Соснин В.В., Балакшин П.В. Введение в параллельные вычисления. – СПб: Университет ИТМО, 2019. – 51 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 «09.01.04 – 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.

Рекомендовано к печати Ученым советом факультета компьютерных технологий и управления, 8 декабря 2015 года, протокол №10.**НЕТ!!!!**

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.

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 этом теоретический материал даётся тезисно и поэтому для полноценного освоения требуется использовать конспекты лекций по соответствующей дисциплине.

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 се-

бя 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 бакалаврской подготовки студентов по специальности "Программная инженерия"или смежных с ней. Однако приводимые примеры практических заданий могут быть при желании адаптированы для использования в магистерских курсах.

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.

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 1, 2, 4, 6, 8, 10, 12, 16 для обозначения количества ядер. Ширина области по вертикали равна относительной частоте использования процессоров соответствующего типа в рассматриваемом году.

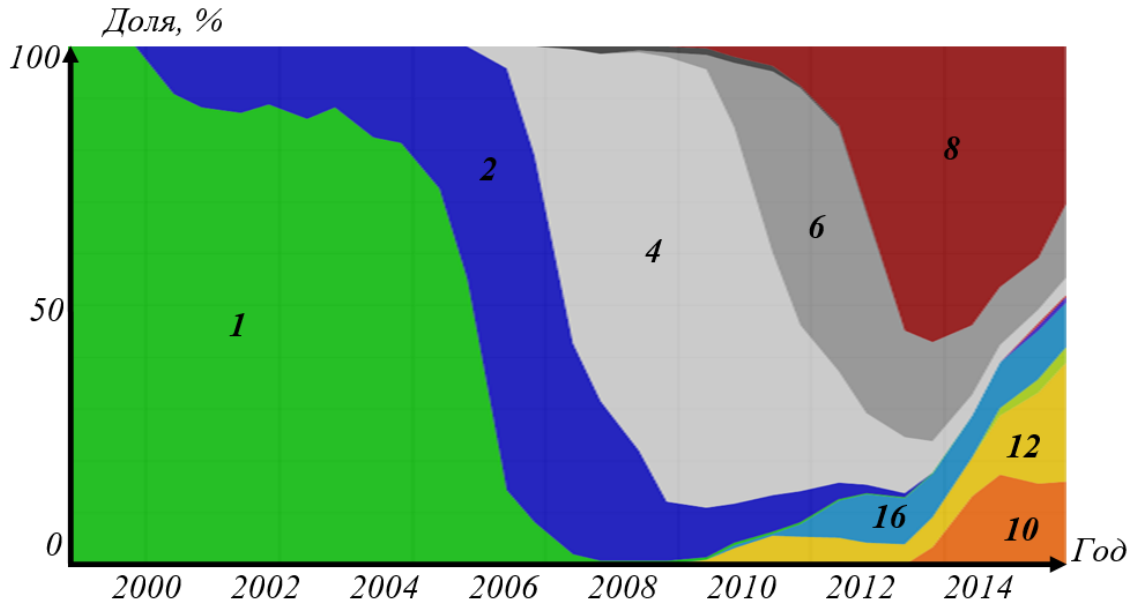

Рис. 1: Частотность использования процессоров с различным числом ядер при создании суперкомпьютеров

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 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 требу-

ется, 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 -O2 -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

1.3 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 двухядерного процессора, как наиболее сложный для понимания. Структурный алгоритм, изображенный на рисунке 2 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.

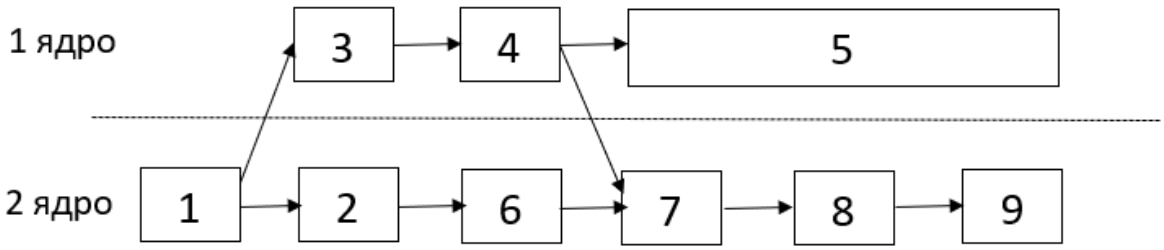

Рис. 2: Пример работы структурного алгоритма на двухъядерном процессоре

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 распараллелива-

ние 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.

1.4 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):

```
/*1*/        movl   w, %ecx
/*2*/        addl   $1, %ecx
/*3*/        movl   %ecx, w
```

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-разрадном 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:

```
/*1*/        uint64_t s;
/*2*/        s = 9999999999999L;
```

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

```
/*1*/        movabsq $9999999999999, %rsi
/*2*/        movq   %rsi, s
```

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.

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 нитей.

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 3 показано, что для настраиваемой виртуальной машины из восьми доступных физических (и логических) процессоров доступными являются только три.

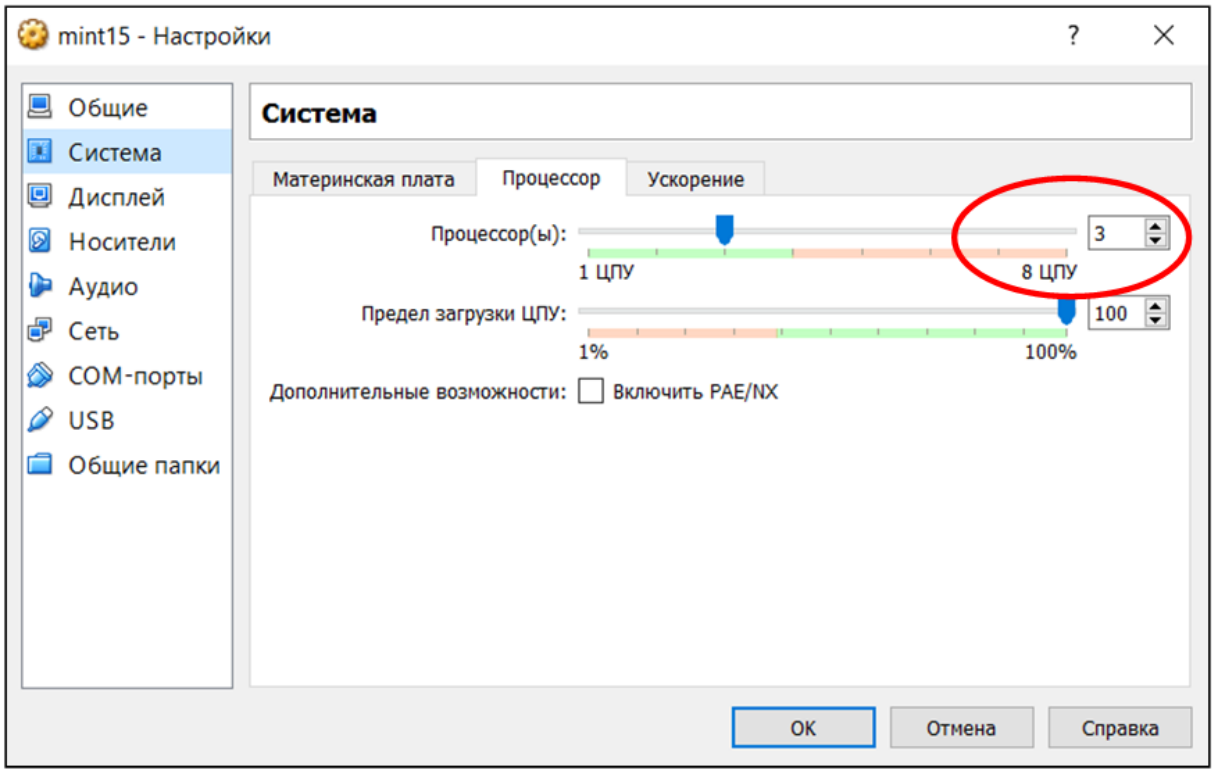

Рис. 3: Выбор количества виртуальных процессоров в Oracle VirtualBox

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 нитей. При создании параллельных программ достаточно часто количество создаваемых в процессе работы про-

граммы нитей 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 уве-

личении 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:

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

Запишем время $t(1)$ следующим образом:

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

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:

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

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

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

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 4:

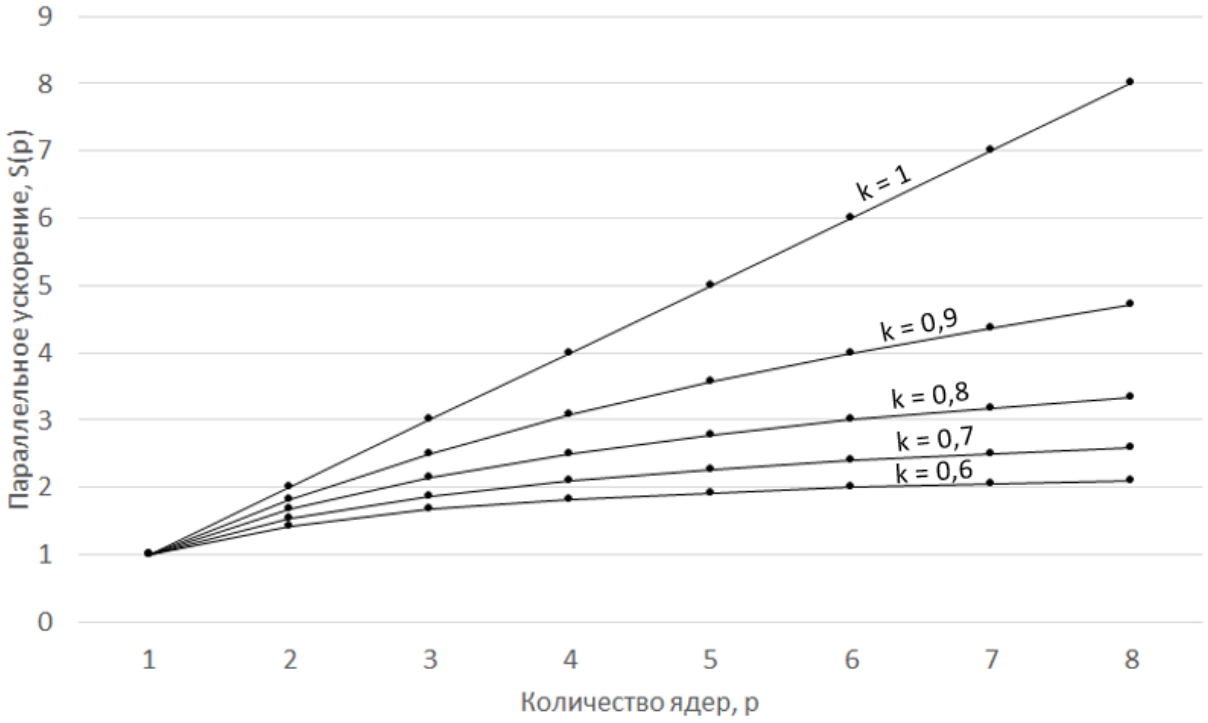

Рис. 4: График зависимости параллельного ускорения от количества ядер по Амдалу

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:

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

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

$$w(1) = w(1) + (k \cdot w(1) - k \cdot w(1)) = k \cdot w(1) + (1 - k) \cdot w(1), \quad (9)$$

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:

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

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

2.4 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 специфических 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:

```
/*1*/      for (i = 1; i < 382; i++) {
/*2*/            omp_set_num_threads(i);
/*3*/            T1 = omp_get_wtime();
/*4*/            #pragma omp parallel          /* начало параллельной области */
/*5*/            #pragma omp master
/*6*/            s++;                          /* конец параллельной области */
/*7*/            T2 = omp_get_wtime();
/*8*/            print_delta(T1, T2);
/*9*/      }
```

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 "3.3 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 5.

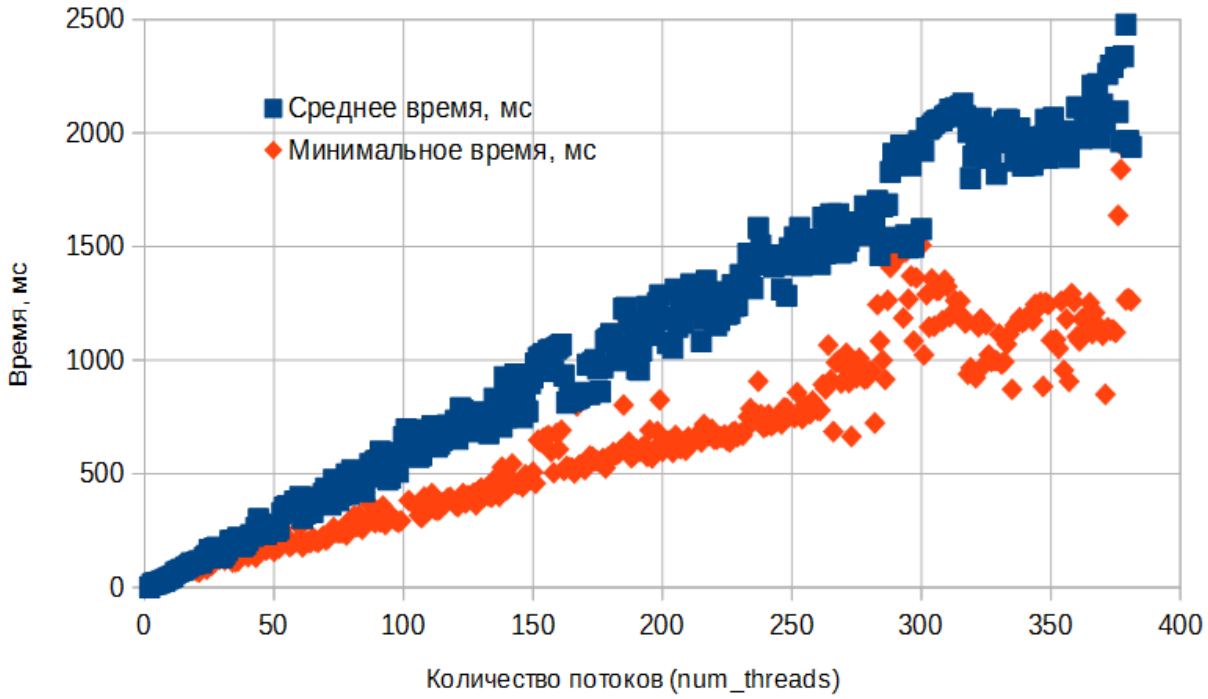

Рис. 5: Результаты измерения накладных расходов OpenMP при создании и удалении потоков

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.

3 Практические аспекты параллельного программирования

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.

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/`

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.

3.3 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 лишь немногие компиляторы поддерживают более новую версию OpenMP 4.0, 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).

Таблица 1: Ключи компиляторов для запуска OpenMP

Название компилятора	**Ключ компилятору для включения OpenMP**
Gcc	-fopenmp
icc (Intel C/C++ compiler)	-openmp
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:

```
/*1*/        #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 позволяет распараллеливать существующую последовательную программу в виде небольших итераций-правок, на каждой из которых будет достигаться всё больший коэффициент распараллеленности программы. Эта особенность является уникальной, т.к. большинство других технологий предполагают существенное изменение структуры распараллеливаемой программы уже на первом этапе процесса распараллеливания, при этом первая работоспособная параллельная версия программы появляется после длительного процесса отладки и программирования новых компонентов, которые неизбежно добавляются при распараллеливании. OpenMP лишён этого недостатка.

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"(лишь немногие компиляторы умеют генерировать эти заглушки при использовании специального ключа).

Высокий уровень абстракций. Одна единственная препроцессораня директива OpenMP после обработки компилятором приводит к существенной трансформации исходной программы с добавлением большого количества новой логики, отвечающей за определение доступного в системе количества процессоров, за запуск и уничтожение потоков, за распределение работы между потоками и т.п. Все эти операции OpenMP берёт на себя, взамен программист получает набор очень высокоуровневых инструментов распараллеливания. У высокоуровневых языков есть и традиционная тёмная сторона: в OpenMP отсутствует возможность изменить некоторую внутренние детали работы с потоками (например, нельзя установить аффинность потоков или уменьшить накладные расходы на создание/удаление потоков).

Низкий коэффициент параллельной трансформации (КПТ). При распараллеливании существующей последовательной программы приходится вносить в неё достаточно большое количество изменений. Пусть КПТ – это отношение строк нового программного кода, который добавился в результате распараллеливания, к общему количеству 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 4.5 наиболее популярные компиляторы уже поддерживают версию OpenMP 4.0. Исключением является только фирма 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 количеством создаваемых потоков при выполнении параллельной работы. Это обеспечивает возможность гибко управлять и

настраивать процесс создания потоков в зависимости от количества доступных системе процессоров (ядер), но при этом требует от программиста большой количества неавтоматизируемой работы. В OpenMP управление масштабированием происходит в автоматическом режиме, т.е. OpenMP сам запрашивает у операционной системы количество доступных процессоров и выбирает количество создаваемых потоков. Но при необходимости OpenMP оставляет возможность устанавливать требуемое количество потоков вручную.

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

```
/*1*/    #pragma omp parallel
/*2*/    printf("Hello, world!");
```

Результатом работы будет выведенное несколько раз в консоль сообщение. Количество сообщений определяется количеством логических процессоров, доступных системе (например, при использовании технологии HyperThreading при двух ядрах количество 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 количество операций в фигурные скобки, чтобы расширить исполняемый блок:

```
/*1*/    int i = 1;
/*2*/    #pragma omp parallel
/*3*/    {
/*4*/        printf("Hello, world!");
/*5*/        #pragma omp atomic
/*6*/        i++;
/*7*/    }
```

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 необ-

ходима, 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:

```
/*1*/    int i = 1;
/*2*/    #pragma omp parallel
/*3*/    {
/*4*/        printf("Hello, world!");
/*5*/        #pragma omp critical
/*6*/        {
/*7*/            i++;
/*8*/            printf("i=%d\n", i);
/*9*/        }
/*10*/   }
```

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:

```
/*1*/        #pragma omp parallel sections
/*2*/        {
/*3*/              #pragma omp section
/*4*/              run_function1();
/*5*/              #pragma omp section
/*6*/              run_function2();
/*7*/        }
```

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

```
/*1*/        int i;
/*2*/        #pragma omp parallel for
/*3*/        for (i = 0; i < 10; ++i) {
/*4*/                a[i] = i*i*i*i*i*i;
/*5*/        }
```

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 количество операций умножения для возведения в шестую степень:

```
/*1*/        int i, tmp;
/*2*/        #pragma omp parallel for
/*3*/        for (i = 0; i < 10; ++i) {
/*4*/                tmp = i*i*i;          /*при попытке сделать оптимизацию */
/*5*/                a[i] = tmp*tmp;  /* массив заполнен с ошибками */
/*6*/        }
```

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 превратиться в 7*7*7, а значение a[2], рассчитываемое первым потоком, превратиться в 7^6, вместо положенных 64. Исправим

допущенную ошибку следующим образом:

```
/*1*/      int i, tmp;
/*2*/      #pragma omp parallel for private(tmp)
/*3*/      for (i = 0; i < 10; ++i) {
/*4*/              tmp = i*i*i;
/*5*/              a[i] = tmp*tmp;
/*6*/      }
```

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 хочется явным образом указать на глобальность переменной, следует рядом с командой private использовать команду shared(x,. . .), где x задаёт список глобальных переменных.

Рассмотрим пример, в котором нужно рассчитать сумму и произведение элементов следующего ряда: $\{ 1^i,\ 2^i,\ 3^i,\ 4^i,\ 5^i \}$ для различных значений i, например: $i = 1,\ 2,\ 3$. Приведём ниже решение поставленной задачи, но умышленно допустим в ней ошибку:

```
/*1*/      int i, j, sum[3], product[3], tmp[5];
/*2*/      #pragma omp parallel for private(tmp)                  /* !ошибка! */
/*3*/      for (i = 0; i < 3; ++i) {
/*4*/              for (j = 1; j <= 5; ++j) tmp[j] = pow(j, i);      /* !ошибка! */
/*5*/              sum[i] = calculate_sum(tmp, 5);
/*6*/              product[i] = calculate_product(tmp, 5);
/*7*/      }
```

В строке 2 происходит запуск параллельной области, но программист забывает указать, что переменные j и массив tmp должны быть локальными для каждого треда. Действительно, в строке 10 происходит ин-

кремент общей для потоков переменной j, который выполняется всеми потоками одновременно. В этой ситуации потоки могут помешать друг другу, переписав чужое значение j. Исправим обе ошибки следующим образом:

```
/*1*/      int i, j, sum[3], product[3];
/*2*/      #pragma omp parallel for private(j)                    /* ок */
/*3*/      for (i = 0; i < 3; ++i) {
/*4*/              int tmp[5];
/*5*/              for (j = 1; j <= 5; ++j) tmp[j] = pow(j, i);    /* ок */
/*6*/              sum[i] = calculate_sum(tmp, 5);
/*7*/              product[i] = calculate_product(tmp, 5);
/*8*/      }
```

Видим, что теперь переменная j явным образом обозначена локальной (private). С массивом tmp решение другое – он весь помещается внутрь параллельной области (т.е. у каждого потока будет свой собственный не зависимый от других экземпляр массива tmp). Почему же нельзя было просто указать переменную tmp в перечне команды private, как это было сделано для j? Ответ связан со спецификой языка С: переменная tmp является указателем, который при работе цикла не меняется, но меняется содержимое памяти, на которое указывает tmp. Это значит, что указыание tmp в качестве private-переменной не решило бы проблему с гонками данных, т.к. все потоки получили бы один и тот же адрес tmp и мешали бы друг другу, записывая новые значения по этому адресу.

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

```
/*1*/      int i, sum = 0;
/*2*/      #pragma omp parallel for
/*3*/      for (i = 0; i < 100; ++i) sum += i;  /* ошибка */
```

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:

```
/*1*/    int i, sum = 0, sum_private = 0;
/*2*/    #pragma omp parallel private (sum_private)
/*3*/    {
/*4*/      sum_private = 0;     /* повторная инициализация! */
/*5*/      #pragma omp for
/*6*/      for (i = 0; i < 100; ++i) sum_private += i;
/*7*/      #pragma omp atomic
/*8*/      sum += sum_private;
/*9*/    }
```

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:

```
/*1*/    int i, sum = 0;
/*2*/    #pragma omp parallel for reduction (+:sum)
/*3*/    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 "*", "-", "/". 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 умножение – едини-

цей.

При распараллеливании цикла может оказаться, что итерации неравноценные по количеству выполняемой работы между собой. Это может привести к тому, что один поток справится с выделенной половиной итераций намного быстрее второго потока и будет простаивать. Для решения этой проблемы OpenMP предлагает четыре разных способа распределения итераций по потокам.

- *Способ по умолчанию:* при этом итерации делятся на количество частей, равное количеству потоков; каждый поток выполняет после этого свою часть и не может взять чужую работу.
- *Статическое распределение (static):* итерации разбиваются на части указанного пользователям размера; затем ещё до начала работы каждый поток получает фиксированное количество частей и выполняет только их без возможности переключиться на другие.
- *Динамическое распределение (dynamic):* итерации разбиваются на части указанного пользователям размера; затем сразу начинается работа цикла и каждый поток получает новую часть итераций по мере завершения работы над предыдущей.
- *Управляемое распределение (guided):* компилятор разбивает итерации на количество частей, равное удвоенному количеству потоков; затем сразу начинается работа цикла и каждый поток получает новую часть итераций по мере завершения работы над предыдущей, при этом размер нововыданной части уменьшается по сравнению с предыдущим разом, но не может стать меньше указанного пользователем константного значения.

Упомянутый в каждом из методов пользовательский параметр называется chunk_size. Каждый из указанных методов имеет свою область применения, в которой он может обеспечить максимальное параллельное ускорение. Отметим, что режимы dynamic и guided несмотря на свою логичность имеют и свои недостатки: они требуют существенных накладных расходов во время работы цикла по сравнению со static. Также важно понимать, что при выборе числа

Рассмотрим пример статического распределения итераций:

```
/*1*/    int i; double sum = 0;
/*2*/    #pragma omp parallel for reduction (+:sum) schedule(static,1)
/*3*/    for (i = 1; i < 100; ++i) sum += 1.0/i;
```

При наличии трёх ядер OpenMP создаст три потока. Первому потоку достанутся итерации $i = 1,\ 4,\ 7,\ \ldots,\ 97$ второму – итерации $i = 2,\ 5,\ 8,\ \ldots,\ 98$, третьему – итерации $i = 3,\ 6,\ 9,\ \ldots,\ 99$. Обратим внимание, что выбор малого значения параметра chunk_size = 1 в данном случае не имеет каких-либо негативных эффектов. Однако если бы i использовалась в качестве индекса при обращении к массиву, то предложенный вариант разбиения привёл бы к обращению в память не подряд по последовательным адресам, а разреженно с шагом 3, что ухудшило бы показатели cache hit при использовании кэширования.

Рассмотрим ещё один пример:

```
/*1*/    double result1, result2, result3;
/*2*/    #pragma omp parallel num_threads(3)
/*3*/    {
/*4*/        #pragma omp for reduction (+:result1) nowait
/*5*/        for (i = 0; i < 100; ++i) result1 += i;
/*6*/        #pragma omp sections
/*7*/        {
/*8*/            #pragma omp section
/*9*/            result2 = calculate_pi();
/*10*/           #pragma omp section
/*11*/           result3 = calculate_e();
/*12*/       }
/*13*/   }
/*14*/   use_results(result1, result2, result3);
```

Здесь приводится пример, как можно указать OpenMP количество создаваемых потоков с помощью опции num_threads (строка 2), не ориентируясь на реально доступное количество ядер (процессоров) на компьютере. Далее три созданных потока делят между собой 100 итераций уже знакомым нам способом. Однако опция nowait позволяет первому справившемуся с работой потоку не дожидаться остальных, а перейти к следующей за циклом работе. За циклом в параллельном режиме выполняются две функции (строки 9 и 11). Каждая из функций заключена в секцию (section), которые должны иметь родительский элемент sections. В итоге первый освободившийся после цикла поток займётся вычислением функции в строке 9. Второй освободившийся поток вычислит функцию в строке 11. Третьему потоку не достанется работы помимо своей

доли итераций в первом цикле. Общим требованием OpenMP к распараллеливаемым циклам является их *каноничность*. Цикл for называется *каноническим*, если можно при его начале заранее рассчитать количество предстоящих итераций. Это возможно, если одновременно выполняются следующие условия:

- внутри цикла нет операций break и return;
- внутри цикла нет операции goto, ведущей вовне цикла;
- переменная цикла (итератор) не изменяется внутри цикла;

При этом запись цикла должна иметь вид "for (i = A; i < B; i+=C)", где числа A, B, C не должны меняться во время работы цикла. При этом второй параметр цикла может использовать не только знак "меньше но и ">", ">=", "<=". Третий параметр цикла может не только инкрементировать, но декрементировать переменную цикла (допускается краткая форма записи "i++").

Если итерация k влияет на результаты итерации m, то цикл нельзя распараллеливать, т.к. нельзя заранее предсказать порядок завершения итераций несколькими потоками. Ответственность за обнаружение таких конфликтов лежит на программисте. Например, OpenMP не обнаружит взаимозависимость итераций и скомпилирует следующую программу:

```
/*1*/    #pragma omp parallel for num_threads(2)
/*2*/    for(i = 1; i < 20; i++)  a[i] = 2*a[i – 1];
```

В этой программе поток 0 скорее всего не успеет заполнить элемент a[9] к тому моменту, когда поток 1 будет вычислять значение a[10] = 2*a[9].

3.4 Ошибки в многопоточных приложениях

Помимо привычных для программиста ошибок, встречающихся в компьютерных программах, существует ряд ошибок, специфичных для параллельного программирования. Эти ошибки обусловлены следующими особенностями параллельных программ:

- **Синхронизация потоков.** Программист должен обеспечить корректную последовательность выполняемых разными потоками операций. В общем случае невозможно точно сказать, в какой последовательности будут выполняться команды потоков, т.к. опе-

рационная система может в произвольный момент времени при остановить выполнение потока.

- **Взаимодействие потоков.** Также программист не должен конфликтов при обращении к общим для потоков областям памяти.
- **Балансировка нагрузки.** Если в распараллеленной программе один из потоков выполняет 99% работы, то даже на 64-ядерной системе параллельное ускорение едва ли превысит значение 1.01.
- **Масштабируемость.** В идеале параллельная программа должна одинаково хорошо распараллеливать выполняемую работу на любом доступном количестве процессоров. Однако добиться этого нелегко и это часто приводит к трудно обнаруживаемым ошибкам.

Рассмотрим далее подробнее следующий неполный перечень типовых ошибок, возникающих в параллельных программах независимо от используемой технологии распараллеливания:

- Потеря точности операций с плавающей точкой.
- Взаимные блокировки (deadlock).
- Состояния гонки (race conditions).
- Проблема ABA.
- Инверсия приоритетов.

Потеря точности. Если параллельная программа используется для проведения операций с плавающей точкой при работе с вещественными переменными, расположенными в общей для потоков памяти, то при каждом запуске программы может получаться разный результат вещественных расчётов. Это объясняется тем, что при работе нескольких потоков невозможно точно предсказать, в каком порядке операционная система предоставит этим потокам процессор, т.к. в любой момент любой поток может быть временно приостановлен по усмотрению ОС. Это в свою очередь приводит к неопределённой последовательности выполнения операций с плавающей точкой, результат которых, как известно, может зависеть от порядка.

Рассмотрим пример, иллюстрирующий сказанное:

```
/*1*/    int i;
/*2*/    float s = 0;
/*3*/    #pragma omp parallel for reduction (+:s) num_threads(8)
/*4*/    for (i = 1; i < 1000000; ++i) {
/*5*/       s += 1.0/i;
/*6*/    }
/*7*/    printf("s=%f\n", s);
```

Здесь в переменную s суммируются результаты вещественных вычислений восемью потоками. В результат получается s=14.393189. Однако если эту же программу выполняет всего один поток (для этого нужно в строке 3 установить значение параметра num_threads в 1), то результат получится иным: s=14.357357. Различие между двумя приведёнными значениями составляет примерно 0.25%.

Получается, что параллельная программа может давать разный результат при запуске на разных платформах. Это следует учитывать, проводя верификацию параллельных программ с использованием однопоточных их нераспараллеленных аналогов.

Взаимные блокировки. Одним из часто используемых примитивов синхронизации является мьютекс, позволяющий нескольким потокам согласованно и последовательно выполнять критические области кода, расположенные внутри параллельных секций кода. Критические секции замедляют работу программы, т.к. в каждый момент времени только один поток может находиться внутри критической секции. С помощью мьютексов, например, реализуются функции omp_set_lock и omp_unset_lock в OpenMP. При обрамлении этими функциями некоторого участка кода можно сделать из него критическую секцию, вход в которую контролируется условным программным замком (lock). В сложных программах может использовать несколько замков. Это может привести к тому, что два потока, захватывающие несколько замков, застопорят выполнение друг друга без всякой возможности выйти из состояния ожидание друг друга. Такая ситуация называется deadlock (взаимная блокировка).

Простейшим примером взаимной блокировки является работа двух потоков, первый из которых захватывает сначала замок1, потом замок2, а второй сначала захватывает замок2, потом замок 1. В результате, возникнет deadlock, если операции будут выполняться в следующем порядке:

- поток1 захватил замок1;
- поток2 захватил замок2;

- поток1 бесконечно ждёт освобождения замка 2;
- поток2 бесконечно ждёт освобождения замка 1.

Одна из неприятных сторон описанной ситуации заключается в том, что далеко не всегда взаимная блокировка происходит при отладке программы, когда её можно было бы легко выявить и исправить, т.к. вероятность наложение событий нужным образом может быть очень мала. В результате работающая и сданная заказчику программа может в случайные моменты времени "зависать" по якобы непонятным причинам. Рассмотрим пример искусственно реализованной взаимоблокировки, в котором можно рассчитать вероятность её возникновения при многократном запуске.

В приведённой ниже программе в строке 7 создаётся поток, который в бесконечном цикле захватывает замок1, замок2 и инкрементирует переменную s, освобождая после этого оба замка. В строке 13 создаётся поток, которой тоже бесконечно инкрементирует s, однако захватывает замки в другом порядке: замок2, замок1. В строке 19 создаётся поток, который следит за состоянием s, опрашивая эту переменную каждые 10 мс. Если последний поток обнаруживает, что переменная s перестала изменяться, он печатает сообщение о возникшей взаимоблокировке и завершает программу.

```
/*1*/     int old_s, s = 0;
/*2*/     omp_lock_t lock1, lock2;
/*3*/     omp_init_lock(&lock1);  /* инициализация замка1 */
/*4*/     omp_init_lock(&lock2);  /* инициализация замка2 */
/*5*/     #pragma omp parallel sections
/*6*/     {
/*7*/         #pragma omp section
/*8*/         for (;;) {
/*9*/                 omp_set_lock(&lock1);       omp_set_lock(&lock2);
/*10*/                s++;
/*11*/                omp_unset_lock(&lock2);   omp_unset_lock(&lock1);
/*12*/        }
/*13*/        #pragma omp section
/*14*/         for (;;) {
/*15*/                omp_set_lock(&lock2);       omp_set_lock(&lock1);
/*16*/                 s++;
/*17*/                omp_unset_lock(&lock1);   omp_unset_lock(&lock2);
/*18*/      }
/*19*/      #pragma omp section
/*20*/      {
/*21*/        for(old_s = !s; old_s != s; old_s = s) usleep(10000);
/*22*/        printf("Обнаружена взаимоблокировка при s=%i\n", s);
/*23*/        omp_destroy_lock(&lock1);     omp_destroy_lock(&lock2);
/*24*/        exit(0);
/*25*/      }
/*26*/    }
```

Эксперименты с приведённой программой проводились на компьютере с процессором Intel Core i5 (4 логических процессора) с 8 гигабайт ОЗУ в операционной системе Debian Wheezy. Программы была запущена 10000 раз, и было получено 10000 значений переменной s на момент возникновения взаимоблокировки. Результаты этих измерения приведены на рисунке 6 в виде гистограммы плотности распределения s.

На приведённом рисунке на оси абсцисс подписаны правые границы столбиков. Последний столбик содержит все попадания от 3000 до бесконечности. Среднее значение s в описанном случае оказалось равным 2445, т.е. два потока успевают примерно 1222.5 раза захватить и отпустить замки в заведомо неверном опасном порядке без возникновения взаимоблокировки.

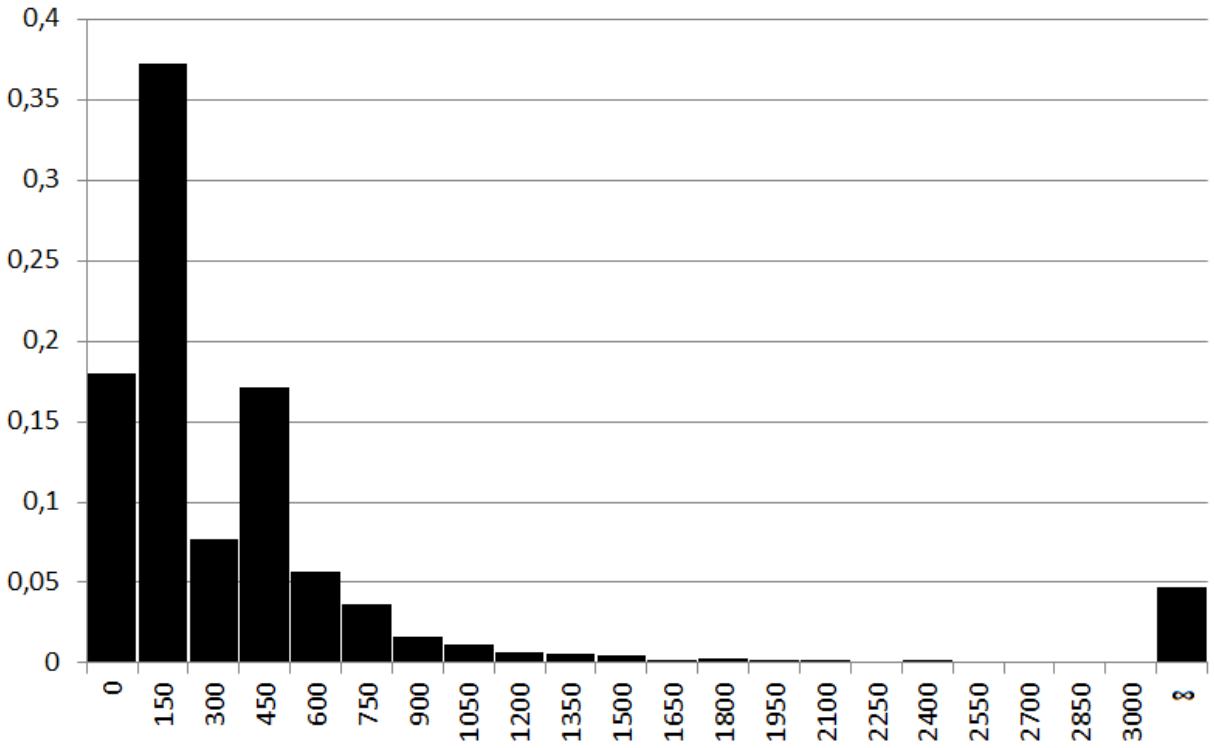

Рис. 6: Гистограмма распределения количества запусков параллельной программ до возникновения взаимоблокировки

Для исправления описанной ошибки нужно сделать порядок захвата замков одинаковым во всех потоках. Иногда советуют во всей программе установить некоторое общее правило захвата замков, например, можно захватывать замки в алфавитном порядке.

Помимо описанной ситуации с неверным порядком захвата мьютексов, существуют и другие причины взаимоблокировок. Например, повторный захват мьютекса (замка). Неверно написанная программа может попытаться повторно захватить уже захваченный ею замок, предварительно его не освободив. При этом повторная попытка захвата полностью останавливает работу потока. Если логика программы требует повторный захват мьютекса (например, для организации рекурсии), следует использовать специальный подвид замков: рекурсивные мьютексы.