Соснин В.В., Балакшин П.В., Шилко Д.С., Пушкарев Д.А. Введение в параллельные вычисления. – СПб: Университет ИТМО, 2021. – ?? с.

В пособии излагаются основные понятия и определения теории параллельных вычислений. Рассматриваются основные принципы построения программ на языке «Си» для многоядерных и многопроцессорных вычислительных комплексов с общей памятью. Предлагается набор заданий для проведения лабораторных и практических занятий.

Учебное пособие предназначено для студентов, обучающихся по магистерским программам направления «09.04.04 – Программная инженерия», «09.04.01 – Информатика и вычислительная техника», и может быть использовано выпускниками (бакалаврами и магистрантами) при написании выпускных квалификационных работ, связанных с проектированием и исследованием многоядерных и многопроцессорных вычислительных комплексов.

УНИВЕРСИТЕТ ИТМО

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

Содержание

Введение 3

1 Теоретические основы параллельных вычислений 4
- 1.1 История развития параллельных вычислений . . . 4
- 1.2 Термины и определения . . . 6
- 1.3 Классификация параллельных систем (архитектур) . . . 9
- 1.4 Методы синхронизации в параллельных программах . . . 11
- 1.5 Автоматическое распараллеливание программ . . . 14
- 1.6 Основные подходы к распараллеливанию . . . 15
- 1.7 Атомарность операций в многопоточной программе . . . 16
- 1.8 Lock-free структуры данных . . . 17

2 Показатели эффективности параллельной программы 20
- 2.1 Параллельное ускорение и параллельная эффективность . . . 20
- 2.2 Метод Амдала . . . 22
- 2.3 Метод Густавсона-Барсиса . . . 24
- 2.4 Модификация закона Амдала (по проф. Бухановскому) . . . 24
- 2.5 Измерение времени выполнения параллельных программ . . . 25
- 2.6 Профилирование параллельных программ. . . . 28

Введение

В настоящее время большинство выпускаемых микропроцессоров являются многоядерными. Это касается не только настольных компьютеров, но и в том числе мобильных телефонов и планшетов (исключением пока являются только встраиваемые вычислительные системы). Для полной реализации потенциала многоядерной системы программисту необходимо использовать специальные методы параллельного программирования, которые становятся всё более востребованными в промышленном программировании. Однако методы параллельного программирования ощутимо сложнее для освоения, чем традиционные методы написания последовательных программ.

Целью настоящего учебного пособия является описание практических заданий (лабораторных работ), которые можно использовать для закрепления теоретических знаний, полученных в рамках лекционного курса, посвященного технологиям параллельного программирования. Кроме этого, в пособии в сжатой форме излагаются основные принципы параллельного программирования.

При программировании многопоточных приложений приходится решать конфликты, возникающие при одновременном доступе к общей памяти нескольких потоков. Для синхронизации одновременного доступа к общей памяти в настоящее время используются следующие три концептуально различных подхода:

1. Явное использование блокирующих примитивов (мьютексы, семафоры, условные переменные). Этот подход исторически появился первым и сейчас является наиболее распространённым и поддерживаемым в большинстве языков программирования. Недостатком метода является достаточно высокий порог вхождения, т.к. от программиста требуется в "ручном режиме" управлять блокирующими примитивами, отслеживая конфликтные ситуации при доступе к общей памяти.

2. Применение программной транзакционной памяти (Software Transactional Memory, STM). Этот метод проще в освоении и применении, чем предыдущий, однако до сих пор имеет ограниченную поддержку в компиляторах, а также в полной мере он сможет себя проявить при более широком распространении процессоров с аппаратной поддержкой STM.

3. Использование неблокирующих алгоритмов (lockless, lock-free, wait-free algorithms). Этот метод подразумевает полный отказ от применения блокирующих примитивов при помощи сложных алгоритмических ухищрений. При этом для корректного функционирования неблокирующего алгоритма требуется, чтобы процессор поддерживал специальные атомарные (бесконфликтные) операции вида "сравнить и обменять" (cmpxchg, "compare and swap"). На данный момент большинство процессоров имеют в составе системы команд этот тип операций (за редким исключением, например: "SPARC 32").

Предлагаемое вниманию методическое пособие посвящено первому из перечисленных методов, т.к. он получил наибольшее освещение в литературе и наибольшее применение в промышленном программировании. Два других метода могут являться предметом изучения углублённых учебных курсов, посвященных параллельным вычислениям.

Авторы ставили целью предложить читателям изложение основных концепций параллельного программирования в сжатой форме в расчёте на самостоятельное изучение пособия в течение двух-трёх месяцев. При использовании пособия в технических вузах рекомендуется приведённый материал использовать в качестве односеместрового учебного курса в рамках подготовки студентов по направлению подготовки "Программная инженерия" или смежных с ней.

1 Теоретические основы параллельных вычислений

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.

На рисунке 1 показано, какую долю занимали процессоры с разным количеством ядер при создании суперкомпьютеров в разное время (по материалам сайта `http://top500.org`). Закрашенные области помечены цифрами для обозначения количества ядер. Ширина области по вертикали равна относительной частоте использования процессоров соответствующего типа в рассматриваемом году.

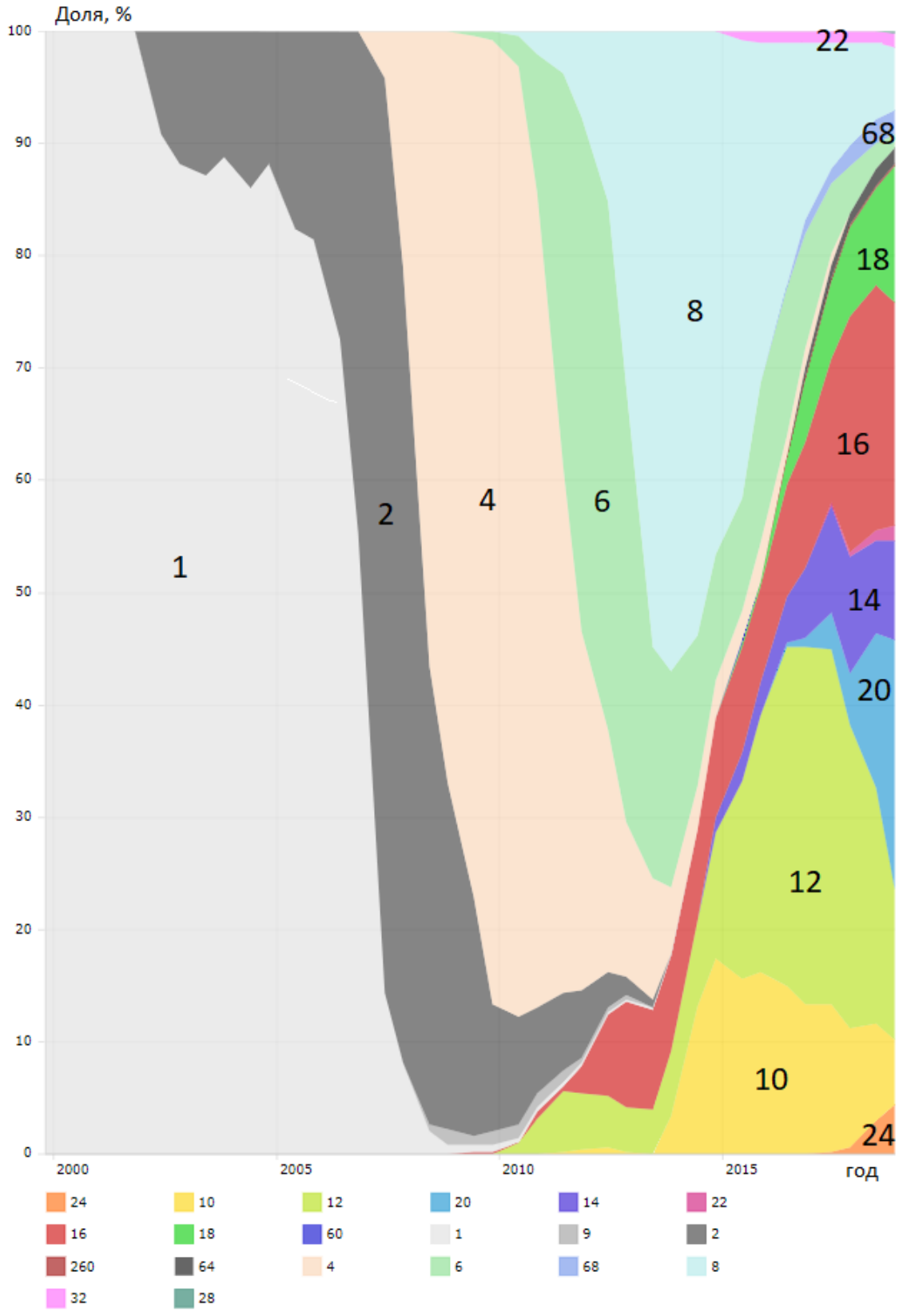

Рис. 1: Частотность использования процессоров с различным числом ядер при создании суперкомпьютеров

Как видим, активное использование двухъядерных процессоров в суперкомпьютерах началось уже в 2002 году, а примерно к 2005 году совершенно сошло на нет, тогда как в настольных компьютерах их применение в 2005 году лишь начиналось. На основании этого можно сделать простой прогноз распространённости многоядерных "настольных" процессоров к нужному году, если считать, что они в общих чертах повторяют развитие многоядерных архитектур суперкомпьютеров.

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 неидентичность 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 оконачания 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 $\neq$ parallel computing.
- параллельные вычисления = concurrent computing;
- многоядерные вычисления = parallel computing;
- parallel computing = multicore computing = SMP;

Распределённые вычисления (distributed computing) – такие ПВ, при которых для решения задачи вычисления происходят на процессорах, расположеных на разных компьютерах, соединённых 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/с (см. детали ниже в этом разделе).

Важно различать понятия "параллельные вычисления" и "параллельные технологии". Разберем следующие понятия, которые, хотя и являются параллельными технологиями (на уровне ядра или межъядерного взаимодействия), однако не являются параллельными вычислениями, но часто по ошибке причисляются к ним:

- 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 2). Этот способ организации вычислений не является параллельными вычислениями, потому что инструкции все равно выполняются последовательно, а так же задействовано только одно ядро.

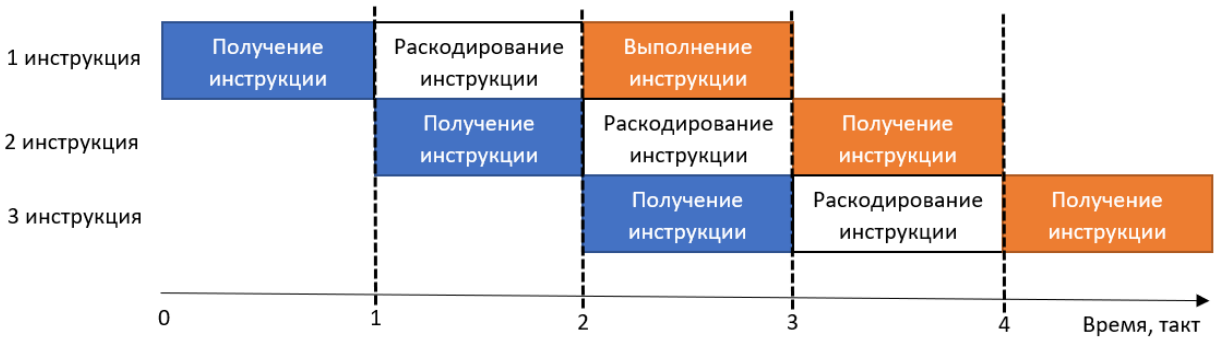

Рис. 2: Конвейерная обработка инструкций

- SIMD-расширения (MMX, SSE) обеспечивают параллелизм на уровне данных. Например, процессор может одновременно умножать 4 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, одна 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 отличии от тредов, которые могут работать на разных ядрах.

Для лучшего понимания тредов схематично рассмотрим его жизненный цикл (lifecycle). На рисунке 3 видно, что поток может находится 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.

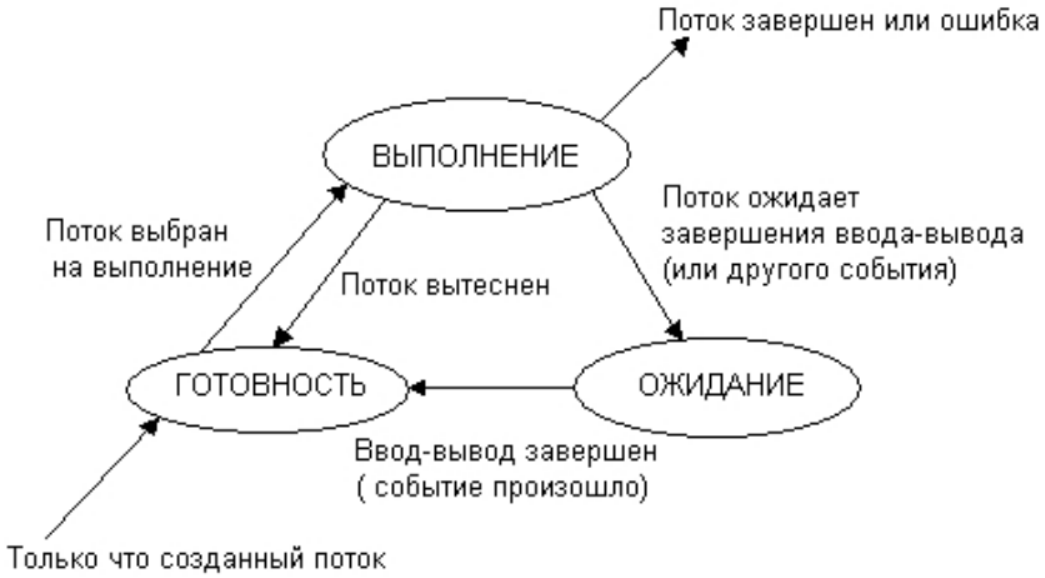

Рис. 3: Жизненный цикл потока

В среде программистов существуют понятия потокобезопасной (thread-safe) и реэнтерабельной (reentrant или reenterable) функции, однако в разных сообществах они могут иметь различные значения. В таблице 1 написаны определения из разных источников.

Таблица 1: Определения thread-safe и reentrant функций

Источник определения	Thread-safe	Reentrant
Qt	Внутри функции обращение ко всем общим переменным осуществляется строго последовательно, а не параллельно (Thread-safe является reentrant, но не наоборот)	При вызове функции одновременно несколькими потоками гарантируется правильная работа, только если потоки не используют общие данные
Linux	Функция показывает правильные результаты, даже если вызвана несколькими тредами одновременно	Функция показывает правильные результаты, даже если повторно вызвана изнутри себя
POSIX	?	Функция показывает правильные результаты, даже если вызвана несколькими тредами одновременно

Рассмотрим примеры функций, подходящие под определение сообщества Linux.

swapExample1.cpp

Данная функция не является не потокобезопасной, не реентерабельной, потому что все потоки вызывающие ее будут использовать общую переменную t. Если вызвать функцию внутри ее самой, то перезапишется значение t и родительская функция отработает неправильно. Попробуем исправить эти ошибки, объявив переменную t типа __thread int.

swapExample2.cpp

Теперь компилятор создаст копию переменной для каждого потока t и функция станет потокобезопасной, однако она все еще не реентерабельна по той же причине. Будем сохранять значение глобальной переменной t в начале функции и восстанавливать ее в конце.

swapExample3.cpp

Новая функция реентерабельна, но снова потоконебезопасна. Наконец, приведем пример стандартной и правильной реализации swap(), которая потокобезопасна и реентерабельна:

swapExample4.cpp

1.3 Классификация параллельных систем (архитектур)

По физической архитектуре параллельные системы можно разделить на 2 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 и имеют равный доступ ко всем ресурсам вычислительной системы (памяти и устройствам ввода-вывода) и управляются все одной копией 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 4. 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.

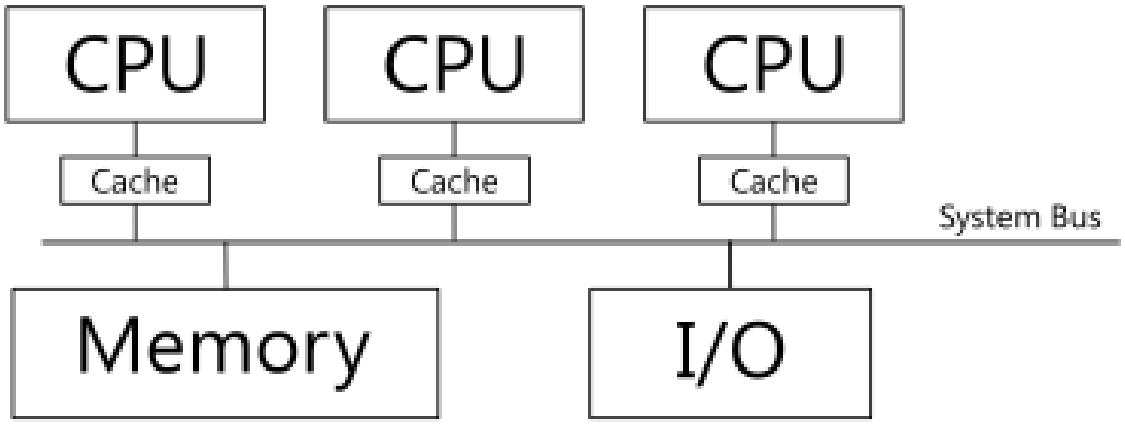

Рис. 4: Архитектура SMP. Подключение процессоров по системной шине

- Коммутируемое соединение (crossbar switch) изображено на рисунке 5. 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.

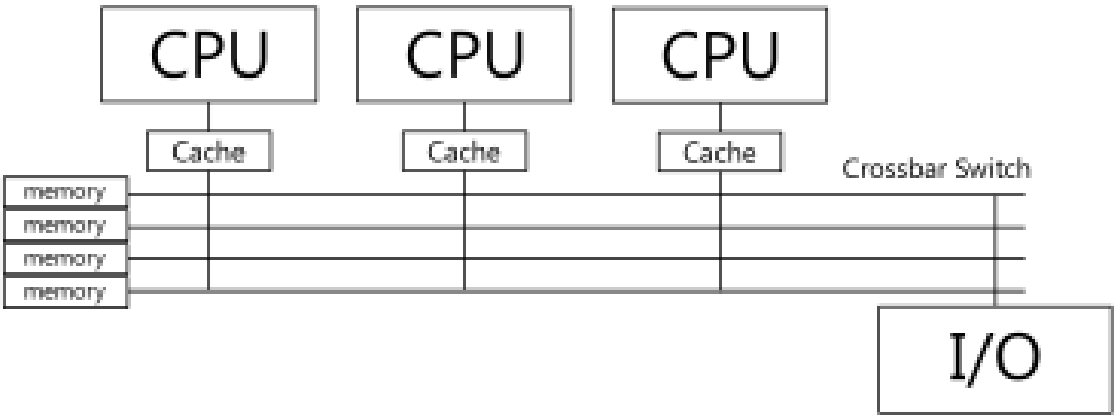

Рис. 5: Архитектура SMP. Подключение процессоров через коммутируемое соединение

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 2 сокета, 3 процессора уже не поставить).

MMP - 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 6).

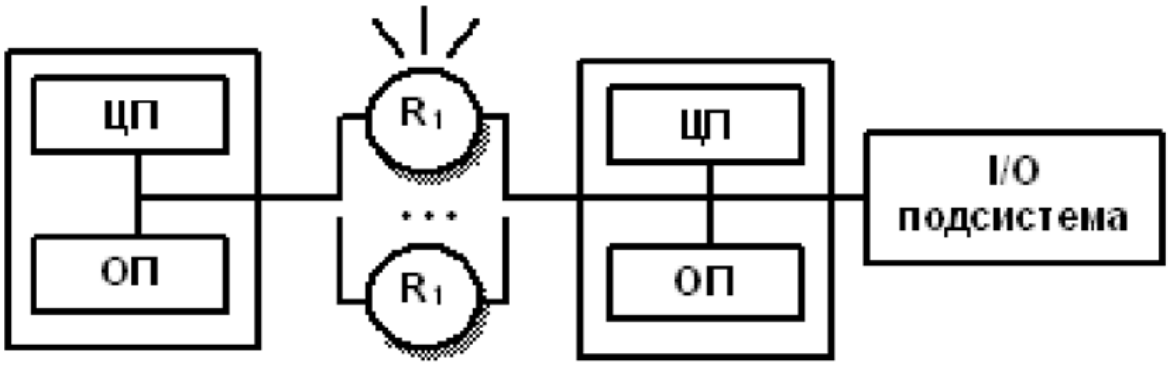

Рис. 6: Архитектура MMP

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.

1.4 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 2, а может и на 1 в зависимости от поведения потоков, так как операция инкрементации представляет собой как минимум 3 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. Подробнее проблема реализации атомарных операций будет поднята в разделе 1.7 Атомарность операций в многопоточной программе.

- Семафор - объект, ограничивающий число потоков, которые могут войти в эту область кода. Как правило это число задается при инициализации семафора. Затем при захвате семафора потоком проверяется количество потоков, захвативших семафор. Если максимальное количество потоков достигнуто, то 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 доступи 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 из курса дискретной математики. С его помощью можно перейти от последовательной программы к каскадной. Однако не для всех программ этот лагоритм работает, иногда он может уйти в бесконечный цикл или завершиться с ошибкой.

- Barrier (барьер) - участок кода, в котором синхронизируется состояние потоков. 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 разделяемой памяти. Также плюсом таких алгоритмов является безопасная обработка прерываний. Для реализации таких алгоритмов часто используются другие технологии синхронизации: read-modify-write, CAS (см. раздел 1.7) 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 не обязательно 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;
    3. __transaction_cancel — явная отмена транзакции;
    4. attribute((transaction_safe)) — указание транзакционно-безо-пасной функции;

5. attribute((transaction_pure)) — указание функции без побочных эффектов.

- Модель акторов - математическая модель параллельных вычислений, в которой программа представляет собой 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 количество 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.

1.5 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.

Приведём примеры того, как с-программа в файле src.c может быть автоматически распараллелена при использовании некоторых популярных компиляторов:

- Компилятор GNU Compiler Collection: gcc -O3 -floop-parallelize-all -ftree-parallelize-loops=K -fdump-tree-parloops-details src.c. При этом программисту даётся возможность выбрать значение параметра K, который рекомендуется устанавливать равным количеству ядер (процессоров). Особенностям реализации автораспараллеливания в gcc посвящён самостоятельный проект:
  `https://gcc.gnu.org/wiki/AutoParInGCC`.
- Компилятор фирмы Intel: icc -c -parallel -par-report file.cc
- Компилятор фирмы Oracle: solarisstudio -cc -O3 -xautopar -xloopinfo src.c

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 независимо 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 7 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 тд, 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.

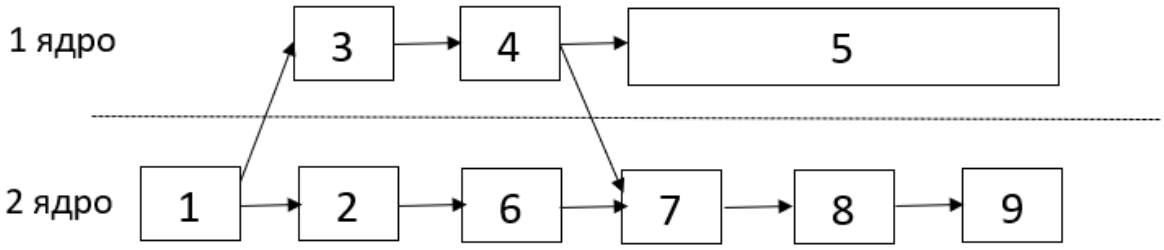

Рис. 7: Пример работы структурного алгоритма на двухъядерном процессоре

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 она 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.

1.7 Атомарность операций в многопоточной программе

Основной проблемой при параллельном программировании является необходимость устранять конфликты при одновременном доступе к общей памяти нескольких потоков. Для решения этой проблемы обычно пытаются упорядочить доступ потоков к общим данным с помощью специальных средств – примитивов синхронизации. Однако возникает вопрос, существуют ли такие элементарные атомарные операции, выполнение которых несколькими потоками одновременно не требует синхронизации действий, т.к. эти операции выполнялись бы процессором "одним махом", или – как принято говорить – "атомарно" (т.е. никакая другая операция не может вытеснить из процессора предыдущую атомарную операцию до её окончания).

Такими операциями являются практически все ассемблерные инструкции, т.к. они на низком уровне используют только те операции, которые присутствуют в системе команд процессора, а значит могут выполняться атомарно (непрерываемо). Однако при компиляции С программы команды языка С транслируются обычно в несколько ассемблерных инструкций. В связи с этим возникает вопрос о возможном существовании С-команд, которые компилируются в одну ассемблерную инструкцию. Такие команды можно было бы не "защищать" примитивами синхронизации (мьютексами) при параллельном программировании.

Однако оказывается, что таких операций крайне мало, а некоторые из них могут вести себя как атомарно, так и не атомарно в зависимости от аппаратной платформы, для которой компилируется С-программа. Рассмотрим простейшую команду инкремента целочисленной переменной (тип int) в языке С: "w++". Можно легко убедиться (например, используя ключ "-S" компилятора gcc), что эта команда будет транслирована в три ассемблерные инструкции (взять из памяти, увеличить, положить обратно):

[language=[x86masm]Assembler]atomicOperationExample1.asm

Значит, выполнять операцию инкремента некоторой переменной в нескольких потоках одновременно - небезопасно, т.к. при выполнении ассемблерной инструкции 2 поток может быть прерван и процессор передан во владение другому потоку, который получит некорректное значение недоинкрементированной переменной.

Логично было бы предположить, что операции присваивания не должны обладать описанным недостатком. Действительно, в Ассемблере есть отдельная инструкция для записи значения переменной по указанному адресу. К сожалению, это предположение не до конца верно: действительно, при выполнении присваивания переменной типа char эта операция будет выполнена единой ассемблерной инструкцией. Однако с другими типами данных этого нельзя сказать наверняка. Общее практическое правило можно грубо сформулировать так: "атомарность операции присваивания гарантируется только для операций с данными, разрядность которых не превышает разрядности процессора".

Например, при присваивании переменной типа int на 32-разрядном процессоре будет сгенерирована одна ассемблерная инструкция. Однако при компиляции этой же операции на 16-разрядном компьютере будет сгенерировано две ассемблерные команды для независимой записи младших и старших бит.

Следует иметь в виду, что сформулированное правило работает при присваивании переменных и выражений, однако не всегда может выполняться при присваивании констант. Рассмотрим пример С-кода, в котором 64-разрядной переменной s (тип uint64_t) присваивается большое число, заведомо превышающее 32-разрядную величину:

atomicOperationExample2.cpp

Этот код будет транслирован в следующий ассемблерный код на 64-разрядном процессоре:

[language=[x86masm]Assembler]atomicOperationExample3.asm

Как видим, операция присваивания была транслирована в две ассемблерные инструкции, что делает невозможным безопасное распараллеливание такой операции.

Сформулированное правило применимо не только к операции присваивания, но и к операции чтения переменной из памяти, поэтому любую из этих операций в потокобезопасной среде придётся защищать мьютексами или критическими секциями.

Особый случай атомарного изменения данных - это изменение структуры. Для этого надо использовать CAS-операцию с указателем на эту структуру. Выполняя такую операцию, процессор создаст вторую структуру данных с заданными полями и сравнит её со старой версией структуры. Если значение хотя бы одного поля поменялось, то он атомарно подменит указатель. В этом есть накладные расходы: даже простое изменение одного поля структуры требует создание полной копии структуры, чтобы потом подменить указатель.

1.8 Lock-free структуры данных

В многопоточных программах проблемы при совместной работе потоков обычно возникать при доступе к общим ресурсам. Помимо блокирующего подхода, использующего примитивы синхронизации, также используется неблокирующий подход. Для того чтобы избежать состояния гонки можно использовать специальные неблокирующие структуры данных. Данный подход основывется на использовании атомарных переменных и lock-free или wait-free объектов.

Разделяемый объект называется lock-free объектом, если он гарантирует, что некоторый поток закончит выполнение операции над объектом за конечное количество шагов вне зависимости от результатов работы других потоков.

Объект является wait-free, если каждый поток завершит операцию над объектом за конечное число шагов.

Может возникнуть вопрос зачем нужны неблокирующие структуры данных, если можно использовать примитивы синхронизации для доступа к обычной структуре данных. Lock-free структуры имеют ряд преимуществ над блокирующими структурами данных. Так, по пропускной способности они превосходят блокирующие в 1.5 – 3 раза, однако как блокирующие, так и неблокирующие очереди имеют слабую масштабируемость относительно числа потоков. По величине задержки элементов в очереди неблокирующие очереди также имеют лучшие характеристики, однако их преимущество достаточно мало. Также использование примитивов синхронизации может привести к deadlock, а также могут возникать ошибки, связанные с забыванием захвата или освобождения примитивов.

Lock-free структуры данных не содержат блокировок и остается в консистентном состоянии в независимости от количества потоков, одновременно обращающихся к ней. Такие структуры данных можно организовать с помощью RMW (read-modify-write) – операции чтения, изменения и записи, происходящая атомарно.

Примером RMW операции может служить CAS. В библиотеке C++ существует два варианта реализации этой операции: weak и strong (рисунок 8). Weak версия может вернуть false в случае, когда считанное значение было равно ожидаемому. Strong всегда возвращает правильное значение.

```
bool
compare_exchange_weak(_Tp& __e, _Tp __i, memory_order __s,
                      memory_order __f) noexcept

bool
compare_exchange_strong(_Tp& __e, _Tp __i, memory_order __s,
                        memory_order __f) noexcept
```

Рис. 8: Сигнатуры CAS операции в библиотеке C++

Альтернативой CAS операций служит пара LL/SC операций в ARM процессорах. Load-link операция загружает значение из памяти, a store-conditional устанавливает новое значение, но только в том случае, если область памяти не менялась. Для реализации LL/SC операций пришлось изменить структуру кэша: к каждой линии кэша добавляется флаг LINK. Флаг устанавливается при

операции LL и сбрасывается при SC или вытеснении кэш-линии. LL/SC операции не подвержены проблеме ABA, однако из-за аппаратной реализации может возникать false sharing. В современных процессорах длина кэш-линии составляет 64 - 128 байт, следовательно, в одной кэш-линии может находиться несколько переменных. При работе с несколькими переменными в одной линии у LL/SC операций будет общий флаг LINK, что может привести к неправильно работе. Чтобы данной проблемы не возникало, следует размещать по одной переменной в линии.

falseSharingLLSCExample.cpp

CAS операцию можно достаточно легко реализовать с помощью LL/SC операций:

LLSCthroughCAS.cpp

Также важно понимать, что lock-free алгоритмы чувствительны к переупорядочению машинных инструкций в их коде. Чтобы избежать этого используются барьеры памяти. Барьер памяти X_Y гарантирует, что все X-операции до барьера будут выполнены до того как начнут выполняться Y-операции после барьера. В теории существует 4 вида барьеров – LoadLoad, LoadStore, StoreLoad, StoreStore, однако не все из них реализованы по всех архитектурах. Существует 4 модели памяти процессоров:

- Relaxed model – возможно переупорядочение любых инструкций обращения к памяти, даже зависящих по данным (DEC Alpha).
- Weak model – возможно переупорядочение любых инструкций чтения и записи, кроме тех, которые имеют зависимости по данным (ARM, PowerPC, Intel Itanium).
- Strong model – возможно только переупорядочение вида чтение до записи (x86).
- Sequentual consistency model – любое переупорядочение запрещено.

Существуют различные lock-free структуры данных: очереди (со строгим и ослабленным порядком), стек, связные списки, хеш-таблицы. В C++ данные структуры данных можно использовать подключая различные библиотеки. Например, Boost содержит реализацию очереди и стека, а Libcds – все перечисленные.

Примером lock-free структуры данных может служить очередь Майкла - Скотта. Эта очередь реализуется на базе односвязного списка и двух указателей, один из которых указывает на голову списка (dummy node), а другой – на хвост (рисунок 9).

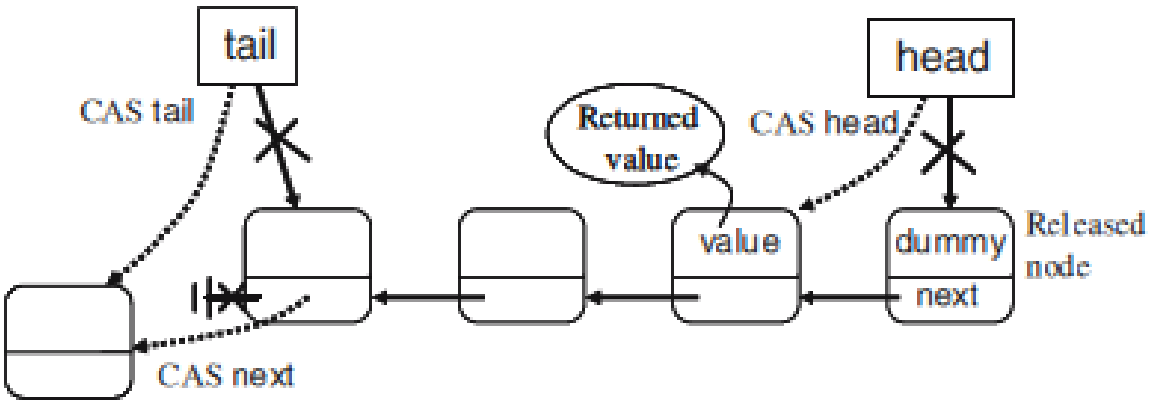

Рис. 9: Очередь Майкла - Скотта

Рассмотрим упрощенный код очереди из библиотеки libcds. Ниже представлена функция enqueue – добавления в очередь. Сначала переданное значение кладется в node. Затем мы пытаемся положить его в хвост очереди. После получения текущего хвоста указатель продвигается, пока не

дойдет до фактического хвоста. Затем значение ставится в конец очереди и хвосту присваивается значение вставленного элемента.

lockFreeQueueEnqueue.cpp

Для того чтобы достать элемент из очереди (функция dequeue), чтобы очередь не была пуста, а также чтобы хвост и голоса были продвинуты. Код приведен ниже.

lockFreeQueueDequeue.cpp

2 Показатели эффективности параллельной программы

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, VirtualBox) 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). Например, на рисунке 10 показано, что для настраиваемой виртуальной машины из восьми доступных физических (и логических) процессоров доступными являются только три.

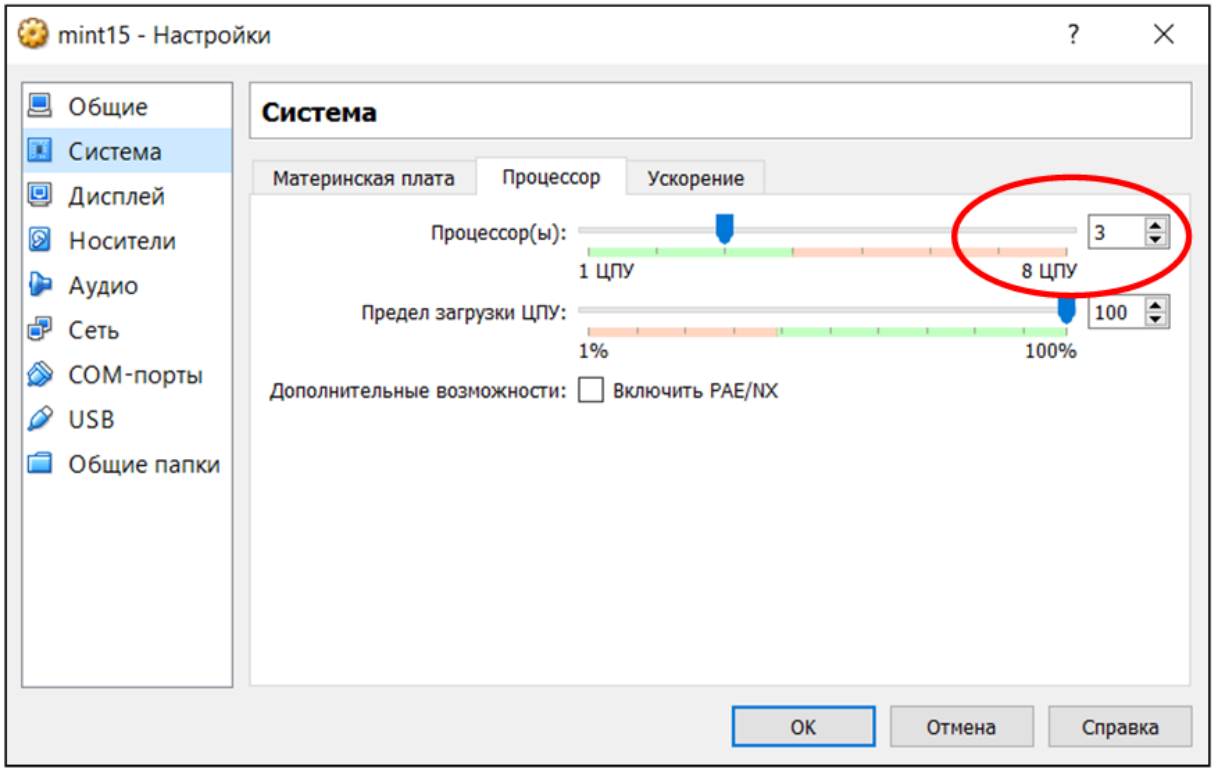

Рис. 10: Выбор количества виртуальных процессоров в Oracle VirtualBox

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 количеством потоков. При создании параллельных программ достаточно часто количество 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 количество 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 количество доступных процессоров (ядер) на используемой для тестирования многопроцессорной аппаратной платформе. Описанный подход позволяет искусственно ограничить количество 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 (см. ниже).

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 количестве процессоров по сравнению с однопроцессорной системой, т.е.

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

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 сверх-линейное 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 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).

Рассмотрим подробнее каждый из указанных методов в двух следующих подразделах.

2.2 Метод Амдала

При оценке эффективности распараллеливания некоторой программы, выполняющей фиксированный объём работы, скорость выполнения 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:

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

Запишем время $t(1)$ следующим образом:

$$t(1) \;=\; t(1) \;+\; (k \;\cdot\; t(1) \;-\; k \;\cdot\; t(1)) \;=\; k \;\cdot\; t(1) \;+\; (1 \;-\; k) \;\cdot\; t(1), \tag{4}$$

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:

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

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

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

которое перепишем в виде

$$S(p)|_{w=const} = S_A(p) = \left(\frac{k}{p} + 1 - k\right)^{-1} \tag{6}$$

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.

График зависимости параллельного ускорения от количества ядер изображен на рисунке 11:

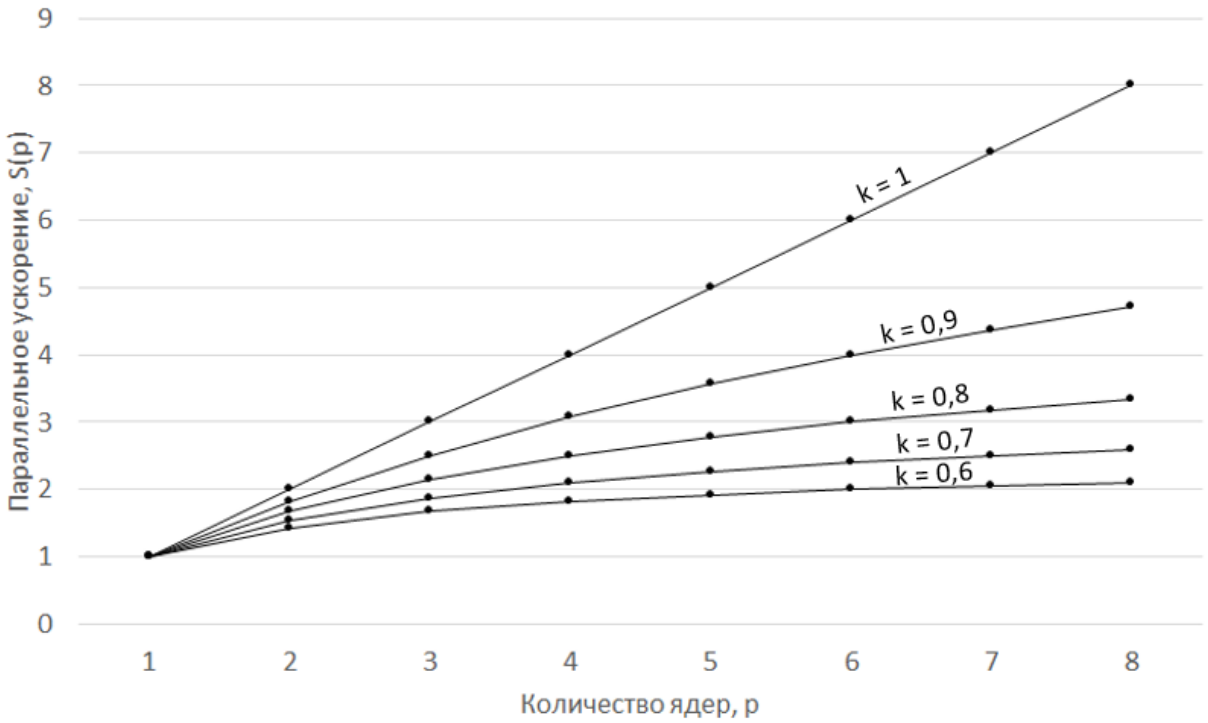

Рис. 11: График зависимости параллельного ускорения от количества ядер по Амдалу

Отметим, что выражение для расчёта параллельной эффективности при использовании метода Амдала можно получить, объединив формулы (2) и (6), а именно:

$$E_A(p) \;=\; (k \;+\; p \;-\; p \;\cdot\; k)^{-1} \tag{7}$$

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 p_{max} 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 известно как "свёрточное", является принципиально нераспараллеливаемым при кодировании выбранного блока данных.

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:

$$S(p)|_{t=const} \;=\; \frac{V(p)}{V(1)} \;=\; \frac{w(p)}{t} \;:\; \frac{w(1)}{t} \;=\; \frac{w(p)}{w(1)}. \tag{8}$$

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

$$w(1) \;=\; w(1) \;+\; (k \;\cdot\; w(1) \;-\; k \;\cdot\; w(1)) \;=\; k \;\cdot\; w(1) \;+\; (1 \;-\; k) \;\cdot\; w(1), \tag{9}$$

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 нераспараллеленных условных единиц работы $(1 - k) \cdot w(1)$, которое не изменится по сравнению с формулой (9).
- количество распараллеленных УЕР, объём которых увеличиться в p раз по сравнению с формулой (??), 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 учетом формулы (8) получим: $\frac{w(p)}{w(1)} = \frac{p \cdot k \cdot w(1) + (1-k) \cdot w(1)}{w(1)}$, что позволяет записать:

$$S(p)|_{t=const} \;=\; S_{GB}(p) \;=\; p \;\cdot\; k \;+\; 1 \;-\; k \tag{10}$$

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

2.4 Модификация закона Амдала (по проф. Бухановскому)

В реальных вычислительных системах ОС тратит ресурсы на создание и удаление новых потоков. Время, затраченное на эти операции не учитывается в законе Амдала. Параллельное ускорение $S(p)$ зависит от количества ядер и доли распараллеливаемых операций, но не зависит от

количества последних. Выведем формулу в которой количество операций для которых необходимо создать поток будет учитываться.

Пусть N – количество распараллеливаемых операций, M – количество нераспараллеливаемых операций, t_c – время выполнения одной операции, p – количество вычислителей(ядер), T_i – время выполнения программы при использовании i параллельных потоков на i вычислителях, α – некий масштабирующий коэффициент, инкапсулирующий в себе количество времени, требуемого на создание, удаление потока и прочие накладные операции. По формуле (3), $S(p) = \frac{T_1}{T_p}$.

Найдем сначала T_1. Так как это код выполняется линейно, то время затраченное на его выполнение будет равно количеству операций помноженному на время выполнения одной операции: $T_1 = t_c(N + M)$.

Время выполнение распараллельнной программы T_p включается в себя время на создание потока: $t_c\alpha(p-1)N$ (нужно создать $(p-1)$ новых потоков, так как главный поток уже создан и для каждого затратить какое-то время α), время работы распараллеливаемоего кода на всех ядрах: $\frac{t_c N}{p}$ и время работы нераспараллеливаемого кода $t_c M$. Итого, разделив T_1 на T_p, получим формулу закона Амдала по проф. Бухановскому:

$$S(p,N) = \frac{T_1}{T_p} = \frac{N+M}{\alpha(p-1)N + \frac{N}{p} + M} \tag{11}$$

Из формулы (11) видно, что с ростом количество ядер после определенного предела $S(p,N)$ не будет расти как в законе Амдала, так как время будет тратиться много времени на создание новых потоков. На рисунке 12 наглядно видно, что $S(p,N)$ уменьшается при большом количестве потоков и становится заметно меньше $S(p)$ по Амдалу даже при небольшом значении α.

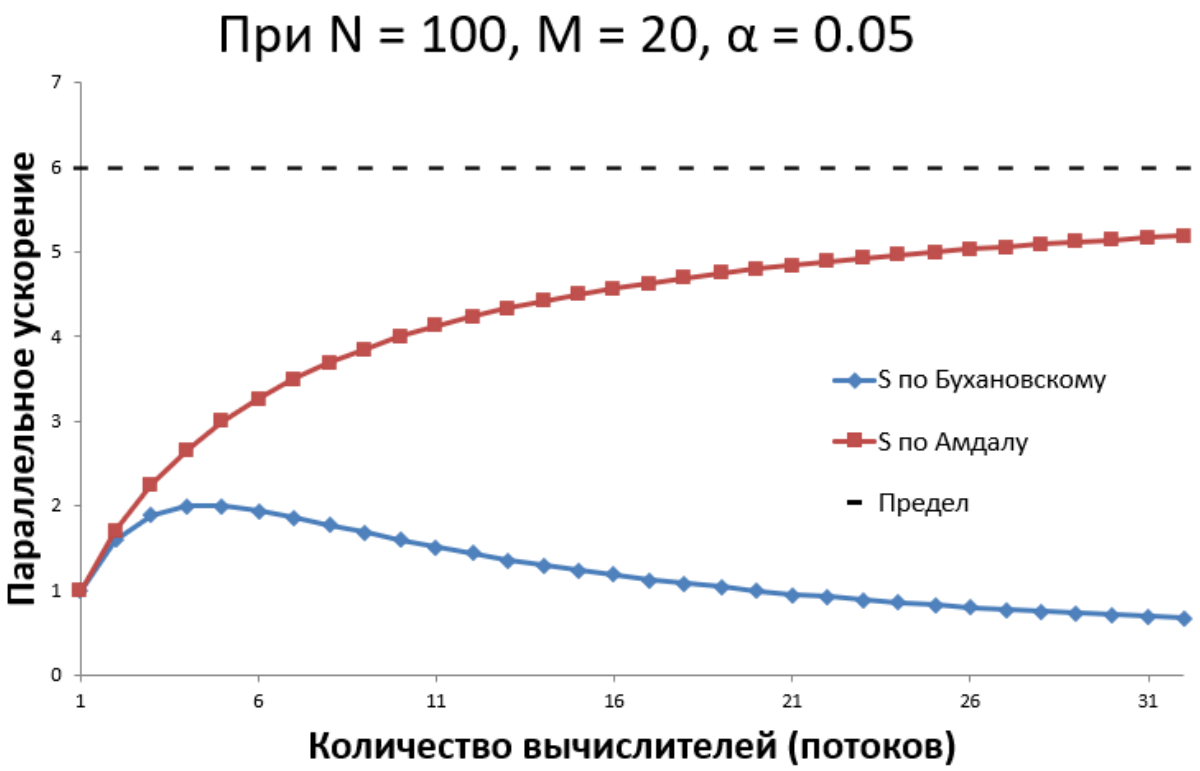

Рис. 12: График зависимости параллельного ускорения от количества потоков

2.5 Измерение времени выполнения параллельных программ

Инструменты измерения времени. Измерение времени работы программы в языке C не является сложной проблемой, однако при параллельном программировании возникает ряд специ-

фических сложностей при выполнении этой операции. Далеко не все функции, пригодные для измерения времени работы последовательной программы, подойдут для измерения времени работы многопоточной программы.

Например, если в однопоточной программе для измерения времени работы участка кода использовать функции ctime или localtime, то они успешно справятся с поставленной задачей. Однако после распараллеливания этого участка кода возможно возникновение трудноидентифицируемых проблем с неправильным измерением времени, т.к. обе указанные функции имеют внутреннюю static-переменную, которая при попытке изменить её одновременно несколькими потоками может принять непредсказуемое значение.

С целью решить описанную проблему в некоторых C-компиляторах (например, gcc) были реализованы потокобезопасные (thread-safe, re-entrant) версии этих функций: ctime_r и localtime_r. К сожалению, эти функции доступны не во всех компиляторах. Например, в компиляторе Visual Studio аналогичную проблему решили использованием функций с совсем иными именами и API: GetTickCount, GetLocalTime, GetSystemTime. Перечислим для полноты изложения некоторые другие gcc-функции, которые также позволяют измерять время: time, getrusage, gmtime, gettimeofday.

Ещё одна стандартная C-функция clock также не может быть использована для измерения времени выполнения многопоточных программ. Однако причина этого не в отсутствии реэнтерабельности, а в особенностях способа, которым эта функция рассчитывает прошедшее время: clock возвращает количество тиков процессора, которые были выполнены при работе программы суммарно всеми её потоками. Очевидно, что это количество остается почти неизменным при выполнении программы разным количеством потоков (”почти”, т.к. накладные расходы на создание, удаление и управление потоками предлагается в целях упрощения изложения считать несущественными).

В итоге оказалось, что удовлетворительного кросс-платформенного решения для потокобезопасного измерения времени с высокой точностью (до микросекунд) средствами чистого языка C пока не существует. Проблему, однако, можно решить, используя сторонние библиотеки, выбирая те из них, которые имеют реализацию на целевых платформах.

Выгодно выделяется среди таких библиотек система OpenMP, которая реализована в абсолютном большинстве современных компиляторов для всех современных операционных систем. В OpenMP есть две функции для измерения времени: omp_get_wtime и omp_get_wtick, которые можно использовать в C-программах, если подключить заголовочный файл omp.h и при компиляции указать нужный ключ (например, в gcc это ключ ”-fopenmp”).

Погрешность измерения времени. Другим интересным моментом при измерении времени работы параллельной программы является способ, с помощью которого исследователь исключает из замеров различные случайные погрешности, неизбежно возникающие при эксперименте в работающей операционной системе, которая может начать процесс обновления или оптимизации, не уведомляя пользователя. Общепринятыми является способ, при котором исследователь проводит не один, а сразу N экспериментов с параллельной программой, не меняя исходные данные. Получается N замеров времени, которые в общем случае будут различными вследствие различных случайных факторов, влияющих на проводимый эксперимент. Далее чаще всего используется один из следующих методов:

1. Расчёт доверительного интервала: с учётом всех N измерений рассчитывается доверительный интервал, например, с помощью метода Стьюдента.

2. Поиск минимального замера: среди N измерений выбирается наименьшее и именно оно используется в качестве окончательного результата.

Первый метод даёт корректный результат, только если ошибки замеров распределены по нормальному закону. Чаще всего это так, поэтому применение метода оправдано и к позволяет получить дополнительную информацию о возможном применении тестируемой программы в живых условиях работающей ОС.

Второй метод не предъявляет требований к виду закона распределения ошибки измерений и этим выгодно отличается от предыдущего. Кроме того, при больших N выбор минимального замера позволит с большой вероятностью исключить из эксперимента все фоновые влияния операционной системы и получить в качестве результата точное измерение времени работы программы в идеальных условиях.

Практический пример. Сравним на примере описанные выше методы избавления от погрешности экспериментальных замеров времени. Будем измерять накладные расходы OpenMP на

создание и удаление потоков следующим образом:

OpenMPExampleTimeMeasurement.cpp

В строке 3 мы даём OpenMP указание, чтобы при входе в параллельную область, расположенную далее в программе, было создано i потоков. Если не давать этого указания, OpenMP создаст количество потоков по количеству доступных в системе вычислителей (ядер или логических процессоров). В строке 4 мы запускаем параллельную область программы, OpenMP создаёт i потоков. В строке 5 мы даём указание выполнять последующую простейшую инструкцию лишь в одном потоке (остальные потоки не будут делать никакой работы. Это нужно, чтобы в замеряемое время работы попали только расходы на создание/удаление потоков, а все прочие расходы терялись бы на их фоне. В строке 6 заканчивается параллельная область, OpenMP удаляет из памяти i потоков. Более подробное описание использованных команд OpenMP можно найти в разделе ?? "Технология OpenMP" данного учебного пособия.

Эксперименты с приведённой программой проводились на компьютере с процессором Intel Core i5 (4 логических процессора) с 8 гигабайт ОЗУ в операционной системе Debian Wheezy. Опытным путём было выявлено, что использованная операционная система на доступной аппаратной платформе не может создать более 381 потока в OpenMP-программе (этим объясняется значение в строке 1). Было проведено в общей сложности N=100 экспериментов, результаты которых обрабатывались каждым из двух описанных методов. Полученные результаты приведены на рисунке 13.

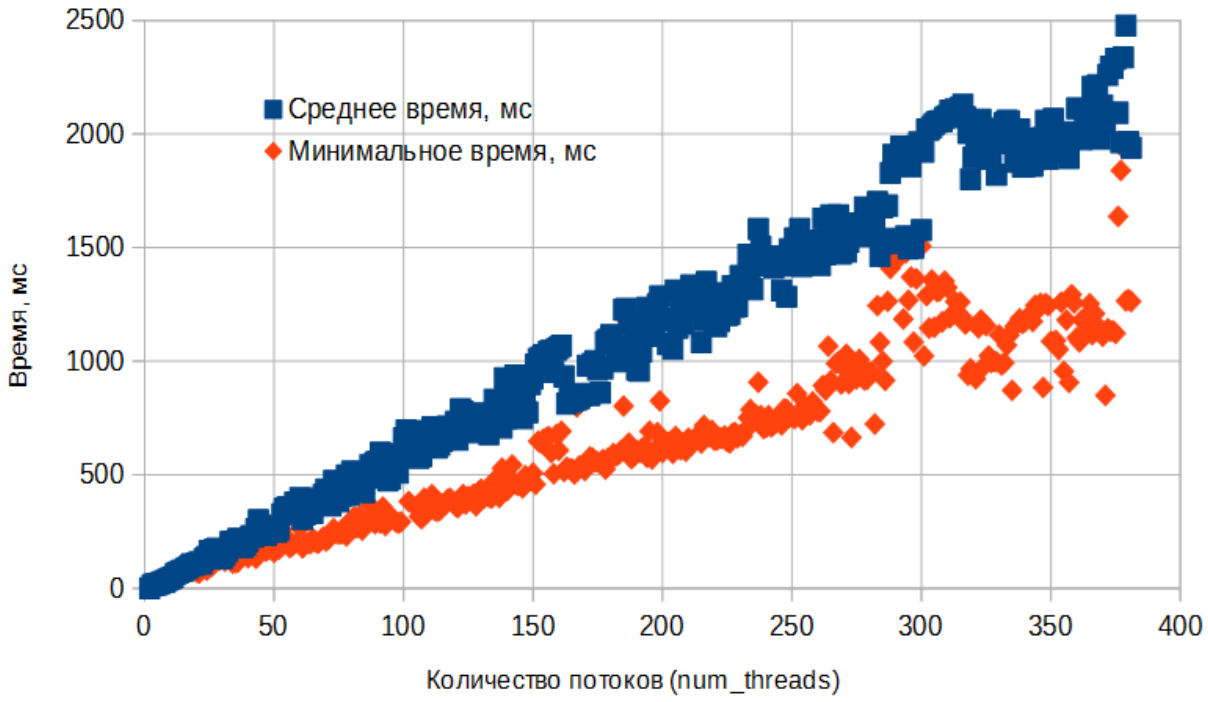

Рис. 13: Результаты измерения накладных расходов OpenMP при создании и удалении потоков

По оси ординат откладывается измеренная величина (T2 - T1) в миллисекундах, по оси абсцисс – значения переменной i, означающие количество создаваемых потоков. Верхний график, состоящий из синих квадратов, показывает усреднённую величину (T2 - T1) по 100 проведённым экспериментам. Доверительный интервал при этом не показан, т.к. он загромождал бы график, не добавляя информативности, однако ширина доверительного интервала с уровнем доверия 90% приблизительно соответствует разбросу по вертикали квадратов верхнего графика для соседних значений i.

Нижний график, состоящий из ромбов, представляет собой минимальные из 100 проведённых замеров величины (T2 - T1) для указанных на оси абсцисс значений i. Видим, что даже

большого количества экспериментов оказалось недостаточно, чтобы нижний график имел бы гладкую непрерывную структуру без заметных флуктуаций.

2.6 Профилирование параллельных программ.

Профилирование — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш-промахов и т. д. Инструмент, используемый для анализа работы, называют профилировщиком или профайлером.

Intel Parallel Amplifier. Этот инструмент позволяет найти те участки кода, которые наиболее часто исполняются на процессоре. Также он позволяет оценить масштабируемость вашего параллельного приложения. И если есть какие-то проблемы с масштабируемостью, то найти те участки кода, которые этой масштабируемости мешают. В Intel Parallel Amplifier представлено три вида анализа:

1. Hotspot-анализ - Позволяет узнать где тратятся вычислительные ресурсы, а также изучить стек вызовов.

2. Concurrency-анализ - Происходит оценка эффективности параллельного кода.

3. Lock & Wait - анализ - Указывает на те места, где программа плохо распараллеливается.

Пройдя все эти этапы анализа, пользователь должен сформировать для себя определенное понимание поведения приложения в плане загрузки микропроцессора и эффективного использования его ресурсов. Далее на основе полученных результатов, можно решать дальнейшие шаги оптимизации программы.

Больше информации о Intel Parallel Amplifier `https://www.ixbt.com/soft/intel-parallel-amplifier.shtml`.