УДК 004.434

ОБЗОР МЕТОДОВ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ, ОСНОВАННЫХ НА ОБМЕНЕ СООБЩЕНИЯМИ

Р.И. Попов
Научный руководитель – к.т.н., доцент А.Е. Платунов

Альтернативой потоковой модели с общей памятью является модель параллелизма, основанная на обмене сообщениями. В статье рассмотрены современные языки и методы программирования, ориентированные на обмен сообщениями. В качестве примера используются функциональный язык Erlang, разработанный для использования в телекоммуникационных системах и более традиционный (императивный) язык Scala, предназначенный для Java платформы.

Ключевые слова: параллельное программирование, обмен сообщениями, надежность, erlang, scala

Введение

Традиционно, в программировании существуют две модели параллелизма: с общей памятью (shared memory) и основанная на обмене сообщениями (message passing). Большинство существующих языков используют модель с общей памятью, как более близкую по своей семантике к машинным инструкциям. Лишь некоторые языки построены на основе обмена сообщениями, к этой группе относятся Erlang, Scala, Occam и Oz.

В модели параллелизма, основанной на обмене сообщениями, постулируется: общей памяти нет, все вычисления должны производятся в изолированных процессах. Единственный способ обмена информацией – асинхронный обмен сообщениями. Такой подход избавляет разработчика от трудно обнаруживаемых ошибок: взаимных блокировок и состояний гонок, свойственных потоковой модели. Помимо этого, обмен сообщениями между процессами дает дополнительные возможности для увеличения надежности и масштабируемости разрабатываемых систем.

На сегодняшний день разработано множество способов борьбы с взаимными блокировками и гонками в приложениях с общей памятью. Наверное, лучший и наиболее широко используемый из них – транзакционная память, и её современная реализация в виде популярных систем управления базами данных. Специалистами по параллельному программированию предложены множество паттернов и программных каркасов, решающих традиционные задачи распараллеливания вычислений. Тем не менее, общая память остается узким местом, ограничивающим производительность основанных на ней систем.

Уже сейчас становятся актуальными задачи создания распределенных систем, не попадающих под существующие шаблоны. Наш мир параллелен, и эта параллельность не основана на общей памяти. Мы обмениваемся сообщениями: звуковыми и световыми сигналами, обновляем наше внутреннее состояние и принимаем решения к действию на основе полученных данных. Природа – живой пример сложной распределенной системы, построенной на принципах обмена сообщениями. Муравейник и пчелиный рой, являются образцами систем, где множество достаточно простых организмов коллективно решают сложные задачи, не прибегая при этом к использованию общей памяти. Инженерам уже сейчас приходится решать подобные задачи, а с развитием искусственного интеллекта, сенсорных сетей, нанотехнологий их доля будет только увеличиваться.

Обзор языка Erlang

Язык, ориентированный на параллельное программирование (в том числе для распределенных систем) и реализующий модель обмена сообщениями должен отвечать следующим требованиям [1]:

- На уровне языка должна быть реализована модель управления процессами. Под процессом можно понимать программу, исполняющуюся в независимой виртуальной машине.
- Несколько процессов, выполняющихся на одном компьютере, должны быть строго изолированы. Ошибка в одном процессе не должна влиять на работу других процессов, только если это не предусмотрено явно.
- С каждым процессом должен быть связан уникальный идентификатор, используя который можно обращаться к данному процессу. Между процессами не должно быть общей памяти. Единственный способ взаимодействия – обмен сообщениями.
- Передача сообщений между процессами считается ненадежной, нет гарантии доставки сообщения.
- Если один из процессов прекращается с ошибкой, другие процессы должны иметь возможность получать уведомления об этом событии и причинах возникновения ошибки.

Разработчики языка Erlang называют подход, отвечающий этим требованиям, параллельно-ориентированным программированием (Concurrency Oriented Programming, COP). Математическая модель, основанная на этих положениях, называется *акторной моделью* [2]. Под актором понимается универсальная единица параллельных вычислений: в ответ на полученное сообщение актор принимает решения, создает другие акторы, посылает сообщения. Впервые акторная модель предложена в 1973 году Карлом Хьюиттом, профессором Массачусетского технологического института [3].

Сложные системы, реализованные в рамках этой парадигмы, могут состоять из тысяч параллельно выполняющихся процессов. Программистам, знакомым с параллельным программированием, известно, что создание процессов операционной системы дорого, также значительного времени требует переключение контекста процессов. Решение этой проблемы предложено в функциональном языке Erlang. Одна из основных концепций функционального программирования – неизменяющиеся данные. Переменная, назначенная один раз, уже не может быть переопределена. Таким образом, сама семантика функционального программирования защищает данные разных процессов, следовательно, несколько процессов могут работать в одном контексте. Создание процессов в Erlang очень дешево. На моем компьютере (Intel Core 2 Duo 2.3Ghz, 3Gb ОЗУ) в случае создания 200000 процессов, создание одного процесса занимает 2.7 мкс процессорного времени.

Язык Erlang поддерживает механизмы обмена сообщениями на уровне синтаксиса, что делает его использование гораздо более удобным, чем использование библиотечных реализаций обмена сообщениями, таких как MPI [4]. У каждого процесса в Erlang есть буфер (mailbox) для хранения принятых сообщений. Для отправки сообщения используется конструкция:

Pid ! Message

где Message – отправляемое сообщение, а Pid – идентификатор процесса, которому предназначается данное сообщение. Для приема сообщений используется следующая конструкция:

receive ... end

Для выборки сообщений из буфера используется проверка на соответствие шаблону. На рис. 1 процесс C принимает сообщения от процессов A и B. Сообщение типа 'foo' обладает приоритетом над сообщениями типа 'bar'.

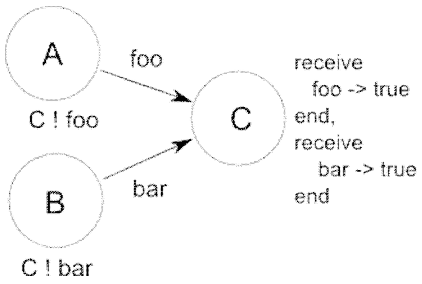

Рис. 1. Прием сообщений в соответствии с приоритетом

Чтобы процесс имел возможность ответить на сообщение, сообщение нужно снабжать адресом отправителя. На рис. 2 процесс A отправляет сообщение процессу B, процесс B передает сообщение от A процессу C. Процесс C отвечает процессу A.

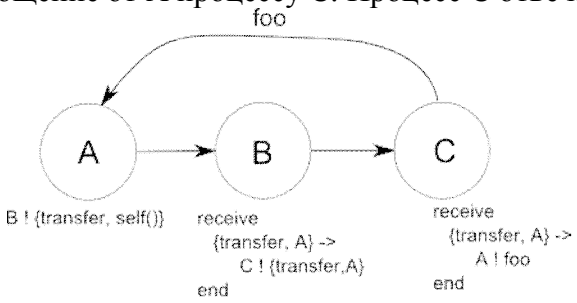

Рис. 2. Передача сообщений с адресом отправителя

В качестве примера программирования на Erlang рассмотрим решение «Santa Concurrency Problem» [5] предложенное Ричардом А. О'Кеффи [6].

Санта Клаус спит в своем доме на северном полюсе, и может быть разбужен, только если все девять северных оленей вернуться из своего отпуска с тропических островов в Тихом океане, или, если нескольким эльфам потребуется помощь с изготовлением игрушек. Если проблемы возникнут только у одного эльфа, тогда это не достаточно серьезная причина, чтобы будить Санту, поэтому эльфы всегда приходят к дому Санты втроем. В то время, пока трое эльфов решают вместе с Сантой свои проблемы, любые другие эльфы, которым также требуется помощь от Санты, должны ждать, пока вернуться предыдущие трое. Если Санта просыпается и обнаруживает, что три эльфа ждут его у двери, но в то же время последний из оленей вернулся из отпуска, он решает, что эльфы могут и подождать, ведь гораздо важнее приготовить сани к наступлению Рождества. (Предполагается, что олени не хотят покидать тропики, и пытаются оставаться там до последнего момента. Они бы рады совсем не возвращаться, но Санта оплачивает все их счета...)

Архитектура решения задачи на Erlang выглядит следующим образом:

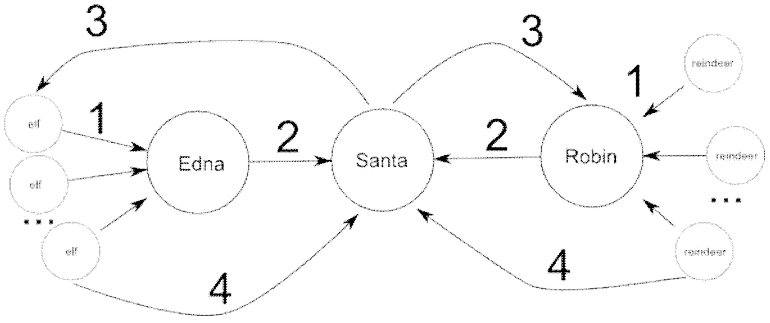

Рис. 3. Последовательность передачи сообщений в решении «Santa Concurrency Problem»

- Создается процесс Santa, который ожидает, пока кто-нибудь пришлет ему сообщение со списком оленей {reindeer,[R1,...,R9]} или эльфов {elves,[E1,E2,E3]}, собравшихся в группу. Используются две вложенных конструкции receive: первая принимает группу оленей, не зависимо от числа ожидающих эльфов, вторая, вложенная, принимает группы любого типа, т.к. может возникнуть такая ситуация, что сразу две группы оленей ожидают Санту. Санта ничего не знает о других процессах; он может только посылать сообщения оленям или эльфам, т.к. их идентификаторы находятся в принимаемых им сообщениях. После получения списка группы он отправляет свой идентификатор каждому члену группы и ждет от них ответа (барьерная синхронизация), после этого процесс повторяется.
- У Санты есть два секретаря (отдельные процессы). Эдна – секретарь, занимающийся эльфами. Когда она собирает группу из трех эльфов, она отправляет их список Санте. Робин, секретарь занимающийся оленями, собирает аналогичный список из девяти оленей.
- Каждый эльф это тоже процесс. Он посылает свой идентификатор Эдне и затем ожидает приглашения от Санты. После получения приглашения он совершает требуемые ему действия и отправляет Санте сообщение о том, что закончил. Эльф знает только идентификатор Эдны, остальные идентификаторы ему не известны.
- Олени отличаются от эльфов только тем, что знают идентификатор Робина, но не знают идентификатор Эдны.

Решение проблемы на Эрланге не требует никаких экзотических управляющих структур и структур данных, используется только простой обмен сообщениями. Для сравнения, решение в рамках потоковой модели, требует использования как минимум 10-ти семафоров.

Методы увеличения надежности

Первая версия языка Эрланг была разработана 1986–87 годах сотрудниками фирмы Ericsson для использования в телекоммуникационных системах. Телекоммуникации – одна из областей промышленности, где предъявляются повышенные требования к надежности систем. Коммуникационное оборудование должно годами работать без сбоев и дополнительного обслуживания. Поэтому, в язык Erlang изначально закладывались требования, характерные для телекоммуникационных систем:

- Необходимость обработки очень большого числа параллельных процессов.
- Некоторые действия должны выполняться точно в заданный момент времени, или за определенный временной промежуток. (Мягкое реальное время).
- Системы могут быть распределены на несколько вычислителей.
- Системы используются для управления аппаратурой.
- Очень большие объемы программного кода.
- Системы реализуют комплексную функциональность, существуют зависимости между функциями.
- Непрерывная работа в течение многих лет.
- Обслуживание программного обеспечения (обновление, конфигурирование) должно осуществляться без остановки системы;
- Устойчивость, как к аппаратным сбоям, так и к ошибкам в программном обеспечении.

На сегодняшний день, один из самых больших проектов, реализованных с использованием Erlang – ATM-коммутатор Erricson AXD301 (более миллиона строк кода). Разработчикам удалось достигнуть девяти девяток надежности (99.999999999%) [1]. Основной механизм надежности в Erlang – процессы супервизоры. Рассмотрим схему его работы.

Пары процессов в Erlang могут быть связаны вместе. Если один из процессов в паре погибает, тогда другой получит сообщение, содержащее причины гибели первого. На основании полученного сообщения оставшийся процесс может принять решения по восстановлению системы после случившегося сбоя. Хороший стиль программирования на Erlang предполагает разделение процессов на два класса: рабочих и супервизоров. Процессы-рабочие выполняют основную функциональность системы, супервизоры следят за состоянием системы и перезапускают процессов-рабочих после сбоев. При использовании кэширования данных становится возможным восстановление процесса без потерь. Для достижения максимальной надежности разработчик может строить иерархии (деревья) супервизоров, которые могут восстанавливать после сбоев целые подсистемы.

Реализация обмена сообщениями в императивных языках

В своей статье «Problem with threads» Эдвард А. Ли отмечает, что, несмотря на существование в течении долгого времени моделей альтернативных потокам, программисты раз за разом выбирают именно потоки для решения своих задач [7]. Прежде всего, это связано с тем, что сам дух программирования, все ключевые абстракции программирования связаны с парадигмой последовательных вычислений. Большинство используемых сегодня языков программирования реализуют именно эту парадигму. Синтаксически потоки являются незначительным расширением для таких языков, или даже могут быть реализованы в виде отдельных библиотек. При этом, к сожалению, семантически потоки полностью разрушают детерминизм последовательных вычислений.

Другая причина популярности императивных языков заключается в высокой эффективности их исполнения в рамках существующих архитектур микропроцессоров. Хотя, по своей природе функциональные (декларативные) языки, такие как Erlang, значительно ближе к семантике модели дискретных событий, в рамках которой разрабатывается современная цифровая аппаратура, прямой аппаратный синтез с таких языков остается невозможным, из-за невыполнимых требований к емкости кристаллов микросхем. Другой вариант – разработка архитектуры микропроцессоров, оптимизированных для исполнения функциональных программ. Такие процессоры находят свое применение в специальных областях.

Эдвард Ли видит решение проблемы в использовании координационных языков, как надстройки над существующими широко применяемыми языками. Задачей координационного языка является организация обмена данными между модулями (акторами), описанными на традиционных императивных языках, таких как Си или Java. Реализацией такого подхода является язык Scala и построенный на базе него координационный язык Reo [8].

Scala – мультипарадигменный язык программирования, сочетающий в себе возможности объектно-ориентированного и функционального программирования. Программы, разработанные на Scala, компилируются в код виртуальной машины JVM, что позволяет использовать в разработке существующие классы Java платформы. Одной из особенностей Scala является то, что в отличие от Java, он не использует потоки и общую память для организации параллельных вычислений. Вместо этого в Scala используется механизм обмена сообщениями, аналогичный используемому в Erlang'e. Аналогом потока в Scala является актор. Каждый актор реализует конструкции для отправки и получения сообщений и буфер для принятых сообщений. На базе библиотеки акторов Scala можно легко реализовывать координационные языки для Java платформы.

Выводы

Модель параллелизма, основанная на обмене сообщениями, обладает большими перспективами в области построения надежных и высокоэффективных распределенных систем. Уже сейчас, такие проекты как yaws и ejabberd доказывают состоятельность этого подхода. С развитием многоядерных процессоров и SMP систем, интерес разработчиков к языкам, ориентированным на эту парадигму, будет расти. В области распределенных встраиваемых систем популярность могут завоевать специализированные процессоры и SoC, аппаратно поддерживающие методы обмена сообщениями между процессами.

Литература

1. Armstrong Joe. Making reliable distributed systems in the presence of software errors.// Ph.D. Dissertation, 2003. The Royal Institute of Technology, Stockholm, Sweden.
2. Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. // Doctoral Dissertation. – 1986. MIT Press.
3. Carl Hewitt; Peter Bishop; Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. // 1973. – IJCAI, Stanford.
4. William Gropp; Ewing Lusk; Anthony Skjellum. Using MPI, 2nd Edition: portable Parallel Programming with the Message Passing Interface. // 1999, MIT Press In Scientific And Engineering Computation Series, Cambridge, MA. – USA. – 395 pp.
5. J.A. Trono. A new exercise in concurrency // SIGCSE Bulletin. – 1994. – Vol. 26. – PP. 8–10.
6. Richard O'Keefe Solving the Santa Claus problem in Erlang.// http://www.cs.otago.ac.nz/staffpriv/ok/santa/index.htm
7. Edward A. Lee. Problem with Threads // 2006, Technical Report No. UCB/EECS-2006-1, EECS Dept. University of California Berkeley.
8. Philipp Haller; Martin Odersky, Actors that Unify Threads and Events. // 2007, Technical report LAMP-REPORT-2007-001. École Polytechnique Fédérale de Lausanne.
9. Edward A. Lee. Disciplined Message Passing // 2009, Technical report No. UCB/EECS-2009-7, EECS Dept. University of California Berkeley.