# 超並列計算機RWC-1上MPC++プログラム向けの再現実行方式

市吉 伸行　関田大吾　西岡利博　吉光 宏
RWCP 超並列 MRI 研究室

並列プログラムは同じ入力を与えても一般に動作が再現せず、デバッグ作業を困難にしている。この問題への対策の一つとして、並列プログラムの動作を記録し、それを元に動作を再現させる再現実行（リプレイ）機構がある。従来、共有メモリ型並列計算機や疎粒度メッセージ通信型並列計算機向けに幾つかの方式が実現されている。本論文では、メッセージ駆動型の細粒度超並列計算機 RWC-1 の並列言語 MPC++ 向けに設計している再現実行方式の実現方法について、非同期事象の記録方式を中心に述べる。

# A Replay Mechanism for MPC++ Programs on the Massively Parallel Computer RWC-1

Nobuyuki ICHIYOSHI, Daigo SEKITA,
Toshihiro NISHIOKA, and Hiroshi YOSHIMITSU

RWCP Massively Parallel Systems MRI Laboratory
2-3-6 Otemachi, Chiyoda-ku, Tokyo 100, JAPAN

Parallel programs are generally difficult to debug, because of the lack of repeatability in the behavior. A replay mechanism addresses this problem by recording the program behavior during an execution in a log and reproducing it according to the log. Replay mechanisms have been realized for shared-memory parallel computers and coarse-grain message-passing parallel computers. In the current paper, we present a replay scheme for MPC++ (Massively Parallel C++) programs on the fine-grain message-driven parallel computer RWC-1, focusing on the recording scheme of asynchronous events.

## 1 再現実行とその方式

逐次プログラムのバグ追求においては、通常、バグが顕在化する箇所の手前の1つないし複数の適当な箇所にブレークポイントやトレースポイントを設定して、途中状態を検証することによってバグ発生場所を狭めるという「繰り返しデバッグ (cyclic debugging)」手法が用いられる。

ところが、並列プログラムでは、要素プロセッサの非同期性に起因する動作の非決定性があるために、プログラム動作に再現性がない。特に、バグ原因追求のために、デバッグライトを挿入したりやブレークポイントを設定すると、プログラム動作に影響が及ぶため、再現性をより大きく損ない、バグ追求作業を困難にしている [10]。

再現実行（リプレイ）とは、プログラム動作を記録し、それを元にしてプログラム動作を再現させることである（図1)。プログラム動作のいろいろな側面を観測しながら、同じ実行を何回も繰り返せるところに再現実行の意味がある。

並列に走る各プロセスは逐次的・決定的だから、非同期な通信を記録して再現すればよい、というのが再現実行の基本的な原理である。しかも、全体の実行を再現すれば、通信における書き込み／送り出し側のプロセスにおいてもデータが再現するため、授受されるデータの内容が再現されることが保証されるので、通信のタイミングのみを記録すれば十分である。

LeBlanc と McCrummey は、複数のプロセスが非同期に共有オブジェクトに対して読み書きを行なうというかなり一般的なモデルを仮定して、再現実行を行なう Instant Replay 方式を示した [3]。Instant Replay 方式では、共有オブジェクトへのアクセスに逐次性が保証されているものとし、共有オブジェクト毎にバージョン番号を管理する（書き込み毎ないしアクセス毎にバージョンを上げる)。記録時に、各プロセスは共有オブジェクトにアクセスする度に、そのプロセス固有のプロセス履歴テープ (PHT) にバージョン番号を記録する[1]。再現時には、各プロセスは共有オブジェクトにアクセスする際、記録と合うバージョン番号になるように待ち合わせを行なう。共有メモリ並列計算機上の具体的実現例を記しているが、非同期メッセージ通信型の計算モデルについては、共有オブジェクトによってモデル化可能と述べているものの、具体的方法を示していない。

高橋 [9] は、Instant Replay 方式に基づきつつ、アクセス順の情報をプロセス毎でなく共有オブジェクト毎のオブジェクト履歴テープ (OHT) に記録する方式を提案している。(また、再現時のデバッグにおいて、ブレークポイントの他に、デマンドポイントという止め方を提案している。)

Leu らは、send/receive などを含む通信用ライブラリによってプロセッサ間通信を行なうプログラムについて、再現を行なう方式を示した [4]。この方式では、各プロセッサ上で通信プリミティブ呼び出しのシーケンス番号を管理し、通信事象発生時にカレントなシーケンス番号と事象の種類（送信元／送信先プロセッサ番号、受信／送信）[2]を記録し、再現時には記録時の順序を守るように、通信ライブラリが然るべく待ち合わせを行なう。

今回、再現実行の対象とする MPC++ プログラムの計算モデルでは、分散メモリ並列計算機の各ノードにおいて一般に複数のスレッドがあり、非同期に他のノードからスレッド起動やリモート変数アクセスがなされる。メッセージ通信型並列計算機を前提とする点では Leu らの場合と共通するが、明示的な通信プリミティブがない点が異なる。通信プリミティブがある場合は、通信がローカルなプロセスに影響を及ぼす時点を静的に限定できるが、非同期なスレッド起動やメモリアクセスがあると、影響の及ぶ範囲が大幅に広がるため、非同期タイミングの記録方法に工夫が必要となる。

以下の節では、MPC++ の言語仕様、実装のターゲットである並列計算機 RWC-1、RWC-1 上の MPC++ 実現方式を説明した後、その再現実行方式について述べる。

## 2 並列言語 MPC++

MPC++ は、C++ 言語を非共有分散メモリ MIMD 型並列計算機向けに拡張したプログラミング言語である [6]。MPC++ の計算モデルでは、並列計算機を構成するプロセッシングノード（以下、ノード）は、単一プロセッサと局所メモリを持ち、

[1] PHT の集合は、プロセスの親子関係を反映した木構造を持つ。

[2] 送信事象があるのは、対象となった iPSC/2 計算機において、send プリミティブがバッファを指定した送信要求であり、送信完了かどうかをチェックするプリミティブが存在するためである。

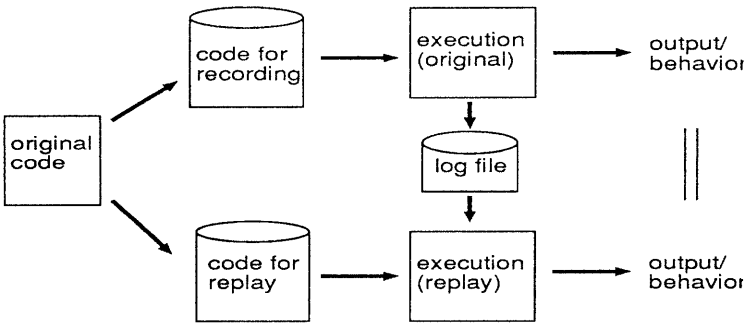


図 1: 再現実行の流れ

任意のノードと互いに通信し合うことができるものとする。このような並列計算機の各ノード上に一般に複数のスレッドがあり、ローカルなプロセッサ資源とメモリ資源を共有する（マルチスレッディング）。

ノードをマルチプロセッサに一般化することも将来的にはあり得るが、本論文で記す再現実行方式では、単一プロセッサであることを本質的な前提とする。

以下、主に再現実行方式を理解するのに必要な部分に要約して言語仕様を述べる。

## 2.1 並列変数

MPC++ プログラムでは、関数ローカルでない変数はデフォルトではノードに局所的だが、特別な宣言により、他ノードからもアクセス可能な並列変数[3]となる。局所変数は同一名のものが各ノードに一つあり、それぞれはそのノード上のスレッドによってのみアクセス可能である。この制約のため、局所変数へのポインタを他ノード宛のスレッド生成の引数とすることは許されない。並列変数は同一名のものが全空間に一つあり、任意のノード上のスレッドによってアクセス可能である。

## 2.2 スレッド生成

MPC++ のスレッドは、スレッド生成式によって生成される。スレッド生成式のシンタックスは、C++ 言語における通常の関数呼び出しの後に “@ <ノード指定>” ないし “@ <合流指定> <ノード指定>” を付加したものである。ここで、<ノード指定> は “[]” ないし “[<ノード番号>]” であり、<合流指定> は “()” ないし “(<トークン>)” である。トークンは合流先を示す特殊なデータ型である。(“()” は「合流なし」を意味する)。スレッド生成式の型は、合流指定がない場合は呼び出す関数の型と同じであり、合流指定がある場合は void 型である。

スレッド生成式の評価により指定されたノード宛の関数呼び出しが行なわれる（なお、“[]” は「自ノード」を意味する)。式の型から示唆されるように、合流指定がない場合は、呼び出し直後で関数値の待ち合わせが行なわれる（通常のリモート手続き呼び出し (RPC) に対応)。合流指定がある場合は、呼び出し側のスレッドは関数値返却を待たずに続行される（スレッドフォーク)。例えば、

```
1 void hoo(int,int);
2 int goo(int);
3 sub1() {
4   int i;
5   hoo(10,20) @ ()[];
6   i = goo(10) @ [3]; }
```

において、5 行目は同一ノード上で関数 hoo を計算するスレッドのフォークであり、6 行目はノード 3 への関数 goo の呼び出しと結果の i への代入である。

## 2.3 スレッド間同期

スレッド間のバイナリ同期のために、トークンとメッセージエントリという機構が用意されている。トークンはデータ／制御の合流先を示す特殊なデータ型であり、トークンに対してリプライ文を実行することにより、合流先へのデータの送出が行なわれる。送受信するデータの数（0 個以上）と型は静的

[3]文献 [6] では「トポロジ上の変数」と呼んでいる。現在、名称・仕様とも変更中のため、仮にこのように呼ぶ。

に宣言しておく。メッセージエントリはトークンの合流先であり、必要ならばスレッドを中断してトークンの到着を待つ。
下に例を示す。

```
 1 void p1(token(int,char) t1) {
 2   ...
 3   t1 <- [10,'h'];      /* リプライ文 */
 4   ... }
 5
 6 main() {
 7   int i,j;
 8   char k;
 9   entry(int,char) L1;
10   ...
11   p1(L1)@()[5];   /* スレッド生成文 */
12   ...
13  L1(j,k):      /* メッセージエントリ */
14   i += j;
15   ... }
```

1行目の関数 p1 の引数宣言では t1 が整数型と文字型を返す先を示すトークンであることが宣言されている。3 行目で t1 に対してリプライ文で 10 と 'h' を送っている。関数 main では、9 行目で自動変数として、整数型と文字型を受けとるメッセージエントリ L1 が宣言されている。11 行目には p1 を実行するスレッドをノード 5 にフォークするスレッド生成文があり、メッセージエントリ L1 に値を返すトークンを渡している（メッセージエントリを右値として参照すると、そのメッセージエントリを合流先とするトークンが得られる）。13 行目では、メッセージエントリ L1 が p1 からの値を変数 j、k に受け取り、14 行目以下の計算で用いている[4]。

### 2.4 スケジューリング

スレッドは内的な原因で中断しない限り、他のスレッドとインターリーブされることなく実行される。中断の原因としては、リモート読み出し（並列変数の読み出しがリモートアクセスになる場合）とメッセージエントリ文でのトークン待ち合わせの 2 つがある。ただし、外部からリモートアクセスは、スレッド実行と非同期な任意の時点で起き得る（したがって、並列変数は同じノード上のスレッドにとって volatile な変数である）。

[4] メッセージエントリはラベルと類似のシンタックスをしているが、一つの関数の中で同一のメッセージエントリが複数回出現しても構わないことになっている。したがって、メッセージエントリは、トークンを引数とする実行文と理解すべきであろう。

各ノード上では、実行中のスレッドが中断ないし終了すると次のスレッドがスケジュールされる（FIFO スケジューリング）。

### 2.5 データ並列処理

データ並列処理のための with 文も用意されているが、詳細な仕様は未定である。そこで以下では、スレッド生成のマルチキャストとそれにそれら全てのスレッド終了の待ち合わせとして with 文が実装されることを仮定し、言語プリミティブとしての with 文は存在しないものとするすなわち、データ並列処理のための特別な対策を再現実行方式では用意しない。

## 3 超並列マシン RWC-1

RWC プロジェクトで開発中の RWC-1 は、MIMD 型の分散メモリ細粒度超並列マシンである [8]。RWC-1 の採用する RICA アーキテクチャは、細粒度プログラムの高効率実行の支援を目指しており、低遅延・大容量ネットワーク、プロセッシングノードにおけるマルチスレッディングのサポート、計算処理へのオーバヘッドの小さいパケット送受信機構などが特徴である。RWC-1 は、64 ビット・プロセッサからなるノードが最大千個程度結合されたシステムとなる予定である。

パケットの生成は自ノード宛、他ノード宛を問わず、同一の命令（mkpkt 命令）によって行なわれる。パケットの内容は、命令セグメントアドレス、データセグメントアドレス、および引数（0～8個）である。これらは宛先ノードで、プログラムカウンタ、データベースアドレスレジスタ、および汎用レジスタにそれぞれ展開され、そのコンテキストで実行が開始される。パケットはコンティニュエーションの、ハードウェアによる直接的表現と言える。なお、2 つのノード間の通信は FIFO である。

パケットおよびスレッドには 4 レベルのハードウェア優先度があり（システムレベル 2 つ: System High, System Low、ユーザレベル 2 つ： User High, User Low）、それらに対応して、各ノードに優先度毎の入力パケットキュー（FIFO キュー）がある。パケットは宛先ノードに到着すると、対応する優先度の入力キューの末尾にエンキューされる。実行中のスレッドが終了すると、最高優先度キューの先頭のパケットの内容がレジスタに展開され、実行が開始される。

ただし、実行中のスレッドより高い優先度のパケットが到着すると、実行中スレッドをプリエンプトして実行が開始される。この際、高い優先度のパケットは実行中スレッドと異なる汎用レジスタセットに展開されるため、コンテキストの退避が不要である[5]。スレッドは終了命令（break 命令）により終わる。

同期に関しては、スレッド間の高速な同期をサポートするバイナリ同期命令（bsync 命令）が用意されている。バイナリ同期とは2つのデータが揃う、ないし、2つの制御が一定箇所に達した時点で、後続処理を行なうということである。そのために、待ち合わせ用領域を確保し、待ち合わせ地点に到着したスレッドは、待ち合わせフラグをチェックし、相手が到着していなければ、必要なコンテキストを待ち合わせ用領域にセーブし、到着したというフラグを立てて、終了する。後から到着したスレッドは、早く到着した相手方のデータをリストアして、後続処理を行なう。これらの処理は不可分に行なわなければならないが、バイナリ同期命令は待ち合わせフラグのロックと条件分岐までを高速に行なう。

RWC-1 はまた、実時間処理をサポートするため、通信を含めた実行中のプログラムのコンテキストを全系で同時に高速にセーブするドレイン機能を備える（ネットワーク上のパケットは最寄りのノードに退避される）。

## 4 RWC-1 上の MPC++ 実現方式

以下、MPC++ 計算モデルにおけるスレッドと RWC-1 ハードウェアの定義するスレッドとを区別する必要のある時は、前者を MPC++ スレッド、後者をハードウェア・スレッドと呼ぶことにする。

RWC-1 上の MPC++ 処理系 [7] では、MPC++ スレッドは User Low ハードウェア・スレッドで実行される。MPC++ スレッド中断の際は、一連の実行をしてきたハードウェア・スレッドが終了し、再開時には、新たなハードウェア・スレッドが起動される（実行コンテキストは然るべく継承される）。自ノード上にない並列変数へのアクセスは、宛先ノードで直ちにに処理されるようにリモート読み書き用の User High パケット送出によって実現される。リモート読み出しの場合は、読み出し後のコンティニュエーションを付けた読み出し用 User High パケットを送出して、ハードウェア・スレッドが終了し（MPC++ スレッドが中断）、読み出し値を持った返答用 User Low パケットがそのコンティニュエーションから処理を開始するスレッドを起動する（MPC++ スレッドが再開）。

[5] ただし、システムレベルのレジスタセットは1つしかないため、System Low スレッドは System High スレッドにプリエンプトされない。

## 5 RWC-1 向け再現実行方式

MPC++ プログラムの再現の自然なやり方は、MPC++ スレッドを逐次的・決定的プロセスと見なして、スレッド間の同期・通信を記録／再現する方式であろう。しかしながら、局所変数は同一ノード上のスレッドの全てがアクセスできる「共有変数」であるため、そのアクセス順の記録／再現が必要だが、オーバヘッドが大きすぎる。実際のプログラムでは、「本当に」共有されている変数とそうでない変数の区別があり、前者へのアクセスには然るべきロック／アンロックを用いるなどすると思われるが、静的解析でそのような情報を抽出することは困難であろう。

そこで、プロセッシングノードを逐次的・決定的プロセスと見なし—単一プロセッサからなるため実際にそうである—、それらの間の相互作用を記録する方針とした。ノードにとっての非同期事象はパケット到着だが、ソフトウェアから観測されるのは、パケット到着そのものでなく、パケットがスケジュールされることによるハードウェア・スレッド起動である。そこでハードウェア・スレッド起動を記録すればよいことになる。また、User Low スレッドと User High スレッドは生起タイミングの粒度が異なるため、それぞれに適当な記録方法を取ることとした。以下、断らない限り、「スレッド」は「ハードウェア・スレッド」を指す。

### 5.1 User Low スレッド起動タイミング記録

User Low スレッド同士はインターリーブされずに実行されるので、スレッドの実行順序（＝起動順序）を記録すればよい。スレッドを同定する情報としては、スレッド送信元ノードで十分である。したがって、各ノード上でスレッド番号 tc を管理し、スレッド番号と送信元ノード番号の対を記録すればよい。連続して記録するなら、順に送信元ノード番号のみを書いていっても良い。

記録量削減対策としては、内部起源のスレッドが多いだろうことを考慮すると、それらについては送信元ノード番号の代わりに、それらが連続して何個続いたかを記録することが考えられる（外部起源スレッドの間隔を記録するのと同じこと）。また、これを一般化した、同一ノードからの連続したスレッド到着を「送信元ノード × 個数」とする記録方法も、ノード間通信パターンにローカリティがあれば有効であろう。

## 5.2 User High スレッド起動タイミング記録

User High スレッドは User Low スレッドをプリエンプトするため、タイミングの記録は User Low スレッド同士の時のように単純ではない。このタイミングの記録方法は、今回の再現方式の設計における中心的課題であった。以下に述べるような方法を検討している。

### 5.2.1 命令カウンタ方式

プリエンプションは任意の命令の切れ目で起き得るため、タイミングを正確に再現するためには、命令レベルの粒度の記録が必要である。プログラムカウンタ (PC) は自然な候補だが、ループや関数呼び出しによって、プログラムコード中の同一命令は一般に複数回実行されるため、PC だけでは情報として不十分である。本来なら、プログラム開始時から非同期タイミングにいたる実行パス（各分岐の方向の情報など）を記録しなくてはならないが、非現実的なオーバヘッドと記録量になろう。幸い、データが再現されていれば、それに基づく条件分岐も再現されるため、プログラム開始時から非同期事象発生までの命令実行数を記録すれば十分である。

実際、再現実行やプロファイリング等を目的とするハードウェアによる命令カウンタ（instruction counter）が提案されている [2, 1]。しかしながら、命令カウンタは、通常のプログラム実行でオーバフローしないだけのビット幅を要し、また、再現時には指定されたカウンタ値で（ないし、カウントダウンして 0 になった時点で）例外を上げるための機構も必要であるなど、ハードウェア的なコストが大きい。一般的でないハードウェアカウンタを前提とした再現実行方式には一般性がないため、（筆者の知る限り）実現例がない。

そこで、ソフトウェアによる方式が提案されている。プログラムカウンタがタイミングの特定に不十分であるのは、同一の命令が複数回実行され得るためだったが、逆に言えば、同一命令の複数回の実行を区別できれば、タイミングが特定できる。そこで、ソフトウェア的にカウンタを導入して、同一命令が 2 回実行される間にインクリメントされるようなコードを生成する。これをソフトウェア命令カウンタ（software instruction counter）と呼ぶ [5]。

なお再現時に、ソフトウェアカウンタと PC の対で指定されるタイミングで例外を上げるためには、カウンタをインクリメントする度に指定されたカウンタ値と比較して[6]、等しくなった時点で、指定された PC 値に例外を設定する（PC 値による例外を上げるハードウェア機構ないし、例外命令の埋め込みによる）。

### 5.2.2 並列変数アクセス・カウンタ方式

これは、MPC++ 言語に依存した方式である。MPC++ プログラムでは、User High スレッドによるプリエンプションは並列変数アクセスの目的のみに用いられるため、その到着は並列変数アクセスの粒度で記録すればよい。

そのために、並列変数毎にタイミング記録用のカウンタを用意するのが自然だが、より簡明に、ノードに一つのアクセス・カウンタ ac を導入し、並列変数アクセスの度にインクリメントする（User Low / User High スレッドとも）。なお、Instant Replay 方式でも、共有オブジェクトの単位を大きくすることが可能だが、それによってプログラム自体の並列性が損なわれてしまう。それに対して、アクセスカウンタを各ノードに一つおくことは、並列性を損なっていない。

気をつけなければならないのは、正に ac のインクリメントのタイミングでプリエンプションが起き得るということである。

User Low スレッドから並列変数に代入を行なう下の例を考えてみよう。

```
pv = x;
ac ++;
```

ここで、pv は並列変数、x は局所変数とする。ac をインクリメントするコードが挿入されている（実際には並列変数に対するロード／ストア命令の直後）。ac

[6] 当然、カウントダウンして 0 と比較してもよい。また、オーバフロー例外をトラップするやり方もあり得る。

はUser Highスレッドからもアクセスされるvolatileな変数でなくてはならないが、acをインクリメントするタイミングでプリエンプションには2つの問題がある。

**インクリメント中のプリエンプション** acを読み出したタイミングでプリエンプションが起きると、リモートアクセスするコードの中でacをインクリメントした結果を、プリエンプション後にUser Lowスレッドが上書きしてしまう、という問題である。

これに対する対策(A)では、コンパイラが大域的にacを汎用レジスタに割り付けることで回避する(その代わり、User Highスレッドからのアクセスがやや面倒になる)。対策(B)では、acをUser Lowスレッドによるアクセス分iacとUser Highスレッドによるアクセス分eacとの2つに分割し、User Lowスレッドはiacのみをインクリメントし、User Highスレッドはeacのみをインクリメントするようにする。User Highスレッドはac = iac + eacを記録する。

**アクセスとのインクリメントの可分性** User Lowスレッドにおいて並列変数アクセスとacのインクリメントとが不可分でないため、acの値がアクセス回数を反映しないタイミングでプリエンプションが起きると正しいアクセス回数が記録できない。

素直な対策は、User Lowスレッドによる上記一連の処理を不可分にすることであろう。例えば、RWC-1ではUser LowスレッドがUser Highスレッドによるプリエンプションを禁止/許可する命令が用意されている。あるいは、acをロックしてもよい。しかし、前者は方式としてポータビリティがなく、また、後者の場合、acがロックされている時に起動されたUser Highスレッドの中断処理が必要となる。

そこで、違った対策として、User Highスレッド側でクリティカルなタイミングを判定するという方法も検討している。すなわち、User Highスレッドは、自分がプリエンプトしたUser Lowスレッドの実行している命令シーケンスを調べ、クリティカルなタイミングであればacの値としてac+1を採用するようにするのである。ac(ないしiac)がグローバルにレジスタに割り付けられている場合は、"inc ac"命令のユニーク性によって、iacがメモリ上の変数に割り付けられている場合は、そのアドレスのユニーク性によって、クリティカルな命令列の判定を行なうことができる。判定の具体的実現はマシン依存だが、方式自体はポータブルと思われる。この方式の利点はUser Lowスレッド側に対策が不要なことであり、欠点はUser Highスレッド側の判定の手間がやや大きいことである。

なお、アクセス・カウンタ方式の実行時における命令列解析オーバヘッドは次のようにして取り除くことができる。すなわち、User Highスレッドはacの他にUser LowスレッドのPCも記録し、実行後にPC値を用いて、対応するacを補正するのである(ただし、記録データ量が増えるこが問題である)。PC値を記録することで、後方分岐を間に含まない連続した並列変数アクセスは区別されるので、User Lowスレッド側のacの「解像度」を落とすことも可能ではある。具体的には、並列変数へのアクセスを含む最小のループ(ないし関数)当たり1回acをインクリメントすれば十分である。(ソフトウェア命令カウンタ方式とアクセス・カウンタ方式の融合と言える)。しかし、このようにacの「解像度」を落すと、ソフトウェア命令カウンタ方式と同様、再現時においてPC値による例外処理が必要となり、むしろ、ソフトウェア命令カウンタ方式に近くなろう。

### 5.2.3 比較

ソフトウェア命令カウンタ方式は、プラットフォームとしてRWC-1を固定した場合、プログラミング言語に依存しない方式という点で優れる。また、既にあるPCという資源を有効活用しているというのも好ましい。一方、再現時には、コード書き換えとソフトウェア例外が頻繁に生じることになり、オーバヘッドがかなり大きいと予想される。

並列変数アクセス・カウンタ方式は、並列変数アクセスの同定が必要なこと、並列変数アクセスとカウンタ・インクリメントが不可分でないことへの対策が必要なこと、などが欠点である。一方、通常のプログラムでは、ソフトウェア命令カウンタよりインクリメントの頻度が低いと考えられ、また、再現時のコード書き換えも不要であり、オーバヘッドがより小さいと期待される。また、PC値(圧縮しないと64ビット)と比べると、acないしその差分を表現するビット数(通常、1バイトないし2バイトで十分であろう。溢れは特別なエンコーディングで表わす)が少なくて済むのも有利な点である。

現時点では、RWC-1特有のプリエンプション禁

止命令に依存した方式を採用するか、または、方式としてのポータビリティを保ちつつ、オーバヘッドが妥当と思われる選択として、アクセス・カウンタ方式において、ac を iac と eac に分割し、iac を大域的に汎用レジスタに割り付け、User High スレッド側でクリティカルなタイミングを判定する方式を採用する方針である。

### 5.3 記録の出力方法、その他

記録用プログラム開始時にメモリ上にログ領域を確保し、非同期タイミングの記録をシリアルに書き出して行く。そして定期的に（例えば 10 msec 毎）、プロセススイッチ時に何らかのプロセススイッチ・ハンドラによって、「フリーズ」されているプログラムのログ領域をディスクにダンプする方式を考えている。これにより、ディスク出力処理が実行に及ぼす擾乱を低く抑えられることが期待される。再現時には、定期的にディスクからログ領域に記録をフィードする。

なお、User Low スレッド番号の記録中に User High スレッドにプリエンプトされても問題のないよう、両者のログ領域は分離すべきであろう。

## 6 おわりに

超並列マシン RWC-1 向けに検討中の再現実行メカニズムについて、非同期事象の記録方式を中心に述べた。非同期並列プログラムのデバッギングの大きな問題点の一つは、再現実行メカニズムによって解決される。今後、RWC-1 シミュレータ上で方式を検証し、その後、再現機構上のデバッガ、プロファイラの詳細な設計などを進めていく予定である。

記録時オーバヘッドの低減（典型的なプログラムで数割程度を目標）、記録量削減も大きな課題である。

## 参考文献


[1] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 194–206, May 1991.

[2] T. A. Cargill and B. N. Locanthi. Cheap hardware support for software debugging and profiling. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 82–83, October 1987.

[3] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, Vol. C-3, No. 4, pp. 471–482, April 1987.

[4] Eric Leu, André Schiper, and Abdelwahab Zramdini. Efficient execution replay technique for distributed memory architectures. In *2nd European Distributed Memory Computing Conference*. Springer-Verlag (LNCS 487), 1991.

[5] J. Mellor-Crummey and T. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 78–86, April 1989.

[6] 石川裕, 他. 超並列プログラミング言語 MPC++ の概要. 情処 93-PRG-13-11 (SWoPP'93), pp. 81–88, 1993.

[7] 石川裕, 他. 並列プログラミング言語 MPC++ の実現. 並列処理シンポジウム JSPP'94, pp. 105–112, 1994.

[8] 坂井修一, 他. 超並列計算機 RWC-1 の基本構想. 並列処理シンポジウム JSPP'93, pp. 87–94, 1993.

[9] 高橋直久. データ共有型並列プログラムの要求駆動再演システムの実現と評価. 並列処理シンポジウム JSPP'90, pp. 361–368, 1990.

[10] 山田剛. 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, 1993.