Проблемы с потоками
The Problem With Threads

Эдвард А. Ли
Edward A. Lee

Отделение электротехники и информационных технологий
Университет штата Калифорния
Беркли

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-1

http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

10 Января 2006

Перевод: Петров А.В.
http://gmdidro.googlepages.com
avpetrov@computer.org
ver 1.0 || 2007 г.

Благодарности

Данная работа осуществлялась при поддержке Центра Гибридных и Встроенных Систем (Center for Hybrid and Embedded Software Systems (CHESS)) университета Беркли. Центр работает при поддержке Национального научного фонда США (National Science Foundation (NSF award No. CCR-0225610)), State of California Micro Program и следующих компаний: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft и Toyota.

Проблемы с потоками

Эдвард А. Ли

Профессор кафедры электротехники, член EECS
Отделение EECS
Университет Беркли, Калифорния

Беркли, CA 94720, С.Ш.А.
eal@eecs.berkeley.edu
http://ptolemy.eecs.berkeley.edu/~eal

January 10, 2006

Предисловие

Потоки, по всей видимости, являются непосредственной адаптацией концепции последовательных вычислений для параллельных систем. Языки программирования требуют весьма незначительных (а подчас и вообще не требуют) синтаксических изменений для поддержки потоков, в то же время поддержка потоков на уровне ОС и на уровне аппаратуры уже достаточно развита на сегодняшний день. При этом не ставится под сомнение, что для более эффективного использования аппаратного параллелизма программисты должны интенсивно использовать потоки. В данной статье я попытаюсь показать ущербность этой идеи. Со стороны может показаться, что многопоточное программирование мало чем отличается от обычного последовательного программирования, однако разница весьма и весьма значительна. Потоки, по сути, выбивают почву из-под ног программиста, поскольку отвергают самое существенное, что есть в последовательном программировании - предсказуемость, детерминизм и понятность программы. Недетерминизм присущ потокам, как модели вычислений. Несмотря на то, что многочисленные исследования и технологии стараются улучшить потоковую модель вычислений, предлагая те или иные средства уменьшения степени недетерминизма, я утверждаю, что это неправильный подход к решению данной проблемы. В основе модели вычислений должны лежать детерминированные компоненты, а не средства сокращения недетерминизма. Нужно иметь возможность добавить недетерминизм при необходимости, а не возможность убрать его при "ненужности". Другими словами не следует искать средства для борьбы с недетерминизмом в исконно недетерминированной модели, нужно создать детерминированную модель параллельных вычислений с возможностью добавления недетерминизма при необходимости. В действительности этот принцип может коренным образом изменить текущее положение дел. Необходимо разработать язык управления параллелизмом, причем этот язык должен опираться на надежные и понятные конструкции. Я уверен, что такой язык обеспечит возможность создания более надежных программ с высокой степенью параллелизма.

1 Введение

Проблемы, связанные с созданием параллельных программ, общеизвестны. Вместе с тем сегодня необходимость в параллелизме постоянно растет. Считается, что по мере того как действие закона Мура будет ослабевать, рост производительности будет достигаться за счет наращивания параллелизма в аппаратуре (многоядерные процессоры, многопроцессорные компьютеры, системы на кристалле)[15]. Если мы хотим получить выигрыш от сего факта, наши программы должны быть готовы к использованию параллелизма.

Одним из возможных решений является автоматизация внедрения параллелизма в последовательные программы, что может быть достигнуто либо за счет низкоуровневых технологий таких как, например, динамическая диспетчеризация, либо за счет автоматического распараллеливания программ[6]. Существует, однако, мнение, что технологии такого рода не найдут поддержки у компьютерного сообщества и что их возможности в плане распараллеливания последовательных программ несколько преувеличены. Все это позволяет сделать вывод о том, что параллельность должна закладываться в сами программы уже на этапе их создания, а не внедряться в них после их написания.

Если мы осознаем, в чем сложность параллельного программирования, мы сможем решить эту проблему. По словам Саттера и Ларуса (Sutter and Larus [47])

> "люди практически ощущают сложность параллельного программирования и понимают насколько сложнее программировать в параллельном стиле, нежели в последовательном. Даже наиболее внимательные из нас упускают из виду возможные комбинации в последовательности исполнения простейшего набора частично упорядоченных команд"

Тем не менее, человечество уже достаточно давно имеет дело с параллельными системами. Наш мир весьма неплохо "распараллелен" и мы ежедневно сталкиваемся с необходимостью реагировать на параллельно протекающие потоки событий. В программировании же проблема состоит в том, что базовой абстракцией параллельных вычислений была выбрана сущность, не имеющая даже приблизительного сходства с параллельностью в физическом мире. Однако мы настолько привыкли к выбранным абстракциям, что принимаем их за прописные истины, не пытаясь изменить им. В этой статье я хочу показать, что именно с этим связана сложность параллельного программирования и в случае, если мы откажемся от выбранных абстракций, появится возможность разрешить некоторые проблемы использования параллелизма.

Некоторую долю оптимизма в обзор текущего положения привносит Барроссо (Barrosso)[7], утверждающий, что как только технологии позволят на полную использовать параллелизм и в серверных системах, и в десктопных приложениях и в системах на микроконтроллерах, и как только такие технологии станут общепринятыми, то проблема программной модели параллельных вычислений и прочие проблемы, отпадут сами собой, поскольку им будет найдено чисто практическое решение. Тем не менее, понятно, что мы должны всерьез отнестись к трудностям использования параллелизма. По словам Стейна(Stein) [46] требуется "заменить привычную метафору последовательности шагов исполнения программы на понятие о множестве взаимодействующих сущностей". Этой идее и посвящена данная статья.

2 Потоки

На данный момент в программировании (в мейнстримном программировании) преобладает единственный подход к параллельному программированию - а именно программирование на основе потоков. Поток(thread) - это последовательный процесс, имеющий доступ к общей для всех потоков памяти. Потоки представляют собой базовую сущность, на основе которой построены модели параллелизма, используемые в современных компьютерах, языках программирования и операционных системах. Большинство используемых сегодня параллельных архитектур (например, SMP-системы и пр.) являются непосредственным воплощением потоков в аппаратуре.

Некоторые приложения могут крайне эффективно использовать потоки. В их число входят так называемые "тривиально распараллеливаемые" приложения, которые представляют собой множество независимых процессов. К таким приложениям можно, например, отнести средства линковки программ(PVM gmake) или веб-сервера. Относительная простота декомпозиции этих программ на независимые процессы существенно упрощает их реализацию, позволяя работать с потоками как с процессами ОС (когда один процесс не может получить доступ к памяти другого). Когда же в приложениях приходится реализовывать взаимодействия через данные, используются такие механизмы управления параллелизмом как транзакции. При этом обычно программисты клиентских приложений лишены таких возможностей.

> "Мир клиентских приложений далеко не так хорошо структурирован, как может показаться. В типичном клиентском приложений небольшие вычисления исполняются в ответ на события от пользователя. Можно распараллелить такое приложение, разделив вычисления по функциональному признаку. Однако, при этом выделенные части (такие как, например вычисления, отвечающие за пользовательский интерфейс и вычисления, отвечающие за собственно полезные расчеты) будут активно взаимодействовать и совместно использовать данные. Неоднородный код; кратковременные взаимодействия со сложной логикой; данные, потенциально доступные по указателю из любого места программы, все это сильно осложняет параллельное программирование клиентских приложений" [47]

Разумеется, потоки не являются единственным средством параллельного программирования. В научных и высокопроизводительных вычислениях в основном используются не потоки, а расширения языков программирования, поддерживающие параллелизм по данным, или различные библиотеки передачи сообщений(такие как PVM [23], MPI [39], или OpenMP). Фактически даже архитектура аппаратуры, на которой проводятся вычисления такого рода, значительно отличается от архитектуры обычных ПК. Например, широко распространена аппаратная поддержка векторной обработки или поточной(streams) обработки данных. Тем не менее, при такой мощной, "железной" поддержке параллельные программы все равно создаются с трудом. В этой области, несмотря на существование других языков программирования, поддерживающих параллелизм по данным (которых, кстати, было создано не мало за долгую историю научных вычислений), доминируют такие языки как C и FORTRAN.

В распределенных системах абстракция потоков почти не используется из-за того, что на практике создание иллюзии совместно используемой памяти обойдется слишком дорого. При этом, одна-

ко, мы можем видеть многочисленные попытки создания механизмов распределенных вычислений, эмулирующих потоки. Примером являются CORBA и .NET, возглавляющие список распределенных объектно-ориентированных технологий. Причем как в CORBA, так и в .NET компоненты(software components) взаимодействуют друг с другом через прокси-объекты, которые ведут себя так, как будто бы удаленный и локальный объекты размещены в одной памяти. Интересно, что именно объектно-ориентированность данных технологий (а именно принцип инкапсуляции) ограничивает необходимость полной эмуляции разделяемой памяти, вследствие чего данные технологии являются достаточно эффективными. В CORBA и .NET распределенное программирование очень похоже на многопоточное.

Встроенные системы(embedded computing systems) также используют модель параллелизма, основанную не на потоках. Архитектура DSP-процессоров обычно основана на принципах VLIW процессоров. Сигнальные процессоры для обработки видео обычно совмещают SIMD, VLIW процессоры и поточную(stream) обработку данных. В сетевых процессорах (например, в коммутаторах) также обеспечивается аппаратная поддержка обработки потоков данных. Несмотря на наличие достаточно интересных исследовательских проектов и разработок в этих областях, на практике в программировании используются достаточно примитивные модели. Разработка ведется на низкоуровневых ассемблерах, подчас жестко связанных с особенностями конкретной архитектуры, и лишь иногда используется язык C в тех местах программы, где производительность не столь критична.

Важным требованием к встроенным системам является надежность и предсказуемость их работы, что подчас более важно, чем производительность и гибкость. Разумеется, данное утверждение остается верным и в области "обычных" приложений. Я утверждаю, что в большинстве случаев надежность и предсказуемость работы систем не может быть достигнута при использовании потоков.

3 Потоки как модель вычислений

В данном разделе я рассмотрю потоки с более фундаментальной точки зрения, безотносительно к конкретным вариантам реализации потоков и их поддержки в языках и библиотеках. Будут показаны серьезные "дефекты" потоков как модели вычислений. Далее я рассмотрю возможные способы устранения этих дефектов. Положим $\mathbb{N} = \{0, 1, 2, \ldots\}$ - множество натуральных чисел, $B = \{0, 1\}$ - множество двоичных цифр. B^* - множество всех конечных последовательностей бит.

$$B^\omega = (\mathbb{N} \to B)$$

- множество всех бесконечных последовательностей бит (причем каждая последовательность является функцией, отображающей $\mathbb{N}$ в B). Согласно [17] $B^{**} = B^* \cup B^\omega$. Мы будем использовать B^{**} для представления состояния вычислительной машины, а также для представления потенциально бесконечных входных и выходных последовательностей программы. Пусть

$$Q = (B^{**} \rightharpoonup B^{**})$$

обозначает множество всех частичных функций, определенных на множестве B^{**} (имеющих область определения и область значения B^{**}).[1]

Назовем императивной машиной пару (A, c), где A - конечное множество атомарных действий и $A \in Q$; c - функция перехода (функция управления), $c\colon B^{**} \to \mathbb{N}$. Множество A представляет множество атомарных операций (инструкций) машины, функция c представляет порядок выполнения этих инструкций. Будем считать, что множество A содержит инструкцию halt: $h \in A$, такую что

$$\vee b \in B^{**}, \quad h(b) = b.$$

То есть инструкция h не изменяет состояние машины.

Будем называть последовательной программой длины $m \in \mathbb{N}$ функцию

$$p\colon \mathbb{N} \to A$$

где

$$\vee n \geqslant m, \quad p(n) = h.$$

То есть последовательная программа это конечная последовательность инструкций, заканчивающаяся бесконечной последовательностью halt-инструкций. Заметим, что множество последовательных программ, которое мы обозначим как P, является счетным бесконечным множеством.

[1] Частичными называют функции, значения которых могут быть, а могут и не быть определены для элементов области определения.

Исполнение такой программы связано с потоком. Поток начинается при начальном состоянии машины - $b_0 \in B^{**}$ и потенциально бесконечном вводе. Причем для всех $n \in \mathbb{N}$,

$$b_{n+1} = p(c(b_n))(b_n). \tag{1}$$

Здесь $c(b_n)$ определяет позицию следующей инструкции $p(c(b_n))$ в программе p. Если машина находится в состоянии b_n, то исполнение этой инструкции переводит машину в состояние b_{n+1}. Если для какого-либо $n \in \mathbb{N}$ $c(b_n) \geqslant m$, то $p(c(b_n)) = h$ и программа завершается в состоянии b_n (т.е. состояние программы больше не будет изменяться). Если же программа завершается при всех начальных состояниях $b_0 \in B^{**}$, то говорят что p задает всюду определенную на Q функцию. Если же программа завершается лишь для некоторых $b_0 \in B^{**}$, то p определяет частично определенную на Q функцию.[2]

Теперь мы можем определить, в чем же заключается "притягательность" последовательных программ. По данной программе и данному начальному состоянию с помощью выражения 1 мы можем определить последовательность исполнения программы. Если эта последовательность завершится инструкцией h, то это будет означать, что функция, заданная программой, полностью определена. Мы можем сравнить любые две программы p и p'. Программы будут эквивалентными, если они задают одну и ту же частично определенную функцию. Это значит, что p и p' будут эквивалентными, если они завершаются при одинаковых начальных состояниях и при этом их конечные состояния одинаковы.[3] Такая теория эквивалентности является существенно важной для определения любого формализма.

Описанные существенные и в тоже время притягательные свойства последовательных программ перестают действовать, когда мы совмещаем несколько потоков исполнения. Рассмотрим две программы p_1 и p_2, исполняющиеся параллельно в разных потоках. Выражение 1 изменится следующим образом:

$$b_{n+1} = p_i(c(b_n))(b_n) \;\; i \in \{1, 2\}. \tag{2}$$

То есть на шаге n любая программа может исполнить следующую инструкцию. Рассмотрим теперь какие изменения произошли в описанной выше теории эквивалентности. Для этого необходимо дать ответ на следующий вопрос: даны две пары многопоточных программ (p_1, p_2) и (p'_1, p'_2) необходимо определить являются ли эти пары эквивалентными. Продолжая логику базовой теории эквивалентности, мы можем сказать, что эти пары будут эквивалентными, если при любых возможных вариантах исполнения(*all interleavings*), программы будут завершаться при данных начальных состояниях и будут при этом находиться в одинаковых конечных состояниях. Число вариантов исполнения крайне велико, что не позволяет эффективно использовать предложенное расширение базовой теории (кроме разве что тривиальных случаев, когда, например, множество состояний B^{**} разделено таким образом, что программы при исполнении не влияют на состояния друг друга).

Ситуация на самом деле еще хуже - даже если мы знаем, что две программы p и p' являются эквивалентными при их последовательном исполнении (которое описывается выражением 1, мы не можем гарантировать их эквивалентность при их параллельном исполнении в многопоточной среде). Дело в том, что мы вынуждены знать обо всех других потоках данной среды, что само по себе не всегда возможно и, кроме того, мы должны проанализировать все возможные варианты исполнения этих потоков. Следовательно, мы можем сделать вывод, что потоки не дают нам возможности создать эффективную теорию эквивалентности программ.

Реализация потоковой модели вычислений создает значительные трудности. Примером этому могут служить достаточно сложные для понимания элементы модели памяти в Java (см. например [41] и [24]), когда даже абсолютно тривиальные на первый взгляд программы вызывают жаркие споры о возможных вариантах их поведения.

Широкое использование при построении языков программирования центральной абстракции вычислений, определенной выражением 1, подчеркивает важность принципа детерминированной композиции детерминированных компонентов. Последовательное исполнение является, по сути, композицией функций - простейшей моделью, в которой композиция детерминированных компонентов дает четко определенный результат.

[2]Заметим теперь, что приведенные рассуждения позволяют продемонстрировать одно из основных достижений компьютерной науки. Легко показать, что множество Q не является счетным множеством. (Даже подмножество Q - множество постоянных функций не является счетным, поскольку множество B^{**} не является таковым. Что может быть легко продемонстрировано с помощью доказательства Кантора) Однако поскольку множество всех конечных программ счетно, мы можем заключить, что не любая функция из Q может быть получена с помощью конечной программы. Следовательно мощность последовательных вычислительных машин ограничена. Тьюринг и Черч [48] показали, что какую бы интерпретацию машины (A, c) мы бы не выбрали, результирующее множество программ P будет давать одно и то же подмножество функций в Q. Функции этого подмножества были названы эффективно вычислимыми функциями.

[3] В рамках классической теории программы, которые не завершаются, не являются сравнимыми друг с другом. Это порождает серьезную проблему в случае применения теории вычислений к встроенным системам, поскольку обычно "хорошие" встроенные системы не должны завершать свою работу[34].

Потоки же являются крайне недетерминированными компонентами. Сегодня задача программиста состоит в том, чтобы по возможности избавиться от этого недетерминизма. Разумеется, на сегодняшний день мы имеем весьма неплохой инструментарий, облегчающий решение этой непростой задачи. Семафоры, мониторы, более современные абстракции, основанные на потоках (мы будем рассматривать их в следующем разделе) хорошо вооружают программиста в его борьбе с недетерминированным исполнением программы. Однако никакой секатор не способен превратить дикий вьюн в надежную живую изгородь.

Другими словами предположим, что мы попросили инженера-конструктора спроектировать двигатель внутреннего сгорания, предложив в качестве отправной точки - цилиндр с молекулами железа, углеводорода, воздуха, произвольно перемещающимися под действием тепла. Работа инженера будет состоять в том, чтобы упорядочить и ограничить это движение таким образом, чтобы двигатель заработал. В термодинамике и химии показано, что данный подход теоретически верен. Возьметесь?

Народная молва описывает безумие как попытки человека делать одну и ту же вещь снова и снова, ожидая при этом различных результатов. Мы можем сказать, что фактически программисты многопоточных систем безумны. Разве смогли бы они понимать свои программы, если были бы нормальными ?!

Я утверждаю, что мы должны (и что мы можем) создать более детерминированную модель параллельных вычислений и что мы должны вводить недетерминизм только тогда, когда мы полностью контролируем положение дел и осознаем необходимость его введения. Недетерминированность должна явно задаваться в программах тогда и только тогда, когда это действительно необходимо, т.е. в этом смысле мы должны поступать также, как поступаем при последовательном программировании. Потоки же "придерживаются" другой точки зрения. Они превращают наши программы в абсолютно недетерминированные программы и предполагают, что программист будет следовать определенным правилам, чтобы в рамках этого недетерминизма достичь вполне определенных целей.

4 Так ли все плохо на практике?

Выше было показано, что с помощью потоков не удается определить сколько-нибудь приемлемое расширение базовой теории вычислений. Тем не менее, сегодня, большинство практикующих программистов пишут многопоточные программы, которые работают.

```
public class ValueHolder {

  private List listeners = new LinkedList();
  private int value;

  public interface Listener
  {
    public void valueChanged(int newValue);
  }

  public void addListener(Listener listener)
  {
    listeners.add(listener);
  }

  public void setValue(int newValue)
  {
    value = newValue;
    Iterator i = listeners.iterator();
    while(i.hasNext())
    {
      ((Listener)i.next()).valueChanged(newValue);
    }
  }
 }
```

Пример 1. Реализация шаблона "наблюдатель" на Java, корректно работающая в однопоточном приложении

Есть ли в этом противоречие? Как уже было сказано, сегодня на вооружении программистов состоит множество средств истребления недетерминизма. Например, объектно-ориентированное программирование ограничивает видимость состояния различных частей программы, что позволяет эффективно разделить множество B^{**} на непересекающиеся подмножества. Если при этом необходимо обеспечить взаимодействие между этими подмножествами, то используются семафоры, взаимные блокировки или мониторы. Тем не менее, на практике эти механизмы позволяют создавать понятные и прозрачные программы только в случае достаточно простых взаимодействий.

Рассмотрим простой и широко используемый шаблон проектирования "наблюдатель" [22]. Выше показана реализация этого шаблона на Java, корректно работающая в однопоточном приложении. Вызов метода setValue() посылает сообщение, вызывая метод valueChanged() у всех объектов, которые были зарегистрированы методом addListener().

Очевидно, что приведенный код не является потокобезопасным - если несколько потоков вызовут метод setValue() или addListener(), то это может вызвать изменение списка listeners во время прохождения по нему итератора, что в свою очередь вызовет исключительную ситуацию и программа, по всей видимости, "упадет".

Простейшим решением является добавление ключевого слова synchronized к определениям методов setValue() и addListener(). Компилятор Java задействует при этом механизм взаимных блокировок, защитив обращения к объектам класса ValueHolder мониторами, что не позволит разным потокам одновременно вызывать методы этого класса. При вызове "синхронизированного" метода вызывающий поток попытается захватить блокировку на объект. Если какой-либо другой поток (назовем его первым) уже сделал это ранее, то вызывающий поток будет остановлен до момента освобождения блокировки первым потоком.

```
public class ValueHolder {

  private List listeners = new LinkedList();
  private int value;

  public interface Listener
  {
    public void valueChanged(int newValue);
  }

  public synchronized void addListener(Listener listener)
  {
    listeners.add(listener);
  }

  public void setValue(int newValue)
  {
    List copyOfListeners;
    synchronized(this)
    {
      value = newValue;
      copyOfListeners = new LinkedList(listeners);
    }
    Iterator i = copyOfListeners.iterator();
    while(i.hasNext())
    {
      ((Listener)i.next()).valueChanged(newValue);
    }
  }
}
```

Пример 2. Вроде бы как потокобезопасная реализация наблюдателя

Однако такое решение может привести к тупиковым ситуациям (дедлокам). В частности предположим, что у нас есть два объекта: объект a класса ValueHolder и объект b другого класса, реализующего интерфейс Listener. В реализации метода b.valueChanged() может происходить все что угодно, в том числе попытка захвата какой-либо блокировки Z. При этом если эта попытка завершится неудачно, то поток будет остановлен, а блокировка объекта a не будет освобождена. Тем временем другой поток,

удерживающий блокировку Z, может попытаться вызвать метод addListener() объекта a, однако для этого ему нужно будет захватить блокировку на этот объект, но она удерживается первым потоком. Таким образом оба потока будут находиться в состоянии бесконечного ожидания и нет никакой надежды, что эта ситуация как-нибудь разрешится. Такой вид дедлоков часто скрывается в программах, использующих мониторы и, разумеется, проявляют себя такие ошибки в самый неподходящий момент.

Как мы видим, корректное решение даже такой простой задачи оказалось непростым. Рассмотрим теперь улучшенную реализацию наблюдателя, представленную выше (пример 2). Удерживая блокировку, метод setValue() делает копию списка listeners. Поскольку метод addListeners() объявлен как synchronized, то это обеспечивает защиту от параллельной модификации списка (т.е. невозможно возникновение исключительной ситуации, описанной в примере 1). Кроме того метод setValue() вызывает метод valueChanged() вне блока синхронизации, что позволяет избежать дедлока.

Кажется, что задача теперь окончательно решена, однако на деле, этот код все еще способен вызвать проблемы. Предположим, два потока вызывают метод setValue(). Один из них последним изменит значение переменной value и именно это значение станет значением объекта. Однако зарегистрированные наблюдатели могут быть оповещены о произошедших изменениях значения в другом порядке. Т.е. наблюдателям будет казаться, что последним изменил значение переменной value другой поток, и они будут "видеть" несоответствующее действительности значение этой переменной!

Разумеется, шаблон наблюдатель может быть корректно реализован на Java (Я думаю, что читатель сам в состоянии придумать такую реализацию). Моя точка зрения состоит в том, что реализация даже настолько простого примера требует довольно-таки запутанных рассуждений и становиться все менее и менее прозрачной. Я полагаю, что в большинстве многопоточных программ имеются ошибки, подобные вышеописанным. И я считаю, что мы не можем мириться с существованием таких ошибок только из-за того, что архитектура современных компьютеров и операционных систем способна обеспечить параллелизм лишь на основе потоков. На сегодняшний день ресурсы, затрачиваемые на переключение контекста в многопоточных приложениях велики, поэтому на практике мы видим лишь небольшой процент всех возможных вариантов исполнения инструкций. Поскольку (по моему мнению) большинство многопоточных приложений содержат ошибки, связанные с параллельным исполнением потоков, я считаю, что можно выдвинуть следующую гипотезу - как только многоядерные системы станут широко используемыми, скрытые ошибки в многопоточных программах станут все чаще и чаще проявлять себя, приводя к падению ранее надежных систем. Такой вариант развития событий выглядит особенно удручающим с точки зрения производителей вычислительных систем - следующее поколение ПК будет характеризоваться в сознании потребителя как поколение компьютеров, на которых перестали нормально работать программы, замечательно выполнявшие свои функции на старых компьютерах.

Интересно, что сами производители процессоров ратуют за многопоточное программирование, стараясь всячески продвинуть его в программистские массы. Их действия обусловлены тем, что с их точки зрения многопоточное программирование позволит использовать параллелизм (многоядерность), который они собираются продавать. Так, например, Intel проводит активную кампанию по поддержке обучения студентов многопоточному программированию. Если эта компания будет иметь успех, то следующее поколение программистов будет еще более рьяно использовать потоки, что приведет к еще более удручающим последствиям.

5 Потоки. Есть секаторы и поострее ...

В этом разделе я хочу уделить внимание методам искоренения недетерминизма, обладающим одной общей чертой - мы рассмотрим методы, которые основываясь на потоках, предлагают программистам еще более сильные способы (подчас экстремальные) уничтожения недетерминизма. Первым способом является улучшение самого процесса разработки ПО. Понятно, что качество процесса разработки является важным при создании многопоточных программ. Однако этого явно недостаточно. История, произошедшая с проектом Ptolemy Project[4] иллюстрирует сей факт. В начале 2000 года моя группа начала разрабатывать ядро Ptolemy II [20] (это среда моделирования, поддерживающая различные модели параллельных вычислений). В то время перед нами стояла цель позволить пользователю с помощью GUI изменять параллельную программу во время ее исполнения (точнее, структурную схему программы (граф акторов), отображаемую на диаграмме). Проблема состояла в том, чтобы гарантировать, что никакой поток не сможет увидеть структуру программы в некорректном состоянии. Решение основывалось на использовании потоков и мониторов в Java.

В рамках проекта мы хотели создать процесс разработки ПО, подходящий для академических проектов. Организованный нами процесс разработки включал систему контроля качества кода (мы

[4] http://www.ptolemy.org

выделили 4 уровня готовности кода: красный, желтый, зеленый и синий), обсуждение проектирования и обсуждение ранее написанного кода (code and design reviews), "ночные билды", регрессионное тестирование и автоматизированные метрики кодирования[43]. Часть ядра, отвечающая за целостность структуры программы, при ее изменении, была написана также в начале 2000 года и получила желтый уровень по проектированию и зеленый по реализации. В обсуждениях и назначении уровней готовности кода участвовали не только мои студенты, но и опытные эксперты в области параллельного программирования (Christopher Hylands (Brooks), Bart Kienhuis, John Reekie). Были разработаны тесты, покрывающие 100% кода. "Ночные билды" и тестирование проходили на двухпроцессорных SMP машинах, что позволяло смоделировать различное поведение потоков (компьютеры, на которых велась разработка, были однопроцессорными). Система Ptolemy II стала широко использоваться, а часть ядра, реализующая решение описанной выше задачи, использовалась практически постоянно. Не было практически никаких проблем, пока 26 апреля 2004 года не обнаружился дедлок. До этого система нормально работала в течение 4 лет.

В определенной степени мы можем утверждать, что организация процесса разработки нашей системы позволила обнаружить и устранить многие ошибки в многопоточном коде. Но сам факт того, что такая серьезная ошибка как тупиковая ситуация не была обнаружена на протяжении 4 лет использования системы не может не настораживать. Интересно сколько еще подобных проблем остались невыявленными? Насколько продолжительным должен быть процесс тестирования, чтобы выявить все проблемы? Очевидно, что мы вынуждены сделать вывод о том, что тестирование не способно помочь нам выявить все ошибки в сложном многопоточном коде.

Нужно упомянуть, что существуют всем известные правила, которые позволяют избежать тупиковых ситуаций. Например - всегда захватывать блокировки в одном и том же порядке [32]. Как вы понимаете, на практике придерживаться этого правила чрезвычайно трудно, поскольку сигнатура метода ни в одном известном мне языке программирования не говорит о том, какие блокировки захватываются методом. Мы вынуждены изучить исходный код всех вызываемых нами методов, и всех методов, которые вызываются в них, для того чтобы гарантировать надежность исполнения метода. Даже если мы устраним эту проблему - введя в сигнатуру метода указание на захватывающиеся в нем блокировки, применение описанного правила все равно представляет собой очень трудную задачу при реализации симметричного доступа (когда взаимодействие может быть инициировано с обеих сторон). Кроме того, сам подход, основанный на взаимном исключении достаточно сложен, и если программисты не смогут понять написанный ими код, то он не может быть надежным.

Можно подумать, что проблема кроется в том, как реализована работа с потоками в Java. Возможно, использование ключевого слова synchronized не является лучшим средством для решения проблем. Но, например, в Java 5.0 (2005г) были добавлены новые механизмы для синхронизации потоков. Эти нововведения дают программисту удобный инструментарий для работы с многопоточным кодом. Но проблема не в этом. Проблема в том, что такие механизмы как, например, семафоры остаются слишком сложными в использовании, что в свою очередь ведет к разработке малопонятных программ и скрытым ошибкам.

Улучшение процесса разработки ПО не решило проблемы. Другим способом является использование проверенных шаблонов проектирования многопоточных приложений (см. например [32] и [44]). Действительно при использовании шаблонов программисту нужно просто подобрать подходящий для решения его проблемы шаблон. При этом правда приходится решать еще две проблемы. Во-первых, сама реализация шаблона подчас дело достаточно непростое. Программист может допустить ошибку и на сегодняшний день не существует средств автоматической проверки корректности реализации шаблонов проектирования. Но более важной проблемой является то, что шаблоны сложно компоновать друг с другом. Что приводит к неоправданной сложности применения нескольких шаблонов в случае достаточно нетривиальных программ.

Наиболее распространенный шаблон параллельных вычислений основывается на идее транзакций в области СУБД. Транзакции поддерживают спекулятивное исполнение параллельных потоков инструкций, оперирующих копиями данных, с последующим откатом или фиксацией результатов исполнения. Фиксация происходит в случае отсутствия конфликтов между транзакциями. Поддержка транзакций вводится обычно в распределенных системах (что постоянно используется в СУБД), кроме того, существуют варианты программной [45] и, что более интересно, аппаратной поддержки транзакций [38]. В последнем случае транзакции хорошо сочетаются с протоколами целостности кэша, которые сами по себе используются в любом компьютере. Транзакции исключают возможность непредвиденных дедлоков, но, несмотря на недавние расширения этого механизма [26] (вводящие возможности компоновки транзакций), они не позволяют детерминировано организовывать взаимодействия в программе. Механизм транзакций хорошо работает в исконно недетерминированных ситуациях, когда, например, несколько активных сущностей соревнуются за доступ к ресурсу.

Интересным примером использования шаблонов параллельного программирования является проект MapReduce [19]. Этот шаблон использовался Google для распределенной обработки больших массивов данных. В то время как большинство шаблонов описывают определенные структуры данных и потокобезопасный доступ к ним, MapReduce представляет собой каркас для построения распределенных приложений. MapReduce использует идею функций высшего порядка, которые используются в Lisp и других функциональных языках программирования. Параметрами шаблона являются не данные или их структуры, а функции.

Сами шаблоны могут быть инкапсулированы в библиотеках, как это было сделано в случае MapReduce, Java 5.0 или STAPL для C++[1]. Такой подход увеличивает надежность реализации шаблона, но требует, чтобы программист описывал все взаимодействия с помощью выбранной библиотеки. Объединение возможностей таких библиотек и возможностей языков программирования, когда синтаксис и семантика конструкций языка способствуют формализации описания взаимодействий, может позволить более просто и изящно конструировать параллельное ПО.

Шаблоны высшего порядка, такие как, например, MapReduce наводят на мысль о недостаточной мощности современных языков программирования. Такие шаблоны функционируют на уровне языков координации взаимодействий (coordination languages), а не на уровне обычных языков программирования. Мне кажется, что новый координационный язык, совместимый с широко распространенными языками программирования (Java, C++ и т.д.) будет более радушно принят общественностью, чем новый язык программирования заменяющий существующие.

Распространенным компромиссным вариантом является расширение существующего языка программирования путем введения в него новых конструкций, поддерживающих параллельное программирование. Подобный компромисс позволяет использовать уже написанный код и требует переписать только тот код, который имеет дело с параллелизмом. Такой путь выбрали, например, языки Split-C [16] и Cilk [13], которые являются Си-подобными языками программирования, поддерживающими многопоточность. В Cilk программист может использовать новые ключевые слова “cilk”, “spawn” и “sync”, продолжая разрабатывать программу на обычном Си. Целью Cilk является поддержка традиционной (Windows-like) многопоточности.

Наряду с добавлением новых языковых конструкций, создатели расширений ЯП иногда напротив намеренно ограничивают мощность существующего языка для придания ему более предсказуемого поведения. Например, язык Guava[5] вводит в стандартную Java ряд ограничений, запрещающих доступ к несинхронизированным объектам из разных потоков. Кроме того, данный язык проводит разделение блокировок на блокировки для чтения и для записи. Такие изменения языка в значительной степени искореняют недетерминизм, при этом не сильно влияют на производительность. Правда, вероятность дедлоков при этом все равно остается.

Другим подходом, который уделяет больше внимания дедлокам, являются “обещания” (promises), реализованные, например Марком Миллером в языке E. Авторами этого подхода могут считаться Бейкер и Хьюит (Baker и Hewitt) [27], правда в их интерпретации “обещания” называются “фьючерсами”.[5] В этом подходе вместо того чтобы реализовывать блокировки доступа к данным, программы работают с заместителями данных (proxy of data), которые они хотят получить в определенный момент времени.

Еще одним подходом к решению проблем управления параллелизмом является использование средств формального анализа программ для идентификации потенциальных ошибок. Примерами таких средств анализа могут служить Blast [28] и Intel thread checker. Автоматический анализ программ позволяет представить поведение программы в более доступном для понимания виде. Менее формальные инструменты, например профайлеры, также являются хорошими помощниками программиста. Несмотря на это, средства формального анализа и профайлеры требуют от программиста значительного опыта.

Все выше перечисленные средства искореняют недетерминизм потоков в той или иной мере. Однако, несмотря на успешность их применения, программист все равно в конечном итоге имеет дело с недетерминированной программой. Для приложений, которым присущ внутренний недетерминизм (сервера, службы параллельного доступа к базам данных и прочим ресурсам), такая ситуация является вполне допустимой. Но достичь детерминированного исполнения программы, отталкиваясь от недетерминированных потоков очень тяжело. Для решения этой задачи нужен другой подход. Создание параллельных систем должно начинаться не с таких механизмов как потоки, мы должны использовать детерминированные компоненты и вводить недетерминизм только при необходимости. Мы рассмотрим этот подход далее.

[5] Futures - Фьючерс - программная конструкция, указывающая на то, что результат некоторого вычисления будет использоваться в программе позже, но само вычисление может планироваться системой в любой произвольный момент времени.

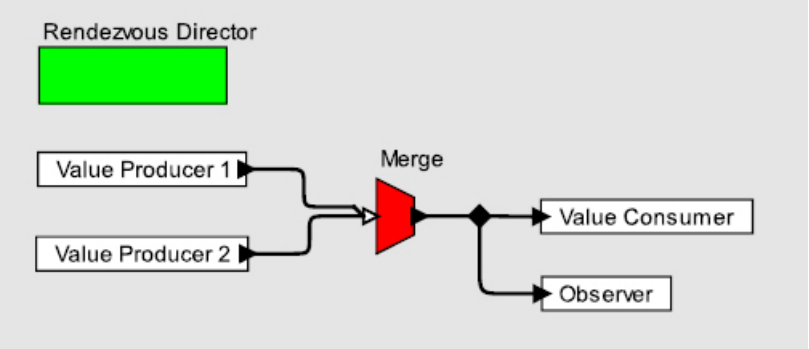

Рисунок 3. Шаблон проектирования “наблюдатель”, реализованный на координационном языке, основанном на механизме рандеву, с использованием визуальных элементов.

6 Альтернатива потокам

Рассмотрим снова ранее описанный шаблон проектирования “наблюдатель”. Это простой (тривиальный) шаблон проектирования [22]. И его реализация должна быть по возможности простой. Тем не менее, как было показано выше (примеры 1 и 2), его не просто реализовать в многопоточной среде.

Рассмотрим рисунок 3, на котором изображена реализация “наблюдателя” на основе механизма рандеву (или другими словами в рамках “Rendezvous domain”) в системе моделирования Ptolemy II [20]. Прямоугольник в левом верхнем углу с подписью “Rendezvous Director” является аннотацией к диаграмме, указывающей, что данная диаграмма изображает модель параллельной программы в рамках CSP-модели параллелизма[29], причем каждый компонент программы (изображенный прямоугольником на диаграмме) является процессом, а взаимодействия между процессами основаны на механизме рандеву. Сами процессы описаны на языке Java. Таким образом, Ptolemy II может рассматриваться как координационный язык (с визуальным в данном случае синтаксисом). Ptolemy II реализует рандеву на основе модели Reo(Reo)[2] и реализация включает “Merge” блок (блок слияния), который определяет условное рандеву. На диаграмме блок “Merge” декларирует, что любой из двух процессов Value Producers может начать рандеву с процессами Value Consumer и Observer. То есть, возможно наличие двух трехсторонних рандеву. Причем порядок возникновения взаимодействий не оговаривается.

Во-первых, обратите внимание, что как только вы узнали смысл прямоугольников на диаграмме, становится понятно, что диаграмма отображает шаблон “наблюдатель”. Во-вторых, заметим, что данная программа полностью детерминирована за исключением добавленного в явном виде недетерминированного блока “Merge”. Если убрать данный блок, то диаграмма будет определять детерминированные взаимодействия между определенными процессами. В-третьих, можно доказать невозможность дедлока в данной программе (в данном случае отсутствие циклов на диаграмме доказывает невозможность дедлока). В-четвертых, поскольку в рандеву включены и Value Consumer, и Observer, то они всегда будут видеть одинаковое значение переданной величины. Шаблон стал тривиальным, как это и должно было быть.

Теперь, когда мы смогли найти простое решение для тривиальной проблемы, мы можем попробовать немного усложнить его, придумав, что-нибудь интересное. На рисунке 4 показан тот же шаблон проектирования, реализованный в этот раз в рамках сети процессов Кана(Kahn) [31]. В Ptolemy II эта модель названа “PN Domain” (PN - process network, сеть процессов). В этой модели каждый прямоугольник вновь представляет процесс, но вместо взаимодействий, основанных на рандеву, в этой модели процессы “общаются” путем посылки сообщений через теоретически бесконечную FIFO-очередь. В оригинальной модели Кана, МакКюина (Kahn и MacQueen) [31], по блоковое чтение данных из очередей гарантирует, что каждая сеть процессов определяет детерминированные вычисления. В нашем случае модель PN была расширена за счет добавления примитива, названного “Недетерминированное слияние” (“NondeterministicMerge”) и выполняющего как следует из названия недетерминированное объединение потоков сообщений. Заметим, что такое недетерминированное расширение PN модели общепринято при моделировании встроенных систем [18].

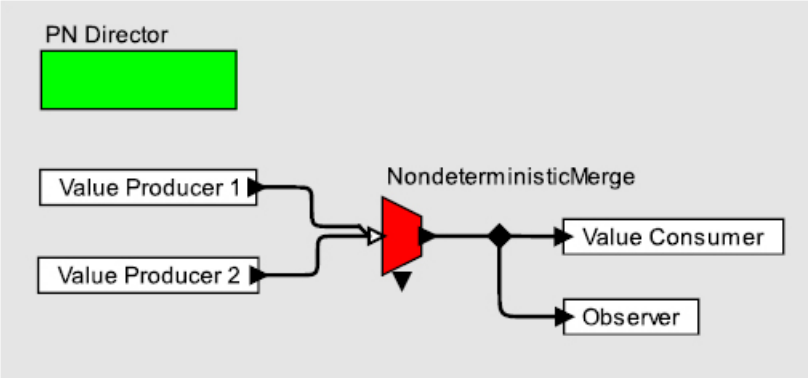

Рисунок 4 Наблюдатель, реализованный в рамках модели сети процессов

Новая реализация наблюдателя, представленная на рисунке 4, обладает всеми упомянутыми выше достоинствами реализации, основанной на рандеву (рисунок 3), при этом новым свойством является то, что Observer больше не должен быть синхронизирован с Value Consumer'ом. Сообщения об изменении данных могут быть поставлены в очередь для последующей обработки. В реализации, основанной на потоках, мы вряд ли бы даже пришли к подобной постановке вопроса, поскольку при многопоточном программировании программист стремится получить хоть какой-нибудь работающий (и по возможности корректный) вариант реализации шаблона.

Третий вариант реализации шаблона "Наблюдатель" делает упор на изменении природы недетерминированного блока "Недетерминированное слияние". Для обеспечения равноправного слияния потоков данных в этом блоке могут быть использованы принципы синхронных языков программирования [8]. В Ptolemy II подобная модель может быть реализована с помощью модели "SR domain" (synchronous/ reactive), которая реализует модели синхронных процессов языков Esterel [12], SIGNAL [10], и Lustre [25]. Последний язык из этого списка был успешно использован при разработке систем авиационного контроля [11]. Использовать потоки для приложений такого рода было бы, мягко говоря, неразумно.

Четвертый вариант реализации сфокусирован на синхронизации по недетерминированным событиям. Для этого в Ptolemy II мы использовали модель "DE domain"(discrete events, дискретные события), которая обеспечивает строго определенную синхронизацию по времени [33] и используется в таких языках описания аппаратуры как VHDL и Verilog, а также в каркасах моделирования вычислительных сетей, например в Opnet Modeler.

Во всех четырех случаях (рандеву, PN, SR, DE) мы в начале использовали фундаментально детерминированные механизмы параллельных взаимодействий (детерминированные в смысле модели вычислений, первые три случая были недетерминированными в смысле синхронизации по времени). Затем недетерминизм явно вводился в программу в тех местах, где это было нам необходимо. Такой подход к проектированию систем абсолютно отличается от подхода, основанного на потоках.

В Ptolemy II реализация вариантов, изображенных на рисунках 3 и 4, основана на потоках Java. Однако модель, которая использовалась программистами для создания этих вариантов, не была основана на потоках. Если сравнить данные варианты с подходами, описанными в предыдущем разделе, то самым близким из них будет каркас MapReduce, который также основан на идее потока данных через сеть вычислителей. Но в отличие от MapReduce, представленные варианты основаны на строго определенном координационном языке, достаточно мощном, чтобы описать широкий спектр возможных взаимодействий. Фактически здесь использовались два разных координационных языка, один, основанный на рандеву, а другой - на посылке сообщений (на PN модели).

Разумеется, предложенные подходы к параллелизму не являются сами по себе чем-то новым. Компонентные архитектуры, в которых используется поток данных (а не поток управления) через компоненты называются актор-ориентированными (actor-oriented[6])[35]. Такие системы принимают разнообразные формы. Каналы Unix отражают идею PN модели, хотя они(unix-каналы/pipes) более ограничены и не поддерживают цикличность в графе передачи сообщений. Библиотеки посылки сообщений, такие как MPI и OpenMP, включают средства, отражающие как модель рандеву, так и PN модель, но имеют менее структурированную форму, что обуславливает их большую выразительность в ущерб детерминированности. Неопытный пользователь таких библиотек может легко потонуть в неожиданных проявлениях недетерминизма.

[6]Прим.перев: При этом если в таких системах актор обладает собственным потоком управления, то такие системы могут быть названы системами на основе активных объектов (active object). Однако актор-ориентированные системы не ограничиваются активными объектами. Класс актор-ориентированных систем шире и включает системы, в которых потоки управления могут *на уровне реализации* свободно перемещаться между акторами. В этом смысле в Ptolemy II актор-ориентированные системы представлены полностью.

Некоторые языки программирования, например Erlang [4], поддерживают параллелизм, основанный на посылке сообщений, на уровне языка. Другие языки, например, Ada основаны на модели рандеву. Функциональные языки [30] и языки с однократным присваиванием также делают упор на детерминированность, но они в меньшей степени поддерживают явный параллелизм, затрудняя управление им. Языки параллелизма на уровне данных делают упор на детерминированность взаимодействий, но требуют переписывания программ на достаточно низком уровне.

Все эти методы, способы и подходы являются частями общего решения. Однако, по всей видимости, не один из них не станет во главе "мейнстрима". Я попытаюсь осветить этот вопрос далее.

7 Проблемы и перспективы

Реальность такова, что, несмотря на существование в течение долгого времени моделей, альтернативных потокам, сегодня только потоки доминируют в программировании, т.е. другими словами - при решении "обычных задач" сегодня используются потоки. Возможно, это связано с тем, что сам дух программирования, все ключевые абстракции программирования крепко-накрепко связаны с парадигмой последовательных вычислений. Большинство используемых языков программирования безоговорочно следуют этой парадигме. Синтаксически потоки, являются незначительным расширением для таких языков (например, потоки в Java) или даже представлены в виде внешних библиотек. При этом, разумеется, семантически потоки абсолютно разрушают детерминизм таких языков. Однако, программисты, по всей видимости, руководимы в первую очередь синтаксисом, а не семантикой. Альтернативы потокам, которые более или менее распространены (имеются ввиду MPI и OpenMP) обладают той же чертой. Они даже не меняют синтаксиса языка. Альтернативы потокам, которые заменяют общеупотребительные языки на языки с полностью новым синтаксисом (например, языки Erlang и Ada) не используются широко и скорей всего уже не станут общеупотребительными. Даже языки, которые совсем чуть-чуть изменяют синтаксис основного языка (например, Split-C и Cilk) остаются изгоями.

Проблема понятна. Мы не должны заменять общепризнанные языки программирования. Вместо этого мы должны основываться на них. Одним из вариантов являются библиотеки, однако их мощность/строгость явно недостаточна. Библиотеки менее структурированы, чем языки, не могу навязывать те или иные стандартные и проверенные решения и обладают низкой компоновочной способностью.

Я уверен, что решение находится в области создания координационных языков. Такие языки вводят новый синтаксис, но этот синтаксис служит целям, ортогональным целям основным языков программирования. В то время как Erlang и Ada определяют полностью новый синтаксис (т.к. являются полноценными языками программирования), координационный язык должен определять только те конструкции, которые нужны для удовлетворения его координационных целей. Учитывая это, синтаксис основного языка не претерпит кардинальных изменений. "Программы", изображенные на рисунках 3 и 4 используют визуальный синтаксис для определения вычислений на акторах (когда между компонентами происходит обмен данными, а не потоками управления). Несмотря на то, что визуальный синтаксис использовался только в педагогических целях, вполне возможно, что визуализация языка обеспечит его масштабируемость и эффективность - точно также как некоторые части UML используются при объектно-ориентированном программировании. Если даже этого и не случится, замена визуального представления структуры текстовым не составляет больших трудностей.

Разумеется, и координационные языки существуют уже на протяжении долгого времени [40]. И все они не смогли занять значимое место в обычном программировании (выпали из мейнстрима). Одной из причин такого положения дел является их однородность(homogeneity). При разработке и исследовании новых языков программирования легко прослеживается следующая тенденция - любой реальный, взрослый, практичный, ... язык программирования должен быть полноценным. Для этого, как минимум, он должен быть достаточно выразительным, для того чтобы на нем можно было написать его собственный компилятор. Кроме того, для программиста смена языка программирования подчас равноценна смене пола (прим. перев.: гиперболизация моя :)). Не секрет, что постоянно происходят "Священные войны" за языки программирования и что лишь некоторые из священников состоят в нескольких религиях.

И все же лед уже тронулся. Совместное использование UML (который, по сути, может быть рассмотрен как семейство языков с визуальным синтаксисом) и таких языков, как C++ или Java, уже вошло в норму современного программирования. Программисты постепенно начинают использовать более чем один язык программирования, когда им необходимы дополняющие друг друга средства различных языков. Программы на рисунках 3 и 4 реализованы в этом же духе - диаграммы 3 и 4 определяют крупно масштабную структуру системы и в этом плане они ортогональны языкам, детально описывающим вычислительный процесс.

Модели параллелизма с более высокой степенью детерминизма, чем у модели потоков, а именно CSP, PN модели, модель потоков данных (dataflows) также были разработаны достаточно давно. Некоторые из них легли в основу языков программирования, например язык Occam [21] (прим.перев.: не путать с OCaml) основан на CSP , на основе других моделей реализованы проблемно-орентированные каркасы(например YAPI [18]). Тем не менее, большинство моделей параллелизма использовались при разработке различных исчислений процессов (process calculi) и не имели большого влияния на практическое программирование. Я уверен, что ситуация изменится, если эти модели будут использоваться при создании новых координационных языков.

На этом пути раскидано множество граблей. Спроектировать хороший координационный язык не проще, чем создать язык программирования общего назначения. Ошибки и компромиссы приходится преодолевать в обоих случаях. Например, разработчик может легко потеряться средь избыточного количества примитивов языка. Известно, что для построения обычного языка программирования достаточно семи примитивов [48], но не один из серьезных языков не ограничивается этим числом.

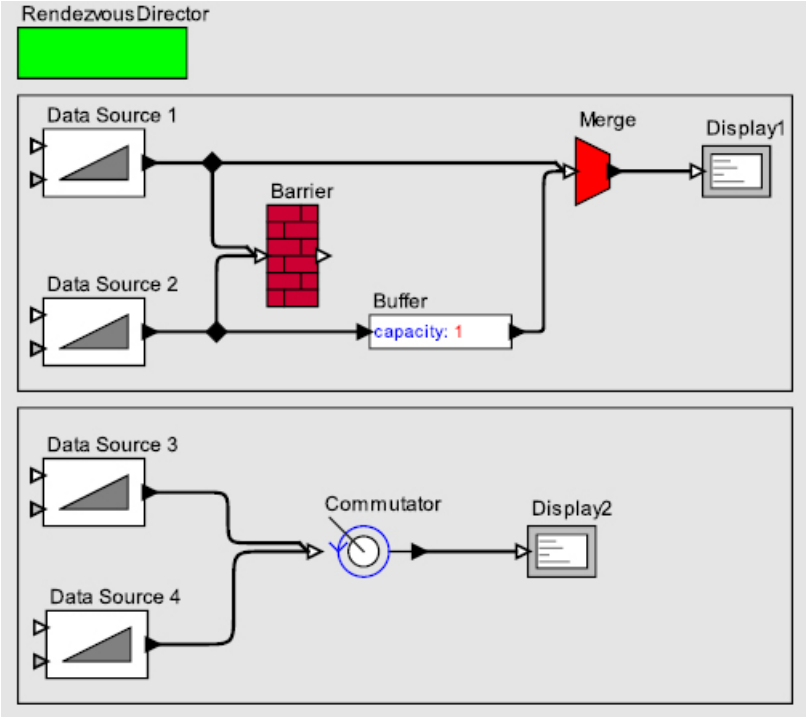

Рисунок 5. Два способа достичь определенного чередования выходных потоков данных с помощью рандеву

На рисунке 5 изображены две реализации простых параллельных вычислений. На верхнем рисунке реализован адаптированный пример из [3]. Последовательные потоки данных от Data Source 1 и Data Source 2 детерминировано смешиваются и затем выводятся в чередующемся порядке на экран (на блок Display). Эта достаточно простая цель достигается достаточно сложным способом. Фактически верхняя часть рисунка 5 напоминает головоломку.[7]

Программа в нижней части рисунка, напротив, достаточно понятна. Рандеву между блоком коммутатор (Commutator) и каждым из “входных” процессов возможно в определенном порядке (свойство блока Commutator) - в данном случае снизу вверх, в результате чего поставленная задача легко решается. Разумный выбор языковых примитивов позволяет легко и непосредственно, а главное понятным (т.е. детерминированным) путем достигать поставленных целей. Программа на верхней диаграмме для этого использует недетерминированные, хотя и более выразительные механизмы. При этом она менее понятна.

Разумеется, координационный язык, также как и язык общего программирования, должен разрабатываться с учетом масштабируемости и модульности. И эта задача разрешима. Например, в Ptolemy II определена достаточно мощная система типов на уровне координационного языка [49]. Более того, эта

[7]Блок (актор) “барьер” (Barrier) гарантирует, что все “входные” по отношению к нему процессы будут участвовать в одном и том же рандеву. Блок “слияние” (Merge) описан выше. Блок “буфер” (Buffer), с емкостью в единицу, изначально пуст и поэтому готов к рандеву с любым “входным” процессом. Как только буфер заполняется, он может инициировать рандеву только с “выходным” процессом. Таким образом, на данной диаграмме возможно возникновение двух рандеву в чередующейся последовательности. Первое рандеву включает в себя оба источника данных(Data Source), буфер и дисплей (Display). Второе - только буфер и дисплей.

система обладает адаптированными формами полиморфизма и наследования, которые были позаимствованы из ООП [35]. Большой потенциал, по-видимому, скрыт в идее адаптации функций высшего порядка для координационных языков. Некоторые предварительные исследования в этом направлении ведутся под управлением Риики (Reekie) [42].

Более сложной задачей представляется адаптация теории вычислений для обеспечения надежного базиса параллельных вычислений. Несмотря на значительный прогресс в этом направлении, мне кажется, что до полной победы еще далеко. В разделе 3 последовательные вычисления моделировались как функция, отображающая последовательность битов на себя. Соответствующая модель для параллельных вычислений рассматривается в [36], где вместо функции

$$f : B^{**} \to B^{**}$$

параллельные вычисления моделируются как

$$f : (T \to B^{**}) \to (T \to B^{**}).$$

Где T - это полностью или частично упорядоченное множество тэгов(tags), где порядок отображает время, причинную обусловленность или более абстрактную зависимость. Вычисление рассматривается как отображение последовательности битовых комбинаций на себя (maps an evolving bit pattern into an evolving bit pattern). Как было показано, эта базовая формулировка применима к большинству моделей параллелизма [9,14,37].

8 Заключение

Параллелизм в программировании штука непростая. Однако значительная часть трудностей связана с тем, какие абстракции, какие модели параллелизма мы используем. На сегодняшний день потоки остаются доминирующей моделью параллельных вычислений. Но нетривиальные многопоточные программы *непостижимы для человеческого мозга.* Шаблоны проектирования, повышение атомарности операций (введение транзакций), улучшение языков и использование формального анализа безусловно улучшают потоковую модель параллелизма. Однако данные методы просто уменьшают степень недетерминированности потоковой модели. Эта модель по сути своей является трудноизлечимой, присущий ей недетерминизм - это хроническая болезнь.

Если мы хотим, чтобы параллельное программирование стало общедоступным, и если при этом мы хотим гарантировать надежность и предсказуемость наших программ, то мы должны отказаться от модели параллелизма, основанной на потоках. Существуют более предсказуемые и понятные модели параллелизма. Все они основаны на очень простом принципе - понятные цели должны достигаться понятными средствами (*deterministic ends should be accomplished with deterministic means*). Недетерминизм, неоднозначность, не-вполне-определенность могут при необходимости использоваться в программах, но мы должны иметь средство явно заявить о том, где и как мы собираемся добавить недетерминизм. Этот принцип кажется очевидным, но его не возможно придерживаться, используя потоки. Потоки должны быть сосланы в машинное отделение программирования, и только специалисты должны иметь доступ к ним.

Список литературы

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), pages 193-208, Cumberland Falls, Kentucky, 2001.

[2] F. Arbab. Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science, 14(3):329-366, 2004.

[3] F. Arbab. A behavioral model for composition of software components. L'Object, to appear, 2006.

[4] J. Armstrong, R. Virding, C.Wikstrom, and M.Williams. Concurrent programming in Erlang. Prentice Hall, second edition, 1996.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, volume 35 of ACM SIGPLAN Notices, pages 382-400, 2000.

[6] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. Proceedings of the IEEE, 81(2):211-243, 1993.

[7] L. A. Barroso. The price of performance. ACM Queue, 3(7), 2005.

[8] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. Proceedings of the IEEE, 79(9):1270-1282, 1991.

[9] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In EMSOFT. Springer, 2003.

[10] A. Benveniste and P. L. Guernic. Hybrid dynamical systems theory and the signal language. IEEE Tr.on Automatic Control, 35(5):525-546, 1990.

[11] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[12] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(2):87-152, 1992.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP), ACM SIGPLAN Notices, 1995.

[14] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Notes on agent algebras. Technical Report UCB/ERL M03/38, University of California, November 2003.

[15] M. Creeger. Multicore CPUs for the masses. ACM Queue, 3(7), 2005.

[16] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. Eicken, and K. Yelick. Parallel programming in Split-C. In Supercomputing, Portland, OR, 1993.

[17] B. A. Davey and H. A. Priestly. Introduction to Lattices and Order. Cambridge University Press, 1990.

[18] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. A. Vissers. Yapi: Application modeling for signal processing systems. In 37th Design Automation Conference (DAC'00), pages 402-405, Los Angeles, CA, 2000.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Sixth Symposium on Operating System Design and Implementation (OSDI), San Francisco, CA, 2004.

[20] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the Ptolemy approach. Proceedings of the IEEE, 91(2), 2003.

[21] J. Galletly. Occam-2. University College London Press, 2nd edition, 1996.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing. MIT Press, Cambridge, MA, 1994.

[24] A. Gontmakher and A. Schuster. Java consistency: nonoperational characterizations for Java memory behavior. ACM Trans. Comput. Syst., 18(4):333-386, 2000.

[25] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. Proceedings of the IEEE, 79(9):1305-1319, 1991.16

[26] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In ACM Conference on Principles and Practice of Parallel Programming (PPoPP), Chicago, IL, 2005.

[27] J. Henry G. Baker and C. Hewitt. The incremental garbage collection of processes. In Proceedings of the Symposium on AI and Programming Languages, volume 12 of ACM SIGPLAN Notices, pages 55-59, 1977.

[28] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In 15th International Conference on Computer-Aided Verification (CAV), volume 2725 of Lecture Notes in Computer Science, pages 262-274. Springer-Verlag, 2003.

[29] C. A. R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8), 1978.

[30] P. Hudak. Conception, evolution, and application of functional programming languages. ACM Computing Surveys, 21(3), 1989.

[31] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, Information Processing. North-Holland Publishing Co., 1977.

[32] D. Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley, Reading MA, 1997.

[33] E. A. Lee. Modeling concurrent real-time processes using discrete events. Annals of Software Engineering, 7:25-45, 1999.

[34] E. A. Lee. What's ahead for embedded software? IEEE Computer Magazine, pages 18-26, 2000.

[35] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In Conference on Formal Methods and Models for Codesign (MEMOCODE), San Diego, CA, USA, 2004.

[36] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. IEEE Transactions on CAD, 17(12), 1998.

[37] X. Liu. Semantic foundation of the tagged signal model. Phd thesis, EECS Department, University of California, December 20 2005.

[38] H. Maurice and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In Proceedings of the 20th annual international symposium on Computer architecture, pages 289-300, San Diego, California, United States, 1993. ACM Press.

[39] Message Passing Interface Forum. MPI2: A message passing interface standard. International Journal of High Performance Computing Applications, 12(1-2):1-299, 1998.

[40] G. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, Advances in Computers - The Engineering of Large Systems, volume 46, pages 329-400. Academic Press, 1998.

[41] W. Pugh. Fixing the Java memory model. In Proceedings of the ACM 1999 conference on Java Grande, pages 89-98, San Francisco, California, United States, 1999. ACM Press.

[42] H. J. Reekie. Toward effective programming for parallel digital signal processing. Ph.D. Thesis Research Report 92.1, University of Technology, Sydney, 1992.

[43] H. J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the Ptolemy project. Technical Report Series GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California, Berkeley, April 1999.

[44] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects. Wiley, 2000.

[45] N. Shavit and D. Touitou. Software transactional memory. In ACM symposium on Principles of Distributed Computing, pages 204-213, Ottowa, Ontario, Canada, 1995. ACM Press.

[46] L. A. Stein. Challenging the computational metaphor: Implications for how we think. Cybernetics and Systems, 30(6), 1999.

[47] H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7), 2005.

[48] A. M. Turing. Computability and -definability. Journal of Symbolic Logic, 2:153-163, 1937.

[49] Y. Xiong. An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1 2002.

Верстка и редактирование данного документа осуществлялось при помощи макропакета LaTeX