Кособуцкая Екатерина Владимировна

Литература:

- Воеводин В. В. "Параллельные вычисления"
- Таненбаум "Распределённые вычисления"

Сайты:

- parallel.ru
- openmp.org

Параллельное программирование

Введение

17.02.2016

Введение

Под *параллельными вычислениями* обычно понимают процессы обработки данных, в которых одновременно могут выполняться несколько операций компьютерной системы.

Достижение параллелизма возможно только при соблюдении следующих требований к архитектурным принципам построения вычислительной среды:

1. Независимость функционирования отдельных устройств вычислительной машины, а именно устройств ввода-вывода, процессоров, оперативной памяти.
2. Избыточность элементов вычислительной системы, которая может осуществляться в следующих формах: использование специальных устройств, например таких, как отдельные процессоры для целочисленной и вещественной арифметики; устройство многоуровневой памяти; дублирование устройств, например использование нескольких однотипных процессоров, несколько устройств оперативной памяти.

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

При рассмотрении проблемы организации параллельных вычислений необходимо различать следующие возможные режимы выполнения отдельных частей программы:

1. *Многозадачный* режим или режим *разделения времени*. Он является псевдопараллельным в силу того, что имеется один процессор, и соответственно только один процесс может быть активным.

2. Режим *параллельного выполнения*, когда в один и тот же момент времени может выполняться несколько команд обработки данных. Подобный режим обеспечивается, когда есть несколько процессоров или имеются конвейерные и векторные обрабатывающие устройства.
3. *Распределённые вычисления*, когда для параллельной обработки используется несколько устройств, достаточно удалённых друг от друга, в которых передача данных приводит к существенным временным задержкам. При необходимости использовать распределённые системы важным является разрабатывать такие параллельные алгоритмы, в которых интенсивность потоков межпроцессорных передач данных является низкой.

Проблемы, возникающие при параллельных и распределённых вычислениях

1. Проблемы координации

Если программа содержит подпрограммы или фрагменты, которые могут быть выполнены параллельно, и эти подпрограммы используют совместно некоторые ресурсы (например файлы, какие-либо устройства, области оперативной памяти), то это приводит к проблемам координации. Для решения этих проблем необходимо обеспечить связь между задачами и синхронизацию их работы.

При некорректной связи или синхронизации обычно возникает четыре вида проблем:

1. *Гонка данных (data race)*

Если несколько задач одновременно попытаются изменить некоторую область данных, а конечное значение данных зависит при этом от того, какая задача обратится к этой области первой, то возникает ситуация, которую называют *состоянием гонки (race condition)*. В том случае, когда несколько задач пытаются обновить один и тот же ресурс данных, то такое состояние гонок и называют гонкой данных.

Какая из задач будет выполняться первой зависит от планировщика задач операционной системы, состояния процессоров, времени ожидания и от различных случайных причин. Это и есть состояние гонок.

Для решения подобных проблем применяют определённые правила, например устанавливают очерёдность для синхронизации действий.

2. *Бесконечная отсрочка*

Это такое планирование, при котором одно или несколько задач должны ожидать до тех пор, пока не произойдёт некоторое событие или не создадутся определённые условия, которые могут оказаться непростыми для реализации. То есть ожидаемое условие или событие не является регулярным или между задачами отсутствует необходимая связь.

В результате, если одна или несколько задач ожидают сеанса связи, необходимой для их

выполнения, а этот сеанс не происходит, или происходит не полностью, или не происходит вообще, то такие задачи могут вообще не выполниться никогда. Таким образом возникает ситуация, называемая *бесконечной отсрочкой*.

3. *Взаимоблокировка (deadlock)*

Эта проблема также связана с ожиданием и состоит в следующем: если параллельно выполняемые задачи имеют доступ к общим данным, которые им разрешено обновить, то возможна ситуация, когда каждая из задач будет ожидать, пока другая освободит доступ к общим данным.

Например, задача B не может освободить данные 1, пока не получит доступ к данным 2, а задача C не может освободить данные 2, пока не получит доступ к данным 1. В результате две задачи оказываются в состоянии взаимной блокировки и могут ввести в состояние бесконечной отсрочки другие задачи.

Отметим, что взаимная блокировка и бесконечная отсрочка – это самые опасные проблемы.

4. *Трудности организации связи*

Многие распространённые параллельные среды состоят из *гетерогенных компьютерных сетей*.

Гетерогенные компьютерные сети – это системы, которые состоят из компьютеров различных типов, работающих в общем случае под управлением различных операционных систем и использующие различные сетевые протоколы. Процессоры в этих системах могут иметь различную архитектуру, обрабатывать слова различной длины и использовать различные машинные языки. Помимо различных операционных систем эти компьютеры могут различаться используемыми стратегиями планирования и системами приоритетов. Более того, все системы могут различаться параметрами передачи данных. Всё это делает обработку ошибок весьма трудной.

Неоднородность системы зачастую усугубляется и другими различиями, например может возникнуть необходимость в организации совместного использования данных программами, написанными на различных языках или разработанными с использованием различных моделей программного обеспечения. Это вносит проблемы межязыковой связи. И даже если распределённая или параллельная среда не является гетерогенной, то всё равно остаются проблемы взаимодействия между несколькими процессами или потоками.

В силу того, что каждый процесс имеет собственное адресное пространство, то для совместного использования переменных, параметров и значений возвращаемых функциями, необходимо применять технологию межпроцессорного взаимодействия. А это образует дополнительный уровень проектирования, тестирования и отладки при создании системы.

Имеется два базовых механизма, которые обеспечивают взаимодействие нескольких задач:

1. Общая память
2. Средство передачи сообщений

Для эффективного использования механизма общей памяти программист должен предусмотреть решение проблем гонки данных, взаимоблокировки и бесконечной отсрочки.

Схема передачи сообщений должна предусмотреть возникновение таких неполадок как прерывистые передачи, искажение, потеря информации, слишком длинные сообщения, просроченные, преждевременные и т. д.

2. Отказы оборудования

Сформируем три вопроса:

1. Что делать в случае отказа одного или нескольких процессоров при совместной работе множества процессоров над решением некоторой задачи? Должна ли программа завершиться (аварийно) или есть возможность перераспределения работы?
2. Что делать в случае выхода из строя канала связи?
3. Поток данных оказывается настолько медленным, что процессы на каждом конце связи превысят выделенный им лимит времени.

24.02.2016

3. Негативные последствия излишнего параллелизма и распределения

Сложность синхронизации или уровень связи между процессорами может потребовать таких затрат вычислительных ресурсов, что они отрицательно скажутся на производительности задач, совместно выполняющих общую задачу. В связи с этим возникают следующие вопросы:

- Как узнать, на сколько процессов, задач или потоков следует разделить программу?
- Существует ли оптимальное количество процессоров для любой заданной параллельной программы?
- В какой точке увеличение числа процессоров или компьютеров в системе приведёт к замедлению, а не ускорению её работы?

Необходимо различать работу и ресурсы, задействованные в управлении параллельными аппаратными средствами от работы, направленной на управление параллельно выполняемыми процессами и потоками в программном обеспечении.

Предел числа программных процессов может быть достигнут задолго до того, как будет достигнуто оптимальное количество процессоров и компьютеров. Также можно наблюдать снижение эффективности оборудования ещё до достижения оптимального количества параллельно выполняемых задач.

4. Выбор архитектуры

Архитектурных решений, которые поддерживают параллелизм, существует достаточно много. Архитектурное решение может считаться корректным, если оно соответствует декомпозиции работ программного обеспечения.

Например, некоторые распределённые архитектуры, прекрасно работающие в web среде, обречены на неудачу в реальном масштабе времени. В частности, распределённые архитектуры, которые рассчитаны на длинные временные задержки, вполне приемлемы в web среде, такие как система функционирования электронной почты, но совершенно неприемлемы для таких систем реального времени, как банкоматы.

Выбранная архитектура также может оказать серьёзное воздействие на параллельную обработку данных. Например, методы векторной обработки наилучшим образом подходят для решения определённых математических задач и проблем имитационного моделирования, но совершенно неэффективны в применении к мультиагентным алгоритмам планирования.

Распространённые архитектуры программного обеспечения, используемые для поддержки параллельного и распределённого программирования.

Модель	Архитектура	Распределённое программирование	Параллельное программирование
Модель ведущего узла клиент-сервер (главный-подчиненённый, управляющий-рабочий)	Есть главный узел, управляющий задачами, который контролирует их выполнение и передаёт работу подчинённым задачам.	+	+
Модель равноправных узлов	Все задачи в основном имеют одинаковый ранг, и работа между ними распределяется равномерно.	-	+
Векторная (конвейерная, поточная) обработка	Один исполнительный узел соответствует каждому элементу массива (вектора) или шагу конвейера.	+	+
Дерево с родительскими и дочерними элементами	Динамически генерируемые исполнители типа "родитель-потомок".	+	+

	Данный тип архитектуры полезно использовать в алгоритмах следующих типов: рекурсивные алгоритмы, алгоритмы "разделяй и властвуй", алгоритмы "и/или" и древовидная обработка.		

5. Тестирование и отладка

При параллельном и распределённом программировании трудно воспроизвести точный контекст параллельных или распределённых задач из-за различных стратегий планирования, применяемых в операционной системе, динамически меняющейся рабочей нагрузке, квантов процессорного времени, приоритетов процессов и потоков, временных задержек при их взаимодействии и собственно выполнении, а также различных случайных изменений ситуаций, характерных для параллельных или распределённых контекстов.

Чтобы воспроизвести точное состояние, в котором находилась среда при тестировании и отладке, необходимо воссоздать каждую задачу, исполняемую операционной системой. При этом должен быть известен режим планирования процессорного времени и точно воспроизведены состояние виртуальной памяти и переключение контекстов. Кроме того, следует воссоздать условия возникновения прерываний и формирования сигналов, а в некоторых случаях даже рабочую нагрузку сети.

При этом надо понимать, что сами средства тестирования и отладки также оказывают влияние на состояние среды. Всё это означает, что точное воспроизведение событий для тестирования и отладки зачастую невозможно.

Два основных подхода к достижению параллельности вычислений

Два базовых подхода к распараллеливанию вычислений: *параллельное* и *распределённое* программирование. Эти подходы разные, но могут пересекаться.

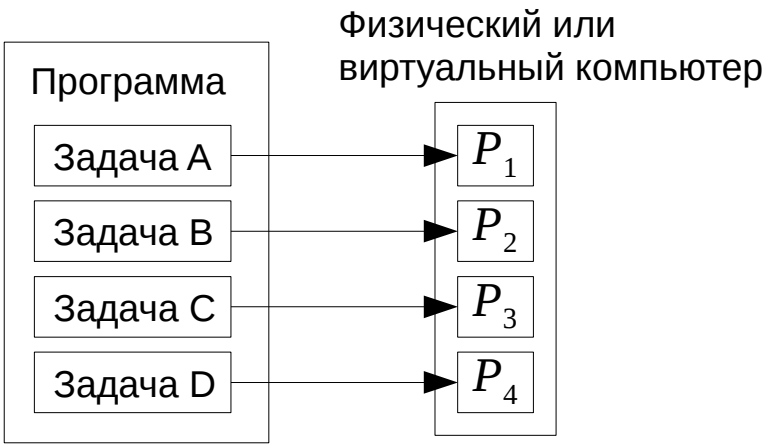
Методы параллельного программирования позволяют распределить работу программы между двумя и более процессорами в рамках одного физического или одного виртуального компьютера.

Методы распределённого программирования позволяют распределить работу программы между двумя и более процессами, причём процессы могут существовать на одном и том же компьютере или на разных. Зачастую части распределённой программы выполняются на разных компьютерах, связываемых по сети.

Надо отметить, что не все распределённые программы включают параллелизм, потому что части распределённой программы могут выполняться по различным запросам и в различные периоды времени.

Например, программу Календарь можно разделить на две составляющие: одна часть должна обеспечивать пользователя информацией, присущей календарю, и способом записи данных о важных для него встречах, а другая часть должна предоставлять пользователю набор сигналов для разных типов встреч. Пользователь составляет расписание встреч, используя одну часть программного обеспечения, в то время как другая часть выполняется независимо от первой. Набор сигналов и компонентов расписания представляют собой единое приложение, которое разделено на две части, выполняемые по отдельности. При так называемом чистом параллелизме одновременно выполняемые части являются компонентами одной и той же программы, а части распределённых приложений обычно реализуются как отдельные программы.

а) Параллельное выполнение задачи



б) Распределённое выполнение задачи

2.03.2016

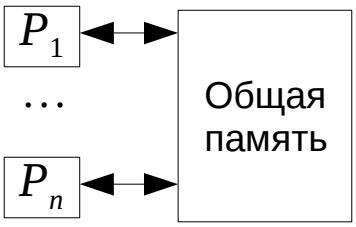
Преимущества параллельного программирования

Одно из основных преимуществ – ускорение работы.

Представим простейшую модель параллельного программирования, которая называется PRAM (parallel random-access machine – параллельная машина с произвольным доступом).

Это упрощённая модель с n процессорами, которые используют общую глобальную память.

Схематично можно изобразить:



Все процессоры имеют доступ для чтения и записи в общую память, могут параллельно выполнять арифметические и логические операции. Кроме того, каждый из теоретических процессоров может обращаться к общей памяти в одну непрерывную единицу времени.

Подобная модель позволяет выполнять как *параллельные*, так и *исключающие* алгоритмы считывания данных. Параллельные алгоритмы считывания данных позволяют одновременно обращаться к одной и той же области памяти без искажения данных, а исключающие алгоритмы считывания данных используются тогда, когда необходима гарантия, что никакие два процессора никогда не будут считывать данные из одной и той же области памяти одновременно.

Указанная модель PRAM позволяет использовать и те и другие алгоритмы считывания данных.

Аналогично *алгоритмы записи данных* также существуют в двух видах: *параллельные* и *исключающие*. Они обладают такими же свойствами.

В связи с этим существует четыре алгоритма доступа к данным, которые могут быть охарактеризованы следующим образом:

1. Исключающее считывание, исключающая запись

   Доступ к данным может получить только одна задача

2. Параллельное считывание, исключающая запись

   Осуществляется множественный доступ для чтения, а записывать может только одна задача. В этой ситуации все считывающие задачи могут прочесть различные значения.

3. Исключающее считывание, параллельная запись

4. Параллельное считывание, параллельная запись

Приведём простейшую классификацию схем параллелизма (также существуют другие схемы, см. Воеводина).

Упрощённая классификация схем функционирования параллельных компьютеров была предложена М. Флинном, и согласно этой классификации первоначально различались две схемы:

- SIMD (single instruction, multiple data) – архитектура с одним потоком команд и многими потоками данных.
- MIMD (multiple instruction, multipe data) – архитектура со множеством потоков инструкций и множеством потоков данных.

Немного позже эти архитектуры были расширены до:

- SPMD (single program, multiple data) – одна программа и несколько потоков данных
- MPMD (multiple program, multipe data) – множество программ и множество потоков данных.

Схемы SIMD или SPMD позволяют нескольким процессорам выполнять одну и ту же инструкцию или программу при условии, что каждый процессор получает доступ к различным данным. Другие две схемы MIMD и MPMD позволяют работать нескольким процессорам, причём все они выполняют различные программы или инструкции и пользуются собственными данными.

Надо отметить, что в реальности возможны гибриды этих моделей, в которых одна группа процессоров образует например SPMD модель, а другая группа – MPMD.

Преимущества распределённого программирования

Преимущество заключается в том, что работа может быть выполнена быстрее, и ресурсы могут быть распределены.

Также существуют модели распределённого программирования, самая простая из которых – это модель клиент-сервер.

В этой модели программа разбивается на две части, одна из которых называется сервером, другая – клиентом. Сервер обычно имеет прямой доступ к некоторым аппаратным и программным ресурсам, которые желает использовать клиент.

В большинстве случаев сервер и клиент располагаются на разных компьютерах. Если говорить в терминологии баз данных, то между клиентом и сервером существует отношение "множество к одному" $\infty \to 1$, то есть один сервер отвечает на запросы множества клиентов.

Иногда вместо клиент-сервер используется термин *изготовитель-потребитель*. Как правило

это программы меньшего объёма.

Также существуют *мультиагентные распределённые системы*. В них используются *агенты*, которые представляют собой компоненты программного обеспечения, характеризующиеся автономностью и тем, что они могут постоянно находиться в состоянии выполнения. Агенты могут как создавать запросы другим программным компонентам, так и отвечать на запросы других программных компонентов. Агенты сотрудничают в пределах групп для коллективного выполнения определённых задач. В такой модели не существует конкретного клиента или сервера. Это модель сети с равноправными узлами и называется peer-to-peer. Права у компонентов одинаковые, при этом у каждого компонента есть что предложить другому.

Агенты являются распределёнными, так как все они располагаются на разных серверах интернета и для связи используют согласованный интернет протокол.

Минимальные требования разработки параллельных и распределённых программ

В любом случае, написанию программы должен предшествовать *процесс проектирования*.

Процесс проектирования параллельного и распределённого программного обеспечения должен включать три составляющие:

1. Декомпозиция
2. Связь
3. Синхронизация

Декомпозиция – процесс разбиения задачи на части. Возможны различные подходы. Части могут компоноваться по используемым ресурсам или по функциям, которые они выполняют.

При декомпозиции определяется, что должны делать различные части программного обеспечения. Одна из основных проблем – определение естественной декомпозиции.

Связь отражает вопрос о взаимосвязи параллельных частей. Здесь могут возникнуть следующие вопросы, которые необходимо решить на этапе проектирования программного обеспечения:

- Каким образом реализовать связь, если части программного обеспечения разнесены по различным процессам или различным компьютерам?
- Должны ли различные части программного обеспечения совместно использовать общую память?
- Каким образом одна часть программного обеспечения узнает о том, что другая справилась с задачей?

- Какая часть должна первой приступить к работе?
- Откуда один компонент знает об отказе другого компонента?

Если отдельным частям программного обеспечения не нужно связываться между собой, значит они в действительности не образуют единое приложение.

Синхронизация понимается как координация работы множества компонент, выполняющих одну задачу. Здесь также можно перечислить ряд вопросов, на которые необходимо ответить при проектировании программного обеспечения:

- Все ли части программного обеспечения должны приступать к работе одновременно или только некоторые, а остальные пока могут находиться в состоянии ожидания?
- Каким двум или более компонентам необходим доступ к одному и тому же ресурсу? Кто имеет право получить его первым?
- Если некоторые части программного обеспечения завершат работу гораздо раньше других, то надо ли поручать им другую работу и кто это должен делать?

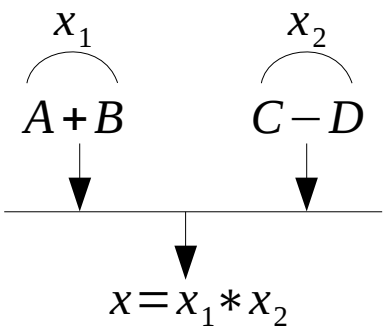
При проектировании также необходимо учитывать, что существует несколько уровней программного параллелизма.

Базовые уровни программного параллелизма

1. Параллелизм на уровне инструкций
2. Параллелизм на уровне подпрограмм
3. Параллелизм на уровне объектов
4. Параллелизм на уровне приложений

Параллелизм на уровне инструкций

Такого рода параллелизм возникает, если несколько частей одной инструкции могут выполняться одновременно. Предположим, есть инструкция $x=(A+B)*(C-D)$.



Параллелизм на уровне подпрограмм

Иногда выполнение отдельных подпрограмм можно поручить различным потокам, и если имеется достаточное число процессоров, которым можно поручать выполнение каждой отдельной подпрограммы, то они будут выполняться одновременно.

Параллелизм на уровне объектов

Когда каждый объект можно назначить отдельному потоку или процессу. Существует стандарт, который позволяет это делать. CORBA (Common Object Request Broker Architecture – общая архитектура брокера объектных запросов) – технология построения распределённых объектных приложений. Он позволяет объектам в разных потоках выполнять свои методы параллельно.

Параллелизм на уровне приложений

Когда несколько приложений могут сообща решать некоторую проблему. Позволяет приложениям сотрудничать между собой. В качестве примера может служить буфер обмена.

Отметим, что второй и третий уровни параллелизма – это основные уровни.

Также существуют уровни операционной системы и аппаратных средств (затрагивать их в курсе не будем).