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

三栄 武† 高橋直久†

メモリ共有型並列プログラムは共有データへのアクセス動作が非決定的となるため，プログラムの動作理解やサイクリックなデバッグが，逐次型プログラムと比べて難しい．この問題を解決するため，適応的再演型ロック命令と呼ぶロック命令を提案し，これを用いた並列プログラムデバッガについて述べる．この命令は共有データへのアクセス順序を効率良く記録し，プロセスの動作を繰り返し再現することを可能にする．この時，要求駆動実行により，指定された命令の動作の再現に必要最小限な命令だけを実行させることができる．また，要求駆動実行とブレークポイントを併用した場合にも，ロック命令の適応的機能によりすべてのプロセスが決定的な位置で停止することを保証する．本稿では，メモリ共有型並列計算機上に作成したデバッガについて，その実現法とデバッグ例を示す．

An Implementation of Parallel Debugger with Adaptively Replayable Lock

TAKESHI MIEI † and NAOHISA TAKAHASHI †

Debugging parallel programs which consist of multiple processes communicating with shared data is difficult, because these programs are often nondeterministic. This paper presents a lock operation, called an Adaptively Replayable Lock (ARL), and a parallel program debugger which uses ARL. The ARL enables to reproduce the nondeterministic execution behavior of parallel programs by using a demand-driven replay method, and to halt the execution programs when it ensures that all processes cannot proceed. This paper describes the prototype implementation of the parallel debugger on a shared memory multiprocessor system, and shows examples of program debugging with the prototype debugger.

1. はじめに

並列プログラムでは，同じ入力を与えて実行させても必ずしも同じ動作をするとは限らない．このため，デバッグなどでプログラムの動作を再現させるには，並列プログラムの実行動作を記録し，その記録（実行記録と呼ぶ）に従って並列プログラムの実行を制御することにより，繰り返し同じ実行動作を得る手法（再演法と呼ぶ）が必要になる．これまでに，再演法として，P-sequence 法[1]や Instant Replay 法[2]など種々の手法が提案されている[7]．しかし，これらの提案では，主に再演制御について議論しており，再演法を用いたデバッガの実現法に関しては十分な議論がなされていない．

このため，本稿では，まず，再演法を用いたデバッガを実現する際に問題となる，最小限の再演実行の実現法，実行停止位置の設定法，再演停止法，デバッガの構成法について議論すべき点を提示する．次に，これらの問題点を解決するために，以下の特長を持つ並列プログラムデバッガモデルを提案し，さらに，このモデルに基づきメモリ共有型マルチプロセッサ計算機上に実現したデバッガについて述べる．

1. 静的なソースコード上と動的な実行履歴上の両方に対して停止位置を設定する手段を提供し，後者に到達するための必須命令だけ実行することを保証している．
2. すべてのプロセスが実行を進められない状態に到達したことを検出して，すべてのプロセスが決定的な状態で停止することを保証している．
3. 適応的再演型ロック命令と呼ぶ排他制御命令の機能の中に再演に関する実行制御機能を閉じ込めて実現することにより，プログラム言語やデバッガコマンドからの独立性を高めている．
4. 各共有データアクセスに対して，アクセス種別

† NTT ソフトウェア研究所
NTT Software Laboratories

(read/write を表す1ビット)とプロセス識別子だけを記録しておけばよいように,上記の再演と停止のための機能を実現しているので,従来よりも実行記録の作成がプログラムに与える影響を低下させている.

2. 並列プログラムデバッガ実現上の課題

2.1 最小限の再演実行

従来の多くの再演法では,プログラム実行で論理的に順序関係がある事象について,実行監視時に先に発生した事象は再演時でも先に発生させることを保証しているだけであるので,順序関係のない事象に対しては再演時に実行順序制御を行っていない.このため,ある命令の実行動作を再演しようとしたとき,その命令の前に実行しておくべき命令がすでに終了していれば,その命令を実行する.言い替えると,図1に例示するように,これらの手法では,あるプロセスがブレークポイント(BP)に到達して停止しても,他のプロセスは可能な限り実行を進めてしまう.このため,プログラマが新たにBPを設定しようとした命令が既に実行済みで,プログラムを始めから実行しなおさなければならないという問題が生じる場合ある.

図1は,2個の共有オブジェクトへアクセスする3個のプロセスからなる並列プログラムの実行動作を仮想データフローグラフ[10]と呼ぶ依存グラフで表している.ノードは再演性を保証するために記録すべき,順序関係を生じさせる命令(プロセスの生成,消滅命令,および共有オブジェクトへのアクセス命令)が実行されたことを表す.アークはノード間の実行順序を表し,プロセス内の命令の実行順序を表すPCアーク(図の太線矢印)とプロセス間でのノードの実行順序を表すIPアーク(細線矢印)からなる.この図では,プロセス2がread obj 1ノードを実行後にBPに到達した場合に,実行されるノードを網掛けで示している.

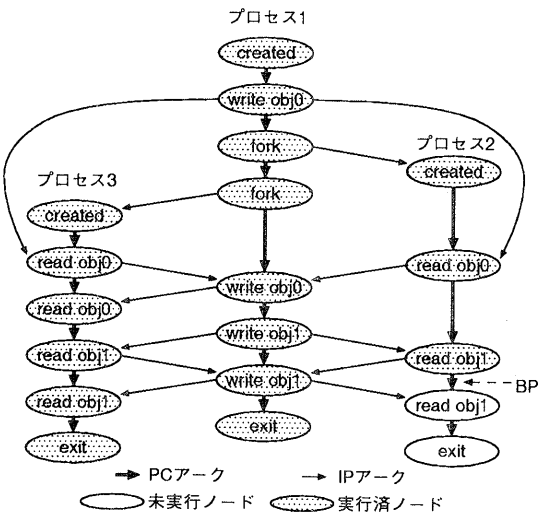


図1 従来の再演法による再演実行
Fig. 1 Re-execution using usual replay techniques.

すべてのプロセスに対して,それぞれ適切な位置に予めBPを設定しておくことにより,上の問題を回避できるが,一般にプロセス数が増大した場合には繁雑な作業となる.このため,あるプロセスがBPに到達して停止した場合には,デバッガは,そのプロセスが停止点に到達するために必要最小限の命令の実行だけを再演するように他のプロセスの実行を抑えることが望ましい.

2.2 実行記録上での停止位置設定

デバッグにおいてプログラマは,プログラムの実行停止位置を設定し,その実行動作の途中経過を観察する.通常のデバッガでは,BPにより静的なソースコード上に実行停止位置を設定する手段を与え,動的な実行動作上で停止位置を設定する手段は提供していない.このため,プログラマが実行動作上に停止位置を考えた場合には,その位置からソースコード上の位置を推察して,BPとして設定しなければならない.また,BPを設定した場合には,プロセスは再演時にそのBPに必ずしも到達するとは限らない.逆に,1つのBPに複数回到達する場合も生じる.さらに,複数のプロセスに対して,いずれかの条件が成立したときに停止するように条件付きBPを設定した場合には,各プロセスを最小限の実行だけで停止させることはできないことが示されている[3].一方,実行動作上に設定する停止位置ならば,その実行動作上に直接的に設定でき,再演時にプロセスが,ただ一度だけ到達することが保証される.通常のデバッガではプログラムの実行動作を保存していないため,実行動作上の停止位置の設定手段が課題となるが,再演法を用いたデバッガでは,これを実行記録として保存しており,これを利用して実行動作上に対しても直接的に停止位置の設定手段を提供することが望まれる.

2.3 再演停止法

サイクリックなデバッグでは,プログラマはすべてのプロセスを停止させた後にプログラムの状態の観察や変更を行うため,停止したプログラムの状態が常に決定的であることが望まれる.しかし,あるプロセスがBPに到達して停止した際に単純に他のプロセスを停止させると,停止位置が非決定的となる.従って,BPで停止したプロセスの発生により,もはやプログ

ラムの実行を進められない状態（安定状態と呼ぶ）にあることを保証できる位置に，全プロセスが到達してから再演を停止させる必要がある．さらに，上記の位置に到達したことをできるだけ早く検出することが望ましい．

2.4 デバッガの構成法

2.1節から2.3節で述べた，最小限な再演実行，実行記録上での停止位置設定や再演停止法の機能を実現した再演型デバッガシステムでは，すべてのプロセスの実行動作を常に制御しなくてはならない．これらの機能をすべてデバッガに組み込んで実現を図ると，デバッガがボトルネックとなり，プログラムの並列性を損なうとともにオーバヘッドが大きくなる恐れがある．また，プログラム言語やデバッガコマンドに強く依存して実現すると，デバッガシステムの移植性が低下する．よって，再演のために必要な機能と，通常のデバッガの機能をできるだけ分離するとともに，その要素機能が並列に動作できるよう実現することが望ましい．さらに，メモリ共有型並列プログラムでは分散プログラムなどに比べ高速な通信(共有データアクセス)が要求されるので，再演性を与えるために必要な記録データをできる限り小さくする必要がある．

2.1節，2.2節については3.1節で述べる．2.3節については3.2節で，2.4節については3.3節にて述べる．

3. 並列プログラムデバッガモデル

3.1 要求駆動を用いた再演実行

本稿で述べるデバッガでは，2.2節で述べたように，実行記録上に停止位置を設定する手段を提供し，この停止位置に到達するのに，2.1節で述べた，必要最小限の命令の実行だけを再現する機構を，先に提案した要求駆動型並列実行制御に基づく再演法[5]を用いて実現している．この手法では，まずプログラマが仮想データフローグラフ上のノードを指定して停止位置を設定する．この時，デバッガはプログラマが指定したノードの直後の実行位置を停止位置であると解釈し，その停止位置に到達するために必要最小限の命令の実行だけを再現する．これら再演する命令に対応するノードは，仮想データフローグラフにおいて停止位置のノードが与えられれば，そのノードからPCアークとIPアークを実行と逆向きにたどることにより決定できる．しかし，本手法では記録するデータ量を削減するために，実行監視時にPCアークを記録せず，IPアークだけを記録するだけで実行記録上の停止位置までの必要最小限の命令を再現する．すなわち，各プロセスが，そのプロセスのプログラムを実行することにより，ノードに対応する命令系列とPCアークの先のノードが得られるので，PCアークは再演時に抽出可能である．また，各プロセスにおいて，実行記録上の停止位置または，そこからIPアークを遡って到達可能なノードの命令は，上記必要最小限の命令に含まれると再演開始時に判定できる．一方，実行記録上の停止位置からPCアークとIPアークの両方を遡って到達可能なノードは，途中まで再演実行を進めることにより，上記必要最小限の命令に含まれると動的に決定可能である．これらのノードを仮想データフローグラフ上の，次の3種類のノードとして定義し，以下にこれらのノードを用いて再演実行の実現法について述べる．

- DP（デマンドポイント）： プログラマが実行記録上に設定した停止位置（指定したノードの直後の実行位置）
- 2次DP： DPの直前のノードからIPアークだけを遡って到達可能なノードのうち，各プロセスで最後に実行されるノードの直後の実行位置
- 3次DP： まず，DPまたは2次DPを持たないプロセスにおいて，DPの直前のノードからPCアークとIPアークを遡って到達可能であり，かつ後述の実行要求信号を受けたノードを考える．これらのノードの中でcreatedノードからの距離が一番長いノードの直後の実行位置を3次DPと呼ぶ．3次DPは，実行要求信号の受信状態に基づいて定まるので，再演実行の進行とともに変化する．

ここで，DPまたは2次DPを持つプロセスをDPプロセスと呼び，それ以外のプロセスを非DPプロセスと呼ぶ．

要求駆動型再演法では，命令の実行契機を，その実行結果を必要とする命令からの実行要求信号によって与える．例えば，**図2**に例示した実行記録の一部分を再演する場合，プロセス2のread obj 1ノードの実行には，プロセス1のwrite obj 1ノードの実行結果が必要である．そこでプロセス2がプロセス1のwrite obj 1ノードに実行要求信号を送ることによって，プロ

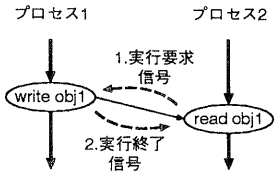


図2 要求駆動型再演法
Fig. 2 Demand driven-based replay technique.

セス1に，そのノードの実行契機を与える．この時，プロセス2のread obj 1ノードに対するプロセス1のwrite obj 1ノードのようなIPアークの根側のノードを先行ノードと呼ぶ．このような，DPと要求駆動型再演法を用いた本デバッガの実行制御法の概要を以下に述べる．

まずプログラマが，**図3**に示すように，DPを設定後，プログラムの再演を開始させる．デバッガはIPアークを用いて2次DPを求め，DPと2次DPノードに実行要求信号を送る．プロセスは，実行要求信号を受信したノードまでの実行を開始する．これは，PCアークだけを遡って実行要求信号を伝播させ，createdノードに到達後，実行を開始したのと同様の効果がある．プロセスは各ノードを次の手順に従って実行する．

1. 要求伝播：そのノードに対する実行要求信号を受信後，先行ノードならびにPCアークの根側のノードに実行要求信号を伝播させる．
2. 発火制御：入力側のすべてのPCアーク，IPアークから実行終了信号が送られるのを待機し，すべての信号が揃ったら，ノードを実行する．ここでのノードの実行とは，既にPCアークの根側のノードが表すread/writeなどの命令までが実行済みであることから，その次の命令から自ノードが表すread/writeなどの命令までの命令系列を逐次的に実行することを意味する．
3. 結果伝播：ノードの実行が終了した時点でIPアークならびにPCアークが指し示すノードに実行終了信号を送る．

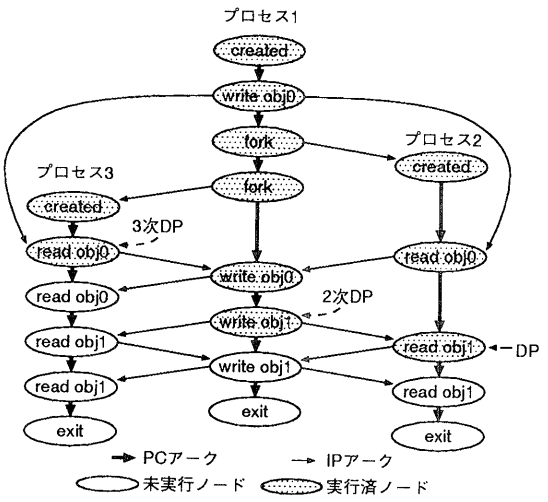


図3 要求駆動を用いた再演実行

Fig. 3 Re-execution using demand driven-based replay technique.

図3に，プロセス2のread obj 1にDPが設定された場合に再演実行されるノードを網掛けで示す．このようにして，本デバッガでは，各プロセスがDP，2次DP，3次DPまでの再演を行うことにより，DPに到達するために必要最小限の命令だけを再演する．

3.2 再演停止法

本デバッガでは，プログラマはDPとBPを用いてプログラムの実行停止位置を設定できる．この時，BPのみを用いた場合には，BPを設定したプロセスの最後の命令に，デバッガがDPを設定してDPとBPを併用していると見なせるので，本節では，DPのみを設定した場合，およびDPとBPを併用して用いた場合に，あるプロセスがDPまたはBPに到達した際のプログラムの停止法について述べる．これらの場合には，前節で述べた要求駆動による再演実行により，DPに到達する必要最小限の命令以外は再演実行されないことが保証される．次の2種類に場合分けをして議論する．

1：DPに到達したプロセスが存在する場合．これは，DPのみを設定し，あるプロセスがDPに到達した場合，および，DPとBPを併用したがBP到達するプロセスが発生せず，あるプロセスがDPに到達した場合を表す．

2：DPに到達したプロセスが存在しない場合．これは，DPとBPを併用し，あるプロセスがBPに到達したことを表す．

1の場合には，プロセスがDPに到達した時点で，要求駆動型再演法の性質[5]により，すべてのプロセスが実行を進められない実行位置にあることが保証できる．その時点で全プロセスを停止させれば，全プロセスの停止位置は決定的となる．2の場合には，一般に，BPにプロセスが到達後，十分長い時間が経つとすべてのプロセスがデータ待ちの状態になるので，その時点で全プロセスを停止させれば，全プロセスの停止位置は決定的となる．このような，すべてのプロセスが実行を進められない状態を安定状態と呼び，安定状態での各プロセスの実行位置を，そのプロセスのSPと呼ぶ．また，BPで停止したプロセスからの実行終了信号を待機するプロセスの実行位置を，そのプロセスの2次BPと呼ぶ．

実際のデバッグでは，プロセスのBP到達後に十分長い時間待機することは許容できない．また，待機せずに単純にすべてのプロセスを強制的に停止させると各プロセスが様々な実行位置で停止してしまい，必ずしも安定状態にならない．よって，できる限り早い時期に，すべてのプロセスがSPに到達したことを検出

し，安定状態で停止させる手段を提供する必要がある．

どのような実行位置をSPと見なせばよいか議論するため，まずプロセスの実行位置を次の3種類に分類する．

（1） BPと2次BP

（2） DP，2次DPと3次DP

（3） その他の実行位置

（1）の場合，BPはその定義より，2次BPは前節で述べた発火制御より，明らかにSPである．（3）の場合，プロセスの実行がそこまで到達し必ず次に実行が進むか，あるいは，実行要求信号が伝播する可能性がなく，プロセスの実行が到達しないかの，いずれかであるので，明らかにSPではない．（2）の場合，DP，2次DP，3次DPは，実行要求信号を与えられたノードの直後の実行位置であり，BPで実行を中断されない場合に各プロセスが到達すべき実行位置であるので，SPとなる可能性がある．DPと2次DPは，その定義より，プロセスが到達すればSPとなる．3次DPは，その定義から，実行要求信号を受信するごとに動的に位置が変化するので，新たな実行要求信号が到着しないことが保証された場合にのみ，SPとなる．よって，プロセスがBP，2次BP，DP，2次DP，または，新たな実行要求を受信する可能性のない3次DPのいずれかに到達した時，そのプロセスはSPに到達したと判定できる．

以下に，SPへの到達判定の実現法の詳細について述べる．再演中に，あるプロセスがDPへ到達した場合には，プログラムが安定状態にあること，およびDPへの到達に必要最小限の命令だけを再演したことが保証されるので，その時点でプログラムを停止すればよい．再演中にあるプロセスがBPに到達した場合には，各プロセスのSP到達を個々に判定する必要があるが，以下の手法を用いることにより，実行監視時にIPアークを記録しておくだけで，再演時に各プロセスが各々独立にSP到達を判定できる．

再演時に，プロセスは受信した実行要求信号の数（デマンド数と呼ぶ），IPアーク，DPプロセスか非DPプロセスかを示すフラグ（DPフラグと呼ぶ），BPや2次BP到達への到達を示すフラグ（BP到達フラグと呼ぶ）およびDPプロセスの総数（DPプロセス数と呼ぶ）を管理する．ここで，デマンド数は実行すべきノード数を表し，プロセスはノードを実行するごとにデマンド数を1減じる．各プロセスは，その種別に応じて次の処理を行い，SP到達を判定する．

DPプロセス：

・デマンド数が0の場合：DPまたは2次DPに到達したことが保証されるので，SPに到達したと判定し，DPプロセス数を1減じる．

・デマンド数が1以上の場合：先行ノードを持つプロセス（先行プロセスと呼ぶ）のBP到達フラグを調べ，BPまたは2次BPに到達しているならば，自プロセスも2次BPに到達したことが保証される．BP到達フラグを変更し，SPに到達したと判定し，DPプロセス数を1減じる．

非DPプロセス：

・デマンド数が0の場合：3次DPに到達したと判定できる．この時，DPプロセス数が0ならば，新たな実行要求信号を受信しないことが保証されるので，SP到達と判定できる．

・デマンド数が1以上の場合：先行プロセスのBP到達フラグを調べ，BPまたは2次BPに到達しているならば，自プロセスも2次BPに到達したことが保証される．BP到達フラグを変更し，SPに到達したと判定する．

図4は，プロセス1が2回目のwrite obj 0ノードを実行した後にBPに到達した場合に再演されるノードを網掛けで示している．プロセス1とプロセス2がDPプロセスであり，プロセス1はBPでSP到達となる．プロセス2はread obj 1ノードを実行する直前に，前述のDPプロセスのデマンド数が1以上の場合の条件が成立し，2次BP到達すなわちSP到達となる．プロセス3はread obj 0ノードを実行し終えた位置で，非DPプロセスのデマンド数が0の場合の条件により3次DP到達となり，さらにすべてのDPプロセスが

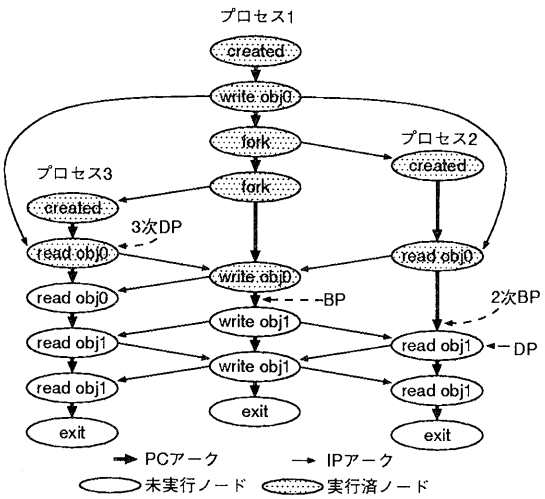


図4 DP，BPを用いた再演停止

Fig. 4 Halting the program using demand driven-based replay technique with DP and BP.

SP に到達していることから，SP 到達となる．

3.3 適応的再演型ロック命令

本節では，**図5**に示すように適応的再演型ロック命令と呼ぶ共有オブジェクトに対する排他制御命令の機能として，要求駆動再演機能と安定状態に基づく再演停止機能を実現する手法について述べる．

適応的再演型ロック命令は，実行時に並列プログラムの実行履歴を記録する実行監視モードと，実行履歴をもとに繰り返し再演を行う再演実行モードを持つ．再演実行モードでは，次の機能を実行する．

(1)要求駆動再演機能 3.1 節で述べた要求伝播，発火制御，結果伝播を行う．実行要求信号の伝播先プロセスが未生成の場合には，その親プロセスの fork ノードへも実行要求信号を送る．

(2)状態適応機能 2 次 BP への到達を検出するために，先に提案した，ロック命令の busy wait 中に他プロセスの状態を監視し，監視結果に応じて自プロセスの状態を変化させる適応型ロック命令[6]の技術を適用する．適応的再演型ロック命令は，先行ノードからの実行終了信号を待機する間に，待機の原因をもたらしているプロセスの状態を監視し，その状態に応じて自らの状態を変化させる．

(3)SP 判定機能 3.2 節で述べた，デマンド数，DP フラグ，DP プロセス数の更新操作を行い，SP へ到達したか判定してデバッガへ報告する．

(4)プロセス生成機能 プロセスが fork ノードを実行した際には，子プロセスを生成するとともに，実行監視時と同じプロセス識別子を与える．生成された子プロセスは，3.1 節の要求伝播，発火制御，結果伝播に従って再演を開始する．

適応的再演型ロック命令を用いることにより，プロセスは自律的な動作が可能となり，プログラムの並列性が損なわれない．また，再演に関する実行制御機能がロック命令に閉じ込められているので，並列プロセスプログラミングモデル[11]に従って，C, Pascal, FORTRAN 等を拡張した様々な並列プログラム言語に対しても，各々の言語の再演機能のないデバッガと組み合わせて用いることが可能となる．

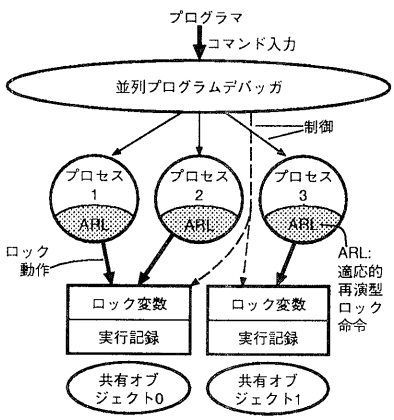


図5 適応的再演型ロック命令を用いたデバッグ時の再演制御
Fig. 5 Parallel program debugger with ARL.

4. メモリ共有型並列計算機上での並列デバッガの実現

4.1 システム構成

適応的再演型ロック命令を用いた並列デバッガシステムのプロトタイプを，8 個の CPU を持つメモリ共有型並列計算機 sequent symmetry 上で作成した．プロ

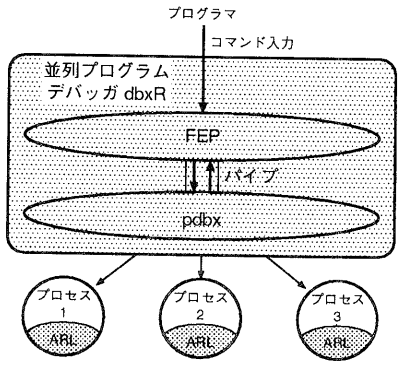


図6 symmetry 上での dbxR の実現
Fig. 6 Implementation of dbxR on symmetry.

表1 dbxR の主要コマンド
Table 1 Commands of dbxR.

停止位置設定	**replay obj OID #ACCESS** 共有オブジェクト **OID** に対する **#ACCESS** 番目のアクセスに DP を設定する
	stop at #LINE ソースコードの **#LINE** 行目に BP を設定する
実行制御	**run ARG1 ARG2 …** **ARG1** …を引数としてプログラムを実行する
	stop all 実行中のプログラムを停止させる
	cont all 停止させたプログラムの実行を再開する
表示	**show obj OID** 共有オブジェクト **OID** の実行記録を表示する
	show proc PID プロセス **PID** のステータスを表示する
	list #LINE ソースコードを **#LINE** 行目から表示する
	print ARG 変数 **ARG** の値を表示する

トタイプは，**図6**のように適応的再演型ロック命令（ARL）と並列プログラムデバッガ（dbxRと呼ぶ）で構成する．実現した適応的再演型ロック命令は，symmetryの標準的なロック命令[9]であるs_lock（），s_unlock（）と同様なインタフェースを持ち，ライブラリの形で提供した．この命令は実行再現時にIPアークを求められるように，実行監視時に各共有データアクセスに対して，アクセス種別（read/writeを表す1ビット）とプロセス識別子を記録し，プロセス生成に対して子プロセスの識別子を記録する．dbxRはsymmetryの標準的なパラレルデバッガpdbx[8]と，これを司るプロセス（FEP）からなる．図6に示したように，FEPはプログラマからのコマンドを受け，pdbxを制御することにより被デバッグプロセスの制御を実現している．dbxRの主要コマンドを**表1**にまとめる．

4.2 データ構造

本並列デバッガは，**図7**に示す実行記録，プロセステーブル，DPプロセス数からなる3種類のデータ構造を用いる．実行記録は共有オブジェクトごとに，プロセステーブルはプロセスごとに1つずつ割り当てる．DPプロセス数はシステム全体で1つ保持する．

実行記録とプロセステーブルの取り得る値の意味は，**表2**，**表3**の通りである．また，DpCountが取り得る値は未実行の実行要求を持つDPプロセスの数である．

実行記録のlockは，並列プログラムの実行監視時に，共有オブジェクトに対する排他制御を実現するためのロック変数である．

実行記録の(Pid, Op, Demand)の3つ組は，図3の1つのノードに対応している．そのうち，(Pid, Op)の対は共有オブジェクトへの1回のアクセスの履歴であり，その共有オブジェクトに対して行われたアクセス順に記録される．このため，実行記録において，隣あった2組の（Pid, Op）からIPアークが抽出できる．また，Demandは対応するアクセス（Pid, Op）に対する，再演時の実行要求の発生，完了状況を表しており，図3のノードの網掛けは，DemandにDONEが設定されていることに対応する．再演開始時には，すべてのDemandはNONEに初期化される．3.1節で述べた要求伝播機能と結果伝播機能は，それぞれ対応するアクセス（Pid, Op）のDemandにDO, DONEを設定することで実現される．また，発火制御機能は実行記録より求まるIPアークから，先行ノードに対応するアクセス（Pid, Op）を調べ，そのDemandがDONEならば，実行終了信号が送られたと判定できるので，ノードを発火させれば良い．

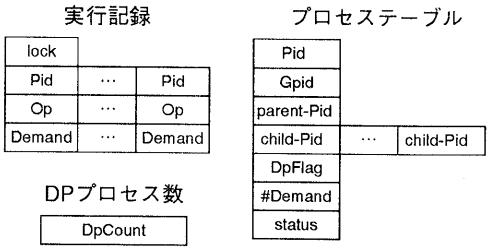


図7 適応的再演型ロック命令のデータ構造
Fig. 7 Data structures of ARL.

表2 実行記録の各フィールドの意味
Table 2 Contents of the execution record's fields.

lock	ロック変数
	=1（共有オブジェクトは施錠されている）
	=0（共有オブジェクトは解錠されている）
Pid	本デバッガが管理するプロセス識別子
	（ローカル識別子と呼ぶ）
Op	アクセス種別
	=r（読み出し操作の時）
	=w（書き込み操作の時）
Demand	実行要求の発生/完了状態
	=NONE（実行要求がない時）
	=DO（未実行の実行要求がある時）
	=DONE（実行済みの時）

表3 プロセステーブルの各フィールドの意味
Table 3 Contents of the process table's fields.

Pid	本デバッガが管理するプロセス識別子
	（ローカル識別子と呼ぶ）
Gpid	OSが管理するプロセス識別子
	（グローバル識別子と呼ぶ）
parent-Pid	親プロセスを示すローカル識別子
child-Pid	子プロセスを示すローカル識別子
DpFlag	DPフラグ
	=1（DPプロセス）
	=0（非DPプロセス）
#Demand	実行要求を受け，未実行なノードの総数
status	プロセスの再演状態
	=DemadWait（実行要求信号待ち状態）
	=OrderWait（実行終了信号待ち状態）
	=Replay（再演実行状態）
	=ARRIVE_DP（DPに到達）
	=ARRIVE_2 DP（2次DPに到達）
	=ARRIVE_3 DP（3次DPに到達）
	=ARRIVE_BP（BPに到達）
	=ARRIVE_2 BP（2次BPに到達）

プロセステーブルのPidは実行記録のPidと同一であり，本デバッガが独自に付与する．Gpidは，計算機のOSが付与したプロセス識別子であり，Pidが与えられた時に，システムコールによりプロセスを制御するために用いる．また，プロセステーブルの3つ組

(Pid, parent-Pid, child-Pid) から，図3の fork と created を結ぶ IP アークが抽出できる．実行要求信号を送るプロセスが未生成の場合には，そのプロセスを生成する fork 命令を持つ親プロセスを求めるために用いる．DpFlag, # Demand はそれぞれ，3.2節で述べた DP フラグならびにデマンド数を保持する．また，status の値から 3.2 節で述べた BP 到達フラグの値を求めることができる．

DpCount は，3.2 節で述べた DP プロセス数である．

4.3 適応的再演型ロック命令の動作

適応的再演型ロック命令は，共有オブジェクトへのアクセスを行う前後に入口動作と出口動作を行う．また実行監視モードと再演実行モードでは動作が異なる．それぞれの動作について以下に述べる．

(1)実行監視モードでの入口動作 指定された共有オブジェクトに対応した，実行記録の変数 lock に test& set 命令などの通常のロック命令を使用する．成功するまでロック動作を繰り返し行い，成功後，実行記録の最後に (Pid, Op) を書き加える．

(2)実行監視モードでの出口動作 指定された共有オブジェクトに対応する実行記録の変数 lock に 0 を書き込み，解鍵する．

(3)再演実行モードでの入口動作

1. Pid が自プロセスを示しているプロセステーブル（自プロセステーブルと呼ぶ）を読み出し，# Demand を調べる．0 ならば 2 へ．1 以上ならば 3 へ行く．
2. 自プロセステーブルの status を DemandWait に変更し，# Demand が 1 以上になるまで待機．
3. 指定された共有オブジェクトの実行記録から，自プロセスの Pid を持ち，かつ Demand が DO である最初の (Pid, Op, Demand) の組（自ノードと呼ぶ）を読み出す．
4. 自プロセステーブルの status を OrderWait に変更する．自ノードの一つ前の (Pid, Op, Demand) の組（先行ノードと呼ぶ）を読み出し，Demand が DONE ならば 5 へ，DO ならば 7 へ，NONE ならば 6 へ行く．
5. 自プロセステーブルの status を Replay に変更し，入口動作終了．
6. 先行ノードの Demand を DO に変更し，先行ノードの Pid が示すプロセステーブル（先行プロセステーブルと呼ぶ）の # Demand を 1 増やす．
7. 先行プロセステーブルの status を監視する．ARRIVE_BP か ARRIVE_2 BP ならば 8 へ行く．それ以外は 4 へ行く．
8. 自プロセステーブルの status を ARRIVE_2 BP に変更する．自プロセスの DpFlag が 1 ならば 9 へ，0 ならば，そのまま待機．
9. DP プロセス数 DpCount を 1 減じ，0 になったら dbxR にシグナルを打ち，被デバッグプログラムの停止を要請する．1 以上ならば待機．

(4)再演実行モードでの出口動作

1. 入口動作の 3 で読み出した自ノードの Demand を DONE に変更し，自プロセステーブルの # Demand を 1 減じる．# Demand が 0 になれば 2 へ，1 以上ならば出口動作終了．
2. 自プロセステーブルの DpFlag を調べ 1 ならば 3 へ，0 ならば 4 へ行く．
3. DP プロセス数 DpCount を 1 減じ，0 になったら dbxR にシグナルを打ち，被デバッグプログラムの停止を要請する．1 以上ならば待機．
4. 自プロセステーブルの # Demand が 1 以上になるまで待機し，出口動作終了．

4.4 並列デバッガ dbxR の動作例

Beck らがパラレルプログラミング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]に dbxR を適用した場合の簡単な動作例を図8と図9に示す．図8は DP だけを使用した

```
 1 dbx version 5.10 of 1/7/90 3:12 (Sequent).
 2 Type 'help' for help.
 3 reading symbolic information ...
 4
 5 (dbxR) show obj all
 6 Access Histories (w: writer  r: reader  *: replayed  $: demanded)
 7 Obj-0   0: 1w  1: 2w  2: 3w  3: 1w  4: 2w  5: 3w  6: 1w
 8 [-2000]
 9 Obj-1   0: 1w  1: 2w  2: 3w  3: 3w  4: 1w  5: 1w  6: 2w  7: 0w
10 [670]
11 (dbxR) run 1000 3 5000
12 (dbxR) replay obj 0 5
13 request:demand to object:0  index:5
14 Access Histories (w: writer  r: reader  *: replayed  $: demanded)
15 Obj-0   0: 1w$  1: 2w$  2: 3w$  3: 1w$  4: 2w$  5: 3w$  6: 1w
16 [5000]
17 Obj-1   0: 1w  1: 2w  2: 3w  3: 3w  4: 1w  5: 1w  6: 2w  7: 0w
18 [0]
19
20 ** PROC:1  WriterEntry [OBJ:0  HIST:0] **
21 ** PROC:1  Writer In [OBJ:0 HIST:0] **
22 ** 被デバッグプログラムの出力 **
23
24  %3  Stopped by stop in block at line 485 in file "process.c"
25  %2  Stopped by stop in block at line 485 in file "process.c"
26  %1  Stopped by stop in block at line 485 in file "process.c"
27  %0  Stopped by stop in block at line 485 in file "process.c"
28 (dbxR) show obj all
29 Access Histories (w: writer  r: reader  *: replayed  $: demanded)
30 Obj-0   0: 1w*  1: 2w*  2: 3w*  3: 1w*  4: 2w*  5: 3w*  6: 1w
31 [-1000]
32 Obj-1   0: 1w*  1: 2w*  2: 3w*  3: 3w  4: 1w  5: 1w  6: 2w  7: 0w
33 [366]
34 (dbxR) quit
35 Good-bye
```

図8 dbxR の実行例1
Fig. 8 Debugging example 1 with dbxR.

```
 1 dbx version 5.10 of 1/7/90 3:12 (Sequent).
 2 Type 'help' for help.
 3 reading symbolic information ...
 4
 5 (dbxR) show obj all
 6 Access Histories (w: writer r: reader *: replayed $: demanded)
 7 Obj-0   0: 1w  1: 2w  2: 3w  3: 1w  4: 2w  5: 3w  6: 1w
 8 [-2000]
 9 Obj-1   0: 1w  1: 2w  2: 3w  3: 3w  4: 1w  5: 1w  6: 2w  7: 0w
10 [670]
11 (dbxR) stop at 113
12 [1] stop at "prime.c":113
13 (dbxR) run 1000 3 5000
14 (dbxR) replay obj 0 5
15 request:demand to object:0 index:5
16 Access Histories (w: writer r: reader *: replayed $: demanded)
17 Obj-0   0: 1w$ 1: 2w$ 2: 3w$ 3: 1w$ 4: 2w$ 5: 3w$ 6: 1w
18 [5000]
19 Obj-1   0: 1w  1: 2w  2: 3w  3: 3w  4: 1w  5: 1w  6: 2w  7: 0w
20 [0]
21
22 ** PROC:1 WriterEntry [OBJ:0 HIST:0] **
23 ** PROC:1 Writer In [OBJ:0 HIST:0] **
24 ** 被デバッグプログラムの出力 **
25
26  %1 Stopped at breakpoint 1 in work at line 113 in file "prime.c"
27  %3 Stopped by stop in WriterEntry at line 161 in file "crew.c"
28  %2 Stopped by stop in WriterEntry at line 161 in file "crew.c"
29  %0 Stopped by stop in block at line 485 in file "process.c"
30 (dbxR) show obj all
31 Access Histories (w: writer r: reader *: replayed $: demanded)
32 Obj-0   0: 1w* 1: 2w* 2: 3w* 3: 1w$ 4: 2w$ 5: 3w$ 6: 1w
33 [2000]
34 Obj-1   0: 1w* 1: 2w* 2: 3w* 3: 3w  4: 1w  5: 1w  6: 2w  7: 0w
35 [119]
36 (dbxR) quit
37 Good-bye
```

図9 dbxRの実行例2
Fig. 9 Debugging example 2 with dbxR.

場合の動作例を示し，図9はDPとBPを併用した場合の動作例を示している．

図8の5行目で文字列“(dbxR)”は，並列デバッガdbxRのプロンプトであり，“show obj all”は全共有オブジェクトの実行記録の表示を指示するプログラマからのコマンドである．6〜10行目はコマンドの実行結果である．7行目が1個の共有オブジェクト(Obj-0)の実行記録であり，図7の実行記録のPidとOpの値を示している．

11行目でプログラマはrunコマンドを用いて，被デバッグプログラムの実行開始を命令しているが，DPが設定されていないので，再演は開始されない．runコマンドの引数は，実行監視モードで与えた引数と同一である．12行目でreplayコマンドを用いて，共有オブジェクトObj-0の5番目のアクセスにDPの設定を行うと，dbxRはDPと2次DPのノードに実行要求信号を送り，“$”マークを付加して実行要求信号が送られたノードを表示する(15行目)．実行要求信号を与えられた被デバッグプログラムは再演を開始し，DPの必須命令を再演後，停止する．その後，24〜27行目のメッセージが，4個のプロセスが停止したことを表示している．29〜33行目で表示している再演後の実行記録では，“*”を付加して再演されたノードを表している．

図9では，図8と同様のノードにDPを設定するとともに，ソースコードの113行目にBPを設定している(11行目)．このため，再演中にプロセス1がBPに到達して停止し(26行目)，これを検出したプロセス2とプロセス3は2次BPに到達となり，停止する(27,28行目)．32行目の実行記録が示すように，プロセス1はObj-0の3番目の書き込みアクセスを行う前にBPに到達して停止しており，これを待ち合わせるプロセス2およびプロセス3のObj-0に対する読み出しも，再演されていない．

5. 考　　察

5.1 従来の再演法との比較

Caverらの P-sequence 法[1]では，共有データへのアクセスをすべて逐次化して，アクセスしたプロセスの識別子の系列を一つの履歴テープに記録する．このため，実行監視時および再演実行時のプログラム実行が逐次化され，ボトルネックが生じる．また，LeBlancらの Instant Replay 法[2]では，共有データへのアクセスを，その共有データごとのバージョン番号で表し，各プロセスが持つ履歴テープにバージョン番号を記録することで，ボトルネックを回避している．しかし，直接，順序関係のないアクセス命令に対しては，再演時に実行制御を行っていないため，プロセスがBPで停止しても他のプロセスは可能な限り再演実行を進めてしまう[4]．

これに対して，本デバッガでは，実行記録の作成機能を排他制御命令であるロック命令に閉じ込めて実現しているので，被デバッグプロセスは自律的に動作可能である．さらに，共有データごとに実行履歴を作成しており，それらへの書き込みを並列に行えるので，実行監視時にプログラムが持つ並列性を損なわない．また，実行記録上の停止位置DPに，ただ一度だけプロセスが到達することを保証し，DPに到達するのに必要最小限の命令以外は再演実行されないことを保証している．

真鍋らは再演法を用いたメッセージパッシング型並列デバッガを提案している[3]．この提案では，条件が成立した際にプロセスの実行を停止する条件付BPを複数のプロセスに設定し，すべての条件付BPが同時に成立するのに必要最小限の命令の実行だけを再現できることを示している．また，いずれかの条件付BPが成立した際に，各プロセスの実行を必要最小限の実行だ

けで停止させることはできないことを証明している．この再演法を用いた場合には，条件付BPは静的なソースコード上に設定するので，再演時にプロセスが条件付BPへ到達することは保証されず，到達しない場合，あるいは条件が成立しない場合には，P-sequence法やInstant Replay法と同様に，最後までプログラムの再演が進んでしまう．また，最小限の再演を行うためには，メッセージを受信して処理を進める度に，変数の値が変わる可能性があるので，その度に条件付BPの成立を検査しながら再演実行を進める必要がある．このため頻繁に実行を継続するのかを判定し，その都度，他プロセスの実行が必要となるまで，実行を遅らせるので，再演時にプログラムの持つ並列性を損なっている．

本デバッガでは，BPだけを用いて再演させた場合に比べ，実行記録上に設定するDPを用いるので，再演時に必ず到達する停止位置が設定できる．また，DPプロセスにおいて，DPと2次DPまでの命令は，実行すべき命令に含まれることが，再演開始時に判定できるため，DPプロセスは並列に再演実行可能であり，再演時においてプログラムの逐次化の影響を小さくすることが可能である．

5.2 実行監視の影響

実行監視によるプログラム実行への影響を考察するため，実行監視機構を組み込んだプログラム実行と，実行監視機構を組み込まない通常のプログラム実行を比較する．実行監視を行う実行では，通常の実行に比べ，共有データへのアクセス命令を実行するごとに，その履歴を実行記録に保存する操作が加えられている．本デバッガでは実行記録からIPアークを抽出できれば良いので，アクセスしたプロセスの識別子とアクセス種別(read/writeを表す1bit)だけを記録する．よって共有データへの1回のアクセスに対して，記録するデータ容量は1バイト程度に抑えられている．また，前節で述べたように，実行記録の作成機能をロック命令に閉じ込めたことにより，実行監視時にプログラムの再演性を損なわない．さらに，要求駆動型再演法の予備的評価では，本実行監視機構が応用プログラムの実行時間に与える影響が0.1〜0.5%程度の延びに抑えられていることが得られている[4),5)]．このように，記録するデータ容量，並列性および実行時間の観点から，定常的に実行監視機構を組み込んでプログラムを実行させても，十分許容できる範囲内にあると言える．また，上記予備的評価では，素数を求める簡単なプログラムを対象としたので，今後は，より実用的な応用プログラムを対象として，実行監視時のプログラム実行に与える影響を定量的に評価する必要がある．

5.3 従来の並列デバッガとの比較

本デバッガは，図6に示したように，再演に関する実行制御機能を閉じ込めた適応的再演型ロック命令ARLとプログラマからのコマンドを受けるFEPとして実現し，標準的なパラレルデバッガpdbxと接続して構成している．このため再演機構は，変数の値の表示やブレークポイントの条件指定など，通常のデバッガの機能に影響を与えない．言い替えると，通常のデバッガからの独立性が高いので，通常のデバッガが提供している機能はそのまま利用可能であり，新しいデバッグ機能を考案した時，これらを組み込んで機能を拡張するのは通常のデバッガと同程度の工数で行えると考えられる．

次に，本デバッガが再演実行時のプログラム実行に与える影響を考察する．まず，再演実行を行わない通常のデバッガを用いた場合と，本デバッガを用いた場合のプログラム実行を比較する．通常のデバッガの場合には，各プロセスは可能な限り実行を進めるので，プログラムが持つ並列性を十分引き出すことができる．しかし，再演性を持たないので，実行ごとに動作が変化してしまう可能性がある．これに対して本デバッガでは，実行監視時と同じ実行動作を保証するのに必要なプロセスの実行制御を行うため，通常のデバッガと比べて，プログラムが持つ並列性を損なう恐れがある．

再演時の並列性を向上させるため，実行監視時にPCアークとIPアークの両方を保存して，DPまでの必要最小限な再演実行を行うように，提案手法を変更した手法（変更手法と呼ぶことにする）が，考えられる．この変更手法では，再演開始時に，DPへの到達に必要最小限の命令がすべて求まり，並列に再演実行が可能となる．

本手法と変更手法は共に，DPへの到達に必要最小限な再演，および，停止したプロセスの実行位置が決定的であることを保証する．この時，本手法がDPと2次DPの位置を再演開始時に決定し，3次DPの位置を再演中に動的に決定するのに対して，変更手法は，DP, 2次DPおよび3次DPの位置を再演開始時に決定する相違点がある．よって，本手法と変更手法では，再演停止時の3次DPの位置が異なる場合がある．また，本手法は，変更手法に比べて再演時の並列性が損なわれる恐れがある．その代わりに，実行監視時にPCアークの保存によるプログラム実行への影響を削減している．本手法を用いた時に再演時のプログラムの並

列性に与える影響，および，変更手法を用いた時に実行監視時の実行時間が許容できる範囲に収まるかについて，定量的な評価が必要である．

6. おわりに

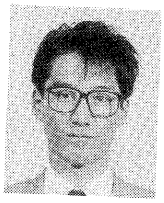
本稿では，再演法を用いたデバッガにおける最小限の再演実行の実現法，実行停止位置の設定法，再演停止法，デバッガ構成法の問題について考察し，これらの問題点を解決する並列プログラムデバッガモデルを提案した．今後は，本モデルに基づいて，メモリ共有型マルチプロセッサ計算機上に作成したデバッガを用いて，5章で述べた考察について定量的評価を進めるとともに，実際の並列プログラムのバグ検出において，再演法を用いたデバッガの有効性および使用上の問題点などを明確にするため，評価実験を行う予定である．

謝辞 日頃，御指導御討論いただく後藤滋樹部長をはじめ広域コンピューティング研究部の皆様に深く感謝します．本論文に有益な御指摘を賜わりました査読者の方々に感謝します．

参考文献

1) Carver, R.H. et al.: Reproducible Testing of Concurrent Programs Based on Shared Variable, *Proc. 6th Int. Conf. Dist. Comp. Sys.*, pp. 428-433 (1986).
2) LeBlanc, T.J. et al.: Debugging Parallel Programs with Instant Replay, *IEEE Trans. Comp.*, Vol. C-36, No. 4, pp. 471-482 (1987).
3) 真鍋, 今瀬: 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, 情報処理 SF-52-34 (1989).
4) 高橋直久: データ共有型並列プログラムの部分再演法について, 信学会 COMP 89-98 (1989).
5) 高橋直久: データ共有型並列プログラムの要求駆動型再演システムの実現と評価, *JSPP '90*, pp. 361-368 (1990).
6) 三栄, 高橋: 適応型ロックを用いた並行プロセスのコスケジューリング機構の実現法, 情報処理 OS-53-3 (1991).
7) 山田: 並列処理システムにおけるプログラムデバッギング, 情報処理, Vol. 34, No. 9, pp. 1170-1179 (1993).
8) SEQUENT COMPUTER SYSTEMS INC.: Pdbx User's Manual (1988).
9) SEQUENT COMPUTER SYSTEMS INC.: Guide to Parallel Programming, Prentice Hall (1989).
10) Davis, A.L. et al.: Data Flow Program Graphs, *IEEE Comp.*, Vol. 15, No. 2, pp. 226-241 (1982).
11) Beck, B. et al.: A Parallel Programming Process Model, *Proc. Winter 1987 USENIX Tech. Conf.*, pp. 84-102 (1989).

(平成6年9月14日受付)
(平成7年4月14日採録)



三栄 武（正会員）

昭和40年生．昭和62年名古屋工業大学工学部情報工学科卒業．平成元年同大学院工学研究科電気情報工学専攻博士前期課程修了．現在NTTソフトウェア研究所勤務．並列プロセススケジューリング，並列仮想計算機，並列プログラム実行環境，広域ネットワークコンピューティングなどの研究に従事．

高橋 直久（正会員）

昭和26年生．昭和49年電気通信大学応用電子卒業．昭和51年同大大学院修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，機能分散型並列計算機，データフロー型計算システム，関数型プログラミング，ソフトウェア・リエンジニアリング，広域ネットワーク・コンピューティングなどの研究に従事．現在，NTTソフトウェア研究所超並列プログラミング研究グループリーダ．工学博士（東京工業大学）．電子情報通信学会，日本ソフトウェア科学会，IEEE-CS, ACM各会員．