УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ П.М. МАШЕРОВА»

Факультет математики и информационных технологий

Кафедра прикладного и системного программирования

Согласовано
Заведующий кафедрой
_____ С.А. Ермоченко
07.06.2021

Согласовано
Декан факультета
_____ Е.Н. Залесская
07.06.2021

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС
ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

РАСПРЕДЕЛЕННЫЕ И ПАРАЛЛЕЛЬНЫЕ СИСТЕМЫ

для направления специальности I ступени высшего образования

1-31 03 07-01 Прикладная информатика (программное обеспечение компьютерных систем)

Составитель: С.В. Сергеенко

Рассмотрено и утверждено
на заседании научно-методического совета 29.06.2021, протокол № 7

УДК 004.75:004.4(075.8)
ББК 16.262я73+32.973я73
Р24

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 4 от 18.02.2021.

Составитель: старший преподаватель кафедры прикладного и системного программирования ВГУ имени П.М. Машерова **С.В. Сергеенко**

Р е ц е н з е н т ы :
заведующий кафедрой «Информационные системы и автоматизация производства» УО «ВГТУ», кандидат технических наук, доцент *В.Е. Казаков*;
доцент кафедры информационных технологий и управления бизнесом ВГУ имени П.М. Машерова, кандидат физико-математических наук *С.А. Прохожий*

Распределенные и параллельные системы для направле-
Р24 ния специальности I ступени высшего образования 1-31 03 07-01 Прикладная информатика (программное обеспечение компьютерных систем) : учебно-методический комплекс по учебной дисциплине / сост. С.В. Сергеенко. – Витебск : ВГУ имени П.М. Машерова, 2021. – 140 с.
ISBN 978-985-517-800-3.

Учебно-методический комплекс по дисциплине «Распределенные и параллельные системы» для направления специальности I ступени высшего образования 1-31 03 07-01 Прикладная информатика (программное обеспечение компьютерных систем) предоставляет студентам материал для обучения будущих специалистов по разработке программного обеспечения методам распараллеливания вычислений с использованием мощных вычислительных систем с распределенной и общей памятью.

В данный комплекс входят фрагменты учебной программы учреждения образования по учебной дисциплине и методические указания. В нем приводятся теоретические сведения, вопросы для самоконтроля, указания по выполнению лабораторных работ, примерные вопросы задания для текущей аттестации и контрольных работ, предусмотренных учебным планом, а также перечень рекомендуемой литературы.

УДК 004.75:004.4(075.8)
ББК 16.262я73+32.973я73

ISBN 978-985-517-800-3

СОДЕРЖАНИЕ

1 Теоретические сведения

1.1 Введение

1.1.1 Применение многопроцессорных вычислительных систем, особенности разработки программ для них

До недавнего времени рост производительности обеспечивался во многом повышением тактовой частоты основных вычислительных элементов компьютеров – процессоров. Но возможности такого подхода оказались не безграничными – после некоторого рубежа дальнейшее увеличение тактовой частоты требует значительных технологических усилий, сопровождается существенным ростом энергопотребления и наталкивается на непреодолимые проблемы теплорегуляции. В таких условиях практически неизбежным явилось кардинальное изменение основного принципа производства компьютерной техники – вместо создания сложных высокочастотных «монолитных» процессоров теперь большинство производителей разрабатывают «составные» процессоры, состоящих из множества равноправных и сравнительно простых вычислительных элементов – ядер. Максимальная производительность процессоров в этом случае является равной сумме производительности вычислительных ядер, входящих в процессоры. Тем самым, размещая в рамках процессоров все большее количество ядер, можно добиваться роста производительности без повышения тактовых частот.

При этом важно понимать, что переход к многоядерности одновременно знаменует и наступление эры параллельных вычислений. На самом деле, задействовать вычислительный потенциал многоядерных процессоров можно только, если осуществить разделение выполняемых вычислений на информационно независимые части и организовать выполнение каждой части вычислений на разных ядрах. Подобный подход позволяет выполнять необходимые вычисления с меньшими затратами времени, и возможность получения максимального ускорения ограничивается только числом имеющихся ядер и количеством «независимых» частей в выполняемых вычислениях. Параллельные вычисления становятся неизбежными и повсеместными.

Однако необходимость организации параллельных вычислений приводит к повышению сложности эффективного применения многоядерных компьютерных систем. Разработка параллельной программы для программиста, привыкшего к последовательной модели программирования, может оказаться на первых порах непростым делом. Ведь в этом случае придется задуматься не только о привычных вещах, таких как структуры данных, последовательность их обработки, интерфейсы, но и об обменах данными между подзадачами, входящими в состав параллельного приложения, об их согласованном выполнении и так далее. Для проведения параллельных вычислений необходимо «параллельное» обобщение традиционной –

последовательной – технологии решения задач на ЭВМ. Так, численные методы в случае многоядерности должны проектироваться как системы параллельных и взаимодействующих между собой процессов, допускающих исполнение на независимых вычислительных ядрах. Применяемые алгоритмические языки и системное программное обеспечение должны обеспечивать создание параллельных программ, организовывать синхронизацию и взаимоисключение асинхронных процессов и тому подобное.

Все перечисленные проблемы организации параллельных вычислений увеличивают существующий разрыв между вычислительным потенциалом современных компьютерных систем и имеющимся алгоритмическим и программным обеспечением применения компьютеров для решения сложных задач. И, как результат, устранение или, по крайней мере, сокращение этого разрыва является одной из наиболее значимых задач современной науки и техники.

В отличие от централизованных алгоритмов, распределенные алгоритмы обладают следующими свойствами, которые на самом деле значительно усложняют их проектирование и реализацию:

– Отсутствие знания глобального состояния. Централизованные алгоритмы обладают полной информацией о состоянии всей системы и определяют следующие действия, исходя из ее текущего состояния. В свою очередь, каждый процесс, реализующий часть распределенного алгоритма, имеет непосредственный доступ только к своему состоянию, но не к глобальному состоянию всей системы.

– Отсутствие общего единого времени. События, составляющие ход выполнения централизованного алгоритма полностью упорядочены: для любой пары событий можно с уверенностью утверждать, что одно из них произошло раньше другого. При выполнении распределенного алгоритма вследствие отсутствия единого для всех процессов времени, события нельзя считать полностью упорядоченными: для некоторых пар событий мы можем утверждать, какое из них произошло раньше другого, для других – нет.

– Отсутствие детерминизма. Выполнение распределенного алгоритма носит недетерминированный характер из-за независимого исполнения процессов с различной и неизвестной скоростью, а также из-за случайных задержек передачи сообщений между ними. Поэтому выполнение распределенного алгоритма может лишь ограниченно рассматриваться как переход из одного глобального состояния в другое, так как для этого же алгоритма выполнение может быть описано другой последовательностью глобальных состояний. Такие альтернативные последовательности обычно состоят из других глобальных состояний, и поэтому нет особого смысла говорить о том, что то или иное состояние достигается по ходу выполнения распределенного алгоритма.

– Устойчивость к отказам. Сбой в любом из процессов или каналов связи не должен вызывать нарушения работы распределенного алгоритма.

1.1.2 Общие принципы построения распределенных и параллельных систем

В общем плане под параллельными вычислениями понимаются процессы обработки данных, в которых одновременно могут выполняться несколько операций компьютерной системы. Достижение параллелизма возможно только при выполнении следующих требований к архитектурным принципам построения вычислительной среды:

– независимость функционирования отдельных устройств ЭВМ;

– избыточность элементов вычислительной системы.

Требование независимости функционирования относится в равной степени ко всем основным компонентам вычислительной системы: к устройствам ввода-вывода, обрабатывающим процессорам и устройствам памяти.

Организация избыточности может осуществляться в следующих основных формах:

– использование специализированных устройств, таких, например, как отдельные процессоры для целочисленной и вещественной арифметики, устройства многоуровневой памяти (регистры, кэш);

– дублирование устройств ЭВМ путем использования, например, нескольких однотипных обрабатывающих процессоров или нескольких устройств оперативной памяти.

Дополнительной формой обеспечения параллелизма может служить конвейерная реализация обрабатывающих устройств, при которой выполнение операций в устройствах представляется в виде исполнения последовательности составляющих операцию подкоманд. Как результат, при вычислениях на таких устройствах на разных стадиях обработки могут находиться одновременно несколько различных элементов данных.

При рассмотрении проблемы организации параллельных вычислений следует различать следующие возможные режимы выполнения независимых частей программы:

– многозадачный режим (режим разделения времени), при котором для выполнения нескольких процессов используется единственный процессор;

– параллельное выполнение, когда в один и тот же момент может выполняться несколько команд обработки данных;

– распределенные вычисления – данный термин обычно применяют для указания параллельной обработки данных, при которой используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам.

Многозадачный режим является псевдопараллельным, когда активным (исполняемым) может быть один, единственный процесс, а все остальные процессы находятся в состоянии ожидания своей очереди;

применение режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидания вводимых данных, процессор может быть задействован для выполнения другого, готового к исполнению процесса). Кроме того, в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и другие), и, как результат, этот режим может быть использован при начальной подготовке параллельных программ.

Режим параллельного выполнения вычислений может быть обеспечен не только при наличии нескольких процессоров, но и при помощи конвейерных и векторных обрабатывающих устройств.

Эффективная обработка данных при распределенном способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных. Перечисленные условия являются характерными, например, при организации вычислений в многомашинных вычислительных комплексах, образуемых объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

В литературе можно найти различные определения распределенных систем, причем ни одно из них не является удовлетворительным и не согласуется с остальными. Для наших задач хватит достаточно вольной характеристики.

Распределенная система – это набор независимых компьютеров, представляющийся их пользователям единой объединенной системой.

В этом определении оговариваются два момента. Первый относится к аппаратуре: все машины автономны. Второй касается программного обеспечения: пользователи думают, что имеют дело с единой системой. Важны оба момента.

Возможно, вместо того чтобы рассматривать определения, разумнее будет сосредоточиться на важных характеристиках распределенных систем. Первая из таких характеристик состоит в том, что от пользователей скрыты различия между компьютерами и способы связи между ними. То же самое относится и к внешней организации распределенных систем. Другой важной характеристикой распределенных систем является способ, при помощи которого пользователи и приложения единообразно работают в распределенных системах, независимо от того, где и когда происходит их взаимодействие.

Распределенные системы должны также относительно легко поддаваться расширению, или масштабированию. Эта характеристика является прямым следствием наличия независимых компьютеров, но в то же время не указывает, каким образом эти компьютеры на самом деле объединяются в единую систему. Распределенные системы обычно существуют постоянно, однако некоторые их части могут временно выходить из строя.

Пользователи и приложения не должны уведомляться о том, что эти части заменены или починены или что добавлены новые части для поддержки дополнительных пользователей или приложений.

Для того чтобы поддержать представление различных компьютеров и сетей в виде единой системы, организация распределенных систем часто включает в себя дополнительный уровень программного обеспечения, находящийся между верхним уровнем, на котором находятся пользователи и приложения, и нижним уровнем, состоящим из операционных систем. Соответственно, такая распределенная система обычно называется системой промежуточного уровня (middleware).

1.1.2 Примеры распределенных и параллельных систем

Взглянем теперь на некоторые примеры распределенных систем. В качестве первого примера рассмотрим сеть рабочих станций в университете или отделе компании. Вдобавок к персональной рабочей станции каждого из пользователей имеется пул процессоров машинного зала, не назначенных заранее ни одному из пользователей, но динамически выделяемых им при необходимости. Эта распределенная система может обладать единой файловой системой, в которой все файлы одинаково доступны со всех машин с использованием постоянного пути доступа. Кроме того, когда пользователь набирает команду, система может найти наилучшее место для выполнения запрашиваемого действия, возможно, на собственной рабочей станции пользователя, возможно, на простаивающей рабочей станции, принадлежащей кому-то другому, а может быть, и на одном из свободных процессоров машинного зала. Если система в целом выглядит и ведет себя как классическая однопроцессорная система с разделением времени (то есть многопользовательская), она считается распределенной системой. В качестве второго примера рассмотрим работу информационной системы, которая поддерживает автоматическую обработку заказов. Обычно подобные системы используются сотрудниками нескольких отделов, возможно в разных местах. Так, сотрудники отдела продаж могут быть разбросаны по обширному региону или даже по всей стране. Заказы передаются с переносных компьютеров, соединяемых с системой при помощи телефонной сети, а возможно, и при помощи сотовых телефонов. Приходящие заказы автоматически передаются в отдел планирования, превращаясь там во внутренние заказы на поставку, которые поступают в отдел доставки, и в заявки на оплату, поступающие в бухгалтерию. Система автоматически пересылает эти документы имеющимся на месте сотрудникам, отвечающим за их обработку. Пользователи остаются в полном неведении о том, как заказы на самом деле курсируют внутри системы, для них все это представляется так, будто вся работа происходит в централизованной базе данных.

Высокопроизводительные вычисления необходимы для ряда важнейших задач, в числе которых задачи прогнозирования погоды и климата в целом, и в особенности, их катастрофических изменений (возникновения ураганов, тайфунов, резких изменений температуры и тому подобное), задачи геологии и геофизики (предсказания землетрясений, вулканических извержений и так далее), задачи астрономии и астрофизики (предсказания поведения Солнца, столкновений метеоритов и болидов с Землей, обоснование космогонических гипотез), задачи, связанные с биологией и с чрезвычайно значимой для человека областью – с медициной.

В качестве последнего примера рассмотрим World Wide Web. Web предоставляет простую, целостную и единообразную модель распределенных документов. Чтобы увидеть документ, пользователю достаточно активизировать ссылку. После этого документ появляется на экране. В теории (но определенно не в текущей практике) нет необходимости знать, с какого сервера доставляется документ, достаточно лишь информации о том, где он расположен. Публикация документа очень проста: вы должны только задать ему уникальное имя в форме унифицированного указателя ресурса, которое ссылается на локальный файл с содержимым документа. Если бы Всемирная паутина представлялась своим пользователям гигантской централизованной системой документооборота, она также могла бы считаться распределенной системой. К сожалению, этот момент еще не наступил. Так, пользователи сознают, что документы находятся в различных местах и распределены по различным серверам.

1.1.3 Предъявляемые к распределенным и параллельным системам требования

Эффективная распределенная система должна обладать следующими свойствами: *прозрачность*, *открытость*, *безопасность*, *масштабирование*. Однако стоит отметить, что, несмотря на кажущуюся простоту и очевидность перечисленных свойств, их реализация на практике часто представляет собой непростую задачу.

Под **прозрачностью** распределенной системы понимают ее способность скрывать свою распределенную природу и представляться в виде единой централизованной компьютерной системы. Наиболее важные из типов прозрачности перечислены ниже.

– *Прозрачность доступа* – обращение к локальным и удаленным ресурсам осуществляется одинаковым образом.

– *Прозрачность местоположения* позволяет обращаться к ресурсам без знания их физического местоположения.

– *Прозрачность перемещения* – перемещение ресурса или процесса в другое физическое местоположение остается незаметным для пользователя распределенной системы.

– *Прозрачность смены местоположения* – требование скрыть факт перемещения ресурса во время его использования.

– *Прозрачность репликации* – факт использования нескольких копий ресурса (реплик) остается скрытым от пользователя.

– *Прозрачность одновременного доступа* позволяет нескольким конкурирующим процессам одновременно выполнять операции над совместно используемым ресурсом без взаимного влияния друг на друга.

– *Прозрачность отказов* – система должна пытаться скрывать частичные отказы, позволяя приложениям выполнить свою работу вне зависимости от сбоев в компонентах распределенной системы, а также факт их последующего восстановления.

Важно отметить, что степень, до которой каждое этих свойств должно быть выполнено, может сильно варьироваться в зависимости от задач построения распределенной системы. Не каждая система в состоянии или даже должна пытаться скрывать все свои особенности от пользователя. Обычно, это утверждение выражается в поиске компромисса между прозрачностью распределенной системы и ее производительностью.

Открытая система – это система, реализующая открытые спецификации на интерфейсы, службы и поддерживаемые форматы данных, достаточные для того, чтобы обеспечить:

– возможность переноса разработанного прикладного программного обеспечения на широкий диапазон систем с минимальными изменениями;

– взаимодействие с другими прикладными приложениями на локальных и удаленных платформах;

– взаимодействие с пользователями в стиле, облегчающим последним переход от системы к системе.

Под открытой спецификацией понимается общедоступная спецификация, которая поддерживается открытым, гласным согласительным процессом, направленным на постоянную адаптацию к новым технологиям.

Синтаксис интерфейсов, то есть имена доступных функций, типы передаваемых параметров, возвращаемых значений и тому подобное, обычно описывается посредством языка определения интерфейсов (IDL). Семантика интерфейсов, то есть то, что на самом деле делают службы, предоставляющие эти интерфейсы, обычно задается неформально, с помощью естественного языка.

Открытость системы позволяет создавать несколько различных реализаций одной и той же службы, работающих с точки зрения внешних процессов одинаково. Из-за этого разные реализации программных компонентов могут взаимодействовать, образуя единую распределенную систему. Так достигается свойство *интероперабельности* – способности к взаимодействию. При этом прикладное приложение, разработанное для одной распределенной системы, может без изменений выполняться в другой

распределенной системе, реализующей те же интерфейсы. То есть, достигается свойство *переносимости*.

Еще одно важное преимущество заключается в *гибкости* – легкости конфигурирования системы, состоящей из различных компонентов, возможно от разных производителей. То есть добавление новых компонентов или замена существующих может осуществляться, не затрагивая других компонентов. Другими словами, важным свойством открытой распределенной системы является *расширяемость*.

В построении таких систем решающим фактором оказывается организация этих систем в виде наборов относительно небольших и легко заменяемых или адаптируемых компонентов. Что влечет необходимость определения не только интерфейсов, с которыми работают пользователи и приложения, но также и интерфейсов внутренних модулей системы.

В общем случае **масштабируемость** определяют, как способность вычислительной системы эффективно справляться с увеличением числа пользователей или поддерживаемых ресурсов без потери производительности и без увеличения административной нагрузки на ее управление.

Для распределенных систем обычно выделяют несколько параметров, характеризующих их масштаб: размер, степень территориальной рассредоточенности компонентов системы и количество административных организаций, обслуживающих части распределенной системы. Поэтому масштабируемость распределенных систем также определяют по соответствующим направлениям:

– Нагрузочная масштабируемость – способность системы увеличивать свою производительность при увеличении нагрузки путем замены существующих аппаратных компонентов на более мощные (вертикальное масштабирование) или путем добавления новых аппаратных средств (горизонтальное масштабирование).

– Географическая масштабируемость – способность системы сохранять свои основные характеристики при территориальном разнесении ее компонентов.

– Административная масштабируемость характеризует простоту управления системой при увеличении количества административно независимых организаций, обслуживающих части одной распределенной системы.

Одна из основных причин плохой географической масштабируемости многих распределенных систем, разработанных для локальных сетей, заключается в использовании *синхронной связи*. В этом виде связи клиент, вызывающий какую-либо службу сервера, блокируется до получения ответа. Это неплохо работает, когда взаимодействие между процессами происходит быстро и незаметно для пользователя. Однако при увеличении задержки на обращение к удаленной службе в глобальной системе подобный подход становится все менее привлекательным и, очень часто, абсолютно неприемлемым.

Другая сложность обеспечения географической масштабируемости состоит в том, что связь в глобальных сетях по своей природе ненадежна и взаимодействие процессов практически всегда является двухточечным. А в локальных сетях связь является высоконадежной и подразумевает использование широковещательных сообщений, что значительно упрощает разработку распределенных приложений.

В большинстве случаев сложности масштабирования проявляются в проблемах с эффективностью функционирования распределенных систем, вызванных ограниченной производительностью ее отдельных компонентов: серверов и сетевых соединений. Существуют несколько основных технологий, позволяющих уменьшить нагрузку на каждый компонент распределенной системы: распространение, репликацию и кэширование.

Распространение предполагает разбиение множества поддерживаемых ресурсов на части с последующим разнесением этих частей по компонентам системы.

Репликация не только повышает доступность ресурсов в случае возникновения частичного отказа, но и помогает балансировать нагрузку между компонентами системы.

Кэширование представляет собой особую форму репликации, когда копия ресурса создается в непосредственной близости от пользователя, использующего этот ресурс. Разница заключается лишь в том, что репликация инициируется владельцем ресурса, а кэширование – пользователем при обращении к этому ресурсу.

Однако, наличие нескольких копий ресурса приводит к необходимости обеспечивать их *непротиворечивость*, что, в свою очередь, может отрицательно сказаться на масштабируемости и производительности системы.

Вопросы для самоконтроля

1. Каковы причины распространения многопроцессорных (многоядерных) вычислительных систем?
2. Какие дополнительные факторы надо учитывать при разработке параллельных программ?
3. Какие свойства распределенных алгоритмов усложняют их проектирование и разработку?
4. Какие архитектурные принципы построения вычислительной среды являются необходимыми для достижения параллелизма?
5. Назовите основные формы организации избыточности элементов вычислительной среды.
6. Опишите режимы выполнения независимых частей программы.
7. Какова роль программного обеспечения промежуточного уровня в распределенных системах?
8. Укажите примеры распределенных систем?

9. Назовите задачи, требующие применения распределенных и параллельных систем?

10. Что понимают под прозрачностью распределенной системы? Приведите примеры различных видов прозрачности.

11. Какая из характеристик распределенной системы может ухудшаться при повышении степени прозрачности?

12. Что такое открытая распределенная система? Какие преимущества дает открытость?

13. Что такое масштабируемость распределенной системы?

14. Назовите основные проблемы при построении масштабируемых систем.

15. Как можно добиться масштабируемости?

1.2 Введение в параллельные вычисления

1.2.1 Классификации параллельных архитектур по Флинну и Хокни

Существует много различных способов организации параллельных вычислительных систем. Здесь можно назвать векторно-конвейерные компьютеры, массивно-параллельные и матричные системы, компьютеры с многопоточной архитектурой, систолические массивы, dataflow-компьютеры и тому подобное. Если же к подобным названиям для полноты описания добавить и сведения о таких важных параметрах, как организация памяти, топология связи между процессорами, синхронность работы отдельных устройств или способ исполнения операций, то число различных архитектур станет и вовсе необозримым.

Активные попытки в классификации архитектур вычислительных систем начались после опубликования **М. Флинном** в конце 60-х годов прошлого столетия первого варианта классификации, который, кстати, используется и в настоящее время. Классификация базируется на понятии потока, под которым понимается последовательность команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

SISD (single instruction stream / single data stream) – одиночный поток команд и одиночный поток данных. К этому классу относятся машины, где есть только один поток команд, все команды обрабатываются последовательно, и каждая команда инициирует одну операцию с одним потоком данных. Для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка – как машина со скалярными функциональными устройствами, так и с конвейерными попадают в этот класс.

SIMD (single instruction stream / multiple data stream) – одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от

предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными – элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производится либо процессорной матрицей, либо с помощью конвейера.

MISD (multiple instruction stream / single data stream) – множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе. Ряд исследователей относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе. Будем считать, что пока данный класс пуст.

MIMD (multiple instruction stream / multiple data stream) – множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

Данная классификация имеет явные недостатки. В частности, некоторые заслуживающие внимания архитектуры, например dataflow и векторно-конвейерные машины, четко не вписываются в данную классификацию. Другой недостаток – архитектуры, которые по Флинну попадают в класс MIMD, совершенно различны по числу процессоров, природе и топологии связи между ними, по способу организации памяти и, конечно же, по технологии программирования.

Наличие пустого класса (MISD) не стоит считать недостатком схемы. Такие классы могут стать чрезвычайно полезными для разработки принципиально новых концепций в теории и практике построения вычислительных систем.

Р. Хокни – известный английский специалист в области параллельных вычислительных систем, разработал свой подход к классификации, введенной им для систематизации компьютеров, попадающих в класс MIMD по систематике Флинна.

Основная идея классификации состоит в следующем. Множественный поток команд может быть обработан двумя способами: либо одним *конвейерным* устройством обработки, работающем в режиме разделения времени для отдельных потоков, либо каждый поток обрабатывается своим собственным устройством. Первая возможность используется в MIMD компьютерах, которые автор называет конвейерными. Архитектуры, использующие вторую возможность, в свою очередь опять делятся на два класса:

– MIMD компьютеры, в которых возможна прямая связь каждого процессора с каждым, реализуемая с помощью *переключателя*;

– MIMD компьютеры, в которых прямая связь каждого процессора возможна только с ближайшими соседями *по сети*, а взаимодействие удаленных процессоров поддерживается специальной системой маршрутизации через процессоры-посредники.

Далее, среди MIMD машин с переключателем Хокни выделяет те, в которых вся память распределена среди процессоров как их локальная память. В этом случае общение самих процессоров реализуется с помощью очень сложного переключателя, составляющего значительную часть компьютера. Такие машины носят название MIMD машин *с распределенной памятью*. Если память является разделяемым ресурсом, доступным всем процессорам через переключатель, то такие MIMD являются системами *с общей памятью*. В соответствии с типом переключателей можно проводить классификацию и далее: простой переключатель, многокаскадный переключатель, общая шина.

Многие современные вычислительные системы имеют как общую разделяемую память, так и распределенную локальную. Такие системы можно рассматривать как гибридные MIMD с переключателем.

При рассмотрении MIMD машин с сетевой структурой считается, что все они имеют распределенную память, а дальнейшая классификация проводится в соответствии с топологией сети: звездообразная сеть, регулярные решетки разной размерности, гиперкубы, сети с иерархической структурой, такой, как деревья, пирамиды, кластеры и, наконец, сети, изменяющие свою конфигурацию.

Заметим, что если архитектура компьютера спроектирована с использованием нескольких сетей с различной топологией, то, по всей видимости, по аналогии с гибридными MIMD с переключателями, их стоит назвать гибридными сетевыми MIMD, а использующие идеи разных классов – просто гибридными MIMD.

1.2.2 Параллельная обработка данных

Параллельная обработка данных предполагает наличие нескольких функционально независимых устройств.

Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени.

В зависимости от организации данных и взаимодействия операций в распределенном алгоритме, выполняется пространственно-временная классификация параллелизма. Различают два основных вида параллельного поведения распределенной системы:

– параллелизм в пространстве;

– параллелизм во времени.

Параллелизм в пространстве свойственен системам, одновременно обрабатывающим один входной набор данных таким образом, что все операции алгоритма принимают на входе элементы этого набора, либо значения, являющиеся производными от этих элементов, полученные операциями-предшественниками. В конечном счете, это приводит к вычислению результирующих значений выходного набора. Операции, выполняющиеся параллельно в пространстве, взаимно не предшествуют друг другу и являются информационно независимыми.

Параллелизм во времени свойственен системам, обрабатывающим поток данных, состоящий из последовательности входных наборов данных. Параллелизм во времени называется также конвейерным параллелизмом. Конвейер состоит из ступеней, выполняющих преобразование информации и взаимодействующих таким образом, что выходные данные одной ступени являются входными данными последующей ступени. Ступени соединены последовательно, однако все они работают параллельно на различных наборах данных, которые шаг за шагом проталкиваются через конвейер. Число одновременно обрабатываемых наборов равно числу ступеней конвейера. Более того, порядок обработки последующего набора может зависеть от результатов обработки предыдущих наборов.

В системе со *смешанным параллелизмом* одновременно присутствует параллелизм в пространстве и параллелизм во времени. Параллелизм в пространстве реализуется внутри каждой ступени, параллелизм во времени проявляется в одновременном выполнении операций на разных ступенях.

1.2.3 Конвейерные вычисления. Синхронные и асинхронные архитектуры

Что необходимо для сложения двух вещественных чисел с плавающей запятой? Целое множество мелких операций, таких как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и тому подобное. Процессоры первых компьютеров выполняли все эти «микрооперации» для каждой пары аргументов последовательно одна за другой до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых.

Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передает результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно

неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если же каждую микрооперацию выделить в отдельный этап (или иначе говорят – ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находиться первые пять пар аргументов, первый результат будет получен через 5 единиц времени, каждый следующий – через одну единицу после предыдущего, а весь набор из ста пар будет обработан за 5+99=104 единицы времени, то есть будет получено ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера).

Приблизительно так же будет и в общем случае. Если конвейерное устройство содержит L ступеней, а каждая ступень срабатывает за одну единицу времени, то время обработки n независимых операций этим устройством составит L + n – 1 единиц. Если это же устройство использовать в монопольном режиме (как последовательное), то время обработки будет равно L × n. В результате получим ускорение почти в L раз за счет использования конвейерной обработки данных.

Казалось бы, конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. Однако стоимость и сложность получившейся системы будет несопоставима со стоимостью и сложностью конвейерного варианта, а производительность будет почти такой же.

Значительное преимущество конвейерной обработки перед последовательной имеет место в идеальном конвейере, в котором отсутствуют конфликты и все команды выполняются друг за другом без перезагрузки конвейера. Наличие конфликтов снижает реальную производительность конвейера по сравнению с идеальным случаем.

Конфликты – это такие ситуации в конвейерной обработке, которые препятствуют выполнению очередной команды в предназначенном для нее такте. Конфликты делятся на три группы:

– структурные конфликты;

– конфликты по управлению;

– конфликты по данным.

Структурные конфликты возникают в том случае, когда аппаратные средства процессора не могут поддерживать все возможные комбинации команд в режиме одновременного выполнения с совмещением.

Конфликты по управлению возникают при конвейеризации команд переходов и других команд, изменяющих значение счетчика команд.

Наиболее эффективным методом снижения потерь от конфликтов по управлению служит предсказание переходов. Суть данного метода заключается в том, что при выполнении команды условного перехода специальный блок микропроцессора определяет наиболее вероятное направление перехода, не дожидаясь формирования признаков, на основании анализа

которых этот переход реализуется. Процессор начинает выбирать из памяти и выполнять команды по предсказанной ветви программы (так называемое исполнение по предположению, или «спекулятивное» исполнение). Однако так как направление перехода может быть предсказано неверно, то получаемые результаты с целью обеспечения возможности их аннулирования не записываются в память или регистры, а накапливаются в специальном буфере результатов.

Если после формирования анализируемых признаков оказалось, что направление перехода выбрано верно, все полученные результаты переписываются из буфера по месту назначения, а выполнение программы продолжается в обычном порядке. Если направление перехода предсказано неверно, то буфер результатов очищается. Также очищается и конвейер, содержащий команды, находящиеся на разных этапах обработки, следующие за командой условного перехода. При этом аннулируются результаты всех уже выполненных этапов этих команд. Конвейер начинает загружаться с первой команды другой ветви программы.

Конфликты по данным возникают в случаях, когда выполнение одной команды зависит от результата выполнения предыдущей команды. Существует несколько типов конфликтов по данным, но конвейерной организацией обработки команд обусловлен только конфликт типа RAW (Read After Write): команда пытается прочитать операнд прежде, чем предшествующая команда запишет на это место свой результат. Уменьшение влияния конфликтов такого типа обеспечивается методом обхода (продвижения) данных. В этом случае результаты, полученные на выходах исполнительных устройств, помимо входов приемника результата передаются также на входы всех исполнительных устройств микропроцессора.

По внутренней организации конвейеры можно разделить на:

– синхронные, в которых время пребывания (вычисления подоперации) на каждой ступени одинаково, и передача данных между разными ступенями происходит одновременно

– асинхронные, в которых каждая ступень требует своего времени вычисления подоперации, и передача данных от одной ступени другой происходит по готовности.

1.2.4 Закон Амдала

Ускорение, получаемое при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом выполнения вычислений определяется величиной

$S_p(n)=T_1(n)/T_p(n)$,

то есть как отношение времени решения задач на скалярной ЭВМ к времени выполнения параллельного алгоритма (величина n применяется для

параметризации вычислительной сложности решаемой задачи и может пониматься, например, как количество входных данных задачи).

При определенных обстоятельствах ускорение может оказаться больше числа используемых процессоров – в этом случае говорят о существовании сверхлинейного ускорения. Одной из причин такого явления может быть неодинаковость условий выполнения последовательной и параллельной программ. Например, при решении задачи на одном процессоре оказывается недостаточно оперативной памяти для хранения всех обрабатываемых данных и тогда становится необходимым использование более медленной внешней памяти. Еще одной причиной сверхлинейного ускорения может быть нелинейный характер зависимости сложности решения задачи от объема обрабатываемых данных. Источником сверхлинейного ускорения может быть и различие вычислительных схем последовательного и параллельного методов.

Достижению максимального ускорения может препятствовать существование в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены. Пусть f есть доля последовательных вычислений в применяемом алгоритме обработки данных, тогда в соответствии с законом Амдала ускорение процесса вычислений при использовании p процессоров ограничивается величиной

$$S_p \le \frac{1}{f+(1-f)/p} \le S^* = \frac{1}{f},$$

Этот закон характеризует одну из самых серьезных проблем в области параллельного программирования, так как алгоритмов без определенной доли последовательных команд практически не существует. Однако часто доля последовательных действий характеризует не возможность параллельного решения задач, а последовательные свойства применяемых алгоритмов. Поэтому доля последовательных вычислений может быть существенно снижена при выборе более подходящих для распараллеливания методов.

Следует отметить также, что для ряда задач доля последовательных вычислений убывает с ростом n. В этом случае ускорение для фиксированного числа процессоров может быть увеличено путем увеличения вычислительной сложности решаемой задачи. То есть, имеет место эффект Амдала: ускорение является возрастающей функцией от параметра n.

1.2.5 Большие задачи

Задачи, решения которых предполагает использование больших вычислительных мощностей. Усложнение математических моделей, используемых в процессе решения таких задач, приводит к усложнению методов реализации численных экспериментов, для поддержки которых используются параллельные вычисления.

На примере проблем моделирования климатической системы рассмотрим как возникают очень большие задачи и что за этим следует дальше.

В современном понимании климатическая система включает в себя атмосферу, океан, сушу, криосферу и биоту. Климатом называется ансамбль состояний, который система проходит за достаточно большой промежуток времени. Климатическая модель – это математическая модель, описывающая климатическую систему. В основе климатической модели лежат уравнения динамики сплошной среды и уравнения равновесной термодинамики. Кроме этого, в модели описываются все энергозначимые физические процессы: перенос излучения в атмосфере, фазовые переходы воды, облака и конвенция, перенос малых газовых примесей и их трансформация, мелкомасштабная турбулентная диффузия тепла и диссипация кинетической энергии и многое другое. В целом модель представляет систему трехмерных нелинейных уравнений с частными производными. Решения этой системы должны воспроизводить все важные характеристики ансамбля состояний реальной климатической системы.

Даже без дальнейших уточнений понятно, что климатическая модель исключительно сложна. Работая с ней, приходится принимать во внимание ряд серьезных обстоятельств. В отличие от многих естественных наук, в климате нельзя поставить глобальный натурный целенаправленный эксперимент. Следовательно, единственный путь изучения климата – это проводить численные эксперименты с математической моделью и сравнивать модельные результаты с результатами наблюдений. Однако и здесь не все так просто. Математические модели для разных составляющих климатической системы развиты не одинаково. Исторически первой стала создаваться модель атмосферы. Она и в настоящее время является наиболее развитой. К тому же, за сотни лет наблюдений за ее состоянием накопилось довольно много эмпирических данных. Так что в атмосфере есть с чем сравнивать модельные результаты. Для других составляющих климатической системы результатов наблюдений существенно меньше. Слабее развиты и соответствующие математические модели. По существу только начинает создаваться модель биоты.

Общая модель климата пока еще далека от своего завершения. Не во всем даже ясно, как могла бы выглядеть совместная идеальная модель. Чтобы приблизить климатическую модель к реальности, приходится проводить очень много численных экспериментов и на основе анализа результатов вносить в нее коррективы. Как правило, пока эксперименты проводятся с моделями для отдельных составляющих климата или для некоторых их комбинаций. Наиболее часто рассматривается совместная модель атмосферы и океана. Недостающие данные от других составляющих берутся либо из результатов наблюдений, либо из каких-то других соображений.

Важнейшей проблемой современности является проблема изменения климата под влиянием изменения концентрации малых газовых

составляющих, таких как углекислый газ, озон и др. Как уже отмечалось, климатический прогноз может быть осуществлен только с помощью численных экспериментов над моделью. Поэтому ясно, что для того чтобы научиться предсказывать изменение климата в будущем, уже сегодня надо иметь возможность проводить большой объем вычислений. Попробуем хотя бы как-то его оценить.

Рассмотрим модель атмосферы как важнейшей составляющей климата и предположим, что мы интересуемся развитием атмосферных процессов на протяжении, например, 100 лет. При построении алгоритмов нахождения численных решений используется упоминавшийся ранее принцип дискретизации. Общее число элементов, на которые разбивается атмосфера в современных моделях, определяется сеткой с шагом в 1° по широте и долготе на всей поверхности земного шара и 40 слоями по высоте. Это дает около $2{,}6 \cdot 10^6$ элементов. Каждый элемент описывается примерно 10 компонентами. Следовательно, в любой фиксированный момент времени состояние атмосферы на земном шаре характеризуется ансамблем из $2{,}6 \cdot 10^7$ чисел. Условия обработки численных результатов требуют нахождения всех ансамблей через каждые 10 минут, то есть за период 100 лет необходимо определить около $5{,}3 \cdot 10^6$ ансамблей. Итого, только за один численный эксперимент приходится вычислять $1{,}4 \cdot 10^{14}$ значимых результатов промежуточных вычислений. Если теперь принять во внимание, что для получения и дальнейшей обработки каждого промежуточного результата нужно выполнить 10^2–10^3 арифметических операций, то это означает, что для проведения одного численного эксперимента с глобальной моделью атмосферы необходимо выполнить порядка 10^{16}–10^{17} арифметических операций с плавающей запятой.

Таким образом, вычислительная система с производительностью 10^{12} операций в секунду будет осуществлять такой эксперимент при полной своей загрузке и эффективном программировании в течение нескольких часов. Использование полной климатической модели увеличивает это время, как минимум, на порядок. Еще на порядок может увеличиться время за счет не лучшего программирования и накладных расходов при компиляции программ и тому подобного. А сколько нужно проводить подобных экспериментов! Поэтому вычислительная система с триллионной скоростью совсем не кажется излишне быстрой с точки зрения потребностей изучения климатических проблем.

Большой объем вычислений в климатической модели и важность связанных с ней выводов для различных сфер деятельности человека являются постоянными стимулами в деле совершенствования вычислительной техники. Так, на заре ее развития одним из следствий необходимости разработки эффективного вычислительного инструмента для решения задач прогноза погоды стало создание Дж. фон Нейманом самой теории построения «обыкновенной вычислительной системы». В нашумевших

в свое время проектах вычислительных систем пятого поколения и во многих современных амбициозных проектах климатические модели постоянно фигурируют как потребители очень больших скоростей счета. Наконец, существуют проекты построения вычислительных систем огромной производительности, предназначенных специально для решения климатических проблем.

Имеется и обратное влияние. Развитие вычислительной техники позволяет решать задачи все больших размеров. Корректное решение больших задач заставляет развивать новые разделы математики. Как уже говорилось, изучение предсказуемости климата требует определения решения системы дифференциальных уравнений на очень большом отрезке времени. Но это имеет смысл делать лишь в том случае, когда решение обладает определенной устойчивостью к малым возмущениям. Одна из составляющих климатической системы, а именно атмосфера, относится к классу так называемых открытых (то есть имеющих внешний приток энергии и ее диссипацию) нелинейных систем, траектории которых неустойчивы поточечно. Такие системы имеют совершенно особое свойство. При больших временах их решения оказываются вблизи некоторого многообразия относительно небольшой размерности. Открытие данного свойства послужило сильным толчком к развитию теории нелинейных диссипативных систем методами качественной теории дифференциальных уравнений. В свою очередь, это существенно усложнило вычислительные задачи, возникающие при климатическом прогнозе, что опять требует больших вычислительных мощностей.

1.2.6 Способы создания параллельных алгоритмов и программ, сложность

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. Для снижения сложности рассматриваемой темы оставим в стороне математические аспекты разработки и доказательства сходимости алгоритмов – эти вопросы в той или иной степени изучаются в ряде «классических» математических учебных курсов. Здесь же мы будем полагать, что вычислительные схемы решения задач, рассматриваемых далее в качестве примеров, уже известны. С учетом высказанных предположений последующие действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

– выполнить анализ имеющихся вычислительных схем и осуществить их разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга;

– выделить для сформированного набора подзадач информационные взаимодействия, которые должны осуществляться в ходе решения исходной поставленной задачи;

– определить необходимую (или доступную) для решения задачи вычислительную систему и выполнить распределение имеющего набора подзадач между процессорами системы.

При самом общем рассмотрении понятно, что объем вычислений для каждого используемого процессора должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (балансировку) процессоров. Кроме того, также понятно, что распределение подзадач между процессорами должно быть выполнено таким образом, чтобы количество информационных связей (коммуникационных взаимодействий) между подзадачами было минимальным.

После выполнения всех перечисленных этапов проектирования можно оценить эффективность разрабатываемых параллельных методов: для этого обычно определяются значения показателей качества порождаемых параллельных вычислений (ускорение, эффективность, масштабируемость). По результатам проведенного анализа может оказаться необходимым повторение отдельных (в предельном случае всех) этапов разработки – следует отметить, что возврат к предшествующим шагам разработки может происходить на любой стадии проектирования параллельных вычислительных схем.

Поэтому часто выполняемым дополнительным действием в приведенной выше схеме проектирования является корректировка состава сформированного множества задач после определения имеющегося количества процессоров – подзадачи могу быть укрупнены (агрегированы) при наличии малого числа процессоров или, наоборот, детализированы в противном случае. В целом, данные действия могут быть определены как масштабирование разрабатываемого алгоритма и выделены в качестве отдельного этапа проектирования параллельных вычислений.

Чтобы применить получаемый в конечном итоге параллельный метод, необходимо выполнить разработку программ для решения сформированного набора подзадач и разместить разработанные программы по процессорам в соответствии с выбранной схемой распределения подзадач. Для проведения вычислений программы запускаются на выполнение (программы на стадии выполнения обычно именуются процессами), для реализации информационных взаимодействий программы должны иметь в своем распоряжении средства обмена данными (каналы передачи сообщений).

Следует отметить, что каждый процессор обычно выделяется для решения единственной подзадачи, однако при наличии большого количества подзадач или использовании ограниченного числа процессоров это правило может не соблюдаться и, в результате, на процессорах может выполняться одновременно несколько программ (процессов). В частности,

при разработке и начальной проверке параллельной программы для выполнения всех процессов может использоваться один процессор (при расположении на одном процессоре процессы выполняются в режиме разделения времени).

Рассмотрев внимательно разработанную схему проектирования и реализации параллельных вычислений, можно отметить, что данный подход в значительной степени ориентирован на вычислительные системы с распределенной памятью, когда необходимые информационные взаимодействия реализуются при помощи передачи сообщений по каналам связи между процессорами. Тем не менее данная схема может быть применена без потери эффективности параллельных вычислений и для разработки параллельных методов для систем с общей памятью – в этом случае механизмы передачи сообщений для обеспечения информационных взаимодействий должны быть заменены операциями доступа к общим (разделяемым) переменным.

1.2.7 Эффективность использования суперкомпьютеров

Эффективность использования параллельным алгоритмом процессоров при решении задачи определяется соотношением

$E_p(n)=T_1(n)/(pT_p(n))=S_p(n)/p$

(эта величина определяет среднюю долю времени выполнения алгоритма, когда процессоры реально задействованы для решения задачи).

Попытки повышения качества параллельных вычислений по одному из показателей (ускорению или эффективности) могут привести к ухудшению ситуации по другому показателю, ибо показатели качества параллельных вычислений являются часто противоречивыми. Так, например, повышение ускорения обычно может быть обеспечено за счет увеличения числа процессоров, что приводит, как правило, к падению эффективности. И наоборот, повышение эффективности достигается во многих случаях при уменьшении числа процессоров. Как результат, разработка методов параллельных вычислений часто предполагает выбор некоторого компромиссного варианта с учетом желаемых показателей ускорения и эффективности.

При выборе надлежащего параллельного способа решения задачи может оказаться полезной оценка стоимости вычислений, определяемой как произведение времени параллельного решения задачи и числа используемых процессоров

$C_p=pT_p$.

В связи с этим можно определить понятие стоимостно-оптимального параллельного алгоритма как метода, стоимость которого пропорциональна времени выполнения наилучшего последовательного алгоритма.

1.2.8 Слабо взаимодействующие последовательные процессы

Определим изолированный последовательный процесс, как последовательность действий, выполняемых автономно (независимо от окружающей обстановки).

Если двум и более последовательным процессам необходимо взаимодействовать между собой, то они должны быть связаны, то есть иметь средства для обмена информацией. Будем считать, что кроме явных, достаточно редких моментов связи, процессы выполняются совершено независимо друг от друга. Кроме того, будем считать, что такие процессы связаны слабо, подразумевая под этим, что кроме некоторых моментов явной связи, эти процессы рассматриваются как совершенно независимые друг от друга.

Будем называть такие процессы слабо связанными последовательными процессами.

Теперь можно определить понятие параллельной программы. Под параллельной программой будем понимать всю совокупность слабо связанных последовательных процессов, запуск которых необходим для решения поставленной задачи. В однопроцессорной системе программе, как правило, соответствует один выполняемый процесс – программный модуль, запускаемый под управлением операционной системы на единственном процессоре. Параллельной программе, выполняющейся на наборе процессоров, соответствует ряд процессов, запущенных в общем случае на разных процессорах.

Рассмотрим пример, иллюстрирующий проблему разделения ресурсов.

В разных концах комнаты расположены две лампочки, самопроизвольно зажигающиеся случайным образом. Вспыхнувшая лампочка продолжает гореть до тех пор, пока ее не погасят. Двум наблюдателям поручено выключать зажигающиеся лампочки и учитывать общее число вспышек, записывая его мелом на доске посередине комнаты.

Рассмотрим последовательность действий:

– вспыхнула левая лампочка;

– первый наблюдатель прочел число, написанное на доске (365) и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;

– погасив лампочку и вернувшись, первый исправляет число на доске, записывая получившуюся у него сумму (366).

Результат на доске правильный, если второй наблюдатель на протяжении всего этого процесса к доске не подходил. В противном случае могло получится следующее:

– вспыхнула левая лампочка;

– первый наблюдатель прочел число, написанное на доске (365) и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;

– вспыхнула правая лампочка;

– второй наблюдатель тоже прочел число, написанное на доске (365) и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;

– погасив лампочку и вернувшись, первый исправляет число на доске, записывая получившуюся у него сумму (366);

– погасив лампочку и вернувшись, второй, в свою очередь, исправляет число на доске, записывая получившуюся у него сумму (366).

Результат не совпадает с ожидаемым. Произошло две вспышки, а общее число вспышек, записанное на доске, возросло только на 1! Ошибка наблюдателей очевидно состоит в том, что между чтением первым наблюдателям числа с доски и записью результата, второй наблюдатель успел прочитать еще не исправленное число.

Вероятно, в случае выполнения всех трех действий – чтения числа, прибавления к нему единицы и записи исправленного числа, в виде одной неделимой операции, ошибок возникать не будет.

Вопросы для самоконтроля

1. К какому классу согласно систематике Флинна относятся последовательные вычислительные системы?
2. Какой класс в систематике Флинна по сути является пустым?
3. Какой класс систематики Флинна подлежит дальнейшему разбиению в систематике Хокни?
4. Что называется параллелизмом в пространстве?
5. Что называется параллелизмом во времени?
6. Каковы преимущества конвейерной архитектуры над увеличением числа независимых вычислительных устройств?
7. В чем может быть причина снижения фактической производительности конвейера по сравнению с теоретически предсказанной?
8. Что такое структурные конфликты?
9. Что такое конфликты по управлению?
10. Как можно снизить число конфликтов по управлению?
11. Что такое конфликты по данным?
12. Чем синхронные конвейеры отличаются от асинхронных?
13. Что такое ускорение?
14. Когда ускорение может превышать число процессоров?
15. Чем ограничено ускорение согласно закону Амдала?
16. Для каких задач возникает необходимость применения высокопроизводительных вычислительных систем?
17. Какие критерии должны выполняться при выполнении декомпозиции задачи и распределения полученных подзадач по процессорам?
18. Как определяется эффективность использования параллельным алгоритмом процессоров?

19. Какие алгоритмы называются стоимостно-оптимальными?

20. Какие проблемы могут возникать при взаимодействии слабосвязанных последовательных процессов?

1.3 Как распараллеливать программу

1.3.1 Что такое распараллеливание?

Распараллеливание программ – процесс адаптации алгоритмов, записанных в виде программ, для их эффективного исполнения на вычислительной системе параллельной архитектуры (в последнее время, как правило, на многопроцессорной вычислительной системе). Заключается либо в переписывании программ на специальный язык, описывающий параллелизм и понятный трансляторам целевой вычислительной системы, либо к вставке специальной разметки (например, инструкций MPI или OpenMP).

Распараллеливание может быть ручным, автоматизированным и полу-автоматизированным.

Целью любого параллелизма является повышение эффективности работы компьютера на различных уровнях, а именно на уровне: задач, данных, алгоритмов, инструкций, битов.

Параллелизм на уровне задач относится к парадигме параллельного программирования и предполагает разбиение задачи на подзадачи. Подзадачи реализуются одновременно.

Параллелизм на уровне данных реализуется компилятором и заключается в замене множества выполнений однотипных операций одной операцией над множеством данных. Обработка осуществляется на векторных процессорах. В программах, обрабатываемых компилятором, задаются директивы компиляции, что предполагает использования языков параллельных вычислений.

Параллелизм на уровне алгоритмов предполагает замену последовательных алгоритмов некоторых вычислений на параллельные. Это касается алгоритмов поиска, сортировки и тому подобных. Организация процесса распараллеливания осуществляется за счет использования различных средств параллельного программирования, таких как, специальные библиотеки, переменные окружения, директивы компилятора и тому подобных. Примером может служить технология OpenMP.

Параллелизмом на уровне команд предполагает совмещение выполнения команд. Этот процесс не меняет результат программы. Рассмотрим проявление рассматриваемого параллелизма на уровне команд на аппаратном уровне и на программном уровне.

На аппаратном уровне параллелизм реализуется в следующих формах:

– Внеочередное исполнение машинных инструкций – технология, реализующая вычисления по мере наличия готовых к обработке данных в регистрах процессора. Тем самым исключаются простои процессора,

вызванные ожиданием данных инструкций, выполняемых в порядке следования в машинном коде.

– Суперскалярный процессор – процессор, вычислительное ядро которого состоит из нескольких АЛУ, модулей операций с плавающей точкой, умножителей других подобных устройств. Вычислительное ядро осуществляет динамическое управление потоком инструкций.

Программный параллелизм на уровне команд осуществляется компилятором. Компилятор формирует исполняемый код непосредственно под конкретный процессор.

1.3.2 Параллельный цикл

Одна из основных задач, возникающих при параллельных вычислениях, связана с распараллеливанием циклов. При распараллеливании все итерации цикла могут выполняться одновременно, могут выполняться в произвольном порядке. Понятно, что не всякий цикл может быть распараллелен, – итерации должны быть независимыми, множества переменных, изменяемых на каждой итерации, не должны пересекаться, дабы избежать проблем с гонкой данных, блокировками, синхронизацией.

Предположим, что цикл допускает распараллеливание. Насколько просто его организовать? Возникают ли какие-нибудь проблемы, связанные с распараллеливанием? Ответ напрашивается – возникают. Отметим несколько серьезных проблем, которые приходится решать в каждом конкретном случае:

– *Накладные расходы*. Когда каждая итерация цикла выполняется в отдельном потоке, то расходы, связанные с привлечением потока (расходы памяти и времени) являются накладными расходами. Если доля накладных расходов мала в сравнении с выигрышем по времени, которое получается за счет параллельного выполнения, то ими можно пожертвовать. Но для «коротких» циклов, где число операций, выполняемых на каждой из итераций, мало, нужно прилагать особые усилия для уменьшения накладных расходов.

– *Балансировка нагрузки*. Итерация итерации рознь. Может оказаться, что некоторые итерации выполняются дольше, чем остальные. В этом случае необходимо предпринимать специальные меры для балансировки нагрузки на используемые потоки.

– *Управление итерациями*. При выполнении одной или нескольких итераций могут возникать условия, требующие завершения цикла. Причины досрочного завершения могут быть разными – достижение требуемого результата, обнаружение ситуации, при которой продолжение цикла должно быть прервано, возникновение исключительной ситуации,

прерывающей выполнение итерации. Во всех этих случаях нужно разумным способом завершить уже выполняемые итерации и не порождать выполнение итераций, еще не начавших свое выполнение.

Блочное распределение итераций предполагает, что распределение итераций цикла по процессорам ведется блоками по несколько последовательных итераций. В простейшем случае количество итераций цикла N делится на число процессоров p, результат округляется до ближайшего целого сверху и число [N / p] определяет количество итераций в блоке. При этом почти все процессоры получают одинаковое количество итераций, однако один или несколько последних процессоров могут простаивать. Выбор меньшего размера блока может уменьшить этот дисбаланс, однако при этом часть итераций останется нераспределенной. Если нераспределенные итерации снова начать распределять такими же блоками, начиная с первого процессора, то получим *блочно-циклическое распределение* итераций. Если уменьшать количество итераций дальше, то дойдем до распределения по одной итерации, которое называется *циклическим распределением*. Циклическое распределение позволяет минимизировать дисбаланс в загрузке процессоров, возникавший при блочных распределениях.

Основная работа, которую обычно производит ЭВМ, это обработка циклов и других повторяющихся участков программ. Потому теория распараллеливания циклических участков программ представляет собой одно из важнейших направлений параллельного программирования.

О распараллеливании циклов принято говорить в терминах пространства итераций. Пространством итераций гнезда (вложенных) циклов называется множество целочисленных векторов, состоящих из координат, каждая из которых соответсвует одной из переменных циклов.

В терминах пространства итераций задача распараллеливания гнезда циклов ставится как задача разбиения пространства итераций на подмножества такие, что вычисления тела цикла в каждом из них могут быть выполнены одновременно (с сохранением информационных связей исходного цикла, естественно).

Распараллеливание циклов производится компиляторами. Если речь идет о распараллеливании для векторно-конвейерной вычислительной системы или для векторно-параллельной вычислительной системы, то такой компилятор называется векторизующим компилятором. Аналогично, если речь идет о распараллеливание циклов для многопроцессорной вычислительной системы, то говорят о распараллеливающем компиляторе.

На этапе синтаксического анализа производится синтаксический анализ последовательной программы, проверяется выполнение некоторых ограничений на тела циклов (линейность индексных выражений, отсутствие операторов и обращений к подпрограммам и тому подобное).

Кроме того, на этапе синтаксического анализа оценивается время выполнения программы. Недостающую информацию для такой оценки компилятор может запрашивать у пользователя в диалоговом режиме.

На этапе распараллеливания циклов проверяется выполнение всех необходимых ограничений на тела циклов. Здесь же для каждого цикла из множества методов, реализованных в компиляторе, производится выбор метода распараллеливания. Для вложенных циклов анализ производится, начиная с внутренних циклов. Для циклов, которые не распараллеливаются, компилятором выдается информация о конкретных причинах невозможности распараллеливания.

На этапе оценки качества распараллеливания оценивается степень распараллеливания каждого из циклов и на этой основе производится оценка ускорения параллельной программы по сравнению с исходной последовательной программой. Если это ускорение является неудовлетворительным, то на основе информации, выданной компилятором на предыдущем этапе, пользователь может попытаться преобразовать циклы так, чтобы они могли быть эффективно распараллелены.

На этапе генерации параллельной программы генерируется программа на параллельном языке высокого уровня.

На этапе генерации машинного кода компилятор генерирует код для используемой параллельной вычислительной системы.

В зависимости от типа областей пространства итераций выделяют несколько методов распараллеливания циклов:

– метод параллелепипедов;

– метод гиперплоскостей;

– метод пирамид.

Методы распараллеливания применимы только при выполнении ряда ограничений на операторы тела цикла. Эти ограничения зависят от типа ЭВМ и метода распараллеливания. Перечислим основные из этих ограничений:

– тело цикла не содержит условных и безусловных переходов вне тела;

– внутри тела цикла передача управления осуществляется только вперед;

– тело цикла не содержит операторов ввода-вывода и обращений к подпрограммам;

– все индексные выражения линейны, т.е. отсутствуют индексы вида;

– отсутствует косвенная адресация вида;

– индексы не изменяются в теле цикла;

– выполнено условие Рассела – в теле цикла не используется простая неиндексированная переменная раньше, чем этой переменной в теле цикла присваивается некоторое значение.

1.3.3 Распараллеливание и посылка сообщений

Особенности распараллеливания циклов на системах с распределенной памятью:

– Обмен данными между процессорами через коммуникационную систему требует значительного времени (латентность коммуникационной системы велика по сравнению со временем выполнения одной машинной команды). Поэтому вычислительная работа должна распределяться между процессорами крупными порциями. Для автоматического распараллеливания на векторно-конвейерных вычислительных системах и векторно-параллельных вычислительных систем достаточно проанализировать на предмет возможности параллельного выполнения только внутренние циклы программы. В случае мультикомпьютеров и вычислительных кластеров приходилось анализировать объемлющие циклы для нахождения более крупных порций работы. Однако крупные фрагменты программы могут включать в себя вызовы процедур и функций, что требует сложного межпроцедурного анализа.

– В отличие от многопроцессорных ЭВМ с общей памятью, на системах с распределенной памятью необходимо произвести не только распределение вычислений, но и распределение данных. Наличие распределенных данных порождает проблему обеспечения эффективного доступа к удаленным данным. Эта проблема требует анализа индексных выражений не только внутри одного цикла, но и между разными циклами, а также поиска сегментов данных, которые должны быть пересланы с одного процессора на другой.

– Распределение вычислений и данных по процессорам вычислительной системы должно быть произведено таким образом, чтобы обеспечить сбалансированность загрузки процессоров. Несбалансированность загрузки процессоров может привести к тому, что параллельная программа будет выполняться гораздо медленнее исходной последовательной программы.

Рассмотрим задачу поиска суммы заданных чисел. Разработка параллельного алгоритма для решения данной задачи не вызывает затруднений – необходимо разделить данные на равные блоки, передать эти блоки процессам, выполнить в процессах суммирование полученных данных, собрать значения вычисленных частных сумм на одном из процессов и сложить значения частичных сумм для получения общего результата решаемой задачи. При последующей разработке демонстрационных программ данный рассмотренный алгоритм будет несколько упрощен – процессам программы будут передаваться весь суммируемый вектор, а не отдельные блоки этого вектора.

Первая проблема при выполнении рассмотренного параллельного алгоритма суммирования состоит в необходимости передачи значений

вектора x всем процессам параллельной программы. Конечно, для решения этой проблемы можно воспользоваться функциями парных операций передачи данных.

Однако такое решение будет крайне неэффективным, поскольку повторение операций передачи приводит к суммированию затрат (латентностей) на подготовку передаваемых сообщений.

Операция передачи данных от одного процесса всем процессам программы – широковещательная рассылка данных.

Процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной операции передачи данных от всех процессов одному процессу. В этой операции над собираемыми значениями осуществляется та или иная обработка данных (для подчеркивания последнего момента данная операция еще именуется операцией редукции данных). Как и ранее, реализация операции редукции при помощи обычных парных операций передачи данных является неэффективной и достаточно трудоемкой.

Перейдём к задаче вычисления всех частных сумм последовательности значений и проведем анализ возможных способов последовательной и параллельной организации вычислений. Вычисление всех частных сумм на скалярном компьютере может быть получено при помощи обычного последовательного алгоритма суммирования при том же количестве операций.

Алгоритм может состоять в следующем: перед началом вычислений создается копия вектора суммируемых значений, далее на каждой итерации суммирования формируется вспомогательный вектор путем сдвига исходного вектора вправо на 2^{i-1} позиций (освобождающиеся при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования исходного и вспомогательного векторов.

Вопросы для самоконтроля

1. На каких уровнях можно проводить распараллеливание?
2. Что такое пространство итераций?
3. Для чего следует добиваться сбалансированности нагрузки на процессоры вычислительной системы?

1.4 Проведение экспериментов

1.4.1 Многоядерные компьютеры

Многоядерный процессор – центральный процессор, содержащий два и более вычислительных ядра на одном процессорном кристалле или в одном корпусе.

Архитектура многоядерных процессоров во многом повторяет архитектуру симметричных мультипроцессоров (SMP-машин) только в меньших масштабах и со своими особенностями.

Первые многоядерные процессоры (first generation CMP) представляли собой самые простые схемы: два процессорных ядра, размещенные на одном кристалле без разделения каких-либо ресурсов, кроме шины памяти (например, Sun UltraSPARC IV и Intel Pentium D). «Настоящим многоядерным» (second generation CMP) процессор считается, когда его вычислительные ядра совместно используют кэш третьего или второго уровня: например, Sun UltraSPARC IV+, Intel Core Duo и все современные многоядерные процессоры.

В многоядерных процессорах тактовая частота, как правило, намеренно снижена. Это позволяет уменьшить энергопотребление процессора без потери производительности: энергопотребление растёт как куб от роста частоты процессора. Удвоив количество ядер процессора и снизив вдвое их тактовую частоту, можно получить практически ту же производительность, при этом энергопотребление такого процессора снизится в 4 раза.

В некоторых процессорах тактовая частота каждого ядра может меняться в зависимости от его индивидуальной нагрузки. Ядро является полноценным микропроцессором, использующим все достижения микропроцессорной техники: конвейеры, внеочередное исполнение кода, многоуровневый кэш, поддержка векторных команд.

Суперскалярность в ядре присутствует не всегда, если, например, производитель процессора стремится максимально упростить ядро.

Каждое ядро может использовать технологию временной многопоточности или, если оно суперскалярное, технологию SMT для одновременного исполнения нескольких потоков, создавая иллюзию нескольких «логических процессоров» на основе каждого ядра. На процессорах компании Intel эта технология носит название Hyper-threading и удваивает число логических процессоров по сравнению с физическими. На процессорах Sun UltraSPARC T2 (2007 г.) такое увеличение может достигать 8 потоков на ядро.

Многоядерные процессоры можно подразделить по наличию поддержки когерентности (общей) кэш-памяти между ядрами. Бывают процессоры с такой поддержкой и без неё. Способ связи между ядрами:

– разделяемая шина;
– сеть (Mesh) на каналах точка-точка;
– сеть с коммутатором;
– общая кэш-память.

1.4.2 Суперкомпьютер СКИФ БГУ

Суперкомпьютер СКИФ К-1000-2 (СКИФ БГУ) является старшей моделью семейства СКИФ. Это – кластер из 288 двухпроцессорных вычислительных узлов на базе 64-разрядных процессоров. Каждый узел СКИФ К-1000 содержит двухпроцессорную системную плату SMP 2xOpteronTM 248, 2,2 GHz.

На СКИФе установлена операционная система GNU/Linux Fedora 8.0. Операционная система обеспечивает основу для выполнения всех остальных программ.

Доступ к суперкомпьютеру СКИФ осуществляется с любого компьютера локальной сети БГУ. Вход осуществляется по протоколу SSH.

Для запуска задачи на кластере используется система управления заданиями Torque PBS. Обычно для этого создается и ставится в очередь специальный скрипт, который содержит, например, информацию о необходимых ресурсах: число узлов кластера, необходимое количество оперативной памяти и необходимое время.

Для построения параллельных программ доступны несколько реализаций технологии MPI.

Пользователи кластера выполняют в основном следующие операции: файловые операции, компиляция и запуск приложений на ресурсах кластера, слежение за процессом выполнения задачи и просмотр результатов. Поэтому для работы на суперкомпьютере с персонального компьютера пользователю необходимо установить программы, которые обеспечат возможность обмена файлами между компьютером и сервером и удаленный доступ к командной строке сервера для запуска задач.

Пользователь с операционной системой Windows на локальной машине может подключиться к командному интерфейсу кластера, используя SSH-клиент PUTTY, который является лучшей из бесплатных программ данного типа.

Для обмена файлами между компьютером и сервером можно использовать программы с графическим интерфейсом. Например, для пользователей, у которых на локальной машине установлена система Windows, программы WinSCP и FileZilla поддерживают передачу файлов и не требуют работы с командной строкой.

А дальше пользователь с неадминистративным уровнем полномочий, работа которого с кластером заключается в выполнении на нем своих вычислительных задач, осуществляет традиционные операции через интерфейс командной строки: компиляция и запуск приложений, просмотр результатов, слежение за процессом выполнения.

1.4.3 Исследование распараллеливания

Рассмотренная ранее схема проектирования и реализации параллельных вычислений дает способ понимания параллельных алгоритмов и программ. На стадии проектирования параллельный метод может быть представлен в виде графа «подзадачи – сообщения», который представляет собой не что иное, как укрупненное (агрегированное) представление графа информационных зависимостей (графа «операции-операнды»). Аналогично на стадии выполнения для описания параллельной программы может быть использована модель в виде графа «процессы – каналы», в которой вместо подзадач используется понятие процессов, а информационные зависимости заменяются каналами основные шаги распараллеливания, возможные неприятности измерение производительности передачи сообщений. В дополнение, на этой модели может быть показано распределение процессов по процессорам вычислительной системы, если количество подзадач превышает число процессоров.

Выбор способа разделения вычислений на независимые части основывается на анализе вычислительной схемы решения исходной задачи. Требования, которым должен удовлетворять выбираемый подход, обычно состоят в обеспечении равного объема вычислений в выделяемых подзадачах и минимума информационных зависимостей между этими подзадачами (при прочих равных условиях нужно отдавать предпочтение редким операциям передачи большего размера сообщений по сравнению с частыми пересылками данных небольшого объема). В общем случае, проведение анализа и выделение задач представляет собой достаточно сложную проблему – ситуацию помогает разрешить существование двух часто встречающихся типов вычислительных схем.

Для большого класса задач вычисления сводятся к выполнению однотипной обработки элемент элементов большого набора данных – к такому виду задач относятся, например, матричные вычисления, численные методы решения уравнений в частных производных и др. В этом случае говорят, что существует параллелизм по данным, и выделение подзадач сводится к разделению имеющихся данных. Так, например, для нашей учебной задачи поиска максимального значения при формировании подзадач исходная матрица A может быть разделена на отдельные строки (или последовательные группы строк) – ленточная схема разделения данных или на прямоугольные наборы элементов – блочная схема разделения данных. Для большого количества решаемых задач разделение вычислений по данным приводит к порождению одно-, двух- и трех- мерных наборов подзадач, для которых информационные связи существуют только между ближайшими соседями (такие схемы обычно именуются сетками или решетками).

Для другой части задач вычисления могут состоять в выполнении разных операций над одним и тем же набором данных – в этом случае говорят о существовании функционального параллелизма. Очень часто функциональная декомпозиция может быть использована для организации конвейерной обработки данных (так, например, при выполнении каких-либо преобразований данных вычисления могут быть сведены к функциональной последовательности ввода, обработки и сохранения данных).

Важный вопрос при выделении подзадач состоит в выборе нужного уровня декомпозиции вычислений. Формирование максимально возможного количества подзадач обеспечивает использование предельно достижимого уровня параллелизма решаемой задачи, однако затрудняет анализ параллельных вычислений. Использование при декомпозиции вычислений только достаточно «крупных» подзадач приводит к ясной схеме параллельных вычислений, однако может затруднить эффективное использование достаточно большого количества процессоров. Возможное разумное сочетание этих двух подходов может состоять в использовании в качестве конструктивных элементов декомпозиции только тех подзадач, для которых методы параллельных вычислений являются известными. Так, например, при анализе задачи матричного умножения в качестве подзадач можно использовать методы скалярного произведения векторов или алгоритмы матрично-векторного произведения. Подобный промежуточный способ декомпозиции вычислений позволит обеспечить и простоту представления вычислительных схем, и эффективность параллельных расчетов. Выбираемые подзадачи при таком подходе будем именовать далее базовыми, которые могут быть элементарными (неделимыми), если не допускают дальнейшего разделения, или составными в противном случае.

При наличии вычислительной схемы решения задачи после выделения базовых подзадач определение информационных зависимостей между подзадачами обычно не вызывает больших затруднений. При этом, однако, следует отметить, что на самом деле этапы выделения подзадач и информационных зависимостей достаточно сложно поддаются разделению. Выделение подзадач должно происходить с учетом возникающих информационных связей; после анализа объема и частоты необходимых информационных обменов между подзадачами может потребоваться повторение этапа разделения вычислений.

При проведении анализа информационных зависимостей между подзадачами следует различать:

– Локальные и глобальные схемы передачи данных – для локальных схем передачи данных в каждый момент времени выполняются только между небольшим числом подзадач (располагаемых, как правило, на соседних процессорах), для глобальных операций передачи данных в процессе коммуникации принимают участие все подзадачи.

– Структурные и произвольные способы взаимодействия – для структурных способов организация взаимодействий приводит к формированию некоторых стандартных схем коммуникации (например, в виде кольца, прямоугольной решетки и так далее), для произвольных структур взаимодействия схема выполняемых операций передач данных не носит характер однородности.

– Статические или динамические схемы передачи данных – для статических схем моменты и участники информационного взаимодействия фиксируются на этапах проектирования и разработки параллельных программ, для динамического варианта взаимодействия структура операции передачи данных определяется в ходе выполняемых вычислений.

– Синхронные и асинхронные способы взаимодействия – для синхронных способов операции передачи данных выполняются только при готовности всех участников взаимодействия и завершаются только после полного окончания всех коммуникационных действий, при асинхронном выполнении операций участники взаимодействия могут не дожидаться полного завершения действий по передаче данных. Для представленных способов взаимодействия достаточно сложно выделить предпочтительные формы организации передачи данных: синхронный вариант, как правило, более прост для использования, в то время как асинхронный способ часто позволяет существенно снизить временные задержки, вызванные операциями информационного взаимодействия.

Масштабирование разработанной вычислительной схемы параллельных вычислений проводится в случае, если количество имеющихся подзадач отличается от числа планируемых к использованию процессоров. Для сокращения количества подзадач необходимо выполнить укрупнение (агрегацию) вычислений. Применяемые здесь правила совпадают с рекомендациями начального этапа выделения подзадач – определяемые подзадачи, как и ранее, должны иметь одинаковую вычислительную сложность, а объем и интенсивность информационных взаимодействий между подзадачами должны оставаться на минимально-возможном уровне. Как результат, первыми претендентами на объединение являются подзадачи с высокой степенью информационной взаимозависимости.

При недостаточном количестве имеющегося набора подзадач для загрузки всех доступных к использованию процессоров необходимо выполнить детализацию (декомпозицию) вычислений. Как правило, проведение подобной декомпозиции не вызывает каких-либо затруднений, если для базовых задач методы параллельных вычислений являются известными.

Выполнение этапа масштабирования вычислений должно свестись, в конечном итоге, к разработке правил агрегации и декомпозиции подзадач, которые должны параметрически зависеть от числа процессоров, применяемых для вычислений.

Распределение подзадач между процессорами является завершающим этапом разработки параллельного метода. Надо отметить, что управление распределением нагрузки для процессоров возможно только для вычислительных систем с распределенной памятью, для мультипроцессоров (систем с общей памятью) распределение нагрузки обычно выполняется операционной системой автоматически. Кроме того, данный этап распределения подзадач между процессорами является избыточным, если количество подзадач совпадает с числом имеющихся процессоров, а топология сети передачи данных вычислительной системы представляет собой полный граф (то есть, все процессоры связаны между собой прямыми линиями связи).

Основной показатель успешности выполнения данного этапа – эффективность использования процессоров, определяемая как относительная доля времени, в течение которого процессоры использовались для вычислений, связанных с решением исходной задачи. Пути достижения хороших результатов в этом направлении остаются прежними – как и ранее, необходимо обеспечить равномерное распределение вычислительной нагрузки между процессорами и минимизировать количество сообщений, передаваемых между процессорами. Точно так же, как и на предшествующих этапах проектирования, оптимальное решение проблемы распределения подзадач между процессорами основывается на анализе информационной связности графа «подзадачи – сообщения». Так, в частности, подзадачи, между которыми имеются информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных.

Следует отметить, что требование минимизации информационных обменов между процессорами может противоречить условию равномерной загрузки процессов. Так, мы можем разместить все подзадачи на одном процессоре и полностью устранить межпроцессорную передачу сообщений, однако, понятно, загрузка большинства процессоров в этом случае будет минимальной.

Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений может изменяться в ходе решения задачи. Причиной этого могут быть, например, неоднородные сетки при решении уравнений в частных производных, разреженность матриц и тому подобное. Кроме того, используемые на этапах проектирования оценки вычислительной сложности решения подзадач могут иметь приближенный характер и, наконец, количество подзадач может изменяться в ходе вычислений. В таких ситуациях может потребоваться перераспределение базовых подзадач между процессорами уже непосредственно в процессе выполнения параллельной программы (или, как обычно говорят, придется выполнить динамическую балансировку вычислительной нагрузки).

В качестве примера дадим краткую характеристику широко используемого способа динамического управления распределением вычислительной

нагрузки, обычно именуемого схемой «менеджер – исполнитель». При использовании данного подхода предполагается, что подзадачи могут возникать и завершаться в ходе вычислений, при этом информационные взаимодействия между подзадачами либо полностью отсутствует, либо минимальны. В соответствии с рассматриваемой схемой для управления распределением нагрузки в системе выделяется отдельный процессор-менеджер, которому доступна информация обо всех имеющихся подзадачах. Остальные процессоры системы являются исполнителями, которые для получения вычислительной нагрузки обращаются к процессору-менеджеру. Порождаемые в ходе вычислений новые подзадачи передаются обратно процессору-менеджеру и могут быть получены для решения при последующих обращениях процессоров-исполнителей. Завершение вычислений происходит в момент, когда процессоры-исполнители завершили решение всех переданных им подзадач, а процессор-менеджер не имеет каких-либо вычислительных работ для выполнения.

Вопросы для самоконтроля

1. Что такое многоядерный процессор?
2. К какому типу относится СКИФ БГУ?
3. Какие проблемы могут возникнуть при распараллеливании?

1.5 Технологии параллельного программирования

1.5.1 Средства разработки параллельных программ: коммуникационные интерфейсы

В самом общем смысле архитектуру компьютера можно определить как способ соединения компьютеров между собой, с памятью и с внешними устройствами. Реализация этого соединения может идти различными путями. Конкретная реализация такого рода соединений называется коммуникационной средой компьютера. Одна из самых простых реализаций – это использование общей шины, к которой подключаются как процессоры, так и память. Сама шина состоит из определенного числа линий связи, необходимых для передачи адресов, данных и управляющих сигналов между процессором и памятью. Этот способ реализован в SMP-системах. Основным недостатком таких систем, как было указано выше, является плохая масштабируемость. Увеличение, даже незначительное, числа устройств на шине вызывает заметные задержки при обмене с памятью и катастрофическое падение производительности системы в целом. Необходимы другие подходы для построения коммуникационной среды, и одним из них является разделение памяти на независимые модули и обеспечение возможности доступа разных процессоров к различным модулям одновременно посредством использования различного рода коммутаторов.

При этом возможны различные конфигурации получающихся систем связи. Так, в компьютерах семейства Cray T3D/T3E все процессоры были объединены специальными высокоскоростными каналами в трехмерный тор, в котором каждый вычислительный узел имел непосредственные связи с шестью соседями. В компьютерах IBM SP/2 взаимодействие процессоров происходит через иерархическую систему коммутаторов, также обеспечивающую возможность соединения каждого процессора с любым другим. Эти оригинальные уникальные решения значительно увеличивают цену компьютеров.

Гораздо более простым и дешевым оказалось использование связей на базе сетей Ethernet – методика, разработанная компанией Xerox. Первоначально использовалась обычная 10-мегабитная сеть, затем стали применять Fast Ethernet, а в последнее время иногда и Gigabit Ethernet. Но для Fast Ethernet характерна большая латентность (задержка в передаче данных), оцениваемая в 160-180 микросекунд, а Gigabit Ethernet отличается высокой стоимостью. Кроме того, он эффективен только при соединении точка–точка, при соединении нескольких узлов его эффективность резко падает, а при соединении более 5–6 узлов она не превосходит по производительности даже Fast Ethernet. Поэтому при создании многопроцессорных вычислительных систем часто предпочтение отдается технологиям SCI, Myrinet или Raceway.

SCI (Scalable Coherent Interface) принят как стандарт в 1992 г. (ANSI/IEEE Std 1596-1992). Он предназначен для достижения высоких скоростей передачи с малым временем задержки и при этом обеспечивает масштабируемую архитектуру, позволяющую строить системы, состоящие из множества блоков. SCI представляет собой комбинацию шины и локальной сети, обеспечивает реализацию когерентности кэш-памяти, размещаемой в узле SCI, посредством механизма распределенных директорий, который улучшает производительность, скрывая затраты на доступ к удаленным данным в модели с распределенной разделяемой памятью. Производительность передачи данных обычно находится в пределах от 200 Мбайт/с до 1000 Мбайт/с на расстояниях десятков метров с использованием электрических кабелей и километров – с использованием оптоволокна. SCI уменьшает время межузловых коммуникаций по сравнению с традиционными схемами передачи данных в сетях путем устранения обращений к программным уровням – операционной системе и библиотекам времени выполнения; коммуникации представляются как часть простой операции загрузки данных процессором (командами load или store). Обычно обращение к данным, физически расположенным в памяти другого вычислительного узла и не находящимся в кэше, приводит к формированию запроса к удаленному узлу для получения необходимых данных, которые в течение нескольких микросекунд доставляются в локальный кэш, и выполнение программы продолжается. Прежний подход требовал формиро-

вания пакетов на программном уровне с последующей передачей их аппаратному обеспечению. Точно так же происходил и прием, в результате чего задержки были в сотни раз больше, чем у SCI. Однако для совместимости SCI имеет возможность переносить пакеты других протоколов.

Еще одно преимущество SCI – использование простых протоколов типа RISC, которые обеспечивают большую пропускную способность. Узлы с адаптерами SCI могут использовать для соединения коммутаторы или же соединяться в кольцо. Обычно каждый узел оказывается включенным в два кольца.

1.5.2 Средства разработки параллельных программ: параллельные языки и расширения языков Fortran и C/C++

Непроцедурный язык Норма предназначен для записи численных методов решения задач математической физики разностными методами. Процесс разработки программ для решения таких задач можно разбить на следующие этапы:

– Постановка задачи. Выходом этого этапа является обычно система дифференциальных уравнений, описывающих задачу.

– Выбирается пространственно-временная сетка и производится дискретизация уравнений с помощью одного из разностных методов.

– Производится выбор метода решения дискретных уравнений. В результате получаются формулы (соотношения), описывающие необходимые вычисления в узлах сетки.

– Полученные формулы программируются на некотором языке, который обеспечивает решение задачи на вычислительной машине.

Главная идея, положенная в основу языка Норма, заключается в том, что полученное на третьем этапе описание решения задачи, почти непосредственно используется для ввода его в вычислительную систему и проведения счета.

Таким образом, язык Норма дает прикладному математику возможность сформулировать свою задачу в привычных для него терминах. Организация процесса вычислений с учетом архитектуры ЭВМ (возможностей параллельной, векторной обработки и т.п.) – это задача транслятора с языка Норма.

Запись на языке Норма – это, по существу, строгая запись численных методов решения математической задачи, запись еще не алгоритмов, а просто расчетных формул и остальной необходимой информации, которую необходимо знать, чтобы написать программу для ЭВМ.

Расчетные формулы, получаемые прикладным специалистом, обычно записываются в виде соотношений. Отметим, что в записи на Норме не требуется никакой информации о порядке счета, способах организации вычислительных (циклических) процессов. Порядок предложений языка

может быть произвольным – информационные взаимосвязи будут выявлены и учтены при организации процесса счета транслятором.

HPF – дальнейшее развитие языка Fortran 90. Включены богатые средства для распределения данных по процессорам. Необходимые коммуникации и синхронизации реализуются компилятором. Часть расширений реализована в виде функций и операторов языка, а часть – в виде директив компилятору (которые являются комментариями языка Fortran).

Fortran D95 – экспериментальный язык программирования, основанный на HPF. Расширения направлены на поддержку основных классов параллельных приложений, работающих с большими массивами данных, нерегулярными и разреженными матрицами и так далее.

В стандарте языка C++ 2011 года появились средства для разработки параллельных многопоточных приложений.

OpenMP (Open Multi-Processing) – открытый стандарт для распараллеливания программ на языках C, C++ и Fortran. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Разработка спецификаций стандарта ведётся некоммерческой организацией OpenMP Architecture Review Board (ARB), в которую входят все основные производители процессоров, а также ряд суперкомпьютерных лабораторий и университетов. Первая версия спецификации вышла в 1997 году, предназначалась только для Fortran, в следующем году вышла версия для C и C++.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой ведущий поток создаёт набор ведомых потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

Задачи, выполняемые потоками параллельно, так же, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка.

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

1.5.3 Средства разработки параллельных программ: специализированные библиотеки

ATLAS (Automatically Tuned Linear Algebra Software) – библиотека, позволяющая автоматически генерировать и оптимизировать численное программное обеспечение для процессоров с многоуровневой организацией памяти и конвейерными функциональными устройствами. Базируется

на BLAS 3 уровня. ATLAS требует некоторого времени для изучения основных параметров архитектуры целевого компьютера, а затем на основе этих параметров получает «оптимальный» код.

Aztec – параллельная библиотека итерационных методов для решения систем линейных уравнений. Позволяет описывать части распределенной матрицы с использованием глобальной адресации.

Библиотека ScaLAPACK включает подмножество процедур LAPACK, переработанных для использования на MPP-компьютерах, включая: решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и др.

Библиотека ScaLAPACK разработана с использованием PBLAS (параллельные версии BLAS уровней 1,2,3) и коммуникационной библиотеки BLACS.

Поддерживается на платформах IBM RS/6000 SP, Intel Paragon, SGI Origin 2000, и HP Exemplar, а также на кластерах рабочих станций. Может быть портирована на любую платформу, где поддерживается PVM или MPI.

1.5.4 Средства разработки параллельных программ: средства автоматического распараллеливания

BERT 77 – средство автоматического распараллеливания Fortran-программ, с генерацией параллельного кода в модели обмена сообщениями (PVM или MPI).

FORGExplorer SMP, FORGExplorer DMP – средства распараллеливания Fortran-программ для SMP и MPP-платформ, работающие на основе базы данных FORGExplorer.

KAP (Kuck Accelerator Package) – оптимизирующий препроцессор Fortran/C-программ с обнаружением параллелизма.

PIPS Workbench – среда автоматического анализа и преобразования вычислительных Fortran-программ, с возможностью генерации OpenMP-директив, или программ в модели передачи сообщений (MPI/PVM).

VAST/Parallel – набор программных продуктов для автоматического распараллеливания Fortran/C-программ для SMP-платформ. Есть возможность генерации OpenMP-кода (VAST/toOpenMP).

1.5.5 Средства разработки параллельных программ: инструментальные системы

CODE – графическая система создания параллельных программ. Параллельная программа представляется в виде графа, вершинами которого являются последовательные участки, а дуги соответствуют пересылкам данных. Последовательные участки могут быть написаны на любом языке, для пересылок используется PVM или MPI.

HeNCE (HEterogeneous Network Computing Environment) предназначено для вычислителей, желающих эффективно использовать имеющуюся сеть рабочих станций, не вдаваясь в подробности параллельного программирования.

HeNCE включает графические средства построения, компиляции, запуска и анализа HeNCE-программ (программист сам не пишет в терминах PVM). Обеспечивает высокий уровень абстракции для спецификации параллелизма. Программа описывается в виде направленного ациклического графа (DAG), в котором вершины соответствуют процедурам, а дуги – зависимостям (зависимости расставляет пользователь). Включает средства графического конфигурирования гетерогенного PVM-кластера для запуска программы.

Converse – среда (framework) для поддержки много-языкового параллельного программирования (взаимодействие модулей, написанных на разных языках). Поддерживаются парадигмы обмена сообщениями, объектно-ориентированного программирования, многопоточности.

Поддерживаемые платформы: IBM SP, Cray T3E, Origin2000, Exemplar, сети рабочих станций (на базе HP-UX 10, Solaris/SunOS, RS/6000, SGI IRIX, Linux/x86). Доступны двоичные файлы для всех поддерживаемых платформ.

1.5.6 Средства разработки параллельных программ: специализированные прикладные пакеты

ANSYS – многоцелевой конечно-элементный пакет для проведения анализа в широкой области инженерных дисциплин (прочность, теплофизика, динамика жидкостей и газов и электромагнетизм).

MSC.Nastran обеспечивает полный набор расчетов, включая расчет напряженно – деформированного состояния, собственных частот и форм колебаний, анализ устойчивости, решение задач теплопередачи, исследование установившихся и неустановившихся процессов, акустических явлений, нелинейных статических процессов, нелинейных динамических переходных процессов, расчет критических частот и вибраций роторных машин, анализ частотных характеристик при воздействии случайных нагрузок, спектральный анализ и исследование аэроупругости. Предусмотрена возможность моделирования практически всех типов материалов, включая композитные и гиперупругие. Расширенные функции включают технологию суперэлементов (подконструкций), модальный синтез и макроязык DMAP для создания пользовательских приложений.

Программный конечно-элементный комплекс ABAQUS – универсальная система общего назначения, предназначенная как для проведения многоцелевого инженерного многодисциплинарного анализа, так и для

научно-исследовательских и учебных целей в самых разных сферах деятельности, в числе которых:

– автомобилестроение (компании BMW, FORD, General Motors, Mercedes, Toyota, Volvo, GoodYear);

– авиастроение и оборонная промышленность (General Dynamics, Lockheed Martin, US Navy, Boeing);

– электроника (HP, Motorola, IBM, Digital);

– металлургия (British Steel, Dupont);

– производство электроэнергии (ABB, AEA Technology, EPRI, <Атомэнергопроект>);

– нефтедобыча и переработка (Exxon/Mobil, Shell, Dow);

– производство товаров народного потребления (3M, Kodak, Gillette);

– общая механика и геомеханика (GeoConsult, ISMES, <Гидропроект>).

LS-DYNA – многоцелевая программа разработки LSTC (Livermore Software Technology Corporation), предназначенная для решения трехмерных динамических нелинейных задач механики деформируемого твердого тела, механики жидкости и газа, теплопереноса, а также связанных задач механики деформированного твердого тела и теплопереноса, механики деформируемого твердого тела и механики жидкости и газа. В LS-DYNA реализованы эффективные методы решения перечисленных задач, в том числе явный и неявный метод конечных элементов, многокомпонентная гидродинамика (Multimaterial Eulerian Hydrodynamics), вычислительная гидродинамика несжимаемых потоков, бессеточный метод сглаженных частиц (SPH – Smoothed Particle Hydrodynamics), бессеточный метод, основанный на методе Галеркина (EFG – Element Free Galerkin method). В LS-DYNA встроены процедуры автоматической перестройки и сглаживания конечно-элементной сетки при вырождении элементов – произвольные лагранжево-эйлеровы сетки (Arbitrary Lagrangian-Euleran), высокоэффективные алгоритмы решения контактных задач, широкий набор моделей материалов, возможности пользовательского программирования, а также процедуры лагранжево-эйлерового связывания и расчета многокомпонентных течений сжимаемых сред на подвижных эйлеровых сетках.

Вопросы для самоконтроля

1. Что такое SCI?
2. Какие расширения языка C++ предназначены для написания параллельных программ?
3. Какие библиотеки специализированы для параллельного программирования?
4. Что такое KAP?
5. Что такое HeNCE?
6. Для чего предназначен LS-DYNA?

1.6 Разработка многопоточных приложений

1.6.1 Операционные системы. Поддержка разработки параллельных программ, использующих модель общей памяти

Процесс – это исполнение программы. Операционная система использует процессы для разделения исполняемых приложений. Поток – это основная единица, которой операционная система выделяет время процессора. Каждый поток имеет приоритет планирования и набор структур, в которых система сохраняет контекст потока, когда выполнение потока приостановлено. Контекст потока содержит все сведения, позволяющие потоку безболезненно возобновить выполнение, в том числе набор регистров процессора и стек потока. Несколько потоков могут выполняться в контексте процесса. Все потоки процесса используют общий диапазон виртуальных адресов. Поток может исполнять любую часть программного кода, включая части, выполняемые в данный момент другим потоком.

По умолчанию программа запускается с одним потоком, часто называемым основным потоком. Тем не менее она может создавать дополнительные потоки для выполнения кода параллельно или одновременно с основным потоком. Эти потоки часто называются рабочими потоками.

Несколько потоков используются, чтобы увеличить скорость реагирования приложения и воспользоваться преимуществами многопроцессорной или многоядерной системы, чтобы увеличить пропускную способность приложения.

Процесс – объект, владеющий памятью и другими ресурсами, но не выполняющий код. Поток – динамический объект, он может быть создан в процессе выполнения кода приложения и может быть удален по ходу выполнения. У процесса может быть несколько одновременно существующих потоков, выполняющих различные фрагменты кода. ОС планирует время процессоров между потоками, и для нее не имеет значение, какому процессу принадлежит тот или иной поток. Говоря о потоках в операционной системе, будем рассматривать общую схему, опуская многие детали, основываясь на стратегии распределения процессорного времени, характерной для ОС Windows. Эта стратегия носит название «вытесняющая приоритетная многозадачность». Многозадачность в данном контексте означает, что планировщик ОС, распределяет время процессора между многими потоками, присутствующими в ОС.

Приоритетность означает, что потоки могут иметь разные приоритеты. В этом случае из двух потоков, готовых к выполнению, на выполнение будет выбран тот, у кого больше приоритет. Более того, если в процессе выполнения потока появился готовый к выполнению поток с большим приоритетом, то выполнение текущего потока будет приостановлено, даже если не истек отведенный ему квант времени. Когда на дороге появляется президентский кортеж, то все участники дорожного движения останавливаются

и ждут, пока кортеж не проедет. Все потоки распределяются по группам приоритетности, потоки из одной группы могут быть выбраны на выполнение только в том случае, если нет готовых к выполнению потоков в группах с высшей приоритетностью.

Значит ли это, что могут быть «обиженные» приложения с низким приоритетом, до выполнения которых никогда не дойдет очередь? Это не так. ОС старается никого не обидеть. Если некоторое приложение долго не выполнялось, то ОС временно повышает его приоритет, так что и оно начнет выполняться.

Вытесняющая многозадачность характеризует стратегию планирования для потоков с одинаковым приоритетом. Все потоки в одной группе выстраиваются в очередь. Каждому из них в соответствии с очередью отводится на выполнение некоторый квант времени процессора. По истечении этого кванта поток переводится в состояние «готовность» независимо от его желания продолжить работу, и в состояние «выполнение» переводится следующий по очереди поток. Эту стратегию иногда называют «каруселью». Карусель сделала несколько оборотов, остановилась, все выходят, и места занимают следующие желающие прокатиться, ожидающие с нетерпением своей очереди.

После создания потока и должной инициализации поток переходит в состояние «готовность», занимая в своей группе приоритетности место в конце очереди. Планировщик ОС в соответствии с описанной стратегией выбирает поток, переводя его в состояние «выполнение». По истечении отведенного кванта времени поток возвращается в состояние «готовность», становясь в хвост очереди в своей группе приоритетности. Из состояния «выполнение» поток может перейти в другие состояния и до завершения отведенного кванта времени. В состояние «готовность» он может перейти, если появился поток с большим приоритетом. В состояние «завершение» поток переходит, выполнив свою работу, завершив выполнение отведенного ему фрагмента кода. В состояние «ожидание» поток может перейти, если его дальнейшее выполнение возможно только после наступления некоторого события (например, ему требуются данные, а устройство компьютера, выполняющее ввод этих данных, еще не завершило свою работу). Из состояния «ожидание» поток может перейти в состояние «готовность», если наступило событие, ожидаемое потоком. За время жизни потока он многократно проходит цикл «готовность – выполнение – ожидание – готовность», иногда минуя переход в состояние «ожидания».

Для понимания картины в целом нужно помнить, что весь процесс вычислений на компьютере управляется событиями. Каждый поток во время своего выполнения многократно прерывается, уступая свое место другому потоку. События, приводящие к приостановке выполнения потока, могут быть асинхронными по отношению к его работе, – они могут произойти в любой момент выполнения потока. Такие события называются

прерываниями. Синхронные события, связанные с тем, что по тем или иным причинам выполнение потока становится невозможным, называются исключениями или исключительными ситуациями. Типичными примерами исключительных ситуаций являются такие ситуации, как попытка деления целого числа на ноль или попытка чтения записи несуществующего файла.

Прерывания инициируются аппаратурой компьютера, чаще всего таймером и устройствами ввода-вывода. ОС в очень коротком цикле рассматривает все возникшие прерывания и должным образом их обрабатывает. Когда возникает прерывание от таймера, то ОС при его обработке из кванта времени, отводимого выполняемому потоку, вычитает время, равное интервалу таймера. Если отводимое потоку время исчерпано, поток снимается с выполнения, переходя в состояние «готовность». Когда устройство ввода заканчивает выполнение очередного задания, оно инициализирует аппаратное прерывание, свидетельствующее о завершении работы. Обрабатывая это прерывание, ОС может перевести некоторый поток из состояния «ожидания» в состояние «готовности», поскольку выполнена его заявка на ввод данных.

У исключений, связанных с самим потоком, более широкий спектр. Потоку, например, может понадобиться ввод внешних данных. Поток не может непосредственно обратиться к устройству ввода. Устройство одно, а потоков много. Поэтому поток вызывает соответствующий системный сервис. С точки зрения ядра ОС возникло исключение. При его обработке поток переводится в режим «ожидания», и начинает работать поток, содержащий соответствующий сервис, который анализирует загруженность устройства, формирует новую заявку для устройства, ставя ее в очередь.

Причина исключения может быть как аппаратной, так и программной. Деление на ноль, это, конечно же, программная ошибка. Исключения, связанные с тем, что не прочитаны требуемые внешние данные, могут быть связаны как со сбоем аппаратуры, так и с неверно заданными адресами в программе. Если письмо не доставлено, то виноватой может быть почтовая служба, а возможно вы послали письмо «на деревню дедушке».

Поток может сам инициировать исключение, уведомляя, например, ОС о том, что он «засыпает» на некоторое фиксированное время. Обрабатывая это прерывание, ОС переводит поток в состояние «ожидание». При обработке одного из очередных прерываний по таймеру, когда завершается время «сна», указанное потоком, поток переводится из состояния «ожидание» в состояние «готовность». Поток может перейти в состояние «ожидание» и по другим причинам, возникающим в ходе выполнения программного кода, например, ожидая завершения работы другого потока.

1.6.2 Встроенные потоки Windows и Unix

В Windows функция CreateThread создает новый поток для процесса. В создаваемом потоке должен быть указан начальный адрес кода, который будет выполняться новым потоком. Как правило, начальный адрес – это имя функции, определенной в программном коде. Эта функция принимает один параметр и возвращает значение типа DWORD . Процесс может иметь несколько потоков, одновременно выполняющих одну и ту же функцию.

Для завершения работы потока, выполняющаяся в нём функция должна вызвать ExitThread или выполнить возврат.

Для принудительного завершения потока используется функция TerminateThread.

Чтобы избежать состояний гонки и взаимоблокировок, необходимо синхронизировать доступ нескольких потоков к общим ресурсам. Синхронизация также необходима, чтобы обеспечить выполнение взаимозависимого кода в соответствующей последовательности.

Существует несколько объектов, дескрипторы которых можно использовать для синхронизации нескольких потоков. К этим объектам относятся:

Входные буферы консоли

– События

– Мьютексы

– Семафоры

– Критические секции

Стандарт POSIX определяет два основных типа данных для потоков: pthread_t – дескриптор потока ; pthread_attr_t – набор атрибутов потока.

Стандарт POSIX специфицирует следующий набор функций для управления потоками:

– pthread_create() – создание потока

– pthread_exit() – завершение потока (должна вызываться функцией потока при завершении)

– pthread_cancel() – отмена потока

– pthread_join() – заблокировать выполнение потока до прекращения другого потока, указанного в вызове функции

– pthread_detach() – освободить ресурсы занимаемые потоком (если поток выполняется, то освобождение ресурсов произойдёт после его завершения)

– pthread_attr_init() – инициализировать структуру атрибутов потока

– pthread_attr_setdetachstate() – указать системе, что после завершения потока она может автоматически освободить ресурсы, занимаемые потоком

– pthread_attr_destroy() – освободить память от структуры атрибутов потока (уничтожить дескриптор).

Имеются следующие примитивы синхронизации POSIX-потоков с помощью мьютексов (mutexes) – аналогов семафоров – и условных переменных (conditional variables) – оба эти типа объектов для синхронизации подробно рассмотрены позже в данном курсе:

– pthread_mutex_init() – создание мьютекса;

– pthread_mutex_destroy() – уничтожение мьютекса;

– pthread_mutex_lock() – закрытие мьютекса;

– pthread_mutex_trylock() – пробное закрытие мьютекса (если он уже закрыт, вызов игнорируется, и поток не блокируется);

– pthread_mutex_unlock() – открытие мьютекса;

– pthread_cond_init() – создание условной переменной;

– pthread_cond_signal() – разблокировка условной переменной;

– pthread_cond_wait() – ожидание по условной переменной.

Вопросы для самоконтроля

1. Что означает, что поток находится в состоянии ожидания?
2. Чем поток отличается от процесса?
3. Какие имеются средства синхронизации потоков?

1.7 Параллельное программирование на C++11

1.7.1 Управление потоками

В C++11, работа с потокам осуществляется по средствам класса std::thread (доступного из заголовочного файла <thread>), который может работать с регулярными функциями, лямбдами и функторами. Кроме того, он позволяет вам передавать любое число параметров в функцию потока.

```
#include <thread>
void threadFunction()
{
	// do smth
}
int main()
{
	std::thread thr(threadFunction);
	thr.join();
	return 0;
}
```

В этом примере, thr – это объект, представляющий поток, в котором будет выполняться функция threadFunction. Вызов join блокирует вызывающий поток (в нашем случае – поток main) до тех пор, пока thr (а точнее threadFunction) не выполнит свою работу.

Помимо метода join, следует рассмотреть еще один, похожий метод – detach. Он позволяет отсоединить поток от объекта, иными словами, сделать его фоновым. К отсоединенным потокам больше нельзя применять join.

В пространстве имен std::this_thread определено несколько полезных функций:

– get_id – возвращает id текущего потока;

– yield – говорит планировщику выполнять другие потоки, может использоваться при активном ожидании;

– sleep_for – блокирует выполнение текущего потока в течение установленного периода;

– sleep_until – блокирует выполнение текущего потока, пока не будет достигнут указанный момент времени.

1.7.2 Разделение данных между потоками

Если функция потока возвращает значение – оно будет проигнорировано. Однако принять функция может любое количество параметров.

```
void threadFunction(int i, double d, const std::string &s)
{
     std::cout << i << ", " << d << ", " << s << std::endl;
}
 int main()
{
     std::thread thr(threadFunction, 1, 2.34, "example");
     thr.join();
     return 0;
}
```

Несмотря на то, что передавать можно любое число параметров, все они были переданы по значению Если в функцию необходимо передать параметры по ссылке, они должны быть обернуты в std::ref или std::cref, как в примере:

```
void threadFunction(int &a)
{
     a++;
}

int main()
{
     int a = 1;
     std::thread thr(threadFunction, std::ref(a));
     thr.join();
     std::cout << a << std::endl;
     return 0;
}
```

Программа напечатает в консоль 2. Если не использовать std::ref, то результатом работы программы будет 1.

Стоит понимать, что на доступность переменных из функций влияют те же правила, что и без многопоточного программирования. Однако доступ к общим переменным следует синхронизировать для предотвращения состояния гонки.

1.7.3 Синхронизация параллельных операций

Мьютекс – базовый элемент синхронизации и в C++11 представлен в 4 формах в заголовочном файле <mutex>:

– mutex – обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock();

– recursive_mutex – может быть повторно захвачен своим владельцем (освобожден должен быть столько же раз сколько и захвачен);

– timed_mutex – в отличие от обычного мьютекса, имеет еще два метода: try_lock_for и try_lock_until, которые позволяют ограничивать время ожидания блокировки;

– recursive_timed_mutex – это комбинация timed_mutex и recursive_mutex.

Перед обращением к общим данным, мьютекс должен быть заблокирован методом lock, а после окончания работы с общими данными – разблокирован методом unlock.

Явная блокировка и разблокировка могут привести к ошибкам, например, если вы забудете разблокировать поток или, наоборот, будет неправильный порядок блокировок – все это приведет к ситуации тупика (хотя бы один поток не может покинуть состояние ожидания). В пространстве имен std есть несколько классов и функций для решения этой проблемы.

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

– lock_guard – когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())

– unique_lock – в отличие от lock_guard, также поддерживает отложенную блокировку, временную блокировку, рекурсивную блокировку и использование условных переменных.

Помимо описанных ранее способов синхронизации, C++11 предоставляет поддержку условных переменных, которые позволяют блокировать один или более потоков, пока либо не будет получено уведомление от другого потока, либо не произойдет «ложное/случайное пробуждение».

Есть две реализации условных переменных, доступных в заголовке <condition_variable>:

– condition_variable – требует от любого потока перед ожиданием сначала выполнить std::unique_lock;

– condition_variable_any – более общая реализация, которая работает с любым типом, который можно заблокировать. Эта реализация может быть более дорогим (с точки зрения ресурсов и производительности) для использования, поэтому ее следует использовать только если необходима те дополнительные возможности, которые она обеспечивает.

Условные переменные работают следующим образом:

– Должен быть хотя бы один поток, ожидающий, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить unique_lock. Эта блокировка передается методу wait(), который освобождает мьютекс и приостанавливает поток, пока не будет получен сигнал от условной переменной. Когда это произойдет, поток пробудится и снова выполнится lock.

– Должен быть хотя бы один поток, сигнализирующий о том, что условие стало истинным. Сигнал может быть послан с помощью notify_one(), при этом будет разблокирован один (любой) поток из ожидающих, или notify_all(), что разблокирует все ожидающие потоки.

– В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемых в многопроцессорных системах, могут происходить ложные пробуждения. Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден. Так как ложные пробуждения могут происходить многократно, такую проверку необходимо организовывать в цикле.

В стандартной библиотеке C++ одноразовые события моделируются с помощью будущего результата. Если поток должен ждать некоего одноразового события, то он каким-то образом получает представляющий его объект-будущее. Затем поток может периодически в течение очень короткого времени ожидать этот объект-будущее, проверяя, произошло ли событие, а между проверками заниматься другим делом. Можно поступить и иначе – выполнять другую работу до тех пор, пока не наступит момент, когда без наступления ожидаемого события двигаться дальше невозможно, и вот тогда ждать готовности будущего результата. С будущим результатом могут быть ассоциированы какие-то данные (например, номер выхода в объявлении на посадку), но это необязательно. После того как событие произошло (то есть будущий результат готов), сбросить объект-будущее в исходное состояние уже невозможно.

В стандартной библиотеке C++ есть две разновидности будущих результатов, реализованные в форме двух шаблонов классов, которые объявлены в заголовке <future>: уникальные будущие результаты (std::future<>) и разделяемые будущие результаты (std::shared_future<>). Эти классы

устроены по образцу std::unique_ptr и std::shared_ptr. На одно событие может ссылаться только один экземпляр std::future, но несколько экземпляров std::shared_future. В последнем случае все экземпляры оказываются готовы одновременно и могут обращаться к ассоциированным с событием данным. Именно из-за ассоциированных данных будущие результаты представлены шаблонами, а не обычными классами; точно так же шаблоны std::unique_ptr и std::shared_ptr параметризованы типом ассоциированных данных. Если ассоциированных данных нет, то следует использовать специализации шаблонов std::future<void> и std::shared_future<void> . Хотя будущие результаты используются как механизм межпоточной коммуникации, сами по себе они не обеспечивают синхронизацию доступа. Если несколько потоков обращаются к единственному объекту-будущему, то они должны защитить доступ с помощью мьютекса или какого-либо другого механизма синхронизации. Однако, каждый из нескольких потоков может работать с собственной копией std::shared_future<> безо всякой синхронизации, даже если все они ссылаются на один и тот же асинхронно получаемый результат.

1.7.4 Проектирование параллельных структур данных

При проектировании структур данных для параллельного доступа нужно учитывать два аспекта: обеспечить безопасность доступа и разрешить истинно параллельный доступ:

– Гарантировать, что ни один поток не может увидеть состояние, в котором инварианты структуры данных нарушены действиями со стороны других потоков.

– Позаботиться о предотвращении состояний гонки, внутренне присущих структуре данных, предоставив такие функции, которые выполняли бы операции целиком, а не частями.

– Обращать внимание на том, как ведет себя структура данных при наличии исключений, – не допускать нарушения инвариантов и в этом случае.

– Минимизировать шансы возникновения взаимоблокировки, ограничивая область действия блокировок и избегая по возможности вложенных блокировок.

Прежде чем задумываться об этих деталях, важно решить, какие ограничения вы собираетесь наложить на использование структуры данных: если некоторый поток обращается к структуре с помощью некоторой функции, то какие функции можно в этот момент безопасно вызывать из других потоков?

Это на самом деле весьма важный вопрос. Обычно конструкторы и деструкторы нуждаются в монопольном доступе к структуре данных, но обязанность не обращаться к структуре до завершения конструирования

или после начала уничтожения возлагается на пользователя. Если структура поддерживает присваивание, функцию swap() или копирующий конструктор, то проектировщик должен решить, безопасно ли вызывать эти операции одновременно с другими или пользователь должен обеспечить на время их выполнения монопольный доступ, хотя большинство других операций можно без опаски выполнять параллельно из разных потоков.

Второй аспект, нуждающийся в рассмотрении, – обеспечение истинно параллельного доступа. Рассмотрение данного аспекта подразумевает поиск ответов на ряд вопросов:

– Можно ли ограничить область действия блокировок, так чтобы некоторые части операции выполнялись не под защитой блокировки?

– Можно ли защитить разные части структуры данных разными мьютексами?

– Все ли операции нуждаются в одинаковом уровне защиты?

– Можно ли с помощью простого изменения структуры данных расширить возможности распараллеливания, не затрагивая семантику операций?

Проектирование параллельных структур данных с блокировками сводится к тому, чтобы захватить нужный мьютекс при доступе к данным и удерживать его минимально возможное время. Это довольно сложно, даже когда имеется только один мьютекс, защищающий всю структуру.

Требуется гарантировать, что к данным невозможно обратиться без защиты со стороны мьютекса и что интерфейс свободен от внутренне присущих состояний гонки. Если для защиты отдельных частей структуры применяются разные мьютексы, то проблема еще усложняется, поскольку в случае, когда некоторые операции требуют захвата нескольких мьютексов, появляется возможность взаимоблокировки. Поэтому к проектированию структуры данных с несколькими мьютексами следует подходить еще более внимательно, чем при наличии единственного мьютекса.

Некорректное использование мьютексов может стать причиной взаимоблокировок, а гранулярность блокировок может влиять на истинно параллельное выполнение программы. Если бы удалось разработать структуры данных, с которыми можно было бы безопасно работать, не прибегая к блокировкам, то эти проблемы вообще не возникали бы. Такие структуры называются структурами данных без блокировок, или свободными от блокировок.

Алгоритмы и структуры данных, в которых для синхронизации доступа используются мьютексы, условные переменные и будущие результаты, называются блокирующими. Приложение вызывает библиотечные функции, которые приостанавливают выполнение потока до тех пор, пока другой поток не завершит некоторое действие. Такие библиотечные функции называются блокирующими, потому что поток не может продвинуться дальше некоторой точки, пока не будет снят блокировка. Обычно ОС полностью приостанавливает заблокированный поток (и отдает его временные кванты другому потоку) до тех пор, пока он не будет разблокирован

в результате выполнения некоторого действия в другом потоке, будь то освобождение мьютекса, сигнал условной переменной или перевод будущего результата в состояние готов.

Структуры данных и алгоритмы, в которые блокирующие библиотечные функции не используются, называются неблокирующими. Но не все такие структуры данных свободны от блокировок.

Чтобы структура данных считалась свободной от блокировок, она должна быть открыта для одновременного доступа со стороны сразу нескольких потоков. Не требуется, чтобы потоки могли выполнять одну и ту же операцию; свободная от блокировок очередь может позволять одного потоку помещать, а другому – извлекать данные, но запрещать одновременное добавление данных со стороны двух потоков. Более того, если один из потоков, обращающихся к структуре данных, будет приостановлен планировщиком в середине операции, то остальные должны иметь возможность завершить операцию, не дожидаясь возобновления приостановленного потока.

Алгоритмы, в которых применяются операции сравнения с обменом, часто содержат циклы. Зачем вообще используются такие операции? Затем, что другой поток может в промежутке модифицировать данные, и тогда программа должна будет повторить часть операции, прежде чем попытается еще раз выполнить сравнение с обменом. Такой код может оставаться свободным от блокировок при условии, что сравнение с обменом в конце концов завершится успешно, если другие потоки будут приостановлены. Если это не так, то мы по существу получаем алгоритм неблокирующий, но не свободный от блокировок.

Свободные от блокировок алгоритмы с такими циклами могут приводить к застреванию (starvation) потоков, когда один поток, выполняющий операции с «неудачным» хронометражем, продвигается вперед, а другой вынужден постоянно повторять одну и ту же операцию. Структуры данных, в которых такой проблемы не возникает, называются свободными от блокировок и ожидания.

Свободная от блокировок структура данных называется свободной от ожидания, если обладает дополнительным свойством: каждый обращающийся к ней поток может завершить свою работу за ограниченное количество шагов вне зависимости от поведения остальных потоков. Алгоритмы, в которых количество шагов может быть неограниченно из-за конфликтов с другими потоками, не свободны от ожидания.

Корректно реализовать свободные от ожидания структуры данных чрезвычайно трудно. Чтобы гарантировать, что каждый поток сможет завершить свою работу за ограниченное количество шагов, необходимо убедиться, что каждая операция может быть выполнена за один проход и что шаги, выполняемые одним потоком, не приводят к ошибке в операциях, выполняемых другими потоками. В результате алгоритмы выполнения

различных операций могут значительно усложниться. Учитывая, насколько трудно правильно реализовать структуру данных, свободную от блокировок и ожидания, нужно иметь весьма веские причины для того, чтобы взяться за это дело; требуется тщательно соотносить затраты с ожидаемым выигрышем.

Вопросы для самоконтроля

1. Как в C++11 создать поток, выполняющий указанную функцию?
2. Как в C++11 дождаться завершения потока?
3. Какие средства синхронизации потоков существуют в C++11?
4. Какие структуры данных называются свободными от блокировок?
5. Какие структуры данных называются свободными от ожидания?

1.8 Программный интерфейс OpenMP

1.8.1 Назначение и компоненты OpenMP

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (symmetric multiprocessors, SMP).

Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время многоядерные процессоры, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство. Для общности излагаемого учебного материала для упоминания одновременно и мультипроцессоров и многоядерных процессоров для обозначения одного вычислительного устройства (одноядерного процессора или одного процессорного ядра) будет использоваться понятие вычислительного элемента (ВЭ).

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное

программирование, так и уже имеющиеся языки программирования, расширенные некоторым набором операторов для параллельных вычислений.

В последнее время активно развивается подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный текст программы остается неизменным, и по нему, в случае отсутствия препроцессора, компилятор построит исходный последовательный программный код. Препроцессор же, будучи примененным, заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки).

Рассмотренный выше подход является основой технологии OpenMP, наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной технологии директивы параллелизма используются для выделения в программе параллельных фрагментов, в которых последовательный исполняемый код может быть разделен на несколько раздельных командных потоков. Далее эти потоки могут исполняться на разных процессорах (процессорных ядрах) вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (однопотоковых) и параллельных (многопотоковых) участков программного кода. Подобный принцип организации параллелизма получил наименование «вилочного» (fork-join) или пульсирующего параллелизма.

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах в модели общей памяти (shared memory model). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

1.8.2 Особенности модели памяти OpenMP

Пусть процесс в критической секции циклически читает и записывает значение некоторых переменных. Даже, если остальные процессоры и не пытаются обращаться к этим переменным до выхода первого процесса из критической секции, для удовлетворения требований рассматриваемых ранее моделей консистентности они должны видеть все записи первого процессора в порядке их выполнения, что, естественно, совершенно не нужно.

Наилучшее решение в такой ситуации – это позволить первому процессу завершить выполнение критической секции и, только после этого, переслать остальным процессам значения модифицированных переменных, не заботясь о пересылке промежуточных результатов.

Модель слабой консистентности, основана на выделении среди переменных специальных синхронизационных переменных и описывается следующими правилами:

1. Доступ к синхронизационным переменным определяется моделью последовательной консистентности.

2. Доступ к синхронизационным переменным запрещен (задерживается), пока не выполнены все предыдущие операции записи.

3. Доступ к данным (запись, чтение) запрещен, пока не выполнены все предыдущие обращения к синхронизационным переменным.

Первое правило определяет, что все процессы видят обращения к синхронизационным переменным в определенном (одном и том же) порядке.

Второе правило гарантирует, что выполнение процессором операции обращения к синхронизационной переменной возможно только после выталкивания конвейера (полного завершения выполнения на всех процессорах всех предыдущих операций записи переменных, выданных данным процессором).

Третье правило определяет, что при обращении к обычным (не синхронизационным) переменным на чтение или запись, все предыдущие обращения к синхронизационным переменным должны быть выполнены полностью. Выполнив синхронизацию перед обращением к общей переменной, процесс может быть уверен, что получит правильное значение этой переменной.

В системе со слабой консистентностью возникает проблема при обращении к синхронизационной переменной: система не имеет информации о цели этого обращения – или процесс завершил модификацию общей переменной, или готовится прочитать значение общей переменной.

Для более эффективной реализации модели консистентности система должна различать две ситуации: вход в критическую секцию и выход из нее.

В модели консистентности по выходу введены специальные функции обращения к синхронизационным переменным:

ACQUIRE – захват синхронизационной переменной, информирует систему о входе в критическую секцию;

RELEASE – освобождение синхронизационной переменной, определяет завершение критической секции.

Следующие правила определяют требования к модели консистентности по выходу:

– До выполнения обращения к общей переменной, должны быть полностью выполнены все предыдущие захваты синхронизационных переменных данным процессором.

– Перед освобождением синхронизационной переменной должны быть закончены все операции чтения/записи, выполнявшиеся процессором прежде.

– Реализация операций захвата и освобождения синхронизационной переменной должны удовлетворять требованиям процессорной консистентности (последовательная консистентность не требуется).

OpenMP предоставляет модель общей памяти со слабой консистентностью. Все потоки OpenMP имеют доступ к месту для хранения и извлечения переменных, называемому памятью. Кроме того, каждому потоку разрешено иметь собственное временное представление памяти. Временное представление памяти для каждого потока не является обязательной частью модели памяти OpenMP, но может представлять любую промежуточную структуру, такую как машинные регистры, кеш или другое локальное хранилище, между потоком и памятью. Временное представление памяти позволяет потоку кэшировать переменные и тем самым избегать обращения к памяти при каждой ссылке на переменную. Каждый поток также имеет доступ к другому типу памяти, который не должен быть доступен другим потокам, называемым личной памятью потока.

Директива parallel определяет два типа доступа к переменным, используемым в связанном структурированном блоке: общий (shared) и частный (private). Каждая переменная, на которую имеется ссылка в структурированном блоке, имеет исходную переменную, которая представляет собой переменную с тем же именем, которая существует в программе непосредственно за пределами параллельной конструкции. Каждая ссылка на общую переменную в структурированном блоке становится ссылкой на исходную переменную. Для каждой частной переменной, на которую имеется ссылка в структурированном блоке, новая версия исходной переменной (того же типа и размера) создается в памяти для каждого потока группы, сформированной для выполнения параллельной области, связанной с параллельной директивой, за исключением, возможно, главного потока команд. Ссылки на частную переменную в структурированном блоке относятся к частной версии исходной переменной текущего потока. Если несколько потоков записывают в одну и ту же общую переменную без синхронизации, результирующее значение переменной в памяти не указывается. Если хотя бы один поток читает из общей переменной и хотя бы один поток записывает в нее без синхронизации, значение, видимое любым потоком чтения, не определено.

Если пересечение множеств переменных, указанных в операциях flush, выполняемых различными нитями не пустое, то результат выполнения операций flush будет таким, как если бы эти операции выполнялись в некоторой последовательности (единой для всех нитей).

Если пересечение множеств переменных, указанных в операциях flush, выполняемых одной нитью не пустое, то результат выполнения операций flush, будет таким, как если бы эти операции выполнялись в порядке определяемом программой.

Если пересечение множеств переменных, указанных в операциях flush, пустое, то операции flush могут выполняться независимо (в любом порядке).

1.8.3 Конструкции для создания потоков

Основные конструкции OpenMP – это директивы компилятора (директивы препроцессора) языка C/C++. Ниже приведен общий вид директивы OpenMP.

```
#pragma omp конструкция [предложение [предложение] … ]
```

Параллельные регионы являются основным понятием в OpenMP. Именно там, где задан этот регион программа исполняется параллельно. Как только компилятор встречает директиву omp parallel, он вставляет инструкции для создания параллельных потоков.

Каждый порожденный поток исполняет блок код в структурном блоке. По умолчанию синхронизация между потоками отсутствует и поэтому последовательность выполнения конкретного оператора различными потоками не определена.

После выполнения параллельного участка кода все потоки, кроме основного завершаются, и только основной поток продолжает исполняться, но уже один.

Каждый поток имеет свой уникальный номер, который изменяется от 0 (для основного потока) до количества потоков – 1. Идентификатор потока может быть определен с помощью функции omp_get_thread_num().

Зная идентификатор потока, можно внутри области параллельного исполнения направит потоки по разным ветвям.

Область параллельного исполнения описывается в C/C++ следующим образом:

```
#pragma omp parallel \
 shared(var1, var2, ....) \
 private(var1, var2, ...) \
 firstprivate(var1, var2, ...) \
 reduction(оператор:var1, var2, ...) \
 if(выражение) \
 default(shared|private|none) \
 copyin(var1, var2, ...) \
 num_threads(целочисленное выражение)
{
 структурный блок
}
```

Существует две модели исполнения: динамическая, когда количество используемых потоков в программе может устанавливаться реализацией самостоятельно в рамках указанного количества, и статическая, когда количество потоков в точности равно указанному.

Модель исполнения контролируется или через переменную окружения OPM_DYNAMIC или с помощью вызова функции omp_set_dynamic().

Задать желаемое количество потоков можно через переменную окружения OMP_NUM_THREADS, с помощью вызова функции omp_set_num_threads или добавив к директиве parallel условие num_threads.

Условие if говорит о том, что параллельное выполнение необходимо только если выражение истинно.

Условием schedule контролируется то, как итерации цикла распределяются между потоками.

1.8.4 Конструкции распределения работы между потоками

В OpenMP имеются три конструкции разделения работы:

– параллельный цикл for;

– параллельные секции (sections);

– конструкция single.

Цель конструкции параллельный цикл for – распределение итераций цикла по потокам. По умолчанию конец цикла является барьером синхронизации для потоков. Все потоки достигнув конца цикла дожидаются тех, кто еще не завершился, после чего они продолжают выполняться дальше. Используя условие nowait для цикла можно удалить неявный барьер синхронизации в конце цикла.

Директива параллельного цикла for имеет следующий синтаксис:

```
#pragma omp for \
 private(var1, var2, ...) \
 shared(var1, var2, ...) \
 firstprivate(var1, var2, ...) \
 lastprivate(var1, var2, ...) \
 reduction(оператор: var1, var2, ....) \
 ordered \
 schedule(тип [, размер блока]) \
 nowait \
 if(выражение)
for (инициализация; var сравнение b; продвижение)
 тело цикла
```

Порой возникает необходимость параллельно выполнить действия, которые не являются итерациями цикла. Конечно можно воспользоваться для этих целей простой директивой parallel, но тогда придется писать

дополнительный код, чтобы различную работу распределить между потоками. Более просто эту задачу можно решить с помощью параллельных секций.

Конструкции параллельный цикл for и секции могут быть совмещены с директивой parallel.

```
#pragma omp parallel sections
{
#pragma omp section
 {
 printf("T%d: foo\n", omp_get_thread_num());
 }
#pragma omp section
 {
 printf("T%d: bar\n", omp_get_thread_num());
 }
}// omp sections
```

Каждая секция выполняется в отдельном потоке, что позволяет производить декомпозицию по коду. Точкой синхронизации является конец блока sections. В случае, когда необходимо чтобы потоки не ждали завершения остальными потоками выполнения секций следует использовать условие nowait.

Если в параллельной секции требуется выполнить какое-либо действие и при этом это действие должно быть выполнено только одним потоком (например, подсчет промежуточного результата), то для этого идеально подходит конструкция single.

1.8.5 Конструкции для управления работой с данными, синхронизации потоков

Условие shared указывает на то, что все перечисленные переменные будут разделяться между потоками.

Условие private указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

Условие firstprivate аналогично условию private за тем исключением, что указанные переменные инициализируются при входе в параллельный участок кода значением, которое имела переменная до входа в параллельную секцию.

Условие lastprivate аналогично private за тем исключением, что указанные переменные сохраняют свое значение, которое они получили при достижении конца параллельного участка кода в последнем элементе распределяемой работы.

Условие reduction гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.

Условие default определяет область видимости переменных внутри параллельного участка кода по умолчанию.

В OpenMP предусмотрены следующие конструкции синхронизации:

– critical – критическая секция;

– atomic – атомарность операции;

– barrier – точка синхронизации;

– ordered – выполнять блок в заданной последовательности;

– flush – немедленный сброс значений разделяемых переменных в память.

Наличие критической секции в параллельном блоке гарантирует, что она в каждый конкретный момент времени будет выполняться только одним потоком. То есть когда один поток находится в критической секции, все остальные потоки, которые готовы в нее войти, находятся в приостановленном состояние. Критические секции могут снабжаться именами. При этом критические секции считаются независимыми, только если они используют разные имена. По умолчанию, все непроименованые критические секции имеют одно имя.

Синтаксис критической секции на C/C++

```
#pragma omp critical [(имя)]
 Структурный блок
```

Барьеры – такой элемент синхронизации, который приостанавливает дальнейшее выполнение программы до тех пор, пока все потоки не достигнут этого барьера. Как только барьер достигнут всеми потоками, выполнение программы продолжается.

Синтаксис на C/C++

```
#pragma omp barrier
```

Директива ordered в параллельных циклах (только там она может встречаться) говорит о том, что указанный блок должен исполняться в строго фиксированной последовательности. Внутри ordered секции одновременно может находиться только один поток.

Конструкция flush осуществляет немедленный сброс значений разделяемых переменных в память. Таким образом гарантируется, что во всех потоках значение переменной будет одинаковое.

1.8.6 Процедуры библиотеки поддержки времени выполнения

Для эффективного использования процессорного времени компьютера и написания гибких OpenMP программ пользователю предоставляется возможность управлять ходом выполнения программы по средством

библиотечных функций. Библиотека OpenMP предоставляет пользователю следующий набор функций:

```
void omp_set_num_threads(int num_threads)
```

Устанавливает количество потоков, которое может быть запрошено для параллельного блока.

```
int omp_get_num_threads()
```

Возвращает количество потоков в текущей команде параллельных потоков.

```
int omp_get_max_threads()
```

Возвращает максимальное количество потоков, которое может быть установлено omp_set_num_threads.

```
int omp_get_thread_num()
```

Возвращает номер потока в команде (целое число от 0 до количества потоков – 1).

```
int omp_get_num_procs()
```

Возвращает количество физических процессоров доступных программе.

```
int omp_in_parallel()
```

Возвращает не нулевое значение, если вызвана внутри параллельного блока. В противном случае возвращается 0.

```
void omp_set_dyamic(expr)
```

Разрешает/запрещает динамическое выделение потоков.

```
int omp_get_dynamic()
```

Возвращает разрешено или запрещено динамическое выделение потоков.

```
void omp_set_nested(expr)
```

Разрешает/запрещает вложенный параллелизм.

```
int omp_get_nested()
```

Возвращает разрешен или запрещен вложенный параллелизм.

Перед использованием функций в C/C++ следует подключить файл заголовков omp.h.

Изменения, сделанные функциями, являются приоритетнее, чем соответствующие переменные окружения. Так, функция omp_set_num_threads() переписывает значение переменной окружения OMP_NUM_THREADS, которое может быть установлено перед запуском программы.

1.8.7 Переменные окружения

OMP_NUM_THREADS устанавливает количество потоков в параллельном блоке. По умолчанию, количество потоков равно количеству виртуальных процессоров.

OMP_SCHEDULE устанавливает тип распределения работ в параллельных циклах с типом runtime.

OMP_DYNAMIC разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений

(в зависимости от загрузки системы). Значение по умолчанию зависит от реализации.

OMP_NESTED разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию – запрещено.

Вопросы для самоконтроля

1. Какая директива позволяет создавать области параллельного выполнения?
2. Как указать, что переменная должна быть разделяемой между потоками?
3. Как работает барьер синхронизации?
4. Какая директива позволяет распределить между потоками выполнение итераций цикла?

1.9 Типичные архитектуры распределенных приложений

1.9.1 Клиент-серверная архитектура

Данный вид архитектуры распределённых приложений является в настоящее время наиболее распространённым для информационных систем. Существует мнение, что все остальные архитектуры могут быть представлены вариациями данной базовой архитектуры.

Данная информационная система представляет собой совокупность взаимодействующих компонент двух типов: клиентов и серверов. Как правило, данные компоненты разнесены по узлам двух типов: узлам-клиентам и узлам-серверам.

Клиенты обращаются к серверам с запросами, которые те обрабатывают и возвращают результат. Один клиент может обращаться с запросами к нескольким серверам. Серверы также могут обращаться с запросами друг к другу. Таким образом, типичный протокол взаимодействия может быть представлен в виде обмена сообщениями: запрос клиента – ответ сервера.

Наиболее часто встречающийся класс приложений, выполненных в архитектуре клиент-сервер, – это приложения, работающие с базами данных. В данном случае в качестве сервера выступает СУБД, обеспечивающая выполнение запросов клиента, который реализует интерфейс пользователя.

Далее представлены несколько моделей, реализующих архитектуру клиент-сервер.

Модель сервиса реализует ситуацию, когда услугу выполняет не один, а несколько серверов, представляемых клиенту как единое целое. То есть сервер имеет сложную структуру.

Данный вариант хорош для критичных сервисов, когда недопустима приостановка обслуживания. Для прекращения обслуживания необходима остановка всех серверов системы. Кроме того, такая система позволяет

сбалансированно распределять нагрузку между серверами. Таким, образом, может быть повышена как производительность, так и устойчивость к сбоям.

С другой стороны такая модель требует более сложной реализации по сравнению с базовой, и могут возникнуть проблемы с репликацией обрабатываемых на нескольких серверах данных или с поддержанием целостности распределённых данных.

Примером модели Proxy является привычная для нас структура: браузер – proxy-сервер – web-сервер.

Отличие от предыдущих моделей заключается в том, что клиент соединяется не с сервером, а с некоторым промежуточным компонентом (посредником).

Посредник может сам решать, какому из серверов передать запрос клиента (можно с его помощью распределять нагрузку). Кроме того, посредник может сохранять последние запросы клиента, чтобы при следующем обращении вернуть ему ответ, не обращаясь к серверу.

Недостаток: при неправильном построении посредник может стать узким местом системы в целом.

При работе в интернет чаще всего используется именно такой подход.

В течение нескольких последних лет наблюдается постоянное увеличение количества применяемых портативных устройств – сотовых телефонов, PDA, планшетов и т.д. Возникает естественное желание использовать такие устройства как средства для работы с информационными системами (например, зайти на wap-сайт туристического агентства и заказать путевку прямо с сотового телефона). Однако, интерфейс, предоставляемый браузерами, весьма ограничен. С другой стороны, в силу ограниченной мощности мобильных устройств в них пока не удается размещать приложения со сложной бизнес-логикой.

Решить эту проблему можно, используя технологию «тонкого клиента». Суть этой технологии состоит в том, что клиент выполняет очень ограниченную по функционалу задачу (часто – только прием ввода с клавиатуры и других устройств и обработка команд рисования). Схема работы подобных систем в простейшем случае следующая. Клиентская программа передает весь ввод пользователя (нажатия клавиш, движение мыши и т.д.) по сети серверу.

1.9.2 Архитектура P2P (Peer to Peer)

Другой архитектурой, встречающейся, в основном, в специальных областях, является архитектура P2P. Приложение, выполненное в такой архитектуре, не имеет четкого разделения на серверные и клиентские модули – все его части равноправны и могут выполняться на любых узлах.

Таким образом, на одном и том же узле в один момент выполняются части системы, обрабатывающие запросы других частей (и узел выполняет

«серверную» часть), а в другой момент времени – части системы, посылающие запросы (и узел выполняет «клиентскую» часть). Причем приложение может быть устроено так, что вызывающая часть не знает, локально или удаленно расположена вызываемая.

Построенные таким образом приложения обладают уникальными свойствами – их части никак не привязаны друг к другу и к узлам, на которых они исполняются. Таким образом, от запуска к запуску может меняться состав модулей, расположенных на узле. Это позволяет организовывать очень изощренные политики распределения нагрузки, а также обеспечивать очень хорошие показатели масштабируемости и отказоустойчивости.

В отличие от архитектуры клиент-сервер, такая организация позволяет сохранять работоспособность сети при любом количестве и любом сочетании доступных узлов.

Такая архитектура активно используется для разработки параллельных вычислительных систем для решения сложных вычислительных задач.

Помимо чистых P2P-сетей, существуют так называемые гибридные сети, в которых существуют серверы, используемые для координации работы, поиска или предоставления информации о существующих машинах сети и их статусе (on-line, off-line и т. д.). Гибридные сети сочетают скорость централизованных сетей и надёжность децентрализованных благодаря гибридным схемам с независимыми индексационными серверами, синхронизирующими информацию между собой. При выходе из строя одного или нескольких серверов, сеть продолжает функционировать. К частично децентрализованным файлообменным сетям относятся, например EDonkey, BitTorrent.

Вопросы для самоконтроля

1. За что в клиент-серверной архитектуре отвечает сервер?
2. За что в клиент-серверной архитектуре отвечает клиент?
3. Что обозначает понятие сервис в клиент-серверной архитектуре?
4. Какова роль посредника (proxy) в клиент-серверной архитектуре?
5. В чем отличие «тонкого клиента»?
6. Каковы преимущества архитектуры P2P?

1.10 Использование сокетов (API java.net)

1.10.1 Interprocess communication (IPC)

Межпроцессное взаимодействие (англ. inter-process communication, IPC) – обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

Из механизмов, предоставляемых ОС и используемых для IPC, можно выделить:

– механизмы обмена сообщениями;

– механизмы синхронизации;

– механизмы разделения памяти;

– механизмы удалённых вызовов (RPC).

Для оценки производительности различных механизмов IPC используют следующие параметры:

– пропускная способность (количество сообщений в единицу времени, которое ядро ОС или процесс способно обработать);

– задержки (время между отправкой сообщения одним потоком и его получением другим потоком).

IPC может называться терминами межпотоковое взаимодействие и межпрограммное взаимодействие.

Межпроцессное взаимодействие, наряду с механизмами адресации памяти, является основой для разграничения адресного пространства между процессами.

Разделяемая память (англ. Shared memory) является самым быстрым средством обмена данными между процессами.

В других средствах межпроцессового взаимодействия (IPC) обмен информацией между процессами проходит через ядро, что приводит к переключению контекста между процессом и ядром, т.е. к потерям производительности.

Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

Некоторые библиотеки языка C++ предлагают доступ к работе с разделяемой памятью в кроссплатформенном виде. Например, библиотека Boost предоставляет класс boost::interprocess::shared_memory_object для POSIX-совместимых операционных систем, а библиотека Qt предоставляет класс QSharedMemory, унифицирующий доступ к разделяемой памяти для разных операционных систем с некоторыми ограничениями.

В Java 7 под операционной системой GNU/Linux разделяемая память может быть реализована отображением файла из каталога /dev/shm/ (либо /run/shm/, в зависимости от дистрибутива) в память с помощью метода map класса java.nio.MappedByteBuffer.

Поддержка разделяемой памяти осуществлена во многих других языках программирования. Так, PHP предоставляет API для создания разделяемой памяти, чьи функции схожи с функциями POSIX.

Обмен сообщениями – один из подходов реализации взаимодействия компонентов и систем, используемый в параллельных вычислениях, объектно-ориентированном программировании, также – одна из форм межпроцессного взаимодействия в операционных системах, в микроядерных операционных системах подход используется для обмена информацией между одним из ядер и одним или более исполняющих блоков.

Распределённые системы доступа к объектам и удалённого вызова методов, вида ONC RPC, CORBA, RMI, DCOM, SOAP, .Net_Remoting, QNX Neutrino RTOS, OpenBinder, D-Bus и им подобные являются системами обмена сообщениями. Широкое применение подходов с обменом сообщениями также свойственно высокопроизводительным вычислениям, в частности, на нём основан интерфейс передачи сообщений MPI. В классе связующего программного обеспечения выделяется особая группа – промежуточное программное обеспечение, ориентированное на обработку сообщений, базирующееся на данном подходе.

1.10.2 Сокеты

Сокет – название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет – абстрактный объект, представляющий конечную точку соединения.

Следует различать клиентские и серверные сокеты. Клиентские сокеты грубо можно сравнить с конечными аппаратами телефонной сети, а серверные – с коммутаторами. Клиентское приложение (например, браузер) использует только клиентские сокеты, а серверное (например, веб-сервер, которому браузер посылает запросы) – как клиентские, так и серверные сокеты.

Интерфейс сокетов впервые появился в BSD Unix. Программный интерфейс сокетов описан в стандарте POSIX.1 и в той или иной мере поддерживается всеми современными операционными системами.

Для взаимодействия между машинами с помощью стека протоколов TCP/IP используются адреса и порты. Адрес представляет собой 32-битную структуру для протокола IPv4, 128-битную для IPv6. Номер порта – целое число в диапазоне от 0 до 65535 (для протокола TCP).

Эта пара определяет сокет («гнездо», соответствующее адресу и порту).

В процессе обмена, как правило, используется два сокета – сокет отправителя и сокет получателя. Например, при обращении к серверу на HTTP-порт сокет будет выглядеть так: 194.106.118.30:80, а ответ будет поступать на mmm.nnn.ppp.qqq: xxxxx.

Каждый процесс может создать «слушающий» сокет (серверный сокет) и привязать его к какому-нибудь порту операционной системы (в UNIX непривилегированные процессы не могут использовать порты меньше 1024).

Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и т. д.

Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (файл сокета) по заданному пути, через который смогут сообщаться любые локальные процессы путём чтения/записи из него (см. сокет домена Unix). Сокеты типа INET доступны из сети и требуют выделения номера порта.

Обычно клиент явно «подсоединяется» к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

1.10.3 Сокетные соединения по протоколу TCP из стека TCP/IP

В рамках пакета java.net программисту предоставляется абстрактный сервис, основанный на использовании потоков. Для передачи данных программист создает TCP-сокет, извлекает из него связанный с ним поток (с сокетом связано два потока – поток ввода и поток вывода) и пишет (читает) данные в поток (из потока). Рассматриваемый подход является ориентированным на соединение. Реализация протокола основана на повторной передаче данных, в случае если истек тайм-аут по ожиданию от получателя подтверждения о приеме.

Классы:

– ServerSocket – сокет на стороне сервера;

– Socket – класс для работы с соединением (клиент и сервер). Имеет конструктор для создания сокета и соединения с удаленным узлом и портом, методы для работы с входными и выходными потоками.

Для организации обмена информацией по протоколу TCP обязательно требуется открыть соединение между конечными точками (характеризующимися IP-адресом и портом). При этом будут использоваться клиентские (их может быть несколько) и серверные (обычно он существует

в единственном экземпляре) сокеты. Клиентские сокеты (на стороне клиента) пытаются открыть соединение с удаленной конечной точкой (на стороне сервера), а серверный сокет (на стороне сервера) ожидает или «слушает» входящие соединения. В общих чертах последовательность действий описывается следующим образом:

на стороне клиента:

– создание клиентского сокета;

– установка параметров сокета (IP-адрес и порт, к которым необходимо подключиться);

– установка соединения между сокетом и удаленной конечной точкой;

– отправка/получение информации;

– разрыв соединения и освобождение сокета.

на стороне сервера:

– создание серверного сокета;

– установка параметров серверного сокета (IP-адрес и порт, на которые ожидаются подключения);

– перевод серверного сокета в режим отслеживания входящих соединений;

– при наличии входящего соединения: получить отдельный сокет для работы с этим конкретным соединением;

– отправка/получение информации;

– по окончании работы с клиентом: разрыв соединения и освобождение сокета, привязанного к этому клиенту;

– по окончании работы сервера: освобождение серверного сокета.

Ниже приведена простая программа-клиент, иллюстрирующая использование рассмотренных классов.

```
import java.net.*;
import java.io.*;
public class TCPClient {
 public static void main (String args[]) {
  Socket s = null;
  try {
   int serverPort = 7896;
   s = new Socket(args[1], serverPort);
   DataInputStream in =
                    new DataInputStream(s.getInputStream());
   DataOutputStream out =
                    new DataOutputStream(s.getOutputStream());
   out.writeUTF(args[0]);
   String data = in.readUTF();
   System.out.println("Received: "+ data);
  } catch (UnknownHostException e) {
   System.out.println("Socket:" + e.getMessage());
  } catch (EOFException e) {
   System.out.println("EOF:" +e.getMessage());
```

```
  } catch (IOException e) {
   System.out.println("readline:" +e.getMessage());
  } finally {
   if(s!=null) try {s.close();} catch (IOException e)
  {System.out.println("close:" + e.getMessage());}}
 }
}
```

Класс TCPClient имеет единственный метод main,который также является точкой входа в клиентскую программу. Метод принимает аргументы (аргументы командной строки), первый из которых рассматривается как строка, передаваемая клиентом серверу, второй – как имя (адрес) сервера. Поскольку создание соединения и передача по нему данных сопряжена с возможностью ошибок, остальные действия производятся в блоке try – catch.

Переменная s, объявленная в строке 5, инициализируется в строке 8, где создается соединение с сервером. Для соединения требуются два параметра – имя сервера и порт. Имя сервера передано нам из командной строки, порт нам известен. В строках 9 и 11 создаются потоки ввода и вывода, с помощью которых можно взаимодействовать с сервером – передавать и принимать данные. В данном случае используются не базовые классы InputStream и OutputStream, а их более специализированные потомки DataInputStream и DataOutputStream, которые содержат готовые методы чтения/записи для всех базовых типов данных Java. Затем производится отправка полученной строки на сервер, после чего клиент ожидает получения информации от сервера (строка 14). Метод чтения данных из потока – блокирующий. Прочитав строку, посланную сервером, клиент печатает ее на экране и завершает работу.

Здесь стоит сделать небольшое отступление. Дело в том, что нам очень повезло: в нашем примере и клиент, и сервер реализованы на одной и той же программной платформе – на java. В реальных ситуациях дело может осложняться тем, что различные компоненты распределенного приложения могут быть реализованы на различных не только программных, но и аппаратных платформах. При передаче данных между такими «разными» компонентами возникает проблема их интерпретации – ведь форматы представления могут быть различными для разных платформ.

Сервер, принимающий данные от клиента и посылающий ему ответ, выполнен следующим образом

```
import java.net.*;
import java.io.*;
public class TCPServer {
 public static void main (String args[]) {
  try {
   int serverPort = 7896;
   ServerSocket listenSocket = new ServerSocket(serverPort);
```

```
    while(true) {
     Socket clientSocket = listenSocket.accept();
     Connection c = new Connection(clientSocket);
    }
   } catch(IOException e) {
    System.out.println("Listen socket:"+e.getMessage());
   }
  }
}
```

Как и клиенту, серверу необходима реализация сетевого ввода/вывода, поэтому он импортирует соответствующие пакеты – строки 1, 2. Так же как и клиент, сервер состоит из одного метода – main, выполняющего всю работу и одновременно являющегося точкой входа в программу. Первое, что делает сервер – создает серверный сокет (строка 7). Для создания серверного сокета необходим один параметр – порт, на котором сокет будет «слушать» сеть, ожидая клиентских соединений – именно по этому порту клиенты смогут затем соединиться с сервером, и, соответственно, этот порт должен указываться в конструкторах клиентских сокетов, наряду с адресом сервера. В случае если создание серверного сокета прошло успешно, сервер запускает цикл ожидания соединений от клиентов. Внутри цикла ожидания сервер вызывает метод accept (строка 9) своего серверного сокета. Этот метод является блокирующим, то есть он возвратит управление только тогда, когда к серверу подсоединится очередной клиент. Можно сказать, что большую часть времени сервер проводит именно в методе accept, ожидая соединения клиентов. Метод accept возвращает в качестве результата своей работы класс Socket, который, наряду с сокетом, созданным на клиенте, представляет собой второй конец соединения. Начиная с этого момента, клиент и сервер могут обмениваться друг с другом данными, используя соответствующие методы потоков, связанных со своими сокетами. Поскольку наш сервер может одновременно взаимодействовать с несколькими клиентами, мы создаем класс Connection, который инкапсулирует в себе всю функциональность обслуживания соответствующего клиента, после чего снова входим в цикл ожидания подсоединения следующего клиента. Следует обратить внимание на тот факт, что использованное API позволяет серверу одновременно обслуживать несколько клиентов, поскольку для каждого клиента после его подсоединения создается собственный канал типа «точка-точка».

1.10.4 Датаграммы и протокол UDP

UDP является протоколом, применяющимся для передачи датаграммы. Этот протокол не является надежным, поскольку сообщения, передаваемые с его помощью, могут теряться или приходить в другой последова-

тельности, отличной от последовательности их отправки. Таким образом, для обеспечения надежной передачи необходимо организовывать надстройку над этим протоколом, обеспечивающую, например, нумерацию пакетов, повторную передачу пакетов при истечении времени ожидания и так далее. Длина одного сообщения (одной датаграммы) при использовании этого протокола ограничена 65536 байтами (причем многие реализации вообще ограничивают размер датаграммы 8 Кб), в случае необходимости пересылки порции данных большего размера они должны быть разбиты на куски отправителем и снова собраны получателем. Передача сообщения – не блокирующая, прием – блокирующий с возможностью прерывания по истечении времени ожидания.

Для работы с UDP в пакете java.net определены следующие классы.

DatagramPacket (датаграмма). Конструктор этого класса принимает массив байт и адрес процесса-получателя (IP-адрес узла и порт). Класс предназначен для представления единичной дата-граммы (сообщения). Этот класс используется как для создания сообщения с целью последующей его отправки, так и при приеме сообщения (функция приема возвращает экземпляр этого класса).

DatagramSocket. Предназначен для посылки/приема UDP-датаграмм. Один из конструкторов принимает в качестве аргумента порт, с которым связывается сокет, другой конструктор, без аргументов, задействует в качестве порта первый попавшийся свободный порт. Класс имеет методы send и receive, для, соответственно, передачи и приема датаграмм. Метод setSoTimeout устанавливает тайм-аут для операций сокета.

Ниже приведены две простые программы, использующие рассмотренные механизмы для организации взаимодействия.

Первая программа создает сокет, соединяется с сервером (порт 6789), пересылает ему сообщение и ждет ответа.

```
import java.net.*;
import java.io.*;
public class UDPClient{
 public static void main(String args[]){
  try {
   DatagramSocket aSocket = new DatagramSocket();
   byte [] message = args[0].getBytes();
   InetAddress aHost = InetAddress.getByName(args[1]);
   int serverPort = 6789;
   DatagramPacket request =
     new DatagramPacket(message, args[0].length(), aHost,
                        serverPort);
   aSocket.send(request);
   byte[] buffer = new byte[1000];
   DatagramPacket reply = new DatagramPacket(buffer,
                                             buffer.length);
   aSocket.receive(reply);
```

```
   System.out.println("Reply: " + new
String(reply.getData()));
   aSocket.close();
  } catch (SocketException e){
   System.out.println("Socket: " + e.getMessage());
  } catch (IOException e){
   System.out.println("IO: " + e.getMessage());
  }
 }
}
```

В первых строках (строки 1, 2) происходит импорт необходимых библиотек (пакетов) классов – в нашем случае это java.net и java.io. Первый пакет содержит необходимые нам классы для работы с UDP, второй определяет необходимые классы ввода/вывода.

Наш класс (строка 3) называется UDPClient, он содержит единственный статический метод main (строка 4), являющийся точкой входа в программу. Метод принимает массив аргументов командной строки, переданных программе при запуске. В качестве аргументов для нашей программы должны быть переданы: строка сообщения (которое будет оправлено на сервер) и адрес узла, на котором запущена программа, – сервер (порт не передается, поскольку в нашем случае он заранее известен). Далее создается сокет для передачи сообщения (строка 6), затем происходит разрешение переданного в качестве аргумента командной строки имени узла сервера в адрес (строка 8). Затем создается датаграмма (строки 10–12), в конструкторе которой передается массив, составляющий передаваемые данные, адрес узла, на котором выполняется сервер и порт (который нам известен заранее). Пакет передается на сервер (строка 13). Обратите внимание – адрес получателя определяется в пакете, а не в сокете, через который мы его передаем. То есть один и тот же сокет может быть использован для передачи пакетов разным узлам, и этот же сокет может применяться для приема пакета от сервера (строка 17). После использования сокет должен быть закрыт (строка 19).

Другая программа (сервер) создает сокет и обслуживает запросы клиента.

```
import java.net.*;
import java.io.*;
public class UDPServer{
 public static void main(String args[]) {
  DatagramSocket aSocket = null;
  try{
   aSocket = new DatagramSocket(6789);
   byte[] buffer = new byte[1000];
   while(true){
    DatagramPacket request = new DatagramPacket(buffer,
```

```
                                                    buffer.length);
     aSocket.receive(request);
     DatagramPacket reply =
        new DatagramPacket(request.getData(),
                           request.getLength(),
                           request.getAddress(),
                           request.getPort());
     aSocket.send(reply);
    }
   } catch (SocketException e) {
    System.out.println("Socket: " + e.getMessage());
   } catch (IOException e) {
    System.out.println("IO: " + e.getMessage());
   } finally {
    if(aSocket != null) aSocket.close();
   }
  }
}
```

Вопросы для самоконтроля

1. Какие существуют средства межпроцессного взаимодействия?
2. Что такое сокет?
3. Чем серверный сокет отличается от клиентского?
4. Как с помощью сокетов в java.net подключится к серверу по протоколу TCP?
5. Какие классы в java.net используются для передачи данных по протоколу UDP?

1.11 Удаленный вызов методов (RMI)

1.11.1 Концепции RMI

Java RMI (Remote Method Invocation – удаленный вызов методов) представляет собой тип удаленного вызова процедур, независимый от сети, облегченный и полностью переносимый, так как написан на языке Java.

Технология RMI основана на более ранней подобной технологии RPC (Remote Procedure Call – удаленного вызова процедур) для процедурного программирования, разработанной в 80-х годах. RPC позволяет процедуре вызывать функцию на другом компьютере столь же легко, как если бы эта функция была частью программы, выполняющейся на том же компьютере. Для устранения недостатков RPC была разработана технология RMI, которая обслуживает маршалинг данных через сеть и дает возможность программам на Java передавать законченные объекты Java с помощью механизма сериализации объектов Java.

Сериализация – это процесс сохранения состояния объекта в последовательность байт; десериализация это процесс восстановления объекта, из этих байт. Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов.

Маршалинг – по смыслу похож на сериализацию, процесс преобразования представления объекта в памяти в формат данных, пригодный для хранения или передачи. Применительно же к компьютерным сетям, маршалинг означает процесс упаковки данных и преобразования их в стандартный вид перед передачей по сети так, чтобы данные могли пройти через сетевые ограничители. Чтобы передать объект во внешнюю сеть, он должен быть преобразован в поток данных, соответствующий структуре пакетов сетевого протокола. Части данных содержатся в буфере до того момента, пока не будут упакованы. Когда данные переданы, компьютер-получатель преобразует упакованные данные обратно в объект.

1.11.2 Применение RMI

RMI дает возможность выполнять объекты Java на различных компьютерах или в отдельных процессах путем взаимодействия их друг с другом посредством удаленных вызовов методов. Технология RMI основана на более ранней подобной технологии удаленного вызова процедур (RPC) для процедурного программирования, разработанной в 80-х годах. RPC позволяет процедуре вызывать функцию на другом компьютере столь же легко, как если бы эта функция была частью программы, выполняющейся на том же компьютере. RPC выполняет всю работу по организации сетевых взаимодействий и маршалинга данных (то есть пакетирования параметров функций и возврата значений для передачи их через сеть). Но RPC не подходит для передачи и возврата объектов Java, потому что она поддерживает ограниченный набор простых типов данных. Есть и другой недостаток у RPC – программисту необходимо знать специальный язык определения интерфейса (IDL) для описания функций, которые допускают удаленный вызов. Для устранения этих недостатков и была разработана технология RMI.

RMI представляет собой реализацию RPC на Java для распределенных коммуникационных взаимодействий «Java-объект – Java-объект». Объект Java регистрируется для удаленного доступа, что дает возможность клиентам получать удаленную ссылку на этот объект – она позволяет использовать этот объект дистанционно. Синтаксис вызова метода идентичен синтаксису вызова методов других объектов в той же программе. RMI обслуживает маршалинг данных через сеть и дает возможность программам на Java передавать законченные объекты Java с помощью механизма сериализации объектов Java. В составе J2SE имеются инструментальные средства создания требуемого кода для сетевых взаимодействий из определенных

интерфейсов программы, это означает, что RMI не требует от программиста знания языка IDL. Кроме того, никакого нейтрального к языку IDL интерфейса не требуется, так как RMI поддерживает только Java; достаточно собственных интерфейсов Java.

1.11.3 Создание распределенной системы с помощью RMI

Здесь рассмотрен пример, использующий RMI. При этом предлагается несколько вариантов решения поставленной задачи.

В примере выполняются четыре основных действия:

– определение удаленного интерфейса с объявлениями методов, которые клиент может вызвать у удаленного объекта;

– определение реализации удаленного объекта для удаленного интерфейса;

– определение клиентского приложения, которое взаимодействует с реализацией интерфейса;

– компиляция и выполнение удаленного объекта и клиента.

Решать предложенную задачу мы будем поэтапно – постепенно наращивая сложность наших программ, при этом иллюстрируя те или иные особенности применяемых технологий.

Для начала рассмотрим совсем простую задачу: создадим систему, функционально состоящую из двух компонентов – сервера (процессингового центра) и клиента (касса). Будем предполагать, что сервер в системе один (именно он обладает всей информацией о зарегистрированных картах и их балансах), а касс много и на них проходят операции регистрации новых карт, а также операции изменения баланса карт (соответственно – оплаты и занесения наличных).

На данном этапе мы никак не будем учитывать то, что связь между нашими объектами неустойчивая, – более того, мы будем считать ее устойчивой. Кроме того, в целях сокращения размеров примеров не будут рассматриваться вопросы долговременного хранения обрабатываемой информации – при перезапуске наша система будет «забывать» все занесенные в нее карты и их балансы.

При проектировании сервера нам необходимо определить выполняемые им функции и, соответственно, методы, которые будет вызывать клиент. Поскольку операциями нашей системы являются регистрация новой карты, занесение денег на карту, оплата картой покупки и просмотр баланса карты, будем считать, что наш сервер будет обладать четырьмя перечисленными методами – именно их клиент (касса) и будет вызывать. Поскольку единственным, что отличает одну карту от другой, является ее номер (код) – он и будет служить идентификатором, карты и поэтому будет присутствовать во всех методах сервера в качестве параметра. Поскольку

этот код, вообще говоря, алфавитно-цифровой, будем использовать тип данных String для его хранения.

Удаленные методы, посредством которых клиент взаимодействует с удаленным объектом, используя RMI, должны быть определены в удаленном интерфейсе. Соответственно, первый этап при создании распределенного приложения с помощью RMI состоит в определении удаленного интерфейса, который описывает эти удаленные методы. Чтобы создать удаленный интерфейс, необходимо определить интерфейс, который будет расширять интерфейс java.rmi.Remote. Интерфейс Remote представляет собой тегирующий интерфейс – он не объявляет каких-либо методов, поэтому не обременен реализацией класса. Распределенное RMI-приложение должно экспортировать объект класса, который реализует интерфейс Remote, чтобы сделать этот удаленный объект доступным для приема удаленных вызовов метода из любой виртуальной машины Java, которая имеет соединение с компьютером, где выполняется удаленный объект.

Интерфейс BillingService, который расширяет интерфейс Remote, представляет собой интерфейс для нашего удаленного объекта (сервера). В нем объявляются методы для работы с пластиковыми картами. Удаленный объект должен реализовать все объявленные в удаленном интерфейсе методы.

```
// BillingService.java
// Интерфейс BillingService объявляет методы для работы
// с пластиковыми картами
package ex1;

// Набор базовых пакетов Java
import java.rmi.*;

public interface BillingService extends Remote {
 // определение новой карты
 public void addNewCard(String personName, String card)
   throws RemoteException;
 // добавить денежные средства на карту
 public void addMoney(String card, double money)
   throws RemoteException;
 // снять денежные средства с карты
 public void subMoney(String card, double money)
   throws RemoteException;
 // получение баланса карты
 public double getCardBalance(String card)
   throws RemoteException;
}
```

Когда узлы взаимодействуют между собой по сети, есть вероятность возникновения проблем при таких взаимодействиях. Например, компьютер сервера может выйти из строя или может отказать какой-либо сетевой

ресурс. Поэтому для контроля подобных проблем взаимодействия в сети каждый метод в интерфейсе Remote должен содержать throws для указания, что метод может возбуждать контролируемые исключения RemoteException.

RMI использует механизм сериализации по умолчанию Java для передачи параметров методу и возврата значений через сеть. В связи с этим все параметры метода и возвращаемые значения должны иметь описатель Serializable или один из примитивных типов.

Следующим этапом является определение реализации удаленного объекта. В нашем случае класс реализации удаленного объекта имеет то же имя, что и удаленный интерфейс, но заканчивается на Impl.

Класс UnicastRemoteObject (пакет java.rmi.server) представляет базовые функциональные возможности, которые необходимы удаленным объектам для обслуживания удаленных запросов.

Конструкторы и методы класса UnicastRemoteObject возбуждают контролируемое исключение RemoteException,поэтому подклассы класса UnicastRemoteObject должны определять конструкторы, которые также возбуждают исключение RemoteException.

Конструктор класса UnicastRemoteObject экспортирует объект, чтобы сделать его доступным для приема удаленных вызовов. Экспорт объекта дает возможность удаленному объекту ожидать соединений с клиентами на анонимном порте (то есть порте, выбираемом компьютером, на котором выполняется удаленный объект). Это дает возможность объекту осуществлять однонаправленное взаимодействие (взаимодействие «точка-точка» между двумя объектами посредством вызовов методов) с использованием стандартных соединений через сокеты. Классам удаленных объектов не нужно расширять этот класс, если эти классы применяют статический метод exportObject класса UnicastRemoteObject для экспорта удаленных объектов. Предполагается, что клиенты RMI должны осуществлять соединение на порте 1099 при попытке найти удаленный объект в реестре RMI. Перегруженный конструктор для класса UnicastRemoteObject дает возможность задавать дополнительную информацию, такую как номер порта для экспорта удаленного объекта. Для этого необходимо определить URL, который клиент может использовать для получения удаленной ссылки на объект. Эта ссылка применяется для вызова методов удаленного объекта. URL обычно имеет форму

```
rmi://хост:порт/ИмяУдаленногоОбъекта,
```

где хост представляет собой имя компьютера, который выполняет сервер реестра (rmiregistry) для удаленных объектов (он также является компьютером, на котором выполняется удаленный объект), порт представляет собой номер порта, на котором выполняется сервер реестра на хост-компьютере, а ИмяУдаленногоОбъекта – имя, которое клиент будет предоставлять при попытках обнаружить удаленный объект в реестре. Утилита

rmi-registry обслуживает реестр удаленных объектов и является составной частью J2SE. Номер порта реестра RMI по умолчанию – 1099.

Для связывания удаленного объекта с реестром используются методы bind или rebind. Метод rebind гарантирует, что если объект уже был зарегистрирован под заданным именем, новый удаленный объект заменит ранее зарегистрированный объект. Это может быть важно, если регистрируется новая версия существующего удаленного объекта.

Класс BillingServiceImpl представляет собой удаленный объект, который реализует удаленный интерфейс BillingService.Клиент взаимодействует с объектом класса BillingServiceImpl,вызывая методы addNewCard, addMoney, subMoney, getCardBalance интерфейса BillingService для обработки информации по пластиковым картам. Класс BillingServiceImpl хранит сведения о картах в хэш-таблице (Hashtable), содержащей баланс пластиковой карты с именем посетителя (personName) и номером пластиковой карты (card), где номер карты является ключом таблицы.

```
// BillingServiceImpl.java
// BillingServiceImpl реализует удаленный интерфейс BillingS-
ervice
// для предоставления удаленного объекта BillingService
package ex1;

// Набор базовых пакетов Java
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class BillingServiceImpl extends UnicastRemoteObject
implements BillingService {

 private   Hashtable hash;    // хэш-таблица для хранения карт
                              // инициализация сервера
 public BillingServiceImpl() throws RemoteException {
  super();
  hash = new Hashtable();
 }

 // реализация метода addNewCard интерфейса BillingService
 public void addNewCard(String personName, String card)
 throws RemoteException {
  hash.put(card, new Double(0.0));
 }

 // реализация метода addMoney интерфейса BillingService
 public void addMoney(String card, double money)
 throws RemoteException {
  Double d = (Double)hash.get(card);
  if (d!=null)
```

```
   hash.put(card,new Double(d.doubleValue()+money));
  else
   throw new NotExistsCardOperation();
 }

 // реализация метода subMoney интерфейса BillingService
 public void subMoney(String card, double money)
 throws RemoteException {
  Double d = (Double)hash.get(card);
  if (d!=null)
   hash.put(card,new Double(d.doubleValue()-money));
  else
    throw new NotExistsCardOperation();
 }

 // реализация метода getCardBalance интерфейса BillingService
 public double getCardBalance(String card)
 throws RemoteException {
  Double d = (Double)hash.get(card);
  if (d!=null)
   return d.doubleValue();
  else
   throw new NotExistsCardOperation();
 }

 // запуск удаленного объекта BillingService
 public static void main (String[] args)
 throws Exception {
  System.out.println("Initializing BillingService...");

  // создание удаленного объекта
  BillingService service = new BillingServiceImpl();

  //задание имени удаленного объекта
  String serviceName = "rmi://localhost/BillingService";
  // регистрация удаленного объекта BillingService в реестре
rmiregistry
  Naming.rebind(serviceName, service);
 }

}
```

Класс BillingServiceImpl реализует методы addNewCard, addMoney, subMoney, getCardBalance интерфейса BillingService,чтобы отвечать на удаленные запросы.

Метод main создает удаленный объект BillingServiceImpl. Когда конструктор выполняется, он экспортирует удаленный объект, чтобы прослушивать удаленные запросы. Затем определяется URL, который клиент

может применить для получения удаленной ссылки на объект для вызова методов удаленного объекта.

В этой программе URL удаленного объекта имеет вид rmi://localhost/BillingService. Из этого следует, что реестр RMI выполняется на машине localhost (т.е. на локальном компьютере), а для обнаружения клиентом сервиса нужно использовать имя BillingService. Имя localhost является синонимом IP-адреса 127.0.0.1.

Далее вызывается статический метод rebind класса Naming (пакет java.rmi) для связывания удаленного объекта service класса BillingServiceImpl в реестре RMI с URL rmi://localhost/BillingService.

Класс NotExistsCardOperation расширяет класс RemoteException.

Далее мы определяем клиентское приложение, которое будет обрабатывать запросы к пластиковым картам и пересылать их серверу. Для работы клиентского приложения необходимо знать URL вызываемого им удаленного объекта. Статический метод lookup класса Naming применяется для получения объектной ссылки на удаленный объект с заданным URL. Метод lookup осуществляет соединение с реестром RMI и возвращает удаленную ссылку на удаленный объект. Клиент может использовать эту удаленную ссылку, если она обращается к локальному объекту, выполняющемуся на той же виртуальной машине. Эта удаленная ссылка обращается к объекту-заглушке на клиенте. Заглушки дают возможность клиентам вызывать методы удаленного объекта. Объекты-заглушки принимают удаленные вызовы метода и передают эти вызовы RMI, который выполняет сетевые соединения, позволяющие клиентам взаимодействовать с удаленным объектом. Уровень RMI отвечает за сетевые соединения с удаленными объектами, поэтому обращения к удаленным объектам являются прозрачными для пользователя. RMI обслуживает соединение с удаленным объектом, передачу параметров и возврат значений.

Класс BillingClient является клиентским приложением, которое вызывает удаленные методы addNewCard, addMoney и getCardBalance интерфейса BillingService для работы с пластиковыми картами посетителей через RMI. В нашем примере мы не реализуем реальную работу касс столовых с пластиковыми картами. Для иллюстрации определения и работы клиентского приложения нам достаточно произвести несколько операций с несколькими картами, используя удаленные методы удаленного объекта. В данном случае клиент в цикле заносит денежные средства на три пластиковые карты и в конце печатает результирующий баланс по этим пластиковым картам. При первом проходе цикла в случае отсутствия карт с заданными номерами клиент их создает.

```
// BillingClient.java
// BillingClient использует удаленный объект BillingService
для
// работы с информацией на пластиковых картах
```

```
package ex1;

// Набор базовых пакетов Java
import java.rmi.*;

public class BillingClient {
 // выполнение BillingClient
 public static void main(String[] args) throws Exception
  // создание строки, содержащей URL удаленного объекта
  String objectName = "rmi://"+args[0]+"/BillingService";
  System.out.println("Starting...\n");
  // соединение с реестром RMI и получение удаленной ссылки
  // на удаленный объект
  BillingService bs =
        (BillingService)Naming.lookup(objectName);
  System.out.println("done");

  // начисление денежных средств на пластиковые карты
  for (int i = 0; i < 10000;  i++) {
   try {
    bs.addMoney("1", 1);
   } catch (RemoteException   e) {
    bs.addNewCard("Piter", "1");
   }
   try {
    bs.addMoney("2", 1);
   } catch (RemoteException   e) {
    bs.addNewCard("Stefan", "2");
   }
   try {
    bs.addMoney("3", 1);
   } catch (RemoteException   e) {
    bs.addNewCard("Nataly", "3");
   }
  }
  // печать текущего баланса обработанных карт
  System.out.println("1:"+bs.getCardBalance("1"));
  System.out.println("2:"+bs.getCardBalance("2"));
  System.out.println("3:"+bs.getCardBalance("3"));
 }
}
```

Метод main принимает в качестве параметра имя компьютера, на котором выполняется удаленный объект BillingService. Затем создается строка objectName, которая содержит URL для нашего удаленного объекта. Потом вызывается метод lookup класса Naming для получения удаленной ссылки на удаленный объект BillingService с заданным URL. Далее производится добавление денежных средств на карты, а потом печать текущего баланса этих карт.

Подготовив отдельные фрагменты, мы можем сформировать и выполнить наше распределенное приложение, но для этого потребуется несколько действий. Для начала необходимо компилировать исходные классы. Далее, нужно компилировать класс удаленного объекта (…Impl), используя компилятор rmic (утилита J2SE) для формирования класса-заглушки (о котором говорилось в предыдущем разделе). Этот класс должен быть доступен для клиента (либо локально, либо путем загрузки по сети), чтобы дать возможность устанавливать удаленное соединение с серверным объектом. В зависимости от параметров командной строки, передаваемых rmic, может быть сгенерировано несколько файлов. В Java 1.1 rmic формирует два класса – класс-заглушку и класс-каркас (skeleton). В Java 2 класс-каркас больше не требуется. Параметр командной строки -v1.2 указывает, что rmic следует создать только класс-заглушку.

Для нашего примера командная строка для компиляции класса удаленного объекта будет выглядеть следующим образом:

```
rmic -v1.2 com.asw.rmi.ex1.BillingServiceImpl
```

которая сгенерирует файл BillingServiceImpl_Stub.class.

Следующий этап – запуск реестра RMI, который зарегистрирует удаленный объект. Командная строка

```
rmiregistry
```

запускает реестр RMI на локальной машине. В окне командной строки в ответ на эту команду никакого текста отображаться не будет. Типичная ошибка заключается в том, что если не запустить реестр RMI прежде чем попытаться привязать удаленный объект к реестру, будет сгенерировано исключение java.rmi.ConnectException, которое указывает, что программа не может соединиться с реестром.

Чтобы удаленный объект мог принимать удаленные вызовы методов, необходимо связать объект с именем в реестре RMI. Для этого нужно запустить серверное приложение из командной строки. В нашем случае командная строка выглядит так:

```
java com.asw.rmi.ex1.BillingServiceImpl
```

В результате отображается сообщение об инициализации BillingService.

Теперь клиентское приложение может соединиться с удаленным объектом, выполняющимся на локальной машине localhost. Команда

```
java com.asw.rmi.ex1.BillingClient
```

соединит BillingClient с объектом BillingServiceImpl.

Если серверное приложение выполняется не на клиенте, можно указать IP-адрес или доменное имя компьютера-сервера в качестве параметра командной строки при выполнении клиента. Например, чтобы осуществить доступ к компьютеру с IP-адресом 192.168.1.1, введем команду

```
java com.asw.rmi.ex1.BillingClient 192.168.1.1
```

Вопросы для самоконтроля

1. Что такое маршалинг?
2. Для чего применяется RMI?
3. Что такое удаленный интерфейс?
4. Как объявить удаленный интерфейс?
5. Каким условиям должны удовлетворять методы удаленного интерфейса?
6. Как определить реализацию удаленного интерфейса?
7. Как зарегистрировать удаленный объект?
8. Как найти удаленный объект?

1.12 Технология CORBA

1.12.1 Обзор архитектуры. Концепции CORBA

CORBA (Common Object Request Broker Architecture – общая архитектура брокера объектных запросов) – это технологический стандарт написания распределённых приложений, продвигаемый консорциумом OMG (Object Management Group – группа управления объектами) и соответствующая ему информационная технология.

Технология CORBA является механизмом для интеграции изолированных информационных систем, который даёт возможность программам, написанным на разных языках программирования и работающим на разных узлах сети, взаимодействовать друг с другом, так как если бы они находились в адресном пространстве одного процесса.

Основой технологии CORBA является ORB (Object Request Broker – брокер объектных запросов) – «объектная шина», которая позволяет передавать сообщения от одного объекта к другому. Она может быть реализована в виде библиотеки, специального сетевого сервиса, объектно-ориентированной СУБД или уже быть включенной в операционную систему. ORB дает возможность объектам, которых он обслуживает, не задумываться о том, где находятся объекты, которым передаются сообщения. Кроме того, он скрывает все детали реализации объектов, оставляя видимыми только их интерфейсы.

Помимо удалённых объектов в CORBA 3.0 определено понятие объект по значению. Код методов таких объектов по умолчанию выполняется локально. Если объект по значению был получен с удалённой стороны, то необходимый код должен либо быть заранее известен обеим сторонам, либо быть динамически загружен. Чтобы это было возможно, запись, определяющая такой объект, содержит поле Code Base – список URL, откуда может быть загружен код.

У объекта по значению могут также быть и удалённые методы, поля, которые передаются вместе с самим объектом. Поля, в свою очередь также могут быть такими объектами, формируя таким образом списки, деревья

или произвольные графы. Объекты по значению могут иметь иерархию классов, включая абстрактные и множественное наследование.

Компонентная модель CORBA (CCM) – дополнение к семейству определений CORBA.

CCM была введена, начиная с CORBA 3.0, и описывает стандартный каркас приложения для компонент CORBA. CCM построена под сильным влиянием Enterprise JavaBeans (EJB) и фактически является его независимым от языка расширением. CCM предоставляет абстракцию сущностей, которые могут предоставлять и получать сервисы через чётко определённые именованные интерфейсы порты.

Модель CCM предоставляет контейнер компонентов, в котором могут поставляться программные компоненты. Контейнер предоставляет набор служб, которые может использовать компонент. Эти службы включают (но не ограничены) службу уведомления, авторизации, персистентности (способности объекта существовать дольше процесса, создавшего его) и управления транзакциями. Это наиболее часто используемые распределённым приложением службы. Перенося реализацию этих сервисов от необходимости реализации самим приложением в функциональность контейнера приложения, можно значительно снизить сложность реализации собственно компонентов.

Object Management Architecture (OMA) – это эталонная архитектура распределенных систем, основанная на концепции брокера объектных запросов. Используя концепции объектно-ориентированных технологий, OMA определяет к использованию рабочее пространство, где объекты, по определению являющиеся общедоступными, открыты для использования любым другим объектом или сервисом посредством объектного брокера. Объектный брокер – это прозрачный коммуникационный механизм, обеспечивающий надежный обмен сообщениями между объектами независимо от их местоположения. OMA определяет абстракцию, которая скрывает тот факт, что различные системы применяют разные языки программирования или несовместимые версии одного и того же языка.

CORBA определяет правила функционирования брокера объектных запросов в условиях использования разных языков программирования. OMA определяет полиморфное рабочее пространство общих сервисов, которое выглядит однородным извне (со стороны API), но может быть разнородным внутри.

Брокеры объектных запросов могут быть реализованы одним из двух способов: в виде библиотек или как процессы-демоны. Ни для клиента, ни для серванта средства реализации не имеют значения. Конструкция объекта брокера объектных запросов скрывает лежащую в ее основе реализацию. Обычно клиент использует брокер объектных запросов на основе библиотеки, а сервер использует ORB в виде процесса-демона. Это деталь

реализации, решение о которой принимают администраторы систем. С позиции активного процесса ничего не меняется.

Брокер объектных запросов играет в OMA основную роль. Теперь мы рассмотрим, как клиент и сервант воспринимают брокер объектных запросов. Клиент взаимодействует с брокером объектных запросов одним из трех способов: посредством статической заглушки (создаваемой IDL-компилятором), динамического интерфейса (применяя API динамических вызовов CORBA) или API брокера объектных запросов. Концептуально брокер объектных запросов взаимодействует с сервантом тремя способами: посредством статического скелета, динамического интерфейса или объектного адаптера серванта (что выглядит так, как если бы брокер объектных запросов взаимодействовал с сервантом напрямую). Сервант, обращающийся к другому серванту, становится клиентом и функционирует как клиент.

Наиболее прямолинейным способом взаимодействия с брокером объектных запросов является использование статических заглушек и скелетов. Они содержат код, необходимый для связи, и предоставляют возможность статического контроля типов на основе их IDL-описания. Динамические вызовы (и со стороны клиента, и со стороны серванта) требуют больших издержек, но являются более гибкими, так как позволяют разработчикам программно управлять вызовами удаленных объектов.

1.12.2 Язык IDL

Язык IDL определяет типы объектов путем спецификации их интерфейсов. Интерфейс состоит из списка операций и их параметров. Несмотря на то, что IDL предоставляет каркас для описания объектов, которыми манипулирует ORB, нет необходимости в том, что брокер имел доступ к исходному коду на IDL. Брокер может работать с эквивалентной информацией в виде заглушек подпрограмм и репозитория интерфейсов.

IDL является средством, с помощью которого реализация объекта сообщает своим потенциальным клиентам, какие операции доступны и как они могут быть вызваны. Из IDL-описания CORBA-объект можно перевести на определенный язык программирования или в другую объектную систему.

Семантически законченное исходное описание на языке IDL, пригодное для конкретного применения, называется OMG IDL спецификацией. Она представляют собой совокупность определений модулей, интерфейсов, констант, типов и исключительных ситуаций. Рассмотрим кратко каждую из этих языковых конструкций.

Модули Синтаксическое определение модуля имеет вид:

<модуль>::= "module" <идентификатор> "{" <определение>+ "}"

Здесь и далее определение понимается как определение модуля, интерфейса, исключительной ситуации, константы или типа.

Модули используются для того, чтобы избежать коллизий между именами, определенными в спецификации OMG IDL, и именами в языках

программирования и других программных системах, совместно с которыми эти спецификации применяются.

С помощью модулей устанавливается область определения имен в спецификациях для последующего управления разрешением видимости. При ссылках в спецификациях или вне их на имена, определенные в некотором IDL-модуле, следует квалифицировать эти имена именем модуля, задавая его как префикс с разделителем – символом операции разрешения видимости («::»). Например, ссылка CORBA::Trans означает языковый объект с именем Trans, определенный в модуле CORBA. Заметим, что в OMG IDL сохраняются и обычные для C++ возможности ссылки на глобальные переменные в области действия одноименной локальной переменной с помощью операции «::».

Интерфейсы Интерфейс – главный объект языка. Синтаксически он определяется следующим образом:

<интерфейс>::=<заголовок_интерфейса> "{" <тело_интерфейса> "}"

Заголовок интерфейса состоит из ключевого слова «interface», за которым следуют имя интерфейса и необязательная спецификация наследования.

Например, спецификация:

interface optim_model:link_model;

представляет собой заголовок интерфейса optim_model, который является производным от базового интерфейса link_model. Синтаксис языка располагает средствами, позволяющими описывать в заголовке интерфейса сложные структуры наследования. Имена базовых интерфейсов при этом могут квалифицироваться символом «::» или префиксом – именем блока с последующим символом операции «::».

Тело интерфейса может включать произвольное число объявлений типов, констант, исключительных ситуаций, атрибутов и операций. Допускаются, в частности, и пустые интерфейсы, тело которых не содержит никаких объявлений. Все указанные виды объявлений будут рассмотрены далее.

Заметим, что, в отличие от объявления интерфейса, непосредственно описывающего его свойства, предусматривается также возможность упреждающего объявления. При этом объявляется имя интерфейса, но не специфицируется его определение. Предполагается, что определение последует позднее в данной OMG IDL-спецификации. Упреждающие объявления позволяют, в частности, задавать определения интерфейсов, которые ссылаются друг на друга.

Объявления типизированных констант, то есть таких переменных, значения которых не могут изменяться, полностью аналогичны с точностью до возможных типов таким декларациям в C++:

<константа> ::= "const" <тип> <идентификатор> = <выражение>

Здесь тип определяет тип объявляемой константы, идентификатор указывает ее имя, а значение константного выражения рассматривается

как значение константы. Это значение может быть числовым, строковым, символьным, булевским или представляет собой какое-либо имя.

В языке OMG IDL предусмотрены синтаксические средства для объявления базовых, конструируемых и шаблонных типов данных объектной модели OMG.

Базовые типы обозначаются ключевыми словами: «any», «boolean», «char», «double», «float», «long», «octet», «short», «unsigned short», «unsigned long». Синтаксис объявлений таких типов приведен ниже и не требует дополнительных пояснений:

<спецификация_базового_типа>::= <ключевое_слово_базового_типа> <декларатор> {","<декларатор>}*

где:

<декларатор>::= <идентификатор> | <идентификатор> "[" <положительная_ целая_константа> "]"

Второй вариант декларатора в приведенной спецификации представляет собой массив заданной длины.

Примеры объявлений базовых типов:

float input_stream, output_stream;

short green_array [25], blue_vari;

Ключевые слова конструируемых типов – «struct», «union», «enum».

Тип «struct» представляет собой совокупность других заданных типов, рассматриваемую как единое целое. Его синтаксическая спецификация имеет вид:

<спецификация_типа_struct>::= "struct"
  <идентификатор> "{" <список_членов> "}"
<список_членов>::= <член>+
<член>::= <спецификация_типа>
  <декларатор> {","<декларатор>}*

Спецификация типа означает здесь и далее ключевое слово любого базового или конструируемого типа.

Пример спецификации типа struct:

```
struct model {
   string title;
   char modtype;
   string institute;
   short dimension;
   boolean implementation;
};
```

Размеченное объединение («union») – это такое объединение других типов, в заголовке которого указывается тип величины, определяющей при конкретном вызове, какой именно член объединения должен быть использован. Тип размеченного объединения специфицируется следующим образом:

<спецификация_типа_union>::= "union" <идентификатор> "switch" "("<тип_ переключателя>")"

"{" <тело_переключателя> "}"

<тип_переключателя>::= short | long | unsigned short | unsigned long | char | boolean | enum

<тело_переключателя>::= <вариант>+

<вариант> ::= <метка_варианта>+ <спецификация_элемента> ";"

<метка_варианта>::= "case" <константное_выражение> ":"

| "default" ":"

<спецификация_элемента>::= <спецификация_типа> <декларатор>

Пример спецификации типа размеченного объединения:

```
union variant switch(char){
   1: char symbol;
   2: float min_price;
   3: struct new_occurence{
   long x; float y; char z;
   };
   default: string line
};
```

Перечислимый тип («enum») по традиции определяет упорядоченное множество идентификаторов и специфицируется следующим образом:

<спецификация_типа_enum>::= "enum" <идентификатор> "{"<идентификатор> {"," <идентификатор>}* "}"

Пример такой спецификации:

enum color {black, white, blue, red, yellow};

Наряду с базовыми и конструируемыми типами, в OMG IDL используются шаблонные типы. К числу таких типов, как уже отмечалось, относятся типы «sequence» (последовательность) и «string» (строка). Эти типы имеют вспомогательный характер и введены из соображений удобства. Они используются, главным образом, в объявлениях переименования типов «typedef».

Последовательность понимается как одномерный массив, обладающий максимальным размером, фиксируемым на стадии компиляции, и длиной, которая определяется в период исполнения. Спецификация типа «sequence» имеет вид:

<спецификация_типа_sequence> ::= "sequence" "<" {<спецификация_базового_типа> | <спецификация_шаблонного_типа>}

["," <положительная_целая_константа>] ">" <идентификатор>

Положительная целая константа, если она задана, указывает максимальный размер последовательности, которая в таком случае называется ограниченной. В противном случае она называется неограниченной.

Примеры спецификаций последовательностей:

typedef sequence bound_sequence;

typedef sequence .unbound_sequence;

Строка трактуется как последовательность любых символов алфавита, за исключением символа «\0». Для строки может быть факультативно задан ее максимальный размер – положительное целое. Если он задан, строка называется ограниченной. В противном случае она называется неограниченной. Спецификация типа «string» выглядит таким образом:

<спецификация_типа_string> ::= "string" ["<" <положительная_целая_константа> ">"] <идентификатор>

Примеры спецификаций строк:

typedef string <25> bound_string;

typedef string unbound_string;

Исключительные ситуации В языке OMG IDL предусмотрены возможности для объявления исключительных ситуаций, которые позволяют установить, какая исключительная ситуация имела место при выполнении заявки. Спецификация исключительной ситуации включает конструкцию, похожую на структуру («struct»), которая описывает возвращаемое значение, и она имеет вид:

<спецификация_исключительной_ситуации> ::= "exception" <идентификатор> "{" <член>* "}"

Здесь синтаксический терм член имеет тот же смысл, что и в определении типа («struct»). Терм идентификатор – это идентификатор данного типа исключительной ситуации. Если при выполнении заявки возникает исключительная ситуация, то значение этого идентификатора доступно для анализа с тем, чтобы установить, какая именно исключительная ситуация имеет место.

В стандарте CORBA 2.0 определен некоторый перечень стандартных исключительных ситуаций, которые могут возникать при выполнении заявок.

Для обеспечения полноты рассмотрения спецификаций интерфейсов рассмотрим теперь, каким образом специфицируются операции в языке OMG IDL.

Приведем синтаксис объявления операций:

<объявление_операции>::=

[<атрибут_операции>] <тип_результата> <идентификатор> <объявления_параметров>

[<исключительные_ситуации>] [<контекстное_выражение>]

1.12.3 Брокер объектных запросов (Object Request Broker)

Borland Enterprise Server, VisiBroker Ed – CORBA 2.6-совместимый коммерческий ORB от Borland, поддерживает Java и C++.

MICO – свободный (LGPL – GNU Lesser General Public License – Стандартная общественная лицензия ограниченного применения GNU) ORB с поддержкой C++.

omniORB – свободный (LGPL) ORB для C++ и Python.

ORBit2 – свободный (LGPL) ORB для C, C++ и Python.

JacORB – свободный (LGPL) ORB, написан на Java.

TAO – The ACE (англ. Adaptive Communication Enviroment – адаптивная коммуникационная среда) ORB, открытый ORB для C++.

Orbacus – коммерческий ORB для C++, Java от IONA Technologies.

Orbix – коммерческий ORB от IONA Technologies.

PolyORB – ORB от AdaCore для языка программирования Ada. Есть как свободная, так и коммерческая версии.

1.12.4 Протокол обмена сообщениями между объектными брокерами (IIOP)

Когда клиент запрашивает операцию, реализуемую распределенным объектом, брокер объектных запросов данного клиента для выполнения вызова использует объектную ссылку. Объектная ссылка, используемая клиентом (поддерживаемая открытым API вызываемого объекта, описанным в IDL-файле), содержит непрозрачную сетевую ссылку (непрозрачную – так как ни клиент, ни сервер не могут извлечь из этой ссылки информацию о деталях реализации). Клиент запрашивает операцию, реализуемую распределенным объектом посредством объектной ссылки, поскольку Interoperable Object Reference (IOR) ссылается именно на объектную ссылку, структура которой хорошо известна брокерам ORB при условии использования OMG-совместимых протоколов (например, IIOP). Источником объектной ссылки является объектный адаптер, указывающий на удаленный объект. Объектная ссылка содержит три важных фрагмента информации: местоположение распределенного объекта; ссылку на адаптер, создавший объектную ссылку; идентификатор объекта серванта. Все CORBA-совместимые объектные брокеры «знают», что такое объектные ссылки и как их использовать для установления соединений клиентов с сервантами. Последнее, что необходимо для обеспечения функциональной совместимости, – это создание и синтаксический анализ IOR в протоколе для отправки запроса через сеть. Чтобы сделать возможным надежное и уверенное взаимодействие брокеров объектных запросов различных производителей, консорциум OMG разработал спецификацию протокола Internet Inter ORB Protocol (IIOP). IIOP – это протокол обмена сообщениями между объектными брокерами.

Все производители должны обеспечивать поддержку IIOP, чтобы их объектные брокеры были CORBA-совместимыми. Производители могут также поддерживать свои собственные протоколы, но, как минимум, они должны поддерживать IIOP. Объектные брокеры, которые предполагают обмениваться между собой сообщениями, должны «говорить» на языке одного и того же протокола. В том случае, если объектные брокеры

поддерживают разные протоколы, должны быть разработаны специальные адаптеры для преобразования протоколов, чтобы обеспечить обмен сообщениями между объектными брокерами, поддерживающими нестандартные протоколы. При установке брокера объектных запросов IIOP должен устанавливаться по умолчанию в качестве стандартного протокола, хотя IIOP может являться не единственным протоколом.

IIOP является реализацией другого стандарта OMG – General Inter ORB Protocol (GIOP). GIOP определяет сообщения, необходимые объектным брокерам для обмена данными между собой, и обеспечивает поддержку лежащего в основе этого взаимодействия транспорта для платформы, на которой функционирует данный брокер объектных запросов. TCP/IP является предпочтительным транспортным средством, хотя спецификация включает также поддержку Novell IPX и OSI. Производители могут обеспечивать поддержку и других транспортных механизмов. Java, с ее встроенной сетевой поддержкой, и CORBA, с ее механизмами обеспечения соединений распределенных объектов, являются взаимодополняющими технологиями.

Для каждого языка программирования, поддерживаемого OMG, должно существовать преобразование из IDL на этот язык. Отображение IDL–Java определяет, как следует преобразовывать ключевые слова и типы данных IDL в конструкции Java. Java 2 был первой версией, выпущенной корпорацией Sun, непосредственно поддерживающей OMG-отображение.

1.12.5 Сервисы CORBA (CORBA services)

Архитектурой CORBA определены различные службы (сервисы) , расширяющие круг основных возможностей брокеров ORB. Некоторые из этих служб функционируют в качестве серверов поверх ORB, другие же следует встраивать в ORB, по крайней мере частично.

В области связующих технологий наиболее важными (можно сказать, основополагающими) для работы в среде ORB являются 3 сервиса:

Сервис именования (Naming Service) ставит в соответствие объекту символическое имя и позволяет клиенту получить ссылку на этот объект, организовав поиск по его имени. Этот сервис является аналогом телефонного справочника для объектов. Имя объекта составное(compound) и является цепочкой контекстных (context) имен (аналога папок или директорий в файловой системе) и простого (simple) имени (аналога названия файла) с необязательным указанием типа объекта (аналога типа файла).

Сервис жизненного цикла (Life Cycle Service) управляет созданием, перемещением (копированием) и уничтожением объекта.

Сервис событий (Event Service) обеспечивает общение клиентов и запрошенных объектов (асинхронные взаимодействия между анонимными объектами). Клиент отправляет событие (сообщение) в канал событий, и оно

передается на все серверы, зарегистрированные в данном канале. Этот механизм используется для организации многоадресной связи («один со многими»).

Упомянем только еще пару очень важных служб:

Сервис защиты (Security Service) поддерживает аутентификацию и проверку полномочий, предоставляя приложениям возможность ограничивать круг клиентов, которым разрешены те или иные операции; кроме того, поддерживает шифрование данных.

Сервис транзакций (Transaction Service) обеспечивает атомарную фиксацию или отмену всех изменений, производимых приложениями, которые работают с несколькими серверами и обновляют несколько баз данных. Данная служба (нередко обозначаемая английским сокращением OTS) определяет простые интерфейсы, необходимые в узлах-клиентах для создания и прекращения транзакций. Остальные ее интерфейсы являются внутренними и используются для абстрагирования протокола двухфазной фиксации и других целей.

Сервисы CORBA намного облегчают жизнь разработчикам приложений, потому как решают множество типичных задач, с которыми сталкиваются разработчики при создании среды взаимодействия между компонентами приложений.

Вопросы для самоконтроля

1. Что представляет собой CORBA?
2. На каком языке описываются интерфейсы при использовании CORBA?
3. За что отвечает брокер объектных запросов?
4. По какому протоколу происходит обмен данных между брокерами объектных запросов?
5. Какие сервисы может предоставлять брокер объектных запросов?

1.13 Параллелизм в распределенных приложениях

1.13.1 Основные проблемы параллелизма

Термин «параллелизм» означает возможность одновременного выполнения многих процессов с доступом к одним и тем же данным, причем в одно и то же время. Как известно, в такой системе для корректной обработки данных параллельными процессами без возникновения конфликтных ситуаций необходимо использовать некоторый метод управления параллелизмом.

Стандартный метод разрешения таких проблем – метод блокировки. Блокировка не является единственным возможным подходом в решении проблемы управления параллелизмом, но она, несомненно, чаще других встречается на практике. Однако с внедрением метода блокировки возникают другие проблемы, среди них наиболее известна проблема тупиковых

ситуаций. Формальный критерий правильности выполнения некоторого набора параллельных транзакций описан в концепции способности к упорядочению.

Следует отметить, что параллелизм является весьма обширной темой, и поэтому в данном разделе будут описаны лишь некоторые важные идеи.

Каждый метод управления параллелизмом предназначен для решения некоторой конкретной задачи. Тем не менее, при обработке правильно составленных транзакций возникают ситуации, которые могут привести к получению неправильного результата из-за взаимных помех среди некоторых транзакций. Основные проблемы, возникающие при параллельной обработке транзакций, следующие:

– проблема потери результатов обновления;

– проблема незафиксированной зависимости;

– проблема несовместимого анализа.

Рассмотрим следующую ситуацию. Первая транзакция извлекает значение баланса, изменяет его и затем фиксирует. Проблема состоит в том, что вторая транзакция, которая также изменяет состояние баланса, делает это относительно «старого», еще не измененного первой транзакцией значения. Таким образом, при выполнении двух транзакций в указанном порядке изменения, выполненные первой транзакцией, будут просто утеряны.

Проблема незафиксированной зависимости появляется, если в некоторой транзакции осуществляется извлечение некоторых данных, которые в данный момент обновляются другой транзакцией, но это обновление еще не завершено. Таким образом, поскольку обновление еще не завершено, есть некоторая вероятность, что оно не будет завершено никогда (может быть выполнен откат транзакции, например). В таком случае в первой транзакции будут принимать участие данные, которых больше не существует.

В том случае, когда изменения, сделанные второй транзакцией, сразу становятся видны всем остальным (режим «грязного чтения» – см. ниже), первая транзакция читает «измененные» данные. Проведя вычисления, первая транзакция фиксирует результат. Вторая же транзакция производит откат изменений, то есть фактически значения баланса, прочитанного первой транзакцией, никогда не существовало! Стоит отметить, что, несмотря на то, что вторая транзакция была отменена, результаты первой транзакции были уже зафиксированы и отменены не будут.

Проблема несовместимого анализа возникает в том случае, когда одна из транзакций в силу длительности выполнения застает часть данных в одном состоянии (неизмененными), а часть – в другом (уже измененными).

Представим, например, «банковское» приложение, которое подсчитывает сумму средств на счетах всех клиентов банка.

Процедура подсчета идет «сверху вниз» по таблице счетов и суммирует их. Допустим, процедура была запущена в момент времени t_1. Далее, в момент времени между t_1 и t_2 была запущена транзакция, которая состояла

из двух действий – снять 100 рублей со счета Иванова и перечислить их на счет Яковлева (к этому моменту процедура суммирования уже учла сумму на счете Иванова, но до Яковлева еще не дошла). Транзакция была очень короткой. После того, как транзакция была успешно завершена, ее результаты стали «видны» первой процедуре. Таким образом, когда она дойдет до записи со счетом Яковлева, ей будет учтена сумма в 500 рублей. В результате сумма счетов в «банке» будет на 100 рублей завышена.

1.13.2 Транзакции, свойства, ресурсы транзакций

Транзакции – основной инструмент управления параллельными процессами в распределенных корпоративных приложениях. Слово «транзакция» часто приходит на ум в связи с коммерческими и банковскими операциями. Обращение к банкомату, ввод пароля и получение наличности – это транзакция.

Эти примеры характеризуют природу типичной транзакции. Во-первых, транзакция представляет собой ограниченную последовательность действий с явно определенными начальной и завершающей операциями. Во-вторых, все ресурсы, затрагиваемые транзакцией, пребывают в согласованном состоянии в момент ее начала и остаются в таковом после ее завершения. Помимо того, транзакция должна либо выполняться целиком, либо не выполняться вовсе.

Транзакции в программных системах часто описывают в терминах свойств, обозначаемых общей аббревиатурой ACID.

Atomicity (атомарность). В контексте транзакции либо выполняются все действия, либо не выполняется ни одно из них. Частичное или избирательное выполнение недопустимо. Например, если клиент банка переводит сумму с одного счета на другой и в момент между завершением расходной операции и началом приходной операции сервер терпит крах, система должна вести себя так, будто расходной операции не было вовсе. Система должна либо осуществить обе операции, либо не выполнить ни одной. Фиксация (commit) результатов служит свидетельством успешного окончания транзакции; откат (rollback) приводит систему в состояние, в котором она пребывала до начала транзакции.

Consistency (согласованность). Системные ресурсы должны пребывать в целостном и непротиворечивом состоянии как до начала транзакции, так и после ее окончания.

Isolation (изолированность). Промежуточные результаты транзакции должны быть закрыты для доступа со стороны любой другой действующей транзакции до момента их фиксации. Иными словами, транзакция протекает так, будто в тот же период времени других параллельных транзакций не существует.

Durability (устойчивость). Результат выполнения транзакции не должен быть утрачен ни при каких условиях.

Наиболее часто термин «транзакция» произносится применительно к СУБД. Однако это более широкое понятие, которое используется во многих других областях. Существует множество других объектов, управляемых с помощью механизмов поддержки транзакций, например, очереди сообщений или заданий на печать, банкоматы и так далее. Таким образом, термин «ресурсы транзакций» служит для обозначения всего, что может быть затребовано параллельно протекающими процессами, определяемыми с помощью модели транзакций. Наиболее распространенным ресурсом транзакций являются базы данных. Поэтому для наглядности и краткости упоминаются именно они, хотя все сказанное применимо и к другим видам ресурсов.

Для обеспечения высокого уровня пропускной способности современные системы управления транзакциями проектируются в расчете на максимально короткие транзакции. Обычно из практики исключаются транзакции, которые охватывают действия по обработке нескольких запросов; если же подобной ситуации избежать не удается, решения реализуются на основе схемы длинных транзакций. Чаще, однако, границы транзакции совпадают с моментами начала и завершения обработки одного запроса. Подобная транзакция запроса – весьма удачная модель, и многие среды поддерживают простой и естественный синтаксис ее описания.

1.13.3 Блокировки

Описанные проблемы параллелизма могут быть разрешены с помощью методики управления параллельным выполнением процессов под названием «блокировка» .Ее основная идея очень проста: в случае, когда для выполнения некоторой транзакции необходимо, чтобы некоторый объект не изменялся непредсказуемо и без ведома этой транзакции (как это обычно бывает), такой объект блокируется. Таким образом, эффект блокировки состоит в том, чтобы заблокировать доступ к этому объекту со стороны других транзакций, а значит, предотвратить непредсказуемое изменение этого объекта. Следовательно, первая транзакция в состоянии выполнить всю необходимую обработку с учетом того, что обрабатываемый объект остается в стабильном состоянии настолько долго, насколько ей это нужно, и лишь затем снять блокировку.

Опишем функционирование блокировки более подробно.

1. Прежде всего предположим, что в системе поддерживается два типа блокировок: блокировка без взаимного доступа (монопольная блокировка), называемая X-блокировкой (X locks – eXclusive locks), и блокировка с взаимным доступом, называемая S-блокировкой (S locks – Shared locks). (Здесь предполагается, что X- и S-блокировки – единственно

возможные, хотя в реальных системах управления блокировками встречаются блокировки других типов).

2. Если транзакция A блокирует объект p без возможности взаимного доступа (X-блокировка), то запрос другой транзакции B с блокировкой этого объекта p будет отменен.

3. Если транзакция A блокирует объект p с возможностью взаимного доступа (S-блокировка), то

– запрос со стороны некоторой транзакции B на X-блокировку объекта будет отвергнут;

– запрос со стороны некоторой транзакции B на S-блокировку объекта p будет принят (то есть транзакция B также будет блокировать объект p с помощью S-блокировки).

Теперь следует ввести протокол доступа к данным, который на основе введения только что описанных X- и S-блокировок позволяет избежать возникновения проблем параллелизма:

1. транзакция, предназначенная для чтения состояния объекта, прежде всего должна наложить S-блокировку на этот кортеж;

2. транзакция, предназначенная для изменения состояния объекта, прежде всего должна наложить X-блокировку на этот объект. Если для последовательности действий типа «извлечение/обновление» для объекта уже задана S-блокировка, то ее необходимо заменить X-блокировкой;

3. если запрашиваемая блокировка со стороны транзакции B отвергается из-за конфликта с некоторой другой блокировкой со стороны транзакции A, то транзакция B переходит в состояние ожидания. Причем транзакция B будет находиться в состоянии ожидания до тех пор, пока не будет снята блокировка, заданная транзакцией A;

4. X-блокировки сохраняются вплоть до конца выполнения транзакции. S-блокировки также обычно сохраняются вплоть до этого момента.

1.13.4 Восстановление после сбоев

Важнейшая задача транзакционной обработки – атомарность – предполагает возможность отката сделанных изменений в случае если какое то из действий, составляющих транзакцию, закончилось неудачей. Другой важной характеристикой транзакции является долговечность, т.е. сохранение результатов транзакции после ее фиксации. Часто выполнение этих двух (вообще говоря, различных) задач объединяют в одном компоненте, называемом менеджером восстановления. Рассмотрим, однако, сначала первую задачу, а именно откат незавершенной транзакции. Первый из способов реализации этого механизма состоит в том, что изменения, которые делает транзакция в процессе работы, записываются не в долговременную память, а во временную. При этом в случае фиксации транзакции

происходит перенос данных из временной памяти в основную, а в случае отката временная память просто очищается.

Одна из проблем здесь состоит в том, что в случае фиксации транзакции требуется немедленно перенести данные в долговременную память (в основном это жесткий диск), что в случае частых коротких транзакций будет плохо отражаться на производительности системы.

Другой проблемой является то, что если произошла необнаруженная (или обнаруженная) ошибка диска, данные завершенной транзакции могут быть потеряны. Поэтому часто применяют другой метод (иногда – совместно с первым), который состоит в журналировании операций, выполняемых транзакцией. При таком подходе все изменения, которые производит транзакция, вместо того чтобы записываться в долговременную память, записываются в журнал операций. В случае отката транзакций часть журнала, относящегося к прерванной транзакции, может быть просто уничтожена (часто в этом случае в журнал операций записывается специальная операция отката транзакции), а в случае фиксации транзакции операции из журнала должны быть «проиграны» на основной долговременной памяти.

Первое преимущество здесь состоит в том, что после фиксации транзакции не нужно сразу же переносить данные в долговременную память – это может делаться для нескольких завершенных транзакций сразу (причем операции в журнале при этом могут быть переупорядочены для увеличения быстродействия). Появляется еще одна полезная возможность – в случае повреждения по той или иной причине части данных в основном долговременном хранилище они могут быть восстановлены по журналу операций, вот почему говорилось о связи восстановления после сбоев и отката транзакции. Существенным минусом такого подхода является то, что появляется время, в течение которого данные, измененные зафиксированными транзакциями в основном долговременном хранилище, оказываются «старыми», неизмененными. Эта проблема может быть решена, например, блокировкой всех измененных, но не сброшенных из журнала операций данных.

Вопросы для самоконтроля

1. В чем заключается проблема потери результатов обновления?
2. В чем заключается проблема незафиксированной зависимости?
3. В чем заключается проблема несовместимого анализа?
4. Что такое транзакция?
5. Какими свойствами может обладать механизм транзакций?
6. Какие бывают блокировки?
7. Что такое атомарность?

1.14 Проектирование Web-сервисов

1.14.1 Простой протокол доступа к объектам (SOAP)

Web-сервисы – новое слово в технологии распределенных систем. Спецификация Open Net Environment (ONE) корпорации Sun Microsystems и инициатива . Net корпорации Microsoft обеспечивают инфраструктуры для написания и развертывания Web-сервисов. В настоящий момент имеется несколько определений Web-сервиса. Web-сервисом может быть любое приложение, имеющее доступ к Web, например, Web-страница с динамическим содержимым. В более узком смысле Web-сервис – это приложение, которое предоставляет открытый интерфейс, пригодный для использования другими приложениями в Web. Спецификация ONE Sun требует, чтобы Web-сервисы были доступны через HTTP и другие Web-протоколы, чтобы дать возможность обмениваться информацией посредством XML-сообщений и чтобы их можно было найти через специальные сервисы – сервисы поиска. Для доступа к Web-сервисам разработан специальный протокол – Simple Object Access Protocol (SOAP),который представляет средства взаимодействия на базе XML для многих Web-сервисов. Web-сервисы особенно привлекательны тем, что могут обеспечить высокую степень совместимости между различными системами.

Гипотетический Web-сервис, разработанный в соответствии с архитектурой ONE Sun, может принимать форму, в которой реестр сервисов публикует описание Web-сервиса в виде документа Universal Description, Discovery and Integration (UDDI).

Огромный потенциал Web-сервисов определяется не технологией, примененной для их создания. HTTP, XML и другие протоколы, используемые Web-сервисами, не новы. Функциональная совместимость и масштабируемость Web-сервисов подразумевает, что разработчики могут быстро создавать большие приложения и более крупные Web-сервисы из меньших Web-сервисов. Спецификация Sun Open Net Environment описывает архитектуру для создания интеллектуальных Web-сервисов. Интеллектуальные Web-сервисы задействуют общее операционное окружение. Совместно используя контекст, интеллектуальные Web-сервисы могут выполнять стандартную аутентификацию для финансовых транзакций, предоставлять рекомендации и указания в зависимости от географического местоположения компаний, участвующих в электронном бизнесе.

Для того чтобы создать приложение, являющееся Web-сервисом, необходимо применить целый ряд технологий.

По сути, Web-сервисы являются одним из вариантов реализации компонентной архитектуры, при которой приложение рассматривается как совокупность компонентов, взаимодействующих друг с другом. Как уже неоднократно говорилось, взаимодействие компонент, выполняющихся на разных платформах, представляет собой достаточно сложную задачу,

в частности, требует разработки коммуникационного протокола, учитывающего особенности передачи данных между различными платформами. Одной из основных идей, положенных в основу рассматриваемой технологии Web-сервисов, является отказ от бинарного коммуникационного протокола. Обмен сообщениями между компонентами системы осуществляется посредством передачи XML-сообщений. Поскольку XML-сообщения представляют собой текстовые файлы, транспортный протокол передачи может быть самый различный – XML-сообщения можно передавать по HTTP-, SMTP-, FTP-протоколам, причем использование различных транспортных протоколов прозрачно для приложений. Как уже говорилось, протокол, обеспечивающий возможность взаимодействия Web-сервисов, называется SOAP (Simple Object Access Protocol). Он определен на основе XML. SOAP обеспечивает взаимодействие распределенных систем, независимо от объектной модели или используемой платформы. Данные в рамках SOAP передаются в виде XML-документов особого формата. SOAP не навязывает какого-либо определенного транспортного протокола. Однако в реальных приложениях наиболее часто реализуется передача SOAP-сообщений по протоколу HTTP. Также широко распространено использование в качестве транспортного протокола SMTP, FTP и даже «чистого» TCP. Итак, SOAP определяет механизм, с помощью которого Web-сервисы могут вызывать функции друг друга. В каком-то смысле работа этого протокола напоминает вызов удаленной процедуры – вызывающая сторона знает имя Web-сервиса, имя его метода, параметры, которые метод принимает, оформляет вызов этого метода в виде SOAP-сообщения и отсылает его Web-сервису.

Базовым протоколом, обеспечивающим взаимодействие в среде Web-сервисов, является протокол SOAP.

Протокол SOAP разработали корпорации IBM, Lotus Development Corporation, Microsoft, Develop-Mentor и Userland Software. Этот протокол основан на HTTP-XML. Он позволяет приложениям взаимодействовать между собой через Internet, используя для этого XML-документы, называемые сообщениями SOAP. Протокол SOAP совместим с любой объектной моделью, поскольку он включает только те функции и методы, которые абсолютно необходимы для формирования коммуникационной инфраструктуры. Таким образом, SOAP является независимым от платформы и конкретных приложений, а для его реализации может применяться любой язык программирования. SOAP поддерживает практически любой транспортный протокол. SOAP также поддерживает любые методы кодирования данных, которые позволяют приложениям, основанным на SOAP, посылать в сообщениях SOAP информацию практически любого типа (например, изображения, объекты, документы и так далее).

Сообщение SOAP содержит конверт, который описывает содержимое, предполагаемого получателя сообщения и требования к обработке

сообщения. Необязательный элемент header (заголовок) сообщения SOAP содержит инструкции по обработке для приложений, которые принимают сообщение. Заголовок также может содержать информацию о маршрутизации. С помощью заголовка header поверх SOAP могут надстраиваться более сложные протоколы. Записи в заголовке могут модульно расширять сообщение для таких задач, как аутентификация, управление транзакциями и проведение платежей. Тело SOAP-сообщения содержит специфичные для приложения данные, предназначенные для предполагаемого получателя сообщения.

1.14.2 Язык для описания web-сервисов WSDL

Язык описания веб-служб (WSDL) является стандартной спецификацией для описания сетевых служб на базе XML. Он предоставляет для поставщиков служб простой способ описания базового формата запросов к системам вне зависимости от базовой реализации выполнения.

WSDL задает формат XML для описания сетевых служб в виде набора конечных точек, работающих с сообщениями, содержащими сведения о документе или процедуре. Операции и методы вначале описываются абстрактно и затем привязываются к конкретному сетевому протоколу и формату сообщения для определения конечной точки. Связанные конкретные конечные точки комбинируются в абстрактные конечные точки (службы). WSDL является расширяемым для поддержания описаний конечных точек и их сообщений, вне зависимости от используемого формата сообщения или сетевого протокола. Это означает, что интерфейсы задаются абстрактно с помощью схемы XML и затем привязываются к конкретным представлениям соответствующего протокола.

WSDL позволяет поставщику служб указать следующие характеристики веб-службы:

– Имя веб-службы и сведения об адресации

– Протокол и стиль кодировки, используемые при доступе к общим операциям веб-службы

– Сведения о типах, таких как операции, параметры и типы данных, составляющие интерфейс веб-службы

Документы WSDL позволяют разработчикам представить свои приложения в виде доступных через сеть служб Internet. В UDDI и WSIL документы WSDL могут быть обнаружены и другими приложениями, и привязаны к ним для выполнения транзакций или других бизнес-процессов.

На данной платформе поддерживается и рекомендуется разработка документов WSDL, соответствующих требованиям стандарта WS-I. Поставщики служб могут разворачивать объекты Javabean и EJB в виде веб-служб и создавать документы WSDL, описывающие эти службы. Они также могут создать каркасы объектов Java и EJB из существующих файлов

WSDL. Клиент бизнес-службы может создать посредника Java из документа WSDL, таким образом предоставив простой для применения интерфейс Java для службы Web. Интерфейс Java скрывает сведения о сетевом соединении от клиента, позволяя поставщику бизнес-служб сфокусироваться на частях бизнеса и процесса приложения.

Помимо средств создания веб-служб, в рабочей среде предусмотрен редактор WSDL с графическим интерфейсом и системой проверки документов WSDL на соответствие правилам языка WSDL и требованиям стандарта WS-I. В окне Структура веб-служб можно тестировать службы без создания посредников.

В документе WSDL службы описаны в качестве набора конечных точек в сети (портов). Абстрактное описание конечных точек и сообщений WSDL отделяется от экземпляров, развернутых в сети, и привязок форматов данных. Такой подход позволяет повторно использовать абстрактные описания: сообщения, которые представляют собой абстрактные описания передаваемых данных, и типы портов, которые представляют собой абстрактные наборы операций.

Спецификации протокола и формата данных для конкретного типа порта образуют привязку с возможностью повторного использования. Порт задается путем связывания сетевого адреса с многократно используемой привязкой; набор портов задает службу. Таким образом, документ WSDL, описывающий сетевые службы, содержит следующие элементы:

– Types – контейнер для определений типов данных в соответствии с выбранной системой, такой как XSD.

– Message – абстрактное определение передаваемых данных конкретного типа.

– Operation – абстрактное описание действия, поддерживаемого службой.

– portType – абстрактный набор операций, поддерживаемых одной или несколькими конечными точками.

– Binding – спецификации протокола и формата данных для конкретного типа порта. Как правило, применяется привязка SOAP с литеральным стилем кодирования и форматирования данных (стили document/literal и rpc/literal).

– Port – отдельная конечная точка, представляющая собой комбинацию привязки и сетевого адреса.

– Служба – набор связанных конечных точек.

1.14.3 RESTful Web-сервисы

RESTful Web Service это Web Service написан на основании структуры REST. REST уже широко используется и заменяет Web Service

основываясь на SOAP и WSDL. RESTful Web Service легкий (lightweigh), легко расширить и поддерживать.

Первые понятия про REST (REpresentational State Transfer) были введены в 2000 году в докторской диссертации Roy Thomas Fielding (соучредитель HTTP). В диссертации он детально знакомит с ограничениями, правилами, как и со способами выполнения в системе для получения системы REST.

REST определяет правила архитектуры для дизайна ваших Web services, фокусируется на систематических ресурсах, включая какого формата состояние ресурсов и передается по HTTP, и написан разными языками. Если посчитать по количеству использующих веб сервисов, REST стал популярным за прошедшие годы как сервис модели дизайна с преимуществом. На самом деле, REST имеет большое влияние и почти заменил SOAP и WSDL так как его намного проще и легче использовать.

REST это набор правил для создания приложения Web Service, который следует 6 основным правилам дизайна:

– модель клиент-сервер;
– не имеет состояния;
– кэширование;
– единообразие интерфейса;
– возможность опосредованного взаимодействия;
– код по требованию.

Первым ограничением, применимым к гибридной модели, является приведение архитектуры к модели клиент-сервер. Разграничение потребностей является принципом, лежащим в основе данного накладываемого ограничения. Отделение потребности интерфейса клиента от потребностей сервера, хранящего данные, повышает переносимость кода клиентского интерфейса на другие платформы, а упрощение серверной части улучшает масштабируемость. Наибольшее же влияние на всемирную паутину, пожалуй, имеет само разграничение, которое позволяет отдельным частям развиваться независимо друг от друга, поддерживая потребности в развитии интернета со стороны различных организаций.

Протокол взаимодействия между клиентом и сервером требует соблюдения следующего условия: в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится (Stateless protocol или «протокол без сохранения состояния»). Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Состояние сессии при этом сохраняется на стороне клиента. Информация о состоянии сессии может быть передана сервером какому-либо другому сервису (например, в службу базы данных) для поддержания устойчивого состояния, например, на период установления аутентификации. Клиент инициирует отправку запросов, когда он готов (возникает необходимость) перейти в новое состояние.

Во время обработки клиентских запросов считается, что клиент находится в переходном состоянии. Каждое отдельное состояние приложения представлено связями, которые могут быть задействованы при следующем обращении клиента.

Как и во Всемирной паутине, клиенты, а также промежуточные узлы, могут выполнять кэширование ответов сервера. Ответы сервера, в свою очередь, должны иметь явное или неявное обозначение как кэшируемые или некэшируемые с целью предотвращения получения клиентами устаревших или неверных данных в ответ на последующие запросы. Правильное использование кэширования способно частично или полностью устранить некоторые клиент-серверные взаимодействия, ещё больше повышая производительность и масштабируемость системы.

Наличие унифицированного интерфейса является фундаментальным требованием дизайна REST-сервисов. Унифицированные интерфейсы позволяют каждому из сервисов развиваться независимо.

К унифицированным интерфейсам предъявляются следующие четыре ограничительных условия:

– Идентификация ресурсов. Все ресурсы идентифицируются в запросах, например, с использованием URI в интернет-системах. Ресурсы концептуально отделены от представлений, которые возвращаются клиентам. Например, сервер может отсылать данные из базы данных в виде HTML, XML или JSON, ни один из которых не является типом хранения внутри сервера.

– Манипуляция ресурсами через представление. Если клиент хранит представление ресурса, включая метаданные – он обладает достаточной информацией для модификации или удаления ресурса.

– «Самоописываемые» сообщения. Каждое сообщение содержит достаточно информации, чтобы понять, каким образом его обрабатывать. К примеру, обработчик сообщения (parser), необходимый для извлечения данных, может быть указан в списке MIME-типов.

– Гипермедиа как средство изменения состояния приложения. Клиенты изменяют состояние системы только через действия, которые динамически определены в гипермедиа на сервере (к примеру, гиперссылки в гипертексте). Исключая простые точки входа в приложение, клиент не может предположить, что доступна какая-то операция над каким-то ресурсом, если не получил информацию об этом в предыдущих запросах к серверу. Не существует универсального формата для предоставления ссылок между ресурсами, Web Linking (RFC 5988, RFC 8288) и JSON Hypermedia API Language являются двумя популярными форматами предоставления ссылок в REST HYPERMEDIA сервисах.

Клиент обычно не способен точно определить, взаимодействует он напрямую с сервером или же с промежуточным узлом, в связи с иерархической структурой сетей (подразумевая, что такая структура образует слои). Применение промежуточных серверов способно повысить масштабируемость

за счёт балансировки нагрузки и распределённого кэширования. Промежуточные узлы также могут подчиняться политике безопасности с целью обеспечения конфиденциальности информации.

REST может позволить расширить функциональность клиента за счёт загрузки кода с сервера в виде апплетов или сценариев. Дополнительное ограничение позволяет проектировать архитектуру, поддерживающую желаемую функциональность в общем случае, но, возможно, за исключением некоторых контекстов.

Вопросы для самоконтроля

1. Что такое web-сервис?
2. Для чего служит SOAP?
3. Какую информацию о сервисе может содержать WSDL файл?
4. Какие требования на сервисы накладывает архитектурный стиль REST?

1.15 Использование Java Message Service (JMS)

1.15.1 Архитектура JMS. Типы сообщений

JMS (Java Messaging System) представляет собой интерфейс к внешним системам, ориентированный на работу через сообщения. JMS является «старой» технологией – первая спецификация была опубликована в 1998 г. В настоящее время пакет javax.jms входит в комплект jdk, а Sun Application Server реализует поддержку JMS в качестве одного из сервисов.

При разработке JMS в качестве основной задачи рассматривалось создание обобщенного Java API для приложений, ориентированных на работу с сообщениями (message-oriented application programming), и обеспечение независимости от конкретных реализаций соответствующих служб обработки сообщений.

Таким образом, программа, написанная с использованием JMS, будет корректно работать с любой системой сообщений, поддерживающей эту спецификацию (или имеющую соответствующие интерфейсы).

Поскольку JMS является лишь оболочкой, или интерфейсом, описывающей доступные для приложения методы, для работы приложения понадобится определенная реализация этих интерфейсов JMS, называемая провайдером JMS. Они создаются независимыми производителями, и в настоящее время таких реализаций существует достаточно много (в том числе, например, реализация, включенная в Sun Application Server и распространяемая вместе с J2EE, а также MQSeries от IBM, служба JMS WebLogic от BEA, SonicMQ от Progress и другие).

Модель обмена сообщениями (и JMS) удобно использовать в том случае, если распределенное приложение обладает следующими характеристиками:

– взаимодействие между компонентами является асинхронным;

– информация (сообщение) должна передаваться нескольким или даже всем компонентам системы (семантика передачи от одного ко многим);

– передаваемая информация используется многими внешними системами, часть из которых неизвестна на момент проектирования системы или интерфейсы которых подвержены частым изменениям (концепция ESB – Enterprise Service Bus);

– обменивающиеся информацией (сообщениями) компоненты выполняются в разное время, что требует наличия посредника для промежуточного хранения переданной информации.

Архитектура JMS выглядит следующим образом:

– прикладные программы Java, использующие JMS, называются клиентами JMS (JMS client);

– система обработки сообщений, управляющая маршрутизацией и доставкой сообщений, называется JMS-провайдером (JMS provider);

– приложение JMS (JMS application) – это прикладная система, состоящая из нескольких JMS клиентов, и, как правило, одного JMS-провайдера. JMS-клиент, посылающий сообщение, называется поставщиком (producer). JMS-клиент, принимающий сообщение, называется потребителем (consumer). Один и тот же JMS-клиент может быть одновременно и поставщиком, и потребителем в разных актах взаимодействия;

– сообщения (Messages) – это объекты, передающиеся и принимающиеся компонентами (клиентами JMS);

– средства администрирования (Administrative tools) – средства управления ресурсами, использующимися клиентами.

Сообщение в JMS – это объект Java, состоящий из двух частей: заголовка (header) и тела (body) сообщения. В заголовке находится служебная информация, тело сообщения содержит в себе пользовательские данные, которые могут быть разной формы: текстовой, сериализуемых объектов, байтовых потоков и так далее.

JMS API определяет несколько типов сообщений:

– BytesMessage предназначен для передачи потока байт, который система никак не интерпретирует;

– MapMessage предназначен для передачи множества элементов типа «имя-значение», где имена являются объектами строкового типа, а значения – объектами примитивных типов данных Java ;

– ObjectMessage предназначен для передачи сериализуемых объектов;

– StreamMessage предназначен для передачи множества элементов примитивных типов данных Java (они могут быть последовательно записаны, а затем прочитаны из тела сообщения этого типа);

– TextMessage предназначен для передачи текстовой информации.

1.15.2 Модели передачи сообщений

JMS предоставляет два подхода к передаче сообщений. Первый называется «издание-подписка» (publish an subscribe) и используется в том случае, если сообщение, отправленное одним клиентом, должно быть получено несколькими.

Второй подход называется «точка-точка» (point to point) и служит для реализации обмена сообщениями между двумя компонентами.

Спецификация JMS называет эти два подхода зонами сообщений (messaging domains).

Модель передачи сообщений «точка-точка» предоставляет возможность клиентам JMS посылать и принимать сообщения (как синхронно, так и асинхронно) через виртуальные каналы, называемые очередями (queues). Модель основывается на методе опроса, при котором сообщения явно запрашиваются (считываются) клиентом из очереди. Несмотря на то, что чтение из очереди могут осуществлять несколько клиентов, каждое сообщение будет прочитано только единожды – провайдер JMS это гарантирует.

При использовании модели взаимодействия «издание-подписка» один клиент (поставщик) может посылать сообщения многим клиентам (потребителям) через виртуальный канал, называемый темой (topic). Потребители могут выбрать подписку (subscribe) на любую тему. Все сообщения, направляемые в тему, передаются всем потребителям данной темы. Каждый потребитель принимает копию каждого сообщения. Модель передачи сообщений «издание-подписка», по существу, представляет собой модель сервера, инициирующего соединение и «проталкивающего» информацию на клиента. В JMS эта концепция реализуется с помощью специальных «слушателей» (листенеров), регистрируемых в системе. При возникновении нового события листенер, закрепленный за данной темой, возбуждается.

Следует отметить, что при использовании модели «издание-подписка» клиенты JMS могут устанавливать долговременные подписки, позволяющие потребителям отсоединиться и позже снова подключиться и получать сообщения, поступившие во время отключения связи.

Вопросы для самоконтроля

1. Какие типы сообщений поддерживает JMS?
2. Какие модели обмена сообщений возможны в JMS?

1.16 Технология Enterprise Java Bean (EJB)
1.16.1 Понятие Enterprise Java Bean

Enterprise JavaBeans – это высокоуровневая, базирующаяся на использовании компонентов технология создания распределенных приложений, которая использует низкоуровневый API для управления транзакциями. Первый вариант спецификации Enterprise JavaBeans появился в марте 1998 г., в настоящий момент актуальна версия 3.0 – технология прошла за это время большой путь и продолжает развиваться.

Enterprise JavaBeans – больше, чем просто технологическая подложка. Ее использование подразумевает еще и технологию (процесс) создания распределенного приложения – навязывает определенную архитектуру приложения, а также определяет стандартные роли для участников разработки. Действительно, анализ того, что такое создание масштабируемых и эффективных серверов приложений с использованием Java, приводит к необходимости рассмотрения следующих стандартных проблем:

– организация удаленных вызовов между объектами, работающими под управлением различных виртуальных машин Java ;

– управление потоками на стороне сервера;

– циклом жизни серверных объектов (создание, взаимодействие с пользователем, уничтожение);

– оптимизация использования ресурсов (время процессора, память, сетевые ресурсы);

– создание схемы взаимодействия контейнеров и операционных сред;

– создание схемы взаимодействия контейнеров и клиентов, включая универсальные средства создания разработки компонентов и включения их в состав контейнеров;

– создание средств администрирования и обеспечение их взаимодействия с существующими системами;

– создание универсальной системы поиска клиентом необходимых серверных компонентов;

– обеспечение универсальной схемы управления транзакциями;

– обеспечение требуемых прав доступа к серверным компонентам;

– обеспечение универсального взаимодействия с СУБД.

Технология Enterprise JavaBeans как раз определяет некоторый набор универсальных и предназначенных для многократного использования компонентов, которые называются Enterprise beans (далее – компоненты EJB).При создании распределенной системы ее бизнес-логика будет реализована в этих компонентах. Каждый компонент EJB состоит из удаленного интерфейса, собственного интерфейса и реализации EJB-компонента. Удаленный интерфейс (remote-интерфейс) определяет бизнес-методы, которые может вызывать клиент EJB. Собственный (домашний) интерфейс (home-интерфейс) предоставляет методы create для создания новых экземпляров

компонентов EJB, методы поиска (finder) для нахождения экземпляров компонентов EJB и методы remove для удаления экземпляров EJB. Реализация EJB-компонента определяет бизнес-методы, объявленные в удаленном интерфейсе, и методы создания, удаления и поиска собственного интерфейса.

После завершения их кодирования наборы компонентов EJB помещаются в специальные файлы (архивы, jar), по одному или более компонентов, вместе со специальными параметрами поставки (deployment). Затем они устанавливаются в специальной операционной среде, в которой запускается контейнер EJB. Контейнер EJB предоставляет окружение выполнения и средства управления жизненным циклом EJB-компонентам. Клиент осуществляет поиск компонентов в контейнере с помощью home-интерфейса соответствующего компонента. После того, как компонент создан и/или найден, клиент выполняет обращение к его методам с помощью remote-интерфейса.

Контейнеры EJB выполняются под управлением сервера EJB, который является связующим звеном между контейнерами и используемой операционной средой. Сервер EJB обеспечивает доступ контейнерам EJB к системным сервисам, таким как управление доступом к базам данных или мониторы транзакций, а также к другим приложениям.

Все экземпляры компонентов EJB выполняются под управлением контейнера EJB. Контейнер предоставляет системные сервисы размещенным в нем компонентам и управляет их (компонентов) жизненным циклом. В общем случае контейнер предназначен для решения следующих задач:

– обеспечение безопасности – дескриптор поставки (deployment descriptor) определяет права доступа клиентов к бизнес-методам компонентов. Обеспечение защиты данных обеспечивается за счет предоставления доступа только для авторизованных клиентов и только к разрешенным методам;

– обеспечение удаленных вызовов – контейнер берет на себя все низкоуровневые вопросы обеспечения взаимодействия и организации удаленных вызовов, полностью скрывая все детали как от разработчика компонентов, так и от клиентов, которые пишут код точно так же, как если бы система работала в локальной конфигурации, т.е. вообще без использования удаленных вызовов;

– управление циклом жизни – клиент создает и уничтожает экземпляры компонентов, однако контейнер для оптимизации ресурсов и повышения производительности системы может самостоятельно выполнять различные действия, например, активизацию и деактивацию этих компонентов, создание их пулов и т.д.;

– управление транзакциями – все параметры, необходимые для управления транзакциями, помещаются в дескриптор поставки. Все вопросы по обеспечению управления распределенными транзакциями в гетерогенных средах и взаимодействия с несколькими базами данных берет на

себя контейнер EJB. Контейнер обеспечивает защиту данных и гарантирует успешное подтверждение внесенных изменений; в противном случае транзакция откатывается.

Существуют два типа компонентов EJB: Session- и Entity-компоненты.

1.16.2 Session- и Entity-компоненты

Session-компонент представляет собой объект, созданный для обслуживания запросов одного клиента. В ответ на удаленный запрос клиента контейнер создает экземпляр такого компонента. Session-компонент всегда сопоставлен с одним клиентом, и его можно рассматривать как "представителя" клиента на стороне EJB-сервера. Такие компоненты могут «знать» о наличии транзакций – они могут отвечать за изменение информации в базах данных, но сами они непосредственно не связаны с представлением данных в БД.

Session-компоненты являются временными объектами. Обычно Session-компонент существует, пока создавший его клиент поддерживает с ним сеанс связи. После завершения связи с клиентом компонент уже никак не сопоставлен с ним. Объект считается временным, так как в случае завершения работы или краха сервера клиент должен будет создать новый компонент.

Обычно Session-компонент содержит параметры, которые характеризуют состояние его взаимодействия с клиентом, т.е. он сохраняет некоторое состояние между вызовами удаленных методов в процессе сеанса связи с клиентом.

Entity-компоненты представляют собой объектное представление данных из базы данных. Например, Entity-компонент может моделировать одну запись из таблицы реляционной базы данных (или набор записей из связанных таблиц). Ключевым отличием от Session-компонента является то, что несколько клиентов могут одновременно обращаться к одному экземпляру такого компонента. Entity-компоненты изменяют состояние сопоставленных с ними баз данных в контексте транзакций.

Состояние Entity-компонентов в общем случае нужно сохранять, и живут они столько, сколько существуют в базе данных те данные, которые они представляют, а не столько, сколько существует клиентский или серверный процесс. Остановка или крах контейнера EJB не приводит к уничтожению содержащихся в нем Entity-компонентов.

1.16.3 Составные части EJB компонента

EJB-компонент физически состоит из нескольких частей, включая сам компонент, реализацию некоторых интерфейсов и информационный

файл. Все это собирается вместе в специальный jar-файл – модуль развертывания.

Сам Enterprise Bean является Java-классом, разработанным поставщиком Enterprise Bean. Он реализует интерфейс Enterprise Bean и обеспечивает реализацию бизнес-методов, которые выполняет компонент. Класс не реализует никаких авторизации, многопоточности или кода транзакции.

Каждый создающийся Enterprise Bean должен иметь ассоциированный домашний интерфейс. Домашний интерфейс применяется как фабрика для компонента EJB. Клиент использует домашний интерфейс для нахождения экземпляра компонента EJB или создания нового экземпляра компонента EJB.

Удаленный интерфейс является Java-интерфейсом, который отображает через рефлексию те методы Enterprise Bean, которые необходимо показывать внешнему миру. Удаленный интерфейс играет ту же роль, что и IDL-интерфейс в CORBA, и обеспечивает возможность обращения клиента к компоненту.

Описатель развертывания является XML-файлом, который содержит информацию относительно компонента EJB. Использование XML позволяет установщику легко менять атрибуты компонента. Конфигурационные атрибуты, определенные в описателе развертывания, включают:

– имена домашнего и удаленного интерфейса;

– имя JNDI для публикации домашнего интерфейса компонента;

– транзакционные атрибуты для каждого метода компонента;

– контрольный список доступа для авторизации.

EJB-Jar-файл – это обычный java-jar-файл, который содержит компонент (компоненты) EJB, домашний и удаленный интерфейсы, а также описатель развертывания.

1.16.4 Роли EJB

Использование Enterprise JavaBeans предполагает разбиение процесса создания распределенного приложения на шесть отдельных этапов, с каждым из которых сопоставлены свои задачи и, соответственно, обязанности (роли) исполнителей. Эти роли можно разбить на три группы: инфраструктура, собственно разработка приложений и их поставка и настройка.

Главная задача такой структуры – облегчение участи разработчиков конечных приложений. При таком подходе разработчики EJB-контейнеров и серверов берут на себя решение многих проблем – в первую очередь, написание системных и платформенно-зависимых сервисов, элементы которых в противном случае пришлось бы писать прикладным программистам.

Роли обеспечения инфраструктуры

Server Provider создает платформу для разработки распределенного приложения и среду для его выполнения (EJB-сервер).

Container Provider предоставляет средства для поставки компонентов EJB и среду выполнения для установленных экземпляров компонентов (EJB-контейнер).

Роли разработки приложений

Bean Provider собственно разрабатывает компоненты – определяет remote- и home-интерфейсы компонента, реализует его бизнес-методы, а также создает дескриптор поставки (Deployment Descriptor) компонента. При разработке он не заботится о таких вещах, как обеспечение удаленного взаимодействия, управление транзакциями и безопасностью и прочими не имеющими к бизнес-логике конкретного компонента аспектами, так как эти проблемы должен решать Container Provider.

Application Assembler – это эксперт в прикладной области, который выполняет сборку приложения из готовых блоков (компонентов EJB) и добавляет некоторые другие элементы, например, апплеты или сервлеты для создания готового приложения. В процессе своей работы он имеет дело с интерфейсами компонентов EJB (а не с кодом его бизнес-методов), а также с дескриптором поставки. Результатом его работы может быть как готовое приложение, так и более сложный составной компонент EJB.

Роли поставки и настройки

Задача Deployer – настроить приложение, собранное из компонентов EJB, для его выполнения к конкретной операционной среде. Достигается это путем изменения свойств компонента. Например, deployer определяет поведение транзакций или профили безопасности путем установки тех или иных значений для свойств, находящихся в дескрипторе поставки. Его задачей также является интеграция приложения с существующими программными средствами управления корпоративной системой.

System Administrator работает с уже установленным приложением. Его задача – задание конфигурации и администрирование информационной инфраструктуры EJB-сервера и контейнеров. Administrator отслеживает поведение системы, определяя ее узкие места. Обычно при этом используются корпоративные средства контроля, с которыми (посредством специальных средств на уровне контейнера) интегрировано конкретное приложение.

В соответствии с такой схемой традиционный прикладной программист – это, как правило, bean provider и, возможно, application assembler. Эти роли позволяют такому разработчику сфокусировать свое внимание на разработке и реализации бизнес-логики системы. Deployer определяет значения параметров поставки в процессе установки компонента. Сложные вопросы реализации требований поставки берет на себя специально подготовленный поставщик программных средств. Хотя процесс создания распределенных приложений остается достаточно сложным, существенно облегчается работа обычного прикладного программиста – за счет делегирования многих вопросов разработчикам EJB-серверов и контейнеров.

Спецификация EJB достигает выполнения всех поставленных задач с помощью введения некоторого числа стандартных конструкций и соглашений. Это ограничивает свободу выбора той или иной архитектуры приложения, но позволяет разработчикам контейнеров и серверов делать определенные предположения о дизайне программ и эффективно управлять ими.

1.16.6 Инфраструктура Enterprise JavaBean

Создатели серверов и контейнеров EJB реализуют инфраструктуру EJB. Инфраструктура обеспечивает удаленное взаимодействие объектов, управление транзакциями и безопасность приложения. Спецификация EJB оговаривает требования к элементам инфраструктуры и определяет Java Application Programming Interface (API); она не касается вопросов выбора платформ, протоколов и других аспектов, связанных с реализацией.

Инфраструктура EJB должна обеспечивать канал связи с клиентом и другими компонентами EJB. Инфраструктура должна также обеспечить соблюдение прав доступа к компонентам EJB.

В общем случае необходимо гарантировать сохранение состояния компонентов в контейнерах. Инфраструктура EJB обязана предоставить возможности для интеграции приложения с существующими системами и приложениями – без этого нельзя говорить о пригодности приложения для функционирования в сложной информационной среде. Все аспекты взаимодействия клиентов с серверными компонентами должны происходить в контексте транзакций, управление которыми возлагается на инфраструктуру EJB. Для успешного выполнения процесса поставки компонентов инфраструктура EJB должна обеспечить возможность взаимодействия со средствами управления приложениями.

Таким образом, спецификация Enterprise JavaBeans – это существенный шаг к стандартизации модели распределенных объектов в Java. В настоящее время существует большое количество инструментов, поддерживающих этот подход и помогающих ускорить разработку EJB-ком-понентов.

Вопросы для самоконтроля

1. Что такое EJB?
2. Для чего служит Session-компонент?
3. Для чего служит Entity-компонент?
4. Какие существуют роли обеспечения инфраструктуры?
5. Что входит в инфраструктуру EJB?

2 Лабораторный практикум

2.1 Лабораторная работа «Как распараллелить программу»

Цель работы: закрепить на практике изученные этапы создание параллельных программ.

Ход работы

1. Разбейте задачу поиска двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников при разбиении отрезока [a,b] на n частей, а отрезока [c,d] – m частей на подзадачи. В качестве функций f(x,y) рассматривайте следующие:

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Выделите информационные зависимости между полученными подзадачами.

3. Определите процедуры необходимые при масштабировании задачи.

4. Определите распределение подзадач вежду процессами так, чтобы интенсивность взаимодействия между ними была минимальной.

5. Выполните шаги 1–4 для задачи вычисления произведения двух квадратных матриц порядка n.

2.2 Лабораторная работа «Разработка многопоточных приложений»

Цель работы: закрепить на практике знания по работе с потоками средствами операционной системы.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Для работы с потоками непосредственно используйте средства операционной системы. Значения K, a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Для работы с потоками непосредственно используйте средства операционной системы. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.3 Лабораторная работа «Параллельное программирование на C++ 11»

Цель работы: закрепить на практике знания по работе с потоками средствами языка C++11.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Используйте std::async из стандартной библиотеки C++ (стандарта 2011 года или позднее) для запуска потоков и соответствующие механизмы обмена информацией. Значения K, a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Используйте std::promise из стандартной библиотеки языка C++ (стандарта 2011 года или позже) для обмена информацией и соответствующих механизмов создания потока. Данные, которые доступны до создания потока, передавать в поток через механизм promise-future не имеет смысла. Также в целях проверки правильности реализации необходимо выводить результат пере-

множения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.4 Лабораторная работа «Программный интерфейс OpenMP»

Цель работы: закрепить на практике знания по работе с потоками средствами OpenMP.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Используйте OpenMP. Значения K, a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Используйте OpenMP. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.5 Лабораторная работа «Использование сокетов (API java.net)»

Цель работы: закрепить на практике знания по работе с потоками средствами сокетов из java.net.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами java.net. Количество потоков определяется при запуске системы. Значения a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами java.net. Количество потоков определяется при запуске системы. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.6 Лабораторная работа «Удаленный вызов методов (RMI)»

Цель работы: закрепить на практике знания по работе с потоками средствами RMI.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычис-

ления. Взаимодействие между JVM должно осуществляться средствами RMI. Количество потоков определяется при запуске системы. Значения a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами RMI. Количество потоков определяется при запуске системы. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.7 Лабораторная работа «Технология CORBA»

Цель работы: закрепить на практике знания по работе с потоками средствами RMI.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами CORBA. Количество потоков определяется при запуске системы. Значения a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$
- $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами CORBA. Количество потоков определяется при запуске системы. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.8 Лабораторная работа «Проектирование Web-сервисов»

Цель работы: закрепить на практике знания по организации работы web-сервисов.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области $D=[a,b]\times[c,d]$ методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Приложение должно представлять собой web-сервис. Количество потоков определяется при запуске системы. Значения a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

- $e^x \sin y$
- $\sin(x+2y)$
- $\cos(x \cdot e^y)$

– $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Приложение должно представлять собой web-сервис. Количество потоков определяется при запуске системы. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

– файл с ответом вычисленным в несколько потоков;
– файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

2.9 Лабораторная работа «Использование Java Message Service (JMS)»

Цель работы: закрепить на практике знания по работе с потоками средствами JMS.

Ход работы

1. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет значение двойного интеграла функции f(x,y) по области D=[a,b]×[c,d] методом прямоугольников, разбивая отрезок [a,b] на n частей, а отрезок [c,d] – m частей. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами JMS. Количество потоков определяется при запуске системы. Значения a, b, c, d, m, n должны задаваться пользователем непосредственно. Также пользователь осуществляет выбор интегрируемой функции f(x,y). Предполагаемые законы соответствия для функции f(x,y):

– $e^x \sin y$
– $\sin(x+2y)$
– $\cos(x \cdot e^y)$
– $\exp(\cos(x^2+y))$, $\exp(z) = e^z$

2. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3. Основываясь на результатах лабораторной работы 1, разработайте приложение, которое в K потоков вычисляет произведение двух квадратных матриц порядка n, заполненных случайными числами. Каждый из этих потоков должен выполняться в рамках отдельной JVM. Допустимо запускать дополнительные программы, координирующие процесс вычисления. Взаимодействие между JVM должно осуществляться средствами JMS. Количество потоков определяется при запуске системы. Также в целях проверки правильности реализации необходимо выводить результат перемножения тех же матриц, рассчитанный однопоточным алгоритмом. Вывод результатов должен осуществляться в два файла:

- файл с ответом вычисленным в несколько потоков;
- файл с ответом вычисленным в один поток.

4. Продемонстрируйте ускорение вычислений при увеличении числа потоков (запускайте с K = 1, K = 2, K = 4, K = 16).

3 Контроль знаний

3.1 Примерные задания контрольной работы № 1

Напишите программу, выполняющую вычисления согласно вашему варианту в K потоков (K – входные данные). Для реализации многопоточности используйте OpenMP.

Варианты

Вариант 1

Для введенной матрицы A подсчитайте минимум среди всех сумм по столбцам значений в строках с четными номерами. То есть,

$$\min_j \sum_i a_{(2i),j}.$$

Столбцы и строки нумеруются начиная с 1.

Вариант 2

Для введенной матрицы A подсчитайте максимум среди всех минимальных по строкам значений элементов в столбцах с нечетными номерами. То есть,

$$\max_i \min_j a_{i,(2j-1)}.$$

Столбцы и строки нумеруются начиная с 1.

Вариант 3

Для введенной матрицы A подсчитайте максимум среди всех сумм по столбцам значений в строках с четными номерами. То есть,

$$\max_j \sum_i a_{(2i),j}.$$

Столбцы и строки нумеруются начиная с 1.

Вариант 4

Для введенной матрицы A подсчитайте сумму среди всех минимальных по строкам значений элементов в столбцах с нечетными номерами. То есть,

$$\sum_i \min_j a_{i,(2j-1)}.$$

Столбцы и строки нумеруются начиная с 1.

3.2 Примерные задания контрольной работы № 2

Напишите программу, выполняющую вычисления согласно вашему варианту в K процессовов (K – параметр запуска).

Для реализации многопоточности используйте MPI.

Варианты

Вариант 1

Для введенной матрицы A подсчитайте

$$\max_j \sum_i a_{i,(2j)}.$$

Столбцы и строки нумеруются начиная с 1.

Вариант 2

Для введенной матрицы A подсчитайте

$$\min_j \max_i a_{(2i-1),j}.$$

Столбцы и строки нумеруются начиная с 1.

Вариант 3

Для введенной матрицы A подсчитайте

$$\min_j \max_i a_{i,(2j)}.$$

Столбцы и строки нумеруются начиная с 1.

Вариант 4

Для введенной матрицы A подсчитайте

$$\sum_i \min_i a_{(2i-1),j}.$$

Столбцы и строки нумеруются начиная с 1.

3.3 Примерный перечень экзаменационных вопросов

1. Особенности многопроцессорных вычислительных систем и разработки программ для них.
2. Пути достижения параллелизма и его режимы. Основные характеристики распределенных систем.
3. Примеры распределенных и параллельных систем. Решаемые с их помощью задачи.
4. Требование прозрачности распределенной системы. Виды прозрачности. Степень прозрачности.
5. Открытая системы. Свойства открытой системы. Преимущества открытой системы.
6. Масштаб распределенной системы и её масштабирование. Трудности и технологии достижение масштабируемости.
7. Классификация параллельных архитектур по Флинну и Хокни.
8. Вычислительный конвейер: принцип работы, классификация и конфликты.
9. Принципы построения параллельных вычислительных систем. Пути достижения параллелизма.
10. Классификация вычислительных систем.
11. Показатели эффективности параллельных алгоритмов. Закон Амдала.
12. Оценка коммуникационной трудоемкости параллельных алгоритмов в сетях с передачей пакетов и топологией типа кольцо.
13. Оценка коммуникационной трудоемкости параллельных алгоритмов в сетях с передачей пакетов и топологией типа решетка-тор.
14. Оценка коммуникационной трудоемкости параллельных алгоритмов в сетях с передачей пакетов и топологией типа гиперкуб.
15. Оценка коммуникационной трудоемкости параллельных алгоритмов в сетях с передачей сообщений и топологией типа кольцо.

16. Оценка коммуникационной трудоемкости параллельных алгоритмов в сетях с передачей сообщений и топологией типа решетка-тор.

17. Оценка коммуникационной трудоемкости параллельных алгоритмов в сетях с передачей сообщений и топологией типа гиперкуб.

18. Принципы проектирования параллельных алгоритмов. Разделение вычислений на независимые части.

19. Принципы проектирования параллельных алгоритмов. Выделение информационных зависимостей.

20. Принципы проектирования параллельных алгоритмов. Масштабирование набора подзадач.

21. Принципы проектирования параллельных алгоритмов. Распределение подзадач между процессорами.

22. Стандарт OpenMP. Модель памяти. Модель выполнения.

23. Стандарт OpenMP. Директива создания потоков и её опции. Директивы распределения работы.

24. Стандарт OpenMP. Директивы синхронизации.

25. Программирование распределенных приложений на основе парадигмы передачи сообщений. Стандарт MPI.

26. Стандарт MPI. Производные типы данных.

27. Стандарт MPI. Коллективные коммуникационные операции.

28. Программирование распределенных приложений с использованием RMI.

29. Технология RMI. Понятия удаленного интерфейса и удаленного объекта.

30. Технология RMI. Процесс передачи параметров, исключений и возвращаемого значения.

31. Технология RMI. Безопасность.

32. Стандарт C++11. Передача данных между потоками.

33. Стандарт C++11. Создание потоков. Асинхронный вызов.

34. Стандарт C++11. Синхронизация потоков.

35. Стандарт CORBA. Понятие брокера. Механизм удаленного вызова.

36. Стандарт CORBA. Язык описания интерфейсов (OMG IDL).

37. Разработка распределенных приложений для типовых классов алгоритмов. Вычисление интегралов.

38. Разработка распределенных приложений для типовых классов алгоритмов. Вычисление матричного произведения.

39. Разработка распределенных приложений для типовых классов алгоритмов. Решение систем линейных алгебраических уравнений.

40. Организация обмена сообщениями с помощью java.net.

41. Проблемы параллелизма. Транзакции. Их свойства и ресурсы.

42. Web-сервисы. SOAP, WSDL. Принципы REST.

43. Модели обмена сообщениями в JMS.

44. Инфраструктура EJB.

3.4 Примерный перечень экзаменационных задач

1. Вычисление интеграла $\int_0^1 \frac{4dx}{1+x^2}$ методом прямоугольников на системе с общей памятью используя OpenMP.

2. Вычисление интеграла $\int_0^{1/2} \frac{6dx}{\sqrt{1-x^2}}$ методом прямоугольников на системе с общей памятью используя OpenMP.

3. Вычисление интеграла $\int_0^1 4\sqrt{1-\mathrm{x}^2}\mathrm{dx}$ методом прямоугольников на системе с общей памятью используя OpenMP.

4. Вычисление интеграла $\int_0^1 \frac{4dx}{1+x^2}$ методом трапеций на системе с общей памятью используя OpenMP.

5. Вычисление интеграла $\int_0^{1/2} \frac{6dx}{\sqrt{1-x^2}}$ методом трапеций на системе с общей памятью используя OpenMP.

6. Вычисление интеграла $\int_0^1 4\sqrt{1-\mathrm{x}^2}\mathrm{dx}$ методом трапеций на системе с общей памятью используя OpenMP.

7. Вычисление частичных сумм ряда $\sum_{\mathrm{k}=0}^{\infty} \frac{4(-1)^k}{2k+1}$ на системе с общей памятью используя OpenMP.

8. Вычисление частичных сумм ряда $\sum_{\mathrm{k}=0}^{\infty} \frac{(50k-6)k!(2k)!}{2^k(3k)!}$ на системе с общей памятью используя OpenMP.

9. Умножение матриц на системе с общей памятью используя OpenMP.

10. Умножение матрицы на вектор на системе с общей памятью используя OpenMP.

11. Вычисление интеграла $\int_0^1 \frac{4dx}{1+x^2}$ методом прямоугольников на распределенной системе с использованием MPI.

12. Вычисление интеграла $\int_0^{1/2} \frac{6dx}{\sqrt{1-x^2}}$ методом прямоугольников на распределенной системе с использованием MPI.

13. Вычисление интеграла $\int_0^1 4\sqrt{1-\mathrm{x}^2}\mathrm{dx}$ методом прямоугольников на распределенной системе с использованием MPI.

14. Вычисление интеграла $\int_0^1 \frac{4dx}{1+x^2}$ методом трапеций на распределенной системе с использованием MPI.

15. Вычисление интеграла $\int_0^{1/2} \frac{6\mathrm{dx}}{\sqrt{1-\mathrm{x}^2}}$ методом трапеций на распределенной системе с использованием MPI.

16. Вычисление интеграла $\int_0^1 4\sqrt{1-\mathrm{x}^2}\mathrm{dx}$ методом трапеций на распределенной системе с использованием MPI.

17. Вычисление частичных сумм ряда $\sum_{k=0}^{\infty} \frac{4(-1)^k}{2k+1}$ на распределенной системе с использованием MPI.

18. Вычисление частичных сумм ряда $\sum_{k=0}^{\infty} \frac{(50k-6)k!(2k)!}{2^k(3k)!}$ на распределенной системе с использованием MPI.

19. Умножение матриц при ленточной схеме разделения данных на распределенной машине с использованием MPI.

20. Умножение матриц алгоритмом Фокса при блочным разделении данных на распределенной системе с использованием MPI.

21. Умножение матриц алгоритмом Кэннона при блочном разделении данных на распределенной системе с использованием MPI.

22. Умножение матрицы на вектор на распределенной системе с использованием MPI.

23. Умножение матрицы на вектор на распределенной системе с использованием RMI.

24. Вычисление суммы ряда $\sum_{k=0}^{\infty} \frac{4(-1)^k}{2k+1}$ на распределенной системе с использованием RMI.

25. Вычисление интеграла $\int_0^1 \frac{4dx}{1+x^2}$ методом прямоугольников на системе с общей памятью используя C++11.

26. Вычисление интеграла $\int_0^1 \frac{4dx}{1+x^2}$ методом трапеций на системе с общей памятью используя C++11.

27. Вычисление интеграла $\int_0^{1/2} \frac{6dx}{\sqrt{1-x^2}}$ методом трапеций на системе с общей памятью используя C++11.

28. Вычисление частичных сумм ряда $\sum_{k=0}^{\infty} \frac{4(-1)^k}{2k+1}$ на системе с общей памятью используя C++11.

29. Умножение матриц на системе с общей памятью используя C++11.

30. Умножение матрицы на вектор на системе с общей памятью используя C++11.

4 Вспомогательный материал
4.1 Пояснительная записка учебной программы

Учебная дисциплина «Распределенные и параллельные системы» входит в государственный компонент цикла специальных дисциплин.

Цель учебной дисциплины – обучение студентов методам проектирования распределенных систем, методам распараллеливания вычислений с использованием мощных вычислительных систем с распределенной и общей памятью.

Задачи учебной дисциплины:

– развитие у студентов доказательного, логического мышления;

– знакомство с различными языками, применяемыми на вычислительных системах с распределенной и общей памятью;

– подготовка к самостоятельному решению различных алгоритмических задач с использованием этих систем.

Курс «Распределенные и параллельные системы» основывается на следующих дисциплинах: «Операционные системы», «Методы вычислений», «Системное программирование» государственного компонента.

В результате изучения данной дисциплины студент должен

знать:

– архитектуру распределенных и параллельных приложений;

– общие принципы проектирования распределенных и параллельных приложений;

уметь:

– выбирать технологии проектирования в соответствии с задачей и имеющимся оборудованием;

– проектировать параллельные и распределенные приложения;

владеть:

– основными приемами проектирования параллельных и распределенных приложений;

– способами оценки эффективности созданных приложений.

Изучение дисциплины «Распределенные и параллельные системы» способствует формированию у студентов следующих академических компетенций. Специалист должен:

– Уметь применять базовые научно-теоретические знания для решения теоретических и практических задач.

– Владеть системным и сравнительным анализом.

– Владеть исследовательскими навыками.

– Уметь работать самостоятельно.

– Быть способным порождать новые идеи (обладать креативностью).

– Владеть междисциплинарным подходом при решении проблем.

А также способствует формированию следующих профессиональных компетенций. Специалист должен быть способен:

Проектно-конструкторская деятельность

– Проектировать, разрабатывать и тестировать программное обеспечение различных видов.

– Разрабатывать техническую документацию на программное обеспечение.

– Проектировать, разрабатывать системы баз данных.

Эксплуатационная деятельность

– На основе технической документации выполнять внедрение и сопровождение программного обеспечения, в том числе разработанного сторонними организациями.

– Выполнять системное администрирование, администрирование баз данных, администрирование насыщенных Интернет приложений.

В соответствии учебным планом для изучения дисциплины отводится 200 часов, из них 102 часа аудиторных, включая 36 лекционных часа, 30 часов лабораторных занятий и 36 часов управляемой самостоятельной работы. Дисциплина изучается на 3 курсе в 5 семестре. По дисциплине предусмотрено 2 контрольные работы и текущая аттестация в форме экзамена.

4.2 Учебно-методическая карта

№ темы	Название темы	Лекций (ч)	Лаб. занятий (ч)	УСР (ч)	Итого (ч)	Форма контроля знаний*
Модуль 1 «Параллельные системы»						
1	Введение	2		2	4	Т
2	Введение в параллельные вычисления	2			2	Т
3	Как распараллеливать программу	2	4	4	10	Т, Л
4	Проведение экспериментов			3	3	Т
5	Технологии параллельного программирования	2			2	Т
6	Разработка многопоточных приложений	4	6	6	16	Т, Л
7	Параллельное программирование на C++11	2	4	4	10	Т, Л
8	Программный интерфейс OpenMP	6	4		10	Т, Л
Итого по модулю 1		20	18	19	57	
Модуль 2 «Распределенный системы»						
9	Типичные архитектуры распределенных приложений	2			2	Т
10	Использование сокетов (API java.net)	2	2	2	6	Т, Л
11	Удаленный вызов методов (RMI)	4	4		8	Т, Л
12	Технология CORBA		2	4	6	Т, Л
13	Параллелизм в распределенных приложениях	4			4	Т
14	Проектирование Web-сервисов		2	6	8	Т, Л
15	Использование Java Message Service (JMS)	4	2		6	Т, Л

16	Технология Enterprise Java Bean (EJB)			5	5	Т, Л
Итого по модулю 2		16	12	17	45	
Итого по дисциплине		36	30	36	102	

*Обозначения:

Т – проверка теоретических знаний на экзамене;

Л – защита лабораторной работы.

4.3 Перечень заданий и контрольных мероприятий УСР

Задания на узнавание

1. Из указанных терминов выберите тот, который обозначает язык, предназначенный для описания web-сервиса и способов доступа к нему: а) SOAP, б) WADL, в) WDDX, г) XML, д) WSDL.

2. Укажите требование, которое не является обязательным для RESTful web-сервисов: а) клиент-серверная архитектура, б) отсутствие состояния, в) осуществление кэширования, г) единообразие интерфейса, д) механизм «код по требованию».

3. Какой из указанных типов компонентов предназначен для работы с персистентной информацией: а) Entity Beans, б) Stateless Session Beans, в) Stateful Session Beans, г) Message-Driven Beans, д) Persistence Beans.

Задание на воспроизведение

4. Объясните требование отсутствия состояния в RESTful web-сервисе.

5. Опишите существенные различия между Stateless Session Beans и Stateful Session Beans.

Задание на применение

6. Выполните распараллеливание алгоритма решения систем алгебраических уравнений методом Гаусса.

7. Определите критические ресурсы и секции в параллельном алгоритме решения систем алгебраических уравнений методом Гаусса. При необходимости добавьте синхронизацию.

8. Проведите измерения производительности параллельного алгоритма решения систем алгебраических уравнений методом Гаусса при выполнении в 1, 2 и 4 потоках. Вычислите ускорение и эффективность.

9. Составьте на языке WSDL описание службы авторизации.

10. Реализовать доступную по протоколу SOAP службу авторизации.

11. Реализовать Entity-компонент содержащий информацию об авторизованном пользователе.

12. Реализовать Session-компонент для авторизации пользователей.

4.4 Перечень рекомендуемой литературы

Основная:

1 Воеводин, В. В. Параллельные вычисления : учеб. пособие для студ. вузов, обуч. по напр. 510200 "Прикладная математика и информатика". – СПб. : БХВ-Петербург , 2002. – 600 с.

2 Хьюз, К. Параллельное и распределенное программирование с использованием C++ / [пер. с англ. и ред. Н. М. Ручко]. – Москва [и др.] : Вильямс, 2004. – 667 с.

3 Таненбаум, Э. Архитектура компьютера / [пер. с англ. Е. Матвеев]. – 6-е изд. – Санкт-Петербург [и др.] : Питер, 2014. – 816 с.

4 Орлов, С. А. Теория и практика языков программирования : учеб. по напр. "Информатика и вычислительная техника" : [для бакалавров и магистров]. – Санкт-Петербург [и др.] : Питер, 2013. – 688 с.

5 Васильев, А. Н. C#. Объектно-ориентированное программирование : учебный курс. – Санкт-Петербург [и др.] : Питер, 2012. – 316 с.

6 Васильев, А. Н. Java. Объектно-ориентированное программирование : для магистров и бакалавров : базовый курс по объектно-ориентированному программированию : [учеб. пособие]. – Санкт-Петербург [и др.] : Питер, 2014. – 397 с.

Дополнительная:

7 Хорстманн, К. С. Библиотека профессионала. Java 2. Том 1. Основы / К. С. Хорстманн, Г. Корнелл. – М.: “Вильямс”, 2004. – 848 с.

8 Хорстманн, К. С. Библиотека профессионала. Java 2. Том 2. Тонкости программирования / – М.: “Вильямс”, 2002. – 1120 с.

9 Дейтел, Х. Технологии программирования на Java 2. Книга 2. Распределенные приложения / Х. Дейтел, П. Дейтел, С. Сантри. – М.: “Бином”, 2009. – 464 с.

10 Дейтел, Х. Технологии программирования на Java 2. Книга 3. Корпоративные системы, сервлеты, JSP, Web-сервисы. / Х. Дейтел, П. Дейтел, С. Сантри. – М.: “Бином”, 2009. – 668 с.

11 Блинов, И. Н. Java. Методы программирования: учебно-методическое пособие / И. Н. Блинов, В. С. Романчик. – Минск: “Четыре четверти”, 2013. – 896 с.

12 Холл., М. Сервлеты и JavaServer Pages. Библиотека программиста. – СПб.: Питер, 2001. – 496 с.

13 Морган, С. Разработка распределенных приложений на платформе Microsoft .Net Framework / С. Морган, Б. Райан, Ш. Хорн, М. Бломсма. – СПб.: Питер, 1-е издание, 2008. – 608 с.

14 Гергель, В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем. – Издательство МГУ, 2010. – 544 с.

15 Эндрюс, Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Вильямс, 2003. – 512 с.

16 Уильямс, Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. – М.: ДМК Пресс, 2012. – 672 с.

17 Антонов, А. С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

18 Щупак, Ю. А. Win32 API. Разработка приложений для Windows. – СПб.: Питер, 2008. – 592 с.

19 Колисниченко, Д. Linux. От новичка к профессионалу. – СПб.: БХВ-Петербург, 2011. – 656 с.

20 Орлов, С. А. Организация ЭВМ и систем. [Фундаментальный курс по архитектуре и структуре современных компьютерных средств] : учеб. для студентов высш. учеб. заведений, обучающихся по напр. подготовки дипломированных специалистов "Информатика и вычислительная техника". – 2-е изд. – Санкт-Петербург [и др.] : Питер, 2011. – 688 с.

21 Трахтенгерц, Э. А. Программное обеспечение параллельных процессов / отв. ред. Б. А. Головкин. – Москва : Наука, 1987. – 272 с.

22 Ломазова, И. А. Вложенные сети Петри: моделирование и анализ распределенных систем с объектной структурой. – Москва : Научный мир, 2004. – 207 с.

23 Воеводин, В. В. Математические модели и методы в параллельных процессах. – Москва : Наука, 1986. – 296 с.

24 Корнеев, В. Д. Параллельное программирование в MPI. – Москва ; Ижевск : Институт компьютерных исследований, 2003. – 303 с.

25 Цимбал, А.А. Технологии создания распределенных систем. – СПб. [и др.] : Питер, 2003. – 575 с.

26 Блинов, И. Н. Java. Промышленное программирование : практическое пособие. – Минск : УниверсалПресс, 2007. – 704 с.

27 Рихтер, Д. Windows via C/C++. Программирование на языке Visual C++ [Электронный ресурс] : пер. с англ. – Электрон. текстовые дан. – Санкт-Петербург [и др.] : Питер, 2009. – 878 с.

28 Лиходед, Н. А. Методы распараллеливания гнезд циклов. – Минск : БГУ, 2008. – 100 с.

29 Паттерсон, Д. Архитектура компьютера и проектирование компьютерных систем – 4-е изд. – Санкт-Петербург [и др.] : Питер, 2012. – 784 с.

30 Бройдо, В. Л. Вычислительные системы, сети и телекоммуникации – 2-е изд. – Санкт-Петербург [и др.] : Питер, 2006. – 703 с.

31 Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. – Спб [и др.] : Питер, 2003 – 880 с.

Учебное издание

РАСПРЕДЕЛЕННЫЕ И ПАРАЛЛЕЛЬНЫЕ СИСТЕМЫ ДЛЯ НАПРАВЛЕНИЯ СПЕЦИАЛЬНОСТИ I СТУПЕНИ ВЫСШЕГО ОБРАЗОВАНИЯ 1-31 03 07-01 ПРИКЛАДНАЯ ИНФОРМАТИКА (ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРНЫХ СИСТЕМ)

Учебно-методический комплекс по учебной дисциплине

Составитель

СЕРГЕЕНКО Сергей Владимирович

Технический редактор *Г.В. Разбоева*

Компьютерный дизайн *А.В. Табанюхова*

Подписано в печать 2021. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.

Усл. печ. л. 8,13. Уч.-изд. л. 7,66. Тираж экз. Заказ

Издатель и полиграфическое исполнение – учреждение образования «Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя, изготовителя, распространителя печатных изданий № 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования «Витебский государственный университет имени П.М. Машерова». 210038, г. Витебск, Московский проспект, 33.