Метою даної роботи є дослідження застосування технік, притаманних функціональним мовам програмування, при об`єктно-орієнтованій розробці паралельного програмного забезпечення. Описуються переваги і недоліки функціонального та імперативного підходів до створенню паралельних програм

Ключові слова: паралелізм, функціональне програмування, модель акторів, scala, erlang

Целью данной работы является исследование применения техник, свойственных функциональным языкам программирования, при объектно-ориентированной разработке параллельного программного обеспечения. Описаны преимущества и недостатки функционального и императивного подходов к созданию параллельных программ

Ключевые слова: параллелизм, функциональное программирование, модель актеров, scala, erlang

The primary intent of this paper is an investigation of applying functional techniques in object-oriented context for concurrent software development. Pros and cons of functional and imperative approaches for concurrent development were described

Keywords: concurrency, functional programming, actor model, scala, erlang

УДК 004.432.4

ПРИМЕНЕНИЕ ФУНКЦИОНАЛЬНОГО ПОДХОДА В ПАРАЛЛЕЛЬНОМ ПРОГРАММИРОВАНИИ

З.В. Дударь
Кандидат технических наук, профессор, исполняющая обязанности заведующего кафедрой*
E-mail: software@kture.kharkov.ua

А.В. Вечур
Кандидат технических наук, доцент*
E-mail: vechur@kture.kharkov.ua

Р.Г. Городищер*
E-mail: roman.gorodischer@gmail.com
*Кафедра ПО ЭВМ
Харьковский национальный университет радиоэлектроники
пр. Ленина, 14, г. Харьков, Украина, 61166

1. Введение

Рост популярности внедрения функциональных концепций в коммерчески успешные языки связан с тем, что использование функционального подхода может значительно упростить решение ряда задач, с которыми сталкиваются разработчики на объектно-ориентированных языках. В частности, одной из наиболее актуальных проблем становится создание параллельных приложений. Это связано с тем, что вычислительные ресурсы современных многоядерных процессоров могут быть использованы в полной степени только при исполнении многопоточных приложений [1]. Создание многопоточных приложений всегда было наиболее сложной задачей для объектно-ориентированных языков. Функциональные языки, в особенности Erlang, отлично зарекомендовали себя как средства создания коммерческих многопоточных приложений.

В данной работе мы рассмотрим то, как успешные функциональные техники параллельного программирования могут быть использованы в объектно-ориентированном контексте.

2. Проблема использования вычислительных ресурсов современных многоядерных процессоров

Основные производители и архитектуры процессоров, включая производителей Intel и AMD, и архитектуры Spark и PowerPC, прекратили наращивать мощности процессоров традиционным способом. Вместо увеличения частоты процессоров и ускорения обработки команд, делается упор многоядерные архитектуры. И это является фундаментальной поворотной точкой в разработке программных продуктов, по крайней мене на ближайшие несколько лет в разработке приложений для обычных персональных компьютеров и для низшего ценового диапазона рынка серверов.

В ближайшие несколько лет, прирост производительности в новых процессорах будет достигаться за счет трех основных направлений:

1) гиперпоточность (hyperthreading);
2) многоядерность;
3) увеличение кэша [1].

Под гиперпоточностью понимается обработка двух или более потоков выполнения параллельно одним процессором. Гиперпоточность в общем случае дает

5-15% прирост производительности для качественно реализованных многопоточных приложений.

Многоядерность означает, что два или более отдельных процессора присутствуют на одном чипе. Прирост производительности должен быть таким же, как и при использовании полноценной двухпроцессорной системы, что на практике означает увеличение производительности в незначительно меньшее число раз, чем количество ядер на одном чипе, но только для хорошо написанных многопоточных приложений. Однопоточные приложения не могут использовать появившиеся дополнительные вычислительные ресурсы в виде нескольких ядер на одном чипе.

Можно рассчитывать, что рост размера кэша будет продолжаться в ближайшие годы. Из трех перечисленных подходов к увеличению вычислительной мощности процессоров, только рост кэша может улучшить производительность большинства существующих приложений, так как качественно созданные многопоточные приложения встречаются крайне редко.

Основной вывод из перечисленных аргументов – в ближайшем будущем будет необходимость создавать преимущественно многопоточные хорошо распараллеливаемые приложения, если хочется в полной степени задействовать возможности современных процессоров.

3. Анализ публикаций

В статье Герба Саттера «Бесплатный обед закончился (The Free Lunch Is Over)» [1] утверждается, что производители микропроцессоров достигли предела тактовой частоты процессоров, из чего делается вывод, что производители сконцентрируют свои усилия на увеличении количества вычислительных блоков (ядер) в одном процессоре, и что разработчики программного обеспечения будут вынуждены создавать многопоточные приложения, чтобы лучше использовать имеющиеся вычислительные ресурсы.

В статье на JavaLobby «Scala: параллелизм без потоков» [2] приведено сравнение двух моделей параллельного программирования: модели с разделяемым состоянием и модели без разделяемого состояния. Вторая модель реализована в функциональных языках Erlang и Scala. Указывается, что хорошая интеграция Scala и Java позволяет использовать модель многопоточности без разделяемого состояния, характерную для функциональных языков, и в объектно-ориентированном контексте.

4. Цели статьи

В рамках данной статьи предполагается описать и сравнить две модели параллелизма, описать особенности применения модели параллелизма, не использующей разделяемое состояние, в нефункциональных языках программирования.

5. Модели параллелизма

5.1. Модель параллелизма с разделяемым состоянием

Большинство популярных объектно-ориентированных языков программирования поддерживают параллелизм с изменяемым состоянием, которое основывается на синхронизации конкурирующих потоков посредством таких механизмов как блокировки, мьютексы, семафоры. Все абстракции, относящиеся к параллелизму, основываются на парадигме разделяемого изменяемого состояния и склонны к таким проблемам, как состояние гонок, борьба за ресурс и взаимная блокировка (deadlock) [3].

Основным источником проблем в этой модели является именно разделяемое состояние. Как модель вычислений, потоки являются недетерминированными, а при разделяемом состоянии, когда потоки могут потенциально получить доступ к общей информации в памяти одновременно, программная модель становится чрезвычайно сложной.

Для иллюстрации двух наиболее часто встречаемых проблем использованы таблицы, состоящие из двух колонок, каждая из которых соответствует одному потоку выполнения, в ячейках отображены операции, совершаемые на определенном шаге.

Первая проблема, показанная в табл. 1, это проблема потерянных изменений (lost-update problem). Представим, что два потока хотят изменить значение разделяемого объекта account. Они оба считывают значение из объекта, наращивают это значение и сохраняют обратно в объекте. Так как эти операции не являются атомарными, то существует вероятность, что выполнение этих операций в обоих потоках пересечется во времени, что приведет к неверному результирующему значению account, как показано в примере.

Таблица 1

Иллюстрация проблемы потерянных изменений

Process 1	Process 2
a = account.get()	
a = a + 100	b = account.get()
	b = b + 50
	account.set(b)
account.set(a)	

Для решения проблемы потерянных изменений используются блокировки. Блокировки обеспечивают взаимное исключение при выполнении потоков, т.е. только один поток может владеть блокировкой в одно и то же время.

Используя протокол блокировок, на основе которого делаются заключения о том, что необходимые блокировки свободны, и разделяемый объект можно безопасно использовать, можно избежать проблемы потерянных изменений.

Однако, с блокировками связаны новые проблемы, одна из которых – взаимная блокировка (deadlock). Пример взаимной блокировки показан в табл. 2. В этом примере два потока пытаются в одно и то же время завладеть двумя блокировками A и B. Ситуация, когда оба потока захватывают по одной блокировке, а потом пытаются захватить вторую, называется взаимной блокировкой. Оба потока ждут друг друга, пока не они не освободят захваченные блокировки, чего никогда не произойдет.

В основе описанных проблем находятся трудности корректной работы с изменяемым состоянием в условиях параллелизма.

Таблица 2

Иллюстрация проблемы взаимной блокировки

Process 1	Process 2
lock(A)	lock(B)
lock(B)	lock(A)

5.2. Модель параллелизма без разделяемого состояния

Альтернативная модель, без изменяемого состояния, которая также известна как «модель актеров», поддерживается в функциональных языках Erlang, Scala и некоторых других. Эта модель распространилась благодаря языку Erlang, программы на котором широко используются в тех областях, где необходимо достичь высокой степени параллелизма. Языку Erlang удалось популяризовать параллельное программирование за счет реализации крайне легковесной модели процессов, абстрагировавшись от реализации потоков, поддерживаемой операционной системой и платформой [2].

Модель актеров в Erlang основывается на семантике крайней изоляции, самодостаточных процессах с асинхронной передачей сообщений. Сильная изоляция гарантируется виртуальной машиной, исполняющей программы, которая принуждает использовать семантику одноразовой инициализации и использованием функциональной модели с неизменяемыми данными.

В модели актеров каждый объект является актером. Актер – это сущность, у которой реализован почтовый ящик и задано поведение. Актеры могут обмениваться сообщениями, которые будут попадать в почтовый ящик актера-получателя.

Для каждого актера определены три простые операции. Перечислим эти операции:

1) отправить;

2) создать;

3) перейти в новое состояние.

Операция «отправить» выполняет асинхронно (т.е. в новом потоке) передачу сообщения известному получателю. Операция «создать» создает нового актера с заданным поведением и состоянием. Операция «перейти в новое состояние» определяет поведение, используемое при обработке очередного входящего сообщения.

Когда актер получает сообщение, выполняется определенное действие, в результате чего актер может: посылать сообщения другим актерам, создавать новых актеров и выполнять новое, отличное действие для обработки последующих сообщений.

Важным элементом этой модели является то, что все общение между актерами происходит асинхронно. Это предполагает, что отправитель не ожидает после отправки сообщения ответного сообщения. Вместо этого он сразу продолжает дальнейшее выполнение. Модель не определяет и не гарантирует, в каком порядке будут получены входящие сообщения актером-получателем, но гарантирует, что они будут доставлены.

Вторым важным свойством является тот факт, что все общение происходит исключительно посредством сообщений: не существует разделяемого между актерами состояния. Если один актер хочет получить информацию о внутреннем состоянии другого актера, ему придется отправить соответствующее сообщение чтобы запросить эту информацию. Актер-получатель, обработав письмо, отправит актеру-отправителю необходимую информацию также при помощи сообщения.

Такой подход позволяет актерам контролировать доступ к их внутреннему состоянию, избегая таких проблем, как проблема утерянных изменений. Все манипуляции внутренним состоянием выполняются при помощи сообщений.

Все актеры работают параллельно: их можно рассматривать как небольшие независимо выполняющиеся процессы.

Актеры в Scala доступны посредством библиотеки scala.actors. Их реализация – отличное доказательство выразительности языка Scala: вся функциональность, операторы и другие необходимые языковые конструкции реализованы на чистом Scala, как библиотека, без внесения изменений в сам язык Scala.

Тестовое приложение, в котором используются актеры, приведено на рис. 1. В этом приложении определен один актер, который выступает в роли простого счетчика.

```
import scala.actors.Actor
import scala.actors.Actor._

case class Inc(amount: Int)
case class Value

class Counter extends Actor {
    var counter: Int = 0;

    def act() = {
        while (true) {
            receive {
                case Inc(amount) =>
                    counter += amount
                case Value =>
                    println("Value is "+counter)
                    exit()
            }
        }
    }
}
```

Рис. 1. Пример параллельного приложения с актерами в Scala

В коде, который приведен на рис. 2, счетчику Counter посылаются сообщения, наращивающие счетчик, а потом посылается запрос распечатать значение счетчика.

```
object ActorTest extends Application {
    val counter = new Counter
    counter.start()

    for (i <- 0 until 100000) {
        counter ! Inc(1)
    }
    counter ! Value
    // Output: Value is 100000
}
```

Рис. 2. Пример использования актеров в Scala

Рассмотрим подробнее, как работают актеры в Scala. Класс Counter определен как подкласс класса Actor. В нем переопределяется метод act(), который инкапсулирует поведение актера. Состояние актера хранится в целочисленном поле counter. В методе act() выполняется бесконечный цикл обработки входящих писем. В данном примере, в методе act() обрабатываются все входящие письма: результатом обработки является либо изменение состояния актера, либо распечатка значения и выход.

В вызывающей программе создается объект счетчика, которому посылается 100000 сообщений Inc и в конце одно сообщение Value. Сообщения посылаются при помощи опрератора «!». Такая нотация была взята из языка Erlang.

В результате выполнения программы было распечатано число 100000. Это означает, что в данном случае все сообщения были доставлены в правильном порядке: сначала все сообщения Inc, потом сообщение Value. Нет гарантий, что так программа будет выполняться всегда: модель актеров не дает гарантий на счет порядки доставки сообщений.

Приведенный пример демонстрирует, насколько легко актеры могут использоваться в Scala несмотря на то, что они не поддерживаются встроенными конструкциями языка [4].

6. Безопасность параллелизма в Scala

Потенциальная опасность допустить ошибку при создании параллельных приложений на Scala истекает из того факта, что в Scala смешана модель актеров, изначально появившаяся в чистом функциональном языке, и объектно-ориентированное программирование. Актеры, имея ссылки на других актеров, могут получать доступ к их внутреннему состоянию не посредством отправки сообщений-запросов, а используя методы его интерфейса (т.е. через открытые методы). Доступ к внутреннему состоянию других актеров делает возможным изменение этого состояния непосредственно за счет вызова соответствующих методов. Кроме того, даже чтение внутреннего состояния одного актера другим переводит всю систему в неверное логическое состояние.

Язык Erlang, в силу своей функциональной природы, вынуждает актеров общаться лишь при помощи сообщений: в нем нет возможности получить или изменить информацию из других процессов (процессами в Erlang называются параллельно выполняющиеся задачи, выполняемые актерами).

Описанная проблема является главным компромиссом между Erlang и Scala: использование чисто функционального языка, которым является Erlang, безопасно, но более сложное в использовании для программистов [4]. Объектно-ориентированный, императивный стиль Scala является более знакомым и значительно облегчает разработку, но требует большей дисциплины и внимания, чтобы создавать безопасные и корректные программы.

Литература

1. Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software// http://www.gotw.ca/publications/concurrency-ddj.htm.
2. Debasish Ghosh. Scala Actors - Threadless and Scalable// http://java.dzone.com/articles/scala-threadless-concurrent.
3. Ward Cunningham. Shared State Concurrency// http://www.c2.com/cgi/wiki?SharedStateConcurrency.
4. Ruben Vermeersch. Concurrency in Erlang & Scala: The Actor Model// http://ruben.savanne.be/articles/concurrency-in-erlang-scalf.