巻き戻し実行をサポートする並列プログラムデバッガ

丸山 真佐夫[†,††] 山本 繁弘[††,☆]
大野 和彦[††,☆☆] 中島 浩[††]

巻き戻し実行（あるいは逆実行）は，強力なプログラムデバッギング手法である．しかし，これを並列プログラムのデバッギングに適用するためには実行の非決定性という，並列プログラム特有の問題を解決する必要がある．並列プログラムでは一般に，複数回同じプログラムを実行したとき，それらの実行で同じ振舞いをすることが保証されない．そのため，巻き戻し実行を利用してデバッギングを行う場合，バグの真の原因の追跡に必要なすべての実行状態を保存しておく必要がある．このことは，再現性の低いバグを追っている場合に，特に問題になる．いつどのような条件で出現するか分からないバグを待って，コストの高いチェックポインティングを続けなくてはならないからである．そこで我々は巻き戻し実行に再演手法を組み合わせることで欠点を補う，並列プログラムのデバッギングモデルを提案する．我々は並列言語 Orgel に対して巻き戻し実行機構を実装し，巻き戻し実行を利用可能な並列デバッガ Order を開発した．本論文では，提案手法および Order の実装と評価について述べる．それによって，提案手法を用いて実装した Order システムが強力で使いやすいデバッギング環境を提供できること，現実的なコストで動作することを示す．

A Parallel Program Debugger Supporting Reverse Execution

Masao Maruyama,[†,††] Shigehiro Yamamoto,[††,☆]
Kazuhiko Ohno[††,☆☆] and Hiroshi Nakashima[††]

Reverse execution is a powerful technique of debugging. Some serious problems arise, however, in applying it to parallel programs because of its indeterministic nature. Since a parallel program's behavior can be different from run to run, entire information for locating the bug must be checkpointed in single execution. It causes, in most cases, the users to collect much larger set of information than really needed. It is particularly serious when locating the irreproductive bugs. The users have to run the program repeatedly with costly checkpointing operation until the bug appears. Therefore we propose a debugging model for parallel programs based on checkpointing/rollbacking combined with replay method to overcome these weak points. We applied it to a parallel programming language Orgel, and developed a debugger supporting execution rollback for Orgel programs. Our mechanism works at practical cost in execution, and can provide useful and powerful functions to the parallel debuggers which justify some extra cost.

1. はじめに

巻き戻し実行（あるいは逆実行）は，強力なプログラムデバッギング手法である．この手法の基本的なアイデアは，プログラムの途中のチェックポイントで実行状態のスナップショットを保存しておくことによって，デバッグ中に過去の状態に戻る必要が生じたときに，プログラムを最初から再実行せずにチェックポイント時点から再開するというものである．一般にチェックポインティングはコストの高い操作だが，繰り返しプログラムを再実行する回数が多いなら，そのコストは回収できる．

複数の実行の流れを扱う並列プログラムデバッギングにおいては，過去の状態に戻るという要求は逐次プログラムと比較して大きい．巻き戻し実行は，並列プログラムデバッガにとっても，魅力的である．

しかし，並列プログラムのデバッギングに巻き戻し実行法を適用するには，実行の非決定性という，並列

† 木更津工業高等専門学校
Kisarazu National College of Technology
†† 豊橋技術科学大学工学部
Faculty of Engineering, Toyohashi University of Technology
☆ 現在，日立ソフトウェアエンジニアリング株式会社
Presently with Hitachi Software Engineering
☆☆ 現在，三重大学工学部
Presently with Faculty of Engineering, Mie University

プログラム特有の問題を解決する必要がある．

並列プログラムでは一般に，複数回同じプログラムを実行したとき，同じ振舞いをすることが保証されない．そのため，巻き戻し実行を利用してデバッギングを行う場合，バグの真の原因の追跡に必要なすべての実行状態を保存しておく必要がある．

このことは，再現性の低いバグを追っている場合に，特に問題になる．いつどのような条件で出現するか分からないバグを待って，コストの高いチェックポインティングを続けなくてはならないからである．

並列プログラムの非決定性に対する別のアプローチとして，再演法がある．再演法では，プログラムの振舞いを変化させるイベントの順序を保存・再演することで，プログラム実行の再現性を保証する．イベント順序の保存コストは，チェックポインティングと比較してはるかに低い．

しかし，デバッグ対象プログラムが毎回の実行で同じ振舞いをするというのは，特段のデバッギング支援のない逐次プログラムと同等の状態にすぎない．しかも，複数のプロセスが関連しあう並列プログラムのデバッギングにおいては，単一プロセス内にとどまらず，プロセスをまたがって過去の状態が必要になることも多い．そのために再演法だけを利用したデバッギングでは，逐次プログラム以上に頻繁に，プログラムの再実行が必要になる．

以上の問題を解決するため，巻き戻し法を基礎にしつつ，再演法を組み合わて，全体としてデバッギング作業の効率を高めるためのデバッギング手法を提案する．

我々は，メッセージパッシング型並列言語 Orgel [1] を対象として，提案手法を組み込んだ並列デバッガ Order を開発した．Order は，巻き戻し実行を実用的，効率的に利用するために不可欠な，イベント・チェックポイント操作機能等を備えたデバッギングシステムである．

以下，まず 2 章で提案するデバッギング手法の概要，3 章で実装について述べる．続いて 4 章で，並列デバッガ Order について述べる．次に 5 章で性能評価の結果を示す．6 章では，提案手法の他システム・プログラミングモデルへの適用について検討する．そして 7 章で関連研究について述べ，最後に 8 章でまとめを行う．

2. 提案するデバッギング手法

1 章で述べたように，本システムは巻き戻し実行法に再演法を組み合わせた手法を採用している．これによって，両手法の利点を生かしながら欠点を補い，全体として，デバッギング効率の向上と使いやすい機能を提供することを目標とする．

表 1 各手法のデバッギング作業過程
Table 1 Debugging processes of reverse execution, replay and the proposed methods.

	1 回目の実行	2 回目	≧ 3 回目
巻戻し法	巻戻し保存	リスタート	リスタート
再演法	イベント保存	再演	再演
提案手法	イベント保存	再演・巻戻し保存	リスタート

2.1 巻き戻し法と再演法の融合

表 1 に，従来の巻き戻し法，再演法，そして提案手法のデバッギング作業の流れを示す．従来の巻き戻し法と提案手法の相違点を次に述べる．

従来の巻き戻し法

従来の巻き戻し法では，チェックポインティングを行う最初の実行（巻き戻し情報保存実行）のコストが，手法の有用性を左右する最大の問題である．

チェックポインティングのコストを小さくするためには，チェックポイントの数をできるだけ少なくしなくてはならない．しかし，すでに述べたように，並列プログラムの非決定性に対処するためには，デバッギングに必要なすべての情報を収集する必要がある．通常，バグに関係する部分（たとえばプロセスの集合）を正確に推測することは不可能であるから，安全を見込むなら，真に必要なものより過剰な情報を収集することになる．最悪の場合には，結局待っていたバグが発現せずに，その実行全体が無駄になることもある．

また，チェックポインティング範囲を限定することができる場合にも，その指示を行う方法が問題になる．チェックポインティングを行うために実行の前には，プログラムの動的な情報を利用することができない．そのため，たとえば「このプロセスのこのイベント以降をチェックポインティングする」等の指定は，静的な情報だけを使うものに限定されてしまう．

提案手法

提案手法は，最初の実行（イベント順序保存実行）では再演法と同様に，イベントの発生順序だけを記録する．このときに保存する情報は 1 イベントにつき数バイトであり，巻き戻し情報保存実行と比較して，オーバヘッドは十分に小さい．

2 回目の実行（再演・巻き戻し情報保存実行）では，最初の実行のイベント発生順序を再演しつつ，チェックポインティングを行う．

従来の巻き戻し法と異なるのは，巻き戻し情報保存実行の前に，プログラムの動的な振舞いであるイベン

ト順序を入手していることである．チェックポインティング対象範囲の限定に，これを利用できる．

また，もし対象範囲を絞りすぎて，後に情報の不足が明らかになった場合でも，イベント順序を保存しているので，不足部分を再演実行したり，チェックポインティングの対象を追加したりできる．

さらに 2 回目の実行では，再演法と同様にイベント順序の情報を利用しながら，巻き戻しを含むデバッギング作業を進めることが可能である．ユーザは，通常の実行よりも多く時間がかかること以外，デバッギング作業を制約されることはない．たとえば，プログラムの途中まで実行した状態で，すでに実行済みの時点へ状態を巻き戻すこともできる．

提案手法では，巻き戻し情報保存実行，巻き戻し後の再実行はいずれも，最初のイベント順序保存実行で記録したイベント順序を再演する．したがって，これらの実行でのイベント順序は確定しており，一般の再演法と同様の再現性が保証される．

従来の巻き戻し法と提案手法を比較すると，最初のイベント順序保存実行の分だけ，提案手法の方が余分な時間コストを必要とする．しかし，1 回目の実行に要する時間コストは提案手法の方が小さいので，実質的なデバッギング作業を開始するまでの時間は短くなる．

2.2 イベント同期チェックポインティング

巻き戻しシステムにおいてチェックポイントをとるタイミングとしては，(1) 一定時間間隔で行う，(2) 特定のイベント（メッセージの送受信等）を実行したときに行う，(3) 実行文あるいは指示文によって，ソースコード上に明示する，が代表的である．

一定時間間隔でのチェックポインティングは，障害からの復旧を目的とする場合には，理にかなった方法である．巻き戻しデバッガでもよく採用されているものの，我々は並列プログラムデバッガにおいては，適切ではないと考える．

デバッガのユーザはプログラムの進行を物理的な時間ではなく，「プロセス p で受信イベント e が発生したとき」等を目印としてとらえている場合が多いと考えられる．イベントをチェックポイントにすることで，デバッガユーザは，興味のある実行時点に直接プログラム状態を巻き戻すことができるようになる．

これに対して，一定時間間隔でチェックポインティングを行う場合，ユーザが実際に巻き戻したい点とチェックポイントは一致しない．そのため，ある時点へ戻るには，まず直前のチェックポイントに巻き戻してから，目標点まで順方向にプログラムを実行する必要があり，効率的でない．

ソースコード上に直接チェックポイントを指定する方法は，ユーザの要求とチェックポイントの位置が一致するという点で優れている．しかし，ユーザ自身がソースコードのあちこちにチェックポイントの指定を挿入してまわる繁雑さを考えると，この方法だけを用いるのは現実的でない．

以上から，我々はチェックポイントをプログラム実行中に発生するイベント上に置くことにした．

また，通信等の通常のイベントに加えて，トレーサイベントと呼ぶデバッギングのための特別なイベントを定義した．ソースコード上にトレーサイベントを発生させる関数呼び出しを挿入することで，任意の位置で明示的にチェックポインティングを行うことができる．これは，上の（3）と同等の機能を提供するものであり，注目箇所があらかじめ明確になっている場合や，プログラム本来のイベントの間隔が長い場合等に有用である．

3. 並列言語 Orgel への巻き戻し機構の実装

本章では，まず提案デバッギングシステムの実装対象とした並列言語 Orgel の概略を示し，次に提案する巻き戻し機構の実装について述べる．

3.1 並列言語 Orgel

Orgel は現在我々が開発を行っている，非数値処理的な分野を主要な対象とした並列プログラミング言語である[1)]．

Orgel ではエージェントと呼ばれる並列実行単位が，ストリームと呼ばれる抽象的な通信路を介してメッセージを送受信しながら動作する．ストリームは方向を持ち，両端（入力側，出力側）に 1 個以上のエージェントが接続される．エージェントは内部状態を表すエージェントメンバ変数（AMV），自身の処理を行うコード，受信メッセージへの応答コードからなる．Orgel の文法は，C に対してストリーム，メッセージ，エージェントの操作を追加したものになっている．

Orgel は共有メモリ型，分散メモリ型の並列計算機で動作するバージョンが開発されており，本システムは現在，共有メモリ版 Orgel 処理系を対象に実装されている．ただし，本提案の巻き戻し機構は，適用対象を共有メモリシステムに限るわけではない．

Orgel コンパイラは，Orgel プログラムを C プログラムへ変換するトランスレータとして実装されている．変換された C プログラムと Orgel 独自の機能を実現するランタイムライブラリを結合して，実行コードを生成する．

共有メモリ版 Orgel では，エージェントの並行/並

```
 1:  stream data { value(int v:in); end; }
2:  agent main(void);
3:  agent worker(data i:in, data o:out);
4:  agent main(void){
5:    worker w1, w2;
6:    data d1, d2;
7:    connect d1 <== self;
8:    connect w1.o ==> d1 ==> w2.i;
9:    connect w2.o ==> d2 ==> w1.i;
10:    task {
11:      d1 <== value(0);
12:      terminate;
13:    }
14:  }
15:  agent worker(data i:in, data o:out){
16:    dispatch (i) {
17:      value(v) :
18:        if (v >= 100) {
19:          o <== end;
20:          terminate;
21:        } else {
22:          tracer(v);
23:          o <== value(v+1);
24:        }
25:      end:  terminate;
26:    }
27:  }
```

行番号は説明のためのもので，プログラムの一部ではない

図 1 Orgel プログラムの例（一部）
Fig. 1 An example of Orgel program.

列実行には POSIX スレッドを利用し，スレッド間で共有するメモリ空間を用いて通信を行う．AMV は C の構造体として表現され，エージェントを実行するスレッドのスタックの底に置かれる．

Orgel プログラムの例を**図 1** に示す．このプログラムは，`worker` 型の 2 つのエージェント `w1`，`w2` の間で整数メッセージを往復させる．

3.2 巻き戻し実行機構の実装概要

イベント順序，巻き戻し情報の保存等の本システムの基本的な機能は，Orgel コンパイラに対する修正と，デバッギング機能を付加した特別なランタイムライブラリとして実装されている．

本システムを用いたデバッギング過程は，

(1) イベント順序保存実行
(2) 巻き戻し情報保存実行（巻き戻し保存実行）
(3) 巻き戻し
(4) （巻き戻し後の）再実行

からなる．(1) で再演に必要なイベントの発生順序を保存し，(2) では (1) のログに従って再演実行しつつ，エージェントの状態を記録する．

実装上 (2)，(4) は区別がない．巻き戻しに関係するイベントを初めて実行したときには必要な情報を保存し，2 回目以降の実行では，保存された情報を利用するといった処理が内部的に行われる．

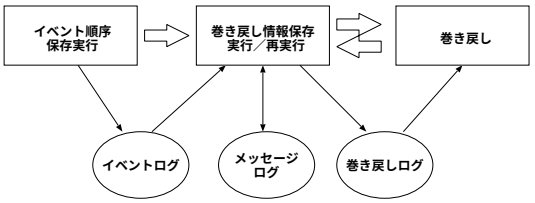


図 2 デバッグ過程とログの記録・参照
Fig. 2 Execution phases and the log generation/reference.

(1) のイベント順序保存実行（以下イベント保存実行）は，被デバッグプログラム単体で実行可能である．それに対して (2)〜(4) は，通常デバッガの制御下で実行する．(1) を単体で動作させられるので，デバッグ版のプログラムを通常と同じように動かしておいて，バグが発見されたときにはじめてデバッガにかけるという手順がとれる．

上の過程で保存，参照される情報は，イベントの発生順序を記録するイベントログ，メッセージの内容を保存するメッセージログ，受信イベント時点のエージェントの状態を記録する巻き戻しログからなる．各過程と記録，参照するログの関係を**図 2** に示す．

3.3 イベント順序保存実行

イベント順序保存実行では，被デバッグプログラムを決定的に再演するために必要な情報を，イベントログに記録する．ロギングは各エージェントと各ストリームで独立に行われ，ログファイルも個別に生成される．

入力ストリームから同じメッセージを同じ順序で読んだ場合，Orgel のエージェントの動作は決定的である．したがって，各エージェントが受信したメッセージ列の順序を保存すれば十分であるが，デバッギングの利便性のために，冗長な情報を保存している．

各エージェントで記録されるエージェントのログは，各イベントについてイベント種類，メッセージ識別子，ストリーム識別子からなり，8 バイトを占める☆．

表 2 に，本システムで定義するエージェント上のイベント一覧を示す．

再演のために記録する必要のあるイベントとして，メッセージの送信（図 1：11，19，23 行）・受信（図 1：16 行）等，エージェント間通信に関連するものが定義されている．

メッセージ受信には，ブロッキング受信，非ブロッキ

☆ 各データのビット数を削って記録している．内部表現では 20 バイトを占める．またオーバヘッドの増加とひきかえにメッセージの型やイベントの発生時刻を付加できる．

表 2 エージェント上のイベントの種類
Table 2 Agent events.

イベント名	意味
CREA	エージェント生成
TERM	エージェント終了
CRES	ストリーム生成
SEND	メッセージ送信
RECV	ブロッキングメッセージ受信
NBRF	非ブロッキング受信失敗
NBRS	非ブロッキング受信成功
TRAC	トレーサイベント

ング受信の 2 種類がある．受信メッセージがないときに実行すべき処理を記述する `task` ハンドラが定義されていないエージェントでは，Orgel のメッセージ受信 `dispatch` はブロッキング動作になる．一方，`task` ハンドラ（図 1：10 行）のあるエージェントでは，非ブロッキング受信が実行される．非ブロッキング受信では，受信の成功または失敗に従って異なるイベント（`NBRS` または `NBRF`）をイベントログに記録する．

再演に必要でないイベントの主要な目的は，デバッガユーザにヒントを与えることである．たとえばエージェントの生成や終了（図 1：12，25 行）等エージェントの動作の節目を記録するイベント，ユーザプログラムからの関数呼び出しによって自由に挿入できるトレーサイベント（図 1：22 行）が定義されている．

トレーサイベントを発生させる関数は 1 個の整数型引数をとる．この値はイベントログに記録されるので，たとえば「変数 x が負になった時点」をイベントログから抽出する等に利用することができる．

ストリームでは，ストリームに到着するメッセージごとにメッセージ識別子（8 バイト）を記録する．

メッセージ識別子は，そのメッセージを生成したエージェントの識別子と，エージェント内でのメッセージ生成番号からなる．同様にストリーム識別子は，エージェント識別子とストリームの生成番号からなる．

3.4 巻き戻し情報保存実行

巻き戻し情報保存実行（巻き戻し保存実行）では，イベントログに従って再演実行をしつつ，各エージェントの状態を巻き戻すための情報を保存する．

3.4.1 チェックポインティング対象

任意のエージェントイベントでチェックポインティングすることが可能であり，ユーザはデバッガを通じてチェックポイントを設定する．各チェックポイントでは，その時点のエージェント状態が，巻き戻しログに保存される．イベントログと同様，巻き戻しログもエージェントごとに独立している．

チェックポイントで保存すべきエージェント状態は，(1) CPU レジスタ，(2) AMV，(3) エージェントが使用しているヒープ，(3) エージェントを実行するスレッドのスタック，である．

対象にした Orgel の実装では，すべての受信イベントは関数呼び出しの入れ子構造における最も外側に位置する関数の，特定の文で実行される．そのため，メッセージ受信時には，スタック上に保存を必要とするデータが残っていない（スタックの底にあって，エージェント生存中保持され続ける AMV を除く）．さらに，各チェックポイントごとに CPU レジスタの保存する必要がない．この性質を利用して，メッセージ受信イベントでは，AMV とヒープだけを保存する．

3.4.2 チェックポインティング手法

チェックポイントにおける対象データの保存手法として，対象データ全体を保存するフルチェックポインティング，前回チェックポイント以降に書き込みのあったページだけを保存するインクリメンタルチェックポインティング，前回からの差分を保存する差分チェックポインティングが一般的に用いられている[2]．

インクリメンタルチェックポインティング，差分チェックポインティングでは，OS の提供するメモリ保護機構を利用して書き込みページを検出することが多い．

チェックポインティング手法の主要な評価基準は，実行時のオーバヘッド，巻き戻しに要する時間，生成されるログ量である．特にデバッギングを目的とするチェックポインティングでは，古い時刻のログデータも破棄できないという制約があるため，ログ量を小さくすることが重要になる．

どの手法が適切であるかは，チェックポインティング対象メモリへのアクセス傾向と評価基準によって異なる．たとえばインクリメンタルチェックポインティングはメモリアクセスの局所性を前提にしているので，これが成立しない場合はログ量を削減する効果が得られない．一方，差分チェックポインティングは，多くの場合他の 2 手法と比較してログ量を小さくできるが，実行時のオーバヘッドが大きい．そこで我々は，AMV，ヒープ，スタックそれぞれ対して異なる保存手法を適用できるようにした．

スタックには，フルチェックポインティングを適用する．Orgel では，受信イベントごとにスタックがいったん空になる．そのため，連続する 2 つのチェックポイントはまったく異なる呼び出し系列を積んでいる可能性が高い．このことから，インクリメンタル/差分チェックポインティングは適さないと考えた．

一方 AMV，ヒープに対しては，ログ量を削減するため，差分チェックポインティングを行う．ただし，差

分チェックポインティングを行うと，メモリの状態を回復する際にログの先頭から順に差分を適用する必要がある．そのため，時刻の大きなチェックポイントに巻き戻すほど時間がかかる．この問題は，適当な間隔でフルチェックポインティングを行うことで解決する．これによって，ログの先頭ではなく最近のフルチェックポイントからの差分適用ですむようになる．

現在，フルチェックポインティングのタイミングを決定する方法として，次の 2 つを提供している．

累積ログ量による方法

前回フルチェックポインティング以降の差分ログの累積量が指定された値を超えると，自動的にフルチェックポインティングを行う．ユーザは，累積ログ量のしきい値を変化させることで，巻き戻しに要する時間と巻き戻し情報保存実行の時間・ログ量のトレードオフをコントロールできる．

ユーザの直接的な指示による方法

設定ファイル上に指定されたイベントでフルチェックポインティングを行う．この指定は，独立したツールやデバッガ上で行うことを想定している．

上の累積ログ量のしきい値による方法と併用できるので，たとえばプログラム全体に対してやや大きなしきい値を指定しておいて，特定タイプのイベントや注目する実行区間では密にフルチェックポインティングするように直接指定する，等が可能である．

3.4.3 ヒープ管理

Orgel エージェントは，スレッドによって実現されている．そのため，ヒープ領域は同一プロセス内で実行される他のエージェントと共有する．オリジナルの Orgel 処理系では，動的メモリの獲得/解放にシステムの提供する `malloc()`/`free()` をそのまま利用できるが，巻き戻しを行う場合には，問題が発生する．

たとえば，あるエージェント a がチェックポイント c を通過後に `malloc()` でアドレス p を獲得し，その後 p を解放したとする．エージェント a をチェックポイント c に巻き戻して再実行させた場合，(a が p を解放した後の時刻にいる）他のエージェントがアドレス p を利用中であるために，同じ `malloc()` で p を獲得できない可能性がある．

この問題を解決するため，動的メモリに関して次の実装によって，ヒープ領域を各エージェントの占有資源として扱うことにした．

各エージェントは `malloc()` で割り当てるための独自の領域を持つ．ユーザプログラム中の `malloc()`/`free()` を，コンパイル時にこの独自領域を利用する関数に置き換える．独自領域はシステムの提供する `malloc()` を利用して確保する．領域が足りなくなったときには，エージェントは新たに確保してつぎ足すが，不要になった場合でも，システムには返却しない．ただし，各エージェントの内部では，ユーザプログラムが解放した領域を使い捨てるわけではなく，以後のメモリ割当てに再利用する．つまり，各エージェントは `malloc()` で確保したピークメモリ量に相当する領域をプロセス終了時点で占有し，各エージェントの占有領域の和がプロセス全体のヒープ使用量になる．

3.4.4 メッセージの保存

受信イベントを含む部分を再実行する場合，送信側も同時に再実行して実際に送信させるか，あるいは巻き戻し機構がメッセージを保存しておいて供給しなくてはならない．

オリジナルの Orgel 処理系では，メッセージを専用のヒープ領域に確保して利用し，使い終わった領域をシステムがガーベジコレクションによって回収する．巻き戻し機構の現在の実装では，このガーベジコレクションを停止することで，全メッセージを保存する．

本実装は，巻き戻し保存実行時のオーバヘッドを小さくすることができる一方，メッセージを主記憶上に保存しているため，送受信するメッセージが大量になると，メモリ不足を起こす可能性がある．今後，メッセージをディスクに退避できるように改良する予定である．

3.5 巻き戻し

エージェントの巻き戻しは，シグナルを利用して実装されている．Pthread ではプロセス外から特定のスレッドにシグナルを送れないため，次のように 2 段階に分けて巻き戻し要求を届けることにした．まず，デバッガ版 Orgel ランタイムライブラリ中で生成するデバッグ支援のためのスレッド（デバッグスレッド）が，ソケットを利用して外部（通常はデバッガ）からの巻き戻し要求を受け取る．そして，デバッグスレッドが `pthread_kill()` を用いて巻き戻し対象のスレッドにシグナルを送る．

3.6 巻き戻し後の再実行

上に述べたように，巻き戻し後の再実行は実装上巻き戻し保存実行と区別がない．たとえばチェックポイントに到達したときに，まだ巻き戻しログが保存されていなければチェックポインティングを実行し，すでに巻き戻しログが保存されていればなにもしない，等の処理が内部的に選択される．

したがって，巻き戻し・再実行をはじめて行う前に，プログラムの最後まで巻き戻し情報を保存する必要は

ない．あるエージェントの実行を全体の半分あたりまで実行したところで停止し，先頭から 4 分の 1 経過したチェックポイントへ巻き戻して，そこから再実行するというような操作も可能である．この場合でも，プログラムはイベントログに従って実行されているので，再現性が損なわれることはない．

4. 巻き戻しデバッガ Order

我々は，前章までに述べた巻き戻し機構を用いて並列言語 Orgel のためのデバッガ Order を実装した．

4.1 Order の構成

Order は，ユーザとの対話を行うフロントエンド部と，被デバッグプログラムを操作する逐次デバッガ（現在は GNU Debugger（gdb）を利用），イベントログの処理のために呼び出す Prolog 言語処理系からなる．Order の持つ基本的な機能は，フロントエンドが gdb を制御して間接的に被デバッグプログラムを操作することで実現されている．巻き戻し等いくつかの操作では，フロントエンド部が被デバッグプログラム内のデバッグスレッドと直接に通信する．

4.2 イベント・チェックポイントの操作

すでに述べたように Order では，イベントログ上に巻き戻しのためのチェックポイントを設定する．イベントログに記録されるイベント数は容易に数万あるいはそれ以上の規模になりうるので，イベント・チェックポイント操作の機能がデバッガの使いやすさにとって特に重要である．

Order は，イベントログの操作に関して次の 2 種類の方法を提供する．

(1) イベントのフィールドに対する条件によるフィルタリング
(2) 発生順序など複数イベント間の関係を含む複雑な条件の記述

それぞれの条件を記述する式は，イベント・チェックポイントを操作するコマンドの引数にすることもでき，あるいは変数に代入することもできる．式・変数に対する集合演算（AND，OR）も可能である．

さらに，ユーザによる特別な条件の追加を容易にするために，イベント操作部分をデバッガ本体から独立させて Prolog で記述している．

なお現在の Order は，ここで述べる条件の中でイベントログの参照だけを許し，Igor [3] が提供しているような実行時の変数値の参照をサポートしていない．しかし，巻き戻し情報から各実行時点の特定の変数の値を再現することは可能なので，容易に Igor と同様の機能を実装できる．

```
1:  $1 = agent(*,Amaster)
2:  $2 = agent_event(#1,*,send,*,Mp|*)
3:  $3 = {E|nth(30, $2, V), after(V, E)}
4:  $4 = {E|nth(31, $2, V), before(V, E)}
5:  $5 = $3 AND $4
6:  setcp $5
7:  run
8:  rollback $2, 31
...
```

行番号は説明のためのもので，入力の一部ではない

図 3 イベント・チェックポイント操作の例
Fig. 3 An example of debugging sequence.

表 3 マッチング条件
Table 3 Provided matching operations.

表現	意味
n_1:n_2	整数 n_1〜n_2 にマッチ（n_1，n_2 は省略可）
$\{e_1,e_2,\cdots\}$	e_1，e_2，$\cdots$ のいずれかにマッチ
$\#v$	集合変数 v のメンバのいずれかにマッチ
*(または_)	任意の値にマッチ
	特に |*は以降の値全体にマッチ

4.2.1 フィルタリング式

フィルタリング条件は，**図 3** の 1・2 行目のように

対象の種類（フィールド 1 の条件，$\cdots$）

の式で指定し，値はマッチした対象の集合になる．

指定できる対象の種類には，エージェント（`agent`），エージェント上のイベント（`agent_event`），ストリーム（`stream`），ストリーム上のイベント（`stream_event`）がある．`agent_event`，`stream_event` を構成するフィールドは，3.3 節で述べたイベントログの内容である．`agent`，`stream` は，対象を識別する ID，型，さらに `stream` では接続されるエージェントのリストからなる．

各フィールドに対して記述できるマッチング条件を**表 3** に示す．

4.2.2 複雑な条件の記述

フィールドに対する単純なマッチングで表現できない条件は，次の式で指定する．

{ *Var*| *Var* を含む述語 }

'|' の右側の記述を満足する *Var* の集合が返される．述語部分はほぼ Prolog の文法に沿うが，Order の変数（$v または #$v$）を書くことができる．$v は集合変数そのものであり，Prolog に対してはリストが渡される．#v は集合 v のメンバのいずれかを意味し，v のメンバのうちの 1 つ以上に対して述語が満足される集合が返される．

現在，Order にはイベント間の半順序関係を扱う述語（例として図 3 の 3・4 行目の `after`，`before`）等が組み込まれており，さらにユーザが述語を追加できる．

4.2.3 実行時に決定する識別子の扱い

Orderで扱うイベントログデータのうち，型名はデバッグ対象プログラムのソースコード上で決定している．一方，エージェントやストリームを識別するIDは，静的には決まらない．本項では，これらのIDの表現とOrder上での指定方法について述べる．

Orgelにおいてエージェントやストリームは，変数と同様の形式で宣言的に記述される（たとえば図1：5〜6行）．エージェントのストリームへの接続も，実行時の変数を含まない形で記述される（同7〜9行）．つまり，エージェントのネットワークの構造は，静的に決定していると考えることができる☆．そのため，エージェントの生成—被生成の親子関係はデバッグ上きわめて有用な情報である．

そこで本システムでは，次のように，エージェント間の親子関係をエージェントIDとして保持することにした．まず，どのOrgelプログラムにも必ず1つだけ存在する`main`エージェントのIDを0とする．そして，IDがxであるエージェントが生成したn番目のエージェントのIDを$x.n$とする．たとえば，`main`エージェントが3番目（0番目から数えて）に生成したエージェントの4番目の子エージェントのIDは0.3.4になる☆☆．エージェント内での子エージェントの生成順序はソースコード上での記述順に従うので，ユーザはこのIDを用いることで，動的に生成されるエージェントの実体とソースコードを対応付けられる．

Order上ではPrologのリスト表現を利用し，たとえば0.3.4は`[0, 3, 4]`と表す．またフィルタリング式中では，「IDが0.4のエージェントが生成した子エージェントのうちの最初の3つ」を`[0, 4, 0:2]`と表現することができる．

ストリームIDもエージェントIDと同様の考え方で，エージェントxが生成したn番目のストリームを[x，n]で表す．

4.2.4 イベント・チェックポイント操作の例

前項までで述べた機能を使って，イベントやチェックポイントを操作する例を示す．図3は，Othelloを自動対局するOrgelプログラムについて，30手目が誤っているという問題をデバッグする場合の，ユーザのOrder上でのコマンド入力列である．

第1行は，“IDが任意，エージェント型名‘`Amaster`’であるエージェント”という条件でマスタエージェントを求め，変数`$1`に代入している（このプログラムでは，‘`Amaster`’型エージェントは1つしか存在しない）．

2〜5行で30手目に関連するイベントの集合を得る．まず第2行は，マスタエージェントにおける（‘`#1`’），任意のイベント番号（‘`*`’）の送信イベント（‘`send`’）で，送るメッセージのIDが任意（‘`*`’），型が‘`Mp`’（局面）であるイベント集合を得ている．3行目では，イベント集合`$2`の30番目の要素`V`（‘`nth(30,$2,V)`’），すなわち30手目に対して，`V`より後（‘`after`’）に起こるイベント`E`の集合を計算する．4行目では逆に31手目より前のイベント集合を得る．最終的に5行目で`$3`，`$4`の積をとることによって，30手目に関連するイベントを絞り込んでいる．

6行目の‘`setcp`’コマンドによって，このイベント集合をチェックポイントに指定し，‘`run`’コマンドでプログラムを巻き戻し保存実行する（7行目）．

これ以降，巻き戻し（‘`rollback`’）コマンドを利用できる．8行目では，マスタエージェントを`$2`の31番目のイベントに巻き戻している．

‘`rollback`’が指定された単一のエージェントを巻き戻すのに対して‘`rollmin`’，‘`rollmax`’は，全エージェントをいっせいに巻き戻す機能を提供する．‘`rollmin`’は，指定されたイベント（チェックポイントである必要はない）を実行するために各エージェントが最低どの時点まで実行する必要があるかを計算して，その直前のチェックポイントへ巻き戻す．これは，三栄らが提案した“最小限の再演実行”[4]に近い機能を提供する．

たとえば**図4**において，Rで示したイベントへの`rollmin`を行うと，エージェント1，エージェント2もrへ巻き戻される．本機能は，メッセージによってエージェントを越えて運ばれるバグの原因をたどる際に有効である．

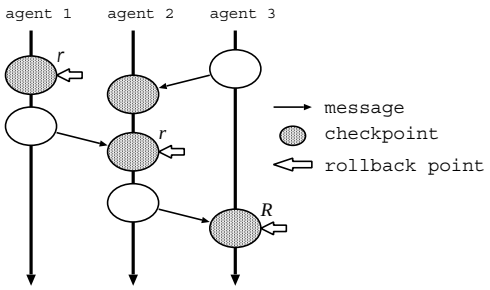


図4 rollminによる巻き戻しの例
Fig. 4 An example of rollmin operation.

☆ ただし，エージェントの再帰的な宣言を許すので，実際に生成される数は実行時まで分からない．

☆☆ イベントログのファイル中では，レコードを短く，かつ固定長にするために上のIDと1対1に対応させた整数値を記録している．

`rollmax` は `rollmin` とは逆に，指定されたイベントまでを実行したときに，他のエージェントが実行可能な最大限の時点を計算して，その直前のチェックポイントへ巻き戻すコマンドである．

以上のように，Orderではイベントログを利用することによって，チェックポイント対象範囲の絞り込み等を，簡潔に記述することが可能である．また，`'before'`，`'after'` のような，エージェントをまたがる順序関係を含む場合でも，巻き戻し保存実行の前にチェックポイント対象イベントが決まるため，実行時間のオーバヘッドを増加させない点でもすぐれている．

Order と同等のイベント・チェックポイント操作機能を，従来の巻き戻し法で実現するのは困難である．たとえば，“ある型のメッセージを送信する直前の受信イベント “は，Order 上では容易に決定できる．ところが従来の巻き戻し法では，未来に発生するイベントの予測が要求されることになってしまう．

5. 性能評価

5.1 基本的なオーバヘッド

単純なプログラムを用いて，イベント保存実行，巻き戻し保存実行のオーバヘッドを測定した．測定は，単一プロセッサ構成の PC（Pentium 4 1.6 GHz，主記憶 512 MB，FreeBSD 4.6R）で行った．

測定に用いたのは，2 つのエージェント `w1`，`w2` が，互いに整数を送受信しあうプログラム（pingpong）である．`w1`，`w2` は受け取った数に 1 を加えた値（初期値は 0）を相手に送り，値が N になった時点で終了する．

本プログラムは，エージェント内部の処理がほとんど存在しない．そのため，イベント保存実行に関しては，オーバヘッドの割合が最も高くなるケースになる．

N=100 万の場合について，デバッギングコードを含まない通常実行，イベント保存実行，差分保存による巻き戻し保存実行（全受信イベントでチェックポインティング，ログ 10 MB ごとにフルチェックポインティング），先頭に巻き戻した後の再実行の実行時間を測定した．Orgel 自体のオーバヘッドと比較するために，Pthread を直接利用して同等の処理を行う C プログラムでの実行時間も測定した．

表 4 に結果を示す．上に述べたように，イベント保存実行のオーバヘッドが最大になる条件で，通常実行に対する実行時間の増加率が 18%程度という良好な結果が得られた．

巻き戻し保存実行時間は，通常実行に対して 6.3 倍と，やや大きなオーバヘッドになった．本プログラムは 4 バイトの AMV に対して 1 ページ分のメモリ保護機構が働いていることになるため，相対的にこのオーバヘッドが大きくなる．その意味では，巻き戻し保存実行についても，最悪の条件を測定しているといえる．AMV のサイズが大きくなるにつれて，1 バイトあたりのメモリ保護のオーバヘッドは小さくなるので，現実のプログラムでは問題にならないと考える．

表 4 pingpong（N=100 万）の実行時間とログ量
Table 4 Execution time and log amount of pingpong program.

	実行時間（s）	時間比	ログ量 ($\times 10^6$B)
通常実行	11.6	1.00	—
イベント保存	13.6	1.18	24.0
巻き戻し保存	72.8	6.28	51.0
再実行	7.11	0.61	—
再実行（単一エージェント）	3.65	0.31	—
C	3.98	0.34	—

全エージェントを最初のイベントまで巻き戻した後の再実行時間は，通常実行と比較して 61%であった．通常実行よりも高速になるのは，3.6 節で述べたように，再実行時にはメッセージ送信を行わないためである．

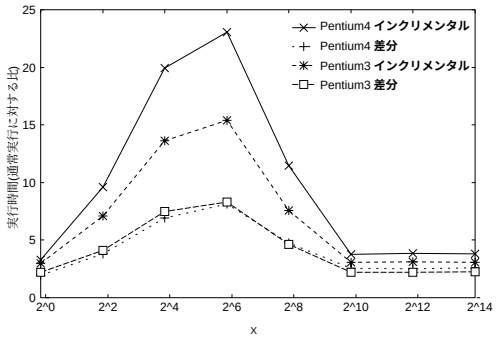
さらに，このプログラムの本来の動作では，2 つのエージェントが交互にしか動けないのに対して，再実行時は，エージェントがメッセージを待たずに実行を継続できる．そのため，メッセージを往復させている一方のエージェントだけを再実行した場合，実行時間はさらに半分程度に短縮される．

巻き戻し保存実行のオーバヘッドが相対的に大きい場合，コストを回収できるのは，巻き戻しが何回か起こるときに限られる．しかし，本実行例のようにメッセージ待ち等のブロッキング時間が長いプログラムや，プロセッサ数がエージェント数より少ない環境では，再実行の時間が通常実行よりも短縮される点で，再演法よりも優れている．

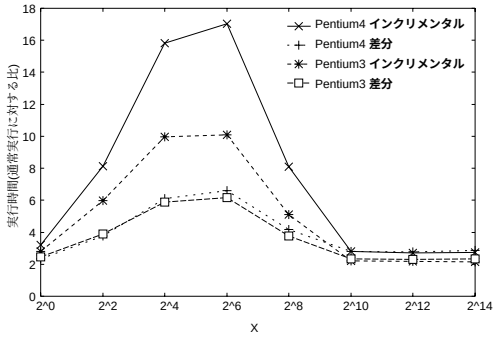
5.2 チェックポインティング手法の比較

メモリ書き込みの局所性を変化させた場合の，各チェックポインティング手法の振舞いを測定した．測定に用いたプログラムの基本的な動作は同じく pingpong（N=10000）である．

`w1`，`w2` の AMV に，ページ境界に整合させて配置した Y 行 X 列のサイズの 2 次元整数配列（`a[Y][X]`）を追加し，メッセージ受信のたびに各行について確率 P_y で書き込みを行う．書き込みを行う行では，各要素を P_x の確率でランダムに書き換える．配列の全体のサイズは一定（2^{17} 要素，512 KB）とし，Y，X の



P_y=0.1, P_x=0.1



P_y=0.1, P_x=1.0

図 5 各チェックポインティング手法の実行時間
Fig. 5 Execution time of incremental and differential checkpointing.

表 5 各チェックポインティング手法のログ量
Table 5 Amount of log data generated by incremental and differential checkpointing.

X	P_y=0.1, P_x=0.1 I($\times 10^9$B)	P_y=0.1, P_x=0.1 D($\times 10^9$B)	P_y=0.1, P_x=1.0 I($\times 10^9$B)	P_y=0.1, P_x=1.0 D($\times 10^9$B)
2^0	3.44	0.40	5.33	0.79
2^2	3.42	0.40	5.33	0.63
2^4	3.33	0.39	5.33	0.60
2^6	2.93	0.35	4.36	0.58
2^8	1.77	0.25	1.89	0.56
2^{10}	0.61	0.14	0.61	0.55
2^{12}	0.60	0.14	0.61	0.55
2^{14}	0.59	0.14	0.60	0.56

I，D はそれぞれインクリメンタル，差分を表す．

組合せを変化させた．

本実験では，CPU がより低速な計算機（PentiumIII 800 MHz，主記憶 256 MB）を用いた測定も行った．**図 5**，**表 5** に，$P_y = 0.1/P_x = 0.1$ と $P_y = 0.1/P_x = 1.0$ の条件で全受信イベントでチェックポインティングを行った場合の実行時間（通常実行に対する相対値），巻き戻しログサイズを示す．

図 5 から分かるとおり，いずれの CPU でも差分チェックポインティングは同等またはより良い結果が得られ，特に Pentium 4 の方が両手法の差が大きい．これは，CPU が高速になるにつれて，差分を生成するコストよりもログをディスクに書き込むコストの方が大きくなるからであると考えられる．

差分チェックポインティングでは，局所性（すなわち X の大きさ）の変化に対して安定しているのに対して，インクリメンタルチェックポインティングでは，X がページサイズ（X= 2^{10}）より小さくなると急激にログ量が増加する．

なお，書き換わる総量が同じでも，X が小さいほど差分チェックポインティングのログ量が増加するのは，2 つのデータをバイトごとに比較していったとき，一致する区間，不一致区間の交代が頻繁に起こりやすくなるためである．現在のアルゴリズムでは，区間ごとに 1 バイトの区間長データが付加される．

これらの結果から，差分チェックポインティングはインクリメンタルチェックポインティングに対して同等以上の性能になると考える．特に Orgel が対象とする非数値計算的な応用では，数値計算ほど局所性が高いと期待できないので，差分チェックポインティングの方が安全である．

表 6 フルチェックポイント間隔と巻き戻しコスト
Table 6 Interval of full checkpoints and the cost of rollback operation.

フル CP 間隔 ($\times 10^6$B)	巻き戻し保存実行時間 (s)	ログ量 ($\times 10^6$B)	巻き戻し時間 (s)
1	73.2	51.2	0.175
10	72.8	51.0	1.82
100	72.7	51.0	4.50

5.3 巻き戻しコスト

差分チェックポインティング手法における平均巻き戻しコストは，フルチェックポイントの間隔が短いほど小さくなる．一方，フルチェックポインティングを頻繁に行うと，巻き戻しログ保存のコストが大きくなる．

本節では，フルチェックポイントの間隔を変化させた場合の巻き戻し保存実行のコストと，巻き戻しに要する時間の測定について述べる．

表 6 は，5.1 節と同じ環境での pingpong プログラム実行結果である．表中の巻き戻し時間は，フルチェックポイントから最も遠い差分チェックポイントへの巻き戻しに要する時間である．

累積ログ量 1 MB ごとにに 1 度フルチェックポインティングを行うと，巻き戻し時間の上限を 0.2 秒におさえることができる．このとき，途中でフルチェックポインティングを行わない場合に対する実行時間と巻き戻しログ量の増加は，いずれも 1%以下である．

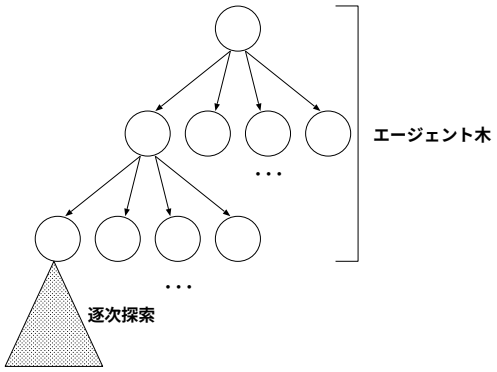


図 6 othello のエージェント木構造
Fig. 6 Agent tree of the othello playing program.

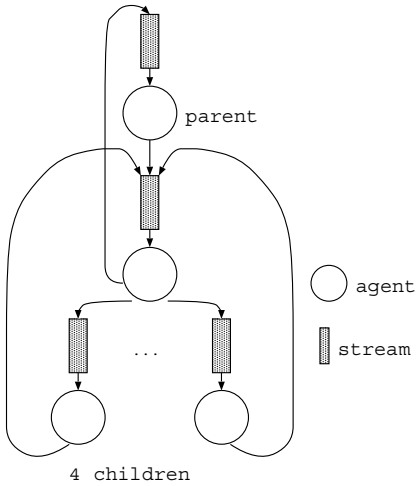


図 7 othello のエージェント接続
Fig. 7 Connection between the agents.

5.4 othello の逐次・並列実行の性能測定

オセロをプレイするプログラム othello を用いて，より現実的なプログラムの実行性能を測定した．

本プログラムは，探索エージェントを深さ 3 の 4 分木状にストリームで接続し，ゲーム木を並列 $\alpha\beta$ 探索する．木の幅と深さには，葉エージェントの数が後述する並列計算機のプロセッサ数に近くなる値を選んだ．

葉以外の探索エージェントは，親から受信した局面に対する次局面の集合を生成し，各子エージェントに 1 つずつ送る．Orgel では，いったんストリームに流したメッセージをキャンセルできない．そこで，無駄な局面の探索を避けるため，子エージェントからの解を受け取るたびに，まだ自身の解が得られていないことを確認してから，別の次局面を送るようにした．

葉エージェントは，$\alpha\beta$ 法による逐次探索を行う．今回の測定では，逐次部分の深さを 3 とした．

エージェント木の全体構造を**図 6** に，また 1 つのエージェントについての，より詳細な接続を**図 7** に示す．

表 7 othello の逐次実行結果
Table 7 Result of sequential execution of othello program.

実行	実行時間	ログ量 (B)	総メッセージ量 (B)
通常実行	7.90	—	5.41×10^5
イベント保存	8.01	1.77×10^5	5.41×10^5
巻き戻し保存	8.32	6.73×10^5	5.41×10^5

表 8 othello の並列実行結果
Table 8 Result of parallel execution of othello program.

	時間 t(s)	評価局面数 p	t/p*
通常実行	31.8	2.06×10^5	1.00
イベント順序保存実行	31.0	1.82×10^5	1.10
巻き戻し情報保存実行	33.2	1.82×10^5	1.17

*通常実行を 1 として正規化

5.1 節と同じ環境での実行結果を，**表 7** に示す．本測定でも，全受信イベントで巻き戻し情報を保存した．このように，通信と計算の比率がより現実的なプログラムでは，巻き戻し保存実行の速度低下は小さくなる．なお，本プログラムで発生したメッセージ数は約 1.5 万であり，全メッセージを保存するのに必要なヒープ領域は 540 KB だった．この程度のメッセージ量の問題に対しては，主記憶上に保存する現在の方法が適用可能であることが分かった．

最後に 20 プロセッサ構成の SPARC Center-2000（SuperSparc，主記憶 1,280 MB，SunOS 5.6）を用いた並列実行の結果を**表 8** に示す．

本プログラムの動作は非決定的であるため，プログラムをそれぞれ 100 回して，評価した局面あたりの時間を比較した．通常実行に対して，イベント順序保存実行で 1.10 倍，巻き戻し保存実行で 1.17 倍という結果が得られ，並列計算機環境でも本システムが良好な性能で動作することが確認できた．

この例のように，メモリの使用量が小さい場合，提案手法の効果は特に高い．再演法と比較すると，巻き戻し保存実行に要する 20%未満の時間コストとひきかえに，以後任意の実行時点に移動することが可能になる．このコストは，多くの場合 1 回の巻き戻しで回収できるので，それ以降の巻き戻しは，すべて時間短縮に貢献する．

6. 他システムへの適用

本手法を他のシステム・プログラミングモデルに適用するためには，

(1) デバッグ対象プログラムの決定的な再演が可能

であること
(2) 並列実行単位（本システムではエージェント）を独立に巻き戻し・再実行可能であること
が必要である．

(1)は，通常の再演型デバッギング手法に求められるのと同様の要件である．MPI等のメッセージパッシングシステムでは，非決定性の原因になるイベントが明確であるため，比較的容易に効率的な再演が実現できる（たとえば文献5)）．

(2)に関しては，他の実行単位からの独立，および通信ライブラリ等からの独立が問題になる．

すでに述べたように，Orgelのエージェントの状態は，AMVとスタックだけで表現される．そのため，任意のエージェントを，巻き戻し保存実行中を含む任意の時点で，他のエージェントや通信状態から独立に巻き戻すことができる．

一方，本手法を一般のシステムに適用する場合には，巻き戻し対象とそれ以外の要素を分離する必要が生じることがある．これは，メッセージパッシングモデルの場合には比較的簡単な実装上の工夫で解決できると考えられる．たとえばMPIを利用するシステムでは，巻き戻し対象プロセスの状態から通信ライブラリの状態を分離して巻き戻し後の通信状態の整合性を保つ必要があるが，MPI内の通信バッファ等を巻き戻しシステムが管理するようにMPI関数を修正すれば，本手法を適用することができる．

共有メモリモデル，特にメモリアクセスを介したスレッド間の通信が任意に生じるような一般化されたモデルの場合は，各並列実行単位の独立性が低く本手法を単純に適用するのは困難である．しかし，OpenMPのようにある一定の枠組みでのみ共有メモリアクセスが生じるようなモデルでは，各スレッドが読み書きする共有メモリ領域や他のスレッドと干渉する操作を特定しやすいことから，本手法を適用できる可能性がある．

7. 関連研究

イベントの発生順序だけをログすることによって，実行を再現する再演手法として，LeBlancらのInstant Replay[6]がある．また，再演法を基礎とする並列プログラムデバッガも多数提案されている（文献7)等）．それらの多くは，再演実行によって，並列プログラムの非決定性を解決できることを重視し，再演実行の繰返しは問題としていない．

再演法に基づくデバッガでは，記録されたメッセージの順序を意図的に入れ替えることによって，別な実行順序をテストするシステムも存在する[8]．本研究は，バグが発現している実行について効率的にデバッギングすることを目的としているので，このような機能は提供しない．しかし巻き戻し機構は，実行の分岐点になる状態を高速に再現するのに利用できると考える．

三栄らは，リプレイ時の“実行の行き過ぎ”を防ぐために，ブレークポイントに到達するうえで必要なコードだけを実行する最小限の再演実行を提案している[4]．これに対しOrderは，行き過ぎた場合，過去の状態に戻すという考え方に基づいている．なお，Orderには最小実行に近い状態に巻き戻す機能が実装されている．

プログラムデバッギングのための巻き戻し実行システムとしては，Igor[3]，Recap[9]等が提案されている．Igorでは，インクリメンタルチェックポインティングによって，Recapではプロセスをフォーク，サスペンドすることによって実行状態を保存している．いずれの手法もメモリアクセスの局所性を期待して保存データ量を削減するものであるが，差分チェックポインティングは，ログデータをより小さくできる．

また両システムはともに，タイマを利用して定期的にチェックポインティングを行っている．これに対して，イベントログ上にチェックポイントを指定する本研究の方法には，(1) ログ量・オーバヘッドの事前の見積りと制御が可能である，(2) 実行時の振舞いを条件としてチェックポイントを指定でき，複雑な条件を指定しても実行時オーバヘッドが増加しない，(3) チェックポイントそれ自体がデバッグ中のユーザが興味を持つ時点になっている，等の利点がある．

このほか，汎用的なチェックポインティングライブラリも発表されている（libckpt[10]等）．これらは細やかな制御やデバッギング情報の取得をサポートしていないため，デバッギング向きではない．

8. おわりに

本論文では，巻き戻し実行法に再演法を組み合わせた並列プログラムのデバッグ手法を提案した．そして実際に，メッセージパッシング型並列言語Orgelを対象として，提案手法を組み込んだ並列デバッガOrderを開発した．Orderは，巻き戻し実行を実用的，効率的に利用するために必要な，イベント・チェックポイント操作機能等を備えたデバッギングシステムである．

さらに，開発した環境のもとで巻き戻し実行機構を組み込んだプログラムの実行時間オーバヘッド，ログ量，巻き戻しコスト等を測定し，提案手法・Orderの実用性を示した．

今後は，Orderにグラフィカルユーザインタフェー

ス等を加え，より使いやすいデバッギング環境を実現し，また Orgel 以外の言語に提案手法を適用して，有効性を検証したい．

参 考 文 献

1) 大野和彦，山本繁弘，岡野孝典，中島 浩：プロセスネットワークを宣言的に記述する並列言語，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.42, No.SIG 12(HPS 4), pp.95–110 (2001).
2) Li, K., Maighton, J.F. and Plank, J.S.: Low-Latency, Concurrent Checkpointing for Parallel Programs, *IEEE Trans. Parallel and Distrib.*, Vol.5, No.8, pp.874–879 (1994).
3) Feldman, S.I. and Brown, C.B.: Igor: A System for Program Debugging via Reversible Execution, *ACM SIGPLAN Notice*, Vol.24, No.1, pp.112–123 (1989).
4) 三栄 武，高橋直久：適応的再演型ロック命令を用いた並列プログラムデバッガの実現，並列処理シンポジウム JSPP'94, pp.241–248 (1994).
5) Clémençon, C., Fristsher, J. and Rühl, R.: Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debuging Tool, *HPCS '95*, pp.393–404 (1995).
6) LeBlanc, T.J. and Mellor-Crummey, J.M.: Debuggin Parallel Programs with Instant Replay, *IEEE Trans. Comp*, Vol.C–36, No.4, pp.471–482 (1987).
7) Kacsuk, P. et al.: A graphical development and debugging environment for parallel programs, *Parallel Computing*, Vol.22, pp.1747–1770 (1997).
8) Kacsuk, P., Lovas, R. and Kovács, J.: Systematic Debugging of Parallel Programs in DIWIDE Based Collective Breakpoints and Macrosteps, *Euro-Para '99*, pp.90–99 (1999).
9) Pan, D.Z. and Linton, M.A.: Supporting Reverse Execution of Parallel Programs, *ACM SIGPLAN Notice*, Vol.24, No.1, pp.124–129 (1989).
10) Plank, J.S., Beck, M., Kingsley, G. and Li, K.: Libckpt: Transparent Checkpointing under Unix, *USENIX Winter 1995 Technical Conference*, pp.213–224 (1995).

(平成 15 年 7 月 31 日受付)
(平成 15 年 11 月 7 日採録)

丸山真佐夫（正会員）

1994 年東京農工大学大学院工学研究科電子情報工学専攻博士前期課程修了．同年木更津工業高等専門学校情報工学科助手．2002 年同講師．現在に至る．2002 年より豊橋技術科学大学大学院工学研究科電子・情報工学専攻博士課程に在学中．並列プログラムのデバッギング手法の研究に従事．

山本 繁弘

2002 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年日立ソフトウェアエンジニアリング（株）入社．在学中は，並列プログラミング言語に関する研究に従事．

大野 和彦（正会員）

1998 年京都大学大学院工学研究科情報工学専攻博士後期課程修了．同年豊橋技術科学大学助手．2003 年三重大学講師．言語の設計・最適化等並列プログラミング環境に関する研究に従事．博士（工学）．

中島 浩（正会員）

1981 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学教授．並列計算機のアーキテクチャ等並列処理に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．IEEE-CS，ACM，ALP，TUG 各会員．