ソフトウェア基礎論　３３－１０
プログラミング言語　２３－１０
（1989. 12. 8）

データ共有型並列プログラムの部分再演法について

高橋　直久
ＮＴＴソフトウェア研究所

あらまし　LeBlancとMellor-Crummeryが提案したInstant Replay は，共有データ型並列プログラムの実行動作を再現し，サイクリックなデバッグを並列プログラムにも適用可能にする手法である．本論文では，Instant Repaly を一般化し，データフローグラフの実行モデルに基づく３つの再演法モデルと基本データ構造の異なる３種類の実現技法の組合せとして表現できる９種の再演法を提示する．これらの組合せの中で，要求駆動型再演モデルをオブジェクトごとに分散したデータ構造により実現すると，指定されたブレークポイントに到達するために必須な命令のみを再現できる，Instant Repaly にはない利点が得られる．この実現法で実行時に記録しなければならないデータ量は，この利点のある再演法の中では最も少なく，また，この利点がない再演法と比べても同程度以下であるので，この再演法が最も優れていることを示す．さらに，バス結合共有メモリ型マルチプロセッサ上に実現した再演システムでの評価結果から，上記再演法は，実行監視時にプログラム性能に対して比較的小さな影響しか及ぼさず，また，再演に必要なメモリ量も許容範囲に納まる，実用性の高い再演システムを与えることを示す．

Partial Replay of Parallel Programs Based on Shared Objects

Naohisa TAKAHASHI
NTT Software Laboratories
3-9-11 Midoricho, Musashino-shi, Tokyo 180 Japan

Atstract　The Instant Replay proposed by LeBlanc and Mellor-Crummey reproduces execution behaviour of parallel programs based on shared objects, which allows cyclic debugging techniques to be applied. This paper presents various variations of the Instant Replay which are represented as the combinations of three replay models based on dataflow execution models and three essential data structures. This paper shows that the combination of demand-driven replay model and data structure dispersed among objects has such advantageous features that it can reproduce only requisite instructions to reach a specified breakpoint. The comparison of all the combinations leads us to conclude that the above combination is superior to other combinations because it requires the least amount of data to be recorded during execution among all the combinations with the advantageous features and because it requires less or nearly equal amount of recorded data than other combinations without the advantageous features. A prototype of replay system, which has been implemented on a shared memory multiprocessor system, demonstrates that the above combination has only minor impact on program performance during program execution and has reasonable storage requirements.

1 はじめに

逐次形プログラムのデバッグでは，トレース条件などを変えながらプログラムを繰り返し実行させて実行動作を調べる会話的でサイクリックな手法が多く用いられている[1]．各サイクルでは，プログラマは，ステップ実行機能やブレークポイント機能を用いてプログラムの中断，状態変更，再開を繰り返し，注目点を少しずつ進めながら実行状態の変更と解析を行なう[1]．

逐次型プログラムでは多くの場合プログラムに同じ入力を与えて再実行させると同じ振舞いをする（再演性があるという）と期待できるので，サイクリックなデバッグ手法が有効となる．また，プログラムの実行を再現させるとき注目地点で実行を中断させると，その地点に至る経路の命令系列だけが再現され他の命令の実行は抑止されている（部分再演性があるという）と期待できるので，実行状態に変更を加えながら注目地点を段階的に進める手法が有効になる．

並列分散プログラムでもサイクリックデバッグを行なうためには，プログラムに再演性や部分再演性があることが望まれる[2]．しかし，メッセージや共有データを介して互いに通信するマルチプロセスからなる一般的な並列分散プログラムでは，メッセージの受信順序や共有データへの排他制御操作の振舞いが非決定的となるので通常再演性がない．このため，実行中にすべてのメッセージの到着順序や内容を記録し，その記録を使ってプログラムを再現する手法が提案されている[3]（本論文では，再演性のないプログラムの実行動作を記録し，その記録を用いてプログラムの再実行動作を制御して再演性があるように見せる手法を再演法と呼び，プログラムに部分再演性があるように見せることができる再演法はプログラムに部分再演性を与えるという）．上記の再演法は，記録に要する時間やメモリ量が多いので，通信コストが高く通信頻度が比較的低い場合にはじめて有効であり，一般に通信処理時間が短く通信頻度が高い共有データを用いた通信の場合には適さない．

CarverらのP-sequence法[4]とLeBlancらのInstant Replay法[5]は，データ共有型並列プログラムに再演性を与える効率的な手法である．P-sequence法では，共有データへのアクセスをすべて逐次化し，共有データにアクセスしたプロセスの識別子(Pid)の系列を一本の履歴テープに記録する．このため，記録データ量は少ないが，モニタ時および再演時に逐次化操作が加わりボトルネックが生じる．

Instant Replay法では，共有オブジェクトごとにその書き込み系列をバージョン番号で表わし，プロセスが共有データにアクセスするとプロセス毎に持つ履歴テープにバージョン番号を記録することによりボトルネックを回避している[5]．この再演法では，複数のプロセスの同時読み出しを許すCREW(Concurrent-Read-Exclusive-Write)アクセスプロトコル[6]を用いたプログラムを再演法の対象にし，各オブジェクトへの読み出し／書き込み系列を半順序関係で表わす．システムは，プロセスが読み出すオブジェクトのバージョン番号がモニタ時と再演時で等しくなるように，プロセス履歴テープを使ってオブジェクトへのアクセスの順序を制御する．

LeBlancらは，Instant Replay法で記録に要する時間がプログラム実行全体の１％以下であったという実験例に基づき，共有データ型並列プログラムに対しても再演法が実際的であることを示した[5]．しかし，彼らが実験で用いた並列計算機ではメモリへのアクセス速度がプロセッサ間で一様でないので，各プロセッサから（恐らく他のプロセッサノードにある）共有データへアクセスする時間が自ノードのデータの場合の数倍程度になる．従って，この実験結果からは，メモリへのアクセス速度がプロセッサ間で一様で共有データのアクセスをより高速に行える（その結果，共有データアクセスを監視するために生じる実行時間の増加がより深刻な問題になる）バス結合型マルチプロセッサでも再演法が実用的な手法になり得るとはいえない．

また，P-Sequence法やInstant Replay法などの従来の再演法は部分再演性を与えないので，プロセスがある地点に到達したときにプログラム（すなわち全プロセス）の実行を中断させると，他のプロセスの状態は非決定的となる．注目地点に到達したプロセスの実行だけを中断させると他のプロセスは先に進んでしまう．このため，プログラマは逐次型プログラムと同様な意味で注目点を少しずつ進めながら実行状態を調べることはできない．

一般に，プログラムに部分再演性を与えるように再演法の制御性を高めると，モニタモードで記録すべきデータ量が増えて実行効率が低下したり，再演時の実行効率が低下するなどの問題が生じると予想される．逆に，P-Sequence法のように，共有オブジェクトへのアクセスを逐次化して再現してよい場合には，モニタモードで記録するデータ量を減らし，かつ，再演時の実行制御を簡単化できると期待される．従来のプログラム再演の実現技法の提案では，このような再演時の制御法とモニタ時に記録するデータ量との関係について議論されていない．また，再演時に必要な制御を行なうためにモニタ時に記録すべき必要最小限のデータは何かについて十分に議論されていない．

本論文の第１の目標は，プログラム再現時の実行動作の制御法と履歴テープの構成法の２つの観点から再演法の実現技法を比較考察し，最適な再演法を選択する判断基準を与えることである．第２の目標は，バス結合メモリ共有型マルチプロセッサに対して，モニタ時にプログラム性能に小さな影響しか及ぼさずに部分再演性をプログラムに与えられる実用性の高い実現手法を与えることである．このため，以下では，データフロープログラムグラフの実行モデル[7,8]に基づき再演法をモデル化し，指定された事象の再現に必須な命令だけを再現できる（プログラムに部分再演性を与える）要求駆動型再演，プログラム全体を並列に再現できるデータ駆動型再演，プログラムの再現時に逐次化操作を加えて制御を簡単化する逐次型再演の３つの再演モデルを提示する．次に，各再演モデルについて履歴テープの構成法に従い実現技法を分類し比較議論する．さらに，バス結合共有メモリ型

ロトタイプを実現し簡単な評価を行なった結果について述べる．

2　プログラム再演法のモデル化

プログラムの再演法は，モニタ時に記録したプロセス間のインタラクション（データ依存関係）を再現するようにプログラムの再実行時の並列動作を制御する．プログラムに部分再演性を与える再演法では，あるプロセスの任意の地点が与えられたとき，その地点に到達するために必須な命令系列だけを再現できなければならない．このような再演法と部分再演性を与えない再演法でのプログラムの実行制御の違いを明確にするには，データフロープログラムの要求駆動型とデータ駆動型の実行モデルの比較議論[7,8]が参考になると考え，本章ではプログラムの実行動作をモデル化し，データフローモデルを用いた再演法の抽象モデルについて述べる．

2.1　並列プログラム実行動作のモデル化

プログラムの再演法を議論するためには，まず，再演可能なプログラムとその実行環境を明確に定めなければならない．LeBlancらの論文[5]はこのための優れた指針を与えている．本論文では，この指針に従い，再演対象とするプログラムと実行環境に対して彼らの Instant Replay法と同等な次のような仮定をおく．

（１）モニタモードから再演モードに移ってもプログラム動作に影響を与える外部環境は変わらない．また，再演モードで実行を繰り返しても変わらない．これは，逐次形プログラムを繰り返し実行させてデバッグするときに，通常プログラマが期待する仮定に対応する．

（２）複数のプロセスが共有オブジェクトの読み出し／書き込み命令を介して通信する並列プログラムを対象にする．オブジェクトへのアクセスはCREWプロトコルに従い逐次化する．再演法では，共有オブジェクトへのアクセス，プロセスの親子関係における非決定動作を再現することを保証し，プログラムに依存した，その他の非決定的な事象はランタイムライブラリなど再演法以外の他の適当なレベルの機能で再現を保証する．

これらの仮定の下では，再演法は，プロセスの親子関係と共有オブジェクトを介したプロセス間のインタラクションを正しく再現する問題と捉えられる．プロセスの生成消滅は，共有オブジェクトへのアクセスに比べ一般に頻度は極めて低く，かつ，処理時間が長いので，プログラムの実行に影響をあまり与えずにプロセスの親子関係を記録できる．従って，共有オブジェクトへの演算系列の記録と再現の実現法が重要になる．

2.2　データフローモデルを用いた再演法の抽象モデル

図１は，再演法の機能を抽象化して表現したモデルである．モニタモードでは（非決定的な）プログラムを実行させ，実行時の全振舞いを表わす（決定的な）仮想データフローグラフVDGを作る．再演モードでは VDGを解釈実行してプログラム実行を再現する．VDGは決定的であるので，再演モードを繰り返せば同じ動作を繰り返し再現できる．

モニタモードでは，各プロセスが実行した命令系列を保存し，共有オブジェクトの読み出し／書き込み命令とプロセス生成／消滅命令が出現すると，保存した命令系列をVDGのノードとして分離する（分離した命令系列をそのノードの命令という）．各オブジェクトへのアクセスの半順序関係は，読み出し／書き込み命令によりできるノード（read／writeノードと呼ぶ）間のアーク（IPアークと呼ぶ）で表わす．プログラムカウンタで制御する実行命令系列の全順序関係は，各プロセスでのノード作成順を表わすアーク（PCアークと呼ぶ）で表わす．

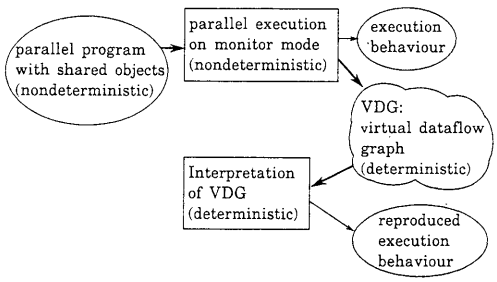

図１　プログラム再演の抽象モデル

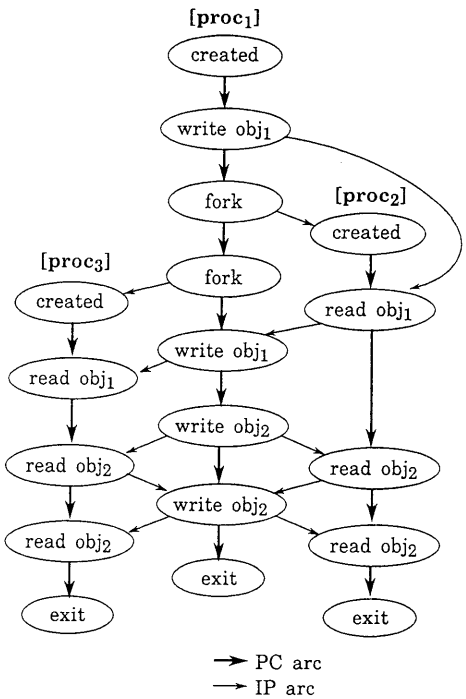

図２　仮想データフローグラフの例

プロセスの生成／消滅はUNIXのfork／exitシステムコール[9]と同様なOS機能を用いて行なう．これらのシステムコールの出現時に作るノードをforkノード，exitノードと呼ぶ．fork後，子プロセスが最初の命令を実行するまでの状態をcreatedノード（空の命令系列のノード）で表わすと，VDGは図２に例示するようにシステムがプログラム実行のため最初に作るプロセス（親プロセスと呼ぶ）のcreatedノードを初期ノードとする有向グラフとなる．プロセスの親子関係は，forkノードとcreatedノードを接続するアークで表わされる．

UNIXのforkなど多くのOSのプロセス生成では，子プロセスに付与する識別子は，そのプロセスが存在している間の一意性を保証するだけであり，プログラムを繰り返し実行させたときに同じ値になる保証はない．これに対して，VDGのforkノードとcreatedノードでは，ひとつのプログラムが実行中に生成するプロセス間で一意性を保証し，かつ，同じforkノードを実行させると毎回同じ値となる識別子を子プロセスに付与するものとする．これは，システムがプロセスの順序を表わす逐次番号を管理し，fork命令が現われたとき，この番号をプロセス識別子として与えれば実現できる．

以下の各章でVDGを参照し議論できるように，表１にVDGに関する記号の定義をまとめる．

表１　VDGに関する記号定義

symbol	meaning
Pid	process identifier
Oid	shared object identifier
v_{oid}	version number of shared object Oid
u_{oid}	serialization number of accesses to shared object Oid
t	serialization number of shared object accesses
s_{pid}	serialization number of nodes ececuted by process Pid
n	number of nodes executed by a program (number of nodes in VDG)
r/w	read node if r/w＝r, write node if r/w＝w and others otherwize (defined in each node)
a	number of input arcs (defined in each node)

3　要求駆動型再演モデル

要求駆動型実行では，命令実行の契機を命令の結果を必要とする命令側からの要求によって与え，不要な命令を抑止する．このため，必要最小限の命令だけを実行させることができるが，要求伝播機構の実現コストと要求伝播による実行効率低下が問題になる[8]．このモデルでは，モニタ時に記録するデータ量の少ない要求駆動制御法を与えることが重要である．

3.1　要求駆動型再演の必須データ

要求駆動型実行モデルでは，データフローグラフのあるノードの結果が必要になると，そのノードの入力アークの先のノード（以後単に入力ノードと呼ぶ）に要求トークンを送って実行を要求する．図２に例示したVDGで$Proc_3$のobj_1への読み出しノードを実行させるために要求トークンが送られた場合を考える．図３で点線矢印につながるノードは，この矢印に従って要求トークンを伝播させる．要求トークンがグラフの初期ノードまで到達すると初期ノードの実行を開始する．各ノードは実行を終えると，すべての出力アークの先のノード（出力ノード）に信号トークンを送る．要求トークンを受けたノードはすべての入力ノードから信号トークンを受けると実行を開始する．

要求駆動型実行モデルでは，論理的にはすべてのノードが並列に実行契機を制御する．ここでは，実際的な実行モデルとして，モニタモードでのプロセス数と同数のプロセスを作り，各プロセスにPidを割り付けて同じPidを持つすべてのノード（そのプロセスのノードと呼ぶ）の実行を次のように制御させるモデル(要求駆動型再演と呼ぶ)を考える．

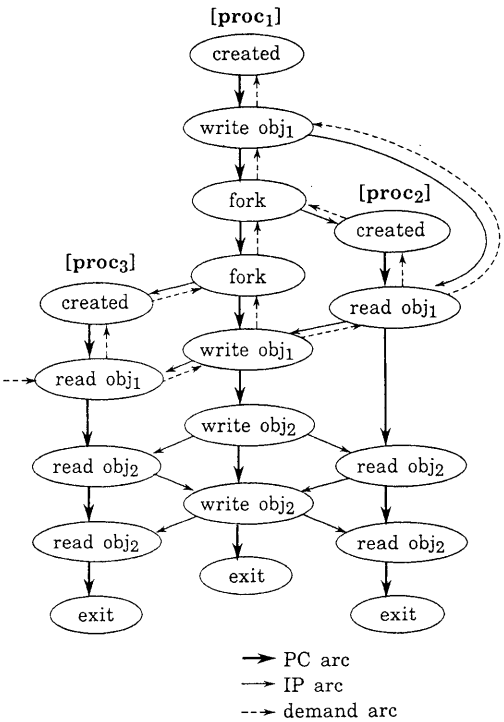

図３　VDGの要求駆動型解釈実行

各プロセスは自分のノードを取り出して，前述のよう

にトークンの送出とノードの実行を行なう．各プロセスが実行するノードの系列は，モニタモードでプログラムが逐次的に実行した命令系列をプロセス間インタラクション命令（IP命令という）で区切ってでできたものである．このため，各プロセスはIP命令が現われるまで元のプログラムを実行すると，１ノード分の命令を実行したことになる．

上述のようにプロセスがノードを逐次的に実行するとし，さらに自分のノードに送られた要求トークンの数を覚えておくとすれば，PCアークを介してトークンを往復させなくても良くなる（具体的な制御法は3.2，3.3に示す）．従って，各プロセスが自分のノードを取り出し，そのノードのIPアークを知ることができれば，要求駆動型実行モデルに基づいて，ノードの実行契機を与えられる．モニタモードで各ノードに（Pid，Oid，v_{Oid}，r/w）を記録すると，再演時にノードの（Oid，v_{Oid}，r/w）を調べるとIPアークを求められるので，（Pid，Oid，v_{Oid}，r/w）が要求駆動型再演の必須データとなる．以下では，VDGのアークのうちIPアークにだけ注目し，入力ノード，出力ノードという場合にはIPアークを介したノードだけを表わす．また，入力IPアークの根側を順々に辿ることにより得られるノードの集合を先行ノード集合と呼ぶ．

3.2 要求駆動型再演制御

再演時には，各ノードが要求トークンを受けたこと，および，信号トークンを送ったことをそれぞれノードごとに設けた変数demand，signalに記録する（各変数は値１のときとき事象がすでに生じたことを示す）．各プロセスは自分のノードに送られた要求トークン数を表わす変数d_{Pid}を管理する．

再演開始時に，システムは全プロセスのd_{Pid}，および全ノードのdemandとsignalを０に初期化する．プログラマが注目するノードを指定すると，システムは，そのノードおよび先行ノード集合のノードに要求トークンを送る．ここで，ノードに要求トークンを送るとは，ノードのdemandを１にし，プロセスPidのd_{Pid}を１増加させることを意味する．

プロセスPはd_Pが正になると，プログラムの実行を開始する．１ノード分の実行を終えるとd_Pを１減じ，０になったならば要求トークンが送られるまで待機し，d_Pが正のままならば実行を継続する．

3.3 共有データへのアクセスの要求駆動型再現制御

CREWプロトコルでオブジェクトへの排他制御を行なう．CREWプロトコルは，読み出しプロセス用手続き（図４）と書き込みプロセス用手続き（図５）からなる．図４，５で，LOCK()，UNLOCK()は，オブジェクトへのアクセスを同時にはただ一つだけ許す排他制御操作である．各手続きはそれぞれ図に示すように入口処理と出口処理の二つのルーチンからなる．システムはオブジェクトのバージョン番号と読み出しアクセス中のプロセス数を管理する．モニタモードでは，各手続きにより，CREWプロトコルを保証するようにアクセスを逐次化する．各手続きの入口処理では，オブジェクトへのアクセスを許されると，必須データ（Pid，Oid，v_{Oid}，r/w）をノードのデータとして記録する．

```
ReaderEntry(object, process);
  if mode = MONITOR then
    LOCK(object.lock);
    object.activeReaders = object.activeReaders + 1;
    WriteNode(process.id, object.id, object.version, r);
    UNLOCK(object.lock);
  else
    LOCK(object.lock);
    myNode = FindMyNode(object, process);
    myWriter = FindMyWriter(object, myNode);
    SendDemandsfromReader(object, myNode, myWriter);
    if (object.currentWriter = myWriter) then
      UNLOCK(object.lock);
    else
      UNLOCK(object.lock);
      delay(process, object);
    end if;
  end if;
end ReaderEntry;

ReaderExit(object, process);
  if mode = MONITOR then
    LOCK(object.lock);
    object.activeReaders = object.activeReaders − 1;
    UNLOCK(object.lock);
  else
    LOCK(object.lock);
    object.Readers = object.Reader − 1;
    if ((object.Readers = 0) and (object.nextWriter ≠ NULL)) then
      ResumeNextWriter(object);
    end if;
    UNLOCK(object.lock);
    done(process, object);
  end if;
end ReaderExit;
```

図４　要求駆動型再演モデルにおける読み出し手続き

```
WriterEntry(object, process);
  if mode = MONITOR then
    LOCK(object.lock);
    while object.activeReaders ≠ 0 do
      UNLOCK(object.lock);
      sleep(process);
      LOCK(object.lock);
    end do;
    WriteNode(process.id, object.id, object.version, w);
  else
    LOCK(object.lock);
    myNode = FindMyNode(object, process);
    SendDemandsfromWriter(object, myNode);
    if ((myNode ≠ object.nextWriter) or
      (object.Readers ≠ 0)) then
      UNLOCK(object.lock);
      delay(process, object);
    else
      UNLOCK(object.lock);
    end if;
  end if;
end WriterEntry;

WriterExit(object, process);
  if mode = MONITOR then
    object.version = object.version+1;
    UNLOCK(object.lock);
  else
    LOCK(object.lock);
    object.currentWriter = object.nextWriter;
    object.nextWriter = FindNextWriter(object);
    object.Readers = NumberofReadersforCurrentWriter(object);
    if (object.Readers = 0) then
      if (object.nextWriter ≠ NULL) then
        ResumeNextWriter(object);
      end if;
    else
        ResumeReaders(object);
    end if;
    UNLOCK(object.lock);
    done(process, object);
  end if;
end WriterExit;
```

図５　要求駆動型再演モデルにおける書き込み手続き

再演時には，システムは，各オブジェクトについて，

現在の値を書き込んだノード，その値を読み出すノード数，次に値を書き込むノードをそれぞれ表わす変数currentWriter，Readers，nextWriterを管理する．書き込み手続きの出口処理がO_{id}，v_{oid}，r/wの値を調べてこれらの変数を求めて書き込む．以下に再演モードでの各ルーチンの動作を述べる．各出口処理を終えると1ノード分の命令を実行したことになり，前節で述べたようにd_Pを1減じ，その値に従って実行の待機か継続か決める．

以下で，あるノードをresumeするとは，そのノードがアクセスを終えたオブジェクトへのアクセスをあるプロセスが遅延させているならば，そのプロセスへ信号を送り実行を再開させることを意味する．

（1）読み出しアクセス入口処理（ReaderEntry）

プロセスPがオブジェクト0の読み出し命令を出すと，Pid=P，Oid=0，r/w=r，signal=0のノードでv_{oid}が1番小さいノード（myNodeと呼ぶ）を取り出す．次に，入力ノード（myWriter）を見つける．myWriterの先行ノード集合のノードとmyNodeについて，各ノードのdemandが0のとき，要求トークンを送る．オブジェクトのcurrentNodeがmyWriterならば，読み出す値がオブジェクトにすでに書き込まれているので，その値を読み出す．そうでないならば，このオブジェクトへのアクセスを遅延させる．

（2）読み出しアクセス出口処理（ReaderExit）

オブジェクトのReadersを1減じ，正のままならば1ノード分の実行を終える．Readersが0になり，nextWriterが存在するならばnextWriterをresumeする．

（3）書き込みアクセス入口処理（WriterEntry）

プロセスPがオブジェクト0の書き込み命令を出すと，Pid=P，Oid=0，r/w=w，signal=0のノードでv_{oid}が1番小さいノード（myNode）を取り出す．myNode，および，その先行ノード集合のすべてのノードについて，demandが0のとき，要求トークンを送る．そのオブジェクトのnextWriterがmyNodeでないか，Readersが0でない場合には，これらの条件が成り立つまでオブジェクトへのアクセスを遅延させる．二つの条件が成り立つとき，オブジェクトへの書き込みを行なう．

（4）書き込みアクセス出口処理（WriterExit）

アクセスしたオブジェクトのcurrentWriter，Readers，nextWriterの値を更新する．書き込み値を読み出すノードがない（Readers＝0）場合には，nexrtWriterが存在するならば，nextWriterをresumeする．そうでない場合には，書き込み値を読み出すノードをすべて（Pid=0，r/w=rでv_{oid}が等しいすべてのノード）resumeする．

4　データ駆動型再演モデル

データ駆動型実行モデルでは，データフローグラフのあるノードのすべての入力ノードの実行が終わり信号トークンを受けると，その実行を開始する．各ノードは実行を終えると，すべての出力ノードに信号トークンを送る．このように，データ駆動型実行では，すべての命令はその実行に必要なオペランドデータが揃った時に実行させる．このため，簡単な実行機構でプログラムの持つ並列処理性を最大限抽出できるが，場合によっては不必要な命令まで駆動されてしまうことがある[8]．

要求駆動型再演と同様に，再現時には，モニタモードでプログラムが作ったのと同数のプロセスを作り，各プロセスにPidを割り付けて同じPidを持つすべてのノード（そのプロセスのノード）の実行を次のように制御させるモデルをデータ駆動型再演と呼ぶ．各プロセスは自分のノードをVDGから取り出して，信号トークンを調べて，全ノードから信号トークンがすでに送られていればノードの命令を実行し，出力ノードに信号トークンを送出する．このとき，要求駆動型再演で述べたように，各プロセスはノードを実行するときにはIP命令が現われるまで元のプログラムを実行すれば良い．

各プロセスが自分のノードを取り出し，そのノードの入力ノードの実行が終了したか調べられれば，データ駆動型実行を実現できる．あるいは，入力ノードにアクセスできなくても，入力ノード数と実行終了した入力ノード数が分かれば良い．前者は，IPアークを知ることを意味し，要求駆動型再演モデルと同様になり，必須データは（Pid，Oid，v_{oid}，r/w）となる．後者は，システムが再演時にも各オブジェクトのバージョン番号，および，そのバージョンのオブジェクトに対して出された読み出しアクセスの数を管理すれば，必須データは（Pid，s_{pid}，v_{oid}，a）となる．ここで，（Pid，s_{pid}）はプロセスが自分のノードを取り出すために必要なデータである．プロセスは，s_{pid}を1ずつ増しながらPidが等しいノードを順に取り出し実行すると，モニタ時のプログラムの実行命令系列を再現できる．

5　逐次化再演モデル

VDGの2つのノードに順序関係がない場合にはこれらのノードを任意の順序で実行させても実行結果と共有オブジェクトの値には影響を与えない．この場合には，ひとつの共有オブジェクトへのアクセスの半順序関係にあるノードを任意に順序付けしてできるただ一つの系列だけを記録して，その系列を逐次的に再現すればよい．プログラム再演をVDGの解釈実行としてモデル化するとき，これは，実行前にデータフロープログラムを逐次化してしまい実行コードと実行機構を簡単化すること対応する．

VDGで半順序関係をなす任意の部分グラフに属するノード集合に対して，任意の順序付けを行なうことによりできるデータフローグラフをデータ駆動実行させることを逐次化再演と呼ぶ．逐次化再演においても，次のように逐次化するとモニタ時には逐次化操作によるボトルネックの発生を避けられる．

CREWプロトコルでは，各オブジェクトに対する読み出しと書き込みは同時には行えないので，各オブジェクトへのアクセスを逐次化する部分がある．この逐次化操作の中で，各オブジェクト毎に逐次番号u_{oid}を管理すれば，モニタモードで新たに逐次化操作を加えなくて済ませられる．この場合には，各ノードに（Pid，Oid，u_{oid}）を記録し，Oidが等しく，u_{oid}の値が1だけ異なるノードの対を，u_{oid}の値が増す方向にアークで接続したデータ

フローグラフに対してデータ駆動型再演を適用する．

6　プログラム再演の実現技法の比較

履歴テープに書き込む単位データをentityと呼び，VDGのノード対応にentityを作ると考える．履歴テープでのentityの位置情報から必須データの一部を正しく推定できる場合があるので，履歴テープの構成によりentityの内容は異なる．たとえば，プロセス毎に履歴テープを設けると，各entityにPidを書き込む必要がなくなる．

以下の各節では，各再演モデルについて，履歴テープをシステム全体で1本とする（SHT(Single History Tape)法），プロセス毎に分割する（PHT(Process History tape)法），および，オブジェクト毎に分割する（OHT(Object History Tape)法）場合のentityの構成とプログラムの動作制御の違いを比較考察する．なお，以下の議論では表1の記号を用いる．

6.1　要求駆動型再演の実現技法

要求駆動型再演では，（Pid, Oid, v_{Oid}, r/w）が必須データである．

（1）SHT法による実現

すべてのプロセスは，システム履歴テープにentityを作る．共有オブジェクトへのアクセスは逐次化され，すべてのentityは作成順に履歴テープ上に整列する．このため，履歴テープでOidが等しいentityを辿りr/w=wのentityを数え上げればv_{Oid}を求められる．従って，（Pid, Oid, r/w）をentityに書き込めばよい．

（2）PHT法による実現

各プロセスは，自分のプロセス履歴テープにentityを作る．各プロセスのentityは作成順に各プロセス履歴テープに整列する履歴テープとPidは一対一対応であるので，各entityにPidを書き込む必要はない．よって，（Oid, v_{Oid}, r/w）をentityに書き込めばよい．

（3）OHT法による実現

各オブジェクトのentityはアクセス順に各オブジェクト履歴テープに整列するので，書き込みアクセスのentity（r/w=w）に番号付けすればv_{Oid}を得る．また，オブジェクト履歴テープはOidと対応付けられる．よって，OHT法ではentityに（Pid, r/w）を書き込めば良い．

要求駆動型再演の実現技法の特徴を表2にまとめる．

6.2　データ駆動型再演の実現技法

データ駆動型再演の必須データは(Pid, Oid, v_{Oid}, r/w) または（Pid, s_{Pid}, v_{Oid}, a）である．

（1）SHT法による実現

要求駆動型再演の実現と同様に，entityに（Pid, Oid, r/w）を書き込めば必須データ（Pid, Oid, v_{Oid}, r/w）を得られる．

（2）PHT法による実現

プロセス履歴テープはPidと対応付けられ，テープ内のentityの順番がs_{Pid}を表わすので，必須データ（Pid, s_{Pid}, v_{Oid}, a）のうち，entityに（v_{Oid}, a）を書き込めばデータ駆動型再演を実現できる．ここで，readノードの入力アーク数は常に1であるので，readノードのentityからはaを取り除くと，Instant Replay法[5]となる．

（3）OHT法による実現

要求駆動型再演の実現と同様に，entityに（Pid, r/w）を書き込めば必須データ（Pid, Oid, v_{Oid}, r/w）を得られる．

データ駆動型再演の実現技法の特徴を表3にまとめる．

表2　要求駆動型再演の実現技法の特徴

method	history tapes		centralized bottleneck		partial replay
	number of entities	entity	monitor mode	replay mode	
SHT	n	[Pid, Oid, r/w]	exist	exist	possible
PHT	n	[Oid, v_{Oid}, r/w]	none	none	possible
OHT	n	[Pid, r/w]	none	none	possible

表3　データ駆動型再演の実現技法の特徴

method	history tapes		centralized bottleneck		partial replay
	number of entities	entity	monitor mode	replay mode	
SHT	n	[Pid, Oid, r/w]	exist	exist	impossible
PHT	n	[v_{Oid}, a]	none	none	impossible
OHT	n	[Pid, r/w]	none	none	impossible

6.3　逐次化再演の実現技法

逐次型再演ではの必須データは（Pid, Oid, u_{Oid}）である．

（1）SHT法による実現

SHT方式では，テープが1本であるのでモニタ時に必ず逐次化してしまう．モニタ時の逐次化も許すとすると，VDGのread/writeノードすべてに逐次番号tを付与し，tの小さい順にノードを接続してできるグラフに対してデータ駆動型再演を適用すると，5章で述べた方法よりも簡単に逐次化再演を実現できる．tはシステム履歴テー

プでのentityの位置に対応するので，entityには（Pid）だけ書き込めば良い．これは P-Sequence法になる．

（２）PHT法による実現

PHT法によるデータ駆動型再演の実現においてａが常に１になると考えればよく，entityには（u_{Oid}）を書き込めば良い．

（３）OHT法による実現

履歴テープがOidを，entityの位置がu_{Oid}を表わすので，entityに（Pid）を書き込めば良い．

逐次化再演の実現技法の特徴を表４にまとめる．

表４　逐次化再演の実現技法の特徴

method	history tapes		centralized bottleneck		partial replay
	number of entities	entity	monitor mode	replay mode	
SHT	n	[Pid]	exist	exist	impossible
PHT	n	[u_{Oid}]	none	exist	impossible
OHT	n	[Pid]	none	exist	impossible

6.4　比較考察

表２～４に基づき再演法の実現技法を比較考察する．議論を簡単にするため，逐次番号を表わすPid，Oid，v_{Oid}，u_{Oid}，ｔのために必要なビット数はすべて等しいとする．また，ａはこれらの値に比べ小さくかつwriteアクセスに対してだけ付けられるので履歴テープの容量の計算では無視できるとする．r/wも１ビットで表現できるので無視できるとする．

要求駆動型再演では，SHT法とPHT法のテープ容量はOHT法の２倍になる（表２）．各実現技法についてデータ駆動型再演と要求駆動型再演でテープ容量を比較すると，SHT法とOHT法は変化せず，PHT法ではデータ駆動型再演の法が半分になる（表２，３）．すなわち，PHT法では，再演法に部分再演性を与えためにはテープ容量を２倍にする必要があり，他の２つの技法ではテープ容量は部分再演性には依存しない．

逐次化再演では，すべての手法のテープ容量が等しくなる（表４）．ただし，SHT法ではモニタ時にもボトルネック存在する．このときのテープ容量はOHT法で要求駆動型再演を実現した場合に等しい．

以上の比較結果から，再演時の制御の柔軟性とテープ容量の点でOHT法により要求駆動型再演を実現するのが最も良いといえる．しかし，どの実現法を選択するかは実用的な問題を考慮してさらに注意深く考察を加える必要がある場合がある．たとえば，プロセス数に比べてオブジェクト数が非常に大きな場合にはPHT法を用いた法がよい場合もある．この場合にOHT法で固定長のオブジェクト履歴テープを各共有オブジェクトに割り付けるとすると，断片化の影響が大きく現われ結果的に大量のメモリを要することになる．履歴テープの領域を動的に割り付ける方式をとると，断片化を避けられるがメモリ割り付けのオーバヘッドが問題になるので注意が必要である．モニタ時および再演時に逐次化を加えても良い場合には，このような断片化の問題を考慮する必要のないSHT法で逐次化再演を行なうのが最も良い．

７．要求駆動型再演システムの実現と評価

OHT法を用いた要求駆動型再演システムのプロトタイプをメモリ共有型並列計算機Sequent Symmetry上に作成した．現在，システムのデバッグと機能確認を終え，評価およびマンマシンインタフェースの改善のための拡張を開始した段階にある．本章では，システムの基本機能と簡単な評価結果について述べる．

7.1　再演システムの構成

我々の用いたSymmetryは８台のプロセッサと主記憶をシステムバスで結合した主記憶共有型の並列計算機である．各プロセッサは16MHz Intel i80386で，64Kバイトのキャッシュ（コピーバック方式）を持つ．各プロセッサは主記憶をすべてアクセスでき，主記憶へのアクセス時間は全プロセッサで一様である．OSはSymmetryの標準OS Dynixで，4.2 BSD UNIX相当機能と並列プログラミングライブラリPPLをサポートする[10]．

再演システムは，共有オブジェクトアクセスライブラリSO-Libと再演制御システムからなる．SO-Libは実行時ライブラリで，Ｃ言語で記述された関数群である．ユーザプログラムは，共有オブジェクトの初期化，共有オブジェクトへの読み出し／書き込みアクセス，プロセスの生成，プロセス識別子の読み出しのときには，SO-Libの関数を使う．それ以外は，DynixのシステムコールとPPLを利用する．再演制御システムは，再演時にプログラマから再演場所を指定され，要求トークンの伝播制御，および，プロセスとオブジェクトの状態を表示する．

7.2　共有オブジェクトへのアクセス操作の実現と評価

SO-Libにおける共有オブジェクトへのアクセス関数は，CREWに従った，読み出し時の排他制御操作r_entry，r_exit，書き込みの排他制御操作w_entry，w_exitからなる．r_exit，r_exit，w_entry，w_exitは，それぞれ3.3節で示したReaderEntry，ReaderExit，WriterEntry，WriterExitで，バージョン番号の計算部分を取り除き，各ノードに（Pid，Oid，v_{Oid}，r/w）を記録する代わりに，OHTに（Pid，r/w）を書き込むようにしたルーチンである．

共有オブジェクトの読み出し，書き込みに要する時間を測定し，プログラムの実行を監視しない場合の排他制御命令の実行時間と比べると表５のようになる．表５の数値は，それぞれ排他制御の入口処理と出口処理を１回

当りの実行時間である．表で(b)はSO-Libの対応する関数からモード判定，OHTへの書き込み操作，再演モードの全操作を取り除いた，CREWアクセスプロトコルにより排他制御を行なう関数である．(c)はDynixのPPL[10]の関数である．PPLの関数は同じオブジェクトへのアクセスはすべて排他制御する(ME: Mutually Exclusive)．

表5　共有オブジェクトへのアクセス操作ライブラリ

library	program execution	access protocol	access class to be serialized	function names	execution time in μsec
(a) SO-Lib	monitored	CREW	write	w_entry, w_exit	26.9
			read	r_entry, r_exit	31.1
(b)	unmonitored	CREW	write	w_lock, w_unlock	14.9
			read	r_lock, r_unlock	18.9
(c) Dynix PPL[10]	unmonitored	ME	read, write	s_lock, s_unlock	8.5

表から，CREWアクセスプロトコルをサポートするとき，実行監視機構を加えると，1回当りのアクセスで約12マイクロ秒実行時間が増加することが分かる．DynixのPPLの関数に比べると約20マイクロ秒実行時間が増加している．従って，実行監視のためにプログラムの実行時間が延びる割合を1％程度に抑えるためには，共有オブジェクトへのアクセス頻度が1プロセッサ当り1～2ミリ秒に1回程度であればよい．これは，プログラム全体に渡る平均値として要求される値である．一般に，メモリの割り付け，プロセスの生成，プロセスの切り替えなどに要する比較的大きな時間がプログラム実行の応答時間に加わるので，共有オブジェクトへのアクセスを比較的頻繁に行なうプログラムでも実効的なアクセス頻度はこの値以下になると期待できる．

OHT法では，共有オブジェクトへのアクセス1回につき1 entityのデータをOHTに書き込む．1entity当りのデータ量は1バイトである．排他制御命令だけを繰り返すような，現実には有り得ない極端な最悪ケースを考えると，OHTの容量は1バイト／25マイクロ秒／プロセッサ，すなわち，40Kバイト／秒／プロセッサとなる．共有オブジェクトへのアクセス頻度が1プロセッサ当り1～2ミリ秒に1回程度であるとすると，0.5～1Kバイト／秒／プロセッサとなる．

バス結合型マルチプロセッサでは，プロセッサの台数が増えると，アクセス衝突のため各プロセッサから共有オブジェクトへアクセスできる頻度が減少するので，上で求めた値から数Kバイト／秒程度のメモリ消費を考慮すれば実用上十分であると推定できる．この値は，10～20Mバイトのメモリがあれば，実行時間が1時間程度のプログラムをモニタできることを意味する．サイクリックにかつ会話的にデバッグするプログラムを再演法の対象とするとき，これらの値は十分許容できる範囲にある．

7.3　応用プログラムへの適用

Bob Beck らがDynixのパラレルプログラミングプロセスモデルに基づくプログラム例として用いた，素数の数を計算する簡単なプログラム[11]をとりあげ，実効監視機構が応用プログラムの性能に与える影響を考察する．

このプログラムは，プロセス数numprocs，最大素数制限値maxprime，きざみ幅biteの3変数を指定すると，1からmaxprimeの範囲の素数の数を計算して返す．numprocs個のプロセスが並列に動作し，maxprimeからbiteで指定されたきざみ幅で素数の数を調べていく．たとえば，numprocs=6，maxprime=1000，bite=10と指定すると，6個の子プロセスが作られて，991～1000の範囲の素数の数，981～990の範囲の素数の数，...，1～10の範囲の素数の数というように空いているプロセスが10（=bite）ずつの範囲で次々と素数の数を求め，それらの合計値をプログラムの値として返す．

共有オブジェクトとして，素数の値に対する現在の上限値sindex，および，現在までに計算した素数の数counterの2変数を使う（元のプログラムでは，sindexのみを共有変数としていたが，共有オブジェクトへのアクセス制御の影響がよく現われるように，counterも共有オブジェクトにした）．各プロセスは，sindexを読み出し，値が正ならばbiteだけ減じてsindexに書き込む．sindexの値が負ならば処理を終える．読み出した値までのbite個の整数が素数であるか判定し，素数の数をcounterに加え込む．

表6にnumproc=8，maxprime=1000000，bite=10000のときの実行時間（秒）を示す．表で，(c)は，元のプログラムに対してcounterを共有変数にし，プロセス生成をライ

表6　素数の数を求める簡単なプログラムの実行

	program execution	shared object access protocol	execution time in seconds
(a)	monitored (Symmetry machine, 8 CPUs)	CREW	27.81
(b)	unmonitored (Symmetry machine 8 CPUs)	CREW	27.80
(c)	unmonitored (Symmetry machine 8 CPUs)	ME (Dynix Library)	27.72
(d)	unmonitored (Balance machine 28 CPUs)	ME (Dynix Library)	34.5

ブラリを使わずにforkで行なうという２点の変更を加えたプログラムである．(a)，(b)は(c)のプログラムについて共有オブジェクトへのアクセスライブラリをそれぞれ表６の(c)から(a)，(b)へ変えたプログラムデバッグある．(d)は，文献［11］から引用した値で，Sequent Balance（28プロセッサ．OSはDynix）を使ったときの実行時間である．SymmetryはBalanceに比べ４～５倍程度性能向上が計られているので，プロセッサ数が1/4程度でも実行時間が短縮されている．

表６に示した実験結果では，実行時監視機構を加えても，0.1～0.5％程度しか実行時間は延びていない．この程度の影響はプログラマに十分許容されるものであると考える．また，SymmetryとBalanceでの測定結果の比較からも分かるように，ハードウェアの進歩による性能向上は目ざましく，実行時間が数％程度まで延びたとしてもプログラム作成が容易になるならば許容されるのではないかと考える．

8　おわりに

本論文では，複数のプロセスが共有オブジェクトへの読み出し／書き込み操作を非同期に行なう並列プログラムに対して，プログラムの実行動作を記録し再現させる技法（再演法）を比較評価した．はじめに，データフロー実行モデルに基づき再演法をモデル化し，指定された事象の再現に必須な命令だけを再現できる（部分再演性がある）要求駆動型再演，プログラム全体を並列に再現できるデータ駆動型再演，プログラムの再現時に逐次化操作を加えて制御を簡単化する逐次型再演の３つの再演モデルを提示した．

さらに，プログラム動作を記録するテープをシステム全体で１本にする（SHT法），プロセス毎に分割する（PHT法），オブジェクト毎に分割する（OHT法）３つの手法を示し，これらの手法を用いて各再演モデルを実現した場合の特徴を比較考察した．この結果，OHT法を用いて要求駆動型再演モデルを実現すると，他の手法を用いた場合よりも少ないテープ容量で部分再演性のある再演法を実現できることを明らかにした．また，この実現法のテープ容量は，他の手法を用いて部分再演性のないデータ駆動型再演や逐次化再演を実現する場合と同程度以下であることを示した．

OHT法を用いた要求駆動型再演システムをメモリ共有型並列計算機Sequent Symmetry上に作成し簡単な評価を行なった．この結果から，実行監視機構を加えることにより生じるメモリ容量と実行時間の増加は十分許容できる範囲に納まることを示した．

今後，共有オブジェクトをポートとして使いプロセス間でメッセージパッシングを行なう場合[12]，および，アドレス空間を共有する複数のスレッドが並列に動作する場合[13]に対しても適用できる，効率的な要求駆動型再演システムを作成する．また，大域的条件を考慮した分散プログラムのデバッグ手法[2]に組み込んで，分散／並列プログラムのデバッグ環境を作成する予定である．

謝辞　本研究の機会を与えられ，ご指導ご支援を頂いたソフトウェア基礎技術研究部塚本克治部長，武末勝主幹研究員に感謝します．今瀬真主幹研究員，真鍋義文研究主任には分散プログラムのデバッグ法に関する討論のなかで本論文に対して有益なご示唆を頂いた．記して感謝します．

参考文献

[1] Fairley, R., “Software Engineering Concepts,” McGraw-Hill Book Company, 1985.

[2] Manabe, Y. and Imase, M. “Global Condition in Debugging Distributed Programs,” *Tech. Rep. COMP89, IEICE,* 1989.

[3] Courtis, R. and Wittie L., “BugNet: A Debugging System for Parallel Programming Environments,” *Proc. 3rd Int. Conf. Distrib. Comput. Syst.*, pp. 394-399, 1982.

[4] Carver, R.H. and Tai, K., “Reproducible Testing of Concurrent Programs Based on Shared Variable,” *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, May 1986, pp.428-433.

[5] LeBlanc, T.J. and Mellor-Crummey, J. M, “Debugging Parllel Programs with Instant. Replay,” *IEEE Trans. Computer,* **C-36**, 4, pp.471-482, 1987.

[6] Courtois, P.J., Heymans, F. and Parnas, D.L., “Concurrent Control with Readers and Writers,” *Commun. ACM,* **14**, 10, pp.667-668, 1971.

[7] Davis, A.L. and Keller, R.M., “ Data Flow Program Graphs,” *IEEE Computer,* **15**, 2, pp.26-41, 1982.

[8] Amamiya, M. and Hasegawa, R., “Dataflow Computing and Eager and Lazy Evaluations,” *New Generation Computing,* **2**, pp.105-129, 1984.

[9] Leffler., S.J., McKusick, M.K, Karels, M.J and Suarterman, J.S., “The Design and Implementation of the 4.3 BSD UNIX Operating System,” Addison-Wesley Publishing Company, 1989.

[10] Sequent Computer Systems, Inc., “Guide to Parallel Programming on Sequent Computer Systems (Second Edition),” Prentice-Hall, 1989.

[11] Beck, B. and Olien D., “A Parallel Programming Process Model,” *Proc. Winter 1987 USENIX Tech. Conf.*, pp.84-102, 1987.

[12] Cheriton, D.R., “UIO: A Uniform I/O System Interface for Distributed Systems,” *ACM Trans. Computer Systems,* **5**, 1, pp.12-46, 1987.

[13] Tevanian, A., Rashid, R.F., Golub, D.B., Black, D.L, Cooper, E. and Young, M.W., “Mach Threads and the Unix kernel: The Battle for control,” *Tech. Rep. CMU-CS-87-149,* August 1987.