関数型言語 Haskell による
並列データベースの実装

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。Haskell は純粋であるため、関数は引数が同じならば必ず同じ値を返すことが保証されている。

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、10.77 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、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 1.87 倍、書き込みで 2.3 倍の性能が確認できた。

目 次

第 1 章　研究背景と目的 **1**

第 2 章　Haskell とは **2**
- 2.1 純粋関数型プログラミング . . . 2
- 2.2 型 . . . 3
- 2.3 モナド . . . 5
- 2.4 並列実行 . . . 8

第 3 章　Haskell による並列データベースの設計 **10**
- 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 . . . 10
- 3.2 非破壊的木構造 . . . 11
- 3.3 ルートノード . . . 14

第 4 章　Haskell による並列データベースの実装 **16**
- 4.1 木構造データベース Jungle . . . 16
- 4.2 木構造データベース Jungle の実装 . . . 22
  - 4.2.1 Jungle . . . 22
  - 4.2.2 Tree . . . 22
  - 4.2.3 Node . . . 23
- 4.3 Haskell の並列処理 . . . 24
- 4.4 Haskell の生産性 . . . 24

第 5 章　性能評価 **25**
- 5.1 計測環境 . . . 25
- 5.2 読み込みの性能計測 . . . 26
- 5.3 書き込みの性能計測 . . . 28
- 5.4 ウェブアプリケーションに組み込んでの性能評価 . . . 30
  - 5.4.1 掲示板ウェブアプリケーションの実装 . . . 30
  - 5.4.2 読み込み . . . 30
  - 5.4.3 書き込み . . . 31
  - 5.4.4 Java を用いた非破壊的木構造データベースとの比較 . . . 32

第 6 章　まとめと今後の課題 **34**
6.1　まとめ 34
6.2　今後の課題 34

付録 **36**
付録 A 計測環境の構築 36
付録 B Warp を用いたウェブアプリケーションの構築 38

謝辞 **40**

参考文献 **41**

発表文献 **42**

図 目 次

2.1 モナドに属する return 関数 5
2.2 モナドに属する >>= (bind) 関数 5

3.1 木構造の非破壊的編集 11
3.2 ステップ 1 : 変更したいノードまでのパスを求める 12
3.3 ステップ 2 : 変更したいノードをコピーし、コピーしたノードの内容を変更した新しいノードを作成する 12
3.4 ステップ 3 : 求めたパス上に存在するノードをルートノードまでコピーする 12
3.5 ステップ 4 : 影響のないノードは共有する 13
3.6 並列に読み書きが可能な非破壊的木構造 13
3.7 排他制御なしの非破壊的木構造のアクセス 14

4.1 複数の木を扱える Jungle 17
4.2 ルートノード 18
4.3 NodePath 19
4.4 Node の構成要素 23

5.1 読み込みの性能向上率 27
5.2 書き込みの性能向上率 29

表 目 次

4.1 内部のデータ型 17

5.1 学科が提供するブレードサーバの仕様 25
5.2 ベンチマークで利用した Haskell と Java のバージョン 26
5.3 読み込みの計測結果 27
5.4 書き込みの計測結果 28
5.5 掲示板ウェブアプリケーションへ実装した機能一覧 30
5.6 掲示板を利用した読み込みの計測結果 31
5.7 Warp の計測結果 31
5.8 掲示板を利用した書き込みの計測結果 32
5.9 Haskell と Java の比較 32

第1章　研究背景と目的

Web サービスの脆弱性狙った攻撃が頻繁に発生している。脆弱性を悪用されると、Web サービス運営者は賠償など多大な損害を受ける可能性がある。純粋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 [1][2][3]。

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、10.77 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 1.87 倍、書き込みで 2.3 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。

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-> は左結合である。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。引数が、0 ならば 2 行目の fib が呼び出される。引数が、1 ならば 3 行目の fib が呼び出される。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。

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。

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。つまりこれは、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。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。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。

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。抽象を導入することで、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細かいことは気にせず、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。

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。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リストで確認を行う。また、リスト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。リストは以下のように定義されている。

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

ソースコード 2.2: 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。[] というのが型名である。

型名の右に a というのがあるが、これは多相型を表すのに使う型変数である。リストは、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、Haskell の型安全を保つ。

= の右側が、新しい型の定義である。| 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。何らかの型の値 a を、: で繋げて、再度リストの定義を呼んでいる。

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。[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。

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、型変数は 1 つであり、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以下の様なエラーが発生する。

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

ソースコード 2.3: 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関数に対して型推論を行ってみる。関数の詳細な動作はデータベースの実装で述べるが、getChildren という、関数がある。

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

ソースコード 2.4: 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。

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

ソースコード 2.5: 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。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。

しかしながら、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。

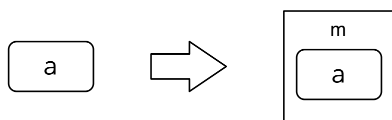
モナドとは、型クラスの 1 つである。型クラスは型の振る舞いを定義するものである。ある型クラスのインスタンスである型は、その型クラスに属する関数の集まりを実装する。これは、それらの関数がその型ではどのような意味になるのか定義するということである。

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。

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

ソースコード 2.6: モナドに属する関数

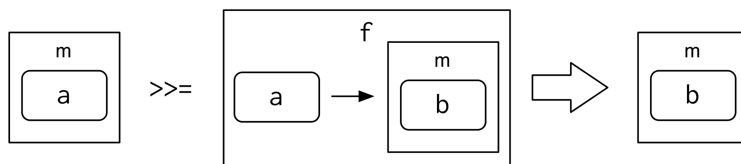
return は値を持ち上げてコンテナに包む機能を実装する（図 2.1）。



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。

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

ソースコード 2.7: 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 型である。何かしらの値が取得できた場合は、Just a として値に 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。

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

ソースコード 2.8: Maybe モナドの定義

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更に説明していく。

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

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

ソースコード 2.9: up 関数と down 関数

関数 up と down を定義した。4 以上はこれ以上、上がれないため失敗、0 以下はこれ以上、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以下のように定義することになる。case 式は、case と of の間の式を評価し、その値によっ評価を分岐させることができる。case を受け取る $->$ の左の部分はパターンマッチを行うこともできる。また、case は式のため、$->$ の右の部分の全ての型は一意である必要がある。Haskell では分岐によって返ってくる値の型が異なるということはできない。

```
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 2.10: Maybe モナドを使わずに up と down を行う

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 モナドを使えば以下のように記述できる。

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

ソースコード 2.11: 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扱うことができる。

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 モナド自体は単なる命令書であり、命令ではない。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。

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、getChar という関数がある。呼び出した状況によって、返ってくる文字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、「一文字読み込む」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 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。

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

ソースコード 2.12: 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 モナドで並列計算を行う使用例を示す。

```
import Control.Parallel.Strategies

main = print (runEval test)

num :: Integer
num = 1000000

test :: Eval (Integer, Integer)
test = do
    a <- rpar (sum' 0 num)
    b <- rpar (sum' num (num*2))
    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 2.13: 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。

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。

```
data Eval a
instance Monad Eval

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

ソースコード 2.14: Eval モナド

並列処理を行うには、rpar を使う。rpar で挟んだ関数は並列に実行される。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。rpar で並列化可能計算を示したあと、runEval で実行する。

test の = のすぐ後にある do はモナドのためのシンタックスシュガーであり、do 構文と呼ばれる。Haskell では、モナドを単一のモナドとするために、do 構文を使う。>>= (bind) を使ってまとめることもできるが、do 構文を使うことで bind の入れ子構造を手続き言語のような書き方で書くことができる。do 構文を用いない場合、以下の様な式になる。

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

ソースコード 2.15: do 構文を使わない場合

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、最後の print がなければそもそも計算が行われない。

並列に動くように処理を分割した後は、Haskell の遅延評価に気をつける必要がある。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。

rpar を使用する際に気をつけるのは、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ということである。例えば、以下のような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 2.16: 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たなければならない [4]。

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ればよい。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。

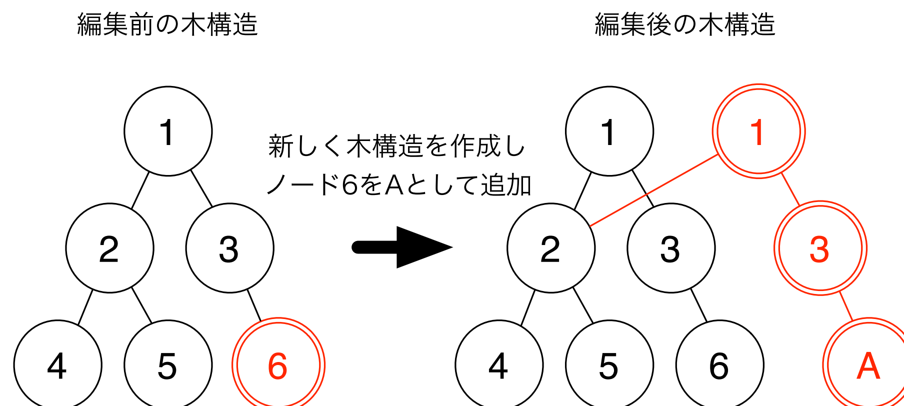


図 3.1: 木構造の非破壊的編集

非破壊的木構造の基本的な戦略は、変更したいノードへのルートノードからのパスを全てコピーする。そして、パス上に存在しない (編集に関係のない) ノードはコピー元の木構造と共有することである。

編集は以下の手順で行われる。

1. 変更したいノードまでのパスを求める (図 3.2)。
2. 変更したいノードをコピーし、コピーしたノードの内容を変更した新しいノードを作成する (図 3.3)。
3. 求めたパス上に存在するノードをルートノードに向かってコピーする。 コピーしたノードに一つ前にコピーしたノードを子供として追加した新しいノードを作成する (図 3.4)。
4. 影響のないノードをコピー元の木構造と共有する (図 3.5)。

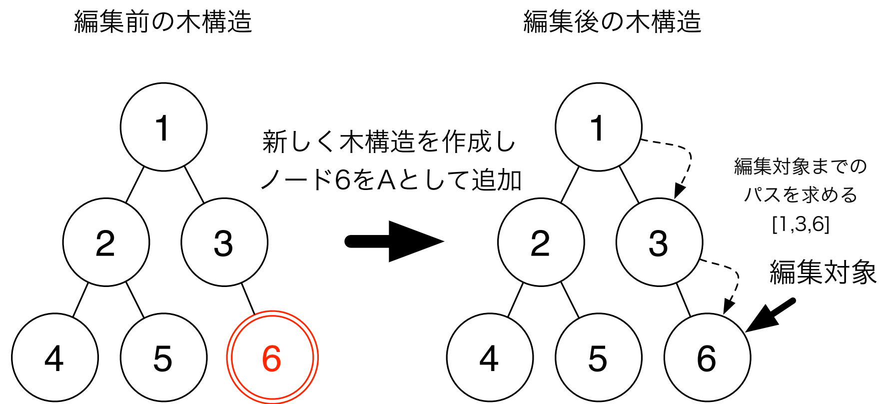


図 3.2: ステップ 1：変更したいノードまでのパスを求める

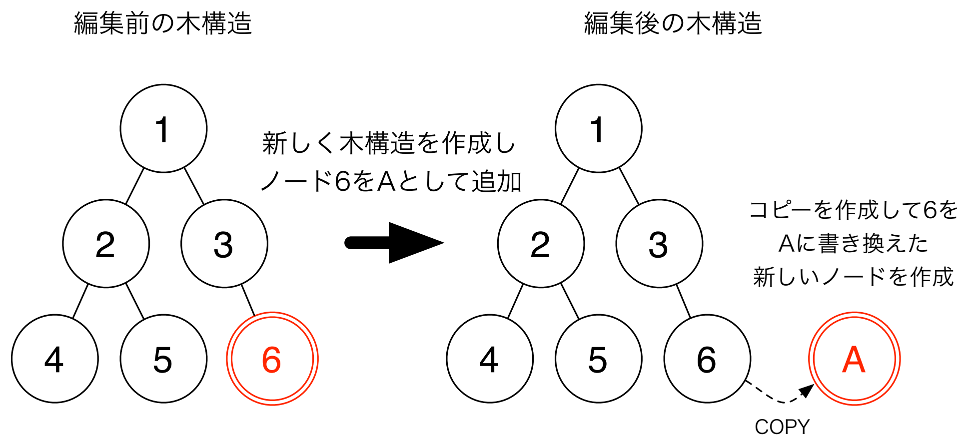


図 3.3: ステップ 2：変更したいノードをコピーし、コピーしたノードの内容を変更した新しいノードを作成する

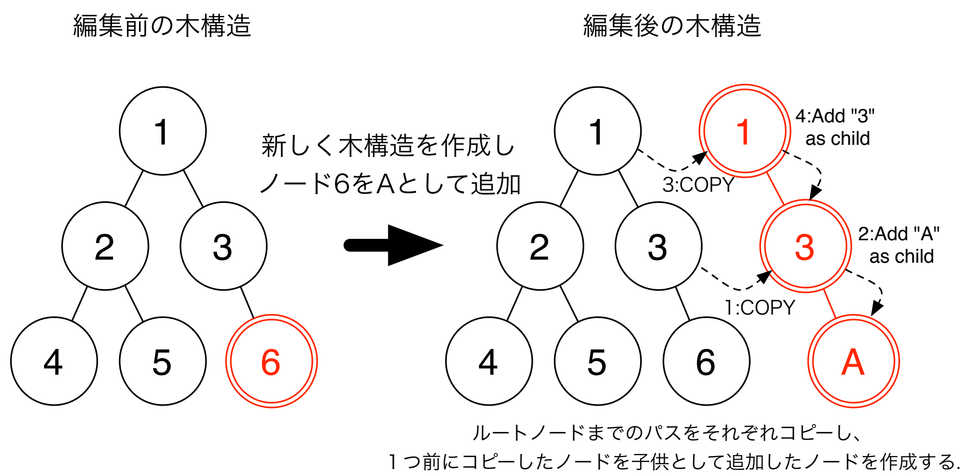


図 3.4: ステップ 3：求めたパス上に存在するノードをルートノードまでコピーする

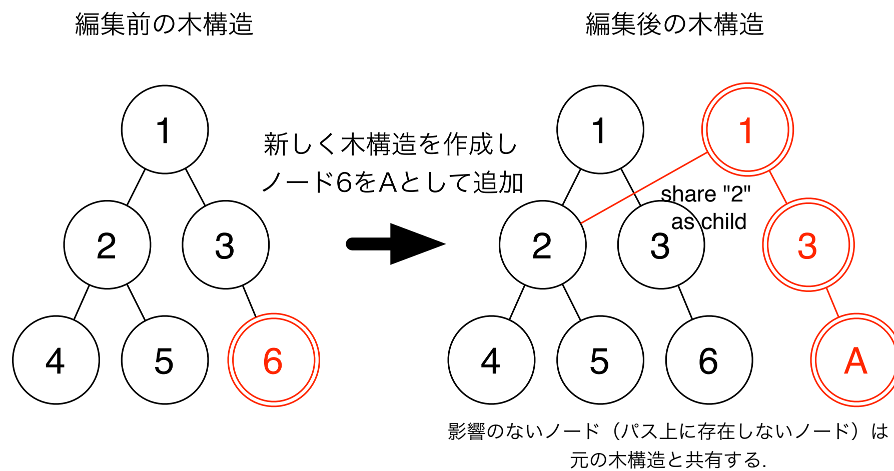


図 3.5: ステップ 4 : 影響のないノードは共有する

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。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。

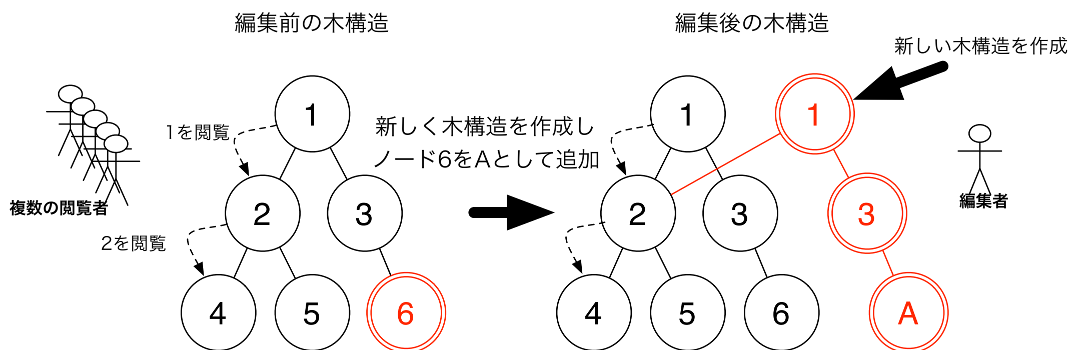


図 3.6: 並列に読み書きが可能な非破壊的木構造

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、その後は排他制御なしに木構造へアクセスできる（図 3.7)。

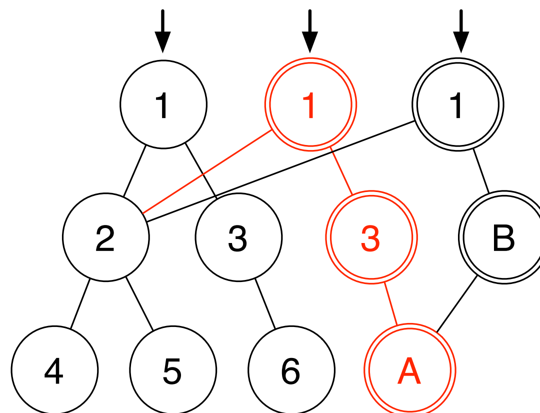


図 3.7: 排他制御なしの非破壊的木構造のアクセス

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 モナドを用いる必要がある。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 は排他制御を行わず、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 は排他制御を行わないため、簡単に扱うことができる。ルートノードの情報の取得だけならば、並列に取得できる。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 [5]。しかしながら、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。まず、実装した非破壊的木構造データベースの利用方法について述べ、次に詳細な設計と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 API 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（Jungle は複数の木を保持できる）
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 内部のデータ型

Jungle の内部のデータ型を表 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。Tree は外部から見えないように実装されている。

Jungle は複数の Tree の集まりである。Jungle と木構造の名前を利用して最新のルートノードを取得することができる。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、データ内部の整合性が保たれ、また型検査でエラーがないか検出することができる。

表 4.1: 内部のデータ型

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。
children	子となるノードを任意の数持つことができる。
attributes	Key と Value の組み合わせを任意の数持つことができる。

Jungle と木の作成

Jungle は複数の非破壊的木構造を扱うことができる（図 4.1)。

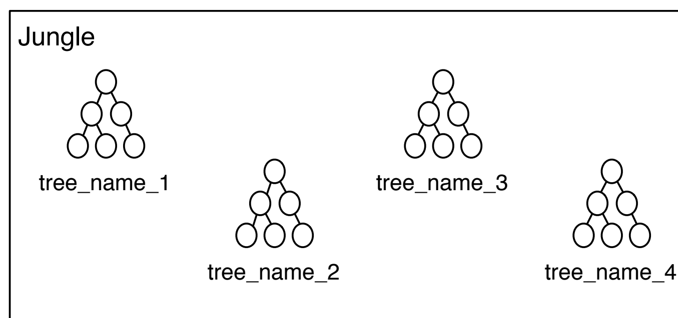


図 4.1: 複数の木を扱える Jungle

木構造の識別には、名前を利用する。名前を付けて作成し、名前を用いることで参照を行う。

createJungle で、Jungle を作成できる。

木を作成するには、createTree を利用する。createTree には、createJungle で作成した Jungle と新しい木の名前を渡す。

型の定義と実際のコードを示す。

```
createJungle :: IO Jungle
createTree :: Jungle -> String -> IO ()

jungle <- createJungle
createTree jungle "name of new tree here"
```

ソースコード 4.1: 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)。

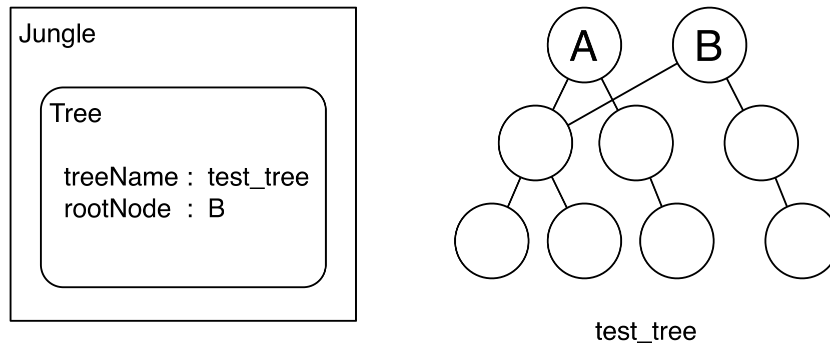


図 4.2: ルートノード

ルートノードに関する API 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 :: Jungle -> String -> IO Node

node <- getRootNode jungle "your tree name here"
```

ソースコード 4.2: 最新のルートノードの取得

木構造を編集する API 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 :: Jungle -> String -> Node -> IO ()

updateRootNode jungle "your tree name here" node
```

ソースコード 4.3: 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。この updateRootNodeWith を利用することで、getRootNode をした後、編集し updateRootNode を行う一連の操作が atomically に行われることが保証される。

```
updateRootNodeWith :: (Node -> Node) -> Jungle -> String -> IO ()

updateRootNodeWith func jungle "your tree name here"
```

ソースコード 4.4: 関数を渡してルートノードの更新

木の編集

木の編集には、Node を使う。木の編集に用いる API 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 API 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。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。

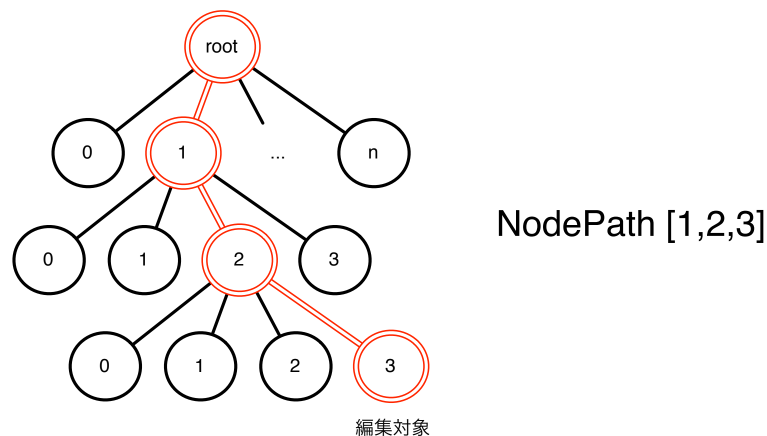


図 4.3: NodePath

addNewChildAt で、ノードに新しい子を追加できる。addNewChildAt には、Node と NodePath を渡す。子には Position という場所の情報があるが、インクリメントしながら自動的に指定される。

```
addNewChildAt :: Node -> Path -> Node

new_node = addNewChildAt node [1,2]
```

ソースコード 4.5: 子の追加

deleteChildAt で、ノードの子を削除できる。deleteChildAt には、Node と NodePath、削除したい子の Position を指定する。

```
deleteChildAt :: Node -> Path -> Position -> Node

new_node = deleteChildAt node [1,2] 0
```

ソースコード 4.6: 子の削除

putAttribute で、ノードに属性を追加できる。putAttribute には、Node と NodePath、Key、Value を渡す。Key は String、 Value は、ByteString である。

```
putAttribute :: Node -> Path -> String -> ByteString -> Node

new_node = putAttribute node [1,2] "key" "value"
```

ソースコード 4.7: 属性の追加

deleteAttribute で、ノードの属性を削除できる。deleteAttribute には、Node と NodePath、Key を渡す。

```
deleteAttribute :: Node -> Path -> String -> Node

new_node = deleteAttribute node [1,2] "key"
```

ソースコード 4.8: 属性の削除

木の参照

木の参照にも Node を用いる。様々な参照の API があるため、ひとつずつ紹介していく。

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

```
getAttributes :: Node -> Path -> String -> Maybe ByteString

bytestring = getAttributes node [1,2] "key"
```

ソースコード 4.9: 属性の取得

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。getChildren は、対象の Node が持つ全ての子を Node のリストとして返す

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

nodelist = getChildren node [1,2]
```

ソースコード 4.10: 対象の Node の全ての子を取得

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。getChildren と違い、子の Position も取得できる。assocsChildren は、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。

```
assocsChildren :: Node -> Path -> [(Int, Node)]
nodelist = assocsChildren node [1,2]
```

ソースコード 4.11: 対象の Node の全ての子と Position を取得

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。assocsAttribute は、対象の Node が持つ全ての属性を、Key と Value のタプルとし、そのタプルのリストを返す。

```
assocsAttribute :: Node -> Path -> [(String, B.ByteString)]
attrlist = assocsAttribute node [1,2]
```

ソースコード 4.12: 対象の Node の全ての Attribute の Key と Value を取得

numOfChild では、対象の Node が持つ子どもの数を取得できる。

```
numOfChild :: Node -> Path -> Int
num = numOfChild node [1,2]
```

ソースコード 4.13: 対象の Node の子どもの数を取得

currentChild では、対象の Node が持つ最新の子を取得できる。

```
currentChild :: Node -> Path -> Maybe Node
node = currentChild node [1,2]
```

ソースコード 4.14: 対象の Node の最新の子を取得

並列実行

木構造データベース Jungle は、並列に実行することができる。アプリケーション側で、データベースを参照や変更する際に各スレッドから呼び出しても問題ない。利用方法も、シングルスレッドで実行する場合と同じである。

4.2　木構造データベース Jungle の実装

開発環境

実装には、Haskell を用いる。コンパイラは、Haskell の並列実行に対応した Glasgow Haskell Compiler (GHC) を用いる。GHC は、Haskell で最も広く使われているコンパイラである [6]。並列実行のための Monad や、スレッドセーフな参照型を提供している。

4.2.1　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、ルートノードの情報も一緒に保持している。

木の取り扱い

Jungle の木の取り扱いには、Haskell の Data.Map を利用している。

Haskell で連想配列を扱いたい場合、平衡木によって実装された Data.Map を一般的に用いる。Haskell のライブラリには配列や、ハッシュ・テーブルといったものも存在するがあまり使われない。配列は参照に適しているが、データを追加する際に配列を再作成するためコストが大きい。ハッシュ・テーブルは更新操作に副作用を伴うため、IO モナドの中でしか使うことが出来ず、扱いにくい。Data.Map は、挿入や参照が O(log n) で済む。

また、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。

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

ソースコード 4.15: Tree のデータ型の定義

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。

4.2.2　Tree

Jungle が保持する木構造は、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。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 4.16: Tree のデータ型の定義

4.2.3 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 としてノードを複数持つことができる（図 4.4)。

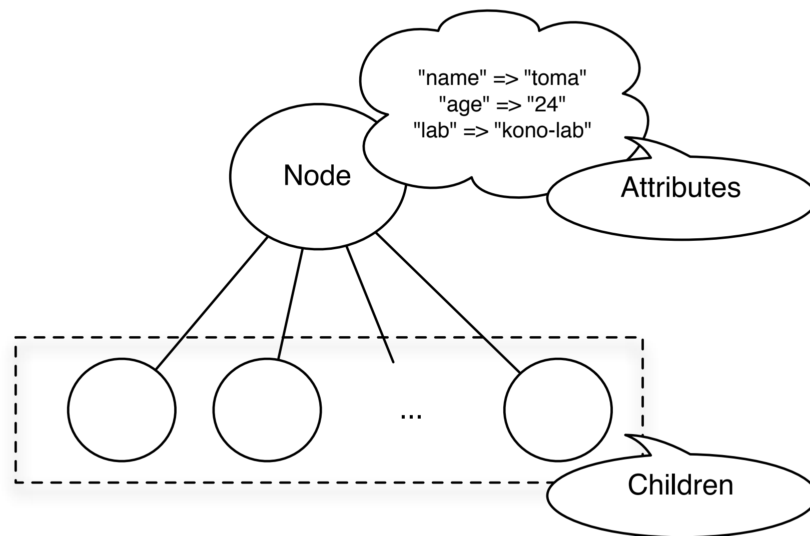


図 4.4: Node の構成要素

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 4.17)。

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

ソースコード 4.17: Node のデータ型の定義

4.3 Haskell の並列処理

純粋関数型言語 Haskell は並列処理に向いていると言われる。しかしながら、安直にそう言い切ることはできない。参照透過性があるため、各々の関数の処理は独立している。そのため、並列で走らせても問題ないように思われるが、Haskell は遅延評価を行うため問題が発生する。遅延評価では、結果が必要になるまで評価されない。実装においては、deepseq を用いて即時評価を挟むか、出力など計算が必要となる処理を挟むようにする。

Haskell では、様々な並列化手法が提供されている。スレッドを直接操作することも可能である。

本研究では、抽象度の高い Eval モナドを利用した。Eval モナドを利用することで、並列処理のために必要な処理の流れを分かりやすく記述することができた。しかしながら Eval モナドは実行順序を細かく制御することはできない。Par モナドを利用すれば、並列処理の流れを細かく記述できるが、Eval モナドのように処理と並列処理の流れを分けて記述し、後からプログラムに並列処理を組み込むというようなことはできない。

Haskell で並列処理を実装する場合は、どの並列化手法を採用するかということをよく考察する必要がある。

4.4 Haskell の生産性

Java を用いた Jungle の実装と比較して、コード行数が約 3000 行から約 300 行へと短くなった。

これは Haskell の表現力が高いためである。Haskell では、独自のデータ型を簡単に作成することができる。再帰的なデータ構造の定義も容易である。共通の性質を扱うための型クラスという仕組みが存在し、既存のライブラリを作成したデータ型に利用できる。また、Haskell は参照透過性を持つため、コードの再利用が行い易く、関数同士の結合も簡単である。

同じような機能を実装する場合でも、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。まずはじめに、並列読み込みと並列書き込みの性能の計測を行う。また、掲示板ウェブアプリケーションを作成し、Java を用いた非破壊的木構造データベースとの性能比較を行う。

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。

表 5.1: 学科が提供するブレードサーバの仕様

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 14

非破壊的木構造データベース Jungle の並列読み込みと並列書き込みの性能の計測には１台、掲示板ウェブアプリケーションを用いた Java との性能比較には２台のブレードサーバを利用する。２台使用するのは、サーバと負荷をかけるクライアントを別々に実行するためである。

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。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の Haskell の拡張が行われている。

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

言語	バージョン
Haskell	Glasgow Haskell Compiler, Version 7.6.3
Java	Java(TM) SE Runtime Environment (build 1.7.0_51-b13)

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。まず、Jungle を利用して木構造を作成する。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。

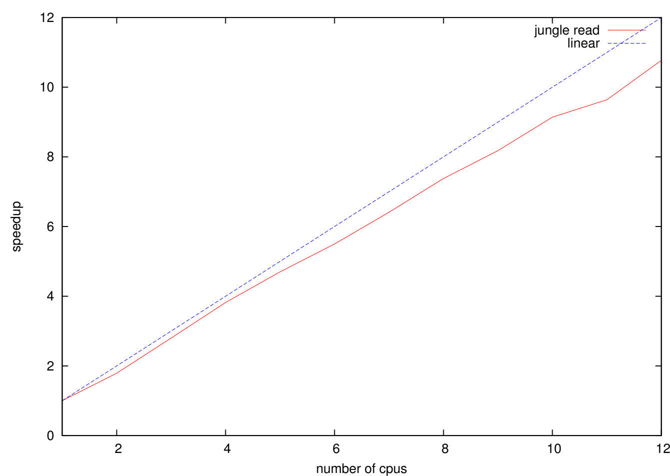
計測結果

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 1.79 倍、12 スレッドで 10.77 倍の性能向上が見られる。13 スレッド以上は、Intel のハイパースレッディング機能を利用して計測した。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る。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。実際の計測では、13 スレッド以上は、12 スレッドより速くなることもあるが、遅くなる場合もあるなど安定しない結果となっている。
性能向上率のグラフを図 5.1 に示す。線形に近い形で性能が向上していることが分かる。12 スレッドで実行した場合の並列化率は 98.96 %で、非破壊的木構造データベース Jungle は読み込みにおいてスケールするデータベースであることが分かる。

表 5.3: 読み込みの計測結果

CPU 数	実行時間
1	59.77 s
2	33.36 s
4	15.63 s
8	8.10 s
12	5.55 s
16	5.65 s
20	5.23 s
24	5.77 s

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 に木構造を並列に書き込んで性能計測を行う。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 1.55 倍、12 スレッドで 3.86 倍の性能向上が見られる。読み込みと比べ、書き込みは Jungle への登録作業があるため並列化率が下がり、性能向上率が低いことが分かる。

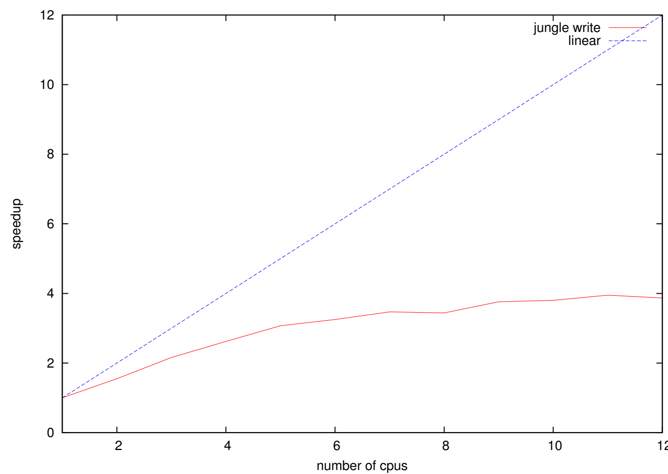
ハイパースレッディングは効果がなく、13 スレッド以上では実行時間が遅くなっている。

表 5.4: 書き込みの計測結果

CPU 数	実行時間
1	52.68 s
2	33.92 s
4	20.11 s
8	15.31 s
12	13.62 s
16	14.92 s
20	18.62 s
24	16.63 s

性能向上率のグラフを図 5.2 に示す。書き込みは並列化率が低く、性能向上が 4 倍程度で止まっている。12 スレッドで実行した場合の並列化率は 80.8 %である。

図 5.2: 書き込みの性能向上率



Jungle へ登録する際に他のスレッドから登録があった場合に、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、並列度が下がっていると思われるが、GHC の問題も考えられる。GHC の IO マネージャーは、マルチスレッドでうまくスケールしないという問題があり、並列度が下がってしまう。GHC の次期バージョンでは IO マネージャーが改善され、スケールするようになる見込みである [7]。

非破壊的木構造データベース Jungle は、書き込みは並列化率が低くなってしまっている。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いえる。

5.4 ウェブアプリケーションに組み込んでの性能評価

実用的なウェブアプリケーションに組み込んで、妥当な性能が出るか調査を行う。

5.4.1 掲示板ウェブアプリケーションの実装

木構造データベース Jungle と Haskell の HTTP サーバ Warp を用いて掲示板ウェブアプリケーションを開発する。Warp を用いたウェブアプリケーションの構築法については付録に記載する。

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。

表 5.5: 掲示板ウェブアプリケーションへ実装した機能一覧

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

5.4.2 読み込み

計測方法

掲示板に対して読み込みを行い、負荷をかける。掲示板を立ち上げるサーバと、性能測定ツール weighttp[8] を用いて負荷をかけるサーバの 2 台ブレードサーバを用いて測定を行った。

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 10、HTTP Keep-Alives を有効とする。

HTTP サーバ 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、12 スレッドまでの計測とする。

計測結果

掲示板の読み込みの計測結果を表 5.6 に示す。

並列で実行した場合、実行時間が短くなっているが性能向上率が低いことが分かる。これは HTTP サーバ Warp がボトルネックとなってしまっているためだと考えられる。

Warp を用いてアクセスした際に、”hello, world” と返すだけのプログラムを作成し測定する。結果を表 5.7 に示す。1 スレッドで実行した場合は、Jungle と組み合わせた掲示板より速い。しかしながら、スレッド数が増えていくと掲示板の読み込みとあまり結果が変わらなくなってしまう。Warp は、HTTP サーバのため IO 処理を多用する。この問題は、GHC の IO マネージャーの改良で改善される可能性が高い。

表 5.6: 掲示板を利用した読み込みの計測結果

CPU 数	実行時間
1	60.72 s
2	37.74 s
4	28.97 s
8	27.73 s
12	28.33 s

表 5.7: Warp の計測結果

CPU 数	実行時間
1	49.28 s
2	35.45 s
4	25.70 s
8	27.90 s
12	29.23 s

ウェブアプリケーションを用いて実験する場合、データベースだけがボトルネックとなるように負荷をかけるのは難しい。ただ単にデータを大きくするだけでは、文字列を HTML に変換するコストが大きくなってしまうためである。

5.4.3 書き込み

計測方法

掲示板に対して書き込みを行い、負荷をかける。掲示板を立ち上げるサーバと、weighttp を用いて負荷をかけるサーバの 2 台ブレードサーバを用いて測定を行った。

weighttp では、GET しかできないため URL のクエリ文字列でデータを書き込めるように掲示板ウェブアプリケーションを変更した。weighttp 起動時のオプションは、読み込みと同じである。

計測結果

掲示板の書き込みの計測結果を表 5.8 に示す。並列で実行した場合、実行時間が短くなっているが性能向上率が低いことが分かる。これも HTTP サーバ Warp がボトルネックとなってしまっているためだと考えられる。

表 5.8: 掲示板を利用した書き込みの計測結果

CPU 数	実行時間
1	54.16 s
2	36.71 s
4	31.74 s
8	31.58 s
10	32.64 s
12	32.68 s

5.4.4 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 版の 2 つ存在する。掲示板ウェブアプリケーション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 版の掲示板ウェブアプリケーションをブレードサーバ上で実行し、weighttttp で負荷をかけ 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 5.9 に示す。

表 5.9: Haskell と Java の比較

測定	Haskell	Java
読み込み	28.33 s	53.13 s
書き込み	32.68 s	76.4 s

Haskell 版は、Java 版と比較して読み込みで 1.87 倍、書き込みで 2.3 倍の性能が出ている。

書き込みが読み込みより性能差が出ている理由として遅延評価が考えられる。Haskell では書き込みを行う際、完全に評価せず thunk として積み上げていく。thunk とは、未評価の式を追跡するのに使われるものである。遅延評価は、グラフ簡約があるため先行評価より簡約ステップ数が増えることはない。また、不要な計算が省かれるので簡約ステップ数が少なくなることもある。しかしながら、計算の所用領域が増えてしまうのが難点で、ハードウェアが効率的に利用できるメモリの範囲内に収まらなければ即時評価より実行時間が遅くなってしまうことがあるので注意が必要である。本実験では、潤沢にメモリを割り当てているためそういった問題は起きない。

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簡潔で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。

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 98.96 %という高い並列化率が確認できた。また、掲示板ウェブアプリケーションを開発し、既存の Java の非破壊的木構造データベースとの比較をおこない、Java のおよそ 2 倍の性能を確認することができた。

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。

書き込み処理の並列度の上昇

データベースへの書き込み処理において 80.8 %の並列度しか出ていない。はじめに、プログラムの変更をせずに、GHC の IO マネージャーの改善によりどの程度並列度が向上するのかを調査する必要がある。GHC の IO マネージャーの影響度を計測した後、変更処理の改善方法に調査する。現在ソフトウェア・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、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

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。簡単な実装としては、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ような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以下のコマンドで取得できる。

```
$ 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、以下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 へ以下の記述を追加する。

```
* 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 に以下の記述を追加する。

```
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以下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ような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 [8]。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、以下の様に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 [9]。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」[10] 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

[10] DEOS. `http://www.dependable-os.net/osddeos/data.html`. [Online; accessed 29-Jan-2014].

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

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