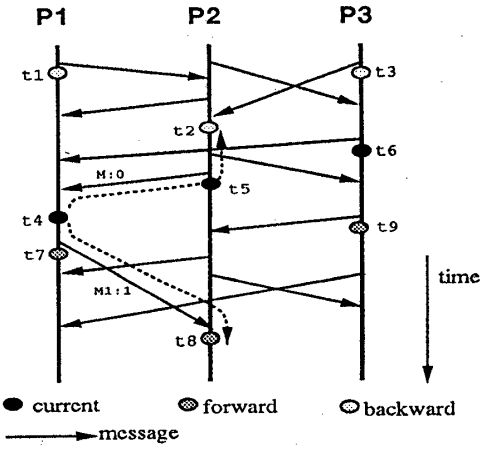


図1: Causal Debugging

このとき目的の地点はすでに実行されてしまっているので、その場所に戻るためには実行をやり直さなくてはならない。causal debuggingではreverse execution法により最初から再実行するのを防ぐ。reverse executionでは各プロセスが随時チェックポイントをとり、目的の地点に最も近いチェックポイントからすべてのプロセスを復帰させ、実行を再開する。causal debuggingのためには戻る場所は矛盾のないconsistent stateであり、かつ戻り場所の指定のないプロセスについてはconsistent stateとなる最初の場所[3]に戻るのが良い。図1で、メッセージMを受信していてその内容が値0であったが、本来は正の値が来るはずであったとする。このときにはメッセージMの送信プロセス $P_2$ の、$t_2$ の場所にロールバックし、デバッグの視点を $P_2$ にする。$P_2$ を $t_2$ から再実行することにより、$P_2$ がメッセージMで値0を送ることになった過程を調べることができる。またプロセス $P_1$, $P_3$ についてはそれぞれconsistentとなる最初の場所 $t_1, t_3$ にロールバックする。

エラーが顕在化した箇所から逆にエラーを顕在化させた原因となったメッセージの送信場所を調べ、次にそのメッセージを送る原因となったメッセージを探し、その送信場所を調べ… のように、メッセージの流れを逆にたどっていくことができれば、エラーの原因となった場所にたどり着くことが容易である。これは先に述べた、メッセージの送信側の原因を調べる方法を次々に行なうことで実現できる。

## 3 チェックポイント・ロールバックのアルゴリズム

各プロセスで、チェックポイントは他のプロセスとは独立に、適当な場所で取る。チェックポイントには1,2,…というIDをつける。これをチェックポイント番号と呼ぶ。各プロセスはメッセージ送信イベントにも同様に1,2,…というIDをつける。これを送信イベント番号と呼ぶ。送信イベント番号とチェックポイント番号のIDは独立である。

任意のチェックポイントはプロセス番号とチェックポイント番号で一意に定まり、また任意のメッセージ送信イベントも同様にプロセス番号と送信イベント番号で一意に定まる。

各プロセスは送信イベントベクトル SV=< $n_1, n_2, \ldots, n_N$ > を保持している。この送信イベントベクトルはMattern[4]のベクトル時

---

A Checkpointing Algorithm for Distributed Program Debugger
Shigemi AOYAGI, Yoshifumi MANABE
NTT Software Laboratories

計とほぼ同じものであるが、時計が進むのはメッセージ送信のイベント発生時だけである。$n_i$ はそのプロセスが知り得るプロセス $P_i$ の最も新しい送信イベント番号を表す。メッセージ送信時には自分のもつSVをメッセージにつけて送る。メッセージ受信の際にはメッセージを処理する前にメッセージと共に送られた送信イベントベクトルを取りだし、自分の持つ送信イベントベクトルの内容を更新する。

### 3.1 メッセージ送信・受信の手続き

Procedure 1 プロセス $P_i$ におけるメッセージ送信・受信、チェックポイントの手続き

初期状態: SV[j]=0(1≤j≤N), CP=0

メッセージ送信時:

1. SV[i] := SV[i] + 1
2. SV と共にメッセージを送信。
3. 送信イベント発生と送信先プロセス番号、SV[i] を履歴に記録。

メッセージ受信時:

1. メッセージから送信イベントベクトル(MSV)をとりだす。
2. SV と MSV の要素を比較し、大きいほうを SV の値にする。すなわち、SV[k] := max(SV[k],MSV[k])(1≤k≤ N)。
3. 受信イベント発生とメッセージ内容、送信プロセス番号、SV を履歴に記録。
4. メッセージを処理。

チェックポイントをとる時: (任意の場所で行える)

1. CP(チェックポイント番号) := CP + 1
2. チェックポイントを取る。
3. チェックポイントイベント発生とチェックポイント番号、SV を履歴に記録。

### 3.2 ロールバックの手続き

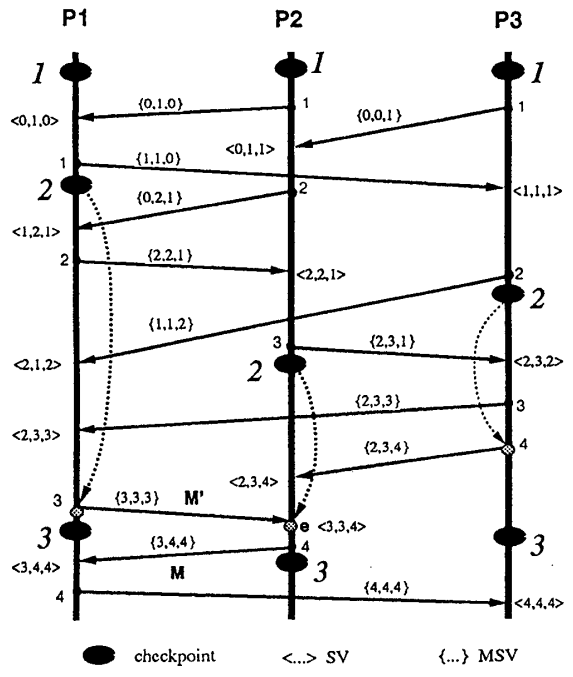


図 2: ロールバック

図2において、プロセス $P_1$ のチェックポイント3以降のデバッグを実行中に、メッセージMについて逆方向の視点移動を行なう場合を考える。causal debuggingではプロセス $P_2$ においてメッセージMを送る以前のメッセージ受信イベントを履歴から探し(図2ではeの地点)、プロセス $P_2$ をeの場所にロールバックさせる。eのメッセージ受信の際にメッセージとともに保存されたSVの値(=< 3,3,4 >)を取り出す。プロセス $P_1, P_3$ については、そのSVの自プロセスに対応する送信イベント番号(プロセス $P_1$ は3、$P_3$ は4)の送信の直後の状態がconsistentな最初の場所であり、その場所にロールバックする。

Procedure 2 ロールバックの手続き

1. メッセージMを指定したとき、受信時の $SV_M$ からMのプロセス $P_i$ における送信イベント番号 $SV_M[i]$ を得る。プロセス $P_i$ の履歴から、番号 $SV_M[i]$ の送信イベント以前で、そのイベントに最も近い受信イベントを探し、その受信イベントを $e$ とする。$e$ と同時に保存されたSVを取り出す。(これを $SV_e$ とする)
2. $P_i$ は $SV_e$ を他のプロセスにブロードキャストし、イベント $e$ 以前に取ったチェックポイントのうち、$e$ に最も近いものを履歴から探してそのチェックポイントを復帰させる。
3. $SV_e$ を受け取ったプロセス $P_k$ は、履歴から $SV_e[k]$ の番号の送信イベント以前で最も近いチェックポイントを探し、その状態を復帰する。
4. すべてのチャネルをクリアする。
5. プロセス $P_i$ はイベント $e$ まで実行する。$P_i$ 以外のプロセスは自分の送信イベント番号の送信イベントまで実行する。各プロセスはメッセージ受信の際には履歴に保存されたメッセージ内容を与えて実行する。送信イベントのときは送るメッセージは捨てる。以上の実行により止まったstate $s = (t_1, t_2, \ldots, t_N)$ の $t_k$ は $k = i$ のときイベントeを実行した直後のstate、そうでないときは $SV_e[k]$ の送信イベントを実行した直後のstateである。
6. プロセス $P_j$ において、5で最後に実行したイベントを $e_j$ とする。$P_j$ の $e_j$ 以降の履歴のうち、$P_k$ からの受信イベントを探し、保存されたSV(これをSV'とする)が $SV'[k] < SV_e[k]$ が成り立つ間はそのメッセージをメッセージキューに戻す。$SV'[k] \geq SV_e[k]$ となった時点で終了する。これを $1 \leq k \leq N (k \neq j)$ について行なう。これらは送信されたが受信されていないメッセージである。

## 4 まとめ

メッセージ流の因果律に基づくデバッグ手法であるcausal debuggingを効果的にサポートするチェックポイント・ロールバックのアルゴリズムを考案し報告した。今後の課題としては、現在のアルゴリズムをプロセス数が動的に変化する場合にも対応させることがある。

### 参考文献


1) 青柳滋己、真鍋義文: "分散プログラミングのためのデバッグ手法の一提案", 第24回情報科学若手の会シンポジウム予稿集(1991-7).

2) S. I. Feldman and C. B. Brown: *"IGOR: A System for Program Debugging via Reversible Execution"*, Proc. of Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices, Vol.24, No.1, pp.112-123, 1989.

3) 真鍋義文、今瀬真: "分散プログラムのデバッグにおける大域的条件について", 信学技報 COMP89-99(1989-12).

4) F.Mattern: *"Virtual Time and Global States of Distributed Systems"*, Proc. Parallel and Distributed Algorithms,Essevier Science Publishers(North-Holland),1988.