6K-01

並列分散軽量プログラミング言語に適したブレークポイント手法

工藤　朋哉†　　小宮　常康‡

電気通信大学†,‡

1. はじめに

プログラミングのデバッグ機能としてブレークポイントがよく知られている。並列プログラムのデバッグにおいては複数のプロセスを一つのグループとしてまとめて扱いたいことがある。その場合 SPMD のように全てのプロセスが同じプログラムを実行する場合は一つのブレークポイントによりすべてのプロセスを停止できるが、異なるプログラムを実行する場合にはそれぞれのプログラムに適切なブレークポイントを設定する必要があり煩わしい。またプロセス群を一斉に停止させる安直な実装としてポーリングが考えられるがオーバーヘッドが大きい。本研究では複数のプロセスが異なるプログラムを実行する並列プログラムにおいて、オーバーヘッドを考慮しつつ、一つのブレークポイントで全てのプロセスを停止させる手法について提案する。

2. 対象とする並列分散処理系の設計

逐次プログラミングと比較して並列分散プログラミングはプロセス間通信、同期など考慮すべき点が多く複雑である。従来からプログラマの負担を軽くするために様々なプログラミングモデルが研究されてきた。本研究では Places[1]のデザインを参考に Ruby に並列分散処理を抽象化して記述できるライブラリを実装した。Places では並列に実行するプログラムを place オブジェクトで抽象化し、place 間でのメッセージパッシングと共有メモリを備えている。本研究のライブラリで２つのフィボナッチ数並列に求める例をソースコード１に示す。`$ref` はその place を生成した place への参照である。

Places のベース言語である Racket と本研究の実装に使用している Ruby は軽量プログラミング言語に分類される言語である。軽量プログラミング言語の軽量とはプログラマの負担の軽重を指し、事前にコンパイルが不要なインタプリタであること、動的型付けであること、関数オブジェクトが扱えることなどが特徴とされている。実行速度では C や FORTRAN に及ばないものの、軽量スクリプト言語は並列分散プログラムにおいても SPMD などの特定のスタイルに限定されない取り回しの良さが実現できると考えられる。しかし自由度の高い並列分散軽量プログラミングのデバッグに対しては従来のブレークポイント方法ではいくつかの問題が生じる。

```
def fib(n) …
def start_fib(n)
  p = Place.new(n) do |n|
    $ref.put fib(n)
  end
end
p1 = start_fib(n1)
p2 = start_fib(n2)
p [p1.get, p2.get]
```

ソースコード 1: 2 つのフィボナッチ数並列に求める例

3. 従来のブレークポイントの問題点

ブレークポイントは多くのデバッガに備えられており、本研究のライブラリでもソースコードに `breakpoint` メソッドを挿入することで実行中のプロセスを一時的に停止させ、対話形式のデバッグモードに入る事ができる。複数のプロセスが並列に実行されるプログラムを対話的にデバッグする場合、どのプロセスと対話しているのかを明確にする必要がある。また複数のプロセスが協調して動作する並列プグラムをデバッグする際にはブレークポイントで停止させるプロセスは一つではなく、関係する複数のプロセスを止める必要がある事がある。

複数のプロセスが同じプログラムを実行する場合は一つの `breakpoint` メソッドで関連する全てのプロセスを停止させる事ができる。このような SPMD のプログラムを記述するのに利用できる並列機能の一つに `Fork-Join` があげられる。例をソースコード 2 に示す。しかし

Breakpoints for parallel distributed lightweight languages.
† Tomoya Kudo, The University of Electro-Communications.
‡ Tsuneyasu Komiya, The University of Electro-Communications.

異なるプログラムを並列に動かす場合、ブレークポイントの適切な挿入箇所が明確ではない場合がある。

ソースコード3はバイトニックソートの実装例である。このプログラムの bitonic_sort メソッドは st 回目の再帰までは新たなプロセスを生成することにより $2^{(st)}$ プロセスを用いて並列に計算を行うが、それ以上はプロセスを生成しないため再帰の深さにより実行するプログラムが異なる。

4. 提案手法

4.1 アクティブな place

まず当研究ではアクティブな place というものを定義している。アクティブな place とは標準入力と標準出力が利用者のキーボードと端末に接続されている状態である。アクティブな place は常に 1 つであり、初期状態は Ruby トップレベルに相当する place である。アクティブな place を切り替えることで、各プロセスに対して逐次プログラムと同じ感覚でデバッグを行うことができる。アクティブな place の切り替えは引数に place への参照を持つ cp メソッドによってプログラム中、または対話中に行う。アクティブでない place からの標準出力はバッファに送られ、アクティブになるまで保持される。ただし標準エラー出力はアクティブでない place であっても端末に出力される。この際どの place からの標準エラー出力であるかの情報が一緒に表示される。

4.2. Communicator Group (CG)

CG とは Places において関連する複数の place をグループとして扱うことで並列プログラムのより高度な抽象化を実現する機能である。CG によりグループ化された place に対する同じ手続きの適用、パイプラインのようなプログラムで必要となる一つ前の place からメッセージを受け取り、次の place にメッセージを送る、などのプログラムを共通のコードで簡潔に表現できる。Places の Fork-Join や Pipeline などの並列機能は CG を用いて実装されている。

本研究の処理系にも CG の概念を拡張しつつ取り入れている。本研究の CG は Ruby の Array クラスを継承しており Array クラスの多くのメソッドが同様に利用できる。また CG に属する place を全て一時停止させる CG#the_world が用意されている。CG に属する各 place は計算を担当するスレッドの他に一時停止シグナルを受け取るスレッドを持っており、シグナルを受けた場合は計算担当スレッドを一時停止させて対話状態になる。place は実行前にコード解析され、ブレークポイントを実行する可能性があるかどうかのフラグを内部に持っている。計算を担当するスレッドは実行ステップ毎に一時停止するかどうかをチェックするが、CG に属する全ての place がブレークポイントを実行する可能性がない場合、一時停止のためのチェックは行わず、また一時停止シグナルを受け取るスレッドも生成しない。これらの設計により、少ないリソースで複数のプロセスの一時停止機能を実現している。

```
N.times do
  p = Place.new(P) do |n|
    …
    $ref.put count.to_f / n
  end
  cg.push(p)
end
puts cg.map{|p|p.get}.sum * 4 / N
```

ソースコード 2: Fork-Join によるモンテカルロ法(抜粋)

```
def bitonic_sort(up, st, array)
  if array.size <= 1
    return array
  else
    if st > 0
      p1 = Place.new …
      p2 = Place.new …
      …
    else
      …
```

ソースコード 3: バイトニックソート(抜粋)

5. 今後の課題とまとめ

既存のブレークポイント手法を並列分散プログラムに適用する際の問題を整理し、並列分散軽量プログラミング言語に適したブレークポイント手法を提案、実装した。今後の課題としてはブレークポイント以外の並列分散軽量プログラミング言語に適したデバッグ手法を研究していく。パフォーマンスと使いやすさでより優れたライブラリを実装していきたい。

参考文献

[1] Kevin Tew et al., Places: adding message-passing parallelism to racket, In Proceedings of the 7th symposium on Dynamic languages, pages 85-96. 2011.