# 関数型言語 Haskell による<br>並列データベースの実装

# Implementation of the Parallel Database using Haskell

平成26年度 学位論文(修士)

琉球大学大学院 理工学研究科
情報工学専攻

當眞 大千

# 要 旨

Haskell は純粋関数型プログラミング言語である. モダンな型システムを持ち, 型推論と型安全により信頼性の高いプログラムを書くことが可能である [1].

本研究では, Haskell を用いて並列に読み書き可能なデータベースの実装を行う. 並列にデータへアクセスする手法として, 元となる木構造を変更することなく編集できる非破壊的木構造を用いる. 非破壊的木構造は, 破壊的代入が存在しない Haskell と相性がよい.

実装した並列データベースの読み込みと書き込みについて性能を計測し, 読み込みに関して 12 コアで実行した場合, 1 コアで実行した場合と比較して, 10.37 倍 という性能向上率が確認でき, マルチコアプロセッサの性能を引き出すことができた.

また, 実用的な用途で利用できるか示すために, Web 掲示板サービスを開発した. 既存の Java の非破壊的木構造データベースを用いた掲示板実装との比較をおこない, 読み込みで 3.25 倍, 書き込みで 3.78 倍の性能が確認できた.

# Abstract

Haskell is a purely-functional programming language. It provides a modern type system, type-safe and type inference makes it possible to write a program reliable[1]. Haskell has referential transparency that allows the programmer and the compiler to reason about program behavior.

In this study, we implement the parallel database using Haskell. It is use non-destructive tree structure. Non-destructive tree structure is not the destruction of data. Editing of data is done creating by new tree. Haskell compatible with non-destructive tree, because Haskell is destructive updates does not exist

We measures the performance for reading and writing of parallel database. We achieve to bring out the performance of the multi-core processor. Further, in order to indicate the availability of practical applications, we have developed a web bulletin board service.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　研究背景と目的

IT システムが巨大化していくにつれ, 障害発生事例が社会に与える影響もより大きな物となる. それに伴い, IT システムにおけるディペンダビリティへの注目が増している.

そこで, DEOS プロジェクトは IT システムにおけるディペンダビリティを担保する技術体系をまとめ, 制度化, さらには事業化を目指している. DEOS プロジェクトは 2006 年に独立行政法人科学技術機構 (JST) は CREST プログラムの 1 つとして始まったプロジェクトである. DEOS プロジェクトは, 変化し続ける目的や環境の中でシステムを適切に対応させ, 継続的にユーザが求めるサービスを提供することができるシステムの構築法を開発することを目標としている [2]. DEOS プロジェクトではそれらの技術体系を「オープンシステムディペンダビリティ」として定義し, それを DEOS プロセスとしてまとめた (図 1.1).

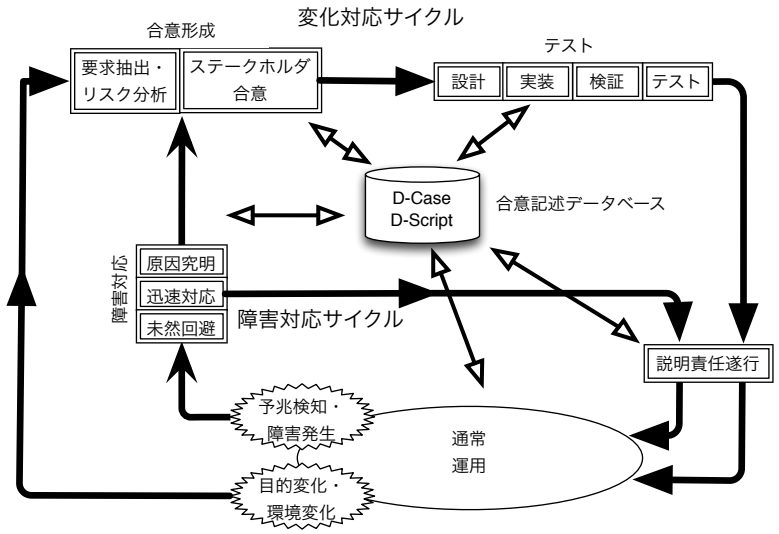



図 1.1: DEOS プロセス

DEOS プロセスには変化対応サイクルと障害対応サイクルの 2 つのサイクルがある. 変化対応サイクルは上流プロセスにおける対象システムのオープンシステムディペンダビリティを担保するためのプロセスである. 障害対応サイクルは対象システムの運用時に必要とされるプロセスである. DEOS プロセスは企画や設計という上流からシステム運用までの対象システムのライフサイクル全体に係わる. 対象システムの運用は, ステークホルダからの対象システムに対する要求に関する合意を始めとしたあらゆる議論に関する合意

をベースに行われる.

DEOS プロセスを構成している各 (サブ) プロセスからアクセスが行われるデータベースとして D-ADD(DEOS Agreement Description Database) がある [3]. D-ADD はステークホルダ合意と対象システムに存在するプログラム・コード, 及び対象システムの運用状態との間の一貫性を常に保つための機構を提供する. このようなデータベースは様々なデータを柔軟に格納する必要があり, データベーススキーマの頻繁な変化に対応する必要がある. これらのデータベースは, Web からアクセスされることも想定される. そのため, DEOS は Web サービスとして捉えることができる.

純粋関数型プログラミング言語 Haskell は, バッファオーバーフローや, クロスサイトスクリプティング, SQL インジェクションを事前の型検査で防ぐことができる. つまり, Haskell を用いることで信頼性の高い Web サービスを開発できると言える.

本研究の目標は, Haskell を用いて信頼性の高い Web サービスおよびデータベースの開発である. また, 並列実行で性能が線形に向上するスケーラビリティの達成を目指す. Web サービス のスケーラビリティを実現するための難点の一つはデータベースであり, データベースは並列にデータにアクセスできる設計が必要となる.

本研究では並列にデータへアクセスする手法として, 非破壊的木構造を利用する. 非破壊的木構造では, 排他制御をせずにデータへアクセスすることが可能でありスケーラビリティを確保できる [4][5][6].

実装した並列データベースの読み込みと書き込みについて性能を計測し, 読み込みに関して 12 コアで実行した場合, 1 コアで実行した場合と比較して, 10.37 倍 という性能向上率が確認でき, マルチコアプロセッサの性能を引き出すことができた.

また, Web 掲示板サービスを開発し, 既存の Java の非破壊的木構造データベースを用いた掲示板実装との比較をおこない, 読み込みで 3.25 倍, 書き込みで 3.78 倍の性能が確認できた.

# 第2章　Haskellとは

Haskell とは純粋関数型プログラミング言語である.

## 2.1　純粋関数型プログラミング

関数とは, 一つの引数を取り一つの結果を返す変換器のことである. 関数型プログラミング言語では, 関数を引数に適用させていくことで計算を行う.

### 関数の定義

既存の手続き型言語と異なり, 手順を記述するのではなく, この関数が何であるかということを記述する. 例えば, Haskell でフィボナッチ数列を定義するにはソースコード 2.1 のように記述する. Haskell は, 関数を引数を適用するという考えに基づいて, 抽象的なプログラミングを可能とする.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

ソースコード 2.1: フィボナッチ数列

fib :: Int -> Int は, 関数の型宣言である. この関数は, Int を受け取って Int を返す関数ということを示す. フィボナッチ数列の関数が三行に渡って書かれているが, これは Haskell のパターンマッチを活用している. 引数の値が 0 ならば 2 行目が呼び出され, 関数の結果は 0 となる. 引数の値が 1 ならば 3 行目が呼び出され, 関数の結果は 1 となる. 上から順に引数と一致する行がないか調べていき, 引数が 0 でも 1 でもなければ引数は n に束縛され, fib (n-2) + fib (n-1) が計算される.

フィボナッチ数列の関数は, 自分自身を使って再帰的に定義している. 再帰は, 関数型プログラミング言語において必要不可欠な要素である. 手続き型言語では, 配列とループを主に用いてプログラミングを行うが Haskell ではリスト構造と再帰を用いる.

### 変数の代入

純粋関数型プログラミングでは, 変数の代入は一度のみで後から書き換えることはできない. フィボナッチ数列の関数でも, 一度引数を束縛した n を書き換えることはできない.

関数にできることは, 何かを計算してその結果を返すことだけであり, 引数が同じならば関数は必ず同じ値を返すことが保証されている. この性質は関数の理解を容易にし, プログラムの証明を可能にする. 正しいと分かる単純な関数を組み合わせて, より複雑な正しい関数を組み立てていくのが関数型言語のプログラミングスタイルである.

### 高階関数

関数型プログラミング言語は, 関数を変数の値にすることができる. これは, 関数を第一級オブジェクトとして扱うことができるということである. Haskell は, 引数として関数を取ったり返り値として関数を返すことができる高階関数を定義できる.

高階関数の例として Haskell のカリー化が挙げられる. Haskell では, 全ての関数は一度に一つの引数だけを取る. 2 つの Int を受け取り, 大きい方を返す max の型はソースコード 2.2 のように定義できる.

```
max :: Int -> Int -> Int
max x y = if x <= y
          then y
          else x
```

ソースコード 2.2: max 関数

この関数定義に現れる-> は左結合である. 複数の引数を取るようにみえる関数は, 実際には 1 つの引数を取り, その次の引数を受け取る関数を返す（ソースコード 2.3）.

```
max :: Int -> (Int -> Int)
```

ソースコード 2.3: max 関数のカリー化

このように関数を返すことで全ての関数を一引数関数として表すことをカリー化という. カリー化によって, 関数を本来より少ない引数で呼び出した際に部分適用された関数を得ることができる（ソースコード 2.4）.

```
x = max 3 -- x は Int -> Int の関数
x 4 -- (max 3) 4 の結果として 4 が返される
```

ソースコード 2.4: 関数の部分適用

## 2.2 型

Haskell では, すべての式, すべての関数に型がある. 値の型は, その値が同じ型の別の値と何らかの性質を共有していることを示す. 例えば, 数値は加算できる, 文字列は表示できるといった性質である.

型はプログラムに抽象をもたらす. 型を導入することで, 低水準の詳細を気にせずプログラミングが可能になる. 例えば, 値の型が文字列ならば, どのように実装されているかということを気にせず, その文字列が他の文字列と同じように振る舞うとみなすことができる.

### 型の定義と型検査

Haskell は静的型検査によりエラーを検出することができる. Haskell では, 評価の際に型に起因するエラーが起きないことを保証している. 例えば, 引数として整数を受け取る関数に文字列を渡そうとしても Haskell のコンパイラはこれを受け付けない. Haskell は, すべての式, すべての関数に型があるためコンパイル時に多くのエラーを捕まえることができる.

エラーの検出の例として Haskell で最もよく使われるデータ構造であるリスト型で確認を行う. また, リスト型の定義とあわせてデータ型の定義についても説明する.

リストとは, 角括弧で包まれた同じ型の要素の並びである. [1,2,3] などと表現する. リストはソースコード 2.5 のように定義されている.

```
data [] a = [] | a : [a]
```

ソースコード 2.5: Haskell のリスト定義

data というのは新しい型を定義する際に利用するキーワードである. data キーワードのすぐ後にある [] a というのが型名である.

a というのは, 型変数と呼ばれるものである. 型変数は任意の型を取ることができる. リスト型は, Int のリスト, Float のリストといった様々な型のリストを作ることができ, 型変数を用いてそれを実現する. 型変数が何の型になるのかという情報は実行時には決まっており, 関数に渡される際に型の不整合が起きることはない.

= の右側には, 新しい型の定義として型の値となるものを列挙する. | は, もしくはという意味である. つまりリストは, [] もしくは a : [a] という値になることが分かる.

[] は空リストを表す. 型名と同じであるが, 型とは名前領域が異なるため問題ない. 型名はプログラム中では注釈としてしか使われないためである.

a : [a] は再帰的なデータ構造である. 値 a を : で繋げて, 再度リストの定義を呼んでいる.

この定義 [] | a : [a] は, a : a : a : .. : a : [] となり, a が任意個続いた後に, [] が来ることとなる. つまり, リストは無限に繋げることができ, リストの終端は空のリスト, つまり [] で終わる.

Haskell では, [1,2,3] という様にリストを表すが, これは単なるシンタックスシュガーであり, 内部では 1 : 2 : 3 : [] のように表現される.

このように定義した型であっても, Haskell は型検査を行う. リストは : を使うことで新しい要素を加えることができるが, 全ての要素は同じ型の要素である必要がある. 違った型の要素を付け加えようとすると Haskell はコンパイル時にエラーを出す. 例えば, Int のリスト [3,4] に, 文字である 'b' を付け加えようとした場合エラーが発生する（ソースコード 2.6）. 型の定義に型変数を用いても Haskell は, [3,4] が何の型になるのかといった情報を推論する. そのため文字である 'b' を Int のリスト [3,4] に付け加えることはできない.

```
<interactive>:3:7:
    Couldn't match type 'Int' with 'Char'
    Expected type: [Char]
      Actual type: [Int]
```

ソースコード 2.6: Haskell のコンパイル時エラー

型検査でも捕まえられないエラーは存在する. 例えば, 式 "1 `div` 0" は, 型エラーではないが, 0 での除算は定義されていないので評価時にエラーとなる.

### 型推論

Haskell は型推論を持つ. 型推論とは, 型の宣言をせずともそれを導くのに使われた関数の型シグネチャなどから自動的に型を決定する機構のことである. 型推論のない静的型付け言語は, プログラマが型の宣言を行うことが強制されるが Haskell では型の宣言は必須ではない.

例として, 開発したデータベースで実装した getChildren という関数に対して型推論を行ってみる（ソースコード 2.7）.

```
getChildren node path = elems (children (getNode node path))
```

ソースコード 2.7: getChildren 関数

型の注釈なしに関数を定義し, Haskell の対話環境である GHCi で型情報を取得してみる. 型情報を取得するには, :type の後ろに関数名を入力する（ソースコード 2.8）.

```
*Jungle> :type getChildren
getChildren :: Node -> [Int] -> [Node]
```

ソースコード 2.8: 型情報の取得

そうすると, 推論された型情報 Node -> [Int] -> [Node] が得られる. この型情報は期待する型の定義と一致する.

この情報は getChildren に含まれるいくつかの関数の型を確認することで導き出すことができる（ソースコード 2.9）.

```
getNode :: Node -> Path -> Node
elems :: Map k a -> [a]
children :: Node -> Map Int Node
```

ソースコード 2.9: getChildren に含まれる関数の型

例えば, getChildren の引数である, node と path は getNode に渡されているため, getNode の型である Node 型と Path 型であることが分かる. 返り値となる型は, elems の [a] となる. この a は, elems が受け取る Map の 2 つ目の型と一致するため, children の返り値である Map Int Node より, [Node] ということが分かる.

Haskell では, プログラマが型の宣言を行わずとも, 型を推論し型安全を保つ. しかし, 明示的な型宣言は可読性の向上や問題の発見に役に立つため, トップレベルの関数には型を明記することが一般的である.

## 2.3 モナド

Haskell では, さまざまな目的にモナドを使う. I/O 処理を行うためには IO モナドを使う必要がある. プログラミングを行うにあたり, I/O 処理は欠かせないため, モナドの説明を行う.

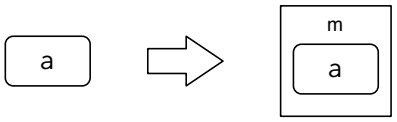
モナドとは, 型クラスの 1 つである. 型クラスは型の振る舞いを定義するものである. ある型クラスのインスタンスである型は, その型クラスに属する関数の集まりを実装する. これは, それらの関数がその型ではどのような意味になるのか定義するということである.

モナドとなる型は, 型変数として具体型をただ 1 つ取る. これにより何かしらのコンテナに包まれた値を実現する. モナドの振る舞いは型クラスとして実装し, 関数として return および >>= (bind) を定義する（ソースコード 2.10）.

```
return :: Monad m => a -> m a
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

ソースコード 2.10: モナドに属する関数の型

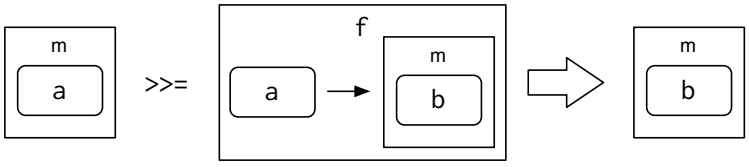
return は値を持ち上げてコンテナに包む機能を実装する（図 2.1）.




```
return :: Monad m => a -> m a
```

図 2.1: モナドに属する return 関数

bind は,「コンテナに包まれた値」と,「普通の値を取りコンテナに包まれた値を返す関数」を引数にとり, コンテナに包まれた値をその関数に適用する（図 2.2）. 適用する際, 前のコンテナの結果に依存して, 後のコンテナの振る舞いを変えられる.




```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

図 2.2: モナドに属する >>= (bind) 関数

この 2 つの関数を利用することにより, 文脈を保ったまま関数を繋いでいくことができる. Haskell の遅延評価は記述した順序で実行することを保証しないが, モナドの bind は実行順序の指定も可能で, IO モナドを bind で繋いだものは記述順に実行することができる.

### Maybe モナド

文脈を保ったまま関数を繋いでいくとはどういうことなのか, 具体例を用いて説明する. Maybe 型は失敗する可能性を扱うデータ型である（ソースコード 2.11）.

```
data Maybe a = Nothing | Just a
```

ソースコード 2.11: Maybe 型の定義

失敗したことを表す Nothing, もしくは成功したことを表す Just a のいずれかの値を取る.

Maybe 型が使われている例として, Data.Map の lookup 関数がある. Data.Map は Key と Value を保持する辞書型のデータ構造である. 何らかの Key を渡して, Data.Map から値を取得しようとした時, 返される値は Maybe Value 型である. 何かしらの値が取得できた場合は, Just a として Value に Just がついて返される. 取得できなければ, Nothing が返る.

Maybe モナドを使いたい場面は, 失敗するかもしれないという計算を繋いでいく時である. Maybe モナドの定義をみていく（ソースコード 2.12）.

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
```

ソースコード 2.12: Maybe モナドの定義

instance Monad Maybe where というのは, Maybe 型をモナド型クラスのインスタンスとするのに使う構文である. モナド型クラスに属させるために必要な関数である return と >>= (bind) を, Maybe ではどう振る舞うかを考え定義していく.

Maybe モナドの return は, 値を Just で包む. これがコンテナに包む機能という意味である.

>>= (bind) は, 「コンテナに包まれた値」と, 「普通の値を取りコンテナに包まれた値を返す関数」を引数にとり, コンテナに包まれた値をその関数に適用すると説明した. Maybe モナドの場合, コンテナが Nothing なら, そのまま Nothing を返す. コンテナが Just ならば, Just に包まれた値を取り出し, 「普通の値を取りコンテナに包まれた値を返す関数」に適用する.

失敗するかもしれない計算を繋いでいくとはどういうことなのか. 単純な関数を定義して説明する（ソースコード 2.13）.

```
up 4 = Nothing
up n = Just (n + 1)

down 0 = Nothing
down n = Just (n − 1)
```

ソースコード 2.13: up 関数と down 関数

関数 up と down を定義した. up の場合, 引数として 4 が渡された場合は上がれないため失敗, down の場合, 引数として 0 が渡された場合は下がれないため失敗と考える.

3 という値に down, down, up, up 繰り返していく時, モナドを使わない場合ソースコード 2.14 のように定義することになる.

```
updown :: Maybe Int
updown = case down 3 of
           Nothing -> Nothing
           Just place1 -> case down place1 of
                            Nothing -> Nothing
                            Just place2 -> case up place2 of
                                             Nothing -> Nothing
                                             Just place3 -> up place3
```

ソースコード 2.14: Maybe モナドを使わずに up と down を行う

case 式は, case と of の間の式を評価し, その値によって評価を分岐させるためのものである. case を受け取る − > の左の部分には式の値を書き, その式の値によって評価を分岐させる. 例えば, 3 行目は down 3 の結果が Nothing なら, Nothing を返すという意味である. Just の場合, 値を place1 という変数に束縛しその後の処理を続ける.

ソースコード 2.14 では, 毎回失敗したか成功したか確認するために非常に煩雑なコードとなってしまった. これを文脈を保ったまま, 関数を繋げられるモナドを使えばソースコード 2.15 のように記述できる.

```
return 3 >>= down >>= down >>= up >>= up
```

ソースコード 2.15: Maybe モナドを用いて up と down を行う

Maybe モナドを使うことで, この計算は失敗しているかもしれないという文脈を保ったまま処理を繋げていくことができる.

### IO モナド

Haskell で副作用を持つ処理を実行するには, IO モナドを利用する. IO モナドは, 処理系が現実世界に対する副作用に変換できるモナドである.

IO モナド自体は, 「文字を 1 文字取ってくる」といった命令書である. bind を使って, 小さな命令書を合成して大きな命令書を作成できる. 最終的に, main という名前をつけることで初めてランタイムにより実行される（ソースコード 2.16）.

```
main :: IO ()
main = putStrLn "Hello, World!"
```

ソースコード 2.16: Hello World と出力

Haskell の関数には副作用がないと述べたが, IO モナドを返す関数にも副作用は存在しない.

例えば, Jungle には getRootNode という IO を返す関数がある. 呼び出した状況によって, 返ってくるノードが違うため副作用があるようにみえる. しかし, 実際にこの関数が返すのは, 「ノードを取得する」という命令書である. どんな状況においても同じ命令書を返すため, 副作用はない.

## 2.4 遅延評価

Haskell では, 必要となるまで式の評価が遅延される. Haskell の対話環境である GHCi でどのように評価されるか確認することができる.

GHCi で評価を強制せずに値を表示する :sprint コマンドを使う. 例えば, x を定義した後に :sprint コマンドを使うとソースコード 2.17 のように表示される [注1].

```
ghci> let x = 1 + 2
ghci> :sprint x
x = _
```

ソースコード 2.17: sprint コマンドの使用例

_ は式が未評価であることを示している. Haskell ではこの未評価の部分を thunk と呼ぶ.

```
ghci> let y = map (+1) [1,2,3] :: [Int]
ghci> :sprint y
y = _
ghci> length y
3
ghci> :sprint y
y = [_,_,_]
ghci> head y
2
ghci> :sprint y
y = [2,_,_]
```

ソースコード 2.18: リストの評価の様子

リストを使ってどのように評価されるのか確認する. ソースコード 2.18 では, まず map 関数を利用してリストの要素を全て (+1) している. しかし, この計算は必要となるまで計算されない. 直後に sprint を行うと, ただ_が表示される.

リストの長さを表示する関数である length を実行後に sprint を行った場合は, リストの要素数を確認しているため、要素数分の thunk を持つリストとなる.

実際に値が必要となる関数を適用する. head はリストの先頭要素を取得する関数である. head を適用後に, sprint を行うと先頭要素だけが評価されたリストとなる.

リストの例より, Haskell では本当に値が必要となるまで決して計算しないことが分かる.

### 引数のメモ化

Haskell の遅延評価では引数のメモ化を行う. ある関数の仮引数が, その関数の本体に複数回出現したとしても評価回数が 1 回のみである. 例えば, ソースコード 2.19 は, 図 2.3 のようにメモ化される. y は未評価の thunk として同じ x を参照する. そのため, x が 2 回評価されることはない.

[注1] GHCi では, 関数や変数を定義する際には let が必要となる.

```
Prelude> let x = 1 + 2 :: Int
Prelude> let y = (x,x)
Prelude> :sprint y
y = (_,_)
```

ソースコード 2.19: メモ化

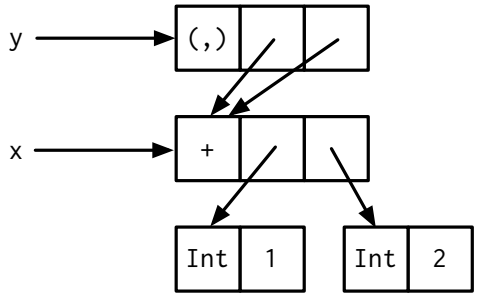



図 2.3: メモ化の様子

## 2.5 並列実行

Haskell はデフォルトではシングルスレッドで走る. 並列に実行したい場合は, -threaded 付きでコンパイルし, RTS の -N オプションを付けて実行する. -N オプションで指定された数だけ, OS のスレッドが立ち上がり実行される（ソースコード 2.20）.

```
$ ghc −O2 par.hs −threaded
$ ./par +RTS −N2
```

ソースコード 2.20: 並列実行の様子

当然これだけでは並列に動かず, 並列に実行できるようにプログラムを書く必要がある.

Control.Parallel.Strategies モジュールにある, Eval モナドを用いた並列化について説明する. Eval モナドは並列処理を行うためのモナドである. Eval モナドで並列処理を行う使用例を示す（ソースコード 2.21）.

```
import Control.Parallel.Strategies

main = print (runEval test)

num :: Integer
num = 1000000


test :: Eval (Integer, Integer)
test = rpar (sum' 0 num) >>= (\a ->
       rpar (sum' num (num*2)) >>= (\b ->
       return (a,b)))

sum' :: Integer -> Integer -> Integer
sum' begin end = if begin < end
                 then begin + (sum' (begin + 1) end)
                 else begin
```

ソースコード 2.21: Eval モナドの使用例

まず, Eval モナドが定義された, Control.Parallel.Strategies をロードし, Eval モナドを利用できるようにしている.

Haskell のプログラムは main という名前と実行したい関数を関連付けることで実行される. 今回は, print (runEval test) が実行される.

並列処理を行いたい箇所には, rpar を使う. rpar の引数とした関数は並列に実行可能であることを示すことができる. Eval モナドの関数の型をみると, rpar は, a を モナドに包み, 逆に runEval はモナドから a を取り出している（ソースコード 2.22）. rpar で並列化可能計算を示したあと, runEval で実行するという流れになる.

```
data Eval a
instance Monad Eval

runEval :: Eval a -> a
rpar :: a -> Eval a
```

ソースコード 2.22: Eval モナドの型

rpar を利用している test 関数は rpar モナドを bind で接続しているが, 入れ子構造となり書き方は煩雑となっている. Haskell にはモナドのために do 構文というシンタックスシュガーが用意されており, それを用いることでソースコード 2.23 のように書くことができる. do 構文を使うことで bind の入れ子構造を簡潔に書ける.

```
test :: Eval (Integer, Integer)
test = do
    a <− rpar (sum' 0 num)
    b <− rpar (sum' num (num∗2))
    return (a,b)
```

ソースコード 2.23: do 構文

sum' は 2 つの引数をとって, 開始点から終了点までの値をすべて足し合わせる関数である. 並列処理に負荷を与えるために使う. if で, 開始点が終了点を超えてないか調べ, 超えてなければ再帰的に呼び出して足し合わせを行う.

test で返ってくる型は, Eval (Integer, Integer) で, その後 runEval 関数を適用することで, (Integer, Integer) となる. そして最後に print で出力される.

Haskell は遅延評価を行うため, 必要となるまで式の評価が遅延される. 今回の場合, main の print がなければそもそも計算が行われない（ソースコード 2.24）.

```
main = return (runEval test)
```

ソースコード 2.24: 計算が行われない例

出力することで, 値が必要となるため計算される. また, test で返される 2 つの計算を 1 つだけ出力するようにした場合, 1 つのみ計算され並列実行は行われない（ソースコード 2.25）. fst は, (a,b) とあったとき a を取得する関数である.

```
main = print (fst (runEval test))
```

ソースコード 2.25: 並列実行されない例

Haskell で並列実行を行いたい場合は, 遅延評価に気をつける必要がある. rpar は単なるマーキングに過ぎず, rpar に渡したからといって直ちに並列実行が行われるわけではない. rpar で作成した Eval モナドを runEval に渡したあと, 値が必要となる print などを行えば, 並列に実行可能な部分が並列に実行される.

また, rpar を使用する際, 別の計算の値に依存する計算がある場合, その 2 つの計算は並列実行できない. 例えば, ソースコード 2.26 の場合は並列実行ができない.

```
test2 :: Eval (Integer, Integer)
test2 = do
    a <− rpar (sum' 0 num)
    b <− rpar (sum' num (if a < num then a else (num∗2)))
    return (a,b)
```

ソースコード 2.26: 前の計算に依存した計算

# 第3章　Haskellによる並列データベースの設計

## 3.1　マルチコアプロセッサで十分な性能を得るためには

現在, CPU はマルチコア化が進んでいる. マルチコアプロセッサで線形に性能向上をするためには, 処理全体で高い並列度を保つ必要がある. アムダールの法則 [7] によると, 並列度が 80 % の場合, どんなにコア数を増やしても性能向上率は 5 倍にしかならない.

CPU コア数に応じて, データベースを線形に性能向上させたい場合, 別々の CPU コアから同時にデータベースへアクセスできるようにし, 並列度を高める必要がある. 通常は, 同一のデータへアクセスする場合, 競合が発生してしまい処理性能に限界が生じる.

本研究では, 非破壊的木構造という手法を用いて競合が発生する問題を解決する. 競合を発生させないためには, 既にあるデータを変更しなければよい. 非破壊的木構造は, 変更元となる木構造を変更しない. そのため, 別々の CPU コアから並列にアクセスが可能であり, スケーラビリティを実現できる.

## 3.2　非破壊的木構造

非破壊的木構造は, 木構造を書き換えることなく編集を行う手法である. 既にあるデータを変更しないため, データの競合状態が発生せず, 並列に読み書きが行える.

また, 元の木構造は破壊されることがないため, 自由にコピーを行うことができる. コピーを複数作成することでアクセスを分散させることも可能である.

図 3.1 では, ノード 6 をノード A へ書き換える処理を行なっている.

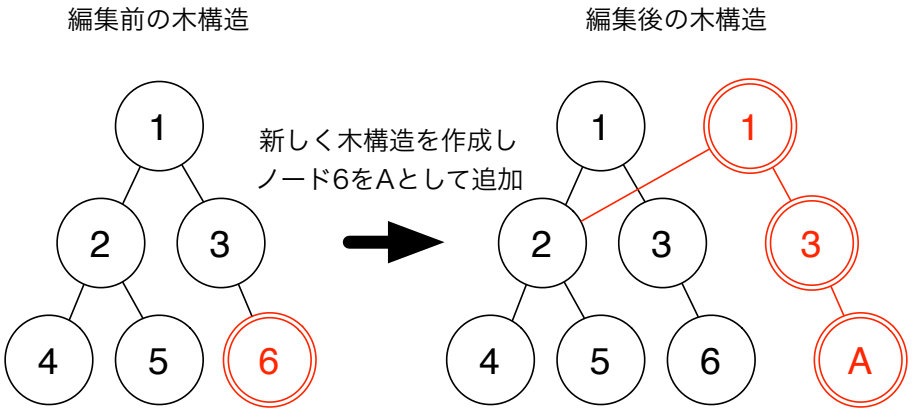



図 3.1: 木構造の非破壊的編集

この編集方法を用いた場合, 閲覧者が木構造を参照してる間に, 木の変更を行っても問題がない. 閲覧者は木が変更されたとしても, 保持しているルートノードから整合性を崩さずに参照が可能である（図 3.2）. 排他制御をせずに並列に読み書きが可能であるため, スケーラブルなシステムに有用である. 元の木構造は破壊されることがないため, 自由にコピーを作成しても構わない. したがってアクセスの負荷の分散も可能である.

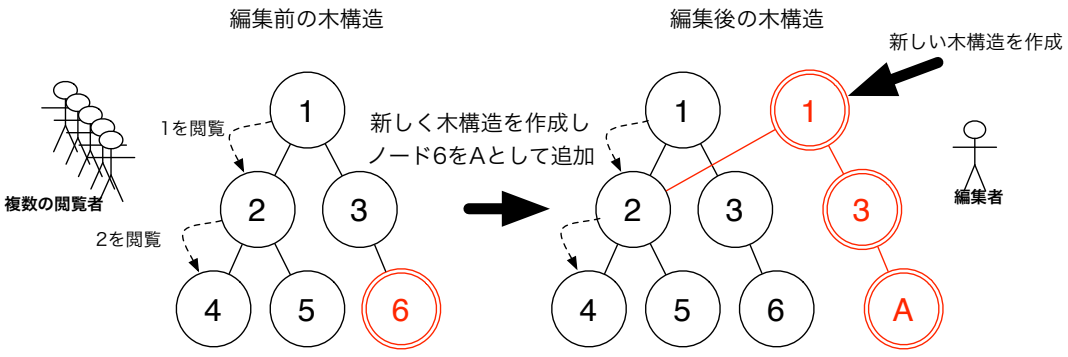



図 3.2: 並列に読み書きが可能な非破壊的木構造

## 3.3　ルートノード

非破壊的木構造では, ルートノードの管理が重要である. ルートノードは, 木の最新の状態を更新・参照するのに使う. ルートノードの情報は, 全てのスレッドで共有する必要があり, スレッドセーフに取り扱う必要がある. 一度ルートノードの情報を取得すれば, その後は自由に木構造へアクセスできる（図 3.3）.

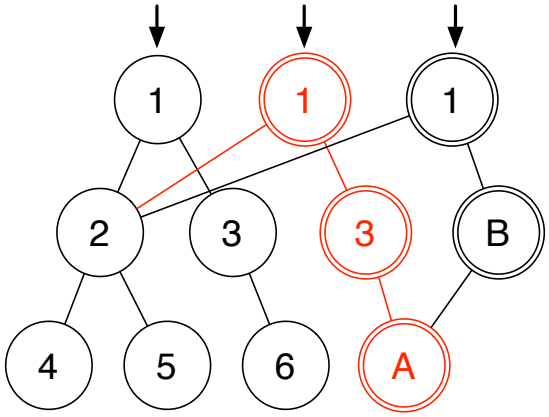



図 3.3: 非破壊的木構造のアクセス

ルートノードはスレッド間で共有する状態を持つため, Haskell では IO モナドを用いて状態を扱う必要がある. これには, Haskell のソフトウェア・トランザクショナル・メモリ（STM）を利用する. STM はブロックせず, スレッドセーフに状態を扱うことができる. STM を利用することでロック忘れによる競合状態や, デッドロックといった問題から解放される.

STM は, STM モナドという特殊なモナドの中でのみ変更できる. STM モナドの中で変更したアクションのブロックを atomically コンビネータを使ってトランザクションとして実行する. （atomically コンビネータを用いることで IO モナドとして返される）. いったんブロック内に入るとそこから出るまでは, そのブロック内の変更は他のスレッドから見ることはできない. こちら側のスレッドからも他のスレッドによる変更はみることはできず, 実行は完全に孤立して行われる. トランザクションから出る時に, 以下のことが 1 つだけ起こる.

- 同じデータを平行して変更したスレッドが他になければ, 加えた変更が他のスレッドから見えるようになる.
- そうでなければ, 変更を実際に実行せずに破棄し, アクションのブロックを再度実行する.

STM はロックの管理という煩雑な処理から逃れられるだけでなく, 並列性も向上する. どのスレッドもリソースにアクセスするために待つ必要はない. ルートノードの情報の取得は, 並列に行うことが可能である. ルートノードの情報の更新の場合は, 他から変更があれば再度やり直すということが自動的に行われる.

以前の実装では, ルートノードだけではなく非破壊的木構造全体を STM で管理していた [8]. しかし, 非破壊的木構造全体を STM で管理すると並列実行時に性能が出ないため, ルートノードのみの管理に変更を行った.

# 第4章　Haskellによる並列データベースの実装

本章では, 並列データベース Jungle の実装について述べる.

## 4.1　木構造データベース Jungle

非破壊的木構造データベース Jungle は, Haskell で実装された並列データベースである. 非破壊的木構造の方法に則った関数を提供する.

Jungle の基本的な使い方の手順について説明する.

1. 木構造を保持する Jungle を作成する
2. Jungle 内に新しい木を名前をつけて作成する
3. 木の名前を用いて, ルートノードの取得を行い, データを参照する
4. もしくは, 木の名前を用いて, ルートノードの更新を行う

### Jungle が持つデータ型

非破壊的木構造データベース Jungle が持つのデータ型を表 4.1 に表す. また, データ型内部のデータ構造を表 4.1 に表す.

木構造の集まりを表現する Jungle, 単体の木構造を表現する Tree がある. Node は子と属性を任意の数持てる. データ型として定義することで, 内部の型の整合性が保たれる. 例えば, Node の 1 つ目の型は (Map Int Node) となり他の型は許されない. 非破壊的木構造データベース Jungle のデータ型について, ひとつずつ説明する.

| 型名 | 概要 |
|---|---|
| Jungle | 木の作成・取得を担当する. |
| Tree | 木の名前とルートノードの情報を保持している. |
| Node | 基本的なデータ構造, 子と属性を任意の数持てる. |

表 4.1: Jungle が持つデータ型

| 型名 | データ構造 |
|---|---|
| Jungle | Jungle (TVar (Map String Tree)) |
| Tree | Tree (TVar Node) String |
| Node | Node (Map Int Node) (Map String ByteString) |

表 4.2: データ型のデータ構造

## 4.2 Jungle

Jungle は木構造の集まりを表現する. 木には名前がついており, Tree の情報と一緒に保持している（ソースコード 4.1）.

```
data Jungle = Jungle { getJungleMap :: (TVar (Map String Tree)) }
```

ソースコード 4.1: Jungle のデータ型の定義

Jungle のデータ構造は, Jungle (TVar (Map String Tree)) である. getJungleMap :: というのは, Haskell のレコード構文である.

レコード構文は, データ構造へのアクセサを提供する. getJungleMap は関数で, ソースコード 4.2 の型を持つ. これは, Jungle を受け取って, TVar (Map String Tree) を返す関数である.

レコード構文はデータ型を受け取って, :: の右側の型の値を取り出せる関数を作成すると思えば良い.

```
getJungleMap :: Jungle -> TVar (Map String Tree)
```

ソースコード 4.2: getJungleMap

Jungle の木の取り扱いには, Haskell の Data.Map を利用している. 型名は, Map である. Map は, 連想配列を扱うことのできるデータ構造である. 平衡木を用いて, 挿入や参照が O (log n) で済むように設計されている. Data.Map を理解するためにはリストで考えると分かりやすい.

```
data Map k a = Map [(k,a)]

lookup' :: Eq k => k -> Map k a -> Maybe a
lookup' k (Map []) = Nothing
lookup' k (Map ((k',a):xs)) = if k == k'
                              then Just a
                              else lookup k xs


insert :: k -> a -> Map k a -> Map k a
insert k a (Map x) = Map ((k,a):x)

test = Map [("key","value"),("fizz","buzz")]
```

ソースコード 4.3: getJungleMap

Map は, キーと値のペアのリストだと考えることができる. キーが一致する値を探す場合, lookup' を用いる. Maybe モナドを用いて, データがなければ Nothing, データがあれば Just に包んで返す. => の前にある, Eq k は, 型クラスの制約である. 内部で k と k' の同値性をテストしているため, k は同値性をチェックできる型クラス Eq に属している型である必要がある.

新たにキーと値のペアを, Map に追加するには insert を用いる. Haskell では, 受け取った引数を変更することができないため, ペアを追加した新しい Map を返す.

木の取り扱いには Haskell のソフトウェア・トランザクショナル・メモリ (STM) を利用して状態を持たせ, スレッド間で共有できるようにしてある. これは, 各スレッドから

木構造を新たに作成できるようにするためである. STM は, スレッド間でデータを共有するためのツールである. STM を利用することでロック忘れによる競合状態や, デッドロックといった問題から解放される. Jungle のデータ構造の Map の前に付いている TVar というのは, Transactional variables の略で, STM で管理する変数に対して利用する.

### Jungle と木の作成

Jungle は, 複数の非破壊的木構造を持つため, Map で木を管理している（図 4.1）. Tree には名前がついており, 複数のバージョンの Tree のノードのどれが最新かという情報を持っている.

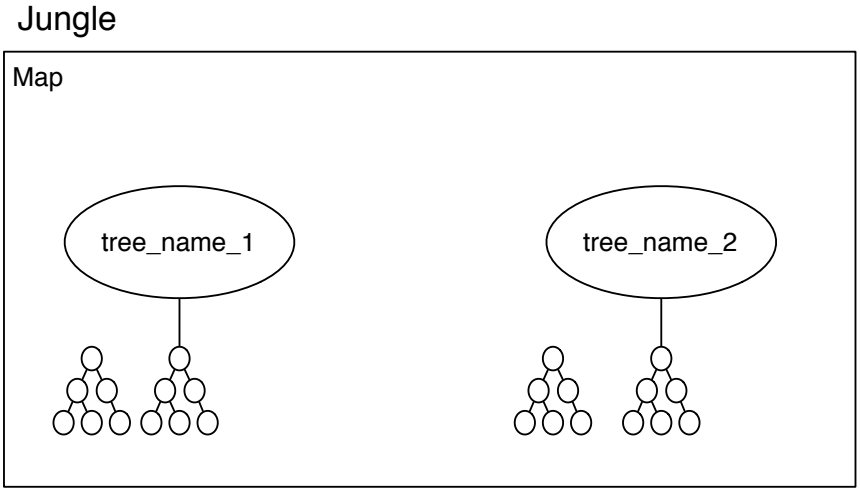



図 4.1: 複数の木を扱える Jungle

木構造の識別, つまり Map の キー には String を利用する. String は Haskell の文字列の型で, Char のリスト [Char] の別名である.

Jungle を作成するには, createJungle を用いる（ソースコード 4.4）. empty は空の Map を作成する関数である.

```
createJungle :: IO Jungle
createJungle = atomically $ do
    map <- newTVar empty
    return (Jungle map)
```

ソースコード 4.4: createJungle

createJungle は, 新たに STM の変数を作成する newTVar を実行する. newTVar などの STM の操作は STM モナド内で行う. 最後に atomically を行うことで, do 構文内がトランザクションとして実行される. STM の関数が持つ型をソースコード 4.5 に示す.

```
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

```
atomically :: STM a −> IO a
```

ソースコード 4.5: STM の関数

atomically の隣にある $ は関数適用演算子である. $ 関数は最も低い優先順位を持っており, 右結合である. 括弧を減らすのに使う. $ を使わない場合はソースコード 4.6 の様に記述することになる.

```
createJungle :: IO Jungle
createJungle = atomically (do
    map <− newTVar empty
    return (Jungle map))
```

ソースコード 4.6: 関数適用演算子を使わない場合

createJungle は, IO を返すため使う際には main に関連付ける必要がある.

## 4.3 Tree

Jungle が保持する木の情報は, 内部的には Tree というデータ型で保持している. Tree は木の名前と, ルートノードの情報を持っている（ソースコード 4.7）. 実際にユーザが Jungle を利用する際は, Jungle と木の名前を使ってルートノードを取ってくるため, Tree という構造は見えない.

ルートノードの情報はスレッド間で状態を共有する必要がある. スレッドセーフに取り扱う必要があるため, この情報も Haskell の ソフトウェア・トランザクショナル・メモリ (STM) を用いて管理している.

```
data Tree = Tree
          { rootNode :: (TVar Node)
          , treeName :: String }
```

ソースコード 4.7: Tree のデータ型の定義

新たな非破壊的木構造を作るには, createTree を用いる（ソースコード 4.8）. createTree は, createJungle で作成した Jungle と木の名前を String で受け取る.

```
createTree :: Jungle -> String -> IO ()
createTree (Jungle tmap) tree_name = atomically $ do
    map <- readTVar tmap
    tree <- emptyTree tree_name
    writeTVar tmap (insert tree_name tree map)

emptyTree :: String -> STM Tree
emptyTree tree_name = do
    node <- newTVar emptyNode
    return (Tree node tree_name)

emptyNode :: Node
emptyNode = Node (empty) (empty)
```

ソースコード 4.8: createTree

createJungle も STM を操作するため IO を返す. Jungle の持つ, 複数の木構造と名前を関連付けた Map を readTVar で取得する. ルートノードの管理のための STM の変数をもった Tree を作成し, Jungle の Map に insert する. そして最後に writeTVar を用いて STM を更新する. writeTVar は更新する先の変数と, 更新内容の 2 つを受け取る STM の関数である.

実際に createJungle と createTree を利用する時はソースコード 4.9 のように記述する.

```
main = do
    jungle <- createJungle
    createTree jungle "name␣of␣new␣tree␣here"
```

ソースコード 4.9: データベースと木の作成

### ルートノード

非破壊的木構造データベース Jungle では, 木の最新の状態を更新・参照するのにルートノードを使う. ルートノードは, 最新の木構造の根がどれかの情報を保持している（図

4.2）.

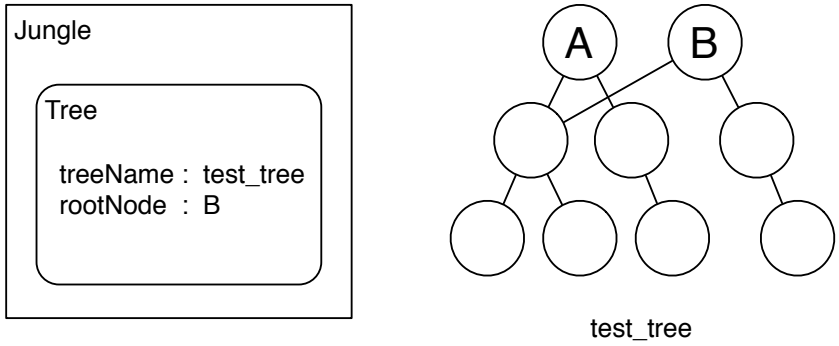



図 4.2: ルートノード

ルートノードに関する関数を説明する. getRootNode は, 最新のルートノードを取得できる. データベースと木の名前を渡すことで利用できる. 例えば, 図 4.2 の状態の時は, B というルートノードが取得できる.

getRootNode 関数の定義を示す（4.10）. まず, readTVar で Jungle が持つ map を参照する. Haskell の where キーワードは, 計算の中間結果に名前をつけるために用いられる. 今回は, root_node という map を受け取る関数を定義している. root_node map では, Jungle が持つ Map をみて取得しようとしている名前の木構造があるかどうか調べている. 木構造があった場合, rootNode という Tree に定義されているレコード構文のアクセサ関数を使って, (TVar Node) を取得する. 最後に, (TVar Node) に対して, readTVar を行うことで最新のルートノードが取得できる.

```
getRootNode :: Jungle −> String −> IO Node
getRootNode (Jungle tmap) tree_name = atomically $ do
    map <− readTVar tmap
    readTVar (root_node map)
    where
      root_node map = case lookup tree_name map of
                        Just x −> rootNode x
```

ソースコード 4.10: 最新のルートノードの取得

木構造を編集する関数は全て Node を受け取って Node を返す. その返ってきた Node をルートノードとして登録することで, 木構造の最新のルートノードが更新される. updateRootNode は, データベースと木の名前, 変更して返ってきた木構造の 3 つを渡す. updateRootNode をした後は, getRootNode で取得できるルートノードが更新された状態になっている.

updateRootNode 関数の定義を示す（4.11）. getRootNode と同じように, Tree の (TVar Node) を取得し, 最後に writeTVar を用いて更新している.

```
updateRootNode :: Jungle −> String −> Node −> IO ()
updateRootNode (Jungle tmap) tree_name node =
  atomically $ do
    map <− readTVar tmap
```

```
    writeTVar (root_node map) node
  where
    root_node map = case lookup tree_name map of
                      Just x -> rootNode x
```

ソースコード 4.11: ルートノードの更新

updateRootNodeWith は, ノードを更新する関数とデータベース, 木の名前を渡して利用する. ノードを更新する関数とは, ノードを受け取ってノードを返す関数である. (Node − > Node) がそれにあたる. この updateRootNodeWith を利用することで, getRootNode をした後に編集し updateRootNode を行う一連の操作が分断されずに行われることが保証される. updateRootNode 関数の定義を示す（4.12）. updateRootNodeWith では, 一連の操作を分断せずに行うために readTVar から writeTVar まで同じ STM モナド内で行っている. atomically に関数に do 構文で繋げた STM モナドを渡すことで, このブロックがトランザクションとして実行される.

```
updateRootNodeWith :: (Node -> Node) -> Jungle -> String -> IO ()
updateRootNodeWith f (Jungle tmap) tree_name =
  atomically $ do
    map <- readTVar tmap
    n <- readTVar (root_node map)
    writeTVar (root_node map) (f n)
  where
    root_node map = case lookup tree_name map of
                      Just x -> rootNode x
```

ソースコード 4.12: ルートノードの更新

並列データベース Jungle で他のスレッドと状態を共有する操作は, createJungle, createTree, getRootNode, updateRootNode, updateRootNodeWith で全てである. 並列データベース Jungle では, なるべく状態を共有しないようにすることで並列実行時の性能の向上を実現する. ソフトウェアトランザクショナルメモリは書き込み時に他から変更があった場合にやり直しという操作はあるものの, 読み込みに関してはノンブロッキングで高速に読み込める.

## 4.4 Node

Node は木構造を表現するデータ構造である．再帰的に定義されている．各ノードは children として子ノードを複数持つことができる（図 4.3）．

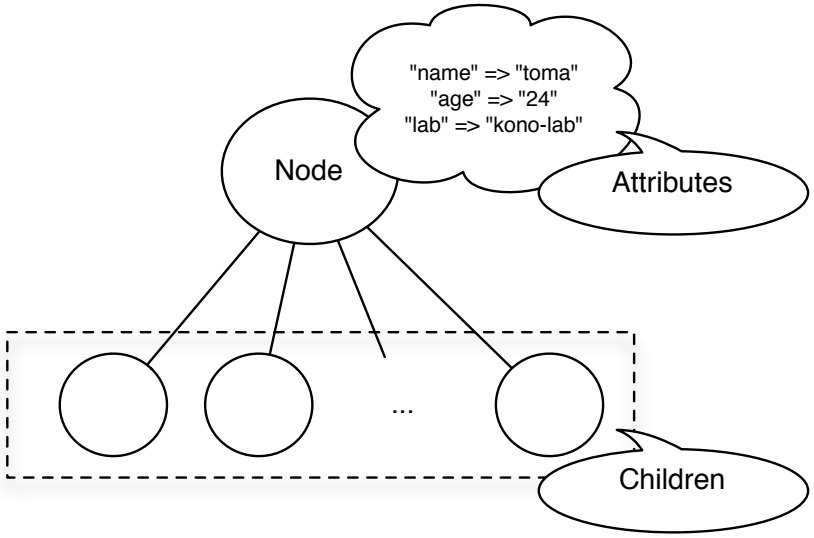



図 4.3: Node の構成要素

children および attributes も Data.Map を用いて定義されている（ソースコード 4.13）．

```
data Node = Node
          { children :: (Map Int Node)
          , attributes :: (Map String ByteString) }
```

ソースコード 4.13: Node のデータ型の定義

### 木の編集

木の編集には, Node を使う．木の編集に用いる関数は全て Node を受け取って Node を返す. 非破壊的木構造を利用しているため, getRootNode などで取得してきた Node は他のスレッドと干渉することなく自由に参照, 編集できる．これらの編集のための関数は, 編集後 updateRootNode するか, ひとつの関数にまとめて updateRootNodeWith をすることで木構造に反映させることができる.

編集対象のノードを指定するには, NodePath を利用する（図 4.4）. NodePath は, ルートノードからスタートし, ノードの子どもの場所を次々に指定したものである. Haskell の基本データ構造であるリストを利用している.

木の編集を行う関数を紹介する. 木の編集を行う関数の型の定義をソースコード 4.14 に示す.

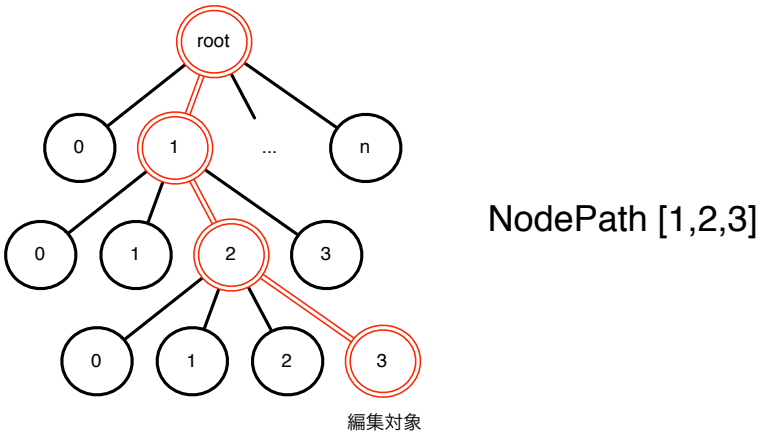



図 4.4: NodePath

```
addNewChildAt :: Node −> Path −> Node
deleteChildAt :: Node −> Path −> Position −> Node
putAttribute :: Node −> Path −> String −> ByteString −> Node
deleteAttribute :: Node −> Path −> String −> Node
```

ソースコード 4.14: 木の編集を行う関数

**addNewChildAt**　ノードに新しい子を追加できる. 更新対象となる木構造の Node と, どこに追加するかの情報である NodePath を渡す必要がある. 子の場所は, インクリメントしながら自動的に指定される.

**deleteChildAt**　ノードの子を削除できる. 更新対象となる木構造の Node と, どこのノードの子を削除するかという情報である NodePath, 削除したい子の場所を指定する Position を渡す必要がある.

**putAttribute**　ノードに属性を追加できる. 更新対象となる木構造の Node と, どこに属性を追加するかの情報である NodePath を渡す必要がある. 属性はキーと値があり, キーは String, 値は ByteString である.

**deleteAttribute**　ノードの属性を削除できる. 更新対象となる木構造の Node と, どこの属性を削除するかの情報である NodePath, 削除したい属性のキーである String を渡す必要がある.

これらの関数は, ほぼ同一の関数で定義できる. addNewChildAt を用いて説明する（ソースコード 4.15）.

```
addNewChildAt :: Node −> Path −> Node
addNewChildAt parent [] = addChildAt parent emptyNode
addNewChildAt parent (x:xs) = addChild parent x $ addNewChildAt x_node xs
  where
    map = children parent
    x_node = case lookup x map of
               Just x −> x

addChild :: Node −> Position −> Node −> Node
addChild node pos child = Node new_child attr
  where
    map = children node
    new_child = insert pos child map
    attr = attributes node

addChildAt :: Node −> Node −> Node
addChildAt node child = Node new_child attr
  where
    map = children node
    pos = (size map) + 1
    new_child = insert pos child map
    attr = attributes node
```

ソースコード 4.15: 木の編集を行う関数

非破壊的木構造の編集は再帰で定義できる．左結合となる$を使い, 対象のノードに到達するまで, addChild を繰り返す. addChild は, 指定したノードの Position に子を追加する. 引数として子となるノードが必要である. addChild を繰り返すことで, 下の階層から徐々に上に作られていく.

addNewChildAt, deleteChildAt, putAttribute, deleteAttribute といった, 非破壊的木構造の編集は, 対象のノードに対する操作以外は全て同じである．Path のリストが空になる, すなわち対象のノードに到達した時の操作だけが異なる．新しい子を追加するのが addNewChildAt, 指定されたポジションの子を削除するのが deleteChildAt, 指定されたキーと値を追加するのが putAttribute, 指定されたキーの値を削除するのが deleteAttribute である.

### 木の参照

木の参照にも参照対象となる木構造の Node を用いる. 参照関数の定義をソースコード 4.16 に示す.

```
getNode :: Node −> Path −> Node
getNode node [] = node
getNode node (x:xs) = getNode child xs
  where
    map = children node
    child = case M.lookup x map of
              Just x −> x
getAttributes :: Node −> Path −> String −> Maybe ByteString
getAttributes node path key = lookup key map
  where
    target = getNode node path
    map = attributes target

getChildren :: Node −> Path −> [Node]
```

```
getChildren node path = elems map
  where
    target = getNode node path
    map = children target

assocsChildren :: Node -> Path -> [(Int, Node)]
assocsChildren node path = assocs map
  where
    target = getNode node path
    map = children target

assocs :: Node -> Path -> [(String, ByteString)]
assocs node path = assocs map
  where
    target = getNode node path
    map = attributes target

numOfChild :: Node -> Path -> Int
numOfChild node path = size map
  where
    target = getNode node path
    map = children target

currentChild :: Node -> Path -> Maybe Node
currentChild node path = lookup pos map
  where
    target = getNode node path
    map = children target
    pos = size map
```

ソースコード 4.16: 参照関数

参照関数の基本的な流れは, getNode 関数を使って参照したい Path のノードを取ってくることである. そのノードには where キーワードを利用して, target という名前をつけている. target に対して, 子の Map や属性の Map を取得した後, lookup 関数などを適用する. elems, assocs, size などは Data.Map の参照関数で, Jungle ではその関数をそのまま利用している.

参照関数の基本的な機能をまとめて説明する.

**getAttributes**　対象の Path に存在する属性を Key を用いて参照できる.

**getChildren**　対象の Node が持つ全ての子を Node のリストとして返す. ある Node に存在する全ての子に対して, 参照を行いたい場合に利用する.

**assocsChildren**　対象の Node が持つ全ての子を Position とのタプルにし, そのタプルのリストを返す. ある Node に存在する全ての子に対して, 子の Position を取得しながら参照を行いたい場合に利用する.

**assocsAttribute**　対象の Node が持つ全ての属性を, キーと値のペアとし, そのペアのリストを返す. ある Node に存在する全ての属性に対して, 参照を行いたい場合に利用する.

**numOfChild**　対象の Node が持つ子どもの数を取得できる.

**currentChild**　対象の Node が持つ最新の子を取得できる.

# 第5章　性能評価

本章では, 非破壊的木構造データベース Jungle がマルチコアプロセッサで性能向上を果たせるのか確認する．また, 実用的な Web サービスが提供できるのか確認するために Web 掲示板サービスを開発し, 読み込みと書き込みの性能測定を行う．最後に開発した Web 掲示板サービスを利用して, Java との比較を行う.

## 5.1　計測環境

マルチコアプロセッサでの性能を確認するためコア数の多いサーバを用いる．本研究では, 学科が提供するブレードサーバを用いて, 計測環境を構築する．ブレードサーバの仕様を表 5.1 に示す.

論理コアは, Intel のハイパースレッディング機能のことである．ハイパースレッディングは, 1 つのプロセッサをあたかも 2 つのプロセッサであるかのように扱う技術であり, 同時に演算器などを利用することはできないため性能が 2 倍になるわけではないが, 概ね 20 %程度クロックあたりの性能が向上すると言われている.

| 名前 | 概要 |
|---|---|
| CPU | Intel(R) Xeon(R) CPU X5650@2.67GHz * 2 |
| 物理コア数 | 12 |
| 論理コア数 | 24 |
| Memory | 126GB |
| OS | Fedora 19 |

表 5.1: 学科が提供するブレードサーバの仕様

### Haskell および Java のバージョン

Haskell のコンパイラには The Glasgow Haskell Compiler（GHC）を使用する．GHC は, Haskell で最も広く使われているコンパイラである [9]. ソフトウェア・トランザクショナル・メモリをサポートするなど, 並列プログラミングのために Haskell の拡張が行われている．GHC の安定版は 7.6.3 だが, 並列実行時の IO マネージャーが改良されているため, リリース候補版である 7.8 を用いる [10].

| 言語 | バージョン |
|---|---|
| Haskell | Glasgow Haskell Compiler, Version 7.8 RC1 |
| Java | Java(TM) SE Runtime Environment (build 1.7.0_51-b13) |

表 5.2: ベンチマークで利用した Haskell と Java のバージョン

Haskell および Java のバージョンを表 5.2 に示す.
計測環境の構築方法については付録に記載する.

## 5.2 読み込みの性能計測

非破壊的木構造データベース Jungle を用いて, マルチコアプロセッサ上で並列に読み込みを行い, 線形に性能向上ができるか調査する.

### 計測方法

ブレードサーバ上で, Jungle で作成した木構造を並列に読み込んで性能計測を行う. 並列に実行する際に, 読み込みに負荷がかかるように木構造はある程度の大きさとする. 今回は木の深さが 8, ノードの数が約 80 万の大きさの木構造を使用する. 木構造を読み込み, ノードの数を数えるタスクを 1,000 個作成し並列実行する. 非破壊的木構造は, 木構造に変更を加えても他の読み込みのタスクに影響を与えない. そのことを確認するために, 木構造は各タスクに渡す前に無作為にノードを追加する.

### 計測結果

非破壊的木構造データベース Jungle の読み込みの計測結果を表 5.3 に示す.

CPU コア数を増やしていくと, 実行時間が短くなっていることが分かる. シングルスレッドで実行した場合と比較して, 2 スレッドで 1.97 倍, 12 スレッドで 7.17 倍の性能向上が見られる. 13 スレッド以上のハイパースレッディングを用いた計測は, 16 スレッド実行時に実行時間が最も短くなる. しかし, それ以降遅くなったりするなど安定しない結果となっている.

Haskell では並列実行時に OS の affinity（親和性）機能を使って OS スレッドを CPU コアに固定することができる. OS スレッドを CPU コアに固定するには, 実行時に-qa オプションを付けてプログラムを起動する. OS の親和性機能を使った場合, 2 スレッドで 1.80 倍, 12 スレッドで 10.37 倍の性能向上が見られ, 12 スレッド時の性能向上率が大幅に伸びている. 並列に読み込む場合, スレッドを同じプロセッサ上で実行させると性能が向上することがわかる.

しかし, 24 スレッドで実行する場合, 実行時間が大幅に伸びている. スレッドが CPU に固定されるため, 性能計測以外のプログラムがうまくスケジューリングされないためだと考えられる.

性能向上率のグラフを図 5.1 に示す. 親和性機能を使わない場合は, 6 スレッドまではほぼ線形に性能が向上する. 親和性機能を使った場合は, 12 スレッドまではほぼ線形に性能が向上する. ハイパースレッディング機能を用いた 13 スレッド以上では性能向上率が落ちる.

親和性機能を使った場合, 6 スレッド以降で性能差が出る理由として, 6 コア CPU を 2 個積んでいることが考えられる. スレッドが CPU をまたいでスケジューリングされなくなるため, 親和性機能を使うことで性能が向上するのだと考えられる.

非破壊的木構造データベース Jungle は 12 スレッドで 10.37 倍の性能向上が得ることができる. 読み込みにおいて, スケールするデータベースであると言える.

| CPU 数 | 通常実行 | 親和性機能利用 |
|---|---|---|
| 1 | 60.95 s | 61.00 s |
| 2 | 30.83 s | 33.95 s |
| 4 | 15.49 s | 16.10 s |
| 8 | 10.31 s | 8.79 s |
| 12 | 8.49 s | 5.88 s |
| 16 | 5.82 s | 5.81 s |
| 20 | 6.54 s | 5.48 s |
| 24 | 8.21 s | 125.09 s |

表 5.3: 読み込みの計測結果

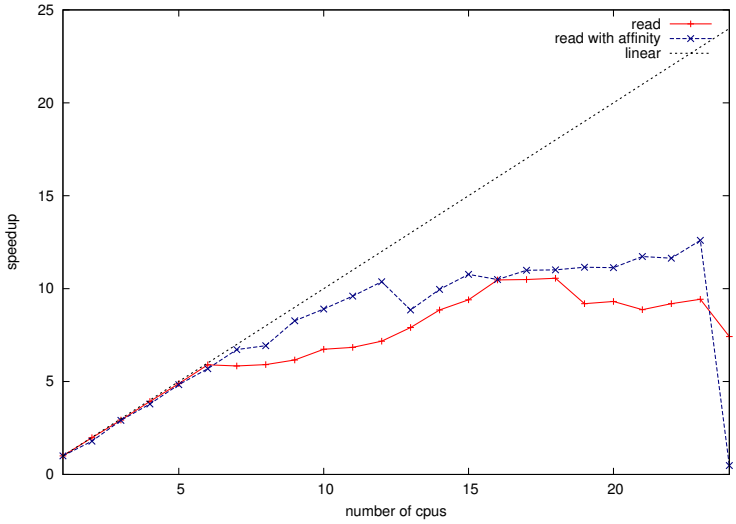



図 5.1: 読み込みの性能向上率

## 5.3　書き込みの性能計測

非破壊的木構造データベース Jungle を用いて, マルチコアプロセッサ上で並列に書き込みを行い, 線形に性能向上ができるか調査する.

### 計測方法

ブレードサーバ上で, Jungle に並列に書き込みを行い性能計測を行う. 木の深さが 6, ノードの数が約 1 万の大きさの木構造を作成し Jungle に登録するタスクを 1,000 個作成し, 並列に実行する. 書き込んだ木構造はノードの数が整合しているかどうか確認する. その後正確に書き込まれてるタスクの数を出力する. Haskell は遅延評価のため, 出力などを挟むことで評価が行われるようにしなければならない.

### 計測結果

非破壊的木構造データベース Jungle の書き込みの計測結果を表 5.4 に示す.

CPU コア数を増やしていくと, 実行時間が短くなっていることが分かる. シングルスレッドで実行した場合と比較して, 2 スレッドで 1.79 倍, 12 スレッドで 3.18 倍の性能向上が見られる. OS の親和性機能を使った場合, 2 スレッドで 1.61 倍, 12 スレッドで 3.82 倍の性能向上が見られ, 4 スレッド以上では親和性機能を使ったほうが実行時間が短くなる. また, 読み込みと同様に親和性機能を使った場合 24 スレッドでの実行時間が大幅に伸びる.

書き込みは, 読み込みと比べると Jungle への木構造の登録作業があるため性能向上率が下がる.

| CPU 数 | 通常実行 | 親和性機能利用 |
|---|---|---|
| 1 | 49.70 s | 49.61 s |
| 2 | 27.77 s | 30.76 s |
| 4 | 18.06 s | 18.05 s |
| 8 | 16.66 s | 12.50 s |
| 12 | 15.62 s | 12.96 s |
| 16 | 14.91 s | 13.11 s |
| 20 | 15.31 s | 13.84 s |
| 24 | 18.11 s | 71.66 s |

表 5.4: 書き込みの計測結果

性能向上率のグラフを図 5.2 に示す. Jungle へ木の登録する際に他のスレッドから登録があった場合, ソフトウェア・トランザクショナル・メモリが処理をやり直すため, 並列

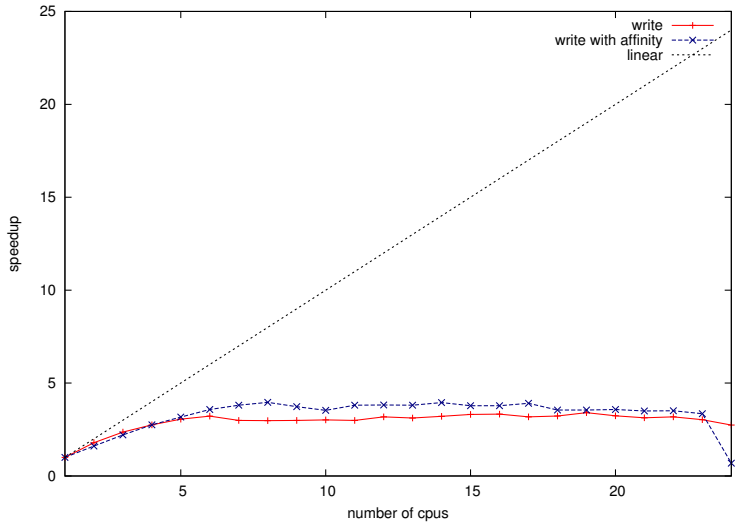



図 5.2: 書き込みの性能向上率

度が下がっていると考えられる. 速度向上が親和性機能を使った 8 スレッド実行時の 3.96 倍で頭打ちになっている.

非破壊的木構造データベース Jungle の書き込みは, 読み込みと比べて並列化率が低い. 読み込みが高速なため, 書き込みより読み込みが多用されるシステムに向いていると言える.

## 5.4　Web サービスに組み込んでの性能評価

並列データベース Jungle が実用的な Web サービスを提供できるのか調査する．Web 掲示板サービス開発し, 性能測定ツール weighttp [11] を用いて性能測定を行った.

### 5.4.1　Web 掲示板サービスの実装

木構造データベース Jungle と Haskell の HTTP サーバ Warp[12] を用いて Web 掲示板サービスを開発する．Warp を用いた Web サービスの構築法については付録に記載する．Warp は, ハイパースレッディングの効果がなくハイパースレッディング利用時に遅くなるため, ハイパースレッディングは利用しない.

Warp は並列に実行可能であり, 並列に実行している Warp に対して, Jungle を繋げるだけで Jungle を並列に動かすことができる．掲示板におけるデータベースへの書き込みは, 板の作成と, 板への書き込みがある．Jungle において, 板の作成は新しい木構造の作成, 板への書き込みは木構造へのノードの追加で表現する．掲示板へ実装した機能を表 5.5 に示す.

| 関数名 | 概要 |
|---|---|
| showBoard | 板の一覧を表示 |
| createBoard | 板の作成 |
| showBoardMessage | 板への書き込みの一覧を表示 |
| createBoardMessage | 板への書き込み |
| editMessage | 板への書き込みの編集 |

表 5.5: Web 掲示板サービスへ実装した機能一覧

Warp は並列 HTTP サーバであるが, 計測環境ではネットワークがボトルネックとなってしまう．ネットワークのボトルネックがどれぐらいあるのか調査するために, アクセスした際に ”hello, world” という文字列を返すだけのプログラムを作成し測定する．ネットワークを介さずに性能測定する場合, 性能測定ツール weighttp に 3 スレッド利用するため, Warp で利用するのは 8 スレッドまでとする．weighttp の設定は, リクエストの総数 100 万, 同時に接続するコネクションの数 1,000, 実行時のスレッド数 3, HTTP Keep-Alives を有効とする．また, 現在の安定版である 7.6.3 は IO マネージャーに問題があるが, どの程度影響があるか調べるために GHC 7.6.3 でコンパイルし, ネットワークを介さない状態での測定も行う．結果を表 5.6 に示す.

1 秒間あたりどれだけのリクエストを捌けるかという指標で比較を行う．大きければ大きいほど性能が良い．ネットワークを介する場合と介さない場合で, 8 コア時に 4.4 倍の性能差がある.

ネットワークを介さない場合は, 線形にスケールしている（図 5.3）．ネットワークを介した場合, およそ 60,000 req/s 程度で頭打ちとなってしまっている.

| CPU 数 | ネットワーク介す | ネットワークを介さない | GHC 7.6.3 |
|---|---|---|---|
| 1 | 44,008 req/s | 44,504 req/s | 31,602 req/s |
| 2 | 57,396 req/s | 95,760 req/s | 69,265 req/s |
| 4 | 60,593 req/s | 179,791 req/s | 133,897 req/s |
| 6 | 57,159 req/s | 241,240 req/s | 83,131 req/s |
| 8 | 59,697 req/s | 264,231 req/s | 31,694 req/s |

表 5.6: Warp の 性能測定結果

GHC 7.6.3 は, 5 スレッドあたりまでスケールするものの, それ以降遅くなり, 8 スレッド実行時はシングルスレッドで実行した場合と変わらないといったバグが存在することがわかる.

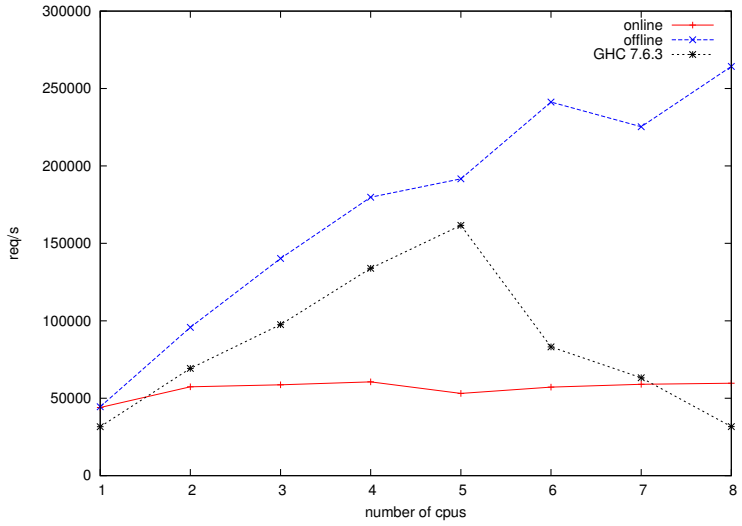



図 5.3: Warp の 性能測定結果

Web 掲示板サービスの性能測定には, 並列環境でどのようにスケールするか計測したいため, ネットワークを介さずに実験を行う. 実験環境を図 5.4 に示す.

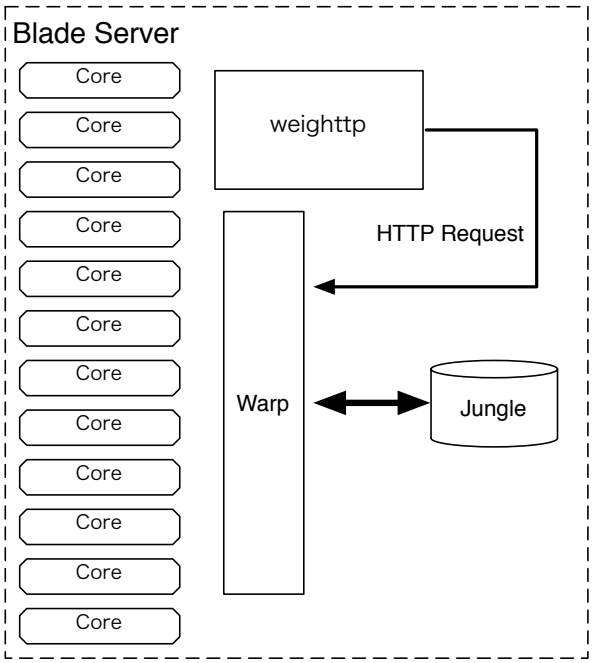



図 5.4: 実験環境

### 5.4.2 読み込みと書き込み

#### 計測方法

掲示板に対して読み込みと書き込みを行い, 負荷をかける. weighttp の設定は, リクエストの総数 100 万, 同時に接続するコネクションの数 1,000, 実行時のスレッド数 3, HTTP Keep-Alives を有効とする.

#### 計測結果

掲示板の読み込みと書き込みの計測結果を表 5.7 および図 5.5 に示す. 少ないスレッド数の場合に, 読み込みより書き込みのスループットが高いのは, 読み込みにあるデータベースの内容を HTML 形式に変換する作業がないためだと考えられる.

性能向上率を読み込みと書き込みで比較する. 読み込みは, シングルスレッドで実行した場合と比較して, 8 スレッドで 6.18 倍の性能向上が見られる. 書き込みは, シングルスレッドで実行した場合と比較して, 8 スレッドで 3.93 倍の性能向上が見られる. この結果は, 並列データベース Jungle が書き込みに比べて読み込みが高速であると示した Jungle 単体での実験結果と一致する.

| CPU 数 | 読み込み | 書き込み |
|---|---|---|
| 1 | 22,624 req/s | 28,552 req/s |
| 2 | 43,083 req/s | 53,765 req/s |
| 4 | 92,548 req/s | 98,691 req/s |
| 6 | 119,310 req/s | 99,009 req/s |
| 8 | 139,965 req/s | 112,212 req/s |

表 5.7: 掲示板を利用した読み込みと書き込みの計測結果

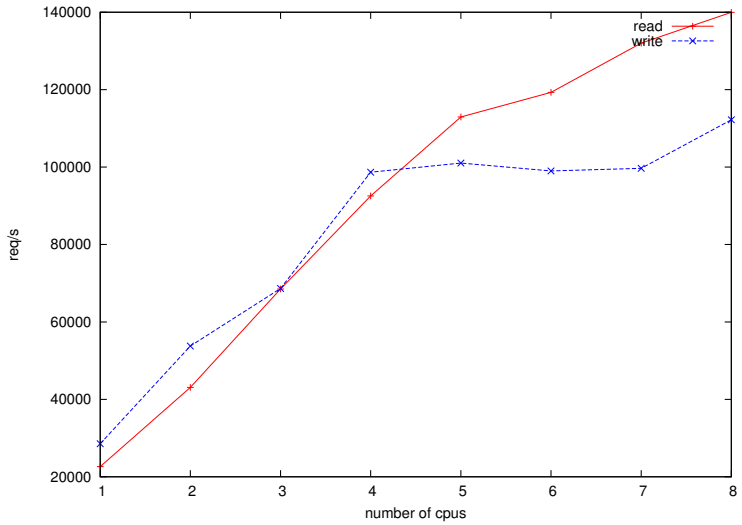



図 5.5: 掲示板を利用した読み込みと書き込みの計測結果

## 5.5　Java を用いた非破壊的木構造データベースとの比較

非破壊的木構造データベースは, Haskell 版と Java 版が存在する. Web 掲示板サービスを両方の言語で実装し, 比較を行う. Haskell ではフロントエンドとして Warp を利用したが, Java では Jetty を利用する. Jetty のバージョンは 6.1.26 を用いる.

Haskell 版と Java 版の Web 掲示板サービスをブレードサーバ上で実行する. 実際の利用を想定して計測を行うため, ブレードサーバをもう 1 台用意し, ネットワークを介して weighttp で負荷をかける. 10 スレッドから, 1 スレッドあたり 100 並列にリクエストを投げ, 合計 100 万リクエストを処理するのにかかる時間を計測する. Haskell と Java の測定結果を表 5.8 に示す.

| 測定 | Haskell | Java |
|---|---|---|
| 読み込み | 16.31 s | 53.13 s |
| 書き込み | 20.17 s | 76.4 s |

表 5.8: Haskell と Java の比較

Haskell 版は, Java 版と比較して読み込みで 3.25 倍, 書き込みで 3.78 倍の性能差が出ている. Haskell は 実用的な Web サービスが開発できる.

### Haskell の生産性

Haskell 版 Jungle が公開している関数の数は 18 で, コード行数は 284 行である. Java を用いた Jungle の実装は, 3390 行で, Haskell の実装は 1/12 程度のサイズとなっている.

コード行数が短くなった要因としては, Haskell が独自の再帰的データ構造の定義を言語としてサポートしていることや, 関数が参照透過性を持つため再利用が行いやすいということが考えられる.

同じ機能を実装する場合でも, Haskell は Java と比較してコード行数が短くなり生産性が向上する.

# 第6章　まとめと今後の課題

## 6.1　まとめ

本研究では, 関数型言語 Haskell を用いて並列データベースの実装をおこなった.

Haskell は, 型推論と型安全により信頼性の高いプログラムを書くことができる. 実装において, Haskell の表現力とコンパイル時に多くのエラーを捕まえるという特徴は, 開発期間およびコード行数の短縮に繋がった. また, 型安全により実行時に型エラーによってプログラムが終了するといったことがない.

読み込みに関して 12 コアで実行した場合, 1 コアで実行した場合と比較して, 10.77 倍という性能向上率が確認でき, マルチコアプロセッサの性能を引き出すことができた. また, Web 掲示板サービスを開発し, 既存の Java の非破壊的木構造データベースを用いた掲示板実装との比較をおこない, 読み込みで 3.25 倍, 書き込みで 3.78 倍の性能が確認できた.

## 6.2　今後の課題

非破壊的木構造データベース Jungle の今後の課題について述べる.

### 書き込み処理の性能向上率の上昇

データベースへの書き込み処理において, 12 スレッド時で 3.82 倍の性能向上となっている. 現在ルートノードの管理には, ソフトウェア・トランザクショナル・メモリを用いているが, 他のスレッドセーフな参照型を用いて性能改善が行えないか確認する. Haskell では, 様々なスレッドセーフな参照型が用意されている. ロックが制限的だが高速な IORef, 一般的なロックの使える MVar などである.

### 分散データベースとしての実装

現在, 並列環境で実行できるが, 今後は分散データベースとして実行できるようにしたい. トポロジーの形成機能や, サーバ間でのデータアクセスの仕組みを実装する必要がある. サーバ間で木構造の変更を共有するには, 木構造を何らかの情報に基づいて, マージする仕組みを導入する必要がある.

### 永続性の実装

非破壊的木構造データベース Jungle は, オンメモリ上で動作するデータベースである. 並列性を損なわない形で, ディスクへの書き出しを実現したい. 実装としては, 書き出しを担当するスレッドを作成するといったことが考えられる.

# 付録 A　計測環境の構築

ウェブアプリケーションのベンチマークを行う際, サーバの設定に注意を払う必要がある. 適切に設定を行わないと, サーバがボトルネックとなってしまい正しい結果が得られない. ウェブアプリケーションのベンチマークを行う際の注意点について述べる.

データを受信したり送信したりするのは OS カーネルである. 多くの TCP パケットを要求し, 各パケットのサイズが 1500 バイトといった大きなファイルを提供する場合, ウェブアプリケーションというより OS カーネルのテストになってしまう.

接続には, HTTP Keep-Alives を利用する. 新しい TCP 接続を確立するのはとても遅く, OS カーネルによって行われる. 毎秒多くの新しい接続を作成するベンチマークを行うと, OS カーネルのテストとなってしまう.

アプリケーションや OS カーネルが完全にハードウェアを使用できるようにするためにいくつか調整を行う必要がある. 最初の問題は, ファイル記述子の欠如である. デフォルトはプロセスあたり, 1,024 files で非常に貧弱な結果しか得られない. ファイル記述子の現在のリミットはソースコード 1 のコマンド取得できる.

```
$ ulimit −aH
```

ソースコード 1: ファイル記述子のリミットの取得

リミットを変更するには, ソースコード 2 のコマンドを実行する.

```
$ sudo sh −c ulimit −HSn 200000
```

ソースコード 2: ファイル記述子のリミットの設定

再起動後も有効にするためには, システムファイルの編集を行う.
/etc/security/limits.conf へソースコード 3 の記述を追加する.

```
* soft nofile 200000
* hard nofile 200000
```

ソースコード 3: リミットの設定の追加

次に問題となるのは listen キューの制限である. listen キューとは, 保留中のコネクションが繋がれるキューのことである. このキューの長さの制限が小さいと, 同時にたくさんのコネクション要求がきた場合, 制限を超えた要求を拒否する. listen キューや, その他の設定も含めてベンチマーク用にサーバの設定を変更する.

/etc/sysctl.conf にソースコード 4 の記述を追加する.

```
fs.file−max = 5000000
net.core.netdev_max_backlog = 400000
net.core.optmem_max = 10000000
net.core.rmem_default = 10000000
net.core.rmem_max = 10000000
net.core.somaxconn = 100000
net.core.wmem_default = 10000000
net.core.wmem_max = 10000000
net.ipv4.conf.all.rp_filter = 1
net.ipv4.conf.default.rp_filter = 1
net.ipv4.ip_local_port_range = 1024 65535
net.ipv4.tcp_congestion_control = bic
net.ipv4.tcp_ecn = 0
net.ipv4.tcp_max_syn_backlog = 12000
net.ipv4.tcp_max_tw_buckets = 2000000
net.ipv4.tcp_mem = 30000000 30000000 30000000
net.ipv4.tcp_rmem = 30000000 30000000 30000000
net.ipv4.tcp_sack = 1
net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_timestamps = 1
net.ipv4.tcp_wmem = 30000000 30000000 30000000
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
```

ソースコード 4: システム設定の変更

ファイルを保存後, 設定を反映させるにはソースコード 5 のコマンドを実行する.

```
$ sudo sysctl −p /etc/sysctl.conf
```

ソースコード 5: 設定の反映

ベンチマークを行う際, 小さなテストでは妥当性が低くなってしまうので注意する. TCP/IP のスタックは保守的な方法で動作し, ダウンロード速度に合わせて徐々に速度を増大させるためである.

また, テストするサーバより遅いベンチマーククライアントを用いると正しい結果は得られない. シングルスレッドで稼働したり, Ruby や Python といった低速なベンチマークツールでテストを行うと, すべてのテストする対象が同じパフォーマンスを持っているように見えてしまう.

### weighttp

ウェブアプリケーションの性能測定には, weighttp を用いる. weighttp は Web サーバの性能測定ツールで, マルチコア CPU を使ってテストできる [11]. また, livev を使うことで, モダンなポール・システムコールを利用し, 測定性能を向上できるといった特徴を持つ. 同様の性能測定ツールには, Apache Bench や httpref が存在するが非力であり, ボトルネックとなってしまうため使用しない.

weighttp を起動するには, ソースコード 6 のコマンドを入力する.

```
$ weighttp −n 1000000 −c 1000 −t 10 −k "http://bldsv12.cr.ie.u-ryukyu.ac.jp:3000"
```

ソースコード 6: weighttp の起動

起動時には対象のサーバの URL を記述する他に, いくつかのオプションを指定できる.

- n ... HTTP リクエストの総数
- c ... 同時に接続するコネクションの数
- t ... 作製するネイティブスレッドの数
- k ... HTTP Keep-Alives を有効にする

# 付録 B　Warp を用いたウェブアプリケーションの構築

Warp は, 軽量・高速な HTTP サーバである [12]. Haskell の軽量スレッドを活かして書かれている. Haskell のウェブフレームワークである Yesod のバックエンドとして用いられており, 現在も開発が続けられている.

Warp を用いてウェブアプリケーションを構築する方法について説明する.

```
application counter request = function counter
  where
    function = routes $ pathInfo request

routes path = findRoute path routeSetting

findRoute path [] = notFound
findRoute path ((p,f):xs)
    | path == p = f
    | otherwise = findRoute path xs

routeSetting = [(["hello"], hello),
                ([hello","world"], world)]

notFound _ = return $
    responseLBS status404 [("Content-type", "text/html")] $ "404"

hello _ = return $
    responseLBS status200 [("Content-type", "text/html")] $ "hello"

world counter = do
    count <- lift $ incCount counter
    return $ responseLBS status200 [("Content-type", "text/html")] $
      fromString $ show count

incCount counter = atomicModifyIORef counter (\c -> (c+1, c))

main = do
  counter <- newIORef 0
  run 3000 $ application counter
```

ソースコード 7: Warp を用いたウェブアプリケーションの例

ソースコード 7 は, URL によって出力する結果を変更するウェブアプリケーションである. /hello/world へアクセスがあった場合は, インクリメントされる counter が表示される.

**main**　HTTP サーバを起動するには, Warp の run 関数を利用する. run 関数は, 利用する Port 番号と, application というリクエストを受けて何かしらのレスポンスを返す関数の 2 つを引数として受け取る.

関数型言語では, 関数を第一級オブジェクトとして扱える. また, 今回は Haskell のカリー化された関数の特性を利用し, main 内で作成した IORef 型の counter を部分適用させている.

IORef を用いることで, Haskell で更新可能な変数を扱うことができる. 参照透過性を失うようにみえるが, Haskell は IO モナドを利用することで純粋性を保っている. IORef 自体が入出力を行うわけではなく, 単なる入出力操作の指示にすぎない. IO モナドとして

糊付けされた単一のアクションに main という名前を付けて実行することで処理系が入出力処理を行う.

**application 及び routes , findRoute**　application の実装では, routes という関数を独自に定義して, URL によって出力を変更している. application に渡されるリクエストはデータ型で様々な情報が含まれている. その中のひとつに pathInfo という, URL から hostname/port と, クエリを取り除いたリストがある. この情報を routes という関数に渡すことで, routeSetting というリストから一致する URL がないか調べる. routeSetting は, URL のリストとレスポンスを返す関数のタプルのリストである.

**notFound 及び hello**　レスポンスを返す関数は, いくつか定義されている. その中で利用されている responseLBS は文字列からレスポンスを構築するためのコンストラクタである.

**world 及び incCount**　world は, インクリメントされる counter を表示するための関数である. IORef 内のデータは直接触ることができないため, incCount 内で atomicModifyIORef を利用してデータの更新を行なっている. atomicModifyIORef は, データの更新をスレッドセーフに行うことができる. また, responseLBS で構築したレスポンスは, Resource T というリーソスの解放を安全に行うために使われるモナドに包まれている. lift 関数を用いて, incCount の型を持ち上げ調整している.

プログラムを例にして説明したが, Warp は容易にプログラムに組み込むことができる. 本研究では, 非破壊的木構造データベース Jungle と Warp を組み合わせて, 掲示板ウェブアプリケーションを開発した.

# 謝辞

本研究を行うにあたり, 日頃より多くの助言, ご指導いただきました河野真治助教授に心より感謝申し上げます.

本研究は, JST/CREST 研究領域「実用化を目指した組み込みシステム用ディペンダブル・ オペレーティングシステム」D-ADD 研究チームとして実施しました. 研究の機会を与えてくださった, 株式会社 Symphony の永山辰巳さんに感謝します.

また, データベースの実装にあたり, 多くの議論にお付き合い頂いた大城信康さん, 並列信頼研究室の全てのメンバーに感謝いたします.

# 参考文献


[1] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[2] 所眞理雄. DEOS プロジェクト研究成果集 Dependability Engineering for Open Systems, 2013.

[3] 永山 辰巳, 横手 靖彦. オープンシステムディペンダビリティと D-Case を繋ぐリポジトリ, 2013.

[4] 玉城将士, 河野真治. Cassandra を使った CMS の PC クラスタを使ったスケーラビリティの検証. 日本ソフトウェア科学会, August 2010.

[5] 玉城将士, 河野真治. Cassandra を使ったスケーラビリティのある CMS の設計. 情報処理学会, March 2011.

[6] 玉城将士, 河野真治. Cassandra と非破壊的構造を用いた CMS のスケーラビリティ検証環境の構築. 日本ソフトウェア科学会, August 2011.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, 1967.

[8] 當眞大千, 河野真治, 永山辰巳. Haskell による非破壊的木構造を用いた CMS の実装. 日本ソフトウェア科学会, September 2013.

[9] The glasgow haskell compiler. `http://www.haskell.org/ghc/`. [Online; accessed 29-Jan-2014].

[10] Andreas Voellmy, Junchang Wang, Paul Hudak, and Kazuhiko Yamamoto. Mio: A high-performance multicore IO manager for GHC. *Haskell Symposium*, September 2013.

[11] lighttpd/weighttp. `https://github.com/lighttpd/weighttp`. [Online; accessed 2-Feb-2014].

[12] The warp package. `http://hackage.haskell.org/package/warp`. [Online; accessed 29-Jan-2014].

# 発表履歴

- Haskell による非破壊的木構造を用いた CMS の実装,
  當眞大千, 河野真治 (琉球大学), 永山辰巳 (株式会社 Symphony)
  日本ソフトウェア科学会 30 回大会, Sep, 2013
- Haskell による Web Service 構築入門,
  當眞大千, 河野真治 (琉球大学)
  オープンソースカンファレンス 2013 Okinawa, July, 2013
- CeriumTaskManager におけるマルチコア上での並列実行機構の実装,
  當眞大千, 金城裕, 河野真治 (琉球大学)
  第 53 回プログラミングシンポジウム, Jan, 2012