マルチコア時代の最新並列並行技術

Haskellから見える世界

山本和彦
kazu@iij.ad.jp

問

並列(parallel)
並行(concurrent)
とは何か？

答え

共通の定義はない

本発表での並列と並行

並列
parallel

計算を速くする
複数のCPUが必須
結果は決定的
効率化のための技術
例) 数値演算の高速化

並行
concurrent

複数の対話を同時(のよう)に扱う
CPUは一つでもよい
結果は非決定的
構造化のための技術
例) ウェブサーバ

- 本発表はマルチコアのみ。分散は対象外。

並列に取り組んでいる言語

関数型	命令型
Haskell	Java
Scala	その他多数

並行に取り組んでいる言語

関数型	命令型
Haskell 軽量スレッド	Go goroutine(高機能コルーチン)
Erlang 軽量プロセス	Rust タスク(軽量プロセス)

並列と並行の両方に取り組んでいる
Haskell の最新技術を紹介

これまでの並列・並行技術

OS スレッド

ロック

- OS スレッドとロックがあればほどんどの並列・並行処理が書ける
- 低レベル過ぎて、正しく扱うのは難しい

Haskell の並列・並行に関する哲学

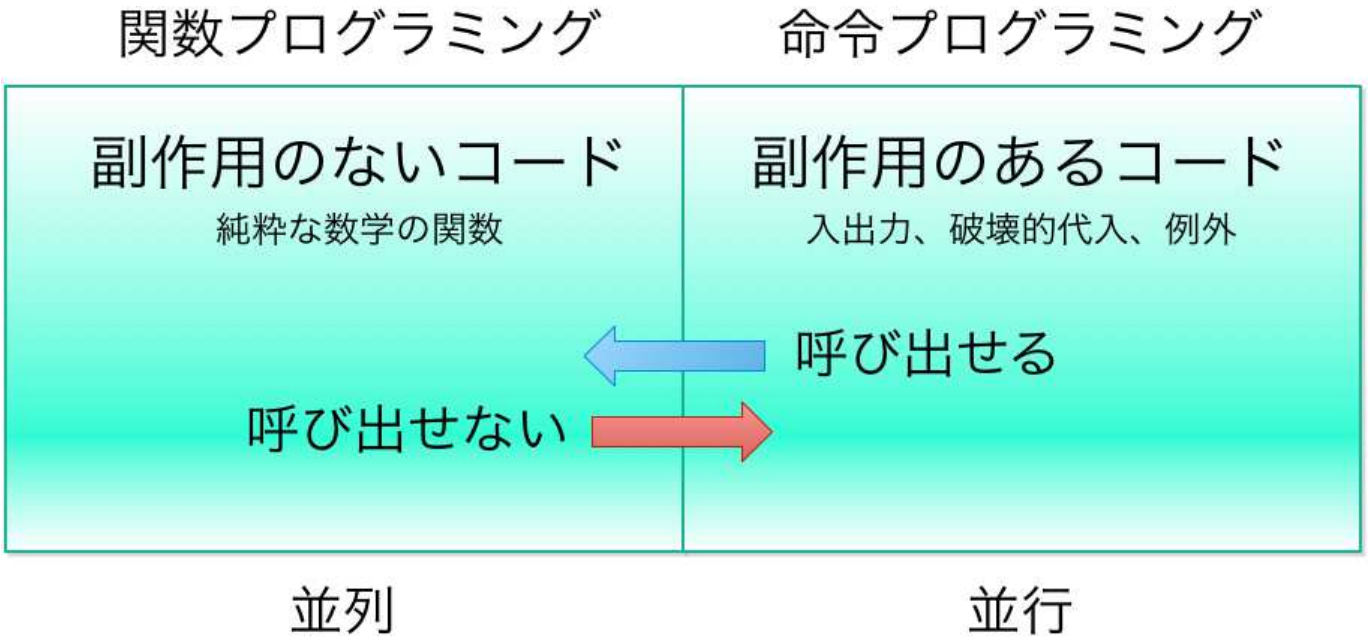
ナイフだけではすべては切れない

Haskell が提供する複数の刃物

並列技術	並行技術
タスク並列	軽量スレッド
データ並列	不変データ
GPU 上のデータ並列	CAS
データフロー並列	STM

Haskell の並列と並行

- Haskell では、型システムの要請により
  関数プログラミングの部分と
  命令プログラミングの部分が明確に分かれる



Haskell の並列技術

Haskell の並列技術

タスク並列

データ並列

GPU 上での
データ並列

データフロー並列

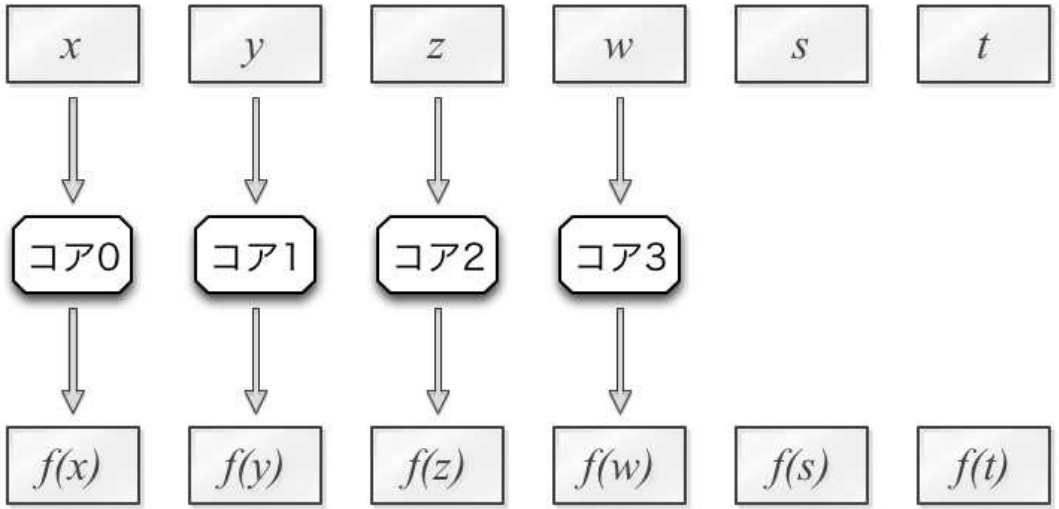
並列モデル

タスク並列	処理を依存関係のないように 分割し並列化する
データ並列	データを依存関係のないように 分割し並列化する
データフロー並列	データの依存関係を記述 並列性は自動的に抽出される

- 例) N枚のカラー画像を白黒に変換する
  - タスク並列：画像ごとにコアを割り当て、画像全体を処理
  - データ並列：分割した画像をコアに割り当てて処理

map によるタスク並列

- 関数型では、map によるタスク並列が基本
  - リストの各要素をそれぞれのコアで処理する



- Google の MapReduce と考え方は同じ

各言語での例

- Java
  - `puzzles.parallel().map(e -> solve(e)); // 自信なし`
- Scala
  - `puzzles.par.map(solve); // 自信なし`
- Haskell
  - `map solve puzzles ‘using‘ parList rseq`
- 注意) オレンジの部分を削除しても結果は同じ
  - 並列は決定的で単に効率を追求している

ランタイムオプション

- コンパイル

```
% ghc -O2 -threaded foo.hs
```

- 実行
  - ランタイムオプションで使用するコア数を指定

```
% ./foo +RTS -N2
```

  - 指定したコア数だけ並列化される
  - 並行処理の場合もこのランタイムオプションを使う

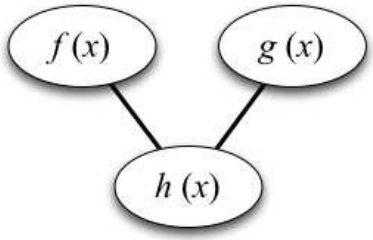
データ並列

- 配列に対するデータ並列ライブラリ Repa
  - ライブラリが提供する関数を使うと
    自動的に並列性が抽出される
- 配列に対するデータ並列ライブラリ Accelerate
  - ライブラリが提供する関数を使うと
    自動的に並列性が抽出される
  - コンパイルすると配列をGPUで処理する
    プログラムが作られる
  - バックエンドは NVIDIA GPU 用の CUDA をサポート
    - デバッグ用にコア自体で動かすこともできる

データフロー並列

- データの依存関係を記述する
  - 並列性は自動的に抽出される

$h(x) = f(x) + g(x)$



並列化可能

```
h x = do {
    fx <- spawn (return (f x));
    gx <- spawn (return (g x));
    a <- get fx;  -- fx を待つ
    b <- get gx;  -- gx を待つ
    return (a+b)  -- 結果を返す
  }
```

注意

直列のコードを単純に並列化
できるとは限らない

必要に応じて並列化できる形に
書き換える必要がある

実践

理論上の限界を知る

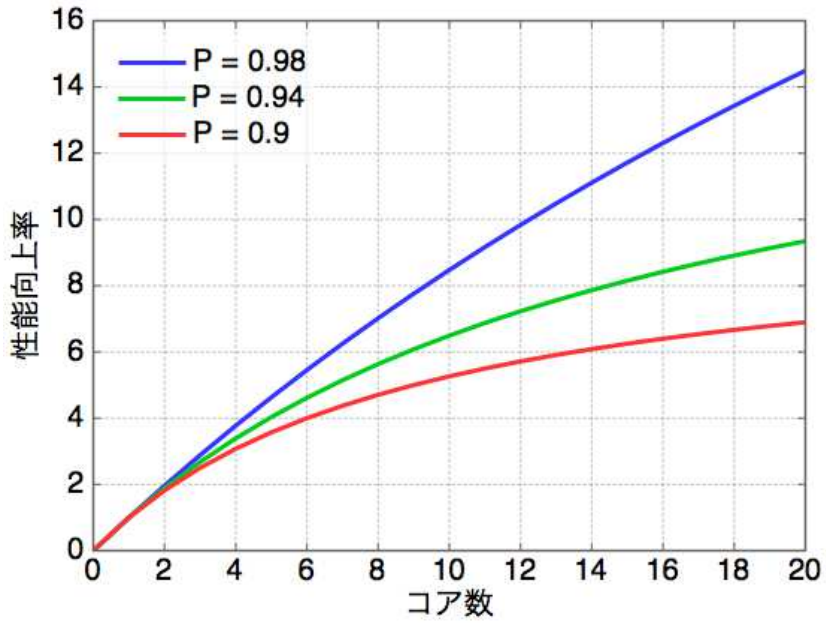
チューニングする

アムダールの法則

$$\text{性能向上率} = \frac{1}{1 - P + \frac{P}{N}}$$

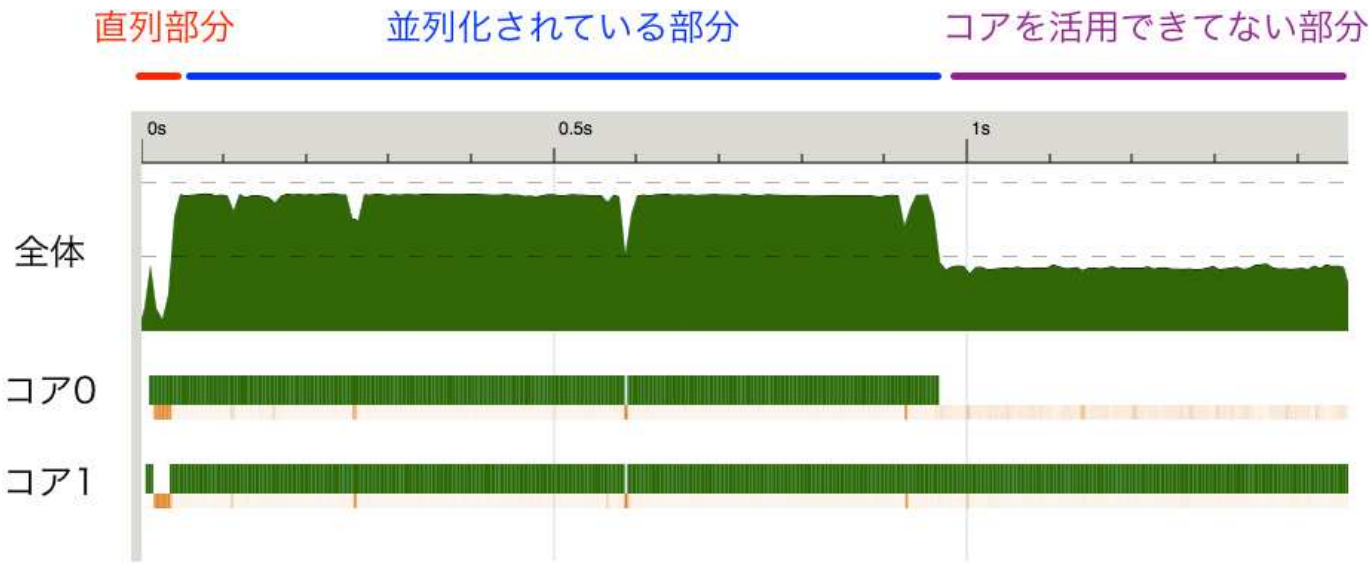
並列可能な部分の実行時間の割合

コア数



Threadscope

- ログを可視化するツール
- 例) タスク並列を2コアで実行したログを表示



- 直列部分はファイルからの読み込み

Haskell の並行技術

Haskellの並行技術

軽量スレッド

永続データ

CAS

STM

ストーリー

1) 複数のクライアントを処理する Web サーバを作る
2) キャッシュを使って Web サーバを高速にする
3) 銀行の送金システムを作る

ストーリー(1)

複数のクライアントを
処理する Web サーバを作る

クライアントが1つの場合

- 見通しのよい一直線のコードが書ける
- 一つのコアしか活用していない

複数のクライアントを扱う技術

OS スレッド

クライアント一つ用と複数用のコードが同じ
一直線のコードを書ける
低速

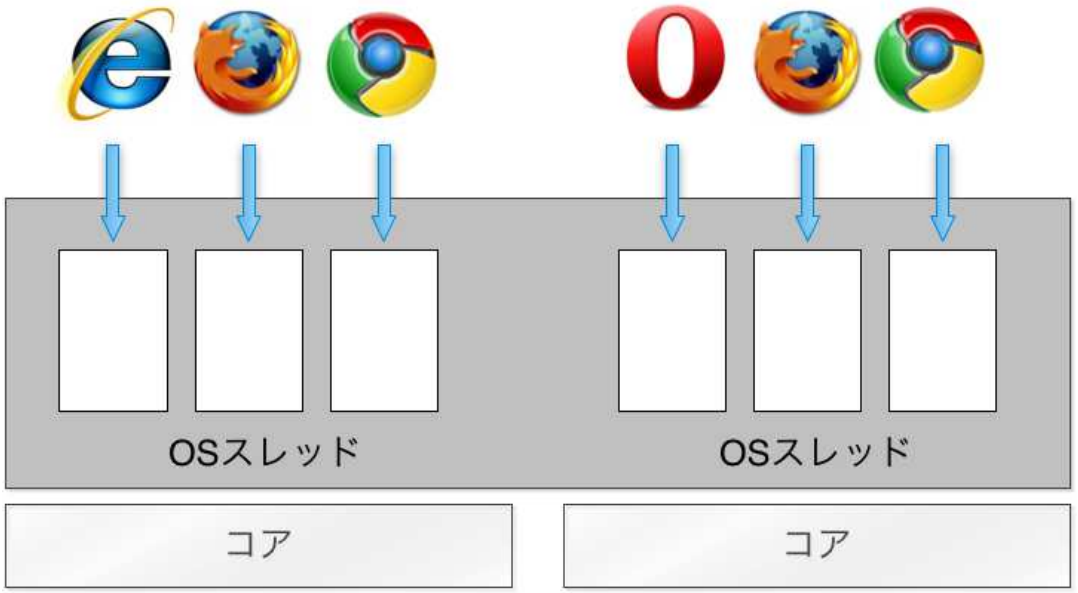
イベント駆動

クライアント一つ用と複数用のコードが異なる
コードを分断し状態を管理
高速

軽量スレッド

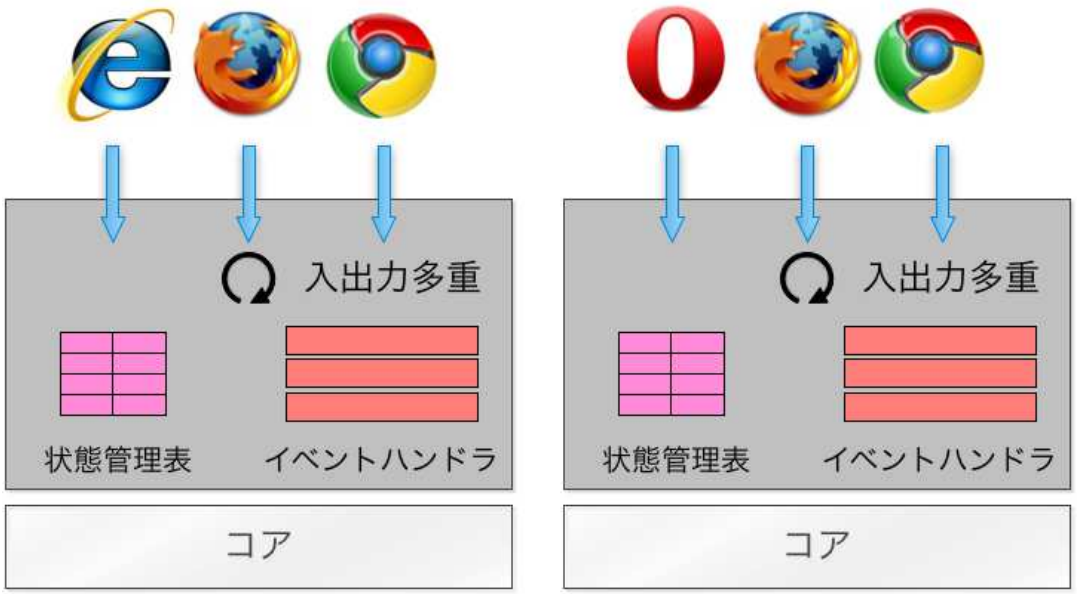
クライアント一つ用と複数用のコードが同じ
一直線のコードを書ける
高速

OSスレッドを使って複数の場合に対応



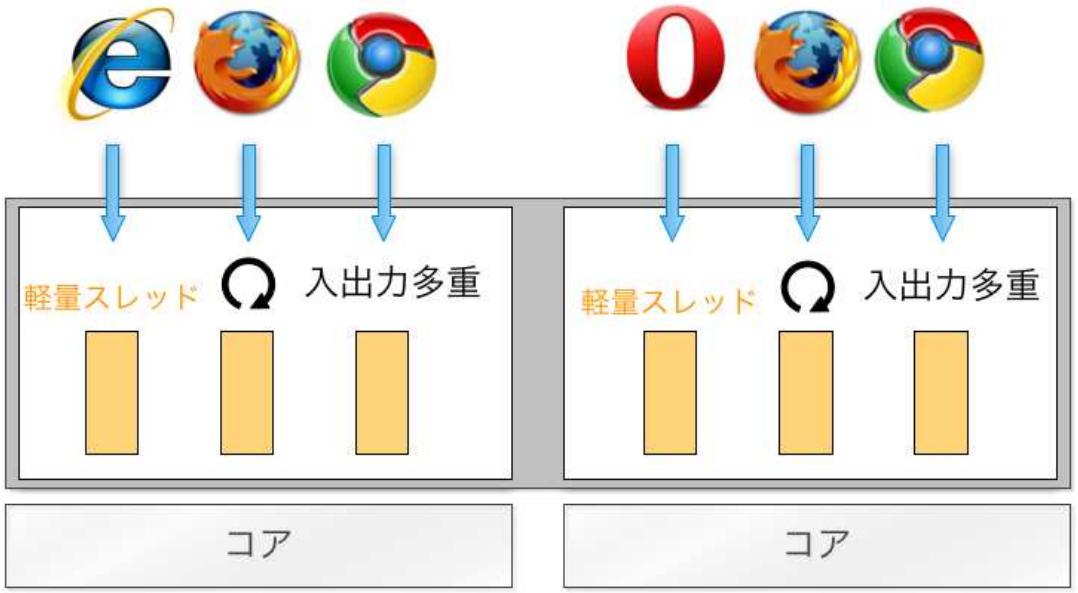
- 利点：見通しのよいコードを再利用できる
  複数のコアを活用できる
- 欠点：OSスレッドの切り替えが多発し遅い

イベント駆動を使って複数の場合に対応



- 利点：コアの性能を引き出せる
  - 複数のコアを利用するには prefork してポートを共有
- 欠点：見通しの悪いコードへ作り直し
  - preforkの欠点：共有メモリがない
    (レート制御などの実現が難しい)

軽量スレッドを使って複数の場合に対応



- 利点：見通しのよいコードを再利用できる
- 利点：コアの性能を引き出せる

ストーリー(2)

キャッシュを使って
Web サーバを高速にする

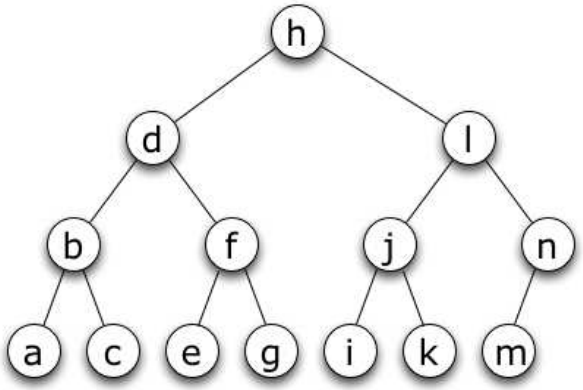
ファイルの open は
カーネル内でロックを取るので遅い

ファイル記述子をキャッシュし
再利用したい

軽量スレッドは共有メモリ中に
キャッシュを持てる

辞書と探索木

- 探索木を高速な辞書として利用
  - 通り抜け順(in-order)で走査するとソートされている木
  - 平衡していれば、挿入、検索、削除は O(log N)

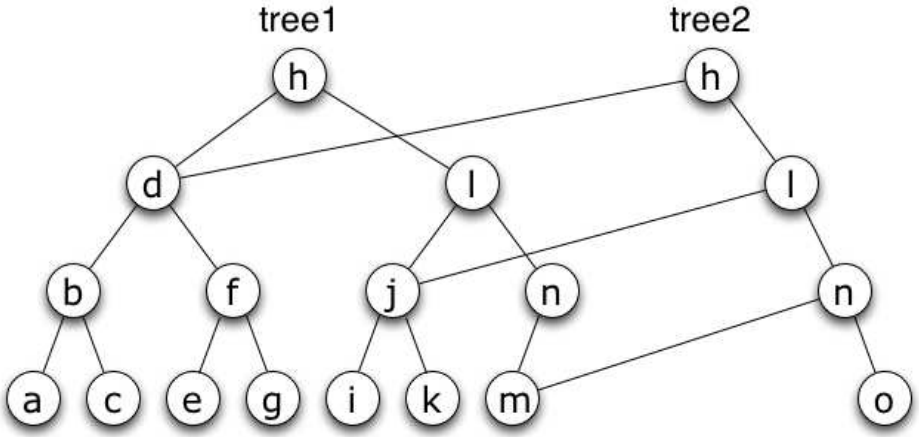


- ファイル記述子のキャッシュ
  - ファイル名がキー、ファイル記述子が値

不変データ

- 不変データはスレッドセーフ
  - 操作にロックは不要

```
let tree2 = insert 'o' tree1
```

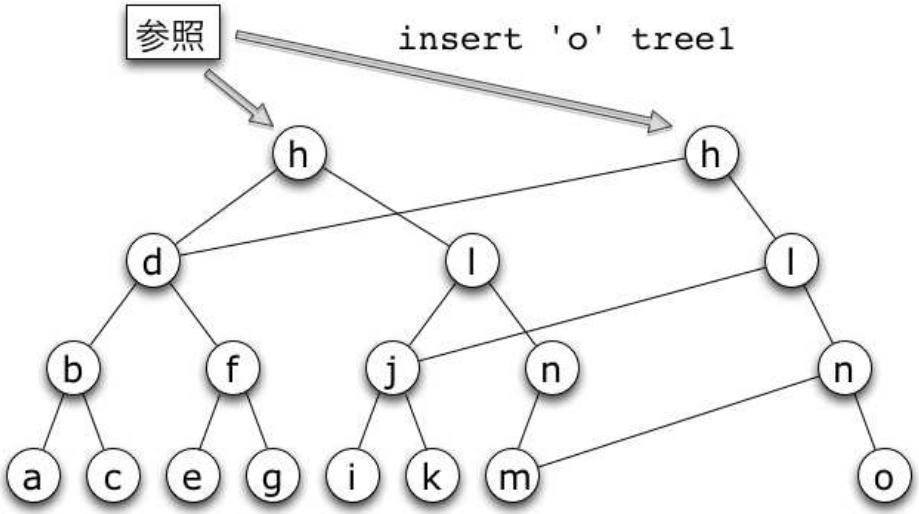


Haskell では、主に不変データが使われるから
軽量スレッドを実現できたとも言える

可変データ

- 不変データは参照で指すと可変の状態を表現できる
  - 参照が可変

```
modifyIORef ref (\tree1 -> insert 'o' tree1)
```

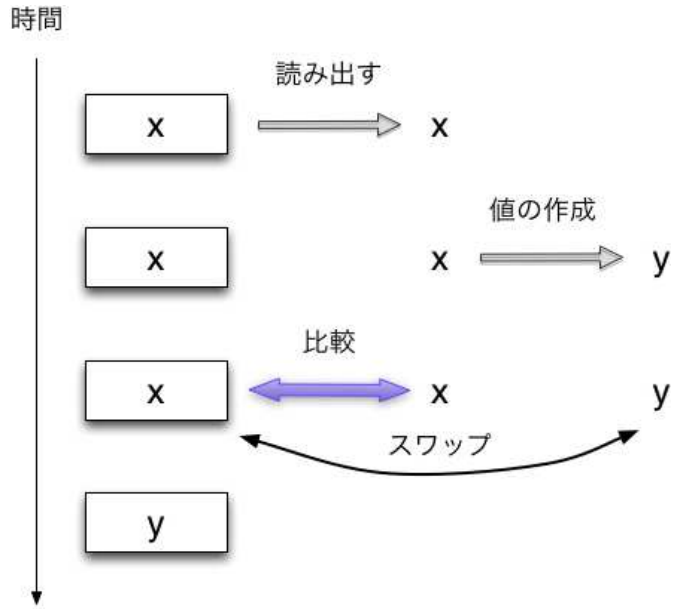


- 可変の参照は競合ポイント
  - 不可分に変更する必要がある
  - 通常はロックを使う

ロックの数を低減するのが
目的だからロックは使えない！

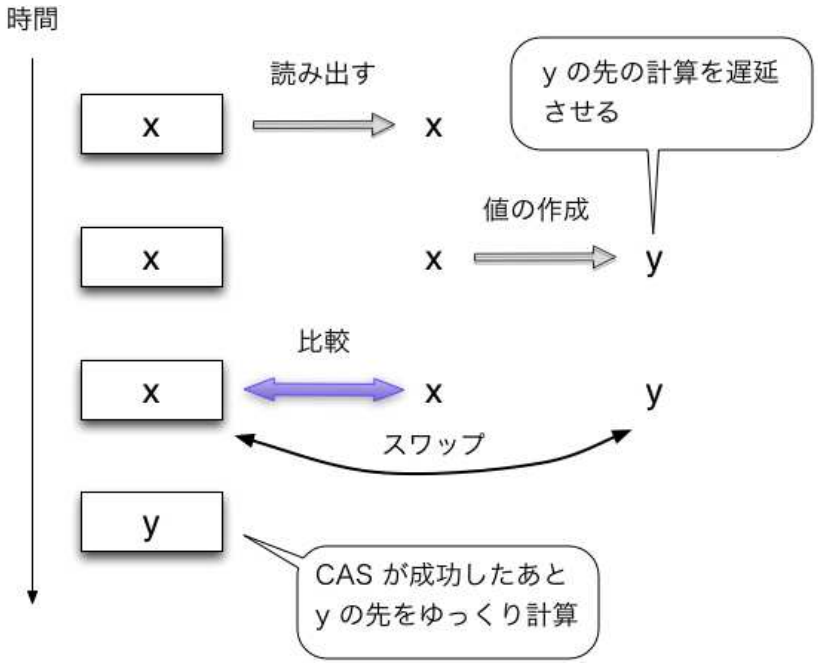
CAS (Compare and Swap)

- 同期を取る必要がないなら CAS が使える
- CAS は CPU の不可分な命令
  - ある「メモリの値」と「指定された値」を比較し、同じであれば「指定された別の値」で「メモリの値」を置き換える



遅延評価と CAS

- 新しい値の作成を遅延させる
  - 「後でこれをやる」という命令書を O(1) で作成
  - 命令書の作成は高速なので、CAS が高い確率で成功する
  - CAS が成功した後に、命令書を実行



ストーリー(3)

銀行の送金システムを作る

ロックを使った送金システム

- 口座(Account)の整合性を保つようにロックを使う

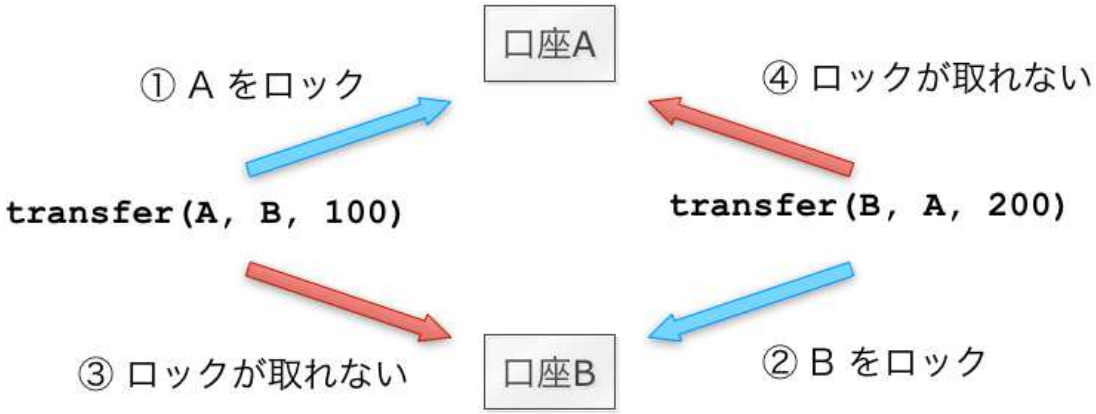
```
void
transfer(Account from,
         Account to,
         Int amount) {
  lock(from); lock(to);
  withdraw(from, amount);
  deposit(to, amount);
  unlock(from); unlock(to);
}
```

ロックの問題点

粒度	小さいと必要なところのロックが漏れがち 大きいと速度が出ない
デッドロック	動きが取れずに飢餓状態に陥る ロックを取る順番が重要
ライブロック	動けるけれど飢餓状態に陥る

デッドロック

- 逆向きの transfer が同時に起こると
  デッドロックする場合がある



デッドロックしない送金システム

- ロックに優先順位を付ける

```
void
transfer(Account from,
         Account to,
         Int amount) {
  if (from < to) then {
    lock(from); lock(to);
  } else {
    lock(to); lock(from);
  }
  withdraw(from, amount);
  deposit(to, amount);
  unlock(from); unlock(to);
}
```

- 面倒で、しかも間違えやすい

STM

- ソフトウェア・トランザクショナル・メモリー
- 複数のロックを一つにみせる技術
  - ロックを取る順番は気にしなくてよい
  - 部品プログラミングが可能
- STM の性質
  - デッドロックフリー
  - ライブロックフリーではない
- 比較) ハードウェア・トランザクショナル・メモリー
  - Intel の Haswell が標準提供

STM を使った送金システム

- STM の TVar はロックの代替品

```
withdraw :: Account -> Int -> STM ()
withdraw acc amount = do {
  bal <- readTVar acc;
  writeTVar acc (bal - amount)
}
```

- 順番を気にせず atomocally で包む

```
transfer :: Account -> Account -> Int
         -> IO ()
transfer from to amount =
  atomically $ do {
    withdraw from amount;
    deposit to amount
  }
```

まとめ

並列技術	並行技術
タスク並列	軽量スレッド
データ並列	不変データ
GPU 上のデータ並列	CAS
データフロー並列	STM

宣伝

- Haskell による並列・並行プログラミング
  - オライリー・ジャパン
  - 並列・並行プログラミングでも部品プログラミングを実践する