プログラミング 3－4
（1995. 9. 19）

MPC++ 向けイベントトレース解析ツールの実現

西岡 利博　村野 正泰　市吉 伸行　関田 大吾
RWCP 超並列 MRI 研究室

制御並列型の並列プログラムでは、実行の非決定性がデバッグを困難にする大きな要因である。また、デバッグの際に扱われるデータの量が大きくなりがちなのも問題である。これらの問題に対して、再現実行機構を提供することにより非決定性を排除し、適切な情報フィルタリング機能を提供することによりプログラマが直接扱うデータの量を減少させることが考えられる。本稿では、再現実行機構のサポートの上で、イベントトレースログを自動的に解析することでデバッグを進める手法について考察する。また、その有効性を確かめるために、C++ の並列拡張言語のひとつである MPC++ 言語を対象に作成した、イベントトレース解析ツールのプロトタイプシステムと、そこから得られた成果について報告する。

An implementation of event trace analyzing tools for MPC++

Toshihiro Nishioka　Masayasu Murano　Nobuyuki Ichiyoshi　Daigo Sekita
RWCP Massively Parallel Systems MRI Laboratory

In control parallel program, the nondeterminacy in the program execution makes debugging difficult. Furthermore, the programmer often have to examine huge amount of data. To tackle with these problems, there are works concerning the replay mechanism which can remove the nondeterminacy and the information filtering facility which can reduce the amount of data the programmer must concern. In this paper, we discuss how event trace analyzing tools on a replay mechanism can effectively solve these problems taking an example from a program written in MPC++ (which is an parallel extension of C++) and report the results from a prototype system of event trace analyzing tools.

1 はじめに

並列計算機の普及に伴い、その利用技術に関する研究もますます進展している。現在でも、実用的な並列プログラムのほとんどはデータ並列プログラムであるが、制御並列プログラムによる目覚しい研究成果も上がっており (例えば [3])、今後増加するものと考えられる。そのためには、制御並列のプログラミング環境の研究は重要である。

制御並列プログラムのデバッグにおける主な問題点として、1) プログラムの実行に非決定性があることと、2) PE の数が増えることで、障害の特定のために調べなければならないデータの量が増大しがちなことの二点がある。著者らは、これに対して以下のアプローチを取っている [8]。

- 再現実行機構により、サイクリックデバッグ環境を提供する。
- その上でイベントトレースログを取り、それを計算機で処理することによって、プログラマが直接扱うデータの量を減少させる

再現実行とは、プログラム実行中の動作を記録し (記録実行)、記録を参照しながらその動作を再現する (再現実行) という手法である [6]。再現実行の際に、トレース出力用のプローブなどを挿入してもユーザプログラムの挙動が変わらないことがデバッギング上重要である。

著者らはこのアプローチにしたがったデバッグ環境の有効性を確かめるために、今回、 MPC++ 言語向けのイベントトレース解析ツールのプロトタイプシステムを開発した。

ここで題材として選んだ MPC++ [4] は、C++ の並列拡張言語のひとつである。メッセージ送信や他 PE へのスレッドのフォークなどが、単純なシンタクスで実現できる上、ユーザ定義の言語仕様の拡張が可能なメタプログラミング機能を有しており、あらゆる種類の制御並列プログラムを記述するのに適している。以下に、特に制御並列部分について、 MPC++ の特徴を述べる。

- 関数の起動には、通常の関数呼び出しの他に、実行スレッドを分離して呼び出す、いわゆるフォークができる。本稿では、以下、フォークしない関数呼び出しの連鎖をスレッドと呼ぶ。
- 関数は (通常の呼び出しであれフォークであれ) 他の PE 上にも起動できる。

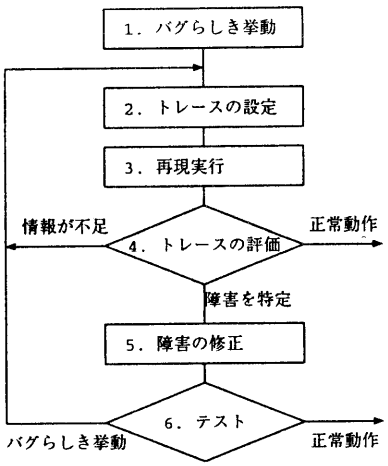

図 1: デバッグ作業フロー

- スレッド間の同期 / 通信手段として、メッセージの送受信が可能である。

本稿では、このようなデバッギングを実現する際の課題を述べた上で、そのために開発したプロトタイプシステムの実現と、そこから得られた知見について報告するものである。2節では、実現しようとしているイベントトレースベースのデバッギングについて述べる。3節では、プロトタイプシステムの実現と成果を報告する。4節で、まとめと、今後の研究方針などについて述べる。

2 イベントトレースベースのデバッギング

この節では、著者らが実現しようとしているイベントトレースベースのデバッギングの概要を述べ、そのための課題を明らかにする。

2.1 デバッグ作業フロー

プログラマは、図1のフローにしたがってデバッグを進める。以下は、フローの説明である。

1. プログラムは常に記録実行の状態で動作させておく[1]。このとき、プログラムの不審な挙動が認められ

[1]記録実行のオーバヘッドの大きさによっては、常に記録実行で動作させられないプログラムもあるであろう。その場合、バグらしき挙動を認めた時点で、同じ挙動が再現するまで記録実行を繰り返してから次のステップに進むこととなる。この選択は、性能上の要求、プロ

たら、デバッグを始める。

2. ソースコードなどから、不審な挙動を確認するために必要な情報を推測し、それを調べるためにどのようなイベントをトレースしたらよいか、設計する。
3. 再現実行によりイベントトレースログファイルを出力させる。
4. 解析ツールを用いて、得られたイベントトレースログを評価する。何らかの障害が特定できた場合は、6 に進む。それには情報が不足しているようであれば 2 に戻る。
5. 障害を修正する。
6. テストする。うまくいかなければ 2 に戻る。

以下ではこのデバッグ手法についてさらに考察する。

2.2 イベントの種類

上述のデバッギングのためには、どのようなイベントを記録できればよいだろうか。

解析したい問題の中には、競合検出やデッドロック解析など、各種のプログラムに共通のものもあるだろうが、大部分は、個々のアプリケーションに固有の問題であろう。例えば、以下のようなものが考えられる。

- ある関数の中で、何かの評価値を計算しているのだが、それが決して減少しないことを確認したい
- 関連のある二つのデータを管理しているのだが、両者の整合性が、ソースコード中の一部のクリティカルセクションの前後で正しく保たれていることを確認したい。

このためには、ユーザ定義イベントのサポートが必要である。一方、解析に必要な情報のすべてをユーザ定義イベントとして得るのでは、ユーザの負担が大きい。メッセージの送受信や関数の起動など、共通に使えるシステムイベントも提供することが望ましい。

2.3 イベントトレースログの解析

指定されたユーザ定義イベントとシステムイベントがイベントトレースログとして出力されたとして、それをどのように解析するのだろうか。

ユーザが、自分の意図しているプログラムの実行モデルを何らかの言語で記述すると、ログがそこで期待されている通りに出力されていることを検証してくれるような処理系が望ましい。この言語にはある程度の記述力が要求されるので、汎用プログラミング言語を採用している研究も知られている [10]。

汎用プログラミング言語を用いる例を示す。ここでは、ソースコード中のある位置で計算している評価値が減少しないことを検証する場合を考える。これを C で書けば、例えば以下のようになろう。

```
Record next (EventType et) {
  Record r;
  for (; (r=next_record ()) != 0;) {
    if (r->event_type == et) {
      return r;
    }
  }
  return 0;
}

void main () {
  Record r1, r2;
  for (r1=next ("eval_point");
       (r2=next ("eval_point")) != 0; r1=r2) {
    if (r1->value > r2->value) {
      alert ("%r, %r: ?value decrease?", r1, r2);
    }
  }
}
```

一方で、解析しようとする内容を宣言的に記述する文法を定義するアプローチも知られている ([5] など)。例えば、以下は一階述語論理風に記述した例である。

$$\begin{aligned}\forall x \forall y.\quad & event\_type(x) = \text{“eval\_point”} \\ & \wedge\ event\_type(y) = \text{“eval\_point”} \\ & \wedge\ time(x) < time(y) \\ \rightarrow\ & value(x) \leq value(y)\end{aligned}$$

ユーザにとって、どのアプローチがよいかは、十分には明らかではない。汎用のプログラミング言語では記述力があり過ぎてかえって複雑になりがちであり、その記述のデバッグをしなければならない可能性が高いが、一方で宣言的な記述は、巧みに設計しないと、効率の良い実装が自明でない上、記述力に難を生じるおそれがある。

以上から、デバッグのためにプログラムの実行モデルを記述する言語が持つべき特性を明らかにすることを課題とした。このために、MPC++ 言語を題材として、デバッグの際によく使われそうな解析ツールの開発を通じて、これを明らかにしていく方針を採用した。以

グラムの安定度などから総合的になされるべきである。

下では、そのような解析ツールとしてメモリアクセス競合検出器を選んで実装した結果を報告する。

3 プロトタイプシステム

今回、小規模の問題を題材として、プログラムの実行モデルを記述する言語に必要な性質を探るために、イベントトレース機構のプロトタイプシステムを開発した。これは、以下の部分からなる。

- システムイベントのトレースログを出力するランタイムライブラリを作成し、それを呼び出すコードを生成するように MPC++ コンパイラを改修した。
- イベントトレースログに対する基本的な操作を提供する基本解析ツールを開発した。
- 基本解析ツールを用いてメモリアクセスの競合を検出する、レースディテクタを開発した。

このレースディテクタは、基本解析ツールを次々と駆使するように実現されるので、この問題のために基本解析ツールにどのような性質が必要かが明らかになる。それらは、プログラムモデルを記述する際に必要な構文要素や、その性質を表している可能性が高い。

以下では、このプロトタイプシステムの実現と、そこから得られた知見について述べる。

3.1 ランタイムライブラリ

システム定義イベントを出力するランタイムライブラリを呼び出すコードを生成するように、コンパイラを改修する必要がある。今回は、並列計算機 RWC-1 のコードを出力する MPC++ コンパイラを元にした。改修内容は以下の通りである。

- そのイベントを記録したスレッドのスレッド ID が記録されるように、スレッドの実行を通じてスレッド ID を保持するようにした。
- レースディテクタの実現に必要なイベント (パケット送受信とメモリアクセス) をすべて記録するようにした。このコンパイラでは、関数の起動 / 復帰や MPC++ のメッセージ送信はすべて、RWC-1 のパケット送受信として実現されており、パケット送受信を記録することで、これらのイベントが記録される。

```
<LOGFILE>::=<RECORD>*
<RECORD>::='{' <ATTR>* '}'
<ATTR>    ::=<LABEL> '=' <VAL> ';'
<LABEL>   ::=< 識別子 > < 型 >
```

図 2: ログファイルの各レコードのフォーマット

```
    :
{etype$="SEND_PACKET"; pe&=0; rn%=2; ti
me%=0; thread&=1; to&=4; receiver_threa
d&=1; priority$="UL"; type$="CALL"; len
gth&=14; source$="fib.cc:12";}
{etype$="MEMORY_ACCESS"; pe&=0; rn%=3;
time%=0; thread&=1; acc_type$="WRITE";
address&=0x12000; size&=10; source$="fi
b.cc:13";}
    :
```

図 3: ログファイルの例

3.2 ログファイルのフォーマット

ランタイムライブラリは、ログファイルを、プロセッサごとにひとつずつ出力する。ひとつのログファイルは、1 イベントを 1 レコードとして生起順に格納している。ログファイルのフォーマットは図 2のように定めた。これは、他の多くのイベントトレースベースのシステム (ParaGraph など) と同様、属性 - 値対を基本としている。

パケット送信やメモリアクセスを含むイベントトレースログファイルの例を図 3に示す。

テキストフォーマットを用いているのは、本プロトタイプシステムのポータビリティとデバッグ効率を考慮した結果である。ただし、実行時にこのようなテキストベースのデータを整形出力するのはオーバヘッドが大きいので、処理系に応じて適切に最適化された形式で出力するものとし、それをこの形式に変換するツールを、基本解析ツールの一部として別途提供することとした。

3.3 基本解析ツール

基本解析ツールは、イベントトレースログファイルに対する基本的な操作を提供する。基本解析ツールを工夫することによってイベントトレースベースのデバッグに必要な要素機能を洗い出そうとしているので、使い方を手軽に試行錯誤できることが望ましい。そこで、基本

解析ツールは、シェルスクリプトから手軽に利用できるようなコマンド群として実現した。各コマンドはそれぞれ単純な機能を実現しており、3.2節の形式のデータをやりとりするフィルタコマンドとして、連結して利用できる。以下の各節では、各コマンドの間でやりとりされるデータのフォーマットと、いくつかのコマンドの機能について述べる。

3.3.1 コマンド群の機能

基本解析ツールとして作成したコマンドの中で、レースディテクタを実現する際に使われるものについて、以下に述べる。

- etcat
  ランタイムライブラリの出力するログファイルを、他の基本解析ツールが扱えるフォーマット (3.2節) に変換する。
- etgrep
  指定されたラベルと値の組を持つレコードだけを抜き出す。
- etuniq
  指定されたラベル (複数) の値がすべて等しいレコードについて、ふたつめ以降を取り除く。
- etcompare
  指定されたふたつのレコードの間の順序を求める。
- etformat
  指定されたレコードの値を、任意のテキストに加工して出力する。

この他に、指定された範囲だけを抜き出すツールや、レコード数を求めるツールなどが提供されている。

etcompare で比較する順序は、レースディテクタに必要とされる性質を満たすように設計されている。しかし、マルチスレッドシステムのデバッグには、一般的に便利な概念と考えられるので、以下に述べる。

MPC++ では、ひとつの PE 内に複数のスレッドが動作する可能性があるので、ログファイル上で生起順に出力されているイベントが、確かにその順でのみ生起するようにプログラムされていたとは限らない。MPC++ で実行順序を保証するには、図4に示すように、三つの方法がある[2]。

[2]実はこの他に、メモリの内容が書き変わるのをビジーウェイトする方法があるが、実行性能上好ましくないことと、ノンプリエンプティブなスケジューリングを行うアーキテクチャでは正しく動作しないのでポータビリティに欠けることから、ここでは検討しない。

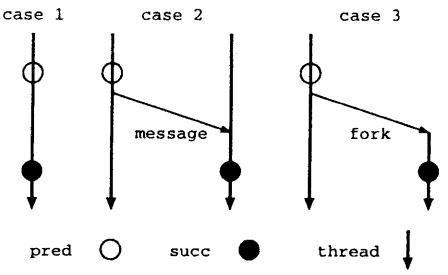

図 4: イベントの順序が決まる場合

1. ひとつのスレッド上のイベントは、必ずその記述順に生起する。
2. スレッド間でメッセージ通信を行っている場合、送信側スレッド上の送信前のイベントは、受信側スレッドの受信以後のイベントに必ず先行する。
3. スレッドがフォークされている場合、フォークしたスレッド上のフォーク以前のイベントは、フォークされたスレッド上のイベントに必ず先行する。

etcompare は、ふたつのイベントの間にこのような順序関係の推移閉包が成り立つかどうかを調べる。よって、etcompare が正しく動作するためには、プログラム実行中に発生したすべてのメッセージ送受信とスレッドのフォークが、記録されていなければならない。

3.4 レースディテクタ

前項で述べた基本解析ツールを用いて、レースディテクタを開発した。

3.4.1 仕様

レースディテクタの仕様は [1] を元にしており、次のようなものである。

1. そのプログラム中でアクセスされているすべてのメモリアドレスについて、
2. そのメモリをアクセスしているすべてのイベントについて、
3. 互いに順序がつくことを確認する。
4. ただし、両方が読み込みアクセスである場合は、順序がつかなくてもよいものとする。

ここで、“順序がつく”とは、[1] では、Lamport の分散時間半順序の意味で順序がつくという意味であったが、ここでは基本解析ツールの節で述べた順序を用

```
A=`etcat | etgrep "type=MEMORY_ACCESS"
         | etuniq "address"
         | etformat ">address\n"`;
for a in A; do
  B=`etcat | etgrep "address=a"
      | etformat ">pe:>record_num(>acc_type)\n"`
  C=
  for b in B; do
    for c in C; do
      if [ `acc_type_of b` = WRITE -o # OR
           `acc_type_of c` = WRITE ]; then
        # 少なくとも一方が書き込みであれば、
        if [ `etcat | etcompare b c` = x ]; then
          # x: 順序がつかない
          Alert;
        fi
      fi
    done
    C="C b"
  done
done
```

図 5: レースディテクタの実現例

いるのが適切である。例えば、同一 PE 内の複数のスレッドが同期をとらずに同じメモリに書き込んでいれば、競合と判定できる。

3.4.2 実現例

上の仕様は、あるメモリに対するアクセスの回数を n とすれば、n^2 回の比較を要求するものである。単純な実現としては図 5 のようになろう。シェルスクリプトをそのまま書くのは繁雑なのでやめ、シェルスクリプト風の疑似コードで表している。

この例題は、基本解析ツールの etgrep, etcompare を用いると、簡単に記述できることが分かった。これらは、イベントトレースの解析に一般に必要な要素機能の候補と考えられる。

4 まとめ

本稿では、再現実行機構を基礎としたデバッギング手法について提案を行った。また、その有効性を調べるために開発した、MPC++ 向けのイベントトレース解析ツールのプロトタイプについて、その実現と、そこから得られた知見について述べた。

今後は、この環境の上にデッドロックアナライザや、メモリリーク検出器など、共通性の高い解析ツールをいくつか実現してみて、イベントトレースログの解析に必要な機能の洗い出しを進める。

また、基本解析ツールは並列実装に適していると考えられるので、並列版の実装と性能評価をする予定である。

参考文献

[1] Fidge, C. J.: "Partial Orders for Parallel Debugging,", ACM SIGPLAN/SIGOPS Proc. of Workshop on Parallel and Distributed Debugging, pp. 183–194, May 1991.

[2] Lamport, L.: "Time Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, **21**(7), pp. 558–565, Jul. 1978.

[3] Fujita, M. et al.: "Automatic Generation of Some Results in Finite Algebra", Proc. of Int. Joint Conf. on Artificial Intelligence, Aug. 1993.

[4] Ishikawa, Y., et al.: "The Implementation of Parallel Programming Language MPC++" Proc. of JSPP '94, pp. 105–112, 1994.

[5] Rosenblum, D.S.: "Specifying Concurrent Systems with TSL", IEEE Software, May 1991.

[6] LeBlanc, T.J. and Mellor-Crummey, J.M.: "Debugging parallel programs with instant replay, IEEE Transactions on Computers, **C-3**(4), pp. 471–482, Apr. 1987.

[7] Kessler, P.B.: "Fast BreakPoints: Design and Implementation", Proc. of ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, pp. 78–84, Jun. 1990.

[8] 市吉他: "超並列プログラミング環境の検討", 情報処理学会第 48 回全国大会予稿集 4H-5, Mar. 1994.

[9] 市吉他: "超並列計算機 RWC-1 上 MPC++ プログラム向けの再現実行方式", 情報処理学会研究会報告 94-PRG-18-18 (SWoPP 琉球 '94), pp. 137–144, Jul. 1994.

[10] 伊藤他: "カスタマイズ機能を持つ分散デバッガーにおけるデバッグ支援機能の設計と実現" 電子情報通信学会技術研究報告 CPSY 95-42, Aug. 1995.