Использование нескольких ядер

8

Одним из основных преимуществ чистоты, присущей языку Haskell, является возможность беспрепятственного выполнения кода в параллельном режиме. Отсутствие побочных эффектов означает, что все зависимости от данных обозначены в коде явным образом. Соответственно компилятор (как и вы сами) может планировать параллельное выполнение различных заданий, не имеющих между собой зависимостей.

Явно выразить те части вашего кода, которые могут получить преимущества от выполнения в параллельном режиме, позволяет монада `Par`. Поддерживаемая `Par` модель позволяет создавать код, используя как модель *фьючерсов*, так и подход, называемый *параллелизмом потоков данных*. Затем заботу о выполнении вашего кода с использованием параллельных программных потоков берет на себя диспетчер. Монада `Par` является весьма эффективной абстракцией, поскольку вам не нужно задумываться об управлении созданием и уничтожением программных потоков: достаточно дать библиотеке возможность делать свое дело.

Однако иногда ряд параллельных заданий нуждается в таком способе совместного использования ресурсов, который невозможно выразить с помощью монады `Par`. В магазине машин времени, к примеру, несколько клиентов могут покупать одни и те же товары, а при этом одновременно должно происходить обновление нескольких баз данных. В таких сценариях важно обеспечить правильность одновременного доступа к ресурсам. В качестве инструмента управления этим поведением Haskell предлагает так называемую *программную транзакционную память*, в которой используется идея транзакций, позаимствованная из систем управления базами данных.

И наконец, может потребоваться распределить вычисления между несколькими узлами, находящимися в разных местах сети. Для таких случаев подойдет пакет облачных решений для Haskell (Cloud Haskell). В этом пакете для обмена данными между узлами, в ходе которого можно отправлять и получать как данные, так и код, используется модель акторов (actor model).

Параллелизм, одновременность, распределенность

При применении по отношению к программированию понятий параллелизма, одновременности и распределенности всегда возникает некая путаница. Поэтому прежде всего следует внести ясность в эту терминологию.

В программной модели *одновременности* (concurrency) вычисления выполняются в рамках нескольких, по большей части независимых программных потоков. Система может либо смешивать вычисления, либо запускать их параллельно, но в любом случае возникает иллюзия, что все они выполняются асинхронно. Одним из основных примеров применения одновременности является веб-сервер: одновременно им обслуживаются запросы от многих клиентов, и с программистской точки зрения каждый из запросов не зависит от остальных и обрабатывается асинхронно.

В большинстве случаев таким программным потокам требуется доступ к некоторым общим ресурсам. И здесь нужно добиться того, чтобы одновременный доступ не ввел систему в некое несогласованное состояние. Для этого было разработано множество технологий программирования, включая блокировки, семафоры или каналы. Что касается языка Haskell, то модель *программной транзакционной памяти* (Software Transactional Memory, STM) вводит в код из баз данных понятие атомарных транзакций, что позволяет поддерживать оптимистичную одновременность с возможностью отката.

Что касается *параллелизма* (parallelism), то он относится к способу одновременного выполнения кода на более чем одном ядре компьютера. Для увеличения производительности одновременно выполняемые задания часто запускаются в параллельном режиме. Эта прибавка в скорости может быть также применена к заданиям, которые изначально не предназначались для одновременного выполнения, но чьи зависимости от данных позволяют выполнять части алгоритма независимо друг от друга. Вспомним алгоритм быстрой сортировки: на каждом этапе список делится на две части, каждая из которых сортируется отдельно. В этом случае вместо последовательной может производиться параллельная сортировка двух списков.

Для этого второго варианта Haskell является идеальной площадкой: чистые вычисления не могут мешать друг другу, а их зависимости от данных выражены предельно ясно. Представляемая далее монада `Par` следует именно этому замыслу и допускает параллелизм выполнения заданий.

Во многих случаях причиной путаницы между параллелизмом и одновременностью являются языки, допускающие побочные эффекты. В таких языках, как Java или C, любая часть кода может иметь доступ к глобальному состоянию и вносить в него изменения. Поэтому при любой степени параллелизма приходится уделять внимание доступу к общим ресурсам, что при программировании требует применения специальных приемов, учитывающих одновременность выполнения заданий. В данном контексте оба понятия по-настоящему отделить друг от друга невозможно.

Параллельное программирование обычно ассоциируют с выполнением заданий на разных ядрах (микропроцессорах или графических процессорах) одной и той же компьютерной системы. Однако вычисления могут быть также распределены между разными компьютерами, обменивающимися данными по сети. В этом случае речь идет о *распределенном программировании* (distributed programming). Поскольку все действующие лица в системе не зависят друг от друга, координирование заданий должно производиться не так, как при параллельном программировании в рамках одной системы. Кроме того, взаимодействие по сети накладывает огра-

ничения, связанные с отказами или большими задержками. По совокупности этих причин для распределенного программирования нужны другие технологии.

Среду распределенного программирования в Haskell предоставляет пакет `distributed-process`. Этот пакет является частью проекта облачных решений для Haskell, и на него большое влияние оказал язык программирования Erlang. В частности, для управления взаимодействием различных систем в сети в нем используется модель акторов.

РЕЗЮМЕ

«Одновременность — это о том, чтобы иметь дело одновременно с множеством вещей. А параллелизм — это о том, чтобы делать одновременно множество вещей». — Роб Пайк (Rob Pike). Как говорится, почувствуйте разницу.

Площадка для параллельного, одновременного и распределенного программирования в Haskell намного шире, чем показано в данной главе. Рассматриваемые здесь библиотеки могут использоваться множеством иных способов, к тому же в Haskell имеется множество других пакетов. Для параллельного программирования используется пакет `parallel`, инструменты которого реализуют стратегический подход. Параллелизм относится не только к процессорам: ускоритель формирует код, запускаемый на графическом процессоре. Существующей в Haskell пакет `base` предлагает в модуле `Control.Concurrent` низкоуровневую функциональность для реализации концепции одновременности, включая изменяемые адреса памяти (`MVar`) и семафоры.

Монада Par

Первым в этой главе мы рассмотрим пакет параллельного программирования `monad-par`. Функции в этой библиотеке связаны с монадой `Par` и используют переменные `IVar` для хранения результатов взаимодействия. Как мы вскоре увидим, с помощью этого пакета вычисления можно моделировать двумя путями: в виде фьючерсов или в виде потоков данных.

Фьючерсы

Начнем с очень простой задачи, раскладывающей несколько чисел на простые множители. Алгоритм факторизации одного числа весьма прост: мы пытаемся его разделить на возрастающие натуральные числа. Если в какой-то момент от деления получится нулевой остаток, значит, число является простым множителем. Таким образом, исходное число может быть поделено на этот простой множитель, и процесс может начаться сначала. В какой-то момент вы получите то же число, с которого начали, и это будет означать, что достигнут последний простой множитель. Haskell-код, реализующий этот подход, выглядит так:

```
findFactors :: Integer -> [Integer]
findFactors 1 = [1]
```

```
findFactors n = let oneFactor = findFactor n 2
                in oneFactor : (findFactors $ n `div` oneFactor)

findFactor :: Integer -> Integer -> Integer
findFactor n m | n == m         = n
               | n `mod` m == 0 = m
               | otherwise      = findFactor n (m + 1)
```

В каком-то месте программы вам будет предложено разложить на множители два разных числа. Код такой функции пишется просто:

```
findTwoFactors :: Integer -> Integer -> ([Integer],[Integer])
findTwoFactors x y = (findFactors x, findFactors y)
```

Однако эффективность этой функции не слишком высока. Даже если доступен не один, а несколько процессоров, вычисление простых множителей `x` и `y` будет производиться последовательно (допустим, что полное вычисление каждого выполняется за одно и то же время). Нужно, как показано на рис. 8.1, поставить себе цель вычислять `findFactors x` в то же самое время, когда вычисляется `findFactors y`.

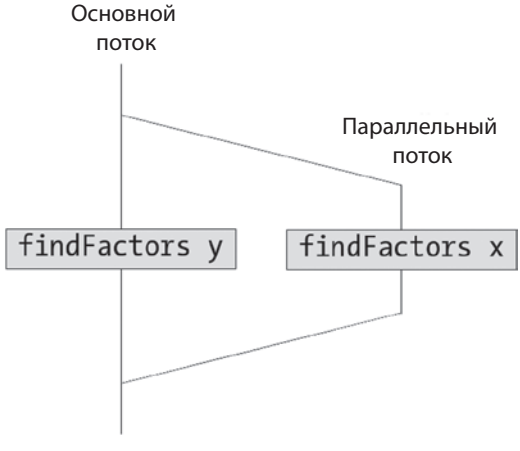

Рис. 8.1. Параллельные вычисления двух разложений на простые множители

С помощью пакета `monad-par` системе можно просто приказать выполнять оба вызова `findFactors` параллельно. В модуле `Control.Monad.Par` есть функция `spawnP`. Давайте присмотримся к ее типу:

```
spawnP :: NFData a => a -> Par (IVar a)
```

Функция `spawnP` предназначена для выполнения вычислений параллельно со всей остальной программой. В этой сигнатуре следует отметить три особенности. Прежде всего, она требует, чтобы тип выполняемых вычислений поддерживался классом типов `NFData`. Если помните, этот тип из пакета `deepseq` гарантирует, что вычисления выполнятся полностью. Функция `spawnP` накладывает это ограничение,

поскольку существует только один способ реально обеспечить параллельное выполнение кода. Если бы этого ограничения здесь не было, то из-за модели ленивых вычислений он был бы выполнен в любое другое время, что привело бы к потере преимуществ параллелизма. Поскольку функция `spawnP` полностью определяет, когда будут выполнены те или иные вычисления, параллельная модель `monad-par` называется *детерминированной*.

Вторая особенность заключается в том, что результат заключен в оболочку типа `Par`. Этот тип является монадой, которая поддерживает параллелизм. И наконец, вместо простого значения результатом выполнения функции `spawnP` является фьючерс `IVar`. *Фьючерс* (future) — это обещание того, что результат вычислений будет доступен по запросу. Для получения результата вычислений внутри `IVar` нужно вызвать функцию `get`. Эта функция тут же возвращает управление, если вычисления завершены, или блокирует выполнение до тех пор, пока не будет доступен результат. Точно такая же модель используется в Scala и в библиотеке параллельных вычислений языка C#.

Вы можете вызывать функции `spawnP` и `get` только внутри монады `Par`. С целью выполнения всех заданий функцию `runPar` нужно вызывать с полной трассой параллелизма. Версия `findTwoFactors`, создающая параллельное задание для вычисления разложения на множители `x` при сохранении факторизации `y` в текущем программном потоке, могла бы выглядеть так:

```
import Control.DeepSeq
import Control.Monad.Par

findTwoFactors :: Integer -> Integer -> ([Integer],[Integer])
findTwoFactors x y = runPar $ do
  factorsXVar <- spawnP $ findFactors x
  let factorsY = findFactors y
      _        = rnf factorsY
  factorsX <- get factorsXVar
  return (factorsX, factorsY)
```

Обратите внимание на вызов функции `rnf` из библиотеки `deepseq` для полного вычисления факторизации `y`.

Следование этим шагам не означает непосредственного создания параллельных заданий — для этого требуется сделать еще два шага. Сначала нужно скомпилировать программу как *многопоточную* (threaded run-time), что позволит GHC создать код, ориентированный на использование нескольких программных потоков. Чтобы добиться этого, добавьте в Cabal-файл флаг `-threaded`:

```
executable chapter8
  hs-source-dirs:  src
  main-is:         Main.hs
  build-depends:   base >= 4, monad-par, deepseq
  ghc-options:     -Wall -threaded
```

Кроме того, вам следует передать своей программе параметры, разрешающие использование нескольких ядер:

```
$ ./dist/build/chapter8/chapter8 +RTS -N2
```

Ключ `+RTS` показывает начало действия ключей, переданных Haskell-системе времени выполнения. В частности, ключ `-N2` указывает на то, что должны быть задействованы два процессора. Можно указать то количество процессоров, которое не превышает число процессоров, имеющихся в системе. Если хотите, можете указать ключ `-N` как таковой (без числа), позволив тем самым Haskell-системе времени выполнения самой принять решение о том, каким количеством процессоров следует воспользоваться.

Параллелизм и переменные IVar

Пакет `monad-par` предоставляет не только фьючерсы, но и обширную модель объявления параллельных вычислений. Вместо простого порождения параллельных вычислений вы указываете несколько этапов вычислений, которые совместно используют промежуточные результаты посредством переменных `IVar`. Эти переменные создаются с помощью ключевого слова `new`. Задания могут записываться в `IVar` с помощью функции `put`, а результат можно получить с помощью функции `get`. Следует заметить, что `IVar` является однократно записываемой переменной.

Рассмотрим пример создания письма для клиента с его счетом на товар. Это подразумевает выполнение четырех различных заданий: одно из них будет заключаться в поиске клиентской информации в базе данных, второе — в том же самом, но в отношении товара. Каждое из этих заданий будет контактировать с другими заданиями через соответствующие переменные `IVar`. Два других задания должны брать эту информацию и генерировать соответственно текст для письма и для конверта. Эта схема показана на рис. 8.2.

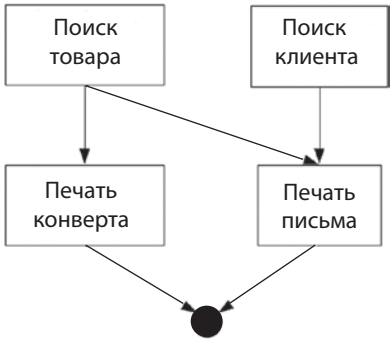

Рис. 8.2. Схема потоков данных при создании письма

Вычисления, формируемые таким способом, всегда следуют показанной схеме, где задания взаимодействуют посредством переменных `IVar`. Поэтому эта модель называется *программированием потоков данных* (dataflow programming). Основным преимуществом такого подхода является то, что все зависимости от данных выражены явным образом — через переменные `IVar`. Эту информацию библиотека `monad-par` использует для планирования параллельного выполнения заданий.

Схема потоков данных, показанная на рис. 8.2, реализована в следующем коде. Обратите внимание, что при использовании этой модели и переменных `IVar` вместо

функции spawnP применяется функция ветвления fork, которая ожидает вычислений типа Par ():

```
printTicket :: Int -> Int -> [(Int,String)] -> [(Int,String)] -> String
printTicket idC idP clients products = runPar $ do
  clientV  <- new
  productV <- new
  fork $ lookupPar clientV  idC clients
  fork $ lookupPar productV idP products
  envV    <- new
  letterV <- new
  fork $ printEnvelope clientV envV
  fork $ printLetter   clientV productV letterV
  envS    <- get envV
  letterS <- get letterV
  return $ envS ++ "\n\n" ++ letterS

lookupPar :: (Eq a, NFData b) => IVar (Maybe b) -> a -> [(a,b)] -> Par ()
lookupPar i _ []                    = put i Nothing
lookupPar i x ((k,v):r) | x == k    = put i $ Just v
                        | otherwise = lookupPar i x r

printEnvelope :: IVar (Maybe String) -> IVar String -> Par ()
printEnvelope clientV envV = do
  clientName <- get clientV
  case clientName of
    Nothing -> put envV "Unknown"
    Just n  -> put envV $ "To: " ++ n

printLetter :: IVar (Maybe String) -> IVar (Maybe String) -> IVar String -> Par ()
printLetter clientV productV letterV = do
  clientName  <- get clientV
  productName <- get productV
  case (clientName, productName) of
    (Nothing, Nothing) -> put letterV "Unknown"
    (Just n,  Nothing) -> put letterV $ n ++ " bought something"
    (Nothing, Just p)  -> put letterV $ "Someone bought " ++ p
    (Just n,  Just p)  -> put letterV $ n ++ " bought " ++ p
```

Одно из любопытных преимуществ отделения зависимостей потоков данных от реального параллельного выполнения заключается в том, что при этом могут применяться несколько стратегий планирования заданий. По умолчанию в monad-par используется планировщик под названием Direct. Доступны еще два планировщика: если вместо Control.Monad.Par импортировать Control.Monad.Par.Scheds.Spark или Control.Monad.Par.Scheds.Trace, то будет использован тот планировщик, который соответствует импортированному модулю.

Распараллеливание априорного алгоритма

Завершим мы этот раздел тем, что рассмотрим возможные способы усовершенствования априорного алгоритма для его выполнения в параллельном режиме. Код будет основан на реализации, предложенной в главе 7.

Если вы работаете со списками, то в пакет `monad-par` включена функция `parMap`. Ее назначение состоит в параллельном применении к каждому элементу списка некоей функции. Порождение задания для каждого элемента может показаться излишним, но планировщик учитывает количество ядер, доступных в системе. Чтобы применить функцию `parMap`, давайте сначала перепишем `generateL1`, чтобы из монадического стиля получить явные вызовы `map` и `concatMap`. Следующий код является полным эквивалентом соответствующего кода из главы 7:

```
generateL1 minSupport transactions =
  let c1 = noDups $ concatMap (\(Transaction t) -> map (FrequentSet . S.singleton)
$ S.toList t)
                              transactions
      l1NotFiltered = map (\fs -> (fs, setSupport transactions fs > minSupport)) c1
   in concatMap (\(fs,b) -> if b then [fs] else []) l1NotFiltered
```

Поскольку основное время в алгоритме тратится на вычисление поддержек, именно эта часть и была выбрана для параллельного выполнения. Прежде вычисления поддержек осуществлялись внутри функции `filter`, где также принималось решение о том, оставлять ли транзакцию в списке. Теперь эти две задачи разделены: поддержки множеств вычисляются в `l1NotFiltered`, а затем, в окончательной версии `concatMap`, принимается решение о том, нужно включать элемент или нет. После этого останется только заменить `map` на `parMap` и заключить все вычисления в оболочку `runPar`, чтобы воспользоваться параллелизмом потоков данных априорного алгоритма. Тогда будет получен следующий результат:

```
generateL1 minSupport transactions = runPar $ do
  let c1 = noDups $ concatMap (\(Transaction t) -> map (FrequentSet . S.singleton)
$ S.toList t)
                              transactions
  l1NotFiltered <- parMap (\fs -> (fs, setSupport transactions fs > minSupport)) c1
  return $ concatMap (\(fs,b) -> if b then [fs] else []) l1NotFiltered
```

ПРИМЕЧАНИЕ

Следует помнить, что при использовании библиотеки monad-par ваши типы данных должны быть экземплярами класса типов NFData.

В некоторых случаях эта стратегия может быть не самой лучшей для создания параллельных заданий. Вместо использования функции `parMap` можно делить список на половины до тех пор, пока не будет достигнута некая минимальная длина. Как только список станет достаточно мал, лучше выполнить его обход последовательно, поскольку создание параллельных заданий связано с издержками. Именно это и сделано в новой версии функции `generateNextLk`:

```
generateNextLk :: Double -> [Transaction] -> (Int, [FrequentSet])
               -> Maybe ([FrequentSet], (Int, [FrequentSet]))
generateNextLk _ _ (_, []) = Nothing
generateNextLk minSupport transactions (k, lk) =
  let ck1 = noDups $ [ FrequentSet $ a `S.union` b | FrequentSet a <- lk,
FrequentSet b <- lk
                                                   , S.size (a `S.intersection` b)
== k - 1 ]
```

```
      lk1 = runPar $ filterLk minSupport transactions ck1
   in Just (lk1, (k+1, lk1))

filterLk :: Double -> [Transaction] -> [FrequentSet] -> Par [FrequentSet]
filterLk minSupport transactions ck =
  let lengthCk = length ck
   in if lengthCk <= 5
      then return $ filter (\fs -> setSupport transactions fs > minSupport) ck
      else let (l,r) = splitAt (lengthCk `div` 2) ck
            in do lVar <- spawn $ filterLk minSupport transactions l
                  lFiltered <- get lVar
                  rVar <- spawn $ filterLk minSupport transactions r
                  rFiltered <- get rVar
                  return $ lFiltered ++ rFiltered
```

Как видите, использование библиотеки `monad-par` упрощает добавление параллелизма в текущий код. Эта библиотека нацелена на фьючерсы и программирование потоков данных. Но существуют и другие подходы. Например, в библиотеке `parallel` используется другая монада под названием `Eval`, которая помогает определить, как может быть обработана в параллельном режиме конкретная структура данных. Дополнительные сведения об этом и о других пакетах можно найти в Haskell-википедии (http://www.haskell.org/haskellwiki/Applications_and_libraries/Concurrency_and_parallelism).

РАСПАРАЛЛЕЛИВАНИЕ ЗАДАНИЙ С ПОБОЧНЫМИ ЭФФЕКТАМИ

О вычислениях с произвольными побочными эффектами мы пока еще не рассуждали. Однако на будущее имейте в виду, что пакет monad-par предоставляет для параллельных вычислений еще одну монаду, которая называется ParIO. Она доступна в модуле Control.Monad.Par.IO и допускает наличие побочных эффектов. Интерфейс у нее такой же, как в обычной монаде Par, за исключением выполнения вычислений через функцию runParIO.

Следует заметить, что реализация не гарантирует какой бы то ни было очередности выполнения заданий, и это выражается в побочном эффекте недетерминированного упорядочения.

С помощью библиотеки `monad-par` в параллельные могут быть превращены многие алгоритмы, работающие со списками или имеющие структуру типа «разделяй и властвуй». В упражнении 8.1 вам предлагается проделать это с еще одним представленным в данной книге алгоритмом анализа данных — с алгоритмом K-средних.

УПРАЖНЕНИЕ 8.1. ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ K-СРЕДНИХ

Напишите параллельную версию алгоритма K-средних, разработанного в главе 6. Для упрощения данной задачи можно посмотреть на первую реализацию, в которой монады не использовались. Следует помнить, что при применении таких функций, как parMap, нужно подумать о том, чтобы издержки от создания параллельных заданий не оказались больше получаемых от этого преимуществ.

Программная транзакционная память

В данном разделе рассматриваются проблемы, возникающие, когда несколько программных потоков взаимодействуют между собой и используют общие ресурсы.

То есть это именно та ситуация, когда имеет место *одновременность*. Хотя Haskell позволяет разрабатывать соответствующий инструментарий классическим способом, используя блокировки, семафоры и пр., в данном разделе мы увидим, что функциональный стиль программирования позволяет применить намного более мощную абстракцию, называемую программной транзакционной памятью.

Сначала нужно учесть тот факт, что код, поддерживающий концепцию одновременности, считается кодом с побочными эффектами. Когда несколько программных потоков выполняются асинхронно и совместно используют ресурсы, порядок, в котором это делается, влияет на наблюдаемый результат. В отличие от этого в чистом коде порядок, в котором вычисляются функции, совершенно не важен, поскольку результат всегда будет одним и тем же.

Больше узнать о том, как справляться с произвольными побочными эффектами, вы сможете в следующей главе. А пока вам достаточно знать, что в Haskell есть специальная монада `IO`, позволяющая использовать побочные эффекты. Единственным отличием кода, созданного при программировании с учетом и без учета побочных эффектов, является do-нотация.

Одновременное использование ресурсов

Давайте начнем путешествие по теме программирования концепции одновременности в Haskell с простого примера: имитации нескольких клиентов, покупающих товары в магазине. В первом приближении мы рассмотрим только те изменения, которые касаются денег, получаемых магазином машин времени. Код, создающий наши три программных потока, будет иметь следующий вид:

```
import Control.Concurrent
main :: IO ()
main = do v <- newMVar 10000
          forkIO $ updateMoney v
          forkIO $ updateMoney v
          forkIO $ updateMoney v
          _ <- getLine
          return ()

updateMoney :: MVar Integer -> IO ()
updateMoney v = do m <- takeMVar v
                   putStrLn $ "Updating value, which is " ++ show m
                   putMVar v (m + 500)  -- suppose a constant price
```

Первое, о чем нужно знать, — создание нового программного потока. Это делается с помощью функции `forkIO` из модуля `Control.Concurrent`. В качестве аргумента эта функция получает действие типа `IO ()`, после чего начинает выполнение кода в параллельном режиме.

ПРИМЕЧАНИЕ

Функция forkIO возвращает идентификатор потока, позволяющий приостанавливать и останавливать только что созданный программный поток. Однако функциональность модуля Control.Concurrent в этой книге не рассматривается.