

Suci ライブラリのスナップショット API を利用した 並列デバッグツールの設計

Design of concurrent debug tool using Suci library's SnapshotAPI

上里 献一¹, 河野真治²
Kenichi Uezato, Shinji Kono

本論文では, 分散アルゴリズムであるスナップショットアルゴリズムを PC Cluster 専用の通信ライブラリである Suci ライブラリへ API として導入した. スナップショットアルゴリズムは状態検知の手法の 1 つである. 実装したスナップショット API の並列プログラミング専用のデバッガへの応用 [6] を示し, 実際の実装方法を示す.

1 はじめに

近年, PC Cluster 上の並列プログラムが多数開発されている並列プログラムの作成は, プログラム作成時に, PC Cluster の各ノードやネットワークを考慮する必要があり, 逐次プログラムよりも困難である. そのため, 並列プログラム用のデバッグツールの必要性が増している.

並列デバッガに必要とされる機能は, まず並列実行している各ノードのプロセスの状態や通信状態の確認である. これらの状態は, 逐次プログラムと異なり, 特定のプログラム・ステートメント (例えば, 行番号) に対して定義することはできない. これらは, スナップショット [3][7] と呼ばれる通信と各ノードの実行履歴に対して定義する必要がある. 本論文ではリング状に巡回するトークンを使用したスナップショットアルゴリズムを用いた並列デバッガを提案する. このデバッガは, 以下のような特徴を持つ.

- 計算を止めずに各ノードの状態を確認できる
- 状態確認時に必要な同期を中断せずに行える

デバッガの対象とするプログラムは, 我々の開発したユーザレベル通信ライブラリ Suci [2][4][5] を使った並列プログラムであり, 分散スナップショットおよびデバッガは, Suci ライブラリ上に実装される. こ

れにより, すべての通信状態をカバーする並列デバッガを実現することが出来る.

2 並列デバッガに必要な機能

逐次型のプログラム・デバッガには以下のような機能がある.

1. 変数値の確認
2. ブレークポイント (プログラムの中断・再実行)
3. ソースの表示
4. スタックトレース

これらに相当する並列デバッガの機能を実現するためには, 上記の 1 と 2 には分散スナップショットが必要である. この部分が並列プログラム特有な問題を有効となる. 3 と 4 は既存の gdb のような逐次型のデバッガを各ノードで実行されるプログラムに対して適用すれば良い. スタックトレースは, 特定ノードでプログラムがセグメンテーション・フォールトするような場合に有効である.

並列プログラム・デバッガに必要な機能は以下のように分類される.

1. 通信に関する機能
2. 大域状態に関する機能
3. プロセスの再現性に関する機能
4. デッドロックの検出
5. パフォーマンスの測定

通信に関しては各ノードが, 現在どのノードと通信し, どのノードからのメッセージを待っているかという情報や, 送受信されるメッセージの内容を調べる. 大域状態に関する機能とは, 各ノードの大域変数の内容を調べるものである. これらは, お互いに関連しており, 通信状態と大域状態は特定のタイミング

¹琉球大学理工学研究科
uezato@cr.ie.u-ryukyu.ac.jp
Intelligent System Engineering, Graduate School of Engineering and Science, University of Ryukyu

²琉球大学 科学技術振興事業団さきがけ研究 21(機能と構成)
kono@ie.u-ryukyu.ac.jp
University of Ryukyu, PRESTO, Japan Science and Ethnology Corporation.

で調べる必要がある。これが分散スナップショットである。

各ノードでの状態のスナップショットを取るタイミングで実行中のプロセスを見ると、そこには4つの通信パターンが存在する。4つのパターンとは、そのタイミングよりも後に送信が行われ、(図1-A)受信側ノードでそのタイミングよりも前に着く場合と、(図1-B)後に辿りつく場合)、送信がそのタイミングよりも後に行われ、(図1-C)受信側ノードにおいてタイミングよりも前に受信される場合と、(図1)後に受信される場合である。

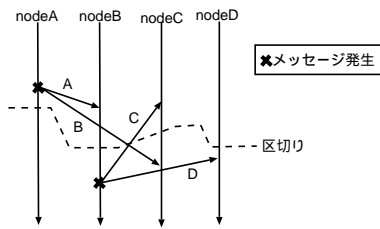


図1: 大域的な状態

3 スナップショットと再実行

この大域状態はプロセスの再現性 [8][9] やプロセスの状態をある時点での検知する場合の同期にも密接に関わってくる。プロセスの再現性は、並列プログラムでのデバッグに関する大きな観点である。並列プログラムでは、各ノードはネットワークを介し他のノードと通信を行っているので、実行の度に実行状況や結果が変わってくる。それが、並列プログラムのデバッグを複雑にする要因の1つとなっている。

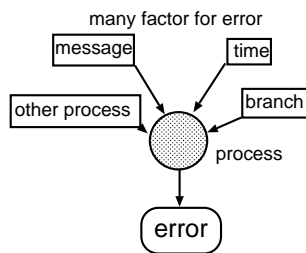


図2: プロセスの再現性

[8] では、ノード間で交換されるメッセージを前もって記録し、それにそって決定的な実行を行なうことで再実行を可能にしている。しかし、すべてのメッセージの記録が妥当でない場合もあると考えられる。

並列プログラムを、あるスナップショットにそって

中断した場合、中断したプログラムの大域的な状態は各ノードのメモリに格納されている。通信状態は、受信したメッセージと送信したメッセージのシーケンス番号によって観測される。

(図1-C)のパターンがある場合は、一般的には再実行を行なうことはできない。このパターンではスナップショット完了後に送信されたメッセージがスナップショット完了前に届く。スナップショット完了後に同じメッセージを送信することは、一般的には保証することはできない。これは、実行時間や、メッセージの受信、外部 I/O を呼び出すシステムコールなどにより異なるメッセージを送信する可能性があるためである。(図2)。

並列プログラムでは、バグを引き起こす特定のノードの状態が確率的に生じる場合があり、長い計算時間を得てしかバグ起らない場合がある。このような場合は、バグ発生までプロセスを何回も再実行することは非現実的であり、通信を含む大域状態の記録が必要である。

4 並列プログラム・デバッガの実装

ここでは、並列プログラム・デバッガの実現方法を説明する。リング状のスナップショット・トークンを用いた並列プログラム・デバッガの実装は、二つのパターンが考えられる。

1. N台でそれぞれ並列デバッガを立ち上げ接続する(図3-a)
2. 1台で並列デバッガを立ち上げ、リモートデバッグを行う(図3-b)

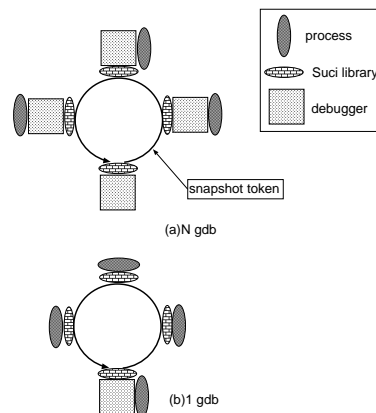


図3: 並列プログラム・デバッガ

上記の 1 の手法では gdb をスナップショットトークンで接続し、デバッグに必要な情報、並列デバッガの入力、出力のやりとりを行うというものである。2 の方法では、立ち上げた 1 台の並列デバッガだけで全てのノードのプログラムとやりとりをするという方法である。しかし、この場合、ノード数が増加すると並列デバッガへ負荷が集中する。そのため我々は 1 の方法を採用する。並列プログラム・デバッガに必要なスナップショットは、Suci ライブラリに内蔵される API を用いて実装される。

5 Suci のスナップショット API

スナップショットの実装方法は Lamport らの手法 ([1]) が知られている。[1] の実装ではマーカーと呼ばれるメッセージをすべての通信路に流し、マーカーを受け取ったノードはノードの状態を記録し、マーカーに自分のノードの通信情報をのせ、隣接するノードに送信する。この手法は PC Cluster のような完全結合並列マシンには向かないので、以下に示すリング形式 (図 4) を採用した。

リング形式は、個々のマーカーのかわりにスナップショット・トークンを PC Cluster の全てのノードを含むリング状の経路に流す。リングを巡回するトークンには、すべてのノードの通信シーケンス番号と大域状態を決定する情報を格納する。トークンの受信によって、各ノードは、その時点での大域状態に関する情報を得る。ここでスナップショット・トークンを含むメッセージの信頼性は Suci ライブラリ自体が保証する。

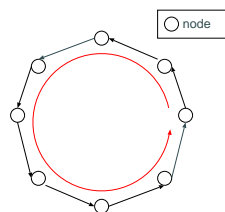


図 4: リング形式

リング形式は、Lamport らの手法よりも輻輳が削減できるが、大域状態はスナップショットトークンの一周によってのみ更新されるため、ネットワークなどの遅延によって決まる間隔での情報しか得ることはできない。一周時間が増加すると大域状態の時間精度は低下する。一周時間はノード数に比例し、トークンサイズもノード数に比例する。これがリング形

式のボトルネックとなるが、実際に計測を行ったところ、実際に際して問題のない結果が得られた (図 5)。

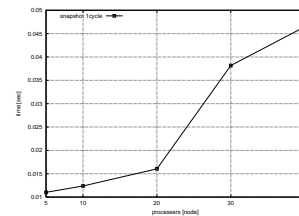


図 5: スナップショット・トークン 1 サイクルにかかる時間

5.1 Suci のスナップショット API

実装したスナップショット API を使う手順の例を示す。

```
int main(int argc, char **argv){
    int myid, max_id;
    int a;

    sock = start_datagram(myid);
    /*スナップショットトークンにのせる変数を登録
    する*/
    regist_value(&a);
    /*ID2 のノードがスナップショットをスタート
    させる*/
    if(myid==2){
        start_snapshot(sock, myid, max_id, 1, 1);
    }
    else {
        start_snapshot(sock, myid, max_id, 1, 0);
    }
    while(1){
        /*メッセージを受信した場合に 1 を返す*/
        if(datagram_ready(sock, 0, 1)){
        }
    }
    return 1;
}
```

この例は、スナップショットのトークンを常に循環させるというものであり、トークンにのせられる情報は、regist_value() で登録した変数値になる。

既存の API に修正を加えた datagram_ready() と新規で実装した start_snapshot(), regist_value() についてそれぞれ説明する。

- `datagram_ready(sock,sec,nsec)` `sec+nsec` の間 `socket` をブロックし, `select()` を行う. また, 受信したパケットのチェック, 組み立てを行う関数を呼び出している. スナップショットのトークンを受け取った時に, 情報を更新し, 次のノードへと送信している.
- `start_snapshot(sock,myid,max_id,rule,mode)` スナップショットを開始するのに必要な設定を行い, スナップショットを開始する.
- `regist_value(value)` スナップショットトークンで交換したい変数を登録する.

`datagram_ready()` を呼び出すことにより, スナップショット・トークンの送受信が自動的に行なわれる.

6 Suci のスナップショットを用いた並列デバグの利点

本並列デバグは, デバグ間通信にリングを巡回するトークンを使用するために, デバグのための通信量が少ない. 従って, デバグ対象となる並列プログラムへの影響が少ないと予想される.

また, スナップショットおよびデバグの通信がライブラリ・レベルで行なわれることと, Suci ライブラリがポーリング・ベースで動作することにより, 並列プログラムの行なうすべての通信を対象としたデバグを行なうことが出来る. アプリケーション・プログラム上で自分でデバグ用の通信を行なうような手法では, これは不可能である.

また, スナップショットを用いて通信を含む大域状態を取ることができるので, プログラムの再現性がある場合には逐次型のプログラムのような二分的なデバグ手法をとることができる. プログラムの再現性がない場合, あるいは, 極めて低い場合は, スナップショットの精度をあげるにより対処できる場合もあるが, デバグによる解決には限界があり万能ではない. このような場合は, 検証などの手法を併用することが必要である.

スナップショットによる情報収集は, デッドロックの検出や, パフォーマンスの測定にも有効である. Suci はポーリング・ベースのライブラリなので, ライブラリ自体ではデッドロックは起きないので, アプリケーションレベルでのデッドロックをライブラリレベルから検出することができる. パフォーマンスの測定は, 通信による待ち時間や計算時間に関する状態の

スナップショットをとることにより行なうことが出来る.

7 まとめと今後の課題

今回, 並列プログラム・デバグ実装の第一歩として Suci ライブラリにスナップショットを組み込むところまで実装できた. これからすべきことは, 並列デバグの実装と並列デバグとスナップショット・トークンの接続である. これには, 現在一般的に使われている `gdb` を並列デバグとして用いるというアイデアがある. `gdb` に Suci ライブラリを組み込むことでスナップショットトークンとの接続が可能となると考えられる.

並列プログラム・デバグはユーザにとって一般にデバグに用いられる `gdb` と近い感覚で使用できることを目標としている. また, 並列デバグに必要な「プロセスの再現性」のための機能もこれから実装していく.

参考文献

- [1] K. Chandy and L. Lamport: Distributed snapshots: Determining global states of distributed systems. ACM Transaction on Computer Systems, Vol. 3, No. 1, pp63-75, 1985.
- [2] 河野真治, 神里健司. UDP を使った分散環境とその応用. 日本ソフトウェア科学会第 16 回大会論文集, 1999
- [3] 屋比久友秀, 河野真治, トランスポート層を考慮したスナップショット・アルゴリズムの考察日本ソフトウェア科学会 第 19 回大会
- [4] 神里 健司, ユーザレベルのフロー制御をもつ通信ライブラリの設計及び実装 Summer Workshop of Parallel Processing 2001
- [5] 屋比久 友秀, 河野 真治. 並列分散ライブラリ Suci の実装と評価. システムソフトウェアとオペレーティングシステム予稿集, 2002
- [6] 上里 献一, 河野真治スナップショットを用いた PC Cluster 用デバグツールシステムソフトウェアとオペレーティングシステム予稿集, 2003
- [7] 亀田恒彦・山下雅史 著分散アルゴリズム近代科学社
- [8] 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グ. 先進的計算基盤システムシンポジウム SACIS2003 論文集, pp65-72, 2003
- [9] 實本英之, 高宮 安仁, 松岡 聡. 自律的な通信回復を行う Fault Tolerant MPI の実装と評価. 先進的計算基盤システムシンポジウム SACIS2003 論文集, pp199-200, 2003.