Параллелизм и Многопоточность

Никита Юрченко

21 января 2017 г.

1 Functional Programming Parallelism

(Конспект книги паралельный и конкуррентный хаскелл товарища Симона Марлоу)
Во многих случаях не делают различия между паралеллизмом и многопоточностью.
Parallelism - разделяем работу меджду несколькими раздельными вычислительными единицами (процессоры, ядра, ГПУ) в надежде что расходы на поддержку параллелизма будут ниже профитов от него
Concurrency - она же многопоточность - исполнение в одной программе нескольких потоков управления. Потоки выполняются одновременно в том смысле что побочные эффекты потоков перемежаются между собой. Количество вычислительных ядер не имеет значения. Используется в задачах ИО (считывание - запись БД и пр). Происходит ли выполнение действительно одновременно - остается вопросом реализации.
В Хаскеле нет и не может быть никакого "Потока управления"(нет побочных эффектов, порядок выполнения не имеет никакого значения)
Единственный профит от Паралелизма - скорость. Многопоточность же позволяет реализовать например асинхронный ввод-вывод.
Детерминистская модель программирования - при каждом запуске получаем одинаковый результат.
Недетерминистская модель программирования - при каждом запуске разлисный результат. Многопоточные программы недетерминированнные тк взаимодействуют с внешними агентами. Недетерминированные программы намного сложнее тестировать.
Лучше всего писать детерминированные паралельные программы. Большиинство процессоров предлагают детерминированный паралеллизм в виде конвейерного исполнения (pipelining) и множественных единиц исполнения.
Старая нерешаемая проблема автоматического паралелизма (см также закон Амдала) заключается в том, что даже в функциональных ЯП компилятор не знает как "разрезать"программу так чтобы затраты на распараллеливание не сводили на нет профит от паралельных вычислений.
Хаскелл дает массу профитов в параллельном программировании. Детерминистская модель программирования означает, что классичесие проблемы параллелизма (race condition, deadlock) впринципе отсутствуют. Благодаря высокоуровневости и декларативности, программист не настраивает напрямую синхронизацию или коммуникацию.Здесь правда есть и минус, характерный, впрочем, для всех хаскелл-программ - тяжело отслеживать низкоуровневые взаимодействия.
Таким образом, главная задача программиста - разделение задачи на малые части, пригодные для параллельного исполнения. Здесь часто встречаются две проблемы - гранулярность(слишком мелкие задачи), и зависимости данных (одна задача зависит от другой, следовательно они должны выполняться последовательно).

1.1 Lazy evaluations, WHNF

Хаскелл на самом деле не ленивый а нестрогий язык, но вот GHC действительно ленивый компилятор.

Дальше идет про ленивые вычисления.
Высичления до требования хранятся в виде тн санков. Интересно, что нет тулы для просмотра этих самых санков. Они представляют из себя ссылочные структуры. когда вычисление сделано, результат замещает наш санк. Дальше идет пример того, что называется WHNF - weak head normal form. в конце функция seq вычисляет z до WHNF, то есть до первого конструктора кортежа. Понять почему такое название можно из примера с мапом.

```
Prelude> import Data.Tuple
Prelude Data.Tuple> let x = 5  + 4:: Int
Prelude Data.Tuple> let y = x  + 4:: Int
Prelude Data.Tuple> let z = (x, y)
Prelude Data.Tuple> :sprint z
z = (_,_)
Prelude Data.Tuple> let z1 = swap z
Prelude Data.Tuple> :sprint z1
z1 = _
Prelude Data.Tuple> seq z1 ()
()
Prelude Data.Tuple> :sprint z1
z1 = (_,_)
Prelude Data.Tuple> seq z ()
()
Prelude Data.Tuple> :sprint z1
z1 = (_,_)
```

1.2 Монада Eval

Монада Eval испльзуется для выражения паралелизма. К ней комплектом идут:

- *чистая* функция runEval :: Eval a -> a
- комбинаторы rseq и rpar, говорящие что функцию нужно выполнять последовательно или паралельно соответственно. Вычисление идет до ВХНФ.

Далее нам показывают три способа просчитать применение функции к двум аргументам паралельно. В первом мы никого не ждем, и сразу завершаем выполнение. Во втором мы ждем только второе вычисление. В третьем ждем оба. Следует заметить, что второй и третий примеры можно подсократить, подставив rseq напрямую вместо соответствующего rpar.

```
v1 = do
     a <- rpar (f x)
     b <- rpar (f y)
     return (a,b)

v2 = do
     a <- rpar (f x)
     b <- rpar (f y)
```

```
    rseq b                                  9
    return (a,b)                            10
                                            11
v3 = do                                     12
    a <- rpar (f x)                         13
    b <- rpar (f y)                         14
    rseq a                                  15
    rseq b                                  16
    return (a,b)                            17
```

Дальше идет простой пример с судоку, список проблем режут напополам и применяют стратегию 3. В результате вторая половина выполняется заметно быстрее первой.Мораль: не надо разделять проблему на небольшое число кусков фиксированной длины. В качестве решения предлагается функция parMap, которая создает 1000 тн спарков. В конце ГХЦ говорит нам, сколько спарков:

- конвертированы - те правильно распаралелены
- переполнилили пул спарков
- "не выстрелили просчитали уже известное вычисление
- подобраны сборщиком мусора
- "сошли с дистанции ???

Здесь еще важно отметить, что чтение из файла лениво. То есть, после того как первый процессор считал первую проблему, она сразу же отправляется второму процессору на параллельную обработку. Мы можем заставить выполнять считываение целиком до обработки. В таком случае на диаграмме тредскоп можно будет четко увидеть, с какого момента начинается параллельное выполнение. Просчитав долю паралельного рантайма в общем (p = (общий - последовательный) / общий), применим закон Амдала и просчитаем максимальный коеффициент ускорения: 1 / ((1 - P) + P/N), где H - количество ядер. Таким образом, например, усорение в 46 раз теоретически недостижимо, независимо от количества вычислительных ядер.

1.3 Полная нормальная форма

Полная нормальная форма не содержит невычисленных выражений (редексов?). Для этого есть функция force и класс типов NFData. Для основных типов он уже определен. В нем нужно переопределять одну единственную функцию - rnf (reduce to normal form). Модуль Control.Seq из пакета parallel позволяет вычислять выражения до определенных степеней (может быть впринципе полезно).

Стратегии вычисления - способы преобразования алгоритма в параллелизуемый код. Иногда требуется переписать алгоритм.

```
type Strategy a = a -> Eval a                1
```

Перепишем первую версию того кода сверху в терминах Стратегии Вычисления. Главная разница тут в том, что здесь мы имеем дело с типами в первую очередь. Для удобства предлагается оператор using.

```
parPair :: Strategy (a,b)                                    1
parPair (a,b) = do                                           2
 a' <- rpar a                                                3
 b' <- rpar b                                                4
 return (a',b')                                              5
                                                             6
using :: a -> Strategy a -> a                                7
x `using` s = runEval (s x)                                  8
                                                             9
(fib 35, fib 36) `using` parPair                             10
```

Для достижения большей гибкости предлагается не писать каждый раз стратегию "с нуля а воспользоваться параметрической стратегией.

```
evalPair :: Strategy a -> Strategy b -> Strategy (a,b)       1
evalPair sa sb (a,b) = do                                    2
 a' <- sa a                                                  3
 b' <- sb b                                                  4
 return (a',b')                                              5
                                                             6
parPair :: Strategy (a,b)                                    7
parPair = evalPair rpar rpar                                 8
```

Напишем функцию параллельного отображения в терминах стратегии. Здесь у нас две части - алгоритм, функция отображения map, и то что касается паралеллизма - факт, что мы вычисляем каждый элемент независимо.

```
parMap :: (a -> b) -> [a] -> [b]                             1
parMap f xs = map f xs `using` parList rseq                  2
                                                             3
evalList :: Strategy a -> Strategy [a]                       4
evalList strat [] = return []                                5
evalList strat (x:xs) = do                                   6
 x' <- strat x                                               7
 xs' <- evalList strat xs                                    8
 return (x':xs')                                             9
                                                             10
parList :: Strategy a -> Strategy [a]                        11
parList strat = evalList (rparWith strat)                    12
```

Отметим, что evalList и parList уже содержатся в Control.Parallel.Strategies