Теория и практика многопоточного программирования.

А. Тормасов, С. Бабичев

2012-2020

Лекция 1
Введение.

Содержание

- Историческое развитие вычислительных систем.
- Совершенствование вычислительных систем
- Современное состояние дел.
- Чему обучают обычно и что изучать нужно.
- Что вы уже знаете и чему должны научиться.

Историческое развитие вычислительных систем.

- Начало истории — 1960-х годы.
- Массовое использование ЯВУ Фортран/Алгол.
- Первые пользователи ЭВМ — программисты.
  - Первая группа сопротивляется введению ЯВУ.
  - Вторая группа заявляла, что время программистов тоже чего-то стоит.
    Уменьшение производительности программ $\rightarrow$ увеличение производительности программистов.

- 1970-е годы.
- Компиляторы производят ужасный код.
- Системных программистов много меньше, чем проблемных. Именно системным программистам важно знание архитектуры. Они пишут компиляторы, операционные системы, библиотеки.

- Производители аппаратуры (процессоров) не обращают внимания на программистов.
- В системе команд IBM System/360 нет команд работы со стеком, с удобной косвенной адресацией.
- Защита памяти в зачаточном состоянии.
- Нет виртуальной адресации.
- Есть команды для работы с десятичным представлением BCD.

- Производители процессоров начали добавлять в систему команд поддержку языков высокого уровня.
- Две известные архитектуры
  - DEC VAX-11 регистрового типа.
  - Burrough (Эльбрус) стекового типа с аппаратными тегами.

- Машинные команды сложные (вычисление CRC32 или вычисление значения полинома по схеме Горнера).
- Каждая команд исполнялась значительное количество тактов и действия по спецификации этой команды часто излишни.
- Если в спецификации *ADD* указывалось, что она должна в зависимости от результата установить флаги процессора Z, C, S, то эти флаги изменяются обязательно даже здесь:

```
add d1,d2
add d1,[a1+4]
add d1,[a1+8]
```

  в котором результаты промежуточных операций никого не интересовали.
- Каждая команда кодировалась различным числом битов и передача аргументов происходила только после определения команды по её коду и всем модификаторам, что задерживало исполнительный конвейер.

Ряд производителей ввели новую архитектуру, которой быстро нашлась новая аббревиатура — *RISC — Reduced Instruction Set Computer*, старые теперь называются *CISC — Complex Instruction Set Computer*.

Основные принципы *RISC*:

- Все машинные команды занимают одинаковое число битов.
- Имеется много регистров.
- В память можно занести значение регистра и из памяти можно загрузить регистр. Других операций с памятью нет!
- Все операции производятся только с регистрами.
- Имеется необходимый минимум машинных команд.
- Все команды, кроме строго ограниченного набора, исполняются ровно один такт.

- Пример: процессор **SPARC**.
- Команда перехода (и вызова функции) завершалась через два такта после её запуска и существовал так называемый *delay slot*, в который можно было поместить ещё одну машинную команду.

```
; func(1)
call func
mov     1,%o0   ; delay slot
```

- Эффективное использование **RISC**-компьютеров невозможно без использования хороших компиляторов.

- 80-е годы. Язык Си вытесняет язык ассемблера из системного программирования.
- Ассемблер остался для тех компонент, для которых не написан генератор кода (MMX, SSE, AltiVec).

Совершенствование вычислительных систем

- Вначале прогресс в переходе на более быстродействующие (и более надёжные) элементы — транзисторы пришли на смену электронным лампам, интегральные микросхемы на смену одиночным транзисторам.
- Размеры процессорных блоков уменьшались, задержки в передаче сигнала тоже уменьшались, тактовые частоты увеличивались.
- В 1967-м году выпуск БЭСМ-6. Элементная база очень слабая, но производительность высока.

Ключевые факторы:

- Конвейерный принцип исполнения команд (разработчики назвали это водопроводным принципом) — от начала исполнения конкретной команды до её завершения проходило достаточно много времени, но одновременно исполнялись несколько команд на разных этапах.
- Параллельный (восьмиканальный) доступ к памяти. При большом времени доступа к одному блоку, примерно в две микросекунды, эффективное время уменьшалось в восемь раз.
- Кэширование запросов к памяти за операндами и кодом программы. Это помогало сэкономить до 60% обращений к памяти.

В настоящее время эти способы увеличения производительности общеприняты.

Нельзя увеличить тактовую частоту — будем:

- параллельно исполнять фрагменты машинной команды.
- увеличим ширину обрабатываемых данных — размер «слова» 36, 45, 48, 60 бит.

Язык Паскаль впервые разрабатывался для CDC-6000 с размером слова в 60 бит. В слове хранилось 10 символов. Обработка одного символа требовала помещения его в 60-битный регистр процессора. Никлаус Вирт придумал атрибут **packed** для строк, состоящих из символов.

```
type
   alpha: packed array[1..10] of char;
var
   a,b: alpha;
begin
   if a = 'Test      ' then begin
      ...
   end;
...
```

- Обработка текстовой информации всё нужнее.
- Обратная реакция от производителей ЭВМ. В набор команд добавились команды работы с байтами.
- Не всем требуются числовые молотилки, имеются и экономические задачи, и задачи обработки текста.
- Создание мини-ЭВМ, а затем и микро-ЭВМ.
- История: Intel: то первый масссовый микропроцессор 8080 — восьмиразрядный.
- Всё делается для удешевления производства.
- Дешёвое производство — массовый выпуск микро-ЭВМ.
- Малоразрядный процессор — низкая производительность.
- Увеличение производительности — увеличение разрядности $8 \to 16 \to 32 \to 64$, даже на телефонах.

Следующий этап — увеличение числа исполнителей машинного кода, вычислительных ядер.

Модель	Год	Транз млн	мкм	MHz	Бит	IPC	Ядер	TDP Ватт
8080	1974	0.005	6	4	8	0.05	1	1.5
8086	1978	0.029	3	8	16	0.07	1	3
80286	1981	0.134	1.5	16	16	0.15	1	3
80386	1985	0.375	1	40	32	0.33	1	4
i486	1989	1.2	1	100	32	0.7	1	6
Pentium	1993	3.3	0.8	200	32	1.2	1	15
Pentium Pro	1995	5.5	0.6	200	32	1.6	1	20
Pentium III	1999	9.5-28.1	0.18	1400	32	1.7	1	40
NetBurst	2001	42-376	0.065	4000	64	1.1	2	105
Nehalem	2006	167-820	0.045	3333	64	1.9	8	130
SandyBridge	2011	664-995	0.032	3400	64	2.2	8	130
IvyBridge	2012	1000	0.022	3800	64	2.6	15	130
Haswell	2013	1400	0.022	3800	64	2.7	18	175

Исторический обзор процессоров Intel.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

1. Повышение тактовой частоты $\rightarrow$ повышение мощности $\rightarrow$ проблема теплоотдачи. Суперкомпьютеры Cray-1 использовали фреон для охлаждения.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

2. Непреодолимость скорости света.
   Для подключения компонентов друг к другу, в Cray-1 использовались соединители, длиной до 180 сантиметров.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

Тактовая частота Cray-1 80 мегагерц, 12.5 наносекунд. За 12.5 наносекунд свет пройдёт 3.75 метра и длины соединителей хватало.
Нужно учесть, что электромагнитное излучение в применяемых проводниках распространяется со скоростью, равной примерно 2/3 скорости света.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

Следующая модель, Cray-2. Тактовая частота 250 мегагерц, такт 4 наносекунды, время прохождения 1.2 метра.
Длина соединителей 40 сантиметров, теплоотвод стал тяжелее.
Все процессорные элементы со всеми соединителями погружались в инертную жидкость.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

Современные процессоры: тактом до 0.15 наносекунд, характеристическая длина составляет 4.5 сантиметра. Сильнейшие ограничения в проектирование как самих процессоров, так и их окружения. Меньшие расстояния — сложнее теплоотвод.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

3. Отсутствие роста скорости памяти (латентность + пропускная способность).

   В современных микрокомпьютерах применяется динамическая память, которая обладает достаточно большой латентностью.

   - 1995 год: типичная латентность элемента памяти 60 наносекунд. Пропускная способность 132 МБ/сек.
   - 2020 год: типичная латентность элемента памяти 50 наносекунд. Пропускная способность 80000 МБ/сек.

Проблемы, которые ограничивают дальнейшее ускорение отдельных вычислительных ядер

4. Количество инструкций, исполняемых в секунду одним ядром почти перестало увеличиваться.
   Наращивать этот параметр можно только серьёзно изменив архитектуру и систему команд.

   Типичные алгоритмы содержат много операций сравнения и, перехода. При конвейерной организации исполнения это оказывается сильнейшим тормозом в увеличении показателя IPC.

Что делать?

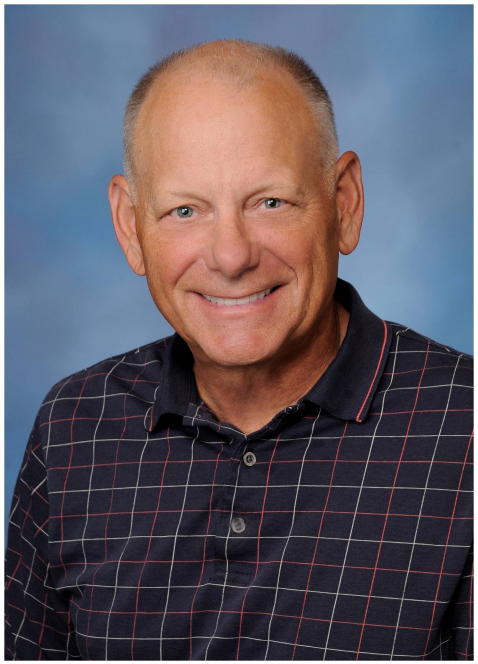
Будем рассматривать вычислительную систему как совокупность аппаратно-программных компонент. В аппаратные компоненты входят

- вычислительные ядра
- оперативная память
- кэш-память

К программным компонентам относятся

- операционные системы
- компиляторы
- библиотеки компилятора.

- Наращивание вычислительных ресурсов требует поддержки от операционных систем и производителей компиляторов и библиотек.
- Бессмысленно устанавливать Windows95 единственной операционной системой на современный компьютер.
- Windows95 - запланированный тупик в проектировании ОС.
- В 1988 году эта же фирма пригласила Дейва Катлера



для проектирования WindowsNT, в которую изначально были заложены многопроцессорность и многопоточность.

Современное состояние дел.

- Перед нами цель: быстрейшее выполнение специфической задачи.
- Задача требует много вычислительных ресурсов.
- По нашему мнению, мы можем ускорить её исполнение, используя особенности платформы.
- Что нам предоставляет производитель вычислительной системы для этих целей?

- Производитель операционной системы предоставляет разрешение процессам создавать вычислительные потоки и использовать все доступные вычислительные ядра.
- Производитель оборудования предоставляет эти вычислительные ядра операционной системе.
- Применяется конвейерная архитектура $\rightarrow$ существуют моменты, когда конвейеру требуется очередная порция данных для обработки, а данных пока нет.
- Не использовать ли пока простаивающие исполнительные узлы конвейера в других потоках, имитируя ещё один процессор? HyperThreading, HT.
- Появилось при реализации процессора Pentium IV, который имел настолько длинный конвейер, что ему всегда что-то мешало.
- Имитация второго процессора позволила иногда увеличить производительность до 30-40%.
- Программист получил в своё распоряжение много виртуальных вычислительных ядер. Что с ними делать — забота программиста.

- Программам предоставляется виртуальная память практически неограниченного размера. Возникает заблуждение, что с памятью можно работать как угодно. Об этом мы поговорим на второй лекции.
- Производитель может предоставить программистам векторные команды (SIMD — Single Instruction Multiple Data). Начало для Intel — Pentium MMX, 8 64-битных регистров.

  $64 = 1 \times 64$ бита $= 2 \times 32$ бита $= 4 \times 16$ битов $= 8 \times 8$ битов

- Но существующие алгоритмы не приспособлены для векторизации и компиляторы пока не способны векторизировать код.

- Разрешить программисту использовать внешние по отношению к процессору устройства, работающие одновременно с процессором. Управлять ими командами ввода/вывода или аппаратными прерываниями.
- Первый 16-битный процессор Intel — 8086 не имел команд работы с вещественной арифметикой.
- Отдельно продавалась микросхема 8087 для аппаратной реализации команд вещественной арифметики — сопроцессор.
- Turbo Pascal генерировал вызовы функций для эмуляции вещественной арифметики целочисленной или код для сопроцессора.
- Microsoft C генерировал универсальный самомодифицирующийся код.
- Сопроцессор ускорял выполнение до 100 раз.

- Продавались платы с установленными *транспьютерами* и локальной для них памятью. Можно было загрузить программу и выполнить её на транспьютерах параллельно с основной через команды ввода/вывода.
- Сейчас мы используем мощь графических карт для исполнения массовой параллельной обработки. Созданы языки Cuda и OpenCL.

Что сейчас происходит с традиционными последовательными программами?

- Их КПД на современных компьютерах крайне низок. На 16-х ядерном сервере наилучшая последовательная программа утилизирует только 6.25% от вычислительной мощности.
- Выход — параллельный запуск множества таких программ (MPI, GRID). Такие задачи не так часто встречаются.

Чему обучают обычно и что изучать нужно.

Традиционные курсы алгоритмов

- **Последовательное программирование.** Все алгоритмы и структуры данных в традиционном программирование изложены последовательным образом. Основные исполняющие структуры - *if-then*, *if-then-else*, *while*, *for*, *procedure/function* последовательны по своей сути. Это помогает при обучении программированию, но это же становится тормозом при создании эффективных программ.
- **Монопольное использование всех ресурсов.** Элементы массива не изменяются самостоятельно. Элемент изменяется только при наличии его в левой части. Это — прекрасная база для верификации программ но это не позволит нам увидеть все возможные проблемы в многопоточном окружении.

То, чему требуется обучать сейчас

- **Параллельное программирование.** Для простых задач будет достаточно простых последовательных алгоритмов, с которыми справятся даже неопытные программисты даже с использованием интерпретируемых языков.
- **Конкуренция.** Процессы операционной системы могут конкурировать с моим за оперативную память, за кэш. Мои процессы могут конкурировать между собой за оперативную память, кэш и операции ввода/вывода. Мои потоки конкурируют со всем, включая ещё и общие объекты в памяти.
- **Выбор алгоритма.** Отличный алгоритм красно-чёрных деревьев и быстрая сортировка будут совершенно отвратительно работать в многопоточных программах. А вот поиск с использованием skiplist и сортировка слиянием в параллельном окружении будут работать хорошо.

То, чему требуется обучать сейчас

- **Справедливость.** Как мои потоки будут влиять на другие мои потоки? Не будет ли излишней конкуренции?
- **Надёжность.** Как не создать себе проблем? Каким образом не попасть в тупик? Как избежать состояния гонок?

- Теория параллельного программирования независимых процессов начала создаваться в 70-х годах 20 века.
- Теория параллельного программирования на реально многопроцессорных системах с общей памятью почти вся создаётся в 21 веке.

Что вы уже знаете и чему должны научиться.

Вы должны уже знать и уметь

- Знать основы операционных систем, процессы, потоки.
- Знать основные проблемы синхронизации и базовые механизмы синхронизации.
- Знать архитектуру компьютера на базовом уровне.
- Знать основные алгоритмы и структуры данных.
- Знать языки Си и C++.
- Уметь реализовывать алгоритмы на каком-либо языке программирования.

Что будет в курсе

- Технические знания
  - Необходимый минимум по архитектуре современных вычислительных систем, необходимый для их более полного использования.
  - Основные концепции параллельного программирования.
- Теоретические знания
  - Новые понятия — консенсус, валентность.
  - Теоремы и их доказательства.
  - Анализ алгоритмов на корректность.
  - Анализ алгоритмов на эффективность в лучшем, худшем и среднем случаях.

Что будет в курсе

- Технические умения
  - Программирование: реализация новых абстракций.
  - Программирование: написание реально работающих программ.
  - Исследование поведения алгоритмов под нагрузкой.
  - Активное исследование программ на корректность.
- Использование всего вместе для создания надёжных эффективных масштабируемых программ.

Содержание курса по темам

- Те основы архитектуры современных ЭВМ, которые влияют на производительность программ.
- Способы использования компонентов аппаратуры в наших целях.
- Система команд и её использование в языках программирования Си и C++.
- Необычные понятия параллельного программирования.
- Уровни абстракции использования аппаратуры.
- Понятие атомарности.
- Понятие времени.
- Корректность исполнения параллельных алгоритмов.
- Особенности взаимодействия параллельных процессов в разделяемой памяти.
- Общие проблемы параллельности.
- Организация синхронного доступа.
- Математика параллельного программирования, математические методы, применяемые для анализа и

Содержание курса по темам

- Введение и формализация таких понятий, как история, завершённость, сериализованность, линеаризованность, легальность.
- Виды прогресса - условный и безусловный.
- Примитивы синхронизации, сравнение их друг с другом, мощность.
- Консенсус, теорема о консенсусе, задача о консенсусе.
- Числа консенсуса для основных примитивов.

Наша цель — свобода

- Свободные алгоритмы.
- Классы свободных алгоритмов lock-free, wait-free и obstruction-free.
- Эффективность свободных алгоритмов.

Наш ограничитель — невозможность

- Невозможность улучшить примитив.
- Невозможность решить задачу о консенсусе с помощью инструкций чтения/записи.
- Невозможность решить задачу о консенсусе для системы со сбоями.
- Невозможность решить не означает нерешаемость. Ethernet.

Неблокирующие алгоритмы

- Когда их применять и зачем.
- Как реализовать неблокирующий алгоритм.
- На базе чего создать новый.

Ожидаемые навыки после прохождения курса

- Уметь анализировать работу параллельных потоков, использующих разделяемую память.
- Понимать терминологию и проблемы параллельного программирования. Уметь использовать математический аппарат, применяемый для описания моделей параллельных алгоритмов, для их анализа.
- Знать современные подходы к организации параллельных вычислений.
- Уметь создавать эффективные параллельные алгоритмы и обосновывать их применение.
- Понимать ограничения параллельных алгоритмов и методов, не пытаться создать вечный двигатель.

Источники


Тормасов А. Г. Параллельное программирование многопоточных систем с разделяемой памятью// М.: Физматкнига, С. 208, 2014


Herlihy M., Shavit N. The art of multiprocessor programming?// Morgan Kaufmann Publishers, 2008


Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд.// СПб.: Питер, 2015. - С. 1120


Карпов В. Е., Коньков К. А. Основы операционных систем. Курс лекций. Учебное пособие. / Под редакцией В. П. Иванникова. -М.: ИНТУИТ.РУ Интернет-Университет информационных технологий, 2004, -С. 1-632


C++ 2011 standard

Дальнейшие исследования:

- Анализ и улучшение существующих неблокирующих алгоритмов
- Автоматизация анализа алгоритмов на корректность: отсутствие race conditions и deadlocks.

Спасибо за внимание.

Следующая тема — архитектурные особенности современных компьютеров. Память.