

Параллелизм и Многопоточность

Никита Юрченко

19 января 2017 г.

1 Functional Programming Parallelism

(Конспект книги параллельный и конкурентный хаскелл товарища Симона Марлоу)

Во многих случаях не делают различия между параллелизмом и многопоточностью.

Parallelism - разделяем работу между несколькими отдельными 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 разный результат. Многопоточные программы недетерминированные так взаимодействуют с внешними агентами. Недетерминированные программы намного сложнее тестировать.

Лучше всего писать детерминированные параллельные программы. Большинство процессоров предлагают детерминированный параллелизм в виде конвейерного исполнения (pipelining) и множественных единиц исполнения.

Старая нерешаемая проблема автоматического параллелизма (см также закон Амдала) заключается в том, что даже в функциональных ЯП компилятор не знает как "разрезать" программу так чтобы затраты на распаралеливание не сводили на нет профит от параллельных вычислений.

Хаскелл дает массу профитов в параллельном программировании. Детерминистская модель программирования означает, что классические проблемы параллелизма (race condition, deadlock) в принципе 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.

Вычисления 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 замечает 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 1
Prelude Data.Tuple> let x = 5 + 4 :: Int 2
Prelude Data.Tuple> let y = x + 4 :: Int 3
Prelude Data.Tuple> let z = (x, y) 4
Prelude Data.Tuple> :sprint z 5
z = (_,_) 6
Prelude Data.Tuple> let z1 = swap z 7
Prelude Data.Tuple> :sprint z1 8
z1 = _ 9
```

```

Prelude Data.Tuple> seq z1 ()      10
()                                  11
Prelude Data.Tuple> :sprint z1     12
z1 = (_,_)                         13
Prelude Data.Tuple> seq z ()      14
()                                  15
Prelude Data.Tuple> :sprint z1     16
z1 = (_,_)                         17

```

1.2 Монада Eval

Монада Eval используется для выражения паралелизма. К ней комплек-
том идут:

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

Далее нам показывают три способа просчитать применение функции
к двум аргументам параллельно. 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                                1
    a <- rpar (f x)                    2
    b <- rpar (f y)                    3
    return (a,b)                       4
                                         5
v2 = do                                6
    a <- rpar (f x)                    7
    b <- rpar (f y)                    8
    rseq b                             9
    return (a,b)                      10
                                         11
v3 = do                                12
    a <- rpar (f x)                    13
    b <- rpar (f y)                    14
    rseq a                             15

```

<code>rseq b</code>	16
<code>return (a, b)</code>	17

Дальше идет простой пример с sudoku, список проблем режут наполам 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
- переполнили 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 считывание целиком до обработки. В таком случае на диаграмме `treemap` можно будет четко увидеть, с какого момента начинается параллельное выполнение. Просчитав долю параллельного рантайма в общем ($p = (\text{общий} - \text{последовательный}) / \text{общий}$), применим закон Амдала и просчитаем максимальный коэффициент ускорения: $1 / ((1 - P) + P/N)$, где N - количество ядер. Таким образом, например, ускорение 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 в принципе полезно).