

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.

© Университет ИТМО, 2019

© Соснин В.В., Балакшин П.В., 2019

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, посвящённых 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 было 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 получила 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.

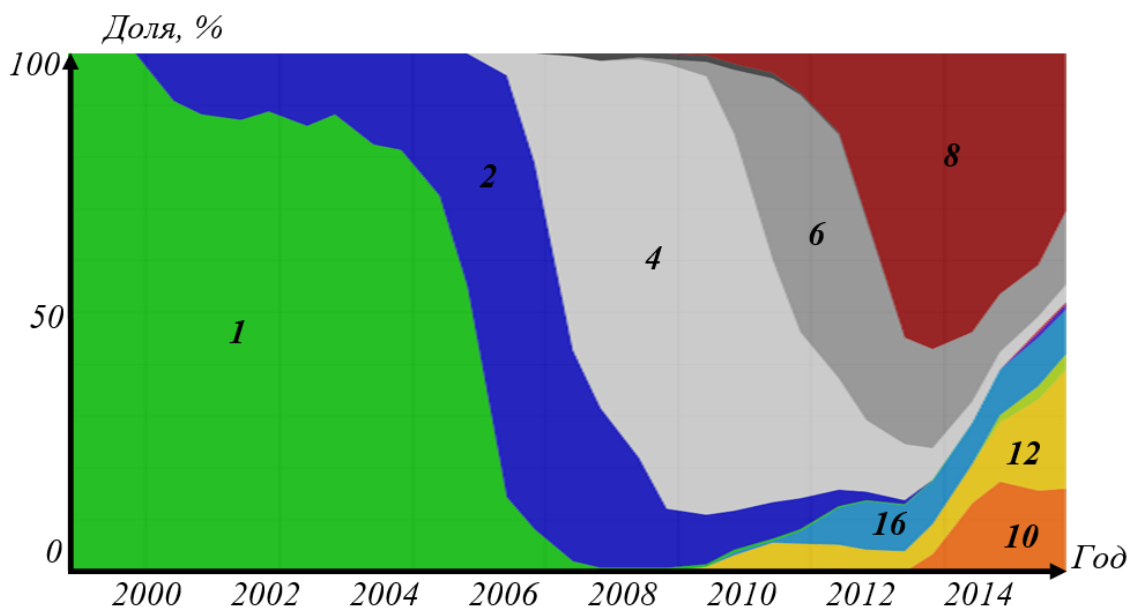


Рис. 1: Частотность использования процессоров с различным числом ядер при создании суперкомпьютеров

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). Особенности 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, 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.

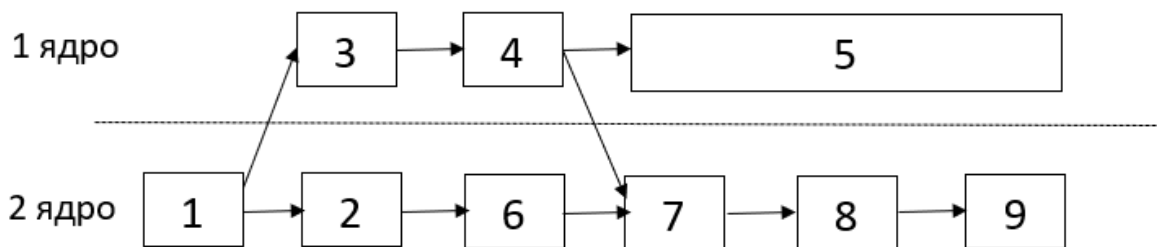


Рис. 2: Пример работы структурного алгоритма на двухъядерном процессоре

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).

Таковыми 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 C программы команды языка C транслируются обычно в несколько ассемблерных инструкций. В связи с этим возникает вопрос о возможном существовании C-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):

<code>/*1*/</code>	<code>movl w, %ecx</code>
<code>/*2*/</code>	<code>addl \$1, %ecx</code>
<code>/*3*/</code>	<code>movl %ecx, w</code>

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:

```
/*1*/      uint64_t s;  
/*2*/      s = 99999999999999L;
```

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

```
/*1*/      movabsq $99999999999999, %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.