

スナップショットを用いた PC Cluster 用デバッグツール

上 里 献 一[†] 河 野 真 治^{†,‡}

本研究では、状態検知の方法の1つであるスナップショットを用いた PC Cluster 用デバッグツール構築について述べる。通信ライブラリには Suci ライブラリを用いる。また、デバッグにおいて発生しうる例題を提示し、スナップショットを用いた場合の改善法や利点について考察する。

making about debug tool for PC Cluster using Snapshot

KENICHI UEZATO[†] and SHINJI KONO^{†,‡}

In this paper, we suggest that making debug tool for PC Cluster using snapshot algorithm, and use Suci library. then show some problem ,occur debugging on distributed environment, we consider improvement and advantage about them.

1. はじめに

近年、通常の PC を LAN によって多数接続した PC クラスタを用いた並列計算が実用的用いられるようになった。ネットワークで繋がれた多数のコンピュータ資源を活用するグリッドコンピューティングも効果を挙げている。

このような環境でのプログラム作成は、MPI^(?), PVM^(?) や S/Core^(?) などの通信ライブラリを用いて行われる。しかし、これらのライブラリは並列プログラムを可能にするが、そのプログラミングの過程で必要となるデバッグ環境を提供していることは少ない。

本論文では、本研究室で作成した Suci ライブラリ^(?) を使った並列プログラムに対するデバッグツールを提案する。

Suci ライブラリとは UDP ベースの通信ライブラリであり、ユーザに通信に関するローレベルのチューニング環境を提供する。これにより、遅延対策や喪失対策が行え、Acknowledge 送信に関しても制御を行うことができる。

本論文で提案するデバッグツールさらに、巡回トークンを用いたスナップショットを利用する点に特徴がある。

本論文の構成は並列分散プログラミングのデバッグ、並列プログラミングに要求される機能、Snapshot を用いたデバッグ手法に対する考察を主とする。

加えてスナップショットを用いたデバッグ手法と、バリア同期を用いたデバッグ手法の比較も行う。

2. 並列分散プログラムのデバッグ

並列分散環境での開発、実験ではいかにデバッグを行うかということが重要になる。並列分散プログラミングでは複数のノードが独立で実行され、互いに通信しあうという実行環境のために逐次プログラムよりもデバッグが難しくなる。現在主に使われている gdb に代表されるデバッガは並列分散環境用に作られていない。そのため複数の N 台のノードに対するデバッグの場合、1 台で gdb を動かしてデバッグをするか、N 台 (または複数のノード) で gdb を動かしてデバッグを行うかのどちらかのスタイルが取られる。これではノード数が増えるとデバッグの手間はかなりのものになり、現実的なデバッグ手法とは言えない。

また、並列分散プログラムではプログラム状態の非決定性が存在する。通信の到着順序によっても分岐が変化してしまうのでは、デバッグ対象のエラーに対して再現性 (図 1) が無い。デバッグにおいてはこれらの点を踏まえて、通信が決定的になるようにテストする、非決定性をシミュレートする、バリア同期で止めてメモリ内容などを調べる、各ノードでログをとって、ログを解析するなどの方法がとられる。

3. 並列プログラム・デバッガに要求される機能

並列プログラム・デバッガに必要な機能、あるいは

[†] 琉球大学理工学研究科総合知能工学専攻
Interdisciplinary Intelligent Systems Engineering, Graduate School of Engineering and Science, University of the Ryukyus.

琉球大学工学部情報工学科

Information Engineering, University of the Ryukyus.

[‡] 科学技術振興事業団さきがけ研究 21 (機能と構成)

PRESTO, Japan Science and Technology Corporation.

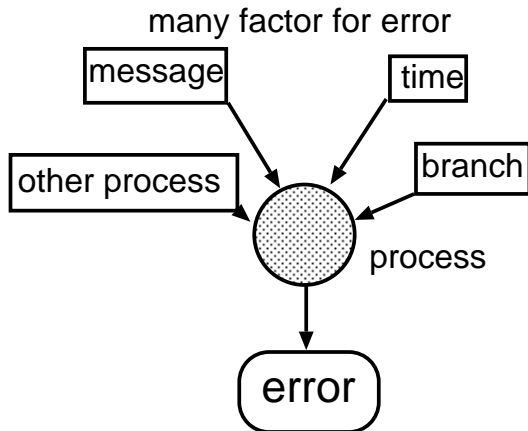


図 1 プロセスの再現性

あれば便利であるという機能を考えてみた。gdb を基準に考えるとまず、最低限必要な機能として以下のようなものがある。

- (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

このうち、3,4 に関しては、1 台のみで gdb を動かすことで十分な場合が多い。並列プログラムでは、通信から通信までの間の処理は、逐次プログラムと同様にデバッグすることが可能であり、その部分には gdb をそのまま使用することができる。

これに対して並列プログラム・デバッガでは、通信や各ノードの状態に依存するデバッグを行う必要がある。gdb の 1,2 に対応する機能として、すべてのノードの状態確認と並列プログラム全体の中断・再実行が必要であると思われる。さらに、並列プログラム・デバッガ特有の機能として、どのメッセージが既に送信され、どこまで受け取られたかという通信状態に関する機能、デッドロックの検出、パフォーマンスの測定の機能があると望ましい。

並列プログラム全体の中断・再実行にはバリア同期を用いる方法もある。本論文では中断 (図 2) を行うことで状態を検知するバリア同期と Snapshot の計測データも比較検証する。検証結果を用いて並列分散環境におけるデバッグにおいて Snapshot を採用する有効性を考察する。

4. Snapshot を用いたデバッグ

Snapshot は中断をせずに並列分散プログラムの大域的な状態を検知する検知方法である。

実行中の各ノードのプロセスの状態は常に変化し、お互いに通信しあっている。単純に特定時刻で個々の

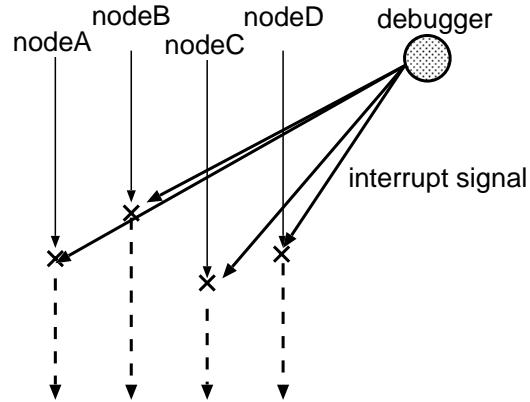


図 2 実行プロセスの中断

ノードの状態を記録するだけでは通信状態を含む大域的な状態とはならない。ここでは、大域的な状態を以下のように定義する。

並列分散プログラムにおける大域的な状態 (図 3) とは、メッセージの送信受信に関するものである。プログラム実行中の状態をあるタイミングで区切ると、そこにはいくつかのメッセージパターンが存在する。送信が区切った時間より前に発信された場合、受信ノードで区切り時間の前に辿りつくか後に辿りつくかというパターンがある。送信が区切り時間の後に送信される場合は、受信ノードにおいて区切り時間の前に受信されるか、後に受信されるかというパターンがあげられる。

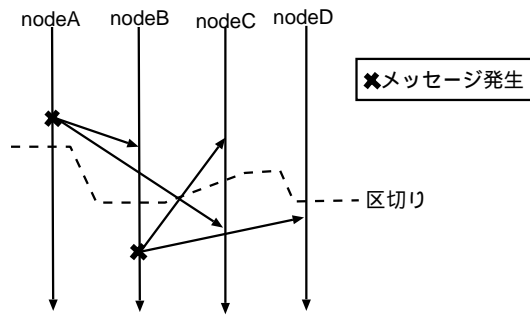


図 3 大域的な状態

なぜ、このようなメッセージ送受信のパターンを考慮する必要があるかというと、同期機構において考えた場合に同期が完了した後に、同期完了以前に送信されたメッセージが同期完了後に届く場合があるためである。そのためメッセージ状況を確認しておく必要がある。

大域的な状態を検出するためには、なんらかの通信または同期が必要である。一つの方法は、バリア同期

を用いて通信の無い状態で全てのノードを停止させることである。しかし、この方法では、ノードが停止してしまうために、検出しない場合の並列分散プログラムの動作と検出される場合の動作が大きく異なってしまふ。状態の検出の与える影響は小さい方が望ましい。影響が大きいと、デバッグを行うことによりバグが出現しないような状況が起きてしまう。これは、観測によって観測される対象が影響を受けると言う観測問題の一種である。

Snapshot は、各ノードで状態と通信履歴を記録するタイミングを決める特別なトークンを流すことにより大域状態を検出する方法である。この方法では、各ノードは実行と通信を中断する必要がないので、元の並列分散プログラムへの影響が少なくデバッグに適した方法であると考えられる。

5. Snapshot の実現手法

Snapshot アルゴリズムにおいては「Lamport らによる Snapshot の手法」が有名である。

[Lamport らによる Snapshot の手法]

Lamport らの Snapshot では、標識 (以下、マーカーとする) というメッセージを利用してネットワーク上の他のプロセスに Snapshot アルゴリズムを動かすタイミングを知らせる。マーカーは Snapshot アルゴリズム開始の合図として用いられることになる。各プロセスはアルゴリズムを開始すると現在のプロセスの状態を記録し、隣接のプロセスに対してマーカーを送信する。Lamport らによる Snapshot の手法の実装例を以下に示す。

(待機状態)

```
マーカーの到着を待つ;
マーカーが到着したら、(動作状態)に遷移;
```

(動作状態)

```
//ここから Snapshot アルゴリズム
プロセスの状態を記録する;
隣接するノードにマーカーを送信する;
while (マーカーを受信していない) do {
    マーカーの到着を待つ;
    マーカーの到着後、マーカーを除くネットワーク上のメッセージの状態を記録する;
}
(待機状態)へ推移;
```

Lamport らの実装では隣接するノード全てにマーカーを送るということになっているが、それを今回の実装においては改良した。以下に Snapshot 実装における実装上のポイントを示す。

[Snapshot をとるタイミング]

Snapshot をとるということはある時点での計算機の状態を記録することになる。記録をとるタイミングはスレッドを走らせて一定時間隔でとることも考えられるが、Suci ライブラリがスレッドセーフでないという点と、他のマシンと通信をするという前提上、スレッドを走らすと予想外の結果を引き起こす可能性がある。そのため実装では、以下のような手順を用いて行うことにした。

```
if(datagram_ready(sock,1,0)){
    datagram_read(sock,recvbuf,READSIZE);

    if(受信したデータがマーカーなら){
        take_Snapshot();
        //recvbuf には他のプロセスの Snapshot が連なっている
        //いるのでそれに自分の状態情報を更新して送る。
        send_Snapshot(recvbuf);
    }
}
```

- datagram_ready(sock,msec,nsec) msec,nsec の間ブロックし、受信できるパケットが完成されていれば 1 を返す。
- take_Snapshot() プロセスの状態情報をとる。
- send_Snapshot(char *) 引数として受け取った char バッファに take_Snapshot() でとられた自分のプロセスを付加して別のプロセスに送る。データが受信できる状態であれば、データを受信する。

バッファに納められた受信データのヘッダを見て、マーカーなら次のプロセスへ自分のプロセスの状態情報を更新したものを送信する。このように Snapshot をとるタイミングをマーカーを受け取った時とする。

Lamport らによる Snapshot の実装ではマーカーをすべての通信路に対して流す (マルチキャスト (図??)) 必要がある。

この方法では PC Cluster のような完全結合路の場合、単純に計算すると、プロセス (計算機) が N 個とした場合 $N*N$ 個のマーカーが流れることになり通信に与える影響が大きい。

また、Snapshot は、マーカーよりも後に送信されたメッセージがマーカーよりも先に到着してしまう場合に備えて、正しく大域状態を記録する必要がある。そのために、ここではメッセージにシーケンス番号を付けることにする。本手法ではシーケンス番号は Suci ライブラリが付加する。

本研究ではマーカーを PC Cluster の全ノードに対してリング状 (図??) に巡回させる手法を用いる。巡

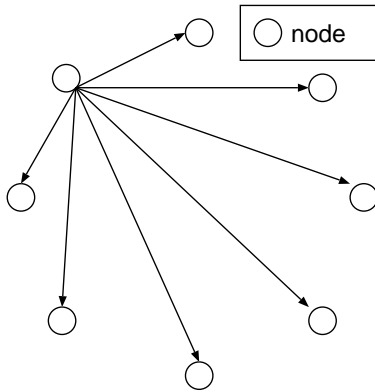


図 4 multicast

回するマーカーに大域状態を決定する情報がすべて含まれるようにすることで、マーカーを受信したノードは、その時点での大域状態に関する情報を得られることになる。メッセージの信頼性は Suci ライブラリ自身が保証するので、この Snapshot のレベルでは信頼性を仮定して良い。

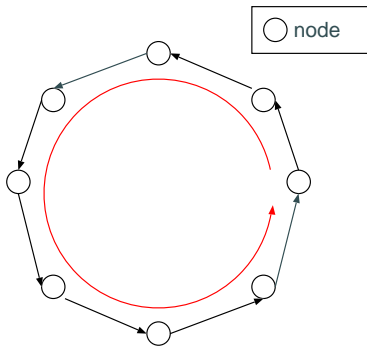


図 5 リング形式

マーカーを受け取ったノードは、マーカーの情報をを用いて自分の持っている大域状態に関するデータを更新し、マーカーに自分の最新の状態の情報を付加して、次のノードに送信する。最低限必要な大域状態を表す情報は、通信に関する情報であり、個々のノードが送り出したメッセージのシーケンス番号である。従って、マーカーは最小限、N 個のシーケンス番号を含む必要がある。マーカーが一周するのに N 回の送受信が必要なので、マーカーに N 台のノードの情報が付加されているとするとこの Snapshot アルゴリズムには、 N^2 の通信量が必要であることがわかる。

また、マーカーをリング状に巡回させることにより、自分自身が Acknowledge になる。つまり、マーカーが戻って来れば、それは、送り先と、巡回先全てを通っ

て来たことを示している。Suci ライブラリは、TCP に依存しないで Acknowledge を制御しているので、実際に Suci ライブラリレベルの Acknowledge を減らすことが可能である。これで通信量は半分に減ることになる。

大域状態はマーカーが一周することにより更新されるので、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。一周の時間が増加すると大域状態の時間精度は低下することになる。また交換しあう Snapshot メッセージの長さはノード数に比例するため、プロセス数が増えると扱われるマーカーに附属するノード情報のサイズも大きくなることが問題になる。

6. デバッグの例題

並列プログラミングには特有の性質があると以前に説明した。ここではそれらを考慮したデバッグの例題を提示し、Snapshot を持ちいてどのようにデバッグするかを示す。

[ギャザー実装におけるデバッグ]

ギャザーとは、並列分散プログラムにおいて全てのノードから情報を集める機能である。ギャザーを行うとギャザーを行ったノードに全てのノードから情報データを含んだメッセージが集中する。そのためノード数にもよるが負荷はかなり集中する。そのため、MPI においても実装では階層型を用いている。階層型とはノード間の通信を階層的に行うもので、そうすることによってギャザーを行ったノードに対する負荷の集中度は分散する。

```
int * gather(){
    //ノードが動いているかチェックする
    check_node();

    //階層構造を構築する
    make_gather_tree();

    //集めた node の情報を返す。
    return get_node_data();
}
```

階層型を取り入れたことによる起こるエラーに対するデバッグを考えてみる。ここで階層構造を決定する場合には各ノードの状態を考慮にいれる必要がある。効率の悪い階層構造を定義してしまうとギャザーを行って情報が集められるまでの時間が予想以上にかかってしまったりすることが考えられる。重たい計算を行っているノードを階層構造の上部に設置した場合には、通信に対する処理が遅れてしまうということもありえる。その場合に階層構造を組み直すためには、

どのノードで詰まっているのか特定する必要がある。これを検知する手段として Snapshot を用いることができる。Snapshot を用いると全体の状態をトークンが一周することでとれるので通信量を押えられる。また、見つけにくいエラーである、ノードまたは階層を1つ飛ばしてギャザーが行われた時に、どの部分で抜けが起こったのかを見つけることも可能である。

別のデバッグ問題としてノードの状態により集められる情報の値が変わってくるという問題があげられる。ノードは互いに通信しあって実行している。分岐は通信の受信順序によっても変わってくる。ギャザーで得られた情報が本当に正しいものなのか確認する必要がある。デバッグにおいてはそれもチェックする必要がある。このデバッグを Snapshot を用いて行うと、正解とされる情報を Snapshot で得、比較することができる。Snapshot を用いてギャザーが行えるとも考えられるが、ギャザーにはギャザーの利点がある。Snapshot はトークンを巡回させながら状態を検知するため、ある特定時間で同時に状態をとるのは少し違うため、ギャザーを用いた方が更に限定した時間間隔での情報を得ることができるのである。ここであげたデバッグは並列分散プログラムの非決定性が関わっている。

Snapshot は非中断性を持ち合わせているので、観測のために観測対象に影響を及ぼさない。そのため、ギャザー動作中の流れがみることができ、どのノードからどのノードへ情報が受け渡しされているのかが見て取れることができ、デバッグが行いやすい。

Snapshot を用いるとこのようなデバッグが行うことができる。

7. Snapshot のオーバーヘッドの測定

デバッグに Snapshot を用いる有効性を『中断』するかどうかという視点で検証する。Snapshot を用いると中断をせずにデバッグを行える。中断をせずにデバッグを行えるという点の有効性を検証するために、中断を行うバリア同期と比較する。バリア同期は、中心プロセスを立て、そのプロセスが同期を管理するという実装にした。その場合、デバッグを中心プロセスとした場合デバッグにかなりの負荷が集中する。プロセス数が多くなればなるほど負荷は集中する。その負荷の影響についても後で考察する。他にプロセス台数を増すごとの Snapshot メッセージの一周の時間も計測する。

計測のための実行環境として LAN 間を1つのスイッチで結ばれた40台のクラスタマシンを用いた。その並列環境上で、Suci ライブラリ上で実装した Snapshot アルゴリズムの検証用プログラムとバリア同期 (図 6) を実装した検証用プログラムを走らせた。

8. 計測結果

検証において

- Snapshot における同期終了までの実行時間 (図 7)
 - Snapshot メッセージ一周にかかる時間 (図 8)
 - バリア同期における同期終了までの実行時間 (図 9)
 - バリア同期における最大応答時間 (図 9)
- をそれぞれ計測した。

「Snapshot メッセージ一周にかかる時間」は先にあげた、Snapshot が一周しないと全体の状態を検知できないという欠点がどの程度実際に影響するかということを検証するために計測した。

プロセス数	実行時間	1 サイクルにかかる時間
5	28.6	0.106
10	28.6	0.109
20	33.6	0.110
30	42.7	0.112
40	56.4	0.114

表 1 Snapshot における同期終了までの実行時間と1サイクルにかかる時間

プロセス数	実行時間	最大応答時間
5	8.05	4.00
10	18.1	14.0
20	38.1	34.0
30	58.2	54.0
40	78.2	74.0

表 2 バリア同期における同期終了までの実行時間と最大待ち時間

9. 評価と考察

Snapshot とバリア同期の計算開始から同期終了までの実行時間を見てみると5台、10台まではバリア同期の方がタイムが速い。しかし、台数がしだいに増えてくるとそのタイムが逆転してくる。その結果の大きな要因となっているのはバリア同期の応答時間だと結果を見ると考えられる。例えば30台においての結果では78秒の実行時間のうち74秒が応答時間となっている。これはベンチマークの内容によるものだと考える。プロセスの状態情報として用意した変数の初期値は個々のプロセスのIDを基に計算していて、最小で200から最大で3000の値からデクリメントしているそのため結果をみて分かるように最初に待ち合わせポイントに到着したプロセスは4秒で既に待ち合わせポイントに到着している。しかし、他のプロセスはまだ待ち合わせポイントにたどり着いていないため、その間かなりの時間を同期確認のメッセージを待つことになる。

Snapshot においても、データはバリア同期と同じようにばらついている。しかし、タイムはバリア同期よりも速くなっている。それを Snapshot とバリア同期の違いという点から考えてみると、メッセージの量、通信の集中度によるものだとも考えられる。バリア同期において送信されるメッセージの量を考えると CLIENT プロセス数 N がそれぞれ 1 回ずつ自らのプロセスの状態情報を送ると全体で N のメッセージが中心プロセスに集中することになる。バリア同期において各プロセスが 1 回ずつ状態情報を送るということは、Snapshot において Snapshot メッセージが一周することに相当する。メッセージが一周するのに要する送信回数は N 個である。これはバリア同期のメッセージ数と等しい。しかし、通信の集中度は Snapshot においては低い。ここで特に要点となるのはメッセージの集中度であると考えられる。Snapshot では 1 つのプロセスに通信が集中することはおこらない。しかし、バリア同期ではこの集中が起こる。ここで考えられるのは、中心プロセスの処理速度である。仮に処理できるメッセージが毎回 1 つならば、多量のメッセージが一斉に押し寄せるとそこで待ち行列が発生してしまう。プロセス数が増えるとその待ち行列はさらに長くなると予想される。こうなってしまうとプロセス数の多い並列分散環境においてバリア同期を実装することは非現実的なものになってしまう。

このように考察したが、実際にバリア同期より Snapshot が速くなった理由はどうか?それを改めて考えさせられるデータが「同期条件を同じくしたバリア同期の実行時間と最大待ち時間」の結果である。この結果を見ると、5 台と 40 台での実行時間の差は誤差の範囲で一致している。計測において全てのプロセスの初期値を同じくした。計測が開始されると全てのプロセスはデクリメントをはじめ。単純に考えるとほぼ同時に待ち合わせポイントにたどり着くと考えられる。実際の結果をみると最大応答時間は 40 台でも 2msec に満たないものだった。しかし、台数ごとの最大応答時間は、台数が増えるごとに確実に増えている。これをまとめるとバリア同期の実行時間は、個々のプロセスが待ち合わせポイントにたどり着くタイミングのばらつきが最も影響し、通信の集中度もある程度影響するということである。

Snapshot とバリア同期の決定的な違いは待つか、待たないかということである。待てば、同期は確実にとれるのだが、プロセス数が膨大になったり、通信の集中度がましてしまえば、そのロスは大きくなる。単純に計算量を考えると、バリア同期を用いて N 個 CLIENT プロセスが同期する場合には、 N 個全てのプロセスから中心プロセスに待ち合わせポイントにたどりついたことを知らせ、中心プロセスから同期確認メッセージを受け取るで、 $O(2N)$ になる。Snapshot

においては、 N 個のプロセスが他の $N-1$ 個のプロセスを確認しないといけないので $O(n^2)$ の計算量である。しかし、バリア同期では待ち合わせポイントにたどりつく時間が各プロセスバラバラであるために単純な計算量で予想できない結果につながっているのだと考えられる。

並列環境によって評価の基準が計算量でなく、実行時間とすれば大規模になる並列分散環境では Snapshot が有効になる。しかし、Snapshot が有効である環境とそうでない環境がある。そもそもリング状に実装することができるのは完全結合型の通信環境であるからで、全ての環境にリング状のプロセスが築けるわけではない。リングが途中で切れてしまったり、メッセージが遅延してしまったりは Snapshot 自体がなりたたなくなる。しかし、結果として完全結合型の通信環境においては Snapshot はバリア同期よりも有効な手段であるといえる。

10. まとめと今後の課題

今回の計測のために作成したベンチマークプログラムでは、同期アルゴリズム以外は動いておらず、同期アルゴリズム以外でのメッセージ送受信は行っていない。そのため、実際に並列で計算をしつつ同期をとるという場合での厳密なデータはとれていない。実際の計算を行いながら同期をとるには Snapshot にはクリアしないといけない問題点がいくつか存在する。

Snapshot を用いないデバッグツールにおいては、ある時点でのプロセスの状況を確認するためにプロセスの中断をする必要がある。中断を行えば、その時点での正確な状況がとれるが、状況確認が何度も行われるようになると中断から実行再開までのターンアラウンドタイムが大きな問題となる。Snapshot を用いた場合、そのターンアラウンドタイムが発生しないことが計測結果でも影響していると思われられる。

また、Snapshot が実行中、常に行われていれば、実行中のログもとることができる。この機能を実装すれば、通信が実行された時に、そのメッセージがどのような内容でどこへ、いつ送信された後から確認することができる。これは先にもとりあげた並列プログラミングにおける難しさである、実行中の分岐の必決定性における強みになる。デバッグツールにおいてエラー箇所や原因を特定することができるということは重要な項目である。一般的な逐次的なプログラムにおけるデバッグにおいてはプログラムを順に実行していけば、エラー箇所に辿り着けるが、並列分散プログラムにおいては、エラー部分に辿りつくまでが困難であることもありえる。限定的な特定の条件が重なった場合にだけ起こりえるエラーの場合、時間的な条件も複雑に絡み、エラー再現までに同じ条件で長時間実行する必要もでてくるとデバッグが厳しくなる。実行中

の全体の状況をログとしてのこせれば、再現性をもたせデバッグが行いやすくなり、エラー箇所の特定制も幾分簡単になる。一般的なデバッグツールの gdb ではコアファイルを用いてデバッグを行うことができる。Snapshot でとったログを加えてコアファイルを拡張し分散環境全体に対するコアファイルとすることも可能である。

gdb には遠隔デバッグ機能もある。しかし、並列に複数のプロセスが動いている場合のデバッグではプロセス1つずつにアクセスしてデバッグすることになり、同時に全プロセスを対象としたデバッグは行うことができない。それに gdb 自体は通信機能を持っていない。そのため遠隔デバッグにおいて必要な通信環境のチューニングが容易に行うことができない。そのため我々は、デバッガ自体に通信ライブラリを組み込むことでデバッグ用のチューニングを行えるようにしたいと考えた。本研究ではその通信ライブラリとして Suci ライブラリを考えている。Suci ライブラリは UDP ベースである高速性に加えて、輻輳制御(フロー制御)ができるなどユーザレベルで通信環境のチューニングが行える特徴を備えている。まだ未実装ではあるが Suci ライブラリ自体に Snapshot を組み込むこともできる。Suci ライブラリを用いることで、先にも書いた Snapshot メッセージ受信時の ACK 削減も実現できる。

今後の課題としては、Suci ライブラリ上の API として実装した Snapshot を Suci ライブラリ内部へ組み込むことがあげられる。それに加えて、並列分散プログラム向けの新しいデバッグスタイルを引き続き考える必要がある。現在のデバッグスタイルは同時に複数のプロセスに行うというよりも、プロセス1つずつに対して行うスタイルであるためである。Snapshot には中断をせずに常に状況が確認できるという強みがあるが、逆に人間の目から見れば、大量に集められるデータに理解が追いつけないため、プロセスのどのような状態を集め、それをどう整理するかも実装上において重要なポイントである。加えて、デバッグのためのユーザインターフェースについても考慮していきたい。

参 考 文 献

- 1) 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
- 2) 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
- 3) 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 Parrallel Processing 2001
- 4) 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
- 5) 亀田恒彦・山下雅史 著分散アルゴリズム近代科学社

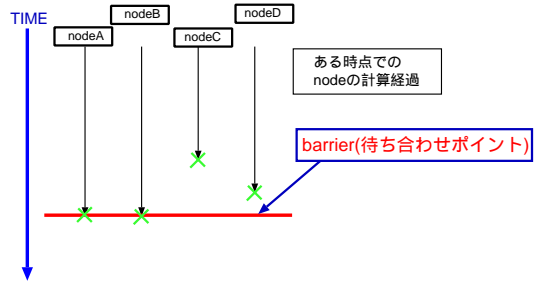


図 6 バリア同期

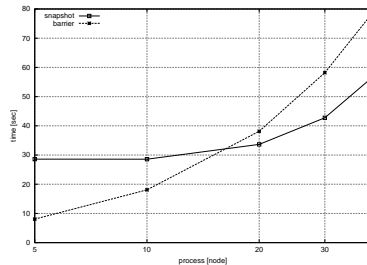


図 7 Snapshot と barrier 同期における同期終了までの実行時間の比較

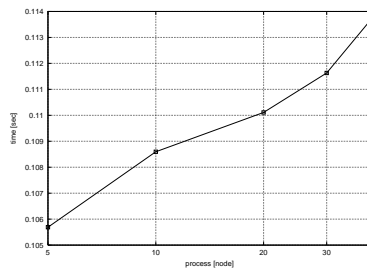


図 8 Snapshot メッセージ 1 サイクルにかかる時間

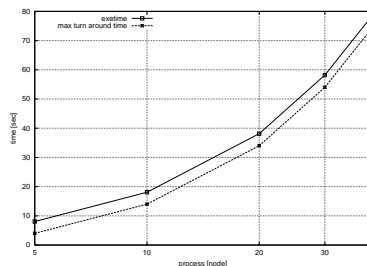


図 9 バリア同期における同期終了までの実行時間と最大応答時間の比較