Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

АРХИТЕКТУРА ВЫЧИСЛЕНИЙ. ОБРАБОТКА БОЛЬШИХ ОБЪЕМОВ ДАННЫХ

Методические рекомендации

Витебск
ВГУ имени П.М. Машерова
2024

УДК 004.2:004.415:004.043(075.8)
ББК 32.971.35-02я73+22.19я73+16.23я73
А87

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 2 от 20.12.2023.

Составители: преподаватели кафедры прикладного и системного программирования ВГУ имени П.М. Машерова **Ю.В. Исаченко, В.А. Степанов**

Р е ц е н з е н т ы :
доцент кафедры информационных технологий и управления бизнесом
ВГУ имени П.М. Машерова,
кандидат физико-математических наук *С.А. Прохожий*;
заведующий кафедрой «Математика и информационные технологии»
УО «ВГТУ», кандидат физико-математических наук,
доцент *Т.В. Никонова*

А87 **Архитектура вычислений. Обработка больших объемов данных :** методические рекомендации / сост.: Ю.В. Исаченко, В.А. Степанов. – Витебск : ВГУ имени П.М. Машерова, 2024. – 45 с.

В методических рекомендациях изложены основные понятия распределенных вычислений, больших объемов данных.

Издание предназначается для студентов первой ступени высшего образования по специальностям: «Программное обеспечение информационных технологий / Программная инженерия» (дисциплины «Распределенные вычисления», «Обработка больших объемов данных») дневной и заочной форм получения образования; «Прикладная информатика» (по направлениям) (дисциплины «Анализ и обработка больших данных», «Методы и обработка больших данных», «Распределенные и параллельные системы»); «Информационные системы и технологии» (дисциплина «Обработка больших объемов информации»); «Прикладная математика» (дисциплина «Анализ и обработка больших данных»), а также студентов второй ступени высшего образования по специальности «Информатика и технологии программирования» (дисциплины «Облачные сервисы в предпринимательской деятельности», «Методы обработки больших объемов данных»).

УДК 004.2:004.415:004.043(075.8)
ББК 32.971.35-02я73+22.19я73+16.23я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сегодняшние реалии таковы, что разработка практически любого программного обеспечения требует хороших знаний параллельного и распределенного программирования. Обе эти области объединяет то, что и параллельное, и распределенное программное обеспечение состоит из нескольких процессов, которые вместе решают одну общую задачу.

За последнее десятилетие объем данных, с которыми приходится иметь дело, многократно увеличился, и в то же время стоимость хранения данных снизилась. Частные компании и исследовательские учреждения обрабатывают терабайты информации о взаимодействиях своих пользователей, бизнесе, социальных сетях, а также собирают данные с датчиков таких устройств, как мобильные телефоны и автомобили. Задача современной эпохи состоит в том, чтобы разобраться в этом объеме данных. Именно здесь аналитика больших данных вступает в свои права.

Big Data Analytics в основном включает в себя сбор данных из разных источников, изменение их таким образом, чтобы они стали доступными для использования аналитиками, и, наконец, предоставление продуктов данных, полезных для бизнеса.

В данном учебном издании изложены основные понятия распределенных вычислений и больших данных.

В первом разделе рассматриваются основные понятия параллельных вычислений и распараллеливания.

Во втором – понятия жизненного цикла данных, больших данных и обработки больших данных.

Материал соответствует темам учебных программ курсов: «Распределенные вычисления», «Обработка больших объемов данных (специальности «Программное обеспечение информационных технологий / Программная инженерия») дневной и заочной форм получения образования; «Анализ и обработка больших данных», «Методы и обработка больших данных», «Распределенные и параллельные системы» (специальность «Прикладная информатика» (по направлениям)); «Обработка больших объемов информации» (специальность «Информационные системы и технологии»); «Анализ и обработка больших данных» (специальность «Прикладная математика»); «Облачные сервисы в предпринимательской деятельности», «Методы обработки больших объемов данных» (специальность «Информатика и технологии программирования»).

РАЗДЕЛ I. АРХИТЕКТУРА ВЫЧИСЛЕНИЙ

1.1. Теоретические основы параллельных вычислений

1.1.1. История развития параллельных вычислений

Разговор о развитии параллельного программирования принято начинать истории развития суперкомпьютеров. Однако первый в мире суперкомпьютер CDC6600, созданный в 1963 г., имел только один центральный процессор, поэтому едва ли можно считать его полноценной SMP-системой. Архитектура SMP (от англ. Symmetric Multiprocessing) подразумевает работу несколько идентичных процессоров с общей для них оперативной памятью. Многоядерный процессор можно считать частным случаем SMP-системы.

Третий в истории суперкомпьютер CDC8600 проектировался для использования четырёх процессоров с общей памятью, что позволяет говорить о первом случае применения SMP, однако CDC8600 так никогда и не был выпущен: его разработка была прекращена в 1972 году.

Лишь в 1983 году удалось создать работающий суперкомпьютер (Cray X-MP), в котором использовалось два центральных процессора, использовавших общую память. Справедливости ради стоит отметить, что чуть раньше (в 1980 году) появился первый отечественный многопроцессорный компьютер Эльбрус-1, однако он по производительности значительно уступал суперкомпьютерам того времени.

Уже в 1994 можно был свободно купить настольный компьютер с двумя процессорами, когда компания ASUS выпустила свою первую материнскую плату с двумя сокетами, т.е. разъёмами для установки процессоров.

Следующей вехой в развитии SMP-систем стало появление многоядерных процессоров. Первым многоядерным процессором массового использования стал POWER4, выпущенный фирмой IBM в 2001 году. Но по-настоящему широкое распространение многоядерная архитектура получала лишь в 2005 году, когда компании AMD и Intel выпустили свои первые двухъядерные процессоры.

1.1.2. Термины и определения

Русскоязычная терминология в области параллельных вычислений (ПВ) не всегда однозначно соответствует англоязычной, поэтому ниже для каждого термина даётся его англоязычный вариант и делается поправка на не идентичность этих терминов, где это необходимо.

Параллельные вычисления (concurrent computing) – способ организации вычислений на одном или нескольких компьютерах, при котором пересекаются периоды жизни нескольких задач. Антонимом этого термина являются **последовательные вычисления (sequential computing)**, при выполнении которых периоды жизни задач не пересекаются. Например, пусть

$start_i$, end_i – это соответственно времена начала и конца жизни вычислительной задачи i, и пусть $start_1 < start_2$, тогда:

при $end_1 < start_2$ имеют место последовательные вычисления;

при $end_1 \geq start_2$ имеют место параллельные вычисления.

Англоязычный термин **parallel computing** переводится на русский язык тем же словосочетанием: параллельные вычисления. Однако в него вкладывается более узкий, чем в concurrent computing, смысл: при parallel computing задачи исполняются физически одновременно на различных процессорах и/или ядрах одного компьютера. Это значит, что понятие concurrent включает в себя parallel, а именно: любые parallel-вычисления являются concurrent, но не всякие concurrent-вычисления являются parallel.

Классический пример concurrent-вычислений, которые не являются parallel, – это реализация многозадачности в операционной системе (ОС) при наличии только одного одноядерного процессора. В этом случае ОС не может физически параллельно выполнять разные задачи и вынуждена запускать их в режиме разделения времени, т.е. поочерёдно разрешая использовать процессор разным задачам, переключаясь с задачи на задачу по несколько раз до окончания выполнения каждой из них.

Иногда parallel computing переводится как многоядерные вычисления (multicore computing), чтобы подчеркнуть отличие от concurrent computing, однако этот термин неидеален, т.к. не позволяет корректно классифицировать вычисления на многопроцессорных компьютерах, в которых каждый процессор является одноядерным. Такие компьютеры позволяют выполнять parallel computing, но не multicore computing. Но этой проблемой можно пренебречь, т.к. подобных компьютеров в данный момент практически нет на рынке. Более точным термином можно считать SMP (shared memory processing), который относится к работе параллельных программ на системах с общей памятью. В таких системах все процессоры/ядра совместно используют общую оперативную память одного компьютера. Итак, можно установить следующие пары соответствий:

параллельные вычисления ≠ parallel computing;

параллельные вычисления = concurrent computing;

многоядерные вычисления = parallel computing;

parallel computing = multicore computing = SMP.

Распределенные вычисления (distributed computing) – такие ПВ, при которых для решения задачи вычисления происходят на процессорах, расположенных на разных компьютерах, соединенных сетью, т.е. для выполнения вычислений приходится передавать программы и/или данные по сети.

Классификация ПВ по особенностям аппаратной реализации:

1. Параллелизм на уровне битов – процессор выполняет операцию для всех битов машинного слова одновременно. Например, 64-разрядный процессор может одновременно инвертировать значение каждого из 64 битов заданного операнда.

2. Параллелизм на уровне операндов – одна инструкция процессора позволяет выполнить некоторую операцию для целого массива операндов параллельно. Например, с помощью технологии SSE за одну операцию можно попарно перемножить элементы двух массивов. (все умножения будут выполнены параллельно во времени).

3. Параллелизм на уровне инструкций – выполнение каждой инструкции разбивается на фазы, каждая из которых может выполняться процессором физически параллельно. Это изменение архитектуры процессора никак не влияет на общее время выполнения одной изолированной инструкции, однако при обработке нескольких подряд идущих инструкций удаётся организовать из них конвейер. В результате подряд идущие инструкции выполняются физически параллельно, что позволяет увеличить общую производительность процессора, выраженную в инструкциях.

Перечисленные далее параллельные технологии в данном пособии рассматриваются кратко, чуть шире, чем определения:

• *Конвейерная обработка данных* (суперскалярность) представляет собой одновременную обработку процессором нескольких инструкций, при котором в один момент времени для каждой из инструкций выполняется различный этап выполнения. Например, если какой-либо процессор может одновременно получать, раскодировать и выполнить инструкцию, то он во время получения первой инструкции может раскодировать вторую и выполнять третью (рисунок 1.1.2.1). Этот способ организации вычислений не является параллельными вычислениями, потому что инструкции все равно выполняются последовательно, а задействовано только одно ядро.

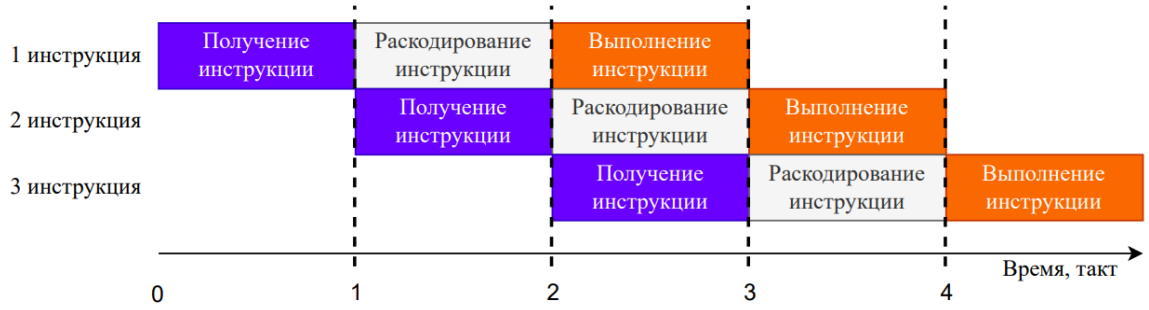

Рис. 1.1.2.1 – Конвейерная обработка инструкций

• *SIMD-расширения* (MMX, SSE) обеспечивают параллелизм на уровне данных. Например, процессор может одновременно умножать четыре числа вместо одного с помощью SSE инструкции. Однако поток команд все равно остается одиночным, т.е. выполняется одна инструкция программы за промежуток времени, что не является случаем параллельных вычислений. *Вытесняющая многозадачность* организуется операционной системой. Несколько процессов стоят в очереди выполнения и ОС сама решает как распорядиться процессорным временем между ними. Если у первого потока задан больший приоритет, чем у второго, то ОС будет выделять больше времени на выполнение первого потока, однако в один момент времени будет выполняться только один поток, следовательно, вытесняющая многозадачность тоже не входит в понятие параллельных вычислений.

Для организации параллельных вычислений используются различные технологии распараллеливания:

• **Process (процесс)** – наиболее тяжеловесный механизм, применяемый для распараллеливания. Каждый процесс имеет свое независимое адресное пространство, поэтому синхронизация данных между процессами долгая и сложная. Может включать в себя несколько потоков исполнения.

• **Thread (поток исполнения, нить, тред, поток)** выполняется независимо от других потоков, но имеет общее адресное пространство с другими потоками в рамках одного процесса. На этом уровне используется механизмы синхронизации данных (будут рассмотрены далее).

• **Fiber (волокно)** – легковесный поток выполнения. Также как и треды, fiber’ы имеют общее адресное пространство, однако используют совместную многозадачность вместо вытесняющей. ОС не переключает контекст из одного треда в другой, вместо этого главный поток сам выделяет время для работы дочернего fiber, либо блокируется логически (то есть жизненным циклом fiber’а управляет программист). Также все fiber’ы работают на одном ядре, в отличие от тредов, которые могут работать на разных ядрах.

Для лучшего понимания тредов схематично рассмотрим его жизненный цикл (lifecycle). На рисунке 1.1.2.2 видно, что поток может находиться в трех состояниях – готовность, ожидание и выполнение. После создания потока он пребывает в состоянии готовности. Затем ОС принимает решение о смене его состояния (вытесняющая многозадачность). Для fiber жизненный цикл такой же, но переходами между ними управляет программист или механизмы синхронизации.

Разные стандарты языков программирования могут добавлять в жизненный цикл потоков новые состояния, например, блокировка потока, прерывание работы потока и остальные, однако общая схема работы остается той же.

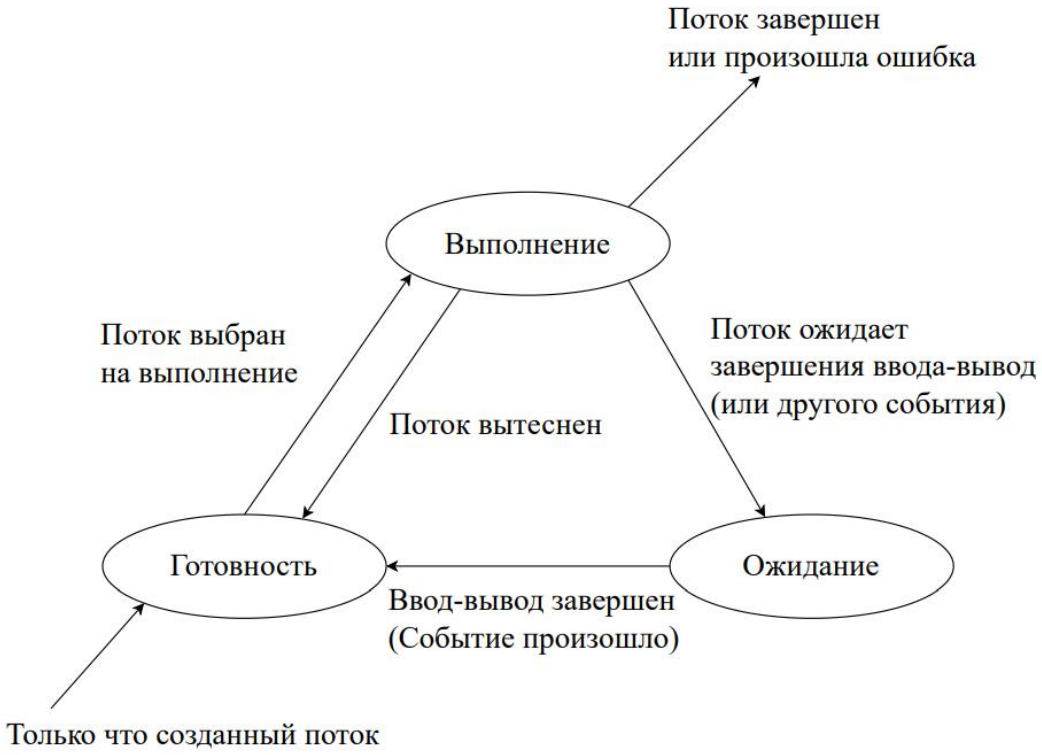

Рис. 1.1.2.2 – Жизненный цикл потока

1.1.3. Классификация параллельных систем (архитектур)

По физической архитектуре параллельные системы можно разделить на два типа:

1. ***SMP*** (Shared Memory Parallelism, Symmetric MultiProcessor system) – многопроцессорность, многоядерность, GPGPU.

2. ***MPP*** (Massively Parallel Processing) – кластерные системы, GRID (распределенные вычисления).

Далее рассмотрим эти две архитектуры подробнее.

SMP – архитектура многопроцессорных систем, в которой два или более одинаковых процессора сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам) и выполняют одни и те же функции (почему, собственно, система и называется симметричной). В английском языке SMP-системы носят также название tightly coupled multiprocessors, так как в этом классе систем процессоры тесно связаны друг с другом через общую шину, имеют равный доступ ко всем ресурсам вычислительной системы (памяти и устройствам ввода-вывода) и управляются лишь одним экземпляром операционной системы. В этой архитектуре все процессоры расположены на одной физической машине, поэтому они имеют общие банки памяти. Существует два вида подключения процессоров к общей памяти.

Соединение по общей шине (*system bus*) изображено на рисунке 1.1.3.1. В этом случае только один процессор может обращаться к памяти в каждый данный момент, что накладывает существенное ограничение на количество процессоров, поддерживаемых в таких системах. Чем больше процессоров, тем больше нагрузка на общую шину, тем дольше должен ждать каждый процессор, пока освободится шина, чтобы обратиться к памяти. Снижение общей производительности такой системы с ростом количества процессоров происходит очень быстро, поэтому обычно в таких системах количество процессоров не превышает 2–4. Примером SMP-машин с таким способом соединения процессоров являются любые многопроцессорные серверы начального уровня.

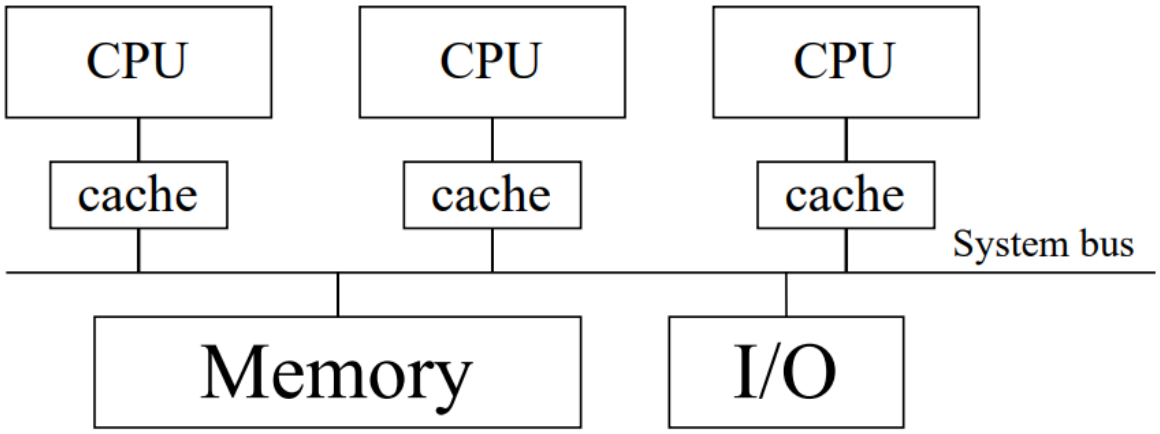

Рис. 1.1.3.1 – Архитектура SMP.
Подключение процессоров по системной шине

Коммутируемое соединение (crossbar switch) изображено на рисунке 1.1.3.2. При таком соединении вся общая память делится на банки памяти, каждый банк памяти имеет свою собственную шину, и процессоры соединены со всеми шинами, имея доступ по ним к любому из банков памяти. Такое соединение схемотехнически более сложное, но оно позволяет процессорам обращаться к общей памяти одновременно. Это позволяет увеличить количество процессоров в системе до 8–16 без заметного снижения общей производительности.

Плюсами такого подхода является высокая скорость обмена данными между процессорами и относительная простота в разработке ПО. Однако могут возникнуть проблемы с масштабируемостью системы (если на материнской плате есть только два сокета, то три процессора уже не поставить).

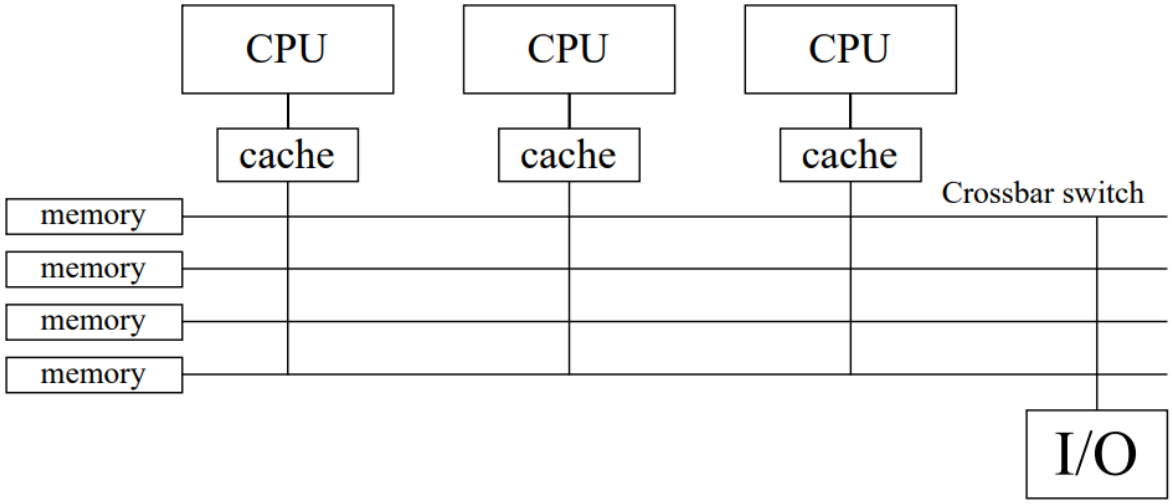

Рис. 1.1.3.2 – Архитектура SMP.
Подключение процессоров через коммутируемое соединение

MM – архитектура многопроцессорных систем, при которой память между процессорами разделена физически. На таких системах проводятся распределенные вычисления. Система строится из отдельных узлов, содержащих процессор, локальный банк оперативной памяти, коммуникационные процессоры или сетевые адаптеры, иногда – жёсткие диски и другие устройства ввода-вывода. Доступ к банку оперативной памяти данного узла имеют только процессоры из этого же узла. Узлы соединяются специальными коммуникационными каналами (рисунок 1.1.3.3).

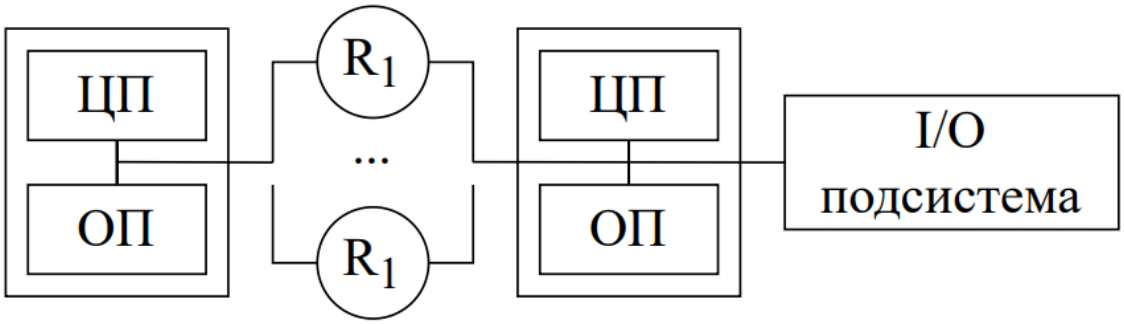

Рис. 1.1.3.3 – Архитектура MMP

Плюсами такого подхода является хорошая масштабируемость (при необходимости в увеличении производительности системы достаточно просто добавить еще узлов). Однако существенно понижается скорость межпроцессорного обмена, так как теперь банки памяти разнесены физически. Также стоимость ПО, распределяющее вычисления, очень высока.

1.1.4. Методы синхронизации в параллельных программах

В параллельных программах разработчик часто сталкивается с проблемой синхронизации между потоками. Как правило, проблемы возникают при доступе к памяти и одновременном выполнении каких-то критических участков кода – критических секций.

Критической областью называют секцию программы, которая должна выполняться с исключительным правом доступа к разделяемым данным, на которые имеются ссылки в этой программе. Процесс, готовящийся войти в критическую область, может быть задержан, если любой другой процесс в это время выполняется в подобной критической области.

В этом разделе будут подробно рассмотрены механизмы синхронизации потоков на программном уровне.

Существуют следующие методы решения проблем синхронизации потоков:

- **Атомарные операции** – операции, которые выполняются целиком или не выполняются вовсе. Например, транзакция к БД является атомарной операцией. Когда два потока пытаются инкрементировать одну и ту же ячейку памяти несинхронизированно, значение может увеличиться на два, а может и на один в зависимости от поведения потоков, так как операция инкрементации представляет собой как минимум три ассемблерные инструкции. Чтобы избежать этого, стоит объявлять тип данных атомарным (если таковой есть в данном языке программирования/библиотеке). Частным случаем атомарных операций являются read-modify- write операции: compare-and-swap, test-and-set, fetch-and-add.
- **Семафор** – объект, ограничивающий число потоков, которые могут войти в эту область кода. Как правило, это число задается при инициализации семафора. Затем при захвате семафора новым потоком проверяется количество потоков, уже захвативших семафор. Если максимальное число потоков достигнуто, то новый поток будет ждать, пока какой-то из потоков, вошедших в область кода, освободит его. Часто использование семафоров неоправданно, так как накладные расходы на создание и поддержку семафора большие. Также следует избегать «утечки семафора», ситуации, при которой поток не выходит из семафора при окончании выполнения области кода, если программист забыл освободить ресурс.
- **Reader/writer semaphore** предоставляет потокам права только на чтение или запись, причем во время записи данных одним потоком остальные потоки не имеет доступа к ресурсу. Однако в таких семафорах может быть проблема ресурсного голодания (starvation), при котором, пока потоки будут читать данные, другие потоки не смогут записать данные долгий промежуток времени или наоборот. Частным решением этой проблемы при равном приоритете потоков может быть поочередный доступ потоков в очереди к чтению и записи.

- **Мьютекс** – частный случай семафора, при котором данную область кода может захватывать только один поток. В случае, если мьютекс обслуживает несколько критических секций, только один поток может находиться в любой из критических секций. Часто используется при организации управления критическими секциями, так как «легче» классического семафора (достаточно хранить одну булеву переменную вместо счетчика), но в отличии от него, предполагается, что один и тот же поток будет захватывать и освобождать мьютекс. Следует отметить, что в стандарте языка C++11, кроме стандартного мьютекса, существуют разные его модификации: recursive_mutex – мьютекс, допускающий повторный вход в критическую секцию этим же потоком, timed_mutex – мьютекс с таймером захвата и recursive_timed_mutex, совмещающий достоинства обеих версий.
- **Spinlock** (циклическая блокировка) – блокировка, при которой поток в цикле ожидает освобождения ресурса. Не всегда является оптимальным решением, так как ожидающий поток работает во время ожидания. Внутри секции кода необходимо избегать прерываний исполнения потока, чтобы избежать deadlock'а.
- **Seqlock** (последовательная блокировка) – механизм синхронизации, предназначенный для быстрой записи переменной несколькими потоками. В ядре Linux работает следующим образом: поток ждет, пока критическая секция освободится(spinlock); при входе в секцию инкрементируется счетчик, поток делает свою работу. При выходе из секции поток проверяет значение счетчика. Если значение счетчика не изменилось, значит, в данный момент никто не записывал данные, и поток выходит из критической секции, иначе он считывает значение переменной заново.
- **Knuth–Bendix completion algorithm** – одним из решений проблем синхронизации является алгоритм Кнута-Бендикса. С его помощью можно перейти от последовательной программы к каскадной. Однако не для всех программ этот алгоритм работает, иногда он может уйти в бесконечный цикл или завершиться с ошибкой.
- **Barrier (барьер) в OpenMP** – участок кода, в котором синхронизируется состояние потоков (не путать с барьером памяти). Например, если для функции в главном потоке требуется, чтобы все дочерние потоки закончили свою работу, можно поставить барьер перед ней. Тогда она будет ждать завершения работы дочерних потоков, после чего все потоки продолжат свою работу. Примером реализации барьера может быть критическая секция, код которой разрешается выполняться только последнему потоку, запросившему выполнение. Остальные потоки должны ожидать его. Для этого необходимо знать, сколько потоков должно прийти в барьер.
- **Неблокирующие алгоритмы.** Часто бывает полезно не использовать стандартные приемы блокировки, а сделать алгоритм неблокирующим. В таком случае программист должен самостоятельно гарантировать, что критические секции кода не будут выполняться одновременно и целостность раз-

деляемой памяти. Также плюсом таких алгоритмов является безопасная обработка прерываний. Для реализации таких алгоритмов часто используются другие технологии синхронизации: read-modify- write, CAS и другие.

- **RCU (read-copy-update)** – алгоритм, позволяющий потокам эффективно считывать данные, оставляя обновление данных на конец работы алгоритма, гарантируя при этом релевантные данные. Только один поток может писать данные, но читать данные могут сразу несколько потоков. Достигается это, например, путем атомарной подмены указателя (CAS). Старые версии данных хранятся для прошлых обращений, пока на них есть хотя бы один указатель. Существуют более новые инструменты для замены указателя: отдельная взаимная блокировка для писателей или механизм membarrier, использующийся в последних версия Linux. RCU может быть полезен при организации структур данных без явных блокировок.
- **Монитор** – объект, инкапсулирующий в себе мьютекс и служебные переменные для обеспечение безопасного доступа к методу или переменной несколькими потоками. Характеризует монитор то, что в один момент только один поток может выполнять любой из его методов. Например, если у нас существует класс (в терминах C++) Account имеющий методы add_money(), sub_money(), то имеет смысл сделать его монитором, чтобы не было конфликтов при проведении операций с аккаунтом.

Однако необязательно организовывать параллельные вычисления, используя синхронизации или блокировки. Некоторые технологии предлагают альтернативный подход к параллельным вычислениям:

- **Программная транзакционная память** – модель памяти, в которой операции, производимые над ячейками памяти атомарны. Плюсы использования: простота использования (заключения блоков кода в блок транзакции), отсутствие блокировок, однако при неправильном использовании возможно падение производительности, а также невозможность использования операций, которые нельзя отменить внутри блока транзакции. В компиляторе GCC поддерживается с версии 4.7 следующим образом:

1. __transaction_atomic {...} – указание, что блок кода – транзакция;
2. __transaction_relaxed {...} – указание, что небезопасный код внутри блока не приводит к побочным эффектам (не поддерживается в версиях 9.x и выше);
3. __transaction_cancel – явная отмена транзакции;
4. attribute((transaction_safe)) – указание транзакционно-безопасной функции;
5. attribute((transaction_pure)) – указание функции без побочных эффектов.

Модель акторов – математическая модель параллельных вычислений, в которой программа представляет собой набор объектов-акторов, которые взаимодействуют между собой и могут создавать новых акторов, отправлять и посылать сообщения друг другу. Предполагается параллелизм вычислений внутри одного актора. Каждый актор имеет адрес, на который

можно отправить сообщение. Каждый актор работает в отдельном потоке. Модель акторов используется для организации электронной почты, некоторых веб-сервисов SOAP и тд.

Несмотря на большое число методов синхронизации, чаще всего надо исходить из решаемой задачи. Например, если мы хотим сделать общую инкрементируемую целочисленную переменную для нескольких потоков, нет смысла создавать mutex или semaphore, более оптимально сделать переменную атомарной. Всегда надо учитывать накладные расходы на создание блокировок и время разработки.

1.1.5. Автоматическое распараллеливание программ

Параллельное программирование – достаточно сложный ручной процесс, поэтому кажется очевидной необходимость его автоматизировать с помощью компилятора. Такие попытки делаются, однако эффективность автораспараллеливания пока что оставляет желать лучшего, т.к. хорошие показатели параллельного ускорения достигаются лишь для ограниченного набора простых for-циклов, в которых отсутствуют зависимости по данным между итерациями и при этом количество итераций не может измениться после начала цикла. Но даже если два указанных условия в некотором for-цикле выполняются, но он имеет сложную неочевидную структуру, то его распараллеливание производиться не будет. Виды автоматического распараллеливания:

- Полностью автоматический: участие программиста не требуется, все действия выполняет компилятор.
- Полуавтоматический: программист даёт указания компилятору в виде специальных ключей, которые позволяют регулировать некоторые аспекты распараллеливания.

Слабые стороны автоматического распараллеливания:

- Возможно ошибочное изменение логики программы.
- Возможно понижение скорости вместо повышения.
- Отсутствие гибкости ручного распараллеливания.
- Эффективно распараллеливаются только циклы.
- Невозможность распараллелить программы со сложным алгоритмом работы.

1.1.6. Основные подходы к распараллеливанию

На практике сложилось достаточное большое количество шаблонов параллельного программирования. Однако все эти шаблоны в своей основе используют три базовых подхода к распараллеливанию:

- **Распараллеливание по данным**: Программист находит в программе массив данных, элементы которого программа последовательно обрабатывает в некоторой функции func. Затем программист пытается разбить этот массив данных на блоки, которые могут быть обработаны в func неза-

висимо друг от друга. Затем программист запускает сразу несколько потоков, каждый из которых выполняет func, но при этом обрабатывает в этой функции отличные от других потоков блоки данных.

- **Распараллеливание по инструкциям**: Программист находит в программе последовательно вызываемые функции, процесс работы которых не влияет друг на друга (такие функции не изменяют общие глобальные переменные, а результаты одной не используются в работе другой). Затем эти функции программист запускает в параллельных потоках.
- **Распараллеливание по информационным потокам**: Программа представляет собой набор выполняемых функций, причем несколько функций могут ожидать результата выполнения предыдущих. В таком случае каждое ядро выполняет ту функцию, данные для которой уже готовы. Рассмотрим этот метод на примере абстрактного двухъядерного процессора как наиболее сложный для понимания. Структурный алгоритм, изображенный на рисунке 1.1.6.1, состоит из 9 функций, некоторые из которых используют результат предыдущей функции в своей работе. Будем считать, что функция 3 использует результат работы функции 1, а функция 7 - результат функций 4 и 6 и т.д., а также функция 5 выполняется по времени примерно столько же, сколько функции 7, 8 и 9, вместе взятые. Тогда на двухъядерной машине этот способ распараллеливания будет оптимальным решением.

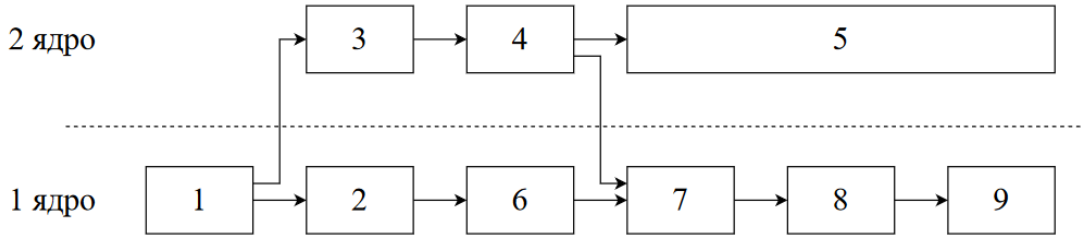

Рис. 1.1.6.1 – Пример работы структурного алгоритма на двухъядерном процессоре

Три описанных метода легче понять на аналогии из обыденной жизни. Пусть два студента получили в стройотряде задание подмести улицу и покрасить забор. Если студенты решат использовать распараллеливание по данным, он будут сначала вместе подметать улицу, а затем вместе же красить забор. Если они решат использовать распараллеливание по инструкциям, то один студент полностью подметёт улицу, а другой покрасит в это время весь забор. Распараллелить по информационным потокам эту ситуацию не получится, так как эти два действия никак не зависят друг от друга. Если предположить, что им обоим нужны инструменты для работы, то один из них должен сначала сходить за ними, а потом они оба начнут делать свою работу.

В большем числе случаев решение об использовании метода является очевидным в силу внутренних особенностей распараллеливаемой программы. Выбор метода определяется тем, какой из них более равномерно

загружает потоки. В идеале все потоки должны приблизительно одновременно заканчивать выделенную им работу, чтобы оптимально загрузить ядра (процессоры) и чтобы закончившие работу потоки не простаивали в ожидании завершения работы соседними потоками, и чтобы закончившие работу потоки не простаивали в ожидании завершения работы соседними потоками.

Контрольные вопросы

1. В чем заключается основное различие между SMP- и MPP-системами?
2. Что такое мьютексы и семафоры?
3. Перечислите основные подходы к распараллеливанию.

1.2. Показатели эффективности параллельной программы

1.2.1. Параллельное ускорение и параллельная эффективность

Для оценки эффективности параллельной программы принято сравнивать показатели скорости исполнения этой программы при её запуске на нескольких идентичных вычислительных системах, которые различаются только количеством центральных процессоров (или ядер). На практике, однако, редко используют для этой цели несколько независимых аппаратных платформ, т.к. обеспечить их полную идентичность по всем параметрам достаточно сложно. Вместо этого измерения проводятся на одной многопроцессорной (многоядерной) вычислительной системе, в которой искусственно ограничивается количество процессоров (ядер), задействованных в вычислениях. Это обычно достигается одним из следующих способов:

- Установка аффинности процессоров (ядер).
- Виртуализация процессоров (ядер).
- Управление количеством потоков выполнения.

Установка аффинности. Под аффинностью (processor affinity/pinning) понимается указание операционной системе запускать указанный поток/процесс на явно заданном процессоре (ядре). Установить аффинность можно либо с помощью специального системного вызова изнутри самой параллельной программы, либо некоторым образом извне параллельной программы (например, средствами «Диспетчера задач» или с помощью команды «start» с ключом «/AFFINITY» в ОС MS Windows, или команды «taskset» в ОС Linux). Недостатки этого метода:

- Необходимость модифицировать исследуемую параллельную программу (при использовании системного вызова изнутри самой программы).
- Невозможность управлять аффиностью на уровне потоков, т.к. обычно ОС позволяет устанавливать аффинность только для процессов (при установке аффиности внешними по отношению к параллельной программе средствами).

Виртуализация процессоров (**ядер**). При создании виртуальной ЭВМ в большинстве специализированных программ (например, VMWare, Virtual- Box) есть возможность «выделить» создаваемой виртуальной машине не все присутствующие в хост-системе процессоры (ядра), а только часть из них. Это можно использовать для имитации тестового окружения с заданным количеством ядер (процессоров).

Недостатком описанного подхода являются накладные расходы виртуализации, которые непредсказуемым образом могут сказаться на результатах экспериментального измерения производительности параллельной программы. Достоинством виртуализации (по сравнению с управляемой аффинностью) является более естественное поведение тестируемой программы при использовании доступных процессоров, т.к. ОС не даётся жёстких указаний, что те или иные потоки всегда должны быть «привязаны» к заранее заданным процессорам (ядрам) – эта особенность позволяет более точно воспроизвести сценарий потенциального «живого» использования тестируемой программы, что повышает достоверность получаемых замеров производительности.

Управление числом потоков. При создании параллельных программ достаточно часто число создаваемых в процессе работы программы потоков не задаётся в виде жёстко фиксированной величины. Напротив, оно является гибко конфигурируемой величиной p, выбор значения которой позволяет оптимальным образом использовать вычислительные ресурсы той аппаратной платформы, на которой запускается программа. Это позволяет программе «адаптироваться» под то число процессоров (ядер), которое есть в наличии на конкретной ЭВМ.

Эту особенность параллельной программы можно использовать для экспериментального измерения её показателей эффективности, для чего параллельную программу запускают при значениях $p = 1, 2, \ldots, n$, где n – это число доступных процессоров (ядер) на используемой для тестирования многопроцессорной аппаратной платформе. Описанный подход позволяет искусственно ограничить число используемых при работе программы процессоров (ядер), т.к. в любой момент времени параллельная программа может исполняться не более чем на p вычислителях. Анализируя измерения скорости работы программы, полученные для различных p, можно рассчитать значения некоторых показателей эффективности распараллеливания.

Параллельное ускорение (parallel speedup). В отличие от применяемого в физике понятия величины ускорения как прироста скорости в единицу времени, в программировании под параллельным ускорением понимают безразмерную величину, отражающую прирост скорости выполнения параллельной программы на заданном числе процессоров по сравнению с однопроцессорной системой, т.е.

$$S(p) = \frac{V(p)}{V(1)}, \qquad (1)$$

где $V(p)$ – средняя скорость выполнения программы на p процессорах (ядрах), выраженная в условных единицах работы в секунду (УЕР/с). Примерами УЕР могут быть количество просуммированных элементов матрицы, количество обработанных фильтром точек изображения, количество записанных в файл байт и т.п. Считается, что значение $S(p)$ никогда не может превысить p, что на интуитивном уровне звучит правдоподобно, ведь при увеличении количества работников, например, в четыре раза невозможно добиться выполнения работы в пять раз быстрее. Однако, как мы рассмотрим ниже, в экспериментах вполне может наблюдаться сверхлинейное параллельное ускорение при увеличении количества процессоров. Конечно, такой результат чаще всего означает ошибку экспериментатора, однако существуют ситуации, когда этот результат можно объяснить тем, что при увеличении количества процессоров означает ошибку экспериментатора, однако существуют ситуации, когда этот результат можно объяснить тем, что при увеличении количества процессоров не только кратно увеличивается их вычислительный ресурс, но так же кратно увеличивается объём кэш-памяти первого уровня, что позволяет в некоторых задачах существенно повысить процент кэш-попаданий и, как следствие, сократить время решения задачи.

Параллельная эффективность (parallel efficiency). Хотя величина параллельного ускорения является безразмерной, её анализ не всегда возможен без информации о значении p. Например, пусть в некотором эксперименте оказалось, что $S(p)=10$. Не зная значение p, мы лишь можешь сказать, что при параллельном выполнение программа стала работать в 10 раз быстрее. Однако если при этом $p=1000$, это ускорение нельзя считать хорошим достижением, т.к. в других условиях можно было добиться почти 1000-кратного прироста скорости работы и не тратить столь внушительные ресурсы на плохо распараллеливаемую задачу. Напротив, при значении $p=11$ можно было бы счесть величину $S(p)=10$ вполне приемлемой.

Эта проблема привела к необходимости определить ещё один показатель эффективности параллельной программы, который бы позволил получить некоторую оценку эффективности распараллеливания с учётом количества процессоров (ядер). Этой величиной является **параллельная эффективность**

$$E(p)=\frac{S(p)}{p}=\frac{V(p)}{p\cdot V(1)} \tag{2}$$

Среднюю скорость выполнения программы $V(p)$ можно измерить следующими двумя неэквивалентными методами:

- **Метод Амдала**: рассчитать $V(p)$, зафиксировав объём выполняемой работы (при этом изменяется время выполнения программы для различных p).
- **Метод Густавсона-Барсиса**: рассчитать $V(p)$, зафиксировав время работы тестовой программы (при этом изменяется количество выполненной работы для различных p).

1.2.2. Метод Амдала

При оценке эффективности распараллеливания некоторой программы, можно выразить следующим образом: $V(p)|_{w=const} = \frac{w}{t(p)}$, где w – это общее количество УЕР, содержащихся в рассматриваемой программе, $t(p)$ – время выполнения работы w при использовании p процессоров. Тогда выражение для параллельного ускорения примет вид:

$$S(p)|_{w=const} = \frac{V(p)}{V(1)} = \frac{w}{t(p)} = \frac{w}{t(1)} = \frac{t(1)}{t(p)} \qquad (3)$$

Запишем время $t(1)$ следующим образом:

$$t(1) = t(1) + (k \cdot t(1) - k \cdot t(1)) = k \cdot t(1) + (1-k) \cdot t(1), \qquad (4)$$

где $k \in [0,1)$ – это коэффициент распараллеленности программы, которым мы обозначим долю времени, в течение которого выполняется идеально распараллеленный код внутри рассматриваемой программы. Такой код можно выполнить ровно в p раз быстрее, если количество процессоров увеличить в p раз. Заметим, что коэффициент k никогда не равен единице, т.к. в любой программе всегда присутствует нераспараллеливаемый код, который приходится выполнять последовательно на одном процессоре (ядре), даже если их доступно несколько. Если для некоторой программы $k = 0$, то при запуске этой программы на любом количестве процессоров p она будет решаться за одинаковое время.

Учитывая, что в методе Амдала количество работы остаётся неизменным при любом p (т.к. $w = const$), можно утверждать, что значение k не изменяется в проводимых экспериментах, следовательно можем записать:

$$t(p) = \frac{k \cdot t(1)}{p} + (1-k) \cdot t(1), \qquad (5)$$

где первое слагаемое даёт время работы распараллеленного в p раз идеально распараллеливаемого кода, а второе слагаемое – время работы нераспараллеленного кода, которое не меняется при любом p. Подставив формулу (5) в (3), получим выражение

$$S(p)|_{w=const} = \frac{t(1)}{t(p)} = \frac{t(1)}{\frac{k \cdot t(1)}{p} + (1-k) \cdot t(1)} = \frac{1}{\frac{k}{p} + 1 - k}, \qquad (6)$$

которое перепишем в виде

$$S(p)|_{w=const} = S_A(p) = \left(\frac{k}{p} + 1 - k \right)^{-1}, \qquad (7)$$

более известном как закон Амдала – по имени американского учёного Джина Амдала, предложившего это выражение в 1967 году. До сих пор

в специализированной литературе по параллельным вычислениям именно этот закон является основополагающим, т.к. позволяет получить теоретическое ограничение сверху для скорости выполнения некоторой заданной программы при распараллеливании.

Отметим, что выражение для расчёта параллельной эффективности при использовании метода Амдала можно получить, объединив формулы (2) и (7), а именно:

$$E_A(p) = (k + p - p \cdot k)^{-1}. \quad (8)$$

Важным допущением закона Амдала является идеализация физического смысла величины k, состоящая в предположении, что идеально распараллеленный код будет давать линейный прирост скорости работы при изменении p от 0 до $+\infty$. При решении реальных задач приходится ограничивать этот интервал сверху некоторым конечным положительным значением $pmax$ и/или исключать из этого интервала все значения, не кратные некоторой величине, обычно задающей размерность задачи.

Например, код программы, выполняющей конволюционное кодирование независимо для пяти равноразмерных файлов, может давать линейное ускорение при изменении p от 1 до 5, но уже при $p = 6$, скорее всего, покажет нулевой прирост скорости выполнения задачи (по сравнению с решением при $p = 5$). Это объясняется тем, что конволюционное кодирование, также известное как «свёрточное», является принципиально нераспараллеливаемым при кодировании выбранного блока данных.

1.2.3. Метод Густавсона–Барсиса

При оценке эффективности распараллеливания некоторой программы, работающей фиксированное время, скорость выполнения можно выразить следующим образом:

$V(p)|_{t=const} = \frac{w(p)}{t}$, где $w(p)$ – это общее количество УЕР, которые программа успевает выполнить за время t при использовании p процессоров. Тогда выражение (1) для параллельного ускорения примет вид:

$$S(p)|_{t=const} = \frac{V(p)}{V(1)} = \frac{w(p)}{t} : \frac{w(1)}{t} = \frac{w(p)}{w(1)}. \quad (9)$$

Запишем количество работы $w(1)$ следующим образом:

$$w(1) = w(1) + (k \cdot w(1) - k \cdot w(1)) = k \cdot w(1) + (1 - k) \cdot w(1), \quad (10)$$

где $k \in [0,1)$ – это уже упомянутый ранее коэффициент распараллеленности программы. Тогда первое слагаемое можно считать количеством работы, которая идеально распараллеливается, а второе – количество работы, которую распараллелить не удается при добавлении процессоров (ядер).

При использовании p процессоров количество выполненной работы $w(p)$ очевидно станет больше, при этом оно будет состоять из двух слагаемых:

- количество не распараллеленных условных единиц работы $(1-k)\cdot w(1)$, которое не изменится по сравнению с формулой (10).
- количество распараллеленных УЕР, объём которых увеличиться в p раз по сравнению с формулой (10), т.к. в работе будет задействовано p процессоров вместо одного.

Учитывая сказанное, получим следующее выражение для $w(p)$:

$w(p) = p\cdot k\cdot w(1) + (1-k)\cdot w(1)$, тогда с учётом формулы (9) получим:

$$\frac{w(p)}{w(1)} = \frac{p\cdot k\cdot w(1) + (1-k)\cdot w(1)}{w(1)}$$, что позволяет записать:

$$S(p)\big|_{t=const} = S_{GB}(p) = p\cdot k + 1 - k. \quad (11)$$

Приведённое выражение называется законом Густавсона–Барсиса, который Джон Густавсон и Эдвин Барсис сформулировали в 1988 году.

Контрольные вопросы

1. Какие показатели эффективности параллельной программы можно сравнивать при запуске на различных вычислительных системах?
2. Какими способами можно искусственно ограничить количество процессоров (ядер), задействованных в вычислениях?
3. Что такое виртуализация процессоров (ядер)?
4. Какие недостатки связаны с виртуализацией процессоров?
5. Чем отличается виртуализация от управляемой аффинности?

1.3. Практические аспекты параллельного программирования

1.3.1. Отладка параллельных программ

Средства отладки параллельных программ встроены в большинство популярных интегрированных сред разработки (IDE), например: Visual Studio, Eclipse CDT, Intel Parallel Studio и т.п. Эти средства включают в себя удобную визуализацию временных диаграмм исполнения потоков, автоматический поиск подозрительных участков программы, в которых могут наблюдаться гонки данных и взаимоблокировки.

Несмотря на эффективность существующих инструментов отладки, при работе в дебаггере (debugger) с параллельной программой возникают существенные затруднения, т.к. для своего корректного функционирования отладчик добавляет в машинный код исходной параллельной программы дополнительные инструкции, которые изменяют временную диаграмму выполнения потоков по отношению друг к другу. Это может приводить

к ситуациям, когда при тестировании программы в отладчике не наблюдаются гонки данных и взаимоблокировки, которые при запуске Release-версии программы проявятся в полной мере.

Также при отладке многопоточной программы следует иметь в виду, что её поведение (как при штатной работе, так и при отладке) может существенным образом различаться при использовании одноядерного и многоядерного процессора. При запуске нескольких потоков на одноядерной машине они будут выполняться в режиме разделения времени, т.е. последовательно. Значит, в этом случае не будут наблюдаться многие проблемы с совместным доступом к памяти и обеспечением когерентности кэшей, присущие многоядерным системам. Кроме того, при отладке программы на одноядерной системе программист может использовать неявные приёмы обеспечения последовательности выполнения операций.

Например, программист может некорректно предполагать, что при выполнении высокоприоритетного потока низкоприоритетный поток не может завладеть процессором. Это предположение корректно только в одноядерной системе, ведь при наличии нескольких ядер и малом количестве высокоприоритетных потоков вполне может наблюдаться ситуация, когда низкоприоритетный поток завладеет одним из ядер, при одновременной работе высокоприоритетного потока на соседнем ядре.

1.3.2. Менеджеры управления памятью для параллельных программ

При вызове функций malloc/free в однопоточной программе не возникает проблем даже при довольно высокой интенсивности вызовов одной из них. Однако в параллельных программах эти функции могут стать узким местом, т.к. при их одновременном использовании из нескольких потоков происходит блокировка общего ресурса (менеджера управления памятью), что может привести к существенной деградации скорости работы многопоточной программы.

Получается, что несмотря на формальную потокобезопасность стандартных функций работы с памятью, они могут стать потоконеэффективными при очень интенсивной работе с памятью нескольких параллельно работающих потоков.

Для решения этой проблемы существует ряд сторонних программ, называющихся «Менеджер управления памятью (МУП)» (Memory Allocator), как платных, так и бесплатных с открытым исходным кодом. Каждое из них обладает своими достоинствами и недостатками, которые следует учитывать при выборе. Перечислим наиболее распространённые МУП с указанием ссылок на официальные сайты:

- tcmalloc: http://goog-perftools.sourceforge.net/doc/tcmalloc.html
- ptmalloc: http://www.malloc.de/malloc/ptmalloc3-current.tar.gz

- dmalloc: http://dmalloc.com
- HOARD: http://www.hoard.org
- nedmalloc: http://www.nedprod.com/programs/portable/nedmalloc
- jemalloc: http://jemalloc.net
- mimalloc: https://github.com/microsoft/mimalloc

Перечисленные МУП разработаны таким образом, что ими можно «незаметно» для параллельной программы подменить стандартные МУП библиотеки libc языка C. Это значит, что выбор конкретного МУП никак не влияет на исходный код программы, поэтому общая практика использования сторонних МУП такова: параллельная программа изначально создаётся с использованием МУП libc, затем проводится профилирование работающей программы, затем при обнаружении узкого места (bottleneck) в функциях malloc/free принимается решение заменить стандартный МУП одним из перечисленных.

Также стоит отметить, что некоторые технологии распараллеливания (например, Intel TBB) уже имеют в своём составе специализированный МУП, оптимизированный для выполнения в многопоточном режиме.

1.3.3. Библиотека Intel IPP

Оптимизация типовых задач обработки данных. Существует немногочисленное количество высокопроизводительных библиотек, состоящих из набора низкоуровневых API для обработки данных: изображений, сигналов, матриц. Одной из таких библиотек является «Intel IPP»1, реализующая следующие функции:

- Кодирование и декодирование видео.
- Кодирование и декодирование аудио.
- Компьютерное зрение.
- Криптография.
- Сжатие данных.
- Преобразование цвета.
- Обработка изображения.
- Трассировка луча/визуализация.
- Обработка сигналов.
- Кодирование речи.
- Распознавание речи.
- Обработка строк.
- Векторная/матричная математика.

Для использования функций данной библиотеки необходимо в исходном коде подключить заголовочный файл IPP:

```
#include <ipp.h>
```

Рассмотрим пример программы, которая вычисляет модуль синуса каждого элемента массива:

```
for (int i=0; i<N; i++){
array[i] = abs(sin(array[i]));
}
```

Теперь воспользуемся функциями IPP, тогда наша программа будет выглядеть так:

```
ippsSin_64f_A21(array, array, N);
ippsAbs_64f_A21(array, array, N);
```

Благодаря использованию данных функция, программа стала компактнее и быстрее.

1.3.4. Технология OpenMP

Краткая характеристика технологии. Первая версия стандарта OpenMP появилась в 1997 году при поддержке крупнейших IT-компаний мира (Intel, IBM, AMD, HP, Nvidia и др.). Целью нового стандарта было предложить кроссплатформенный инструмент для распараллеливания, который был бы более высокоуровневый, чем API управления потоками, предлагаемые операционной системой. На данный момент OpenMP стандартизована для трёх языков программирования: C, C++ и Фортран.

Поддержка компиляторами. Абсолютное большинство существующих современных компиляторов C/C++ поддерживают OpenMP версии 2.0 (например, как gcc, так и Visual Studio). Однако не все компиляторы поддерживают последнюю версию OpenMP 5.2, поэтому далее при изложении материала будет в качестве «общего знаменателя» использоваться технология OpenMP 2.0.

OpenMP определяет набор директив препроцессору, которые дают указание компилятору заменить следующий за ними исходный код на его параллельную версию с помощью доступных компилятору средств, например с помощью POSIX Threads в Linux или Windows Threads в операционных системах Microsoft. Для корректной трансляции директив необходимо при компиляции указать специальный ключ, значение которого зависит от компилятора (примеры приведены в таблице 1.3.3.1).

Таблица 1.3.3.1 – Ключи компиляторов для запуска OpenMP

Название компилятора	**Ключ компилятору для включения OpenMP**
gcc	-fopenmp
icc (Intel C/C++ compiler)	-qopenmp
Sun C/C++ compiler	-xopenmp
Visual Studio C/C++ compiler	/openmp
PGI (Nvidia C/C++ compiler)	-mp

Помимо препроцессорных директив, OpenMP определяет набор библиотечных функций, для вызова которых в исходном коде потребуется подключить заголовочный файл OpenMP:

```
#include <omp.h>
```

Отличительные особенности. Среди прочих технологий распараллеливания OpenMP выделяется следующими важными и характеристиками:

- Инкрементное распараллеливание.
- Обратная совместимость.
- Высокий уровень абстракций.
- Низкий коэффициент трансформации.
- Поддержка крупнейшими IT-гигантами.
- Автоматическое масштабирование.

Инкрементное распараллеливание. OpenMP предлагает инкрементное распараллеливание, позволяя изменять последовательную программу постепенно, без необходимости сразу переписывать всю структуру. Это уникально, так как многие другие технологии требуют существенных изменений с самого начала.

Обратная совместимость. Большинство программных технологий развиваются с обеспечением обратной совместимости (backward compatibility), когда более новая версия программы поддерживает работоспособность старых файлов. Термин *«прямая совместимость»* (forward compatibility) имеет противоположный смысл: файлы, созданные в программе новой версии, остаются работоспособными при использовании старой версии программы. В случае OpenMP это проявляется в том, что распараллеленная программа будет корректно скомпилирована в однопоточном режиме даже на старом компиляторе, который не поддерживает OpenMP. Важно отметить, что прямая совместимость обеспечивается, если при распараллеливании не используются библиотечные функции OpenMP, а присутствуют только препроцессорные директивы. При наличии библиотечных функций для обеспечения обратной совместимости потребуется написать функции-заглушки в файле «omp.h» (некоторые компиляторы умеют генерировать эти заглушки при использовании специального ключа).

Высокий уровень абстракций. OpenMP предоставляет высокий уровень абстракций, что упрощает параллельное программирование, но не позволяет детально настраивать некоторые аспекты работы с потоками.

Низкий коэффициент параллельной трансформации (КПТ). При распараллеливании существующей последовательной программы приходится вносить в неё достаточно большое число изменений. Пусть КПТ – это отношение строк нового программного кода, который добавился в результате распараллеливания, к общему числу строк кода в программе. В OpenMP КПТ обычно существенно ниже, чем у большинства других технологий распараллеливания. Это объясняется высоким уровнем абстракции языка OpenMP (см. предыдущий пункт). Поддержка крупнейшими IT-гигантами. Уже при разработке

OpenMP о его поддержке заявили крупнейшие игроки IT-мира. Это обеспечило не только высокое качество разработки стандарта, но и наличие готовых реализаций стандарта в популярных компиляторах. Несмотря на прошедшие два десятка лет, OpenMP не растерял приверженцев, и поддержка новейших версий OpenMP с достаточно малой задержкой появляется в компиляторах. Например, при текущей версии стандарта OpenMP 5.2 наиболее популярные компиляторы уже поддерживают версию OpenMP 4.0. Исключением является только компания Microsoft. Их компилятор вот уже несколько версий неизменно поддерживает только OpenMP 2.0.

Автоматическое масштабирование. Низкоуровневые технологии распараллеливания (POSIX Threads, OpenCL) предлагают программисту вручную управлять числом создаваемых потоков при выполнении параллельной работы. Это обеспечивает возможность гибко управлять и настраивать процесс создания потоков в зависимости от числа доступных системе процессоров (ядер), но при этом требует от программиста большое количество неавтоматизируемой работы. В OpenMP управление масштабированием происходит в автоматическом режиме, т.е. OpenMP сам запрашивает у операционной системы число доступных процессоров и выбирает число создаваемых потоков. Но при необходимости OpenMP оставляет возможность устанавливать требуемое число потоков вручную.

Примеры OpenMP-программ. Рассмотрим ниже простейшие примеры работающих параллельных программ, начиная с традиционного для программирования примера «Hello, World»:

```
#pragma omp parallel
printf("Hello, world!");
```

Результатом работы будет выведенное несколько раз в консоль сообщение. Число сообщений определяется числом логических процессоров, доступных системе (например, при использовании технологии HyperThreading при двух ядрах число логических процессоров будет равно четырём).

Действие директивы pragma распространяется на следующий за ней исполняемый блок. В данном случае это вызов функции printf, но можно было бы заключить произвольное число операций в фигурные скобки, чтобы расширить исполняемый блок:

```
int i = 1;
#pragma omp parallel
{
printf("Hello, world!");
#pragma omp atomic
i++;
}
```

В этой программе заключенный в фигурные скобки блок операций выполняется одновременно на нескольких ядрах. При этом в строке 5 процессору даётся указание выполнить операцию «i++» атомарно, т.е. не параллельно, а последовательно каждым из потоков.

С одной стороны, это приводит к тому, что операция инкремента перестаёт быть распараллеленной, что снижает скорость многоядерного выполнения. С другой стороны, директива atomic в данном случае необходима, т.к. иначе могла бы возникнуть сложно обнаружимая проблема с гонкой данных, проявляющаяся в конфликте при записи данных в общую область памяти одновременно несколькими потоками в переменную i. Заметим, что директива atomic может применяться только для однострочных простых команд присваивания.

Для изоляции более сложных составных команд с возможным вызовом пользовательских и системных функций следует использовать директиву critical, которая допускает (в отличие от директивы atomic) возможность расширения своей области действия на блок операций, заключённый в фигурные скобки, при этом каждая critical-секция может иметь имя, позволяющее сгруппировать разные критические секции по этому имени, чтобы предотвратить появление единой распределённой по всей программе критической секции:

```
int i = 1;
#pragma omp parallel
{
printf("Hello, world!");
#pragma omp critical
{
i++;
printf("i=%d\n", i);
}
}
```

В этом случае функция printf в строке 4 выполняется всеми потоками параллельно, что может привести к перемешиванию выводимых символов.

Напротив, функция printf в строке 8 выполняется потоками строго по очереди, что предотвращает возможные конфликты между ними, однако замедляет выполнение программы из-за искусственного ограничения коэффициента распараллеленности.

Приведём пример распараллеливания программы, содержащей последовательный вызов функций run_function1 и run_function2, которые не зависят друг от друга (т.е. не используют общих данных и результаты работы одной не влияют на результаты работы другой) и поэтому допускающих удобное распараллеливание по инструкциям в чистом виде:

```
#pragma omp parallel sections
{
#pragma omp section
run_function1();
#pragma omp section
run_function2();
}
```

Рассмотрим пример распараллеливания цикла с использованием OpenMP. Пусть в каждую ячейку одномерного массива нужно записать индекс этой ячейки, возведённый в шестую степень:

```
int i; int a[10];
#pragma omp parallel for
for (i = 0; i < 10; ++i) {
a[i] = i*i*i*i*i*i;
}
```

Пусть указанная программа выполняется на двухъядерном процессоре.

Тогда первый процессор рассчитает значения с a[0] по a[4], второй процессор – значения с a[5] по a[9]. Видимо, что при записи в массив процессору не мешают друг другу, т.к. работают с разными частями массива. Попробуем оптимизировать предыдущий вариант, сократив число операций умножения для возведения в шестую степень:

```
int i, tmp;
#pragma omp parallel for
for (i = 0; i < 10; ++i) {
tmp = i*i*i; /* attemp to optimize */
a[i] = tmp*tmp; /* error */
}
```

В указанном случае программа будет корректно работать только при наличии одного процессора (ядра). При наличии нескольких ядер будет наблюдаться состояние гонки данных при одновременной записи нового значения в переменную tmp (строка 4) несколькими потоками, в результате массив будет заполнен некорректно. Например, пусть первый поток, выполняющий итерацию $i = 2$, записал в tmp число 8. Теперь при вычислении a[2] поток попытается записать число $8 * 8$, однако если до начала строки 5 успеет вклиниться второй поток, работающей с итерацией $i = 7$, то значение tmp превратится в $7 * 7 * 7$, а значение a[2], рассчитываемое первым потоком, превратится в 7^6 ($7 * 7 * 7$ в квадрате), вместо положенных 64. Исправим допущенную ошибку следующим образом:

```
int i, tmp;
#pragma omp parallel for private(tmp)
for (i = 0; i < 10; ++i) {
tmp = i*i*i;
a[i] = tmp*tmp;
}
```

В директиве препроцессору появился новый элемент: private. Этот элемент задаёт через запятую перечень локальных (приватных) для каждого потока переменных. В данном случае такая переменная одна: tmp. Другой равноценный способ исправить ошибку – это перенести объявление переменной «int tmp» внутрь параллельной области, что заставить OpenMP считать эту переменную локальной для каждого потока. Может возникнуть вопрос, почему в перечень локальных переменных не добавлена i. Ответ не очевиден: OpenMP по умолчанию считает переменную распараллеливаемого цикла локальной.

Любая переменная, объявленная внутри параллельной области, считается в OpenMP локальной, поэтому такие переменные не нужно указывать в списке. Любая переменная, объявленная вне этой области, является глобальной (в нашем случае глобальной переменной является указатель на массив a). Но если требуется явным образом указать на глобальность переменной, следует рядом с командой private использовать команду shared(x, ...), где x задаёт список глобальных переменных.

Рассмотрим ещё одну типичную для параллельного программирования ошибку. Следующая программа считает сумму чисел от 1 до 100:

```
int i, sum = 0;
#pragma omp parallel for
for (i = 0; i < 100; ++i) /* error */
sum += i;
```

Переменная sum является глобальной, поэтому при попытке записать в неё новое значение потоки будут мешать друг другу. Чтобы исправить ошибку, нам придётся использовать локальную для каждого потока сумму, а затем потребуется сложить все эти локальные суммы:

```
int i, sum = 0, sum_private = 0;
#pragma omp parallel private (sum_private)
{
sum_private = 0; /* repeated initialization! */
#pragma omp for
for (i = 0; i < 100; ++i)
sum_private += i;
#pragma omp atomic
sum += sum_private;
}
```

Видим начало параллельной области в строке 2 – именно в этом месте OpenMP создаёт несколько потоков. В строке 6 новые потоки не создаются (т.к. отсутствует ключевое слово parallel), но входящие в цикл потоке делят итерации между собой, а не выполняют каждый все итерации целиком. В строке 8 рассчитавший свою частичную сумму поток пытается прибавить эту сумму к общей сумме. Это приходится делать с помощью директивы atomic, которая гарантирует, что потоки не будут мешать друг другу при перезаписи sum.

Ещё один сложный момент – это повторная инициализация переменной sum_private в строке 4: необходимость в этом возникает, т.к. OpenMP не инициализирует локальные переменные, даже если есть глобальные переменные с идентичными именами. Подобное решение призвано уменьшить накладные расходы на копирование переменных.

Описанный подход является работоспособным, однако он почти не используется на практике, т.к. стандарт OpenMP для целого класса подобных задач предлагает более высокоуровневое и простое решение. Оно состоит в использовании команды reduction:

```
int i, sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i = 0; i < 100; ++i)
sum += i;
```

Команда reduction помечает перечисленный переменные как локальные, а в конце параллельной области все локальные переменные объединяет (агрегирует) в одну глобальную переменную с тем же именем, используя указанную операцию. В нашем случае операцией является суммирование. Но OpenMP допускает вместо знака «+» использовать «*», «-», «/», а в последних версиях и функции, написанные разработчиками. Важно, что reduction, кроме прочего, выполняет инициализацию переменных не значениями исходных глобальных переменных, а наиболее соответствующими логики агрегации значениями: например, при суммировании переменная инициализируется нулём, а при умножении – единицей.

1.3.5. POSIX Threads

В данной библиотеке более 100 разных функций, но всех их можно разделить на 4 основные группы:

- Управление потоками: create, join и т.д.
- Мьютексы.
- Условные переменные.
- Синхронизация между тредами.

Для того, чтобы воспользоваться библиотекой PThreads в Unix-like и POSIX-совместимой операционной системе, достаточно подключить следующий заголовочный файл PThreads:

#include <pthread.h>

В отличие от OpenMP, PThreads является более низкоуровневой библиотекой, где от разработчика требуется заранее продумать всю логику работы потоков.

Рассмотрим задачу добавление числа 10 к каждому элементу массива:

```
for (int i=0; i<N; i++)
array[i] = array[i] + 10;
```

Теперь воспользуемся библиотекой PThreads для распараллеливания данной задачи (для понимания того, что происходит, приведен код всей программы):

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// Struct for thread parameters
typedef struct {
int* array;
int start, end;
} part_of_array;
// Thread function
void* plus_ten(void *input){
part_of_array *data = (part_of_array*) input;
for(int i=data->start; i < data->end; i++){
data->array[i] += 10;
}
}
int main(){
int N=10;
int array[N];
// Init threads
pthread_t thread_1, thread_2;
// Fill array with values
for (int i=0; i < N; i++){
array[i] = i;
}
// Parameters for thread_1 and thread_2
part_of_array pthrFirst = {array, 0, 5};
part_of_array pthrSecond = {array, 5, 10};
// Creating threads
if (pthread_create(&thread_1, NULL, plus_ten, &pthrFirst) == -1){
printf("Поток 1 не создан.");
}
if (pthread_create(&thread_2, NULL, plus_ten, &pthrSecond) == -1){
```

```
printf("Поток 2 не создан.");
}
// Execute threads
pthread_join(thread_1, NULL);
pthread_join(thread_2, NULL);
}
```

Теперь разберем, что происходит в данной программе. Начнем с инициализации тредов:

```
// First option
pthread_t thread_1, thread_2;
// Second option
pthread_t threads[2];
```

В данном коде представлено два варианта того, как можно инициализировать несколько потоков. Это может быть как отдельная переменная, так и массив потоков.

После инициализации необходимо создать поток:

```
if (pthread_create(&thread_1, NULL, plus_ten, &ptrFirst) == -1)
printf("Поток 1 не создан.");
```

Функция pthread_create принимает четыре параметра: указатель на поток, атрибуты потока (если используются атрибуты по умолчанию, то передается NULL), функция которую будет выполнять поток, аргумент функции.

В случае, если поток успешно создан, возвращается 0. Иначе могут быть возвращены следующие значения:

- EAGAIN – у системы нет ресурсов для создания нового потока, или система не может больше создавать потоков, так как число потоков превысило значение PTHREAD_THREADS_MAX.
- EINVAL – неправильные атрибуты потока (переданные аргументом attr).
- EPERM – Вызывающий поток не имеет должных прав для того, чтобы задать нужные параметры или политики планировщика.

Все коды ошибок можно изучить по данной ссылке https://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Errors/unix_system_errors.html.

Аргумент функции должен быть типа void*. Чтобы передать несколько параметров, их необходимо обернуть в структуру. В нашем случае необходимо передать указатель на массив и интервал, на котором необходимо провести вычисления:

```
typedef struct {
int* array;
int start, end;
} part_of_array;
```

Сразу после того, как поток создан, он начинает выполнение. По стандарту выход из функции вызывает функцию pthread_exit, а возвращаемое значение будет передано при вызове pthread_join, как статус.

В свою очередь, функция pthread_join заставляет основной поток ожидать завершения порожденных им потоков.

При успешном завершении потока функция pthread_join возвращает 0, иначе данная функция может вывести следующие ошибки:

- EINVAL – thread указывает на не объединяемый поток.
- ESRCH – не существует потока с таким идентификатором, который хранит переменная thread.
- EDEADLK – был обнаружен дедлок (взаимная блокировка), или же в качестве объединяемого потока указан сам вызывающий поток.

Механизмы синхронизации потоков. Взаимное исключение mutex выполняет функцию ограничения доступа потоков к одному ресурсу. Mutex – переменная, которая может быть или заблокирована, или свободна. При этом, если один поток её заблокировал, другие потоки будут ожидать освобождения ресурса.

```
// Initialize mutex
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *mutexattr);
// Lock resource
int pthread_mutex_lock(pthread_mutex_t *mutex);
// Free resource
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// Delete mutex
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

pthread_mutex_t тип данных описывающий mutex. Атрибуты mutex можно контролировать функцией pthread_mutex_init().

Рассмотрим пример функции, в которой необходима синхронизация:

```
void* sum_of_array(void *input){
int number;
for (int i=0; i<100; i++){
number = *(int *)input;
sum += number;
input += sizeof(int);
}
}
```

Данная программа добавляет значения элементов массива в глобальную переменную sum, при обращении нескольких потоков к данной переменной конечный результат может измениться.

Необходимо добавить mutex в данную функцию, который бы ограничил доступ к данной переменной одновременно нескольким потокам. Тогда программа будет выглядеть следующим образом:

```
void* sum_of_array(void *input){
int number;
for (int i=0; i<100; i++){
number = *(int *)input;
pthread_mutex_lock(&simple_mutex);
sum += number;
pthread_mutex_unlock(&simple_mutex);
input += sizeof(int);
}
}
```

Семафор. Следующим примитивом синхронизации является семафор. Его задача такая же, как и у mutex, главное отличие в том, что mutex захватывает один поток в то время, как семафор может захватывать несколько потоков.

Чтобы воспользоваться семафором, необходимо подключить заголовочный файл:

```
#include <semaphore.h>
```

Важно, что семафор после окончания работы с ним необходимо удалять, как это показано ниже:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

Функция sem_init() принимает следующие параметры: sem – объект который необходимо инициализировать, pshared - (0) данный семафор будет общим для всех потоков, (1) общим для процессов, value - начальное значение семафора.

Контрольные вопросы

1. Чем отличается потокобезопасная функция от реентерабельной?
2. Укажите основные недостатки использования Hyper-Threading.
3. В чем заключается основное различие между распараллеливанием по данным и распараллеливанием по задачам?
4. Для чего используется функция pthread_exit() и можно ли обойтись без неё?
5. Какие атрибуты можно передать в функцию pthread_create()?

Задания для самостоятельного выполнения

1. Реализуйте программу используя OpenMP, которая позволяет пользователю вводить размер массива и количество потоков. Затем распределите вычисление суммы элементов массива между указанным количеством потоков. Каждый поток будет обрабатывать свою часть массива, а затем объедините результаты.

2. Реализуйте программу используя OpenMP, которая позволяет пользователю вводить диапазон чисел и количество потоков. Разделите поиск простых чисел на подзадачи, где каждый поток будет проверять свою часть диапазона на простоту. Объедините результаты, чтобы найти наибольшее простое число в заданном диапазоне.

3. Реализуйте программу используя OpenMP, которая позволяет пользователю вводить число, для которого нужно вычислить факториал. Затем разделите вычисление на подзадачи, где каждый поток будет вычислять факториал для своего диапазона чисел.

4. Написать функцию, которая один раз в секунду выводит в консоль сообщение о текущем проценте завершения работы программы. Указанную функцию необходимо запустить в отдельном потоке, параллельно работающем с основным вычислительным циклом. Нельзя использовать PThreads, сделать только средствами OpenMP.

5. Распараллелить вычисления на этапе Sort, для чего выполнить сортировку в два этапа:

- Отсортировать первую и вторую половину массива в двух независимых нитях (можно использовать OpenMP-директиву «parallel sections»).
- Объединить отсортированные половины в единый массив.

РАЗДЕЛ II. ОБРАБОТКА БОЛЬШИХ ОБЪЕМОВ ДАННЫХ

2.1. Жизненный цикл данных

Жизненный цикл данных – это последовательность этапов, которую конкретная порция данных проходит от начального этапа создания или получения до момента архивации или удаления. Основные этапы жизненного цикла данных представлены на рисунке 2.1.1.1. Рассмотрим эти этапы подробнее.

2.1.1. Создание данных (Data Generation/Data Capture)

На этом этапе данные генерируются или захватываются. Этот этап обычно еще делят на три типа получения данных:

1. Приобретение данных (Data Acquisition) Получение организацией данных, уже сгенерированных вне предприятия.

2. Запись данных (Data Entry) Создание новых данных оператором или компьютером. Данные имеют ценность для предприятия.

3. Регистрация сигналов (Signal Reception)

Захват данных устройствами. Особенно важно в системах управления, но в последнее время особенно ценно при использовании такого подхода, как Интернет вещей.

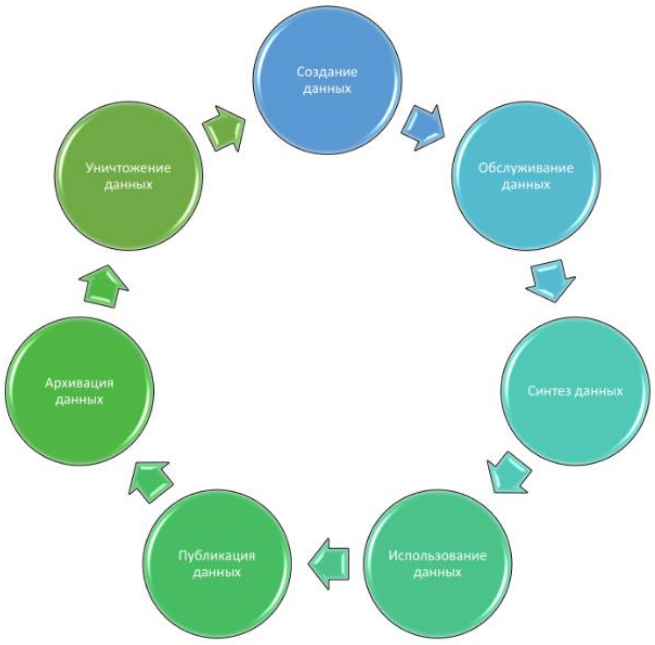

Рис. 2.1.1.1 – Жизненный цикл данных

Все три варианта получения данных очень важны в рамках рассмотрения процесса управления данными.

2.1.2. Обслуживание данных (Data Maintenance)

После того, как данные были созданы, необходимо их хранить и обслуживать. Необходимо осуществлять доставку данных в точку, где их будут использовать или производить над ними манипуляции (например, операции синтеза).

Можно говорить о том, что обслуживание данных – это обработка данных без получения из извлечения из них поле зной информации для предприятия.

Зачастую обслуживание данных включает в себя такие действия с данными, как перемещение, интеграция, очистка, обогащение и ETL-процессы (Extract, Transform, Load).

Обслуживание данных обычно подразумевает применение широкого спектра методов из области управления данными (Data Management).

2.1.3. Синтез данных (Data Synthesis)

Это сравнительно недавно появившаяся стадия в жизненном цикле данных. Используется не во всех моделях жизненных циклах данных.

Синтез данных – это процесс получения дополнительной ценности из данных при помощи использования индуктивной логики и сторонних информационных источников.

Это стадия, на которой с данными работаю аналитики, причем они могут использовать в своей работы методы моделирования рисков, актуарного моделирования, моделирования для принятия инвестиционных решений и др. На этой стадии используется индуктивная логика, не дедуктивная. Индуктивная логика требует использования экспертного мнения, т.к. именно компетенции экспертов необходимы для построения моделей скоринга и др.

2.1.4. Использование данных (Data Usage)

До сих пор шла речь об использовании данных внутри одного предприятия, которые возможно были подвержены очистке и обогащению на стадии обслуживания данных и использовались совместно с дополнительным третьими источниками данных на стадии синтеза данных.На стадии использования данных они применяются в качестве полезной информации для задач, которые должны выполняться и управляться на основе данных.

Эти задачи могут быть вне жизненного цикла данных. Тем не менее, данные становятся все более значимой частью бизнес -процессов предприятий. Данные сами могут быть продуктом или услугой (или быть частью продукта или услуги), предлагаемой предприятием. Использование данных имеет специальные задачи в рамках управления данными (Data Governance). Одна из задач заключается в законном использовании данных в требуемом виде. Это называется «разрешенное использование данных» (permitted use of data). Могут существовать регулирующие или договорные ограничения на то, как фактически можно использовать данные, а часть роли управления данными (Data Governance) заключается в обеспечении соблюдения этих ограничений.

2.1.5. Публикация данных (Data Publication)

При использовании данных возможна ситуация, когда данные отправляются за пределы предприятия. В этом случае говорят о публикации данных. Публикация данных – это вынос данных за пределы предприятия. Примером этого процесса может быть маклер, рассылающий ежемесячные отчеты клиентам. Все данные, которые были разосланы, уже не могут быть отозваны. Если были разосланы данные с неверными значениям, то такие данные не могут быть исправлены, поскольку он и уже становятся недоступны для предприятия. Управление данными (Data Governance) может по требоваться, чтобы помочь принять решение о том, как будут обрабатываться неверные данные, которые были отправлены из предприятия.

2.1.6. Архивация данных (Data Archival)

Данные могут быть использованы как однократно, так и несколько раз. Но затем рано или поздно жизненный цикл данных начинает подходить к концу.

Первая стадия этого состояния заключается в архивации данных. Архивация данных – это копирование данных в пассивную среду, в которой они хранится, для тех случаев, когда они понадобятся снова в активной производственной среде, и удаление этих данных из всех активных производственных сред.

Архив данных – это просто место, где хранятся данные, без их обслуживания, использования или публикации. В случае необходимости данные могут быть восстановлены из архива.

2.1.7. Уничтожение данных (Data Purging)

Уничтожение данных – это последовательность операций для выполнения необратимого удаления данных, делающая невозможным как восстановление данных, так и получение остаточной информации (Data Remanence) о них. Это одна из самых сложно реализуемых процедур управления данными.

Даже с теоретической точки зрения существует команда записи значения в ячейку памяти, но команды стирания значения как таковой нет. Для уничтожения данных необходимо изготовить высокопроизводительный источник случайных чисел и перезаписать ими весь носитель информации (перезаписи области хранения недостаточно, так как сохраняется информация об исходном количестве данных).

Иначе говоря, при уничтожении данных необходимо не только сделать недоступными от восстановления на физическом уровне сами данные, но и связанную с ними информацию в других наборах данных.

Контрольные вопросы

1. Какие три типа получения данных обычно выделяют на этапе создания данных?
2. Что включает в себя обслуживание данных?
3. Что представляет собой стадия синтеза данных?

4. Какие задачи выполняются на стадии использования данных?
5. Что означает публикация данных?
6. Что представляет собой архивация данных?
7. Что означает уничтожение данных?

2.2 Большие данные. Системы управления большими данными

Если давать краткое определение, то большие данные – это данные, которые не помещаются в оперативную память компьютера. По сути это определение обозначает то, что свойство «быть большим» является не самостоятельным свойством данных, а зависит от характеристики системы, применяемой для их обработки.

Например, обычному человеку затруднительно запомнить какая именно температура была в нашем городе каждый день за прошедший месяц. Таким образом, три десятка значений вполне могут быть примером больших данных. Однако вот человек уверенно сообщает «прошедший месяц был холодным». Это сообщение несет информацию об обработанных данных: по мнению собеседника, средняя температура за прошедший месяц была ни же, чем обычно в этом месяце за несколько десятков лет.

Другим примером могут быть данные об объектах, которые теоретически несут важную информацию, однако имеющие такой размер, что эти данные практически невозможно не только обработать или сохранить, но даже собрать. Рассмотрим, к примеру, набор данных, содержащий координаты и скорости молекул в воздушном столбе над территорией аэропорта. Имеются также метаданные с описанием в какой момент проводилось измерение и что это за молекула. Такой набор данных несет информацию о погодных условиях над аэропортом, включая температуру, давление, влажность, облачность, особые погодные условия – проходящий торнадо или падающий град. С другой стороны, для корректной обработки данные для всех молекул должны быть достаточно полны и репрезентативны для статистической обработки. В результате такого мысленного эксперимента мы понимаем, что для эффективной работы с большими данными нужна модель данных, позволяющая сформировать методы работы с данными.

Данные могут быть различных типов. Информацию, полученную в результате учёта или измерения каких -либо объектов или параметров, называют мастер-данными (Master Data). Например, учёт количества, замеры координат и скоростей конкретных молекул – это мастер-данные. Транзакционные данные (в англоязычной литературе применяются термины Transactional Data, Application Specific Data, Operational Data) – это данные, отображающие результат выполнения каких-либо операций. Например, данные о взаимодействии молекул между собой, а именно о пересечении границ

рассматриваемой области, о траектории конкретной молекулы, об испарении капель дождя – это транзакционные данные. Транзакционные данные описывают взаимодействие объектов друг с другом или с окружающим миром, которые можно получить при помощи обработки мастер-данных.

Ретроспективные данные (Historical data) – это данные, снабженные метками времени. Например, с одной стороны мы можем сохранять данные о координате и векторе скорости каждой молекулы, но если у нас есть набор координат в зависимости от времени, то скорость молекулы становится лишней, она вычисляется исходя из модели, описываемой ньютоновской механикой.

Ссылочные данные (справочники, НСИ, нормативно-ссылочная информация, Reference Data, Lookup Data, Dictionaries) – это базовые неизменяемые данные, заранее известные из внешних источников, такие как нормативы, сокращения, акронимы, словари, стандарты. Например, удельные веса молекул, зависимость температуры замерзания и кипения от давления, зависимость средней скорости молекул (скорости звука) от температуры.

2.2.1. Формат данных. Структурированные данные имеют заранее определенный формат

Полуструктурированные или слабоструктурированные данные – это данные, зачастую собранные из различных источников. Структура данных документирована, но в зависимости от источника данных конкретный формат представления информации может быть разным. Неструктурированные данные требуют обязательной обработки и последующей валидации перед использованием.

Например, данные о координатах и скоростях молекул, в которых некоторые координаты пропущены или некоторые записи повторяются, являются полуструктурированными. Нам нужно понять, почему так произошло и перед использованием либо исключить такие данные (что может привести к систематической ошибке), либо, исходя из модели данных, восстановить пропущенные значения.

Данные, в которых координаты измеряются в разных единицах измерения, числа иногда записаны словами, иногда латинскими цифрами, а иногда в виде сканированного изображения почерка лаборанта, являются неструктурированными данными.

Обычно большие данные описываются при помощи следующих характеристик:

- ***Объем*** (*Volume*) – количество сгенерированных и хранящихся данных. Размер данных определяет значимость и потенциал данных, а также то, могут ли они быть рассмотрены как Большие данные.
- ***Разнообразие*** (*Variety*) – тип данных. Большие данные могут состоять из текста, изображений, аудио, видео. Большие данные при сопоставлении друг с другом могут дополнять отсутствующие данные.

• ***Скорость*** (*Velocity*) – скорость. Здесь подразумевается скорость, с которой данные генерируются и обрабатываются. Очень часто Большие данные используются в режиме реального времени.

• ***Изменчивость*** (*Variability*) – противоречивость наборов данных может препятствовать их обработке и управлению ими.

• ***Достоверность*** (*Veracity*) – качество данных напрямую влияет на точность проведения анализа данных.

2.2.2 Архитектура технологий обработки больших данных

В 2004 году Google опубликовал статью, посвященную процессу MapReduce. Каркас MapReduce использует модель параллельной обработки очень большого объема данных. В рамках MapReduce запросы разделяются и распределяются по параллельным узлам, при этом они обрабатываются параллельно (этап распределения (Map)). Затем осуществляется сбор и доставка результатов (этап предварительного преобразования (Reduce)). Этот каркас был весьма эффективен, поэтому другие компании также захотели скопировать алгоритм. Каркас MapReduce был использован в проекте Apache с открытым исходным кодом, носящем название Hadoop.

Последние исследования показали, что многоуровневая архитектура может успешно использоваться для анализа больших данных. Архитектура распределенной параллельной обработки распределяет данные по многочисленным узлам, которые, производя, соответственно, параллельную обработку, генерируют результат гораздо быстрее, чем другие системы. В рамках такой архитектуры данные передаются в параллельные СУБД с использованием каркасов MapReduce и Hadoop. При этом задействуются фронтальные серверы приложений.

2.2.3 Решение практических задач с помощью технологии обработки больших данных

Обработка больших данных сейчас предполагает, как правило, внедрение специальных программных комплексов, таких как, Hadoop, позволяющих производить обработку больших объёмов данных на основании концепции Map-Reduce.

Hadoop на данный момент является «де-факто» стандартом обработки больших данных. Hadoop представляет собой фреймворк, на основе которого разрабатываются приложения для анализа и визуализации больших данных.

Хранение данных в данном фреймворке осуществляется с помощью специальной распределённой файловой системы HDFS (Hadoop Distributed File System), которая лежит в основе Hadoop и позволяет хранить и предоставлять доступ к данным сразу на нескольких узлах кластера. Таким образом, если один или несколько узлов кластера выходят из строя, то риск потери информации сводится к минимуму и кластер продолжает работу в штатном режиме.

2.2.4 Процесс обработки данных в Hadoop-приложении

Для обработки больших данных используется алгоритм Map-Reduce, при этом все стадии Map должны завершить свою работу до начала Reduce. Также входные данные требуют предварительной обработки. Таким образом, получается общий алгоритм работы, изображенный на рисунке 2.2.4.1.

Рис. 2.2.4.1 – Алгоритм анализа данных

Одной из самых актуальных задач современных информационных технологий является задача быстрой обработки больших объёмов данных. Эффективное решение данной задачи, позволяет быстрее принимать решения на основе данных, полученных в прошлом. В работе были проанализированы методы и подходы к технологии обработке больших данных.

Контрольные вопросы

1. Что представляет собой каркас MapReduce?
2. Какие преимущества предоставляет многоуровневая архитектура для анализа больших данных?
3. Что такое Hadoop, и почему он является «де-факто» стандартом обработки больших данных?
4. Какой алгоритм используется для обработки больших данных в Hadoop-приложении?

Задания для самостоятельного выполнения

Создайте схему базы данных для одной из предметных областей (в упрощённой постановке):

- Интернет-магазин
- Библиотека
- Интернет-банкинг

Для выбранной предметной области оцените:

- Скорость прироста данных в системе, считая что среднее количество посетителей в день – 10000. Количество запросов на модификацию данных – 20%, на чтение – 80% от общего количества запросов.
- Объём хранимых данных через 10 лет, считая, что никакие данные удаляться не будут

СПИСОК РЕКОМЕНДОВАННЫХ ИСТОЧНИКОВ

1. Franks, B. Taming the Big Data Tidal Wave Finding Opportunities in Huge Data Streams with Advanced Analytics / Bill Franks,2012. – 45 с.

2. Gantz, J. The digital universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East - United States / J. Gantz, D. Rainsel // IDC Country brief, 2013.

3. Антонов, А.С. Параллельное программирование с использованием технологии OpenMP / А.С. Антонов. – Москва: МГУ, 2009. – 77 с.

4. Валях, Е. Последовательно-параллельные вычисления / пер. И.А. Николаева, А.М. Степанова. – Москва: Мир, 1985. – 456 с.

5. Воеводин, В.В. Параллельные вычисления / Вл. В. Воеводин – Санкт-Петербург: БХВ Петербург, 2002. – 608 с.

6. Гергель, В.П. Теория и практика параллельных вычислений: учебное пособие / В.П. Гергель. – 2-е изд. – Москва: ИНТУИТ, 2016. – 500 с.

7. Кормен, Т.Х. Алгоритмы: построение и анализ / Лейзерсон Ч.Э., Ривест Р. Л. – 3-е изд. – Москва: ООО «И.Д. Вильямс», 2013. – 1328 с.

8. 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.

9. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. – М.: Вильямс, 2003. – 512 с.