平成 26 年度 修士論文

# 対話型環境を持つ
# 分散型スクリプト言語の設計と実装

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1353008　工藤 朋哉

指導教員　小宮 常康　准教授
多田 好克　教授
新谷 隆彦　准教授

提出日　平成 27 年 1 月 26 日

# 目次

# 図目次

# ソースコード目次

# 第 1 章

# はじめに

コンピュータの進化に従い、CPU のコア数は増加し、コンピュータは様々な並列処理により性能向上を図るようになった。またビッグデータの普及などに後押しされ、1 台のコンピュータだけではなく複数のコンピュータを用いて処理を行う分散処理も一般的になってきている。しかし並列分散プログラミングは逐次プログラミングと比較して複雑である。同期の問題、並列処理に適したアルゴリズムの選択、非決定性によるテストの難しさなど、考慮すべき点が多い。

この問題を解消するため、従来より並列分散プログラミングを使いやすくするためにさまざまな研究が行われている。しかしその対象は C などの低レベル言語向けのものが多くを占め、Ruby のような軽量スクリプト言語が取り扱われることは少ない。低レベルな言語は高速であるが、開発が難しく、生産性が低くなる傾向がある。高級言語は低級言語と比較して絶対的な速度では劣るものの、習得のしやすさ、生産性の高さなどメリットもあり、それは並列分散プログラミングにおいても同じである。

本研究では逐次型の軽量スクリプト言語である Ruby に、高度に抽象化された並列分散処理の仕組みと、並列処理に適したデバッグ手法を提案し、実装した。Ruby はクラス定義、ガベージコレクション、強力な正規表現、マルチスレッド、例外処理、イテレータ、クロージャ、Mixin、演算子オーバーロードなどを持つ、非常に多機能な軽量スクリプト言語である。可読性の高い構文を持ち、整数や文字列なども含め全てのデータ型がオブジェクトであるという純粋なオブジェクト指向言語である。

既存の逐次プログラミング言語に並列分散処理の仕組みを加えた研究の一つとして、Racket[7] を対象にした Kevin Tew らの Places[2] がある。Racket は Scheme[8] から派生したプログラミング言語であり、関数型に分類される言語である。本研究は Places を参考に、place のセマンティクスを取り入れ並列処理の抽象化を実現した。その上で並列分散プログラミングにおけるデバッグの難しさを解消するため、並列分散プログラミングに適した対話型環境を設計、実装した。

文法の設計においては Ruby の文法から逸脱しないことを一つの前提とした。Ruby の並行処理で使用される Thread を参考に、Ruby ユーザーにとって分かりやすい構文を設計した。またプログラミングの生産性を考慮する上で総合開発環境は欠かせないものとなっているが、本研究の処理系を使用しても問題なく使用できるよう配慮されている。

広く使われている Ruby の実装は C 言語であるが、本研究の処理系の実装については Ruby ライブラリのみで行った。また実装に使用しているライブラリは Ruby 標準ライブラリのみである。そのため、すでに Ruby がインストールされている環境であればすぐ本研究の並列分散機能を使うことができる。プロセス間の通信も dRuby を使用しているため、特別なプロトコルなどを必要としない。環境に依存しないのも Ruby の強みである。

Ruby と Racket はどちらも動的言語に違いはないが、プログラミング言語としての性質は大きく異なる。Places を参考とする上で特に問題となったのは、Racket の持つ非常に強力なマクロと、洗練された高階関数である。Ruby ではどうしても代替できない機能もある。しかし逆に Ruby であるから実現できた機能もある。本研究では性質の違う 2 つの言語を比較することで、プログラミング言語に対する新しい知見を得ることができた。

本研究ではプログラミング言語の設計という分野を扱う。プログラミング言語の設計に最適解はなく、設計者の価値観による影響も大きい。本研究をするにあたって、Ruby の価値観を大事にしたいと考えて設計を行った。メタプログラミング Ruby[1] の序章で、Ruby の親であるまつもとゆきひろ氏は次のように述べている。

「Ruby は君を信頼する。Ruby は君を分別のあるプログラマとして扱う。Ruby はメタプログラミングのような強力な力を与える。ただし、大いなる力には、大いなる責任が伴うことを忘れてはいけない」

# 第 2 章

# 背景

## 2.1 抽象化

抽象化とは、いくつかの概念をまとめてより高次の概念を作り出すことである。プログラミングは機械語から始まり、星の数ほどのプログラミング言語が生まれたが、現在でもプロセッサが読んでいるのは機械語であり、プログラミング言語は機械と人間の橋渡しをしているに過ぎない。

プログラミング言語の歴史とは抽象化の歴史でもあった。たとえばサブルーチンは何度も繰り返される処理を抽象化したものであり、オブジェクト指向におけるオブジェクトとは、データとそれを走査する手続きをまとめて抽象化したものである。抽象度の高いプログラムは、抽象度が低いプログラムよりも簡潔になり、意図が明確になる。例として「配列 ary に n に等しい要素があれば、それが何番目か返す」というプログラムを、C 言語で記述したものをソースコード 2.1に、Ruby で記述したものをソースコード 2.2に示す。

ソースコード 2.1 find_index(C)

```
1 int find_index(int n, int *ary, int len) {
2   int i;
3   for (i=0; i<len; i++) {
4     if (n == ptr[i]) return i;
5   }
6   return -1;
7 }
```

ソースコード 2.2 find_index(Ruby)

```
1 ary.find_index{|x| n == x}
```

C 言語にはループを抽象化する機能がないので、毎回 for ループを記述する必要がある。しかし Ruby では「配列に条件を満たす要素がある場合、それが何番目か返す」という処理を抽象化できる。高度な抽象化により、記述量が減るだけではなく、「なにをした

いのか」を明確に表している。

フレデリック・ブルックスは人月の神話 [5] の中で、“適切な高水準言語を使えば、プログラミングの生産性が、5 倍も上昇する可能性がある。” と述べている。言語の違いによる生産性の向上は、抽象化によってのみ実現できる。

## 2.2 Ruby における既存の並列/分散機能

### 2.2.1 Thread

Ruby では簡単にマルチスレッドプログラミングをすることができる。現在の Ruby のスレッドはネイティブスレッドであるが、グローバルインタプリタロックを用いているため、同時に実行されるスレッドは常に一つである。IO 待ちの軽減などには有用であるが、複数の CPU コアを活用することはできない。この問題は Python など他のスクリプト言語にもみられる。

### 2.2.2 dRuby

dRuby[6] は Ruby 専用の分散オブジェクトシステムである。dRuby は以下の機能を提供する。

- リモートオブジェクトへの参照を作る (DRbObject)
- リモートからメソッドを受け取り、メソッドを呼び出し返り値を返す (DRbServer)
- メソッド呼び出しの引数/返り値を Marshal でバイト列に変換 (マーシャリング) して渡す

プロセス間の通信は任意の方法を用いることができるが、TCP が使われることが多い。dRuby を用いて、任意のジョブを別のプロセスで実行するためのジョブサーバーを実装する例をソースコード 2.3とソースコード 2.4に示す。ソースコード 2.3では Queue をリモートから参照できるようにしている。その後 Queue から要素を pop し、eval する。ソースコード 2.4では Queue へのリモート参照を作り、リモート経由で Queue に要素を push している。

ソースコード 2.3 ジョブサーバー (サーバー)

```
1 require 'drb/drb'
2 q = Queue.new
3 DRb.start_service("druby://localhost:12345", q)
4 loop do
5   eval(q.pop)
6 end
```

ソースコード 2.4　ジョブサーバー (クライアント)

```
1 require 'drb/drb'
2 q = DRbObject.new_with_uri("druby://localhost:12345")
3 job = "(1..100).inject(:+)"
4 q.push(q)
```

dRuby ではリモートプロセスにあるオブジェクトはローカルでは DRbObject のインスタンスとして扱われる。このオブジェクトはリモートオブジェクトのプロキシのようにふるまい、メソッドが呼び出された場合はリモートオブジェクトに転送される。

メソッド呼び出しはそれを受け取ったプロセスの DRbServer オブジェクトが処理する。リモートメソッド呼び出しの引数や返り値には任意の Ruby オブジェクトが使える。デフォルトではオブジェクトをはマーシャリングして渡され、受け取った側が元のオブジェクトに戻す。つまりコピーである。マーシャリングできないオブジェクトは DRbObject のインスタンスとして扱われ、リファレンスのようなふるまいをする。オブジェクトをマーシャリングできなくする DRbUndumped という module が用意されており、これにより任意のオブジェクトをリファレンスとして渡すことができる。

先ほど実装したジョブサーバーにはいくつかの問題がある。まずジョブを文字列として渡している点である。関数や変数などの環境を引き継ぐことができず、文字列として記述する処理は IDE で文法チェックの対象にならず、また実行される際も単に eval されるため例外の位置も特定できないなど非常に使い勝手が悪い。このような場合、ソースコード 2.5、ソースコード 2.6のようにジョブをクロージャで渡したいと考えるのが自然である。

ソースコード 2.5　ジョブサーバーにクロージャを渡そうとする例 (サーバー)

```
1 require 'drb/drb'
2 q = Queue.new
3 DRb.start_service("druby://localhost:12345", q)
4 loop do
5   q.pop.call
6 end
7
8 require 'drb/drb'
9 q = DRbObject.new_with_uri("druby://localhost:12345")
10 job = Proc.new do
11   (1..100).inject(:+)
12 end
13 q.push(job)
```

ソースコード 2.6　ジョブサーバーにクロージャを渡そうとする例 (クライアント)

```
1 require 'drb/drb'
2 q = DRbObject.new_with_uri("druby://localhost:12345")
3 job = Proc.new do
4   (1..100).inject(:+)
5 end
6 q.push(q)
```

しかしこのプログラムは期待通りには動かない。Ruby において Proc や lambda はマーシャリングが不可能なオブジェクトなのである。クライアントはサーバーの Queue#push を引数 job で呼びだそうとするが、この job はマーシャリングが行えない Proc クラスであるため、サーバーに送られることなくローカルで実行される。Ruby ではメソッドを含め全てのブロックは内部で Proc クラスとして扱われているため、非常に大きな制約となる。

クロージャが渡せないというのはとても大きな問題である。Ruby のクロージャは Proc クラスで表現される。同じような働きをするものに Kernel.#lambda メソッドがある。Ruby のクロージャは意識して使わない人もいるかもしれないが、実は多くの場所で使われている。Ruby のメソッドの引数としてのブロックはクロージャである。

```
1 array.each do |a|
2   ...
3 end
```

上記のようにブロックを引数に取るメソッドがあるとする。a は配列の要素を受け取る仮引数である。ここでのブロックは do〜end に相当する。ブロック引数はメソッド定義では通常以下のように使用されている。この b がブロックへの仮引数になる。

```
1 def each(...)
2   ...
3   yield b
4 end
```

これは以下の定義と等しい。proc は Proc クラスによってクロージャにされたブロック引数である。yield とは暗黙的に受け取った Proc オブジェクトを、引数と共に呼び出す、というシンタックスシュガーである。

```
1 def each(..., &proc)
2   ...
3   proc.call(a)
```

```
4 end
```

また並列分散プログラミングにおいては、複数のプロセスが協調して動作することが求められる。そのためにはメッセージパッシングや共有メモリなどの通信手段が必要になるが、dRuby で実現するにはほぼ全ての部分を明示的に書く必要があり、プログラミングの負担は大きなものとなる。Ruby で並列分散処理をうまく扱うためには、並列分散処理を抽象化する必要がある。

## 2.3 関連研究

### 2.3.1 Places

既存の逐次プログラミング言語に並列処理の機能を設計・実装した研究として、Kevin Tew らの Places[2] がある。通常の Racket[7] はマルチスレッドをサポートしているが、仮想マシン上で実装されたマルチスレッドである。このスレッドの仕組みはグリーンスレッドと呼ばれるが、図 2.1に示すように OS レベルではシングルスレッドであるため、マルチスレッドによるマルチコアの恩恵は受けることができない。

Places は Racket 仮想マシンに OS でスケジューリングされたスレッドを導入し、独立したメモリ空間を割り当てる。これを place と呼ぶ。イメージを図 2.2に示す。各 place はそれ一つだけをみると逐次で動く Racket 仮想マシンのようなものであり、メモリを共有しないため並列処理における同期処理などが発生しない。それぞれの place が最大のパフォーマンスで動くことができる。また place はそれぞれが独立したガベージコレクタを持っており、place 間のオブジェクトの参照を厳密に制限することで他の place の影響をうけることなく独立して走ることができる。プロセス全体に作用するマスターガベージコレクタもあるが、プロセス大半のオブジェクトはどこかの place に属しており、属さないオブジェクトは共有メモリとして確保されたオブジェクトや place チャネルに限られる。

処理系を OS レベルでマルチスレッドに対応させるためには処理系の実装をスレッドセーフにする必要がある。Places の場合も Racket 処理系のグローバル変数を取り除く必

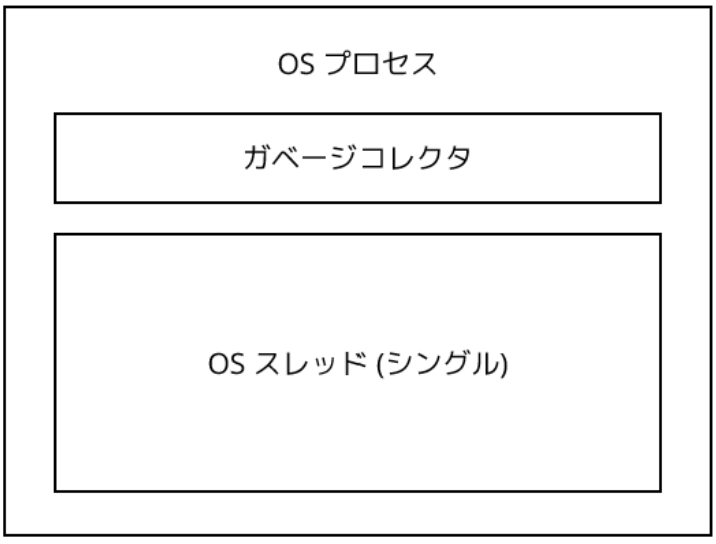


図 2.1 Racket のプロセス

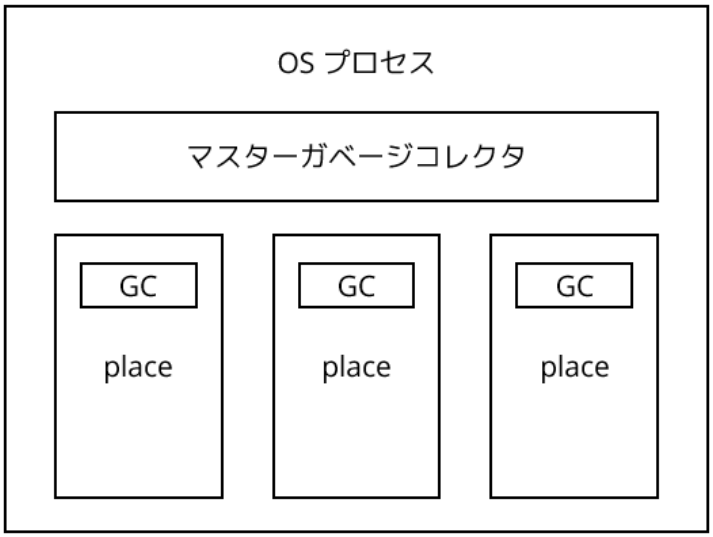


図 2.2　Racket(Places) のプロセス

要があり、その数は 719 個であったとされている [2]。Ruby の処理系である CRuby は 1.8 までグリーンスレッドであったが、1.9 からはネイティブスレッドとなった。しかし多くの部分がスレッドセーフではないため一度に動くスレッドは一つに制限されており、Places のような並列化手法は難しいとされている。

### 2.3.2　Distributed Places

同じく Kevin Tew らの研究に Places を拡張し、複数コンピュータでの分散処理機能を加えた Distributed Places[3] がある。異なるコンピュータとの通信を行うメッセージルーター place と、計算を行う place を合わせてノードという概念を作った。イメージを図 2.3に示す。異なるコンピュータ間の place の通信は SSH プロトコルが使用されている。

### 2.3.3　Teleporter

Ruby オブジェクトの効率的なプロセス間転送・共有機構の設計と実装が提案されている [4]。マルチコアプロセッサによる並列プログラミングを実現するため、複数の Ruby プロセス間でオブジェクトの転送と共有を行うための機能を提供する。dRuby と異なり共有メモリを使用しており、転送時のマーシャリングが軽量、または必要ないため、オーバーヘッドの削減が期待できる。Ruby 処理系には手を加えずライブラリにて実装されている。

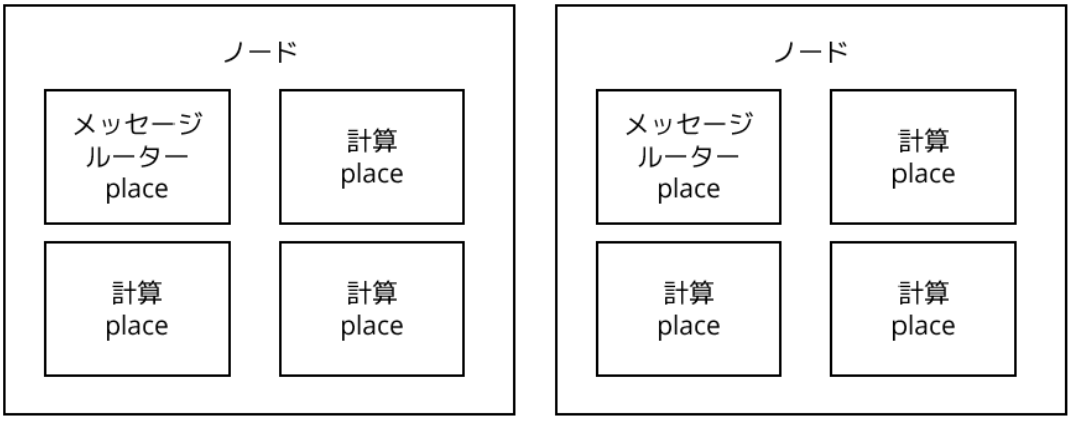


図 2.3 Distributed Place

# 第 3 章

# 言語の設計

この章では本研究で設計・実装した並列分散プログラミング環境の言語設計について説明する。

## 3.1 基本方針

本研究ではベースとなる言語を Ruby とし、Ruby に対して並列分散機能を追加した。この機能を実現するライブラリを本研究では Ixia という名前で呼んでいる。このように既存のプログラミング言語に並列分散機能を追加する先行研究の一つに Kevin Tew らの Places がある。Places は Racket に並列処理のための拡張を追加した研究である。同じ Kevin Tew らの研究で複数ノードでの分散処理に対応した Distributed Places がある。本研究の並列分散プログラミング環境の設計の基本方針は Ruby の文法や作法を尊重しつつ、Places のセマンティクスを再現するというものである。この章では先行研究である Places の設計と比較しながら、Ixia の言語設計について説明する。

## 3.2 place

Places の基本となる概念として place がある。2.3.1節 (p.8) で説明しているが、place は並列処理を抽象化したものである。実行されるコードはそれぞれの place で個別に読み出され実行される。それぞれの place は独立したデータ空間とガベージコレクタを持っている。place 間のデータのやりとりはメッセージパッシングと共有メモリを利用して行われる。このうち、メッセージパッシングはチャネルというものを通して行われる。チャネルとはファイルディスクリプタに似た概念である。実際の記述例として、Places の論文で紹介されているフィボナッチ数を並列に求める例をソースコード 3.1に引用する。

ソースコード 3.1　Places でフィボナッチ数を並列に求める例

```
1 (define (fib n) ....)
2
3 (define (start-fib n)
4   (define p
5     (place ch
6       (define n (place-channel-get ch))
7       (place-channel-put ch (fib n))))
8   (place-channel-put p n)
9   p)
10
11 (define p1 (start-fib n1))
12 (define p2 (start-fib n2))
13 (values (place-channel-get p1)
14         (place-channel-get p2))
```

start-fib が place を作成する手続きである。フィボナッチ数を求める place を作成し、変数 p に束縛する。この p が place とのメッセージパッシングに使用するディスクリプタになる。一方作成された place からのメッセージパッシングに使用するディスクリプタはソースコード 3.1では ch に束縛されている。start-fib の中で作られた place から fib を呼び出していることも注意しておきたい。place ごとに独立したデータ空間を持っていると先ほど述べたが、グローバルな関数については利用できるようになっている。

次にソースコード 3.1を Ixia で記述した例をソースコード 3.2に示す。

ソースコード 3.2　本研究の言語でフィボナッチ数を並列に求める例

```
1 def fib(n)
2   ...
3 end
4
5 def start_fib(n)
6   p = Place.new(n) do |n|
7     $ref.put fib(n)
8   end
9 end
10
11 p1 = start_fib(n1)
12 p2 = start_fib(n2)
13 p [p1.get, p2.get]
```

同じく start-fib が place を作成する手続きである。Places ではソースコード 3.1の ch のように内部からのディスクリプタを束縛する変数を指定できたが、Ruby の文法上再現ができないため、特殊変数$ref に作成された place からのメッセージパッシングに使用するディスクリプタが入っている。

Place クラスの設計は Thread クラスを参考にしており、Thread クラスと同じように引数を渡すことができる。Ruby ユーザーが Place クラスを扱う上で Thread クラスと大きく異なる所はスコープと標準入出力である。Thread のスコープの扱いは Ruby のブロックのそれに準ずる。ブロック内で定義された変数はブロック内からのみ参照できるが、ブロック内からブロック外で定義された変数を参照し、代入を行うことができる。Place クラスではスコープが独立しており、ブロック外の変数を参照することができない。ただしクラスやトップレベルメソッドについては Places と同様に、どの place でも定義されている。

本研究の実装に使用している dRuby ではメッセージパッシングのやりとりは値渡しで行われる。

place のセマンティクスは Ixia でもほぼ同じであるが、並列分散プログラミングをする上で意識しなければいけない実装上の違いの一つが Places がマルチスレッドであることに対して、Ixia がマルチプロセスであることである。具体的な問題点についてはで説明する。

## 3.3 共有メモリ

主にパフォーマンス的な観点から、place 間の通信にはメッセージパッシングを用いるべきである。しかしアルゴリズムによっては共有メモリを用いることによって簡潔に表現できるものが存在する。Places でもアトミック性の確保された共有メモリが用意されており、マンデルブロ集合を求めるプログラムが例として紹介されている。Places での共有メモリで使用できるデータはプリミティブ型のベクタに限られる。またベクタ長の指定 (size) が必須となっている。整数型と浮動小数点型の共有メモリを作成する例をソースコード 3.3に示す。x はベクタの要素の初期値であり省略可能な引数である。

ソースコード 3.3 Places での共有メモリ

```
1 (make-shared-fxvector size [x])
2 (make-shared-flvector size [x])
```

Ixia でも共有メモリが使用できる。共有メモリを作成する例をソースコード 3.4に示す。$shared は Hash クラスであり、この例では x という名前の共有変数を作る操作に等しい。特に Places のような制限はなく、普通の変数と同じように扱える。ただし利用については十分考慮するべきである。

ソースコード 3.4　Ixia での共有メモリ

```
1 $shared['x']
```

## 3.4　CG

CG は Communication Gruop の略で、複数の place をまとめて扱うことにより、高度な並列処理をより抽象化して表現することを可能とする概念である。

Ixia の CG クラスは Array クラスを元にしていくつかの機能が追加されている。ソースコード 3.5に追加されたメソッドの一部を抜粋する。CG#push は CG クラスに place を追加する操作である。4-5 行目で place に追加された CG の情報を結びつけている。これにより place から place の属する CG への参照を得ることができるようになる。

CG#sync_each はバリア同期を取ってから値を返す並列イテレーターである。これにより CG に属する全ての place からのメッセージを受け取る処理などが簡単に実現できる。

ソースコード 3.5　CG(抜粋)

```
1  class CG
2    ...
3    def push(place)
4      @places << place.set_cg(self)
5      $master.add_place_cg(client_id:$client_id, cg_id:@id, place_id:place.id)
6    end
7
8    def sync_each()
9      threads = []
10     t = Thread.new do
11       @places.each do |p|
12         yield p
13       end
14     end
15     threads << t
16     threads.each {|t| t.join}
17   end
18   ...
```

### 3.4.1 Fork-join

Places で Fork-join を実装し、並列に FFT を求めるコードをソースコード 3.6に引用する。このプログラムのイメージを図で表したものを図 3.1に示す。例として 0 から 900 までの配列のそれぞれの数に対して関数 compute-FFT-x、compute-FFT-y、compute-FFT-z を順に適用したいとする。それぞれの数に対して同じ関数は同時に適用しても構わない。ただし、すべての数に対して処理が終わるまで次の関数は実行できない。

ソースコード 3.6　Fork-join による FFT(Places)

```
1 (fork-join (processor-count) cg ([N n])
2   (CGfor cg ([i (in-range N)])
3     (compute-FFT-x i))
4   (CGBarrier cg)
5   (CGfor cg ([i (in-range N)])
6     (compute-FFT-y i))
7   (CGBarrier cg)
8   (CGfor cg ([i (in-range N)])
9     (compute-FFT-z i))))
```

ソースコード 3.6と同じ処理をするプログラムを Ixia で実装したものがソースコード 3.7である。この cg_for は引数にブロックを取るため、複数の処理を 1 つの cg_for に書くことが出来る。そのため明示的にバリアを行わず、cg_for のブロックの終わりに暗黙的

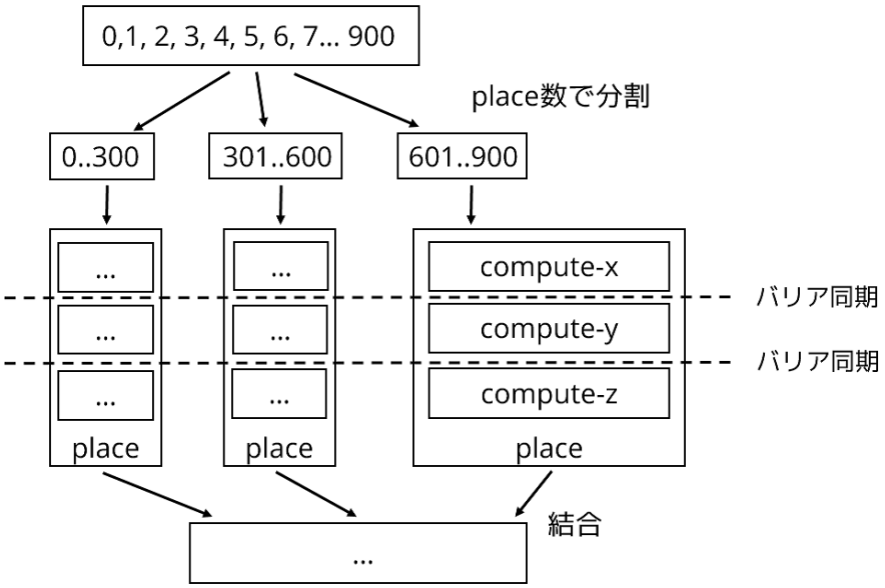


図 3.1　Fork-join

に cg_barrier を呼び出すようになっている。バリアを行いたくない場合は同じ cg_for のブロックに続けて処理を書けばよい。

ソースコード 3.7 Fork-join による FFT(Ixia)

```
1 fork_join((0..n), processor_count) do |cg|
2   cg_for(cg) {|i| compute_FFT_x(i) }
3   cg_for(cg) {|i| compute_FFT_y(i) }
4   cg_for(cg) {|i| compute_FFT_z(i) }
5 end
```

Places の Fork-join の実装プログラムをソースコード 3.8に示す。マクロで実装されている。2 行目の fork-join は NP 個の place を持つ cg を作成する。各 place は 7 行目でデータを受け取っている。ここで受け取っているデータは 14-16 行目で送信されている。8 行目では 7 行目で受け取ったデータをパターンマッチで分解している。9 行目の make-CG は CG のコンストラクタである。10 行目で place が受け取ったリストに対して関数を適用し、11 行目で結果を返している。

また Ixia の Fork-join の実装プログラムをソースコード 3.9に示す。fork_join メソッドは初期値となる配列 (cg_args) と place の作成数 (pc)、ブロック引数を受け取る。2 行目で CG のインスタンスを作成する。4 行目から place を作成する手続きである。各 place はまず 6 行目で cg を受け取る。ここで受け取るデータは 22 行目で送信されている。その後各 place は文字列データで表現された Proc オブジェクトを受け取り、配列に適用している。Ixia のでは Proc クラスのインスタンスをソースコードの文字列に変換する Proc.#to_source と、それを元の Proc オブジェクトに復元する ProcSource#.to_proc が用意されている。Proc クラスがマーシャリングできないのはクロージャが文字列では表現できない環境データを含むのが原因であり、本研究では ProcSource クラスを実装し相互変換を可能とすることでソースコードとして手続きを保持するだけの機能を追加した。ただし ProcSource#.to_proc は単に eval するだけの簡易な実装であり、例外の扱いが難しいなどの欠点がある。

place は Proc が送られてくるたびに配列に適用する。25-27 行目でバリア同期が取られている。place が__end という Symbol を受け取ると place は値を返して終了する。31 行目で送信されている。

Racket の S 式とマクロは非常に強力であり、Ruby ですべてを再現するのは現時点では難しいだろう。ブロックで処理内容を渡すのではなく、メソッド名を渡して send メソッドから呼び出すことでこの問題を回避できるが、Ruby においてはメタプログラミング以外ではあまり見ない使い方である。バリア同期が必要なければ簡潔に書くことはできる。

ソースコード 3.8 Fork-join(Places)

```
(define-syntax-rule
  (fork-join NP cg ([params args] ...) body ...)
    (define ps
      (for/list ([i (in-range n)])
        (place ch
          (define (do-work cg params ...) body ...)
          (match (place-channel-get ch)
            [(list-rest id np ps rargs)
             (define cg (make-CG id np (cons ch ps)))
             (define r (apply do-work cg rargs))
             (place-channel-put ch r)])))))

    (for ([i (in-range NP)] [ch ps])
      (place-channel-put
       ch
       (list i NP ps args ...)))

    (for/vector ([i (in-range NP)] [ch ps])
      (place-channel-get ch)))
```

ソースコード 3.9 Fork-join(Ixia)

```
def fork_join(cg_args, pc, &block)
  cg = CG.new
  cg.args = cg_args
  pc.times do
    p = Place.new do
      cg = $ref.get
      loop do
        proc_s, args = *($ref.get)
        if proc_s == :__end
          break
        end
        proc = proc_s.to_proc
        results = args.map do |argv|
          proc.call(cg, argv)
        end
        $ref.put(results)
      end
    end
    cg.add_place(p)
  end
  cg.each do |p|
    p.put(cg)
  end
  yield cg
  (cg.wait_count-1).times do
```

```
26      cg_barrier(cg)
27    end
28    results = cg_barrier(cg)
29    cg.each do |p|
30      p.put(:__end)
31    end
32    return results
33  end
```

### 3.4.2 CGPipeline

パイプライン型の並列処理は特に行列計算などにおいてよく使われる形である。Placesで並列に行列計算を行うコードをソースコード 3.10に引用する。このプログラムのイメージを図で表したものを図 3.2に示す。異なる行に計算上の依存関係がないため、行の計算がすべて終わる前に処理が終わった place は次の行の計算にとりかかることができる。各 place を並べ、place は 1 つ前の place の計算結果を受け取り、計算結果を次の place に送る。1 番目の place は init_value の値を参照し、最後の place の計算結果が求める解となる。

ソースコード 3.10 pipeline(Places)

```
1  (define v (flvector 0.0 1.0 2.0 3.0 4.0
2                      0.1 1.1 2.1 3.1 4.1
3                      0.2 1.2 2.2 3.2 4.2
4                      0.3 1.3 2.3 3.3 4.3
5                      0.4 1.4 2.4 3.4 4.4))
6
7  (fork-join 5 cg ()
8    (for ([i (in-range 5)])
9      (CGpipeline cg prev-value 0.0
10       (define idx (+ (* i 5) (CG-id cg)))
11       (define (fl-sqr v) (fl* v v))
12       (fl-sqr (fl+ (fl-vector-ref v idx)
13                    prev-value)))))
```

ソースコード 3.10と同じ処理をするプログラムを Ixia で実装したものがソースコード 3.11である。

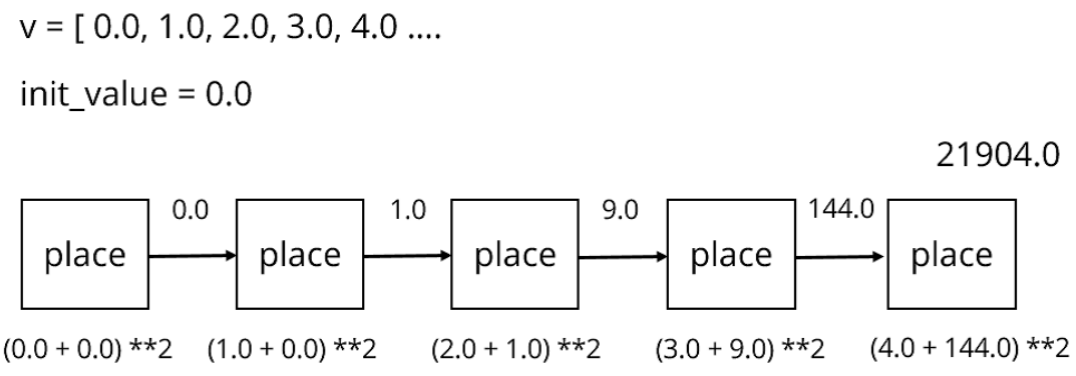


図 3.2 Pipeline

ソースコード 3.11 pipeline(Ixia)

```
1 v =  [0.0, 1.0, 2.0, 3.0, 4.0,
2       0.1, 1.1, 2.1, 3.1, 4,1,
3       0.2, 1.2, 2.2, 3.2, 4.2,
4       0.3, 1.3, 2.3, 3.3, 4.3,
5       0.4, 1.4, 2.4, 3.4, 4.4]
6
7 fork_join(v, 5) do |cg|
8   5.times do |i|
9   cg_pipeline(cg, 0.0) do |prev_value|
10    idx = i*5 + cg.id
11    (cg.args[idx] + prev_value) ** 2
12  end
13 end
```

Places の CGpipeline の実装プログラムをソースコード 3.12に示す。CGPipeline というマクロを定義し、引数には cd、前の place からの値 (prev-value)、初期値 (init-value)、処理内容 (body) を受け取る。4 行目で cg をマッチングで分解し値を得る。6 行目は次の place に値を送る send-value を定義している。7 行目は前の place から値を受け取る prev-value を定義している。もし自分が先頭の place(id が 0) の場合は初期値を使用する。12 行目にて自分が最後尾の place でなければ次の place に値を送信されている。

また Ixia の CGpipeline の実装プログラムをソースコード 3.13に示す。こちらの実装内容は Places のものとほぼ同じである。3 行目の sync_each_with_index は CG に実装されているイテレータであり、この例では p に place を、i にインデックス値 (何回目のループであるか) を代入しながらブロック内を実行する。fork_join で渡される配列は cg.args から参照できるようになっている。v は共有メモリを用いてもよい。

ソースコード 3.12 CGpipeline(Ixia)

```
1 (define-syntax-rule
2   (CGpipeline cg prev-value init-value body ...)
3   (match cg
4     [(CG id np pls)
5      (define (send-value v)
6        (place-channel-put (list-ref pls (add1 id)) v))
7      (define prev-value
8        (if (= id 0)
9          init-value
10         (place-channel-get (car pls))))
11     (define result (begin body ...))
12     (unless (= id (sub1 np)) (send-value result))
13     result]))
```

ソースコード 3.13　CGpipeline(Ixia)

```
1 def cg_pipeline(cg, init_value, &block)
2   results = []
3   cg.sync_each_with_index do |p, i|
4     prev_value = cg.first?(p) ? init_value : cg.prev(p).get
5     p.put([block.to_source, prev_value])
6     results[i] = p.get
7   end
8   return results[-1]
9 end
```

# 第 4 章

# 対話型環境の設計

本研究で扱う対話型環境とは、コンピュータと対話をするようにデバッグを進めていく手法である。多くの言語に対話型環境が存在し、Ruby にも IRB や Pry といったものがあり良く使われている。しかしこれらの環境は逐次プログラミングのためのものであり、並列分散プログラミングで使用するためにはいくつかの問題が発生する。

## 4.1 前提知識

### 4.1.1 ステップ実行

ステップ実行とは、プログラムを 1 ステートメントごとに実行することである。ステップ実行と対話型環境を組み合わせることにより、プログラムの任意の場所で途中のデータや条件判定などを確認することが可能になる。ステップ実行には一般的に以下のようなモードがある。

- ステップイン
  現在のステートメントを実行してから次のステートメントで停止する。現在のステートメントがブロックである場合、デバッガはそれらの中のステートメントに対してステップインを実行する。
- ステップオーバー
  現在のステートメントを実行してから次のステートメントで停止する。ブロック呼び出しの場合はそのブロックを実行して次のステートメントで停止する。ブロックの中には入らない。
- ステップアウト
  現在のブロックの外にステップする。ブロックがネストされている場合 1 つ上のレベルにステップする。トップレベルの場合は次のブレークポイント、または終了までプログラムを実行する。
- 続行
  次のブレークポイント、または終了までプログラムを実行する。

### 4.1.2 ブレークポイント

ステップ実行はデバッグに非常な有効な手段であるが、毎回プログラムの初めからステップ実行するのは実用的ではない。そのため実行中のプログラムを意図的に一時停止させる箇所を予め決めておき、そこからステップ実行を行うのが一般的である。この箇所を指定する方法をブレークポイントと呼ぶ。Ixia ではプログラム中に breakpoint と記述することで、記述した箇所で place は一時停止し、対話モードに入る。

## 4.2 既存の対話型環境の問題点

逐次プログラミングで使用することを想定した対話型環境を並列分散プログラミングに適用した時に起こりうる問題について説明する。

### 4.2.1 対話対象

逐次プログラムでは同時に実行されるプログラムは 1 つであるため、対話する対象は明確である。並列分散プログラムにおいては複数のプログラムが同時に動いているため、どのプログラムと対話しているのかを意識しなければならない。

### 4.2.2 ブレークポイントの挿入

並列分散プログラムを対話的にデバッグする場面では、関連するプロセスすべてを一時停止する必要があることがある。特にプロセス間でデータのやりとりが行われている場合などは、バグの原因が違うプロセスに潜んでいる事も多い。実際にプロセスを一時停止する方法であるが、関連するプロセスすべてが分岐もせず同じコードを実行する場合には容易である。例としてモンテカルロ法で円周率を求めるプログラムの一部をソースコード 4.1に抜粋する。

ソースコード 4.1　モンテカルロ法 (抜粋)

```
1  def square(n)
2    return n*n
3  end
4
5  cg = CG.new
6
7  N.times do
8    p = Place.new(P) do |n|
9      count = 0
10     n.times do
```

```
11        if Math::sqrt(square(rand) + square(rand)) < 1
12          count += 1
13        end
14      end
15      $ref.put count.to_f / n
16    end
17    cg.push(p)
18  end
19
20  puts cg.map{|p|p.get}.sum * 4 / N
```

ソースコード 4.1の場合、並列計算のために作られた place は 9-15 行目を必ず実行する。このどこかに breakpoint を挿入すればすべての計算 place をほぼ同じタイミングで停止することができる。

次にブレークポイントの挿入が問題となるプログラムについて説明する。例としてバイトニックソートを行うプログラムの一部をソースコード 6.1に示す。

ソースコード 4.2　バイトニックソート (抜粋)

```
1  def bitonic_sort(up, st, array)
2    if array.size <= 1
3      return array
4    else
5      if st > 0
6        p1 = Place.new(true, st, array[0...array.size/2]) do |up, st, array|
7          $ref.put bitonic_sort(up, 0, array)
8        end
9        p2 = Place.new(false, st, array[array.size/2..-1]) do |up, st, array|
10         $ref.put bitonic_sort(up, 0, array)
11       end
12       first = p1.get
13       second = p2.get
14     else
15       first = bitonic_sort(true, st-1, array[0...array.size/2])
16       second = bitonic_sort(false, st-1, array[array.size/2..-1])
17     end
18     return bitonic_merge(up, first + second)
19   end
20 end
```

ソースコード 6.1では、配列を再帰的に分割しながらソートを行うが、st 回目の再帰までは place を作成することにより $2^{st}$ プロセスで並列に計算を行っている。このプログラムでは引数によって実行されるコードが異なる。2 行目に breakpoint を挿入した場合は再帰的に実行されることなくブレークポイントに達してしまう。

## 4.3 本研究で提案するデバッグ環境

### 4.3.1 アクティブな place

当研究ではアクティブな place というものを定義している。実装については IOSimulator(5.2.5節 (p.36)) と PStream(5.3.1節 (p.37)) で説明するのでここでは触れない。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。place1 がアクティブな状態のイメージを図 4.1に、place2 がアクティブである状態のイメージを図 4.2に示す。

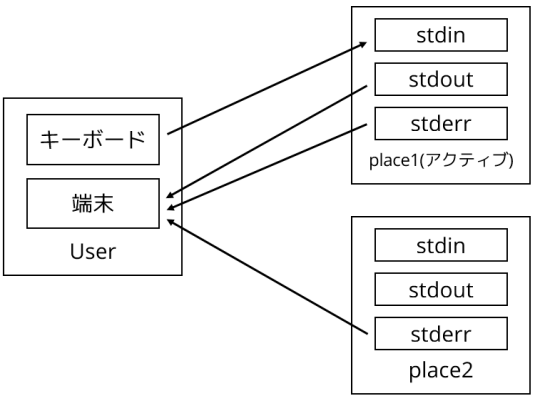


図 4.1　place1 がアクティブである状態

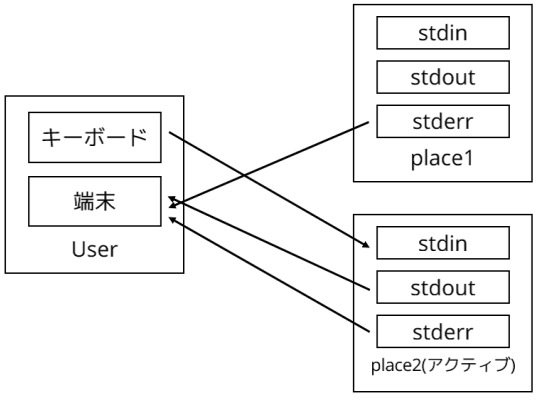


図 4.2　place2 がアクティブである状態

### 4.3.2 CG#the_world

4.2.2節 (p.23) の問題点を解消するために、CG に結びついているすべての place を対話状態に移行する。ブレークポイントにより place が対話環境に入るかどうかはフラグとポーリングで管理されており、このメソッドは CG に属するすべての place のフラグを操作する。つまり CG 中のどれか 1 つの place がブレークポイントで停止すると CG 中の他のすべての place も停止する。フラグが立っている場合 Interpreter は対話環境用のメソッドを呼び出す。トップレベルに CG を問わずに同クライアントの全ての place に作用する the_world が定義されている。

### 4.3.3 CG#time_starts

CG に結びついている place のブレークポイントフラグを全て無効にするメソッドである。the_world で停止したすべてのプロセスを実行状態に移行する際に使用する。トップレベルに CG を問わずに同クライアントの全ての place に作用する time_starts が定義されている。

## 4.4 デバッグ実例

実際に当研究の対話環境を利用してデバッグを行う例を示す。

### 4.4.1 例外が起きるケース

ソースコード 4.3は 9 行目で、定義されていない c を参照している例である。この例の場合のデバッグの流れを図 4.3に示す。他のプロセスで起きた place の例外がキャッチされていない場合、ユーザーの端末まで届けられる。この例外情報はどの place の、どの行の、何のコードを実行したか、の情報を含んでいる。

図 4.3の中で丸付き数字で示すのはユーザーの入力である。それぞれの入力についての説明を以下に示す。

1. p メソッドで p に束縛されている place の情報を確認する。例外情報の place_id と一致することが分かる。
2. cp メソッドでアクティブな place を p に変更する。例外を起こした place は自動でブレークポイント状態になり、例外を起こす 1 ステートメント前で停止している。

3. プログラムを修正する。edit コマンドで任意のテキストエディタで編集することもできる。
4. run コマンドで place をステップアウトする。ステップアウトした際、アクティブな place は$ref に移る。
5. 元の place に戻ったのでこちらもステップアウトする。無事に計算が終了した。

ソースコード 4.3　例外が起きるプログラム

```
1 require_relative "../ixia"
2 include Ixia
3
4 p = Place.new do
5   a = 100
6   b = 200
7   $ref.put a + c
8 end
9
10 breakpoint
11 puts p.get
```

### 4.4.2　the_world を使用するケース

例外は起こらないが、想定通りにプログラムが動かない場合のデバッグ方法について説明する。サンプルコードをソースコード 4.4に示す。このプログラムでは place の引数によって実行するコードが異なる。しかし 12 行目の the_world にてどこを通っていてもすべての place が対話状態になる。

このプログラム自体にはあまり意味が無いが、例えば例外をキャッチした際にすべての place を止める、などの使い道が考えられる。ただし本研究ではブロックに対するステップインが実装できていないため、現時点では使いにくい場合もある。この点についてはあとの章で考察する。

ソースコード 4.4それぞれの入力についての説明を以下に示す。

1. cg に入っている place の一覧を確認する。
2. cg に入っている 1 番目の place をアクティブにする。　このプログラムは 15 行目から 17 行目までを実行するので単体ではブレークポイントにかかっていない。
3. the_world により place が対話状態になっていることが確認できた。

ソースコード 4.4　the_world を使用するプログラム

```
1 require_relative "../ixia"
```

```
====================================
0000 : require "../ixia"
0001 : include Ixia
0002 : p = Place.new do
         a = 100
         b = 200
         $ref.put a + c
       end
0003 : breakpoint
0004 : puts p.get
place(#0)>
#<Exception place_id=1, line=2, exec="$ref.put a + c", mess="undefined local variable or method
 `c' for main:Object">

place(#0)> p p                                                          ①
#<Place:0x6ac698 @client_id=2, @id=1, @node=nil, @cg=nil, @args=[],  @ref_id=0>
place(#0)> cp p
====================================
0000 : a = 100                                                          ②
0001 : b = 200
0002 : $ref.put a + c
place(#1)> c = 200                                                      ③
place(#1)> run                                                          ④
place(#0)> run                                                          ⑤
300
```

図 4.3　例外が起きた場合のイメージ

```
2  include Ixia
3
4  cg = CG.new
5  10.times do |i|
6    p = Place.new(i) do |i|
7      if i == 0
8        sleep 3
9        the_world
10       puts i
11     elsif i == 1
12       sleep 3
13       puts i
14     elsif i == 2
15       sleep 3
16       puts i
17     end
18     puts i*i
19   end
20   cg.push(p)
21 end
22 breakpoint
23 cg.each{|a|a.join}
```

```
==================================
0000 : require "../ixia"
0001 : include Ixia
0002 : cg = CG.new
0003 : 10.times do |i|
         p = Place.new(i) do |i|
           if i == 0
             sleep 3
             the_world
             puts i
           elsif i == 1
             sleep 3
             puts i
           elsif i == 2
             sleep 3
             puts i
           end
           puts i * i
         end
         cg.push(p)
       end
0004 : breakpoint
0005 : cg.each{|a| a.join}
place(#0)> p cg.list
[#<Place:0x24b8758 @client_id=0, @id=1, @node=nil, @cg=#<CG:0x24bbbc0 @id=0, @places=[...],
 @wait_count=0, @args=nil>, @args=[[:i, 0]],  @ref_id=0>, #<Place:0x25b44a0 @client_id=0, @id=2,
 @node=nil, @cg=#<CG:0x24bbbc0 @id=0, @places=[...], @wait_count=0, @args=nil>, @args=[[:i, 1]],
  @ref_id=0>, #<Place:0x26432e8 @client_id=0, @id=3, @node=nil, @cg=#<CG:0x24bbbc0 @id=0,
place(#0)> cp cg.place(1)
1
==================================
0000 : i = $master.places[0][2]["args"][:i]
0001 : if i == 0
         sleep 3
         the_world
         puts i
       elsif i == 1
         sleep 3
         puts i
       elsif i == 2
         sleep 3
         puts i
       end
0002 : puts i * i
place(#2)> p i
1
```

① ② ③

図 4.4　the_world を使用するイメージ

# 第 5 章

# 処理系の実装

## 5.1 システムの概要

Ixia を使用するためには予め Master(5.1.1節 (p.31)) と Node(5.1.2節 (p.32)) を起動しておく必要がある。Master はシステム全体で 1 つ、Node は処理を割り当てたい物理ノードごとに 1 ずつ起動する。Node を起動する際にオプションで Worker(5.1.3節 (p.32)) を起動する数を指定する。この Worker が実際の処理を行うプロセスで、通常、論理 CPU の数だけ起動するのがよいと思われる。図 5.1にシステムの概要図を示す。実戦の矢印は dRuby によって接続されていることを示し、点線の矢印は Node によって Worker が管理され起動されることを表している。

### 5.1.1 Master

Master は MasterObjcect クラスのインスタンスを持ち、dRuby 経由で共有するためのホストである。place の管理やデータのやりとりはこの Master を通じて行われる。Ixia の基幹となるプログラムである。

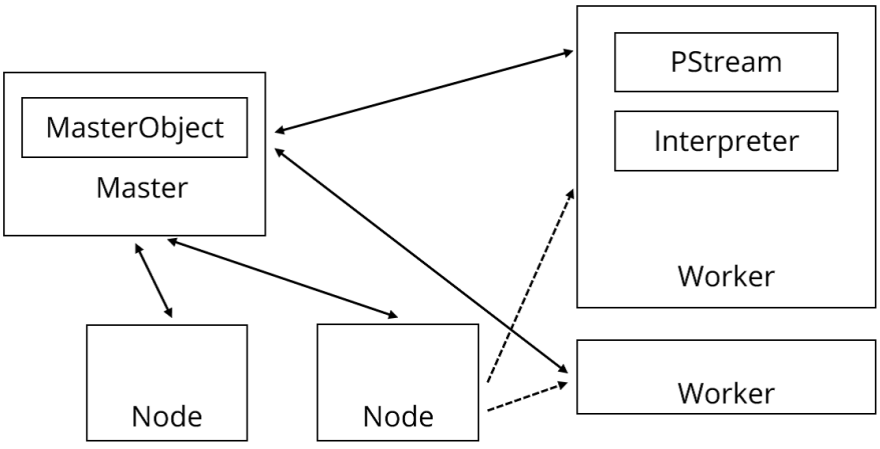


図 5.1 システムの概要図

### 5.1.2 Node

Worker を管理するプログラムである。常に指定された数の Worker が起動しているように監視を行う。また place の起動時に Node を指定することで place を処理する物理ノードを指定することが出来る。

### 5.1.3 Worker

place の実行を行う。5.3節 (p.37) にて説明する。

## 5.2 place の作成

実際のソースコードから place を生成する手順と、その実装方法について説明する。Ixia をロードしたプロセスは以下の様なふるまいをする。

1. クライアントを作成する。
2. ソースコードから ParserModule(5.2.3節 (p.33)) を実行してクラス・メソッド定義を取り出す。ここで取り出した定義はクライアントに紐付いて登録され、各 place が実行される前に読み出され定義される。
3. ソースコードから SRProgram(5.2.4節 (p.35)) を用いてースコードをステートメント単位に分割し、そのデータを持つ Place オブジェクトを作成する。この時静的解析によりその place が the_world を実行する可能性があるかどうかのチェックを行っている。この place の中で Place.new が行われていれば再帰的に新たな place が作成される。作成された place は Master に送られ実行されるのを待つ。
4. IOSimulator(5.2.5節 (p.36)) オブジェクトを生成し、実行する。

トップレベル以外で作成される place は place オブジェクトの作成のみが行われる。トップレベル以外の place が作成される際、引数があれば渡されたオブジェクトを一緒に Master に送る。

### 5.2.1 クライアント

クライアントとは、Ixia をロードしたプログラムが実行されるたびに作成される、関連する place やオブジェクトをまとめる概念である。IxiaMaster サーバーは同時に複数のクライアントを実行できる。この時クライアントが名前空間の役割を果たし、別々のクライアントの place が影響しあうことなく動作することができる。

### 5.2.2 Place(クラス)

ここでは実装上の Place クラスについて説明する。セマンティクスとしての place は 2.3.1節 (p.8) を、place の使い方については 3.2節 (p.11) を参照して欲しい。Place オブジェクトが持つデータは p メソッドなどで見ることができる。実際に place オブジェクトを確認する例をソースコード 5.1に、表示されるオブジェクトの概要をソースコード 5.2に示す。

作られた place はで説明するように実行されるが、図 5.2に示すように、place の外側からみた Place オブジェクトと、その place の内側からみた place オブジェクトは、同じ place 識別子を持つ同じ Place クラスであるが別のオブジェクトである。これにより place の寿命を考慮しないプログラミング設計ができる。

ソースコード 5.1 place オブジェクト

```
1 p = Place.new do
2   a = 100
3   b = 200
4   c = 300
5   breakpoint
6   $ref.put a * b + c
7 end
```

ソースコード 5.2 place オブジェクトの概要

```
1 <Place:0x2956b30
2   @client\_id=0,
3   @id=1,
4   @node=nil,
5   @cg=<DRb::DRbObject:0x20ac8d0 @uri="druby://illya:54144", @ref=18917532>,
6   @args=[],
7   @source=<SRProgram:0x2956a58 ...>
```

### 5.2.3 ParserModule

パーサーを扱うメソッドをまとめた本研究で実装したクラスモジュールである。内部では Ripper ライブラリを用いている。Ripper は Ruby 処理系のパーサーを拡張しモジュールとしたものであるため、文法の互換性が極めて正確であるとされている。place の生成の際にも利用するためモジュールとなっているが、ここではクラス・メソッド定義

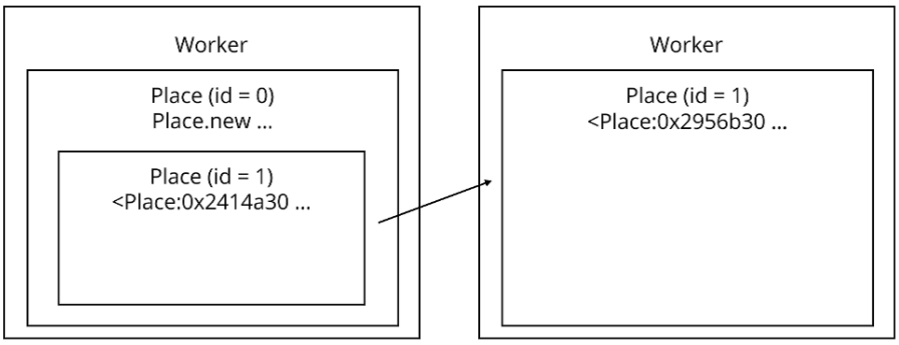


図 5.2　Place オブジェクト

を抜き出す手法について説明する。実際のメソッド定義を含むプログラムを Ripper で字句解析した際に得られる配列の一部をソースコード 5.3に抜粋する。

ソースコード 5.3　Ripper で字句解析した例 (抜粋)

```
1  [[4, 0], :on_kw, "def"],
2  [[4, 3], :on_sp, " "],
3  [[4, 4], :on_ident, "bitonic_sort"],
4  [[4, 16], :on_lparen, "("],
5  [[4, 17], :on_ident, "up"],
6  [[4, 19], :on_comma, ","],
7  [[4, 20], :on_sp, " "],
8  [[4, 21], :on_ident, "st"],
9  [[4, 23], :on_comma, ","],
10 [[4, 24], :on_sp, " "],
11 [[4, 25], :on_ident, "array"],
12 [[4, 30], :on_rparen, ")"],
13 [[4, 31], :on_ignored_nl, "\n"],
14 [[5, 0], :on_sp, "  "],
15 [[5, 2], :on_kw, "if"],
16 [[5, 4], :on_sp, " "]
17 ...
```

この配列は各要素が [[行数, 行頭からの文字数], 字句の分類, 字句] という構造をしている。ここではクラス・メソッド定義を抜き出すことが目的であるので、予約語を示す on_kw に注目する。特殊なメソッドを用いて動的にメソッドを定義するような場合を除いて、Ruby のプログラムには以下の様な特徴がある。

- メソッド定義は def〜end という形をとる。
- クラス定義は class〜end という形をとる。
- 予約語である end はブロックの終わりを意味しており、ブロックは必ずネスト構造である。ブロックをまたぐようなブロックは存在しない。

- end に対応する予約語は module, class, def, if, unless, case, while, until, for, begin, do だけである。ただし Ruby における if と unless は後置で記述でき、こちらはブロックを持たないため end がない。これは行頭からの文字数で区別できる。

現在のブロックのネストの深さをカウントしながら予約語をチェックすることで、クラス・メソッド定義を抜き出すことができる。

### 5.2.4 SRProgram

Ruby プログラムをステートメント単位に分割するためのクラスである。対話型環境でステップ実行を実現するために必要となる。まず、Ruby プログラムを Ripper を用いて構文木に変換する。Ripper で構文木解析プログラムの例の一部をソースコード 5.4に抜粋する。

ソースコード 5.4 Ripper で構文木解析した例 (抜粋)

```
1  [:program,
2   [[:command,
3     [:@ident, "require_relative", [1, 0]],
4     [:args_add_block,
5      [[:string_literal,
6        [:string_content, [:@tstring_content, "../ixia", [1, 18]]]]],
7      false]],
8    [:command,
9     [:@ident, "include", [2, 0]],
10    [:args_add_block, [[:var_ref, [:@const, "Ixia", [2, 8]]]], false]],
11   [:def,
12    [:@ident, "bitonic_sort", [4, 4]],
13    [:paren,
14     [:params,
15      [[:@ident, "up", [4, 17]],
16       [:@ident, "st", [4, 21]],
17       [:@ident, "array", [4, 25]]],
18      nil,
19      nil,
20      nil,
21      nil,
22      nil,
23      nil]],
24    [:bodystmt,
25     [[:if,
26       [:binary,
27        [:call,
28         [:var_ref, [:@ident, "array", [5, 5]]],
29         :".",
30         [:@ident, "size", [5, 11]]],
```

```
31          :<=,
32          [:@int, "1", [5, 19]]],
33        [[:return,
34          [:args_add_block, [[:var_ref, [:@ident, "array", [6, 11]]]], false]]],
35        [:else,
36 ...
```

構文木にすることでステートメント単位に分割することができたため、ステートメントごとに Ruby プログラムに復元する。これに適したライブラリが見つからなかったため、規則を調べ復元するプログラムを実装した。これが SRProgram のほぼすべてを占めている。S 式に似た構造であるため再帰的に構造を見ていくことで復元することができる。復元のために実装したプログラムの一部をソースコード 5.5に抜粋する。

ソースコード 5.5　SRProgram(抜粋)

```
1 case sexp[0]
2 when :command
3   command = p_parse(sexp[1])
4   args = p_parse(sexp[2])
5   return command + " " + args
6 when :assign
7   var = p_parse(sexp[1])
8   value = p_parse(sexp[2])
9   return var + " = " + value
10 when :massign
11   vars = []
12   sexp[1].each do |s|
13     vars << p_parse(s)
14   end
15   values = p_parse(sexp[2])
16   return vars.join(", ") + " = " + values
17 ...
```

### 5.2.5　IOSimulator

IOSimlator は複数の place との対話を可能にするためのクラスである。通常、ユーザーの標準ストリームは Ixia をロードしたプログラムのプロセスと接続している状態である。Ixia をロードしたプログラムは place となり別の Interpreter の上で実行されるため、ユーザーの標準ストリームと橋渡しをする必要がある。前提として、Ruby のストリームはただのオブジェクトであり、必要となるメソッドが用意されているオブジェクトであれば自由に置き換えが可能である。それぞれの Interpreter は標準ストリームを PStream(5.3.1節 (p.37)) クラスのオブジェクトで置き換えており、バッファに保

存されている。IOSimlator はアクティブな place の PStream と標準ストリームをストリーミングすることで、place との対話を実現している。ただし標準エラー出力については Master がクライアントごとにキューを持っており、PStream から Master に送られ、IOSimulator は Master からストリーミングを行う。これは標準エラー出力の性質上、アクティブでない place のエラーについてもすぐに確認できることが適切だと考えるためである。

## 5.3 place の実行

Master にキューイングされた place は Worker によって取り出され実行される。Worker は以下の様なふるまいをする。

1. Master のキューから place オブジェクトを取り出す。
2. $ref と$self を設定する。$ref が place を作成した place への参照、$self は自分自身への参照である。
3. 標準ストリームを PStream に置き換える。
4. Interpreter(5.3.2節 (p.38)) を起動する。

プロセスの持つ環境をリセットするため、Worker は一つの place を処理するたびに終了し、Node によって再度起動される。

### 5.3.1 PStream

place の標準ストリームを置き換えるためのクラスである。IOSimulator と接続し、異なるプロセス間の入出力を実現する。標準ストリームを置き換えるオブジェクトに要求されるメソッドは Ruby の仕様で決められている。現在最低限の gets, write, input, output が実装されており、その実装は極めて単純である。PSream の実装をソースコード 5.6に示す。

ソースコード 5.6 PStream

```
1 class PStream
2   def initialize
3     @stdin = Queue.new
4     @stdout = Queue.new
5     @stderr = Queue.new
6   end
7
8   def gets(rs = "\n")
9     return @stdin.pop
```

```
10    end
11
12    def write(a)
13      @stdout.push(a)
14    end
15
16    def input(a)
17      @stdin.push(a)
18    end
19
20    def output
21      return @stdout.pop
22    end
23  end
```

### 5.3.2 Interpreter

place の保持するソースコードの実行と、対話型デバッグを実現するためのクラスである。デバッグモードで Ixia が実行されている時、Interpreter は 1 ステップずつソースコードを実行する。この時に place がブレークポイントで停止する可能性がある場合は、ブレークポイントのフラグをポーリングでチェックしながら実行する。place がブレークポイントで停止する可能性があるのはブレークポイントが実行された時は Interpreter クラス内の対話用メソッドを呼び出す。対話用デバッグコマンドも Interpreter 内で定義されている。

# 第 6 章

# 実験と考察

## 6.1 実験

本研究でバイトニックソートを実装し、12 コア 24 スレッドの計算機上でソートにかかる時間を計測した。実験に使用したソースコードをソースコード 6.1に、実行時間のグラフを図 6.1に示す。2 の st 乗の place に分割して計算を行う実装である。

ソースコード 6.1 バイトニックソート

```
1  require_relative '../ixia'
2  include Ixia
3
4  def bitonic_sort(up, st, array)
5    if array.size <= 1
6      return array
7    else
8      if st > 0
9        p1 = Place.new(true, st, array[0...array.size/2]) do |up, st, array|
10         $ref.put bitonic_sort(up, st -1, array)
11       end
12       p2 = Place.new(false, st, array[array.size/2..-1]) do |up, st, array|
13         $ref.put bitonic_sort(up, st -1, array)
14       end
15       first = p1.get
16       second = p2.get
17     else
18       first = bitonic_sort(true, st-1, array[0...array.size/2])
19       second = bitonic_sort(false, st-1, array[array.size/2..-1])
20     end
21     return bitonic_merge(up, first + second)
22   end
23 end
24
25 def bitonic_merge(up, array)
26   if array.size == 1
27     return array
28   else
29     bitonic_compare(up, array)
```

```
30      first = bitonic_merge(up, array[0...array.size/2])
31      second = bitonic_merge(up, array[array.size/2..-1])
32      return first + second
33    end
34  end
35
36  def bitonic_compare(up, array)
37    dist = array.size / 2
38    dist.times do |i|
39      if (array[i] > array[i+dist]) == up
40        array[i], array[i+dist] = array[i+dist], array[i]
41      end
42    end
43  end
```

## 6.2 考察

### 6.2.1 実装コスト

実験に使用したソースコード 6.1は、通常の逐次で実行されるバイトニックソートに手を加えたものである。並列化される前の bitonic_sort をソースコード 6.2に、並列化した

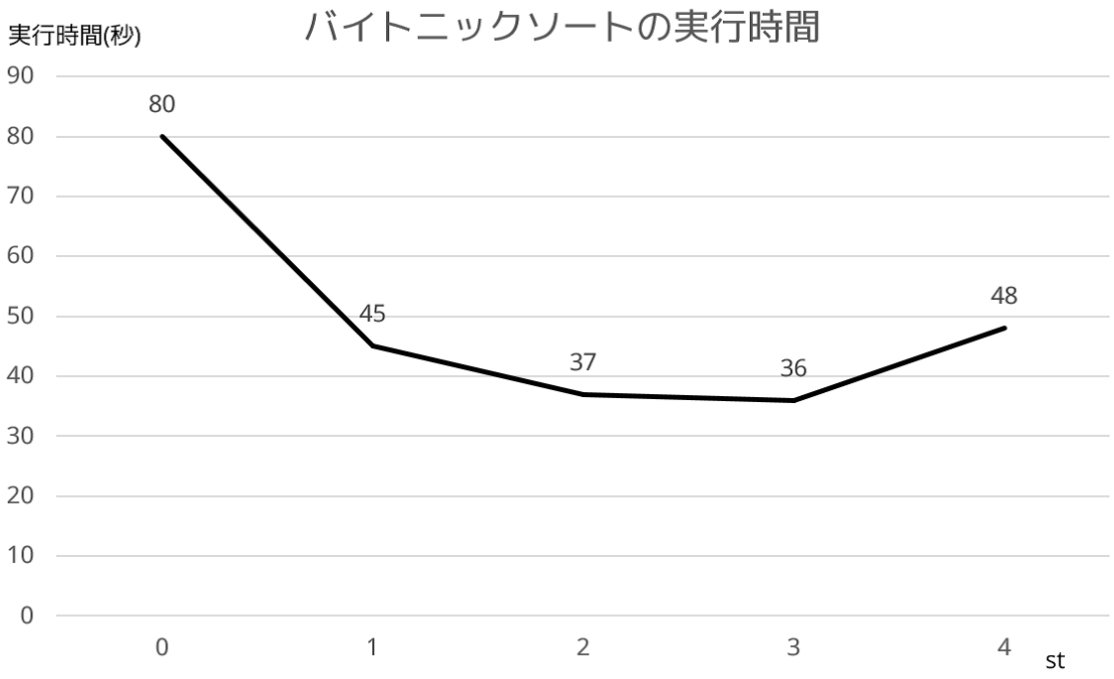


図 6.1　バイトニックソートの実行時間

後の bitonic_sort をソースコード 6.3に示す。

このバイトニックソートは再帰によって分割しつつソートを行っていく実装であるが、引数 st を新しく咥え、一定回数の再帰までは新しい place でメソッドを呼び出すことにより、並列処理を実現した。再帰による place 生成回数を制限しているのは本研究の place を作成するコストが高く、ある程度の数以上になると並列化のボトルネックの方が大きくなってしまうためである。

ソースコード 6.2　並列化前の bitonic_sort

```
1 def bitonic_sort(up, array)
2   if array.size <= 1
3     return array
4   else
5     first = bitonic_sort(true, array[0...array.size/2])
6     second = bitonic_sort(false, array[array.size/2..-1])
7     return bitonic_merge(up, first + second)
8   end
9 end
```

ソースコード 6.3　並列化後の bitonic_sort

```
1 def bitonic_sort(up, st, array)
2   if array.size <= 1
3     return array
4   else
5     if st > 0
6       p1 = Place.new(true, st, array[0...array.size/2]) do |up, st, array|
7         $ref.put bitonic_sort(up, st -1, array)
8       end
9       p2 = Place.new(false, st, array[array.size/2..-1]) do |up, st, array|
10        $ref.put bitonic_sort(up, st -1, array)
11      end
12      first = p1.get
13      second = p2.get
14    else
15      first = bitonic_sort(true, st-1, array[0...array.size/2])
16      second = bitonic_sort(false, st-1, array[array.size/2..-1])
17    end
18    return bitonic_merge(up, first + second)
19  end
20 end
```

### 6.2.2 性能評価

現時点ではよいとは言えない。対話環境の実装のため多数の Thread を使用しており、place 数が増えてくると Master の負荷が高く性能が頭打ちになってしまう。また place の生成コストは Ruby プログラムを毎回起動することと等しく、プロセス間の通信も dRuby 経由であるためコストがかかってしまう。Master の負荷についてはスレッドを多用せずイベントドリブンでの実装を行うなどの工夫が必要だと思われる。通信コストについては Teleporter[4] の導入により削減できる可能性がある。

# 第 7 章

# 今後の課題

## 7.1 ステップインの実装

### 7.1.1 Ruby におけるブロック

本研究の対話型環境は、現時点ではブロックにステップインできないという問題がある。Ruby はブロックを多用する構文であり、実用的なデバッグのためには欠かせない今後の課題である。ブロックをブロックとしてパースし、再帰的なソース構造を得ることは難しくない。問題になるのはブロックはそのままではブロック単位でしか評価できない点にある、例えば以下の様なコードは eval できない。

```
1 if a == 1
2   puts 'test'
```

このようなコードでステップインを実現するためには、if を eval せずに if の構文を再現しなければいけない。これは if に限らず、Ruby における「文」とクロージャすべてに必要なことである。Ruby における代表的な文と、それの実装方法についての考察を以下に述べる。

### 7.1.2 class

```
1 class Person
2   ...
3 end
```

クラス定義は文である。Ruby における class はインスタンス変数やクラス変数などの状態を内部で持っているので、これらの処理も再現しなければならない。これは Ruby 処理系を再度実装するのに等しいことで、不可能ではないが大変難しいと思われる。

### 7.1.3 def

```
1 def get()
2   ...
3 end
```

メソッド定義は文である。Ruby において他の言語における関数というものは存在せず、厳密にはすべてメソッドである。Ruby ではプログラムが実行されると Object クラスのオブジェクトである main インスタンスが作成され、トップレベルで定義したメソッドは main のメソッドとなる。メソッドはクラスの一部なので実装が難しいことには変わりがない。ただし内部に状態を持たない参照透過性のあるメソッドに限れば少ないコストで実装でき、それだけでも便利だと思われる。

# 第 8 章

# おわりに

Ruby にて並列分散処理を行うライブラリを設計、実装した。並列処理の抽象化については Places[2] を参考にしたが、このセマンティクスは Ruby においても有効であった。Thread に慣れている Ruby ユーザーであれば学習コストも低く、Ruby の高い生産性を並列分散プログラムにおいても十分に発揮できるであろう。

対話型環境ではアクティブな place の切り替え、the_world などを設計、実装し、並列分散プログラムを効率的にデバッグする手法について示した。

実装面では Ruby ライブラリでの実装が苦しいと感じる場面が多々あった。まず Proc クラスがマーシャリングできないために、ソースコードをパースする必要があり、かなりのオーバーヘッドが発生する。

パーサーの実装も楽ではなかった。Racket と比較して Ruby の構文はとても複雑である。またオブジェクト指向ゆえにインスタンスが状態を持つことが、ステップインの実装の大きな妨げになった。Racket は純粋関数型言語ではないものの、手続き型言語と比べて副作用の使用は限定的である。

# 謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。時間もないので多くは書けませんが、この研究室に来て本当によかったと思います。私は幸せものでした。

# 参考文献

[1] Paolo Perrotta: “メタプログラミング Ruby“, アスキー・メディアワークス, ISBN 978-4048687157.

[2] K. Tew, J. Swaine, M. Flatt, R. Findler, and P. Dinda: Places: adding message-passing parallelism to racket, *Proceedings of the 7th symposium on Dynamic languages*, pages 8596. ACM, 2011.

[3] K. Tew, J. Swaine, M. Flatt, R. Findler, and P. Dinda: Distributed Places, *14th International Symposium*, pages 14-16, TFP, 2013.

[4] 中川 博貴, 笹田 耕一: Ruby オブジェクトの効率的なプロセス間転送・共有機構の設計と実装, 情報処理学会論文誌プログラミング（*PRO*）, pages 1-16, 2012.

[5] Jr Frederick P. Brooks: 人月の神話, 丸善出版, ISBN 978-4621066089.

[6] “dRuby“, `http://docs.ruby-lang.org/ja/1.8.7/library/drb.html`, 2015 年 1 月 25 日参照.

[7] “The Racket Language“, `http://racket-lang.org/`, 2015 年 1 月 25 日参照.

[8] “The Revised6 Report on the Algorithmic Language Scheme”, `http://www.r6rs.org/`, 2015 年 1 月 25 日参照.