データ再演法による並列プログラムデバッギング

丸山 真佐夫[†,††] 津邑 公暁[††] 中島 浩[††]

並列プログラムのデバッギングの障害になる「実行の非決定性」を解決するために，実行中に発生するイベントの順序を保存，再現する「順序再演法」が広く用いられている．しかし順序再演法は，ロギングのオーバヘッドが小さいという特徴を持つ一方，再演時に全プロセスを動かさなくてはならない，各プロセスを自由な時点で停止させられないなどの制約がある．そこで我々は，順序ではなくイベントの内容自体を保存，再現することで決定的な再演を可能にする「データ再演法」を用いたデバッギングを提案する．データ再演法は，ロギングのオーバヘッド増加とひきかえに，各プロセスを単独で再演できる．また，「データ再演法」を巻き戻し実行と組み合わせることで，さらに強力なデバッギングシステムを実現できる．本稿では提案手法と MPI 上での実装，性能評価について述べる．並列計算機上での実験結果から，本手法が十分に現実的な速度（Nas Parallel Benchmarks でロギング実行は平均 24%の速度低下，再演実行は 38%の速度向上）とログサイズで動作することを示す．

Parallel Program Debugging Based on Data-replay

Masao Maruyama,[†,††] Tomoaki Tsumura[††]
and Hiroshi Nakashima[††]

Nondeterministic nature of parallel programs is the major difficulty in debugging. *Order-replay*, a technique to solve this problem, is widely used because of its small overhead. It has, however, several serious drawbacks: all processes of the parallel program have to participate in replay even when some of them are clearly not involved with the bug; and the programmer cannot stop the process being debugged at an arbitrary point. We adopt another method for deterministic replay, *data-replay*, which logs contents of the events rather than their order, and makes it possible to run and stop each process independently. Data-replay is well able to cooperate with checkpointing/rollback mechanism. We applied the data-replay mechanism to MPI based parallel programs. The result of our experiment with Nas Parallel Benchmarks shows that our mechanism works at a practical cost. Logging communicated data incurs only 24% overhead while it accelerates replayed execution by 38% both in average.

1. はじめに

並列プログラムのデバッギングは，逐次プログラムのそれと比較して，一般により困難である．この難しさの原因としては，

- 並列プログラムの実行における非決定性，
- 複数のプロセスが並行して動作するというそもそもの複雑さ，

があげられる．

従来，実行の非決定性に対する解決として，順序再演法を用いるデバッギング手法が提案されてきた．順序再演法は，プログラムの振舞いを変化させるイベント（たとえば複数の送信元からのメッセージを受け付ける受信動作）の実行履歴を保存・再演することで，プログラム実行の再現性を保証する．順序再演法には，保存すべきデータのサイズが小さく（一般にイベントごとに数バイト程度），したがってまたデータ保存の時間的なオーバヘッドも小さいという特徴がある．

一方，順序再演法には，並列プログラムを構成する全プロセスが，必ず再演実行に参加しなくてはならないという制約がある．たとえバグに関与していないことが明らかなプロセスがあっても，実行を省略できない．この制約は，単に余分なプロセスの存在が煩わしいというだけでなく，デバッギング作業のために貴重な計算機システムを占有するという問題を発生させる．

また，順序再演法を用いたデバッギングでは，各プロセスを任意の実行時点で停止させることができない．たとえば，プロセス A からプロセス B へメッセージ m を送信するプログラムで，プロセス B をメッセージ m の受信後の時点，プロセス A を m の送信以前

† 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
Toyohashi University of Technology

の時点で同時に止めることは不可能である．こういった制約は，プロセスの実行が進み過ぎて先頭からの再実行を余儀なくされる回数を増やす．プロセス B が受け取ったメッセージ m に誤りがあると判明したときには，プロセス A はその原因の痕跡が残らないほど，実行が進んでしまっているかもしれない．あるいは，このような事態を避けるためには，ブレークポイントを注意深く設定しなくてはならず，デバッガユーザの負担を増大させることになる．

そこで本稿では，以上に述べたような性質のある順序再演法と対照的な，データ再演法による並列プログラムデバッガを提案する．

データ再演法は，順序再演法と同様に 1 回の実行で再演に必要なデータを収集し，以後の実行ではこれを利用して再演を行う．ただし順序再演法と異なり，イベントの発生順序ではなく，イベントの内容（たとえば，受信イベントでは受信データそのもの）を保存する．

データ再演は並列プログラム全体ではなくプロセス単位で実行される．そのため，ユーザが注目するプロセスだけを選択して実行することができる．また，プロセスをまたがるイベントの発生順序の制約を受けないので，各プロセスを任意の時点で停止させることが可能である．さらに，単独のプロセスとして実行される再演は，チェックポインティングとの組合せに適する．

歴史的には，データ再演法は順序再演法よりも古くから存在する[1]．LeBlanc らによって Instant Replay[2] として提案された順序再演法は，イベントの内容を保存せずに実行を再現する新しいテクニックとして登場し，広く用いられるようになった．しかし，PC クラスタなどの普及によって数百以上のプロセスからなる並列プログラムが一般的なった現在，デバッギングのしやすさという点から，データ再演法の利点が大きくなっていると考える．

我々は，標準的なメッセージパッシングライブラリである MPI に対してデータ再演法を適用し，現在これを中核とする並列デバッガを開発している．本稿では，提案手法とその実装，性能評価，開発中の並列デバッガについて述べる．

以下，2 章では提案するデバッギング手法の概要，3 章では，提案手法の MPI 上での実装について述べる．続いて 4 章で，システムの性能評価を行う．5 章では，データ再演法を用いる並列デバッガについて述べる．そして 6 章では関連研究を示し，最後に 7 章でまとめを行う．

2. データ再演法

メッセージパッシング並列プログラムにおいて，実行の非決定性の原因がメッセージ受信だけであるなら，各プロセスで発生する受信イベントを正確に記録し再現することによって，プログラムの決定的な再実行が可能である．

順序再演法は，イベントの発生順序だけを保存するというテクニックによって，ログデータを劇的に小さくすることができた．半面，1 章で述べたような再演実行時の制約が生ずる．

さて，イベントの発生順序ではなく，より素朴にイベントの結果そのもの（たとえば受信データ）を保存し，再演時には，実際のイベント処理を行うかわりに保存されたデータを利用して実行を進めるという方法でも，各プロセスを決定的に再実行することが可能である．データ再演法は，この原理に基づく．

順序再演法と比較したデータ再演法の特徴は，次のとおりである．

(1) プロセス単体が再演の対象になる

順序再演法は，並列プログラムを実行するプロセス全体を再演する．それに対してデータ再演法では，各プロセスを単独で再演することが可能である．

PC クラスタなどの発達によって広く普及したとはいえ，大規模な並列計算機システムは依然として，デバッギングのために長時間占有できる計算機資源ではない．この点でデータ再演法は，ユーザが注目しているプロセスだけを選択して再演実行できるので，バグに関与しないことが明らかなプロセスのために計算機資源を浪費せずにすむ．さらに，再演されるプロセスを，ロギング実行を行ったものと別の計算機で実行できるので，並列計算機上ではロギング実行だけを行い，それ以後のデバッギング作業は，手元の計算機で行うことも可能である．

(2) ブレークポイントを自由に設定できる

各プロセスは単独で再演されるため，他のプロセスの進行に影響されずに，自由にブレークポイントを設定できる．いいかえると，ユーザはいつでも，注目しているプロセスを望む時点まで実行させることができる．

(3) チェックポインティングとの組合せに適する

本手法とチェックポインティング・巻き戻しを組み合わせて用いる場合，単一プロセスに対するチェックポインティングのテクニックが適用可能である．On-the-fly メッセージの処理などの，プロセス間通信にともなう問題を考慮する必要がないため，チェックポイ

ンティングによるオーバヘッドを軽減できる．

それでもなお，チェックポインティングには大きなコストがかかるが，次のようなデバッガの機能によって，ユーザに対してコストを隠蔽することが可能である．ユーザがあるプロセスの再演を要求したとき，デバッガは通常の再演実行を行うプロセスとともに，別の空いている計算機上で，チェックポインティングを行うプロセスを走らせるのである．「表」の再演プロセスはしばしばユーザによって停止させられるので，「裏」のチェックポインティングプロセスは大きく遅れることなく，場合によっては先んじて進行すると期待できる．ユーザは，「未来」の状態を含めたチェックポイント生成済みの時点に，自由に実行を巻き戻すことが可能になる．

順序再演法をベースにして，バックグラウンドでのチェックポインティングを行う場合，同規模の並列計算機をもう 1 式用意しなくてはならないので，現実的でない．

(4) ログサイズとロギング時間のオーバヘッドが大きい

イベントの順序だけを保存する順序再演法と比較すると，一般にデータ再演のためのログデータは大きくなると考えられる．それにともなって，ロギングの時間的オーバヘッドも増大する．

ただし，次の理由から，ログデータやオーバヘッドの大きさは，本手法を非現実的なものにするほどではないと考える．一般的なプログラムでは，受信メッセージがログデータの大半を占めると予想される．プロセス間で通信されるメッセージの量は，並列プログラム自体のパフォーマンスにも大きく影響するため，プログラム作成者は，通信量を低くおさえようとするはずである．したがって，ネットワークとディスク I/O のスループットが同程度であることを勘案すれば，通信量と同等のロギングを行うことは，少なくともオーバヘッドに関しては許容しうる値となることが予想される．実際，4 章に示す評価により，ログサイズとロギングオーバヘッドの双方について，データ再演が現実的であることが示されている．

3. データ再演・チェックポインティング機構の実装

本章では，提案するデータ再演機構の実装について述べる．我々のシステムは，データ再演とともに利用可能なチェックポインティング・巻き戻し実行機構も実装しており，これについても 3.3 節に示す．

現在のシステムは Linux 2.4.7 上の MPICH-1.2.5.2 を対象に開発している．3.1 節で述べるように，特定の MPI 実装には依存しないので，LAM/MPI など別システムへの適用も容易である．

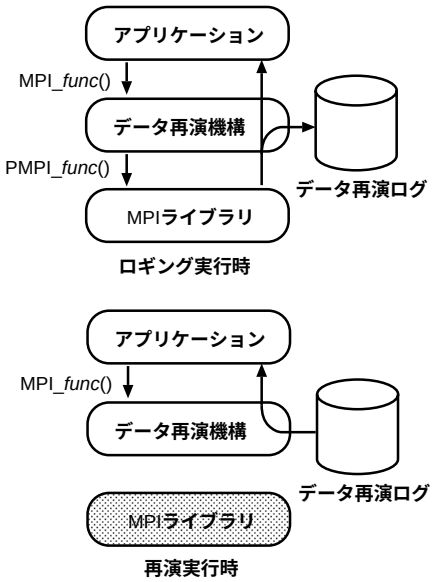


図 1 データ再演機構の動作
Fig. 1 Data logging/replay mechanism.

3.1 データ再演機構の構成

すでに述べたように，提案手法は並列プログラムを構成する各プロセスを単独で再演実行させる点に特徴がある．これを実現するためには，各プロセスがロギング実行時に，自身で再現不可能なイベントの結果をすべて保存しておく必要がある．

保存を必要とするデータは，「外部」と「内部」の境界の引き方によって異なる．もし TCP ソケットを境界とするなら，ソケットを経由して読み込まれるデータを保存することになるだろう．この方法では，MPI 内の状態も含めて再現可能になる．しかし，アプリケーションのデバッギングにおいては通常，通信路や MPI 内部の状態は不要なので，これは利点にはならない．

我々は，MPI 実装および OS への依存性を低くするために，アプリケーションコードと MPI ライブラリ関数の間に境界を引いた．ロギング実行では，MPI ライブラリ関数呼び出しの結果（戻り値，出力パラメータ）をすべて保存する．そして，再演実行時は MPI 関数をまったく呼び出さずに，保存しておいた値を返す．いわば MPI をシミュレートすることになる．**図 1** に，本システムの実装を示す．

我々のロギング/再演機構は，MPI 標準のプロファイリングインタフェースだけを用いて，MPI ライブラリ関数へのラッパとして実装されている．そのため，

本システムを利用するためには，MPI ライブラリおよびアプリケーションプログラムのソースコードの修正を必要としない．

3.2 ログデータの保存と再現

本システムでは，データ再演実行を行うのに必要な最低限のデータ（戻り値と出力パラメータ）に加えて，イベント間の因果関係解析に必要な情報なども保存する．この中には，送受信の際のコミュニケータ，メッセージタグ，送信データ型などが含まれる．

たとえば MPI_Recv() 関数に対しては，

- 関数の戻り値
- ステータス
- 受信バッファ
- 送信元
- メッセージタグ
- コミュニケータ

をロギング実行時に保存する．このうち，再演に必須のデータは最初の 3 つである．それ以外のデータは，再演時には読み捨てられる．

一方，MPI_Send() 関数に対しては，

- 関数の戻り値
- 送信先
- メッセージタグ
- コミュニケータ

を保存する．送信データを保存する必要はない．

次に示すように，一部の関数については，戻り値やパラメータを保存，再現する以外の処理を行っている．

(1) 非ブロッキング受信

MPI_Wait() などで非ブロッキング受信を完了する場合，受信バッファのアドレスは Request ハンドルによって間接的に指定され，関数パラメータに含まれない．

本システムでは，Request ハンドルと受信バッファの対応表を保持することで，この問題に対応する．MPI_Irecv() などで Request ハンドルを作成した際にハンドルとバッファアドレスを表に登録し，MPI_Wait() などではこの表を参照して保存すべき（再演時には保存されたデータを読み込むべき）受信バッファのアドレスを決定する．

(2) コミュニケータの生成

コミュニケータに属するプロセス集合の情報は，データ再演には不要なものの，デバッギング作業上重要な情報である．そこで，MPI_Comm_create() などのコミュニケータを生成する関数に対しては，生成されたコミュニケータのハンドルとともに，そのコミュニケータに属する各プロセスの MPI_COMM_WORLD におけるランクを保存する．

上に述べたとおり，本システムはデータ再演には必要でない情報も保存している．これらの付加的な情報は順序再演に必要なデータを包含するので，イベント間の順序関係の解析など，順序ログを用いる並列デバッガの機能は，本システムでも容易に実現できる．たとえば本システムでは，受信イベントを実行するために，対応する送信イベントの実行を必要としないが，受信プロセスの再演を受信イベントの直前で停止させたとき，送信側のプロセスを対応する送信イベントまで実行させることなどは，デバッガの機能として実現可能である．

このように，再演をプロセス単位で行うことは，プロセスをまたぐバグに対する本システムのデバッグ能力を制約するものではない．通信内容の誤りや非決定性が関係するバグなど，複数プロセスを扱う必要のあるケースについても，本システムを用いてデバッグすることが可能である．

3.3 チェックポインティング・巻き戻し

本システムのチェックポインティング機構は，単一プロセスのメモリ状態の保存と回復の機能を提供する．これとデータ再演機構を組み合わせることによって，各プロセスを個別に，任意のチェックポイント時点から再実行する機能を提供する．

チェックポイントデータは，スタックコンテキスト，静的データ，ヒープ領域，スタックからなる．静的データとヒープ領域については，前回のチェックポイント以降に変更され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を採用している．ページ書き換えの検出には，mprotect() を利用している．これに対してスタックは，スタックトップまでの全領域をそのまま保存する．スタックコンテキストは，setjmp() を用いて保存する．

現在の実装では，ローカルファイル操作など，MPI を用いない入出力を考慮していないが，従来提案されている方法でサポートすることは可能である．

チェックポインティングのインターバルは実行時にユーザが指定する．ただし，本システムでは 3.1 節で述べた MPI 関数ラッパ中でタイマをポーリングしてチェックポインティングを行うため，ユーザが指定する値は最低のインターバルという意味になる．このような実装を採用したのは，MPI 関数の呼び出しというイベントが，実際のデバッギング中の巻き戻し先として指定される可能性が高いと考えたからである．タイマの割込みなどによる非同期的なチェックポインティングを実装することは可能である．

表 1 測定環境
Table 1 Specification of the computer for the experiments.

CPU	PentiumIII 866 MHz
主記憶	512 MB
ネットワーク	1000Base-T
使用ノード数	16（NPB）/43（OG）
OS	Linux 2.4.7

巻き戻し時は，チェックポイントデータを逆向きにたどっていく．すなわち，まず巻き戻し先チェックポイントで保存されているメモリページをリストアする．次に，その 1 つ前のチェックポイントのうち，まだリストアされていないページだけを書き戻す．これを最初のチェックポイントまで繰り返すことで，同一ページを 2 回以上リストアするのを防げる．

データ・ヒープ領域のリストア後，スタック領域を書き戻し，最後に longjmp() でコンテキストを回復する．

4. 性能評価

提案手法の有効性を検証するため，クラスタ並列計算機上のベンチマークプログラムに適用して，実行時間，ログデータのサイズなどを測定した．

測定には**表 1** に示す計算機環境を用いた．ベンチマークプログラムとして，Nas Parallel Benchmarks 2.3（NPB）の BT，CG，EP，IS，LU，MG，SP の 7 つと，初手から終局までオセロを自動的に対局するプログラム OG を用いた．

OG は我々が作成したプログラムである．プロセスを深さ d の n 分木状に構成して，ゲーム木を並列 $\alpha\beta$ 探索する．葉にあたるプロセスは，逐次 $\alpha\beta$ 探索を行う．本プログラムの振舞いは非決定的であり，同一局面に対する着手が，実行ごとに異なる可能性がある．**図 2** に，$d = 2$，$n = 3$ とした例を示す．

NPB の計算のクラスは B，プロセス数は 16 である．OG については，プロセス木の深さ d を 2，分岐の数 n を 6 とし，全体で 43 のプロセスを用いた．また探索するゲーム木全体の深さを 10 とした．

なお本章で示す測定値は，10 回の実行の平均である．

4.1 実行時間とログサイズ

オリジナルのプログラムの実行と，ロギング実行，データ再演実行，さらに，比較のために実装した，順序再演法の実行時間も測定した．測定結果を**表 2** に示す．また，両再演法が保存したログデータ（それぞれデータログ，順序ログと呼ぶ）のサイズを**表 3** に示す．ログデータは各ノードのローカルハードディスクに保存した．表 3 の値は，NPB については，ランク 0 のプロセスのログデータのサイズである．測定対象とした NPB のプログラムでは，ランク 0 プロセスの計算量は，他のプロセスと同等以上である．OG は，最初に探索局面が渡されるために計算量が最も多いと期待される，左端の葉プロセス（ランク 7）の値を示している．

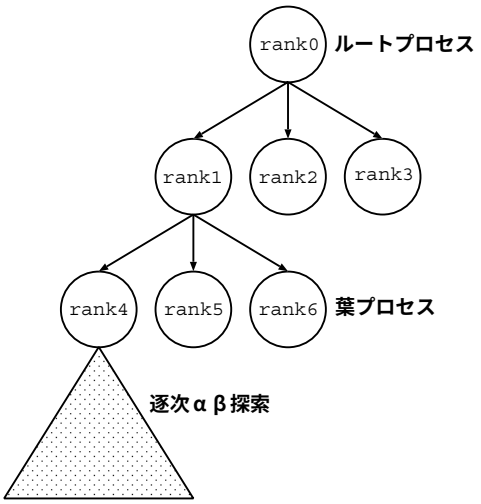


図 2 プログラム OG のプロセス構造
Fig. 2 Process tree of the program OG.

データ再演法の再演については，ランク 0（NPB），ランク 7（OG）プロセスのみを実行した．順序再演法では単独での再演が不可能なため，全プロセスを実行した．このように，必要なプロセスだけを選択的に再演できる柔軟性は，データ再演法の利点である．

(1) 実行時間

データ再演法のロギング実行は，オリジナルの実行時間と比較して最小 1.00 倍（OG）から最大 1.68 倍（IS）の範囲にあり，平均では 1.24 倍だった．

順序再演法のロギング実行はオリジナルの実行に対してほとんどオーバヘッドがない．

2 つの方法を比較すると，ロギング実行に関しては，予想どおり順序再演法の方が優れた結果になった．データ再演法の 20〜30%程度のオーバヘッドは，十分に許容できる程度であると考える．

ただし，実行のタイミングに依存するバグの中には，オリジナルの実行との動作の違いが大きくなると発現しなくなるものがある．この問題については，実行時間からの影響の受けやすさがバグごとに異なるため，今後，本手法のオーバヘッドによって現実のデバッギングでどの程度の影響が生じるかを評価する必要があると考える．

表 2 ベンチマークプログラムの実行時間 [s]（括弧内は基本実行を 1 として正規化）
Table 2 Execution time of benchmark programs.

	基本	データログ	データ再演	順序ログ	順序再演
BT	459.6(1.00)	487.4(1.06)	398.1(0.87)	457.3(0.99)	457.3(0.99)
CG	211.2(1.00)	337.2(1.60)	66.1(0.31)	228.0(1.08)	230.6(1.09)
EP	70.4(1.00)	71.4(1.01)	67.6(0.96)	70.1(1.00)	70.1(1.00)
IS	20.4(1.00)	34.3(1.68)	6.1(0.30)	20.5(1.00)	20.4(1.00)
LU	240.1(1.00)	264.0(1.10)	182.3(0.76)	240.5(1.00)	241.0(1.00)
MG	23.4(1.00)	32.5(1.39)	10.2(0.44)	23.8(1.02)	24.2(1.03)
SP	414.2(1.00)	456.7(1.10)	282.9(0.68)	416.3(1.01)	416.5(1.01)
OG	290.7(1.00)	291.3(1.00)	190.9(0.66)	291.8(1.00)	291.8(1.00)
平均	(1.00)	(1.24)	(0.62)	(1.01)	(1.02)

表 3 ベンチマークプログラムのログサイズ
Table 3 Amount of logs of benchmark programs.

	イベント数	データログ [MB]	順序ログ [MB]
BT	36,436	506.6	0.46
CG	69,928	850.4	0.96
EP	11	0.0	0.00
IS	42	88.9	0.00
LU	102,561	153.4	1.37
MG	13,627	45.7	0.19
SP	65,418	907.2	0.84
OG	2,823	0.2	0.20

また，ごくまれにしか発現しないバグをとらえるために，ロギング実行を有効にしたままテスト運用をするようなケースでは，可能な限りスローダウンが小さいことが望ましい．このような目的には，オーバヘッドがより小さい順序再演法のロギングの方が適している場合がある．

なお提案手法は，順序再演を行っているプロセスのデータロギングを行うことも可能である．この手順ではデータログを得るために 1 回多くプログラム全体を実行する必要があるが，低オーバヘッドな順序ロギングで得た順序を利用しながら，提案手法のメリットを受けることができる．

再演実行に関しては，最小で 0.30 倍（IS），最大で 0.96 倍（EP），平均 0.62 倍になった．オリジナルの実行よりも高速になる主要な原因は，再演実行ではバリア同期やメッセージ受信などのイベントに対して，実際には他のプロセスを待たずに実行を継続することにある．一方，順序再演法の場合，イベントの順序を入れ換えることはあっても，処理内容自体はロギング実行時と同じであるため，実行時間はほとんど変わらない．

提案手法は，再演回数が増えるに従って実行速度に関して効果が大きくなる．たとえば SP の場合，オリジナルのプログラムを 4 回実行すると 1,656 秒かかるのに対して，ロギング実行のあと再演実行を 4 回行うのに要する時間は，1,588 秒である．つまり，ロギング実行の時間を完全なオーバヘッドと見なしても，再演実行の高速性によってそのコストを回収できている．

より一般的な状況を想定して，1 回のロギング実行の後，プログラム全体の半分の時点まで n 回再演実行するというデバッギングのシナリオに，表 2 の平均値をあてはめてみる．基本実行 1 回の実行時間を 1 としたときの，データ再演法，順序再演法の相対的な実行時間をそれぞれ $T_d(n)$，$T_o(n)$ とすると，

$$T_d(n) = 1.24 + 0.62n/2,$$
$$T_o(n) = 1.01 + 1.02n/2$$

と予測できる．1 回目の再演ですでに $T_d(1) = 1.55$ は $T_o(1) = 1.52$ とほぼ等しく，以後再演回数が増えるごとにデータ再演法の有利性が大きくなる．

実際のデバッギングでは，1 回のロギング実行に対して 1 回の再演実行でバグにたどりつけることはまれなので，これはデータ再演法の重要な利点であると考える．しかも，これらの再演実行の繰返しを 1 プロセスごとに行えるので，つねに全プロセスを動かす必要のある順序再演法と比較すると，計算機資源を大きく節約できることになる．

(2) ログサイズ

データ再演法のログサイズは，例外的に小さな EP，OG を除くと，46 MB（MG）から 907 MB（SP）となった．これに対して順序再演法では最大でも 1.4 MB（LU）であり，それ以外は 1 MB 未満である．

1 GB 程度のデータは，現在の計算機の水準からすると，特に問題にはならない大きさである．ただし今回測定したプログラムは，最長でも 8 分程度と，実行時間が短い．そこで，より現実的な実行時間でのログサイズを予測するために，オリジナル実行 1 秒あたりのログサイズを求めた（**表 4**）．

時間あたりのログサイズが最も大きい IS をこのまま 1 時間実行すると，約 15 GB のログデータが生成されることになる．各ノードがローカルなハードディ

表 4 1 秒あたりのデータログサイズ [MB/s]
Table 4 Amount of logs per second.

BT	CG	EP	IS	LU	MG	SP	OG
1.10	4.03	0.00	4.36	0.64	1.95	2.19	0.00

表 5 チェックポインティングの実行結果
Table 5 Execution results of checkpointing.

	サイズ [MB]	回数	実行時間 [s]	対基本比
BT	753.3	11.5	501.5	1.09
CG	43.5	10.9	72.6	0.34
EP	1.2	1.0	67.6	0.96
IS	68.5	3.0	7.8	0.38
LU	114.1	10.0	191.2	0.80
MG	222.5	10.0	34.0	1.45
SP	233.4	10.5	313.5	0.76
OG	0.4	9.0	191.0	0.66

スクを備えるシステムであれば，数時間程度の実行をロギングが可能であると予想できる．

4.2 チェックポインティング・巻き戻し

次に，チェックポインティングの実行時間とデータサイズの測定を行った．

実験では，ランク 0（OG はランク 7）のプロセスについて，データ再演実行を行いつつ，チェックポイントをとる．チェックポイントのインターバルは，データ再演実行時間の 1/10 に設定した．**表 5** に，生成されたチェックポイントデータの大きさ，実行時間と，実際にとられたチェックポイントの数を示す．

3.3 節で述べたように，本システムはイベントに同期してチェックポインティングを実行するため，ユーザが指定したインターバルを保証しない．しかし EP，IS 以外は，ほぼ 10 回チェックポイントがとられている．

今回の測定結果では BT，EP，MG 以外の 4 プログラムの場合，チェックポイントデータの大きさがデータ再演ログよりも小さい（EP は再演ログが極端に小さいので例外的）．また BT，MG 以外のプログラムの実行時間は，オリジナルの実行よりも高速である．以上から，単独のデータ再演法が適用できるアプリケーションに対しては，全実行時間の 1/10 のインターバルでのチェックポインティングも，現実的なコストで適用可能であると考える．

チェックポイントのインターバルを小さくすれば，ユーザが巻き戻したい実行時点までの，その直前のチェックポイントからの実行時間が短くなる．一方，チェックポインティングの回数が増加すると，当然そのコスト（時間，データサイズ）は大きくなる．ただし，チェックポイントデータの大きさは，単純にインターバルに比例するものではない．したがって，今後さらに実行時間の長いプログラムを用いた検証が必要である．

表 6 巻き戻しの時間 [s]
Table 6 Execution time for rollback.

	1 番目の CP	6 番目の CP	11 番目の CP
BT	5.42	5.86	6.03
CG	5.78	5.90	6.00
LU	5.45	5.77	5.86
MG	2.40	2.55	2.71
SP	5.41	5.72	5.86
OG	0.10	0.10	0.12

最後に，チェックポイントへの巻き戻しを測定した．チェックポインティング回数の少ない EP，IS を除いた各プログラムについて 11 回のチェックポイントをとり，進行度順に 1，6，11 番目の 3 つのチェックポイントへの巻き戻しに要する時間を調べた．**表 6** に，測定結果を示す．3.3 節で述べたとおり，各ページについて 1 回だけデータをリストアするため，チェックポイントファイルが大きい BT でも，巻き戻し時間は 6 秒程度である．なお，BT では 1 回の巻き戻しで約 17,400 ページ，68 MB のメモリ領域がリストアされる．巻き戻し後，プロセスが正常に動作を再開できることを確認した．

今回のプログラムはいずれも，各チェックポイントでリストアされるページ数がほぼ一定である．そのため，1 番目と 11 番目のチェックポイントへの巻き戻し時間の差が 10 個のチェックポイントを逆向きにたどるコストであると考えられる．このことから，仮にチェックポイントの数が 100 カ所に増えたとしても，巻き戻しに要する時間は数秒程度の増加にとどまると予想できる．

5. データ再演法を用いた並列デバッガ

本章では，現在開発中のデータ再演法を利用する並列デバッガについて述べる．

5.1 イベント操作言語 eml

多数のプロセスやイベントなどが複雑に関係しあう並列プログラムのデバッギングには，それらの情報をデバッガユーザに分かりやすく示したり，効率的に扱えるようにしたりする機能が求められる．

並列プログラムの動作の大まかな様子を見せるという点では，イベントグラフなどのグラフィカルな表示は効果的な方法である．しかし，デバッギング作業を進めるうえでは，イベントなどを論理的あるいは記号的に扱うことも必要になる．たとえば，「コミュニケータ C に属するいずれかのプロセスで発生するイベントのうち，ランク p からのメッセージを受信する

もの」を求める場合，おそらくグラフィカルなインタフェースよりも，上の条件を記号的に指定する方が適している．

我々はこれまでに，並列言語 Orgel のためのデバッガ Order に，イベントの集合を扱ういくつかの機能を組み込んできた[3]．本研究では，より一般的で強力なイベント操作の機能を提供するために，並列デバッガのためのログ操作言語 eml を開発した．eml は，アプリケーションによって数万あるいはそれ以上にものぼるイベント（MPI ライブラリ関数の呼び出し）を効率的に扱うことを目的とする，集合操作を基礎とする言語である．

また，イベントの操作とデバッグ対象プロセスの操作をシームレスに行えるようにするために，eml は並列デバッガの中で，コマンドライン処理言語としても利用することを想定している．

eml の言語仕様の要点は次のとおりである．なお eml の文法は，Pascal あるいは Modula-2 に近い．

(1) データと変数

eml で扱える型は，

- 数値（整数，浮動小数点数）
- 文字列
- プロセス，イベント，コミュニケータの各集合

である．

変数は型を持たず，上記のいずれの値も代入することができる．また変数は宣言なしに利用することができ，その場合はグローバル変数と見なされる．関数内で宣言された変数は，その関数を抜けるまで生存するローカル変数になる．

eml のシステムはデータ再演システムのログを元に，集合データを自動的に生成し，それぞれ変数 PROC, EVENT, COMM に格納する．集合の各要素は，ログデータから作られるフィールドによって構成されるレコードである．たとえば MPI_Recv() に対しては，

- seq：プロセス内でのイベントの番号
- type：イベントの種類（すなわち MPI_RECV）
- proc：このイベントを発生させたプロセス
- source：メッセージの送信元
- comm：コミュニケータ
- tag：メッセージタグ
- dtype：受信データ型

というフィールドからなるレコードが生成される．EVENT 要素のフィールドは，イベントの種類ごとに異なる．

PROC, COMM の要素のレコードは固定である．プロセスは rank（ランク），comms（プロセスが属するコミュニケータの集合）など，コミュニケータは procs（コミュニケータに含まれるプロセスの集合）などのフィールドを持つ．

ユーザは内包的集合式，列挙集合式および **(2)** で述べる演算を用いて，これらの集合のサブセットを扱うことができる．

内包的集合式は，

```
eset := {e :: EVENT | e.type = MPI_RECV
                    & e.source = PROC[0] };
```

のように表現する．“::” は ∈ を表す．また，`PROC[0]` は集合 PROC の 0 番目の要素を意味し，PROC がランクの昇順に順序づけられていることから，ランク 0 のプロセスを指すことになる．したがって，上の文は「集合 EVENT のうち，ランク 0 のプロセスからの受信イベントのサブセットを変数 `eset` に代入する文」になる．

列挙集合式は，

```
PROC[ 0, 5:n ]
```

のように，要素の番号または範囲によって指定されたサブセットを生成する．なお，eml は集合の要素と 1 つの要素からなる集合を区別しないので，`PROC[0]` が要素であるか集合であるかのあいまいさは問題にならない．

(2) 集合に対する演算子

集合に対する演算子として “+”（和），“-”（差），“*”（交わり），“~”（補集合）が定義されている．“::” は内包的集合式以外の一般の式の中でも，∈ の意味で用いることができる．

(3) 制 御 構 造

eml は，制御構造として分岐（`if-elsif-else`），反復（`while`，`for`）を持つ．`for` については，

```
for a:=0 to n by c do
  文 ...
end
```

の形式のほか，

```
for(e :: SET) do
  文 ...
end
```

のような，集合の全要素に対する適用も書ける．

(4) 関　　数

eml では，ユーザが関数を定義することができる．**(1)** で述べたように関数内にローカルな変数を作れるので，再帰関数を書くことも可能である．

現在の eml 処理系は LISP へのトランスレータとして実装されており，eml の出力を別プロセスの CommonLisp 処理系がインタプリティブに実行するとい

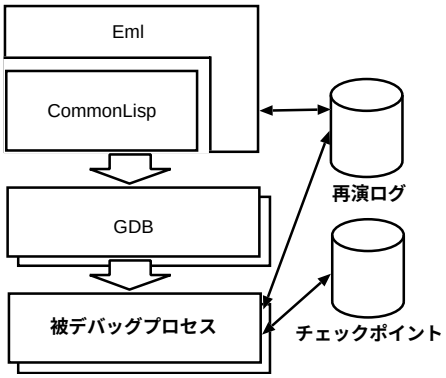


図 3 デバッガの構成
Fig. 3 The debugger architecture.

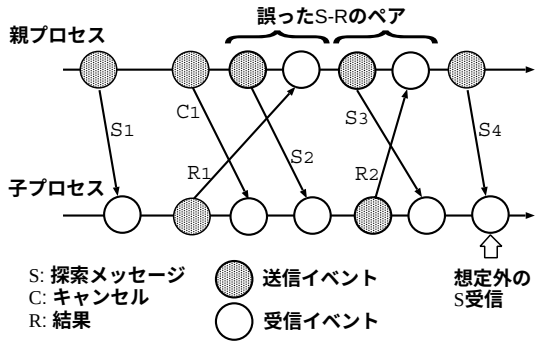


図 4 オセロのイベントログ
Fig. 4 An example of event log of OG.

う構成をとっている．

5.2 並列デバッガの構成

eml を用いた並列デバッガの構成を**図 3** に示す．上述のとおり，eml がログ操作などの高レベルの機能を担う．各デバッグ対象プロセスは，（おそらく別計算機上で）逐次デバッガの制御下で実行され，eml はデバッガを通してプロセスにアクセスする．逐次デバッガとしては，GNU Debugger（GDB）を想定している．

5.3 本システムによるデバッギング

4 章で用いたオセロ対局プログラム OG を例に，本システムを利用したデバッギングにおいて想定される進行を述べる．

OG では，親プロセスから子プロセスに対して，探索を指示するメッセージ Search，現在の探索の中断または $\alpha\beta$ 値更新のいずれかを指示するメッセージ Control が送られる．子プロセスからは，解（局面評価値）Result が返される．これらのメッセージは，MPI のメッセージタグによって区別される．

ゲーム木探索では，枝刈りによって，あるノード以下の部分木の計算が不要になる場合がある．本プログラムでは，枝刈りが可能になると，子プロセスに探索中断の Control メッセージ（以下では Cancel メッセージという）を送り，探索を中断させる．子プロセスからは，Cancel メッセージに対する応答をせず，一方親プロセスは，Cancel メッセージを送ったままにして，次の処理に進む．

図 4 の C_1 と R_1 のように，探索中断を指示する親からの Cancel メッセージと子からの Result メッセージが行き違いになる場合がある．このままでは，親プロセスはキャンセルしたつもりの探索の Result メッセージを，次の Search メッセージに対する解として受け取ってしまう（図 4 では R_1）．

そこで親プロセスは，メッセージ送信ごとにカウントアップするシーケンス番号を，Search メッセージに付加する．子プロセスは，受け取ったシーケンス番号を Result メッセージに含めて返す．親プロセスは，受信した Result メッセージのシーケンス番号が，Cancel 発生時以前の値であるなら，破棄すればよい．

さて，この探索結果の破棄の処理において，

```
if (result の seq ≦ キャンセルした seq)
  result を破棄;
```

とすべきところで，誤って「≦」を「<」と書いてしまったとしよう．すると，上に述べた Result メッセージの誤受信が発生しうる状態になる．OG では，個々の子プロセスから返ってくるはずのシーケンス番号を記憶しない（論理的には不要である）ため，Result メッセージの誤りを認識できない．実行が進んで，たまたま前の Search メッセージに対する探索が完了する前に次の Search を受信したとき（S_4 の受信），不正なメッセージであることが検出可能である．

本システム上で，この問題のデバッグを行う場合，次のような手順が想定される．

最初にロギング実行を行い，バグ発現時のデータログを得てから，実質的なデバッグ作業を開始する．まず将来の巻き戻しに備えて，逐次探索を行う葉ノードプロセスを，バックグラウンドでチェックポインティング実行しておく．計算機資源を節約する必要がある場合は，Search-Result の不整合を検出したプロセスの部分木に属するノードだけに限定してもよい（プログラムの特性上，それ以外の葉ノードがバグに関与している可能性は低いと予想できるため）．このように，プロセスごとに巻き戻し機構を利用するかどうかを選べることも，本システムの利点の 1 つである．

次に，問題のプロセスを S_4 受信時点まで再演実行する．OG は，葉ノード以外のプロセスの計算量がごく小さいので，このプロセスが葉プロセスでないなら，データ再演法を用いる本システムでは，巻き戻し機構

を利用するまでもなく一瞬で問題の時点に到達する．

受信したメッセージ自体には問題がないため，以後デバッガユーザは，S_4 を送信したプロセスの部分木に属するプロセス集合を中心に再演実行を繰り返しながら，バグの真の原因を探すことになる．

デバッギングの際，図 4 のようなイベントグラフは非常に有用である．eml のイベント操作機能によって，ユーザが注目するプロセスやイベントだけを抽出して表示することが容易に行える．この例では，問題のメッセージ S_4 の送受信プロセスのイベントだけを表示することで，ユーザは一見して，Search S_3 とその探索結果であるはずの Result R_2 が，並行して送信されている点に気づくことができる．

順序再演法を用いてこの問題のデバッグを行う場合，過去の状態にさかのぼるたびに，通常実行と同程度の時間を要する再演実行をしなくてはならない．順序再演法と巻き戻し法を併用するとしても，巻き戻しとチェックポイントからの順方向実行に要する時間は，提案手法よりも大きい．

また，疑わしいプロセスのそれぞれを独立して進行させられないため，再演実行の回数が増える可能性が高くなると予想する．たとえば，親プロセスを C_1 送信時点で止めたまま，子プロセスを R_1 送信時点から S_4 受信時点まで実行してみることは，順序再演法では不可能である．

6. 関連研究

初期のデータ再演法のシステムとして，BugNet [1] がある．また，データ再演法とチェックポインティングを組み合わせたデバッギングシステムとしては，Igor [4]，Recap [5] などが提案されている．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た．Recap は VAX11/780 上で，1 MB/s の再演ログデータを出力する．この値は，当時の計算機の能力に対しては過大なコストであった．Ronsse らは，この結果を引いて，データ再演法はコストの点で現実的でないために，現在は用いられていないと述べている [6]．

順序再演法として，LeBlanc らの Instant Replay [2] がある．また，順序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) など）．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．

Netzer らは順序再演を基本としながら，「ドミノ効果」を防ぐために必要な一部のメッセージ内容（全メッセージの 1〜10%程度）を保存することで，並列プログラムのチェックポインティング/巻き戻しを実現する方法を提案した [8]．チェックポインティングは，単独（uncoordinated）で行われる．

この手法の利点は，メッセージロギングのコストを小さくできることである．一方，本提案手法と比較すると，(1) 巻き戻しと再演実行が遅い，(2) 実装が MPI などの通信ライブラリ（またはそれ以下の層）に強く依存する，という問題がある．

(1) については，すでに述べたように本提案手法の再演実行は本来の実行よりも高速であることが期待できる．それに対して Netzer らの手法では，通常実行と同等であるか，むしろ遅くなる．これは，あるプロセスを 1 チェックポイント区間再演するためにそれ以外のプロセスを 1 区間より多く（文献 8) によると，実験的には最大で 1〜2 区間程度）再演しなくてはならない場合があることによる．

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 [9]．データ再演法を用いる我々のシステムでは，各プロセスを任意の時点で停止できるので，この意味での実行の行き過ぎを防ぐのに，特別の工夫は不要である．

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] など）．我々のシステムでは，チェックポインティング対象は個々のプロセスなので，シングルプロセス向けのチェックポインティングライブラリや高速化手法を利用することが可能である．

7. おわりに

本稿では，データ再演法とチェックポインティング・巻き戻し手法に基づくデバッギング手法を提案した．データ再演法は，再演実行の単位がプロセスであることが特徴であり，順序再演法と比較して実際のデバッギングでの柔軟性が高い．

また我々は，MPI を対象として提案手法を実装し，順序再演法との比較を含む性能評価を行った．その結果，提案手法が十分に現実的なコストで実行可能であり，実行時間については順序再演法よりも優れていることを示した．

さらに，本手法を用いた並列デバッガの主要な構成要素になるイベント操作言語について述べた．

今後は，開発中の並列デバッガを実現し，実際のデ

バッギング作業における提案手法の有効性を検証したい．

謝辞 本研究の一部は科学技術振興機構・戦略的創造研究推進事業（CREST）の研究プロジェクト「低電力化とモデリング技術によるメガスケールコンピューティング」による．

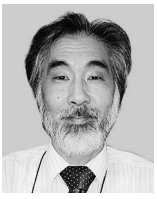
参 考 文 献

1) Curtis, R. and Wittie, L.: BUGNET: A Debugging System for Parallel Programming Environments, *Proc. 3rd International Conference on Distributed Computing Systems*, pp.394–399 (1982).
2) 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).
3) 丸山真佐夫，山本繁弘，大野和彦，中島 浩：巻き戻し実行をサポートする並列プログラムデバッガ，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.SIG 3(ACS 5), pp.109–121 (2004).
4) Feldman, S.I. and Brown, C.B.: Igor: A System for Program Debugging via Reversible Execution, *ACM SIGPLAN Notices*, Vol.24, No.1, pp.112–123 (1989).
5) Pan, D.Z. and Linton, M.A.: Supporting Reverse Execution of Parallel Programs, *ACM SIGPLAN Notices*, Vol.24, No.1, pp.124–129 (1989).
6) Ronsse, M., et al.: Execution replay and debugging, *Proc. 4th Intl. Workshop on Automated Debugging* (*AADEBUG 2000*), pp.5–18 (2000).
7) Kacsuk, P., et al.: A graphical development and debugging environment for parallel programs, *Parallel Computing*, Vol.22, pp.1747–1770 (1997).
8) Netzer, R.H.B. and Xu, J.: Adaptive message logging for incremental replay of message-passing programs, *Proc. 1993 ACM/IEEE Conference on Supercomputing*, pp.840–849 (1993).
9) 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).
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 17 年 1 月 24 日受付)
(平成 17 年 5 月 16 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．現在同助教授．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．

津邑 公暁（正会員）
1998 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．2001 年同大学院情報学研究科博士後期課程学修認定退学．同年同大学院経済学研究科助手．博士（情報学）．2004 年豊橋技術科学大学工学部助手．計算機アーキテクチャ，並列処理応用，脳型情報処理等に関する研究に従事．電子情報通信学会，人工知能学会，日本神経回路学会各会員．



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．